torchbearer Documentation

Release 0.5.3.dev

Torchbearer Contributors

Feb 17, 2020

Notes

CHAPTER 1

The Trial Class

The core class in torchbearer is *Trial*. This class contains the *Trial.run()* method which performs the main train-eval loop. In this note we'll go into detail regarding how to create and use a trial. We'll start with instantiation before looking at loading data (or running without data) and finally covering means of controlling verbosity. To get an understanding of actually running a trial and using metrics / callbacks, have a look at our example library.

1.1 Instantiation

If desired, we can instantiate a trial with no arguments by passing *None* as the *model* argument. However, assuming we have a model, we can simply write:

```
from torchbearer import Trial
trial = Trial(model)
```

If we would like our trial also perform optimization of some criterion we can use:

```
from torchbearer import Trial
trial = Trial(model, optimizer, criterion)
```

1.1.1 Criterions

Criterions can be given to a trial either as standard *torch.nn* criterions (i.e. functions of y_true, y_pred) or as functions of state. State is passed around a lot in torchbearer and contains (mutably) all of the different variables that are relevant to the fitting process at that point in time. Underneath it's just a dictionary but should be accessed with *StateKey* objects. The list of built-in state keys can be found here. In the case of criterions, this

```
from torchbearer import Trial

def my_criterion(y_pred, y_true):
    return (y_pred - y_true).abs()
```

(continues on next page)

(continued from previous page)

```
trial = Trial(model, optimizer, my_criterion)
```

is equivalent to

```
import torchbearer
from torchbearer import Trial

def my_criterion(state):
    return (state[torchbearer.PREDICTION] - state[torchbearer.TARGET]).abs()

trial = Trial(model, optimizer, my_criterion)
```

1.2 Loading Data

To load data into a trial we can use generators (such as a *torch.utils.data.DataLoader*) or tensors. We can further use different methods for loading train, val and test data or load all at once. Finally, we can let torchbearer decide how many steps per epoch to perform or tell it explicitly. All of these methods mutate the underlying trial and are chainable.

1.2.1 Generators

To populate the trial with a train generator we can use the *Trial.with_train_generator()* method. Equivalent methods *Trial.with_val_generator()* and *Trial.with_test_generator()* exist for validation and test data respectively. To use this method we write

```
from torchbearer import Trial

trial = Trial(model).with_train_generator(train_loader)
```

For simplicity, we can also load several data sets in one call using *Trial.with_generators()* like this

```
from torchbearer import Trial

trial = Trial(model).with_generators(train_loader, val_loader, test_loader)
```

To control the number of steps, we can either pass an integer argument *steps* to the *with_XXX_generator* methods or pass *train_steps*, *val_steps* and *test_steps* individually to *Trial.with_generators()*. Finally, we can use: *Trial.for_train_steps()*, *Trial.for_val_steps()*, *Trial.for_test_steps()*, *Trial.for_steps()*. That is, the following are all equivalent

```
trial = Trial(model).with_train_generator(train_loader, steps=10)
trial = Trial(model).with_generators(train_loader, train_steps=10)
trial = Trial(model).with_train_generator(train_loader).for_train_steps(10)
etc.
```

A final option is to tell the trial to run for infinitely many training steps (until stopped) for which we can use *Trial.with_inf_train_loader()*. For example

```
trial = Trial(model).with_train_generator(train_loader).with_inf_loader()
```

For more info on data loaders see the custom data loaders example

1.2.2 Tensors

If we want to load tensors instead we can use the *with_XXX_data* methods or the *Trial.with_data()* method in much the same way as before. There are some additional arguments to control batch size, shuffle and number of workers. Here are some examples:

```
# Shuffled training data
trial = Trial(model).with_train_data(x, y, shuffle=True, batch_size=128)

# Test data (no targets)
trial = Trial(model).with_test_data(x, batch_size=128)

# with_data
trial = Trial(model).with_data(x_train, y_train, x_val, y_val, x_test, shuffle=True,␣
↪batch_size=128)
```

To change the number of steps we can use the same *steps* arguments or *for_steps* methods as before.

1.2.3 Running Without Data

If we want to run an optimisation or similar which does not require data, we simply call the *for_steps* methods without calling any *with_generator* / *with_data* methods. For example, to run for 100 train steps per cycle without any data, we use:

```
trial = Trial(model).for_train_steps(100)
```

In this case, the model will be given *None* as input at each step.

1.3 Controlling Verbosity

The verbosity of a trial can be controlled in two ways. First, a global verbosity is set in the init. Second each of the *run* / *evaluate* / *predict* methods can take a local verbosity argument which gets priority. If *verbose=2*, the *Tqdm* callback will be loaded with *on_epoch=False* so that a new progress bar is created for each epoch. If *verbose=1*, the *Tqdm* callback will be loaded with *on_epoch=True* so that only one progress bar is created. If *verbose=0*, no *Tqdm* callback will be loaded so that the trial produces no output. The default behaviour is *verbose=2*. For example, to suppress output we can use:

```
trial = Trial(model)
trial.run(10, verbose=0)
```

CHAPTER 2

The Metric API

There are a few levels of complexity to the metric API. You've probably already seen keys such as 'acc' and 'loss' can be used to reference pre-built metrics, so we'll have a look at how these get mapped 'under the hood'. We'll also take a look at how the metric *decorator API* can be used to construct powerful metrics which report running and terminal statistics. Finally, we'll take a closer look at the *MetricTree* and *MetricList* which make all of this happen internally.

2.1 Default Keys

In typical usage of torchbearer, we rarely interface directly with the metric API, instead just providing keys to the Model such as 'acc' and 'loss'. These keys are managed in a dict maintained by the decorator *default_for_key(key)*. Inside the torchbearer model, metrics are stored in an instance of *MetricList*, which is a wrapper that calls each metric in turn, collecting the results in a dict. If *MetricList* is given a string, it will look up the metric in the default metrics dict and use that instead. If you have defined a class that implements *Metric* and simply want to refer to it with a key, decorate it with *default_for_key()*.

2.2 Metric Decorators

Now that we have explained some of the basic aspects of the metric API, lets have a look at an example:

```
@default_for_key('binary_accuracy')
@default_for_key('binary_acc')
@running_mean
@mean
class BinaryAccuracy(Metric):
```

This is the definition of the default accuracy metric in torchbearer, let's break it down.

mean(), *std()* and *running_mean()* are all decorators which collect statistics about the underlying metric. *CategoricalAccuracy* simply returns a boolean tensor with an entry for each item in a batch. The *mean()* and *std()* decorators will take a mean / standard deviation value over the whole epoch (by keeping a sum and a number

of values). The *running_mean()* will collect a rolling mean for a given window size. That is, the running mean is only computed over the last 50 batches by default (however, this can be changed to suit your needs). Running metrics also have a step size, designed to reduce the need for constant computation when not a lot is changing. The default value of 10 means that the running mean is only updated every 10 batches.

Finally, the *default_for_key()* decorator is used to bind the metric to the keys 'acc' and 'accuracy'.

2.2.1 Lambda Metrics

One decorator we haven't covered is the *lambda_metric()*. This decorator allows you to decorate a function instead of a class. Here's another possible definition of the accuracy metric which uses a function:

```
@metrics.default_for_key('acc')
@metrics.running_mean
@metrics.std
@metrics.mean
@metrics.lambda_metric('acc', on_epoch=False)
def categorical_accuracy(y_pred, y_true):
   _, y_pred = torch.max(y_pred, 1)
   return (y_pred == y_true).float()
```

The *lambda_metric()* here converts the function into a `MetricFactory`. This can then be used in the normal way. By default and in our example, the lambda metric will execute the function with each batch of output (y_pred, y_true). If we set *on_epoch=True*, the decorator will use an *EpochLambda* instead of a *BatchLambda*. The *EpochLambda* collects the data over a whole epoch and then executes the metric at the end.

2.2.2 Metric Output - to_dict

At the root level, torchbearer expects metrics to output a dictionary which maps the metric name to the value. Clearly, this is not done in our accuracy function above as the aggregators expect input as numbers / tensors instead of dictionaries. We could change this and just have everything return a dictionary but then we would be unable to tell the difference between metrics we wish to display / log and intermediate stages (like the tensor output in our example above). Instead then, we have the *to_dict()* decorator. This decorator is used to wrap the output of a metric in a dictionary so that it will be picked up by the loggers. The aggregators all do this internally (with 'running_', '_std', etc. added to the name) so there's no need there, however, in case you have a metric that outputs precisely the correct value, the *to_dict()* decorator can make things a little easier.

2.3 Data Flow - The Metric Tree

Ok, so we've covered the *decorator API* and have seen how to implement all but the most complex metrics in torchbearer. Each of the decorators described above can be easily associated with one of the metric aggregator or wrapper classes so we won't go into that any further. Instead we'll just briefly explain the *MetricTree*. The *MetricTree* is a very simple tree implementation which has a root and some children. Each child could be another tree and so this supports trees of arbitrary depth. The main motivation of the metric tree is to co-ordinate data flow from some root metric (like our accuracy above) to a series of leaves (mean, std, etc.). When *Metric.process()* is called on a *MetricTree*, the output of the call from the root is given to each of the children in turn. The results from the children are then collected in a dictionary. The main reason for including this was to enable encapsulation of the different statistics without each one needing to compute the underlying metric individually. In theory the *MetricTree* means that vastly complex metrics could be computed for specific use cases, although I can't think of any right now...

CHAPTER 3

The Callback API

This note gives a very brief, high level overview of the callback API. For a much more detailed discussion and guides on specific callbacks see the example library.

3.1 Aims

The aim of callbacks is to enable the user to dynamically change the behaviour of the core training loop of a trial. To do this, callbacks are given state which is a dictionary containing all of the variables currently in use by a trial. Furthermore, callbacks can be made persistent by implementing the *state_dict* and *load_state_dict* methods, these will be automatically handled by `Trial.state_dict()` and `Trial.load_state_dict()`.

A key property of torchbearers construction is that the user is permitted to do more or less anything without triggering errors or warnings. That is, we assume all users are super users and that all actions are taken deliberately. A consequence of this approach is that some level of debugging skill can be required to get complex things to work correctly. However, the benefit of this approach is that torchbearer is a uniquely flexible library that never tries to stop you doing something you want to do. To understand further how the training process can be changed have a look at the callbacks example and the custom data loaders example

3.2 Fluent

Fluent interfaces are used in various places in torchbearer. In particular, many of our built-in callbacks employ method chaining and naturally readable method names in order to promote more readable code and a more usable API. A typical example would be the base `ImagingCallback` class. To see this in action have a look at the imaging example. For more information on fluent take a look at the original blog post.

CHAPTER 4

Using DistributedDataParallel with Torchbearer on CPU

This note will quickly cover how we can use torchbearer to train over multiple nodes. We shall do this by training a simple model to classify and for a massive amount of overkill we will be doing this on MNIST. Most of the code for this example is based off the Distributed Data Parallel (DDP) tutorial and the imagenet example from the PyTorch docs. We recommend you read at least the DDP tutorial before continuing with this note.

4.1 Setup, Cleanup and Model

We keep similar setup, cleanup and model from the DDP tutorial. All that is changed is taking rank, world size and master address from terminal arguments and changing the model to apply to MNIST. Note that we are keeping to the GLOO backend since this part of the note will be purely on the CPU.

```
def setup():
    os.environ['MASTER_ADDR'] = args.master
    os.environ['MASTER_PORT'] = '29500'

    # initialize the process group
    dist.init_process_group("gloo", rank=args.rank, world_size=args.world_size)

    # Explicitly setting seed makes sure that models created in two processes
    # start from same random weights and biases. Alternatively, sync models
    # on start with the callback below.
    #torch.manual_seed(42)


def cleanup():
    dist.destroy_process_group()


class ToyModel(nn.Module):
    def __init__(self):
        super(ToyModel, self).__init__()
        self.net1 = nn.Linear(784, 100)
```

(continues on next page)

(continued from previous page)

```
        self.relu = nn.ReLU()
        self.net2 = nn.Linear(100, 10)

    def forward(self, x):
        return self.net2(self.relu(self.net1(x)))
```

4.2 Sync Methods

Since we are working across multiple machines we need a way to synchronise the model itself and its gradients. To do this we utilise methods similar to that of the distributed applications tutorial from PyTorch.

```
def sync_model(model):
    size = float(dist.get_world_size())
    for param in model.parameters():
        dist.all_reduce(param.data, op=dist.ReduceOp.SUM)
        param.data /= size


def average_gradients(model):
    size = float(dist.get_world_size())
    for param in model.parameters():
        dist.all_reduce(param.grad.data, op=dist.ReduceOp.SUM)
        param.grad.data /= size
```

Since we require the gradients to be synced every step we implement both of these methods as Torchbearer callbacks. We sync the model itself on init and sync the gradients every step after the backward call.

```
@torchbearer.callbacks.on_init
def sync(state):
    sync_model(state[torchbearer.MODEL])


@torchbearer.callbacks.on_backward
def grad(state):
    average_gradients(state[torchbearer.MODEL])
```

4.3 Worker Function

Now we need to define the main worker function that each process will be running. We need this to setup the environment, actually run the training process and cleanup the environment after we finish. This function outside of calling *setup* and *cleanup* is exactly the same as any Torchbearer training function.

```
def worker():
    setup()
    print("Rank and node: {}-{}".format(args.rank, platform.node()))

    model = ToyModel().to('cpu')
    ddp_model = DDP(model)

    kwargs = {}
```

(continues on next page)

(continued from previous page)

```
    ds = datasets.MNIST('./data/mnist/', train=True, download=True,
         transform=transforms.Compose([
             transforms.ToTensor(),
             transforms.Normalize((0.1307,), (0.3081,))
          ]))

    train_sampler = torch.utils.data.distributed.DistributedSampler(ds)
    train_loader = torch.utils.data.DataLoader(ds,
        batch_size=128, sampler=train_sampler, **kwargs)

    test_ds = datasets.MNIST('./data/mnist', train=False,
              transform=transforms.Compose([
                 transforms.ToTensor(),
                 transforms.Normalize((0.1307,), (0.3081,))
                 ]))
    test_sampler = torch.utils.data.distributed.DistributedSampler(test_ds)
    test_loader = torch.utils.data.DataLoader(test_ds,
        batch_size=128, sampler=test_sampler,  **kwargs)

    loss_fn = nn.CrossEntropyLoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)

    trial = torchbearer.Trial(ddp_model, optimizer, loss_fn, metrics=['loss', 'acc'],
        callbacks=[sync, grad, flatten])
    trial.with_train_generator(train_loader)
    trial.run(10, verbose=2)

    print("Model hash: {}".format(hash(model)))
    print('First parameter: {}'.format(next(model.parameters())))

    cleanup()
```

You might have noticed that we had an extra flatten callback in the Trial, the only purpose of this was to flatten each image.

```
@torchbearer.callbacks.on_sample
def flatten(state):
    state[torchbearer.X] = state[torchbearer.X].view(state[torchbearer.X].shape[0], -
↪1)
```

4.4 Running

All we need to do now is write a *__main__* function to run the worker function.

```
if __name__ == "__main__":
    worker()
    print('done')
```

We can then ssh into each node on which we want to run the training and run the following code replacing i with the rank of each process.

```
python distributed_data_parallel.py --world-size 2 --rank i --host (host address)
```

4.5 Running on machines with GPUs

Coming soon.

4.6 Source Code

The source code for this example is given below:

Download Python source code: distributed_data_parallel.py

CHAPTER 5

Using the Tensorboard Callback

In this note we will cover the use of the *TensorBoard callback*. This is one of three callbacks in torchbearer which use the TensorboardX library. The PyPi version of tensorboardX (1.4) is somewhat outdated at the time of writing so it may be worth installing from source if some of the examples don't run correctly:

```
pip install git+https://github.com/lanpa/tensorboardX
```

The *TensorBoard callback* is simply used to log metric values (and optionally a model graph) to tensorboard. Let's have a look at some examples.

5.1 Setup

We'll begin with the data and simple model from our quickstart example.

```
BATCH_SIZE = 128

normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])

dataset = torchvision.datasets.CIFAR10(root='./data/cifar', train=True, download=True,
                                       transform=transforms.Compose([transforms.
↪ToTensor(), normalize]))
splitter = DatasetValidationSplitter(len(dataset), 0.1)
trainset = splitter.get_train_dataset(dataset)
valset = splitter.get_val_dataset(dataset)

traingen = torch.utils.data.DataLoader(trainset, pin_memory=True, batch_size=BATCH_
↪SIZE, shuffle=True, num_workers=10)
valgen = torch.utils.data.DataLoader(valset, pin_memory=True, batch_size=BATCH_SIZE,␣
↪shuffle=True, num_workers=10)


testset = torchvision.datasets.CIFAR10(root='./data/cifar', train=False,␣
↪download=True,
```

(continues on next page)

(continued from previous page)

```
                                        transform=transforms.Compose([transforms.
↪ToTensor(), normalize]))
testgen = torch.utils.data.DataLoader(testset, pin_memory=True, batch_size=BATCH_SIZE,
↪ shuffle=False, num_workers=10)
```

```
class SimpleModel(nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.convs = nn.Sequential(
            nn.Conv2d(3, 16, stride=2, kernel_size=3),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.Conv2d(16, 32, stride=2, kernel_size=3),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.Conv2d(32, 64, stride=2, kernel_size=3),
            nn.BatchNorm2d(64),
            nn.ReLU()
        )

        self.classifier = nn.Linear(576, 10)

    def forward(self, x):
        x = self.convs(x)
        x = x.view(-1, 576)
        return self.classifier(x)


model = SimpleModel()

optimizer = optim.Adam(filter(lambda p: p.requires_grad, model.parameters()), lr=0.
↪001)
loss = nn.CrossEntropyLoss()
```

The callback has three capabilities that we will demonstrate in this guide:

1. It can log a graph of the model
2. It can log the batch metrics
3. It can log the epoch metrics

5.2 Logging the Model Graph

One of the advantages of PyTorch is that it doesn't construct a model graph internally like other frameworks such as TensorFlow. This means that determining the model structure requires a forward pass through the model with some dummy data and parsing the subsequent graph built by autograd. Thankfully, TensorboardX can do this for us. The *TensorBoard callback* makes things a little easier by creating the dummy data for us and handling the interaction with TensorboardX. The size of the dummy data is chosen to match the size of the data in the dataset / data loader, this means that we need at least one batch of training data for the graph to be written. Let's train for one epoch just to see a model graph:

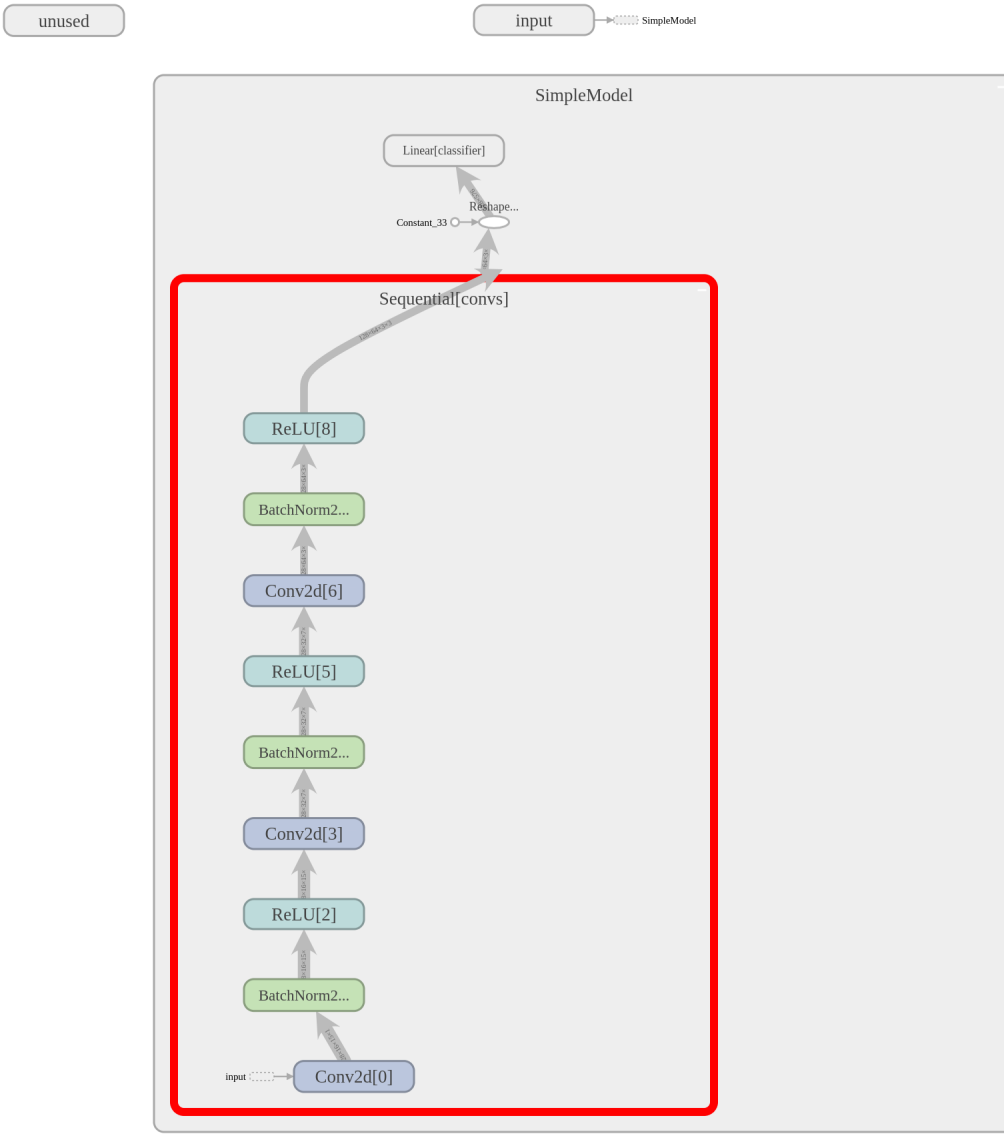
```
from torchbearer import Trial
from torchbearer.callbacks import TensorBoard
```

(continues on next page)

(continued from previous page)

```
torchbearer_trial = Trial(model, optimizer, loss, metrics=['acc', 'loss'],␣
↪callbacks=[TensorBoard(write_graph=True, write_batch_metrics=False, write_epoch_
↪metrics=False)]).to('cuda')
torchbearer_trial.with_generators(train_generator=traingen, val_generator=valgen)
torchbearer_trial.run(epochs=1)
```

To see the result, navigate to the project directory and execute the command `tensorboard --logdir logs`, then open a web browser and navigate to localhost:6006. After a bit of clicking around you should be able to see and download something like the following:



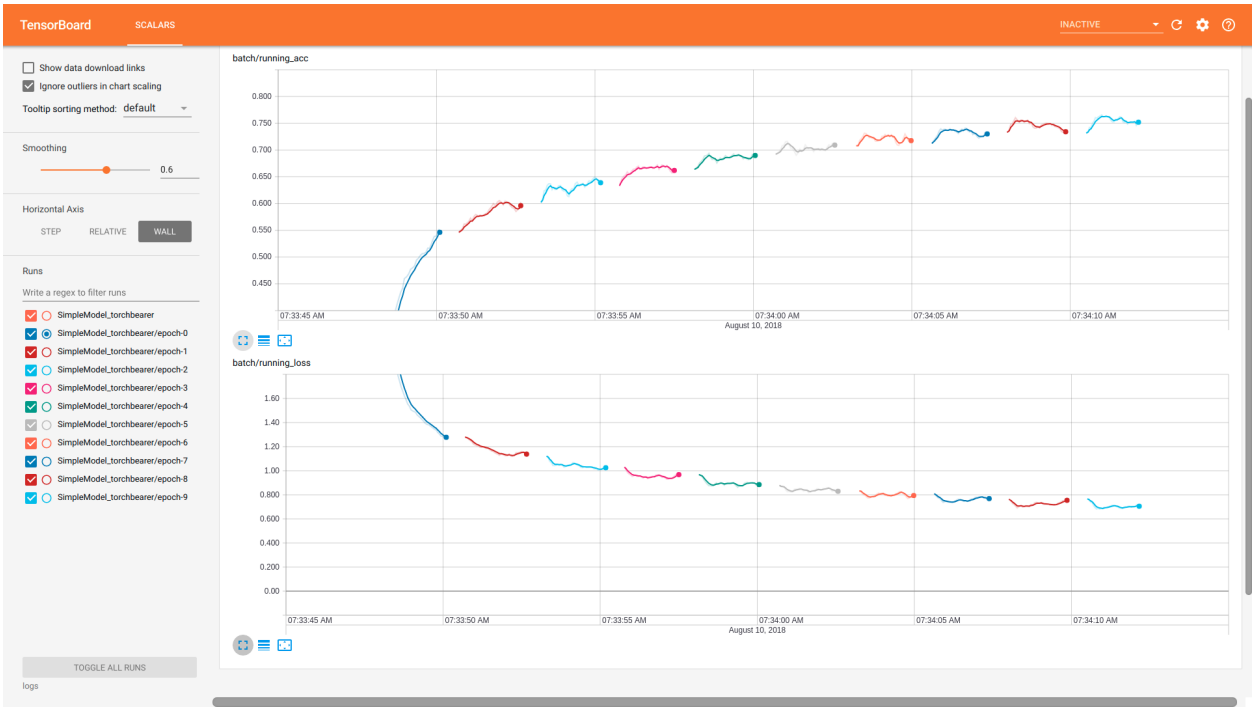
The dynamic graph construction does introduce some weirdness, however, this is about as good as model graphs typically get.

