Segmentation Models Documentation

Release 0.1.2

Pavel Yakubovskiy

Jan 10, 2020

Contents:

CHAPTER 1

Installation

Requirements

1) Python 3
2) Keras >= 2.2.0 or TensorFlow >= 1.13
3) keras-applications >= 1.0.7, <=1.0.8
4) image-classifiers == 1.0.0
5) efficientnet == 1.0.0

Note: This library does not have Tensorflow in a requirements.txt for installation. Please, choose suitable version ('cpu'/'gpu') and install it manually using official Guide.

Pip package

```
$ pip install segmentation-models
```

Latest version

```
$ pip install git+https://github.com/qubvel/segmentation_models
```

CHAPTER 2

Tutorial

Segmentation models is python library with Neural Networks for Image Segmentation based on Keras (Tensorflow) framework.

The main features of this library are:

- High level API (just two lines to create NN)
- **4** models architectures for binary and multi class segmentation (including legendary **Unet**)
- **25** available backbones for each architecture
- All backbones have **pre-trained** weights for faster and better convergence

2.1 Quick start

Since the library is built on the Keras framework, created segmentation model is just a Keras Model, which can be created as easy as:

```
from segmentation_models import Unet

model = Unet()
```

Depending on the task, you can change the network architecture by choosing backbones with fewer or more parameters and use pretrainded weights to initialize it:

```
model = Unet('resnet34', encoder_weights='imagenet')
```

Change number of output classes in the model:

```
model = Unet('resnet34', classes=3, activation='softmax')
```

Change input shape of the model:

```
model = Unet('resnet34', input_shape=(None, None, 6), encoder_weights=None)
```

2.2 Simple training pipeline

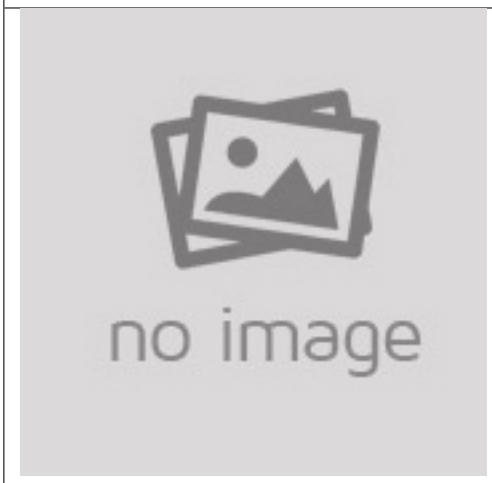
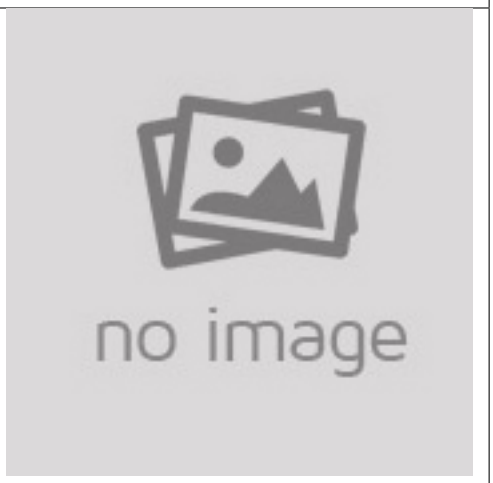
```
from segmentation_models import Unet
from segmentation_models import get_preprocessing
from segmentation_models.losses import bce_jaccard_loss
from segmentation_models.metrics import iou_score

BACKBONE = 'resnet34'
preprocess_input = get_preprocessing(BACKBONE)

# load your data
x_train, y_train, x_val, y_val = load_data(...)

# preprocess input
x_train = preprocess_input(x_train)
x_val = preprocess_input(x_val)

# define model
model = Unet(BACKBONE, encoder_weights='imagenet')
model.compile('Adam', loss=bce_jaccard_loss, metrics=[iou_score])

# fit model
model.fit(
    x=x_train,
    y=y_train,
    batch_size=16,
    epochs=100,
    validation_data=(x_val, y_val),
)
```

