
Segmentation Models

Pavel Iakubovskii

Aug 07, 2023

CONTENTS:

1	Installation	1
2	Quick Start	3
3	Segmentation Models	5
3.1	Unet	5
3.2	Unet++	6
3.3	MAnet	7
3.4	Linknet	8
3.5	FPN	9
3.6	PSPNet	11
3.7	PAN	12
3.8	DeepLabV3	13
3.9	DeepLabV3+	14
4	Available Encoders	17
4.1	ResNet	17
4.2	ResNeXt	17
4.3	ResNeSt	17
4.4	Res2Ne(X)t	18
4.5	RegNet(x/y)	18
4.6	GERNet	19
4.7	SE-Net	19
4.8	SK-ResNe(X)t	19
4.9	DenseNet	19
4.10	Inception	19
4.11	EfficientNet	20
4.12	MobileNet	20
4.13	DPN	21
4.14	VGG	21
4.15	Mix Visual Transformer	21
4.16	MobileOne	21
5	Timm Encoders	23
6	Losses	35
6.1	Constants	35
6.2	JaccardLoss	35
6.3	DiceLoss	36
6.4	TverskyLoss	36
6.5	FocalLoss	37

6.6	LovaszLoss	38
6.7	SoftBCEWithLogitsLoss	38
6.8	SoftCrossEntropyLoss	39
6.9	MCCLoss	39
7	Metrics	41
7.1	Functional metrics	41
8	Insights	65
8.1	1. Models architecture	65
8.2	2. Creating your own encoder	65
8.3	3. Aux classification output	67
9	Indices and tables	69
	Python Module Index	71
	Index	73

INSTALLATION

PyPI version:

```
$ pip install -U segmentation-models-pytorch
```

Latest version from source:

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


QUICK START

1. Create segmentation model

Segmentation model is just a PyTorch nn.Module, which can be created as easy as:

```
import segmentation_models_pytorch as smp

model = smp.Unet(
    encoder_name="resnet34",          # choose encoder, e.g. mobilenet_v2 or efficientnet-
    ↪ b7                             # use `imagenet` pre-trained weights for encoder ↪
    ↪ initialization                  # model input channels (1 for gray-scale images, 3 ↪
    ↪ for RGB, etc.)                 # model output channels (number of classes in your ↪
    ↪ dataset)                       ↪
)
```

- see table with available model architectures
- see table with available encoders and its corresponding weights

2. Configure data preprocessing

All encoders have pretrained weights. Preparing your data the same way as during weights pre-training may give your better results (higher metric score and faster convergence). But it is relevant only for 1-2-3-channels images and **not necessary** in case you train the whole model, not only decoder.

```
from segmentation_models_pytorch.encoders import get_preprocessing_fn

preprocess_input = get_preprocessing_fn('resnet18', pretrained='imagenet')
```

3. Congratulations!

You are done! Now you can train your model with your favorite framework!

SEGMENTATION MODELS

3.1 Unet

```
class segmentation_models_pytorch.Unet(encoder_name='resnet34', encoder_depth=5,
                                         encoder_weights='imagenet', decoder_use_batchnorm=True,
                                         decoder_channels=(256, 128, 64, 32, 16),
                                         decoder_attention_type=None, in_channels=3, classes=1,
                                         activation=None, aux_params=None)
```

Unet is a fully convolution neural network for image semantic segmentation. Consist of *encoder* and *decoder* parts connected with *skip connections*. Encoder extract features of different spatial resolution (skip connections) which are used by decoder to define accurate segmentation mask. Use *concatenation* for fusing decoder blocks with skip connections.

Parameters

- **encoder_name** – Name of the classification model that will be used as an encoder (a.k.a backbone) to extract features of different spatial resolution
- **encoder_depth** – A number of stages used in encoder in range [3, 5]. Each stage generate features two times smaller in spatial dimensions than previous one (e.g. for depth 0 we will have features with shapes [(N, C, H, W),], for depth 1 - [(N, C, H, W), (N, C, H // 2, W // 2)] and so on). Default is 5
- **encoder_weights** – One of **None** (random initialization), **“imagenet”** (pre-training on ImageNet) and other pretrained weights (see table with available weights for each encoder_name)
- **decoder_channels** – List of integers which specify **in_channels** parameter for convolutions used in decoder. Length of the list should be the same as **encoder_depth**
- **decoder_use_batchnorm** – If **True**, BatchNorm2d layer between Conv2D and Activation layers is used. If **“inplace”** InplaceABN will be used, allows to decrease memory consumption. Available options are **True**, **False**, **“inplace”**
- **decoder_attention_type** – Attention module used in decoder of the model. Available options are **None** and **scse** (<https://arxiv.org/abs/1808.08127>).
- **in_channels** – A number of input channels for the model, default is 3 (RGB images)
- **classes** – A number of classes for output mask (or you can think as a number of channels of output mask)
- **activation** – An activation function to apply after the final convolution layer. Available options are **“sigmoid”**, **“softmax”**, **“logsoftmax”**, **“tanh”**, **“identity”**, **callable** and **None**.

Default is **None**

- **aux_params** – Dictionary with parameters of the auxiliary output (classification head). Auxiliary output is build on top of encoder if **aux_params** is not **None** (default). Supported params:
 - **classes** (int): A number of classes
 - **pooling** (str): One of “max”, “avg”. Default is “avg”
 - **dropout** (float): Dropout factor in [0, 1)
 - **activation** (str): An activation function to apply “sigmoid”/“softmax” (could be **None** to return logits)

Returns

Unet

Return type

`torch.nn.Module`

3.2 Unet++

```
class segmentation_models_pytorch.UnetPlusPlus(encoder_name='resnet34', encoder_depth=5,
                                                encoder_weights='imagenet',
                                                decoder_use_batchnorm=True,
                                                decoder_channels=(256, 128, 64, 32, 16),
                                                decoder_attention_type=None, in_channels=3,
                                                classes=1, activation=None, aux_params=None)
```

Unet++ is a fully convolution neural network for image semantic segmentation. Consist of *encoder* and *decoder* parts connected with *skip connections*. Encoder extract features of different spatial resolution (skip connections) which are used by decoder to define accurate segmentation mask. Decoder of Unet++ is more complex than in usual Unet.

Parameters

- **encoder_name** – Name of the classification model that will be used as an encoder (a.k.a backbone) to extract features of different spatial resolution
- **encoder_depth** – A number of stages used in encoder in range [3, 5]. Each stage generate features two times smaller in spatial dimensions than previous one (e.g. for depth 0 we will have features with shapes [(N, C, H, W),], for depth 1 - [(N, C, H, W), (N, C, H // 2, W // 2)] and so on). Default is 5
- **encoder_weights** – One of **None** (random initialization), “**imagenet**” (pre-training on ImageNet) and other pretrained weights (see table with available weights for each encoder_name)
- **decoder_channels** – List of integers which specify **in_channels** parameter for convolutions used in decoder. Length of the list should be the same as **encoder_depth**
- **decoder_use_batchnorm** – If **True**, BatchNorm2d layer between Conv2D and Activation layers is used. If “**inplace**” InplaceABN will be used, allows to decrease memory consumption. Available options are **True**, **False**, “**inplace**”
- **decoder_attention_type** – Attention module used in decoder of the model. Available options are **None** and **scse** (<https://arxiv.org/abs/1808.08127>).
- **in_channels** – A number of input channels for the model, default is 3 (RGB images)

- **classes** – A number of classes for output mask (or you can think as a number of channels of output mask)
- **activation** – An activation function to apply after the final convolution layer. Available options are “sigmoid”, “softmax”, “logsoftmax”, “tanh”, “identity”, callable and None.

Default is **None**

- **aux_params** – Dictionary with parameters of the auxiliary output (classification head). Auxiliary output is build on top of encoder if **aux_params** is not **None** (default). Supported params:
 - **classes** (int): A number of classes
 - **pooling** (str): One of “max”, “avg”. Default is “avg”
 - **dropout** (float): Dropout factor in [0, 1)
 - **activation** (str): An activation function to apply “sigmoid”/”softmax” (could be **None** to return logits)

Returns

Unet++

Return type

torch.nn.Module

Reference:

<https://arxiv.org/abs/1807.10165>

3.3 MAnet

```
class segmentation_models_pytorch.MAnet(encoder_name='resnet34', encoder_depth=5,
                                         encoder_weights='imagenet', decoder_use_batchnorm=True,
                                         decoder_channels=(256, 128, 64, 32, 16),
                                         decoder_pab_channels=64, in_channels=3, classes=1,
                                         activation=None, aux_params=None)
```

MAnet: Multi-scale Attention Net. The MA-Net can capture rich contextual dependencies based on the attention mechanism, using two blocks:

- Position-wise Attention Block (PAB), which captures the spatial dependencies between pixels in a global view
- Multi-scale Fusion Attention Block (MFAB), which captures the channel dependencies between any feature map by multi-scale semantic feature fusion

Parameters

- **encoder_name** – Name of the classification model that will be used as an encoder (a.k.a backbone) to extract features of different spatial resolution
- **encoder_depth** – A number of stages used in encoder in range [3, 5]. Each stage generate features two times smaller in spatial dimensions than previous one (e.g. for depth 0 we will have features with shapes [(N, C, H, W),], for depth 1 - [(N, C, H, W), (N, C, H // 2, W // 2)] and so on). Default is 5

- **encoder_weights** – One of **None** (random initialization), **“imagenet”** (pre-training on ImageNet) and other pretrained weights (see table with available weights for each encoder_name)
- **decoder_channels** – List of integers which specify **in_channels** parameter for convolutions used in decoder. Length of the list should be the same as **encoder_depth**
- **decoder_use_batchnorm** – If **True**, BatchNorm2d layer between Conv2D and Activation layers is used. If **“inplace”** InplaceABN will be used, allows to decrease memory consumption. Available options are **True, False, “inplace”**
- **decoder_pab_channels** – A number of channels for PAB module in decoder. Default is 64.
- **in_channels** – A number of input channels for the model, default is 3 (RGB images)
- **classes** – A number of classes for output mask (or you can think as a number of channels of output mask)
- **activation** – An activation function to apply after the final convolution layer. Available options are **“sigmoid”, “softmax”, “logsoftmax”, “tanh”, “identity”, callable** and **None**.
Default is **None**
- **aux_params** – Dictionary with parameters of the auxiliary output (classification head). Auxiliary output is build on top of encoder if **aux_params** is not **None** (default). Supported params:
 - **classes** (int): A number of classes
 - **pooling** (str): One of **“max”, “avg”**. Default is **“avg”**
 - **dropout** (float): Dropout factor in [0, 1)
 - **activation** (str): An activation function to apply **“sigmoid”/“softmax”** (could be **None** to return logits)

Returns

MANet

Return type

torch.nn.Module

3.4 Linknet

```
class segmentation_models_pytorch.Linknet(encoder_name='resnet34', encoder_depth=5,  
                                           encoder_weights='imagenet', decoder_use_batchnorm=True,  
                                           in_channels=3, classes=1, activation=None,  
                                           aux_params=None)
```

Linknet is a fully convolution neural network for image semantic segmentation. Consist of *encoder* and *decoder* parts connected with *skip connections*. Encoder extract features of different spatial resolution (skip connections) which are used by decoder to define accurate segmentation mask. Use *sum* for fusing decoder blocks with skip connections.

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

Parameters

- **encoder_name** – Name of the classification model that will be used as an encoder (a.k.a backbone) to extract features of different spatial resolution
- **encoder_depth** – A number of stages used in encoder in range [3, 5]. Each stage generate features two times smaller in spatial dimensions than previous one (e.g. for depth 0 we will have features with shapes [(N, C, H, W),], for depth 1 - [(N, C, H, W), (N, C, H // 2, W // 2)] and so on). Default is 5
- **encoder_weights** – One of **None** (random initialization), **“imagenet”** (pre-training on ImageNet) and other pretrained weights (see table with available weights for each encoder_name)
- **decoder_use_batchnorm** – If **True**, BatchNorm2d layer between Conv2D and Activation layers is used. If **“inplace”** InplaceABN will be used, allows to decrease memory consumption. Available options are **True**, **False**, **“inplace”**
- **in_channels** – A number of input channels for the model, default is 3 (RGB images)
- **classes** – A number of classes for output mask (or you can think as a number of channels of output mask)
- **activation** – An activation function to apply after the final convolution layer. Available options are **“sigmoid”**, **“softmax”**, **“logsoftmax”**, **“tanh”**, **“identity”**, **callable** and **None**.

Default is **None**

- **aux_params** – Dictionary with parameters of the auxiliary output (classification head). Auxiliary output is build on top of encoder if **aux_params** is not **None** (default). Supported params:
 - **classes** (int): A number of classes
 - **pooling** (str): One of **“max”**, **“avg”**. Default is **“avg”**
 - **dropout** (float): Dropout factor in [0, 1)
 - **activation** (str): An activation function to apply **“sigmoid”/“softmax”** (could be **None** to return logits)

Returns

Linknet

Return type

`torch.nn.Module`

3.5 FPN

```
class segmentation_models_pytorch.FPN(encoder_name='resnet34', encoder_depth=5,
                                       encoder_weights='imagenet', decoder_pyramid_channels=256,
                                       decoder_segmentation_channels=128,
                                       decoder_merge_policy='add', decoder_dropout=0.2,
                                       in_channels=3, classes=1, activation=None, upsampling=4,
                                       aux_params=None)
```

FPN is a fully convolution neural network for image semantic segmentation.

