pywick Documentation

Release 0.5.3

Achaiah

Jan 20, 2020

Contents

CHAPTER 1

About

Pywick is a high-level Pytorch training framework that aims to get you up and running quickly with state of the art neural networks. Does the world need another Pytorch framework? Probably not. But we started this project when no good frameworks were available and it just kept growing. So here we are.

Pywick tries to stay on the bleeding edge of research into neural networks. If you just wish to run a vanilla CNN, this is probably going to be overkill. However, if you want to get lost in the world of neural networks, fine-tuning and hyperparameter optimization for months on end then this is probably the right place for you :)

CHAPTER 2

Guide

We started this project because of the work we were doing on image classification and segmentation so this is where most of the updates are happening. However, along the way we've added many powerful tools for fine-tuning your results, from specifying custom *Constraints* on your network layers, to awesomely flexible *Dataloaders* for your data needs, to a variety of standard and not-so-standard *loss functions* to *Optimizers*, *Regularizers* and *Transforms*. You'll find a pretty decent description of each one of them in the navigation pane.

And of course, if you have any questions, feel free to drop by our github page

2.1 Pywick

2.1.1 High-Level Training framework for Pytorch

Pywick is a high-level Pytorch training framework that aims to get you up and running quickly with state of the art neural networks. *Does the world need another Pytorch framework?* Probably not. But we started this project when no good frameworks were available and it just kept growing. So here we are.

Pywick tries to stay on the bleeding edge of research into neural networks. If you just wish to run a vanilla CNN, this is probably going to be overkill. However, if you want to get lost in the world of neural networks, fine-tuning and hyperparameter optimization for months on end then this is probably the right place for you :)

Among other things Pywick includes:

- State of the art normalization, activation, loss functions and optimizers not included in the standard Pytorch library.
- A high-level module for training with callbacks, constraints, metrics, conditions and regularizers.
- Dozens of popular object classification and semantic segmentation models.
- Comprehensive data loading, augmentation, transforms, and sampling capability.
- Utility tensor functions.
- Useful meters.

- Basic GridSearch (exhaustive and random).

2.1.2 Docs

Hey, check this out, we now have docs! They're still a work in progress though so apologies for anything that's broken.

2.1.3 What's New (highlights)

- **Jan. 20, 2020**
  - New release: 0.5.6 (minor fix from 0.5.5 for pypi)
  - Mish activation function (SoTA)
  - rwightman's models of pretrained/ported variants for classification (44 total)
    - efficientnet Tensorflow port b0-b8, with and without AP, el/em/es, cc
    - mixnet L/M/S
    - mobilenetv3
    - mnasnet
    - spnasnet
  - Additional loss functions
- **Aug. 1, 2019**
  - New segmentation NNs: BiSeNet, DANet, DenseASPP, DUNet, OCNet, PSANet
  - New Loss Functions: Focal Tversky Loss, OHEM CrossEntropy Loss, various combination losses
  - Major restructuring and standardization of NN models and loading functionality
  - General bug fixes and code improvements

2.1.4 Install

Pywick requires **pytorch >= 1.0**

```
pip install pywick
```

or specific version from git:

```
pip install git+https://github.com/achaiah/pywick.git@v0.5.6
```

2.1.5 ModuleTrainer

The `ModuleTrainer` class provides a high-level training interface which abstracts away the training loop while providing callbacks, constraints, initializers, regularizers, and more.

Example:

```
from pywick.modules import ModuleTrainer
from pywick.initializers import XavierUniform
from pywick.metrics import CategoricalAccuracySingleInput
import torch.nn as nn
import torch.functional as F

# Define your model EXACTLY as normal
class Network(nn.Module):
    def __init__(self):
        super(Network, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3)
        self.fc1 = nn.Linear(1600, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = x.view(-1, 1600)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x)

model = Network()
trainer = ModuleTrainer(model)   # optionally supply cuda_devices as a parameter

initializers = [XavierUniform(bias=False, module_filter='fc*')]

# initialize metrics with top1 and top5
metrics = [CategoricalAccuracySingleInput(top_k=1),␣
↪CategoricalAccuracySingleInput(top_k=5)]

trainer.compile(loss='cross_entropy',
                # callbacks=callbacks,          # define your callbacks here (e.g.␣
↪model saver, LR scheduler)
                # regularizers=regularizers,    # define regularizers
                # constraints=constraints,      # define constraints
                optimizer='sgd',
                initializers=initializers,
                metrics=metrics)

trainer.fit_loader(train_dataset_loader,
            val_loader=val_dataset_loader,
            num_epoch=20,
            verbose=1)
```

You also have access to the standard evaluation and prediction functions:

```
loss = trainer.evaluate(x_train, y_train)
y_pred = trainer.predict(x_train)
```

PyWick provides a wide range of callbacks, generally mimicking the interface found in `Keras`:

- `CSVLogger` - Logs epoch-level metrics to a CSV file
- `CyclicLRScheduler` - Cycles through min-max learning rate
- `EarlyStopping` - Provides ability to stop training early based on supplied criteria

- `History` - Keeps history of metrics etc. during the learning process
- `LambdaCallback` - Allows you to implement your own callbacks on the fly
- `LRScheduler` - Simple learning rate scheduler based on function or supplied schedule
- `ModelCheckpoint` - Comprehensive model saver
- `ReduceLROnPlateau` - Reduces learning rate (LR) when a plateau has been reached
- `SimpleModelCheckpoint` - Simple model saver
- Additionally, a `TensorboardLogger` is incredibly easy to implement via the TensorboardX (now part of pytorch 1.1 release!)

```
from pywick.callbacks import EarlyStopping

callbacks = [EarlyStopping(monitor='val_loss', patience=5)]
trainer.set_callbacks(callbacks)
```

PyWick also provides regularizers:

- `L1Regularizer`
- `L2Regularizer`
- `L1L2Regularizer`

and constraints:

- `UnitNorm`
- `MaxNorm`
- `NonNeg`

Both regularizers and constraints can be selectively applied on layers using regular expressions and the `module_filter` argument. Constraints can be explicit (hard) constraints applied at an arbitrary batch or epoch frequency, or they can be implicit (soft) constraints similar to regularizers where the the constraint deviation is added as a penalty to the total model loss.

```
from pywick.constraints import MaxNorm, NonNeg
from pywick.regularizers import L1Regularizer

# hard constraint applied every 5 batches
hard_constraint = MaxNorm(value=2., frequency=5, unit='batch', module_filter='*fc*')
# implicit constraint added as a penalty term to model loss
soft_constraint = NonNeg(lagrangian=True, scale=1e-3, module_filter='*fc*')
constraints = [hard_constraint, soft_constraint]
trainer.set_constraints(constraints)

regularizers = [L1Regularizer(scale=1e-4, module_filter='*conv*')]
trainer.set_regularizers(regularizers)
```

You can also fit directly on a `torch.utils.data.DataLoader` and can have a validation set as well :

```
from pywick import TensorDataset
from torch.utils.data import DataLoader

train_dataset = TensorDataset(x_train, y_train)
train_loader = DataLoader(train_dataset, batch_size=32)

val_dataset = TensorDataset(x_val, y_val)
```

(continues on next page)

(continued from previous page)

```
val_loader = DataLoader(val_dataset, batch_size=32)

trainer.fit_loader(loader, val_loader=val_loader, num_epoch=100)
```

2.1.6 Extensive Library of Image Classification Models (most are pretrained!)

- All standard models from Pytorch:
  - Densenet
  - Inception v3
  - MobileNet v2
  - ResNet
  - ShuffleNet v2
  - SqueezeNet
  - VGG
- BatchNorm Inception
- Dual Path Networks
- EfficientNet variants b0-b8
- FBResnet
- FBNet-C
- Inception v4
- InceptionResnet v2
- Mixnet L/M/S
- MnasNet
- MobileNet V3
- NasNet and NasNet Mobile
- PNASNet
- Polynet
- Pyramid Resnet
- Resnet + Swish
- ResNext
- SE Net
- SE Inception
- Single-Pass NAS Net
- Wide Resnet
- XCeption

2.1.7 Image Segmentation Models

- **BiSeNet** (Bilateral Segmentation Network for Real-time Semantic Segmentation)
- **DANet** (Dual Attention Network for Scene Segmentation)
- **Deeplab v2** (DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs)
- **Deeplab v3** (Rethinking Atrous Convolution for Semantic Image Segmentation)
- **DenseASPP** (DenseASPP for Semantic Segmentation in Street Scenes)
- **DRNNet** (Dilated Residual Networks)
- **DUC, HDC** (understanding convolution for semantic segmentation)
- **DUNet** (Decoders Matter for Semantic Segmentation)
- **ENet** (ENet: A Deep Neural Network Architecture for Real-Time Semantic Segmentation)
- **Vanilla FCN:** FCN32, FCN16, FCN8, in the versions of VGG, ResNet and OptDenseNet respectively (Fully convolutional networks for semantic segmentation)
- **FRRN** (Full Resolution Residual Networks for Semantic Segmentation in Street Scenes)
- **FusionNet** (FusionNet in Tensorflow by Hyungjoo Andrew Cho)
- **GCN** (Large Kernel Matters)
- **LinkNet** (Link-Net)
- **OCNet** (Object Context Network for Scene Parsing)
- **PSPNet** (Pyramid scene parsing network)
- **RefineNet** (RefineNet)
- **SegNet** (Segnet: A deep convolutional encoder-decoder architecture for image segmentation)
- **Tiramisu** (The One Hundred Layers Tiramisu: Fully Convolutional DenseNets for Semantic Segmentation)
- **U-Net** (U-net: Convolutional networks for biomedical image segmentation)
- Additional variations of many of the above

To load one of these models:

Read the docs for useful details! Then dive in:

```
# use the `get_model` utility
from pywick.models.model_utils import get_model, ModelType

model = get_model(model_type=ModelType.CLASSIFICATION, model_name='resnet18', num_
↪classes=1000, pretrained=True)
```

For a complete list of models (including many experimental ones) you can call the `get_supported_models` method e.g. `pywick.models.model_utils.get_supported_models(ModelType.SEGMENTATION)`

2.1.8 Data Augmentation and Datasets

The PyWick package provides wide variety of good data augmentation and transformation tools which can be applied during data loading. The package also provides the flexible `TensorDataset`, `FolderDataset` and `MultiFolderDataset` classes to handle most dataset needs.

Torch Transforms

These transforms work directly on torch tensors

- `AddChannel`
- `ChannelsFirst`
- `ChannelsLast`
- `Compose`
- `ExpandAxis`
- `Pad`
- `PadNumpy`
- `RandomChoiceCompose`
- `RandomCrop`
- `RandomFlip`
- `RandomOrder`
- `RangeNormalize`
- `Slice2D`
- `SpecialCrop`
- `StdNormalize`
- `ToFile`
- `ToNumpyType`
- `ToTensor`
- `Transpose`
- `TypeCast`

Additionally, we provide image-specific manipulations directly on tensors:

- `Brightness`
- `Contrast`
- `Gamma`
- `Grayscale`
- `RandomBrightness`
- `RandomChoiceBrightness`

- RandomChoiceContrast
- RandomChoiceGamma
- RandomChoiceSaturation
- RandomContrast
- RandomGamma
- RandomGrayscale
- RandomSaturation
- Saturation

Affine Transforms (perform affine or affine-like transforms on torch tensors)

- RandomAffine
- RandomChoiceRotate
- RandomChoiceShear
- RandomChoiceTranslate
- RandomChoiceZoom
- RandomRotate
- RandomShear
- RandomSquareZoom
- RandomTranslate
- RandomZoom
- Rotate
- Shear
- Translate
- Zoom

We also provide a class for stringing multiple affine transformations together so that only one interpolation takes place:

- Affine
- AffineCompose

Blur and Scramble transforms (for tensors)

- Blur
- RandomChoiceBlur
- RandomChoiceScramble
- Scramble

Datasets and Sampling

We provide the following datasets which provide general structure and iterators for sampling from and using transforms on in-memory or out-of-memory data. In particular, the FolderDataset has been designed to fit most of your dataset needs. It has extensive options for data filtering and manipulation. It supports loading images for classification, segmentation and even arbitrary source/target mapping. Take a good look at its documentation for more info.

- `ClonedDataset`
- `CSVDataset`
- `FolderDataset`
- `MultiFolderDataset`
- `TensorDataset`
- `tnt.BatchDataset`
- `tnt.ConcatDataset`
- `tnt.ListDataset`
- `tnt.MultiPartitionDataset`
- `tnt.ResampleDataset`
- `tnt.ShuffleDataset`
- `tnt.TensorDataset`
- `tnt.TransformDataset`

Imbalanced Datasets

In many scenarios it is important to ensure that your traing set is properly balanced, however, it may not be practical in real life to obtain such a perfect dataset. In these cases you can use the `ImbalancedDatasetSampler` as a drop-in replacement for the basic sampler provided by the DataLoader. More information can be found here

```
from pywick.samplers import ImbalancedDatasetSampler

train_loader = torch.utils.data.DataLoader(train_dataset,
    sampler=ImbalancedDatasetSampler(train_dataset),
    batch_size=args.batch_size, **kwargs)
```

2.1.9 Utility Functions

PyWick provides a few utility functions not commonly found:

Tensor Functions

- `th_iterproduct` (mimics itertools.product)
- `th_gather_nd` (N-dimensional version of torch.gather)
- `th_random_choice` (mimics np.random.choice)
- `th_pearsonr` (mimics scipy.stats.pearsonr)
- `th_corrcoef` (mimics np.corrcoef)

- `th_affine2d` and `th_affine3d` (affine transforms on torch.Tensors)

2.1.10 Acknowledgements and References

We stand on the shoulders of (github?) giants and couldn't have done this without the rich github ecosystem and community. This framework is based in part on the excellent Torchsample framework originally published by @ncullen93. Additionally, many models have been gently borrowed/modified from @Cadene pretrained models repo as well as @Tramac segmentation repo.

Thank you to the following people and the projects they maintain:

- @ncullen93
- @cadene
- @deallynomore
- @recastrodiaz
- @zijundeng
- @Tramac
- And many others! (attributions listed in the codebase as they occur)

Thank you to the following projects from which we gently borrowed code and models

- PyTorchNet
- pretrained-models.pytorch
- DeepLab_pytorch
- Pytorch for Semantic Segmentation
- Binseg Pytorch
- awesome-semantic-segmentation-pytorch
- And many others! (attributions listed in the codebase as they occur)

| *Thangs are broken matey! Arrr!!!* | |————————————| | We're working on this project as time permits so you might discover bugs here and there. Feel free to report them, or better yet, to submit a pull request! |

2.2 Classification

In a short while we will publish a walk-through that will go into detail on how to do classification with Pywick. In the meantime, if you feel adventurous feel free to look at our README.

2.3 Segmentation

In a short while we will publish a walk-through that will go into detail on how to do segmentation with Pywick. In the meantime, if you feel adventurous feel free to look at our README.

2.4 Callbacks

Callbacks are the primary mechanism by which one can embed event hooks into the training process. Many useful callbacks are provided out of the box but in all likelihood you will want to implement your own to execute actions based on training events. To do so, simply extend the pywick.callbacks.Callback class and overwrite functions that you are interested in acting upon.

2.4.1 CSVLogger

class pywick.callbacks.CSVLogger.**CSVLogger**(*file*, *separator=','*, *append=False*)

Bases: *pywick.callbacks.Callback.Callback*

Logs epoch-level metrics to a CSV file

Parameters

- **file** – (string) path to csv file
- **separator** – (string) delimiter for file
- **append** – (bool) whether to append result to existing file or make new file

on_epoch_end(*epoch*, *logs=None*)

on_train_begin(*logs=None*)

on_train_end(*logs=None*)

2.4.2 Callback

class pywick.callbacks.Callback.**Callback**

Bases: object

Abstract base class used to build new callbacks. Extend this class to build your own callbacks and overwrite functions that you want to monitor. Functions will be called automatically from the trainer once per relevant training event (e.g. at the beginning of epoch, end of epoch, beginning of batch, end of batch etc.)

on_batch_begin(*batch*, *logs=None*)

on_batch_end(*batch*, *logs=None*)

on_epoch_begin(*epoch*, *logs=None*)

on_epoch_end(*epoch*, *logs=None*)

on_train_begin(*logs=None*)

on_train_end(*logs=None*)

set_params(*params*)

set_trainer(*trainer*)

2.4.3 CyclicLRScheduler

class pywick.callbacks.CyclicLRScheduler.**CyclicLRScheduler**(*optimizer*, *base_lr=0.001*, *max_lr=0.006*, *step_size=2000*, *mode='triangular'*, *gamma=1.0*, *scale_fn=None*, *scale_mode='cycle'*, *verbose=True*)

Bases: *pywick.callbacks.Callback.Callback*

Sets the learning rate of each parameter group according to cyclical learning rate policy (CLR). The policy cycles the learning rate between two boundaries with a constant frequency, as detailed in the paper Cyclical Learning Rates for Training Neural Networks. The distance between the two boundaries can be scaled on a per-iteration or per-cycle basis.

