

---

# PyTorch Lightning Spells

*Release 0.1.0*

**Ceshine Lee**

**May 23, 2022**



## CONTENTS:

<b>1</b>	<b>CV</b>	<b>3</b>
1.1	Augmentation . . . . .	3
<b>2</b>	<b>NLP</b>	<b>5</b>
<b>3</b>	<b>Optimization</b>	<b>7</b>
3.1	Lookahead . . . . .	7
3.2	Learning Rate Schedulers . . . . .	7
<b>4</b>	<b>Metrics</b>	<b>9</b>
<b>5</b>	<b>Utility</b>	<b>11</b>
5.1	pytorch_lightning_spells package . . . . .	11
5.1.1	Submodules . . . . .	13
<b>6</b>	<b>Indices and tables</b>	<b>31</b>
	<b>Python Module Index</b>	<b>33</b>
	<b>Index</b>	<b>35</b>



This package contains some useful plugins for PyTorch Lightning. Many of those are based on others' implementations; I just made some adaptations to make it work with PyTorch Lightning. Please let me know (ceshine at veritable.pw) if you feel any original authors are not credited adequately in my code and documentation.

The following is a categorized list of available classes and functions:



## 1.1 Augmentation

**Warning:** The following three callbacks require `MixupSoftmaxLoss` to be used. The target 1-D tensor will be converted to a 2-D one after the callback. The `MixupSoftmaxLoss` will calculate the correct cross-entropy loss from the 2-D tensor.

- `MixUpCallback`
- `CutMixCallback`
- `SnapMixCallback`

A notebook is available on [Kaggle](#) demonstrating the effect of `MixUp`, `CutMix`, and `SnapMix`.

`RandomAugmentationChoiceCallback` randomly picks one of the given callbacks for each batch. It also supports a no-op warmup period and setting a no-op probability.





The training and inference speed of NLP models can be improved by sorting the input examples by their lengths. This reduces the average number of padding tokens per batch (i.e., the input matrices are smaller). Two samplers are provided to achieve this goal:

- **SortSampler**: Suitable for validation and test datasets, where the order of input examples doesn't matter.
- **SortishSampler**: Suitable for training datasets, where we want to add some randomness in the order of input examples between epochs.



## OPTIMIZATION

- RAdam
- `set_trainable`: a function that freezes or unfreezes a layer group (a `nn.Module` or `nn.ModuleList`).
- `freeze_layers`: a function that freezes or unfreezes **a list of layer groups**.

### 3.1 Lookahead

- `Lookahead`: A PyTorch optimizer wrapper to implement the lookahead mechanism.
- `LookaheadCallback`: A callback that switches the model parameters to the *slow* ones before a validation round starts and switches back to the *fast* ones after it ends.
- `LookaheadModelCheckpoint`: A combination of `LookaheadCallback` and `ModelCheckpoint`, so the *slow* parameters are kept in the checkpoints instead of the fast ones.

### 3.2 Learning Rate Schedulers

- `MultiStageScheduler`: Allows you to combine several schedulers (e.g., linear warmup and cosine decay).
- `LinearLR`: Can be used to achieve both linear warmups and linear decays.
- `ExponentialLR`



## METRICS

PyTorch Lightning did not implement metrics that require the entire dataset to have predictions (e.g., AUC, the Spearman correlation). They do have implemented some of them now in the new [TorchMetrics](#) package.

- [GlobalMetric](#): Extends this class to create new metrics.
- [AUC](#)
- [SpearmanCorrelation](#)
- [FBeta](#)

**Warning:** These metrics require the entire set of labels and predictions to be stored in memory. You might encounter out-of-memory errors if your target tensor is relatively large (e.g., in semantic segmentation tasks) or your validation/test dataset is too large. You'll have to use some approximation techniques in those cases.



## UTILITY

- **BaseModule**: A boilerplate Lightning Module to be extended upon.
- **ScreenLogger**: A logger that prints metrics to the screen.
- **TelegramCallback**: Sent a Telegram message to you when the training starts, ends, and a validation round is finished.
- **EMATracker**: A exponential moving average aggregator.
- **count\_parameters**: A function that returns the total number of parameters in a model.
- **separate\_parameters**: A function that split the parameters of a module into two groups (Batch-Norm/GroupNorm/LayerNorm and others), so you can use weight decay on only one of them.

### 5.1 pytorch\_lightning\_spells package

#### Classes:

---

<i>BaseModule</i> (ema_alpha)	A boilerplate module with some sensible defaults.
-------------------------------	---

---

**class** pytorch\_lightning\_spells.**BaseModule**(ema\_alpha=0.02)

Bases: pytorch\_lightning.core.lightning.LightningModule

A boilerplate module with some sensible defaults.

It logs the exponentially smoothed training losses, and the validation metrics.

You need to implement the *training\_step()* and *validation\_step()* methods. Please refer to the *training\_step\_end()* and *validation\_step\_end()* methods for the expected output format.

**Parameters** **ema\_alpha** (*float*, *optional*) – the weight of the new training loss for the EMA aggregator. Defaults to 0.02.

**Example**

```

>>> class TestModule(BaseModule):
...     def __init__(self):
...         super().__init__(ema_alpha=0.02)
...
...     def training_step(self, batch, batch_idx):
...         return {
...             "loss": torch.tensor(1),
...             "log": batch_idx % self.trainer.accumulate_grad_batches == 0
...         }
...
...     def validation_step(self, batch, batch_idx):
...         return {
...             'loss': torch.tensor(1),
...             'preds': torch.ones_like(batch[1]),
...             'target': batch[1]
...         }
...
>>> module = TestModule()

```

**get\_progress\_bar\_dict()**

Removes *v\_num* from the progress bar.

**test\_step(batch, batch\_idx)**

Simply defer to *.validation\_step()* method.

**test\_step\_end(outputs)**

Basically the same as *.validation\_step\_ends()* method, but with a different prefix.