5.3 Logging Batch Metrics

If we have some metrics that output every batch, we might want to log them to tensorboard. This is useful particularly if epochs are long and we want to watch them progress. For this we can set `write_batch_metrics=True` in the *`TensorBoard callback`* constructor. Setting this flag will cause the batch metrics to be written as graphs to tensorboard. We are also able to change the frequency of updates by choosing the `batch_step_size`. This is the number of batches to wait between updates and can help with reducing the size of the logs, 10 seems reasonable. We run this for 10 epochs with the following:

```
torchbearer_trial = Trial(model, optimizer, loss, metrics=['acc', 'loss'],␣
↪callbacks=[TensorBoard(write_graph=False, write_batch_metrics=True, batch_step_
↪size=10, write_epoch_metrics=False)]).to('cuda')
torchbearer_trial.with_generators(train_generator=traingen, val_generator=valgen)
torchbearer_trial.run(epochs=10)
```

Runnng tensorboard again with `tensorboard --logdir logs`, navigating to localhost:6006 and selecting 'WALL' for the horizontal axis we can see the following:

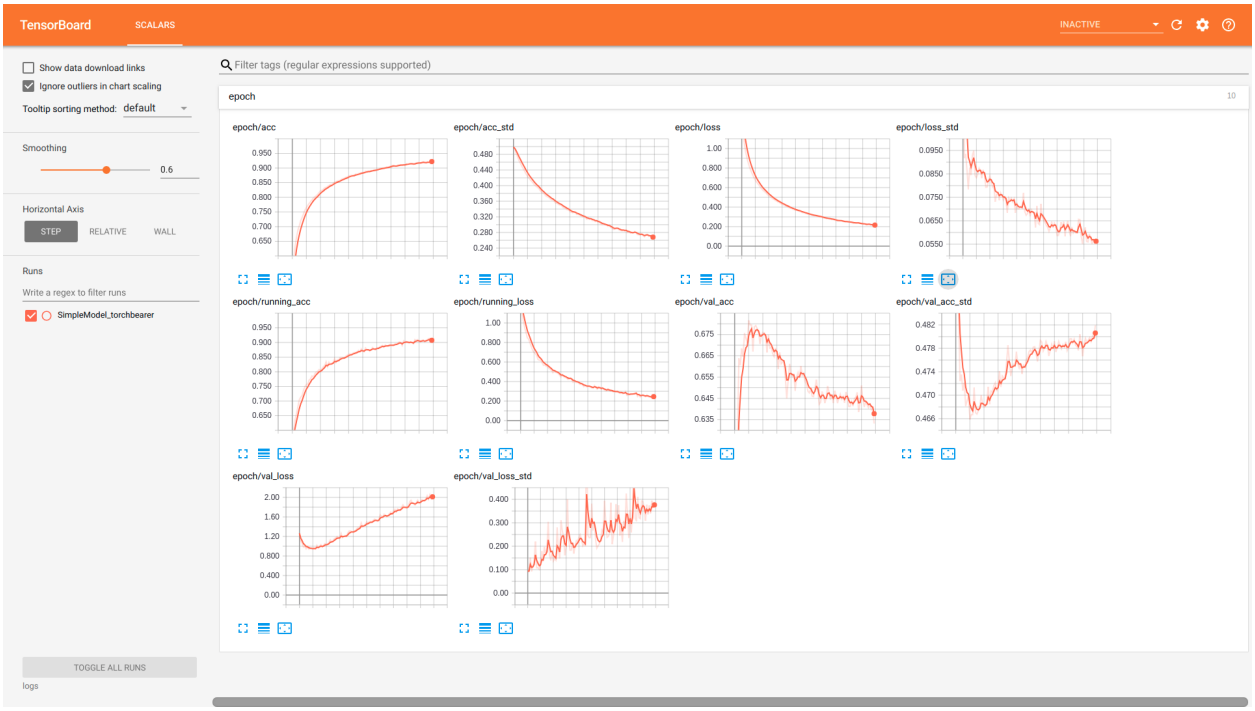


5.4 Logging Epoch Metrics

Logging epoch metrics is perhaps the most typical use case of TensorBoard and the *`TensorBoard callback`*. Using the same model as before, but setting `write_epoch_metrics=True` we can log epoch metrics with the following:

```
torchbearer_trial = Trial(model, optimizer, loss, metrics=['acc', 'loss'], ␣
↪callbacks=[TensorBoard(write_graph=False, write_batch_metrics=False, write_epoch_
↪metrics=True)]).to('cuda')
torchbearer_trial.with_generators(train_generator=traingen, val_generator=valgen)
torchbearer_trial.run(epochs=10)
```

Again, runnng tensorboard with `tensorboard --logdir logs` and navigating to localhost:6006 we see the following:



Note that we also get the batch metrics here. In fact this is the terminal value of the batch metric, which means that by default it is an average over the last 50 batches. This can be useful when looking at over-fitting as it gives a more accurate depiction of the model performance on the training data (the other training metrics are an average over the whole epoch despite model performance changing throughout).

5.5 Source Code

The source code for these examples is given below:

`Download Python source code: tensorboard.py`

CHAPTER 6

Logging to Visdom

In this note we will cover the use of the *TensorBoard callback* to log to visdom. See the tensorboard note for more on the callback in general.

6.1 Model Setup

We'll use the same setup as the tensorboard note.

```
BATCH_SIZE = 128

normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])

dataset = torchvision.datasets.CIFAR10(root='./data/cifar', train=True, download=True,
                                       transform=transforms.Compose([transforms.
↪ToTensor(), normalize]))
splitter = DatasetValidationSplitter(len(dataset), 0.1)
trainset = splitter.get_train_dataset(dataset)
valset = splitter.get_val_dataset(dataset)

traingen = torch.utils.data.DataLoader(trainset, pin_memory=True, batch_size=BATCH_
↪SIZE, shuffle=True, num_workers=10)
valgen = torch.utils.data.DataLoader(valset, pin_memory=True, batch_size=BATCH_SIZE,␣
↪shuffle=True, num_workers=10)


testset = torchvision.datasets.CIFAR10(root='./data/cifar', train=False,␣
↪download=True,
                                       transform=transforms.Compose([transforms.
↪ToTensor(), normalize]))
testgen = torch.utils.data.DataLoader(testset, pin_memory=True, batch_size=BATCH_SIZE,
↪ shuffle=False, num_workers=10)
```

```
class SimpleModel(nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.convs = nn.Sequential(
            nn.Conv2d(3, 16, stride=2, kernel_size=3),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.Conv2d(16, 32, stride=2, kernel_size=3),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.Conv2d(32, 64, stride=2, kernel_size=3),
            nn.BatchNorm2d(64),
            nn.ReLU()
        )

        self.classifier = nn.Linear(576, 10)

    def forward(self, x):
        x = self.convs(x)
        x = x.view(-1, 576)
        return self.classifier(x)


model = SimpleModel()

optimizer = optim.Adam(filter(lambda p: p.requires_grad, model.parameters()), lr=0.
↪001)
loss = nn.CrossEntropyLoss()
```

6.2 Logging Epoch and Batch Metrics

Visdom does not support logging model graphs so we shall start with logging epoch and batch metrics. The only change we need to make to the tensorboard example is setting `visdom=True` in the *TensorBoard callback* constructor.

```
torchbearer_trial = Trial(model, optimizer, loss, metrics=['acc', 'loss'],␣
↪callbacks=[TensorBoard(visdom=True, write_graph=True, write_batch_metrics=True,␣
↪batch_step_size=10, write_epoch_metrics=True)]).to('cuda')
torchbearer_trial.with_generators(train_generator=traingen, val_generator=valgen)
torchbearer_trial.run(epochs=5)
```

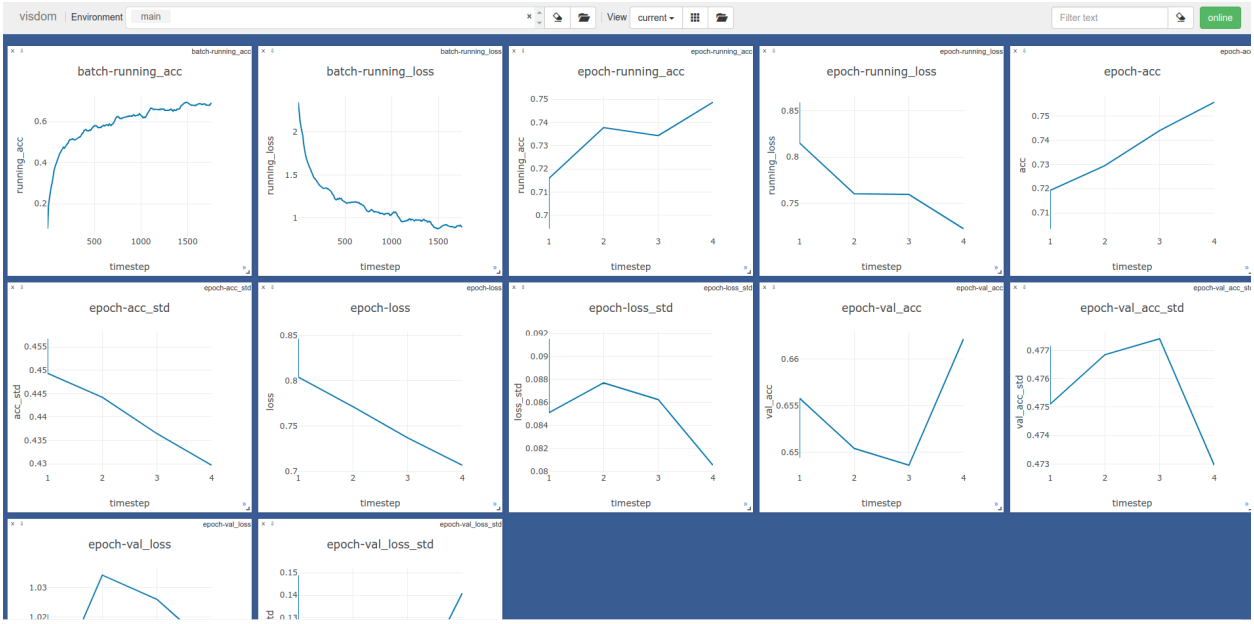
If your visdom server is running then you should see something similar to the figure below:

6.3 Visdom Client Parameters

The visdom client defaults to logging to localhost:8097 in the main environment however this is rather restrictive. We would like to be able to log to any server on any port and in any environment. To do this we need to edit the *VisdomParams* class.

```
class VisdomParams:
    """ ... """
    SERVER = 'http://localhost'
```

(continues on next page)



(continued from previous page)

```
    ENDPOINT = 'events'
    PORT = 8097
    IPV6 = True
    HTTP_PROXY_HOST = None
    HTTP_PROXY_PORT = None
    ENV = 'main'
    SEND = True
    RAISE_EXCEPTIONS = None
    USE_INCOMING_SOCKET = True
    LOG_TO_FILENAME = None
```

We first import the tensorboard file.

```
import torchbearer.callbacks.tensor_board as tensorboard
```

We can then edit the visdom client parameters, for example, changing the environment to "Test".

```
tensorboard.VisdomParams.ENV = 'Test'
```

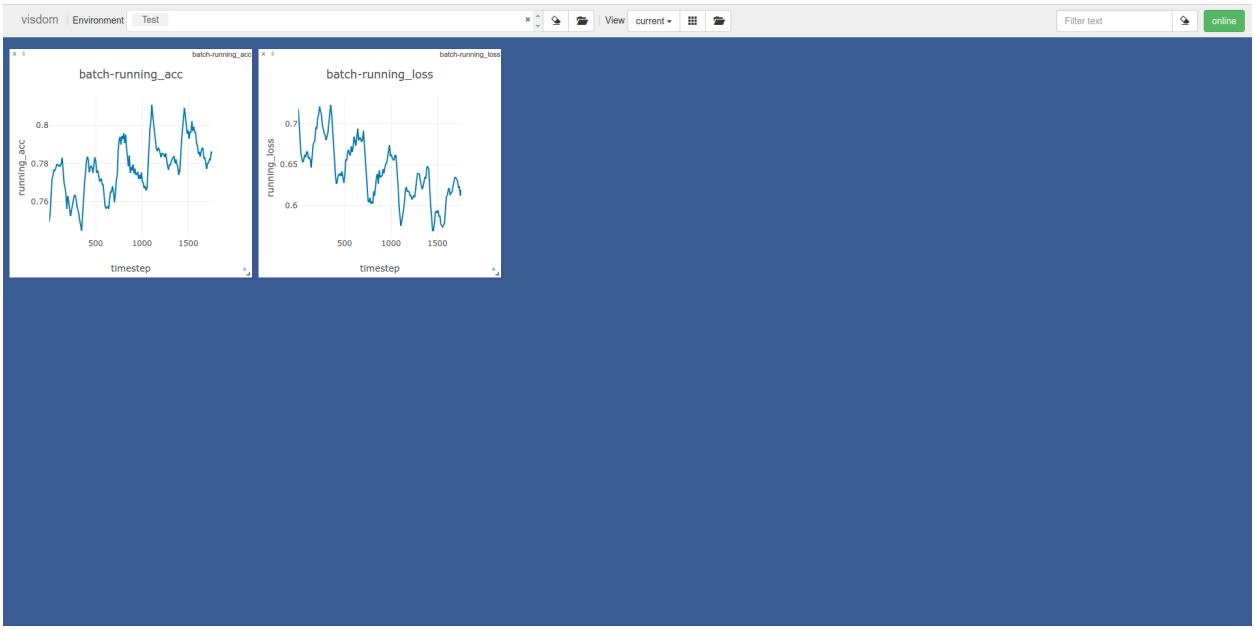
Running another fit call, we can see we are now logging to the "Test" environment.

The only paramenter that the *TensorBoard callback* sets explicity (and cannot be overrided) is the *LOG_TO_FILENAME* parameter. This is set to the *log_dir* given on the callback init.

6.4 Source Code

The source code for this example is given below:

Download Python source code: visdom.py


visdom
Environment
Test
View
current
Filter text
online
batch-running_acc
running_acc
0.8
0.78
0.76
500
1000
1500
timestep
batch-running_loss
running_loss
0.7
0.65
0.6
500
1000
1500
timestep

CHAPTER 7

torchbearer

7.1 Trial

class torchbearer.**Trial**(*model*, *optimizer=None*, *criterion=None*, *metrics=[]*, *callbacks=[]*, *verbose=2*)

The trial class contains all of the required hyper-parameters for model running in torchbearer and presents an API for model fitting, evaluating and predicting.

Example:

```
>>> import torch
>>> from torchbearer import Trial

# Example trial that attempts to minimise the output of a linear layer.
# Makes use of a callback to input the random data at each batch and a loss that␣
↪is the absolute value of the
# linear layer output. Runs for 10 iterations and a single epoch.
>>> model = torch.nn.Linear(2,1)
>>> optimiser = torch.optim.Adam(model.parameters(), lr=3e-4)

>>> @torchbearer.callbacks.on_sample
... def initial_data(state):
...     state[torchbearer.X] = torch.rand(1, 2)*10
>>> def minimise_output_loss(y_pred, y_true):
...     return torch.abs(y_pred)
>>> trial = Trial(model, optimiser, minimise_output_loss, ['loss'], [initial_
↪data]).for_steps(10).run(1)
```

```
@article{2018torchbearer,
  title={Torchbearer: A Model Fitting Library for PyTorch},
  author={Harris, Ethan and Painter, Matthew and Hare, Jonathon},
  journal={arXiv preprint arXiv:1809.03363},
  year={2018}
}
```

Parameters

- **model** (*torch.nn.Module*) – The base pytorch model
- **optimizer** (*torch.optim.Optimizer*) – The optimizer used for pytorch model weight updates
- **criterion** (*func / None*) – The final loss criterion that provides a loss value to the optimizer
- **metrics** (*list*) – The list of *torchbearer.Metric* instances to process during fitting
- **callbacks** (*list*) – The list of *torchbearer.Callback* instances to call during fitting
- **verbose** (*int*) – Global verbosity .If 2: use tqdm on batch, If 1: use tqdm on epoch, If 0: display no training progress

for_train_steps (*steps*)

Run this trial for the given number of training steps. Note that the generator will output (None, None) if it has not been set. Useful for differentiable programming. Returns self so that methods can be chained for convenience. If steps is larger than dataset size then loader will be refreshed like if it was a new epoch. If steps is -1 then loader will be refreshed until stopped by STOP_TRAINING flag or similar.

Example:

```
# Simple trial that runs for 100 training iterations, in this case optimising␣
↪nothing
>>> from torchbearer import Trial
>>> trial = Trial(None).for_train_steps(100)
```

Parameters **steps** (*int*) – The number of training steps per epoch to run.

Returns self

Return type *Trial*

with_train_generator (*generator*, *steps=None*)

Use this trial with the given train generator. Returns self so that methods can be chained for convenience.

Example:

```
# Simple trial that runs for 100 training iterations on the MNIST dataset
>>> from torchbearer import Trial
>>> from torchvision.datasets import MNIST
>>> from torch.utils.data import DataLoader
>>> dataloader = DataLoader(MNIST('./data/', train=True))
>>> trial = Trial(None).with_train_generator(dataloader).for_steps(100).run(1)
```

Parameters

- **generator** – The train data generator to use during calls to *run()*
- **steps** (*int*) – The number of steps per epoch to take when using this generator.

Returns self

Return type *Trial*

with_train_data(*x*, *y*, *batch_size=1*, *shuffle=True*, *num_workers=1*, *steps=None*)
Use this trial with the given train data. Returns self so that methods can be chained for convenience.

Example:

```
# Simple trial that runs for 10 training iterations on some random data
>>> from torchbearer import Trial
>>> data = torch.rand(10, 1)
>>> targets = torch.rand(10, 1)
>>> trial = Trial(None).with_val_data(data, targets).for_steps(10).run(1)
```

Parameters

- **x** (*torch.Tensor*) – The train x data to use during calls to *run()*
- **y** (*torch.Tensor*) – The train labels to use during calls to *run()*
- **batch_size** (*int*) – The size of each batch to sample from the data
- **shuffle** (*bool*) – If True, then data will be shuffled each epoch
- **num_workers** (*int*) – Number of worker threads to use in the data loader
- **steps** (*int*) – The number of steps per epoch to take when using this data

Returns self

Return type *Trial*

for_val_steps(*steps*)
Run this trial for the given number of validation steps. Note that the generator will output (None, None) if it has not been set. Useful for differentiable programming. Returns self so that methods can be chained for convenience. If steps larger than dataset size then loader will be refreshed like if it was a new epoch. If steps -1 then loader will be refreshed until stopped by STOP_TRAINING flag or similar.

Example:

```
# Simple trial that runs for 10 validation iterations on no data
>>> from torchbearer import Trial
>>> data = torch.rand(10, 1)
>>> trial = Trial(None).for_val_steps(10).run(1)
```

Parameters **steps** (*int*) – The number of validation steps per epoch to run

Returns self

Return type *Trial*

with_val_generator(*generator*, *steps=None*)
Use this trial with the given validation generator. Returns self so that methods can be chained for convenience.

Example:

```
# Simple trial that runs for 100 validation iterations on the MNIST dataset
>>> from torchbearer import Trial
>>> from torchvision.datasets import MNIST
>>> from torch.utils.data import DataLoader
>>> dataloader = DataLoader(MNIST('./data/', train=False))
>>> trial = Trial(None).with_val_generator(dataloader).for_steps(100).run(1)
```

Parameters

- **generator** – The validation data generator to use during calls to *run()* and *evaluate()*
- **steps** (*int*) – The number of steps per epoch to take when using this generator

Returns self

Return type *Trial*

with_val_data(*x*, *y*, *batch_size=1*, *shuffle=True*, *num_workers=1*, *steps=None*)

Use this trial with the given validation data. Returns self so that methods can be chained for convenience.

Example:

```
# Simple trial that runs for 10 validation iterations on some random data
>>> from torchbearer import Trial
>>> data = torch.rand(10, 1)
>>> targets = torch.rand(10, 1)
>>> trial = Trial(None).with_val_data(data, targets).for_steps(10).run(1)
```

Parameters

- **x** (*torch.Tensor*) – The validation x data to use during calls to *run()* and *evaluate()*
- **y** (*torch.Tensor*) – The validation labels to use during calls to *run()* and *evaluate()*
- **batch_size** (*int*) – The size of each batch to sample from the data
- **shuffle** (*bool*) – If True, then data will be shuffled each epoch
- **num_workers** (*int*) – Number of worker threads to use in the data loader
- **steps** (*int*) – The number of steps per epoch to take when using this data

Returns self

Return type *Trial*

for_test_steps(*steps*)

Run this trial for the given number of test steps. Note that the generator will output (None, None) if it has not been set. Useful for differentiable programming. Returns self so that methods can be chained for convenience. If steps larger than dataset size then loader will be refreshed like if it was a new epoch. If steps -1 then loader will be refreshed until stopped by STOP_TRAINING flag or similar.

