TorchFusion Documentation

Release 0.2.0

"John Olafenwa" "Moses Olafenwa"

Nov 23, 2018

Training Neural Networks with TorchFusion:

TorchFusion is built to accelerate research and developement of modern AI systems. It is based on **PyTorch** and allows unimpeded access to all of PyTorch's features. In creating **TorchFusion**, our goal is to build a deep learning framework that can easily support complex research projects while being incredibly simple enough to allow researchers focus more on research ideas rather than dealing with framework complexity. To achieve this, **TorchFusion** is built with multiple layers of abstractions, allowing researchers to remain productive while doing research projects needing varying levels of complexity. At all levels of abstraction, **TorchFusion** allows you to seamlessly use all standard **PyTorch** code and functions including its support libraries. The entire framework is highly decoupled allowing you to take advantage of various features even without using TorchFusion's trainers.

TorchFusion is a project developed by John Olafenwa and Moses Olafenwa, the AI Commons team.

The Official GitHub Repository of **TorchFusion** is https://github.com/johnolafenwa/TorchFusion

Installing TorchFusion

- **Install PyTorch** 0.4.1 or higher : visit pytorch.org

Install Torchfusion

```
pip3 install --upgrade torchfusion
```

CHAPTER 1

Hello FASHION MNIST!

TorchFusion makes data loading, network definition and training very easy. As you will see in this tutorial. We shall be training a basic pytorch model on the Fashion MNIST dataset.

FASHION MNIST DESCRIPTION

MNIST has been over-explored, state-of-the-art on MNIST doesn't make much sense with over 99% already achieved. Fashion MNIST provides a more challenging version of the MNIST dataset. It contains 10 classes of grayscale diagrams of fashion items. It is exactly the same size, dimension and format as MNIST, but it is more challenging, hence, it is provides a dataset that is both fast to train and yet challenging enough to benchmark new models. Below are samples from the FashionMNIST dataset.

To learn more visit. Fashion MNIST

Import Classes

```
from torchfusion.layers import *
from torchfusion.datasets import *
from torchfusion.metrics import *
import torch.nn as nn
import torch.cuda as cuda
from torch.optim import Adam
from torchfusion.learners import StandardLearner
```

Load the dataset

```
train_loader = fashionmnist_loader(size=28,batch_size=32)
test_loader = fashionmnist_loader(size=28,train=False,batch_size=32)
```

If you have used PyTorch before, you will notice just how simpler the data loading process is, this function still allows you to specify custom transformations. By default, TorchFusion loaders will normalize the images to range between -1 to 1, you can control the default normalization using the mean and std args.

Define the model

```
model = nn.Sequential(
    Flatten(),
    Linear(784,100),
    Swish(),
    Linear(100,100),
    Swish(),
    Linear(100,100),
    Swish(),
    Linear(100,10)
)
```

The above is a simple 4 layer MLP, notice that all the layers above are from torchfusion. Unlike pure pytorch layers, torchfusion layers have optimal initialization by default, and you can easily specify custom initialization for them. However, they are still 100% compatible with their equivalent pytorch layers. You can also mix pure pytorch and torchfusion layers in the same model.

Define optimizer and loss

```
if cuda.is_available():
    model = model.cuda()

optimizer = Adam(model.parameters())

loss_fn = nn.CrossEntropyLoss()

train_metrics = [Accuracy()]
test_metrics = [Accuracy()]
```

Print Summary and Train the model

```
learner = StandardLearner(model)

if __name__ == "__main__":
    learner.summary((1,28,28))
    learner.train(train_loader,train_metrics=train_metrics,optimizer=optimizer,loss_
↪fn=loss_fn,test_loader=test_loader,test_metrics=test_metrics,num_epochs=40,batch_
↪log=False)
```

PUTTING IT ALL TOGETHER

```
from torchfusion.layers import *
from torchfusion.datasets import *
from torchfusion.metrics import *
import torch.nn as nn
import torch.cuda as cuda
from torch.optim import Adam
from torchfusion.learners import StandardLearner
```

(continues on next page)

(continued from previous page)

```
train_loader = fashionmnist_loader(size=28,batch_size=32)
test_loader = fashionmnist_loader(size=28,train=False,batch_size=32)

model = nn.Sequential(
    Flatten(),
    Linear(784,100),
    Swish(),
    Linear(100,100),
    Swish(),
    Linear(100,100),
    Swish(),
    Linear(100,10)
)

if cuda.is_available():
    model = model.cuda()

optimizer = Adam(model.parameters())

loss_fn = nn.CrossEntropyLoss()

train_metrics = [Accuracy()]
test_metrics = [Accuracy()]

learner = StandardLearner(model)

if __name__ == "__main__":

    print(learner.summary((1,28,28)))
    learner.train(train_loader,train_metrics=train_metrics,optimizer=optimizer,loss_
↪fn=loss_fn,test_loader=test_loader,test_metrics=test_metrics,num_epochs=40,batch_
↪log=False)
```

Running the code above should reach an accuracy of about 90% after 30 epochs.

You can enable and disable epoch-end visualizations with the boolean args: **display_metrics** and **save_metrics**

PERFORMANCE METRICS

The Accuracy class measures the the topK accuracy. The default is top1, however, you can easily specify any K level.

Top K metric example:

```
train_metrics = [Accuracy(),Accuracy(topK=2),Accuracy(topK=5)]
```

Load the saved weights and evaluate performance on test set

We have just trained a classifier on Fashion MNIST and evaluated the performance at the end of each epoch. You can also use the evaluation function to evaluate the test performance separately.

Run evaluation

```
if __name__ == "__main__":
    top1_acc = Accuracy()
    top5_acc = Accuracy(topK=5)

    learner.load_model("best-models/model_3.pth")
```

(continues on next page)

(continued from previous page)

```
    learner.evaluate(test_loader,[top1_acc,top5_acc])
    print("Top1 Acc: {} Top5 Acc: {}".format(top1_acc.getValue(),top5_acc.getValue()))
```

This produces Top1 Acc: 0.871399998664856 Top5 Acc: 0.996999979019165

Inference

The ultimate goal of training models is to use them to classify new images, now that we have trained the model on fashion images, save the images below and use the code after to classify them

Inference code

```
import torch
from torchfusion.layers import *
import torch.nn as nn
import torch.cuda as cuda
from torchfusion.learners import StandardLearner
from torchfusion.utils import load_image

model = nn.Sequential(
    Flatten(),
    Linear(784,100),
    Swish(),
    Linear(100,100),
    Swish(),
    Linear(100,100),
    Swish(),
    Linear(100,10)
)

if cuda.is_available():
    model = model.cuda()


learner = StandardLearner(model)
learner.load_model("best_models\model_20.pth")

if __name__ == "__main__":

    #map class indexes to class names
    class_map = {0:"T-Shirt",1:"Trouser",2:"Pullover",3:"Dress",4:"Coat",5:"Sandal",6:
↪"Shirt",7:"Sneaker",8:"Bag",9:"Ankle Boot"}

    #Load the image
    image = load_image("sample-1.jpg",grayscale=True,target_size=28,mean=0.5,std=0.5)
```

(continues on next page)

(continued from previous page)

```
#add batch dimension
image = image.unsqueeze(0)

#run prediction
pred = learner.predict(image)

#convert prediction to probabilities
pred = torch.softmax(pred,0)

#get the predicted class
pred_class = pred.argmax().item()

#get confidence for the prediction
pred_conf = pred.max().item()

#Map class_index to name
class_name = class_map[pred_class]
print("Predicted Class: {}, Confidence: {}".format(class_name,pred_conf))
```

CHAPTER 2

Training CIFAR10!