**training: bool****training\_step\_end(outputs)**

This method logs the training loss for you.

It follows the *log\_every\_n\_steps* attribute of the associated Trainer.

The output from *.validation\_step()* method must contains these two entries:

1. loss: the training loss.
2. log: a boolean value indicating if this is a loggable step.

A loggable step is a step that involves an optimizer step. The opposite is a step that only updates the gradients but not the parameters(e.g., in gradient accumulation).

**Parameters** **outputs** (*Dict*) – the output from *.training\_step()* method.

**validation\_step\_end(outputs)**

This method logs the validation loss and metrics for you.

The output from *.validation\_step()* method must contains these three entries:

1. loss: the validation loss.
2. pred: the predicted labels or values.
3. target: the ground truth lables or values.

**Parameters** **outputs** (*Dict*) – the output from *.validation\_step()* method.



### 5.1.1 Submodules

#### pytorch\_lightning\_spells.callbacks module

##### Classes:

<code>CutMixCallback</code> ([alpha, softmax_target, minmax])	Callback that perform CutMix augmentation on the input batch.
<code>LookaheadCallback</code> ()	Switch to the slow weights before evaluation and switch back after.
<code>LookaheadModelCheckpoint</code> ([dirpath, ...])	Combines LookaheadCallback and ModelCheckpoint
<code>MixUpCallback</code> ([alpha, softmax_target])	Callback that perform MixUp augmentation on the input batch.
<code>RandomAugmentationChoiceCallback</code> (callbacks, p)	Randomly pick an augmentation callback to use for each batch.
<code>SnapMixCallback</code> (model, image_size[, half, ...])	Callback that perform SnapMix augmentation on the input batch.
<code>TelegramCallback</code> (token, chat_id, name[, ...])	A Telegram notification callback

```
class pytorch_lightning_spells.callbacks.CutMixCallback(alpha=0.4, softmax_target=False,
                                                    minmax=None)
```

Bases: `pytorch_lightning.callbacks.base.Callback`

Callback that perform CutMix augmentation on the input batch.

Assumes the first dimension is batch.

Reference: [rwightman/pytorch-image-models/](https://github.com/rwightman/pytorch-image-models/)

##### Parameters

- **alpha** (*float*) –
- **softmax\_target** (*bool*) –
- **minmax** (*Optional[Tuple[float, float]]*) –

**on\_train\_batch\_start**(*trainer, pl\_module, batch, batch\_idx, dataloader\_idx*)

```
class pytorch_lightning_spells.callbacks.LookaheadCallback
```

Bases: `pytorch_lightning.callbacks.base.Callback`

Switch to the slow weights before evaluation and switch back after.

**on\_validation\_end**(*trainer, pl\_module*)

**on\_validation\_start**(*trainer, pl\_module*)

```
class pytorch_lightning_spells.callbacks.LookaheadModelCheckpoint(dirpath=None,  
                                                                filename=None,  
                                                                monitor=None,  
                                                                verbose=False,  
                                                                save_last=None,  
                                                                save_top_k=1,  
                                                                save_weights_only=False,  
                                                                mode='min',  
                                                                auto_insert_metric_name=True,  
                                                                every_n_train_steps=None,  
                                                                train_time_interval=None,  
                                                                every_n_epochs=None,  
                                                                save_on_train_epoch_end=None)
```

Bases: `pytorch_lightning.callbacks.model_checkpoint.ModelCheckpoint`

Combines LookaheadCallback and ModelCheckpoint

#### Parameters

- **dirpath** (*Optional[Union[str, pathlib.Path]]*) –
- **filename** (*Optional[str]*) –
- **monitor** (*Optional[str]*) –
- **verbose** (*bool*) –
- **save\_last** (*Optional[bool]*) –
- **save\_top\_k** (*int*) –
- **save\_weights\_only** (*bool*) –
- **mode** (*str*) –
- **auto\_insert\_metric\_name** (*bool*) –
- **every\_n\_train\_steps** (*Optional[int]*) –
- **train\_time\_interval** (*Optional[datetime.timedelta]*) –
- **every\_n\_epochs** (*Optional[int]*) –
- **save\_on\_train\_epoch\_end** (*Optional[bool]*) –

`on_validation_end(trainer, pl_module)`

`on_validation_start(trainer, pl_module)`

```
class pytorch_lightning_spells.callbacks.MixUpCallback(alpha=0.4, softmax_target=False)
```

Bases: `pytorch_lightning.callbacks.base.Callback`

Callback that perform MixUp augmentation on the input batch.

Assumes the first dimension is batch.

Works best with `pytorch_lightning_spells.losses.MixupSoftmaxLoss`

Reference: [Fast.ai's implementation](#)

#### Parameters

- **alpha** (*float*) –
- **softmax\_target** (*bool*) –

`on_train_batch_start(trainer, pl_module, batch, batch_idx, dataloader_idx)`

```
class pytorch_lightning_spells.callbacks.RandomAugmentationChoiceCallback(callbacks, p,
                                                                           no_op_warmup=0,
                                                                           no_op_prob=0)
```

Bases: `pytorch_lightning.callbacks.base.Callback`

Randomly pick an augmentation callback to use for each batch.

Also supports no-op warmups and no-op probability.

#### Parameters

- **callbacks** (*Sequence[Callback]*) – A sequence of callbacks to choose from.
- **p** (*Sequence[Callback]*) – A sequence of probabilities for the callbacks.
- **no\_op\_warmup** (*int, optional*) – the number of initial steps that should not have any augmentation. Defaults to 0.
- **no\_op\_prob** (*float, optional*) – the probability of a step that has no augmentation. Defaults to 0.

`get_callback()`

`on_train_batch_start(trainer, pl_module, batch, batch_idx, dataloader_idx)`

```
class pytorch_lightning_spells.callbacks.SnapMixCallback(model, image_size, half=False,
                                                         minmax=None, cutmix_bbox=False,
                                                         alpha=0.4, softmax_target=True)
```

Bases: `pytorch_lightning.callbacks.base.Callback`

Callback that perform SnapMix augmentation on the input batch.

