zliu93_HW10_1

April 29, 2019

1 Homework 10 - CIFAR10 Image Classification with PyTorch

1.1 About

The goal of the homework is to train a convolutional neural network on the standard CIFAR10 image classfication dataset.

When solving machine learning tasks using neural networks, one typically starts with a simple network architecture and then improves the network by adding new layers, retraining, adjusting parameters, retraining, etc. We attempt to illustrate this process below with several architecture improvements.

1.2 Dev Environment

1.2.1 Working on Google Colab

You may choose to work locally or on Google Colaboratory. You have access to free compute through this service. Colab is recommended since it will be setup correctly and will have access to GPU resources. 1. Visit https://colab.research.google.com/drive 2. Navigate to the `Upload` tab, and upload your `HW10.ipynb` 3. Now on the top right corner, under the `Comment` and `Share` options, you should see a `Connect` option. Once you are connected, you will have access to a VM with 12GB RAM, 50 GB disk space and a single GPU. The dropdown menu will allow you to connect to a local runtime as well.

Notes: * If you do not have a working setup for Python 3, this is your best bet. It will also save you from heavy installations like `tensorflow` if you don't want to deal with those. * ***There is a downside*. You can only use this instance for a single 12-hour stretch, after which your data will be deleted, and you would have redownload all your datasets, any libraries not already on the VM, and regenerate your logs**.

1.2.2 Installing PyTorch and Dependencies

The instructions for installing and setting up PyTorch can be found at https://pytorch.org/get-started/locally/. Make sure you follow the instructions for your machine. For any of the remaining libraries used in this assignment: * We have provided a `hw8_requirements.txt` file on the homework web page. * Download this file, and in the same directory you can run `pip3 install -r hw8_requirements.txt` Check that PyTorch installed correctly by running the following:

```
In [1]: import torch
        torch.rand(5, 3)
```

```
Out[1]: tensor([[0.4562, 0.3110, 0.8691],
                [0.3174, 0.4489, 0.0897],
                [0.2088, 0.8086, 0.4461],
                [0.1041, 0.8156, 0.4711],
                [0.1282, 0.8470, 0.2414]])
```

1.3 Part 0 Imports and Basic Setup (5 Points)

First, import the required libraries as follows. The libraries we will use will be the same as those in HW8.

```
In [1]: import numpy as np
        import torch
        from torch import nn
        from torch import optim

        import matplotlib.pyplot as plt
```

GPU Support

Training of large network can take a long time. PyTorch supports GPU with just a small amount of effort.

When creating our networks, we will call `net.to(device)` to tell the network to train on the GPU, if one is available. Note, if the network utilizes the GPU, it is important that any tensors we use with it (such as the data) also reside on the CPU. Thus, a call like `images = images.to(device)` is necessary with any data we want to use with the GPU.

Note: If you can't get access to a GPU, don't worry to much. Since we use very small networks, the difference between CPU and GPU isn't large and in some cases GPU will actually be slower.

```
In [7]: import torch.cuda as cuda

        # Use a GPU, i.e. cuda:0 device if it available.
        device = torch.device("cuda:0" if cuda.is_available() else "cpu")
        print(device)
```

```
cpu
```

1.3.1 Training Code

```
In [2]: import time

        class Flatten(nn.Module):
          """NN Module that flattens the incoming tensor."""
          def forward(self, input):
            return input.view(input.size(0), -1)

        def train(model, train_loader, test_loader, loss_func, opt, num_epochs=10):
          all_training_loss = np.zeros((0,2))
          all_training_acc = np.zeros((0,2))
```

```
all_test_loss = np.zeros((0,2))
all_test_acc = np.zeros((0,2))

training_step = 0
training_loss, training_acc = 2.0, 0.0
print_every = 1000

start = time.clock()

for i in range(num_epochs):
  epoch_start = time.clock()

  model.train()
  for images, labels in train_loader:
    images, labels = images.to(device), labels.to(device)
    opt.zero_grad()

    preds = model(images)
    loss = loss_func(preds, labels)
    loss.backward()
    opt.step()

    training_loss += loss.item()
    training_acc += (torch.argmax(preds, dim=1)==labels).float().mean()

    if training_step % print_every == 0:
      training_loss /= print_every
      training_acc /= print_every

      all_training_loss = np.concatenate((all_training_loss, [[training_step, trainin
      all_training_acc = np.concatenate((all_training_acc, [[training_step, training_

      print('  Epoch %d @ step %d: Train Loss: %3f, Train Accuracy: %3f' % (
          i, training_step, training_loss, training_acc))
      training_loss, training_acc = 0.0, 0.0

    training_step+=1

  model.eval()
  with torch.no_grad():
    validation_loss, validation_acc = 0.0, 0.0
    count = 0
    for images, labels in test_loader:
      images, labels = images.to(device), labels.to(device)
      output = model(images)
      validation_loss+=loss_func(output,labels)
      validation_acc+=(torch.argmax(output, dim=1) == labels).float().mean()
      count += 1
```

```
            validation_loss/=count
            validation_acc/=count

            all_test_loss = np.concatenate((all_test_loss, [[training_step, validation_loss]]
            all_test_acc = np.concatenate((all_test_acc, [[training_step, validation_acc]]))

            epoch_time = time.clock() - epoch_start

            print('Epoch %d Test Loss: %3f, Test Accuracy: %3f, time: %.1fs' % (
                i, validation_loss, validation_acc, epoch_time))

        total_time = time.clock() - start
        print('Final Test Loss: %3f, Test Accuracy: %3f, Total time: %.1fs' % (
            validation_loss, validation_acc, total_time))

        return {'loss': { 'train': all_training_loss, 'test': all_test_loss },
                'accuracy': { 'train': all_training_acc, 'test': all_test_acc }}

    def plot_graphs(model_name, metrics):
      for metric, values in metrics.items():
        for name, v in values.items():
          plt.plot(v[:,0], v[:,1], label=name)
        plt.title(f'{metric} for {model_name}')
        plt.legend()
        plt.xlabel("Training Steps")
        plt.ylabel(metric)
        plt.show()
```

