torchbearer Documentation

Release 0.1.3

Ethan Harris and Matthew Painter

Jul 23, 2018

Examples

1	Quickstart Guide	1
2	Training a Variational Auto-Encoder	5
3	Training a GAN	11
4	torchbearer	15
5	torchbearer.callbacks	21
6	torchbearer.metrics	35
7	Indices and tables	47
	Python Module Index	49

CHAPTER 1

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.

1.1 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])

trainset = torchvision.datasets.CIFAR10(root='./data/cifar', train=True,␣
↪download=True,
                                        transform=transforms.Compose([transforms.
↪ToTensor(), normalize]))
traingen = torch.utils.data.DataLoader(trainset, 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(
```

(continues on next page)

(continued from previous page)

```
            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()
```

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.

1.2 Training on Cifar10

```
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=testgen)
```

Running the above produces the following output:

```
Files already downloaded and verified
Files already downloaded and verified
0/10(t): 100%|| 391/391 [00:01<00:00, 211.19it/s, running_acc=0.549, running_loss=1.
↪25, acc=0.469, acc_std=0.499, loss=1.48, loss_std=0.238]
0/10(v): 100%|| 79/79 [00:00<00:00, 265.14it/s, val_acc=0.556, val_acc_std=0.497, val_
↪loss=1.25, val_loss_std=0.0785]
1/10(t): 100%|| 391/391 [00:01<00:00, 209.80it/s, running_acc=0.61, running_loss=1.09,
↪ acc=0.593, acc_std=0.491, loss=1.14, loss_std=0.0968]
1/10(v): 100%|| 79/79 [00:00<00:00, 227.97it/s, val_acc=0.593, val_acc_std=0.491, val_
↪loss=1.14, val_loss_std=0.0865]
2/10(t): 100%|| 391/391 [00:01<00:00, 220.70it/s, running_acc=0.656, running_loss=0.
↪972, acc=0.645, acc_std=0.478, loss=1.01, loss_std=0.0915]
2/10(v): 100%|| 79/79 [00:00<00:00, 218.91it/s, val_acc=0.631, val_acc_std=0.482, val_
↪loss=1.04, val_loss_std=0.0951]
3/10(t): 100%|| 391/391 [00:01<00:00, 208.67it/s, running_acc=0.682, running_loss=0.
↪906, acc=0.675, acc_std=0.468, loss=0.922, loss_std=0.0895]
3/10(v): 100%|| 79/79 [00:00<00:00, 86.95it/s, val_acc=0.657, val_acc_std=0.475, val_
↪loss=0.97, val_loss_std=0.0925]
```

(continues on next page)

(continued from previous page)

```
4/10(t): 100%|| 391/391 [00:01<00:00, 211.22it/s, running_acc=0.693, running_loss=0.
↪866, acc=0.699, acc_std=0.459, loss=0.86, loss_std=0.092]
4/10(v): 100%|| 79/79 [00:00<00:00, 249.74it/s, val_acc=0.662, val_acc_std=0.473, val_
↪loss=0.957, val_loss_std=0.093]
5/10(t): 100%|| 391/391 [00:01<00:00, 205.12it/s, running_acc=0.71, running_loss=0.
↪826, acc=0.713, acc_std=0.452, loss=0.818, loss_std=0.0904]
5/10(v): 100%|| 79/79 [00:00<00:00, 230.12it/s, val_acc=0.661, val_acc_std=0.473, val_
↪loss=0.962, val_loss_std=0.0966]
6/10(t): 100%|| 391/391 [00:01<00:00, 210.87it/s, running_acc=0.714, running_loss=0.
↪81, acc=0.727, acc_std=0.445, loss=0.779, loss_std=0.0904]
6/10(v): 100%|| 79/79 [00:00<00:00, 241.95it/s, val_acc=0.667, val_acc_std=0.471, val_
↪loss=0.952, val_loss_std=0.11]
7/10(t): 100%|| 391/391 [00:01<00:00, 209.94it/s, running_acc=0.727, running_loss=0.
↪791, acc=0.74, acc_std=0.439, loss=0.747, loss_std=0.0911]
7/10(v): 100%|| 79/79 [00:00<00:00, 223.23it/s, val_acc=0.673, val_acc_std=0.469, val_
↪loss=0.938, val_loss_std=0.107]
8/10(t): 100%|| 391/391 [00:01<00:00, 203.16it/s, running_acc=0.747, running_loss=0.
↪736, acc=0.752, acc_std=0.432, loss=0.716, loss_std=0.0899]
8/10(v): 100%|| 79/79 [00:00<00:00, 221.55it/s, val_acc=0.679, val_acc_std=0.467, val_
↪loss=0.923, val_loss_std=0.113]
9/10(t): 100%|| 391/391 [00:01<00:00, 213.23it/s, running_acc=0.756, running_loss=0.
↪701, acc=0.759, acc_std=0.428, loss=0.695, loss_std=0.0915]
9/10(v): 100%|| 79/79 [00:00<00:00, 245.33it/s, val_acc=0.676, val_acc_std=0.468, val_
↪loss=0.951, val_loss_std=0.111]
```

1.3 Source Code

The source code for the example is given below:

`Download Python source code: quickstart.py`

CHAPTER 2

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:

2.1 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
```

