imgaug Documentation

Release 0.3.0

Alexander Jung

Sep 24, 2019

Contents:

imgaug is a library for image augmentation in machine learning experiments. It supports a wide range of augmentation techniques, allows to easily combine these and to execute them in random order or on multiple CPU cores, has a simple yet powerful stochastic interface and can not only augment images, but also keypoints/landmarks, bounding boxes, heatmaps and segmentation maps.

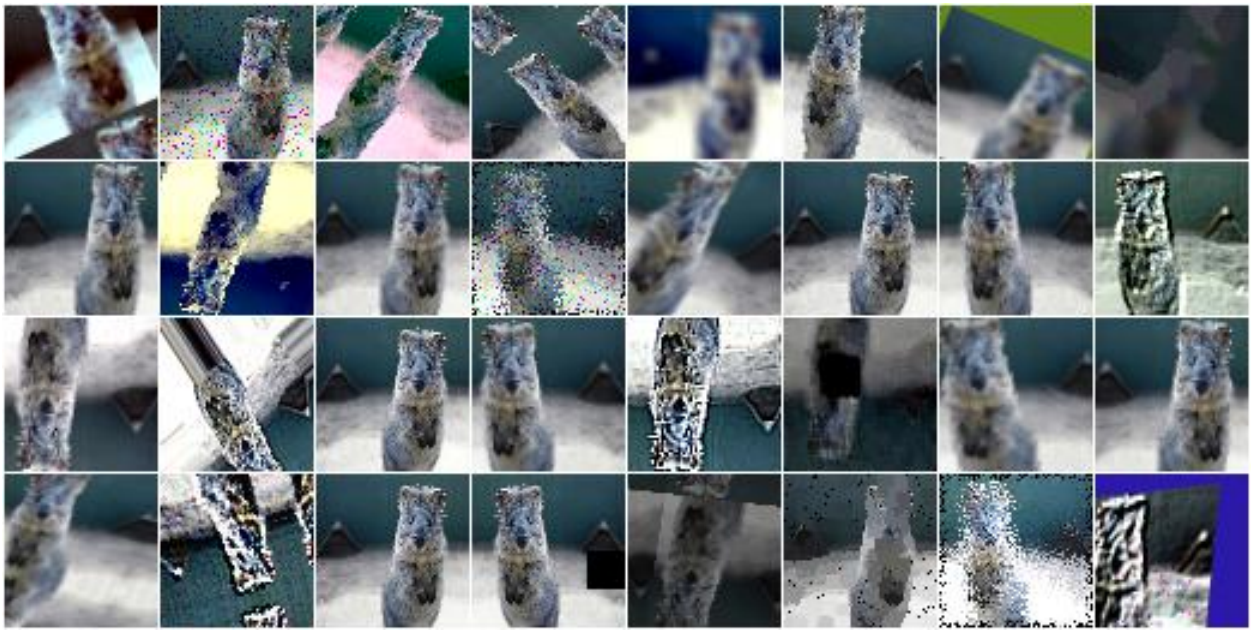


Fig. 1: Example augmentations of a single input image.

CHAPTER 1

Installation

The library uses python, which must be installed. Python 2.7, 3.4, 3.5, 3.6 and 3.7 are supported.

The below sections explain how to install the library in anaconda or via pip. If you don't know what anaconda (or conda) are, simply use pip instead as that should always work.

1.1 Installation in Anaconda

To install in anaconda simply perform the following commands

```
conda config --add channels conda-forge
conda install imgaug
```

Note that you may also use the pip-based installation commands described below. They work with and without anaconda.

1.2 Installation in pip

1.2.1 Install Requirements

To install all requirements, use

```
pip install six numpy scipy Pillow matplotlib scikit-image opencv-python imageio␣
↪Shapely
```

Note that if you already have OpenCV, you might not need `opencv-python`. If you get any "permission denied" errors, try adding `sudo` in front of the command. If your encounter issues installing `Shapely` you can skip that library. It is only imported when actually needed. At least polygon and line string augmentation will likely crash without it.

1.2.2 Install Library

Once the required packages are available, `imgaug` can be installed using the following command:

```
pip install imgaug
```

This installs the latest version from pypi, which often lags behind the latest version on github by a few months. To instead get the very latest version use

```
pip install git+https://github.com/aleju/imgaug
```

1.3 Installation from Source

In rare cases, one might prefer to install directly from source. This is possible using

```
git clone https://github.com/aleju/imgaug.git && cd imgaug && python setup.py install
```

Note that this is effectively identical to using `pip install <github link>`.

1.4 Uninstall

To deinstall the library use

```
conda remove imgaug
```

on anaconda and

```
pip uninstall imgaug
```

otherwise.

CHAPTER 2

Examples: Basics

2.1 A standard use case

The following example shows a standard use case. An augmentation sequence (crop + horizontal flips + gaussian blur) is defined once at the start of the script. Then many batches are loaded and augmented before being used for training.

```
from imgaug import augmenters as iaa

seq = iaa.Sequential([
    iaa.Crop(px=(0, 16)), # crop images from each side by 0 to 16px (randomly chosen)
    iaa.Fliplr(0.5), # horizontally flip 50% of the images
    iaa.GaussianBlur(sigma=(0, 3.0)) # blur images with a sigma of 0 to 3.0
])

for batch_idx in range(1000):
    # 'images' should be either a 4D numpy array of shape (N, height, width, channels)
    # or a list of 3D numpy arrays, each having shape (height, width, channels).
    # Grayscale images must have shape (height, width, 1) each.
    # All images must have numpy's dtype uint8. Values are expected to be in
    # range 0-255.
    images = load_batch(batch_idx)
    images_aug = seq(images=images)
    train_on_images(images_aug)
```

2.2 A simple and common augmentation sequence

The following example shows an augmentation sequence that might be useful for many common experiments. It applies crops and affine transformations to images, flips some of the images horizontally, adds a bit of noise and blur and also changes the contrast as well as brightness.

```
import numpy as np
import imgaug as ia
import imgaug.augmenters as iaa


ia.seed(1)

# Example batch of images.
# The array has shape (32, 64, 64, 3) and dtype uint8.
images = np.array(
    [ia.quokka(size=(64, 64)) for _ in range(32)],
    dtype=np.uint8
)

seq = iaa.Sequential([
    iaa.Fliplr(0.5), # horizontal flips
    iaa.Crop(percent=(0, 0.1)), # random crops
    # Small gaussian blur with random sigma between 0 and 0.5.
    # But we only blur about 50% of all images.
    iaa.Sometimes(0.5,
        iaa.GaussianBlur(sigma=(0, 0.5))
    ),
    # Strengthen or weaken the contrast in each image.
    iaa.ContrastNormalization((0.75, 1.5)),
    # Add gaussian noise.
    # For 50% of all images, we sample the noise once per pixel.
    # For the other 50% of all images, we sample the noise per pixel AND
    # channel. This can change the color (not only brightness) of the
    # pixels.
    iaa.AdditiveGaussianNoise(loc=0, scale=(0.0, 0.05*255), per_channel=0.5),
    # Make some images brighter and some darker.
    # In 20% of all cases, we sample the multiplier once per channel,
    # which can end up changing the color of the images.
    iaa.Multiply((0.8, 1.2), per_channel=0.2),
    # Apply affine transformations to each image.
    # Scale/zoom them, translate/move them, rotate them and shear them.
    iaa.Affine(
        scale={"x": (0.8, 1.2), "y": (0.8, 1.2)},
        translate_percent={"x": (-0.2, 0.2), "y": (-0.2, 0.2)},
        rotate=(-25, 25),
        shear=(-8, 8)
    )
], random_order=True) # apply augmenters in random order

images_aug = seq(images=images)
```

2.3 Heavy Augmentations

The following example shows a large augmentation sequence containing many different augmenters, leading to significant changes in the augmented images. Depending on the use case, the sequence might be too strong. Occasionally it can also break images by changing them too much. To weaken the effects you can lower the value of `iaa.SomeOf((0, 5), ...)` to e.g. `(0, 3)` or decrease the probability of some augmenters to be applied by decreasing in `sometimes = lambda aug: iaa.Sometimes(0.5, aug)` the value `0.5` to e.g. `0.3`.

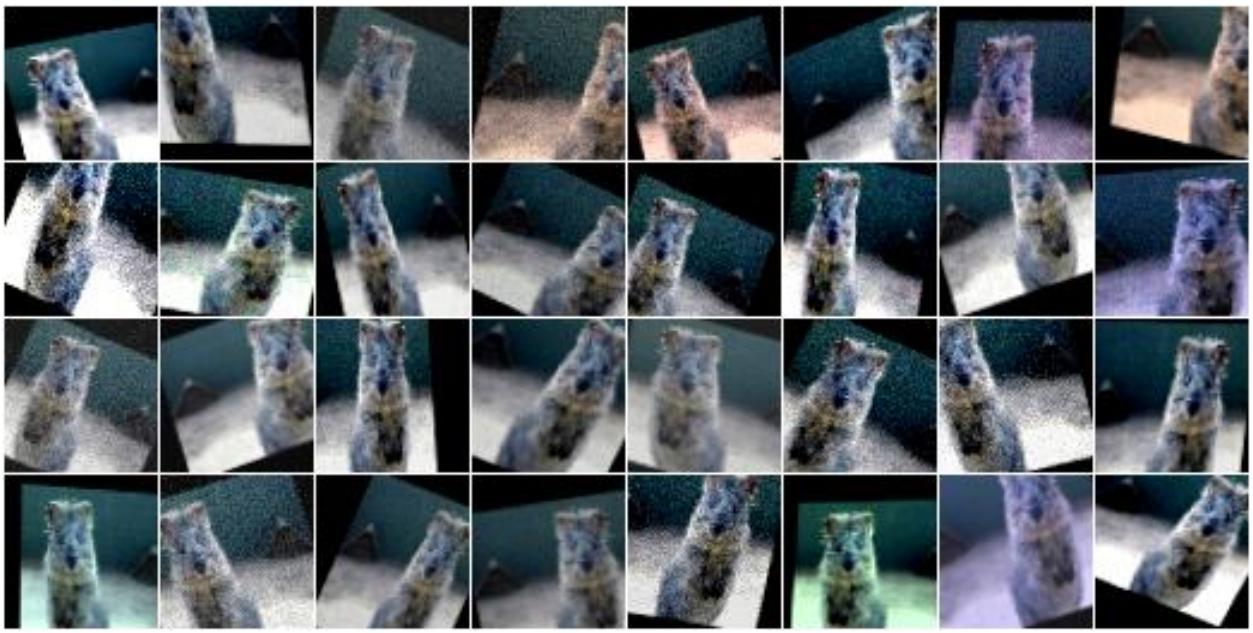


Fig. 1: Example results of the above simple augmentation sequence.

```
import numpy as np
import imgaug as ia
import imgaug.augmenters as iaa


ia.seed(1)

# Example batch of images.
# The array has shape (32, 64, 64, 3) and dtype uint8.
images = np.array(
    [ia.quokka(size=(64, 64)) for _ in range(32)],
    dtype=np.uint8
)

# Sometimes(0.5, ...) applies the given augmenter in 50% of all cases,
# e.g. Sometimes(0.5, GaussianBlur(0.3)) would blur roughly every second
# image.
sometimes = lambda aug: iaa.Sometimes(0.5, aug)

# Define our sequence of augmentation steps that will be applied to every image.
seq = iaa.Sequential(
    [
        #
        # Apply the following augmenters to most images.
        #
        iaa.Fliplr(0.5), # horizontally flip 50% of all images
        iaa.Flipud(0.2), # vertically flip 20% of all images

        # crop some of the images by 0-10% of their height/width
        sometimes(iaa.Crop(percent=(0, 0.1))),

        # Apply affine transformations to some of the images
```

(continues on next page)

(continued from previous page)

```
        # - scale to 80-120% of image height/width (each axis independently)
        # - translate by -20 to +20 relative to height/width (per axis)
        # - rotate by -45 to +45 degrees
        # - shear by -16 to +16 degrees
        # - order: use nearest neighbour or bilinear interpolation (fast)
        # - mode: use any available mode to fill newly created pixels
        #         see API or scikit-image for which modes are available
        # - cval: if the mode is constant, then use a random brightness
        #         for the newly created pixels (e.g. sometimes black,
        #         sometimes white)
        sometimes(iaa.Affine(
            scale={"x": (0.8, 1.2), "y": (0.8, 1.2)},
            translate_percent={"x": (-0.2, 0.2), "y": (-0.2, 0.2)},
            rotate=(-45, 45),
            shear=(-16, 16),
            order=[0, 1],
            cval=(0, 255),
            mode=ia.ALL
        )),

        #
        # Execute 0 to 5 of the following (less important) augmenters per
        # image. Don't execute all of them, as that would often be way too
        # strong.
        #
        iaa.SomeOf((0, 5),
            [
                # Convert some images into their superpixel representation,
                # sample between 20 and 200 superpixels per image, but do
                # not replace all superpixels with their average, only
                # some of them (p_replace).
                sometimes(
                    iaa.Superpixels(
                        p_replace=(0, 1.0),
                        n_segments=(20, 200)
                    )
                ),

                # Blur each image with varying strength using
                # gaussian blur (sigma between 0 and 3.0),
                # average/uniform blur (kernel size between 2x2 and 7x7)
                # median blur (kernel size between 3x3 and 11x11).
                iaa.OneOf([
                    iaa.GaussianBlur((0, 3.0)),
                    iaa.AverageBlur(k=(2, 7)),
                    iaa.MedianBlur(k=(3, 11)),
                ]),

                # Sharpen each image, overlay the result with the original
                # image using an alpha between 0 (no sharpening) and 1
                # (full sharpening effect).
                iaa.Sharpen(alpha=(0, 1.0), lightness=(0.75, 1.5)),

                # Same as sharpen, but for an embossing effect.
                iaa.Emboss(alpha=(0, 1.0), strength=(0, 2.0)),

                # Search in some images either for all edges or for
```

(continues on next page)

(continued from previous page)

```
            # directed edges. These edges are then marked in a black
            # and white image and overlayed with the original image
            # using an alpha of 0 to 0.7.
            sometimes(iaa.OneOf([
                iaa.EdgeDetect(alpha=(0, 0.7)),
                iaa.DirectedEdgeDetect(
                    alpha=(0, 0.7), direction=(0.0, 1.0)
                ),
            ])),

            # Add gaussian noise to some images.
            # In 50% of these cases, the noise is randomly sampled per
            # channel and pixel.
            # In the other 50% of all cases it is sampled once per
            # pixel (i.e. brightness change).
            iaa.AdditiveGaussianNoise(
                loc=0, scale=(0.0, 0.05*255), per_channel=0.5
            ),

            # Either drop randomly 1 to 10% of all pixels (i.e. set
            # them to black) or drop them on an image with 2-5% percent
            # of the original size, leading to large dropped
            # rectangles.
            iaa.OneOf([
                iaa.Dropout((0.01, 0.1), per_channel=0.5),
                iaa.CoarseDropout(
                    (0.03, 0.15), size_percent=(0.02, 0.05),
                    per_channel=0.2
                ),
            ]),

            # Invert each image's channel with 5% probability.
            # This sets each pixel value v to 255-v.
            iaa.Invert(0.05, per_channel=True), # invert color channels

            # Add a value of -10 to 10 to each pixel.
            iaa.Add((-10, 10), per_channel=0.5),

            # Change brightness of images (50-150% of original value).
            iaa.Multiply((0.5, 1.5), per_channel=0.5),

            # Improve or worsen the contrast of images.
            iaa.ContrastNormalization((0.5, 2.0), per_channel=0.5),

            # Convert each image to grayscale and then overlay the
            # result with the original with random alpha. I.e. remove
            # colors with varying strengths.
            iaa.Grayscale(alpha=(0.0, 1.0)),

            # In some images move pixels locally around (with random
            # strengths).
            sometimes(
                iaa.ElasticTransformation(alpha=(0.5, 3.5), sigma=0.25)
            ),

            # In some images distort local areas with varying strength.
            sometimes(iaa.PiecewiseAffine(scale=(0.01, 0.05)))
```

(continues on next page)

(continued from previous page)

```
            ],
            # do all of the above augmentations in random order
            random_order=True
        )
    ],
    # do all of the above augmentations in random order
    random_order=True
)

images_aug = seq(images=images)
```

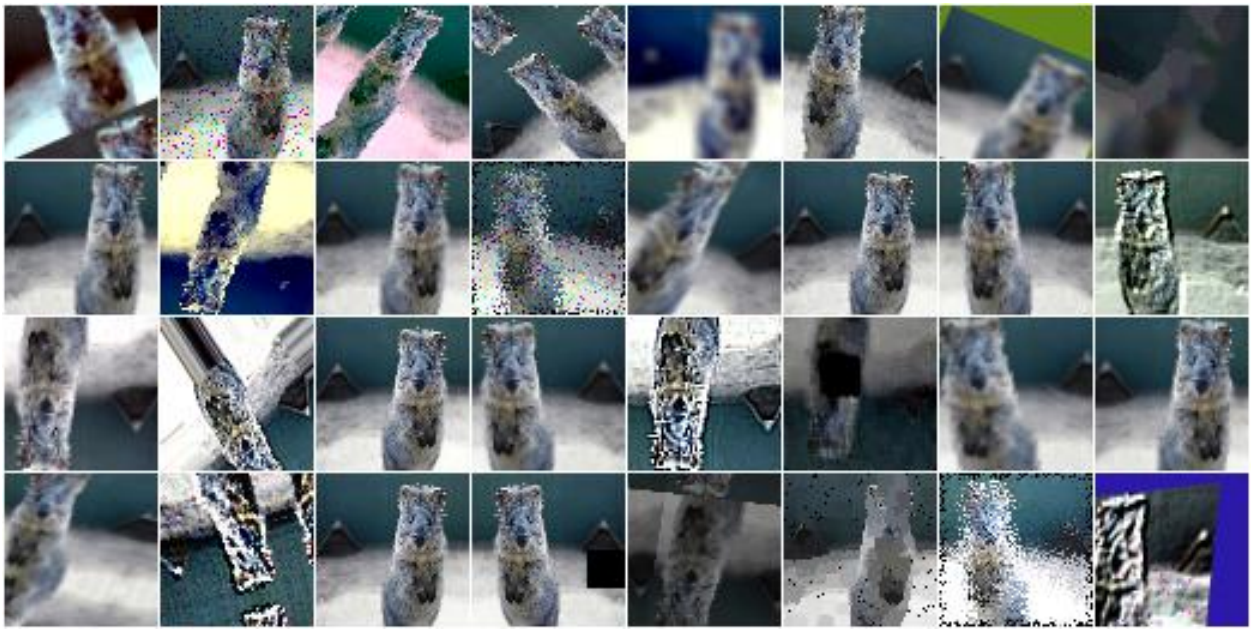


Fig. 2: Example results of the above heavy augmentation sequence.

CHAPTER 3

Examples: Keypoints

imgaug can handle not only images, but also keypoints/landmarks on these. E.g. if an image is rotated during augmentation, the library can also rotate all landmarks correspondingly.

3.1 Notebook

A jupyter notebook for keypoint augmentation is available at *Jupyter Notebooks*. The notebooks are usually more up to date and contain more examples than the ReadTheDocs documentation.

3.2 A simple example

The following example loads an image and places four keypoints on it. The image is then augmented to be brighter, slightly rotated and scaled. These augmentations are also applied to the keypoints. The image is then shown before and after augmentation (with keypoints drawn on it).

```
import imgaug as ia
import imgaug.augmenters as iaa
from imgaug.augmentables import Keypoint, KeypointsOnImage


ia.seed(1)

image = ia.quokka(size=(256, 256))
kps = KeypointsOnImage([
    Keypoint(x=65, y=100),
    Keypoint(x=75, y=200),
    Keypoint(x=100, y=100),
    Keypoint(x=200, y=80)
], shape=image.shape)

seq = iaa.Sequential([
```

(continues on next page)

(continued from previous page)

```
    iaa.Multiply((1.2, 1.5)), # change brightness, doesn't affect keypoints
    iaa.Affine(
        rotate=10,
        scale=(0.5, 0.7)
    ) # rotate by exactly 10deg and scale to 50-70%, affects keypoints
])

# Augment keypoints and images.
image_aug, kps_aug = seq(image=image, keypoints=kps)

# print coordinates before/after augmentation (see below)
# use after.x_int and after.y_int to get rounded integer coordinates
for i in range(len(kps.keypoints)):
    before = kps.keypoints[i]
    after = kps_aug.keypoints[i]
    print("Keypoint %d: (%.8f, %.8f) -> (%.8f, %.8f)" % (
        i, before.x, before.y, after.x, after.y)
    )

# image with keypoints before/after augmentation (shown below)
image_before = kps.draw_on_image(image, size=7)
image_after = kps_aug.draw_on_image(image_aug, size=7)
```

Console output of the example:

```
Keypoint 0: (65.00000000, 100.00000000) -> (97.86113503, 107.69632182)
Keypoint 1: (75.00000000, 200.00000000) -> (93.93710117, 160.01366917)
Keypoint 2: (100.00000000, 100.00000000) -> (115.85492750, 110.86911292)
Keypoint 3: (200.00000000, 80.00000000) -> (169.07878659, 109.65206321)
```

Fig. 1: Image with keypoints, before (left) and after (right) augmentation. Keypoints are shown in green and drawn in after the augmentation process.

CHAPTER 4

Examples: Bounding Boxes

imgaug offers support for bounding boxes (aka rectangles, regions of interest). E.g. if an image is rotated during augmentation, the library can also rotate all bounding boxes on it correspondingly.

Features of the library's bounding box support:

- Represent bounding boxes as objects (*imgaug.augmentables.bbs.BoundingBox*).
- Augment bounding boxes.
- Draw bounding boxes on images.
- Move/shift bounding boxes on images, project them onto other images (e.g. onto the same image after resizing), compute their intersections/unions and IoU values.

4.1 Notebook

A jupyter notebook for bounding box augmentation is available at *Jupyter Notebooks*. The notebooks are usually more up to date and contain more examples than the ReadTheDocs documentation.

4.2 A simple example

The following example loads an image and places two bounding boxes on it. The image is then augmented to be brighter, slightly rotated and scaled. These augmentations are also applied to the bounding boxes. The image is then shown before and after augmentation (with bounding boxes drawn on it).

```
import imgaug as ia
import imgaug.augmenters as iaa
from imgaug.augmentables.bbs import BoundingBox, BoundingBoxesOnImage


ia.seed(1)
```

(continues on next page)

(continued from previous page)

```
image = ia.quokka(size=(256, 256))
bbs = BoundingBoxesOnImage([
    BoundingBox(x1=65, y1=100, x2=200, y2=150),
    BoundingBox(x1=150, y1=80, x2=200, y2=130)
], shape=image.shape)

seq = iaa.Sequential([
    iaa.Multiply((1.2, 1.5)), # change brightness, doesn't affect BBs
    iaa.Affine(
        translate_px={"x": 40, "y": 60},
        scale=(0.5, 0.7)
    ) # translate by 40/60px on x/y axis, and scale to 50-70%, affects BBs
])

# Augment BBs and images.
image_aug, bbs_aug = seq(image=image, bounding_boxes=bbs)

# print coordinates before/after augmentation (see below)
# use .x1_int, .y_int, ... to get integer coordinates
for i in range(len(bbs.bounding_boxes)):
    before = bbs.bounding_boxes[i]
    after = bbs_aug.bounding_boxes[i]
    print("BB %d: (%.4f, %.4f, %.4f, %.4f) -> (%.4f, %.4f, %.4f, %.4f)" % (
        i,
        before.x1, before.y1, before.x2, before.y2,
        after.x1, after.y1, after.x2, after.y2)
    )

# image with BBs before/after augmentation (shown below)
image_before = bbs.draw_on_image(image, size=2)
image_after = bbs_aug.draw_on_image(image_aug, size=2, color=[0, 0, 255])
```

Console output of the example:

```
BB 0: (65.0000, 100.0000, 200.0000, 150.0000) -> (130.7524, 171.3311, 210.1272, 200.
↪7291)
BB 1: (150.0000, 80.0000, 200.0000, 130.0000) -> (180.7291, 159.5718, 210.1272, 188.
↪9699)
```

Note that the bounding box augmentation works by augmenting each box's edge coordinates and then drawing a bounding box around these augmented coordinates. Each of these new bounding boxes is therefore axis-aligned. This can sometimes lead to oversized new bounding boxes, especially in the case of rotation. The following image shows the result of the same code as in the example above, but *Affine* was replaced by *Affine(rotate=45)*:

4.3 Dealing with bounding boxes outside of the image

When augmenting images and their respective bounding boxes, the boxes can end up fully or partially outside of the image plane. By default, the library still returns these boxes, even though that may not be desired. The following example shows how to (a) remove bounding boxes that are fully/partially outside of the image and (b) how to clip bounding boxes that are partially outside of the image so that their are fully inside.

```
import numpy as np
import imgaug as ia
```

(continues on next page)

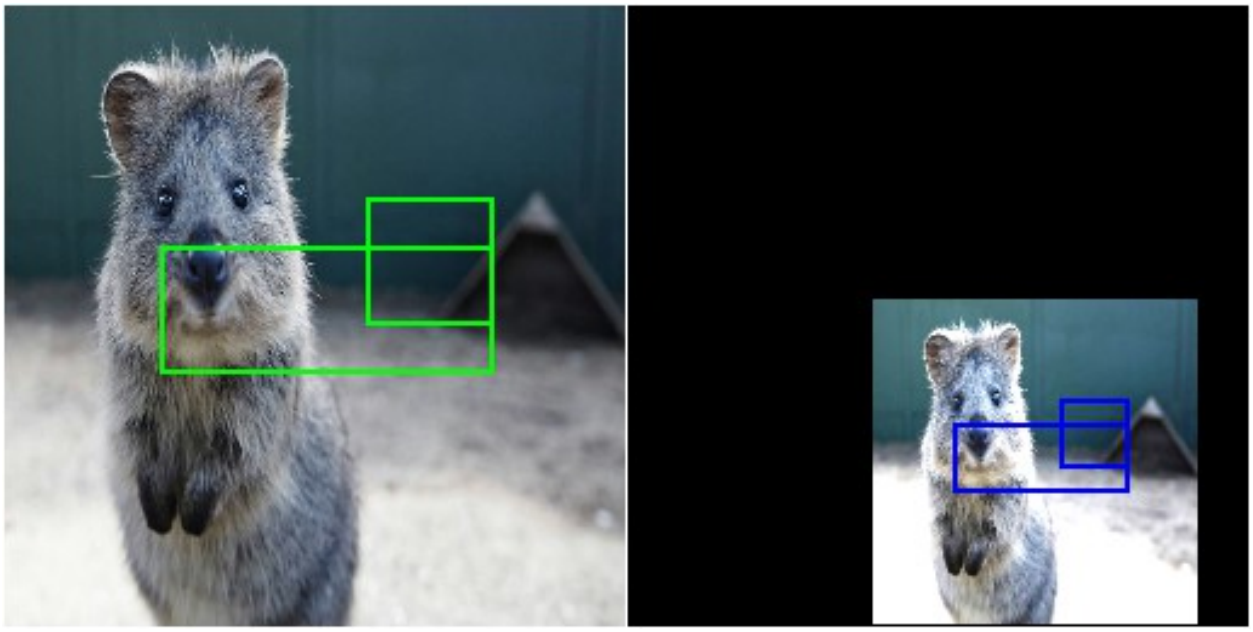


Fig. 1: Image with bounding boxes, before (left) and after (right) augmentation. Bounding boxes are shown in green (before augmentation) and blue (after augmentation).

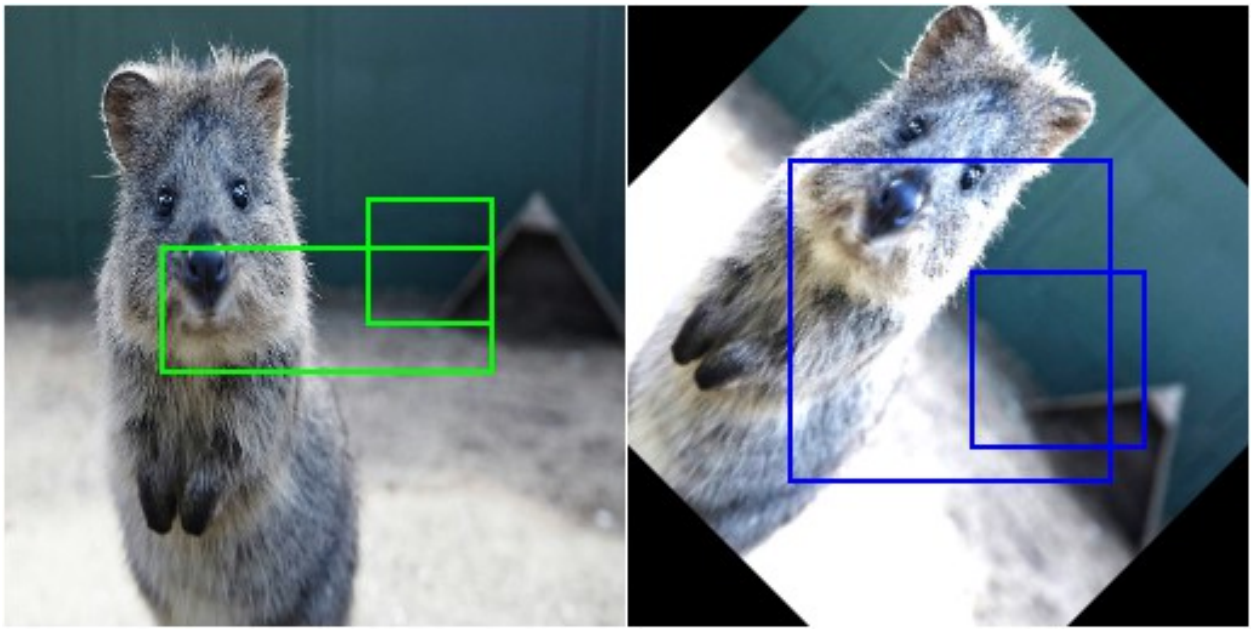


Fig. 2: Image with bounding boxes, before (left) and after (right) augmentation. The image was augmentated by rotating it by 45 degrees. The axis-aligned bounding box around the augmented keypoints ends up being oversized.

(continued from previous page)

```
import imgaug.augmenters as iaa
from imgaug.augmentables.bbs import BoundingBox, BoundingBoxesOnImage


ia.seed(1)

GREEN = [0, 255, 0]
ORANGE = [255, 140, 0]
RED = [255, 0, 0]

# Pad image with a 1px white and (BY-1)px black border
def pad(image, by):
    image_border1 = ia.pad(image, top=1, right=1, bottom=1, left=1,
                           mode="constant", cval=255)
    image_border2 = ia.pad(image_border1, top=by-1, right=by-1,
                           bottom=by-1, left=by-1,
                           mode="constant", cval=0)
    return image_border2

# Draw BBs on an image
# and before doing that, extend the image plane by BORDER pixels.
# Mark BBs inside the image plane with green color, those partially inside
# with orange and those fully outside with red.
def draw_bbs(image, bbs, border):
    image_border = pad(image, border)
    for bb in bbs.bounding_boxes:
        if bb.is_fully_within_image(image.shape):
            color = GREEN
        elif bb.is_partly_within_image(image.shape):
            color = ORANGE
        else:
            color = RED
        image_border = bb.shift(left=border, top=border)\
                         .draw_on_image(image_border, size=2, color=color)

    return image_border

# Define example image with three small square BBs next to each other.
# Augment these BBs by shifting them to the right.
image = ia.quokka(size=(256, 256))
bbs = BoundingBoxesOnImage([
    BoundingBox(x1=25, x2=75, y1=25, y2=75),
    BoundingBox(x1=100, x2=150, y1=25, y2=75),
    BoundingBox(x1=175, x2=225, y1=25, y2=75)
], shape=image.shape)

seq = iaa.Affine(translate_px={"x": 120})
image_aug, bbs_aug = seq(image=image, bounding_boxes=bbs)

# Draw the BBs (a) in their original form, (b) after augmentation,
# (c) after augmentation and removing those fully outside the image,
# (d) after augmentation and removing those fully outside the image and
# clipping those partially inside the image so that they are fully inside.
image_before = draw_bbs(image, bbs, 100)
image_after1 = draw_bbs(image_aug, bbs_aug, 100)
image_after2 = draw_bbs(image_aug, bbs_aug.remove_out_of_image(), 100)
image_after3 = draw_bbs(image_aug, bbs_aug.remove_out_of_image().clip_out_of_image(),␣
↪100)
```

(continues on next page)

(continued from previous page)

4.4 Shifting/Moving Bounding Boxes

The function *shift(top=<int>, right=<int>, bottom=<int>, left=<int>)* can be used to change the x/y position of all or specific bounding boxes.

```
import imgaug as ia
from imgaug.augmentables.bbs import BoundingBox, BoundingBoxesOnImage


ia.seed(1)

# Define image and two bounding boxes
image = ia.quokka(size=(256, 256))
bbs = BoundingBoxesOnImage([
    BoundingBox(x1=25, x2=75, y1=25, y2=75),
    BoundingBox(x1=100, x2=150, y1=25, y2=75)
], shape=image.shape)

# Move both BBs 25px to the right and the second BB 25px down
bbs_shifted = bbs.shift(left=25)
bbs_shifted.bounding_boxes[1] = bbs_shifted.bounding_boxes[1].shift(top=25)

# Draw images before/after moving BBs
image = bbs.draw_on_image(image, color=[0, 255, 0], size=2, alpha=0.75)
image = bbs_shifted.draw_on_image(image, color=[0, 0, 255], size=2, alpha=0.75)
```

4.5 Projection of BBs Onto Rescaled Images

Bounding boxes can easily be projected onto rescaled versions of the same image using the function *.on(image)*. This changes the coordinates of the bounding boxes. E.g. if the top left coordinate of the bounding box was before at x=10% and y=15%, it will still be at x/y 10%/15% on the new image, though the absolute pixel values will change depending on the height/width of the new image.

```
import imgaug as ia
from imgaug.augmentables.bbs import BoundingBox, BoundingBoxesOnImage


ia.seed(1)

# Define image with two bounding boxes
image = ia.quokka(size=(256, 256))
bbs = BoundingBoxesOnImage([
    BoundingBox(x1=25, x2=75, y1=25, y2=75),
    BoundingBox(x1=100, x2=150, y1=25, y2=75)
], shape=image.shape)

# Rescale image and bounding boxes
image_rescaled = ia.imresize_single_image(image, (512, 512))
bbs_rescaled = bbs.on(image_rescaled)
```

(continues on next page)

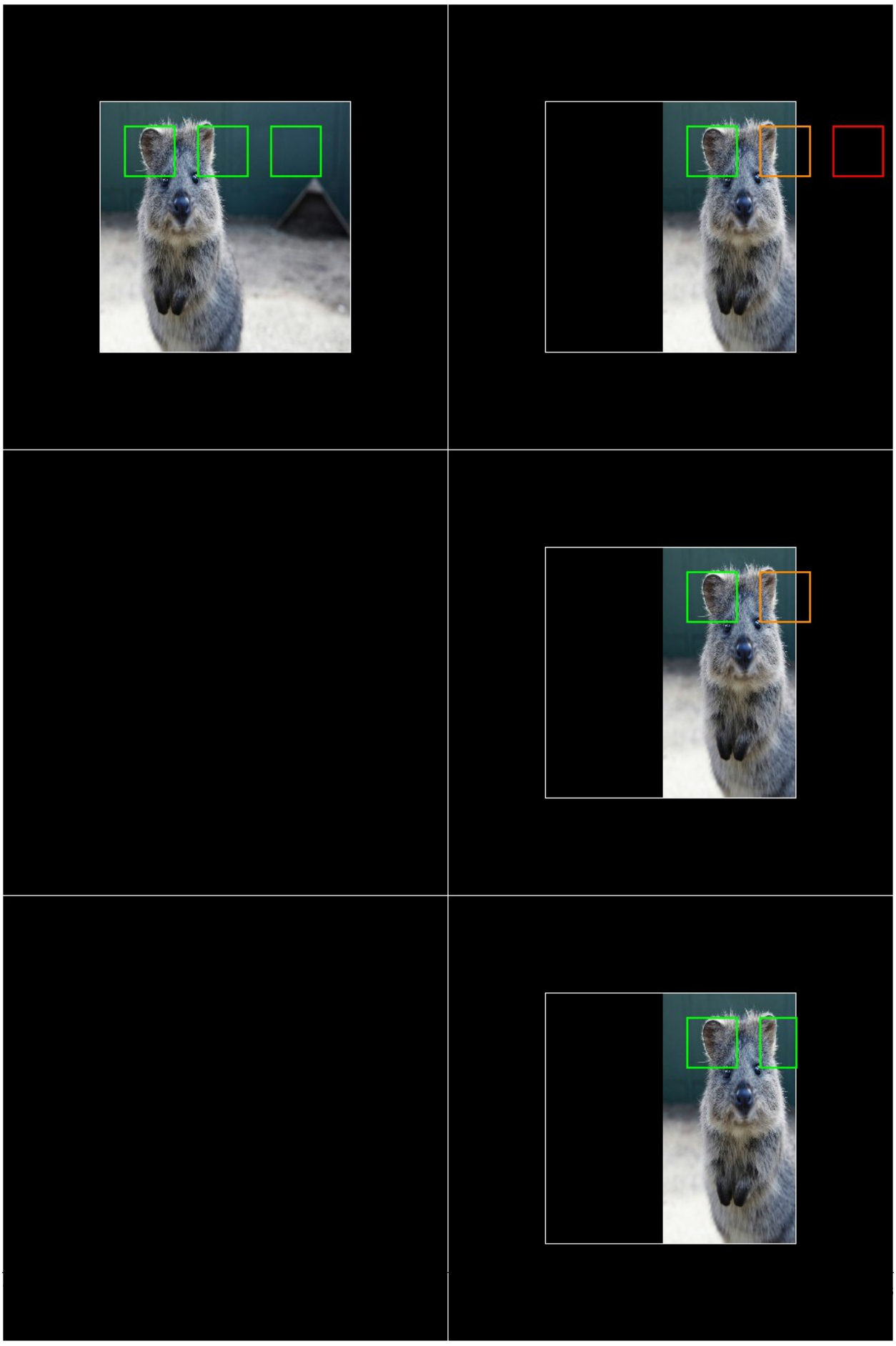


Fig. 3: Results of the above example code. Top left: Original/unaugmented image with bounding boxes (here visual

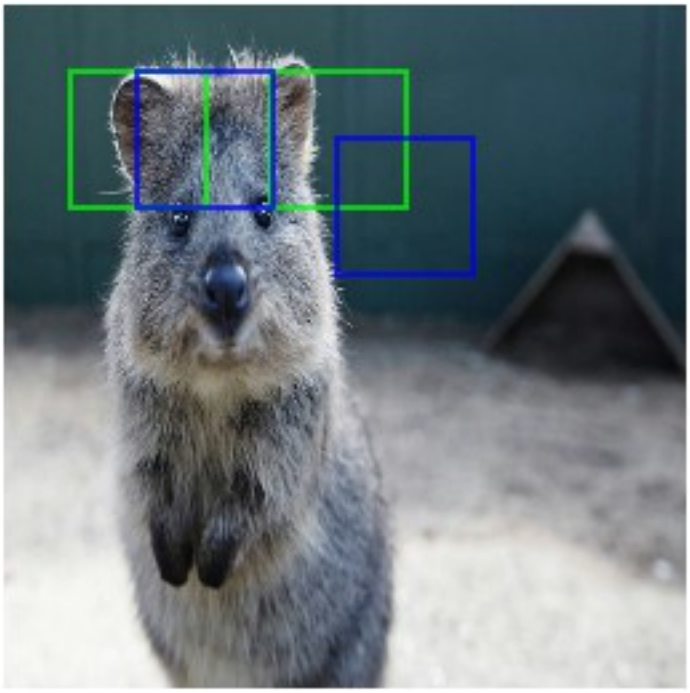


Fig. 4: Using *shift()* to move bounding boxes around (green: original BBs, blue: shifted/moved BBs).

(continued from previous page)

```
# Draw image before/after rescaling and with rescaled bounding boxes
image_bbs = bbs.draw_on_image(image, size=2)
image_rescaled_bbs = bbs_rescaled.draw_on_image(image_rescaled, size=2)
```

4.6 Computing Intersections, Unions and IoUs

Computing intersections, unions and especially IoU values (intersection over union) is common for many machine learning experiments. The library offers easy functions for that.

```
import numpy as np
import imgaug as ia
from imgaug.augmentables.bbs import BoundingBox


ia.seed(1)

# Define image with two bounding boxes.
image = ia.quokka(size=(256, 256))
bb1 = BoundingBox(x1=50, x2=100, y1=25, y2=75)
bb2 = BoundingBox(x1=75, x2=125, y1=50, y2=100)

# Compute intersection, union and IoU value
# Intersection and union are both bounding boxes. They are here
# decreased/increased in size purely for better visualization.
bb_inters = bb1.intersection(bb2).extend(all_sides=-1)
```

(continues on next page)

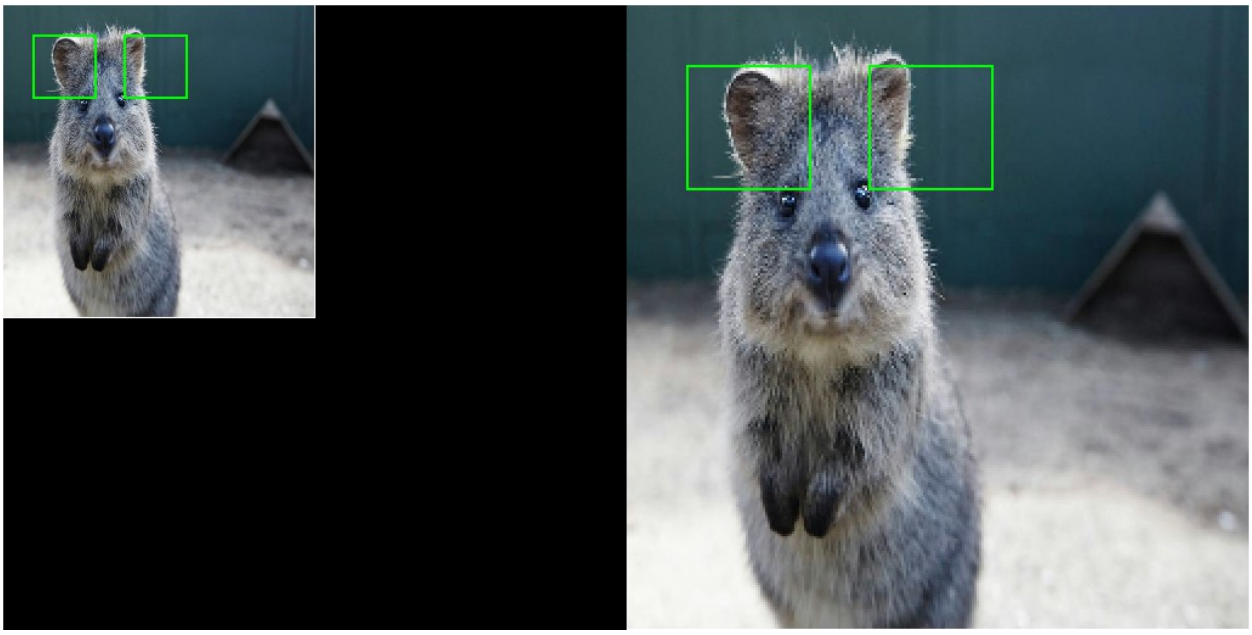


Fig. 5: Using *on()* to project bounding boxes from one image to the other, here onto an image of 2x the original size. New coordinates are determined based on their relative positions on the old image.

(continued from previous page)

```
bb_union = bb1.union(bb2).extend(all_sides=2)
iou = bb1.iou(bb2)

# Draw bounding boxes, intersection, union and IoU value on image.
image_bbs = np.copy(image)
image_bbs = bb1.draw_on_image(image_bbs, size=2, color=[0, 255, 0])
image_bbs = bb2.draw_on_image(image_bbs, size=2, color=[0, 255, 0])
image_bbs = bb_inters.draw_on_image(image_bbs, size=2, color=[255, 0, 0])
image_bbs = bb_union.draw_on_image(image_bbs, size=2, color=[0, 0, 255])
image_bbs = ia.draw_text(
    image_bbs, text="IoU=%.2f" % (iou,),
    x=bb_union.x2+10, y=bb_union.y1+bb_union.height//2,
    color=[255, 255, 255], size=13
)
```

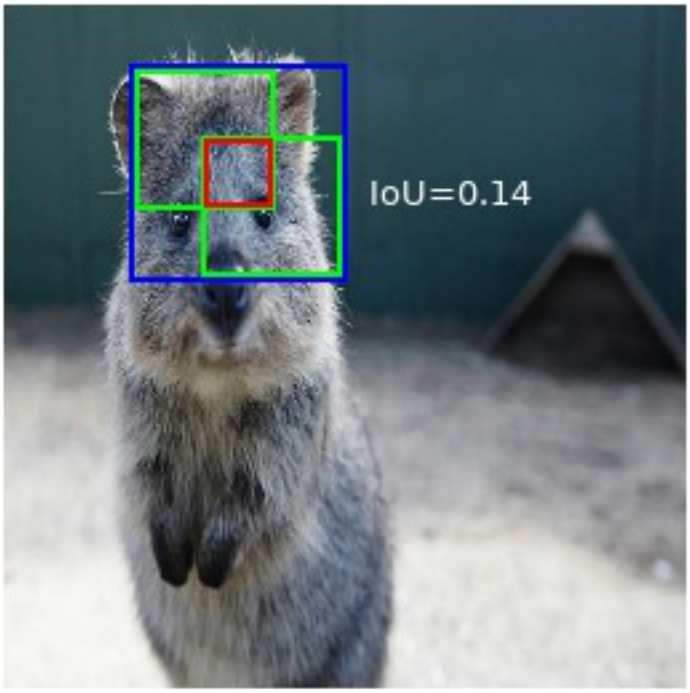


Fig. 6: Two bounding boxes on an image (green), their intersection (red, slightly shrunk), their union (blue, slightly extended) and their IoU value (white).

CHAPTER 5

Examples: Heatmaps

`imgaug` offers support for heatmap-like data. This can be used e.g. for depth map or keypoint/landmark localization maps. Heatmaps can be augmented correspondingly to images, e.g. if an image is rotated by 45°, the corresponding heatmap for that image will also be rotated by 45°.

Note:

- Heatmaps have to be bounded within value ranges, e.g. `0.0` to `1.0` for keypoint localization maps or something like `0.0` to `200.0` (meters) for depth maps. Choosing arbitrarily low/high min/max values for unbounded heatmaps is not recommended as it could lead to numerical inaccuracies.
- All augmentation functions for heatmaps are implemented under the assumption of augmenting **ground truth** data. As such, heatmaps will be affected by augmentations that change the geometry of images (e.g. affine transformations, cropping, resizing), but not by other augmentations (e.g. gaussian noise, saturation changes, grayscaling, dropout, ...).

Features of the library's heatmap support:

- Represent heatmaps as objects (`imgaug.augmentables.heatmaps.HeatmapsOnImage`).
- Augment heatmaps (only geometry-affecting augmentations, e.g. affine transformations, cropping, ...).
- Use different resolutions for heatmaps than for images (e.g. `32x32` heatmaps for `256x256` images).
- Draw heatmaps – on their own or on images (`HeatmapsOnImage.draw()`, `HeatmapsOnImage.draw_on_image()`).
- Resize, average pool or max pool heatmaps (`HeatmapsOnImage.scale()`, `HeatmapsOnImage.avg_pool()`, `HeatmapsOnImage.max_pool()`).
- Pad heatmaps by pixel amounts or to desired aspect ratios (`HeatmapsOnImage.pad()`, `HeatmapsOnImage.pad_to_aspect_ratio()`).

5.1 Notebook

A jupyter notebook for heatmap augmentation is available at *Jupyter Notebooks*. The notebooks are usually more up to date and contain more examples than the ReadTheDocs documentation.

5.2 A simple example

The following example loads a standard image and a generates a corresponding heatmap. The heatmap is supposed to be a depth map, i.e. is supposed to resemble the depth of objects in the image, where higher values indicate that objects are further away. (For simplicity we just use a simple gradient as a depth map with a cross in the center, so there is no real correspondence between the image and the depth values.)

This example shows:

- Creating heatmaps via `HeatmapsOnImage(heatmap_array, shape=image_shape)`.
- Using value ranges outside of simple `0.0` to `1.0` (here `0.0` to `50.0`) by setting `min_value` and `max_value` in the `HeatmapsOnImage` contructor.
- Resizing heatmaps, here via `HeatmapsOnImage.avg_pool(kernel_size)` (i.e. average pooling).
- Augmenting heatmaps via `Augmenter.__call__()`, which is equivalent to `Augmenter.augment()`.
- Drawing heatmaps as overlays over images `HeatmapsOnImage.draw_on_image(image)`.
- Drawing heatmaps on their own via `HeatmapsOnImage.draw()` in jet color map or via `HeatmapsOnImage.draw(cmap=None)` as intensity maps.

```
import imageio
import numpy as np
import imgaug as ia
import imgaug.augmenters as iaa
from imgaug.augmentables.heatmaps import HeatmapsOnImage

ia.seed(1)

# Load an example image (uint8, 128x128x3).
image = ia.quokka(size=(128, 128), extract="square")

# Create an example depth map (float32, 128x128).
# Here, we use a simple gradient that has low values (around 0.0)
# towards the left of the image and high values (around 50.0)
# towards the right. This is obviously a very unrealistic depth
# map, but makes the example easier.
depth = np.linspace(0, 50, 128).astype(np.float32)  # 128 values from 0.0 to 50.0
depth = np.tile(depth.reshape(1, 128), (128, 1))    # change to a horizontal gradient

# We add a cross to the center of the depth map, so that we can more
# easily see the effects of augmentations.
depth[64-2:64+2, 16:128-16] = 0.75 * 50.0  # line from left to right
depth[16:128-16, 64-2:64+2] = 1.0 * 50.0   # line from top to bottom

# Convert our numpy array depth map to a heatmap object.
# We have to add the shape of the underlying image, as that is necessary
# for some augmentations.
depth = HeatmapsOnImage(
    depth, shape=image.shape, min_value=0.0, max_value=50.0)

# To save some computation time, we want our models to perform downscaling
# and hence need the ground truth depth maps to be at a resolution of
# 64x64 instead of the 128x128 of the input image.
# Here, we use simple average pooling to perform the downscaling.
depth = depth.avg_pool(2)
```

(continues on next page)

(continued from previous page)

```
# Define our augmentation pipeline.
seq = iaa.Sequential([
    iaa.Dropout([0.05, 0.2]),      # drop 5% or 20% of all pixels
    iaa.Sharpen((0.0, 1.0)),       # sharpen the image
    iaa.Affine(rotate=(-45, 45)),  # rotate by -45 to 45 degrees (affects heatmaps)
    iaa.ElasticTransformation(alpha=50, sigma=5)  # apply water effect (affects␣
↪heatmaps)
], random_order=True)

# Augment images and heatmaps.
images_aug = []
heatmaps_aug = []
for _ in range(5):
    images_aug_i, heatmaps_aug_i = seq(image=image, heatmaps=depth)
    images_aug.append(images_aug_i)
    heatmaps_aug.append(heatmaps_aug_i)

# We want to generate an image of original input images and heatmaps
# before/after augmentation.
# It is supposed to have five columns:
# (1) original image,
# (2) augmented image,
# (3) augmented heatmap on top of augmented image,
# (4) augmented heatmap on its own in jet color map,
# (5) augmented heatmap on its own in intensity colormap.
# We now generate the cells of these columns.
#
# Note that we add a [0] after each heatmap draw command. That's because
# the heatmaps object can contain many sub-heatmaps and hence we draw
# command returns a list of drawn sub-heatmaps.
# We only used one sub-heatmap, so our lists always have one entry.
cells = []
for image_aug, heatmap_aug in zip(images_aug, heatmaps_aug):
    cells.append(image)                                                    # column 1
    cells.append(image_aug)                                                # column 2
    cells.append(heatmap_aug.draw_on_image(image_aug)[0])                  # column 3
    cells.append(heatmap_aug.draw(size=image_aug.shape[:2])[0])            # column 4
    cells.append(heatmap_aug.draw(size=image_aug.shape[:2], cmap=None)[0])  # column 5

# Convert cells to grid image and save.
grid_image = ia.draw_grid(cells, cols=5)
imageio.imwrite("example_heatmaps.jpg", grid_image)
```

5.3 Multiple sub-heatmaps per heatmaps object

The above example augmented a single heatmap with shape `(H, W)` for the example image. If you want to augment more heatmaps per image, you can simply extend the heatmap array's shape to `(H, W, C)`, where `C` is the number of heatmaps. The following example instantiates one heatmap object containing three sub-heatmaps and draws them onto the image. Heatmap augmentation would be done in the exactly same way as in the previous example.

```
import imageio
import numpy as np
import imgaug as ia
from imgaug.augmentables.heatmaps import HeatmapsOnImage
```

(continues on next page)

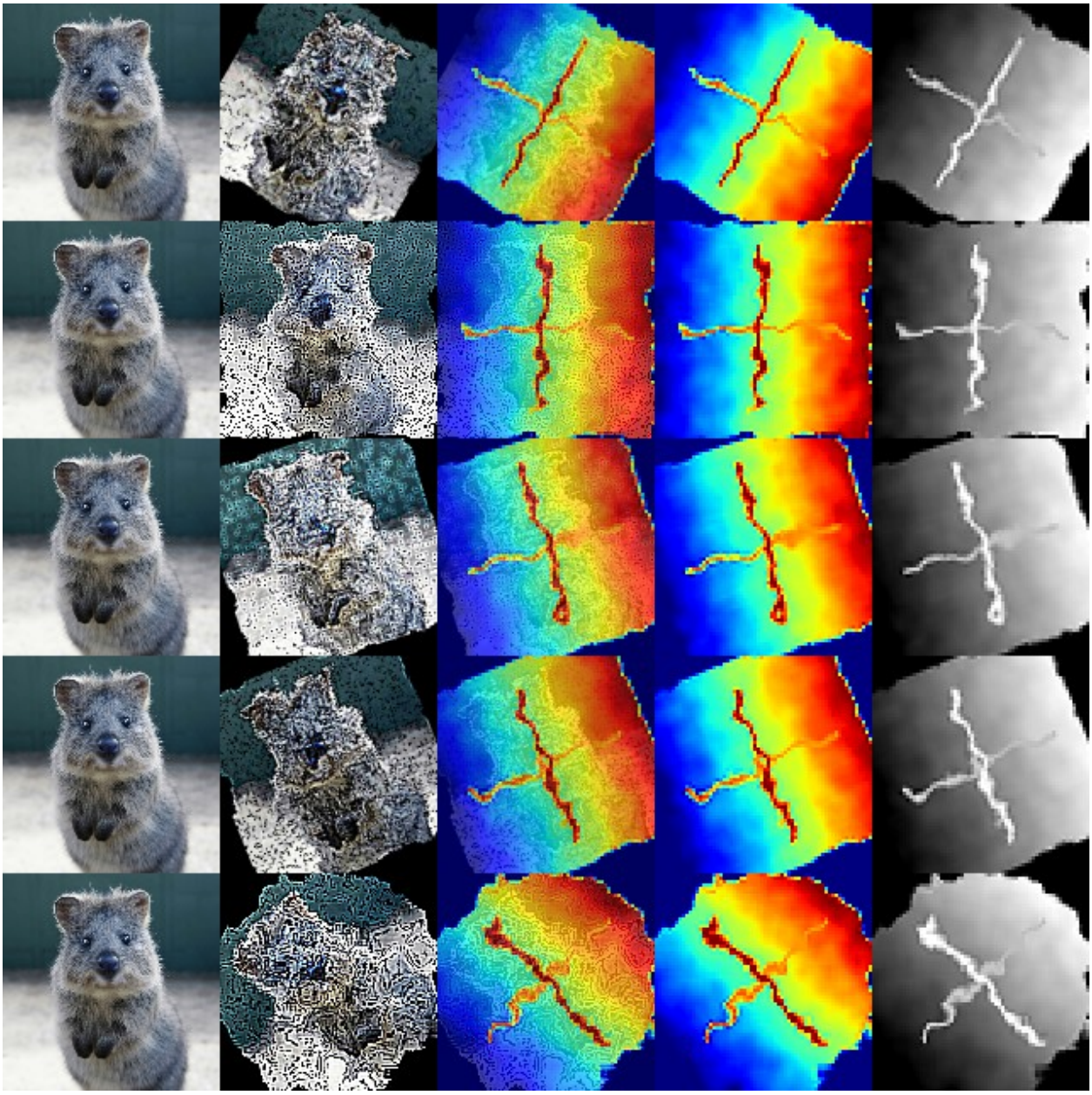


Fig. 1: Results of the above example code. Columns show: (1) Original image, (2) augmented image, (3) augmented heatmap overlayed with augmented image, (4) augmented heatmap alone in jet color map, (5) augmented heatmap alone as intensity map.

(continued from previous page)

```
# Load an image and generate a heatmap array with three sub-heatmaps.
# Each sub-heatmap contains just three horizontal lines, with one of them
# having a higher value (1.0) than the other two (0.2).
image = ia.quokka(size=(128, 128), extract="square")
heatmap = np.zeros((128, 128, 3), dtype=np.float32)
for i in range(3):
    heatmap[1*30-5:1*30+5, 10:-10, i] = 1.0 if i == 0 else 0.5
    heatmap[2*30-5:2*30+5, 10:-10, i] = 1.0 if i == 1 else 0.5
    heatmap[3*30-5:3*30+5, 10:-10, i] = 1.0 if i == 2 else 0.5
heatmap = HeatmapsOnImage(heatmap, shape=image.shape)

# Draw image and the three sub-heatmaps on it.
# We draw four columns: (1) image, (2-4) heatmaps one to three drawn on
# top of the image.
subheatmaps_drawn = heatmap.draw_on_image(image)
cells = [image, subheatmaps_drawn[0], subheatmaps_drawn[1],
         subheatmaps_drawn[2]]
grid_image = np.hstack(cells)  # Horizontally stack the images
imageio.imwrite("example_multiple_heatmaps.jpg", grid_image)
```

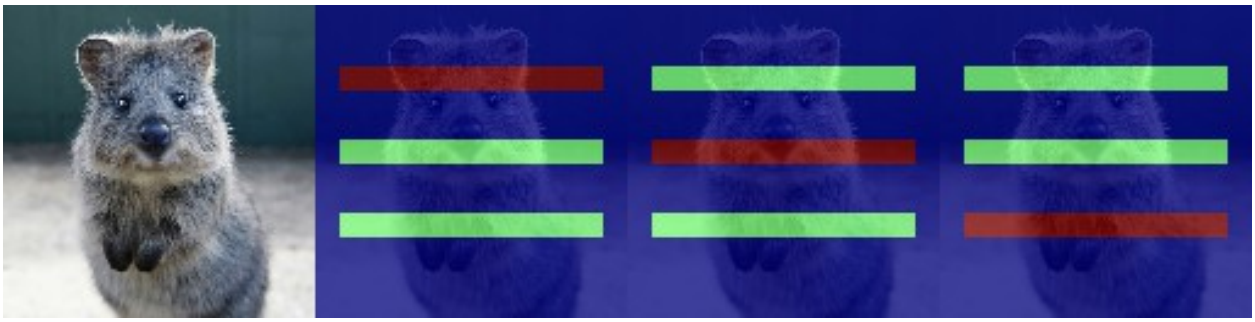


Fig. 2: Results of the above example code. It shows the original image with three heatmaps. The three heatmaps were combined in one `HeatmapsOnImage` object.

5.4 Accessing the heatmap array

After augmentation you probably want to access the heatmap's numpy array. This is done using the function `HeatmapsOnImage.get_arr()`. That functions output shape will match your original heatmap array's shape, i.e. either `(H, W)` or `(H, W, C)`. The below code shows an example, where that function's result is changed and then used to instantiate a new `HeatmapsOnImage` object.

Alternatively you could also change the heatmap object's internal array, saved as `HeatmapsOnImage.arr_0to1`. As the name indicates, it is always normalized to the range `0.0` to `1.0`, while `get_arr()` reverses that normalization. It has also always shape `(H, W, C)`, with `C>=1`.

```
import imageio
import numpy as np
import imgaug as ia
from imgaug.augmentables.heatmaps import HeatmapsOnImage

# Load an image and generate a heatmap array containing one horizontal line.
```

(continues on next page)

(continued from previous page)

```
image = ia.quokka(size=(128, 128), extract="square")
heatmap = np.zeros((128, 128, 1), dtype=np.float32)
heatmap[64-4:64+4, 10:-10, 0] = 1.0
heatmap1 = HeatmapsOnImage(heatmap, shape=image.shape)

# Extract the heatmap array from the heatmap object, change it and create
# a second heatmap.
arr = heatmap1.get_arr()
arr[10:-10, 64-4:64+4] = 0.5
heatmap2 = HeatmapsOnImage(arr, shape=image.shape)

# Draw image and heatmaps before/after changing the array.
# We draw three columns:
# (1) original image,
# (2) heatmap drawn on image,
# (3) heatmap drawn on image, with some changes made to the heatmap array.
cells = [image,
         heatmap1.draw_on_image(image)[0],
         heatmap2.draw_on_image(image)[0]]
grid_image = np.hstack(cells)  # Horizontally stack the images
imageio.imwrite("example_heatmaps_arr.jpg", grid_image)
```

Fig. 3: Results of the above example code. It shows the original image, a corresponding heatmap and again the same heatmap after its array was read out and changed.

5.5 Resizing heatmaps

When working with heatmaps it is common that the size of the input images and the heatmap sizes don't match or are supposed to not match (e.g. because predicted network output are of low resolution). `HeatmapsOnImage` offers several functions to deal with such situations: `HeatmapsOnImage.avg_pool(kernel_size)` applies average pooling to images, `HeatmapsOnImage.max_pool(kernel_size)` analogously max pooling and `HeatmapsOnImage.resize(size, [interpolation])` performs resizing. For the pooling functions the kernel size is expected to be a single integer or a tuple of two/three entries (size along each dimension). For `resize`, the size is expected to be a `(height, width)` tuple and `interpolation` can be one of the strings `nearest` (nearest neighbour interpolation), `linear`, `cubic` (default) or `area`.

The below code shows an example. It instantiates a simple `128x128` heatmap with two horizontal lines (one of which is blurred) and a small square in the center. It then applies average pooling, max pooling and resizing to heatmap sizes `64x64`, `32x32` and `16x16`. Then, an output image is generated with six rows: The first three show the results of average/max pooling and resizing, while the rows three to six show the same results after again resizing them to `128x128` using nearest neighbour upscaling.

```
import imageio
import numpy as np
import imgaug as ia
import imgaug.augmenters as iaa
from imgaug.augmentables.heatmaps import HeatmapsOnImage


def pad_by(image, amount):
    return ia.pad(image,
                  top=amount, right=amount, bottom=amount, left=amount)

def draw_heatmaps(heatmaps, upscale=False):
    drawn = []
    for heatmap in heatmaps:
        if upscale:
            drawn.append(
                heatmap.resize((128, 128), interpolation="nearest")
                       .draw()[0]
            )
        else:
            size = heatmap.get_arr().shape[0]
            pad_amount = (128-size)//2
            drawn.append(pad_by(heatmap.draw()[0], pad_amount))
    return drawn

# Generate an example heatmap with two horizontal lines (first one blurry,
# second not) and a small square.
heatmap = np.zeros((128, 128, 1), dtype=np.float32)
heatmap[32-4:32+4, 10:-10, 0] = 1.0
heatmap = iaa.GaussianBlur(3.0).augment_image(heatmap)
heatmap[96-4:96+4, 10:-10, 0] = 1.0
heatmap[64-2:64+2, 64-2:64+2, 0] = 1.0
heatmap = HeatmapsOnImage(heatmap, shape=(128, 128, 1))

# Scale the heatmaps using average pooling, max pooling and resizing with
# default interpolation (cubic).
avg_pooled = [heatmap, heatmap.avg_pool(2), heatmap.avg_pool(4),
              heatmap.avg_pool(8)]
max_pooled = [heatmap, heatmap.max_pool(2), heatmap.max_pool(4),
              heatmap.max_pool(8)]
resized = [heatmap, heatmap.resize((64, 64)), heatmap.resize((32, 32)),
           heatmap.resize((16, 16))]

# Draw an image of all scaled heatmaps.
cells = draw_heatmaps(avg_pooled)\
    + draw_heatmaps(max_pooled)\
    + draw_heatmaps(resized)\
    + draw_heatmaps(avg_pooled, upscale=True)\
    + draw_heatmaps(max_pooled, upscale=True)\
    + draw_heatmaps(resized, upscale=True)
grid_image = ia.draw_grid(cells, cols=4)
imageio.imwrite("example_heatmaps_scaling.jpg", grid_image)
```

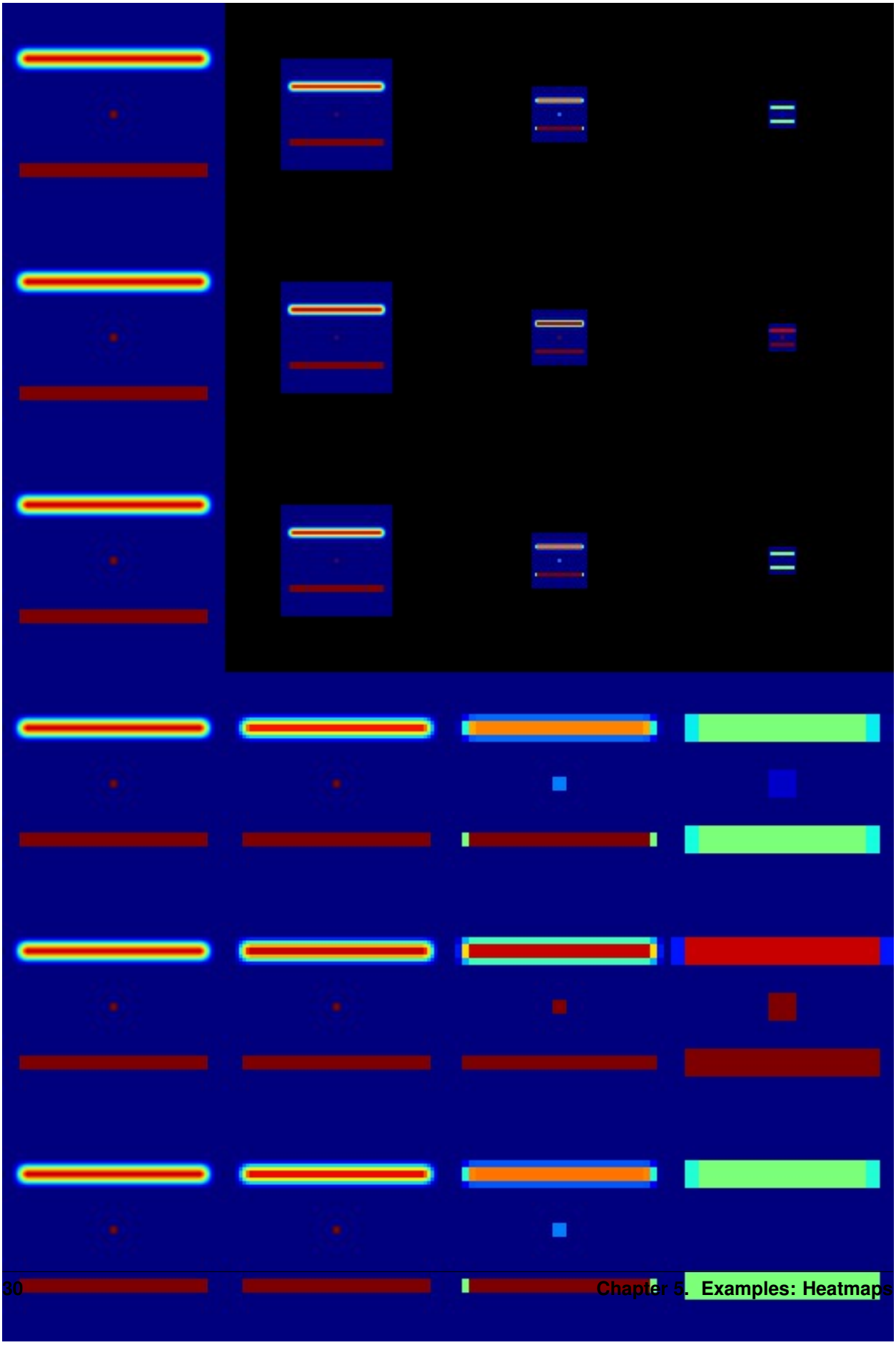


Fig. 4: Results of the above example code. It shows six rows: (Rows 1-3) scaling via average pooling, max pooling and

5.6 Padding heatmaps

Another common operation is padding of images and heatmaps, especially to squared sizes. This is done for images using `imgaug.pad(image, [top], [right], [bottom], [left], [mode], [cval])` and `imgaug.pad_to_aspect_ratio(image, aspect_ratio, [mode], [cval], [return_pad_amounts])`. For heatmaps it is done using `HeatmapsOnImage.pad([top], [right], [bottom], [left], [mode], [cval])` and `HeatmapsOnImage.pad_to_aspect_ratio(aspect_ratio, [mode], [cval], [return_pad_amounts])`. In both cases, `pad()` expects pixel amounts (i.e. integers) and `pad_to_aspect_ratio()` the target aspect ratio, given as a float denoting `ratio = width / height` (i.e. a value of `1.0` would lead to a squared image/heatmap, while `2.0` would lead to a fairly wide image/heatmap).

The below code shows an example for padding. It starts with a squared sized image and heatmap, cuts both so that they are more wide than high and then zero-pads both back to squared size.

```
import imageio
import numpy as np
import imgaug as ia
from imgaug.augmentables.heatmaps import HeatmapsOnImage

# Load example image and generate example heatmap with one horizontal line
image = ia.quokka((128, 128), extract="square")
heatmap = np.zeros((128, 128, 1), dtype=np.float32)
heatmap[64-4:64+4, 10:-10, 0] = 1.0

# Cut image and heatmap so that they are no longer squared
image = image[32:-32, :, :]
heatmap = heatmap[32:-32, :, :]

heatmap = HeatmapsOnImage(heatmap, shape=(128, 128, 1))

# Pad images and heatmaps by pixel amounts or to aspect ratios
# We pad both back to squared size of 128x128
images_padded = [
    ia.pad(image, top=32, bottom=32),
    ia.pad_to_aspect_ratio(image, 1.0)
]
heatmaps_padded = [
    heatmap.pad(top=32, bottom=32),
    heatmap.pad_to_aspect_ratio(1.0)
]

# Draw an image of all padded images and heatmaps
cells = [
    images_padded[0],
    heatmaps_padded[0].draw_on_image(images_padded[0])[0],
    images_padded[1],
    heatmaps_padded[1].draw_on_image(images_padded[1])[0]
]

grid_image = ia.draw_grid(cells, cols=2)
imageio.imwrite("example_heatmaps_padding.jpg", grid_image)
```

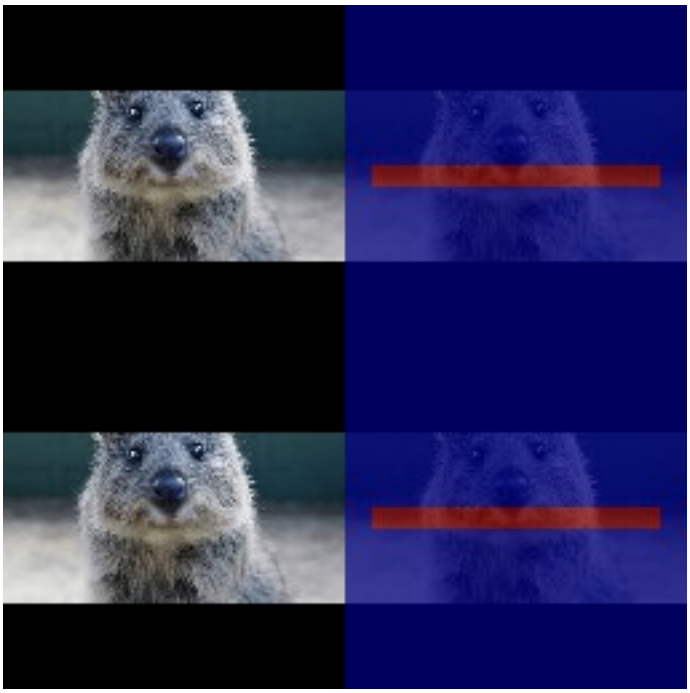


Fig. 5: Results of the above example code. It shows an input image and a heatmap that were both first cut to `64x128` and then padded back to squared size of `128x128`. First row uses `pad()`, second uses `pad_to_aspect_ratio()`.

CHAPTER 6

Examples: Segmentation Maps and Masks

`imgaug` offers support for segmentation map data, such as semantic segmentation maps, instance segmentation maps or ordinary masks. Segmentation maps can be augmented correspondingly to images. E.g. if an image is rotated by 45°, the corresponding segmentation map for that image will also be rotated by 45°.

Note: All augmentation functions for segmentation maps are implemented under the assumption of augmenting **ground truth** data. As such, segmentation maps will be affected by augmentations that change the geometry of images (e.g. affine transformations, cropping, resizing), but not by other augmentations (e.g. gaussian noise, saturation changes, grayscaling, dropout, ...).

Features of the library's segmentation map support:

- Represent segmentation maps as objects (`imgaug.augmentables.segmaps.SegmentationMapsOnImage`).
- Support integer maps (integer dtypes, usually `int32`) and boolean masks (dtype `numpy.bool_`).
- Augment segmentation maps (only geometry-affecting augmentations, e.g. affine transformations, cropping, ...).
- Use different resolutions for segmentation maps and images (e.g. `32x32` segmentation maps and `256x256` for the corresponding images).
- Draw segmentation maps – on their own or on images (`SegmentationMapsOnImage.draw()`, `SegmentationMapsOnImage.draw_on_image()`).
- Resize segmentation maps (`SegmentationMapsOnImage.resize()`).
- Pad segmentation maps by pixel amounts or to desired aspect ratios (`SegmentationMapsOnImage.pad()`, `SegmentationMapsOnImage.pad_to_aspect_ratio()`).

6.1 Notebook

A jupyter notebook for segmentation map augmentation is available at *Jupyter Notebooks*. The notebooks are usually more up to date and contain more examples than the ReadTheDocs documentation.

6.2 A simple example

The following example loads a standard image and defines a corresponding `int32` segmentation map. The image and segmentation map are augmented in the same way and the results are visualized.

```
import imageio
import numpy as np
import imgaug as ia
import imgaug.augmenters as iaa
from imgaug.augmentables.segmaps import SegmentationMapsOnImage

ia.seed(1)

# Load an example image (uint8, 128x128x3).
image = ia.quokka(size=(128, 128), extract="square")

# Define an example segmentation map (int32, 128x128).
# Here, we arbitrarily place some squares on the image.
# Class 0 is our intended background class.
segmap = np.zeros((128, 128, 1), dtype=np.int32)
segmap[28:71, 35:85, 0] = 1
segmap[10:25, 30:45, 0] = 2
segmap[10:25, 70:85, 0] = 3
segmap[10:110, 5:10, 0] = 4
segmap[118:123, 10:110, 0] = 5
segmap = SegmentationMapsOnImage(segmap, shape=image.shape)

# Define our augmentation pipeline.
seq = iaa.Sequential([
    iaa.Dropout([0.05, 0.2]),      # drop 5% or 20% of all pixels
    iaa.Sharpen((0.0, 1.0)),       # sharpen the image
    iaa.Affine(rotate=(-45, 45)),  # rotate by -45 to 45 degrees (affects segmaps)
    iaa.ElasticTransformation(alpha=50, sigma=5)  # apply water effect (affects␣
↪segmaps)
], random_order=True)

# Augment images and segmaps.
images_aug = []
segmaps_aug = []
for _ in range(5):
    images_aug_i, segmaps_aug_i = seq(image=image, segmentation_maps=segmap)
    images_aug.append(images_aug_i)
    segmaps_aug.append(segmaps_aug_i)

# We want to generate an image containing the original input image and
# segmentation maps before/after augmentation. (Both multiple times for
# multiple augmentations.)
#
# The whole image is supposed to have five columns:
# (1) original image,
# (2) original image with segmap,
# (3) augmented image,
# (4) augmented segmap on augmented image,
# (5) augmented segmap on its own in.
#
# We now generate the cells of these columns.
#
```

(continues on next page)

(continued from previous page)

```
# Note that draw_on_image() and draw() both return lists of drawn
# images. Assuming that the segmentation map array has shape (H,W,C),
# the list contains C items.
cells = []
for image_aug, segmap_aug in zip(images_aug, segmaps_aug):
    cells.append(image)                                       # column 1
    cells.append(segmap.draw_on_image(image)[0])              # column 2
    cells.append(image_aug)                                   # column 3
    cells.append(segmap_aug.draw_on_image(image_aug)[0])      # column 4
    cells.append(segmap_aug.draw(size=image_aug.shape[:2])[0])  # column 5

# Convert cells to a grid image and save.
grid_image = ia.draw_grid(cells, cols=5)
imageio.imwrite("example_segmaps.jpg", grid_image)
```

6.3 Using boolean masks

In order to augment masks, you can simply use boolean arrays. Everything else is identical to `int32` maps. The below code shows an example and is very similar to the previous code for `int32` maps. It noteably changes `np.zeros((128, 128, 1), dtype=np.int32)` to `np.zeros((128, 128, 1), dtype=bool)`.

```
import imageio
import numpy as np
import imgaug as ia
from imgaug.augmentables.segmaps import SegmentationMapsOnImage

# Load an example image (uint8, 128x128x3).
image = ia.quokka(size=(128, 128), extract="square")

# Create an example mask (bool, 128x128).
# Here, we arbitrarily place a square on the image.
segmap = np.zeros((128, 128, 1), dtype=bool)
segmap[28:71, 35:85, 0] = True
segmap = SegmentationMapsOnImage(segmap, shape=image.shape)

# Draw three columns: (1) original image,
# (2) original image with mask on top, (3) only mask
cells = [
    image,
    segmap.draw_on_image(image)[0],
    segmap.draw(size=image.shape[:2])[0]
]

# Convert cells to a grid image and save.
grid_image = ia.draw_grid(cells, cols=3)
imageio.imwrite("example_segmaps_bool.jpg", grid_image)
```

6.4 Accessing the segmentation map array

After augmentation it is often desired to re-access the segmentation map array. This can be done using `SegmentationMapsOnImage.get_arr()`, which returns a segmentation map array with the same shape and dtype as was originally provided as `arr` to `SegmentationMapsOnImage(arr, ...)`.

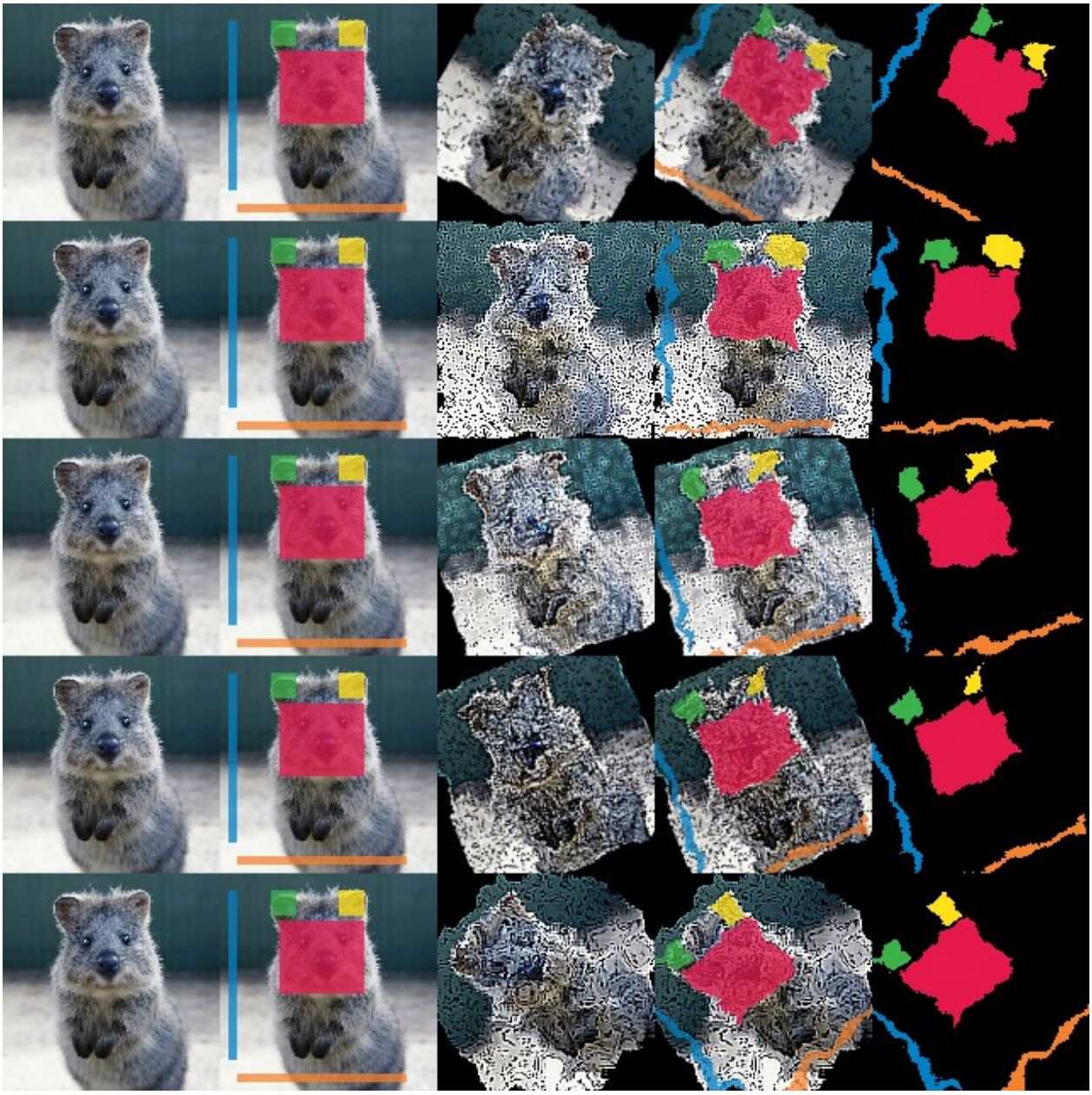


Fig. 1: Results of the above example code. Columns show: (1) Original image, (2) original segmentation map drawn on original image, (3) augmented image, (4) augmented segmentation map drawn on augmented image, (5) augmented segmentation map drawn on its own.

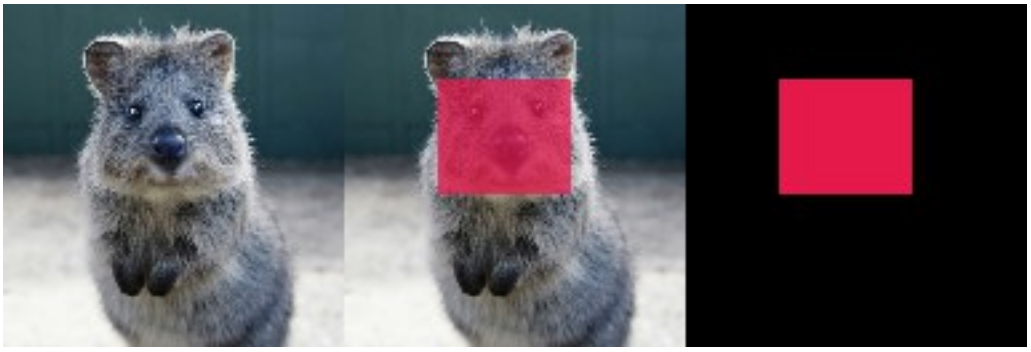


Fig. 2: Results of the above example code. Columns show: (1) Original image, (2) boolean segmentation map (i.e. mask) drawn on image, (3) boolean segmentation map drawn on its own.

The below code shows an example that accesses and changes the array.

```
import imageio
import numpy as np
import imgaug as ia
from imgaug.augmentables.segmaps import SegmentationMapsOnImage

# Load an example image (uint8, 128x128x3).
image = ia.quokka(size=(128, 128), extract="square")

# Create an example segmentation map (int32, 128x128).
# Here, we arbitrarily place some squares on the image.
# Class 0 is the background class.
segmap = np.zeros((128, 128, 1), dtype=np.int32)
segmap[28:71, 35:85, 0] = 1
segmap[10:25, 30:45, 0] = 2
segmap[10:25, 70:85, 0] = 3
segmap[10:110, 5:10, 0] = 4
segmap[118:123, 10:110, 0] = 5
segmap1 = SegmentationMapsOnImage(segmap, shape=image.shape)

# Read out the segmentation map's array, change it and create a new
# segmentation map
arr = segmap1.get_arr()
arr[10:110, 5:10, 0] = 5
segmap2 = ia.SegmentationMapsOnImage(arr, shape=image.shape)

# Draw three columns: (1) original image, (2) original image with
# unaltered segmentation map on top, (3) original image with altered
# segmentation map on top
cells = [
    image,
    segmap1.draw_on_image(image)[0],
    segmap2.draw_on_image(image)[0]
]

# Convert cells to grid image and save.
grid_image = ia.draw_grid(cells, cols=3)
imageio.imwrite("example_segmaps_array.jpg", grid_image)
```

Fig. 3: Results of the above example code. Columns show: (1) Original image, (2) original segmentation map drawn on original image, (3) segmentation map with modified array drawn on image.

6.5 Resizing and padding

Segmentation maps can easily be resized and padded. The methods are identical to the ones used for heatmaps (see :doc:`examples_heatmaps`), though segmentation maps do not offer resizing via average or max pooling. The `resize()` method also defaults to nearest neighbour interpolation (instead of `cubic` interpolation) and it is recommended to not change that.

The functions for resizing and padding are:

- `SegmentationMapsOnImage.resize(sizes, interpolation="nearest")`: Resizes to `sizes` given as a tuple `(height, width)`. Interpolation can be `nearest`, `linear`, `cubic` and `area`, but only `nearest` is actually recommended.
- `SegmentationMapsOnImage.pad(top=0, right=0, bottom=0, left=0, mode="constant", cval=0)`: Pads the segmentation map by given pixel amounts. Uses by default constant value padding with value `0`, i.e. zero-padding. Possible padding modes are the same as for `numpy.pad()`, i.e. `constant`, `edge`, `linear_ramp`, `maximum`, `mean`, `median`, `minimum`, `reflect`, `symmetric` and `wrap`.
- `SegmentationMapsOnImage.pad_to_aspect_ratio(aspect_ratio, mode="constant", cval=0, return_pad_amounts=False)`: Same as `pad()`, but pads an image towards a desired aspect ratio (`ratio = width / height`). E.g. use `1.0` for squared segmentation maps or `2.0` for maps that are twice as wide as they are high.

CHAPTER 7

Stochastic Parameters

7.1 Introduction

When augmenting images during experiments, usually one wants to augment each image in different ways. E.g. when rotating images, not every image is supposed to be rotated by 10 degrees. Instead, only some are supposed to be rotated by 10 degrees, while others should be rotated by 17 degrees or 5 degrees or -12 degrees - and so on. This can be achieved using random functions, but reimplementing these, making sure that they generate the expected values and getting them to work with *determinism* is cumbersome. To avoid all of this work, the library uses *Stochastic Parameters*. These are usually abstract representations of probability distributions, e.g. the normal distribution *N(0, 1.0)* or the uniform range *[0.0, 10.0]*. Basically all augmenters accept these stochastic parameters, making it easy to control value ranges. They are all adapted to work with *determinism* out of the box.

The below code shows their usage:

```
from imgaug import augmenters as iaa
from imgaug import parameters as iap

seq = iaa.Sequential([
    iaa.GaussianBlur(
        sigma=iap.Uniform(0.0, 1.0)
    ),
    iaa.ContrastNormalization(
        iap.Choice(
            [1.0, 1.5, 3.0],
            p=[0.5, 0.3, 0.2]
        )
    ),
    iaa.Affine(
        rotate=iap.Normal(0.0, 30),
        translate_px=iap.RandomSign(iap.Poisson(3))
    ),
    iaa.AddElementwise(
        iap.Discretize(
            (iap.Beta(0.5, 0.5) * 2 - 1.0) * 64
```

(continues on next page)

(continued from previous page)

```
        )
    ),
    iaa.Multiply(
        iap.Positive(iap.Normal(0.0, 0.1)) + 1.0
    )
])
```

The example does the following:

- Blur each image by *sigma*, where *sigma* is sampled from the uniform range *[0.0, 1.0)*. Example values: *0.053, 0.414, 0.389, 0.277, 0.981*.
- Increase the contrast either to 100% (50% chance of being chosen) or by 150% (30% chance of being chosen) or 300% (20% chance of being chosen).
- Rotate each image by a random amount of degrees, where the degree is sampled from the normal distribution *N(0, 30)*. Most of the values will be in the range *-60* to *60*.
- Translate each image by n pixels, where n is sampled from a poisson distribution with alpha=3 (pick should be around x=3). As we cant translate by a fraction of a pixel, we pick a discrete distribution here, which poisson is. However, we do not just want to translate towards the right/top (only positive values). So we randomly flip the sign sometimes to get negative pixel amounts too.
- Add to each pixel a random value, sampled from the beta distribution *Beta(0.5, 0.5)*. This distribution has its peaks around 0.0 and 1.0. We multiply this with 2 and subtract 1 to get it into the range *[-1, 1]*. Then we multiply by 64 to get the range *[-64, 64]*. As we beta distribution is continuous, we convert it to a discrete distribution. The result is that a lot of pixel intensities are shifted by -64 or 64 (or a value very close to these two). Some other pixel intensities are kept (mostly) at their old values.
- We use Multiply to make each image brighter. The brightness increase is sampled from a normal distribution, converted to have only positive values. So most values are expected to be in the range *0.0* to *0.2*. We add *1.0* to set the brightness to *1.0* (100%) to *1.2* (120%).

7.2 Continuous Probability Distributions

The following continuous probability distributions are available:

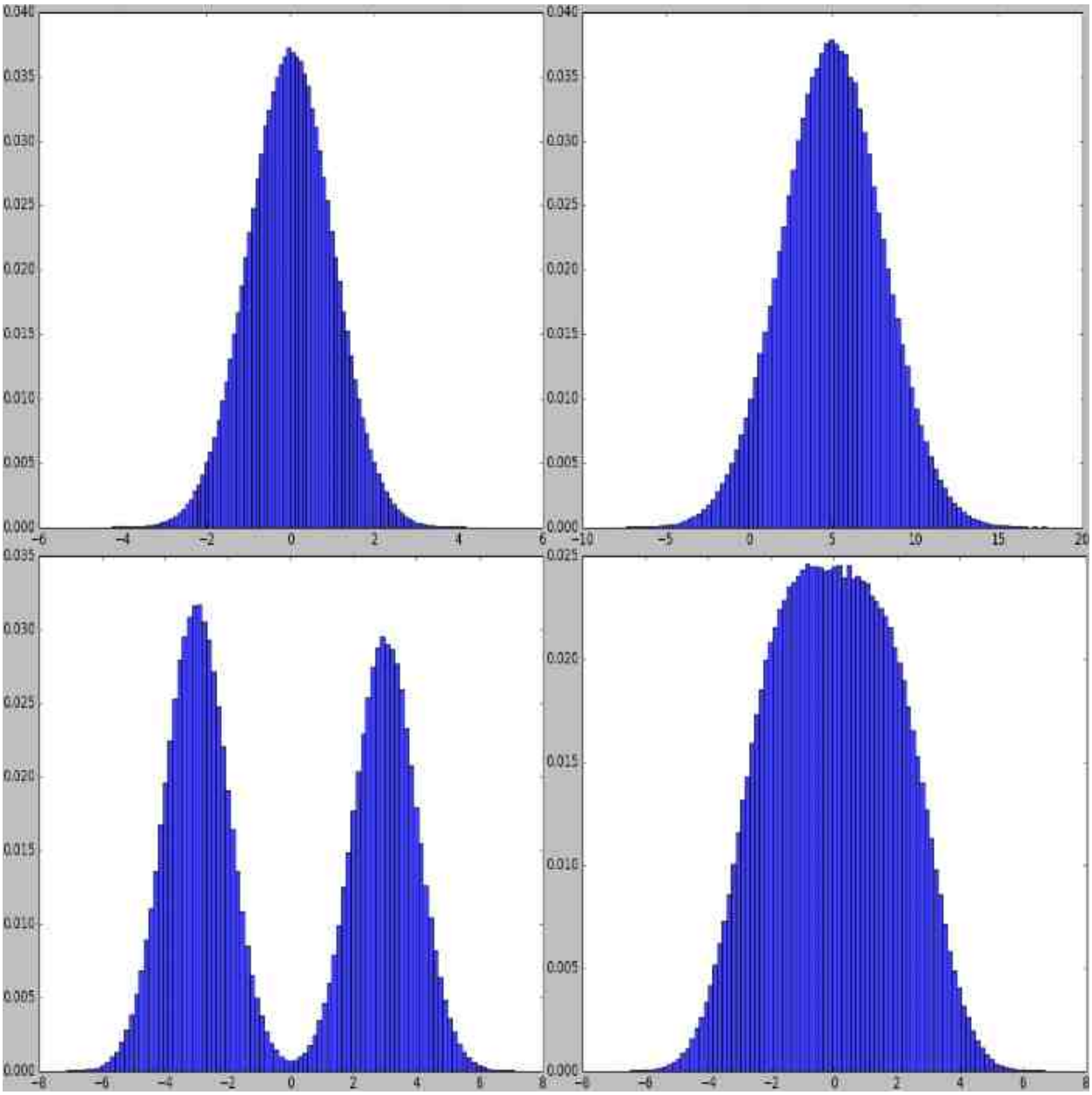
- *Normal(loc, scale)*: The popular normal distribution with mean *loc* and standard deviation *scale*. Example:

```
from imgaug import parameters as iap
params = [
    iap.Normal(0, 1),
    iap.Normal(5, 3),
    iap.Normal(iap.Choice([-3, 3]), 1),
    iap.Normal(iap.Uniform(-3, 3), 1)
]
iap.show_distributions_grid(params)
```

- *Laplace(loc, scale)*: Similarly shaped to a normal distribution. Has its peak at *loc* and width *scale*. Example:

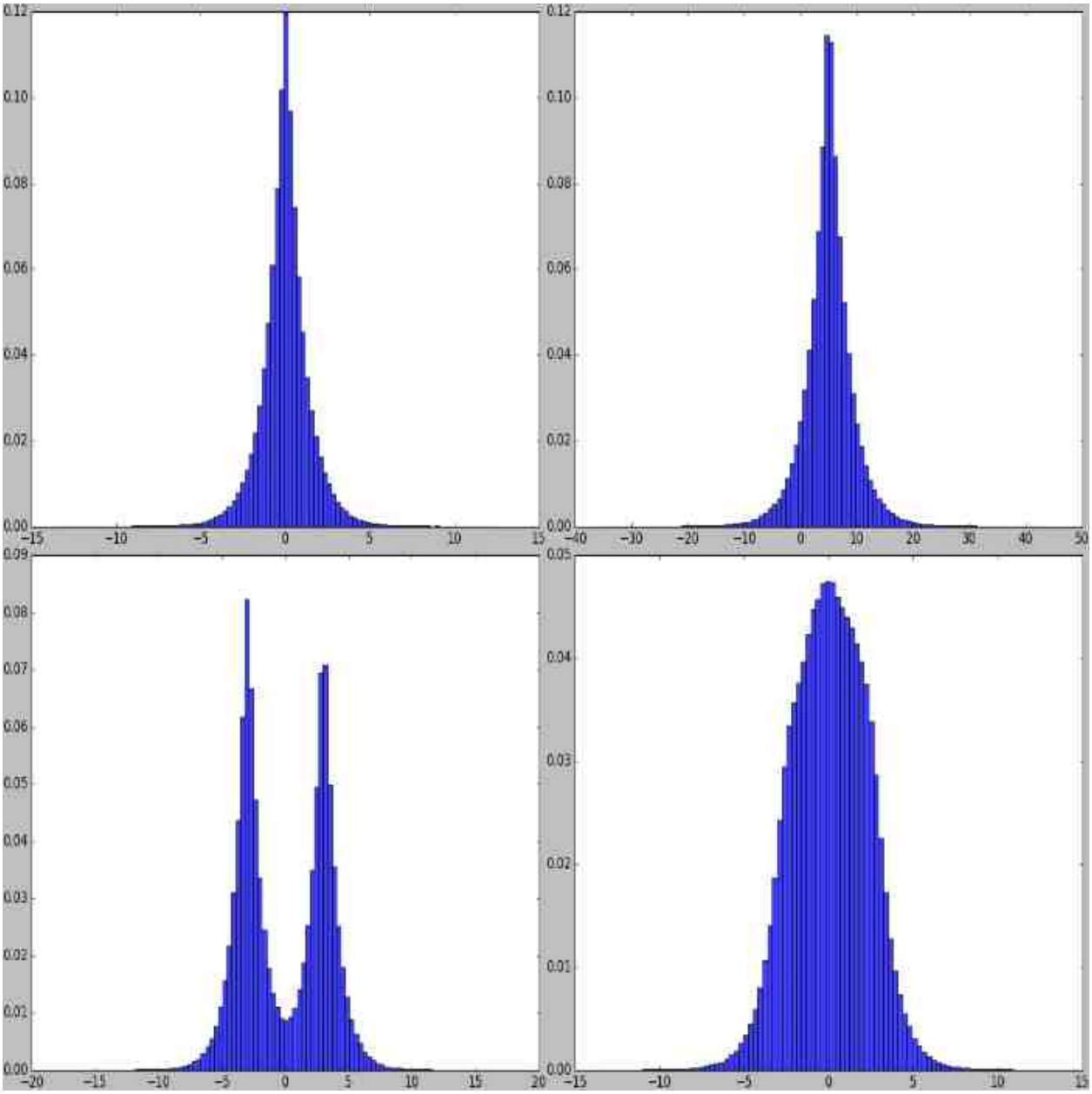
```
from imgaug import parameters as iap
params = [
    iap.Laplace(0, 1),
    iap.Laplace(5, 3),
    iap.Laplace(iap.Choice([-3, 3]), 1),
    iap.Laplace(iap.Uniform(-3, 3), 1)
```

(continues on next page)



(continued from previous page)

```
]
iap.show_distributions_grid(params)
```



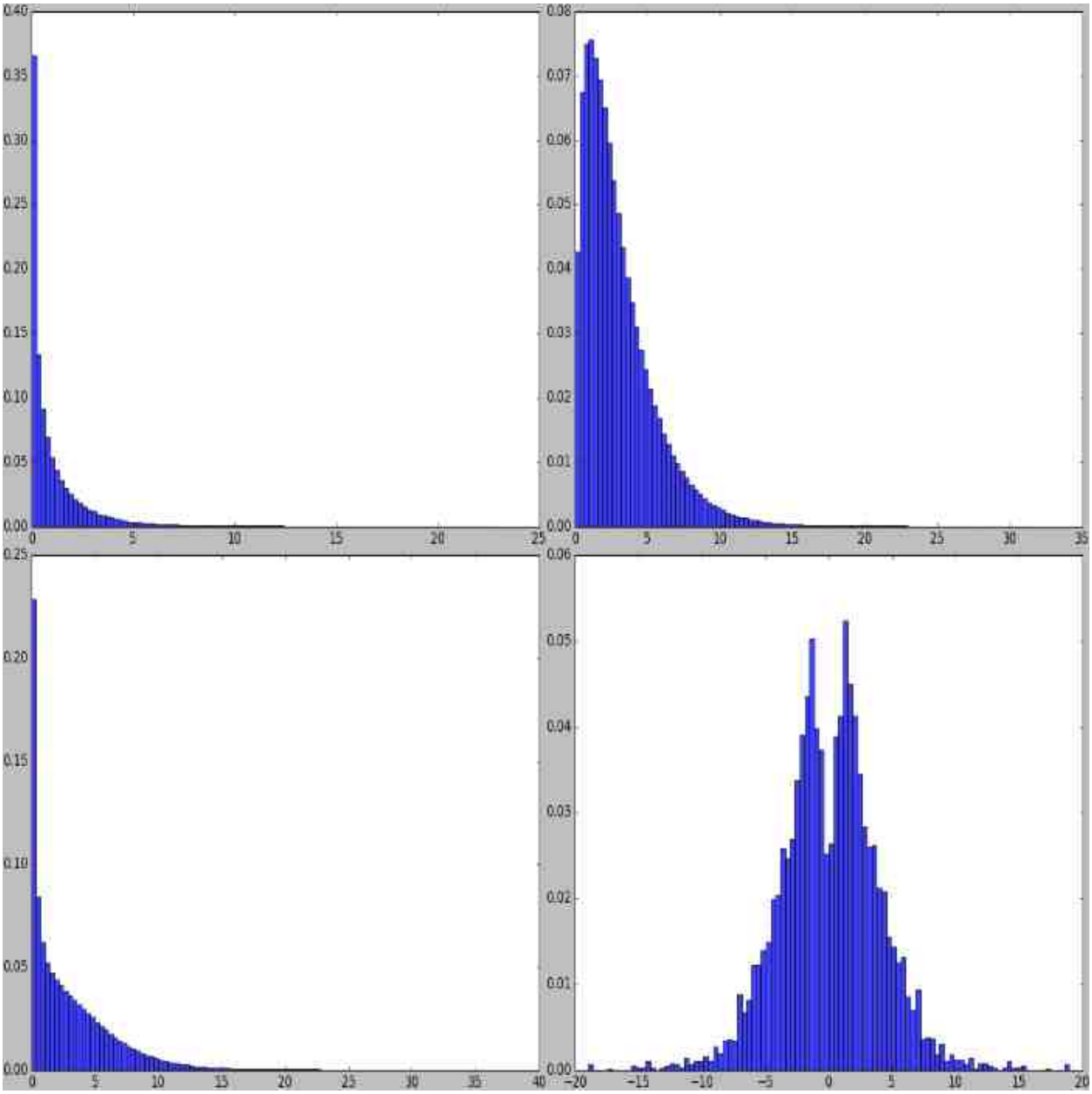
- ***ChiSquare(df)*: The chi-square ("X^2") distribution with *df* degrees of freedom.** Roughly similar to a continuous version of the poisson distribution. Has its peak at *df* and no negative values, only positive ones. Example:

```
from imgaug import parameters as iap
params = [
    iap.ChiSquare(1),
    iap.ChiSquare(3),
    iap.ChiSquare(iap.Choice([1, 5])),
```

(continues on next page)

(continued from previous page)

```
    iap.RandomSign(iap.ChiSquare(3))
]
iap.show_distributions_grid(params)
```



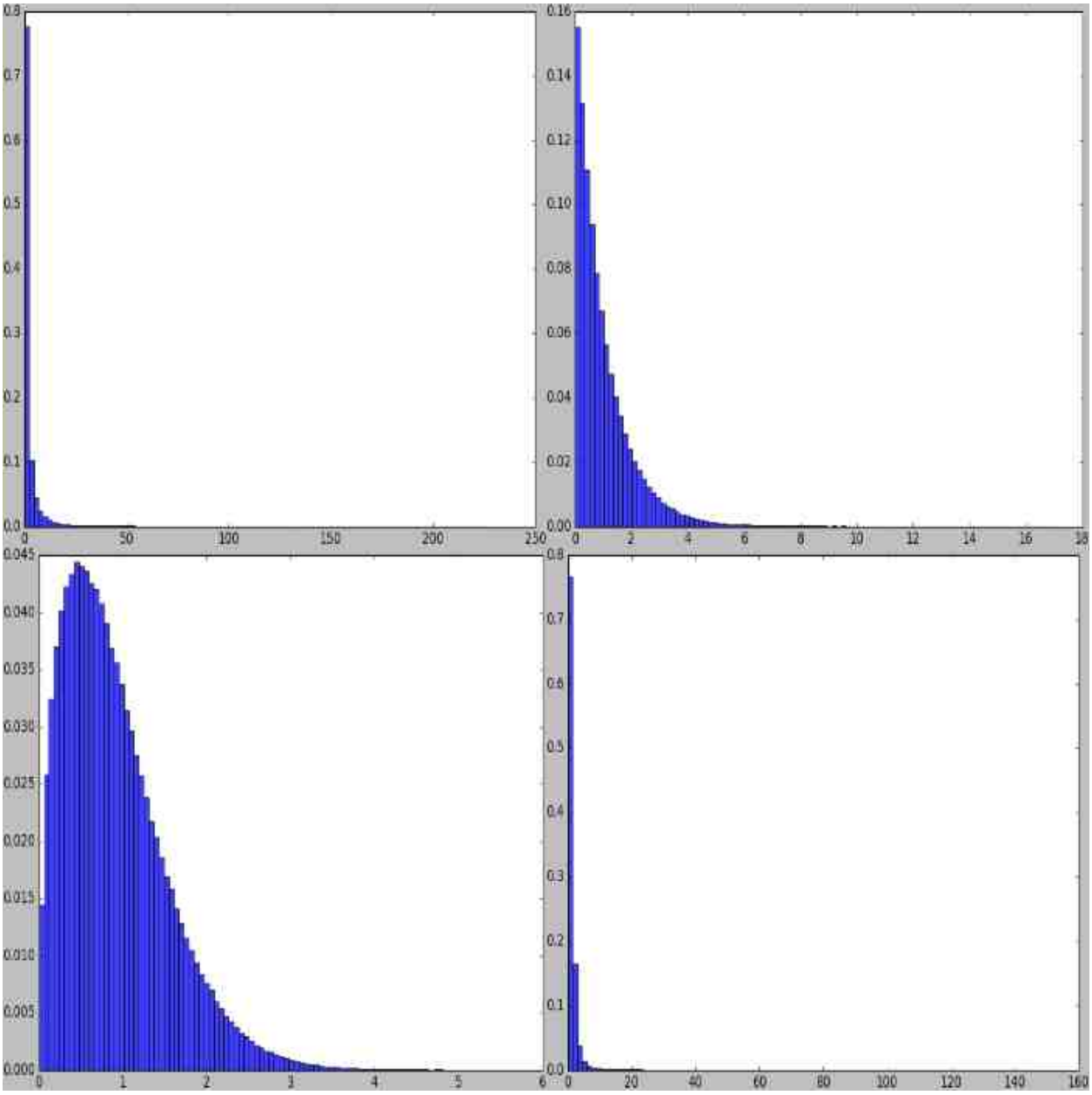
- *Weibull(a)*: Weibull distribution with shape *a*. Example:

```
from imgaug import parameters as iap
params = [
    iap.Weibull(0.5),
    iap.Weibull(1),
    iap.Weibull(1.5),
    iap.Weibull((0.5, 1.5))
]
```

(continues on next page)

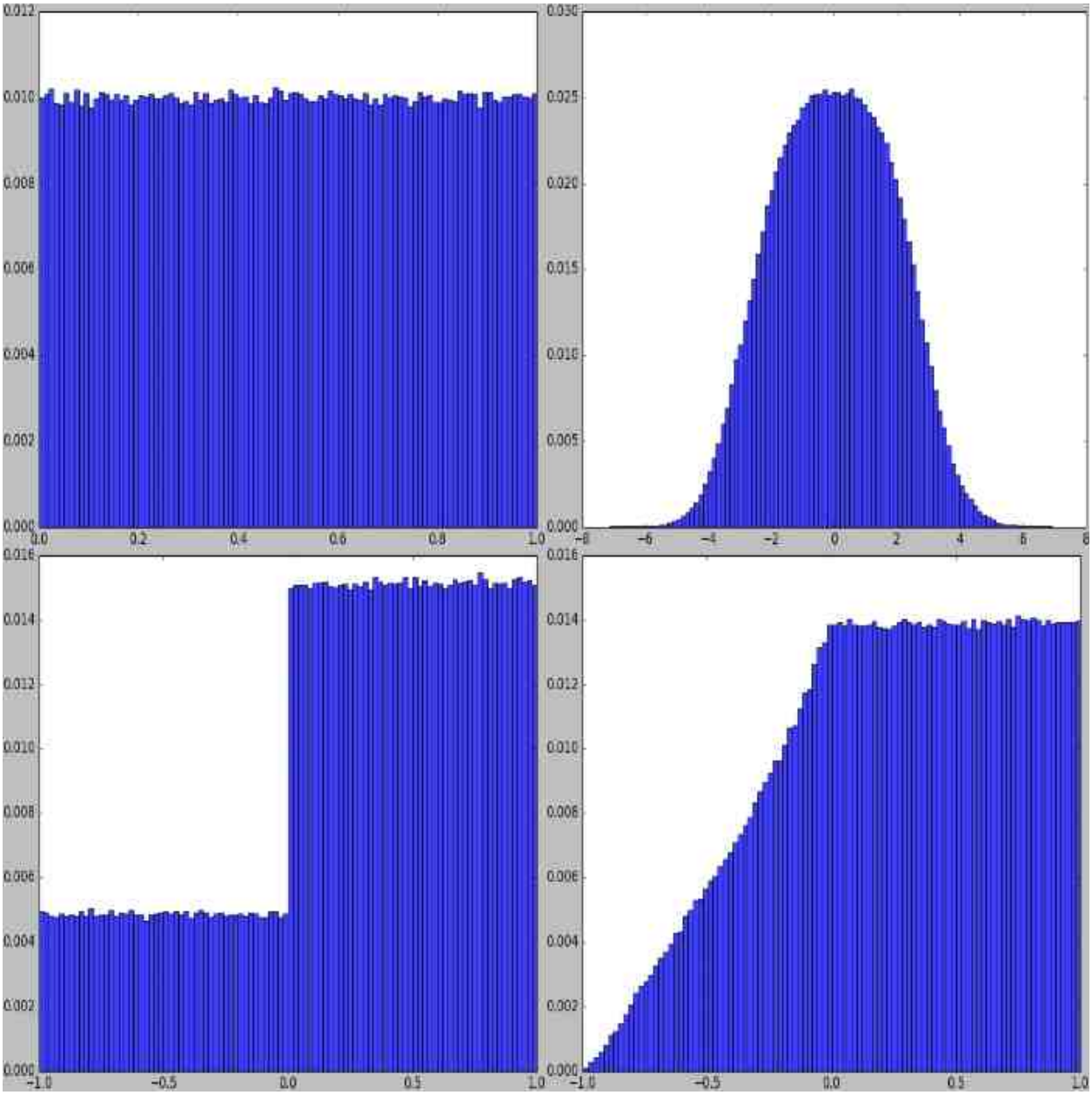
(continued from previous page)

```
iap.show_distributions_grid(params)
```



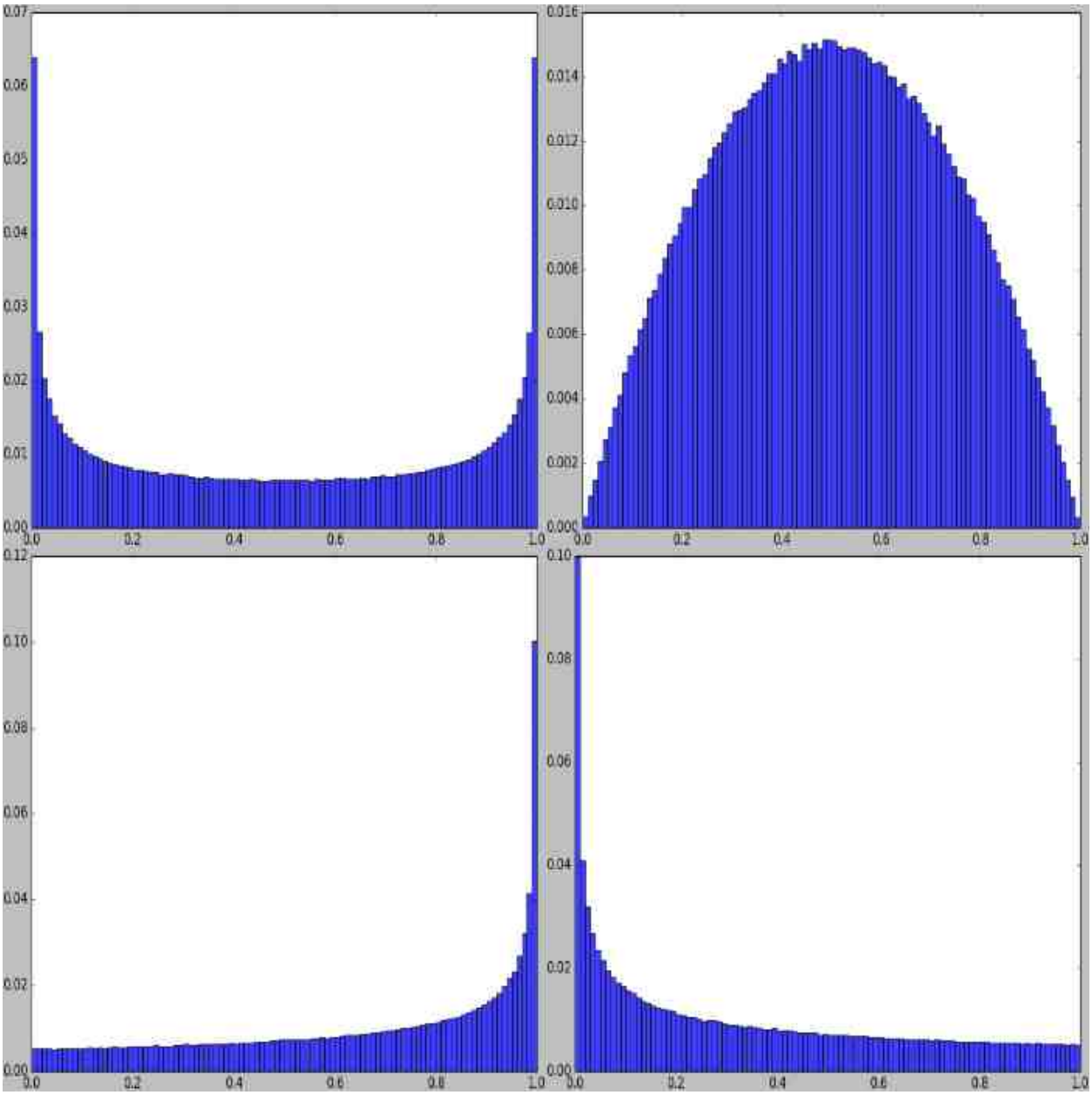
- *Uniform(a, b)*: Uniform distribution in the range *[a, b)*. Example:

```
from imgaug import parameters as iap
params = [
    iap.Uniform(0, 1),
    iap.Uniform(iap.Normal(-3, 1), iap.Normal(3, 1)),
    iap.Uniform([-1, 0], 1),
    iap.Uniform((-1, 0), 1)
]
iap.show_distributions_grid(params)
```



- *Beta(alpha, beta)*: Beta distribution with parameters *alpha* and *beta*. Example:

```
from imgaug import parameters as iap
params = [
    iap.Beta(0.5, 0.5),
    iap.Beta(2.0, 2.0),
    iap.Beta(1.0, 0.5),
    iap.Beta(0.5, 1.0)
]
iap.show_distributions_grid(params)
```

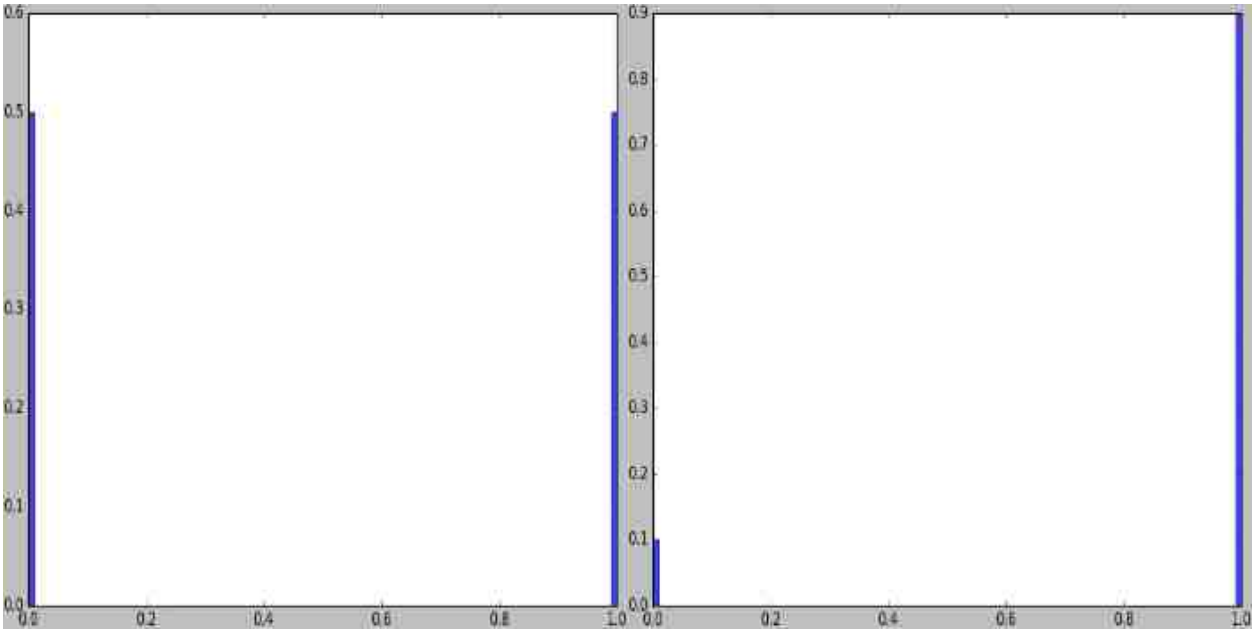


7.3 Discrete Probability Distributions

The following discrete probability distributions are available:

- *Binomial(p)*: The common binomial distribution with probability *p*. Useful to simulate coinflips. Example:

```
from imgaug import parameters as iap
params = [
    iap.Binomial(0.5),
    iap.Binomial(0.9)
]
iap.show_distributions_grid(params)
```

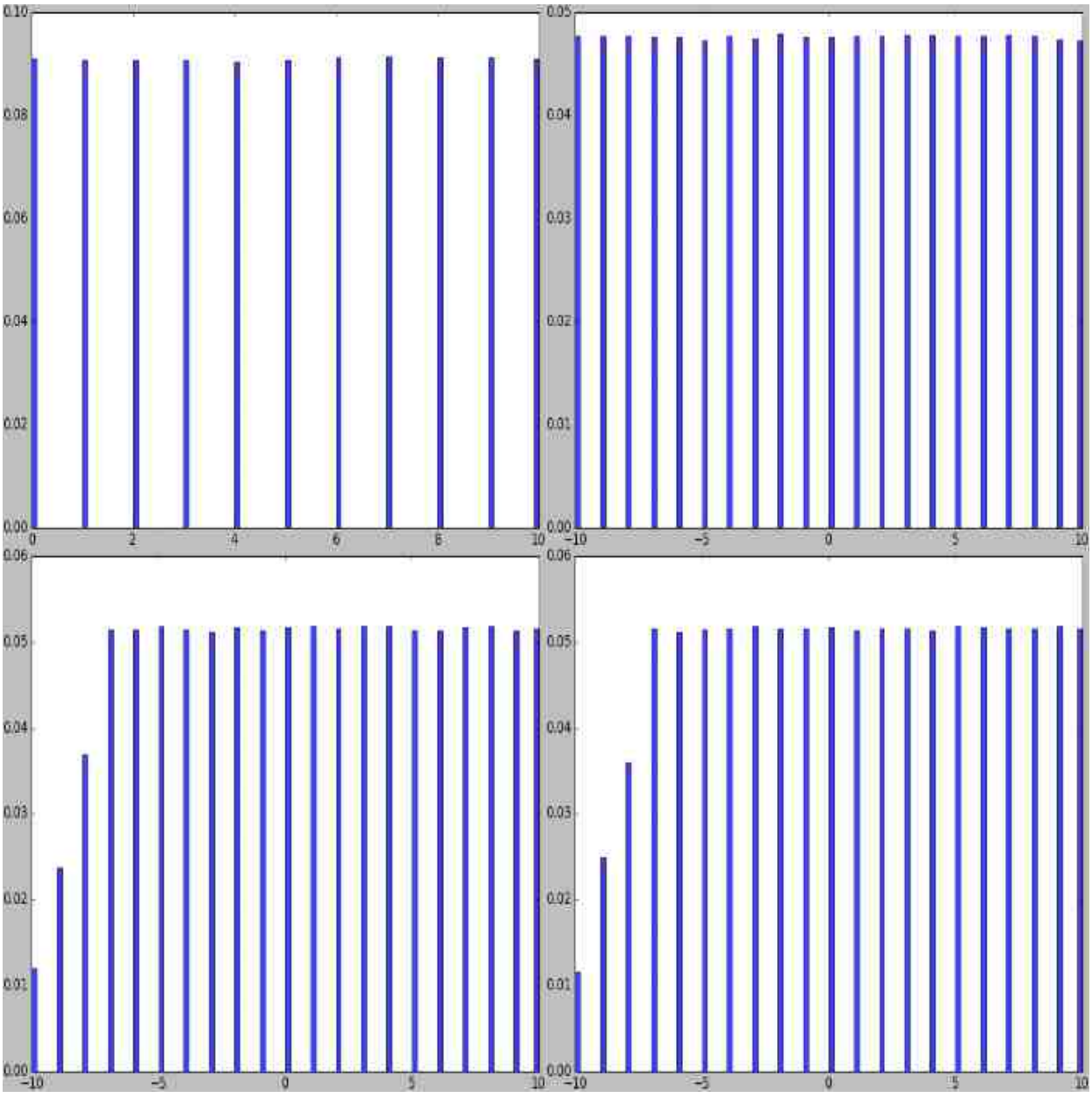


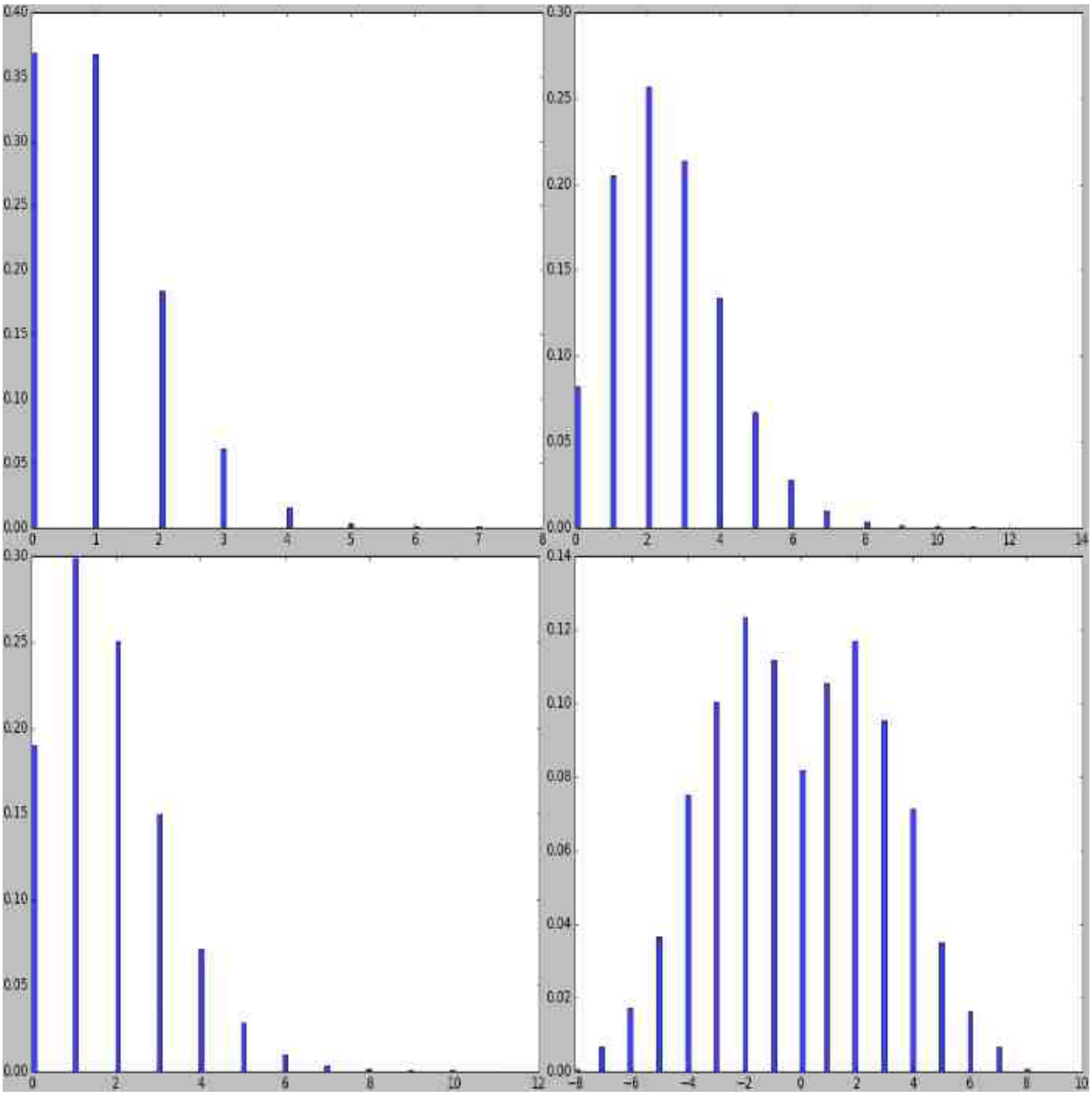
- *DiscreteUniform(a, b)*: The discrete uniform distribution in the range *[a..b]*. Example:

```
from imgaug import parameters as iap
params = [
    iap.DiscreteUniform(0, 10),
    iap.DiscreteUniform(-10, 10),
    iap.DiscreteUniform([-10, -9, -8, -7], 10),
    iap.DiscreteUniform((-10, -7), 10)
]
iap.show_distributions_grid(params)
```

- *Poisson(lam)*: Poisson distribution with shape *lam*. Generates no negative values. Example:

```
from imgaug import parameters as iap
params = [
    iap.Poisson(1),
    iap.Poisson(2.5),
    iap.Poisson((1, 2.5)),
    iap.RandomSign(iap.Poisson(2.5))
]
iap.show_distributions_grid(params)
```



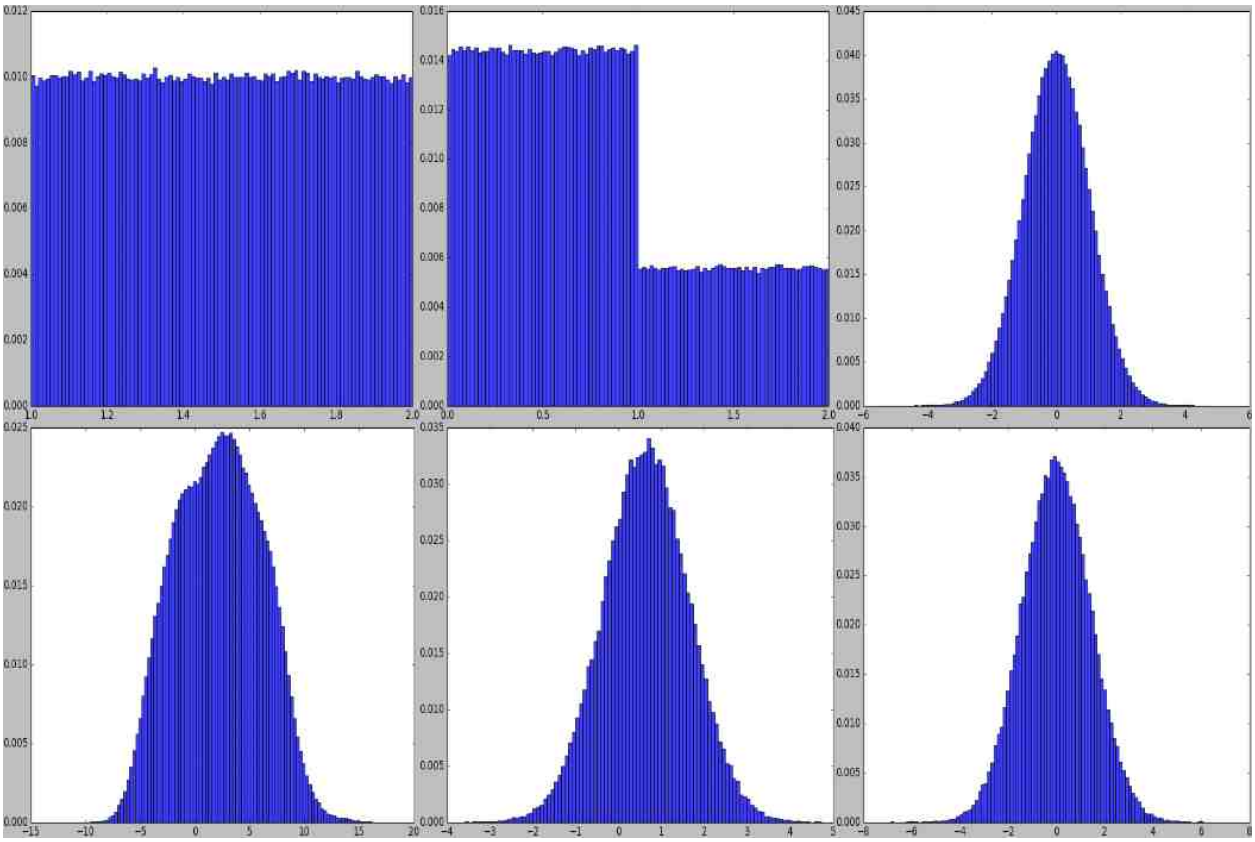


7.4 Arithmetic

The library supports arithmetic operations on stochastic parameters. This allows to modify values sampled from distributions or combine several distributions with each other.

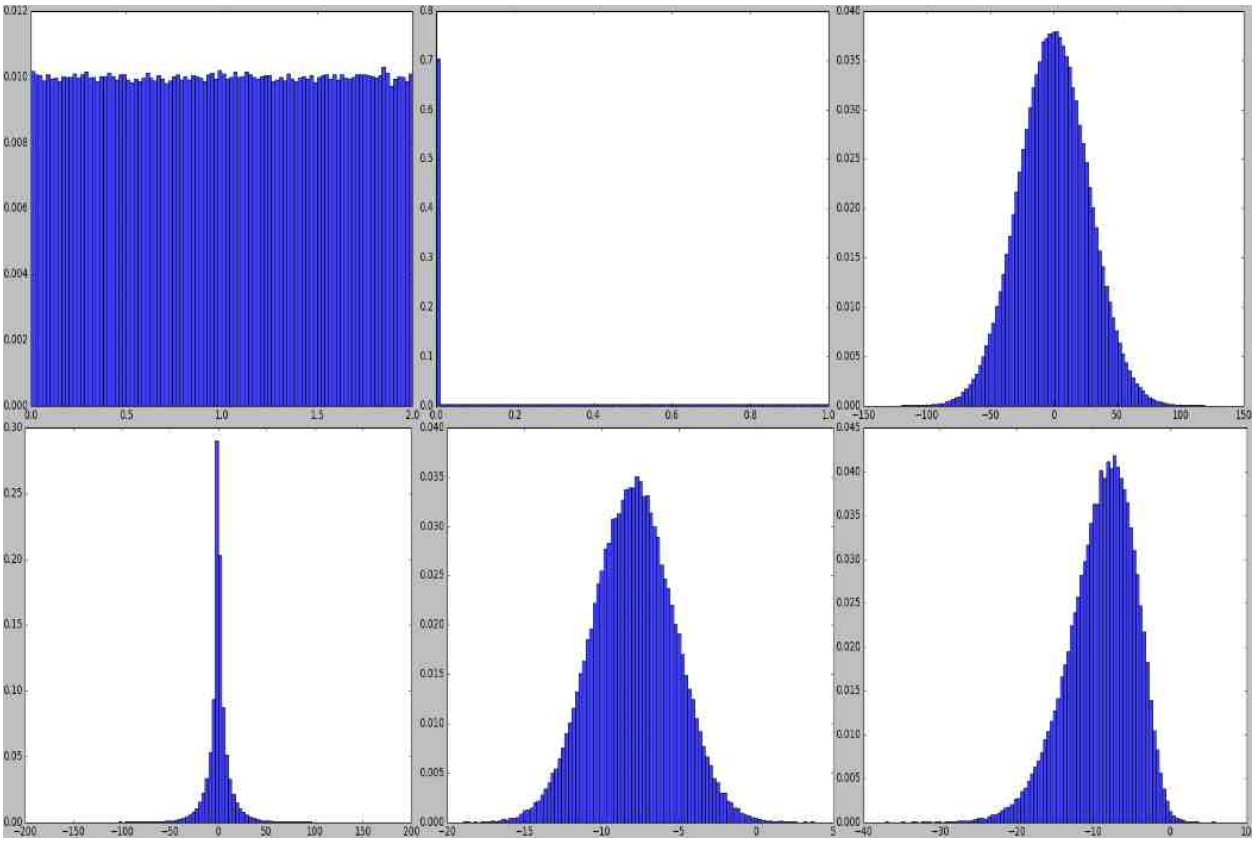
- *Add(param, val, elementwise)*: Add *val* to the values sampled from *param*. The shortcut is +, e.g. *Uniform(...) + 1*. *val* can be a stochastic parameter itself. Usually, only one value is sampled from *val* per sampling run and added to all samples generated by *param*. Alternatively, *elementwise* can be set to *True* in order to generate as many samples from *val* as from *param* and add them elementwise. Note that *Add* merely adds to the results of *param* and does *not* combine probability density functions (see e.g. example image 3 and 4). Example:

```
from imgaug import parameters as iap
params = [
    iap.Uniform(0, 1) + 1, # identical to: Add(Uniform(0, 1), 1)
    iap.Add(iap.Uniform(0, 1), iap.Choice([0, 1], p=[0.7, 0.3])),
    iap.Normal(0, 1) + iap.Uniform(-5.5, -5) + iap.Uniform(5, 5.5),
    iap.Normal(0, 1) + iap.Uniform(-7, 5) + iap.Poisson(3),
    iap.Add(iap.Normal(-3, 1), iap.Normal(3, 1)),
    iap.Add(iap.Normal(-3, 1), iap.Normal(3, 1), elementwise=True)
]
iap.show_distributions_grid(
    params,
    rows=2,
    sample_sizes=[ # (iterations, samples per iteration)
        (1000, 1000), (1000, 1000), (1000, 1000),
        (1000, 1000), (1, 100000), (1, 100000)
    ]
)
```



- *Subtract(param, val, elementwise)*: Same as *Add*, but subtracts *val* from the results of *param*. The shortcut is -, e.g. *Uniform(...) - 1*.
- *Multiply(param, val, elementwise)*: Same as *Add*, but multiplies *val* with the results of *param*. The shortcut is *, e.g. *Uniform(...) * 2*. Example:

```
from imgaug import parameters as iap
params = [
    iap.Uniform(0, 1) * 2, # identical to: Multiply(Uniform(0, 1), 2)
    iap.Multiply(iap.Uniform(0, 1), iap.Choice([0, 1], p=[0.7, 0.3])),
    (iap.Normal(0, 1) * iap.Uniform(-5.5, -5)) * iap.Uniform(5, 5.5),
    (iap.Normal(0, 1) * iap.Uniform(-7, 5)) * iap.Poisson(3),
    iap.Multiply(iap.Normal(-3, 1), iap.Normal(3, 1)),
    iap.Multiply(iap.Normal(-3, 1), iap.Normal(3, 1), elementwise=True)
]
iap.show_distributions_grid(
    params,
    rows=2,
    sample_sizes=[ # (iterations, samples per iteration)
        (1000, 1000), (1000, 1000), (1000, 1000),
        (1000, 1000), (1, 100000), (1, 100000)
    ]
)
```



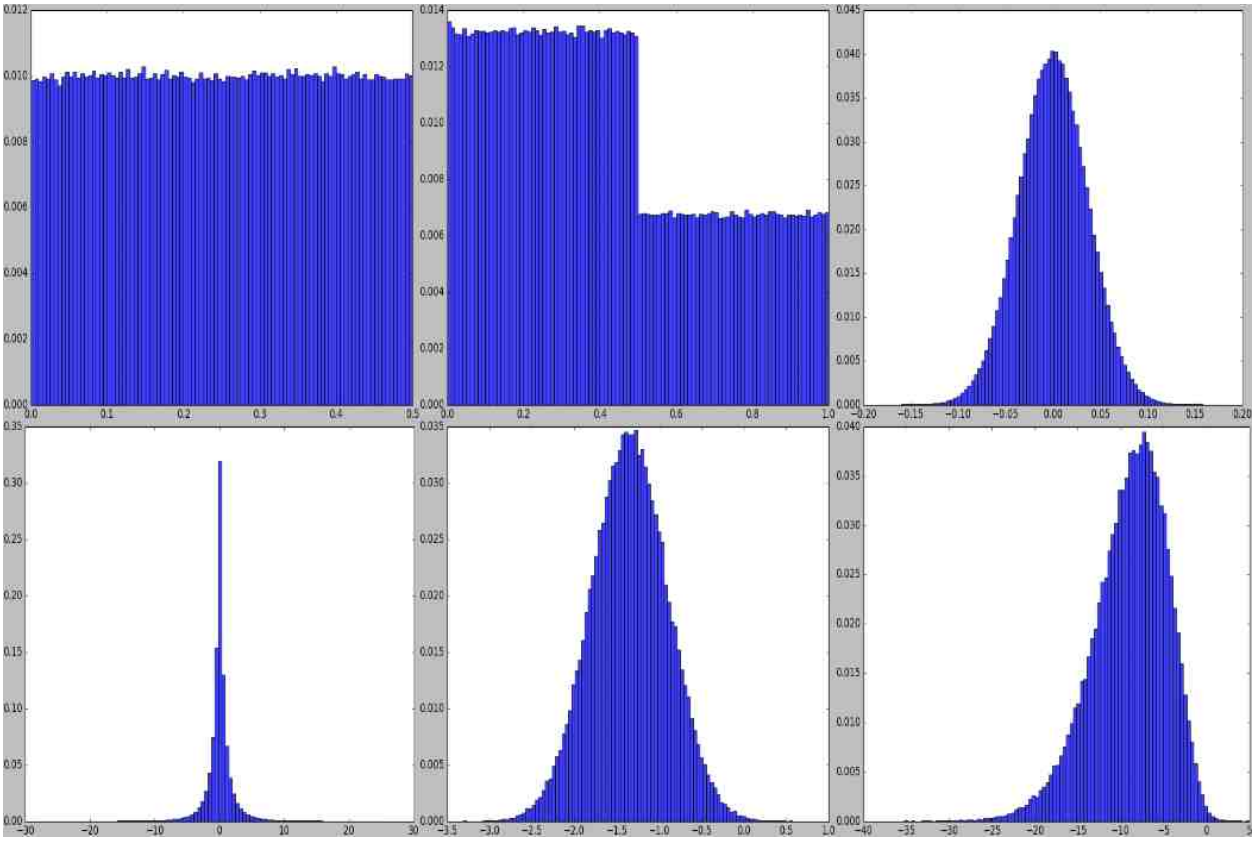
- *Divide(param, val, elementwise)*: Same as *Multiply*, but divides by *val*. The shortcut is */*, e.g. *Uniform(...) / 2*. Division by zero is automatically prevented (zeros are replaced by ones). Example:

```
from imgaug import parameters as iap
```

(continues on next page)

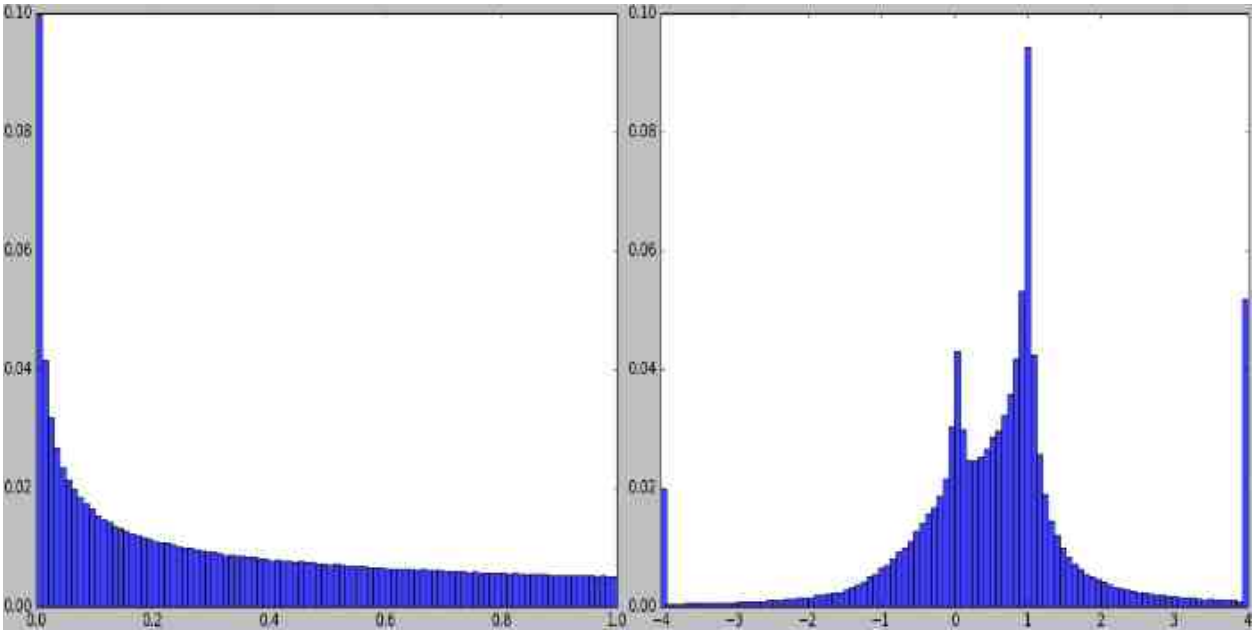
(continued from previous page)

```
params = [
    iap.Uniform(0, 1) / 2, # identical to: Divide(Uniform(0, 1), 2)
    iap.Divide(iap.Uniform(0, 1), iap.Choice([0, 2], p=[0.7, 0.3])),
    (iap.Normal(0, 1) / iap.Uniform(-5.5, -5)) / iap.Uniform(5, 5.5),
    (iap.Normal(0, 1) * iap.Uniform(-7, 5)) / iap.Poisson(3),
    iap.Divide(iap.Normal(-3, 1), iap.Normal(3, 1)),
    iap.Divide(iap.Normal(-3, 1), iap.Normal(3, 1), elementwise=True)
]
iap.show_distributions_grid(
    params,
    rows=2,
    sample_sizes=[ # (iterations, samples per iteration)
        (1000, 1000), (1000, 1000), (1000, 1000),
        (1000, 1000), (1, 100000), (1, 100000)
    ]
)
```



- *Power(param, val, elementwise)*: Same as *Add*, but raises sampled values to the exponent *val*. The shortcut is **. Example:

```
from imgaug import parameters as iap
params = [
    iap.Uniform(0, 1) ** 2, # identical to: Power(Uniform(0, 1), 2)
    iap.Clip(iap.Uniform(-1, 1) ** iap.Normal(0, 1), -4, 4)
]
iap.show_distributions_grid(params, rows=1)
```



7.5 Special Parameters

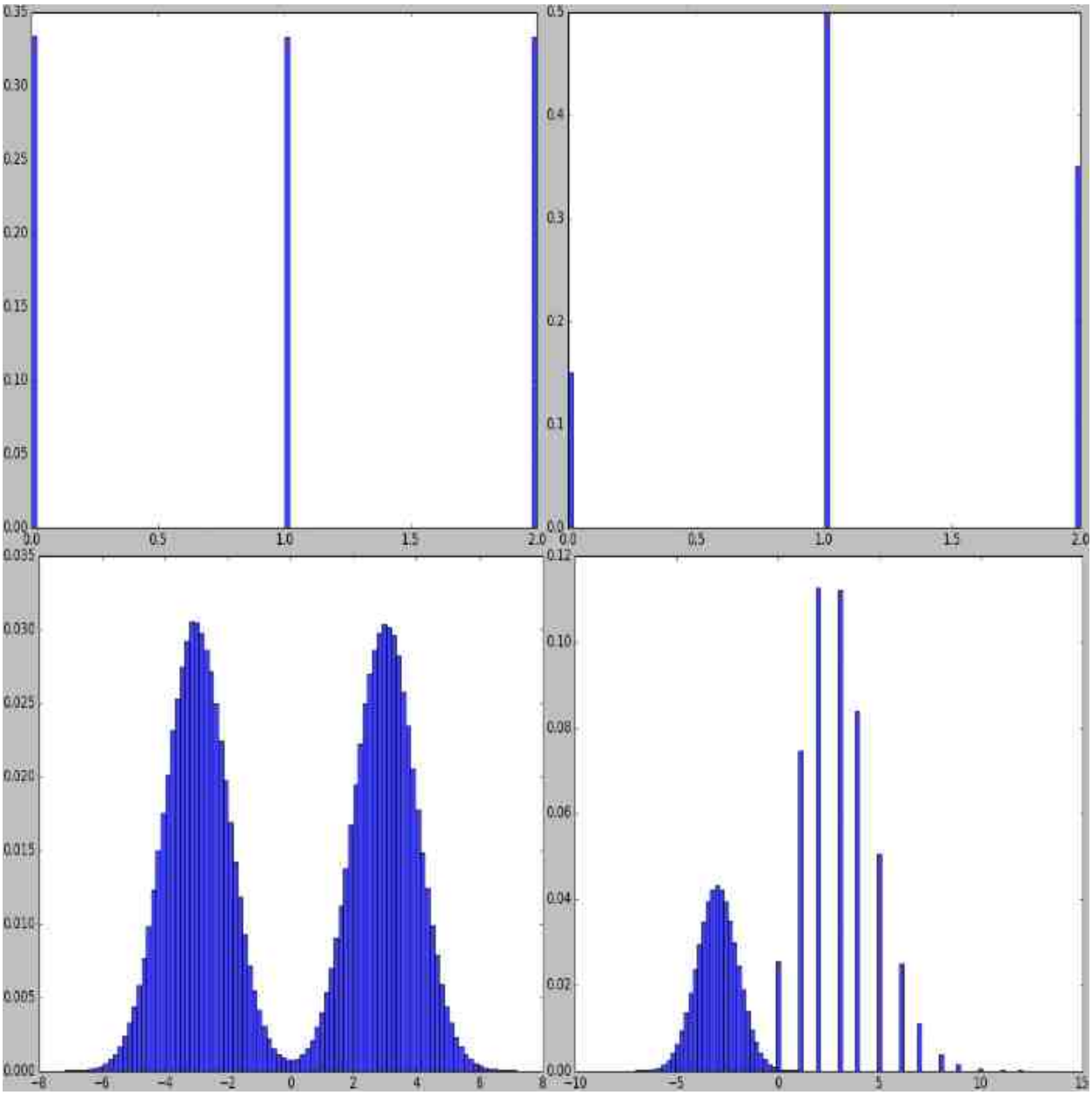
- *Deterministic(v)*: A constant. Upon sampling, this always returns *v*.
- *Choice(values, replace=True, p=None)*: Upon sampling, this parameter picks randomly elements from a list *values*. If *replace* is set to *True* (default), the picking happens with replacement. By default, all elements have the same probability of being picked. This can be modified using *p*. Note that *values* may also contain strings and other stochastic parameters. In the latter case, each picked parameter will be replaced by a sample from that parameter. This allows merging of probability mass functions, but is a rather slow process. All elements in *values* should have the same datatype (except for stochastic parameters). Example:

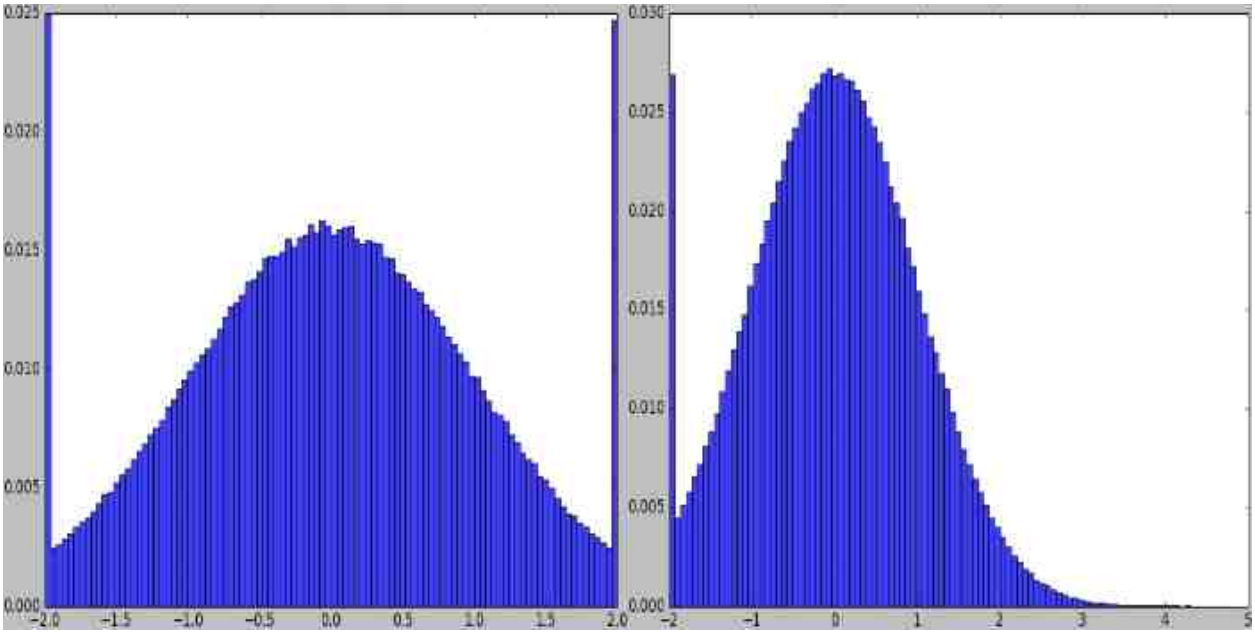
```
from imgaug import parameters as iap
params = [
    iap.Choice([0, 1, 2]),
    iap.Choice([0, 1, 2], p=[0.15, 0.5, 0.35]),
    iap.Choice([iap.Normal(-3, 1), iap.Normal(3, 1)]),
    iap.Choice([iap.Normal(-3, 1), iap.Poisson(3)])
]
iap.show_distributions_grid(params)
```

- *Clip(param, minval=None, maxval=None)*: Clips the values sampled from *param* to the range *[minval, maxval]*. *minval* and *maxval* may be *None*. In that case, only minimum or maximum clipping is applied (depending on what is *None*). Example:

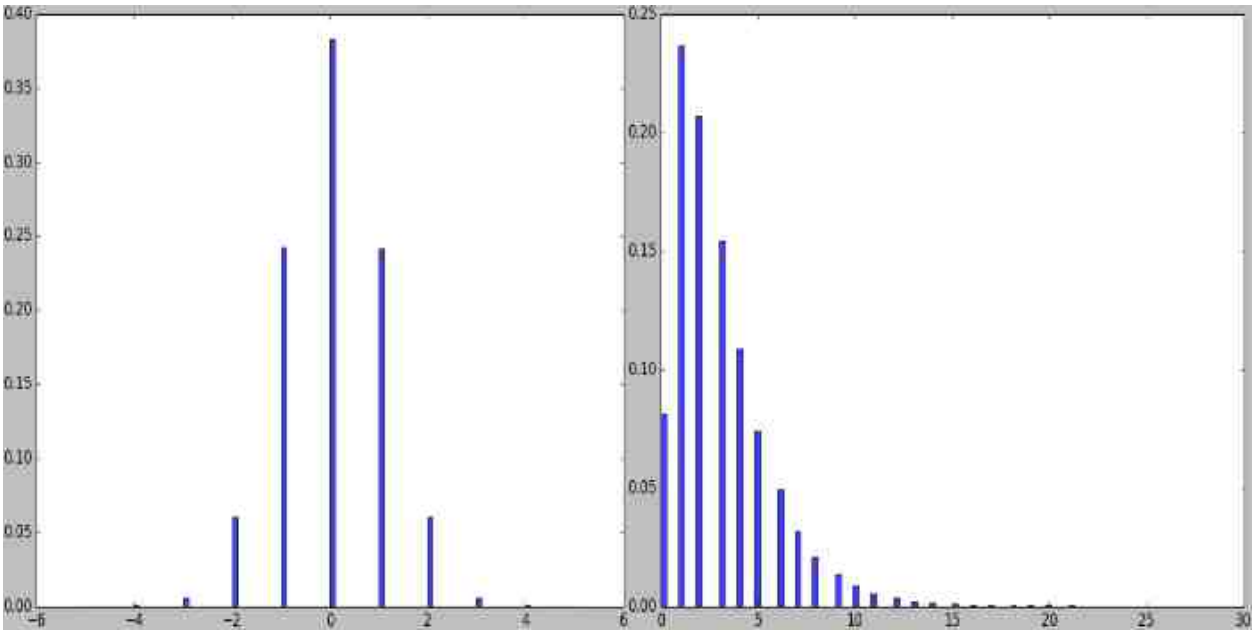
```
from imgaug import parameters as iap
params = [
    iap.Clip(iap.Normal(0, 1), -2, 2),
    iap.Clip(iap.Normal(0, 1), -2, None)
]
iap.show_distributions_grid(params, rows=1)
```

- *Discretize(param)*: Converts a continuous parameter *param* into a discrete one (using rounding). Discrete parameters are not changed. Example:





```
from imgaug import parameters as iap
params = [
    iap.Discretize(iap.Normal(0, 1)),
    iap.Discretize(iap.ChiSquare(3))
]
iap.show_distributions_grid(params, rows=1)
```



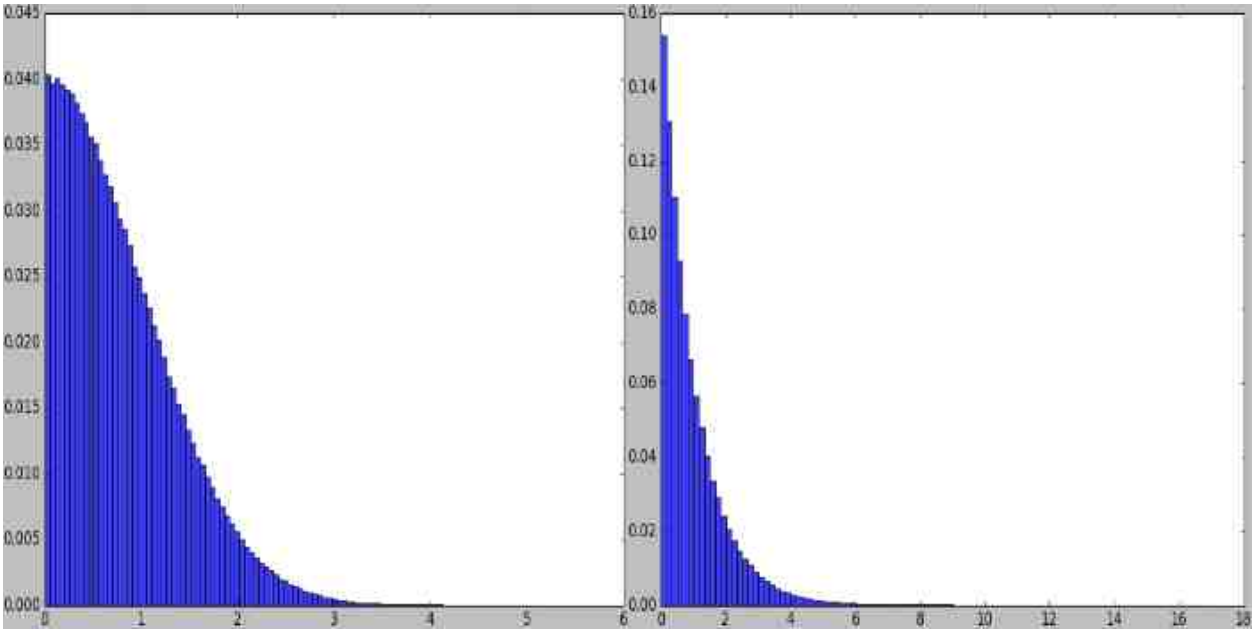
- *Absolute(param)*: Applies an absolute function to each value sampled from *param*, turning them to positive ones. Example:

```
from imgaug import parameters as iap
```

(continues on next page)

(continued from previous page)

```
params = [
    iap.Absolute(iap.Normal(0, 1)),
    iap.Absolute(iap.Laplace(0, 1))
]
iap.show_distributions_grid(params, rows=1)
```



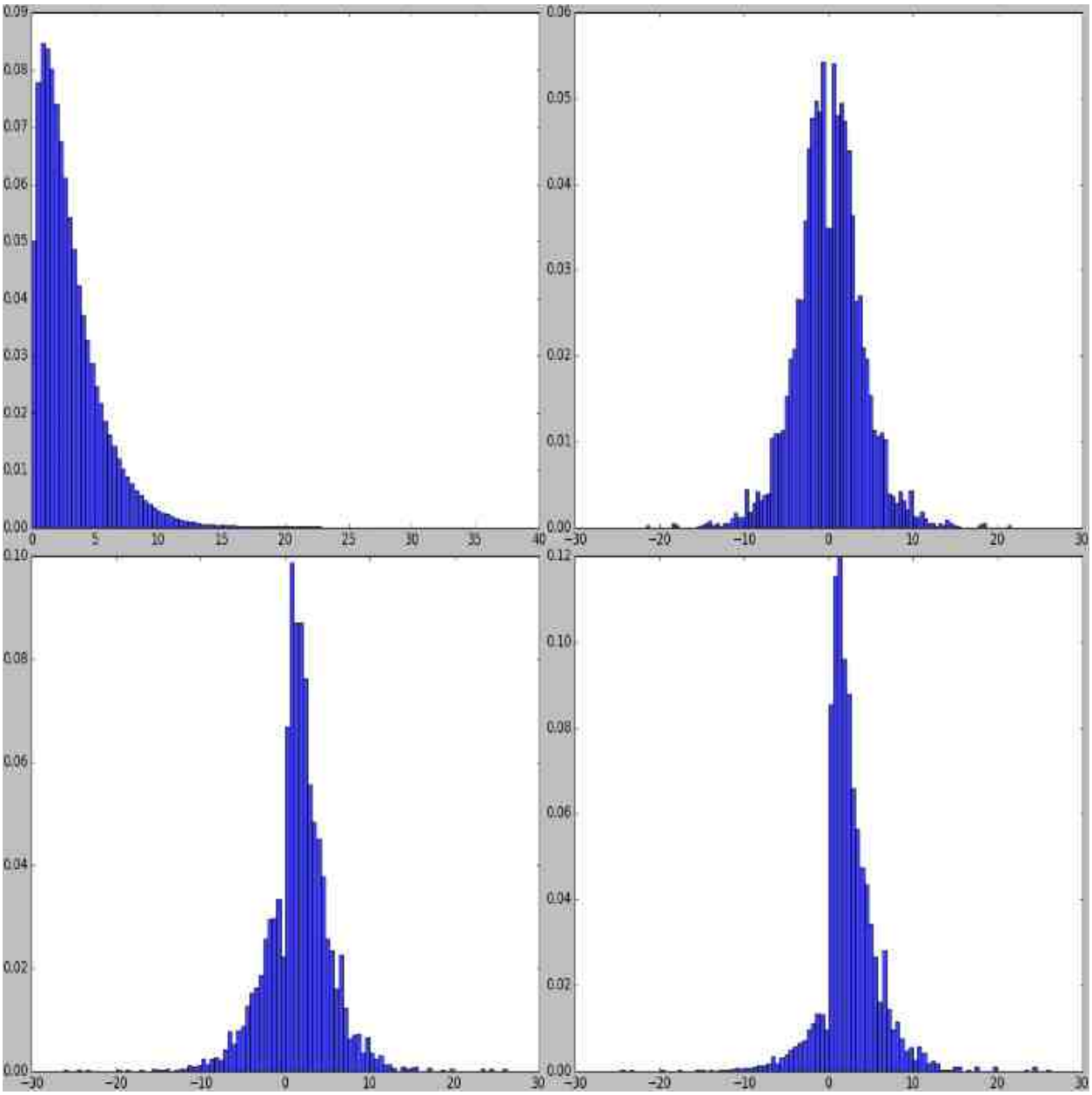
- *RandomSign(param, p_positive=0.5)*: Randomly flips the signs of values sampled from *param*. Optionally, the probability of flipping a value's sign towards positive can be set. Example:

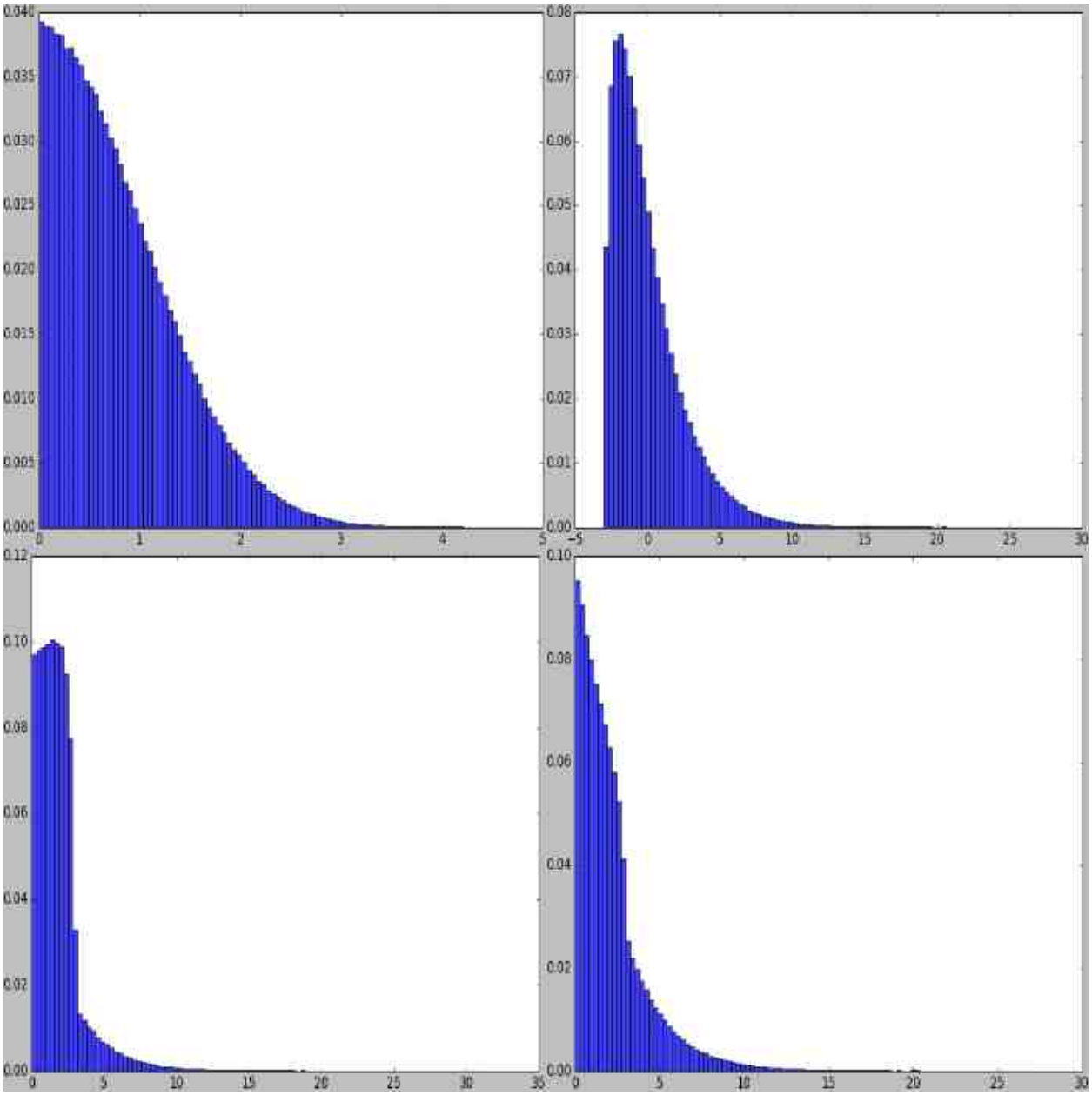
```
from imgaug import parameters as iap
params = [
    iap.ChiSquare(3),
    iap.RandomSign(iap.ChiSquare(3)),
    iap.RandomSign(iap.ChiSquare(3), p_positive=0.75),
    iap.RandomSign(iap.ChiSquare(3), p_positive=0.9)
]
iap.show_distributions_grid(params)
```

- *ForceSign(param, positive, mode="invert", reroll_count_max=2)*: Converts all values sampled from *param* to positive or negative ones. Signs of positive/negative values may simply be flipped (*mode="invert"*) or re-sampled from param (mode="reroll"). When rerolling, the number of iterations is limited to *reroll_count_max* (afterwards mode="invert" is used). Example:

```
from imgaug import parameters as iap
params = [
    iap.ForceSign(iap.Normal(0, 1), positive=True),
    iap.ChiSquare(3) - 3.0,
    iap.ForceSign(iap.ChiSquare(3) - 3.0, positive=True, mode="invert"),
    iap.ForceSign(iap.ChiSquare(3) - 3.0, positive=True, mode="reroll")
]
iap.show_distributions_grid(params)
```

- *Positive(other_param, mode="invert", reroll_count_max=2)*: Shortcut for *ForceSign* with *positive=True*. E.g. *Positive(Normal(0, 1))* restricts a normal distribution to only positive values.