Example:

```
# Simple trial that runs for 10 test iterations on some random data
>>> from torchbearer import Trial
>>> data = torch.rand(10, 1)
>>> trial = Trial(None).with_test_data(data).for_test_steps(10).run(1)
```

Parameters **steps** (*int*) – The number of test steps per epoch to run (when using *predict()*)

Returns self

Return type *Trial*

with_test_generator(*generator*, *steps=None*)
Use this trial with the given test generator. Returns self so that methods can be chained for convenience.

Example:

```
# Simple trial that runs for 10 test iterations on no data
>>> from torchbearer import Trial
>>> data = torch.rand(10, 1)
>>> trial = Trial(None).with_test_data(data).for_test_steps(10).run(1)
```

Parameters

- **generator** – The test data generator to use during calls to *predict()*
- **steps** (*int*) – The number of steps per epoch to take when using this generator

Returns self

Return type *Trial*

with_test_data(*x*, *batch_size=1*, *num_workers=1*, *steps=None*)
Use this trial with the given test data. Returns self so that methods can be chained for convenience.

Example:

```
# Simple trial that runs for 10 test iterations on some random data
>>> from torchbearer import Trial
>>> data = torch.rand(10, 1)
>>> trial = Trial(None).with_test_data(data).for_test_steps(10).run(1)
```

Parameters

- **x** (*torch.Tensor*) – The test x data to use during calls to *predict()*
- **batch_size** (*int*) – The size of each batch to sample from the data
- **num_workers** (*int*) – Number of worker threads to use in the data loader
- **steps** (*int*) – The number of steps per epoch to take when using this data

Returns self

Return type *Trial*

for_steps(*train_steps=None*, *val_steps=None*, *test_steps=None*)
Use this trial for the given number of train, val and test steps. Returns self so that methods can be chained for convenience. If steps larger than dataset size then loader will be refreshed like if it was a new epoch. If steps -1 then loader will be refreshed until stopped by STOP_TRAINING flag or similar.

Example:

```
# Simple trial that runs for 10 training, validation and test iterations on␣
↪some random data
>>> from torchbearer import Trial
>>> train_data = torch.rand(10, 1)
>>> val_data = torch.rand(10, 1)
>>> test_data = torch.rand(10, 1)
>>> trial = Trial(None).with_train_data(train_data).with_val_data(val_data).
↪with_test_data(test_data)
>>> trial.for_steps(10, 10, 10).run(1)
```

Parameters

- **train_steps** (*int*) – The number of training steps per epoch to run
- **val_steps** (*int*) – The number of validation steps per epoch to run
- **test_steps** (*int*) – The number of test steps per epoch to run (when using *predict()*)

Returns self

Return type *Trial*

with_generators(*train_generator=None*, *val_generator=None*, *test_generator=None*, *train_steps=None*, *val_steps=None*, *test_steps=None*)

Use this trial with the given generators. Returns self so that methods can be chained for convenience.

Example:

```
# Simple trial that runs for 100 steps from a training and validation data␣
↪generator
>>> from torchbearer import Trial
>>> from torchvision.datasets import MNIST
>>> from torch.utils.data import DataLoader
>>> trainloader = DataLoader(MNIST('./data/', train=True))
>>> valloader = DataLoader(MNIST('./data/', train=False))
>>> trial = Trial(None).with_generators(trainloader, valloader, train_
↪steps=100, val_steps=100).run(1)
```

Parameters

- **train_generator** – The training data generator to use during calls to *run()*
- **val_generator** – The validation data generator to use during calls to *run()* and *evaluate()*
- **test_generator** – The testing data generator to use during calls to *predict()*
- **train_steps** (*int*) – The number of steps per epoch to take when using the training generator
- **val_steps** (*int*) – The number of steps per epoch to take when using the validation generator
- **test_steps** (*int*) – The number of steps per epoch to take when using the testing generator

Returns self

Return type *Trial*

with_data(*x_train=None*, *y_train=None*, *x_val=None*, *y_val=None*, *x_test=None*, *batch_size=1*, *num_workers=1*, *train_steps=None*, *val_steps=None*, *test_steps=None*, *shuffle=True*)

Use this trial with the given data. Returns self so that methods can be chained for convenience.

Example:

```
# Simple trial that runs for 10 test iterations on some random data
>>> from torchbearer import Trial
>>> data = torch.rand(10, 1)
>>> targets = torch.rand(10, 1)
>>> test_data = torch.rand(10, 1)
```

(continues on next page)

(continued from previous page)

```
>>> trial = Trial(None).with_data(x_train=data, y_train=targets, x_test=test_
↪data)
>>> trial.for_test_steps(10).run(1)
```

Parameters

- **x_train** (*torch.Tensor*) – The training data to use
- **y_train** (*torch.Tensor*) – The training targets to use
- **x_val** (*torch.Tensor*) – The validation data to use
- **y_val** (*torch.Tensor*) – The validation targets to use
- **x_test** (*torch.Tensor*) – The test data to use
- **batch_size** (*int*) – Batch size to use in mini-batching
- **num_workers** (*int*) – Number of workers to use for data loading and batching
- **train_steps** (*int*) – Number of steps for each training pass
- **val_steps** (*int*) – Number of steps for each validation pass
- **test_steps** (*int*) – Number of steps for each test pass
- **shuffle** (*bool*) – If True, shuffle training and validation data.

Returns self

Return type *Trial*

for_inf_train_steps()

Use this trial with an infinite number of training steps (until stopped via STOP_TRAINING flag or similar). Returns self so that methods can be chained for convenience.

Example:

```
# Simple trial that runs training data until stopped
>>> from torchbearer import Trial
>>> from torchvision.datasets import MNIST
>>> from torch.utils.data import DataLoader
>>> trainloader = DataLoader(MNIST('./data/', train=True))
>>> trial = Trial(None).with_train_generator(trainloader).for_inf_train_
↪steps()
>>> trial.run(1)
```

Returns self

Return type *Trial*

for_inf_val_steps()

Use this trial with an infinite number of validation steps (until stopped via STOP_TRAINING flag or similar). Returns self so that methods can be chained for convenience.

Example:

```
# Simple trial that runs validation data until stopped
>>> from torchbearer import Trial
>>> from torchvision.datasets import MNIST
```

(continues on next page)

(continued from previous page)

```
>>> from torch.utils.data import DataLoader
>>> valloader = DataLoader(MNIST('./data/', train=False))
>>> trial = Trial(None).with_val_generator(valloader).for_inf_val_steps()
>>> trial.run(1)
```

Returns self

Return type *Trial*

for_inf_test_steps()

Use this trial with an infinite number of test steps (until stopped via STOP_TRAINING flag or similar). Returns self so that methods can be chained for convenience.

Example:

```
# Simple trial that runs test data until stopped
>>> from torchbearer import Trial
>>> test_data = torch.rand(1000, 10)
>>> trial = Trial(None).with_test_data(test_data).for_inf_test_steps()
>>> trial.run(1)
```

Returns self

Return type *Trial*

for_inf_steps(*train=True*, *val=True*, *test=True*)

Use this trail with infinite steps. Returns self so that methods can be chained for convenience.

Example:

```
# Simple trial that runs training and test data until stopped
>>> from torchbearer import Trial
>>> from torchvision.datasets import MNIST
>>> from torch.utils.data import DataLoader
>>> trainloader = DataLoader(MNIST('./data/', train=True))
>>> valloader = DataLoader(MNIST('./data/', train=False))
>>> trial = Trial(None).with_train_generator(trainloader).for_inf_
↪steps(valloader)
>>> trial.with_inf_test_loader(True, False, True).run(1)
```

Parameters

- **train** (*bool*) – Use an infinite number of training steps
- **val** (*bool*) – Use an infinite number of validation steps
- **test** (*bool*) – Use an infinite number of test steps

Returns self

Return type *Trial*

with_inf_train_loader()

Use this trial with a training iterator that refreshes when it finishes instead of each epoch. This allows for setting training steps less than the size of the generator and model will still be trained on all training samples if enough "epochs" are run.

Example:

```
# Simple trial that runs 10 epochs of 100 iterations of a training generator␣
↪without reshuffling until all data has been seen
>>> from torchbearer import Trial
>>> from torchvision.datasets import MNIST
>>> from torch.utils.data import DataLoader
>>> trainloader = DataLoader(MNIST('./data/', train=True))
>>> trial = Trial(None).with_train_generator(trainloader).with_inf_train_
↪loader()
>>> trial.run(10)
```

Returns self:

Return type *Trial*

with_loader(*batch_loader*)

Use this trial with custom batch loader. Usually calls next on state[torchbearer.ITERATOR] and populates state[torchbearer.X] and state[torchbearer.Y_TRUE]

Example:

```
# Simple trial that runs with a custom loader function that populates X and Y_
↪TRUE in state with random data
>>> from torchbearer import Trial
>>> def custom_loader(state):
...     state[X], state[Y_TRUE] = torch.rand(5, 5), torch.rand(5, 5)
>>> trial = Trial(None).with_loader(custom_loader)
>>> trial.run(10)
```

Parameters **batch_loader** (*function*) – Function of state that extracts data from data loader (stored under torchbearer.ITERATOR), stores it in state and sends it to the correct device

Returns self:

Return type *Trial*

with_closure(*closure*)

Use this trial with custom closure

Example:

```
# Simple trial that runs with a custom closure
>>> from torchbearer import Trial
>>> def custom_closure(state):
...     print(state[torchbearer.BATCH])
>>> trial = Trial(None).with_closure(custom_closure).for_steps(3)
>>> _ = trial.run(1)
0
1
2
```

Parameters **closure** (*function*) – Function of state that defines the custom closure

Returns self:

Return type *Trial*

run (*epochs=1*, *verbose=-1*)
Run this trial for the given number of epochs, starting from the last trained epoch.

Example:

```
# Simple trial that runs with a custom closure
>>> from torchbearer import Trial
>>> trial = Trial(None).for_steps(100)
>>> _ = trial.run(1)
```

Parameters

- **epochs** (*int, optional*) – The number of epochs to run for
- **verbose** (*int, optional*) – If 2: use tqdm on batch, If 1: use tqdm on epoch, If 0: display no training progress, If -1: Automatic

State Requirements:

- `torchbearer.state.MODEL`: Model should be callable and not none, set on Trial init

Returns The model history (list of tuple of steps summary and epoch metric dicts)

Return type list

evaluate (*verbose=-1*, *data_key=None*)
Evaluate this trial on the validation data.

Example:

```
# Simple trial to evaluate on both validation and test data
>>> from torchbearer import Trial
>>> test_data = torch.rand(5, 5)
>>> val_data = torch.rand(5, 5)
>>> t = Trial(None).with_val_data(val_data).with_test_data(test_data)
>>> t.evaluate(data_key=torchbearer.VALIDATION_DATA).evaluate(data_
↪key=torchbearer.TEST_DATA)
```

Parameters

- **verbose** (*int*) – If 2: use tqdm on batch, If 1: use tqdm on epoch, If 0: display no training progress, If -1: Automatic
- **data_key** (*StateKey*) – Optional *StateKey* for the data to evaluate on. Default: torchbearer.VALIDATION_DATA

Returns The final metric values

Return type dict

predict (*verbose=-1*, *data_key=None*)
Determine predictions for this trial on the test data.

Example:

```
# Simple trial to predict on some validation and test data
>>> from torchbearer import Trial
>>> val_data = torch.rand(5, 5)
>>> test_data = torch.rand(5, 5)
```

(continues on next page)

(continued from previous page)

```
>>> t = Trial(None).with_test_data(test_data)
>>> test_predictions = t.predict(data_key=torchbearer.TEST_DATA)
```

Parameters

- **verbose** (*int*) – If 2: use tqdm on batch, If 1: use tqdm on epoch, If 0: display no training progress, If -1: Automatic
- **data_key** (StateKey) – Optional *StateKey* for the data to predict on. Default: torchbearer.TEST_DATA

Returns Model outputs as a list

Return type list

replay(*callbacks=None*, *verbose=2*, *one_batch=False*)

Replay the fit passes stored in history with given callbacks, useful when reloading a saved Trial. Note that only progress and metric information is populated in state during a replay.

Example:

```
>>> from torchbearer import Trial
>>> state = torch.load('some_state.pt')
>>> t = Trial(None).load_state_dict(state)
>>> t.replay()
```

Parameters

- **callbacks** (*list*) – List of callbacks to be run during the replay
- **verbose** (*int*) – If 2: use tqdm on batch, If 1: use tqdm on epoch, If 0: display no training progress
- **one_batch** (*bool*) – If True, only one batch per epoch is replayed. If False, all batches are replayed

Returns self

Return type *Trial*

train()

Set model and metrics to training mode.

Example: ::

```
>>> from torchbearer import Trial
>>> t = Trial(None).train()
```

Returns self

Return type *Trial*

eval()

Set model and metrics to evaluation mode

Example: ::

```
>>> from torchbearer import Trial
>>> t = Trial(None).eval()
```

Returns self

Return type *Trial*

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

Example: ::

```
>>> from torchbearer import Trial
>>> t = Trial(None).to('cuda:1')
```

Parameters

- **args** – See: torch.nn.Module.to
- **kwargs** – See: torch.nn.Module.to

Returns self

Return type *Trial*

cuda(*device=None*)
Moves all model parameters and buffers to the GPU.

Example: ::

```
>>> from torchbearer import Trial
>>> t = Trial(None).cuda()
```

Parameters **device** (*int*) – if specified, all parameters will be copied to that device

Returns self

Return type *Trial*

cpu()
Moves all model parameters and buffers to the CPU.

Example: ::

```
>>> from torchbearer import Trial
>>> t = Trial(None).cpu()
```

Returns self

Return type *Trial*

state_dict(***kwargs*)
Get a dict containing the model and optimizer states, as well as the model history.

Example: ::

```
>>> from torchbearer import Trial
>>> t = Trial(None)
>>> state = t.state_dict() # State dict that can now be saved with torch.
↪save
```

Parameters **kwargs** – See: torch.nn.Module.state_dict

Returns A dict containing parameters and persistent buffers.

Return type dict

load_state_dict(*state_dict*, *resume=True*, ***kwargs*)
Resume this trial from the given state. Expects that this trial was constructed in the same way. Optionally, just load the model state when resume=False.

Example: ::

```
>>> from torchbearer import Trial
>>> t = Trial(None)
>>> state = torch.load('some_state.pt')
>>> t.load_state_dict(state)
```

Parameters

- **state_dict** (*dict*) – The state dict to reload
- **resume** (*bool*) – If True, resume from the given state. Else, just load in the model weights.
- **kwargs** – See: torch.nn.Module.load_state_dict

Returns self

Return type *Trial*

7.1.1 Batch Loaders

torchbearer.trial.**load_batch_infinite**(*loader*)
Wraps a batch loader and refreshes the iterator once it has been completed.

Parameters **loader** – batch loader to wrap

torchbearer.trial.**load_batch_none**(*state*)
Load a none (none, none) tuple mini-batch into state

Parameters **state** (*dict*) – The current state dict of the Trial.

torchbearer.trial.**load_batch_predict**(*state*)
Load a prediction (input data, target) or (input data) mini-batch from iterator into state

Parameters **state** (*dict*) – The current state dict of the Trial.

torchbearer.trial.**load_batch_standard**(*state*)
Load a standard (input data, target) tuple mini-batch from iterator into state

Parameters **state** (*dict*) – The current state dict of the Trial.

7.1.2 Misc

torchbearer.trial.**deep_to**(*batch*, *device*, *dtype*)
Static method to call to() on tensors, tuples or dicts. All items will have *deep_to()* called

Example:

```
>>> import torch
>>> from torchbearer import deep_to
>>> example_dict = {'a': torch.ones(5)*2.1, 'b': torch.ones(1)*5.9}
>>> deep_to(example_dict, device='cpu', dtype=torch.int)
{'a': tensor([2, 2, 2, 2, 2], dtype=torch.int32), 'b': tensor([5], dtype=torch.
↪int32)}
```

Parameters

- **batch** (*tuple / list / torch.Tensor / dict*) – The mini-batch which requires a `to()` call
- **device** (*torch.device*) – The desired device of the batch
- **dtype** (*torch.dtype*) – The desired datatype of the batch

Returns The moved or casted batch

Return type tuple / list / torch.Tensor

`torchbearer.trial.`**update_device_and_dtype**(*state*, **args*, ***kwargs*)

Function gets data type and device values from the args / kwargs and updates state.

Parameters

- **state** (`State`) – The *State* to update
- **args** – Arguments to the `Trial.to()` function
- **kwargs** – Keyword arguments to the `Trial.to()` function

Returns state

7.2 State

The state is central in torchbearer, storing all of the relevant intermediate values that may be changed or replaced during model fitting. This module defines classes for interacting with state and all of the built in state keys used throughout torchbearer. The `state_key()` function can be used to create custom state keys for use in callbacks or metrics.

Example:

```
>>> from torchbearer import state_key
>>> MY_KEY = state_key('my_test_key')
```

7.2.1 State

class `torchbearer.state.`**State**

State dictionary that behaves like a python dict but accepts StateKeys

data

get_key(*statekey*)

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

class torchbearer.state.**StateKey**(*key*)
StateKey class that is a unique state key based on the input string key. State keys are also metrics which retrieve themselves from state.

Parameters **key** (*str*) – Base key

process(*state*)
Process the state and update the metric for one iteration.

Parameters **args** – Arguments given to the metric. If this is a root level metric, will be given state

Returns None, or the value of the metric for this batch

process_final(*state*)
Process the terminal state and output the final value of the metric.

Parameters **args** – Arguments given to the metric. If this is a root level metric, will be given state

Returns None or the value of the metric for this epoch

torchbearer.state.**state_key**(*key*)
Computes and returns a non-conflicting key for the state dictionary when given a seed key

Parameters **key** (*str*) – The seed key - basis for new state key

Returns New state key

Return type *StateKey*

7.2.2 Key List

torchbearer.state.**BACKWARD_ARGS = backward_args**
The optional arguments which should be passed to the backward call

torchbearer.state.**BATCH = t**
The current batch number

torchbearer.state.**CALLBACK_LIST = callback_list**
The *CallbackList* object which is called by the Trial

torchbearer.state.**CRITERION = criterion**
The criterion to use when model fitting

torchbearer.state.**DATA = data**
The string name of the current data

torchbearer.state.**DATA_TYPE = dtype**
The data type of tensors in use by the model, match this to avoid type issues

torchbearer.state.**DEVICE = device**
The device currently in use by the *Trial* and PyTorch model

torchbearer.state.**EPOCH = epoch**
The current epoch number

torchbearer.state.**FINAL_PREDICTIONS = final_predictions**
The key which maps to the predictions over the dataset when calling predict

torchbearer.state.**GENERATOR = generator**
The current data generator (DataLoader)

torchbearer.state.**HISTORY = history**
The history list of the Trial instance

torchbearer.state.**INF_TRAIN_LOADING = inf_train_loading**
Flag for refreshing of training iterator when finished instead of each epoch

torchbearer.state.**INPUT = x**
The current batch of inputs

torchbearer.state.**ITERATOR = iterator**
The current iterator

torchbearer.state.**LOADER = loader**
The batch loader which handles formatting data from each batch

torchbearer.state.**LOSS = loss**
The current value for the loss

torchbearer.state.**MAX_EPOCHS = max_epochs**
The total number of epochs to run for

torchbearer.state.**METRICS = metrics**
The metric dict from the current batch of data

torchbearer.state.**METRIC_LIST = metric_list**
The list of metrics in use by the *Trial*

torchbearer.state.**MIXUP_LAMBDA = mixup_lambda**
The lambda coefficient of the linear combination of inputs

torchbearer.state.**MIXUP_PERMUTATION = mixup_permutation**
The permutation of input indices for input mixup

torchbearer.state.**MODEL = model**
The PyTorch module / model that will be trained

torchbearer.state.**OPTIMIZER = optimizer**
The optimizer to use when model fitting

torchbearer.state.**PREDICTION = y_pred**
The current batch of predictions

torchbearer.state.**SAMPLER = sampler**
The sampler which loads data from the generator onto the correct device

torchbearer.state.**SELF = self**
A self refrence to the Trial object for persistence etc.

torchbearer.state.**STEPS = steps**
The current number of steps per epoch

torchbearer.state.**STOP_TRAINING = stop_training**
A flag that can be set to true to stop the current fit call

torchbearer.state.**TARGET = y_true**
The current batch of ground truth data

torchbearer.state.**TEST_DATA = test_data**
The flag representing test data

torchbearer.state.**TEST_GENERATOR = test_generator**
The test data generator in the Trial object

torchbearer.state.**TEST_STEPS = test_steps**
The number of test steps to take

torchbearer.state.**TIMINGS = timings**
The timings keys used by the timer callback

torchbearer.state.**TRAIN_DATA = train_data**
The flag representing train data

torchbearer.state.**TRAIN_GENERATOR = train_generator**
The train data generator in the Trial object

torchbearer.state.**TRAIN_STEPS = train_steps**
The number of train steps to take

torchbearer.state.**VALIDATION_DATA = validation_data**
The flag representing validation data

torchbearer.state.**VALIDATION_GENERATOR = validation_generator**
The validation data generator in the Trial object

torchbearer.state.**VALIDATION_STEPS = validation_steps**
The number of validation steps to take

torchbearer.state.**VERSION = torchbearer_version**
The torchbearer version

torchbearer.state.**X = x**
The current batch of inputs

torchbearer.state.**Y_PRED = y_pred**
The current batch of predictions

torchbearer.state.**Y_TRUE = y_true**
The current batch of ground truth data

7.3 Utilities

class torchbearer.cv_utils.**DatasetValidationSplitter**(*dataset_len*, *split_fraction*, *shuffle_seed=None*)
Generates training and validation split indicies for a given dataset length and creates training and validation datasets using these splits

Parameters

- **dataset_len** – The length of the dataset to be split into training and validation
- **split_fraction** – The fraction of the whole dataset to be used for validation
- **shuffle_seed** – Optional random seed for the shuffling process

get_train_dataset(*dataset*)
Creates a training dataset from existing dataset

Parameters **dataset** (*torch.utils.data.Dataset*) – Dataset to be split into a training dataset

Returns Training dataset split from whole dataset

Return type torch.utils.data.Dataset

get_val_dataset(*dataset*)
Creates a validation dataset from existing dataset

Args: dataset (torch.utils.data.Dataset): Dataset to be split into a validation dataset

Returns Validation dataset split from whole dataset

Return type torch.utils.data.Dataset

class torchbearer.cv_utils.**SubsetDataset**(*dataset*, *ids*)
Dataset that consists of a subset of a previous dataset

Parameters

- **dataset** (*torch.utils.data.Dataset*) – Complete dataset
- **ids** (*list*) – List of subset IDs

torchbearer.cv_utils.**get_train_valid_sets**(*x*, *y*, *validation_data*, *validation_split*, *shuffle=True*)
Generate validation and training datasets from whole dataset tensors

Parameters

- **x** (*torch.Tensor*) – Data tensor for dataset
- **y** (*torch.Tensor*) – Label tensor for dataset
- **validation_data** (*(torch.Tensor, torch.Tensor)*) – Optional validation data (x_val, y_val) to be used instead of splitting x and y tensors
- **validation_split** (*float*) – Fraction of dataset to be used for validation
- **shuffle** (*bool*) – If True randomize tensor order before splitting else do not randomize

Returns Training and validation datasets

torchbearer.cv_utils.**train_valid_splitter**(*x*, *y*, *split*, *shuffle=True*)
Generate training and validation tensors from whole dataset data and label tensors

Parameters

- **x** (*torch.Tensor*) – Data tensor for whole dataset
- **y** (*torch.Tensor*) – Label tensor for whole dataset
- **split** (*float*) – Fraction of dataset to be used for validation
- **shuffle** (*bool*) – If True randomize tensor order before splitting else do not randomize

Returns Training and validation tensors (training data, training labels, validation data, validation labels)

torchbearer.bases.**base_closure**(*x*, *model*, *y_pred*, *y_true*, *crit*, *loss*, *opt*)
Function to create a standard pytorch closure using objects taken from state under the given keys.