2.2 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)
```

2.3 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. 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.

2.3.1 PyTorch method

The loss function from the PyTorch example is:

```
def loss_function(recon_x, x, mu, logvar):
    BCE = F.binary_cross_entropy(recon_x, x.view(-1, 784), size_average=False)

    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())

    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
```

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

2.3.2 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=[AddKLDLoss(), SaveReconstruction()], 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 loss can then be separated into a standard reconstruction loss and a separate KL-divergence loss using intermediate tensor values.

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

Since loss functions cannot access state, we utilise a simple callback to complete the loss calculation.

```
class AddKLDLoss(Callback):
    def on_criterion(self, state):
        super().on_criterion(state)
        KLD = self.KLD_Loss(state['mu'], state['logvar'])
        state[torchbearer.LOSS] = state[torchbearer.LOSS] + KLD

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

2.4 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.

```
class SaveReconstruction(Callback):
    def __init__(self, num_images=8, folder='results/'):
        super().__init__()
        self.num_images = num_images
        self.folder = folder

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

2.5 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
```

The visualised results after ten epochs then look like this:

2.6 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 3

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.

3.1 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)
```

3.2 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 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.

3.3 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.

3.4 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]
```

3.5 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])
```

3.6 Visualising

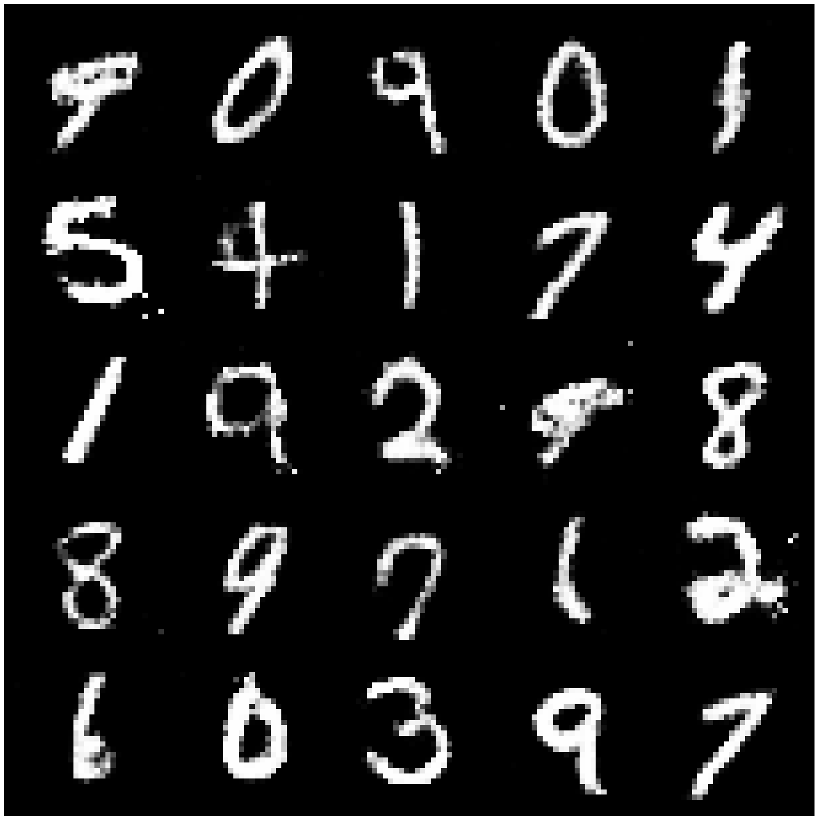
We borrow the image saving method from PyTorch_GAN and put it in a call back to save on training step - again using decorators.