In this section, we shall be using convolutional neural networks to train an Image Classification model on the CIFAR10 Dataset. We shall also explore more advanced concepts such as custom data transformations, learning rate scheduling and metric visualization.

CIFAR10 DESCRIPTION

Cifar10 is a dataset of 60000 images in 10 different categories. The dataset is split into a train set of 50000 images and a test set of 10000 images. CIFAR10 was collected by Alex Krizhevsky in 2009, and it is the most widely used dataset for research in Image Classification.

To learn more visit. To learn more visit. Cifar 10

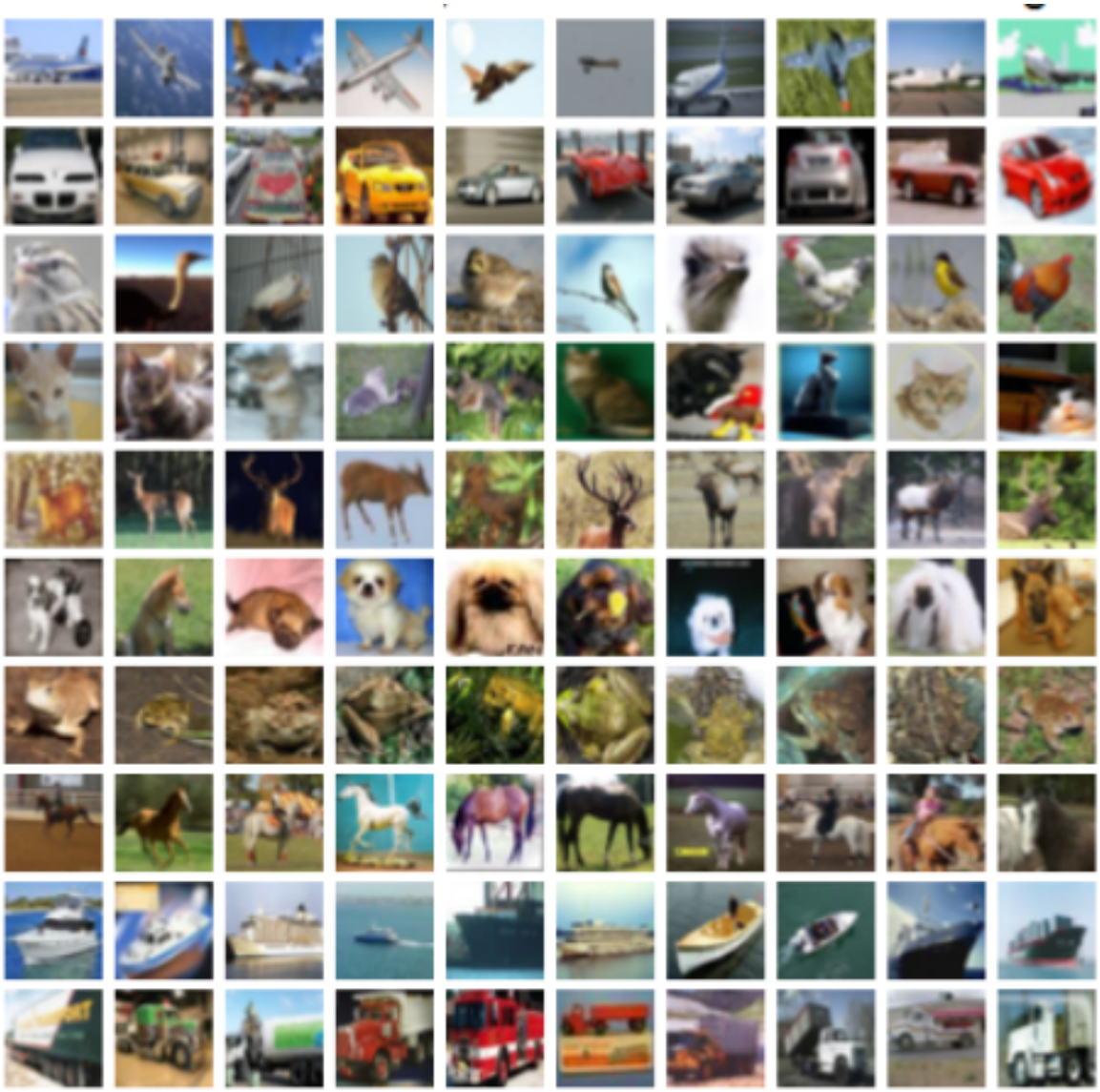
Import Classes

```
from torchfusion.layers import *
from torchfusion.datasets import *
from torchfusion.metrics import *
from torchfusion.initializers import Kaiming_Normal, Xavier_Normal
import torchvision.transforms as transforms
import torch.nn as nn
import torch.cuda as cuda
from torch.optim import Adam
from torch.optim.lr_scheduler import StepLR
from torchfusion.learners import StandardLearner
```

Load the dataset

```
train_transforms = transforms.Compose([
    transforms.RandomCrop(32,padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))
])

test_transforms = transforms.Compose([
```

(continues on next page)



(continued from previous page)

```
    transforms.CenterCrop(32),
    transforms.ToTensor(),
    transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))
])

train_loader = cifar10_loader(transform=train_transforms,batch_size=32)
test_loader = cifar10_loader(transform=test_transforms,train=False,batch_size=32)
```

Data augmentation helps to improve the performance of our models, hence, for the train set we overrided the default transformations of torchfusion with a new one containing our custom transforms. For the test set, we simply use the default transforms.

Define the model

```
class Unit(nn.Module):
    def __init__(self,in_channels,out_channels):
        super(Unit,self).__init__()
        self.conv = Conv2d(in_channels,out_channels,kernel_size=3,padding=1,weight_
↪init=Kaiming_Normal())
        self.bn = BatchNorm2d(out_channels)
        self.activation = Swish()

    def forward(self,inputs):
        outputs = self.conv(inputs)
        outputs = self.bn(outputs)
        return self.activation(outputs)

model = nn.Sequential(
    Unit(3,64),
    Unit(64,64),
    Unit(64,64),
    nn.Dropout(0.25),

    nn.MaxPool2d(kernel_size=3,stride=2),

    Unit(64,128),
    Unit(128,128),
    Unit(128,128),
    nn.Dropout(0.25),

    nn.MaxPool2d(kernel_size=3,stride=2),

    Unit(128,256),
    Unit(256,256),
    Unit(256,256),

    GlobalAvgPool2d(),

    Linear(256, 10,weight_init=Xavier_Normal())
)
```

To make the code more compact above, we first defined a *Unit* module that we reused in the model. Notice how we initialized the convolution layer with *Kaiming Normal* in the above, all torchfusion convolution layers are by default initialized with *Kaiming_Normal* and all Linear layers have default init of *Xavier_Normal*, however, we explicitly defined the intialization here to demonstrate how you can use any of the many initializers that torchfusion provides to initialize your layers. The *bias_init* arguement also allows you to initialize the bias as you want.

