torchbearer Documentation

Release 0.3.0

Ethan Harris, Matthew Painter and Jonathon Hare

Feb 28, 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:

```
def super(_, obj):
    return old_super(obj.__class__, obj)


@default_for_key('binary_accuracy')
```

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:

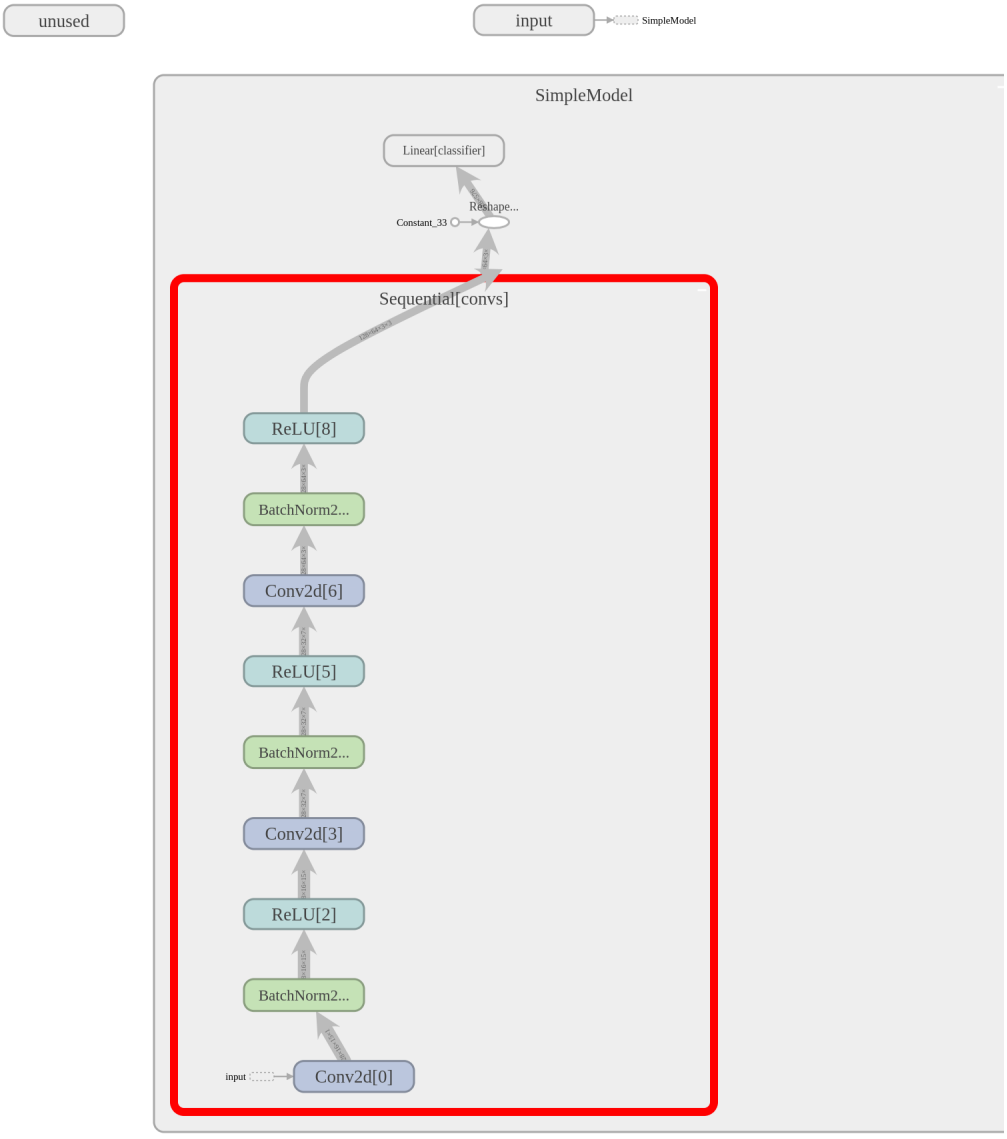
```
from torchbearer import Trial
from torchbearer.callbacks import TensorBoard
```

(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:



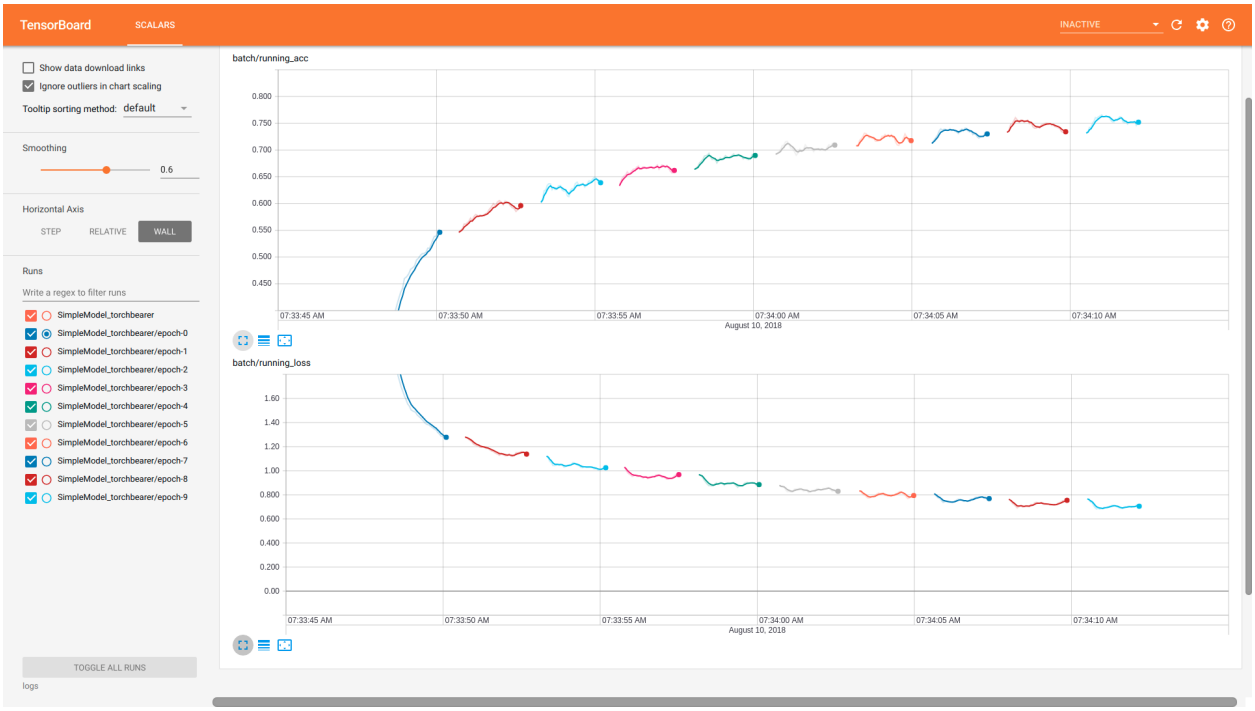
The dynamic graph construction does introduce some weirdness, however, this is about as good as model graphs typically get.

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:

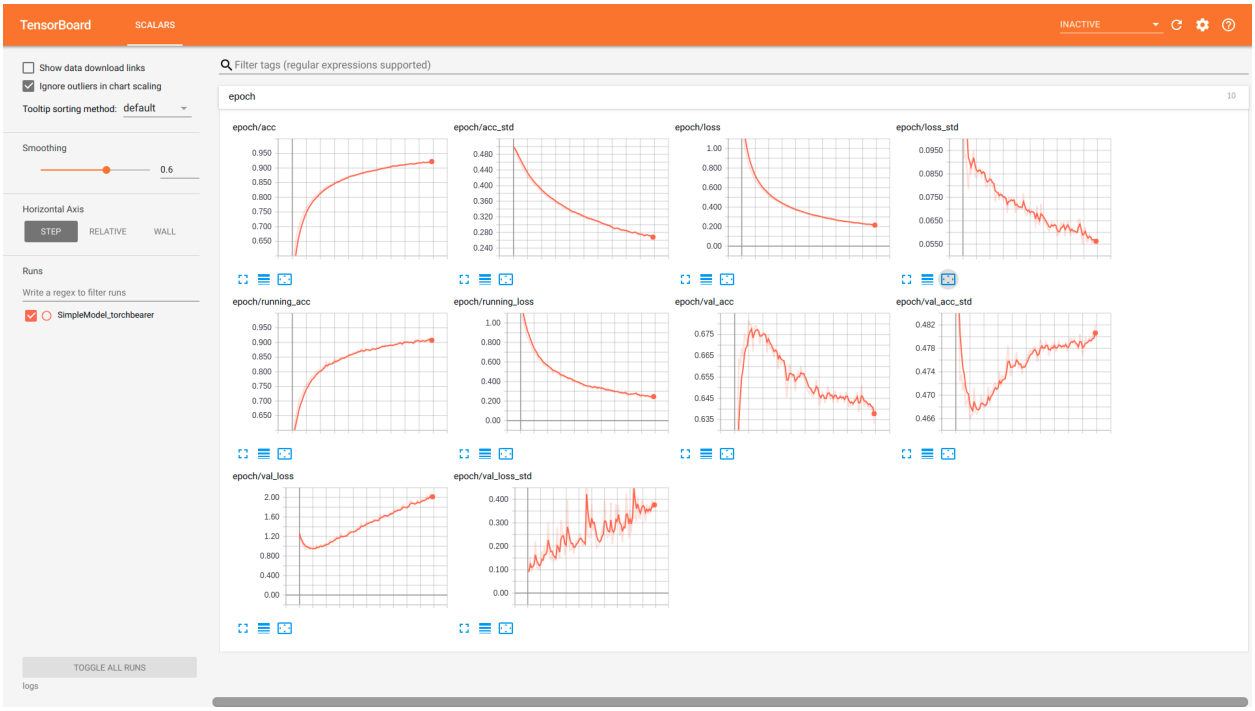


3.4 Logging Epoch Metrics

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)
```

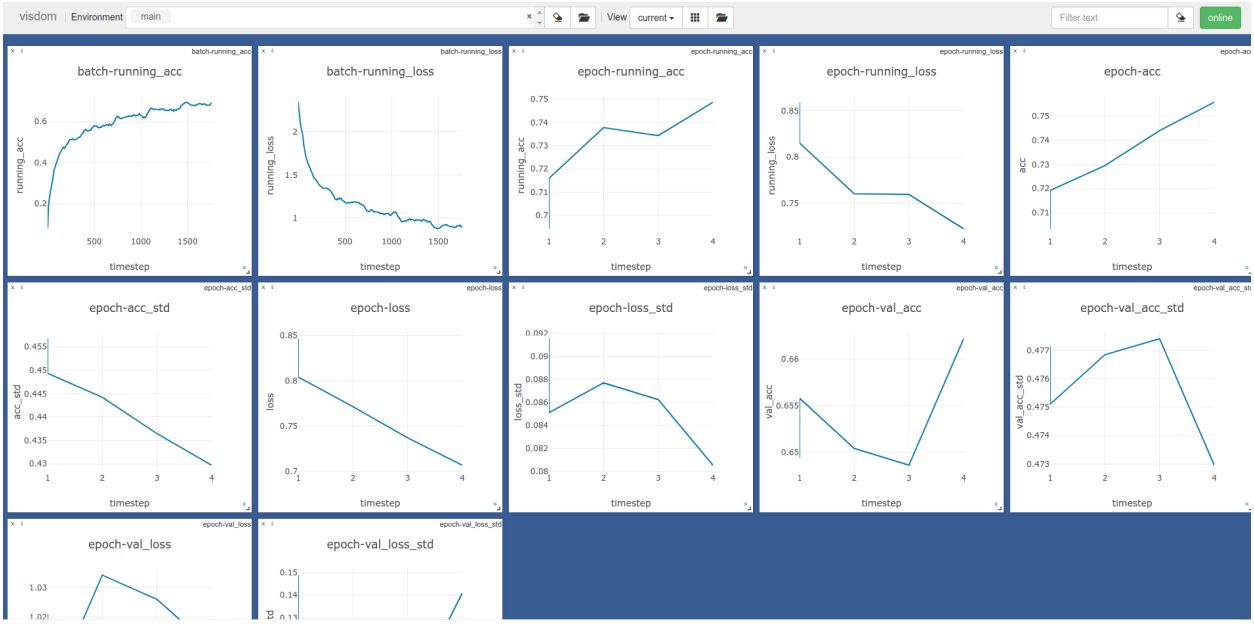
If your visdom server is running then you should see something similar to the figure below:

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'
```