```
@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.

3.7 Source Code

The source code for the example is given below:



Download Python source code: gan.py

CHAPTER 4

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=0.0*, *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*)

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 5

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_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_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`.

5.1 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='140290296390600'>*, *pickle_protocol=<MagicMock name='mock.DEFAULT_PROTOCOL' id='140290296415568'>*)

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='140290296436664'>*, *pickle_protocol=<MagicMock name='mock.DEFAULT_PROTOCOL' id='140290296453440'>*)

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='140290296352608'>*, *pickle_protocol=<MagicMock name='mock.DEFAULT_PROTOCOL' id='140290296373544'>*)

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.

5.2 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

5.3 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`.

5.4 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.

5.5 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

5.6 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`.

5.7 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

5.8 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_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 6

torchbearer.metrics

6.1 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.

6.2 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`

6.3 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.

6.4 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.

6.5 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

class torchbearer.metrics.roc_auc_score.**RocAucScore** (*one_hot_labels=True*, *one_hot_offset=0*, *one_hot_classes=10*)

Area Under ROC curve metric.

Note: Requires `sklearn.metrics`.

Parameters

- **one_hot_labels** (*bool*) – If True, convert the labels to a one hot encoding. Required if they are not already.
- **one_hot_offset** (*int*) – Subtracted from class labels, use if not already zero based.
- **one_hot_classes** (*int*) – Number of classes for the one hot encoding.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

torchbearer, 15
torchbearer.callbacks, 21
torchbearer.callbacks.callbacks, 21
torchbearer.callbacks.checkpointers, 24
torchbearer.callbacks.csv_logger, 25
torchbearer.callbacks.decorators, 31
torchbearer.callbacks.early_stopping, 28
torchbearer.callbacks.gradient_clipping, 29
torchbearer.callbacks.printer, 25
torchbearer.callbacks.tensor_board, 26
torchbearer.callbacks.terminate_on_nan, 28
torchbearer.callbacks.torch_scheduler, 29
torchbearer.callbacks.weight_decay, 31
torchbearer.cv_utils, 18
torchbearer.metrics, 35
torchbearer.metrics.aggregators, 43
torchbearer.metrics.decorators, 38
torchbearer.metrics.metrics, 35
torchbearer.metrics.primitives, 44
torchbearer.metrics.roc_auc_score, 45
torchbearer.metrics.wrappers, 41
torchbearer.state, 18
torchbearer.torchbearer, 15

Index

A

add_child() (torchbearer.metrics.metrics.MetricTree method), 37
add_to_loss() (in module torchbearer.callbacks.decorators), 31
AdvancedMetric (class in torchbearer.metrics.metrics), 35

B

BatchLambda (class in torchbearer.metrics.wrappers), 41
Best (class in torchbearer.callbacks.checkpointers), 24
build() (torchbearer.metrics.metrics.MetricFactory method), 37

C

Callback (class in torchbearer.callbacks.callbacks), 21
CallbackList (class in torchbearer.callbacks.callbacks), 22
CategoricalAccuracy (class in torchbearer.metrics.primitives), 44
ConsolePrinter (class in torchbearer.callbacks.printer), 25
CosineAnnealingLR (class in torchbearer.callbacks.torch_scheduler), 29
cpu() (torchbearer.torchbearer.Model method), 15
CSVLogger (class in torchbearer.callbacks.csv_logger), 25
cuda() (torchbearer.torchbearer.Model method), 15

