interpret-segmentation

Jun 13, 2019

Contents

interpret-segmentation is a one-stop shop for the interpretability of image segmentation models. This code was extracted from the code of my bachelor thesis: https://github.com/andef4/thesis-code.

The PDF of the thesis is available here: https://github.com/andef4/thesis-doc/releases/download/release/thesis.pdf. It contains detailed explanations of the methods used here.

The following methods are currently implemented:

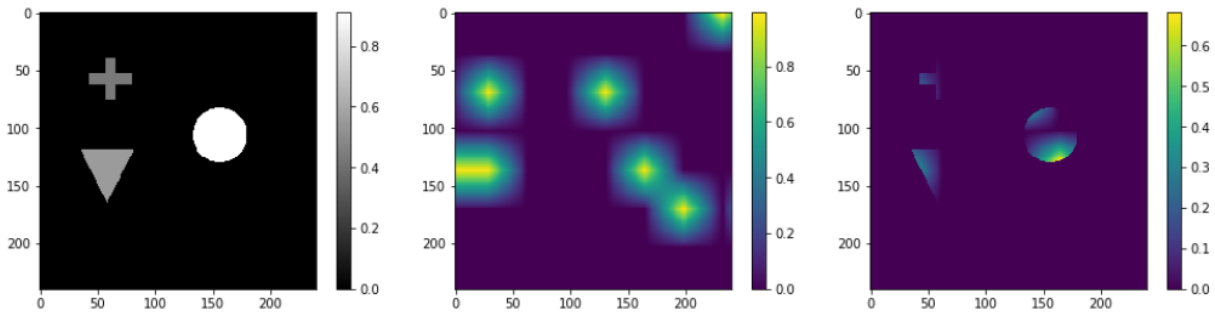
CHAPTER 1

RISE

1.1 Introduction

RISE (Randomized Input Sampling for Explanation) is a black box interpretability method built for image classification tasks. The original paper is available here: https://arxiv.org/abs/1806.07421.

RISE generates masks that are applied to the input images by multiplying the mask with the input image pixel values:



(a) Original image (b) RISE mask (c) RISE mask applied to original image by multiplication

The modified images are passed through the neural network and the classification score for a specific class are recorded. A high classification score for a class on a modified input image means that the pixels preserved by the mask are important for the classification.

To visualize the results, the classification scores and masks are summed up and converted into a saliency map.

1.2 Modifications for image segmentation interpretability

RISE was built for image classification tasks. To make it work with segmentation, we handle every pixel of the output segment as if they are their own distinct class. We let RISE generate a saliency map for every one of these pixels and

then merge the generated saliency maps. Currently, the RISEResult class supports two merge methods: `max()` and `mean()`.

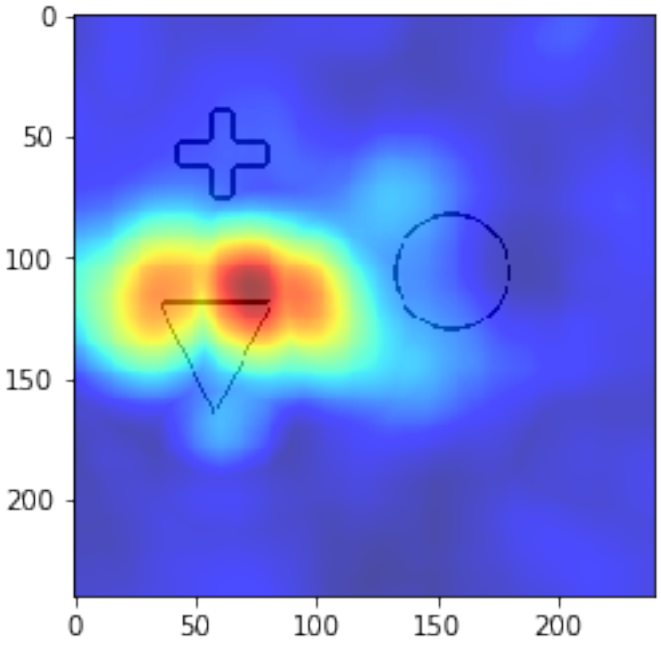


Fig. 1: Saliency map generated using the max function overlaid on a testnet input image.