Load the** CIFA-10** dataset and define the transformations. You may also want to print its structure, size, as well as sample a few images to get a sense of how to design the network.

```
In [5]: !mkdir hw10_data
```

```
A subdirectory or file hw10_data already exists.
```

```
In [3]: # Download the data.
        from torchvision import datasets, transforms

        transformations = transforms.Compose(
            [transforms.ToTensor(),
             transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
        train_set = datasets.CIFAR10(root='hw10_data/', download=True, transform=transformatio
        test_set = datasets.CIFAR10(root='hw10_data', download=True, train=False, transform=tra
```

```
0it [00:00, ?it/s]
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to hw10_data/cifar-10-pytho
```

```
100%|| 170049536/170498071 [00:27<00:00, 4855486.31it/s]

Files already downloaded and verified
```

Use `DataLoader` to create a loader for the training set and a loader for the testing set. You can use a `batch_size` of 8 to start, and change it if you wish.

```
In [4]: from torch.utils.data import DataLoader

        batch_size = 20
        train_loader = torch.utils.data.DataLoader(train_set, batch_size, shuffle=True, num_wo
        test_loader = torch.utils.data.DataLoader(test_set, batch_size, shuffle=True, num_worke

        input_shape = np.array(train_set[0][0]).shape
        input_dim = input_shape[1]*input_shape[2]*input_shape[0]

In [5]: training_epochs = 5

170500096it [00:40, 4855486.31it/s]

In [9]: #print(input_shape) (3, 32, 32)
        #print(input_dim) 3072
        #print(len(train_set)) 50000
        #print(len(test_set))  10000
```

1.4 Part 1 CIFAR10 with Fully Connected Neural Netowrk (25 Points)

As a warm-up, let's begin by training a two-layer fully connected neural network model on ** CIFAR-10** dataset. You may go back to check HW8 for some basics.

We will give you this code to use as a baseline to compare against your CNN models.

```
In [13]: # Run as administrator
         class TwoLayerModel(nn.Module):
           def __init__(self):
             super(TwoLayerModel, self).__init__()
             self.net = nn.Sequential(
               Flatten(),
               nn.Linear(input_dim, 64),
               nn.ReLU(),
               nn.Linear(64, 10))

           def forward(self, x):
             return self.net(x)

         model = TwoLayerModel().to(device)

         loss = nn.CrossEntropyLoss()
         optimizer = optim.RMSprop(model.parameters(), lr=0.001, weight_decay=0.01)
```