Parameters

- **encoder_name** – Name of the classification model that will be used as an encoder (a.k.a backbone) to extract features of different spatial resolution
- **encoder_depth** – A number of stages used in encoder in range [3, 5]. Each stage generate features two times smaller in spatial dimensions than previous one (e.g. for depth 0 we will have features with shapes [(N, C, H, W),], for depth 1 - [(N, C, H, W), (N, C, H // 2, W // 2)] and so on). Default is 5
- **encoder_weights** – One of **None** (random initialization), **“imagenet”** (pre-training on ImageNet) and other pretrained weights (see table with available weights for each encoder_name)
- **decoder_pyramid_channels** – A number of convolution filters in Feature Pyramid of FPN
- **decoder_segmentation_channels** – A number of convolution filters in segmentation blocks of FPN
- **decoder_merge_policy** – Determines how to merge pyramid features inside FPN. Available options are **add** and **cat**
- **decoder_dropout** – Spatial dropout rate in range (0, 1) for feature pyramid in FPN
- **in_channels** – A number of input channels for the model, default is 3 (RGB images)
- **classes** – A number of classes for output mask (or you can think as a number of channels of output mask)
- **activation** – An activation function to apply after the final convolution layer. Available options are **“sigmoid”**, **“softmax”**, **“logsoftmax”**, **“tanh”**, **“identity”**, **callable** and **None**.
Default is **None**
- **upsampling** – Final upsampling factor. Default is 4 to preserve input-output spatial shape identity
- **aux_params** – Dictionary with parameters of the auxiliary output (classification head). Auxiliary output is build on top of encoder if **aux_params** is not **None** (default). Supported params:
 - **classes** (int): A number of classes
 - **pooling** (str): One of **“max”**, **“avg”**. Default is **“avg”**
 - **dropout** (float): Dropout factor in [0, 1)
 - **activation** (str): An activation function to apply **“sigmoid”/“softmax”** (could be **None** to return logits)

Returns

FPN

Return type

`torch.nn.Module`

3.6 PSPNet

```
class segmentation_models_pytorch.PSPNet(encoder_name='resnet34', encoder_weights='imagenet',
                                          encoder_depth=3, psp_out_channels=512,
                                          psp_use_batchnorm=True, psp_dropout=0.2, in_channels=3,
                                          classes=1, activation=None, upsampling=8,
                                          aux_params=None)
```

PSPNet is a fully convolution neural network for image semantic segmentation. Consist of *encoder* and *Spatial Pyramid* (decoder). Spatial Pyramid build on top of encoder and does not use “fine-features” (features of high spatial resolution). PSPNet can be used for multiclass segmentation of high resolution images, however it is not good for detecting small objects and producing accurate, pixel-level mask.

Parameters

- **encoder_name** – Name of the classification model that will be used as an encoder (a.k.a backbone) to extract features of different spatial resolution
- **encoder_depth** – A number of stages used in encoder in range [3, 5]. Each stage generate features two times smaller in spatial dimensions than previous one (e.g. for depth 0 we will have features with shapes [(N, C, H, W),], for depth 1 - [(N, C, H, W), (N, C, H // 2, W // 2)] and so on). Default is 5
- **encoder_weights** – One of **None** (random initialization), “**imagenet**” (pre-training on ImageNet) and other pretrained weights (see table with available weights for each encoder_name)
- **psp_out_channels** – A number of filters in Spatial Pyramid
- **psp_use_batchnorm** – If **True**, BatchNorm2d layer between Conv2D and Activation layers is used. If “**inplace**” InplaceABN will be used, allows to decrease memory consumption. Available options are **True**, **False**, “**inplace**”
- **psp_dropout** – Spatial dropout rate in [0, 1) used in Spatial Pyramid
- **in_channels** – A number of input channels for the model, default is 3 (RGB images)
- **classes** – A number of classes for output mask (or you can think as a number of channels of output mask)
- **activation** – An activation function to apply after the final convolution layer. Available options are “**sigmoid**”, “**softmax**”, “**logsoftmax**”, “**tanh**”, “**identity**”, callable and **None**.

Default is **None**

- **upsampling** – Final upsampling factor. Default is 8 to preserve input-output spatial shape identity
- **aux_params** – Dictionary with parameters of the auxiliary output (classification head). Auxiliary output is build on top of encoder if **aux_params** is not **None** (default). Supported params:
 - **classes** (int): A number of classes
 - **pooling** (str): One of “**max**”, “**avg**”. Default is “**avg**”
 - **dropout** (float): Dropout factor in [0, 1)
 - **activation** (str): An activation function to apply “**sigmoid**”/”**softmax**” (could be **None** to return logits)

Returns

PSPNet

Return type

`torch.nn.Module`

3.7 PAN

```
class segmentation_models_pytorch.PAN(encoder_name='resnet34', encoder_weights='imagenet',
                                         encoder_output_stride=16, decoder_channels=32, in_channels=3,
                                         classes=1, activation=None, upsampling=4, aux_params=None)
```

Implementation of [PAN](#) (Pyramid Attention Network).

Note: Currently works with shape of input tensor $\geq [B \times C \times 128 \times 128]$ for pytorch $\leq 1.1.0$ and with shape of input tensor $\geq [B \times C \times 256 \times 256]$ for pytorch $= 1.3.1$

Parameters

- **encoder_name** – Name of the classification model that will be used as an encoder (a.k.a backbone) to extract features of different spatial resolution
- **encoder_weights** – One of **None** (random initialization), “**imagenet**” (pre-training on ImageNet) and other pretrained weights (see table with available weights for each encoder_name)
- **encoder_output_stride** – 16 or 32, if 16 use dilation in encoder last layer. Doesn’t work with ***ception***, **vgg***, **densenet*** backbones. Default is 16.
- **decoder_channels** – A number of convolution layer filters in decoder blocks
- **in_channels** – A number of input channels for the model, default is 3 (RGB images)
- **classes** – A number of classes for output mask (or you can think as a number of channels of output mask)
- **activation** – An activation function to apply after the final convolution layer. Available options are “**sigmoid**”, “**softmax**”, “**logsoftmax**”, “**tanh**”, “**identity**”, callable and **None**.

Default is **None**

- **upsampling** – Final upsampling factor. Default is 4 to preserve input-output spatial shape identity
- **aux_params** – Dictionary with parameters of the auxiliary output (classification head). Auxiliary output is build on top of encoder if **aux_params** is not **None** (default). Supported params:
 - **classes** (int): A number of classes
 - **pooling** (str): One of “**max**”, “**avg**”. Default is “**avg**”
 - **dropout** (float): Dropout factor in [0, 1)
 - **activation** (str): An activation function to apply “**sigmoid**”/”**softmax**” (could be **None** to return logits)

Returns**PAN****Return type**`torch.nn.Module`

3.8 DeepLabV3

```
class segmentation_models_pytorch.DeepLabV3(encoder_name='resnet34', encoder_depth=5,
                                             encoder_weights='imagenet', decoder_channels=256,
                                             in_channels=3, classes=1, activation=None,
                                             upsampling=8, aux_params=None)
```

DeepLabV3 implementation from “Rethinking Atrous Convolution for Semantic Image Segmentation”

Parameters

- **encoder_name** – Name of the classification model that will be used as an encoder (a.k.a backbone) to extract features of different spatial resolution
- **encoder_depth** – A number of stages used in encoder in range [3, 5]. Each stage generate features two times smaller in spatial dimensions than previous one (e.g. for depth 0 we will have features with shapes [(N, C, H, W),], for depth 1 - [(N, C, H, W), (N, C, H // 2, W // 2)] and so on). Default is 5
- **encoder_weights** – One of **None** (random initialization), “**imagenet**” (pre-training on ImageNet) and other pretrained weights (see table with available weights for each encoder_name)
- **decoder_channels** – A number of convolution filters in ASPP module. Default is 256
- **in_channels** – A number of input channels for the model, default is 3 (RGB images)
- **classes** – A number of classes for output mask (or you can think as a number of channels of output mask)
- **activation** – An activation function to apply after the final convolution layer. Available options are “**sigmoid**”, “**softmax**”, “**logsoftmax**”, “**tanh**”, “**identity**”, **callable** and **None**.
Default is **None**
- **upsampling** – Final upsampling factor. Default is 8 to preserve input-output spatial shape identity
- **aux_params** – Dictionary with parameters of the auxiliary output (classification head). Auxiliary output is build on top of encoder if **aux_params** is not **None** (default). Supported params:
 - **classes** (int): A number of classes
 - **pooling** (str): One of “**max**”, “**avg**”. Default is “**avg**”
 - **dropout** (float): Dropout factor in [0, 1)
 - **activation** (str): An activation function to apply “**sigmoid**”/”**softmax**” (could be **None** to return logits)

Returns**DeepLabV3**

Return type
`torch.nn.Module`

3.9 DeepLabV3+

```
class segmentation_models_pytorch.DeepLabV3Plus(encoder_name='resnet34', encoder_depth=5,
                                                encoder_weights='imagenet',
                                                encoder_output_stride=16, decoder_channels=256,
                                                decoder_atrous_rates=(12, 24, 36), in_channels=3,
                                                classes=1, activation=None, upsampling=4,
                                                aux_params=None)
```

DeepLabV3+ implementation from “Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation”

Parameters

- **encoder_name** – Name of the classification model that will be used as an encoder (a.k.a backbone) to extract features of different spatial resolution
- **encoder_depth** – A number of stages used in encoder in range [3, 5]. Each stage generate features two times smaller in spatial dimensions than previous one (e.g. for depth 0 we will have features with shapes [(N, C, H, W),], for depth 1 - [(N, C, H, W), (N, C, H // 2, W // 2)] and so on). Default is 5
- **encoder_weights** – One of **None** (random initialization), “**imagenet**” (pre-training on ImageNet) and other pretrained weights (see table with available weights for each encoder_name)
- **encoder_output_stride** – Downsampling factor for last encoder features (see original paper for explanation)
- **decoder_atrous_rates** – Dilation rates for ASPP module (should be a tuple of 3 integer values)
- **decoder_channels** – A number of convolution filters in ASPP module. Default is 256
- **in_channels** – A number of input channels for the model, default is 3 (RGB images)
- **classes** – A number of classes for output mask (or you can think as a number of channels of output mask)
- **activation** – An activation function to apply after the final convolution layer. Available options are “**sigmoid**”, “**softmax**”, “**logsoftmax**”, “**tanh**”, “**identity**”, callable and **None**.

Default is **None**

- **upsampling** – Final upsampling factor. Default is 4 to preserve input-output spatial shape identity
- **aux_params** – Dictionary with parameters of the auxiliary output (classification head). Auxiliary output is build on top of encoder if **aux_params** is not **None** (default). Supported params:
 - **classes** (int): A number of classes
 - **pooling** (str): One of “**max**”, “**avg**”. Default is “**avg**”
 - **dropout** (float): Dropout factor in [0, 1)

- **activation (str):** An activation function to apply “sigmoid”/”softmax”
(could be **None** to return logits)

Returns**DeepLabV3Plus****Return type**`torch.nn.Module`**Reference:**<https://arxiv.org/abs/1802.02611v3>

AVAILABLE ENCODERS

4.1 ResNet

Encoder	Weights	Params, M
resnet18	imagenet / ssl / swsl	11M
resnet34	imagenet	21M
resnet50	imagenet / ssl / swsl	23M
resnet101	imagenet	42M
resnet152	imagenet	58M

4.2 ResNeXt

Encoder	Weights	Params, M
resnext50_32x4d	imagenet / ssl / swsl	22M
resnext101_32x4d	ssl / swsl	42M
resnext101_32x8d	imagenet / instagram / ssl / swsl	86M
resnext101_32x16d	instagram / ssl / swsl	191M
resnext101_32x32d	instagram	466M
resnext101_32x48d	instagram	826M

4.3 ResNeSt

Encoder	Weights	Params, M
timm-resnest14d	imagenet	8M
timm-resnest26d	imagenet	15M
timm-resnest50d	imagenet	25M
timm-resnest101e	imagenet	46M
timm-resnest200e	imagenet	68M
timm-resnest269e	imagenet	108M
timm-resnest50d_4s2x40d	imagenet	28M
timm-resnest50d_1s4x24d	imagenet	23M

4.4 Res2Ne(X)t

Encoder	Weights	Params, M
timm-res2net50_26w_4s	imagenet	23M
timm-res2net101_26w_4s	imagenet	43M
timm-res2net50_26w_6s	imagenet	35M
timm-res2net50_26w_8s	imagenet	46M
timm-res2net50_48w_2s	imagenet	23M
timm-res2net50_14w_8s	imagenet	23M
timm-res2next50	imagenet	22M

