torchbearer Documentation

Release 0.1.7

Ethan Harris and Matthew Painter

Aug 14, 2018

Notes

CHAPTER 1

Using 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.

1.1 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()*.

1.2 Metric Decorators

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

```
@metrics.default_for_key('acc')
@metrics.default_for_key('accuracy')
@metrics.running_mean
@metrics.std
@metrics.mean
class CategoricalAccuracyFactory(metrics.MetricFactory):
    def build(self):
        return CategoricalAccuracy()
```

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

`CategoricalAccuracyFactory` is a `MetricFactory` which simply returns a `CategoricalAccuracy` instance on build. We don't need to do this, the decorators can simply take a `Metric` implementation, however, for torchbearer we wanted to keep the `CategoricalAccuracy` class clean so that it could still be used in cases where running means are not desirable.

`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'.

1.2.1 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.

1.2.2 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.

1.3 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 2

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.

2.1 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

2.2 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:

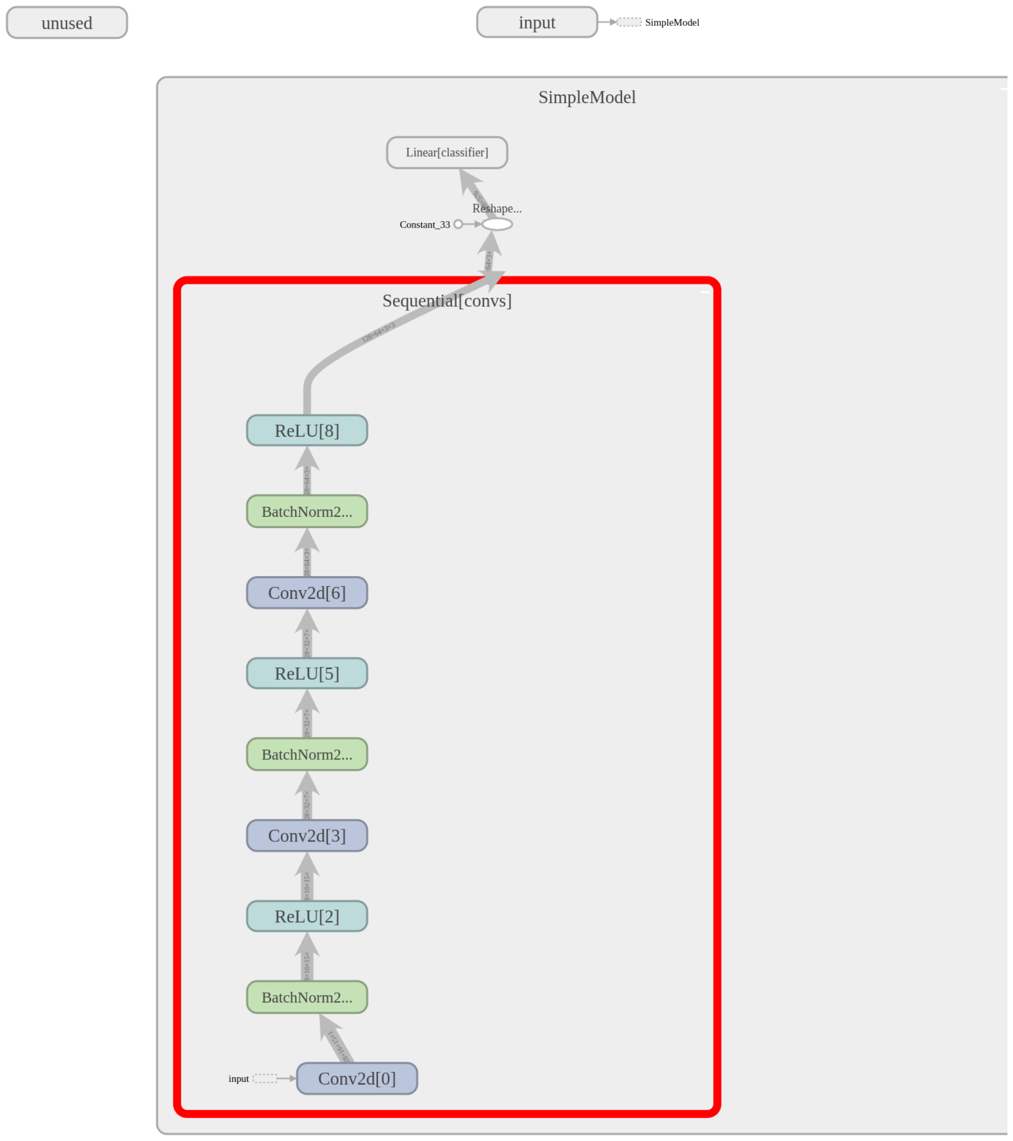
```
from torchbearer import Model
from torchbearer.callbacks import TensorBoard
```

(continues on next page)

(continued from previous page)

```
torchbearer_model = Model(model, optimizer, loss, metrics=['acc', 'loss']).to('cuda')

torchbearer_model.fit_generator(traingen, epochs=1, validation_generator=valgen,
                                callbacks=[TensorBoard(write_graph=True, write_batch_
↪metrics=False, write_epoch_metrics=False)])
```

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:



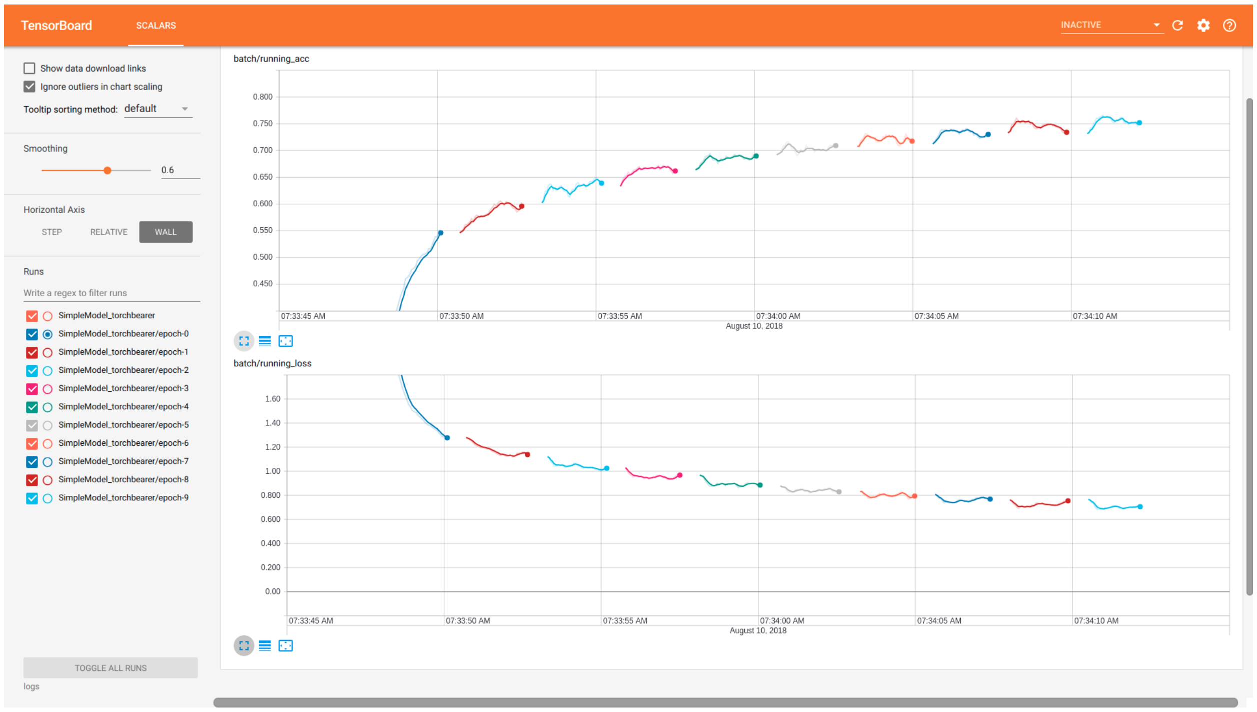
The dynamic graph construction does introduce some weirdness, however, this is about as good as model graphs typically get.

2.3 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 5 epochs with the following:

```
torchbearer_model.fit_generator(traingen, epochs=10, validation_generator=valgen,
                                callbacks=[TensorBoard(write_graph=False, write_batch_
↪metrics=True, batch_step_size=10, write_epoch_metrics=False)])
```