- *Negative(other_param, mode="invert", reroll_count_max=2)*: Shortcut for *ForceSign* with *positive=False*. E.g. *Negative(Normal(0, 1))* restricts a normal distribution to only negative values.
- *FromLowerResolution(other_param, size_percent=None, size_px=None, method="nearest", min_size=1)*: Intended for 2d-sampling processes, e.g. for masks. Samples these in a lower resolution space. E.g. instead of sampling a mask at 100x100, this allows to sample it at 10x10 and then upsample to 100x100. One advantage is, that this can be faster. Another possible use is, that the upsampling may result in large, correlated blobs (linear interpolation) or rectangles (nearest neighbour interpolation).

7.6 Noise Parameters

TODO

CHAPTER 8

Blending/Overlaying images

8.1 Introduction

Most augmenters in the library affect images in uniform ways per image. Sometimes one might not want that and instead desires more localized effects (e.g. change the color of some image regions, while keeping the others unchanged) or wants to keep a fraction of the old image (e.g. blur the image and mix in a bit of the unblurred image). Alpha-based augmenters are intended for these use cases. They either mix two images using a constant alpha factor or using a pixel-wise mask. Below image shows examples.

```
# First row
iaa.Alpha(
    (0.0, 1.0),
    first=iaa.MedianBlur(11),
    per_channel=True
)

# Second row
iaa.SimplexNoiseAlpha(
    first=iaa.EdgeDetect(1.0),
    per_channel=False
)

# Third row
iaa.SimplexNoiseAlpha(
    first=iaa.EdgeDetect(1.0),
    second=iaa.ContrastNormalization((0.5, 2.0)),
    per_channel=0.5
)

# Forth row
iaa.FrequencyNoiseAlpha(
    first=iaa.Affine(
        rotate=(-10, 10),
        translate_px={"x": (-4, 4), "y": (-4, 4)}
```

(continues on next page)

(continued from previous page)

```
    ),
    second=iaa.AddToHueAndSaturation((-40, 40)),
    per_channel=0.5
)

# Fifth row
iaa.SimplexNoiseAlpha(
    first=iaa.SimplexNoiseAlpha(
        first=iaa.EdgeDetect(1.0),
        second=iaa.ContrastNormalization((0.5, 2.0)),
        per_channel=True
    ),
    second=iaa.FrequencyNoiseAlpha(
        exponent=(-2.5, -1.0),
        first=iaa.Affine(
            rotate=(-10, 10),
            translate_px={"x": (-4, 4), "y": (-4, 4)}
        ),
        second=iaa.AddToHueAndSaturation((-40, 40)),
        per_channel=True
    ),
    per_channel=True,
    aggregation_method="max",
    sigmoid=False
)
```

8.2 Constant Alpha

The augmenter *Alpha* allows to mix the results of two image sources using an alpha factor that is constant throughout the whole image, i.e. it follows roughly *I_blend = alpha * I_a + (1 - alpha) * I_b* per image, where *I_a* is the image from the first image source and *I_b* is the image from the second image source. Often, the first source will be an augmented version of the image and the second source will be the original image, leading to a blend of augmented and unaugmented image. The second image source can also be an augmented version of the image, leading to a blend of two distinct augmentation effects. Alpha is already built into some augmenters as a parameter, e.g. into *EdgeDetect*.

The below example code generates images that are a blend between *Sharpen* and *CoarseDropout*. Notice how the sharpening does not affect the black rectangles from dropout, as the two augmenters are both applied to the original images and merely blended.

```
import imgaug as ia
from imgaug import augmenters as iaa

ia.seed(1)

# Example batch of images.
# The array has shape (8, 128, 128, 3) and dtype uint8.
images = np.array(
    [ia.quokka(size=(128, 128)) for _ in range(8)],
    dtype=np.uint8
)

seq = iaa.Alpha(
    factor=(0.2, 0.8),
    first=iaa.Sharpen(1.0, lightness=2),
```

(continues on next page)

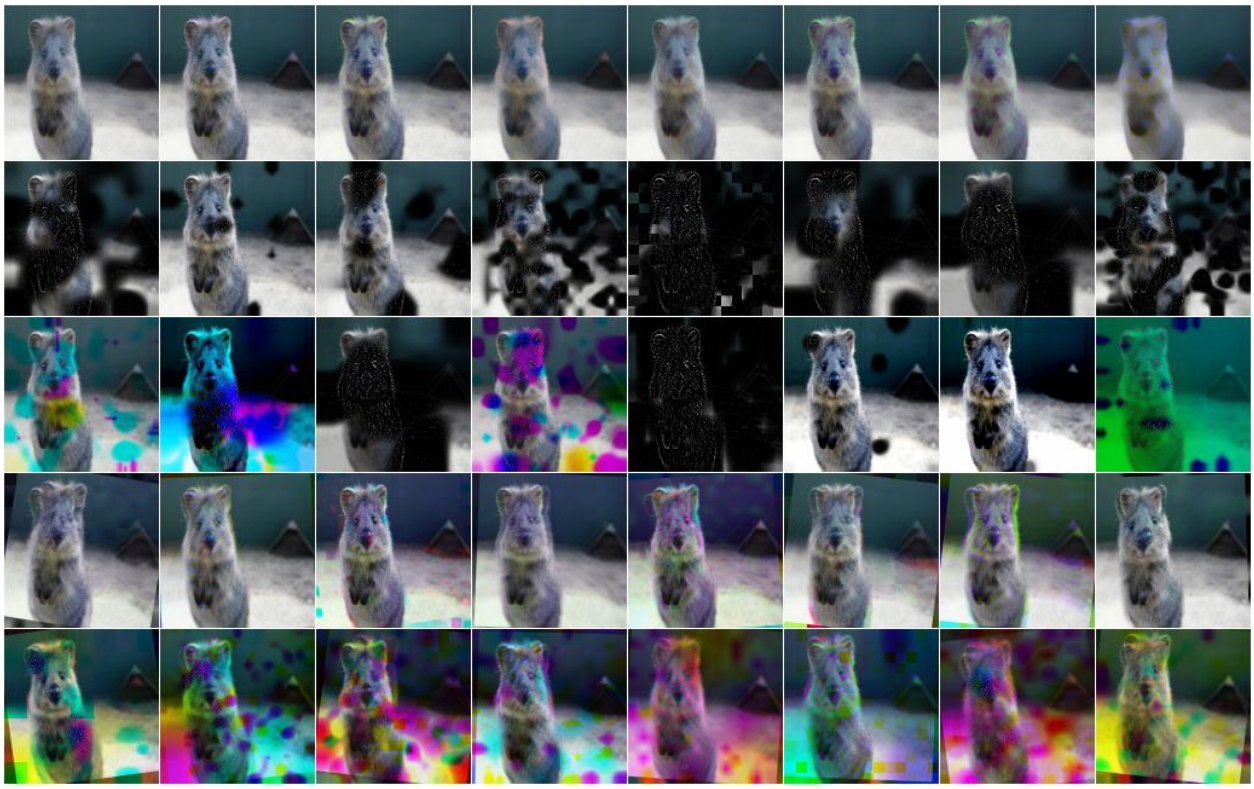


Fig. 1: Various effects of combining alpha-augmenters with other augmenters. First row shows *Alpha* with *MedianBlur*, second *SimplexNoiseAlpha* with *EdgeDetect*, third *SimplexNoiseAlpha* with *EdgeDetect* and *ContrastNormalization*, third shows *FrequencyNoiseAlpha* with *Affine* and *AddToHueAndSaturation* and forth row shows a mixture *SimplexNoiseAlpha* and *FrequencyNoiseAlpha*.

(continued from previous page)

```
    second=iaa.CoarseDropout(p=0.1, size_px=8)
)

images_aug = seq.augment_images(images)
```

Fig. 2: Mixing Sharpen and CoarseDropout via Alpha - not the same as executing them one after the other.

Similar to other augmenters, *Alpha* supports a *per_channel* mode, in which it samples overlay strengths for each channel independently. As a result, some channels may show more from the first (or second) image source than other channels. This can lead to visible color effects. The following example is the same as the one above, only *per_channel* was activated.

```
iaa.Alpha(..., per_channel=True)
```

Alpha can also be used with augmenters that change the position of pixels, leading to "ghost" images. (This should not be done when also augmenting keypoints, as their position becomes unclear.)

```
seq = iaa.Alpha(
    factor=(0.2, 0.8),
    first=iaa.Affine(rotate=(-20, 20)),
    per_channel=True
)
```

8.3 SimplexNoiseAlpha

Alpha uses a constant blending factor per image (or per channel). This limits the possibilities. Often a more localized factor is desired to create unusual patterns. *SimplexNoiseAlpha* is an augmenter that does that. It generates continuous masks following simplex noise and uses them to perform local blending. The following example shows a combination of *SimplexNoiseAlpha* and *Multiply* (with *per_channel=True*) that creates blobs of various colors in the image.

Fig. 3: Mixing Sharpen and CoarseDropout via Alpha and per_channel set to True.

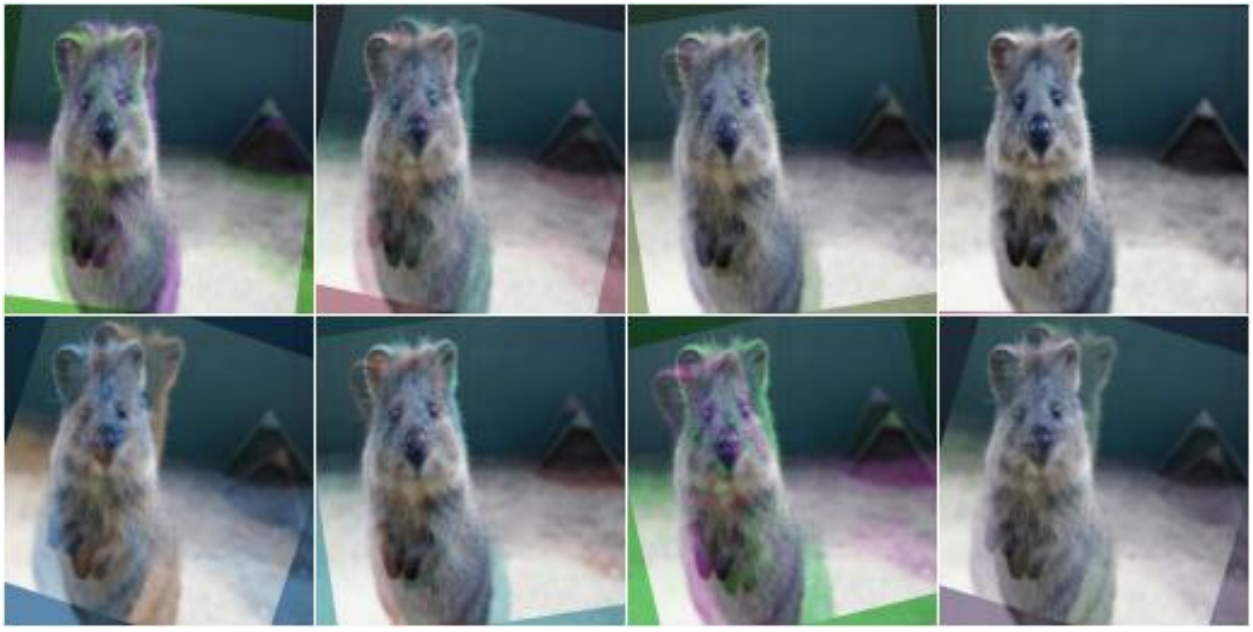


Fig. 4: Mixing original images with their rotated version. Some channels are more visibly rotated than others.

```
import imgaug as ia
from imgaug import augmenters as iaa

ia.seed(1)

# Example batch of images.
# The array has shape (8, 128, 128, 3) and dtype uint8.
images = np.array(
    [ia.quokka(size=(128, 128)) for _ in range(8)],
    dtype=np.uint8
)

seq = iaa.SimplexNoiseAlpha(
    first=iaa.Multiply(iap.Choice([0.5, 1.5]), per_channel=True)
)

images_aug = seq.augment_images(images)
```

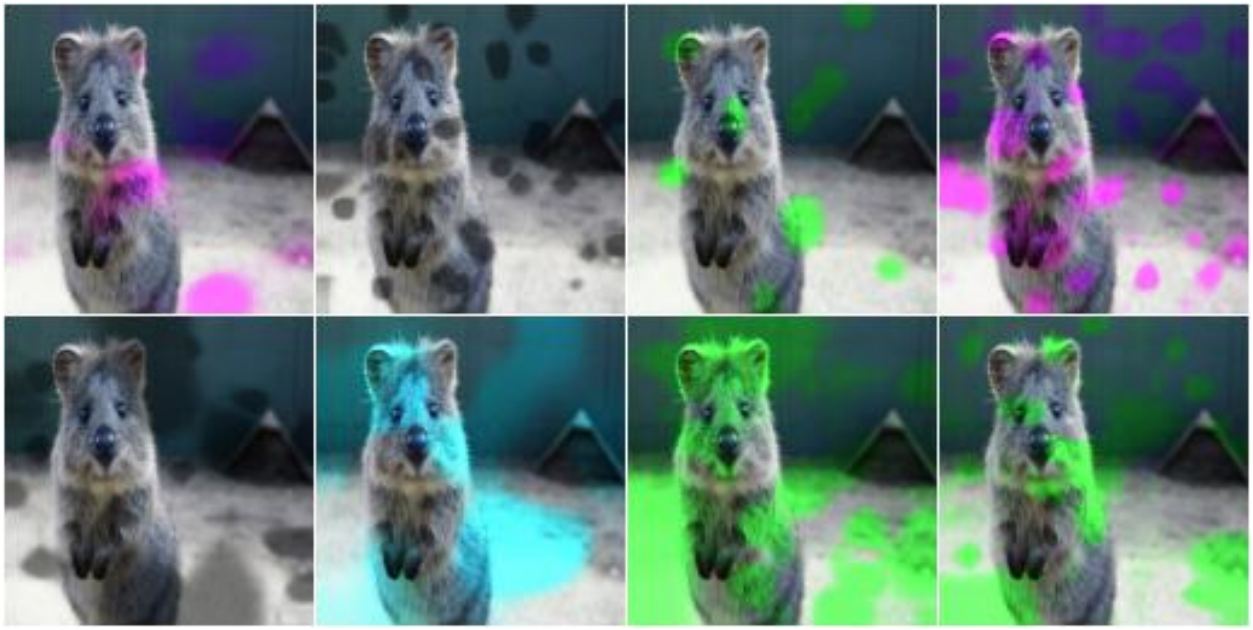


Fig. 5: Mixing original images with their versions modified by Multiply (with per_channel set to True). Simplex noise masks are used for the blending process, leading to blobby patterns.

SimplexNoiseAlpha also supports *per_channel=True*, leading to unique noise masks sampled per channel. The following example shows the combination of *SimplexNoiseAlpha* (with *per_channel=True*) and *EdgeDetect*. Even though *EdgeDetect* usually generates black and white images (white=edges, black=everything else), here the combination leads to strong color effects as the channel-wise noise masks only blend EdgeDetect's result for some channels.

```
seq = iaa.SimplexNoiseAlpha(
    first=iaa.EdgeDetect(1.0),
    per_channel=True
)
```

SimplexNoiseAlpha uses continuous noise masks (2d arrays with values in the range [0.0, 1.0]) to blend images. The below image shows examples of 64x64 noise masks generated by *SimplexNoiseAlpha* with default settings. Values close to 1.0 (white) indicate that pixel colors will be taken from the first image source, while 0.0 (black) values indicate

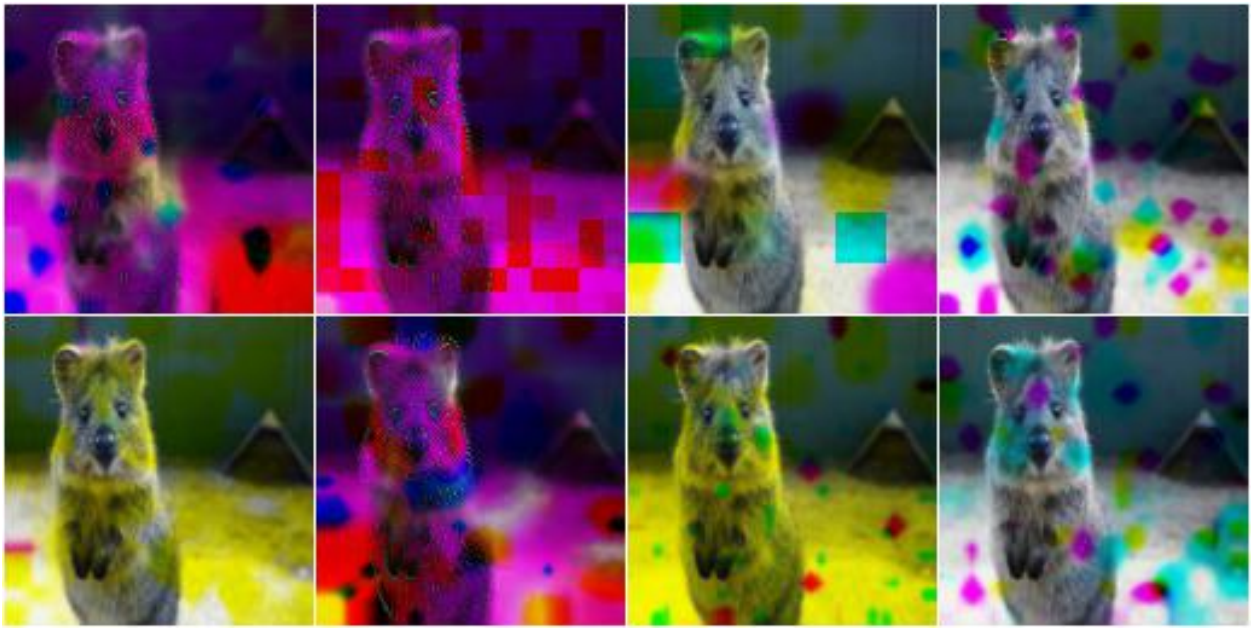


Fig. 6: Blending images via simplex noise can lead to unexpected but diverse patterns when per_channel is set to True. Here, a mixture of original images with *EdgeDetect(1.0)* is used.

that pixel colors will be taken from the second image source. (Often only one image source will be given in the form of augmenters and the second will fall back to the original images fed into *SimplexNoiseAlpha*.)

Fig. 7: Examples of noise masks generated by SimplexNoiseAlpha using default settings.

SimplexNoiseAlpha generates its noise masks in low resolution images and then upscales the masks to the size of the input images. During upscaling it usually uses nearest neighbour interpolation (*nearest*), linear interpolation (*linear*) or cubic interpolation (*cubic*). Nearest neighbour interpolation leads to noise maps with rectangular blobs. The below example shows noise maps generated when only using nearest neighbour interpolation.

```
seq = iaa.SimplexNoiseAlpha(
    ...,
    upscale_method="nearest"
)
```

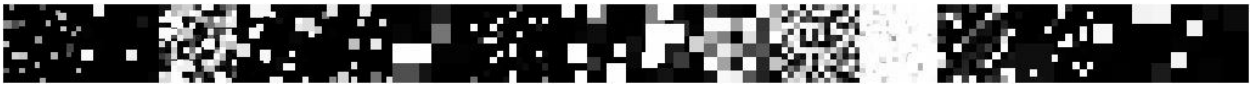


Fig. 8: Examples of noise masks generated by SimplexNoiseAlpha when restricting the upscaling method to *nearest*.

Similarly, the following example shows noise maps generated when only using linear interpolation.

```
seq = iaa.SimplexNoiseAlpha(
    ...,
    upscale_method="linear"
)
```

Fig. 9: Examples of noise masks generated by SimplexNoiseAlpha when restricting the upscaling method to *linear*.

8.4 FrequencyNoiseAlpha

FrequencyNoiseAlpha is mostly identical to *SimplexNoiseAlpha*. In contrast to *SimplexNoiseAlpha* it uses a different sampling process to generate the noise maps. The process is based on starting with random frequencies, weighting them with a random exponent and then transforming from frequency domain to spatial domain. When using a low exponent value this leads to large, smooth blobs. Slightly higher exponents lead to cloudy patterns. High exponent values lead to recurring, small patterns. The below example shows the usage of *FrequencyNoiseAlpha*.

```
import imgaug as ia
from imgaug import augmenters as iaa
from imgaug import parameters as iap

ia.seed(1)

# Example batch of images.
# The array has shape (8, 64, 64, 3) and dtype uint8.
images = np.array(
    [ia.quokka(size=(128, 128)) for _ in range(8)],
    dtype=np.uint8
)

seq = iaa.FrequencyNoiseAlpha(
    first=iaa.Multiply(iap.Choice([0.5, 1.5]), per_channel=True)
)

images_aug = seq.augment_images(images)
```

Similarly to simplex noise, *FrequencyNoiseAlpha* also supports *per_channel=True*, leading to different noise maps per image channel.

```
seq = iaa.FrequencyNoiseAlpha(
    first=iaa.EdgeDetect(1.0),
    per_channel=True
)
```

The below image shows random example noise masks generated by *FrequencyNoiseAlpha* with default settings.

The following image shows the effects of varying *exponent* between *-4.0* and *4.0*. To show these effects more clearly, a few features of *FrequencyNoiseAlpha* were deactivated (e.g. multiple iterations). In the code, *E* is the value of the exponent (e.g. *E=-2.0*).

```
seq = iaa.FrequencyNoiseAlpha(
    exponent=E,
    first=iaa.Multiply(iap.Choice([0.5, 1.5]), per_channel=True),
    size_px_max=32,
    upscale_method="linear",
    iterations=1,
    sigmoid=False
)
```

Similarly to *SimplexNoiseAlpha*, *FrequencyNoiseAlpha* also generates the noise masks as low resolution versions and

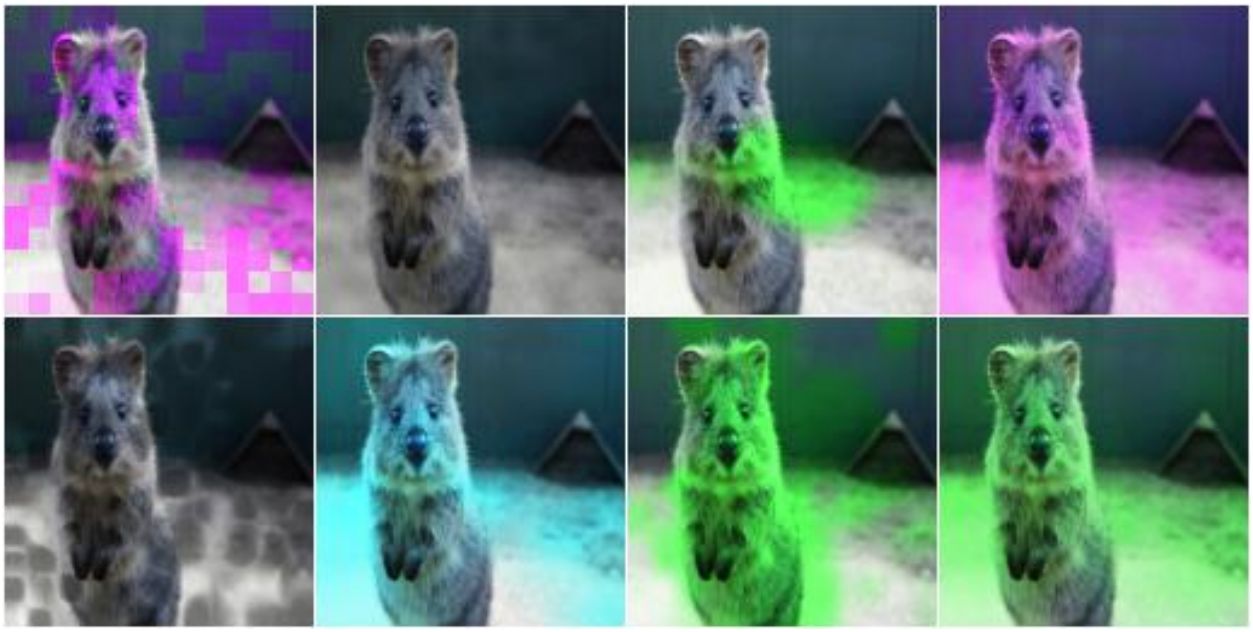


Fig. 10: Mixing original images with their versions modified by Multiply (with per_channel set to True). Simplex noise masks are used for the blending process, leading to blobby patterns.

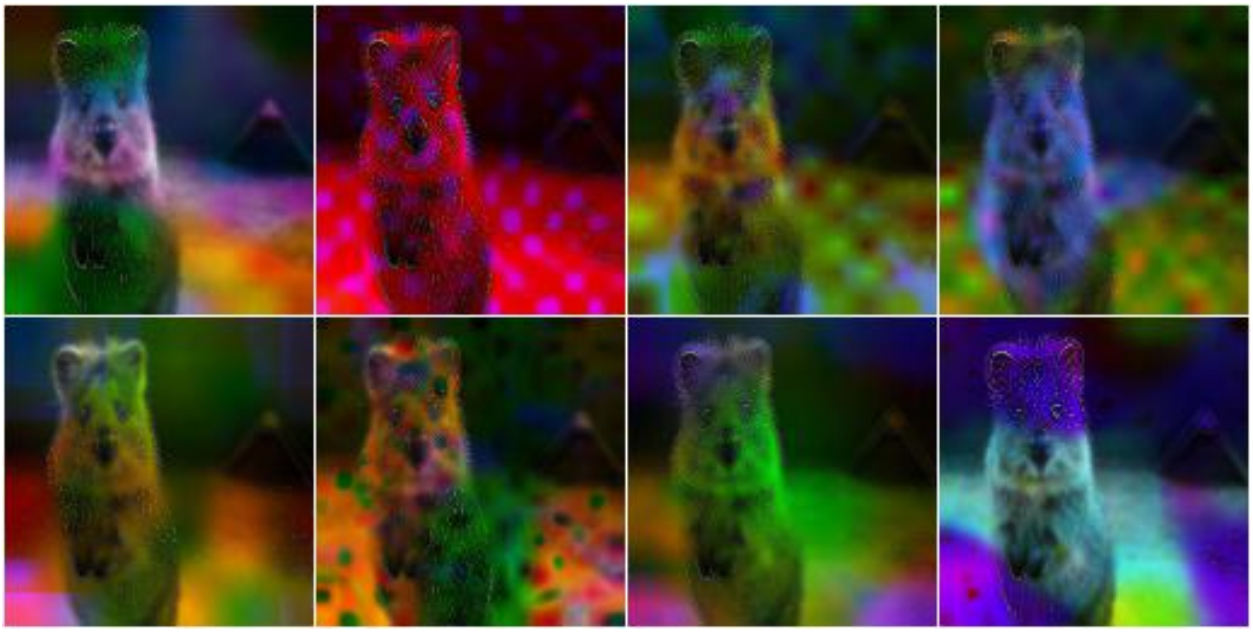


Fig. 11: Blending images via frequency noise can lead to unexpected but diverse patterns when per_channel is set to True. Here, a mixture of original images with *EdgeDetect(1.0)* is used.

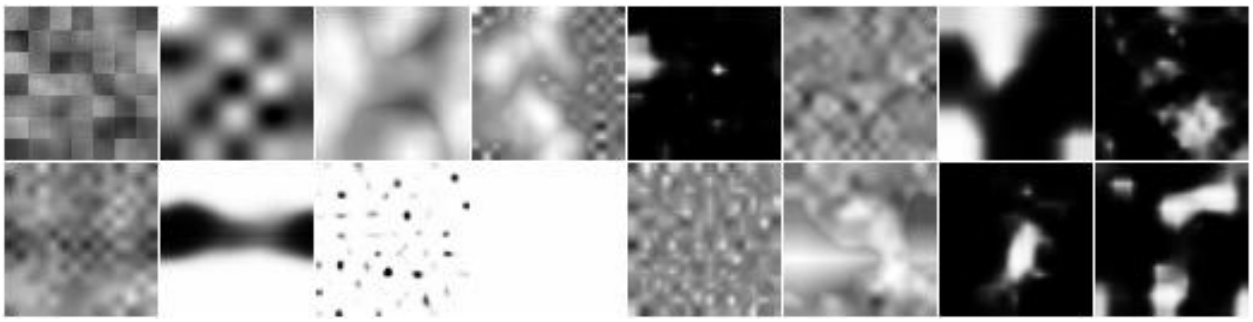


Fig. 12: Examples of noise masks generated by FrequencyNoiseAlpha using default settings.

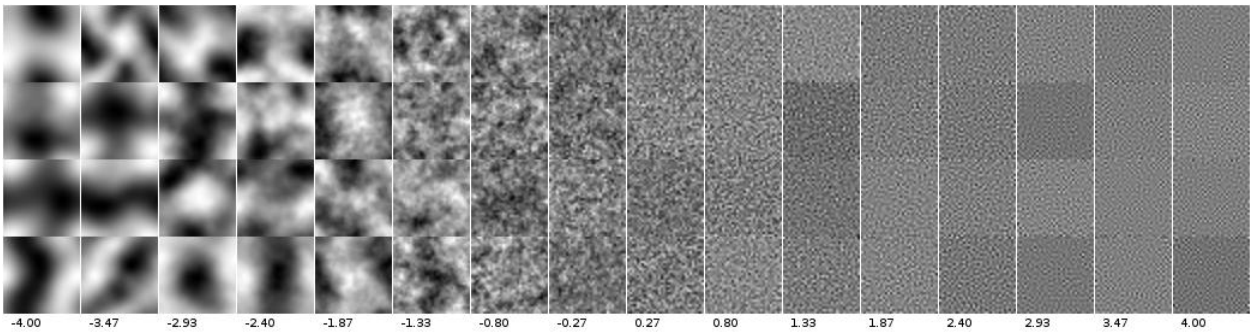


Fig. 13: Examples of noise masks generated by FrequencyNoiseAlpha using default settings with varying exponents.

then upscales them to the full image size. The following images show the usage of nearest neighbour interpolation (*upscale_method="nearest"*) and linear interpolation (*upscale_method="linear"*).

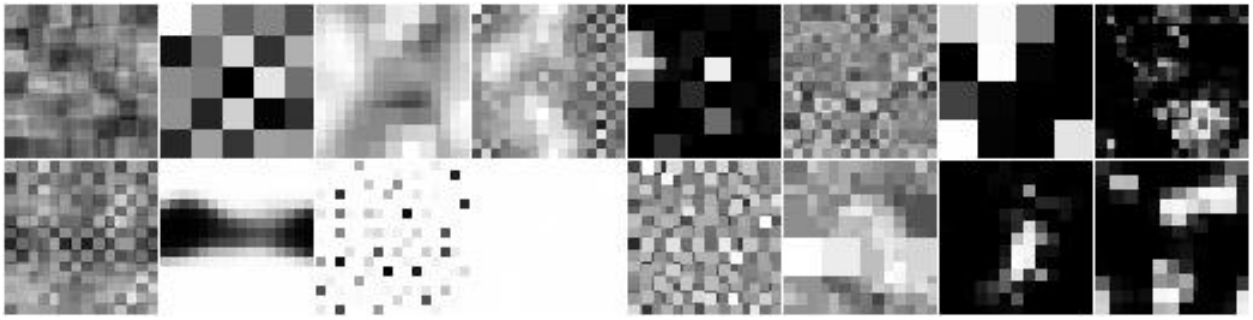


Fig. 14: Examples of noise masks generated by FrequencyNoiseAlpha when restricting the upscaling method to *nearest*.

8.5 IterativeNoiseAggregator

Both *SimplexNoiseAlpha* and *FrequencyNoiseAlpha* wrap around *IterativeNoiseAggregator*, a component to generate noise masks in multiple iterations. It has parameters for the number of iterations (1 to N) and for the aggregation methods, which controls how the noise masks from the different iterations are to be combined. Valid aggregation methods are *"min"*, *"avg"* and *"max"*, where *min* takes the minimum over all iteration's masks, *max* the maxmimum and *avg* the average. As a result, masks generated with method *min* tend to be close to 0.0 (mostly black values), those generated with *max* close to 1.0 and *avg* converges towards 0.5. (0.0 means that the results of the second image

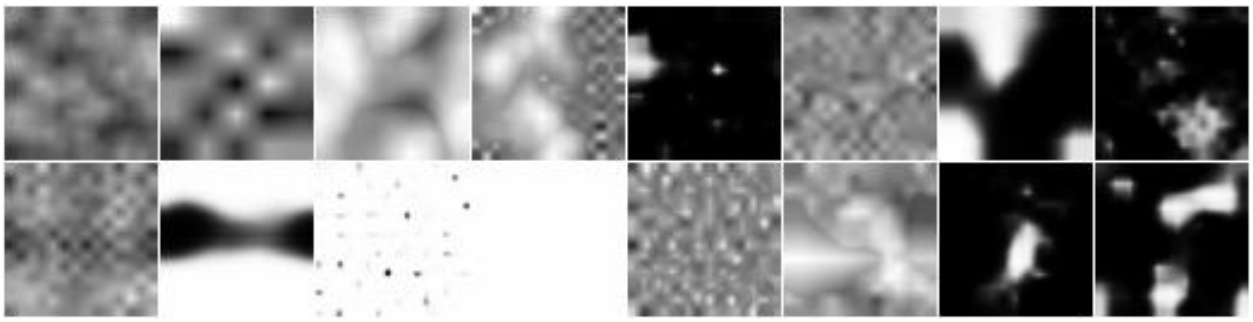


Fig. 15: Examples of noise masks generated by FrequencyNoiseAlpha when restricting the upscaling method to *linear*.

dominate the final image, so in many cases the original images before the augmenter). The following image shows the effects of changing the number of iterations when combining *FrequencyNoise* with *IterativeNoiseAggregator*.

```
# This is how the iterations would be changed for FrequencyNoiseAlpha.
# (Same for `SimplexNoiseAlpha`.)
seq = iaa.FrequencyNoiseAlpha(
    ...,
    iterations=N
)
```

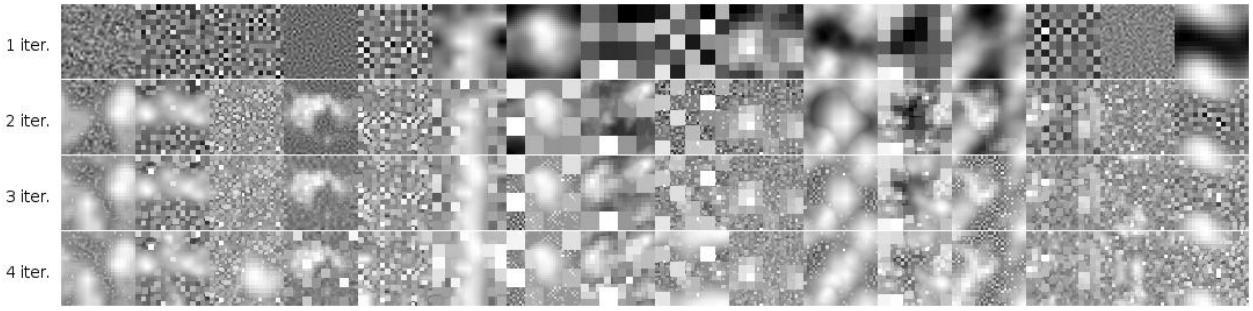


Fig. 16: Examples of varying the number of iterations in IterativeNoiseAggregator (here in combination with FrequencyNoise).

The following image shows the effects of changing the aggregation mode (with varying iterations).

```
# This is how the iterations and aggregation method would be changed for
# FrequencyNoiseAlpha. (Same for `SimplexNoiseAlpha`.)
seq = iaa.FrequencyNoiseAlpha(
    ...,
    iterations=N,
    aggregation_method=M
)
```

8.6 Sigmoid

Generated noise masks can often end up having many values around 0.5, especially when running *IterativeNoiseAggregator* with many iterations and aggregation method *avg*. This can be undesired. *Sigmoid* is a method to compensate that. It applies a sigmoid function to the noise masks, forcing the values to mostly lie close to 0.0 or 1.0 and only rarely in between. This can lead to blobs of values close to 1.0 ("use only colors from images coming from source

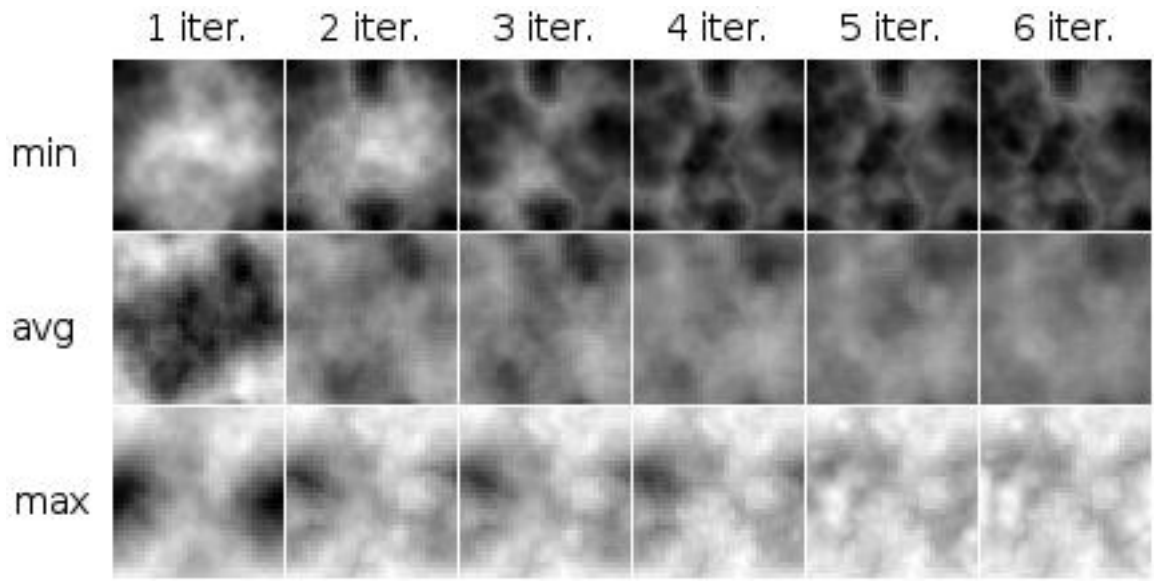


Fig. 17: Examples of varying the methods and iterations in IterativeNoiseAggregator (here in combination with FrequencyNoise).

A"), surrounded by blobs with values close to 0.0 ("use only colors from images coming from source B"). This is similar to taking *either* from one image source (per pixel) or the other, but usually not both. Sigmoid is integrated into both *SimplexNoiseAlpha* and *FrequencyNoiseAlpha*. It can be dynamically activated/deactivated and has a threshold parameter that controls how aggressive and pushes the noise values towards 1.0.

```
# This is how the Sigmoid would be activated/deactivated for
# FrequencyNoiseAlpha (same for SimplexNoiseAlpha). P is the probability
# of the Sigmoid being activated (can be True/False), T is the
# threshold (sane values are usually around -10 to +10, can be a
# tuple, e.g. sigmoid_thresh=(-10, 10), to indicate a uniform range).
seq = iaa.FrequencyNoiseAlpha(
    ...,
    sigmoid=P,
    sigmoid_thresh=T
)
```

The below image shows the effects of applying Sigmoid to noise masks generated by *FrequencyNoise*.

Fig. 18: Examples of noise maps without and with activated Sigmoid (noise maps here from FrequencyNoise).

The below image shows the effects of varying the sigmoid's threshold. Lower values place the threshold further to the "left" (lower x values), leading to more x-values being above the threshold values, leading to more 1.0s in the noise masks.

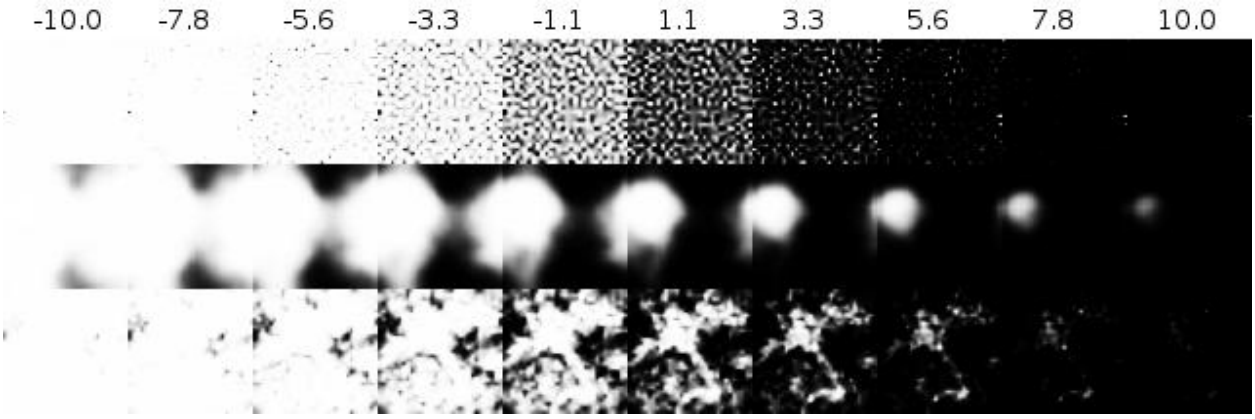


Fig. 19: Examples of varying the Sigmoid threshold from -10.0 to 10.0.

CHAPTER 9

Overview of Augmenters

9.1 augmenters.meta

9.1.1 Sequential

List augmenter that may contain other augmenters to apply in sequence or random order.

API link: *Sequential*

Example. Apply in predefined order:

```
import imgaug.augmenters as iaa
aug = iaa.Sequential([
    iaa.Affine(translate_px={"x":-40}),
    iaa.AdditiveGaussianNoise(scale=0.1*255)
])
```

Example. Apply in random order (note that the order is sampled once per batch and then the same for all images within the batch):

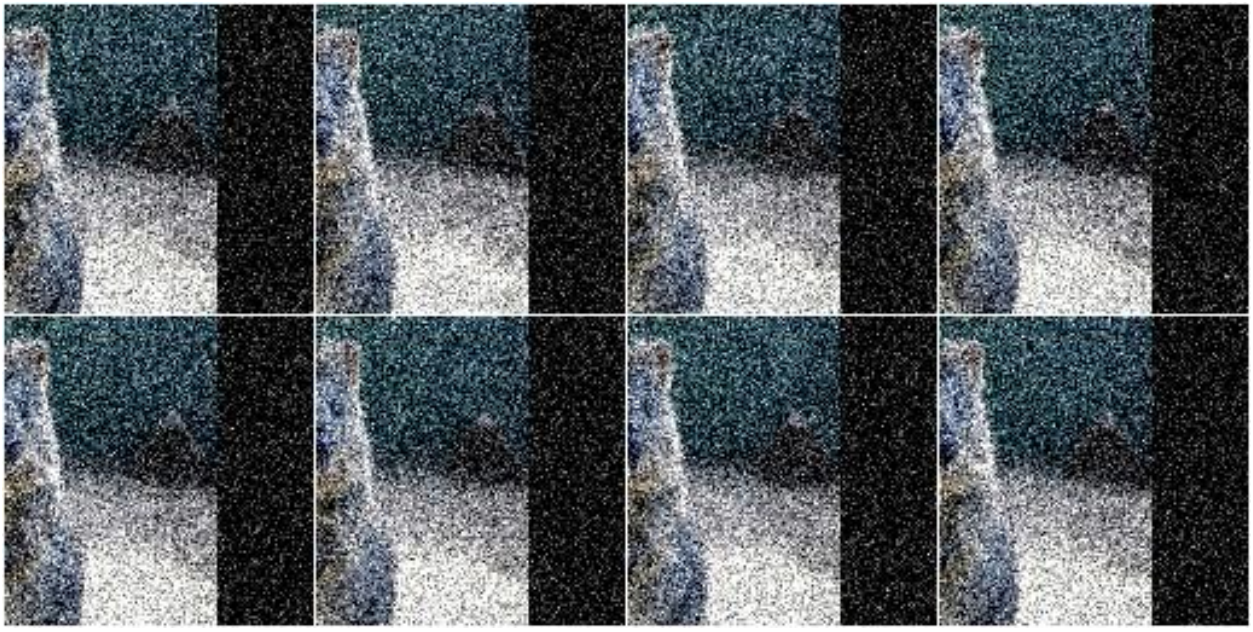
```
aug = iaa.Sequential([
      iaa.Affine(translate_px={"x":-40}),
      iaa.AdditiveGaussianNoise(scale=0.1*255)
], random_order=True)
```

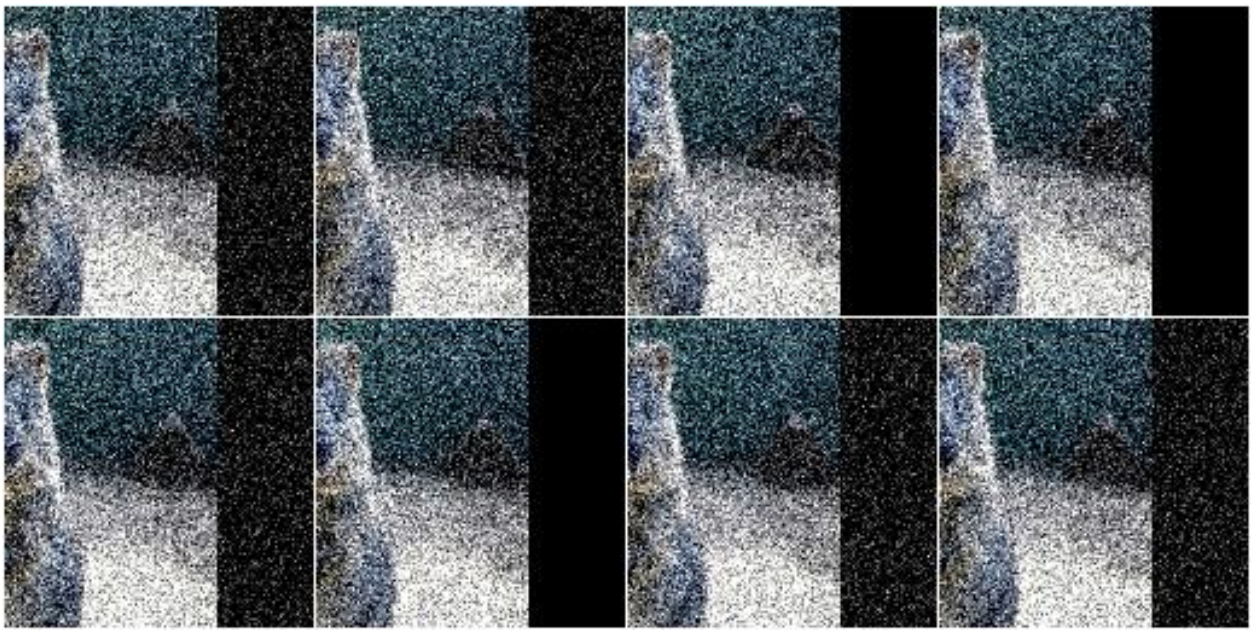
9.1.2 SomeOf

List augmenter that applies only some of its children to images.

API link: *SomeOf*

Example. Apply two of four given augmenters:





```
import imgaug.augmenters as iaa
aug = iaa.SomeOf(2, [
    iaa.Affine(rotate=45),
    iaa.AdditiveGaussianNoise(scale=0.2*255),
    iaa.Add(50, per_channel=True),
    iaa.Sharpen(alpha=0.5)
])
```

Example. Apply `0` to `<max>` given augmenters (where `<max>` is automatically replaced with the number of children):

```
aug = iaa.SomeOf((0, None), [
    iaa.Affine(rotate=45),
    iaa.AdditiveGaussianNoise(scale=0.2*255),
    iaa.Add(50, per_channel=True),
    iaa.Sharpen(alpha=0.5)
])
```

Example. Pick two of four given augmenters and apply them in random order:

```
aug = iaa.SomeOf(2, [
    iaa.Affine(rotate=45),
    iaa.AdditiveGaussianNoise(scale=0.2*255),
    iaa.Add(50, per_channel=True),
    iaa.Sharpen(alpha=0.5)
], random_order=True)
```

9.1.3 OneOf

Augmenter that always executes exactly one of its children.

API link: *OneOf()*

Example. Apply one of four augmenters to each image:

```
import imgaug.augmenters as iaa
aug = iaa.OneOf([
    iaa.Affine(rotate=45),
    iaa.AdditiveGaussianNoise(scale=0.2*255),
    iaa.Add(50, per_channel=True),
    iaa.Sharpen(alpha=0.5)
])
```

9.1.4 Sometimes

Augment only p percent of all images with one or more augmenters.

API link: *Sometimes*

Example. Apply gaussian blur to about 50% of all images:

```
import imgaug.augmenters as iaa
aug = iaa.Sometimes(0.5, iaa.GaussianBlur(sigma=2.0))
```

Example. Apply gaussian blur to about 50% of all images. Apply a mixture of affine rotations and sharpening to the other 50%.

```
aug = iaa.Sometimes(
      0.5,
      iaa.GaussianBlur(sigma=2.0),
      iaa.Sequential([iaa.Affine(rotate=45), iaa.Sharpen(alpha=1.0)])
  )
```

9.1.5 WithChannels

Apply child augmenters to specific channels.

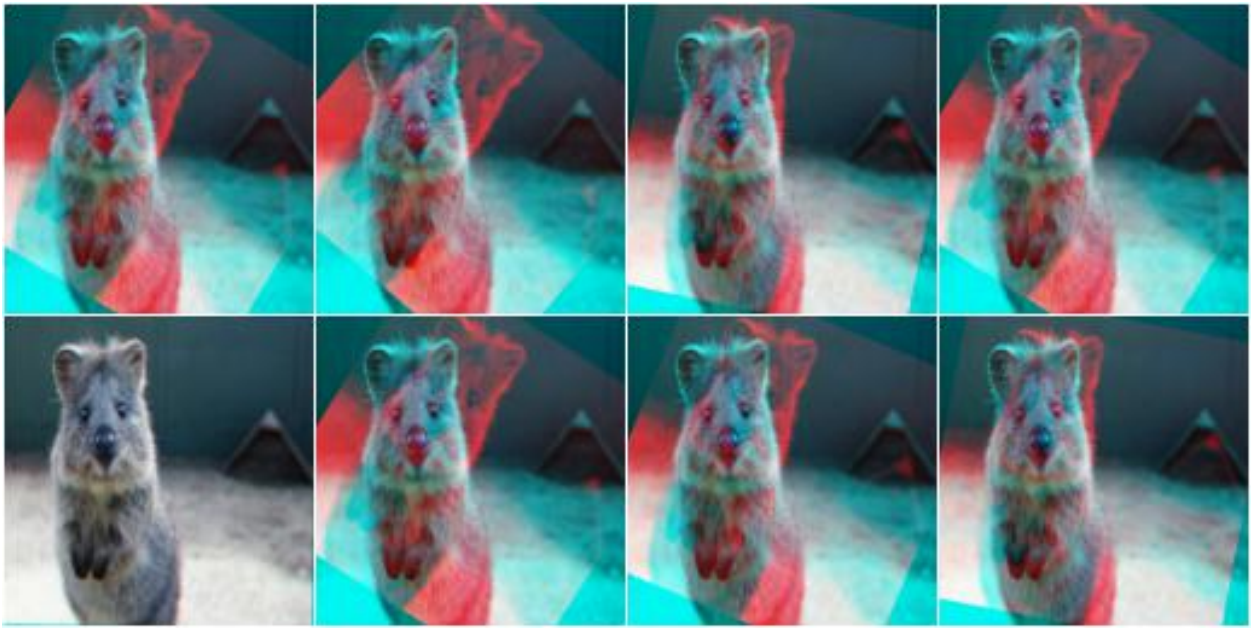
API link: *WithChannels*

Example. Increase each pixel's R-value (redness) by 10 to 100:

```
import imgaug.augmenters as iaa
aug = iaa.WithChannels(0, iaa.Add((10, 100)))
```

Example. Rotate each image's red channel by 0 to 45 degrees:

```
aug = iaa.WithChannels(0, iaa.Affine(rotate=(0, 45)))
```



9.1.6 Noop

Augmenter that never changes input images ("no operation").

API link: *Noop*

Example. Create an augmenter that does nothing:

```
import imgaug.augmenters as iaa
aug = iaa.Noop()
```

9.1.7 Lambda

Augmenter that calls a lambda function for each batch of input image.

API link: *Lambda*

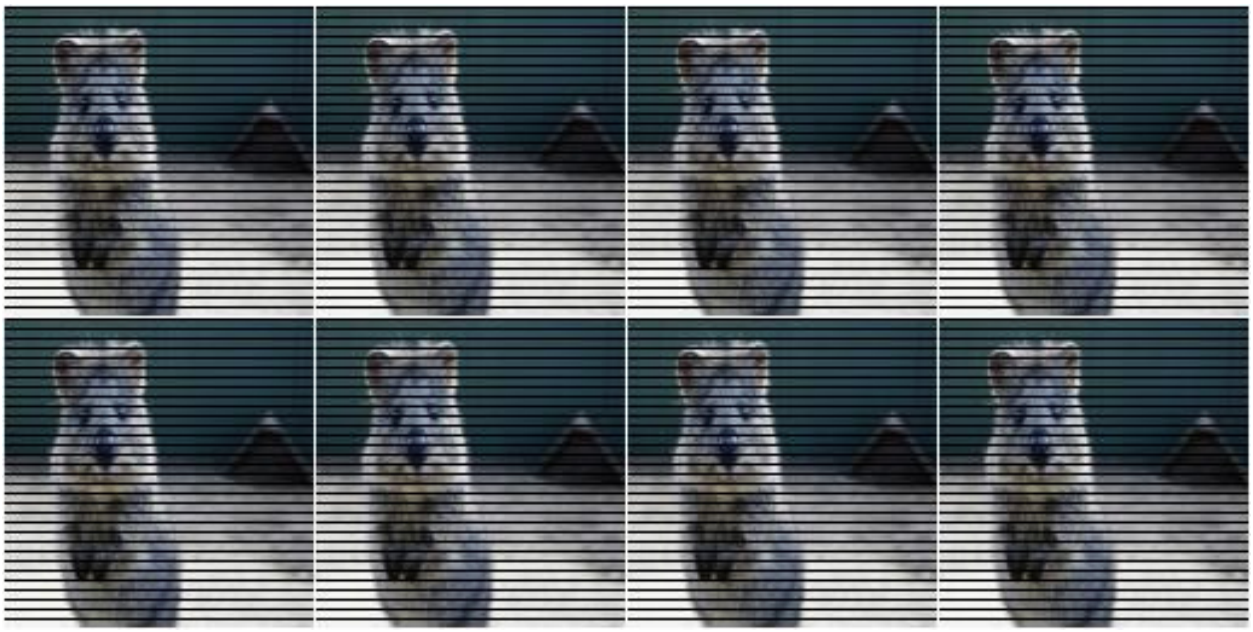
Example. Replace in every image each fourth row with black pixels:

```
import imgaug.augmenters as iaa

def img_func(images, random_state, parents, hooks):
    for img in images:
        img[::4] = 0
    return images

def keypoint_func(keypoints_on_images, random_state, parents, hooks):
    return keypoints_on_images

aug = iaa.Lambda(img_func, keypoint_func)
```



9.1.8 AssertLambda

Augmenter that runs an assert on each batch of input images using a lambda function as condition.

API link: *AssertLambda*

TODO examples

9.1.9 AssertShape

Augmenter to make assumptions about the shape of input image(s) and keypoints.

API link: *AssertShape*

Example. Check if each image in a batch has shape `32x32x3`, otherwise raise an exception:

```
import imgaug.augmenters as iaa
seq = iaa.Sequential([
    iaa.AssertShape((None, 32, 32, 3)),
    iaa.Fliplr(0.5) # only executed if shape matches
])
```

Example. Check if each image in a batch has a height in the range `32<=x<64`, a width of exactly `64` and either `1` or `3` channels:

```
seq = iaa.Sequential([
    iaa.AssertShape((None, (32, 64), 32, [1, 3])),
    iaa.Fliplr(0.5)
])
```

9.1.10 ChannelShuffle

Randomize the order of channels in input images.

API link: *ChannelShuffle*

Example. Shuffle all channels of 35% of all images:

```
import imgaug.augmenters as iaa
aug = iaa.ChannelShuffle(0.35)
```

Example. Shuffle only channels `0` and `1` of 35% of all images. As the new channel orders `0, 1` and `1, 0` are both valid outcomes of the shuffling, it means that for `0.35 * 0.5 = 0.175` or 17.5% of all images the order of channels `0` and `1` is inverted.

```
aug = iaa.ChannelShuffle(0.35, channels=[0, 1])
```

9.2 augmenters.arithmetic

9.2.1 Add

Add a value to all pixels in an image.

API link: *Add*

Example. Add random values between -40 and 40 to images, with each value being sampled once per image and then being the same for all pixels:

```
import imgaug.augmenters as iaa
aug = iaa.Add((-40, 40))
```

Example. Add random values between -40 and 40 to images. In 50% of all images the values differ per channel (3 sampled value). In the other 50% of all images the value is the same for all channels:

```
aug = iaa.Add((-40, 40), per_channel=0.5)
```

9.2.2 AddElementwise

Add values to the pixels of images with possibly different values for neighbouring pixels.

API link: *AddElementwise*

Example. Add random values between -40 and 40 to images, with each value being sampled per pixel:

```
import imgaug.augmenters as iaa
aug = iaa.AddElementwise((-40, 40))
```

Example. Add random values between -40 and 40 to images. In 50% of all images the values differ per channel (3 sampled values per pixel). In the other 50% of all images the value is the same for all channels per pixel:

```
aug = iaa.AddElementwise((-40, 40), per_channel=0.5)
```

9.2.3 AdditiveGaussianNoise

Add noise sampled from gaussian distributions elementwise to images.

API link: *AdditiveGaussianNoise()*

Example. Add gaussian noise to an image, sampled once per pixel from a normal distribution `N(0, s)`, where `s` is sampled per image and varies between `0` and `0.2*255`:

```
import imgaug.augmenters as iaa
aug = iaa.AdditiveGaussianNoise(scale=(0, 0.2*255))
```

Example. Add gaussian noise to an image, sampled once per pixel from a normal distribution `N(0, 0.05*255)`:

```
aug = iaa.AdditiveGaussianNoise(scale=0.2*255)
```

Example. Add laplace noise to an image, sampled channelwise from `N(0, 0.2*255)` (i.e. three independent samples per pixel):

```
aug = iaa.AdditiveGaussianNoise(scale=0.2*255, per_channel=True)
```

9.2.4 AdditiveLaplaceNoise

Add noise sampled from laplace distributions elementwise to images.

The laplace distribution is similar to the gaussian distribution, but puts more weight on the long tail. Hence, this noise will add more outliers (very high/low values). It is somewhere between gaussian noise and salt and pepper noise.

API link: *AdditiveLaplaceNoise()*

Example. Add laplace noise to an image, sampled once per pixel from `Laplace(0, s)`, where `s` is sampled per image and varies between `0` and `0.2*255`:

```
import imgaug.augmenters as iaa
aug = iaa.AdditiveLaplaceNoise(scale=(0, 0.2*255))
```

Example. Add laplace noise to an image, sampled once per pixel from `Laplace(0, 0.2*255)`:

```
aug = iaa.AdditiveLaplaceNoise(scale=0.2*255)
```

Example. Add laplace noise to an image, sampled channelwise from `Laplace(0, 0.2*255)` (i.e. three independent samples per pixel):

```
aug = iaa.AdditiveLaplaceNoise(scale=0.2*255, per_channel=True)
```

9.2.5 AdditivePoissonNoise

Add noise sampled from poisson distributions elementwise to images.

Poisson noise is comparable to gaussian noise, as e.g. generated via `AdditiveGaussianNoise`. As poisson distributions produce only positive numbers, the sign of the sampled values are here randomly flipped.

Values of around `20.0` for `lam` lead to visible noise (for `uint8`). Values of around `40.0` for `lam` lead to very visible noise (for `uint8`). It is recommended to usually set `per_channel` to `True`.

API link: *AdditivePoissonNoise()*

Example. Add poisson noise to an image, sampled once per pixel from `Poisson(lam)`, where `lam` is sampled per image and varies between `0` and `40`:

```
import imgaug.augmenters as iaa
aug = iaa.AdditivePoissonNoise(scale=(0, 40))
```

Example. Add poisson noise to an image, sampled once per pixel from `Poisson(40)`:

```
aug = iaa.AdditivePoissonNoise(40)
```

Example. Add poisson noise to an image, sampled channelwise from `Poisson(40)` (i.e. three independent samples per pixel):

```
aug = iaa.AdditivePoissonNoise(scale=40, per_channel=True)
```

9.2.6 Multiply

Multiply all pixels in an image with a specific value, thereby making the image darker or brighter.

API link: *Multiply*

Example. Multiply each image with a random value between 0.5 and 1.5:

```
import imgaug.augmenters as iaa
aug = iaa.Multiply((0.5, 1.5))
```

Example. Multiply 50% of all images with a random value between 0.5 and 1.5 and multiply the remaining 50% channel-wise, i.e. sample one multiplier independently per channel:

```
aug = iaa.Multiply((0.5, 1.5), per_channel=0.5)
```

9.2.7 MultiplyElementwise

Multiply values of pixels with possibly different values for neighbouring pixels, making each pixel darker or brighter.

API link: *MultiplyElementwise*

Example. Multiply each pixel with a random value between 0.5 and 1.5:

```
import imgaug.augmenters as iaa
aug = iaa.MultiplyElementwise((0.5, 1.5))
```

Example. Multiply in 50% of all images each pixel with random values between 0.5 and 1.5 and multiply in the remaining 50% of all images the pixels channel-wise, i.e. sample one multiplier independently per channel and pixel:

```
aug = iaa.MultiplyElementwise((0.5, 1.5), per_channel=0.5)
```

9.2.8 Dropout

Augmenter that sets a certain fraction of pixels in images to zero.

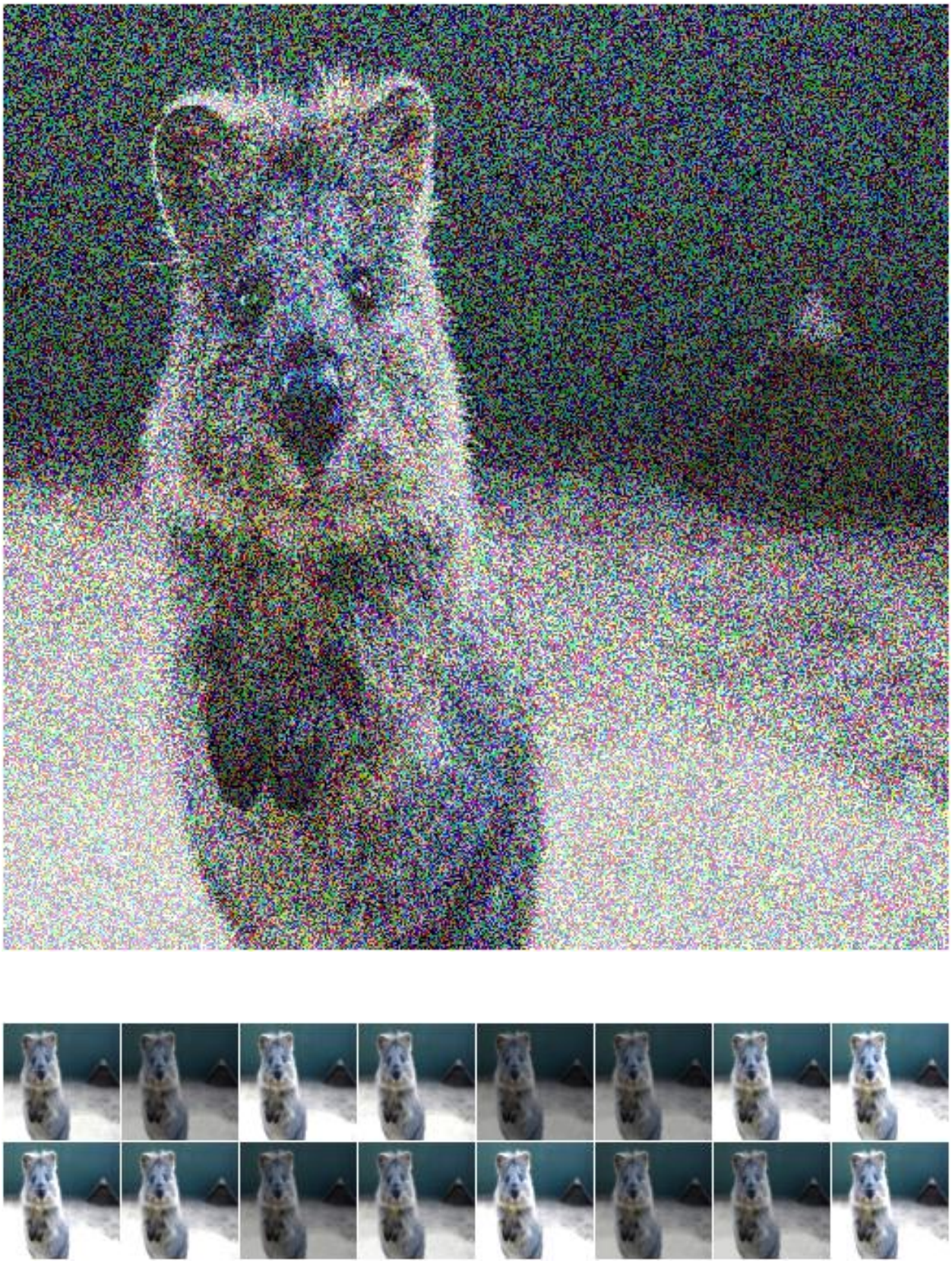
API link: *Dropout()*

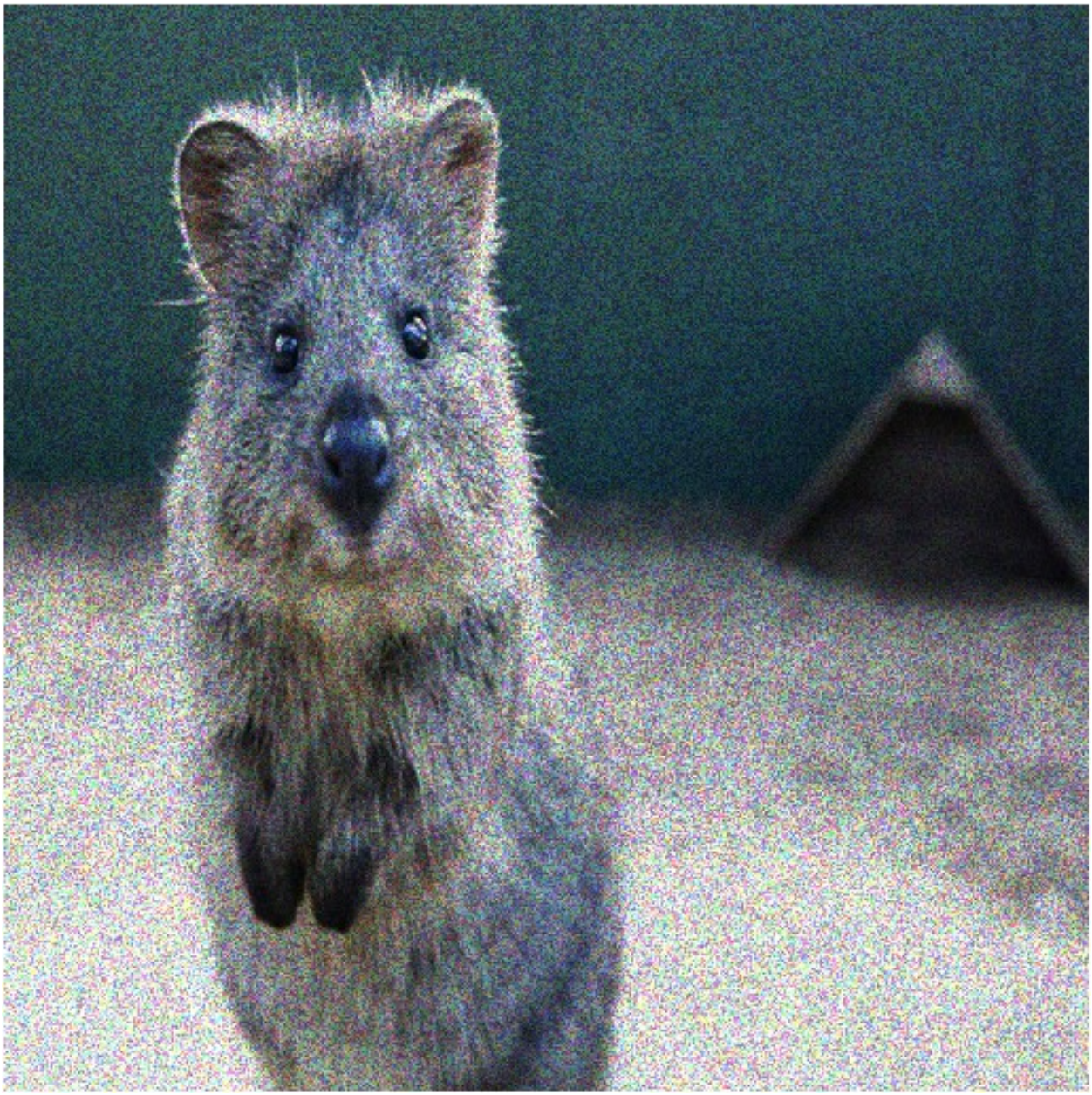
Example. Sample per image a value `p` from the range `0<=p<=0.2` and then drop `p` percent of all pixels in the image (i.e. convert them to black pixels):

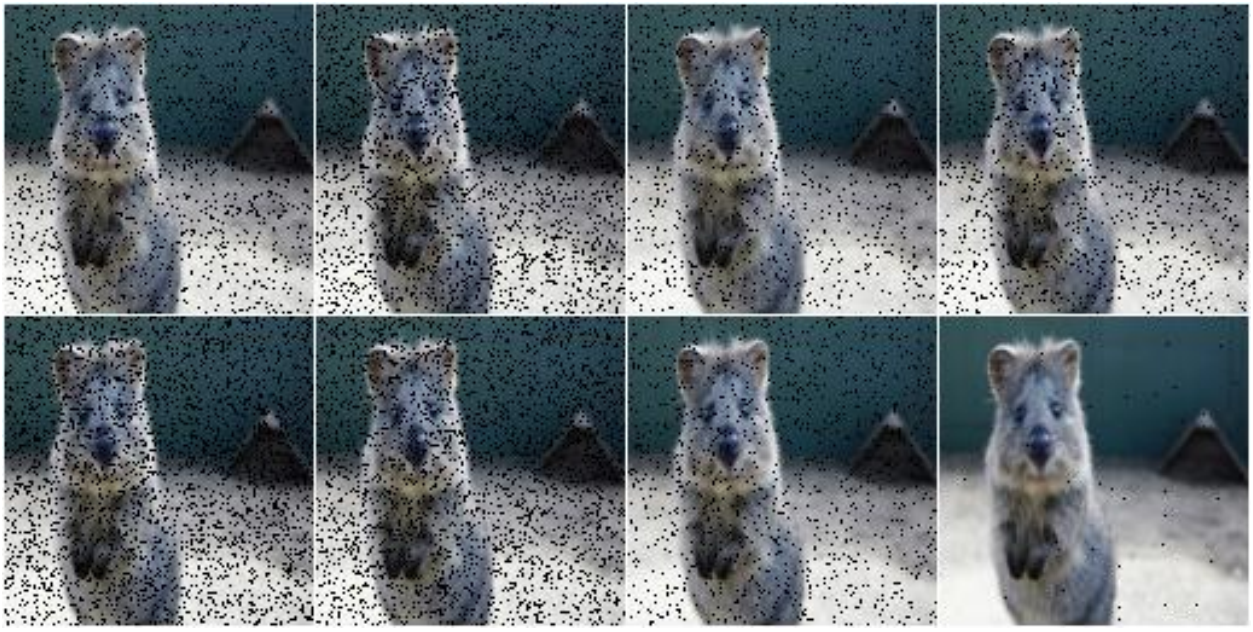
```
import imgaug.augmenters as iaa
aug = iaa.Dropout(p=(0, 0.2))
```

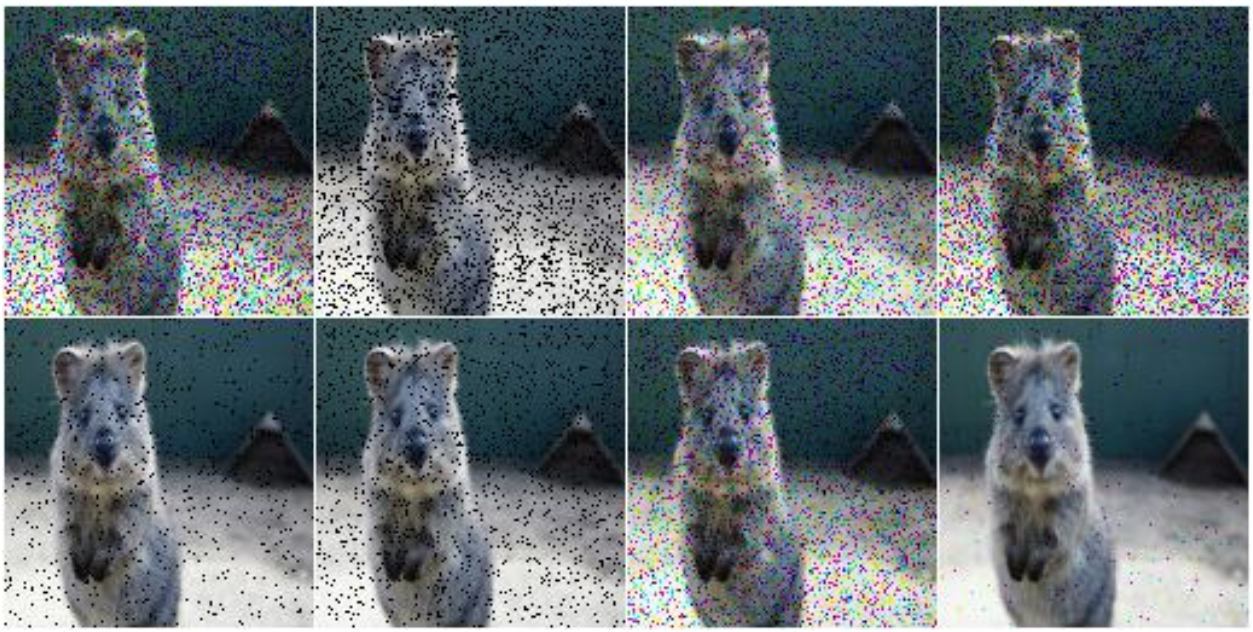
Example. Sample per image a value `p` from the range `0<=p<=0.2` and then drop `p` percent of all pixels in the image (i.e. convert them to black pixels), but do this independently per channel in 50% of all images:

```
aug = iaa.Dropout(p=(0, 0.2), per_channel=0.5)
```









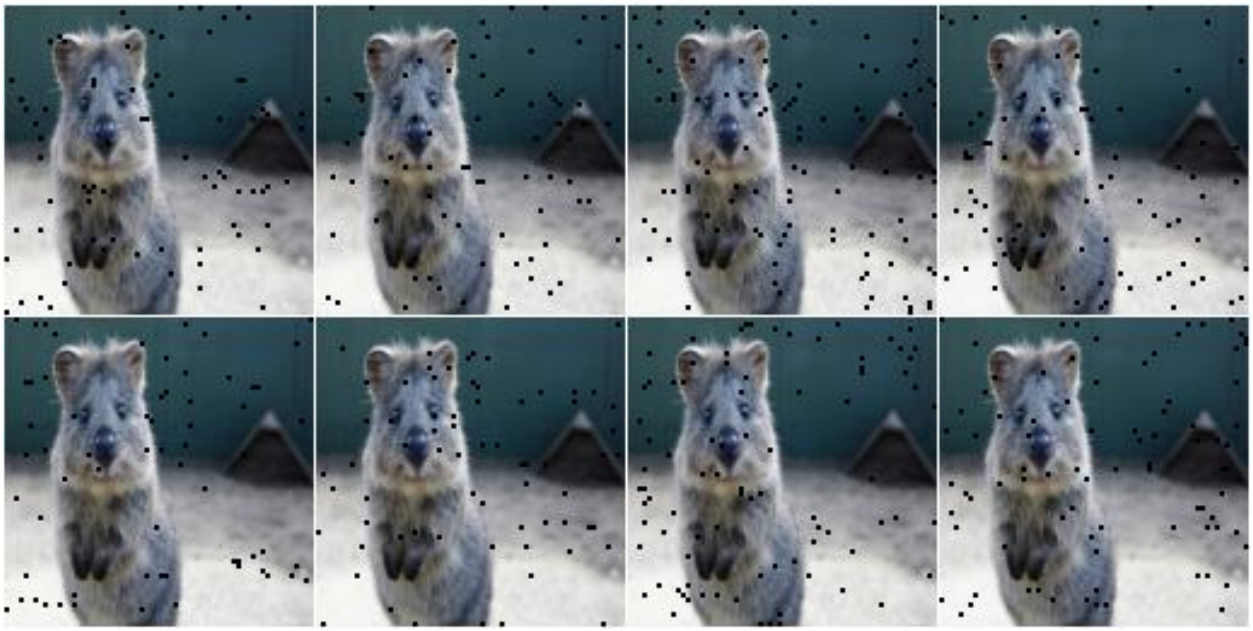
9.2.9 CoarseDropout

Augmenter that sets rectangular areas within images to zero.

API link: *CoarseDropout()*

Example. Drop 2% of all pixels by converting them to black pixels, but do that on a lower-resolution version of the image that has 50% of the original size, leading to 2x2 squares being dropped:

```
import imgaug.augmenters as iaa
aug = iaa.CoarseDropout(0.02, size_percent=0.5)
```



Example. Drop 0 to 5% of all pixels by converting them to black pixels, but do that on a lower-resolution version of the image that has 5% to 50% of the original size, leading to large rectangular areas being dropped:

```
import imgaug.augmenters as iaa
aug = iaa.CoarseDropout((0.0, 0.05), size_percent=(0.02, 0.25))
```

Example. Drop 2% of all pixels by converting them to black pixels, but do that on a lower-resolution version of the image that has 50% of the original size, leading to 2x2 squares being dropped. Also do this in 50% of all images channel-wise, so that only the information of some channels in set to 0 while others remain untouched:

```
aug = iaa.CoarseDropout(0.02, size_percent=0.15, per_channel=0.5)
```

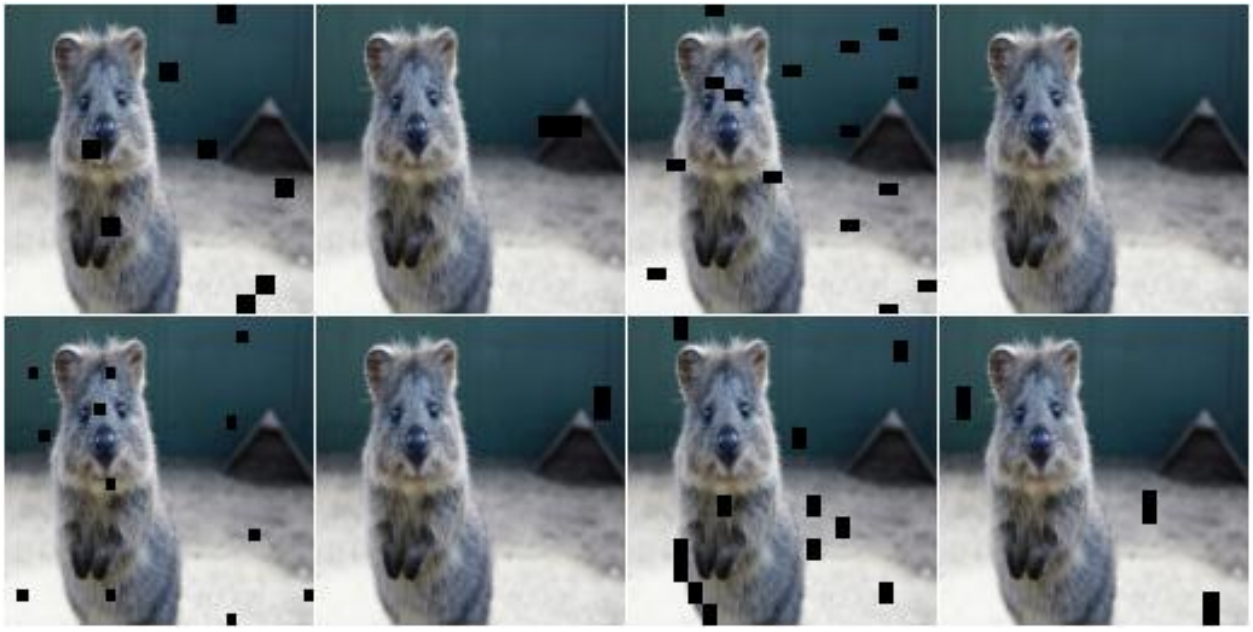
9.2.10 ReplaceElementwise

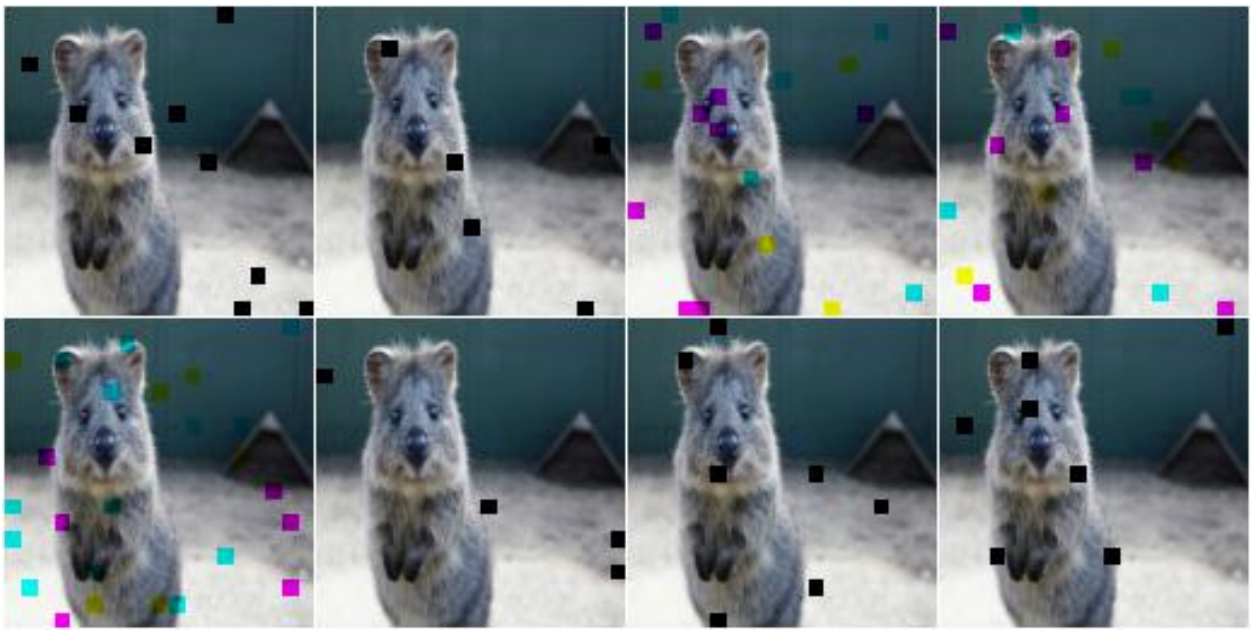
Replace pixels in an image with new values.

API link: *ReplaceElementwise*

Example. Replace `10%` of all pixels with either the value `0` or the value `255`:

```
import imgaug.augmenters as iaa
aug = ReplaceElementwise(0.1, [0, 255])
```





Example. For `50%` of all images, replace `10%` of all pixels with either the value `0` or the value `255` (same as in the previous example). For the other `50%` of all images, replace *channelwise* `10%` of all pixels with either the value `0` or the value `255`. So, it will be very rare for each pixel to have all channels replaced by `255` or `0`.

```
aug = ReplaceElementwise(0.1, [0, 255], per_channel=0.5)
```

Example. Replace `10%` of all pixels by gaussian noise centered around `128`. Both the replacement mask and the gaussian noise are sampled for `50%` of all images.

```
import imgaug.parameters as iap
aug = ReplaceElementwise(0.1, iap.Normal(128, 0.4*128), per_channel=0.5)
```

Example. Replace `10%` of all pixels by gaussian noise centered around `128`. Sample the replacement mask at a lower

resolution (`8x8` pixels) and upscale it to the image size, resulting in coarse areas being replaced by gaussian noise.

```
aug = ReplaceElementwise(
    iap.FromLowerResolution(iap.Binomial(0.1), size_px=8),
    iap.Normal(128, 0.4*128),
    per_channel=0.5)
```

9.2.11 ImpulseNoise

Add impulse noise to images.

This is identical to `SaltAndPepper`, except that `per_channel` is always set to `True`.

API link: *ImpulseNoise()*

Example. Replace `10%` of all pixels with impulse noise:

```
import imgaug.augmenters as iaa
aug = iaa.ImpulseNoise(0.1)
```

9.2.12 SaltAndPepper

Replace pixels in images with salt/pepper noise (white/black-ish colors).

API link: *SaltAndPepper()*

Example. Replace `10%` of all pixels with salt and pepper noise:

```
import imgaug.augmenters as iaa
aug = iaa.SaltAndPepper(0.1)
```

Example. Replace *channelwise* `10%` of all pixels with salt and pepper noise:

```
aug = iaa.SaltAndPepper(0.1, per_channel=True)
```

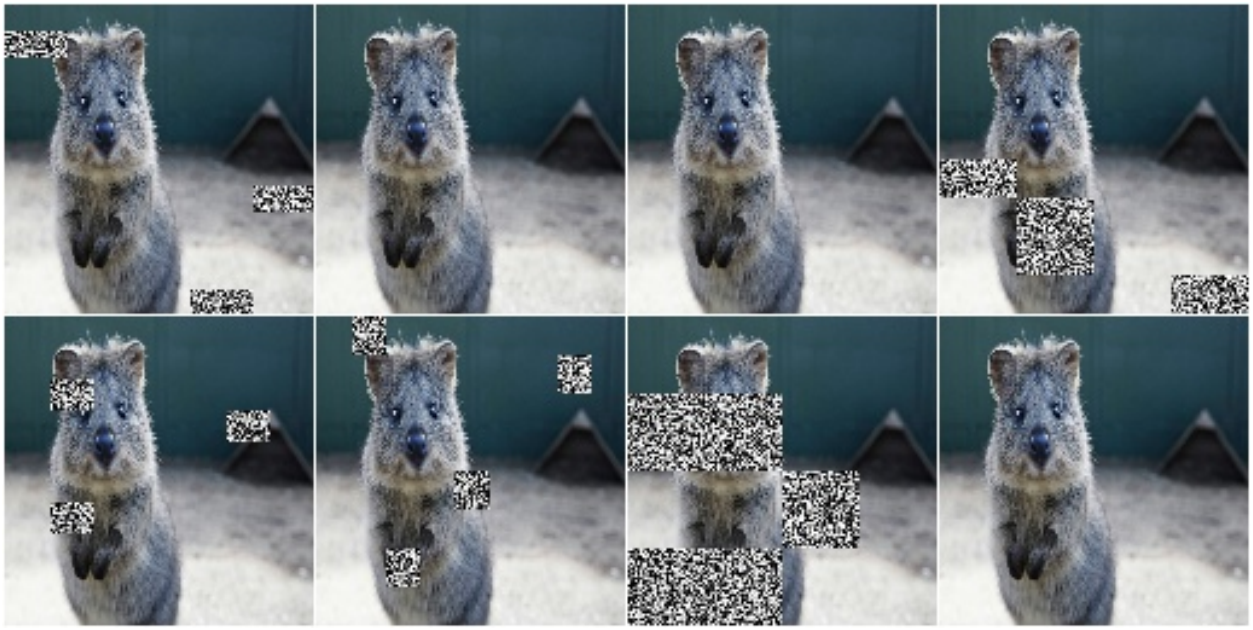
9.2.13 CoarseSaltAndPepper

Replace rectangular areas in images with white/black-ish pixel noise.

API link: *CoarseSaltAndPepper()*

Example. Mark `5%` of all pixels in a mask to be replaced by salt/pepper noise. The mask has `1%` to `10%` the size of the input image. The mask is then upscaled to the input image size, leading to large rectangular areas being marked as to be replaced. These areas are then replaced in the input image by salt/pepper noise.

```
import imgaug.augmenters as iaa
aug = iaa.CoarseSaltAndPepper(0.05, size_percent=(0.01, 0.1))
```

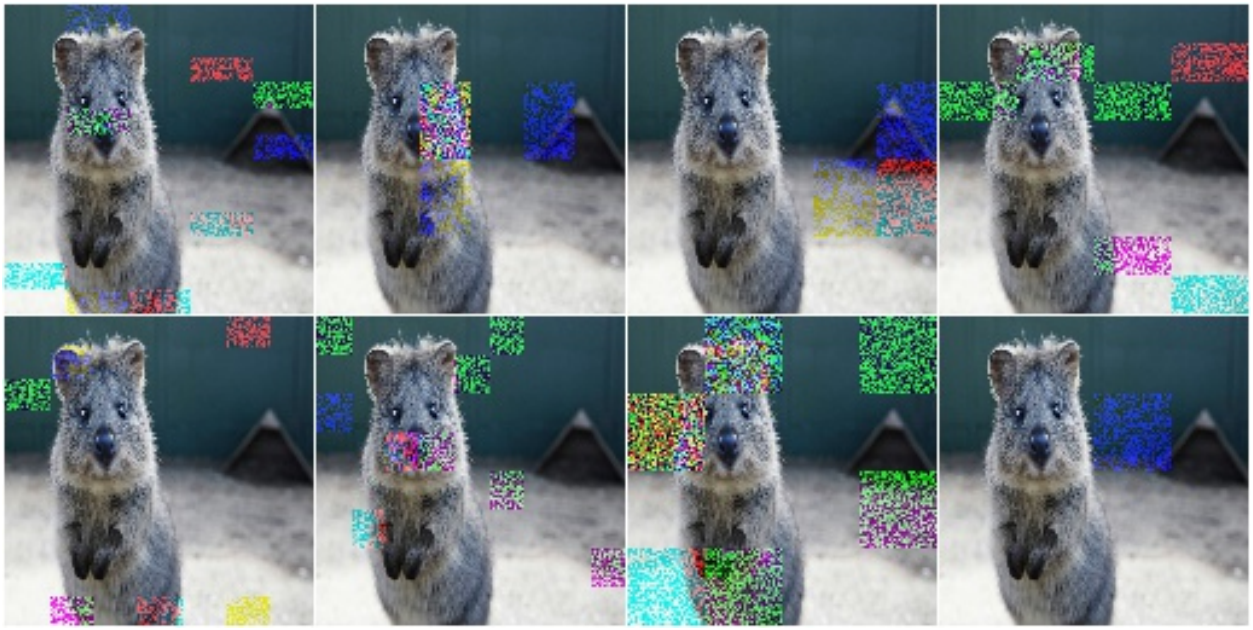


Example. Same as in the previous example, but the replacement mask before upscaling has a size between `4x4` and `16x16` pixels (the axis sizes are sampled independently, i.e. the mask may be rectangular).

```
aug = iaa.CoarseSaltAndPepper(0.05, size_px=(4, 16))
```

Example. Same as in the first example, but mask and replacement are each sampled independently per image channel.

```
aug = iaa.CoarseSaltAndPepper(
    0.05, size_percent=(0.01, 0.1), per_channel=True)
```



9.2.14 Salt

Replace pixels in images with salt noise, i.e. white-ish pixels.

This augmenter is similar to `SaltAndPepper`, but adds no pepper noise to images.

API link: *`Salt()`*

Example. Replace `10%` of all pixels with salt noise (white-ish colors):

```
import imgaug.augmenters as iaa
aug = iaa.Salt(0.1)
```

Example. Similar to `SaltAndPepper`, this augmenter also supports the `per_channel` argument, which is

skipped here for brevity.

9.2.15 CoarseSalt

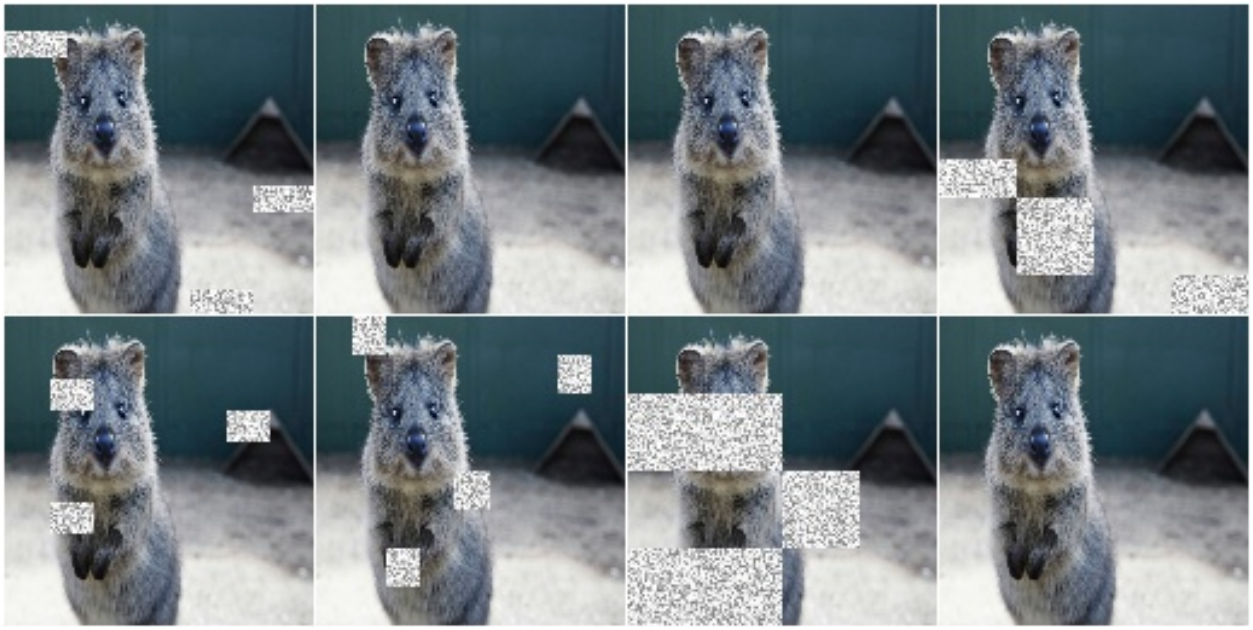
Replace rectangular areas in images with white-ish pixel noise.

This augmenter is similar to `CoarseSaltAndPepper`, but adds no pepper noise to images.

API link: *CoarseSalt()*

Example. Mark `5%` of all pixels in a mask to be replaced by salt noise. The mask has `1%` to `10%` the size of the input image. The mask is then upscaled to the input image size, leading to large rectangular areas being marked as to be replaced. These areas are then replaced in the input image by salt noise.

```
import imgaug.augmenters as iaa
aug = iaa.CoarseSalt(0.05, size_percent=(0.01, 0.1))
```



Similar to `CoarseSaltAndPepper`, this augmenter also supports the `per_channel` argument, which is skipped here for brevity

9.2.16 Pepper

Replace pixels in images with pepper noise, i.e. black-ish pixels.

This augmenter is similar to `SaltAndPepper`, but adds no salt noise to images.

This augmenter is similar to `Dropout`, but slower and the black pixels are not uniformly black.

API link: *Pepper()*

Example. Replace `10%` of all pixels with pepper noise (black-ish colors):

```
import imgaug.augmenters as iaa
aug = iaa.Pepper(0.1)
```

Similar to `SaltAndPepper`, this augmenter also supports the `per_channel` argument, which is skipped here for brevity.

9.2.17 CoarsePepper

Replace rectangular areas in images with black-ish pixel noise.

This augmenter is similar to `CoarseSaltAndPepper`, but adds no salt noise to images.

API link: *CoarsePepper()*

Example. Mark `5%` of all pixels in a mask to be replaced by pepper noise. The mask has `1%` to `10%` the size of the input image. The mask is then upscaled to the input image size, leading to large rectangular areas being marked as to be replaced. These areas are then replaced in the input image by pepper noise.

```
import imgaug.augmenters as iaa
aug = iaa.CoarsePepper(0.05, size_percent=(0.01, 0.1))
```

Similar to `CoarseSaltAndPepper`, this augmenter also supports the `per_channel` argument, which is skipped here for brevity

9.2.18 Invert

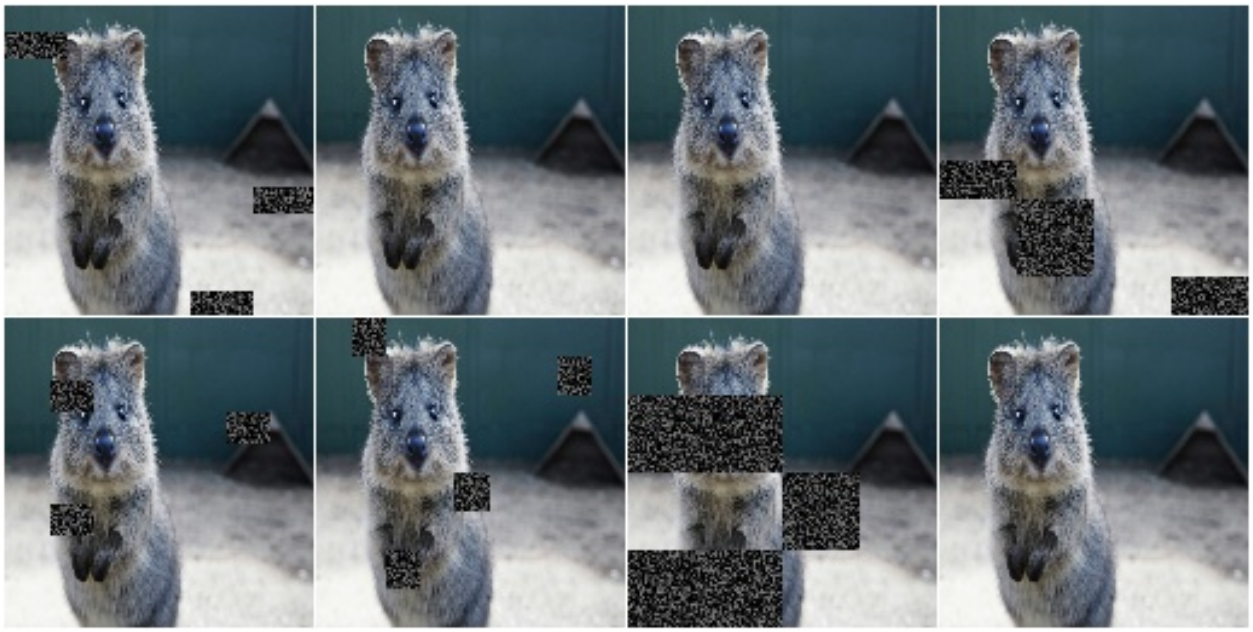
Augmenter that inverts all values in images, i.e. sets a pixel from value `v` to `255-v`.

API link: *Invert*

Example. Invert in 50% of all images all pixels:

```
import imgaug.augmenters as iaa
aug = iaa.Invert(0.5)
```

Example. For 50% of all images, invert all pixels in these images with 25% probability (per image). In the remaining 50% of all images, invert 25% of all channels:



```
aug = iaa.Invert(0.25, per_channel=0.5)
```

9.2.19 ContrastNormalization

Augmenter that changes the contrast of images.

API link: *ContrastNormalization*

Example. Normalize contrast by a factor of 0.5 to 1.5, sampled randomly per image:

```
import imgaug.augmenters as iaa
aug = iaa.ContrastNormalization((0.5, 1.5))
```

Example. Normalize contrast by a factor of 0.5 to 1.5, sampled randomly per image and for 50% of all images also independently per channel:

```
aug = iaa.ContrastNormalization((0.5, 1.5), per_channel=0.5)
```

9.2.20 JpegCompression

Degrade the quality of images by JPEG-compressing them.

API link: *JpegCompression*

Example. Remove high frequency components in images via JPEG compression with a *compression strength* between `80` and `95` (randomly and uniformly sampled per image). This corresponds to a (very low) *quality* setting of `5` to `20`.

```
import imgaug.augmenters as iaa
aug = iaa.JpegCompression(compression=(70, 99))
```

9.3 augmenters.blend

Note: It is not recommended to use blending augmenter with child augmenters that change the geometry of images (e.g. horizontal flips, affine transformations) if you *also* want to augment coordinates (e.g. keypoints, bounding boxes, polygons, ...), as it is not clear which of the two coordinate results (first or second branch) should be used as the coordinates after augmentation. Currently, all blending augmenters try to use the augmented coordinates of the branch that makes up most of the augmented image.

9.3.1 Alpha

Alpha-blend two image sources using an alpha/opacity value.

Currently, if `factor >= 0.5` (per image), the results of the first branch are used as the new coordinates, otherwise the results of the second branch.

API link: *Alpha*

Example. Convert each image to pure grayscale and alpha-blend the result with the original image using an alpha of `50%`, thereby removing about `50%` of all color. This is equivalent to `iaa.Grayscale(0.5)`.

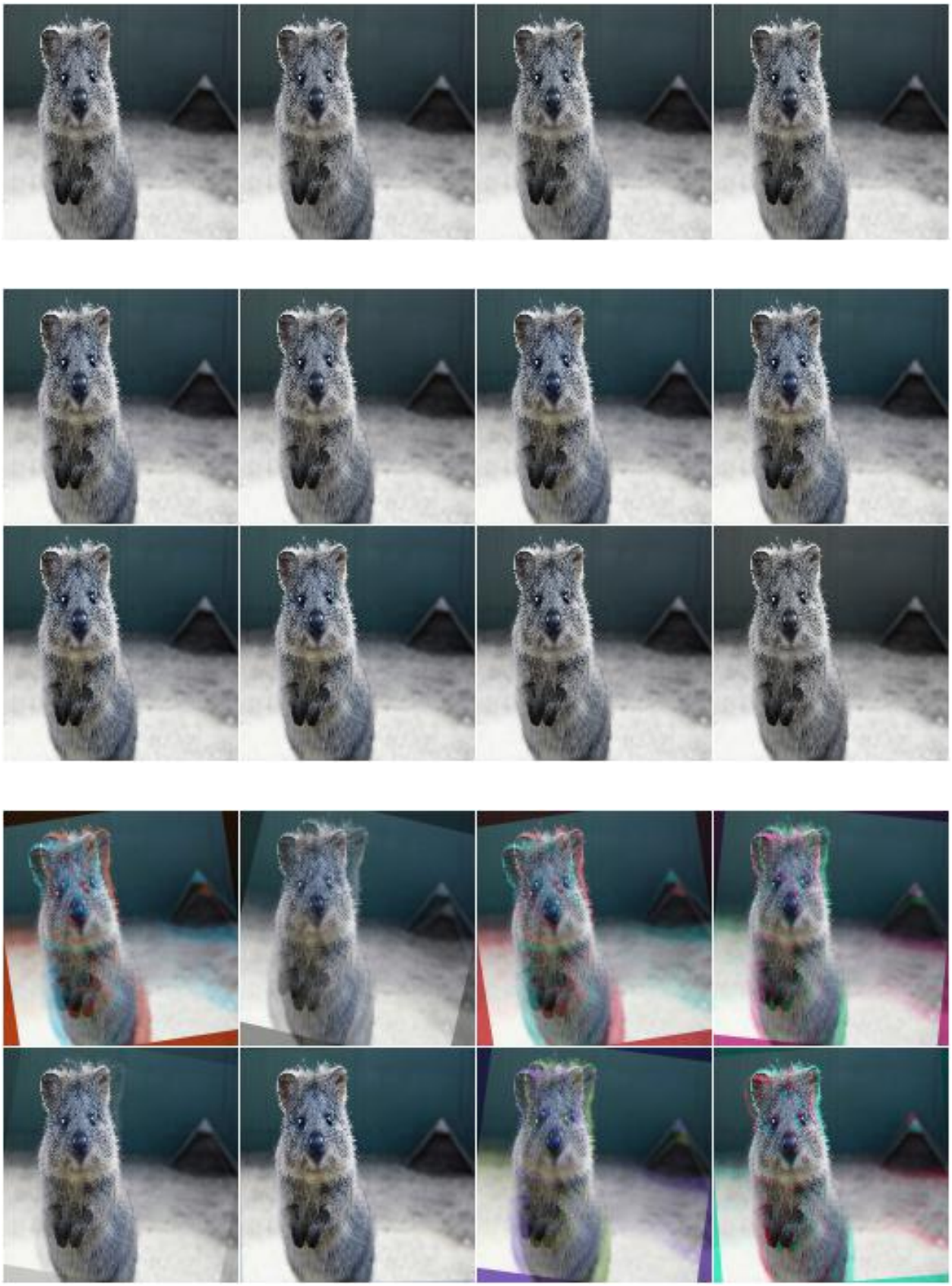
```
import imgaug.augmenters as iaa
aug = iaa.Alpha(0.5, iaa.Grayscale(1.0))
```

Example. Same as in the previous example, but the alpha factor is sampled uniformly from the interval `[0.0, 1.0]` once per image, thereby removing a random fraction of all colors. This is equivalent to `iaa.Grayscale((0.0, 1.0))`.

```
aug = iaa.Alpha((0.0, 1.0), iaa.Grayscale(1.0))
```

Example. First, rotate each image by a random degree sampled uniformly from the interval `[-20, 20]`. Then, alpha-blend that new image with the original one using a random factor sampled uniformly from the interval `[0.0, 1.0]`. For `50%` of all images, the blending happens channel-wise and the factor is sampled independently per channel (`per_channel=0.5`). As a result, e.g. the red channel may look visibly rotated (factor near `1.0`), while the green and blue channels may not look rotated (factors near `0.0`).

```
aug = iaa.Alpha(
    (0.0, 1.0),
    iaa.Affine(rotate=(-20, 20)),
    per_channel=0.5)
```



Example. Apply two branches of augmenters – `A` and `B` – *independently* to input images and alpha-blend the results of these branches using a factor `f`. Branch `A` increases image pixel intensities by `100` and `B` multiplies the pixel intensities by `0.2`. `f` is sampled uniformly from the interval `[0.0, 1.0]` per image. The resulting images contain a bit of `A` and a bit of `B`.

```
aug = iaa.Alpha(
    (0.0, 1.0),
    first=iaa.Add(100),
    second=iaa.Multiply(0.2))
```

Example. Apply median blur to each image and alpha-blend the result with the original image using an alpha factor of either exactly `0.25` or exactly `0.75` (sampled once per image).

```
aug = iaa.Alpha([0.25, 0.75], iaa.MedianBlur(13))
```

9.3.2 AlphaElementwise

Alpha-blend two image sources using alpha/opacity values sampled per pixel.

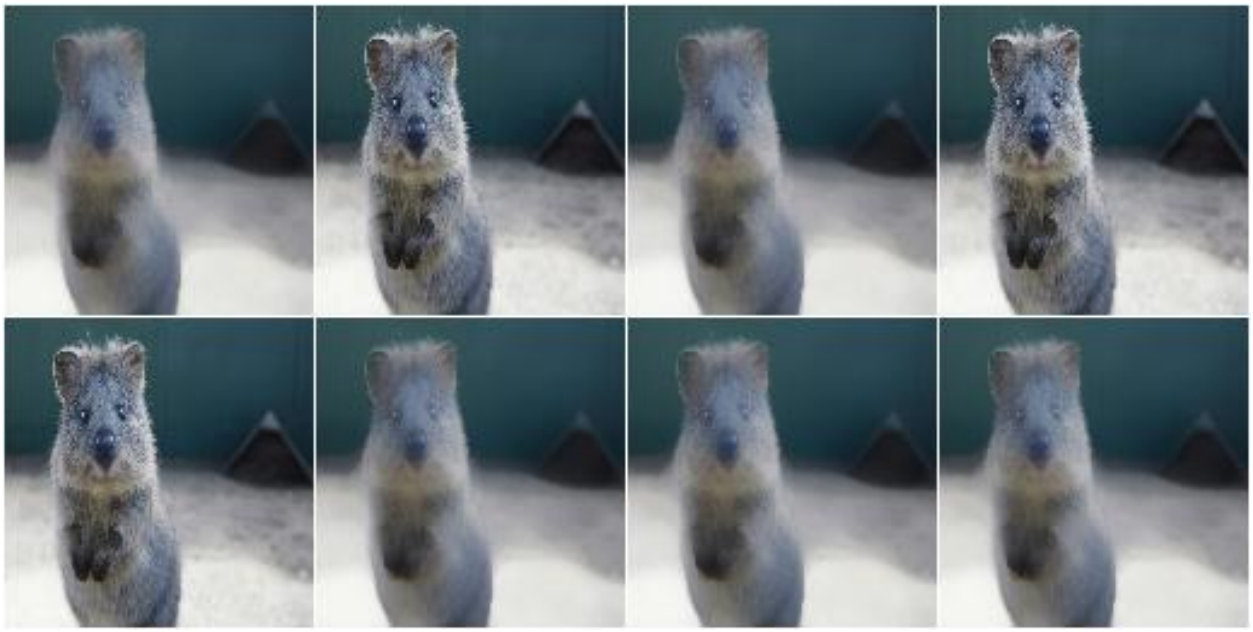
This is the same as `Alpha`, except that the opacity factor is sampled once per *pixel* instead of once per *image* (or a few times per image, if `Alpha.per_channel` is set to `True`).

Currently, if `factor >= 0.5` (per pixel), the results of the first branch are used as the new coordinates, otherwise the results of the second branch.

API link: *AlphaElementwise*

Example. Convert each image to pure grayscale and alpha-blend the result with the original image using an alpha of `50%` for all pixels, thereby removing about `50%` of all color. This is equivalent to `iaa.Grayscale(0.5)`. This is also equivalent to `iaa.Alpha(0.5, iaa.Grayscale(1.0))`, as the opacity has a fixed value of `0.5` and is hence identical for all pixels.

```
import imgaug.augmenters as iaa
aug = iaa.AlphaElementwise(0.5, iaa.Grayscale(1.0))
```



Example. Same as in the previous example, but the alpha factor is sampled uniformly from the interval `[0.0, 1.0]` once per pixel, thereby removing a random fraction of all colors from each pixel. This is equivalent to `iaa.Grayscale((0.0, 1.0))`.

```
aug = iaa.AlphaElementwise((0, 1.0), iaa.Grayscale(1.0))
```

Example. First, rotate each image by a random degree sampled uniformly from the interval `[-20, 20]`. Then, alpha-blend that new image with the original one using a random factor sampled uniformly from the interval `[0.0, 1.0]` per pixel. For `50%` of all images, the blending happens channel-wise and the factor is sampled independently per pixel *and* channel (`per_channel=0.5`). As a result, e.g. the red channel may look visibly rotated (factor near `1.0`), while the green and blue channels may not look rotated (factors near `0.0`).

```
aug = iaa.AlphaElementwise(
    (0.0, 1.0),
    iaa.Affine(rotate=(-20, 20)),
    per_channel=0.5)
```

Example. Apply two branches of augmenters – `A` and `B` – *independently* to input images and alpha-blend the results of these branches using a factor `f`. Branch `A` increases image pixel intensities by `100` and `B` multiplies the pixel intensities by `0.2`. `f` is sampled uniformly from the interval `[0.0, 1.0]` per pixel. The resulting images contain a bit of `A` and a bit of `B`.

```
aug = iaa.AlphaElementwise(
    (0.0, 1.0),
    first=iaa.Add(100),
    second=iaa.Multiply(0.2))
```

Example. Apply median blur to each image and alpha-blend the result with the original image using an alpha factor of either exactly `0.25` or exactly `0.75` (sampled once per pixel).

```
aug = iaa.AlphaElementwise([0.25, 0.75], iaa.MedianBlur(13))
```

9.3.3 SimplexNoiseAlpha

Alpha-blend two image sources using simplex noise alpha masks.

The alpha masks are sampled using a simplex noise method, roughly creating connected blobs of 1s surrounded by 0s. If nearest neighbour upsampling is used, these blobs can be rectangular with sharp edges.

API link: *SimplexNoiseAlpha*

Example. Detect per image all edges, mark them in a black and white image and then alpha-blend the result with the original image using simplex noise masks.

```
import imgaug.augmenters as iaa
aug = iaa.SimplexNoiseAlpha(iaa.EdgeDetect(1.0))
```

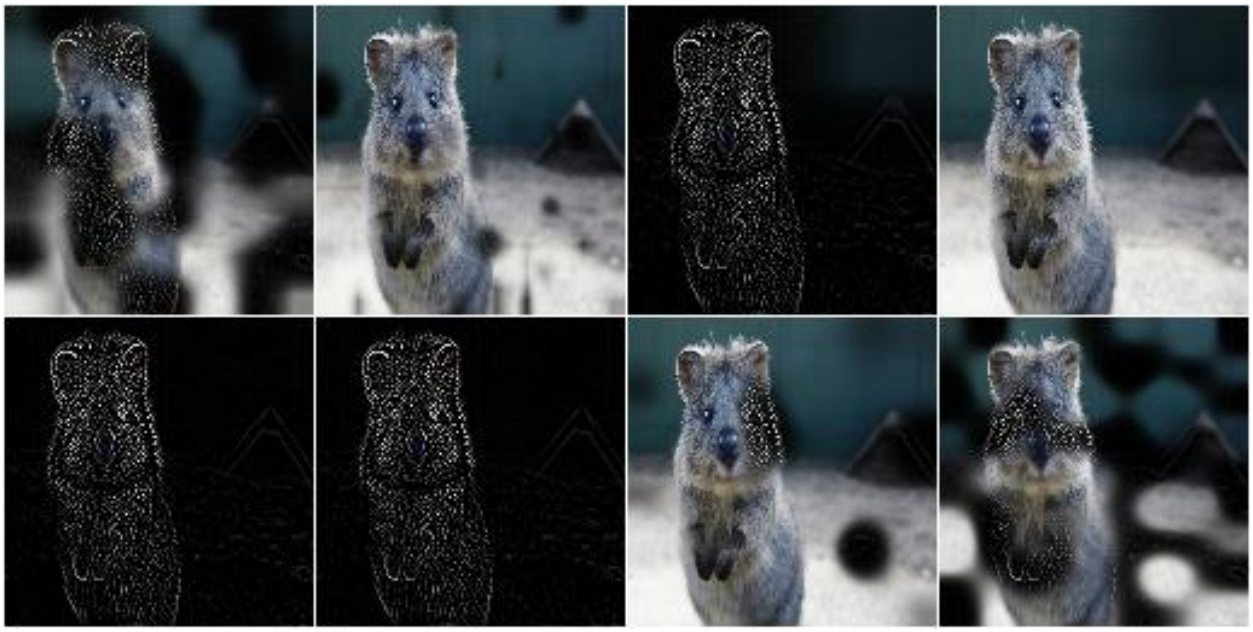
Example. Same as in the previous example, but using only nearest neighbour upscaling to scale the simplex noise masks to the final image sizes, i.e. no nearest linear upsampling is used. This leads to rectangles with sharp edges.

```
aug = iaa.SimplexNoiseAlpha(
    iaa.EdgeDetect(1.0),
    upscale_method="nearest")
```

Example. Same as in the previous example, but using only linear upscaling to scale the simplex noise masks to the final image sizes, i.e. no nearest neighbour upsampling is used. This leads to rectangles with smooth edges.

```
aug = iaa.SimplexNoiseAlpha(
    iaa.EdgeDetect(1.0),
    upscale_method="linear")
```

Example. Same as in the first example, but using a threshold for the sigmoid function that is further to the right. This is more conservative, i.e. the generated noise masks will be mostly black (values around `0.0`), which means that most of the original images (parameter/branch *second*) will be kept, rather than using the results of the augmentation (parameter/branch *first*).



```
import imgaug.parameters as iap
aug = iaa.SimplexNoiseAlpha(
    iaa.EdgeDetect(1.0),
    sigmoid_thresh=iap.Normal(10.0, 5.0))
```

9.3.4 FrequencyNoiseAlpha

Alpha-blend two image sources using frequency noise masks.

The alpha masks are sampled using frequency noise of varying scales, which can sometimes create large connected blobs of 1s surrounded by 0s and other times results in smaller patterns. If nearest neighbour upsampling is used, these blobs can be rectangular with sharp edges.

API link: *FrequencyNoiseAlpha*

Example. Detect per image all edges, mark them in a black and white image and then alpha-blend the result with the original image using frequency noise masks.

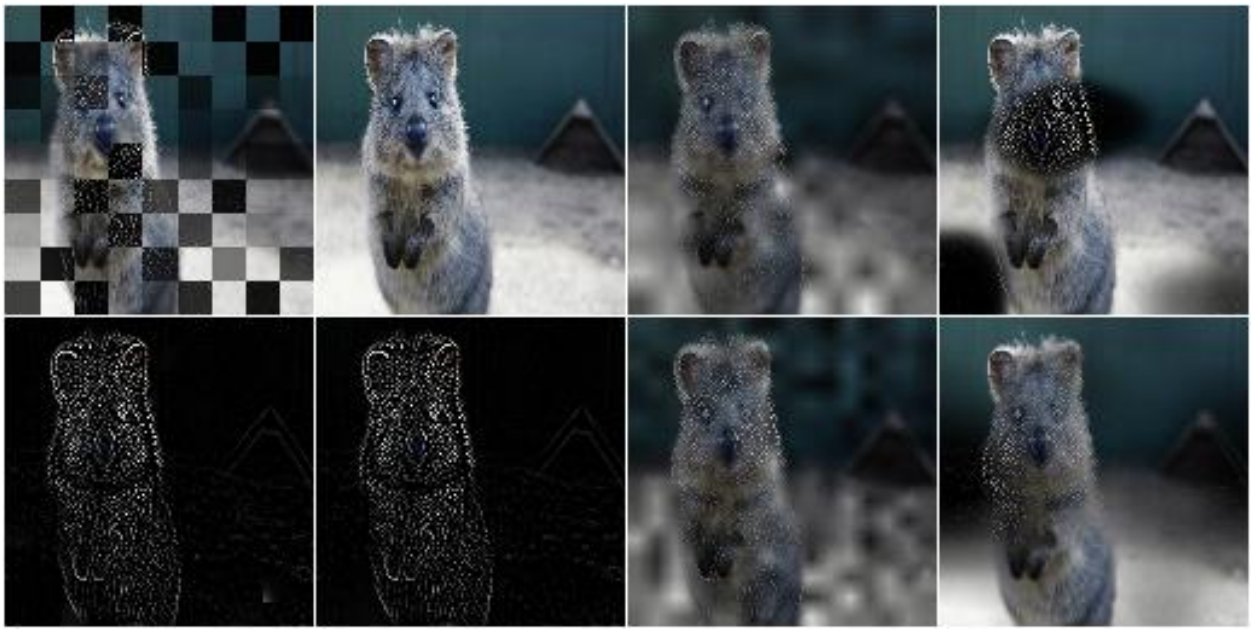
```
import imgaug.augmenters as iaa
aug = iaa.FrequencyNoiseAlpha(first=iaa.EdgeDetect(1.0))
```

Example. Same as the first example, but using only linear upscaling to scale the frequency noise masks to the final image sizes, i.e. no nearest neighbour upsampling is used. This results in smooth edges.

```
aug = iaa.FrequencyNoiseAlpha(
    first=iaa.EdgeDetect(1.0),
    upscale_method="nearest")
```

Example. Same as the first example, but using only linear upscaling to scale the frequency noise masks to the final image sizes, i.e. no nearest neighbour upsampling is used. This results in smooth edges.

```
aug = iaa.FrequencyNoiseAlpha(
    first=iaa.EdgeDetect(1.0),
    upscale_method="linear")
```



Example. Same as in the previous example, but with the exponent set to a constant `-2` and the sigmoid deactivated, resulting in cloud-like patterns without sharp edges.

```
aug = iaa.FrequencyNoiseAlpha(
    first=iaa.EdgeDetect(1.0),
    upscale_method="linear",
    exponent=-2,
    sigmoid=False)
```

Example. Same as the first example, but using a threshold for the sigmoid function that is further to the right. This is more conservative, i.e. the generated noise masks will be mostly black (values around `0.0`), which means that most of the original images (parameter/branch *second*) will be kept, rather than using the results of the augmentation (parameter/branch *first*).

```
import imgaug.parameters as iap
aug = iaa.FrequencyNoiseAlpha(
    first=iaa.EdgeDetect(1.0),
    sigmoid_thresh=iap.Normal(10.0, 5.0))
```

9.4 augmenters.blur

9.4.1 GaussianBlur

Augmenter to blur images using gaussian kernels.

API link: *GaussianBlur*

Example. Blur each image with a gaussian kernel with a sigma of `3.0`:

```
import imgaug.augmenters as iaa
aug = iaa.GaussianBlur(sigma=(0.0, 3.0))
```

9.4.2 AverageBlur

Blur an image by computing simple means over neighbourhoods.

API link: *AverageBlur*

Example. Blur each image using a mean over neihbourhoods that have a random size between 2x2 and 11x11:

```
import imgaug.augmenters as iaa
aug = iaa.AverageBlur(k=(2, 11))
```

Example. Blur each image using a mean over neihbourhoods that have random sizes, which can vary between 5 and 11 in height and 1 and 3 in width:

```
aug = iaa.AverageBlur(k=((5, 11), (1, 3)))
```

9.4.3 MedianBlur

Blur an image by computing median values over neighbourhoods.

API link: *MedianBlur*

Example. Blur each image using a median over neihbourhoods that have a random size between 3x3 and 11x11:

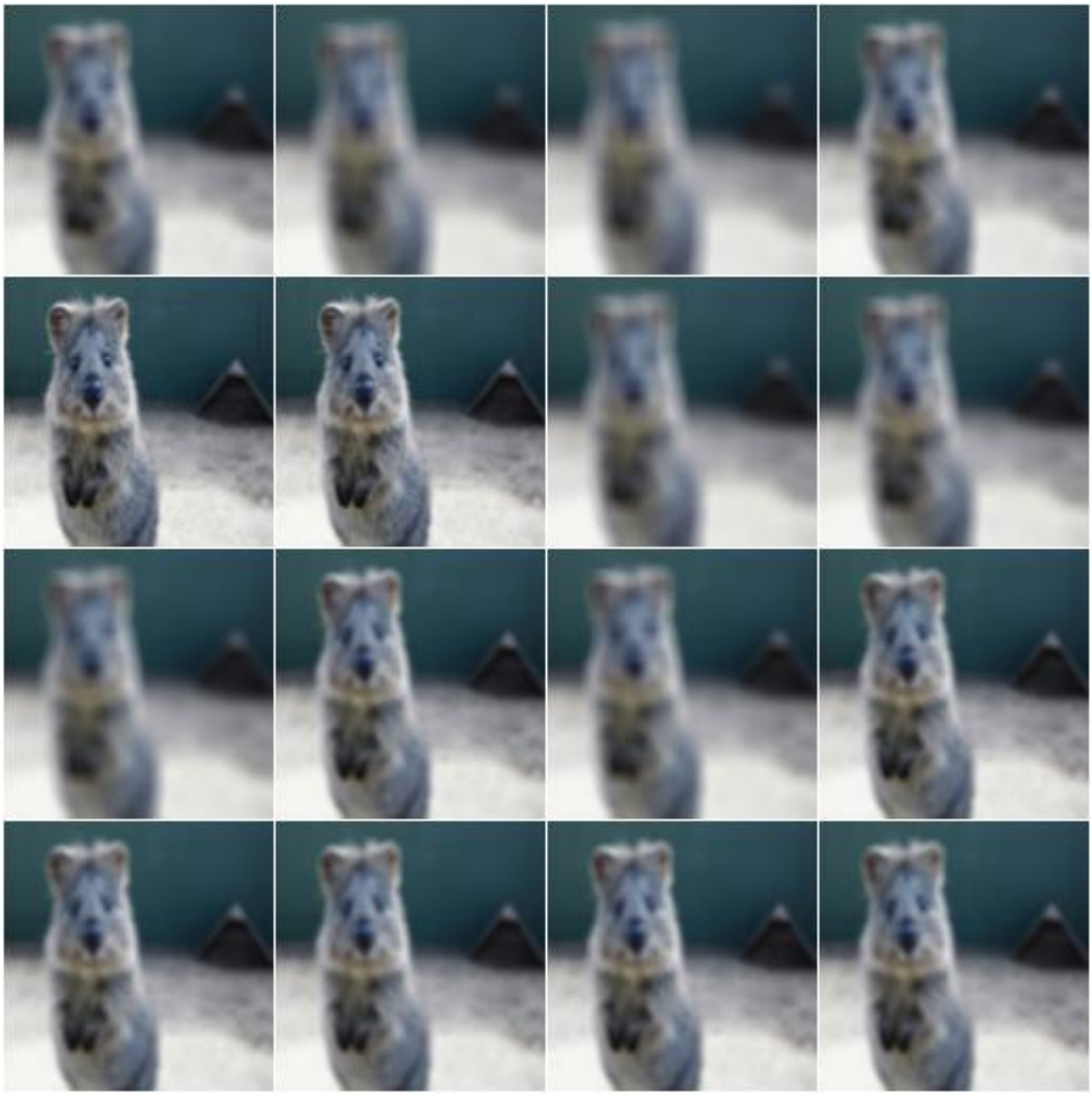
```
import imgaug.augmenters as iaa
aug = iaa.MedianBlur(k=(3, 11))
```

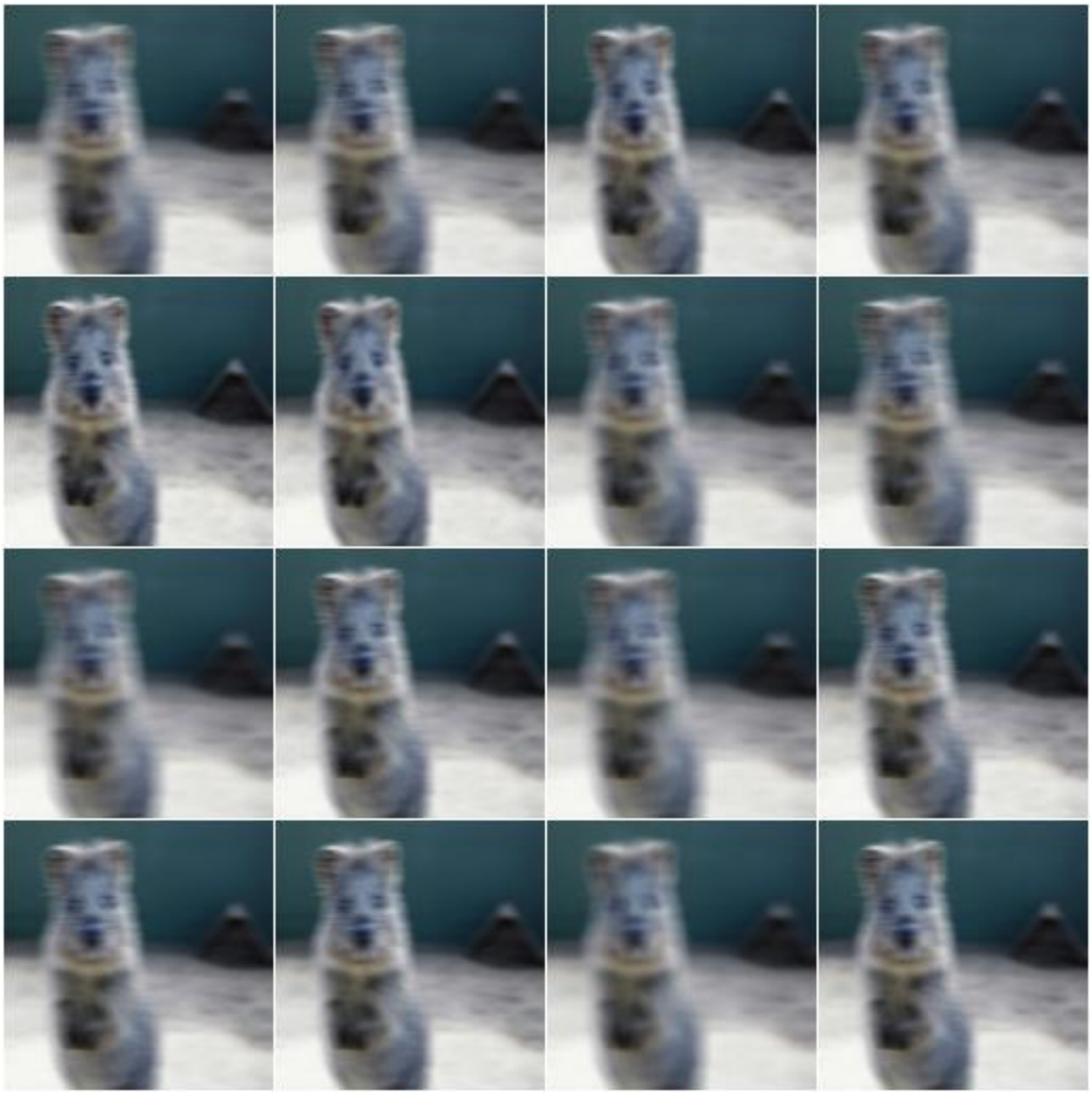
9.4.4 BilateralBlur

Blur/Denoise an image using a bilateral filter.

Bilateral filters blur homogenous and textured areas, while trying to preserve edges.

API link: *BilateralBlur*





Example. Blur all images using a bilateral filter with a *max distance* sampled uniformly from the interval `[3, 10]` and wide ranges for *sigma_color* and *sigma_space*:

```
import imgaug.augmenters as iaa
aug = iaa.BilateralBlur(
    d=(3, 10), sigma_color=(10, 250), sigma_space=(10, 250))
```

9.4.5 MotionBlur

Blur images in a way that fakes camera or object movements.

API link: *MotionBlur*

Example. Apply motion blur with a kernel size of `15x15` pixels to images:

```
import imgaug.augmenters as iaa
aug = iaa.MotionBlur(k=15)
```

Example. Apply motion blur with a kernel size of `15x15` pixels and a blur angle of either `-45` or `45` degrees (randomly picked per image):

```
aug = iaa.MotionBlur(k=15, angle=[-45, 45])
```

9.5 augmenters.color

9.5.1 WithColorspace

Apply child augmenters within a specific colorspace.

This augumenter takes a source colorspace A and a target colorspace B as well as children C. It changes images from A to B, then applies the child augmenters C and finally changes the colorspace back from B to A. See also ChangeColorspace() for more.

API link: *WithColorspace*

Example. Convert to `HSV` colorspace, add a value between `0` and `50` (uniformly sampled per image) to the Hue channel, then convert back to the input colorspace (`RGB`).

```
import imgaug.augmenters as iaa
aug = iaa.WithColorspace(
    to_colorspace="HSV",
    from_colorspace="RGB",
    children=iaa.WithChannels(
        0,
        iaa.Add((0, 50))
    )
)
```

9.5.2 WithHueAndSaturation

Apply child augmenters to hue and saturation channels.

This augumenter takes an image in a source colorspace, converts it to HSV, extracts the H (hue) and S (saturation) channels, applies the provided child augmenters to these channels and finally converts back to the original colorspace.

The image array generated by this augmenter and provided to its children is in `int16` (**sic!** only augmenters that can handle `int16` arrays can be children!). The hue channel is mapped to the value range `[0, 255]`. Before converting

back to the source colorspace, the saturation channel's values are clipped to `[0, 255]`. A modulo operation is applied to the hue channel's values, followed by a mapping from `[0, 255]` to `[0, 180]` (and finally the colorspace conversion).

API link: *WithHueAndSaturation*

Example. Create an augmenter that will add a random value between `0` and `50` (uniformly sampled per image) hue channel in HSV colorspace. It automatically accounts for the hue being in angular representation, i.e. if the angle goes beyond 360 degrees, it will start again at 0 degrees. The colorspace is finally converted back to `RGB` (default setting).

```
import imgaug.augmenters as iaa
aug = iaa.WithHueAndSaturation(
    iaa.WithChannels(0, iaa.Add((0, 50)))
)
```

Example. Create an augmenter that adds a random value sampled uniformly from the range `[-30, 10]` to the hue and multiplies the saturation by a random factor sampled uniformly from `[0.5, 1.5]`. It also modifies the contrast of the saturation channel. After these steps, the `HSV` image is converted back to `RGB`.

```
aug = iaa.WithHueAndSaturation([
    iaa.WithChannels(0, iaa.Add((-30, 10))),
    iaa.WithChannels(1, [
        iaa.Multiply((0.5, 1.5)),
        iaa.LinearContrast((0.75, 1.25))
    ])
])
```

9.5.3 MultiplyHueAndSaturation

Multipy hue and saturation by random values.

The augmenter first transforms images to HSV colorspace, then multiplies the pixel values in the H and S channels and afterwards converts back to RGB.

This augmenter is a wrapper around `WithHueAndSaturation`.