Parameters

- **x** – State key under which the input data is stored
- **model** – State key under which the pytorch model is stored
- **y_pred** – State key under which the predictions will be stored
- **y_true** – State key under which the targets are stored
- **crit** – State key under which the criterion function is stored (function of state or (y_pred, y_true))

- **loss** – State key under which the loss will be stored
- **opt** – State key under which the optimsiser is stored

Returns Standard closure function

Return type function

CHAPTER 8

torchbearer.callbacks

8.1 Base Classes

class torchbearer.bases.**Callback**
Base callback class.

Note: All callbacks should override this class.

state_dict()
Get a dict containing the callback state.

Returns A dict containing parameters and persistent buffers.

Return type dict

load_state_dict(*state_dict*)
Resume this callback from the given state. Expects that this callback was constructed in the same way.

Parameters **state_dict** (*dict*) – The state dict to reload

Returns self

Return type *Callback*

on_init(*state*)
Perform some action with the given state as context at the init of a trial instance

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_start(*state*)
Perform some action with the given state as context at the start of a model fit.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_start_epoch(*state*)
Perform some action with the given state as context at the start of each epoch.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_start_training(*state*)
Perform some action with the given state as context at the start of the training loop.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_sample(*state*)
Perform some action with the given state as context after data has been sampled from the generator.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_forward(*state*)
Perform some action with the given state as context after the forward pass (model output) has been completed.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_criterion(*state*)
Perform some action with the given state as context after the criterion has been evaluated.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_backward(*state*)
Perform some action with the given state as context after backward has been called on the loss.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_step_training(*state*)
Perform some action with the given state as context after step has been called on the optimiser.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_end_training(*state*)
Perform some action with the given state as context after the training loop has completed.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_start_validation(*state*)
Perform some action with the given state as context at the start of the validation loop.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_sample_validation(*state*)
Perform some action with the given state as context after data has been sampled from the validation generator.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_forward_validation(*state*)
Perform some action with the given state as context after the forward pass (model output) has been completed with the validation data.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_criterion_validation(*state*)
Perform some action with the given state as context after the criterion evaluation has been completed with the validation data.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_step_validation(*state*)
Perform some action with the given state as context at the end of each validation step.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_end_validation(*state*)
Perform some action with the given state as context at the end of the validation loop.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_end_epoch(*state*)
Perform some action with the given state as context at the end of each epoch.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_checkpoint(*state*)
Perform some action with the state after all other callbacks have completed at the end of an epoch and the history has been updated. Should only be used for taking checkpoints or snapshots and will only be called by the run method of Trial.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_end(*state*)
Perform some action with the given state as context at the end of the model fitting.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

class torchbearer.callbacks.callbacks.**CallbackList**(*callback_list*)
The *CallbackList* class is a wrapper for a list of callbacks which acts as a single *Callback* and internally calls each *Callback* in the given list in turn.

Parameters **callback_list** (*list*) – The list of callbacks to be wrapped. If the list contains a *CallbackList*, this will be unwrapped.

CALLBACK_STATES = 'callback_states'

CALLBACK_TYPES = 'callback_types'

state_dict()
Get a dict containing all of the callback states.

Returns A dict containing parameters and persistent buffers.

Return type dict

load_state_dict(*state_dict*)
Resume this callback list from the given state. Callbacks must be given in the same order for this to work.

Parameters **state_dict** (*dict*) – The state dict to reload

Returns self

Return type *CallbackList*

copy()

append(*callback_list*)

on_init(*state*)
Call on_init on each callback in turn with the given state.

Parameters **state** (*dict[str,any]*) – The current state dict of the *Trial*.

on_start(*state*)
Call on_start on each callback in turn with the given state.

Parameters **state** (*dict[str,any]*) – The current state dict of the *Trial*.

on_start_epoch(*state*)
Call on_start_epoch on each callback in turn with the given state.

Parameters **state** (*dict[str,any]*) – The current state dict of the *Trial*.

on_start_training(*state*)
Call on_start_training on each callback in turn with the given state.

Parameters **state** (*dict[str,any]*) – The current state dict of the *Trial*.

on_sample(*state*)
Call on_sample on each callback in turn with the given state.

Parameters **state** (*dict[str,any]*) – The current state dict of the *Trial*.

on_forward(*state*)
Call on_forward on each callback in turn with the given state.

Parameters **state** (*dict[str,any]*) – The current state dict of the *Trial*.

on_criterion(*state*)
Call on_criterion on each callback in turn with the given state.

Parameters **state** (*dict[str,any]*) – The current state dict of the *Trial*.

on_backward(*state*)
Call on_backward on each callback in turn with the given state.

Parameters **state** (*dict[str,any]*) – The current state dict of the *Trial*.

on_step_training(*state*)
Call on_step_training on each callback in turn with the given state.

Parameters **state** (*dict[str,any]*) – The current state dict of the *Trial*.

on_end_training(*state*)
Call on_end_training on each callback in turn with the given state.

Parameters **state** (*dict[str,any]*) – The current state dict of the *Trial*.

on_start_validation(*state*)
Call on_start_validation on each callback in turn with the given state.

Parameters **state** (*dict[str,any]*) – The current state dict of the *Trial*.

on_sample_validation(*state*)
Call on_sample_validation on each callback in turn with the given state.

Parameters **state** (*dict[str,any]*) – The current state dict of the *Trial*.

on_forward_validation(*state*)
Call on_forward_validation on each callback in turn with the given state.

Parameters **state** (*dict[str,any]*) – The current state dict of the *Trial*.

on_criterion_validation(*state*)
Call on_criterion_validation on each callback in turn with the given state.

Parameters **state** (*dict[str,any]*) – The current state dict of the *Trial*.

on_step_validation(*state*)
Call on_step_validation on each callback in turn with the given state.

Parameters **state** (*dict[str,any]*) – The current state dict of the *Trial*.

on_end_validation(*state*)
Call on_end_validation on each callback in turn with the given state.

Parameters **state** (*dict[str,any]*) – The current state dict of the *Trial*.

on_end_epoch(*state*)
Call on_end_epoch on each callback in turn with the given state.

Parameters **state** (*dict[str,any]*) – The current state dict of the *Trial*.

on_checkpoint(*state*)
Call on_checkpoint on each callback in turn with the given state.

Parameters **state** (*dict[str,any]*) – The current state dict of the *Trial*.

on_end(*state*)
Call on_end on each callback in turn with the given state.

Parameters **state** (*dict[str,any]*) – The current state dict of the *Trial*.

8.2 Imaging

8.2.1 Main Classes

class torchbearer.callbacks.imaging.imaging.**CachingImagingCallback**(*key=x*, *transform=None*, *num_images=16*)
The *CachingImagingCallback* is an *ImagingCallback* which caches batches of images from the given state key up to the required amount before passing this along with state to the implementing class, once per epoch.

Parameters

- **key** (StateKey) – The *StateKey* containing image data (tensor of size [b, c, w, h])
- **transform** (*callable, optional*) – A function/transform that takes in a Tensor and returns a transformed version. This will be applied to the image before it is sent to output.
- **num_images** – The number of images to cache

on_cache(*cache*, *state*)
This method should be implemented by the overriding class to return an image from the cache.

Parameters

- **cache** (*tensor*) – The collected cache of size (num_images, C, W, H)
- **state** (*dict*) – The trial state dict

Returns The processed image

class torchbearer.callbacks.imaging.imaging.**FromState**(*key*, *transform=None*, *decorator=None*)
The *FromState* callback is an *ImagingCallback* which retrieves and image from state when called. The number of times the function is called can be controlled with a provided decorator (once_per_epoch, only_if etc.)

Parameters

- **key** (StateKey) – The *StateKey* containing the image (tensor of size [c, w, h])
- **transform** (*callable, optional*) – A function/transform that takes in a Tensor and returns a transformed version. This will be applied to the image before it is sent to output.
- **decorator** – A function which will be used to wrap the callback function. once_per_epoch by default

on_batch(*state*)

class torchbearer.callbacks.imaging.imaging.**ImagingCallback**(*transform=None*)

The *ImagingCallback* provides a generic interface for callbacks which yield images that should be sent to a file, tensorboard, visdom etc. without needing bespoke code. This allows the user to easily define custom visualisations by only writing the code to produce the image.

Parameters **transform** (*callable, optional*) – A function/transform that takes in a Tensor and returns a transformed version. This will be applied to the image before it is sent to output.

cache(*num_images*)

Cache images **before** they are passed to handlers. Once per epoch, a single cache will be returned, containing the first *num_images* to be returned.

Parameters **num_images** (*int*) – The number of images to cache

Returns self

Return type *ImagingCallback*

make_grid(*nrow=8, padding=2, normalize=False, norm_range=None, scale_each=False, pad_value=0*)

Use *torchvision.utils.make_grid* to make a grid of the images being returned by this callback. Recommended for use alongside *cache*.

Parameters

- **nrow** – See torchvision.utils.make_grid
- **padding** – See torchvision.utils.make_grid
- **normalize** – See torchvision.utils.make_grid
- **norm_range** – See torchvision.utils.make_grid
- **scale_each** – See torchvision.utils.make_grid
- **pad_value** – See torchvision.utils.make_grid

Returns self

Return type *ImagingCallback*

on_batch(*state*)

on_test()

Process this callback for test batches

Returns self

Return type *ImagingCallback*

on_train()

Process this callback for training batches

Returns self

Return type *ImagingCallback*

on_val()

Process this callback for validation batches

Returns self

Return type *ImagingCallback*

process(*state*)

to_file(*filename*, *index=None*)
Send images from this callback to the given file

Parameters

- **filename** (*str*) – The filename to store the image to
- **index** (*int or list or None*) – If not None, only apply the handler on this index / list of indices

Returns self

Return type *ImagingCallback*

to_pyplot(*title=None*, *show=True*, *index=None*)
Show images from this callback with pyplot

Parameters

- **title** (*str or None*) – If not None, plt.title will be called with the given string
- **show** (*bool*) – If True (default), show will be called after each image is plotted
- **index** (*int or list or None*) – If not None, only apply the handler on this index / list of indices

Returns self

Return type *ImagingCallback*

to_state(*keys*, *index=None*)
Put images from this callback in state with the given key

Parameters

- **keys** (StateKey *or list[*StateKey*]*) – The state key or keys to use for the images
- **index** (*int or list or None*) – If not None, only apply the handler on this index / list of indices

Returns self

Return type *ImagingCallback*

to_tensorboard(*name='Image'*, *log_dir='./logs'*, *comment='torchbearer'*, *index=None*)
Direct images from this callback to tensorboard with the given parameters

Parameters

- **name** (*str*) – The name of the image
- **log_dir** (*str*) – The tensorboard log path for output
- **comment** (*str*) – Descriptive comment to append to path
- **index** (*int or list or None*) – if not None, only apply the handler on this index / list of indices

Returns self

Return type *ImagingCallback*

to_visdom(*name='Image'*, *log_dir='./logs'*, *comment='torchbearer'*, *visdom_params=None*, *index=None*)
Direct images from this callback to visdom with the given parameters

Parameters

- **name** (*str*) – The name of the image
- **log_dir** (*str*) – The visdom log path for output
- **comment** (*str*) – Descriptive comment to append to path
- **visdom_params** (*VisdomParams*) – Visdom parameter settings object, uses default if None
- **index** (*int or list or None*) – if not None, only apply the handler on this index / list of indices

Returns self

Return type *ImagingCallback*

with_handler(*handler*, *index=None*)

Append the given output handler to the list of handlers

Parameters

- **handler** – A function of image and state which stores the given image in some way
- **index** (*int or list or None*) – If not None, only apply the handler on this index / list of indices

Returns self

Return type *ImagingCallback*

class torchbearer.callbacks.imaging.imaging.**MakeGrid**(*key=x*, *transform=None*, *num_images=16*, *nrow=8*, *padding=2*, *normalize=False*, *norm_range=None*, *scale_each=False*, *pad_value=0*)

The *MakeGrid* callback is a *CachingImagingCallback* which calls make grid on the cache with the provided parameters.

Parameters

- **key** (*StateKey*) – The *StateKey* containing image data (tensor of size [b, c, w, h])
- **transform** (*callable, optional*) – A function/transform that takes in a Tensor and returns a transformed version. This will be applied to the image before it is sent to output.
- **num_images** – The number of images to cache
- **nrow** – See torchvision.utils.make_grid
- **padding** – See torchvision.utils.make_grid
- **normalize** – See torchvision.utils.make_grid
- **norm_range** – See torchvision.utils.make_grid
- **scale_each** – See torchvision.utils.make_grid
- **pad_value** – See torchvision.utils.make_grid

on_cache(*cache*, *state*)

This method should be implemented by the overriding class to return an image from the cache.

Parameters

- **cache** (*tensor*) – The collected cache of size (num_images, C, W, H)
- **state** (*dict*) – The trial state dict

Returns The processed image

8.2.2 Deep Inside Convolutional Networks

class torchbearer.callbacks.imaging.inside_cnns.**ClassAppearanceModel**(*nclasses*, *input_size*, *optimizer_factory=<function ClassAppearanceModel.<lambda>>*, *steps=256*, *logit_key=y_pred*, *target=-10*, *decay=0.01*, *verbose=0*, *in_transform=None*, *transform=None*)

The *ClassAppearanceModel* callback implements Figure 1 from Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. This is a simple gradient ascent on an image (initialised to zero) with a sum-squares regularizer. Internally this creates a new *Trial* instance which then performs the optimization.

```
@article{simonyan2013deep,
  title={Deep inside convolutional networks: Visualising image classification
↪models and saliency maps},
  author={Simonyan, Karen and Vedaldi, Andrea and Zisserman, Andrew},
  journal={arXiv preprint arXiv:1312.6034},
  year={2013}
}
```

Parameters

- **nclasses** (*int*) – The number of output classes
- **input_size** (*tuple*) – The size to use for the input image
- **optimizer_factory** – A function of parameters which returns an optimizer to use
- **logit_key** (StateKey) – *StateKey* storing the class logits
- **target** (*int*) – Target class for the optimisation or RANDOM
- **steps** (*int*) – Number of optimisation steps to take
- **decay** (*float*) – Lambda for the L2 decay on the image
- **verbose** (*int*) – Verbosity level to pass to the internal *Trial* instance
- **transform** (*callable, optional*) – A function/transform that takes in a Tensor and returns a transformed version. This will be applied to the image before it is sent to output

on_batch(*state*)

target_to_key(*key*)

torchbearer.callbacks.imaging.inside_cnns.**RANDOM = -10**
Flag that when passed as the target chosses a random target

8.3 Model Checkpointers

class torchbearer.callbacks.checkpointers.**Best**(*filepath='model.{epoch:02d}-{val_loss:.2f}.pt'*, *save_model_params_only=False*, *monitor='val_loss'*, *mode='auto'*, *period=1*, *min_delta=0*, *pickle_module=<sphinx.ext.autodoc.importer._MockObject object>*, *pickle_protocol=<sphinx.ext.autodoc.importer._MockObject object>*)

Model checkpointer which saves the best model according to the given configurations. *filepath* can contain named formatting options, which will be filled any values from state. For example: if *filepath* is *weights.{epoch:02d}-{val_loss:.2f}*, then the model checkpoints will be saved with the epoch number and the validation loss in the filename.

Example:

```
>>> from torchbearer.callbacks import Best
>>> from torchbearer import Trial
>>> import torch

# Example Trial (without optimiser or loss criterion) which uses this checkpointer
>>> model = torch.nn.Linear(1,1)
>>> checkpoint = Best('my_path.pt', monitor='val_acc', mode='max')
>>> trial = Trial(model, callbacks=[checkpoint], metrics=['acc'])
```

Parameters

- **filepath** (*str*) – Path to save the model file
- **save_model_params_only** (*bool*) – If *save_model_params_only=True*, only model parameters will be saved so that the results can be loaded into a PyTorch nn.Module. The other option, *save_model_params_only=False*, should be used only if the results will be loaded into a Torchbearer Trial object later.
- **monitor** (*str*) – Quantity to monitor
- **mode** (*str*) – One of {auto, min, max}. If *save_best_only=True*, the decision to overwrite the current save file is made based on either the maximization or the minimization of the monitored quantity. For *val_acc*, this should be *max*, for *val_loss* this should be *min*, etc. In *auto* mode, the direction is automatically inferred from the name of the monitored quantity.
- **period** (*int*) – Interval (number of epochs) between checkpoints
- **min_delta** (*float*) – If *save_best_only=True*, this is the minimum improvement required to trigger a save
- **pickle_module** (*module*) – The pickle module to use, default is 'torch.serialization.pickle'

- **pickle_protocol** (*int*) – The pickle protocol to use, default is 'torch.serialization.DEFAULT_PROTOCOL'

State Requirements:

- `torchbearer.state.MODEL`: Model should have the *state_dict* method
- `torchbearer.state.METRICS`: Metrics dictionary should exist, with the *monitor* key populated
- `torchbearer.state.SELF`: Self should be the *torchbearer.Trial* which is running this callback

load_state_dict(*state_dict*)
Resume this callback from the given state. Expects that this callback was constructed in the same way.

Parameters **state_dict** (*dict*) – The state dict to reload

Returns self

Return type *Callback*

on_checkpoint(*state*)
Perform some action with the state after all other callbacks have completed at the end of an epoch and the history has been updated. Should only be used for taking checkpoints or snapshots and will only be called by the run method of Trial.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_start(*state*)
Perform some action with the given state as context at the start of a model fit.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

state_dict()
Get a dict containing the callback state.

Returns A dict containing parameters and persistent buffers.

Return type dict

class torchbearer.callbacks.checkpointers.**Interval**(*filepath='model.{epoch:02d}-{val_loss:.2f}.pt'*, *save_model_params_only=False*, *period=1*, *on_batch=False*, *pickle_module=<sphinx.ext.autodoc.importer._MockObject object>*, *pickle_protocol=<sphinx.ext.autodoc.importer._MockObject object>*)
Model checkpointer which which saves the model every 'period' epochs to the given filepath. *filepath* can contain named formatting options, which will be filled any values from state. For example: if *filepath* is *weights.{epoch:02d}-{val_loss:.2f}*, then the model checkpoints will be saved with the epoch number and the validation loss in the filename.

Example:

```
>>> from torchbearer.callbacks import Interval
>>> from torchbearer import Trial
>>> import torch

# Example Trial (without optimiser or loss criterion) which uses this checkpointer
```

(continues on next page)

(continued from previous page)

```
>>> model = torch.nn.Linear(1,1)
>>> checkpoint = Interval('my_path.pt', period=100, on_batch=True)
>>> trial = Trial(model, callbacks=[checkpoint], metrics=['acc'])
```

Parameters

- **filepath** (*str*) – Path to save the model file
- **save_model_params_only** (*bool*) – If *save_model_params_only=True*, only model parameters will be saved so that the results can be loaded into a PyTorch nn.Module. The other option, *save_model_params_only=False*, should be used only if the results will be loaded into a Torchbearer Trial object later.
- **period** (*int*) – Interval (number of steps) between checkpoints
- **on_batch** (*bool*) – If true step each batch, if false step each epoch.
- **period** – Interval (number of epochs) between checkpoints
- **pickle_module** (*module*) – The pickle module to use, default is 'torch.serialization.pickle'
- **pickle_protocol** (*int*) – The pickle protocol to use, default is 'torch.serialization.DEFAULT_PROTOCOL'

State Requirements:

- `torchbearer.state.MODEL`: Model should have the *state_dict* method
- `torchbearer.state.METRICS`: Metrics dictionary should exist
- `torchbearer.state.SELF`: Self should be the *torchbearer.Trial* which is running this callback

load_state_dict(*state_dict*)

Resume this callback from the given state. Expects that this callback was constructed in the same way.

Parameters **state_dict** (*dict*) – The state dict to reload

Returns self

Return type *Callback*

on_checkpoint(*state*)

Perform some action with the state after all other callbacks have completed at the end of an epoch and the history has been updated. Should only be used for taking checkpoints or snapshots and will only be called by the run method of Trial.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

state_dict()

Get a dict containing the callback state.

Returns A dict containing parameters and persistent buffers.

Return type dict

torchbearer.callbacks.checkpointers.**ModelCheckpoint**(*filepath='model.{epoch:02d}-{val_loss:.2f}.pt'*, *save_model_params_only=False*, *monitor='val_loss'*, *save_best_only=False*, *mode='auto'*, *period=1*, *min_delta=0*)

Save the model after every epoch. *filepath* can contain named formatting options, which will be filled any values from state. For example: if *filepath* is *weights.{epoch:02d}-{val_loss:.2f}*, then the model checkpoints will be saved with the epoch number and the validation loss in the filename. The torch *Trial* will be saved to filename.

Example:

```
>>> from torchbearer.callbacks import ModelCheckpoint
>>> from torchbearer import Trial
>>> import torch

# Example Trial (without optimiser or loss criterion) which uses this checkpointer
>>> model = torch.nn.Linear(1,1)
>>> checkpoint = ModelCheckpoint('my_path.pt', monitor='val_acc', mode='max')
>>> trial = Trial(model, callbacks=[checkpoint], metrics=['acc'])
```

Parameters

- **filepath** (*str*) – Path to save the model file
- **save_model_params_only** (*bool*) – If *save_model_params_only=True*, only model parameters will be saved so that the results can be loaded into a PyTorch nn.Module. The other option, *save_model_params_only=False*, should be used only if the results will be loaded into a Torchbearer Trial object later.
- **monitor** (*str*) – Quantity to monitor
- **save_best_only** (*bool*) – If *save_best_only=True*, the latest best model according to the quantity monitored will not be overwritten
- **mode** (*str*) – One of {auto, min, max}. If *save_best_only=True*, the decision to overwrite the current save file is made based on either the maximization or the minimization of the monitored quantity. For *val_acc*, this should be *max*, for *val_loss* this should be *min*, etc. In *auto* mode, the direction is automatically inferred from the name of the monitored quantity.
- **period** (*int*) – Interval (number of epochs) between checkpoints
- **min_delta** (*float*) – If *save_best_only=True*, this is the minimum improvement required to trigger a save

State Requirements:

- `torchbearer.state.MODEL`: Model should have the *state_dict* method
- `torchbearer.state.METRICS`: Metrics dictionary should exist
- `torchbearer.state.SELF`: Self should be the *torchbearer.Trial* which is running this callback

class torchbearer.callbacks.checkpointers.**MostRecent**(*filepath='model.{epoch:02d}-{val_loss:.2f}.pt'*, *save_model_params_only=False*, *pickle_module=<sphinx.ext.autodoc.importer._MockObject>*, *pickle_protocol=<sphinx.ext.autodoc.importer._MockObject>*)

Model checkpointer which saves the most recent model to a given filepath. *filepath* can contain named formatting options, which will be filled any values from state. For example: if *filepath* is *weights.{epoch:02d}-{val_loss:.2f}*, then the model checkpoints will be saved with the epoch number and the validation loss in the filename.