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

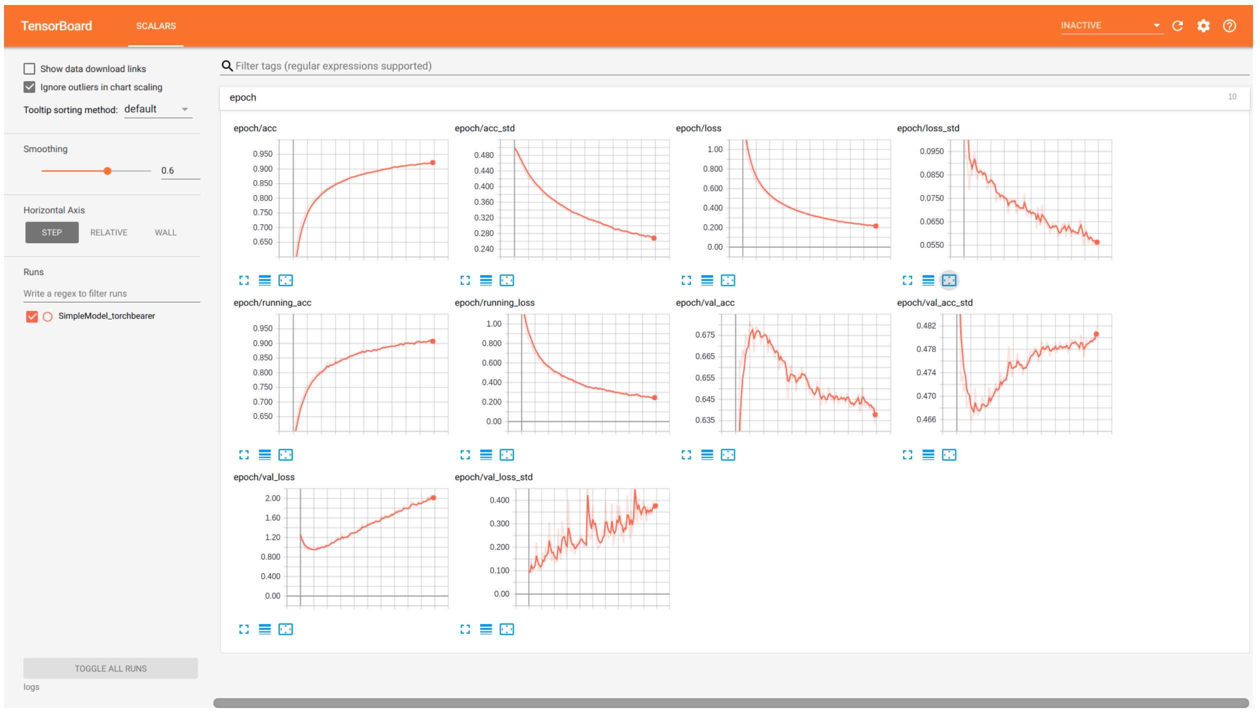


2.4 Logging Epoch Metrics

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_model.fit_generator(traingen, epochs=100, validation_generator=valgen,
                                callbacks=[TensorBoard(write_graph=False, write_batch_
↪metrics=False, write_epoch_metrics=True)])
```

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).

2.5 Source Code

The source code for these examples is given below:

    Download Python source code: tensorboard.py

CHAPTER 3

Using Visdom Logging in the Tensorboard Callbacks

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.

3.1 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()
```

3.2 Logging Epoch and Batch Metrics

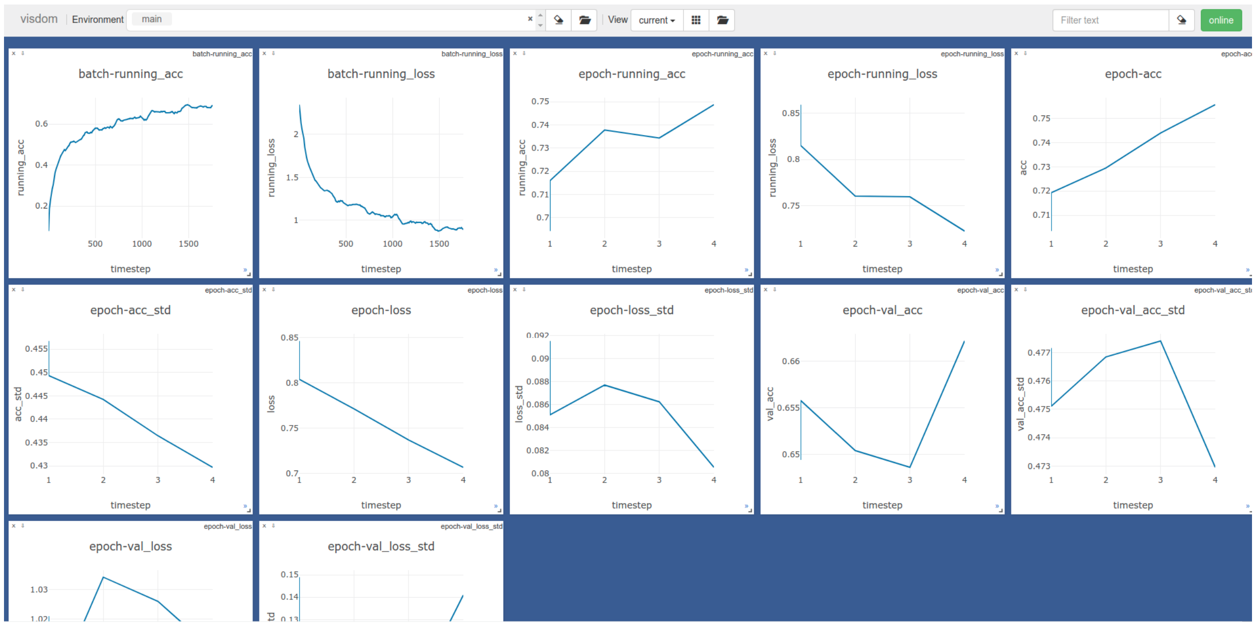
Visdom does not support logging model graphs so we shall start with logging epoch and batch metrics. The only change we need to make to the tensorboard example is setting `visdom=True` in the *TensorBoard callback* constructor.

```
torchbearer_model.fit_generator(traingen, epochs=5, validation_generator=valgen,
                                callbacks=[TensorBoard(visdom=True, write_graph=False,
↪ write_batch_metrics=True, batch_step_size=10, write_epoch_metrics=False)])

torchbearer_model.fit_generator(traingen, epochs=5, validation_generator=valgen,
                                callbacks=[TensorBoard(visdom=True, write_graph=False,
↪ write_batch_metrics=False, write_epoch_metrics=True)])
```

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

3.3 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'
    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'
```

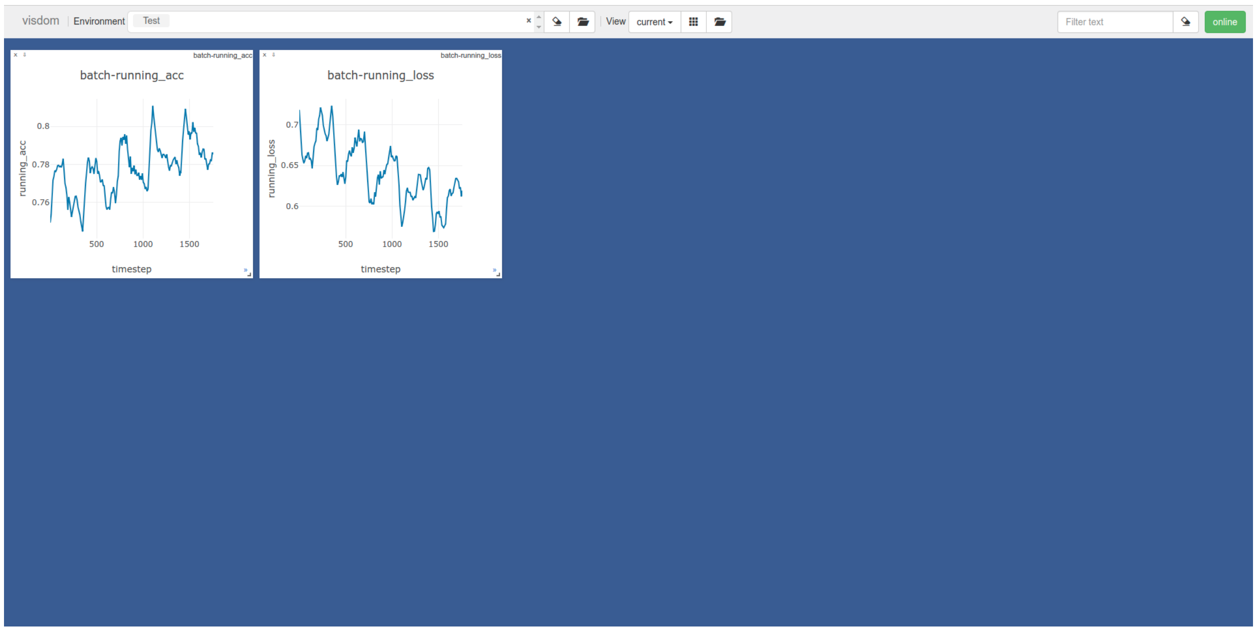
Running another fit call, we can see we are now logging to the "Test" environment.

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.

3.4 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 4

Quickstart Guide

This guide will give a quick intro to training PyTorch models with torchbearer. We'll start by loading in some data and defining a model, then we'll train it for a few epochs and see how well it does.

4.1 Defining the Model

Let's get using torchbearer. Here's some data from Cifar10 and a simple 3 layer strided CNN:

```
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)
```

(continues on next page)

(continued from previous page)

```
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()
```

Note that we use torchbearers `DatasetValidationSplitter` here to create a validation set (10% of the data). This is essential to avoid over-fitting to your test data.

4.2 Training on Cifar10

Typically we would need a training loop and a series of calls to backward, step etc. Instead, with torchbearer, we can define our optimiser and some metrics (just ‘acc’ and ‘loss’ for now) and let it do the work.

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

from torchbearer import Model

torchbearer_model = Model(model, optimizer, loss, metrics=['acc', 'loss']).to('cuda')
torchbearer_model.fit_generator(traingen, epochs=10, validation_generator=valgen)

torchbearer_model.evaluate_generator(testgen)
```

Running the above produces the following output:

```
Files already downloaded and verified
Files already downloaded and verified
0/10(t): 100%|| 352/352 [00:01<00:00, 233.36it/s, running_acc=0.536, running_loss=1.
↪32, acc=0.459, acc_std=0.498, loss=1.52, loss_std=0.239]
0/10(v): 100%|| 40/40 [00:00<00:00, 239.40it/s, val_acc=0.536, val_acc_std=0.499, val_
↪loss=1.29, val_loss_std=0.0731]
1/10(t): 100%|| 352/352 [00:01<00:00, 211.19it/s, running_acc=0.599, running_loss=1.
↪13, acc=0.578, acc_std=0.494, loss=1.18, loss_std=0.096]
```

