pytorch-accelerated

Release 0.1.3

Chris Hughes

Jul 05, 2023

GET STARTED:

CHAPTER

ONE

WHAT IS *PYTORCH-ACCELERATED*?

pytorch-accelerated is a lightweight library designed to accelerate the process of training PyTorch models by providing a minimal, but extensible training loop - encapsulated in a single `Trainer` object - which is flexible enough to handle the majority of use cases, and capable of utilizing different hardware options with no code changes required.

pytorch-accelerated offers a streamlined feature set, and places a huge emphasis on **simplicity** and **transparency**, to enable users to understand exactly what is going on under the hood, but without having to write and maintain the boilerplate themselves!

CHAPTER

TWO

KEY FEATURES

- A simple and contained, but easily customisable, training loop, which should work out of the box in straightforward cases; behaviour can be customised using inheritance and/or callbacks.
- Handles device placement, mixed-precision, DeepSpeed integration, multi-GPU and distributed training with no code changes.
- Uses pure PyTorch components, with no additional modifications or wrappers, and easily interoperates with other popular libraries such as timm, transformers and torchmetrics.
- A small, streamlined API ensures that there is a minimal learning curve for existing PyTorch users.

Significant effort has been taken to ensure that every part of the library - both internal and external components - is as clear and simple as possible, making it easy to customise, debug and understand exactly what is going on behind the scenes at each step; most of the behaviour of the trainer is contained in a single class!

In the spirit of Python, nothing is hidden and everything is accessible.

pytorch-accelerated is proudly and transparently built on top of Hugging Face's accelerate, which is responsible for the movement of data between devices and launching of training configurations. When customizing the trainer, or launching training, users are encouraged to consult the Accelerate documentation to understand all available options; Accelerate provides convenient functions for operations such gathering tensors, usage of which can be seen in the *pytorch-accelerated* examples folder!

To learn more about the motivations behind this library, along with a detailed getting started guide, check out this blog post.

2.1 Who is pytorch-accelerated aimed at?

- Users that are familiar with PyTorch but would like to avoid having to write the common training loop boilerplate to focus on the interesting parts of the training loop.
- Users who like, and are comfortable with, selecting and creating their own models, loss functions, optimizers and datasets.
- Users who value a simple and streamlined feature set, where the behaviour is easy to debug, understand, and reason about!

2.2 When shouldn't I use pytorch-accelerated?

- If you are looking for an end-to-end solution, encompassing everything from loading data to inference, which helps you to select a model, optimizer or loss function, you would probably be better suited to fastai. *pytorch-accelerated* focuses only on the training process, with all other concerns being left to the responsibility of the user.
- If you would like to write the entire training loop yourself, just without all of the device management headaches, you would probably be best suited to using accelerate directly! Whilst it is possible to customize every part of the `Trainer`, the training loop is fundamentally broken up into a number of different methods that you would have to override. But, before you go, is writing those *for* loops really important enough to warrant starting from scratch *again* .
- If you are working on a custom, highly complex, use case which does not fit the patterns of usual training loops and want to squeeze out every last bit of performance on your chosen hardware, you are probably best off sticking with vanilla PyTorch; any high-level API becomes an overhead in highly specialized cases!

CHAPTER

THREE

ACKNOWLEDGEMENTS

Many aspects behind the design and features of *pytorch-accelerated* were greatly inspired by a number of excellent libraries and frameworks such as fastai, timm, PyTorch-lightning and Hugging Face accelerate. Each of these tools have made an enormous impact on both this library and the machine learning community, and their influence can not be stated enough!

pytorch-accelerated has taken only inspiration from these tools, and all of the functionality contained has been implemented from scratch in a way that benefits this library. The only exceptions to this are some of the scripts in the examples folder in which existing resources were taken and modified in order to showcase the features of *pytorch-accelerated*; these cases are clearly marked, with acknowledgement being given to the original authors.

3.1 Installation

pytorch-accelerated can be installed from pip using the following command:

```
pip install pytorch-accelerated
```

To make the package as slim as possible, the packages required to run the examples are not included by default. To include these packages, you can use the following command:

```
pip install pytorch-accelerated[examples]
```

3.2 Quickstart

To get started, simply import and use the *pytorch-accelerated* `pytorch_accelerated.trainer.Trainer`, as demonstrated in the following snippet, and then launch training using the accelerate CLI as described below:

```
# examples/vision/train_mnist.py
import os

from torch import nn, optim
from torch.utils.data import random_split
from torchvision import transforms
from torchvision.datasets import MNIST

from pytorch_accelerated import Trainer

class MNISTModel(nn.Module):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        super().__init__()
        self.main = nn.Sequential(
            nn.Linear(in_features=784, out_features=128),
            nn.ReLU(),
            nn.Linear(in_features=128, out_features=64),
            nn.ReLU(),
            nn.Linear(in_features=64, out_features=10),
        )

    def forward(self, input):
        return self.main(input.view(input.shape[0], -1))

def main():
    dataset = MNIST(os.getcwd(), download=True, transform=transforms.ToTensor())
    train_dataset, validation_dataset, test_dataset = random_split(dataset, [50000, 5000,
↪ 5000])
    model = MNISTModel()
    optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
    loss_func = nn.CrossEntropyLoss()

    trainer = Trainer(
            model,
            loss_func=loss_func,
            optimizer=optimizer,
    )

    trainer.train(
        train_dataset=train_dataset,
        eval_dataset=validation_dataset,
        num_epochs=8,
        per_device_batch_size=32,
    )

    trainer.evaluate(
        dataset=test_dataset,
        per_device_batch_size=64,
    )

if __name__ == "__main__":
    main()
```

To launch training using the accelerate CLI on your machine(s), run:

```
accelerate config --config_file accelerate_config.yaml
```

and answer the questions asked. This will generate a config file that will be used to properly set the default options when doing:

```
accelerate launch --config_file accelerate_config.yaml train.py [--training-args]
```

Note: Using the accelerate CLI is completely optional, training can also be launched in the usual way using:

```
python train.py / python -m torch.distributed ...
```

depending on your infrastructure configuration, for users who would like to maintain a more fine-grained control over the launch command.

3.2.1 Running in a Notebook

Accelerate also provides a `notebook_launcher()` function, that can be used to launch distributed training from a notebook; which is especially useful for Colab or Kaggle notebooks.

To train a model using *pytorch_accelerated* from a notebook, just define the *`Trainer()`* in a training_function, and use this as an argument into `notebook_launcher`. To run the example in above in a notebook, we would use:

```
notebook_launcher(main, num_processes=num_gpus)
```

More information about training in a notebook can be found here

3.2.2 Debugging with an IDE

Whilst `pytorch_accelerated` is primarily designed to be launched using the accelerate CLI, sometimes it's useful to debug a training script in your favourite editor to see exactly what's going on!

In these cases, we can simply use the `notebook_launcher()` function as described above. To debug the example above, after setting some breakpoints, replace the lines:

```
if __name__ == "__main__":
    main()
```

with:

```
notebook_launcher(main, num_processes=num_gpus)
```

3.2.3 Next steps

More complex training examples can be seen in the examples folder here

Alternatively, if you would prefer to read more about the *`Trainer`*, you can do so here: *Trainer*.

3.3 Trainer

class pytorch_accelerated.trainer.**Trainer**(*model*, *loss_func*, *optimizer*, *callbacks=(<class 'pytorch_accelerated.callbacks.MoveModulesToDeviceCallback'>, <class 'pytorch_accelerated.callbacks.TerminateOnNaNCallback'>, <class 'pytorch_accelerated.callbacks.PrintProgressCallback'>, <class 'pytorch_accelerated.callbacks.ProgressBarCallback'>, <class 'pytorch_accelerated.callbacks.LogMetricsCallback'>)*, *run_history=None*)

The Trainer is designed to encapsulate an entire training loop for a specific task, bringing together the model, loss function and optimizer, and providing a specification of the behaviour to execute for each step of the training process.