Example:

```
>>> from torchbearer.callbacks import MostRecent
>>> from torchbearer import Trial
>>> import torch

# Example Trial (without optimiser or loss criterion) which uses this checkpointer
>>> model = torch.nn.Linear(1,1)
>>> checkpoint = MostRecent('my_path.pt')
>>> trial = Trial(model, callbacks=[checkpoint], metrics=['acc'])
```

Parameters

- **filepath** (*str*) – Path to save the model file
- **save_model_params_only** (*bool*) – If *save_model_params_only=True*, only model parameters will be saved so that the results can be loaded into a PyTorch nn.Module. The other option, *save_model_params_only=False*, should be used only if the results will be loaded into a Torchbearer Trial object later.
- **pickle_module** (*module*) – The pickle module to use, default is 'torch.serialization.pickle'
- **pickle_protocol** (*int*) – The pickle protocol to use, default is 'torch.serialization.DEFAULT_PROTOCOL'

State Requirements:

- torchbearer.state.MODEL: Model should have the *state_dict* method
- torchbearer.state.METRICS: Metrics dictionary should exist
- torchbearer.state.SELF: Self should be the *torchbearer.Trial* which is running this callback

on_checkpoint(*state*)

Perform some action with the state after all other callbacks have completed at the end of an epoch and the history has been updated. Should only be used for taking checkpoints or snapshots and will only be called by the run method of Trial.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

8.4 Logging

class torchbearer.callbacks.csv_logger.**CSVLogger**(*filename*, *separator=','*, *batch_granularity=False*, *write_header=True*, *append=False*)

Callback to log metrics to a given csv file.

Example:

```
>>> from torchbearer.callbacks import CSVLogger
>>> from torchbearer import Trial
>>> import torch

# Example Trial (without optimiser or loss criterion) which writes metrics to a␣
↪csv file appending to previous content
>>> logger = CSVLogger('my_path.pt', separator=',', append=True)
>>> trial = Trial(None, callbacks=[logger], metrics=['acc'])
```

Parameters

- **filename** (*str*) – The name of the file to output to
- **separator** (*str*) – The delimiter to use (e.g. comma, tab etc.)
- **batch_granularity** (*bool*) – If True, write on each batch, else on each epoch
- **write_header** (*bool*) – If True, write the CSV header at the beginning of training
- **append** (*bool*) – If True, append to the file instead of replacing it

State Requirements:

- torchbearer.state.EPOCH: State should have the current epoch stored
- torchbearer.state.METRICS: Metrics dictionary should exist
- torchbearer.state.BATCH: State should have the current batch stored if using *batch_granularity*

on_end(*state*)

Perform some action with the given state as context at the end of the model fitting.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_end_epoch(*state*)

Perform some action with the given state as context at the end of each epoch.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_start(*state*)

Perform some action with the given state as context at the start of a model fit.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_step_training(*state*)

Perform some action with the given state as context after step has been called on the optimiser.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

class torchbearer.callbacks.printer.**ConsolePrinter**(*validation_label_letter='v'*, *precision=4*)

The ConsolePrinter callback simply outputs the training metrics to the console.

Example:

```
>>> import torch.nn
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import ConsolePrinter

# Example Trial which forgoes the usual printer for a console printer
>>> printer = ConsolePrinter()
>>> trial = Trial(None, callbacks=[printer], verbose=0).for_steps(1).run()
0/1(t):
```

Parameters

- **validation_label_letter** (*str*) – This is the letter displayed after the epoch number indicating the current phase of training
- **precision** (*int*) – Precision of the number format in decimal places

State Requirements:

- `torchbearer.state.EPOCH`: The current epoch number
- `torchbearer.state.MAX_EPOCHS`: The total number of epochs for this run
- `torchbearer.state.BATCH`: The current batch / iteration number
- `torchbearer.state.STEPS`: The total number of steps / batches / iterations for this epoch
- `torchbearer.state.METRICS`: The metrics dict to print

on_end_training(*state*)
Perform some action with the given state as context after the training loop has completed.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_end_validation(*state*)
Perform some action with the given state as context at the end of the validation loop.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_step_training(*state*)
Perform some action with the given state as context after step has been called on the optimiser.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_step_validation(*state*)
Perform some action with the given state as context at the end of each validation step.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

class torchbearer.callbacks.printer.**Tqdm**(*tqdm_module=None*, *validation_label_letter='v'*, *precision=4*, *on_epoch=False*, ***tqdm_args*)
The Tqdm callback outputs the progress and metrics for training and validation loops to the console using TQDM. The given key is used to label validation output.

Example:

```
>>> import torch.nn
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import Tqdm

# Example Trial which forgoes the usual printer for a customised tqdm printer.
```

(continues on next page)

(continued from previous page)

```
>>> printer = Tqdm(precision=8)
# Note that outputs are written to stderr, not stdout as shown in this example
>>> trial = Trial(None, callbacks=[printer], verbose=0).for_steps(1).run(1)
0/1(t): 100%|...| 1/1 [00:00<00:00, 29.40it/s]
```

Parameters

- **tqdm_module** – The tqdm module to use. If none, defaults to tqdm or tqdm_notebook if in notebook
- **validation_label_letter** (*str*) – The letter to use for validation outputs.
- **precision** (*int*) – Precision of the number format in decimal places
- **on_epoch** (*bool*) – If True, output a single progress bar which tracks epochs
- **tqdm_args** – Any extra keyword args provided here will be passed through to the tqdm module constructor. See github.com/tqdm/tqdm#parameters for more details.

State Requirements:

- `torchbearer.state.EPOCH`: The current epoch number
- `torchbearer.state.MAX_EPOCHS`: The total number of epochs for this run
- `torchbearer.state.STEPS`: The total number of steps / batches / iterations for this epoch
- `torchbearer.state.METRICS`: The metrics dict to print
- `torchbearer.state.HISTORY`: The history of the *Trial* object

on_end(*state*)
Perform some action with the given state as context at the end of the model fitting.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_end_epoch(*state*)
Perform some action with the given state as context at the end of each epoch.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_end_training(*state*)
Update the bar with the terminal training metrics and then close.

Parameters **state** (*dict*) – The *Trial* state

on_end_validation(*state*)
Update the bar with the terminal validation metrics and then close.

Parameters **state** (*dict*) – The *Trial* state

on_start(*state*)
Perform some action with the given state as context at the start of a model fit.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_start_training(*state*)
Initialise the TQDM bar for this training phase.

Parameters **state** (*dict*) – The *Trial* state

on_start_validation(*state*)
Initialise the TQDM bar for this validation phase.

Parameters **state** (*dict*) – The *Trial* state

on_step_training(*state*)
Update the bar with the metrics from this step.

Parameters **state** (*dict*) – The *Trial* state

on_step_validation(*state*)
Update the bar with the metrics from this step.

Parameters **state** (*dict*) – The *Trial* state

8.5 Tensorboard, Visdom and Others

class torchbearer.callbacks.tensor_board.**AbstractTensorBoard**(*log_dir='./logs'*, *comment='torchbearer'*, *visdom=False*, *visdom_params=None*)
TensorBoard callback which writes metrics to the given log directory. Requires the TensorboardX library for python.

Parameters

- **log_dir** (*str*) – The tensorboard log path for output
- **comment** (*str*) – Descriptive comment to append to path
- **visdom** (*bool*) – If true, log to visdom instead of tensorboard
- **visdom_params** (VisdomParams) – Visdom parameter settings object, uses default if None

State Requirements:

- torchbearer.state.MODEL: PyTorch model

static add_metric(*add_fn*, *tag*, *metric*, **args*, ***kwargs*)
Static method that recurses through *metric* until the *add_fn* can be applied. Useful when metric is an iterable of tensors so that the tensors can all be passed to an *add_fn* such as writer.add_scalar. For example, if passed *metric* as [[A, B], [C,], D, {'E': E}] then *add_fn* would be called on A, B, C, D and E and the respective tags (with base tag 'met') would be: met_0_0, met_0_1, met_1_0, met_2, met_E. Throws a warning if *add_fn* fails to parse a metric.

Parameters

- **add_fn** – Function to be called to log a metric, e.g. SummaryWriter.add_scalar
- **tag** – Tag under which to log the metric
- **metric** – Iterable of metrics.
- ***args** – Args for *add_fn*
- ****kwargs** – Keyword args for *add_fn*

Returns:

close_writer(*log_dir=None*)
Decrement the reference count for a writer belonging to the given log directory (or the default writer if the directory is not given). If the reference count gets to zero, the writer will be closed and removed.

Parameters **log_dir** (*str*) – the (optional) directory

get_writer(*log_dir=None*, *visdom=False*, *visdom_params=None*)
Get a SummaryWriter for the given directory (or the default writer if the directory is not given). If you are getting a *SummaryWriter* for a custom directory, it is your responsibility to close it using *close_writer*.

Parameters

- **log_dir** (*str*) – the (optional) directory
- **visdom** (*bool*) – If true, return VisdomWriter, if false return tensorboard SummaryWriter
- **visdom_params** (VisdomParams) – Visdom parameter settings object, uses default if None

Returns the *SummaryWriter* or *VisdomWriter*

on_end(*state*)
Perform some action with the given state as context at the end of the model fitting.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_start(*state*)
Perform some action with the given state as context at the start of a model fit.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

class torchbearer.callbacks.tensor_board.**TensorBoard**(*log_dir='./logs'*, *write_graph=True*, *write_batch_metrics=False*, *batch_step_size=10*, *write_epoch_metrics=True*, *comment='torchbearer'*, *visdom=False*, *visdom_params=None*)
TensorBoard callback which writes metrics to the given log directory. Requires the TensorboardX library for python.

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import TensorBoard
>>> import datetime
>>> current_time = datetime.now().strftime('%b%d_%H-%M-%S')
# Callback that will log to tensorboard under "(model name)_(current time)"
>>> tb = TensorBoard(log_dir='./logs', write_graph=False, comment=current_time)
# Trial that will run the callback and log accuracy and loss metrics
>>> t = Trial(None, callbacks=[tb], metrics=['acc', 'loss'])
```

Parameters

- **log_dir** (*str*) – The tensorboard log path for output
- **write_graph** (*bool*) – If True, the model graph will be written using the TensorboardX library
- **write_batch_metrics** (*bool*) – If True, batch metrics will be written
- **batch_step_size** (*int*) – The step size to use when writing batch metrics, make this larger to reduce latency

- **write_epoch_metrics** (*bool*) – If True, metrics from the end of the epoch will be written
- **comment** (*str*) – Descriptive comment to append to path
- **visdom** (*bool*) – If true, log to visdom instead of tensorboard
- **visdom_params** (VisdomParams) – Visdom parameter settings object, uses default if None

State Requirements:

- `torchbearer.state.MODEL`: PyTorch model
- `torchbearer.state.EPOCH`: State should have the current epoch stored
- `torchbearer.state.X`: State should have the current data stored if a model graph is to be built
- `torchbearer.state.BATCH`: State should have the current batch number stored if logging batch metrics
- `torchbearer.state.TRAIN_STEPS`: State should have the number of training steps stored
- `torchbearer.state.METRICS`: State should have a dictionary of metrics stored

on_end(*state*)
Perform some action with the given state as context at the end of the model fitting.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_end_epoch(*state*)
Perform some action with the given state as context at the end of each epoch.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_sample(*state*)
Perform some action with the given state as context after data has been sampled from the generator.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_start_epoch(*state*)
Perform some action with the given state as context at the start of each epoch.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_step_training(*state*)
Perform some action with the given state as context after step has been called on the optimiser.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_step_validation(*state*)
Perform some action with the given state as context at the end of each validation step.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

class torchbearer.callbacks.tensor_board.**TensorBoardImages**(*log_dir='./logs'*, *comment='torchbearer'*, *name='Image'*, *key=y_pred*, *write_each_epoch=True*, *num_images=16*, *nrow=8*, *padding=2*, *normalize=False*, *norm_range=None*, *scale_each=False*, *pad_value=0*, *visdom=False*, *visdom_params=None*)

The TensorBoardImages callback will write a selection of images from the validation pass to tensorboard using the TensorboardX library and torchvision.utils.make_grid (requires torchvision). Images are selected from the given key and saved to the given path. Full name of image sub directory will be model name + _ + comment.

Example:

```
>>> from torchbearer import Trial, state_key
>>> from torchbearer.callbacks import TensorBoardImages
>>> import datetime
>>> current_time = datetime.now().strftime('%b%d_%H-%M-%S')
>>> IMAGE_KEY = state_key('image_key')

>>> # Callback that will log to tensorboard under "(model name)_(current time)"
>>> tb = TensorBoardImages(comment=current_time, name='test_image', key=IMAGE_KEY)
>>> # Trial that will run log to tensorboard images stored under IMAGE_KEY
>>> t = Trial(None, callbacks=[tb], metrics=['acc', 'loss'])
```

Parameters

- **log_dir** (*str*) – The tensorboard log path for output
- **comment** (*str*) – Descriptive comment to append to path
- **name** (*str*) – The name of the image
- **key** (StateKey) – The key in state containing image data (tensor of size [c, w, h] or [b, c, w, h])
- **write_each_epoch** (*bool*) – If True, write data on every epoch, else write only for the first epoch.
- **num_images** (*int*) – The number of images to write
- **nrow** – See torchvision.utils.make_grid
- **padding** – See torchvision.utils.make_grid
- **normalize** – See torchvision.utils.make_grid
- **norm_range** – See torchvision.utils.make_grid
- **scale_each** – See torchvision.utils.make_grid
- **pad_value** – See torchvision.utils.make_grid
- **visdom** (*bool*) – If true, log to visdom instead of tensorboard
- **visdom_params** (VisdomParams) – Visdom parameter settings object, uses default if None

State Requirements:

- `torchbearer.state.EPOCH`: State should have the current epoch stored
- *key*: State should have images stored under the given state key

on_end_epoch(*state*)
Perform some action with the given state as context at the end of each epoch.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_step_validation(*state*)
Perform some action with the given state as context at the end of each validation step.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

class torchbearer.callbacks.tensor_board.**TensorBoardProjector**(*log_dir='./logs'*, *comment='torchbearer'*, *num_images=100*, *avg_pool_size=1*, *avg_data_channels=True*, *write_data=True*, *write_features=True*, *features_key=y_pred*)
The TensorBoardProjector callback is used to write images from the validation pass to Tensorboard using the TensorboardX library. Images are written to the given directory and, if required, so are associated features.

Parameters

- **log_dir** (*str*) – The tensorboard log path for output
- **comment** (*str*) – Descriptive comment to append to path
- **num_images** (*int*) – The number of images to write
- **avg_pool_size** (*int*) – Size of the average pool to perform on the image. This is recommended to reduce the overall image sizes and improve latency
- **avg_data_channels** (*bool*) – If True, the image data will be averaged in the channel dimension
- **write_data** (*bool*) – If True, the raw data will be written as an embedding
- **write_features** (*bool*) – If True, the image features will be written as an embedding
- **features_key** (StateKey) – The key in state to use for the embedding. Typically model output but can be used to show features from any layer of the model.

State Requirements:

- `torchbearer.state.EPOCH`: State should have the current epoch stored
- `torchbearer.state.X`: State should have the current data stored
- `torchbearer.state.Y_TRUE`: State should have the current targets stored

on_end_epoch(*state*)
Perform some action with the given state as context at the end of each epoch.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_step_validation(*state*)
Perform some action with the given state as context at the end of each validation step.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

class torchbearer.callbacks.tensor_board.**TensorBoardText**(*log_dir='./logs'*, *write_epoch_metrics=True*, *write_batch_metrics=False*, *log_trial_summary=True*, *batch_step_size=100*, *comment='torchbearer'*, *visdom=False*, *visdom_params=None*)
TensorBoard callback which writes metrics as text to the given log directory. Requires the TensorboardX library for python.

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import TensorBoardText
>>> import datetime
>>> current_time = datetime.now().strftime('%b%d_%H-%M-%S')
# Callback that will log to tensorboard under "(model name)_(current time)"
>>> tb = TensorBoardText(comment=current_time)
# Trial that will run the callback and log accuracy and loss metrics as text to␣
↪tensorboard
>>> t = Trial(None, callbacks=[tb], metrics=['acc', 'loss'])
```

Parameters

- **log_dir** (*str*) – The tensorboard log path for output
- **write_epoch_metrics** (*bool*) – If True, metrics from the end of the epoch will be written
- **log_trial_summary** (*bool*) – If True logs a string summary of the Trial
- **batch_step_size** (*int*) – The step size to use when writing batch metrics, make this larger to reduce latency
- **comment** (*str*) – Descriptive comment to append to path
- **visdom** (*bool*) – If true, log to visdom instead of tensorboard
- **visdom_params** (VisdomParams) – Visdom parameter settings object, uses default if None

State Requirements:

- torchbearer.state.SELF: The *torchbearer.Trial* running this callback
- torchbearer.state.EPOCH: State should have the current epoch stored
- torchbearer.state.BATCH: State should have the current batch number stored if logging batch metrics
- torchbearer.state.METRICS: State should have a dictionary of metrics stored

on_end(*state*)
Perform some action with the given state as context at the end of the model fitting.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_end_epoch(*state*)
Perform some action with the given state as context at the end of each epoch.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_start(*state*)
Perform some action with the given state as context at the start of a model fit.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_start_epoch(*state*)
Perform some action with the given state as context at the start of each epoch.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_step_training(*state*)
Perform some action with the given state as context after step has been called on the optimiser.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

static table_formatter(*string*)

class torchbearer.callbacks.tensor_board.**VisdomParams**
Class to hold visdom client arguments. Modify member variables before initialising tensorboard callbacks for custom arguments. See: visdom

ENDPOINT = 'events'

ENV = 'main'

HTTP_PROXY_HOST = None

HTTP_PROXY_PORT = None

IPV6 = True

LOG_TO_FILENAME = None

PORT = 8097

RAISE_EXCEPTIONS = None

SEND = True

SERVER = 'http://localhost'

USE_INCOMING_SOCKET = True

torchbearer.callbacks.tensor_board.**close_writer**(*log_dir*, *logger*)
Decrement the reference count for a writer belonging to a specific log directory. If the reference count gets to zero, the writer will be closed and removed.

Parameters

- **log_dir** (*str*) – the log directory
- **logger** – the object releasing the writer

torchbearer.callbacks.tensor_board.**get_writer**(*log_dir*, *logger*, *visdom=False*, *visdom_params=None*)
Get the writer assigned to the given log directory. If the writer doesn't exist it will be created, and a reference to the logger added.

Parameters

- **log_dir** (*str*) – the log directory

- **logger** – the object requesting the writer. That object should call *close_writer* when its finished
- **visdom** (*bool*) – if true VisdomWriter is returned instead of tensorboard SummaryWriter
- **visdom_params** (VisdomParams) – Visdom parameter settings object, uses default if None

Returns the *SummaryWriter* or *VisdomWriter* object

class torchbearer.callbacks.live_loss_plot.**LiveLossPlot**(*on_batch=False*, *batch_step_size=10*, *on_epoch=True*, *draw_once=False*, ***kwargs*)

Callback to write metrics to LiveLossPlot, a library for visualisation in notebooks

Example:

```
>>> import torch.nn
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import LiveLossPlot

# Example Trial which clips all model gradients norms at 2 under the L1 norm.
>>> model = torch.nn.Linear(1,1)
>>> live_loss_plot = LiveLossPlot()
>>> trial = Trial(model, callbacks=[live_loss_plot], metrics=['acc'])
```

Parameters

- **on_batch** (*bool*) – If True, batch metrics will be logged. Else batch metrics will not be logged
- **batch_step_size** (*int*) – The number of batches between logging metrics
- **on_epoch** (*bool*) – If True, epoch metrics will be logged every epoch. Else epoch metrics will not be logged
- **draw_once** (*bool*) – If True, draw the plot only at the end of training. Else draw every time metrics are logged
- **kwargs** – Keyword arguments for livelossplot.PlotLosses

State Requirements:

- `torchbearer.state.METRICS`: Metrics should be a dict containing the metrics to be plotted
- `torchbearer.state.BATCH`: Batch should be the current batch or iteration number in the epoch

on_end(*state*)

Perform some action with the given state as context at the end of the model fitting.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_start(*state*)

Perform some action with the given state as context at the start of a model fit.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

8.6 Early Stopping

class torchbearer.callbacks.early_stopping.**EarlyStopping**(*monitor='val_loss'*, *min_delta=0*, *patience=0*, *mode='auto'*, *step_on_batch=False*)

Callback to stop training when a monitored quantity has stopped improving.