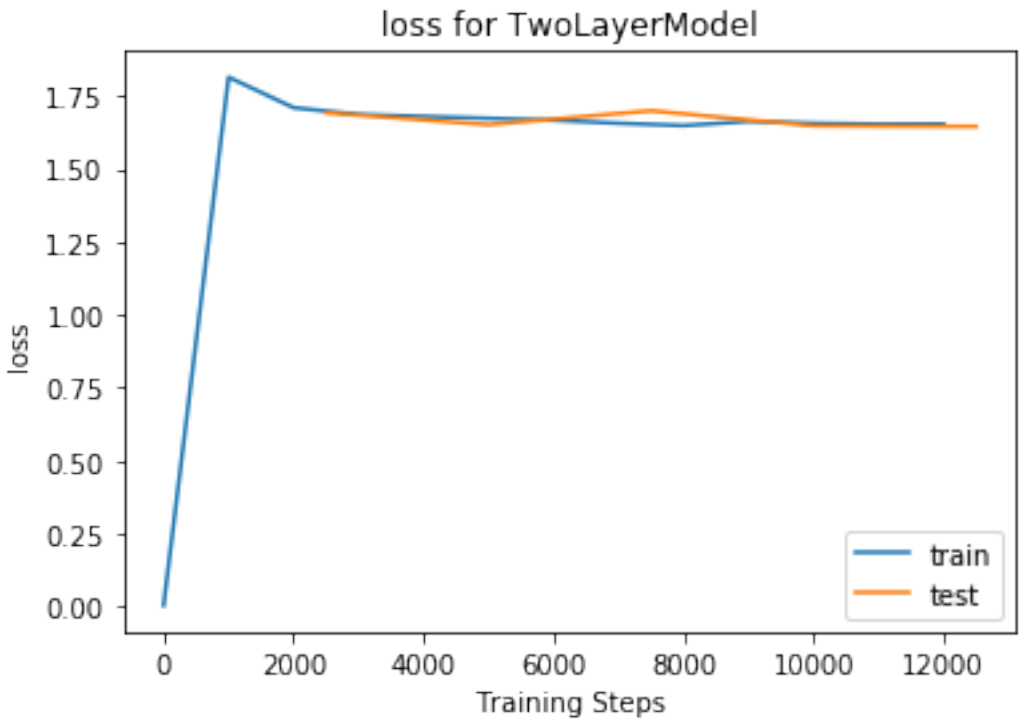
```
        # Training epoch should be about 15-20 sec each on GPU.
        metrics = train(model, train_loader, test_loader, loss, optimizer, training_epochs)
```

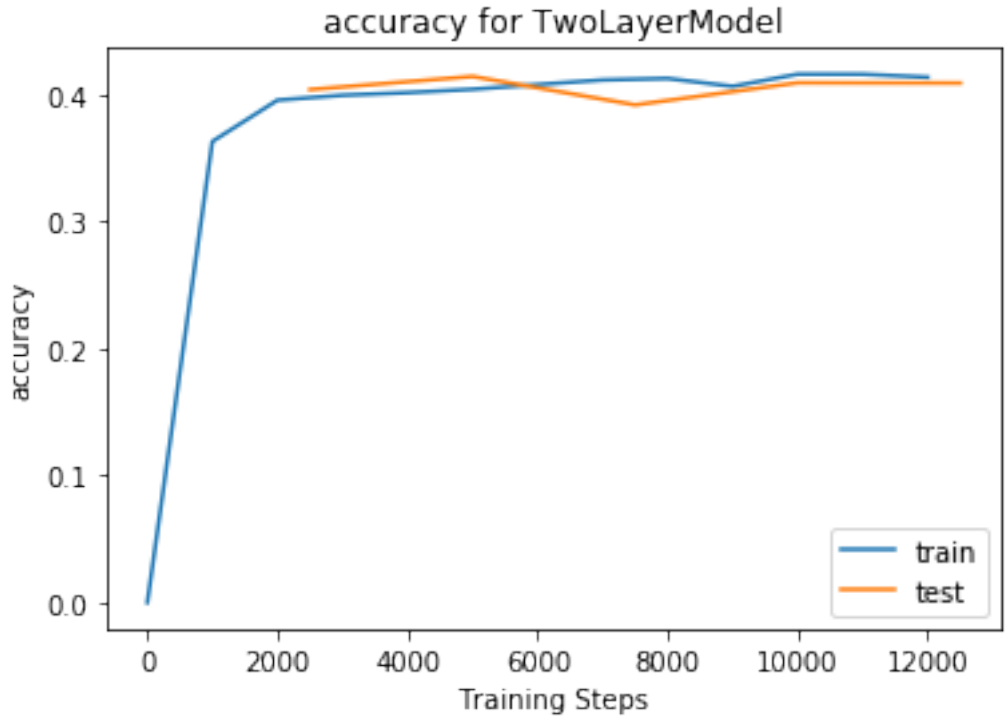
```
  Epoch 0 @ step 0: Train Loss: 0.004325, Train Accuracy: 0.000100
  Epoch 0 @ step 1000: Train Loss: 1.814288, Train Accuracy: 0.363200
  Epoch 0 @ step 2000: Train Loss: 1.709777, Train Accuracy: 0.395750
Epoch 0 Test Loss: 1.692053, Test Accuracy: 0.404100, time: 12.5s
  Epoch 1 @ step 3000: Train Loss: 1.686569, Train Accuracy: 0.399600
  Epoch 1 @ step 4000: Train Loss: 1.679454, Train Accuracy: 0.401650
Epoch 1 Test Loss: 1.651539, Test Accuracy: 0.414400, time: 12.2s
  Epoch 2 @ step 5000: Train Loss: 1.673214, Train Accuracy: 0.404300
  Epoch 2 @ step 6000: Train Loss: 1.666689, Train Accuracy: 0.408000
  Epoch 2 @ step 7000: Train Loss: 1.656478, Train Accuracy: 0.411649
Epoch 2 Test Loss: 1.699264, Test Accuracy: 0.392000, time: 12.0s
  Epoch 3 @ step 8000: Train Loss: 1.648354, Train Accuracy: 0.412800
  Epoch 3 @ step 9000: Train Loss: 1.663044, Train Accuracy: 0.406600
Epoch 3 Test Loss: 1.647666, Test Accuracy: 0.409400, time: 12.0s
  Epoch 4 @ step 10000: Train Loss: 1.656457, Train Accuracy: 0.416200
  Epoch 4 @ step 11000: Train Loss: 1.653537, Train Accuracy: 0.416150
  Epoch 4 @ step 12000: Train Loss: 1.653855, Train Accuracy: 0.413800
Epoch 4 Test Loss: 1.645478, Test Accuracy: 0.409200, time: 12.2s
Final Test Loss: 1.645478, Test Accuracy: 0.409200, Total time: 60.8s
```