4.5 RegNet(x/y)

Encoder	Weights	Params, M
timm-regnetx_002	imagenet	2M
timm-regnetx_004	imagenet	4M
timm-regnetx_006	imagenet	5M
timm-regnetx_008	imagenet	6M
timm-regnetx_016	imagenet	8M
timm-regnetx_032	imagenet	14M
timm-regnetx_040	imagenet	20M
timm-regnetx_064	imagenet	24M
timm-regnetx_080	imagenet	37M
timm-regnetx_120	imagenet	43M
timm-regnetx_160	imagenet	52M
timm-regnetx_320	imagenet	105M
timm-regnety_002	imagenet	2M
timm-regnety_004	imagenet	3M
timm-regnety_006	imagenet	5M
timm-regnety_008	imagenet	5M
timm-regnety_016	imagenet	10M
timm-regnety_032	imagenet	17M
timm-regnety_040	imagenet	19M
timm-regnety_064	imagenet	29M
timm-regnety_080	imagenet	37M
timm-regnety_120	imagenet	49M
timm-regnety_160	imagenet	80M
timm-regnety_320	imagenet	141M

4.6 GERNet

Encoder	Weights	Params, M
timm-gernet_s	imagenet	6M
timm-gernet_m	imagenet	18M
timm-gernet_l	imagenet	28M

4.7 SE-Net

Encoder	Weights	Params, M
senet154	imagenet	113M
se_resnet50	imagenet	26M
se_resnet101	imagenet	47M
se_resnet152	imagenet	64M
se_resnext50_32x4d	imagenet	25M
se_resnext101_32x4d	imagenet	46M

4.8 SK-ResNe(X)t

Encoder	Weights	Params, M
timm-skresnet18	imagenet	11M
timm-skresnet34	imagenet	21M
timm-skresnext50_32x4d	imagenet	25M

4.9 DenseNet

Encoder	Weights	Params, M
densenet121	imagenet	6M
densenet169	imagenet	12M
densenet201	imagenet	18M
densenet161	imagenet	26M

4.10 Inception

Encoder	Weights	Params, M
inceptionresnetv2	imagenet / imagenet+background	54M
inceptionv4	imagenet / imagenet+background	41M
xception	imagenet	22M

4.11 EfficientNet

Encoder	Weights	Params, M
efficientnet-b0	imagenet	4M
efficientnet-b1	imagenet	6M
efficientnet-b2	imagenet	7M
efficientnet-b3	imagenet	10M
efficientnet-b4	imagenet	17M
efficientnet-b5	imagenet	28M
efficientnet-b6	imagenet	40M
efficientnet-b7	imagenet	63M
timm-efficientnet-b0	imagenet / advprop / noisy-student	4M
timm-efficientnet-b1	imagenet / advprop / noisy-student	6M
timm-efficientnet-b2	imagenet / advprop / noisy-student	7M
timm-efficientnet-b3	imagenet / advprop / noisy-student	10M
timm-efficientnet-b4	imagenet / advprop / noisy-student	17M
timm-efficientnet-b5	imagenet / advprop / noisy-student	28M
timm-efficientnet-b6	imagenet / advprop / noisy-student	40M
timm-efficientnet-b7	imagenet / advprop / noisy-student	63M
timm-efficientnet-b8	imagenet / advprop	84M
timm-efficientnet-l2	noisy-student / noisy-student-475	474M
timm-efficientnet-lite0	imagenet	4M
timm-efficientnet-lite1	imagenet	4M
timm-efficientnet-lite2	imagenet	6M
timm-efficientnet-lite3	imagenet	8M
timm-efficientnet-lite4	imagenet	13M

4.12 MobileNet

Encoder	Weights	Params, M
mobilenet_v2	imagenet	2M
timm-mobilenetv3_large_075	imagenet	1.78M
timm-mobilenetv3_large_100	imagenet	2.97M
timm-mobilenetv3_large_minimal_100	imagenet	1.41M
timm-mobilenetv3_small_075	imagenet	0.57M
timm-mobilenetv3_small_100	imagenet	0.93M
timm-mobilenetv3_small_minimal_100	imagenet	0.43M

4.13 DPN

Encoder	Weights	Params, M
dpn68	imagenet	11M
dpn68b	imagenet+5k	11M
dpn92	imagenet+5k	34M
dpn98	imagenet	58M
dpn107	imagenet+5k	84M
dpn131	imagenet	76M

4.14 VGG

Encoder	Weights	Params, M
vgg11	imagenet	9M
vgg11_bn	imagenet	9M
vgg13	imagenet	9M
vgg13_bn	imagenet	9M
vgg16	imagenet	14M
vgg16_bn	imagenet	14M
vgg19	imagenet	20M
vgg19_bn	imagenet	20M

4.15 Mix Visual Transformer

Encoder	Weights	Params, M
mit_b0	imagenet	3M
mit_b1	imagenet	13M
mit_b2	imagenet	24M
mit_b3	imagenet	44M
mit_b4	imagenet	60M
mit_b5	imagenet	81M

4.16 MobileOne

Encoder	Weights	Params, M
mobileone_s0	imagenet	4.6M
mobileone_s1	imagenet	4.0M
mobileone_s2	imagenet	6.5M
mobileone_s3	imagenet	8.8M
mobileone_s4	imagenet	13.6M

TIMM ENCODERS

Pytorch Image Models (a.k.a. timm) has a lot of pretrained models and interface which allows using these models as encoders in smp, however, not all models are supported

- not all transformer models have `features_only` functionality implemented that is required for encoder
- some models have inappropriate strides

Below is a table of suitable encoders (for DeepLabV3, DeepLabV3+, and PAN dilation support is needed also)

Total number of encoders: 549

Note: To use following encoders you have to add prefix `tu-`, e.g. `tu-adv_inception_v3`

Encoder name	Support dilation
SelecSls42	
SelecSls42b	
SelecSls60	
SelecSls60b	
SelecSls84	
bat_resnext26ts	
botnet26t_256	
botnet50ts_256	
coatnet_0_224	
coatnet_0_rw_224	
coatnet_1_224	
coatnet_1_rw_224	
coatnet_2_224	
coatnet_2_rw_224	
coatnet_3_224	
coatnet_3_rw_224	
coatnet_4_224	
coatnet_5_224	
coatnet_bn_0_rw_224	
coatnet_nano_cc_224	
coatnet_nano_rw_224	
coatnet_pico_rw_224	
coatnet_rmlp_0_rw_224	
coatnet_rmlp_1_rw2_224	
coatnet_rmlp_1_rw_224	

continues on next page

Table 1 – continued from previous page

Encoder name	Support dilation
coatnet_rmlp_2_rw_224	
coatnet_rmlp_2_rw_384	
coatnet_rmlp_3_rw_224	
coatnet_rmlp_nano_rw_224	
coatnext_nano_rw_224	
cs3darknet_focus_l	
cs3darknet_focus_m	
cs3darknet_focus_s	
cs3darknet_focus_x	
cs3darknet_l	
cs3darknet_m	
cs3darknet_s	
cs3darknet_x	
cs3edgenet_x	
cs3se_edgenet_x	
cs3sedarknet_l	
cs3sedarknet_x	
cs3sedarknet_xdw	
cspresnet50	
cspresnet50d	
cspresnet50w	
cspresnext50	
densenet121	
densenet161	
densenet169	
densenet201	
densenet264d	
densenetblur121d	
dla102	
dla102x	
dla102x2	
dla169	
dla34	
dla46_c	
dla46x_c	
dla60	
dla60_res2net	
dla60_res2next	
dla60x	
dla60x_c	
dm_nfnet_f0	
dm_nfnet_f1	
dm_nfnet_f2	
dm_nfnet_f3	
dm_nfnet_f4	
dm_nfnet_f5	
dm_nfnet_f6	
dpn107	
dpn131	

continues on next page

Table 1 – continued from previous page

Encoder name	Support dilation
dpn48b	
dpn68	
dpn68b	
dpn92	
dpn98	
eca_botnext26ts_256	
eca_halonext26ts	
eca_nfnet_l0	
eca_nfnet_l1	
eca_nfnet_l2	
eca_nfnet_l3	
eca_resnet33ts	
eca_resnext26ts	
eca_vovnet39b	
ecaresnet101d	
ecaresnet101d_pruned	
ecaresnet200d	
ecaresnet269d	
ecaresnet26t	
ecaresnet50d	
ecaresnet50d_pruned	
ecaresnet50t	
ecaresnetlight	
ecaresnext26t_32x4d	
ecaresnext50t_32x4d	
efficientnet_b0	
efficientnet_b0_g16_evos	
efficientnet_b0_g8_gn	
efficientnet_b0_gn	
efficientnet_b1	
efficientnet_b1_pruned	
efficientnet_b2	
efficientnet_b2_pruned	
efficientnet_b2a	
efficientnet_b3	
efficientnet_b3_g8_gn	
efficientnet_b3_gn	
efficientnet_b3_pruned	
efficientnet_b3a	
efficientnet_b4	
efficientnet_b5	
efficientnet_b6	
efficientnet_b7	
efficientnet_b8	
efficientnet_cc_b0_4e	
efficientnet_cc_b0_8e	
efficientnet_cc_b1_8e	
efficientnet_el	
efficientnet_el_pruned	

continues on next page

Table 1 – continued from previous page

Encoder name	Support dilation
efficientnet_em	
efficientnet_es	
efficientnet_es_pruned	
efficientnet_l2	
efficientnet_lite0	
efficientnet_lite1	
efficientnet_lite2	
efficientnet_lite3	
efficientnet_lite4	
efficientnetv2_l	
efficientnetv2_m	
efficientnetv2_rw_m	
efficientnetv2_rw_s	
efficientnetv2_rw_t	
efficientnetv2_s	
efficientnetv2_xl	
ese_vovnet19b_dw	
ese_vovnet19b_slim	
ese_vovnet19b_slim_dw	
ese_vovnet39b	
ese_vovnet39b_evos	
ese_vovnet57b	
ese_vovnet99b	
fbnetc_100	
fbnetv3_b	
fbnetv3_d	
fbnetv3_g	
gc_efficientnetv2_rw_t	
gcrsnet33ts	
gcrsnet50t	
gcrsnext26ts	
gcrsnext50ts	
gernet_l	
gernet_m	
gernet_s	
ghostnet_050	
ghostnet_100	
ghostnet_130	
halo2botnet50ts_256	
halonet26t	
halonet50ts	
halonet_h1	
haloregnetz_b	
hardcorenas_a	
hardcorenas_b	
hardcorenas_c	
hardcorenas_d	
hardcorenas_e	
hardcorenas_f	

continues on next page

Table 1 – continued from previous page

Encoder name	Support dilation
hrnet_w18	
hrnet_w18_small	
hrnet_w18_small_v2	
hrnet_w18_ssld	
hrnet_w30	
hrnet_w32	
hrnet_w40	
hrnet_w44	
hrnet_w48	
hrnet_w48_ssld	
hrnet_w64	
inception_resnet_v2	
inception_v3	
inception_v4	
lambda_resnet26rpt_256	
lambda_resnet26t	
lambda_resnet50ts	
lamhalobotnet50ts_256	
lcnet_035	
lcnet_050	
lcnet_075	
lcnet_100	
lcnet_150	
legacy_senet154	
legacy_seresnet101	
legacy_seresnet152	
legacy_seresnet18	
legacy_seresnet34	
legacy_seresnet50	
legacy_seresnext101_32x4d	
legacy_seresnext26_32x4d	
legacy_seresnext50_32x4d	
legacy_xception	
maxvit_base_tf_224	
maxvit_base_tf_384	
maxvit_base_tf_512	
maxvit_large_tf_224	
maxvit_large_tf_384	
maxvit_large_tf_512	
maxvit_nano_rw_256	
maxvit_pico_rw_256	
maxvit_rmlp_base_rw_224	
maxvit_rmlp_base_rw_384	
maxvit_rmlp_nano_rw_256	
maxvit_rmlp_pico_rw_256	
maxvit_rmlp_small_rw_224	
maxvit_rmlp_small_rw_256	
maxvit_rmlp_tiny_rw_256	
maxvit_small_tf_224	

continues on next page

Table 1 – continued from previous page

Encoder name	Support dilation
maxvit_small_tf_384	
maxvit_small_tf_512	
maxvit_tiny_pm_256	
maxvit_tiny_rw_224	
maxvit_tiny_rw_256	
maxvit_tiny_tf_224	
maxvit_tiny_tf_384	
maxvit_tiny_tf_512	
maxvit_xlarge_tf_224	
maxvit_xlarge_tf_384	
maxvit_xlarge_tf_512	
maxxvit_rmlp_nano_rw_256	
maxxvit_rmlp_small_rw_256	
maxxvit_rmlp_tiny_rw_256	
maxxvitv2_nano_rw_256	
maxxvitv2_rmlp_base_rw_224	
maxxvitv2_rmlp_base_rw_384	
maxxvitv2_rmlp_large_rw_224	
mixnet_l	
mixnet_m	
mixnet_s	
mixnet_xl	
mixnet_xxl	
mnasnet_050	
mnasnet_075	
mnasnet_100	
mnasnet_140	
mnasnet_a1	
mnasnet_b1	
mnasnet_small	
mobilenetv2_035	
mobilenetv2_050	
mobilenetv2_075	
mobilenetv2_100	
mobilenetv2_110d	
mobilenetv2_120d	
mobilenetv2_140	
mobilenetv3_large_075	
mobilenetv3_large_100	
mobilenetv3_rw	
mobilenetv3_small_050	
mobilenetv3_small_075	
mobilenetv3_small_100	
mobilevit_s	
mobilevit_xs	
mobilevit_xxs	
mobilevitv2_050	
mobilevitv2_075	
mobilevitv2_100	