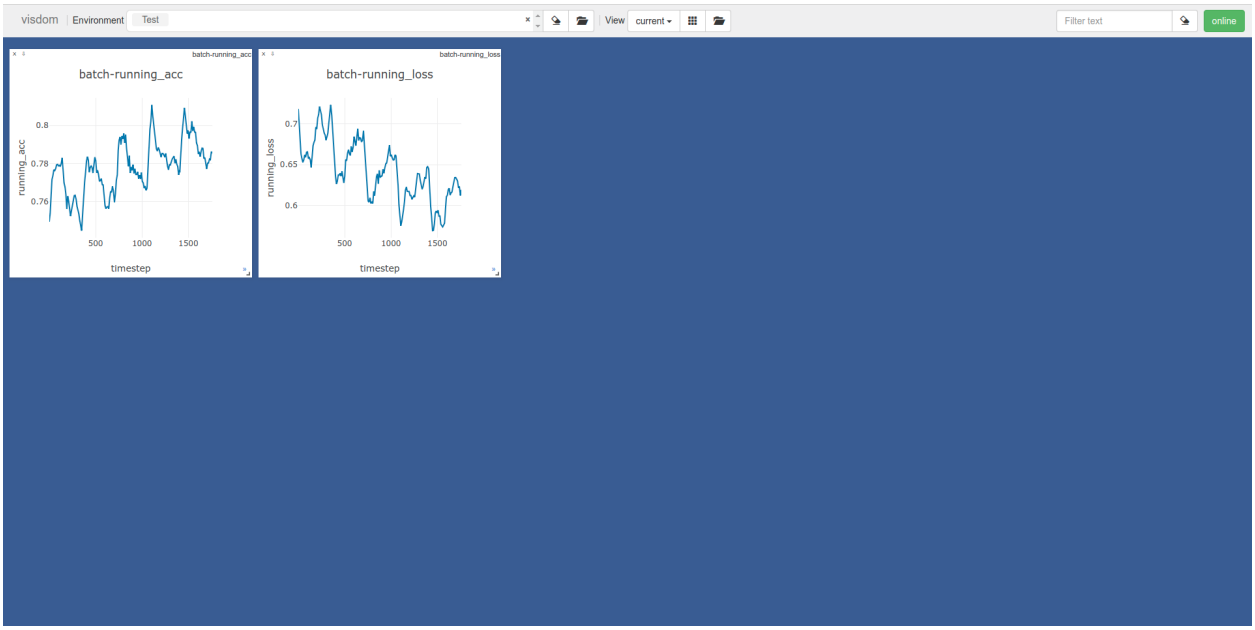
Running another fit call, we can see we are now logging to the "Test" environment.

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.

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

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

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

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

```
os.makedirs('./data/mnist', exist_ok=True)
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 and combine them into a model that performs a single forward pass.

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

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

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

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

7.3 Loss

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

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

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

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. We can then create metrics from these by decorating simple state fetcher metrics.

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

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

7.5 Training

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

```
torchbearertrial = tb.Trial(model, optim, criterion=None, metrics=['loss', g_loss(),␣
↪d_loss()],
                            callbacks=[loss_callback, saver_callback])
torchbearertrial.with_train_generator(dataloader)
torchbearertrial.to(device)
torchbearertrial.run(epochs=200)
```

7.6 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 visulisation.

```
batch = torch.randn(25, latent_dim).to(device)
@callbacks.on_step_training
def saver_callback(state):
    batches_done = state[tb.EPOCH] * len(state[tb.GENERATOR]) + state[tb.BATCH]
    if batches_done % sample_interval == 0:
        samples = state[tb.MODEL].generator(batch)
        save_image(samples, 'images/%d.png' % batches_done, nrow=5, normalize=True)
```

Here is a Gif created from the saved images.

7.7 Source Code

The source code for the example is given below:

Download Python source code: gan.py

CHAPTER 8

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.

8.1 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()
```

8.2 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
```

8.3 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.

8.4 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])

8.5 Source Code

The source code for the example is given below:

Download Python source code: basic_opt.py

CHAPTER 9

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.

9.1 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:

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

9.2 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))
```

9.3 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)
```

9.4 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()
```

9.5 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)
```

9.6 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?

9.7 Source Code

The source code for the example is given below:

Download Python source code: svm_linear.py

CHAPTER 10

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.

10.1 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()
```