Cyclical learning rate policy changes the learning rate after every batch. *batch_step* should be called after a batch has been used for training. To resume training, save *last_batch_iteration* and use it to instantiate *CycleLR*.

This class has three built-in policies, as put forth in the paper: "triangular":

A basic triangular cycle w/ no amplitude scaling.

"triangular2": A basic triangular cycle that scales initial amplitude by half each cycle.

"exp_range": A cycle that scales initial amplitude by gamma**(cycle iterations) at each cycle iteration.

This implementation was adapted from the github repo: bckenstler/CLR

Parameters

- **optimizer** – (Optimizer): Wrapped optimizer.
- **base_lr** – (float or list): Initial learning rate which is the lower boundary in the cycle for eachparam groups. Default: 0.001
- **max_lr** – (float or list): Upper boundaries in the cycle for each parameter group. Functionally, it defines the cycle amplitude (max_lr - base_lr). The lr at any cycle is the sum of base_lr and some scaling of the amplitude; therefore max_lr may not actually be reached depending on scaling function. Default: 0.006
- **step_size** – (int): Number of training iterations per half cycle. Authors suggest setting step_size 2-8 x training iterations in epoch. Default: 2000
- **mode** – (str): One of {triangular, triangular2, exp_range}. Values correspond to policies detailed above. If scale_fn is not None, this argument is ignored. Default: 'triangular'
- **gamma** – (float): Constant in 'exp_range' scaling function: gamma**(cycle iterations) Default: 1.0
- **scale_fn** – (function): Custom scaling policy defined by a single argument lambda function, where 0 <= scale_fn(x) <= 1 for all x >= 0. mode paramater is ignored Default: None
- **scale_mode** – (str): {'cycle', 'iterations'}. Defines whether scale_fn is evaluated on cycle number or cycle iterations (training iterations since start of cycle). Default: 'cycle'
- **verbose** – (bool): Whether to produce some output during initialization Default: True

Example:

```
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
>>> scheduler = torch.optim.CyclicLR(optimizer)
>>> data_loader = torch.utils.data.DataLoader(...)
>>> for epoch in range(10):
>>>     for batch in data_loader:
>>>         scheduler.batch_step()
>>>         train_batch(...)
```

get_lr()

on_batch_end(*batch*, *logs=None*)

pywick.callbacks.CyclicLRScheduler.**widegap_scale_fn**(*x*)

2.4.4 EarlyStopping

class pywick.callbacks.EarlyStopping.**EarlyStopping**(*monitor='val_loss'*, *min_delta=0*, *patience=5*)

Bases: *pywick.callbacks.Callback.Callback*

Early Stopping to terminate training early under certain conditions

EarlyStopping callback to exit the training loop if training or validation loss does not improve by a certain amount for a certain number of epochs

Parameters

- **monitor** – (string in {'val_loss', 'loss'}): whether to monitor train or val loss
- **min_delta** – (float): minimum change in monitored value to qualify as improvement. This number should be positive.
- **patience** – (int): number of epochs to wait for improvment before terminating. the counter be reset after each improvment

on_epoch_end(*epoch*, *logs=None*)

on_train_begin(*logs=None*)

on_train_end(*logs*)

2.4.5 ExperimentLogger

class pywick.callbacks.ExperimentLogger.**ExperimentLogger**(*directory*, *filename='Experiment_Logger.csv'*, *save_prefix='Model_'*, *separator=','*, *append=True*)

Bases: *pywick.callbacks.Callback.Callback*

Generic logger callback for dumping experiment data. Can be extended for more utility.

on_train_begin(*logs=None*)

on_train_end(*logs=None*)

2.4.6 History

class pywick.callbacks.History.**History**(*trainer*)
Bases: *pywick.callbacks.Callback.Callback*

Callback that records events into a *History* object.

This callback is automatically applied to every SuperModule.

on_batch_end(*batch*, *logs=None*)

on_epoch_begin(*epoch*, *logs=None*)

on_epoch_end(*epoch*, *logs=None*)

on_train_begin(*logs=None*)

2.4.7 LRScheduler

class pywick.callbacks.LRScheduler.**LRScheduler**(*schedule*)
Bases: *pywick.callbacks.Callback.Callback*

Schedule the learning rate according to some function of the current epoch index, current learning rate, and current train/val loss.

Parameters **schedule** – (callable): should return a number of learning rates equal to the number of optimizer.param_groups. It should take the epoch index and **kwargs (or logs) as argument. **kwargs (or logs) will return the epoch logs such as mean training and validation loss from the epoch

on_epoch_begin(*epoch*, *logs=None*)
WARNING: Do NOT use this callback with self-adjusting learners like Yellowfin

schedule_from_dict(*epoch*, *logs=None*)

2.4.8 LambdaCallback

class pywick.callbacks.LambdaCallback.**LambdaCallback**(*on_epoch_begin=None*, *on_epoch_end=None*, *on_batch_begin=None*, *on_batch_end=None*, *on_train_begin=None*, *on_train_end=None*, ***kwargs*)
Bases: *pywick.callbacks.Callback.Callback*

Callback for creating simple, custom callbacks on-the-fly.

2.4.9 ModelCheckpoint

class pywick.callbacks.ModelCheckpoint.**ModelCheckpoint**(*run_id*, *monitored_log_key*, *save_dir*, *addl_k_v={}*, *epoch_log_keys=[]*, *save_interval=5*, *save_best_only=False*, *max_saves=5*, *custom_func=None*, *do_minimize=True*, *verbose=False*)

Bases: *pywick.callbacks.Callback.Callback*

Model Checkpoint to save model weights during training. 'Best' is determined by minimizing the value found under monitored_log_key in the logs Saved checkpoints contain these keys by default:

'run_id' 'epoch' 'loss_type' 'loss_val' 'best_epoch' - plus any additional key/value pairs produced by custom_func

Additionally saves a .json file with statistics about the run such as: 'run_id' 'num_epochs' 'best_epoch' 'best_loss_or_gain' 'metric_name' - plus any additional key/value pairs produced by custom_func

Parameters

- **run_id** – (string): Uniquely identifies the run
- **monitored_log_key** – (string): Name of the key in the logs that will contain the value we want to minimize (and thus that will dictate whether the model is 'best')
- **save_dir** – (string): Path indicating where to save the checkpoint
- **addl_k_v** – (dict): dictionary of additional key/value pairs to save with the model. Typically these include some initialization parameters, name of the model etc. (e.g. from the initialization dictionary 'opt'), as well as other useful params (e.g. mean, std, proc_type: gpu/cpu etc)
- **epoch_log_keys** – (list): list of keys to save from the epoch log dictionary (Note: the logs dictionary is automatically provided by the learning framework)
- **save_interval** – (int): How often to save the model (if none then will default to every 5 iterations)
- **save_best_only** – (bool): Whether only to save the best result (and overwrite all previous) Default: False
- **max_saves** – (integer > 0 or -1): the max number of models to save. Older model checkpoints will be overwritten if necessary. Set equal to -1 to have no limit. Default: 5
- **custom_func** – func(k_v_dict, logs, out_dict, monitored_log_key, is_end_training): Custom function for performing any additional logic (to add values to the model). The function will be passed the addl_k_v dictionary, the event logs dictionary, an output dictionary to process, the monitored_log_key and a bool indicating whether the training is finished. The function is expected to modify the output dictionary in order to preserve values across epochs. The function will be called at the end of each epoch and at the end of the training (with is_end_traing = True)
- **do_minimize** – (bool): whether to minimize or maximize the 'monitored_log_key' value
- **verbose** – (bool): verbosity of the console output Default: False

on_epoch_end(*epoch*, *logs=None*)

on_train_end(*logs=None*)

pywick.callbacks.ModelCheckpoint.**generate_checkpoint_name**(*run_id*, *kv_dict*, *epoch*, *is_best*)

pywick.callbacks.ModelCheckpoint.**generate_statsfile_name**(*run_id*, *save_dir*)

pywick.callbacks.ModelCheckpoint.**save_checkpoint**(*state*, *is_best=False*, *save_path='.'*, *filename=None*)

Saves checkpoint to file.

Parameters

- **state** – (dict): the dictionary to save. Can have other values besides just model weights.
- **is_best** – (bool): whether this is the best result we've seen thus far
- **save_path** – (string): local dir to save to
- **filename** – (string): name of the file to save under *save_path*

Returns

2.4.10 ReduceLROnPlateau

class pywick.callbacks.ReduceLROnPlateau.**ReduceLROnPlateau**(*monitor='val_loss'*, *factor=0.1*, *patience=10*, *epsilon=0*, *cooldown=0*, *min_lr=0*, *verbose=0*)

Bases: *pywick.callbacks.Callback.Callback*

Reduce the learning rate if the train or validation loss plateaus

Parameters

- **monitor** – (string in {'loss', 'val_loss'}): which metric to monitor
- **factor** – (float): factor to decrease learning rate by
- **patience** – (int): number of epochs to wait for loss improvement before reducing lr
- **epsilon** – (float): how much improvement must be made to reset patience
- **cooldown** – (int): number of epochs to cooldown after a lr reduction
- **min_lr** – (float): minimum value to ever let the learning rate decrease to
- **verbose** – (int): whether to print reduction to console

on_epoch_end(*epoch*, *logs=None*)

on_train_begin(*logs=None*)

2.4.11 SimpleModelCheckpoint

class pywick.callbacks.SimpleModelCheckpoint.**SimpleModelCheckpoint**(*directory*, *filename='ckpt.pth.tar'*, *monitor='val_loss'*, *save_best_only=False*, *save_weights_only=True*, *max_save=-1*, *verbose=0*)

Bases: *pywick.callbacks.Callback.Callback*

Simple Checkpoint to save model weights during training. This class is mostly superceded by ModelCheckpoint which provides flexible saving functionality.

Parameters

- **file** – (string): file to which model will be saved. It can be written 'filename_{epoch}_{loss}' and those values will be filled in before saving.
- **monitor** – (string in {'val_loss', 'loss'}): whether to monitor train or val loss
- **save_best_only** – (bool): whether to only save if monitored value has improved
- **save_weights_only** – (bool): whether to save entire model or just weights NOTE: only *True* is supported at the moment
- **max_save** – (integer > 0 or -1): the max number of models to save. Older model checkpoints will be overwritten if necessary. Set equal to -1 to have no limit
- **verbose** – (integer in {0, 1}): verbosity level

on_epoch_end(*epoch*, *logs=None*)

save_checkpoint(*epoch*, *file*, *is_best=False*)

Saves checkpoint to file :param epoch: (int): epoch number :param file: (string): file location :param is_best: (bool): whether this is the best result seen thus far :return:

2.5 Conditions

Conditions are useful for any custom pre- and post-processing that must be done on batches of data. Module trainer maintains two separate condition lists that are executed before/after the network forward pass.

An example of a condition could be an Assert that needs to be performed before data is processed. A more advanced example of a condition could be code that modifies the network based on input or output

class pywick.conditions.**Condition**

Default class from which all other Condition implementations inherit.

reset()

class pywick.conditions.**SegmentationInputAsserts**

Executes segmentation-specific asserts before executing forward pass on inputs

reset()

class pywick.conditions.**SegmentationOutputAsserts**(*num_classes*)

Executes segmentation-specific asserts after executing forward pass on inputs

reset()

2.6 Constraints

Constraints can be selectively applied on layers using regular expressions. Constraints can be explicit (hard) constraints applied at an arbitrary batch or epoch frequency, or they can be implicit (soft) constraints similar to regularizers where the the constraint deviation is added as a penalty to the total model loss.

class pywick.constraints.**Constraint**
Default class from which all Constraint implementations inherit.

class pywick.constraints.**MaxNorm**(*value*, *axis=0*, *frequency=1*, *unit='batch'*, *module_filter='*'*)
MaxNorm weight constraint.

Constrains the weights incident to each hidden unit to have a norm less than or equal to a desired value.

Any hidden unit vector with a norm less than the max norm constaint will not be altered.

class pywick.constraints.**NonNeg**(*frequency=1*, *unit='batch'*, *module_filter='*'*)
Constrains the weights to be non-negative.

class pywick.constraints.**UnitNorm**(*frequency=1*, *unit='batch'*, *module_filter='*'*)
UnitNorm constraint.

Constraints the weights to have column-wise unit norm

2.7 Datasets

Datasets are the primary mechanism by which Pytorch assembles training and testing data to be used while training neural networks. While *pytorch* already provides a number of handy datasets and *torchvision* further extends them to common academic sets, the implementations below provide some very powerful options for loading all kinds of data. We had to extend the default Pytorch implementation as by default it does not keep track of some useful metadata. That said, you can use our datasets in the normal fashion you're used to with Pytorch.

2.7.1 BaseDataset

class pywick.datasets.BaseDataset.**BaseDataset**
Bases: object

An abstract class representing a Dataset.

All other datasets should subclass it. All subclasses should override __len__, that provides the size of the dataset, and __getitem__, supporting integer indexing in range from 0 to len(self) exclusive.

add_co_transform(*transform*, *add_to_front=True*, *idx=None*)

add_input_transform(*transform*, *add_to_front=True*, *idx=None*)

add_target_transform(*transform*, *add_to_front=True*, *idx=None*)

fit_transforms()
Make a single pass through the entire dataset in order to fit any parameters of the transforms which require the entire dataset. e.g. StandardScaler() requires mean and std for the entire dataset.

If you dont call this fit function, then transforms which require properties of the entire dataset will just work at the batch level. e.g. StandardScaler() will normalize each batch by the specific batch mean/std

load(*num_samples=None*, *load_range=None*)
Load all data or a subset of the data into actual memory. For instance, if the inputs are paths to image files, then this function will actually load those images.

Parameters

- **num_samples** – (int (optional)): number of samples to load. if None, will load all
- **load_range** – (numpy array of integers (optional)): the index range of images to load e.g. np.arange(4) loads the first 4 inputs+targets

2.7.2 CSVDataset

class pywick.datasets.CSVDataset.**CSVDataset**(*csv*, *input_cols=None*, *target_cols=None*, *input_transform=None*, *target_transform=None*, *co_transform=None*, *apply_transforms_individually=False*)
Bases: *pywick.datasets.BaseDataset.BaseDataset*

Initialize a Dataset from a CSV file/dataframe. This does NOT actually load the data into memory if the `csv` parameter contains filepaths.

Parameters

- **csv** – (string or pandas.DataFrame): if string, should be a path to a .csv file which can be loaded as a pandas dataframe
- **input_cols** – (list of ints, or list of strings): which column(s) to use as input arrays. If int(s), should be column indicies. If str(s), should be column names
- **target_cols** – (list of ints, or list of strings): which column(s) to use as input arrays. If int(s), should be column indicies. If str(s), should be column names
- **input_transform** – (transform): tranform to apply to inputs during runtime loading
- **target_tranform** – (transform): transform to apply to targets during runtime loading
- **co_transform** – (transform): transform to apply to both inputs and targets simultaneously during runtime loading
- **apply_transforms_individually** – (bool): Whether to apply transforms to individual inputs or to an input row as a whole (default: False)

copy(*df=None*)
Creates a copy of itself (including transforms and other params).

Parameters **df** – dataframe to include in the copy. If not specified, uses the internal dataframe inside this instance (if any)

Returns

split_by_column(*col*)
Split this dataset object into multiple dataset objects based on the unique factors of the given column. The number of returned datasets will be equal to the number of unique values in the given column. The transforms and original dataframe will all be transferred to the new datasets

Useful for splitting a dataset into train/val/test datasets.

Parameters **col** – (integer or string) which column to split the data on. if int, should be column index. if str, should be column name

Returns list of new datasets with transforms copied

train_test_split(*train_size*)
Define a split for the current dataset where some part of it is used for training while the remainder is used for testing

Parameters **train_size** – (int): length of the training dataset. The remainder will be returned as the test dataset

Returns tuple of datasets (train, test)

2.7.3 ClonedFolderDataset

class pywick.datasets.ClonedFolderDataset.**ClonedFolderDataset**(*data*, *meta_data*, ***kwargs*)
Bases: *pywick.datasets.FolderDataset.FolderDataset*

Dataset that can be initialized with a dictionary of internal parameters (useful when trying to clone a FolderDataset)

Parameters

- **data** – (list): list of data on which the dataset operates
- **meta_data** – (dict): parameters that correspond to the target dataset's attributes
- **kwargs** – (args): variable set of key-value pairs to set as attributes for the dataset

pywick.datasets.ClonedFolderDataset.**random_split_dataset**(*orig_dataset*, *splitRatio=0.8*, *random_seed=None*)
Randomly split the given dataset into two datasets based on the provided ratio

Parameters

- **orig_dataset** – (UsefulDataset): dataset to split (of type pywick.datasets.UsefulDataset)
- **splitRatio** – (float): ratio to use when splitting the data
- **random_seed** – (int): random seed for replicability of results

Returns tuple of split ClonedFolderDatasets

2.7.4 FolderDataset

class pywick.datasets.FolderDataset.**FolderDataset**(*root*, *class_mode='label'*, *class_to_idx=None*, *input_regex='*'*, *rel_target_root=''*, *target_prefix=''*, *target_postfix=''*, *target_extension='png'*, *transform=None*, *target_transform=None*, *co_transform=None*, *apply_co_transform_first=True*, *default_loader='pil'*, *target_loader=None*, *exclusion_file=None*, *target_index_map=None*)
Bases: *pywick.datasets.UsefulDataset.UsefulDataset*

An incredibly versatile dataset class for loading out-of-memory data.