Reference: [Shaoli-Huang/SnapMix](#)

#### Warning:

1. Requires the model to have implemented `extract_features` and `get_fc` methods.
2. Can only run in CUDA-enabled environments.

#### Parameters

- **half** (*bool*) –
- **minmax** (*Optional[Tuple[float, float]]*) –
- **cutmix\_bbox** (*bool*) –
- **alpha** (*float*) –
- **softmax\_target** (*bool*) –

`on_train_batch_start(trainer, pl_module, batch, batch_idx, dataloader_idx)`

```
class pytorch_lightning_spells.callbacks.TelegramCallback(token, chat_id, name,
                                                           report_evals=False)
```

Bases: `pytorch_lightning.callbacks.base.Callback`

A Telegram notification callback

Reference: [huggingface/knockknock](https://huggingface.co/knockknock)

**Parameters**

- **token** (*str*) –
- **chat\_id** (*int*) –
- **name** (*str*) –
- **report\_evals** (*bool*) –

`DATE_FORMAT = '%Y-%m-%d %H:%M:%d'`

`on_train_end(trainer, pl_module)`

**Parameters**

- **trainer** (*pytorch\_lightning.trainer.trainer.Trainer*) –
- **pl\_module** (*pytorch\_lightning.core.lightning.LightningModule*) –

`on_train_start(trainer, pl_module)`

**Parameters**

- **trainer** (*pytorch\_lightning.trainer.trainer.Trainer*) –
- **pl\_module** (*pytorch\_lightning.core.lightning.LightningModule*) –

`on_validation_end(trainer, pl_module)`

**Parameters**

- **trainer** (*pytorch\_lightning.trainer.trainer.Trainer*) –
- **pl\_module** (*pytorch\_lightning.core.lightning.LightningModule*) –

`send_message(text)`

**pytorch\_lightning\_spells.cutmix\_utils module**

**Functions:**

<code>cutmix_bbox_and_lam(img_shape, lam[, ...])</code>	Generate bbox and apply lambda correction.
<code>rand_bbox(img_shape, lam[, margin, count])</code>	Standard CutMix bounding-box
<code>rand_bbox_minmax(img_shape, minmax[, count])</code>	Min-Max CutMix bounding-box

`pytorch_lightning_spells.cutmix_utils.cutmix_bbox_and_lam(img_shape, lam, ratio_minmax=None, correct_lam=True, count=None)`

Generate bbox and apply lambda correction.

**Parameters**

- **img\_shape** (*Tuple*) –
- **count** (*Optional[int]*) –

`pytorch_lightning_spells.cutmix_utils.rand_bbox(img_shape, lam, margin=0.0, count=None)`

Standard CutMix bounding-box

Generates a random square bbox based on lambda value. This impl includes support for enforcing a border margin as percent of bbox dimensions.

**Parameters**

- **img\_shape** (*Tuple*) – Image shape as tuple
- **lam** (*float*) – Cutmix lambda value
- **margin** (*float*) – Percentage of bbox dimension to enforce as margin (reduce amount of box outside image)
- **count** (*Optional[int]*) – Number of bbox to generate

`pytorch_lightning_spells.cutmix_utils.rand_bbox_minmax(img_shape, minmax, count=None)`

Min-Max CutMix bounding-box

Inspired by Darknet cutmix impl, generates a random rectangular bbox based on min/max percent values applied to each dimension of the input image. Typical defaults for minmax are usually in the .2-.3 for min and .8-.9 range for max.

**Parameters**

- **img\_shape** (*Tuple*) – Image shape as tuple
- **minmax** (*Union[Tuple, List]*) – Min and max bbox ratios (as percent of image size)
- **count** (*Optional[int]*) – Number of bbox to generate

**pytorch\_lightning\_spells.loggers module****Classes:**


---

`ScreenLogger()`

A logger that prints metrics to the screen.

---

**class pytorch\_lightning\_spells.loggers.ScreenLogger**

Bases: `pytorch_lightning.loggers.base.LightningLoggerBase`

A logger that prints metrics to the screen.

Suitable in situation where you want to check the training progress directly in the console.

**property experiment**

**log\_hyperparams**(*params*)

**log\_metrics**(*metrics, step=None*)

**Parameters**

- **metrics** (*Dict[str, float]*) –
- **step** (*Optional[int]*) –

**Return type** None

**property name**

**property version:** int

## pytorch\_lightning\_spells.losses module

## Classes:

<code>LabelSmoothCrossEntropy(eps)</code>	Cross Entropy with Label Smoothing
<code>MixupSoftmaxLoss([class_weights, reduction, ...])</code>	A softmax loss that supports MixUp augmentation.
<code>Poly1CrossEntropyLoss([epsilon, reduction, ...])</code>	Poly-1 Cross-Entropy Loss
<code>Poly1FocalLoss(num_classes[, epsilon, ...])</code>	Poly-1 Focal Loss

**class** pytorch\_lightning\_spells.losses.**LabelSmoothCrossEntropy**(*eps*)

Bases: torch.nn.modules.module.Module

Cross Entropy with Label Smoothing

Reference: [wangleiofficial/lable-smoothing-pytorch](https://wangleiofficial.github.io/lable-smoothing-pytorch)

The ground truth label will have a value of  $1-eps$  in the target vector.

**Parameters** *eps* (*float*) – the smoothing factor.

**forward**(*preds*, *targets*, *weight=None*)

**training:** **bool**

**class** pytorch\_lightning\_spells.losses.**MixupSoftmaxLoss**(*class\_weights=None*, *reduction='mean'*, *label\_smooth\_eps=0*, *poly1\_eps=0*)

Bases: torch.nn.modules.module.Module

A softmax loss that supports MixUp augmentation.