continues on next page

Table 1 – continued from previous page

Encoder name	Support dilation
mobilevitv2_125	
mobilevitv2_150	
mobilevitv2_175	
mobilevitv2_200	
nasnetalarge	
nf_ecaresnet101	
nf_ecaresnet26	
nf_ecaresnet50	
nf_regnet_b0	
nf_regnet_b1	
nf_regnet_b2	
nf_regnet_b3	
nf_regnet_b4	
nf_regnet_b5	
nf_resnet101	
nf_resnet26	
nf_resnet50	
nf_seresnet101	
nf_seresnet26	
nf_seresnet50	
nfnet_f0	
nfnet_f1	
nfnet_f2	
nfnet_f3	
nfnet_f4	
nfnet_f5	
nfnet_f6	
nfnet_f7	
nfnet_l0	
pnasnet5large	
regnetv_040	
regnetv_064	
regnetx_002	
regnetx_004	
regnetx_004_tv	
regnetx_006	
regnetx_008	
regnetx_016	
regnetx_032	
regnetx_040	
regnetx_064	
regnetx_080	
regnetx_120	
regnetx_160	
regnetx_320	
regnety_002	
regnety_004	
regnety_006	
regnety_008	

continues on next page

Table 1 – continued from previous page

Encoder name	Support dilation
regnety_008_tv	
regnety_016	
regnety_032	
regnety_040	
regnety_040_sgn	
regnety_064	
regnety_080	
regnety_080_tv	
regnety_120	
regnety_1280	
regnety_160	
regnety_2560	
regnety_320	
regnety_640	
regnetz_005	
regnetz_040	
regnetz_040_h	
regnetz_b16	
regnetz_b16_evos	
regnetz_c16	
regnetz_c16_evos	
regnetz_d32	
regnetz_d8	
regnetz_d8_evos	
regnetz_e8	
repvgg_a2	
repvgg_b0	
repvgg_b1	
repvgg_b1g4	
repvgg_b2	
repvgg_b2g4	
repvgg_b3	
repvgg_b3g4	
res2net101_26w_4s	
res2net101d	
res2net50_14w_8s	
res2net50_26w_4s	
res2net50_26w_6s	
res2net50_26w_8s	
res2net50_48w_2s	
res2net50d	
res2next50	
resnest101e	
resnest14d	
resnest200e	
resnest269e	
resnest26d	
resnest50d	
resnest50d_1s4x24d	

continues on next page

Table 1 – continued from previous page

Encoder name	Support dilation
resnest50d_4s2x40d	
resnet101	
resnet101c	
resnet101d	
resnet101s	
resnet10t	
resnet14t	
resnet152	
resnet152c	
resnet152d	
resnet152s	
resnet18	
resnet18d	
resnet200	
resnet200d	
resnet26	
resnet26d	
resnet26t	
resnet32ts	
resnet33ts	
resnet34	
resnet34d	
resnet50	
resnet50_gn	
resnet50c	
resnet50d	
resnet50s	
resnet50t	
resnet51q	
resnet61q	
resnetaa101d	
resnetaa34d	
resnetaa50	
resnetaa50d	
resnetblur101d	
resnetblur18	
resnetblur50	
resnetblur50d	
resnetrs101	
resnetrs152	
resnetrs200	
resnetrs270	
resnetrs350	
resnetrs420	
resnetrs50	
resnetv2_101	
resnetv2_101d	
resnetv2_101x1_bit	
resnetv2_101x3_bit	

continues on next page

Table 1 – continued from previous page

Encoder name	Support dilation
resnetv2_152	
resnetv2_152d	
resnetv2_152x2_bit	
resnetv2_152x4_bit	
resnetv2_50	
resnetv2_50d	
resnetv2_50d_evos	
resnetv2_50d_frn	
resnetv2_50d_gn	
resnetv2_50t	
resnetv2_50x1_bit	
resnetv2_50x3_bit	
resnext101_32x16d	
resnext101_32x32d	
resnext101_32x4d	
resnext101_32x8d	
resnext101_64x4d	
resnext26ts	
resnext50_32x4d	
resnext50d_32x4d	
rexnet_100	
rexnet_130	
rexnet_150	
rexnet_200	
rexnet_300	
rexnetr_100	
rexnetr_130	
rexnetr_150	
rexnetr_200	
rexnetr_300	
sebotnet33ts_256	
sehalonet33ts	
semnasnet_050	
semnasnet_075	
semnasnet_100	
semnasnet_140	
senet154	
seresnet101	
seresnet152	
seresnet152d	
seresnet18	
seresnet200d	
seresnet269d	
seresnet33ts	
seresnet34	
seresnet50	
seresnet50t	
seresnetaa50d	
seresnext101_32x4d	

continues on next page

Table 1 – continued from previous page

Encoder name	Support dilation
seresnext101_32x8d	
seresnext101_64x4d	
seresnext101d_32x8d	
seresnext26d_32x4d	
seresnext26t_32x4d	
seresnext26tn_32x4d	
seresnext26ts	
seresnext50_32x4d	
seresnextaa101d_32x8d	
skresnet18	
skresnet34	
skresnet50	
skresnet50d	
skresnext50_32x4d	
spnasnet_100	
tf_efficientnet_b0	
tf_efficientnet_b1	
tf_efficientnet_b2	
tf_efficientnet_b3	
tf_efficientnet_b4	
tf_efficientnet_b5	
tf_efficientnet_b6	
tf_efficientnet_b7	
tf_efficientnet_b8	
tf_efficientnet_cc_b0_4e	
tf_efficientnet_cc_b0_8e	
tf_efficientnet_cc_b1_8e	
tf_efficientnet_el	
tf_efficientnet_em	
tf_efficientnet_es	
tf_efficientnet_l2	
tf_efficientnet_lite0	
tf_efficientnet_lite1	
tf_efficientnet_lite2	
tf_efficientnet_lite3	
tf_efficientnet_lite4	
tf_efficientnetv2_b0	
tf_efficientnetv2_b1	
tf_efficientnetv2_b2	
tf_efficientnetv2_b3	
tf_efficientnetv2_l	
tf_efficientnetv2_m	
tf_efficientnetv2_s	
tf_efficientnetv2_xl	
tf_mixnet_l	
tf_mixnet_m	
tf_mixnet_s	
tf_mobilenetv3_large_075	
tf_mobilenetv3_large_100	

continues on next page

Table 1 – continued from previous page

Encoder name	Support dilation
tf_mobilenetv3_large_minimal_100	
tf_mobilenetv3_small_075	
tf_mobilenetv3_small_100	
tf_mobilenetv3_small_minimal_100	
tinynet_a	
tinynet_b	
tinynet_c	
tinynet_d	
tinynet_e	
vovnet39a	
vovnet57a	
wide_resnet101_2	
wide_resnet50_2	
xception41	
xception41p	
xception65	
xception65p	
xception71	

LOSSES

Collection of popular semantic segmentation losses. Adapted from an awesome repo with pytorch utils <https://github.com/BloodAxe/pytorch-toolbelt>

6.1 Constants

`segmentation_models_pytorch.losses.constants.BINARY_MODE: str = 'binary'`

Loss binary mode suppose you are solving binary segmentation task. That mean you have only one class which pixels are labeled as **1**, the rest pixels are background and labeled as **0**. Target mask shape - (N, H, W), model output mask shape (N, 1, H, W).

`segmentation_models_pytorch.losses.constants.MULTICLASS_MODE: str = 'multiclass'`

Loss multiclass mode suppose you are solving multi-**class** segmentation task. That mean you have $C = 1..N$ classes which have unique label values, classes are mutually exclusive and all pixels are labeled with these values. Target mask shape - (N, H, W), model output mask shape (N, C, H, W).

`segmentation_models_pytorch.losses.constants.MULTILABEL_MODE: str = 'multilabel'`

Loss multilabel mode suppose you are solving multi-**label** segmentation task. That mean you have $C = 1..N$ classes which pixels are labeled as **1**, classes are not mutually exclusive and each class have its own *channel*, pixels in each channel which are not belong to class labeled as **0**. Target mask shape - (N, C, H, W), model output mask shape (N, C, H, W).

6.2 JaccardLoss

`class segmentation_models_pytorch.losses.JaccardLoss(mode, classes=None, log_loss=False, from_logits=True, smooth=0.0, eps=1e-07)`

Jaccard loss for image segmentation task. It supports binary, multiclass and multilabel cases

Parameters

- **mode** – Loss mode ‘binary’, ‘multiclass’ or ‘multilabel’
- **classes** – List of classes that contribute in loss computation. By default, all channels are included.
- **log_loss** – If True, loss computed as $-\log(\text{jaccard_coeff})$, otherwise $1 - \text{jaccard_coeff}$
- **from_logits** – If True, assumes input is raw logits
- **smooth** – Smoothness constant for dice coefficient

- **eps** – A small epsilon for numerical stability to avoid zero division error (denominator will be always greater or equal to eps)

Shape

- **y_pred** - torch.Tensor of shape (N, C, H, W)
- **y_true** - torch.Tensor of shape (N, H, W) or (N, C, H, W)

Reference

<https://github.com/BloodAxe/pytorch-toolbelt>

6.3 DiceLoss

```
class segmentation_models_pytorch.losses.DiceLoss(mode, classes=None, log_loss=False,  
                                                from_logits=True, smooth=0.0,  
                                                ignore_index=None, eps=1e-07)
```

Dice loss for image segmentation task. It supports binary, multiclass and multilabel cases

Parameters

- **mode** – Loss mode ‘binary’, ‘multiclass’ or ‘multilabel’
- **classes** – List of classes that contribute in loss computation. By default, all channels are included.
- **log_loss** – If True, loss computed as $-\log(\text{dice_coeff})$, otherwise $1 - \text{dice_coeff}$
- **from_logits** – If True, assumes input is raw logits
- **smooth** – Smoothness constant for dice coefficient (a)
- **ignore_index** – Label that indicates ignored pixels (does not contribute to loss)
- **eps** – A small epsilon for numerical stability to avoid zero division error (denominator will be always greater or equal to eps)

Shape

- **y_pred** - torch.Tensor of shape (N, C, H, W)
- **y_true** - torch.Tensor of shape (N, H, W) or (N, C, H, W)

Reference

<https://github.com/BloodAxe/pytorch-toolbelt>

6.4 TverskyLoss

```
class segmentation_models_pytorch.losses.TverskyLoss(mode, classes=None, log_loss=False,  
                                                    from_logits=True, smooth=0.0,  
                                                    ignore_index=None, eps=1e-07, alpha=0.5,  
                                                    beta=0.5, gamma=1.0)
```

Tversky loss for image segmentation task. Where FP and FN is weighted by alpha and beta params. With $\alpha == \beta == 0.5$, this loss becomes equal DiceLoss. It supports binary, multiclass and multilabel cases

Parameters

- **mode** – Metric mode {‘binary’, ‘multiclass’, ‘multilabel’}

- **classes** – Optional list of classes that contribute in loss computation;
- **default** (*By*) –
- **included.** (*all channels are*) –
- **log_loss** – If True, loss computed as $-\log(\text{tversky})$ otherwise $1 - \text{tversky}$
- **from_logits** – If True assumes input is raw logits
- **smooth** –
- **ignore_index** – Label that indicates ignored pixels (does not contribute to loss)
- **eps** – Small epsilon for numerical stability
- **alpha** – Weight constant that penalize model for FPs (False Positives)
- **beta** – Weight constant that penalize model for FNs (False Negatives)
- **gamma** – Constant that squares the error function. Defaults to 1.0

Returns

torch.Tensor

Return type

loss

6.5 FocalLoss

```
class segmentation_models_pytorch.losses.FocalLoss(mode, alpha=None, gamma=2.0,
                                                    ignore_index=None, reduction='mean',
                                                    normalized=False, reduced_threshold=None)
```

Compute Focal loss

Parameters

- **mode** – Loss mode ‘binary’, ‘multiclass’ or ‘multilabel’
- **alpha** – Prior probability of having positive value in target.
- **gamma** – Power factor for dampening weight (focal strength).
- **ignore_index** – If not None, targets may contain values to be ignored. Target values equal to ignore_index will be ignored from loss computation.
- **normalized** – Compute normalized focal loss (<https://arxiv.org/pdf/1909.07829.pdf>).
- **reduced_threshold** – Switch to reduced focal loss. Note, when using this mode you should use *reduction="sum"*.