First, the relevant directory structures are traversed to find all necessary files.

Then provided loader(s) is/(are) invoked on inputs and targets.

Finally provided transforms are applied with optional ability to specify the order of individual and co-transforms.

The rel_target_root parameter is used for image segmentation cases Typically the structure will look like the following:

|- root (aka training images)

- dir1

- dir2

|- masks (aka label images)

- dir1

- dir2

Parameters

- **root** – (string): path to main directory
- **class_mode** – (string in *{'label', 'image', 'path'}*): type of target sample to look for and return

  label = return class folder as target

  image = return another image as target (determined by optional target_prefix/postfix). NOTE: if class_mode == 'image', in addition to input, you must also provide `rel_target_root`, `target_prefix` or `target_postfix` (in any combination).

  path = determines paths for inputs and targets and applies the respective loaders to the path
- **class_to_idx** – (dict): If specified, the given class_to_idx map will be used. Otherwise one will be derived from the directory structure.
- **input_regex** – (string *(default is any valid image file)*): regular expression to find input images. e.g. if all your inputs have the word 'input', you'd enter something like input_regex='*input*'
- **rel_target_root** – (string *(default is Nothing)*): root of directory where to look for target images RELATIVE to the root dir (first arg)
- **target_prefix** – (string *(default is Nothing)*): prefix to use (if any) when trying to locate the matching target
- **target_postfix** – (string): postfix to use (if any) when trying to locate the matching target
- **transform** – (torch transform): transform to apply to input sample individually
- **target_transform** – (torch transform): transform to apply to target sample individually

- **co_transform** – (torch transform): transform to apply to both the input and the target
- **apply_co_transform_first** – (bool): whether to apply the co-transform before or after individual transforms (default: True = before)
- **default_loader** – (string in *{'npy', 'pil'}* or function *(default: pil)*): defines how to load samples from file. Will be applied to both input and target unless a separate target_loader is defined. if a function is provided, it should take in a file path as input and return the loaded sample.
- **target_loader** – (string in *{'npy', 'pil'}* or function *(default: pil)*): defines how to load target samples from file. If a function is provided, it should take in a file path as input and return the loaded sample.
- **exclusion_file** – (string): list of files to exclude when enumerating all files. The list must be a full path relative to the root parameter
- **target_index_map** – (dict *(defaults to binary mask: {255:1})*): a dictionary that maps pixel values in the image to classes to be recognized.

  Used in conjunction with 'image' class_mode to produce a label for semantic segmentation For semantic segmentation this is required so the default is a binary mask. However, if you want to turn off this feature then specify target_index_map=None

getdata()

Data that the Dataset class operates on. Typically iterable/list of tuple(label,target). Note: This is different than simply calling myDataset.data because some datasets are comprised of multiple other datasets! The dataset returned should be the *combined* dataset!

Returns iterable - Representation of the entire dataset (combined if necessary from multiple other datasets)

getmeta_data()

Additional data to return that might be useful to consumer. Typically a dict.

Returns dict(any)

pywick.datasets.FolderDataset.**bw_image_loader**(*path*)

pywick.datasets.FolderDataset.**identity_x**(*x*)

pywick.datasets.FolderDataset.**rgb_image_loader**(*path*)

pywick.datasets.FolderDataset.**rgba_image_loader**(*path*)

2.7.5 MultiFolderDataset

class pywick.datasets.MultiFolderDataset.**MultiFolderDataset**(*roots*, *class_mode='label'*, *class_to_idx=None*, *input_regex='*'*, *rel_target_root=''*, *target_prefix=''*, *target_postfix=''*, *target_extension='png'*, *transform=None*, *target_transform=None*, *co_transform=None*, *apply_co_transform_first=True*, *default_loader='pil'*, *target_loader=None*, *exclusion_file=None*, *target_index_map=None*)

Bases: *pywick.datasets.FolderDataset.FolderDataset*

This class extends the FolderDataset with abilty to supply multiple root directories. The `rel_target_root` must exist relative to each root directory. For complete description of functionality see `FolderDataset`

Parameters

- **roots** – (list): list of root directories to traverse
- **class_mode** – (string in *{'label', 'image', 'path'}):* type of target sample to look for and return

  label = return class folder as target

  image = return another image as target (determined by optional target_prefix/postfix)

  NOTE: if class_mode == 'image', in addition to input, you must also provide rel_target_root, target_prefix or target_postfix (in any combination).

  path = determines paths for inputs and targets and applies the respective loaders to the path
- **class_to_idx** – (dict): If specified, the given class_to_idx map will be used. Otherwise one will be derived from the directory structure.
- **input_regex** – (string *(default is any valid image file)*): regular expression to find input images

  e.g. if all your inputs have the word 'input', you'd enter something like input_regex='*input*'
- **rel_target_root** – (string *(default is Nothing)*): root of directory where to look for target images RELATIVE to the root dir (first arg)
- **target_prefix** – (string *(default is Nothing)*): prefix to use (if any) when trying to locate the matching target
- **target_postfix** – (string): postfix to use (if any) when trying to locate the matching target
- **transform** – (torch transform): transform to apply to input sample individually

- **target_transform** – (torch transform): transform to apply to target sample individually
- **co_transform** – (torch transform): transform to apply to both the input and the target
- **apply_co_transform_first** – (bool): whether to apply the co-transform before or after individual transforms (default: True = before)
- **default_loader** – (string in *{'npy', 'pil'}* or function *(default: pil)*): defines how to load samples from file. Will be applied to both input and target unless a separate target_loader is defined.

  if a function is provided, it should take in a file path as input and return the loaded sample.
- **target_loader** – (string in *{'npy', 'pil'}* or function *(default: pil)*): defines how to load target samples from file

  if a function is provided, it should take in a file path as input and return the loaded sample.
- **exclusion_file** – (string): list of files to exclude when enumerating all files. The list must be a full path relative to the root parameter
- **target_index_map** – (dict '(defaults to binary mask: {255:1})): a dictionary that maps pixel values in the image to classes to be recognized.

  Used in conjunction with 'image' class_mode to produce a label for semantic segmentation For semantic segmentation this is required so the default is a binary mask. However, if you want to turn off this feature then specify target_index_map=None

2.7.6 PredictFolderDataset

class pywick.datasets.PredictFolderDataset.**PredictFolderDataset**(*root*, *input_regex='*'*, *input_transform=None*, *input_loader=<function <lambda>>*, *target_loader=None*, *exclusion_file=None*)

Bases: *pywick.datasets.FolderDataset.FolderDataset*

Convenience class for loading out-of-memory data that is more geared toward prediction data loading (where ground truth is not available).

If not transformed in any way (either via one of the loaders or transforms) the inputs and targets will be identical (paths to the discovered files)

Instead, the intended use is that the input path is loaded into some kind of binary representation (usually an image), while the target is either left as a path or is post-processed to accommodate some special need.

Parameters

- **root** – (string): path to main directory
- **input_regex** – (string *(default is any valid image file)*): regular expression to find inputs. e.g. if all your inputs have the word 'input', you'd enter something like input_regex='*input*'
- **input_transform** – (torch transform): transform to apply to each input before returning

- **input_loader** – (callable *(default: identity)*): defines how to load input samples from file. If a function is provided, it should take in a file path as input and return the loaded sample. Identity simply returns the input.
- **target_loader** – (callable *(default: None)*): defines how to load target samples from file (which, in our case, are the same as inputs) If a function is provided, it should take in a file path as input and return the loaded sample.
- **exclusion_file** – (string): list of files to exclude when enumerating all files. The list must be a full path relative to the root parameter

2.7.7 TensorDataset

class pywick.datasets.TensorDataset.**TensorDataset**(*inputs*, *targets=None*, *input_transform=None*, *target_transform=None*, *co_transform=None*)

Bases: *pywick.datasets.BaseDataset.BaseDataset*

Dataset class for loading in-memory data.

Parameters

- **inputs** – (numpy array)
- **targets** – (numpy array)
- **input_transform** – (transform): transform to apply to input sample individually
- **target_transform** – (transform): transform to apply to target sample individually
- **co_transform** – (transform): transform to apply to both input and target sample simultaneously

2.7.8 UsefulDataset

class pywick.datasets.UsefulDataset.**UsefulDataset**

Bases: sphinx.ext.autodoc.importer._MockObject

A torch.utils.data.Dataset class with additional useful functions.

getdata()

Data that the Dataset class operates on. Typically iterable/list of tuple(label,target). Note: This is different than simply calling myDataset.data because some datasets are comprised of multiple other datasets! The dataset returned should be the *combined* dataset!

Returns iterable - Representation of the entire dataset (combined if necessary from multiple other datasets)

getmeta_data()

Additional data to return that might be useful to consumer. Typically a dict.

Returns dict(any)

2.7.9 data utilities

pywick.datasets.data_utils.**get_dataset_mean_std**(*data_set*, *img_size=256*, *output_div=255.0*)

Computes channel-wise mean and std of the dataset. The process is memory-intensive as the entire dataset must

fit into memory. Therefore, each image is scaled down to img_size first (default: 256).

Assumptions:

1. dataset uses PIL to read images
2. Images are in RGB format.

Parameters

- **data_set** – (pytorch Dataset)
- **img_size** – (int): scale of images at which to compute mean/std (default: 256)
- **output_div** – (float *{1.0, 255.0}*): Image values are naturally in 0-255 value range so the returned output is divided by output_div. For example, if output_div = 255.0 then mean/std will be in 0-1 range.

Returns (mean, std) as per-channel values ([r,g,b], [r,g,b])

`pywick.datasets.data_utils.`**`npy_loader`**(*path*, *color_space=None*)
Convenience loader for numeric files (e.g. arrays of numbers)

`pywick.datasets.data_utils.`**`pil_loader`**(*path*, *color_space=''*)
Attempts to load a file using PIL with provided `color_space`.

Parameters

- **path** – (string): file to load
- **color_space** – (string, one of *{rgb, rgba, L, 1, binary}*): Specifies the colorspace to use for PIL loading. If not provided a simple `Image.open(path)` will be performed.

Returns PIL image

`pywick.datasets.data_utils.`**`pil_loader_bw`**(*path*)
Convenience loader for B/W files (e.g. *.png with only one color chanel*)

`pywick.datasets.data_utils.`**`pil_loader_rgb`**(*path*)
Convenience loader for RGB files (e.g. *.jpg*)

2.8 Functions

Here you can find a collection of functions that are used in neural networks. One of the most important aspects of a neural network is a good activation function. Pytorch already has a solid collection of activation functions but here are a few more experimental ones to play around with.

2.8.1 CyclicLR

class `pywick.functions.cyclicLR.`**`CyclicLR`**(*optimizer*, *base_lr=0.001*, *max_lr=0.006*, *step_size=2000*, *mode='triangular'*, *gamma=1.0*, *scale_fn=None*, *scale_mode='cycle'*, *last_batch_iteration=-1*)

Bases: `object`

Sets the learning rate of each parameter group according to cyclical learning rate policy (CLR). The policy cycles the learning rate between two boundaries with a constant frequency, as detailed in the paper Cyclical

Learning Rates for Training Neural Networks. The distance between the two boundaries can be scaled on a per-iteration or per-cycle basis.

Cyclical learning rate policy changes the learning rate after every batch. *batch_step* should be called after a batch has been used for training. To resume training, save *last_batch_iteration* and use it to instantiate *CycleLR*.

This class has three built-in policies, as put forth in the paper:

triangular: A basic triangular cycle w/ no amplitude scaling.

triangular2: A basic triangular cycle that scales initial amplitude by half each cycle.

exp_range: A cycle that scales initial amplitude by gamma**(cycle iterations) at each cycle iteration.

This implementation was adapted from the github repo: bckenstler/CLR

Parameters

- **optimizer** – (Optimizer): Wrapped optimizer.
- **base_lr** – (float or list): Initial learning rate which is the lower boundary in the cycle for each param groups. Default: 0.001
- **max_lr** – (float or list): Upper boundaries in the cycle for each parameter group. Functionally, it defines the cycle amplitude (max_lr - base_lr). The lr at any cycle is the sum of base_lr and some scaling of the amplitude; therefore max_lr may not actually be reached depending on scaling function. Default: 0.006
- **step_size** – (int): Number of training iterations per half cycle. Authors suggest setting step_size 2-8 x training iterations in epoch. Default: 2000
- **mode** – (str): One of {triangular, triangular2, exp_range}. Values correspond to policies detailed above. If scale_fn is not None, this argument is ignored. Default: 'triangular'
- **gamma** – (float): Constant in 'exp_range' scaling function: gamma**(cycle iterations) Default: 1.0
- **scale_fn** – (function): Custom scaling policy defined by a single argument lambda function, where 0 <= scale_fn(x) <= 1 for all x >= 0. mode paramater is ignored Default: None
- **scale_mode** – (str): {'cycle', 'iterations'}. Defines whether scale_fn is evaluated on cycle number or cycle iterations (training iterations since start of cycle). Default: 'cycle'
- **last_batch_iteration** – (int): The index of the last batch. Default: -1

Example:

```
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
>>> scheduler = torch.optim.CyclicLR(optimizer)
>>> data_loader = torch.utils.data.DataLoader(...)
>>> for epoch in range(10):
>>>     for batch in data_loader:
>>>         scheduler.batch_step()
>>>         train_batch(...)
```

batch_step(*batch_iteration=None*)

get_lr()

2.8.2 Aria + Swish

class pywick.functions.swish.**Aria**(*A=0*, *K=1.0*, *B=1.0*, *v=1.0*, *C=1.0*, *Q=1.0*)
Bases: sphinx.ext.autodoc.importer._MockObject

Aria activation function described in this paper.

forward(*x*)

class pywick.functions.swish.**Aria2**(*a=1.5*, *b=2.0*)
Bases: sphinx.ext.autodoc.importer._MockObject

ARiA2 activation function, a special case of ARiA, for ARiA = f(x, 1, 0, 1, 1, b, 1/a)

forward(*x*)

class pywick.functions.swish.**HardSwish**(*inplace: bool = False*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*x*)

class pywick.functions.swish.**Swish**(*b=1.0*)
Bases: sphinx.ext.autodoc.importer._MockObject

Swish activation function, a special case of ARiA, for ARiA = f(x, 1, 0, 1, 1, b, 1)

forward(*x*)

pywick.functions.swish.**hard_swish**(*x*, *inplace: bool = False*)

2.9 Gridsearch

When trying to find the right hyperparameters for your neural network, sometimes you just have to do a lot of trial and error. Currently, our Gridsearch implementation is pretty basic, but it allows you to supply ranges of input values for various metaparameters and then executes training runs in either random or sequential fashion.

Warning: this class is a bit underdeveloped. Tread with care.

2.9.1 Gridsearch

class pywick.gridsearch.gridsearch.**GridSearch**(*function*, *grid_params*, *search_behavior='exhaustive'*, *args_as_dict=True*)

Simple GridSearch to apply to a generic function

Parameters

- **function** – (function): function to perform grid search on
- **grid_params** – (dict): dictionary mapping variable names to lists of possible inputs aka..

  {'input_a':['dog', 'cat', 'stuff'], 'input_b':[3, 10, 22]}
- **search_behavior** – (string): how to perform the search. Options are: 'exhaustive', 'sampled_x.x' (where *x.x* is sample threshold 0.0 < 1.0)

  exhaustive - try every parameter in order they are specified in the dictionary (last key gets all its values searched first)

  sampled - sample from the dictionary of params with specified threshold. The random tries *below* the threshold will be executed

- **args_as_dict** – (bool): There are two ways to pass parameters into a function:

  1. Simply use each key in grid_params as a variable to pass to the function (and change those variable values according to the mapping inside grid_params)

  2. Pass a single dictionary to the function where the keys of the dictionary themselves are changed according to the grid_params

  defaults to dict

run()
Runs GridSearch by iterating over options as specified :return:

2.9.2 Pipeline

class pywick.gridsearch.pipeline.**Pipeline**(*ordered_func_list*, *func_args=None*)
Defines a pipeline for operating on data. Output of first function will be passed to the second and so forth.