Define optimizer, lr scheduler and loss

```
if cuda.is_available():
    model = model.cuda()

optimizer = Adam(model.parameters(),lr=0.001)

lr_scheduler = StepLR(optimizer,step_size=30,gamma=0.1)

loss_fn = nn.CrossEntropyLoss()

train_metrics = [Accuracy()]
test_metrics = [Accuracy()]
```

In the above, we defined a learning rate scheduler to reduce the learning rate by a factor of 10 every 30 epochs. There are many learning rate schedulers in pyorch's lr_scheduler package, you can use any of them here.

Train the model

```
learner = StandardLearner(model)

if __name__ == "__main__":
    learner.train(train_loader,train_metrics=train_metrics,optimizer=optimizer,loss_
↪fn=loss_fn,model_dir="./cifar10-models",test_loader=test_loader,test_metrics=test_
↪metrics,num_epochs=200,batch_log=False,lr_scheduler=lr_scheduler,save_logs="cifar10-
↪logs.txt",display_metrics=True,save_metrics=True)
```

Here we specified a number of additional arguements, first we specified the *lr_scheduler* we earlier created, next we specified *save_logs*, this will save all logs to the file we specified, finally, *save_metrics* and *display_metrics* will display visualization of loss and metrics and save the generated plots. The save plots,logs and models can all be found in the directory *cifar10-models* that we specified above.

PUTTING IT ALL TOGETHER

```
from torchfusion.layers import *
from torchfusion.datasets import *
from torchfusion.metrics import *
from torchfusion.initializers import Kaiming_Normal, Xavier_Normal
import torchvision.transforms as transforms
import torch.nn as nn
import torch.cuda as cuda
from torch.optim import Adam
from torch.optim.lr_scheduler import StepLR
from torchfusion.learners import StandardLearner

train_transforms = transforms.Compose([
    transforms.RandomCrop(32,padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))
])

test_transforms = transforms.Compose([
    transforms.CenterCrop(32),
    transforms.ToTensor(),
    transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))
])
```

(continues on next page)

(continued from previous page)

```
train_loader = cifar10_loader(transform=train_transforms,batch_size=32)
test_loader = cifar10_loader(transform=test_transforms,train=False,batch_size=32)

class Unit(nn.Module):
    def __init__(self,in_channels,out_channels):
        super(Unit,self).__init__()
        self.conv = Conv2d(in_channels,out_channels,kernel_size=3,padding=1,weight_
↪init=Kaiming_Normal())
        self.bn = BatchNorm2d(out_channels)
        self.activation = Swish()

    def forward(self,inputs):
        outputs = self.conv(inputs)
        outputs = self.bn(outputs)
        return self.activation(outputs)

model = nn.Sequential(
    Unit(3,64),
    Unit(64,64),
    Unit(64,64),
    nn.Dropout(0.25),

    nn.MaxPool2d(kernel_size=3,stride=2),

    Unit(64,128),
    Unit(128,128),
    Unit(128,128),
    nn.Dropout(0.25),

    nn.MaxPool2d(kernel_size=3,stride=2),

    Unit(128,256),
    Unit(256,256),
    Unit(256,256),

    GlobalAvgPool2d(),

    Linear(256, 10,weight_init=Xavier_Normal())
)


if cuda.is_available():
    model = model.cuda()

optimizer = Adam(model.parameters(),lr=0.001)

lr_scheduler = StepLR(optimizer,step_size=30,gamma=0.1)

loss_fn = nn.CrossEntropyLoss()

train_metrics = [Accuracy()]
test_metrics = [Accuracy()]

learner = StandardLearner(model)

if __name__ == "__main__":
    learner.train(train_loader,train_metrics=train_metrics,optimizer=optimizer,loss_
↪fn=loss_fn,model_dir="./cifar10-models",test_loader=test_loader,test_metrics=test_
↪metrics,num_epochs=30,batch_log=False,lr_scheduler=lr_scheduler,save_logs="cifar10-
↪logs.txt",display_metrics=True,save_metrics=True)
```

(continues on next page)

(continued from previous page)

CHAPTER 3

Mixed Precision Training

Deep Learning models are usually trained using standard 32 bit floating point arithmetic. To speed up the training of deep learning models, a lot of research has gone into using lower bit precison arithmetic such as 8 bit and 16 bit arithmetic. The lower bits are much faster than the 32 bit precision arithmetic. NVIDIA Volta GPUs have tensor cores specialized for 16 bit precison arithmetic, taking advantage of them could lead to significant speed up in training of large deep learning models.

To learn more about mixed precision training, read ‘‘Nvidia Mixed Precision Training https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html/‘_

The greatest challenge with training in mixed precision made is radical decrease in model accuracy. Hence, a naive implementation of mixed precision training will result in very poor performance. TorchFusion includes highly optimized procedures for training Deep Learning Models in mixed precision without compromising performance.

PROCEDURE

The following are the procedures you need to follow to train torchfusion models in mixed precision.

Import Classes

```
from torchfusion.fp16_utils import half_model, FP16_Optimizer
```

CONVERT MODEL AND OPTIMIZER INTO FP16

```
model = half_model(model)
optimizer = FP16_Optimizer(Adam(model.parameters()))
learner = StandardLearner(model)
learner.half()

# if using lr_scheduler
lr_scheduler = StepLR(optimizer.optimizer,step_size=30,gamma=0.1)
```

PUTTING IT ALL TOGETHER

```
from torchfusion.layers import *
from torchfusion.datasets import *
```

(continues on next page)

(continued from previous page)

```
from torchfusion.metrics import *
from torchfusion.initializers import Kaiming_Normal, Xavier_Normal
import torchvision.transforms as transforms
import torch.nn as nn
import torch.cuda as cuda
from torch.optim import Adam
from torch.optim.lr_scheduler import StepLR
from torchfusion.learners import StandardLearner

train_transforms = transforms.Compose([
    transforms.RandomCrop(32,padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))
])

test_transforms = transforms.Compose([
    transforms.CenterCrop(32),
    transforms.ToTensor(),
    transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))
])

train_loader = cifar10_loader(transform=train_transforms,batch_size=32)
test_loader = cifar10_loader(transform=test_transforms,train=False,batch_size=32)

class Unit(nn.Module):
    def __init__(self,in_channels,out_channels):
        super(Unit,self).__init__()
        self.conv = Conv2d(in_channels,out_channels,kernel_size=3,padding=1,weight_
↪init=Kaiming_Normal())
        self.bn = BatchNorm2d(out_channels)
        self.activation = Swish()

    def forward(self,inputs):
        outputs = self.conv(inputs)
        outputs = self.bn(outputs)
        return self.activation(outputs)

model = nn.Sequential(
    Unit(3,64),
    Unit(64,64),
    Unit(64,64),
    nn.Dropout(0.25),

    nn.MaxPool2d(kernel_size=3,stride=2),

    Unit(64,128),
    Unit(128,128),
    Unit(128,128),
    nn.Dropout(0.25),

    nn.MaxPool2d(kernel_size=3,stride=2),

    Unit(128,256),
    Unit(256,256),
    Unit(256,256),
```