Shape

- **y_pred** - torch.Tensor of shape (N, C, H, W)
- **y_true** - torch.Tensor of shape (N, H, W) or (N, C, H, W)

Reference
<https://github.com/BloodAxe/pytorch-toolbelt>

6.6 LovaszLoss

```
class segmentation_models_pytorch.losses.LovaszLoss(mode, per_image=False, ignore_index=None,
                                                    from_logits=True)
```

Lovasz loss for image segmentation task. It supports binary, multiclass and multilabel cases

Parameters

- **mode** – Loss mode ‘binary’, ‘multiclass’ or ‘multilabel’
- **ignore_index** – Label that indicates ignored pixels (does not contribute to loss)
- **per_image** – If True loss computed per each image and then averaged, else computed per whole batch

Shape

- **y_pred** - torch.Tensor of shape (N, C, H, W)
- **y_true** - torch.Tensor of shape (N, H, W) or (N, C, H, W)

Reference

<https://github.com/BloodAxe/pytorch-toolbelt>

6.7 SoftBCEWithLogitsLoss

```
class segmentation_models_pytorch.losses.SoftBCEWithLogitsLoss(weight=None, ignore_index=-100,
                                                                reduction='mean',
                                                                smooth_factor=None,
                                                                pos_weight=None)
```

Drop-in replacement for torch.nn.BCEWithLogitsLoss with few additions: ignore_index and label_smoothing

Parameters

- **ignore_index** – Specifies a target value that is ignored and does not contribute to the input gradient.
- **smooth_factor** – Factor to smooth target (e.g. if smooth_factor=0.1 then [1, 0, 1] -> [0.9, 0.1, 0.9])

Shape

- **y_pred** - torch.Tensor of shape NxCxHxW
- **y_true** - torch.Tensor of shape NxHxW or Nx1xHxW

Reference

<https://github.com/BloodAxe/pytorch-toolbelt>

6.8 SoftCrossEntropyLoss

```
class segmentation_models_pytorch.losses.SoftCrossEntropyLoss(reduction='mean',
                                                                smooth_factor=None,
                                                                ignore_index=-100, dim=1)
```

Drop-in replacement for torch.nn.CrossEntropyLoss with label_smoothing

Parameters

smooth_factor – Factor to smooth target (e.g. if smooth_factor=0.1 then [1, 0, 0] -> [0.9, 0.05, 0.05])

Shape

- **y_pred** - torch.Tensor of shape (N, C, H, W)
- **y_true** - torch.Tensor of shape (N, H, W)

Reference

<https://github.com/BloodAxe/pytorch-toolbelt>

6.9 MCCLoss

```
class segmentation_models_pytorch.losses.MCCLoss(eps=1e-05)
```

Compute Matthews Correlation Coefficient Loss for image segmentation task. It only supports binary mode.

Parameters

eps (*float*) – Small epsilon to handle situations where all the samples in the dataset belong to one class

Reference:

<https://github.com/kakumarabhishek/MCC-Loss>

```
forward(y_pred, y_true)
```

Compute MCC loss

Parameters

- **y_pred** (*torch.Tensor*) – model prediction of shape (N, H, W) or (N, 1, H, W)
- **y_true** (*torch.Tensor*) – ground truth labels of shape (N, H, W) or (N, 1, H, W)

Returns

loss value (1 - mcc)

Return type

torch.Tensor

7.1 Functional metrics

Various metrics based on Type I and Type II errors.

References

https://en.wikipedia.org/wiki/Confusion_matrix

Example

```
import segmentation_models_pytorch as smp

# lets assume we have multilabel prediction for 3 classes
output = torch.rand([10, 3, 256, 256])
target = torch.rand([10, 3, 256, 256]).round().long()

# first compute statistics for true positives, false positives, false negative and
# true negative "pixels"
tp, fp, fn, tn = smp.metrics.get_stats(output, target, mode='multilabel', threshold=0.5)

# then compute metrics with required reduction (see metric docs)
iou_score = smp.metrics.iou_score(tp, fp, fn, tn, reduction="micro")
f1_score = smp.metrics.f1_score(tp, fp, fn, tn, reduction="micro")
f2_score = smp.metrics.fbeta_score(tp, fp, fn, tn, beta=2, reduction="micro")
accuracy = smp.metrics.accuracy(tp, fp, fn, tn, reduction="macro")
recall = smp.metrics.recall(tp, fp, fn, tn, reduction="micro-image-wise")
```

Functions:

<code>get_stats(output, target, mode[, ...])</code>	Compute true positive, false positive, false negative, true negative 'pixels' for each image and each class.
<code>fbeta_score(tp, fp, fn, tn[, beta, ...])</code>	F beta score
<code>f1_score(tp, fp, fn, tn[, reduction, ...])</code>	F1 score
<code>iou_score(tp, fp, fn, tn[, reduction, ...])</code>	IoU score or Jaccard index
<code>accuracy(tp, fp, fn, tn[, reduction, ...])</code>	Accuracy
<code>precision(tp, fp, fn, tn[, reduction, ...])</code>	Precision or positive predictive value (PPV)
<code>recall(tp, fp, fn, tn[, reduction, ...])</code>	Sensitivity, recall, hit rate, or true positive rate (TPR)
<code>sensitivity(tp, fp, fn, tn[, reduction, ...])</code>	Sensitivity, recall, hit rate, or true positive rate (TPR)
<code>specificity(tp, fp, fn, tn[, reduction, ...])</code>	Specificity, selectivity or true negative rate (TNR)
<code>balanced_accuracy(tp, fp, fn, tn[, ...])</code>	Balanced accuracy
<code>positive_predictive_value(tp, fp, fn, tn[, ...])</code>	Precision or positive predictive value (PPV)
<code>negative_predictive_value(tp, fp, fn, tn[, ...])</code>	Negative predictive value (NPV)
<code>false_negative_rate(tp, fp, fn, tn[, ...])</code>	Miss rate or false negative rate (FNR)
<code>false_positive_rate(tp, fp, fn, tn[, ...])</code>	Fall-out or false positive rate (FPR)
<code>false_discovery_rate(tp, fp, fn, tn[, ...])</code>	False discovery rate (FDR)
<code>false_omission_rate(tp, fp, fn, tn[, ...])</code>	False omission rate (FOR)
<code>positive_likelihood_ratio(tp, fp, fn, tn[, ...])</code>	Positive likelihood ratio (LR+)
<code>negative_likelihood_ratio(tp, fp, fn, tn[, ...])</code>	Negative likelihood ratio (LR-)

`segmentation_models_pytorch.metrics.functional.get_stats(output, target, mode, ignore_index=None, threshold=None, num_classes=None)`

Compute true positive, false positive, false negative, true negative 'pixels' for each image and each class.

Parameters

- **output** (*Union[torch.LongTensor, torch.FloatTensor]*) – Model output with following shapes and types depending on the specified mode:
 - **'binary'**
shape (N, 1, ...) and torch.LongTensor or torch.FloatTensor
 - **'multilabel'**
shape (N, C, ...) and torch.LongTensor or torch.FloatTensor
 - **'multiclass'**
shape (N, ...) and torch.LongTensor
- **target** (*torch.LongTensor*) – Targets with following shapes depending on the specified mode:
 - **'binary'**
shape (N, 1, ...)
 - **'multilabel'**
shape (N, C, ...)
 - **'multiclass'**
shape (N, ...)
- **mode** (*str*) – One of 'binary' | 'multilabel' | 'multiclass'
- **ignore_index** (*Optional[int]*) – Label to ignore on for metric computation. **Not** supported for 'binary' and 'multilabel' modes. Defaults to None.
- **threshold** (*Optional[float, List[float]]*) – Binarization threshold for output in case of 'binary' or 'multilabel' modes. Defaults to None.

- **num_classes** (*Optional[int]*) – Number of classes, necessary attribute only for 'multiclass' mode. Class values should be in range 0..(num_classes - 1). If ignore_index is specified it should be outside the classes range, e.g. -1 or 255.

Raises

ValueError – in case of misconfiguration.

Returns

true_positive, false_positive, false_negative,
true_negative tensors (N, C) shape each.

Return type

Tuple[torch.LongTensor]

segmentation_models_pytorch.metrics.functional.**fbeta_score**(*tp, fp, fn, tn, beta=1.0,*
reduction=None, class_weights=None,
zero_division=1.0)

F beta score

Parameters

- **tp** (*torch.LongTensor*) – tensor of shape (N, C), true positive cases
- **fp** (*torch.LongTensor*) – tensor of shape (N, C), false positive cases
- **fn** (*torch.LongTensor*) – tensor of shape (N, C), false negative cases
- **tn** (*torch.LongTensor*) – tensor of shape (N, C), true negative cases
- **reduction** (*Optional[str]*) – Define how to aggregate metric between classes and images:
 - **'micro'**
Sum true positive, false positive, false negative and true negative pixels over all images and all classes and then compute score.
 - **'macro'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average labels scores. This does not take label imbalance into account.
 - **'weighted'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average weighted labels scores.
 - **'micro-imagewise'**
Sum true positive, false positive, false negative and true negative pixels for **each image**, then compute score for **each image** and average scores over dataset. All images contribute equally to final score, however takes into account class imbalance for each image.
 - **'macro-imagewise'**
Compute score for each image and for each class on that image separately, then compute average score on each image over labels and average image scores over dataset. Does not take into account label imbalance on each image.
 - **'weighted-imagewise'**
Compute score for each image and for each class on that image separately, then compute weighted average score on each image over labels and average image scores over dataset.

- **'none' or None**

Same as 'macro-imagewise', but without any reduction.

For 'binary' case 'micro' = 'macro' = 'weighted' and 'micro-imagewise' = 'macro-imagewise' = 'weighted-imagewise'.

Prefixes 'micro', 'macro' and 'weighted' define how the scores for classes will be aggregated, while postfix 'imagewise' defines how scores between the images will be aggregated.

- **class_weights** (*Optional[List[float]]*) – list of class weights for metric aggregation, in case of *weighted** reduction is chosen. Defaults to None.
- **zero_division** (*Union[str, float]*) – Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to “warn”, this acts as 0, but warnings are also raised. Defaults to 1.
- **beta** (*float*) –

Returns

if 'reduction' is not None or 'none' returns scalar metric,
else returns tensor of shape (N, C)

Return type

torch.Tensor

References

https://en.wikipedia.org/wiki/Confusion_matrix

segmentation_models_pytorch.metrics.functional.f1_score(*tp, fp, fn, tn, reduction=None, class_weights=None, zero_division=1.0*)

F1 score

Parameters

- **tp** (*torch.LongTensor*) – tensor of shape (N, C), true positive cases
- **fp** (*torch.LongTensor*) – tensor of shape (N, C), false positive cases
- **fn** (*torch.LongTensor*) – tensor of shape (N, C), false negative cases
- **tn** (*torch.LongTensor*) – tensor of shape (N, C), true negative cases
- **reduction** (*Optional[str]*) – Define how to aggregate metric between classes and images:
 - **'micro'**
Sum true positive, false positive, false negative and true negative pixels over all images and all classes and then compute score.
 - **'macro'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average labels scores. This does not take label imbalance into account.
 - **'weighted'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average weighted labels scores.

- **'micro-imagewise'**
Sum true positive, false positive, false negative and true negative pixels for **each image**, then compute score for **each image** and average scores over dataset. All images contribute equally to final score, however takes into account class imbalance for each image.
- **'macro-imagewise'**
Compute score for each image and for each class on that image separately, then compute average score on each image over labels and average image scores over dataset. Does not take into account label imbalance on each image.
- **'weighted-imagewise'**
Compute score for each image and for each class on that image separately, then compute weighted average score on each image over labels and average image scores over dataset.
- **'none' or None**
Same as 'macro-imagewise', but without any reduction.

For 'binary' case 'micro' = 'macro' = 'weighted' and 'micro-imagewise' = 'macro-imagewise' = 'weighted-imagewise'.

Prefixes 'micro', 'macro' and 'weighted' define how the scores for classes will be aggregated, while postfix 'imagewise' defines how scores between the images will be aggregated.

- **class_weights** (*Optional[List[float]*) – list of class weights for metric aggregation, in case of *weighted** reduction is chosen. Defaults to None.
- **zero_division** (*Union[str, float]*) – Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to “warn”, this acts as 0, but warnings are also raised. Defaults to 1.

Returns

if **'reduction' is not None or 'none'** returns scalar metric,
else returns tensor of shape (N, C)

Return type

torch.Tensor

References

https://en.wikipedia.org/wiki/Confusion_matrix

segmentation_models_pytorch.metrics.functional.iou_score(*tp, fp, fn, tn, reduction=None, class_weights=None, zero_division=1.0*)

IoU score or Jaccard index

Parameters

- **tp** (*torch.LongTensor*) – tensor of shape (N, C), true positive cases
- **fp** (*torch.LongTensor*) – tensor of shape (N, C), false positive cases
- **fn** (*torch.LongTensor*) – tensor of shape (N, C), false negative cases
- **tn** (*torch.LongTensor*) – tensor of shape (N, C), true negative cases
- **reduction** (*Optional[str]*) – Define how to aggregate metric between classes and images:

- **'micro'**
Sum true positive, false positive, false negative and true negative pixels over all images and all classes and then compute score.
- **'macro'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average labels scores. This does not take label imbalance into account.
- **'weighted'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average weighted labels scores.
- **'micro-imagewise'**
Sum true positive, false positive, false negative and true negative pixels for **each image**, then compute score for **each image** and average scores over dataset. All images contribute equally to final score, however takes into account class imbalance for each image.
- **'macro-imagewise'**
Compute score for each image and for each class on that image separately, then compute average score on each image over labels and average image scores over dataset. Does not take into account label imbalance on each image.
- **'weighted-imagewise'**
Compute score for each image and for each class on that image separately, then compute weighted average score on each image over labels and average image scores over dataset.
- **'none' or None**
Same as 'macro-imagewise', but without any reduction.

