# torchbearer Documentation

***Release 0.1.5***

**Ethan Harris and Matthew Painter**

**Jul 30, 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

# 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.

## 2.1 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.

## 2.2 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]
```

## 2.3 Source Code

The source code for the example is given below:

Download Python source code: `quickstart.py`

CHAPTER 3

# 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:

## 3.1 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
```

## 3.2 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)
```

## 3.3 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.

### 3.3.1 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
```

### 3.3.2 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
```

## 3.4 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
```

## 3.5 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:

## 3.6 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 4

## 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.

### 4.1 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. 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)
```

## 4.2 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])
        # We don't want to keep discriminator gradients on the generator forward pass
        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.

## 4.3 Loss

Since our loss is complicated in this example, we shall forgo the basic loss criterion used in normal torchbearer models.

```
def zero_loss(y_pred, y_true):
    return torch.zeros(y_true.shape[0], 1)
```

Instead use a callback to provide the loss. Since this callback is very simple we can use callback decorators on a function (which takes state) to tell torchbearer when it should be called.

```
@callbacks.on_criterion
def loss_callback(state):
    fake_loss = adversarial_loss(state[DISC_GEN_DET], fake)
    real_loss = adversarial_loss(state[DISC_REAL], valid)
```

(continues on next page)

(continued from previous page)

```
    state[G_LOSS] = adversarial_loss(state[DISC_GEN], valid)
    state[D_LOSS] = (real_loss + fake_loss) / 2
    # This is the loss that backward is called on.
    state[tb.LOSS] = 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.

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

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

## 4.5 Training

We can then train the torchbearer model on the GPU in the standard way.

```
torchbearermodel = tb.Model(model, optim, zero_loss, ['loss', g_loss(), d_loss()])
torchbearermodel.to(device)
torchbearermodel.fit_generator(dataloader, epochs=200, pass_state=True,␣
↪callbacks=[loss_callback, saver_callback])
```

## 4.6 Visualising

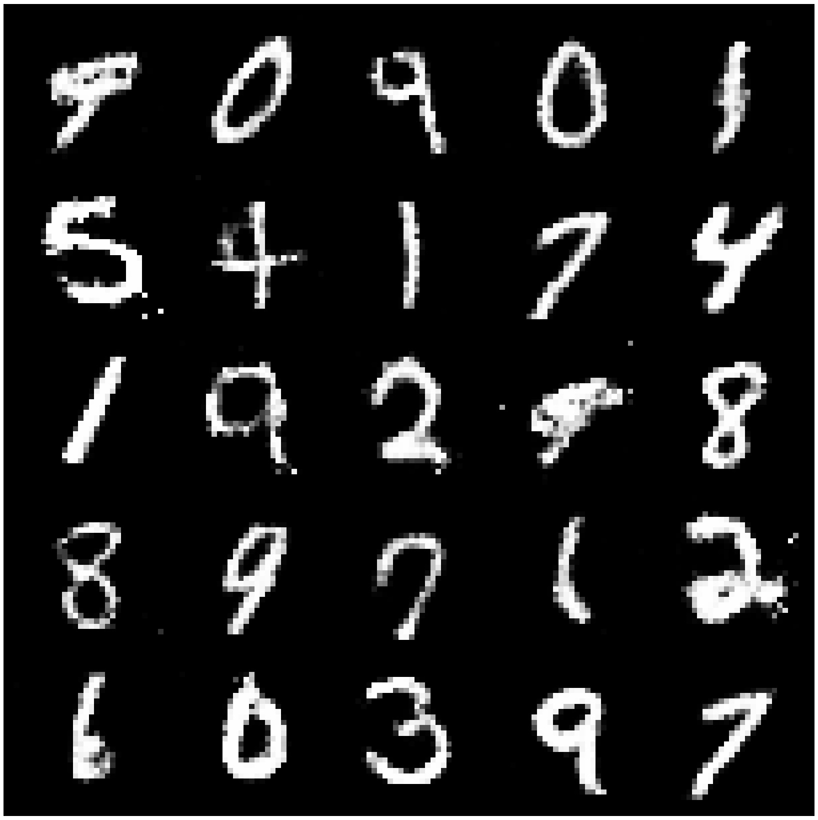
We borrow the image saving method from PyTorch_GAN and put it in a call back to save on training step - again using decorators.

```
@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)
```

After 172400 iterations we see the following.

## 4.7 Source Code

The source code for the example is given below:



`Download Python source code: gan.py`

# CHAPTER 5

# 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.

## 5.1 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()
```

## 5.2 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
```

## 5.3 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.

```
steps = torch.tensor(list(range(50000)))
p = torch.tensor([2.0, 1.0, 10.0])
```

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 (giving as "data" and "targets" the number of steps desired) and print the final minimum estimate.

```
tbmodel = tb.Model(model, optim, loss, [est(), 'loss'])
tbmodel.fit(steps, steps, 1, pass_state=True)
print(list(model.parameters())[0].data)
```

Note that we could use targets that are meaningful as they are given to the loss function, however this is not done for this example.