1.3 Example

```
from interpret_segmentation.rise import SegmentationRISE
import torch
import matplotlib.pyplot as plt
from pathlib import Path

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# a PyTorch model
model = ...

# a PyTorch dataset
dataset = ...

# ground truth segment (PyTorch 2D tensor)
segment = ...

# input image  (PyTorch 2D tensor)
image = ...

# initialize the explainer with image width and height
explainer = SegmentationRISE(model, (240, 240), device)

# load or generate RISE masks
masks_path = Path('rise_masks.npy')
```

(continues on next page)

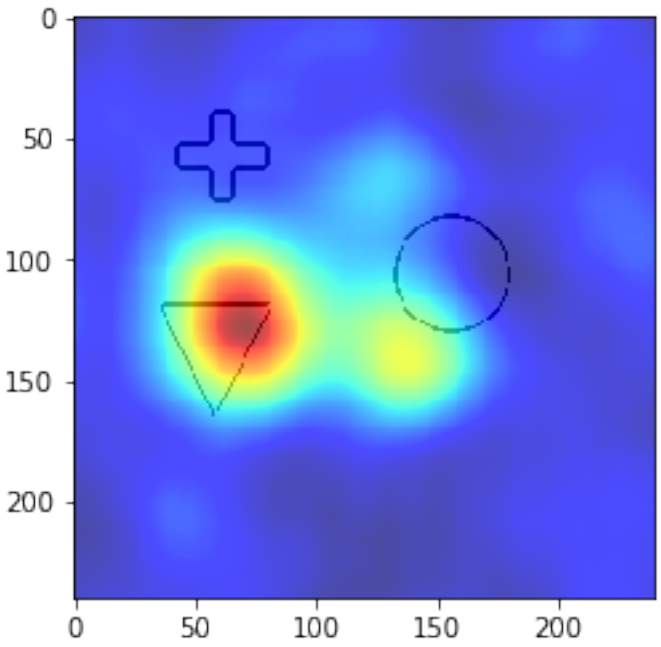


Fig. 2: Saliency map generated using the mean function overlaid on a testnet input image.

(continued from previous page)

```
if not masks_path.exists():
    explainer.generate_masks(N=3000, s=8, p1=0.1, savepath=masks_path)
else:
    explainer.load_masks(masks_path)

# generate the saliency map
with torch.set_grad_enabled(False):
    result = explainer(image)

rise_max = result.max()
plt.imshow(rise_max)
plt.show()

rise_mean = result.mean()
plt.imshow(rise_mean)
plt.show()
```

1.4 Class documentation

CHAPTER 2

Hausdorff Distance Masks

2.1 Introduction

Hausdorff Distance Masks is a new method developed for the interpretability of image segmentation models. Like RISE, it is a black box method. The output of the method has a higher resolution than RISE and is more accurate.

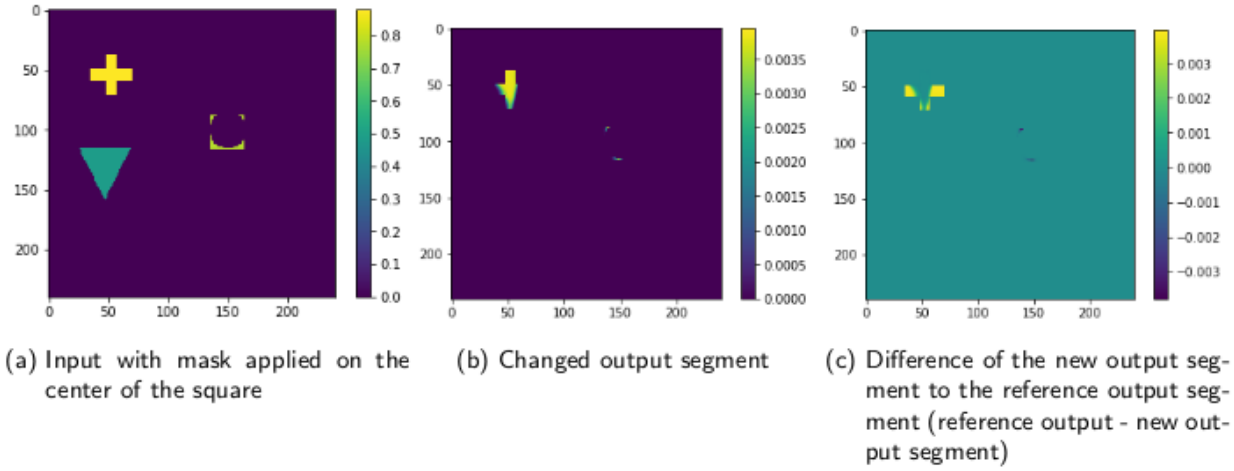
2.2 How does it work?

