torchbearer Documentation

Release 0.3.2

Torchbearer Contributors

Jun 05, 2019

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:

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

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

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

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

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

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

Serializing a Trial

This guide will explain the two different ways to how to save and reload your results from a Trial.

2.1 Setting up a Mock Example

Let's assume we have a basic binary classification task where we have 100-dimensional samples as input and a binary label as output. Let's also assume that we would like to solve this problem with a 2-layer neural network. Finally, we also want to keep track of the sum of hidden outputs for some arbitrary reason. Therefore we use the state functionality of Torchbearer.

We create a state key for the mock sum we wanted to track using state.

```
MOCK = torchbearer.state_key('mock')
```

Here is our basic 2-layer neural network.

```
class BasicModel(nn.Module):
    def __init__(self):
        super(BasicModel, self).__init__()
        self.linear1 = nn.Linear(100, 25)
        self.linear2 = nn.Linear(25, 1)

    def forward(self, x, state):
        x = self.linear1(x)
        # The following step is here to showcase a useless but simple of example a␣
↪forward method that uses state
        state[MOCK] = torch.sum(x)
        x = self.linear2(x)
        return torch.sigmoid(x)
```

We create some random training dataset and put them in a DataLoader.

```
n_sample = 100
X = torch.rand(n_sample, 100)
y = torch.randint(0, 2, [n_sample, 1]).float()
traingen = DataLoader(TensorDataset(X, y))
```

Let's say we would like to save the model every time we get a better training loss. Torchbearer's Best checkpoint callback is perfect for this job. We then run the model for 3 epochs.

```
model = BasicModel()
# Create a checkpointer that track val_loss and saves a model.pt whenever we get a␣
↪better loss
checkpointer = torchbearer.callbacks.checkpointers.Best(filepath='model.pt', monitor=
↪'loss')
optimizer = optim.SGD(filter(lambda p: p.requires_grad, model.parameters()), lr=0.001)
torchbearer_trial = Trial(model, optimizer=optimizer, criterion=F.binary_cross_
↪entropy, metrics=['loss'],
                          callbacks=[checkpointer])
torchbearer_trial.with_train_generator(traingen)
torchbearer_trial.run(epochs=3)
```

2.2 Reloading the Trial for More Epochs

Given we recreate the exact same Trial structure, we can easily resume our run from the last checkpoint. The following code block shows how it's done. Remember here that the `epochs` parameter we pass to Trial acts cumulative. In other words, the following run will complement the entire training to a total of 6 epochs.

```
state_dict = torch.load('model.pt')
model = BasicModel()
trial_reloaded = Trial(model, optimizer=optimizer, criterion=F.binary_cross_entropy,␣
↪metrics=['loss'],
                       callbacks=[checkpointer])
trial_reloaded.load_state_dict(state_dict)
trial_reloaded.with_train_generator(traingen)
trial_reloaded.run(epochs=6)
```

2.3 Trying to Reload to a PyTorch Module

We try to load the `state_dict` to a regular PyTorch Module, as described in PyTorch's own documentation here:

```
model = BasicModel()
try:
    model.load_state_dict(state_dict)
except AttributeError as e:
    print("\n")
    print(e)
```

We will get the following error:

```
'StateKey' object has no attribute 'startswith'
```

The reason is that the `state_dict` has Trial related attributes that are unknown to a native PyTorch model. This is why we have the `save_model_params_only` option for our checkpointers. We try again with that option

```
model = BasicModel()
checkpointer = torchbearer.callbacks.checkpointers.Best(filepath='model.pt', monitor=
↪'loss', save_model_params_only=True)
optimizer = optim.SGD(filter(lambda p: p.requires_grad, model.parameters()), lr=0.001)
torchbearer_trial = Trial(model, optimizer=optimizer, criterion=F.binary_cross_
↪entropy, metrics=['loss'],
                          callbacks=[checkpointer])
torchbearer_trial.with_train_generator(traingen)
torchbearer_trial.run(epochs=3)

# Try once again to load the module, forward another random sample for testing
state_dict = torch.load('model.pt')
model = BasicModel()
model.load_state_dict(state_dict)
```

No errors this time, but we still have to test. Here is a test sample and we run it through the model.

```
X_test = torch.rand(5, 100)
try:
    model(X_test)
except TypeError as e:
    print("\n")
    print(e)
```

```
forward() missing 1 required positional argument: 'state'
```

Now we get a different error, stating that we should also be passing `state` as an argument to module's forward. This should not be a surprise as we defined `state` parameter in the forward method of `BasicModule` as a required argument.

2.4 Robust Signature for Module

We define the model with a better signature this time, so it gracefully handles the problem above.

```
class BetterSignatureModel(nn.Module):
    def __init__(self):
        super(BetterSignatureModel, self).__init__()
        self.linear1 = nn.Linear(100, 25)
        self.linear2 = nn.Linear(25, 1)

    def forward(self, x, **state):
        x = self.linear1(x)
        # Using kwargs instead of state is safer from a serialization perspective
        if state is not None:
            state = state
            state[MOCK] = torch.sum(x)
        x = self.linear2(x)
        return torch.sigmoid(x)
```

Finally, we wrap it up once again to test the new definition of the model.

```
model = BetterSignatureModel()
checkpointer = torchbearer.callbacks.checkpointers.Best(filepath='model.pt', monitor=
↪'loss', save_model_params_only=True)
optimizer = optim.SGD(filter(lambda p: p.requires_grad, model.parameters()), lr=0.001)
```

(continues on next page)

(continued from previous page)

```
torchbearer_trial = Trial(model, optimizer=optimizer, criterion=F.binary_cross_
↪entropy, metrics=['loss'],
                          callbacks=[checkpointer])
torchbearer_trial.with_train_generator(traingen)
torchbearer_trial.run(epochs=3)

# This time, the forward function should work without the need for a state argument
state_dict = torch.load('model.pt')
model = BetterSignatureModel()
model.load_state_dict(state_dict)
X_test = torch.rand(5, 100)
model(X_test)
```

2.5 Source Code

The source code for the example are given below:

Download Python source code: serialization.py

CHAPTER 3

Using the Tensorboard Callback

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

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

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

3.1 Setup

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

```
BATCH_SIZE = 128

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

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

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


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

(continues on next page)

(continued from previous page)

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

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

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

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


model = SimpleModel()

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

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

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

3.2 Logging the Model Graph

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

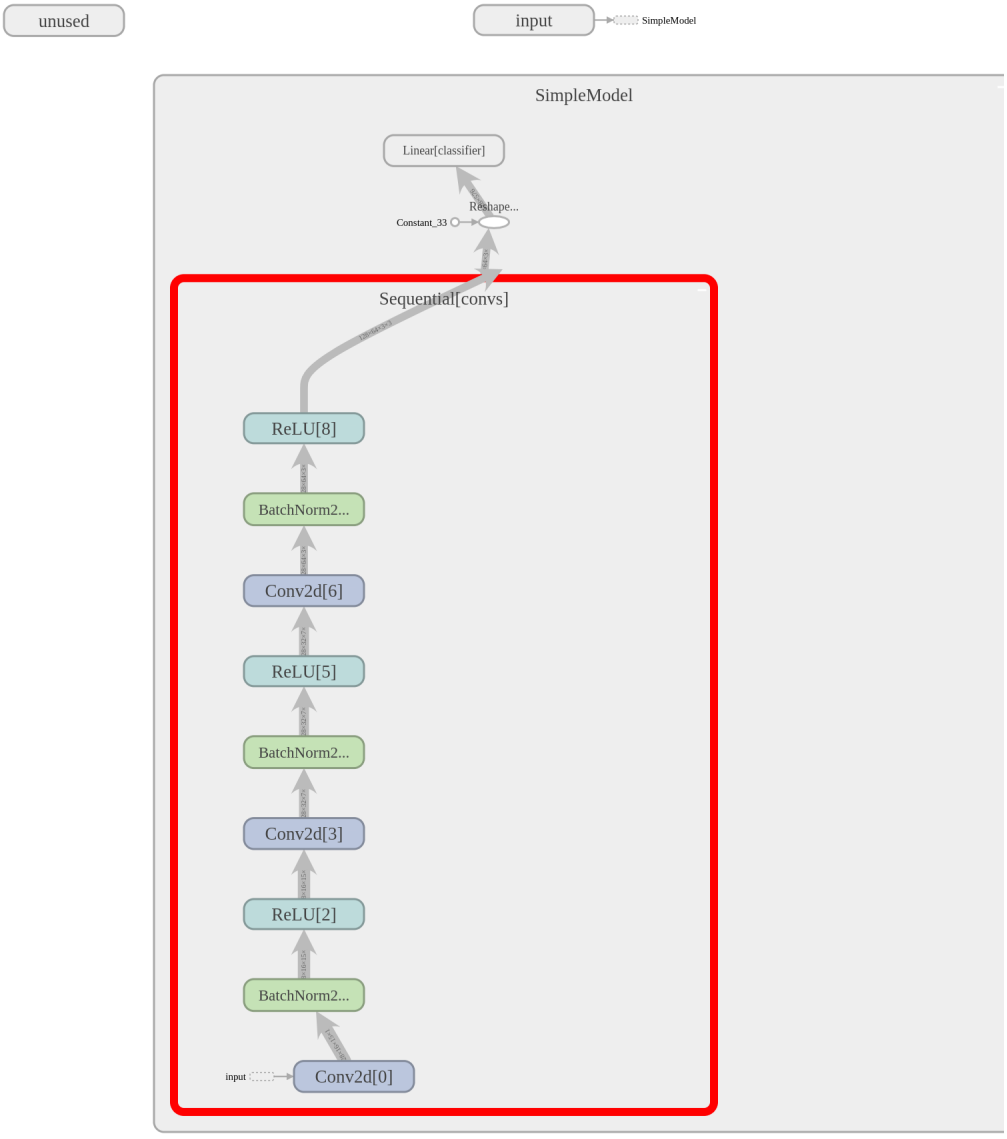
```
from torchbearer import Trial
from torchbearer.callbacks import TensorBoard
```

(continues on next page)

(continued from previous page)

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

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



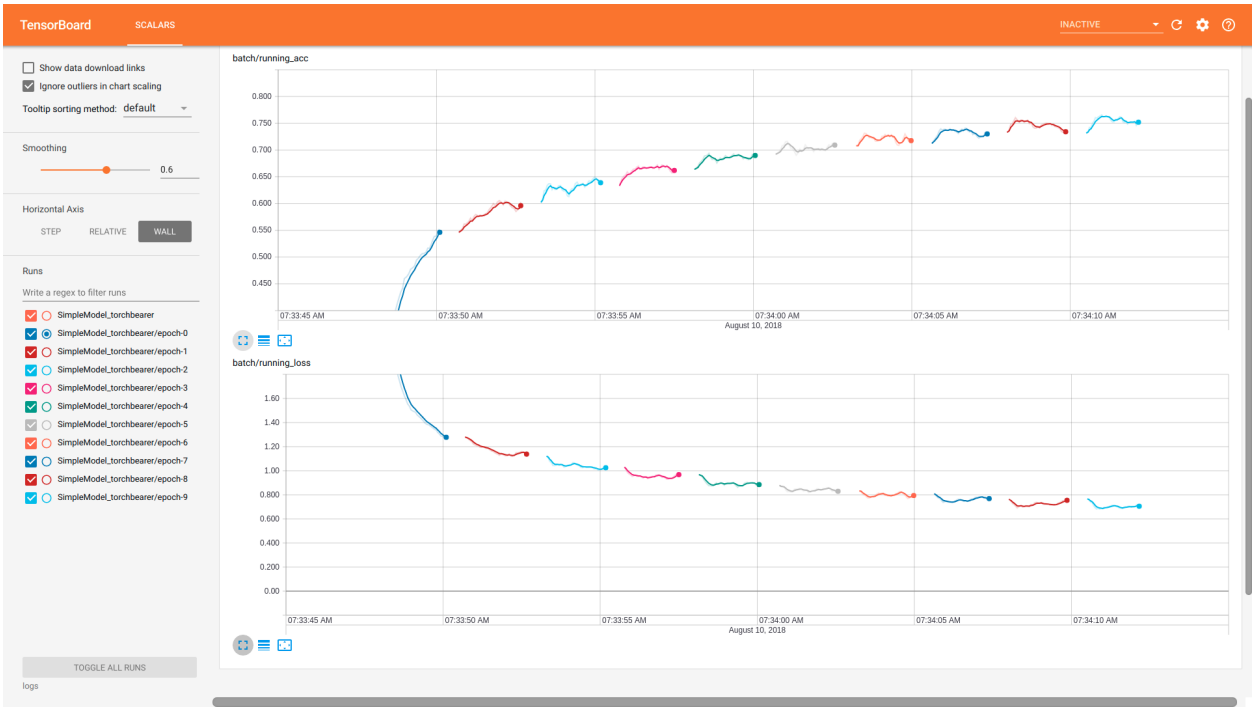
The dynamic graph construction does introduce some weirdness, however, this is about as good as model graphs typically get.

3.3 Logging Batch Metrics

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

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

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

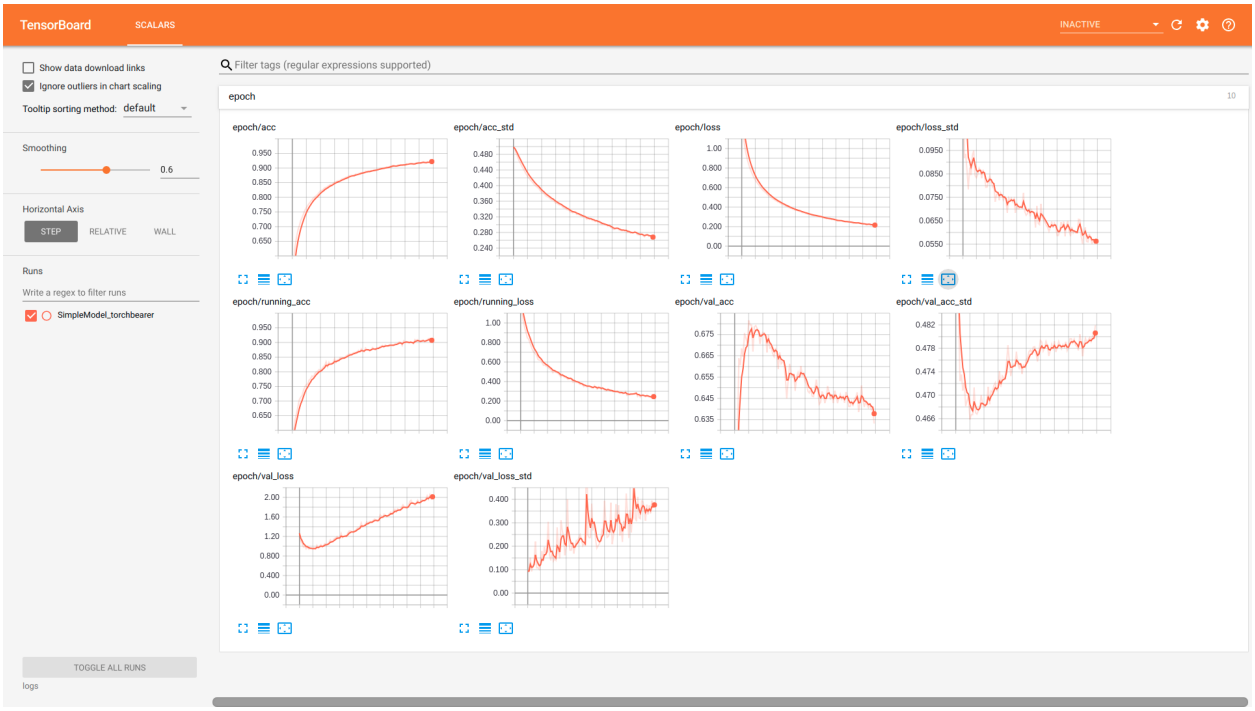


3.4 Logging Epoch Metrics

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

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

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



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

3.5 Source Code

The source code for these examples is given below:

`Download Python source code: tensorboard.py`

CHAPTER 4

Logging to Visdom

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

4.1 Model Setup

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