API link: *MultiplyHueAndSaturation()*

Example. Multiply hue and saturation by random values between `0.5` and `1.5` (independently per channel and the same value for all pixels within that channel). The hue will be automatically projected to an angular representation.

```
import imgaug.augmenters as iaa
aug = iaa.MultiplyHueAndSaturation((0.5, 1.5), per_channel=True)
```

Example. Multiply only the hue by random values between `0.5` and `1.5`.

```
aug = iaa.MultiplyHueAndSaturation(mul_hue=(0.5, 1.5))
```

Example. Multiply only the saturation by random values between `0.5` and `1.5`.

```
aug = iaa.MultiplyHueAndSaturation(mul_saturation=(0.5, 1.5))
```

9.5.4 MultiplyHue

Multiply the hue of images by random values.

The augmenter first transforms images to HSV colorspace, then multiplies the pixel values in the H channel and afterwards converts back to RGB.

This augmenter is a shortcut for `MultiplyHueAndSaturation(mul_hue=...)`.

API link: *MultiplyHue()*

Example. Multiply the hue channel of images using random values between `0.5` and `1.5`:

```
import imgaug.augmenters as iaa
aug = iaa.MultiplyHue((0.5, 1.5))
```

9.5.5 MultiplySaturation

Multiply the saturation of images by random values.

The augmenter first transforms images to HSV colorspace, then multiplies the pixel values in the H channel and afterwards converts back to RGB.

This augmenter is a shortcut for `MultiplyHueAndSaturation(mul_saturation=...)`.

API link: *MultiplySaturation()*

Example. Multiply the saturation channel of images using random values between `0.5` and `1.5`:

```
import imgaug.augmenters as iaa
aug = iaa.MultiplySaturation((0.5, 1.5))
```

9.5.6 AddToHueAndSaturation

Increases or decreases hue and saturation by random values.

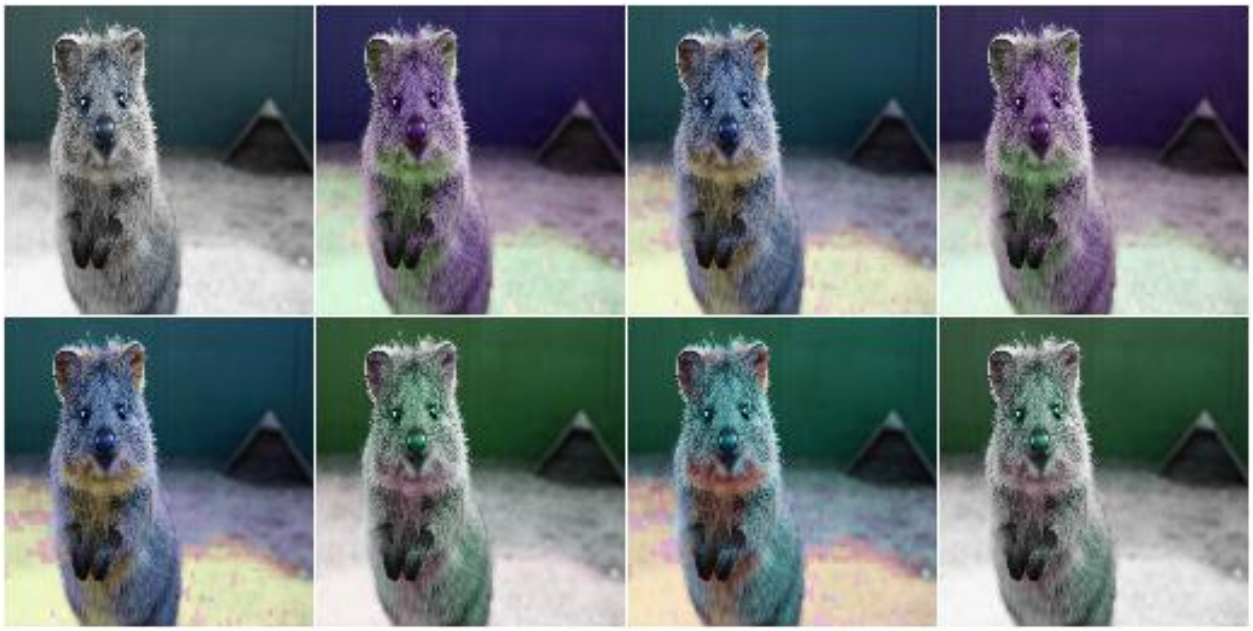
The augmenter first transforms images to HSV colorspace, then adds random values to the H and S channels and afterwards converts back to RGB.

This augmenter is faster than using `WithHueAndSaturation` in combination with `Add`.

API link: *AddToHueAndSaturation*

Example. Add random values between `-50` and `50` to the hue and saturation (independently per channel and the same value for all pixels within that channel):

```
import imgaug.augmenters as iaa
aug = iaa.AddToHueAndSaturation((-50, 50), per_channel=True)
```



9.5.7 AddToHue

Add random values to the hue of images.

The augmenter first transforms images to HSV colorspace, then adds random values to the H channel and afterwards converts back to RGB.

If you want to change both the hue and the saturation, it is recommended to use `AddToHueAndSaturation` as

otherwise the image will be converted twice to HSV and back to RGB.

This augmenter is a shortcut for `AddToHueAndSaturation(value_hue=...)`.

API link: *AddToHue()*

Example. Sample random values from the discrete uniform range `[-50..50]`, convert them to angular representation and add them to the hue, i.e. to the `H` channel in `HSV` colorspace:

```
import imgaug.augmenters as iaa
aug = iaa.AddToHue((-50, 50))
```

9.5.8 AddToSaturation

Add random values to the saturation of images.

The augmenter first transforms images to HSV colorspace, then adds random values to the S channel and afterwards converts back to RGB.

If you want to change both the hue and the saturation, it is recommended to use `AddToHueAndSaturation` as otherwise the image will be converted twice to HSV and back to RGB.

This augmenter is a shortcut for `AddToHueAndSaturation(value_saturation=...)`.

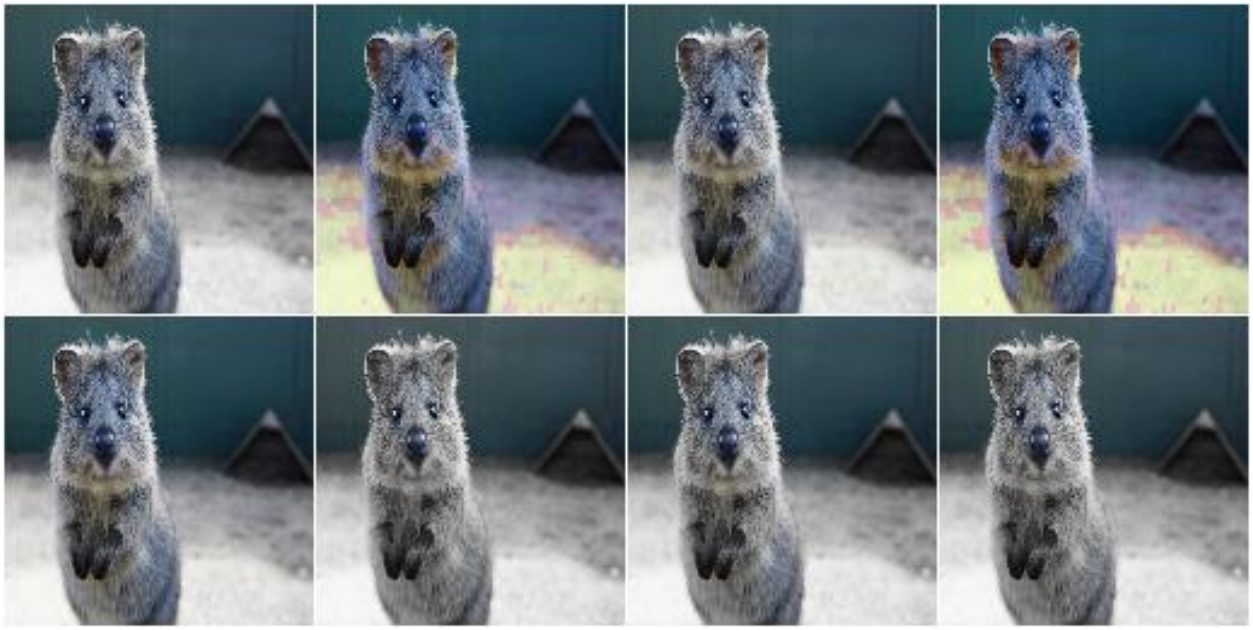
API link: *AddToSaturation()*

Example. Sample random values from the discrete uniform range `[-50..50]`, and add them to the saturation, i.e. to the `S` channel in `HSV` colorspace:

```
import imgaug.augmenters as iaa
aug = iaa.AddToSaturation((-50, 50))
```

9.5.9 ChangeColorspace

Augmenter to change the colorspace of images.



API link: *ChangeColorspace*

Example. The following example shows how to change the colorspace from RGB to HSV, then add 50-100 to the first channel, then convert back to RGB. This increases the hue value of each image.

```
import imgaug.augmenters as iaa
aug = iaa.Sequential([
    iaa.ChangeColorspace(from_colorspace="RGB", to_colorspace="HSV"),
    iaa.WithChannels(0, iaa.Add((50, 100))),
    iaa.ChangeColorspace(from_colorspace="HSV", to_colorspace="RGB")
])
```

9.5.10 Grayscale

Augmenter to convert images to their grayscale versions.

API link: *Grayscale*

Example. Change images to grayscale and overlay them with the original image by varying strengths, effectively removing 0 to 100% of the color:

```
import imgaug.augmenters as iaa
aug = iaa.Grayscale(alpha=(0.0, 1.0))
```

Example. Visualization of increasing `alpha` from `0.0` to `1.0` in eight steps:

9.5.11 KMeansColorQuantization

Quantize colors using k-Means clustering.

This "collects" the colors from the input image, groups them into `k` clusters using k-Means clustering and replaces the colors in the input image using the cluster centroids.

This is slower than `UniformColorQuantization`, but adapts dynamically to the color range in the input image.

Note: This augmenter expects input images to be either grayscale or to have 3 or 4 channels and use colorspace *from_colorspace*. If images have 4 channels, it is assumed that the 4th channel is an alpha channel and it will not be quantized.

API link: *KMeansColorQuantization*

Example. Create an augmenter to apply k-Means color quantization to images using a random amount of colors, sampled uniformly from the interval `[2..16]`. It assumes the input image colorspace to be `RGB` and clusters colors randomly in `RGB` or `Lab` colorspace.

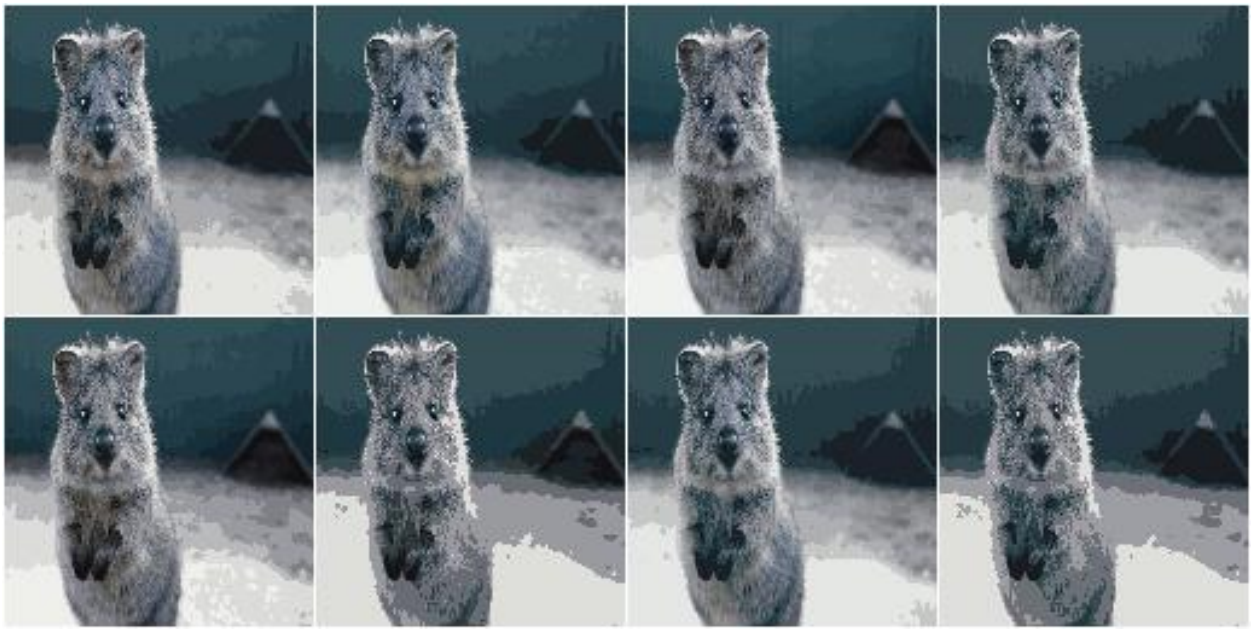
```
import imgaug.augmenters as iaa
aug = iaa.KMeansColorQuantization()
```

Example. Create an augmenter that quantizes images to (up to) eight colors:

```
aug = iaa.KMeansColorQuantization(n_colors=8)
```

Example. Create an augmenter that quantizes images to (up to) `n` colors, where `n` is randomly and uniformly sampled from the discrete interval `[4..32]`:

```
aug = iaa.KMeansColorQuantization(n_colors=(4, 16))
```



Example. Create an augmenter that quantizes input images that are in `BGR` colorspace. The quantization happens in `RGB` or `Lab` colorspace, into which the images are temporarily converted.

```
aug = iaa.KMeansColorQuantization(
    from_colorspace=iaa.ChangeColorspace.BGR)
```

Example. Create an augmenter that quantizes images by clustering colors randomly in either `RGB` or `HSV` colorspace. The assumed input colorspace of images is `RGB`.

```
aug = iaa.KMeansColorQuantization(
    to_colorspace=[iaa.ChangeColorspace.RGB, iaa.ChangeColorspace.HSV])
```

9.5.12 UniformColorQuantization

Quantize colors into N bins with regular distance.

For `uint8` images the equation is `floor(v/q)*q + q/2` with `q = 256/N`, where `v` is a pixel intensity value and `N` is the target number of colors after quantization.

This augmenter is faster than `KMeansColorQuantization`, but the set of possible output colors is constant (i.e. independent of the input images). It may produce unsatisfying outputs for input images that are made up of very similar colors.

Note: This augmenter expects input images to be either grayscale or to have 3 or 4 channels and use colorspace *from_colorspace*. If images have 4 channels, it is assumed that the 4th channel is an alpha channel and it will not be quantized.

API link: *UniformColorQuantization*

Example. Create an augmenter to apply uniform color quantization to images using a random amount of colors, sampled uniformly from the discrete interval `[2..16]`:

```
import imgaug.augmenters as iaa
aug = iaa.UniformColorQuantization()
```

Example. Create an augmenter that quantizes images to (up to) eight colors:

```
aug = iaa.UniformColorQuantization(n_colors=8)
```

Example. Create an augmenter that quantizes images to (up to) `n` colors, where `n` is randomly and uniformly sampled from the discrete interval `[4..32]`:

```
aug = iaa.UniformColorQuantization(n_colors=(4, 16))
```

Example. Create an augmenter that uniformly quantizes images in either `RGB` or `HSV` colorspace (randomly picked per image). The input colorspace of all images has to be `BGR`.

```
aug = iaa.UniformColorQuantization(
    from_colorspace=iaa.ChangeColorspace.BGR,
    to_colorspace=[iaa.ChangeColorspace.RGB, iaa.ChangeColorspace.HSV])
```

9.6 augmenters.contrast

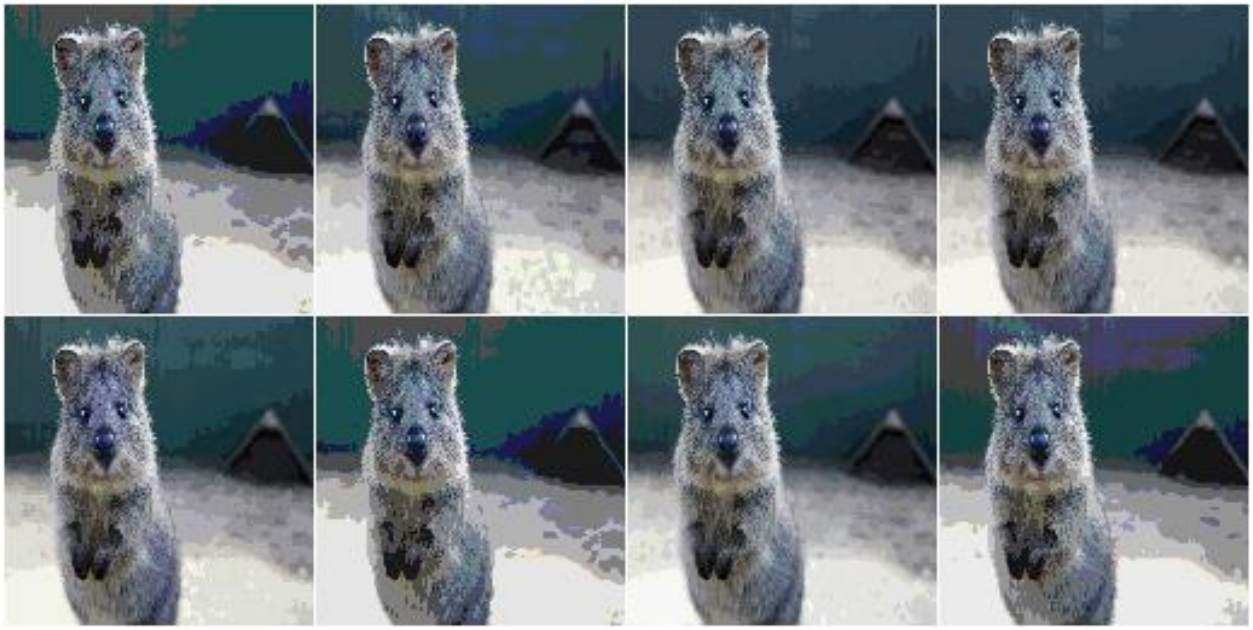
9.6.1 GammaContrast

Adjust image contrast by scaling pixel values to `255*((v/255)**gamma)`.

Values in the range `gamma=(0.5, 2.0)` seem to be sensible.

API link: *`GammaContrast()`*

Example. Modify the contrast of images according to `255*((v/255)**gamma)`, where `v` is a pixel value and `gamma` is sampled uniformly from the interval `[0.5, 2.0]` (once per image):



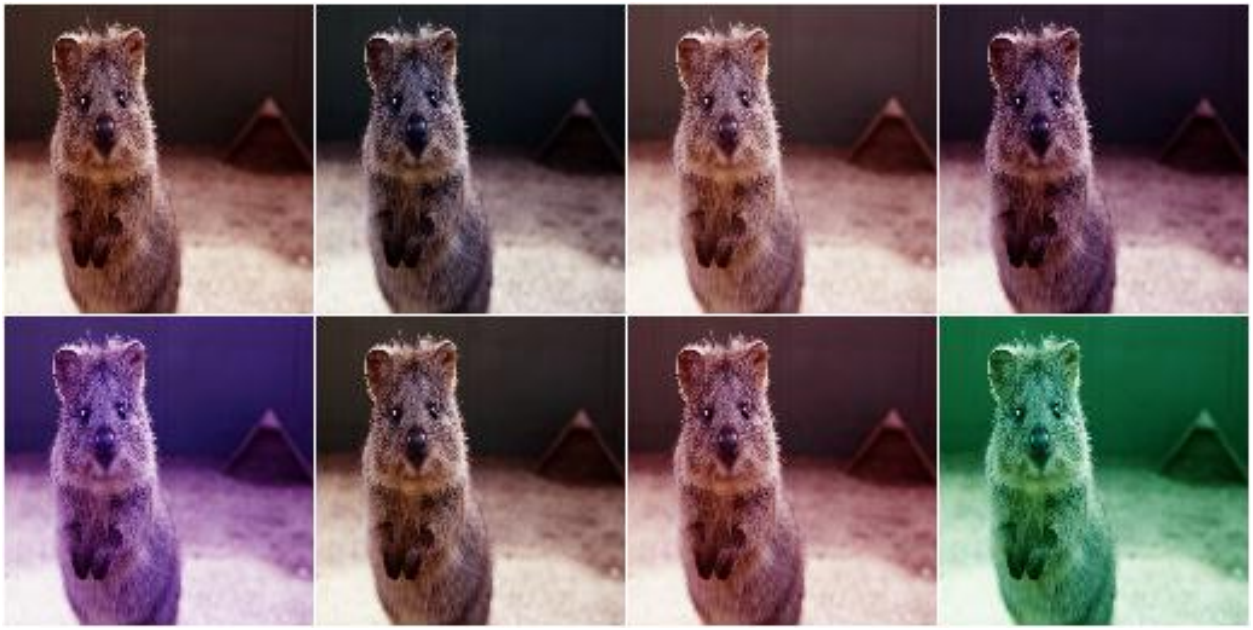
```
import imgaug.augmenters as iaa
aug = iaa.GammaContrast((0.5, 2.0))
```

Example. Same as in the previous example, but `gamma` is sampled once per image *and* channel:

```
aug = iaa.GammaContrast((0.5, 2.0), per_channel=True)
```

9.6.2 SigmoidContrast

Adjust image contrast to `255*1/(1+exp(gain*(cutoff-I_ij/255)))`.



Values in the range `gain=(5, 20)` and `cutoff=(0.25, 0.75)` seem to be sensible.

API link: *SigmoidContrast()*

Example. Modify the contrast of images according to `255*1/(1+exp(gain*(cutoff-v/255)))`, where `v` is a pixel value, `gain` is sampled uniformly from the interval `[3, 10]` (once per image) and `cutoff` is sampled uniformly from the interval `[0.4, 0.6]` (also once per image).

```
import imgaug.augmenters as iaa
aug = iaa.SigmoidContrast(gain=(3, 10), cutoff=(0.4, 0.6))
```

Example. Same as in the previous example, but `gain` and `cutoff` are each sampled once per image *and* channel:

```
aug = iaa.SigmoidContrast(
    gain=(3, 10), cutoff=(0.4, 0.6), per_channel=True)
```

9.6.3 LogContrast

Adjust image contrast by scaling pixels to `255*gain*log_2(1+v/255)`.

This augmenter is fairly similar to `imgaug.augmenters.arithmetic.Multiply`.

API link: *LogContrast()*

Example. Modify the contrast of images according to `255*gain*log_2(1+v/255)`, where `v` is a pixel value and `gain` is sampled uniformly from the interval `[0.6, 1.4]` (once per image):

```
import imgaug.augmenters as iaa
aug = iaa.LogContrast(gain=(0.6, 1.4))
```

Example. Same as in the previous example, but `gain` is sampled once per image *and* channel:

```
aug = iaa.LogContrast(gain=(0.6, 1.4), per_channel=True)
```

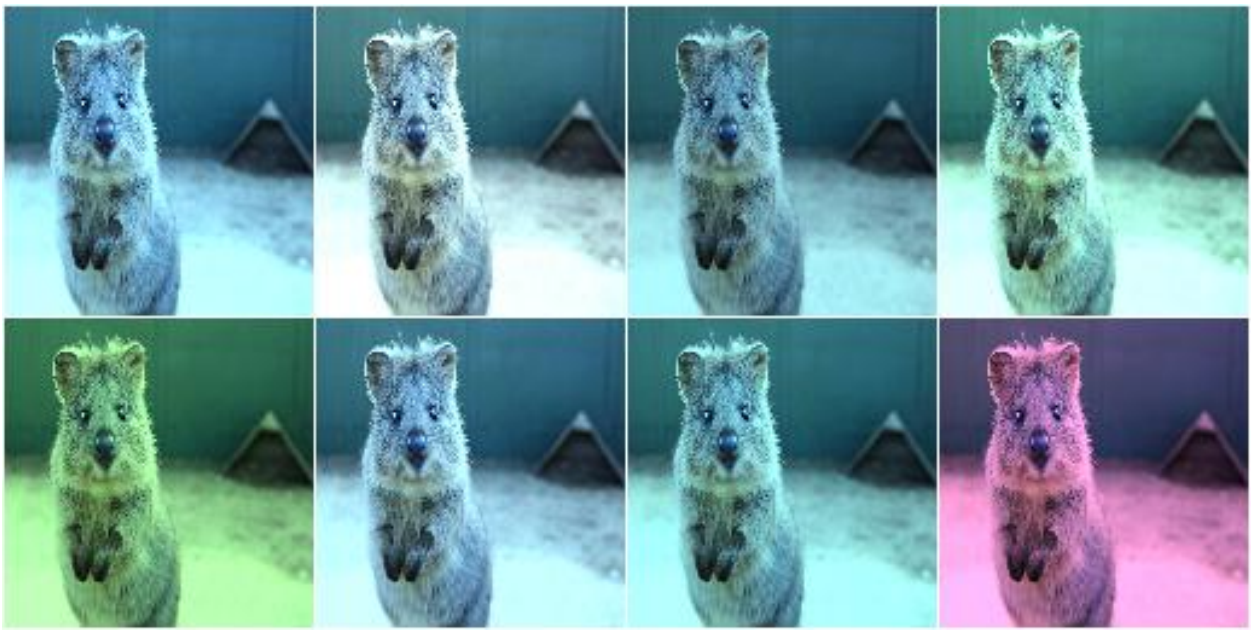
9.6.4 LinearContrast

Adjust contrast by scaling each pixel to `127 + alpha*(v-127)`.

API link: *LinearContrast()*

Example. Modify the contrast of images according to *127 + alpha*(v-127)*‘, where `v` is a pixel value and `alpha` is sampled uniformly from the interval `[0.4, 1.6]` (once per image):

```
import imgaug.augmenters as iaa
aug = iaa.LinearContrast((0.4, 1.6))
```



Example. Same as in the previous example, but `alpha` is sampled once per image *and* channel:

```
aug = iaa.LinearContrast((0.4, 1.6), per_channel=True)
```

9.6.5 AllChannelsCLAHE

Apply CLAHE to all channels of images in their original colorspaces.

CLAHE (Contrast Limited Adaptive Histogram Equalization) performs histogram equilization within image patches, i.e. over local neighbourhoods.

In contrast to `imgaug.augmenters.contrast.CLAHE`, this augmenter operates directly on all channels of the

input images. It does not perform any colorspace transformations and does not focus on specific channels (e.g. `L` in `Lab` colorspace).

API link: *AllChannelsCLAHE*

Example. Create an augmenter that applies CLAHE to all channels of input images:

```
import imgaug.augmenters as iaa
aug = iaa.AllChannelsCLAHE()
```

Example. Same as in the previous example, but the *clip_limit* used by CLAHE is uniformly sampled per image from the interval `[1, 10]`. Some images will therefore have stronger contrast than others (i.e. higher clip limit values).

```
aug = iaa.AllChannelsCLAHE(clip_limit=(1, 10))
```

Example. Same as in the previous example, but the *clip_limit* is sampled per image *and* channel, leading to different levels of contrast for each channel:

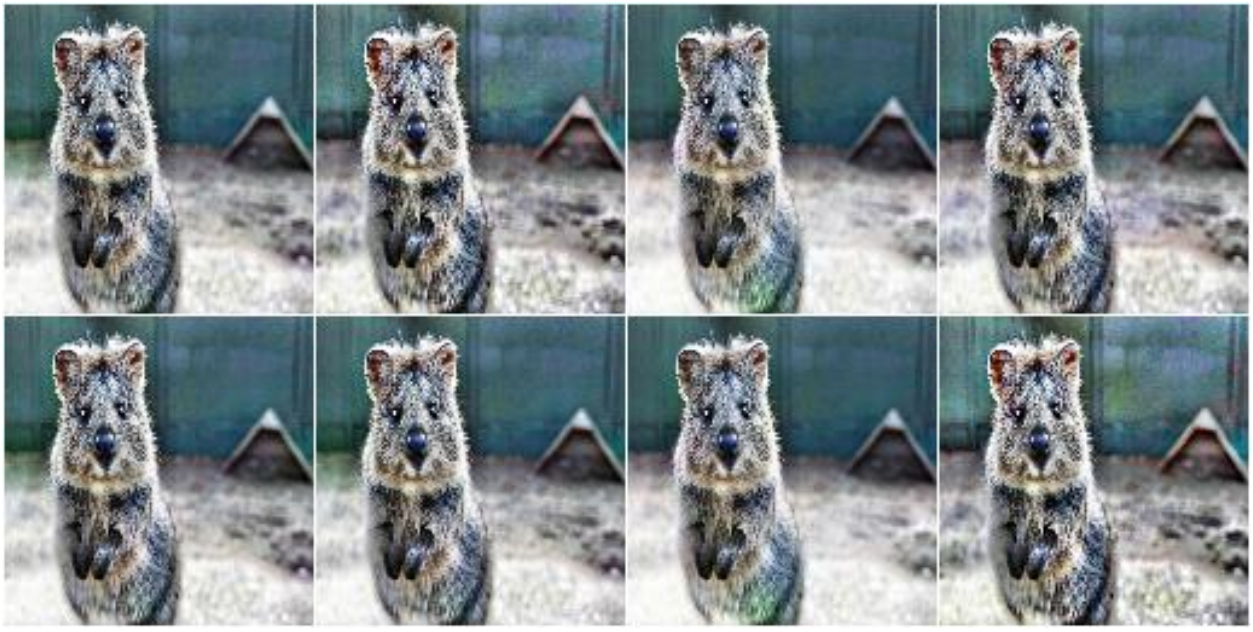
```
aug = iaa.AllChannelsCLAHE(clip_limit=(1, 10), per_channel=True)
```

9.6.6 CLAHE

Apply CLAHE to L/V/L channels in HLS/HSV/Lab colorspaces.

This augmenter applies CLAHE (Contrast Limited Adaptive Histogram Equalization) to images, a form of histogram equalization that normalizes within local image patches. The augmenter transforms input images to a target colorspace (e.g. `Lab`), extracts an intensity-related channel from the converted images (e.g. `L` for `Lab`), applies CLAHE to the channel and then converts the resulting image back to the original colorspace.

Grayscale images (images without channel axis or with only one channel axis) are automatically handled, *from_colorspace* does not have to be adjusted for them. For images with four channels (e.g. `RGBA`), the fourth channel is ignored in the colorspace conversion (e.g. from an `RGBA` image, only the `RGB` part is converted, normalized, converted back and concatenated with the input `A` channel). Images with unusual channel numbers (2, 5 or more than 5) are normalized channel-by-channel (same behaviour as `AllChannelsCLAHE`, though a warning will be raised).



If you want to apply CLAHE to each channel of the original input image's colorspace (without any colorspace conversion), use `imgaug.augmenters.contrast.AllChannelsCLAHE` instead.

API link: *CLAHE*

Example. Create a standard CLAHE augmenter:

```
import imgaug.augmenters as iaa
aug = iaa.CLAHE()
```

Example. Create a CLAHE augmenter with a clip limit uniformly sampled from `[1..10]`, where `1` is rather low contrast and `10` is rather high contrast:

```
aug = iaa.CLAHE(clip_limit=(1, 10))
```

Example. Create a CLAHE augmenter with kernel sizes of `SxS`, where `S` is uniformly sampled from `[3..21]`. Sampling happens once per image.

```
aug = iaa.CLAHE(tile_grid_size_px=(3, 21))
```

Example. Create a CLAHE augmenter with kernel sizes of `SxS`, where `S` is sampled from `N(7, 2)`, but does not go below `3`:

```
import imgaug.parameters as iap
aug = iaa.CLAHE(
    tile_grid_size_px=iap.Discretize(iap.Normal(loc=7, scale=2)),
    tile_grid_size_px_min=3)
```

Example. Create a CLAHE augmenter with kernel sizes of `HxW`, where `H` is uniformly sampled from `[3..21]` and `W` is randomly picked from the list `[3, 5, 7]`:

```
aug = iaa.CLAHE(tile_grid_size_px=((3, 21), [3, 5, 7]))
```

Example. Create a CLAHE augmenter that converts images from BGR colorspace to HSV colorspace and then applies the local histogram equalization to the `V` channel of the images (before converting back to `BGR`). Alternatively,

`Lab` (default) or `HLS` can be used as the target colorspace. Grayscale images (no channels / one channel) are never converted and are instead directly normalized (i.e. *from_colorspace* does not have to be changed for them).

```
aug = iaa.CLAHE(
    from_colorspace=iaa.CLAHE.BGR,
    to_colorspace=iaa.CLAHE.HSV)
```

9.6.7 AllChannelsHistogramEqualization

Apply Histogram Eq. to all channels of images in their original colorspaces.

In contrast to `imgaug.augmenters.contrast.HistogramEqualization`, this augmenter operates directly on all channels of the input images. It does not perform any colorspace transformations and does not focus on specific channels (e.g. `L` in `Lab` colorspace).

API link: *AllChannelsHistogramEqualization*

Example. Create an augmenter that applies histogram equalization to all channels of input images in the original colorspaces:

```
import imgaug.augmenters as iaa
aug = iaa.AllChannelsHistogramEqualization()
```

Example. Same as in the previous example, but alpha-blends the contrast-enhanced augmented images with the original input images using random blend strengths. This leads to random strengths of the contrast adjustment.

```
aug = iaa.Alpha((0.0, 1.0), iaa.AllChannelsHistogramEqualization())
```

9.6.8 HistogramEqualization

Apply Histogram Eq. to L/V/L channels of images in HLS/HSV/Lab colorspaces.

This augmenter is similar to `imgaug.augmenters.contrast.CLAHE`.

The augmenter transforms input images to a target colorspace (e.g. `Lab`), extracts an intensity-related channel from the converted images (e.g. `L` for `Lab`), applies Histogram Equalization to the channel and then converts the resulting image back to the original colorspace.

Grayscale images (images without channel axis or with only one channel axis) are automatically handled, *from_colorspace* does not have to be adjusted for them. For images with four channels (e.g. RGBA), the fourth channel is ignored in the colorspace conversion (e.g. from an `RGBA` image, only the `RGB` part is converted, normalized, converted back and concatenated with the input `A` channel). Images with unusual channel numbers (2, 5 or more than 5) are normalized channel-by-channel (same behaviour as `AllChannelsHistogramEqualization`, though a warning will be raised).

If you want to apply HistogramEqualization to each channel of the original input image's colorspace (without any colorspace conversion), use `imgaug.augmenters.contrast.AllChannelsHistogramEqualization` instead.

API link: *HistogramEqualization*

Example. Create an augmenter that converts images to `HLS`/`HSV`/`Lab` colorspaces, extracts intensity-related channels (i.e. `L`/`V`/`L`), applies histogram equalization to these channels and converts back to the input colorspace:

```
import imgaug.augmenters as iaa
aug = iaa.HistogramEqualization()
```

Example. Same as in the previous example, but alpha blends the result, leading to various strengths of contrast normalization:

```
aug = iaa.Alpha((0.0, 1.0), iaa.HistogramEqualization())
```

Example. Same as in the first example, but the colorspace of input images has to be `BGR` (instead of default `RGB`) and the histogram equalization is applied to the `V` channel in `HSV` colorspace:

```
aug = iaa.HistogramEqualization(
    from_colorspace=iaa.HistogramEqualization.BGR,
    to_colorspace=iaa.HistogramEqualization.HSV)
```

9.7 augmenters.convolutional

9.7.1 Convolve

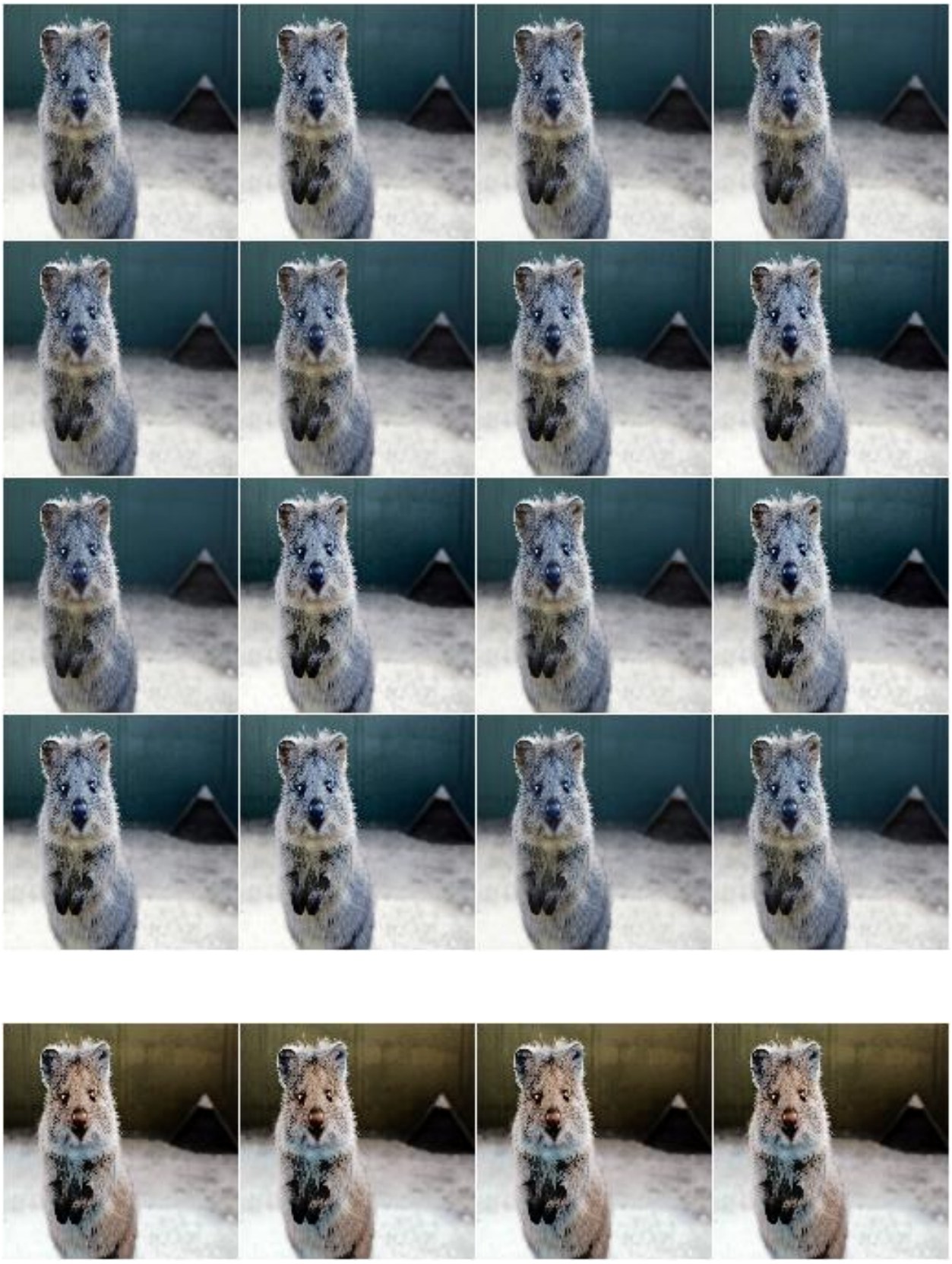
Apply a Convolution to input images.

API link: *Convolve*

Example. Convolve each image with a 3x3 kernel:

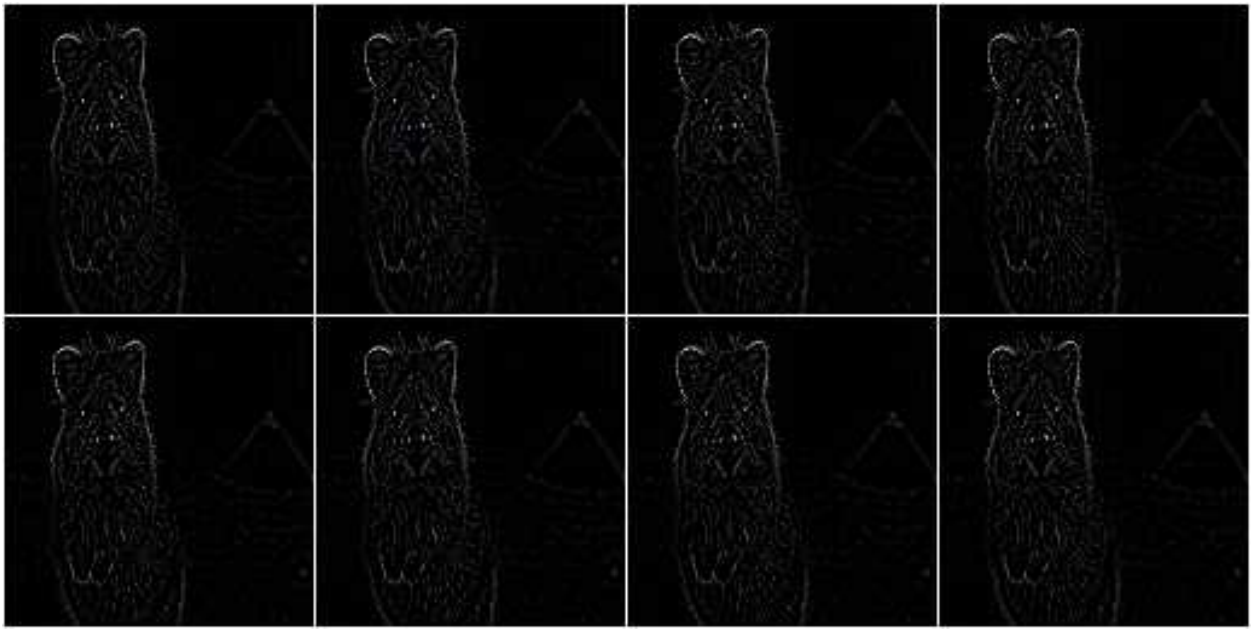
```
import imgaug.augmenters as iaa
matrix = np.array([[0, -1, 0],
```

(continues on next page)



(continued from previous page)

```
                    [-1, 4, -1],
                    [0, -1, 0]])
aug = iaa.Convolve(matrix=matrix)
```



Example. Convolve each image with a 3x3 kernel, which is chosen dynamically per image:

```
def gen_matrix(image, nb_channels, random_state):
      matrix_A = np.array([[0, -1, 0],
                           [-1, 4, -1],
                           [0, -1, 0]])
     matrix_B = np.array([[0, 0, 0],
                          [0, -4, 1],
                          [0, 2, 1]])
      if random_state.rand() < 0.5:
          return [matrix_A] * nb_channels
      else:
          return [matrix_B] * nb_channels
aug = iaa.Convolve(matrix=gen_matrix)
```

9.7.2 Sharpen

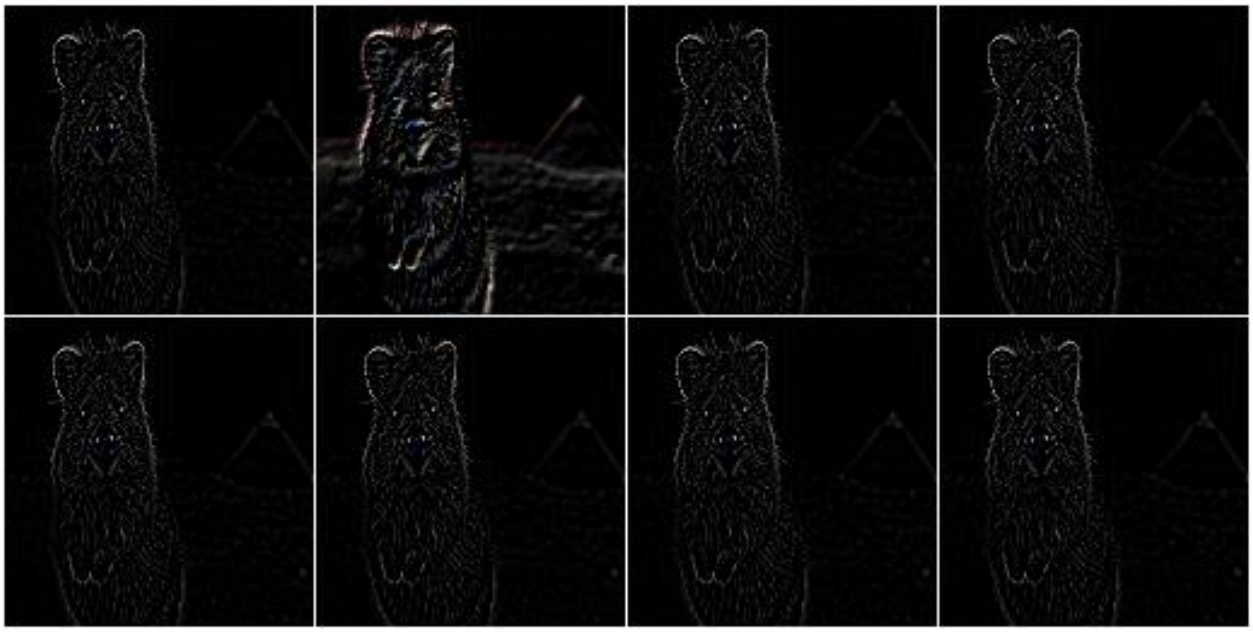
Augmenter that sharpens images and overlays the result with the original image.

API link: *Sharpen()*

Example. Sharpen an image, then overlay the results with the original using an alpha between 0.0 and 1.0:

```
import imgaug.augmenters as iaa
aug = iaa.Sharpen(alpha=(0.0, 1.0), lightness=(0.75, 2.0))
```

Example. Effects of keeping `lightness` fixed at `1.0` and then varying `alpha` between `0.0` and `1.0` in eight steps:



Example. Effects of keeping `alpha` fixed at `1.0` and then varying `lightness` between `0.75` and `1.5` in eight steps:

9.7.3 Emboss

Augmenter that embosses images and overlays the result with the original image.

API link: *Emboss()*

Example. Emboss an image, then overlay the results with the original using an alpha between `0.0` and `1.0`:

```
import imgaug.augmenters as iaa
aug = iaa.Emboss(alpha=(0.0, 1.0), strength=(0.5, 1.5))
```

Example. Effects of keeping `strength` fixed at `1.0` and then varying `alpha` between `0.0` and `1.0` in eight steps:

Example. Effects of keeping `alpha` fixed at `1.0` and then varying `strength` between `0.5` and `1.5` in eight steps:

9.7.4 EdgeDetect

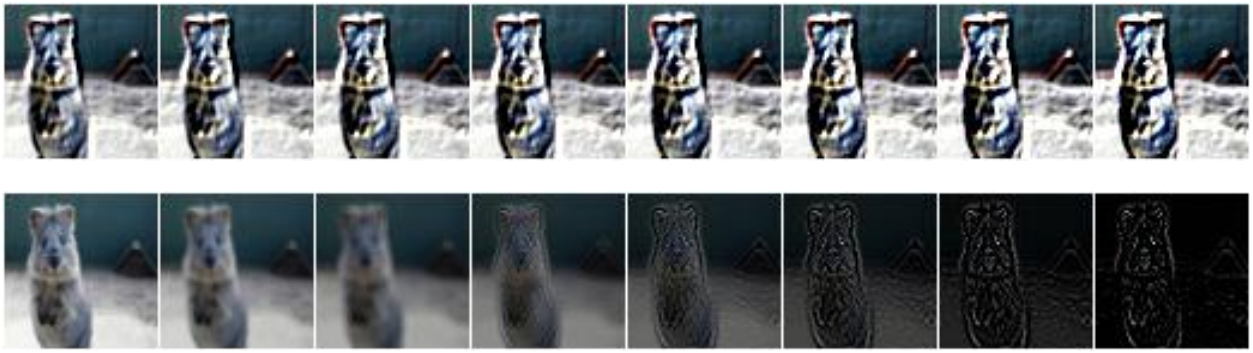
Augmenter that detects all edges in images, marks them in a black and white image and then overlays the result with the original image.

API link: *EdgeDetect()*

Example. Detect edges in images, turning them into black and white images and then overlay these with the original images using random alphas between 0.0 and 1.0:

```
import imgaug.augmenters as iaa
aug = iaa.EdgeDetect(alpha=(0.0, 1.0))
```

Example. Effect of increasing `alpha` from `0.0` to `1.0` in eight steps:



9.7.5 DirectedEdgeDetect

Augmenter that detects edges that have certain directions and marks them in a black and white image and then overlays the result with the original image.

API link: *DirectedEdgeDetect()*

Example. Detect edges having random directions (0 to 360 degrees) in images, turning the images into black and white versions and then overlay these with the original images using random alphas between 0.0 and 1.0:

```
import imgaug.augmenters as iaa
aug = iaa.DirectedEdgeDetect(alpha=(0.0, 1.0), direction=(0.0, 1.0))
```

Example. Effect of fixing `direction` to 0.0 and then increasing `alpha` from `0.0` to `1.0` in eight steps:

Example. Effect of fixing `alpha` to `1.0` and then increasing `direction` from `0.0` to `1.0` (0 to 360 degrees) in eight steps:

9.8 augmenters.edges

9.8.1 Canny

Apply a canny edge detector to input images.

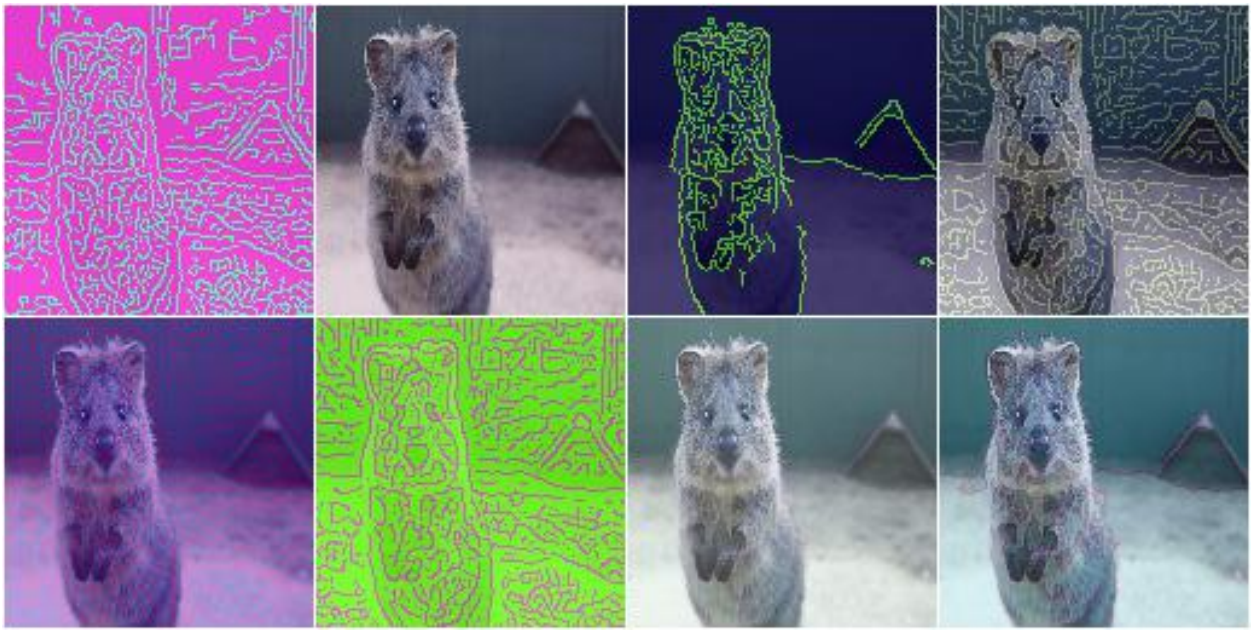
API link: *Canny*

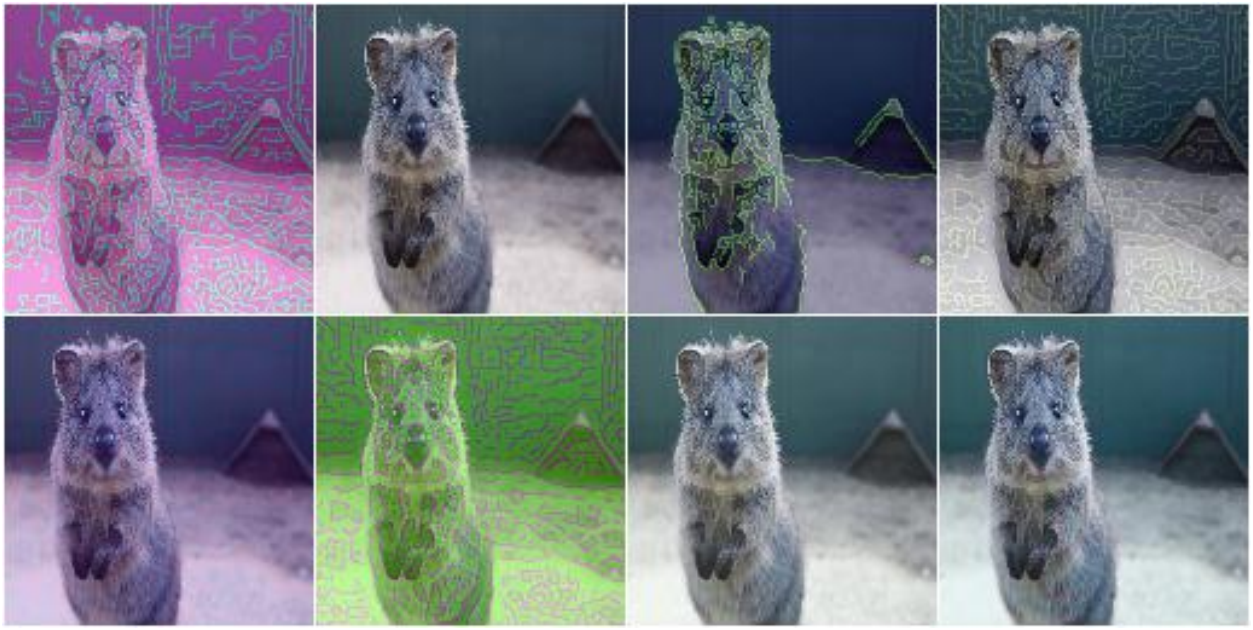
Example. Create an augmenter that generates random blends between images and their canny edge representations:

```
import imgaug.augmenters as iaa
aug = iaa.Canny()
```

Example. Create a canny edge augmenter that generates edge images with a blending factor of max `50%`, i.e. the original (non-edge) image is always at least partially visible:

```
aug = iaa.Canny(alpha=(0.0, 0.5))
```





Example. Same as in the previous example, but the edge image always uses the color white for edges and black for the background:

```
aug = iaa.Canny(
    alpha=(0.0, 0.5),
    colorizer=iaa.RandomColorsBinaryImageColorizer(
        color_true=255,
        color_false=0
    )
)
```

Example. Create a canny edge augmenter that initially preprocesses images using a sobel filter with kernel size of either `3x3` or `13x13` and alpha-blends with result using a strength of `50%` (both images equally visible) to `100%` (only edge image visible).

```
aug = iaa.Canny(alpha=(0.5, 1.0), sobel_kernel_size=[3, 7])
```

Example. Create an augmenter that blends a canny edge image with a median-blurred version of the input image. The median blur uses a fixed kernel size of `13x13` pixels.

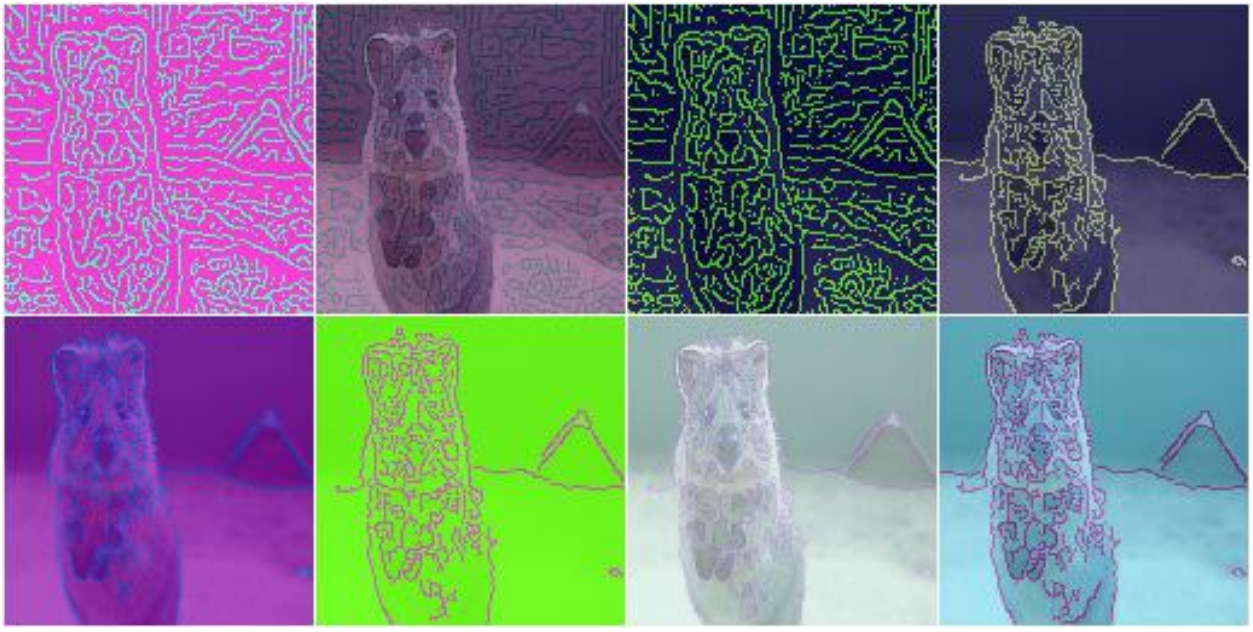
```
aug = iaa.Alpha(
    (0.0, 1.0),
    iaa.Canny(alpha=1),
    iaa.MedianBlur(13)
)
```

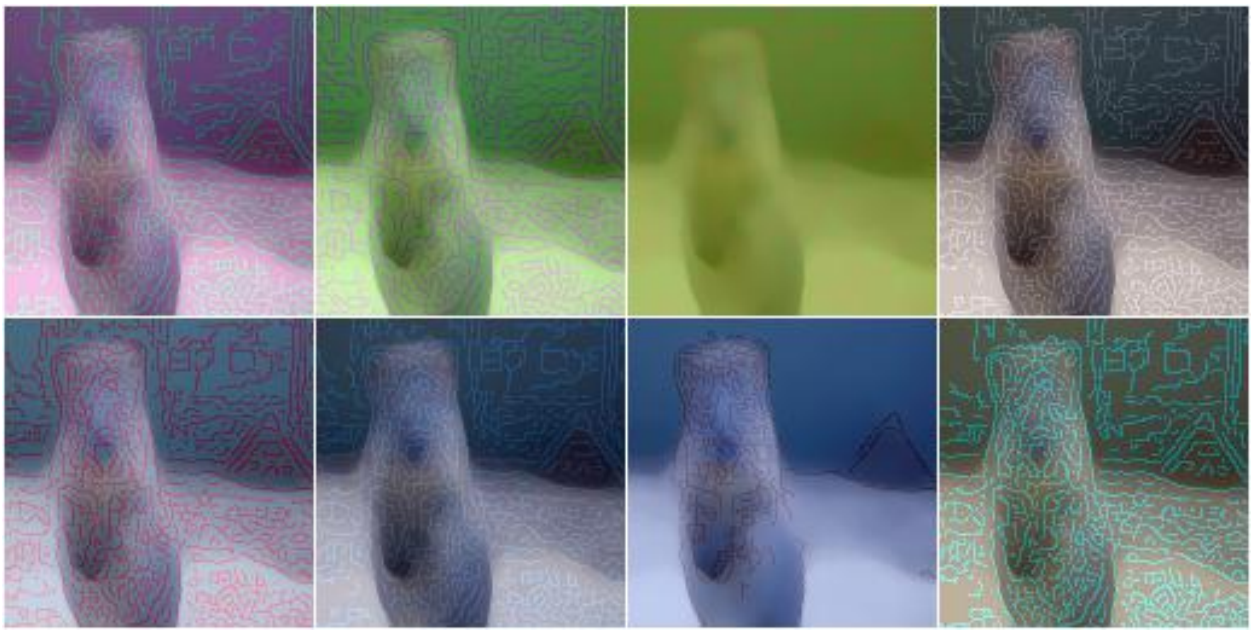
9.9 augmenters.flip

9.9.1 HorizontalFlip

Alias for `Fliplr`.

API link: *HorizontalFlip*





9.9.2 VericalFlip

Alias for `Flipud`.

API link: *VerticalFlip*

9.9.3 Fliplr

Flip/mirror input images horizontally.

Note: The default value for the probability is `0.0`. So, to flip *all* input image use `Fliplr(1.0)` and *not* just `Fliplr()`.

API link: *Fliplr*

Example. Flip 50% of all images horizontally:

```
import imgaug.augmenters as iaa
aug = iaa.Fliplr(0.5)
```

9.9.4 Flipud

Flip/mirror input images vertically.

Note: The default value for the probability is `0.0`. So, to flip *all* input image use `Flipud(1.0)` and *not* just `Flipud()`.

API link: *Flipud*

Example. Flip 50% of all images vertically:

```
aug = iaa.Flipud(0.5)
```

9.10 augmenters.geometric

9.10.1 Affine

Augmenter to apply affine transformations to images.

API link: *Affine*

Example. Scale images to a value of 50 to 150% of their original size:

```
import imgaug.augmenters as iaa
aug = iaa.Affine(scale=(0.5, 1.5))
```

Example. Scale images to a value of 50 to 150% of their original size, but do this independently per axis (i.e. sample two values per image):

```
aug = iaa.Affine(scale={"x": (0.5, 1.5), "y": (0.5, 1.5)})
```

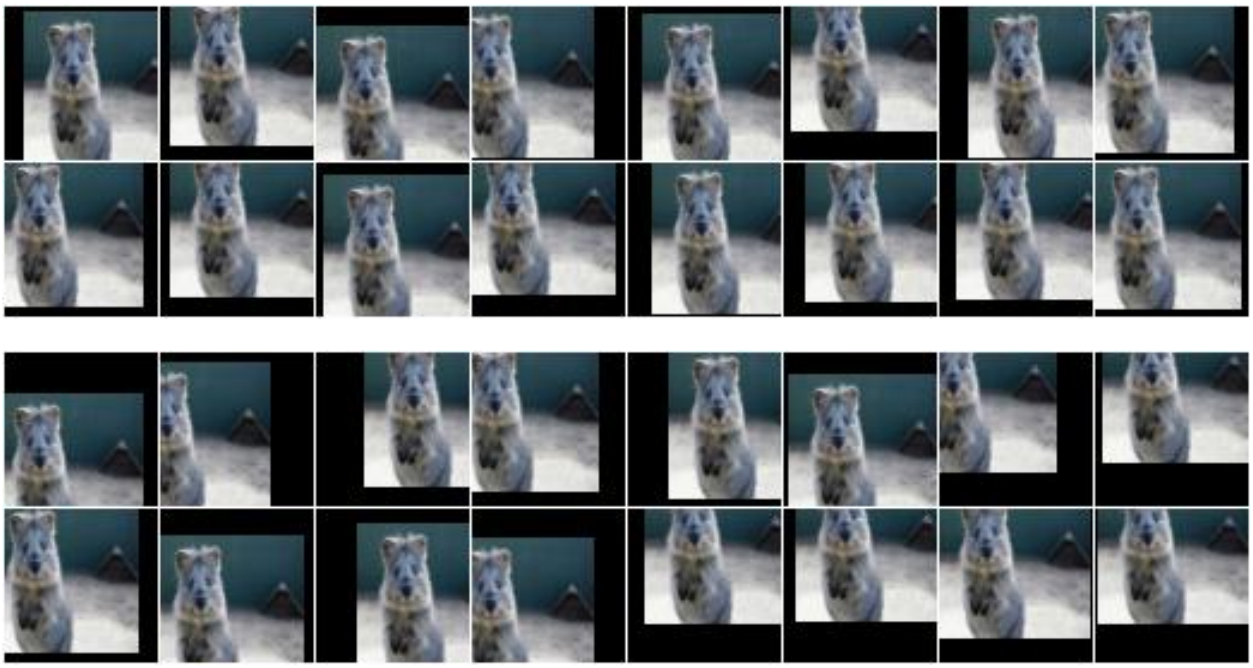
Example. Translate images by -20 to +20% on x- and y-axis independently:

```
aug = iaa.Affine(translate_percent={"x": (-0.2, 0.2), "y": (-0.2, 0.2)})
```

Example. Translate images by -20 to 20 pixels on x- and y-axis independently:

```
aug = iaa.Affine(translate_px={"x": (-20, 20), "y": (-20, 20)})
```

Example. Rotate images by -45 to 45 degrees:



```
aug = iaa.Affine(rotate=(-45, 45))
```

Example. Shear images by -16 to 16 degrees:

```
aug = iaa.Affine(shear=(-16, 16))
```

Example. When applying affine transformations, new pixels are often generated, e.g. when translating to the left, pixels are generated on the right. Various modes exist to set how these pixels are ought to be filled. Below code shows an example that uses all modes, sampled randomly per image. If the mode is `constant` (fill all with one constant value), then a random brightness between 0 and 255 is used:

```
aug = iaa.Affine(translate_percent={"x": -0.20}, mode=ia.ALL, cval=(0, 255))
```

9.10.2 PiecewiseAffine

Apply affine transformations that differ between local neighbourhoods.

This augmenter places a regular grid of points on an image and randomly moves the neighbourhood of these point around via affine transformations. This leads to local distortions.

This is mostly a wrapper around scikit-image's `PiecewiseAffine`. See also `Affine` for a similar technique.

Note: This augmenter is very slow. See *Performance*. Try to use `ElasticTransformation` instead, which is at least 10x faster.

Note: For coordinate-based inputs (keypoints, bounding boxes, polygons, ...), this augmenter still has to perform an image-based augmentation, which will make it significantly slower for such inputs than other augmenters. See *Performance*.

API link: *PiecewiseAffine*

Example. Distort images locally by moving points around, each with a distance v (percent relative to image size), where v is sampled per point from `N(0, z)` `z` is sampled per image from the range 0.01 to 0.05:

```
import imgaug.augmenters as iaa
aug = iaa.PiecewiseAffine(scale=(0.01, 0.05))
```

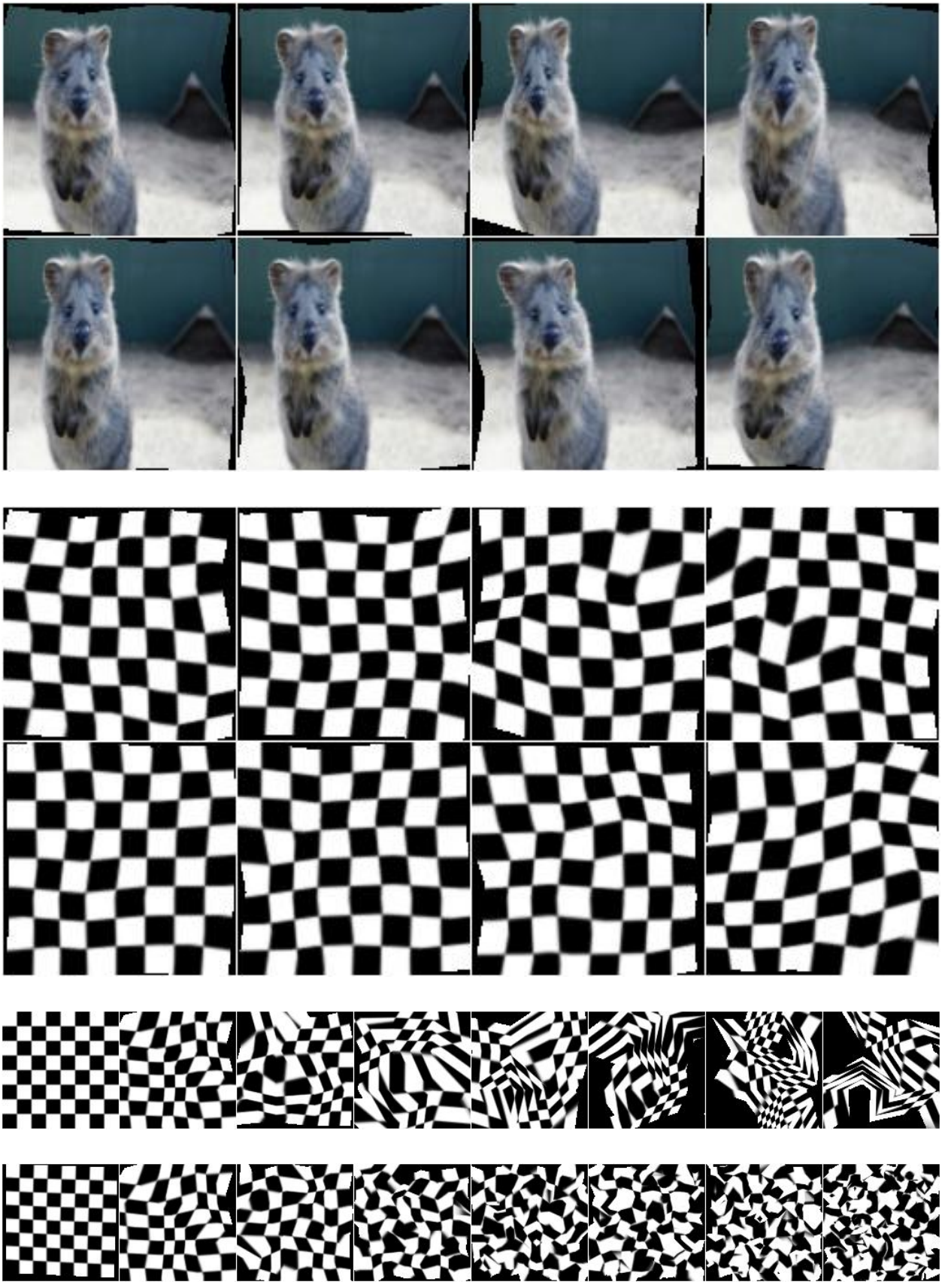
Example. Effect of increasing `scale` from `0.01` to `0.3` in eight steps:

Example. PiecewiseAffine works by placing a regular grid of points on the image and moving them around. By default this grid consists of 4x4 points. The below image shows the effect of increasing that value from 2x2 to 16x16 in 8 steps:

9.10.3 PerspectiveTransform

Apply random four point perspective transformations to images.

Each of the four points is placed on the image using a random distance from its respective corner. The distance is sampled from a normal distribution. As a result, most transformations don't change the image very much, while some "focus" on polygons far inside the image.



The results of this augmenter have some similarity with `Crop`.

API link: *PerspectiveTransform*

Example. Apply perspective transformations using a random scale between `0.01` and `0.15` per image, where the scale is roughly a measure of how far the perspective transformation's corner points may be distanced from the image's corner points:

```
import imgaug.augmenters as iaa
aug = iaa.PerspectiveTransform(scale=(0.01, 0.15))
```

Example. Same as in the previous example, but images are not resized back to the input image size after augmentation. This will lead to smaller output images.

```
aug = iaa.PerspectiveTransform(scale=(0.01, 0.15), keep_size=False)
```

9.10.4 ElasticTransformation

Transform images by moving pixels locally around using displacement fields.

The augmenter has the parameters `alpha` and `sigma`. `alpha` controls the strength of the displacement: higher values mean that pixels are moved further. `sigma` controls the smoothness of the displacement: higher values lead to smoother patterns – as if the image was below water – while low values will cause indivdual pixels to be moved very differently from their neighbours, leading to noisy and pixelated images.

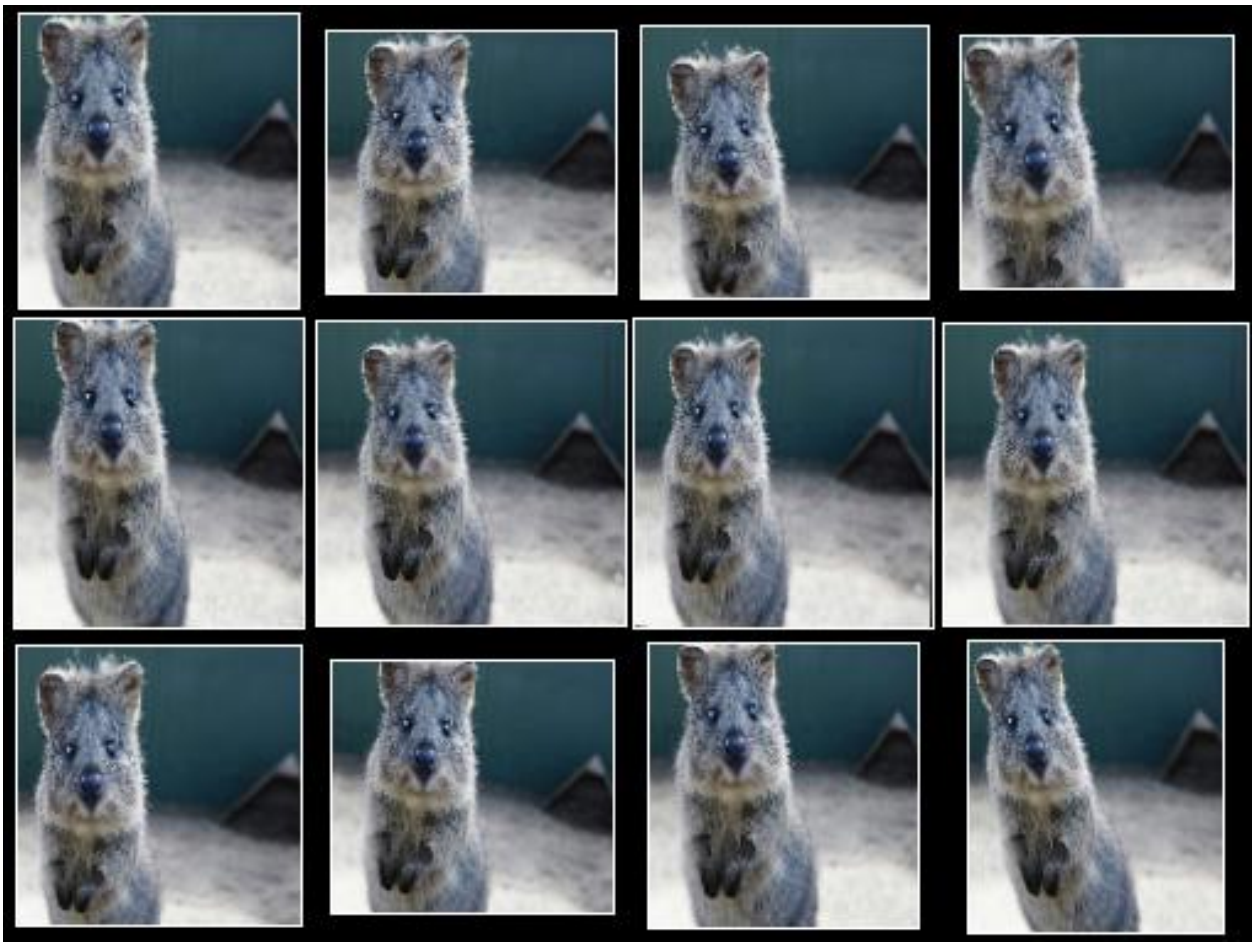


Fig. 1: `PerspectiveTransform` with `keep_size` set to `False`. Note that the individual images are here padded after augmentation in order to align them in a grid (i.e. purely for visualization purposes).

A relation of 10:1 seems to be good for `alpha` and `sigma`, e.g. `alpha=10` and `sigma=1` or `alpha=50`, `sigma=5`. For `128x128` a setting of `alpha=(0, 70.0)`, `sigma=(4.0, 6.0)` may be a good choice and will lead to a water-like effect.

For a detailed explanation, see

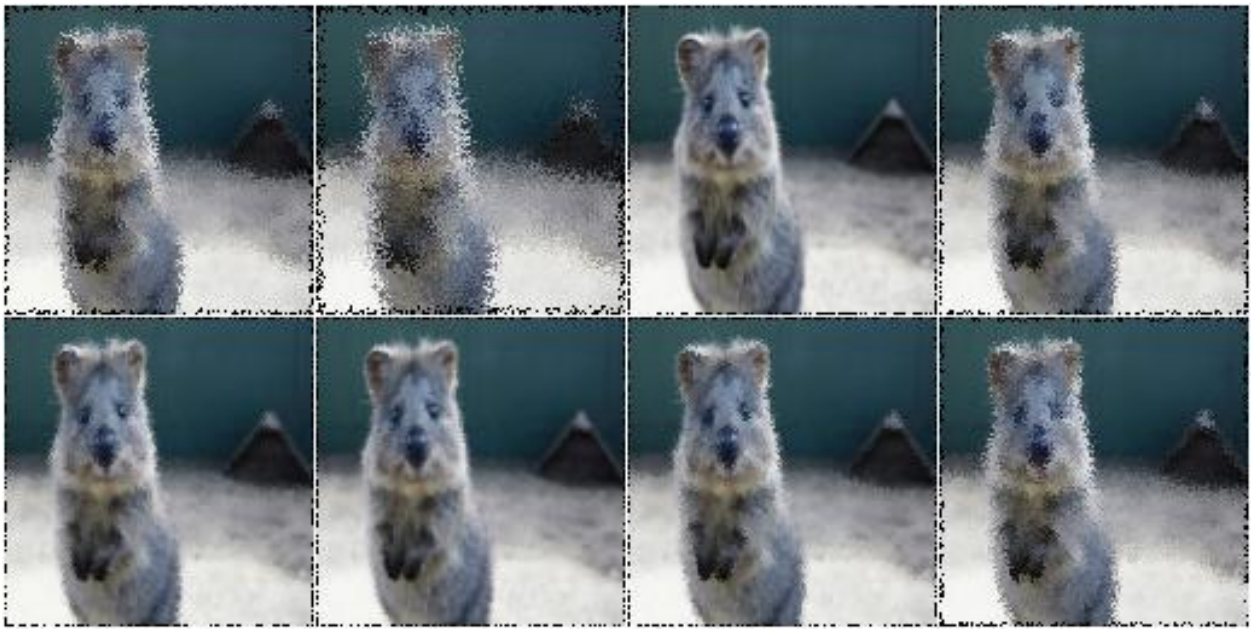
```
Simard, Steinkraus and Platt
Best Practices for Convolutional Neural Networks applied to Visual
Document Analysis
in Proc. of the International Conference on Document Analysis and
Recognition, 2003
```

Note: For coordinate-based inputs (keypoints, bounding boxes, polygons, ...), this augmenter still has to perform an image-based augmentation, which will make it significantly slower for such inputs than other augmenters. See *Performance*.

API link: *ElasticTransformation*

Example. Distort images locally by moving individual pixels around following a distortions field with strength 0.25. The strength of the movement is sampled per pixel from the range 0 to 5.0:

```
import imgaug.augmenters as iaa
aug = iaa.ElasticTransformation(alpha=(0, 5.0), sigma=0.25)
```



Example. Effect of keeping sigma fixed at `0.25` and increasing alpha from `0` to `5.0` in eight steps:

Example. Effect of keeping alpha fixed at `2.5` and increasing sigma from `0.01` to `1.0` in eight steps:

9.10.5 Rot90

Rotate images clockwise by multiples of 90 degrees.

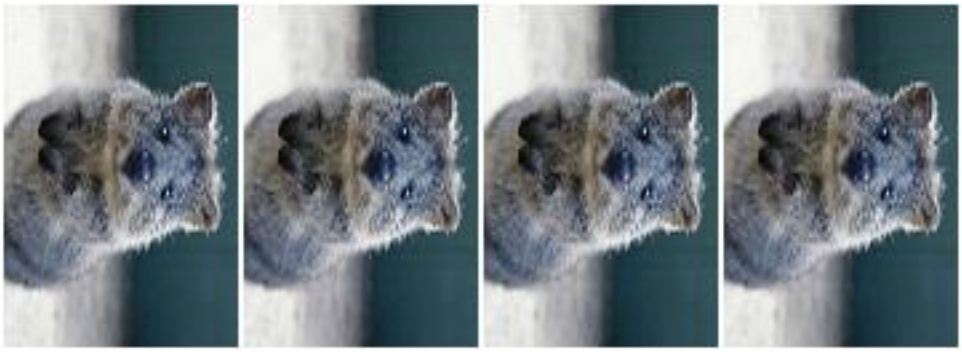
This could also be achieved using `Affine`, but `Rot90` is significantly more efficient.

API link: *Rot90*

Fig. 2: The below examples use this input image, which slightly deviates from the examples for other augmenters (i.e. it is not square).

Example. Rotate all images by 90 degrees. Resize these images afterwards to keep the size that they had before augmentation. This may cause the images to look distorted.

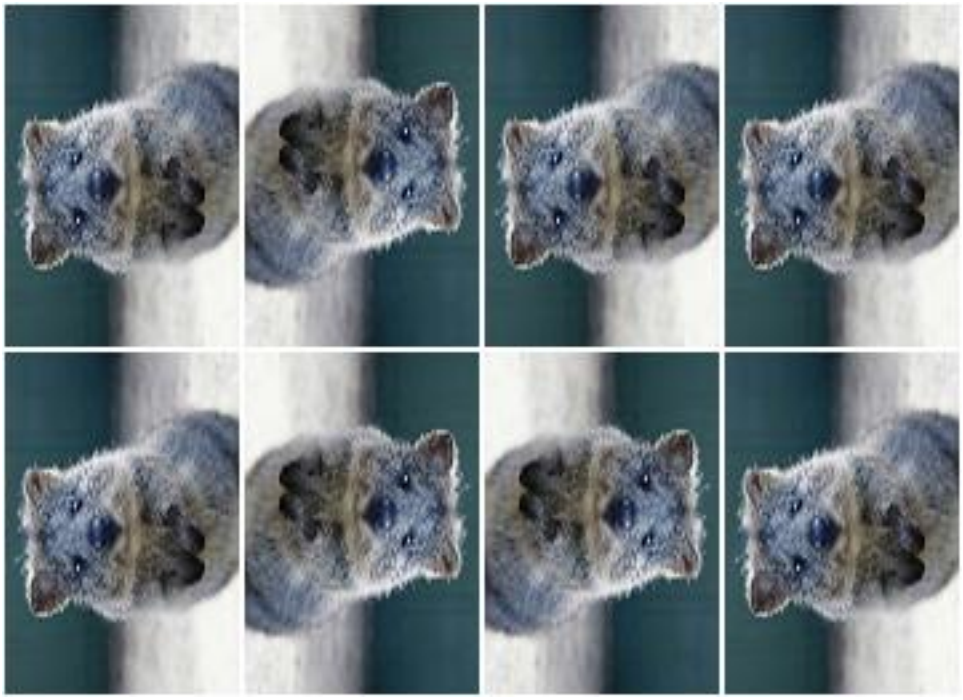
```
import imgaug.augmenters as iaa
aug = iaa.Rot90(1)
```



Example. Rotate all images by 90 or 270 degrees. Resize these images afterwards to keep the size that they had before augmentation. This may cause the images to look distorted.

```
aug = iaa.Rot90([1, 3])
```

Example. Rotate all images by 90, 180 or 270 degrees. Resize these images afterwards to keep the size that they had before augmentation. This may cause the images to look distorted.



```
aug = iaa.Rot90((1, 3))
```

Example. Rotate all images by 90, 180 or 270 degrees. Does not resize to the original image size afterwards, i.e. each image's size may change.

```
aug = iaa.Rot90((1, 3), keep_size=False)
```

9.11 augmenters.pooling

9.11.1 AveragePooling

Apply average pooling to images.

This augmenter pools images with kernel sizes `H x W` by averaging the pixel values within these windows. For e.g. `2 x 2` this halves the image size. Optionally, the augmenter will automatically re-upscale the image to the input size (by default this is activated).

This augmenter does not affect heatmaps, segmentation maps or coordinates-based augmentables (e.g. keypoints, bounding boxes, ...).

Note that this augmenter is very similar to `AverageBlur`. `AverageBlur` applies averaging within windows of given kernel size *without* striding, while `AveragePooling` applies striding corresponding to the kernel size, with optional upscaling afterwards. The upscaling is configured to create "pixelated"/"blocky" images by default.

API link: *AveragePooling*

Example. Create an augmenter that always pools with a kernel size of `2 x 2`:

```
import imgaug.augmenters as iaa
aug = iaa.AveragePooling(2)
```

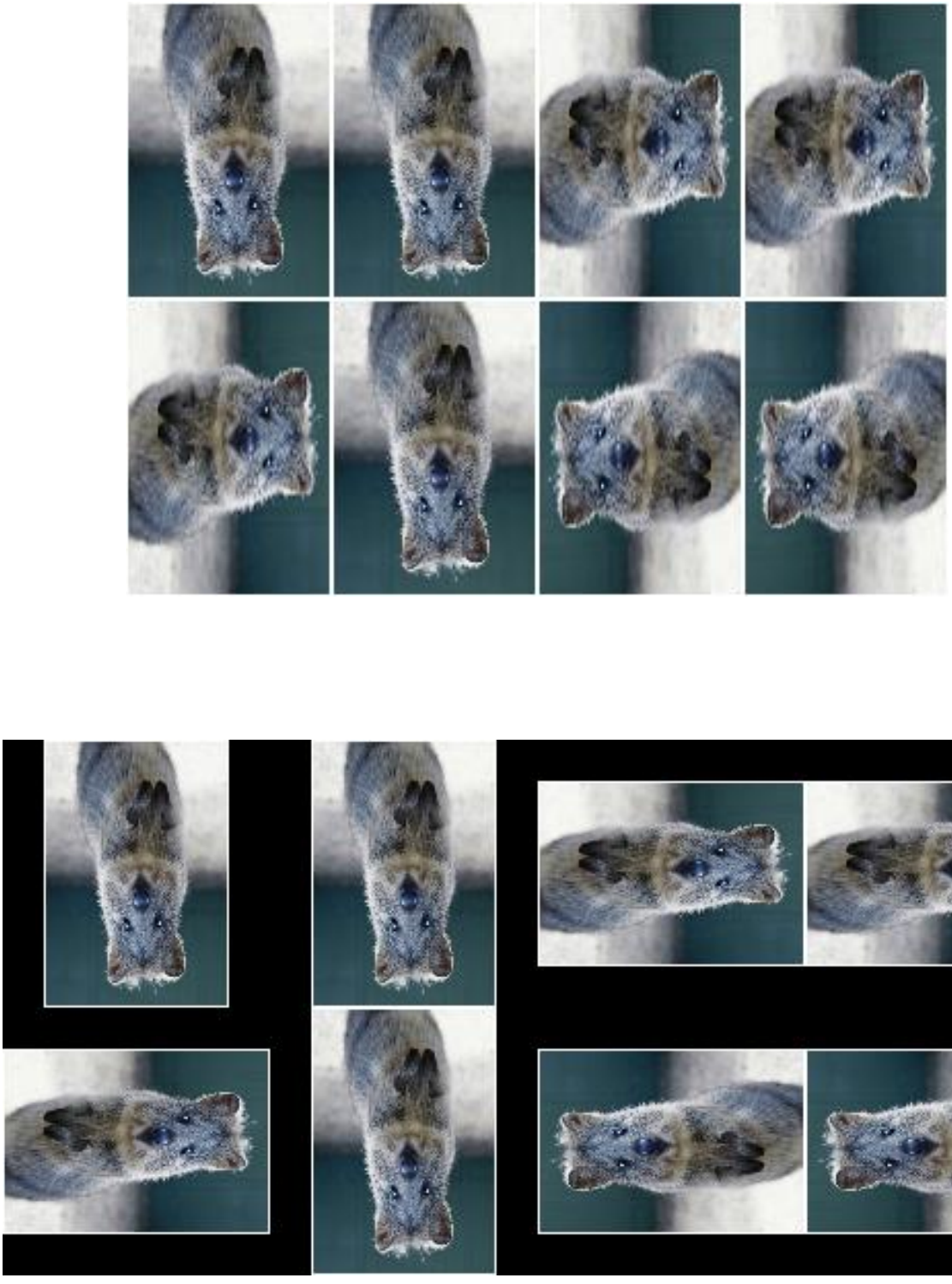


Fig. 3: `Rot90` with `keep_size` set to `False`. Note that the individual images are here padded after augmentation in order to align them in a grid (i.e. purely for visualization purposes).

Example. Create an augmenter that always pools with a kernel size of `2 x 2` and does *not* resize back to the input image size, i.e. the resulting images have half the resolution:

```
aug = iaa.AveragePooling(2, keep_size=False)
```

Example. Create an augmenter that always pools either with a kernel size of `2 x 2` or `8 x 8`:

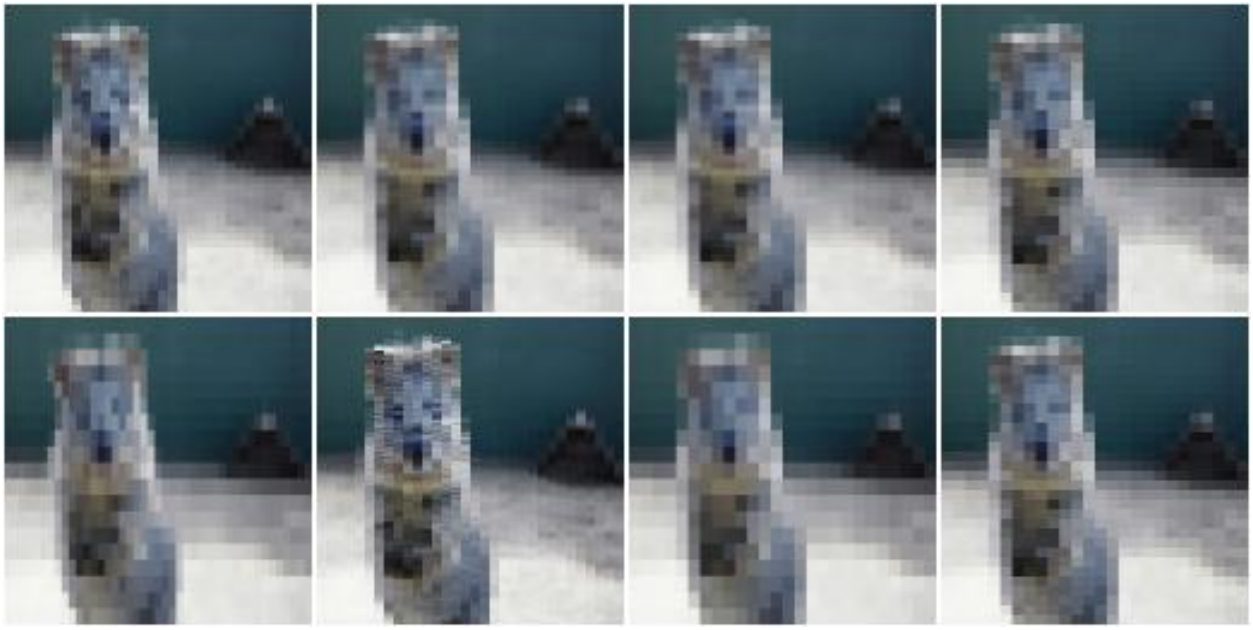
```
aug = iaa.AveragePooling([2, 8])
```

Example. Create an augmenter that always pools with a kernel size of `1 x 1` (does nothing) to `7 x 7`. The kernel sizes are always symmetric.

```
aug = iaa.AveragePooling((1, 7))
```

Example. Create an augmenter that always pools with a kernel size of `H x W` where `H` and `W` are both sampled independently from the range `[1..7]`. E.g. resulting kernel sizes could be `3 x 7` or `5 x 1`.

```
aug = iaa.AveragePooling(((1, 7), (1, 7)))
```



9.11.2 MaxPooling

Apply max pooling to images.

This augmenter pools images with kernel sizes `H x W` by taking the maximum pixel value over windows. For e.g. `2 x 2` this halves the image size. Optionally, the augmenter will automatically re-upscale the image to the input size (by default this is activated).

The maximum within each pixel window is always taken channelwise.

This augmenter does not affect heatmaps, segmentation maps or coordinates-based augmentables (e.g. keypoints, bounding boxes, ...).

API link: *MaxPooling*

Example. Create an augmenter that always pools with a kernel size of `2 x 2`:

```
import imgaug.augmenters as iaa
aug = iaa.MaxPooling(2)
```

Example. Create an augmenter that always pools with a kernel size of `2 x 2` and does *not* resize back to the input image size, i.e. the resulting images have half the resolution:

```
aug = iaa.MaxPooling(2, keep_size=False)
```

Example. Create an augmenter that always pools either with a kernel size of `2 x 2` or `8 x 8`:

```
aug = iaa.MaxPooling([2, 8])
```

Example. Create an augmenter that always pools with a kernel size of `1 x 1` (does nothing) to `7 x 7`. The kernel sizes are always symmetric.

```
aug = iaa.MaxPooling((1, 7))
```

Example. Create an augmenter that always pools with a kernel size of `H x W` where `H` and `W` are both sampled independently from the range `[1..7]`. E.g. resulting kernel sizes could be `3 x 7` or `5 x 1`.

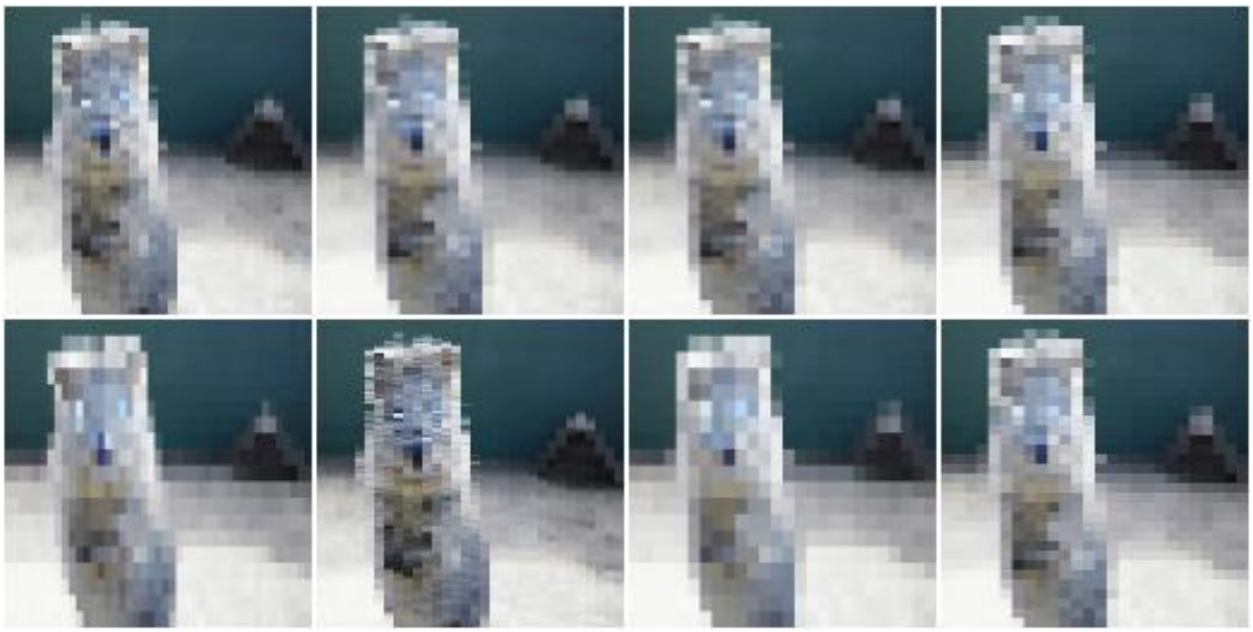
```
aug = iaa.MaxPooling(((1, 7), (1, 7)))
```

9.11.3 MinPooling

Apply minimum pooling to images.

This augmenter pools images with kernel sizes `H x W` by taking the minimum pixel value over windows. For e.g. `2 x 2` this halves the image size. Optionally, the augmenter will automatically re-upscale the image to the input size (by default this is activated).

The minimum within each pixel window is always taken channelwise.



This augmenter does not affect heatmaps, segmentation maps or coordinates-based augmentables (e.g. keypoints, bounding boxes, ...).

API link: *MinPooling*

Example. Create an augmenter that always pools with a kernel size of `2 x 2`:

```
import imgaug.augmenters as iaa
aug = iaa.MinPooling(2)
```

Example. Create an augmenter that always pools with a kernel size of `2 x 2` and does *not* resize back to the input image size, i.e. the resulting images have half the resolution.

```
aug = iaa.MinPooling(2, keep_size=False)
```

Example. Create an augmenter that always pools either with a kernel size of `2 x 2` or `8 x 8`:

```
aug = iaa.MinPooling([2, 8])
```

Example. Create an augmenter that always pools with a kernel size of `1 x 1` (does nothing) to `7 x 7`. The kernel sizes are always symmetric.

```
aug = iaa.MinPooling((1, 7))
```

Example. Create an augmenter that always pools with a kernel size of `H x W` where `H` and `W` are both sampled independently from the range `[1..7]`. E.g. resulting kernel sizes could be `3 x 7` or `5 x 1`.

```
aug = iaa.MinPooling(((1, 7), (1, 7)))
```

9.11.4 MedianPooling

Apply median pooling to images.

This augmenter pools images with kernel sizes `H x W` by taking the median pixel value over windows. For e.g. `2 x 2` this halves the image size. Optionally, the augmenter will automatically re-upscale the image to the input size (by default this is activated).

The median within each pixel window is always taken channelwise.

This augmenter does not affect heatmaps, segmentation maps or coordinates-based augmentables (e.g. keypoints, bounding boxes, ...).

API link: *MedianPooling*

Example. Create an augmenter that always pools with a kernel size of `2 x 2`:

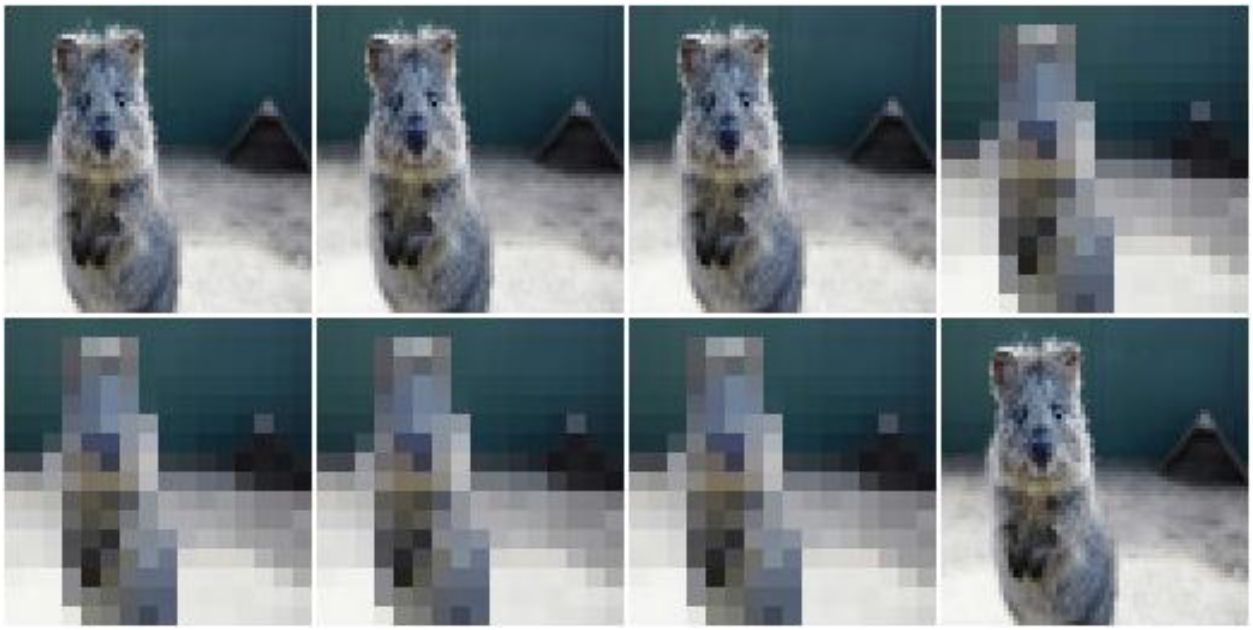
```
import imgaug.augmenters as iaa
aug = iaa.MedianPooling(2)
```

Example. Create an augmenter that always pools with a kernel size of `2 x 2` and does *not* resize back to the input image size, i.e. the resulting images have half the resolution:

```
aug = iaa.MedianPooling(2, keep_size=False)
```

Example. Create an augmenter that always pools either with a kernel size of `2 x 2` or `8 x 8`:

```
aug = iaa.MedianPooling([2, 8])
```



Example. Create an augmenter that always pools with a kernel size of `1 x 1` (does nothing) to `7 x 7`. The kernel sizes are always symmetric.

```
aug = iaa.MedianPooling((1, 7))
```

Example. Create an augmenter that always pools with a kernel size of `H x W` where `H` and `W` are both sampled independently from the range `[1..7]`. E.g. resulting kernel sizes could be `3 x 7` or `5 x 1`.

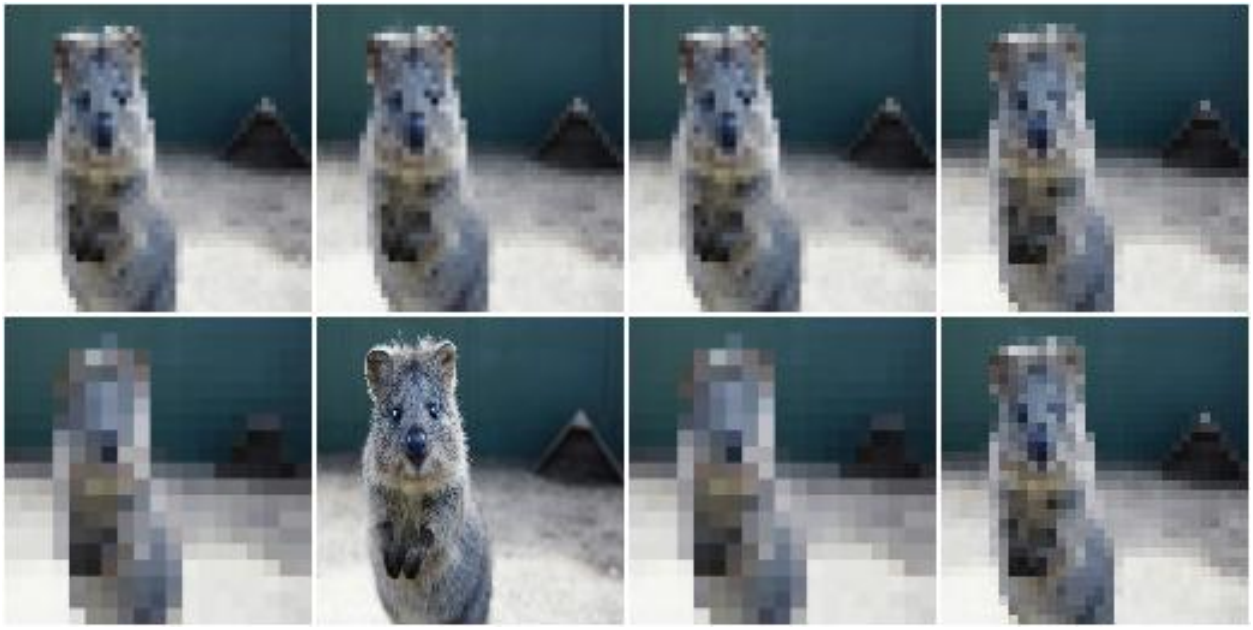
```
aug = iaa.MedianPooling(((1, 7), (1, 7)))
```

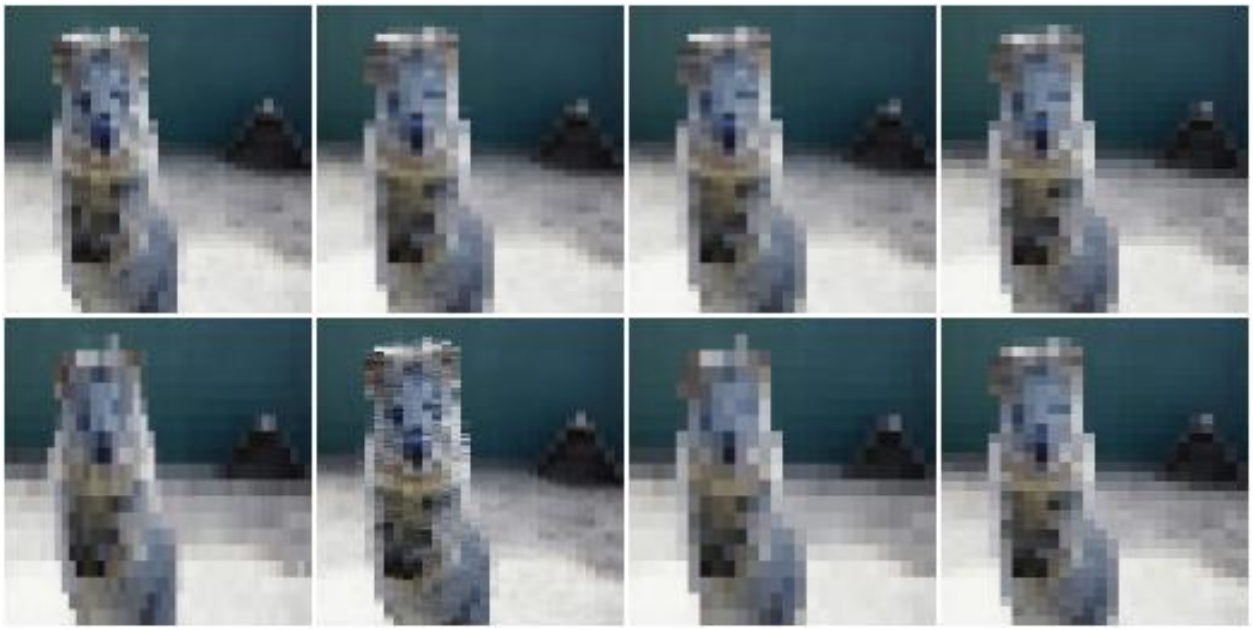
9.12 augmenters.segmentation

9.12.1 Superpixels

Completely or partially transform images to their superpixel representation.

Note: This augmenter is fairly slow. See *Performance*.





API link: *Superpixels*

Example. Generate about `64` superpixels per image. Replace each one with a probability of `50%` by its average pixel color.

```
import imgaug.augmenters as iaa
aug = iaa.Superpixels(p_replace=0.5, n_segments=64)
```

Example. Generate `16` to `128` superpixels per image. Replace each superpixel with a probability between `10` and `100%` (sampled once per image) by its average pixel color.

```
aug = iaa.Superpixels(p_replace=(0.1, 1.0), n_segments=(16, 128))
```

Example. Effect of setting `n_segments` to a fixed value of `64` and then increasing `p_replace` from `0.0` and `1.0`:

Example. Effect of setting `p_replace` to a fixed value of `1.0` and then increasing `n_segments` from `1*16` to `9*16=144`:

9.12.2 Voronoi

Average colors of an image within Voronoi cells.

This augmenter performs the following steps:

1. Query *points_sampler* to sample random coordinates of cell centers. On the image.
2. Estimate for each pixel to which voronoi cell (i.e. segment) it belongs. Each pixel belongs to the cell with the closest center coordinate (euclidean distance).
3. Compute for each cell the average color of the pixels within it.
4. Replace the pixels of *p_replace* percent of all cells by their average color. Do not change the pixels of `(1 - p_replace)` percent of all cells. (The percentages are average values over many images. Some images may get more/less cells replaced by their average color.)

API link: *Voronoi*

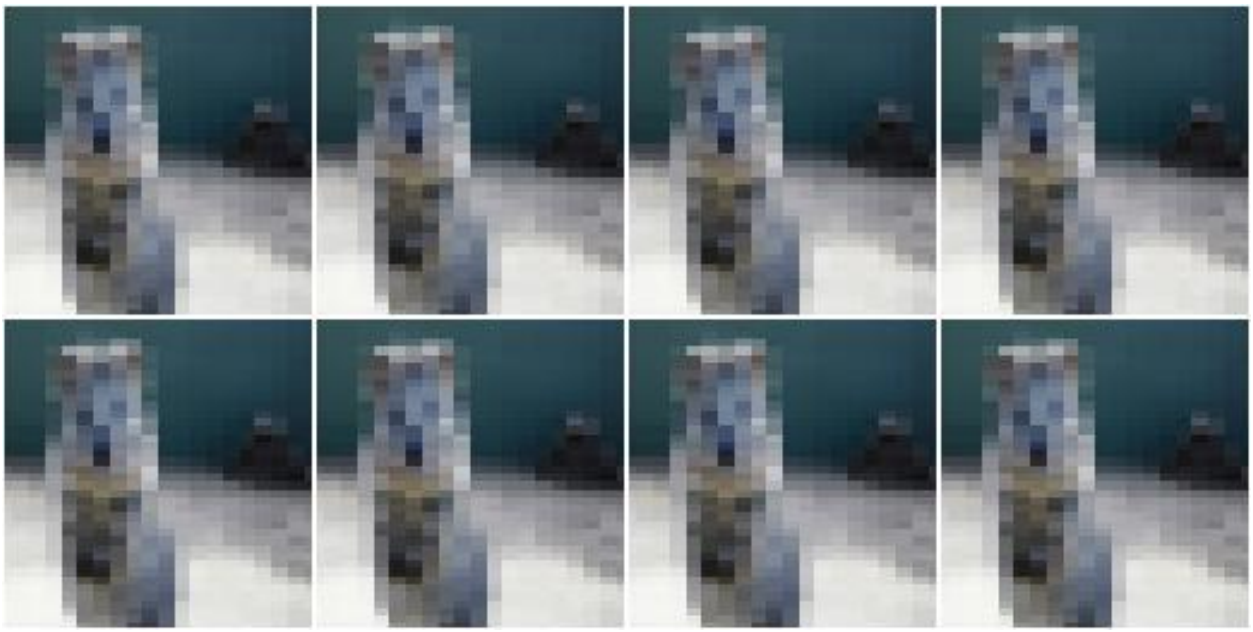
Example. Create an augmenter that places a `20x40` (`HxW`) grid of cells on the image and replaces all pixels within each cell by the cell's average color. The process is performed at an image size not exceeding 128px on any side. If necessary, the downscaling is performed using linear interpolation.

```
import imgaug.augmenters as iaa
points_sampler = iaa.RegularGridPointsSampler(n_cols=20, n_rows=40)
aug = iaa.Voronoi(points_sampler)
```

Example. Create a voronoi augmenter that generates a grid of cells dynamically adapted to the image size. Larger images get more cells. On the x-axis, the distance between two cells is `w * W` pixels, where `W` is the width of the image and `w` is always `0.1`. On the y-axis, the distance between two cells is `h * H` pixels, where `H` is the height of the image and `h` is sampled uniformly from the interval `[0.05, 0.2]`. To make the voronoi pattern less regular, about `20` percent of the cell coordinates are randomly dropped (i.e. the remaining cells grow in size). In contrast to the first example, the image is not resized (if it was, the sampling would happen *after* the resizing, which would affect `W` and `H`). Not all voronoi cells are replaced by their average color, only around `90` percent of them. The remaining `10` percent's pixels remain unchanged.

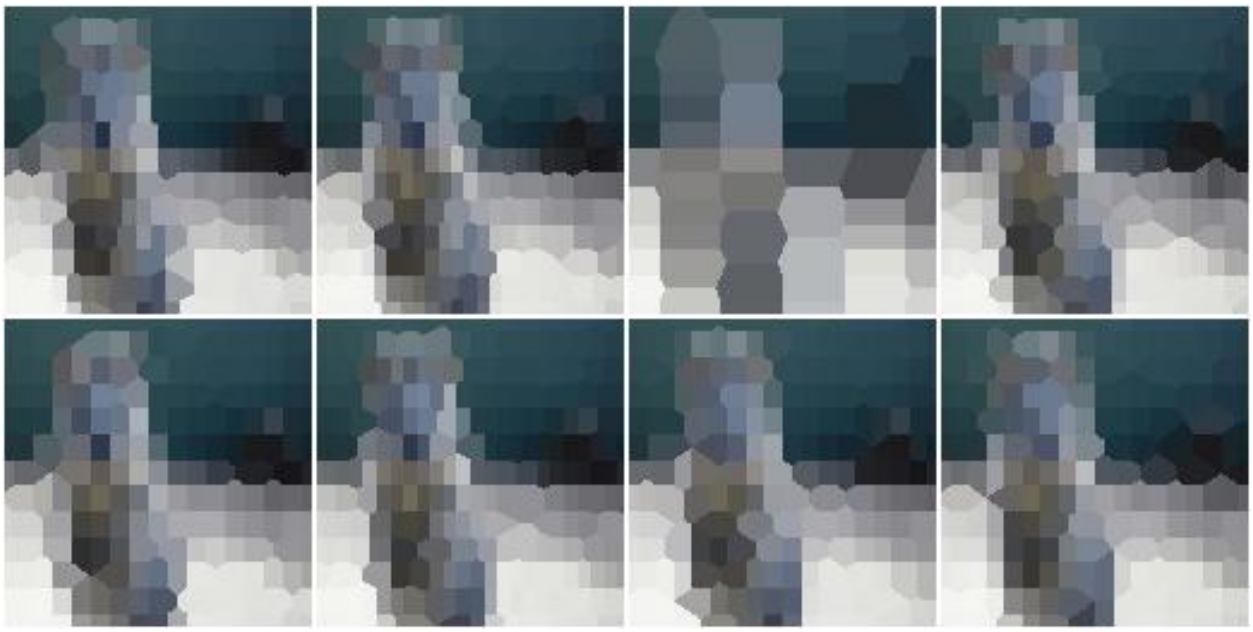
```
points_sampler = iaa.DropoutPointsSampler(
    iaa.RelativeRegularGridPointsSampler(
```

(continues on next page)



(continued from previous page)

```
        n_cols_frac=(0.05, 0.2),
        n_rows_frac=0.1),
    0.2)
aug = iaa.Voronoi(points_sampler, p_replace=0.9, max_size=None)
```



9.12.3 UniformVoronoi

Uniformly sample Voronoi cells on images and average colors within them.

This augmenter is a shortcut for the combination of `Voronoi` with `UniformPointsSampler`. Hence, it generates

a fixed amount of N random coordinates of voronoi cells on each image. The cell coordinates are sampled uniformly using the image height and width as maxima.

API link: *UniformVoronoi*

Example. Sample for each image uniformly the number of voronoi cells N from the interval [100, 500]. Then generates N coordinates by sampling uniformly the x-coordinates from [0, W] and the y-coordinates from [0, H], where H is the image height and W the image width. Then uses these coordinates to group the image pixels into voronoi cells and averages the colors within them. The process is performed at an image size not exceeding 128px on any side. If necessary, the downscaling is performed using linear interpolation.

```
import imgaug.augmenters as iaa
aug = iaa.UniformVoronoi((100, 500))
```

Example. Same as above, but always samples N=250 cells, replaces only 90 percent of them with their average color (the pixels of the remaining 10 percent are not changed) and performs the transformation at the original image size.

```
aug = iaa.UniformVoronoi(250, p_replace=0.9, max_size=None)
```

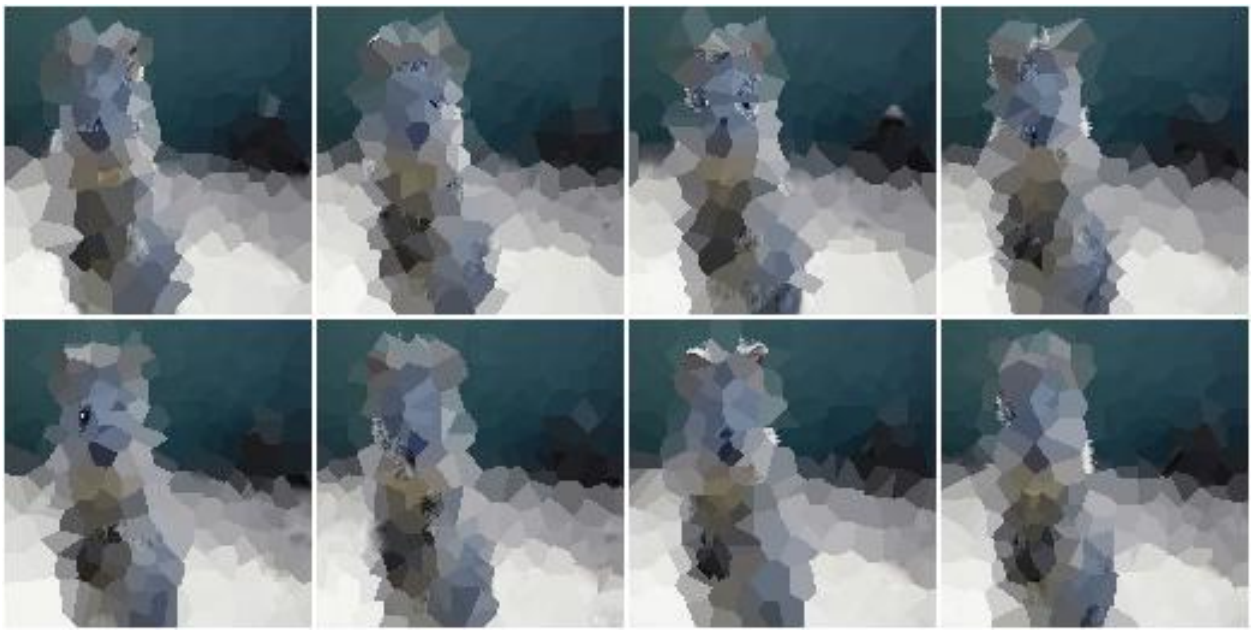
9.12.4 RegularGridVoronoi

Sample Voronoi cells from regular grids and color-average them.

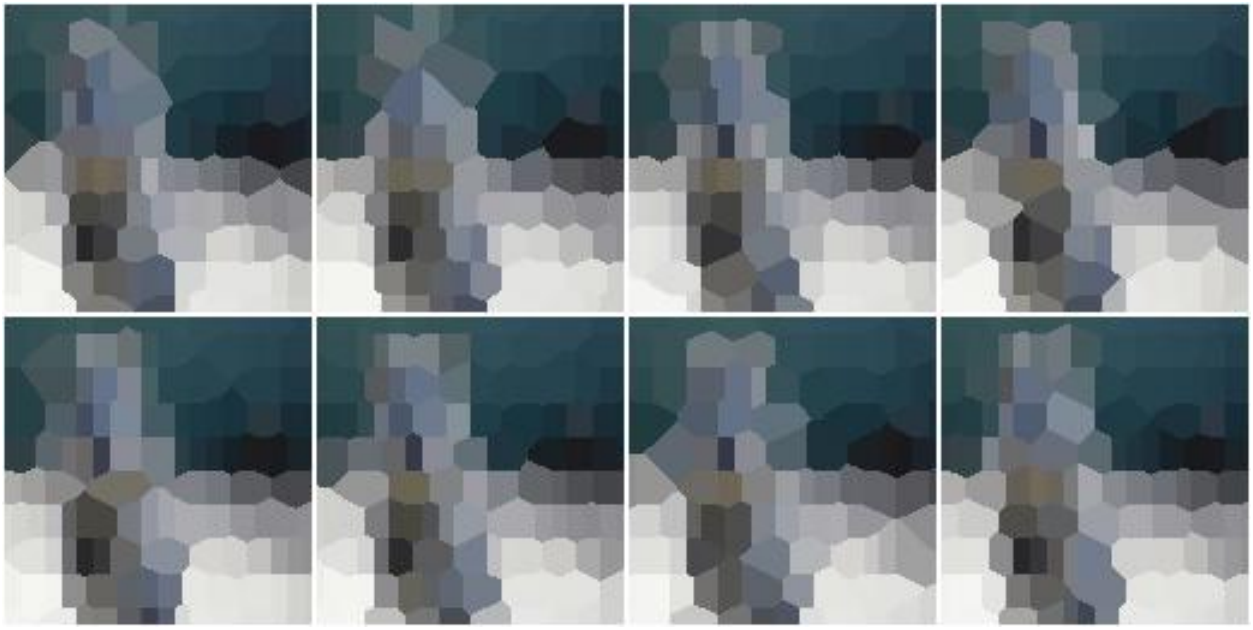
This augmenter is a shortcut for the combination of Voronoi, RegularGridPointsSampler and DropoutPointsSampler. Hence, it generates a regular grid with R rows and C columns of coordinates on each image. Then, it drops p percent of the R*C coordinates to randomize the grid. Each image pixel then belongs to the voronoi cell with the closest coordinate.

API link: *RegularGridVoronoi*

Example. Place a regular grid of 10x20 (height x width) coordinates on each image. Randomly drop on average 20 percent of these points to create a less regular pattern. Then use the remaining coordinates to group the image pixels into voronoi cells and average the colors within them. The process is performed at an image size not exceeding 128px on any side. If necessary, the downscaling is performed using linear interpolation.

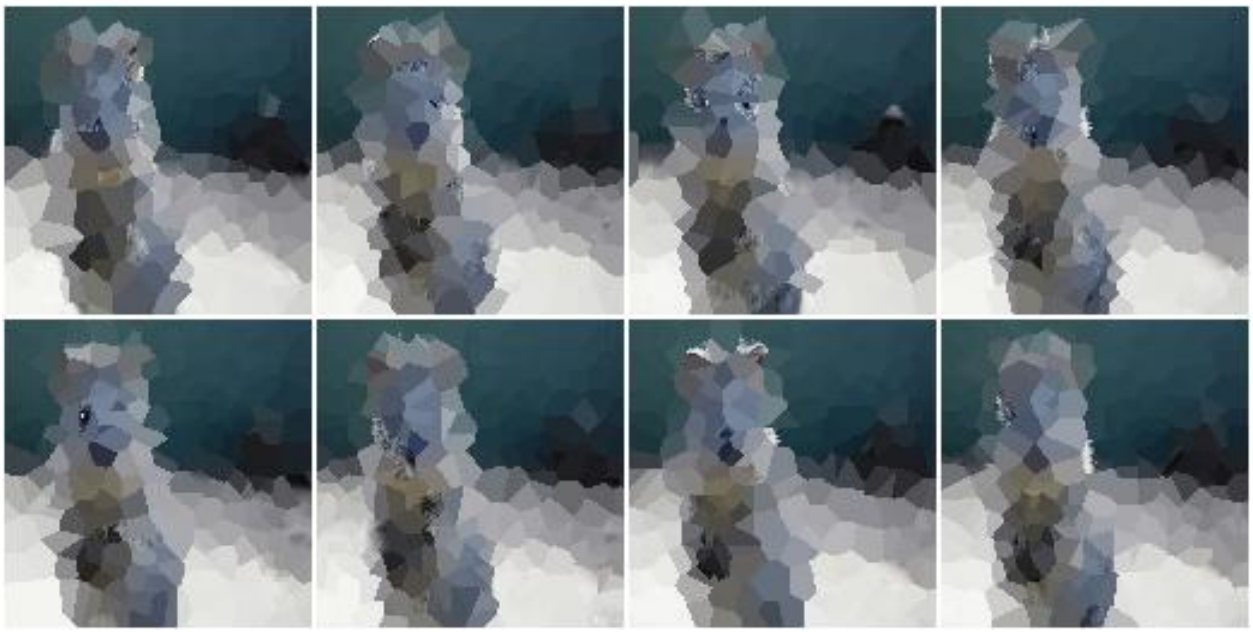


```
import imgaug.augmenters as iaa
aug = iaa.RegularGridVoronoi(10, 20)
```



Example. Same as above, generates a grid with randomly `10` to `30` rows, drops none of the generated points, replaces only `90` percent of the voronoi cells with their average color (the pixels of the remaining `10` percent are not changed) and performs the transformation at the original image size.

```
aug = iaa.RegularGridVoronoi(
    (10, 30), 20, p_drop_points=0.0, p_replace=0.9, max_size=None)
```



9.12.5 RelativeRegularGridVoronoi

Sample Voronoi cells from image-dependent grids and color-average them.

This augmenter is a shortcut for the combination of `Voronoi`, `RegularGridPointsSampler` and `DropoutPointsSampler`. Hence, it generates a regular grid with `R` rows and `C` columns of coordinates on each image. Then, it drops `p` percent of the `R*C` coordinates to randomize the grid. Each image pixel then belongs to the voronoi cell with the closest coordinate.

Note: In contrast to the other Voronoi augmenters, this one uses `None` as the default value for *max_size*, i.e. the color averaging is always performed at full resolution. This enables the augmenter to make most use of the added points for larger images. It does however slow down the augmentation process.

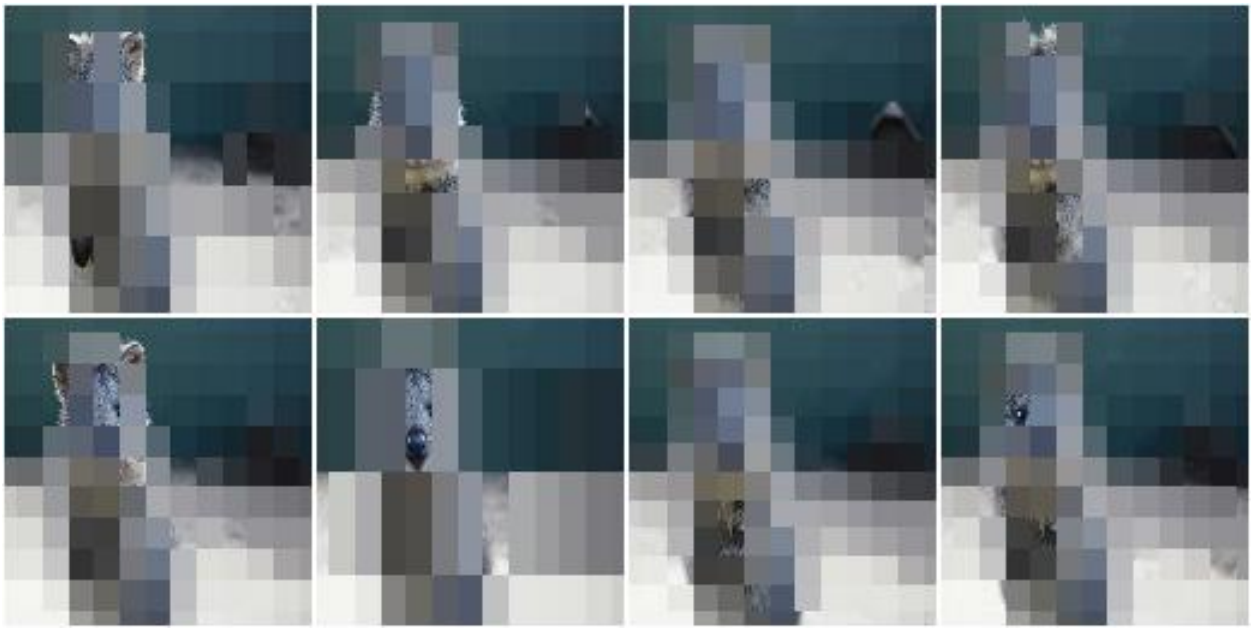
API link: *RelativeRegularGridVoronoi*

Example. Place a regular grid of `R x C` coordinates on each image, where `R` is the number of rows and computed as `R=0.1*H` with `H` being the height of the input image. `C` is the number of columns and analogously estimated from the image width `W` as `C=0.25*W`. Larger images will lead to larger `R` and `C` values. On average, `20` percent of these grid coordinates are randomly dropped to create a less regular pattern. Then, the remaining coordinates are used to group the image pixels into voronoi cells and the colors within them are averaged.

```
import imgaug.augmenters as iaa
aug = iaa.RelativeRegularGridVoronoi(0.1, 0.25)
```

Example. Same as above, generates a grid with randomly `R=r*H` rows, where `r` is sampled uniformly from the interval `[0.03, 0.1]` and `C=0.1*W` rows. No points are dropped. The augmenter replaces only `90` percent of the voronoi cells with their average color (the pixels of the remaining `10` percent are not changed). Images larger than `512px` are temporarily downscaled (*before* sampling the grid points) so that no side exceeds `512px`. This improves performance, but degrades the quality of the resulting image.

```
aug = iaa.RelativeRegularGridVoronoi(
    (0.03, 0.1), 0.1, p_drop_points=0.0, p_replace=0.9, max_size=512)
```



9.13 augmenters.size

9.13.1 Resize

Augmenter that resizes images to specified heights and widths.

API link: *Resize*

Example. Resize each image to height=32 and width=64:

```
import imgaug.augmenters as iaa
aug = iaa.Resize({"height": 32, "width": 64})
```

Example. Resize each image to height=32 and keep the aspect ratio for width the same:

```
aug = iaa.Resize({"height": 32, "width": "keep-aspect-ratio"})
```

Example. Resize each image to something between 50 and 100% of its original size:

```
aug = iaa.Resize((0.5, 1.0))
```

Example. Resize each image's height to 50-75% of its original size and width to either 16px or 32px or 64px:

```
aug = iaa.Resize({"height": (0.5, 0.75), "width": [16, 32, 64]})
```

9.13.2 CropAndPad

Crop/pad images by pixel amounts or fractions of image sizes.

Cropping removes pixels at the sides (i.e. extracts a subimage from a given full image). Padding adds pixels to the sides (e.g. black pixels).

Note: This augmenter automatically resizes images back to their original size after it has augmented them. To deactivate this, add the parameter `keep_size=False`.

API link: *CropAndPad*

Example. Crop or pad each side by up to 10 percent relative to its original size (negative values result in cropping, positive in padding):

```
import imgaug.augmenters as iaa
aug = iaa.CropAndPad(percent=(-0.25, 0.25))
```

Example. Pad each side by 0 to 20 percent. This adds new pixels to the sides. These pixels will either be filled with a constant value (mode=constant) or filled with the value on the closest edge (mode=edge). If a constant value is used, it will be a random value between 0 and 128 (sampled per image).

```
aug = iaa.CropAndPad(
    percent=(0, 0.2),
    pad_mode=["constant", "edge"],
    pad_cval=(0, 128)
)
```

Example. Pad the top side of each image by 0 to 30 pixels, the right side by 0-10px, bottom side by 0-30px and left side by 0-10px. Use any of the available modes to fill new pixels and if the mode is *constant* then use a constant value between 0 and 128.

```
aug = iaa.CropAndPad(
    px=((0, 30), (0, 10), (0, 30), (0, 10)),
    pad_mode=ia.ALL,
    pad_cval=(0, 128)
)
```

Example. Crop/pad each side by up to 10px. The value will be sampled once per image and used for all sides (i.e. all sides gain/lose the same number of rows/colums).

```
aug = iaa.CropAndPad(
    px=(-10, 10),
    sample_independently=False
)
```

9.13.3 Pad

Pad images, i.e. adds columns/rows of pixels to them.

This is a shortcut for `CropAndPad`. It only accepts positive pixel/percent values.

API link: *Pad*

9.13.4 Crop

Crop images, i.e. remove columns/rows of pixels at the sides of images.

This is a shortcut for `CropAndPad`. It only accepts positive pixel/percent values and transfers them as negative values to `CropAndPad`.

API link: *Crop*

9.13.5 PadToFixedSize

Pad images to minimum width/height.

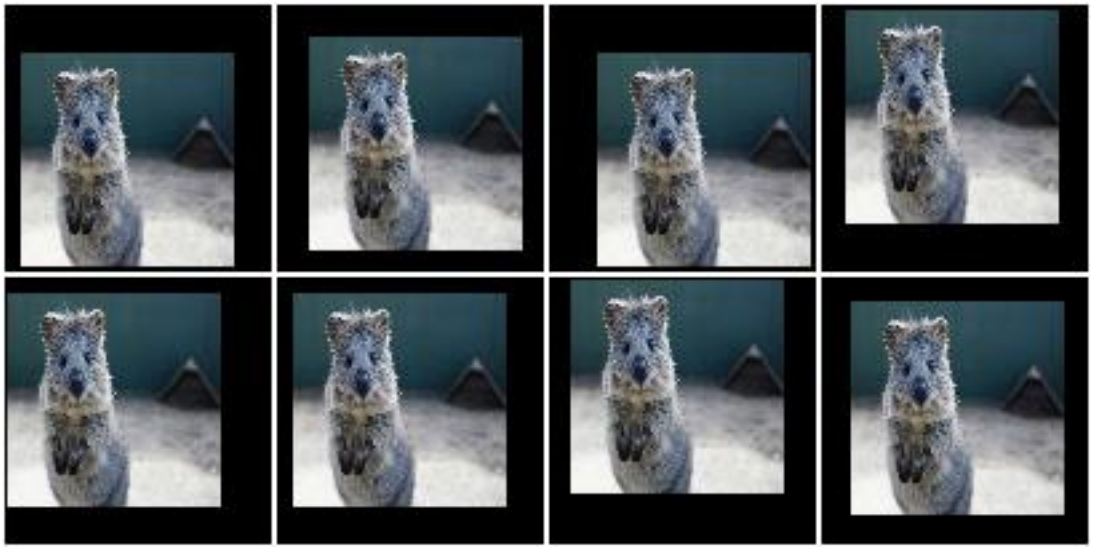
If images are already at the minimum width/height or are larger, they will not be padded. Note that this also means that images will not be cropped if they exceed the required width/height.

The augmenter randomly decides per image how to distribute the required padding amounts over the image axis. E.g. if 2px have to be padded on the left or right to reach the required width, the augmenter will sometimes add 2px to the left and 0px to the right, sometimes add 2px to the right and 0px to the left and sometimes add 1px to both sides. Set *position* to `center` to prevent that.

API link: *PadToFixedSize*

Example. For image sides smaller than `100` pixels, pad to `100` pixels. Do nothing for the other edges. The padding is randomly (uniformly) distributed over the sides, so that e.g. sometimes most of the required padding is applied to the left, sometimes to the right (analogous top/bottom). The input image here has a size of `80x80`.

```
import imgaug.augmenters as iaa
aug = iaa.PadToFixedSize(width=100, height=100)
```



Example. For image sides smaller than `100` pixels, pad to `100` pixels. Do nothing for the other image sides. The padding is always equally distributed over the left/right and top/bottom sides. The input image here has a size of `80x80`.

```
aug = iaa.PadToFixedSize(width=100, height=100, position="center")
```

Example. For image sides smaller than `100` pixels, pad to `100` pixels and use any possible padding mode for that. Do nothing for the other image sides. The padding is always equally distributed over the left/right and top/bottom sides. The input image here has a size of `80x80`.

```
aug = iaa.PadToFixedSize(width=100, height=100, pad_mode=ia.ALL)
```

Example. Pad images smaller than `100x100` until they reach `100x100`. Analogously, crop images larger than `100x100` until they reach `100x100`. The output images therefore have a fixed size of `100x100`. The input image here has a size of `80x120`, so that the top/bottom sides have to be cropped and the left/right sides have to be padded. Note that the original image was resized to `80x120`, leading to a bit of an distorted appearance.

```
aug = iaa.Sequential([
    iaa.PadToFixedSize(width=100, height=100),
    iaa.CropToFixedSize(width=100, height=100)
])
```

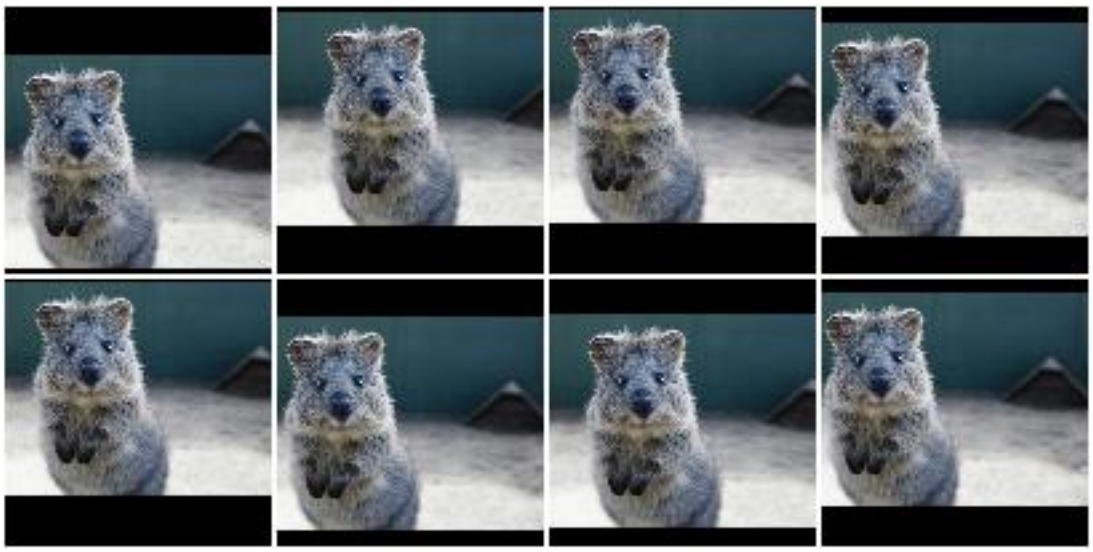
9.13.6 CropToFixedSize

Crop images down to a fixed maximum width/height.

If images are already at the maximum width/height or are smaller, they will not be cropped. Note that this also means that images will not be padded if they are below the required width/height.

The augmenter randomly decides per image how to distribute the required cropping amounts over the image axis. E.g. if 2px have to be cropped on the left or right to reach the required width, the augmenter will sometimes remove 2px from the left and 0px from the right, sometimes remove 2px from the right and 0px from the left and sometimes remove 1px from both sides. Set *position* to `center` to prevent that.

API link: *CropToFixedSize*



Example. For image sides larger than `100` pixels, crop to `100` pixels. Do nothing for the other sides. The cropping amounts are randomly (and uniformly) distributed over the sides of the image. The input image here has a size of `120x120`.

```
import imgaug.augmenters as iaa
aug = iaa.CropToFixedSize(width=100, height=100)
```

Example. For sides larger than `100` pixels, crop to `100` pixels. Do nothing for the other sides. The cropping amounts are always equally distributed over the left/right sides of the image (and analogously for top/bottom). The input image here has a size of `120x120`.

```
aug = iaa.CropToFixedSize(width=100, height=100, position="center")
```

Example. Pad images smaller than `100x100` until they reach `100x100`. Analogously, crop images larger than `100x100` until they reach `100x100`. The output images therefore have a fixed size of `100x100`. The input image here has a size of `80x120`, so that the top/bottom sides have to be cropped and the left/right sides have to be padded. Note that the original image was resized to `80x120`, leading to a bit of an distorted appearance.

```
aug = iaa.Sequential([
    iaa.PadToFixedSize(width=100, height=100),
    iaa.CropToFixedSize(width=100, height=100)
])
```

9.13.7 KeepSizeByResize

Resize images back to their input sizes after applying child augmenters.

Combining this with e.g. a cropping augmenter as the child will lead to images being resized back to the input size after the crop operation was applied. Some augmenters have a `keep_size` argument that achieves the same goal (if set to `True`), though this augmenter offers control over the interpolation mode and which augmentables to resize (images, heatmaps, segmentation maps).

API link: *KeepSizeByResize*

Example. Apply random cropping to input images, then resize them back to their original input sizes. The resizing is done using this augmenter instead of the corresponding internal resizing operation in `Crop`.

```
import imgaug.augmenters as iaa
aug = iaa.KeepSizeByResize(
    iaa.Crop((20, 40), keep_size=False)
)
```

Example. Same as in the previous example, but images are now always resized using nearest neighbour interpolation.

```
aug = iaa.KeepSizeByResize(
    iaa.Crop((20, 40), keep_size=False),
    interpolation="nearest"
)
```

Example. Similar to the previous example, but images are now sometimes resized using linear interpolation and sometimes using nearest neighbour interpolation. Heatmaps are resized using the same interpolation as was used for the corresponding image. Segmentation maps are not resized and will therefore remain at their size after cropping.

```
aug = iaa.KeepSizeByResize(
    iaa.Crop((20, 40), keep_size=False),
```

(continues on next page)

(continued from previous page)

```
    interpolation=["nearest", "cubic"],
    interpolation_heatmaps=iaa.KeepSizeByResize.SAME_AS_IMAGES,
    interpolation_segmaps=iaa.KeepSizeByResize.NO_RESIZE
)
```

9.14 augmenters.weather

Note: All examples below use the following input image:

9.14.1 FastSnowyLandscape

Convert non-snowy landscapes to snowy ones.

This augmenter expects to get an image that roughly shows a landscape.

API link: *FastSnowyLandscape*

Example. Search for all pixels in the image with a lightness value in HLS colorspace of less than `140` and increase their lightness by a factor of `2.5`.

```
import imgaug.augmenters as iaa
aug = iaa.FastSnowyLandscape(
    lightness_threshold=140,
    lightness_multiplier=2.5
)
```

Example. Search for all pixels in the image with a lightness value in HLS colorspace of less than `128` or less than `200` (one of these values is picked per image) and multiply their lightness by a factor of `x` with `x` being sampled from `uniform(1.5, 3.5)` (once per image).

```
aug = iaa.FastSnowyLandscape(
    lightness_threshold=[128, 200],
    lightness_multiplier=(1.5, 3.5)
)
```

Example. Similar to the previous example, but the lightness threshold is sampled from `uniform(100, 255)` (per image) and the multiplier from `uniform(1.0, 4.0)` (per image). This seems to produce good and varied results.

```
aug = iaa.FastSnowyLandscape(
    lightness_threshold=(100, 255),
    lightness_multiplier=(1.0, 4.0)
)
```

9.14.2 Clouds

Add clouds to images.

This is a wrapper around `CloudLayer`. It executes 1 to 2 layers per image, leading to varying densities and frequency patterns of clouds.

This augmenter seems to be fairly robust w.r.t. the image size. Tested with `96x128`, `192x256` and `960x1280`.

API link: *Clouds()*

Example. Create an augmenter that adds clouds to images:

```
import imgaug.augmenters as iaa
aug = iaa.Clouds()
```

9.14.3 Fog

Add fog to images.

This is a wrapper around `CloudLayer`. It executes a single layer per image with a configuration leading to fairly dense clouds with low-frequency patterns.

This augmenter seems to be fairly robust w.r.t. the image size. Tested with `96x128`, `192x256` and `960x1280`.

API link: *Fog()*

Example. Create an augmenter that adds fog to images:

```
import imgaug.augmenters as iaa
aug = iaa.Fog()
```

9.14.4 CloudLayer

Add a single layer of clouds to an image.

API link: *CloudLayer*

TODO add examples

9.14.5 Snowflakes

Add falling snowflakes to images.

This is a wrapper around `SnowflakesLayer`. It executes 1 to 3 layers per image.

API link: *Snowflakes()*

Example. Add snowflakes to small images (around `96x128`):

```
import imgaug.augmenters as iaa
aug = iaa.Snowflakes(flake_size=(0.1, 0.4), speed=(0.01, 0.05))
```

Example. Add snowflakes to medium-sized images (around `192x256`):

```
aug = iaa.Snowflakes(flake_size=(0.2, 0.7), speed=(0.007, 0.03))
```

Example. Add snowflakes to large images (around `960x1280`):

```
aug = iaa.Snowflakes(flake_size=(0.7, 0.95), speed=(0.001, 0.03))
```

9.14.6 SnowflakesLayer

Add a single layer of falling snowflakes to images.

API link: *SnowflakesLayer*

TODO add examples

CHAPTER 10

Performance

Below are performance measurements of each augmenter for image augmentation (`augment_images()`), heatmap augmentation (`augment_heatmaps()`) and keypoint/landmark augmentation (`augment_keypoints()`). (Last updated for `0.3.0`)

System: The numbers were computed based on a haswell-generation i7 3.2Ghz CPU with DDR3 memory. That is a rather dated system by today's standards. A modern, high-end system should achieve higher bandwidths.

All experiments were conducted using python 3.7 and numpy 1.17.0. Note that the precise python/numpy version can have significant impact on your performance.

Experiments Settings: All augmenters were run with reasonable parameter choices that should reflect expected real-world usage, while avoiding too simple parameter values that would lead to inflated scores. Some parameter choices are listed below, the remaining ones can be looked up in `measure_performance.py`. Kernel sizes were all set to `3x3`, unless otherwise mentioned. The inputs focused on a small and large image-size setting, using `64x64x3` and `224x224x3` as the respective sizes. The base image was taken from skimage.data.astronaut, which should be a representative real-world image. Batch sizes of `1` and `128` were tested. Each augmenter was run at least `40` times on the generated input and the average of the measured runtimes was computed to derive bandwidth in mbit per second and th raw number of augmented items (e.g. images) per second.

10.1 Results Overview

From the results, the following points can be derived.

Inputs:

- Use large batch sizes whenever possible. Many augmenters are significantly faster with these.
- Large image sizes lead to higher throughput based on mbit/sec. Smaller images lead to lower throughput, but significantly more items/sec (roughly 4-10x more). Use small images whenever possible.
- For keypoint-based and heatmap-based augmentation, try to increase the number of items per augmented instance. E.g. `augment_keypoints()` accepts a list of `KeypointsOnImage` instances, with each such instance representing the keypoints on an image. Try to place for each image all keypoints in the respective

`KeypointsOnImage` instance instead of splitting them into multiple such instances (which would be more work anyways). The same is true for bounding boxes, heatmaps and segmentation maps.

- Keypoint- and heatmap-based inputs are only affected by augmenters that change the geometry of the image (e.g. `Crop` or `Affine`). Other augmenters are essentially free to execute as they do not perform any changes.
- Keypoint-based augmentation is very fast for almost all augmenters, reaching several 100k keypoints per second. Slower augmenters are `ElasticTransformation` and `PiecewiseAffine`, as these currently have to fall back to image-based algorithms.

Parameter choices:

- When possible, nearest neighbour interpolation or linear interpolation should be used as these are significantly faster than other options. Most augmenters that use interpolation offer either an `order` parameter (0=nearest neighbour, 1=linear) or an `interpolation` parameter ("nearest", "linear").
- Using `keep_size=True` is the default setting in all augmenters that change image sizes. It is convenient, as it ensures that image sizes are not altered by the augmentation. It does however incur a significant performance penalty, often more than halving the bandwidth. Try `keep_size=False` when possible. You can still resize images manually after augmentation or by using `KeepSizeByResize(Sequential(<augmenters>))`.
- When augmenters offer modes to fill newly created pixels in user-defined ways (e.g. `pad_mode=constant` in `Pad` to fill up all padded pixels with a specified constant color), using `edge` instead of `constant` will usually not incur a significant performance penalty.

Specific Augmenter suggestions:

- For augmenters where an elementwise sibling exists (e.g. `Multiply` and `MultiplyElementwise`), the elementwise augmenter is usually significantly slower than the non-elementwise one.
- If blurring is required, `AverageBlur` is the fastest choice, followed by `GaussianBlur`.
- Augmenters that operate on coarser images (e.g. `CoarseDropout` vs `Dropout`) can be significantly faster than their non-coarse siblings.
- Contrast normalizing augmenters are all comparable in performance, except for histogram-based ones, which are significantly slower.
- `PiecewiseAffine` is a very slow augmenter and should usually be replaced by ElasticTransformation, which achieves similar outputs and is quite a bit faster.
- `Superpixels` is a fairly slow augmenter and should usually be wrapped in e.g. `Sometimes` to not apply it very often and reduce its performance impact.
- Weather augmenters other than `FastSnowyLandscape` are rather slow and should only be used when sensible.

10.2 Images

Numbers below are for small images (`64x64x3`) and large images (`224x224x3`). `B=1` denotes a batch size of `1`, `B=128` one of `128`.