The trainer has been implemented such that it provides (overridable) implementations of the parts of training that rarely change after they have been defined – such as creating a data loader, or how a batch of data is fed to the model – whilst remaining decoupled from components that are likely to change, such as the model, dataset, loss function and optimizer.

__init__(*model*, *loss_func*, *optimizer*, *callbacks=(<class 'pytorch_accelerated.callbacks.MoveModulesToDeviceCallback'>, <class 'pytorch_accelerated.callbacks.TerminateOnNaNCallback'>, <class 'pytorch_accelerated.callbacks.PrintProgressCallback'>, <class 'pytorch_accelerated.callbacks.ProgressBarCallback'>, <class 'pytorch_accelerated.callbacks.LogMetricsCallback'>)*, *run_history=None*)

Create a new trainer object which can be used to train the given model using the provided loss function and optimizer.

Parameters

- **model** – a subclass of nn.Module to be trained
- **loss_func** – the loss function to use when training the model
- **optimizer** – the optimizer to update the model's parameters
- **callbacks** – a list of callbacks to use during training runs. If a list of callbacks is not provided, the default selection will be used.
- **run_history** – an instance of a RunHistory subclass to track training runs. If this is not provided, a new one will be created.

The callbacks that are used by default are (*MoveModulesToDeviceCallback*, *TerminateOnNaNCallback*, *PrintProgressCallback*, *ProgressBarCallback*, *LogMetricsCallback*,)

class pytorch_accelerated.trainer.**TrainerWithTimmScheduler**(**args*, ***kwargs*)

Subclass of the *Trainer* that works with timm schedulers instead of standard PyTorch learning rate schedulers

3.3.1 Training a model

The main entrypoint for the *Trainer* is the *train()* method, which is used to launch a training run.

Trainer.**train**(*train_dataset*, *num_epochs*, *eval_dataset=None*, *per_device_batch_size=8*, *max_num_train_steps=None*, *gradient_accumulation_steps=1*, *gradient_clip_value=None*, *create_scheduler_fn=None*, *train_dataloader_kwargs: dict | None = None*, *eval_dataloader_kwargs: dict | None = None*, *reset_run_history=True*, *collate_fn=None*)

Start a training run. If an evaluation dataset is provided, this routine will include both training and evaluation epochs.

Note: As the optimizer needs to be internally prepared prior to training, in order to use a learning rate scheduler, a factory function must be provided to `create_scheduler_fn`. This must be a function which accepts the optimizer as a single parameter and returns an instance of a learning rate scheduler. Passing an instance of a learning rate scheduler will not work here.

Parameters

- **train_dataset** – the dataset to use during training epochs
- **num_epochs** – the number of epochs to train for
- **eval_dataset** – the dataset to use during evaluation epochs, if this is not provided, evaluation is skipped.
- **per_device_batch_size** – the batch size to use per device
- **max_num_train_steps** – the maximum number of steps to train for. If provided, this will override num_epochs
- **gradient_accumulation_steps** – accumulate gradients to the specified number of steps to simulate a bigger batch size. By default, this is set to 1
- **gradient_clip_value** – if specified, the gradients of the model's parameters will be clipped to the range `[-gradient_clip_value, gradient_clip_value]`
- **create_scheduler_fn** – a function which accepts an optimizer as an argument and returns a learning rate scheduler
- **train_dataloader_kwargs** – : a dictionary of keyword arguments to pass to the training dataloader constructor, for details see `torch.utils.data.DataLoader`
- **eval_dataloader_kwargs** – a dictionary of keyword arguments to pass to the evaluation dataloader constructor, for details see `torch.utils.data.DataLoader`
- **reset_run_history** – reset any run history saved by the trainer from previous training runs
- **collate_fn** – the collate function to be used by the training and evaluation dataloaders

Using learning rate schedulers

As Pytorch schedulers are not consistently called in the same way, to enable maximum flexibility, PyTorch-accelerated's *Trainer* expects that a given scheduler should be called after **each optimizer update** by default.

Note that, as the optimizer and dataloaders need to be internally prepared prior to training, in order to use a learning rate scheduler, a factory function must be provided to *train()* as the `create_scheduler_fn` argument. This must be a function which accepts the optimizer as a single parameter and returns an instance of a learning rate scheduler.

Note: Passing an instance of a PyTorch learning rate scheduler as the `create_scheduler_fn` argument to *train()* will **not** work as intended.

A simple method of creating a scheduler factory function this is by using `functools.partial()` like so:

```
from functools import Partial

from torch.optim import lr_scheduler

create_scheduler_fn = partial(lr_scheduler.StepLR, step_size=7, gamma=0.1)
```

Note: The *Trainer* calls a step on the provided scheduler after every batch. This can lead to unexpected results as some PyTorch schedulers are expected to step only after every epoch.

For instance, in the example above, the learning rate would be multiplied by `0.1` at every batch. As this particular scheduler is designed to be called once per epoch, this is not the desired behaviour! We can resolve this by representing the `step_size` in terms of the number of updates, like this:

```
from functools import Partial

from torch.optim import lr_scheduler

from pytorch_accelerated import TrainerPlaceholderValues

epochs_step_size = 7

create_scheduler_fn = partial(
    lr_scheduler.StepLR,
    step_size=TrainerPlaceHolderValues.NUM_UPDATE_STEPS_PER_EPOCH * epochs_step_size
)
```

Here, to determine the value of the number of steps per epoch, we have used a *TrainerPlaceholderValues* placeholder, which are described below.

Using TrainerPlaceHolderValues

class pytorch_accelerated.trainer.**TrainerPlaceholderValues**(*value*)

Some learning rate schedulers require information such as the total number of steps that will take place during a training run. As this information is not accessible prior to creating the training dataloader - which will be done as part of the *train()* method - a placeholder value can be used in the cases, as demonstrated below:

```
from functools import Partial

from pytorch_accelerated import TrainerPlaceholderValues
from torch.optim.lr_scheduler import OneCycleLR

create_scheduler_fn = partial(
            OneCycleLR,
            max_lr=config.lr,
            epochs=TrainerPlaceholderValues.NUM_EPOCHS,
            steps_per_epoch=TrainerPlaceholderValues.NUM_UPDATE_STEPS_PER_EPOCH,
        )
```

These placeholders will be replaced by the trainer with the correct values during training.

The list of the available placeholders are:

- NUM_EPOCHS
- NUM_UPDATE_STEPS_PER_EPOCH
- TRAIN_DATALOADER_LEN
- EVAL_DATALOADER_LEN

Alternatively, the same outcome could be achieved by overriding the *Trainer*'s *create_scheduler()* method.

Using PyTorch-accelerated schedulers

PyTorch-accelerated includes some implementations of schedulers, which have the same interface as PyTorch schedulers, as well as base classes to easily create custom schedules; these are discussed in more detail in *Schedulers*.

These scheduler implementations have an alternative constructor, which can be passed to *train()* as the the `create_scheduler_fn` argument directly, as demonstrated below:

```
from pytorch_accelerated.schedulers import CosineLrScheduler

trainer.train(
        train_dataset=train_dataset,
        num_epochs=num_epochs,
        per_device_batch_size=batch_size,
        create_scheduler_fn=CosineLrScheduler.create_scheduler_fn(num_warmup_epochs=5,
                                                                  warmup_starting_lr=1e-
↪6,
                                                                  num_cooldown_epochs=5),
        )
```

Using timm schedulers

The schedulers included in timm have a different interface to the native PyTorch schedulers, so do not work with the base *Trainer* by default.

PyTorch-accelerated includes an alternative trainer *TrainerWithTimmScheduler*, which is compatible with timm schedulers; schedulers should be passed to this trainer as a factory function the same way as described above.