Plot the model results

Normally we would want to use Tensorboard for looking at metrics. However, if colab reset while we are working, we might lose our logs and therefore our metrics. Let's just plot some graphs that will survive across colab instances.

```
In [14]: plot_graphs("TwoLayerModel", metrics)
```


loss for TwoLayerModel
loss
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
2000
4000
6000
8000
10000
12000
Training Steps
train
test


accuracy for TwoLayerModel
accuracy
0.4
0.3
0.2
0.1
0.0
0
2000
4000
6000
8000
10000
12000
Training Steps
train
test

1.5 Part 2 Convolutional Neural Network (CNN) (35 Points)

Now, let's design a convolution neural netwrok!

Build a simple CNN model, inserting 2 CNN layers in from of our 2 layer fully connect model from above:

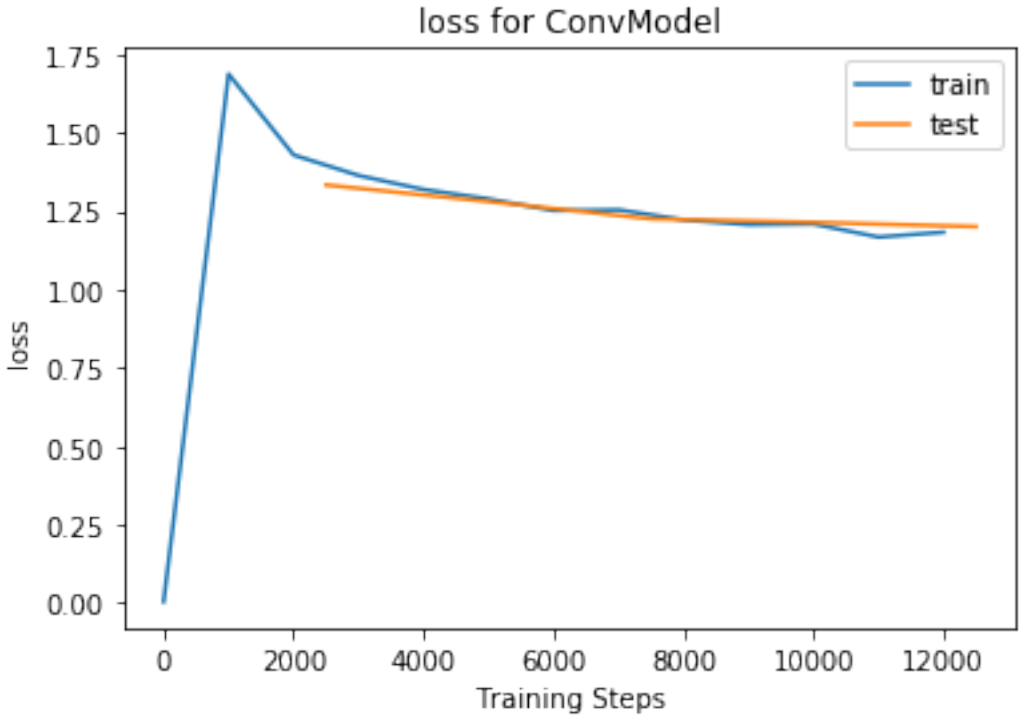
1. A convolution with 3x3 filter, 16 output channels, stride = 1, padding=1
2. A ReLU activation
3. A Max-Pooling layer with 2x2 window
4. A convolution, 3x3 filter, 16 output channels, stride = 1, padding=1
5. A ReLU activation
6. Flatten layer
7. Fully connected linear layer with output size 64
8. ReLU
9. Fully connected linear layer, with output size 10