(continues on next page)

(continued from previous page)

```
    GlobalAvgPool2d(),

    Linear(256, 10,weight_init=Xavier_Normal())
)


if cuda.is_available():
    model = model.cuda()
model = half_model(model)

optimizer = FP16_Optimizer(Adam(model.parameters(),lr=0.001))

lr_scheduler = StepLR(optimizer.optimizer,step_size=30,gamma=0.1)

loss_fn = nn.CrossEntropyLoss()

train_metrics = [Accuracy()]
test_metrics = [Accuracy()]

learner = StandardLearner(model)
learner.half()

if __name__ == "__main__":
    learner.train(train_loader,train_metrics=train_metrics,optimizer=optimizer,loss_
↪fn=loss_fn,model_dir="./cifar10-models",test_loader=test_loader,test_metrics=test_
↪metrics,num_epochs=30,batch_log=False,lr_scheduler=lr_scheduler,save_logs="cifar10-
↪logs.txt",display_metrics=True,save_metrics=True)
```

CHAPTER 4

Training With Custom Datasets!

While TorchFusion provides pre-defined loaders for popular standard datasets. Very often, you will need to train on your own custom datasets. TorcFusion provides loaders for any image dataset organized into a single folder with subfolders representing each class of images. For example, if you are training a model to recognize cats and dogs, you should have a train folder with two subfolders, one for dogs and one for cats.

Import Classes

```
from torchfusion.layers import *
from torchfusion.datasets import *
from torchfusion.metrics import *
from torchfusion.initializers import Kaiming_Normal, Xavier_Normal
import torchvision.transforms as transforms
import torch.nn as nn
import torch.cuda as cuda
from torch.optim import Adam
from torch.optim.lr_scheduler import StepLR
from torchfusion.learners import StandardLearner
```

Load the dataset

```
train_transforms = transforms.Compose([
    transforms.RandomCrop(224,padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))
])

test_transforms = transforms.Compose([
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))
])

train_loader = imagefolder_loader(transform=train_transforms,batch_size=32,
↪shuffle=True,root="path-to-train-folder")
```

(continues on next page)

(continued from previous page)

```
test_loader = imagefolder_loader(transform=test_transforms,shuffle=False,batch_
↪size=32,root="path-to-test-folder")
```

Define the model

```
class Unit(nn.Module):
    def __init__(self,in_channels,out_channels):
        super(Unit,self).__init__()
        self.conv = Conv2d(in_channels,out_channels,kernel_size=3,padding=1,weight_
↪init=Kaiming_Normal())
        self.bn = BatchNorm2d(out_channels)
        self.activation = Swish()

    def forward(self,inputs):
        outputs = self.conv(inputs)
        outputs = self.bn(outputs)
        return self.activation(outputs)

model = nn.Sequential(
    Unit(3,64),
    Unit(64,64),
    Unit(64,64),
    nn.Dropout(0.25),

    nn.MaxPool2d(kernel_size=3,stride=2),

    Unit(64,128),
    Unit(128,128),
    Unit(128,128),
    nn.Dropout(0.25),

    nn.MaxPool2d(kernel_size=3,stride=2),

    Unit(128,256),
    Unit(256,256),
    Unit(256,256),

    GlobalAvgPool2d(),

    Linear(256, 10,weight_init=Xavier_Normal())
)
```

Define optimizer, lr scheduler and loss

```
if cuda.is_available():
    model = model.cuda()

optimizer = Adam(model.parameters(),lr=0.001)

lr_scheduler = StepLR(optimizer,step_size=30,gamma=0.1)

loss_fn = nn.CrossEntropyLoss()

train_metrics = [Accuracy()]
test_metrics = [Accuracy()]
```

Train the model

```
learner = StandardLearner(model)

if __name__ == "__main__":
    learner.train(train_loader,train_metrics=train_metrics,optimizer=optimizer,loss_
↪fn=loss_fn,model_dir="./custom-models",test_loader=test_loader,test_metrics=test_
↪metrics,num_epochs=200,batch_log=False,lr_scheduler=lr_scheduler,save_logs="custom-
↪model-logs.txt",display_metrics=True,save_metrics=True)
```

PUTTING IT ALL TOGETHER

```
from torchfusion.layers import *
from torchfusion.datasets import *
from torchfusion.metrics import *
from torchfusion.initializers import Kaiming_Normal, Xavier_Normal
import torchvision.transforms as transforms
import torch.nn as nn
import torch.cuda as cuda
from torch.optim import Adam
from torch.optim.lr_scheduler import StepLR
from torchfusion.learners import StandardLearner

train_transforms = transforms.Compose([
    transforms.RandomCrop(224,padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))
])

test_transforms = transforms.Compose([
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))
])

train_loader = imagefolder_loader(transform=train_transforms,batch_size=32,
↪shuffle=True,root="path-to-train-folder")
test_loader = imagefolder_loader(transform=test_tranforms,shuffle=False,batch_size=32,
↪root="path-to-test-folder")

class Unit(nn.Module):
    def __init__(self,in_channels,out_channels):
        super(Unit,self).__init__()
        self.conv = Conv2d(in_channels,out_channels,kernel_size=3,padding=1,weight_
↪init=Kaiming_Normal())
        self.bn = BatchNorm2d(out_channels)
        self.activation = Swish()

    def forward(self,inputs):
        outputs = self.conv(inputs)
        outputs = self.bn(outputs)
        return self.activation(outputs)

model = nn.Sequential(
    Unit(3,64),
    Unit(64,64),
    Unit(64,64),
    nn.Dropout(0.25),
```

(continues on next page)

(continued from previous page)

```
    nn.MaxPool2d(kernel_size=3,stride=2),

    Unit(64,128),
    Unit(128,128),
    Unit(128,128),
    nn.Dropout(0.25),

    nn.MaxPool2d(kernel_size=3,stride=2),

    Unit(128,256),
    Unit(256,256),
    Unit(256,256),

    GlobalAvgPool2d(),

    Linear(256, 10,weight_init=Xavier_Normal())
)


if cuda.is_available():
    model = model.cuda()

optimizer = Adam(model.parameters(),lr=0.001)

lr_scheduler = StepLR(optimizer,step_size=30,gamma=0.1)

loss_fn = nn.CrossEntropyLoss()

train_metrics = [Accuracy()]
test_metrics = [Accuracy()]

learner = StandardLearner(model)

learner = StandardLearner(model)

if __name__ == "__main__":
    learner.train(train_loader,train_metrics=train_metrics,optimizer=optimizer,loss_
↪fn=loss_fn,model_dir="./custom-models",test_loader=test_loader,test_metrics=test_
↪metrics,num_epochs=30,batch_log=False,lr_scheduler=lr_scheduler,save_logs="custom-
↪models-logs.txt",display_metrics=True,save_metrics=True)
```

CHAPTER 5

Logging and Visualizing the Training Process!