3.3.2 Evaluating a model

Once a model has been trained, or loaded from a checkpoint, the *Trainer* can also be used for evaluation, which consists of running a single epoch, using the *Trainer*'s evaluation loop logic, on the given dataset.

`Trainer.`**`evaluate`**(*dataset=None*, *per_device_batch_size=8*, *dataloader_kwargs: dict | None = None*, *collate_fn=None*)

Start an evaluation run.

Note: Starting an evaluation run will reset the *Trainer*'s run history.

Note: During distributed evaluation, if the *per_device_batch_size* * the number of processes used does not exactly divide the dataset, and *drop_last=False* has not been passed as a dataloader kwarg, the dataloader will repeat from the start in processes that run out of batches. This should be taken into consideration when calculating metrics.

Parameters

- **`dataset`** – the dataset to use during evaluation
- **`per_device_batch_size`** – the batch size to use per device

- **dataloader_kwargs** – a dictionary of keyword arguments to pass to the dataloader constructor, for details see `torch.utils.data.DataLoader`
- **collate_fn** – the collate function to be used by the dataloader

3.3.3 Utility Methods

`Trainer.save_checkpoint(save_path, checkpoint_kwargs=None, save_optimizer=True, save_per_node=True)`

Save the model, optimizer and specified args as a checkpoint file.

Parameters

- **save_path** – the path where to save the checkpoint, this should end in '.pt'
- **checkpoint_kwargs** – additional objects to include in the checkpoint
- **save_optimizer** – flag to indicate whether to include the optimizer in the checkpoint
- **save_per_node** – flag to indicate whether to save the checkpoint once per machine, if False, the checkpoint will only be saved from the world process zero. This is True by default.

`Trainer.load_checkpoint(checkpoint_path, load_optimizer=True)`

Load the model and optimizer from a checkpoint file.

Parameters

- **checkpoint_path** – the path of the checkpoint file to load
- **load_optimizer** – flag to indicate whether to load the optimizer if it is included in the checkpoint

`Trainer.print(*args, **kwargs)`

Use in replacement of `print()` to only print once per node.

`Trainer.gather(tensor, padding_value=None)`

Gather the values in *tensor* across all processes and concatenate them on the first dimension. This can be useful to regroup the predictions from all processes when doing evaluation.

If a padding value is provided, padding will be applied along the first dimension where necessary, to ensure that tensors in all processes have the same shape.

Note: The given value of *padding_value* should ideally not appear in the expected range of values that the tensor may contain

---

Parameters

- **tensor** – (`torch.Tensor`, or a nested tuple/list/dictionary of `torch.Tensor`) The tensors to gather across all processes.
- **padding_value** – if provided, the value with which to pad tensors to ensure that all processes have the same shape

Returns

The gathered tensor(s) (`torch.Tensor`, or a nested tuple/list/dictionary of `torch.Tensor`). The first dimension of the result is *num_processes* multiplied by the first dimension of the input tensors.

Note: This gather happens in all processes.

`Trainer.get_model()`

Extract the model in *Trainer* from its distributed containers. Useful before saving a model.

Returns
the model in *Trainer*, subclassed from `Module`

3.3.4 Customizing Trainer Behaviour

Whilst the *Trainer* should work out of the box in straightforward use cases, subclassing the trainer and overriding its methods is intended and encouraged - think of the base implementation as a set of 'sensible defaults'!

Note: Methods which are prefixed with a verb such as *create* or *calculate* expect a value to be returned, all other methods are used to set internal state (e.g. `optimizer.step()`)

Setup Methods

`Trainer.create_train_dataloader(batch_size: int, train_dl_kwargs: dict | None = None) → Iterable`

Create a dataloader to be used during training. This is initialised with the train_dataset and collate function which have been passed to the Trainer.

If no arguments are passed, the arguments returned by *Trainer.get_default_train_dl_kwargs()* are used.

Note: if batch size is included in train_dl_kwargs, this takes precedence over the batch_size argument.

Parameters

- **batch_size** – the batch size to use per device
- **train_dl_kwargs** – a dictionary of keyword arguments to pass to the dataloader constructor, for details see `torch.utils.data.DataLoader`

Returns
an instance of `DataLoader`

`Trainer.get_default_train_dl_kwargs(batch_size) → dict`

Return the default arguments that will be used by the training dataloader.

Parameters
batch_size – the batch size to use during training

Returns
a dictionary containing the default arguments for the training dataloader

`Trainer.create_eval_dataloader(batch_size: int, eval_dl_kwargs: dict | None = None) → Iterable`

Create a dataloader to be used during evaluation. This is initialised with the eval_dataset and collate function which have been passed to the Trainer.

If no arguments are passed, the arguments returned by *Trainer.get_default_eval_dl_kwargs()* are used.

Note: if batch size is included in eval_dl_kwargs, this takes precedence over the batch_size argument.

Parameters

- **batch_size** – the batch size to use per device
- **eval_dl_kwargs** – a dictionary of keyword arguments to pass to the dataloader constructor, for details see `torch.utils.data.DataLoader`

Returns
an instance of `torch.utils.data.DataLoader`

`Trainer.get_default_eval_dl_kwargs(`*batch_size*`)` → dict

Return the default arguments that will be used by the evaluation dataloader.

Parameters
batch_size – the batch size to use during evaluation

Returns
a dictionary containing the default arguments for the evaluation dataloader

`Trainer.create_scheduler()`

Create a learning rate scheduler based on the `create_scheduler_fn` function which has been passed to the Trainer. :return: a learning rate scheduler instance

Training Run Methods

`Trainer.training_run_start()`

This method is called at the start of a training run.

`Trainer.training_run_epoch_end()`

This method is called during a training run after both training and evaluation epochs have been completed.

`Trainer.training_run_end()`

This method is called at the end of a training run.

Training epoch Methods

`Trainer.train_epoch_start()`

This method is called at the start of a training epoch.

The default behaviour of this method is to call `self.model.train()`

`Trainer.calculate_train_batch_loss(`*batch*`)` → dict

Calculates the training loss and return this along with the batch size and model outputs. Any additional values returned will be available in the `on_train_step_end()` callback method.

Parameters
batch – the output of the train dataloader

Returns
A dictionary containing the training loss, model outputs and batch size. Can include any keys, but must include the keys 'loss', 'model_outputs' and 'batch_size'

Trainer.**backward_step**(*loss*)

Use the accelerator to perform the backward pass on the calculated value of the loss returned by *calculate_train_batch_loss()*. If gradient accumulation is enabled, this loss has been scaled by 1 / accumulation steps.

Parameters
loss – The loss tensor returned by *calculate_train_batch_loss()*.

Trainer.**optimizer_step**()

Performs a single optimization step and updates the parameters which have been passed to `self.optimizer`.

Trainer.**scheduler_step**()

Performs a single scheduler step if `self.scheduler` has been assigned.

Trainer.**optimizer_zero_grad**()

Sets the gradients of all optimized `torch.Tensor` s to zero.

Trainer.**train_epoch_end**()

This method is called at the end of each training epoch.

Evaluation epoch Methods

Trainer.**eval_epoch_start**()

This method is called at the start of an evaluation epoch.

The default behaviour of this method is to call `self.model.eval()`

Trainer.**calculate_eval_batch_loss**(*batch*) → dict

Calculates the evaluation loss and return this along with the batch size and model outputs. Any additional values returned will be available in the `on_eval_step_end()` callback.

Parameters
batch – the output of the eval dataloader

Returns
A dictionary containing the evaluation loss, model outputs and batch size. Can include any keys, but must include the keys `loss`, `model_outputs` and `batch_size`

Trainer.**eval_epoch_end**()

This method is called at the end of an evaluation epoch.

Evaluation Run Methods

Trainer.**evaluation_run_start**()

This method is called at the start of an evaluation run.

Trainer.**evaluation_run_end**()

This method is called at the end of an evaluation run.