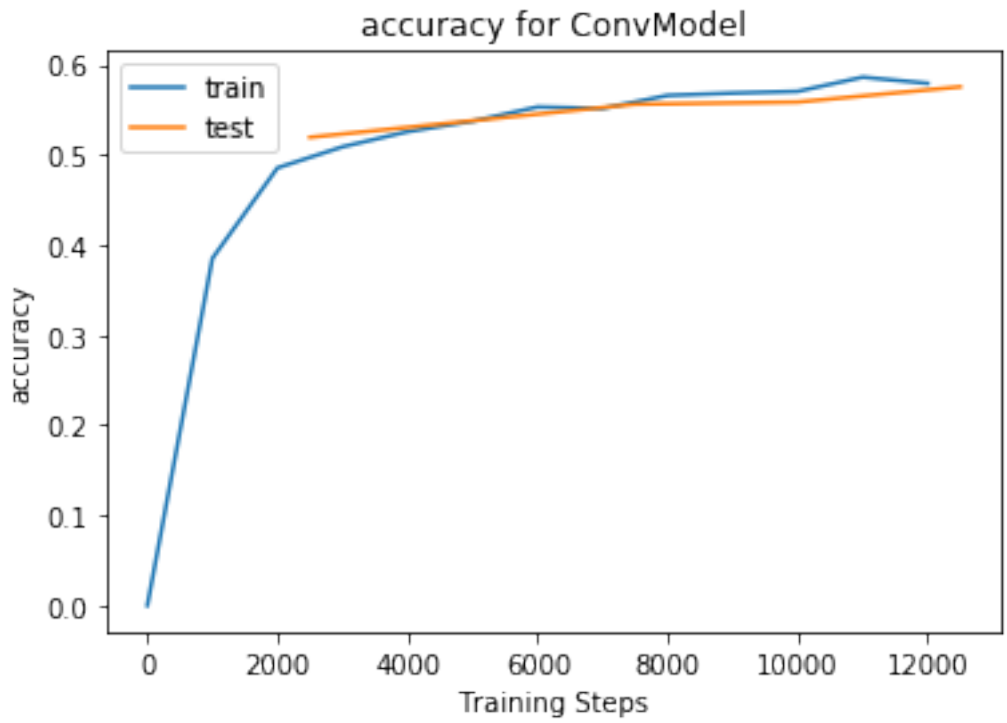
You will have to figure out the input sizes of the first fully connnected layer based on the previous layer sizes. Note that you also need to fill those in the report section (see report section in the notebook for details)

```
In [8]: class ConvModel(nn.Module):
          # Your Code Here
          def __init__(self):
            super(ConvModel, self).__init__()
            self.net = nn.Sequential(
              #Input channels = input_dim, output channels = 16
              nn.Conv2d(in_channels = 3, out_channels = 16, kernel_size = 3, stride = 1, paddi
              nn.ReLU(),
              nn.MaxPool2d(kernel_size=2, padding=0),
              nn.Conv2d(in_channels = 16, out_channels = 16, kernel_size = 3, stride = 1, padd
              nn.ReLU(),
              Flatten(),
              nn.Linear(16*16*16, 64),
              nn.ReLU(),
              nn.Linear(64, 10))

          def forward(self, x):
            return self.net(x)

        model = ConvModel().to(device)

        loss = nn.CrossEntropyLoss()
        optimizer = optim.RMSprop(model.parameters(), lr=0.001, weight_decay=0.01)

        metrics = train(model, train_loader, test_loader, loss, optimizer, training_epochs)

  Epoch 0 @ step 0: Train Loss: 0.004283, Train Accuracy: 0.000050
  Epoch 0 @ step 1000: Train Loss: 1.687688, Train Accuracy: 0.385050
  Epoch 0 @ step 2000: Train Loss: 1.429551, Train Accuracy: 0.485400
```

```
Epoch 0 Test Loss: 1.333801, Test Accuracy: 0.519300, time: 272.8s
  Epoch 1 @ step 3000: Train Loss: 1.364027, Train Accuracy: 0.508750
  Epoch 1 @ step 4000: Train Loss: 1.319344, Train Accuracy: 0.525551
Epoch 1 Test Loss: 1.280845, Test Accuracy: 0.537700, time: 339.3s
  Epoch 2 @ step 5000: Train Loss: 1.289175, Train Accuracy: 0.537150
  Epoch 2 @ step 6000: Train Loss: 1.252055, Train Accuracy: 0.552950
  Epoch 2 @ step 7000: Train Loss: 1.254503, Train Accuracy: 0.551350
Epoch 2 Test Loss: 1.225646, Test Accuracy: 0.556000, time: 351.3s
  Epoch 3 @ step 8000: Train Loss: 1.222496, Train Accuracy: 0.565750
  Epoch 3 @ step 9000: Train Loss: 1.207310, Train Accuracy: 0.568350
Epoch 3 Test Loss: 1.214683, Test Accuracy: 0.558400, time: 345.4s
  Epoch 4 @ step 10000: Train Loss: 1.209189, Train Accuracy: 0.570250
  Epoch 4 @ step 11000: Train Loss: 1.167576, Train Accuracy: 0.586000
  Epoch 4 @ step 12000: Train Loss: 1.183064, Train Accuracy: 0.579150
Epoch 4 Test Loss: 1.200890, Test Accuracy: 0.575300, time: 358.4s
Final Test Loss: 1.200890, Test Accuracy: 0.575300, Total time: 1667.2s
```

```
In [9]: plot_graphs("ConvModel", metrics)
```