In mbit/sec:

	64x64x3, uint8		224x224x3
Augmenter	B=1	B=128	B=1
Sequential (2xNoop)	10291.8	51537.8	37374.1

Table 1 – continued from previous page

Sequential (2xNoop, random_order)	1160.2	37697.3	11243.0
SomeOf (1-3, 3xNoop)	286.2	1588.5	2671.4
SomeOf (1-3, 3xNoop, random_order)	239.2	1633.3	2277.9
OneOf (3xNoop)	786.0	1994.0	6091.6
Sometimes (Noop)	671.1	15784.4	6165.2
WithChannels ([1,2], Noop)	2103.2	10006.9	7839.0
Noop	16708.9	53947.0	48487.4
Lambda (return input)	16040.9	54141.6	45396.7
AssertLambda (return True)	15666.0	53733.9	46598.6
AssertShape (None, H, W, None)	6315.4	18876.3	29643.0
ChannelShuffle (0.5)	528.8	1605.4	3653.2
Add	276.1	573.3	2560.3
AddElementwise	265.8	319.3	990.9
AdditiveGaussianNoise	206.1	237.7	749.9
AdditiveLaplaceNoise	171.6	193.0	441.6
AdditivePoissonNoise	145.4	156.7	310.4
Multiply	360.8	1014.7	3074.5
MultiplyElementwise	239.4	288.1	983.5
Dropout (1-5%)	303.3	379.5	1121.2
CoarseDropout (1-5%, size=1-10%)	165.8	185.7	1171.7
ReplaceElementwise	152.8	169.9	738.8
ImpulseNoise	130.1	141.7	494.1
SaltAndPepper	136.7	150.2	695.5
CoarseSaltAndPepper	99.6	105.7	693.6
Salt	113.5	121.0	622.4
CoarseSalt	86.7	90.7	640.0
Pepper	112.6	120.9	620.9
CoarsePepper	86.5	91.0	642.5
Invert (10%)	664.6	12262.6	6715.9
JpegCompression (50-99%)	80.1	91.9	300.0
Alpha (Noop)	264.2	506.4	918.8
AlphaElementwise (Noop)	188.9	215.8	471.1
SimplexNoiseAlpha (Noop)	29.8	28.3	187.9
FrequencyNoiseAlpha (Noop)	37.1	36.0	216.7
GaussianBlur (sigma=(1,5))	283.4	629.4	2367.5
AverageBlur	435.4	3457.5	3101.8
MedianBlur	173.2	265.2	306.1
BilateralBlur	158.2	366.5	447.9
MotionBlur	74.7	75.2	703.1
WithColorspace (HSV, Noop)	695.5	1067.5	1444.4
WithHueAndSaturation	229.7	335.8	646.7
MultiplyHueAndSaturation	99.3	150.1	410.9
MultiplyHue	93.4	151.9	431.8
MultiplySaturation	93.8	152.5	427.2
AddToHueAndSaturation	228.3	768.0	944.5
AddToHue	267.3	769.4	1002.7
AddToSaturation	269.1	767.3	985.8
ChangeColorspace (HSV)	440.1	800.9	1641.4
Grayscale	208.8	332.3	680.6
KMeansColorQuantization (2-16 colors)	23.9	40.9	209.3

Table 1 – continued from previous page

UniformColorQuantization (2-16 colors)	187.3	327.1	638.7
GammaContrast	259.6	325.1	2319.9
SigmoidContrast	206.5	245.9	1974.1
LogContrast	257.0	325.8	2334.9
LinearContrast	324.3	430.9	2743.4
AllChannelsHistogramEqualization	1110.6	1912.8	2191.6
HistogramEqualization	470.5	878.1	1187.3
AllChannelsCLAHE	143.2	344.8	995.4
CLAHE	136.9	432.1	766.9
Convolve (3x3)	1303.6	2820.4	4369.1
Sharpen	261.0	295.0	1708.7
Emboss	261.9	296.1	1769.5
EdgeDetect	383.4	459.2	2260.1
DirectedEdgeDetect	99.6	102.0	890.8
Canny	63.0	109.7	295.5
Fliplr (p=100%)	1165.1	4625.7	5828.7
Flipud (p=100%)	1468.1	13368.8	8842.6
Affine (order=0, constant)	96.3	272.4	943.3
Affine (order=1, constant)	93.5	247.3	840.3
Affine (order=3, constant)	84.6	206.1	498.6
Affine (order=1, edge)	93.3	246.3	830.6
Affine (order=1, constant, skimage)	49.9	74.5	178.7
PiecewiseAffine (4x4, order=1, constant)	5.0	4.9	27.4
PiecewiseAffine (4x4, order=0, constant)	5.2	5.1	33.2
PiecewiseAffine (4x4, order=1, edge)	5.0	4.9	27.5
PiecewiseAffine (8x8, order=1, constant)	1.1	1.1	9.2
PerspectiveTransform	155.9	221.3	1129.9
PerspectiveTransform (keep_size)	134.2	178.5	831.7
ElasticTransformation (order=0, constant)	108.2	183.0	572.1
ElasticTransformation (order=1, constant)	102.8	168.6	531.7
ElasticTransformation (order=1, nearest)	104.1	169.6	532.7
ElasticTransformation (order=1, reflect)	102.5	168.6	526.5
Rot90	463.3	4300.0	4820.7
Rot90 (keep_size)	400.2	2267.4	2398.0
AveragePooling	134.5	187.4	481.4
AveragePooling (keep_size)	119.7	157.3	422.2
MaxPooling	143.7	202.8	518.6
MaxPooling (keep_size)	127.3	168.3	476.1
MinPooling	144.3	197.1	518.0
MinPooling (keep_size)	128.5	165.6	496.8
MedianPooling	133.5	178.5	607.7
MedianPooling (keep_size)	118.7	151.8	573.2
Superpixels (max_size=64, cubic)	10.4	10.8	124.7
Superpixels (max_size=64, linear)	11.0	10.8	131.4
Superpixels (max_size=128, linear)	10.8	10.9	52.7
Superpixels (max_size=224, linear)	10.5	11.1	20.0
UniformVoronoi (250-1000k points, linear)	3.4	3.4	10.4
RegularGridVoronoi (16-31 rows/cols)	3.4	3.4	10.7
RelativeRegularGridVoronoi (7%-14% rows/cols)	3.5	3.5	3.5
Resize (nearest)	302.7	780.1	2436.5

Table 1 – continued from previous page

Resize (linear)	287.4	679.6	1802.3
Resize (cubic)	267.9	586.8	1357.7
CropAndPad	208.9	228.8	2099.8
CropAndPad (edge)	209.0	231.2	2105.4
CropAndPad (keep_size)	169.2	178.5	1278.1
Crop	331.1	381.7	3198.3
Crop (keep_size)	242.0	261.9	1700.4
Pad	208.1	226.8	2009.4
Pad (edge)	207.3	227.4	1977.3
Pad (keep_size)	167.7	176.3	1197.4
PadToFixedSize	202.1	1330.1	2134.7
CropToFixedSize	388.5	3856.7	3747.3
KeepSizeByResize (CropToFixedSize(nearest))	203.7	1032.9	1746.1
KeepSizeByResize (CropToFixedSize(linear))	196.7	884.4	1406.0
KeepSizeByResize (CropToFixedSize(cubic))	186.8	745.9	1120.7
FastSnowyLandscape	157.5	270.3	514.1
Clouds	19.9	20.1	60.8
Fog	33.9	33.9	99.8
CloudLayer	33.1	33.2	99.0
Snowflakes	16.4	16.7	87.9
SnowflakesLayer	33.1	33.6	192.0

In images/sec:

	64x64x3, uint8		224x224x
Augmenter	B=1	B=128	B=1
Sequential (2xNoop)	109779.4	549736.6	32543.4
Sequential (2xNoop, random_order)	12375.0	402104.3	9789.9
SomeOf (1-3, 3xNoop)	3053.2	16944.4	2326.1
SomeOf (1-3, 3xNoop, random_order)	2551.2	17421.4	1983.5
OneOf (3xNoop)	8384.3	21269.4	5304.3
Sometimes (Noop)	7158.0	168366.5	5368.3
WithChannels ([1,2], Noop)	22434.6	106739.9	6825.8
Noop	178228.2	575434.6	42220.3
Lambda (return input)	171103.0	577510.3	39529.1
AssertLambda (return True)	167103.7	573161.6	40575.6
AssertShape (None, H, W, None)	67363.9	201347.2	25811.6
ChannelShuffle (0.5)	5640.1	17123.9	3181.1
Add	2945.4	6114.7	2229.4
AddElementwise	2835.7	3406.0	862.9
AdditiveGaussianNoise	2197.9	2535.8	653.0
AdditiveLaplaceNoise	1830.4	2058.9	384.6
AdditivePoissonNoise	1551.4	1671.6	270.3
Multiply	3848.9	10823.4	2677.1
MultiplyElementwise	2553.6	3072.9	856.4
Dropout (1-5%)	3235.3	4047.5	976.2
CoarseDropout (1-5%, size=1-10%)	1768.0	1980.8	1020.3
ReplaceElementwise	1630.1	1812.7	643.3
ImpulseNoise	1387.5	1511.0	430.2
SaltAndPepper	1458.0	1602.4	605.6

Table 2 – continued from previous page

CoarseSaltAndPepper	1062.3	1128.0	604.0
Salt	1210.5	1290.3	542.0
CoarseSalt	925.1	967.4	557.3
Pepper	1201.0	1289.8	540.6
CoarsePepper	922.2	970.6	559.4
Invert (10%)	7089.3	130801.3	5847.8
JpegCompression (50-99%)	854.3	980.4	261.2
Alpha (Noop)	2818.0	5401.1	800.1
AlphaElementwise (Noop)	2015.3	2301.7	410.2
SimplexNoiseAlpha (Noop)	317.8	301.8	163.6
FrequencyNoiseAlpha (Noop)	395.5	384.3	188.7
GaussianBlur (sigma=(1,5))	3023.1	6713.1	2061.5
AverageBlur	4643.9	36880.1	2700.9
MedianBlur	1847.1	2829.0	266.5
BilateralBlur	1687.5	3909.4	390.0
MotionBlur	797.3	801.9	612.2
WithColorspace (HSV, Noop)	7418.5	11386.4	1257.7
WithHueAndSaturation	2450.6	3581.9	563.1
MultiplyHueAndSaturation	1058.9	1601.1	357.8
MultiplyHue	996.2	1620.1	376.0
MultiplySaturation	1000.5	1626.7	372.0
AddToHueAndSaturation	2435.2	8192.2	822.4
AddToHue	2851.1	8207.3	873.1
AddToSaturation	2870.5	8184.4	858.4
ChangeColorspace (HSV)	4694.4	8542.6	1429.2
Grayscale	2227.6	3544.3	592.6
KMeansColorQuantization (2-16 colors)	255.2	436.1	182.2
UniformColorQuantization (2-16 colors)	1997.6	3489.2	556.1
GammaContrast	2769.1	3467.9	2020.0
SigmoidContrast	2202.9	2623.4	1719.0
LogContrast	2740.9	3474.9	2033.1
LinearContrast	3459.0	4596.5	2388.8
AllChannelsHistogramEqualization	11846.3	20403.2	1908.3
HistogramEqualization	5019.1	9366.5	1033.8
AllChannelsCLAHE	1527.9	3678.2	866.7
CLAHE	1459.9	4609.3	667.8
Convolve (3x3)	13905.2	30084.2	3804.4
Sharpen	2784.0	3146.7	1487.8
Emboss	2793.5	3158.7	1540.8
EdgeDetect	4089.5	4897.9	1968.0
DirectedEdgeDetect	1062.8	1088.3	775.7
Canny	671.8	1169.9	257.3
Fliplr (p=100%)	12427.9	49341.2	5075.3
Flipud (p=100%)	15659.5	142600.4	7699.7
Affine (order=0, constant)	1026.9	2906.0	821.4
Affine (order=1, constant)	997.7	2638.0	731.7
Affine (order=3, constant)	902.0	2198.5	434.2
Affine (order=1, edge)	995.6	2626.9	723.2
Affine (order=1, constant, skimage)	532.0	794.7	155.6
PiecewiseAffine (4x4, order=1, constant)	53.5	52.1	23.9

Table 2 – continued from previous page

PiecewiseAffine (4x4, order=0, constant)	55.3	54.2	28.9
PiecewiseAffine (4x4, order=1, edge)	53.4	52.4	23.9
PiecewiseAffine (8x8, order=1, constant)	12.1	11.8	8.0
PerspectiveTransform	1663.0	2360.1	983.9
PerspectiveTransform (keep_size)	1431.2	1904.2	724.2
ElasticTransformation (order=0, constant)	1154.1	1952.2	498.2
ElasticTransformation (order=1, constant)	1096.4	1798.2	463.0
ElasticTransformation (order=1, nearest)	1110.1	1809.5	463.8
ElasticTransformation (order=1, reflect)	1093.3	1798.3	458.4
Rot90	4942.1	45866.6	4197.6
Rot90 (keep_size)	4268.9	24186.1	2088.0
AveragePooling	1434.7	1999.3	419.2
AveragePooling (keep_size)	1276.9	1678.1	367.6
MaxPooling	1533.3	2162.8	451.6
MaxPooling (keep_size)	1358.2	1795.6	414.6
MinPooling	1539.0	2102.2	451.1
MinPooling (keep_size)	1370.6	1766.1	432.6
MedianPooling	1424.2	1903.7	529.2
MedianPooling (keep_size)	1266.0	1619.0	499.1
Superpixels (max_size=64, cubic)	111.3	115.7	108.6
Superpixels (max_size=64, linear)	117.4	115.2	114.4
Superpixels (max_size=128, linear)	115.6	116.3	45.9
Superpixels (max_size=224, linear)	112.0	118.2	17.4
UniformVoronoi (250-1000k points, linear)	36.4	36.2	9.1
RegularGridVoronoi (16-31 rows/cols)	36.6	36.2	9.3
RelativeRegularGridVoronoi (7%-14% rows/cols)	37.7	37.2	3.1
Resize (nearest)	3229.3	8321.3	2121.6
Resize (linear)	3065.2	7248.8	1569.4
Resize (cubic)	2857.5	6259.3	1182.2
CropAndPad	2228.8	2440.1	1828.4
CropAndPad (edge)	2229.1	2465.8	1833.3
CropAndPad (keep_size)	1804.6	1903.5	1112.9
Crop	3531.9	4071.6	2784.9
Crop (keep_size)	2581.0	2794.1	1480.6
Pad	2220.0	2418.7	1749.7
Pad (edge)	2210.9	2425.1	1721.7
Pad (keep_size)	1789.2	1880.7	1042.6
PadToFixedSize	2155.9	14188.0	1858.8
CropToFixedSize	4144.2	41138.4	3262.9
KeepSizeByResize (CropToFixedSize(nearest))	2172.6	11017.3	1520.4
KeepSizeByResize (CropToFixedSize(linear))	2098.0	9433.9	1224.3
KeepSizeByResize (CropToFixedSize(cubic))	1992.2	7956.1	975.9
FastSnowyLandscape	1679.9	2883.6	447.7
Clouds	212.7	214.5	52.9
Fog	361.2	362.0	86.9
CloudLayer	353.2	354.2	86.2
Snowflakes	174.5	178.3	76.6
SnowflakesLayer	353.4	358.5	167.2

10.3 Heatmaps

Numbers below are for heatmaps on large images, i.e. `224x224x3`. Smaller images were skipped for brevity. The heatmaps themselves can be small (`64x64xN`) or large (`224x224xN`), with `N` denoting the number of heatmaps per `HeatmapsOnImage` instance (i.e. the number of channels in the heatmaps array), for which below `1` and `5` are used. `B=1` denotes a batch size of `1` , `B=128` one of `128`.

mbit/sec for 64x64x5 or 224x224x5 heatmaps on 224x224x3 images:

	64x64x5 on 224x224x3		224x224x5 o
Augmenter	B=1	B=128	B=1
Sequential (2xNoop)	1811.6	6545.7	14317.0
Sequential (2xNoop, random_order)	1290.8	6483.3	11412.4
SomeOf (1-3, 3xNoop)	814.4	4438.9	8120.7
SomeOf (1-3, 3xNoop, random_order)	734.9	4407.0	7510.9
OneOf (3xNoop)	1200.8	4592.7	10832.6
Sometimes (Noop)	1124.8	6425.9	10353.8
WithChannels ([1,2], Noop)	1730.2	6506.7	13820.9
Noop	2006.4	6592.0	14986.0
Lambda (return input)	1926.7	6544.8	14780.5
AssertLambda (return True)	1925.7	6527.8	14777.5
AssertShape (None, H, W, None)	1822.6	6321.0	14228.5
ChannelShuffle (0.5)	2005.9	6571.7	14996.3
Add	1999.1	6546.0	15010.3
AddElementwise	1985.0	6572.4	14994.6
AdditiveGaussianNoise	2005.5	6544.4	15006.6
AdditiveLaplaceNoise	2002.2	6546.6	15007.0
AdditivePoissonNoise	2003.5	6545.2	15037.2
Multiply	1990.3	6572.0	15011.0
MultiplyElementwise	2000.5	6532.7	14990.5
Dropout (1-5%)	2000.1	6568.4	15002.4
CoarseDropout (1-5%, size=1-10%)	1998.8	6579.9	14979.5
ReplaceElementwise	1993.7	6582.4	15063.5
ImpulseNoise	2006.6	6587.6	15017.4
SaltAndPepper	1995.9	6575.9	15051.1
CoarseSaltAndPepper	1994.3	6580.8	14983.9
Salt	1992.8	6570.6	15079.1
CoarseSalt	1986.9	6558.7	14965.5
Pepper	1998.6	6525.3	14978.7
CoarsePepper	2007.1	6513.3	15004.5
Invert (10%)	2001.0	6540.9	15036.9
JpegCompression (50-99%)	2004.2	6542.0	14930.0
Alpha (Noop)	842.8	3055.5	6924.0
AlphaElementwise (Noop)	216.9	279.7	1151.0
SimplexNoiseAlpha (Noop)	94.2	102.5	764.1
FrequencyNoiseAlpha (Noop)	108.3	121.8	810.8
GaussianBlur (sigma=(1,5))	1995.2	6549.6	15021.8
AverageBlur	1997.9	6563.8	14984.2
MedianBlur	2010.0	6547.5	15021.5
BilateralBlur	2009.0	6539.5	14965.5
MotionBlur	2004.0	6567.3	14914.4

Table 3 – continued from previous page

WithColorspace (HSV, Noop)	1797.1	6522.3	14165.3
WithHueAndSaturation	1803.6	6506.9	14155.9
MultiplyHueAndSaturation	1786.8	6513.5	13256.6
MultiplyHue	1753.9	6529.9	13036.3
MultiplySaturation	1745.1	6521.2	13044.0
AddToHueAndSaturation	2002.2	6591.6	14707.4
AddToHue	2004.7	6584.2	15035.1
AddToSaturation	1996.1	6558.1	15087.1
ChangeColorspace (HSV)	2006.0	6544.7	15076.4
Grayscale	1999.8	6555.8	15147.1
KMeansColorQuantization (2-16 colors)	2025.4	6560.4	15069.4
UniformColorQuantization (2-16 colors)	2002.5	6566.2	15089.7
GammaContrast	2005.9	6575.9	15006.6
SigmoidContrast	2014.0	6596.6	14980.7
LogContrast	2010.4	6570.3	15075.3
LinearContrast	2007.0	6564.9	15126.8
AllChannelsHistogramEqualization	2019.7	6550.6	15050.0
HistogramEqualization	2118.0	6541.2	15044.9
AllChannelsCLAHE	1999.5	6546.9	15077.3
CLAHE	2001.9	6557.7	15246.5
Convolve (3x3)	2118.1	6529.0	15005.7
Sharpen	2116.3	6578.0	14990.8
Emboss	2024.0	6586.0	14954.4
EdgeDetect	2010.2	6591.2	15027.1
DirectedEdgeDetect	2020.4	6589.9	15033.9
Canny	2028.8	6588.8	15085.6
Fliplr (p=100%)	1423.2	6095.4	12118.5
Flipud (p=100%)	1455.5	6372.2	12312.4
Affine (order=0, constant)	295.7	665.8	1168.9
Affine (order=1, constant)	293.8	665.6	1172.2
Affine (order=3, constant)	294.0	665.7	1173.7
Affine (order=1, edge)	294.2	663.3	1176.6
Affine (order=1, constant, skimage)	169.4	255.5	374.2
PiecewiseAffine (4x4, order=1, constant)	21.9	22.5	52.9
PiecewiseAffine (4x4, order=0, constant)	21.7	22.6	52.7
PiecewiseAffine (4x4, order=1, edge)	21.8	22.5	52.8
PiecewiseAffine (8x8, order=1, constant)	6.5	6.7	33.8
PerspectiveTransform	289.2	554.3	1496.4
PerspectiveTransform (keep_size)	317.4	558.4	1270.5
ElasticTransformation (order=0, constant)	85.6	101.1	1279.6
ElasticTransformation (order=1, constant)	85.7	101.7	1279.4
ElasticTransformation (order=1, nearest)	85.8	100.5	1276.9
ElasticTransformation (order=1, reflect)	85.9	97.7	1279.1
Rot90	958.4	5192.0	8719.2
Rot90 (keep_size)	678.2	1903.0	4427.4
AveragePooling	2013.5	6614.1	14378.3
AveragePooling (keep_size)	2008.3	6611.3	14285.2
MaxPooling	2010.8	6620.7	14377.5
MaxPooling (keep_size)	1998.8	6632.3	14292.3
MinPooling	1999.2	6632.3	14382.4

Table 3 – continued from previous page

MinPooling (keep_size)	2000.9	6632.6	14316.7
MedianPooling	2011.0	6627.9	14319.1
MedianPooling (keep_size)	2013.9	6629.7	14347.0
Superpixels (max_size=64, cubic)	2013.0	6657.9	14412.7
Superpixels (max_size=64, linear)	2013.6	6670.9	14474.7
Superpixels (max_size=128, linear)	2010.7	6623.2	14484.4
Superpixels (max_size=224, linear)	2008.8	6645.7	14433.6
UniformVoronoi (250-1000k points, linear)	2010.2	6623.9	14397.1
RegularGridVoronoi (16-31 rows/cols)	2002.7	6633.8	14186.9
RelativeRegularGridVoronoi (7%-14% rows/cols)	1996.0	6581.1	14244.8
Resize (nearest)	599.9	1414.1	4050.8
Resize (linear)	586.1	1348.9	3614.8
Resize (cubic)	571.3	1275.2	3196.3
CropAndPad	541.6	876.0	3611.8
CropAndPad (edge)	543.8	873.3	3587.6
CropAndPad (keep_size)	405.2	588.1	2142.2
Crop	742.3	1396.0	6703.0
Crop (keep_size)	503.2	795.6	3211.1
Pad	522.3	836.6	3004.5
Pad (edge)	521.2	837.3	3012.9
Pad (keep_size)	391.9	564.3	1859.8
PadToFixedSize	527.8	1775.4	3131.2
CropToFixedSize	788.6	3006.1	6630.7
KeepSizeByResize (CropToFixedSize(nearest))	470.3	1295.1	3214.6
KeepSizeByResize (CropToFixedSize(linear))	468.9	1249.5	2986.4
KeepSizeByResize (CropToFixedSize(cubic))	458.2	1182.5	2690.6
FastSnowyLandscape	1994.5	6602.0	14273.7
Clouds	825.0	4406.1	7776.7
Fog	2012.6	6548.7	14466.8
CloudLayer	2000.5	6555.3	14257.2
Snowflakes	817.1	4399.1	7732.0
SnowflakesLayer	2008.3	6552.2	14323.4

Number of heatmap instances per sec for 64x64x5 or 224x224x5 heatmaps on 224x224x3 images:

	64x64x5 on 224x224x3		224x224x5
Augmenter	B=1	B=128	B=1
Sequential (2xNoop)	14492.5	52365.2	9349.9
Sequential (2xNoop, random_order)	10326.6	51866.5	7453.0
SomeOf (1-3, 3xNoop)	6515.1	35511.3	5303.3
SomeOf (1-3, 3xNoop, random_order)	5879.3	35256.3	4905.1
OneOf (3xNoop)	9606.2	36741.3	7074.3
Sometimes (Noop)	8998.3	51407.2	6761.7
WithChannels ([1,2], Noop)	13841.9	52053.5	9025.9
Noop	16051.4	52735.9	9786.8
Lambda (return input)	15413.6	52358.2	9652.6
AssertLambda (return True)	15405.2	52222.0	9650.6
AssertShape (None, H, W, None)	14581.0	50567.8	9292.1
ChannelShuffle (0.5)	16047.2	52573.7	9793.5
Add	15992.5	52367.9	9802.6

Table 4 – continued from previous page

AddElementwise	15880.3	52579.4	9792.4
AdditiveGaussianNoise	16044.4	52355.3	9800.2
AdditiveLaplaceNoise	16017.8	52372.8	9800.5
AdditivePoissonNoise	16027.9	52361.3	9820.2
Multiply	15922.2	52575.8	9803.1
MultiplyElementwise	16004.3	52261.8	9789.7
Dropout (1-5%)	16000.6	52547.4	9797.5
CoarseDropout (1-5%, size=1-10%)	15990.8	52639.1	9782.5
ReplaceElementwise	15949.5	52659.1	9837.4
ImpulseNoise	16052.9	52700.5	9807.3
SaltAndPepper	15966.8	52607.1	9829.3
CoarseSaltAndPepper	15954.6	52646.4	9785.4
Salt	15942.2	52564.5	9847.6
CoarseSalt	15895.4	52469.7	9773.4
Pepper	15989.1	52202.4	9782.0
CoarsePepper	16056.6	52106.7	9798.9
Invert (10%)	16007.9	52327.4	9820.0
JpegCompression (50-99%)	16033.4	52335.7	9750.2
Alpha (Noop)	6742.3	24444.1	4521.8
AlphaElementwise (Noop)	1735.5	2237.3	751.7
SimplexNoiseAlpha (Noop)	753.3	820.3	499.0
FrequencyNoiseAlpha (Noop)	866.5	974.5	529.5
GaussianBlur (sigma=(1,5))	15961.3	52396.9	9810.1
AverageBlur	15983.1	52510.6	9785.6
MedianBlur	16080.3	52379.9	9809.9
BilateralBlur	16072.3	52315.9	9773.4
MotionBlur	16031.8	52538.4	9740.0
WithColorspace (HSV, Noop)	14376.8	52178.3	9250.8
WithHueAndSaturation	14428.5	52055.2	9244.7
MultiplyHueAndSaturation	14294.4	52107.7	8657.4
MultiplyHue	14031.0	52239.0	8513.5
MultiplySaturation	13960.5	52169.9	8518.5
AddToHueAndSaturation	16017.9	52732.8	9604.8
AddToHue	16037.9	52673.3	9818.9
AddToSaturation	15969.1	52464.7	9852.8
ChangeColorspace (HSV)	16047.9	52357.9	9845.8
Grayscale	15998.7	52446.6	9892.0
KMeansColorQuantization (2-16 colors)	16203.6	52482.9	9841.2
UniformColorQuantization (2-16 colors)	16020.1	52529.7	9854.5
GammaContrast	16047.1	52607.0	9800.2
SigmoidContrast	16111.9	52772.8	9783.3
LogContrast	16082.8	52562.6	9845.1
LinearContrast	16055.7	52519.5	9878.7
AllChannelsHistogramEqualization	16157.8	52404.9	9828.6
HistogramEqualization	16944.0	52329.6	9825.3
AllChannelsCLAHE	15995.7	52375.2	9846.4
CLAHE	16015.5	52461.5	9956.9
Convolve (3x3)	16944.7	52232.4	9799.6
Sharpen	16930.6	52624.0	9789.9
Emboss	16192.1	52688.1	9766.1

Table 4 – continued from previous page

EdgeDetect	16081.3	52730.0	9813.6
DirectedEdgeDetect	16162.9	52719.2	9818.0
Canny	16230.8	52710.4	9851.8
Fliplr (p=100%)	11386.0	48763.0	7914.1
Flipud (p=100%)	11644.2	50977.6	8040.8
Affine (order=0, constant)	2365.5	5326.1	763.3
Affine (order=1, constant)	2350.2	5324.7	765.5
Affine (order=3, constant)	2352.3	5325.9	766.5
Affine (order=1, edge)	2353.7	5306.7	768.4
Affine (order=1, constant, skimage)	1355.4	2044.0	244.4
PiecewiseAffine (4x4, order=1, constant)	175.4	180.3	34.5
PiecewiseAffine (4x4, order=0, constant)	173.8	180.5	34.4
PiecewiseAffine (4x4, order=1, edge)	174.1	180.1	34.5
PiecewiseAffine (8x8, order=1, constant)	52.3	53.7	22.1
PerspectiveTransform	2313.6	4434.0	977.2
PerspectiveTransform (keep_size)	2539.1	4467.3	829.7
ElasticTransformation (order=0, constant)	684.6	809.0	835.6
ElasticTransformation (order=1, constant)	685.3	813.7	835.5
ElasticTransformation (order=1, nearest)	686.7	803.7	833.9
ElasticTransformation (order=1, reflect)	686.9	781.7	835.4
Rot90	7667.3	41535.9	5694.2
Rot90 (keep_size)	5425.4	15223.7	2891.3
AveragePooling	16108.0	52912.6	9389.9
AveragePooling (keep_size)	16066.1	52890.2	9329.1
MaxPooling	16086.2	52965.4	9389.4
MaxPooling (keep_size)	15990.2	53058.5	9333.7
MinPooling	15993.5	53058.5	9392.6
MinPooling (keep_size)	16006.9	53060.4	9349.7
MedianPooling	16087.7	53023.6	9351.2
MedianPooling (keep_size)	16110.8	53037.3	9369.5
Superpixels (max_size=64, cubic)	16104.4	53263.3	9412.4
Superpixels (max_size=64, linear)	16109.0	53367.3	9452.9
Superpixels (max_size=128, linear)	16085.2	52985.9	9459.2
Superpixels (max_size=224, linear)	16070.1	53165.8	9426.0
UniformVoronoi (250-1000k points, linear)	16081.6	52991.4	9402.2
RegularGridVoronoi (16-31 rows/cols)	16021.4	53070.2	9264.9
RelativeRegularGridVoronoi (7%-14% rows/cols)	15968.0	52649.0	9302.7
Resize (nearest)	4799.3	11313.0	2645.4
Resize (linear)	4689.1	10791.3	2360.7
Resize (cubic)	4570.4	10201.3	2087.4
CropAndPad	4332.5	7008.1	2358.8
CropAndPad (edge)	4350.3	6986.6	2342.9
CropAndPad (keep_size)	3241.6	4704.6	1399.0
Crop	5938.7	11168.3	4377.5
Crop (keep_size)	4025.6	6364.8	2097.0
Pad	4178.2	6692.7	1962.1
Pad (edge)	4169.9	6698.1	1967.6
Pad (keep_size)	3135.3	4514.0	1214.6
PadToFixedSize	4222.2	14203.1	2044.9
CropToFixedSize	6308.6	24048.9	4330.3

Table 4 – continued from previous page

KeepSizeByResize (CropToFixedSize(nearest))	3762.0	10360.8	2099.3
KeepSizeByResize (CropToFixedSize(linear))	3751.4	9996.4	1950.3
KeepSizeByResize (CropToFixedSize(cubic))	3665.4	9459.8	1757.1
FastSnowyLandscape	15956.0	52816.3	9321.6
Clouds	6599.7	35248.4	5078.7
Fog	16100.6	52389.4	9447.7
CloudLayer	16003.9	52442.6	9310.9
Snowflakes	6536.7	35192.9	5049.5
SnowflakesLayer	16066.5	52417.9	9354.0

10.4 Keypoints and Bounding Boxes

Numbers below are for keypoints on small and large images. Each `KeypointsOnImage` instance contained `10` `Keypoint` instances. `B=1` denotes a batch size of `1` , `B=128` one of `128`.

The numbers for bounding boxes can be derived by dividing each value by 4.

Number of augmented Keypoint instances per sec (divide by 10 for KeypointsOnImage instances):

	10 KPs on 64x64x3		10 KPs o
Augmenter	B=1	B=128	B=1
Sequential (2xNoop)	117845.1	1271754.9	116128.9
Sequential (2xNoop, random_order)	38948.3	1078527.6	39444.1
SomeOf (1-3, 3xNoop)	17187.8	182412.7	17412.7
SomeOf (1-3, 3xNoop, random_order)	15297.5	180090.2	15496.1
OneOf (3xNoop)	33475.4	201685.5	34231.4
Sometimes (Noop)	27682.4	940970.8	28066.7
WithChannels ([1,2], Noop)	97010.3	1140444.1	96834.8
Noop	150981.1	1280647.5	152547.9
Lambda (return input)	142708.7	1240949.9	142966.5
AssertLambda (return True)	142859.4	1236418.2	143342.7
AssertShape (None, H, W, None)	106638.5	905189.5	108450.9
ChannelShuffle (0.5)	151875.8	1256490.9	154113.6
Add	152366.9	1266360.3	154145.7
AddElementwise	152018.9	1265638.8	153296.8
AdditiveGaussianNoise	152221.2	1257915.8	153586.9
AdditiveLaplaceNoise	152344.7	1256556.6	152549.7
AdditivePoissonNoise	150791.1	1266635.2	152699.7
Multiply	151215.1	1269893.9	153156.9
MultiplyElementwise	151501.0	1274457.9	152840.6
Dropout (1-5%)	150917.7	1270715.4	154313.9
CoarseDropout (1-5%, size=1-10%)	152883.4	1271874.4	153311.8
ReplaceElementwise	152256.2	1263005.8	152812.8
ImpulseNoise	151877.7	1261434.9	153015.4
SaltAndPepper	153000.5	1261012.2	152155.0
CoarseSaltAndPepper	151226.0	1259696.5	153405.2
Salt	152177.1	1262169.4	152970.7
CoarseSalt	150292.2	1261297.6	149935.8
Pepper	153444.5	1261691.9	150211.4

Table 5 – continued from previous page

CoarsePepper	152280.2	1259434.5	151182.4
Invert (10%)	153145.7	1264599.4	152459.2
JpegCompression (50-99%)	152236.0	1270393.7	152342.9
Alpha (Noop)	23645.3	405669.6	23982.6
AlphaElementwise (Noop)	12494.4	22371.1	7059.9
SimplexNoiseAlpha (Noop)	2703.5	3065.8	1840.6
FrequencyNoiseAlpha (Noop)	3129.0	3814.4	2198.9
GaussianBlur (sigma=(1,5))	151802.5	1242964.9	154881.9
AverageBlur	150304.7	1253134.7	151146.1
MedianBlur	152096.1	1257904.0	151599.5
BilateralBlur	152809.1	1260036.5	150718.8
MotionBlur	153229.6	1264048.5	151687.2
WithColorspace (HSV, Noop)	109771.7	1211901.0	109518.5
WithHueAndSaturation	110064.6	1208509.1	110056.9
MultiplyHueAndSaturation	110358.0	1199819.7	108910.0
MultiplyHue	100584.4	1143882.4	99356.6
MultiplySaturation	100817.3	1143674.5	99957.2
AddToHueAndSaturation	153052.6	1251988.2	151059.0
AddToHue	151903.3	1252179.0	152361.3
AddToSaturation	153332.3	1255361.7	151322.4
ChangeColorspace (HSV)	153737.0	1260859.2	153283.8
Grayscale	153703.2	1259484.8	152053.8
KMeansColorQuantization (2-16 colors)	155598.2	1260118.4	150182.8
UniformColorQuantization (2-16 colors)	153718.2	1253918.2	151320.6
GammaContrast	155114.8	1258490.8	150885.1
SigmoidContrast	154779.0	1263570.6	150888.7
LogContrast	154281.8	1263292.1	151375.2
LinearContrast	153508.2	1265834.8	151062.6
AllChannelsHistogramEqualization	150080.7	1253029.5	145686.1
HistogramEqualization	153661.9	1251161.5	150362.2
AllChannelsCLAHE	153465.1	1259210.0	150306.5
CLAHE	156022.6	1267221.1	151630.6
Convolve (3x3)	154329.1	1258661.9	147537.8
Sharpen	154140.0	1262945.4	151269.6
Emboss	153480.1	1261828.3	151669.0
EdgeDetect	151433.5	1266341.4	150504.3
DirectedEdgeDetect	151453.5	1264529.9	151118.9
Canny	152562.7	1261503.1	152644.1
Fliplr (p=100%)	41840.8	767543.2	42707.5
Flipud (p=100%)	42311.4	759172.6	42454.8
Affine (order=0, constant)	7878.6	35591.2	7909.1
Affine (order=1, constant)	7829.1	35621.3	7913.4
Affine (order=3, constant)	7826.7	35533.0	7912.8
Affine (order=1, edge)	7781.4	35404.6	7900.8
Affine (order=1, constant, skimage)	7835.9	35408.7	7941.2
PiecewiseAffine (4x4, order=1, constant)	428.7	443.6	125.2
PiecewiseAffine (4x4, order=0, constant)	432.1	442.9	124.9
PiecewiseAffine (4x4, order=1, edge)	428.3	444.1	124.7
PiecewiseAffine (8x8, order=1, constant)	111.7	112.2	61.2
PerspectiveTransform	12129.5	25392.6	12296.1

Table 5 – continued from previous page

PerspectiveTransform (keep_size)	10071.7	18472.1	10195.8
ElasticTransformation (order=0, constant)	1414.1	1668.3	1225.3
ElasticTransformation (order=1, constant)	1615.8	1699.2	1415.4
ElasticTransformation (order=1, nearest)	1316.7	1680.3	1402.2
ElasticTransformation (order=1, reflect)	1448.0	1690.5	1472.5
Rot90	20993.4	148342.8	23813.4
Rot90 (keep_size)	20933.8	145783.9	23829.1
AveragePooling	148330.9	1291550.6	147253.0
AveragePooling (keep_size)	148397.4	1292584.1	148131.9
MaxPooling	151398.9	1289432.0	150720.6
MaxPooling (keep_size)	150419.7	1293971.5	150753.1
MinPooling	151048.1	1295758.9	153239.0
MinPooling (keep_size)	150087.8	1297152.1	151253.3
MedianPooling	151568.5	1291223.5	152510.9
MedianPooling (keep_size)	149866.2	1292432.6	151905.2
Superpixels (max_size=64, cubic)	151120.7	1291059.9	151960.2
Superpixels (max_size=64, linear)	151273.3	1292381.8	151455.4
Superpixels (max_size=128, linear)	152195.5	1290679.2	153615.0
Superpixels (max_size=224, linear)	150319.1	1291764.0	154444.6
UniformVoronoi (250-1000k points, linear)	151222.4	1287975.0	151055.4
RegularGridVoronoi (16-31 rows/cols)	146866.2	1290315.2	151747.6
RelativeRegularGridVoronoi (7%-14% rows/cols)	150527.7	1293253.5	152772.0
Resize (nearest)	17175.4	52737.9	17414.3
Resize (linear)	17243.3	52844.3	17381.1
Resize (cubic)	17180.8	52668.0	17451.6
CropAndPad	13521.4	22962.3	15101.4
CropAndPad (edge)	13488.6	22848.6	15046.7
CropAndPad (keep_size)	11022.8	16300.5	11953.3
Crop	16725.9	30934.7	18981.1
Crop (keep_size)	12868.3	20043.9	14234.8
Pad	13565.9	23164.6	15140.9
Pad (edge)	13603.5	23199.1	15087.0
Pad (keep_size)	10961.7	16365.4	11954.7
PadToFixedSize	13644.6	128647.2	15016.8
CropToFixedSize	18874.6	129670.2	21340.0
KeepSizeByResize (CropToFixedSize(nearest))	11664.4	46163.9	12589.2
KeepSizeByResize (CropToFixedSize(linear))	11611.3	45703.5	12591.7
KeepSizeByResize (CropToFixedSize(cubic))	11649.4	45623.2	12575.0
FastSnowyLandscape	155068.9	1295390.0	152534.9
Clouds	17421.8	188714.3	17714.3
Fog	153315.5	1270560.0	152950.3
CloudLayer	155061.3	1278450.8	153938.2
Snowflakes	17105.7	186278.8	17504.0
SnowflakesLayer	153196.1	1270147.2	154954.3

CHAPTER 11

dtype support

The function `augment_images()`, which all augmenters in `imgaug` offer, works by default with numpy arrays. In most cases, these arrays will have the numpy dtype `uint8`, i.e. the images will have values in the range `[0, 255]`. This is the datatype returned by most image loading functions. Sometimes however you may want to augment other datatypes, e.g. `float64`. While all augmenters support `uint8`, the support for other datatypes varies. The tables further below show which datatype is supported by which augmenter (alongside the dtype support in some helper functions). The API of each augmenter may contain more details.

Note: Whenever possible it is suggested to use `uint8` as that is the most thoroughly tested dtype. In general, the use of large dtypes (i.e. `uint64`, `int64`, `float128`) is discouraged, even when they are marked as supported. That is because writing correct tests for these dtypes can be difficult as no larger dtypes are available to which values can be temporarily cast. Additionally, when using inputs for which precise discrete values are important (e.g. segmentation maps, where an accidental change by `1` would break the map), medium sized dtypes (`uint32`, `int32`) should be used with caution. This is because other libraries may convert temporarily to `float64`, which could lead to inaccuracies for some numbers.

11.1 Legend

Support level (color of table cells):

- Green: Datatype is considered supported by the augmenter.
- Yellow: Limited support for the datatype, e.g. due to inaccuracies around large values. See the API for the respective augmenter for more details.
- Red: Datatype is not supported by the augmenter.

Test level (symbols in table cells):

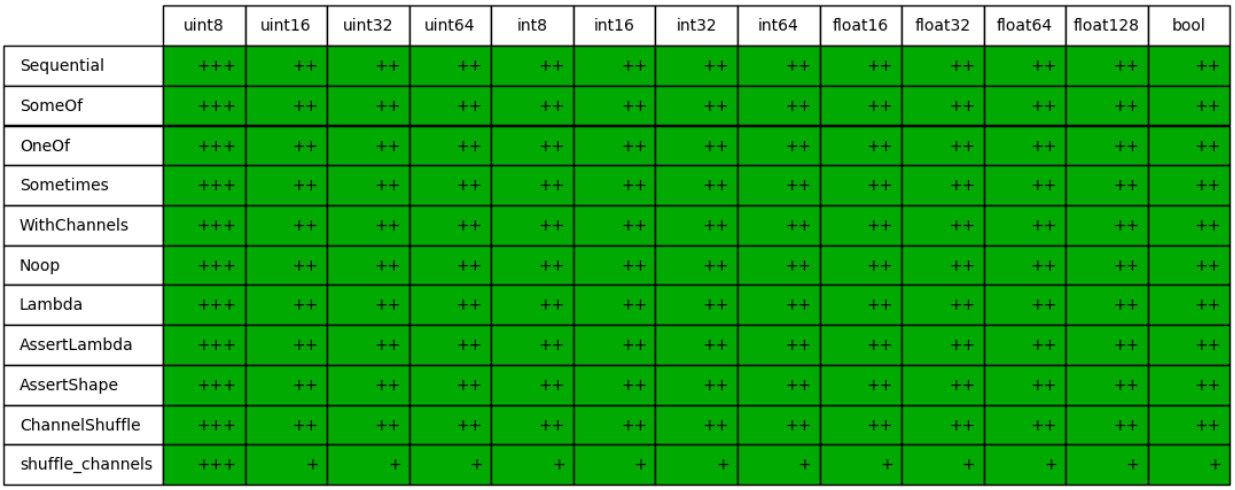
- +++: Datatype support is thoroughly tested (via unittests or integration tests).
- ++: Datatype support is tested, though not thoroughly.
- +: Datatype support is indirectly tested via tests for other augmenters.
- −: Datatype support is not tested.

- `?`: Unknown support level for the datatype.

11.2 imgaug helper functions

	uint8	uint16	uint32	uint64	int8	int16	int32	int64	float16	float32	float64	float128	bool
draw_text	+++									-			
imresize_many_images	+++	++			++	++	++		++	++	++		++
imresize_single_image	+++	++			++	++	++		++	++	++		++
pad	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	++
pad_to_aspect_ratio	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	++
pad_to_multiples_of	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	++
pool	+++	++	++		++	++	++		++	++	++	++	++
avg_pool	+++	++	++		++	++	++		++	++	++	++	++
max_pool	+++	++	++		++	++	++		++	++	++	++	++
min_pool	+++	++	++		++	++	++		++	++	++	++	++
median_pool	+++	++	++		++	++	++		++	++	++	++	++
draw_grid	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++
show_grid	-	?	?	?	?	?	?	?	?	?	?	?	?
imshow	-	?	?	?	?	?	?	?	?	?	?	?	?

Fig. 1: Dtype support of helper functions in `imgaug`, e.g. `import imgaug; imgaug.imresize_single_image(array, size)`.



	uint8	uint16	uint32	uint64	int8	int16	int32	int64	float16	float32	float64	float128	bool
Sequential	+++	++	++	++	++	++	++	++	++	++	++	++	++
SomeOf	+++	++	++	++	++	++	++	++	++	++	++	++	++
OneOf	+++	++	++	++	++	++	++	++	++	++	++	++	++
Sometimes	+++	++	++	++	++	++	++	++	++	++	++	++	++
WithChannels	+++	++	++	++	++	++	++	++	++	++	++	++	++
Noop	+++	++	++	++	++	++	++	++	++	++	++	++	++
Lambda	+++	++	++	++	++	++	++	++	++	++	++	++	++
AssertLambda	+++	++	++	++	++	++	++	++	++	++	++	++	++
AssertShape	+++	++	++	++	++	++	++	++	++	++	++	++	++
ChannelShuffle	+++	++	++	++	++	++	++	++	++	++	++	++	++
shuffle_channels	+++	+	+	+	+	+	+	+	+	+	+	+	+

Fig. 2: Dtype support for `augment_images(arrays)`, `augment_image(arr)` and helper functions in `imgaug.augmenters.meta`.

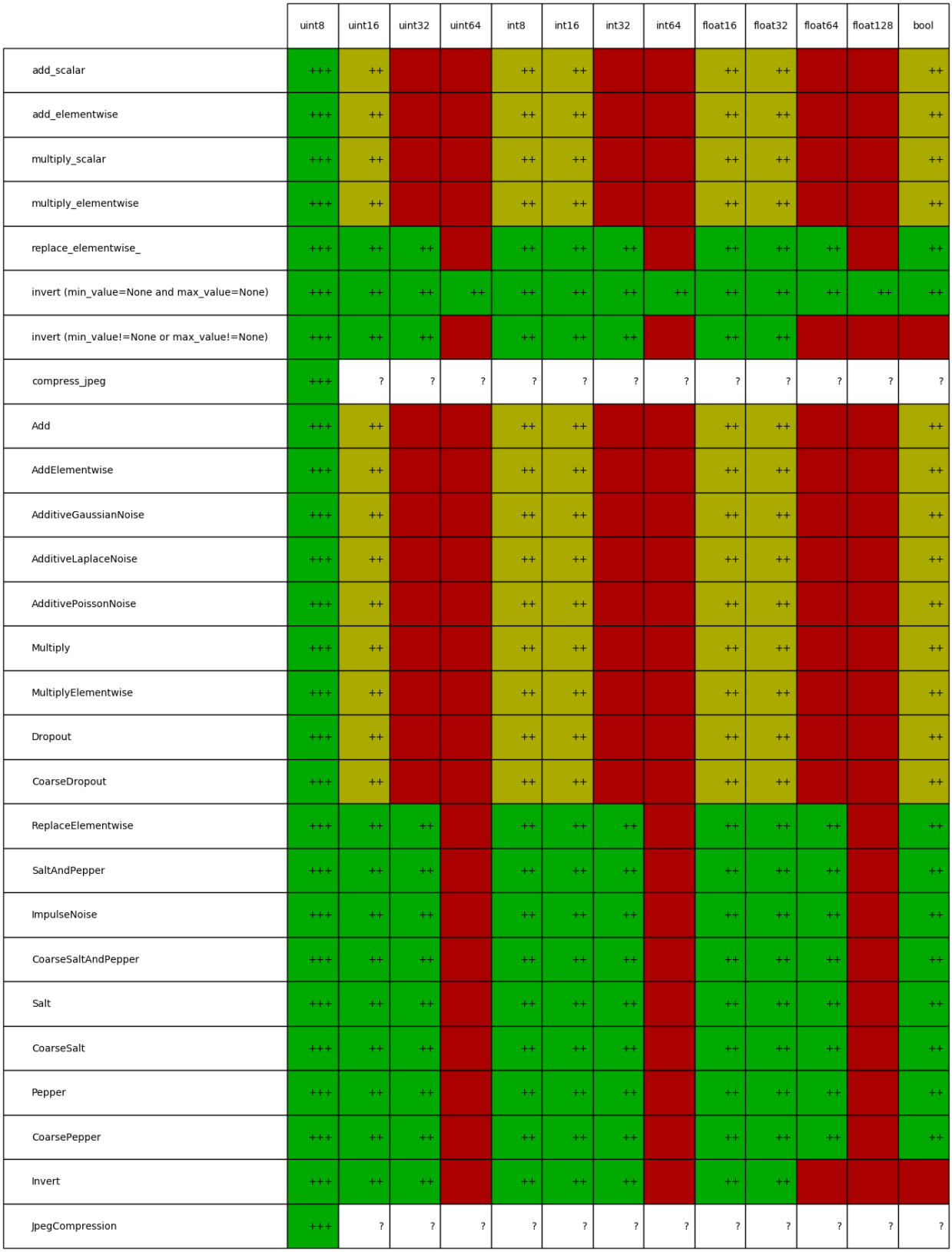


Fig. 3: Dtype support for `augment_images(arrays)`, `augment_image(arr)` and helper functions in `imgaug.augmenters.arithmetic`.

	uint8	uint16	uint32	uint64	int8	int16	int32	int64	float16	float32	float64	float128	bool
blend_alpha	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++		+++
Alpha	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++		+++
AlphaElementwise	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++		+++
SimplexNoiseAlpha	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++		+++
FrequencyNoiseAlpha	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++		+++

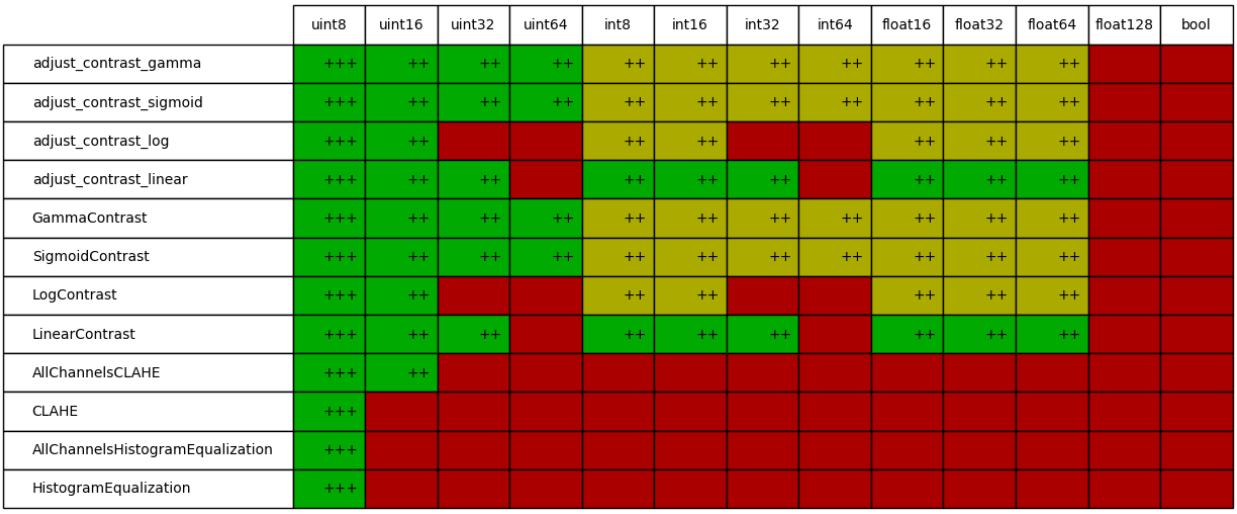
Fig. 4: Dtype support for `augment_images(arrays)`, `augment_image(arr)` and helper functions in `imgaug.augmenters.blend`.

	uint8	uint16	uint32	uint64	int8	int16	int32	int64	float16	float32	float64	float128	bool
blur_gaussian_ (backend="auto")	+++	++	++	++	++	++	++	++	++	++	++		++
blur_gaussian_ (backend="cv2")	+++	++			++	++	++		++	++	++		++
blur_gaussian_ (backend="scipy")	+++	++	++	++	++	++	++	++	++	++	++		++
GaussianBlur	+++	++	++	++	++	++	++	++	++	++	++		++
AverageBlur	+++	++			++	++			++	++	++		++
MedianBlur	+++	?	?	?	?	?	?	?	?	?	?	?	?
BilateralBlur	-	?	?	?	?	?	?	?	?	?	?	?	?
MotionBlur	+++	++			++	++			++	++	++		++

Fig. 5: Dtype support for `augment_images(arrays)`, `augment_image(arr)` and helper functions in `imgaug.augmenters.blur`.

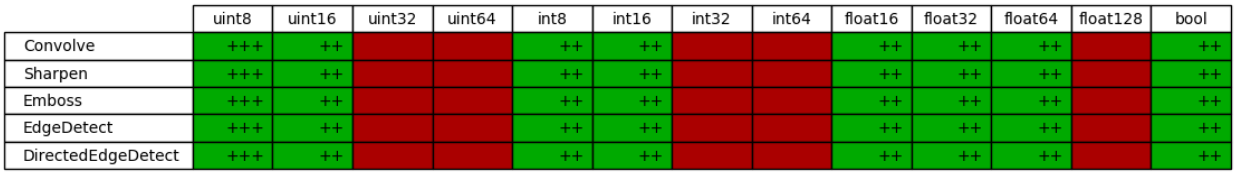
	uint8	uint16	uint32	uint64	int8	int16	int32	int64	float16	float32	float64	float128	bool
change_colorspace_	+++												
change_colorspaces_	+++												
WithColorspace	+++												
WithHueAndSaturation	+++												
AddToHueAndSaturation	+++												
ChangeColorspace	+++												
Grayscale	+++												
KMeansColorQuantization (image size <= max_size)	+++												
KMeansColorQuantization (image size > max_size)	+++												
quantize_colors_kmeans	+++												
UniformColorQuantization (image size <= max_size)	+++												
UniformColorQuantization (image size > max_size)	+++												
quantize_colors_uniform	+++												

Fig. 6: Dtype support for `augment_images(arrays)`, `augment_image(arr)` and helper functions in `imgaug.augmenters.color`.



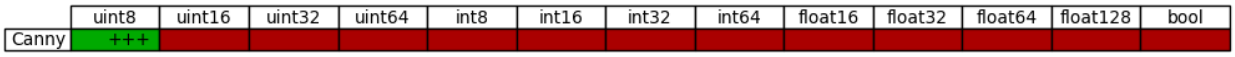
	uint8	uint16	uint32	uint64	int8	int16	int32	int64	float16	float32	float64	float128	bool
adjust_contrast_gamma	+++	++	++	++	++	++	++	++	++	++	++		
adjust_contrast_sigmoid	+++	++	++	++	++	++	++	++	++	++	++		
adjust_contrast_log	+++	++			++	++			++	++	++		
adjust_contrast_linear	+++	++	++		++	++	++		++	++	++		
GammaContrast	+++	++	++	++	++	++	++	++	++	++	++		
SigmoidContrast	+++	++	++	++	++	++	++	++	++	++	++		
LogContrast	+++	++			++	++			++	++	++		
LinearContrast	+++	++	++		++	++	++		++	++	++		
AllChannelsCLAHE	+++	++											
CLAHE	+++												
AllChannelsHistogramEqualization	+++												
HistogramEqualization	+++												

Fig. 7: Dtype support for `augment_images(arrays)`, `augment_image(arr)` and helper functions in `imgaug.augmenters.contrast`.



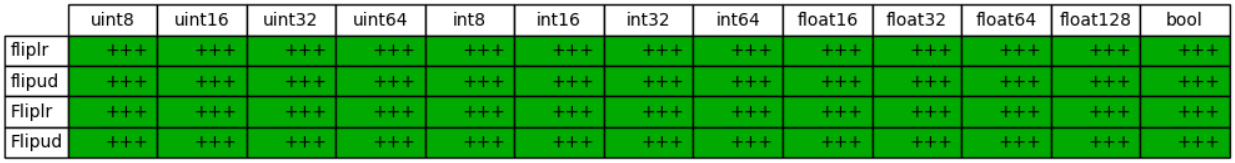
	uint8	uint16	uint32	uint64	int8	int16	int32	int64	float16	float32	float64	float128	bool
Convolve	+++	++			++	++			++	++	++		++
Sharpen	+++	++			++	++			++	++	++		++
Emboss	+++	++			++	++			++	++	++		++
EdgeDetect	+++	++			++	++			++	++	++		++
DirectedEdgeDetect	+++	++			++	++			++	++	++		++

Fig. 8: Dtype support for `augment_images(arrays)`, `augment_image(arr)` and helper functions in `imgaug.augmenters.convolutional`.



	uint8	uint16	uint32	uint64	int8	int16	int32	int64	float16	float32	float64	float128	bool
Canny	+++												

Fig. 9: Dtype support for `augment_images(arrays)`, `augment_image(arr)` and helper functions in `imgaug.augmenters.edges`.



	uint8	uint16	uint32	uint64	int8	int16	int32	int64	float16	float32	float64	float128	bool
fliplr	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++
flipud	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++
Fliplr	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++
Flipud	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++

Fig. 10: Dtype support for `augment_images(arrays)`, `augment_image(arr)` and helper functions in `imgaug.augmenters.flip`.

	uint8	uint16	uint32	uint64	int8	int16	int32	int64	float16	float32	float64	float128	bool
Affine (backend="skimage", order in [0, 1])	++	++	++		++	++	++		++	++	++		++
Affine (backend="skimage", order in [3, 4])	++	++	++		++	++	++		++	++	++		++
Affine (backend="skimage", order=5])	++	++	++		++	++	++		++	++	-		++
Affine (backend="cv2", order=0)	++	++			++	++	++		++	++	++		++
Affine (backend="cv2", order=1)	+++	++			++	++			++	++	++		++
Affine (backend="cv2", order=3)	++	++			++	++			++	++	++		++
AffineCv2	+++	?	?	?	?	?	?	?	?	?	?	?	?
PiecewiseAffine	+++	++	++		++	++	++		++	++	++		++
PerspectiveTransform (keep_size=False)	+++	++			++	++			++	++	++		++
PerspectiveTransform (keep_size=True)	+++	++			++	++			++	++	++		++
ElasticTransformation	+++	++	++	++	++	++	++	++	++	++	++		++
Rot90 (keep_size=False)	+++	++	++	++	++	++	++	++	++	++	++	++	++
Rot90 (keep_size=True)	+++	++			++	++	++		++	++	++		++

Fig. 11: Dtype support for `augment_images(arrays)`, `augment_image(arr)` and helper functions in `imgaug.augmenters.geometric`.

	uint8	uint16	uint32	uint64	int8	int16	int32	int64	float16	float32	float64	float128	bool
Superpixels (image size <= max_size)	+++	++	++	-	++	++	++	-					++
Superpixels (image size > max_size)	+++	++			++	++	++						++
Voronoi (image size <= max_size)	+++												
Voronoi (image size > max_size)	+++												
UniformVoronoi	+++												
RegularGridVoronoi	+++												
RelativeRegularGridVoronoi	+++												

Fig. 12: Dtype support for `augment_images(arrays)`, `augment_image(arr)` and helper functions in `imgaug.augmenters.segmentation`.

	uint8	uint16	uint32	uint64	int8	int16	int32	int64	float16	float32	float64	float128	bool
Resize	+++	++			++	++	++		++	++	++		++
CropAndPad (keep_size=False)	+++	++	++	++	++	++	++	++	++	++	++	++	++
CropAndPad (keep_size=True)	+++	++			++	++	++		++	++	++		++
Pad	+++	++			++	++	++		++	++	++		++
Crop	+++	++			++	++	++		++	++	++		++
PadToFixedSize	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	++
CropToFixedSize	+++	++	++	++	++	++	++	++	++	++	++	++	++
KeepSizeByResize	+++	++			++	++	++		++	++	++		++

Fig. 13: Dtype support for `augment_images(arrays)`, `augment_image(arr)` and helper functions in `imgaug.augmenters.size`.

11.3 imgaug.augmenters.meta

11.4 imgaug.augmenters.arithmetic

11.5 imgaug.augmenters.blend

11.6 imgaug.augmenters.blur

11.7 imgaug.augmenters.color

11.8 imgaug.augmenters.contrast

11.9 imgaug.augmenters.convolutional

11.10 imgaug.augmenters.edges

11.11 imgaug.augmenters.flip

11.12 imgaug.augmenters.geometric

11.13 imgaug.augmenters.segmentation

11.14 imgaug.augmenters.size

11.15 imgaug.augmenters.weather

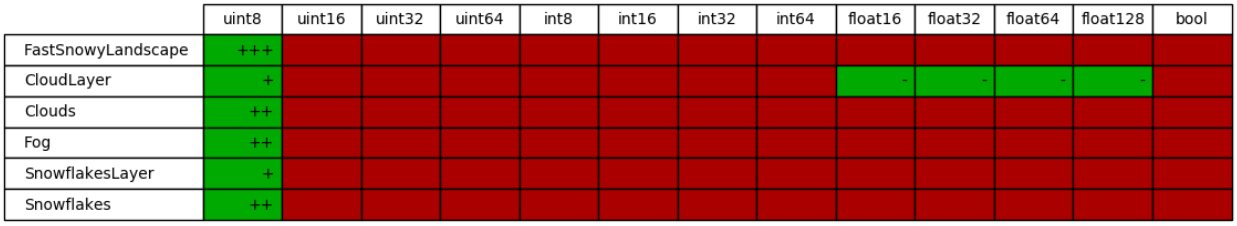


Fig. 14: Dtype support for `augment_images(arrays)`, `augment_image(arr)` and helper functions in `imgaug.augmenters.weather`.

CHAPTER 12

Jupyter Notebooks

Several jupyter notebooks are available that provide tutorials about *imgaug*'s functionality. They are hosted at imgaug-doc/notebooks. The notebooks can be downloaded to interactively modify the examples.

List of Notebooks:

- A01 - Load and Augment an Image
- A02 - Stochastic and Deterministic Augmentation
- A03 - Multicore Augmentation
- B01 - Augment Keypoints (aka Landmarks)
- B02 - Augment Bounding Boxes
- B03 - Augment Polygons
- B06 - Augment Line Strings
- B04 - Augment Heatmaps
- B05 - Augment Segmentation Maps
- C01 - Using Probability Distributions as Parameters
- C02 - Using imgaug with more Control Flow
- C03 - Copying Random States and Using Multiple Augmentation Sequences

CHAPTER 13

API

13.1 imgaug

imgaug.imgaug.**BackgroundAugmenter**(**args*, ***kwargs*)

imgaug.imgaug.**Batch**(**args*, ***kwargs*)

imgaug.imgaug.**BatchLoader**(**args*, ***kwargs*)

imgaug.imgaug.**BoundingBox**(**args*, ***kwargs*)

imgaug.imgaug.**BoundingBoxesOnImage**(**args*, ***kwargs*)

exception imgaug.imgaug.**DeprecationWarning**

Bases: Warning

Warning for deprecated calls.

Since python 2.7 DeprecatedWarning is silent by default. So we define our own DeprecatedWarning here so that it is not silent by default.

imgaug.imgaug.**HeatmapsOnImage**(**args*, ***kwargs*)

class imgaug.imgaug.**HooksHeatmaps**(*activator=None*, *propagator=None*, *preprocessor=None*, *postprocessor=None*)

Bases: *imgaug.imgaug.HooksImages*

Class to intervene with heatmap augmentation runs.

This is e.g. useful to dynamically deactivate some augmenters.

This class is currently the same as the one for images. This may or may not change in the future.

Methods

is_activated(self, images, augmenter, ...)	Estimate whether an augmenter may be executed.

Continued on next page

Table 1 – continued from previous page

is_propagating(self, images, augmenter, ...)	Estimate whether an augmenter may call its children.
postprocess(self, images, augmenter, parents)	Postprocess input data per augmenter after augmentation.
preprocess(self, images, augmenter, parents)	Preprocess input data per augmenter before augmentation.

class imgaug.imgaug.**HooksImages**(*activator=None*, *propagator=None*, *preprocessor=None*, *postprocessor=None*)

Bases: object

Class to intervene with image augmentation runs.

This is e.g. useful to dynamically deactivate some augmenters.

Parameters

- **activator** (*None or callable, optional*) – A function that gives permission to execute an augmenter. The expected interface is:

```
``f(images, augmenter, parents, default)``
```

  where images are the input images to augment, augmenter is the instance of the augmenter to execute, parents are previously executed augmenters and default is an expected default value to be returned if the activator function does not plan to make a decision for the given inputs.

- **propagator** (*None or callable, optional*) – A function that gives permission to propagate the augmentation further to the children of an augmenter. This happens after the activator. In theory, an augmenter may augment images itself (if allowed by the activator) and then execute child augmenters afterwards (if allowed by the propagator). If the activator returned False, the propagation step will never be executed. The expected interface is:

```
``f(images, augmenter, parents, default)``
```

  with all arguments having identical meaning to the activator.

- **preprocessor** (*None or callable, optional*) – A function to call before an augmenter performed any augmentations. The interface is:

```
f(images, augmenter, parents)
```

  with all arguments having identical meaning to the activator. It is expected to return the input images, optionally modified.

- **postprocessor** (*None or callable, optional*) – A function to call after an augmenter performed augmentations. The interface is the same as for the *preprocessor*.

Examples

```
>>> import numpy as np
>>> import imgaug as ia
>>> import imgaug.augmenters as iaa
>>> seq = iaa.Sequential([
>>>     iaa.GaussianBlur(3.0, name="blur"),
>>>     iaa.Dropout(0.05, name="dropout"),
>>>     iaa.Affine(translate_px=-5, name="affine")
>>> ])
```

(continues on next page)

(continued from previous page)

```
>>> images = [np.zeros((10, 10), dtype=np.uint8)]
>>>
>>> def activator(images, augmenter, parents, default):
>>>     return False if augmenter.name in ["blur", "dropout"] else default
>>>
>>> seq_det = seq.to_deterministic()
>>> images_aug = seq_det.augment_images(images)
>>> heatmaps = [np.random.rand(*(3, 10, 10))]
>>> heatmaps_aug = seq_det.augment_images(
>>>     heatmaps,
>>>     hooks=ia.HooksImages(activator=activator)
>>> )
```

This augments images and their respective heatmaps in the same way. The heatmaps however are only modified by `Affine`, not by `GaussianBlur` or `Dropout`.

Methods

is_activated(self, images, augmenter, ...)	Estimate whether an augmenter may be executed.
is_propagating(self, images, augmenter, ...)	Estimate whether an augmenter may call its children.
postprocess(self, images, augmenter, parents)	Postprocess input data per augmenter after augmentation.
preprocess(self, images, augmenter, parents)	Preprocess input data per augmenter before augmentation.

is_activated (*self*, *images*, *augmenter*, *parents*, *default*)
Estimate whether an augmenter may be executed.

This also affects propagation of data to child augmenters.

Returns If `True`, the augmenter may be executed. Otherwise `False`.

Return type bool

is_propagating (*self*, *images*, *augmenter*, *parents*, *default*)
Estimate whether an augmenter may call its children.

This function decides whether an augmenter with children is allowed to call these in order to further augment the inputs. Note that if the augmenter itself performs augmentations (before/after calling its children), these may still be executed, even if this method returns `False`.

Returns If `True`, the augmenter may propagate data to its children. Otherwise `False`.

Return type bool

postprocess (*self*, *images*, *augmenter*, *parents*)
Postprocess input data per augmenter after augmentation.

Returns The input images, optionally modified.

Return type (N,H,W,C) ndarray or (N,H,W) ndarray or list of (H,W,C) ndarray or list of (H,W) ndarray

preprocess (*self*, *images*, *augmenter*, *parents*)
Preprocess input data per augmenter before augmentation.

Returns The input images, optionally modified.

Return type (N,H,W,C) ndarray or (N,H,W) ndarray or list of (H,W,C) ndarray or list of (H,W) ndarray

class imgaug.imgaug.**HooksKeypoints**(*activator=None*, *propagator=None*, *preprocessor=None*, *postprocessor=None*)

Bases: *imgaug.imgaug.HooksImages*

Class to intervene with keypoint augmentation runs.

This is e.g. useful to dynamically deactivate some augmenters.

This class is currently the same as the one for images. This may or may not change in the future.

Methods

is_activated(self, images, augmenter, ...)	Estimate whether an augmenter may be executed.
is_propagating(self, images, augmenter, ...)	Estimate whether an augmenter may call its children.
postprocess(self, images, augmenter, parents)	Postprocess input data per augmenter after augmentation.
preprocess(self, images, augmenter, parents)	Preprocess input data per augmenter before augmentation.

imgaug.imgaug.**Keypoint**(**args*, ***kwargs*)

imgaug.imgaug.**KeypointsOnImage**(**args*, ***kwargs*)

imgaug.imgaug.**MultiPolygon**(**args*, ***kwargs*)

imgaug.imgaug.**Polygon**(**args*, ***kwargs*)

imgaug.imgaug.**PolygonsOnImage**(**args*, ***kwargs*)

imgaug.imgaug.**SegmentationMapsOnImage**(**args*, ***kwargs*)

imgaug.imgaug.**angle_between_vectors**(*v1*, *v2*)

Calculcate the angle in radians between vectors *v1* and *v2*.

From http://stackoverflow.com/questions/2827393/angles-between-two-n-dimensional-vectors-in-python

Parameters

- **v1** (*(N,) ndarray*) – First vector.
- **v2** (*(N,) ndarray*) – Second vector.

Returns Angle in radians.

Return type float

Examples

```
>>> angle_between_vectors(np.float32([1, 0, 0]), np.float32([0, 1, 0]))
1.570796...
```

```
>>> angle_between_vectors(np.float32([1, 0, 0]), np.float32([1, 0, 0]))
0.0
```

```
>>> angle_between_vectors(np.float32([1, 0, 0]), np.float32([-1, 0, 0]))
3.141592...
```

imgaug.imgaug.**avg_pool**(*arr*, *block_size*, *pad_mode='reflect'*, *pad_cval=128*, *preserve_dtype=True*, *cval=None*)

Resize an array using average pooling.

Defaults to `pad_mode="reflect"` to ensure that padded values do not affect the average.

dtype support:

```
See :func:`imgaug.imgaug.pool`.
```

Parameters

- **arr** (*(H,W) ndarray or (H,W,C) ndarray*) – Image-like array to pool. See *imgaug.imgaug.pool()* for details.
- **block_size** (*int or tuple of int or tuple of int*) – Size of each block of values to pool. See *imgaug.imgaug.pool()* for details.
- **pad_mode** (*str, optional*) – Padding mode to use if the array cannot be divided by *block_size* without remainder. See *imgaug.imgaug.pad()* for details.
- **pad_cval** (*number, optional*) – Padding value. See *imgaug.imgaug.pool()* for details.
- **preserve_dtype** (*bool, optional*) – Whether to preserve the input array dtype. See *imgaug.imgaug.pool()* for details.
- **cval** (*None or number, optional*) – Deprecated. Old name for *pad_cval*.

Returns Array after average pooling.

Return type (H',W') ndarray or (H',W',C') ndarray

imgaug.imgaug.**caller_name**()

Return the name of the caller, e.g. a function.

Returns The name of the caller as a string

Return type str

imgaug.imgaug.**compute_geometric_median**(**args*, ***kwargs*)

imgaug.imgaug.**compute_line_intersection_point**(*x1*, *y1*, *x2*, *y2*, *x3*, *y3*, *x4*, *y4*)

Compute the intersection point of two lines.

Taken from https://stackoverflow.com/a/20679579 .

Parameters

- **x1** (*number*) – x coordinate of the first point on line 1. (The lines extends beyond this point.)
- **y1** (*number*) – y coordinate of the first point on line 1. (The lines extends beyond this point.)
- **x2** (*number*) – x coordinate of the second point on line 1. (The lines extends beyond this point.)
- **y2** (*number*) – y coordinate of the second point on line 1. (The lines extends beyond this point.)
- **x3** (*number*) – x coordinate of the first point on line 2. (The lines extends beyond this point.)
- **y3** (*number*) – y coordinate of the first point on line 2. (The lines extends beyond this point.)
- **x4** (*number*) – x coordinate of the second point on line 2. (The lines extends beyond this point.)

- **y4** (*number*) – y coordinate of the second point on line 2. (The lines extends beyond this point.)

Returns The coordinate of the intersection point as a `tuple (x, y)`. If the lines are parallel (no intersection point or an infinite number of them), the result is `False`.

Return type tuple of number or bool

`imgaug.imgaug.`**`compute_paddings_for_aspect_ratio`**(*arr*, *aspect_ratio*)

Compute pad amounts required to fulfill an aspect ratio.

"Pad amounts" here denotes the number of pixels that have to be added to each side to fulfill the desired constraint.

The aspect ratio is given as `ratio = width / height`. Depending on which dimension is smaller (height or width), only the corresponding sides (top/bottom or left/right) will be padded.

The axis-wise padding amounts are always distributed equally over the sides of the respective axis (i.e. left and right, top and bottom). For odd pixel amounts, one pixel will be left over after the equal distribution and could be added to either side of the axis. This function will always add such a left over pixel to the bottom (y-axis) or right (x-axis) side.

Parameters

- **arr** (*(H,W) ndarray or (H,W,C) ndarray*) – Image-like array for which to compute pad amounts.
- **aspect_ratio** (*float*) – Target aspect ratio, given as width/height. E.g. `2.0` denotes the image having twice as much width as height.

Returns Required padding amounts to reach the target aspect ratio, given as a `tuple` of the form `(top, right, bottom, left)`.

Return type tuple of int

`imgaug.imgaug.`**`compute_paddings_to_reach_multiples_of`**(*arr*, *height_multiple*, *width_multiple*)

Compute pad amounts until img height/width are multiples of given values.

See *imgaug.imgaug.compute_paddings_for_aspect_ratio()* for an explanation of how the required padding amounts are distributed per image axis.

Parameters

- **arr** (*(H,W) ndarray or (H,W,C) ndarray*) – Image-like array for which to compute pad amounts.
- **height_multiple** (*None or int*) – The desired multiple of the height. The computed padding amount will reflect a padding that increases the y axis size until it is a multiple of this value.
- **width_multiple** (*None or int*) – The desired multiple of the width. The computed padding amount will reflect a padding that increases the x axis size until it is a multiple of this value.

Returns Required padding amounts to reach multiples of the provided values, given as a `tuple` of the form `(top, right, bottom, left)`.

Return type tuple of int

`imgaug.imgaug.`**`copy_random_state`**(*random_state*, *force_copy=False*)

Deprecated. Use `imgaug.random.copy_generator_unless_global_rng` instead.

Copy an existing numpy (random number) generator.

Parameters

- **random_state** (*numpy.random.Generator or numpy.random.RandomState*) – The generator to copy.
- **force_copy** (*bool, optional*) – If `True`, this function will always create a copy of every random state. If `False`, it will not copy numpy's default random state, but all other random states.

Returns **rs_copy** – The copied random state.

Return type numpy.random.RandomState

`imgaug.imgaug.`**`current_random_state`**`()`

Deprecated. Use `imgaug.random.get_global_rng` instead.

Get or create the current global RNG of imgaug.

Note that the first call to this function will create a global RNG.

Returns The global RNG to use.

Return type *imgaug.random.RNG*

class `imgaug.imgaug.`**`deprecated`**`(`*alt_func=None*, *behavior='warn'*, *removed_version=None*, *comment=None*`)`

Bases: `object`

Decorator to mark deprecated functions with warning.

Adapted from <https://github.com/scikit-image/scikit-image/blob/master/skimage/_shared/utils.py>.

Parameters

- **alt_func** (*None or str, optional*) – If given, tell user what function to use instead.
- **behavior** (*{'warn', 'raise'}, optional*) – Behavior during call to deprecated function: `warn` means that the user is warned that the function is deprecated; `raise` means that an error is raised.
- **removed_version** (*None or str, optional*) – The package version in which the deprecated function will be removed.
- **comment** (*None or str, optional*) – An optional comment that will be appended to the warning message.

Methods

`__call__`(self, func)	Call self as a function.

`imgaug.imgaug.`**`derive_random_state`**`(`*random_state*`)`

Deprecated. Use `imgaug.random.derive_generator_` instead.

Derive a child numpy random generator from another one.

Parameters **random_state** (*numpy.random.Generator or numpy.random.RandomState*) – The generator from which to derive a new child generator.

Returns In numpy <=1.16 a `RandomState`, in 1.17+ a `Generator`. In both cases a derived child generator.

Return type numpy.random.Generator or numpy.random.RandomState

`imgaug.imgaug.`**`derive_random_states`**`(`*random_state*, *n=1*`)`

Deprecated. Use `imgaug.random.derive_generators_` instead.

Derive child numpy random generators from another one.

Parameters

- **random_state** (*numpy.random.Generator or numpy.random.RandomState*) – The generator from which to derive new child generators.
- **n** (*int, optional*) – Number of child generators to derive.

Returns In numpy <=1.16 a `list` of `RandomState` s, in 1.17+ a `list` of `Generator` s. In both cases lists of derived child generators.

Return type list of numpy.random.Generator or list of numpy.random.RandomState

`imgaug.imgaug.`**`do_assert`**(*condition*, *message='Assertion failed.'*)

Assert that a `condition` holds or raise an `Exception` otherwise.

This was added because *assert* statements are removed in optimized code. It replaced *assert* statements throughout the library, but that was reverted again for readability and performance reasons.

Parameters

- **condition** (*bool*) – If `False`, an exception is raised.
- **message** (*str, optional*) – Error message.

`imgaug.imgaug.`**`draw_grid`**(*images*, *rows=None*, *cols=None*)

Combine multiple images into a single grid-like image.

Calling this function with four images of the same shape and `rows=2`, `cols=2` will combine the four images to a single image array of shape `(2*H, 2*W, C)`, where `H` is the height of any of the images (analogous `W`) and `C` is the number of channels of any image.

Calling this function with four images of the same shape and `rows=4`, `cols=1` is analogous to calling `numpy.vstack()` on the images.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; fully tested
* ``uint32``: yes; fully tested
* ``uint64``: yes; fully tested
* ``int8``: yes; fully tested
* ``int16``: yes; fully tested
* ``int32``: yes; fully tested
* ``int64``: yes; fully tested
* ``float16``: yes; fully tested
* ``float32``: yes; fully tested
* ``float64``: yes; fully tested
* ``float128``: yes; fully tested
* ``bool``: yes; fully tested
```

Parameters

- **images** (*(N,H,W,3) ndarray or iterable of (H,W,3) array*) – The input images to convert to a grid.
- **rows** (*None or int, optional*) – The number of rows to show in the grid. If `None`, it will be automatically derived.
- **cols** (*None or int, optional*) – The number of cols to show in the grid. If `None`, it will be automatically derived.

Returns Image of the generated grid.

Return type (H',W',3) ndarray

imgaug.imgaug.**draw_text**(*img*, *y*, *x*, *text*, *color=(0, 255, 0)*, *size=25*)

Draw text on an image.

This uses by default DejaVuSans as its font, which is included in this library.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: no
* ``uint32``: no
* ``uint64``: no
* ``int8``: no
* ``int16``: no
* ``int32``: no
* ``int64``: no
* ``float16``: no
* ``float32``: yes; not tested
* ``float64``: no
* ``float128``: no
* ``bool``: no

TODO check if other dtypes could be enabled
```

Parameters

- **img** (*(H,W,3) ndarray*) – The image array to draw text on. Expected to be of dtype `uint8` or `float32` (expected value range is `[0.0, 255.0]`).
- **y** (*int*) – x-coordinate of the top left corner of the text.
- **x** (*int*) – y- coordinate of the top left corner of the text.
- **text** (*str*) – The text to draw.
- **color** (*iterable of int, optional*) – Color of the text to draw. For RGB-images this is expected to be an RGB color.
- **size** (*int, optional*) – Font size of the text to draw.

Returns Input image with text drawn on it.

Return type (H,W,3) ndarray

imgaug.imgaug.**dummy_random_state**()

Deprecated. Use `imgaug.random.convert_seed_to_rng` instead.

Create a dummy random state using a seed of `1`.

Returns The new random state.

Return type *imgaug.random.RNG*

imgaug.imgaug.**flatten**(*nested_iterable*)

Flatten arbitrarily nested lists/tuples.

Code partially taken from https://stackoverflow.com/a/10824420.

Parameters **nested_iterable** – A `list` or `tuple` of arbitrarily nested values.

Yields *any* – All values in *nested_iterable*, flattened.

imgaug.imgaug.**forward_random_state**(*random_state*)

Deprecated. Use `imgaug.random.advance_generator_` instead.

Advance a numpy random generator's internal state.

Parameters **random_state** (*numpy.random.Generator or numpy.random.RandomState*) – Generator of which to advance the internal state.

imgaug.imgaug.**imresize_many_images**(*images*, *sizes=None*, *interpolation=None*)

Resize each image in a list or array to a specified size.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; tested
* ``uint32``: no (1)
* ``uint64``: no (2)
* ``int8``: yes; tested (3)
* ``int16``: yes; tested
* ``int32``: limited; tested (4)
* ``int64``: no (2)
* ``float16``: yes; tested (5)
* ``float32``: yes; tested
* ``float64``: yes; tested
* ``float128``: no (1)
* ``bool``: yes; tested (6)

- (1) rejected by ``cv2.imresize``
- (2) results too inaccurate
- (3) mapped internally to ``int16`` when interpolation!="nearest"
- (4) only supported for interpolation="nearest", other interpolations
      lead to cv2 error
- (5) mapped internally to ``float32``
- (6) mapped internally to ``uint8``
```

Parameters

- **images** (*(N,H,W,[C]) ndarray or list of (H,W,[C]) ndarray*) – Array of the images to resize. Usually recommended to be of dtype `uint8`.
- **sizes** (*float or iterable of int or iterable of float*) – The new size of the images, given either as a fraction (a single float) or as a `(height, width) tuple` of two integers or as a `(height fraction, width fraction) tuple` of two floats.
- **interpolation** (*None or str or int, optional*) – The interpolation to use during resize. If `int`, then expected to be one of:
  - `cv2.INTER_NEAREST` (nearest neighbour interpolation)
  - `cv2.INTER_LINEAR` (linear interpolation)
  - `cv2.INTER_AREA` (area interpolation)
  - `cv2.INTER_CUBIC` (cubic interpolation)

  If `str`, then expected to be one of:

  - `nearest` (identical to `cv2.INTER_NEAREST`)
  - `linear` (identical to `cv2.INTER_LINEAR`)
  - `area` (identical to `cv2.INTER_AREA`)

– `cubic` (identical to `cv2.INTER_CUBIC`)

If `None`, the interpolation will be chosen automatically. For size increases, `area` interpolation will be picked and for size decreases, `linear` interpolation will be picked.

Returns Array of the resized images.

Return type (N,H',W',[C]) ndarray

Examples

```
>>> import imgaug as ia
>>> images = np.zeros((2, 8, 16, 3), dtype=np.uint8)
>>> images_resized = ia.imresize_many_images(images, 2.0)
>>> images_resized.shape
(2, 16, 32, 3)
```

Convert two RGB images of height `8` and width `16` to images of height `2*8=16` and width `2*16=32`.

```
>>> images_resized = ia.imresize_many_images(images, (2.0, 4.0))
>>> images_resized.shape
(2, 16, 64, 3)
```

Convert two RGB images of height `8` and width `16` to images of height `2*8=16` and width `4*16=64`.

```
>>> images_resized = ia.imresize_many_images(images, (16, 32))
>>> images_resized.shape
(2, 16, 32, 3)
```

Converts two RGB images of height `8` and width `16` to images of height `16` and width `32`.

`imgaug.imgaug.`**`imresize_single_image`**(*image*, *sizes*, *interpolation=None*)

Resize a single image.

dtype support:

```
See :func:`imgaug.imgaug.imresize_many_images`.
```

Parameters

- **image** (*(H,W,C) ndarray or (H,W) ndarray*) – Array of the image to resize. Usually recommended to be of dtype `uint8`.
- **sizes** (*float or iterable of int or iterable of float*) – See *imgaug.imgaug.imresize_many_images()*.
- **interpolation** (*None or str or int, optional*) – See *imgaug.imgaug.imresize_many_images()*.

Returns The resized image.

Return type (H',W',C) ndarray or (H',W') ndarray

`imgaug.imgaug.`**`imshow`**(*image*, *backend='matplotlib'*)

Show an image in a window.

dtype support:

```
* ``uint8``: yes; not tested
* ``uint16``: ?
* ``uint32``: ?
* ``uint64``: ?
* ``int8``: ?
* ``int16``: ?
* ``int32``: ?
* ``int64``: ?
* ``float16``: ?
* ``float32``: ?
* ``float64``: ?
* ``float128``: ?
* ``bool``: ?
```

Parameters

- **image** (*(H,W,3) ndarray*) – Image to show.
- **backend** (*{'matplotlib', 'cv2'}, optional*) – Library to use to show the image. May be either matplotlib or OpenCV ('cv2'). OpenCV tends to be faster, but apparently causes more technical issues.

imgaug.imgaug.**is_callable**(*val*)
Check whether a variable is a callable, e.g. a function.

Parameters **val** – The variable to check.

Returns `True` if the variable is a callable. Otherwise `False`.

Return type bool

imgaug.imgaug.**is_float_array**(*val*)
Check whether a variable is a numpy float array.

Parameters **val** – The variable to check.

Returns `True` if the variable is a numpy float array. Otherwise `False`.

Return type bool

imgaug.imgaug.**is_generator**(*val*)
Check whether a variable is a generator.

Parameters **val** – The variable to check.

Returns `True` is the variable is a generator. Otherwise `False`.

Return type bool

imgaug.imgaug.**is_integer_array**(*val*)
Check whether a variable is a numpy integer array.

Parameters **val** – The variable to check.

Returns `True` if the variable is a numpy integer array. Otherwise `False`.

Return type bool

imgaug.imgaug.**is_iterable**(*val*)
Checks whether a variable is iterable.

Parameters **val** – The variable to check.

Returns `True` if the variable is an iterable. Otherwise `False`.

Return type bool

imgaug.imgaug.**is_np_array**(*val*)
Check whether a variable is a numpy array.

Parameters **val** – The variable to check.

Returns `True` if the variable is a numpy array. Otherwise `False`.

Return type bool

imgaug.imgaug.**is_np_scalar**(*val*)
Check whether a variable is a numpy scalar.

Parameters **val** – The variable to check.

Returns `True` if the variable is a numpy scalar. Otherwise `False`.

Return type bool

imgaug.imgaug.**is_single_bool**(*val*)
Check whether a variable is a `bool`.

Parameters **val** – The variable to check.

Returns `True` if the variable is a `bool`. Otherwise `False`.

Return type bool

imgaug.imgaug.**is_single_float**(*val*)
Check whether a variable is a `float`.

Parameters **val** – The variable to check.

Returns `True` if the variable is a `float`. Otherwise `False`.

Return type bool

imgaug.imgaug.**is_single_integer**(*val*)
Check whether a variable is an `int`.

Parameters **val** – The variable to check.

Returns `True` if the variable is an `int`. Otherwise `False`.

Return type bool

imgaug.imgaug.**is_single_number**(*val*)
Check whether a variable is a `number`, i.e. an `int` or `float`.

Parameters **val** – The variable to check.

Returns `True` if the variable is a `number`. Otherwise `False`.

Return type bool

imgaug.imgaug.**is_string**(*val*)
Check whether a variable is a string.

Parameters **val** – The variable to check.

Returns `True` if the variable is a string. Otherwise `False`.

Return type bool

imgaug.imgaug.**max_pool**(*arr*, *block_size*, *pad_mode='edge'*, *pad_cval=0*, *preserve_dtype=True*, *cval=None*)
Resize an array using max-pooling.

Defaults to `pad_mode="edge"` to ensure that padded values do not affect the maximum, even if the dtype was something else than `uint8`.

dtype support:

```
See :func:`imgaug.imgaug.pool`.
```

Parameters

- **arr** (*(H,W) ndarray or (H,W,C) ndarray*) – Image-like array to pool. See *imgaug.imgaug.pool()* for details.
- **block_size** (*int or tuple of int or tuple of int*) – Size of each block of values to pool. See *imgaug.imgaug.pool()* for details.
- **pad_mode** (*str, optional*) – Padding mode to use if the array cannot be divided by *block_size* without remainder. See *imgaug.imgaug.pad()* for details.
- **pad_cval** (*number, optional*) – Padding value. See *imgaug.imgaug.pool()* for details.
- **preserve_dtype** (*bool, optional*) – Whether to preserve the input array dtype. See *imgaug.imgaug.pool()* for details.
- **cval** (*None or number, optional*) – Deprecated. Old name for *pad_cval*.

Returns Array after max-pooling.

Return type (H',W') ndarray or (H',W',C') ndarray

`imgaug.imgaug.`**`median_pool`**(*arr*, *block_size*, *pad_mode='reflect'*, *pad_cval=128*, *preserve_dtype=True*)

Resize an array using median-pooling.

Defaults to `pad_mode="reflect"` to ensure that padded values do not affect the average.

dtype support:

```
See :func:`imgaug.imgaug.pool`.
```

Parameters

- **arr** (*(H,W) ndarray or (H,W,C) ndarray*) – Image-like array to pool. See *imgaug.imgaug.pool()* for details.
- **block_size** (*int or tuple of int or tuple of int*) – Size of each block of values to pool. See *imgaug.imgaug.pool()* for details.
- **pad_mode** (*str, optional*) – Padding mode to use if the array cannot be divided by *block_size* without remainder. See *imgaug.imgaug.pad()* for details.
- **pad_cval** (*number, optional*) – Padding value. See *imgaug.imgaug.pool()* for details.
- **preserve_dtype** (*bool, optional*) – Whether to preserve the input array dtype. See *imgaug.imgaug.pool()* for details.

Returns Array after min-pooling.

Return type (H',W') ndarray or (H',W',C') ndarray

`imgaug.imgaug.`**`min_pool`**(*arr*, *block_size*, *pad_mode='edge'*, *pad_cval=255*, *preserve_dtype=True*)

Resize an array using min-pooling.

Defaults to `pad_mode="edge"` to ensure that padded values do not affect the minimum, even if the dtype was something else than `uint8`.

dtype support:

```
See :func:`imgaug.imgaug.pool`.
```

Parameters

- **arr** (*(H,W) ndarray or (H,W,C) ndarray*) – Image-like array to pool. See *imgaug.imgaug.pool()* for details.
- **block_size** (*int or tuple of int or tuple of int*) – Size of each block of values to pool. See *imgaug.imgaug.pool()* for details.
- **pad_mode** (*str, optional*) – Padding mode to use if the array cannot be divided by *block_size* without remainder. See *imgaug.imgaug.pad()* for details.
- **pad_cval** (*number, optional*) – Padding value. See *imgaug.imgaug.pool()* for details.
- **preserve_dtype** (*bool, optional*) – Whether to preserve the input array dtype. See *imgaug.imgaug.pool()* for details.

Returns Array after min-pooling.

Return type (H',W') ndarray or (H',W',C') ndarray

imgaug.imgaug.**new_random_state**(*seed=None*, *fully_random=False*)

Deprecated. Use `imgaug.random.convert_seed_to_rng` instead.

Create a new numpy random number generator.

Parameters

- **seed** (*None or int, optional*) – The seed value to use. If `None` and *fully_random* is `False`, the seed will be derived from the global RNG. If *fully_random* is `True`, the seed will be provided by the OS.
- **fully_random** (*bool, optional*) – Whether the seed will be provided by the OS.

Returns In numpy <=1.16 a `RandomState`, in 1.17+ a `Generator`. Both are initialized with the provided seed.

Return type numpy.random.Generator or numpy.random.RandomState

imgaug.imgaug.**normalize_random_state**(*random_state*)

Deprecated. Use `imgaug.random.normalize_generator` instead.

Normalize various inputs to a numpy random generator.

Parameters **random_state** (*None or int or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.bit_generator.SeedSequence or numpy.random.RandomState*) – See *imgaug.random.normalize_generator()*.

Returns In numpy <=1.16 a `RandomState`, in 1.17+ a `Generator` (even if the input was a `RandomState`).

Return type numpy.random.Generator or numpy.random.RandomState

imgaug.imgaug.**pad**(*arr*, *top=0*, *right=0*, *bottom=0*, *left=0*, *mode='constant'*, *cval=0*)

Pad an image-like array on its top/right/bottom/left side.

This function is a wrapper around `numpy.pad()`.

dtype support:

```
* ``uint8``: yes; fully tested (1)
* ``uint16``: yes; fully tested (1)
* ``uint32``: yes; fully tested (2) (3)
* ``uint64``: yes; fully tested (2) (3)
* ``int8``: yes; fully tested (1)
* ``int16``: yes; fully tested (1)
* ``int32``: yes; fully tested (1)
* ``int64``: yes; fully tested (2) (3)
* ``float16``: yes; fully tested (2) (3)
* ``float32``: yes; fully tested (1)
* ``float64``: yes; fully tested (1)
* ``float128``: yes; fully tested (2) (3)
* ``bool``: yes; tested (2) (3)

- (1) Uses ``cv2`` if `mode` is one of: ``"constant"``, ``"edge"``,
      ``"reflect"``, ``"symmetric"``. Otherwise uses ``numpy``.
- (2) Uses ``numpy``.
- (3) Rejected by ``cv2``.
```

Parameters

- **arr** (*(H,W) ndarray or (H,W,C) ndarray*) – Image-like array to pad.
- **top** (*int, optional*) – Amount of pixels to add to the top side of the image. Must be `0` or greater.
- **right** (*int, optional*) – Amount of pixels to add to the right side of the image. Must be `0` or greater.
- **bottom** (*int, optional*) – Amount of pixels to add to the bottom side of the image. Must be `0` or greater.
- **left** (*int, optional*) – Amount of pixels to add to the left side of the image. Must be `0` or greater.
- **mode** (*str, optional*) – Padding mode to use. See `numpy.pad()` for details. In case of mode `constant`, the parameter *cval* will be used as the `constant_values` parameter to `numpy.pad()`. In case of mode `linear_ramp`, the parameter *cval* will be used as the `end_values` parameter to `numpy.pad()`.
- **cval** (*number, optional*) – Value to use for padding if *mode* is `constant`. See `numpy.pad()` for details. The cval is expected to match the input array's dtype and value range.

Returns Padded array with height `H'=H+top+bottom` and width `W'=W+left+right`.

Return type (H',W') ndarray or (H',W',C) ndarray

imgaug.imgaug.**pad_to_aspect_ratio**(*arr*, *aspect_ratio*, *mode='constant'*, *cval=0*, *return_pad_amounts=False*)

Pad an image array on its sides so that it matches a target aspect ratio.

See *imgaug.imgaug.compute_paddings_for_aspect_ratio()* for an explanation of how the required padding amounts are distributed per image axis.

dtype support:

```
See :func:`imgaug.imgaug.pad`.
```

Parameters

- **arr** (*(H,W) ndarray or (H,W,C) ndarray*) – Image-like array to pad.
- **aspect_ratio** (*float*) – Target aspect ratio, given as width/height. E.g. `2.0` denotes the image having twice as much width as height.
- **mode** (*str, optional*) – Padding mode to use. See *imgaug.imgaug.pad()* for details.
- **cval** (*number, optional*) – Value to use for padding if *mode* is `constant`. See `numpy.pad()` for details.
- **return_pad_amounts** (*bool, optional*) – If `False`, then only the padded image will be returned. If `True`, a `tuple` with two entries will be returned, where the first entry is the padded image and the second entry are the amounts by which each image side was padded. These amounts are again a `tuple` of the form `(top, right, bottom, left)`, with each value being an `int`.

Returns

- *(H',W') ndarray or (H',W',C) ndarray* – Padded image as `(H',W')` or `(H',W',C)` ndarray, fulfilling the given *aspect_ratio*.
- *tuple of int* – Amounts by which the image was padded on each side, given as a `tuple (top, right, bottom, left)`. This `tuple` is only returned if *return_pad_amounts* was set to `True`.

imgaug.imgaug.**pad_to_multiples_of**(*arr*, *height_multiple*, *width_multiple*, *mode='constant'*, *cval=0*, *return_pad_amounts=False*)

Pad an image array until its side lengths are multiples of given values.

See *imgaug.imgaug.compute_paddings_for_aspect_ratio()* for an explanation of how the required padding amounts are distributed per image axis.

dtype support:

```
See :func:`imgaug.imgaug.pad`.
```

Parameters

- **arr** (*(H,W) ndarray or (H,W,C) ndarray*) – Image-like array to pad.
- **height_multiple** (*None or int*) – The desired multiple of the height. The computed padding amount will reflect a padding that increases the y axis size until it is a multiple of this value.
- **width_multiple** (*None or int*) – The desired multiple of the width. The computed padding amount will reflect a padding that increases the x axis size until it is a multiple of this value.
- **mode** (*str, optional*) – Padding mode to use. See *imgaug.imgaug.pad()* for details.
- **cval** (*number, optional*) – Value to use for padding if *mode* is `constant`. See `numpy.pad()` for details.
- **return_pad_amounts** (*bool, optional*) – If `False`, then only the padded image will be returned. If `True`, a `tuple` with two entries will be returned, where the first entry is the padded image and the second entry are the amounts by which each image side was padded. These amounts are again a `tuple` of the form `(top, right, bottom, left)`, with each value being an integer.

Returns

- *(H',W') ndarray or (H',W',C) ndarray* – Padded image as `(H',W')` or `(H',W',C)` ndarray.

- *tuple of int* – Amounts by which the image was padded on each side, given as a `tuple (top, right, bottom, left)`. This `tuple` is only returned if *return_pad_amounts* was set to `True`.

imgaug.imgaug.**pool**(*arr*, *block_size*, *func*, *pad_mode='constant'*, *pad_cval=0*, *preserve_dtype=True*, *cval=None*)

Resize an array by pooling values within blocks.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; tested
* ``uint32``: yes; tested (2)
* ``uint64``: no (1)
* ``int8``: yes; tested
* ``int16``: yes; tested
* ``int32``: yes; tested (2)
* ``int64``: no (1)
* ``float16``: yes; tested
* ``float32``: yes; tested
* ``float64``: yes; tested
* ``float128``: yes; tested (2)
* ``bool``: yes; tested

- (1) results too inaccurate (at least when using np.average as func)
- (2) Note that scikit-image documentation says that the wrapped
      pooling function converts inputs to ``float64``. Actual tests
      showed no indication of that happening (at least when using
      preserve_dtype=True).
```

Parameters

- **arr** (*(H,W) ndarray or (H,W,C) ndarray*) – Image-like array to pool. Ideally of datatype `float64`.
- **block_size** (*int or tuple of int*) –

  Spatial size of each group of values to pool, aka kernel size.

  - If a single `int`, then a symmetric block of that size along height and width will be used.
  - If a `tuple` of two values, it is assumed to be the block size along height and width of the image-like, with pooling happening per channel.
  - If a `tuple` of three values, it is assumed to be the block size along height, width and channels.
- **func** (*callable*) – Function to apply to a given block in order to convert it to a single number, e.g. `numpy.average()`, `numpy.min()`, `numpy.max()`.
- **pad_mode** (*str, optional*) – Padding mode to use if the array cannot be divided by *block_size* without remainder. See *imgaug.imgaug.pad()* for details.
- **pad_cval** (*number, optional*) – Value to use for padding if *mode* is `constant`. See `numpy.pad()` for details.
- **preserve_dtype** (*bool, optional*) – Whether to convert the array back to the input datatype if it is changed away from that in the pooling process.
- **cval** (*None or number, optional*) – Deprecated. Old name for *pad_cval*.

Returns Array after pooling.

Return type (H',W') ndarray or (H',W',C') ndarray

imgaug.imgaug.**quokka**(*size=None*, *extract=None*)
Return an image of a quokka as a numpy array.

Parameters

- **size** (*None or float or tuple of int, optional*) – Size of the output image. Input into *imgaug.imgaug.imresize_single_image()*. Usually expected to be a `tuple (H, W)`, where `H` is the desired height and `W` is the width. If `None`, then the image will not be resized.
- **extract** (*None or 'square' or tuple of number or imgaug.augmentables.bbs.BoundingBox or imgaug.augmentables.bbs.BoundingBoxesOnImage*) –

  Subarea of the quokka image to extract:

  - If `None`, then the whole image will be used.
  - If `str square`, then a squared area `(x: 0 to max 643, y: 0 to max 643)` will be extracted from the image.
  - If a `tuple`, then expected to contain four `number` s denoting `(x1, y1, x2, y2)`.
  - If a *imgaug.augmentables.bbs.BoundingBox*, then that bounding box's area will be extracted from the image.
  - If a *imgaug.augmentables.bbs.BoundingBoxesOnImage*, then expected to contain exactly one bounding box and a shape matching the full image dimensions (i.e. `(643, 960, *)`). Then the one bounding box will be used similar to `BoundingBox` above.

Returns The image array of dtype `uint8`.

Return type (H,W,3) ndarray

imgaug.imgaug.**quokka_bounding_boxes**(*size=None*, *extract=None*)
Return example bounding boxes on the standard example quokke image.

Currently only a single bounding box is returned that covers the quokka.

Parameters

- **size** (*None or float or tuple of int or tuple of float, optional*) – Size of the output image on which the BBs are placed. If `None`, then the BBs are not projected to any new size (positions on the original image are used). `float` s lead to relative size changes, `int` s to absolute sizes in pixels.
- **extract** (*None or 'square' or tuple of number or imgaug.augmentables.bbs.BoundingBox or imgaug.augmentables.bbs.BoundingBoxesOnImage*) – Subarea to extract from the image. See *imgaug.imgaug.quokka()*.

Returns Example BBs on the quokka image.

Return type *imgaug.augmentables.bbs.BoundingBoxesOnImage*

imgaug.imgaug.**quokka_heatmap**(*size=None*, *extract=None*)
Return a heatmap (here: depth map) for the standard example quokka image.

Parameters

- **size** (*None or float or tuple of int, optional*) – See *imgaug.imgaug.quokka()*.

- **extract** (*None or 'square' or tuple of number or imgaug.augmentables.bbs.BoundingBox or imgaug.augmentables.bbs.BoundingBoxesOnImage*) – See *imgaug.imgaug.quokka()*.

Returns Depth map as an heatmap object. Values close to `0.0` denote objects that are close to the camera. Values close to `1.0` denote objects that are furthest away (among all shown objects).

Return type *imgaug.augmentables.heatmaps.HeatmapsOnImage*

imgaug.imgaug.**quokka_keypoints**(*size=None*, *extract=None*)

Return example keypoints on the standard example quokke image.

The keypoints cover the eyes, ears, nose and paws.

Parameters

- **size** (*None or float or tuple of int or tuple of float, optional*) – Size of the output image on which the keypoints are placed. If `None`, then the keypoints are not projected to any new size (positions on the original image are used). `float` s lead to relative size changes, `int` s to absolute sizes in pixels.
- **extract** (*None or 'square' or tuple of number or imgaug.augmentables.bbs.BoundingBox or imgaug.augmentables.bbs.BoundingBoxesOnImage*) – Subarea to extract from the image. See *imgaug.imgaug.quokka()*.

Returns Example keypoints on the quokka image.

Return type *imgaug.augmentables.kps.KeypointsOnImage*

imgaug.imgaug.**quokka_polygons**(*size=None*, *extract=None*)

Returns example polygons on the standard example quokke image.

The result contains one polygon, covering the quokka's outline.

Parameters

- **size** (*None or float or tuple of int or tuple of float, optional*) – Size of the output image on which the polygons are placed. If `None`, then the polygons are not projected to any new size (positions on the original image are used). `float` s lead to relative size changes, `int` s to absolute sizes in pixels.
- **extract** (*None or 'square' or tuple of number or imgaug.augmentables.bbs.BoundingBox or imgaug.augmentables.bbs.BoundingBoxesOnImage*) – Subarea to extract from the image. See *imgaug.imgaug.quokka()*.

Returns Example polygons on the quokka image.

Return type *imgaug.augmentables.polys.PolygonsOnImage*

imgaug.imgaug.**quokka_segmentation_map**(*size=None*, *extract=None*)

Return a segmentation map for the standard example quokka image.

Parameters

- **size** (*None or float or tuple of int, optional*) – See *imgaug.imgaug.quokka()*.
- **extract** (*None or 'square' or tuple of number or imgaug.augmentables.bbs.BoundingBox or imgaug.augmentables.bbs.BoundingBoxesOnImage*) – See *imgaug.imgaug.quokka()*.

Returns Segmentation map object.

Return type *imgaug.augmentables.segmaps.SegmentationMapsOnImage*

imgaug.imgaug.**quokka_square**(*size=None*)
Return an (square) image of a quokka as a numpy array.

Parameters **size** (*None or float or tuple of int, optional*) – Size of the output image. Input into *imgaug.imgaug.imresize_single_image()*. Usually expected to be a `tuple (H, W)`, where `H` is the desired height and `W` is the width. If `None`, then the image will not be resized.

Returns The image array of dtype `uint8`.

Return type (H,W,3) ndarray

imgaug.imgaug.**seed**(*entropy=None*, *seedval=None*)
Set the seed of imgaug's global RNG.

The global RNG controls most of the "randomness" in imgaug.

The global RNG is the default one used by all augmenters. Under special circumstances (e.g. when an augmenter is switched to deterministic mode), the global RNG is replaced with a local one. The state of that replacement may be dependent on the global RNG's state at the time of creating the child RNG.

Note: This function is not yet marked as deprecated, but might be in the future. The preferred way to seed *imgaug* is via *imgaug.random.seed()*.

Parameters

- **entropy** (*int*) – The seed value to use.
- **seedval** (*None or int, optional*) – Deprecated.

imgaug.imgaug.**show_grid**(*images*, *rows=None*, *cols=None*)
Combine multiple images into a single image and plot the result.

This will show a window of the results of *imgaug.imgaug.draw_grid()*.

dtype support:

```
minimum of (
    :func:`imgaug.imgaug.draw_grid`,
    :func:`imgaug.imgaug.imshow`
)
```

Parameters

- **images** (*(N,H,W,3) ndarray or iterable of (H,W,3) array*) – See *imgaug.imgaug.draw_grid()*.
- **rows** (*None or int, optional*) – See *imgaug.imgaug.draw_grid()*.
- **cols** (*None or int, optional*) – See *imgaug.imgaug.draw_grid()*.

imgaug.imgaug.**warn**(*msg*, *category=<class 'UserWarning'>*, *stacklevel=2*)
Generate a a warning with stacktrace.

Parameters

- **msg** (*str*) – The message of the warning.
- **category** (*class*) – The class of the warning to produce.
- **stacklevel** (*int, optional*) – How many steps above this function to "jump" in the stacktrace when displaying file and line number of the error message. Usually `2`.

imgaug.imgaug.**warn_deprecated**(*msg*, *stacklevel=2*)
Generate a non-silent deprecation warning with stacktrace.

The used warning is `imgaug.imgaug.DeprecationWarning`.

Parameters

- **msg** (*str*) – The message of the warning.
- **stacklevel** (*int, optional*) – How many steps above this function to "jump" in the stacktrace when displaying file and line number of the error message. Usually `2`

13.2 imgaug.parameters

class imgaug.parameters.**Absolute**(*other_param*)
Bases: *imgaug.parameters.StochasticParameter*

Convert the samples of another parameter to their absolute values.

Parameters **other_param** (*imgaug.parameters.StochasticParameter*) – Other parameter which's sampled values are to be modified.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.Absolute(iap.Uniform(-1.0, 1.0))
```

Convert a uniform distribution from `[-1.0, 1.0)` to `[0.0, 1.0]`.

Methods

copy(self)	Create a shallow copy of this parameter.
deepcopy(self)	Create a deep copy of this parameter.
draw_distribution_graph(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
draw_sample(self[, random_state])	Draws a single sample value from this parameter.
draw_samples(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**Add**(*other_param*, *val*, *elementwise=False*)
Bases: *imgaug.parameters.StochasticParameter*

Add to the samples of another stochastic parameter.

Parameters

- **other_param** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Samples of *val* will be added to samples of this parameter. Let `S` be the requested shape of samples, then the datatype behaviour is as follows:
  - If a single `number`, this `number` will be used as a constant value to fill an array of shape `S`.
  - If a `tuple` of two `number`s `(a, b)`, an array of shape `S` will be filled with uniformly sampled values from the continuous interval `[a, b)`.

- If a `list` of `number`, an array of shape `S` will be filled with randomly picked values from the `list`.
- If a *StochasticParameter*, that parameter will be queried once per call to generate an array of shape `S`.

"per call" denotes a call of `Add.draw_sample()` or `Add.draw_samples()`.

- **val** (*number or tuple of two number or list of number or imgaug.parameters.StochasticParameter*) – Value to add to the samples of *other_param*. Datatype behaviour is analogous to *other_param*, though if `elementwise=False` (the default), only a single sample will be generated per call instead of `S`.
- **elementwise** (*bool, optional*) – Controls the sampling behaviour of *val*. If set to `False`, a single samples will be requested from *val* and used as the constant multiplier. If set to `True`, samples of shape `S` will be requested from *val* and added elementwise with the samples of *other_param*.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.Add(Uniform(0.0, 1.0), 1.0)
```

Convert a uniform distribution from `[0.0, 1.0)` to `[1.0, 2.0)`.

Methods

`copy`(self)	Create a shallow copy of this parameter.
`deepcopy`(self)	Create a deep copy of this parameter.
`draw_distribution_graph`(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
`draw_sample`(self[, random_state])	Draws a single sample value from this parameter.
`draw_samples`(self, size[, random_state])	Draw one or more samples from the parameter.

class `imgaug.parameters.`**Beta**(*alpha*, *beta*, *epsilon=0.0001*)

Bases: *imgaug.parameters.StochasticParameter*

Parameter that resembles a (continuous) beta distribution.

Parameters

- **alpha** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – alpha parameter of the beta distribution. Expected value range is `(0, inf)`. Values below `0` are automatically clipped to `0+epsilon`.
  - If a single `number`, this `number` will be used as a constant value.
  - If a `tuple` of two `number` s `(a, b)`, the value will be sampled from the continuous interval `[a, b)` once per call.
  - If a `list` of `number`, a random value will be picked from the `list` once per call.
  - If a *StochasticParameter*, that parameter will be queried once per call.

  "per call" denotes a call of `Beta.draw_sample()` or `Beta.draw_samples()`.
- **beta** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Beta parameter of the beta distribution. Analogous

to *alpha*.

- **epsilon** (*number*) – Clipping parameter. If *alpha* or *beta* end up <=0, they are clipped to 0+epsilon.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.Beta(0.4, 0.6)
```

Create a beta distribution with alpha=0.4 and beta=0.6.

Methods

copy(self)	Create a shallow copy of this parameter.
deepcopy(self)	Create a deep copy of this parameter.
draw_distribution_graph(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
draw_sample(self[, random_state])	Draws a single sample value from this parameter.
draw_samples(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**Binomial**(*p*)

Bases: *imgaug.parameters.StochasticParameter*

Binomial distribution.

Parameters p (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Probability of the binomial distribution. Expected to be in the interval [0.0, 1.0].

- If a single number, this number will be used as a constant value.
- If a tuple of two number s (a, b), the value will be sampled from the continuous interval [a, b) once per call.
- If a list of number, a random value will be picked from the list once per call.
- If a *StochasticParameter*, that parameter will be queried once per call.

"per call" denotes a call of Binomial.draw_sample() or Binomial.draw_samples().

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.Binomial(Uniform(0.01, 0.2))
```

Create a binomial distribution that uses a varying probability between 0.01 and 0.2, randomly and uniformly estimated once per sampling call.

Methods

copy(self)	Create a shallow copy of this parameter.
deepcopy(self)	Create a deep copy of this parameter.
draw_distribution_graph(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
draw_sample(self[, random_state])	Draws a single sample value from this parameter.
draw_samples(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**ChiSquare**(*df*)

Bases: *imgaug.parameters.StochasticParameter*

Parameter that resembles a (continuous) chi-square distribution.

This is a wrapper around numpy's `numpy.random.chisquare()`.

Parameters **df** (*int or tuple of two int or list of int or imgaug.parameters.StochasticParameter*) –

Degrees of freedom. Expected value range is `[1, inf)`.

- If a single `int`, this `int` will be used as a constant value.
- If a `tuple` of two `int` s `(a, b)`, the value will be sampled from the discrete interval `[a..b]` once per call.
- If a `list` of `int`, a random value will be picked from the `list` once per call.
- If a *StochasticParameter*, that parameter will be queried once per call.

"per call" denotes a call of `ChiSquare.draw_sample()` or `ChiSquare.draw_samples()`.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.ChiSquare(df=2)
```

Create a chi-square distribution with two degrees of freedom.

Methods

copy(self)	Create a shallow copy of this parameter.
deepcopy(self)	Create a deep copy of this parameter.
draw_distribution_graph(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
draw_sample(self[, random_state])	Draws a single sample value from this parameter.
draw_samples(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**Choice**(*a*, *replace=True*, *p=None*)

Bases: *imgaug.parameters.StochasticParameter*

Parameter that samples value from a list of allowed values.

Parameters

- **a** (*iterable*) – List of allowed values. Usually expected to be `int` s, `float` s or `str` s. May also contain `StochasticParameter` s. Each `StochasticParameter` that is randomly picked will automatically be replaced by a sample of itself (or by `N` samples if the

parameter was picked `N` times).

- **replace** (*bool, optional*) – Whether to perform sampling with or without replacing.
- **p** (*None or iterable of number, optional*) – Probabilities of each element in *a*. Must have the same length as *a* (if provided).

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.Choice([5, 17, 25], p=[0.25, 0.5, 0.25])
>>> sample = param.draw_sample()
>>> assert sample in [5, 17, 25]
```

Create and sample from a parameter, which will produce with `50%` probability the sample `17` and in the other `50%` of all cases the sample `5` or `25`..

Methods

`copy`(self)	Create a shallow copy of this parameter.
`deepcopy`(self)	Create a deep copy of this parameter.
`draw_distribution_graph`(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
`draw_sample`(self[, random_state])	Draws a single sample value from this parameter.
`draw_samples`(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**Clip**(*other_param*, *minval=None*, *maxval=None*)

Bases: *imgaug.parameters.StochasticParameter*

Clip another parameter to a defined value range.

Parameters

- **other_param** (*imgaug.parameters.StochasticParameter*) – The other parameter, which's values are to be clipped.
- **minval** (*None or number, optional*) – The minimum value to use. If `None`, no minimum will be used.
- **maxval** (*None or number, optional*) – The maximum value to use. If `None`, no maximum will be used.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.Clip(Normal(0, 1.0), minval=-2.0, maxval=2.0)
```

Create a standard gaussian distribution, which's values never go below `-2.0` or above `2.0`. Note that this will lead to small "bumps" of higher probability at `-2.0` and `2.0`, as values below/above these will be clipped to them. For smoother limitations on gaussian distributions, see *TruncatedNormal*.

Methods

copy(self)	Create a shallow copy of this parameter.
deepcopy(self)	Create a deep copy of this parameter.
draw_distribution_graph(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
draw_sample(self[, random_state])	Draws a single sample value from this parameter.
draw_samples(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**Deterministic**(*value*)

Bases: *imgaug.parameters.StochasticParameter*

Parameter that is a constant value.

If N values are sampled from this parameter, it will return N times V, where V is the constant value.

Parameters **value** (*number or str or imgaug.parameters.StochasticParameter*) – A constant value to use. A string may be provided to generate arrays of strings. If this is a StochasticParameter, a single value will be sampled from it exactly once and then used as the constant value.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.Deterministic(10)
>>> param.draw_sample()
10
```

Will always sample the value 10.

Methods

copy(self)	Create a shallow copy of this parameter.
deepcopy(self)	Create a deep copy of this parameter.
draw_distribution_graph(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
draw_sample(self[, random_state])	Draws a single sample value from this parameter.
draw_samples(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**DiscreteUniform**(*a*, *b*)

Bases: *imgaug.parameters.StochasticParameter*

Uniform distribution over the discrete interval [a..b].

Parameters

- **a** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter*) – Lower bound of the interval. If a>b, *a* and *b* will automatically be flipped. If a==b, all generated values will be identical to *a*.
  - If a single int, this int will be used as a constant value.
  - If a tuple of two int s (a, b), the value will be sampled from the discrete interval [a..b] once per call.
  - If a list of int, a random value will be picked from the list once per call.
  - If a *StochasticParameter*, that parameter will be queried once per call.

"per call" denotes a call of `DiscreteUniform.draw_sample()` or `DiscreteUniform.draw_samples()`.

- **b** (*int or imgaug.parameters.StochasticParameter*) – Upper bound of the interval. Analogous to *a*.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.DiscreteUniform(10, Choice([20, 30, 40]))
>>> sample = param.draw_sample()
>>> assert 10 <= sample <= 40
```

Create a discrete uniform distribution which's interval differs between calls and can be `[10..20]`, `[10..30]` or `[10..40]`.

Methods

copy(self)	Create a shallow copy of this parameter.
deepcopy(self)	Create a deep copy of this parameter.
draw_distribution_graph(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
draw_sample(self[, random_state])	Draws a single sample value from this parameter.
draw_samples(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**Discretize**(*other_param*)

Bases: *imgaug.parameters.StochasticParameter*

Convert a continuous distribution to a discrete one.

This will round the values and then cast them to integers. Values sampled from already discrete distributions are not changed.

Parameters **other_param** (*imgaug.parameters.StochasticParameter*) – The other parameter, which's values are to be discretized.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.Discretize(iap.Normal(0, 1.0))
```

Create a discrete standard gaussian distribution.

Methods

copy(self)	Create a shallow copy of this parameter.
deepcopy(self)	Create a deep copy of this parameter.
draw_distribution_graph(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
draw_sample(self[, random_state])	Draws a single sample value from this parameter.

Continued on next page

Table 14 – continued from previous page

draw_samples(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**Divide**(*other_param*, *val*, *elementwise=False*)

Bases: *imgaug.parameters.StochasticParameter*

Divide the samples of another stochastic parameter.

This parameter will automatically prevent division by zero (uses 1.0) as the denominator in these cases.

Parameters

- **other_param** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Other parameter which's sampled values are to be divided by *val*. Let `S` be the requested shape of samples, then the datatype behaviour is as follows:
  - If a single `number`, this `number` will be used as a constant value to fill an array of shape `S`.
  - If a `tuple` of two `number` s `(a, b)`, an array of shape `S` will be filled with uniformly sampled values from the continuous interval `[a, b)`.
  - If a `list` of `number`, an array of shape `S` will be filled with randomly picked values from the `list`.
  - If a *StochasticParameter*, that parameter will be queried once per call to generate an array of shape `S`.

  "per call" denotes a call of `Divide.draw_sample()` or `Divide.draw_samples()`.
- **val** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Denominator to use. Datatype behaviour is analogous to *other_param*, though if `elementwise=False` (the default), only a single sample will be generated per call instead of `S`.
- **elementwise** (*bool, optional*) – Controls the sampling behaviour of *val*. If set to `False`, a single samples will be requested from *val* and used as the constant denominator. If set to `True`, samples of shape `S` will be requested from *val* and used to divide the samples of *other_param* elementwise.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.Divide(iap.Uniform(0.0, 1.0), 2)
```

Convert a uniform distribution `[0.0, 1.0)` to `[0, 0.5)`.

Methods

copy(self)	Create a shallow copy of this parameter.
deepcopy(self)	Create a deep copy of this parameter.
draw_distribution_graph(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
draw_sample(self[, random_state])	Draws a single sample value from this parameter.

Continued on next page

Table 15 – continued from previous page

draw_samples(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**ForceSign**(*other_param*, *positive*, *mode='invert'*, *reroll_count_max=2*)

Bases: *imgaug.parameters.StochasticParameter*

Convert a parameter's samples to either positive or negative values.

Parameters

- **other_param** (*imgaug.parameters.StochasticParameter*) – Other parameter which's sampled values are to be modified.
- **positive** (*bool*) – Whether to force all signs to be positive (`True`) or negative (`False`).
- **mode** (*{'invert', 'reroll'}, optional*) – Method to change the signs. Valid values are `invert` and `reroll`. `invert` means that wrong signs are simply flipped. `reroll` means that all samples with wrong signs are sampled again, optionally many times, until they randomly end up having the correct sign.
- **reroll_count_max** (*int, optional*) – If *mode* is set to `reroll`, this determines how often values may be rerolled before giving up and simply flipping the sign (as in `mode="invert"`). This shouldn't be set too high, as rerolling is expensive.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.ForceSign(iap.Poisson(1), positive=False)
```

Create a poisson distribution with `alpha=1` that is flipped towards negative values.

Methods

copy(self)	Create a shallow copy of this parameter.
deepcopy(self)	Create a deep copy of this parameter.
draw_distribution_graph(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
draw_sample(self[, random_state])	Draws a single sample value from this parameter.
draw_samples(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**FrequencyNoise**(*exponent=(-4, 4)*, *size_px_max=(4, 32)*, *upscale_method=['linear', 'nearest']*)

Bases: *imgaug.parameters.StochasticParameter*

Parameter to generate noise of varying frequencies.

This parameter expects to sample noise for 2d planes, i.e. for sizes `(H, W)` and will return a value in the range `[0.0, 1.0]` per spatial location in that plane.

The exponent controls the frequencies and therefore noise patterns. Small values (around `-4.0`) will result in large blobs. Large values (around `4.0`) will result in small, repetitive patterns.

The noise is sampled from low resolution planes and upscaled to the requested height and width. The size of the low resolution plane may be defined (high values can be slow) and the interpolation method for upscaling can be set.

Parameters

- **exponent** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Exponent to use when scaling in the frequency domain. Sane values are in the range `-4` (large blobs) to `4` (small patterns). To generate cloud-like structures, use roughly `-2`.
  - If a single `number`, this `number` will be used as a constant value.
  - If a `tuple` of two `number`s `(a, b)`, the value will be sampled from the continuous interval `[a, b)` once per call.
  - If a `list` of `number`, a random value will be picked from the `list` once per call.
  - If a *`StochasticParameter`*, that parameter will be queried once per call.
- **size_px_max** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) – Maximum height and width in pixels of the low resolution plane. Upon any sampling call, the requested shape will be downscaled until the height or width (whichever is larger) does not exceed this maximum value anymore. Then the noise will be sampled at that shape and later upscaled back to the requested shape.
  - If a single `int`, this `int` will be used as a constant value.
  - If a `tuple` of two `int`s `(a, b)`, the value will be sampled from the discrete interval `[a..b]` once per call.
  - If a `list` of `int`, a random value will be picked from the `list` once per call.
  - If a *`StochasticParameter`*, that parameter will be queried once per call.

  "per call" denotes a call of `FrequencyNoise.draw_sample()` or `FrequencyNoise.draw_samples()`.
- **upscale_method** (*imgaug.ALL or str or list of str or imgaug.parameters.StochasticParameter, optional*) – After generating the noise maps in low resolution environments, they have to be upscaled to the originally requested shape (i.e. usually the image size). This parameter controls the interpolation method to use. See also *`imgaug.imgaug.imresize_many_images()`* for a description of possible values.
  - If `imgaug.ALL`, then either `nearest` or `linear` or `area` or `cubic` is picked per iteration (all same probability).
  - If `str`, then that value will always be used as the method (must be `nearest` or `linear` or `area` or `cubic`).
  - If `list` of `str`, then a random value will be picked from that list per call.
  - If *`StochasticParameter`*, then a random value will be sampled from that parameter per call.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.FrequencyNoise(
>>>     exponent=-2,
>>>     size_px_max=(16, 32),
>>>     upscale_method="linear")
```

Create a parameter that produces noise with cloud-like patterns.

Methods

copy(self)	Create a shallow copy of this parameter.
deepcopy(self)	Create a deep copy of this parameter.
draw_distribution_graph(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
draw_sample(self[, random_state])	Draws a single sample value from this parameter.
draw_samples(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**FromLowerResolution**(*other_param*, *size_percent=None*, *size_px=None*, *method='nearest'*, *min_size=1*)

Bases: *imgaug.parameters.StochasticParameter*

Parameter to sample from other parameters at lower image resolutions.

This parameter is intended to be used with parameters that would usually sample one value per pixel (or one value per pixel and channel). Instead of sampling from the other parameter at full resolution, it samples at lower resolution, e.g. `0.5*H x 0.5*W` with `H` being the height and `W` being the width. After the low-resolution sampling this parameter then upscales the result to `HxW`.

This parameter is intended to produce coarse samples. E.g. combining this with *Binomial* can lead to large rectangular areas of `1` s and `0` s.

Parameters

- **other_param** (*imgaug.parameters.StochasticParameter*) – The other parameter which is to be sampled on a coarser image.
- **size_percent** (*None or number or iterable of number or imgaug.parameters.StochasticParameter, optional*) – Size of the 2d sampling plane in percent of the requested size. I.e. this is relative to the size provided in the call to `draw_samples(size)`. Lower values will result in smaller sampling planes, which are then upsampled to *size*. This means that lower values will result in larger rectangles. The size may be provided as a constant value or a tuple `(a, b)`, which will automatically be converted to the continuous uniform range `[a, b)` or a *StochasticParameter*, which will be queried per call to `FromLowerResolution.draw_sample()` and `FromLowerResolution.draw_samples()`.
- **size_px** (*None or number or iterable of numbers or imgaug.parameters.StochasticParameter, optional*) – Size of the 2d sampling plane in pixels. Lower values will result in smaller sampling planes, which are then upsampled to the input *size* of `draw_samples(size)`. This means that lower values will result in larger rectangles. The size may be provided as a constant value or a tuple `(a, b)`, which will automatically be converted to the discrete uniform range `[a..b]` or a *StochasticParameter*, which will be queried once per call to `FromLowerResolution.draw_sample()` and `FromLowerResolution.draw_samples()`.
- **method** (*str or int or imgaug.parameters.StochasticParameter, optional*) – Upsampling/interpolation method to use. This is used after the sampling is finished and the low resolution plane has to be upsampled to the requested *size* in `draw_samples(size, ...)`. The method may be the same as in *imgaug.imgaug.imresize_many_images()*. Usually `nearest` or `linear` are good choices. `nearest` will result in rectangles with sharp edges and `linear` in rectangles with blurry and round edges. The method may be provided as a *StochasticParameter*, which will be queried once per call to `FromLowerResolution.draw_sample()` and `FromLowerResolution.draw_samples()`.

- **min_size** (*int, optional*) – Minimum size in pixels of the low resolution sampling plane.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.FromLowerResolution(
>>>     Binomial(0.05),
>>>     size_px=(2, 16),
>>>     method=Choice(["nearest", "linear"]))
```

Samples from a binomial distribution with `p=0.05`. The sampling plane will always have a size HxWxC with H and W being independently sampled from `[2..16]` (i.e. it may range from `2x2xC` up to `16x16xC` max, but may also be e.g. `4x8xC`). The upsampling method will be `nearest` in `50%` of all cases and `linear` in the other 50 percent. The result will sometimes be rectangular patches of sharp `1` s surrounded by `0` s and sometimes blurry blobs of `1``s, surrounded by values ``<1.0`.

Methods

copy(self)	Create a shallow copy of this parameter.
deepcopy(self)	Create a deep copy of this parameter.
draw_distribution_graph(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
draw_sample(self[, random_state])	Draws a single sample value from this parameter.
draw_samples(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**IterativeNoiseAggregator**(*other_param, iterations=(1, 3), aggregation_method=['max', 'avg']*)

Bases: *imgaug.parameters.StochasticParameter*

Aggregate multiple iterations of samples from another parameter.

This is supposed to be used in conjunction with *SimplexNoise* or *FrequencyNoise*. If a shape `S` is requested, it will request `I` times `S` samples from the underlying parameter, where `I` is the number of iterations. The `I` arrays will be combined to a single array of shape `S` using an aggregation method, e.g. simple averaging.

Parameters

- **other_param** (*StochasticParameter*) – The other parameter from which to sample one or more times.
- **iterations** (*int or iterable of int or list of int or imgaug.parameters.StochasticParameter, optional*) –

  The number of iterations.

  – If a single `int`, this `int` will be used as a constant value.

  – If a `tuple` of two `int` s `(a, b)`, the value will be sampled from the discrete interval `[a..b]` once per call.

  – If a `list` of `int`, a random value will be picked from the `list` once per call.

  – If a *StochasticParameter*, that parameter will be queried once per call.

  "per call" denotes a call of `IterativeNoiseAggregator.draw_sample()` or `IterativeNoiseAggregator.draw_samples()`.

- **aggregation_method** (*imgaug.ALL or {'min', 'avg', 'max'} or list of str or imgaug.parameters.StochasticParameter, optional*) – The method to use to aggregate the samples of multiple iterations to a single output array. All methods combine several arrays of shape `S` each to a single array of shape `S` and hence work elementwise. Known methods are `min` (take the minimum over all iterations), `max` (take the maximum) and `avg` (take the average).
  - If an `str`, it must be one of the described methods and will be used for all calls..
  - If a `list` of `str`, it must contain one or more of the described methods and a random one will be samples once per call.
  - If `imgaug.ALL`, then equivalent to the `list ["min", "max", "avg"]`.
  - If *StochasticParameter*, a value will be sampled from that parameter once per call and must be one of the described methods..

  "per call" denotes a call of `IterativeNoiseAggregator.draw_sample()` or `IterativeNoiseAggregator.draw_samples()`.

Examples

```
>>> import imgaug.parameters as iap
>>> noise = iap.IterativeNoiseAggregator(
>>>     iap.SimplexNoise(),
>>>     iterations=(2, 5),
>>>     aggregation_method="max")
```

Create a parameter that – upon each call – generates `2` to `5` arrays of simplex noise with the same shape. Then it combines these noise maps to a single map using elementwise maximum.

Methods

`copy`(self)	Create a shallow copy of this parameter.
`deepcopy`(self)	Create a deep copy of this parameter.
`draw_distribution_graph`(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
`draw_sample`(self[, random_state])	Draws a single sample value from this parameter.
`draw_samples`(self, size[, random_state])	Draw one or more samples from the parameter.

class `imgaug.parameters.`**Laplace**(*loc*, *scale*)

Bases: *imgaug.parameters.StochasticParameter*

Parameter that resembles a (continuous) laplace distribution.

This is a wrapper around numpy's `numpy.random.laplace()`.

Parameters

- **loc** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – The position of the distribution peak, similar to the mean in normal distributions.
  - If a single `number`, this `number` will be used as a constant value.
  - If a `tuple` of two `number` s `(a, b)`, the value will be sampled from the continuous interval `[a, b)` once per call.

- If a `list` of `number`, a random value will be picked from the `list` once per call.
- If a *StochasticParameter*, that parameter will be queried once per call.

"per call" denotes a call of `Laplace.draw_sample()` or `Laplace.draw_samples()`.

- **scale** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – The exponential decay factor, similar to the standard deviation in gaussian distributions. If this parameter reaches `0`, the output array will be filled with *loc*. Datatype behaviour is the analogous to *loc*.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.Laplace(0, 1.0)
```

Create a laplace distribution, which's peak is at `0` and decay is `1.0`.

Methods

`copy`(self)	Create a shallow copy of this parameter.
`deepcopy`(self)	Create a deep copy of this parameter.
`draw_distribution_graph`(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
`draw_sample`(self[, random_state])	Draws a single sample value from this parameter.
`draw_samples`(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**Multiply**(*other_param*, *val*, *elementwise=False*)

Bases: *imgaug.parameters.StochasticParameter*

Multiply the samples of another stochastic parameter.

Parameters

- **other_param** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Other parameter which's sampled values are to be multiplied with *val*. Let `S` be the requested shape of samples, then the datatype behaviour is as follows:
  - If a single `number`, this `number` will be used as a constant value to fill an array of shape `S`.
  - If a `tuple` of two `number`s `(a, b)`, an array of shape `S` will be filled with uniformly sampled values from the continuous interval `[a, b)`.
  - If a `list` of `number`, an array of shape `S` will be filled with randomly picked values from the `list`.
  - If a *StochasticParameter*, that parameter will be queried once per call to generate an array of shape `S`.

  "per call" denotes a call of `Multiply.draw_sample()` or `Multiply.draw_samples()`.
- **val** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Multiplier to use. Datatype behaviour is

analogous to *other_param*, though if `elementwise=False` (the default), only a single sample will be generated per call instead of `S`.

- **elementwise** (*bool, optional*) – Controls the sampling behaviour of *val*. If set to `False`, a single samples will be requested from *val* and used as the constant multiplier. If set to `True`, samples of shape `S` will be requested from *val* and multiplied elementwise with the samples of *other_param*.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.Multiply(iap.Uniform(0.0, 1.0), -1)
```

Convert a uniform distribution from `[0.0, 1.0)` to `(-1.0, 0.0]`.

Methods

`copy`(self)	Create a shallow copy of this parameter.
`deepcopy`(self)	Create a deep copy of this parameter.
`draw_distribution_graph`(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
`draw_sample`(self[, random_state])	Draws a single sample value from this parameter.
`draw_samples`(self, size[, random_state])	Draw one or more samples from the parameter.

`imgaug.parameters.`**`Negative`**(*other_param*, *mode='invert'*, *reroll_count_max=2*)

Convert another parameter's results to negative values.

Parameters

- **other_param** (*imgaug.parameters.StochasticParameter*) – Other parameter which's sampled values are to be modified.
- **mode** (*{'invert', 'reroll'}, optional*) – How to change the signs. Valid values are `invert` and `reroll`. `invert` means that wrong signs are simply flipped. `reroll` means that all samples with wrong signs are sampled again, optionally many times, until they randomly end up having the correct sign.
- **reroll_count_max** (*int, optional*) – If *mode* is set to `reroll`, this determines how often values may be rerolled before giving up and simply flipping the sign (as in `mode="invert"`). This shouldn't be set too high, as rerolling is expensive.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.Negative(iap.Normal(0, 1), mode="reroll")
```

Create a gaussian distribution that has only negative values. If any positive value is sampled in the process, that sample is resampled up to two times to get a negative one. If it isn't negative after the second resampling step, the sign is simply flipped.

`class` `imgaug.parameters.`**`Normal`**(*loc*, *scale*)

Bases: *imgaug.parameters.StochasticParameter*

Parameter that resembles a normal/gaussian distribution.

Parameters

- **loc** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) –

  The mean of the normal distribution.

  – If a single `number`, this `number` will be used as a constant value.

  – If a `tuple` of two `number` s `(a, b)`, the value will be sampled from the continuous interval `[a, b)` once per call.

  – If a `list` of `number`, a random value will be picked from the `list` once per call.

  – If a *StochasticParameter*, that parameter will be queried once per call.

  "per call" denotes a call of `Laplace.draw_sample()` or `Laplace.draw_samples()`.

- **scale** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – The standard deviation of the normal distribution. If this parameter reaches `0`, the output array will be filled with *loc*. Datatype behaviour is the analogous to *loc*.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.Normal(Choice([-1.0, 1.0]), 1.0)
```

Create a gaussian distribution with a mean that differs by call. Samples values may sometimes follow `N(-1.0, 1.0)` and sometimes `N(1.0, 1.0)`.

Methods

`copy`(self)	Create a shallow copy of this parameter.
`deepcopy`(self)	Create a deep copy of this parameter.
`draw_distribution_graph`(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
`draw_sample`(self[, random_state])	Draws a single sample value from this parameter.
`draw_samples`(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**Poisson**(*lam*)

Bases: *imgaug.parameters.StochasticParameter*

Parameter that resembles a poisson distribution.

A poisson distribution with `lambda=0` has its highest probability at point `0` and decreases quickly from there. Poisson distributions are discrete and never negative.

Parameters **lam** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) –

Lambda parameter of the poisson distribution.

- If a single `number`, this `number` will be used as a constant value.
- If a `tuple` of two `number` s `(a, b)`, the value will be sampled from the continuous interval `[a, b)` once per call.

- If a `list` of `number`, a random value will be picked from the `list` once per call.
- If a *`StochasticParameter`*, that parameter will be queried once per call.

"per call" denotes a call of `Poisson.draw_sample()` or `Poisson.draw_samples()`.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.Poisson(1)
>>> sample = param.draw_sample()
>>> assert sample >= 0
```

Create a poisson distribution with `lambda=1` and sample a value from it.

Methods

`copy`(self)	Create a shallow copy of this parameter.
`deepcopy`(self)	Create a deep copy of this parameter.
`draw_distribution_graph`(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
`draw_sample`(self[, random_state])	Draws a single sample value from this parameter.
`draw_samples`(self, size[, random_state])	Draw one or more samples from the parameter.

`imgaug.parameters.`**`Positive`**(*other_param*, *mode='invert'*, *reroll_count_max=2*)

Convert another parameter's results to positive values.

Parameters

- **other_param** (*imgaug.parameters.StochasticParameter*) – Other parameter which's sampled values are to be modified.
- **mode** (*{'invert', 'reroll'}, optional*) – How to change the signs. Valid values are `invert` and `reroll`. `invert` means that wrong signs are simply flipped. `reroll` means that all samples with wrong signs are sampled again, optionally many times, until they randomly end up having the correct sign.
- **reroll_count_max** (*int, optional*) – If *mode* is set to `reroll`, this determines how often values may be rerolled before giving up and simply flipping the sign (as in `mode="invert"`). This shouldn't be set too high, as rerolling is expensive.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.Positive(iap.Normal(0, 1), mode="reroll")
```

Create a gaussian distribution that has only positive values. If any negative value is sampled in the process, that sample is resampled up to two times to get a positive one. If it isn't positive after the second resampling step, the sign is simply flipped.

`class` `imgaug.parameters.`**`Power`**(*other_param*, *val*, *elementwise=False*)

Bases: *`imgaug.parameters.StochasticParameter`*

Exponentiate the samples of another stochastic parameter.

Parameters

- **other_param** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Other parameter which's sampled values are to be exponentiated by *val*. Let `S` be the requested shape of samples, then the datatype behaviour is as follows:
  - If a single `number`, this `number` will be used as a constant value to fill an array of shape `S`.
  - If a `tuple` of two `number` s `(a, b)`, an array of shape `S` will be filled with uniformly sampled values from the continuous interval `[a, b)`.
  - If a `list` of `number`, an array of shape `S` will be filled with randomly picked values from the `list`.
  - If a *StochasticParameter*, that parameter will be queried once per call to generate an array of shape `S`.

  "per call" denotes a call of `Power.draw_sample()` or `Power.draw_samples()`.
- **val** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Value to use exponentiate the samples of *other_param*. Datatype behaviour is analogous to *other_param*, though if `elementwise=False` (the default), only a single sample will be generated per call instead of `S`.
- **elementwise** (*bool, optional*) – Controls the sampling behaviour of *val*. If set to `False`, a single samples will be requested from *val* and used as the constant multiplier. If set to `True`, samples of shape `S` will be requested from *val* and used to exponentiate elementwise the samples of *other_param*.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.Power(iap.Uniform(0.0, 1.0), 2)
```

Converts a uniform range `[0.0, 1.0)` to a distribution that is peaked towards 1.0.

Methods

`copy`(self)	Create a shallow copy of this parameter.
`deepcopy`(self)	Create a deep copy of this parameter.
`draw_distribution_graph`(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
`draw_sample`(self[, random_state])	Draws a single sample value from this parameter.
`draw_samples`(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**RandomSign**(*other_param*, *p_positive=0.5*)

Bases: *imgaug.parameters.StochasticParameter*

Convert a parameter's samples randomly to positive or negative values.

Parameters

- **other_param** (*imgaug.parameters.StochasticParameter*) – Other parameter which's sampled values are to be modified.

- **p_positive** (*number*) – Fraction of values that are supposed to be turned to positive values.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.RandomSign(iap.Poisson(1))
```

Create a poisson distribution with `alpha=1` that is mirrored/copied (not flipped) at the y-axis.

Methods

`copy`(self)	Create a shallow copy of this parameter.
`deepcopy`(self)	Create a deep copy of this parameter.
`draw_distribution_graph`(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
`draw_sample`(self[, random_state])	Draws a single sample value from this parameter.
`draw_samples`(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**Sigmoid**(*other_param*, *threshold=(-10, 10)*, *activated=True*, *mul=1*, *add=0*)

Bases: *imgaug.parameters.StochasticParameter*

Apply a sigmoid function to the outputs of another parameter.

This is intended to be used in combination with *SimplexNoise* or *FrequencyNoise*. It pushes the noise values away from `~0.5` and towards `0.0` or `1.0`, making the noise maps more binary.

Parameters

- **other_param** (*imgaug.parameters.StochasticParameter*) – The other parameter to which the sigmoid will be applied.
- **threshold** (*number or tuple of number or iterable of number or imgaug.parameters.StochasticParameter, optional*) – Sets the value of the sigmoid's saddle point, i.e. where values start to quickly shift from `0.0` to `1.0`.
  - If a single `number`, this `number` will be used as a constant value.
  - If a `tuple` of two `number`s `(a, b)`, the value will be sampled from the continuous interval `[a, b)` once per call.
  - If a `list` of `number`, a random value will be picked from the `list` once per call.
  - If a *StochasticParameter*, that parameter will be queried once per call.

  "per call" denotes a call of `Sigmoid.draw_sample()` or `Sigmoid.draw_samples()`.
- **activated** (*bool or number, optional*) – Defines whether the sigmoid is activated. If this is `False`, the results of *other_param* will not be altered. This may be set to a `float p` in value range''[0.0, 1.0]'', which will result in *activated* being `True` in `p` percent of all calls.
- **mul** (*number, optional*) – The results of *other_param* will be multiplied with this value before applying the sigmoid. For noise values (range `[0.0, 1.0]`) this should be set to about `20`.

- **add** (*number, optional*) – This value will be added to the results of *other_param* before applying the sigmoid. For noise values (range `[0.0, 1.0]`) this should be set to about `-10.0`, provided *mul* was set to `20`.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.Sigmoid(
>>>     iap.SimplexNoise(),
>>>     activated=0.5,
>>>     mul=20,
>>>     add=-10)
```

Applies a sigmoid to simplex noise in `50%` of all calls. The noise results are modified to match the sigmoid's expected value range. The sigmoid's outputs are in the range `[0.0, 1.0]`.

Methods

copy(self)	Create a shallow copy of this parameter.
create_for_noise(other_param[, threshold, ...])	Create a Sigmoid adjusted for noise parameters.
deepcopy(self)	Create a deep copy of this parameter.
draw_distribution_graph(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
draw_sample(self[, random_state])	Draws a single sample value from this parameter.
draw_samples(self, size[, random_state])	Draw one or more samples from the parameter.

static create_for_noise(*other_param*, *threshold=(-10, 10)*, *activated=True*)

Create a Sigmoid adjusted for noise parameters.

"noise" here denotes *SimplexNoise* and *FrequencyNoise*.

Parameters

- **other_param** (*imgaug.parameters.StochasticParameter*) – See `imgaug.parameters.Sigmoid.__init__()`.
- **threshold** (*number or tuple of number or iterable of number or imgaug.parameters.StochasticParameter, optional*) – See `imgaug.parameters.Sigmoid.__init__()`.
- **activated** (*bool or number, optional*) – See `imgaug.parameters.Sigmoid.__init__()`.

Returns A sigmoid adjusted to be used with noise.

Return type *Sigmoid*

class imgaug.parameters.**SimplexNoise**(*size_px_max=(2, 16), upscale_method=['linear', 'nearest']*)

Bases: *imgaug.parameters.StochasticParameter*

Parameter that generates simplex noise of varying resolutions.

This parameter expects to sample noise for 2d planes, i.e. for sizes `(H, W)` and will return a value in the range `[0.0, 1.0]` per spatial location in that plane.

The noise is sampled from low resolution planes and upscaled to the requested height and width. The size of the low resolution plane may be defined (large values can be slow) and the interpolation method for upscaling can be set.

Parameters

- **size_px_max** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) – Maximum height and width in pixels of the low resolution plane. Upon any sampling call, the requested shape will be downscaled until the height or width (whichever is larger) does not exceed this maximum value anymore. Then the noise will be sampled at that shape and later upscaled back to the requested shape.
  - If a single `int`, this `int` will be used as a constant value.
  - If a `tuple` of two `int` s `(a, b)`, the value will be sampled from the discrete interval `[a..b]` once per call.
  - If a `list` of `int`, a random value will be picked from the `list` once per call.
  - If a *`StochasticParameter`*, that parameter will be queried once per call.

  "per call" denotes a call of `SimplexNoise.draw_sample()` or `SimplexNoise.draw_samples()`.
- **upscale_method** (*str or int or list of str or list of int or imgaug.parameters.StochasticParameter, optional*) – After generating the noise maps in low resolution environments, they have to be upscaled to the originally requested shape (i.e. usually the image size). This parameter controls the interpolation method to use. See also *`imgaug.imgaug.imresize_many_images()`* for a description of possible values.
  - If `imgaug.ALL`, then either `nearest` or `linear` or `area` or `cubic` is picked per iteration (all same probability).
  - If `str`, then that value will always be used as the method (must be `nearest` or `linear` or `area` or `cubic`).
  - If `list` of `str`, then a random value will be picked from that list per call.
  - If *`StochasticParameter`*, then a random value will be sampled from that parameter per call.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.SimplexNoise(upscale_method="linear")
```

Create a parameter that produces smooth simplex noise of varying sizes.

```
>>> param = iap.SimplexNoise(
>>>     size_px_max=(8, 16),
>>>     upscale_method="nearest")
```

Create a parameter that produces rectangular simplex noise of rather high detail.

Methods

`copy`(self)	Create a shallow copy of this parameter.

Continued on next page

Table 27 – continued from previous page

deepcopy(self)	Create a deep copy of this parameter.
draw_distribution_graph(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
draw_sample(self[, random_state])	Draws a single sample value from this parameter.
draw_samples(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**StochasticParameter**

Bases: object

Abstract parent class for all stochastic parameters.

Stochastic parameters are here all parameters from which values are supposed to be sampled. Usually the sampled values are to a degree random. E.g. a stochastic parameter may be the uniform distribution over the interval `[-10, 10]`. Samples from that distribution (and therefore the stochastic parameter) could be `5.2`, `-3.7`, `-9.7`, `6.4`, etc.

Methods

copy(self)	Create a shallow copy of this parameter.
deepcopy(self)	Create a deep copy of this parameter.
draw_distribution_graph(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
draw_sample(self[, random_state])	Draws a single sample value from this parameter.
draw_samples(self, size[, random_state])	Draw one or more samples from the parameter.

copy(*self*)

Create a shallow copy of this parameter.

Returns Shallow copy.

Return type *imgaug.parameters.StochasticParameter*

deepcopy(*self*)

Create a deep copy of this parameter.

Returns Deep copy.

Return type *imgaug.parameters.StochasticParameter*

draw_distribution_graph(*self*, *title=None*, *size=(1000, 1000)*, *bins=100*)

Generate an image visualizing the parameter's sample distribution.

Parameters

- **title** (*None or False or str, optional*) – Title of the plot. `None` is automatically replaced by a title derived from `str(param)`. If set to `False`, no title will be shown.
- **size** (*tuple of int*) – Number of points to sample. This is always expected to have at least two values. The first defines the number of sampling runs, the second (and further) dimensions define the size assigned to each *imgaug.parameters.StochasticParameter.draw_samples()* call. E.g. `(10, 20, 15)` will lead to `10` calls of `draw_samples(size=(20, 15))`. The results will be merged to a single 1d array.
- **bins** (*int*) – Number of bins in the plot histograms.

Returns **data** – Image of the plot.

Return type (H,W,3) ndarray

draw_sample(*self*, *random_state=None*)
Draws a single sample value from this parameter.

Parameters **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – A seed or random number generator to use during the sampling process. If `None`, the global RNG will be used. See also `imgaug.augmenters.meta.Augmenter.__init__()` for a similar parameter with more details.

Returns A single sample value.

Return type any

draw_samples(*self*, *size*, *random_state=None*)
Draw one or more samples from the parameter.

Parameters

- **size** (*tuple of int or int*) – Number of samples by dimension.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – A seed or random number generator to use during the sampling process. If `None`, the global RNG will be used. See also `imgaug.augmenters.meta.Augmenter.__init__()` for a similar parameter with more details.

Returns Sampled values. Usually a numpy ndarray of basically any dtype, though not strictly limited to numpy arrays. Its shape is expected to match *size*.

Return type ndarray

class imgaug.parameters.**Subtract**(*other_param*, *val*, *elementwise=False*)
Bases: *imgaug.parameters.StochasticParameter*

Subtract from the samples of another stochastic parameter.

Parameters

- **other_param** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Samples of *val* will be subtracted from samples of this parameter. Let `S` be the requested shape of samples, then the datatype behaviour is as follows:
  - If a single `number`, this `number` will be used as a constant value to fill an array of shape `S`.
  - If a `tuple` of two `number`s `(a, b)`, an array of shape `S` will be filled with uniformly sampled values from the continuous interval `[a, b)`.
  - If a `list` of `number`, an array of shape `S` will be filled with randomly picked values from the `list`.
  - If a *StochasticParameter*, that parameter will be queried once per call to generate an array of shape `S`.

  "per call" denotes a call of `Subtract.draw_sample()` or `Subtract.draw_samples()`.
- **val** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Value to subtract from the other parameter.

Datatype behaviour is analogous to *other_param*, though if `elementwise=False` (the default), only a single sample will be generated per call instead of `S`.

- **elementwise** (*bool, optional*) – Controls the sampling behaviour of *val*. If set to `False`, a single samples will be requested from *val* and used as the constant multiplier. If set to `True`, samples of shape `S` will be requested from *val* and subtracted elementwise from the samples of *other_param*.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.Subtract(iap.Uniform(0.0, 1.0), 1.0)
```

Convert a uniform distribution from `[0.0, 1.0)` to `[-1.0, 0.0)`.

Methods

`copy`(self)	Create a shallow copy of this parameter.
`deepcopy`(self)	Create a deep copy of this parameter.
`draw_distribution_graph`(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
`draw_sample`(self[, random_state])	Draws a single sample value from this parameter.
`draw_samples`(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**TruncatedNormal**(*loc*, *scale*, *low=-inf*, *high=inf*)

Bases: *imgaug.parameters.StochasticParameter*

Parameter that resembles a truncated normal distribution.

A truncated normal distribution is similar to a normal distribution, except the domain is smoothly bounded to a min and max value.

This is a wrapper around `scipy.stats.truncnorm()`.

Parameters

- **loc** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) –

  The mean of the normal distribution.

  – If a single `number`, this `number` will be used as a constant value.

  – If a `tuple` of two `number`s `(a, b)`, the value will be sampled from the continuous interval `[a, b)` once per call.

  – If a `list` of `number`, a random value will be picked from the `list` once per call.

  – If a *StochasticParameter*, that parameter will be queried once per call.

  "per call" denotes a call of `TruncatedNormal.draw_sample()` or `TruncatedNormal.draw_samples()`.

- **scale** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – The standard deviation of the normal distribution. If this parameter reaches `0`, the output array will be filled with *loc*. Datatype behaviour is the same as for *loc*.

- **low** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – The minimum value of the truncated normal distribution. Datatype behaviour is the same as for *loc*.
- **high** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – The maximum value of the truncated normal distribution. Datatype behaviour is the same as for *loc*.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.TruncatedNormal(0, 5.0, low=-10, high=10)
>>> samples = param.draw_samples(100, random_state=0)
>>> assert np.all(samples >= -10)
>>> assert np.all(samples <= 10)
```

Create a truncated normal distribution with its minimum at `-10.0` and its maximum at `10.0`.