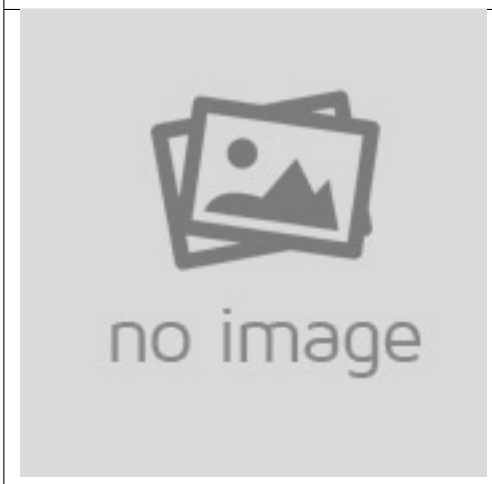
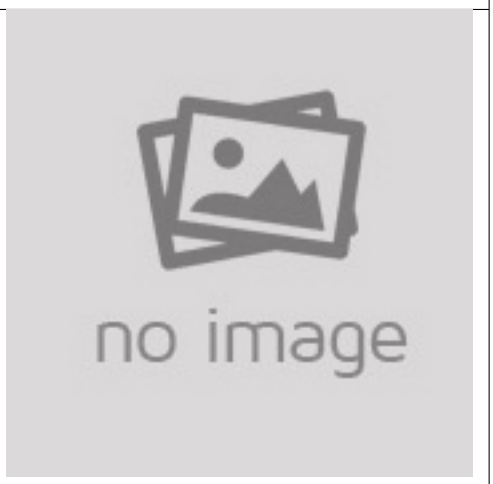
Same manimulations can be done with `Linknet`, `PSPNet` and `FPN`. For more detailed information about models API and use cases Read the Docs.

2.3 Models and Backbones

Models

- Unet
- FPN
- Linknet
- PSPNet

Unet	Linknet
	

PSPNet	FPN
	

Backbones

All backbones have weights trained on 2012 ILSVRC ImageNet dataset (`encoder_weights='imagenet'`).

2.4 Fine tuning

Some times, it is useful to train only randomly initialized *decoder* in order not to damage weights of properly trained *encoder* with huge gradients during first steps of training. In this case, all you need is just pass `encoder_freeze = True` argument while initializing the model.

```
from segmentation_models import Unet
from segmentation_models.utils import set_trainable

model = Unet(backbone_name='resnet34', encoder_weights='imagenet', encoder_
↪freeze=True)
model.compile('Adam', 'binary_crossentropy', ['binary_accuracy'])
```

(continues on next page)

(continued from previous page)

```
# pretrain model decoder
model.fit(x, y, epochs=2)

# release all layers for training
set_trainable(model) # set all layers trainable and recompile model

# continue training
model.fit(x, y, epochs=100)
```

2.5 Training with non-RGB data

In case you have non RGB images (e.g. grayscale or some medical/remote sensing data) you have few different options:

1. Train network from scratch with randomly initialized weights

```
from segmentation_models import Unet

# read/scale/preprocess data
x, y = ...

# define number of channels
N = x.shape[-1]

# define model
model = Unet(backbone_name='resnet34', encoder_weights=None, input_shape=(None, None,␣
↪N))

# continue with usual steps: compile, fit, etc..
```

2. Add extra convolution layer to map `N -> 3` channels data and train with pretrained weights

```
from segmentation_models import Unet
from keras.layers import Input, Conv2D
from keras.models import Model

# read/scale/preprocess data
x, y = ...

# define number of channels
N = x.shape[-1]

base_model = Unet(backbone_name='resnet34', encoder_weights='imagenet')

inp = Input(shape=(None, None, N))
l1 = Conv2D(3, (1, 1))(inp) # map N channels data to 3 channels
out = base_model(l1)

model = Model(inp, out, name=base_model.name)

# continue with usual steps: compile, fit, etc..
```

CHAPTER 3

Segmentation Models Python API

Getting started with segmentation models is easy.