Internal Methods

Warning: In the spirit of Python, nothing is truly hidden within the *Trainer*. However, care must be taken as, by overriding these methods, you are fundamentally changing how the *Trainer* is working internally and this may have untended consequences. When modifying one or more internal methods, it is the responsibility of the user to ensure that the *Trainer* continues to work as intended!

Internal Setup

`Trainer._create_accelerator()`

Create an instance of `accelerate.Accelerator` which will be used to manage training.

`Trainer._prepare_model_optimizer_and_dataloaders()`

Uses the trainer's instance of `accelerate.Accelerator` to wrap the model, optimizer and dataloaders in any wrappers necessary for training. (e.g. `torch.nn.parallel.DistributedDataParallel`) and ensures the parameters are placed on the appropriate device.

By default, this will convert each dataloader to an instance of `accelerate.data_loader.DataLoaderShard`. Depending on the value of the *drop_last* attribute of the dataloaders, either iterations will stop at the first batch that would be too small / not present on all processes or loop with batches from the beginning on processes which run out of data, so that all batch sizes are the same size.

Note: This may change the length of the dataloaders, so this should be called *before* the number of update steps per epoch is calculated, i.e. to initialise a learning rate scheduler

---

`Trainer._create_run_config(`*per_device_batch_size*, *num_epochs*, *gradient_accumulation_steps*, *max_num_train_steps*, *gradient_clip_value*`)` → *TrainerRunConfig*

Create an instance of *TrainerRunConfig* representing the current state of the trainer.

Parameters

- **per_device_batch_size** – the batch size per device
- **num_epochs** – the number of epochs in the current training run
- **gradient_accumulation_steps** – the number of gradient accumulation steps which will be used during the training run
- **max_num_train_steps** – If specified, the maximum number of steps to train for. If present, this will take precedence over `num_epochs`
- **gradient_clip_value** – the value used to determine the threshold to clip the gradients of the model's parameters

Training run behaviour

`Trainer._run_training()`

The method responsible for the orchestration of the high level steps which will be executed during a training run.

Training epoch behaviour

`Trainer._run_train_epoch(train_dl)`

The method responsible for the behaviour of each training epoch.

Parameters
train_dl – the dataloader to be used during training

`Trainer._clip_gradients()`

Clip the gradients of the model's parameters that fall outside of the threshold specified in *train()*.

By default, this clips the gradients using `accelerate.Accelerator.clip_grad_value_()`

Evaluation epoch behaviour

`Trainer._run_eval_epoch(valid_dl, is_training: bool = True)`

The method responsible for the behaviour of each evaluation epoch.

Parameters

- **valid_dl** – the dataloader to be used during evaluation
- **is_training** – signals whether the evaluation is being run as part of a training run

Should I subclass the Trainer or use a callback?

The behaviour of the *Trainer* can also be extended using Callbacks. All callback methods are prefixed with on_.

It is recommended that callbacks are used to contain 'infrastructure' code, which is not essential to the operation of the training loop, such as logging, but this decision is left to the judgement of the user based on the specific use case. If it seems overkill to subclass the *Trainer* for the modification you wish to make, it may be better to use a callback instead.

For more information on callbacks, see *Callbacks*.

3.3.5 Recording metrics

The *Trainer* contains an instance of *RunHistory*, which can be used to store and retrieve the values of any metrics to track during training. By default, the only metrics that are recorded by the *Trainer* are the losses observed during training and evaluation.

Note: If the callback `PrintMetricsCallback` is being used, any metrics recorded in the run history will be printed to the console automatically.

The API for *RunHistory* is detailed at *RunHistory*.

Here is an example of how we can subclass the `Trainer` and use the *RunHistory* to track metrics computed using TorchMetrics:

```
from torchmetrics import MetricCollection, Accuracy, Precision, Recall
from pytorch_accelerated import Trainer

class TrainerWithMetrics(Trainer):
    def __init__(self, num_classes, *args, **kwargs):
        super().__init__(*args, **kwargs)

        # this will be moved to the correct device automatically by the
        # MoveModulesToDeviceCallback callback, which is used by default
        self.metrics = MetricCollection(
            {
                "accuracy": Accuracy(num_classes=num_classes),
                "precision": Precision(num_classes=num_classes),
                "recall": Recall(num_classes=num_classes),
            }
        )

    def calculate_eval_batch_loss(self, batch):
        batch_output = super().calculate_eval_batch_loss(batch)
        preds = batch_output["model_outputs"].argmax(dim=-1)

        self.metrics.update(preds, batch[1])

        return batch_output

    def eval_epoch_end(self):
        metrics = self.metrics.compute()
        self.run_history.update_metric("accuracy", metrics["accuracy"].cpu())
        self.run_history.update_metric("precision", metrics["precision"].cpu())
        self.run_history.update_metric("recall", metrics["recall"].cpu())

        self.metrics.reset()
```

Note: If you feel that subclassing the *Trainer* seems too excessive for this use case, this could also be done using a callback as demonstrated in *Example: Tracking metrics using a callback*.

3.3.6 What goes on inside the Trainer?

In pseudocode, the execution of a training run can be depicted as:

```
train_dl = create_train_dataloader()
eval_dl = create_eval_dataloader()
scheduler = create_scheduler()

training_run_start()
on_training_run_start()

for epoch in num_epochs:
    train_epoch_start()
    on_train_epoch_start()
```

(continues on next page)

(continued from previous page)

```
    for batch in train_dl:
        on_train_step_start()
        batch_output = calculate_train_batch_loss(batch)
        on_train_step_end(batch, batch_output)
        backward_step(batch_output["loss"])
        optimizer_step()
        scheduler_step()
        optimizer_zero_grad()
    train_epoch_end()
    on_train_epoch_end()

    eval_epoch_start()
    on_eval_epoch_start()
    for batch in eval_dl:
        on_eval_step_start()
        batch_output = calculate_eval_batch_loss(batch)
        on_eval_step_end(batch, batch_output)
    eval_epoch_end()
    on_eval_epoch_end()

    training_run_epoch_end()
    on_training_run_epoch_end()

training_run_end()
on_training_run_end()
```

Similarly, the execution of an evaluation run can be depicted as:

```
eval_dl = create_eval_dataloader()

evaluation_run_start()
on_evaluation_run_start()

eval_epoch_start()
on_eval_epoch_start()
for batch in eval_dl:
    on_eval_step_start()
    batch_output = calculate_eval_batch_loss(batch)
    on_eval_step_end(batch, batch_output)
eval_epoch_end()
on_eval_epoch_end()

evaluation_run_end()
on_evaluation_run_end()
```

The best way to understand how the *Trainer* works internally is by examining the source code for the *train()* method; significant care has gone into making the internal methods as clean and clear as possible.

3.4 Callbacks

In addition to overridable hooks, the *Trainer* also includes a callback system.

It is recommended that callbacks are used to contain 'infrastructure' code, which is not essential to the operation of the training loop, such as logging, but this decision is left to the judgement of the user based on the specific use case.

Warning: Callbacks are executed sequentially, so if a callback is used to modify state, such as updating a metric, it is the responsibility of the user to ensure that this callback is placed before any callback which will read this state (i.e. for logging purposes)!

Note: Callbacks are called **after** their corresponding hooks, e.g., a callback's `on_train_epoch_end` method is called *after* the method *pytorch_accelerated.trainer.Trainer.train_epoch_end()*. This is done to support the pattern of updating the trainer's state in a method before reading this state in a callback.

For more info on execution order within the training loop, see: *What goes on inside the Trainer?*.

3.4.1 Implemented Callbacks

class pytorch_accelerated.callbacks.**TerminateOnNaNCallback**

Bases: *TrainerCallback*

A callback that terminates the training run if a `NaN` loss is observed during either training or evaluation.

class pytorch_accelerated.callbacks.**LogMetricsCallback**

Bases: *TrainerCallback*

A callback that logs the latest values of any metric which has been updated in the trainer's run history. By default, this just prints to the command line once per machine.