The first part of the algorithm is the occlusion of parts of the input image. We iterate over the image in a linear fashion, from left to right and from top to bottom, based on a pixel offset between every row and column defined as a parameter of the algorithm. For every position that is encountered, we create a new image. On this image, we draw a filled black circle at the specific position.

The images with the masks applied from above are then passed through the neural networks. The output segmentation may not change or only change slightly when the mask occludes an unimportant part of the image. Applying the mask on important parts of the image can change the segmentation output significantly.

To asses how big the change of the segmentation output is, we use the Hausdorff distance function between the new segmentation and the ground truth.

To visualize all the distances from the output of the masked image, a new blank image with the same size as the input image is generated. Next, we iterate over all the positions where masks have been applied to the input image. Each position has an associated Hausdorff distance which represents the distance of the output segment generated by the masked image and the ground truth segment. At each position, we draw a circle with the same diameter as used when generating the mask. The color used to fill this circle represents the Hausdorff distance between the output segment generated by placing a circle at this exact position and the ground truth segment. The color map is scaled to the minimum and maximum Hausdorff distance encountered on all positions.



(a) Input with mask applied on the center of the square

(b) Changed output segment

(c) Difference of the new output segment to the reference output segment (reference output - new output segment)

Fig. 1: Applying the mask on center of the square (a) significantly changes the segment output (b) of the neural network. The network even includes a part of the square in the output segment.

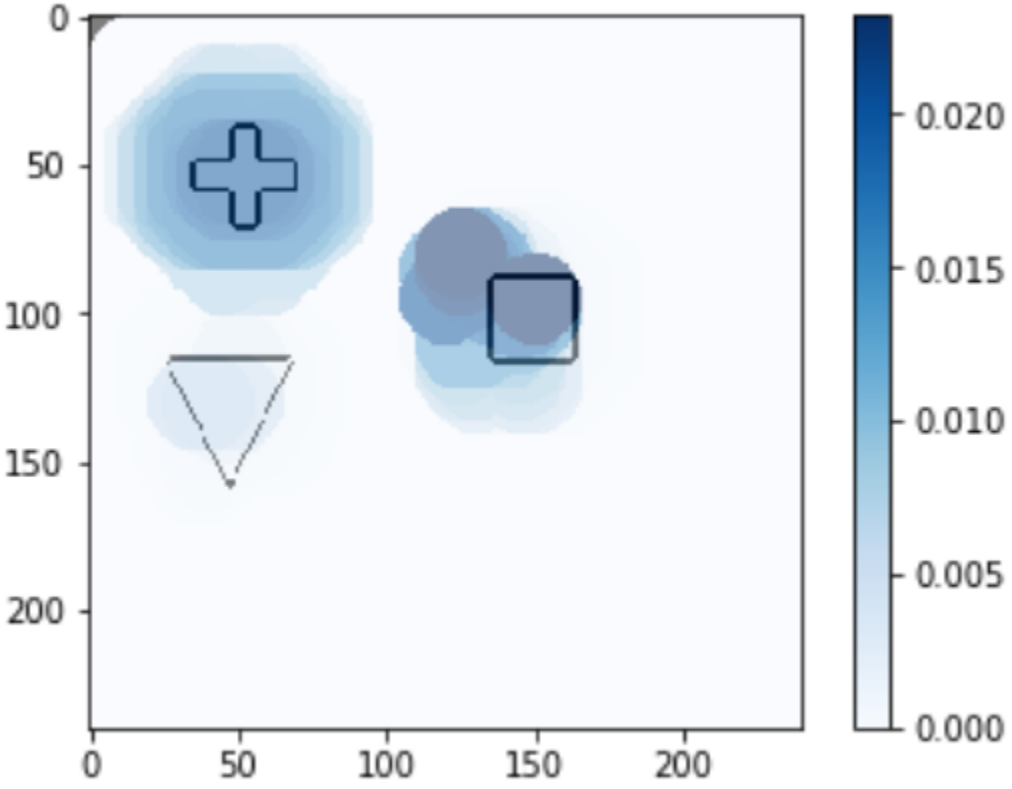


Fig. 2: Visualization of all Hausdorff distances corresponding to a mask at the same position. Intensity of the circle color is based on the Hausdorff distance at this position. The input image was processed with the canny edge detector.