3.1 Unet

segmentation_models.**Unet**(*backbone_name='vgg16'*, *input_shape=(None*, *None*, *3)*, *classes=1*, *activation='sigmoid'*, *weights=None*, *encoder_weights='imagenet'*, *encoder_freeze=False*, *encoder_features='default'*, *decoder_block_type='upsampling'*, *decoder_filters=(256*, *128*, *64*, *32*, *16)*, *decoder_use_batchnorm=True*, ***kwargs*)

Unet is a fully convolution neural network for image semantic segmentation

Parameters

- **backbone_name** – name of classification model (without last dense layers) used as feature extractor to build segmentation model.
- **input_shape** – shape of input data/image `(H, W, C)`, in general case you do not need to set `H` and `W` shapes, just pass `(None, None, C)` to make your model be able to process images af any size, but `H` and `W` of input images should be divisible by factor `32`.
- **classes** – a number of classes for output (output shape - `(h, w, classes)`).
- **activation** – name of one of `keras.activations` for last model layer (e.g. `sigmoid`, `softmax`, `linear`).
- **weights** – optional, path to model weights.
- **encoder_weights** – one of `None` (random initialization), `imagenet` (pre-training on ImageNet).
- **encoder_freeze** – if `True` set all layers of encoder (backbone model) as non-trainable.
- **encoder_features** – a list of layer numbers or names starting from top of the model. Each of these layers will be concatenated with corresponding decoder block. If `default` is used layer names are taken from `DEFAULT_SKIP_CONNECTIONS`.

- **decoder_block_type** – one of blocks with following layers structure:
  - *upsampling*: `UpSampling2D` -> `Conv2D` -> `Conv2D`
  - *transpose*: `Transpose2D` -> `Conv2D`
- **decoder_filters** – list of numbers of `Conv2D` layer filters in decoder blocks
- **decoder_use_batchnorm** – if `True`, `BatchNormalisation` layer between `Conv2D` and `Activation` layers is used.

Returns Unet

Return type `keras.models.Model`

3.2 Linknet

`segmentation_models.`**Linknet**(*backbone_name='vgg16'*, *input_shape=(None*, *None*, *3)*, *classes=1*, *activation='sigmoid'*, *weights=None*, *encoder_weights='imagenet'*, *encoder_freeze=False*, *encoder_features='default'*, *decoder_block_type='upsampling'*, *decoder_filters=(None*, *None*, *None*, *None*, *16)*, *decoder_use_batchnorm=True*, ***kwargs*)

Linknet is a fully convolution neural network for fast image semantic segmentation

Note: This implementation by default has 4 skip connections (original - 3).

Parameters

- **backbone_name** – name of classification model (without last dense layers) used as feature extractor to build segmentation model.
- **input_shape** – shape of input data/image `(H, W, C)`, in general case you do not need to set `H` and `W` shapes, just pass `(None, None, C)` to make your model be able to process images af any size, but `H` and `W` of input images should be divisible by factor `32`.
- **classes** – a number of classes for output (output shape - `(h, w, classes)`).
- **activation** – name of one of `keras.activations` for last model layer (e.g. `sigmoid`, `softmax`, `linear`).
- **weights** – optional, path to model weights.
- **encoder_weights** – one of `None` (random initialization), `imagenet` (pre-training on ImageNet).
- **encoder_freeze** – if `True` set all layers of encoder (backbone model) as non-trainable.
- **encoder_features** – a list of layer numbers or names starting from top of the model. Each of these layers will be concatenated with corresponding decoder block. If `default` is used layer names are taken from `DEFAULT_SKIP_CONNECTIONS`.
- **decoder_filters** – list of numbers of `Conv2D` layer filters in decoder blocks, for block with skip connection a number of filters is equal to number of filters in corresponding encoder block (estimates automatically and can be passed as `None` value).
- **decoder_use_batchnorm** – if `True`, `BatchNormalisation` layer between `Conv2D` and `Activation` layers is used.