Metrics prefixed with 'train' are logged at the end of a training epoch, all other metrics are logged after evaluation.

This can be subclassed to create loggers for different platforms by overriding the `log_metrics()` method.

class pytorch_accelerated.callbacks.**PrintProgressCallback**

Bases: *TrainerCallback*

A callback which prints a message at the start and end of a run, as well as at the start of each epoch.

class pytorch_accelerated.callbacks.**ProgressBarCallback**

Bases: *TrainerCallback*

A callback which visualises the state of each training and evaluation epoch using a progress bar

class pytorch_accelerated.callbacks.**SaveBestModelCallback**(*save_path='best_model.pt'*, *watch_metric='eval_loss_epoch'*, *greater_is_better: bool = False*, *reset_on_train: bool = True*, *save_optimizer: bool = True*)

Bases: *TrainerCallback*

A callback which saves the best model during a training run, according to a given metric. The best model weights are loaded at the end of the training run.

__init__(*save_path='best_model.pt'*, *watch_metric='eval_loss_epoch'*, *greater_is_better: bool = False*, *reset_on_train: bool = True*, *save_optimizer: bool = True*)

Parameters

- **save_path** – The path to save the checkpoint to. This should end in `.pt`.
- **watch_metric** – the metric used to compare model performance. This should be accessible from the trainer's run history.
- **greater_is_better** – whether an increase in the `watch_metric` should be interpreted as the model performing better.
- **reset_on_train** – whether to reset the best metric on subsequent training runs. If `True`, only the metrics observed during the current training run will be compared.
- **save_optimizer** – whether to also save the optimizer as part of the model checkpoint

class pytorch_accelerated.callbacks.**ModelEmaCallback**(*decay: float = 0.99*, *evaluate_during_training: bool = True*, *save_path: str = 'ema_model.pt'*, *watch_metric: str = 'ema_model_eval_loss_epoch'*, *greater_is_better: bool = False*, *model_ema=<class 'pytorch_accelerated.utils.ModelEma'>*, *callbacks=()*)

Bases: *SaveBestModelCallback*

A callback which maintains and saves an exponential moving average of the weights of the model that is currently being trained.

This callback offers the option of evaluating the EMA model during. If enabled, this is done by running an additional validation after each training epoch, which will use additional GPU resources. During this additional epoch, only the provided callbacks will be executed.

Note: This callback is sensitive to the order that it is executed. This should be placed after any callbacks that modify state (e.g. metrics) but before any callbacks that read state (e.g. loggers) or *ConvertSyncBatchNormCallback*.

__init__(*decay: float = 0.99*, *evaluate_during_training: bool = True*, *save_path: str = 'ema_model.pt'*, *watch_metric: str = 'ema_model_eval_loss_epoch'*, *greater_is_better: bool = False*, *model_ema=<class 'pytorch_accelerated.utils.ModelEma'>*, *callbacks=()*)

Parameters

- **decay** – the amount of decay to use, which determines how much of the previous state will be maintained.
- **evaluate_during_training** – whether to evaluate the EMA model during training. If True, an additional validation epoch will be conducted after each training epoch, which will use additional GPU resources, and the best model will be saved. If False, the saved EMA model checkpoint will be updated at the end of each epoch.
- **watch_metric** – the metric used to compare model performance. This should be accessible from the trainer's run history. This is only used when `evaluate_during_training` is enabled.
- **greater_is_better** – whether an increase in the `watch_metric` should be interpreted as the model performing better.

- **model_ema** – the class which is responsible for maintaining the moving average of the model.
- **callbacks** – an iterable of callbacks that will be executed during the evaluation loop of the EMA model

class pytorch_accelerated.callbacks.**EarlyStoppingCallback**(*early_stopping_patience: int = 1*, *early_stopping_threshold: float = 0.01*, *watch_metric='eval_loss_epoch'*, *greater_is_better: bool = False*, *reset_on_train: bool = True*)

Bases: *TrainerCallback*

A callback which stops training early if progress is not being observed.

__init__(*early_stopping_patience: int = 1*, *early_stopping_threshold: float = 0.01*, *watch_metric='eval_loss_epoch'*, *greater_is_better: bool = False*, *reset_on_train: bool = True*)

Parameters

- **early_stopping_patience** – the number of epochs with no improvement after which training will be stopped.
- **early_stopping_threshold** – the minimum change in the `watch_metric` to qualify as an improvement, i.e. an absolute change of less than this threshold, will count as no improvement.
- **watch_metric** – the metric used to compare model performance. This should be accessible from the trainer's run history.
- **greater_is_better** – whether an increase in the `watch_metric` should be interpreted as the model performing better.
- **reset_on_train** – whether to reset the best metric on subsequent training runs. If `True`, only the metrics observed during the current training run will be compared.

class pytorch_accelerated.callbacks.**MoveModulesToDeviceCallback**

Bases: *TrainerCallback*

A callback which moves any *Trainer* attributes which are instances of `torch.nn.Module` on to the appropriate device at the start of a training or evaluation run.

Note: This does **not** include the model, as this will be prepared separately by the *Trainer*'s internal instance of `accelerate.Accelerator`.

---

class pytorch_accelerated.callbacks.**ConvertSyncBatchNormCallback**

Bases: *TrainerCallback*

A callback which converts all BatchNorm*D layers in the model to `torch.nn.SyncBatchNorm` layers.

3.4.2 Creating New Callbacks

To create a new callback containing custom behaviour, e.g. logging to an external platform, it is recommended to subclass *TrainerCallback*. To avoid confusion with the *Trainer*'s methods, all callback methods are prefixed with _on.

Warning: For maximum flexibility, the current instance of the *Trainer* is available in every callback method. However, changing the trainer state within a callback can have unintended consequences, as this may affect other parts of the training run. If a callback is used to modify *Trainer* state, it is responsibility of the user to ensure that everything continues to work as intended.

class pytorch_accelerated.callbacks.**TrainerCallback**

The abstract base class to be subclassed when creating new callbacks.

on_init_end(*trainer*, ***kwargs*)

Event called at the end of trainer initialisation.

on_training_run_start(*trainer*, ***kwargs*)

Event called at the start of training run.

on_train_epoch_start(*trainer*, ***kwargs*)

Event called at the beginning of a training epoch.

on_train_step_start(*trainer*, ***kwargs*)

Event called at the beginning of a training step.

on_train_step_end(*trainer*, *batch*, *batch_output*, ***kwargs*)

Event called at the end of a training step.

Parameters

- **batch** – the current batch of training data
- **batch_output** – the outputs returned by *pytorch_accelerated.trainer.Trainer.calculate_train_batch_loss()*

on_train_epoch_end(*trainer*, ***kwargs*)

Event called at the end of a training epoch.

on_eval_epoch_start(*trainer*, ***kwargs*)

Event called at the beginning of an evaluation epoch.

on_eval_step_start(*trainer*, ***kwargs*)

Event called at the beginning of a evaluation step.

on_eval_step_end(*trainer*, *batch*, *batch_output*, ***kwargs*)

Event called at the end of an evaluation step.

Parameters

- **batch** – the current batch of evaluation data
- **batch_output** – the outputs returned by *pytorch_accelerated.trainer.Trainer.calculate_eval_batch_loss()*

on_eval_epoch_end(*trainer*, ***kwargs*)

Event called at the end of evaluation.

on_training_run_end(*trainer*, ***kwargs*)

Event called at the end of training run.

on_stop_training_error(*trainer*, ***kwargs*)

Event called when a stop training error is raised

Stopping Training Early

A training run may be stopped early by raising a `StopTrainingError`

Example: Tracking metrics using a callback

By default, the only metrics that are recorded by the *pytorch_accelerated.trainer.Trainer* are the losses observed during training and evaluation. To track additional metrics, we can extend this behaviour using a callback.