(continues on next page)

(continued from previous page)

```
1/10(v): 100%|| 40/40 [00:00<00:00, 232.97it/s, val_acc=0.594, val_acc_std=0.491, val_
↪loss=1.14, val_loss_std=0.101]
2/10(t): 100%|| 352/352 [00:01<00:00, 216.68it/s, running_acc=0.636, running_loss=1.
↪04, acc=0.631, acc_std=0.482, loss=1.04, loss_std=0.0944]
2/10(v): 100%|| 40/40 [00:00<00:00, 210.73it/s, val_acc=0.626, val_acc_std=0.484, val_
↪loss=1.07, val_loss_std=0.0974]
3/10(t): 100%|| 352/352 [00:01<00:00, 190.88it/s, running_acc=0.671, running_loss=0.
↪929, acc=0.664, acc_std=0.472, loss=0.957, loss_std=0.0929]
3/10(v): 100%|| 40/40 [00:00<00:00, 221.79it/s, val_acc=0.639, val_acc_std=0.48, val_
↪loss=1.02, val_loss_std=0.103]
4/10(t): 100%|| 352/352 [00:01<00:00, 212.43it/s, running_acc=0.685, running_loss=0.
↪897, acc=0.689, acc_std=0.463, loss=0.891, loss_std=0.0888]
4/10(v): 100%|| 40/40 [00:00<00:00, 249.99it/s, val_acc=0.655, val_acc_std=0.475, val_
↪loss=0.983, val_loss_std=0.113]
5/10(t): 100%|| 352/352 [00:01<00:00, 209.45it/s, running_acc=0.711, running_loss=0.
↪835, acc=0.706, acc_std=0.456, loss=0.844, loss_std=0.088]
5/10(v): 100%|| 40/40 [00:00<00:00, 240.80it/s, val_acc=0.648, val_acc_std=0.477, val_
↪loss=0.965, val_loss_std=0.107]
6/10(t): 100%|| 352/352 [00:01<00:00, 216.89it/s, running_acc=0.713, running_loss=0.
↪826, acc=0.72, acc_std=0.449, loss=0.802, loss_std=0.0903]
6/10(v): 100%|| 40/40 [00:00<00:00, 238.17it/s, val_acc=0.655, val_acc_std=0.475, val_
↪loss=0.97, val_loss_std=0.0997]
7/10(t): 100%|| 352/352 [00:01<00:00, 213.82it/s, running_acc=0.737, running_loss=0.
↪773, acc=0.734, acc_std=0.442, loss=0.765, loss_std=0.0878]
7/10(v): 100%|| 40/40 [00:00<00:00, 202.45it/s, val_acc=0.677, val_acc_std=0.468, val_
↪loss=0.936, val_loss_std=0.0985]
8/10(t): 100%|| 352/352 [00:01<00:00, 211.36it/s, running_acc=0.732, running_loss=0.
↪744, acc=0.746, acc_std=0.435, loss=0.728, loss_std=0.0902]
8/10(v): 100%|| 40/40 [00:00<00:00, 204.52it/s, val_acc=0.674, val_acc_std=0.469, val_
↪loss=0.949, val_loss_std=0.124]
9/10(t): 100%|| 352/352 [00:01<00:00, 215.76it/s, running_acc=0.741, running_loss=0.
↪735, acc=0.754, acc_std=0.431, loss=0.703, loss_std=0.0897]
9/10(v): 100%|| 40/40 [00:00<00:00, 222.72it/s, val_acc=0.68, val_acc_std=0.466, val_
↪loss=0.948, val_loss_std=0.181]
0/1(e): 100%|| 79/79 [00:00<00:00, 268.70it/s, val_acc=0.678, val_acc_std=0.467, val_
↪loss=0.925, val_loss_std=0.109]
```

4.3 Source Code

The source code for the example is given below:

`Download Python source code: quickstart.py`

CHAPTER 5

Training a Variational Auto-Encoder

This guide will give a quick guide on training a variational auto-encoder (VAE) in torchbearer. We will use the VAE example from the pytorch examples here:

5.1 Defining the Model

We shall first copy the VAE example model.

```
class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()

        self.fc1 = nn.Linear(784, 400)
        self.fc21 = nn.Linear(400, 20)
        self.fc22 = nn.Linear(400, 20)
        self.fc3 = nn.Linear(20, 400)
        self.fc4 = nn.Linear(400, 784)

    def encode(self, x):
        h1 = F.relu(self.fc1(x))
        return self.fc21(h1), self.fc22(h1)

    def reparameterize(self, mu, logvar):
        if self.training:
            std = torch.exp(0.5*logvar)
            eps = torch.randn_like(std)
            return eps.mul(std).add_(mu)
        else:
            return mu

    def decode(self, z):
        h3 = F.relu(self.fc3(z))
        return F.sigmoid(self.fc4(h3))
```

(continues on next page)

(continued from previous page)

```
    def forward(self, x):
        mu, logvar = self.encode(x.view(-1, 784))
        z = self.reparameterize(mu, logvar)
        return self.decode(z), mu, logvar
```

5.2 Defining the Data

We get the MNIST dataset from torchvision and transform them to torch tensors.

```
BATCH_SIZE = 128

normalize = transforms.Compose([transforms.ToTensor()])

# Define standard classification mnist dataset

basetrainset = torchvision.datasets.MNIST('./data/mnist', train=True, download=True,␣
↪transform=normalize)

basetestset = torchvision.datasets.MNIST('./data/mnist', train=False, download=True,␣
↪transform=normalize)
```

The output label from this dataset is the classification label, since we are doing a auto-encoding problem, we wish the label to be the original image. To fix this we create a wrapper class which replaces the classification label with the image.

```
class AutoEncoderMNIST(Dataset):
    def __init__(self, mnist_dataset):
        super().__init__()
        self.mnist_dataset = mnist_dataset

    def __getitem__(self, index):
        character, label = self.mnist_dataset.__getitem__(index)
        return character, character

    def __len__(self):
        return len(self.mnist_dataset)
```

We then wrap the original datasets and create training and testing data generators in the standard pytorch way.

```
# Wrap base classification mnist dataset to return the image as the target

trainset = AutoEncoderMNIST(basetrainset)

testset = AutoEncoderMNIST(basetestset)

traingen = torch.utils.data.DataLoader(trainset, batch_size=BATCH_SIZE, shuffle=True,␣
↪num_workers=8)

testgen = torch.utils.data.DataLoader(testset, batch_size=BATCH_SIZE, shuffle=True,␣
↪num_workers=8)
```

5.3 Defining the Loss

Now we have the model and data, we will need a loss function to optimize. VAEs typically take the sum of a reconstruction loss and a KL-divergence loss to form the final loss value.

```
def bce_loss(y_pred, y_true):
    BCE = F.binary_cross_entropy(y_pred, y_true.view(-1, 784), size_average=False)
    return BCE
```

```
def kld_Loss(mu, logvar):
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return KLD
```

There are two ways this can be done in torchbearer - one is very similar to the PyTorch example method and the other utilises the torchbearer state.

5.3.1 PyTorch method

The loss function slightly modified from the PyTorch example is:

```
def loss_function(y_pred, y_true):
    recon_x, mu, logvar = y_pred
    x = y_true

    BCE = bce_loss(recon_x, x)

    KLD = kld_Loss(mu, logvar)

    return BCE + KLD
```

This requires the packing of the reconstruction, mean and log-variance into the model output and unpacking it for the loss function to use.

```
    def forward(self, x):
        mu, logvar = self.encode(x.view(-1, 784))
        z = self.reparameterize(mu, logvar)
        return self.decode(z), mu, logvar
```

5.3.2 Using Torchbearer State

Instead of having to pack and unpack the mean and variance in the forward pass, in torchbearer there is a persistent state dictionary which can be used to conveniently hold such intermediate tensors.

By default the model forward pass does not have access to the state dictionary, but setting the `pass_state` flag to true in the fit_generator call gives the model access to state on forward.

```
torchbearer_model.fit_generator(traingen, epochs=10, validation_generator=testgen,
                                callbacks=[add_kld_loss_callback, save_reconstruction_
↪callback()], pass_state=True)
```

We can then modify the model forward pass to store the mean and log-variance under suitable keys.

```
    def forward(self, x, state):
        mu, logvar = self.encode(x.view(-1, 784))
        z = self.reparameterize(mu, logvar)
        state['mu'] = mu
        state['logvar'] = logvar
        return self.decode(z)
```

The reconstruction loss is a standard loss taking network output and the true label

```
loss = bce_loss
```

Since loss functions cannot access state, we utilise a simple callback to combine the kld loss which does not act on network output or true label.

```
@torchbearer.callbacks.add_to_loss
def add_kld_loss_callback(state):
    KLD = kld_Loss(state['mu'], state['logvar'])
    return KLD
```

5.4 Visualising Results

For auto-encoding problems it is often useful to visualise the reconstructions. We can do this in torchbearer by using another simple callback. We stack the first 8 images from the first validation batch and pass them to torchvisions save_image function which saves out visualisations.

```
def save_reconstruction_callback(num_images=8, folder='results/'):
    import os
    os.makedirs(os.path.dirname(folder), exist_ok=True)

    @torchbearer.callbacks.on_step_validation
    def saver(state):
        if state[torchbearer.BATCH] == 0:
            data = state[torchbearer.X]
            recon_batch = state[torchbearer.Y_PRED]
            comparison = torch.cat([data[:num_images],
                                    recon_batch.view(128, 1, 28, 28)[:num_images]])
            save_image(comparison.cpu(),
                       str(folder) + 'reconstruction_' + str(state[torchbearer.
↪EPOCH]) + '.png', nrow=num_images)
    return saver
```

5.5 Training the Model

We train the model by creating a torchmodel and a torchbearermodel and calling fit_generator.