```
BATCH_SIZE = 128

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

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

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


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

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

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

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


model = SimpleModel()

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

4.2 Logging Epoch and Batch Metrics

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

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

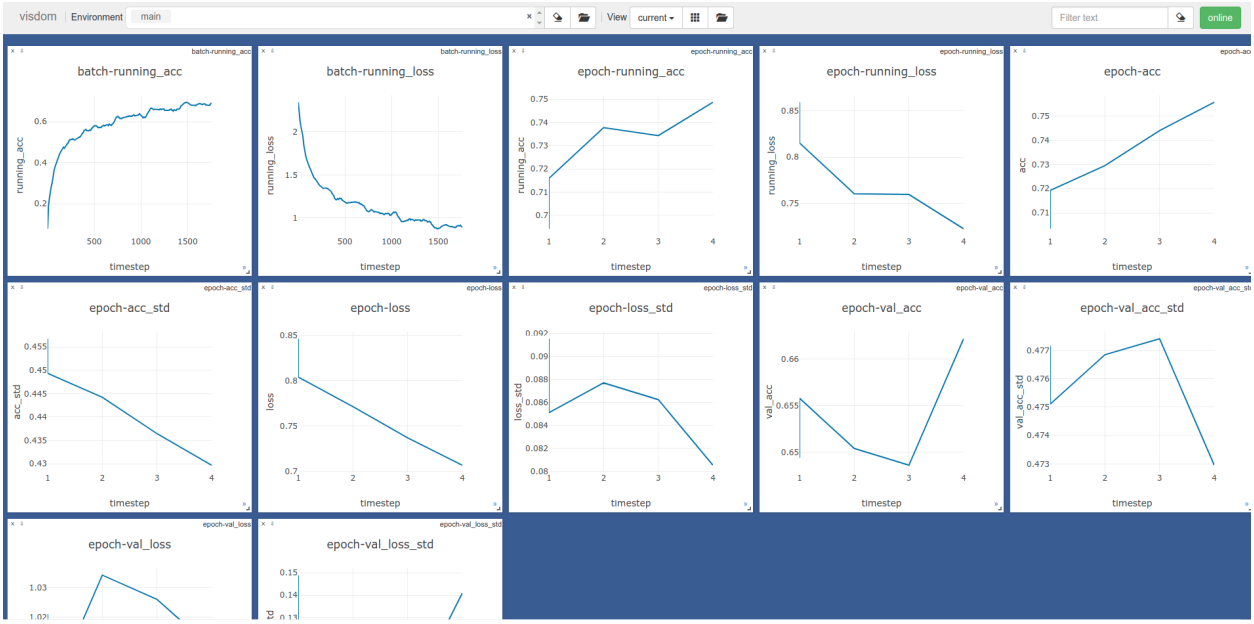
If your visdom server is running then you should see something similar to the figure below:

4.3 Visdom Client Parameters

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

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

(continues on next page)



(continued from previous page)

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

We first import the tensorboard file.

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

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

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

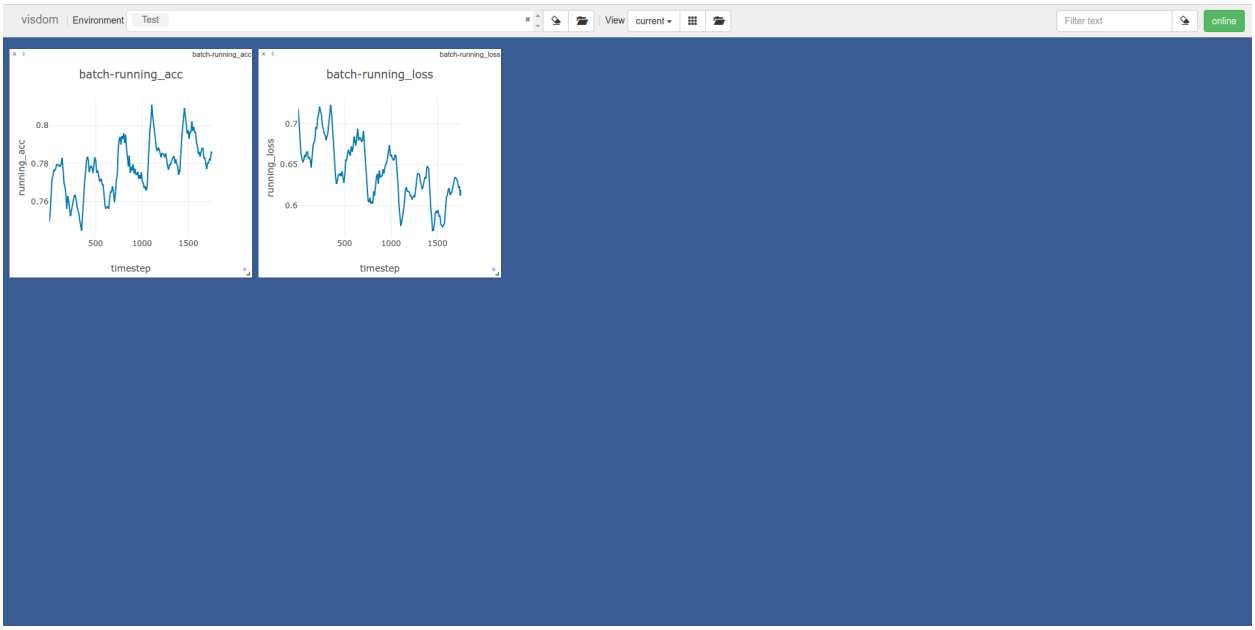
Running another fit call, we can see we are now logging to the "Test" environment.

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

4.4 Source Code

The source code for this example is given below:

Download Python source code: visdom.py


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

CHAPTER 5

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.

5.1 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.

5.2 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()

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

torchbearer_trial = Trial(model, optimizer, loss, metrics=['acc', 'loss'],␣
↪callbacks=[TensorBoard(write_batch_metrics=True)]).to('cuda')
torchbearer_trial.with_generators(train_generator=traingen, val_generator=valgen,␣
↪test_generator=testgen)
torchbearer_trial.run(epochs=10)
torchbearer_trial.evaluate(data_key=torchbearer.TEST_DATA)
```

Running the above produces the following output:

```
Files already downloaded and verified
Files already downloaded and verified
0/10(t): 100%|| 352/352 [00:02<00:00, 163.98it/s, acc=0.4339, loss=1.5776, running_
↪acc=0.5202, running_loss=1.3494]
```

(continues on next page)

(continued from previous page)

```
0/10(v): 100%|| 40/40 [00:00<00:00, 365.42it/s, val_acc=0.5266, val_loss=1.3208]
1/10(t): 100%|| 352/352 [00:02<00:00, 171.36it/s, acc=0.5636, loss=1.2176, running_
↪acc=0.5922, running_loss=1.1418]
1/10(v): 100%|| 40/40 [00:00<00:00, 292.15it/s, val_acc=0.5888, val_loss=1.1657]
2/10(t): 100%|| 352/352 [00:02<00:00, 124.04it/s, acc=0.6226, loss=1.0671, running_
↪acc=0.6222, running_loss=1.0566]
2/10(v): 100%|| 40/40 [00:00<00:00, 359.21it/s, val_acc=0.626, val_loss=1.0555]
3/10(t): 100%|| 352/352 [00:02<00:00, 151.69it/s, acc=0.6587, loss=0.972, running_
↪acc=0.6634, running_loss=0.9589]
3/10(v): 100%|| 40/40 [00:00<00:00, 222.62it/s, val_acc=0.6414, val_loss=1.0064]
4/10(t): 100%|| 352/352 [00:02<00:00, 131.49it/s, acc=0.6829, loss=0.9061, running_
↪acc=0.6764, running_loss=0.918]
4/10(v): 100%|| 40/40 [00:00<00:00, 346.88it/s, val_acc=0.6636, val_loss=0.9449]
5/10(t): 100%|| 352/352 [00:02<00:00, 164.28it/s, acc=0.6988, loss=0.8563, running_
↪acc=0.6919, running_loss=0.858]
5/10(v): 100%|| 40/40 [00:00<00:00, 244.97it/s, val_acc=0.663, val_loss=0.9404]
6/10(t): 100%|| 352/352 [00:02<00:00, 149.52it/s, acc=0.7169, loss=0.8131, running_
↪acc=0.7095, running_loss=0.8421]
6/10(v): 100%|| 40/40 [00:00<00:00, 329.26it/s, val_acc=0.6704, val_loss=0.9209]
7/10(t): 100%|| 352/352 [00:02<00:00, 160.60it/s, acc=0.7302, loss=0.7756, running_
↪acc=0.738, running_loss=0.767]
7/10(v): 100%|| 40/40 [00:00<00:00, 349.86it/s, val_acc=0.6716, val_loss=0.9313]
8/10(t): 100%|| 352/352 [00:02<00:00, 155.08it/s, acc=0.7412, loss=0.7444, running_
↪acc=0.7347, running_loss=0.7547]
8/10(v): 100%|| 40/40 [00:00<00:00, 350.05it/s, val_acc=0.673, val_loss=0.9324]
9/10(t): 100%|| 352/352 [00:02<00:00, 165.28it/s, acc=0.7515, loss=0.715, running_
↪acc=0.7352, running_loss=0.7492]
9/10(v): 100%|| 40/40 [00:00<00:00, 310.76it/s, val_acc=0.6792, val_loss=0.9743]
0/1(e): 100%|| 79/79 [00:00<00:00, 233.06it/s, test_acc=0.6673, test_loss=0.9741]
```

5.3 Source Code

The source code for the example is given below:

Download Python source code: quickstart.py

CHAPTER 6

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:

6.1 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 torch.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
```

6.2 Defining the Data

We get the MNIST dataset from torchvision, split it into a train and validation set and transform them to torch tensors.

```
BATCH_SIZE = 128

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

# Define standard classification mnist dataset with random validation set

dataset = torchvision.datasets.MNIST('./data/mnist', train=True, download=True,␣
↪transform=transform)
splitter = DatasetValidationSplitter(len(dataset), 0.1)
basetrainset = splitter.get_train_dataset(dataset)
basevalset = splitter.get_val_dataset(dataset)
```

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.

```
trainset = AutoEncoderMNIST(basetrainset)

valset = AutoEncoderMNIST(basevalset)

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

valgen = torch.utils.data.DataLoader(valset, batch_size=BATCH_SIZE, shuffle=True, num_
↪workers=8)
```

6.3 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 binary_cross_entropy(y_pred, y_true):
    BCE = F.binary_cross_entropy(y_pred, y_true, reduction='sum')
    return BCE
```

```
def kld(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.

6.3.1 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
```

6.3.2 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. We can (and should) generate unique state keys for interacting with state:

```
# State keys
MU, LOGVAR = torchbearer.state_key('mu'), torchbearer.state_key('logvar')
```

By default the model forward pass does not have access to the state dictionary, but torchbearer will infer the `state` argument from the model forward.

```
from torchbearer import Trial
```

(continues on next page)

(continued from previous page)

```
torchbearer_trial = Trial(model, optimizer, loss, metrics=['acc', 'loss'],
                          callbacks=[add_kld_loss_callback, save_reconstruction_
↪callback()]).to('cuda')
```

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 = binary_cross_entropy
```

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(state[MU], state[LOGVAR])
    return KLD
```

6.4 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
```

6.5 Training the Model

We train the model by creating a torchmodel and a torchbearertrialand calling run. As our loss is named binary_cross_entropy, we can use the 'acc' metric to get a binary accuracy.