Methods

copy(self)	Create a shallow copy of this parameter.
deepcopy(self)	Create a deep copy of this parameter.
draw_distribution_graph(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
draw_sample(self[, random_state])	Draws a single sample value from this parameter.
draw_samples(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**Uniform**(*a*, *b*)

Bases: *imgaug.parameters.StochasticParameter*

Parameter that resembles a uniform distribution over `[a, b)`.

Parameters

- **a** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Lower bound of the interval. If `a>b`, *a* and *b* will automatically be flipped. If `a==b`, all generated values will be identical to *a*.
  - If a single `number`, this `number` will be used as a constant value.
  - If a `tuple` of two `number` s `(a, b)`, the value will be sampled from the continuous interval `[a, b)` once per call.
  - If a `list` of `number`, a random value will be picked from the `list` once per call.
  - If a *StochasticParameter*, that parameter will be queried once per call.

  "per call" denotes a call of `Uniform.draw_sample()` or `Uniform.draw_samples()`.
- **b** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Upper bound of the interval. Analogous to *a*.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.Uniform(0, 10.0)
>>> sample = param.draw_sample()
>>> assert 0 <= sample < 10.0
```

Create and sample from a uniform distribution over `[0, 10.0)`.

Methods

`copy`(self)	Create a shallow copy of this parameter.
`deepcopy`(self)	Create a deep copy of this parameter.
`draw_distribution_graph`(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
`draw_sample`(self[, random_state])	Draws a single sample value from this parameter.
`draw_samples`(self, size[, random_state])	Draw one or more samples from the parameter.

class imgaug.parameters.**Weibull**(*a*)

Bases: *imgaug.parameters.StochasticParameter*

Parameter that resembles a (continuous) weibull distribution.

This is a wrapper around numpy's `numpy.random.weibull()`.

Parameters a (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) –

Shape parameter of the distribution.

- If a single `number`, this `number` will be used as a constant value.
- If a `tuple` of two `number` s `(a, b)`, the value will be sampled from the continuous interval `[a, b)` once per call.
- If a `list` of `number`, a random value will be picked from the `list` once per call.
- If a *StochasticParameter*, that parameter will be queried once per call.

"per call" denotes a call of `Weibull.draw_sample()` or `Weibull.draw_samples()`.

Examples

```
>>> import imgaug.parameters as iap
>>> param = iap.Weibull(a=0.5)
```

Create a weibull distribution with shape 0.5.

Methods

`copy`(self)	Create a shallow copy of this parameter.
`deepcopy`(self)	Create a deep copy of this parameter.

Continued on next page

Table 32 – continued from previous page

draw_distribution_graph(self[, title, size, ...])	Generate an image visualizing the parameter's sample distribution.
draw_sample(self[, random_state])	Draws a single sample value from this parameter.
draw_samples(self, size[, random_state])	Draw one or more samples from the parameter.

imgaug.parameters.**both_np_float_if_one_is_float**(*a*, *b*)

imgaug.parameters.**draw_distributions_grid**(*params*, *rows=None*, *cols=None*, *graph_sizes=(350, 350)*, *sample_sizes=None*, *titles=None*)

imgaug.parameters.**force_np_float_dtype**(*val*)

imgaug.parameters.**handle_continuous_param**(*param*, *name*, *value_range=None*, *tuple_to_uniform=True*, *list_to_choice=True*)

imgaug.parameters.**handle_discrete_kernel_size_param**(*param*, *name*, *value_range=(1, None)*, *allow_floats=True*)

imgaug.parameters.**handle_discrete_param**(*param*, *name*, *value_range=None*, *tuple_to_uniform=True*, *list_to_choice=True*, *allow_floats=True*)

imgaug.parameters.**handle_probability_param**(*param*, *name*, *tuple_to_uniform=False*, *list_to_choice=False*)

imgaug.parameters.**show_distributions_grid**(*params*, *rows=None*, *cols=None*, *graph_sizes=(350, 350)*, *sample_sizes=None*, *titles=None*)

13.3 imgaug.multicore

Classes and functions dealing with augmentation on multiple CPU cores.

class imgaug.multicore.**BackgroundAugmenter**(*batch_loader*, *augseq*, *queue_size=50*, *nb_workers='auto'*)

Bases: object

Deprecated. Augment batches in the background processes.

Deprecated. Use imgaug.multicore.Pool instead.

This is a wrapper around the multiprocessing module.

Parameters

- **batch_loader** (*BatchLoader or multiprocessing.Queue*) – BatchLoader object that loads the data fed into the BackgroundAugmenter, or alternatively a Queue. If a Queue, then it must be made sure that a final None in the Queue signals that the loading is finished and no more batches will follow. Otherwise the BackgroundAugmenter will wait forever for the next batch.
- **augseq** (*Augmenter*) – An augmenter to apply to all loaded images. This may be e.g. a Sequential to apply multiple augmenters.
- **queue_size** (*int*) – Size of the queue that is used to temporarily save the augmentation results. Larger values offer the background processes more room to save results when the main process doesn't load much, i.e. they can lead to smoother and faster training. For large images, high values can block a lot of RAM though.

- **nb_workers** (*'auto' or int*) – Number of background workers to spawn. If `auto`, it will be set to `C-1`, where `C` is the number of CPU cores.

Methods

get_batch(self)	Returns a batch from the queue of augmented batches.
terminate(self)	Terminates all background processes immediately.

all_finished	

all_finished(*self*)

get_batch(*self*)
Returns a batch from the queue of augmented batches.

If workers are still running and there are no batches in the queue, it will automatically wait for the next batch.

Returns **out** – One batch or None if all workers have finished.

Return type None or imgaug.Batch

terminate(*self*)
Terminates all background processes immediately.

This will also free their RAM.

class `imgaug.multicore.`**BatchLoader**(*load_batch_func*, *queue_size=50*, *nb_workers=1*, *threaded=True*)

Bases: `object`

Deprecated. Load batches in the background.

Deprecated. Use `imgaug.multicore.Pool` instead.

Loaded batches can be accesses using `imgaug.BatchLoader.queue`.

Parameters

- **load_batch_func** (*callable or generator*) – Generator or generator function (i.e. function that yields Batch objects) or a function that returns a list of Batch objects. Background loading automatically stops when the last batch was yielded or the last batch in the list was reached.
- **queue_size** (*int, optional*) – Maximum number of batches to store in the queue. May be set higher for small images and/or small batches.
- **nb_workers** (*int, optional*) – Number of workers to run in the background.
- **threaded** (*bool, optional*) – Whether to run the background processes using threads (True) or full processes (False).

Methods

all_finished(self)	Determine whether the workers have finished the loading process.
terminate(self)	Stop all workers.

count_workers_alive	

all_finished(*self*)
Determine whether the workers have finished the loading process.

Returns **out** – True if all workers have finished. Else False.

Return type bool

count_workers_alive(*self*)

terminate(*self*)
Stop all workers.

class imgaug.multicore.**Pool**(*augseq*, *processes=None*, *maxtasksperchild=None*, *seed=None*)
Bases: object

Wrapper around multiprocessing.Pool for multicore augmentation.

Parameters

- **augseq** (*imgaug.meta.Augmenter*) – The augmentation sequence to apply to batches.
- **processes** (*None or int, optional*) – The number of background workers, similar to the same parameter in multiprocessing.Pool. If None, the number of the machine's CPU cores will be used (this counts hyperthreads as CPU cores). If this is set to a negative value p, then P - abs(p) will be used, where P is the number of CPU cores. E.g. -1 would use all cores except one (this is useful to e.g. reserve one core to feed batches to the GPU).
- **maxtasksperchild** (*None or int, optional*) – The number of tasks done per worker process before the process is killed and restarted, similar to the same parameter in multiprocessing.Pool. If None, worker processes will not be automatically restarted.
- **seed** (*None or int, optional*) – The seed to use for child processes. If None, a random seed will be used.

Attributes

pool	Return or create the multiprocessing.Pool instance.

Methods

close(self)	Close the pool gracefully.
imap_batches(self, batches[, chunksize, ...])	Augment batches from a generator.
imap_batches_unordered(self, batches[, ...])	Augment batches from a generator (without preservation of order).
join(self)	Wait for the workers to exit.
map_batches(self, batches[, chunksize])	Augment a list of batches.
map_batches_async(self, batches[, ...])	Augment batches asynchonously.
terminate(self)	Terminate the pool immediately.

close(*self*)
Close the pool gracefully.

imap_batches(*self*, *batches*, *chunksize=1*, *output_buffer_size=None*)
Augment batches from a generator.

Pattern for output buffer constraint is from https://stackoverflow.com/a/47058399.

Parameters

- **batches** (*generator of imgaug.augmentables.batches.Batch*) – The batches to augment, provided as a generator. Each call to the generator should yield exactly one batch.
- **chunksize** (*None or int, optional*) – Rough indicator of how many tasks should be sent to each worker. Increasing this number can improve performance.
- **output_buffer_size** (*None or int, optional*) – Max number of batches to handle *at the same time* in the *whole* pipeline (including already augmented batches that are waiting to be requested). If the buffer size is reached, no new batches will be loaded from *batches* until a produced (i.e. augmented) batch is consumed (i.e. requested from this method). The buffer is unlimited if this is set to `None`. For large datasets, this should be set to an integer value to avoid filling the whole RAM if loading+augmentation happens faster than training.

  New in version 0.3.0.

Yields *imgaug.augmentables.batches.Batch* – Augmented batch.

imap_batches_unordered(*self*, *batches*, *chunksize=1*, *output_buffer_size=None*)
Augment batches from a generator (without preservation of order).

Pattern for output buffer constraint is from https://stackoverflow.com/a/47058399.

Parameters

- **batches** (*generator of imgaug.augmentables.batches.Batch*) – The batches to augment, provided as a generator. Each call to the generator should yield exactly one batch.
- **chunksize** (*None or int, optional*) – Rough indicator of how many tasks should be sent to each worker. Increasing this number can improve performance.
- **output_buffer_size** (*None or int, optional*) – Max number of batches to handle *at the same time* in the *whole* pipeline (including already augmented batches that are waiting to be requested). If the buffer size is reached, no new batches will be loaded from *batches* until a produced (i.e. augmented) batch is consumed (i.e. requested from this method). The buffer is unlimited if this is set to `None`. For large datasets, this should be set to an integer value to avoid filling the whole RAM if loading+augmentation happens faster than training.

  New in version 0.3.0.

Yields *imgaug.augmentables.batches.Batch* – Augmented batch.

join(*self*)
Wait for the workers to exit.

This may only be called after first calling *imgaug.multicore.Pool.close()* or *imgaug.multicore.Pool.terminate()*.

map_batches(*self*, *batches*, *chunksize=None*)
Augment a list of batches.

Parameters

- **batches** (*list of imgaug.augmentables.batches.Batch*) – The batches to augment.
- **chunksize** (*None or int, optional*) – Rough indicator of how many tasks should be sent to each worker. Increasing this number can improve performance.

Returns Augmented batches.

Return type list of imgaug.augmentables.batches.Batch

map_batches_async(*self*, *batches*, *chunksize=None*, *callback=None*, *error_callback=None*)
Augment batches asynchonously.

Parameters

- **batches** (*list of imgaug.augmentables.batches.Batch*) – The batches to augment.
- **chunksize** (*None or int, optional*) – Rough indicator of how many tasks should be sent to each worker. Increasing this number can improve performance.
- **callback** (*None or callable, optional*) – Function to call upon finish. See `multiprocessing.Pool`.
- **error_callback** (*None or callable, optional*) – Function to call upon errors. See `multiprocessing.Pool`.

Returns Asynchonous result. See `multiprocessing.Pool`.

Return type multiprocessing.MapResult

pool
Return or create the `multiprocessing.Pool` instance.

This creates a new instance upon the first call and afterwards returns that instance (until the property `_pool` is set to `None` again).

Returns The `multiprocessing.Pool` used internally by this `imgaug.multicore.Pool`.

Return type multiprocessing.Pool

terminate(*self*)
Terminate the pool immediately.

13.4 imgaug.dtypes

imgaug.dtypes.**change_dtype_**(*arr*, *dtype*, *clip=True*, *round=True*)

imgaug.dtypes.**change_dtypes_**(*images*, *dtypes*, *clip=True*, *round=True*)

imgaug.dtypes.**clip_**(*array*, *min_value*, *max_value*)

imgaug.dtypes.**clip_to_dtype_value_range_**(*array*, *dtype*, *validate=True*, *validate_values=None*)

imgaug.dtypes.**copy_dtypes_for_restore**(*images*, *force_list=False*)

imgaug.dtypes.**gate_dtypes**(*dtypes*, *allowed*, *disallowed*, *augmenter=None*)

imgaug.dtypes.**get_minimal_dtype**(*arrays*, *increase_itemsize_factor=1*)

imgaug.dtypes.**get_value_range_of_dtype**(*dtype*)

imgaug.dtypes.**increase_array_resolutions_**(*arrays*, *factor*)

imgaug.dtypes.**increase_itemsize_of_dtype**(*dtype*, *factor*)

imgaug.dtypes.**normalize_dtype**(*dtype*)

imgaug.dtypes.**normalize_dtypes**(*dtypes*)

imgaug.dtypes.**promote_array_dtypes_**(*arrays*, *dtypes=None*, *increase_itemsize_factor=1*)

imgaug.dtypes.**restore_dtypes_**(*images*, *dtypes*, *clip=True*, *round=True*)

13.5 imgaug.random

Classes and functions related to pseudo-random number generation.

This module deals with the generation of pseudo-random numbers. It provides the *imgaug.random.RNG* class, which is the primary random number generator in imgaug. It also provides various utility functions related random number generation, such as copying random number generators or setting their state.

The main benefit of this module is to hide the actually used random number generation classes and methods behin imgaug-specific classes and methods. This allows to deal with numpy using two different interfaces (one old interface in numpy <=1.16 and a new one in numpy 1.17+). It also allows to potentially switch to a different framework/library in the future.

13.5.1 Definitions

- *numpy generator* or *numpy random number generator*: Usually an instance of numpy.random.Generator. Can often also denote an instance of numpy.random.RandomState as both have almost the same interface.
- *RandomState*: An instance of *numpy.random.RandomState*. Note that outside of this module, the term "random state" often roughly translates to "any random number generator with numpy-like interface in a given state", i.e. it can then include instances of numpy.random.Generator or *imgaug.random.RNG*.
- *RNG*: An instance of *imgaug.random.RNG*.

Examples

```
>>> import imgaug.random as iarandom
>>> rng = iarandom.RNG(1234)
>>> rng.integers(0, 1000)
```

Initialize a random number generator with seed 1234, then sample a single integer from the discrete interval [0, 1000). This will use a numpy.random.Generator in numpy 1.17+ and automatically fall back to numpy.random.RandomState in numpy <=1.16.

class imgaug.random.**RNG**(*generator*)

Bases: object

Random number generator for imgaug.

This class is a wrapper around numpy.random.Generator and automatically falls back to numpy.random.RandomState in case of numpy version 1.16 or lower. It allows to use numpy 1.17's sampling functions in 1.16 too and supports a variety of useful functions on the wrapped sampler, e.g. gettings its state or copying it.

Not supported sampling functions of numpy <=1.16:

- numpy.random.RandomState.rand()
- numpy.random.RandomState.randn()

- numpy.random.RandomState.randint()
- numpy.random.RandomState.random_integers()
- numpy.random.RandomState.random_sample()
- numpy.random.RandomState.ranf()
- numpy.random.RandomState.sample()
- numpy.random.RandomState.seed()
- numpy.random.RandomState.get_state()
- numpy.random.RandomState.set_state()

In *imgaug.random.RNG.choice()*, the *axis* argument is not yet supported.

Parameters **generator** (*None or int or RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState*) – The numpy random number generator to use. In case of numpy version 1.17 or later, this shouldn't be a RandomState as that class is outdated. Behaviour for different datatypes:

- If None: The global RNG is wrapped by this RNG (they are then effectively identical, any sampling on this RNG will affect the global RNG).
- If int: In numpy 1.17+, the value is used as a seed for a Generator wrapped by this RNG. I.e. it will be provided as the entropy to a SeedSequence, which will then be used for an SFC64 bit generator and wrapped by a Generator. In numpy <=1.16, the value is used as a seed for a RandomState, which is then wrapped by this RNG.
- If *RNG*: That RNG's generator attribute will be used as the generator for this RNG, i.e. the same as RNG(other_rng.generator).
- If numpy.random.Generator: That generator will be wrapped.
- If numpy.random.bit_generator.BitGenerator: A numpy generator will be created (and wrapped by this RNG) that contains the bit generator.
- If numpy.random.SeedSequence: A numpy generator will be created (and wrapped by this RNG) that contains an SFC64 bit generator initialized with the given SeedSequence.
- If numpy.random.RandomState: In numpy <=1.16, this RandomState will be wrapped and used to sample random values. In numpy 1.17+, a seed will be derived from this RandomState and a new numpy.generator.Generator based on an SFC64 bit generator will be created and wrapped by this RNG.

Attributes

state	Get the state of this RNG.

Methods

*advance_*(self)	Advance the RNG's internal state in-place by one step.
beta(self, a, b[, size])	Call numpy.random.Generator.beta().
binomial(self, n, p[, size])	Call numpy.random.Generator.binomial().

Continued on next page

Table 36 – continued from previous page

bytes(self, length)	Call `numpy.random.Generator.bytes()`.
chisquare(self, df[, size])	Call `numpy.random.Generator.chisquare()`.
choice(self, a[, size, replace, p])	Call `numpy.random.Generator.choice()`.
copy(self)	Create a copy of this RNG.
copy_unless_global_rng(self)	Create a copy of this RNG unless it is the global RNG.
create_fully_random()	Create a new RNG, based on entropy provided from the OS.
*create_pseudo_random_*()	Create a new RNG in pseudo-random fashion.
*derive_rng_*(self)	Create a child RNG.
*derive_rngs_*(self, n)	Create *n* child RNGs.
dirichlet(self, alpha[, size])	Call `numpy.random.Generator.dirichlet()`.
duplicate(self, n)	Create a list containing *n* times this RNG.
equals(self, other)	Estimate whether this RNG and *other* have the same state.
equals_global_rng(self)	Estimate whether this RNG has the same state as the global RNG.
exponential(self[, scale, size])	Call `numpy.random.Generator.exponential()`.
f(self, dfnum, dfden[, size])	Call `numpy.random.Generator.f()`.
gamma(self, shape[, scale, size])	Call `numpy.random.Generator.gamma()`.
*generate_seed_*(self)	Sample a random seed.
*generate_seeds_*(self, n)	Generate *n* random seed values.
geometric(self, p[, size])	Call `numpy.random.Generator.geometric()`.
gumbel(self[, loc, scale, size])	Call `numpy.random.Generator.gumbel()`.
hypergeometric(self, ngood, nbad, nsample[, ...])	Call `numpy.random.Generator.hypergeometric()`.
integers(self, low[, high, size, dtype, ...])	Call numpy's `integers()` or `randint()`.
is_global_rng(self)	Estimate whether this RNG is identical to the global RNG.
laplace(self[, loc, scale, size])	Call `numpy.random.Generator.laplace()`.
logistic(self[, loc, scale, size])	Call `numpy.random.Generator.logistic()`.
lognormal(self[, mean, sigma, size])	Call `numpy.random.Generator.lognormal()`.
logseries(self, p[, size])	Call `numpy.random.Generator.logseries()`.
multinomial(self, n, pvals[, size])	Call `numpy.random.Generator.multinomial()`.
multivariate_normal(self, mean, cov[, size, ...])	Call `numpy.random.Generator.multivariate_normal()`.
negative_binomial(self, n, p[, size])	Call `numpy.random.Generator.negative_binomial()`.
noncentral_chisquare(self, df, nonc[, size])	Call `numpy.random.Generator.noncentral_chisquare()`.
noncentral_f(self, dfnum, dfden, nonc[, size])	Call `numpy.random.Generator.noncentral_f()`.

Continued on next page

Table 36 – continued from previous page

normal(self[, loc, scale, size])	Call `numpy.random.Generator.normal()`.
pareto(self, a[, size])	Call `numpy.random.Generator.pareto()`.
permutation(self, x)	Call `numpy.random.Generator.permutation()`.
poisson(self[, lam, size])	Call `numpy.random.Generator.poisson()`.
power(self, a[, size])	Call `numpy.random.Generator.power()`.
random(self, size[, dtype, out])	Call numpy's `random()` or `random_sample()`.
rayleigh(self[, scale, size])	Call `numpy.random.Generator.rayleigh()`.
*reset_cache_*(self)	Reset all cache of this RNG.
*set_state_*(self, value)	Set the state if the RNG in-place.
shuffle(self, x)	Call `numpy.random.Generator.shuffle()`.
standard_cauchy(self[, size])	Call `numpy.random.Generator.standard_cauchy()`.
standard_exponential(self[, size, dtype, ...])	Call `numpy.random.Generator.standard_exponential()`.
standard_gamma(self, shape[, size, dtype, out])	Call `numpy.random.Generator.standard_gamma()`.
standard_normal(self[, size, dtype, out])	Call `numpy.random.Generator.standard_normal()`.
standard_t(self, df[, size])	Call `numpy.random.Generator.standard_t()`.
triangular(self, left, mode, right[, size])	Call `numpy.random.Generator.triangular()`.
uniform(self[, low, high, size])	Call `numpy.random.Generator.uniform()`.
*use_state_of_*(self, other)	Copy and use (in-place) the state of another RNG.
vonmises(self, mu, kappa[, size])	Call `numpy.random.Generator.vonmises()`.
wald(self, mean, scale[, size])	Call `numpy.random.Generator.wald()`.
weibull(self, a[, size])	Call `numpy.random.Generator.weibull()`.
zipf(self, a[, size])	Call `numpy.random.Generator.zipf()`.

advance_(*self*)

Advance the RNG's internal state in-place by one step.

This advances the underlying generator's state.

Note: This simply samples one or more random values. This means that a call of this method will not completely change the outputs of the next called sampling method. To achieve more drastic output changes, call *imgaug.random.RNG.derive_rng_()*.

Returns The RNG itself.

Return type *RNG*

beta(*self*, *a*, *b*, *size=None*)

Call `numpy.random.Generator.beta()`.

binomial (*self*, *n*, *p*, *size=None*)
Call `numpy.random.Generator.binomial()`.

bytes (*self*, *length*)
Call `numpy.random.Generator.bytes()`.

chisquare (*self*, *df*, *size=None*)
Call `numpy.random.Generator.chisquare()`.

choice (*self*, *a*, *size=None*, *replace=True*, *p=None*)
Call `numpy.random.Generator.choice()`.

copy (*self*)
Create a copy of this RNG.

Returns Copy of this RNG. The copy will produce the same random samples.

Return type *RNG*

copy_unless_global_rng (*self*)
Create a copy of this RNG unless it is the global RNG.

Returns Copy of this RNG unless it is the global RNG. In the latter case the RNG instance itself will be returned without any changes.

Return type *RNG*

classmethod create_fully_random()
Create a new RNG, based on entropy provided from the OS.

Returns A new RNG. It is not derived from any other previously created RNG, nor does it depend on the seeding of imgaug or numpy.

Return type *RNG*

classmethod create_pseudo_random_()
Create a new RNG in pseudo-random fashion.

A seed will be sampled from the current global RNG and used to initialize the new RNG.

This advandes the global RNG's state.

Returns A new RNG, derived from the current global RNG.

Return type *RNG*

derive_rng_ (*self*)
Create a child RNG.

This advances the underlying generator's state.

Returns A child RNG.

Return type *RNG*

derive_rngs_ (*self*, *n*)
Create *n* child RNGs.

This advances the underlying generator's state.

Parameters **n** (*int*) – Number of child RNGs to derive.

Returns Child RNGs.

Return type list of RNG

dirichlet(*self*, *alpha*, *size=None*)
Call `numpy.random.Generator.dirichlet()`.

duplicate(*self*, *n*)
Create a list containing *n* times this RNG.

This method was mainly introduced as a replacement for previous calls of *imgaug.random.RNG.derive_rngs_()*. These calls turned out to be very slow in numpy 1.17+ and were hence replaced by simple duplication (except for the cases where child RNGs absolutely *had* to be created). This RNG duplication method doesn't help very much against code repetition, but it does *mark* the points where it would be desirable to create child RNGs for various reasons. Once deriving child RNGs is somehow sped up in the future, these calls can again be easily found and replaced.

Parameters **n** (*int*) – Length of the output list.

Returns List containing *n* times this RNG (same instances, no copies).

Return type list of RNG

equals(*self*, *other*)
Estimate whether this RNG and *other* have the same state.

Returns `True` if this RNG's generator and the generator of *other* have equal internal states. `False` otherwise.

Return type bool

equals_global_rng(*self*)
Estimate whether this RNG has the same state as the global RNG.

Returns `True` is this RNG has the same state as the global RNG, i.e. it will lead to the same sampled values given the same sampling method calls. The RNGs *don't* have to be identical object instances, which protects against e.g. copy effects. `False` otherwise.

Return type bool

exponential(*self*, *scale=1.0*, *size=None*)
Call `numpy.random.Generator.exponential()`.

f(*self*, *dfnum*, *dfden*, *size=None*)
Call `numpy.random.Generator.f()`.

gamma(*self*, *shape*, *scale=1.0*, *size=None*)
Call `numpy.random.Generator.gamma()`.

generate_seed_(*self*)
Sample a random seed.

This advances the underlying generator's state.

See `SEED_MIN_VALUE` and `SEED_MAX_VALUE` for the seed's value range.

Returns The sampled seed.

Return type int

generate_seeds_(*self*, *n*)
Generate *n* random seed values.

This advances the underlying generator's state.

See `SEED_MIN_VALUE` and `SEED_MAX_VALUE` for the seed's value range.

Parameters **n** (*int*) – Number of seeds to sample.

Returns 1D-array of `int32` seeds.

Return type ndarray

geometric(*self*, *p*, *size=None*)
Call `numpy.random.Generator.geometric()`.

gumbel(*self*, *loc=0.0*, *scale=1.0*, *size=None*)
Call `numpy.random.Generator.gumbel()`.

hypergeometric(*self*, *ngood*, *nbad*, *nsample*, *size=None*)
Call `numpy.random.Generator.hypergeometric()`.

integers(*self*, *low*, *high=None*, *size=None*, *dtype='int32'*, *endpoint=False*)
Call numpy's `integers()` or `randint()`.

Note: Changed *dtype* argument default value from numpy's `int64` to `int32`.

---

is_global_rng(*self*)
Estimate whether this RNG is identical to the global RNG.

Returns `True` is this RNG's underlying generator is identical to the global RNG's underlying generator. The RNGs themselves may be different, only the wrapped generator matters. `False` otherwise.

Return type bool

laplace(*self*, *loc=0.0*, *scale=1.0*, *size=None*)
Call `numpy.random.Generator.laplace()`.

logistic(*self*, *loc=0.0*, *scale=1.0*, *size=None*)
Call `numpy.random.Generator.logistic()`.

lognormal(*self*, *mean=0.0*, *sigma=1.0*, *size=None*)
Call `numpy.random.Generator.lognormal()`.

logseries(*self*, *p*, *size=None*)
Call `numpy.random.Generator.logseries()`.

multinomial(*self*, *n*, *pvals*, *size=None*)
Call `numpy.random.Generator.multinomial()`.

multivariate_normal(*self*, *mean*, *cov*, *size=None*, *check_valid='warn'*, *tol=1e-08*)
Call `numpy.random.Generator.multivariate_normal()`.

negative_binomial(*self*, *n*, *p*, *size=None*)
Call `numpy.random.Generator.negative_binomial()`.

noncentral_chisquare(*self*, *df*, *nonc*, *size=None*)
Call `numpy.random.Generator.noncentral_chisquare()`.

noncentral_f(*self*, *dfnum*, *dfden*, *nonc*, *size=None*)
Call `numpy.random.Generator.noncentral_f()`.

normal(*self*, *loc=0.0*, *scale=1.0*, *size=None*)
Call `numpy.random.Generator.normal()`.

pareto(*self*, *a*, *size=None*)
Call `numpy.random.Generator.pareto()`.

permutation(*self*, *x*)
Call `numpy.random.Generator.permutation()`.

poisson (*self*, *lam=1.0*, *size=None*)
Call `numpy.random.Generator.poisson()`.

power (*self*, *a*, *size=None*)
Call `numpy.random.Generator.power()`.

random (*self*, *size*, *dtype='float32'*, *out=None*)
Call numpy's `random()` or `random_sample()`.

Note: Changed *dtype* argument default value from numpy's `d` to `float32`.

---

rayleigh (*self*, *scale=1.0*, *size=None*)
Call `numpy.random.Generator.rayleigh()`.

reset_cache_ (*self*)
Reset all cache of this RNG.

Returns The RNG itself.

Return type *RNG*

set_state_ (*self*, *value*)
Set the state if the RNG in-place.

Parameters **value** (*tuple or dict*) – The new state of the RNG. Should correspond to the output of the `state` property.

Returns The RNG itself.

Return type *RNG*

shuffle (*self*, *x*)
Call `numpy.random.Generator.shuffle()`.

standard_cauchy (*self*, *size=None*)
Call `numpy.random.Generator.standard_cauchy()`.

standard_exponential (*self*, *size=None*, *dtype='float32'*, *method='zig'*, *out=None*)
Call `numpy.random.Generator.standard_exponential()`.

Note: Changed *dtype* argument default value from numpy's `d` to `float32`.

---

standard_gamma (*self*, *shape*, *size=None*, *dtype='float32'*, *out=None*)
Call `numpy.random.Generator.standard_gamma()`.

Note: Changed *dtype* argument default value from numpy's `d` to `float32`.

---

standard_normal (*self*, *size=None*, *dtype='float32'*, *out=None*)
Call `numpy.random.Generator.standard_normal()`.

Note: Changed *dtype* argument default value from numpy's `d` to `float32`.

---

standard_t (*self*, *df*, *size=None*)
Call `numpy.random.Generator.standard_t()`.

state
Get the state of this RNG.

Returns The state of the RNG. In numpy 1.17+, the bit generator's state will be returned. In numpy <=1.16, the `RandomState` 's state is returned. In both cases the state is a copy. In-place changes will not affect the RNG.

Return type tuple or dict

triangular(*self*, *left*, *mode*, *right*, *size=None*)
Call `numpy.random.Generator.triangular()`.

uniform(*self*, *low=0.0*, *high=1.0*, *size=None*)
Call `numpy.random.Generator.uniform()`.

use_state_of_(*self*, *other*)
Copy and use (in-place) the state of another RNG.

Note: It is often sensible to first verify that neither this RNG nor *other* are identical to the global RNG.

Parameters **other** (*RNG*) – The other RNG, which's state will be copied.

Returns The RNG itself.

Return type *RNG*

vonmises(*self*, *mu*, *kappa*, *size=None*)
Call `numpy.random.Generator.vonmises()`.

wald(*self*, *mean*, *scale*, *size=None*)
Call `numpy.random.Generator.wald()`.

weibull(*self*, *a*, *size=None*)
Call `numpy.random.Generator.weibull()`.

zipf(*self*, *a*, *size=None*)
Call `numpy.random.Generator.zipf()`.

imgaug.random.**advance_generator_**(*generator*)
Advance a numpy random generator's internal state in-place by one step.

This advances the generator's state.

Note: This simply samples one or more random values. This means that a call of this method will not completely change the outputs of the next called sampling method. To achieve more drastic output changes, call *imgaug.random.derive_generator_()*.

Parameters **generator** (*numpy.random.Generator or numpy.random.RandomState*) – Generator of which to advance the internal state.

imgaug.random.**convert_seed_sequence_to_generator**(*seed_sequence*)
Convert a seed sequence to a numpy (random number) generator.

Parameters **seed_sequence** (*numpy.random.SeedSequence*) – The seed value to use.

Returns Generator initialized with the provided seed sequence.

Return type numpy.random.Generator

imgaug.random.**convert_seed_to_generator**(*entropy*)
Convert a seed value to a numpy (random number) generator.

Parameters **entropy** (*int*) – The seed value to use.

Returns In numpy <=1.16 a `RandomState`, in 1.17+ a `Generator`. Both are initialized with the provided seed.

Return type numpy.random.Generator or numpy.random.RandomState

imgaug.random.**copy_generator**(*generator*)
Copy an existing numpy (random number) generator.

Parameters **generator** (*numpy.random.Generator or numpy.random.RandomState*) – The generator to copy.

Returns In numpy <=1.16 a `RandomState`, in 1.17+ a `Generator`. Both are copies of the input argument.

Return type numpy.random.Generator or numpy.random.RandomState

imgaug.random.**copy_generator_unless_global_generator**(*generator*)
Copy a numpy generator unless it is the current global generator.

"global generator" here denotes the generator contained in the global RNG's `.generator` attribute.

Parameters **generator** (*numpy.random.Generator or numpy.random.RandomState*) – The generator to copy.

Returns In numpy <=1.16 a `RandomState`, in 1.17+ a `Generator`. Both are copies of the input argument, unless that input is identical to the global generator. If it is identical, the instance itself will be returned without copying it.

Return type numpy.random.Generator or numpy.random.RandomState

imgaug.random.**create_fully_random_generator**()
Create a new numpy (random) generator, derived from OS's entropy.

Returns In numpy <=1.16 a `RandomState`, in 1.17+ a `Generator`. Both are initialized with entropy requested from the OS. They are hence independent of entered seeds or the library's global RNG.

Return type numpy.random.Generator or numpy.random.RandomState

imgaug.random.**create_pseudo_random_generator_**()
Create a new numpy (random) generator, derived from the global RNG.

This function advances the global RNG's state.

Returns In numpy <=1.16 a `RandomState`, in 1.17+ a `Generator`. Both are initialized with a seed sampled from the global RNG.

Return type numpy.random.Generator or numpy.random.RandomState

imgaug.random.**derive_generator_**(*generator*)
Create a child numpy (random number) generator from an existing one.

This advances the generator's state.

Parameters **generator** (*numpy.random.Generator or numpy.random.RandomState*) – The generator from which to derive a new child generator.

Returns In numpy <=1.16 a `RandomState`, in 1.17+ a `Generator`. In both cases a derived child generator.

Return type numpy.random.Generator or numpy.random.RandomState

imgaug.random.**derive_generators_**(*generator*, *n*)
Create child numpy (random number) generators from an existing one.

Parameters

- **generator** (*numpy.random.Generator or numpy.random.RandomState*) – The generator from which to derive new child generators.
- **n** (*int*) – Number of child generators to derive.

Returns In numpy <=1.16 a list of `RandomState` s, in 1.17+ a list of `Generator` s. In both cases lists of derived child generators.

Return type list of numpy.random.Generator or list of numpy.random.RandomState

imgaug.random.**generate_seed_**(*generator*)
Sample a seed from the provided generator.

This function advances the generator's state.

See `SEED_MIN_VALUE` and `SEED_MAX_VALUE` for the seed's value range.

Parameters **generator** (*numpy.random.Generator or numpy.random.RandomState*) – The generator from which to sample the seed.

Returns The sampled seed.

Return type int

imgaug.random.**generate_seeds_**(*generator*, *n*)
Sample *n* seeds from the provided generator.

This function advances the generator's state.

Parameters

- **generator** (*numpy.random.Generator or numpy.random.RandomState*) – The generator from which to sample the seed.
- **n** (*int*) – Number of seeds to sample.

Returns 1D-array of `int32` seeds.

Return type ndarray

imgaug.random.**get_generator_state**(*generator*)
Get the state of this provided generator.

Parameters **generator** (*numpy.random.Generator or numpy.random.RandomState*) – The generator, which's state is supposed to be extracted.

Returns The state of the generator. In numpy 1.17+, the bit generator's state will be returned. In numpy <=1.16, the `RandomState` 's state is returned. In both cases the state is a copy. In-place changes will not affect the RNG.

Return type tuple or dict

imgaug.random.**get_global_rng**()
Get or create the current global RNG of imgaug.

Note that the first call to this function will create a global RNG.

Returns The global RNG to use.

Return type *RNG*

imgaug.random.**is_generator_equal_to**(*generator*, *other_generator*)
Estimate whether two generator have the same class and state.

Parameters

- **generator** (*numpy.random.Generator or numpy.random.RandomState*) – First generator used in the comparison.
- **other_generator** (*numpy.random.Generator or numpy.random.RandomState*) – Second generator used in the comparison.

Returns `True` if *generator* 's class and state are the same as the class and state of *other_generator*. `False` otherwise.

Return type bool

imgaug.random.**normalize_generator**(*generator*)
Normalize various inputs to a numpy (random number) generator.

This function will first copy the provided argument, i.e. it never returns a provided instance itself.

Parameters **generator** (*None or int or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState*) – The numpy random number generator to normalize. In case of numpy version 1.17 or later, this shouldn't be a `RandomState` as that class is outdated. Behaviour for different datatypes:

- If `None`: The global RNG's generator is returned.
- If `int`: In numpy 1.17+, the value is used as a seed for a `Generator`, i.e. it will be provided as the entropy to a `SeedSequence`, which will then be used for an `SFC64` bit generator and wrapped by a `Generator`, which is then returned. In numpy <=1.16, the value is used as a seed for a `RandomState`, which will then be returned.
- If `numpy.random.Generator`: That generator will be returned.
- If `numpy.random.bit_generator.BitGenerator`: A numpy generator will be created and returned that contains the bit generator.
- If `numpy.random.SeedSequence`: A numpy generator will be created and returned that contains an `SFC64` bit generator initialized with the given `SeedSequence`.
- If `numpy.random.RandomState`: In numpy <=1.16, this `RandomState` will be returned. In numpy 1.17+, a seed will be derived from this `RandomState` and a new `numpy.generator.Generator` based on an `SFC64` bit generator will be created and returned.

Returns In numpy <=1.16 a `RandomState`, in 1.17+ a `Generator` (even if the input was a `RandomState`).

Return type numpy.random.Generator or numpy.random.RandomState

imgaug.random.**normalize_generator_**(*generator*)
Normalize in-place various inputs to a numpy (random number) generator.

This function will try to return the provided instance itself.

Parameters **generator** (*None or int or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState*) – See *imgaug.random.normalize_generator()*.

Returns In numpy <=1.16 a `RandomState`, in 1.17+ a `Generator` (even if the input was a `RandomState`).

Return type numpy.random.Generator or numpy.random.RandomState

imgaug.random.**polyfill_integers**(*generator*, *low*, *high=None*, *size=None*, *dtype='int32'*, *endpoint=False*)

Sample integers from a generator in different numpy versions.

Parameters

- **generator** (*numpy.random.Generator or numpy.random.RandomState*) – The generator to sample from. If it is a `RandomState`, `numpy.random.RandomState.randint()` will be called, otherwise `numpy.random.Generator.integers()`.
- **low** (*int or array-like of ints*) – See `numpy.random.Generator.integers()`.
- **high** (*int or array-like of ints, optional*) – See `numpy.random.Generator.integers()`.
- **size** (*int or tuple of ints, optional*) – See `numpy.random.Generator.integers()`.
- **dtype** (*{str, dtype}, optional*) – See `numpy.random.Generator.integers()`.
- **endpoint** (*bool, optional*) – See `numpy.random.Generator.integers()`.

Returns See `numpy.random.Generator.integers()`.

Return type int or ndarray of ints

imgaug.random.**polyfill_random**(*generator*, *size*, *dtype='float32'*, *out=None*)

Sample random floats from a generator in different numpy versions.

Parameters

- **generator** (*numpy.random.Generator or numpy.random.RandomState*) – The generator to sample from. Both `RandomState` and `Generator` support `random()`, but with different interfaces.
- **size** (*int or tuple of ints, optional*) – See `numpy.random.Generator.random()`.
- **dtype** (*{str, dtype}, optional*) – See `numpy.random.Generator.random()`.
- **out** (*ndarray, optional*) – See `numpy.random.Generator.random()`.

Returns See `numpy.random.Generator.random()`.

Return type float or ndarray of floats

imgaug.random.**reset_generator_cache_**(*generator*)

Reset a numpy (random number) generator's internal cache.

This function modifies the generator's state in-place.

Parameters **generator** (*numpy.random.Generator or numpy.random.RandomState*) – The generator of which to reset the cache.

Returns In numpy <=1.16 a `RandomState`, in 1.17+ a `Generator`. In both cases the input argument itself.

Return type numpy.random.Generator or numpy.random.RandomState

imgaug.random.**seed**(*entropy*)

Set the seed of imgaug's global RNG (in-place).

The global RNG controls most of the "randomness" in imgaug.

The global RNG is the default one used by all augmenters. Under special circumstances (e.g. when an augmenter is switched to deterministic mode), the global RNG is replaced with a local one. The state of that replacement may be dependent on the global RNG's state at the time of creating the child RNG.

Parameters **entropy** (*int*) – The seed value to use.

imgaug.random.**set_generator_state_**(*generator*, *state*)
Set the state of a numpy (random number) generator in-place.

Parameters

- **generator** (*numpy.random.Generator or numpy.random.RandomState*) – The generator, which's state is supposed to be modified.
- **state** (*tuple or dict*) – The new state of the generator. Should correspond to the output of *imgaug.random.get_generator_state()*.

imgaug.random.**supports_new_numpy_rng_style**()
Determine whether numpy supports the new `random` interface (v1.17+).

Returns `True` if the new `random` interface is supported by numpy, i.e. if numpy has version 1.17 or later. Otherwise `False`, i.e. numpy has version 1.16 or older and `numpy.random.RandomState` should be used instead.

Return type bool

13.6 imgaug.validation

Helper functions to validate input data and produce error messages.

imgaug.validation.**assert_is_iterable_of**(*iterable_var*, *classes*)
Assert that *iterable_var* only contains instances of given classes.

Parameters

- **iterable_var** (*iterable*) – See *imgaug.validation.is_iterable_of()*.
- **classes** (*type or iterable of type*) – See *imgaug.validation.is_iterable_of()*.

imgaug.validation.**convert_iterable_to_string_of_types**(*iterable_var*)
Convert an iterable of values to a string of their types.

Parameters **iterable_var** (*iterable*) – An iterable of variables, e.g. a list of integers.

Returns String representation of the types in *iterable_var*. One per item in *iterable_var*. Separated by commas.

Return type str

imgaug.validation.**is_iterable_of**(*iterable_var*, *classes*)
Check whether *iterable_var* contains only instances of given classes.

Parameters

- **iterable_var** (*iterable*) – An iterable of items that will be matched against *classes*.
- **classes** (*type or iterable of type*) – One or more classes that each item in *var* must be an instanceof. If this is an iterable, a single match per item is enough.

Returns Whether *var* only contains instances of *classes*. If *var* was empty, `True` will be returned.

Return type bool

13.7 imgaug.augmentables.batches

class imgaug.augmentables.batches.**Batch**(*images=None*, *heatmaps=None*, *segmentation_maps=None*, *keypoints=None*, *bounding_boxes=None*, *polygons=None*, *line_strings=None*, *data=None*)

Bases: object

Class encapsulating a batch before and after augmentation.

Parameters

- **images** (*None or (N,H,W,C) ndarray or list of (H,W,C) ndarray*) – The images to augment.
- **heatmaps** (*None or list of imgaug.augmentables.heatmaps.HeatmapsOnImage*) – The heatmaps to augment.
- **segmentation_maps** (*None or list of imgaug.augmentables.segmaps.SegmentationMapsOnImage*) – The segmentation maps to augment.
- **keypoints** (*None or list of imgaug.augmentables.kps.KeypointOnImage*) – The keypoints to augment.
- **bounding_boxes** (*None or list of imgaug.augmentables.bbs.BoundingBoxesOnImage*) – The bounding boxes to augment.
- **polygons** (*None or list of imgaug.augmentables.polys.PolygonsOnImage*) – The polygons to augment.
- **line_strings** (*None or list of imgaug.augmentables.lines.LineStringsOnImage*) – The line strings to augment.
- **data** – Additional data that is saved in the batch and may be read out after augmentation. This could e.g. contain filepaths to each image in *images*. As this object is usually used for background augmentation with multiple processes, the augmented Batch objects might not be returned in the original order, making this information useful.

Attributes

bounding_boxes **Deprecated**. Use `Batch.bounding_boxes_unaug` instead.

heatmaps **Deprecated**. Use `Batch.heatmaps_unaug` instead.

images **Deprecated**. Use `Batch.images_unaug` instead.

keypoints **Deprecated**. Use `Batch.keypoints_unaug` instead.

segmentation_maps **Deprecated**. Use `Batch.segmentation_maps_unaug` instead.

Methods

deepcopy	

bounding_boxes

Deprecated. Use `Batch.bounding_boxes_unaug` instead.

deepcopy(*self*, *images_unaug='DEFAULT'*, *images_aug='DEFAULT'*, *heatmaps_unaug='DEFAULT'*, *heatmaps_aug='DEFAULT'*, *segmentation_maps_unaug='DEFAULT'*, *segmentation_maps_aug='DEFAULT'*, *keypoints_unaug='DEFAULT'*, *keypoints_aug='DEFAULT'*, *bounding_boxes_unaug='DEFAULT'*, *bounding_boxes_aug='DEFAULT'*, *polygons_unaug='DEFAULT'*, *polygons_aug='DEFAULT'*, *line_strings_unaug='DEFAULT'*, *line_strings_aug='DEFAULT'*)

heatmaps

Deprecated. Use `Batch.heatmaps_unaug` instead.

images

Deprecated. Use `Batch.images_unaug` instead.

keypoints

Deprecated. Use `Batch.keypoints_unaug` instead.

segmentation_maps

Deprecated. Use `Batch.segmentation_maps_unaug` instead.

class imgaug.augmentables.batches.**UnnormalizedBatch**(*images=None*, *heatmaps=None*, *segmentation_maps=None*, *keypoints=None*, *bounding_boxes=None*, *polygons=None*, *line_strings=None*, *data=None*)

Bases: `object`

Class for batches of unnormalized data before and after augmentation.

Parameters

- **images** (*None or (N,H,W,C) ndarray or (N,H,W) ndarray or iterable of (H,W,C) ndarray or iterable of (H,W) ndarray*) – The images to augment.
- **heatmaps** (*None or (N,H,W,C) ndarray or imgaug.augmentables.heatmaps.HeatmapsOnImage or iterable of (H,W,C) ndarray or iterable of imgaug.augmentables.heatmaps.HeatmapsOnImage*) – The heatmaps to augment. If anything else than `HeatmapsOnImage`, then the number of heatmaps must match the number of images provided via parameter *images*. The number is contained either in `N` or the first iterable's size.
- **segmentation_maps** (*None or (N,H,W) ndarray or imgaug.augmentables.segmaps.SegmentationMapsOnImage or iterable of (H,W) ndarray or iterable of imgaug.augmentables.segmaps.SegmentationMapsOnImage*) – The segmentation maps to augment. If anything else than `SegmentationMapsOnImage`, then the number of segmaps must match the number of images provided via parameter *images*. The number is contained either in `N` or the first iterable's size.
- **keypoints** (*None or list of (N,K,2) ndarray or tuple of number or imgaug.augmentables.kps.Keypoint or iterable of (K,2) ndarray or iterable of tuple of number or iterable of imgaug.augmentables.kps.Keypoint or iterable of imgaug.augmentables.kps.KeypointOnImage or iterable of iterable of tuple of number or iterable of iterable of imgaug.augmentables.kps.Keypoint*) – The keypoints to augment. If a tuple (or iterable(s) of tuple), then iterpreted as (x,y) coordinates and must hence contain two numbers. A single tuple represents a single coordinate on one image, an iterable of tuples the coordinates on one image and an iterable of iterable of tuples the coordinates on several images. Analogous if `Keypoint` objects are used instead of tuples. If an ndarray, then `N` denotes the number of images and `K` the number of keypoints on each image. If anything else than `KeypointsOnImage` is provided, then the number of keypoint groups

must match the number of images provided via parameter *images*. The number is contained e.g. in `N` or in case of "iterable of iterable of tuples" in the first iterable's size.

- **bounding_boxes** (*None or (N,B,4) ndarray or tuple of number or imgaug.augmentables.bbs.BoundingBox or imgaug.augmentables.bbs.BoundingBoxesOnImage or iterable of (B,4) ndarray or iterable of tuple of number or iterable of imgaug.augmentables.bbs.BoundingBox or iterable of imgaug.augmentables.bbs.BoundingBoxesOnImage or iterable of iterable of tuple of number or iterable of iterable imgaug.augmentables.bbs.BoundingBox*) – The bounding boxes to augment. This is analogous to the *keypoints* parameter. However, each tuple – and also the last index in case of arrays – has size 4, denoting the bounding box coordinates `x1`, `y1`, `x2` and `y2`.

- **polygons** (*None or (N,#polys,#points,2) ndarray or imgaug.augmentables.polys.Polygon or imgaug.augmentables.polys.PolygonsOnImage or iterable of (#polys,#points,2) ndarray or iterable of tuple of number or iterable of imgaug.augmentables.kps.Keypoint or iterable of imgaug.augmentables.polys.Polygon or iterable of imgaug.augmentables.polys.PolygonsOnImage or iterable of iterable of (#points,2) ndarray or iterable of iterable of tuple of number or iterable of iterable of imgaug.augmentables.kps.Keypoint or iterable of iterable of imgaug.augmentables.polys.Polygon or iterable of iterable of iterable of tuple of number or iterable of iterable of iterable of tuple of imgaug.augmentables.kps.Keypoint*) – The polygons to augment. This is similar to the *keypoints* parameter. However, each polygon may be made up of several `(x,y)` coordinates (three or more are required for valid polygons). The following datatypes will be interpreted as a single polygon on a single image:

  – `imgaug.augmentables.polys.Polygon`

  – `iterable of tuple of number`

  – `iterable of imgaug.augmentables.kps.Keypoint`

  The following datatypes will be interpreted as multiple polygons on a single image:

  – `imgaug.augmentables.polys.PolygonsOnImage`

  – `iterable of imgaug.augmentables.polys.Polygon`

  – `iterable of iterable of tuple of number`

  – `iterable of iterable of imgaug.augmentables.kps.Keypoint`

  – `iterable of iterable of imgaug.augmentables.polys.Polygon`

  The following datatypes will be interpreted as multiple polygons on multiple images:

  – `(N,#polys,#points,2) ndarray`

  – `iterable of (#polys,#points,2) ndarray`

  – `iterable of iterable of (#points,2) ndarray`

  – `iterable of iterable of iterable of tuple of number`

  – `iterable of iterable of iterable of tuple of imgaug.augmentables.kps.Keypoint`

- **line_strings** (*None or (N,#lines,#points,2) ndarray or imgaug.augmentables.lines.LineString or imgaug.augmentables.lines.LineStringOnImage or iterable of (#lines,#points,2) ndarray or iterable of tuple of number or iterable of imgaug.augmentables.kps.Keypoint or iterable of imgaug.augmentables.lines.LineString*

or iterable of imgaug.augmentables.lines.LineStringOnImage or iterable of iterable of (#points,2) ndarray or iterable of iterable of tuple of number or iterable of iterable of imgaug.augmentables.kps.Keypoint or iterable of iterable of imgaug.augmentables.polys.LineString or iterable of iterable of iterable of tuple of number or iterable of iterable of iterable of tuple of imgaug.augmentables.kps.Keypoint) – The line strings to augment. See *polygons* for more details as polygons follow a similar structure to line strings.

- **data** – Additional data that is saved in the batch and may be read out after augmentation. This could e.g. contain filepaths to each image in *images*. As this object is usually used for background augmentation with multiple processes, the augmented Batch objects might not be returned in the original order, making this information useful.

Methods

fill_from_augmented_normalized_batch(self, ...)	Fill this batch with (normalized) augmentation results.
to_normalized_batch(self)	Convert this unnormalized batch to an instance of Batch.

fill_from_augmented_normalized_batch(*self*, *batch_aug_norm*)
Fill this batch with (normalized) augmentation results.

This method receives a (normalized) Batch instance, takes all `*_aug` attributes out if it and assigns them to this batch *in unnormalized form*. Hence, the datatypes of all `*_aug` attributes will match the datatypes of the `*_unaug` attributes.

Parameters **batch_aug_norm** (*imgaug.augmentables.batches.Batch*) – Batch after normalization and augmentation.

Returns New UnnormalizedBatch instance. All `*_unaug` attributes are taken from the old UnnormalizedBatch (without deepcopying them) and all `*_aug` attributes are taken from *batch_normalized* converted to unnormalized form.

Return type *imgaug.augmentables.batches.UnnormalizedBatch*

to_normalized_batch(*self*)
Convert this unnormalized batch to an instance of Batch.

As this method is intended to be called before augmentation, it assumes that none of the `*_aug` attributes is yet set. It will produce an AssertionError otherwise.

The newly created Batch's `*_unaug` attributes will match the ones in this batch, just in normalized form.

Returns The batch, with `*_unaug` attributes being normalized.

Return type *imgaug.augmentables.batches.Batch*

13.8 imgaug.augmentables.bbs

class imgaug.augmentables.bbs.**BoundingBox**(*x1*, *y1*, *x2*, *y2*, *label=None*)
Bases: object

Class representing bounding boxes.

Each bounding box is parameterized by its top left and bottom right corners. Both are given as x and y-coordinates. The corners are intended to lie inside the bounding box area. As a result, a bounding box that lies completely inside the image but has maximum extensions would have coordinates `(0.0, 0.0)` and `(W - epsilon, H - epsilon)`. Note that coordinates are saved internally as floats.

Parameters

- **x1** (*number*) – X-coordinate of the top left of the bounding box.
- **y1** (*number*) – Y-coordinate of the top left of the bounding box.
- **x2** (*number*) – X-coordinate of the bottom right of the bounding box.
- **y2** (*number*) – Y-coordinate of the bottom right of the bounding box.
- **label** (*None or str, optional*) – Label of the bounding box, e.g. a string representing the class.

Attributes

area Estimate the area of the bounding box.

center_x Estimate the x-coordinate of the center point of the bounding box.

center_y Estimate the y-coordinate of the center point of the bounding box.

height Estimate the height of the bounding box.

width Estimate the width of the bounding box.

x1_int Get the x-coordinate of the top left corner as an integer.

x2_int Get the x-coordinate of the bottom left corner as an integer.

y1_int Get the y-coordinate of the top left corner as an integer.

y2_int Get the y-coordinate of the bottom left corner as an integer.

Methods

clip_out_of_image(self, image)	Clip off all parts of the BB box that are outside of the image.
contains(self, other)	Estimate whether the bounding box contains a given point.
copy(self[, x1, y1, x2, y2, label])	Create a shallow copy of this BoundingBox instance.
cut_out_of_image(self, *args, **kwargs)	**Deprecated**.
deepcopy(self[, x1, y1, x2, y2, label])	Create a deep copy of the BoundingBox object.
draw_on_image(self, image[, color, alpha, ...])	Draw the bounding box on an image.
extend(self[, all_sides, top, right, ...])	Extend the size of the bounding box along its sides.
extract_from_image(self, image[, pad, ...])	Extract the image pixels within the bounding box.
intersection(self, other[, default])	Compute the intersection BB between this BB and another BB.
iou(self, other)	Compute the IoU between this bounding box and another one.
is_fully_within_image(self, image)	Estimate whether the bounding box is fully inside the image area.
is_out_of_image(self, image[, fully, partly])	Estimate whether the BB is partially/fully outside of the image area.

Continued on next page

Table 38 – continued from previous page

is_partly_within_image(self, image)	Estimate whether the BB is at least partially inside the image area.
project(self, from_shape, to_shape)	Project the bounding box onto a differently shaped image.
shift(self[, top, right, bottom, left])	Move this bounding box along the x/y-axis.
to_keypoints(self)	Convert the BB's corners to keypoints (clockwise, from top left).
union(self, other)	Compute the union BB between this BB and another BB.

area
Estimate the area of the bounding box.

Returns Area of the bounding box, i.e. `height * width`.

Return type number

center_x
Estimate the x-coordinate of the center point of the bounding box.

Returns X-coordinate of the center point of the bounding box.

Return type number

center_y
Estimate the y-coordinate of the center point of the bounding box.

Returns Y-coordinate of the center point of the bounding box.

Return type number

clip_out_of_image(*self*, *image*)
Clip off all parts of the BB box that are outside of the image.

Parameters **image** (*(H,W,...) ndarray or tuple of int*) – Image dimensions to use for the clipping of the bounding box. If an `ndarray`, its shape will be used. If a `tuple`, it is assumed to represent the image shape and must contain at least two integers.

Returns Bounding box, clipped to fall within the image dimensions.

Return type *imgaug.augmentables.bbs.BoundingBox*

contains(*self*, *other*)
Estimate whether the bounding box contains a given point.

Parameters **other** (*tuple of number or imgaug.augmentables.kps.Keypoint*) – Point to check for.

Returns `True` if the point is contained in the bounding box, `False` otherwise.

Return type bool

copy(*self*, *x1=None*, *y1=None*, *x2=None*, *y2=None*, *label=None*)
Create a shallow copy of this BoundingBox instance.

Parameters

- **x1** (*None or number*) – If not `None`, then the `x1` coordinate of the copied object will be set to this value.
- **y1** (*None or number*) – If not `None`, then the `y1` coordinate of the copied object will be set to this value.

- **x2** (*None or number*) – If not `None`, then the `x2` coordinate of the copied object will be set to this value.
- **y2** (*None or number*) – If not `None`, then the `y2` coordinate of the copied object will be set to this value.
- **label** (*None or string*) – If not `None`, then the `label` of the copied object will be set to this value.

Returns Shallow copy.

Return type *imgaug.augmentables.bbs.BoundingBox*

cut_out_of_image(*self*, **args*, ***kwargs*)

Deprecated. Use `BoundingBox.clip_out_of_image()` instead. clip_out_of_image() has the exactly same interface.

deepcopy(*self*, *x1=None*, *y1=None*, *x2=None*, *y2=None*, *label=None*)

Create a deep copy of the BoundingBox object.

Parameters

- **x1** (*None or number*) – If not `None`, then the `x1` coordinate of the copied object will be set to this value.
- **y1** (*None or number*) – If not `None`, then the `y1` coordinate of the copied object will be set to this value.
- **x2** (*None or number*) – If not `None`, then the `x2` coordinate of the copied object will be set to this value.
- **y2** (*None or number*) – If not `None`, then the `y2` coordinate of the copied object will be set to this value.
- **label** (*None or string*) – If not `None`, then the `label` of the copied object will be set to this value.

Returns Deep copy.

Return type *imgaug.augmentables.bbs.BoundingBox*

draw_on_image(*self*, *image*, *color=(0, 255, 0)*, *alpha=1.0*, *size=1*, *copy=True*, *raise_if_out_of_image=False*, *thickness=None*)

Draw the bounding box on an image.

Parameters

- **image** (*(H,W,C) ndarray*) – The image onto which to draw the bounding box. Currently expected to be `uint8`.
- **color** (*iterable of int, optional*) – The color to use, corresponding to the channel layout of the image. Usually RGB.
- **alpha** (*float, optional*) – The transparency of the drawn bounding box, where `1.0` denotes no transparency and `0.0` is invisible.
- **size** (*int, optional*) – The thickness of the bounding box in pixels. If the value is larger than `1`, then additional pixels will be added around the bounding box (i.e. extension towards the outside).
- **copy** (*bool, optional*) – Whether to copy the input image or change it in-place.
- **raise_if_out_of_image** (*bool, optional*) – Whether to raise an error if the bounding box is fully outside of the image. If set to `False`, no error will be raised and only the parts inside the image will be drawn.

- **thickness** (*None or int, optional*) – Deprecated.

Returns Image with bounding box drawn on it.

Return type (H,W,C) ndarray(uint8)

extend(*self*, *all_sides=0*, *top=0*, *right=0*, *bottom=0*, *left=0*)
Extend the size of the bounding box along its sides.

Parameters

- **all_sides** (*number, optional*) – Value by which to extend the bounding box size along all sides.
- **top** (*number, optional*) – Value by which to extend the bounding box size along its top side.
- **right** (*number, optional*) – Value by which to extend the bounding box size along its right side.
- **bottom** (*number, optional*) – Value by which to extend the bounding box size along its bottom side.
- **left** (*number, optional*) – Value by which to extend the bounding box size along its left side.

Returns Extended bounding box.

Return type imgaug.BoundingBox

extract_from_image(*self*, *image*, *pad=True*, *pad_max=None*, *prevent_zero_size=True*)
Extract the image pixels within the bounding box.

This function will zero-pad the image if the bounding box is partially/fully outside of the image.

Parameters

- **image** (*(H,W) ndarray or (H,W,C) ndarray*) – The image from which to extract the pixels within the bounding box.
- **pad** (*bool, optional*) – Whether to zero-pad the image if the object is partially/fully outside of it.
- **pad_max** (*None or int, optional*) – The maximum number of pixels that may be zero-paded on any side, i.e. if this has value `N` the total maximum of added pixels is `4*N`. This option exists to prevent extremely large images as a result of single points being moved very far away during augmentation.
- **prevent_zero_size** (*bool, optional*) – Whether to prevent the height or width of the extracted image from becoming zero. If this is set to `True` and the height or width of the bounding box is below `1`, the height/width will be increased to `1`. This can be useful to prevent problems, e.g. with image saving or plotting. If it is set to `False`, images will be returned as `(H', W')` or `(H', W', 3)` with `H` or `W` potentially being 0.

Returns Pixels within the bounding box. Zero-padded if the bounding box is partially/fully outside of the image. If *prevent_zero_size* is activated, it is guarantueed that `H'>0` and `W'>0`, otherwise only `H'>=0` and `W'>=0`.

Return type (H',W') ndarray or (H',W',C) ndarray

height
Estimate the height of the bounding box.

Returns Height of the bounding box.

Return type number

intersection (*self*, *other*, *default=None*)
Compute the intersection BB between this BB and another BB.

Note that in extreme cases, the intersection can be a single point. In that case the intersection bounding box exists and it will be returned, but it will have a height and width of zero.

Parameters

- **other** (*imgaug.augmentables.bbs.BoundingBox*) – Other bounding box with which to generate the intersection.
- **default** (*any, optional*) – Default value to return if there is no intersection.

Returns Intersection bounding box of the two bounding boxes if there is an intersection. If there is no intersection, the default value will be returned, which can by anything.

Return type *imgaug.augmentables.bbs.BoundingBox* or any

iou (*self*, *other*)
Compute the IoU between this bounding box and another one.

IoU is the intersection over union, defined as:

```
``area(intersection(A, B)) / area(union(A, B))``
``= area(intersection(A, B))
    / (area(A) + area(B) - area(intersection(A, B)))``
```

Parameters **other** (*imgaug.augmentables.bbs.BoundingBox*) – Other bounding box with which to compare.

Returns IoU between the two bounding boxes.

Return type float

is_fully_within_image (*self*, *image*)
Estimate whether the bounding box is fully inside the image area.

Parameters **image** (*(H,W,...) ndarray or tuple of int*) – Image dimensions to use. If an `ndarray`, its shape will be used. If a `tuple`, it is assumed to represent the image shape and must contain at least two integers.

Returns `True` if the bounding box is fully inside the image area. `False` otherwise.

Return type bool

is_out_of_image (*self*, *image*, *fully=True*, *partly=False*)
Estimate whether the BB is partially/fully outside of the image area.

Parameters

- **image** (*(H,W,...) ndarray or tuple of int*) – Image dimensions to use. If an `ndarray`, its shape will be used. If a `tuple`, it is assumed to represent the image shape and must contain at least two integers.
- **fully** (*bool, optional*) – Whether to return `True` if the bounding box is fully outside of the image area.
- **partly** (*bool, optional*) – Whether to return `True` if the bounding box is at least partially outside fo the image area.

Returns `True` if the bounding box is partially/fully outside of the image area, depending on defined parameters. `False` otherwise.

Return type bool

is_partly_within_image(*self*, *image*)
Estimate whether the BB is at least partially inside the image area.

Parameters **image** (*(H,W,...) ndarray or tuple of int*) – Image dimensions to use. If an `ndarray`, its shape will be used. If a `tuple`, it is assumed to represent the image shape and must contain at least two integers.

Returns `True` if the bounding box is at least partially inside the image area. `False` otherwise.

Return type bool

project(*self*, *from_shape*, *to_shape*)
Project the bounding box onto a differently shaped image.

E.g. if the bounding box is on its original image at `x1=(10 of 100 pixels)` and `y1=(20 of 100 pixels)` and is projected onto a new image with size `(width=200, height=200)`, its new position will be `(x1=20, y1=40)`. (Analogous for `x2`/`y2`.)

This is intended for cases where the original image is resized. It cannot be used for more complex changes (e.g. padding, cropping).

Parameters

- **from_shape** (*tuple of int or ndarray*) – Shape of the original image. (Before resize.)
- **to_shape** (*tuple of int or ndarray*) – Shape of the new image. (After resize.)

Returns `BoundingBox` instance with new coordinates.

Return type *imgaug.augmentables.bbs.BoundingBox*

shift(*self*, *top=None*, *right=None*, *bottom=None*, *left=None*)
Move this bounding box along the x/y-axis.

Parameters

- **top** (*None or int, optional*) – Amount of pixels by which to shift this object *from* the top (towards the bottom).
- **right** (*None or int, optional*) – Amount of pixels by which to shift this object *from* the right (towards the left).
- **bottom** (*None or int, optional*) – Amount of pixels by which to shift this object *from* the bottom (towards the top).
- **left** (*None or int, optional*) – Amount of pixels by which to shift this object *from* the left (towards the right).

Returns Shifted bounding box.

Return type *imgaug.augmentables.bbs.BoundingBox*

to_keypoints(*self*)
Convert the BB's corners to keypoints (clockwise, from top left).

Returns Corners of the bounding box as keypoints.

Return type list of imgaug.augmentables.kps.Keypoint

union(*self*, *other*)
Compute the union BB between this BB and another BB.

This is equivalent to drawing a bounding box around all corner points of both bounding boxes.

Parameters **other** (*imgaug.augmentables.bbs.BoundingBox*) – Other bounding box with which to generate the union.

Returns Union bounding box of the two bounding boxes.

Return type *imgaug.augmentables.bbs.BoundingBox*

width
Estimate the width of the bounding box.

Returns Width of the bounding box.

Return type number

x1_int
Get the x-coordinate of the top left corner as an integer.

Returns X-coordinate of the top left corner, rounded to the closest integer.

Return type int

x2_int
Get the x-coordinate of the bottom left corner as an integer.

Returns X-coordinate of the bottom left corner, rounded to the closest integer.

Return type int

y1_int
Get the y-coordinate of the top left corner as an integer.

Returns Y-coordinate of the top left corner, rounded to the closest integer.

Return type int

y2_int
Get the y-coordinate of the bottom left corner as an integer.

Returns Y-coordinate of the bottom left corner, rounded to the closest integer.

Return type int

class imgaug.augmentables.bbs.**BoundingBoxesOnImage**(*bounding_boxes*, *shape*)
Bases: object

Container for the list of all bounding boxes on a single image.

Parameters

- **bounding_boxes** (*list of imgaug.augmentables.bbs.BoundingBox*) – List of bounding boxes on the image.
- **shape** (*tuple of int*) – The shape of the image on which the bounding boxes are placed.

Examples

```
>>> import numpy as np
>>> from imgaug.augmentables.bbs import BoundingBox, BoundingBoxesOnImage
>>>
>>> image = np.zeros((100, 100))
>>> bbs = [
>>>     BoundingBox(x1=10, y1=20, x2=20, y2=30),
>>>     BoundingBox(x1=25, y1=50, x2=30, y2=70)
```

(continues on next page)

(continued from previous page)

```
>>> ]
>>> bbs_oi = BoundingBoxesOnImage(bbs, shape=image.shape)
```

Attributes

empty Determine whether this instance contains zero bounding boxes.

height Get the height of the image on which the bounding boxes fall.

width Get the width of the image on which the bounding boxes fall.

Methods

clip_out_of_image(self)	Clip off all parts from all BBs that are outside of the image.
copy(self)	Create a shallow copy of the `BoundingBoxesOnImage` instance.
cut_out_of_image(self)	**Deprecated**.
deepcopy(self)	Create a deep copy of the `BoundingBoxesOnImage` object.
draw_on_image(self, image[, color, alpha, ...])	Draw all bounding boxes onto a given image.
from_xyxy_array(xyxy, shape)	Convert an `(N,4)` `ndarray` to a `BoundingBoxesOnImage` instance.
on(self, image)	Project bounding boxes from one image (shape) to a another one.
remove_out_of_image(self[, fully, partly])	Remove all BBs that are fully/partially outside of the image.
shift(self[, top, right, bottom, left])	Move all all BBs along the x/y-axis.
to_xyxy_array(self[, dtype])	Convert the `BoundingBoxesOnImage` object to an `(N,4)` `ndarray`.

clip_out_of_image(*self*)
Clip off all parts from all BBs that are outside of the image.

Returns Bounding boxes, clipped to fall within the image dimensions.

Return type *imgaug.augmentables.bbs.BoundingBoxesOnImage*

copy(*self*)
Create a shallow copy of the `BoundingBoxesOnImage` instance.

Returns Shallow copy.

Return type *imgaug.augmentables.bbs.BoundingBoxesOnImage*

cut_out_of_image(*self*)
Deprecated. Use `BoundingBoxesOnImage.clip_out_of_image()` instead. clip_out_of_image() has the exactly same interface.

deepcopy(*self*)
Create a deep copy of the `BoundingBoxesOnImage` object.

Returns Deep copy.

Return type *imgaug.augmentables.bbs.BoundingBoxesOnImage*

draw_on_image(*self*, *image*, *color=(0, 255, 0)*, *alpha=1.0*, *size=1*, *copy=True*, *raise_if_out_of_image=False*, *thickness=None*)

Draw all bounding boxes onto a given image.

Parameters

- **image** (*(H,W,3) ndarray*) – The image onto which to draw the bounding boxes. This image should usually have the same shape as set in `BoundingBoxesOnImage.shape`.
- **color** (*int or list of int or tuple of int or (3,) ndarray, optional*) – The RGB color of all bounding boxes. If a single `int C`, then that is equivalent to `(C,C,C)`.
- **alpha** (*float, optional*) – Alpha/transparency of the bounding box.
- **size** (*int, optional*) – Thickness in pixels.
- **copy** (*bool, optional*) – Whether to copy the image before drawing the bounding boxes.
- **raise_if_out_of_image** (*bool, optional*) – Whether to raise an exception if any bounding box is outside of the image.
- **thickness** (*None or int, optional*) – Deprecated.

Returns Image with drawn bounding boxes.

Return type (H,W,3) ndarray

empty

Determine whether this instance contains zero bounding boxes.

Returns True if this object contains zero bounding boxes.

Return type bool

classmethod from_xyxy_array(*xyxy*, *shape*)

Convert an `(N,4) ndarray` to a `BoundingBoxesOnImage` instance.

This is the inverse of `imgaug.BoundingBoxesOnImage.to_xyxy_array()`.

Parameters

- **xyxy** (*(N,4) ndarray*) – Array containing the corner coordinates (top-left, bottom-right) of `N` bounding boxes in the form `(x1, y1, x2, y2)`. Should usually be of dtype `float32`.
- **shape** (*tuple of int*) – Shape of the image on which the bounding boxes are placed. Should usually be `(H, W, C)` or `(H, W)`.

Returns Object containing a list of *BoundingBox* instances derived from the provided corner coordinates.

Return type *imgaug.augmentables.bbs.BoundingBoxesOnImage*

height

Get the height of the image on which the bounding boxes fall.

Returns Image height.

Return type int

on(*self*, *image*)

Project bounding boxes from one image (shape) to a another one.

Parameters **image** (*ndarray or tuple of int*) – New image onto which the bounding boxes are to be projected. May also simply be that new image's shape tuple.

Returns Object containing the same bounding boxes after projection to the new image shape.

Return type *imgaug.augmentables.bbs.BoundingBoxesOnImage*

remove_out_of_image(*self*, *fully=True*, *partly=False*)
Remove all BBs that are fully/partially outside of the image.

Parameters

- **fully** (*bool, optional*) – Whether to remove bounding boxes that are fully outside of the image.
- **partly** (*bool, optional*) – Whether to remove bounding boxes that are partially outside of the image.

Returns Reduced set of bounding boxes, with those that were fully/partially outside of the image being removed.

Return type *imgaug.augmentables.bbs.BoundingBoxesOnImage*

shift(*self*, *top=None*, *right=None*, *bottom=None*, *left=None*)
Move all all BBs along the x/y-axis.

Parameters

- **top** (*None or int, optional*) – Amount of pixels by which to shift all objects *from* the top (towards the bottom).
- **right** (*None or int, optional*) – Amount of pixels by which to shift all objects *from* the right (towads the left).
- **bottom** (*None or int, optional*) – Amount of pixels by which to shift all objects *from* the bottom (towards the top).
- **left** (*None or int, optional*) – Amount of pixels by which to shift all objects *from* the left (towards the right).

Returns Shifted bounding boxes.

Return type *imgaug.augmentables.bbs.BoundingBoxesOnImage*

to_xyxy_array(*self*, *dtype=<class 'numpy.float32'>*)
Convert the `BoundingBoxesOnImage` object to an `(N,4) ndarray`.

This is the inverse of `imgaug.BoundingBoxesOnImage.from_xyxy_array()`.

Parameters **dtype** (*numpy.dtype, optional*) – Desired output datatype of the ndarray.

Returns `(N,4) ndarray`, where `N` denotes the number of bounding boxes and `4` denotes the top-left and bottom-right bounding box corner coordinates in form `(x1, y1, x2, y2)`.

Return type ndarray

width
Get the width of the image on which the bounding boxes fall.

Returns Image width.

Return type int

13.9 imgaug.augmentables.heatmaps

class imgaug.augmentables.heatmaps.**HeatmapsOnImage**(*arr*, *shape*, *min_value=0.0*, *max_value=1.0*)

Bases: `object`

Object representing heatmaps on a single image.

Parameters

- **arr** (*(H,W) ndarray or (H,W,C) ndarray*) – Array representing the heatmap(s) on a single image. Multiple heatmaps may be provided, in which case `C` is expected to denote the heatmap index. The array must be of dtype `float32`.
- **shape** (*tuple of int*) – Shape of the image on which the heatmap(s) is/are placed. **Not** the shape of the heatmap(s) array, unless it is identical to the image shape (note the likely difference between the arrays in the number of channels). This is expected to be `(H, W)` or `(H, W, C)` with `C` usually being `3`. If there is no corresponding image, use `(H_arr, W_arr)` instead, where `H_arr` is the height of the heatmap(s) array (analogous `W_arr`).
- **min_value** (*float, optional*) – Minimum value for the heatmaps that *arr* represents. This will usually be `0.0`.
- **max_value** (*float, optional*) – Maximum value for the heatmaps that *arr* represents. This will usually be `1.0`.

Methods

avg_pool(self, block_size)	Average-pool the heatmap(s) array using a given block/kernel size.
change_normalization(arr, source, target)	Change the value range of a heatmap array.
copy(self)	Create a shallow copy of the heatmaps object.
deepcopy(self)	Create a deep copy of the heatmaps object.
draw(self[, size, cmap])	Render the heatmaps as RGB images.
draw_on_image(self, image[, alpha, cmap, resize])	Draw the heatmaps as overlays over an image.
from_0to1(arr_0to1, shape[, min_value, ...])	Create a heatmaps object from a `[0.0, 1.0]` float array.
from_uint8(arr_uint8, shape[, min_value, ...])	Create a `float`-based heatmaps object from an `uint8` array.
get_arr(self)	Get the heatmap's array in value range provided to `__init__()`.
invert(self)	Invert each component in the heatmap.
max_pool(self, block_size)	Max-pool the heatmap(s) array using a given block/kernel size.
pad(self[, top, right, bottom, left, mode, cval])	Pad the heatmaps at their top/right/bottom/left side.
pad_to_aspect_ratio(self, aspect_ratio[, ...])	Pad the heatmaps until they match a target aspect ratio.
resize(self, sizes[, interpolation])	Resize the heatmap(s) array given a target size and interpolation.
scale(self, *args, **kwargs)	**Deprecated**.
to_uint8(self)	Convert this heatmaps object to an `uint8` array.

avg_pool(*self*, *block_size*)

Average-pool the heatmap(s) array using a given block/kernel size.

Parameters **block_size** (*int or tuple of int*) – Size of each block of values to pool, aka kernel size. See *imgaug.imgaug.pool()* for details.

Returns Heatmaps after average pooling.

Return type *imgaug.augmentables.heatmaps.HeatmapsOnImage*

classmethod change_normalization(*arr*, *source*, *target*)
Change the value range of a heatmap array.

E.g. the value range may be changed from the interval `[0.0, 1.0]` to `[-1.0, 1.0]`.

Parameters

- **arr** (*ndarray*) – Heatmap array to modify.
- **source** (*tuple of float*) – Current value range of the input array, given as a tuple `(min, max)`, where both are `float` values.
- **target** (*tuple of float*) – Desired output value range of the array, given as a tuple `(min, max)`, where both are `float` values.

Returns Input array, with value range projected to the desired target value range.

Return type ndarray

copy(*self*)
Create a shallow copy of the heatmaps object.

Returns Shallow copy.

Return type *imgaug.augmentables.heatmaps.HeatmapsOnImage*

deepcopy(*self*)
Create a deep copy of the heatmaps object.

Returns Deep copy.

Return type *imgaug.augmentables.heatmaps.HeatmapsOnImage*

draw(*self*, *size=None*, *cmap='jet'*)
Render the heatmaps as RGB images.

Parameters

- **size** (*None or float or iterable of int or iterable of float, optional*) – Size of the rendered RGB image as `(height, width)`. See *imgaug.imgaug.imresize_single_image()* for details. If set to `None`, no resizing is performed and the size of the heatmaps array is used.
- **cmap** (*str or None, optional*) – Name of the `matplotlib` color map to use when convert the heatmaps to RGB images. If set to `None`, no color map will be used and the heatmaps will be converted to simple intensity maps.

Returns Rendered heatmaps as `uint8` arrays. Always a **list** containing one RGB image per heatmap array channel.

Return type list of (H,W,3) ndarray

draw_on_image(*self*, *image*, *alpha=0.75*, *cmap='jet'*, *resize='heatmaps'*)
Draw the heatmaps as overlays over an image.

Parameters

- **image** (*(H,W,3) ndarray*) – Image onto which to draw the heatmaps. Expected to be of dtype `uint8`.
- **alpha** (*float, optional*) – Alpha/opacity value to use for the mixing of image and heatmaps. Larger values mean that the heatmaps will be more visible and the image less visible.
- **cmap** (*str or None, optional*) – Name of the `matplotlib` color map to use. See *HeatmapsOnImage.draw()* for details.

- **resize** (*{'heatmaps', 'image'}, optional*) – In case of size differences between the image and heatmaps, either the image or the heatmaps can be resized. This parameter controls which of the two will be resized to the other's size.

Returns Rendered overlays as `uint8` arrays. Always a **list** containing one RGB image per heatmap array channel.

Return type list of (H,W,3) ndarray

static from_0to1(*arr_0to1*, *shape*, *min_value=0.0*, *max_value=1.0*)

Create a heatmaps object from a `[0.0, 1.0]` float array.

Parameters

- **arr_0to1** (*(H,W) or (H,W,C) ndarray*) – Heatmap(s) array, where `H` is the height, `W` is the width and `C` is the number of heatmap channels. Expected dtype is `float32`.
- **shape** (*tuple of ints*) – Shape of the image on which the heatmap(s) is/are placed. **Not** the shape of the heatmap(s) array, unless it is identical to the image shape (note the likely difference between the arrays in the number of channels). If there is not a corresponding image, use the shape of the heatmaps array.
- **min_value** (*float, optional*) – Minimum value of the float heatmaps that the input array represents. This will usually be 0.0. In most other cases it will be close to the interval `[0.0, 1.0]`. Calling `imgaug.HeatmapsOnImage.get_arr()`, will automatically convert the interval `[0.0, 1.0]` float array to this `[min, max]` interval.
- **max_value** (*float, optional*) – Minimum value of the float heatmaps that the input array represents. This will usually be 1.0. See parameter *min_value* for details.

Returns Heatmaps object.

Return type *imgaug.augmentables.heatmaps.HeatmapsOnImage*

static from_uint8(*arr_uint8*, *shape*, *min_value=0.0*, *max_value=1.0*)

Create a `float`-based heatmaps object from an `uint8` array.

Parameters

- **arr_uint8** (*(H,W) ndarray or (H,W,C) ndarray*) – Heatmap(s) array, where `H` is height, `W` is width and `C` is the number of heatmap channels. Expected dtype is `uint8`.
- **shape** (*tuple of int*) – Shape of the image on which the heatmap(s) is/are placed. **Not** the shape of the heatmap(s) array, unless it is identical to the image shape (note the likely difference between the arrays in the number of channels). If there is not a corresponding image, use the shape of the heatmaps array.
- **min_value** (*float, optional*) – Minimum value of the float heatmaps that the input array represents. This will usually be 0.0. In most other cases it will be close to the interval `[0.0, 1.0]`. Calling `imgaug.HeatmapsOnImage.get_arr()`, will automatically convert the interval `[0.0, 1.0]` float array to this `[min, max]` interval.
- **max_value** (*float, optional*) – Minimum value of the float heatmaps that the input array represents. This will usually be 1.0. See parameter *min_value* for details.

Returns Heatmaps object.

Return type *imgaug.augmentables.heatmaps.HeatmapsOnImage*

get_arr(*self*)

Get the heatmap's array in value range provided to `__init__()`.

The *HeatmapsOnImage* object saves heatmaps internally in the value range `[0.0, 1.0]`. This function converts the internal representation to `[min, max]`, where `min` and `max` are provided to `HeatmapsOnImage.__init__()` upon instantiation of the object.

Returns Heatmap array of dtype `float32`.

Return type (H,W) ndarray or (H,W,C) ndarray

invert (*self*)

Invert each component in the heatmap.

This shifts low values towards high values and vice versa.

This changes each value to:

```
v' = max - (v - min)
```

where `v` is the value at a spatial location, `min` is the minimum value in the heatmap and `max` is the maximum value. As the heatmap uses internally a `0.0` to `1.0` representation, this simply becomes `v' = 1.0 - v`.

This function can be useful e.g. when working with depth maps, where algorithms might have an easier time representing the furthest away points with zeros, requiring an inverted depth map.

Returns Inverted heatmap.

Return type *imgaug.augmentables.heatmaps.HeatmapsOnImage*

max_pool (*self*, *block_size*)

Max-pool the heatmap(s) array using a given block/kernel size.

Parameters **block_size** (*int or tuple of int*) – Size of each block of values to pool, aka kernel size. See *imgaug.imgaug.pool()* for details.

Returns Heatmaps after max-pooling.

Return type *imgaug.augmentables.heatmaps.HeatmapsOnImage*

pad (*self*, *top=0*, *right=0*, *bottom=0*, *left=0*, *mode='constant'*, *cval=0.0*)

Pad the heatmaps at their top/right/bottom/left side.

Parameters

- **top** (*int, optional*) – Amount of pixels to add at the top side of the heatmaps. Must be `0` or greater.
- **right** (*int, optional*) – Amount of pixels to add at the right side of the heatmaps. Must be `0` or greater.
- **bottom** (*int, optional*) – Amount of pixels to add at the bottom side of the heatmaps. Must be `0` or greater.
- **left** (*int, optional*) – Amount of pixels to add at the left side of the heatmaps. Must be `0` or greater.
- **mode** (*string, optional*) – Padding mode to use. See *imgaug.imgaug.pad()* for details.
- **cval** (*number, optional*) – Value to use for padding *mode* is `constant`. See *imgaug.imgaug.pad()* for details.

Returns Padded heatmaps of height `H'=H+top+bottom` and width `W'=W+left+right`.

Return type *imgaug.augmentables.heatmaps.HeatmapsOnImage*

pad_to_aspect_ratio(*self*, *aspect_ratio*, *mode='constant'*, *cval=0.0*, *return_pad_amounts=False*)

Pad the heatmaps until they match a target aspect ratio.

Depending on which dimension is smaller (height or width), only the corresponding sides (left/right or top/bottom) will be padded. In each case, both of the sides will be padded equally.

Parameters

- **aspect_ratio** (*float*) – Target aspect ratio, given as width/height. E.g. `2.0` denotes the image having twice as much width as height.
- **mode** (*str, optional*) – Padding mode to use. See *imgaug.imgaug.pad()* for details.
- **cval** (*number, optional*) – Value to use for padding if *mode* is `constant`. See *imgaug.imgaug.pad()* for details.
- **return_pad_amounts** (*bool, optional*) – If `False`, then only the padded instance will be returned. If `True`, a tuple with two entries will be returned, where the first entry is the padded instance and the second entry are the amounts by which each array side was padded. These amounts are again a tuple of the form `(top, right, bottom, left)`, with each value being an integer.

Returns

- *imgaug.augmentables.heatmaps.HeatmapsOnImage* – Padded heatmaps as *HeatmapsOnImage* instance.
- *tuple of int* – Amounts by which the instance's array was padded on each side, given as a tuple `(top, right, bottom, left)`. This tuple is only returned if *return_pad_amounts* was set to `True`.

resize(*self*, *sizes*, *interpolation='cubic'*)

Resize the heatmap(s) array given a target size and interpolation.

Parameters

- **sizes** (*float or iterable of int or iterable of float*) – New size of the array in `(height, width)`. See *imgaug.imgaug.imresize_single_image()* for details.
- **interpolation** (*None or str or int, optional*) – The interpolation to use during resize. See *imgaug.imgaug.imresize_single_image()* for details.

Returns Resized heatmaps object.

Return type *imgaug.augmentables.heatmaps.HeatmapsOnImage*

scale(*self*, **args*, ***kwargs*)

Deprecated. Use `HeatmapsOnImage.resize()` instead. resize() has the exactly same interface.

to_uint8(*self*)

Convert this heatmaps object to an `uint8` array.

Returns Heatmap as an `uint8` array, i.e. with the discrete value range `[0, 255]`.

Return type (H,W,C) ndarray

13.10 imgaug.augmentables.kps

class imgaug.augmentables.kps.**Keypoint**(*x*, *y*)

Bases: `object`

A single keypoint (aka landmark) on an image.

Parameters

- **x** (*number*) – Coordinate of the keypoint on the x axis.
- **y** (*number*) – Coordinate of the keypoint on the y axis.

Attributes

x_int	Get the keypoint's x-coordinate, rounded to the closest integer.
y_int	Get the keypoint's y-coordinate, rounded to the closest integer.

Methods

copy(self[, x, y])	Create a shallow copy of the keypoint instance.
deepcopy(self[, x, y])	Create a deep copy of the keypoint instance.
draw_on_image(self, image[, color, alpha, ...])	Draw the keypoint onto a given image.
generate_similar_points_manhattan(self, ...)	Generate nearby points based on manhattan distance.
project(self, from_shape, to_shape)	Project the keypoint onto a new position on a new image.
shift(self[, x, y])	Move the keypoint around on an image.

copy (*self*, *x=None*, *y=None*)
Create a shallow copy of the keypoint instance.

Parameters

- **x** (*None or number, optional*) – Coordinate of the keypoint on the x axis. If `None`, the instance's value will be copied.
- **y** (*None or number, optional*) – Coordinate of the keypoint on the y axis. If `None`, the instance's value will be copied.

Returns Shallow copy.

Return type *imgaug.augmentables.kps.Keypoint*

deepcopy (*self*, *x=None*, *y=None*)
Create a deep copy of the keypoint instance.

Parameters

- **x** (*None or number, optional*) – Coordinate of the keypoint on the x axis. If `None`, the instance's value will be copied.
- **y** (*None or number, optional*) – Coordinate of the keypoint on the y axis. If `None`, the instance's value will be copied.

Returns Deep copy.

Return type *imgaug.augmentables.kps.Keypoint*

draw_on_image (*self*, *image*, *color=(0, 255, 0)*, *alpha=1.0*, *size=3*, *copy=True*, *raise_if_out_of_image=False*)
Draw the keypoint onto a given image.

The keypoint is drawn as a square.

Parameters

- **image** (*(H,W,3) ndarray*) – The image onto which to draw the keypoint.
- **color** (*int or list of int or tuple of int or (3,) ndarray, optional*) – The RGB color of the keypoint. If a single `int C`, then that is equivalent to `(C,C,C)`.
- **alpha** (*float, optional*) – The opacity of the drawn keypoint, where `1.0` denotes a fully visible keypoint and `0.0` an invisible one.
- **size** (*int, optional*) – The size of the keypoint. If set to `S`, each square will have size `S x S`.
- **copy** (*bool, optional*) – Whether to copy the image before drawing the keypoint.
- **raise_if_out_of_image** (*bool, optional*) – Whether to raise an exception if the keypoint is outside of the image.

Returns **image** – Image with drawn keypoint.

Return type (H,W,3) ndarray

generate_similar_points_manhattan(*self*, *nb_steps*, *step_size*, *return_array=False*)

Generate nearby points based on manhattan distance.

To generate the first neighbouring points, a distance of `S` (step size) is moved from the center point (this keypoint) to the top, right, bottom and left, resulting in four new points. From these new points, the pattern is repeated. Overlapping points are ignored.

The resulting points have a shape similar to a square rotated by `45` degrees.

Parameters

- **nb_steps** (*int*) – The number of steps to move from the center point. `nb_steps=1` results in a total of `5` output points (one center point + four neighbours).
- **step_size** (*number*) – The step size to move from every point to its neighbours.
- **return_array** (*bool, optional*) – Whether to return the generated points as a list of *`Keypoint`* or an array of shape `(N,2)`, where `N` is the number of generated points and the second axis contains the x-/y-coordinates.

Returns If *return_array* was `False`, then a list of *`Keypoint`*. Otherwise a numpy array of shape `(N,2)`, where `N` is the number of generated points and the second axis contains the x-/y-coordinates. The center keypoint (the one on which this function was called) is always included.

Return type list of imgaug.augmentables.kps.Keypoint or (N,2) ndarray

project(*self*, *from_shape*, *to_shape*)

Project the keypoint onto a new position on a new image.

E.g. if the keypoint is on its original image at `x=(10 of 100 pixels)` and `y=(20 of 100 pixels)` and is projected onto a new image with size `(width=200, height=200)`, its new position will be `(20, 40)`.

This is intended for cases where the original image is resized. It cannot be used for more complex changes (e.g. padding, cropping).

Parameters

- **from_shape** (*tuple of int*) – Shape of the original image. (Before resize.)
- **to_shape** (*tuple of int*) – Shape of the new image. (After resize.)

Returns Keypoint object with new coordinates.

Return type *imgaug.augmentables.kps.Keypoint*

shift (*self*, *x=0*, *y=0*)
Move the keypoint around on an image.

Parameters

- **x** (*number, optional*) – Move by this value on the x axis.
- **y** (*number, optional*) – Move by this value on the y axis.

Returns Keypoint object with new coordinates.

Return type *imgaug.augmentables.kps.Keypoint*

x_int
Get the keypoint's x-coordinate, rounded to the closest integer.

Returns **result** – Keypoint's x-coordinate, rounded to the closest integer.

Return type int

y_int
Get the keypoint's y-coordinate, rounded to the closest integer.

Returns **result** – Keypoint's y-coordinate, rounded to the closest integer.

Return type int

class imgaug.augmentables.kps.**KeypointsOnImage**(*keypoints*, *shape*)
Bases: object

Container for all keypoints on a single image.

Parameters

- **keypoints** (*list of imgaug.augmentables.kps.Keypoint*) – List of keypoints on the image.
- **shape** (*tuple of int*) – The shape of the image on which the keypoints are placed.

Examples

```
>>> import numpy as np
>>> from imgaug.augmentables.kps import Keypoint, KeypointsOnImage
>>>
>>> image = np.zeros((70, 70))
>>> kps = [Keypoint(x=10, y=20), Keypoint(x=34, y=60)]
>>> kps_oi = KeypointsOnImage(kps, shape=image.shape)
```

Attributes

empty Determine whether this object contains zero keypoints.

height

width

Methods

copy(self[, keypoints, shape])	Create a shallow copy of the KeypointsOnImage object.

Continued on next page

Table 42 – continued from previous page

deepcopy(self[, keypoints, shape])	Create a deep copy of the `KeypointsOnImage` object.
draw_on_image(self, image[, color, alpha, ...])	Draw all keypoints onto a given image.
from_coords_array(coords, shape)	**Deprecated**.
from_distance_maps(distance_maps[, ...])	Convert outputs of `to_distance_maps()` to `KeypointsOnImage`.
from_keypoint_image(image[, ...])	Convert `to_keypoint_image()` outputs to `KeypointsOnImage`.
from_xy_array(xy, shape)	Convert an `(N,2)` array to a `KeypointsOnImage` object.
get_coords_array(self)	**Deprecated**.
on(self, image)	Project all keypoints from one image shape to a new one.
shift(self[, x, y])	Move the keypoints on the x/y-axis.
to_distance_maps(self[, inverted])	Generate a `(H,W,N)` array of distance maps for `N` keypoints.
to_keypoint_image(self[, size])	Create an `(H,W,N)` image with keypoint coordinates set to `255`.
to_xy_array(self)	Convert all keypoint coordinates to an array of shape `(N,2)`.

copy (*self*, *keypoints=None*, *shape=None*)
Create a shallow copy of the `KeypointsOnImage` object.

Parameters

- **keypoints** (*None or list of imgaug.Keypoint, optional*) – List of keypoints on the image. If `None`, the instance's keypoints will be copied.
- **shape** (*tuple of int, optional*) – The shape of the image on which the keypoints are placed. If `None`, the instance's shape will be copied.

Returns Shallow copy.

Return type *imgaug.augmentables.kps.KeypointsOnImage*

deepcopy (*self*, *keypoints=None*, *shape=None*)
Create a deep copy of the `KeypointsOnImage` object.

Parameters

- **keypoints** (*None or list of imgaug.Keypoint, optional*) – List of keypoints on the image. If `None`, the instance's keypoints will be copied.
- **shape** (*tuple of int, optional*) – The shape of the image on which the keypoints are placed. If `None`, the instance's shape will be copied.

Returns Deep copy.

Return type *imgaug.augmentables.kps.KeypointsOnImage*

draw_on_image (*self*, *image*, *color=(0, 255, 0)*, *alpha=1.0*, *size=3*, *copy=True*, *raise_if_out_of_image=False*)
Draw all keypoints onto a given image.

Each keypoint is drawn as a square of provided color and size.

Parameters

- **image** (*(H,W,3) ndarray*) – The image onto which to draw the keypoints. This image should usually have the same shape as set in `KeypointsOnImage.shape`.
- **color** (*int or list of int or tuple of int or (3,) ndarray, optional*) – The RGB color of all keypoints. If a single `int C`, then that is equivalent to `(C,C,C)`.
- **alpha** (*float, optional*) – The opacity of the drawn keypoint, where `1.0` denotes a fully visible keypoint and `0.0` an invisible one.
- **size** (*int, optional*) – The size of each point. If set to `C`, each square will have size `C x C`.
- **copy** (*bool, optional*) – Whether to copy the image before drawing the points.
- **raise_if_out_of_image** (*bool, optional*) – Whether to raise an exception if any keypoint is outside of the image.

Returns Image with drawn keypoints.

Return type (H,W,3) ndarray

empty

Determine whether this object contains zero keypoints.

Returns `True` if this object contains zero keypoints.

Return type bool

static from_coords_array(*coords*, *shape*)

Deprecated. Use `KeypointsOnImage.from_xy_array()` instead.

Convert an `(N,2)` array to a `KeypointsOnImage` object.

Parameters

coords [(N, 2) ndarray]

Coordinates of `N` keypoints on an image, given as a `(N,2)` array of xy-coordinates.

shape [tuple] The shape of the image on which the keypoints are placed.

Returns

imgaug.augmentables.kps.KeypointsOnImage *KeypointsOnImage* object containing the array's keypoints.

static from_distance_maps(*distance_maps*, *inverted=False*, *if_not_found_coords={'x': -1, 'y': -1}*, *threshold=None*, *nb_channels=None*)

Convert outputs of `to_distance_maps()` to `KeypointsOnImage`.

This is the inverse of *KeypointsOnImage.to_distance_maps()*.

Parameters

- **distance_maps** (*(H,W,N) ndarray*) – The distance maps. `N` is the number of keypoints.
- **inverted** (*bool, optional*) – Whether the given distance maps were generated in inverted mode (i.e. *KeypointsOnImage.to_distance_maps()* was called with `inverted=True`) or in non-inverted mode.
- **if_not_found_coords** (*tuple or list or dict or None, optional*) – Coordinates to use for keypoints that cannot be found in *distance_maps*.
  - If this is a `list`/`tuple`, it must contain two `int` values.
  - If it is a `dict`, it must contain the keys `x` and `y` with each containing one `int` value.

- If this is `None`, then the keypoint will not be added to the final *KeypointsOnImage* object.

- **threshold** (*float, optional*) – The search for keypoints works by searching for the argmin (non-inverted) or argmax (inverted) in each channel. This parameters contains the maximum (non-inverted) or minimum (inverted) value to accept in order to view a hit as a keypoint. Use `None` to use no min/max.

- **nb_channels** (*None or int, optional*) – Number of channels of the image on which the keypoints are placed. Some keypoint augmenters require that information. If set to `None`, the keypoint's shape will be set to `(height, width)`, otherwise `(height, width, nb_channels)`.

Returns The extracted keypoints.

Return type *imgaug.augmentables.kps.KeypointsOnImage*

static from_keypoint_image(*image*, *if_not_found_coords={'x': -1, 'y': -1}*, *threshold=1*, *nb_channels=None*)

Convert `to_keypoint_image()` outputs to `KeypointsOnImage`.

This is the inverse of *KeypointsOnImage.to_keypoint_image()*.

Parameters

- **image** (*(H,W,N) ndarray*) – The keypoints image. N is the number of keypoints.

- **if_not_found_coords** (*tuple or list or dict or None, optional*) – Coordinates to use for keypoints that cannot be found in *image*.

  - If this is a `list`/`tuple`, it must contain two `int` values.

  - If it is a `dict`, it must contain the keys `x` and `y` with each containing one `int` value.

  - If this is `None`, then the keypoint will not be added to the final *KeypointsOnImage* object.

- **threshold** (*int, optional*) – The search for keypoints works by searching for the argmax in each channel. This parameters contains the minimum value that the max must have in order to be viewed as a keypoint.

- **nb_channels** (*None or int, optional*) – Number of channels of the image on which the keypoints are placed. Some keypoint augmenters require that information. If set to `None`, the keypoint's shape will be set to `(height, width)`, otherwise `(height, width, nb_channels)`.

Returns The extracted keypoints.

Return type *imgaug.augmentables.kps.KeypointsOnImage*

classmethod from_xy_array(*xy*, *shape*)

Convert an `(N,2)` array to a `KeypointsOnImage` object.

Parameters

- **xy** (*(N, 2) ndarray*) – Coordinates of `N` keypoints on an image, given as a `(N,2)` array of xy-coordinates.

- **shape** (*tuple of int or ndarray*) – The shape of the image on which the keypoints are placed.

Returns *KeypointsOnImage* object containing the array's keypoints.

Return type *imgaug.augmentables.kps.KeypointsOnImage*

get_coords_array(*self*)
Deprecated. Use `KeypointsOnImage.to_xy_array()` instead.

Convert all keypoint coordinates to an array of shape `(N,2)`.

Returns

(N, 2) ndarray Array containing the coordinates of all keypoints. `N` denotes the number of keypoints. The second axis denotes the x/y-coordinates.

height

on(*self*, *image*)
Project all keypoints from one image shape to a new one.

Parameters **image** (*ndarray or tuple of int*) – New image onto which the keypoints are to be projected. May also simply be that new image's shape tuple.

Returns Object containing all projected keypoints.

Return type *imgaug.augmentables.kps.KeypointsOnImage*

shift(*self*, *x=0*, *y=0*)
Move the keypoints on the x/y-axis.

Parameters

- **x** (*number, optional*) – Move each keypoint by this value on the x axis.
- **y** (*number, optional*) – Move each keypoint by this value on the y axis.

Returns Keypoints after moving them.

Return type *imgaug.augmentables.kps.KeypointsOnImage*

to_distance_maps(*self*, *inverted=False*)
Generate a `(H,W,N)` array of distance maps for `N` keypoints.

The `n`-th distance map contains at every location `(y, x)` the euclidean distance to the `n`-th keypoint.

This function can be used as a helper when augmenting keypoints with a method that only supports the augmentation of images.

Parameters **inverted** (*bool, optional*) – If `True`, inverted distance maps are returned where each distance value `d` is replaced by `d/(d+1)`, i.e. the distance maps have values in the range `(0.0, 1.0]` with `1.0` denoting exactly the position of the respective keypoint.

Returns A `float32` array containing `N` distance maps for `N` keypoints. Each location `(y, x, n)` in the array denotes the euclidean distance at `(y, x)` to the `n`-th keypoint. If *inverted* is `True`, the distance `d` is replaced by `d/(d+1)`. The height and width of the array match the height and width in `KeypointsOnImage.shape`.

Return type (H,W,N) ndarray

to_keypoint_image(*self*, *size=1*)
Create an `(H,W,N)` image with keypoint coordinates set to `255`.

This method generates a new `uint8` array of shape `(H,W,N)`, where `H` is the `.shape` height, `W` the `.shape` width and `N` is the number of keypoints. The array is filled with zeros. The coordinate of the `n`-th keypoint is set to `255` in the `n`-th channel.

This function can be used as a helper when augmenting keypoints with a method that only supports the augmentation of images.

Parameters **size** (*int*) – Size of each (squared) point.

Returns Image in which the keypoints are marked. `H` is the height, defined in `KeypointsOnImage.shape[0]` (analogous `W`). `N` is the number of keypoints.

Return type (H,W,N) ndarray

to_xy_array(*self*)
Convert all keypoint coordinates to an array of shape `(N, 2)`.

Returns Array containing the coordinates of all keypoints. `N` denotes the number of keypoints. The second axis denotes the x/y-coordinates.

Return type (N, 2) ndarray

width

imgaug.augmentables.kps.**compute_geometric_median**(*points=None*, *eps=1e-05*, *X=None*)
Estimate the geometric median of points in 2D.

Code from https://stackoverflow.com/a/30305181

Parameters

- **points** (*(N,2) ndarray*) – Points in 2D. Second axis must be given in xy-form.
- **eps** (*float, optional*) – Distance threshold when to return the median.
- **X** (*None or (N,2) ndarray, optional*) – Deprecated.

Returns Geometric median as xy-coordinate.

Return type (2,) ndarray

13.11 imgaug.augmentables.lines

class imgaug.augmentables.lines.**LineString**(*coords*, *label=None*)
Bases: `object`

Class representing line strings.

A line string is a collection of connected line segments, each having a start and end point. Each point is given as its `(x, y)` absolute (sub-)pixel coordinates. The end point of each segment is also the start point of the next segment.

The line string is not closed, i.e. start and end point are expected to differ and will not be connected in drawings.

Parameters

- **coords** (*iterable of tuple of number or ndarray*) – The points of the line string.
- **label** (*None or str, optional*) – The label of the line string.

Attributes

height Compute the height of a bounding box encapsulating the line.

length Compute the total euclidean length of the line string.

width Compute the width of a bounding box encapsulating the line.

xx Get an array of x-coordinates of all points of the line string.

xx_int Get an array of discrete x-coordinates of all points.

yy Get an array of y-coordinates of all points of the line string.

yy_int Get an array of discrete y-coordinates of all points.

Methods

almost_equals(self, other[, max_distance, ...])	Compare this and another line string.
clip_out_of_image(self, image)	Clip off all parts of the line string that are outside of the image.
compute_distance(self, other[, default])	Compute the minimal distance between the line string and *other*.
compute_neighbour_distances(self)	Compute the euclidean distance between each two consecutive points.
compute_pointwise_distances(self, other[, ...])	Compute min distances between points of this and another line string.
concatenate(self, other)	Concatenate this line string with another one.
contains(self, other[, max_distance])	Estimate whether a point is on this line string.
coords_almost_equals(self, other[, ...])	Compare this and another LineString's coordinates.
copy(self[, coords, label])	Create a shallow copy of this line string.
deepcopy(self[, coords, label])	Create a deep copy of this line string.
draw_heatmap_array(self, image_shape[, ...])	Draw the line segments and points of the line string as a heatmap array.
draw_lines_heatmap_array(self, image_shape)	Draw the line segments of this line string as a heatmap array.
draw_lines_on_image(self, image[, color, ...])	Draw the line segments of this line string on a given image.
draw_mask(self, image_shape[, size_lines, ...])	Draw this line segment as a binary image mask.
draw_on_image(self, image[, color, ...])	Draw this line string onto an image.
draw_points_heatmap_array(self, image_shape)	Draw the points of this line string as a heatmap array.
draw_points_on_image(self, image[, color, ...])	Draw the points of this line string onto a given image.
extract_from_image(self, image[, size, pad, ...])	Extract all image pixels covered by the line string.
find_intersections_with(self, other)	Find all intersection points between this line string and *other*.
get_pointwise_inside_image_mask(self, image)	Determine per point whether it is inside of a given image plane.
is_fully_within_image(self, image[, default])	Estimate whether the line string is fully inside an image plane.
is_out_of_image(self, image[, fully, ...])	Estimate whether the line is partially/fully outside of the image area.
is_partly_within_image(self, image[, default])	Estimate whether the line string is at least partially inside the image.
project(self, from_shape, to_shape)	Project the line string onto a differently shaped image.
shift(self[, top, right, bottom, left])	Move this line string along the x/y-axis.
subdivide(self, points_per_edge)	Derive a new line string with `N` interpolated points per edge.
to_bounding_box(self)	Generate a bounding box encapsulating the line string.
to_heatmap(self, image_shape[, size_lines, ...])	Generate a heatmap object from the line string.
to_keypoints(self)	Convert the line string points to keypoints.
to_polygon(self)	Generate a polygon from the line string points.

Continued on next page

Table 43 – continued from previous page

to_segmentation_map(self, image_shape[, ...])	Generate a segmentation map object from the line string.

almost_equals (*self*, *other*, *max_distance=0.0001*, *points_per_edge=8*)
Compare this and another line string.

Parameters

- **other** (*imgaug.augmentables.lines.LineString*) – The other line string. Must be a LineString instance, not just its coordinates.
- **max_distance** (*float, optional*) – See *imgaug.augmentables.lines.LineString.coords_almost_equals()*.
- **points_per_edge** (*int, optional*) – See *imgaug.augmentables.lines.LineString.coords_almost_equals()*.

Returns `True` if the coordinates are almost equal according to *imgaug.augmentables.lines.LineString.coords_almost_equals()* and additionally the labels are equal. Otherwise `False`.

Return type bool

clip_out_of_image (*self*, *image*)
Clip off all parts of the line string that are outside of the image.

Parameters **image** (*ndarray or tuple of int*) – Either an image with shape `(H,W,[C])` or a `tuple` denoting such an image shape.

Returns Line strings, clipped to the image shape. The result may contain any number of line strins, including zero.

Return type list of imgaug.augmentables.lines.LineString

compute_distance (*self*, *other*, *default=None*)
Compute the minimal distance between the line string and *other*.

Parameters

- **other** (*tuple of number or imgaug.augmentables.kps.Keypoint or imgaug.augmentables.LineString*) – Other object to which to compute the distance.
- **default** (*any*) – Value to return if this line string or *other* contain no points.

Returns Minimal distance to *other* or *default* if no distance could be computed.

Return type float or any

compute_neighbour_distances (*self*)
Compute the euclidean distance between each two consecutive points.

Returns `(N-1,) float32` array of euclidean distances between point pairs. Same order as in *coords*.

Return type ndarray

compute_pointwise_distances (*self*, *other*, *default=None*)
Compute min distances between points of this and another line string.

Parameters

- **other** (*tuple of number or imgaug.augmentables.kps.Keypoint or imgaug.augmentables.LineString*) – Other object to which to compute the distances.

- **default** (*any*) – Value to return if *other* contains no points.

Returns For each coordinate of this line string, the distance to any closest location on *other*. *default* if no distance could be computed.

Return type list of float or any

concatenate(*self*, *other*)
Concatenate this line string with another one.

This will add a line segment between the end point of this line string and the start point of *other*.

Parameters **other** (*imgaug.augmentables.lines.LineString or ndarray or iterable of tuple of number*) – The points to add to this line string.

Returns New line string with concatenated points. The *label* of this line string will be kept.

Return type *imgaug.augmentables.lines.LineString*

contains(*self*, *other*, *max_distance=0.0001*)
Estimate whether a point is on this line string.

This method uses a maximum distance to estimate whether a point is on a line string.

Parameters

- **other** (*tuple of number or imgaug.augmentables.kps.Keypoint*) – Point to check for.
- **max_distance** (*float*) – Maximum allowed euclidean distance between the point and the closest point on the line. If the threshold is exceeded, the point is not considered to fall on the line.

Returns `True` if the point is on the line string, `False` otherwise.

Return type bool

coords_almost_equals(*self*, *other*, *max_distance=0.0001*, *points_per_edge=8*)
Compare this and another LineString's coordinates.

This is an approximate method based on pointwise distances and can in rare corner cases produce wrong outputs.

Parameters

- **other** (*imgaug.augmentables.lines.LineString or tuple of number or ndarray or list of ndarray or list of tuple of number*) – The other line string or its coordinates.
- **max_distance** (*float, optional*) – Max distance of any point from the other line string before the two line strings are evaluated to be unequal.
- **points_per_edge** (*int, optional*) – How many points to interpolate on each edge.

Returns Whether the two LineString's coordinates are almost identical, i.e. the max distance is below the threshold. If both have no coordinates, `True` is returned. If only one has no coordinates, `False` is returned. Beyond that, the number of points is not evaluated.

Return type bool

copy(*self*, *coords=None*, *label=None*)
Create a shallow copy of this line string.

Parameters

- **coords** (*None or iterable of tuple of number or ndarray*) – If not `None`, then the coords of the copied object will be set to this value.

- **label** (*None or str*) – If not `None`, then the label of the copied object will be set to this value.

Returns Shallow copy.

Return type *imgaug.augmentables.lines.LineString*

deepcopy(*self*, *coords=None*, *label=None*)
Create a deep copy of this line string.

Parameters

- **coords** (*None or iterable of tuple of number or ndarray*) – If not `None`, then the coords of the copied object will be set to this value.
- **label** (*None or str*) – If not `None`, then the label of the copied object will be set to this value.

Returns Deep copy.

Return type *imgaug.augmentables.lines.LineString*

draw_heatmap_array(*self*, *image_shape*, *alpha_lines=1.0*, *alpha_points=1.0*, *size_lines=1*, *size_points=0*, *antialiased=True*, *raise_if_out_of_image=False*)
Draw the line segments and points of the line string as a heatmap array.

Parameters

- **image_shape** (*tuple of int*) – The shape of the image onto which to draw the line mask.
- **alpha_lines** (*float, optional*) – Opacity of the line string. Higher values denote a more visible line string.
- **alpha_points** (*float, optional*) – Opacity of the line string points. Higher values denote a more visible points.
- **size_lines** (*int, optional*) – Thickness of the line segments.
- **size_points** (*int, optional*) – Size of the points in pixels.
- **antialiased** (*bool, optional*) – Whether to draw the line with anti-aliasing activated.
- **raise_if_out_of_image** (*bool, optional*) – Whether to raise an error if the line string is fully outside of the image. If set to `False`, no error will be raised and only the parts inside the image will be drawn.

Returns `float32` array of shape *image_shape* (no channel axis) with drawn line segments and points. All values are in the interval `[0.0, 1.0]`.

Return type ndarray

draw_lines_heatmap_array(*self*, *image_shape*, *alpha=1.0*, *size=1*, *antialiased=True*, *raise_if_out_of_image=False*)
Draw the line segments of this line string as a heatmap array.

Parameters

- **image_shape** (*tuple of int*) – The shape of the image onto which to draw the line mask.
- **alpha** (*float, optional*) – Opacity of the line string. Higher values denote a more visible line string.
- **size** (*int, optional*) – Thickness of the line segments.
- **antialiased** (*bool, optional*) – Whether to draw the line with anti-aliasing activated.

- **raise_if_out_of_image** (*bool, optional*) – Whether to raise an error if the line string is fully outside of the image. If set to `False`, no error will be raised and only the parts inside the image will be drawn.

Returns `float32` array of shape *image_shape* (no channel axis) with drawn line string. All values are in the interval `[0.0, 1.0]`.

Return type ndarray

draw_lines_on_image(*self*, *image*, *color=(0, 255, 0)*, *alpha=1.0*, *size=3*, *antialiased=True*, *raise_if_out_of_image=False*)

Draw the line segments of this line string on a given image.

Parameters

- **image** (*ndarray or tuple of int*) – The image onto which to draw. Expected to be `uint8` and of shape `(H, W, C)` with `C` usually being `3` (other values are not tested). If a tuple, expected to be `(H, W, C)` and will lead to a new `uint8` array of zeros being created.
- **color** (*int or iterable of int*) – Color to use as RGB, i.e. three values.
- **alpha** (*float, optional*) – Opacity of the line string. Higher values denote a more visible line string.
- **size** (*int, optional*) – Thickness of the line segments.
- **antialiased** (*bool, optional*) – Whether to draw the line with anti-aliasing activated.
- **raise_if_out_of_image** (*bool, optional*) – Whether to raise an error if the line string is fully outside of the image. If set to `False`, no error will be raised and only the parts inside the image will be drawn.

Returns *image* with line drawn on it.

Return type ndarray

draw_mask(*self*, *image_shape*, *size_lines=1*, *size_points=0*, *raise_if_out_of_image=False*)

Draw this line segment as a binary image mask.

Parameters

- **image_shape** (*tuple of int*) – The shape of the image onto which to draw the line mask.
- **size_lines** (*int, optional*) – Thickness of the line segments.
- **size_points** (*int, optional*) – Size of the points in pixels.
- **raise_if_out_of_image** (*bool, optional*) – Whether to raise an error if the line string is fully outside of the image. If set to `False`, no error will be raised and only the parts inside the image will be drawn.

Returns Boolean line mask of shape *image_shape* (no channel axis).

Return type ndarray

draw_on_image(*self*, *image*, *color=(0, 255, 0)*, *color_lines=None*, *color_points=None*, *alpha=1.0*, *alpha_lines=None*, *alpha_points=None*, *size=1*, *size_lines=None*, *size_points=None*, *antialiased=True*, *raise_if_out_of_image=False*)

Draw this line string onto an image.

Parameters

- **image** (*ndarray*) – The *(H,W,C) uint8* image onto which to draw the line string.
- **color** (*iterable of int, optional*) – Color to use as RGB, i.e. three values. The color of the line and points are derived from this value, unless they are set.

- **color_lines** (*None or iterable of int*) – Color to use for the line segments as RGB, i.e. three values. If `None`, this value is derived from *color*.
- **color_points** (*None or iterable of int*) – Color to use for the points as RGB, i.e. three values. If `None`, this value is derived from `0.5 * color`.
- **alpha** (*float, optional*) – Opacity of the line string. Higher values denote more visible points. The alphas of the line and points are derived from this value, unless they are set.
- **alpha_lines** (*None or float, optional*) – Opacity of the line string. Higher values denote more visible line string. If `None`, this value is derived from *alpha*.
- **alpha_points** (*None or float, optional*) – Opacity of the line string points. Higher values denote more visible points. If `None`, this value is derived from *alpha*.
- **size** (*int, optional*) – Size of the line string. The sizes of the line and points are derived from this value, unless they are set.
- **size_lines** (*None or int, optional*) – Thickness of the line segments. If `None`, this value is derived from *size*.
- **size_points** (*None or int, optional*) – Size of the points in pixels. If `None`, this value is derived from `3 * size`.
- **antialiased** (*bool, optional*) – Whether to draw the line with anti-aliasing activated. This does currently not affect the point drawing.
- **raise_if_out_of_image** (*bool, optional*) – Whether to raise an error if the line string is fully outside of the image. If set to `False`, no error will be raised and only the parts inside the image will be drawn.

Returns Image with line string drawn on it.

Return type ndarray

draw_points_heatmap_array(*self*, *image_shape*, *alpha=1.0*, *size=1*, *raise_if_out_of_image=False*)

Draw the points of this line string as a heatmap array.

Parameters

- **image_shape** (*tuple of int*) – The shape of the image onto which to draw the point mask.
- **alpha** (*float, optional*) – Opacity of the line string points. Higher values denote a more visible points.
- **size** (*int, optional*) – Size of the points in pixels.
- **raise_if_out_of_image** (*bool, optional*) – Whether to raise an error if the line string is fully outside of the image. If set to `False`, no error will be raised and only the parts inside the image will be drawn.

Returns `float32` array of shape *image_shape* (no channel axis) with drawn line string points. All values are in the interval `[0.0, 1.0]`.

Return type ndarray

draw_points_on_image(*self*, *image*, *color=(0, 128, 0)*, *alpha=1.0*, *size=3*, *copy=True*, *raise_if_out_of_image=False*)

Draw the points of this line string onto a given image.

Parameters

- **image** (*ndarray or tuple of int*) – The image onto which to draw. Expected to be `uint8` and of shape `(H, W, C)` with `C` usually being `3` (other values are not tested). If a tuple, expected to be `(H, W, C)` and will lead to a new `uint8` array of zeros being created.
- **color** (*iterable of int*) – Color to use as RGB, i.e. three values.
- **alpha** (*float, optional*) – Opacity of the line string points. Higher values denote a more visible points.
- **size** (*int, optional*) – Size of the points in pixels.
- **copy** (*bool, optional*) – Whether it is allowed to draw directly in the input array (`False`) or it has to be copied (`True`). The routine may still have to copy, even if `copy=False` was used. Always use the return value.
- **raise_if_out_of_image** (*bool, optional*) – Whether to raise an error if the line string is fully outside of the image. If set to `False`, no error will be raised and only the parts inside the image will be drawn.

Returns `float32` array of shape *image_shape* (no channel axis) with drawn line string points. All values are in the interval `[0.0, 1.0]`.

Return type ndarray

extract_from_image(*self*, *image*, *size=1*, *pad=True*, *pad_max=None*, *antialiased=True*, *prevent_zero_size=True*)

Extract all image pixels covered by the line string.

This will only extract pixels overlapping with the line string. As a rectangular image array has to be returned, non-overlapping pixels will be set to zero.

This function will by default zero-pad the image if the line string is partially/fully outside of the image. This is for consistency with the same methods for bounding boxes and polygons.

Parameters

- **image** (*ndarray*) – The image of shape *(H,W,[C])* from which to extract the pixels within the line string.
- **size** (*int, optional*) – Thickness of the line.
- **pad** (*bool, optional*) – Whether to zero-pad the image if the object is partially/fully outside of it.
- **pad_max** (*None or int, optional*) – The maximum number of pixels that may be zero-paded on any side, i.e. if this has value `N` the total maximum of added pixels is `4*N`. This option exists to prevent extremely large images as a result of single points being moved very far away during augmentation.
- **antialiased** (*bool, optional*) – Whether to apply anti-aliasing to the line string.
- **prevent_zero_size** (*bool, optional*) – Whether to prevent height or width of the extracted image from becoming zero. If this is set to `True` and height or width of the line string is below `1`, the height/width will be increased to `1`. This can be useful to prevent problems, e.g. with image saving or plotting. If it is set to `False`, images will be returned as `(H', W')` or `(H', W', 3)` with `H` or `W` potentially being `0`.

Returns Pixels overlapping with the line string. Zero-padded if the line string is partially/fully outside of the image and `pad=True`. If *prevent_zero_size* is activated, it is guarantueed that `H'>0` and `W'>0`, otherwise only `H'>=0` and `W'>=0`.

Return type (H',W') ndarray or (H',W',C) ndarray

find_intersections_with(*self*, *other*)
Find all intersection points between this line string and *other*.

Parameters **other** (*tuple of number or list of tuple of number or list of LineString or LineString*) – The other geometry to use during intersection tests.

Returns All intersection points. One list per pair of consecutive start and end point, i.e. *N-1* lists of *N* points. Each list may be empty or may contain multiple points.

Return type list of list of tuple of number

get_pointwise_inside_image_mask(*self*, *image*)
Determine per point whether it is inside of a given image plane.

Parameters **image** (*ndarray or tuple of int*) – Either an image with shape `(H,W,[C])` or a `tuple` denoting such an image shape.

Returns `(N,)` ``bool array with one value for each of the `N` points indicating whether it is inside of the provided image plane (`True`) or not (`False`).

Return type ndarray

height
Compute the height of a bounding box encapsulating the line.

The height is computed based on the two points with lowest and largest y-coordinates.

Returns The height of the line string.

Return type float

is_fully_within_image(*self*, *image*, *default=False*)
Estimate whether the line string is fully inside an image plane.

Parameters

- **image** (*ndarray or tuple of int*) – Either an image with shape `(H,W,[C])` or a `tuple` denoting such an image shape.
- **default** (*any*) – Default value to return if the line string contains no points.

Returns `True` if the line string is fully inside the image area. `False` otherwise. Will return *default* if this line string contains no points.

Return type bool or any

is_out_of_image(*self*, *image*, *fully=True*, *partly=False*, *default=True*)
Estimate whether the line is partially/fully outside of the image area.

Parameters

- **image** (*ndarray or tuple of int*) – Either an image with shape `(H,W,[C])` or a tuple denoting such an image shape.
- **fully** (*bool, optional*) – Whether to return `True` if the line string is fully outside of the image area.
- **partly** (*bool, optional*) – Whether to return `True` if the line string is at least partially outside fo the image area.
- **default** (*any*) – Default value to return if the line string contains no points.

Returns `True` if the line string is partially/fully outside of the image area, depending on defined parameters. `False` otherwise. Will return *default* if this line string contains no points.

Return type bool or any

is_partly_within_image(*self*, *image*, *default=False*)
Estimate whether the line string is at least partially inside the image.

Parameters

- **image** (*ndarray or tuple of int*) – Either an image with shape `(H,W,[C])` or a `tuple` denoting such an image shape.
- **default** (*any*) – Default value to return if the line string contains no points.

Returns `True` if the line string is at least partially inside the image area. `False` otherwise. Will return *default* if this line string contains no points.

Return type bool or any

length
Compute the total euclidean length of the line string.

Returns The length based on euclidean distance, i.e. the sum of the lengths of each line segment.

Return type float

project(*self*, *from_shape*, *to_shape*)
Project the line string onto a differently shaped image.

E.g. if a point of the line string is on its original image at `x=(10 of 100 pixels)` and `y=(20 of 100 pixels)` and is projected onto a new image with size `(width=200, height=200)`, its new position will be `(x=20, y=40)`.

This is intended for cases where the original image is resized. It cannot be used for more complex changes (e.g. padding, cropping).

Parameters

- **from_shape** (*tuple of int or ndarray*) – Shape of the original image. (Before resize.)
- **to_shape** (*tuple of int or ndarray*) – Shape of the new image. (After resize.)

Returns Line string with new coordinates.

Return type *imgaug.augmentables.lines.LineString*

shift(*self*, *top=None*, *right=None*, *bottom=None*, *left=None*)
Move this line string along the x/y-axis.

Parameters

- **top** (*None or int, optional*) – Amount of pixels by which to shift this object *from* the top (towards the bottom).
- **right** (*None or int, optional*) – Amount of pixels by which to shift this object *from* the right (towards the left).
- **bottom** (*None or int, optional*) – Amount of pixels by which to shift this object *from* the bottom (towards the top).
- **left** (*None or int, optional*) – Amount of pixels by which to shift this object *from* the left (towards the right).

Returns **result** – Shifted line string.

Return type *imgaug.augmentables.lines.LineString*

subdivide(*self*, *points_per_edge*)
Derive a new line string with `N` interpolated points per edge.

The interpolated points have (per edge) regular distances to each other.

For each edge between points `A` and `B` this adds points at `A + (i/(1+N)) * (B - A)`, where `i` is the index of the added point and `N` is the number of points to add per edge.

Calling this method two times will split each edge at its center and then again split each newly created edge at their center. It is equivalent to calling *subdivide(3)*.

Parameters **points_per_edge** (*int*) – Number of points to interpolate on each edge.

Returns Line string with subdivided edges.

Return type *imgaug.augmentables.lines.LineString*

to_bounding_box(*self*)
Generate a bounding box encapsulating the line string.

Returns Bounding box encapsulating the line string. `None` if the line string contained no points.

Return type None or *imgaug.augmentables.bbs.BoundingBox*

to_heatmap(*self*, *image_shape*, *size_lines=1*, *size_points=0*, *antialiased=True*, *raise_if_out_of_image=False*)
Generate a heatmap object from the line string.

This is similar to *imgaug.augmentables.lines.LineString.draw_lines_heatmap_array()*, executed with `alpha=1.0`. The result is wrapped in a *imgaug.augmentables.heatmaps.HeatmapsOnImage* object instead of just an array. No points are drawn.

Parameters

- **image_shape** (*tuple of int*) – The shape of the image onto which to draw the line mask.
- **size_lines** (*int, optional*) – Thickness of the line.
- **size_points** (*int, optional*) – Size of the points in pixels.
- **antialiased** (*bool, optional*) – Whether to draw the line with anti-aliasing activated.
- **raise_if_out_of_image** (*bool, optional*) – Whether to raise an error if the line string is fully outside of the image. If set to `False`, no error will be raised and only the parts inside the image will be drawn.

Returns Heatmap object containing drawn line string.

Return type *imgaug.augmentables.heatmaps.HeatmapsOnImage*

to_keypoints(*self*)
Convert the line string points to keypoints.

Returns Points of the line string as keypoints.

Return type list of imgaug.augmentables.kps.Keypoint

to_polygon(*self*)
Generate a polygon from the line string points.

Returns Polygon with the same corner points as the line string. Note that the polygon might be invalid, e.g. contain less than `3` points or have self-intersections.

Return type *imgaug.augmentables.polys.Polygon*

to_segmentation_map(*self*, *image_shape*, *size_lines=1*, *size_points=0*, *raise_if_out_of_image=False*)
Generate a segmentation map object from the line string.

This is similar to *imgaug.augmentables.lines.LineString.draw_mask()*. The result is wrapped in a `SegmentationMapsOnImage` object instead of just an array.

Parameters

- **image_shape** (*tuple of int*) – The shape of the image onto which to draw the line mask.
- **size_lines** (*int, optional*) – Thickness of the line.
- **size_points** (*int, optional*) – Size of the points in pixels.
- **raise_if_out_of_image** (*bool, optional*) – Whether to raise an error if the line string is fully outside of the image. If set to `False`, no error will be raised and only the parts inside the image will be drawn.

Returns Segmentation map object containing drawn line string.

Return type *imgaug.augmentables.segmaps.SegmentationMapsOnImage*

width

Compute the width of a bounding box encapsulating the line.

The width is computed based on the two points with lowest and largest x-coordinates.

Returns The width of the line string.

Return type float

xx

Get an array of x-coordinates of all points of the line string.

Returns `float32` x-coordinates of the line string points.

Return type ndarray

xx_int

Get an array of discrete x-coordinates of all points.

The conversion from `float32` coordinates to `int32` is done by first rounding the coordinates to the closest integer and then removing everything after the decimal point.

Returns `int32` x-coordinates of the line string points.

Return type ndarray

yy

Get an array of y-coordinates of all points of the line string.

Returns `float32` y-coordinates of the line string points.

Return type ndarray

yy_int

Get an array of discrete y-coordinates of all points.

The conversion from `float32` coordinates to `int32` is done by first rounding the coordinates to the closest integer and then removing everything after the decimal point.

Returns `int32` y-coordinates of the line string points.

Return type ndarray

class imgaug.augmentables.lines.**LineStringsOnImage**(*line_strings*, *shape*)

Bases: `object`

Object that represents all line strings on a single image.

Parameters

- **line_strings** (*list of imgaug.augmentables.lines.LineString*) – List of line strings on the image.

- **shape** (*tuple of int or ndarray*) – The shape of the image on which the objects are placed. Either an image with shape `(H,W,[C])` or a `tuple` denoting such an image shape.

Examples

```
>>> import numpy as np
>>> from imgaug.augmentables.lines import LineString, LineStringsOnImage
>>>
>>> image = np.zeros((100, 100))
>>> lss = [
>>>     LineString([(0, 0), (10, 0)]),
>>>     LineString([(10, 20), (30, 30), (50, 70)])
>>> ]
>>> lsoi = LineStringsOnImage(lss, shape=image.shape)
```

Attributes

empty	Estimate whether this object contains zero line strings.

Methods

clip_out_of_image(self)	Clip off all parts of the line strings that are outside of an image.
copy(self[, line_strings, shape])	Create a shallow copy of this object.
deepcopy(self[, line_strings, shape])	Create a deep copy of the object.
draw_on_image(self, image[, color, ...])	Draw all line strings onto a given image.
from_xy_arrays(xy, shape)	Convert an `(N,M,2)` ndarray to a `LineStringsOnImage` object.
on(self, image)	Project the line strings from one image shape to a new one.
remove_out_of_image(self[, fully, partly])	Remove all line strings that are fully/partially outside of an image.
shift(self[, top, right, bottom, left])	Move the line strings along the x/y-axis.
to_xy_arrays(self[, dtype])	Convert this object to an iterable of `(M,2)` arrays of points.

clip_out_of_image(*self*)

Clip off all parts of the line strings that are outside of an image.

Note: The result can contain fewer line strings than the input did. That happens when a polygon is fully outside of the image plane.

Note: The result can also contain *more* line strings than the input did. That happens when distinct parts of a line string are only connected by line segments that are outside of the image plane and hence will be clipped off, resulting in two or more unconnected line string parts that are left in the image plane.

Returns Line strings, clipped to fall within the image dimensions. The count of output line strings may differ from the input count.

Return type *imgaug.augmentables.lines.LineStringsOnImage*

copy (*self*, *line_strings=None*, *shape=None*)
Create a shallow copy of this object.

Parameters

- **line_strings** (*None or list of imgaug.augmentables.lines.LineString, optional*) – List of line strings on the image. If not `None`, then the `line_strings` attribute of the copied object will be set to this value.
- **shape** (*None or tuple of int or ndarray, optional*) – The shape of the image on which the objects are placed. Either an image with shape `(H,W,[C])` or a tuple denoting such an image shape. If not `None`, then the `shape` attribute of the copied object will be set to this value.

Returns Shallow copy.

Return type *imgaug.augmentables.lines.LineStringsOnImage*

deepcopy (*self*, *line_strings=None*, *shape=None*)
Create a deep copy of the object.

Parameters

- **line_strings** (*None or list of imgaug.augmentables.lines.LineString, optional*) – List of line strings on the image. If not `None`, then the `line_strings` attribute of the copied object will be set to this value.
- **shape** (*None or tuple of int or ndarray, optional*) – The shape of the image on which the objects are placed. Either an image with shape `(H,W,[C])` or a tuple denoting such an image shape. If not `None`, then the `shape` attribute of the copied object will be set to this value.

Returns Deep copy.

Return type *imgaug.augmentables.lines.LineStringsOnImage*

draw_on_image (*self*, *image*, *color=(0, 255, 0)*, *color_lines=None*, *color_points=None*, *alpha=1.0*, *alpha_lines=None*, *alpha_points=None*, *size=1*, *size_lines=None*, *size_points=None*, *antialiased=True*, *raise_if_out_of_image=False*)
Draw all line strings onto a given image.

Parameters

- **image** (*ndarray*) – The `(H,W,C) uint8` image onto which to draw the line strings.
- **color** (*iterable of int, optional*) – Color to use as RGB, i.e. three values. The color of the lines and points are derived from this value, unless they are set.
- **color_lines** (*None or iterable of int*) – Color to use for the line segments as RGB, i.e. three values. If `None`, this value is derived from *color*.
- **color_points** (*None or iterable of int*) – Color to use for the points as RGB, i.e. three values. If `None`, this value is derived from `0.5 * color`.
- **alpha** (*float, optional*) – Opacity of the line strings. Higher values denote more visible points. The alphas of the line and points are derived from this value, unless they are set.
- **alpha_lines** (*None or float, optional*) – Opacity of the line strings. Higher values denote more visible line string. If `None`, this value is derived from *alpha*.
- **alpha_points** (*None or float, optional*) – Opacity of the line string points. Higher values denote more visible points. If `None`, this value is derived from *alpha*.

- **size** (*int, optional*) – Size of the line strings. The sizes of the line and points are derived from this value, unless they are set.
- **size_lines** (*None or int, optional*) – Thickness of the line segments. If `None`, this value is derived from *size*.
- **size_points** (*None or int, optional*) – Size of the points in pixels. If `None`, this value is derived from `3 * size`.
- **antialiased** (*bool, optional*) – Whether to draw the lines with anti-aliasing activated. This does currently not affect the point drawing.
- **raise_if_out_of_image** (*bool, optional*) – Whether to raise an error if a line string is fully outside of the image. If set to `False`, no error will be raised and only the parts inside the image will be drawn.

Returns Image with line strings drawn on it.

Return type ndarray

empty
Estimate whether this object contains zero line strings.

Returns `True` if this object contains zero line strings.

Return type bool

classmethod from_xy_arrays(*xy*, *shape*)
Convert an `(N,M,2)` ndarray to a `LineStringsOnImage` object.

This is the inverse of `imgaug.augmentables.lines.LineStringsOnImage.to_xy_array()`.

Parameters

- **xy** (*(N,M,2) ndarray or iterable of (M,2) ndarray*) – Array containing the point coordinates `N` line strings with each `M` points given as `(x,y)` coordinates. `M` may differ if an iterable of arrays is used. Each array should usually be of dtype `float32`.
- **shape** (*tuple of int*) – `(H,W,[C])` shape of the image on which the line strings are placed.

Returns Object containing a list of `LineString` objects following the provided point coordinates.

Return type *imgaug.augmentables.lines.LineStringsOnImage*

on(*self*, *image*)
Project the line strings from one image shape to a new one.

Parameters **image** (*ndarray or tuple of int*) – The new image onto which to project. Either an image with shape `(H,W,[C])` or a tuple denoting such an image shape.

Returns Object containing all projected line strings.

Return type imgaug.augmentables.lines.LineStrings

remove_out_of_image(*self*, *fully=True*, *partly=False*)
Remove all line strings that are fully/partially outside of an image.

Parameters

- **fully** (*bool, optional*) – Whether to remove line strings that are fully outside of the image.
- **partly** (*bool, optional*) – Whether to remove line strings that are partially outside of the image.

Returns Reduced set of line strings. Those that are fully/partially outside of the given image plane are removed.

Return type *imgaug.augmentables.lines.LineStringsOnImage*

shift(*self*, *top=None*, *right=None*, *bottom=None*, *left=None*)
Move the line strings along the x/y-axis.

Parameters

- **top** (*None or int, optional*) – Amount of pixels by which to shift all objects *from* the top (towards the bottom).
- **right** (*None or int, optional*) – Amount of pixels by which to shift all objects *from* the right (towads the left).
- **bottom** (*None or int, optional*) – Amount of pixels by which to shift all objects *from* the bottom (towards the top).
- **left** (*None or int, optional*) – Amount of pixels by which to shift all objects *from* the left (towards the right).

Returns Shifted line strings.

Return type *imgaug.augmentables.lines.LineStringsOnImage*

to_xy_arrays(*self*, *dtype=<class 'numpy.float32'>*)
Convert this object to an iterable of `(M,2)` arrays of points.

This is the inverse of `imgaug.augmentables.lines.LineStringsOnImage.from_xy_array()`.

Parameters **dtype** (*numpy.dtype, optional*) – Desired output datatype of the ndarray.

Returns The arrays of point coordinates, each given as `(M,2)`.

Return type list of ndarray

13.12 imgaug.augmentables.normalization

imgaug.augmentables.normalization.**estimate_bounding_boxes_norm_type**(*bounding_boxes*)

imgaug.augmentables.normalization.**estimate_heatmaps_norm_type**(*heatmaps*)

imgaug.augmentables.normalization.**estimate_keypoints_norm_type**(*keypoints*)

imgaug.augmentables.normalization.**estimate_line_strings_norm_type**(*line_strings*)

imgaug.augmentables.normalization.**estimate_normalization_type**(*inputs*)

imgaug.augmentables.normalization.**estimate_polygons_norm_type**(*polygons*)

imgaug.augmentables.normalization.**estimate_segmaps_norm_type**(*segmentation_maps*)

imgaug.augmentables.normalization.**find_first_nonempty**(*attr*, *parents=None*)

imgaug.augmentables.normalization.**invert_normalize_bounding_boxes**(*bounding_boxes*, *bounding_boxes_old*)

imgaug.augmentables.normalization.**invert_normalize_heatmaps**(*heatmaps*, *heatmaps_old*)

imgaug.augmentables.normalization.**invert_normalize_images**(*images*, *images_old*)

imgaug.augmentables.normalization.**invert_normalize_keypoints**(*keypoints*, *keypoints_old*)

imgaug.augmentables.normalization.**invert_normalize_line_strings**(*line_strings*, *line_strings_old*)

imgaug.augmentables.normalization.**invert_normalize_polygons**(*polygons*, *polygons_old*)

imgaug.augmentables.normalization.**invert_normalize_segmentation_maps**(*segmentation_maps*, *segmentation_maps_old*)

imgaug.augmentables.normalization.**normalize_bounding_boxes**(*inputs*, *shapes=None*)

imgaug.augmentables.normalization.**normalize_heatmaps**(*inputs*, *shapes=None*)

imgaug.augmentables.normalization.**normalize_images**(*images*)

imgaug.augmentables.normalization.**normalize_keypoints**(*inputs*, *shapes=None*)

imgaug.augmentables.normalization.**normalize_line_strings**(*inputs*, *shapes=None*)

imgaug.augmentables.normalization.**normalize_polygons**(*inputs*, *shapes=None*)

imgaug.augmentables.normalization.**normalize_segmentation_maps**(*inputs*, *shapes=None*)

imgaug.augmentables.normalization.**restore_dtype_and_merge**(*arr*, *input_dtype*)

13.13 imgaug.augmentables.polys

class imgaug.augmentables.polys.**MultiPolygon**(*geoms*)

Bases: `object`

Class that represents several polygons.

Parameters **geoms** (*list of imgaug.augmentables.polys.Polygon*) – List of the polygons.

Methods

from_shapely(geometry[, label])	Create a MultiPolygon from a shapely object.

static from_shapely(*geometry*, *label=None*)

Create a MultiPolygon from a shapely object.

This also creates all necessary `Polygon` s contained in this `MultiPolygon`.

Parameters

- **geometry** (*shapely.geometry.MultiPolygon or shapely.geometry.Polygon or shapely.geometry.collection.GeometryCollection*) – The object to convert to a MultiPolygon.
- **label** (*None or str, optional*) – A label assigned to all Polygons within the MultiPolygon.

Returns The derived MultiPolygon.

Return type *imgaug.augmentables.polys.MultiPolygon*

class imgaug.augmentables.polys.**Polygon**(*exterior*, *label=None*)

Bases: object

Class representing polygons.

Each polygon is parameterized by its corner points, given as absolute x- and y-coordinates with sub-pixel accuracy.

Parameters

- **exterior** (*list of imgaug.augmentables.kps.Keypoint or list of tuple of float or (N,2) ndarray*) – List of points defining the polygon. May be either a `list` of *imgaug.augmentables.kps.Keypoint* objects or a `list` of `tuple` s in xy-form or a numpy array of shape (N,2) for `N` points in xy-form. All coordinates are expected to be the absolute subpixel-coordinates on the image, given as `float` s, e.g. `x=10.7` and `y=3.4` for a point at coordinates `(10.7, 3.4)`. Their order is expected to be clock-wise. They are expected to not be closed (i.e. first and last coordinate differ).
- **label** (*None or str, optional*) – Label of the polygon, e.g. a string representing the class.

Attributes

area Compute the area of the polygon.

height Compute the height of a bounding box encapsulating the polygon.

is_valid Estimate whether the polygon has a valid geometry.

width Compute the width of a bounding box encapsulating the polygon.

xx Get the x-coordinates of all points on the exterior.

xx_int Get the discretized x-coordinates of all points on the exterior.

yy Get the y-coordinates of all points on the exterior.

yy_int Get the discretized y-coordinates of all points on the exterior.

Methods

almost_equals(self, other[, max_distance, ...])	Estimate if this polygon's and another's geometry/labels are similar.
change_first_point_by_coords(self, x, y[, ...])	Reorder exterior points so that the point closest to given x/y is first.
change_first_point_by_index(self, point_idx)	Reorder exterior points so that the point with given index is first.
clip_out_of_image(self, image)	Cut off all parts of the polygon that are outside of an image.
copy(self[, exterior, label])	Create a shallow copy of this object.
cut_out_of_image(self, image)	**Deprecated**.
deepcopy(self[, exterior, label])	Create a deep copy of this object.
draw_on_image(self, image[, color, ...])	Draw the polygon on an image.
exterior_almost_equals(self, other[, ...])	Estimate if this and another polygon's exterior are almost identical.
extract_from_image(self, image)	Extract all image pixels within the polygon area.
find_closest_point_index(self, x, y[, ...])	Find the index of the exterior point closest to given coordinates.
from_shapely(polygon_shapely[, label])	Create a polygon from a `Shapely` `Polygon`.

Continued on next page

Table 46 – continued from previous page

is_fully_within_image(self, image)	Estimate whether the polygon is fully inside an image plane.
is_out_of_image(self, image[, fully, partly])	Estimate whether the polygon is partially/fully outside of an image.
is_partly_within_image(self, image)	Estimate whether the polygon is at least partially inside an image.
project(self, from_shape, to_shape)	Project the polygon onto an image with different shape.
shift(self[, top, right, bottom, left])	Move this polygon along the x/y-axis.
to_bounding_box(self)	Convert this polygon to a bounding box containing the polygon.
to_keypoints(self)	Convert this polygon's exterior to `Keypoint` instances.
to_line_string(self[, closed])	Convert this polygon's exterior to a `LineString` instance.
to_shapely_line_string(self[, closed, ...])	Convert this polygon to a `Shapely` `LineString` object.
to_shapely_polygon(self)	Convert this polygon to a `Shapely` `Polygon`.

almost_equals(*self*, *other*, *max_distance=0.0001*, *points_per_edge=8*)

Estimate if this polygon's and another's geometry/labels are similar.

This is the same as *imgaug.augmentables.polys.Polygon.exterior_almost_equals()* but additionally compares the labels.

Parameters

- **other** (*imgaug.augmentables.polys.Polygon or any*) – The object to compare against. If not a *Polygon*, `False` will always be returned.
- **max_distance** (*float, optional*) – See *imgaug.augmentables.polys.Polygon.exterior_almost_equals()*.
- **points_per_edge** (*int, optional*) – See *imgaug.augmentables.polys.Polygon.exterior_almost_equals()*.

Returns Whether the two polygons can be viewed as equal. In the case of the exteriors this is an approximate test.

Return type bool

area

Compute the area of the polygon.

Returns Area of the polygon.

Return type number

change_first_point_by_coords(*self*, *x*, *y*, *max_distance=0.0001*, *raise_if_too_far_away=True*)

Reorder exterior points so that the point closest to given x/y is first.

This method takes a given `(x,y)` coordinate, finds the closest corner point on the exterior and reorders all exterior corner points so that the found point becomes the first one in the array.

If no matching points are found, an exception is raised.

Parameters

- **x** (*number*) – X-coordinate of the point.

- **y** (*number*) – Y-coordinate of the point.
- **max_distance** (*None or number, optional*) – Maximum distance past which possible matches are ignored. If `None` the distance limit is deactivated.
- **raise_if_too_far_away** (*bool, optional*) – Whether to raise an exception if the closest found point is too far away (`True`) or simply return an unchanged copy if this object (`False`).

Returns Copy of this polygon with the new point order.

Return type *imgaug.augmentables.polys.Polygon*

change_first_point_by_index(*self*, *point_idx*)

Reorder exterior points so that the point with given index is first.

This method takes a given index and reorders all exterior corner points so that the point with that index becomes the first one in the array.

An `AssertionError` will be raised if the index does not match any exterior point's index or the exterior does not contain any points.

Parameters **point_idx** (*int*) – Index of the desired starting point.

Returns Copy of this polygon with the new point order.

Return type *imgaug.augmentables.polys.Polygon*

clip_out_of_image(*self*, *image*)

Cut off all parts of the polygon that are outside of an image.

This operation may lead to new points being created. As a single polygon may be split into multiple new polygons, the result is always a list, which may contain more than one output polygon.

This operation will return an empty list if the polygon is completely outside of the image plane.

Parameters **image** (*(H,W,...) ndarray or tuple of int*) – Image dimensions to use for the clipping of the polygon. If an `ndarray`, its shape will be used. If a `tuple`, it is assumed to represent the image shape and must contain at least two `int` s.

Returns Polygon, clipped to fall within the image dimensions. Returned as a `list`, because the clipping can split the polygon into multiple parts. The list may also be empty, if the polygon was fully outside of the image plane.

Return type list of imgaug.augmentables.polys.Polygon

copy(*self*, *exterior=None*, *label=None*)

Create a shallow copy of this object.

Parameters

- **exterior** (*list of imgaug.augmentables.kps.Keypoint or list of tuple or (N,2) ndarray, optional*) – List of points defining the polygon. See `imgaug.augmentables.polys.Polygon.__init__()` for details.
- **label** (*None or str, optional*) – If not `None`, the `label` of the copied object will be set to this value.

Returns Shallow copy.

Return type *imgaug.augmentables.polys.Polygon*

cut_out_of_image(*self*, *image*)

Deprecated. Use `Polygon.clip_out_of_image()` instead. clip_out_of_image() has the exactly same interface.

deepcopy (*self*, *exterior=None*, *label=None*)
Create a deep copy of this object.

Parameters

- **exterior** (*list of Keypoint or list of tuple or (N,2) ndarray, optional*) – List of points defining the polygon. See *imgaug.augmentables.polys.Polygon.__init__* for details.
- **label** (*None or str*) – If not `None`, the `label` of the copied object will be set to this value.

Returns Deep copy.

Return type *imgaug.augmentables.polys.Polygon*

draw_on_image (*self*, *image*, *color=(0, 255, 0)*, *color_face=None*, *color_lines=None*, *color_points=None*, *alpha=1.0*, *alpha_face=None*, *alpha_lines=None*, *alpha_points=None*, *size=1*, *size_lines=None*, *size_points=None*, *raise_if_out_of_image=False*)
Draw the polygon on an image.

Parameters

- **image** (*(H,W,C) ndarray*) – The image onto which to draw the polygon. Usually expected to be of dtype `uint8`, though other dtypes are also handled.
- **color** (*iterable of int, optional*) – The color to use for the whole polygon. Must correspond to the channel layout of the image. Usually RGB. The values for *color_face*, *color_lines* and *color_points* will be derived from this color if they are set to `None`. This argument has no effect if *color_face*, *color_lines* and *color_points* are all set anything other than `None`.
- **color_face** (*None or iterable of int, optional*) – The color to use for the inner polygon area (excluding perimeter). Must correspond to the channel layout of the image. Usually RGB. If this is `None`, it will be derived from `color * 1.0`.
- **color_lines** (*None or iterable of int, optional*) – The color to use for the line (aka perimeter/border) of the polygon. Must correspond to the channel layout of the image. Usually RGB. If this is `None`, it will be derived from `color * 0.5`.
- **color_points** (*None or iterable of int, optional*) – The color to use for the corner points of the polygon. Must correspond to the channel layout of the image. Usually RGB. If this is `None`, it will be derived from `color * 0.5`.
- **alpha** (*float, optional*) – The opacity of the whole polygon, where `1.0` denotes a completely visible polygon and `0.0` an invisible one. The values for *alpha_face*, *alpha_lines* and *alpha_points* will be derived from this alpha value if they are set to `None`. This argument has no effect if *alpha_face*, *alpha_lines* and *alpha_points* are all set anything other than `None`.
- **alpha_face** (*None or number, optional*) – The opacity of the polygon's inner area (excluding the perimeter), where `1.0` denotes a completely visible inner area and `0.0` an invisible one. If this is `None`, it will be derived from `alpha * 0.5`.
- **alpha_lines** (*None or number, optional*) – The opacity of the polygon's line (aka perimeter/border), where `1.0` denotes a completely visible line and `0.0` an invisible one. If this is `None`, it will be derived from `alpha * 1.0`.
- **alpha_points** (*None or number, optional*) – The opacity of the polygon's corner points, where `1.0` denotes completely visible corners and `0.0` invisible ones. If this is `None`, it will be derived from `alpha * 1.0`.
- **size** (*int, optional*) – Size of the polygon. The sizes of the line and points are derived from this value, unless they are set.

- **size_lines** (*None or int, optional*) – Thickness of the polygon's line (aka perimeter/border). If `None`, this value is derived from *size*.
- **size_points** (*int, optional*) – Size of the points in pixels. If `None`, this value is derived from `3 * size`.
- **raise_if_out_of_image** (*bool, optional*) – Whether to raise an error if the polygon is fully outside of the image. If set to `False`, no error will be raised and only the parts inside the image will be drawn.

Returns Image with the polygon drawn on it. Result dtype is the same as the input dtype.

Return type (H,W,C) ndarray

exterior_almost_equals (*self*, *other*, *max_distance=0.0001*, *points_per_edge=8*)

Estimate if this and another polygon's exterior are almost identical.

The two exteriors can have different numbers of points, but any point randomly sampled on the exterior of one polygon should be close to the closest point on the exterior of the other polygon.

Note: This method works in an approximative way. One can come up with polygons with fairly different shapes that will still be estimated as equal by this method. In practice however this should be unlikely to be the case. The probability for something like that goes down as the interpolation parameter is increased.

Parameters

- **other** (*imgaug.augmentables.polys.Polygon or (N,2) ndarray or list of tuple*) – The other polygon with which to compare the exterior. If this is an `ndarray`, it is assumed to represent an exterior. It must then have dtype `float32` and shape `(N,2)` with the second dimension denoting xy-coordinates. If this is a `list` of `tuple` s, it is assumed to represent an exterior. Each tuple then must contain exactly two `number` s, denoting xy-coordinates.
- **max_distance** (*number, optional*) – The maximum euclidean distance between a point on one polygon and the closest point on the other polygon. If the distance is exceeded for any such pair, the two exteriors are not viewed as equal. The points are other the points contained in the polygon's exterior ndarray or interpolated points between these.
- **points_per_edge** (*int, optional*) – How many points to interpolate on each edge.

Returns Whether the two polygon's exteriors can be viewed as equal (approximate test).

Return type bool

extract_from_image (*self*, *image*)

Extract all image pixels within the polygon area.

This method returns a rectangular image array. All pixels within that rectangle that do not belong to the polygon area will be filled with zeros (i.e. they will be black). The method will also zero-pad the image if the polygon is partially/fully outside of the image.

Parameters **image** (*(H,W) ndarray or (H,W,C) ndarray*) – The image from which to extract the pixels within the polygon.

Returns Pixels within the polygon. Zero-padded if the polygon is partially/fully outside of the image.

Return type (H',W') ndarray or (H',W',C) ndarray

find_closest_point_index (*self*, *x*, *y*, *return_distance=False*)
Find the index of the exterior point closest to given coordinates.

"Closeness" is here defined based on euclidean distance. This method will raise an `AssertionError` if the exterior contains no points.

Parameters

- **x** (*number*) – X-coordinate around which to search for close points.
- **y** (*number*) – Y-coordinate around which to search for close points.
- **return_distance** (*bool, optional*) – Whether to also return the distance of the closest point.

Returns

- *int* – Index of the closest point.
- *number* – Euclidean distance to the closest point. This value is only returned if *return_distance* was set to `True`.

static from_shapely (*polygon_shapely*, *label=None*)
Create a polygon from a `Shapely Polygon`.

Note: This will remove any holes in the shapely polygon.

Parameters

- **polygon_shapely** (*shapely.geometry.Polygon*) – The shapely polygon.
- **label** (*None or str, optional*) – The label of the new polygon.

Returns A polygon with the same exterior as the `Shapely Polygon`.

Return type *imgaug.augmentables.polys.Polygon*

height
Compute the height of a bounding box encapsulating the polygon.

The height is computed based on the two exterior coordinates with lowest and largest x-coordinates.

Returns Height of the polygon.

Return type number

is_fully_within_image (*self*, *image*)
Estimate whether the polygon is fully inside an image plane.

Parameters **image** (*(H,W,...) ndarray or tuple of int*) – Image dimensions to use. If an `ndarray`, its shape will be used. If a `tuple`, it is assumed to represent the image shape and must contain at least two `int` s.

Returns `True` if the polygon is fully inside the image area. `False` otherwise.

Return type bool

is_out_of_image (*self*, *image*, *fully=True*, *partly=False*)
Estimate whether the polygon is partially/fully outside of an image.

Parameters

- **image** (*(H,W,...) ndarray or tuple of int*) – Image dimensions to use. If an `ndarray`, its shape will be used. If a `tuple`, it is assumed to represent the image shape and must contain at least two `int` s.

- **fully** (*bool, optional*) – Whether to return `True` if the polygon is fully outside of the image area.
- **partly** (*bool, optional*) – Whether to return `True` if the polygon is at least partially outside fo the image area.

Returns `True` if the polygon is partially/fully outside of the image area, depending on defined parameters. `False` otherwise.

Return type bool

is_partly_within_image(*self*, *image*)

Estimate whether the polygon is at least partially inside an image.

Parameters **image** (*(H,W,...) ndarray or tuple of int*) – Image dimensions to use. If an `ndarray`, its shape will be used. If a `tuple`, it is assumed to represent the image shape and must contain at least two `int` s.

Returns `True` if the polygon is at least partially inside the image area. `False` otherwise.

Return type bool

is_valid

Estimate whether the polygon has a valid geometry.

To to be considered valid, the polygon must be made up of at least `3` points and have a concave shape, i.e. line segments may not intersect or overlap. Multiple consecutive points are allowed to have the same coordinates.

Returns `True` if polygon has at least `3` points and is concave, otherwise `False`.

Return type bool

project(*self*, *from_shape*, *to_shape*)

Project the polygon onto an image with different shape.

The relative coordinates of all points remain the same. E.g. a point at `(x=20, y=20)` on an image `(width=100, height=200)` will be projected on a new image `(width=200, height=100)` to `(x=40, y=10)`.

This is intended for cases where the original image is resized. It cannot be used for more complex changes (e.g. padding, cropping).

Parameters

- **from_shape** (*tuple of int*) – Shape of the original image. (Before resize.)
- **to_shape** (*tuple of int*) – Shape of the new image. (After resize.)

Returns Polygon object with new coordinates.

Return type *imgaug.augmentables.polys.Polygon*

shift(*self*, *top=None*, *right=None*, *bottom=None*, *left=None*)

Move this polygon along the x/y-axis.

Parameters

- **top** (*None or int, optional*) – Amount of pixels by which to shift this object *from* the top (towards the bottom).
- **right** (*None or int, optional*) – Amount of pixels by which to shift this object *from* the right (towards the left).
- **bottom** (*None or int, optional*) – Amount of pixels by which to shift this object *from* the bottom (towards the top).

- **left** (*None or int, optional*) – Amount of pixels by which to shift this object *from* the left (towards the right).

Returns Shifted polygon.

Return type *imgaug.augmentables.polys.Polygon*

to_bounding_box(*self*)
Convert this polygon to a bounding box containing the polygon.

Returns Bounding box that tightly encapsulates the polygon.

Return type *imgaug.augmentables.bbs.BoundingBox*

to_keypoints(*self*)
Convert this polygon's exterior to `Keypoint` instances.

Returns Exterior vertices as *`imgaug.augmentables.kps.Keypoint`* instances.

Return type list of imgaug.augmentables.kps.Keypoint

to_line_string(*self*, *closed=True*)
Convert this polygon's exterior to a `LineString` instance.

Parameters **closed** (*bool, optional*) – Whether to close the line string, i.e. to add the first point of the *exterior* also as the last point at the end of the line string. This has no effect if the polygon has a single point or zero points.

Returns Exterior of the polygon as a line string.

Return type *imgaug.augmentables.lines.LineString*

to_shapely_line_string(*self*, *closed=False*, *interpolate=0*)
Convert this polygon to a `Shapely` `LineString` object.

Parameters

- **closed** (*bool, optional*) – Whether to return the line string with the last point being identical to the first point.
- **interpolate** (*int, optional*) – Number of points to interpolate between any pair of two consecutive points. These points are added to the final line string.

Returns The `Shapely` `LineString` matching the polygon's exterior.

Return type shapely.geometry.LineString

to_shapely_polygon(*self*)
Convert this polygon to a `Shapely` `Polygon`.

Returns The `Shapely` `Polygon` matching this polygon's exterior.

Return type shapely.geometry.Polygon

width
Compute the width of a bounding box encapsulating the polygon.

The width is computed based on the two exterior coordinates with lowest and largest x-coordinates.

Returns Width of the polygon.

Return type number

xx
Get the x-coordinates of all points on the exterior.

Returns `float32` x-coordinates array of all points on the exterior.

Return type (N,2) ndarray

xx_int
Get the discretized x-coordinates of all points on the exterior.

The conversion from `float32` coordinates to `int32` is done by first rounding the coordinates to the closest integer and then removing everything after the decimal point.

Returns `int32` x-coordinates of all points on the exterior.

Return type (N,2) ndarray

yy
Get the y-coordinates of all points on the exterior.

Returns `float32` y-coordinates array of all points on the exterior.

Return type (N,2) ndarray

yy_int
Get the discretized y-coordinates of all points on the exterior.

The conversion from `float32` coordinates to `int32` is done by first rounding the coordinates to the closest integer and then removing everything after the decimal point.

Returns `int32` y-coordinates of all points on the exterior.

Return type (N,2) ndarray

class imgaug.augmentables.polys.**PolygonsOnImage**(*polygons*, *shape*)
Bases: `object`

Container for all polygons on a single image.

Parameters

- **polygons** (*list of imgaug.augmentables.polys.Polygon*) – List of polygons on the image.
- **shape** (*tuple of int*) – The shape of the image on which the objects are placed. Either an image with shape `(H,W,[C])` or a `tuple` denoting such an image shape.

Examples