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

from torchbearer import Model

```

(continues on next page)

(continued from previous page)

```
torchbearer_model = Model(model, optimizer, loss, metrics=['loss']).to('cuda')
torchbearer_model.fit_generator(traingen, epochs=10, validation_generator=testgen,
                                callbacks=[add_kld_loss_callback, save_reconstruction_
↪callback()], pass_state=True)
```

The visualised results after ten epochs then look like this:

5.6 Source Code

The source code for the example are given below:

Standard:

    Download Python source code: vae_standard.py

Using state:

    Download Python source code: vae.py

CHAPTER 6

Training a GAN

We shall try to implement something more complicated using torchbearer - a Generative Adverserial Network (GAN). This tutorial is a modified version of the GAN from the brilliant collection of GAN implementations PyTorch_GAN by eriklindernoren on github.

6.1 Data and Constants

We first define all constants for the example.

```
epochs = 200
batch_size = 64
lr = 0.0002
nworkers = 8
latent_dim = 100
sample_interval = 400
img_shape = (1, 28, 28)
adversarial_loss = torch.nn.BCELoss()
device = 'cuda'
valid = torch.ones(batch_size, 1, device=device)
fake = torch.zeros(batch_size, 1, device=device)
```

We then define a number of state keys for convenience using `state_key()`. This is optional, however, it automatically avoids key conflicts.

```
GEN_IMGS = state_key('gen_imgs')
DISC_GEN = state_key('disc_gen')
DISC_GEN_DET = state_key('disc_gen_det')
DISC_REAL = state_key('disc_real')
G_LOSS = state_key('g_loss')
D_LOSS = state_key('d_loss')
```

We then define the dataset and dataloader - for this example, MNIST.

```
transform = transforms.Compose([
                        transforms.ToTensor(),
                        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
                    ])

dataset = datasets.MNIST('./data/mnist', train=True, download=True,␣
↪transform=transform)

dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, shuffle=True,
↪ drop_last=True)
```

6.2 Model

We use the generator and discriminator from PyTorch_GAN and combine them into a model that performs a single forward pass.

```
class GAN(nn.Module):
    def __init__(self):
        super().__init__()
        self.discriminator = Discriminator()
        self.generator = Generator()

    def forward(self, real_imgs, state):
        # Generator Forward
        z = Variable(torch.Tensor(np.random.normal(0, 1, (real_imgs.shape[0], latent_
↪dim)))).to(state[tb.DEVICE])
        state[GEN_IMGS] = self.generator(z)
        state[DISC_GEN] = self.discriminator(state[GEN_IMGS])
        # This clears the function graph built up for the discriminator
        self.discriminator.zero_grad()

        # Discriminator Forward
        state[DISC_GEN_DET] = self.discriminator(state[GEN_IMGS].detach())
        state[DISC_REAL] = self.discriminator(real_imgs)
```

Note that we have to be careful to remove the gradient information from the discriminator after doing the generator forward pass.

6.3 Loss

Since our loss computation in this example is complicated, we shall forgo the basic loss criterion used in normal torchbearer models. Instead we use a callback to provide the loss, in this case we use the *add_to_loss()* callback decorator. This decorates a function that returns a loss and automatically adds this to the main loss in training and validation.

```
@callbacks.add_to_loss
def loss_callback(state):
    fake_loss = adversarial_loss(state[DISC_GEN_DET], fake)
    real_loss = adversarial_loss(state[DISC_REAL], valid)
    state[G_LOSS] = adversarial_loss(state[DISC_GEN], valid)
    state[D_LOSS] = (real_loss + fake_loss) / 2
    return state[G_LOSS] + state[D_LOSS]
```

Note that we have summed the separate discriminator and generator losses, since their graphs are separated, this is allowable.

6.4 Metrics

We would like to follow the discriminator and generator losses during training - note that we added these to state during the model definition. We can then create metrics from these by decorating simple state fetcher metrics.

```
@tb.metrics.mean
class g_loss(tb.metrics.Metric):
    def __init__(self):
        super().__init__(G_LOSS)

    def process(self, state):
        return state[G_LOSS]
```

6.5 Training

We can then train the torchbearer model on the GPU in the standard way. Note that when torchbearer is passed a `None` criterion it automatically sets the base loss to 0.

```
torchbearermodel = tb.Model(model, optim, criterion=None, metrics=['loss', g_loss(),␣
↪d_loss()])
torchbearermodel.to(device)
torchbearermodel.fit_generator(dataloader, epochs=200, pass_state=True,␣
↪callbacks=[loss_callback, saver_callback])
```

6.6 Visualising

We borrow the image saving method from PyTorch_GAN and put it in a call back to save *`on_step_training()`* - again using decorators.

```
@callbacks.on_step_training
def saver_callback(state):
    batches_done = state[tb.EPOCH] * len(state[tb.GENERATOR]) + state[tb.BATCH]
    if batches_done % sample_interval == 0:
        save_image(state[GEN_IMGS].data[:25], 'images/%d.png' % batches_done, nrow=5,␣
↪normalize=True)
```

Here is a Gif created from the saved images.

6.7 Source Code

The source code for the example is given below:

    Download Python source code: gan.py

CHAPTER 7

Optimising functions

Now for something a bit different. PyTorch is a tensor processing library and whilst it has a focus on neural networks, it can also be used for more standard funciton optimisation. In this example we will use torchbearer to minimise a simple function.

7.1 The Model

First we will need to create something that looks very similar to a neural network model - but with the purpose of minimising our function. We store the current estimates for the minimum as parameters in the model (so PyTorch optimisers can find and optimise them) and we return the function value in the forward method.

```
class Net(Module):
    def __init__(self, x):
        super().__init__()
        self.pars = torch.nn.Parameter(x)

    def f(self):
        """
        function to be minimised:
        f(x) = (x[0]-5)^2 + x[1]^2 + (x[2]-1)^2
        Solution:
        x = [5,0,1]
        """
        out = torch.zeros_like(self.pars)
        out[0] = self.pars[0]-5
        out[1] = self.pars[1]
        out[2] = self.pars[2]-1
        return torch.sum(out**2)

    def forward(self, _, state):
        state['est'] = self.pars
        return self.f()
```

7.2 The Loss

For function minimisation we have an analogue to neural network losses - we minimise the value of the function under the current estimates of the minimum. Note that as we are using a base loss, torchbearer passes this the network output and the "label" (which is of no use here).

```
def loss(y_pred, y_true):
    return y_pred
```

7.3 Optimising

We need two more things before we can start optimising with torchbearer. We need our initial guess - which we've set to [2.0, 1.0, 10.0] and we need to tell torchbearer how "long" an epoch is - I.e. how many optimisation steps we want for each epoch. For our simple function, we can complete the optimisation in a single epoch, but for more complex optimisations we might want to take multiple epochs and include tensorboard logging and perhaps learning rate annealing to find a final solution. We have set the number of optimisation steps for this example as 50000.

```
p = torch.tensor([2.0, 1.0, 10.0])
training_steps = 50000
```

The learning rate chosen for this example is very low and we could get convergence much faster with a larger rate, however this allows us to view convergence in real time. We define the model and optimiser in the standard way.

```
model = Net(p)
optim = torch.optim.SGD(model.parameters(), lr=0.0001)
```

Finally we start the optimising on the GPU and print the final minimum estimate.

```
tbmodel = tb.Model(model, optim, loss, [est(), 'loss']).to('cuda')
tbmodel.fit_generator(None, pass_state=True, train_steps=training_steps)
print(list(model.parameters())[0].data)
```

Usually torchbearer will infer the number of training steps from the data generator. Since for this example we have no data to give the model (which will be passed *None*), we need to tell torchbearer how many steps to run by setting the `training_steps` argument.

7.4 Viewing Progress

You might have noticed in the previous snippet that the example uses a metric we've not seen before. This simple metric is used to display the estimate throughout the optimisation process - although this is probably only useful for very small optimisation problems.

```
@tb.metrics.to_dict
class est(tb.metrics.Metric):
    def __init__(self):
        super().__init__('est')

    def process(self, state):
        return state['est'].data
```

The final estimate is very close to our desired minimum at [5, 0, 1]:

tensor([4.9988e+00, 4.5355e-05, 1.0004e+00])

7.5 Source Code

The source code for the example is given below:

    Download Python source code: basic_opt.py

CHAPTER 8

Linear Support Vector Machine (SVM)

We've seen how to frame a problem as a differentiable program in the Optimising Functions example. Now we can take a look a more usable example; a linear Support Vector Machine (SVM). Note that the model and loss used in this guide are based on the code found here.

8.1 SVM Recap

Recall that an SVM tries to find the maximum margin hyperplane which separates the data classes. For a soft margin SVM where $\mathbf{x}$ is our data, we minimize:

$$\left[\frac{1}{n}\sum_{i=1}^{n}\max\left(0, 1 - y_i(\mathbf{w}\cdot\mathbf{x}_i - b)\right)\right] + \lambda\|\mathbf{w}\|^2$$

We can formulate this as an optimization over our weights $\mathbf{w}$ and bias b, where we minimize the hinge loss subject to a level 2 weight decay term. The hinge loss for some model outputs $z = \mathbf{w}\mathbf{x} + b$ with targets y is given by:

$$\ell(y, z) = \max\left(0, 1 - yz\right)$$

8.2 Defining the Model

Let's put this into code. First we can define our module which will project the data through our weights and offset by a bias. Note that this is identical to the function of a linear layer.