Parameters

- **ordered_func_list** – (list): list of functions to call
- **func_args** – (dict): optional dictionary of params to pass to functions in addition to last output the dictionary should be in the form of: func_name: list(params)

add_after(*func*, *args_dict=None*)
Add a function to be applied at the end of the pipeline

Parameters **func** – The function to apply

add_before(*func*, *args_dict=None*)
Add a function to be applied before the rest in the pipeline

Parameters **func** – The function to apply

call(*input*)
Apply the functions in current Pipeline to an input.

Parameters **input** – The input to process with the Pipeline.

static identity(*x*)
Return a copy of the input.

This is here for serialization compatibility with pickle.

2.10 Initializers

It is very important to initialize your neural network with correct weights before training. This is not as trivial as it seems as simple initialization like 0, 1, or even the normal distribution usually yield poor results. Most commonly, weights are initialized to be small non-zero values. See this discussion for more info.

class pywick.initializers.**ConstantInitializer**(*value*, *bias=False*, *bias_only=False*, *module_filter='*'*)
Bases: *pywick.initializers.Initializer*

class pywick.initializers.**GeneralInitializer**(*initializer*, *bias=False*, *bias_only=False*, ***kwargs*)
Bases: *pywick.initializers.Initializer*

class pywick.initializers.**Initializer**
Bases: object

Blank Initializer class from which all other Initializers must inherit

class pywick.initializers.**InitializerContainer**(*initializers*)
Bases: object

apply(*model*)

class pywick.initializers.**KaimingNormal**(*a=0*, *mode='fan_in'*, *bias=False*, *bias_only=False*, *module_filter='*'*)
Bases: *pywick.initializers.Initializer*

class pywick.initializers.**KaimingUniform**(*a=0*, *mode='fan_in'*, *bias=False*, *bias_only=False*, *module_filter='*'*)
Bases: *pywick.initializers.Initializer*

class pywick.initializers.**Normal**(*mean=0.0*, *std=0.02*, *bias=False*, *bias_only=False*, *module_filter='*'*)
Bases: *pywick.initializers.Initializer*

class pywick.initializers.**Orthogonal**(*gain=1*, *bias=False*, *bias_only=False*, *module_filter='*'*)
Bases: *pywick.initializers.Initializer*

class pywick.initializers.**Sparse**(*sparsity*, *std=0.01*, *bias=False*, *bias_only=False*, *module_filter='*'*)
Bases: *pywick.initializers.Initializer*

class pywick.initializers.**Uniform**(*a=0*, *b=1*, *bias=False*, *bias_only=False*, *module_filter='*'*)
Bases: *pywick.initializers.Initializer*

class pywick.initializers.**XavierNormal**(*gain=1*, *bias=False*, *bias_only=False*, *module_filter='*'*)
Bases: *pywick.initializers.Initializer*

class pywick.initializers.**XavierUniform**(*gain=1*, *bias=False*, *bias_only=False*, *module_filter='*'*)
Bases: *pywick.initializers.Initializer*

2.11 Losses

Losses are critical to training a neural network well. The training can only make progress if you provide a meaningful measure of loss for each training step. What the loss looks like usually depends on your application. Pytorch has a number of loss functions that you can use out of the box. However, some more advanced and cutting edge loss functions exist that are not (yet) part of Pytorch. We include those below for your experimenting.

Caution: if you decide to use one of these, you will definitely want to peruse the source code first, as it has many additional useful notes and references which will help you.

Keep in mind that losses are specific to the type of task. Classification losses are computed differently from Segmentation losses. Within segmentation domain make sure to use BCE (Binary Cross Entropy) for any work involving binary masks (e.g. num_classes = 1) Make sure to read the documentation and notes (in the code) for each loss to understand how it is applied.

Read this blog post

Note: Logit is the vector of raw (non-normalized) predictions that a classification model generates, which is ordinarily then passed to a normalization function. If the model is solving a multi-class classification problem, logits

typically become an input to the softmax function. The softmax function then generates a vector of (normalized) probabilities with one value for each possible class.

For example, BCEWithLogitsLoss is a BCE that accepts R((-inf, inf)) and automatically applies torch.sigmoid to convert it to ([0,1]) space.

class pywick.losses.**ActiveContourLoss**(*len_w=1.0*, *reg_w=1.0*, *apply_log=True*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

Learning Active Contour Models for Medical Image Segmentation Note that is only works for B/W masks right now... which is kind of the point of this loss as contours in RGB should be cast to B/W before computing the loss.

Params:

param len_w (float, default=1.0) - The multiplier to use when adding boundary loss.

param reg_w (float, default=1.0) - The multiplier to use when adding region loss.

param apply_log (bool, default=True) - Whether to transform the log into log space (due to the

forward(*logits*, *target*)

class pywick.losses.**BCEDiceFocalLoss**(*focal_param, weights=[1.0, 1.0, 1.0], **kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

Parameters

- **num_classes** – number of classes
- **gamma** – (float,double) gamma > 0 reduces the relative loss for well-classified examples (p>0.5) putting more focus on hard misclassified example
- **size_average** – (bool, optional) By default, the losses are averaged over each loss element in the batch.
- **weights** – (list(), default = [1,1,1]) Optional weighing (0.0-1.0) of the losses in order of [bce, dice, focal]

forward(*logits*, *targets*)

class pywick.losses.**BCEDiceLoss**(*weight=None*, *size_average=True*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*logits*, *targets*)

class pywick.losses.**BCEDicePenalizeBorderLoss**(*kernel_size=21*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*logits*, *labels*)

to(*device*)

class pywick.losses.**BCEDiceTL1Loss**(*threshold=0.5*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*logits*, *targets*)

class pywick.losses.**BCELoss2d**(*weight=None*, *size_average=True*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*logits*, *targets*)

class pywick.losses.**BCEWithLogitsViewLoss**(*weight=None*, *size_average=True*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

Silly wrapper of nn.BCEWithLogitsLoss because BCEWithLogitsLoss only takes a 1-D array

forward(*input*, *target*)

Parameters

- **input** –
- **target** –

Returns

Simply passes along input.view(-1), target.view(-1)

class pywick.losses.**BDLoss**(***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*logits*, *target*, *bound*)
Takes 2D or 3D logits.

logits: (batch_size, class, x,y,(z)) target: ground truth, shape: (batch_size, 1, x,y,(z)) bound: precomputed distance map, shape (batch_size, class, x,y,(z))

Torch Eigensum description: https://stackoverflow.com/questions/55894693/understanding-pytorch-einsum

class pywick.losses.**BinaryFocalLoss**(*gamma=1.333*, *eps=1e-06*, *alpha=1.0*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

Implementation of binary focal loss. For multi-class focal loss use one of the other implementations.

gamma = 0 is equivalent to BinaryCrossEntropy Loss

forward(*inputs*, *targets*)

class pywick.losses.**ComboBCEDiceLoss**(*use_running_mean=False*, *bce_weight=1*, *dice_weight=1*, *eps=1e-06*, *gamma=0.9*, *combined_loss_only=True*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

Combination BinaryCrossEntropy (BCE) and Dice Loss with an optional running mean and loss weighing.

forward(*outputs*, *targets*)

reset_parameters()

to(*device*)

class pywick.losses.**ComboSemsegLossWeighted**(*use_running_mean=False*, *bce_weight=1*, *dice_weight=1*, *eps=1e-06*, *gamma=0.9*, *use_weight_mask=False*, *combined_loss_only=False*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*outputs*, *targets*, *weights*)

reset_parameters()

to(*device*)

class pywick.losses.**EncNetLoss**(*se_loss=True*, *se_weight=0.2*, *nclass=19*, *aux=False*, *aux_weight=0.4*, *weight=None*, *ignore_index=-1*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

2D Cross Entropy Loss with SE Loss

Specifically used for EncNet. se_loss is the Semantic Encoding Loss from the paper Context Encoding for Semantic Segmentation. It computes probabilities of contexts appearing together.

Without SE_loss and Aux_loss this class simply forwards inputs to Torch's Cross Entropy Loss (nn.CrossEntropyLoss)

forward(**inputs*)

class pywick.losses.**FocalBinaryTverskyFunc**(*alpha=0.5*, *beta=0.7*, *gamma=1.0*, *reduction='mean'*)

Bases: sphinx.ext.autodoc.importer._MockObject

Focal Tversky Loss as defined in this paper

Authors' implementation in Keras.

Params:

param alpha controls the penalty for false positives.

param beta penalty for false negative.

:param gamma : focal coefficient range[1,3] :param reduction: return mode

Notes: alpha = beta = 0.5 => dice coeff alpha = beta = 1 => tanimoto coeff alpha + beta = 1 => F beta coeff add focal index -> loss=(1-T_index)**(1/gamma)

backward(*grad_out*)

Parameters

- **ctx** –
- **grad_out** –

Returns

d_loss/dT_loss=(1/gamma)*(T_loss)**(1/gamma-1) (dT_loss/d_P1) = 2*P_G*[G*(P_G+alpha*P_NG+beta*NP_G)-(G+alpha*NG)]/[(P_G+alpha*P_NG+beta*NP_G)**2]

= 2*P_G

(dT_loss/d_p0)=

forward(*input*, *target*)

class pywick.losses.**FocalBinaryTverskyLoss**(*alpha=0.5*, *beta=0.7*, *gamma=1.0*, *reduction='mean'*, ***kwargs*)

Bases: *pywick.losses.MultiTverskyLoss*

Binary version of Focal Tversky Loss as defined in this paper

Authors' implementation in Keras.

Params:

param alpha controls the penalty for false positives.

param beta penalty for false negative.

:param gamma : focal coefficient range[1,3] :param reduction: return mode

Notes: alpha = beta = 0.5 => dice coeff alpha = beta = 1 => tanimoto coeff alpha + beta = 1 => F beta coeff add focal index -> loss=(1-T_index)**(1/gamma)

forward(*inputs*, *targets*)

class pywick.losses.**FocalLoss**(*l=0.5*, *eps=1e-06*)

Bases: sphinx.ext.autodoc.importer._MockObject

Weighs the contribution of each sample to the loss based in the classification error. If a sample is already classified correctly by the CNN, its contribution to the loss decreases.

Eps Focusing parameter. eps=0 is equivalent to BCE_loss

forward(*logits*, *targets*)

class pywick.losses.**FocalLoss2**(*num_class*, *alpha=None*, *gamma=2*, *balance_index=-1*, *smooth=None*, *size_average=True*)

Bases: sphinx.ext.autodoc.importer._MockObject

This is a implementation of Focal Loss with smooth label cross entropy supported which is proposed in 'Focal Loss for Dense Object Detection. (https://arxiv.org/abs/1708.02002)'

Focal_Loss= -1*alpha*(1-pt)*log(pt)

Params:

param num_class

param alpha (tensor) 3D or 4D the scalar factor for this criterion

param gamma (float,double) gamma > 0 reduces the relative loss for well-classified examples (p>0.5) putting more focus on hard misclassified example

param smooth (float,double) smooth value when cross entropy

param balance_index (int) balance class index, should be specific when alpha is float

param size_average (bool, optional) By default, the losses are averaged over each loss element in the batch.

forward(*logit*, *target*)

class pywick.losses.**FocalLoss3**(*class_num*, *alpha=None*, *gamma=2*, *size_average=True*)

Bases: sphinx.ext.autodoc.importer._MockObject

This criterion is a implemenation of Focal Loss, which is proposed in Focal Loss for Dense Object Detection. Loss(x, class) = - lpha (1-softmax(x)[class])^gamma log(softmax(x)[class])

The losses are averaged across observations for each minibatch.

Params:

param alpha (1D Tensor, Variable) - the scalar factor for this criterion

param gamma (float, double) - gamma > 0

param size_average (bool) - size_average(bool): By default, the losses are averaged over observations for each minibatch. However, if the field size_average is set to False, the losses are instead summed for each minibatch.

forward(*inputs*, *targets*)

class pywick.losses.**L1Loss3d**(*bias=1e-12*, ***kwargs*)

Bases: sphinx.ext.autodoc.importer._MockObject

forward(*output*, *target*)

class pywick.losses.**LovaszSoftmax**(*reduction='mean'*, ***kwargs*)

Bases: sphinx.ext.autodoc.importer._MockObject

forward(*inputs*, *targets*)

lovasz_softmax_flat(*inputs*, *targets*)

prob_flatten(*input*, *target*)

class pywick.losses.**MSE3D**

Bases: sphinx.ext.autodoc.importer._MockObject

forward(*output*, *target*)

class pywick.losses.**MixSoftmaxCrossEntropyOHEMLoss**(*aux=False*, *aux_weight=0.4*, *weight=None*, *ignore_index=-1*, ***kwargs*)

Bases: *pywick.losses.OhemCrossEntropy2d*

Loss taking into consideration class and segmentation targets together, as well as, using OHEM

forward(**inputs*)

Args: predict:(n, c, h, w) target:(n, h, w) weight (Tensor, optional): a manual rescaling weight given to each class.

If given, has to be a Tensor of size "nclasses"

to(*device*)

class pywick.losses.**MultiTverskyLoss**(*alpha=0.5*, *beta=0.5*, *gamma=1.0*, *reduction='mean'*, *weights=None*)

Bases: sphinx.ext.autodoc.importer._MockObject

Tversky Loss for segmentation adaptive with multi class segmentation

Args

param alpha (Tensor, float, optional) controls the penalty for false positives.

param beta (Tensor, float, optional) controls the penalty for false negative.

param gamma (Tensor, float, optional) focal coefficient

param weights (Tensor, optional) a manual rescaling weight given to each class. If given, it has to be a Tensor of size *C*

forward(*inputs*, *targets*)

class pywick.losses.**OHEMSegmentationLosses**(*se_loss=False*, *se_weight=0.2*, *nclass=-1*, *aux=False*, *aux_weight=0.4*, *weight=None*, *ignore_index=-1*)

Bases: *pywick.losses.OhemCrossEntropy2d*

2D Cross Entropy Loss with Auxiliary Loss

forward(**inputs*)

Args: predict:(n, c, h, w) target:(n, h, w) weight (Tensor, optional): a manual rescaling weight given to each class.

If given, has to be a Tensor of size "nclasses"

to(*device*)

class pywick.losses.**OhemCrossEntropy2d**(*ignore_label=-1*, *thresh=0.7*, *min_kept=100000*, *use_weight=True*, ***kwargs*)

Bases: sphinx.ext.autodoc.importer._MockObject

Online Hard Example Loss with Cross Entropy (used for classification)

OHEM description: http://www.erogol.com/online-hard-example-mining-pytorch/

forward(*predict*, *target*, *weight=None*)

Args: predict:(n, c, h, w) target:(n, h, w) weight (Tensor, optional): a manual rescaling weight given to each class.

If given, has to be a Tensor of size "nclasses"

to(*device*)

class pywick.losses.**PoissonLoss**(*bias=1e-12*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*output*, *target*)

class pywick.losses.**PoissonLoss3d**(*bias=1e-12*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*output*, *target*)

class pywick.losses.**SoftDiceLoss**(*smooth=1.0*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*logits*, *targets*)

class pywick.losses.**StableBCELoss**(***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*input*, *target*)

class pywick.losses.**ThresholdedL1Loss**(*threshold=0.5*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*logits*, *targets*)

class pywick.losses.**TverskyLoss**(*alpha*, *beta*, *eps=1e-07*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

Computes the Tversky loss [1]. Args:

param alpha controls the penalty for false positives.

param beta controls the penalty for false negatives.

param eps added to the denominator for numerical stability.

Returns: tversky_loss: the Tversky loss.

Notes: alpha = beta = 0.5 => dice coeff alpha = beta = 1 => tanimoto coeff alpha + beta = 1 => F beta coeff

References: [1]: https://arxiv.org/abs/1706.05721

forward(*logits*, *targets*)

Args:

param logits a tensor of shape [B, C, H, W]. Corresponds to the raw output or logits of the model.

param targets a tensor of shape [B, H, W] or [B, 1, H, W].

return loss

class pywick.losses.**WeightedBCELoss2d**(***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*logits*, *labels*, *weights*)

class pywick.losses.**WeightedSoftDiceLoss**(***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*logits*, *labels*, *weights*)

pywick.losses.**binaryXloss**(*logits*, *label*)

pywick.losses.**compute_step_length**(*x*, *grad*, *active*, *eps=1e-06*)

pywick.losses.**dice_coeff**(*pred*, *target*)

pywick.losses.**dice_coeff_hard_np**(*y_true*, *y_pred*)

pywick.losses.**dice_coefficient**(*logit*, *label*, *isCuda=True*)
WARNING THIS IS VERY SLOW FOR SOME REASON!!