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import EarlyStopping

# Example Trial which does early stopping if the validation accuracy drops below␣
↪the max seen for 5 epochs in a row
>>> stopping = EarlyStopping(monitor='val_acc', patience=5, mode='max')
>>> trial = Trial(None, callbacks=[stopping], metrics=['acc'])
```

Parameters

- **monitor** (*str*) – Name of quantity in metrics to be monitored
- **min_delta** (*float*) – Minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than min_delta, will count as no improvement.
- **patience** (*int*) – Number of epochs with no improvement after which training will be stopped.
- **mode** (*str*) – One of {auto, min, max}. In *min* mode, training will stop when the quantity monitored has stopped decreasing; in *max* mode it will stop when the quantity monitored has stopped increasing; in *auto* mode, the direction is automatically inferred from the name of the monitored quantity.

State Requirements:

- `torchbearer.state.METRICS`: Metrics should be a dict containing the given monitor key as a minimum

load_state_dict(*state_dict*)

Resume this callback from the given state. Expects that this callback was constructed in the same way.

Parameters **state_dict** (*dict*) – The state dict to reload

Returns self

Return type *Callback*

on_end_epoch(*state*)

on_step_training(*state*)

state_dict()

Get a dict containing the callback state.

Returns A dict containing parameters and persistent buffers.

Return type dict

step(*state*)

class torchbearer.callbacks.terminate_on_nan.**TerminateOnNaN**(*monitor='running_loss'*)
Callback which montiors the given metric and halts training if its value is nan or inf.

Example:

```
>>> import torch.nn
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import TerminateOnNaN

# Example Trial which terminates on a NaN, forced by a separate callback.␣
↪Terminates on the 11th batch since
the running loss only updates every 10 iterations.
>>> term = TerminateOnNaN(monitor='running_loss')
>>> @torchbearer.callbacks.on_criterion
... def force_terminate(state):
...     if state[torchbearer.BATCH] == 5:
...         state[torchbearer.LOSS] = state[torchbearer.LOSS] * torch.
↪Tensor([float('NaN')])
>>> trial = Trial(None, callbacks=[term, force_terminate], metrics=['loss'],␣
↪verbose=2).for_steps(30).run(1)
Invalid running_loss, terminating
```

Parameters **monitor** (*str*) – The name of the metric to monitor

State Requirements:

- torchbearer.state.METRICS: Metrics should be a dict containing at least the key *monitor*

on_end_epoch(*state*)
Perform some action with the given state as context at the end of each epoch.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_step_training(*state*)
Perform some action with the given state as context after step has been called on the optimiser.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_step_validation(*state*)
Perform some action with the given state as context at the end of each validation step.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

8.7 Gradient Clipping

class torchbearer.callbacks.gradient_clipping.**GradientClipping**(*clip_value*, *params=None*)
GradientClipping callback, which uses 'torch.nn.utils.clip_grad_value_' to clip the gradients of the given parameters to the given value. If params is None they will be retrieved from state.

Example:

```
>>> import torch.nn
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import GradientClipping

# Example Trial which clips all model gradients at 2 under the L1 norm.
```

(continues on next page)

(continued from previous page)

```
>>> model = torch.nn.Linear(1,1)
>>> clip = GradientNormClipping(2, 1)
>>> trial = Trial(model, callbacks=[clip], metrics=['acc'])
```

Parameters

- **clip_value** (*float or int*) – maximum allowed value of the gradients The gradients are clipped in the range [-clip_value, clip_value]
- **params** (*Iterable[Tensor] or Tensor, optional*) – an iterable of Tensors or a single Tensor that will have gradients normalized, otherwise this is retrieved from state

State Requirements:

- `torchbearer.state.MODEL`: Model should have the *parameters* method

on_backward(*state*)
Between the backward pass (which computes the gradients) and the step call (which updates the parameters), clip the gradient.

Parameters **state** (*dict*) – The *Trial* state

on_start(*state*)
If params is None then retrieve from the model.

Parameters **state** (*dict*) – The *Trial* state

class torchbearer.callbacks.gradient_clipping.**GradientNormClipping**(*max_norm*, *norm_type=2*, *params=None*)
GradientNormClipping callback, which uses 'torch.nn.utils.clip_grad_norm_' to clip the gradient norms to the given value. If params is None they will be retrieved from state.

Example:

```
>>> import torch.nn
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import GradientNormClipping

# Example Trial which clips all model gradients norms at 2 under the L1 norm.
>>> model = torch.nn.Linear(1,1)
>>> clip = GradientNormClipping(2, 1)
>>> trial = Trial(model, callbacks=[clip], metrics=['acc'])
```

Parameters

- **max_norm** (*float or int*) – max norm of the gradients
- **norm_type** (*float or int*) – type of the used p-norm. Can be `'inf'` for infinity norm.
- **params** (*Iterable[Tensor] or Tensor, optional*) – an iterable of Tensors or a single Tensor that will have gradients normalized, otherwise this is retrieved from state

State Requirements:

- `torchbearer.state.MODEL`: Model should have the *parameters* method

on_backward(*state*)
Between the backward pass (which computes the gradients) and the step call (which updates the parameters), clip the gradient.

Parameters **state** (*dict*) – The *Trial* state

on_start(*state*)
If params is None then retrieve from the model.

Parameters **state** (*dict*) – The *Trial* state

8.8 Learning Rate Schedulers

class torchbearer.callbacks.torch_scheduler.**CosineAnnealingLR**(*T_max*, *eta_min=0*, *last_epoch=-1*, *step_on_batch=False*)

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import CosineAnnealingLR

>>> # Example scheduler which uses cosine learning rate annealing - see PyTorch␣
↪docs
>>> scheduler = MultiStepLR(milestones=[30,80], gamma=0.1)
>>> trial = Trial(None, callbacks=[scheduler], metrics=['loss'], verbose=2).for_
↪steps(10).run(1)
```

Parameters **step_on_batch** (*bool*) – If True, step will be called on each training iteration rather than on each epoch

See: PyTorch CosineAnnealingLR

class torchbearer.callbacks.torch_scheduler.**CyclicLR**(*base_lr*, *max_lr*, *monitor='val_loss'*, *step_size_up=2000*, *step_size_down=None*, *mode='triangular'*, *gamma=1.0*, *scale_fn=None*, *scale_mode='cycle'*, *cycle_momentum=True*, *base_momentum=0.8*, *max_momentum=0.9*, *last_epoch=-1*, *step_on_batch=False*)

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import CyclicLR

>>> # Example scheduler which cycles the learning rate between 0.01 and 0.1
>>> scheduler = CyclicLR(0.01, 0.1)
>>> trial = Trial(None, callbacks=[scheduler], metrics=['loss'], verbose=2).for_
↪steps(10).for_val_steps(10).run(1)
```

Parameters

- **monitor** (*str*) – The name of the quantity in metrics to monitor. (Default value = 'val_loss')
- **step_on_batch** (*bool*) – If True, step will be called on each training iteration rather than on each epoch

See: PyTorch ReduceLROnPlateau

class torchbearer.callbacks.torch_scheduler.**ExponentialLR**(*gamma*, *last_epoch=-1*, *step_on_batch=False*)

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import ExponentialLR

>>> # Example scheduler which multiplies the learning rate by 0.1 every epoch
>>> scheduler = ExponentialLR(gamma=0.1)
>>> trial = Trial(None, callbacks=[scheduler], metrics=['loss'], verbose=2).for_
↪steps(10).run(1)
```

Parameters **step_on_batch** (*bool*) – If True, step will be called on each training iteration rather than on each epoch

See: PyTorch ExponentialLR

class torchbearer.callbacks.torch_scheduler.**LambdaLR**(*lr_lambda*, *last_epoch=-1*, *step_on_batch=False*)

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import LambdaLR

# Example Trial which performs the two learning rate lambdas from the PyTorch docs
>>> lambda1 = lambda epoch: epoch // 30
>>> lambda2 = lambda epoch: 0.95 ** epoch
>>> scheduler = LambdaLR(lr_lambda=[lambda1, lambda2])
>>> trial = Trial(None, callbacks=[scheduler], metrics=['loss'], verbose=2).for_
↪steps(10).run(1)
```

Parameters **step_on_batch** (*bool*) – If True, step will be called on each training iteration rather than on each epoch

See: PyTorch LambdaLR

class torchbearer.callbacks.torch_scheduler.**MultiStepLR**(*milestones*, *gamma=0.1*, *last_epoch=-1*, *step_on_batch=False*)

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import MultiStepLR

>>> # Assuming optimizer uses lr = 0.05 for all groups
>>> # lr = 0.05     if epoch < 30
```

(continues on next page)

(continued from previous page)

```
>>> # lr = 0.005     if 30 <= epoch < 80
>>> # lr = 0.0005    if epoch >= 80
>>> scheduler = MultiStepLR(milestones=[30,80], gamma=0.1)
>>> trial = Trial(None, callbacks=[scheduler], metrics=['loss'], verbose=2).for_
↪steps(10).run(1)
```

Parameters **step_on_batch** (*bool*) – If True, step will be called on each training iteration rather than on each epoch

See: PyTorch MultiStepLR

class torchbearer.callbacks.torch_scheduler.**ReduceLROnPlateau**(*monitor='val_loss'*, *mode='min'*, *factor=0.1*, *patience=10*, *verbose=False*, *threshold=0.0001*, *threshold_mode='rel'*, *cooldown=0*, *min_lr=0*, *eps=1e-08*, *step_on_batch=False*)

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import ReduceLROnPlateau

>>> # Example scheduler which divides the learning rate by 10 on plateaus of 5␣
↪epochs without significant
>>> # validation loss decrease, in order to stop overshooting the local minima.␣
↪new_lr = lr * factor
>>> scheduler = ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=5)
>>> trial = Trial(None, callbacks=[scheduler], metrics=['loss'], verbose=2).for_
↪steps(10).for_val_steps(10).run(1)
```

Parameters

- **monitor** (*str*) – The name of the quantity in metrics to monitor. (Default value = 'val_loss')
- **step_on_batch** (*bool*) – If True, step will be called on each training iteration rather than on each epoch

See: PyTorch ReduceLROnPlateau

class torchbearer.callbacks.torch_scheduler.**StepLR**(*step_size*, *gamma=0.1*, *last_epoch=-1*, *step_on_batch=False*)

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import StepLR
```

(continues on next page)

(continued from previous page)

```
>>> # Assuming optimizer uses lr = 0.05 for all groups
>>> # lr = 0.05     if epoch < 30
>>> # lr = 0.005    if 30 <= epoch < 60
>>> # lr = 0.0005   if 60 <= epoch < 90
>>> scheduler = StepLR(step_size=30, gamma=0.1)
>>> trial = Trial(None, callbacks=[scheduler], metrics=['loss'], verbose=2).for_
↪steps(10).run(1)
```

Parameters **step_on_batch** (*bool*) – If True, step will be called on each training iteration rather than on each epoch

See: PyTorch StepLR

class torchbearer.callbacks.torch_scheduler.**TorchScheduler**(*scheduler_builder*, *monitor=None*, *step_on_batch=False*)

on_end_epoch(*state*)
Perform some action with the given state as context at the end of each epoch.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_sample(*state*)
Perform some action with the given state as context after data has been sampled from the generator.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_start(*state*)
Perform some action with the given state as context at the start of a model fit.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_step_training(*state*)
Perform some action with the given state as context after step has been called on the optimiser.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

8.9 Weight Decay

class torchbearer.callbacks.weight_decay.**L1WeightDecay**(*rate=0.0005*, *params=None*)

WeightDecay callback which uses an L1 norm with the given rate and parameters. If params is None (default) then the parameters will be retrieved from the model.

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import L1WeightDecay

# Example Trial which runs a trial with weight decay on the model using an L1 norm
>>> decay = L1WeightDecay()
>>> trial = Trial(None, callbacks=[decay], metrics=['loss'], verbose=2).for_
↪steps(10).run(1)
```

Parameters

- **rate** (*float*) – The decay rate or lambda
- **params** (*Iterable[Tensor] or Tensor, optional*) – an iterable of Tensors or a single Tensor that will have gradients normalized, otherwise this is retrieved from state

State Requirements:

- `torchbearer.state.MODEL`: Model should have the *parameters* method
- `torchbearer.state.LOSS`: Loss should be a tensor that can be incremented

class torchbearer.callbacks.weight_decay.**L2WeightDecay**(*rate=0.0005*, *params=None*)

WeightDecay callback which uses an L2 norm with the given rate and parameters. If params is None (default) then the parameters will be retrieved from the model.

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import L2WeightDecay

# Example Trial which runs a trial with weight decay on the model using an L2 norm
>>> decay = L2WeightDecay()
>>> trial = Trial(None, callbacks=[decay], metrics=['loss'], verbose=2).for_
↪steps(10).run(1)
```

Parameters

- **rate** (*float*) – The decay rate or lambda
- **params** (*Iterable[Tensor] or Tensor, optional*) – an iterable of Tensors or a single Tensor that will have gradients normalized, otherwise this is retrieved from state

State Requirements:

- `torchbearer.state.MODEL`: Model should have the *parameters* method
- `torchbearer.state.LOSS`: Loss should be a tensor that can be incremented

class torchbearer.callbacks.weight_decay.**WeightDecay**(*rate=0.0005*, *p=2*, *params=None*)

Create a WeightDecay callback which uses the given norm on the given parameters and with the given decay rate. If params is None (default) then the parameters will be retrieved from the model.

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import WeightDecay

# Example Trial which runs a trial with weight decay on the model
>>> decay = WeightDecay()
>>> trial = Trial(None, callbacks=[decay], metrics=['loss'], verbose=2).for_
↪steps(10).run(1)
```

Parameters

- **rate** (*float*) – The decay rate or lambda
- **p** (*int*) – The norm level

- **params** (*Iterable[Tensor] or Tensor, optional*) – an iterable of Tensors or a single Tensor that will have gradients normalized, otherwise this is retrieved from state

State Requirements:

- `torchbearer.state.MODEL`: Model should have the *parameters* method
- `torchbearer.state.LOSS`: Loss should be a tensor that can be incremented

on_criterion(*state*)
Calculate the decay term and add to state['loss'].

Parameters **state** (*dict*) – The *Trial* state

on_start(*state*)
Retrieve params from state['model'] if required.

Parameters **state** (*dict*) – The *Trial* state

8.10 Weight / Bias Initialisation

class torchbearer.callbacks.init.**KaimingNormal**(*a=0, mode='fan_in', nonlinearity='leaky_relu', modules=None, targets=['Conv', 'Linear', 'Bilinear']*)

Kaiming Normal weight initialisation. Uses *torch.nn.init.kaiming_normal_* on the `weight` attribute of the filtered modules.

Example:

```
>>> import torch
>>> import torch.nn as nn
>>> from torchbearer import Trial
>>> from torchbearer.callbacks.init import KaimingNormal

# 100 random data points
>>> data = torch.rand(100, 3, 5, 5)
>>> example_batch = data[:3]
>>> initialiser = KaimingNormal()

# Model and trail using kaiming init for some random data
>>> model = nn.Sequential(nn.Conv2d(3, 1, 3), nn.ReLU())
>>> trial = Trial(model, callbacks=[initialiser]).with_train_data(data, data+5)
```

```
@inproceedings{he2015delving,
  title={Delving deep into rectifiers: Surpassing human-level performance on
↪imagenet classification},
  author={He, Kaiming and Zhang, Xiangyu and Ren, Shaoqing and Sun, Jian},
  booktitle={Proceedings of the IEEE international conference on computer vision},
  pages={1026--1034},
  year={2015}
}
```

Parameters

- **a** (*int*) – See PyTorch kaiming_normal_
- **mode** (*str*) – See PyTorch kaiming_normal_

- **nonlinearity** (*str*) – See PyTorch kaiming_normal_
- **modules** (*Iterable[nn.Module] or nn.Module, optional*) – an iterable of nn.Modules or a single nn.Module that will have weights initialised, otherwise this is retrieved from the model
- **targets** (*list[String]*) – A list of lookup strings to match which modules will be initialised

See: PyTorch kaiming_normal_

class torchbearer.callbacks.init.**KaimingUniform**(*a=0, mode='fan_in', nonlinearity='leaky_relu', modules=None, targets=['Conv', 'Linear', 'Bilinear']*)

Kaiming Uniform weight initialisation. Uses *torch.nn.init.kaiming_uniform_* on the `weight` attribute of the filtered modules.

Example:

```
>>> import torch
>>> import torch.nn as nn
>>> from torchbearer import Trial
>>> from torchbearer.callbacks.init import KaimingUniform

# 100 random data points
>>> data = torch.rand(100, 3, 5, 5)
>>> example_batch = data[:3]
>>> initialiser = KaimingUniform()

# Model and trail using kaiming init for some random data
>>> model = nn.Sequential(nn.Conv2d(3, 1, 3), nn.ReLU())
>>> trial = Trial(model, callbacks=[initialiser]).with_train_data(data, data+5)
```

```
@inproceedings{he2015delving,
  title={Delving deep into rectifiers: Surpassing human-level performance on
↪imagenet classification},
  author={He, Kaiming and Zhang, Xiangyu and Ren, Shaoqing and Sun, Jian},
  booktitle={Proceedings of the IEEE international conference on computer vision},
  pages={1026--1034},
  year={2015}
}
```

Parameters

- **a** (*int*) – See PyTorch kaiming_uniform_
- **mode** (*str*) – See PyTorch kaiming_uniform_
- **nonlinearity** (*str*) – See PyTorch kaiming_uniform_
- **modules** (*Iterable[nn.Module] or nn.Module, optional*) – an iterable of nn.Modules or a single nn.Module that will have weights initialised, otherwise this is retrieved from the model
- **targets** (*list[String]*) – A list of lookup strings to match which modules will be initialised

See: PyTorch kaiming_uniform_

class torchbearer.callbacks.init.**LsuvInit**(*data_item*, *weight_lambda=None*, *needed_std=1.0*, *std_tol=0.1*, *max_attempts=10*, *do_orthonorm=True*)

Layer-sequential unit-variance (LSUV) initialization as described in All you need is a good init and modified from the code by ducha-aiki. To be consistent with the paper, LsuvInit should be preceeded by a ZeroBias init on the Linear and Conv layers.

Example:

```
>>> import torch
>>> import torch.nn as nn
>>> from torchbearer import Trial
>>> from torchbearer.callbacks.init import LsuvInit

# 100 random data points
>>> data = torch.rand(100, 3, 5, 5)
>>> example_batch = data[:3]
>>> lsuv = LsuvInit(example_batch)

# Model and trail using lsuv init for some random data
>>> model = nn.Sequential(nn.Conv2d(3, 1, 3), nn.ReLU())
>>> trial = Trial(model, callbacks=[lsuv]).with_train_data(data, data+5)
```

```
@article{mishkin2015all,
  title={All you need is a good init},
  author={Mishkin, Dmytro and Matas, Jiri},
  journal={arXiv preprint arXiv:1511.06422},
  year={2015}
}
```

Parameters

- **data_item** (*torch.Tensor*) – A representative data item to put through the model
- **weight_lambda** (*lambda*) – A function that takes a module and returns the weight attribute. If none defaults to module.weight.
- **needed_std** – See paper, where needed_std is always 1.0
- **std_tol** – See paper, Tol_{var}
- **max_attempts** – See paper, T_{max}
- **do_orthonorm** – See paper, first pre-initialise with orthonormal matricies

State Requirements:

- torchbearer.state.MODEL: Model should have the *modules* method if modules is None

on_init(*state*)

Perform some action with the given state as context at the init of a trial instance

Parameters **state** (*dict*) – The current state dict of the *Trial*.

class torchbearer.callbacks.init.**WeightInit**(*initialiser=<function WeightInit.<lambda>>*, *modules=None*, *targets=['Conv', 'Linear', 'Bilinear']*)

Base class for weight initialisations. Performs the provided function for each module when on_init is called.

Parameters

- **initialiser** (*lambda*) – a function which initialises an nn.Module **inplace**
- **modules** (*Iterable[nn.Module] or nn.Module, optional*) – an iterable of nn.Modules or a single nn.Module that will have weights initialised, otherwise this is retrieved from the model
- **targets** (*list[String]*) – A list of lookup strings to match which modules will be initialised

State Requirements:

- `torchbearer.state.MODEL`: Model should have the *modules* method if modules is None

on_init (*state*)
Perform some action with the given state as context at the init of a trial instance

Parameters **state** (*dict*) – The current state dict of the *Trial*.

class `torchbearer.callbacks.init.`**XavierNormal**(*gain=1, modules=None, targets=['Conv', 'Linear', 'Bilinear']*)
Xavier Normal weight initialisation. Uses `torch.nn.init.xavier_normal_` on the `weight` attribute of the filtered modules.

Example:

```
>>> import torch
>>> import torch.nn as nn
>>> from torchbearer import Trial
>>> from torchbearer.callbacks.init import XavierNormal

# 100 random data points
>>> data = torch.rand(100, 3, 5, 5)
>>> example_batch = data[:3]
>>> initialiser = XavierNormal()

# Model and trail using Xavier init for some random data
>>> model = nn.Sequential(nn.Conv2d(3, 1, 3), nn.ReLU())
>>> trial = Trial(model, callbacks=[initialiser]).with_train_data(data, data+5)
```

```
@inproceedings{glorot2010understanding,
  title={Understanding the difficulty of training deep feedforward neural␣
↪networks},
  author={Glorot, Xavier and Bengio, Yoshua},
  booktitle={Proceedings of the thirteenth international conference on artificial␣
↪intelligence and statistics},
  pages={249--256},
  year={2010}
}
```

Parameters

- **gain** (*int*) – See PyTorch xavier_normal_
- **modules** (*Iterable[nn.Module] or nn.Module, optional*) – an iterable of nn.Modules or a single nn.Module that will have weights initialised, otherwise this is retrieved from the model
- **targets** (*list[String]*) – A list of lookup strings to match which modules will be initialised

See: PyTorch xavier_normal_

class torchbearer.callbacks.init.**XavierUniform**(*gain=1, modules=None, targets=['Conv', 'Linear', 'Bilinear']*)

Xavier Uniform weight initialisation. Uses `torch.nn.init.xavier_uniform_` on the `weight` attribute of the filtered modules.

Example:

```
>>> import torch
>>> import torch.nn as nn
>>> from torchbearer import Trial
>>> from torchbearer.callbacks.init import XavierUniform

# 100 random data points
>>> data = torch.rand(100, 3, 5, 5)
>>> example_batch = data[:3]
>>> initialiser = XavierUniform()

# Model and trail using Xavier init for some random data
>>> model = nn.Sequential(nn.Conv2d(3, 1, 3), nn.ReLU())
>>> trial = Trial(model, callbacks=[initialiser]).with_train_data(data, data+5)
```

```
@inproceedings{glorot2010understanding,
  title={Understanding the difficulty of training deep feedforward neural␣
↪networks},
  author={Glorot, Xavier and Bengio, Yoshua},
  booktitle={Proceedings of the thirteenth international conference on artificial␣
↪intelligence and statistics},
  pages={249--256},
  year={2010}
}
```

Parameters

- **gain** (*int*) – See PyTorch xavier_uniform_
- **modules** (*Iterable[nn.Module] or nn.Module, optional*) – an iterable of nn.Modules or a single nn.Module that will have weights initialised, otherwise this is retrieved from the model
- **targets** (*list[String]*) – A list of lookup strings to match which modules will be initialised

See: PyTorch xavier_uniform_

class torchbearer.callbacks.init.**ZeroBias**(*modules=None, targets=['Conv', 'Linear', 'Bilinear']*)

Zero initialisation for the `bias` attributes of filtered modules. This is recommended for use in conjunction with weight initialisation schemes.

Example:

```
>>> import torch
>>> import torch.nn as nn
>>> from torchbearer import Trial
>>> from torchbearer.callbacks.init import ZeroBias
```

(continues on next page)

(continued from previous page)

```
# 100 random data points
>>> data = torch.rand(100, 3, 5, 5)
>>> example_batch = data[:3]
>>> initialiser = ZeroBias()

# Model and trail using zero bias init for some random data
>>> model = nn.Sequential(nn.Conv2d(3, 1, 3), nn.ReLU())
>>> trial = Trial(model, callbacks=[initialiser]).with_train_data(data, data+5)
```

Parameters

- **modules** (*Iterable[nn.Module] or nn.Module, optional*) – an iterable of nn.Modules or a single nn.Module that will have weights initialised, otherwise this is retrieved from the model
- **targets** (*list[String]*) – A list of lookup strings to match which modules will be initialised

8.11 Regularisers

class torchbearer.callbacks.cutout.**Cutout**(*n_holes*, *length*, *constant=0.0*)

Cutout callback which randomly masks out patches of image data. Implementation a modified version of the code found here.

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import Cutout

# Example Trial which does Cutout regularisation
>>> cutout = Cutout(1, 10)
>>> trial = Trial(None, callbacks=[cutout], metrics=['acc'])
```

```
@article{devries2017improved,
  title={Improved regularization of convolutional neural networks with Cutout},
  author={DeVries, Terrance and Taylor, Graham W},
  journal={arXiv preprint arXiv:1708.04552},
  year={2017}
}
```

Parameters

- **n_holes** (*int*) – Number of patches to cut out of each image.
- **length** (*int*) – The length (in pixels) of each square patch.
- **constant** (*float*) – Constant value for each square patch

State Requirements:

- torchbearer.state.X: State should have the current data stored

on_sample(*state*)

Perform some action with the given state as context after data has been sampled from the generator.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

class torchbearer.callbacks.cutout.**RandomErase**(*n_holes*, *length*)

Random erase callback which replaces random patches of image data with random noise. Implementation a modified version of the cutout code found here.