While torchfusion allows you to easily visualize the training process using matplotlib based charts, for more advanced visualization, Torchfusion has in-built support for visualizing the training process in both Visdom and Tensorboard.

Logging with Visdom

Visdom is a visualizing kit developed by Facebook AI Research, visdom was installed the first time you installed Torchfusion.

To visualize your training process in visdom, follow the steps below.

Step 1: Import the visdom logger:

```
from torchfusion.utils import VisdomLogger
```

Step 2: Specify the logger in your train func

```
visdom_logger = VisdomLogger()
if __name__ == "__main__":
    learner.train(train_loader,train_metrics=train_metrics,optimizer=optimizer,loss_
↪fn=loss_fn,visdom_log=visdom_logger)
```

Step 3: Start the visdom server from the commad promp

```
python -m visdom.server
```

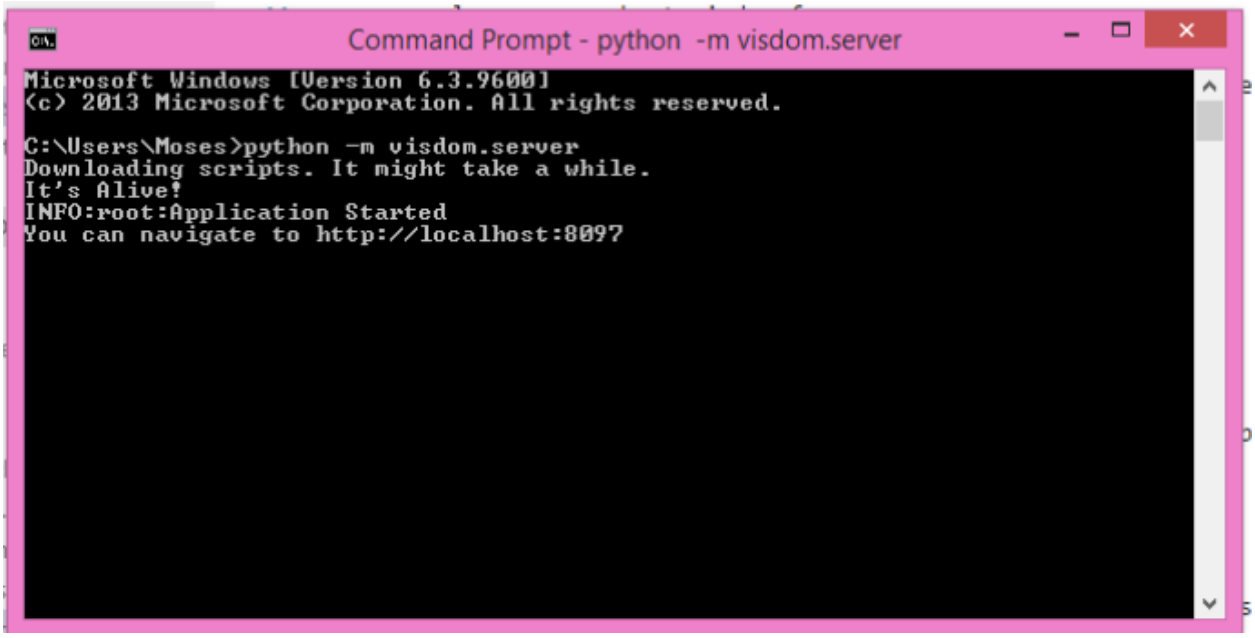
Ensure you are connected to the intenet when you run this, as visdom will need to download a few scripts.

Notice the output above, open your browser and navigate to the url given, in this case: localhost:8097

Run this to see visdom in action

```
from torchfusion.layers import *
from torchfusion.datasets import *
from torchfusion.metrics import *
import torch.nn as nn
import torch.cuda as cuda
```

(continues on next page)



(continued from previous page)

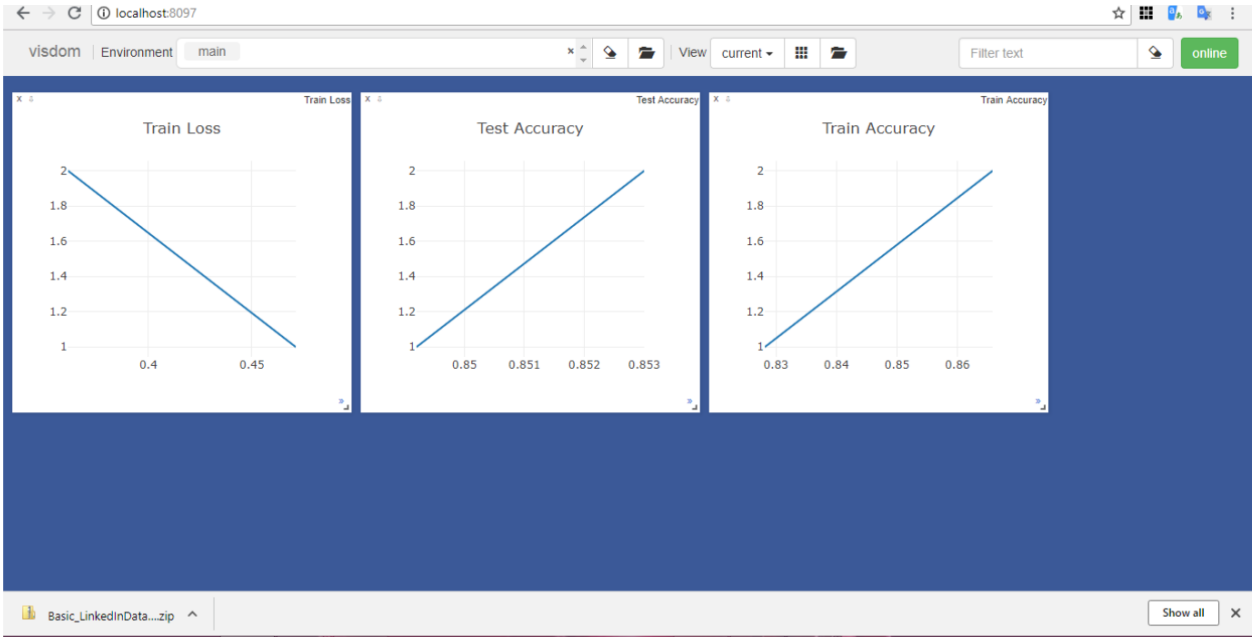
```
from torch.optim import Adam
from torchfusion.learners import StandardLearner
from torchfusion.utils import VisdomLogger

train_loader = fashionmnist_loader(size=28,batch_size=32)
test_loader = fashionmnist_loader(size=28,train=False,batch_size=32)

model = nn.Sequential(
    Flatten(),
    Linear(784,100),
    Swish(),
    Linear(100,100),
    Swish(),
    Linear(100,100),
    Swish(),
    Linear(100,10)
)

if cuda.is_available():
    model = model.cuda()

optimizer = Adam(model.parameters())

loss_fn = nn.CrossEntropyLoss()

train_metrics = [Accuracy()]
test_metrics = [Accuracy()]

visdom_logger = VisdomLogger()

learner = StandardLearner(model)

if __name__ == "__main__":
```

(continues on next page)

(continued from previous page)

```
    print(learner.summary((1,28,28)))
    learner.train(train_loader,train_metrics=train_metrics,visdom_log=visdom_logger,
↪optimizer=optimizer,loss_fn=loss_fn,test_loader=test_loader,test_metrics=test_
↪metrics,num_epochs=30,batch_log=False)
```

Generated Visuals



Using Tensorboard Torchfusion can also generate tensorboard logs that you can view with tensorboard, while Torchfusion does not require tensorboard or tensorflow installed to generate the logs as it uses TensorboardX, you need to install both tensorflow and tensorboard to view the generated logs.