For 'binary' case 'micro' = 'macro' = 'weighted' and 'micro-imagewise' = 'macro-imagewise' = 'weighted-imagewise'.

Prefixes 'micro', 'macro' and 'weighted' define how the scores for classes will be aggregated, while postfix 'imagewise' defines how scores between the images will be aggregated.

- **class_weights** (*Optional[List[float]]*) – list of class weights for metric aggregation, in case of *weighted** reduction is chosen. Defaults to None.
- **zero_division** (*Union[str, float]*) – Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to “warn”, this acts as 0, but warnings are also raised. Defaults to 1.

Returns

if **'reduction'** is not **None** or **'none'** returns scalar metric,
else returns tensor of shape (N, C)

Return type

torch.Tensor

References

https://en.wikipedia.org/wiki/Confusion_matrix

`segmentation_models_pytorch.metrics.functional.accuracy(tp, fp, fn, tn, reduction=None, class_weights=None, zero_division=1.0)`

Accuracy

Parameters

- **tp** (*torch.LongTensor*) – tensor of shape (N, C), true positive cases
- **fp** (*torch.LongTensor*) – tensor of shape (N, C), false positive cases
- **fn** (*torch.LongTensor*) – tensor of shape (N, C), false negative cases
- **tn** (*torch.LongTensor*) – tensor of shape (N, C), true negative cases
- **reduction** (*Optional[str]*) – Define how to aggregate metric between classes and images:
 - **'micro'**
Sum true positive, false positive, false negative and true negative pixels over all images and all classes and then compute score.
 - **'macro'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average labels scores. This does not take label imbalance into account.
 - **'weighted'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average weighted labels scores.
 - **'micro-imagewise'**
Sum true positive, false positive, false negative and true negative pixels for **each image**, then compute score for **each image** and average scores over dataset. All images contribute equally to final score, however takes into account class imbalance for each image.
 - **'macro-imagewise'**
Compute score for each image and for each class on that image separately, then compute average score on each image over labels and average image scores over dataset. Does not take into account label imbalance on each image.
 - **'weighted-imagewise'**
Compute score for each image and for each class on that image separately, then compute weighted average score on each image over labels and average image scores over dataset.
 - **'none' or None**
Same as 'macro-imagewise', but without any reduction.

For 'binary' case 'micro' = 'macro' = 'weighted' and 'micro-imagewise' = 'macro-imagewise' = 'weighted-imagewise'.

Prefixes 'micro', 'macro' and 'weighted' define how the scores for classes will be aggregated, while postfix 'imagewise' defines how scores between the images will be aggregated.

- **class_weights** (*Optional[List[float]]*) – list of class weights for metric aggregation, in case of *weighted** reduction is chosen. Defaults to None.

- **zero_division** (*Union[str, float]*) – Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to “warn”, this acts as 0, but warnings are also raised. Defaults to 1.

Returns

if 'reduction' is not None or 'none' returns scalar metric,
else returns tensor of shape (N, C)

Return type

torch.Tensor

References

https://en.wikipedia.org/wiki/Confusion_matrix

`segmentation_models_pytorch.metrics.functional.precision(tp, fp, fn, tn, reduction=None, class_weights=None, zero_division=1.0)`

Precision or positive predictive value (PPV)

Parameters

- **tp** (*torch.LongTensor*) – tensor of shape (N, C), true positive cases
- **fp** (*torch.LongTensor*) – tensor of shape (N, C), false positive cases
- **fn** (*torch.LongTensor*) – tensor of shape (N, C), false negative cases
- **tn** (*torch.LongTensor*) – tensor of shape (N, C), true negative cases
- **reduction** (*Optional[str]*) – Define how to aggregate metric between classes and images:
 - **'micro'**
Sum true positive, false positive, false negative and true negative pixels over all images and all classes and then compute score.
 - **'macro'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average labels scores. This does not take label imbalance into account.
 - **'weighted'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average weighted labels scores.
 - **'micro-imagewise'**
Sum true positive, false positive, false negative and true negative pixels for **each image**, then compute score for **each image** and average scores over dataset. All images contribute equally to final score, however takes into account class imbalance for each image.
 - **'macro-imagewise'**
Compute score for each image and for each class on that image separately, then compute average score on each image over labels and average image scores over dataset. Does not take into account label imbalance on each image.
 - **'weighted-imagewise'**
Compute score for each image and for each class on that image separately, then compute weighted average score on each image over labels and average image scores over dataset.

- **'none' or None**

Same as 'macro-imagewise', but without any reduction.

For 'binary' case 'micro' = 'macro' = 'weighted' and 'micro-imagewise' = 'macro-imagewise' = 'weighted-imagewise'.

Prefixes 'micro', 'macro' and 'weighted' define how the scores for classes will be aggregated, while postfix 'imagewise' defines how scores between the images will be aggregated.

- **class_weights** (*Optional[List[float]*) – list of class weights for metric aggregation, in case of *weighted** reduction is chosen. Defaults to None.
- **zero_division** (*Union[str, float]*) – Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to “warn”, this acts as 0, but warnings are also raised. Defaults to 1.

Returns

if 'reduction' is not None or 'none' returns scalar metric,
else returns tensor of shape (N, C)

Return type

torch.Tensor

References

https://en.wikipedia.org/wiki/Confusion_matrix

`segmentation_models_pytorch.metrics.functional.recall(tp, fp, fn, tn, reduction=None, class_weights=None, zero_division=1.0)`

Sensitivity, recall, hit rate, or true positive rate (TPR)

Parameters

- **tp** (*torch.LongTensor*) – tensor of shape (N, C), true positive cases
- **fp** (*torch.LongTensor*) – tensor of shape (N, C), false positive cases
- **fn** (*torch.LongTensor*) – tensor of shape (N, C), false negative cases
- **tn** (*torch.LongTensor*) – tensor of shape (N, C), true negative cases
- **reduction** (*Optional[str]*) – Define how to aggregate metric between classes and images:
 - **'micro'**
Sum true positive, false positive, false negative and true negative pixels over all images and all classes and then compute score.
 - **'macro'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average labels scores. This does not take label imbalance into account.
 - **'weighted'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average weighted labels scores.

- **'micro-imagewise'**
Sum true positive, false positive, false negative and true negative pixels for **each image**, then compute score for **each image** and average scores over dataset. All images contribute equally to final score, however takes into account class imbalance for each image.
- **'macro-imagewise'**
Compute score for each image and for each class on that image separately, then compute average score on each image over labels and average image scores over dataset. Does not take into account label imbalance on each image.
- **'weighted-imagewise'**
Compute score for each image and for each class on that image separately, then compute weighted average score on each image over labels and average image scores over dataset.
- **'none' or None**
Same as 'macro-imagewise', but without any reduction.

For 'binary' case 'micro' = 'macro' = 'weighted' and 'micro-imagewise' = 'macro-imagewise' = 'weighted-imagewise'.

Prefixes 'micro', 'macro' and 'weighted' define how the scores for classes will be aggregated, while postfix 'imagewise' defines how scores between the images will be aggregated.

- **class_weights** (*Optional[List[float]]*) – list of class weights for metric aggregation, in case of *weighted** reduction is chosen. Defaults to None.
- **zero_division** (*Union[str, float]*) – Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to “warn”, this acts as 0, but warnings are also raised. Defaults to 1.

Returns

if **'reduction' is not None or 'none'** returns scalar metric,
else returns tensor of shape (N, C)

Return type

torch.Tensor

References

https://en.wikipedia.org/wiki/Confusion_matrix

segmentation_models_pytorch.metrics.functional.sensitivity(*tp, fp, fn, tn, reduction=None, class_weights=None, zero_division=1.0*)

Sensitivity, recall, hit rate, or true positive rate (TPR)

Parameters

- **tp** (*torch.LongTensor*) – tensor of shape (N, C), true positive cases
- **fp** (*torch.LongTensor*) – tensor of shape (N, C), false positive cases
- **fn** (*torch.LongTensor*) – tensor of shape (N, C), false negative cases
- **tn** (*torch.LongTensor*) – tensor of shape (N, C), true negative cases
- **reduction** (*Optional[str]*) – Define how to aggregate metric between classes and images:

- **'micro'**
Sum true positive, false positive, false negative and true negative pixels over all images and all classes and then compute score.
- **'macro'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average labels scores. This does not take label imbalance into account.
- **'weighted'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average weighted labels scores.
- **'micro-imagewise'**
Sum true positive, false positive, false negative and true negative pixels for **each image**, then compute score for **each image** and average scores over dataset. All images contribute equally to final score, however takes into account class imbalance for each image.
- **'macro-imagewise'**
Compute score for each image and for each class on that image separately, then compute average score on each image over labels and average image scores over dataset. Does not take into account label imbalance on each image.
- **'weighted-imagewise'**
Compute score for each image and for each class on that image separately, then compute weighted average score on each image over labels and average image scores over dataset.
- **'none' or None**
Same as 'macro-imagewise', but without any reduction.

For 'binary' case 'micro' = 'macro' = 'weighted' and 'micro-imagewise' = 'macro-imagewise' = 'weighted-imagewise'.

Prefixes 'micro', 'macro' and 'weighted' define how the scores for classes will be aggregated, while postfix 'imagewise' defines how scores between the images will be aggregated.

- **class_weights** (*Optional[List[float]*) – list of class weights for metric aggregation, in case of *weighted** reduction is chosen. Defaults to None.
- **zero_division** (*Union[str, float]*) – Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to “warn”, this acts as 0, but warnings are also raised. Defaults to 1.

Returns

if **'reduction'** is not **None** or **'none'** returns scalar metric,
else returns tensor of shape (N, C)

Return type

torch.Tensor

References

https://en.wikipedia.org/wiki/Confusion_matrix

`segmentation_models_pytorch.metrics.functional.specificity(tp, fp, fn, tn, reduction=None, class_weights=None, zero_division=1.0)`

Specificity, selectivity or true negative rate (TNR)

Parameters

- **tp** (*torch.LongTensor*) – tensor of shape (N, C), true positive cases
- **fp** (*torch.LongTensor*) – tensor of shape (N, C), false positive cases
- **fn** (*torch.LongTensor*) – tensor of shape (N, C), false negative cases
- **tn** (*torch.LongTensor*) – tensor of shape (N, C), true negative cases
- **reduction** (*Optional[str]*) – Define how to aggregate metric between classes and images:
 - **'micro'**
Sum true positive, false positive, false negative and true negative pixels over all images and all classes and then compute score.
 - **'macro'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average labels scores. This does not take label imbalance into account.
 - **'weighted'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average weighted labels scores.
 - **'micro-imagewise'**
Sum true positive, false positive, false negative and true negative pixels for **each image**, then compute score for **each image** and average scores over dataset. All images contribute equally to final score, however takes into account class imbalance for each image.
 - **'macro-imagewise'**
Compute score for each image and for each class on that image separately, then compute average score on each image over labels and average image scores over dataset. Does not take into account label imbalance on each image.
 - **'weighted-imagewise'**
Compute score for each image and for each class on that image separately, then compute weighted average score on each image over labels and average image scores over dataset.
 - **'none' or None**
Same as 'macro-imagewise', but without any reduction.

For 'binary' case 'micro' = 'macro' = 'weighted' and 'micro-imagewise' = 'macro-imagewise' = 'weighted-imagewise'.

Prefixes 'micro', 'macro' and 'weighted' define how the scores for classes will be aggregated, while postfix 'imagewise' defines how scores between the images will be aggregated.

- **class_weights** (*Optional[List[float]]*) – list of class weights for metric aggregation, in case of *weighted** reduction is chosen. Defaults to None.
- **zero_division** (*Union[str, float]*) – Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to “warn”, this acts as 0, but warnings are also raised. Defaults to 1.

Returns

if 'reduction' is not None or 'none' returns scalar metric,
else returns tensor of shape (N, C)

Return type

torch.Tensor

References

https://en.wikipedia.org/wiki/Confusion_matrix

segmentation_models_pytorch.metrics.functional.balanced_accuracy(*tp, fp, fn, tn, reduction=None, class_weights=None, zero_division=1.0*)

Balanced accuracy

Parameters

- **tp** (*torch.LongTensor*) – tensor of shape (N, C), true positive cases
- **fp** (*torch.LongTensor*) – tensor of shape (N, C), false positive cases
- **fn** (*torch.LongTensor*) – tensor of shape (N, C), false negative cases
- **tn** (*torch.LongTensor*) – tensor of shape (N, C), true negative cases
- **reduction** (*Optional[str]*) – Define how to aggregate metric between classes and images:
 - **'micro'**
Sum true positive, false positive, false negative and true negative pixels over all images and all classes and then compute score.
 - **'macro'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average labels scores. This does not take label imbalance into account.
 - **'weighted'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average weighted labels scores.
 - **'micro-imagewise'**
Sum true positive, false positive, false negative and true negative pixels for **each image**, then compute score for **each image** and average scores over dataset. All images contribute equally to final score, however takes into account class imbalance for each image.
 - **'macro-imagewise'**
Compute score for each image and for each class on that image separately, then compute average score on each image over labels and average image scores over dataset. Does not take into account label imbalance on each image.

- **'weighted-imagewise'**

Compute score for each image and for each class on that image separately, then compute weighted average score on each image over labels and average image scores over dataset.