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

from torchbearer import Trial

torchbearer_trial = Trial(model, optimizer, loss, metrics=['acc', 'loss'],
                          callbacks=[add_kld_loss_callback, save_reconstruction_
↪callback()]).to('cuda')
torchbearer_trial.with_generators(train_generator=traingen, val_generator=valgen)
torchbearer_trial.run(epochs=10)
```

This gives the following output:

```
0/10(t): 100%|| 422/422 [00:01<00:00, 219.71it/s, binary_acc=0.9139, loss=2.139e+4,␣
↪running_binary_acc=0.9416, running_loss=1.685e+4]
0/10(v): 100%|| 47/47 [00:00<00:00, 269.77it/s, val_binary_acc=0.9505, val_loss=1.
↪558e+4]
1/10(t): 100%|| 422/422 [00:01<00:00, 219.80it/s, binary_acc=0.9492, loss=1.573e+4,␣
↪running_binary_acc=0.9531, running_loss=1.52e+4]
1/10(v): 100%|| 47/47 [00:00<00:00, 300.54it/s, val_binary_acc=0.9614, val_loss=1.
↪399e+4]
2/10(t): 100%|| 422/422 [00:01<00:00, 232.41it/s, binary_acc=0.9558, loss=1.476e+4,␣
↪running_binary_acc=0.9571, running_loss=1.457e+4]
2/10(v): 100%|| 47/47 [00:00<00:00, 296.49it/s, val_binary_acc=0.9652, val_loss=1.
↪336e+4]
3/10(t): 100%|| 422/422 [00:01<00:00, 213.10it/s, binary_acc=0.9585, loss=1.437e+4,␣
↪running_binary_acc=0.9595, running_loss=1.423e+4]
3/10(v): 100%|| 47/47 [00:00<00:00, 313.42it/s, val_binary_acc=0.9672, val_loss=1.
↪304e+4]
4/10(t): 100%|| 422/422 [00:01<00:00, 213.43it/s, binary_acc=0.9601, loss=1.413e+4,␣
↪running_binary_acc=0.9605, running_loss=1.409e+4]
4/10(v): 100%|| 47/47 [00:00<00:00, 242.23it/s, val_binary_acc=0.9683, val_loss=1.
↪282e+4]
5/10(t): 100%|| 422/422 [00:01<00:00, 220.94it/s, binary_acc=0.9611, loss=1.398e+4,␣
↪running_binary_acc=0.9614, running_loss=1.397e+4]
5/10(v): 100%|| 47/47 [00:00<00:00, 316.69it/s, val_binary_acc=0.9689, val_loss=1.
↪281e+4]
6/10(t): 100%|| 422/422 [00:01<00:00, 230.53it/s, binary_acc=0.9619, loss=1.385e+4,␣
↪running_binary_acc=0.9621, running_loss=1.38e+4]
6/10(v): 100%|| 47/47 [00:00<00:00, 241.06it/s, val_binary_acc=0.9695, val_loss=1.
↪275e+4]
7/10(t): 100%|| 422/422 [00:01<00:00, 227.49it/s, binary_acc=0.9624, loss=1.377e+4,␣
↪running_binary_acc=0.9624, running_loss=1.381e+4]
7/10(v): 100%|| 47/47 [00:00<00:00, 237.75it/s, val_binary_acc=0.97, val_loss=1.
↪258e+4]
8/10(t): 100%|| 422/422 [00:01<00:00, 220.68it/s, binary_acc=0.9629, loss=1.37e+4,␣
↪running_binary_acc=0.9629, running_loss=1.369e+4]
8/10(v): 100%|| 47/47 [00:00<00:00, 301.59it/s, val_binary_acc=0.9704, val_loss=1.
↪255e+4]
9/10(t): 100%|| 422/422 [00:01<00:00, 215.23it/s, binary_acc=0.9633, loss=1.364e+4,␣
↪running_binary_acc=0.9633, running_loss=1.366e+4]
9/10(v): 100%|| 47/47 [00:00<00:00, 309.51it/s, val_binary_acc=0.9707, val_loss=1.
↪25e+4]
```

The visualised results after ten epochs then look like this:

6.6 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 7

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.

7.1 Data and Constants

We first define all constants for the example.

```
# Define constants
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)
batch = torch.randn(25, latent_dim).to(device)
```

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

```
# Register state keys (optional)
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')
```

(continues on next page)

(continued from previous page)

```
DISC_OPT = state_key('disc_opt')
GEN_OPT = state_key('gen_opt')
DISC_MODEL = state_key('disc_model')
DISC_IMGS = state_key('disc_imgs')
DISC_CRIT = state_key('disc_crit')
```

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)
```

7.2 Model

We use the generator and discriminator from PyTorch_GAN.

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

        def block(in_feat, out_feat, normalize=True):
            layers = [nn.Linear(in_feat, out_feat)]
            if normalize:
                layers.append(nn.BatchNorm1d(out_feat, 0.8))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers

        self.model = nn.Sequential(
            *block(latent_dim, 128, normalize=False),
            *block(128, 256),
            *block(256, 512),
            *block(512, 1024),
            nn.Linear(1024, int(np.prod(img_shape))),
            nn.Tanh()
        )

    def forward(self, real_imgs, state):
        z = Variable(torch.Tensor(np.random.normal(0, 1, (real_imgs.shape[0], latent_
↪dim)))).to(state[tb.DEVICE])
        img = self.model(z)
        img = img.view(img.size(0), *img_shape)
        return img


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

(continues on next page)

(continued from previous page)

```
        self.model = nn.Sequential(
            nn.Linear(int(np.prod(img_shape)), 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, img, state):
        img_flat = img.view(img.size(0), -1)
        validity = self.model(img_flat)

        return validity
```

We then create the models and optimisers.

```
# Model and optimizer
generator = Generator()
discriminator = Discriminator()
optimizer_G = torch.optim.Adam(generator.parameters(), lr=lr, betas=(0.5, 0.999))
optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=lr, betas=(0.5, 0.999))
```

7.3 Loss

GANs usually require two different losses, one for the generator and one for the discriminator. We define these as functions of state so that we can access the discriminator model for the additional forward passes required.

```
def gen_crit(state):
    loss =  adversarial_loss(state[DISC_MODEL](state[tb.Y_PRED], state), valid)
    state[G_LOSS] = loss
    return loss


def disc_crit(state):
    real_loss = adversarial_loss(state[DISC_MODEL](state[tb.X], state), valid)
    fake_loss = adversarial_loss(state[DISC_MODEL](state[tb.Y_PRED].detach(), state),␣
↪fake)
    loss = (real_loss + fake_loss) / 2
    state[D_LOSS] = loss
    return loss
```

We will see later how we get a torchbearer trial to use these losses.

7.4 Metrics

We would like to follow the discriminator and generator losses during training - note that we added these to state during the model definition. In torchbearer, state keys are also metrics, so we can take means and running means of them and tell torchbearer to output them as metrics.

```
from torchbearer.metrics import mean, running_mean
metrics = ['loss', mean(running_mean(D_LOSS)), mean(running_mean(G_LOSS))]
```

We will add this metric list to the trial when we create it.

7.5 Closures

The training loop of a GAN is a bit different to a standard model training loop. GANs require separate forward and backward passes for the generator and discriminator. To achieve this in torchbearer we can write a new closure. Since the individual training loops for the generator and discriminator are the same as a standard training loop we can use a `base_closure()`. The base closure takes state keys for required objects (data, model, optimiser, etc.) and returns a standard closure consisting of:

1. Zero gradients
2. Forward pass
3. Loss calculation
4. Backward pass

We create a separate closure for the generator and discriminator. We use separate state keys for some objects so we can use them separately, although the loss is easier to deal with in a single key.

```
from torchbearer.bases import base_closure
closure_gen = base_closure(tb.X, tb.MODEL, tb.Y_PRED, tb.Y_TRUE, tb.CRITERION, tb.
↪LOSS, GEN_OPT)
closure_disc = base_closure(tb.Y_PRED, DISC_MODEL, None, DISC_IMGS, DISC_CRIT, tb.
↪LOSS, DISC_OPT)
```

We then create a main closure (a simple function of state) that runs both of these and steps the optimisers.

```
def closure(state):
    closure_gen(state)
    state[GEN_OPT].step()
    closure_disc(state)
    state[DISC_OPT].step()
```

We will add this closure to the trial next.

7.6 Training

We now create the torchbearer trial on the GPU in the standard way. Note that when torchbearer is passed a `None` optimiser it creates a mock optimser that will just run the closure. Since we are using the standard torchbearer state keys for the generator model and criterion, we can pass them in here.

```
trial = tb.Trial(generator, None, criterion=gen_crit, metrics=metrics,␣
↪callbacks=[saver_callback])
trial.with_train_generator(dataloader, steps=200000)
trial.to(device)
```

We now update state with the keys required for the discriminators closure and add the new closure to the trial. Note that torchbearer doesn't know the discriminator model is a model here, so we have to sent it to the GPU ourselves.

```
new_keys = {DISC_MODEL: discriminator.to(device), DISC_OPT: optimizer_D, GEN_OPT:␣
↪optimizer_G, DISC_CRIT: disc_crit}
trial.state.update(new_keys)
trial.with_closure(closure)
```

Finally we run the trial.

```
trial.run(epochs=1)
```

7.7 Visualising

We borrow the image saving method from PyTorch_GAN and put it in a call back to save *on_step_training()*. We generate from the same inputs each time to get a better visualisation.

```
@callbacks.on_step_training
@callbacks.only_if(lambda state: state[tb.BATCH] % sample_interval == 0)
def saver_callback(state):
    samples = state[tb.MODEL](batch, state)
    save_image(samples, 'images/%d.png' % state[tb.BATCH], nrow=5, normalize=True)
```

Here is a Gif created from the saved images.

7.8 Source Code

The source code for the example is given below:

Download Python source code: gan.py

CHAPTER 8

Visualising CNNs: The Class Appearance Model

In this example we will demonstrate the *ClassAppearanceModel* callback included in torchbearer. This implements one of the most simple (and therefore not always the most successful) deep visualisation techniques, discussed in the paper Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps.

8.1 Background

The process to obtain Figure 1 from the paper is simple, given a particular target class c, we use back-propagation to obtain

$\text{argmax}_I \ S_c(I) - \lambda \|I\|_2^2$,

where $S_c(I)$ is the un-normalised score of c for the image I given by the network. The regularisation term $\|I\|_2^2$ is necessary to prevent the resultant image from becoming overly noisy. More recent visualisation techniques use much more advanced regularisers to obtain smoother, more realistic images.

8.2 Loading the Model

Since we are just running the callback on a pre-trained model, we don't need to load any data in this example. Instead, we use torchvision to load an Inception V1 trained on ImageNet with the following:

```
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.net = torchvision.models.googlenet(True)

    def forward(self, input):
        if input is not None:
            return self.net(input)


model = Model()
```

We need to include the *None* check as we will initialise the *Trial* without a dataloader, and so it will pass *None* to the model forward.

8.3 Running with the Callback

When using imaging callbacks, we commonly need to include an inverse transform to return the images to the right space. For torchvision, ImageNet models we can use the following:

```
inv_normalize = transforms.Normalize(
    mean=[-0.485/0.229, -0.456/0.224, -0.406/0.225],
    std=[1/0.229, 1/0.224, 1/0.225]
)
```

Finally we can construct and run the *Trial* with:

```
trial = Trial(model, callbacks=[
    imaging.ClassAppearanceModel(1000, (3, 224, 224), steps=10000, target=951,␣
↪transform=inv_normalize)
                .on_val().to_file('lemon.png'),
    imaging.ClassAppearanceModel(1000, (3, 224, 224), steps=10000, target=968,␣
↪transform=inv_normalize)
                .on_val().to_file('cup.png')
])
trial.for_val_steps(1).to('cuda')
trial.evaluate()
```

Here we create two *ClassAppearanceModel* instances which target the *lemon* and *cup* classes respectively. Since the *ClassAppearanceModel* is an *ImagingCallback*, we use the imaging API to send each of these to files. Finally, we evaluate the model for a single step to generate the results.