- **decoder_block_type** – one of - *upsampling*: use `UpSampling2D` keras layer - *transpose*: use `Transpose2D` keras layer

Returns Linknet

Return type `keras.models.Model`

3.3 FPN

`segmentation_models.`**FPN**(*backbone_name='vgg16'*, *input_shape=(None, None, 3)*, *classes=21*, *activation='softmax'*, *weights=None*, *encoder_weights='imagenet'*, *encoder_freeze=False*, *encoder_features='default'*, *pyramid_block_filters=256*, *pyramid_use_batchnorm=True*, *pyramid_aggregation='concat'*, *pyramid_dropout=None*, ***kwargs*)

FPN is a fully convolution neural network for image semantic segmentation

Parameters

- **backbone_name** – name of classification model (without last dense layers) used as feature extractor to build segmentation model.
- **input_shape** – shape of input data/image `(H, W, C)`, in general case you do not need to set `H` and `W` shapes, just pass `(None, None, C)` to make your model be able to process images af any size, but `H` and `W` of input images should be divisible by factor `32`.
- **classes** – a number of classes for output (output shape - `(h, w, classes)`).
- **weights** – optional, path to model weights.
- **activation** – name of one of `keras.activations` for last model layer (e.g. `sigmoid`, `softmax`, `linear`).
- **encoder_weights** – one of `None` (random initialization), `imagenet` (pre-training on ImageNet).
- **encoder_freeze** – if `True` set all layers of encoder (backbone model) as non-trainable.
- **encoder_features** – a list of layer numbers or names starting from top of the model. Each of these layers will be used to build features pyramid. If `default` is used layer names are taken from `DEFAULT_FEATURE_PYRAMID_LAYERS`.
- **pyramid_block_filters** – a number of filters in Feature Pyramid Block of FPN.
- **pyramid_use_batchnorm** – if `True`, `BatchNormalisation` layer between `Conv2D` and `Activation` layers is used.
- **pyramid_aggregation** – one of 'sum' or 'concat'. The way to aggregate pyramid blocks.
- **pyramid_dropout** – spatial dropout rate for feature pyramid in range (0, 1).

Returns FPN

Return type `keras.models.Model`

3.4 PSPNet

segmentation_models.**PSPNet**(*backbone_name='vgg16'*, *input_shape=(384, 384, 3)*, *classes=21*, *activation='softmax'*, *weights=None*, *encoder_weights='imagenet'*, *encoder_freeze=False*, *downsample_factor=8*, *psp_conv_filters=512*, *psp_pooling_type='avg'*, *psp_use_batchnorm=True*, *psp_dropout=None*, ***kwargs*)

PSPNet is a fully convolution neural network for image semantic segmentation

Parameters

- **backbone_name** – name of classification model used as feature extractor to build segmentation model.
- **input_shape** – shape of input data/image `(H, W, C)`. `H` and `W` should be divisible by `6 * downsample_factor` and **NOT** `None`!
- **classes** – a number of classes for output (output shape - `(h, w, classes)`).
- **activation** – name of one of `keras.activations` for last model layer (e.g. `sigmoid`, `softmax`, `linear`).
- **weights** – optional, path to model weights.
- **encoder_weights** – one of `None` (random initialization), `imagenet` (pre-training on ImageNet).
- **encoder_freeze** – if `True` set all layers of encoder (backbone model) as non-trainable.
- **downsample_factor** – one of 4, 8 and 16. Downsampling rate or in other words backbone depth to construct PSP module on it.
- **psp_conv_filters** – number of filters in `Conv2D` layer in each PSP block.
- **psp_pooling_type** – one of 'avg', 'max'. PSP block pooling type (maximum or average).
- **psp_use_batchnorm** – if `True`, `BatchNormalisation` layer between `Conv2D` and `Activation` layers is used.
- **psp_dropout** – dropout rate between 0 and 1.

Returns PSPNet

Return type `keras.models.Model`

3.5 metrics

segmentation_models.metrics.**IOUScore**(*class_weights=None*, *class_indexes=None*, *threshold=None*, *per_image=False*, *smooth=1e-05*, *name=None*)

The Jaccard index, also known as Intersection over Union and the Jaccard similarity coefficient (originally coined coefficient de communauté by Paul Jaccard), is a statistic used for comparing the similarity and diversity of sample sets. The Jaccard coefficient measures similarity between finite sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets:

$$J(A,B) = \frac{A \cap B}{A \cup B}$$

Parameters

- **class_weights** –

  1. or `np.array` of class weights (`len(weights) = num_classes`).