```
>>> import numpy as np
>>> from imgaug.augmentables.polys import Polygon, PolygonsOnImage
>>> image = np.zeros((100, 100))
>>> polys = [
>>>     Polygon([(0.5, 0.5), (100.5, 0.5), (100.5, 100.5), (0.5, 100.5)]),
>>>     Polygon([(50.5, 0.5), (100.5, 50.5), (50.5, 100.5), (0.5, 50.5)])
>>> ]
>>> polys_oi = PolygonsOnImage(polys, shape=image.shape)
```

Attributes

empty Estimate whether this object contains zero polygons.

Methods

clip_out_of_image(self)	Clip off all parts from all polygons that are outside of an image.
copy(self)	Create a shallow copy of this object.
deepcopy(self)	Create a deep copy of this object.
draw_on_image(self, image[, color, ...])	Draw all polygons onto a given image.
on(self, image)	Project all polygons from one image shape to a new one.
remove_out_of_image(self[, fully, partly])	Remove all polygons that are fully/partially outside of an image.
shift(self[, top, right, bottom, left])	Move the polygons along the x/y-axis.

clip_out_of_image(*self*)
Clip off all parts from all polygons that are outside of an image.

Note: The result can contain fewer polygons than the input did. That happens when a polygon is fully outside of the image plane.

Note: The result can also contain *more* polygons than the input did. That happens when distinct parts of a polygon are only connected by areas that are outside of the image plane and hence will be clipped off, resulting in two or more unconnected polygon parts that are left in the image plane.

Returns Polygons, clipped to fall within the image dimensions. The count of output polygons may differ from the input count.

Return type *imgaug.augmentables.polys.PolygonsOnImage*

copy(*self*)
Create a shallow copy of this object.

Returns Shallow copy.

Return type *imgaug.augmentables.polys.PolygonsOnImage*

deepcopy(*self*)
Create a deep copy of this object.

Returns Deep copy.

Return type *imgaug.augmentables.polys.PolygonsOnImage*

draw_on_image(*self*, *image*, *color=(0, 255, 0)*, *color_face=None*, *color_lines=None*, *color_points=None*, *alpha=1.0*, *alpha_face=None*, *alpha_lines=None*, *alpha_points=None*, *size=1*, *size_lines=None*, *size_points=None*, *raise_if_out_of_image=False*)
Draw all polygons onto a given image.

Parameters

- **image** (*(H,W,C) ndarray*) – The image onto which to draw the bounding boxes. This image should usually have the same shape as set in `PolygonsOnImage.shape`.
- **color** (*iterable of int, optional*) – The color to use for the whole polygons. Must correspond to the channel layout of the image. Usually RGB. The values for *color_face*, *color_lines* and *color_points* will be derived from this color if they are set to `None`. This argument has no effect if *color_face*, *color_lines* and *color_points* are all set anything other than `None`.

- **color_face** (*None or iterable of int, optional*) – The color to use for the inner polygon areas (excluding perimeters). Must correspond to the channel layout of the image. Usually RGB. If this is `None`, it will be derived from `color * 1.0`.
- **color_lines** (*None or iterable of int, optional*) – The color to use for the lines (aka perimeters/borders) of the polygons. Must correspond to the channel layout of the image. Usually RGB. If this is `None`, it will be derived from `color * 0.5`.
- **color_points** (*None or iterable of int, optional*) – The color to use for the corner points of the polygons. Must correspond to the channel layout of the image. Usually RGB. If this is `None`, it will be derived from `color * 0.5`.
- **alpha** (*float, optional*) – The opacity of the whole polygons, where `1.0` denotes completely visible polygons and `0.0` invisible ones. The values for *alpha_face*, *alpha_lines* and *alpha_points* will be derived from this alpha value if they are set to `None`. This argument has no effect if *alpha_face*, *alpha_lines* and *alpha_points* are all set anything other than `None`.
- **alpha_face** (*None or number, optional*) – The opacity of the polygon's inner areas (excluding the perimeters), where `1.0` denotes completely visible inner areas and `0.0` invisible ones. If this is `None`, it will be derived from `alpha * 0.5`.
- **alpha_lines** (*None or number, optional*) – The opacity of the polygon's lines (aka perimeters/borders), where `1.0` denotes completely visible perimeters and `0.0` invisible ones. If this is `None`, it will be derived from `alpha * 1.0`.
- **alpha_points** (*None or number, optional*) – The opacity of the polygon's corner points, where `1.0` denotes completely visible corners and `0.0` invisible ones. Currently this is an on/off choice, i.e. only `0.0` or `1.0` are allowed. If this is `None`, it will be derived from `alpha * 1.0`.
- **size** (*int, optional*) – Size of the polygons. The sizes of the line and points are derived from this value, unless they are set.
- **size_lines** (*None or int, optional*) – Thickness of the polygon lines (aka perimeter/border). If `None`, this value is derived from *size*.
- **size_points** (*int, optional*) – The size of all corner points. If set to `C`, each corner point will be drawn as a square of size `C x C`.
- **raise_if_out_of_image** (*bool, optional*) – Whether to raise an error if any polygon is fully outside of the image. If set to False, no error will be raised and only the parts inside the image will be drawn.

Returns Image with drawn polygons.

Return type (H,W,C) ndarray

empty

Estimate whether this object contains zero polygons.

Returns `True` if this object contains zero polygons.

Return type bool

on(*self*, *image*)

Project all polygons from one image shape to a new one.

Parameters **image** (*ndarray or tuple of int*) – New image onto which the polygons are to be projected. May also simply be that new image's shape `tuple`.

Returns Object containing all projected polygons.

Return type *imgaug.augmentables.polys.PolygonsOnImage*

remove_out_of_image(*self*, *fully=True*, *partly=False*)
Remove all polygons that are fully/partially outside of an image.

Parameters

- **fully** (*bool, optional*) – Whether to remove polygons that are fully outside of the image.
- **partly** (*bool, optional*) – Whether to remove polygons that are partially outside of the image.

Returns Reduced set of polygons. Those that are fully/partially outside of the given image plane are removed.

Return type *imgaug.augmentables.polys.PolygonsOnImage*

shift(*self*, *top=None*, *right=None*, *bottom=None*, *left=None*)
Move the polygons along the x/y-axis.

Parameters

- **top** (*None or int, optional*) – Amount of pixels by which to shift all objects *from* the top (towards the bottom).
- **right** (*None or int, optional*) – Amount of pixels by which to shift all objects *from* the right (towads the left).
- **bottom** (*None or int, optional*) – Amount of pixels by which to shift all objects *from* the bottom (towards the top).
- **left** (*None or int, optional*) – Amount of pixels by which to shift all objects *from* the left (towards the right).

Returns Shifted polygons.

Return type *imgaug.augmentables.polys.PolygonsOnImage*

13.14 imgaug.augmentables.segmaps

imgaug.augmentables.segmaps.**SegmentationMapOnImage**(**args*, ***kwargs*)
Deprecated. Use `SegmentationMapsOnImage` instead. (Note the plural 'Maps' instead of old 'Map'.).

class imgaug.augmentables.segmaps.**SegmentationMapsOnImage**(*arr*, *shape*, *nb_classes=None*)
Bases: `object`

Object representing a segmentation map associated with an image.

Variables ***DEFAULT_SEGMENT_COLORS*** (*list of tuple of int*) – Standard RGB colors to use during drawing, ordered by class index.

Parameters

- **arr** (*(H,W) ndarray or (H,W,C) ndarray*) – Array representing the segmentation map(s). May have dtypes bool, int or uint.
- **shape** (*tuple of int*) – Shape of the image on which the segmentation map(s) is/are placed. **Not** the shape of the segmentation map(s) array, unless it is identical to the image shape (note the likely difference between the arrays in the number of channels). This is expected to be `(H, W)` or `(H, W, C)` with `C` usually being `3`. If there is no corresponding image, use `(H_arr, W_arr)` instead, where `H_arr` is the height of the segmentation map(s) array (analogous `W_arr`).

- **nb_classes** (*None or int, optional*) – Deprecated.

Methods

copy(self[, arr, shape])	Create a shallow copy of the segmentation map object.
deepcopy(self[, arr, shape])	Create a deep copy of the segmentation map object.
draw(self[, size, colors])	Render the segmentation map as an RGB image.
draw_on_image(self, image[, alpha, resize, ...])	Draw the segmentation map as an overlay over an image.
get_arr(self)	Return the seg.map array, with original dtype and shape ndim.
get_arr_int(self, *args, **kwargs)	**Deprecated**.
pad(self[, top, right, bottom, left, mode, cval])	Pad the segmentation maps at their top/right/bottom/left side.
pad_to_aspect_ratio(self, aspect_ratio[, ...])	Pad the segmentation maps until they match a target aspect ratio.
resize(self, sizes[, interpolation])	Resize the seg.map(s) array given a target size and interpolation.
scale(self, *args, **kwargs)	**Deprecated**.

DEFAULT_SEGMENT_COLORS = [(0, 0, 0), (230, 25, 75), (60, 180, 75), (255, 225, 25), (0,

copy(*self*, *arr=None*, *shape=None*)

Create a shallow copy of the segmentation map object.

Parameters

- **arr** (*None or (H,W) ndarray or (H,W,C) ndarray, optional*) – Optionally the *arr* attribute to use for the new segmentation map instance. Will be copied from the old instance if not provided. See `imgaug.augmentables.segmaps.SegmentationMapsOnImage.__init__()` for details.
- **shape** (*None or tuple of int, optional*) – Optionally the shape attribute to use for the the new segmentation map instance. Will be copied from the old instance if not provided. See `imgaug.augmentables.segmaps.SegmentationMapsOnImage.__init__()` for details.

Returns Shallow copy.

Return type *imgaug.augmentables.segmaps.SegmentationMapsOnImage*

deepcopy(*self*, *arr=None*, *shape=None*)

Create a deep copy of the segmentation map object.

Parameters

- **arr** (*None or (H,W) ndarray or (H,W,C) ndarray, optional*) – Optionally the *arr* attribute to use for the new segmentation map instance. Will be copied from the old instance if not provided. See `imgaug.augmentables.segmaps.SegmentationMapsOnImage.__init__()` for details.
- **shape** (*None or tuple of int, optional*) – Optionally the shape attribute to use for the the new segmentation map instance. Will be copied from the old instance if not provided. See `imgaug.augmentables.segmaps.SegmentationMapsOnImage.__init__()` for details.

Returns Deep copy.

Return type *imgaug.augmentables.segmaps.SegmentationMapsOnImage*

draw(*self*, *size=None*, *colors=None*)
Render the segmentation map as an RGB image.

Parameters

- **size** (*None or float or iterable of int or iterable of float, optional*) – Size of the rendered RGB image as `(height, width)`. See *imgaug.imgaug.imresize_single_image()* for details. If set to `None`, no resizing is performed and the size of the segmentation map array is used.
- **colors** (*None or list of tuple of int, optional*) – Colors to use. One for each class to draw. If `None`, then default colors will be used.

Returns Rendered segmentation map (dtype is `uint8`). One per `C` in the original input array `(H,W,C)`.

Return type list of (H,W,3) ndarray

draw_on_image(*self*, *image*, *alpha=0.75*, *resize='segmentation_map'*, *colors=None*, *draw_background=False*, *background_class_id=0*, *background_threshold=None*)
Draw the segmentation map as an overlay over an image.

Parameters

- **image** (*(H,W,3) ndarray*) – Image onto which to draw the segmentation map. Expected dtype is `uint8`.
- **alpha** (*float, optional*) – Alpha/opacity value to use for the mixing of image and segmentation map. Larger values mean that the segmentation map will be more visible and the image less visible.
- **resize** (*{'segmentation_map', 'image'}, optional*) – In case of size differences between the image and segmentation map, either the image or the segmentation map can be resized. This parameter controls which of the two will be resized to the other's size.
- **colors** (*None or list of tuple of int, optional*) – Colors to use. One for each class to draw. If `None`, then default colors will be used.
- **draw_background** (*bool, optional*) – If `True`, the background will be drawn like any other class. If `False`, the background will not be drawn, i.e. the respective background pixels will be identical with the image's RGB color at the corresponding spatial location and no color overlay will be applied.
- **background_class_id** (*int, optional*) – Class id to interpret as the background class. See *draw_background*.
- **background_threshold** (*None, optional*) – Deprecated. This parameter is ignored.

Returns Rendered overlays as `uint8` arrays. Always a **list** containing one RGB image per segmentation map array channel.

Return type list of (H,W,3) ndarray

get_arr(*self*)
Return the seg.map array, with original dtype and shape ndim.

Here, "original" denotes the dtype and number of shape dimensions that was used when the *SegmentationMapsOnImage* instance was created, i.e. upon the call of `SegmentationMapsOnImage.__init__()`. Internally, this class may use a different dtype and shape to simplify computations.

Note: The height and width may have changed compared to the original input due to e.g. pooling operations.

Returns Segmentation map array. Same dtype and number of dimensions as was originally used when the *SegmentationMapsOnImage* instance was created.

Return type ndarray

get_arr_int(*self*, **args*, ***kwargs*)
Deprecated. Use `SegmentationMapsOnImage.get_arr()` instead.

pad(*self*, *top=0*, *right=0*, *bottom=0*, *left=0*, *mode='constant'*, *cval=0*)
Pad the segmentation maps at their top/right/bottom/left side.

Parameters

- **top** (*int, optional*) – Amount of pixels to add at the top side of the segmentation map. Must be `0` or greater.
- **right** (*int, optional*) – Amount of pixels to add at the right side of the segmentation map. Must be `0` or greater.
- **bottom** (*int, optional*) – Amount of pixels to add at the bottom side of the segmentation map. Must be `0` or greater.
- **left** (*int, optional*) – Amount of pixels to add at the left side of the segmentation map. Must be `0` or greater.
- **mode** (*str, optional*) – Padding mode to use. See *imgaug.imgaug.pad()* for details.
- **cval** (*number, optional*) – Value to use for padding if *mode* is `constant`. See *imgaug.imgaug.pad()* for details.

Returns Padded segmentation map with height `H'=H+top+bottom` and width `W'=W+left+right`.

Return type *imgaug.augmentables.segmaps.SegmentationMapsOnImage*

pad_to_aspect_ratio(*self*, *aspect_ratio*, *mode='constant'*, *cval=0*, *return_pad_amounts=False*)
Pad the segmentation maps until they match a target aspect ratio.

Depending on which dimension is smaller (height or width), only the corresponding sides (left/right or top/bottom) will be padded. In each case, both of the sides will be padded equally.

Parameters

- **aspect_ratio** (*float*) – Target aspect ratio, given as width/height. E.g. `2.0` denotes the image having twice as much width as height.
- **mode** (*str, optional*) – Padding mode to use. See *imgaug.imgaug.pad()* for details.
- **cval** (*number, optional*) – Value to use for padding if *mode* is `constant`. See *imgaug.imgaug.pad()* for details.
- **return_pad_amounts** (*bool, optional*) – If `False`, then only the padded instance will be returned. If `True`, a tuple with two entries will be returned, where the first entry is the padded instance and the second entry are the amounts by which each array side was padded. These amounts are again a tuple of the form `(top, right, bottom, left)`, with each value being an integer.

Returns

- *imgaug.augmentables.segmaps.SegmentationMapsOnImage* – Padded segmentation map as `SegmentationMapsOnImage` instance.
- *tuple of int* – Amounts by which the instance's array was padded on each side, given as a tuple `(top, right, bottom, left)`. This tuple is only returned if *return_pad_amounts* was set to `True`.

resize(*self*, *sizes*, *interpolation='nearest'*)

Resize the seg.map(s) array given a target size and interpolation.

Parameters

- **sizes** (*float or iterable of int or iterable of float*) – New size of the array in `(height, width)`. See `imgaug.imgaug.imresize_single_image()` for details.
- **interpolation** (*None or str or int, optional*) – The interpolation to use during resize. Nearest neighbour interpolation (`"nearest"`) is almost always the best choice. See `imgaug.imgaug.imresize_single_image()` for details.

Returns Resized segmentation map object.

Return type *imgaug.augmentables.segmaps.SegmentationMapsOnImage*

scale(*self*, **args*, ***kwargs*)

Deprecated. Use `SegmentationMapsOnImage.resize()` instead. resize() has the exactly same interface.

13.15 imgaug.augmentables.utils

imgaug.augmentables.utils.**copy_augmentables**(*augmentables*)

imgaug.augmentables.utils.**interpolate_point_pair**(*point_a*, *point_b*, *nb_steps*)

Interpolate `N` points on a line segment.

Parameters

- **point_a** (*iterable of number*) – Start point of the line segment, given as `(x,y)` coordinates.
- **point_b** (*iterable of number*) – End point of the line segment, given as `(x,y)` coordinates.
- **nb_steps** (*int*) – Number of points to interpolate between *point_a* and *point_b*.

Returns The interpolated points. Does not include *point_a*.

Return type list of tuple of number

imgaug.augmentables.utils.**interpolate_points**(*points*, *nb_steps*, *closed=True*)

Interpolate `N` on each line segment in a line string.

Parameters

- **points** (*iterable of iterable of number*) – Points on the line segments, each one given as `(x,y)` coordinates. They are assumed to form one connected line string.
- **nb_steps** (*int*) – Number of points to interpolate on each individual line string.
- **closed** (*bool, optional*) – If `True` the output contains the last point in *points*. Otherwise it does not (but it will contain the interpolated points leading to the last point).

Returns Coordinates of *points*, with additional *nb_steps* new points interpolated between each point pair. If *closed* is `False`, the last point in *points* is not returned.

Return type list of tuple of number

imgaug.augmentables.utils.**interpolate_points_by_max_distance**(*points*, *max_distance*, *closed=True*)

Interpolate points with distance `d` on a line string.

For a list of points `A, B, C`, if the distance between `A` and `B` is greater than *max_distance*, it will place at least one point between `A` and `B` at `A + max_distance * (B - A)`. Multiple points can be placed between the two points if they are far enough away from each other. The process is repeated for `B` and `C`.

Parameters

- **points** (*iterable of iterable of number*) – Points on the line segments, each one given as `(x,y)` coordinates. They are assumed to form one connected line string.
- **max_distance** (*number*) – Maximum distance between any two points in the result.
- **closed** (*bool, optional*) – If `True` the output contains the last point in *points*. Otherwise it does not (but it will contain the interpolated points leading to the last point).

Returns Coordinates of *points*, with interpolated points added to the iterable. If *closed* is `False`, the last point in *points* is not returned.

Return type list of tuple of number

imgaug.augmentables.utils.**normalize_shape**(*shape*)

Normalize a shape `tuple` or `array` to a shape `tuple`.

Parameters **shape** (*tuple of int or ndarray*) – The input to normalize. May optionally be an array.

Returns Shape `tuple`.

Return type tuple of int

imgaug.augmentables.utils.**project_coords**(*coords*, *from_shape*, *to_shape*)

Project coordinates from one image shape to another.

This performs a relative projection, e.g. a point at `60%` of the old image width will be at `60%` of the new image width after projection.

Parameters

- **coords** (*ndarray or list of tuple of number*) – Coordinates to project. Either an `(N,2)` numpy array or a `list` containing `(x,y)` coordinate `tuple` s.
- **from_shape** (*tuple of int or ndarray*) – Old image shape.
- **to_shape** (*tuple of int or ndarray*) – New image shape.

Returns Projected coordinates as `(N,2) float32` numpy array.

Return type ndarray

13.16 imgaug.augmenters.meta

Augmenters that don't apply augmentations themselves, but are needed for meta usage.

Do not import directly from this file, as the categorization is not final. Use instead

```
from imgaug import augmenters as iaa
```

and then e.g.

```
seq = iaa.Sequential([...])
```

List of augmenters:

- Augmenter (base class for all augmenters)
- Sequential
- SomeOf
- OneOf
- Sometimes
- WithChannels
- Noop
- Lambda
- AssertLambda
- AssertShape
- ChannelShuffle

Note: WithColorspace is in `color.py`.

class imgaug.augmenters.meta.**AssertLambda**(*func_images=None*, *func_heatmaps=None*, *func_segmentation_maps=None*, *func_keypoints=None*, *func_polygons=None*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Lambda*

Assert conditions based on lambda-function to be the case for input data.

This augmenter applies a lambda function to each image or other input. The lambda function must return `True` or `False`. If `False` is returned, an assertion error is produced.

This is useful to ensure that generic assumption about the input data are actually the case and error out early otherwise.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; tested
* ``uint32``: yes; tested
* ``uint64``: yes; tested
* ``int8``: yes; tested
* ``int16``: yes; tested
* ``int32``: yes; tested
* ``int64``: yes; tested
* ``float16``: yes; tested
* ``float32``: yes; tested
* ``float64``: yes; tested
* ``float128``: yes; tested
* ``bool``: yes; tested
```

Parameters

- **func_images** (*None or callable, optional*) – The function to call for each batch of images. It must follow the form:

```
function(images, random_state, parents, hooks)
```

and return either `True` (valid input) or `False` (invalid input). It essentially re-uses the interface of `imgaug.augmenters.meta.Augmenter._augment_images()`.

- **func_heatmaps** (*None or callable, optional*) – The function to call for each batch of heatmaps. It must follow the form:

```
function(heatmaps, random_state, parents, hooks)
```

and return either `True` (valid input) or `False` (invalid input). It essentially re-uses the interface of `imgaug.augmenters.meta.Augmenter._augment_heatmaps()`.

- **func_segmentation_maps** (*None or callable, optional*) – The function to call for each batch of segmentation maps. It must follow the form:

```
function(segmaps, random_state, parents, hooks)
```

and return either `True` (valid input) or `False` (invalid input). It essentially re-uses the interface of `imgaug.augmenters.meta.Augmenter._augment_segmentation_maps()`.

- **func_keypoints** (*None or callable, optional*) – The function to call for each batch of keypoints. It must follow the form:

```
function(keypoints_on_images, random_state, parents, hooks)
```

and return either `True` (valid input) or `False` (invalid input). It essentially re-uses the interface of `imgaug.augmenters.meta.Augmenter._augment_keypoints()`.

- **func_polygons** (*None or callable, optional*) – The function to call for each batch of polygons. It must follow the form:

```
function(polygons_on_images, random_state, parents, hooks)
```

and return either `True` (valid input) or `False` (invalid input). It essentially re-uses the interface of `imgaug.augmenters.meta.Augmenter._augment_polygons()`.

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.

Continued on next page

Table 49 – continued from previous page

augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ...[, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters	

class imgaug.augmenters.meta.**AssertShape**(*shape*, *check_images=True*, *check_heatmaps=True*, *check_segmentation_maps=True*, *check_keypoints=True*, *check_polygons=True*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Lambda*

Assert that inputs have a specified shape.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; tested
* ``uint32``: yes; tested
* ``uint64``: yes; tested
* ``int8``: yes; tested
* ``int16``: yes; tested
* ``int32``: yes; tested
* ``int64``: yes; tested
* ``float16``: yes; tested
* ``float32``: yes; tested
* ``float64``: yes; tested
* ``float128``: yes; tested
* ``bool``: yes; tested
```

Parameters

- **shape** (*tuple*) – The expected shape, given as a `tuple`. The number of entries in the `tuple` must match the number of dimensions, i.e. it must contain four entries for `(N, H, W, C)`. If only a single entity is augmented, e.g. via *imgaug.augmenters.meta.Augmenter.augment_image()*, then `N` is `1` in the input to this augmenter. Images that don't have a channel axis will automatically have one assigned, i.e. `C` is at least `1`. For each component of the `tuple` one of the following datatypes may be used:
  - If a component is `None`, any value for that dimensions is accepted.
  - If a component is `int`, exactly that value (and no other one) will be accepted for that dimension.
  - If a component is a `tuple` of two `int` s with values `a` and `b`, only a value within the interval `[a, b)` will be accepted for that dimension.
  - If an entry is a `list` of `int` s, only a value from that `list` will be accepted for that dimension.
- **check_images** (*bool, optional*) – Whether to validate input images via the given shape.
- **check_heatmaps** (*bool, optional*) – Whether to validate input heatmaps via the given shape. The number of heatmaps will be verified as `N`. For each *imgaug.augmentables.heatmaps.HeatmapsOnImage* instance its array's height and width will be verified as `H` and `W`, but not the channel count.
- **check_segmentation_maps** (*bool, optional*) – Whether to validate input segmentation maps via the given shape. The number of segmentation maps will be verified as `N`. For each *imgaug.augmentables.segmaps.SegmentationMapOnImage* instance its array's height and width will be verified as `H` and `W`, but not the channel count.
- **check_keypoints** (*bool, optional*) – Whether to validate input keypoints via the given shape. This will check (a) the number of keypoints and (b) for each *imgaug.augmentables.*

kps.KeypointsOnImage instance the `.shape` attribute, i.e. the shape of the corresponding image.

- **check_polygons** (*bool, optional*) – Whether to validate input keypoints via the given shape. This will check (a) the number of polygons and (b) for each *imgaug.augmentables.polys.PolygonsOnImage* instance the `.shape` attribute, i.e. the shape of the corresponding image.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> seq = iaa.Sequential([
>>>     iaa.AssertShape((None, 32, 32, 3)),
>>>     iaa.Fliplr(0.5)
>>> ])
```

Verify first for each image batch if it contains a variable number of `32x32` images with `3` channels each. Only if that check succeeds, the horizontal flip will be executed. Otherwise an assertion error will be raised.

```
>>> seq = iaa.Sequential([
>>>     iaa.AssertShape((None, (32, 64), 32, [1, 3])),
>>>     iaa.Fliplr(0.5)
>>> ])
```

Similar to the above example, but now the height may be in the interval `[32, 64)` and the number of channels may be either `1` or `3`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.

Continued on next page

Table 50 – continued from previous page

augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.meta.**Augmenter**(*name=None*, *deterministic=False*, *random_state=None*)

Bases: object

Base class for Augmenter objects. All augmenters derive from this class.

Parameters

- **name** (*None or str, optional*) – Name given to the Augmenter instance. This name is used when converting the instance to a string, e.g. for print statements. It is also used for find, remove or similar operations on augmenters with children. If None, UnnamedX will be used as the name, where X is the Augmenter's class name.
- **deterministic** (*bool, optional*) – Whether the augmenter instance's random state will be saved before augmenting a batch and then reset to that initial saved state after the augmentation was finished. I.e. if set to True, each batch will be augmented in the same

way (e.g. first image might always be flipped horizontally, second image will never be flipped etc.). This is useful when you want to transform multiple batches in the same way, or when you want to augment images and corresponding data (e.g. keypoints or segmentation maps) on these images. Usually, there is no need to set this variable by hand. Instead, instantiate the augmenter and then use `imgaug.augmenters.Augmenter.to_deterministic()`.

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – The RNG (random number generator) to use for this augmenter. Setting this parameter allows to control/influence the random number sampling of the augmenter. Usually, there is no need to set this parameter.
  - If `None`: The global RNG is used (shared by all augmenters).
  - If `int`: The value will be used as a seed for a new *imgaug.random.RNG* instance.
  - If *imgaug.random.RNG*: The `RNG` instance will be used without changes.
  - If `imgaug.random.Generator`: A new *imgaug.random.RNG* instance will be created, containing that generator.
  - If `imgaug.random.bit_generator.BitGenerator`: Will be wrapped in a `imgaug.random.Generator`. Then similar behaviour to `imgaug.random.Generator` parameters.
  - If `imgaug.random.SeedSequence`: Will be wrapped in a new bit generator and `imgaug.random.Generator`. Then similar behaviour to `imgaug.random.Generator` parameters.
  - If `imgaug.random.RandomState`: Similar behaviour to `imgaug.random.Generator`. Outdated in numpy 1.17+.

  If a new bit generator has to be created, it will be an instance of `numpy.random.SFC64`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
augment(self[, return_batch, hooks])	Augment a batch.
augment_batch(self, batch[, hooks])	Augment a single batch.
augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ...[, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.

Continued on next page

Table 51 – continued from previous page

copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

augment (*self*, *return_batch=False*, *hooks=None*, ***kwargs*)

Augment a batch.

This method is a wrapper around *imgaug.augmentables.batches.UnnormalizedBatch* and *imgaug.augmenters.meta.Augmenter.augment_batch()*. Hence, it supports the same datatypes as *imgaug.augmentables.batches.UnnormalizedBatch*.

If *return_batch* was set to `False` (the default), the method will return a tuple of augmentables. It will return the same types of augmentables (but in augmented form) as input into the method. This behaviour is partly specific to the python version:

- In **python 3.6+** (if `return_batch=False`):
  - Any number of augmentables may be provided as input.
  - None of the provided named arguments *has to be image* or *images* (but of coarse you *may* provide them).
  - The return order matches the order of the named arguments, e.g. `x_aug, y_aug, z_aug = augment(X=x, Y=y, Z=z)`.
- In **python <3.6** (if `return_batch=False`):

  - One or two augmentables may be used as input, not more than that.
  - One of the input arguments has to be *image* or *images*.
  - The augmented images are *always* returned first, independent of the input argument order, e.g. `a_aug, b_aug = augment(b=b, images=a)`. This also means that the output of the function can only be one of the following three cases: a batch, list/array of images, tuple of images and something (like images + segmentation maps).

If *return_batch* was set to `True`, an instance of *imgaug.augmentables.batches.UnnormalizedBatch* will be returned. The output is the same for all python version and any number or combination of augmentables may be provided.

So, to keep code downward compatible for python <3.6, use one of the following three options:

- Use `batch = augment(images=X, ..., return_batch=True)`.
- Call `images = augment(images=X)`.
- Call `images, other = augment(images=X, <something_else>=Y)`.

All augmentables must be provided as named arguments. E.g. `augment(<array>)` will crash, but `augment(images=<array>)` will work.

Parameters

- **image** (*None or (H,W,C) ndarray or (H,W) ndarray, optional*) – The image to augment. Only this or *images* can be set, not both. If *return_batch* is `False` and the python version is below 3.6, either this or *images* **must** be provided.
- **images** (*None or (N,H,W,C) ndarray or (N,H,W) ndarray or iterable of (H,W,C) ndarray or iterable of (H,W) ndarray, optional*) – The images to augment. Only this or *image* can be set, not both. If *return_batch* is `False` and the python version is below 3.6, either this or *image* **must** be provided.
- **heatmaps** (*None or (N,H,W,C) ndarray or imgaug.augmentables.heatmaps.HeatmapsOnImage or iterable of (H,W,C) ndarray or iterable of imgaug.augmentables.heatmaps.HeatmapsOnImage, optional*) – The heatmaps to augment. If anything else than *imgaug.augmentables.heatmaps.HeatmapsOnImage*, then the number of heatmaps must match the number of images provided via parameter *images*. The number is contained either in `N` or the first iterable's size.
- **segmentation_maps** (*None or (N,H,W) ndarray or imgaug.augmentables.segmaps.SegmentationMapsOnImage or iterable of (H,W) ndarray or iterable of imgaug.augmentables.segmaps.SegmentationMapsOnImage, optional*) – The segmentation maps to augment. If anything else than *imgaug.augmentables.segmaps.SegmentationMapsOnImage*, then the number of segmaps must match the number of images provided via parameter *images*. The number is contained either in `N` or the first iterable's size.
- **keypoints** (*None or list of (N,K,2) ndarray or tuple of number or imgaug.augmentables.kps.Keypoint or iterable of (K,2) ndarray or iterable of tuple of number or iterable of imgaug.augmentables.kps.Keypoint or iterable of imgaug.augmentables.kps.KeypointOnImage or iterable of iterable of tuple of number or iterable of iterable of imgaug.augmentables.kps.Keypoint, optional*) – The keypoints to augment. If a tuple (or iterable(s) of tuple), then iterpreted as `(x,y)` coordinates and must hence contain two numbers. A single tuple represents a single coordinate on one image, an iterable of tuples the coordinates on one image and an iterable of iterable of tuples the coordinates on several images. Analogous if *imgaug.augmentables.kps.Keypoint* instances are used instead of tuples. If an ndarray, then `N` denotes the number of images and `K` the number of keypoints on each image. If anything else than

imgaug.augmentables.kps.KeypointsOnImage is provided, then the number of keypoint groups must match the number of images provided via parameter *images*. The number is contained e.g. in `N` or in case of "iterable of iterable of tuples" in the first iterable's size.

- **bounding_boxes** (*None or (N,B,4) ndarray or tuple of number or imgaug.augmentables.bbs.BoundingBox or imgaug.augmentables.bbs.BoundingBoxesOnImage or iterable of (B,4) ndarray or iterable of tuple of number or iterable of imgaug.augmentables.bbs.BoundingBox or iterable of imgaug.augmentables.bbs.BoundingBoxesOnImage or iterable of iterable of tuple of number or iterable of iterable imgaug.augmentables.bbs.BoundingBox, optional*) – The bounding boxes to augment. This is analogous to the *keypoints* parameter. However, each tuple – and also the last index in case of arrays – has size `4`, denoting the bounding box coordinates `x1`, `y1`, `x2` and `y2`.

- **polygons** (*None or (N,#polys,#points,2) ndarray or imgaug.augmentables.polys.Polygon or imgaug.augmentables.polys.PolygonsOnImage or iterable of (#polys,#points,2) ndarray or iterable of tuple of number or iterable of imgaug.augmentables.kps.Keypoint or iterable of imgaug.augmentables.polys.Polygon or iterable of imgaug.augmentables.polys.PolygonsOnImage or iterable of iterable of (#points,2) ndarray or iterable of iterable of tuple of number or iterable of iterable of imgaug.augmentables.kps.Keypoint or iterable of iterable of imgaug.augmentables.polys.Polygon or iterable of iterable of iterable of tuple of number or iterable of iterable of iterable of tuple of imgaug.augmentables.kps.Keypoint, optional*) – The polygons to augment. This is similar to the *keypoints* parameter. However, each polygon may be made up of several ‘‘(x,y) ‘‘coordinates (three or more are required for valid polygons). The following datatypes will be interpreted as a single polygon on a single image:

  – `imgaug.augmentables.polys.Polygon`

  – `iterable of tuple of number`

  – `iterable of imgaug.augmentables.kps.Keypoint`

  The following datatypes will be interpreted as multiple polygons on a single image:

  – `imgaug.augmentables.polys.PolygonsOnImage`

  – `iterable of imgaug.augmentables.polys.Polygon`

  – `iterable of iterable of tuple of number`

  – `iterable of iterable of imgaug.augmentables.kps.Keypoint`

  – `iterable of iterable of imgaug.augmentables.polys.Polygon`

  The following datatypes will be interpreted as multiple polygons on multiple images:

  – `(N,#polys,#points,2) ndarray`

  – `iterable of (#polys,#points,2) ndarray`

  – `iterable of iterable of (#points,2) ndarray`

  – `iterable of iterable of iterable of tuple of number`

  – `iterable of iterable of iterable of tuple of imgaug.augmentables.kps.Keypoint`

- **line_strings** (*None or (N,#lines,#points,2) ndarray or imgaug.augmentables.lines.LineString or imgaug.augmentables.lines.LineStringOnImage or iterable of (#polys,#points,2) ndarray or iterable of tuple of number or iterable of*

imgaug.augmentables.kps.Keypoint or iterable of imgaug.augmentables.lines.LineString or iterable of imgaug.augmentables.lines.LineStringOnImage or iterable of iterable of (#points,2) ndarray or iterable of iterable of tuple of number or iterable of iterable of imgaug.augmentables.kps.Keypoint or iterable of iterable of imgaug.augmentables.lines.LineString or iterable of iterable of iterable of tuple of number or iterable of iterable of iterable of tuple of imgaug.augmentables.kps.Keypoint, optional) – The line strings to augment. See *polygons*, which behaves similarly.

- **return_batch** (*bool, optional*) – Whether to return an instance of *imgaug.augmentables.batches.UnnormalizedBatch*. If the python version is below 3.6 and more than two augmentables were provided (e.g. images, keypoints and polygons), then this must be set to `True`. Otherwise an error will be raised.
- **hooks** (*None or imgaug.imgaug.HooksImages, optional*) – Hooks object to dynamically interfere with the augmentation process.

Returns If *return_batch* was set to `True`, a instance of `UnnormalizedBatch` will be returned. If *return_batch* was set to `False`, a tuple of augmentables will be returned, e.g. `(augmented images, augmented keypoints)`. The datatypes match the input datatypes of the corresponding named arguments. In python <3.6, augmented images are always the first entry in the returned tuple. In python 3.6+ the order matches the order of the named arguments.

Return type tuple or *imgaug.augmentables.batches.UnnormalizedBatch*

Examples

```
>>> import numpy as np
>>> import imgaug as ia
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Affine(rotate=(-25, 25))
>>> image = np.zeros((64, 64, 3), dtype=np.uint8)
>>> keypoints = [(10, 20), (30, 32)]  # (x,y) coordinates
>>> images_aug, keypoints_aug = aug.augment(
>>>     image=image, keypoints=keypoints)
```

Create a single image and a set of two keypoints on it, then augment both by applying a random rotation between `-25` deg and `+25` deg. The sampled rotation value is automatically aligned between image and keypoints. Note that in python <3.6, augmented images will always be returned first, independent of the order of the named input arguments. So `keypoints_aug, images_aug = aug.augment(keypoints=keypoints, image=image)` would **not** be correct (but in python 3.6+ it would be).

```
>>> import numpy as np
>>> import imgaug as ia
>>> import imgaug.augmenters as iaa
>>> from imgaug.augmentables.bbs import BoundingBox
>>> aug = iaa.Affine(rotate=(-25, 25))
>>> images = [np.zeros((64, 64, 3), dtype=np.uint8),
>>>           np.zeros((32, 32, 3), dtype=np.uint8)]
>>> keypoints = [[(10, 20), (30, 32)],  # KPs on first image
>>>              [(22, 10), (12, 14)]]  # KPs on second image
>>> bbs = [
>>>           [BoundingBox(x1=5, y1=5, x2=50, y2=45)],
>>>           [BoundingBox(x1=4, y1=6, x2=10, y2=15),
>>>            BoundingBox(x1=8, y1=9, x2=16, y2=30)]
```

(continues on next page)

(continued from previous page)

```
>>>         ]  # one BB on first image, two BBs on second image
>>> batch_aug = aug.augment(
>>>     images=images, keypoints=keypoints, bounding_boxes=bbs,
>>>     return_batch=True)
```

Create two images of size `64x64` and `32x32`, two sets of keypoints (each containing two keypoints) and two sets of bounding boxes (the first containing one bounding box, the second two bounding boxes). These augmentables are then augmented by applying random rotations between `-25` deg and `+25` deg to them. The rotation values are sampled by image and aligned between all augmentables on the same image. The method finally returns an instance of *imgaug.augmentables.batches.UnnormalizedBatch* from which the augmented data can be retrieved via `batch_aug.images_aug`, `batch_aug.keypoints_aug`, and `batch_aug.bounding_boxes_aug`. In python 3.6+, *return_batch* can be kept at `False` and the augmented data can be retrieved as `images_aug, keypoints_aug, bbs_aug = augment(...)`.

augment_batch (*self*, *batch*, *hooks=None*)

Augment a single batch.

Parameters

- **batch** (*imgaug.augmentables.batches.Batch or imgaug.augmentables.batches.UnnormalizedBatch*) – A single batch to augment.
- **hooks** (*None or imgaug.HooksImages, optional*) – HooksImages object to dynamically interfere with the augmentation process.

Returns Augmented batch.

Return type *imgaug.augmentables.batches.Batch* or *imgaug.augmentables.batches.UnnormalizedBatch*

augment_batches (*self*, *batches*, *hooks=None*, *background=False*)

Augment multiple batches.

In contrast to other `augment_*` method, this one **yields** batches instead of returning a full list. This is more suited for most training loops.

This method also also supports augmentation on multiple cpu cores, activated via the *background* flag. If the *background* flag is activated, an instance of *imgaug.multicore.Pool* will be spawned using all available logical CPU cores and an `output_buffer_size` of `C*10`, where `C` is the number of logical CPU cores. I.e. a maximum of `C*10` batches will be somewhere in the augmentation pipeline (or waiting to be retrieved by downstream functions) before this method temporarily stops the loading of new batches from *batches*.

Parameters

- **batches** (*imgaug.augmentables.batches.Batch or imgaug.augmentables.batches.UnnormalizedBatch or iterable of imgaug.augmentables.batches.Batch or iterable of imgaug.augmentables.batches.UnnormalizedBatch*) – A single batch or a list of batches to augment.
- **hooks** (*None or imgaug.HooksImages, optional*) – HooksImages object to dynamically interfere with the augmentation process.
- **background** (*bool, optional*) – Whether to augment the batches in background processes. If `True`, hooks can currently not be used as that would require pickling functions. Note that multicore augmentation distributes the batches onto different CPU cores. It does *not* split the data *within* batches. It is therefore *not* sensible to use `background=True` to augment a single batch. Only use it for multiple batches. Note also that multicore

augmentation needs some time to start. It is therefore not recommended to use it for very few batches.

Yields *imgaug.augmentables.batches.Batch or imgaug.augmentables.batches.UnnormalizedBatch or iterable of imgaug.augmentables.batches.Batch or iterable of imgaug.augmentables.batches.UnnormalizedBatch* – Augmented batches.

augment_bounding_boxes(*self*, *bounding_boxes_on_images*, *hooks=None*)

Augment a batch of bounding boxes.

This is the corresponding function to *Augmenter.augment_images()*, just for bounding boxes. Usually you will want to call *Augmenter.augment_images()* with a list of images, e.g. `augment_images([A, B, C])` and then `augment_bounding_boxes()` with the corresponding list of bounding boxes on these images, e.g. `augment_bounding_boxes([Abb, Bbb, Cbb])`, where `Abb` are the bounding boxes on image `A`.

Make sure to first convert the augmenter(s) to deterministic states before augmenting images and their corresponding bounding boxes, e.g. by

```
>>> import imgaug.augmenters as iaa
>>> from imgaug.augmentables.bbs import BoundingBox
>>> from imgaug.augmentables.bbs import BoundingBoxesOnImage
>>> A = B = C = np.ones((10, 10), dtype=np.uint8)
>>> Abb = Bbb = Cbb = BoundingBoxesOnImage([
>>>     BoundingBox(1, 1, 9, 9)], (10, 10))
>>> seq = iaa.Fliplr(0.5)
>>> seq_det = seq.to_deterministic()
>>> imgs_aug = seq_det.augment_images([A, B, C])
>>> bbs_aug = seq_det.augment_bounding_boxes([Abb, Bbb, Cbb])
```

Otherwise, different random values will be sampled for the image and bounding box augmentations, resulting in different augmentations (e.g. images might be rotated by `30deg` and bounding boxes by `-10deg`). Also make sure to call *Augmenter.to_deterministic()* again for each new batch, otherwise you would augment all batches in the same way.

Note that there is also *Augmenter.augment()*, which automatically handles the random state alignment.

Parameters

- **bounding_boxes_on_images** (*imgaug.augmentables.bbs.BoundingBoxesOnImage or list of imgaug.augmentables.bbs.BoundingBoxesOnImage*) – The bounding boxes to augment. Either a single instance of *imgaug.augmentables.bbs.BoundingBoxesOnImage* or a list of such instances, with each one of them containing the bounding boxes of a single image.
- **hooks** (*None or imgaug.imgaug.HooksKeypoints, optional*) – *imgaug.imgaug.HooksKeypoints* object to dynamically interfere with the augmentation process.

Returns Augmented bounding boxes.

Return type *imgaug.augmentables.bbs.BoundingBoxesOnImage* or list of imgaug.augmentables.bbs.BoundingBoxesOnImage

augment_heatmaps(*self*, *heatmaps*, *parents=None*, *hooks=None*)

Augment a batch of heatmaps.

Parameters

- **heatmaps** (*imgaug.augmentables.heatmaps.HeatmapsOnImage or list of imgaug.augmentables.heatmaps.HeatmapsOnImage*) – Heatmap(s) to augment. Either

a single heatmap or a list of heatmaps.

- **parents** (*None or list of imgaug.augmenters.meta.Augmenter, optional*) – Parent augmenters that have previously been called before the call to this function. Usually you can leave this parameter as `None`. It is set automatically for child augmenters.
- **hooks** (*None or imaug.imgaug.HooksHeatmaps, optional*) – *imgaug.imgaug.HooksHeatmaps* object to dynamically interfere with the augmentation process.

Returns Corresponding augmented heatmap(s).

Return type *imgaug.augmentables.heatmaps.HeatmapsOnImage* or list of imgaug.augmentables.heatmaps.HeatmapsOnImage

augment_image(*self*, *image*, *hooks=None*)

Augment a single image.

Parameters

- **image** (*(H,W,C) ndarray or (H,W) ndarray*) – The image to augment. Channel-axis is optional, but expected to be the last axis if present. In most cases, this array should be of dtype `uint8`, which is supported by all augmenters. Support for other dtypes varies by augmenter – see the respective augmenter-specific documentation for more details.
- **hooks** (*None or imgaug.HooksImages, optional*) – HooksImages object to dynamically interfere with the augmentation process.

Returns **img** – The corresponding augmented image.

Return type ndarray

augment_images(*self*, *images*, *parents=None*, *hooks=None*)

Augment a batch of images.

Parameters

- **images** (*(N,H,W,C) ndarray or (N,H,W) ndarray or list of (H,W,C) ndarray or list of (H,W) ndarray*) – Images to augment. The input can be a list of numpy arrays or a single array. Each array is expected to have shape `(H, W, C)` or `(H, W)`, where `H` is the height, `W` is the width and `C` are the channels. The number of channels may differ between images. If a list is provided, the height, width and channels may differ between images within the provided batch. In most cases, the image array(s) should be of dtype `uint8`, which is supported by all augmenters. Support for other dtypes varies by augmenter – see the respective augmenter-specific documentation for more details.
- **parents** (*None or list of imgaug.augmenters.Augmenter, optional*) – Parent augmenters that have previously been called before the call to this function. Usually you can leave this parameter as `None`. It is set automatically for child augmenters.
- **hooks** (*None or imgaug.imgaug.HooksImages, optional*) – *imgaug.imgaug.HooksImages* object to dynamically interfere with the augmentation process.

Returns Corresponding augmented images. If the input was an `ndarray`, the output is also an `ndarray`, unless the used augmentations have led to different output image sizes (as can happen in e.g. cropping).

Return type ndarray or list

Examples

```
>>> import imgaug.augmenters as iaa
>>> import numpy as np
>>> aug = iaa.GaussianBlur((0.0, 3.0))
>>> # create empty example images
>>> images = np.zeros((2, 64, 64, 3), dtype=np.uint8)
>>> images_aug = aug.augment_images(images)
```

Create 2 empty (i.e. black) example numpy images and apply gaussian blurring to them.

augment_keypoints(*self*, *keypoints_on_images*, *parents=None*, *hooks=None*)

Augment a batch of keypoints/landmarks.

This is the corresponding function to *Augmenter.augment_images()*, just for keypoints/landmarks (i.e. points on images). Usually you will want to call *Augmenter.augment_images()* with a list of images, e.g. `augment_images([A, B, C])` and then `augment_keypoints()` with the corresponding list of keypoints on these images, e.g. `augment_keypoints([Ak, Bk, Ck])`, where `Ak` are the keypoints on image `A`.

Make sure to first convert the augmenter(s) to deterministic states before augmenting images and their corresponding keypoints, e.g. by

```
>>> import imgaug.augmenters as iaa
>>> from imgaug.augmentables.kps import Keypoint
>>> from imgaug.augmentables.kps import KeypointsOnImage
>>> A = B = C = np.zeros((10, 10), dtype=np.uint8)
>>> Ak = Bk = Ck = KeypointsOnImage([Keypoint(2, 2)], (10, 10))
>>> seq = iaa.Fliplr(0.5)
>>> seq_det = seq.to_deterministic()
>>> imgs_aug = seq_det.augment_images([A, B, C])
>>> kps_aug = seq_det.augment_keypoints([Ak, Bk, Ck])
```

Otherwise, different random values will be sampled for the image and keypoint augmentations, resulting in different augmentations (e.g. images might be rotated by `30deg` and keypoints by `-10deg`). Also make sure to call *Augmenter.to_deterministic()* again for each new batch, otherwise you would augment all batches in the same way.

Note that there is also *Augmenter.augment()*, which automatically handles the random state alignment.

Parameters

- **keypoints_on_images** (*imgaug.augmentables.kps.KeypointsOnImage or list of imgaug.augmentables.kps.KeypointsOnImage*) – The keypoints/landmarks to augment. Either a single instance of *imgaug.augmentables.kps.KeypointsOnImage* or a list of such instances. Each instance must contain the keypoints of a single image.
- **parents** (*None or list of imgaug.augmenters.meta.Augmenter, optional*) – Parent augmenters that have previously been called before the call to this function. Usually you can leave this parameter as `None`. It is set automatically for child augmenters.
- **hooks** (*None or imgaug.imgaug.HooksKeypoints, optional*) – *imgaug.imgaug.HooksKeypoints* object to dynamically interfere with the augmentation process.

Returns Augmented keypoints.

Return type *imgaug.augmentables.kps.KeypointsOnImage* or list of imgaug.augmentables.kps.KeypointsOnImage

augment_line_strings(*self*, *line_strings_on_images*, *parents=None*, *hooks=None*)
Augment a batch of line strings.

This is the corresponding function to `Augmenter.augment_images()`, just for line strings. Usually you will want to call *Augmenter.augment_images()* with a list of images, e.g. `augment_images([A, B, C])` and then `augment_line_strings()` with the corresponding list of line strings on these images, e.g. `augment_line_strings([A_line, B_line, C_line])`, where `A_line` are the line strings on image `A`.

Make sure to first convert the augmenter(s) to deterministic states before augmenting images and their corresponding line strings, e.g. by

```
>>> import imgaug.augmenters as iaa
>>> from imgaug.augmentables.lines import LineString
>>> from imgaug.augmentables.lines import LineStringsOnImage
>>> A = B = C = np.ones((10, 10), dtype=np.uint8)
>>> A_line = B_line = C_line = LineStringsOnImage(
>>>     [LineString([(0, 0), (1, 0), (1, 1), (0, 1)])],
>>>     shape=(10, 10))
>>> seq = iaa.Fliplr(0.5)
>>> seq_det = seq.to_deterministic()
>>> imgs_aug = seq_det.augment_images([A, B, C])
>>> lines_aug = seq_det.augment_line_strings([A_line, B_line, C_line])
```

Otherwise, different random values will be sampled for the image and line string augmentations, resulting in different augmentations (e.g. images might be rotated by `30deg` and line strings by `-10deg`). Also make sure to call `to_deterministic()` again for each new batch, otherwise you would augment all batches in the same way.

Note that there is also *Augmenter.augment()*, which automatically handles the random state alignment.

Parameters

- **line_strings_on_images** (*imgaug.augmentables.lines.LineStringsOnImage or list of imgaug.augmentables.lines.LineStringsOnImage*) – The line strings to augment. Either a single instance of *imgaug.augmentables.lines.LineStringsOnImage* or a list of such instances, with each one of them containing the line strings of a single image.
- **parents** (*None or list of imgaug.augmenters.meta.Augmenter, optional*) – Parent augmenters that have previously been called before the call to this function. Usually you can leave this parameter as None. It is set automatically for child augmenters.
- **hooks** (*None or imgaug.imgaug.HooksKeypoints, optional*) – *imgaug.imgaug.HooksKeypoints* object to dynamically interfere with the augmentation process.

Returns Augmented line strings.

Return type *imgaug.augmentables.lines.LineStringsOnImage* or list of imgaug.augmentables.lines.LineStringsOnImage

augment_polygons(*self*, *polygons_on_images*, *parents=None*, *hooks=None*)
Augment a batch of polygons.

This is the corresponding function to *Augmenter.augment_images()*, just for polygons. Usually you will want to call `Augmenter.augment_images()` with a list of images, e.g. `augment_images([A, B, C])` and then `augment_polygons()` with the corresponding list of polygons on these images, e.g. `augment_polygons([A_poly, B_poly, C_poly])`, where `A_poly` are the polygons on image `A`.

Make sure to first convert the augmenter(s) to deterministic states before augmenting images and their corresponding polygons, e.g. by

```
>>> import imgaug.augmenters as iaa
>>> from imgaug.augmentables.polys import Polygon, PolygonsOnImage
>>> A = B = C = np.ones((10, 10), dtype=np.uint8)
>>> Apoly = Bpoly = Cpoly = PolygonsOnImage(
>>>     [Polygon([(0, 0), (1, 0), (1, 1), (0, 1)])],
>>>     shape=(10, 10))
>>> seq = iaa.Fliplr(0.5)
>>> seq_det = seq.to_deterministic()
>>> imgs_aug = seq_det.augment_images([A, B, C])
>>> polys_aug = seq_det.augment_polygons([Apoly, Bpoly, Cpoly])
```

Otherwise, different random values will be sampled for the image and polygon augmentations, resulting in different augmentations (e.g. images might be rotated by `30deg` and polygons by `-10deg`). Also make sure to call `to_deterministic()` again for each new batch, otherwise you would augment all batches in the same way.

Note that there is also *Augmenter.augment()*, which automatically handles the random state alignment.

Parameters

- **polygons_on_images** (*imgaug.augmentables.polys.PolygonsOnImage or list of imgaug.augmentables.polys.PolygonsOnImage*) – The polygons to augment. Either a single instance of *imgaug.augmentables.polys.PolygonsOnImage* or a list of such instances, with each one of them containing the polygons of a single image.
- **parents** (*None or list of imgaug.augmenters.meta.Augmenter, optional*) – Parent augmenters that have previously been called before the call to this function. Usually you can leave this parameter as `None`. It is set automatically for child augmenters.
- **hooks** (*None or imgaug.imgaug.HooksKeypoints, optional*) – *imgaug.imgaug.HooksKeypoints* object to dynamically interfere with the augmentation process.

Returns Augmented polygons.

Return type *imgaug.augmentables.polys.PolygonsOnImage* or list of imgaug.augmentables.polys.PolygonsOnImage

augment_segmentation_maps(*self*, *segmaps*, *parents=None*, *hooks=None*)

Augment a batch of segmentation maps.

Parameters

- **segmaps** (*imgaug.augmentables.segmaps.SegmentationMapsOnImage or list of imgaug.augmentables.segmaps.SegmentationMapsOnImage*) – Segmentation map(s) to augment. Either a single segmentation map or a list of segmentation maps.
- **parents** (*None or list of imgaug.augmenters.meta.Augmenter, optional*) – Parent augmenters that have previously been called before the call to this function. Usually you can leave this parameter as `None`. It is set automatically for child augmenters.
- **hooks** (*None or imgaug.HooksHeatmaps, optional*) – *imgaug.imgaug.HooksHeatmaps* object to dynamically interfere with the augmentation process.

Returns Corresponding augmented segmentation map(s).

Return type *imgaug.augmentables.segmaps.SegmentationMapsOnImage* or list of imgaug.augmentables.segmaps.SegmentationMapsOnImage

copy(*self*)
Create a shallow copy of this Augmenter instance.

Returns Shallow copy of this Augmenter instance.

Return type *imgaug.augmenters.meta.Augmenter*

copy_random_state(*self*, *source*, *recursive=True*, *matching='position'*, *matching_tolerant=True*, *copy_determinism=False*)
Copy the RNGs from a source augmenter sequence.

Parameters

- **source** (*imgaug.augmenters.meta.Augmenter*) – See *imgaug.augmenters.meta.Augmenter.copy_random_state_()*.
- **recursive** (*bool, optional*) – See *imgaug.augmenters.meta.Augmenter.copy_random_state_()*.
- **matching** (*{'position', 'name'}, optional*) – See *imgaug.augmenters.meta.Augmenter.copy_random_state_()*.
- **matching_tolerant** (*bool, optional*) – See *imgaug.augmenters.meta.Augmenter.copy_random_state_()*.
- **copy_determinism** (*bool, optional*) – See *imgaug.augmenters.meta.Augmenter.copy_random_state_()*.

Returns Copy of the augmenter itself (with copied RNGs).

Return type *imgaug.augmenters.meta.Augmenter*

copy_random_state_(*self*, *source*, *recursive=True*, *matching='position'*, *matching_tolerant=True*, *copy_determinism=False*)
Copy the RNGs from a source augmenter sequence (in-place).

Note: The source augmenters are not allowed to use the global RNG. Call *imgaug.augmenters.meta.Augmenter.localize_random_state_()* once on the source to localize all random states.

Parameters

- **source** (*imgaug.augmenters.meta.Augmenter*) – The source augmenter(s) from where to copy the RNG(s). The source may have children (e.g. the source can be a *imgaug.augmenters.meta.Sequential*).
- **recursive** (*bool, optional*) – Whether to copy the RNGs of the source augmenter *and* all of its children (`True`) or just the source augmenter (`False`).
- **matching** (*{'position', 'name'}, optional*) – Defines the matching mode to use during recursive copy. This is used to associate source augmenters with target augmenters. If `position` then the target and source sequences of augmenters are turned into flattened lists and are associated based on their list indices. If `name` then the target and source augmenters are matched based on their names (i.e. `augmenter.name`).
- **matching_tolerant** (*bool, optional*) – Whether to use tolerant matching between source and target augmenters. If set to `False`: Name matching will raise an exception for any target augmenter which's name does not appear among the source augmenters. Position matching will raise an exception if source and target augmenter have an unequal number of children.

- **copy_determinism** (*bool, optional*) – Whether to copy the `deterministic` attributes from source to target augmenters too.

Returns The augmenter itself.

Return type *imgaug.augmenters.meta.Augmenter*

deepcopy(*self*)
Create a deep copy of this Augmenter instance.

Returns Deep copy of this Augmenter instance.

Return type *imgaug.augmenters.meta.Augmenter*

draw_grid(*self*, *images*, *rows*, *cols*)
Augment images and draw the results as a single grid-like image.

This method applies this augmenter to the provided images and returns a grid image of the results. Each cell in the grid contains a single augmented version of an input image.

If multiple input images are provided, the row count is multiplied by the number of images and each image gets its own row. E.g. for `images = [A, B]`, `rows=2`, `cols=3`:

```
A A A
B B B
A A A
B B B
```

for `images = [A]`, `rows=2`, `cols=3`:

```
A A A
A A A
```

Parameters

- **images** (*(N,H,W,3) ndarray or (H,W,3) ndarray or (H,W) ndarray or list of (H,W,3) ndarray or list of (H,W) ndarray*) – List of images to augment and draw in the grid. If a list, then each element is expected to have shape `(H, W)` or `(H, W, 3)`. If a single array, then it is expected to have shape `(N, H, W, 3)` or `(H, W, 3)` or `(H, W)`.
- **rows** (*int*) – Number of rows in the grid. If `N` input images are given, this value will automatically be multiplied by `N` to create rows for each image.
- **cols** (*int*) – Number of columns in the grid.

Returns The generated grid image with augmented versions of the input images. Here, `Hg` and `Wg` reference the output size of the grid, and *not* the sizes of the input images.

Return type (Hg, Wg, 3) ndarray

find_augmenters(*self*, *func*, *parents=None*, *flat=True*)
Find augmenters that match a condition.

This function will compare this augmenter and all of its children with a condition. The condition is a lambda function.

Parameters

- **func** (*callable*) – A function that receives a *imgaug.augmenters.meta.Augmenter* instance and a list of parent *imgaug.augmenters.meta.Augmenter* instances and must return `True`, if that augmenter is valid match or `False` otherwise.

- **parents** (*None or list of imgaug.augmenters.meta.Augmenter, optional*) – List of parent augmenters. Intended for nested calls and can usually be left as `None`.
- **flat** (*bool, optional*) – Whether to return the result as a flat list (`True`) or a nested list (`False`). In the latter case, the nesting matches each augmenters position among the children.

Returns Nested list if *flat* was set to `False`. Flat list if *flat* was set to `True`.

Return type list of imgaug.augmenters.meta.Augmenter

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Sequential([
>>>     iaa.Fliplr(0.5, name="fliplr"),
>>>     iaa.Flipud(0.5, name="flipud")
>>> ])
>>> print(aug.find_augmenters(lambda a, parents: a.name == "fliplr"))
```

Return the first child augmenter (`Fliplr` instance).

find_augmenters_by_name (*self*, *name*, *regex=False*, *flat=True*)

Find augmenter(s) by name.

Parameters

- **name** (*str*) – Name of the augmenter(s) to search for.
- **regex** (*bool, optional*) – Whether *name* parameter is a regular expression.
- **flat** (*bool, optional*) – See *imgaug.augmenters.meta.Augmenter.find_augmenters()*.

Returns **augmenters** – Nested list if *flat* was set to `False`. Flat list if *flat* was set to `True`.

Return type list of imgaug.augmenters.meta.Augmenter

find_augmenters_by_names (*self*, *names*, *regex=False*, *flat=True*)

Find augmenter(s) by names.

Parameters

- **names** (*list of str*) – Names of the augmenter(s) to search for.
- **regex** (*bool, optional*) – Whether *names* is a list of regular expressions. If it is, an augmenter is considered a match if *at least* one of these expressions is a match.
- **flat** (*boolean, optional*) – See *imgaug.augmenters.meta.Augmenter.find_augmenters()*.

Returns **augmenters** – Nested list if *flat* was set to `False`. Flat list if *flat* was set to `True`.

Return type list of imgaug.augmenters.meta.Augmenter

get_all_children (*self*, *flat=False*)

Get all children of this augmenter as a list.

If the augmenter has no children, the returned list is empty.

Parameters **flat** (*bool*) – If set to `True`, the returned list will be flat.

Returns The children as a nested or flat list.

Return type list of imgaug.augmenters.meta.Augmenter

get_children_lists(*self*)
Get a list of lists of children of this augmenter.

For most augmenters, the result will be a single empty list. For augmenters with children it will often be a list with one sublist containing all children. In some cases the augmenter will contain multiple distinct lists of children, e.g. an if-list and an else-list. This will lead to a result consisting of a single list with multiple sublists, each representing the respective sublist of children.

E.g. for an if/else-augmenter that executes the children `A1`, `A2` if a condition is met and otherwise executes the children `B1`, `B2`, `B3` the result will be `[[A1, A2], [B1, B2, B3]]`.

IMPORTANT: While the topmost list may be newly created, each of the sublist must be editable inplace resulting in a changed children list of the augmenter. E.g. if an Augmenter `IfElse(condition, [A1, A2], [B1, B2, B3])` returns `[[A1, A2], [B1, B2, B3]]` for a call to *imgaug.augmenters.meta.Augmenter.get_children_lists()* and `A2` is removed inplace from `[A1, A2]`, then the children lists of `IfElse(...)` must also change to `[A1]`, `[B1, B2, B3]`. This is used in `imgaug.augmeneters.meta.Augmenter.remove_augmenters_inplace()`.

Returns One or more lists of child augmenter. Can also be a single empty list.

Return type list of list of imgaug.augmenters.meta.Augmenter

get_parameters(*self*)

localize_random_state(*self*, *recursive=True*)
Assign augmenter-specific RNGs to this augmenter and its children.

See *Augmenter.localize_random_state_()* for more details.

Parameters **recursive** (*bool, optional*) – See *imgaug.augmenters.meta.Augmenter.localize_random_state_()*.

Returns Copy of the augmenter and its children, with localized RNGs.

Return type *imgaug.augmenters.meta.Augmenter*

localize_random_state_(*self*, *recursive=True*)
Assign augmenter-specific RNGs to this augmenter and its children.

This method iterates over this augmenter and all of its children and replaces any pointer to the global RNG with a new local (i.e. augmenter-specific) RNG.

A random number generator (RNG) is used for the sampling of random values. The global random number generator exists exactly once throughout the library and is shared by many augmenters. A local RNG (usually) exists within exactly one augmenter and is only used by that augmenter.

Usually there is no need to change global into local RNGs. The only noteworthy exceptions are

- Whenever you want to use determinism (so that the global RNG is not accidentally reverted).
- Whenever you want to copy RNGs from one augmenter to another. (Copying the global RNG would usually not be useful. Copying the global RNG from augmenter A to B, then executing A and then B would result in B's (global) RNG's state having already changed because of A's sampling. So the samples of A and B would differ.)

The case of determinism is handled automatically by *imgaug.augmenters.meta.Augmenter.to_deterministic()*. Only when you copy RNGs (via *imgaug.augmenters.meta.Augmenter.copy_random_state()*), you need to call this function first.

Parameters **recursive** (*bool, optional*) – Whether to localize the RNGs of the augmenter's children too.

Returns Returns itself (with localized RNGs).

Return type *imgaug.augmenters.meta.Augmenter*

pool (*self*, *processes=None*, *maxtasksperchild=None*, *seed=None*)
Create a pool used for multicore augmentation.

Parameters

- **processes** (*None or int, optional*) – Same as in `imgaug.multicore.Pool.__init__()`. The number of background workers. If `None`, the number of the machine's CPU cores will be used (this counts hyperthreads as CPU cores). If this is set to a negative value `p`, then `P - abs(p)` will be used, where `P` is the number of CPU cores. E.g. `-1` would use all cores except one (this is useful to e.g. reserve one core to feed batches to the GPU).
- **maxtasksperchild** (*None or int, optional*) – Same as for `imgaug.multicore.Pool.__init__()`. The number of tasks done per worker process before the process is killed and restarted. If `None`, worker processes will not be automatically restarted.
- **seed** (*None or int, optional*) – Same as for `imgaug.multicore.Pool.__init__()`. The seed to use for child processes. If `None`, a random seed will be used.

Returns Pool for multicore augmentation.

Return type *imgaug.multicore.Pool*

Examples

```
>>> import numpy as np
>>> import imgaug as ia
>>> import imgaug.augmenters as iaa
>>> from imgaug.augmentables.batches import Batch
>>>
>>> aug = iaa.Add(1)
>>> images = np.zeros((16, 128, 128, 3), dtype=np.uint8)
>>> batches = [Batch(images=np.copy(images)) for _ in range(100)]
>>> with aug.pool(processes=-1, seed=2) as pool:
>>>     batches_aug = pool.map_batches(batches, chunksize=8)
>>> print(np.sum(batches_aug[0].images_aug[0]))
49152
```

Create `100` batches of empty images. Each batch contains `16` images of size `128x128`. The batches are then augmented on all CPU cores except one (`processes=-1`). After augmentation, the sum of pixel values from the first augmented image is printed.

```
>>> import numpy as np
>>> import imgaug as ia
>>> import imgaug.augmenters as iaa
>>> from imgaug.augmentables.batches import Batch
>>>
>>> aug = iaa.Add(1)
>>> images = np.zeros((16, 128, 128, 3), dtype=np.uint8)
>>> def generate_batches():
>>>     for _ in range(100):
>>>         yield Batch(images=np.copy(images))
>>>
```

(continues on next page)

(continued from previous page)

```
>>> with aug.pool(processes=-1, seed=2) as pool:
>>>     batches_aug = pool.imap_batches(generate_batches(), chunksize=8)
>>>     batch_aug = next(batches_aug)
>>>     print(np.sum(batch_aug.images_aug[0]))
49152
```

Same as above. This time, a generator is used to generate batches of images. Again, the first augmented image's sum of pixels is printed.

remove_augmenters (*self*, *func*, *copy=True*, *noop_if_topmost=True*)

Remove this augmenter or children that match a condition.

Parameters

- **func** (*callable*) – Condition to match per augmenter. The function must expect the augmenter itself and a list of parent augmenters and returns `True` if that augmenter is supposed to be removed, or `False` otherwise. E.g. `lambda a, parents: a.name == "fliplr" and len(parents) == 1` removes an augmenter with name `fliplr` if it is the direct child of the augmenter upon which `remove_augmenters()` was initially called.
- **copy** (*bool, optional*) – Whether to copy this augmenter and all if its children before removing. If `False`, removal is performed in-place.
- **noop_if_topmost** (*bool, optional*) – If `True` and the condition (lambda function) leads to the removal of the topmost augmenter (the one this function is called on initially), then that topmost augmenter will be replaced by an instance of *imgaug.augmenters.meta.Noop* (i.e. an augmenter that doesn't change its inputs). If `False`, `None` will be returned in these cases. This can only be `False` if copy is set to `True`.

Returns This augmenter after the removal was performed. `None` is returned if the condition was matched for the topmost augmenter, *copy* was set to `True` and *noop_if_topmost* was set to `False`.

Return type *imgaug.augmenters.meta.Augmenter* or None

Examples

```
>>> import imgaug.augmenters as iaa
>>> seq = iaa.Sequential([
>>>     iaa.Fliplr(0.5, name="fliplr"),
>>>     iaa.Flipud(0.5, name="flipud"),
>>> ])
>>> seq = seq.remove_augmenters(lambda a, parents: a.name == "fliplr")
```

This removes the augmenter `Fliplr` from the `Sequential` object's children.

remove_augmenters_inplace (*self*, *func*, *parents=None*)

Remove in-place children of this augmenter that match a condition.

This is functionally identical to `imgaug.augmenters.meta.remove_augmenters()` with `copy=False`, except that it does not affect the topmost augmenter (the one on which this function is initially called on).

Parameters

- **func** (*callable*) – See *imgaug.augmenters.meta.Augmenter.remove_augmenters()*.

- **parents** (*None or list of imgaug.augmenters.meta.Augmenter, optional*) – List of parent *imgaug.augmenters.meta.Augmenter* instances that lead to this augmenter. If `None`, an empty list will be used. This parameter can usually be left empty and will be set automatically for children.

Examples

```
>>> import imgaug.augmenters as iaa
>>> seq = iaa.Sequential([
>>>     iaa.Fliplr(0.5, name="fliplr"),
>>>     iaa.Flipud(0.5, name="flipud"),
>>> ])
>>> seq.remove_augmenters_inplace(lambda a, parents: a.name == "fliplr")
```

This removes the augmenter `Fliplr` from the `Sequential` object's children.

reseed(*self*, *random_state=None*, *deterministic_too=False*)

Reseed this augmenter and all of its children.

This method assigns a new random number generator to the augmenter and all of its children (if it has any). The new random number generator is derived from the provided one or from the global random number generator.

If this augmenter or any child augmenter had a random numer generator that pointed to the global random state, it will automatically be replaced with a local random state. This is similar to what *imgaug.augmenters.meta.Augmenter.localize_random_state()* does.

This method is useful when augmentations are run in the background (i.e. on multiple cores). It should be called before sending this *imgaug.augmenters.meta.Augmenter* instance to a background worker or once within each worker with different seeds (i.e., if `N` workers are used, the function should be called `N` times). Otherwise, all background workers will use the same seeds and therefore apply the same augmentations. Note that *Augmenter.augment_batches()* and *Augmenter.pool()* already do this automatically.

Parameters

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – A seed or random number generator that is used to derive new random number generators for this augmenter and its children. If an `int` is provided, it will be interpreted as a seed. If `None` is provided, the global random number generator will be used.
- **deterministic_too** (*bool, optional*) – Whether to also change the seed of an augmenter `A`, if `A` is deterministic. This is the case both when this augmenter object is `A` or one of its children is `A`.

show_grid(*self*, *images*, *rows*, *cols*)

Augment images and plot the results as a single grid-like image.

This calls *imgaug.augmenters.meta.Augmenter.draw_grid()* and simply shows the results. See that method for details.

Parameters

- **images** (*(N,H,W,3) ndarray or (H,W,3) ndarray or (H,W) ndarray or list of (H,W,3) ndarray or list of (H,W) ndarray*) – List of images to augment and draw in the grid. If a list, then each element is expected to have shape `(H, W)` or `(H, W, 3)`. If a single array, then it is expected to have shape `(N, H, W, 3)` or `(H, W, 3)` or `(H, W)`.

- **rows** (*int*) – Number of rows in the grid. If `N` input images are given, this value will automatically be multiplied by `N` to create rows for each image.
- **cols** (*int*) – Number of columns in the grid.

to_deterministic(*self*, *n=None*)

Convert this augmenter from a stochastic to a deterministic one.

A stochastic augmenter samples pseudo-random values for each parameter, image and batch. A deterministic augmenter also samples new values for each parameter and image, but not batch. Instead, for consecutive batches it will sample the same values (provided the number of images and their sizes don't change). From a technical perspective this means that a deterministic augmenter starts each batch's augmentation with a random number generator in the same state (i.e. same seed), instead of advancing that state from batch to batch.

Using determinism is useful to (a) get the same augmentations for two or more image batches (e.g. for stereo cameras), (b) to augment images and corresponding data on them (e.g. segmentation maps or bounding boxes) in the same way.

Parameters **n** (*None or int, optional*) – Number of deterministic augmenters to return. If `None` then only one *imgaug.augmenters.meta.Augmenter* instance will be returned. If `1` or higher, a list containing `n` *imgaug.augmenters.meta.Augmenter* instances will be returned.

Returns A single Augmenter object if *n* was None, otherwise a list of Augmenter objects (even if *n* was `1`).

Return type *imgaug.augmenters.meta.Augmenter* or list of imgaug.augmenters.meta.Augmenter

class imgaug.augmenters.meta.**ChannelShuffle**(*p=1.0*, *channels=None*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Randomize the order of channels in input images.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; tested
* ``uint32``: yes; tested
* ``uint64``: yes; tested
* ``int8``: yes; tested
* ``int16``: yes; tested
* ``int32``: yes; tested
* ``int64``: yes; tested
* ``float16``: yes; tested
* ``float32``: yes; tested
* ``float64``: yes; tested
* ``float128``: yes; tested
* ``bool``: yes; tested
```

Parameters

- **p** (*float or imgaug.parameters.StochasticParameter, optional*) – Probability of shuffling channels in any given image. May be a fixed probability as a `float`, or a *imgaug.parameters.StochasticParameter* that returns `0` s and `1` s.
- **channels** (*None or imgaug.ALL or list of int, optional*) – Which channels are allowed to be shuffled with each other. If this is `None` or `imgaug.ALL`, then all channels may be

shuffled. If it is a `list` of `int` s, then only the channels with indices in that list may be shuffled. (Values start at `0`. All channel indices in the list must exist in each image.)

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.ChannelShuffle(0.35)
```

Shuffle all channels of `35%` of all images.

```
>>> aug = iaa.ChannelShuffle(0.35, channels=[0, 1])
```

Shuffle only channels `0` and `1` of `35%` of all images. As the new channel orders `0, 1` and `1, 0` are both valid outcomes of the shuffling, it means that for `0.35 * 0.5 = 0.175` or `17.5%` of all images the order of channels `0` and `1` is inverted.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).

Continued on next page

Table 52 – continued from previous page

deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.meta.**Lambda**(*func_images=None*, *func_heatmaps=None*, *func_segmentation_maps=None*, *func_keypoints=None*, *func_polygons='keypoints'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Augmenter that calls a lambda function for each input batch.

This is useful to add missing functions to a list of augmenters.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; tested
* ``uint32``: yes; tested
* ``uint64``: yes; tested
* ``int8``: yes; tested
* ``int16``: yes; tested
* ``int32``: yes; tested
* ``int64``: yes; tested
* ``float16``: yes; tested
* ``float32``: yes; tested
* ``float64``: yes; tested
* ``float128``: yes; tested
* ``bool``: yes; tested
```

Parameters

- **func_images** (*None or callable, optional*) – The function to call for each batch of images. It must follow the form:

```
function(images, random_state, parents, hooks)
```

  and return the changed images (may be transformed in-place). This is essentially the interface of `imgaug.augmenters.meta.Augmenter._augment_images()`. If this is `None` instead of a function, the images will not be altered.

- **func_heatmaps** (*None or callable, optional*) – The function to call for each batch of heatmaps. It must follow the form:

```
function(heatmaps, random_state, parents, hooks)
```

  and return the changed heatmaps (may be transformed in-place). This is essentially the interface of `imgaug.augmenters.meta.Augmenter._augment_heatmaps()`. If this is `None` instead of a function, the heatmaps will not be altered.

- **func_segmentation_maps** (*None or callable, optional*) – The function to call for each batch of segmentation maps. It must follow the form:

```
function(segmaps, random_state, parents, hooks)
```

  and return the changed segmaps (may be transformed in-place). This is essentially the interface of `imgaug.augmenters.meta.Augmenter._augment_segmentation_maps()`. If this is `None` instead of a function, the segmentatio maps will not be altered.

- **func_keypoints** (*None or callable, optional*) – The function to call for each batch of image keypoints. It must follow the form:

```
function(keypoints_on_images, random_state, parents, hooks)
```

  and return the changed keypoints (may be transformed in-place). This is essentially the interface of `imgaug.augmenters.meta.Augmenter._augment_keypoints()`. If this is `None` instead of a function, the keypoints will not be altered.

- **func_polygons** (*"keypoints" or None or callable, optional*) – The function to call for each batch of image polygons. It must follow the form:

```
function(polygons_on_images, random_state, parents, hooks)
```

  and return the changed polygons (may be transformed in-place). This is essentially the interface of `imgaug.augmenters.meta.Augmenter._augment_polygons()`. If this is `None` instead of a function, the polygons will not be altered. If this is the string `"keypoints"` instead of a function, the polygons will automatically be augmented by transforming their corner vertices to keypoints and calling *func_keypoints*.

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>>
>>> def func_images(images, random_state, parents, hooks):
>>>     images[:, ::2, :, :] = 0
>>>     return images
>>>
>>> aug = iaa.Lambda(
>>>     func_images=func_images
>>> )
```

Replace every second row in input images with black pixels. Leave other data (e.g. heatmaps, keypoints) unchanged.

```
>>> def func_images(images, random_state, parents, hooks):
>>>     images[:, ::2, :, :] = 0
>>>     return images
>>>
>>> def func_heatmaps(heatmaps, random_state, parents, hooks):
>>>     for heatmaps_i in heatmaps:
>>>         heatmaps.arr_0to1[::2, :, :] = 0
>>>     return heatmaps
>>>
>>> def func_keypoints(keypoints_on_images, random_state, parents, hooks):
>>>     return keypoints_on_images
>>>
>>> aug = iaa.Lambda(
>>>     func_images=func_images,
>>>     func_heatmaps=func_heatmaps,
>>>     func_keypoints=func_keypoints
>>> )
```

Replace every second row in images with black pixels, set every second row in heatmaps to zero and leave other data (e.g. keypoints) unchanged.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.

Continued on next page

Table 53 – continued from previous page

augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters	

get_parameters(*self*)

class imgaug.augmenters.meta.**Noop**(*name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Augmenter that never changes input images ("no operation").

This augmenter is useful when you just want to use a placeholder augmenter in some situation, so that you can continue to call augmentation methods without actually transforming the input data. This allows to use the same code for training and test.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; tested
* ``uint32``: yes; tested
* ``uint64``: yes; tested
```

(continues on next page)

(continued from previous page)

```
* ``int8``: yes; tested
* ``int16``: yes; tested
* ``int32``: yes; tested
* ``int64``: yes; tested
* ``float16``: yes; tested
* ``float32``: yes; tested
* ``float64``: yes; tested
* ``float128``: yes; tested
* ``bool``: yes; tested
```

Parameters

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.

Continued on next page

Table 54 – continued from previous page

find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.meta.**OneOf**(*children*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.SomeOf*

Augmenter that always executes exactly one of its children.

dtype support:

```
See ``imgaug.augmenters.meta.SomeOf``.
```

Parameters

- **children** (*imgaug.augmenters.meta.Augmenter or list of imgaug.augmenters.meta.Augmenter*) – The choices of augmenters to apply.
- **name** (*None or str, optional*) – See imgaug.augmenters.meta.Augmenter.__init__().
- **deterministic** (*bool, optional*) – See imgaug.augmenters.meta.Augmenter.__init__().
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See imgaug.augmenters.meta.Augmenter.__init__().

Examples

```
>>> import imgaug.augmenters as iaa
>>> images = [np.ones((10, 10), dtype=np.uint8)]  # dummy example images
>>> seq = iaa.OneOf([
>>>     iaa.Fliplr(1.0),
>>>     iaa.Flipud(1.0)
>>> ])
>>> images_aug = seq.augment_images(images)
```

Flip each image either horizontally or vertically.

```
>>> images = [np.ones((10, 10), dtype=np.uint8)]  # dummy example images
>>> seq = iaa.OneOf([
>>>     iaa.Fliplr(1.0),
>>>     iaa.Sequential([
>>>         iaa.GaussianBlur(1.0),
>>>         iaa.Dropout(0.05),
>>>         iaa.AdditiveGaussianNoise(0.1*255)
>>>     ]),
>>>     iaa.Noop()
>>> ])
>>> images_aug = seq.augment_images(images)
```

Either flip each image horizontally, or add blur+dropout+noise or do nothing.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`add`(self, augmenter)	Add an augmenter to the list of child augmenters.
`append`(self, object, /)	Append object to the end of the list.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`clear`(self, /)	Remove all items from list.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.

Continued on next page

Table 55 – continued from previous page

copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
count(self, value, /)	Return number of occurrences of value.
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
extend(self, iterable, /)	Extend list by appending elements from the iterable.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
index(self, value[, start, stop])	Return first index of value.
insert(self, index, object, /)	Insert object before index.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
pop(self[, index])	Remove and return item at index (default last).
remove(self, value, /)	Remove first occurrence of value.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
reverse(self, /)	Reverse *IN PLACE*.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
sort(self, /, *[, key, reverse])	Stable sort *IN PLACE*.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.meta.**Sequential**(*children=None*, *random_order=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*, list

List augmenter containing child augmenters to apply to inputs.

This augmenter is simply a list of other augmenters. To augment an image or any other data, it iterates over its children and applies each one of them independently to the data. (This also means that the second applied augmenter will already receive augmented input data and augment it further.)

This augmenter offers the option to apply its children in random order using the *random_order* parameter. This should often be activated as it greatly increases the space of possible augmentations.

Note: You are *not* forced to use *imgaug.augmenters.meta.Sequential* in order to use other aug-

menters. Each augmenter can be used on its own, e.g the following defines an augmenter for horizontal flips and then augments a single image:

```
>>> import numpy as np
>>> import imgaug.augmenters as iaa
>>> image = np.zeros((32, 32, 3), dtype=np.uint8)
>>> aug = iaa.Fliplr(0.5)
>>> image_aug = aug.augment_image(image)
```

---

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; tested
* ``uint32``: yes; tested
* ``uint64``: yes; tested
* ``int8``: yes; tested
* ``int16``: yes; tested
* ``int32``: yes; tested
* ``int64``: yes; tested
* ``float16``: yes; tested
* ``float32``: yes; tested
* ``float64``: yes; tested
* ``float128``: yes; tested
* ``bool``: yes; tested
```

Parameters

- **children** (*imgaug.augmenters.meta.Augmenter or list of imgaug.augmenters.meta.Augmenter or None, optional*) – The augmenters to apply to images.
- **random_order** (*bool, optional*) – Whether to apply the child augmenters in random order. If `True`, the order will be randomly sampled once per batch.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import numpy as np
>>> import imgaug.augmenters as iaa
>>> imgs = [np.random.rand(10, 10)]
>>> seq = iaa.Sequential([
>>>     iaa.Fliplr(0.5),
>>>     iaa.Flipud(0.5)
>>> ])
>>> imgs_aug = seq.augment_images(imgs)
```

Create a *imgaug.augmenters.meta.Sequential* that always first applies a horizontal flip augmenter and then a vertical flip augmenter. Each of these two augmenters has a 50% probability of actually flipping the image.

```
>>> seq = iaa.Sequential([
>>>     iaa.Fliplr(0.5),
>>>     iaa.Flipud(0.5)
>>> ], random_order=True)
>>> imgs_aug = seq.augment_images(imgs)
```

Create a *imgaug.augmenters.meta.Sequential* that sometimes first applies a horizontal flip augmenter (followed by a vertical flip augmenter) and sometimes first a vertical flip augmenter (followed by a horizontal flip augmenter). Again, each of them has a 50% probability of actually flipping the image.

Methods

__call__(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
add(self, augmenter)	Add an augmenter to the list of child augmenters.
append(self, object, /)	Append object to the end of the list.
augment(self[, return_batch, hooks])	Augment a batch.
augment_batch(self, batch[, hooks])	Augment a single batch.
augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ...[, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
clear(self, /)	Remove all items from list.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
count(self, value, /)	Return number of occurrences of value.
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
extend(self, iterable, /)	Extend list by appending elements from the iterable.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.

Continued on next page

Table 56 – continued from previous page

find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
index(self, value[, start, stop])	Return first index of value.
insert(self, index, object, /)	Insert object before index.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
pop(self[, index])	Remove and return item at index (default last).
remove(self, value, /)	Remove first occurrence of value.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
reverse(self, /)	Reverse *IN PLACE*.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
sort(self, /, *[, key, reverse])	Stable sort *IN PLACE*.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

add(*self*, *augmenter*)

Add an augmenter to the list of child augmenters.

Parameters **imgaug.augmenters.meta.Augmenter** – The augmenter to add.

get_children_lists(*self*)

Get a list of lists of children of this augmenter.

For most augmenters, the result will be a single empty list. For augmenters with children it will often be a list with one sublist containing all children. In some cases the augmenter will contain multiple distinct lists of children, e.g. an if-list and an else-list. This will lead to a result consisting of a single list with multiple sublists, each representing the respective sublist of children.

E.g. for an if/else-augmenter that executes the children `A1`, `A2` if a condition is met and otherwise executes the children `B1`, `B2`, `B3` the result will be `[[A1, A2], [B1, B2, B3]]`.

IMPORTANT: While the topmost list may be newly created, each of the sublist must be editable inplace resulting in a changed children list of the augmenter. E.g. if an Augmenter `IfElse(condition, [A1, A2], [B1, B2, B3])` returns `[[A1, A2], [B1, B2, B3]]` for a call to *imgaug.augmenters.meta.Augmenter.get_children_lists()* and `A2` is removed inplace from `[A1, A2]`, then the children lists of `IfElse(...)` must also change to `[A1]`, `[B1, B2, B3]`. This is used in `imgaug.augmeneters.meta.Augmenter.remove_augmenters_inplace()`.

Returns One or more lists of child augmenter. Can also be a single empty list.

Return type list of list of imgaug.augmenters.meta.Augmenter

get_parameters(*self*)

class imgaug.augmenters.meta.**SomeOf**(*n=None*, *children=None*, *random_order=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*, list

List augmenter that applies only some of its children to inputs.

This augmenter is similar to *imgaug.augmenters.meta.Sequential*, but may apply only a fixed or random subset of its child augmenters to inputs. E.g. the augmenter could be initialized with a list of 20 child augmenters and then apply 5 randomly chosen child augmenters to images.

The subset of augmenters to apply (and their order) is sampled once *per image*. If *random_order* is `True`, the order will be sampled once *per batch* (similar to *imgaug.augmenters.meta.Sequential*).

This augmenter currently does not support replacing (i.e. picking the same child multiple times) due to implementation difficulties in connection with deterministic augmenters.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; tested
* ``uint32``: yes; tested
* ``uint64``: yes; tested
* ``int8``: yes; tested
* ``int16``: yes; tested
* ``int32``: yes; tested
* ``int64``: yes; tested
* ``float16``: yes; tested
* ``float32``: yes; tested
* ``float64``: yes; tested
* ``float128``: yes; tested
* ``bool``: yes; tested
```

Parameters

- **n** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter or None, optional*) –

  Count of augmenters to apply.

  - If `int`, then exactly *n* of the child augmenters are applied to every image.
  - If tuple of two `int`s `(a, b)`, then a random value will be uniformly sampled per image from the discrete interval `[a..b]` and denote the number of child augmenters to pick and apply. `b` may be set to `None`, which is then equivalent to `(a..C)` with `C` denoting the number of children that the augmenter has.
  - If `StochasticParameter`, then `N` numbers will be sampled for `N` images. The parameter is expected to be discrete.
  - If `None`, then the total number of available children will be used (i.e. all children will be applied).

- **children** (*imgaug.augmenters.meta.Augmenter or list of imgaug.augmenters.meta.Augmenter or None, optional*) – The augmenters to apply to images. If this is a list of augmenters, it will be converted to a *imgaug.augmenters.meta.Sequential*.

- **random_order** (*boolean, optional*) – Whether to apply the child augmenters in random order. If `True`, the order will be randomly sampled once per batch.

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> imgs = [np.random.rand(10, 10)]
>>> seq = iaa.SomeOf(1, [
>>>     iaa.Fliplr(1.0),
>>>     iaa.Flipud(1.0)
>>> ])
>>> imgs_aug = seq.augment_images(imgs)
```

Apply either `Fliplr` or `Flipud` to images.

```
>>> seq = iaa.SomeOf((1, 3), [
>>>     iaa.Fliplr(1.0),
>>>     iaa.Flipud(1.0),
>>>     iaa.GaussianBlur(1.0)
>>> ])
>>> imgs_aug = seq.augment_images(imgs)
```

Apply one to three of the listed augmenters (`Fliplr`, `Flipud`, `GaussianBlur`) to images. They are always applied in the provided order, i.e. first `Fliplr`, second `Flipud`, third `GaussianBlur`.

```
>>> seq = iaa.SomeOf((1, None), [
>>>     iaa.Fliplr(1.0),
>>>     iaa.Flipud(1.0),
>>>     iaa.GaussianBlur(1.0)
>>> ], random_order=True)
>>> imgs_aug = seq.augment_images(imgs)
```

Apply one to all of the listed augmenters (`Fliplr`, `Flipud`, `GaussianBlur`) to images. They are applied in random order, i.e. sometimes `GaussianBlur` first, followed by `Fliplr`, sometimes `Fliplr` followed by `Flipud` followed by `Blur` etc. The order is sampled once per batch.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
add(self, augmenter)	Add an augmenter to the list of child augmenters.
`append`(self, object, /)	Append object to the end of the list.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.

Continued on next page

Table 57 – continued from previous page

augment_bounding_boxes(self, ... [, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ... [, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
clear(self, /)	Remove all items from list.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
count(self, value, /)	Return number of occurrences of value.
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
extend(self, iterable, /)	Extend list by appending elements from the iterable.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
index(self, value[, start, stop])	Return first index of value.
insert(self, index, object, /)	Insert object before index.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
pop(self[, index])	Remove and return item at index (default last).
remove(self, value, /)	Remove first occurrence of value.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
reverse(self, /)	Reverse *IN PLACE*.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
sort(self, /, *[, key, reverse])	Stable sort *IN PLACE*.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters	

add (*self*, *augmenter*)
Add an augmenter to the list of child augmenters.

Parameters **augmenter** (*imgaug.augmenters.meta.Augmenter*) – The augmenter to add.

get_children_lists (*self*)
Get a list of lists of children of this augmenter.

For most augmenters, the result will be a single empty list. For augmenters with children it will often be a list with one sublist containing all children. In some cases the augmenter will contain multiple distinct lists of children, e.g. an if-list and an else-list. This will lead to a result consisting of a single list with multiple sublists, each representing the respective sublist of children.

E.g. for an if/else-augmenter that executes the children `A1`, `A2` if a condition is met and otherwise executes the children `B1`, `B2`, `B3` the result will be `[[A1, A2], [B1, B2, B3]]`.

IMPORTANT: While the topmost list may be newly created, each of the sublist must be editable inplace resulting in a changed children list of the augmenter. E.g. if an Augmenter `IfElse(condition, [A1, A2], [B1, B2, B3])` returns `[[A1, A2], [B1, B2, B3]]` for a call to *imgaug.augmenters.meta.Augmenter.get_children_lists()* and `A2` is removed inplace from `[A1, A2]`, then the children lists of `IfElse(...)` must also change to `[A1], [B1, B2, B3]`. This is used in `imgaug.augmeneters.meta.Augmenter.remove_augmenters_inplace()`.

Returns One or more lists of child augmenter. Can also be a single empty list.

Return type list of list of imgaug.augmenters.meta.Augmenter

get_parameters (*self*)

class imgaug.augmenters.meta.**Sometimes** (*p=0.5*, *then_list=None*, *else_list=None*, *name=None*, *deterministic=False*, *random_state=None*)
Bases: *imgaug.augmenters.meta.Augmenter*

Apply child augmenter(s) with a probability of *p*.

Let `C` be one or more child augmenters given to *imgaug.augmenters.meta.Sometimes*. Let `p` be the fraction of images (or other data) to augment. Let `I` be the input images (or other data). Let `N` be the number of input images (or other entities). Then (on average) `p*N` images of `I` will be augmented using `C`.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; tested
* ``uint32``: yes; tested
* ``uint64``: yes; tested
* ``int8``: yes; tested
* ``int16``: yes; tested
* ``int32``: yes; tested
* ``int64``: yes; tested
* ``float16``: yes; tested
* ``float32``: yes; tested
* ``float64``: yes; tested
* ``float128``: yes; tested
* ``bool``: yes; tested
```

Parameters

- **p** (*float or imgaug.parameters.StochasticParameter, optional*) – Sets the probability with which the given augmenters will be applied to input images/data. E.g. a value of `0.5` will result in `50%` of all input images (or other augmentables) being augmented.
- **then_list** (*None or imgaug.augmenters.meta.Augmenter or list of imgaug.augmenters.meta.Augmenter, optional*) – Augmenter(s) to apply to *p%* percent of all images. If this is a list of augmenters, it will be converted to a *imgaug.augmenters.meta.Sequential*.
- **else_list** (*None or imgaug.augmenters.meta.Augmenter or list of imgaug.augmenters.meta.Augmenter, optional*) – Augmenter(s) to apply to `(1-p)` percent of all images. These augmenters will be applied only when the ones in *then_list* are *not* applied (either-or-relationship). If this is a list of augmenters, it will be converted to a *imgaug.augmenters.meta.Sequential*.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Sometimes(0.5, iaa.GaussianBlur(0.3))
```

Apply `GaussianBlur` to `50%` of all input images.

```
>>> aug = iaa.Sometimes(0.5, iaa.GaussianBlur(0.3), iaa.Fliplr(1.0))
```

Apply `GaussianBlur` to `50%` of all input images. Apply `Fliplr` to the other `50%` of all input images.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.

Continued on next page

Table 58 – continued from previous page

augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters	

get_children_lists(*self*)

Get a list of lists of children of this augmenter.

For most augmenters, the result will be a single empty list. For augmenters with children it will often be a list with one sublist containing all children. In some cases the augmenter will contain multiple distinct lists of children, e.g. an if-list and an else-list. This will lead to a result consisting of a single list with multiple sublists, each representing the respective sublist of children.

E.g. for an if/else-augmenter that executes the children `A1`, `A2` if a condition is met and otherwise executes the children `B1`, `B2`, `B3` the result will be `[[A1, A2], [B1, B2, B3]]`.

IMPORTANT: While the topmost list may be newly created, each of the sublist must be editable inplace resulting in a changed children list of the augmenter. E.g. if an Augmenter `IfElse(condition, [A1, A2], [B1, B2, B3])` returns `[[A1, A2], [B1, B2, B3]]` for a call to *imgaug.augmenters.meta.Augmenter.get_children_lists()* and `A2` is removed inplace from `[A1, A2]`, then the children lists of `IfElse(...)` must also change

to [A1], [B1, B2, B3]. This is used in imgaug.augmeneters.meta.Augmenter.remove_augmenters_inplace().

Returns One or more lists of child augmenter. Can also be a single empty list.

Return type list of list of imgaug.augmenters.meta.Augmenter

get_parameters(*self*)

class imgaug.augmenters.meta.**WithChannels**(*channels=None*, *children=None*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Apply child augmenters to specific channels.

Let C be one or more child augmenters given to this augmenter. Let H be a list of channels. Let I be the input images. Then this augmenter will pick the channels H from each image in I (resulting in new images) and apply C to them. The result of the augmentation will be merged back into the original images.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; tested
* ``uint32``: yes; tested
* ``uint64``: yes; tested
* ``int8``: yes; tested
* ``int16``: yes; tested
* ``int32``: yes; tested
* ``int64``: yes; tested
* ``float16``: yes; tested
* ``float32``: yes; tested
* ``float64``: yes; tested
* ``float128``: yes; tested
* ``bool``: yes; tested
```

Parameters

- **channels** (*None or int or list of int, optional*) – Sets the channels to be extracted from each image. If None, all channels will be used. Note that this is not stochastic - the extracted channels are always the same ones.
- **children** (*Augmenter or list of imgaug.augmenters.meta.Augmenter or None, optional*) – One or more augmenters to apply to images, after the channels are extracted.
- **name** (*None or str, optional*) – See imgaug.augmenters.meta.Augmenter.__init__().
- **deterministic** (*bool, optional*) – See imgaug.augmenters.meta.Augmenter.__init__().
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See imgaug.augmenters.meta.Augmenter.__init__().

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.WithChannels([0], iaa.Add(10))
```

Assuming input images are RGB, then this augmenter will add `10` only to the first channel, i.e. it will make images appear more red.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
`remove_augmenters_inplace`(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
`reseed`(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
`show_grid`(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.

Continued on next page

Table 59 – continued from previous page

to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_children_lists(*self*)
Get a list of lists of children of this augmenter.

For most augmenters, the result will be a single empty list. For augmenters with children it will often be a list with one sublist containing all children. In some cases the augmenter will contain multiple distinct lists of children, e.g. an if-list and an else-list. This will lead to a result consisting of a single list with multiple sublists, each representing the respective sublist of children.

E.g. for an if/else-augmenter that executes the children `A1`, `A2` if a condition is met and otherwise executes the children `B1`, `B2`, `B3` the result will be `[[A1, A2], [B1, B2, B3]]`.

IMPORTANT: While the topmost list may be newly created, each of the sublist must be editable inplace resulting in a changed children list of the augmenter. E.g. if an Augmenter `IfElse(condition, [A1, A2], [B1, B2, B3])` returns `[[A1, A2], [B1, B2, B3]]` for a call to *imgaug.augmenters.meta.Augmenter.get_children_lists()* and `A2` is removed inplace from `[A1, A2]`, then the children lists of `IfElse(...)` must also change to `[A1]`, `[B1, B2, B3]`. This is used in `imgaug.augmeneters.meta.Augmenter.remove_augmenters_inplace()`.

Returns One or more lists of child augmenter. Can also be a single empty list.

Return type list of list of imgaug.augmenters.meta.Augmenter

get_parameters(*self*)

imgaug.augmenters.meta.**clip_augmented_image**(*image*, *min_value*, *max_value*)
Deprecated. Use `imgaug.dtypes.clip_` instead.

imgaug.augmenters.meta.**clip_augmented_image_**(*image*, *min_value*, *max_value*)
Deprecated. Use `imgaug.dtypes.clip_` instead.

imgaug.augmenters.meta.**clip_augmented_images**(*images*, *min_value*, *max_value*)
Deprecated. Use `imgaug.dtypes.clip_` instead.

imgaug.augmenters.meta.**clip_augmented_images_**(*images*, *min_value*, *max_value*)
Deprecated. Use `imgaug.dtypes.clip_` instead.

imgaug.augmenters.meta.**copy_arrays**(*arrays*)

imgaug.augmenters.meta.**estimate_max_number_of_channels**(*images*)

imgaug.augmenters.meta.**handle_children_list**(*lst*, *augmenter_name*, *lst_name*, *default='sequential'*)

imgaug.augmenters.meta.**invert_reduce_to_nonempty**(*objs*, *ids*, *objs_reduced*)

imgaug.augmenters.meta.**reduce_to_nonempty**(*objs*)

imgaug.augmenters.meta.**shuffle_channels**(*image*, *random_state*, *channels=None*)
Randomize the order of (color) channels in an image.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; indirectly tested (1)
* ``uint32``: yes; indirectly tested (1)
* ``uint64``: yes; indirectly tested (1)
* ``int8``: yes; indirectly tested (1)
* ``int16``: yes; indirectly tested (1)
* ``int32``: yes; indirectly tested (1)
* ``int64``: yes; indirectly tested (1)
* ``float16``: yes; indirectly tested (1)
* ``float32``: yes; indirectly tested (1)
* ``float64``: yes; indirectly tested (1)
* ``float128``: yes; indirectly tested (1)
* ``bool``: yes; indirectly tested (1)

- (1) Indirectly tested via ``ChannelShuffle``.
```

Parameters

- **image** (*((H,W,[C]) ndarray*) – Image of any dtype for which to shuffle the channels.
- **random_state** (*imgaug.random.RNG*) – The random state to use for this shuffling operation.
- **channels** (*None or imgaug.ALL or list of int, optional*) – Which channels are allowed to be shuffled with each other. If this is `None` or `imgaug.ALL`, then all channels may be shuffled. If it is a `list` of `int` s, then only the channels with indices in that list may be shuffled. (Values start at `0`. All channel indices in the list must exist in the image.)

Returns The input image with shuffled channels.

Return type ndarray

13.17 imgaug.augmenters.arithmetic

Augmenters that perform simple arithmetic changes.

Do not import directly from this file, as the categorization is not final. Use instead:

```
from imgaug import augmenters as iaa
```

and then e.g.:

```
seq = iaa.Sequential([iaa.Add((-5, 5)), iaa.Multiply((0.9, 1.1))])
```

List of augmenters:

- Add
- AddElementwise
- AdditiveGaussianNoise
- AdditiveLaplaceNoise
- AdditivePoissonNoise
- Multiply
- MultiplyElementwise
- Dropout

- CoarseDropout
- ReplaceElementwise
- ImpulseNoise
- SaltAndPepper
- CoarseSaltAndPepper
- Salt
- CoarseSalt
- Pepper
- CoarsePepper
- Invert
- ContrastNormalization
- JpegCompression

class imgaug.augmenters.arithmetic.**Add**(*value=0*, *per_channel=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Add a value to all pixels in an image.

dtype support:

```
See :func:`imgaug.augmenters.arithmetic.add_scalar`.
```

Parameters

- **value** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) –

  Value to add to all pixels.

  – If a number, exactly that value will always be used.

  – If a tuple `(a, b)`, then a value from the discrete interval `[a..b]` will be sampled per image.

  – If a list, then a random value will be sampled from that list per image.

  – If a `StochasticParameter`, then a value will be sampled per image from that parameter.

- **per_channel** (*bool or float or imgaug.parameters.StochasticParameter, optional*) – Whether to use (imagewise) the same sample(s) for all channels (`False`) or to sample value(s) for each channel (`True`). Setting this to `True` will therefore lead to different transformations per image *and* channel, otherwise only per image. If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`. If it is a `StochasticParameter` it is expected to produce samples with values between `0.0` and `1.0`, where values `>0.5` will lead to per-channel behaviour (i.e. same as `True`).

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Add(10)
```

Always adds a value of 10 to all channels of all pixels of all input images.

```
>>> aug = iaa.Add((-10, 10))
```

Adds a value from the discrete interval `[-10..10]` to all pixels of input images. The exact value is sampled per image.

```
>>> aug = iaa.Add((-10, 10), per_channel=True)
```

Adds a value from the discrete interval `[-10..10]` to all pixels of input images. The exact value is sampled per image *and* channel, i.e. to a red-channel it might add 5 while subtracting 7 from the blue channel of the same image.

```
>>> aug = iaa.Add((-10, 10), per_channel=0.5)
```

Identical to the previous example, but the *per_channel* feature is only active for 50 percent of all images.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.

Continued on next page

Table 60 – continued from previous page

copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.arithmetic.**AddElementwise**(*value=0*, *per_channel=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Add to the pixels of images values that are pixelwise randomly sampled.

While the Add Augmenter samples one value to add *per image* (and optionally per channel), this augmenter samples different values per image and *per pixel* (and optionally per channel), i.e. intensities of neighbouring pixels may be increased/decreased by different amounts.

dtype support:

```
See :func:`imgaug.augmenters.arithmetic.add_elementwise`.
```

Parameters

- **value** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) –

  Value to add to the pixels.

  – If an int, exactly that value will always be used.

  - If a tuple `(a, b)`, then values from the discrete interval `[a..b]` will be sampled per image and pixel.
  - If a list of integers, a random value will be sampled from the list per image and pixel.
  - If a `StochasticParameter`, then values will be sampled per image and pixel from that parameter.

- **per_channel** (*bool or float or imgaug.parameters.StochasticParameter, optional*) – Whether to use (imagewise) the same sample(s) for all channels (`False`) or to sample value(s) for each channel (`True`). Setting this to `True` will therefore lead to different transformations per image *and* channel, otherwise only per image. If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`. If it is a `StochasticParameter` it is expected to produce samples with values between `0.0` and `1.0`, where values `>0.5` will lead to per-channel behaviour (i.e. same as `True`).
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.AddElementwise(10)
```

Always adds a value of 10 to all channels of all pixels of all input images.

```
>>> aug = iaa.AddElementwise((-10, 10))
```

Samples per image and pixel a value from the discrete interval `[-10..10]` and adds that value to the respective pixel.

```
>>> aug = iaa.AddElementwise((-10, 10), per_channel=True)
```

Samples per image, pixel *and also channel* a value from the discrete interval `[-10..10]` and adds it to the respective pixel's channel value. Therefore, added values may differ between channels of the same pixel.

```
>>> aug = iaa.AddElementwise((-10, 10), per_channel=0.5)
```

Identical to the previous example, but the *per_channel* feature is only active for 50 percent of all images.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.

Continued on next page

Table 61 – continued from previous page

augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ...[, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.arithmetic.**AdditiveGaussianNoise**(*loc=0*, *scale=0*, *per_channel=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.arithmetic.AddElementwise*

Add noise sampled from gaussian distributions elementwise to images.

This augmenter samples and adds noise elementwise, i.e. it can add different noise values to neighbouring pixels and is comparable to `AddElementwise`.

dtype support:

```
See ``imgaug.augmenters.arithmetic.AddElementwise``.
```

Parameters

- **loc** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) –

  Mean of the normal distribution from which the noise is sampled.

  – If a number, exactly that value will always be used.

  – If a tuple `(a, b)`, a random value from the interval `[a, b]` will be sampled per image.

  – If a list, then a random value will be sampled from that list per image.

  – If a `StochasticParameter`, a value will be sampled from the parameter per image.

- **scale** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Standard deviation of the normal distribution that generates the noise. Must be `>=0`. If `0` then *loc* will simply be added to all pixels.

  – If a number, exactly that value will always be used.

  – If a tuple `(a, b)`, a random value from the interval `[a, b]` will be sampled per image.

  – If a list, then a random value will be sampled from that list per image.

  – If a `StochasticParameter`, a value will be sampled from the parameter per image.

- **per_channel** (*bool or float or imgaug.parameters.StochasticParameter, optional*) – Whether to use (imagewise) the same sample(s) for all channels (`False`) or to sample value(s) for each channel (`True`). Setting this to `True` will therefore lead to different transformations per image *and* channel, otherwise only per image. If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`. If it is a `StochasticParameter` it is expected to produce samples with values between `0.0` and `1.0`, where values `>0.5` will lead to per-channel behaviour (i.e. same as `True`).

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.AdditiveGaussianNoise(scale=0.1*255)
```

Adds gaussian noise from the distribution `N(0, 0.1*255)` to images. The samples are drawn per image and pixel.

```
>>> aug = iaa.AdditiveGaussianNoise(scale=(0, 0.1*255))
```

Adds gaussian noise from the distribution `N(0, s)` to images, where `s` is sampled per image from the interval `[0, 0.1*255]`.

```
>>> aug = iaa.AdditiveGaussianNoise(scale=0.1*255, per_channel=True)
```

Adds gaussian noise from the distribution `N(0, 0.1*255)` to images, where the noise value is different per image and pixel *and* channel (e.g. a different one for red, green and blue channels of the same pixel). This leads to "colorful" noise.

```
>>> aug = iaa.AdditiveGaussianNoise(scale=0.1*255, per_channel=0.5)
```

Identical to the previous example, but the *per_channel* feature is only active for 50 percent of all images.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.

Continued on next page

Table 62 – continued from previous page

find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.arithmetic.**AdditiveLaplaceNoise**(*loc=0*, *scale=0*, *per_channel=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.arithmetic.AddElementwise*

Add noise sampled from laplace distributions elementwise to images.

The laplace distribution is similar to the gaussian distribution, but puts more weight on the long tail. Hence, this noise will add more outliers (very high/low values). It is somewhere between gaussian noise and salt and pepper noise.

Values of around `255 * 0.05` for *scale* lead to visible noise (for `uint8`). Values of around `255 * 0.10` for *scale* lead to very visible noise (for `uint8`). It is recommended to usually set *per_channel* to `True`.

This augmenter samples and adds noise elementwise, i.e. it can add different noise values to neighbouring pixels and is comparable to `AddElementwise`.

dtype support:

```
See ``imgaug.augmenters.arithmetic.AddElementwise``.
```

Parameters

- **loc** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) –

  Mean of the laplace distribution that generates the noise.

  – If a number, exactly that value will always be used.

  – If a tuple `(a, b)`, a random value from the interval `[a, b]` will be sampled per image.

  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, a value will be sampled from the parameter per image.

- **scale** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Standard deviation of the laplace distribution that generates the noise. Must be `>=0`. If `0` then only *loc* will be used. Recommended to be around `255*0.05`.
  - If a number, exactly that value will always be used.
  - If a tuple `(a, b)`, a random value from the interval `[a, b]` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, a value will be sampled from the parameter per image.
- **per_channel** (*bool or float or imgaug.parameters.StochasticParameter, optional*) – Whether to use (imagewise) the same sample(s) for all channels (`False`) or to sample value(s) for each channel (`True`). Setting this to `True` will therefore lead to different transformations per image *and* channel, otherwise only per image. If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`. If it is a `StochasticParameter` it is expected to produce samples with values between `0.0` and `1.0`, where values `>0.5` will lead to per-channel behaviour (i.e. same as `True`).
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.AdditiveLaplaceNoise(scale=0.1*255)
```

Adds laplace noise from the distribution `Laplace(0, 0.1*255)` to images. The samples are drawn per image and pixel.

```
>>> aug = iaa.AdditiveLaplaceNoise(scale=(0, 0.1*255))
```

Adds laplace noise from the distribution `Laplace(0, s)` to images, where `s` is sampled per image from the interval `[0, 0.1*255]`.

```
>>> aug = iaa.AdditiveLaplaceNoise(scale=0.1*255, per_channel=True)
```

Adds laplace noise from the distribution `Laplace(0, 0.1*255)` to images, where the noise value is different per image and pixel *and* channel (e.g. a different one for the red, green and blue channels of the same pixel). This leads to "colorful" noise.

```
>>> aug = iaa.AdditiveLaplaceNoise(scale=0.1*255, per_channel=0.5)
```

Identical to the previous example, but the *per_channel* feature is only active for 50 percent of all images.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
`remove_augmenters_inplace`(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
`reseed`(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
`show_grid`(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
`to_deterministic`(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.arithmetic.**AdditivePoissonNoise**(*lam=0*, *per_channel=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.arithmetic.AddElementwise*

Add noise sampled from poisson distributions elementwise to images.

Poisson noise is comparable to gaussian noise, as e.g. generated via `AdditiveGaussianNoise`. As poisson distributions produce only positive numbers, the sign of the sampled values are here randomly flipped.

Values of around `10.0` for *lam* lead to visible noise (for `uint8`). Values of around `20.0` for *lam* lead to very visible noise (for `uint8`). It is recommended to usually set *per_channel* to `True`.

This augmenter samples and adds noise elementwise, i.e. it can add different noise values to neighbouring pixels and is comparable to `AddElementwise`.

dtype support:

```
See ``imgaug.augmenters.arithmetic.AddElementwise``.
```

Parameters

- **lam** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Lambda parameter of the poisson distribution. Must be `>=0`. Recommended values are around `0.0` to `10.0`.
  - If a number, exactly that value will always be used.
  - If a tuple `(a, b)`, a random value from the interval `[a, b]` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, a value will be sampled from the parameter per image.
- **per_channel** (*bool or float or imgaug.parameters.StochasticParameter, optional*) – Whether to use (imagewise) the same sample(s) for all channels (`False`) or to sample value(s) for each channel (`True`). Setting this to `True` will therefore lead to different transformations per image *and* channel, otherwise only per image. If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`. If it is a `StochasticParameter` it is expected to produce samples with values between `0.0` and `1.0`, where values `>0.5` will lead to per-channel behaviour (i.e. same as `True`).
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.AdditivePoissonNoise(lam=5.0)
```

Adds poisson noise sampled from a poisson distribution with a `lambda` parameter of `5.0` to images. The samples are drawn per image and pixel.

```
>>> aug = iaa.AdditivePoissonNoise(lam=(0.0, 10.0))
```

Adds poisson noise sampled from `Poisson(x)` to images, where `x` is randomly sampled per image from the interval `[0.0, 10.0]`.

```
>>> aug = iaa.AdditivePoissonNoise(lam=5.0, per_channel=True)
```

Adds poisson noise sampled from `Poisson(5.0)` to images, where the values are different per image and pixel *and* channel (e.g. a different one for red, green and blue channels for the same pixel).

```
>>> aug = iaa.AdditivePoissonNoise(lam=(0.0, 10.0), per_channel=True)
```

Adds poisson noise sampled from `Poisson(x)` to images, with `x` being sampled from `uniform(0.0, 10.0)` per image and channel. This is the *recommended* configuration.

```
>>> aug = iaa.AdditivePoissonNoise(lam=(0.0, 10.0), per_channel=0.5)
```

Identical to the previous example, but the *per_channel* feature is only active for 50 percent of all images.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.

Continued on next page

Table 64 – continued from previous page

find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.arithmetic.**CoarseDropout**(*p=0*, *size_px=None*, *size_percent=None*, *per_channel=False*, *min_size=4*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.arithmetic.MultiplyElementwise*

Set rectangular areas within images to zero.

In contrast to Dropout, these areas can have larger sizes. (E.g. you might end up with three large black rectangles in an image.) Note that the current implementation leads to correlated sizes, so if e.g. there is any thin and high rectangle that is dropped, there is a high likelihood that all other dropped areas are also thin and high.

This method is implemented by generating the dropout mask at a lower resolution (than the image has) and then upsampling the mask before dropping the pixels.

This augmenter is similar to Cutout. Usually, cutout is defined as an operation that drops exactly one rectangle from an image, while here CoarseDropout can drop multiple rectangles (with some correlation between the sizes of these rectangles).

dtype support:

```
See ``imgaug.augmenters.arithmetic.MultiplyElementwise``.
```

Parameters

- **p** (*float or tuple of float or imgaug.parameters.StochasticParameter, optional*) – The probability of any pixel being dropped (i.e. set to zero) in the lower-resolution dropout mask.
  - If a float, then that value will be used for all pixels. A value of 1.0 would mean, that all pixels will be dropped. A value of 0.0 would lead to no pixels being dropped.

  - If a tuple `(a, b)`, then a value `p` will be sampled from the interval `[a, b]` per image and be used as the dropout probability.
  - If a `StochasticParameter`, then this parameter will be used to determine per pixel whether it should be *kept* (sampled value of `>0.5`) or shouldn't be kept (sampled value of `<=0.5`). If you instead want to provide the probability as a stochastic parameter, you can usually do `imgaug.parameters.Binomial(1-p)` to convert parameter *p* to a 0/1 representation.

- **size_px** (*None or int or tuple of int or imgaug.parameters.StochasticParameter, optional*) – The size of the lower resolution image from which to sample the dropout mask in absolute pixel dimensions. Note that this means that *lower* values of this parameter lead to *larger* areas being dropped (as any pixel in the lower resolution image will correspond to a larger area at the original resolution).
  - If `None` then *size_percent* must be set.
  - If an integer, then that size will always be used for both height and width. E.g. a value of `3` would lead to a `3x3` mask, which is then upsampled to `HxW`, where `H` is the image size and `W` the image width.
  - If a tuple `(a, b)`, then two values `M`, `N` will be sampled from the discrete interval `[a..b]`. The dropout mask will then be generated at size `MxN` and upsampled to `HxW`.
  - If a `StochasticParameter`, then this parameter will be used to determine the sizes. It is expected to be discrete.

- **size_percent** (*None or float or tuple of float or imgaug.parameters.StochasticParameter, optional*) – The size of the lower resolution image from which to sample the dropout mask *in percent* of the input image. Note that this means that *lower* values of this parameter lead to *larger* areas being dropped (as any pixel in the lower resolution image will correspond to a larger area at the original resolution).
  - If `None` then *size_px* must be set.
  - If a float, then that value will always be used as the percentage of the height and width (relative to the original size). E.g. for value `p`, the mask will be sampled from `(p*H)x(p*W)` and later upsampled to `HxW`.
  - If a tuple `(a, b)`, then two values `m`, `n` will be sampled from the interval `(a, b)` and used as the size fractions, i.e the mask size will be `(m*H)x(n*W)`.
  - If a `StochasticParameter`, then this parameter will be used to sample the percentage values. It is expected to be continuous.

- **per_channel** (*bool or float or imgaug.parameters.StochasticParameter, optional*) – Whether to use (imagewise) the same sample(s) for all channels (`False`) or to sample value(s) for each channel (`True`). Setting this to `True` will therefore lead to different transformations per image *and* channel, otherwise only per image. If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`. If it is a `StochasticParameter` it is expected to produce samples with values between `0.0` and `1.0`, where values `>0.5` will lead to per-channel behaviour (i.e. same as `True`).

- **min_size** (*int, optional*) – Minimum height and width of the low resolution mask. If *size_percent* or *size_px* leads to a lower value than this, *min_size* will be used instead. This should never have a value of less than `2`, otherwise one may end up with a `1x1` low resolution mask, leading easily to the whole image being dropped.

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.CoarseDropout(0.02, size_percent=0.5)
```

Drops 2 percent of all pixels on a lower-resolution image that has 50 percent of the original image's size, leading to dropped areas that have roughly 2x2 pixels size.

```
>>> aug = iaa.CoarseDropout((0.0, 0.05), size_percent=(0.05, 0.5))
```

Generates a dropout mask at 5 to 50 percent of each input image's size. In that mask, 0 to 5 percent of all pixels are marked as being dropped. The mask is afterwards projected to the input image's size to apply the actual dropout operation.

```
>>> aug = iaa.CoarseDropout((0.0, 0.05), size_px=(2, 16))
```

Same as the previous example, but the lower resolution image has 2 to 16 pixels size. On images of e.g. `224x224` pixels in size this would lead to fairly large areas being dropped (height/width of ``224/2 to 224/16).

```
>>> aug = iaa.CoarseDropout(0.02, size_percent=0.5, per_channel=True)
```

Drops 2 percent of all pixels at 50 percent resolution (2x2 sizes) in a channel-wise fashion, i.e. it is unlikely for any pixel to have all channels set to zero (black pixels).

```
>>> aug = iaa.CoarseDropout(0.02, size_percent=0.5, per_channel=0.5)
```

Same as the previous example, but the *per_channel* feature is only active for 50 percent of all images.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.

Continued on next page

Table 65 – continued from previous page

augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.arithmetic.**CoarsePepper**(*p=0*, *size_px=None*, *size_percent=None*, *per_channel=False*, *min_size=4*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.arithmetic.ReplaceElementwise*

Replace rectangular areas in images with black-ish pixel noise.

dtype support:

```
See ``imgaug.augmenters.arithmetic.ReplaceElementwise``.
```

Parameters

- **p** (*float or tuple of float or list of float or imgaug.parameters.StochasticParameter, optional*) –

  Probability of changing a pixel to pepper noise.

  – If a float, then that value will always be used as the probability.

  – If a tuple `(a, b)`, then a probability will be sampled uniformly per image from the interval `[a, b]`.

  – If a list, then a random value will be sampled from that list per image.

  – If a `StochasticParameter`, then a lower-resolution mask will be sampled from that parameter per image. Any value `>0.5` in that mask will denote a spatial location that is to be replaced by pepper noise.

- **size_px** (*int or tuple of int or imgaug.parameters.StochasticParameter, optional*) – The size of the lower resolution image from which to sample the replacement mask in absolute pixel dimensions. Note that this means that *lower* values of this parameter lead to *larger* areas being replaced (as any pixel in the lower resolution image will correspond to a larger area at the original resolution).

  – If `None` then *size_percent* must be set.

  – If an integer, then that size will always be used for both height and width. E.g. a value of `3` would lead to a `3x3` mask, which is then upsampled to `HxW`, where `H` is the image size and `W` the image width.

  – If a tuple `(a, b)`, then two values `M`, `N` will be sampled from the discrete interval `[a..b]`. The mask will then be generated at size `MxN` and upsampled to `HxW`.

  – If a `StochasticParameter`, then this parameter will be used to determine the sizes. It is expected to be discrete.

- **size_percent** (*float or tuple of float or imgaug.parameters.StochasticParameter, optional*) – The size of the lower resolution image from which to sample the replacement mask *in percent* of the input image. Note that this means that *lower* values of this parameter lead to *larger* areas being replaced (as any pixel in the lower resolution image will correspond to a larger area at the original resolution).

  – If `None` then *size_px* must be set.

  – If a float, then that value will always be used as the percentage of the height and width (relative to the original size). E.g. for value `p`, the mask will be sampled from `(p*H)x(p*W)` and later upsampled to `HxW`.

  – If a tuple `(a, b)`, then two values `m`, `n` will be sampled from the interval `(a, b)` and used as the size fractions, i.e the mask size will be `(m*H)x(n*W)`.

  – If a `StochasticParameter`, then this parameter will be used to sample the percentage values. It is expected to be continuous.

- **per_channel** (*bool or float or imgaug.parameters.StochasticParameter, optional*) – Whether to use (imagewise) the same sample(s) for all channels (`False`) or to sample value(s) for each channel (`True`). Setting this to `True` will therefore lead to different transformations per image *and* channel, otherwise only per image. If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`. If it is a `StochasticParameter` it is expected to produce samples with values between `0.0` and `1.0`, where values `>0.5` will lead to per-channel behaviour (i.e. same as `True`).

- **min_size** (*int, optional*) – Minimum size of the low resolution mask, both width and height. If *size_percent* or *size_px* leads to a lower value than this, *min_size* will be used instead.

This should never have a value of less than 2, otherwise one may end up with a `1x1` low resolution mask, leading easily to the whole image being replaced.

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.CoarsePepper(0.05, size_percent=(0.01, 0.1))
```

Mark `5%` of all pixels in a mask to be replaced by pepper noise. The mask has `1%` to `10%` the size of the input image. The mask is then upscaled to the input image size, leading to large rectangular areas being marked as to be replaced. These areas are then replaced in the input image by pepper noise.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.

Continued on next page

Table 66 – continued from previous page

find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.arithmetic.**CoarseSalt**(*p=0*, *size_px=None*, *size_percent=None*, *per_channel=False*, *min_size=4*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.arithmetic.ReplaceElementwise*

Replace rectangular areas in images with white-ish pixel noise.

See also the similar `CoarseSaltAndPepper`.

dtype support:

```
See ``imgaug.augmenters.arithmetic.ReplaceElementwise``.
```

Parameters

- **p** (*float or tuple of float or list of float or imgaug.parameters.StochasticParameter, optional*) –

  Probability of changing a pixel to salt noise.

  - If a float, then that value will always be used as the probability.
  - If a tuple `(a, b)`, then a probability will be sampled uniformly per image from the interval `[a, b]`.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, then a lower-resolution mask will be sampled from that parameter per image. Any value `>0.5` in that mask will denote a spatial location that is to be replaced by salt noise.

- **size_px** (*int or tuple of int or imgaug.parameters.StochasticParameter, optional*) – The size of the lower resolution image from which to sample the replacement mask in absolute pixel dimensions. Note that this means that *lower* values of this parameter lead to *larger* areas being replaced (as any pixel in the lower resolution image will correspond to a larger area at the original resolution).
  - If `None` then *size_percent* must be set.
  - If an integer, then that size will always be used for both height and width. E.g. a value of `3` would lead to a `3x3` mask, which is then upsampled to `HxW`, where `H` is the image size and `W` the image width.
  - If a tuple `(a, b)`, then two values `M`, `N` will be sampled from the discrete interval `[a..b]`. The mask will then be generated at size `MxN` and upsampled to `HxW`.
  - If a `StochasticParameter`, then this parameter will be used to determine the sizes. It is expected to be discrete.
- **size_percent** (*float or tuple of float or imgaug.parameters.StochasticParameter, optional*) – The size of the lower resolution image from which to sample the replacement mask *in percent* of the input image. Note that this means that *lower* values of this parameter lead to *larger* areas being replaced (as any pixel in the lower resolution image will correspond to a larger area at the original resolution).
  - If `None` then *size_px* must be set.
  - If a float, then that value will always be used as the percentage of the height and width (relative to the original size). E.g. for value `p`, the mask will be sampled from `(p*H)x(p*W)` and later upsampled to `HxW`.
  - If a tuple `(a, b)`, then two values `m`, `n` will be sampled from the interval `(a, b)` and used as the size fractions, i.e the mask size will be `(m*H)x(n*W)`.
  - If a `StochasticParameter`, then this parameter will be used to sample the percentage values. It is expected to be continuous.
- **per_channel** (*bool or float or imgaug.parameters.StochasticParameter, optional*) – Whether to use (imagewise) the same sample(s) for all channels (`False`) or to sample value(s) for each channel (`True`). Setting this to `True` will therefore lead to different transformations per image *and* channel, otherwise only per image. If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`. If it is a `StochasticParameter` it is expected to produce samples with values between `0.0` and `1.0`, where values `>0.5` will lead to per-channel behaviour (i.e. same as `True`).
- **min_size** (*int, optional*) – Minimum height and width of the low resolution mask. If *size_percent* or *size_px* leads to a lower value than this, *min_size* will be used instead. This should never have a value of less than `2`, otherwise one may end up with a `1x1` low resolution mask, leading easily to the whole image being replaced.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.CoarseSalt(0.05, size_percent=(0.01, 0.1))
```

Mark 5% of all pixels in a mask to be replaced by salt noise. The mask has 1% to 10% the size of the input image. The mask is then upscaled to the input image size, leading to large rectangular areas being marked as to be replaced. These areas are then replaced in the input image by salt noise.

Methods

__call__(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
augment(self[, return_batch, hooks])	Augment a batch.
augment_batch(self, batch[, hooks])	Augment a single batch.
augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ...[, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.

Continued on next page

Table 67 – continued from previous page

remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.arithmetic.**CoarseSaltAndPepper**(*p=0*, *size_px=None*, *size_percent=None*, *per_channel=False*, *min_size=4*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.arithmetic.ReplaceElementwise*

Replace rectangular areas in images with white/black-ish pixel noise.

This adds salt and pepper noise (noisy white-ish and black-ish pixels) to rectangular areas within the image. Note that this means that within these rectangular areas the color varies instead of each rectangle having only one color.

See also the similar `CoarseDropout`.

TODO replace dtype support with uint8 only, because replacement is geared towards that value range

dtype support:

```
See ``imgaug.augmenters.arithmetic.ReplaceElementwise``.
```

Parameters

- **p** (*float or tuple of float or list of float or imgaug.parameters.StochasticParameter, optional*) –

  Probability of changing a pixel to salt/pepper noise.

  - If a float, then that value will always be used as the probability.
  - If a tuple `(a, b)`, then a probability will be sampled uniformly per image from the interval `[a, b]`.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, then a lower-resolution mask will be sampled from that parameter per image. Any value `>0.5` in that mask will denote a spatial location that is to be replaced by salt and pepper noise.

- **size_px** (*int or tuple of int or imgaug.parameters.StochasticParameter, optional*) – The size of the lower resolution image from which to sample the replacement mask in absolute pixel dimensions. Note that this means that *lower* values of this parameter lead to *larger* areas being replaced (as any pixel in the lower resolution image will correspond to a larger area at the original resolution).

  - If None then *size_percent* must be set.
  - If an integer, then that size will always be used for both height and width. E.g. a value of 3 would lead to a 3x3 mask, which is then upsampled to HxW, where H is the image size and W the image width.
  - If a tuple (a, b), then two values M, N will be sampled from the discrete interval [a..b]. The mask will then be generated at size MxN and upsampled to HxW.
  - If a StochasticParameter, then this parameter will be used to determine the sizes. It is expected to be discrete.

- **size_percent** (*float or tuple of float or imgaug.parameters.StochasticParameter, optional*) – The size of the lower resolution image from which to sample the replacement mask *in percent* of the input image. Note that this means that *lower* values of this parameter lead to *larger* areas being replaced (as any pixel in the lower resolution image will correspond to a larger area at the original resolution).
  - If None then *size_px* must be set.
  - If a float, then that value will always be used as the percentage of the height and width (relative to the original size). E.g. for value p, the mask will be sampled from (p*H)x(p*W) and later upsampled to HxW.
  - If a tuple (a, b), then two values m, n will be sampled from the interval (a, b) and used as the size fractions, i.e the mask size will be (m*H)x(n*W).
  - If a StochasticParameter, then this parameter will be used to sample the percentage values. It is expected to be continuous.
- **per_channel** (*bool or float or imgaug.parameters.StochasticParameter, optional*) – Whether to use (imagewise) the same sample(s) for all channels (False) or to sample value(s) for each channel (True). Setting this to True will therefore lead to different transformations per image *and* channel, otherwise only per image. If this value is a float p, then for p percent of all images *per_channel* will be treated as True. If it is a StochasticParameter it is expected to produce samples with values between 0.0 and 1.0, where values >0.5 will lead to per-channel behaviour (i.e. same as True).
- **min_size** (*int, optional*) – Minimum height and width of the low resolution mask. If *size_percent* or *size_px* leads to a lower value than this, *min_size* will be used instead. This should never have a value of less than 2, otherwise one may end up with a 1x1 low resolution mask, leading easily to the whole image being replaced.
- **name** (*None or str, optional*) – See imgaug.augmenters.meta.Augmenter.__init__().
- **deterministic** (*bool, optional*) – See imgaug.augmenters.meta.Augmenter.__init__().
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See imgaug.augmenters.meta.Augmenter.__init__().

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.CoarseSaltAndPepper(0.05, size_percent=(0.01, 0.1))
```

Marks 5% of all pixels in a mask to be replaced by salt/pepper noise. The mask has 1% to 10% the size of the input image. The mask is then upscaled to the input image size, leading to large rectangular areas being marked as to be replaced. These areas are then replaced in the input image by salt/pepper noise.

```
>>> aug = iaa.CoarseSaltAndPepper(0.05, size_px=(4, 16))
```

Same as in the previous example, but the replacement mask before upscaling has a size between 4x4 and 16x16 pixels (the axis sizes are sampled independently, i.e. the mask may be rectangular).

```
>>> aug = iaa.CoarseSaltAndPepper(
>>>     0.05, size_percent=(0.01, 0.1), per_channel=True)
```

Same as in the first example, but mask and replacement are each sampled independently per image channel.

Methods

__call__(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
augment(self[, return_batch, hooks])	Augment a batch.
augment_batch(self, batch[, hooks])	Augment a single batch.
augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ...[, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.

Continued on next page

Table 68 – continued from previous page

localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

imgaug.augmenters.arithmetic.**ContrastNormalization**(*alpha=1.0*, *per_channel=False*, *name=None*, *deterministic=False*, *random_state=None*)

Deprecated. Use `imgaug.contrast.LinearContrast` instead.

Change the contrast of images.

dtype support:

See `imgaug.augmenters.contrast.LinearContrast`.

Parameters

- **alpha** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Strength of the contrast normalization. Higher values than 1.0 lead to higher contrast, lower values decrease the contrast.
  - If a number, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value will be sampled per image uniformly from the interval `[a, b]` and be used as the alpha value.
  - If a list, then a random value will be picked per image from that list.
  - If a `StochasticParameter`, then this parameter will be used to sample the alpha value per image.
- **per_channel** (*bool or float or imgaug.parameters.StochasticParameter, optional*) – Whether to use (imagewise) the same sample(s) for all channels (`False`) or to sample value(s) for each channel (`True`). Setting this to `True` will therefore lead to different transformations per image *and* channel, otherwise only per image. If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`. If it is a `StochasticParameter` it is expected to produce samples with values between `0.0` and `1.0`, where values `>0.5` will lead to per-channel behaviour (i.e. same as `True`).
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or*

numpy.random.RandomState, optional) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> iaa.ContrastNormalization((0.5, 1.5))
```

Decreases oder improves contrast per image by a random factor between `0.5` and `1.5`. The factor `0.5` means that any difference from the center value (i.e. 128) will be halved, leading to less contrast.

```
>>> iaa.ContrastNormalization((0.5, 1.5), per_channel=0.5)
```

Same as before, but for 50 percent of all images the normalization is done independently per channel (i.e. factors can vary per channel for the same image). In the other 50 percent of all images, the factor is the same for all channels.

class imgaug.augmenters.arithmetic.**Dropout**(*p=0*, *per_channel=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.arithmetic.MultiplyElementwise*

Set a fraction of pixels in images to zero.

dtype support:

```
See ``imgaug.augmenters.arithmetic.MultiplyElementwise``.
```

Parameters

- **p** (*float or tuple of float or imgaug.parameters.StochasticParameter, optional*) –

  The probability of any pixel being dropped (i.e. to set it to zero).

  - If a float, then that value will be used for all images. A value of `1.0` would mean that all pixels will be dropped and `0.0` that no pixels will be dropped. A value of `0.05` corresponds to `5` percent of all pixels being dropped.
  - If a tuple `(a, b)`, then a value `p` will be sampled from the interval `[a, b]` per image and be used as the pixel's dropout probability.
  - If a `StochasticParameter`, then this parameter will be used to determine per pixel whether it should be *kept* (sampled value of `>0.5`) or shouldn't be kept (sampled value of `<=0.5`). If you instead want to provide the probability as a stochastic parameter, you can usually do `imgaug.parameters.Binomial(1-p)` to convert parameter *p* to a 0/1 representation.

- **per_channel** (*bool or float or imgaug.parameters.StochasticParameter, optional*) – Whether to use (imagewise) the same sample(s) for all channels (`False`) or to sample value(s) for each channel (`True`). Setting this to `True` will therefore lead to different transformations per image *and* channel, otherwise only per image. If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`. If it is a `StochasticParameter` it is expected to produce samples with values between `0.0` and `1.0`, where values `>0.5` will lead to per-channel behaviour (i.e. same as `True`).
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Dropout(0.02)
```

Drops 2 percent of all pixels.

```
>>> aug = iaa.Dropout((0.0, 0.05))
```

Drops in each image a random fraction of all pixels, where the fraction is uniformly sampled from the interval `[0.0, 0.05]`.

```
>>> aug = iaa.Dropout(0.02, per_channel=True)
```

Drops 2 percent of all pixels in a channelwise fashion, i.e. it is unlikely for any pixel to have all channels set to zero (black pixels).

```
>>> aug = iaa.Dropout(0.02, per_channel=0.5)
```

Identical to the previous example, but the *per_channel* feature is only active for 50 percent of all images.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).

Continued on next page

Table 69 – continued from previous page

deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.arithmetic.**ImpulseNoise**(*p=0*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.arithmetic.SaltAndPepper*

Add impulse noise to images.

This is identical to `SaltAndPepper`, except that *per_channel* is always set to `True`.

dtype support:

```
See ``imgaug.augmenters.arithmetic.SaltAndPepper``.
```

Parameters

- **p** (*float or tuple of float or list of float or imgaug.parameters.StochasticParameter, optional*) –

  Probability of replacing a pixel to impulse noise.

  – If a float, then that value will always be used as the probability.

  – If a tuple `(a, b)`, then a probability will be sampled uniformly per image from the interval `[a, b]`.

  – If a list, then a random value will be sampled from that list per image.

  – If a `StochasticParameter`, then a image-sized mask will be sampled from that parameter per image. Any value `>0.5` in that mask will be replaced with impulse noise noise.

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.ImpulseNoise(0.1)
```

Replace `10%` of all pixels with impulse noise.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.

Continued on next page

Table 70 – continued from previous page

get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters	

class imgaug.augmenters.arithmetic.**Invert**(*p=0*, *per_channel=False*, *min_value=None*, *max_value=None*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Invert all values in images, e.g. turn `5` into `255-5=250`.

For the standard value range of 0-255 it converts `0` to `255`, `255` to `0` and `10` to `(255-10)=245`. Let `M` be the maximum value possible, `m` the minimum value possible, `v` a value. Then the distance of `v` to `m` is `d=abs(v-m)` and the new value is given by `v'=M-d`.

dtype support:

```
See :func:`imgaug.augmenters.arithmetic.invert`.
```

Parameters

- **p** (*float or imgaug.parameters.StochasticParameter, optional*) –

  The probability of an image to be inverted.

  – If a float, then that probability will be used for all images, i.e. *p* percent of all images will be inverted.

  – If a `StochasticParameter`, then that parameter will be queried per image and is expected to return values in the interval `[0.0, 1.0]`, where values `>0.5` mean that the image is supposed to be inverted. Recommended to be some form of `imgaug.parameters.Binomial`.

- **per_channel** (*bool or float or imgaug.parameters.StochasticParameter, optional*) – Whether to use (imagewise) the same sample(s) for all channels (`False`) or to sample value(s) for each channel (`True`). Setting this to `True` will therefore lead to different transformations per image *and* channel, otherwise only per image. If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`. If it is a `StochasticParameter` it is expected to produce samples with values between `0.0` and `1.0`, where values `>0.5` will lead to per-channel behaviour (i.e. same as `True`).

- **min_value** (*None or number, optional*) – Minimum of the value range of input images, e.g. `0` for `uint8` images. If set to `None`, the value will be automatically derived from the image's dtype.
- **max_value** (*None or number, optional*) – Maximum of the value range of input images, e.g. `255` for `uint8` images. If set to `None`, the value will be automatically derived from the image's dtype.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Invert(0.1)
```

Inverts the colors in `10` percent of all images.

```
>>> aug = iaa.Invert(0.1, per_channel=True)
```

Inverts the colors in `10` percent of all image channels. This may or may not lead to multiple channels in an image being inverted.

```
>>> aug = iaa.Invert(0.1, per_channel=0.5)
```

Identical to the previous example, but the *per_channel* feature is only active for 50 percent of all images.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.

Continued on next page

Table 71 – continued from previous page

augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

ALLOW_DTYPES_CUSTOM_MINMAX = [dtype('uint8'), dtype('uint16'), dtype('uint32'), dtype(

get_parameters(*self*)

class imgaug.augmenters.arithmetic.**JpegCompression**(*compression=50*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Degrade the quality of images by JPEG-compressing them.

During JPEG compression, high frequency components (e.g. edges) are removed. With low compression (strength) only the highest frequency components are removed, while very high compression (strength) will lead to only the lowest frequency components "surviving". This lowers the image quality. For more details, see https://en.wikipedia.org/wiki/Compression_artifact.

Note that this augmenter still returns images as numpy arrays (i.e. saves the images with JPEG compression and then reloads them into arrays). It does not return the raw JPEG file content.

dtype support:

```
See :func:`imgaug.augmenters.arithmetic.compress_jpeg`.
```

Parameters

- **compression** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Degree of compression used during JPEG compression within value range `[0, 100]`. Higher values denote stronger compression and will cause low-frequency components to disappear. Note that JPEG's compression strength is also often set as a *quality*, which is the inverse of this parameter. Common choices for the *quality* setting are around 80 to 95, depending on the image. This translates here to a *compression* parameter of around 20 to 5.
  - If a single number, then that value always will be used as the compression.
  - If a tuple `(a, b)`, then the compression will be a value sampled uniformly from the interval `[a, b]`.
  - If a list, then a random value will be sampled from that list per image and used as the compression.
  - If a `StochasticParameter`, then `N` samples will be drawn from that parameter per `N` input images, each representing the compression for the `n`-th image.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.JpegCompression(compression=(70, 99))
```

Remove high frequency components in images via JPEG compression with a *compression strength* between `70` and `99` (randomly and uniformly sampled per image). This corresponds to a (very low) *quality* setting of `1` to `30`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *`imgaug.augmenters.meta.Augmenter.augment()`*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.

Continued on next page

Table 72 – continued from previous page

augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.arithmetic.**Multiply**(*mul=1.0*, *per_channel=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Multiply all pixels in an image with a random value sampled once per image.

This augmenter can be used to make images lighter or darker.

dtype support:

```
See :func:`imgaug.augmenters.arithmetic.multiply_scalar`.
```

Parameters

- **mul** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) –

  The value with which to multiply the pixel values in each image.

  – If a number, then that value will always be used.

  – If a tuple `(a, b)`, then a value from the interval `[a, b]` will be sampled per image and used for all pixels.

  – If a list, then a random value will be sampled from that list per image.

  – If a `StochasticParameter`, then that parameter will be used to sample a new value per image.

- **per_channel** (*bool or float or imgaug.parameters.StochasticParameter, optional*) – Whether to use (imagewise) the same sample(s) for all channels (`False`) or to sample value(s) for each channel (`True`). Setting this to `True` will therefore lead to different transformations per image *and* channel, otherwise only per image. If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`. If it is a `StochasticParameter` it is expected to produce samples with values between `0.0` and `1.0`, where values `>0.5` will lead to per-channel behaviour (i.e. same as `True`).

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Multiply(2.0)
```

Multiplies all images by a factor of `2`, making the images significantly brighter.

```
>>> aug = iaa.Multiply((0.5, 1.5))
```

Multiplies images by a random value sampled uniformly from the interval `[0.5, 1.5]`, making some images darker and others brighter.

```
>>> aug = iaa.Multiply((0.5, 1.5), per_channel=True)
```

Identical to the previous example, but the sampled multipliers differ by image *and* channel, instead of only by image.

```
>>> aug = iaa.Multiply((0.5, 1.5), per_channel=0.5)
```

Identical to the previous example, but the *per_channel* feature is only active for 50 percent of all images.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
`remove_augmenters_inplace`(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
`reseed`(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
`show_grid`(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
`to_deterministic`(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.arithmetic.**MultiplyElementwise**(*mul=1.0*, *per_channel=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Multiply image pixels with values that are pixelwise randomly sampled.