2.3 Example

```
from interpret_segmentation import hdm
import torch
import matplotlib.pyplot as plt

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# a PyTorch model
model = ...

# a PyTorch dataset
dataset = ...

# ground truth segment (PyTorch 2D tensor)
segment = ...

# input image  (PyTorch 2D tensor)
image = ...

# initialize the explainer with image width and height
explainer = hdm.HausdorffDistanceMasks(240, 240)

# generate masks
explainer.generate_masks(circle_size=25, offset=5)

# apply masks and calculate distances
result = explainer.explain(model, image, segment, device)

# generate circle map visualizations
raw = result.circle_map(hdm.RAW, color_map='Blues')
better = result.circle_map(hdm.BETTER_ONLY, color_map='Greens')
worse = result.circle_map(hdm.WORSE_ONLY, color_map='Reds')

# show with matplotlib...
plt.imshow(raw)
plt.show()

# ...or save to disk
raw.save('raw.png')
```

2.4 Class documentation

class interpret_segmentation.hdm.**HausdorffDistanceMasks**(*image_width*, *image_height*)

HausdorffDistanceMasks explainer class.

__init__(*image_width*, *image_height*)

Initialize the explainer.

Parameters

- **image_width** – Input image width
- **image_height** – Input image height

generate_masks(*circle_size*, *offset*, *normalize=False*)
Generate the masks for the explainer. A circle_size of 15 pixels and an offset of 5 pixel work good on a 240x240 image.

Parameters

- **circle_size** – Diameter in pixels of the circles drawn onto the image
- **offset** – The distance in pixel between every drawn circle
- **normalize** – Normalize generated masks to mean 0.5

explain(*model*, *image*, *segment*, *device*, *channel=-1*)
Explain a single instance with Hausdorff Distance masks. The model needs to reside on the device given as a parameter to this method.

Returns a *HDMResult* instance.

Parameters

- **model** – A PyTorch module
- **image** – The input image to be explained (2D PyTorch tensor or numpy array)
- **segment** – The ground truth segment (2D PyTorch tensor or numpy array)
- **device** – A PyTorch device
- **channel** – Channel on which the mask should be applied, -1 for all channels (default)

Returns An instance of *HDMResult*

apply_mask(*image*, *mask*)
Apply a mask on an image. By default, this does a `torch.min(image, mask)`, but can be overwritten to do something else.

Parameters

- **image** – The input image, 2D numpy array
- **mask** – The mask, 2D numpy array

Returns

calculate_distance(*output*, *segment*)
Calculate the difference between the network output and the ground truth segment. Default implementation is the Hausdorff distance, but this can be replaced by any other distance function.

Parameters

- **output** – Neural network output, 2D numpy array
- **segment** – Ground truth segment, 2D numpy array

Returns A number representing the distance between output and segment

class interpret_segmentation.hdm.**HDMResult**(*distances*, *baseline*, *image_width*, *image_height*, *circle_size*, *offset*)
Result class for the Hausdorff Distance masks algorithm. Instanced by HausdorffDistanceMasks class.

distances(*result_type*)
Returns distances as a 2D matrix. Every matrix entry corresponds to one applied mask.

- hdm.RAW: The raw Hausdorff Distance
- hdm.BETTER_ONLY: Only distances where the occlusion by the mask increased the accuracy of the output.

- hdm.WORSE_ONLY: Only distances where the occlusion by the mask decreased the accuracy of the output.

Parameters **result_type** – hdm.RAW, hdm.BETTER_ONLY, hdm.WORSE_ONLY

Returns numpy 2D matrix

circle_map(*result_type*, *color_map='Reds'*)

Generates the Hausdorff Distance Mask visualization.

- hdm.RAW: The raw Hausdorff Distance
- hdm.BETTER_ONLY: Only distances where the occlusion by the mask increased the accuracy of the output.
- hdm.WORSE_ONLY: Only distances where the occlusion by the mask decreased the accuracy of the output.