It requires the input batch to be manipulated into certain format. Works best with MixUpCallback, CutMixCallback, and SnapMixCallback.

Reference: [Fast.ai's implementation](#)

**Parameters**

- **class\_weights** (*torch.Tensor*, *optional*) – The weight of each class. Defaults to the same weight.
- **reduction** (*str*, *optional*) – Loss reduction method. Defaults to ‘mean’.
- **label\_smooth\_eps** (*float*, *optional*) – If larger than zero, use *LabelSmoothed-CrossEntropy* instead of *CrossEntropy*. Defaults to 0.
- **poly1\_eps** (*float*) –

**forward**(*output*, *target*)

The feed-forward.

The target tensor should have three columns:

1. the first class.
2. the second class.
3. the lambda value to mix the above two classes.

**Parameters**

- **output** (*torch.Tensor*) – the model output.
- **target** (*torch.Tensor*) – Shaped (batch\_size, 3).

**Returns** the result loss

**Return type** torch.Tensor

**training:** bool

```
class pytorch_lightning_spells.losses.Poly1CrossEntropyLoss(epsilon=1.0, reduction='none',
                                                         weight=None)
```

Bases: torch.nn.modules.module.Module

Poly-1 Cross-Entropy Loss

Adapted from [abhuse/polyloss-pytorch](#).

Reference: [PolyLoss: A Polynomial Expansion Perspective of Classification Loss Functions](#).

#### Parameters

- **epsilon** (*float*) –
- **reduction** (*str*) –
- **weight** (*Optional[torch.Tensor]*) –

**forward**(*logits, labels, \*\*kwargs*)

Forward pass :param logits: tensor of shape [N, num\_classes] :param labels: tensor of shape [N] :return: poly cross-entropy loss

**training:** bool

```
class pytorch_lightning_spells.losses.Poly1FocalLoss(num_classes, epsilon=1.0, alpha=0.25,
                                                    gamma=2.0, reduction='none', weight=None,
                                                    label_is_onehot=False)
```

Bases: torch.nn.modules.module.Module

Poly-1 Focal Loss

Adapted from [abhuse/polyloss-pytorch](#).

Reference: [PolyLoss: A Polynomial Expansion Perspective of Classification Loss Functions](#).

#### Parameters

- **num\_classes** (*int*) –
- **epsilon** (*float*) –
- **alpha** (*float*) –
- **gamma** (*float*) –
- **reduction** (*str*) –
- **weight** (*torch.Tensor*) –
- **label\_is\_onehot** (*bool*) –

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

Forward pass :param logits: output of neural network of shape [N, num\_classes] or [N, num\_classes, ...]  
:param labels: ground truth tensor of shape [N] or [N, ...] with class ids if label\_is\_onehot was set to False,  
otherwise

one-hot encoded tensor of same shape as logits

**Returns** poly focal loss

**training:** bool

## pytorch\_lightning\_spells.lr\_schedulers module

### Classes:

---

*BaseLRScheduler*(optimizer[, last\_epoch, verbose])

---

*CosineAnnealingScheduler*(optimizer, T\_max[, ...])

---

*ExponentialLR*(optimizer, min\_lr\_ratio, ...) Exponentially increases the learning rate between two boundaries over a number of iterations.

---

*LinearLR*(optimizer, min\_lr\_ratio, total\_epochs) Linearly increases or decrease the learning rate between two boundaries over a number of iterations.

---

*MultiStageScheduler*(schedulers, start\_at\_epochs)

---

**class** pytorch\_lightning\_spells.lr\_schedulers.**BaseLRScheduler**(*optimizer, last\_epoch=- 1, verbose=False*)

Bases: torch.optim.lr\_scheduler.\_LRScheduler

**clear\_optimizer**()

**switch\_optimizer**(*optimizer*)

**class** pytorch\_lightning\_spells.lr\_schedulers.**CosineAnnealingScheduler**(*optimizer, T\_max, eta\_min=0, last\_epoch=- 1, verbose=False*)

Bases: torch.optim.lr\_scheduler.CosineAnnealingLR, *pytorch\_lightning\_spells.lr\_schedulers.BaseLRScheduler*

**class** pytorch\_lightning\_spells.lr\_schedulers.**ExponentialLR**(*optimizer, min\_lr\_ratio, total\_epochs, last\_epoch=- 1*)

Bases: *pytorch\_lightning\_spells.lr\_schedulers.BaseLRScheduler*

Exponentially increases the learning rate between two boundaries over a number of iterations.

Mainly used by LR finders.

**\_\_init\_\_**(*optimizer, min\_lr\_ratio, total\_epochs, last\_epoch=- 1*)

Initialize a scheduler.

#### Parameters

- **optimizer** (*Union[torch.optim.Optimizer, apex.fp16\_utils.fp16\_optimizer.FP16\_Optimizer]*) –
- **min\_lr\_ratio** (*float*) – min\_lr\_ratio \* base\_lr will be the starting learning rate.
- **total\_epochs** (*int*) – the total number of “steps” in this run.
- **last\_epoch** (*int, optional*) – the index of last epoch, by default -1.

**get\_lr**()



**class** `pytorch_lightning_spells.lr_schedulers.LinearLR`(*optimizer, min\_lr\_ratio, total\_epochs, upward=True, last\_epoch=- 1*)

Bases: `pytorch_lightning_spells.lr_schedulers.BaseLRScheduler`

Linearly increases or decrease the learning rate between two boundaries over a number of iterations.

#### Parameters

- **optimizer** (`torch.optim.optimizer.Optimizer`) –
- **min\_lr\_ratio** (`float`) –
- **total\_epochs** (`float`) –
- **upward** (`bool`) –
- **last\_epoch** (`int`) –

**\_\_init\_\_**(*optimizer, min\_lr\_ratio, total\_epochs, upward=True, last\_epoch=- 1*)

Initialize a scheduler.