While the `Multiply` Augmenter uses a constant multiplier *per image* (and optionally channel), this augmenter samples the multipliers to use per image and *per pixel* (and optionally per channel).

dtype support:

```
See :func:`imgaug.augmenters.arithmetic.multiply_elementwise`.
```

Parameters

- **mul** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) –

  The value with which to multiply pixel values in the image.

  - If a number, then that value will always be used.
  - If a tuple `(a, b)`, then a value from the interval `[a, b]` will be sampled per image and pixel.
  - If a list, then a random value will be sampled from that list per image and pixel.
  - If a `StochasticParameter`, then that parameter will be used to sample a new value per image and pixel.

- **per_channel** (*bool or float or imgaug.parameters.StochasticParameter, optional*) – Whether to use (imagewise) the same sample(s) for all channels (`False`) or to sample value(s) for each channel (`True`). Setting this to `True` will therefore lead to different transformations per image *and* channel, otherwise only per image. If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`. If it is a `StochasticParameter` it is expected to produce samples with values between `0.0` and `1.0`, where values `>0.5` will lead to per-channel behaviour (i.e. same as `True`).

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.MultiplyElementwise(2.0)
```

Multiply all images by a factor of `2.0`, making them significantly bighter.

```
>>> aug = iaa.MultiplyElementwise((0.5, 1.5))
```

Samples per image and pixel uniformly a value from the interval `[0.5, 1.5]` and multiplies the pixel with that value.

```
>>> aug = iaa.MultiplyElementwise((0.5, 1.5), per_channel=True)
```

Samples per image and pixel *and channel* uniformly a value from the interval `[0.5, 1.5]` and multiplies the pixel with that value. Therefore, used multipliers may differ between channels of the same pixel.

```
>>> aug = iaa.MultiplyElementwise((0.5, 1.5), per_channel=0.5)
```

Identical to the previous example, but the *per_channel* feature is only active for 50 percent of all images.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.

Continued on next page

Table 74 – continued from previous page

localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.arithmetic.**Pepper**(*p=0*, *per_channel=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.arithmetic.ReplaceElementwise*

Replace pixels in images with pepper noise, i.e. black-ish pixels.

This augmenter is similar to SaltAndPepper, but adds no salt noise to images.

This augmenter is similar to Dropout, but slower and the black pixels are not uniformly black.

dtype support:

```
See ``imgaug.augmenters.arithmetic.ReplaceElementwise``.
```

Parameters

- **p** (*float or tuple of float or list of float or imgaug.parameters.StochasticParameter, optional*) –

  Probability of replacing a pixel with pepper noise.

  - If a float, then that value will always be used as the probability.
  - If a tuple (a, b), then a probability will be sampled uniformly per image from the interval [a, b].
  - If a list, then a random value will be sampled from that list per image.
  - If a StochasticParameter, then a image-sized mask will be sampled from that parameter per image. Any value >0.5 in that mask will be replaced with pepper noise.

- **per_channel** (*bool or float or imgaug.parameters.StochasticParameter, optional*) – Whether to use (imagewise) the same sample(s) for all channels (False) or to sample value(s) for each channel (True). Setting this to True will therefore lead to different transformations per image *and* channel, otherwise only per image. If this value is a float p, then for p percent of all images *per_channel* will be treated as True. If it is a StochasticParameter it is expected to produce samples with values between 0.0 and 1.0, where values >0.5 will lead to per-channel behaviour (i.e. same as True).

- **name** (*None or str, optional*) – See imgaug.augmenters.meta.Augmenter.__init__().

- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Pepper(0.05)
```

Replace `5%` of all pixels with pepper noise (black-ish colors).

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.

Continued on next page

Table 75 – continued from previous page

localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.arithmetic.**ReplaceElementwise**(*mask*, *replacement*, *per_channel=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Replace pixels in an image with new values.

dtype support:

```
See :func:`imgaug.augmenters.arithmetic.replace_elementwise_`.
```

Parameters

- **mask** (*float or tuple of float or list of float or imgaug.parameters.StochasticParameter*) – Mask that indicates the pixels that are supposed to be replaced. The mask will be binarized using a threshold of `0.5`. A value of `1` then indicates a pixel that is supposed to be replaced.
  - If this is a float, then that value will be used as the probability of being a `1` in the mask (sampled per image and pixel) and hence being replaced.
  - If a tuple `(a, b)`, then the probability will be uniformly sampled per image from the interval `[a, b]`.
  - If a list, then a random value will be sampled from that list per image and pixel.
  - If a `StochasticParameter`, then this parameter will be used to sample a mask per image.
- **replacement** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – The replacement to use at all locations that are marked as `1` in the mask.
  - If this is a number, then that value will always be used as the replacement.
  - If a tuple `(a, b)`, then the replacement will be sampled uniformly per image and pixel from the interval `[a, b]`.
  - If a list, then a random value will be sampled from that list per image and pixel.

  - If a `StochasticParameter`, then this parameter will be used sample replacement values per image and pixel.

- **per_channel** (*bool or float or imgaug.parameters.StochasticParameter, optional*) – Whether to use (imagewise) the same sample(s) for all channels (`False`) or to sample value(s) for each channel (`True`). Setting this to `True` will therefore lead to different transformations per image *and* channel, otherwise only per image. If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`. If it is a `StochasticParameter` it is expected to produce samples with values between `0.0` and `1.0`, where values `>0.5` will lead to per-channel behaviour (i.e. same as `True`).
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = ReplaceElementwise(0.05, [0, 255])
```

Replaces `5` percent of all pixels in each image by either `0` or `255`.

```
>>> import imgaug.augmenters as iaa
>>> aug = ReplaceElementwise(0.1, [0, 255], per_channel=0.5)
```

For `50%` of all images, replace `10%` of all pixels with either the value `0` or the value `255` (same as in the previous example). For the other `50%` of all images, replace *channelwise* `10%` of all pixels with either the value `0` or the value `255`. So, it will be very rare for each pixel to have all channels replaced by `255` or `0`.

```
>>> import imgaug.augmenters as iaa
>>> import imgaug.parameters as iap
>>> aug = ReplaceElementwise(0.1, iap.Normal(128, 0.4*128), per_channel=0.5)
```

Replace `10%` of all pixels by gaussian noise centered around `128`. Both the replacement mask and the gaussian noise are sampled channelwise for `50%` of all images.

```
>>> import imgaug.augmenters as iaa
>>> import imgaug.parameters as iap
>>> aug = ReplaceElementwise(
>>>     iap.FromLowerResolution(iap.Binomial(0.1), size_px=8),
>>>     iap.Normal(128, 0.4*128),
>>>     per_channel=0.5)
```

Replace `10%` of all pixels by gaussian noise centered around `128`. Sample the replacement mask at a lower resolution (`8x8` pixels) and upscale it to the image size, resulting in coarse areas being replaced by gaussian noise.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
`remove_augmenters_inplace`(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
`reseed`(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
`show_grid`(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
`to_deterministic`(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.arithmetic.**Salt**(*p=0*, *per_channel=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.arithmetic.ReplaceElementwise*

Replace pixels in images with salt noise, i.e. white-ish pixels.

This augmenter is similar to `SaltAndPepper`, but adds no pepper noise to images.

dtype support:

```
See ``imgaug.augmenters.arithmetic.ReplaceElementwise``.
```

Parameters

- **p** (*float or tuple of float or list of float or imgaug.parameters.StochasticParameter, optional*) –

  Probability of replacing a pixel with salt noise.

  - If a float, then that value will always be used as the probability.
  - If a tuple `(a, b)`, then a probability will be sampled uniformly per image from the interval `[a, b]`.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, then a image-sized mask will be sampled from that parameter per image. Any value `>0.5` in that mask will be replaced with salt noise.

- **per_channel** (*bool or float or imgaug.parameters.StochasticParameter, optional*) – Whether to use (imagewise) the same sample(s) for all channels (`False`) or to sample value(s) for each channel (`True`). Setting this to `True` will therefore lead to different transformations per image *and* channel, otherwise only per image. If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`. If it is a `StochasticParameter` it is expected to produce samples with values between `0.0` and `1.0`, where values `>0.5` will lead to per-channel behaviour (i.e. same as `True`).

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Salt(0.05)
```

Replace `5%` of all pixels with salt noise (white-ish colors).

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
`remove_augmenters_inplace`(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
`reseed`(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
`show_grid`(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
`to_deterministic`(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters	

class imgaug.augmenters.arithmetic.**SaltAndPepper**(*p=0*, *per_channel=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.arithmetic.ReplaceElementwise*

Replace pixels in images with salt/pepper noise (white/black-ish colors).

dtype support:

```
See ``imgaug.augmenters.arithmetic.ReplaceElementwise``.
```

Parameters

- **p** (*float or tuple of float or list of float or imgaug.parameters.StochasticParameter, optional*) –

  Probability of replacing a pixel to salt/pepper noise.

  – If a float, then that value will always be used as the probability.

  – If a tuple `(a, b)`, then a probability will be sampled uniformly per image from the interval `[a, b]`.

  – If a list, then a random value will be sampled from that list per image.

  – If a `StochasticParameter`, then a image-sized mask will be sampled from that parameter per image. Any value `>0.5` in that mask will be replaced with salt and pepper noise.

- **per_channel** (*bool or float or imgaug.parameters.StochasticParameter, optional*) – Whether to use (imagewise) the same sample(s) for all channels (`False`) or to sample value(s) for each channel (`True`). Setting this to `True` will therefore lead to different transformations per image *and* channel, otherwise only per image. If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`. If it is a `StochasticParameter` it is expected to produce samples with values between `0.0` and `1.0`, where values `>0.5` will lead to per-channel behaviour (i.e. same as `True`).

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.SaltAndPepper(0.05)
```

Replace `5%` of all pixels with salt and pepper noise.

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.SaltAndPepper(0.05, per_channel=True)
```

Replace *channelwise* `5%` of all pixels with salt and pepper noise.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
`remove_augmenters_inplace`(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
`reseed`(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
`show_grid`(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
`to_deterministic`(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

imgaug.augmenters.arithmetic.**add_elementwise**(*image*, *values*)

Add an array of values to an image.

This method ensures that `uint8` does not overflow during the addition.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: limited; tested (1)
* ``uint32``: no
* ``uint64``: no
* ``int8``: limited; tested (1)
* ``int16``: limited; tested (1)
* ``int32``: no
* ``int64``: no
* ``float16``: limited; tested (1)
* ``float32``: limited; tested (1)
* ``float64``: no
* ``float128``: no
* ``bool``: limited; tested (1)

- (1) Non-uint8 dtypes can overflow. For floats, this can result
      in +/-inf.
```

Parameters

- **image** (*ndarray*) – Image array of shape `(H,W,[C])`.
- **values** (*ndarray*) – The values to add to the image. Expected to have the same height and width as *image* and either no channels or one channel or the same number of channels as *image*.

Returns Image with values added to it.

Return type ndarray

imgaug.augmenters.arithmetic.**add_scalar**(*image*, *value*)

Add a single scalar value or one scalar value per channel to an image.

This method ensures that `uint8` does not overflow during the addition.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: limited; tested (1)
* ``uint32``: no
* ``uint64``: no
* ``int8``: limited; tested (1)
* ``int16``: limited; tested (1)
* ``int32``: no
* ``int64``: no
* ``float16``: limited; tested (1)
* ``float32``: limited; tested (1)
* ``float64``: no
* ``float128``: no
* ``bool``: limited; tested (1)

- (1) Non-uint8 dtypes can overflow. For floats, this can result
      in +/-inf.
```

Parameters

- **image** (*ndarray*) – Image array of shape `(H,W,[C])`. If *value* contains more than one value, the shape of the image is expected to be `(H,W,C)`.
- **value** (*number or ndarray*) – The value to add to the image. Either a single value or an array containing exactly one component per channel, i.e. `C` components.

Returns Image with value added to it.

Return type ndarray

imgaug.augmenters.arithmetic.**compress_jpeg**(*image*, *compression*)

Compress an image using jpeg compression.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: ?
* ``uint32``: ?
* ``uint64``: ?
* ``int8``: ?
* ``int16``: ?
* ``int32``: ?
* ``int64``: ?
* ``float16``: ?
* ``float32``: ?
* ``float64``: ?
* ``float128``: ?
* ``bool``: ?
```

Parameters

- **image** (*ndarray*) – Image of dtype `uint8` and shape `(H,W,[C])`. If `C` is provided, it must be `1` or `3`.
- **compression** (*int*) – Strength of the compression in the interval `[0, 100]`.

Returns Input image after applying jpeg compression to it and reloading the result into a new array. Same shape and dtype as the input.

Return type ndarray

imgaug.augmenters.arithmetic.**invert**(*image*, *min_value=None*, *max_value=None*)

Invert an array.

dtype support:

```
if (min_value=None and max_value=None)::

    * ``uint8``: yes; fully tested
    * ``uint16``: yes; tested
    * ``uint32``: yes; tested
    * ``uint64``: yes; tested
    * ``int8``: yes; tested
    * ``int16``: yes; tested
    * ``int32``: yes; tested
    * ``int64``: yes; tested
    * ``float16``: yes; tested
    * ``float32``: yes; tested
```

(continues on next page)

(continued from previous page)

```
    * ``float64``: yes; tested
    * ``float128``: yes; tested
    * ``bool``: yes; tested

if (min_value!=None or max_value!=None)::

    * ``uint8``: yes; fully tested
    * ``uint16``: yes; tested
    * ``uint32``: yes; tested
    * ``uint64``: no (1)
    * ``int8``: yes; tested
    * ``int16``: yes; tested
    * ``int32``: yes; tested
    * ``int64``: no (1)
    * ``float16``: yes; tested
    * ``float32``: yes; tested
    * ``float64``: no (1)
    * ``float128``: no (2)
    * ``bool``: no (3)

    - (1) Not allowed due to numpy's clip converting from ``uint64`` to
          ``float64``.
    - (2) Not allowed as int/float have to be increased in resolution
          when using min/max values.
    - (3) Not tested.
    - (4) Makes no sense when using min/max values.
```

Parameters

- **image** (*ndarray*) – Image array of shape `(H,W,[C])`.
- **min_value** (*None or number, optional*) – Minimum of the value range of input images, e.g. `0` for `uint8` images. If set to `None`, the value will be automatically derived from the image's dtype.
- **max_value** (*None or number, optional*) – Maximum of the value range of input images, e.g. `255` for `uint8` images. If set to `None`, the value will be automatically derived from the image's dtype.

Returns Inverted image.

Return type ndarray

imgaug.augmenters.arithmetic.**multiply_elementwise**(*image*, *multipliers*)

Multiply an image with an array of values.

This method ensures that `uint8` does not overflow during the addition.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: limited; tested (1)
* ``uint32``: no
* ``uint64``: no
* ``int8``: limited; tested (1)
* ``int16``: limited; tested (1)
* ``int32``: no
* ``int64``: no
```

(continues on next page)

(continued from previous page)

```
* ``float16``: limited; tested (1)
* ``float32``: limited; tested (1)
* ``float64``: no
* ``float128``: no
* ``bool``: limited; tested (1)

- (1) Non-uint8 dtypes can overflow. For floats, this can result
      in +/-inf.

Note: tests were only conducted for rather small multipliers, around
``-10.0`` to ``+10.0``.

In general, the multipliers sampled from `multipliers` must be in a
value range that corresponds to the input image's dtype. E.g. if the
input image has dtype ``uint16`` and the samples generated from
`multipliers` are ``float64``, this function will still force all
samples to be within the value range of ``float16``, as it has the
same number of bytes (two) as ``uint16``. This is done to make
overflows less likely to occur.
```

Parameters

- **image** (*ndarray*) – Image array of shape `(H,W,[C])`.
- **multipliers** (*ndarray*) – The multipliers with which to multiply the image. Expected to have the same height and width as *image* and either no channels or one channel or the same number of channels as *image*.

Returns Image, multiplied by *multipliers*.

Return type ndarray

imgaug.augmenters.arithmetic.**multiply_scalar**(*image*, *multiplier*)

Multiply an image by a single scalar or one scalar per channel.

This method ensures that `uint8` does not overflow during the multiplication.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: limited; tested (1)
* ``uint32``: no
* ``uint64``: no
* ``int8``: limited; tested (1)
* ``int16``: limited; tested (1)
* ``int32``: no
* ``int64``: no
* ``float16``: limited; tested (1)
* ``float32``: limited; tested (1)
* ``float64``: no
* ``float128``: no
* ``bool``: limited; tested (1)

- (1) Non-uint8 dtypes can overflow. For floats, this can result in
      +/-inf.

Note: tests were only conducted for rather small multipliers, around
``-10.0`` to ``+10.0``.
```

(continues on next page)

(continued from previous page)

```
In general, the multipliers sampled from `multiplier` must be in a
value range that corresponds to the input image's dtype. E.g. if the
input image has dtype ``uint16`` and the samples generated from
`multiplier` are ``float64``, this function will still force all
samples to be within the value range of ``float16``, as it has the
same number of bytes (two) as ``uint16``. This is done to make
overflows less likely to occur.
```

Parameters

- **image** (*ndarray*) – Image array of shape `(H,W,[C])`. If *value* contains more than one value, the shape of the image is expected to be `(H,W,C)`.
- **multiplier** (*number or ndarray*) – The multiplier to use. Either a single value or an array containing exactly one component per channel, i.e. `C` components.

Returns Image, multiplied by *multiplier*.

Return type ndarray

imgaug.augmenters.arithmetic.**replace_elementwise_**(*image*, *mask*, *replacements*)

Replace components in an image array with new values.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; tested
* ``uint32``: yes; tested
* ``uint64``: no (1)
* ``int8``: yes; tested
* ``int16``: yes; tested
* ``int32``: yes; tested
* ``int64``: no (2)
* ``float16``: yes; tested
* ``float32``: yes; tested
* ``float64``: yes; tested
* ``float128``: no
* ``bool``: yes; tested

- (1) ``uint64`` is currently not supported, because
      :func:`imgaug.dtypes.clip_to_dtype_value_range_()` does not
      support it, which again is because numpy.clip() seems to not
      support it.
- (2) `int64` is disallowed due to being converted to `float64`
      by :func:`numpy.clip` since 1.17 (possibly also before?).
```

Parameters

- **image** (*ndarray*) – Image array of shape `(H,W,[C])`.
- **mask** (*ndarray*) – Mask of shape `(H,W,[C])` denoting which components to replace. If `C` is provided, it must be `1` or match the `C` of *image*. May contain floats in the interval `[0.0, 1.0]`.
- **replacements** (*iterable*) – Replacements to place in *image* at the locations defined by *mask*. This 1-dimensional iterable must contain exactly as many values as there are replaced components in *image*.

Returns Image with replaced components.

Return type ndarray

13.18 imgaug.augmenters.blend

Augmenters that blend two images with each other.

Do not import directly from this file, as the categorization is not final. Use instead

```
from imgaug import augmenters as iaa
```

and then e.g.

```
seq = iaa.Sequential([
    iaa.Alpha(0.5, iaa.Add((-5, 5)))
])
```

List of augmenters:

- Alpha
- AlphaElementwise
- SimplexNoiseAlpha
- FrequencyNoiseAlpha

class imgaug.augmenters.blend.**Alpha**(*factor=0*, *first=None*, *second=None*, *per_channel=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Alpha-blend two image sources using an alpha/opacity value.

The two image sources can be imagined as branches. If a source is not given, it is automatically the same as the input. Let A be the first branch and B be the second branch. Then the result images are defined as `factor * A + (1-factor) * B`, where `factor` is an overlay factor.

Note: It is not recommended to use `Alpha` with augmenters that change the geometry of images (e.g. horizontal flips, affine transformations) if you *also* want to augment coordinates (e.g. keypoints, polygons, ...), as it is unclear which of the two coordinate results (first or second branch) should be used as the coordinates after augmentation.

Currently, if `factor >= 0.5` (per image), the results of the first branch are used as the new coordinates, otherwise the results of the second branch.

---

dtype support:

```
See :func:`imgaug.augmenters.blend.blend_alpha`.
```

Parameters

- **factor** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Weighting of the results of the first branch. Values close to `0` mean that the results from the second branch (see parameter *second*) make up most of the final image.
  - If float, then that value will be used for all images.

  - If tuple `(a, b)`, then a random value from the interval `[a, b]` will be sampled per image.
  - If a list, then a random value will be picked from that list per image.
  - If `StochasticParameter`, then that parameter will be used to sample a value per image.

- **first** (*None or imgaug.augmenters.meta.Augmenter or iterable of imgaug.augmenters.meta.Augmenter, optional*) –

  Augmenter(s) that make up the first of the two branches.

  - If `None`, then the input images will be reused as the output of the first branch.
  - If `Augmenter`, then that augmenter will be used as the branch.
  - If iterable of `Augmenter`, then that iterable will be converted into a `Sequential` and used as the augmenter.

- **second** (*None or imgaug.augmenters.meta.Augmenter or iterable of imgaug.augmenters.meta.Augmenter, optional*) –

  Augmenter(s) that make up the second of the two branches.

  - If `None`, then the input images will be reused as the output of the second branch.
  - If `Augmenter`, then that augmenter will be used as the branch.
  - If iterable of `Augmenter`, then that iterable will be converted into a `Sequential` and used as the augmenter.

- **per_channel** (*bool or float, optional*) – Whether to use the same factor for all channels (`False`) or to sample a new value for each channel (`True`). If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as True, otherwise as False.

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Alpha(0.5, iaa.Grayscale(1.0))
```

Convert each image to pure grayscale and alpha-blend the result with the original image using an alpha of `50%`, thereby removing about `50%` of all color. This is equivalent to `iaa.Grayscale(0.5)`.

```
>>> aug = iaa.Alpha((0.0, 1.0), iaa.Grayscale(1.0))
```

Same as in the previous example, but the alpha factor is sampled uniformly from the interval `[0.0, 1.0]` once per image, thereby removing a random fraction of all colors. This is equivalent to `iaa.Grayscale((0.0, 1.0))`.

```
>>> aug = iaa.Alpha(
>>>     (0.0, 1.0),
>>>     iaa.Affine(rotate=(-20, 20)),
>>>     per_channel=0.5)
```

First, rotate each image by a random degree sampled uniformly from the interval `[-20, 20]`. Then, alpha-blend that new image with the original one using a random factor sampled uniformly from the interval `[0.0, 1.0]`. For `50%` of all images, the blending happens channel-wise and the factor is sampled independently per channel (`per_channel=0.5`). As a result, e.g. the red channel may look visibly rotated (factor near `1.0`), while the green and blue channels may not look rotated (factors near `0.0`).

```
>>> aug = iaa.Alpha(
>>>     (0.0, 1.0),
>>>     first=iaa.Add(100),
>>>     second=iaa.Multiply(0.2))
```

Apply two branches of augmenters – `A` and `B` – *independently* to input images and alpha-blend the results of these branches using a factor `f`. Branch `A` increases image pixel intensities by `100` and `B` multiplies the pixel intensities by `0.2`. `f` is sampled uniformly from the interval `[0.0, 1.0]` per image. The resulting images contain a bit of `A` and a bit of `B`.

```
>>> aug = iaa.Alpha([0.25, 0.75], iaa.MedianBlur(13))
```

Apply median blur to each image and alpha-blend the result with the original image using an alpha factor of either exactly `0.25` or exactly `0.75` (sampled once per image).

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.

Continued on next page

Table 79 – continued from previous page

draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_children_lists(*self*)

Get a list of lists of children of this augmenter.

For most augmenters, the result will be a single empty list. For augmenters with children it will often be a list with one sublist containing all children. In some cases the augmenter will contain multiple distinct lists of children, e.g. an if-list and an else-list. This will lead to a result consisting of a single list with multiple sublists, each representing the respective sublist of children.

E.g. for an if/else-augmenter that executes the children `A1`, `A2` if a condition is met and otherwise executes the children `B1`, `B2`, `B3` the result will be `[[A1, A2], [B1, B2, B3]]`.

IMPORTANT: While the topmost list may be newly created, each of the sublist must be editable inplace resulting in a changed children list of the augmenter. E.g. if an Augmenter `IfElse(condition, [A1, A2], [B1, B2, B3])` returns `[[A1, A2], [B1, B2, B3]]` for a call to *imgaug.augmenters.meta.Augmenter.get_children_lists()* and `A2` is removed inplace from `[A1, A2]`, then the children lists of `IfElse(...)` must also change to `[A1]`, `[B1, B2, B3]`. This is used in `imgaug.augmeneters.meta.Augmenter.remove_augmenters_inplace()`.

Returns One or more lists of child augmenter. Can also be a single empty list.

Return type list of list of imgaug.augmenters.meta.Augmenter

get_parameters(*self*)

class imgaug.augmenters.blend.**AlphaElementwise**(*factor=0*, *first=None*, *second=None*, *per_channel=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.blend.Alpha*

Alpha-blend two image sources using alpha/opacity values sampled per pixel.

This is the same as `Alpha`, except that the opacity factor is sampled once per *pixel* instead of once per *image* (or a few times per image, if `Alpha.per_channel` is set to `True`).

See `Alpha` for more details.

Note: It is not recommended to use `AlphaElementwise` with augmenters that change the geometry of images (e.g. horizontal flips, affine transformations) if you *also* want to augment coordinates (e.g. keypoints, polygons, ...), as it is unclear which of the two coordinate results (first or second branch) should be used as the coordinates after augmentation.

Currently, the for keypoints and line strings the results of the first and second branch will be mixed. For each coordinate, the augmented one from the first or second branch will be picked based on the average alpha mask value at the corresponding spatial location.

For polygons, only all polygons of the first or all of the second branch will be used, based on the average over the whole alpha mask.

---

dtype support:

```
See :func:`imgaug.augmenters.blend.blend_alpha`.
```

Parameters

- **factor** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Weighting of the results of the first branch. Values close to 0 mean that the results from the second branch (see parameter *second*) make up most of the final image.
  - If float, then that value will be used for all images.
  - If tuple `(a, b)`, then a random value from the interval `[a, b]` will be sampled per image.
  - If a list, then a random value will be picked from that list per image.
  - If `StochasticParameter`, then that parameter will be used to sample a value per image.
- **first** (*None or imgaug.augmenters.meta.Augmenter or iterable of imgaug.augmenters.meta.Augmenter, optional*) –

  Augmenter(s) that make up the first of the two branches.
  - If `None`, then the input images will be reused as the output of the first branch.
  - If `Augmenter`, then that augmenter will be used as the branch.
  - If iterable of `Augmenter`, then that iterable will be converted into a `Sequential` and used as the augmenter.
- **second** (*None or imgaug.augmenters.meta.Augmenter or iterable of imgaug.augmenters.meta.Augmenter, optional*) –

  Augmenter(s) that make up the second of the two branches.
  - If `None`, then the input images will be reused as the output of the second branch.
  - If `Augmenter`, then that augmenter will be used as the branch.

  - If iterable of `Augmenter`, then that iterable will be converted into a `Sequential` and used as the augmenter.

- **per_channel** (*bool or float, optional*) – Whether to use the same factor for all channels (`False`) or to sample a new value for each channel (`True`). If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as True, otherwise as False.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.AlphaElementwise(0.5, iaa.Grayscale(1.0))
```

Convert each image to pure grayscale and alpha-blend the result with the original image using an alpha of `50%` for all pixels, thereby removing about `50%` of all color. This is equivalent to `iaa.Grayscale(0.5)`. This is also equivalent to `iaa.Alpha(0.5, iaa.Grayscale(1.0))`, as the opacity has a fixed value of `0.5` and is hence identical for all pixels.

```
>>> aug = iaa.AlphaElementwise((0, 1.0), iaa.Grayscale(1.0))
```

Same as in the previous example, but the alpha factor is sampled uniformly from the interval `[0.0, 1.0]` once per pixel, thereby removing a random fraction of all colors from each pixel. This is equivalent to `iaa.Grayscale((0.0, 1.0))`.

```
>>> aug = iaa.AlphaElementwise(
>>>     (0.0, 1.0),
>>>     iaa.Affine(rotate=(-20, 20)),
>>>     per_channel=0.5)
```

First, rotate each image by a random degree sampled uniformly from the interval `[-20, 20]`. Then, alpha-blend that new image with the original one using a random factor sampled uniformly from the interval `[0.0, 1.0]` per pixel. For `50%` of all images, the blending happens channel-wise and the factor is sampled independently per pixel *and* channel (`per_channel=0.5`). As a result, e.g. the red channel may look visibly rotated (factor near `1.0`), while the green and blue channels may not look rotated (factors near `0.0`).

```
>>> aug = iaa.AlphaElementwise(
>>>     (0.0, 1.0),
>>>     first=iaa.Add(100),
>>>     second=iaa.Multiply(0.2))
```

Apply two branches of augmenters – `A` and `B` – *independently* to input images and alpha-blend the results of these branches using a factor `f`. Branch `A` increases image pixel intensities by `100` and `B` multiplies the pixel intensities by `0.2`. `f` is sampled uniformly from the interval `[0.0, 1.0]` per pixel. The resulting images contain a bit of `A` and a bit of `B`.

```
>>> aug = iaa.AlphaElementwise([0.25, 0.75], iaa.MedianBlur(13))
```

Apply median blur to each image and alpha-blend the result with the original image using an alpha factor of either exactly `0.25` or exactly `0.75` (sampled once per pixel).

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
`remove_augmenters_inplace`(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
`reseed`(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
`show_grid`(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.

Continued on next page

Table 80 – continued from previous page

to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.blend.**FrequencyNoiseAlpha**(*exponent=(-4, 4), first=None, second=None, per_channel=False, size_px_max=(4, 16), upscale_method=None, iterations=(1, 3), aggregation_method=['avg', 'max'], sigmoid=0.5, sigmoid_thresh=None, name=None, deterministic=False, random_state=None*)

Bases: *imgaug.augmenters.blend.AlphaElementwise*

Alpha-blend two image sources using frequency noise masks.

The alpha masks are sampled using frequency noise of varying scales, which can sometimes create large connected blobs of `1` s surrounded by `0` s and other times results in smaller patterns. If nearest neighbour upsampling is used, these blobs can be rectangular with sharp edges.

dtype support:

```
See ``imgaug.augmenters.blend.AlphaElementwise``.
```

Parameters

- **exponent** (*number or tuple of number of list of number or imgaug.parameters.StochasticParameter, optional*) – Exponent to use when scaling in the frequency domain. Sane values are in the range `-4` (large blobs) to `4` (small patterns). To generate cloud-like structures, use roughly `-2`.
  - If number, then that number will be used as the exponent for all iterations.
  - If tuple of two numbers `(a, b)`, then a value will be sampled per iteration from the interval `[a, b]`.
  - If a list of numbers, then a value will be picked per iteration at random from that list.
  - If a `StochasticParameter`, then a value will be sampled from that parameter per iteration.
- **first** (*None or imgaug.augmenters.meta.Augmenter or iterable of imgaug.augmenters.meta.Augmenter, optional*) –

  Augmenter(s) that make up the first of the two branches.
  - If `None`, then the input images will be reused as the output of the first branch.
  - If `Augmenter`, then that augmenter will be used as the branch.
  - If iterable of `Augmenter`, then that iterable will be converted into a `Sequential` and used as the augmenter.
- **second** (*None or imgaug.augmenters.meta.Augmenter or iterable of imgaug.augmenters.meta.Augmenter, optional*) –

  Augmenter(s) that make up the second of the two branches.

  - If None, then the input images will be reused as the output of the second branch.
  - If Augmenter, then that augmenter will be used as the branch.
  - If iterable of Augmenter, then that iterable will be converted into a Sequential and used as the augmenter.
- **per_channel** (*bool or float, optional*) – Whether to use the same factor for all channels (False) or to sample a new value for each channel (True). If this value is a float p, then for p percent of all images *per_channel* will be treated as True, otherwise as False.
- **size_px_max** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) – The noise is generated in a low resolution environment. This parameter defines the maximum size of that environment (in pixels). The environment is initialized at the same size as the input image and then downscaled, so that no side exceeds *size_px_max* (aspect ratio is kept).
  - If int, then that number will be used as the size for all iterations.
  - If tuple of two int s (a, b), then a value will be sampled per iteration from the discrete interval [a..b].
  - If a list of int s, then a value will be picked per iteration at random from that list.
  - If a StochasticParameter, then a value will be sampled from that parameter per iteration.
- **upscale_method** (*None or imgaug.ALL or str or list of str or imgaug.parameters.StochasticParameter, optional*) – After generating the noise maps in low resolution environments, they have to be upscaled to the input image size. This parameter controls the upscaling method.
  - If None, then either nearest or linear or cubic is picked. Most weight is put on linear, followed by cubic.
  - If imgaug.ALL, then either nearest or linear or area or cubic is picked per iteration (all same probability).
  - If string, then that value will be used as the method (must be nearest or linear or area or cubic).
  - If list of string, then a random value will be picked from that list per iteration.
  - If StochasticParameter, then a random value will be sampled from that parameter per iteration.
- **iterations** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) – How often to repeat the simplex noise generation process per image.
  - If int, then that number will be used as the iterations for all images.
  - If tuple of two int s (a, b), then a value will be sampled per image from the discrete interval [a..b].
  - If a list of int s, then a value will be picked per image at random from that list.
  - If a StochasticParameter, then a value will be sampled from that parameter per image.
- **aggregation_method** (*imgaug.ALL or str or list of str or imgaug.parameters.StochasticParameter, optional*) – The noise maps (from each iteration) are combined to one noise map using an aggregation process. This parameter defines the method used for that process. Valid methods are min, max or avg, where 'min' combines

the noise maps by taking the (elementwise) minimum over all iteration's results, `max` the (elementwise) maximum and `avg` the (elementwise) average.

  - If `imgaug.ALL`, then a random value will be picked per image from the valid ones.
  - If a string, then that value will always be used as the method.
  - If a list of string, then a random value will be picked from that list per image.
  - If a `StochasticParameter`, then a random value will be sampled from that parameter per image.

- **sigmoid** (*bool or number, optional*) – Whether to apply a sigmoid function to the final noise maps, resulting in maps that have more extreme values (close to `0.0` or `1.0`).
  - If `bool`, then a sigmoid will always (`True`) or never (`False`) be applied.
  - If a number `p` with `0<=p<=1`, then a sigmoid will be applied to `p` percent of all final noise maps.
- **sigmoid_thresh** (*None or number or tuple of number or imgaug.parameters.StochasticParameter, optional*) – Threshold of the sigmoid, when applied. Thresholds above zero (e.g. `5.0`) will move the saddle point towards the right, leading to more values close to `0.0`.
  - If `None`, then `Normal(0, 5.0)` will be used.
  - If number, then that threshold will be used for all images.
  - If tuple of two numbers `(a, b)`, then a random value will be sampled per image from the range `[a, b]`.
  - If `StochasticParameter`, then a random value will be sampled from that parameter per image.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.FrequencyNoiseAlpha(first=iaa.EdgeDetect(1.0))
```

Detect per image all edges, mark them in a black and white image and then alpha-blend the result with the original image using frequency noise masks.

```
>>> aug = iaa.FrequencyNoiseAlpha(
>>>     first=iaa.EdgeDetect(1.0),
>>>     upscale_method="nearest")
```

Same as the first example, but using only linear upscaling to scale the frequency noise masks to the final image sizes, i.e. no nearest neighbour upsampling is used. This results in smooth edges.

```
>>> aug = iaa.FrequencyNoiseAlpha(
>>>     first=iaa.EdgeDetect(1.0),
>>>     upscale_method="linear")
```

Same as the first example, but using only linear upscaling to scale the frequency noise masks to the final image sizes, i.e. no nearest neighbour upsampling is used. This results in smooth edges.

```
>>> aug = iaa.FrequencyNoiseAlpha(
>>>     first=iaa.EdgeDetect(1.0),
>>>     upscale_method="linear",
>>>     exponent=-2,
>>>     sigmoid=False)
```

Same as in the previous example, but with the exponent set to a constant `-2` and the sigmoid deactivated, resulting in cloud-like patterns without sharp edges.

```
>>> aug = iaa.FrequencyNoiseAlpha(
>>>     first=iaa.EdgeDetect(1.0),
>>>     sigmoid_thresh=iap.Normal(10.0, 5.0))
```

Same as the first example, but using a threshold for the sigmoid function that is further to the right. This is more conservative, i.e. the generated noise masks will be mostly black (values around `0.0`), which means that most of the original images (parameter/branch *second*) will be kept, rather than using the results of the augmentation (parameter/branch *first*).

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.

Continued on next page

Table 81 – continued from previous page

draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.blend.**SimplexNoiseAlpha**(*first=None*, *second=None*, *per_channel=False*, *size_px_max=(2, 16)*, *upscale_method=None*, *iterations=(1, 3)*, *aggregation_method='max'*, *sigmoid=True*, *sigmoid_thresh=None*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.blend.AlphaElementwise*

Alpha-blend two image sources using simplex noise alpha masks.

The alpha masks are sampled using a simplex noise method, roughly creating connected blobs of 1s surrounded by 0s. If nearest neighbour upsampling is used, these blobs can be rectangular with sharp edges.

dtype support:

```
See ``imgaug.augmenters.blend.AlphaElementwise``.
```

Parameters

- **first** (*None or imgaug.augmenters.meta.Augmenter or iterable of imgaug.augmenters.meta.Augmenter, optional*) –

  Augmenter(s) that make up the first of the two branches.

  – If None, then the input images will be reused as the output of the first branch.

  – If Augmenter, then that augmenter will be used as the branch.

  - If iterable of `Augmenter`, then that iterable will be converted into a Sequential and used as the augmenter.

- **second** (*None or imgaug.augmenters.meta.Augmenter or iterable of imgaug.augmenters.meta.Augmenter, optional*) –

  Augmenter(s) that make up the second of the two branches.

  - If `None`, then the input images will be reused as the output of the second branch.
  - If `Augmenter`, then that augmenter will be used as the branch.
  - If iterable of `Augmenter`, then that iterable will be converted into a Sequential and used as the augmenter.

- **per_channel** (*bool or float, optional*) – Whether to use the same factor for all channels (`False`) or to sample a new value for each channel (`True`). If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`, otherwise as `False`.

- **size_px_max** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) – The simplex noise is always generated in a low resolution environment. This parameter defines the maximum size of that environment (in pixels). The environment is initialized at the same size as the input image and then downscaled, so that no side exceeds *size_px_max* (aspect ratio is kept).

  - If int, then that number will be used as the size for all iterations.
  - If tuple of two `int` s `(a, b)`, then a value will be sampled per iteration from the discrete interval `[a..b]`.
  - If a list of `int` s, then a value will be picked per iteration at random from that list.
  - If a `StochasticParameter`, then a value will be sampled from that parameter per iteration.

- **upscale_method** (*None or imgaug.ALL or str or list of str or imgaug.parameters.StochasticParameter, optional*) – After generating the noise maps in low resolution environments, they have to be upscaled to the input image size. This parameter controls the upscaling method.

  - If `None`, then either `nearest` or `linear` or `cubic` is picked. Most weight is put on `linear`, followed by `cubic`.
  - If `imgaug.ALL`, then either `nearest` or `linear` or `area` or `cubic` is picked per iteration (all same probability).
  - If a string, then that value will be used as the method (must be `nearest` or `linear` or `area` or `cubic`).
  - If list of string, then a random value will be picked from that list per iteration.
  - If `StochasticParameter`, then a random value will be sampled from that parameter per iteration.

- **iterations** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) –

  How often to repeat the simplex noise generation process per image.

  - If `int`, then that number will be used as the iterations for all images.
  - If tuple of two `int` s `(a, b)`, then a value will be sampled per image from the discrete interval `[a..b]`.
  - If a list of `int` s, then a value will be picked per image at random from that list.

  - If a `StochasticParameter`, then a value will be sampled from that parameter per image.

- **aggregation_method** (*imgaug.ALL or str or list of str or imgaug.parameters.StochasticParameter, optional*) – The noise maps (from each iteration) are combined to one noise map using an aggregation process. This parameter defines the method used for that process. Valid methods are `min`, `max` or `avg`, where `min` combines the noise maps by taking the (elementwise) minimum over all iteration's results, `max` the (elementwise) maximum and `avg` the (elementwise) average.
  - If `imgaug.ALL`, then a random value will be picked per image from the valid ones.
  - If a string, then that value will always be used as the method.
  - If a list of string, then a random value will be picked from that list per image.
  - If a `StochasticParameter`, then a random value will be sampled from that paramter per image.
- **sigmoid** (*bool or number, optional*) – Whether to apply a sigmoid function to the final noise maps, resulting in maps that have more extreme values (close to 0.0 or 1.0).
  - If `bool`, then a sigmoid will always (`True`) or never (`False`) be applied.
  - If a number `p` with `0<=p<=1`, then a sigmoid will be applied to `p` percent of all final noise maps.
- **sigmoid_thresh** (*None or number or tuple of number or imgaug.parameters.StochasticParameter, optional*) – Threshold of the sigmoid, when applied. Thresholds above zero (e.g. `5.0`) will move the saddle point towards the right, leading to more values close to 0.0.
  - If `None`, then `Normal(0, 5.0)` will be used.
  - If number, then that threshold will be used for all images.
  - If tuple of two numbers `(a, b)`, then a random value will be sampled per image from the interval `[a, b]`.
  - If `StochasticParameter`, then a random value will be sampled from that parameter per image.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.SimplexNoiseAlpha(iaa.EdgeDetect(1.0))
```

Detect per image all edges, mark them in a black and white image and then alpha-blend the result with the original image using simplex noise masks.

```
>>> aug = iaa.SimplexNoiseAlpha(
>>>     iaa.EdgeDetect(1.0),
>>>     upscale_method="nearest")
```

Same as in the previous example, but using only nearest neighbour upscaling to scale the simplex noise masks to the final image sizes, i.e. no nearest linear upsampling is used. This leads to rectangles with sharp edges.

```
>>> aug = iaa.SimplexNoiseAlpha(
>>>     iaa.EdgeDetect(1.0),
>>>     upscale_method="linear")
```

Same as in the previous example, but using only linear upscaling to scale the simplex noise masks to the final image sizes, i.e. no nearest neighbour upsampling is used. This leads to rectangles with smooth edges.

```
>>> aug = iaa.SimplexNoiseAlpha(
>>>     iaa.EdgeDetect(1.0),
>>>     sigmoid_thresh=iap.Normal(10.0, 5.0))
```

Same as in the first example, but using a threshold for the sigmoid function that is further to the right. This is more conservative, i.e. the generated noise masks will be mostly black (values around `0.0`), which means that most of the original images (parameter/branch *second*) will be kept, rather than using the results of the augmentation (parameter/branch *first*).

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.

Continued on next page

Table 82 – continued from previous page

find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

imgaug.augmenters.blend.**blend_alpha**(*image_fg*, *image_bg*, *alpha*, *eps=0.01*)

Blend two images using an alpha blending.

In alpha blending, the two images are naively mixed using a multiplier. Let `A` be the foreground image and `B` the background image and `a` is the alpha value. Each pixel intensity is then computed as `a * A_ij + (1-a) * B_ij`.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; fully tested
* ``uint32``: yes; fully tested
* ``uint64``: yes; fully tested (1)
* ``int8``: yes; fully tested
* ``int16``: yes; fully tested
* ``int32``: yes; fully tested
* ``int64``: yes; fully tested (1)
* ``float16``: yes; fully tested
* ``float32``: yes; fully tested
* ``float64``: yes; fully tested (1)
* ``float128``: no (2)
* ``bool``: yes; fully tested (2)

- (1) Tests show that these dtypes work, but a conversion to
      ``float128`` happens, which only has 96 bits of size instead of
      true 128 bits and hence not twice as much resolution. It is
      possible that these dtypes result in inaccuracies, though the
      tests did not indicate that.
- (2) Not available due to the input dtype having to be increased to
```

(continues on next page)

(continued from previous page)

```
      an equivalent float dtype with two times the input resolution.
- (3) Mapped internally to ``float16``.
```

Parameters

- **image_fg** (*(H,W,[C]) ndarray*) – Foreground image. Shape and dtype kind must match the one of the background image.
- **image_bg** (*(H,W,[C]) ndarray*) – Background image. Shape and dtype kind must match the one of the foreground image.
- **alpha** (*number or iterable of number or ndarray*) – The blending factor, between `0.0` and `1.0`. Can be interpreted as the opacity of the foreground image. Values around `1.0` result in only the foreground image being visible. Values around `0.0` result in only the background image being visible. Multiple alphas may be provided. In these cases, there must be exactly one alpha per channel in the foreground/background image. Alternatively, for `(H,W,C)` images, either one `(H,W)` array or an `(H,W,C)` array of alphas may be provided, denoting the elementwise alpha value.
- **eps** (*number, optional*) – Controls when an alpha is to be interpreted as exactly `1.0` or exactly `0.0`, resulting in only the foreground/background being visible and skipping the actual computation.

Returns **image_blend** – Blend of foreground and background image.

Return type (H,W,C) ndarray

13.19 imgaug.augmenters.blur

Augmenters that blur images.

Do not import directly from this file, as the categorization is not final. Use instead

```
from imgaug import augmenters as iaa
```

and then e.g.

```
seq = iaa.Sequential([
    iaa.GaussianBlur((0.0, 3.0)),
    iaa.AverageBlur((2, 5))
])
```

List of augmenters:

- GaussianBlur
- AverageBlur
- MedianBlur
- BilateralBlur
- MotionBlur

class imgaug.augmenters.blur.**AverageBlur**(*k=1*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Blur an image by computing simple means over neighbourhoods.

The padding behaviour around the image borders is cv2's `BORDER_REFLECT_101`.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; tested
* ``uint32``: no (1)
* ``uint64``: no (2)
* ``int8``: yes; tested (3)
* ``int16``: yes; tested
* ``int32``: no (4)
* ``int64``: no (5)
* ``float16``: yes; tested (6)
* ``float32``: yes; tested
* ``float64``: yes; tested
* ``float128``: no
* ``bool``: yes; tested (7)

- (1) rejected by ``cv2.blur()``
- (2) loss of resolution in ``cv2.blur()`` (result is ``int32``)
- (3) ``int8`` is mapped internally to ``int16``, ``int8`` itself
      leads to cv2 error "Unsupported combination of source format
      (=1), and buffer format (=4) in function 'getRowSumFilter'" in
      ``cv2``
- (4) results too inaccurate
- (5) loss of resolution in ``cv2.blur()`` (result is ``int32``)
- (6) ``float16`` is mapped internally to ``float32``
- (7) ``bool`` is mapped internally to ``float32``
```

Parameters

- **k** (*int or tuple of int or tuple of tuple of int or imgaug.parameters.StochasticParameter or tuple of StochasticParameter, optional*) –

  Kernel size to use.

  - If a single `int`, then that value will be used for the height and width of the kernel.
  - If a tuple of two `int` s `(a, b)`, then the kernel size will be sampled from the interval `[a..b]`.
  - If a tuple of two tuples of `int` s `((a, b), (c, d))`, then per image a random kernel height will be sampled from the interval `[a..b]` and a random kernel width will be sampled from the interval `[c..d]`.
  - If a `StochasticParameter`, then `N` samples will be drawn from that parameter per `N` input images, each representing the kernel size for the n-th image.
  - If a tuple `(a, b)`, where either `a` or `b` is a tuple, then `a` and `b` will be treated according to the rules above. This leads to different values for height and width of the kernel.

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.AverageBlur(k=5)
```

Blur all images using a kernel size of `5x5`.

```
>>> aug = iaa.AverageBlur(k=(2, 5))
```

Blur images using a varying kernel size, which is sampled (per image) uniformly from the interval `[2..5]`.

```
>>> aug = iaa.AverageBlur(k=((5, 7), (1, 3)))
```

Blur images using a varying kernel size, which's height is sampled (per image) uniformly from the interval `[5..7]` and which's width is sampled (per image) uniformly from `[1..3]`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.

Continued on next page

Table 83 – continued from previous page

localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.blur.**BilateralBlur**(*d=1*, *sigma_color=(10, 250)*, *sigma_space=(10, 250)*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Blur/Denoise an image using a bilateral filter.

Bilateral filters blur homogenous and textured areas, while trying to preserve edges.

See http://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html#bilateralfilter for more information regarding the parameters.

dtype support:

```
* ``uint8``: yes; not tested
* ``uint16``: ?
* ``uint32``: ?
* ``uint64``: ?
* ``int8``: ?
* ``int16``: ?
* ``int32``: ?
* ``int64``: ?
* ``float16``: ?
* ``float32``: ?
* ``float64``: ?
* ``float128``: ?
* ``bool``: ?
```

Parameters

- **d** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) – Diameter of each pixel neighborhood with value range `[1 .. inf)`. High values for *d* lead to significantly worse performance. Values equal or less than `10` seem to be good. Use `<5` for real-time applications.
  - If a single `int`, then that value will be used for the diameter.

  - If a tuple of two `int`s `(a, b)`, then the diameter will be a value sampled from the interval `[a..b]`.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, then `N` samples will be drawn from that parameter per `N` input images, each representing the diameter for the n-th image. Expected to be discrete.

- **sigma_color** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Filter sigma in the color space with value range `[1, inf)`. A large value of the parameter means that farther colors within the pixel neighborhood (see *sigma_space*) will be mixed together, resulting in larger areas of semi-equal color.
  - If a single `int`, then that value will be used for the diameter.
  - If a tuple of two `int`s `(a, b)`, then the diameter will be a value sampled from the interval `[a, b]`.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, then `N` samples will be drawn from that parameter per `N` input images, each representing the diameter for the n-th image. Expected to be discrete.
- **sigma_space** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Filter sigma in the coordinate space with value range `[1, inf)`. A large value of the parameter means that farther pixels will influence each other as long as their colors are close enough (see *sigma_color*).
  - If a single `int`, then that value will be used for the diameter.
  - If a tuple of two `int`s `(a, b)`, then the diameter will be a value sampled from the interval `[a, b]`.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, then `N` samples will be drawn from that parameter per `N` input images, each representing the diameter for the n-th image. Expected to be discrete.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.BilateralBlur(
>>>     d=(3, 10), sigma_color=(10, 250), sigma_space=(10, 250))
```

Blur all images using a bilateral filter with a *max distance* sampled uniformly from the interval `[3, 10]` and wide ranges for *sigma_color* and *sigma_space*.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
`remove_augmenters_inplace`(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
`reseed`(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
`show_grid`(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
`to_deterministic`(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.blur.**GaussianBlur**(*sigma=0*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Augmenter to blur images using gaussian kernels.

dtype support:

```
See :func:`imgaug.augmenters.blur.blur_gaussian_(backend="auto")`.
```

Parameters

- **sigma** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Standard deviation of the gaussian kernel. Values in the range `0.0` (no blur) to `3.0` (strong blur) are common.
  - If a single `float`, that value will always be used as the standard deviation.
  - If a tuple `(a, b)`, then a random value from the interval `[a, b]` will be picked per image.
  - If a list, then a random value will be sampled per image from that list.
  - If a `StochasticParameter`, then `N` samples will be drawn from that parameter per `N` input images.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.GaussianBlur(sigma=1.5)
```

Blur all images using a gaussian kernel with a standard deviation of `1.5`.

```
>>> aug = iaa.GaussianBlur(sigma=(0.0, 3.0))
```

Blur images using a gaussian kernel with a random standard deviation sampled uniformly (per image) from the interval `[0.0, 3.0]`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.

Continued on next page

Table 85 – continued from previous page

augment_batch(self, batch[, hooks])	Augment a single batch.
augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ...[, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.blur.**MedianBlur**(*k=1*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Blur an image by computing median values over neighbourhoods.

Median blurring can be used to remove small dirt from images. At larger kernel sizes, its effects have some similarity with Superpixels.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: ?
* ``uint32``: ?
* ``uint64``: ?
* ``int8``: ?
* ``int16``: ?
* ``int32``: ?
* ``int64``: ?
* ``float16``: ?
* ``float32``: ?
* ``float64``: ?
* ``float128``: ?
* ``bool``: ?
```

Parameters

- **k** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) –

  Kernel size.

  - If a single `int`, then that value will be used for the height and width of the kernel. Must be an odd value.
  - If a tuple of two ints `(a, b)`, then the kernel size will be an odd value sampled from the interval `[a..b]`. `a` and `b` must both be odd values.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, then `N` samples will be drawn from that parameter per `N` input images, each representing the kernel size for the nth image. Expected to be discrete. If a sampled value is not odd, then that value will be increased by `1`.

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.MedianBlur(k=5)
```

Blur all images using a kernel size of `5x5`.

```
>>> aug = iaa.MedianBlur(k=(3, 7))
```

Blur images using varying kernel sizes, which are sampled uniformly from the interval `[3..7]`. Only odd values will be sampled, i.e. `3` or `5` or `7`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
`remove_augmenters_inplace`(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
`reseed`(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
`show_grid`(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.

Continued on next page

Table 86 – continued from previous page

to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.blur.**MotionBlur**(*k=5*, *angle=(0, 360)*, *direction=(-1.0, 1.0)*, *order=1*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.convolutional.Convolve*

Blur images in a way that fakes camera or object movements.

dtype support:

```
See ``imgaug.augmenters.convolutional.Convolve``.
```

Parameters

- **k** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) –

  Kernel size to use.

  - If a single `int`, then that value will be used for the height and width of the kernel.
  - If a tuple of two `int` s `(a, b)`, then the kernel size will be sampled from the interval `[a..b]`.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, then `N` samples will be drawn from that parameter per `N` input images, each representing the kernel size for the n-th image.

- **angle** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Angle of the motion blur in degrees (clockwise, relative to top center direction).

  - If a number, exactly that value will be used.
  - If a tuple `(a, b)`, a random value from the interval `[a, b]` will be uniformly sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, a value will be sampled from the parameter per image.

- **direction** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Forward/backward direction of the motion blur. Lower values towards `-1.0` will point the motion blur towards the back (with angle provided via *angle*). Higher values towards `1.0` will point the motion blur forward. A value of `0.0` leads to a uniformly (but still angled) motion blur.

  - If a number, exactly that value will be used.
  - If a tuple `(a, b)`, a random value from the interval `[a, b]` will be uniformly sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, a value will be sampled from the parameter per image.

- **order** (*int or iterable of int or imgaug.ALL or imgaug.parameters.StochasticParameter, optional*) – Interpolation order to use when rotating the kernel according to *angle*. See `imgaug.augmenters.geometric.Affine.__init__()`. Recommended to be `0` or `1`, with `0` being faster, but less continuous/smooth as *angle* is changed, particularly around multiple of `45` degrees.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.MotionBlur(k=15)
```

Apply motion blur with a kernel size of `15x15` pixels to images.

```
>>> aug = iaa.MotionBlur(k=15, angle=[-45, 45])
```

Apply motion blur with a kernel size of `15x15` pixels and a blur angle of either `-45` or `45` degrees (randomly picked per image).

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.

Continued on next page

Table 87 – continued from previous page

copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

imgaug.augmenters.blur.**blur_gaussian_**(*image*, *sigma*, *ksize=None*, *backend='auto'*, *eps=0.001*)

Blur an image using gaussian blurring.

This operation might change the input image in-place.

dtype support:

```
if (backend="auto")::

    * ``uint8``: yes; fully tested (1)
    * ``uint16``: yes; tested (1)
    * ``uint32``: yes; tested (2)
    * ``uint64``: yes; tested (2)
    * ``int8``: yes; tested (1)
    * ``int16``: yes; tested (1)
    * ``int32``: yes; tested (1)
    * ``int64``: yes; tested (2)
    * ``float16``: yes; tested (1)
    * ``float32``: yes; tested (1)
    * ``float64``: yes; tested (1)
    * ``float128``: no
    * ``bool``: yes; tested (1)
```

(continues on next page)

(continued from previous page)

```
    - (1) Handled by ``cv2``. See ``backend="cv2"``.
    - (2) Handled by ``scipy``. See ``backend="scipy"``.

if (backend="cv2")::

    * ``uint8``: yes; fully tested
    * ``uint16``: yes; tested
    * ``uint32``: no (2)
    * ``uint64``: no (3)
    * ``int8``: yes; tested (4)
    * ``int16``: yes; tested
    * ``int32``: yes; tested (5)
    * ``int64``: no (6)
    * ``float16``: yes; tested (7)
    * ``float32``: yes; tested
    * ``float64``: yes; tested
    * ``float128``: no (8)
    * ``bool``: yes; tested (1)

    - (1) Mapped internally to ``float32``. Otherwise causes
          ``TypeError: src data type = 0 is not supported``.
    - (2) Causes ``TypeError: src data type = 6 is not supported``.
    - (3) Causes ``cv2.error: OpenCV(3.4.5) (...)/filter.cpp:2957:
          error: (-213:The function/feature is not implemented)
          Unsupported combination of source format (=4), and buffer
          format (=5) in function 'getLinearRowFilter'``.
    - (4) Mapped internally to ``int16``. Otherwise causes
          ``cv2.error: OpenCV(3.4.5) (...)/filter.cpp:2957: error:
          (-213:The function/feature is not implemented) Unsupported
          combination of source format (=1), and buffer format (=5)
          in function 'getLinearRowFilter'``.
    - (5) Mapped internally to ``float64``. Otherwise causes
          ``cv2.error: OpenCV(3.4.5) (...)/filter.cpp:2957: error:
          (-213:The function/feature is not implemented) Unsupported
          combination of source format (=4), and buffer format (=5)
          in function 'getLinearRowFilter'``.
    - (6) Causes ``cv2.error: OpenCV(3.4.5) (...)/filter.cpp:2957:
          error: (-213:The function/feature is not implemented)
          Unsupported combination of source format (=4), and buffer
          format (=5) in function 'getLinearRowFilter'``.
    - (7) Mapped internally to ``float32``. Otherwise causes
          ``TypeError: src data type = 23 is not supported``.
    - (8) Causes ``TypeError: src data type = 13 is not supported``.

if (backend="scipy")::

    * ``uint8``: yes; fully tested
    * ``uint16``: yes; tested
    * ``uint32``: yes; tested
    * ``uint64``: yes; tested
    * ``int8``: yes; tested
    * ``int16``: yes; tested
    * ``int32``: yes; tested
    * ``int64``: yes; tested
    * ``float16``: yes; tested (1)
    * ``float32``: yes; tested
    * ``float64``: yes; tested
```

(continues on next page)

(continued from previous page)

```
    * ``float128``: no (2)
    * ``bool``: yes; tested (3)

    - (1) Mapped internally to ``float32``. Otherwise causes
          ``RuntimeError: array type dtype('float16') not supported``.
    - (2) Causes ``RuntimeError: array type dtype('float128') not
          supported``.
    - (3) Mapped internally to ``float32``. Otherwise too inaccurate.
```

Parameters

- **image** (*numpy.ndarray*) – The image to blur. Expected to be of shape `(H, W)` or `(H, W, C)`.
- **sigma** (*number*) – Standard deviation of the gaussian blur. Larger numbers result in more large-scale blurring, which is overall slower than small-scale blurring.
- **ksize** (*None or int, optional*) – Size in height/width of the gaussian kernel. This argument is only understood by the `cv2` backend. If it is set to `None`, an appropriate value for *ksize* will automatically be derived from *sigma*. The value is chosen tighter for larger sigmas to avoid as much as possible very large kernel sizes and therey improve performance.
- **backend** (*{'auto', 'cv2', 'scipy'}, optional*) – Backend library to use. If `auto`, then the likely best library will be automatically picked per image. That is usually equivalent to `cv2` (OpenCV) and it will fall back to `scipy` for datatypes not supported by OpenCV.
- **eps** (*number, optional*) – A threshold used to decide whether *sigma* can be considered zero.

Returns **image** – The blurred image. Same shape and dtype as the input.

Return type numpy.ndarray

13.20 imgaug.augmenters.color

Augmenters that affect image colors or image colorspaces.

Do not import directly from this file, as the categorization is not final. Use instead

```
from imgaug import augmenters as iaa
```

and then e.g.

```
seq = iaa.Sequential([
    iaa.Grayscale((0.0, 1.0)),
    iaa.AddToHueAndSaturation((-10, 10))
])
```

List of augmenters:

- InColorspace (deprecated)
- WithColorspace
- WithHueAndSaturation
- MultiplyHueAndSaturation
- MultiplyHue

- MultiplySaturation
- AddToHueAndSaturation
- AddToHue
- AddToSaturation
- ChangeColorspace
- Grayscale
- KMeansColorQuantization
- UniformColorQuantization

class imgaug.augmenters.color.**AddToHue**(*value=(-255, 255), from_colorspace='RGB', name=None, deterministic=False, random_state=None*)

Bases: *imgaug.augmenters.color.AddToHueAndSaturation*

Add random values to the hue of images.

The augmenter first transforms images to HSV colorspace, then adds random values to the H channel and afterwards converts back to RGB.

If you want to change both the hue and the saturation, it is recommended to use `AddToHueAndSaturation` as otherwise the image will be converted twice to HSV and back to RGB.

This augmenter is a shortcut for `AddToHueAndSaturation(value_hue=...)`.

dtype support:

```
See `imgaug.augmenters.color.AddToHueAndSaturation`.
```

Parameters

- **value** (*None or int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) – Value to add to the hue of all pixels. This is expected to be in the range `-255` to `+255` and will automatically be projected to an angular representation using `(hue/255) * (360/2)` (OpenCV's hue representation is in the range `[0, 180]` instead of `[0, 360]`).
  - If an integer, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value from the discrete range `[a, b]` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a StochasticParameter, then a value will be sampled from that parameter per image.
- **from_colorspace** (*str, optional*) – See *imgaug.augmenters.color.change_colorspace_()*.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.AddToHue((-50, 50))
```

Sample random values from the discrete uniform range `[-50..50]`, convert them to angular representation and add them to the hue, i.e. to the `H` channel in `HSV` colorspace.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.

Continued on next page

Table 88 – continued from previous page

remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.color.**AddToHueAndSaturation**(*value=None*, *value_hue=None*, *value_saturation=None*, *per_channel=False*, *from_colorspace='RGB'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Increases or decreases hue and saturation by random values.

The augmenter first transforms images to HSV colorspace, then adds random values to the H and S channels and afterwards converts back to RGB.

This augmenter is faster than using WithHueAndSaturation in combination with Add.

TODO add float support

dtype support:

```
See :func:`imgaug.augmenters.color.change_colorspace_`.
```

Parameters

- **value** (*None or int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) – Value to add to the hue *and* saturation of all pixels. It is expected to be in the range `-255` to `+255`.
  - If this is `None`, *value_hue* and/or *value_saturation* may be set to values other than `None`.
  - If an integer, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value from the discrete range `[a, b]` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a StochasticParameter, then a value will be sampled from that parameter per image.
- **value_hue** (*None or int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) – Value to add to the hue of all pixels. This is expected to be in the range `-255` to `+255` and will automatically be projected to an angular representation using `(hue/255) * (360/2)` (OpenCV's hue representation is in the range `[0, 180]` instead of `[0, 360]`). Only this or *value* may be set, not both.
  - If this and *value_saturation* are both `None`, *value* may be set to a non-`None` value.
  - If an integer, then that value will be used for all images.

  - If a tuple `(a, b)`, then a value from the discrete range `[a, b]` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a StochasticParameter, then a value will be sampled from that parameter per image.

- **value_saturation** (*None or int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) – Value to add to the saturation of all pixels. It is expected to be in the range `-255` to `+255`. Only this or *value* may be set, not both.
  - If this and *value_hue* are both `None`, *value* may be set to a non-`None` value.
  - If an integer, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value from the discrete range `[a, b]` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a StochasticParameter, then a value will be sampled from that parameter per image.
- **per_channel** (*bool or float, optional*) – Whether to sample per image only one value from *value* and use it for both hue and saturation (`False`) or to sample independently one value for hue and one for saturation (`True`). If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`, otherwise as `False`.

  This parameter has no effect is *value_hue* and/or *value_saturation* are used instead of *value*.
- **from_colorspace** (*str, optional*) – See *imgaug.augmenters.color.change_colorspace_()*.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.AddToHueAndSaturation((-50, 50), per_channel=True)
```

Add random values between `-50` and `50` to the hue and saturation (independently per channel and the same value for all pixels within that channel).

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.

Continued on next page

Table 89 – continued from previous page

augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ..., [, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ..., [, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.color.**AddToSaturation**(*value=(-75, 75), from_colorspace='RGB', name=None, deterministic=False, random_state=None*)

Bases: *imgaug.augmenters.color.AddToHueAndSaturation*

Add random values to the saturation of images.

The augmenter first transforms images to HSV colorspace, then adds random values to the S channel and afterwards converts back to RGB.

If you want to change both the hue and the saturation, it is recommended to use `AddToHueAndSaturation` as otherwise the image will be converted twice to HSV and back to RGB.

This augmenter is a shortcut for `AddToHueAndSaturation(value_saturation=...)`.

dtype support:

```
See `imgaug.augmenters.color.AddToHueAndSaturation`.
```

Parameters

- **value** (*None or int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) – Value to add to the saturation of all pixels. It is expected to be in the range `-255` to `+255`.
  - If an integer, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value from the discrete range `[a, b]` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a StochasticParameter, then a value will be sampled from that parameter per image.
- **from_colorspace** (*str, optional*) – See *imgaug.augmenters.color.change_colorspace_()*.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.AddToSaturation((-50, 50))
```

Sample random values from the discrete uniform range `[-50..50]`, and add them to the saturation, i.e. to the `S` channel in `HSV` colorspace.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.

Continued on next page

Table 90 – continued from previous page

augment(self[, return_batch, hooks])	Augment a batch.
augment_batch(self, batch[, hooks])	Augment a single batch.
augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ...[, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters	

class imgaug.augmenters.color.**ChangeColorspace**(*to_colorspace*, *from_colorspace='RGB'*, *alpha=1.0*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Augmenter to change the colorspace of images.

Note: This augmenter is not tested. Some colorspaces might work, others might not.

..note

```
This augmenter tries to project the colorspace value range on
0-255. It outputs dtype=uint8 images.
```

dtype support:

```
See :func:`imgaug.augmenters.color.change_colorspace_`.
```

Parameters

- **to_colorspace** (*str or list of str or imgaug.parameters.StochasticParameter*) – The target colorspace. Allowed strings are: `RGB`, `BGR`, `GRAY`, `CIE`, `YCrCb`, `HSV`, `HLS`, `Lab`, `Luv`. These are also accessible via `imgaug.augmenters.color.CSPACE_<NAME>`, e.g. `imgaug.augmenters.CSPACE_YCrCb`.
  - If a string, it must be among the allowed colorspaces.
  - If a list, it is expected to be a list of strings, each one being an allowed colorspace. A random element from the list will be chosen per image.
  - If a StochasticParameter, it is expected to return string. A new sample will be drawn per image.
- **from_colorspace** (*str, optional*) – The source colorspace (of the input images). See *to_colorspace*. Only a single string is allowed.
- **alpha** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – The alpha value of the new colorspace when overlayed over the old one. A value close to 1.0 means that mostly the new colorspace is visible. A value close to 0.0 means, that mostly the old image is visible.
  - If an int or float, exactly that value will be used.
  - If a tuple `(a, b)`, a random value from the range `a <= x <= b` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a StochasticParameter, a value will be sampled from the parameter per image.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or*

numpy.random.RandomState, optional) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
`remove_augmenters_inplace`(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
`reseed`(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
`show_grid`(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.

Continued on next page

Table 91 – continued from previous page

to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

BGR = 'BGR'

CIE = 'CIE'

COLORSPACES = {'BGR', 'CIE', 'GRAY', 'HLS', 'HSV', 'Lab', 'Luv', 'RGB', 'YCrCb'}

CV_VARS = {'BGR2CIE': <MagicMock id='140380134233760'>, 'BGR2GRAY': <MagicMock id='140

GRAY = 'GRAY'

HLS = 'HLS'

HSV = 'HSV'

Lab = 'Lab'

Luv = 'Luv'

RGB = 'RGB'

YCrCb = 'YCrCb'

get_parameters(*self*)

class imgaug.augmenters.color.**Grayscale**(*alpha=0*, *from_colorspace='RGB'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.color.ChangeColorspace*

Augmenter to convert images to their grayscale versions.

Note: Number of output channels is still 3, i.e. this augmenter just "removes" color.

TODO check dtype support

dtype support:

```
See :func:`imgaug.augmenters.color.change_colorspace_`.
```

Parameters

- **alpha** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – The alpha value of the grayscale image when overlayed over the old image. A value close to 1.0 means, that mostly the new grayscale image is visible. A value close to 0.0 means, that mostly the old image is visible.
  - If a number, exactly that value will always be used.
  - If a tuple `(a, b)`, a random value from the range `a <= x <= b` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a StochasticParameter, a value will be sampled from the parameter per image.

- **from_colorspace** (*str, optional*) – The source colorspace (of the input images). Allowed strings are: `RGB`, `BGR`, `GRAY`, `CIE`, `YCrCb`, `HSV`, `HLS`, `Lab`, `Luv`. See *imgaug.augmenters.color.change_colorspace_()*.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Grayscale(alpha=1.0)
```

Creates an augmenter that turns images to their grayscale versions.

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Grayscale(alpha=(0.0, 1.0))
```

Creates an augmenter that turns images to their grayscale versions with an alpha value in the range `0 <= alpha <= 1`. An alpha value of 0.5 would mean, that the output image is 50 percent of the input image and 50 percent of the grayscale image (i.e. 50 percent of color removed).

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.

Continued on next page

Table 92 – continued from previous page

copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

imgaug.augmenters.color.**InColorspace**(*to_colorspace*, *from_colorspace='RGB'*, *children=None*, *name=None*, *deterministic=False*, *random_state=None*)

Deprecated. Use WithColorspace instead.

Convert images to another colorspace.

class imgaug.augmenters.color.**KMeansColorQuantization**(*n_colors=(2, 16)*, *from_colorspace='RGB'*, *to_colorspace=['RGB', 'Lab']*, *max_size=128*, *interpolation='linear'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: imgaug.augmenters.color._AbstractColorQuantization

Quantize colors using k-Means clustering.

This "collects" the colors from the input image, groups them into k clusters using k-Means clustering and replaces the colors in the input image using the cluster centroids.

This is slower than UniformColorQuantization, but adapts dynamically to the color range in the input image.

Note: This augmenter expects input images to be either grayscale or to have 3 or 4 channels and use colorspace *from_colorspace*. If images have 4 channels, it is assumed that the 4th channel is an alpha channel and it will not be quantized.

dtype support:

```
if (image size <= max_size)::

    minimum of (
        ``imgaug.augmenters.color.ChangeColorspace``,
        :func:`imgaug.augmenters.color.quantize_colors_kmeans`
    )

if (image size > max_size)::

    minimum of (
        ``imgaug.augmenters.color.ChangeColorspace``,
        :func:`imgaug.augmenters.color.quantize_colors_kmeans`,
        :func:`imgaug.imgaug.imresize_single_image`
    )
```

Parameters

- **n_colors** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) – Target number of colors in the generated output image. This corresponds to the number of clusters in k-Means, i.e. `k`. Sampled values below `2` will always be clipped to `2`.
  - If a number, exactly that value will always be used.
  - If a tuple `(a, b)`, then a value from the discrete interval `[a..b]` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.
- **to_colorspace** (*None or str or list of str or imgaug.parameters.StochasticParameter*) – The colorspace in which to perform the quantization. See *imgaug.augmenters.color.change_colorspace_()* for valid values. This will be ignored for grayscale input images.
  - If `None` the colorspace of input images will not be changed.
  - If a string, it must be among the allowed colorspaces.
  - If a list, it is expected to be a list of strings, each one being an allowed colorspace. A random element from the list will be chosen per image.
  - If a StochasticParameter, it is expected to return string. A new sample will be drawn per image.
- **from_colorspace** (*str, optional*) – The colorspace of the input images. See *to_colorspace*. Only a single string is allowed.
- **max_size** (*int or None, optional*) – Maximum image size at which to perform the augmentation. If the width or height of an image exceeds this value, it will be downscaled before running the augmentation so that the longest side matches *max_size*. This is done to speed

up the augmentation. The final output image has the same size as the input image. Use `None` to apply no downscaling.

- **interpolation** (*int or str, optional*) – Interpolation method to use during downscaling when *max_size* is exceeded. Valid methods are the same as in *imgaug.imgaug.imresize_single_image()*.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.KMeansColorQuantization()
```

Create an augmenter to apply k-Means color quantization to images using a random amount of colors, sampled uniformly from the interval `[2..16]`. It assumes the input image colorspace to be `RGB` and clusters colors randomly in `RGB` or `Lab` colorspace.

```
>>> aug = iaa.KMeansColorQuantization(n_colors=8)
```

Create an augmenter that quantizes images to (up to) eight colors.

```
>>> aug = iaa.KMeansColorQuantization(n_colors=(4, 16))
```

Create an augmenter that quantizes images to (up to) `n` colors, where `n` is randomly and uniformly sampled from the discrete interval `[4..16]`.

```
>>> aug = iaa.KMeansColorQuantization(
>>>     from_colorspace=iaa.CSPACE_BGR)
```

Create an augmenter that quantizes input images that are in `BGR` colorspace. The quantization happens in `RGB` or `Lab` colorspace, into which the images are temporarily converted.

```
>>> aug = iaa.KMeansColorQuantization(
>>>     to_colorspace=[iaa.CSPACE_RGB, iaa.CSPACE_HSV])
```

Create an augmenter that quantizes images by clustering colors randomly in either `RGB` or `HSV` colorspace. The assumed input colorspace of images is `RGB`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.

Continued on next page

Table 93 – continued from previous page

augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ...[, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.color.**MultiplyHue**(*mul=(-1.0, 1.0)*, *from_colorspace='RGB'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.color.MultiplyHueAndSaturation*

Multiply the hue of images by random values.

The augmenter first transforms images to HSV colorspace, then multiplies the pixel values in the H channel and afterwards converts back to RGB.

This augmenter is a shortcut for `MultiplyHueAndSaturation(mul_hue=...)`.

dtype support:

```
See `imgaug.augmenters.color.MultiplyHueAndSaturation`.
```

Parameters

- **mul** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Multiplier with which to multiply all hue values. This is expected to be in the range `-10.0` to `+10.0` and will automatically be projected to an angular representation using `(hue/255) * (360/2)` (OpenCV's hue representation is in the range `[0, 180]` instead of `[0, 360]`). Only this or *mul* may be set, not both.
  - If a number, then that multiplier will be used for all images.
  - If a tuple `(a, b)`, then a value from the continuous range `[a, b]` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a StochasticParameter, then a value will be sampled from that parameter per image.
- **from_colorspace** (*str, optional*) – See *imgaug.augmenters.color.change_colorspace_()*.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.MultiplyHue((0.5, 1.5))
```

Multiply the hue channel of images using random values between `0.5` and `1.5`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.

Continued on next page

Table 94 – continued from previous page

augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.color.**MultiplyHueAndSaturation**(*mul=None*, *mul_hue=None*, *mul_saturation=None*, *per_channel=False*, *from_colorspace='RGB'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.color.WithHueAndSaturation*

Multipy hue and saturation by random values.

The augmenter first transforms images to HSV colorspace, then multiplies the pixel values in the H and S channels and afterwards converts back to RGB.

This augmenter is a wrapper around `WithHueAndSaturation`.

dtype support:

```
See `imgaug.augmenters.color.WithHueAndSaturation`.
```

Parameters

- **mul** (*None or number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Multiplier with which to multiply all hue *and* saturation values of all pixels. It is expected to be in the range `-10.0` to `+10.0`. Note that values of `0.0` or lower will remove all saturation.
  - If this is `None`, *mul_hue* and/or *mul_saturation* may be set to values other than `None`.
  - If a number, then that multiplier will be used for all images.
  - If a tuple `(a, b)`, then a value from the continuous range `[a, b]` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a StochasticParameter, then a value will be sampled from that parameter per image.
- **mul_hue** (*None or number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Multiplier with which to multiply all hue values. This is expected to be in the range `-10.0` to `+10.0` and will automatically be projected to an angular representation using `(hue/255) * (360/2)` (OpenCV's hue representation is in the range `[0, 180]` instead of `[0, 360]`). Only this or *mul* may be set, not both.
  - If this and *mul_saturation* are both `None`, *mul* may be set to a non-`None` value.
  - If a number, then that multiplier will be used for all images.
  - If a tuple `(a, b)`, then a value from the continuous range `[a, b]` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a StochasticParameter, then a value will be sampled from that parameter per image.
- **mul_saturation** (*None or number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Multiplier with which to multiply all saturation values. It is expected to be in the range `0.0` to `+10.0`. Only this or *mul* may be set, not both.
  - If this and *mul_hue* are both `None`, *mul* may be set to a non-`None` value.
  - If a number, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value from the continuous range `[a, b]` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a StochasticParameter, then a value will be sampled from that parameter per image.

- **per_channel** (*bool or float, optional*) – Whether to sample per image only one value from *mul* and use it for both hue and saturation (`False`) or to sample independently one value for hue and one for saturation (`True`). If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`, otherwise as `False`.

  This parameter has no effect if *mul_hue* and/or *mul_saturation* are used instead of *mul*.

- **from_colorspace** (*str, optional*) – See *imgaug.augmenters.color.change_colorspace_()*.

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.MultiplyHueAndSaturation((0.5, 1.5), per_channel=True)
```

Multiply hue and saturation by random values between `0.5` and `1.5` (independently per channel and the same value for all pixels within that channel). The hue will be automatically projected to an angular representation.

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.MultiplyHueAndSaturation(mul_hue=(0.5, 1.5))
```

Multiply only the hue by random values between `0.5` and `1.5`.

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.MultiplyHueAndSaturation(mul_saturation=(0.5, 1.5))
```

Multiply only the saturation by random values between `0.5` and `1.5`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.

Continued on next page

Table 95 – continued from previous page

augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.color.**MultiplySaturation**(*mul=(0.0, 3.0), from_colorspace='RGB', name=None, deterministic=False, random_state=None*)

Bases: *imgaug.augmenters.color.MultiplyHueAndSaturation*

Multiply the saturation of images by random values.

The augmenter first transforms images to HSV colorspace, then multiplies the pixel values in the H channel and afterwards converts back to RGB.

This augmenter is a shortcut for `MultiplyHueAndSaturation(mul_saturation=...)`.

dtype support:

```
See `imgaug.augmenters.color.MultiplyHueAndSaturation`.
```

Parameters

- **mul** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Multiplier with which to multiply all saturation values. It is expected to be in the range `0.0` to `+10.0`.
  - If a number, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value from the continuous range `[a, b]` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a StochasticParameter, then a value will be sampled from that parameter per image.
- **from_colorspace** (*str, optional*) – See *imgaug.augmenters.color.change_colorspace_()*.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.MultiplySaturation((0.5, 1.5))
```

Multiply the saturation channel of images using random values between `0.5` and `1.5`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.

Continued on next page

Table 96 – continued from previous page

augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.color.**UniformColorQuantization**(*n_colors=(2, 16)*, *from_colorspace='RGB'*, *to_colorspace=None*, *max_size=None*, *interpolation='linear'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: imgaug.augmenters.color._AbstractColorQuantization

Quantize colors into N bins with regular distance.

For uint8 images the equation is floor(v/q)*q + q/2 with q = 256/N, where v is a pixel intensity value and N is the target number of colors after quantization.

This augmenter is faster than KMeansColorQuantization, but the set of possible output colors is constant (i.e. independent of the input images). It may produce unsatisfying outputs for input images that are made up of very similar colors.

Note: This augmenter expects input images to be either grayscale or to have 3 or 4 channels and use colorspace *from_colorspace*. If images have 4 channels, it is assumed that the 4th channel is an alpha channel and it will not be quantized.

dtype support:

```
if (image size <= max_size)::

    minimum of (
        ``imgaug.augmenters.color.ChangeColorspace``,
        :func:`imgaug.augmenters.color.quantize_colors_uniform`
    )

if (image size > max_size)::

    minimum of (
        ``imgaug.augmenters.color.ChangeColorspace``,
        :func:`imgaug.augmenters.color.quantize_colors_uniform`,
        :func:`imgaug.imgaug.imresize_single_image`
    )
```

Parameters

- **n_colors** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) –

  Target number of colors to use in the generated output image.

  - If a number, exactly that value will always be used.
  - If a tuple `(a, b)`, then a value from the discrete interval `[a..b]` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.

- **to_colorspace** (*None or str or list of str or imgaug.parameters.StochasticParameter*) – The colorspace in which to perform the quantization. See *imgaug.augmenters.color.change_colorspace_()* for valid values. This will be ignored for grayscale input images.

  - If `None` the colorspace of input images will not be changed.
  - If a string, it must be among the allowed colorspaces.
  - If a list, it is expected to be a list of strings, each one being an allowed colorspace. A random element from the list will be chosen per image.
  - If a StochasticParameter, it is expected to return string. A new sample will be drawn per image.

- **from_colorspace** (*str, optional*) – The colorspace of the input images. See *to_colorspace*. Only a single string is allowed.

- **max_size** (*None or int, optional*) – Maximum image size at which to perform the augmentation. If the width or height of an image exceeds this value, it will be downscaled before running the augmentation so that the longest side matches *max_size*. This is done to speed

up the augmentation. The final output image has the same size as the input image. Use `None` to apply no downscaling.

- **interpolation** (*int or str, optional*) – Interpolation method to use during downscaling when *max_size* is exceeded. Valid methods are the same as in *imgaug.imgaug.imresize_single_image()*.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.UniformColorQuantization()
```

Create an augmenter to apply uniform color quantization to images using a random amount of colors, sampled uniformly from the discrete interval `[2..16]`.

```
>>> aug = iaa.UniformColorQuantization(n_colors=8)
```

Create an augmenter that quantizes images to (up to) eight colors.

```
>>> aug = iaa.UniformColorQuantization(n_colors=(4, 16))
```

Create an augmenter that quantizes images to (up to) `n` colors, where `n` is randomly and uniformly sampled from the discrete interval `[4..16]`.

```
>>> aug = iaa.UniformColorQuantization(
>>>     from_colorspace=iaa.CSPACE_BGR,
>>>     to_colorspace=[iaa.CSPACE_RGB, iaa.CSPACE_HSV])
```

Create an augmenter that uniformly quantizes images in either `RGB` or `HSV` colorspace (randomly picked per image). The input colorspace of all images has to be `BGR`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.

Continued on next page

Table 97 – continued from previous page

augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.color.**WithColorspace**(*to_colorspace*, *from_colorspace='RGB'*, *children=None*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Apply child augmenters within a specific colorspace.

This augumenter takes a source colorspace A and a target colorspace B as well as children C. It changes images from A to B, then applies the child augmenters C and finally changes the colorspace back from B to A. See also ChangeColorspace() for more.

dtype support:

```
See :func:`imgaug.augmenters.color.change_colorspaces_`.
```

Parameters

- **to_colorspace** (*str*) – See *imgaug.augmenters.color.change_colorspace_()*.
- **from_colorspace** (*str, optional*) – See *imgaug.augmenters.color.change_colorspace_()*.
- **children** (*None or Augmenter or list of Augmenters, optional*) – See `imgaug.augmenters.ChangeColorspace.__init__()`.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.WithColorspace(
>>>     to_colorspace=iaa.CSPACE_HSV,
>>>     from_colorspace=iaa.CSPACE_RGB,
>>>     children=iaa.WithChannels(
>>>         0,
>>>         iaa.Add((0, 50))
>>>     )
>>> )
```

Convert to `HSV` colorspace, add a value between `0` and `50` (uniformly sampled per image) to the Hue channel, then convert back to the input colorspace (`RGB`).

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.

Continued on next page

Table 98 – continued from previous page

augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_children_lists (*self*)

Get a list of lists of children of this augmenter.

For most augmenters, the result will be a single empty list. For augmenters with children it will often be a list with one sublist containing all children. In some cases the augmenter will contain multiple distinct lists of children, e.g. an if-list and an else-list. This will lead to a result consisting of a single list with multiple sublists, each representing the respective sublist of children.

E.g. for an if/else-augmenter that executes the children `A1`, `A2` if a condition is met and otherwise executes the children `B1`, `B2`, `B3` the result will be `[[A1, A2], [B1, B2, B3]]`.

IMPORTANT: While the topmost list may be newly created, each of the sublist must be editable inplace resulting in a changed children list of the augmenter. E.g. if an Augmenter `IfElse(condition, [A1, A2], [B1, B2, B3])` returns `[[A1, A2], [B1, B2, B3]]` for a call to *imgaug.augmenters.meta.Augmenter.get_children_lists()* and `A2` is

removed inplace from `[A1, A2]`, then the children lists of `IfElse(...)` must also change to `[A1]`, `[B1, B2, B3]`. This is used in `imgaug.augmeneters.meta.Augmenter.remove_augmenters_inplace()`.

Returns One or more lists of child augmenter. Can also be a single empty list.

Return type list of list of imgaug.augmenters.meta.Augmenter

get_parameters(*self*)

class imgaug.augmenters.color.**WithHueAndSaturation**(*children=None*, *from_colorspace='RGB'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Apply child augmenters to hue and saturation channels.

This augumenter takes an image in a source colorspace, converts it to HSV, extracts the H (hue) and S (saturation) channels, applies the provided child augmenters to these channels and finally converts back to the original colorspace.

The image array generated by this augmenter and provided to its children is in `int16` (**sic!** only augmenters that can handle `int16` arrays can be children!). The hue channel is mapped to the value range `[0, 255]`. Before converting back to the source colorspace, the saturation channel's values are clipped to `[0, 255]`. A modulo operation is applied to the hue channel's values, followed by a mapping from `[0, 255]` to `[0, 180]` (and finally the colorspace conversion).

dtype support:

```
See :func:`imgaug.augmenters.color.change_colorspaces_`.
```

Parameters

- **from_colorspace** (*str, optional*) – See *imgaug.augmenters.color.change_colorspace_()*.
- **children** (*None or Augmenter or list of Augmenters, optional*) – See `imgaug.augmenters.ChangeColorspace.__init__()`.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.WithHueAndSaturation(
>>>     iaa.WithChannels(0, iaa.Add((0, 50)))
>>> )
```

Create an augmenter that will add a random value between `0` and `50` (uniformly sampled per image) hue channel in HSV colorspace. It automatically accounts for the hue being in angular representation, i.e. if the angle goes

beyond 360 degrees, it will start again at 0 degrees. The colorspace is finally converted back to `RGB` (default setting).

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.WithHueAndSaturation([
>>>     iaa.WithChannels(0, iaa.Add((-30, 10))),
>>>     iaa.WithChannels(1, [
>>>         iaa.Multiply((0.5, 1.5)),
>>>         iaa.LinearContrast((0.75, 1.25))
>>>     ])
>>> ])
```

Create an augmenter that adds a random value sampled uniformly from the range `[-30, 10]` to the hue and multiplies the saturation by a random factor sampled uniformly from `[0.5, 1.5]`. It also modifies the contrast of the saturation channel. After these steps, the `HSV` image is converted back to `RGB`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.

Continued on next page

Table 99 – continued from previous page

localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_children_lists(*self*)

Get a list of lists of children of this augmenter.

For most augmenters, the result will be a single empty list. For augmenters with children it will often be a list with one sublist containing all children. In some cases the augmenter will contain multiple distinct lists of children, e.g. an if-list and an else-list. This will lead to a result consisting of a single list with multiple sublists, each representing the respective sublist of children.

E.g. for an if/else-augmenter that executes the children `A1`, `A2` if a condition is met and otherwise executes the children `B1`, `B2`, `B3` the result will be `[[A1, A2], [B1, B2, B3]]`.

IMPORTANT: While the topmost list may be newly created, each of the sublist must be editable inplace resulting in a changed children list of the augmenter. E.g. if an Augmenter `IfElse(condition, [A1, A2], [B1, B2, B3])` returns `[[A1, A2], [B1, B2, B3]]` for a call to *imgaug.augmenters.meta.Augmenter.get_children_lists()* and `A2` is removed inplace from `[A1, A2]`, then the children lists of `IfElse(...)` must also change to `[A1]`, `[B1, B2, B3]`. This is used in `imgaug.augmeneters.meta.Augmenter.remove_augmenters_inplace()`.

Returns One or more lists of child augmenter. Can also be a single empty list.

Return type list of list of imgaug.augmenters.meta.Augmenter

get_parameters(*self*)

imgaug.augmenters.color.**change_colorspace_**(*image*, *to_colorspace*, *from_colorspace='RGB'*)

Change the colorspace of an image inplace.

Note: All outputs of this function are *uint8*. For some colorspaces this may not be optimal.

Note: Output grayscale images will still have three channels.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: no
* ``uint32``: no
* ``uint64``: no
* ``int8``: no
* ``int16``: no
* ``int32``: no
* ``int64``: no
* ``float16``: no
* ``float32``: no
* ``float64``: no
* ``float128``: no
* ``bool``: no
```

Parameters

- **image** (*ndarray*) – The image to convert from one colorspace into another. Usually expected to have shape `(H,W,3)`.
- **to_colorspace** (*str*) – The target colorspace. See the `CSPACE` constants, e.g. `imgaug.augmenters.color.CSPACE_RGB`.
- **from_colorspace** (*str, optional*) – The source colorspace. Analogous to *to_colorspace*. Defaults to `RGB`.

Returns Image with target colorspace. *Can* be the same array instance as was originally provided (i.e. changed inplace). Grayscale images will still have three channels.

Return type ndarray

Examples

```
>>> import imgaug.augmenters as iaa
>>> import numpy as np
>>> # fake RGB image
>>> image_rgb = np.arange(4*4*3).astype(np.uint8).reshape((4, 4, 3))
>>> image_bgr = iaa.change_colorspace_(np.copy(image_rgb), iaa.CSPACE_BGR)
```

imgaug.augmenters.color.**change_colorspaces_**(*images*, *to_colorspaces*, *from_colorspaces='RGB'*)

Change the colorspaces of a batch of images inplace.

Note: All outputs of this function are *uint8*. For some colorspaces this may not be optimal.

Note: Output grayscale images will still have three channels.

---

dtype support:

```
See :func:`imgaug.augmenters.color.change_colorspace_`.
```

Parameters

- **images** (*ndarray or list of ndarray*) – The images to convert from one colorspace into another. Either a list of `(H,W,3)` arrays or a single `(N,H,W,3)` array.

- **to_colorspaces** (*str or list of str*) – The target colorspaces. Either a single string (all images will be converted to the same colorspace) or a list of strings (one per image). See the `CSPACE` constants, e.g. `imgaug.augmenters.color.CSPACE_RGB`.
- **from_colorspaces** (*str or list of str, optional*) – The source colorspace. Analogous to *to_colorspace*. Defaults to `RGB`.

Returns Images with target colorspaces. *Can* contain the same array instances as were originally provided (i.e. changed inplace). Grayscale images will still have three channels.

Return type ndarray or list of ndarray

Examples

```
>>> import imgaug.augmenters as iaa
>>> import numpy as np
>>> # fake RGB image
>>> image_rgb = np.arange(4*4*3).astype(np.uint8).reshape((4, 4, 3))
>>> images_rgb = [image_rgb, image_rgb, image_rgb]
>>> images_rgb_copy = [np.copy(image_rgb) for image_rgb in images_rgb]
>>> images_bgr = iaa.change_colorspaces_(images_rgb_copy, iaa.CSPACE_BGR)
```

Create three example `RGB` images and convert them to `BGR` colorspace.

```
>>> images_rgb_copy = [np.copy(image_rgb) for image_rgb in images_rgb]
>>> images_various = iaa.change_colorspaces_(
>>>     images_rgb_copy, [iaa.CSPACE_BGR, iaa.CSPACE_HSV, iaa.CSPACE_GRAY])
```

Chnage the colorspace of the first image to `BGR`, the one of the second image to `HSV` and the one of the third image to `grayscale` (note that in the latter case the image will still have shape `(H,W,3)`, not `(H,W,1)`).

`imgaug.augmenters.color.`**`quantize_colors_kmeans`**(*image*, *n_colors*, *n_max_iter=10*, *eps=1.0*)

Apply k-Means color quantization to an image.

Code similar to https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_ml/ py_kmeans/py_kmeans_opencv/py_kmeans_opencv.htm

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: no
* ``uint32``: no
* ``uint64``: no
* ``int8``: no
* ``int16``: no
* ``int32``: no
* ``int64``: no
* ``float16``: no
* ``float32``: no
* ``float64``: no
* ``float128``: no
* ``bool``: no
```

Parameters

- **image** (*ndarray*) – Image in which to quantize colors. Expected to be of shape `(H,W)` or `(H,W,C)` with `C` usually being `1` or `3`.
- **n_colors** (*int*) – Maximum number of output colors.

- **n_max_iter** (*int, optional*) – Maximum number of iterations in k-Means.
- **eps** (*float, optional*) – Minimum change of all clusters per k-Means iteration. If all clusters change by less than this amount in an iteration, the clustering is stopped.

Returns Image with quantized colors.

Return type ndarray

Examples

```
>>> import imgaug.augmenters as iaa
>>> import numpy as np
>>> image = np.arange(4 * 4 * 3, dtype=np.uint8).reshape((4, 4, 3))
>>> image_quantized = iaa.quantize_colors_kmeans(image, 6)
```

Generates a `4x4` image with `3` channels, containing consecutive values from `0` to `4*4*3`, leading to an equal number of colors. These colors are then quantized so that only `6` are remaining. Note that the six remaining colors do have to appear in the input image.

`imgaug.augmenters.color.`**`quantize_colors_uniform`**(*image*, *n_colors*)

Quantize colors into N bins with regular distance.

For `uint8` images the equation is `floor(v/q)*q + q/2` with `q = 256/N`, where `v` is a pixel intensity value and `N` is the target number of colors after quantization.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: no
* ``uint32``: no
* ``uint64``: no
* ``int8``: no
* ``int16``: no
* ``int32``: no
* ``int64``: no
* ``float16``: no
* ``float32``: no
* ``float64``: no
* ``float128``: no
* ``bool``: no
```

Parameters

- **image** (*ndarray*) – Image in which to quantize colors. Expected to be of shape `(H,W)` or `(H,W,C)` with `C` usually being `1` or `3`.
- **n_colors** (*int*) – Maximum number of output colors.

Returns Image with quantized colors.

Return type ndarray

Examples

```
>>> import imgaug.augmenters as iaa
>>> import numpy as np
>>> image = np.arange(4 * 4 * 3, dtype=np.uint8).reshape((4, 4, 3))
>>> image_quantized = iaa.quantize_colors_uniform(image, 6)
```

Generates a `4x4` image with `3` channels, containing consecutive values from `0` to `4*4*3`, leading to an equal number of colors. These colors are then quantized so that only `6` are remaining. Note that the six remaining colors do have to appear in the input image.

13.21 imgaug.augmenters.contrast

Augmenters that perform contrast changes.

Do not import directly from this file, as the categorization is not final. Use instead

```
from imgaug import augmenters as iaa
```

and then e.g.

```
seq = iaa.Sequential([iaa.GammaContrast((0.5, 1.5))])
```

List of augmenters:

- GammaContrast
- SigmoidContrast
- LogContrast
- LinearContrast
- AllChannelsHistogramEqualization
- HistogramEqualization
- AllChannelsCLAHE
- CLAHE

class imgaug.augmenters.contrast.**AllChannelsCLAHE**(*clip_limit=40*, *tile_grid_size_px=8*, *tile_grid_size_px_min=3*, *per_channel=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Apply CLAHE to all channels of images in their original colorspaces.

CLAHE (Contrast Limited Adaptive Histogram Equalization) performs histogram equilization within image patches, i.e. over local neighbourhoods.

In contrast to `imgaug.augmenters.contrast.CLAHE`, this augmenter operates directly on all channels of the input images. It does not perform any colorspace transformations and does not focus on specific channels (e.g. `L` in `Lab` colorspace).

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; tested
* ``uint32``: no (1)
* ``uint64``: no (2)
* ``int8``: no (2)
* ``int16``: no (2)
* ``int32``: no (2)
* ``int64``: no (2)
* ``float16``: no (2)
* ``float32``: no (2)
* ``float64``: no (2)
* ``float128``: no (1)
* ``bool``: no (1)

- (1) rejected by cv2
- (2) results in error in cv2: ``cv2.error:
      OpenCV(3.4.2) (...)/clahe.cpp:351: error: (-215:Assertion
      failed) src.type() == (((0) & ((1 << 3) - 1)) + (((1)-1) << 3))
      || _src.type() == (((2) & ((1 << 3) - 1)) + (((1)-1) << 3)) in
      function 'apply'``
```

Parameters

- **clip_limit** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – See `imgaug.augmenters.contrast.CLAHE`.
- **tile_grid_size_px** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter or tuple of tuple of int or tuple of list of int or tuple of imgaug.parameters.StochasticParameter, optional*) – See `imgaug.augmenters.contrast.CLAHE`.
- **tile_grid_size_px_min** (*int, optional*) – See `imgaug.augmenters.contrast.CLAHE`.
- **per_channel** (*bool or float, optional*) – Whether to use the same value for all channels (`False`) or to sample a new value for each channel (`True`). If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`, otherwise as `False`.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.AllChannelsCLAHE()
```

Create an augmenter that applies CLAHE to all channels of input images.

```
>>> aug = iaa.AllChannelsCLAHE(clip_limit=(1, 10))
```

Same as in the previous example, but the *clip_limit* used by CLAHE is uniformly sampled per image from the interval `[1, 10]`. Some images will therefore have stronger contrast than others (i.e. higher clip limit values).

```
>>> aug = iaa.AllChannelsCLAHE(clip_limit=(1, 10), per_channel=True)
```

Same as in the previous example, but the *clip_limit* is sampled per image *and* channel, leading to different levels of contrast for each channel.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.

Continued on next page

Table 100 – continued from previous page

remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.contrast.**AllChannelsHistogramEqualization**(*name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Apply Histogram Eq. to all channels of images in their original colorspaces.

In contrast to imgaug.augmenters.contrast.HistogramEqualization, this augmenter operates directly on all channels of the input images. It does not perform any colorspace transformations and does not focus on specific channels (e.g. L in Lab colorspace).

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: no (1)
* ``uint32``: no (2)
* ``uint64``: no (1)
* ``int8``: no (1)
* ``int16``: no (1)
* ``int32``: no (1)
* ``int64``: no (1)
* ``float16``: no (2)
* ``float32``: no (1)
* ``float64``: no (1)
* ``float128``: no (2)
* ``bool``: no (1)

- (1) causes cv2 error: ``cv2.error:
      OpenCV(3.4.5) (...)/histogram.cpp:3345: error: (-215:Assertion
      failed) src.type() == CV_8UC1 in function 'equalizeHist'``
- (2) rejected by cv2
```

Parameters

- **name** (*None or str, optional*) – See imgaug.augmenters.meta.Augmenter.__init__().
- **deterministic** (*bool, optional*) – See imgaug.augmenters.meta.Augmenter.__init__().

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.AllChannelsHistogramEqualization()
```

Create an augmenter that applies histogram equalization to all channels of input images in the original colorspaces.

```
>>> aug = iaa.Alpha((0.0, 1.0), iaa.AllChannelsHistogramEqualization())
```

Same as in the previous example, but alpha-blends the contrast-enhanced augmented images with the original input images using random blend strengths. This leads to random strengths of the contrast adjustment.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.

Continued on next page

Table 101 – continued from previous page

find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.contrast.**CLAHE**(*clip_limit=40*, *tile_grid_size_px=8*, *tile_grid_size_px_min=3*, *from_colorspace='RGB'*, *to_colorspace='Lab'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Apply CLAHE to L/V/L channels in HLS/HSV/Lab colorspaces.

This augmenter applies CLAHE (Contrast Limited Adaptive Histogram Equalization) to images, a form of histogram equalization that normalizes within local image patches. The augmenter transforms input images to a target colorspace (e.g. `Lab`), extracts an intensity-related channel from the converted images (e.g. `L` for `Lab`), applies CLAHE to the channel and then converts the resulting image back to the original colorspace.

Grayscale images (images without channel axis or with only one channel axis) are automatically handled, *from_colorspace* does not have to be adjusted for them. For images with four channels (e.g. `RGBA`), the fourth channel is ignored in the colorspace conversion (e.g. from an `RGBA` image, only the `RGB` part is converted, normalized, converted back and concatenated with the input `A` channel). Images with unusual channel numbers (2, 5 or more than 5) are normalized channel-by-channel (same behaviour as `AllChannelsCLAHE`, though a warning will be raised).

If you want to apply CLAHE to each channel of the original input image's colorspace (without any colorspace conversion), use `imgaug.augmenters.contrast.AllChannelsCLAHE` instead.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: no (1)
* ``uint32``: no (1)
* ``uint64``: no (1)
* ``int8``: no (1)
* ``int16``: no (1)
* ``int32``: no (1)
```

(continues on next page)

(continued from previous page)

```
* ``int64``: no (1)
* ``float16``: no (1)
* ``float32``: no (1)
* ``float64``: no (1)
* ``float128``: no (1)
* ``bool``: no (1)

- (1) This augmenter uses ChangeColorspace, which is currently
      limited to ``uint8``.
```

Parameters

- **clip_limit** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Clipping limit. Higher values result in stronger contrast. OpenCV uses a default of `40`, though values around `5` seem to already produce decent contrast.
  - If a number, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value from the range `[a, b]` will be used per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.
- **tile_grid_size_px** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter or tuple of tuple of int or tuple of list of int or tuple of imgaug.parameters.StochasticParameter, optional*) –

  Kernel size, i.e. size of each local neighbourhood in pixels.
  - If an `int`, then that value will be used for all images for both kernel height and width.
  - If a tuple `(a, b)`, then a value from the discrete interval `[a..b]` will be uniformly sampled per image.
  - If a list, then a random value will be sampled from that list per image and used for both kernel height and width.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter per image and used for both kernel height and width.
  - If a tuple of tuple of `int` given as `((a, b), (c, d))`, then two values will be sampled independently from the discrete ranges `[a..b]` and `[c..d]` per image and used as the kernel height and width.
  - If a tuple of lists of `int`, then two values will be sampled independently per image, one from the first list and one from the second, and used as the kernel height and width.
  - If a tuple of `StochasticParameter`, then two values will be sampled indepdently per image, one from the first parameter and one from the second, and used as the kernel height and width.
- **tile_grid_size_px_min** (*int, optional*) – Minimum kernel size in px, per axis. If the sampling results in a value lower than this minimum, it will be clipped to this value.
- **from_colorspace** (*{"RGB", "BGR", "HSV", "HLS", "Lab"}, optional*) – Colorspace of the input images. If any input image has only one or zero channels, this setting will be ignored and it will be assumed that the input is grayscale. If a fourth channel is present in

an input image, it will be removed before the colorspace conversion and later re-added. See also *imgaug.augmenters.color.change_colorspace_()* for details.

- **to_colorspace** (*{"Lab", "HLS", "HSV"}, optional*) – Colorspace in which to perform CLAHE. For `Lab`, CLAHE will only be applied to the first channel (`L`), for `HLS` to the second (`L`) and for `HSV` to the third (`V`). To apply CLAHE to all channels of an input image (without colorspace conversion), see `imgaug.augmenters.contrast.AllChannelsCLAHE`.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.CLAHE()
```

Create a standard CLAHE augmenter.

```
>>> aug = iaa.CLAHE(clip_limit=(1, 10))
```

Create a CLAHE augmenter with a clip limit uniformly sampled from `[1..10]`, where `1` is rather low contrast and `10` is rather high contrast.

```
>>> aug = iaa.CLAHE(tile_grid_size_px=(3, 21))
```

Create a CLAHE augmenter with kernel sizes of `SxS`, where `S` is uniformly sampled from `[3..21]`. Sampling happens once per image.

```
>>> aug = iaa.CLAHE(
>>>     tile_grid_size_px=iap.Discretize(iap.Normal(loc=7, scale=2)),
>>>     tile_grid_size_px_min=3)
```

Create a CLAHE augmenter with kernel sizes of `SxS`, where `S` is sampled from `N(7, 2)`, but does not go below `3`.

```
>>> aug = iaa.CLAHE(tile_grid_size_px=((3, 21), [3, 5, 7]))
```

Create a CLAHE augmenter with kernel sizes of `HxW`, where `H` is uniformly sampled from `[3..21]` and `W` is randomly picked from the list `[3, 5, 7]`.

```
>>> aug = iaa.CLAHE(
>>>     from_colorspace=iaa.CSPACE_BGR,
>>>     to_colorspace=iaa.CSPACE_HSV)
```

Create a CLAHE augmenter that converts images from BGR colorspace to HSV colorspace and then applies the local histogram equalization to the `V` channel of the images (before converting back to `BGR`). Alternatively, `Lab` (default) or `HLS` can be used as the target colorspace. Grayscale images (no channels / one channel) are never converted and are instead directly normalized (i.e. *from_colorspace* does not have to be changed for them).

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
`remove_augmenters_inplace`(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
`reseed`(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
`show_grid`(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
`to_deterministic`(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

BGR = 'BGR'

HLS = 'HLS'

HSV = 'HSV'

Lab = 'Lab'

RGB = 'RGB'

get_parameters(*self*)

class imgaug.augmenters.contrast.**GammaContrast**(*gamma=1*, *per_channel=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: imgaug.augmenters.contrast._ContrastFuncWrapper

Adjust image contrast by scaling pixel values to 255*((v/255)**gamma).

Values in the range gamma=(0.5, 2.0) seem to be sensible.

dtype support:

```
See :func:`imgaug.augmenters.contrast.adjust_contrast_gamma`.
```

Parameters

- **gamma** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) –

  Exponent for the contrast adjustment. Higher values darken the image.

  - If a number, then that value will be used for all images.
  - If a tuple (a, b), then a value from the range [a, b] will be used per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a StochasticParameter, then a value will be sampled per image from that parameter.

- **per_channel** (*bool or float, optional*) – Whether to use the same value for all channels (False) or to sample a new value for each channel (True). If this value is a float p, then for p percent of all images *per_channel* will be treated as True, otherwise as False.
- **name** (*None or str, optional*) – See imgaug.augmenters.meta.Augmenter.__init__().
- **deterministic** (*bool, optional*) – See imgaug.augmenters.meta.Augmenter.__init__().
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See imgaug.augmenters.meta.Augmenter.__init__().

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.GammaContrast((0.5, 2.0))
```

Modify the contrast of images according to 255*((v/255)**gamma), where v is a pixel value and gamma is sampled uniformly from the interval [0.5, 2.0] (once per image).

```
>>> aug = iaa.GammaContrast((0.5, 2.0), per_channel=True)
```

Same as in the previous example, but `gamma` is sampled once per image *and* channel.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
`remove_augmenters_inplace`(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
`reseed`(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.

Continued on next page

Table 103 – continued from previous page

show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.contrast.**HistogramEqualization**(*from_colorspace='RGB'*, *to_colorspace='Lab'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Apply Histogram Eq. to L/V/L channels of images in HLS/HSV/Lab colorspaces.

This augmenter is similar to imgaug.augmenters.contrast.CLAHE.

The augmenter transforms input images to a target colorspace (e.g. Lab), extracts an intensity-related channel from the converted images (e.g. L for Lab), applies Histogram Equalization to the channel and then converts the resulting image back to the original colorspace.

Grayscale images (images without channel axis or with only one channel axis) are automatically handled, *from_colorspace* does not have to be adjusted for them. For images with four channels (e.g. RGBA), the fourth channel is ignored in the colorspace conversion (e.g. from an RGBA image, only the RGB part is converted, normalized, converted back and concatenated with the input A channel). Images with unusual channel numbers (2, 5 or more than 5) are normalized channel-by-channel (same behaviour as AllChannelsHistogramEqualization, though a warning will be raised).

If you want to apply HistogramEqualization to each channel of the original input image's colorspace (without any colorspace conversion), use imgaug.augmenters.contrast.AllChannelsHistogramEqualization instead.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: no (1)
* ``uint32``: no (1)
* ``uint64``: no (1)
* ``int8``: no (1)
* ``int16``: no (1)
* ``int32``: no (1)
* ``int64``: no (1)
* ``float16``: no (1)
* ``float32``: no (1)
* ``float64``: no (1)
* ``float128``: no (1)
* ``bool``: no (1)

- (1) This augmenter uses AllChannelsHistogramEqualization, which only supports␣
↪``uint8``.
```

Parameters

- **from_colorspace** (*{"RGB", "BGR", "HSV", "HLS", "Lab"}, optional*) – Colorspace of the input images. If any input image has only one or zero channels, this setting will be ignored and it will be assumed that the input is grayscale. If a fourth channel is present in

an input image, it will be removed before the colorspace conversion and later re-added. See also *imgaug.augmenters.color.change_colorspace_()* for details.

- **to_colorspace** (*{"Lab", "HLS", "HSV"}, optional*) – Colorspace in which to perform Histogram Equalization. For `Lab`, the equalization will only be applied to the first channel (`L`), for `HLS` to the second (`L`) and for `HSV` to the third (`V`). To apply histogram equalization to all channels of an input image (without colorspace conversion), see `imgaug.augmenters.contrast.AllChannelsHistogramEqualization`.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.HistogramEqualization()
```

Create an augmenter that converts images to `HLS`/`HSV`/`Lab` colorspaces, extracts intensity-related channels (i.e. `L`/`V`/`L`), applies histogram equalization to these channels and converts back to the input colorspace.

```
>>> aug = iaa.Alpha((0.0, 1.0), iaa.HistogramEqualization())
```

Same as in the previous example, but alpha blends the result, leading to various strengths of contrast normalization.

```
>>> aug = iaa.HistogramEqualization(
>>>     from_colorspace=iaa.CSPACE_BGR,
>>>     to_colorspace=iaa.CSPACE_HSV)
```

Same as in the first example, but the colorspace of input images has to be `BGR` (instead of default `RGB`) and the histogram equalization is applied to the `V` channel in `HSV` colorspace.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.

Continued on next page

Table 104 – continued from previous page

augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

BGR = 'BGR'

HLS = 'HLS'

HSV = 'HSV'

Lab = 'Lab'

RGB = 'RGB'

get_parameters(*self*)

class imgaug.augmenters.contrast.**LinearContrast**(*alpha=1*, *per_channel=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: `imgaug.augmenters.contrast._ContrastFuncWrapper`

Adjust contrast by scaling each pixel to `127 + alpha*(v-127)`.

dtype support:

```
See :func:`imgaug.augmenters.contrast.adjust_contrast_linear`.
```

Parameters

- **alpha** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Multiplier to linearly pronounce (`>1.0`), dampen (`0.0` to `1.0`) or invert (`<0.0`) the difference between each pixel value and the dtype's center value, e.g. `127` for `uint8`.
  - If a number, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value from the interval `[a, b]` will be used per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.
- **per_channel** (*bool or float, optional*) – Whether to use the same value for all channels (`False`) or to sample a new value for each channel (`True`). If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`, otherwise as `False`.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.LinearContrast((0.4, 1.6))
```

Modify the contrast of images according to *127 + alpha*(v-127)*‘, where `v` is a pixel value and `alpha` is sampled uniformly from the interval `[0.4, 1.6]` (once per image).

```
>>> aug = iaa.LinearContrast((0.4, 1.6), per_channel=True)
```

Same as in the previous example, but `alpha` is sampled once per image *and* channel.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.

Continued on next page

Table 105 – continued from previous page

augment_batch(self, batch[, hooks])	Augment a single batch.
augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ...[, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.contrast.**LogContrast**(*gain=1*, *per_channel=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: imgaug.augmenters.contrast._ContrastFuncWrapper

Adjust image contrast by scaling pixels to 255*gain*log_2(1+v/255).

This augmenter is fairly similar to `imgaug.augmenters.arithmetic.Multiply`.

dtype support:

```
See :func:`imgaug.augmenters.contrast.adjust_contrast_log`.
```

Parameters

- **gain** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Multiplier for the logarithm result. Values around `1.0` lead to a contrast-adjusted images. Values above `1.0` quickly lead to partially broken images due to exceeding the datatype's value range.
  - If a number, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value from the interval `[a, b]` will uniformly sampled be used per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.
- **per_channel** (*bool or float, optional*) – Whether to use the same value for all channels (`False`) or to sample a new value for each channel (`True`). If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`, otherwise as `False`.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.LogContrast(gain=(0.6, 1.4))
```

Modify the contrast of images according to `255*gain*log_2(1+v/255)`, where `v` is a pixel value and `gain` is sampled uniformly from the interval `[0.6, 1.4]` (once per image).

```
>>> aug = iaa.LogContrast(gain=(0.6, 1.4), per_channel=True)
```

Same as in the previous example, but `gain` is sampled once per image *and* channel.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.

Continued on next page

Table 106 – continued from previous page

augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ...[, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.contrast.**SigmoidContrast**(*gain=10*, *cutoff=0.5*, *per_channel=False*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: imgaug.augmenters.contrast._ContrastFuncWrapper

Adjust image contrast to `255*1/(1+exp(gain*(cutoff-I_ij/255)))`.

Values in the range `gain=(5, 20)` and `cutoff=(0.25, 0.75)` seem to be sensible.

dtype support:

```
See :func:`imgaug.augmenters.contrast.adjust_contrast_sigmoid`.
```

Parameters

- **gain** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Multiplier for the sigmoid function's output. Higher values lead to quicker changes from dark to light pixels.
  - If a number, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value from the interval `[a, b]` will be sampled uniformly per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.
- **cutoff** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Cutoff that shifts the sigmoid function in horizontal direction. Higher values mean that the switch from dark to light pixels happens later, i.e. the pixels will remain darker.
  - If a number, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value from the range `[a, b]` will be used per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.
- **per_channel** (*bool or float, optional*) – Whether to use the same value for all channels (`False`) or to sample a new value for each channel (`True`). If this value is a float `p`, then for `p` percent of all images *per_channel* will be treated as `True`, otherwise as `False`.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.SigmoidContrast(gain=(3, 10), cutoff=(0.4, 0.6))
```

Modify the contrast of images according to `255*1/(1+exp(gain*(cutoff-v/255)))`, where `v` is a pixel value, `gain` is sampled uniformly from the interval `[3, 10]` (once per image) and `cutoff` is sampled uniformly from the interval `[0.4, 0.6]` (also once per image).

```
>>> aug = iaa.SigmoidContrast(
>>>     gain=(3, 10), cutoff=(0.4, 0.6), per_channel=True)
```

Same as in the previous example, but `gain` and `cutoff` are each sampled once per image *and* channel.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
`remove_augmenters_inplace`(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
`reseed`(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.

Continued on next page

Table 107 – continued from previous page

show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

imgaug.augmenters.contrast.**adjust_contrast_gamma**(*arr*, *gamma*)

Adjust image contrast by scaling pixel values to `255*((v/255)**gamma)`.

dtype support:

```
* ``uint8``: yes; fully tested (1) (2) (3)
* ``uint16``: yes; tested (2) (3)
* ``uint32``: yes; tested (2) (3)
* ``uint64``: yes; tested (2) (3) (4)
* ``int8``: limited; tested (2) (3) (5)
* ``int16``: limited; tested (2) (3) (5)
* ``int32``: limited; tested (2) (3) (5)
* ``int64``: limited; tested (2) (3) (4) (5)
* ``float16``: limited; tested (5)
* ``float32``: limited; tested (5)
* ``float64``: limited; tested (5)
* ``float128``: no (6)
* ``bool``: no (7)

- (1) Handled by ``cv2``. Other dtypes are handled by ``skimage``.
- (2) Normalization is done as ``I_ij/max``, where ``max`` is the
      maximum value of the dtype, e.g. 255 for ``uint8``. The
      normalization is reversed afterwards, e.g. ``result*255`` for
      ``uint8``.
- (3) Integer-like values are not rounded after applying the contrast
      adjustment equation (before inverting the normalization to
      ``[0.0, 1.0]`` space), i.e. projection from continuous
      space to discrete happens according to floor function.
- (4) Note that scikit-image doc says that integers are converted to
      ``float64`` values before applying the contrast normalization
      method. This might lead to inaccuracies for large 64bit integer
      values. Tests showed no indication of that happening though.
- (5) Must not contain negative values. Values >=0 are fully supported.
- (6) Leads to error in scikit-image.
- (7) Does not make sense for contrast adjustments.
```

Parameters

- **arr** (*numpy.ndarray*) – Array for which to adjust the contrast. Dtype `uint8` is fastest.
- **gamma** (*number*) – Exponent for the contrast adjustment. Higher values darken the image.

Returns Array with adjusted contrast.

Return type numpy.ndarray

imgaug.augmenters.contrast.**adjust_contrast_linear**(*arr*, *alpha*)

Adjust contrast by scaling each pixel to `127 + alpha*(v-127)`.

dtype support:

```
* ``uint8``: yes; fully tested (1) (2)
* ``uint16``: yes; tested (2)
* ``uint32``: yes; tested (2)
* ``uint64``: no (3)
* ``int8``: yes; tested (2)
* ``int16``: yes; tested (2)
* ``int32``: yes; tested (2)
* ``int64``: no (2)
* ``float16``: yes; tested (2)
* ``float32``: yes; tested (2)
* ``float64``: yes; tested (2)
* ``float128``: no (2)
* ``bool``: no (4)

- (1) Handled by ``cv2``. Other dtypes are handled by raw ``numpy``.
- (2) Only tested for reasonable alphas with up to a value of
      around ``100``.
- (3) Conversion to ``float64`` is done during augmentation, hence
      ``uint64``, ``int64``, and ``float128`` support cannot be
      guaranteed.
- (4) Does not make sense for contrast adjustments.
```

Parameters

- **arr** (*numpy.ndarray*) – Array for which to adjust the contrast. Dtype `uint8` is fastest.
- **alpha** (*number*) – Multiplier to linearly pronounce (`>1.0`), dampen (`0.0` to `1.0`) or invert (`<0.0`) the difference between each pixel value and the dtype's center value, e.g. `127` for `uint8`.

Returns Array with adjusted contrast.

Return type numpy.ndarray

`imgaug.augmenters.contrast.`**`adjust_contrast_log`**(*arr*, *gain*)

Adjust image contrast by scaling pixels to `255*gain*log_2(1+v/255)`.

dtype support:

```
* ``uint8``: yes; fully tested (1) (2) (3)
* ``uint16``: yes; tested (2) (3)
* ``uint32``: no; tested (2) (3) (8)
* ``uint64``: no; tested (2) (3) (4) (8)
* ``int8``: limited; tested (2) (3) (5)
* ``int16``: limited; tested (2) (3) (5)
* ``int32``: no; tested (2) (3) (5) (8)
* ``int64``: no; tested (2) (3) (4) (5) (8)
* ``float16``: limited; tested (5)
* ``float32``: limited; tested (5)
* ``float64``: limited; tested (5)
* ``float128``: no (6)
* ``bool``: no (7)

- (1) Handled by ``cv2``. Other dtypes are handled by ``skimage``.
- (2) Normalization is done as ``I_ij/max``, where ``max`` is the
      maximum value of the dtype, e.g. 255 for ``uint8``. The
      normalization is reversed afterwards, e.g. ``result*255`` for
      ``uint8``.
```

(continues on next page)

(continued from previous page)

```
- (3) Integer-like values are not rounded after applying the contrast
     adjustment equation (before inverting the normalization
     to ``[0.0, 1.0]`` space), i.e. projection from continuous
     space to discrete happens according to floor function.
- (4) Note that scikit-image doc says that integers are converted to
     ``float64`` values before applying the contrast normalization
     method. This might lead to inaccuracies for large 64bit integer
     values. Tests showed no indication of that happening though.
- (5) Must not contain negative values. Values >=0 are fully supported.
- (6) Leads to error in scikit-image.
- (7) Does not make sense for contrast adjustments.
- (8) No longer supported since numpy 1.17. Previously: 'yes' for
     ``uint32``, ``uint64``; 'limited' for ``int32``, ``int64``.
```

Parameters

- **arr** (*numpy.ndarray*) – Array for which to adjust the contrast. Dtype `uint8` is fastest.
- **gain** (*number*) – Multiplier for the logarithm result. Values around 1.0 lead to a contrast-adjusted images. Values above 1.0 quickly lead to partially broken images due to exceeding the datatype's value range.

Returns Array with adjusted contrast.

Return type numpy.ndarray

imgaug.augmenters.contrast.**adjust_contrast_sigmoid**(*arr*, *gain*, *cutoff*)

Adjust image contrast to `255*1/(1+exp(gain*(cutoff-I_ij/255)))`.

dtype support:

```
* ``uint8``: yes; fully tested (1) (2) (3)
* ``uint16``: yes; tested (2) (3)
* ``uint32``: yes; tested (2) (3)
* ``uint64``: yes; tested (2) (3) (4)
* ``int8``: limited; tested (2) (3) (5)
* ``int16``: limited; tested (2) (3) (5)
* ``int32``: limited; tested (2) (3) (5)
* ``int64``: limited; tested (2) (3) (4) (5)
* ``float16``: limited; tested (5)
* ``float32``: limited; tested (5)
* ``float64``: limited; tested (5)
* ``float128``: no (6)
* ``bool``: no (7)

- (1) Handled by ``cv2``. Other dtypes are handled by ``skimage``.
- (2) Normalization is done as ``I_ij/max``, where ``max`` is the
     maximum value of the dtype, e.g. 255 for ``uint8``. The
     normalization is reversed afterwards, e.g. ``result*255``
     for ``uint8``.
- (3) Integer-like values are not rounded after applying the contrast
     adjustment equation before inverting the normalization
     to ``[0.0, 1.0]`` space), i.e. projection from continuous
     space to discrete happens according to floor function.
- (4) Note that scikit-image doc says that integers are converted to
     ``float64`` values before applying the contrast normalization
     method. This might lead to inaccuracies for large 64bit integer
```

(continues on next page)

(continued from previous page)

```
      values. Tests showed no indication of that happening though.
- (5) Must not contain negative values. Values >=0 are fully supported.
- (6) Leads to error in scikit-image.
- (7) Does not make sense for contrast adjustments.
```

Parameters

- **arr** (*numpy.ndarray*) – Array for which to adjust the contrast. Dtype `uint8` is fastest.
- **gain** (*number*) – Multiplier for the sigmoid function's output. Higher values lead to quicker changes from dark to light pixels.
- **cutoff** (*number*) – Cutoff that shifts the sigmoid function in horizontal direction. Higher values mean that the switch from dark to light pixels happens later, i.e. the pixels will remain darker.

Returns Array with adjusted contrast.

Return type numpy.ndarray

13.22 imgaug.augmenters.convolutional

Augmenters that are based on applying convolution kernels to images.

Do not import directly from this file, as the categorization is not final. Use instead

```
from imgaug import augmenters as iaa
```

and then e.g.

```
seq = iaa.Sequential([
    iaa.Sharpen((0.0, 1.0)),
    iaa.Emboss((0.0, 1.0))
])
```

List of augmenters:

- Convolve
- Sharpen
- Emboss
- EdgeDetect
- DirectedEdgeDetect

For MotionBlur, see `blur.py`.

class imgaug.augmenters.convolutional.**Convolve**(*matrix=None*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Apply a convolution to input images.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; tested
* ``uint32``: no (1)
* ``uint64``: no (2)
* ``int8``: yes; tested (3)
* ``int16``: yes; tested
* ``int32``: no (2)
* ``int64``: no (2)
* ``float16``: yes; tested (4)
* ``float32``: yes; tested
* ``float64``: yes; tested
* ``float128``: no (1)
* ``bool``: yes; tested (4)

- (1) rejected by ``cv2.filter2D()``.
- (2) causes error: cv2.error: OpenCV(3.4.2) (...)/filter.cpp:4487:
      error: (-213:The function/feature is not implemented)
      Unsupported combination of source format (=1), and destination
      format (=1) in function 'getLinearFilter'.
- (3) mapped internally to ``int16``.
- (4) mapped internally to ``float32``.
```

Parameters

- **matrix** (*None or (H, W) ndarray or imgaug.parameters.StochasticParameter or callable, optional*) –

  The weight matrix of the convolution kernel to apply.

  – If `None`, the input images will not be changed.

  – If a 2D numpy array, that array will always be used for all images and channels as the kernel.

  – If a callable, that method will be called for each image via `parameter(image, C, random_state)`. The function must either return a list of `C` matrices (i.e. one per channel) or a 2D numpy array (will be used for all channels) or a 3D `HxWxC` numpy array. If a list is returned, each entry may be `None`, which will result in no changes to the respective channel.

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> matrix = np.array([[0, -1, 0],
>>>                    [-1, 4, -1],
>>>                    [0, -1, 0]])
>>> aug = iaa.Convolve(matrix=matrix)
```

Convolves all input images with the kernel shown in the `matrix` variable.

```
>>> def gen_matrix(image, nb_channels, random_state):
>>>     matrix_A = np.array([[0, -1, 0],
>>>                          [-1, 4, -1],
>>>                          [0, -1, 0]])
>>>     matrix_B = np.array([[0, 1, 0],
>>>                          [1, -4, 1],
>>>                          [0, 1, 0]])
>>>     if image.shape[0] % 2 == 0:
>>>         return [matrix_A] * nb_channels
>>>     else:
>>>         return [matrix_B] * nb_channels
>>> aug = iaa.Convolve(matrix=gen_matrix)
```

Convolves images that have an even height with matrix A and images having an odd height with matrix B.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.

Continued on next page

Table 108 – continued from previous page

localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.convolutional.**DirectedEdgeDetect**(*alpha=0*, *direction=(0.0, 1.0)*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.convolutional.Convolve*

Detect edges from specified angles and alpha-blend with the input image.

This augmenter first detects edges along a certain angle. Usually, edges are detected in x- or y-direction, while here the edge detection kernel is rotated to match a specified angle. The result of applying the kernel is a black (non-edges) and white (edges) image. That image is alpha-blended with the input image.

dtype support:

```
See ``imgaug.augmenters.convolutional.Convolve``.
```

Parameters

- **alpha** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Blending factor of the edge image. At `0.0`, only the original image is visible, at `1.0` only the edge image is visible.
  - If a number, exactly that value will always be used.
  - If a tuple `(a, b)`, a random value will be sampled from the interval `[a, b]` per image.
  - If a list, a random value will be sampled from that list per image.
  - If a `StochasticParameter`, a value will be sampled from that parameter per image.
- **direction** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Angle (in degrees) of edges to pronounce, where `0` represents `0` degrees and `1.0` represents 360 degrees (both clockwise, starting at the top). Default value is `(0.0, 1.0)`, i.e. pick a random angle per image.
  - If a number, exactly that value will always be used.
  - If a tuple `(a, b)`, a random value will be sampled from the interval `[a, b]` will be sampled per image.

  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, a value will be sampled from the parameter per image.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.DirectedEdgeDetect(alpha=1.0, direction=0)
```

Turn input images into edge images in which edges are detected from the top side of the image (i.e. the top sides of horizontal edges are part of the edge image, while vertical edges are ignored).

```
>>> aug = iaa.DirectedEdgeDetect(alpha=1.0, direction=90/360)
```

Same as before, but edges are detected from the right. Horizontal edges are now ignored.

```
>>> aug = iaa.DirectedEdgeDetect(alpha=1.0, direction=(0.0, 1.0))
```

Same as before, but edges are detected from a random angle sampled uniformly from the interval `[0deg, 360deg]`.

```
>>> aug = iaa.DirectedEdgeDetect(alpha=(0.0, 0.3), direction=0)
```

Similar to the previous examples, but here the edge image is alpha-blended with the input image. The result is a mixture between the edge image and the input image. The blending factor is randomly sampled between `0%` and `30%`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.

Continued on next page

Table 109 – continued from previous page

augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters	

class imgaug.augmenters.convolutional.**EdgeDetect**(*alpha=0*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.convolutional.Convolve*

Generate a black & white edge image and alpha-blend it with the input image.

dtype support:

```
See ``imgaug.augmenters.convolutional.Convolve``.
```

Parameters

- **alpha** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Blending factor of the edge image. At `0.0`, only the original image is visible, at `1.0` only the edge image is visible.

- If a number, exactly that value will always be used.
- If a tuple `(a, b)`, a random value will be sampled from the interval `[a, b]` per image.
- If a list, a random value will be sampled from that list per image.
- If a `StochasticParameter`, a value will be sampled from that parameter per image.

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.EdgeDetect(alpha=(0.0, 1.0))
```

Detect edges in an image, mark them as black (non-edge) and white (edges) and alpha-blend the result with the original input image using a random blending factor between `0%` and `100%`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.

Continued on next page

Table 110 – continued from previous page

draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.convolutional.**Emboss**(*alpha=0*, *strength=1*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.convolutional.Convolve*

Emboss images and alpha-blend the result with the original input images.

The embossed version pronounces highlights and shadows, letting the image look as if it was recreated on a metal plate ("embossed").

dtype support:

```
See ``imgaug.augmenters.convolutional.Convolve``.
```

Parameters

- **alpha** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Blending factor of the embossed image. At `0.0`, only the original image is visible, at `1.0` only its embossed version is visible.
  - If a number, exactly that value will always be used.
  - If a tuple `(a, b)`, a random value will be sampled from the interval `[a, b]` per image.
  - If a list, a random value will be sampled from that list per image.
  - If a `StochasticParameter`, a value will be sampled from that parameter per image.
- **strength** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Parameter that controls the strength of

the embossing. Sane values are somewhere in the interval `[0.0, 2.0]` with `1.0` being the standard embossing effect. Default value is `1.0`.

- If a number, exactly that value will always be used.
- If a tuple `(a, b)`, a random value will be sampled from the interval `[a, b]` per image.
- If a list, then a random value will be sampled from that list per image.
- If a `StochasticParameter`, a value will be sampled from the parameter per image.

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Emboss(alpha=(0.0, 1.0), strength=(0.5, 1.5))
```

Emboss an image with a strength sampled uniformly from the interval `[0.5, 1.5]` and alpha-blend the result with the original input image using a random blending factor between `0%` and `100%`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.

Continued on next page

Table 111 – continued from previous page

copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.convolutional.**Sharpen**(*alpha=0*, *lightness=1*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.convolutional.Convolve*

Sharpen images and alpha-blend the result with the original input images.

dtype support:

```
See ``imgaug.augmenters.convolutional.Convolve``.
```

Parameters

- **alpha** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Blending factor of the sharpened image. At `0.0`, only the original image is visible, at `1.0` only its sharpened version is visible.
  - If a number, exactly that value will always be used.
  - If a tuple `(a, b)`, a random value will be sampled from the interval `[a, b]` per image.
  - If a list, a random value will be sampled from that list per image.
  - If a `StochasticParameter`, a value will be sampled from that parameter per image.

- **lightness** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Lightness/brightness of the sharped image. Sane values are somewhere in the interval `[0.5, 2.0]`. The value `0.0` results in an edge map. Values higher than `1.0` create bright images. Default value is `1.0`.
  - If a number, exactly that value will always be used.
  - If a tuple `(a, b)`, a random value will be sampled from the interval `[a, b]` per image.
  - If a list, a random value will be sampled from that list per image.
  - If a `StochasticParameter`, a value will be sampled from that parameter per image.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Sharpen(alpha=(0.0, 1.0))
```

Sharpens input images and blends the sharpened image with the input image using a random blending factor between `0%` and `100%` (uniformly sampled).

```
>>> aug = iaa.Sharpen(alpha=(0.0, 1.0), lightness=(0.75, 2.0))
```

Sharpens input images with a variable *lightness* sampled uniformly from the interval `[0.75, 2.0]` and with a fully random blending factor (as in the above example).

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.

Continued on next page

Table 112 – continued from previous page

augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters	

13.23 imgaug.augmenters.edges

Augmenters that deal with edge detection.

Do not import directly from this file, as the categorization is not final. Use instead

```
from imgaug import augmenters as iaa
```

and then e.g.

```
seq = iaa.Sequential([
    iaa.Canny()
])
```

List of augmenters:

- Canny

EdgeDetect and DirectedEdgeDetect are currently still in *convolutional.py*.

class imgaug.augmenters.edges.**Canny**(*alpha=(0.0, 1.0)*, *hysteresis_thresholds=((60, 140), (160, 240))*, *sobel_kernel_size=(3, 7)*, *colorizer=None*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Apply a canny edge detector to input images.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: no; not tested
* ``uint32``: no; not tested
* ``uint64``: no; not tested
* ``int8``: no; not tested
* ``int16``: no; not tested
* ``int32``: no; not tested
* ``int64``: no; not tested
* ``float16``: no; not tested
* ``float32``: no; not tested
* ``float64``: no; not tested
* ``float128``: no; not tested
* ``bool``: no; not tested
```

Parameters

- **alpha** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Blending factor to use in alpha blending. A value close to 1.0 means that only the edge image is visible. A value close to 0.0 means that only the original image is visible. A value close to 0.5 means that the images are merged according to *0.5*image + 0.5*edge_image*. If a sample from this parameter is 0, no action will be performed for the corresponding image.
  - If an int or float, exactly that value will be used.
  - If a tuple `(a, b)`, a random value from the range `a <= x <= b` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a StochasticParameter, a value will be sampled from the parameter per image.
- **hysteresis_thresholds** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter or tuple of tuple of number or tuple of list of number or tuple of imgaug.parameters.StochasticParameter, optional*) – Min and max values to use in hysteresis thresholding. (This parameter seems to have not very much effect on the results.) Either a single parameter or a tuple of two parameters. If a single parameter is provided, the sampling happens once for all images with *(N,2)* samples being requested from the parameter, where each first value denotes the hysteresis minimum and each second the maximum. If a tuple of two parameters is provided, one sampling of *(N,)* values is independently performed per parameter (first parameter: hysteresis minimum, second: hysteresis maximum).
  - If this is a single number, both min and max value will always be exactly that value.
  - If this is a tuple of numbers `(a, b)`, two random values from the range `a <= x <= b` will be sampled per image.

  - If this is a list, two random values will be sampled from that list per image.
  - If this is a StochasticParameter, two random values will be sampled from that parameter per image.
  - If this is a tuple `(min, max)` with `min` and `max` both *not* being numbers, they will be treated according to the rules above (i.e. may be a number, tuple, list or StochasticParameter). A single value will be sampled per image and parameter.

- **sobel_kernel_size** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) – Kernel size of the sobel operator initially applied to each image. This corresponds to `apertureSize` in `cv2.Canny()`. If a sample from this parameter is `<=1`, no action will be performed for the corresponding image. The maximum for this parameter is `7` (inclusive). Higher values are not accepted by OpenCV. If an even value `v` is sampled, it is automatically changed to `v-1`.
  - If this is a single integer, the kernel size always matches that value.
  - If this is a tuple of integers `(a, b)`, a random discrete value will be sampled from the range `a <= x <= b` per image.
  - If this is a list, a random value will be sampled from that list per image.
  - If this is a StochasticParameter, a random value will be sampled from that parameter per image.
- **colorizer** (*None or imgaug.augmenters.edges.IBinaryImageColorizer, optional*) – A strategy to convert binary edge images to color images. If this is `None`, an instance of `RandomColorBinaryImageColorizer` is created, which means that each edge image is converted into an `uint8` image, where edge and non-edge pixels each have a different color that was uniformly randomly sampled from the space of all `uint8` colors.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Canny()
```

Create an augmenter that generates random blends between images and their canny edge representations.

```
>>> aug = iaa.Canny(alpha=(0.0, 0.5))
```

Create a canny edge augmenter that generates edge images with a blending factor of max `50%`, i.e. the original (non-edge) image is always at least partially visible.

```
>>> aug = iaa.Canny(
>>>     alpha=(0.0, 0.5),
>>>     colorizer=iaa.RandomColorsBinaryImageColorizer(
>>>         color_true=255,
```

(continues on next page)

(continued from previous page)

```
>>>         color_false=0
>>>     )
>>> )
```

Same as in the previous example, but the edge image always uses the color white for edges and black for the background.

```
>>> aug = iaa.Canny(alpha=(0.5, 1.0), sobel_kernel_size=[3, 7])
```

Create a canny edge augmenter that initially preprocesses images using a sobel filter with kernel size of either `3x3` or `13x13` and alpha-blends with result using a strength of `50%` (both images equally visible) to `100%` (only edge image visible).

```
>>> aug = iaa.Alpha(
>>>     (0.0, 1.0),
>>>     iaa.Canny(alpha=1),
>>>     iaa.MedianBlur(13)
>>> )
```

Create an augmenter that blends a canny edge image with a median-blurred version of the input image. The median blur uses a fixed kernel size of `13x13` pixels.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.

Continued on next page

Table 113 – continued from previous page

find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters (*self*)

class imgaug.augmenters.edges.**IBinaryImageColorizer**

Bases: object

Methods

colorize(self, image_binary, image_original, ...)	Convert a binary image to a colorized one.

colorize (*self*, *image_binary*, *image_original*, *nth_image*, *random_state*)

Convert a binary image to a colorized one.

Parameters

- **image_binary** (*ndarray*) – Boolean `(H,W)` image.
- **image_original** (*ndarray*) – Original `(H,W,C)` input image.
- **nth_image** (*int*) – Index of the image in the batch.
- **random_state** (*imgaug.random.RNG*) – Random state to use.

Returns Colorized form of *image_binary*.

Return type ndarray

class imgaug.augmenters.edges.**RandomColorsBinaryImageColorizer**(*color_true=(0, 255)*, *color_false=(0, 255)*)

Bases: *imgaug.augmenters.edges.IBinaryImageColorizer*

Colorizer using two randomly sampled foreground/background colors.

Parameters

- **color_true** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) – Color of the foreground, i.e. all pixels in binary images that are `True`. This parameter will be queried once per image to generate `(3,)` samples denoting the color. (Note that even for grayscale images three values will be sampled and converted to grayscale according to `0.299*R + 0.587*G + 0.114*B`. This is the same equation that is also used by OpenCV.)
  - If an int, exactly that value will always be used, i.e. every color will be `(v, v, v)` for value `v`.
  - If a tuple `(a, b)`, three random values from the range `a <= x <= b` will be sampled per image.
  - If a list, then three random values will be sampled from that list per image.
  - If a StochasticParameter, three values will be sampled from the parameter per image.
- **color_false** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) – Analogous to *color_true*, but denotes the color for all pixels that are `False` in the binary input image.

Methods

colorize(self, image_binary, image_original, ...)	Convert a binary image to a colorized one.

colorize (*self*, *image_binary*, *image_original*, *nth_image*, *random_state*)

Convert a binary image to a colorized one.

Parameters

- **image_binary** (*ndarray*) – Boolean `(H,W)` image.
- **image_original** (*ndarray*) – Original `(H,W,C)` input image.
- **nth_image** (*int*) – Index of the image in the batch.
- **random_state** (*imgaug.random.RNG*) – Random state to use.

Returns Colorized form of *image_binary*.

Return type ndarray

13.24 imgaug.augmenters.flip

Augmenters that apply mirroring/flipping operations to images.

Do not import directly from this file, as the categorization is not final. Use instead

```
from imgaug import augmenters as iaa
```

and then e.g.

```
seq = iaa.Sequential([
    iaa.Fliplr((0.0, 1.0)),
    iaa.Flipud((0.0, 1.0))
])
```

List of augmenters:

- Fliplr
- Flipud

class imgaug.augmenters.flip.**Fliplr**(*p=0*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Flip/mirror input images horizontally.

Note: The default value for the probability is `0.0`. So, to flip *all* input images use `Fliplr(1.0)` and *not* just `Fliplr()`.

---

dtype support:

```
See :func:`imgaug.augmenters.flip.fliplr`.
```

Parameters

- **p** (*number or imgaug.parameters.StochasticParameter, optional*) – Probability of each image to get flipped.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Fliplr(0.5)
```

Flip `50` percent of all images horizontally.

```
>>> aug = iaa.Fliplr(1.0)
```

Flip all images horizontally.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
`remove_augmenters_inplace`(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
`reseed`(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
`show_grid`(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
`to_deterministic`(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.flip.**Flipud**(*p=0*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Flip/mirror input images vertically.

Note: The default value for the probability is `0.0`. So, to flip *all* input images use `Flipud(1.0)` and *not* just `Flipud()`.

---

dtype support:

```
See :func:`imgaug.augmenters.flip.flipud`.
```

Parameters

- **p** (*number or imgaug.parameters.StochasticParameter, optional*) – Probability of each image to get flipped.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Flipud(0.5)
```

Flip `50` percent of all images vertically.

```
>>> aug = iaa.Flipud(1.0)
```

Flip all images vertically.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.

Continued on next page

Table 117 – continued from previous page

augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

imgaug.augmenters.flip.**HorizontalFlip**(**args*, ***kwargs*)
Alias for Fliplr.

imgaug.augmenters.flip.**VerticalFlip**(**args*, ***kwargs*)
Alias for Flipud.

imgaug.augmenters.flip.**fliplr**(*arr*)
Flip an image-like array horizontally.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; fully tested
* ``uint32``: yes; fully tested
* ``uint64``: yes; fully tested
* ``int8``: yes; fully tested
* ``int16``: yes; fully tested
* ``int32``: yes; fully tested
* ``int64``: yes; fully tested
* ``float16``: yes; fully tested
* ``float32``: yes; fully tested
* ``float64``: yes; fully tested
* ``float128``: yes; fully tested
* ``bool``: yes; fully tested
```

Parameters **arr** (*ndarray*) – A 2D/3D *(H, W, [C])* image array.

Returns Horizontally flipped array.

Return type ndarray

Examples

```
>>> import numpy as np
>>> import imgaug.augmenters.flip as flip
>>> arr = np.arange(16).reshape((4, 4))
>>> arr_flipped = flip.fliplr(arr)
```

Create a `4x4` array and flip it horizontally.

`imgaug.augmenters.flip.`**`flipud`**(*arr*)

Flip an image-like array vertically.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; fully tested
* ``uint32``: yes; fully tested
* ``uint64``: yes; fully tested
* ``int8``: yes; fully tested
* ``int16``: yes; fully tested
* ``int32``: yes; fully tested
* ``int64``: yes; fully tested
* ``float16``: yes; fully tested
* ``float32``: yes; fully tested
* ``float64``: yes; fully tested
* ``float128``: yes; fully tested
* ``bool``: yes; fully tested
```

Parameters **arr** (*ndarray*) – A 2D/3D *(H, W, [C])* image array.

Returns Vertically flipped array.

Return type ndarray

Examples

```
>>> import numpy as np
>>> import imgaug.augmenters.flip as flip
>>> arr = np.arange(16).reshape((4, 4))
>>> arr_flipped = flip.flipud(arr)
```

Create a `4x4` array and flip it vertically.

13.25 imgaug.augmenters.geometric

Augmenters that apply affine or similar transformations.

Do not import directly from this file, as the categorization is not final. Use instead

```
from imgaug import augmenters as iaa
```

and then e.g.

```
seq = iaa.Sequential([
    iaa.Affine(...),
    iaa.PerspectiveTransform(...)
])
```

List of augmenters:

- Affine
- AffineCv2
- PiecewiseAffine
- PerspectiveTransform
- ElasticTransformation
- Rot90

class imgaug.augmenters.geometric.**Affine**(*scale=1.0*, *translate_percent=None*, *translate_px=None*, *rotate=0.0*, *shear=0.0*, *order=1*, *cval=0*, *mode='constant'*, *fit_output=False*, *backend='auto'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Augmenter to apply affine transformations to images.

This is mostly a wrapper around the corresponding classes and functions in OpenCV and skimage..

Affine transformations involve:

- Translation ("move" image on the x-/y-axis)
- Rotation
- Scaling ("zoom" in/out)
- Shear (move one side of the image, turning a square into a trapezoid)

All such transformations can create "new" pixels in the image without a defined content, e.g. if the image is translated to the left, pixels are created on the right. A method has to be defined to deal with these pixel values. The parameters *cval* and *mode* of this class deal with this.

Some transformations involve interpolations between several pixels of the input image to generate output pixel values. The parameter *order* deals with the method of interpolation used for this.

dtype support:

```
if (backend="skimage", order in [0, 1])::

    * ``uint8``: yes; tested
    * ``uint16``: yes; tested
    * ``uint32``: yes; tested (1)
    * ``uint64``: no (2)
    * ``int8``: yes; tested
    * ``int16``: yes; tested
    * ``int32``: yes; tested  (1)
    * ``int64``: no (2)
    * ``float16``: yes; tested
    * ``float32``: yes; tested
    * ``float64``: yes; tested
    * ``float128``: no (2)
    * ``bool``: yes; tested

    - (1) scikit-image converts internally to float64, which might
          affect the accuracy of large integers. In tests this seemed
          to not be an issue.
    - (2) results too inaccurate

if (backend="skimage", order in [3, 4])::

    * ``uint8``: yes; tested
    * ``uint16``: yes; tested
    * ``uint32``: yes; tested (1)
    * ``uint64``: no (2)
    * ``int8``: yes; tested
    * ``int16``: yes; tested
    * ``int32``: yes; tested  (1)
    * ``int64``: no (2)
    * ``float16``: yes; tested
    * ``float32``: yes; tested
    * ``float64``: limited; tested (3)
    * ``float128``: no (2)
    * ``bool``: yes; tested

    - (1) scikit-image converts internally to float64, which might
          affect the accuracy of large integers. In tests this seemed
          to not be an issue.
    - (2) results too inaccurate
    - (3) ``NaN`` around minimum and maximum of float64 value range

if (backend="skimage", order=5])::

        * ``uint8``: yes; tested
        * ``uint16``: yes; tested
        * ``uint32``: yes; tested (1)
        * ``uint64``: no (2)
```

(continues on next page)

(continued from previous page)

```
        * ``int8``: yes; tested
        * ``int16``: yes; tested
        * ``int32``: yes; tested  (1)
        * ``int64``: no (2)
        * ``float16``: yes; tested
        * ``float32``: yes; tested
        * ``float64``: limited; not tested (3)
        * ``float128``: no (2)
        * ``bool``: yes; tested

        - (1) scikit-image converts internally to ``float64``, which
              might affect the accuracy of large integers. In tests
              this seemed to not be an issue.
        - (2) results too inaccurate
        - (3) ``NaN`` around minimum and maximum of float64 value range

if (backend="cv2", order=0)::

    * ``uint8``: yes; tested
    * ``uint16``: yes; tested
    * ``uint32``: no (1)
    * ``uint64``: no (2)
    * ``int8``: yes; tested
    * ``int16``: yes; tested
    * ``int32``: yes; tested
    * ``int64``: no (2)
    * ``float16``: yes; tested (3)
    * ``float32``: yes; tested
    * ``float64``: yes; tested
    * ``float128``: no (1)
    * ``bool``: yes; tested (3)

    - (1) rejected by cv2
    - (2) changed to ``int32`` by cv2
    - (3) mapped internally to ``float32``

if (backend="cv2", order=1):

    * ``uint8``: yes; fully tested
    * ``uint16``: yes; tested
    * ``uint32``: no (1)
    * ``uint64``: no (2)
    * ``int8``: yes; tested (3)
    * ``int16``: yes; tested
    * ``int32``: no (2)
    * ``int64``: no (2)
    * ``float16``: yes; tested (4)
    * ``float32``: yes; tested
    * ``float64``: yes; tested
    * ``float128``: no (1)
    * ``bool``: yes; tested (4)

    - (1) rejected by cv2
    - (2) causes cv2 error: ``cv2.error: OpenCV(3.4.4)
          (...)imgwarp.cpp:1805: error:
          (-215:Assertion failed) ifunc != 0 in function 'remap'``
    - (3) mapped internally to ``int16``
```

(continues on next page)

(continued from previous page)

```
    - (4) mapped internally to ``float32``

if (backend="cv2", order=3):

    * ``uint8``: yes; tested
    * ``uint16``: yes; tested
    * ``uint32``: no (1)
    * ``uint64``: no (2)
    * ``int8``: yes; tested (3)
    * ``int16``: yes; tested
    * ``int32``: no (2)
    * ``int64``: no (2)
    * ``float16``: yes; tested (4)
    * ``float32``: yes; tested
    * ``float64``: yes; tested
    * ``float128``: no (1)
    * ``bool``: yes; tested (4)

    - (1) rejected by cv2
    - (2) causes cv2 error: ``cv2.error: OpenCV(3.4.4)
          (...)imgwarp.cpp:1805: error:
          (-215:Assertion failed) ifunc != 0 in function 'remap'``
    - (3) mapped internally to ``int16``
    - (4) mapped internally to ``float32``
```

Parameters

- **scale** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter or dict {"x": number/tuple/list/StochasticParameter, "y": number/tuple/list/StochasticParameter}, optional*) – Scaling factor to use, where `1.0` denotes "no change" and `0.5` is zoomed out to `50` percent of the original size.
  - If a single number, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`. That value will be used identically for both x- and y-axis.
  - If a list, then a random value will be sampled from that list per image (again, used for both x- and y-axis).
  - If a `StochasticParameter`, then from that parameter a value will be sampled per image (again, used for both x- and y-axis).
  - If a dictionary, then it is expected to have the keys `x` and/or `y`. Each of these keys can have the same values as described above. Using a dictionary allows to set different values for the two axis and sampling will then happen *independently* per axis, resulting in samples that differ between the axes.
- **translate_percent** (*None or number or tuple of number or list of number or imgaug.parameters.StochasticParameter or dict {"x": number/tuple/list/StochasticParameter, "y": number/tuple/list/StochasticParameter}, optional*) – Translation as a fraction of the image height/width (x-translation, y-translation), where `0` denotes "no change" and `0.5` denotes "half of the axis size".
  - If `None` then equivalent to `0.0` unless *translate_px* has a value other than `None`.
  - If a single number, then that value will be used for all images.