Vist tensorflow.org for instructions on installing tensorflow and https://github.com/tensorflow/tensorboard for instructions on installing tensorbord

Once installed, you can use tensorboard in just ONE Line.

Specify the tensorboard_log in your train func

```
visdom_logger = VisdomLogger()
if __name__ == "__main__":
    learner.train(train_loader,train_metrics=train_metrics,optimizer=optimizer,loss_
↪fn=loss_fn,tensorboard_log="./tboard-logs",visdom_log=visdom_logger)
```

Notice how we use both tensorboard and visdom here, we can use either independently or both if we want to.

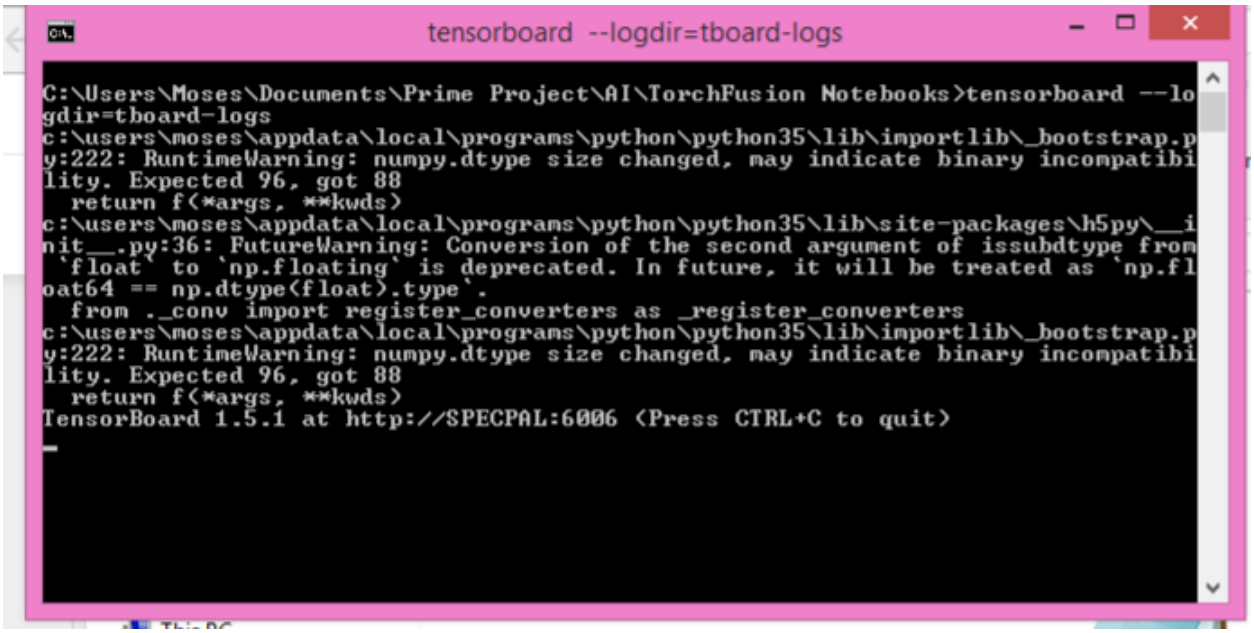
Start the tensorboard server from the command prompt :: tensorboard –logdir=tboard-logs

Notice the output above, open your browser and navigate to the url given, in this case: specpal:6006

Run this to see tensorboard in action

```
from torchfusion.layers import *
from torchfusion.datasets import *
from torchfusion.metrics import *
import torch.nn as nn
```

(continues on next page)



(continued from previous page)

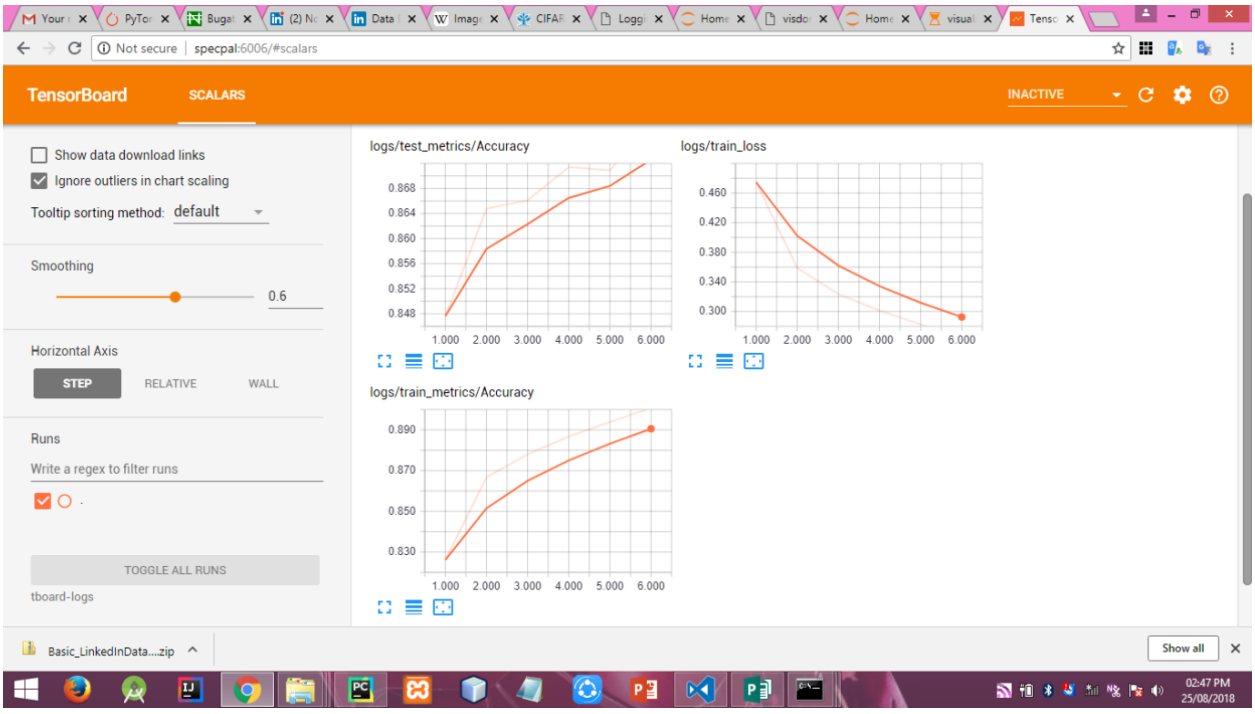
```
import torch.cuda as cuda
from torch.optim import Adam
from torchfusion.learners import StandardLearner
from torchfusion.utils import VisdomLogger

train_loader = fashionmnist_loader(size=28,batch_size=32)
test_loader = fashionmnist_loader(size=28,train=False,batch_size=32)

model = nn.Sequential(
    Flatten(),
    Linear(784,100),
    Swish(),
    Linear(100,100),
    Swish(),
    Linear(100,100),
    Swish(),
    Linear(100,10)
)

if cuda.is_available():
    model = model.cuda()

optimizer = Adam(model.parameters())

loss_fn = nn.CrossEntropyLoss()

train_metrics = [Accuracy()]
test_metrics = [Accuracy()]

visdom_logger = VisdomLogger()

learner = StandardLearner(model)
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":

    print(learner.summary((1,28,28)))
    learner.train(train_loader,train_metrics=train_metrics,tensorboard_log="./tboard-
↪logs",visdom_log=visdom_logger,optimizer=optimizer,loss_fn=loss_fn,test_loader=test_
↪loader,test_metrics=test_metrics,num_epochs=30,batch_log=False)
```