D

default_for_key() (in module torchbearer.metrics.decorators), 38

E

EarlyStopping (class in torchbearer.callbacks.early_stopping), 28
Epoch (class in torchbearer.metrics.primitives), 45
EpochLambda (class in torchbearer.metrics.wrappers), 41
eval() (torchbearer.metrics.metrics.AdvancedMetric method), 35
eval() (torchbearer.metrics.metrics.Metric method), 36
eval() (torchbearer.metrics.metrics.MetricList method), 37
eval() (torchbearer.metrics.metrics.MetricTree method), 37
eval() (torchbearer.metrics.wrappers.ToDict method), 42
eval() (torchbearer.torchbearer.Model method), 15
evaluate() (torchbearer.torchbearer.Model method), 15
evaluate_generator() (torchbearer.torchbearer.Model method), 16
ExponentialLR (class in torchbearer.callbacks.torch_scheduler), 29

F

fit() (torchbearer.torchbearer.Model method), 16
fit_generator() (torchbearer.torchbearer.Model method), 17

G

get_train_valid_sets() (in module torchbearer.cv_utils), 18
GradientClipping (class in torchbearer.callbacks.gradient_clipping), 29
GradientNormClipping (class in torchbearer.callbacks.gradient_clipping), 29

I

Interval (class in torchbearer.callbacks.checkpointers), 24

L

L1WeightDecay (class in torchbearer.callbacks.weight_decay), 31
L2WeightDecay (class in torchbearer.callbacks.weight_decay), 31
lambda_metric() (in module torchbearer.metrics.decorators), 38
LambdaLR (class in torchbearer.callbacks.torch_scheduler), 29
load_state_dict() (torchbearer.torchbearer.Model method), 17

Loss (class in torchbearer.metrics.primitives), 45

M

Mean (class in torchbearer.metrics.aggregators), 43
mean() (in module torchbearer.metrics.decorators), 38
Metric (class in torchbearer.metrics.metrics), 36
MetricFactory (class in torchbearer.metrics.metrics), 36
MetricList (class in torchbearer.metrics.metrics), 37
MetricTree (class in torchbearer.metrics.metrics), 37
Model (class in torchbearer.torchbearer), 15
ModelCheckpoint() (in module torchbearer.callbacks.checkpointers), 24
MostRecent (class in torchbearer.callbacks.checkpointers), 25
MultiStepLR (class in torchbearer.callbacks.torch_scheduler), 30

O

on_backward() (in module torchbearer.callbacks.decorators), 31
on_backward() (torchbearer.callbacks.callbacks.Callback method), 21
on_backward() (torchbearer.callbacks.callbacks.CallbackList method), 22
on_backward() (torchbearer.callbacks.gradient_clipping.GradientClipping method), 29
on_backward() (torchbearer.callbacks.gradient_clipping.GradientNormClipping method), 29
on_criterion() (in module torchbearer.callbacks.decorators), 31
on_criterion() (torchbearer.callbacks.callbacks.Callback method), 21
on_criterion() (torchbearer.callbacks.callbacks.CallbackList method), 22
on_criterion() (torchbearer.callbacks.weight_decay.WeightDecay method), 31
on_end() (in module torchbearer.callbacks.decorators), 31
on_end() (torchbearer.callbacks.callbacks.Callback method), 21
on_end() (torchbearer.callbacks.callbacks.CallbackList method), 22
on_end() (torchbearer.callbacks.csv_logger.CSVLogger method), 25
on_end() (torchbearer.callbacks.early_stopping.EarlyStopping method), 28
on_end() (torchbearer.callbacks.tensor_board.TensorBoard method), 26
on_end() (torchbearer.callbacks.tensor_board.TensorBoardImages method), 27
on_end() (torchbearer.callbacks.tensor_board.TensorBoardProjector method), 28
on_end_epoch() (in module torchbearer.callbacks.decorators), 32
on_end_epoch() (torchbearer.callbacks.callbacks.Callback method), 21
on_end_epoch() (torchbearer.callbacks.callbacks.CallbackList method), 23
on_end_epoch() (torchbearer.callbacks.checkpointers.Best method), 24
on_end_epoch() (torchbearer.callbacks.checkpointers.Interval method), 24
on_end_epoch() (torchbearer.callbacks.checkpointers.MostRecent method), 25
on_end_epoch() (torchbearer.callbacks.csv_logger.CSVLogger method), 25
on_end_epoch() (torchbearer.callbacks.early_stopping.EarlyStopping method), 28
on_end_epoch() (torchbearer.callbacks.tensor_board.TensorBoard method), 26
on_end_epoch() (torchbearer.callbacks.tensor_board.TensorBoardImages method), 27
on_end_epoch() (torchbearer.callbacks.tensor_board.TensorBoardProjector method), 28
on_end_epoch() (torchbearer.callbacks.terminate_on_nan.TerminateOnNaN method), 28
on_end_epoch() (torchbearer.callbacks.torch_scheduler.TorchScheduler method), 30
on_end_training() (in module torchbearer.callbacks.decorators), 32
on_end_training() (torchbearer.callbacks.callbacks.Callback method), 21
on_end_training() (torchbearer.callbacks.callbacks.CallbackList method), 23
on_end_training() (torchbearer.callbacks.printer.ConsolePrinter method), 25
on_end_training() (torchbearer.callbacks.printer.Tqdm method), 26
on_end_validation() (in module torchbearer.callbacks.decorators), 32
on_end_validation() (torchbearer.callbacks.callbacks.Callback method), 21