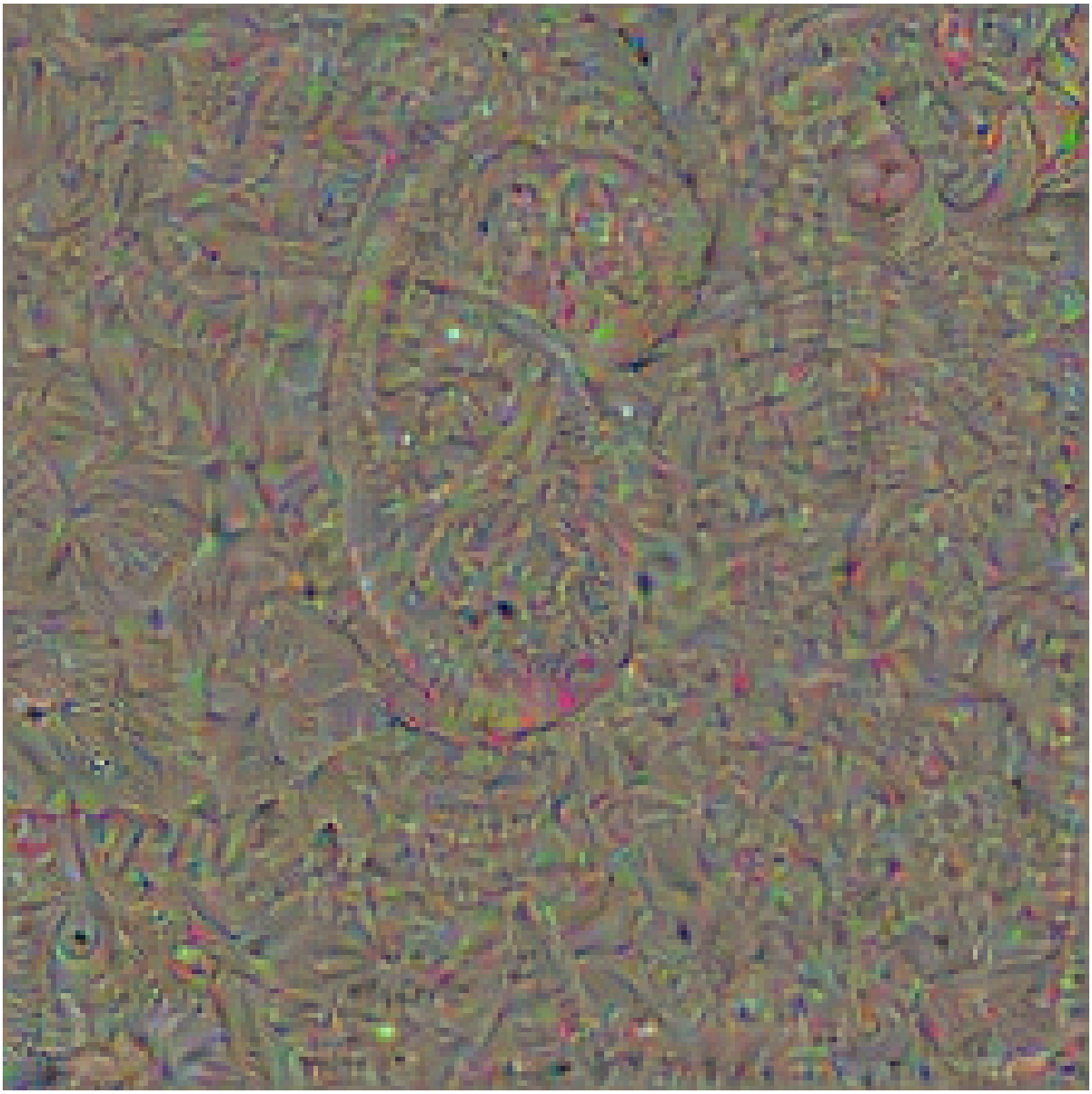
8.4 Results

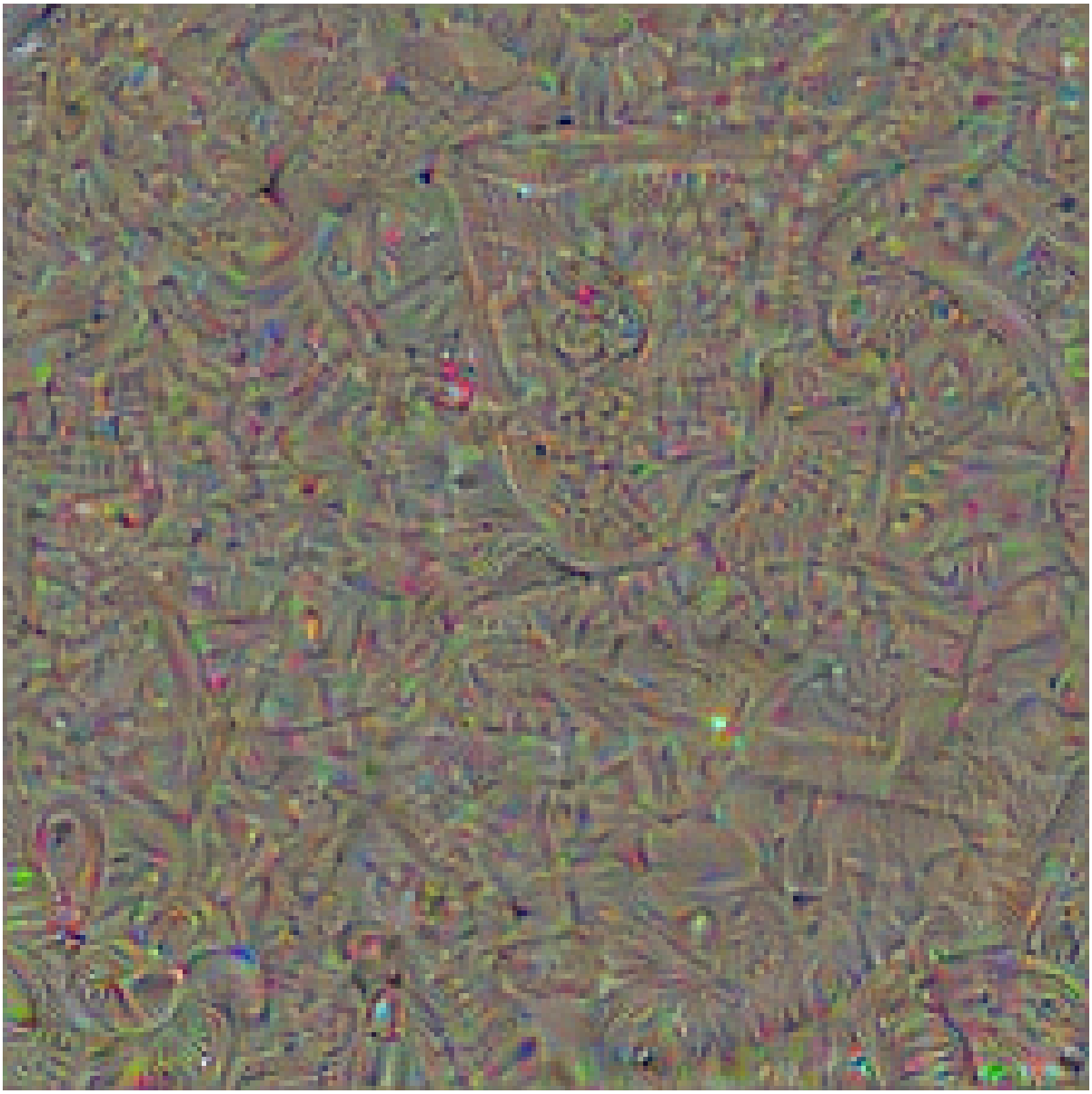
The results for the above code are given below. There some shapes which resemble a lemon or cup, however, not to the same extent shown in the paper. Because of the simplistic regularisation and objective, this model is highly sensitive to hyper-parameter choices. These results could almost certainly be improved with some more careful selection.

8.5 Source Code

The source code for the example is given below:

Download Python source code: cam.py





CHAPTER 9

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.

9.1 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[ESTIMATE] = self.pars.detach().unsqueeze(1)
        return self.f()
```

9.2 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
```

9.3 Optimising

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

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

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

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

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

```
tbtrial = tb.Trial(model, optim, loss, [tb.metrics.running_mean(ESTIMATE, dim=1),
↪'loss'])
tbtrial.for_train_steps(training_steps).to('cuda')
tbtrial.run()
print(list(model.parameters())[0].data)
```

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

9.4 Viewing Progress

You might have noticed in the previous snippet that the example uses a metric we've not seen before. The state key that represents our estimate in state can also act as a metric and is created at the beginning of the file with:

```

```

Putting all of it together and running provides the following output:

```
0/1(t): 100%|| 50000/50000 [00:53<00:00, 931.36it/s, loss=4.5502, running_est=[4.9988,
↪ 0.0, 1.0004], running_loss=0.0]
```

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

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

9.5 Source Code

The source code for the example is given below:

Download Python source code: basic_opt.py

CHAPTER 10

Linear Support Vector Machine (SVM)

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

10.1 SVM Recap

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

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

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

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

10.2 Defining the Model

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

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

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

    def forward(self, x):
```

(continues on next page)

(continued from previous page)

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

Next, we define the hinge loss function:

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

10.3 Creating Synthetic Data

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

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

10.4 Subgradient Descent

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

```
svm = LinearSVM()
model = Trial(svm, optim.SGD(svm.parameters(), 0.1), hinge_loss, ['loss'],
              callbacks=[scatter, draw_margin, ExponentialLR(0.999, step_on_
↪batch=True), L2WeightDecay(0.01, params=[svm.w])]).to('cuda')
model.with_train_data(X, Y, batch_size=32)
model.run(epochs=50, verbose=1)

plt.ioff()
plt.show()
```

10.5 Visualizing the Training

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

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

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

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

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

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

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

For our outputs $z \in \mathbf{Z}$, we can make some observations about the decision boundary. First, that we are outside the margin if $z - 1$ or $z1$. Conversely, we are inside the margine where $z - 1$ or $z1$. This gives us some rules for colouring, which we use here:

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

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

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

        mypause(0.001)
```

10.6 Final Comments

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

10.7 Source Code

The source code for the example is given below:

Download Python source code: svm_linear.py

CHAPTER 11

Breaking ADAM

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

11.1 Online Optimization

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

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

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

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

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

        return res
```

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

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


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

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

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

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

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

```
training_steps = 6000000

model = Online()
optim = torch.optim.Adam(model.parameters(), lr=0.001, betas=[0.9, 0.99])
tbtrial = tb.Trial(model, optim, loss, [est()], pass_state=True, callbacks=[greedy_
↪update, TensorBoard(comment='adam', write_graph=False, write_batch_metrics=True,␣
↪write_epoch_metrics=False)])
tbtrial.for_train_steps(training_steps).run()

model = Online()
optim = torch.optim.Adam(model.parameters(), lr=0.001, betas=[0.9, 0.99],␣
↪amsgrad=True)
tbtrial = tb.Trial(model, optim, loss, [est()], pass_state=True, callbacks=[greedy_
↪update, TensorBoard(comment='amsgrad', write_graph=False, write_batch_metrics=True,␣
↪write_epoch_metrics=False)])
tbtrial.for_train_steps(training_steps).run()
```

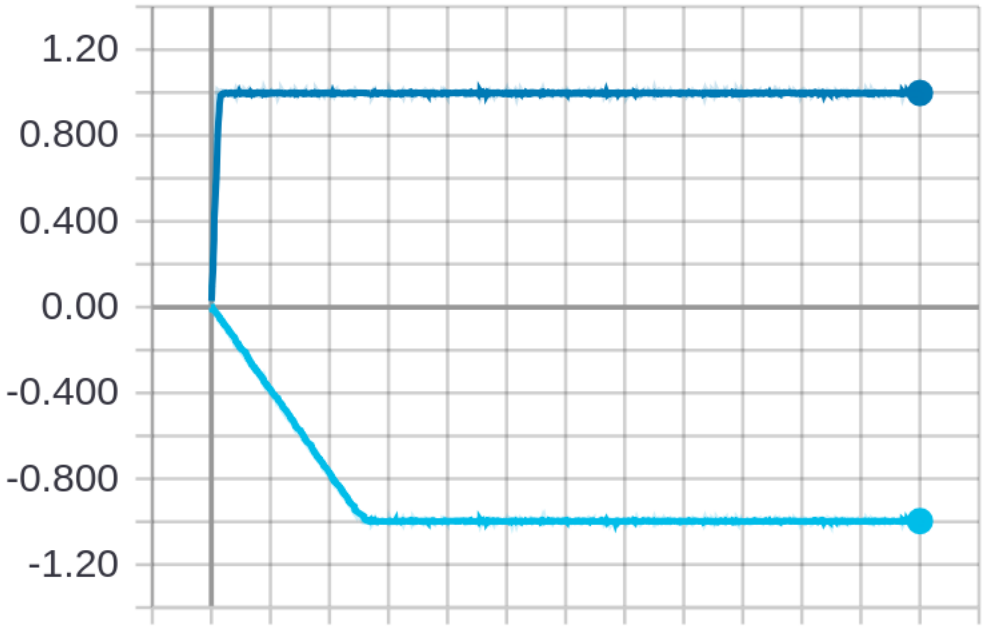
Note that we have logged to TensorBoard here and after completion, running `tensorboard --logdir logs` and navigating to localhost:6006, we can see a graph like the one in Figure 1 from the paper, where the top line is with ADAM and the bottom with AMSGrad:

11.2 Stochastic Optimization

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

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

We can again formulate this as a PyToch model:




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

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

        return res
```

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

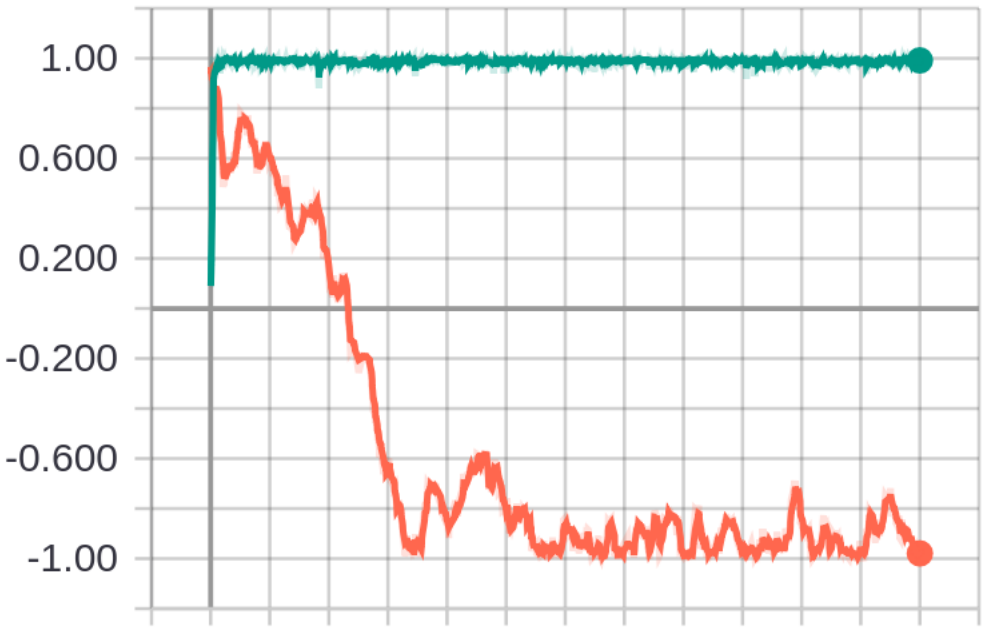
```
model = Stochastic()
optim = torch.optim.Adam(model.parameters(), lr=0.001, betas=[0.9, 0.99])
tbtrial = tb.Trial(model, optim, loss, [est()], callbacks=[greedy_update,␣
↪TensorBoard(comment='adam', write_graph=False, write_batch_metrics=True, write_
↪epoch_metrics=False)])
tbtrial.for_train_steps(training_steps).run()

model = Stochastic()
optim = torch.optim.Adam(model.parameters(), lr=0.001, betas=[0.9, 0.99],␣
↪amsgrad=True)
tbtrial = tb.Trial(model, optim, loss, [est()], callbacks=[greedy_update,␣
↪TensorBoard(comment='amsgrad', write_graph=False, write_batch_metrics=True, write_
↪epoch_metrics=False)])
```

(continues on next page)

(continued from previous page)

```
tbtrial.for_train_steps(training_steps).run()
```

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



11.3 Conclusions

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

11.4 Source Code

The source code for this example can be downloaded below:

`Download Python source code: amsgrad.py`

CHAPTER 12

torchbearer

12.1 Trial

class torchbearer.trial.**CallbackListInjection**(*callback*, *callback_list*)

This class allows for an callback to be injected into a callback list, without masking the methods available for mutating the list. In this way, callbacks (such as printers) can be injected seamlessly into the methods of the trial class.

Parameters

- **callback** (Callback) – The *Callback* to inject
- **callback_list** (CallbackList) – The underlying *CallbackList*

append(*callback_list*)

copy()

load_state_dict(*state_dict*)

Resume this callback list from the given state. Callbacks must be given in the same order for this to work.

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

Returns self

Return type *CallbackList*

state_dict()

Get a dict containing all of the callback states.

Returns A dict containing parameters and persistent buffers.

Return type dict

class torchbearer.trial.**MockOptimizer**

The Mock Optimizer will be used inplace of an optimizer in the event that none is passed to the Trial class.

add_param_group(*param_group*)

load_state_dict(*state_dict*)

state_dict()

step(*closure=None*)

zero_grad()

class torchbearer.trial.**Sampler**(*batch_loader*)
Sampler wraps a batch loader function and executes it when *Sampler.sample()* is called

Parameters **batch_loader** (*func*) – The batch loader to execute

sample(*state*)

class torchbearer.trial.**Trial**(*model*, *optimizer=None*, *criterion=None*, *metrics=[]*, *callbacks=[]*, *verbose=2*)
The trial class contains all of the required hyper-parameters for model running in torchbearer and presents an API for model fitting, evaluating and predicting.

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

Parameters

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

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

Returns self

Return type *Trial*

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

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

Returns self

Return type *Trial*

eval()
Set model and metrics to evaluation mode

Returns self

Return type *Trial*

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

Parameters

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

Returns The final metric values

Return type dict

for_inf_steps(*train=True*, *val=True*, *test=True*)
Use this trail with infinite steps. Returns self so that methods can be chained for convenience.