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import RandomErase

# Example Trial which does Cutout regularisation
>>> erase = RandomErase(1, 10)
>>> trial = Trial(None, callbacks=[erase], metrics=['acc'])
```

```
@article{zhong2017random,
  title={Random erasing data augmentation},
  author={Zhong, Zhun and Zheng, Liang and Kang, Guoliang and Li, Shaozi and Yang,
↪ Yi},
  journal={arXiv preprint arXiv:1708.04896},
  year={2017}
}
```

Parameters

- **n_holes** (*int*) – Number of patches to cut out of each image.
- **length** (*int*) – The length (in pixels) of each square patch.

State Requirements:

- torchbearer.state.X: State should have the current data stored

on_sample(*state*)

Perform some action with the given state as context after data has been sampled from the generator.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

class torchbearer.callbacks.cutout.**CutMix**(*alpha*, *classes=-1*, *mixup_loss=False*)

Cutmix callback which replaces a random patch of image data with the corresponding patch from another image. This callback also converts labels to one hot before combining them according to the lambda parameters, sampled from a beta distribution as is done in the paper.

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import CutMix

# Example Trial which does CutMix regularisation
>>> cutmix = CutMix(1, classes=10)
>>> trial = Trial(None, callbacks=[cutmix], metrics=['acc'])
```

```
@article{yun2019cutmix,
  title={Cutmix: Regularization strategy to train strong classifiers with␣
↪localizable features},
  author={Yun, Sangdoo and Han, Dongyoon and Oh, Seong Joon and Chun, Sanghyuk␣
↪and Choe, Junsuk and Yoo, Youngjoon},
  journal={arXiv preprint arXiv:1905.04899},
  year={2019}
}
```

Parameters

- **alpha** (*float*) – The alpha value for the beta distribution.
- **classes** (*int*) – The number of classes for conversion to one hot.

State Requirements:

- `torchbearer.state.X`: State should have the current data stored
- `torchbearer.state.Y_TRUE`: State should have the current data stored

on_sample(*state*)
Perform some action with the given state as context after data has been sampled from the generator.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_sample_validation(*state*)
Perform some action with the given state as context after data has been sampled from the validation generator.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

class `torchbearer.callbacks.mixup.`**Mixup**(*alpha=1.0*, *lam=-10.0*)
Perform mixup on the model inputs. Requires use of `MixupInputs.loss()`, otherwise lambdas can be found in state under `MIXUP_LAMBDA`. Model targets will be a tuple containing the original target and permuted target.

Note: The accuracy metric for mixup is different on training to deal with the different targets, but for validation it is exactly the categorical accuracy, despite being called "val_mixup_acc"

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import Mixup

# Example Trial which does Mixup regularisation
>>> mixup = Mixup(0.9)
>>> trial = Trial(None, criterion=Mixup.mixup_loss, callbacks=[mixup], metrics=[
↪'acc'])
```

```
@inproceedings{zhang2018mixup,
  title={mixup: Beyond Empirical Risk Minimization},
  author={Hongyi Zhang and Moustapha Cisse and Yann N. Dauphin and David Lopez-
↪Paz},
  booktitle={International Conference on Learning Representations},
  year={2018}
}
```

Parameters

- **lam** (*float*) – Mixup inputs by fraction lam. If RANDOM, choose lambda from Beta(alpha, alpha). Else, lambda=lam
- **alpha** (*float*) – The alpha value to use in the beta distribution.

RANDOM = -10.0

static mixup_loss(*state*)
The standard cross entropy loss formulated for mixup (weighted combination of *F.cross_entropy*).

Parameters **state** – The current `Trial` state.

on_sample(*state*)
Perform some action with the given state as context after data has been sampled from the generator.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

class torchbearer.callbacks.between_class.**BCPlus**(*mixup_loss=False*, *alpha=1*, *classes=-1*)
BC+ callback which mixes images by treating them as waveforms. For standard BC, see *Mixup*. This callback can optionally convert labels to one hot before combining them according to the lambda parameters, sampled from a beta distribution, use alpha=1 to replicate the paper. Use with *BCPlus.bc_loss()* or set *mixup_loss* = *True* and use *Mixup.mixup_loss()*.

Note: This callback first sets all images to have zero mean. Consider adding an offset (e.g. 0.5) back before visualising.

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import BCPlus

# Example Trial which does BCPlus regularisation
>>> bcplus = BCPlus(classes=10)
>>> trial = Trial(None, criterion=BCPlus.bc_loss, callbacks=[bcplus], metrics=[
↪'acc'])
```

```
@inproceedings{tokozume2018between,
  title={Between-class learning for image classification},
  author={Tokozume, Yuji and Ushiku, Yoshitaka and Harada, Tatsuya},
  booktitle={Proceedings of the IEEE Conference on Computer Vision and Pattern␣
↪Recognition},
  pages={5486--5494},
  year={2018}
}
```

Parameters

- **mixup_loss** (*bool*) – If True, the lambda and targets will be stored for use with the mixup loss function.
- **alpha** (*float*) – The alpha value for the beta distribution.
- **classes** (*int*) – The number of classes for conversion to one hot.

State Requirements:

- `torchbearer.state.X`: State should have the current data stored and correctly normalised
- `torchbearer.state.Y_TRUE`: State should have the current data stored

static bc_loss(*state*)
The KL divergence between the outputs of the model and the ratio labels. Model ouputs should be unnormalised logits as this function performs a log_softmax.

Parameters **state** – The current `Trial` state.

on_sample(*state*)
Perform some action with the given state as context after data has been sampled from the generator.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_sample_validation(*state*)
Perform some action with the given state as context after data has been sampled from the validation generator.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

class torchbearer.callbacks.sample_pairing.**SamplePairing**(*policy=None*)
Perform SamplePairing on the model inputs. This is the process of averaging each image with another random image without changing the targets. The key here is to use the policy function to only do this some of the time.

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import SamplePairing

# Example Trial which does Sample Pairing regularisation with the policy from the␣
↪paper
>>> pairing = SamplePairing()
>>> trial = Trial(None, criterion=Mixup.loss, callbacks=[pairing], metrics=['acc
↪'])
```

```
@article{inoue2018data,
  title={Data augmentation by pairing samples for images classification},
  author={Inoue, Hiroshi},
  journal={arXiv preprint arXiv:1801.02929},
  year={2018}
}
```

Parameters **policy** – A function of state which returns True if the current batch should be paired.

static default_policy(*start_epoch*, *end_epoch*, *on_epochs*, *off_epochs*)
Return a policy which performs sample pairing according to the process defined in the paper.

Parameters

- **start_epoch** (*int*) – Epoch to start pairing on
- **end_epoch** (*int*) – Epoch to end pairing on (and begin fine-tuning)
- **on_epochs** (*int*) – Number of epochs to run sample pairing for before a break
- **off_epochs** (*int*) – Number of epochs to break for

Returns A policy function

on_sample(*state*)
Perform some action with the given state as context after data has been sampled from the generator.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

class torchbearer.callbacks.label_smoothing.**LabelSmoothingRegularisation**(*epsilon*, *classes=-1*)
Perform Label Smoothing Regularisation (LSR) on the targets during training. This involves converting the target to a one-hot vector and smoothing according to the value epsilon.

Note: Requires a multi-label loss, such as nn.BCELoss

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import LabelSmoothingRegularisation

# Example Trial which does label smoothing regularisation
>>> smoothing = LabelSmoothingRegularisation()
>>> trial = Trial(None, criterion=nn.BCELoss(), callbacks=[smoothing], metrics=[
↪'acc'])
```

```
@article{szegedy2015rethinking,
  title={Rethinking the inception architecture for computer vision. arXiv 2015},
  author={Szegedy, Christian and Vanhoucke, Vincent and Ioffe, Sergey and Shlens,␣
↪Jonathon and Wojna, Zbigniew},
  journal={arXiv preprint arXiv:1512.00567},
  volume={1512},
  year={2015}
}
```

Parameters

- **epsilon** (*float*) – The epsilon parameter from the paper
- **classes** (*int*) – The number of target classes, not required if the target is already one-hot encoded

on_sample(*state*)
Perform some action with the given state as context after data has been sampled from the generator.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_sample_validation(*state*)
Perform some action with the given state as context after data has been sampled from the validation generator.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

to_one_hot(*state*)

8.12 Unpack State

torchbearer.callbacks.**unpack_state**
alias of *torchbearer.callbacks.unpack_state*

8.13 Decorators

8.13.1 Main

The main callback decorators simply take a function and bind it to a callback point, returning the result.

torchbearer.callbacks.decorators.**on_init**(*func*)

The *on_init()* decorator is used to initialise a *Callback* with *on_init()* calling the decorated function

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import on_init

# Example callback on start
>>> @on_init
... def print_callback(state):
...     print('Initialised trial.')

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).run()
Initialised trial.
```

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *on_init()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**on_start**(*func*)

The *on_start()* decorator is used to initialise a *Callback* with *on_start()* calling the decorated function

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import on_start

# Example callback on start
>>> @on_start
... def print_callback(state):
...     print('Starting training.')

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).run()
Starting training.
```

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *on_start()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**on_start_epoch**(*func*)

The *on_start_epoch()* decorator is used to initialise a *Callback* with *on_start_epoch()* calling the decorated function

Example:

```
>>> import torchbearer
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import on_start_epoch

# Example callback running at start of each epoch
>>> @on_start_epoch
... def print_callback(state):
```

(continues on next page)

(continued from previous page)

```
...     print('Starting epoch {}.'.format(state[torchbearer.EPOCH]))

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).run()
Starting epoch 0.

Args:
func (function): The function(state) to *decorate*
```

Returns Initialised callback with *on_start_epoch()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**on_start_training**(*func*)

The *on_start_training()* decorator is used to initialise a *Callback* with *on_start_training()* calling the decorated function

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import on_start_training

# Example callback running at start of the training pass
>>> @on_start_training
... def print_callback(state):
...     print('Starting training.')

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).run()
Starting training.
```

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *on_start_training()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**on_sample**(*func*)

The *on_sample()* decorator is used to initialise a *Callback* with *on_sample()* calling the decorated function

Example:

```
>>> import torchbearer
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import on_sample

# Example callback running each time a sample is taken from the dataset
>>> @on_sample
... def print_callback(state):
...     print('Current sample {}.'.format(state[torchbearer.X]))

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).run()
Current sample None.
```

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *on_sample()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**on_forward**(*func*)

The *on_forward()* decorator is used to initialise a *Callback* with *on_forward()* calling the decorated function

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import on_forward

# Example callback running after each training forward pass of the torch model
>>> @on_forward
... def print_callback(state):
...     print('Evaluated training batch.')

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).run()
Evaluated training batch.
```

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *on_forward()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**on_criterion**(*func*)

The *on_criterion()* decorator is used to initialise a *Callback* with *on_criterion()* calling the decorated function

Example:

```
>>> import torchbearer
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import on_criterion

# Example callback running after each evaluation of the loss
>>> @on_criterion
... def print_callback(state):
...     print('Current loss {}.'.format(state[torchbearer.LOSS].item()))

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).run()
Current loss 0.0.
```

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *on_criterion()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**on_backward**(*func*)

The *on_backward()* decorator is used to initialise a *Callback* with *on_backward()* calling the decorated function

Example:

```
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import on_backward
```

(continues on next page)

(continued from previous page)

```
# Example callback running after each backward pass of the torch model
>>> @on_backward
... def print_callback(state):
...     print('Doing backward.')

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).run()
Doing backward.
```

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *on_backward()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**on_step_training**(*func*)

The *on_step_training()* decorator is used to initialise a *Callback* with *on_step_training()* calling the decorated function

Example:

```
>>> import torchbearer
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import on_step_training

# Example callback running after each training step
>>> @on_step_training
... def print_callback(state):
...     print('Step {}.'.format(state[torchbearer.BATCH]))

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).run()
Step 0.
```

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *on_step_training()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**on_end_training**(*func*)

The *on_end_training()* decorator is used to initialise a *Callback* with *on_end_training()* calling the decorated function

Example:

```
>>> import torchbearer
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import on_end_training

# Example callback running after each training pass
>>> @on_end_training
... def print_callback(state):
...     print('Finished training pass.')

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).run()
Finished training pass.
```

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *on_end_training()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**on_start_validation**(*func*)

The *on_start_validation()* decorator is used to initialise a *Callback* with *on_start_validation()* calling the decorated function

Example:

```
>>> import torchbearer
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import on_start_validation

# Example callback running when each validation pass starts.
>>> @on_start_validation
... def print_callback(state):
...     print('Starting validation.')

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).for_val_steps(1).
↪run()
Starting validation.
```

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *on_start_validation()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**on_sample_validation**(*func*)

The *on_sample_validation()* decorator is used to initialise a *Callback* with *on_sample_validation()* calling the decorated function

Example:

```
>>> import torchbearer
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import on_sample_validation

# Example callback running after each validation sample is drawn.
>>> @on_sample_validation
... def print_callback(state):
...     print('Sampled validation data {}.'.format(state[torchbearer.X]))

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).for_val_steps(1).
↪run()
Sampled validation data None.
```

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *on_sample_validation()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**on_forward_validation**(*func*)

The *on_forward_validation()* decorator is used to initialise a *Callback* with *on_forward_validation()* calling the decorated function

Example:

```
>>> import torchbearer
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import on_forward_validation

# Example callback running after each torch model forward pass in validation.
>>> @on_forward_validation
... def print_callback(state):
...     print('Evaluated validation batch.')

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).for_val_steps(1).
↪run()
Evaluated validation batch.
```

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *on_forward_validation()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**on_criterion_validation**(*func*)

The *on_criterion_validation()* decorator is used to initialise a *Callback* with *on_criterion_validation()* calling the decorated function

Example:

```
>>> import torchbearer
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import on_criterion_validation

# Example callback running after each criterion evaluation in validation.
>>> @on_criterion_validation
... def print_callback(state):
...     print('Current val loss {}.'.format(state[torchbearer.LOSS].item()))

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).for_val_steps(1).
↪run()
Current val loss 0.0.
```

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *on_criterion_validation()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**on_step_validation**(*func*)

The *on_step_validation()* decorator is used to initialise a *Callback* with *on_step_validation()* calling the decorated function

Example:

```
>>> import torchbearer
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import on_step_validation

# Example callback running at the end of each validation step.
>>> @on_step_validation
... def print_callback(state):
...     print('Validation step {}.'.format(state[torchbearer.BATCH]))
```

(continues on next page)

(continued from previous page)

```
>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).for_val_steps(1).
↪run()
Validation step 0.
```

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *on_step_validation()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**on_end_validation**(*func*)

The *on_end_validation()* decorator is used to initialise a *Callback* with *on_end_validation()* calling the decorated function

Example:

```
>>> import torchbearer
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import on_end_validation

# Example callback running at the end of each validation pass.
>>> @on_end_validation
... def print_callback(state):
...     print('Finished validating.')

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).for_val_steps(1).
↪run()
Finished validating.
```

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *on_end_validation()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**on_end_epoch**(*func*)

The *on_end_epoch()* decorator is used to initialise a *Callback* with *on_end_epoch()* calling the decorated function

Example:

```
>>> import torchbearer
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import on_end_epoch

# Example callback running each epoch
>>> @on_end_epoch
... def print_callback(state):
...     print('Finished epoch {}.'.format(state[torchbearer.EPOCH]))

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).run()
Finished epoch 0.
```

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *on_end_epoch()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**on_checkpoint**(*func*)

The *on_checkpoint()* decorator is used to initialise a *Callback* with *on_checkpoint()* calling the decorated function

Example:

```
>>> import torchbearer
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import on_checkpoint

# Example callback running at checkpoint time.
>>> @on_checkpoint
... def print_callback(state):
...     print('Checkpointing.')

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).run()
Checkpointing.
```

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *on_checkpoint()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**on_end**(*func*)

The *on_end()* decorator is used to initialise a *Callback* with *on_end()* calling the decorated function

Example:

```
>>> import torchbearer
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import on_end

# Example callback running after all training is finished.
>>> @on_end
... def print_callback(state):
...     print('Finished training model.')

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).run()
Finished training model.
```

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *on_end()* calling func

Return type *Callback*

8.13.2 Utility

Alongside the base callback decorators that simply bind a function to a callback point, Torchbearer has a number of utility decorators that help simplify callback construction.

torchbearer.callbacks.decorators.**add_to_loss**(*func*)

The add_to_loss() decorator is used to initialise a *Callback* with the value returned from func being added to the loss

Example:

```
>>> import torch
>>> import torchbearer
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import add_to_loss

# Example callback to add a quantity to the loss each step.
>>> @add_to_loss
... def loss_callback(state):
...     return torch.Tensor([1.125])

>>> trial = Trial(None, callbacks=[loss_callback], metrics=['loss']).for_steps(1).
↪run()
>>> print(trial[0][1]['loss'])
1.125
```

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback which adds the returned value from func to the loss

Return type *Callback*

torchbearer.callbacks.decorators.**once**(*fcn*)

Decorator to fire a callback once in the lifetime of the callback. If the callback is a class method, each instance of the class will fire only once. For functions, only the first instance will fire (even if more than one function is present in the callback list).

Example:

```
>>> import torchbearer
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import once, on_step_training

# Example callback to be called exactly once on the very first training step
>>> @once
... @on_step_training
... def print_callback(state):
...     print('This happens once ever')

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).run()
This happens once ever
```

Parameters **fcn** (*function*) – the *torchbearer callback* function to decorate.

Returns the decorator

torchbearer.callbacks.decorators.**once_per_epoch**(*fcn*)

Decorator to fire a callback once (on the first call) in any given epoch. If the callback is a class method, each instance of the class will fire once per epoch. For functions, only the first instance will fire (even if more than one function is present in the callback list).

Example:

```
>>> import torchbearer
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import once_per_epoch, on_step_training

# Example callback to be called exactly once per epoch, on the first training step
```

(continues on next page)

(continued from previous page)

```
>>> @once_per_epoch
... @on_step_training
... def print_callback(state):
...     print('This happens once per epoch')

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(1).run()
This happens once per epoch
```

Note: The decorated callback may exhibit unusual behaviour if it is reused

Parameters **fcn** (*function*) – the *torchbearer callback* function to decorate.

Returns the decorator

torchbearer.callbacks.decorators.**only_if**(*condition_expr*)

Decorator to fire a callback only if the given conditional expression function returns True. The conditional expression can be a function of state or self and state. If the decorated function is not a class method (i.e. it does not take state) the decorated function will be passed instead. This enables the storing of temporary variables.

Example:

```
>>> import torchbearer
>>> from torchbearer import Trial
>>> from torchbearer.callbacks import only_if, on_step_training

# Example callback to be called only when the given condition is true on each␣
↪training step
>>> @only_if(lambda state: state[torchbearer.BATCH] == 100)
... @on_step_training
... def print_callback(state):
...     print('This is the 100th batch')

>>> trial = Trial(None, callbacks=[print_callback]).for_steps(101).run()
This is the 100th batch
```

Parameters **condition_expr** (*function(self, state) or function(self)*) – a function/lambda which takes state and optionally self that must evaluate to true for the decorated *torchbearer callback* to be called. The *state* object passed to the callback will be passed as an argument to the condition function.

Returns the decorator

CHAPTER 9

torchbearer.metrics

The base metric classes exist to enable complex data flow requirements between metrics. All metrics are either instances of *Metric* or MetricFactory. These can then be collected in a MetricList or a MetricTree. The MetricList simply aggregates calls from a list of metrics, whereas the MetricTree will pass data from its root metric to each child and collect the outputs. This enables complex running metrics and statistics, without needing to compute the underlying values more than once. Typically, constructions of this kind should be handled using the *decorator API*.

9.1 Base Classes

class torchbearer.bases.**Metric**(*name*)

Base metric class. Process will be called on each batch, process-final at the end of each epoch. The metric contract allows for metrics to take any args but not kwargs. The initial metric call will be given state, however, subsequent metrics can pass any values desired.

Note: All metrics must extend this class.

Parameters **name** (*str*) – The name of the metric

eval(*data_key=None*)

Put the metric in eval mode during model validation.

process(**args*)

Process the state and update the metric for one iteration.

Parameters **args** – Arguments given to the metric. If this is a root level metric, will be given state

Returns None, or the value of the metric for this batch

process_final(**args*)

Process the terminal state and output the final value of the metric.

Parameters **args** – Arguments given to the metric. If this is a root level metric, will be given state

Returns None or the value of the metric for this epoch

reset(*state*)
Reset the metric, called before the start of an epoch.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

train()
Put the metric in train mode during model training.

class torchbearer.metrics.metrics.**AdvancedMetric**(*name*)
The *AdvancedMetric* class is a metric which provides different process methods for training and validation. This enables running metrics which do not output intermediate steps during validation.

Parameters **name** (*str*) – The name of the metric.

eval(*data_key=None*)
Put the metric in eval mode.

Parameters **data_key** (StateKey) – The torchbearer data_key, if used

process(**args*)
Depending on the current mode, return the result of either 'process_train' or 'process_validate'.

Returns The metric value.

process_final(**args*)
Depending on the current mode, return the result of either 'process_final_train' or 'process_final_validate'.

Returns The final metric value.

process_final_train(**args*)
Process the given state and return the final metric value for a training iteration.

Returns The final metric value for a training iteration.

process_final_validate(**args*)
Process the given state and return the final metric value for a validation iteration.

Returns The final metric value for a validation iteration.

process_train(**args*)
Process the given state and return the metric value for a training iteration.

Returns The metric value for a training iteration.

process_validate(**args*)
Process the given state and return the metric value for a validation iteration.

Returns The metric value for a validation iteration.

train()
Put the metric in train mode.

class torchbearer.metrics.metrics.**MetricList**(*metric_list*)
The *MetricList* class is a wrapper for a list of metrics which acts as a single metric and produces a dictionary of outputs.

Parameters **metric_list** (*list*) – The list of metrics to be wrapped. If the list contains a *MetricList*, this will be unwrapped. Any strings in the list will be retrieved from metrics.DEFAULT_METRICS.

eval(*data_key=None*)
Put each metric in eval mode

process(**args*)
Process each metric an wrap in a dictionary which maps metric names to values.

Returns A dictionary which maps metric names to values.