```
class LinearSVM(nn.Module):
    """Support Vector Machine"""

    def __init__(self):
        super(LinearSVM, self).__init__()
        self.w = nn.Parameter(torch.randn(1, 2), requires_grad=True)
        self.b = nn.Parameter(torch.randn(1), requires_grad=True)

    def forward(self, x):
```

(continues on next page)

(continued from previous page)

```
        h = x.matmul(self.w.t()) + self.b
        return h
```

Next, we define the hinge loss function:

```
def hinge_loss(y_pred, y_true):
    return torch.mean(torch.clamp(1 - y_pred.t() * y_true, min=0))
```

8.3 Creating Synthetic Data

Now for some data, 1024 samples should do the trick. We normalise here so that our random init is in the same space as the data:

```
X, Y = make_blobs(n_samples=1024, centers=2, cluster_std=1.2, random_state=1)
X = (X - X.mean()) / X.std()
Y[np.where(Y == 0)] = -1
X, Y = torch.FloatTensor(X), torch.FloatTensor(Y)
```

8.4 Subgradient Descent

Since we don't know that our data is linearly separable, we would like to use a soft-margin SVM. That is, an SVM for which the data does not all have to be outside of the margin. This takes the form of a weight decay term, $\lambda\|\mathbf{w}\|^2$ in the above equation. This term is called weight decay because the gradient corresponds to subtracting some amount $(2\lambda\mathbf{w})$ from our weights at each step. With torchbearer we can use the *L2WeightDecay* callback to do this. This whole process is known as subgradient descent because we only use a mini-batch (of size 32 in our example) at each step to approximate the gradient over all of the data. This is proven to converge to the minimum for convex functions such as our SVM. At this point we are ready to create and train our model:

```
svm = LinearSVM()
model = Model(svm, optim.SGD(svm.parameters(), 0.1), hinge_loss, ['loss']).to('cuda')

model.fit(X, Y, batch_size=32, epochs=50, verbose=1,
                  callbacks=[scatter,
                             draw_margin,
                             ExponentialLR(0.999, step_on_batch=True),
                             L2WeightDecay(0.01, params=[svm.w])])
```

8.5 Visualizing the Training

You might have noticed some strange things in that call to *Model.fit()*. Specifically, we use the *ExponentialLR* callback to anneal the convergence a little and we have a couple of other callbacks: `scatter` and `draw_margin`. These callbacks produce the following live visualisation (note, doesn't work in PyCharm, best run from terminal):

The code for the visualisation (using pyplot) is a bit ugly but we'll try to explain it to some degree. First, we need a mesh grid `xy` over the range of our data:

```
delta = 0.01
x = np.arange(X[:, 0].min(), X[:, 0].max(), delta)
y = np.arange(X[:, 1].min(), X[:, 1].max(), delta)
x, y = np.meshgrid(x, y)
xy = list(map(np.ravel, [x, y]))
```

Next, we have the scatter callback. This happens once at the start of our fit call and draws the figure with a scatter plot of our data:

```
@callbacks.on_start
def scatter(_):
    plt.figure(figsize=(5, 5))
    plt.ion()
    plt.scatter(x=X[:, 0], y=X[:, 1], c="black", s=10)
```

Now things get a little strange. We start by evaluating our model over the mesh grid from earlier:

```
@callbacks.on_step_training
def draw_margin(state):
    if state[torchbearer.BATCH] % 10 == 0:
        w = state[torchbearer.MODEL].w[0].detach().to('cpu').numpy()
        b = state[torchbearer.MODEL].b[0].detach().to('cpu').numpy()
```

For our outputs $z \in \mathbf{Z}$, we can make some observations about the decision boundary. First, that we are outside the margin if $z - 1$ or $z 1$. Conversely, we are inside the margine where $z - 1$ or $z 1$. This gives us some rules for colouring, which we use here:

```
        z = (w.dot(xy) + b).reshape(x.shape)
        z[np.where(z > 1.)] = 4
        z[np.where((z > 0.) & (z <= 1.))] = 3
        z[np.where((z > -1.) & (z <= 0.))] = 2
        z[np.where(z <= -1.)] = 1
```

So far it's been relatively straight forward. The next bit is a bit of a hack to get the update of the contour plot working. If a reference to the plot is already in state we just remove the old one and add a new one, otherwise we add it and show the plot. Finally, we call `mypause` to trigger an update. You could just use `plt.pause`, however, it grabs the mouse focus each time it is called which can be annoying. Instead, `mypause` is taken from stackoverflow.

```
        if 'contour' in state:
            for coll in state['contour'].collections:
                coll.remove()
            state['contour'] = plt.contourf(x, y, z, cmap=plt.cm.jet, alpha=0.5)
        else:
            state['contour'] = plt.contourf(x, y, z, cmap=plt.cm.jet, alpha=0.5)
            plt.tight_layout()
            plt.show()

        mypause(0.001)
```

8.6 Final Comments

So, there you have it, a fun differentiable programming example with a live visualisation in under 100 lines of code with torchbearer. It's easy to see how this could become more useful, perhaps finding a way to use the kernel trick with the standard form of an SVM (essentially an RBF network). You could also attempt to write some code that saves the gif from earlier. We had some but it was beyond a hack, can you do better?

8.7 Source Code

The source code for the example is given below:

    Download Python source code: svm_linear.py

CHAPTER 9

Breaking ADAM

In case you haven't heard, one of the top papers at ICLR 2018 (pronounced: eye-clear, who knew?) was On the Convergence of Adam and Beyond. In the paper, the authors determine a flaw in the convergence proof of the ubiquitous ADAM optimizer. They also give an example of a simple function for which ADAM does not converge to the correct solution. We've seen how torchbearer can be used for simple function optimization before and we can do something similar to reproduce the results from the paper.

9.1 Online Optimization

Online learning basically just means learning from one example at a time, in sequence. The function given in the paper is defined as follows:

$$f_t(x) = \begin{cases} 1010x, & \text{for } t \bmod 101 = 1 \\ -10x, & \text{otherwise} \end{cases}$$

We can then write this as a PyTorch model whose forward is a function of its parameters with the following:

```
class Online(Module):
    def __init__(self):
        super().__init__()
        self.x = torch.nn.Parameter(torch.zeros(1))

    def forward(self, _, state):
        """
        function to be minimised:
        f(x) = 1010x if t mod 101 = 1, else -10x
        """
        if state[tb.BATCH] % 101 == 1:
            res = 1010 * self.x
        else:
            res = -10 * self.x

        return res
```

We now define a loss (simply return the model output) and a metric which returns the value of our parameter x:

```
def loss(y_pred, _):
    return y_pred


@tb.metrics.to_dict
class est(tb.metrics.Metric):
    def __init__(self):
        super().__init__('est')

    def process(self, state):
        return state[tb.MODEL].x.data
```

In the paper, x can only hold values in $[-1, 1]$. We don't strictly need to do anything but we can write a callback that greedily updates x if it is outside of its range as follows:

```
@tb.callbacks.on_step_training
def greedy_update(state):
    if state[tb.MODEL].x > 1:
        state[tb.MODEL].x.data.fill_(1)
    elif state[tb.MODEL].x < -1:
        state[tb.MODEL].x.data.fill_(-1)
```

Finally, we can train this model twice; once with ADAM and once with AMSGrad (included in PyTorch) with just a few lines:

```
training_steps = 6000000

model = Online()

optim = torch.optim.Adam(model.parameters(), lr=0.001, betas=[0.9, 0.99])
tbmodel = tb.Model(model, optim, loss, [est()])
tbmodel.fit_generator(None, pass_state=True, train_steps=training_steps,␣
↪callbacks=[greedy_update, TensorBoard(comment='adam', write_graph=False, write_
↪batch_metrics=True, write_epoch_metrics=False)])

optim = torch.optim.Adam(model.parameters(), lr=0.001, betas=[0.9, 0.99],␣
↪amsgrad=True)
tbmodel = tb.Model(model, optim, loss, [est()])
tbmodel.fit_generator(None, pass_state=True, train_steps=training_steps,␣
↪callbacks=[greedy_update, TensorBoard(comment='amsgrad', write_graph=False, write_
↪batch_metrics=True, write_epoch_metrics=False)])
```

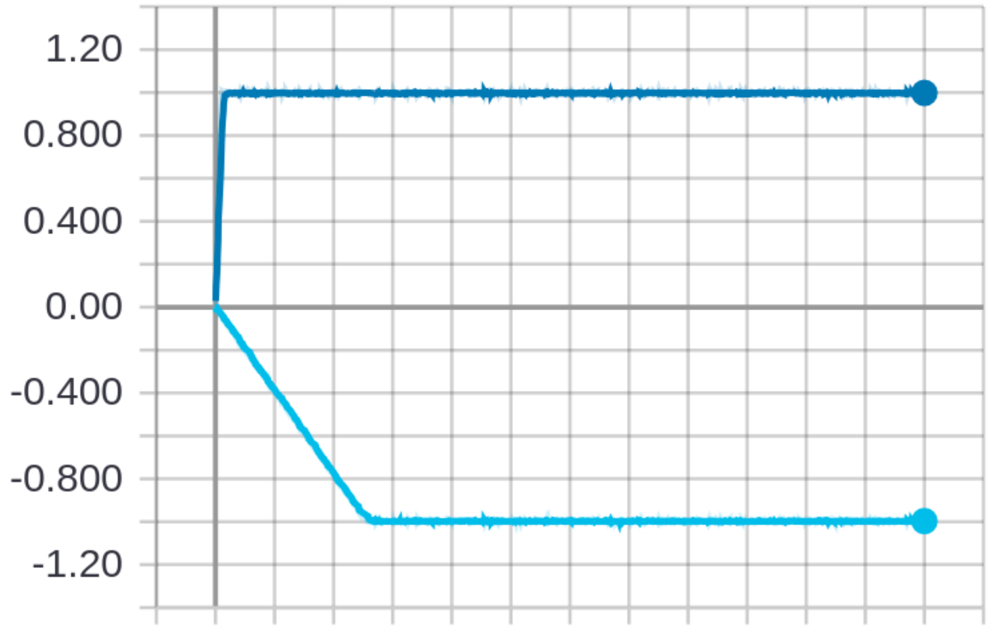
Note that we have logged to TensorBoard here and after completion, running `tensorboard --logdir logs` and navigating to localhost:6006, we can see a graph like the one in Figure 1 from the paper, where the top line is with ADAM and the bottom with AMSGrad:

9.2 Stochastic Optimization

To simulate a stochastic setting, the authors use a slight variant of the function, which changes with some probability:

$$f_t(x) = \begin{cases} 1010x, & \text{with probability } 0.01 \\ -10x, & \text{otherwise} \end{cases}$$

We can again formulate this as a PyToch model:



```
class Stochastic(Module):
    def __init__(self):
        super().__init__()
        self.x = torch.nn.Parameter(torch.zeros(1))

    def forward(self, _):
        """
        function to be minimised:
        f(x) = 1010x with probability 0.01, else -10x
        """
        if random.random() <= 0.01:
            res = 1010 * self.x
        else:
            res = -10 * self.x

        return res
```

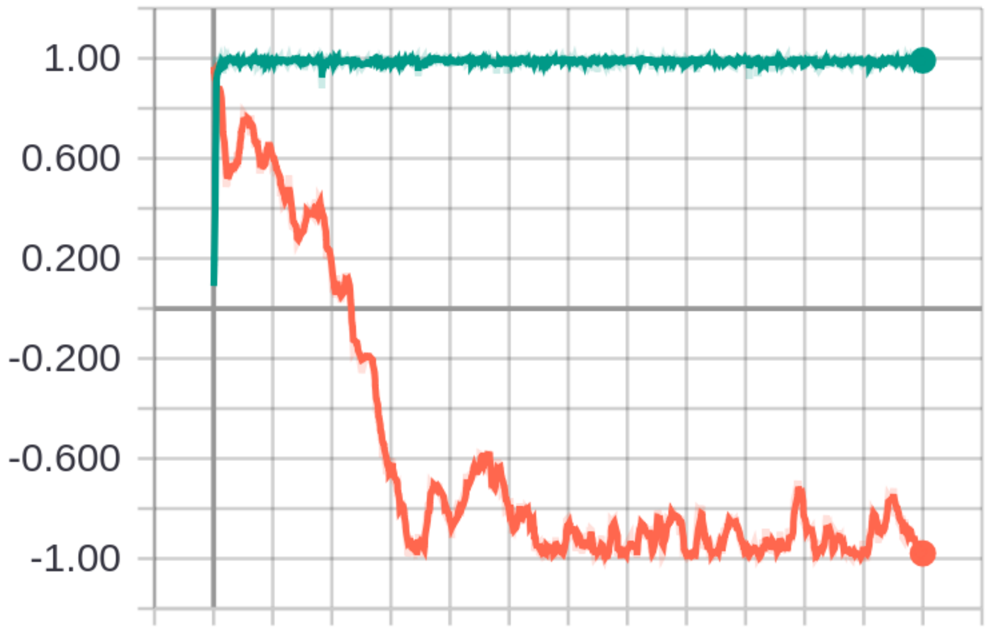
Using the loss, callback and metric from our previous example, we can train with the following:

```
model = Stochastic()

optim = torch.optim.Adam(model.parameters(), lr=0.001, betas=[0.9, 0.99])
tbmodel = tb.Model(model, optim, loss, [est()])
tbmodel.fit_generator(None, train_steps=training_steps, callbacks=[greedy_update,␣
↪TensorBoard(comment='adam', write_graph=False, write_batch_metrics=True, write_
↪epoch_metrics=False)])