Parameters

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

Returns self

Return type *Trial*

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

Returns self

Return type *Trial*

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

Returns self

Return type *Trial*

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

Returns self

Return type *Trial*

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

Parameters

- **train_steps** (*int*) – The number of training steps per epoch to run
- **val_steps** (*int*) – The number of validation steps per epoch to run

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

Returns self

Return type *Trial*

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

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

Returns self

Return type *Trial*

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

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

Returns self

Return type *Trial*

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

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

Returns self

Return type *Trial*

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

Parameters

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

Returns self

Return type *Trial*

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

Parameters

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

Returns Model outputs as a list

Return type list

replay(*callbacks=[]*, *verbose=2*, *one_batch=False*)
Replay the fit passes stored in history with given callbacks, useful when reloading a saved Trial. Note that only progress and metric information is populated in state during a replay.

Parameters

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

Returns self

Return type *Trial*

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

Parameters

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

State Requirements:

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

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

Return type list

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

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

Return type *Trial*

train()
Set model and metrics to training mode.

Returns self

Return type *Trial*

with_closure(*closure*)
Use this trial with custom closure

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

Returns self:

Return type *Trial*

with_generators(*train_generator=None*, *val_generator=None*, *test_generator=None*, *train_steps=None*, *val_steps=None*, *test_steps=None*)
Use this trial with the given generators. Returns self so that methods can be chained for convenience.

Parameters

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

Returns self

Return type *Trial*

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

Returns self:

Return type *Trial*

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

Parameters

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

Returns self

Return type *Trial*

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

Parameters

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

Returns self

Return type *Trial*

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

Parameters

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

Returns self

Return type *Trial*

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

Parameters

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

Returns self

Return type *Trial*

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

Parameters

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

Returns self

Return type *Trial*

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

Parameters

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

Returns self

Return type *Trial*

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

Parameters

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

Returns The moved or casted batch

Return type tuple / list / torch.Tensor

torchbearer.trial.**get_default**(*fcn*, *arg*)

torchbearer.trial.**get_printer**(*verbose*, *validation_label_letter*)

torchbearer.trial.**inject_callback**(*callback*)
Decorator to inject a callback into the callback list and remove the callback after the decorated function has executed

Parameters **callback** (Callback) – *Callback* to be injected

Returns The decorator

torchbearer.trial.**inject_printer**(*validation_label_letter='v'*)
The inject printer decorator is used to inject the appropriate printer callback, according to the verbosity level.

Parameters **validation_label_letter** (*str*) – The validation label letter to use

Returns A decorator

torchbearer.trial.**inject_sampler**(*data_key*, *predict=False*)
Decorator to inject a *Sampler* into state[torchbearer.SAMPLER] along with the specified generator into state[torchbearer.GENERATOR] and number of steps into state[torchbearer.STEPS]

Parameters

- **data_key** (StateKey) – *StateKey* for the data to inject
- **predict** (*bool*) – If true, the prediction batch loader is used, if false the standard data loader is used

Returns The decorator

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

Parameters **loader** – batch loader to wrap

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

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

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

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

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

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

torchbearer.trial.**update_device_and_dtype**(*state*, **args*, ***kwargs*)
Function get data type and device values from the args / kwargs and update state.

Parameters

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

Returns device, dtype pair

12.2 State

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

Example:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

class torchbearer.state.**State**
State dictionary that behaves like a python dict but accepts StateKeys

data

get_key(*statekey*)

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

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

Parameters **key** (*str*) – Base key

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

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

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

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

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

Returns None or the value of the metric for this epoch

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Returns New state key

Return type *StateKey*

12.3 Utilities

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

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

Returns Validation dataset split from whole dataset

Return type torch.utils.data.Dataset

class torchbearer.cv_utils.**SubsetDataset**(*dataset*, *ids*)

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

Parameters

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

Returns Training and validation datasets

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

Parameters

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

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

CHAPTER 13

torchbearer.callbacks

13.1 Base Classes

class torchbearer.bases.**Callback**

Base callback class.

Note: All callbacks should override this class.

load_state_dict(*state_dict*)

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

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

Returns self

Return type *Callback*

on_backward(*state*)

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

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

on_checkpoint(*state*)

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

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

on_criterion(*state*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

state_dict()
Get a dict containing the callback state.

Returns A dict containing parameters and persistent buffers.

Return type dict

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

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

CALLBACK_STATES = 'callback_states'

CALLBACK_TYPES = 'callback_types'

append(*callback_list*)

copy()

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

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

Returns self

Return type *CallbackList*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Returns A dict containing parameters and persistent buffers.

Return type dict

13.2 Imaging

13.2.1 Main Classes

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

Parameters

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

on_batch(*state*)

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

Parameters

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

Returns The processed image

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

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

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

Parameters

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

on_batch(*state*)

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

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

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

on_batch(*state*)

on_test()

Process this callback for test batches

Returns self

Return type *ImagingCallback*

on_train()

Process this callback for training batches

Returns self

Return type *ImagingCallback*

on_val()

Process this callback for validation batches

Returns self

Return type *ImagingCallback*

process(*state*)

to_file(*filename*)

Send images from this callback to the given file

Parameters **filename** (*str*) – the filename to store the image to

Returns self

Return type *ImagingCallback*

to_pyplot()

Show images from this callback with pyplot

Returns self

Return type *ImagingCallback*

to_state(*key*)

Put images from this callback in state with the given key

Parameters **key** (StateKey) – The state key to use for the image

Returns self

Return type *ImagingCallback*

to_tensorboard(*name='Image'*, *log_dir='./logs'*, *comment='torchbearer'*)
Direct images from this callback to tensorboard with the given parameters

Parameters

- **name** (*str*) – The name of the image
- **log_dir** (*str*) – The tensorboard log path for output
- **comment** (*str*) – Descriptive comment to append to path

Returns self

Return type *ImagingCallback*

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

Parameters

- **name** (*str*) – The name of the image
- **log_dir** (*str*) – The visdom log path for output
- **comment** (*str*) – Descriptive comment to append to path
- **visdom_params** (*VisdomParams*) – Visdom parameter settings object, uses default if None

Returns self

Return type *ImagingCallback*

with_handler(*handler*)
Append the given output handler to the list of handlers

Parameters **handler** – A function of image and state which stores the given image in some way

Returns self

Return type *ImagingCallback*

class torchbearer.callbacks.imaging.imaging.**MakeGrid**(*key=x*, *transform=None*, *num_images=16*, *nrow=8*, *padding=2*, *normalize=False*, *norm_range=None*, *scale_each=False*, *pad_value=0*)
The *MakeGrid* callback is a *CachingImagingCallback* which calls make grid on the cache with the provided parameters.

Parameters

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

- **norm_range** – See torchvision.utils.make_grid
- **scale_each** – See torchvision.utils.make_grid
- **pad_value** – See torchvision.utils.make_grid

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

Parameters

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

Returns The processed image

13.2.2 Deep Inside Convolutional Networks

class torchbearer.callbacks.imaging.inside_cnns.**ClassAppearanceModel**(*nclasses*, *input_size*, *optimizer_factory=<function ClassAppearanceModel.<lambda>>*, *steps=1024*, *logit_key=y_pred*, *prob_key=None*, *target=-10*, *decay=0.001*, *verbose=0*, *transform=None*)

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

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

Parameters

- **nclasses** (*int*) – The number of output classes
- **input_size** (*tuple*) – The size to use for the input image
- **optimizer_factory** – A function of parameters which returns an optimizer to use

- **logit_key** (StateKey) – *StateKey* storing the class logits
- **prob_key** (StateKey) – *StateKey* storing the class probabilities or None if using logits
- **target** (*int*) – Target class for the optimisation or RANDOM
- **steps** (*int*) – Number of optimisation steps to take
- **decay** (*float*) – Lambda for the L2 decay on the image
- **verbose** (*int*) – Verbosity level to pass to the internal *Trial* instance
- **transform** (*callable, optional*) – A function/transform that takes in a Tensor and returns a transformed version. This will be applied to the image before it is sent to output

on_batch(*state*)

target_to_key(*key*)

13.3 Model Checkpointers

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

Model checkpointer which saves the best model according to the given configurations.

Parameters

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

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

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

Returns self

Return type *Callback*

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

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

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

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

state_dict()
Get a dict containing the callback state.

Returns A dict containing parameters and persistent buffers.

Return type dict

class torchbearer.callbacks.checkpointers.**Interval**(*filepath='model.{epoch:02d}-{val_loss:.2f}.pt'*, *save_model_params_only=False*, *period=1*, *on_batch=False*, *pickle_module=<sphinx.ext.autodoc.importer._MockObject object>*, *pickle_protocol=<sphinx.ext.autodoc.importer._MockObject object>*)
Model checkpointer which which saves the model every 'period' epochs to the given filepath.

Parameters

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

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

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

Returns self

Return type *Callback*

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

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

state_dict()
Get a dict containing the callback state.

Returns A dict containing parameters and persistent buffers.

Return type dict

torchbearer.callbacks.checkpointers.**ModelCheckpoint**(*filepath='model.{epoch:02d}-{val_loss:.2f}.pt'*, *save_model_params_only=False*, *monitor='val_loss'*, *save_best_only=False*, *mode='auto'*, *period=1*, *min_delta=0*)
Save the model after every epoch. *filepath* can contain named formatting options, which will be filled any values from state. For example: if *filepath* is *weights.{epoch:02d}-{val_loss:.2f}*, then the model checkpoints will be saved with the epoch number and the validation loss in the filename. The torch *Trial* will be saved to filename.

Parameters

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

class torchbearer.callbacks.checkpointers.**MostRecent**(*filepath='model.{epoch:02d}-{val_loss:.2f}.pt'*, *save_model_params_only=False*, *pickle_module=<sphinx.ext.autodoc.importer._MockObj object>*, *pickle_protocol=<sphinx.ext.autodoc.importer._MockO object>*)
Model checkpointer which saves the most recent model to a given filepath.

Parameters

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

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

on_checkpoint(*state*)

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

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

13.4 Logging

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

Callback to log metrics to a given csv file.

Parameters

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

on_end(*state*)

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

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

on_end_epoch(*state*)

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

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

on_step_training(*state*)

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

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

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

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

Parameters

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

State Requirements:

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

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

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

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

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

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

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

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

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

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

Parameters

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

State Requirements:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

13.5 Tensorboard, Visdom and Others

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

Parameters

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

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

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

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

Parameters

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

Returns the *SummaryWriter* or *VisdomWriter*

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

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

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

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

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

Parameters

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

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

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

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

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

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

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

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

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

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

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

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

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

class torchbearer.callbacks.tensor_board.**TensorBoardImages**(*log_dir='./logs'*, *comment='torchbearer'*, *name='Image'*, *key=y_pred*, *write_each_epoch=True*, *num_images=16*, *nrow=8*, *padding=2*, *normalize=False*, *norm_range=None*, *scale_each=False*, *pad_value=0*, *visdom=False*, *visdom_params=None*)
The TensorBoardImages callback will write a selection of images from the validation pass to tensorboard using the TensorboardX library and torchvision.utils.make_grid (requires torchvision). Images are selected from the given key and saved to the given path. Full name of image sub directory will be model name + _ + comment.

Parameters

- **log_dir** (*str*) – The tensorboard log path for output
- **comment** (*str*) – Descriptive comment to append to path
- **name** (*str*) – The name of the image
- **key** (StateKey) – The key in state containing image data (tensor of size [c, w, h] or [b, c, w, h])
- **write_each_epoch** (*bool*) – If True, write data on every epoch, else write only for the first epoch.
- **num_images** (*int*) – The number of images to write
- **nrow** – See torchvision.utils.make_grid

- **padding** – See torchvision.utils.make_grid
- **normalize** – See torchvision.utils.make_grid
- **norm_range** – See torchvision.utils.make_grid
- **scale_each** – See torchvision.utils.make_grid
- **pad_value** – See torchvision.utils.make_grid
- **visdom** (*bool*) – If true, log to visdom instead of tensorboard
- **visdom_params** (VisdomParams) – Visdom parameter settings object, uses default if None

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

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

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

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

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

Parameters

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

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

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

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

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

class torchbearer.callbacks.tensor_board.**TensorBoardText**(*log_dir='./logs'*, *write_epoch_metrics=True*, *write_batch_metrics=False*, *log_trial_summary=True*, *batch_step_size=100*, *comment='torchbearer'*, *visdom=False*, *visdom_params=None*)

TensorBoard callback which writes metrics as text to the given log directory. Requires the TensorboardX library for python.

Parameters

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

on_end(*state*)

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

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

on_end_epoch(*state*)

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

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

on_start(*state*)

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

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

on_start_epoch(*state*)

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

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

on_step_training(*state*)

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

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

static table_formatter(*string*)

class torchbearer.callbacks.tensor_board.**VisdomParams**

Class to hold visdom client arguments. Modify member variables before initialising tensorboard callbacks for custom arguments. See: visdom

ENDPOINT = 'events'

ENV = 'main'

HTTP_PROXY_HOST = None

HTTP_PROXY_PORT = None

IPV6 = True

LOG_TO_FILENAME = None

PORT = 8097

RAISE_EXCEPTIONS = None

SEND = True

SERVER = 'http://localhost'

USE_INCOMING_SOCKET = True

torchbearer.callbacks.tensor_board.**close_writer**(*log_dir*, *logger*)

Decrement the reference count for a writer belonging to a specific log directory. If the reference count gets to zero, the writer will be closed and removed.

Parameters

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

torchbearer.callbacks.tensor_board.**get_writer**(*log_dir*, *logger*, *visdom=False*, *visdom_params=None*)

Get the writer assigned to the given log directory. If the writer doesn't exist it will be created, and a reference to the logger added.

Parameters

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

Returns the *SummaryWriter* or *VisdomWriter* object

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

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

Parameters

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

- **kwargs** – Keyword arguments for livelossplot.PlotLosses

State Requirements:

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

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

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

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

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

13.6 Early Stopping

class torchbearer.callbacks.early_stopping.**EarlyStopping**(*monitor='val_loss'*, *min_delta=0*, *patience=0*, *verbose=0*, *mode='auto'*)
Callback to stop training when a monitored quantity has stopped improving.

Parameters

- **monitor** (*str*) – Name of quantity in metrics to be monitored
- **min_delta** (*float*) – Minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than min_delta, will count as no improvement.
- **patience** (*int*) – Number of epochs with no improvement after which training will be stopped.
- **verbose** (*int*) – Verbosity mode, will print stopping info if verbose > 0
- **mode** (*str*) – One of {auto, min, max}. In *min* mode, training will stop when the quantity monitored has stopped decreasing; in *max* mode it will stop when the quantity monitored has stopped increasing; in *auto* mode, the direction is automatically inferred from the name of the monitored quantity.

State Requirements:

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

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

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

Returns self

Return type *Callback*

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

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

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

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

state_dict()
Get a dict containing the callback state.

Returns A dict containing parameters and persistent buffers.