## 5.4 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])

# CHAPTER 6

# torchbearer

**class** torchbearer.torchbearer.**Model**(*model*, *optimizer*, *loss_criterion*, *metrics=[]*)
Torchbearermodel to wrap base torch model and provide training environment around it

**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=1*, *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 1 use tqdm progress frontend, else display no training 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=1*, *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 1 use tqdm progress frontend, else display no training 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=1*, *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 1 use tqdm progress frontend, 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=1*, *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 1 use tqdm progress frontend, 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=1*, *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 1 use tqdm progress frontend, else display no training 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=1*, *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 1 use tqdm progress frontend, else display no training 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

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

**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 7

# 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.

**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`.

## 7.1 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='140274768857240'>, pickle_protocol=<MagicMock name='mock.DEFAULT_PROTOCOL' id='140274768878112'>*)
Model checkpointer which saves the best model according to a metric.

**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='140274769271160'>, pickle_protocol=<MagicMock name='mock.DEFAULT_PROTOCOL' id='140274768920024'>*)
Model checkpointer which saves the model every given number of epochs.

**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='140274768819312'>, pickle_protocol=<MagicMock name='mock.DEFAULT_PROTOCOL' id='140274768840184'>*)

Model checkpointer which saves the most recent model.

**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`.

## 7.2 Logging

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

Callback to log metrics to a csv file.

**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.

**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**(*validation_label_letter='v'*)
The Tqdm callback outputs the progress and metrics for training and validation loops to the console using TQDM.

**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_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

**class** torchbearer.callbacks.timer.**TimerCallback**

**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_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`.

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

## 7.3 Tensorboard

**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'*)

The TensorBoard callback is used to write metric graphs to tensorboard. Requires the TensorboardX library for python.

**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**(*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_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*, *range=None*, *scale_each=False*, *pad_value=0*)

The TensorBoardImages callback will write a selection of images from the validation pass to tensorboard using the TensorboardX library and torchvision.utils.make_grid

**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`.

**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.

**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`.

**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`.

## 7.4 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.

**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 that terminates training when the given metric is nan or inf.

**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`.

## 7.5 Gradient Clipping

**class** torchbearer.callbacks.gradient_clipping.**GradientClipping**(*clip_value, params=None*)
GradientClipping callback, uses 'torch.nn.utils.clip_grad_value_'

**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, uses 'torch.nn.utils.clip_grad_norm_'

**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

## 7.6 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`.

## 7.7 Weight Decay

**class** torchbearer.callbacks.weight_decay.**L1WeightDecay** (*rate=0.0005*, *params=None*)
WeightDecay callback which uses an L1 norm

**class** torchbearer.callbacks.weight_decay.**L2WeightDecay** (*rate=0.0005*, *params=None*)
WeightDecay callback which uses an L2 norm

**class** torchbearer.callbacks.weight_decay.**WeightDecay** (*rate=0.0005*, *p=2*, *params=None*)
Callback which adds a weight decay term to the loss for the given parameters.

**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

## 7.8 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`

# CHAPTER 8

## torchbearer.metrics

### 8.1 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'.

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

**Returns** The metric value.

**process_final**(**args*)

Depending on the current mode, return the result of either 'process_final_train' or 'process_final_validate'.

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

**Returns** The final metric value.

**process_final_train**(**args*)

Process the given state and return the final metric value for a training iteration.

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

**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.

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

**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.

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

**Returns** The metric value for a training iteration.

**process_validate**(**args*)
Process the given state and return the metric value for a validation iteration.

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

**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**(*state*)

Process each metric an wrap in a dictionary which maps metric names to values.

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

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

**process_final**(*state*)

Process each metric an wrap in a dictionary which maps metric names to values.

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

**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.

## 8.2 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
{}
>>> 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
...
```

(continues on next page)

(continued from previous page)

```
>>> 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`

## 8.3 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**(*state*)

Return the output of the wrapped function.

**Parameters** **state** (*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**(*state*)

Evaluate the function with the aggregated outputs.

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

**Returns** The result of the function.

**process_final_validate**(*state*)

Evaluate the function with the aggregated outputs.

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

**Returns** The result of the function.

**process_train** (*state*)
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** **state** (*dict*) – The *torchbearer.Model* state.

**Returns** The current running result.

**process_validate** (*state*)
During validation, just concatenate 'y_true' and y_pred'.

**Parameters** **state** (*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.

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

**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.

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

**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.

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

**Returns** The metric value for a training iteration.

**process_validate**(**args*)
Process the given state and return the metric value for a validation iteration.

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

**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.

## 8.4 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**(*data*)
Add the input to the rolling sum.

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

**process_final**(*data*)
Compute and return the mean of all metric values since the last call to reset.

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

**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** (*data*)

Compute values required for the std from the input.

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

**process_final** (*data*)

Compute and return the final standard deviation.

**Parameters** **data** (*torch.Tensor*) – The output of some previous call to *Metric.process_final()*.

**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.

## 8.5 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`**(*state*)
Process the state and update the metric for one iteration.

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

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

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

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

**Returns** None or the value of the metric for this epoch

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

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

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

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

# CHAPTER 9

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## t

# Index

## L



## M



## O

## P

## R



## S



## T

## U



## W