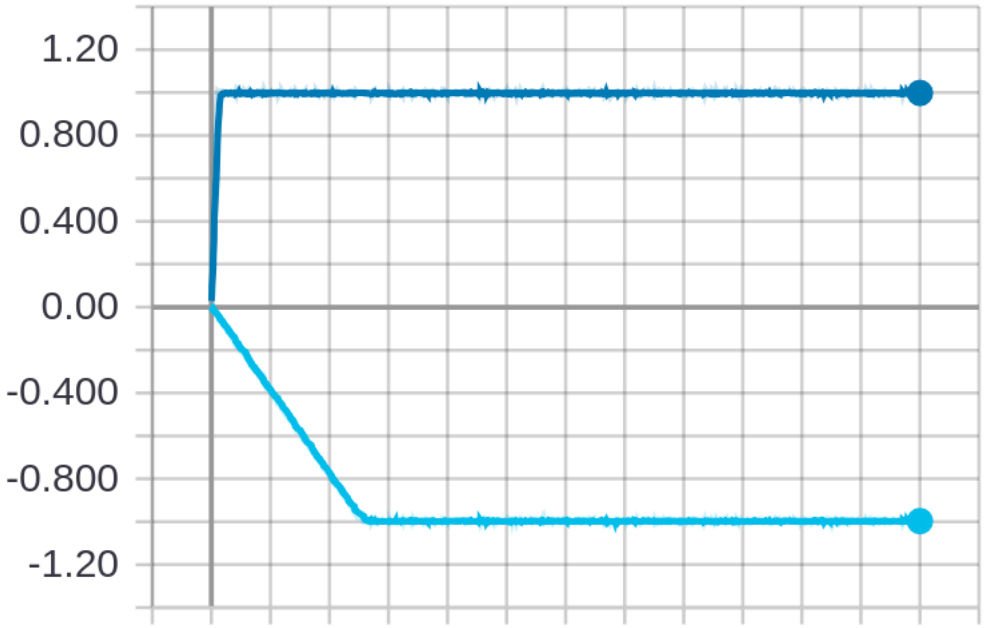
Note that we have logged to TensorBoard here and after completion, running `tensorboard --logdir logs` and navigating to localhost:6006, we can see a graph like the one in Figure 1 from the paper, where the top line is with ADAM and the bottom with AMSGrad:

10.2 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:

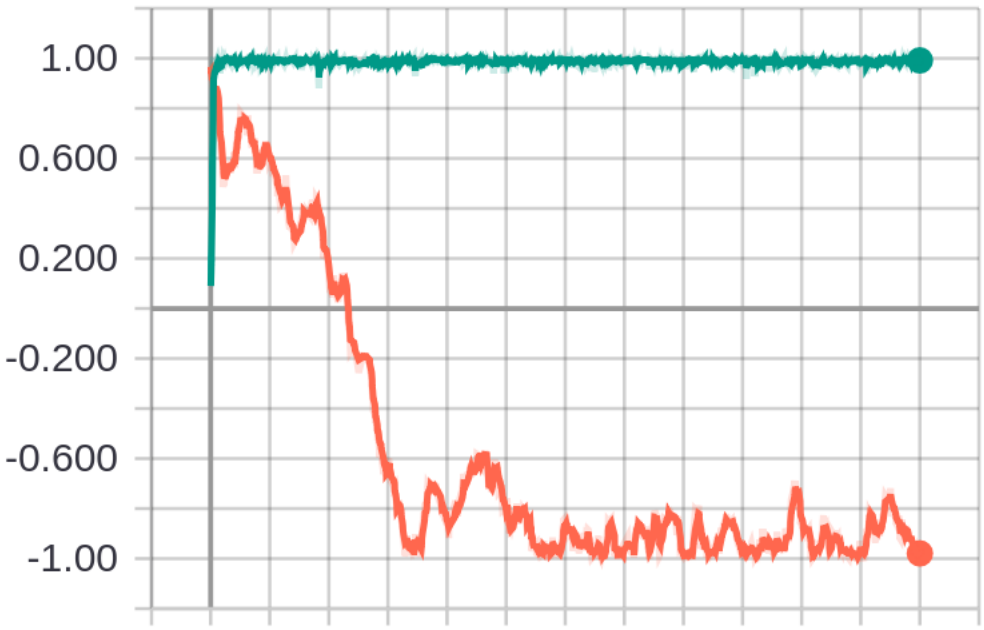
```
model = Stochastic()
optim = torch.optim.Adam(model.parameters(), lr=0.001, betas=[0.9, 0.99])
tbtrial = tb.Trial(model, optim, loss, [est()], callbacks=[greedy_update,␣
↪TensorBoard(comment='adam', write_graph=False, write_batch_metrics=True, write_
↪epoch_metrics=False)])
tbtrial.for_train_steps(training_steps).run()

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:



10.3 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?

10.4 Source Code

The source code for this example can be downloaded below:

`Download Python source code: amsgrad.py`

CHAPTER 11

torchbearer

CHAPTER 12

torchbearer.callbacks

CHAPTER 13

torchbearer.metrics

CHAPTER 14

torchbearer.variational

CHAPTER 15

Indices and tables

- genindex
- modindex
- search