Parameters

- **logit** – calculated guess (expects torch.Tensor)
- **label** – truth label (expects torch.Tensor)

Returns dice coefficient

pywick.losses.**find_proximal**(*x0*, *gam*, *lam*, *eps=1e-06*, *max_steps=20*, *debug={}*)

pywick.losses.**gamma_fast**(*gt*, *permutation*)

pywick.losses.**get_tp_fp_fn**(*net_output*, *gt*, *axes=None*, *mask=None*, *square=False*)
net_output must be (b, c, x, y(, z))) gt must be a label map (shape (b, 1, x, y(, z)) OR shape (b, x, y(, z))) or one hot encoding (b, c, x, y(, z)) if mask is provided it must have shape (b, 1, x, y(, z))) :param net_output: :param gt: :param axes: :param mask: mask must be 1 for valid pixels and 0 for invalid pixels :param square: if True then fp, tp and fn will be squared before summation :return:

pywick.losses.**haussdorf**(*preds: <sphinx.ext.autodoc.importer._MockObject object at 0x7fc0636a6a20>*, *target: <sphinx.ext.autodoc.importer._MockObject object at 0x7fc0636a6a20>*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7fc0636a6a20>

pywick.losses.**hingeloss**(*logits*, *label*)

pywick.losses.**iouloss**(*pred*, *gt*)

pywick.losses.**lovasz_binary**(*margins*, *label*, *prox=False*, *max_steps=20*, *debug={}*)

pywick.losses.**lovasz_grad**(*gt_sorted*)
Computes gradient of the Lovasz extension w.r.t sorted errors See Alg. 1 in paper

pywick.losses.**lovasz_single**(*logit*, *label*, *prox=False*, *max_steps=20*, *debug={}*)

pywick.losses.**lovaszloss**(*logits*, *labels*, *prox=False*, *max_steps=20*, *debug={}*)
The Lovasz-Softmax loss

Parameters

- **logits** –
- **labels** –
- **prox** –
- **max_steps** –
- **debug** –

Returns

class pywick.losses.**mIoULoss**(*weight=None*, *size_average=True*, *num_classes=2*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*inputs*, *target_oneHot*)

pywick.losses.**multi_class_dice_loss**(*output*, *target*, *weights=None*, *ignore_index=None*)

pywick.losses.**naive_single**(*logit*, *label*)

pywick.losses.**naiveloss**(*logits*, *labels*)

pywick.losses.**numpy_haussdorf**(*pred: <sphinx.ext.autodoc.importer._MockObject object at 0x7fc063631a90>*, *target: <sphinx.ext.autodoc.importer._MockObject object at 0x7fc063631ac8>*) → float

pywick.losses.**one_hot**(*t: <sphinx.ext.autodoc.importer._MockObject object at 0x7fc0636a6a20>*, *axis=1*) → bool

pywick.losses.**project**(*gam*, *active*, *members*)

pywick.losses.**simplex**(*t: <sphinx.ext.autodoc.importer._MockObject object at 0x7fc0636a6a20>*, *axis=1*) → bool

pywick.losses.**soft_multiclass_dice_loss**(*y_true*, *y_pred*, *epsilon=1e-06*)

Soft dice loss calculation for arbitrary batch size, number of classes, and number of spatial dimensions. Assumes the *channels_last* format.

 # Arguments y_true: b x X x Y(x Z...) x c One hot encoding of ground truth y_pred: b x X x Y(x Z...) x c Network output, must sum to 1 over c channel (such as after softmax) epsilon: Used for numerical stability to avoid divide by zero errors

 # References V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation https://arxiv.org/abs/1606.04797 More details on Dice loss formulation https://mediatum.ub.tum.de/doc/1395260/1395260.pdf (page 72)

Adapted from https://github.com/Lasagne/Recipes/issues/99#issuecomment-347775022

pywick.losses.**softmax_helper**(*x*)

pywick.losses.**sset**(*a: <sphinx.ext.autodoc.importer._MockObject object at 0x7fc0636a6a20>*, *sub: Iterable*) → bool

pywick.losses.**sum_tensor**(*inp*, *axes*, *keepdim=False*)

pywick.losses.**to_one_hot**(*tensor*, *nClasses*)

pywick.losses.**uniq**(*a: <sphinx.ext.autodoc.importer._MockObject object at 0x7fc0636a6a20>*) → Set

2.12 Meters

Meters are used to accumulate values over time or batch and generally provide some statistical measure of your process.

2.12.1 APMeter

class pywick.meters.apmeter.**APMeter**

Bases: *pywick.meters.meter.Meter*

The APMeter measures the average precision per class.

The APMeter is designed to operate on *NxK* Tensors *output* and *target*, and optionally a *Nx1* Tensor weight where (1) the *output* contains model output scores for *N* examples and *K* classes that ought to be higher when the model is more convinced that the example should be positively labeled, and smaller when the model believes the example should be negatively labeled (for instance, the output of a sigmoid function); (2) the *target* contains only values 0 (for negative examples) and 1 (for positive examples); and (3) the *weight* (> 0) represents weight for each sample.

add(*output*, *target*, *weight=None*)

Add a new observation

Args:

output (Tensor): NxK tensor that for each of the N examples indicates the probability of the example belonging to each of the K classes, according to the model. The probabilities should sum to one over all classes

target (Tensor): binary NxK tensort that encodes which of the K classes are associated with the N-th input (eg: a row [0, 1, 0, 1] indicates that the example is associated with classes 2 and 4)

weight (optional, Tensor): Nx1 tensor representing the weight for each example (each weight > 0)

reset()
Resets the meter with empty member variables

value()
Returns the model's average precision for each class

Return: ap (FloatTensor): 1xK tensor, with avg precision for each class k

2.12.2 AUCMeter

class pywick.meters.aucmeter.**AUCMeter**
Bases: *pywick.meters.meter.Meter*

The AUCMeter measures the area under the receiver-operating characteristic (ROC) curve for binary classification problems. The area under the curve (AUC) can be interpreted as the probability that, given a randomly selected positive example and a randomly selected negative example, the positive example is assigned a higher score by the classification model than the negative example.

The AUCMeter is designed to operate on one-dimensional Tensors *output* and *target*, where (1) the *output* contains model output scores that ought to be higher when the model is more convinced that the example should be positively labeled, and smaller when the model believes the example should be negatively labeled (for instance, the output of a signoid function); and (2) the *target* contains only values 0 (for negative examples) and 1 (for positive examples).

add(*output*, *target*)

reset()

value()

2.12.3 AverageMeter

class pywick.meters.averagemeter.**AverageMeter**
Bases: object

Computes and stores the average and current value

reset()

update(*val*, *n=1*)

2.12.4 AverageValueMeter

class pywick.meters.averagevaluemeter.**AverageValueMeter**
Bases: *pywick.meters.meter.Meter*

Keeps track of mean and standard deviation for some value.

add(*value*, *n=1*)

reset()

value()

2.12.5 ClassErrorMeter

class pywick.meters.classerrormeter.**ClassErrorMeter**(*topk=[1], accuracy=False*)
Bases: *pywick.meters.meter.Meter*

add(*output*, *target*)

reset()

value(*k=-1*)

2.12.6 ConfusionMeter

class pywick.meters.confusionmeter.**ConfusionMeter**(*k*, *normalized=False*)
Bases: *pywick.meters.meter.Meter*

Maintains a confusion matrix for a given classification problem.

The ConfusionMeter constructs a confusion matrix for a multi-class classification problems. It does not support multi-label, multi-class problems: for such problems, please use MultiLabelConfusionMeter.

Parameters

- **(int)** (*k*) – number of classes in the classification problem
- **(boolean)** (*normalized*) – Determines whether or not the confusion matrix is normalized or not

add(*predicted*, *target*)
Computes the confusion matrix of K x K size where K is no of classes

Parameters

- **(tensor)** (*target*) – Can be an N x K tensor of predicted scores obtained from the model for N examples and K classes or an N-tensor of integer values between 0 and K-1.
- **(tensor)** – Can be a N-tensor of integer values assumed to be integer values between 0 and K-1 or N x K tensor, where targets are assumed to be provided as one-hot vectors

reset()

value()

Returns: Confustion matrix of K rows and K columns, where rows corresponds to ground-truth targets and columns corresponds to predicted targets.

2.12.7 MAPMeter

class pywick.meters.mapmeter.**mAPMeter**
Bases: *pywick.meters.meter.Meter*

The mAPMeter measures the mean average precision over all classes.

The mAPMeter is designed to operate on *NxK* Tensors *output* and *target*, and optionally a *Nx1* Tensor weight where (1) the *output* contains model output scores for *N* examples and *K* classes that ought to be higher when

the model is more convinced that the example should be positively labeled, and smaller when the model believes the example should be negatively labeled (for instance, the output of a sigmoid function); (2) the *target* contains only values 0 (for negative examples) and 1 (for positive examples); and (3) the *weight* (> 0) represents weight for each sample.

add(*output*, *target*, *weight=None*)

reset()

value()

2.12.8 Meter

class pywick.meters.meter.**Meter**
Bases: object

Abstract meter class from which all other meters inherit

add()

reset()

value()

2.12.9 MovingAverageValueMeter

class pywick.meters.movingaveragevaluemeter.**MovingAverageValueMeter**(*windowsize*)
Bases: *pywick.meters.meter.Meter*

Keeps track of mean and standard deviation of some value for a given window.

add(*value*)

reset()

value()

2.12.10 MSEMeter

class pywick.meters.msemeter.**MSEMeter**(*root=False*)
Bases: *pywick.meters.meter.Meter*

add(*output*, *target*)

reset()

value()

2.12.11 TimeMeter

class pywick.meters.timemeter.**TimeMeter**(*unit*)
Bases: *pywick.meters.meter.Meter*

This meter is designed to measure the time between events and can be used to measure, for instance, the average processing time per batch of data. It is different from most other meters in terms of the methods it provides:

Mmethods:

- *reset()* resets the timer, setting the timer and unit counter to zero.

- *value()* returns the time passed since the last *reset()*; divided by the counter value when *unit=true*.

reset()

value()

2.13 Models

Neural network models is what deep learning is all about! While you can download some standard models from torchvision, we strive to create a library of models that are on the cutting edge of AI. Whenever possible, *we provide pretrained solutions as well!*

That said, we didn't come up with any of these on our own so we owe a huge debt of gratitude to the many researchers who have shared their models and weights on github.

Caution: While we strive to ensure that all models can be used out of the box, sometimes things become broken due to Pytorch updates or misalignment of the planets. Please don't yell at us. Gently point out what's broken, or even better, submit a pull request to fix it!

Here Be Dragons: Aaand one more thing - we constantly plumb the depths of github for new models or tweaks to existing ones. While we don't list this in the docs, there is a special *testnets* directory with tons of probably broken, semi-working, and at times crazy awesome models and model-variations. If you're interested in the bleeding edge, that's where you'd look (see `models.__init__.py` for what's available)

2.13.1 Torchvision Models

All standard torchvision models are supported out of the box.

- AlexNet
- Densenet (121, 161, 169, 201)
- GoogLeNet
- Inception V3
- Mobilenet V2
- ResNet (18, 34, 50, 101, 152)
- ShuffleNet V2
- SqueezeNet (1.0, 1.1)
- VGG (11, 13, 16, 19)

Keep in mind that if you use torvision loading methods (e.g. `torchvision.models.alexnet(...)`) you will get a vanilla pretrained model based on Imagenet with 1000 classes. However, more typically, you'll want to use a pretrained model with your own dataset (and your own number of classes). In that case you should instead use Pywick's `models.model_utils.get_model(...)` utility function which will do all the dirty work for you and give you a pretrained model but with your custom number of classes!

2.13.2 Classification

Below you will find all the latest image classification models. By convention, model names starting with lowercase are pretrained on imagenet while uppercase are not (vanilla). To load one of the pretrained models with your own number of classes use the `models.model_utils.get_model(...)` function and specify the name of the model exactly

like the pretrained model method name (e.g. if the method name reads `pywick.models.classification.dpn.dualpath.dpn68` then use *dpn68* as the model name for `models.model_utils.get_model(...)`.

BatchNormInception

Implementation of BNInception as described in this paper.

class pywick.models.classification.bninception.**BNInception**(*num_classes=1000*)

pywick.models.classification.bninception.**bninception**(*pretrained='imagenet'*)
Pretrained BNInception

DualPathNet

PyTorch implementation of Dual Path Networks. Based on original MXNet implementation with many ideas from another PyTorch implementation.

This implementation is compatible with the pretrained weights from cypw's MXNet implementation.

class pywick.models.classification.dpn.dualpath.**DPN**(*small=False*, *num_init_features=64*, *k_r=96*, *groups=32*, *b=False*, *k_sec=(3*, *4*, *20*, *3)*, *inc_sec=(16*, *32*, *24*, *128)*, *num_classes=1000*, *test_time_pool=False*)

pywick.models.classification.dpn.dualpath.**dpn68**(*num_classes=1000*, *pretrained=False*, *test_time_pool=True*)
Pretrained DPN68 model

pywick.models.classification.dpn.dualpath.**dpn68b**(*num_classes=1000*, *pretrained=False*, *test_time_pool=True*)
Pretrained DPN68b model

pywick.models.classification.dpn.dualpath.**dpn98**(*num_classes=1000*, *pretrained=False*, *test_time_pool=True*)
Pretrained DPN98 model

pywick.models.classification.dpn.dualpath.**dpn131**(*num_classes=1000*, *pretrained=False*, *test_time_pool=True*)
Pretrained DPN131 model

pywick.models.classification.dpn.dualpath.**dpn107**(*num_classes=1000*, *pretrained=False*, *test_time_pool=True*)
Pretrained DPN107 model

FBResnet

Facebook implementation of ResNet

class pywick.models.classification.fbresnet.**FBResNet**(*block*, *layers*, *num_classes=1000*)

pywick.models.classification.fbresnet.**FBResNet18**(*num_classes=1000*)
Constructs a ResNet-18 model.

Args: num_classes

pywick.models.classification.fbresnet.**FBResNet34**(*num_classes=1000*)
Constructs a ResNet-34 model.

Args: num_classes

pywick.models.classification.fbresnet.**FBResNet50**(*num_classes=1000*)
Constructs a ResNet-50 model.

Args: num_classes

pywick.models.classification.fbresnet.**FBResNet101**(*num_classes=1000*)
Constructs a ResNet-101 model.

Args: num_classes

pywick.models.classification.fbresnet.**fbresnet152**(*num_classes=1000*, *pretrained='imagenet'*)
Constructs a ResNet-152 model.

Args: pretrained (bool): If True, returns a model pre-trained on ImageNet

Inception_Resv2_wide

Inception Resnet V2 Wide implementation

class pywick.models.classification.inception_resv2_wide.**InceptionResV2**(*num_classes=1000*)

InceptionResnetV2

InceptionResNetV2 model architecture from the "InceptionV4, Inception-ResNet..." paper.

class pywick.models.classification.inceptionresnetv2.**InceptionResNetV2**(*num_classes=1001*)

pywick.models.classification.inceptionresnetv2.**inceptionresnetv2**(*pretrained='imagenet'*)

InceptionV4

class pywick.models.classification.inceptionv4.**InceptionV4**(*num_classes=1001*)

pywick.models.classification.inceptionv4.**inceptionv4**(*pretrained='imagenet'*)

NASNet

NASNetALarge model architecture from the "NASNet" paper.

pywick.models.classification.nasnet.**nasnetalarge**(*pretrained='imagenet'*)
pretrained NASNet

class pywick.models.classification.nasnet.**NASNetALarge**(*num_classes=1001*)

NASNet_mobile

NASNet Mobile following the paper: Learning Transferable Architectures for Scalable Image Recognition

pywick.models.classification.nasnet_mobile.**nasnetamobile**(*pretrained='imagenet'*)
Pretrained version of NASNet_mobile

class pywick.models.classification.nasnet_mobile.**NASNetAMobile**(*num_classes=1001*, *stem_filters=32*, *penultimate_filters=1056*, *filters_multiplier=2*)

NASNetAMobile (4 @ 1056)

PNASNnet

PNASNet-5 model architecture from the "Progressive Neural Architecture Search" paper.

pywick.models.classification.pnasnet.**pnasnet5large**(*pretrained='imagenet'*)

Pretrained PNASNet

class pywick.models.classification.pnasnet.**PNASNet5Large**(*num_classes=1001*)

Polynet

PolyNet architecture from the paper PolyNet: A Pursuit of Structural Diversity in Very Deep Networks.

class pywick.models.classification.polynet.**PolyNet**(*num_classes=1000*)

pywick.models.classification.polynet.**polynet**(*pretrained='imagenet'*)

Pretrained PolyNet model

Pyramid_Resnet

Implementation from paper: Deep Pyramidal Residual Networks. Not pretrained.

pywick.models.classification.pyramid_resnet.**PyResNet18**(*pretrained=None*, ***kwargs*)

Not Pretrained

pywick.models.classification.pyramid_resnet.**PyResNet34**(*pretrained=None*, ***kwargs*)

Not Pretrained

class pywick.models.classification.pyramid_resnet.**PyResNet**(*block*, *layers*, *in_shape=(3, 256, 256)*, *num_classes=17*)

make_layer(*block*, *planes*, *blocks*, *stride=1*)

Resnet_preact

Preact_Resnet models. Not pretrained.

pywick.models.classification.resnet_preact.**PreactResnet110**(*num_classes*)

pywick.models.classification.resnet_preact.**PreactResnet164_bottleneck**(*num_classes*)

Resnet_swish

Resnet model combined with Swish activation function

class pywick.models.classification.resnet_swish.**ResNet_swish**(*block*, *layers*, *num_classes=1000*)

pywick.models.classification.resnet_swish.**ResNet18_swish**(*pretrained=False*, ***kwargs*)

Constructs a ResNet-18 model. Not pretrained.

pywick.models.classification.resnet_swish.**ResNet34_swish**(*pretrained=False*, ***kwargs*)

Constructs a ResNet-34 model. Not pretrained.

pywick.models.classification.resnet_swish.**ResNet50_swish**(*pretrained=False*, ***kwargs*)

Constructs a ResNet-50 model. Not pretrained.

pywick.models.classification.resnet_swish.**ResNet101_swish**(*pretrained=False*, ***kwargs*)

Constructs a ResNet-101 model. Not pretrained.

pywick.models.classification.resnet_swish.**ResNet152_swish**(*pretrained=False*, ***kwargs*)

Constructs a ResNet-152 model. Not pretrained.