- **class_indexes** – Optional integer or list of integers, classes to consider, if `None` all classes are used.
- **smooth** – value to avoid division by zero
- **per_image** – if `True`, metric is calculated as mean over images in batch (B), else over whole batch
- **threshold** – value to round predictions (use > comparison), if `None` prediction will not be round

Returns A callable `iou_score` instance. Can be used in `model.compile(...)` function.

Example:

```
metric = IOUScore()
model.compile('SGD', loss=loss, metrics=[metric])
```

segmentation_models.metrics.**FScore**(*beta=1*, *class_weights=None*, *class_indexes=None*, *threshold=None*, *per_image=False*, *smooth=1e-05*, *name=None*)

The F-score (Dice coefficient) can be interpreted as a weighted average of the precision and recall, where an F-score reaches its best value at 1 and worst score at 0. The relative contribution of `precision` and `recall` to the F1-score are equal. The formula for the F score is:

$$F_\beta(precision, recall) = (1 + \beta^2)\frac{precision \cdot recall}{\beta^2 \cdot precision + recall}$$

The formula in terms of *Type I* and *Type II* errors:

$$L(tp, fp, fn) = \frac{(1 + \beta^2) \cdot tp}{(1 + \beta^2) \cdot fp + \beta^2 \cdot fn + fp}$$

where:

- tp - true positives;
- fp - false positives;
- fn - false negatives;

Parameters

- **beta** – Integer of float f-score coefficient to balance precision and recall.
- **class_weights** –

  1. or `np.array` of class weights (`len(weights) = num_classes`)

- **class_indexes** – Optional integer or list of integers, classes to consider, if `None` all classes are used.
- **smooth** – Float value to avoid division by zero.
- **per_image** – If `True`, metric is calculated as mean over images in batch (B), else over whole batch.
- **threshold** – Float value to round predictions (use > comparison), if `None` prediction will not be round.
- **name** – Optional string, if `None` default `f{beta}-score` name is used.

Returns A callable `f_score` instance. Can be used in `model.compile(...)` function.

Example:

```
metric = FScore()
model.compile('SGD', loss=loss, metrics=[metric])
```

3.6 losses

`segmentation_models.losses.`**`JaccardLoss`**(*class_weights=None*, *class_indexes=None*, *per_image=False*, *smooth=1e-05*)

Creates a criterion to measure Jaccard loss:

$$L(A, B) = 1 - \frac{A \cap B}{A \cup B}$$

Parameters

- **`class_weights`** – Array (`np.array`) of class weights (`len(weights) = num_classes`).
- **`class_indexes`** – Optional integer or list of integers, classes to consider, if `None` all classes are used.
- **`per_image`** – If `True` loss is calculated for each image in batch and then averaged, else loss is calculated for the whole batch.
- **`smooth`** – Value to avoid division by zero.

Returns A callable `jaccard_loss` instance. Can be used in `model.compile(...)` function or combined with other losses.

Example:

```
loss = JaccardLoss()
model.compile('SGD', loss=loss)
```

`segmentation_models.losses.`**`DiceLoss`**(*beta=1*, *class_weights=None*, *class_indexes=None*, *per_image=False*, *smooth=1e-05*)

Creates a criterion to measure Dice loss:

$$L(precision, recall) = 1 - (1 + \beta^2) \frac{precision \cdot recall}{\beta^2 \cdot precision + recall}$$

The formula in terms of *Type I* and *Type II* errors:

$$L(tp, fp, fn) = \frac{(1 + \beta^2) \cdot tp}{(1 + \beta^2) \cdot fp + \beta^2 \cdot fn + fp}$$

where:

- tp - true positives;
- fp - false positives;
- fn - false negatives;

Parameters

- **`beta`** – Float or integer coefficient for precision and recall balance.