- **'none' or None**

Same as 'macro-imagewise', but without any reduction.

For 'binary' case 'micro' = 'macro' = 'weighted' and 'micro-imagewise' = 'macro-imagewise' = 'weighted-imagewise'.

Prefixes 'micro', 'macro' and 'weighted' define how the scores for classes will be aggregated, while postfix 'imagewise' defines how scores between the images will be aggregated.

- **class_weights** (*Optional[List[float]*) – list of class weights for metric aggregation, in case of *weighted** reduction is chosen. Defaults to None.
- **zero_division** (*Union[str, float]*) – Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to “warn”, this acts as 0, but warnings are also raised. Defaults to 1.

Returns

if **'reduction'** is not **None** or **'none'** returns scalar metric,
else returns tensor of shape (N, C)

Return type

torch.Tensor

References

https://en.wikipedia.org/wiki/Confusion_matrix

segmentation_models_pytorch.metrics.functional. **positive_predictive_value**(*tp, fp, fn, tn,*
reduction=None,
class_weights=None,
zero_division=1.0)

Precision or positive predictive value (PPV)

Parameters

- **tp** (*torch.LongTensor*) – tensor of shape (N, C), true positive cases
- **fp** (*torch.LongTensor*) – tensor of shape (N, C), false positive cases
- **fn** (*torch.LongTensor*) – tensor of shape (N, C), false negative cases
- **tn** (*torch.LongTensor*) – tensor of shape (N, C), true negative cases
- **reduction** (*Optional[str]*) – Define how to aggregate metric between classes and images:
 - **'micro'**
Sum true positive, false positive, false negative and true negative pixels over all images and all classes and then compute score.
 - **'macro'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average labels scores. This does not take label imbalance into account.

- **'weighted'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average weighted labels scores.
- **'micro-imagewise'**
Sum true positive, false positive, false negative and true negative pixels for **each image**, then compute score for **each image** and average scores over dataset. All images contribute equally to final score, however takes into account class imbalance for each image.
- **'macro-imagewise'**
Compute score for each image and for each class on that image separately, then compute average score on each image over labels and average image scores over dataset. Does not take into account label imbalance on each image.
- **'weighted-imagewise'**
Compute score for each image and for each class on that image separately, then compute weighted average score on each image over labels and average image scores over dataset.
- **'none' or None**
Same as 'macro-imagewise', but without any reduction.

For 'binary' case 'micro' = 'macro' = 'weighted' and 'micro-imagewise' = 'macro-imagewise' = 'weighted-imagewise'.

Prefixes 'micro', 'macro' and 'weighted' define how the scores for classes will be aggregated, while postfix 'imagewise' defines how scores between the images will be aggregated.

- **class_weights** (*Optional[List[float]]*) – list of class weights for metric aggregation, in case of *weighted** reduction is chosen. Defaults to None.
- **zero_division** (*Union[str, float]*) – Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to “warn”, this acts as 0, but warnings are also raised. Defaults to 1.

Returns

if **'reduction'** is not **None** or **'none'** returns scalar metric,
else returns tensor of shape (N, C)

Return type

torch.Tensor

References

https://en.wikipedia.org/wiki/Confusion_matrix

```
segmentation_models_pytorch.metrics.functional.negative_predictive_value(tp, fp, fn, tn,
                                                                    reduction=None,
                                                                    class_weights=None,
                                                                    zero_division=1.0)
```

Negative predictive value (NPV)

Parameters

- **tp** (*torch.LongTensor*) – tensor of shape (N, C), true positive cases
- **fp** (*torch.LongTensor*) – tensor of shape (N, C), false positive cases

- **fn** (*torch.LongTensor*) – tensor of shape (N, C), false negative cases
- **tn** (*torch.LongTensor*) – tensor of shape (N, C), true negative cases
- **reduction** (*Optional[str]*) – Define how to aggregate metric between classes and images:
 - **'micro'**
Sum true positive, false positive, false negative and true negative pixels over all images and all classes and then compute score.
 - **'macro'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average labels scores. This does not take label imbalance into account.
 - **'weighted'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average weighted labels scores.
 - **'micro-imagewise'**
Sum true positive, false positive, false negative and true negative pixels for **each image**, then compute score for **each image** and average scores over dataset. All images contribute equally to final score, however takes into account class imbalance for each image.
 - **'macro-imagewise'**
Compute score for each image and for each class on that image separately, then compute average score on each image over labels and average image scores over dataset. Does not take into account label imbalance on each image.
 - **'weighted-imagewise'**
Compute score for each image and for each class on that image separately, then compute weighted average score on each image over labels and average image scores over dataset.
 - **'none' or None**
Same as 'macro-imagewise', but without any reduction.

For 'binary' case 'micro' = 'macro' = 'weighted' and 'micro-imagewise' = 'macro-imagewise' = 'weighted-imagewise'.

Prefixes 'micro', 'macro' and 'weighted' define how the scores for classes will be aggregated, while postfix 'imagewise' defines how scores between the images will be aggregated.

- **class_weights** (*Optional[List[float]]*) – list of class weights for metric aggregation, in case of *weighted** reduction is chosen. Defaults to None.
- **zero_division** (*Union[str, float]*) – Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to “warn”, this acts as 0, but warnings are also raised. Defaults to 1.

Returns

if **'reduction'** is not **None** or **'none'** returns scalar metric,
else returns tensor of shape (N, C)

Return type

torch.Tensor

References

https://en.wikipedia.org/wiki/Confusion_matrix

```
segmentation_models_pytorch.metrics.functional.false_negative_rate(tp, fp, fn, tn,
                                                                    reduction=None,
                                                                    class_weights=None,
                                                                    zero_division=1.0)
```

Miss rate or false negative rate (FNR)

Parameters

- **tp** (*torch.LongTensor*) – tensor of shape (N, C), true positive cases
- **fp** (*torch.LongTensor*) – tensor of shape (N, C), false positive cases
- **fn** (*torch.LongTensor*) – tensor of shape (N, C), false negative cases
- **tn** (*torch.LongTensor*) – tensor of shape (N, C), true negative cases
- **reduction** (*Optional[str]*) – Define how to aggregate metric between classes and images:
 - **'micro'**
Sum true positive, false positive, false negative and true negative pixels over all images and all classes and then compute score.
 - **'macro'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average labels scores. This does not take label imbalance into account.
 - **'weighted'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average weighted labels scores.
 - **'micro-imagewise'**
Sum true positive, false positive, false negative and true negative pixels for **each image**, then compute score for **each image** and average scores over dataset. All images contribute equally to final score, however takes into account class imbalance for each image.
 - **'macro-imagewise'**
Compute score for each image and for each class on that image separately, then compute average score on each image over labels and average image scores over dataset. Does not take into account label imbalance on each image.
 - **'weighted-imagewise'**
Compute score for each image and for each class on that image separately, then compute weighted average score on each image over labels and average image scores over dataset.
 - **'none' or None**
Same as 'macro-imagewise', but without any reduction.

For 'binary' case 'micro' = 'macro' = 'weighted' and 'micro-imagewise' = 'macro-imagewise' = 'weighted-imagewise'.

Prefixes 'micro', 'macro' and 'weighted' define how the scores for classes will be aggregated, while postfix 'imagewise' defines how scores between the images will be aggregated.

- **class_weights** (*Optional[List[float]]*) – list of class weights for metric aggregation, in case of *weighted** reduction is chosen. Defaults to None.
- **zero_division** (*Union[str, float]*) – Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to “warn”, this acts as 0, but warnings are also raised. Defaults to 1.

Returns

if 'reduction' is not None or 'none' returns scalar metric,
else returns tensor of shape (N, C)

Return type

torch.Tensor

References

https://en.wikipedia.org/wiki/Confusion_matrix

segmentation_models_pytorch.metrics.functional.false_positive_rate(*tp, fp, fn, tn,*
reduction=None,
class_weights=None,
zero_division=1.0)

Fall-out or false positive rate (FPR)

Parameters

- **tp** (*torch.LongTensor*) – tensor of shape (N, C), true positive cases
- **fp** (*torch.LongTensor*) – tensor of shape (N, C), false positive cases
- **fn** (*torch.LongTensor*) – tensor of shape (N, C), false negative cases
- **tn** (*torch.LongTensor*) – tensor of shape (N, C), true negative cases
- **reduction** (*Optional[str]*) – Define how to aggregate metric between classes and images:
 - **'micro'**
Sum true positive, false positive, false negative and true negative pixels over all images and all classes and then compute score.
 - **'macro'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average labels scores. This does not take label imbalance into account.
 - **'weighted'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average weighted labels scores.
 - **'micro-imagewise'**
Sum true positive, false positive, false negative and true negative pixels for **each image**, then compute score for **each image** and average scores over dataset. All images contribute equally to final score, however takes into account class imbalance for each image.
 - **'macro-imagewise'**
Compute score for each image and for each class on that image separately, then compute

average score on each image over labels and average image scores over dataset. Does not take into account label imbalance on each image.

- **'weighted-imagewise'**
Compute score for each image and for each class on that image separately, then compute weighted average score on each image over labels and average image scores over dataset.
- **'none' or None**
Same as 'macro-imagewise', but without any reduction.

For 'binary' case 'micro' = 'macro' = 'weighted' and 'micro-imagewise' = 'macro-imagewise' = 'weighted-imagewise'.

Prefixes 'micro', 'macro' and 'weighted' define how the scores for classes will be aggregated, while postfix 'imagewise' defines how scores between the images will be aggregated.

- **class_weights** (*Optional[List[float]]*) – list of class weights for metric aggregation, in case of *weighted** reduction is chosen. Defaults to None.
- **zero_division** (*Union[str, float]*) – Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to “warn”, this acts as 0, but warnings are also raised. Defaults to 1.

Returns

if 'reduction' is not None or 'none' returns scalar metric,
else returns tensor of shape (N, C)

Return type

torch.Tensor

References

https://en.wikipedia.org/wiki/Confusion_matrix

```
segmentation_models_pytorch.metrics.functional.false_discovery_rate(tp, fp, fn, tn,
                                                                    reduction=None,
                                                                    class_weights=None,
                                                                    zero_division=1.0)
```

False discovery rate (FDR)

Parameters

- **tp** (*torch.LongTensor*) – tensor of shape (N, C), true positive cases
- **fp** (*torch.LongTensor*) – tensor of shape (N, C), false positive cases
- **fn** (*torch.LongTensor*) – tensor of shape (N, C), false negative cases
- **tn** (*torch.LongTensor*) – tensor of shape (N, C), true negative cases
- **reduction** (*Optional[str]*) – Define how to aggregate metric between classes and images:
 - **'micro'**
Sum true positive, false positive, false negative and true negative pixels over all images and all classes and then compute score.
 - **'macro'**
Sum true positive, false positive, false negative and true negative pixels over all images

for each label, then compute score for each label separately and average labels scores. This does not take label imbalance into account.

- **'weighted'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average weighted labels scores.
- **'micro-imagewise'**
Sum true positive, false positive, false negative and true negative pixels for **each image**, then compute score for **each image** and average scores over dataset. All images contribute equally to final score, however takes into account class imbalance for each image.
- **'macro-imagewise'**
Compute score for each image and for each class on that image separately, then compute average score on each image over labels and average image scores over dataset. Does not take into account label imbalance on each image.
- **'weighted-imagewise'**
Compute score for each image and for each class on that image separately, then compute weighted average score on each image over labels and average image scores over dataset.
- **'none' or None**
Same as 'macro-imagewise', but without any reduction.

For 'binary' case 'micro' = 'macro' = 'weighted' and 'micro-imagewise' = 'macro-imagewise' = 'weighted-imagewise'.

Prefixes 'micro', 'macro' and 'weighted' define how the scores for classes will be aggregated, while postfix 'imagewise' defines how scores between the images will be aggregated.

- **class_weights** (*Optional[List[float]]*) – list of class weights for metric aggregation, in case of *weighted** reduction is chosen. Defaults to None.
- **zero_division** (*Union[str, float]*) – Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to “warn”, this acts as 0, but warnings are also raised. Defaults to 1.

Returns

if **'reduction' is not None or 'none'** returns scalar metric,
else returns tensor of shape (N, C)

Return type

torch.Tensor

References

https://en.wikipedia.org/wiki/Confusion_matrix

```
segmentation_models_pytorch.metrics.functional.false_omission_rate(tp, fp, fn, tn,
                                                                    reduction=None,
                                                                    class_weights=None,
                                                                    zero_division=1.0)
```

False omission rate (FOR)

Parameters

- **tp** (*torch.LongTensor*) – tensor of shape (N, C), true positive cases
- **fp** (*torch.LongTensor*) – tensor of shape (N, C), false positive cases
- **fn** (*torch.LongTensor*) – tensor of shape (N, C), false negative cases
- **tn** (*torch.LongTensor*) – tensor of shape (N, C), true negative cases
- **reduction** (*Optional[str]*) – Define how to aggregate metric between classes and images:
 - **'micro'**
Sum true positive, false positive, false negative and true negative pixels over all images and all classes and then compute score.
 - **'macro'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average labels scores. This does not take label imbalance into account.
 - **'weighted'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average weighted labels scores.
 - **'micro-imagewise'**
Sum true positive, false positive, false negative and true negative pixels for **each image**, then compute score for **each image** and average scores over dataset. All images contribute equally to final score, however takes into account class imbalance for each image.
 - **'macro-imagewise'**
Compute score for each image and for each class on that image separately, then compute average score on each image over labels and average image scores over dataset. Does not take into account label imbalance on each image.
 - **'weighted-imagewise'**
Compute score for each image and for each class on that image separately, then compute weighted average score on each image over labels and average image scores over dataset.
 - **'none' or None**
Same as 'macro-imagewise', but without any reduction.