Return type dict

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

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

State Requirements:

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

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

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

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

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

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

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

13.7 Gradient Clipping

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

Parameters

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

State Requirements:

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

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

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

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

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

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

Parameters

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

State Requirements:

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

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

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

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

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

13.8 Learning Rate Schedulers

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

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

See: PyTorch CosineAnnealingLR

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

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

See: PyTorch ExponentialLR

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

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

See: PyTorch LambdaLR

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

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

See: PyTorch MultiStepLR

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

Parameters

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

See: PyTorch ReduceLROnPlateau

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

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

See: PyTorch StepLR

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

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

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

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

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

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

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

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

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

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

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

13.9 Learning Rate Finders

class torchbearer.callbacks.lr_finder.**CyclicLR**(*base_lr=0.001*, *max_lr=0.006*, *step_size=2000*, *mode='triangular'*, *scale_fn=None*, *scale_mode='cycle'*, *gamma=1.0*)
Learning rate finder that cyclicly varies the rate. Based off of the keras implementation referenced in the paper.

```
@inproceedings{smith2017cyclical,
  title={Cyclical learning rates for training neural networks},
  author={Smith, Leslie N},
  booktitle={2017 IEEE Winter Conference on Applications of Computer Vision
↪(WACV)},
  pages={464--472},
  year={2017},
  organization={IEEE}
}
```

Parameters

- **base_lr** (*float / list*) – Float or list of floats for the base (min) learning rate for each optimiser parameter group
- **max_lr** (*float / list*) – Float or list of floats for the max learning rate for each optimiser parameter group
- **step_size** (*int / list*) – int or list of ints for the step size (half cyclic period) for each optimiser parameter group
- **mode** (*str*) – One of (triangular, triangular2, exp_range) - the mode to use
- **scale_fn** (*function*) – Scale function for learning rates over time. Default is defined by mode.
- **scale_mode** (*str*) – One of (cycle, iterations). Argument passed to the scale function each step
- **gamma** (*float*) – Scaling factor for exp_range mode

next_lr(*epoch_count*, *group_id*)

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

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

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

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

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

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

update_lrs()

13.10 Weight Decay

class torchbearer.callbacks.weight_decay.**L1WeightDecay**(*rate=0.0005*, *params=None*)
WeightDecay callback which uses an L1 norm with the given rate and parameters. If params is None (default) then the parameters will be retrieved from the model.

Parameters

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

State Requirements:

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

class torchbearer.callbacks.weight_decay.**L2WeightDecay**(*rate=0.0005*, *params=None*)
WeightDecay callback which uses an L2 norm with the given rate and parameters. If params is None (default) then the parameters will be retrieved from the model.

Parameters

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

State Requirements:

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

class torchbearer.callbacks.weight_decay.**WeightDecay**(*rate=0.0005*, *p=2*, *params=None*)
Create a WeightDecay callback which uses the given norm on the given parameters and with the given decay rate. If params is None (default) then the parameters will be retrieved from the model.

Parameters

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

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

State Requirements:

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

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

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

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

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

13.11 Weight / Bias Initialisation

class torchbearer.callbacks.init.**KaimingNormal**(*a=0, mode='fan_in', nonlinearity='leaky_relu', modules=None, targets=['Conv', 'Linear', 'Bilinear']*)
Kaiming Normal weight initialisation. Uses `torch.nn.init.kaiming_normal_` on the `weight` attribute of the filtered modules.

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

Parameters

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

See: PyTorch kaiming_normal_

class torchbearer.callbacks.init.**KaimingUniform**(*a=0, mode='fan_in', nonlinearity='leaky_relu', modules=None, targets=['Conv', 'Linear', 'Bilinear']*)
Kaiming Uniform weight initialisation. Uses `torch.nn.init.kaiming_uniform_` on the `weight` attribute of the filtered modules.

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

Parameters

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

See: PyTorch kaiming_uniform_

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

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

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

Parameters

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

State Requirements:

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

on_init(*state*)

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

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

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

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

Parameters

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

State Requirements:

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

on_init(*state*)

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

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

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

Xavier Normal weight initialisation. Uses `torch.nn.init.xavier_normal_` on the `weight` attribute of the filtered modules.

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

Parameters

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

See: PyTorch xavier_normal_

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

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

```
@inproceedings{glorot2010understanding,
  title={Understanding the difficulty of training deep feedforward neural
↪networks},
```

(continues on next page)

(continued from previous page)

```
  author={Glorot, Xavier and Bengio, Yoshua},
  booktitle={Proceedings of the thirteenth international conference on artificial␣
↪intelligence and statistics},
  pages={249--256},
  year={2010}
}
```

Parameters

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

See: PyTorch xavier_uniform_

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

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

Parameters

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

13.12 Decorators

class torchbearer.callbacks.decorators.**LambdaCallback**(*func*)

on_lambda(*state*)

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.**bind_to**(*target*)

torchbearer.callbacks.decorators.**count_args**(*fcn*)

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 *on_backward()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**on_checkpoint**(*func*)
The *on_checkpoint()* decorator is used to initialise a *Callback* with *on_checkpoint()* calling the decorated function

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

Returns Initialised callback with *on_checkpoint()* 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 *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 *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 *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 *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 *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 *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 *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 *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 *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 *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

Args: func (function): The function(state) to *decorate*

Returns Initialised callback with *on_start_epoch()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**on_start_training**(*func*)
The *on_start_training()* decorator is used to initialise a *Callback* with *on_start_training()* calling the decorated function

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

Returns Initialised callback with *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 *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 *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 *on_step_validation()* calling func

Return type *Callback*

torchbearer.callbacks.decorators.**once**(*fcn*)
Decorator to fire a callback once in the lifetime of the callback. If the callback is a class method, each instance of the class will fire only once. For functions, only the first instance will fire (even if more than one function is present in the callback list).

Parameters **fcn** (*function*) – the *torchbearer callback* function to decorate.

Returns the decorator

torchbearer.callbacks.decorators.**once_per_epoch**(*fcn*)
Decorator to fire a callback once (on the first call) in any given epoch. If the callback is a class method, each instance of the class will fire once per epoch. For functions, only the first instance will fire (even if more than one function is present in the callback list).

Note: The decorated callback may exhibit unusual behaviour if it is reused

---

Parameters **fcn** (*function*) – the *torchbearer callback* function to decorate.

Returns the decorator

torchbearer.callbacks.decorators.**only_if**(*condition_expr*)
Decorator to fire a callback only if the given conditional expression function returns True. The conditional

expression can be a function of state or self and state. If the decorated function is not a class method (i.e. it does not take state) the decorated function will be passed instead. This enables the storing of temporary variables.

Parameters **condition_expr** (*function(self, state) or function(self)*) – a function/lambda which takes state and optionally self that must evaluate to true for the decorated *torchbearer callback* to be called. The *state* object passed to the callback will be passed as an argument to the condition function.

Returns the decorator

CHAPTER 14

torchbearer.metrics

14.1 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.bases.**Metric**(*name*)

Base metric class. Process will be called on each batch, process-final at the end of each epoch. The metric contract allows for metrics to take any args but not kwargs. The initial metric call will be given state, however, subsequent metrics can pass any values desired.

Note: All metrics must extend this class.

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

eval(*data_key=None*)

Put the metric in eval mode during model validation.

process(**args*)

Process the state and update the metric for one iteration.

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

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

process_final(**args*)

Process the terminal state and output the final value of the metric.

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

Returns None or the value of the metric for this epoch

reset(*state*)
Reset the metric, called before the start of an epoch.

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

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

class torchbearer.metrics.metrics.**AdvancedMetric**(*name*)
The *AdvancedMetric* class is a metric which provides different process methods for training and validation. This enables running metrics which do not output intermediate steps during validation.

Parameters **name** (*str*) – The name of the metric.

eval(*data_key=None*)
Put the metric in eval mode.

Parameters **data_key** (StateKey) – The torchbearer data_key, if used

process(**args*)
Depending on the current mode, return the result of either 'process_train' or 'process_validate'.

Returns The metric value.

process_final(**args*)
Depending on the current mode, return the result of either 'process_final_train' or 'process_final_validate'.

Returns The final metric value.

process_final_train(**args*)
Process the given state and return the final metric value for a training iteration.

Returns The final metric value for a training iteration.

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

Returns The final metric value for a validation iteration.

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

Returns The metric value for a training iteration.

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

Returns The metric value for a validation iteration.

train()
Put the metric in train mode.

class torchbearer.metrics.metrics.**MetricList**(*metric_list*)
The *MetricList* class is a wrapper for a list of metrics which acts as a single metric and produces a dictionary of outputs.

Parameters **metric_list** (*list*) – The list of metrics to be wrapped. If the list contains a *MetricList*, this will be unwrapped. Any strings in the list will be retrieved from metrics.DEFAULT_METRICS.

eval(*data_key=None*)
Put each metric in eval mode

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

Returns A dictionary which maps metric names to values.

Return type dict[str,any]

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

Returns A dictionary which maps metric names to values.

Return type dict[str,any]

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

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

train()
Put each metric in train mode.

class torchbearer.metrics.metrics.**MetricTree**(*metric*)
A tree structure which has a node *Metric* and some children. Upon execution, the node is called with the input and its output is passed to each of the children. A dict is updated with the results.

Note: If the node output is already a dict (i.e. the node is a standalone metric), this is unwrapped before passing the **first** value to the children.

Parameters **metric** (Metric) – The metric to act as the root node of the tree / subtree

add_child(*child*)
Add a child to this node of the tree

Parameters **child** (Metric) – The child to add

eval(*data_key=None*)
Put the metric in eval mode during model validation.

process(**args*)
Process this node and then pass the output to each child.

Returns A dict containing all results from the children

process_final(**args*)
Process this node and then pass the output to each child.

Returns A dict containing all results from the children

reset(*state*)
Reset the metric, called before the start of an epoch.

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

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

torchbearer.metrics.metrics.**add_default**(*key*, *metric*, **args*, ***kwargs*)

torchbearer.metrics.metrics.**get_default**(*key*)

14.2 Decorators - The Decorator API

The decorator API is the core way to interact with metrics in torchbearer. All of the classes and functionality handled here can be reproduced by manually interacting with the classes if necessary. Broadly speaking, the decorator API is used to construct a `MetricFactory` which will build a *`MetricTree`* that handles data flow between instances of *`Mean`*, *`RunningMean`*, *`Std`* etc.

`torchbearer.metrics.decorators.`**`default_for_key`**(*key*, **args*, ***kwargs*)

The *`default_for_key()`* decorator will register the given metric in the global metric dict (*metrics.DEFAULT_METRICS*) so that it can be referenced by name in instances of *`MetricList`* such as in the list given to the `torchbearer.Model`.

Example:

```
@default_for_key('acc')
class CategoricalAccuracy(metrics.BatchLambda):
    ...
```

Parameters

- **`key`** (*`str`*) – The key to use when referencing the metric
- **`args`** – Any args to pass to the underlying metric when constructed
- **`kwargs`** – Any keyword args to pass to the underlying metric when constructed

`torchbearer.metrics.decorators.`**`lambda_metric`**(*name*, *on_epoch=False*)

The *`lambda_metric()`* decorator is used to convert a lambda function *y_pred, y_true* into a *`Metric`* instance. This can be used as in the following example:

```
@metrics.lambda_metric('my_metric')
def my_metric(y_pred, y_true):
    ... # Calculate some metric

model = Model(metrics=[my_metric])
```

Parameters

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

Returns A decorator which replaces a function with a *`Metric`*

`torchbearer.metrics.decorators.`**`mean`**(*clazz=None*, *dim=None*)

The *`mean()`* decorator is used to add a *`Mean`* to the *`MetricTree`* which will will output a mean value at the end of each epoch. At build time, if the inner class is not a *`MetricTree`*, one will be created. The *`Mean`* will also be wrapped in a *`ToDict`* for simplicity.

Example:

```
>>> import torch
>>> from torchbearer import metrics

>>> @metrics.mean
... @metrics.lambda_metric('my_metric')
... def metric(y_pred, y_true):
```

(continues on next page)

(continued from previous page)

```
...     return y_pred + y_true
...
>>> metric.reset({})
>>> metric.process({'y_pred':torch.Tensor([2]), 'y_true':torch.Tensor([2])}) # 4
{}
>>> metric.process({'y_pred':torch.Tensor([3]), 'y_true':torch.Tensor([3])}) # 6
{}
>>> metric.process({'y_pred':torch.Tensor([4]), 'y_true':torch.Tensor([4])}) # 8
{}
>>> metric.process_final()
{'my_metric': 6.0}
```

Parameters

- **clazz** – The class to *decorate*
- **dim** (*int, tuple*) – See *Mean*

Returns A *MetricTree* with a *Mean* appended or a wrapper class that extends *MetricTree*

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

The *running_mean()* decorator is used to add a *RunningMean* to the *MetricTree*. If the inner class is not a *MetricTree* then one will be created. The *RunningMean* will be wrapped in a *ToDict* (with 'running_' prepended to the name) for simplicity.

Note: The decorator function does not need to be called if not desired, both: *@running_mean* and *@running_mean()* are acceptable.

Example:

```
>>> import torch
>>> from torchbearer import metrics

>>> @metrics.running_mean(step_size=2) # Update every 2 steps
... @metrics.lambda_metric('my_metric')
... def metric(y_pred, y_true):
...     return y_pred + y_true
...
>>> metric.reset({})
>>> metric.process({'y_pred':torch.Tensor([2]), 'y_true':torch.Tensor([2])}) # 4
{'running_my_metric': 4.0}
>>> metric.process({'y_pred':torch.Tensor([3]), 'y_true':torch.Tensor([3])}) # 6
{'running_my_metric': 4.0}
>>> metric.process({'y_pred':torch.Tensor([4]), 'y_true':torch.Tensor([4])}) # 8,␣
↪triggers update
{'running_my_metric': 6.0}
```

Parameters

- **clazz** – The class to *decorate*
- **batch_size** (*int*) – See *RunningMean*
- **step_size** (*int*) – See *RunningMean*
- **dim** (*int, tuple*) – See *RunningMean*

Returns decorator or *MetricTree* instance or wrapper

torchbearer.metrics.decorators.**std**(*clazz=None*, *unbiased=True*, *dim=None*)

The *std()* decorator is used to add a *Std* to the *MetricTree* which will will output a sample standard deviation value at the end of each epoch. At build time, if the inner class is not a *MetricTree*, one will be created. The *Std* will also be wrapped in a *ToDict* (with '_std' appended) for simplicity.