#### Parameters

- **optimizer** (`Union[torch.optim.Optimizer, apex.fp16_utils.fp16_optimizer.FP16_Optimizer]`) –
- **min\_lr\_ratio** (`float`) – `min_lr_ratio * base_lr` will be the starting learning rate.
- **total\_epochs** (`float`) – the total number of “steps” in this run.
- **upward** (`bool`) – whether the learning rate goes up or down. Defaults to True.
- **last\_epoch** (`int`) – the index of last epoch. Defaults to -1.

**get\_lr**()

**class** `pytorch_lightning_spells.lr_schedulers.MultiStageScheduler`(*schedulers, start\_at\_epochs, last\_epoch=- 1*)

Bases: `torch.optim.lr_scheduler._LRScheduler`

#### Parameters

- **schedulers** (`Sequence`) –
- **start\_at\_epochs** (`Sequence[int]`) –
- **last\_epoch** (`int`) –

**\_\_init\_\_**(*schedulers, start\_at\_epochs, last\_epoch=- 1*)

#### Parameters

- **schedulers** (`Sequence`) –
- **start\_at\_epochs** (`Sequence[int]`) –
- **last\_epoch** (`int`) –

**clear\_optimizer**()

**load\_state\_dict**(*state\_dict*)

Loads the schedulers state.

**Parameters** **state\_dict** (`dict`) – scheduler state. Should be an object returned from a call to `state_dict()`.

**state\_dict()**

Returns the state of the scheduler as a dict.

It contains an entry for every variable in self.\_\_dict\_\_ which is not the optimizer.

**step(*epoch=None*)**

**switch\_optimizer(*optimizer*)**

**pytorch\_lightning\_spells.metrics module**

**Classes:**

<code>AUC</code> ([compute_on_step, dist_sync_on_step, ...])	AUC: Area Under the ROC Curve
<code>FBeta</code> ([step, beta, compute_on_step, ...])	The F-beta score is the weighted harmonic mean of precision and recall
<code>GlobalMetric</code> ([compute_on_step, ...])	
<code>SpearmanCorrelation</code> ([sigmoid, ...])	

**class** pytorch\_lightning\_spells.metrics.**AUC**(*compute\_on\_step=False, dist\_sync\_on\_step=False, process\_group=None*)

Bases: `pytorch_lightning_spells.metrics.GlobalMetric`

AUC: Area Under the ROC Curve

**Binary mode**

```
>>> auc = AUC()
>>> auc(torch.tensor([0.3, 0.8, 0.2]), torch.tensor([0, 1, 0]))
>>> auc(torch.tensor([0.3, 0.3, 0.9]), torch.tensor([1, 1, 0]))
>>> round(auc.compute().item(), 2)
0.56
```

**Multi-class mode**

This will use the first column as the negative case, and the rest collectively as the positive case.

```
>>> auc = AUC()
>>> auc(torch.tensor([[0.3, 0.8, 0.2], [0.2, 0.1, 0.1], [0.5, 0.1, 0.7]]).t(),
↳ torch.tensor([0, 1, 0]))
>>> auc(torch.tensor([[0.3, 0.3, 0.8], [0.2, 0.6, 0.1], [0.5, 0.1, 0.1]]).t(),
↳ torch.tensor([1, 1, 0]))
>>> round(auc.compute().item(), 2)
0.39
```

**Parameters**

- **compute\_on\_step** (*bool*) –
- **dist\_sync\_on\_step** (*bool*) –
- **process\_group** (*Optional[Any]*) –

`compute()`

```
class pytorch_lightning_spells.metrics.FBeta(step=0.02, beta=2, compute_on_step=False,
                                             dist_sync_on_step=False, process_group=None)
```

Bases: `pytorch_lightning_spells.metrics.GlobalMetric`

The F-beta score is the weighted harmonic mean of precision and recall

#### Binary mode

```
>>> fbeta = FBeta()
>>> fbeta(torch.tensor([0.3, 0.8, 0.2]), torch.tensor([0, 1, 0]))
>>> fbeta(torch.tensor([0.3, 0.3, 0.9]), torch.tensor([1, 1, 0]))
>>> round(fbeta.compute().item(), 2)
0.88
```

#### Multi-class mode

This will use the first column as the negative case, and the rest collectively as the positive case.

```
>>> fbeta = FBeta()
>>> fbeta(torch.tensor([[0.8, 0.3, 0.7], [0.1, 0.1, 0.1], [0.1, 0.6, 0.2]]).t(),
          torch.tensor([0, 1, 0]))
>>> fbeta(torch.tensor([[0.3, 0.7, 0.8], [0.2, 0.2, 0.1], [0.5, 0.1, 0.1]]).t(),
          torch.tensor([1, 1, 0]))
>>> round(fbeta.compute().item(), 4)
0.9375
```

#### Parameters

- **step** (*float*) –
- **beta** (*int*) –
- **compute\_on\_step** (*bool*) –
- **dist\_sync\_on\_step** (*bool*) –
- **process\_group** (*Optional[Any]*) –

`compute()`

`find_best_fbeta_threshold(truth, probs)`

```
class pytorch_lightning_spells.metrics.GlobalMetric(compute_on_step=False,
                                                    dist_sync_on_step=False,
                                                    process_group=None)
```

Bases: `torchmetrics.metric.Metric`

#### Parameters

- **compute\_on\_step** (*bool*) –
- **dist\_sync\_on\_step** (*bool*) –
- **process\_group** (*Optional[Any]*) –

`update(preds, target)`

Update state with predictions and targets.