Do you notice the improvement over the accuracy compared to that in Part 1?
Yes I noticed. The accuracy improved from 0.4078 to 0.5744.

1.6 Part 3 Open Design Competition (35 Points + 10 bonus points)

Try to beat the previous models by adding additional layers, changing parameters, etc. You should add at least one layer.

Possible changes include: * Dropout * Batch Normalization * More layers * Residual Connections (harder) * Change layer size * Pooling layers, stride * Different optimizer * Train for longer

Once you have a model you think is great, evaluate it against our hidden test data (see hidden_loader above) and upload the results to the leader board on gradescope. **The top 3 scorers will get a bonus 10 points.**

You can steal model structures found on the internet if you want. The only constraint is that **you must train the model from scratch**.

```
In [54]: asb_training_epochs = 10
         # You Awesome Super Best model code here
         class ConvModel2(nn.Module):
           # Your Code Here
           def __init__(self):
             super(ConvModel2, self).__init__()
             self.net = nn.Sequential(

               # Convolution with 64 different filters in size of (3x3)
               nn.Conv2d(in_channels = 3, out_channels = 64, kernel_size = 3, stride = 1, paddi
```

```
# Max Pooling by 2
nn.MaxPool2d(kernel_size=2, padding=0),
# - ReLU activation function
nn.ReLU(),
# - Batch Normalization
nn.BatchNorm2d(64),

# Convolution with 128 different filters in size of (3x3)
nn.Conv2d(in_channels = 64, out_channels = 128, kernel_size = 3, stride = 1, pad
# Max Pooling by 2
nn.MaxPool2d(kernel_size=2, padding=0),
# - ReLU activation function
nn.ReLU(),
# - Batch Normalization
nn.BatchNorm2d(128),

# Convolution with 256 different filters in size of (3x3)
nn.Conv2d(in_channels = 128, out_channels = 256, kernel_size = 3, stride = 1, pa
# Max Pooling by 2
nn.MaxPool2d(kernel_size=2, padding=0),
# - ReLU activation function
nn.ReLU(),
# - Batch Normalization
nn.BatchNorm2d(256),

# Convolution with 512 different filters in size of (3x3)
nn.Conv2d(in_channels = 256, out_channels = 512, kernel_size = 3, stride = 1, pa
# Max Pooling by 2
nn.MaxPool2d(kernel_size=2, padding=0),
# - ReLU activation function
nn.ReLU(),
# - Batch Normalization
nn.BatchNorm2d(512),

# Flattening the 3-D output of the last convolving operations.
Flatten(),

# Fully Connected Layer with 128 units
nn.Linear(2*2*512, 128),
nn.ReLU(),
# - Dropout
nn.Dropout(p = 0.25),
# - Batch Normalization
nn.BatchNorm1d(128),

# Fully Connected Layer with 256 units
nn.Linear(128, 256),
nn.ReLU(),
```