Example:

```
>>> import torch
>>> from torchbearer import metrics

>>> @metrics.std
... @metrics.lambda_metric('my_metric')
... def metric(y_pred, y_true):
...     return y_pred + y_true
...
>>> metric.reset({})
>>> metric.process({'y_pred':torch.Tensor([2]), 'y_true':torch.Tensor([2])}) # 4
{}
>>> metric.process({'y_pred':torch.Tensor([3]), 'y_true':torch.Tensor([3])}) # 6
{}
>>> metric.process({'y_pred':torch.Tensor([4]), 'y_true':torch.Tensor([4])}) # 8
{}
>>> '%.4f' % metric.process_final()['my_metric_std']
'2.0000'
```

Parameters

- **clazz** – The class to *decorate*
- **unbiased** (*bool*) – See *Std*
- **dim** (*int, tuple*) – See *Std*

Returns A *MetricTree* with a *Std* appended or a wrapper class that extends *MetricTree*

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

The *to_dict()* decorator is used to wrap either a *Metric* class or a *Metric* instance with a *ToDict* instance. The result is that future output will be wrapped in a *dict[name, value]*.

Example:

```
>>> from torchbearer import metrics

>>> @metrics.lambda_metric('my_metric')
... def my_metric(y_pred, y_true):
...     return y_pred + y_true
...
>>> my_metric.process({'y_pred':4, 'y_true':5})
9

>>> @metrics.to_dict
... @metrics.lambda_metric('my_metric')
... def my_metric(y_pred, y_true):
...     return y_pred + y_true
...
>>> my_metric.process({'y_pred':4, 'y_true':5})
{'my_metric': 9}
```

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

Returns A *ToDict* instance or a *ToDict* wrapper of the given class

torchbearer.metrics.decorators.**var**(*clazz=None*, *unbiased=True*, *dim=None*)

The *var()* decorator is used to add a *Var* to the *MetricTree* which will will output a sample variance value at the end of each epoch. At build time, if the inner class is not a *MetricTree*, one will be created. The *Var* will also be wrapped in a *ToDict* (with '_var' appended) for simplicity.

Example:

```
>>> import torch
>>> from torchbearer import metrics

>>> @metrics.var
... @metrics.lambda_metric('my_metric')
... def metric(y_pred, y_true):
...     return y_pred + y_true
...
>>> metric.reset({})
>>> metric.process({'y_pred':torch.Tensor([2]), 'y_true':torch.Tensor([2])}) # 4
{}
>>> metric.process({'y_pred':torch.Tensor([3]), 'y_true':torch.Tensor([3])}) # 6
{}
>>> metric.process({'y_pred':torch.Tensor([4]), 'y_true':torch.Tensor([4])}) # 8
{}
>>> '%.4f' % metric.process_final()['my_metric_var']
'4.0000'
```

Parameters

- **clazz** – The class to *decorate*
- **unbiased** (*bool*) – See *Var*
- **dim** (*int, tuple*) – See *Var*

Returns A *MetricTree* with a *Var* appended or a wrapper class that extends *MetricTree*

14.3 Metric Wrappers

Metric wrappers are classes which wrap instances of *Metric* or, in the case of *EpochLambda* and *BatchLambda*, functions. Typically, these should **not** be used directly (although this is entirely possible), but via the *decorator API*.

class torchbearer.metrics.wrappers.**BatchLambda**(*name*, *metric_function*)

A metric which returns the output of the given function on each batch.

Parameters

- **name** (*str*) – The name of the metric.
- **metric_function** (*func*) – A metric function('y_pred', 'y_true') to wrap.

process(**args*)

Return the output of the wrapped function.

Parameters **args** – The `torchbearer.Trial` state.

Returns The value of the metric function('y_pred', 'y_true').

class torchbearer.metrics.wrappers.**EpochLambda**(*name*, *metric_function*, *running=True*, *step_size=50*)

A metric wrapper which computes the given function for concatenated values of 'y_true' and 'y_pred' each epoch. Can be used as a running metric which computes the function for batches of outputs with a given step size during training.

Parameters

- **name** (*str*) – The name of the metric.
- **metric_function** (*func*) – The function('y_pred', 'y_true') to use as the metric.
- **running** (*bool*) – True if this should act as a running metric.
- **step_size** (*int*) – Step size to use between calls if running=True.

process_final_train(**args*)

Evaluate the function with the aggregated outputs.

Returns The result of the function.

process_final_validate(**args*)

Evaluate the function with the aggregated outputs.

Returns The result of the function.

process_train(**args*)

Concatenate the 'y_true' and 'y_pred' from the state along the 0 dimension, this must be the batch dimension. If this is a running metric, evaluates the function every number of steps.

Parameters **args** – The `torchbearer.Trial` state.

Returns The current running result.

process_validate(**args*)

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

Parameters **args** – The `torchbearer.Trial` state.

reset(*state*)

Reset the 'y_true' and 'y_pred' caches.

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

class torchbearer.metrics.wrappers.**ToDict**(*metric*)

The *ToDict* class is an *AdvancedMetric* which will put output from the inner *Metric* in a dict (mapping metric name to value) before returning. When in *eval* mode, 'val_' will be prepended to the metric name.

Example:

```
>>> from torchbearer import metrics

>>> @metrics.lambda_metric('my_metric')
... def my_metric(y_pred, y_true):
...     return y_pred + y_true
...
>>> metric = metrics.ToDict(my_metric().build())
>>> metric.process({'y_pred': 4, 'y_true': 5})
{'my_metric': 9}
>>> metric.eval()
>>> metric.process({'y_pred': 4, 'y_true': 5})
{'val_my_metric': 9}
```

Parameters **metric** (Metric) – The *Metric* instance to *wrap*.

eval(*data_key=None*)
Put the metric in eval mode.

Parameters **data_key** (StateKey) – The torchbearer data_key, if used

process_final_train(**args*)
Process the given state and return the final metric value for a training iteration.

Returns The final metric value for a training iteration.

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

Returns The final metric value for a validation iteration.

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

Returns The metric value for a training iteration.

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

Returns The metric value for a validation iteration.

reset(*state*)
Reset the metric, called before the start of an epoch.

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

train()
Put the metric in train mode.

14.4 Metric Aggregators

Aggregators are a special kind of *Metric* which takes as input, the output from a previous metric or metrics. As a result, via a *MetricTree*, a series of aggregators can collect statistics such as Mean or Standard Deviation without needing to compute the underlying metric multiple times. This can, however, make the aggregators complex to use. It is therefore typically better to use the *decorator API*.

class torchbearer.metrics.aggregators.**Mean**(*name*, *dim=None*)
Metric aggregator which calculates the mean of process outputs between calls to reset.

Parameters

- **name** (*str*) – The name of this metric.
- **dim** (*int, tuple*) – The dimension(s) on which to perform the mean. If left as None, this will mean over the whole Tensor

process(**args*)
Add the input to the rolling sum. Input must be a torch tensor.

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

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

Returns The mean of the metric values since the last call to reset.

reset(*state*)
Reset the running count and total.

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

class torchbearer.metrics.aggregators.**RunningMean**(*name*, *batch_size=50*, *step_size=10*, *dim=None*)

A *RunningMetric* which outputs the running mean of its input tensors over the course of an epoch.

Parameters

- **name** (*str*) – The name of this running mean.
- **batch_size** (*int*) – The size of the deque to store of previous results.
- **step_size** (*int*) – The number of iterations between aggregations.
- **dim** (*int, tuple*) – The dimension(s) on which to perform the mean. If left as None, this will mean over the whole Tensor

class torchbearer.metrics.aggregators.**RunningMetric**(*name*, *batch_size=50*, *step_size=10*)

A metric which aggregates batches of results and presents a method to periodically process these into a value.

Note: Running metrics only provide output during training.

Parameters

- **name** (*str*) – The name of the metric.
- **batch_size** (*int*) – The size of the deque to store of previous results.
- **step_size** (*int*) – The number of iterations between aggregations.

process_train(**args*)

Add the current metric value to the cache and call '_step' is needed.

Parameters **args** – The output of some *Metric*

Returns The current metric value.

reset(*state*)

Reset the step counter. Does not clear the cache.

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

class torchbearer.metrics.aggregators.**Std**(*name*, *unbiased=True*, *dim=None*)

Metric aggregator which calculates the **sample** standard deviation of process outputs between calls to reset. Optionally calculate the population std if `unbiased = False`.

Parameters

- **name** (*str*) – The name of this metric.
- **unbiased** (*bool*) – If True (default), calculates the sample standard deviation, else, the population standard deviation
- **dim** (*int, tuple*) – The dimension(s) on which to compute the std. If left as None, this will operate over the whole Tensor

process_final(**args*)

Compute and return the final standard deviation.

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

class torchbearer.metrics.aggregators.**Var**(*name*, *unbiased=True*, *dim=None*)
Metric aggregator which calculates the **sample** variance of process outputs between calls to reset. Optionally calculate the population variance if `unbiased = False`.

Parameters

- **name** (*str*) – The name of this metric.
- **unbiased** (*bool*) – If True (default), calculates the sample variance, else, the population variance
- **dim** (*int, tuple*) – The dimension(s) on which to compute the std. If left as None, this will operate over the whole Tensor

process(**args*)
Compute values required for the variance from the input. The input should be a torch Tensor. The sum and sum of squares will be computed over the provided dimension.

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

process_final(**args*)
Compute and return the final variance.

Returns The variance of each observation since the last reset call.

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

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

14.5 Base Metrics

Base metrics are the base classes which represent the metrics supplied with torchbearer. They all use the *default_for_key()* decorator so that they can be accessed in the call to `torchbearer.Model` via the following strings:

- '*acc*' or '*accuracy*': The *DefaultAccuracy* metric
- '*binary_acc*' or '*binary_accuracy*': The *BinaryAccuracy* metric
- '*cat_acc*' or '*cat_accuracy*': The *CategoricalAccuracy* metric
- '*top_5_acc*' or '*top_5_accuracy*': The *TopKCategoricalAccuracy* metric
- '*top_10_acc*' or '*top_10_accuracy*': The *TopKCategoricalAccuracy* metric with k=10
- '*mse*': The *MeanSquaredError* metric
- '*loss*': The *Loss* metric
- '*epoch*': The *Epoch* metric
- '*lr*': The `LR` metric
- '*roc_auc*' or '*roc_auc_score*': The *RocAucScore* metric

class torchbearer.metrics.default.**DefaultAccuracy**
The default accuracy metric loads in a different accuracy metric depending on the loss function or criterion in use at the start of training. Default for keys: *acc*, *accuracy*. The following bindings are in place for both nn and functional variants:

- cross entropy loss -> *CategoricalAccuracy* [DEFAULT]

- nll loss -> *CategoricalAccuracy*
- mse loss -> *MeanSquaredError*
- bce loss -> *BinaryAccuracy*
- bce loss with logits -> *BinaryAccuracy*

eval(*data_key=None*)
Put the metric in eval mode during model validation.

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

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

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

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

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

Returns None or the value of the metric for this epoch

reset(*state*)
Reset the metric, called before the start of an epoch.

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

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

class torchbearer.metrics.primitives.**BinaryAccuracy**
Binary accuracy metric. Uses torch.eq to compare predictions to targets. Decorated with a mean and running_mean. Default for key: 'binary_acc'.

Parameters

- **pred_key** (StateKey) – The key in state which holds the predicted values
- **target_key** (StateKey) – The key in state which holds the target values
- **threshold** (*float*) – value between 0 and 1 to use as a threshold when binarizing predictions and targets

class torchbearer.metrics.primitives.**CategoricalAccuracy**(*ignore_index=-100*)
Categorical accuracy metric. Uses torch.max to determine predictions and compares to targets. Decorated with a mean, running_mean and std. Default for key: 'cat_acc'

Parameters

- **pred_key** (StateKey) – The key in state which holds the predicted values
- **target_key** (StateKey) – The key in state which holds the target values
- **ignore_index** (*int*) – Specifies a target value that is ignored and does not contribute to the metric output. See https://pytorch.org/docs/stable/nn.html#crossentropyloss

class torchbearer.metrics.primitives.**TopKCategoricalAccuracy**(*k=5*, *ignore_index=-100*)
Top K Categorical accuracy metric. Uses torch.topk to determine the top k predictions and compares to targets. Decorated with a mean, running_mean and std. Default for keys: 'top_5_acc', 'top_10_acc'.

Parameters

- **pred_key** (StateKey) – The key in state which holds the predicted values
- **target_key** (StateKey) – The key in state which holds the target values
- **ignore_index** (*int*) – Specifies a target value that is ignored and does not contribute to the metric output. See https://pytorch.org/docs/stable/nn.html#crossentropyloss

class torchbearer.metrics.primitives.**MeanSquaredError**

Mean squared error metric. Computes the pixelwise squared error which is then averaged with decorators. Decorated with a mean and running_mean. Default for key: 'mse'.

Parameters

- **pred_key** (StateKey) – The key in state which holds the predicted values
- **target_key** (StateKey) – The key in state which holds the target values

class torchbearer.metrics.primitives.**Loss**

Simply returns the 'loss' value from the model state. Decorated with a mean, running_mean and std. Default for key: 'loss'.

State Requirements:

- *torchbearer.state.LOSS*: This key should map to the loss for the current batch

class torchbearer.metrics.primitives.**Epoch**

Returns the 'epoch' from the model state. Default for key: 'epoch'.

State Requirements:

- *torchbearer.state.EPOCH*: This key should map to the number of the current epoch

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

Area Under ROC curve metric. Default for keys: 'roc_auc', 'roc_auc_score'.

Note: Requires `sklearn.metrics`.

Parameters

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

14.6 Timer

class torchbearer.metrics.timer.**TimerMetric**(*time_keys=()*)

get_timings()

on_backward(*state*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

reset(*state*)
Reset the metric, called before the start of an epoch.

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

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

CHAPTER 15

torchbearer.variational

15.1 Distributions

The distributions module is an extension of the *torch.distributions* package intended to facilitate implementations required for specific variational approaches through the *SimpleDistribution* class. Generally, using a `torch.distributions.Distribution` object should be preferred over a *SimpleDistribution*, for better argument validation and more complete implementations. However, if you need to implement something new for a specific variational approach, then a *SimpleDistribution* may be more forgiving. Furthermore, you may find it easier to understand the function of the implementations here.

class torchbearer.variational.distributions.**SimpleDistribution**(*batch_shape=<sphinx.ext.autodoc.impor object>*, *event_shape=<sphinx.ext.autodoc.impor object>*)

Abstract base class for a simple distribution which only implements rsample and log_prob. If the log_prob function is not differentiable with respect to the distribution parameters or the given value, then this should be mentioned in the documentation.

arg_constraints

cdf(*value*)

entropy()

enumerate_support(*expand=True*)

expand(*batch_shape*, *_instance=None*)

has_rsample = True

icdf(*value*)

log_prob(*value*)

Returns the log of the probability density/mass function evaluated at *value*. :param value: Value at which to evaluate log probabilty :type value: torch.Tensor, Number

mean

rsample(*sample_shape=<sphinx.ext.autodoc.importer._MockObject object>*)
Returns a reparameterized sample or batch of reparameterized samples if the distribution parameters are batched.

support

variance

class torchbearer.variational.distributions.**SimpleExponential**(*lograte*)
The SimpleExponential class is a *SimpleDistribution* which implements a straight forward Exponential distribution with the given lograte. This performs significantly fewer checks than *torch.distributions.Exponential* , but should be sufficient for the purpose of implementing a VAE. By using a lograte, the log_prob can be computed in a stable fashion, without taking a logarithm.

