PyTorch Lightning Spells

Release 0.1.1

Ceshine Lee

Feb 27, 2024

CONTENTS:

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:

CHAPTER

ONE

CV

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.

CHAPTER

TWO

NLP

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.

CHAPTER

THREE

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

CHAPTER

FOUR

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.

CHAPTER

FIVE

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 (BatchNorm/GroupNorm/LayerNorm and others), so you can use weight decay on only one of them.

5.1 pytorch_lightning_spells package

Classes:

BaseModule([ema_alpha])	A boilerplate module with some sensible defaults.

class pytorch_lightning_spells.**BaseModule**(*ema_alpha=0.02*)

Bases: 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_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:

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

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

Bases: Callback

Callback that perform CutMix augmentation on the input batch.

Assumes the first dimension is batch.

Reference: rwightman/pytorch-image-models/

Parameters

- **alpha** (*float*) –
- **softmax_target** (*bool*) –
- **minmax** (*Tuple[float, float] | None*) –

on_train_batch_start(*trainer*, *pl_module*, *batch*, *batch_idx*)

class pytorch_lightning_spells.callbacks.**LookaheadCallback**

Bases: 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*, *enable_version_counter=True*)

Bases: ModelCheckpoint

Combines LookaheadCallback and ModelCheckpoint

Parameters

- **dirpath** (*str | Path | None*) –
- **filename** (*str | None*) –
- **monitor** (*str | None*) –
- **verbose** (*bool*) –
- **save_last** (*bool | None*) –
- **save_top_k** (*int*) –
- **save_weights_only** (*bool*) –
- **mode** (*str*) –
- **auto_insert_metric_name** (*bool*) –
- **every_n_train_steps** (*int | None*) –
- **train_time_interval** (*timedelta | None*) –
- **every_n_epochs** (*int | None*) –
- **save_on_train_epoch_end** (*bool | None*) –
- **enable_version_counter** (*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: 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*)

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

Bases: 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 calbacks 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*)

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

Bases: 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** (*Tuple[float, float] | None*) –
- **cutmix_bbox** (*bool*) –
- **alpha** (*float*) –
- **softmax_target** (*bool*) –

on_train_batch_start(*trainer*, *pl_module*, *batch*, *batch_idx*)

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

Bases: Callback

A Telegram notification callback

Reference: huggingface/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** (*Trainer*) –
- **pl_module** (*LightningModule*) –

on_train_start(*trainer*, *pl_module*)

Parameters

- **trainer** (*Trainer*) –
- **pl_module** (*LightningModule*) –

on_validation_end(*trainer*, *pl_module*)

Parameters

- **trainer** (*Trainer*) –
- **pl_module** (*LightningModule*) –

send_message(*text*)

pytorch_lightning_spells.cutmix_utils module

Functions:

cutmix_bbox_and_lam(img_shape, lam[, ...])	Generate bbox and apply lambda correction.
rand_bbox(img_shape, lam[, margin, count])	Standard CutMix bounding-box
rand_bbox_minmax(img_shape, minmax[, count])	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** (*int | None*) –

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** (*int | None*) – 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** (*Tuple | List*) – Min and max bbox ratios (as percent of image size)
- **count** (*int | None*) – 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: Logger

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** (*int | None*) –

Return type
None

property name

property version: int

pytorch_lightning_spells.losses module

Classes:

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

class pytorch_lightning_spells.losses.**LabelSmoothCrossEntropy**(*eps*)

Bases: Module

Cross Entropy with Label Smoothing

Reference: wangleiofficial/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*)

class pytorch_lightning_spells.losses.**MixupSoftmaxLoss**(*class_weights=None*, *reduction='mean'*, *label_smooth_eps=0*, *poly1_eps=0*)

Bases: 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 *LabelSmoothedCrossEntropy* 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

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

Bases: 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** (*Tensor | None*) –

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

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: 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** (*Tensor*) –
- **label_is_onehot** (*bool*) –

forward(*logits*, *labels*)

Forward pass :param logits: output of neural netwrok 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

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='deprecated'*)

Bases: _LRScheduler

clear_optimizer()

switch_optimizer(*optimizer*)

class pytorch_lightning_spells.lr_schedulers.**CosineAnnealingScheduler**(*optimizer*, *T_max*, *eta_min=0*, *last_epoch=-1*, *verbose='deprecated'*)

Bases: CosineAnnealingLR, *BaseLRScheduler*

class pytorch_lightning_spells.lr_schedulers.**ExponentialLR**(*optimizer*, *min_lr_ratio*, *total_epochs*, *last_epoch=-1*)

Bases: *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: *BaseLRScheduler*

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

Parameters

- **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: _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:

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

class pytorch_lightning_spells.metrics.**AUC**(*compute_on_step=False*, *dist_sync_on_step=False*, *process_group=None*)

Bases: *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** (*Any | None*) –

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: *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** (*Any | None*) –

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: Metric

Parameters

- **compute_on_step** (*bool*) –
- **dist_sync_on_step** (*bool*) –

- **process_group** (*Any | None*) –

update(*preds*, *target*)

Update state with predictions and targets.

Parameters

- **preds** (*Tensor*) – Predictions from model
- **target** (*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: *GlobalMetric*

Parameters

- **sigmoid** (*bool*) –
- **compute_on_step** (*bool*) –
- **dist_sync_on_step** (*bool*) –
- **process_group** (*Any | None*) –

compute()

pytorch_lightning_spells.optimizers module

Classes:

Lookahead(optimizer[, alpha, k, ...])	Lookahead Wrapper
RAdam(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: 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** (*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: 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:

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

class pytorch_lightning_spells.samplers.**SortSampler**(*data_source*, *key*)

Bases: 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: 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:

count_parameters(parameters)	Count the number of parameters
freeze_layers(layer_groups, freeze_flags)	Freeze or unfreeze groups of layers
separate_parameters(module[, skip_list])	Separate BatchNorm2d, GroupNorm, and LayerNorm paremeters from others
set_trainable(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 paremeters 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
```

CHAPTER

SIX

INDICES AND TABLES

- modindex
- search
- genindex

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

L

M

N

O

P

R

S

T

U

V

Z