Here is an example of how we can define a callback and use the *RunHistory* to track metrics computed using TorchMetrics:

```
from torchmetrics import MetricCollection, Accuracy, Precision, Recall

class ClassificationMetricsCallback(TrainerCallback):
    def __init__(self, num_classes):
        self.metrics = MetricCollection(
            {
                "accuracy": Accuracy(task="multiclass", num_classes=num_classes),
                "precision": Precision(task="multiclass", num_classes=num_classes),
                "recall": Recall(task="multiclass", num_classes=num_classes),
            }
        )

    def _move_to_device(self, trainer):
        self.metrics.to(trainer.device)

    def on_training_run_start(self, trainer, **kwargs):
        self._move_to_device(trainer)

    def on_evaluation_run_start(self, trainer, **kwargs):
        self._move_to_device(trainer)

    def on_eval_step_end(self, trainer, batch, batch_output, **kwargs):
        preds = batch_output["model_outputs"].argmax(dim=-1)
        self.metrics.update(preds, batch[1])

    def on_eval_epoch_end(self, trainer, **kwargs):
        metrics = self.metrics.compute()
        trainer.run_history.update_metric("accuracy", metrics["accuracy"].cpu())
        trainer.run_history.update_metric("precision", metrics["precision"].cpu())
        trainer.run_history.update_metric("recall", metrics["recall"].cpu())

        self.metrics.reset()
```

Note: If you feel that it would be clearer to compute metrics as part of the training loop, this could also be done by subclassing the *pytorch_accelerated.trainer.Trainer* as demonstrated in *Recording metrics*.

Example: Create a custom logging callback

It is recommended that callbacks are used to handle logging, to keep the training loop focused on the ML related code. It is easy to create loggers for other platforms by subclassing the *LogMetricsCallback* callback. For example, we can create a logger for AzureML (which uses the MLFlow API) as demonstrated below:

```
import mlflow

class AzureMLLoggerCallback(LogMetricsCallback):
    def __init__(self):
        mlflow.set_tracking_uri(os.environ['MLFLOW_TRACKING_URI'])

    def on_training_run_start(self, trainer, **kwargs):
        mlflow.set_tags(trainer.run_config.to_dict())

    def log_metrics(self, trainer, metrics):
        if trainer.run_config.is_world_process_zero:
            mlflow.log_metrics(metrics)
```

Example: Create a custom callback to save predictions on evaluation

Here is an example custom callback to record predictions during evaluation and then save them to csv at the end of the evaluation epoch:

```
from collections import defaultdict
import pandas as pd

class SavePredictionsCallback(TrainerCallback):

    def __init__(self, out_filename='./outputs/valid_predictions.csv') -> None:
        super().__init__()
        self.predictions = defaultdict(list)
        self.out_filename = out_filename

    def on_eval_step_end(self, trainer, batch, batch_output, **kwargs):
        input_features, targets = batch
        class_preds = trainer.gather(batch_output['model_outputs']).argmax(dim=-1)
        self.predictions['prediction'].extend(class_preds.cpu().tolist())
        self.predictions['targets'].extend(targets.cpu().tolist())

    def on_eval_epoch_end(self, trainer, **kwargs):
        trainer._accelerator.wait_for_everyone()
        if trainer.run_config.is_local_process_zero:
            df = pd.DataFrame.from_dict(self.predictions)
            df.to_csv(f'{self.out_filename}', index=False)
```

3.4.3 Callback handler

The execution of any callbacks passed to the *Trainer* is handled by an instance of an internal callback handler class.

class pytorch_accelerated.callbacks.**CallbackHandler**(*callbacks*)

The *CallbackHandler* is responsible for calling a list of callbacks. This class calls the callbacks in the order that they are given.

add_callback(*callback*)

Add a callbacks to the callback handler

Parameters
callback – an instance of a subclass of *TrainerCallback*

add_callbacks(*callbacks*)

Add a list of callbacks to the callback handler

Parameters
callbacks – a list of *TrainerCallback*

call_event(*event*, **args*, ***kwargs*)

For each callback which has been registered, sequentially call the method corresponding to the given event.

Parameters

- **event** – The event corresponding to the method to call on each callback
- **args** – a list of arguments to be passed to each callback
- **kwargs** – a list of keyword arguments to be passed to each callback

Creating new callback events

To add even more flexibility, it is relatively simple to define custom callback events, and use them in the training loop:

```
class VerifyBatchCallback(TrainerCallback):
    def verify_train_batch(self, trainer, xb, yb):
        assert xb.shape[0] == trainer.run_config["train_per_device_batch_size"]
        assert xb.shape[1] == 1
        assert xb.shape[2] == 28
        assert xb.shape[3] == 28
        assert yb.shape[0] == trainer.run_config["train_per_device_batch_size"]


class TrainerWithCustomCallbackEvent(Trainer):
    def calculate_train_batch_loss(self, batch) -> dict:
        xb, yb = batch
        self.callback_handler.call_event(
            "verify_train_batch", trainer=self, xb=xb, yb=yb
        )
        return super().calculate_train_batch_loss(batch)
```

3.5 Tracking

3.5.1 RunHistory

class pytorch_accelerated.tracking.**RunHistory**

The abstract base class which defines the API for a *Trainer*'s run history.

abstract property current_epoch: int

Return the value of the current epoch.

Returns
an int representing the value of the current epoch

abstract get_latest_metric(*metric_name*)

Return the most recent value that has been recorded for the given metric.

Parameters
metric_name – the name of the metric being tracked

Returns
the last recorded value

abstract get_metric_names() → Iterable

Return a set containing of all unique metric names which are being tracked.

Returns
an iterable of the unique metric names

abstract get_metric_values(*metric_name*) → Iterable

Return all of the values that have been recorded for the given metric.

Parameters
metric_name – the name of the metric being tracked

Returns
an ordered iterable of values that have been recorded for that metric

abstract property metric_name_prefix

Returns
the prefix which wil be prepended to any metric name

abstract reset()

Reset the state of the *RunHistory*

abstract set_metric_name_prefix(*prefix=''*)

Set a prefix which will be prepended to any metric name which is tracked.

Parameters
prefix – a prefix which will be prepended to any metric name which is tracked

abstract update_metric(*metric_name*, *metric_value*)

Record the value for the given metric.

Parameters

- **metric_name** – the name of the metric being tracked
- **metric_value** – the value to record

3.5.2 Implementations

class pytorch_accelerated.tracking.**InMemoryRunHistory**

Bases: *RunHistory*

An implementation of *RunHistory* which stores all recorded values in memory.

3.6 Run Config

3.6.1 RunConfig

class pytorch_accelerated.run_config.**TrainerRunConfig**(*num_epochs: int*, *train_per_device_batch_size: int*, *train_dl_kwargs: dict*, *eval_per_device_batch_size: int*, *eval_dl_kwargs: dict*, *gradient_accumulation_steps: int*, *gradient_clip_value: Number | None*, *train_total_batch_size: int*, *eval_total_batch_size: int*, *num_update_steps_per_epoch: int*, *max_num_train_steps: int | None*, *is_local_process_zero: bool*, *is_world_process_zero: bool*, *is_distributed: bool*, *mixed_precision: str*, *num_processes: int*)

An immutable dataclass holding values representing the current state of the *Trainer*

Parameters

- **num_epochs** – the number of epochs in the current training run
- **train_per_device_batch_size** – the device size per batch used during training epochs
- **train_dl_kwargs** – the arguments that have been used to create the training dataloader
- **eval_per_device_batch_size** – the device size per batch used during evaluation epochs
- **eval_dl_kwargs** – the arguments that have been used to create the evaluation dataloader
- **gradient_accumulation_steps** – the number of gradient accumulation steps which will be used during training
- **gradient_clip_value** – the value used to determine the threshold to clip the gradients of the model's parameters
- **train_total_batch_size** – the total batch size used during training
- **eval_total_batch_size** – the total batch size used during evaluation
- **num_update_steps_per_epoch** – the number of steps per training epoch where the model's parameters will be updated
- **max_num_train_steps** – the maximum number of steps to train for, if present, this will take precedence over `num_epochs`
- **is_local_process_zero** – `True` if the current process is the main process on the current node, `False` otherwise

- **is_world_process_zero** – `True` if the current process is the main process across all nodes, `False` otherwise
- **is_distributed** – `True` if the trainer is set up to perform distributed training, `False` otherwise
- **mixed_precision** – A string containing the type of mixed precision the trainer is set up to use, `no` otherwise
- **num_processes** – the number of processes being used during training

3.7 Fine-tuning

3.7.1 ModelFreezer

class pytorch_accelerated.finetuning.**ModelFreezer**(*model*, *freeze_batch_norms=False*)

A class to freeze and unfreeze different parts of a model, to simplify the process of fine-tuning during transfer learning.