Resnext

Implementation of paper: Aggregated Residual Transformations for Deep Neural Networks.

class pywick.models.classification.resnext.**ResNeXt50_32x4d**(*num_classes=1000*)

pywick.models.classification.resnext.**resnext50_32x4d**(*num_classes=1000*, *pretrained='imagenet'*)

Pretrained Resnext50_32x4d model

class pywick.models.classification.resnext.**ResNeXt101_32x4d**(*num_classes=1000*)

pywick.models.classification.resnext.**resnext101_32x4d**(*pretrained='imagenet'*)

Pretrained Resnext101_32x4d model

class pywick.models.classification.resnext.**ResNeXt101_64x4d**(*num_classes=1000*)

pywick.models.classification.resnext.**resnext101_64x4d**(*pretrained='imagenet'*)

Pretrained ResNeXt101_64x4d model

SENet

SENet implementation as described in: Squeeze-and-Excitation Networks.

class pywick.models.classification.senet.**SENet**(*block*, *layers*, *groups*, *reduction*, *dropout_p=0.2*, *inplanes=128*, *input_3x3=True*, *downsample_kernel_size=3*, *downsample_padding=1*, *num_classes=1000*)

pywick.models.classification.senet.**senet154**(*num_classes=1000*, *pretrained='imagenet'*)

Pretrained SENet154 model

pywick.models.classification.senet.**se_resnet50**(*num_classes=1000*, *pretrained='imagenet'*)
Pretrained SEResNet50 model

pywick.models.classification.senet.**se_resnet101**(*num_classes=1000*, *pretrained='imagenet'*)
Pretrained SEResNet101 model

pywick.models.classification.senet.**se_resnet152**(*num_classes=1000*, *pretrained='imagenet'*)
Pretrained SEResNet152 model

pywick.models.classification.senet.**se_resnext50_32x4d**(*num_classes=1000*, *pretrained='imagenet'*)
Pretrained SEResNext50 model

pywick.models.classification.senet.**se_resnext101_32x4d**(*num_classes=1000*, *pretrained='imagenet'*)
Pretrained SEResNext101 model

WideResnet

Implementation of WideResNet as described in: Wide Residual Networks.

class pywick.models.classification.wideresnet.**WideResNet**(*pooling*, *f*, *params*)

pywick.models.classification.wideresnet.**wideresnet50**(*pooling*)
Pretrained WideResnet50 model

XCeption

Ported to pytorch thanks to [tstandley](https://github.com/tstandley/Xception-PyTorch)

@author: tstandley Adapted by cadene

Creates an Xception Model as defined in:

Francois Chollet Xception: Deep Learning with Depthwise Separable Convolutions.

class pywick.models.classification.xception.**Xception**(*num_classes=1000*)

pywick.models.classification.xception.**xception**(*pretrained='imagenet'*)
Pretrained Xception model.

2.13.3 Localization

FPN

FPN in PyTorch.

Implementation of Feature Pyramid Networks for Object Detection.

class pywick.models.localization.fpn.**FPN**(*block*, *num_blocks*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*x*)

pywick.models.localization.fpn.**FPN101**()

Retina_FPN

RetinaFPN in PyTorch.

Implementation of Focal Loss for Dense Object Detection.

class pywick.models.localization.retina_fpn.**RetinaFPN**(*block*, *num_blocks*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*x*)

pywick.models.localization.retina_fpn.**RetinaFPN101**()

2.13.4 Segmentation

Below you will find all the latest image segmentation models. To get a list of specific model names that are available programmatically, call the `pywick.models.model_utils.get_supported_models(...)` method. To load one of these models with your own number of classes you have two options: 1. You can always load the model directly from the API. Most models allow you to customize *number of classes* as well as *pretrained* options. 2. You can use the `pywick.models.model_utils.get_model(...)` method and pass the name of the model that you want as a string. Note: Some models allow you to customize additional parameters. You can take a look at the `pywick.models.model_utils.get_model(...)` method or at the definition of the model to see what's possible. `pywick.models.model_utils.get_model(...)` takes in a `**kwargs` argument that you can populate with whatever parameters you'd like to pass to the model you are creating.

BiSeNet

Implementation of BiSeNet: Bilateral Segmentation Network for Real-time Semantic Segmentation

class pywick.models.segmentation.bisenet.**BiSeNet**(*num_classes*, *pretrained=True*, *backbone='resnet18'*, *aux=False*, ***kwargs*)

pywick.models.segmentation.bisenet.**BiSeNet_Resnet18**(*num_classes=1*, ***kwargs*)

DANet

Implementation of Dual Attention Network for Scene Segmentation

class pywick.models.segmentation.danet.**DANet**(*num_classes*, *pretrained=True*, *backbone='resnet101'*, *aux=False*, ***kwargs*)

Pyramid Scene Parsing Network

nclass [int] Number of categories for the training dataset.

backbone [string] Pre-trained dilated backbone network type (default:'resnet50'; 'resnet50', 'resnet101' or 'resnet152').

norm_layer [object] Normalization layer used in backbone network (default: `mxnet.gluon.nn.BatchNorm`; for Synchronized Cross-GPU BachNormalization).

aux [bool] Auxiliary loss.

Reference: Jun Fu, Jing Liu, Haijie Tian, Yong Li, Yongjun Bao, Zhiwei Fang,and Hanqing Lu. "Dual Attention Network for Scene Segmentation." *CVPR*, 2019

pywick.models.segmentation.danet.**DANet_Resnet50**(*num_classes=1*, ***kwargs*)

pywick.models.segmentation.danet.**DANet_Resnet101**(*num_classes=1*, ***kwargs*)

pywick.models.segmentation.danet.**DANet_Resnet152**(*num_classes=1*, ***kwargs*)

DenseASPP

Implementation of DenseASPP for Semantic Segmentation in Street Scenes

class pywick.models.segmentation.denseaspp.**DenseASPP**(*num_classes*, *pretrained=True*, *backbone='densenet161'*, *aux=False*, *dilate_scale=8*, ***kwargs*)

Deeplab V2 Resnet

DeepLab v2 - DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs <https://arxiv.org/abs/1606.00915>'_

class pywick.models.segmentation.deeplab_v2_res.**DeepLabv2_ASPP**(*num_classes*, *small=True*, *pretrained=False*, ***kwargs*)

DeeplabV2 Resnet implementation with ASPP.

class pywick.models.segmentation.deeplab_v2_res.**DeepLabv2_FOV**(*num_classes*, *pretrained=True*, ***kwargs*)

DeeplabV2 Resnet implementation with FOV.

Deeplab V3

DeepLab v3 - Rethinking Atrous Convolution for Semantic Image Segmentation

class pywick.models.segmentation.deeplab_v3.**DeepLabv3**(*num_classes*, *small=True*, *pretrained=True*, ***kwargs*)

Deeplab V3+

DeepLab v3+ Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation

class pywick.models.segmentation.deeplab_v3_plus.**DeepLabv3_plus**(*num_classes*, *small=True*, *pretrained=True*, ***kwargs*)

DRNNet

Implementation of Dilated Residual Networks

class pywick.models.segmentation.drn.**DRN**(*block*, *layers*, *num_classes=1000*, *channels=(16*, *32*, *64*, *128*, *256*, *512*, *512*, *512)*, *out_map=False*, *out_middle=False*, *pool_size=28*, *arch='D'*)

pywick.models.segmentation.drn.**drn_c_26**(*pretrained=False*, ***kwargs*)

pywick.models.segmentation.drn.**drn_c_42**(*pretrained=False*, ***kwargs*)

pywick.models.segmentation.drn.**drn_c_58**(*pretrained=False*, ***kwargs*)

pywick.models.segmentation.drn.**drn_d_105**(*pretrained=False*, ***kwargs*)

pywick.models.segmentation.drn.**drn_d_107**(*pretrained=False*, ***kwargs*)

pywick.models.segmentation.drn.**drn_d_22**(*pretrained=False*, ***kwargs*)

pywick.models.segmentation.drn.**drn_d_24**(*pretrained=False*, ***kwargs*)

pywick.models.segmentation.drn.**drn_d_38**(*pretrained=False*, ***kwargs*)

pywick.models.segmentation.drn.**drn_d_40**(*pretrained=False*, ***kwargs*)

pywick.models.segmentation.drn.**drn_d_54**(*pretrained=False*, ***kwargs*)

pywick.models.segmentation.drn.**drn_d_56**(*pretrained=False*, ***kwargs*)

DUC, HDC

Implementation of: Understanding Convolution for Semantic Segmentation

class pywick.models.segmentation.duc_hdc.**ResNetDUC**(*num_classes*, *pretrained=True*, ***kwargs*)

class pywick.models.segmentation.duc_hdc.**ResNetDUCHDC**(*num_classes*, *pretrained=True*, ***kwargs*)

DUNet

Implementation of Decoders Matter for Semantic Segmentation: Data-Dependent Decoding Enables Flexible Feature Aggregation

class pywick.models.segmentation.dunet.**DUNet**(*num_classes*, *pretrained=True*, *backbone='resnet101'*, *aux=False*, ***kwargs*)

Decoders Matter for Semantic Segmentation

Reference: Zhi Tian, Tong He, Chunhua Shen, and Youliang Yan. "Decoders Matter for Semantic Segmentation: Data-Dependent Decoding Enables Flexible Feature Aggregation." CVPR, 2019

pywick.models.segmentation.dunet.**DUNet_Resnet50**(*num_classes=1*, ***kwargs*)

pywick.models.segmentation.dunet.**DUNet_Resnet101**(*num_classes=1*, ***kwargs*)

pywick.models.segmentation.dunet.**DUNet_Resnet152**(*num_classes=1*, ***kwargs*)

ENet

Implementation of ENet: A Deep Neural Network Architecture for Real-Time Semantic Segmentation

class pywick.models.segmentation.enet.**ENet**(*num_classes*, *encoder_relu=False*, *decoder_relu=True*, ***kwargs*)

Generate the ENet model.

Parameters

- **num_classes** – (int): the number of classes to segment.
- **encoder_relu** – (bool, optional): When `True` ReLU is used as the activation function in the encoder blocks/layers; otherwise, PReLU is used. Default: False.

- **decoder_relu** – (bool, optional): When `True` ReLU is used as the activation function in the decoder blocks/layers; otherwise, PReLU is used. Default: True.

FCN8 VGG

Implementation of Fully Convolutional Networks for Semantic Segmentation

class pywick.models.segmentation.fcn8s.**FCN8s**(*num_classes*, *pretrained=True*, *caffe=False*, ***kwargs*)

FCN16 VGG

Implementation of Fully Convolutional Networks for Semantic Segmentation

class pywick.models.segmentation.fcn16s.**FCN16VGG**(*num_classes*, *pretrained=True*, ***kwargs*)

FCN32 VGG

Implementation of Fully Convolutional Networks for Semantic Segmentation

class pywick.models.segmentation.fcn32s.**FCN32VGG**(*num_classes*, *pretrained=True*, ***kwargs*)

FRRN

Implementation of Full Resolution Residual Networks for Semantic Segmentation in Street Scenes

class pywick.models.segmentation.frrn.**frrn**(*num_classes=21*, *model_type=None*, ***kwargs*)

Full Resolution Residual Networks for Semantic Segmentation URL: https://arxiv.org/abs/1611.08323

References: 1) Original Author's code: https://github.com/TobyPDE/FRRN 2) TF implementation by @kiwonjoon: https://github.com/hiwonjoon/tf-frrn

FusionNet

Implementation of FusionNet: A deep fully residual convolutional neural network for image segmentation in connectomics

class pywick.models.segmentation.fusionnet.**FusionNet**(*num_classes*, ***kwargs*)

GCN

class pywick.models.segmentation.gcn.**GCN**(*num_classes*, *pretrained=True*, *k=7*)

GCN Densenet

Implementation of Large Kernel Matters with Densenet backend

class pywick.models.segmentation.gcnnets.gcn_densenet.**GCN_Densenet**(*num_classes*, *pre-trained=True*, *k=7*, ***kwargs*)

GCN NASNet

Implementation of Large Kernel Matters with NASNet backend

class pywick.models.segmentation.gcnnets.gcn_nasnet.**GCN_NASNet**(*num_classes*, *pre-trained=True*, *k=7*)

GCN PSP

Implementation of Large Kernel Matters with PSP backend

class pywick.models.segmentation.gcnnets.gcn_psp.**GCN_PSP**(*num_classes*, *pre-trained=True*, *k=7*, *input_size=512*)

GCN Resnet

Implementation of Large Kernel Matters with Resnet backend.

class pywick.models.segmentation.resnet_gcn.**ResnetGCN**(*num_classes*, *pre-trained=True*, ***kwargs*)

GCN ResNext

Implementation of Large Kernel Matters with Resnext backend

class pywick.models.segmentation.gcnnets.gcn_resnext.**GCN_Resnext**(*num_classes*, *pre-trained=True*, *k=7*, *in-put_size=512*, ***kwargs*)

initialize_weights(**models*)

Linknet

Implementation of LinkNet: Exploiting Encoder Representations for Efficient Semantic Segmentation

class pywick.models.segmentation.mnas_linknets.linknet.**LinkCeption**(*num_classes*, *pretrained=True*, *num_channels=3*, *is_deconv=False*, *decoder_kernel_size=4*, ***kwargs*)

class pywick.models.segmentation.mnas_linknets.linknet.**LinkDenseNet121**(*num_classes*, *pretrained=True*, *num_channels=3*, *is_deconv=False*, *decoder_kernel_size=4*, ***kwargs*)

class pywick.models.segmentation.mnas_linknets.linknet.**LinkDenseNet161**(*num_classes*, *pretrained=True*, *num_channels=3*, *is_deconv=False*, *decoder_kernel_size=4*, ***kwargs*)

class pywick.models.segmentation.mnas_linknets.linknet.**LinkInceptionResNet**(*num_classes*, *pretrained=True*, *num_channels=3*, *is_deconv=False*, *decoder_kernel_size=3*, ***kwargs*)

class pywick.models.segmentation.mnas_linknets.linknet.**LinkNet18**(*num_classes*, *pretrained=True*, *num_channels=3*, *is_deconv=False*, *decoder_kernel_size=4*, ***kwargs*)

class pywick.models.segmentation.mnas_linknets.linknet.**LinkNet34**(*num_classes*, *pretrained=True*, *num_channels=3*, *is_deconv=False*, *decoder_kernel_size=4*, ***kwargs*)

class pywick.models.segmentation.mnas_linknets.linknet.**LinkNet50**(*num_classes*, *pretrained=True*, *num_channels=3*, *is_deconv=False*, *decoder_kernel_size=4*, ***kwargs*)

class pywick.models.segmentation.mnas_linknets.linknet.**LinkNet101**(*num_classes*, *pretrained=True*, *num_channels=3*, *is_deconv=False*, *decoder_kernel_size=4*, ***kwargs*)

class pywick.models.segmentation.mnas_linknets.linknet.**LinkNet152**(*num_classes*, *pretrained=True*, *num_channels=3*, *is_deconv=False*, *decoder_kernel_size=3*, ***kwargs*)

class pywick.models.segmentation.mnas_linknets.linknet.**LinkNeXt**(*num_classes*, *pretrained=True*, *num_channels=3*, *is_deconv=False*, *decoder_kernel_size=4*, ***kwargs*)

class pywick.models.segmentation.mnas_linknets.linknet.**CoarseLinkNet50**(*num_classes*, *pretrained=True*, *num_channels=3*, *is_deconv=False*, *decoder_kernel_size=4*, ***kwargs*)

OCNet

Implementation of OCNet: Object Context Network for Scene Parsing

class pywick.models.segmentation.ocnet.**OCNet**(*num_classes*, *pretrained=True*, *backbone='resnet101'*, *oc_arch='base'*, *aux=False*, ***kwargs*)

nclass [int] Number of categories for the training dataset.

backbone [string] Pre-trained dilated backbone network type (default:'resnet50'; 'resnet50', 'resnet101' or 'resnet152').

norm_layer [object] Normalization layer used in backbone network (default: `nn.BatchNorm`; for Synchronized Cross-GPU BachNormalization).

aux [bool] Auxiliary loss.