on_end_validation() (torchbearer.callbacks.callbacks.CallbackList method), 23
on_end_validation() (torchbearer.callbacks.printer.ConsolePrinter method), 25
on_end_validation() (torchbearer.callbacks.printer.Tqdm method), 26
on_forward() (in module torchbearer.callbacks.decorators), 32
on_forward() (torchbearer.callbacks.callbacks.Callback method), 21
on_forward() (torchbearer.callbacks.callbacks.CallbackList method), 23
on_forward_validation() (in module torchbearer.callbacks.decorators), 32
on_forward_validation() (torchbearer.callbacks.callbacks.Callback method), 22
on_forward_validation() (torchbearer.callbacks.callbacks.CallbackList method), 23
on_sample() (in module torchbearer.callbacks.decorators), 32
on_sample() (torchbearer.callbacks.callbacks.Callback method), 22
on_sample() (torchbearer.callbacks.callbacks.CallbackList method), 23
on_sample() (torchbearer.callbacks.tensor_board.TensorBoard method), 27
on_sample() (torchbearer.callbacks.torch_scheduler.TorchScheduler method), 30
on_sample_validation() (in module torchbearer.callbacks.decorators), 32
on_sample_validation() (torchbearer.callbacks.callbacks.Callback method), 22
on_sample_validation() (torchbearer.callbacks.callbacks.CallbackList method), 23
on_start() (in module torchbearer.callbacks.decorators), 33
on_start() (torchbearer.callbacks.callbacks.Callback method), 22
on_start() (torchbearer.callbacks.callbacks.CallbackList method), 23
on_start() (torchbearer.callbacks.checkpointers.Best method), 24
on_start() (torchbearer.callbacks.early_stopping.EarlyStopping method), 28
on_start() (torchbearer.callbacks.gradient_clipping.GradientClipping method), 29
on_start() (torchbearer.callbacks.gradient_clipping.GradientNormClipping method), 29
on_start() (torchbearer.callbacks.tensor_board.TensorBoard method), 27
on_start() (torchbearer.callbacks.tensor_board.TensorBoardImages method), 27
on_start() (torchbearer.callbacks.tensor_board.TensorBoardProjector method), 28
on_start() (torchbearer.callbacks.torch_scheduler.TorchScheduler method), 30
on_start() (torchbearer.callbacks.weight_decay.WeightDecay method), 31
on_start_epoch() (in module torchbearer.callbacks.decorators), 33
on_start_epoch() (torchbearer.callbacks.callbacks.Callback method), 22
on_start_epoch() (torchbearer.callbacks.callbacks.CallbackList method), 23
on_start_epoch() (torchbearer.callbacks.tensor_board.TensorBoard method), 27
on_start_training() (in module torchbearer.callbacks.decorators), 33
on_start_training() (torchbearer.callbacks.callbacks.Callback method), 22
on_start_training() (torchbearer.callbacks.callbacks.CallbackList method), 23
on_start_training() (torchbearer.callbacks.printer.Tqdm method), 26
on_start_training() (torchbearer.callbacks.torch_scheduler.TorchScheduler method), 30
on_start_validation() (in module torchbearer.callbacks.decorators), 33
on_start_validation() (torchbearer.callbacks.callbacks.Callback method), 22
on_start_validation() (torchbearer.callbacks.callbacks.CallbackList method), 23
on_start_validation() (torchbearer.callbacks.printer.Tqdm method), 26
on_step_training() (in module torchbearer.callbacks.decorators), 33
on_step_training() (torchbearer.callbacks.callbacks.Callback method), 22
on_step_training() (torchbearer.callbacks.callbacks.CallbackList method), 23
on_step_training() (torchbearer.callbacks.csv_logger.CSVLogger