This class uses the following abstractions:

- *Layer*: A subclass of `torch.nn.Module` with a depth of 1. i.e. The module is not nested.
- *LayerGroup*: The modules which have been defined as attributes of a model. These may be Layers or nested modules.

For example, let's consider the following model:

```
from torch import nn

class MyModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.input = nn.Linear(100, 100)
        self.block_1 = nn.Sequential(
            nn.Linear(100, 100),
            nn.BatchNorm1d(100),
            nn.Sequential(
                nn.Linear(100, 100),
                nn.BatchNorm1d(100),
                nn.ReLU(),
            ),
        )
        self.output = nn.Linear(100, 10)

    def forward(self, x):
        x = self.input(x)
        x = self.block_1(x)
        out = self.output(x)
        return out
```

Here, the layer groups would be the modules [*input*, *block_1*, *output*], whereas the layers would be ordered, flattened list of Linear, BatchNorm and ReLU modules.

freeze(*from_index=0*, *to_index=-2*, *set_modules_as_eval=False*)

Freeze layer groups corresponding to the specified indexes, which are inclusive. By default, this freezes all layer groups except the final one.

Parameters

- **from_index** – The index of the first layer group to freeze.
- **to_index** – The index of the final layer group to freeze.
- **set_modules_as_eval** – If True, frozen modules will also be placed in *eval* mode. This is False by default.

get_layer_groups() → List[LayerGroup]

Return all of the model's layer groups. A layer group is any module which has been defined as an attribute of the model.

Returns
a list of all layer groups in the model.

get_layers() → List[Layer]

Return all of the model's layers. A Layer is any non-nested module which is included in the model.

Returns
a list of all layers in the model.

get_trainable_parameters()

Return a list of all unfrozen model parameters, which will be updated during training.

Returns
a list of all trainable parameters

unfreeze(*from_index=-1*, *to_index=0*, *set_modules_as_training=True*)

Unfreeze layer groups corresponding to the specified indexes, which are inclusive. By default, this unfreezes all layer groups. For each layer group, any parameters which have been unfrozen are returned, so that they can be added to an optimizer if needed.

Parameters

- **from_index** – The index of the first layer group to unfreeze.
- **to_index** – The index of the final layer group to unfreeze.
- **set_modules_as_training** – If True, unfrozen modules will also be placed in *train* mode. This is True by default.

Returns
a dictionary containing the parameters which have been unfrozen for each layer group.

3.8 Schedulers

PyTorch-accelerated provides some scheduler implementations which can be used in any PyTorch training loop. However, unlike PyTorch's native schedulers - which can be called at different points in the training loop - all Pytorch-accelerated schedulers expect to be called after **each optimizer update**.

3.8.1 Implemented Schedulers

class pytorch_accelerated.schedulers.cosine_scheduler.**CosineLrScheduler**(*optimizer: Optimizer*, *total_num_epochs: int*, *num_update_steps_per_epoch: int*, *k_decay=1.0*, *lr_min: float = 1e-06*, *min_lr_ratio=None*, *num_warmup_epochs: int = 0*, *warmup_starting_lr=1e-06*, *warmup_starting_lr_ratio=None*, *num_cooldown_epochs=0*)

Bases: *StatefulSchedulerBase*

A stateful Cosine annealing learning rate scheduler, as described in this paper, but without restarts.

This scheduler differs from the PyTorch's `CosineAnnealingLR` as it provides options to add warmup and cooldown epochs. Additionally, the annealing rate can be modified by adjusting the k-decay parameter, for which the rate of change of the learning rate is changed by its k-th order derivative, as described in here.

If warmup epochs are specified, the learning rate will increase in constant increments from the `warmup_starting_lr` provided until the learning rate specified in the parameter group is reached.

If cooldown epochs are specified, the learning rate will be fixed at the minimum lr value given. This behaviour will continue if the scheduler is called after the training cycle has completed.

__init__(*optimizer: Optimizer*, *total_num_epochs: int*, *num_update_steps_per_epoch: int*, *k_decay=1.0*, *lr_min: float = 1e-06*, *min_lr_ratio=None*, *num_warmup_epochs: int = 0*, *warmup_starting_lr=1e-06*, *warmup_starting_lr_ratio=None*, *num_cooldown_epochs=0*)

Create a new ConsineLrScheduler object which can be used to modify the learning rate in an optimizer's parameter groups.

Parameters

- **optimizer** – a PyTorch optimizer containing one or more parameter groups
- **total_num_epochs** – the total number of training epochs, inclusive of any warmup and cooldown epochs
- **num_update_steps_per_epoch** – the number of optimizer updates that take place per epoch
- **k_decay** – adjusts the rate of annealing. Higher values will maintain a higher lr for longer
- **lr_min** – the minimum value that the learning rate should decay to for all parameter groups. This will be held fixed during cooldown epochs
- **min_lr_ratio** – this can be used to represent the minimum lr for each parameter group as a ratio of its maximum lr. If set, this will take precedence over `lr_min`
- **num_warmup_epochs** – the number of epochs to gradually increase the lr until it reaches the maximum value
- **warmup_starting_lr** – the starting lr that will be used for all parameter groups at the beginning of training if `num_warmup_epochs` is greater than 0

- **warmup_starting_lr_ratio** – this can be used to represent the warmup starting lr for each parameter group as a ratio of its maximum lr. If set, this will take precedence over `warmup_starting_lr`
- **num_cooldown_epochs** – the number of epochs to hold the lr at its minimum value

classmethod create_scheduler_fn(*total_num_epochs: int = TrainerPlaceholderValues.NUM_EPOCHS*, *num_update_steps_per_epoch: int = TrainerPlaceholderValues.NUM_UPDATE_STEPS_PER_EPOCH*, *k_decay=1.0*, *lr_min: float = 1e-06*, *min_lr_ratio=None*, *num_warmup_epochs: int = 0*, *warmup_starting_lr=1e-06*, *warmup_starting_lr_ratio=None*, *num_cooldown_epochs=0*) → Callable

An alternative constructor which returns a function that accepts an optimizer and creates an instance of `CosineLrScheduler`. This is primarily intended to be used with the *Trainer* as illustrated below:

```
trainer.train(
train_dataset=train_dataset,
num_epochs=num_epochs,
per_device_batch_size=batch_size,
create_scheduler_fn=CosineLrScheduler.create_scheduler_fn(num_warmup_epochs=5),
)
```

By default, the `total_num_epochs` and `num_iterations_per_epoch` arguments will be set by the *Trainer* with the correct values at runtime.