Return type dict[str,any]

process_final(**args*)
Process each metric an wrap in a dictionary which maps metric names to values.

Returns A dictionary which maps metric names to values.

Return type dict[str,any]

reset(*state*)
Reset each metric with the given state.

Parameters **state** – The current state dict of the *Trial*.

train()
Put each metric in train mode.

class torchbearer.metrics.metrics.**MetricTree**(*metric*)
A tree structure which has a node *Metric* and some children. Upon execution, the node is called with the input and its output is passed to each of the children. A dict is updated with the results.

Note: If the node output is already a dict (i.e. the node is a standalone metric), this is unwrapped before passing the **first** value to the children.

Parameters **metric** (Metric) – The metric to act as the root node of the tree / subtree

add_child(*child*)
Add a child to this node of the tree

Parameters **child** (Metric) – The child to add

eval(*data_key=None*)
Put the metric in eval mode during model validation.

process(**args*)
Process this node and then pass the output to each child.

Returns A dict containing all results from the children

process_final(**args*)
Process this node and then pass the output to each child.

Returns A dict containing all results from the children

reset(*state*)
Reset the metric, called before the start of an epoch.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

train()
Put the metric in train mode during model training.

torchbearer.metrics.metrics.**add_default**(*key*, *metric*, **args*, ***kwargs*)

torchbearer.metrics.metrics.**get_default**(*key*)

9.2 Decorators - The Decorator API

The decorator API is the core way to interact with metrics in torchbearer. All of the classes and functionality handled here can be reproduced by manually interacting with the classes if necessary. Broadly speaking, the decorator API is used to construct a `MetricFactory` which will build a *MetricTree* that handles data flow between instances of *Mean*, *RunningMean*, *Std* etc.

`torchbearer.metrics.decorators.`**`default_for_key`**(*key*, **args*, ***kwargs*)

The *default_for_key()* decorator will register the given metric in the global metric dict (*metrics.DEFAULT_METRICS*) so that it can be referenced by name in instances of *MetricList* such as in the list given to the `torchbearer.Model`.

Example:

```
@default_for_key('acc')
class CategoricalAccuracy(metrics.BatchLambda):
    ...
```

Parameters

- **key** (*str*) – The key to use when referencing the metric
- **args** – Any args to pass to the underlying metric when constructed
- **kwargs** – Any keyword args to pass to the underlying metric when constructed

`torchbearer.metrics.decorators.`**`lambda_metric`**(*name*, *on_epoch=False*)

The *lambda_metric()* decorator is used to convert a lambda function *y_pred, y_true* into a *Metric* instance. This can be used as in the following example:

```
@metrics.lambda_metric('my_metric')
def my_metric(y_pred, y_true):
    ... # Calculate some metric

model = Model(metrics=[my_metric])
```

Parameters

- **name** (*str*) – The name of the metric (e.g. 'loss')
- **on_epoch** (*bool*) – If True the metric will be an instance of *EpochLambda* instead of *BatchLambda*

Returns A decorator which replaces a function with a *Metric*

`torchbearer.metrics.decorators.`**`mean`**(*clazz=None*, *dim=None*)

The *mean()* decorator is used to add a *Mean* to the *MetricTree* which will will output a mean value at the end of each epoch. At build time, if the inner class is not a *MetricTree*, one will be created. The *Mean* will also be wrapped in a *ToDict* for simplicity.

Example:

```
>>> import torch
>>> from torchbearer import metrics

>>> @metrics.mean
... @metrics.lambda_metric('my_metric')
... def metric(y_pred, y_true):
```

(continues on next page)

(continued from previous page)

```
...     return y_pred + y_true
...
>>> metric.reset({})
>>> metric.process({'y_pred':torch.Tensor([2]), 'y_true':torch.Tensor([2])}) # 4
{}
>>> metric.process({'y_pred':torch.Tensor([3]), 'y_true':torch.Tensor([3])}) # 6
{}
>>> metric.process({'y_pred':torch.Tensor([4]), 'y_true':torch.Tensor([4])}) # 8
{}
>>> metric.process_final()
{'my_metric': 6.0}
```

Parameters

- **clazz** – The class to *decorate*
- **dim** (*int, tuple*) – See *Mean*

Returns A *MetricTree* with a *Mean* appended or a wrapper class that extends *MetricTree*

torchbearer.metrics.decorators.**running_mean**(*clazz=None*, *batch_size=50*, *step_size=10*, *dim=None*)

The *running_mean()* decorator is used to add a *RunningMean* to the *MetricTree*. If the inner class is not a *MetricTree* then one will be created. The *RunningMean* will be wrapped in a *ToDict* (with 'running_' prepended to the name) for simplicity.

Note: The decorator function does not need to be called if not desired, both: *@running_mean* and *@running_mean()* are acceptable.

Example:

```
>>> import torch
>>> from torchbearer import metrics

>>> @metrics.running_mean(step_size=2) # Update every 2 steps
... @metrics.lambda_metric('my_metric')
... def metric(y_pred, y_true):
...     return y_pred + y_true
...
>>> metric.reset({})
>>> metric.process({'y_pred':torch.Tensor([2]), 'y_true':torch.Tensor([2])}) # 4
{'running_my_metric': 4.0}
>>> metric.process({'y_pred':torch.Tensor([3]), 'y_true':torch.Tensor([3])}) # 6
{'running_my_metric': 4.0}
>>> metric.process({'y_pred':torch.Tensor([4]), 'y_true':torch.Tensor([4])}) # 8,␣
↪triggers update
{'running_my_metric': 6.0}
```

Parameters

- **clazz** – The class to *decorate*
- **batch_size** (*int*) – See *RunningMean*
- **step_size** (*int*) – See *RunningMean*
- **dim** (*int, tuple*) – See *RunningMean*

Returns decorator or *MetricTree* instance or wrapper

torchbearer.metrics.decorators.**std**(*clazz=None*, *unbiased=True*, *dim=None*)

The *std()* decorator is used to add a *Std* to the *MetricTree* which will will output a sample standard deviation value at the end of each epoch. At build time, if the inner class is not a *MetricTree*, one will be created. The *Std* will also be wrapped in a *ToDict* (with '_std' appended) for simplicity.

Example:

```
>>> import torch
>>> from torchbearer import metrics

>>> @metrics.std
... @metrics.lambda_metric('my_metric')
... def metric(y_pred, y_true):
...     return y_pred + y_true
...
>>> metric.reset({})
>>> metric.process({'y_pred':torch.Tensor([2]), 'y_true':torch.Tensor([2])}) # 4
{}
>>> metric.process({'y_pred':torch.Tensor([3]), 'y_true':torch.Tensor([3])}) # 6
{}
>>> metric.process({'y_pred':torch.Tensor([4]), 'y_true':torch.Tensor([4])}) # 8
{}
>>> '%.4f' % metric.process_final()['my_metric_std']
'2.0000'
```

Parameters

- **clazz** – The class to *decorate*
- **unbiased** (*bool*) – See *Std*
- **dim** (*int, tuple*) – See *Std*

Returns A *MetricTree* with a *Std* appended or a wrapper class that extends *MetricTree*

torchbearer.metrics.decorators.**to_dict**(*clazz*)

The *to_dict()* decorator is used to wrap either a *Metric* class or a *Metric* instance with a *ToDict* instance. The result is that future output will be wrapped in a *dict[name, value]*.

Example:

```
>>> from torchbearer import metrics

>>> @metrics.lambda_metric('my_metric')
... def my_metric(y_pred, y_true):
...     return y_pred + y_true
...
>>> my_metric.process({'y_pred':4, 'y_true':5})
9

>>> @metrics.to_dict
... @metrics.lambda_metric('my_metric')
... def my_metric(y_pred, y_true):
...     return y_pred + y_true
...
>>> my_metric.process({'y_pred':4, 'y_true':5})
{'my_metric': 9}
```

Parameters **clazz** – The class to *decorate*

Returns A *ToDict* instance or a *ToDict* wrapper of the given class

torchbearer.metrics.decorators.**var**(*clazz=None*, *unbiased=True*, *dim=None*)

The *var()* decorator is used to add a *Var* to the *MetricTree* which will will output a sample variance value at the end of each epoch. At build time, if the inner class is not a *MetricTree*, one will be created. The *Var* will also be wrapped in a *ToDict* (with '_var' appended) for simplicity.

Example:

```
>>> import torch
>>> from torchbearer import metrics

>>> @metrics.var
... @metrics.lambda_metric('my_metric')
... def metric(y_pred, y_true):
...     return y_pred + y_true
...
>>> metric.reset({})
>>> metric.process({'y_pred':torch.Tensor([2]), 'y_true':torch.Tensor([2])}) # 4
{}
>>> metric.process({'y_pred':torch.Tensor([3]), 'y_true':torch.Tensor([3])}) # 6
{}
>>> metric.process({'y_pred':torch.Tensor([4]), 'y_true':torch.Tensor([4])}) # 8
{}
>>> '%.4f' % metric.process_final()['my_metric_var']
'4.0000'
```

Parameters

- **clazz** – The class to *decorate*
- **unbiased** (*bool*) – See *Var*
- **dim** (*int, tuple*) – See *Var*

Returns A *MetricTree* with a *Var* appended or a wrapper class that extends *MetricTree*

9.3 Metric Wrappers

Metric wrappers are classes which wrap instances of *Metric* or, in the case of *EpochLambda* and *BatchLambda*, functions. Typically, these should **not** be used directly (although this is entirely possible), but via the *decorator API*.

class torchbearer.metrics.wrappers.**BatchLambda**(*name*, *metric_function*)

A metric which returns the output of the given function on each batch.

Parameters

- **name** (*str*) – The name of the metric.
- **metric_function** (*func*) – A metric function('y_pred', 'y_true') to wrap.

process(**args*)

Return the output of the wrapped function.

Parameters **args** – The *torchbearer.Trial* state.

Returns The value of the metric function('y_pred', 'y_true').

class torchbearer.metrics.wrappers.**EpochLambda**(*name*, *metric_function*, *running=True*, *step_size=50*)

A metric wrapper which computes the given function for concatenated values of 'y_true' and 'y_pred' each epoch. Can be used as a running metric which computes the function for batches of outputs with a given step size during training.

Parameters

- **name** (*str*) – The name of the metric.
- **metric_function** (*func*) – The function('y_pred', 'y_true') to use as the metric.
- **running** (*bool*) – True if this should act as a running metric.
- **step_size** (*int*) – Step size to use between calls if running=True.

process_final_train(**args*)

Evaluate the function with the aggregated outputs.

Returns The result of the function.

process_final_validate(**args*)

Evaluate the function with the aggregated outputs.

Returns The result of the function.

process_train(**args*)

Concatenate the 'y_true' and 'y_pred' from the state along the 0 dimension, this must be the batch dimension. If this is a running metric, evaluates the function every number of steps.

Parameters **args** – The *torchbearer.Trial* state.

Returns The current running result.

process_validate(**args*)

During validation, just concatenate 'y_true' and y_pred'.

Parameters **args** – The *torchbearer.Trial* state.

reset(*state*)

Reset the 'y_true' and 'y_pred' caches.

Parameters **state** (*dict*) – The *torchbearer.Trial* state.

class torchbearer.metrics.wrappers.**ToDict**(*metric*)

The *ToDict* class is an *AdvancedMetric* which will put output from the inner *Metric* in a dict (mapping metric name to value) before returning. When in *eval* mode, 'val_' will be prepended to the metric name.

Example:

```
>>> from torchbearer import metrics

>>> @metrics.lambda_metric('my_metric')
... def my_metric(y_pred, y_true):
...     return y_pred + y_true
...
>>> metric = metrics.ToDict(my_metric().build())
>>> metric.process({'y_pred': 4, 'y_true': 5})
{'my_metric': 9}
>>> metric.eval()
>>> metric.process({'y_pred': 4, 'y_true': 5})
{'val_my_metric': 9}
```

Parameters **metric** (Metric) – The *Metric* instance to *wrap*.

eval(*data_key=None*)
Put the metric in eval mode.

Parameters **data_key** (StateKey) – The torchbearer data_key, if used

process_final_train(**args*)
Process the given state and return the final metric value for a training iteration.

Returns The final metric value for a training iteration.

process_final_validate(**args*)
Process the given state and return the final metric value for a validation iteration.

Returns The final metric value for a validation iteration.

process_train(**args*)
Process the given state and return the metric value for a training iteration.

Returns The metric value for a training iteration.

process_validate(**args*)
Process the given state and return the metric value for a validation iteration.

Returns The metric value for a validation iteration.

reset(*state*)
Reset the metric, called before the start of an epoch.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

train()
Put the metric in train mode.

9.4 Metric Aggregators

Aggregators are a special kind of *Metric* which takes as input, the output from a previous metric or metrics. As a result, via a *MetricTree*, a series of aggregators can collect statistics such as Mean or Standard Deviation without needing to compute the underlying metric multiple times. This can, however, make the aggregators complex to use. It is therefore typically better to use the *decorator API*.

class torchbearer.metrics.aggregators.**Mean**(*name*, *dim=None*)
Metric aggregator which calculates the mean of process outputs between calls to reset.

Parameters

- **name** (*str*) – The name of this metric.
- **dim** (*int, tuple*) – The dimension(s) on which to perform the mean. If left as None, this will mean over the whole Tensor

process(**args*)
Add the input to the rolling sum. Input must be a torch tensor.

Parameters **args** – The output of some previous call to *Metric.process()*.

process_final(**args*)
Compute and return the mean of all metric values since the last call to reset.

Returns The mean of the metric values since the last call to reset.

reset(*state*)
Reset the running count and total.

Parameters **state** (*dict*) – The model state.

class torchbearer.metrics.aggregators.**RunningMean**(*name*, *batch_size=50*, *step_size=10*, *dim=None*)

A *RunningMetric* which outputs the running mean of its input tensors over the course of an epoch.

Parameters

- **name** (*str*) – The name of this running mean.
- **batch_size** (*int*) – The size of the deque to store of previous results.
- **step_size** (*int*) – The number of iterations between aggregations.
- **dim** (*int, tuple*) – The dimension(s) on which to perform the mean. If left as None, this will mean over the whole Tensor

class torchbearer.metrics.aggregators.**RunningMetric**(*name*, *batch_size=50*, *step_size=10*)

A metric which aggregates batches of results and presents a method to periodically process these into a value.

Note: Running metrics only provide output during training.

Parameters

- **name** (*str*) – The name of the metric.
- **batch_size** (*int*) – The size of the deque to store of previous results.
- **step_size** (*int*) – The number of iterations between aggregations.

process_train(**args*)

Add the current metric value to the cache and call '_step' is needed.

Parameters **args** – The output of some *Metric*

Returns The current metric value.

reset(*state*)

Reset the step counter. Does not clear the cache.

Parameters **state** (*dict*) – The current model state.

class torchbearer.metrics.aggregators.**Std**(*name*, *unbiased=True*, *dim=None*)

Metric aggregator which calculates the **sample** standard deviation of process outputs between calls to reset. Optionally calculate the population std if `unbiased = False`.

Parameters

- **name** (*str*) – The name of this metric.
- **unbiased** (*bool*) – If True (default), calculates the sample standard deviation, else, the population standard deviation
- **dim** (*int, tuple*) – The dimension(s) on which to compute the std. If left as None, this will operate over the whole Tensor

process_final(**args*)

Compute and return the final standard deviation.

Returns The standard deviation of each observation since the last reset call.

class torchbearer.metrics.aggregators.**Var**(*name*, *unbiased=True*, *dim=None*)

Metric aggregator which calculates the **sample** variance of process outputs between calls to reset. Optionally calculate the population variance if `unbiased = False`.

Parameters

- **name** (*str*) – The name of this metric.
- **unbiased** (*bool*) – If True (default), calculates the sample variance, else, the population variance
- **dim** (*int, tuple*) – The dimension(s) on which to compute the std. If left as None, this will operate over the whole Tensor

process(**args*)

Compute values required for the variance from the input. The input should be a torch Tensor. The sum and sum of squares will be computed over the provided dimension.

Parameters **args** (*torch.Tensor*) – The output of some previous call to *Metric.process()*.

process_final(**args*)

Compute and return the final variance.

Returns The variance of each observation since the last reset call.

reset(*state*)

Reset the statistics to compute the next variance.

Parameters **state** (*dict*) – The model state.

9.5 Base Metrics

Base metrics are the base classes which represent the metrics supplied with torchbearer. They all use the *default_for_key()* decorator so that they can be accessed in the call to `torchbearer.Model` via the following strings:

- '*acc*' or '*accuracy*': The *DefaultAccuracy* metric
- '*binary_acc*' or '*binary_accuracy*': The *BinaryAccuracy* metric
- '*cat_acc*' or '*cat_accuracy*': The *CategoricalAccuracy* metric
- '*top_5_acc*' or '*top_5_accuracy*': The *TopKCategoricalAccuracy* metric
- '*top_10_acc*' or '*top_10_accuracy*': The *TopKCategoricalAccuracy* metric with k=10
- '*mse*': The *MeanSquaredError* metric
- '*loss*': The *Loss* metric
- '*epoch*': The *Epoch* metric
- '*lr*': The `LR` metric
- '*roc_auc*' or '*roc_auc_score*': The *RocAucScore* metric

class torchbearer.metrics.default.**DefaultAccuracy**

The default accuracy metric loads in a different accuracy metric depending on the loss function or criterion in use at the start of training. Default for keys: *acc*, *accuracy*. The following bindings are in place for both nn and functional variants:

- cross entropy loss -> *CategoricalAccuracy* [DEFAULT]

- nll loss -> *CategoricalAccuracy*
- mse loss -> *MeanSquaredError*
- bce loss -> *BinaryAccuracy*
- bce loss with logits -> *BinaryAccuracy*

eval(*data_key=None*)
Put the metric in eval mode during model validation.

process(**args*)
Process the state and update the metric for one iteration.

Parameters **args** – Arguments given to the metric. If this is a root level metric, will be given state

Returns None, or the value of the metric for this batch

process_final(**args*)
Process the terminal state and output the final value of the metric.

Parameters **args** – Arguments given to the metric. If this is a root level metric, will be given state

Returns None or the value of the metric for this epoch

reset(*state*)
Reset the metric, called before the start of an epoch.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

train()
Put the metric in train mode during model training.

class torchbearer.metrics.primitives.**BinaryAccuracy**
Binary accuracy metric. Uses torch.eq to compare predictions to targets. Decorated with a mean and running_mean. Default for key: 'binary_acc'.

Parameters

- **pred_key** (StateKey) – The key in state which holds the predicted values
- **target_key** (StateKey) – The key in state which holds the target values
- **threshold** (*float*) – value between 0 and 1 to use as a threshold when binarizing predictions and targets

class torchbearer.metrics.primitives.**CategoricalAccuracy**(*ignore_index=-100*)
Categorical accuracy metric. Uses torch.max to determine predictions and compares to targets. Decorated with a mean, running_mean and std. Default for key: 'cat_acc'

Parameters

- **pred_key** (StateKey) – The key in state which holds the predicted values
- **target_key** (StateKey) – The key in state which holds the target values
- **ignore_index** (*int*) – Specifies a target value that is ignored and does not contribute to the metric output. See https://pytorch.org/docs/stable/nn.html#crossentropyloss

class torchbearer.metrics.primitives.**TopKCategoricalAccuracy**(*k=5*, *ignore_index=-100*)
Top K Categorical accuracy metric. Uses torch.topk to determine the top k predictions and compares to targets. Decorated with a mean, running_mean and std. Default for keys: 'top_5_acc', 'top_10_acc'.

Parameters

- **pred_key** (StateKey) – The key in state which holds the predicted values
- **target_key** (StateKey) – The key in state which holds the target values
- **ignore_index** (*int*) – Specifies a target value that is ignored and does not contribute to the metric output. See https://pytorch.org/docs/stable/nn.html#crossentropyloss

class torchbearer.metrics.primitives.**MeanSquaredError**

Mean squared error metric. Computes the pixelwise squared error which is then averaged with decorators. Decorated with a mean and running_mean. Default for key: 'mse'.

Parameters

- **pred_key** (StateKey) – The key in state which holds the predicted values
- **target_key** (StateKey) – The key in state which holds the target values

class torchbearer.metrics.primitives.**Loss**

Simply returns the 'loss' value from the model state. Decorated with a mean, running_mean and std. Default for key: 'loss'.

State Requirements:

- torchbearer.state.LOSS: This key should map to the loss for the current batch

class torchbearer.metrics.primitives.**Epoch**

Returns the 'epoch' from the model state. Default for key: 'epoch'.

State Requirements:

- torchbearer.state.EPOCH: This key should map to the number of the current epoch

class torchbearer.metrics.roc_auc_score.**RocAucScore**(*one_hot_labels=True*, *one_hot_offset=0*, *one_hot_classes=10*)

Area Under ROC curve metric. Default for keys: 'roc_auc', 'roc_auc_score'.

Note: Requires sklearn.metrics.

Parameters

- **one_hot_labels** (*bool*) – If True, convert the labels to a one hot encoding. Required if they are not already.
- **one_hot_offset** (*int*) – Subtracted from class labels, use if not already zero based.
- **one_hot_classes** (*int*) – Number of classes for the one hot encoding.

9.6 Timer

class torchbearer.metrics.timer.**TimerMetric**(*time_keys=()*)

get_timings()

on_backward(*state*)

Perform some action with the given state as context after backward has been called on the loss.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_criterion(*state*)
Perform some action with the given state as context after the criterion has been evaluated.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_criterion_validation(*state*)
Perform some action with the given state as context after the criterion evaluation has been completed with the validation data.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_end(*state*)
Perform some action with the given state as context at the end of the model fitting.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_end_epoch(*state*)
Perform some action with the given state as context at the end of each epoch.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_end_training(*state*)
Perform some action with the given state as context after the training loop has completed.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_end_validation(*state*)
Perform some action with the given state as context at the end of the validation loop.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_forward(*state*)
Perform some action with the given state as context after the forward pass (model output) has been completed.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_forward_validation(*state*)
Perform some action with the given state as context after the forward pass (model output) has been completed with the validation data.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_sample(*state*)
Perform some action with the given state as context after data has been sampled from the generator.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_sample_validation(*state*)
Perform some action with the given state as context after data has been sampled from the validation generator.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_start(*state*)
Perform some action with the given state as context at the start of a model fit.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_start_epoch(*state*)
Perform some action with the given state as context at the start of each epoch.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_start_training(*state*)
Perform some action with the given state as context at the start of the training loop.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_start_validation(*state*)
Perform some action with the given state as context at the start of the validation loop.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_step_training(*state*)
Perform some action with the given state as context after step has been called on the optimiser.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

on_step_validation(*state*)
Perform some action with the given state as context at the end of each validation step.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

process(**args*)
Process the state and update the metric for one iteration.

Parameters **args** – Arguments given to the metric. If this is a root level metric, will be given state

Returns None, or the value of the metric for this batch

reset(*state*)
Reset the metric, called before the start of an epoch.

Parameters **state** (*dict*) – The current state dict of the *Trial*.

update_time(*text*, *metric*, *state*)

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

E

F

G

H

I

K

L

M

O

P

R

U

V

W

X

Z