Reference: Yuhui Yuan, Jingdong Wang. "OCNet: Object Context Network for Scene Parsing." arXiv preprint arXiv:1809.00916 (2018).

pywick.models.segmentation.ocnet.**OCNet_Base_Resnet101**(*num_classes=1*, ***kwargs*)

pywick.models.segmentation.ocnet.**OCNet_Pyramid_Resnet101**(*num_classes=1*, ***kwargs*)

pywick.models.segmentation.ocnet.**OCNet_ASP_Resnet101**(*num_classes=1*, ***kwargs*)

pywick.models.segmentation.ocnet.**OCNet_Base_Resnet152**(*num_classes=1*, ***kwargs*)

pywick.models.segmentation.ocnet.**OCNet_Pyramid_Resnet152**(*num_classes=1*, ***kwargs*)

pywick.models.segmentation.ocnet.**OCNet_ASP_Resnet152**(*num_classes=1*, ***kwargs*)

RefineNet

Implementation of RefineNet: Multi-Path Refinement Networks for High-Resolution Semantic Segmentation.

class pywick.models.segmentation.refinenet.refinenet.**RefineNet4Cascade**(*num_classes=1*, *pretrained=True*, *input_shape=(1*, *512)*, *features=256*, *resnet_factory=<sphinx.ext.o object>*, *freeze_resnet=False*, ***kwargs*)

class pywick.models.segmentation.refinenet.refinenet.**RefineNet4CascadePoolingImproved**(*num_c pretrained in-put_sh 512)*, *features= resnet_ object>*, *freeze_ **kwa*

PSP

Implementation of Pyramid Scene Parsing Network

class pywick.models.segmentation.lexpsp.**PSPNet**(*num_classes=18*, *pretrained=True*, *backend='densenet121'*, *sizes=(1, 2, 3, 6)*, *psp_size=2048*, *deep_features_size=1024*, ***kwargs*)

SegNet

Implementation of Segnet: A deep convolutional encoder-decoder architecture for image segmentation

class pywick.models.segmentation.seg_net.**SegNet**(*num_classes*, *pretrained=True*, ***kwargs*)

Tiramisu

Implementation of The One Hundred Layers Tiramisu: Fully Convolutional DenseNets for Semantic Segmentation

class pywick.models.segmentation.tiramisu.**FCDenseNet**(*in_channels=3*, *down_blocks=(5, 5, 5, 5, 5)*, *up_blocks=(5, 5, 5, 5, 5)*, *bottleneck_layers=5*, *growth_rate=16*, *out_chans_first_conv=48*, *num_classes=12*, ***kwargs*)

pywick.models.segmentation.tiramisu.**Tiramisu57**(*num_classes*, ***kwargs*)

pywick.models.segmentation.tiramisu.**Tiramisu67**(*num_classes*, ***kwargs*)

pywick.models.segmentation.tiramisu.**Tiramisu103**(*num_classes*, ***kwargs*)

Unet

Implementation of U-net: Convolutional networks for biomedical image segmentation

class pywick.models.segmentation.u_net.**UNet**(*num_classes*, ***kwargs*)
Basic Unet

Unet Sized

Implementation of U-net: Convolutional networks for biomedical image segmentation

class pywick.models.segmentation.carvana_unet.**UNet128**(*in_shape=(3, 128, 128)*, ***kwargs*)

class pywick.models.segmentation.carvana_unet.**UNet256**(*in_shape=(3, 256, 256)*, ***kwargs*)

class pywick.models.segmentation.carvana_unet.**UNet512**(*in_shape=(3, 512, 512)*, ***kwargs*)

class pywick.models.segmentation.carvana_unet.**UNet1024**(*in_shape=(3, 1024, 1024)*, ***kwargs*)

Unet Dilated

Implementation of U-net: Convolutional networks for biomedical image segmentation with dilation convolution operation

class pywick.models.segmentation.unet_dilated.**UNetDilated**(*num_classes*, ***kwargs*)

Unet utilizing dilation

Unet Residual

Implementation of U-net: Convolutional networks for biomedical image segmentation

class pywick.models.segmentation.unet_res.**UNetRes**(*num_classes*, ***kwargs*)

Unet_stack

Implementation of stacked U-net: Convolutional networks for biomedical image segmentation

class pywick.models.segmentation.unet_stack.**UNet_stack**(*input_size=512*, *filters=12*, *kernel_size=3*, *max_stacks=6*, ***kwargs*)

get_n_stacks(*input_size*, ***kwargs*)

class pywick.models.segmentation.unet_stack.**UNet960**(*filters=12*, *kernel_size=3*, ***kwargs*)

2.13.5 utility functions

pywick.models.model_utils.**load_checkpoint**(*checkpoint_path*, *model=None*, *device='cpu'*, *strict=True*, *ignore_chkpt_layers=None*)

Loads weights from a checkpoint into memory. If model is not None then the weights are loaded into the model.

Parameters

- **checkpoint_path** – (string): path to a pretrained network to load weights from
- **model** – the model object to load weights onto (default: None)
- **device** – (string): which device to load model onto (default:'cpu')
- **strict** – (bool): whether to ensure strict key matching (True) or to ignore non-matching keys. (default: True)
- **ignore_chkpt_layers** – one of {string, list) – CURRENTLY UNIMPLEMENTED: whether to ignore some subset of layers from checkpoint. This is usually done when loading checkpoint data into a model with a different number of final classes. In that case, you can pass in a special string: 'last_layer' which will trigger the logic to chop off the last layer of the checkpoint dictionary. Otherwise you can pass in a list of layers to remove from the checkpoint before loading it (e.g. you would do that when loading an inception model that has more than one output layer).

Returns checkpoint

pywick.models.model_utils.**get_model**(*model_type*, *model_name*, *num_classes*, *pretrained=True*, ***kwargs*)

Parameters

- **model_type** – (ModelType): type of model we're trying to obtain (classification or segmentation)
- **model_name** – (string): name of the model. By convention (for classification models) lowercase names represent pretrained model variants while Uppercase do not.
- **num_classes** – (int): number of classes to initialize with (this will replace the last classification layer or set the number of segmented classes)
- **pretrained** – (bool): whether to load the default pretrained version of the model NOTE! NOTE! For classification, the lowercase model names are the pretrained variants while the Uppercase model names are not. The only exception applies to torch.hub models (all efficientnet, mixnet, mobilenetv3, mnasnet, spnasnet variants) where a single lower-case string can be used for vanilla and pretrained versions. Otherwise, it is IN ERROR to specify an Uppercase model name variant with pretrained=True but one can specify a lowercase model variant with pretrained=False (default: True)

Returns model

pywick.models.model_utils.**get_fc_names**(*model_name*, *model_type=<ModelType.CLASSIFICATION: 'classification'>*)

Look up the name of the FC (fully connected) layer(s) of a model. Typically these are the layers that are replaced when transfer-learning from another model. Note that only a handful of models have more than one FC layer. Currently only 'classification' models are supported.

Parameters

- **model_name** – (string) name of the model
- **model_type** – (ModelType) only classification is supported at this time

Returns list names of the FC layers (usually a single one)

pywick.models.model_utils.**get_supported_models**(*type*)

Parameters **type** – (ModelType): classification or segmentation

Returns list (strings) of supported models

2.14 Optimizers

Optimizers govern the path that your neural network takes as it tries to minimize error. Picking the right optimizer and initializing it with the right parameters will either make your network learn successfully or will cause it not to learn at all! Pytorch already implements the most widely used flavors such as SGD, Adam, RMSProp etc. Here we strive to include optimizers that Pytorch has missed (and any cutting edge ones that have not yet been added).

2.14.1 AdamW

class pywick.optimizers.adamw.**AdamW**(*params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0.01*, *amsgrad=False*)

Implements AdamW algorithm.

The original Adam algorithm was proposed in Adam: A Method for Stochastic Optimization. The AdamW variant was proposed in Decoupled Weight Decay Regularization.

Arguments:

params (iterable): iterable of parameters to optimize or dicts defining parameter groups

lr (float, optional): learning rate (default: 1e-3) betas (Tuple[float, float], optional): coefficients used for computing

running averages of gradient and its square (default: (0.9, 0.999))

eps (float, optional): term added to the denominator to improve numerical stability (default: 1e-8)

weight_decay (float, optional): weight decay coefficient (default: 1e-2) amsgrad (boolean, optional): whether to use the AMSGrad variant of this

algorithm from the paper On the Convergence of Adam and Beyond (default: False)

step(*closure=None*)

Performs a single optimization step.

Arguments:

closure (callable, optional): A closure that reevaluates the model and returns the loss.

2.14.2 AddSign

class pywick.optimizers.addsign.**AddSign**(*params*, *lr=0.001*, *beta=0.9*, *alpha=1*, *sign_internal_decay=None*)

Implements AddSign algorithm.

It has been proposed in Neural Optimizer Search with Reinforcement Learning.

Parameters

- **params** – (iterable): iterable of parameters to optimize or dicts defining parameter groups
- **lr** – (float, optional): learning rate (default: 1e-3)
- **beta** – (float, optional): coefficients used for computing running averages of gradient (default: 0.9)
- **alpha** – (float, optional): term added to the internal_decay * sign(g) * sign(m) (default: 1)
- **sign_internal_decay** – (callable, optional): a function that returns an internal decay calculated based on the current training step and the total number of training steps. If None, the internal decay is assumed to be 1.

step(*closure=None*)

Performs a single optimization step.

Parameters **closure** – (callable, optional): A closure that reevaluates the model and returns the loss.

2.14.3 Eve

class pywick.optimizers.eve.**Eve**(*params*, *lr=0.001*, *betas=(0.9*, *0.999*, *0.999)*, *eps=1e-08*, *k=0.1*, *K=10*, *weight_decay=0*)

Implementation of Eve: A Gradient Based Optimization Method with Locally and Globally Adaptive Learning Rates

step(*closure*)

Parameters **closure** – (closure). see http://pytorch.org/docs/optim.html#optimizer-step-closure

Returns loss

2.14.4 Nadam

class pywick.optimizers.nadam.**Nadam**(*params*, *lr=0.002*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0*, *schedule_decay=0.004*)

Implements Nadam algorithm (a variant of Adam based on Nesterov momentum).

It has been proposed in **'Incorporating Nesterov Momentum into Adam'__**.

Parameters

- **params** – (iterable): iterable of parameters to optimize or dicts defining parameter groups
- **lr** – (float, optional): learning rate (default: 2e-3)
- **betas** – (Tuple[float, float], optional): coefficients used for computing running averages of gradient and its square
- **eps** – (float, optional): term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** – (float, optional): weight decay (L2 penalty) (default: 0)
- **schedule_decay** – (float, optional): momentum schedule decay (default: 4e-3)

step(*closure=None*)

Performs a single optimization step.

Parameters **closure** – (callable, optional): A closure that reevaluates the model and returns the loss.

2.14.5 PowerSign

class pywick.optimizers.powersign.**PowerSign**(*params*, *lr=0.001*, *beta=0.9*, *alpha=2.718281828459045*, *sign_internal_decay=None*)

Implements PowerSign algorithm.

It has been proposed in Neural Optimizer Search with Reinforcement Learning.

Parameters

- **params** – (iterable): iterable of parameters to optimize or dicts defining parameter groups
- **lr** – (float, optional): learning rate (default: 1e-3)
- **beta** – (float, optional): coefficients used for computing running averages of gradient (default: 0.9)
- **alpha** – (float, optional): term powered to the internal_decay * sign(g) * sign(m) (default: math.e)
- **sign_internal_decay** – (callable, optional): a function that returns an internal decay calculated based on the current training step and the total number of training steps. If None, the internal decay is assumed to be 1.

step(*closure=None*)

Performs a single optimization step.

Parameters **closure** – (callable, optional): A closure that reevaluates the model and returns the loss.

2.14.6 SGDW

class pywick.optimizers.sgdw.**SGDW**(*params*, *lr=0.003*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*)

Implements stochastic gradient descent warm (optionally with momentum).

It has been proposed in Fixing Weight Decay Regularization in Adam.

Nesterov momentum is based on the formula from On the importance of initialization and momentum in deep learning.

Parameters

- **(iterable)** (*params*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** – (float): learning rate
- **momentum** – (float, optional): momentum factor (default: 0)
- **weight_decay** – (float, optional): weight decay (L2 penalty) (default: 0)
- **dampening** – (float, optional): dampening for momentum (default: 0)
- **nesterov** – (bool, optional): enables Nesterov momentum (default: False)

Example:

```
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
>>> optimizer.zero_grad()
>>> loss_fn(model(input), target).backward()
>>> optimizer.step()
```

Note: The implementation of SGD with Momentum/Nesterov subtly differs from Sutskever et. al. and implementations in some other frameworks.

Considering the specific case of Momentum, the update can be written as

$$v = \rho * v + g$$
$$p = p - lr * v$$

where p, g, v and ρ denote the parameters, gradient, velocity, and momentum respectively.

This is in contrast to Sutskever et. al. and other frameworks which employ an update of the form

$$v = \rho * v + lr * g$$
$$p = p - v$$

The Nesterov version is analogously modified.

2.14.7 SWA

class pywick.optimizers.swa.**SWA**(*optimizer*, *swa_start=None*, *swa_freq=None*, *swa_lr=None*)

Implements Stochastic Weight Averaging (SWA).

Stochastic Weight Averaging was proposed in Averaging Weights Leads to Wider Optima and Better Generalization by Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov and Andrew Gordon Wilson (UAI 2018).

SWA is implemented as a wrapper class taking optimizer instance as input and applying SWA on top of that optimizer.

SWA can be used in two modes: automatic and manual. In the automatic mode SWA running averages are automatically updated every `swa_freq` steps after `swa_start` steps of optimization. If `swa_lr` is provided, the learning rate of the optimizer is reset to `swa_lr` at every step starting from `swa_start`. To use SWA in automatic mode provide values for both `swa_start` and `swa_freq` arguments.

Alternatively, in the manual mode, use *update_swa()* or *update_swa_group()* methods to update the SWA running averages.

In the end of training use *swap_swa_sgd* method to set the optimized variables to the computed averages.

Parameters

- **optimizer** – (torch.optim.Optimizer): optimizer to use with SWA
- **swa_start** – (int): number of steps before starting to apply SWA in automatic mode; if None, manual mode is selected (default: None)
- **swa_freq** – (int): number of steps between subsequent updates of SWA running averages in automatic mode; if None, manual mode is selected (default: None)
- **swa_lr** – (float): learning rate to use starting from step swa_start in automatic mode; if None, learning rate is not changed (default: None)

Examples:

```
>>> from pywick.optimizers import SWA
>>> # automatic mode
>>> base_opt = torch.optim.SGD(model.parameters(), lr=0.1)
>>> opt = SWA(base_opt, swa_start=10, swa_freq=5, swa_lr=0.05)
>>> for _ in range(100):
>>>     opt.zero_grad()
>>>     loss_fn(model(input), target).backward()
>>>     opt.step()
>>> opt.swap_swa_sgd()
>>> # manual mode
>>> opt = SWA(base_opt)
>>> for i in range(100):
>>>     opt.zero_grad()
>>>     loss_fn(model(input), target).backward()
>>>     opt.step()
>>>     if i > 10 and i % 5 == 0:
>>>         opt.update_swa()
>>> opt.swap_swa_sgd()
```

Note: SWA does not support parameter-specific values of `swa_start`, `swa_freq` or `swa_lr`. In automatic mode SWA uses the same `swa_start`, `swa_freq` and `swa_lr` for all parameter groups. If needed, use manual mode with *update_swa_group()* to use different update schedules for different parameter groups.

Note: Call *swap_swa_sgd()* in the end of training to use the computed running averages.

Note: If you are using SWA to optimize the parameters of a Neural Network containing Batch Normalization layers, you need to update the `running_mean` and `running_var` statistics of the Batch Normalization

module. You can do so by using *torchcontrib.optim.swa.bn_update* utility. For further description see this article.

---

add_param_group(*param_group*)
Add a param group to the `Optimizer` s *param_groups*.

This can be useful when fine tuning a pre-trained network as frozen layers can be made trainable and added to the `Optimizer` as training progresses.

Parameters **(dict)** (*param_group*) – Specifies what Tensors should be optimized along

with group specific optimization options.

static bn_update(*loader*, *model*, *device=None*)
Updates BatchNorm running_mean, running_var buffers in the model.

It performs one pass over data in *loader* to estimate the activation statistics for BatchNorm layers in the model.