Generated Visuals



CHAPTER 6

Buiding Custom Trainers!

While Tochfusion strives to provide very good trainers, we know researchers often need custom training logic. TorchFusion makes using custom training logic easy. All you need to do is extend the Learners.

Sample Custom Trainer

```
#Extend the StandardLearner
   class CustomLearner(StandardLearner):

       #Override the train logic
       def __train_func__(self, data):

           self.optimizer.zero_grad()

           if self.clip_grads is not None:
               clip_grads(self.model,self.clip_grads[0],self.clip_grads[1])

           train_x, train_y = data

           batch_size = train_x.size(0)

           train_x = Variable(train_x.cuda() if self.cuda else train_x)

           train_y = Variable(train_y.cuda() if self.cuda else train_y)

           outputs = self.model(train_x)
           loss = self.loss_fn(outputs, train_y)
           loss.backward()

           self.optimizer.step()

           self.train_running_loss.add_(loss.cpu() * batch_size)

           for metric in self.train_metrics:
               metric.update(outputs, train_y)
```

(continues on next page)

(continued from previous page)

```
        #Override the evaluation logic
        def __eval_function__(self, data):

            test_x, test_y = data

            test_x = Variable(test_x.cuda() if self.cuda else test_x)

            test_y = Variable(test_y.cuda() if self.cuda else test_y)

            outputs = self.model(test_x)

            for metric in self.test_metrics:
                metric.update(outputs, test_y)

        #Override the validation logic

        def __val_function__(self, data):

            val_x, val_y = data
            val_x = Variable(val_x.cuda() if self.cuda else val_x)

            val_y = Variable(val_y.cuda() if self.cuda else val_y)

            outputs = self.model(val_x)

            for metric in self.val_metrics:
                metric.update(outputs, val_y)

        #override the prediction logic

        def __predict_func__(self, inputs):

            inputs = Variable(inputs.cuda() if self.cuda else inputs)

            return self.model(inputs)
```

CHAPTER 7

Introduction to Generative Adversarial Networks

Classification and regression models are used for predictive tasks, they map diverse inputs to fixed outputs, these class of models are called discriminative models. Generative Models do the opposite, they generate diverse outputs from fixed inputs. An example generative model is a model that can generate new pictures of cars simply from a text description. Different generative models exist, the most successful are Generative Adversarial Networks by Goood-fellow et al,2014 These models consist of a generator model which is responsible for generating new outputs, and a discriminator model that attempts to tell if the generated outputs are real or fake. During training, the discriminator is presented with both real and generated images. The discriminator is trained is trained to correctly tell the real images apart from generated images, while the generator is trained to generate images that are so real that the discriminaor will classify them as real. Hence, the two networks are competing with each other and the generator is trying to fool the discriminator. While the logic of GANs can be slighly complicated, TorchFusion makes using them a breeze and provides a highly sophiscated framework for doing research with custom GAN logic.

Below is are two pictures generated by a GAN. Karras et al,2017

UNCONDITIONAL GAN EXAMPLE

Earlier on, we learnt to correctly classify grayscale fashion images, now we shall attempt to generate them instead.

Step 1: Imports!

```
from torchfusion.gan.learners import *
from torchfusion.gan.applications import StandardGenerator,
↪StandardProjectionDiscriminator
from torch.optim import Adam
from torchfusion.datasets import fashionmnist_loader
import torch.cuda as cuda
import torch.nn as nn
```

Define Generator and Discriminator

```
G = StandardGenerator(output_size=(1,32,32),latent_size=128)
D = StandardProjectionDiscriminator(input_size=(1,32,32),apply_sigmoid=False)

if cuda.is_available():
```

(continues on next page)

(continued from previous page)

```
    G = nn.DataParallel(G.cuda())
    D = nn.DataParallel(D.cuda())
```

Here, we use predefined Generator and Discriminator in torchfusion, we set the size of the generated images to be 1,32,32 and the latent_size as 128. The images will be generated from the latent_code which will be of the size 128.

Setup optimizers

```
g_optim = Adam(G.parameters(),lr=0.0002,betas=(0.5,0.999))
d_optim = Adam(D.parameters(),lr=0.0002,betas=(0.5,0.999))
```

Since our generator and discriminator are separately trained, we need to specify different optimizers for them, try to stick to the hyper-parameters here as GANs can be very sensitive to this values.

load dataset

```
dataset = fashionmnist_loader(size=32,batch_size=64)
```

The image size here is set to be the same as the size of the images to be generated.

Define the learner

```
learner = RStandardGanLearner(G,D)
```

The Learner does all the heavy-lifting

Train the Models

```
if __name__ == "__main__":
    learner.train(dataset,gen_optimizer=g_optim,disc_optimizer=d_optim,save_outputs_
↪interval=500,model_dir="./fashion-gan",latent_size=128,num_epochs=50,batch_
↪log=False)
```

By specifying the *save_outputs_interval* as 500, every 500 batch iterations it will print sample generated immages. Note that this is different from number of epochs.