optim = torch.optim.Adam(model.parameters(), lr=0.001, betas=[0.9, 0.99],␣
↪amsgrad=True)
tbmodel = tb.Model(model, optim, loss, [est()])
tbmodel.fit_generator(None, train_steps=training_steps, callbacks=[greedy_update,␣
↪TensorBoard(comment='amsgrad', write_graph=False, write_batch_metrics=True, write_
↪epoch_metrics=False)])
```

(continues on next page)

(continued from previous page)

After execution has finished, again running `tensorboard --logdir logs` and navigating to localhost:6006, we see another graph similar to that of the stochastic setting in Figure 1 of the paper, where the top line is with ADAM and the bottom with AMSGrad:



9.3 Conclusions

So, whatever your thoughts on the AMSGrad optimizer in practice, it's probably the sign of a good paper that you can re-implement the example and get very similar results without having to try too hard and (thanks to torchbearer) only writing a small amount of code. The paper includes some more complex, 'real-world' examples, can you re-implement those too?

9.4 Source Code

The source code for this example can be downloaded below:

    Download Python source code: amsgrad.py

CHAPTER 10

torchbearer

10.1 Model

class `torchbearer.torchbearer.`**`Model`**(*model*, *optimizer*, *criterion=None*, *metrics=[]*)

Create torchbearermodel which wraps a base torchmodel and provides a training environment surrounding it

Parameters

- **`model`** (*`torch.nn.Module`*) – The base pytorch model
- **`optimizer`** (*`torch.optim.Optimizer`*) – The optimizer used for pytorch model weight updates
- **`criterion`** (*`function or None`*) – The final loss criterion that provides a loss value to the optimizer
- **`metrics`** (*`list`*) – Additional metrics for display and use within callbacks

`cpu`()

Moves all model parameters and buffers to the CPU.

Returns Self torchbearermodel

Return type *Model*

`cuda`(*device=None*)

Moves all model parameters and buffers to the GPU.

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

Returns Self torchbearermodel

Return type *Model*

`eval`()

Set model and metrics to evaluation mode

evaluate (*x=None*, *y=None*, *batch_size=32*, *verbose=2*, *steps=None*, *pass_state=False*)
Perform an evaluation loop on given data and label tensors to evaluate metrics

Parameters

- **x** (*torch.Tensor*) – The input data tensor
- **y** (*torch.Tensor*) – The target labels for data tensor x
- **batch_size** (*int*) – The mini-batch size (number of samples processed for a single weight update)
- **verbose** (*int*) – If 2: use tqdm on batch, If 1: use tqdm on epoch, Else: display no progress
- **steps** (*int*) – The number of evaluation mini-batches to run
- **pass_state** (*bool*) – If True the state dictionary is passed to the torch model forward method, if False only the input data is passed

Returns The dictionary containing final metrics

Return type dict[str,any]

evaluate_generator (*generator*, *verbose=2*, *steps=None*, *pass_state=False*)
Perform an evaluation loop on given data generator to evaluate metrics

Parameters

- **generator** (*DataLoader*) – The evaluation data generator (usually a pytorch DataLoader)
- **verbose** (*int*) – If 2: use tqdm on batch, If 1: use tqdm on epoch, Else: display no progress
- **steps** (*int*) – The number of evaluation mini-batches to run
- **pass_state** (*bool*) – If True the state dictionary is passed to the torch model forward method, if False only the input data is passed

Returns The dictionary containing final metrics

Return type dict[str,any]

fit (*x*, *y*, *batch_size=None*, *epochs=1*, *verbose=2*, *callbacks=[]*, *validation_split=None*, *validation_data=None*, *shuffle=True*, *initial_epoch=0*, *steps_per_epoch=None*, *validation_steps=None*, *workers=1*, *pass_state=False*)
Perform fitting of a model to given data and label tensors

Parameters

- **x** (*torch.Tensor*) – The input data tensor
- **y** (*torch.Tensor*) – The target labels for data tensor x
- **batch_size** (*int*) – The mini-batch size (number of samples processed for a single weight update)
- **epochs** (*int*) – The number of training epochs to be run (each sample from the dataset is viewed exactly once)
- **verbose** (*int*) – If 2: use tqdm on batch, If 1: use tqdm on epoch, Else: display no training progress
- **callbacks** (*list*) – The list of torchbearer callbacks to be called during training and validation

- **validation_split** (*float*) – Fraction of the training dataset to be set aside for validation testing
- **validation_data** (*(torch.Tensor, torch.Tensor)*) – Optional validation data tensor
- **shuffle** (*bool*) – If True mini-batches of training/validation data are randomly selected, if False mini-batches samples are selected in order defined by dataset
- **initial_epoch** (*int*) – The integer value representing the first epoch - useful for continuing training after a number of epochs
- **steps_per_epoch** (*int*) – The number of training mini-batches to run per epoch
- **validation_steps** (*int*) – The number of validation mini-batches to run per epoch
- **workers** (*int*) – The number of cpu workers devoted to batch loading and aggregating
- **pass_state** (*bool*) – If True the state dictionary is passed to the torch model forward method, if False only the input data is passed

Returns The final state context dictionary

Return type dict[str,any]

fit_generator(*generator*, *train_steps=None*, *epochs=1*, *verbose=2*, *callbacks=[]*, *validation_generator=None*, *validation_steps=None*, *initial_epoch=0*, *pass_state=False*)

Perform fitting of a model to given data generator

Parameters

- **generator** (*DataLoader*) – The training data generator (usually a pytorch DataLoader)
- **train_steps** (*int*) – The number of training mini-batches to run per epoch
- **epochs** (*int*) – The number of training epochs to be run (each sample from the dataset is viewed exactly once)
- **verbose** (*int*) – If 2: use tqdm on batch, If 1: use tqdm on epoch, Else: display no training progress
- **callbacks** (*list*) – The list of torchbearer callbacks to be called during training and validation
- **validation_generator** (*DataLoader*) – The validation data generator (usually a pytorch DataLoader)
- **validation_steps** (*int*) – The number of validation mini-batches to run per epoch
- **initial_epoch** (*int*) – The integer value representing the first epoch - useful for continuing training after a number of epochs
- **pass_state** (*bool*) – If True the state dictionary is passed to the torch model forward method, if False only the input data is passed

Returns The final state context dictionary

Return type dict[str,any]

load_state_dict(*state_dict*, ***kwargs*)

Copies parameters and buffers from `state_dict()` into this module and its descendants.

Parameters

- **state_dict** (*dict*) – A dict containing parameters and persistent buffers.

- **kwargs** – See: torch.nn.Module.load_state_dict

predict(*x=None*, *batch_size=32*, *verbose=2*, *steps=None*, *pass_state=False*)
Perform a prediction loop on given data tensor to predict labels

Parameters

- **x** (*torch.Tensor*) – The input data tensor
- **batch_size** (*int*) – The mini-batch size (number of samples processed for a single weight update)
- **verbose** (*int*) – If 2: use tqdm on batch, If 1: use tqdm on epoch, Else: display no progress
- **steps** (*int*) – The number of evaluation mini-batches to run
- **pass_state** (*bool*) – If True the state dictionary is passed to the torch model forward method, if False only the input data is passed

Returns Tensor of final predicted labels

Return type torch.Tensor

predict_generator(*generator*, *verbose=2*, *steps=None*, *pass_state=False*)
Perform a prediction loop on given data generator to predict labels

Parameters

- **generator** (*DataLoader*) – The prediction data generator (usually a pytorch DataLoader)
- **verbose** (*int*) – If 2: use tqdm on batch, If 1: use tqdm on epoch, Else: display no progress
- **steps** (*int*) – The number of evaluation mini-batches to run
- **pass_state** (*bool*) – If True the state dictionary is passed to the torch model forward method, if False only the input data is passed

Returns Tensor of final predicted labels

Return type torch.Tensor

state_dict(***kwargs*)

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

Returns A dict containing parameters and persistent buffers.

Return type dict

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

Parameters

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

Returns Self torchbearermodel

Return type *Model*

train()
Set model and metrics to training mode

10.2 Utilities

`torchbearer.state.`**`state_key`**(*key*)

Computes and returns a non-conflicting key for the state dictionary when given a seed key

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

Returns New state key

Return type String

class `torchbearer.cv_utils.`**`DatasetValidationSplitter`**(*dataset_len*, *split_fraction*, *shuffle_seed=None*)

`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

Parameters **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

`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

Return type tuple

`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)

Return type tuple

CHAPTER 11

torchbearer.callbacks

class torchbearer.callbacks.callbacks.**Callback**

Base callback class.

Note: All callbacks should override this class.

on_backward(*state*)

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

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

on_criterion(*state*)

Perform some action with the given state as context after the criterion has been evaluated.

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

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[str, any]*) – The current state dict of the *Model*.

on_end(*state*)

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

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

on_end_epoch(*state*)

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

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

on_end_training(*state*)

Perform some action with the given state as context after the training loop has completed.

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

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

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

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

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

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[str, any]*) – The current state dict of the *Model*.

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

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

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[str, any]*) – The current state dict of the *Model*.

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

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

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

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

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

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

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

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

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

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

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

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

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.

:param callback_list:The list of callbacks to be wrapped. If the list contains a *CallbackList*, this will be unwrapped. :type callback_list:list

append(*callback_list*)

copy()

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 *Model*.

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 *Model*.

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 *Model*.

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 *Model*.

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 *Model*.

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 *Model*.

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 *Model*.

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 *Model*.

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 *Model*.

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 *Model*.

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 *Model*.

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 *Model*.

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 *Model*.

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 `Model`.

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 `Model`.

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 `Model`.

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 `Model`.

11.1 Model Checkpointers

class torchbearer.callbacks.checkpointers.**Best**(*filepath='model.{epoch:02d}-{val_loss:.2f}.pt', monitor='val_loss', mode='auto', period=1, min_delta=0, pickle_module=<MagicMock name='mock.pickle' id='139800635575656'>, pickle_protocol=<MagicMock name='mock.DEFAULT_PROTOCOL' id='139800635542720'>*)
Model checkpointer which saves the best model according to the given configurations.

Parameters

- **filepath** (*str*) – Path to save the model file
- **monitor** (*str*) – Quantity to monitor
- **mode** (*str*) – One of {auto, min, max}. 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*) – This is the minimum improvement required to trigger a save
- **pickle_module** – The pickle module to use, default is 'torch.serialization.pickle'
- **pickle_protocol** – The pickle protocol to use, default is 'torch.serialization.DEFAULT_PROTOCOL'

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

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

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

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

class torchbearer.callbacks.checkpointers.**Interval**(*filepath='model.{epoch:02d}-{val_loss:.2f}.pt', period=1, pickle_module=<MagicMock name='mock.pickle' id='139800635477800'>, pickle_protocol=<MagicMock name='mock.DEFAULT_PROTOCOL' id='139800635494640'>*)

Model checkpointer which which saves the model every 'period' epochs to the given filepath.

Parameters

- **filepath** (*str*) – Path to save the model file
- **period** (*int*) – Interval (number of epochs) between checkpoints
- **pickle_module** – The pickle module to use, default is 'torch.serialization.pickle'
- **pickle_protocol** – The pickle protocol to use, default is 'torch.serialization.DEFAULT_PROTOCOL'

on_end_epoch(*model_state*)

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

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

torchbearer.callbacks.checkpointers.**ModelCheckpoint**(*filepath='model.{epoch:02d}-{val_loss:.2f}.pt', 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 model will be saved to filename.pt and the torchbearermodel state will be saved to filename.torchbearer.

Parameters

- **filepath** (*str*) – Path to save the model file
- **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

class torchbearer.callbacks.checkpointers.**MostRecent**(*filepath='model.{epoch:02d}-{val_loss:.2f}.pt'*, *pickle_module=<MagicMock name='mock.pickle' id='139800635298536'>*, *pickle_protocol=<MagicMock name='mock.DEFAULT_PROTOCOL' id='139800635223456'>*)

Model checkpointer which saves the most recent model to a given filepath.

Parameters

- **filepath** (*str*) – Path to save the model file
- **pickle_module** – The pickle module to use, default is 'torch.serialization.pickle'
- **pickle_protocol** – The pickle protocol to use, default is 'torch.serialization.DEFAULT_PROTOCOL'

on_end_epoch(*state*)

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

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

11.2 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.

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

on_end(*state*)

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

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

on_end_epoch(*state*)

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

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

on_step_training(*state*)

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

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

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

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

Parameters validation_label_letter (*String*) – This is the letter displayed after the epoch number indicating the current phase of training

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

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

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

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

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

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

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

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

class torchbearer.callbacks.printer.**Tqdm** (*tqdm_module=<MagicMock id='139800635314864'>, validation_label_letter='v', on_epoch=False*)
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.

Parameters

- **validation_label_letter** (*str*) – The letter to use for validation outputs.
- **on_epoch** (*bool*) – If True, output a single progress bar which tracks epochs

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

11.3 Tensorboard

class torchbearer.callbacks.tensor_board.**AbstractTensorBoard**(*log_dir='./logs', comment='torchbearer', visdom=False*)
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

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. :param log_dir: the (optional) directory :type log_dir: str

get_writer(*log_dir=None, visdom=False*)
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*. :param log_dir: the (optional) directory :type log_dir: str :param visdom: If true, return VisdomWriter, if false return tensorboard SummaryWriter :type visdom: bool :return: 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[str, any]*) – The current state dict of the Model.

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

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

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*)
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

- **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** (*True*) – 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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** (*str*) – 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 https://pytorch.org/docs/stable/torchvision/utils.html#torchvision.utils.make*
- **padding** – See *torchvision.utils.make_grid https://pytorch.org/docs/stable/torchvision/utils.html#torchvision.utils.*
- **normalize** – See *torchvision.utils.make_grid https://pytorch.org/docs/stable/torchvision/utils.html#torchvision.ut*
- **norm_range** – See *torchvision.utils.make_grid https://pytorch.org/docs/stable/torchvision/utils.html#torchvision.*
- **scale_each** – See *torchvision.utils.make_grid https://pytorch.org/docs/stable/torchvision/utils.html#torchvision.*
- **pad_value** – See *torchvision.utils.make_grid https://pytorch.org/docs/stable/torchvision/utils.html#torchvision.ut*
- **visdom** (*bool*) – If true, log to visdom instead of tensorboard

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

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

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

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

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`** (*`str`*) – 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.

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

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

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

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

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`** – the log directory
- **`logger`** – the object releasing the writer

`torchbearer.callbacks.tensor_board.`**`get_writer`**(*log_dir*, *logger*, *visdom=False*)
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`** – the log directory
- **`logger`** – the object requesting the writer. That object should call *close_writer* when its finished
- **`visdom`** – if true VisdomWriter is returned instead of tensorboard SummaryWriter

Returns the *SummaryWriter* or *VisdomWriter* object

11.4 Early Stopping

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

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

Parameters

- **`monitor`** (*`str`*) – Quantity 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.
- **`verbose`** (*`int`*) – Verbosity mode, will print stopping info if verbose > 0
- **`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.

`on_end`(*state*)

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

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

`on_end_epoch`(*state*)

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

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

`on_start`(*state*)

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

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

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.

Parameters **`monitor`** (*`str`*) – The metric name to monitor

`on_end_epoch`(*state*)

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

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

`on_step_training`(*state*)

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

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

`on_step_validation`(*state*)

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

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

11.5 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.

Parameters

- **clip_value** – The maximum absolute value of the gradient
- **params** – The parameters to clip or None

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 Model state

on_start(*state*)

If params is None then retrieve from the model.

Parameters **state** (`dict`) – The Model 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.

Parameters

- **max_norm** – The max norm value
- **norm_type** – The norm type to use
- **params** – The parameters to clip or None

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 Model state

on_start(*state*)

If params is None then retrieve from the model.

Parameters **state** (`dict`) – The Model state

11.6 Learning Rate Schedulers

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

See: PyTorch CosineAnnealingLR

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

See: PyTorch ExponentialLR

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

See: PyTorch LambdaLR

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

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*)

Parameters **`monitor`** (*`str`*) – The quantity to monitor. (Default value = 'val_loss')

See: PyTorch ReduceLROnPlateau

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

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[str, any]`*) – The current state dict of the `Model`.

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

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

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

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

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

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

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

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

11.7 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.

Parameters

- **`rate`** (*`float`*) – The decay rate
- **`params`** (*`list`*) – The parameters to use (or None)

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.

Parameters

- **`rate`** (*`float`*) – The decay rate
- **`params`** (*`list`*) – The parameters to use (or None)

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.

Parameters

- **`rate`** (*`float`*) – The decay rate
- **`p`** (*`int`*) – The norm level
- **`params`** (*`list`*) – The parameters to use (or None)

`on_criterion`(*state*)

Calculate the decay term and add to state['loss'].

Parameters **`state`** (*`dict`*) – The Model state

`on_start`(*state*)

Retrieve params from state['model'] if required.

Parameters **`state`** (*`dict`*) – The Model state

11.8 Decorators

`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

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.`**`on_backward`**(*func*)

The *`on_backward()`* decorator is used to initialise a *`Callback`* with *`on_backward()`* calling the decorated function

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *Callback.on_backward()* 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

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *Callback.on_criterion()* 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

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *Callback.on_criterion_validation()* 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

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *Callback.on_end()* 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

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *Callback.on_end_epoch()* 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

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *Callback.on_end_training()* 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

Parameters **func** (*function*) – The function(state) to *decorate*

Returns Initialised callback with *Callback.on_end_validation()* 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

Parameters **`func`** (`function`) – The function(state) to *decorate*

Returns Initialised callback with `Callback.on_forward()` 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

Parameters **`func`** (`function`) – The function(state) to *decorate*

Returns Initialised callback with `Callback.on_forward_validation()` 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

Parameters **`func`** (`function`) – The function(state) to *decorate*

Returns Initialised callback with `Callback.on_sample()` 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

Parameters **`func`** (`function`) – The function(state) to *decorate*

Returns Initialised callback with `Callback.on_sample_validation()` 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

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

Parameters **`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

Parameters **`func`** (`function`) – The function(state) to *decorate*

Returns Initialised callback with `Callback.on_start_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

Parameters **`func`** (*`function`*) – The function(state) to *decorate*

Returns Initialised callback with `Callback.on_start_validation()` 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

Parameters **`func`** (*`function`*) – The function(state) to *decorate*

Returns Initialised callback with `Callback.on_step_training()` 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

Parameters **`func`** (*`function`*) – The function(state) to *decorate*

Returns Initialised callback with `Callback.on_step_validation()` calling func

Return type `Callback`

`torchbearer.callbacks.decorators.`**`once`**(*fcn*)

Decorator to fire a callback once in the entire fitting procedure. :param fcn: the *torchbearer callback* function to decorate. :return: the decorator

`torchbearer.callbacks.decorators.`**`once_per_epoch`**(*fcn*)

Decorator to fire a callback once (on the first call) in any given epoch. :param fcn: the *torchbearer callback* function to decorate. :return: the decorator

`torchbearer.callbacks.decorators.`**`only_if`**(*condition_expr*)

Decorator to fire a callback only if the given conditional expression function returns True. :param condition_expr: a function/lambda 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. :return: the decorator

CHAPTER 12

torchbearer.metrics

12.1 Base Classes

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*.

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()

Put the metric in eval mode.

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.`**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()
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** – The current state dict of the `Model`.

train()
Put the metric in train mode during model training.

class `torchbearer.metrics.metrics.`**MetricFactory**
A simple implementation of a factory pattern. Used to enable construction of complex metrics using decorators.

build()
Build and return a usable `Metric` instance.

Returns The constructed `Metric`

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()
Put each metric in eval mode

process(**args*)
Process each metric an wrap in a dictionary which maps metric names to values.

Returns dict[str,any] – A dictionary which maps metric names to values.

process_final(**args*)
Process each metric an wrap in a dictionary which maps metric names to values.

Returns dict[str,any] – A dictionary which maps metric names to values.

reset(*state*)
Reset each metric with the given state.

Parameters **state** – The current state dict of the *Model*.

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.

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

Returns None

eval()
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** – The current state dict of the *Model*.

train()
Put the metric in train mode during model training.

12.2 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*)

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

`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. In fact it return a `MetricFactory` which is used to build a `Metric`. This can make things complicated 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()]) # Note we have to call `my_metric` in order␣
↪to instantiate the class
```