#### Parameters

- **preds** (*torch.Tensor*) – Predictions from model
- **target** (*torch.Tensor*) – Ground truth values

```
class pytorch_lightning_spells.metrics.SpearmanCorrelation(sigmoid=False,
                                                         compute_on_step=False,
                                                         dist_sync_on_step=False,
                                                         process_group=None)
```

Bases: *pytorch\_lightning\_spells.metrics.GlobalMetric*

**Parameters**

- **sigmoid** (*bool*) –
- **compute\_on\_step** (*bool*) –
- **dist\_sync\_on\_step** (*bool*) –
- **process\_group** (*Optional[Any]*) –

**compute()**

**pytorch\_lightning\_spells.optimizers module**

Classes:

<i>Lookahead</i> (optimizer[, alpha, k, ...])	Lookahead Wrapper
<i>RAdam</i> (params[, lr, betas, eps, ...])	RAdam optimizer, a theoretically sound variant of Adam.

```
class pytorch_lightning_spells.optimizers.Lookahead(optimizer, alpha=0.5, k=6,
                                                    pullback_momentum='none')
```

Bases: *torch.optim.optimizer.Optimizer*

Lookahead Wrapper

- Code: [lonePatient/lookahead\\_pytorch](#)
- Paper: [Lookahead Optimizer](#)

Works best with *LookaheadCallback* or *LookaheadModelCheckpoint*.

**Parameters**

- **optimizer** (*Optimizer*) – The inner optimizer.
- **alpha** (*float, optional*) – The linear interpolation factor. 1.0 recovers the inner optimizer. Defaults to 0.5.
- **k** (*int, optional*) – The number of lookahead steps. Defaults to 6.
- **pullback\_momentum** (*str, optional*) – Change to inner optimizer momentum on interpolation update. Defaults to “none”.

---

**Note:** Currently *pullback\_momentum* only supports SGD optimizers with momentum.

---

**Raises ValueError** – Invalid slow update rate or invalid lookahead steps

**Parameters**

- **optimizer** (*torch.optim.optimizer.Optimizer*) –
- **alpha** (*float*) –
- **k** (*int*) –
- **pullback\_momentum** (*str*) –

### Example

```
>>> model = torch.nn.Linear(10, 1)
>>> optimizer = Lookahead(
...     torch.optim.SGD(model.parameters(), momentum=0.9, lr=0.1),
...     alpha=0.5, k=6, pullback_momentum="pullback")
...
>>> for _ in range(10):
...     optimizer.zero_grad()
...     loss = model(torch.rand(10))
...     loss.backward()
...     optimizer.step()
...
...

```

**load\_state\_dict**(*state\_dict*)

**state\_dict**()

**step**(*closure=None*)

Performs a single Lookahead optimization step.

**zero\_grad**()

**class** `pytorch_lightning_spells.optimizers.RAdam`(*params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight\_decay=0, degenerated\_to\_sgd=True*)

Bases: `torch.optim.optimizer.Optimizer`

RAdam optimizer, a theoretically sound variant of Adam.

Source: [LiyuanLucasLiu/RAdam](#)

Under Apache License 2.0

**step**(*closure=None*)

### `pytorch_lightning_spells.samplers` module

#### Classes:

<code>SortSampler</code> ( <i>data_source, key</i> )	Go through the text data by order of length (longest to shortest).
<code>SortishSampler</code> ( <i>data_source, key, bs[, ...]</i> )	Go through the text data by order of length with a bit of randomness.

**class** `pytorch_lightning_spells.samplers.SortSampler`(*data\_source*, *key*)

Bases: `torch.utils.data.sampler.Sampler`

Go through the text data by order of length (longest to shortest).

Taken from [Fast.ai](#) library.

### Parameters

- **data\_source** (*Iterable*) – The data you want to sample from.
- **key** (*Callable*) – A function to get keys to sort. Input: the index number of the entry in `data_source`.

### Example

```
>>> data_source = [[0], [0, 1], [0, 1, 2, 3]]
>>> sampler = SortSampler(data_source, key=lambda idx: len(data_source[idx]))
>>> len(list(sampler))
3
>>> next(iter(sampler)) # the longest entry is the third one.
2
```

**class** `pytorch_lightning_spells.samplers.SortishSampler`(*data\_source*, *key*, *bs*, *chunk\_size=100*)

Bases: `torch.utils.data.sampler.Sampler`

Go through the text data by order of length with a bit of randomness.

Returns an iterator that traverses the the data in randomly ordered batches that are approximately the same size.

The data is first randomly shuffled and then put into a number of chunks. The data in each chunk is then sorted and sliced to get batches that are approximately the same size.

The max key size batch is always returned in the first call because of pytorch cuda memory allocation sequencing.

Without that max key returned first multiple buffers may be allocated when the first created isn't large enough to hold the next in the sequence.

Taken from [Fast.ai](#).

### Parameters

- **data\_source** (*Iterable*) – The data you want to sample from.
- **key** (*Callable*) – A function to get keys to sort. Input: the index number of the entry in `data_source`.
- **bs** (*int*) – the batch size for the data loader
- **chunk\_size** (*int*, *optional*) – the number of batches one chunk contains. Defaults to 100.

**Example**

```

>>> data_source = [[0], [0, 1]] * 100
>>> sampler = SortishSampler(data_source, key=lambda idx: len(data_source[idx]),
↳ bs=2, chunk_size=2)
>>> len(list(sampler))
200
>>> len(data_source[next(iter(sampler))]) # the largest/longest batch always goes
↳ first
2

```

**pytorch\_lightning\_spells.snapmix\_utils module**

SnapMix Utility Functions

Reference: Shaoli-Huang/SnapMix

**Functions:**


---

*get\_spm*(input\_tensor, target, model, image\_size)

---

pytorch\_lightning\_spells.snapmix\_utils.get\_spm(input\_tensor, target, model, image\_size, half=False)

**Parameters** *half* (*bool*) –

**pytorch\_lightning\_spells.utils module****Classes:**


---

*EMATracker*([alpha]) Keeps the exponential moving average for a single series.

---

**Functions:**


---

<i>count_parameters</i> (parameters)	Count the number of parameters
<i>freeze_layers</i> (layer_groups, freeze_flags)	Freeze or unfreeze groups of layers
<i>separate_parameters</i> (module[, skip_list])	Separate BatchNorm2d, GroupNorm, and LayerNorm parameters from others
<i>set_trainable</i> (layer, trainable)	Freeze or unfreeze all parameters in the layer.