method), 25
on_step_training() (torchbearer.callbacks.printer.ConsolePrinter method), 25
on_step_training() (torchbearer.callbacks.printer.Tqdm method), 26
on_step_training() (torchbearer.callbacks.tensor_board.TensorBoard method), 27
on_step_training() (torchbearer.callbacks.terminate_on_nan.TerminateOnNaN method), 29
on_step_training() (torchbearer.callbacks.torch_scheduler.TorchScheduler method), 30
on_step_validation() (in module torchbearer.callbacks.decorators), 33
on_step_validation() (torchbearer.callbacks.callbacks.Callback method), 22
on_step_validation() (torchbearer.callbacks.callbacks.CallbackList method), 23
on_step_validation() (torchbearer.callbacks.printer.ConsolePrinter method), 26
on_step_validation() (torchbearer.callbacks.printer.Tqdm method), 26
on_step_validation() (torchbearer.callbacks.tensor_board.TensorBoard method), 27
on_step_validation() (torchbearer.callbacks.tensor_board.TensorBoardImages method), 27
on_step_validation() (torchbearer.callbacks.tensor_board.TensorBoardProjector method), 28
on_step_validation() (torchbearer.callbacks.terminate_on_nan.TerminateOnNaN method), 29

P

predict() (torchbearer.torchbearer.Model method), 17
predict_generator() (torchbearer.torchbearer.Model method), 18
process() (torchbearer.metrics.aggregators.Mean method), 43
process() (torchbearer.metrics.aggregators.Std method), 44
process() (torchbearer.metrics.metrics.AdvancedMetric method), 35
process() (torchbearer.metrics.metrics.Metric method), 36
process() (torchbearer.metrics.metrics.MetricList method), 37
process() (torchbearer.metrics.metrics.MetricTree method), 37
process() (torchbearer.metrics.primitives.Epoch method), 45
process() (torchbearer.metrics.primitives.Loss method), 45
process() (torchbearer.metrics.wrappers.BatchLambda method), 41
process_final() (torchbearer.metrics.aggregators.Mean method), 43
process_final() (torchbearer.metrics.aggregators.Std method), 44
process_final() (torchbearer.metrics.metrics.AdvancedMetric method), 35
process_final() (torchbearer.metrics.metrics.Metric method), 36
process_final() (torchbearer.metrics.metrics.MetricList method), 37
process_final() (torchbearer.metrics.metrics.MetricTree method), 37
process_final() (torchbearer.metrics.primitives.Epoch method), 45
process_final_train() (torchbearer.metrics.metrics.AdvancedMetric method), 35
process_final_train() (torchbearer.metrics.wrappers.EpochLambda method), 41
process_final_train() (torchbearer.metrics.wrappers.ToDict method), 42
process_final_validate() (torchbearer.metrics.metrics.AdvancedMetric method), 36
process_final_validate() (torchbearer.metrics.wrappers.EpochLambda method), 41
process_final_validate() (torchbearer.metrics.wrappers.ToDict method), 42
process_train() (torchbearer.metrics.aggregators.RunningMetric method), 44
process_train() (torchbearer.metrics.metrics.AdvancedMetric method), 36
process_train() (torchbearer.metrics.wrappers.EpochLambda method), 41
process_train() (torchbearer.metrics.wrappers.ToDict method), 42
process_validate() (torchbearer.metrics.metrics.AdvancedMetric method), 36
process_validate() (torchbearer.metrics.wrappers.EpochLambda method), 42