Parameters

- **(torch.utils.data.DataLoader)** (*loader*) – dataset loader to compute the activation statistics on. Each data batch should be either a tensor, or a list/tuple whose first element is a tensor containing data.
- **(torch.nn.Module)** (*model*) – model for which we seek to update BatchNorm statistics.
- **(torch.device, optional)** (*device*) – If set, data will be trasferred to `device` before being passed into `model`.

load_state_dict(*state_dict*)
Loads the optimizer state.

Parameters **(dict)** (*state_dict*) – SWA optimizer state. Should be an object returned from a call to *state_dict*.

state_dict()
Returns the state of SWA as a `dict`.

It contains three entries:

- **opt_state - a dict holding current optimization state of the base** optimizer. Its content differs between optimizer classes.
- **swa_state - a dict containing current state of SWA. For each** optimized variable it contains swa_buffer keeping the running average of the variable
- param_groups - a dict containing all parameter groups

swap_swa_sgd()
Swaps the values of the optimized variables and swa buffers.

It's meant to be called in the end of training to use the collected swa running averages. It can also be used to evaluate the running averages during training; to continue training *swap_swa_sgd* should be called again.

update_swa()
Updates the SWA running averages of all optimized parameters.

update_swa_group(*group*)
Updates the SWA running averages for the given parameter group.

Parameters **(dict)** (*group*) – Specifies for what parameter group SWA running averages should be updated

Examples:

```
>>> # automatic mode
>>> base_opt = torch.optim.SGD([{'params': [x]},
>>>             {'params': [y], 'lr': 1e-3}], lr=1e-2, momentum=0.9)
>>> opt = torchcontrib.optim.SWA(base_opt)
>>> for i in range(100):
>>>     opt.zero_grad()
>>>     loss_fn(model(input), target).backward()
>>>     opt.step()
>>>     if i > 10 and i % 5 == 0:
>>>         # Update SWA for the second parameter group
>>>         opt.update_swa_group(opt.param_groups[1])
>>> opt.swap_swa_sgd()
```

2.15 Regularizers

class pywick.regularizers.**L1L2Regularizer**(*l1_scale=0.001*, *l2_scale=0.001*, *module_filter='*'*)

Bases: *pywick.regularizers.Regularizer*

reset()

class pywick.regularizers.**L1Regularizer**(*scale=0.001*, *module_filter='*'*)

Bases: *pywick.regularizers.Regularizer*

reset()

class pywick.regularizers.**L2Regularizer**(*scale=0.001*, *module_filter='*'*)

Bases: *pywick.regularizers.Regularizer*

reset()

class pywick.regularizers.**MaxNormRegularizer**(*scale=0.001*, *module_filter='*'*)

Bases: *pywick.regularizers.Regularizer*

MaxNorm regularizer on Weights

Constraints the weights to have column-wise unit norm

reset()

class pywick.regularizers.**NonNegRegularizer**(*scale=0.001*, *module_filter='*'*)

Bases: *pywick.regularizers.Regularizer*

Non-Negativity regularizer on Weights

Constraints the weights to have column-wise unit norm

reset()

class pywick.regularizers.**Regularizer**

Bases: object

reset()

class pywick.regularizers.**RegularizerCallback**(*container*)

Bases: *pywick.callbacks.Callback.Callback*

on_batch_end(*batch*, *logs=None*)

class pywick.regularizers.**RegularizerContainer**(*regularizers*)
Bases: object

get_value()

register_forward_hooks(*model*)

reset()

unregister_forward_hooks()

class pywick.regularizers.**UnitNormRegularizer**(*scale=0.001*, *module_filter='*'*)
Bases: *pywick.regularizers.Regularizer*

UnitNorm constraint on Weights

Constraints the weights to have column-wise unit norm

reset()

2.16 Samplers

Samplers are used during the training phase and are especially useful when your training data is not uniformly distributed among all of your classes.

class pywick.samplers.**ImbalancedDatasetSampler**(*dataset*, *indices=None*, *num_samples=None*)
Bases: sphinx.ext.autodoc.importer._MockObject

Samples elements randomly from a given list of indices for imbalanced dataset

Parameters

- **indices** – (list, optional): a list of indices
- **num_samples** – (int, optional): number of samples to draw

class pywick.samplers.**MultiSampler**(*nb_samples*, *desired_samples*, *shuffle=False*)
Bases: sphinx.ext.autodoc.importer._MockObject

Samples elements more than once in a single pass through the data.

This allows the number of samples per epoch to be larger than the number of samples itself, which can be useful when training on 2D slices taken from 3D images, for instance.

gen_sample_array()

class pywick.samplers.**StratifiedSampler**(*class_vector*, *batch_size*)
Bases: sphinx.ext.autodoc.importer._MockObject

Stratified Sampling

Provides equal representation of target classes in each batch

Parameters

- **class_vector** – (torch tensor): a vector of class labels
- **batch_size** – (int): size of the batch

gen_sample_array()

2.17 Transforms

2.17.1 Affine

Affine transforms implemented on torch tensors, and requiring only one interpolation

class pywick.transforms.affine_transforms.**Affine**(*tform_matrix*, *interp='bilinear'*)

class pywick.transforms.affine_transforms.**AffineCompose**(*transforms*, *interp='bilinear'*)

class pywick.transforms.affine_transforms.**RandomAffine**(*rotation_range=None*, *translation_range=None*, *shear_range=None*, *zoom_range=None*, *interp='bilinear'*, *lazy=False*)

class pywick.transforms.affine_transforms.**RandomChoiceRotate**(*values*, *p=None*, *interp='bilinear'*, *lazy=False*)

class pywick.transforms.affine_transforms.**RandomChoiceShear**(*values*, *p=None*, *interp='bilinear'*, *lazy=False*)

class pywick.transforms.affine_transforms.**RandomChoiceTranslate**(*values*, *p=None*, *interp='bilinear'*, *lazy=False*)

class pywick.transforms.affine_transforms.**RandomChoiceZoom**(*values*, *p=None*, *interp='bilinear'*, *lazy=False*)

class pywick.transforms.affine_transforms.**RandomRotate**(*rotation_range*, *interp='bilinear'*, *lazy=False*)

class pywick.transforms.affine_transforms.**RandomShear**(*shear_range*, *interp='bilinear'*, *lazy=False*)

class pywick.transforms.affine_transforms.**RandomSquareZoom**(*zoom_range*, *interp='bilinear'*, *lazy=False*)

class pywick.transforms.affine_transforms.**RandomTranslate**(*translation_range*, *interp='bilinear'*, *lazy=False*)

class pywick.transforms.affine_transforms.**RandomZoom**(*zoom_range*, *interp='bilinear'*, *lazy=False*)

class pywick.transforms.affine_transforms.**Rotate**(*value*, *interp='bilinear'*, *lazy=False*)

class pywick.transforms.affine_transforms.**Shear**(*value*, *interp='bilinear'*, *lazy=False*)

class pywick.transforms.affine_transforms.**Translate**(*value*, *interp='bilinear'*, *lazy=False*)

class pywick.transforms.affine_transforms.**Zoom**(*value*, *interp='bilinear'*, *lazy=False*)

2.17.2 Distortion

Transforms to distort local or global information of an image

class pywick.transforms.distortion_transforms.**Blur**(*threshold*, *order=5*)
Blur an image with a Butterworth filter with a frequency cutoff matching local block size

class pywick.transforms.distortion_transforms.**RandomChoiceBlur**(*thresholds*, *order=5*)

class pywick.transforms.distortion_transforms.**RandomChoiceScramble**(*blocksizes*)

class pywick.transforms.distortion_transforms.**Scramble**(*blocksize*)
Create blocks of an image and scramble them

2.17.3 Image

Transforms very specific to images such as color, lighting, contrast, brightness, etc transforms

NOTE: Most of these transforms assume your image intensity is between 0 and 1, and are torch tensors (NOT numpy or PIL)

class pywick.transforms.image_transforms.**Brightness**(*value*)

class pywick.transforms.image_transforms.**Contrast**(*value*)

class pywick.transforms.image_transforms.**Gamma**(*value*)

class pywick.transforms.image_transforms.**Grayscale**(*keep_channels=False*)

class pywick.transforms.image_transforms.**RandomBrightness**(*min_val*, *max_val*)

class pywick.transforms.image_transforms.**RandomChoiceBrightness**(*values*, *p=None*)

class pywick.transforms.image_transforms.**RandomChoiceContrast**(*values*, *p=None*)

class pywick.transforms.image_transforms.**RandomChoiceGamma**(*values*, *p=None*)

class pywick.transforms.image_transforms.**RandomChoiceSaturation**(*values*, *p=None*)

class pywick.transforms.image_transforms.**RandomContrast**(*min_val*, *max_val*)

class pywick.transforms.image_transforms.**RandomGamma**(*min_val*, *max_val*)

class pywick.transforms.image_transforms.**RandomGrayscale**(*p=0.5*)

class pywick.transforms.image_transforms.**RandomSaturation**(*min_val*, *max_val*)

class pywick.transforms.image_transforms.**Saturation**(*value*)

pywick.transforms.image_transforms.**rgb_to_hsv**(*x*)
Convert from RGB to HSV

2.17.4 Tensor

class pywick.transforms.tensor_transforms.**AddChannel**(*axis=0*)
Adds a dummy channel to an image, also known as expanding an axis or unsqueezing a dim This will make an image of size (28, 28) to now be of size (1, 28, 28), for example.

param axis: (int): dimension to be expanded to singleton

pywick.transforms.tensor_transforms.**CDHW**
alias of *pywick.transforms.tensor_transforms.ChannelsFirst*

pywick.transforms.tensor_transforms.**CHW**
alias of *pywick.transforms.tensor_transforms.ChannelsFirst*

class pywick.transforms.tensor_transforms.**ChannelsFirst**(*safe_check=False*)
Transposes a tensor so that the channel dim is first. *CHW* and *CDHW* are aliases for this transform.

Parameters **safe_check** – (bool): if true, will check if channels are already first and, if so, will just return the inputs

class pywick.transforms.tensor_transforms.**ChannelsLast**(*safe_check=False*)
Transposes a tensor so that the channel dim is last *HWC* and *DHWC* are aliases for this transform.

Parameters **safe_check** – (bool): if true, will check if channels are already last and, if so, will just return the inputs

class pywick.transforms.tensor_transforms.**Compose**(*transforms*)
Composes (chains) several transforms together.

Parameters **transforms** – (list of transforms) to apply sequentially

pywick.transforms.tensor_transforms.**DHWC**
alias of *pywick.transforms.tensor_transforms.ChannelsLast*

pywick.transforms.tensor_transforms.**ExpandAxis**
alias of *pywick.transforms.tensor_transforms.AddChannel*

pywick.transforms.tensor_transforms.**HWC**
alias of *pywick.transforms.tensor_transforms.ChannelsLast*

class pywick.transforms.tensor_transforms.**Pad**(*size*)
Pads an image to the given size

Parameters **size** – (tuple or list): size of crop

class pywick.transforms.tensor_transforms.**PadNumpy**(*size*)
Pads a Numpy image to the given size Return a Numpy image / image pair Arguments ——— :param size: (tuple or list):

size of crop

class pywick.transforms.tensor_transforms.**RandomChoiceCompose**(*transforms*)
Randomly choose to apply one transform from a collection of transforms

e.g. to randomly apply EITHER 0-1 or -1-1 normalization to an input:

```
>>> transform = RandomChoiceCompose([RangeNormalize(0,1),
                                     RangeNormalize(-1,1)])
>>> x_norm = transform(x) # only one of the two normalizations is applied
```

Parameters **transforms** – (list of transforms) to choose from at random

class pywick.transforms.tensor_transforms.**RandomCrop**(*size*)
Randomly crop a torch tensor

Parameters **size** – (tuple or list): dimensions of the crop

class pywick.transforms.tensor_transforms.**RandomFlip**(*h=True*, *v=False*, *p=0.5*)
Randomly flip an image horizontally and/or vertically with some probability.

Parameters

- **h** – (bool): whether to horizontally flip w/ probability p
- **v** – (bool): whether to vertically flip w/ probability p
- **p** – (float between [0,1]): probability with which to apply allowed flipping operations

class pywick.transforms.tensor_transforms.**RandomOrder**
Randomly permute the channels of an image

class pywick.transforms.tensor_transforms.**RangeNormalize**(*min_val*, *max_val*)
Given min_val: (R, G, B) and max_val: (R,G,B), will normalize each channel of the th.*Tensor to the provided min and max values.

Works by calculating : a = (max'-min')/(max-min)

b = max' - a * max

new_value = a * value + b

where min' & max' are given values, and min & max are observed min/max for each channel

Parameters

- **min_val** – (float or integer): Lower bound of normalized tensor
- **max_val** – (float or integer): Upper bound of normalized tensor

Example:

```
>>> x = th.rand(3,5,5)
>>> rn = RangeNormalize((0,0,10),(1,1,11))
>>> x_norm = rn(x)
```

Also works with just one value for min/max:

```
>>> x = th.rand(3,5,5)
>>> rn = RangeNormalize(0,1)
>>> x_norm = rn(x)
```

class pywick.transforms.tensor_transforms.**Slice2D**(*axis=0*, *reject_zeros=False*)
Take a random 2D slice from a 3D image along a given axis. This image should not have a 4th channel dim.

Parameters

- **axis** – (int *in {0, 1, 2}*): the axis on which to take slices
- **reject_zeros** – (bool): whether to reject slices that are all zeros

class pywick.transforms.tensor_transforms.**SpecialCrop**(*size*, *crop_type=0*)
Perform a special crop - one of the four corners or center crop

Parameters

- **size** – (tuple or list): dimensions of the crop
- **crop_type** – (int in *{0,1,2,3,4}*): 0 = center crop 1 = top left crop 2 = top right crop 3 = bottom right crop 4 = bottom left crop

class pywick.transforms.tensor_transforms.**StdNormalize**
Normalize torch tensor to have zero mean and unit std deviation

class pywick.transforms.tensor_transforms.**ToFile**(*root*)
Saves an image to file. Useful as a pass-through transform when wanting to observe how augmentation affects the data

NOTE: Only supports saving to Numpy currently

Parameters **root** – (string): path to main directory in which images will be saved

class pywick.transforms.tensor_transforms.**ToNumpyType**(*type*)
Converts an object to a specific numpy type (with the idea to be passed to ToTensor() next)

Parameters **type** – (one of ‘{numpy.double, numpy.float, numpy.int64, numpy.int32, and numpy.uint8})

class pywick.transforms.tensor_transforms.**ToTensor**
Converts a numpy array to torch.Tensor

class pywick.transforms.tensor_transforms.**Transpose**(*dim1*, *dim2*)
Swaps two dimensions of a tensor

Parameters

- **dim1** – (int): first dim to switch
- **dim2** – (int): second dim to switch

class pywick.transforms.tensor_transforms.**TypeCast**(*dtype='float'*)
Cast a torch.Tensor to a different type param dtype: (string or torch.*Tensor literal or list) of such

data type to which input(s) will be cast. If list, it should be the same length as inputs.

pywick.transforms.tensor_transforms.**Unsqueeze**
alias of *pywick.transforms.tensor_transforms.AddChannel*

2.18 License

2.18.1 The MIT License (MIT)

```
Some contributions by Nicholas Cullen
Copyright (c) 2017, Nicholas Cullen
All rights reserved.

Some contributions by François Chollet
Copyright (c) 2015, François Chollet
All rights reserved.

Some contributions by ZijunDeng
Copyright (c) 2017, ZijunDeng.
All rights reserved.

Some contributions by Google
Copyright (c) 2015, Google, Inc.
All rights reserved.

All other contributions:
Copyright (c) 2015, the respective contributors.
All rights reserved.
```

LICENSE

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use,

copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

2.18.2 BSD 3-Clause License

BSD 3-Clause License

```
Some contributions by Remi Cadene
Copyright (c) 2017, Remi Cadene
All rights reserved.

All other contributions:
Copyright (c) 2015, the respective contributors.
All rights reserved.
```

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

2.18.3 BSD 2-Clause License

```
Some contributions by ericsun99
Copyright (c) 2018, ericsun99
All rights reserved.

All other contributions:
```

(continues on next page)

(continued from previous page)

```
Copyright (c) 2015, the respective contributors.
All rights reserved.
```

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

2.18.4 BSD License

```
Some contributions by Facebook, Inc.
Copyright (c) 2016, Facebook, Inc. All rights reserved.
For fb.resnet.torch software

All other contributions:
Copyright (c) 2015, the respective contributors.
All rights reserved.
```

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name Facebook nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

2.18.5 Apache 2.0 License

```
Some contributions by Google Inc.
Copyright 2015 Google Inc.
All Rights Reserved.

All other contributions:
Copyright (c) 2015, the respective contributors.
All rights reserved.
```

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

2.18.6 LIP6 License

Copyright (c) 2017 LIP6 Lab

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

–

This product contains portions of third party software provided under Apache 2.0 license dump_filters.py (x)

2.19 Help

Please visit our github page.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z