Putting it all Together

```
from torchfusion.gan.learners import *
from torchfusion.gan.applications import StandardGenerator,
↪StandardProjectionDiscriminator
from torch.optim import Adam
from torchfusion.datasets import fashionmnist_loader
import torch.cuda as cuda
import torch.nn as nn

G = StandardGenerator(output_size=(1,32,32),latent_size=128)
D = StandardProjectionDiscriminator(input_size=(1,32,32),apply_sigmoid=False)

if cuda.is_available():
    G = nn.DataParallel(G.cuda())
    D = nn.DataParallel(D.cuda())

g_optim = Adam(G.parameters(),lr=0.0002,betas=(0.5,0.999))
d_optim = Adam(D.parameters(),lr=0.0002,betas=(0.5,0.999))

dataset = fashionmnist_loader(size=32,batch_size=64)

learner = RStandardGanLearner(G,D)

if __name__ == "__main__":
    learner.train(dataset,gen_optimizer=g_optim,disc_optimizer=d_optim,save_outputs_
↪interval=500,model_dir="./fashion-gan",latent_size=128,num_epochs=50,batch_
↪log=False)
```

CHAPTER 8

Conditional Generative Adversarial Networks

In the previous chapter, images were randomly generated without respect to classes. Here we shall generated Images of specific classes While Conditional GANs are complex, torchfusion makes this super easy, all you have to do is state the num_classes in the Generator, Discriminator and in the Learner

Use classes in Generator and Discriminator

```
G = StandardGenerator(output_size=(1,32,32),latent_size=128,num_classes=10)
D = StandardProjectionDiscriminator(input_size=(1,32,32),apply_sigmoid=False,num_
↪classes=10)
```

Define num_classes in Learner

```
if __name__ == "__main__":
    learner.train(dataset,num_classes=10,gen_optimizer=g_optim,disc_optimizer=d_optim,
↪save_outputs_interval=500,model_dir="./fashion-gan",latent_size=128,num_epochs=50,
↪batch_log=False)
```

And thats it ! The full code is below

Putting it all Together

```
from torchfusion.gan.learners import *
from torchfusion.gan.applications import StandardGenerator,
↪StandardProjectionDiscriminator
from torch.optim import Adam
from torchfusion.datasets import fashionmnist_loader
import torch.cuda as cuda
import torch.nn as nn

G = StandardGenerator(output_size=(1,32,32),latent_size=128,num_classes=10)
D = StandardProjectionDiscriminator(input_size=(1,32,32),apply_sigmoid=False,num_
↪classes=10)

if cuda.is_available():
    G = nn.DataParallel(G.cuda())
```

(continues on next page)

(continued from previous page)

```
    D = nn.DataParallel(D.cuda())

g_optim = Adam(G.parameters(),lr=0.0002,betas=(0.5,0.999))
d_optim = Adam(D.parameters(),lr=0.0002,betas=(0.5,0.999))

dataset = fashionmnist_loader(size=32,batch_size=64)

learner = RStandardGanLearner(G,D)

if __name__ == "__main__":
    learner.train(dataset,num_classes=10,gen_optimizer=g_optim,disc_optimizer=d_optim,
↪save_outputs_interval=500,model_dir="./fashion-gan",latent_size=128,num_epochs=50,
↪batch_log=False)
```

After just 17 epochs, this produces

CHAPTER 9

GAN Inference

Now that we have learn't how to generate images of specific classes, here we shall use the trained generator for inference

Putting it all Together

```
from torchfusion.gan.learners import *
from torchfusion.gan.applications import StandardGenerator
import torch.cuda as cuda
import torch.nn as nn
from torchvision.utils import save_image
import torch
from torch.distributions import Normal


G = StandardGenerator(output_size=(1,32,32),latent_size=128,num_classes=10)

if cuda.is_available():
    G = nn.DataParallel(G.cuda())

learner = RStandardGanLearner(G,None)
learner.load_generator("path-to-trained-gen")

if __name__ == "__main__":
    "Define an instance of the normal distribution"
    dist = Normal(0,1)

    #Get a sample latent vector from the distribution
    latent_vector = dist.sample((1,128))

    #Define the class of the image you want to generate
    label = torch.LongTensor(1).fill_(5)

    #Run inference
    image = learner.predict([latent_vector,label])
```

(continues on next page)

(continued from previous page)

```
    #Save generated image
    save_image(image, "image.jpg")
```

CHAPTER 10

Buiding Custom Trainers!

Torchfusion provides a wide variety of GAN Learners, you will find them in the torchfusion.gan.learners package However, lots of research is ongoing into improved techniques for GANs, hence, we provide multiple levels of abstractions to faciliate research.

Custom Loss

```
#Extend the StandardBaseGanLearner
class CustomGanLearner(StandardBaseGanLearner):
    #Override the __update_discriminator_loss__
    def __update_discriminator_loss__(self, real_images, gen_images, real_preds, gen_
↪preds):

        pred_loss = -torch.mean(real_preds - gen_preds)

        return pred_loss

    #Override the __update_generator_loss__
    def __update_generator_loss__(self,real_images,gen_images,real_preds,gen_preds):

        pred_loss = -torch.mean(gen_preds - real_preds)
        return pred_loss
```

Custom Training Logic

```
#Extend BaseGanCore
class CustomGanLearner(BaseGanCore):

    #Extend train
    def train(self,train_loader, gen_optimizer,disc_optimizer,latent_size,loss_fn=nn.
↪BCELoss(),**kwargs):

        self.latent_size = latent_size
        self.loss_fn = loss_fn
        super().__train_loop__(train_loader,gen_optimizer,disc_optimizer,**kwargs)
```

(continues on next page)

(continued from previous page)

```
    #Extend __disc_train_func__
    def __disc_train_func__(self, data):

        super().__disc_train_func__(data)

        self.disc_optimizer.zero_grad()

        if isinstance(data, list) or isinstance(data, tuple):
            x = data[0]
        else:
            x = data

        batch_size = x.size(0)

        source = self.dist.sample((batch_size,self.latent_size))

        real_labels = torch.ones(batch_size,1)
        fake_labels = torch.zeros(batch_size,1)

        if self.cuda:
            x = x.cuda()
            source = source.cuda()
            real_labels = real_labels.cuda()
            fake_labels = fake_labels.cuda()

        x = Variable(x)
        source = Variable(source)

        outputs = self.disc_model(x)

        generated = self.gen_model(source)
        gen_outputs = self.disc_model(generated.detach())

        gen_loss = self.loss_fn(gen_outputs,fake_labels)

        real_loss = self.loss_fn(outputs,real_labels)

        loss = gen_loss + real_loss
        loss.backward()
        self.disc_optimizer.step()

        self.disc_running_loss.add_(loss.cpu() * batch_size)


    #Extend __gen_train_func__
    def __gen_train_func__(self, data):

        super().__gen_train_func__(data)

        self.gen_optimizer.zero_grad()

        if isinstance(data, list) or isinstance(data, tuple):
            x = data[0]
        else:
            x = data
        batch_size = x.size(0)
```

(continues on next page)

(continued from previous page)

```
        source = self.dist.sample((batch_size,self.latent_size))

        real_labels = torch.ones(batch_size,1)

        if self.cuda:
            source = source.cuda()
            real_labels = real_labels.cuda()

        source = Variable(source)

        fake_images = self.gen_model(source)
        outputs = self.disc_model(fake_images)

        loss = self.loss_fn(outputs,real_labels)
        loss.backward()

        self.gen_optimizer.step()

        self.gen_running_loss.add_(loss.cpu() * batch_size)
```

Examples Visit https://github.com/AICommons/TorchfusionExamples for example codes in TorchFusion