Parameters

- **result_type** – hdm.RAW, hdm.BETTER_ONLY, hdm.WORSE_ONLY
- **color_map** – A matplotlib color map

Returns PIL image

CHAPTER 3

Installation

For pip based environments:

```
pip install interpret_segmentation
```

For anaconda users:

```
conda install interpret_segmentation
```

All dependencies except pytorch and torchvisison are installed automatically. Please install pytorch and torchvision manually as described on https://pytorch.org/get-started/locally/.

CHAPTER 4

Examples

Examples how to use the two algorithms are provided in the examples subdirectory in the git repository. The examples use the testnet dataset, which was specifically built as a showcase for these algorithms.

4.1 Examples

The GitHub repository has an `examples/` folder which contains two Python scripts to show how to apply the methods in this library on a PyTorch model. The used dataset and neural network is "testnet", a simple generated segmentation dataset using the U-Net architecture. See *Testnet* for more information.

4.1.1 Run the examples

- Install interpret-segmentation into a pip virtualenv or anaconda environment
- Clone GitHub repository: `git clone https://github.com/andef4/interpret-segmentation`
- Install additional dependencies: `pip install scikit-image requests` / `conda install scikit-image requests`

The example uses the "testnet" dataset, you can download the dataset and a pretrained model by running the `examples/testnet/download.py` script. Alternatively, you can generate and train the dataset yourself with the `examples/testnet/generate.py` and `examples/testnet/train.py` scripts.

The run one of the example scripts:

- `python3 examples/hdm.py`
- `python3 examples/rise.py`

Both scripts generate PNG visualizations in the `examples` directory. The runtime of the scripts are around 30-60 seconds on a current generation high-end graphics card (GeForce 1080 Ti/RTX 2080).

4.2 Testnet

4.2.1 Dataset

The basic idea in this dataset is to show that not only the pixel data inside the segmentation region is relevant for the network, but also other parts of the image. To show this, we built a dataset where one part of the image is essential to generate a correct segmentation output, but is not contained in the segmentation output itself.

The four 2D shapes circle, square, cross and triangle are drawn onto images with the Python Imaging Library. On the left side of an image, there is always a cross and a triangle displayed. On the right side, in 50% of the cases a circle is drawn, in the other 50% a square is drawn. Depending on the shape on the right side (circle or square), one of the shapes on the left side is segmented. If the right shape is a circle, the triangle is segmented. If the right shape is a square, the cross is segmented.