---

**class** pytorch\_lightning\_spells.utils.**EMATracker**(*alpha=0.05*)

Bases: object

Keeps the exponential moving average for a single series.

**Parameters** *alpha* (*float*, *optional*) – the weight of the new value, by default 0.05

## Examples

```
>>> tracker = EMATracker(0.1)
>>> tracker.update(1.)
>>> tracker.value
1.0
>>> tracker.update(2.)
>>> tracker.value # 1 * 0.9 + 2 * 0.1
1.1
>>> tracker.update(float('nan')) # this won't have any effect
>>> tracker.value
1.1
```

### **update**(*new\_value*)

Adds a new value to the tracker.

It will ignore NaNs and raise a warning in those cases.

**Parameters** *new\_value* (*Union*[*float*, *torch.Tensor*]) – the incoming value.

### **property value**

The smoothed value.

`pytorch_lightning_spells.utils.count_parameters(parameters)`

Count the number of parameters

**Parameters** *parameters* (*Iterable*[*Union*[*torch.Tensor*, *Parameter*]]) – parameters you want to count.

**Returns** the number of parameters counted.

**Return type** `int`

## Example

```
>>> count_parameters([torch.rand(100), torch.rand(10)])
110
>>> count_parameters([torch.rand(100, 2), torch.rand(10, 3)])
230
```

`pytorch_lightning_spells.utils.freeze_layers(layer_groups, freeze_flags)`

Freeze or unfreeze groups of layers

### **Parameters**

- **layer\_groups** (*Sequence*[*Layer*]) – the target lists of layers
- **freeze\_flags** (*Sequence*[*bool*]) – the corresponding trainable flags

**Warning:** The value in *freeze\_flag* has the opposite meaning as in *trainable* of *set\_trainable*.

Set True to freeze; False to unfreeze.



## Examples

```

>>> model = nn.Sequential(nn.Linear(10, 100), nn.Linear(100, 1))
>>> freeze_layers([model[0], model[1]], [True, False])
>>> model[0].weight.requires_grad
False
>>> model[1].weight.requires_grad
True
>>> freeze_layers([model[0], model[1]], [False, True])
>>> model[0].weight.requires_grad
True
>>> model[1].weight.requires_grad
False

```

`pytorch_lightning_spells.utils.separate_parameters(module, skip_list=('bias',))`

Separate BatchNorm2d, GroupNorm, and LayerNorm parameters from others

### Parameters

- **module** (`Union[Parameter, nn.Module, List[nn.Module]]`) – to be separated.
- **skip\_list** (`Sequence[str]`) –

**Returns** lists of decay and no-decay parameters.

**Return type** `Tuple[List[Parameter], List[Parameter]]`

## Example

```

>>> model = nn.Sequential(nn.Linear(100, 10, bias=True), nn.BatchNorm1d(10))
>>> _ = nn.init.constant_(model[0].weight, 2.)
>>> _ = nn.init.constant_(model[0].bias, 1.)
>>> _ = nn.init.constant_(model[1].weight, 1.)
>>> _ = nn.init.constant_(model[1].bias, 1.)
>>> model[0].weight.data.sum().item()
2000.0
>>> model[0].bias.data.sum().item()
10.0
>>> model[1].weight.data.sum().item()
10.0
>>> model[1].bias.data.sum().item()
10.0
>>> decay, no_decay = separate_parameters(model) # separate the parameters
>>> np.sum([x.sum().detach().numpy() for x in decay]) # nn.Linear
2000.0
>>> np.sum([x.sum().detach().numpy() for x in no_decay]) # nn.BatchNorm1d
30.0
>>> optimizer = torch.optim.AdamW([
...     "params": decay, "weight_decay": 0.1
... }, {
...     "params": no_decay, "weight_decay": 0
... }], lr = 1e-3)

```

`pytorch_lightning_spells.utils.set_trainable(layer, trainable)`

Freeze or unfreeze all parameters in the layer.

### Parameters

- **layer** (*Union[torch.nn.Module, torch.nn.ModuleList]*) – the target layer
- **trainable** (*bool*) – True to unfreeze; False to freeze

### Example

```
>>> model = nn.Sequential(nn.Linear(10, 100), nn.Linear(100, 1))
>>> model[0].weight.requires_grad
True
>>> set_trainable(model, False)
>>> model[0].weight.requires_grad
False
>>> set_trainable(model, True)
>>> model[0].weight.requires_grad
True
```

## INDICES AND TABLES

- modindex
- search
- genindex



## PYTHON MODULE INDEX

### p

- `pytorch_lightning_spells`, 11
- `pytorch_lightning_spells.callbacks`, 13
- `pytorch_lightning_spells.cutmix_utils`, 16
- `pytorch_lightning_spells.loggers`, 17
- `pytorch_lightning_spells.losses`, 18
- `pytorch_lightning_spells.lr_schedulers`, 20
- `pytorch_lightning_spells.metrics`, 22
- `pytorch_lightning_spells.optimizers`, 24
- `pytorch_lightning_spells.samplers`, 25
- `pytorch_lightning_spells.snapmix_utils`, 27
- `pytorch_lightning_spells.utils`, 27



## Symbols

`__init__()` (*pytorch\_lightning\_spells.lr\_schedulers.ExponentialLR* method), 20  
`__init__()` (*pytorch\_lightning\_spells.lr\_schedulers.LinearLR* method), 21  
`__init__()` (*pytorch\_lightning\_spells.lr\_schedulers.MultiStageScheduler* method), 21

## A

AUC (class in *pytorch\_lightning\_spells.metrics*), 22

## B

BaseLRScheduler (class in *pytorch\_lightning\_spells.lr\_schedulers*), 20  
 BaseModule (class in *pytorch\_lightning\_spells*), 11