For 'binary' case 'micro' = 'macro' = 'weighted' and 'micro-imagewise' = 'macro-imagewise' = 'weighted-imagewise'.

Prefixes 'micro', 'macro' and 'weighted' define how the scores for classes will be aggregated, while postfix 'imagewise' defines how scores between the images will be aggregated.

- **class_weights** (*Optional[List[float]]*) – list of class weights for metric aggregation, in case of *weighted** reduction is chosen. Defaults to None.
- **zero_division** (*Union[str, float]*) – Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to “warn”, this acts as 0, but warnings are also raised. Defaults to 1.

Returns

if **'reduction'** is not **None** or **'none'** returns scalar metric,
else returns tensor of shape (N, C)

Return type

torch.Tensor

References

https://en.wikipedia.org/wiki/Confusion_matrix

`segmentation_models_pytorch.metrics.functional.positive_likelihood_ratio(tp, fp, fn, tn,
reduction=None,
class_weights=None,
zero_division=1.0)`

Positive likelihood ratio (LR+)

Parameters

- **tp** (*torch.LongTensor*) – tensor of shape (N, C), true positive cases
- **fp** (*torch.LongTensor*) – tensor of shape (N, C), false positive cases
- **fn** (*torch.LongTensor*) – tensor of shape (N, C), false negative cases
- **tn** (*torch.LongTensor*) – tensor of shape (N, C), true negative cases
- **reduction** (*Optional[str]*) – Define how to aggregate metric between classes and images:
 - **'micro'**
Sum true positive, false positive, false negative and true negative pixels over all images and all classes and then compute score.
 - **'macro'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average labels scores. This does not take label imbalance into account.
 - **'weighted'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average weighted labels scores.
 - **'micro-imagewise'**
Sum true positive, false positive, false negative and true negative pixels for **each image**, then compute score for **each image** and average scores over dataset. All images contribute equally to final score, however takes into account class imbalance for each image.
 - **'macro-imagewise'**
Compute score for each image and for each class on that image separately, then compute average score on each image over labels and average image scores over dataset. Does not take into account label imbalance on each image.
 - **'weighted-imagewise'**
Compute score for each image and for each class on that image separately, then compute weighted average score on each image over labels and average image scores over dataset.
 - **'none' or None**
Same as 'macro-imagewise', but without any reduction.

For 'binary' case 'micro' = 'macro' = 'weighted' and 'micro-imagewise' = 'macro-imagewise' = 'weighted-imagewise'.

Prefixes 'micro', 'macro' and 'weighted' define how the scores for classes will be aggregated, while postfix 'imagewise' defines how scores between the images will be aggregated.

- **class_weights** (*Optional[List[float]]*) – list of class weights for metric aggregation, in case of *weighted** reduction is chosen. Defaults to None.
- **zero_division** (*Union[str, float]*) – Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to “warn”, this acts as 0, but warnings are also raised. Defaults to 1.

Returns

if 'reduction' is not None or 'none' returns scalar metric,
else returns tensor of shape (N, C)

Return type

torch.Tensor

References

https://en.wikipedia.org/wiki/Confusion_matrix

segmentation_models_pytorch.metrics.functional.negative_likelihood_ratio(*tp, fp, fn, tn,*
reduction=None,
class_weights=None,
zero_division=1.0)

Negative likelihood ratio (LR-)

Parameters

- **tp** (*torch.LongTensor*) – tensor of shape (N, C), true positive cases
- **fp** (*torch.LongTensor*) – tensor of shape (N, C), false positive cases
- **fn** (*torch.LongTensor*) – tensor of shape (N, C), false negative cases
- **tn** (*torch.LongTensor*) – tensor of shape (N, C), true negative cases
- **reduction** (*Optional[str]*) – Define how to aggregate metric between classes and images:
 - **'micro'**
Sum true positive, false positive, false negative and true negative pixels over all images and all classes and then compute score.
 - **'macro'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average labels scores. This does not take label imbalance into account.
 - **'weighted'**
Sum true positive, false positive, false negative and true negative pixels over all images for each label, then compute score for each label separately and average weighted labels scores.
 - **'micro-imagewise'**
Sum true positive, false positive, false negative and true negative pixels for **each image**, then compute score for **each image** and average scores over dataset. All images contribute equally to final score, however takes into account class imbalance for each image.

- **'macro-imagewise'**
Compute score for each image and for each class on that image separately, then compute average score on each image over labels and average image scores over dataset. Does not take into account label imbalance on each image.
- **'weighted-imagewise'**
Compute score for each image and for each class on that image separately, then compute weighted average score on each image over labels and average image scores over dataset.
- **'none' or None**
Same as 'macro-imagewise', but without any reduction.

For 'binary' case 'micro' = 'macro' = 'weighted' and 'micro-imagewise' = 'macro-imagewise' = 'weighted-imagewise'.

Prefixes 'micro', 'macro' and 'weighted' define how the scores for classes will be aggregated, while postfix 'imagewise' defines how scores between the images will be aggregated.

- **class_weights** (*Optional[List[float]]*) – list of class weights for metric aggregation, in case of *weighted** reduction is chosen. Defaults to None.
- **zero_division** (*Union[str, float]*) – Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to “warn”, this acts as 0, but warnings are also raised. Defaults to 1.

Returns

if **'reduction'** is not **None** or **'none'** returns scalar metric,
else returns tensor of shape (N, C)

Return type

torch.Tensor

References

https://en.wikipedia.org/wiki/Confusion_matrix

8.1 1. Models architecture

All segmentation models in SMP (this library short name) are made of:

- encoder (feature extractor, a.k.a backbone)
- decoder (features fusion block to create segmentation *mask*)
- segmentation head (final head to reduce number of channels from decoder and upsample mask to preserve input-output spatial resolution identity)
- classification head (optional head which build on top of deepest encoder features)

8.2 2. Creating your own encoder

Encoder is a “classification model” which extract features from image and pass it to decoder. Each encoder should have following attributes and methods and be inherited from `segmentation_models_pytorch.encoders._base.EncoderMixin`

```
class MyEncoder(torch.nn.Module, EncoderMixin):

    def __init__(self, **kwargs):
        super().__init__()

        # A number of channels for each encoder feature tensor, list of integers
        self._out_channels: List[int] = [3, 16, 64, 128, 256, 512]

        # A number of stages in decoder (in other words number of downsampling_
↳ operations), integer
        # use in in forward pass to reduce number of returning features
        self._depth: int = 5

        # Default number of input channels in first Conv2d layer for encoder (usually 3)
        self._in_channels: int = 3

        # Define encoder modules below
        ...

    def forward(self, x: torch.Tensor) -> List[torch.Tensor]:
        """Produce list of features of different spatial resolutions, each feature is a_
↳ 4D torch.tensor of
```

(continues on next page)

(continued from previous page)

```

    shape NCHW (features should be sorted in descending order according to spatial_
    ↪resolution, starting
    with resolution same as input `x` tensor).

    Input: `x` with shape (1, 3, 64, 64)
    Output: [f0, f1, f2, f3, f4, f5] - features with corresponding shapes
            [(1, 3, 64, 64), (1, 64, 32, 32), (1, 128, 16, 16), (1, 256, 8, 8),
            (1, 512, 4, 4), (1, 1024, 2, 2)] (C - dim may differ)

    also should support number of features according to specified depth, e.g. if_
    ↪depth = 5,
    number of feature tensors = 6 (one with same resolution as input and 5_
    ↪downsampled),
    depth = 3 -> number of feature tensors = 4 (one with same resolution as input_
    ↪and 3 downsampled).
    """

    return [feat1, feat2, feat3, feat4, feat5, feat6]

```

When you write your own Encoder class register its build parameters

```

smp.encoders.encoders["my_awesome_encoder"] = {
    "encoder": MyEncoder, # encoder class here
    "pretrained_settings": {
        "imagenet": {
            "mean": [0.485, 0.456, 0.406],
            "std": [0.229, 0.224, 0.225],
            "url": "https://some-url.com/my-model-weights",
            "input_space": "RGB",
            "input_range": [0, 1],
        },
    },
    "params": {
        # init params for encoder if any
    },
},

```

Now you can use your encoder

```
model = smp.Unet(encoder_name="my_awesome_encoder")
```

For better understanding see more examples of encoder in smp.encoders module.

Note: If it works fine, don't forget to contribute your work and make a PR to SMP

8.3 3. Aux classification output

All models support `aux_params` parameter, which is default set to `None`. If `aux_params = None` than classification auxiliary output is not created, else model produce not only mask, but also `label` output with shape (N, C) .

Classification head consist of following layers:

1. GlobalPooling
2. Dropout (optional)
3. Linear
4. Activation (optional)

Example:

```
aux_params=dict(
    pooling='avg',           # one of 'avg', 'max'
    dropout=0.5,             # dropout ratio, default is None
    activation='sigmoid',    # activation function, default is None
    classes=4,               # define number of output labels
)

model = smp.Unet('resnet34', classes=4, aux_params=aux_params)
mask, label = model(x)

mask.shape, label.shape
# (N, 4, H, W), (N, 4)
```


INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

S

`segmentation_models_pytorch.losses.constants,`

[35](#)

`segmentation_models_pytorch.metrics.functional,`

[41](#)

A

`accuracy()` (in module `segmentation_models_pytorch.metrics.functional`), 47

B

`balanced_accuracy()` (in module `segmentation_models_pytorch.metrics.functional`), 53

`BINARY_MODE` (in module `segmentation_models_pytorch.losses.constants`), 35

D

`DeepLabV3` (class in `segmentation_models_pytorch`), 13

`DeepLabV3Plus` (class in `segmentation_models_pytorch`), 14

`DiceLoss` (class in `segmentation_models_pytorch.losses`), 36

F

`f1_score()` (in module `segmentation_models_pytorch.metrics.functional`), 44

`false_discovery_rate()` (in module `segmentation_models_pytorch.metrics.functional`), 59

`false_negative_rate()` (in module `segmentation_models_pytorch.metrics.functional`), 57

`false_omission_rate()` (in module `segmentation_models_pytorch.metrics.functional`), 60

`false_positive_rate()` (in module `segmentation_models_pytorch.metrics.functional`), 58

`fbeta_score()` (in module `segmentation_models_pytorch.metrics.functional`), 43

`FocalLoss` (class in `segmentation_models_pytorch.losses`), 37

`forward()` (in module `segmentation_models_pytorch.losses.MCCLoss` method), 39

`FPN` (class in `segmentation_models_pytorch`), 9

G

`get_stats()` (in module `segmentation_models_pytorch.metrics.functional`), 42

I

`iou_score()` (in module `segmentation_models_pytorch.metrics.functional`), 45

J

`JaccardLoss` (class in `segmentation_models_pytorch.losses`), 35

L

`Linknet` (class in `segmentation_models_pytorch`), 8

`LovaszLoss` (class in `segmentation_models_pytorch.losses`), 38

M

`MAnet` (class in `segmentation_models_pytorch`), 7

`MCCLoss` (class in `segmentation_models_pytorch.losses`), 39

module

`segmentation_models_pytorch.losses.constants`, 35

`segmentation_models_pytorch.metrics.functional`, 41

`MULTICLASS_MODE` (in module `segmentation_models_pytorch.losses.constants`), 35

`MULTILABEL_MODE` (in module `segmentation_models_pytorch.losses.constants`), 35

N

`negative_likelihood_ratio()` (in module `segmentation_models_pytorch.metrics.functional`), 63

`negative_predictive_value()` (in module `segmentation_models_pytorch.metrics.functional`), 55

P

`PAN` (class in `segmentation_models_pytorch`), 12

`positive_likelihood_ratio()` (in module `segmentation_models_pytorch.metrics.functional`), 62

`positive_predictive_value()` (in module `segmentation_models_pytorch.metrics.functional`), 54

`precision()` (in module `segmentation_models_pytorch.metrics.functional`), 48

`PSPNet` (class in `segmentation_models_pytorch`), 11

R

`recall()` (in module `segmentation_models_pytorch.metrics.functional`), 49

S

`segmentation_models_pytorch.losses.constants` module, 35

`segmentation_models_pytorch.metrics.functional` module, 41

`sensitivity()` (in module `segmentation_models_pytorch.metrics.functional`), 50

`SoftBCEWithLogitsLoss` (class in `segmentation_models_pytorch.losses`), 38

`SoftCrossEntropyLoss` (class in `segmentation_models_pytorch.losses`), 39

`specificity()` (in module `segmentation_models_pytorch.metrics.functional`), 52

T

`TverskyLoss` (class in `segmentation_models_pytorch.losses`), 36

U

`Unet` (class in `segmentation_models_pytorch`), 5

`UnetPlusPlus` (class in `segmentation_models_pytorch`), 6