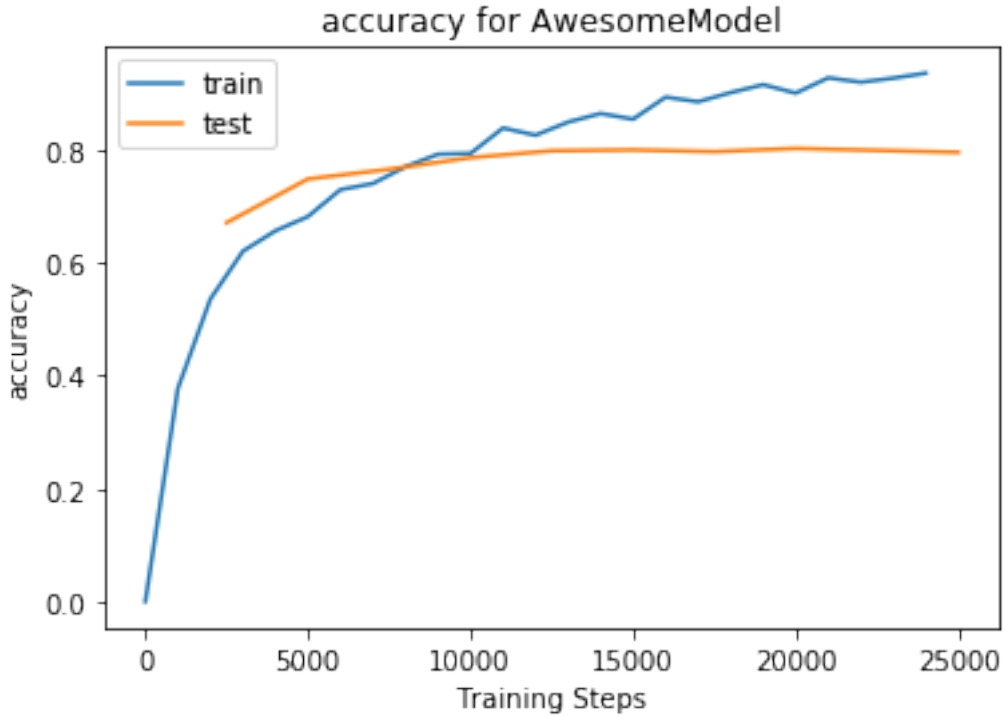
```
            # - Dropout
            nn.Dropout(p = 0.25),
            # - Batch Normalization
            nn.BatchNorm1d(256),

            # Fully Connected Layer with 512 units
            nn.Linear(256, 512),
            nn.ReLU(),
            # - Dropout
            nn.Dropout(p = 0.25),
            # - Batch Normalization
            nn.BatchNorm1d(512),

            # Fully Connected Layer with 1024 units
            nn.Linear(512, 1024),
            nn.ReLU(),
            # - Dropout
            nn.Dropout(p = 0.25),
            # - Batch Normalization
            nn.BatchNorm1d(1024),

            nn.Linear(1024, 10))

        def forward(self, x):
          return self.net(x)

    model3 = ConvModel2().to(device)

    loss = nn.CrossEntropyLoss()
    optimizer = optim.RMSprop(model3.parameters(), lr=0.001, weight_decay=1e-6)

    metrics = train(model3, train_loader, test_loader, loss, optimizer, asb_training_epoch
```

```
  Epoch 0 @ step 0: Train Loss: 0.004413, Train Accuracy: 0.000150
  Epoch 0 @ step 1000: Train Loss: 1.699582, Train Accuracy: 0.377650
  Epoch 0 @ step 2000: Train Loss: 1.313297, Train Accuracy: 0.536250
Epoch 0 Test Loss: 0.958509, Test Accuracy: 0.671500, time: 57.9s
  Epoch 1 @ step 3000: Train Loss: 1.099963, Train Accuracy: 0.620900
  Epoch 1 @ step 4000: Train Loss: 1.007975, Train Accuracy: 0.657050
Epoch 1 Test Loss: 0.741405, Test Accuracy: 0.748900, time: 59.3s
  Epoch 2 @ step 5000: Train Loss: 0.942329, Train Accuracy: 0.682350
  Epoch 2 @ step 6000: Train Loss: 0.804467, Train Accuracy: 0.730001
  Epoch 2 @ step 7000: Train Loss: 0.778122, Train Accuracy: 0.740850
Epoch 2 Test Loss: 0.687638, Test Accuracy: 0.766100, time: 61.8s
  Epoch 3 @ step 8000: Train Loss: 0.682857, Train Accuracy: 0.770599
  Epoch 3 @ step 9000: Train Loss: 0.625800, Train Accuracy: 0.792750
Epoch 3 Test Loss: 0.643632, Test Accuracy: 0.786700, time: 58.1s
  Epoch 4 @ step 10000: Train Loss: 0.633456, Train Accuracy: 0.794099
```

```
  Epoch 4 @ step 11000: Train Loss: 0.495053, Train Accuracy: 0.839200
  Epoch 4 @ step 12000: Train Loss: 0.520816, Train Accuracy: 0.826450
Epoch 4 Test Loss: 0.598472, Test Accuracy: 0.798900, time: 63.8s
  Epoch 5 @ step 13000: Train Loss: 0.450866, Train Accuracy: 0.849751
  Epoch 5 @ step 14000: Train Loss: 0.411425, Train Accuracy: 0.864850
Epoch 5 Test Loss: 0.614346, Test Accuracy: 0.800400, time: 63.4s
  Epoch 6 @ step 15000: Train Loss: 0.438493, Train Accuracy: 0.854900
  Epoch 6 @ step 16000: Train Loss: 0.323710, Train Accuracy: 0.893952
  Epoch 6 @ step 17000: Train Loss: 0.346614, Train Accuracy: 0.885753
Epoch 6 Test Loss: 0.643161, Test Accuracy: 0.797400, time: 61.3s
  Epoch 7 @ step 18000: Train Loss: 0.304251, Train Accuracy: 0.901953
  Epoch 7 @ step 19000: Train Loss: 0.265824, Train Accuracy: 0.916004
Epoch 7 Test Loss: 0.635537, Test Accuracy: 0.803000, time: 61.9s
  Epoch 8 @ step 20000: Train Loss: 0.293848, Train Accuracy: 0.901052
  Epoch 8 @ step 21000: Train Loss: 0.216530, Train Accuracy: 0.928203
  Epoch 8 @ step 22000: Train Loss: 0.244811, Train Accuracy: 0.920404
Epoch 8 Test Loss: 0.652733, Test Accuracy: 0.799600, time: 60.8s
  Epoch 9 @ step 23000: Train Loss: 0.222258, Train Accuracy: 0.927603
  Epoch 9 @ step 24000: Train Loss: 0.203590, Train Accuracy: 0.936404
Epoch 9 Test Loss: 0.672336, Test Accuracy: 0.796200, time: 61.3s
Final Test Loss: 0.672336, Test Accuracy: 0.796200, Total time: 609.5s
```