Parameters

- **total_num_epochs** – the total number of training epochs, inclusive of any warmup and cooldown epochs
- **num_update_steps_per_epoch** – the number of optimizer updates that take place per epoch
- **k_decay** – adjusts the rate of annealing. Higher values will maintain a higher lr for longer
- **lr_min** – the minimum value that the learning rate should decay to for all parameter groups. This will be held fixed during cooldown epochs
- **min_lr_ratio** – this can be used to represent the minimum lr for each parameter group as a ratio of its maximum lr. If set, this will take precedence over `lr_min`
- **num_warmup_epochs** – the number of epochs to gradually increase the lr until it reaches the maximum value
- **warmup_starting_lr** – the starting lr that will be used for all parameter groups at the beginning of training if `num_warmup_epochs` is greater than 0
- **warmup_starting_lr_ratio** – this can be used to represent the warmup starting lr for each parameter group as a ratio of its maximum lr. If set, this will take precedence over `warmup_starting_lr`
- **num_cooldown_epochs** – the number of epochs to hold the lr at its minimum value

Returns

a function which accepts an optimizer as an argument and returns an instance of *CosineLrScheduler*

get_updated_values(*num_updates: int*)

Calculate the learning rate for a particular step given the number of previous updates.

If warmup epochs are specified, the learning rate will increase in constant increments from the `warmup_starting_lr` provided until the learning rate specified in the parameter group is reached.

If cooldown epochs are specified, the learning rate will be fixed at the minimum lr value given. This behaviour will continue if the scheduler is called after the training cycle has completed.

Between any warmup or cooldown epochs, the cosine annealing strategy will be used.

Parameters
num_updates – the number of previous updates

Returns
the learning rates with which to update each parameter group

3.8.2 Base Schedulers

PyTorch-accelerated provides base classes for two types of schedulers.

Stateful Schedulers

Stateful schedulers maintain an internal count corresponding to how many times the scheduler's *step()* method has beeen called. As these schedulers have the same interface as the native PyTorch schedulers, these are supported by the *Trainer* by default.

class pytorch_accelerated.schedulers.scheduler_base.**StatefulSchedulerBase**(*optimizer*, *param_group_field: str = 'lr'*)

A stateful parameter scheduler base class that can be used to update any field within an optimizer's parameter groups. The most common use case for this is learning rate scheduling.

Unlike PyTorch's schedulers, which can be called at different points in the training loop depending on the implementation, this class is intended to be consistently called at the end of each optimizer update.

This class is responsible for maintaining the number of updates, incrementing an internal count each time that the scheduler step is calculated.

The usage of this class is illustrated below:

```
for current_epoch, epoch in enumerate(num_epochs):
    for batch in train_dataloader:
        xb, yb = batch
        predictions = model(xb)
        loss = loss_func(predictions, yb)

        loss.backward()
        optimizer.step()

        scheduler.step()
```

__init__(*optimizer*, *param_group_field: str = 'lr'*)

Create a new instance of a stateful parameter scheduler.

Parameters

- **optimizer** – a PyTorch optimizer
- **param_group_field** – the field in the optimizer's parameter groups corresponding to the parameter to be scheduled

step()

Calculate the updated value of the scheduled parameter and update the optimizer's parameter groups.

Stateless Schedulers

These schedulers maintain no internal state about the current training run, and therefore require that the current number of updates is explicitly provided when called. To use a stateless scheduler with the *Trainer*, this would require subclassing the *Trainer* and overriding the *scheduler_step()* method.

class pytorch_accelerated.schedulers.scheduler_base.**SchedulerBase**(*optimizer: Optimizer*, *param_group_field: str = 'lr'*)

A parameter scheduler base class that can be used to update any field within an optimizer's parameter groups. The most common use case for this is learning rate scheduling.

Unlike PyTorch's schedulers, which can be called at different points in the training loop depending on the implementation, this class is intended to be consistently called at the end of each optimizer update.

As this class is stateless by default, it expects that the number of updates is explicitly provided, as illustrated below:

```
for current_epoch, epoch in enumerate(num_epochs):
    num_updates = current_epoch * num_update_steps_per_epoch
    for batch in train_dataloader:
        xb, yb = batch
        predictions = model(xb)
        loss = loss_func(predictions, yb)

        loss.backward()
        optimizer.step()

        num_updates +=1
        scheduler.step_update(num_updates)
```

__init__(*optimizer: Optimizer*, *param_group_field: str = 'lr'*)

Create a new instance of a parameter scheduler.

Parameters

- **optimizer** – a PyTorch optimizer
- **param_group_field** – the field in the optimizer's parameter groups corresponding to the parameter to be scheduled

abstract get_updated_values(*num_updates: int*) → None | Number | Iterable[Number]

Calculate updated values for the scheduled parameter.

If a single value is returned, all parameter groups will be updated with this value.

To update each parameter group with a different value, an iterable collection, containing an updated value for each parameter group, should be returned.

If None is returned, the parameter groups will not be updated.

Parameters

num_updates – the number of optimizer updates

Returns

the updated values of the scheduled parameter. This should be either a single value, or an iterable collection containing a value for each parameter group.

load_state_dict(*state_dict*)

Updates the attributes of the given scheduler from the given state dict.

Parameters
state_dict – the state dict to be loaded

state_dict()

Get the state dict for the scheduler, containing all attributes except the optimizer, which should be saved separately.

Returns
the scheduler's state dict

step_update(*num_updates: int*)

Calculate the updated value of the scheduled parameter and update the optimizer's parameter groups.

Parameters
num_updates – the number of optimizer updates

3.8.3 Creating New Schedulers

Whilst schedulers are usually used to schedule learning rates, the scheduler base classes in PyTorch-accelerated can be used to schedule any parameter in an optimizer's parameter group.

To create a new scheduler, in most cases, all that is required is to subclass one of the base classes and override the *get_updated_values()* method.

Example: Creating a simple milestone lr scheduler

Here is an example of how we can implement a scheduler to adjust the learning rate for each parameter group by a factor gamma each time an epoch milestone is reached:

```
from pytorch_accelerated.schedulers import StatefulSchedulerBase

class MilestoneLrScheduler(StatefulSchedulerBase):
    def __init__(
        self, optimizer, gamma=0.5, epoch_milestones=(2, 4, 5), num_steps_per_epoch=100
    ):
        super().__init__(optimizer, param_group_field="lr")
        self.milestones = set(
            (num_steps_per_epoch * milestone for milestone in epoch_milestones)
        )
        self.gamma = gamma

    def get_updated_values(self, num_updates: int):
        if num_updates in self.milestones:
            lr_values = [
                group[self.param_group_field] for group in self.optimizer.param_groups
            ]
            updated_lrs = [lr * self.gamma for lr in lr_values]
            return updated_lrs
```

Example: Scheduling weight decay

Here is an example of how we can define a scheduler to incrementally increase the amount of weight decay by a factor gamma every n steps:

```
from pytorch_accelerated.schedulers import StatefulSchedulerBase

class StepWdScheduler(StatefulSchedulerBase):
    def __init__(self, optimizer, n=1000, gamma=1.1):
        super().__init__(optimizer, param_group_field="weight_decay")
        self.n = n
        self.gamma = gamma

    def get_updated_values(self, num_updates: int):
        if num_updates % self.n == 0 and num_updates > 0:
            wd_values = [
                group[self.param_group_field] for group in self.optimizer.param_groups
            ]
            updated_wd_values = [wd * self.gamma for wd in wd_values]
            return updated_wd_values
```

3.9 Utils

3.9.1 Utils

class pytorch_accelerated.utils.**ModelEma**(*model*, *decay=0.9999*)

Maintains a moving average of everything in the model state_dict (parameters and buffers), based on the ideas from https://www.tensorflow.org/api_docs/python/tf/train/ExponentialMovingAverage.

This class maintains a copy of the model that we are training. However, rather than updating all of the parameters of this model after every update step, we set these parameters using a linear combination of the existing parameter values and the updated values

---

Note: It is important to note that this class is sensitive to where it is initialised. During distributed training, it should be applied before before the conversion to `SyncBatchNorm` takes place and before the `torch.nn.parallel.DistributedDataParallel` wrapper is used!

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- search

INDEX

Symbols

A

B

C

E

F

G

I

L

M

O