process_validate() (torchbearer.metrics.wrappers.ToDict method), 42

R

ReduceLROnPlateau (class in torchbearer.callbacks.torch_scheduler), 30
reset() (torchbearer.metrics.aggregators.Mean method), 43
reset() (torchbearer.metrics.aggregators.RunningMetric method), 44
reset() (torchbearer.metrics.aggregators.Std method), 44
reset() (torchbearer.metrics.metrics.Metric method), 36
reset() (torchbearer.metrics.metrics.MetricList method), 37
reset() (torchbearer.metrics.metrics.MetricTree method), 38
reset() (torchbearer.metrics.wrappers.EpochLambda method), 42
reset() (torchbearer.metrics.wrappers.ToDict method), 43
RocAucScore (class in torchbearer.metrics.roc_auc_score), 45
running_mean() (in module torchbearer.metrics.decorators), 39
RunningMean (class in torchbearer.metrics.aggregators), 43
RunningMetric (class in torchbearer.metrics.aggregators), 43

S

state_dict() (torchbearer.torchbearer.Model method), 18
state_key() (in module torchbearer.state), 18
Std (class in torchbearer.metrics.aggregators), 44
std() (in module torchbearer.metrics.decorators), 40
StepLR (class in torchbearer.callbacks.torch_scheduler), 30

T

TensorBoard (class in torchbearer.callbacks.tensor_board), 26
TensorBoardImages (class in torchbearer.callbacks.tensor_board), 27
TensorBoardProjector (class in torchbearer.callbacks.tensor_board), 27
TerminateOnNaN (class in torchbearer.callbacks.terminate_on_nan), 28
to() (torchbearer.torchbearer.Model method), 18
to_dict() (in module torchbearer.metrics.decorators), 40
ToDict (class in torchbearer.metrics.wrappers), 42
torchbearer (module), 15
torchbearer.callbacks (module), 21
torchbearer.callbacks.callbacks (module), 21
torchbearer.callbacks.checkpointers (module), 24
torchbearer.callbacks.csv_logger (module), 25
torchbearer.callbacks.decorators (module), 31
torchbearer.callbacks.early_stopping (module), 28
torchbearer.callbacks.gradient_clipping (module), 29
torchbearer.callbacks.printer (module), 25
torchbearer.callbacks.tensor_board (module), 26
torchbearer.callbacks.terminate_on_nan (module), 28
torchbearer.callbacks.torch_scheduler (module), 29
torchbearer.callbacks.weight_decay (module), 31
torchbearer.cv_utils (module), 18
torchbearer.metrics (module), 35
torchbearer.metrics.aggregators (module), 43
torchbearer.metrics.decorators (module), 38
torchbearer.metrics.metrics (module), 35
torchbearer.metrics.primitives (module), 44
torchbearer.metrics.roc_auc_score (module), 45
torchbearer.metrics.wrappers (module), 41
torchbearer.state (module), 18
torchbearer.torchbearer (module), 15
TorchScheduler (class in torchbearer.callbacks.torch_scheduler), 30
Tqdm (class in torchbearer.callbacks.printer), 26
train() (torchbearer.metrics.metrics.AdvancedMetric method), 36
train() (torchbearer.metrics.metrics.Metric method), 36
train() (torchbearer.metrics.metrics.MetricList method), 37
train() (torchbearer.metrics.metrics.MetricTree method), 38
train() (torchbearer.metrics.wrappers.ToDict method), 43
train() (torchbearer.torchbearer.Model method), 18
train_valid_splitter() (in module torchbearer.cv_utils), 18

W

WeightDecay (class in torchbearer.callbacks.weight_decay), 31