- **class_weights** – Array (`np.array`) of class weights (`len(weights) = num_classes`).
- **class_indexes** – Optional integer or list of integers, classes to consider, if `None` all classes are used.
- **per_image** – If `True` loss is calculated for each image in batch and then averaged,
- **loss is calculated for the whole batch.** (*else*) –
- **smooth** – Value to avoid division by zero.

Returns A callable `dice_loss` instance. Can be used in `model.compile(...)` function‘ or combined with other losses.

Example:

```
loss = DiceLoss()
model.compile('SGD', loss=loss)
```

segmentation_models.losses.**BinaryCELoss**()

Creates a criterion that measures the Binary Cross Entropy between the ground truth (gt) and the prediction (pr).

$$L(gt, pr) = -gt \cdot \log(pr) - (1 - gt) \cdot \log(1 - pr)$$

Returns A callable `binary_crossentropy` instance. Can be used in `model.compile(...)` function or combined with other losses.

Example:

```
loss = BinaryCELoss()
model.compile('SGD', loss=loss)
```

segmentation_models.losses.**CategoricalCELoss**(*class_weights=None*, *class_indexes=None*)

Creates a criterion that measures the Categorical Cross Entropy between the ground truth (gt) and the prediction (pr).

$$L(gt, pr) = -gt \cdot \log(pr)$$

Parameters

- **class_weights** – Array (`np.array`) of class weights (`len(weights) = num_classes`).
- **class_indexes** – Optional integer or list of integers, classes to consider, if `None` all classes are used.

Returns A callable `categorical_crossentropy` instance. Can be used in `model.compile(...)` function or combined with other losses.

Example:

```
loss = CategoricalCELoss()
model.compile('SGD', loss=loss)
```

segmentation_models.losses.**BinaryFocalLoss**(*alpha=0.25*, *gamma=2.0*)

Creates a criterion that measures the Binary Focal Loss between the ground truth (gt) and the prediction (pr).

$$L(gt, pr) = -gt\alpha(1 - pr)^{\gamma} \log(pr) - (1 - gt)\alpha pr^{\gamma} \log(1 - pr)$$

Parameters

- **alpha** – Float or integer, the same as weighting factor in balanced cross entropy, default 0.25.
- **gamma** – Float or integer, focusing parameter for modulating factor (1 - p), default 2.0.

Returns A callable `binary_focal_loss` instance. Can be used in `model.compile(...)` function or combined with other losses.

Example:

```
loss = BinaryFocalLoss()
model.compile('SGD', loss=loss)
```

segmentation_models.losses.**CategoricalFocalLoss**(*alpha=0.25*, *gamma=2.0*, *class_indexes=None*)

Creates a criterion that measures the Categorical Focal Loss between the ground truth (gt) and the prediction (pr).

$$L(gt, pr) = -gt \cdot \alpha \cdot (1 - pr)^{\gamma} \cdot \log(pr)$$

Parameters

- **alpha** – Float or integer, the same as weighting factor in balanced cross entropy, default 0.25.
- **gamma** – Float or integer, focusing parameter for modulating factor (1 - p), default 2.0.
- **class_indexes** – Optional integer or list of integers, classes to consider, if `None` all classes are used.

Returns A callable `categorical_focal_loss` instance. Can be used in `model.compile(...)` function or combined with other losses.

Example

```
loss = CategoricalFocalLoss()
model.compile('SGD', loss=loss)
```

3.7 utils

segmentation_models.utils.**set_trainable**(*model*, *recompile=True*, ***kwargs*)

Set all layers of model trainable and recompile it

Note: Model is recompiled using same optimizer, loss and metrics:

```
model.compile(
    model.optimizer,
    loss=model.loss,
    metrics=model.metrics,
    loss_weights=model.loss_weights,
    sample_weight_mode=model.sample_weight_mode,
    weighted_metrics=model.weighted_metrics,
)
```

Parameters **model** (keras.models.Model) – instance of keras model

CHAPTER 4

Support

The easiest way to get help with the project is to create issue or PR on github.

Github: http://github.com/qubvel/segmentation_models/issues

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index