Parameters **lograte** (*torch.Tensor, Number*) – The natural log of the rate of the distribution, numbers will be cast to tensors

log_prob(*value*)
Calculates the log probability that the given value was drawn from this distribution. The log_prob for this distribution is fully differentiable and has stable gradient since we use the lograte here.

Parameters **value** (*torch.Tensor, Number*) – The sampled value

Returns The log probability that the given value was drawn from this distribution

rsample(*sample_shape=<sphinx.ext.autodoc.importer._MockObject object>*)
Simple rsample for an Exponential distribution.

Parameters **sample_shape** (*torch.Size, tuple*) – Shape of the sample (per lograte given)

Returns A reparameterized sample with gradient with respect to the distribution parameters

class torchbearer.variational.distributions.**SimpleNormal**(*mu*, *logvar*)
The SimpleNormal class is a *SimpleDistribution* which implements a straight forward Normal / Gaussian distribution. This performs significantly fewer checks than *torch.distributions.Normal*, but should be sufficient for the purpose of implementing a VAE.

Parameters

- **mu** (*torch.Tensor, Number*) – The mean of the distribution, numbers will be cast to tensors
- **logvar** (*torch.Tensor, Number*) – The log variance of the distribution, numbers will be cast to tensors

log_prob(*value*)
Calculates the log probability that the given value was drawn from this distribution. Since the density of a Gaussian is differentiable, this function is differentiable.

Parameters **value** (*torch.Tensor, Number*) – The sampled value

Returns The log probability that the given value was drawn from this distribution

rsample(*sample_shape=<sphinx.ext.autodoc.importer._MockObject object>*)
Simple rsample for a Normal distribution.

Parameters **sample_shape** (*torch.Size, tuple*) – Shape of the sample (per mean / variance given)

Returns A reparameterized sample with gradient with respect to the distribution parameters

class torchbearer.variational.distributions.**SimpleUniform**(*low*, *high*)

The SimpleUniform class is a *SimpleDistribution* which implements a straight forward Uniform distribution in the interval `[low, high)`. This performs significantly fewer checks than *torch.distributions.Uniform*, but should be sufficient for the purpose of implementing a VAE.

Parameters

- **low** (*torch.Tensor, Number*) – The lower range of the distribution (inclusive), numbers will be cast to tensors
- **high** (*torch.Tensor, Number*) – The upper range of the distribution (exclusive), numbers will be cast to tensors

log_prob(*value*)

Calculates the log probability that the given value was drawn from this distribution. Since this distribution is uniform, the log probability is `-log(high - low)` for all values in the range `[low, high)` and -inf elsewhere. This function is therefore only piecewise differentiable.

Parameters **value** (*torch.Tensor, Number*) – The sampled value

Returns The log probability that the given value was drawn from this distribution

rsample(*sample_shape=<sphinx.ext.autodoc.importer._MockObject object>*)

Simple rsample for a Uniform distribution.

Parameters **sample_shape** (*torch.Size, tuple*) – Shape of the sample (per low / high given)

Returns A reparameterized sample with gradient with respect to the distribution parameters

class torchbearer.variational.distributions.**SimpleWeibull**(*l*, *k*)

The SimpleWeibull class is a *SimpleDistribution* which implements a straight forward Weibull distribution. This performs significantly fewer checks than *torch.distributions.Weibull*, but should be sufficient for the purpose of implementing a VAE.

```
@article{squires2019a,
title={A Variational Autoencoder for Probabilistic Non-Negative Matrix␣
↪Factorisation},
author={Steven Squires and Adam Prugel-Bennett and Mahesan Niranjan},
year={2019}
}
```

Parameters

- **l** (*torch.Tensor, Number*) – The scale parameter of the distribution, numbers will be cast to tensors
- **k** (*torch.Tensor, Number*) – The shape parameter of the distribution, numbers will be cast to tensors

log_prob(*value*)

Calculates the log probability that the given value was drawn from this distribution. This function is differentiable and its log probability is -inf for values less than 0.

Parameters **value** (*torch.Tensor, Number*) – The sampled value

Returns The log probability that the given value was drawn from this distribution

rsample(*sample_shape=<sphinx.ext.autodoc.importer._MockObject object>*)

Simple rsample for a Weibull distribution.

Parameters **sample_shape** (*torch.Size, tuple*) – Shape of the sample (per k / lambda given)

Returns A reparameterized sample with gradient with respect to the distribution parameters

15.2 Divergences

class torchbearer.variational.divergence.**DivergenceBase**(*keys*, *state_key=None*)

The *DivergenceBase* class is an abstract base class which defines a series of useful methods for dealing with divergences. The keys dict given on init is used to map objects in state to kwargs in the compute function.

Parameters

- **keys** (*dict*) – Dictionary which maps kwarg names to *StateKey* objects. When *compute()* is called, the given kwargs are mapped to their associated values in state.
- **state_key** – If not None, the value outputted by *compute()* is stored in state with the given key.

compute(***kwargs*)

Compute the loss with the given kwargs defined in the constructor.

Parameters **kwargs** – The bound kwargs, taken from state with the keys given in the constructor

Returns The calculated divergence as a two dimensional tensor (batch, distribution dimensions)

loss(*state*)

on_criterion(*state*)

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

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

on_criterion_validation(*state*)

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

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

with_beta(*beta*)

Multiply the divergence by the given beta, as introduced by beta-vae.

```
@article{higgins2016beta,
  title={beta-vae: Learning basic visual concepts with a constrained␣
↪variational framework},
  author={Higgins, Irina and Matthey, Loic and Pal, Arka and Burgess,␣
↪Christopher and Glorot, Xavier and Botvinick, Matthew and Mohamed, Shakir␣
↪and Lerchner, Alexander},
  year={2016}
}
```

Parameters **beta** (*float*) – The beta (> 1) to multiply by.

Returns self

Return type Divergence

with_linear_capacity(*min_c=0*, *max_c=25*, *steps=100000*, *gamma=1000*)
Limit divergence by capacity, linearly increased from min_c to max_c for steps, as introduced in *Understanding disentangling in beta-VAE*.

```
@article{burgess2018understanding,
  title={Understanding disentangling in beta-vae},
  author={Burgess, Christopher P and Higgins, Irina and Pal, Arka and Matthey,
↪ Loic and Watters, Nick and Desjardins, Guillaume and Lerchner, Alexander},
  journal={arXiv preprint arXiv:1804.03599},
  year={2018}
}
```

Parameters

- **min_c** (*float*) – Minimum capacity
- **max_c** (*float*) – Maximum capacity
- **steps** (*int*) – Number of steps to increase over
- **gamma** (*float*) – Multiplicative gamma, usually a high number

Returns self

Return type Divergence

with_post_function(*post_fcn*)
Register the given post function, to be applied after to loss after reduction.

Parameters **post_fcn** – A function of loss which applies some operation (e.g. multiplying by beta)

Returns self

Return type Divergence

with_reduction(*reduction_fcn*)
Override the reduction operation with the given function, use this if your divergence doesn't output a two dimensional tensor.

Parameters **reduction_fcn** – The function to be applied to the divergence output and return a single value

Returns self

Return type Divergence

with_sum_mean_reduction()
Override the reduction function to take a sum over dimension one and a mean over dimension zero. (default)

Returns self

Return type Divergence

with_sum_sum_reduction()
Override the reduction function to take a sum over all dimensions.

Returns self

Return type Divergence

class torchbearer.variational.divergence.**SimpleExponentialSimpleExponentialKL**(*input_key*, *target_key*, *state_key=None*)

A KL divergence between two SimpleExponential (or similar) distributions.

Note:

The distribution object must have lograte attribute

Args: input_key: *StateKey* instance which will be mapped to the input distribution object. target_key: *StateKey* instance which will be mapped to the target distribution object. state_key: If not None, the value outputted by *compute()* is stored in state with the given key.

---

compute(*input*, *target*)

Compute the loss with the given kwargs defined in the constructor.

Parameters **kwargs** – The bound kwargs, taken from state with the keys given in the constructor

Returns The calculated divergence as a two dimensional tensor (batch, distribution dimensions)

class torchbearer.variational.divergence.**SimpleNormalSimpleNormalKL**(*input_key*, *target_key*, *state_key=None*)

A KL divergence between two SimpleNormal (or similar) distributions.

Note: The distribution objects must have mu and logvar attributes

---

Parameters

- **input_key** – *StateKey* instance which will be mapped to the input distribution object.
- **target_key** – *StateKey* instance which will be mapped to the target distribution object.
- **state_key** – If not None, the value outputted by *compute()* is stored in state with the given key.

compute(*input*, *target*)

Compute the loss with the given kwargs defined in the constructor.

Parameters **kwargs** – The bound kwargs, taken from state with the keys given in the constructor

Returns The calculated divergence as a two dimensional tensor (batch, distribution dimensions)

class torchbearer.variational.divergence.**SimpleNormalUnitNormalKL**(*input_key*, *state_key=None*)

A KL divergence between a SimpleNormal (or similar) instance and a fixed unit normal (N[0, 1]) target.

Note: The distribution object must have mu and logvar attributes

---

Parameters

- **input_key** – *StateKey* instance which will be mapped to the distribution object.
- **state_key** – If not None, the value outputted by *compute()* is stored in state with the given key.

compute(*input*)
Compute the loss with the given kwargs defined in the constructor.

Parameters **kwargs** – The bound kwargs, taken from state with the keys given in the constructor

Returns The calculated divergence as a two dimensional tensor (batch, distribution dimensions)

class torchbearer.variational.divergence.**SimpleWeibullSimpleWeibullKL**(*input_key*, *target_key*, *state_key=None*)

A KL divergence between two SimpleWeibull (or similar) distributions.

Note:

The distribution object must have lambda (scale) and k (shape) attributes

```
@article{DBLP:journals/corr/Bauckhage14,
  author    = {Christian Bauckhage},
  title     = {Computing the Kullback-Leibler Divergence between two Generalized
               Gamma Distributions},
  journal   = {CoRR},
  volume    = {abs/1401.6853},
  year      = {2014}
}
```

Args: input_key: *StateKey* instance which will be mapped to the input distribution object. target_key: *StateKey* instance which will be mapped to the target distribution object. state_key: If not None, the value outputted by *compute()* is stored in state with the given key.

---

compute(*input*, *target*)
Compute the loss with the given kwargs defined in the constructor.

Parameters **kwargs** – The bound kwargs, taken from state with the keys given in the constructor

Returns The calculated divergence as a two dimensional tensor (batch, distribution dimensions)

15.3 Auto-Encoding

class torchbearer.variational.auto_encoder.**AutoEncoderBase**(*latent_dims*)

decode(*sample*, *state=None*)
Decode the given latent space sample batch to images.

Parameters

- **sample** – The latent space samples
- **state** – The trial state

Returns Decoded images

encode(*x*, *state=None*)
Encode the given batch of images and return latent space sample for each.

Parameters

- **x** – Batch of images to encode
- **state** – The trial state

Returns Encoded samples / tuple of samples for different spaces

forward(*x*, *state=None*)
Encode then decode the inputs, returning the result. Also binds the target as the input images in state.

Parameters

- **x** – Model input batch
- **state** – The trial state

Returns Auto-Encoded images

15.4 Datasets

class torchbearer.variational.datasets.**CelebA**(*root*, *transform=None*, *target_transform=None*)

class torchbearer.variational.datasets.**CelebA_HQ**(*root*, *as_npy=False*, *transform=None*)

static npy_loader(*path*)

class torchbearer.variational.datasets.**SimpleImageFolder**(*root*, *loader=None*, *extensions=None*, *transform=None*, *target_transform=None*)

class torchbearer.variational.datasets.**dSprites**(*root*, *download=False*, *transform=None*)

download()

get_img_by_latent(*latent_code*)
Returns the image defined by the latent code

Parameters **latent_code** (`list` of `int`) – Latent code of length 6 defining each generative factor

Returns Image defined by given code

load_data()

torchbearer.variational.datasets.**make_dataset**(*dir*, *extensions*)

15.5 Visualisation

class torchbearer.variational.visualisation.**CodePathWalker**(*num_steps*, *p1*, *p2*)

vis(*state*)
Create the tensor of images to be displayed

class torchbearer.variational.visualisation.**ImagePathWalker**(*num_steps*, *im1*, *im2*)

vis(*state*)
Create the tensor of images to be displayed

class torchbearer.variational.visualisation.**LatentWalker**(*same_image*, *row_size*)

for_data(*data_key*)

Parameters **data_key** (*StateKey*) – State key which will contain data to act on

Returns self

Return type *LatentWalker*

for_space(*space_id*)
Sets the ID for which latent space to vary when model outputs [latent_space_0, latent_space_1, . . .]

Parameters **space_id** (*int*) – ID of the latent space to vary

Returns self

Return type *LatentWalker*

on_train()
Sets the walker to run during training

Returns self

Return type *LatentWalker*

on_val()
Sets the walker to run during validation

Returns self

Return type *LatentWalker*

to_file(*file*)

Parameters **file** (*string, pathlib.Path object or file object*) – File in which result is saved

Returns self

Return type *LatentWalker*

to_key(*state_key*)

Parameters **state_key** (*StateKey*) – State key under which to store result

Returns self

Return type *LatentWalker*

vis(*state*)
Create the tensor of images to be displayed

class torchbearer.variational.visualisation.**LinSpaceWalker**(*lin_start=-1, lin_end=1, lin_steps=8, dims_to_walk=[0], zero_init=False, same_image=False*)

vis(*state*)
Create the tensor of images to be displayed

class torchbearer.variational.visualisation.**RandomWalker**(*var=1, num_images=32, uniform=False, row_size=8*)

vis(*state*)
Create the tensor of images to be displayed

class torchbearer.variational.visualisation.**ReconstructionViewer**(*row_size=8, recon_key=y_pred*)

vis(*state*)
Create the tensor of images to be displayed

CHAPTER 16

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z