## C

`clear_optimizer()` (*pytorch\_lightning\_spells.lr\_schedulers.BaseLRScheduler* method), 20  
`clear_optimizer()` (*pytorch\_lightning\_spells.lr\_schedulers.MultiStageScheduler* method), 21  
`compute()` (*pytorch\_lightning\_spells.metrics.AUC* method), 22  
`compute()` (*pytorch\_lightning\_spells.metrics.FBeta* method), 23  
`compute()` (*pytorch\_lightning\_spells.metrics.SpearmanCorrelation* method), 24  
 CosineAnnealingScheduler (class in *pytorch\_lightning\_spells.lr\_schedulers*), 20  
`count_parameters()` (in module *pytorch\_lightning\_spells.utils*), 28  
`cutmix_bbox_and_lam()` (in module *pytorch\_lightning\_spells.cutmix\_utils*), 16  
 CutMixCallback (class in *pytorch\_lightning\_spells.callbacks*), 13

## D

DATE\_FORMAT (*pytorch\_lightning\_spells.callbacks.TelegramCallback* attribute), 16

## E

EMAHooker (class in *pytorch\_lightning\_spells.utils*), 27  
 experiment (*pytorch\_lightning\_spells.loggers.ScreenLogger* property), 17  
 ExponentialLR (class in *pytorch\_lightning\_spells.lr\_schedulers*), 20

## F

FBeta (class in *pytorch\_lightning\_spells.metrics*), 23  
`find_best_fbeta_threshold()` (*pytorch\_lightning\_spells.metrics.FBeta* method), 23  
`forward()` (*pytorch\_lightning\_spells.losses.LabelSmoothCrossEntropy* method), 18  
`forward()` (*pytorch\_lightning\_spells.losses.MixupSoftmaxLoss* method), 18  
`forward()` (*pytorch\_lightning\_spells.losses.PolyJCrossEntropyLoss* method), 19  
`forward()` (*pytorch\_lightning\_spells.losses.PolyJFocalLoss* method), 19  
`freeze_layers()` (in module *pytorch\_lightning\_spells.utils*), 28

## G

`get_callback()` (*pytorch\_lightning\_spells.callbacks.RandomAugmentation* method), 15  
`get_lr()` (*pytorch\_lightning\_spells.lr\_schedulers.ExponentialLR* method), 20  
`get_lr()` (*pytorch\_lightning\_spells.lr\_schedulers.LinearLR* method), 21  
`get_progress_bar_dict()` (*pytorch\_lightning\_spells.BaseModule* method), 12  
`get_spm()` (in module *pytorch\_lightning\_spells.snapmix\_utils*), 27  
 GlobalMetric (class in *pytorch\_lightning\_spells.metrics*), 23

## L

LabelSmoothCrossEntropy (class in *pytorch\_lightning\_spells.losses*), 18





## R

RAdam (class in `pytorch_lightning_spells.optimizers`), 25  
 rand\_bbox() (in module `pytorch_lightning_spells.cutmix_utils`), 16  
 rand\_bbox\_minmax() (in module `pytorch_lightning_spells.cutmix_utils`), 17  
 RandomAugmentationChoiceCallback (class in `pytorch_lightning_spells.callbacks`), 15

## S

ScreenLogger (class in `pytorch_lightning_spells.loggers`), 17  
 send\_message() (`pytorch_lightning_spells.callbacks.TelegramCallback` method), 16  
 separate\_parameters() (in module `pytorch_lightning_spells.utils`), 29  
 set\_trainable() (in module `pytorch_lightning_spells.utils`), 29  
 SnapMixCallback (class in `pytorch_lightning_spells.callbacks`), 15  
 SortishSampler (class in `pytorch_lightning_spells.samplers`), 26  
 SortSampler (class in `pytorch_lightning_spells.samplers`), 25  
 SpearmanCorrelation (class in `pytorch_lightning_spells.metrics`), 24  
 state\_dict() (`pytorch_lightning_spells.lr_schedulers.MultiStageScheduler` method), 21  
 state\_dict() (`pytorch_lightning_spells.optimizers.Lookahead` method), 25  
 step() (`pytorch_lightning_spells.lr_schedulers.MultiStageScheduler` method), 22  
 step() (`pytorch_lightning_spells.optimizers.Lookahead` method), 25  
 step() (`pytorch_lightning_spells.optimizers.RAdam` method), 25  
 switch\_optimizer() (`pytorch_lightning_spells.lr_schedulers.BaseLRScheduler` method), 20  
 switch\_optimizer() (`pytorch_lightning_spells.lr_schedulers.MultiStageScheduler` method), 22

## T

TelegramCallback (class in `pytorch_lightning_spells.callbacks`), 15  
 test\_step() (`pytorch_lightning_spells.BaseModule` method), 12  
 test\_step\_end() (`pytorch_lightning_spells.BaseModule` method), 12  
 training (`pytorch_lightning_spells.BaseModule` attribute), 12

training (`pytorch_lightning_spells.losses.LabelSmoothCrossEntropy` attribute), 18  
 training (`pytorch_lightning_spells.losses.MixupSoftmaxLoss` attribute), 19  
 training (`pytorch_lightning_spells.losses.Poly1CrossEntropyLoss` attribute), 19  
 training (`pytorch_lightning_spells.losses.Poly1FocalLoss` attribute), 20  
 training\_step\_end() (`pytorch_lightning_spells.BaseModule` method), 12

## U

update() (`pytorch_lightning_spells.metrics.GlobalMetric` method), 23  
 update() (`pytorch_lightning_spells.utils.EMATracker` method), 28

## V

validation\_step\_end() (`pytorch_lightning_spells.BaseModule` method), 12  
 value (`pytorch_lightning_spells.utils.EMATracker` property), 28  
 version (`pytorch_lightning_spells.loggers.ScreenLogger` property), 17

## Z

zero\_grad() (`pytorch_lightning_spells.optimizers.Lookahead` method), 25