Parameters

- **name** – The name of the metric (e.g. 'loss')
- **on_epoch** – If True the metric will be an instance of `EpochLambda` instead of `BatchLambda`

Returns A decorator which replaces a function with a `MetricFactory`

`torchbearer.metrics.decorators.`**`mean`**(*clazz*)

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 my_metric(y_pred, y_true):
...     return y_pred + y_true
...
>>> metric = my_metric().build()
>>> 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
{}
```

(continues on next page)

(continued from previous page)

```
>>> 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*

Returns A `MetricFactory` which can be instantiated and built to append a `Mean` to the `MetricTree`

`torchbearer.metrics.decorators.`**`running_mean`**`(`*clazz=None*, *batch_size=50*, *step_size=10*`)`

The `running_mean()` decorator is used to add a `RunningMean` to the `MetricTree`. As with the other decorators, a `MetricFactory` is created which will do this upon the call to `MetricFactory.build()`. If the inner class is not / does not build 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 my_metric(y_pred, y_true):
...     return y_pred + y_true
...
>>> metric = my_metric().build()
>>> 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** – See `RunningMean`
- **step_size** – See `RunningMean`

Returns decorator or `MetricFactory`

`torchbearer.metrics.decorators.`**`std`**`(`*clazz*`)`

The `std()` decorator is used to add a `Std` to the `MetricTree` which will will output a population 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 my_metric(y_pred, y_true):
...     return y_pred + y_true
...
>>> metric = my_metric().build()
>>> 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']
'1.6330'
```

Parameters **`clazz`** – The class to *decorate*

Returns A `MetricFactory` which can be instantiated and built to append a `Mean` to the `MetricTree`

`torchbearer.metrics.decorators.`**`to_dict`**(*clazz*)

The `to_dict()` decorator is used to wrap either a `Metric` or `MetricFactory` 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().build().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().build().process({'y_pred':4, 'y_true':5})
{'my_metric': 9}
```

Parameters **`clazz`** – The class to *decorate*

Returns A `MetricFactory` which can be instantiated and built to wrap the given class in a `ToDict`

12.3 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** – A metric function('y_pred', 'y_true') to wrap.

`process`(**args*)

Return the output of the wrapped function.

Parameters **args** (`dict`) – The `torchbearer.Model` 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** – 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. If this is a running metric, evaluates the function every number of steps.

Parameters **args** (`dict`) – The `torchbearer.Model` state.

Returns The current running result.

`process_validate`(**args*)

During validation, just concatenate 'y_true' and y_pred'.

Parameters **args** (`dict`) – The `torchbearer.Model` state.

`reset`(*state*)

Reset the 'y_true' and 'y_pred' caches.

Parameters **state** (`dict`) – The `torchbearer.Model` 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** (`metrics.Metric`) – The `Metric` instance to *wrap*.

eval()

Put the metric in eval mode.

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** – The current state dict of the `Model`.

train()

Put the metric in train mode.

12.4 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*)

Metric aggregator which calculates the mean of process outputs between calls to reset.

Parameters **`name`** (*`str`*) – The name of this metric.

`process`(**args*)

Add the input to the rolling sum.

Parameters **`args`** (*`torch.Tensor`*) – 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*)

A *`RunningMetric`* which outputs the mean of a sequence of its input 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.

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*)

Metric aggregator which calculates the standard deviation of process outputs between calls to reset.

Parameters **`name`** (*`str`*) – The name of this metric.

process(**args*)
Compute values required for the std from the input.

Parameters **args** (`torch.Tensor`) – The output of some previous call to `Metric.process()`.

process_final(**args*)
Compute and return the final standard deviation.

Returns The standard deviation of each observation since the last reset call.

reset(*state*)
Reset the statistics to compute the next deviation.

Parameters **state** (`dict`) – The model state.

12.5 Base Metrics

Base metrics are the base classes which represent the metrics supplied with torchbearer. The 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 `CategoricalAccuracy` metric
- '*loss*': The `Loss` metric
- '*epoch*': The `Epoch` metric
- '*roc_auc*' or '*roc_auc_score*': The `RocAucScore` metric

class torchbearer.metrics.primitives.**CategoricalAccuracy**
Categorical accuracy metric. Uses torch.max to determine predictions and compares to targets.

class torchbearer.metrics.primitives.**Epoch**
Returns the 'epoch' from the model state.

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

class torchbearer.metrics.primitives.**Loss**
Simply returns the 'loss' value from the model state.

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

12.6 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[str, any]*) – The current state dict of the `Model`.

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

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

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[str, any]*) – The current state dict of the `Model`.

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

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

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

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

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

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

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

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

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

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

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[str, any]*) – The current state dict of the `Model`.

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

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

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[str, any]*) – The current state dict of the `Model`.

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

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

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

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

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

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

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

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

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

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

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

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

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** – The current state dict of the *Model*.

update_time(*text*, *metric*, *state*)

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index