4.2.2 Examples

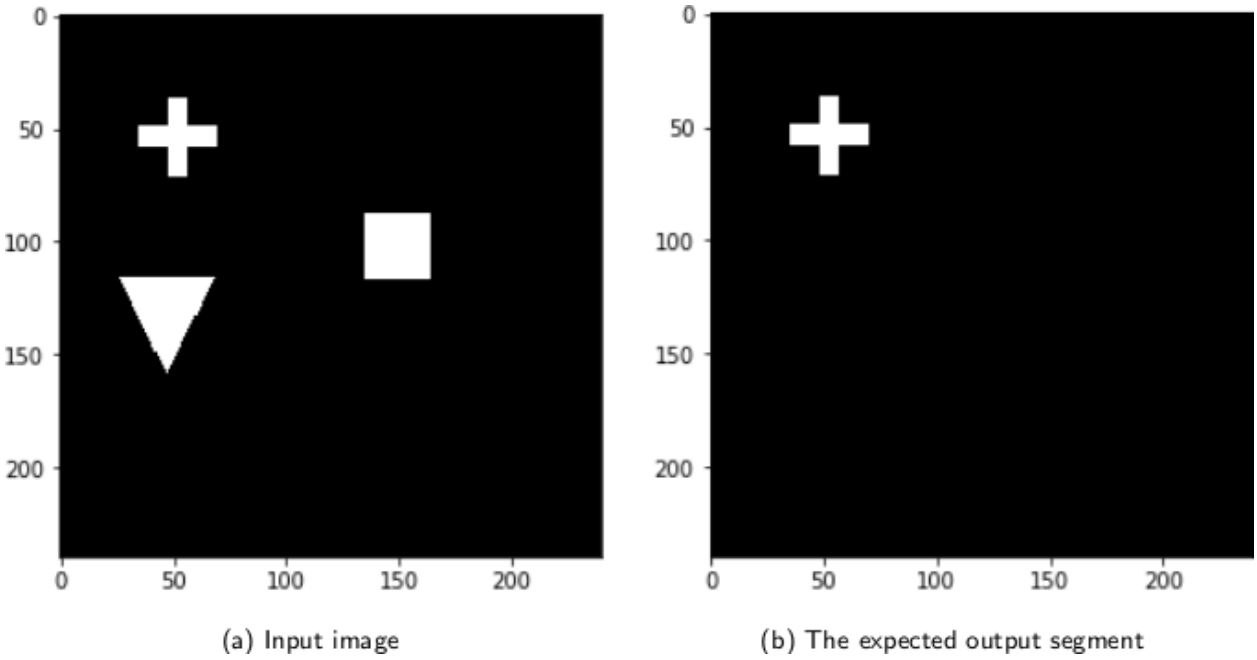


Fig. 1: The right shape in the input image is a square. Therefore the cross is segmented in the output.

A good interpretability method should not only show the importance of the segment output (circle and cross), but also on the shapes on the right (circle, cross).

4.2.3 Model

The model uses the standard U-net architecture for image segmentation tasks. The dataset generation code and the model training code is located in the `examples/testnet/` directory of the git repository.

4.2.4 Generation and Training

The data can be generated by running the `examples/testnet/generate.py` script. It will create the `examples/testnet/dataset` directory with the image files. If you have already downloaded the dataset from

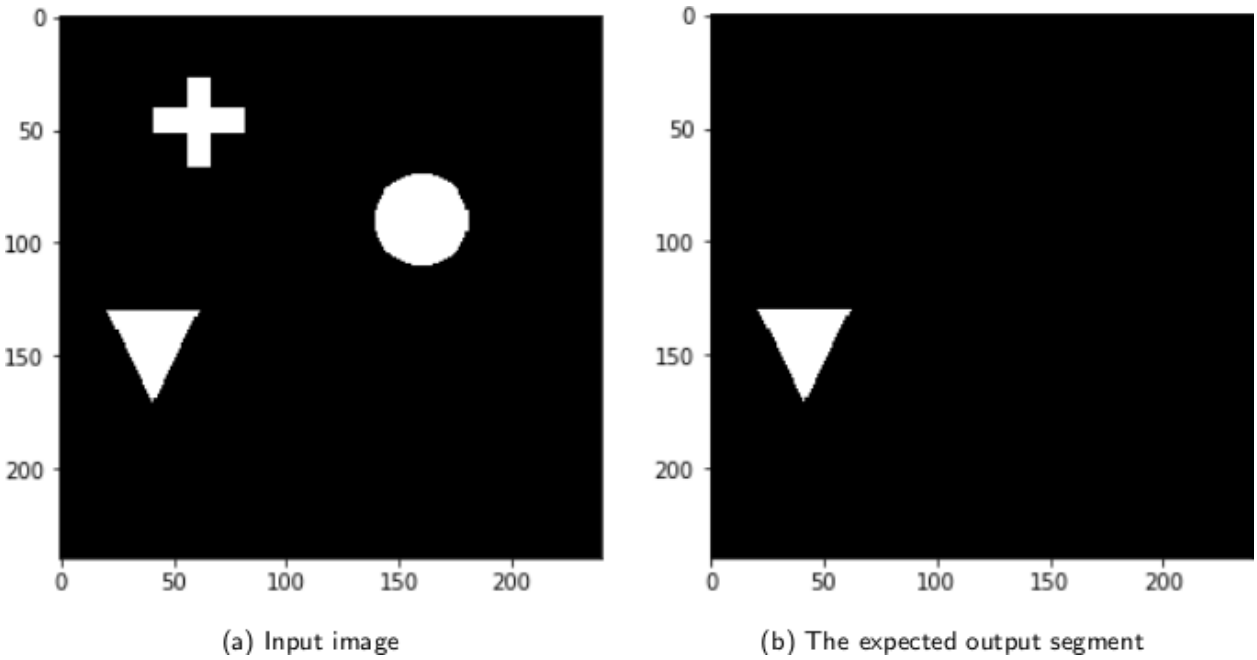


Fig. 2: The right shape in the input is a circle. Therefore the triangle is segmented in the output.

GitHub, remove this directory first.

The neural network can be trained with the `examples/testnet/train.py` script. It will print the results of every epoch on standard out and also into a file inside the `examples/testnet/results` directory. The neural network is saved on every epoch as `testnet_<epoch_number>.pth` in `examples/testnet`. To use one of the generated models, rename the file to `testnet.pth` and run the example scripts.

Index

Symbols

A

C

D

E

G

H