  - If a tuple `(a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`. That sampled fraction value will be used identically for both x- and y-axis.
  - If a list, then a random value will be sampled from that list per image (again, used for both x- and y-axis).
  - If a `StochasticParameter`, then from that parameter a value will be sampled per image (again, used for both x- and y-axis).
  - If a dictionary, then it is expected to have the keys `x` and/or `y`. Each of these keys can have the same values as described above. Using a dictionary allows to set different values for the two axis and sampling will then happen *independently* per axis, resulting in samples that differ between the axes.

- **translate_px** (*None or int or tuple of int or list of int or imgaug.parameters.StochasticParameter or dict {"x": int/tuple/list/StochasticParameter, "y": int/tuple/list/StochasticParameter}, optional*) –

  Translation in pixels.

  - If `None` then equivalent to `0` unless *translate_percent* has a value other than `None`.
  - If a single int, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value will be uniformly sampled per image from the discrete interval `[a..b]`. That number will be used identically for both x- and y-axis.
  - If a list, then a random value will be sampled from that list per image (again, used for both x- and y-axis).
  - If a `StochasticParameter`, then from that parameter a value will be sampled per image (again, used for both x- and y-axis).
  - If a dictionary, then it is expected to have the keys `x` and/or `y`. Each of these keys can have the same values as described above. Using a dictionary allows to set different values for the two axis and sampling will then happen *independently* per axis, resulting in samples that differ between the axes.

- **rotate** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Rotation in degrees (**NOT** radians), i.e. expected value range is around `[-360, 360]`. Rotation happens around the *center* of the image, not the top left corner as in some other frameworks.
  - If a number, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]` and used as the rotation value.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, then this parameter will be used to sample the rotation value per image.

- **shear** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Shear in degrees (**NOT** radians), i.e. expected value range is around `[-360, 360]`.
  - If a number, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]` and be used as the rotation value.
  - If a list, then a random value will be sampled from that list per image.

  - If a `StochasticParameter`, then this parameter will be used to sample the shear value per image.

- **order** (*int or iterable of int or imgaug.ALL or imgaug.parameters.StochasticParameter, optional*) –

  Interpolation order to use. Same meaning as in `skimage`:

  - `0`: `Nearest-neighbor`
  - `1`: `Bi-linear` (default)
  - `2`: `Bi-quadratic` (not recommended by skimage)
  - `3`: `Bi-cubic`
  - `4`: `Bi-quartic`
  - `5`: `Bi-quintic`

  Method `0` and `1` are fast, `3` is a bit slower, `4` and `5` are very slow. If the backend is `cv2`, the mapping to OpenCV's interpolation modes is as follows:

  - `0` -> `cv2.INTER_NEAREST`
  - `1` -> `cv2.INTER_LINEAR`
  - `2` -> `cv2.INTER_CUBIC`
  - `3` -> `cv2.INTER_CUBIC`
  - `4` -> `cv2.INTER_CUBIC`

  As datatypes this parameter accepts:

  - If a single `int`, then that order will be used for all images.
  - If a list, then a random value will be sampled from that list per image.
  - If `imgaug.ALL`, then equivalant to list `[0, 1, 3, 4, 5]` in case of `backend=skimage` and otherwise `[0, 1, 3]`.
  - If `StochasticParameter`, then that parameter is queried per image to sample the order value to use.

- **cval** (*number or tuple of number or list of number or imgaug.ALL or imgaug.parameters.StochasticParameter, optional*) – The constant value to use when filling in newly created pixels. (E.g. translating by 1px to the right will create a new 1px-wide column of pixels on the left of the image). The value is only used when *mode=constant*. The expected value range is `[0, 255]` for `uint8` images. It may be a float value.

  - If this is a single number, then that value will be used (e.g. 0 results in black pixels).
  - If a tuple `(a, b)`, then three values (for three image channels) will be uniformly sampled per image from the interval `[a, b]`.
  - If a list, then a random value will be sampled from that list per image.
  - If `imgaug.ALL` then equivalent to tuple ‘‘(0, 255)‘.
  - If a `StochasticParameter`, a new value will be sampled from the parameter per image.

- **mode** (*str or list of str or imgaug.ALL or imgaug.parameters.StochasticParameter, optional*) – Method to use when filling in newly created pixels. Same meaning as in `skimage` (and `numpy.pad()`):

  - `constant`: Pads with a constant value
  - `edge`: Pads with the edge values of array
  - `symmetric`: Pads with the reflection of the vector mirrored along the edge of the array.
  - `reflect`: Pads with the reflection of the vector mirrored on the first and last values of the vector along each axis.
  - `wrap`: Pads with the wrap of the vector along the axis. The first values are used to pad the end and the end values are used to pad the beginning.

  If `cv2` is chosen as the backend the mapping is as follows:

  - `constant` -> `cv2.BORDER_CONSTANT`
  - `edge` -> `cv2.BORDER_REPLICATE`
  - `symmetric` -> `cv2.BORDER_REFLECT`
  - `reflect` -> `cv2.BORDER_REFLECT_101`
  - `wrap` -> `cv2.BORDER_WRAP`

  The datatype of the parameter may be:

  - If a single string, then that mode will be used for all images.
  - If a list of strings, then a random mode will be picked from that list per image.
  - If `imgaug.ALL`, then a random mode from all possible modes will be picked.
  - If `StochasticParameter`, then the mode will be sampled from that parameter per image, i.e. it must return only the above mentioned strings.

- **fit_output** (*bool, optional*) – Whether to modify the affine transformation so that the whole output image is always contained in the image plane (`True`) or accept parts of the image being outside the image plane (`False`). This can be thought of as first applying the affine transformation and then applying a second transformation to "zoom in" on the new image so that it fits the image plane, This is useful to avoid corners of the image being outside of the image plane after applying rotations. It will however negate translation and scaling. Note also that activating this may lead to image sizes differing from the input image sizes. To avoid this, wrap `Affine` in *imgaug.augmenters.size.KeepSizeByResize*, e.g. `KeepSizeByResize(Affine(...))`.

- **backend** (*str, optional*) – Framework to use as a backend. Valid values are `auto`, `skimage` (scikit-image's warp) and `cv2` (OpenCV's warp). If `auto` is used, the augmenter will automatically try to use `cv2` whenever possible (order must be in `[0, 1, 3]`). It will silently fall back to skimage if order/dtype is not supported by cv2. cv2 is generally faster than skimage. It also supports RGB cvals, while skimage will resort to intensity cvals (i.e. 3x the same value as RGB). If `cv2` is chosen and order is `2` or `4`, it will automatically fall back to order `3`.

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Affine(scale=2.0)
```

Zoom in on all images by a factor of `2`.

```
>>> aug = iaa.Affine(translate_px=16)
```

Translate all images on the x- and y-axis by 16 pixels (towards the bottom right) and fill up any new pixels with zero (black values).

```
>>> aug = iaa.Affine(translate_percent=0.1)
```

Translate all images on the x- and y-axis by `10` percent of their width/height (towards the bottom right). The pixel values are computed per axis based on that axis' size. Fill up any new pixels with zero (black values).

```
>>> aug = iaa.Affine(rotate=35)
```

Rotate all images by `35` *degrees*. Fill up any new pixels with zero (black values).

```
>>> aug = iaa.Affine(shear=15)
```

Shear all images by `15` *degrees*. Fill up any new pixels with zero (black values).

```
>>> aug = iaa.Affine(translate_px=(-16, 16))
```

Translate all images on the x- and y-axis by a random value between `-16` and `16` pixels (to the bottom right) and fill up any new pixels with zero (black values). The translation value is sampled once per image and is the same for both axis.

```
>>> aug = iaa.Affine(translate_px={"x": (-16, 16), "y": (-4, 4)})
```

Translate all images on the x-axis by a random value between `-16` and `16` pixels (to the right) and on the y-axis by a random value between `-4` and `4` pixels to the bottom. The sampling happens independently per axis, so even if both intervals were identical, the sampled axis-wise values would likely be different. This also fills up any new pixels with zero (black values).

```
>>> aug = iaa.Affine(scale=2.0, order=[0, 1])
```

Same as in the above *scale* example, but uses (randomly) either nearest neighbour interpolation or linear interpolation. If *order* is not specified, `order=1` would be used by default.

```
>>> aug = iaa.Affine(translate_px=16, cval=(0, 255))
```

Same as in the *translate_px* example above, but newly created pixels are now filled with a random color (sampled once per image and the same for all newly created pixels within that image).

```
>>> aug = iaa.Affine(translate_px=16, mode=["constant", "edge"])
```

Similar to the previous example, but the newly created pixels are filled with black pixels in half of all images (mode `constant` with default *cval* being `0`) and in the other half of all images using `edge` mode, which repeats the color of the spatially closest pixel of the corresponding image edge.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
`remove_augmenters_inplace`(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
`reseed`(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
`show_grid`(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
`to_deterministic`(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.geometric.**AffineCv2**(*scale=1.0*, *translate_percent=None*, *translate_px=None*, *rotate=0.0*, *shear=0.0*, *order=<MagicMock id='140380133237816'>*, *cval=0*, *mode=<MagicMock id='140380133250328'>*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Augmenter to apply affine transformations to images using cv2 (i.e. opencv) backend.

Warning: This augmenter might be removed in the future as `Affine` already offers a cv2 backend (use `backend="cv2"`).

Affine transformations involve:

- Translation ("move" image on the x-/y-axis)
- Rotation
- Scaling ("zoom" in/out)
- Shear (move one side of the image, turning a square into a trapezoid)

All such transformations can create "new" pixels in the image without a defined content, e.g. if the image is translated to the left, pixels are created on the right. A method has to be defined to deal with these pixel values. The parameters *cval* and *mode* of this class deal with this.

Some transformations involve interpolations between several pixels of the input image to generate output pixel values. The parameter *order* deals with the method of interpolation used for this.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: ?
* ``uint32``: ?
* ``uint64``: ?
* ``int8``: ?
* ``int16``: ?
* ``int32``: ?
* ``int64``: ?
* ``float16``: ?
* ``float32``: ?
* ``float64``: ?
* ``float128``: ?
* ``bool``: ?
```

Parameters

- **scale** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter or dict {"x": number/tuple/list/StochasticParameter, "y": number/tuple/list/StochasticParameter}, optional*) – Scaling factor to use, where `1.0` denotes "no change" and `0.5` is zoomed out to `50` percent of the original size.
  - If a single number, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`. That value will be used identically for both x- and y-axis.

  - If a list, then a random value will be sampled from that list per image (again, used for both x- and y-axis).
  - If a `StochasticParameter`, then from that parameter a value will be sampled per image (again, used for both x- and y-axis).
  - If a dictionary, then it is expected to have the keys `x` and/or `y`. Each of these keys can have the same values as described above. Using a dictionary allows to set different values for the two axis and sampling will then happen *independently* per axis, resulting in samples that differ between the axes.

- **translate_percent** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter or dict {"x": number/tuple/list/StochasticParameter, "y": number/tuple/list/StochasticParameter}, optional*) – Translation as a fraction of the image height/width (x-translation, y-translation), where `0` denotes "no change" and `0.5` denotes "half of the axis size".
  - If `None` then equivalent to `0.0` unless *translate_px* has a value other than `None`.
  - If a single number, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`. That sampled fraction value will be used identically for both x- and y-axis.
  - If a list, then a random value will be sampled from that list per image (again, used for both x- and y-axis).
  - If a `StochasticParameter`, then from that parameter a value will be sampled per image (again, used for both x- and y-axis).
  - If a dictionary, then it is expected to have the keys `x` and/or `y`. Each of these keys can have the same values as described above. Using a dictionary allows to set different values for the two axis and sampling will then happen *independently* per axis, resulting in samples that differ between the axes.

- **translate_px** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter or dict {"x": int/tuple/list/StochasticParameter, "y": int/tuple/list/StochasticParameter}, optional*) –

  Translation in pixels.
  - If `None` then equivalent to `0` unless *translate_percent* has a value other than `None`.
  - If a single int, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value will be uniformly sampled per image from the discrete interval `[a..b]`. That number will be used identically for both x- and y-axis.
  - If a list, then a random value will be sampled from that list per image (again, used for both x- and y-axis).
  - If a `StochasticParameter`, then from that parameter a value will be sampled per image (again, used for both x- and y-axis).
  - If a dictionary, then it is expected to have the keys `x` and/or `y`. Each of these keys can have the same values as described above. Using a dictionary allows to set different values for the two axis and sampling will then happen *independently* per axis, resulting in samples that differ between the axes.

- **rotate** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Rotation in degrees (**NOT** radians), i.e. expected value range is around `[-360, 360]`. Rotation happens around the *center* of the image, not the top left corner as in some other frameworks.

  - If a number, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]` and used as the rotation value.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, then this parameter will be used to sample the rotation value per image.
- **shear** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Shear in degrees (**NOT** radians), i.e. expected value range is around `[-360, 360]`.
  - If a number, then that value will be used for all images.
  - If a tuple `(a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]` and be used as the rotation value.
  - If a list, then a random value will be sampled from that list per image.
  - If a `StochasticParameter`, then this parameter will be used to sample the shear value per image.
- **order** (*int or list of int or str or list of str or imaug.ALL or imgaug.parameters.StochasticParameter, optional*) –

  Interpolation order to use. Allowed are:
  - `cv2.INTER_NEAREST` (nearest-neighbor interpolation)
  - `cv2.INTER_LINEAR` (bilinear interpolation, used by default)
  - **`cv2.INTER_CUBIC` (bicubic interpolation over `4x4` pixel** neighborhood)
  - `cv2.INTER_LANCZOS4`
  - string `nearest` (same as `cv2.INTER_NEAREST`)
  - string `linear` (same as `cv2.INTER_LINEAR`)
  - string `cubic` (same as `cv2.INTER_CUBIC`)
  - string `lanczos4` (same as `cv2.INTER_LANCZOS`)

  `INTER_NEAREST` (nearest neighbour interpolation) and `INTER_NEAREST` (linear interpolation) are the fastest.
  - If a single `int`, then that order will be used for all images.
  - If a string, then it must be one of: `nearest`, `linear`, `cubic`, `lanczos4`.
  - If an iterable of `int`/`str`, then for each image a random value will be sampled from that iterable (i.e. list of allowed order values).
  - If `imgaug.ALL`, then equivalant to list `[cv2.INTER_NEAREST, cv2.INTER_LINEAR, cv2.INTER_CUBIC, cv2.INTER_LANCZOS4]`.
  - If `StochasticParameter`, then that parameter is queried per image to sample the order value to use.
- **cval** (*number or tuple of number or list of number or imaug.ALL or imgaug.parameters.StochasticParameter, optional*) – The constant value to use when filling in newly created pixels. (E.g. translating by 1px to the right will create a new 1px-wide column of pixels on the left of the image). The value is only used when *mode=constant*. The expected value range is `[0, 255]` for `uint8` images. It may be a float value.

  - If this is a single number, then that value will be used (e.g. 0 results in black pixels).
  - If a tuple `(a, b)`, then three values (for three image channels) will be uniformly sampled per image from the interval `[a, b]`.
  - If a list, then a random value will be sampled from that list per image.
  - If `imgaug.ALL` then equivalent to tuple ‘‘(0, 255)‘.
  - If a `StochasticParameter`, a new value will be sampled from the parameter per image.

- **mode** (*int or str or list of str or list of int or imgaug.ALL or imgaug.parameters.StochasticParameter,*) – optional Method to use when filling in newly created pixels. Same meaning as in OpenCV's border mode. Let `abcdefgh` be an image's content and `|` be an image boundary after which new pixels are filled in, then the valid modes and their behaviour are the following:
  - `cv2.BORDER_REPLICATE`: `aaaaaa|abcdefgh|hhhhhhh`
  - `cv2.BORDER_REFLECT`: `fedcba|abcdefgh|hgfedcb`
  - `cv2.BORDER_REFLECT_101`: `gfedcb|abcdefgh|gfedcba`
  - `cv2.BORDER_WRAP`: `cdefgh|abcdefgh|abcdefg`
  - **`cv2.BORDER_CONSTANT`: `iiiiii|abcdefgh|iiiiiii`,** where `i` is the defined cval.
  - `replicate`: Same as `cv2.BORDER_REPLICATE`.
  - `reflect`: Same as `cv2.BORDER_REFLECT`.
  - `reflect_101`: Same as `cv2.BORDER_REFLECT_101`.
  - `wrap`: Same as `cv2.BORDER_WRAP`.
  - `constant`: Same as `cv2.BORDER_CONSTANT`.

  The datatype of the parameter may be:
  - If a single `int`, then it must be one of the `cv2.BORDER_*` constants.
  - If a single string, then it must be one of: `replicate`, `reflect`, `reflect_101`, `wrap`, `constant`.
  - If a list of `int`/`str`, then per image a random mode will be picked from that list.
  - If `imgaug.ALL`, then a random mode from all possible modes will be picked.
  - If `StochasticParameter`, then the mode will be sampled from that parameter per image, i.e. it must return only the above mentioned strings.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.AffineCv2(scale=2.0)
```

Zoom in on all images by a factor of `2`.

```
>>> aug = iaa.AffineCv2(translate_px=16)
```

Translate all images on the x- and y-axis by 16 pixels (towards the bottom right) and fill up any new pixels with zero (black values).

```
>>> aug = iaa.AffineCv2(translate_percent=0.1)
```

Translate all images on the x- and y-axis by `10` percent of their width/height (towards the bottom right). The pixel values are computed per axis based on that axis' size. Fill up any new pixels with zero (black values).

```
>>> aug = iaa.AffineCv2(rotate=35)
```

Rotate all images by `35` *degrees*. Fill up any new pixels with zero (black values).

```
>>> aug = iaa.AffineCv2(shear=15)
```

Shear all images by `15` *degrees*. Fill up any new pixels with zero (black values).

```
>>> aug = iaa.AffineCv2(translate_px=(-16, 16))
```

Translate all images on the x- and y-axis by a random value between `-16` and `16` pixels (to the bottom right) and fill up any new pixels with zero (black values). The translation value is sampled once per image and is the same for both axis.

```
>>> aug = iaa.AffineCv2(translate_px={"x": (-16, 16), "y": (-4, 4)})
```

Translate all images on the x-axis by a random value between `-16` and `16` pixels (to the right) and on the y-axis by a random value between `-4` and `4` pixels to the bottom. The sampling happens independently per axis, so even if both intervals were identical, the sampled axis-wise values would likely be different. This also fills up any new pixels with zero (black values).

```
>>> aug = iaa.AffineCv2(scale=2.0, order=[0, 1])
```

Same as in the above *scale* example, but uses (randomly) either nearest neighbour interpolation or linear interpolation. If *order* is not specified, `order=1` would be used by default.

```
>>> aug = iaa.AffineCv2(translate_px=16, cval=(0, 255))
```

Same as in the *translate_px* example above, but newly created pixels are now filled with a random color (sampled once per image and the same for all newly created pixels within that image).

```
>>> aug = iaa.AffineCv2(translate_px=16, mode=["constant", "replicate"])
```

Similar to the previous example, but the newly created pixels are filled with black pixels in half of all images (mode `constant` with default *cval* being `0`) and in the other half of all images using `replicate` mode, which repeats the color of the spatially closest pixel of the corresponding image edge.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
`remove_augmenters_inplace`(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
`reseed`(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
`show_grid`(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
`to_deterministic`(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.geometric.**ElasticTransformation**(*alpha=0*, *sigma=0*, *order=3*, *cval=0*, *mode='constant'*, *polygon_recoverer='auto'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Transform images by moving pixels locally around using displacement fields.

The augmenter has the parameters `alpha` and `sigma`. `alpha` controls the strength of the displacement: higher values mean that pixels are moved further. `sigma` controls the smoothness of the displacement: higher values lead to smoother patterns – as if the image was below water – while low values will cause indivdual pixels to be moved very differently from their neighbours, leading to noisy and pixelated images.

A relation of 10:1 seems to be good for `alpha` and `sigma`, e.g. `alpha=10` and `sigma=1` or `alpha=50`, `sigma=5`. For `128x128` a setting of `alpha=(0, 70.0)`, `sigma=(4.0, 6.0)` may be a good choice and will lead to a water-like effect.

Code here was initially inspired by https://gist.github.com/chsasank/4d8f68caf01f041a6453e67fb30f8f5a

For a detailed explanation, see

```
Simard, Steinkraus and Platt
Best Practices for Convolutional Neural Networks applied to Visual
Document Analysis
in Proc. of the International Conference on Document Analysis and
Recognition, 2003
```

Note: For coordinate-based inputs (keypoints, bounding boxes, polygons, ...), this augmenter still has to perform an image-based augmentation, which will make it significantly slower for such inputs than other augmenters. See *Performance*.

dtype support:

```
* ``uint8``: yes; fully tested (1)
* ``uint16``: yes; tested (1)
* ``uint32``: yes; tested (2)
* ``uint64``: limited; tested (3)
* ``int8``: yes; tested (1) (4) (5)
* ``int16``: yes; tested (4) (6)
* ``int32``: yes; tested (4) (6)
* ``int64``: limited; tested (3)
* ``float16``: yes; tested (1)
* ``float32``: yes; tested (1)
* ``float64``: yes; tested (1)
* ``float128``: no
* ``bool``: yes; tested (1) (7)

- (1) Always handled by ``cv2``.
- (2) Always handled by ``scipy``.
- (3) Only supported for ``order != 0``. Will fail for ``order=0``.
- (4) Mapped internally to ``float64`` when ``order=1``.
- (5) Mapped internally to ``int16`` when ``order>=2``.
```

(continues on next page)

(continued from previous page)

```
- (6) Handled by ``cv2`` when ``order=0`` or ``order=1``, otherwise by
      ``scipy``.
- (7) Mapped internally to ``float32``.
```

Parameters

- **alpha** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Strength of the distortion field. Higher values mean that pixels are moved further with respect to the distortion field's direction. Set this to around 10 times the value of *sigma* for visible effects.
  - If number, then that value will be used for all images.
  - If tuple `(a, b)`, then a random value will be uniformly sampled per image from the interval `[a, b]`.
  - If a list, then for each image a random value will be sampled from that list.
  - If `StochasticParameter`, then that parameter will be used to sample a value per image.
- **sigma** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Standard deviation of the gaussian kernel used to smooth the distortion fields. Higher values (for `128x128` images around 5.0) lead to more water-like effects, while lower values (for `128x128` images around `1.0` and lower) lead to more noisy, pixelated images. Set this to around 1/10th of *alpha* for visible effects.
  - If number, then that value will be used for all images.
  - If tuple `(a, b)`, then a random value will be uniformly sampled per image from the interval `[a, b]`.
  - If a list, then for each image a random value will be sampled from that list.
  - If `StochasticParameter`, then that parameter will be used to sample a value per image.
- **order** (*int or list of int or imaug.ALL or imgaug.parameters.StochasticParameter, optional*) – Interpolation order to use. Same meaning as in `scipy.ndimage.map_coordinates()` and may take any integer value in the range `0` to `5`, where orders close to `0` are faster.
  - If a single int, then that order will be used for all images.
  - If a tuple `(a, b)`, then a random value will be uniformly sampled per image from the interval `[a, b]`.
  - If a list, then for each image a random value will be sampled from that list.
  - If `imgaug.ALL`, then equivalant to list `[0, 1, 2, 3, 4, 5]`.
  - If `StochasticParameter`, then that parameter is queried per image to sample the order value to use.
- **cval** (*number or tuple of number or list of number or imgaug.ALL or imgaug.parameters.StochasticParameter, optional*) – The constant intensity value used to fill in new pixels. This value is only used if *mode* is set to `constant`. For standard `uint8` images (value range `0` to `255`), this value may also should also be in the range `0` to `255`. It may be a `float` value, even for images with integer dtypes.

  - If this is a single number, then that value will be used (e.g. `0` results in black pixels).
  - If a tuple `(a, b)`, then a random value will be uniformly sampled per image from the interval `[a, b]`.
  - If a list, then a random value will be picked from that list per image.
  - If `imgaug.ALL`, a value from the discrete range `[0..255]` will be sampled per image.
  - If a `StochasticParameter`, a new value will be sampled from the parameter per image.

- **mode** (*str or list of str or imgaug.ALL or imgaug.parameters.StochasticParameter, optional*) – Parameter that defines the handling of newly created pixels. May take the same values as in `scipy.ndimage.map_coordinates()`, i.e. `constant`, `nearest`, `reflect` or `wrap`.
  - If a single string, then that mode will be used for all images.
  - If a list of strings, then per image a random mode will be picked from that list.
  - If `imgaug.ALL`, then a random mode from all possible modes will be picked.
  - If `StochasticParameter`, then the mode will be sampled from that parameter per image, i.e. it must return only the above mentioned strings.
- **polygon_recoverer** (*'auto' or None or imgaug.augmentables.polygons._ConcavePolygonRecoverer, optional*) – The class to use to repair invalid polygons. If `"auto"`, a new instance of :class'imgaug.augmentables.polygons._ConcavePolygonRecoverer' will be created. If `None`, no polygon recoverer will be used. If an object, then that object will be used and must provide a `recover_from()` method, similar to `imgaug.augmentables.polygons._ConcavePolygonRecoverer`.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.ElasticTransformation(alpha=50.0, sigma=5.0)
```

Apply elastic transformations with a strength/alpha of `50.0` and smoothness of `5.0` to all images.

```
>>> aug = iaa.ElasticTransformation(alpha=(0.0, 70.0), sigma=5.0)
```

Apply elastic transformations with a strength/alpha that comes from the interval `[0.0, 70.0]` (randomly picked per image) and with a smoothness of `5.0`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
`remove_augmenters_inplace`(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
`reseed`(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
`show_grid`(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
`to_deterministic`(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters	

KEYPOINT_AUG_ALPHA_THRESH = 0.05

KEYPOINT_AUG_SIGMA_THRESH = 1.0

NB_NEIGHBOURING_KEYPOINTS = 3

NEIGHBOURING_KEYPOINTS_DISTANCE = 1.0

get_parameters(*self*)

class imgaug.augmenters.geometric.**PerspectiveTransform**(*scale=0*, *cval=0*, *mode='constant'*, *keep_size=True*, *polygon_recoverer='auto'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Apply random four point perspective transformations to images.

Each of the four points is placed on the image using a random distance from its respective corner. The distance is sampled from a normal distribution. As a result, most transformations don't change the image very much, while some "focus" on polygons far inside the image.

The results of this augmenter have some similarity with `Crop`.

Code partially from http://www.pyimagesearch.com/2014/08/25/4-point-opencv-getperspective-transform-example/

dtype support:

```
if (keep_size=False)::

    * ``uint8``: yes; fully tested
    * ``uint16``: yes; tested
    * ``uint32``: no (1)
    * ``uint64``: no (2)
    * ``int8``: yes; tested (3)
    * ``int16``: yes; tested
    * ``int32``: no (2)
    * ``int64``: no (2)
    * ``float16``: yes; tested (4)
    * ``float32``: yes; tested
    * ``float64``: yes; tested
    * ``float128``: no (1)
    * ``bool``: yes; tested (4)

    - (1) rejected by opencv
    - (2) leads to opencv error: cv2.error: ``OpenCV(3.4.4)
          (...)imgwarp.cpp:1805: error: (-215:Assertion failed)
          ifunc != 0 in function 'remap'``.
    - (3) mapped internally to ``int16``.
    - (4) mapped intenally to ``float32``.

if (keep_size=True)::

    minimum of (
        ``imgaug.augmenters.geometric.PerspectiveTransform(keep_size=False)``,
        :func:`imgaug.imgaug.imresize_many_images`
    )
```

Parameters

- **scale** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Standard deviation of the normal distributions. These are used to sample the random distances of the subimage's corners from the full image's corners. The sampled values reflect percentage values (with respect to image height/width). Recommended values are in the range `0.0` to `0.1`.
  - If a single number, then that value will always be used as the scale.
  - If a tuple `(a, b)` of numbers, then a random value will be uniformly sampled per image from the interval `(a, b)`.
  - If a list of values, a random value will be picked from the list per image.
  - If a `StochasticParameter`, then that parameter will be queried to draw one value per image.
- **keep_size** (*bool, optional*) – Whether to resize image's back to their original size after applying the perspective transform. If set to `False`, the resulting images may end up having different shapes and will always be a list, never an array.
- **cval** (*number or tuple of number or list of number or imaug.ALL or imgaug.parameters.StochasticParameter, optional*) – The constant value used to fill up pixels in the result image that didn't exist in the input image (e.g. when translating to the left, some new pixels are created at the right). Such a fill-up with a constant value only happens, when *mode* is `constant`. The expected value range is `[0, 255]` for `uint8` images. It may be a float value.
  - If this is a single int or float, then that value will be used (e.g. 0 results in black pixels).
  - If a tuple `(a, b)`, then a random value is uniformly sampled per image from the interval `[a, b]`.
  - If a list, then a random value will be sampled from that list per image.
  - If `imgaug.ALL`, then equivalent to tuple `(0, 255)`.
  - If a `StochasticParameter`, a new value will be sampled from the parameter per image.
- **mode** (*int or str or list of str or list of int or imgaug.ALL or imgaug.parameters.StochasticParameter, optional*) – Parameter that defines the handling of newly created pixels. Same meaning as in OpenCV's border mode. Let `abcdefgh` be an image's content and `|` be an image boundary, then:
  - `cv2.BORDER_REPLICATE`: `aaaaaa|abcdefgh|hhhhhhh`
  - `cv2.BORDER_CONSTANT`: `iiiiii|abcdefgh|iiiiiii`, where `i` is the defined cval.
  - `replicate`: Same as `cv2.BORDER_REPLICATE`.
  - `constant`: Same as `cv2.BORDER_CONSTANT`.

  The datatype of the parameter may be:
  - If a single `int`, then it must be one of `cv2.BORDER_*`.
  - If a single string, then it must be one of: `replicate`, `reflect`, `reflect_101`, `wrap`, `constant`.
  - If a list of ints/strings, then per image a random mode will be picked from that list.
  - If `imgaug.ALL`, then a random mode from all possible modes will be picked per image.

  - If `StochasticParameter`, then the mode will be sampled from that parameter per image, i.e. it must return only the above mentioned strings.

- **polygon_recoverer** (*'auto' or None or imgaug.augmentables.polygons._ConcavePolygonRecoverer, optional*) – The class to use to repair invalid polygons. If `"auto"`, a new instance of :class'imgaug.augmentables.polygons._ConcavePolygonRecoverer' will be created. If `None`, no polygon recoverer will be used. If an object, then that object will be used and must provide a `recover_from()` method, similar to `imgaug.augmentables.polygons._ConcavePolygonRecoverer`.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.PerspectiveTransform(scale=(0.01, 0.15))
```

Apply perspective transformations using a random scale between `0.01` and `0.15` per image, where the scale is roughly a measure of how far the perspective transformation's corner points may be distanced from the image's corner points. Higher scale values lead to stronger "zoom-in" effects (and thereby stronger distortions).

```
>>> aug = iaa.PerspectiveTransform(scale=(0.01, 0.15), keep_size=False)
```

Same as in the previous example, but images are not resized back to the input image size after augmentation. This will lead to smaller output images.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.

Continued on next page

Table 121 – continued from previous page

augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.geometric.**PiecewiseAffine**(*scale=0*, *nb_rows=4*, *nb_cols=4*, *order=1*, *cval=0*, *mode='constant'*, *absolute_scale=False*, *polygon_recoverer=None*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Apply affine transformations that differ between local neighbourhoods.

This augmenter places a regular grid of points on an image and randomly moves the neighbourhood of these point around via affine transformations. This leads to local distortions.

This is mostly a wrapper around scikit-image's PiecewiseAffine. See also Affine for a similar technique.

Note: This augmenter is very slow. See *Performance*. Try to use ElasticTransformation instead,

which is at least 10x faster.

Note: For coordinate-based inputs (keypoints, bounding boxes, polygons, ...), this augmenter still has to perform an image-based augmentation, which will make it significantly slower for such inputs than other augmenters. See *Performance*.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; tested (1)
* ``uint32``: yes; tested (1) (2)
* ``uint64``: no (3)
* ``int8``: yes; tested (1)
* ``int16``: yes; tested (1)
* ``int32``: yes; tested (1) (2)
* ``int64``: no (3)
* ``float16``: yes; tested (1)
* ``float32``: yes; tested (1)
* ``float64``: yes; tested (1)
* ``float128``: no (3)
* ``bool``: yes; tested (1) (4)

- (1) Only tested with `order` set to ``0``.
- (2) scikit-image converts internally to ``float64``, which might
      introduce inaccuracies. Tests showed that these inaccuracies
      seemed to not be an issue.
- (3) Results too inaccurate.
- (4) Mapped internally to ``float64``.
```

Parameters

- **scale** (*float or tuple of float or imgaug.parameters.StochasticParameter, optional*) – Each point on the regular grid is moved around via a normal distribution. This scale factor is equivalent to the normal distribution's sigma. Note that the jitter (how far each point is moved in which direction) is multiplied by the height/width of the image if `absolute_scale=False` (default), so this scale can be the same for different sized images. Recommended values are in the range `0.01` to `0.05` (weak to strong augmentations).
  - If a single `float`, then that value will always be used as the scale.
  - If a tuple `(a, b)` of `float` s, then a random value will be uniformly sampled per image from the interval `[a, b]`.
  - If a list, then a random value will be picked from that list per image.
  - If a `StochasticParameter`, then that parameter will be queried to draw one value per image.
- **nb_rows** (*int or tuple of int or imgaug.parameters.StochasticParameter, optional*) – Number of rows of points that the regular grid should have. Must be at least `2`. For large images, you might want to pick a higher value than `4`. You might have to then adjust scale to lower values.
  - If a single `int`, then that value will always be used as the number of rows.

  - If a tuple `(a, b)`, then a value from the discrete interval `[a..b]` will be uniformly sampled per image.
  - If a list, then a random value will be picked from that list per image.
  - If a StochasticParameter, then that parameter will be queried to draw one value per image.
- **nb_cols** (*int or tuple of int or imgaug.parameters.StochasticParameter, optional*) – Number of columns. Analogous to *nb_rows*.
- **order** (*int or list of int or imgaug.ALL or imgaug.parameters.StochasticParameter, optional*) – See `imgaug.augmenters.geometric.Affine.__init__()`.
- **cval** (*int or float or tuple of float or imgaug.ALL or imgaug.parameters.StochasticParameter, optional*) – See `imgaug.augmenters.geometric.Affine.__init__()`.
- **mode** (*str or list of str or imgaug.ALL or imgaug.parameters.StochasticParameter, optional*) – See `imgaug.augmenters.geometric.Affine.__init__()`.
- **absolute_scale** (*bool, optional*) – Take *scale* as an absolute value rather than a relative value.
- **polygon_recoverer** (*'auto' or None or imgaug.augmentables.polygons._ConcavePolygonRecoverer, optional*) – The class to use to repair invalid polygons. If `"auto"`, a new instance of :class'imgaug.augmentables.polygons._ConcavePolygonRecoverer' will be created. If `None`, no polygon recoverer will be used. If an object, then that object will be used and must provide a `recover_from()` method, similar to `imgaug.augmentables.polygons._ConcavePolygonRecoverer`.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.PiecewiseAffine(scale=(0.01, 0.05))
```

Place a regular grid of points on each image and then randomly move each point around by `1` to `5` percent (with respect to the image height/width). Pixels between these points will be moved accordingly.

```
>>> aug = iaa.PiecewiseAffine(scale=(0.01, 0.05), nb_rows=8, nb_cols=8)
```

Same as the previous example, but uses a denser grid of `8x8` points (default is `4x4`). This can be useful for large images.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.

Continued on next page

Table 122 – continued from previous page

augment(self[, return_batch, hooks])	Augment a batch.
augment_batch(self, batch[, hooks])	Augment a single batch.
augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ...[, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.geometric.**Rot90**(*k*, *keep_size=True*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Rotate images clockwise by multiples of 90 degrees.

This could also be achieved using `Affine`, but `Rot90` is significantly more efficient.

dtype support:

```
if (keep_size=False)::

    * ``uint8``: yes; fully tested
    * ``uint16``: yes; tested
    * ``uint32``: yes; tested
    * ``uint64``: yes; tested
    * ``int8``: yes; tested
    * ``int16``: yes; tested
    * ``int32``: yes; tested
    * ``int64``: yes; tested
    * ``float16``: yes; tested
    * ``float32``: yes; tested
    * ``float64``: yes; tested
    * ``float128``: yes; tested
    * ``bool``: yes; tested

if (keep_size=True)::

    minimum of (
        ``imgaug.augmenters.geometric.Rot90(keep_size=False)``,
        :func:`imgaug.imgaug.imresize_many_images`
    )
```

Parameters

- **k** (*int or list of int or tuple of int or imaug.ALL or imgaug.parameters.StochasticParameter, optional*) –

  How often to rotate clockwise by 90 degrees.

  - If a single `int`, then that value will be used for all images.
  - If a tuple `(a, b)`, then a random value will be uniformly sampled per image from the discrete interval `[a..b]`.
  - If a list, then for each image a random value will be sampled from that list.
  - If `imgaug.ALL`, then equivalant to list `[0, 1, 2, 3]`.
  - If `StochasticParameter`, then that parameter is queried per image to sample the value to use.

- **keep_size** (*bool, optional*) – After rotation by an odd-valued *k* (e.g. 1 or 3), the resulting image may have a different height/width than the original image. If this parameter is set to `True`, then the rotated image will be resized to the input image's size. Note that this might also cause the augmented image to look distorted.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Rot90(1)
```

Rotate all images by 90 degrees. Resize these images afterwards to keep the size that they had before augmentation. This may cause the images to look distorted.

```
>>> aug = iaa.Rot90([1, 3])
```

Rotate all images by 90 or 270 degrees. Resize these images afterwards to keep the size that they had before augmentation. This may cause the images to look distorted.

```
>>> aug = iaa.Rot90((1, 3))
```

Rotate all images by 90, 180 or 270 degrees. Resize these images afterwards to keep the size that they had before augmentation. This may cause the images to look distorted.

```
>>> aug = iaa.Rot90((1, 3), keep_size=False)
```

Rotate all images by 90, 180 or 270 degrees. Does not resize to the original image size afterwards, i.e. each image's size may change.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.

Continued on next page

Table 123 – continued from previous page

copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters (*self*)

13.26 imgaug.augmenters.pooling

Augmenters that apply pooling operations to images.

Do not import directly from this file, as the categorization is not final. Use instead

```
from imgaug import augmenters as iaa
```

and then e.g.

```
seq = iaa.Sequential([
    iaa.AveragePooling((1, 3))
])
```

List of augmenters:

- AveragePooling
- MaxPooling
- MinPooling

- MedianPooling

class imgaug.augmenters.pooling.**AveragePooling**(*kernel_size*, *keep_size=True*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: imgaug.augmenters.pooling._AbstractPoolingBase

Apply average pooling to images.

This augmenter pools images with kernel sizes `H x W` by averaging the pixel values within these windows. For e.g. `2 x 2` this halves the image size. Optionally, the augmenter will automatically re-upscale the image to the input size (by default this is activated).

Note that this augmenter is very similar to `AverageBlur`. `AverageBlur` applies averaging within windows of given kernel size *without* striding, while `AveragePooling` applies striding corresponding to the kernel size, with optional upscaling afterwards. The upscaling is configured to create "pixelated"/"blocky" images by default.

Note: During heatmap or segmentation map augmentation, the respective arrays are not changed, only the shapes of the underlying images are updated. This is because imgaug can handle maps/maks that are larger/smaller than their corresponding image.

---

dtype support:

```
See :func:`imgaug.imgaug.avg_pool`.
```

Variables

- **kernel_size** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter or tuple of tuple of int or tuple of list of int or tuple of imgaug.parameters.StochasticParameter, optional*) – The kernel size of the pooling operation.
  - If an int, then that value will be used for all images for both kernel height and width.
  - If a tuple `(a, b)`, then a value from the discrete range `[a..b]` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image and used for both kernel height and width.
  - If a StochasticParameter, then a value will be sampled per image from that parameter per image and used for both kernel height and width.
  - If a tuple of tuple of int given as `((a, b), (c, d))`, then two values will be sampled independently from the discrete ranges `[a..b]` and `[c..d]` per image and used as the kernel height and width.
  - If a tuple of lists of int, then two values will be sampled independently per image, one from the first list and one from the second, and used as the kernel height and width.
  - If a tuple of StochasticParameter, then two values will be sampled indepdently per image, one from the first parameter and one from the second, and used as the kernel height and width.
- **keep_size** (*bool, optional*) – After pooling, the result image will usually have a different height/width compared to the original input image. If this parameter is set to True, then the pooled image will be resized to the input image's size, i.e. the augmenter's output shape is always identical to the input shape.

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.AveragePooling(2)
```

Create an augmenter that always pools with a kernel size of `2 x 2`.

```
>>> aug = iaa.AveragePooling(2, keep_size=False)
```

Create an augmenter that always pools with a kernel size of `2 x 2` and does *not* resize back to the input image size, i.e. the resulting images have half the resolution.

```
>>> aug = iaa.AveragePooling([2, 8])
```

Create an augmenter that always pools either with a kernel size of `2 x 2` or `8 x 8`.

```
>>> aug = iaa.AveragePooling((1, 7))
```

Create an augmenter that always pools with a kernel size of `1 x 1` (does nothing) to `7 x 7`. The kernel sizes are always symmetric.

```
>>> aug = iaa.AveragePooling(((1, 7), (1, 7)))
```

Create an augmenter that always pools with a kernel size of `H x W` where `H` and `W` are both sampled independently from the range `[1..7]`. E.g. resulting kernel sizes could be `3 x 7` or `5 x 1`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.

Continued on next page

Table 124 – continued from previous page

augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.pooling.**MaxPooling**(*kernel_size*, *keep_size=True*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: imgaug.augmenters.pooling._AbstractPoolingBase

Apply max pooling to images.

This augmenter pools images with kernel sizes `H x W` by taking the maximum pixel value over windows. For e.g. `2 x 2` this halves the image size. Optionally, the augmenter will automatically re-upscale the image to the input size (by default this is activated).

The maximum within each pixel window is always taken channelwise..

Note: During heatmap or segmentation map augmentation, the respective arrays are not changed, only the shapes of the underlying images are updated. This is because imgaug can handle maps/maks that are larger/smaller than their corresponding image.

dtype support:

```
See :func:`imgaug.imgaug.max_pool`.
```

Variables

- **kernel_size** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter or tuple of tuple of int or tuple of list of int or tuple of imgaug.parameters.StochasticParameter, optional*) – The kernel size of the pooling operation.
  - If an int, then that value will be used for all images for both kernel height and width.
  - If a tuple `(a, b)`, then a value from the discrete range `[a..b]` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image and used for both kernel height and width.
  - If a StochasticParameter, then a value will be sampled per image from that parameter per image and used for both kernel height and width.
  - If a tuple of tuple of int given as `((a, b), (c, d))`, then two values will be sampled independently from the discrete ranges `[a..b]` and `[c..d]` per image and used as the kernel height and width.
  - If a tuple of lists of int, then two values will be sampled independently per image, one from the first list and one from the second, and used as the kernel height and width.
  - If a tuple of StochasticParameter, then two values will be sampled indepdently per image, one from the first parameter and one from the second, and used as the kernel height and width.
- **keep_size** (*bool, optional*) – After pooling, the result image will usually have a different height/width compared to the original input image. If this parameter is set to True, then the pooled image will be resized to the input image's size, i.e. the augmenter's output shape is always identical to the input shape.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.MaxPooling(2)
```

Create an augmenter that always pools with a kernel size of `2 x 2`.

```
>>> aug = iaa.MaxPooling(2, keep_size=False)
```

Create an augmenter that always pools with a kernel size of `2 x 2` and does *not* resize back to the input image size, i.e. the resulting images have half the resolution.

```
>>> aug = iaa.MaxPooling([2, 8])
```

Create an augmenter that always pools either with a kernel size of `2 x 2` or `8 x 8`.

```
>>> aug = iaa.MaxPooling((1, 7))
```

Create an augmenter that always pools with a kernel size of `1 x 1` (does nothing) to `7 x 7`. The kernel sizes are always symmetric.

```
>>> aug = iaa.MaxPooling(((1, 7), (1, 7)))
```

Create an augmenter that always pools with a kernel size of `H x W` where `H` and `W` are both sampled independently from the range `[1..7]`. E.g. resulting kernel sizes could be `3 x 7` or `5 x 1`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.

Continued on next page

Table 125 – continued from previous page

localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.pooling.**MedianPooling**(*kernel_size*, *keep_size=True*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: imgaug.augmenters.pooling._AbstractPoolingBase

Apply median pooling to images.

This augmenter pools images with kernel sizes `H x W` by taking the median pixel value over windows. For e.g. `2 x 2` this halves the image size. Optionally, the augmenter will automatically re-upscale the image to the input size (by default this is activated).

The median within each pixel window is always taken channelwise.

Note: During heatmap or segmentation map augmentation, the respective arrays are not changed, only the shapes of the underlying images are updated. This is because imgaug can handle maps/maks that are larger/smaller than their corresponding image.

dtype support:

```
See :func:`imgaug.imgaug.pool`.
```

Variables

- **kernel_size** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter or tuple of tuple of int or tuple of list of int or tuple of imgaug.parameters.StochasticParameter, optional*) – The kernel size of the pooling operation.
  - If an int, then that value will be used for all images for both kernel height and width.
  - If a tuple `(a, b)`, then a value from the discrete range `[a..b]` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image and used for both kernel height and width.
  - If a StochasticParameter, then a value will be sampled per image from that parameter per image and used for both kernel height and width.

  - If a tuple of tuple of int given as `((a, b), (c, d))`, then two values will be sampled independently from the discrete ranges `[a..b]` and `[c..d]` per image and used as the kernel height and width.
  - If a tuple of lists of int, then two values will be sampled independently per image, one from the first list and one from the second, and used as the kernel height and width.
  - If a tuple of StochasticParameter, then two values will be sampled indepdently per image, one from the first parameter and one from the second, and used as the kernel height and width.

- **keep_size** (*bool, optional*) – After pooling, the result image will usually have a different height/width compared to the original input image. If this parameter is set to True, then the pooled image will be resized to the input image's size, i.e. the augmenter's output shape is always identical to the input shape.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.MedianPooling(2)
```

Create an augmenter that always pools with a kernel size of `2 x 2`.

```
>>> aug = iaa.MedianPooling(2, keep_size=False)
```

Create an augmenter that always pools with a kernel size of `2 x 2` and does *not* resize back to the input image size, i.e. the resulting images have half the resolution.

```
>>> aug = iaa.MedianPooling([2, 8])
```

Create an augmenter that always pools either with a kernel size of `2 x 2` or `8 x 8`.

```
>>> aug = iaa.MedianPooling((1, 7))
```

Create an augmenter that always pools with a kernel size of `1 x 1` (does nothing) to `7 x 7`. The kernel sizes are always symmetric.

```
>>> aug = iaa.MedianPooling(((1, 7), (1, 7)))
```

Create an augmenter that always pools with a kernel size of `H x W` where `H` and `W` are both sampled independently from the range `[1..7]`. E.g. resulting kernel sizes could be `3 x 7` or `5 x 1`.

Methods

__call__(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
augment(self[, return_batch, hooks])	Augment a batch.
augment_batch(self, batch[, hooks])	Augment a single batch.
augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ...[, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters	

class imgaug.augmenters.pooling.**MinPooling**(*kernel_size*, *keep_size=True*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: imgaug.augmenters.pooling._AbstractPoolingBase

Apply minimum pooling to images.

This augmenter pools images with kernel sizes `H x W` by taking the minimum pixel value over windows. For e.g. `2 x 2` this halves the image size. Optionally, the augmenter will automatically re-upscale the image to the input size (by default this is activated).

The minimum within each pixel window is always taken channelwise.

Note: During heatmap or segmentation map augmentation, the respective arrays are not changed, only the shapes of the underlying images are updated. This is because imgaug can handle maps/maks that are larger/smaller than their corresponding image.

---

dtype support:

```
See :func:`imgaug.imgaug.pool`.
```

Variables

- **kernel_size** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter or tuple of tuple of int or tuple of list of int or tuple of imgaug.parameters.StochasticParameter, optional*) – The kernel size of the pooling operation.
  - If an int, then that value will be used for all images for both kernel height and width.
  - If a tuple `(a, b)`, then a value from the discrete range `[a..b]` will be sampled per image.
  - If a list, then a random value will be sampled from that list per image and used for both kernel height and width.
  - If a StochasticParameter, then a value will be sampled per image from that parameter per image and used for both kernel height and width.
  - If a tuple of tuple of int given as `((a, b), (c, d))`, then two values will be sampled independently from the discrete ranges `[a..b]` and `[c..d]` per image and used as the kernel height and width.
  - If a tuple of lists of int, then two values will be sampled independently per image, one from the first list and one from the second, and used as the kernel height and width.
  - If a tuple of StochasticParameter, then two values will be sampled indepdently per image, one from the first parameter and one from the second, and used as the kernel height and width.
- **keep_size** (*bool, optional*) – After pooling, the result image will usually have a different height/width compared to the original input image. If this parameter is set to True, then the pooled image will be resized to the input image's size, i.e. the augmenter's output shape is always identical to the input shape.
- **name** (*None or str, optional*) – See imgaug.augmenters.meta.Augmenter.__init__().
- **deterministic** (*bool, optional*) – See imgaug.augmenters.meta.Augmenter.__init__().

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.MinPooling(2)
```

Create an augmenter that always pools with a kernel size of `2 x 2`.

```
>>> aug = iaa.MinPooling(2, keep_size=False)
```

Create an augmenter that always pools with a kernel size of `2 x 2` and does *not* resize back to the input image size, i.e. the resulting images have half the resolution.

```
>>> aug = iaa.MinPooling([2, 8])
```

Create an augmenter that always pools either with a kernel size of `2 x 2` or `8 x 8`.

```
>>> aug = iaa.MinPooling((1, 7))
```

Create an augmenter that always pools with a kernel size of `1 x 1` (does nothing) to `7 x 7`. The kernel sizes are always symmetric.

```
>>> aug = iaa.MinPooling(((1, 7), (1, 7)))
```

Create an augmenter that always pools with a kernel size of `H x W` where `H` and `W` are both sampled independently from the range `[1..7]`. E.g. resulting kernel sizes could be `3 x 7` or `5 x 1`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.

Continued on next page

Table 127 – continued from previous page

copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters	

13.27 imgaug.augmenters.segmentation

Augmenters that apply changes to images based on segmentation methods.

Do not import directly from this file, as the categorization is not final. Use instead

```
from imgaug import augmenters as iaa
```

and then e.g.

```
seq = iaa.Sequential([
    iaa.Superpixels(...)
])
```

List of augmenters:

- Superpixels
- Voronoi

- UniformVoronoi
- RegularGridVoronoi
- RelativeRegularGridVoronoi

class imgaug.augmenters.segmentation.**DropoutPointsSampler**(*other_points_sampler*, *p_drop*)

Bases: *imgaug.augmenters.segmentation.IPointsSampler*

Remove a defined fraction of sampled points.

Parameters

- **other_points_sampler** (*IPointsSampler*) – Another point sampler that is queried to generate a list of points. The dropout operation will be applied to that list.
- **p_drop** (*number or tuple of number or imgaug.parameters.StochasticParameter*) – The probability that a coordinate will be removed from the list of all sampled coordinates. A value of `1.0` would mean that (on average) `100` percent of all coordinates will be dropped, while `0.0` denotes `0` percent. Note that this sampler will always ensure that at least one coordinate is left after the dropout operation, i.e. even `1.0` will only drop all *except one* coordinate.
  - If a `float`, then that value will be used for all images.
  - If a `tuple (a, b)`, then a value `p` will be sampled from the interval `[a, b]` per image.
  - If a `StochasticParameter`, then this parameter will be used to determine per coordinate whether it should be *kept* (sampled value of `>0.5`) or shouldn't be kept (sampled value of `<=0.5`). If you instead want to provide the probability as a stochastic parameter, you can usually do `imgaug.parameters.Binomial(1-p)` to convert parameter *p* to a 0/1 representation.

Examples

```
>>> import imgaug.augmenters as iaa
>>> sampler = iaa.DropoutPointsSampler(
>>>     iaa.RegularGridPointsSampler(10, 20),
>>>     0.2)
```

Create a point sampler that first generates points following a regular grid of `10` rows and `20` columns, then randomly drops `20` percent of these points.

Methods

sample_points(self, images, random_state)	Generate coordinates of points on images.

sample_points(*self*, *images*, *random_state*)

Generate coordinates of points on images.

Parameters

- **images** (*ndarray or list of ndarray*) – One or more images for which to generate points. If this is a `list` of arrays, each one of them is expected to have three dimensions. If this is an array, it must be four-dimensional and the first axis is expected to denote the image index. For `RGB` images the array would hence have to be of shape `(N, H, W, 3)`.

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState*) – A random state to use for any probabilistic function required during the point sampling. See *imgaug.random.RNG()* for details.

Returns An `(N,2) float32` array containing `(x,y)` subpixel coordinates, all of which being within the intervals `[0.0, width]` and `[0.0, height]`.

Return type ndarray

class imgaug.augmenters.segmentation.**IPointsSampler**
Bases: `object`

Interface for all point samplers.

Point samplers return coordinate arrays of shape `Nx2`. These coordinates can be used in other augmenters, see e.g. *imgaug.augmenters.segmentation.Voronoi*.

Methods

sample_points(self, images, random_state)	Generate coordinates of points on images.

sample_points(*self*, *images*, *random_state*)
Generate coordinates of points on images.

Parameters

- **images** (*ndarray or list of ndarray*) – One or more images for which to generate points. If this is a `list` of arrays, each one of them is expected to have three dimensions. If this is an array, it must be four-dimensional and the first axis is expected to denote the image index. For `RGB` images the array would hence have to be of shape `(N, H, W, 3)`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState*) – A random state to use for any probabilistic function required during the point sampling. See *imgaug.random.RNG()* for details.

Returns An `(N,2) float32` array containing `(x,y)` subpixel coordinates, all of which being within the intervals `[0.0, width]` and `[0.0, height]`.

Return type ndarray

class imgaug.augmenters.segmentation.**RegularGridPointsSampler**(*n_rows*, *n_cols*)
Bases: *imgaug.augmenters.segmentation.IPointsSampler*

Sampler that generates a regular grid of coordinates on an image.

'Regular grid' here means that on each axis all coordinates have the same distance from each other. Note that the distance may change between axis.

Parameters

- **n_rows** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) – Number of rows of coordinates to place on each image, i.e. the number of coordinates on the y-axis. Note that for each image, the sampled value is clipped to the interval `[1..H]`, where `H` is the image height.
  - If a single `int`, then that value will always be used.
  - If a `tuple (a, b)`, then a value from the discrete interval `[a..b]` will be sampled per image.

  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, then that parameter will be queried to draw one value per image.

- **n_cols** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) – Number of columns of coordinates to place on each image, i.e. the number of coordinates on the x-axis. Note that for each image, the sampled value is clipped to the interval `[1..W]`, where `W` is the image width.
  - If a single `int`, then that value will always be used.
  - If a `tuple (a, b)`, then a value from the discrete interval `[a..b]` will be sampled per image.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, then that parameter will be queried to draw one value per image.

Examples

```
>>> import imgaug.augmenters as iaa
>>> sampler = iaa.RegularGridPointsSampler(
>>>     n_rows=(5, 20),
>>>     n_cols=50)
```

Create a point sampler that generates regular grids of points. These grids contain `r` points on the y-axis, where `r` is sampled uniformly from the discrete interval `[5..20]` per image. On the x-axis, the grids always contain `50` points.

Methods

sample_points(self, images, random_state)	Generate coordinates of points on images.

sample_points (*self*, *images*, *random_state*)
Generate coordinates of points on images.

Parameters

- **images** (*ndarray or list of ndarray*) – One or more images for which to generate points. If this is a `list` of arrays, each one of them is expected to have three dimensions. If this is an array, it must be four-dimensional and the first axis is expected to denote the image index. For `RGB` images the array would hence have to be of shape `(N, H, W, 3)`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState*) – A random state to use for any probabilistic function required during the point sampling. See *imgaug.random.RNG()* for details.

Returns An `(N,2) float32` array containing `(x,y)` subpixel coordinates, all of which being within the intervals `[0.0, width]` and `[0.0, height]`.

Return type ndarray

class imgaug.augmenters.segmentation.**RegularGridVoronoi**(*n_rows*, *n_cols*, *p_drop_points=0.4*, *p_replace=1.0*, *max_size=128*, *interpolation='linear'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.segmentation.Voronoi*

Sample Voronoi cells from regular grids and color-average them.

This augmenter is a shortcut for the combination of *imgaug.augmenters.segmentation.Voronoi*, *imgaug.augmenters.segmentation.RegularGridPointsSampler* and *imgaug.augmenters.segmentation.DropoutPointsSampler*. Hence, it generates a regular grid with `R` rows and `C` columns of coordinates on each image. Then, it drops `p` percent of the `R*C` coordinates to randomize the grid. Each image pixel then belongs to the voronoi cell with the closest coordinate.

dtype support:

```
See ``imgaug.augmenters.segmentation.Voronoi``.
```

Parameters

- **n_rows** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) – Number of rows of coordinates to place on each image, i.e. the number of coordinates on the y-axis. Note that for each image, the sampled value is clipped to the interval `[1..H]`, where `H` is the image height.
  - If a single `int`, then that value will always be used.
  - If a `tuple (a, b)`, then a value from the discrete interval `[a..b]` will be sampled per image.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, then that parameter will be queried to draw one value per image.
- **n_cols** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) – Number of columns of coordinates to place on each image, i.e. the number of coordinates on the x-axis. Note that for each image, the sampled value is clipped to the interval `[1..W]`, where `W` is the image width.
  - If a single `int`, then that value will always be used.
  - If a `tuple (a, b)`, then a value from the discrete interval `[a..b]` will be sampled per image.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, then that parameter will be queried to draw one value per image.
- **p_drop_points** (*number or tuple of number or imgaug.parameters.StochasticParameter, optional*) – The probability that a coordinate will be removed from the list of all sampled coordinates. A value of `1.0` would mean that (on average) `100` percent of all coordinates will be dropped, while `0.0` denotes `0` percent. Note that this sampler will always ensure that at least one coordinate is left after the dropout operation, i.e. even `1.0` will only drop all *except one* coordinate.

  - If a `float`, then that value will be used for all images.
  - If a `tuple (a, b)`, then a value `p` will be sampled from the interval `[a, b]` per image.
  - If a `StochasticParameter`, then this parameter will be used to determine per coordinate whether it should be *kept* (sampled value of `>0.5`) or shouldn't be kept (sampled value of `<=0.5`). If you instead want to provide the probability as a stochastic parameter, you can usually do `imgaug.parameters.Binomial(1-p)` to convert parameter *p* to a 0/1 representation.

- **p_replace** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Defines for any segment the probability that the pixels within that segment are replaced by their average color (otherwise, the pixels are not changed). Examples:
  - A probability of `0.0` would mean, that the pixels in no segment are replaced by their average color (image is not changed at all).
  - A probability of `0.5` would mean, that around half of all segments are replaced by their average color.
  - A probability of `1.0` would mean, that all segments are replaced by their average color (resulting in a voronoi image).

  Behaviour based on chosen datatypes for this parameter:
  - If a `number`, then that number will always be used.
  - If `tuple (a, b)`, then a random probability will be sampled from the interval `[a, b]` per image.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, it is expected to return values between `0.0` and `1.0` and will be queried *for each individual segment* to determine whether it is supposed to be averaged (`>0.5`) or not (`<=0.5`). Recommended to be some form of `Binomial(...)`.
- **max_size** (*int or None, optional*) – Maximum image size at which the augmentation is performed. If the width or height of an image exceeds this value, it will be downscaled before the augmentation so that the longest side matches *max_size*. This is done to speed up the process. The final output image has the same size as the input image. Note that in case *p_replace* is below `1.0`, the down-/upscaling will affect the not-replaced pixels too. Use `None` to apply no down-/upscaling.
- **interpolation** (*int or str, optional*) – Interpolation method to use during downscaling when *max_size* is exceeded. Valid methods are the same as in *imgaug.imgaug.imresize_single_image()*.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.RegularGridVoronoi(10, 20)
```

Place a regular grid of `10x20` (`height x width`) coordinates on each image. Randomly drop on average `20` percent of these points to create a less regular pattern. Then use the remaining coordinates to group the image pixels into voronoi cells and average the colors within them. The process is performed at an image size not exceeding `128` px on any side (default). If necessary, the downscaling is performed using `linear` interpolation (default).

```
>>> aug = iaa.RegularGridVoronoi(
>>>     (10, 30), 20, p_drop_points=0.0, p_replace=0.9, max_size=None)
```

Same as above, generates a grid with randomly `10` to `30` rows, drops none of the generates points, replaces only `90` percent of the voronoi cells with their average color (the pixels of the remaining `10` percent are not changed) and performs the transformation at the original image size (`max_size=None`).

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.

Continued on next page

Table 131 – continued from previous page

get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters	

class imgaug.augmenters.segmentation.**RelativeRegularGridPointsSampler**(*n_rows_frac*, *n_cols_frac*)

Bases: *imgaug.augmenters.segmentation.IPointsSampler*

Regular grid coordinate sampler; places more points on larger images.

This is similar to RegularGridPointsSampler, but the number of rows and columns is given as fractions of each image's height and width. Hence, more coordinates are generated for larger images.

Parameters

- **n_rows_frac** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Relative number of coordinates to place on the y-axis. For a value y and image height H the number of actually placed coordinates (i.e. computed rows) is given by int(round(y*H)). Note that for each image, the number of coordinates is clipped to the interval [1,H], where H is the image height.
  - If a single number, then that value will always be used.
  - If a tuple (a, b), then a value from the interval [a, b] will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a StochasticParameter, then that parameter will be queried to draw one value per image.
- **n_cols_frac** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Relative number of coordinates to place on the x-axis. For a value x and image height W the number of actually placed coordinates (i.e. computed columns) is given by int(round(x*W)). Note that for each image, the number of coordinates is clipped to the interval [1,W], where W is the image width.
  - If a single number, then that value will always be used.
  - If a tuple (a, b), then a value from the interval [a, b] will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.

– If a `StochasticParameter`, then that parameter will be queried to draw one value per image.

Examples

```
>>> import imgaug.augmenters as iaa
>>> sampler = iaa.RelativeRegularGridPointsSampler(
>>>     n_rows_frac=(0.01, 0.1),
>>>     n_cols_frac=0.2)
```

Create a point sampler that generates regular grids of points. These grids contain `round(y*H)` points on the y-axis, where `y` is sampled uniformly from the interval `[0.01, 0.1]` per image and `H` is the image height. On the x-axis, the grids always contain `0.2*W` points, where `W` is the image width.

Methods

sample_points(self, images, random_state)	Generate coordinates of points on images.

sample_points (*self*, *images*, *random_state*)
Generate coordinates of points on images.

Parameters

- **images** (*ndarray or list of ndarray*) – One or more images for which to generate points. If this is a `list` of arrays, each one of them is expected to have three dimensions. If this is an array, it must be four-dimensional and the first axis is expected to denote the image index. For `RGB` images the array would hence have to be of shape `(N, H, W, 3)`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState*) – A random state to use for any probabilistic function required during the point sampling. See *imgaug.random.RNG()* for details.

Returns An `(N,2) float32` array containing `(x,y)` subpixel coordinates, all of which being within the intervals `[0.0, width]` and `[0.0, height]`.

Return type ndarray

class imgaug.augmenters.segmentation.**RelativeRegularGridVoronoi**(*n_rows_frac*, *n_cols_frac*, *p_drop_points=0.4*, *p_replace=1.0*, *max_size=None*, *interpolation='linear'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.segmentation.Voronoi*

Sample Voronoi cells from image-dependent grids and color-average them.

This augmenter is a shortcut for the combination of *imgaug.augmenters.segmentation.Voronoi*, *imgaug.augmenters.segmentation.RegularGridPointsSampler* and *imgaug.augmenters.segmentation.DropoutPointsSampler*. Hence, it generates a regular grid with `R`

rows and C columns of coordinates on each image. Then, it drops p percent of the R*C coordinates to randomize the grid. Each image pixel then belongs to the voronoi cell with the closest coordinate.

Note: In contrast to the other voronoi augmenters, this one uses None as the default value for *max_size*, i.e. the color averaging is always performed at full resolution. This enables the augmenter to make most use of the added points for larger images. It does however slow down the augmentation process.

---

dtype support:

```
See ``imgaug.augmenters.segmentation.Voronoi``.
```

Parameters

- **n_rows_frac** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Relative number of coordinates to place on the y-axis. For a value y and image height H the number of actually placed coordinates (i.e. computed rows) is given by int(round(y*H)). Note that for each image, the number of coordinates is clipped to the interval [1,H], where H is the image height.
  - If a single number, then that value will always be used.
  - If a tuple (a, b), then a value from the interval [a, b] will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a StochasticParameter, then that parameter will be queried to draw one value per image.
- **n_cols_frac** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Relative number of coordinates to place on the x-axis. For a value x and image height W the number of actually placed coordinates (i.e. computed columns) is given by int(round(x*W)). Note that for each image, the number of coordinates is clipped to the interval [1,W], where W is the image width.
  - If a single number, then that value will always be used.
  - If a tuple (a, b), then a value from the interval [a, b] will be sampled per image.
  - If a list, then a random value will be sampled from that list per image.
  - If a StochasticParameter, then that parameter will be queried to draw one value per image.
- **p_drop_points** (*number or tuple of number or imgaug.parameters.StochasticParameter, optional*) – The probability that a coordinate will be removed from the list of all sampled coordinates. A value of 1.0 would mean that (on average) 100 percent of all coordinates will be dropped, while 0.0 denotes 0 percent. Note that this sampler will always ensure that at least one coordinate is left after the dropout operation, i.e. even 1.0 will only drop all *except one* coordinate.
  - If a float, then that value will be used for all images.
  - If a tuple (a, b), then a value p will be sampled from the interval [a, b] per image.
  - If a StochasticParameter, then this parameter will be used to determine per coordinate whether it should be *kept* (sampled value of >0.5) or shouldn't be kept (sampled value of <=0.5). If you instead want to provide the probability as a stochastic parameter,

you can usually do `imgaug.parameters.Binomial(1-p)` to convert parameter *p* to a 0/1 representation.

- **p_replace** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Defines for any segment the probability that the pixels within that segment are replaced by their average color (otherwise, the pixels are not changed). Examples:
  - A probability of `0.0` would mean, that the pixels in no segment are replaced by their average color (image is not changed at all).
  - A probability of `0.5` would mean, that around half of all segments are replaced by their average color.
  - A probability of `1.0` would mean, that all segments are replaced by their average color (resulting in a voronoi image).

  Behaviour based on chosen datatypes for this parameter:
  - If a `number`, then that `number` will always be used.
  - If `tuple (a, b)`, then a random probability will be sampled from the interval `[a, b]` per image.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, it is expected to return values between `0.0` and `1.0` and will be queried *for each individual segment* to determine whether it is supposed to be averaged (`>0.5`) or not (`<=0.5`). Recommended to be some form of `Binomial(...)`.
- **max_size** (*int or None, optional*) – Maximum image size at which the augmentation is performed. If the width or height of an image exceeds this value, it will be downscaled before the augmentation so that the longest side matches *max_size*. This is done to speed up the process. The final output image has the same size as the input image. Note that in case *p_replace* is below `1.0`, the down-/upscaling will affect the not-replaced pixels too. Use `None` to apply no down-/upscaling.
- **interpolation** (*int or str, optional*) – Interpolation method to use during downscaling when *max_size* is exceeded. Valid methods are the same as in *imgaug.imgaug.imresize_single_image()*.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.RelativeRegularGridVoronoi(0.1, 0.25)
```

Place a regular grid of `R x C` coordinates on each image, where `R` is the number of rows and computed as `R=0.1*H` with `H` being the height of the input image. `C` is the number of columns and analogously estimated

from the image width W as C=0.25*W. Larger images will lead to larger R and C values. On average, 20 percent of these grid coordinates are randomly dropped to create a less regular pattern. Then, the remaining coordinates are used to group the image pixels into voronoi cells and the colors within them are averaged.

```
>>> aug = iaa.RelativeRegularGridVoronoi(
>>>     (0.03, 0.1), 0.1, p_drop_points=0.0, p_replace=0.9, max_size=512)
```

Same as above, generates a grid with randomly R=r*H rows, where r is sampled uniformly from the interval [0.03, 0.1] and C=0.1*W rows. No points are dropped. The augmenter replaces only 90 percent of the voronoi cells with their average color (the pixels of the remaining 10 percent are not changed). Images larger than 512 px are temporarily downscaled (*before* sampling the grid points) so that no side exceeds 512 px. This improves performance, but degrades the quality of the resulting image.

Methods

__call__(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
augment(self[, return_batch, hooks])	Augment a batch.
augment_batch(self, batch[, hooks])	Augment a single batch.
augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ...[, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.

Continued on next page

Table 133 – continued from previous page

localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.segmentation.**SubsamplingPointsSampler**(*other_points_sampler*, *n_points_max*)

Bases: *imgaug.augmenters.segmentation.IPointsSampler*

Ensure that the number of sampled points is below a maximum.

This point sampler will sample points from another sampler and then – in case more points were generated than an allowed maximum – will randomly pick *n_points_max* of these.

Parameters

- **other_points_sampler** (*IPointsSampler*) – Another point sampler that is queried to generate a `list` of points. The dropout operation will be applied to that `list`.
- **n_points_max** (*int*) – Maximum number of allowed points. If *other_points_sampler* generates more points than this maximum, a random subset of size *n_points_max* will be selected.

Examples

```
>>> import imgaug.augmenters as iaa
>>> sampler = iaa.SubsamplingPointsSampler(
>>>     iaa.RelativeRegularGridPointsSampler(0.1, 0.2),
>>>     50
>>> )
```

Create a points sampler that places `y*H` points on the y-axis (with `y` being `0.1` and `H` being an image's height) and `x*W` on the x-axis (analogous). Then, if that number of placed points exceeds `50` (can easily happen for larger images), a random subset of `50` points will be picked and returned.

Methods

sample_points(self, images, random_state)	Generate coordinates of points on images.

sample_points(*self*, *images*, *random_state*)

Generate coordinates of points on images.

Parameters

- **images** (*ndarray or list of ndarray*) – One or more images for which to generate points. If this is a `list` of arrays, each one of them is expected to have three dimensions. If this is an array, it must be four-dimensional and the first axis is expected to denote the image index. For `RGB` images the array would hence have to be of shape `(N, H, W, 3)`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState*) – A random state to use for any probabilistic function required during the point sampling. See *imgaug.random.RNG()* for details.

Returns An `(N,2) float32` array containing `(x,y)` subpixel coordinates, all of which being within the intervals `[0.0, width]` and `[0.0, height]`.

Return type ndarray

class imgaug.augmenters.segmentation.**Superpixels**(*p_replace=0*, *n_segments=100*, *max_size=128*, *interpolation='linear'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Transform images parially/completely to their superpixel representation.

This implementation uses skimage's version of the SLIC algorithm.

Note: This augmenter is fairly slow. See *Performance*.

---

dtype support:

```
if (image size <= max_size)::

    * ``uint8``: yes; fully tested
    * ``uint16``: yes; tested
    * ``uint32``: yes; tested
    * ``uint64``: limited (1)
    * ``int8``: yes; tested
    * ``int16``: yes; tested
    * ``int32``: yes; tested
    * ``int64``: limited (1)
    * ``float16``: no (2)
    * ``float32``: no (2)
    * ``float64``: no (3)
    * ``float128``: no (2)
    * ``bool``: yes; tested

    - (1) Superpixel mean intensity replacement requires computing
          these means as float64s. This can cause inaccuracies for
          large integer values.
    - (2) Error in scikit-image.
    - (3) Loss of resolution in scikit-image.

if (image size > max_size)::

    minimum of (
        ``imgaug.augmenters.segmentation.Superpixels(image size <= max_size)``,
        :func:`imgaug.augmenters.segmentation._ensure_image_max_size`
    )
```

Parameters

- **p_replace** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Defines for any segment the probability that the pixels within that segment are replaced by their average color (otherwise, the pixels are not changed). Examples:
  - A probability of `0.0` would mean, that the pixels in no segment are replaced by their average color (image is not changed at all).
  - A probability of `0.5` would mean, that around half of all segments are replaced by their average color.
  - A probability of `1.0` would mean, that all segments are replaced by their average color (resulting in a voronoi image).

  Behaviour based on chosen datatypes for this parameter:
  - If a `number`, then that `number` will always be used.
  - If `tuple (a, b)`, then a random probability will be sampled from the interval `[a, b]` per image.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, it is expected to return values between `0.0` and `1.0` and will be queried *for each individual segment* to determine whether it is supposed to be averaged (`>0.5`) or not (`<=0.5`). Recommended to be some form of `Binomial(...)`.
- **n_segments** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) – Rough target number of how many superpixels to generate (the algorithm may deviate from this number). Lower value will lead to coarser superpixels. Higher values are computationally more intensive and will hence lead to a slowdown.
  - If a single `int`, then that value will always be used as the number of segments.
  - If a `tuple (a, b)`, then a value from the discrete interval `[a..b]` will be sampled per image.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, then that parameter will be queried to draw one value per image.
- **max_size** (*int or None, optional*) – Maximum image size at which the augmentation is performed. If the width or height of an image exceeds this value, it will be downscaled before the augmentation so that the longest side matches *max_size*. This is done to speed up the process. The final output image has the same size as the input image. Note that in case *p_replace* is below `1.0`, the down-/upscaling will affect the not-replaced pixels too. Use `None` to apply no down-/upscaling.
- **interpolation** (*int or str, optional*) – Interpolation method to use during downscaling when *max_size* is exceeded. Valid methods are the same as in *imgaug.imgaug.imresize_single_image()*.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Superpixels(p_replace=1.0, n_segments=64)
```

Generate around `64` superpixels per image and replace all of them with their average color (standard superpixel image).

```
>>> aug = iaa.Superpixels(p_replace=0.5, n_segments=64)
```

Generate around `64` superpixels per image and replace half of them with their average color, while the other half are left unchanged (i.e. they still show the input image's content).

```
>>> aug = iaa.Superpixels(p_replace=(0.25, 1.0), n_segments=(16, 128))
```

Generate between `16` and `128` superpixels per image and replace `25` to `100` percent of them with their average color.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.

Continued on next page

Table 135 – continued from previous page

find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.segmentation.**UniformPointsSampler**(*n_points*)

Bases: *imgaug.augmenters.segmentation.IPointsSampler*

Sample points uniformly on images.

This point sampler generates *n_points* points per image. The x- and y-coordinates are both sampled from uniform distributions matching the respective image width and height.

Parameters **n_points** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) –

Number of points to sample on each image.

- If a single `int`, then that value will always be used.
- If a `tuple (a, b)`, then a value from the discrete interval `[a..b]` will be sampled per image.
- If a `list`, then a random value will be sampled from that `list` per image.
- If a `StochasticParameter`, then that parameter will be queried to draw one value per image.

Examples

```
>>> import imgaug.augmenters as iaa
>>> sampler = iaa.UniformPointsSampler(500)
```

Create a point sampler that generates an array of `500` random points for each input image. The x- and y-coordinates of each point are sampled from uniform distributions.

Methods

sample_points(self, images, random_state)	Generate coordinates of points on images.

sample_points (*self*, *images*, *random_state*)
Generate coordinates of points on images.

Parameters

- **images** (*ndarray or list of ndarray*) – One or more images for which to generate points. If this is a `list` of arrays, each one of them is expected to have three dimensions. If this is an array, it must be four-dimensional and the first axis is expected to denote the image index. For `RGB` images the array would hence have to be of shape `(N, H, W, 3)`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState*) – A random state to use for any probabilistic function required during the point sampling. See *imgaug.random.RNG()* for details.

Returns An `(N,2) float32` array containing `(x,y)` subpixel coordinates, all of which being within the intervals `[0.0, width]` and `[0.0, height]`.

Return type ndarray

class imgaug.augmenters.segmentation.**UniformVoronoi**(*n_points*, *p_replace=1.0*, *max_size=128*, *interpolation='linear'*, *name=None*, *deterministic=False*, *random_state=None*)
Bases: *imgaug.augmenters.segmentation.Voronoi*

Uniformly sample Voronoi cells on images and average colors within them.

This augmenter is a shortcut for the combination of *imgaug.augmenters.segmentation.Voronoi* with *imgaug.augmenters.segmentation.UniformPointsSampler*. Hence, it generates a fixed amount of `N` random coordinates of voronoi cells on each image. The cell coordinates are sampled uniformly using the image height and width as maxima.

dtype support:

```
See ``imgaug.augmenters.segmentation.Voronoi``.
```

Parameters

- **n_points** (*int or tuple of int or list of int or imgaug.parameters.StochasticParameter, optional*) –

  Number of points to sample on each image.

  – If a single `int`, then that value will always be used.

  – If a `tuple (a, b)`, then a value from the discrete interval `[a..b]` will be sampled per image.

  – If a `list`, then a random value will be sampled from that `list` per image.

  - If a `StochasticParameter`, then that parameter will be queried to draw one value per image.

- **p_replace** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Defines for any segment the probability that the pixels within that segment are replaced by their average color (otherwise, the pixels are not changed). Examples:

  - A probability of `0.0` would mean, that the pixels in no segment are replaced by their average color (image is not changed at all).
  - A probability of `0.5` would mean, that around half of all segments are replaced by their average color.
  - A probability of `1.0` would mean, that all segments are replaced by their average color (resulting in a voronoi image).

  Behaviour based on chosen datatypes for this parameter:

  - If a `number`, then that `number` will always be used.
  - If `tuple (a, b)`, then a random probability will be sampled from the interval `[a, b]` per image.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, it is expected to return values between `0.0` and `1.0` and will be queried *for each individual segment* to determine whether it is supposed to be averaged (`>0.5`) or not (`<=0.5`). Recommended to be some form of `Binomial(...)`.

- **max_size** (*int or None, optional*) – Maximum image size at which the augmentation is performed. If the width or height of an image exceeds this value, it will be downscaled before the augmentation so that the longest side matches *max_size*. This is done to speed up the process. The final output image has the same size as the input image. Note that in case *p_replace* is below `1.0`, the down-/upscaling will affect the not-replaced pixels too. Use `None` to apply no down-/upscaling.

- **interpolation** (*int or str, optional*) – Interpolation method to use during downscaling when *max_size* is exceeded. Valid methods are the same as in *imgaug.imgaug.imresize_single_image()*.

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.UniformVoronoi((100, 500))
```

Sample for each image uniformly the number of voronoi cells `N` from the interval `[100, 500]`. Then generate `N` coordinates by sampling uniformly the x-coordinates from `[0, W]` and the y-coordinates from `[0, H]`,

where H is the image height and W the image width. Then use these coordinates to group the image pixels into voronoi cells and average the colors within them. The process is performed at an image size not exceeding 128 px on any side (default). If necessary, the downscaling is performed using linear interpolation (default).

```
>>> aug = iaa.UniformVoronoi(250, p_replace=0.9, max_size=None)
```

Same as above, but always samples N=250 cells, replaces only 90 percent of them with their average color (the pixels of the remaining 10 percent are not changed) and performs the transformation at the original image size (max_size=None).

Methods

__call__(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
augment(self[, return_batch, hooks])	Augment a batch.
augment_batch(self, batch[, hooks])	Augment a single batch.
augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ...[, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.

Continued on next page

Table 137 – continued from previous page

remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.segmentation.**Voronoi**(*points_sampler*, *p_replace=1.0*, *max_size=128*, *interpolation='linear'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Average colors of an image within Voronoi cells.

This augmenter performs the following steps:

1. Query *points_sampler* to sample random coordinates of cell centers. On the image.
2. Estimate for each pixel to which voronoi cell (i.e. segment) it belongs. Each pixel belongs to the cell with the closest center coordinate (euclidean distance).
3. Compute for each cell the average color of the pixels within it.
4. Replace the pixels of *p_replace* percent of all cells by their average color. Do not change the pixels of `(1 - p_replace)` percent of all cells. (The percentages are average values over many images. Some images may get more/less cells replaced by their average color.)

This code is very loosely based on https://codegolf.stackexchange.com/questions/50299/draw-an-image-as-a-voronoi-map/50345#50345

dtype support:

```
if (image size <= max_size)::

    * ``uint8``: yes; fully tested
    * ``uint16``: no; not tested
    * ``uint32``: no; not tested
    * ``uint64``: no; not tested
    * ``int8``: no; not tested
    * ``int16``: no; not tested
    * ``int32``: no; not tested
    * ``int64``: no; not tested
    * ``float16``: no; not tested
    * ``float32``: no; not tested
    * ``float64``: no; not tested
    * ``float128``: no; not tested
    * ``bool``: no; not tested

if (image size > max_size)::

    minimum of (
```

(continues on next page)

(continued from previous page)

```
        ``imgaug.augmenters.segmentation.Voronoi(image size <= max_size)``,
        :func:`imgaug.augmenters.segmentation._ensure_image_max_size`
    )
```

Parameters

- **points_sampler** (*IPointsSampler*) – A points sampler which will be queried per image to generate the coordinates of the centers of voronoi cells.
- **p_replace** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Defines for any segment the probability that the pixels within that segment are replaced by their average color (otherwise, the pixels are not changed). Examples:
  - A probability of `0.0` would mean, that the pixels in no segment are replaced by their average color (image is not changed at all).
  - A probability of `0.5` would mean, that around half of all segments are replaced by their average color.
  - A probability of `1.0` would mean, that all segments are replaced by their average color (resulting in a voronoi image).

  Behaviour based on chosen datatypes for this parameter:
  - If a `number`, then that `number` will always be used.
  - If `tuple (a, b)`, then a random probability will be sampled from the interval `[a, b]` per image.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, it is expected to return values between `0.0` and `1.0` and will be queried *for each individual segment* to determine whether it is supposed to be averaged (`>0.5`) or not (`<=0.5`). Recommended to be some form of `Binomial(...)`.
- **max_size** (*int or None, optional*) – Maximum image size at which the augmentation is performed. If the width or height of an image exceeds this value, it will be downscaled before the augmentation so that the longest side matches *max_size*. This is done to speed up the process. The final output image has the same size as the input image. Note that in case *p_replace* is below `1.0`, the down-/upscaling will affect the not-replaced pixels too. Use `None` to apply no down-/upscaling.
- **interpolation** (*int or str, optional*) – Interpolation method to use during downscaling when *max_size* is exceeded. Valid methods are the same as in *imgaug.imgaug.imresize_single_image()*.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> points_sampler = iaa.RegularGridPointsSampler(n_cols=20, n_rows=40)
>>> aug = iaa.Voronoi(points_sampler)
```

Create an augmenter that places a `20x40` (`HxW`) grid of cells on the image and replaces all pixels within each cell by the cell's average color. The process is performed at an image size not exceeding `128` px on any side (default). If necessary, the downscaling is performed using `linear` interpolation (default).

```
>>> points_sampler = iaa.DropoutPointsSampler(
>>>     iaa.RelativeRegularGridPointsSampler(
>>>         n_cols_frac=(0.05, 0.2),
>>>         n_rows_frac=0.1),
>>>     0.2)
>>> aug = iaa.Voronoi(points_sampler, p_replace=0.9, max_size=None)
```

Create a voronoi augmenter that generates a grid of cells dynamically adapted to the image size. Larger images get more cells. On the x-axis, the distance between two cells is `w * W` pixels, where `W` is the width of the image and `w` is always `0.1`. On the y-axis, the distance between two cells is `h * H` pixels, where `H` is the height of the image and `h` is sampled uniformly from the interval `[0.05, 0.2]`. To make the voronoi pattern less regular, about `20` percent of the cell coordinates are randomly dropped (i.e. the remaining cells grow in size). In contrast to the first example, the image is not resized (if it was, the sampling would happen *after* the resizing, which would affect `W` and `H`). Not all voronoi cells are replaced by their average color, only around `90` percent of them. The remaining `10` percent's pixels remain unchanged.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.

Continued on next page

Table 138 – continued from previous page

draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

imgaug.augmenters.segmentation.**segment_voronoi**(*image*, *cell_coordinates*, *replace_mask=None*)

Average colors within voronoi cells of an image.

Parameters

- **image** (*ndarray*) – The image to convert to a voronoi image. May be `HxW` or `HxWxC`. Note that for `RGBA` images the alpha channel will currently also by averaged.
- **cell_coordinates** (*ndarray*) – A `Nx2` float array containing the center coordinates of voronoi cells on the image. Values are expected to be in the interval `[0.0, height-1.0]` for the y-axis (x-axis analogous). If this array contains no coordinate, the image will not be changed.
- **replace_mask** (*None or ndarray, optional*) – Boolean mask of the same length as *cell_coordinates*, denoting for each cell whether its pixels are supposed to be replaced by the cell's average color (`True`) or left untouched (`False`). If this is set to `None`, all cells will be replaced.

Returns Voronoi image.

Return type ndarray

13.28 imgaug.augmenters.size

Augmenters that somehow change the size of the images.

Do not import directly from this file, as the categorization is not final. Use instead

```
from imgaug import augmenters as iaa
```

and then e.g.

```
seq = iaa.Sequential([
    iaa.Resize({"height": 32, "width": 64})
    iaa.Crop((0, 20))
])
```

List of augmenters:

- Resize
- CropAndPad
- Crop
- Pad
- PadToFixedSize
- CropToFixedSize
- KeepSizeByResize

class imgaug.augmenters.size.**Crop**(*px=None*, *percent=None*, *keep_size=True*, *sample_independently=True*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.size.CropAndPad*

Crop images, i.e. remove columns/rows of pixels at the sides of images.

This augmenter allows to extract smaller-sized subimages from given full-sized input images. The number of pixels to cut off may be defined in absolute values or as fractions of the image sizes.

This augmenter will never crop images below a height or width of `1`.

dtype support:

```
See ``imgaug.augmenters.size.CropAndPad``.
```

Parameters

- **px** (*None or int or imgaug.parameters.StochasticParameter or tuple, optional*) – The number of pixels to crop on each side of the image. Expected value range is `[0, inf)`. Either this or the parameter *percent* may be set, not both at the same time.
  - If `None`, then pixel-based cropping will not be used.
  - If `int`, then that exact number of pixels will always be cropped.
  - If `StochasticParameter`, then that parameter will be used for each image. Four samples will be drawn per image (top, right, bottom, left), unless *sample_independently* is set to `False`, as then only one value will be sampled per image and used for all sides.
  - If a `tuple` of two `int` s with values `a` and `b`, then each side will be cropped by a random amount sampled uniformly per image and side from the inteval `[a, b]`. If however *sample_independently* is set to `False`, only one value will be sampled per image and used for all sides.

  - If a `tuple` of four entries, then the entries represent top, right, bottom, left. Each entry may be a single `int` (always crop by exactly that value), a `tuple` of two `int` s `a` and `b` (crop by an amount within `[a, b]`), a `list` of `int` s (crop by a random value that is contained in the `list`) or a `StochasticParameter` (sample the amount to crop from that parameter).

- **percent** (*None or int or float or imgaug.parameters.StochasticParameter or tuple, optional*) – The number of pixels to crop on each side of the image given as a *fraction* of the image height/width. E.g. if this is set to `0.1`, the augmenter will always crop `10%` of the image's height at both the top and the bottom (both `10%` each), as well as `10%` of the width at the right and left. Expected value range is `[0.0, 1.0)`. Either this or the parameter *px* may be set, not both at the same time.

  - If `None`, then fraction-based cropping will not be used.

  - If `number`, then that fraction will always be cropped.

  - If `StochasticParameter`, then that parameter will be used for each image. Four samples will be drawn per image (top, right, bottom, left). If however *sample_independently* is set to `False`, only one value will be sampled per image and used for all sides.

  - If a `tuple` of two `float` s with values `a` and `b`, then each side will be cropped by a random fraction sampled uniformly per image and side from the interval `[a, b]`. If however *sample_independently* is set to `False`, only one value will be sampled per image and used for all sides.

  - If a `tuple` of four entries, then the entries represent top, right, bottom, left. Each entry may be a single `float` (always crop by exactly that fraction), a `tuple` of two `float` s `a` and `b` (crop by a fraction from `[a, b]`), a `list` of `float` s (crop by a random value that is contained in the list) or a `StochasticParameter` (sample the percentage to crop from that parameter).

- **keep_size** (*bool, optional*) – After cropping, the result image will usually have a different height/width compared to the original input image. If this parameter is set to `True`, then the cropped image will be resized to the input image's size, i.e. the augmenter's output shape is always identical to the input shape.

- **sample_independently** (*bool, optional*) – If `False` *and* the values for *px*/*percent* result in exactly *one* probability distribution for all image sides, only one single value will be sampled from that probability distribution and used for all sides. I.e. the crop amount then is the same for all sides. If `True`, four values will be sampled independently, one per side.

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Crop(px=(0, 10))
```

Crop each side by a random pixel value sampled uniformly per image and side from the discrete interval `[0..10]`.

```
>>> aug = iaa.Crop(px=(0, 10), sample_independently=False)
```

Crop each side by a random pixel value sampled uniformly once per image from the discrete interval `[0..10]`. Each sampled value is used for *all* sides of the corresponding image.

```
>>> aug = iaa.Crop(px=(0, 10), keep_size=False)
```

Crop each side by a random pixel value sampled uniformly per image and side from the discrete interval `[0..10]`. Afterwards, do **not** resize the cropped image back to the input image's size. This will decrease the image's height and width by a maximum of `20` pixels.

```
>>> aug = iaa.Crop(px=((0, 10), (0, 5), (0, 10), (0, 5)))
```

Crop the top and bottom by a random pixel value sampled uniformly from the discrete interval `[0..10]`. Crop the left and right analogously by a random value sampled from `[0..5]`. Each value is always sampled independently.

```
>>> aug = iaa.Crop(percent=(0, 0.1))
```

Crop each side by a random fraction sampled uniformly from the continuous interval `[0.0, 0.10]`. The fraction is sampled once per image and side. E.g. a sampled fraction of `0.1` for the top side would crop by `0.1*H`, where `H` is the height of the input image.

```
>>> aug = iaa.Crop(
>>>     percent=([0.05, 0.1], [0.05, 0.1], [0.05, 0.1], [0.05, 0.1]))
```

Crops each side by either `5%` or `10%`. The values are sampled once per side and image.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.

Continued on next page

Table 139 – continued from previous page

copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.size.**CropAndPad**(*px=None*, *percent=None*, *pad_mode='constant'*, *pad_cval=0*, *keep_size=True*, *sample_independently=True*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Crop/pad images by pixel amounts or fractions of image sizes.

Cropping removes pixels at the sides (i.e. extracts a subimage from a given full image). Padding adds pixels to the sides (e.g. black pixels).

This augmenter will never crop images below a height or width of `1`.

Note: This augmenter automatically resizes images back to their original size after it has augmented them. To deactivate this, add the parameter `keep_size=False`.

dtype support:

```
if (keep_size=False)::

    * ``uint8``: yes; fully tested
```

(continues on next page)

(continued from previous page)

```
    * ``uint16``: yes; tested
    * ``uint32``: yes; tested
    * ``uint64``: yes; tested
    * ``int8``: yes; tested
    * ``int16``: yes; tested
    * ``int32``: yes; tested
    * ``int64``: yes; tested
    * ``float16``: yes; tested
    * ``float32``: yes; tested
    * ``float64``: yes; tested
    * ``float128``: yes; tested
    * ``bool``: yes; tested

if (keep_size=True)::

    minimum of (
        ``imgaug.augmenters.size.CropAndPad(keep_size=False)``,
        :func:`imgaug.imgaug.imresize_many_images`
    )
```

Parameters

- **px** (*None or int or imgaug.parameters.StochasticParameter or tuple, optional*) – The number of pixels to crop (negative values) or pad (positive values) on each side of the image. Either this or the parameter *percent* may be set, not both at the same time.
  - If `None`, then pixel-based cropping/padding will not be used.
  - If `int`, then that exact number of pixels will always be cropped/padded.
  - If `StochasticParameter`, then that parameter will be used for each image. Four samples will be drawn per image (top, right, bottom, left), unless *sample_independently* is set to `False`, as then only one value will be sampled per image and used for all sides.
  - If a `tuple` of two `int` s with values `a` and `b`, then each side will be cropped/padded by a random amount sampled uniformly per image and side from the inteval `[a, b]`. If however *sample_independently* is set to `False`, only one value will be sampled per image and used for all sides.
  - If a `tuple` of four entries, then the entries represent top, right, bottom, left. Each entry may be a single `int` (always crop/pad by exactly that value), a `tuple` of two `int` s `a` and `b` (crop/pad by an amount within `[a, b]`), a `list` of `int` s (crop/pad by a random value that is contained in the `list`) or a `StochasticParameter` (sample the amount to crop/pad from that parameter).
- **percent** (*None or number or imgaug.parameters.StochasticParameter or tuple, optional*) – The number of pixels to crop (negative values) or pad (positive values) on each side of the image given as a *fraction* of the image height/width. E.g. if this is set to `-0.1`, the augmenter will always crop away `10%` of the image's height at both the top and the bottom (both `10%` each), as well as `10%` of the width at the right and left. Expected value range is `(-1.0, inf)`. Either this or the parameter *px* may be set, not both at the same time.
  - If `None`, then fraction-based cropping/padding will not be used.
  - If `number`, then that fraction will always be cropped/padded.
  - If `StochasticParameter`, then that parameter will be used for each image. Four samples will be drawn per image (top, right, bottom, left). If however *sam-*

ple_independently is set to `False`, only one value will be sampled per image and used for all sides.

- If a `tuple` of two `float` s with values `a` and `b`, then each side will be cropped/padded by a random fraction sampled uniformly per image and side from the interval `[a, b]`. If however *sample_independently* is set to `False`, only one value will be sampled per image and used for all sides.
- If a `tuple` of four entries, then the entries represent top, right, bottom, left. Each entry may be a single `float` (always crop/pad by exactly that percent value), a `tuple` of two `float` s `a` and `b` (crop/pad by a fraction from `[a, b]`), a `list` of `float` s (crop/pad by a random value that is contained in the list) or a `StochasticParameter` (sample the percentage to crop/pad from that parameter).

- **pad_mode** (*imgaug.ALL or str or list of str or imgaug.parameters.StochasticParameter, optional*) – Padding mode to use. The available modes match the numpy padding modes, i.e. `constant`, `edge`, `linear_ramp`, `maximum`, `median`, `minimum`, `reflect`, `symmetric`, `wrap`. The modes `constant` and `linear_ramp` use extra values, which are provided by `pad_cval` when necessary. See *imgaug.imgaug.pad()* for more details.
  - If `imgaug.ALL`, then a random mode from all available modes will be sampled per image.
  - If a `str`, it will be used as the pad mode for all images.
  - If a `list` of `str`, a random one of these will be sampled per image and used as the mode.
  - If `StochasticParameter`, a random mode will be sampled from this parameter per image.
- **pad_cval** (*number or tuple of number list of number or imgaug.parameters.StochasticParameter, optional*) – The constant value to use if the pad mode is `constant` or the end value to use if the mode is `linear_ramp`. See *imgaug.imgaug.pad()* for more details.
  - If `number`, then that value will be used.
  - If a `tuple` of two `number` s and at least one of them is a `float`, then a random number will be uniformly sampled per image from the continuous interval `[a, b]` and used as the value. If both `number` s are `int` s, the interval is discrete.
  - If a `list` of `number`, then a random value will be chosen from the elements of the `list` and used as the value.
  - If `StochasticParameter`, a random value will be sampled from that parameter per image.
- **keep_size** (*bool, optional*) – After cropping and padding, the result image will usually have a different height/width compared to the original input image. If this parameter is set to `True`, then the cropped/padded image will be resized to the input image's size, i.e. the augmenter's output shape is always identical to the input shape.
- **sample_independently** (*bool, optional*) – If `False` *and* the values for *px/percent* result in exactly *one* probability distribution for all image sides, only one single value will be sampled from that probability distribution and used for all sides. I.e. the crop/pad amount then is the same for all sides. If `True`, four values will be sampled independently, one per side.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.CropAndPad(px=(-10, 0))
```

Crop each side by a random pixel value sampled uniformly per image and side from the discrete interval `[-10..0]`.

```
>>> aug = iaa.CropAndPad(px=(0, 10))
```

Pad each side by a random pixel value sampled uniformly per image and side from the discrete interval `[0..10]`. The padding happens by zero-padding, i.e. it adds black pixels (default setting).

```
>>> aug = iaa.CropAndPad(px=(0, 10), pad_mode="edge")
```

Pad each side by a random pixel value sampled uniformly per image and side from the discrete interval `[0..10]`. The padding uses the `edge` mode from numpy's pad function, i.e. the pixel colors around the image sides are repeated.

```
>>> aug = iaa.CropAndPad(px=(0, 10), pad_mode=["constant", "edge"])
```

Similar to the previous example, but uses zero-padding (`constant`) for half of the images and `edge` padding for the other half.

```
>>> aug = iaa.CropAndPad(px=(0, 10), pad_mode=ia.ALL, pad_cval=(0, 255))
```

Similar to the previous example, but uses any available padding mode. In case the padding mode ends up being `constant` or `linear_ramp`, and random intensity is uniformly sampled (once per image) from the discrete interval `[0..255]` and used as the intensity of the new pixels.

```
>>> aug = iaa.CropAndPad(px=(0, 10), sample_independently=False)
```

Pad each side by a random pixel value sampled uniformly once per image from the discrete interval `[0..10]`. Each sampled value is used for *all* sides of the corresponding image.

```
>>> aug = iaa.CropAndPad(px=(0, 10), keep_size=False)
```

Pad each side by a random pixel value sampled uniformly per image and side from the discrete interval `[0..10]`. Afterwards, do **not** resize the padded image back to the input image's size. This will increase the image's height and width by a maximum of `20` pixels.

```
>>> aug = iaa.CropAndPad(px=((0, 10), (0, 5), (0, 10), (0, 5)))
```

Pad the top and bottom by a random pixel value sampled uniformly from the discrete interval `[0..10]`. Pad the left and right analogously by a random value sampled from `[0..5]`. Each value is always sampled independently.

```
>>> aug = iaa.CropAndPad(percent=(0, 0.1))
```

Pad each side by a random fraction sampled uniformly from the continuous interval `[0.0, 0.10]`. The fraction is sampled once per image and side. E.g. a sampled fraction of `0.1` for the top side would pad by `0.1*H`, where `H` is the height of the input image.

```
>>> aug = iaa.CropAndPad(
>>>     percent=([0.05, 0.1], [0.05, 0.1], [0.05, 0.1], [0.05, 0.1]))
```

Pads each side by either `5%` or `10%`. The values are sampled once per side and image.

```
>>> aug = iaa.CropAndPad(px=(-10, 10))
```

Sample uniformly per image and side a value `v` from the discrete range `[-10..10]`. Then either crop (negative sample) or pad (positive sample) the side by `v` pixels.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.

Continued on next page

Table 140 – continued from previous page

localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters	

get_parameters(*self*)

class imgaug.augmenters.size.**CropToFixedSize**(*width*, *height*, *position='uniform'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Crop images down to a predefined maximum width and/or height.

If images are already at the maximum width/height or are smaller, they will not be cropped. Note that this also means that images will not be padded if they are below the required width/height.

The augmenter randomly decides per image how to distribute the required cropping amounts over the image axis. E.g. if 2px have to be cropped on the left or right to reach the required width, the augmenter will sometimes remove 2px from the left and 0px from the right, sometimes remove 2px from the right and 0px from the left and sometimes remove 1px from both sides. Set *position* to `center` to prevent that.

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: yes; tested
* ``uint32``: yes; tested
* ``uint64``: yes; tested
* ``int8``: yes; tested
* ``int16``: yes; tested
* ``int32``: yes; tested
* ``int64``: yes; tested
* ``float16``: yes; tested
* ``float32``: yes; tested
* ``float64``: yes; tested
* ``float128``: yes; tested
* ``bool``: yes; tested
```

Parameters

- **width** (*int*) – Crop images down to this maximum width.
- **height** (*int*) – Crop images down to this maximum height.

- **position** (*{'uniform', 'normal', 'center', 'left-top', 'left-center', 'left-bottom', 'center-top', 'center-center', 'center-bottom', 'right-top', 'right-center', 'right-bottom'} or tuple of float or StochasticParameter or tuple of StochasticParameter, optional*) – Sets the center point of the cropping, which determines how the required cropping amounts are distributed to each side. For a `tuple (a, b)`, both `a` and `b` are expected to be in range `[0.0, 1.0]` and describe the fraction of cropping applied to the left/right (low/high values for `a`) and the fraction of cropping applied to the top/bottom (low/high values for `b`). A cropping position at `(0.5, 0.5)` would be the center of the image and distribute the cropping equally over all sides. A cropping position at `(1.0, 0.0)` would be the right-top and would apply 100% of the required cropping to the right and top sides of the image.
  - If string `uniform` then the share of cropping is randomly and uniformly distributed over each side. Equivalent to `(Uniform(0.0, 1.0), Uniform(0.0, 1.0))`.
  - If string `normal` then the share of cropping is distributed based on a normal distribution, leading to a focus on the center of the images. Equivalent to `(Clip(Normal(0.5, 0.45/2), 0, 1), Clip(Normal(0.5, 0.45/2), 0, 1))`.
  - If string `center` then center point of the cropping is identical to the image center. Equivalent to `(0.5, 0.5)`.
  - If a string matching regex `^(left|center|right)-(top|center|bottom)$`, e.g. `left-top` or `center-bottom` then sets the center point of the cropping to the X-Y position matching that description.
  - If a tuple of float, then expected to have exactly two entries between `0.0` and `1.0`, which will always be used as the combination the position matching (x, y) form.
  - If a `StochasticParameter`, then that parameter will be queried once per call to `augment_*()` to get `Nx2` center positions in `(x, y)` form (with `N` the number of images).
  - If a `tuple` of `StochasticParameter`, then expected to have exactly two entries that will both be queried per call to `augment_*()`, each for `(N,)` values, to get the center positions. First parameter is used for `x` coordinates, second for `y` coordinates.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.CropToFixedSize(width=100, height=100)
```

For image sides larger than `100` pixels, crop to `100` pixels. Do nothing for the other sides. The cropping amounts are randomly (and uniformly) distributed over the sides of the image.

```
>>> aug = iaa.CropToFixedSize(width=100, height=100, position="center")
```

For sides larger than 100 pixels, crop to 100 pixels. Do nothing for the other sides. The cropping amounts are always equally distributed over the left/right sides of the image (and analogously for top/bottom).

```
>>> aug = iaa.Sequential([
>>>     iaa.PadToFixedSize(width=100, height=100),
>>>     iaa.CropToFixedSize(width=100, height=100)
>>> ])
```

Pad images smaller than 100x100 until they reach 100x100. Analogously, crop images larger than 100x100 until they reach 100x100. The output images therefore have a fixed size of 100x100.

Methods

__call__(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
augment(self[, return_batch, hooks])	Augment a batch.
augment_batch(self, batch[, hooks])	Augment a single batch.
augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ...[, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.

Continued on next page

Table 141 – continued from previous page

remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.size.**KeepSizeByResize**(*children*, *interpolation='cubic'*, *interpolation_heatmaps='SAME_AS_IMAGES'*, *interpolation_segmaps='nearest'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Resize images back to their input sizes after applying child augmenters.

Combining this with e.g. a cropping augmenter as the child will lead to images being resized back to the input size after the crop operation was applied. Some augmenters have a `keep_size` argument that achieves the same goal (if set to `True`), though this augmenter offers control over the interpolation mode and which augmentables to resize (images, heatmaps, segmentation maps).

dtype support:

```
See :func:`imgaug.imgaug.imresize_many_images`.
```

Parameters

- **children** (*Augmenter or list of imgaug.augmenters.meta.Augmenter or None, optional*) – One or more augmenters to apply to images. These augmenters may change the image size.
- **interpolation** (*KeepSizeByResize.NO_RESIZE or {'nearest', 'linear', 'area', 'cubic'} or {cv2.INTER_NEAREST, cv2.INTER_LINEAR, cv2.INTER_AREA, cv2.INTER_CUBIC} or list of str or list of int or StochasticParameter, optional*) – The interpolation mode to use when resizing images. Can take any value that *imgaug.imgaug.imresize_single_image()* accepts, e.g. `cubic`.
  - If this is `KeepSizeByResize.NO_RESIZE` then images will not be resized.
  - If this is a single `str`, it is expected to have one of the following values: `nearest`, `linear`, `area`, `cubic`.
  - If this is a single integer, it is expected to have a value identical to one of: `cv2.INTER_NEAREST`, `cv2.INTER_LINEAR`, `cv2.INTER_AREA`, `cv2.INTER_CUBIC`.
  - If this is a `list` of `str` or `int`, it is expected that each `str`/`int` is one of the above mentioned valid ones. A random one of these values will be sampled per image.

  - If this is a `StochasticParameter`, it will be queried once per call to `_augment_images()` and must return `N` `str` s or `int` s (matching the above mentioned ones) for `N` images.

- **interpolation_heatmaps** (*KeepSizeByResize.SAME_AS_IMAGES or KeepSizeByResize.NO_RESIZE or {'nearest', 'linear', 'area', 'cubic'} or {cv2.INTER_NEAREST, cv2.INTER_LINEAR, cv2.INTER_AREA, cv2.INTER_CUBIC} or list of str or list of int or StochasticParameter, optional*) – The interpolation mode to use when resizing heatmaps. Meaning and valid values are similar to *interpolation*. This parameter may also take the value `KeepSizeByResize.SAME_AS_IMAGES`, which will lead to copying the interpolation modes used for the corresponding images. The value may also be returned on a per-image basis if *interpolation_heatmaps* is provided as a `StochasticParameter` or may be one possible value if it is provided as a `list` of `str`.
- **interpolation_segmaps** (*KeepSizeByResize.SAME_AS_IMAGES or KeepSizeByResize.NO_RESIZE or {'nearest', 'linear', 'area', 'cubic'} or {cv2.INTER_NEAREST, cv2.INTER_LINEAR, cv2.INTER_AREA, cv2.INTER_CUBIC} or list of str or list of int or StochasticParameter, optional*) – The interpolation mode to use when resizing segmentation maps. Similar to *interpolation_heatmaps*. **Note**: For segmentation maps, only `NO_RESIZE` or nearest neighbour interpolation (i.e. `nearest`) make sense in the vast majority of all cases.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.KeepSizeByResize(
>>>     iaa.Crop((20, 40), keep_size=False)
>>> )
```

Apply random cropping to input images, then resize them back to their original input sizes. The resizing is done using this augmenter instead of the corresponding internal resizing operation in `Crop`.

```
>>> aug = iaa.KeepSizeByResize(
>>>     iaa.Crop((20, 40), keep_size=False),
>>>     interpolation="nearest"
>>> )
```

Same as in the previous example, but images are now always resized using nearest neighbour interpolation.

```
>>> aug = iaa.KeepSizeByResize(
>>>     iaa.Crop((20, 40), keep_size=False),
>>>     interpolation=["nearest", "cubic"],
>>>     interpolation_heatmaps=iaa.KeepSizeByResize.SAME_AS_IMAGES,
>>>     interpolation_segmaps=iaa.KeepSizeByResize.NO_RESIZE
>>> )
```

Similar to the previous example, but images are now sometimes resized using linear interpolation and sometimes using nearest neighbour interpolation. Heatmaps are resized using the same interpolation as was used for the corresponding image. Segmentation maps are not resized and will therefore remain at their size after cropping.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
`remove_augmenters_inplace`(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
`reseed`(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
`show_grid`(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.

Continued on next page

Table 142 – continued from previous page

to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

NO_RESIZE = 'NO_RESIZE'

SAME_AS_IMAGES = 'SAME_AS_IMAGES'

get_children_lists(*self*)

Get a list of lists of children of this augmenter.

For most augmenters, the result will be a single empty list. For augmenters with children it will often be a list with one sublist containing all children. In some cases the augmenter will contain multiple distinct lists of children, e.g. an if-list and an else-list. This will lead to a result consisting of a single list with multiple sublists, each representing the respective sublist of children.

E.g. for an if/else-augmenter that executes the children `A1`, `A2` if a condition is met and otherwise executes the children `B1`, `B2`, `B3` the result will be `[[A1, A2], [B1, B2, B3]]`.

IMPORTANT: While the topmost list may be newly created, each of the sublist must be editable inplace resulting in a changed children list of the augmenter. E.g. if an Augmenter `IfElse(condition, [A1, A2], [B1, B2, B3])` returns `[[A1, A2], [B1, B2, B3]]` for a call to *imgaug.augmenters.meta.Augmenter.get_children_lists()* and `A2` is removed inplace from `[A1, A2]`, then the children lists of `IfElse(...)` must also change to `[A1]`, `[B1, B2, B3]`. This is used in `imgaug.augmeneters.meta.Augmenter.remove_augmenters_inplace()`.

Returns One or more lists of child augmenter. Can also be a single empty list.

Return type list of list of imgaug.augmenters.meta.Augmenter

get_parameters(*self*)

class imgaug.augmenters.size.**Pad**(*px=None*, *percent=None*, *pad_mode='constant'*, *pad_cval=0*, *keep_size=True*, *sample_independently=True*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.size.CropAndPad*

Pad images, i.e. adds columns/rows of pixels to them.

dtype support:

```
See ``imgaug.augmenters.size.CropAndPad``.
```

Parameters

- **px** (*None or int or imgaug.parameters.StochasticParameter or tuple, optional*) – The number of pixels to pad on each side of the image. Expected value range is `[0, inf)`. Either this or the parameter *percent* may be set, not both at the same time.
  - If `None`, then pixel-based padding will not be used.
  - If `int`, then that exact number of pixels will always be padded.
  - If `StochasticParameter`, then that parameter will be used for each image. Four samples will be drawn per image (top, right, bottom, left), unless *sample_independently* is set to `False`, as then only one value will be sampled per image and used for all sides.

  - If a `tuple` of two `int` s with values `a` and `b`, then each side will be padded by a random amount sampled uniformly per image and side from the inteval `[a, b]`. If however *sample_independently* is set to `False`, only one value will be sampled per image and used for all sides.
  - If a `tuple` of four entries, then the entries represent top, right, bottom, left. Each entry may be a single `int` (always pad by exactly that value), a `tuple` of two `int` s `a` and `b` (pad by an amount within `[a, b]`), a `list` of `int` s (pad by a random value that is contained in the `list`) or a `StochasticParameter` (sample the amount to pad from that parameter).

- **percent** (*None or int or float or imgaug.parameters.StochasticParameter or tuple, optional*) – The number of pixels to pad on each side of the image given as a *fraction* of the image height/width. E.g. if this is set to `0.1`, the augmenter will always pad `10%` of the image's height at both the top and the bottom (both `10%` each), as well as `10%` of the width at the right and left. Expected value range is `[0.0, inf)`. Either this or the parameter *px* may be set, not both at the same time.
  - If `None`, then fraction-based padding will not be used.
  - If `number`, then that fraction will always be padded.
  - If `StochasticParameter`, then that parameter will be used for each image. Four samples will be drawn per image (top, right, bottom, left). If however *sample_independently* is set to `False`, only one value will be sampled per image and used for all sides.
  - If a `tuple` of two `float` s with values `a` and `b`, then each side will be padded by a random fraction sampled uniformly per image and side from the interval `[a, b]`. If however *sample_independently* is set to `False`, only one value will be sampled per image and used for all sides.
  - If a `tuple` of four entries, then the entries represent top, right, bottom, left. Each entry may be a single `float` (always pad by exactly that fraction), a `tuple` of two `float` s `a` and `b` (pad by a fraction from `[a, b]`), a `list` of `float` s (pad by a random value that is contained in the list) or a `StochasticParameter` (sample the percentage to pad from that parameter).

- **pad_mode** (*imgaug.ALL or str or list of str or imgaug.parameters.StochasticParameter, optional*) – Padding mode to use. The available modes match the numpy padding modes, i.e. `constant`, `edge`, `linear_ramp`, `maximum`, `median`, `minimum`, `reflect`, `symmetric`, `wrap`. The modes `constant` and `linear_ramp` use extra values, which are provided by `pad_cval` when necessary. See *imgaug.imgaug.pad()* for more details.
  - If `imgaug.ALL`, then a random mode from all available modes will be sampled per image.
  - If a `str`, it will be used as the pad mode for all images.
  - If a `list` of `str`, a random one of these will be sampled per image and used as the mode.
  - If `StochasticParameter`, a random mode will be sampled from this parameter per image.

- **pad_cval** (*number or tuple of number list of number or imgaug.parameters.StochasticParameter, optional*) – The constant value to use if the pad mode is `constant` or the end value to use if the mode is `linear_ramp`. See *imgaug.imgaug.pad()* for more details.

  - If `number`, then that value will be used.
  - If a `tuple` of two `number` s and at least one of them is a `float`, then a random number will be uniformly sampled per image from the continuous interval `[a, b]` and used as the value. If both `number` s are `int` s, the interval is discrete.
  - If a `list` of `number`, then a random value will be chosen from the elements of the `list` and used as the value.
  - If `StochasticParameter`, a random value will be sampled from that parameter per image.

- **keep_size** (*bool, optional*) – After padding, the result image will usually have a different height/width compared to the original input image. If this parameter is set to `True`, then the padded image will be resized to the input image's size, i.e. the augmenter's output shape is always identical to the input shape.
- **sample_independently** (*bool, optional*) – If `False` *and* the values for *px/percent* result in exactly *one* probability distribution for all image sides, only one single value will be sampled from that probability distribution and used for all sides. I.e. the pad amount then is the same for all sides. If `True`, four values will be sampled independently, one per side.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Pad(px=(0, 10))
```

Pad each side by a random pixel value sampled uniformly per image and side from the discrete interval `[0..10]`. The padding happens by zero-padding, i.e. it adds black pixels (default setting).

```
>>> aug = iaa.Pad(px=(0, 10), pad_mode="edge")
```

Pad each side by a random pixel value sampled uniformly per image and side from the discrete interval `[0..10]`. The padding uses the `edge` mode from numpy's pad function, i.e. the pixel colors around the image sides are repeated.

```
>>> aug = iaa.Pad(px=(0, 10), pad_mode=["constant", "edge"])
```

Similar to the previous example, but uses zero-padding (`constant`) for half of the images and `edge` padding for the other half.

```
>>> aug = iaa.Pad(px=(0, 10), pad_mode=ia.ALL, pad_cval=(0, 255))
```

Similar to the previous example, but uses any available padding mode. In case the padding mode ends up being `constant` or `linear_ramp`, and random intensity is uniformly sampled (once per image) from the discrete interval `[0..255]` and used as the intensity of the new pixels.

```
>>> aug = iaa.Pad(px=(0, 10), sample_independently=False)
```

Pad each side by a random pixel value sampled uniformly once per image from the discrete interval `[0..10]`. Each sampled value is used for *all* sides of the corresponding image.

```
>>> aug = iaa.Pad(px=(0, 10), keep_size=False)
```

Pad each side by a random pixel value sampled uniformly per image and side from the discrete interval `[0..10]`. Afterwards, do **not** resize the padded image back to the input image's size. This will increase the image's height and width by a maximum of `20` pixels.

```
>>> aug = iaa.Pad(px=((0, 10), (0, 5), (0, 10), (0, 5)))
```

Pad the top and bottom by a random pixel value sampled uniformly from the discrete interval `[0..10]`. Pad the left and right analogously by a random value sampled from `[0..5]`. Each value is always sampled independently.

```
>>> aug = iaa.Pad(percent=(0, 0.1))
```

Pad each side by a random fraction sampled uniformly from the continuous interval `[0.0, 0.10]`. The fraction is sampled once per image and side. E.g. a sampled fraction of `0.1` for the top side would pad by `0.1*H`, where `H` is the height of the input image.

```
>>> aug = iaa.Pad(
>>>     percent=([0.05, 0.1], [0.05, 0.1], [0.05, 0.1], [0.05, 0.1]))
```

Pads each side by either `5%` or `10%`. The values are sampled once per side and image.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.

Continued on next page

Table 143 – continued from previous page

copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.size.**PadToFixedSize**(*width*, *height*, *pad_mode='constant'*, *pad_cval=0*, *position='uniform'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Pad images to a predefined minimum width and/or height.

If images are already at the minimum width/height or are larger, they will not be padded. Note that this also means that images will not be cropped if they exceed the required width/height.

The augmenter randomly decides per image how to distribute the required padding amounts over the image axis. E.g. if 2px have to be padded on the left or right to reach the required width, the augmenter will sometimes add 2px to the left and 0px to the right, sometimes add 2px to the right and 0px to the left and sometimes add 1px to both sides. Set *position* to `center` to prevent that.

dtype support:

```
See :func:`imgaug.imgaug.pad`.
```

Parameters

- **width** (*int*) – Pad images up to this minimum width.
- **height** (*int*) – Pad images up to this minimum height.

- **pad_mode** (*imgaug.ALL or str or list of str or imgaug.parameters.StochasticParameter, optional*) – See `imgaug.augmenters.size.CropAndPad.__init__()`.
- **pad_cval** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – See `imgaug.augmenters.size.CropAndPad.__init__()`.
- **position** (*{'uniform', 'normal', 'center', 'left-top', 'left-center', 'left-bottom', 'center-top', 'center-center', 'center-bottom', 'right-top', 'right-center', 'right-bottom'} or tuple of float or StochasticParameter or tuple of StochasticParameter, optional*) – Sets the center point of the padding, which determines how the required padding amounts are distributed to each side. For a `tuple (a, b)`, both `a` and `b` are expected to be in range `[0.0, 1.0]` and describe the fraction of padding applied to the left/right (low/high values for `a`) and the fraction of padding applied to the top/bottom (low/high values for `b`). A padding position at `(0.5, 0.5)` would be the center of the image and distribute the padding equally to all sides. A padding position at `(0.0, 1.0)` would be the left-bottom and would apply 100% of the required padding to the bottom and left sides of the image so that the bottom left corner becomes more and more the new image center (depending on how much is padded).
  - If string `uniform` then the share of padding is randomly and uniformly distributed over each side. Equivalent to `(Uniform(0.0, 1.0), Uniform(0.0, 1.0))`.
  - If string `normal` then the share of padding is distributed based on a normal distribution, leading to a focus on the center of the images. Equivalent to `(Clip(Normal(0.5, 0.45/2), 0, 1), Clip(Normal(0.5, 0.45/2), 0, 1))`.
  - If string `center` then center point of the padding is identical to the image center. Equivalent to `(0.5, 0.5)`.
  - If a string matching regex `^(left|center|right)-(top|center|bottom)$`, e.g. `left-top` or `center-bottom` then sets the center point of the padding to the X-Y position matching that description.
  - If a tuple of float, then expected to have exactly two entries between `0.0` and `1.0`, which will always be used as the combination the position matching (x, y) form.
  - If a `StochasticParameter`, then that parameter will be queried once per call to `augment_*()` to get `Nx2` center positions in `(x, y)` form (with `N` the number of images).
  - If a `tuple` of `StochasticParameter`, then expected to have exactly two entries that will both be queried per call to `augment_*()`, each for `(N,)` values, to get the center positions. First parameter is used for `x` coordinates, second for `y` coordinates.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.PadToFixedSize(width=100, height=100)
```

For image sides smaller than 100 pixels, pad to 100 pixels. Do nothing for the other edges. The padding is randomly (uniformly) distributed over the sides, so that e.g. sometimes most of the required padding is applied to the left, sometimes to the right (analogous top/bottom).

```
>>> aug = iaa.PadToFixedSize(width=100, height=100, position="center")
```

For image sides smaller than 100 pixels, pad to 100 pixels. Do nothing for the other image sides. The padding is always equally distributed over the left/right and top/bottom sides.

```
>>> aug = iaa.PadToFixedSize(width=100, height=100, pad_mode=ia.ALL)
```

For image sides smaller than 100 pixels, pad to 100 pixels and use any possible padding mode for that. Do nothing for the other image sides. The padding is always equally distributed over the left/right and top/bottom sides.

```
>>> aug = iaa.Sequential([
>>>     iaa.PadToFixedSize(width=100, height=100),
>>>     iaa.CropToFixedSize(width=100, height=100)
>>> ])
```

Pad images smaller than 100x100 until they reach 100x100. Analogously, crop images larger than 100x100 until they reach 100x100. The output images therefore have a fixed size of 100x100.

Methods

__call__(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
augment(self[, return_batch, hooks])	Augment a batch.
augment_batch(self, batch[, hooks])	Augment a single batch.
augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ...[, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.

Continued on next page

Table 144 – continued from previous page

find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.size.**Resize**(*size*, *interpolation='cubic'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Augmenter that resizes images to specified heights and widths.

dtype support:

```
See :func:`imgaug.imgaug.imresize_many_images`.
```

Parameters

- **size** (*'keep' or int or float or tuple of int or tuple of float or list of int or list of float or imgaug.parameters.StochasticParameter or dict*) –

  The new size of the images.

  - If this has the string value `keep`, the original height and width values will be kept (image is not resized).
  - If this is an `int`, this value will always be used as the new height and width of the images.
  - If this is a `float` `v`, then per image the image's height `H` and width `W` will be changed to `H*v` and `W*v`.
  - If this is a `tuple`, it is expected to have two entries `(a, b)`. If at least one of these are `float` s, a value will be sampled from range `[a, b]` and used as the `float` value to resize the image (see above). If both are `int` s, a value will be sampled from the discrete range `[a..b]` and used as the integer value to resize the image (see above).
  - If this is a `list`, a random value from the `list` will be picked to resize the image. All values in the `list` must be `int` s or `float` s (no mixture is possible).

  - If this is a `StochasticParameter`, then this parameter will first be queried once per image. The resulting value will be used for both height and width.
  - If this is a `dict`, it may contain the keys `height` and `width` or the keys `shorter-side` and `longer-side`. Each key may have the same datatypes as above and describes the scaling on x and y-axis or the shorter and longer axis, respectively. Both axis are sampled independently. Additionally, one of the keys may have the value `keep-aspect-ratio`, which means that the respective side of the image will be resized so that the original aspect ratio is kept. This is useful when only resizing one image size by a pixel value (e.g. resize images to a height of `64` pixels and resize the width so that the overall aspect ratio is maintained).

- **interpolation** (*imgaug.ALL or int or str or list of int or list of str or imgaug.parameters.StochasticParameter, optional*) –

  Interpolation to use.

  - If `imgaug.ALL`, then a random interpolation from `nearest`, `linear`, `area` or `cubic` will be picked (per image).
  - If `int`, then this interpolation will always be used. Expected to be any of the following: `cv2.INTER_NEAREST`, `cv2.INTER_LINEAR`, `cv2.INTER_AREA`, `cv2.INTER_CUBIC`
  - If string, then this interpolation will always be used. Expected to be any of the following: `nearest`, `linear`, `area`, `cubic`
  - If `list` of `int` / `str`, then a random one of the values will be picked per image as the interpolation.
  - If a `StochasticParameter`, then this parameter will be queried per image and is expected to return an `int` or `str`.

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Resize(32)
```

Resize all images to `32x32` pixels.

```
>>> aug = iaa.Resize(0.5)
```

Resize all images to `50` percent of their original size.

```
>>> aug = iaa.Resize((16, 22))
```

Resize all images to a random height and width within the discrete interval `[16..22]` (uniformly sampled per image).

```
>>> aug = iaa.Resize((0.5, 0.75))
```

Resize all any input image so that its height (`H`) and width (`W`) become `H*v` and `W*v`, where `v` is uniformly sampled from the interval `[0.5, 0.75]`.

```
>>> aug = iaa.Resize([16, 32, 64])
```

Resize all images either to `16x16`, `32x32` or `64x64` pixels.

```
>>> aug = iaa.Resize({"height": 32})
```

Resize all images to a height of `32` pixels and keeps the original width.

```
>>> aug = iaa.Resize({"height": 32, "width": 48})
```

Resize all images to a height of `32` pixels and a width of `48`.

```
>>> aug = iaa.Resize({"height": 32, "width": "keep-aspect-ratio"})
```

Resize all images to a height of `32` pixels and resizes the x-axis (width) so that the aspect ratio is maintained.

```
>>> aug = iaa.Resize(
>>>     {"shorter-side": 224, "longer-side": "keep-aspect-ratio"})
```

Resize all images to a height/width of `224` pixels, depending on which axis is shorter and resize the other axis so that the aspect ratio is maintained.

```
>>> aug = iaa.Resize({"height": (0.5, 0.75), "width": [16, 32, 64]})
```

Resize all images to a height of `H*v`, where `H` is the original height and `v` is a random value sampled from the interval `[0.5, 0.75]`. The width/x-axis of each image is resized to either `16` or `32` or `64` pixels.

```
>>> aug = iaa.Resize(32, interpolation=["linear", "cubic"])
```

Resize all images to `32x32` pixels. Randomly use either `linear` or `cubic` interpolation.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.

Continued on next page

Table 145 – continued from previous page

augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

imgaug.augmenters.size.**Scale**(**args*, ***kwargs*)

Deprecated. Use Resize instead. Resize has the exactly same interface as Scale.

13.29 imgaug.augmenters.weather

Augmenters that create weather effects.

Do not import directly from this file, as the categorization is not final. Use instead:

```
from imgaug import augmenters as iaa
```

and then e.g.:

```
seq = iaa.Sequential([iaa.Snowflakes()])
```

List of augmenters:

- FastSnowyLandscape
- Clouds
- Fog
- CloudLayer
- Snowflakes
- SnowflakesLayer

class imgaug.augmenters.weather.**CloudLayer**(*intensity_mean*, *intensity_freq_exponent*, *intensity_coarse_scale*, *alpha_min*, *alpha_multiplier*, *alpha_size_px_max*, *alpha_freq_exponent*, *sparsity*, *density_multiplier*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Add a single layer of clouds to an image.

dtype support:

```
* ``uint8``: yes; indirectly tested (1)
* ``uint16``: no
* ``uint32``: no
* ``uint64``: no
* ``int8``: no
* ``int16``: no
* ``int32``: no
* ``int64``: no
* ``float16``: yes; not tested
* ``float32``: yes; not tested
* ``float64``: yes; not tested
* ``float128``: yes; not tested (2)
* ``bool``: no

- (1) Indirectly tested via tests for ``Clouds`` and ``Fog``
- (2) Note that random values are usually sampled as ``int64`` or
      ``float64``, which ``float128`` images would exceed. Note also
      that random values might have to upscaled, which is done
      via :func:`imgaug.imgaug.imresize_many_images` and has its own
      limited dtype support (includes however floats up to ``64bit``).
```

Parameters

- **intensity_mean** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Mean intensity of the clouds (i.e. mean color). Recommended to be in the interval `[190, 255]`.
  - If a `number`, then that value will always be used.
  - If a `tuple (a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`.
  - If a `list`, then a random value will be sampled from that `list` per image.

  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.

- **intensity_freq_exponent** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Exponent of the frequency noise used to add fine intensity to the mean intensity. Recommended to be in the interval `[-2.5, -1.5]`. See `imgaug.parameters.FrequencyNoise.__init__()` for details.

- **intensity_coarse_scale** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Standard deviation of the gaussian distribution used to add more localized intensity to the mean intensity. Sampled in low resolution space, i.e. affects final intensity on a coarse level. Recommended to be in the interval `(0, 10]`.

  - If a `number`, then that value will always be used.

  - If a `tuple (a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`.

  - If a `list`, then a random value will be sampled from that `list` per image.

  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.

- **alpha_min** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Minimum alpha when blending cloud noise with the image. High values will lead to clouds being "everywhere". Recommended to usually be at around `0.0` for clouds and `>0` for fog.

  - If a `number`, then that value will always be used.

  - If a `tuple (a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`.

  - If a `list`, then a random value will be sampled from that `list` per image.

  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.

- **alpha_multiplier** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Multiplier for the sampled alpha values. High values will lead to denser clouds wherever they are visible. Recommended to be in the interval `[0.3, 1.0]`. Note that this parameter currently overlaps with *density_multiplier*, which is applied a bit later to the alpha mask.

  - If a `number`, then that value will always be used.

  - If a `tuple (a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`.

  - If a `list`, then a random value will be sampled from that `list` per image.

  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.

- **alpha_size_px_max** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Controls the image size at which the alpha mask is sampled. Lower values will lead to coarser alpha masks and hence larger clouds (and empty areas). See `imgaug.parameters.FrequencyNoise.__init__()` for details.

- **alpha_freq_exponent** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Exponent of the frequency noise used to sample the alpha mask. Similarly to *alpha_size_max_px*, lower values will lead to coarser

alpha patterns. Recommended to be in the interval `[-4.0, -1.5]`. See `imgaug.parameters.FrequencyNoise.__init__()` for details.

- **sparsity** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Exponent applied late to the alpha mask. Lower values will lead to coarser cloud patterns, higher values to finer patterns. Recommended to be somewhere around `1.0`. Do not deviate far from that value, otherwise the alpha mask might get weird patterns with sudden fall-offs to zero that look very unnatural.
  - If a `number`, then that value will always be used.
  - If a `tuple (a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.
- **density_multiplier** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Late multiplier for the alpha mask, similar to *alpha_multiplier*. Set this higher to get "denser" clouds wherever they are visible. Recommended to be around `[0.5, 1.5]`.
  - If a `number`, then that value will always be used.
  - If a `tuple (a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.

Continued on next page

Table 146 – continued from previous page

augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

draw_on_image	
generate_maps	
get_parameters	

draw_on_image(*self*, *image*, *random_state*)

generate_maps(*self*, *image*, *random_state*)

get_parameters(*self*)

class imgaug.augmenters.weather.**Clouds**(*name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.SomeOf*

Add clouds to images.

This is a wrapper around *imgaug.augmenters.weather.CloudLayer*. It executes 1 to 2 layers per image, leading to varying densities and frequency patterns of clouds.

This augmenter seems to be fairly robust w.r.t. the image size. Tested with `96x128`, `192x256` and `960x1280`.

dtype support:

```
* ``uint8``: yes; tested
* ``uint16``: no (1)
* ``uint32``: no (1)
* ``uint64``: no (1)
* ``int8``: no (1)
* ``int16``: no (1)
* ``int32``: no (1)
* ``int64``: no (1)
* ``float16``: no (1)
* ``float32``: no (1)
* ``float64``: no (1)
* ``float128``: no (1)
* ``bool``: no (1)

- (1) Parameters of this augmenter are optimized for the value range
      of ``uint8``. While other dtypes may be accepted, they will lead
      to images augmented in ways inappropriate for the respective
      dtype.
```

Parameters

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Clouds()
```

Create an augmenter that adds clouds to images.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`add`(self, augmenter)	Add an augmenter to the list of child augmenters.
`append`(self, object, /)	Append object to the end of the list.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.

Continued on next page

Table 147 – continued from previous page

augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ...[, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
clear(self, /)	Remove all items from list.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
count(self, value, /)	Return number of occurrences of value.
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
extend(self, iterable, /)	Extend list by appending elements from the iterable.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
index(self, value[, start, stop])	Return first index of value.
insert(self, index, object, /)	Insert object before index.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
pop(self[, index])	Remove and return item at index (default last).
remove(self, value, /)	Remove first occurrence of value.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
reverse(self, /)	Reverse *IN PLACE*.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
sort(self, /, *[, key, reverse])	Stable sort *IN PLACE*.

Continued on next page

Table 147 – continued from previous page

`to_deterministic`(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.weather.**FastSnowyLandscape**(*lightness_threshold=(100, 255)*, *lightness_multiplier=(1.0, 4.0)*, *from_colorspace='RGB'*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Convert non-snowy landscapes to snowy ones.

This augmenter expects to get an image that roughly shows a landscape.

This augmenter is based on the method proposed in https://medium.freecodecamp.org/image-augmentation-make-it-rain-make-it-snow-how-to-modify-a-photo-with-machine-learning-163c0cb3843f?gi=bca4a13e634c

dtype support:

```
* ``uint8``: yes; fully tested
* ``uint16``: no (1)
* ``uint32``: no (1)
* ``uint64``: no (1)
* ``int8``: no (1)
* ``int16``: no (1)
* ``int32``: no (1)
* ``int64``: no (1)
* ``float16``: no (1)
* ``float32``: no (1)
* ``float64``: no (1)
* ``float128``: no (1)
* ``bool``: no (1)

- (1) This augmenter is based on a colorspace conversion to HLS.
      Hence, only RGB ``uint8`` inputs are sensible.
```

Parameters

- **lightness_threshold** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – All pixels with lightness in HLS colorspace that is below this value will have their lightness increased by *lightness_multiplier*.
  - If a `number`, then that value will always be used.
  - If a `tuple (a, b)`, then a value will be uniformly sampled per image from the discrete interval `[a..b]`.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.
- **lightness_multiplier** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter, optional*) – Multiplier for pixel's lightness value in HLS colorspace. Affects all pixels selected via *lightness_threshold*.

  - If a `number`, then that value will always be used.
  - If a `tuple (a, b)`, then a value will be uniformly sampled per image from the discrete interval `[a..b]`.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.

- **from_colorspace** (*str, optional*) – The source colorspace of the input images. See `imgaug.augmenters.color.ChangeColorspace.__init__()`.
- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.FastSnowyLandscape(
>>>     lightness_threshold=140,
>>>     lightness_multiplier=2.5
>>> )
```

Search for all pixels in the image with a lightness value in HLS colorspace of less than `140` and increase their lightness by a factor of `2.5`.

```
>>> aug = iaa.FastSnowyLandscape(
>>>     lightness_threshold=[128, 200],
>>>     lightness_multiplier=(1.5, 3.5)
>>> )
```

Search for all pixels in the image with a lightness value in HLS colorspace of less than `128` or less than `200` (one of these values is picked per image) and multiply their lightness by a factor of `x` with `x` being sampled from `uniform(1.5, 3.5)` (once per image).

```
>>> aug = iaa.FastSnowyLandscape(
>>>     lightness_threshold=(100, 255),
>>>     lightness_multiplier=(1.0, 4.0)
>>> )
```

Similar to the previous example, but the lightness threshold is sampled from `uniform(100, 255)` (per image) and the multiplier from `uniform(1.0, 4.0)` (per image). This seems to produce good and varied results.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
`remove_augmenters_inplace`(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
`reseed`(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
`show_grid`(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
`to_deterministic`(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

get_parameters(*self*)

class imgaug.augmenters.weather.**Fog**(*name=None*, *deterministic=False*, *random_state=None*)
Bases: *imgaug.augmenters.weather.CloudLayer*

Add fog to images.

This is a wrapper around *imgaug.augmenters.weather.CloudLayer*. It executes a single layer per image with a configuration leading to fairly dense clouds with low-frequency patterns.

This augmenter seems to be fairly robust w.r.t. the image size. Tested with `96x128`, `192x256` and `960x1280`.

dtype support:

```
* ``uint8``: yes; tested
* ``uint16``: no (1)
* ``uint32``: no (1)
* ``uint64``: no (1)
* ``int8``: no (1)
* ``int16``: no (1)
* ``int32``: no (1)
* ``int64``: no (1)
* ``float16``: no (1)
* ``float32``: no (1)
* ``float64``: no (1)
* ``float128``: no (1)
* ``bool``: no (1)

- (1) Parameters of this augmenter are optimized for the value range
      of ``uint8``. While other dtypes may be accepted, they will lead
      to images augmented in ways inappropriate for the respective
      dtype.
```

Parameters

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Fog()
```

Create an augmenter that adds fog to images.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.

Continued on next page

Table 149 – continued from previous page

augment(self[, return_batch, hooks])	Augment a batch.
augment_batch(self, batch[, hooks])	Augment a single batch.
augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ...[, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
to_deterministic(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

draw_on_image	
generate_maps	
get_parameters	

class imgaug.augmenters.weather.**Snowflakes**(*density=(0.005, 0.075), density_uniformity=(0.3, 0.9), flake_size=(0.2, 0.7), flake_size_uniformity=(0.4, 0.8), angle=(-30, 30), speed=(0.007, 0.03), name=None, deterministic=False, random_state=None*)

Bases: *imgaug.augmenters.meta.SomeOf*

Add falling snowflakes to images.

This is a wrapper around *imgaug.augmenters.weather.SnowflakesLayer*. It executes 1 to 3 layers per image.

dtype support:

```
* ``uint8``: yes; tested
* ``uint16``: no (1)
* ``uint32``: no (1)
* ``uint64``: no (1)
* ``int8``: no (1)
* ``int16``: no (1)
* ``int32``: no (1)
* ``int64``: no (1)
* ``float16``: no (1)
* ``float32``: no (1)
* ``float64``: no (1)
* ``float128``: no (1)
* ``bool``: no (1)

- (1) Parameters of this augmenter are optimized for the value range
      of ``uint8``. While other dtypes may be accepted, they will lead
      to images augmented in ways inappropriate for the respective
      dtype.
```

Parameters

- **density** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Density of the snowflake layer, as a probability of each pixel in low resolution space to be a snowflake. Valid values are in the interval `[0.0, 1.0]`. Recommended to be in the interval `[0.01, 0.075]`.
  - If a `number`, then that value will always be used.
  - If a `tuple (a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.
- **density_uniformity** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Size uniformity of the snowflakes. Higher values denote more similarly sized snowflakes. Valid values are in the interval `[0.0, 1.0]`. Recommended to be around `0.5`.
  - If a `number`, then that value will always be used.
  - If a `tuple (a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`.

  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.

- **flake_size** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Size of the snowflakes. This parameter controls the resolution at which snowflakes are sampled. Higher values mean that the resolution is closer to the input image's resolution and hence each sampled snowflake will be smaller (because of the smaller pixel size).

  Valid values are in the interval `(0.0, 1.0]`. Recommended values:

  - On `96x128` a value of `(0.1, 0.4)` worked well.
  - On `192x256` a value of `(0.2, 0.7)` worked well.
  - On `960x1280` a value of `(0.7, 0.95)` worked well.

  Datatype behaviour:

  - If a `number`, then that value will always be used.
  - If a `tuple (a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.

- **flake_size_uniformity** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Controls the size uniformity of the snowflakes. Higher values mean that the snowflakes are more similarly sized. Valid values are in the interval `[0.0, 1.0]`. Recommended to be around `0.5`.

  - If a `number`, then that value will always be used.
  - If a `tuple (a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.

- **angle** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Angle in degrees of motion blur applied to the snowflakes, where `0.0` is motion blur that points straight upwards. Recommended to be in the interval `[-30, 30]`. See also `imgaug.augmenters.blur.MotionBlur.__init__()`.

  - If a `number`, then that value will always be used.
  - If a `tuple (a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.

- **speed** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Perceived falling speed of the snowflakes. This parameter controls the motion blur's kernel size. It follows roughly the form

`kernel_size = image_size * speed`. Hence, values around `1.0` denote that the motion blur should "stretch" each snowflake over the whole image.

Valid values are in the interval `[0.0, 1.0]`. Recommended values:

- On `96x128` a value of `(0.01, 0.05)` worked well.
- On `192x256` a value of `(0.007, 0.03)` worked well.
- On `960x1280` a value of `(0.001, 0.03)` worked well.

Datatype behaviour:

- If a `number`, then that value will always be used.
- If a `tuple (a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`.
- If a `list`, then a random value will be sampled from that `list` per image.
- If a `StochasticParameter`, then a value will be sampled per image from that parameter.

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.
- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Examples

```
>>> import imgaug.augmenters as iaa
>>> aug = iaa.Snowflakes(flake_size=(0.1, 0.4), speed=(0.01, 0.05))
```

Add snowflakes to small images (around `96x128`).

```
>>> aug = iaa.Snowflakes(flake_size=(0.2, 0.7), speed=(0.007, 0.03))
```

Add snowflakes to medium-sized images (around `192x256`).

```
>>> aug = iaa.Snowflakes(flake_size=(0.7, 0.95), speed=(0.001, 0.03))
```

Add snowflakes to large images (around `960x1280`).

Methods

`__call__`(self, *args, **kwargs)	Alias for *`imgaug.augmenters.meta.Augmenter.augment()`*.
`add`(self, augmenter)	Add an augmenter to the list of child augmenters.
`append`(self, object, /)	Append object to the end of the list.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.

Continued on next page

Table 150 – continued from previous page

augment_batches(self, batches[, hooks, ...])	Augment multiple batches.
augment_bounding_boxes(self, ...[, hooks])	Augment a batch of bounding boxes.
augment_heatmaps(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
augment_image(self, image[, hooks])	Augment a single image.
augment_images(self, images[, parents, hooks])	Augment a batch of images.
augment_keypoints(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
augment_line_strings(self, ...[, parents, hooks])	Augment a batch of line strings.
augment_polygons(self, polygons_on_images[, ...])	Augment a batch of polygons.
augment_segmentation_maps(self, segmaps[, ...])	Augment a batch of segmentation maps.
clear(self, /)	Remove all items from list.
copy(self)	Create a shallow copy of this Augmenter instance.
copy_random_state(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
copy_random_state_(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
count(self, value, /)	Return number of occurrences of value.
deepcopy(self)	Create a deep copy of this Augmenter instance.
draw_grid(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
extend(self, iterable, /)	Extend list by appending elements from the iterable.
find_augmenters(self, func[, parents, flat])	Find augmenters that match a condition.
find_augmenters_by_name(self, name[, regex, ...])	Find augmenter(s) by name.
find_augmenters_by_names(self, names[, ...])	Find augmenter(s) by names.
get_all_children(self[, flat])	Get all children of this augmenter as a list.
get_children_lists(self)	Get a list of lists of children of this augmenter.
index(self, value[, start, stop])	Return first index of value.
insert(self, index, object, /)	Insert object before index.
localize_random_state(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
localize_random_state_(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
pool(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
pop(self[, index])	Remove and return item at index (default last).
remove(self, value, /)	Remove first occurrence of value.
remove_augmenters(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
remove_augmenters_inplace(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
reseed(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
reverse(self, /)	Reverse *IN PLACE*.
show_grid(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
sort(self, /, *[, key, reverse])	Stable sort *IN PLACE*.

Continued on next page

Table 150 – continued from previous page

`to_deterministic`(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

get_parameters

class imgaug.augmenters.weather.**SnowflakesLayer**(*density*, *density_uniformity*, *flake_size*, *flake_size_uniformity*, *angle*, *speed*, *blur_sigma_fraction*, *blur_sigma_limits=(0.5, 3.75)*, *name=None*, *deterministic=False*, *random_state=None*)

Bases: *imgaug.augmenters.meta.Augmenter*

Add a single layer of falling snowflakes to images.

dtype support:

```
* ``uint8``: yes; indirectly tested (1)
* ``uint16``: no
* ``uint32``: no
* ``uint64``: no
* ``int8``: no
* ``int16``: no
* ``int32``: no
* ``int64``: no
* ``float16``: no
* ``float32``: no
* ``float64``: no
* ``float128``: no
* ``bool``: no

- (1) indirectly tested via tests for ``Snowflakes``
```

Parameters

- **density** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Density of the snowflake layer, as a probability of each pixel in low resolution space to be a snowflake. Valid values are in the interval `[0.0, 1.0]`. Recommended to be in the interval `[0.01, 0.075]`.
  - If a `number`, then that value will always be used.
  - If a `tuple (a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.
- **density_uniformity** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Size uniformity of the snowflakes. Higher values denote more similarly sized snowflakes. Valid values are in the interval `[0.0, 1.0]`. Recommended to be around `0.5`.
  - If a `number`, then that value will always be used.

  - If a `tuple (a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.

- **flake_size** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Size of the snowflakes. This parameter controls the resolution at which snowflakes are sampled. Higher values mean that the resolution is closer to the input image's resolution and hence each sampled snowflake will be smaller (because of the smaller pixel size).

  Valid values are in the interval `(0.0, 1.0]`. Recommended values:

  - On 96x128 a value of `(0.1, 0.4)` worked well.
  - On 192x256 a value of `(0.2, 0.7)` worked well.
  - On 960x1280 a value of `(0.7, 0.95)` worked well.

  Datatype behaviour:

  - If a `number`, then that value will always be used.
  - If a `tuple (a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.

- **flake_size_uniformity** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Controls the size uniformity of the snowflakes. Higher values mean that the snowflakes are more similarly sized. Valid values are in the interval `[0.0, 1.0]`. Recommended to be around `0.5`.

  - If a `number`, then that value will always be used.
  - If a `tuple (a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.

- **angle** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Angle in degrees of motion blur applied to the snowflakes, where `0.0` is motion blur that points straight upwards. Recommended to be in the interval `[-30, 30]`. See also `imgaug.augmenters.blur.MotionBlur.__init__()`.

  - If a `number`, then that value will always be used.
  - If a `tuple (a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`.
  - If a `list`, then a random value will be sampled from that `list` per image.
  - If a `StochasticParameter`, then a value will be sampled per image from that parameter.

- **speed** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Perceived falling speed of the snowflakes. This parameter controls the motion blur's kernel size. It follows roughly the form `kernel_size = image_size * speed`. Hence, values around `1.0` denote that the motion blur should "stretch" each snowflake over the whole image.

  Valid values are in the interval `[0.0, 1.0]`. Recommended values:

  – On 96x128 a value of `(0.01, 0.05)` worked well.

  – On 192x256 a value of `(0.007, 0.03)` worked well.

  – On 960x1280 a value of `(0.001, 0.03)` worked well.

  Datatype behaviour:

  – If a `number`, then that value will always be used.

  – If a `tuple (a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`.

  – If a `list`, then a random value will be sampled from that `list` per image.

  – If a `StochasticParameter`, then a value will be sampled per image from that parameter.

- **blur_sigma_fraction** (*number or tuple of number or list of number or imgaug.parameters.StochasticParameter*) – Standard deviation (as a fraction of the image size) of gaussian blur applied to the snowflakes. Valid values are in the interval `[0.0, 1.0]`. Recommended to be in the interval `[0.0001, 0.001]`. May still require tinkering based on image size.

  – If a `number`, then that value will always be used.

  – If a `tuple (a, b)`, then a value will be uniformly sampled per image from the interval `[a, b]`.

  – If a `list`, then a random value will be sampled from that `list` per image.

  – If a `StochasticParameter`, then a value will be sampled per image from that parameter.

- **blur_sigma_limits** (*tuple of float, optional*) – Controls allowed min and max values of *blur_sigma_fraction* after(!) multiplication with the image size. First value is the minimum, second value is the maximum. Values outside of that range will be clipped to be within that range. This prevents extreme values for very small or large images.

- **name** (*None or str, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **deterministic** (*bool, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

- **random_state** (*None or int or imgaug.random.RNG or numpy.random.Generator or numpy.random.bit_generator.BitGenerator or numpy.random.SeedSequence or numpy.random.RandomState, optional*) – See `imgaug.augmenters.meta.Augmenter.__init__()`.

Methods

`__call__`(self, *args, **kwargs)	Alias for *imgaug.augmenters.meta.Augmenter.augment()*.
`augment`(self[, return_batch, hooks])	Augment a batch.
`augment_batch`(self, batch[, hooks])	Augment a single batch.
`augment_batches`(self, batches[, hooks, ...])	Augment multiple batches.
`augment_bounding_boxes`(self, ...[, hooks])	Augment a batch of bounding boxes.
`augment_heatmaps`(self, heatmaps[, parents, ...])	Augment a batch of heatmaps.
`augment_image`(self, image[, hooks])	Augment a single image.
`augment_images`(self, images[, parents, hooks])	Augment a batch of images.
`augment_keypoints`(self, keypoints_on_images)	Augment a batch of keypoints/landmarks.
`augment_line_strings`(self, ...[, parents, hooks])	Augment a batch of line strings.
`augment_polygons`(self, polygons_on_images[, ...])	Augment a batch of polygons.
`augment_segmentation_maps`(self, segmaps[, ...])	Augment a batch of segmentation maps.
`copy`(self)	Create a shallow copy of this Augmenter instance.
`copy_random_state`(self, source[, recursive, ...])	Copy the RNGs from a source augmenter sequence.
`copy_random_state_`(self, source[, ...])	Copy the RNGs from a source augmenter sequence (in-place).
`deepcopy`(self)	Create a deep copy of this Augmenter instance.
`draw_grid`(self, images, rows, cols)	Augment images and draw the results as a single grid-like image.
`find_augmenters`(self, func[, parents, flat])	Find augmenters that match a condition.
`find_augmenters_by_name`(self, name[, regex, ...])	Find augmenter(s) by name.
`find_augmenters_by_names`(self, names[, ...])	Find augmenter(s) by names.
`get_all_children`(self[, flat])	Get all children of this augmenter as a list.
`get_children_lists`(self)	Get a list of lists of children of this augmenter.
`localize_random_state`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`localize_random_state_`(self[, recursive])	Assign augmenter-specific RNGs to this augmenter and its children.
`pool`(self[, processes, maxtasksperchild, seed])	Create a pool used for multicore augmentation.
`remove_augmenters`(self, func[, copy, ...])	Remove this augmenter or children that match a condition.
`remove_augmenters_inplace`(self, func[, parents])	Remove in-place children of this augmenter that match a condition.
`reseed`(self[, random_state, deterministic_too])	Reseed this augmenter and all of its children.
`show_grid`(self, images, rows, cols)	Augment images and plot the results as a single grid-like image.
`to_deterministic`(self[, n])	Convert this augmenter from a stochastic to a deterministic one.

draw_on_image	
get_parameters	

draw_on_image(*self*, *image*, *random_state*)

get_parameters(*self*)

See modindex for API.

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

Python Module Index

i

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U