What changes did you make to improve your model?

```
In [55]: plot_graphs("AwesomeModel", metrics)
```



After you get a nice model, download the test_file.zip and unzip it to get test_file.pt. In colab, you can explore your files from the left side bar. You can also download the files to your machine from there.

```
In [ ]: #!wget http://courses.engr.illinois.edu/cs498aml/sp2019/homeworks/test_file.zip
        #!unzip test_file.zip
```

Then use your model to predict the label of the test images. Fill the remaining code below, where x has two dimensions (batch_size x one image size). Remember to reshpe x accordingly before feeding it into your model. The submission.txt should contain one predicted label (0~9) each line. Submit your submission.txt to the competition in gradscope.

```
In [56]: import torch.utils.data as Data

         test_file = 'test_file.pt'
         pred_file = 'submission.txt'

         f_pred = open(pred_file,'w')
         tensor = torch.load(test_file)
         torch_dataset = Data.TensorDataset(tensor)
         test_loader2 = torch.utils.data.DataLoader(torch_dataset, batch_size, shuffle=False, r
```

```
In [57]: model3.eval()
         for ele in test_loader2:
           x = ele[0]
           # Fill your code here
           x_reshaped = x.reshape((batch_size, 3, 32, 32)).to(device)
           output = model3(x_reshaped)
           labels = torch.argmax(output, dim = 1)
           for i in range(batch_size):
             f_pred.write(str(labels[i].item()) + '\n')
           #f_pred.write('\n')
         f_pred.close()
```

2 Report

2.1 Part 0: Imports and Basic Setup (5 Points)

Nothing to report for this part. You will be just scored for finishing the setup.

2.2 Part 1: Fully connected neural networks (25 Points)

Test (on validation set) accuracy (5 Points): 0.409200
Test loss (5 Points): 1.645478
Training time (5 Points): 60.8s
Plots:

- Plot a graph of accuracy on validation set vs training steps (5 Points)
- Plot a graph of loss on validation set vs training steps (5 Points)

See figure of TwoLayerModel above ## Part 2: Convolution Network (Basic) (35 Points)
Tensor dimensions: A good way to debug your network for size mismatches is to print the dimension of output after every layers:
(10 Points)
Output dimension after 1st conv layer: 16, 32, 32
Output dimension after 1st max pooling: 16, 16, 16
Output dimension after 2nd conv layer: 16, 16, 16
Output dimension after flatten layer: 4096
Output dimension after 1st fully connected layer: 64
Output dimension after 2nd fully connected layer: 10
Test (on validation set) Accuracy (5 Points): 0.537600
Test loss (5 Points): 1.297206
Training time (5 Points): 1427.5s
Plots:

- Plot a graph of accuracy on validation set vs training steps (5 Points)
- Plot a graph of loss on validation set vs training steps (5 Points)

See figure of ConvModel above

2.3 Part 3: Convolution Network (Add one or more suggested changes) (35 Points)

Describe the additional changes implemented, your intuition for as to why it works, you may also describe other approaches you experimented with (10 Points):

We have added a repeated pattern of layer: Convolution, Maxpooling, Relu, Batch normalization about 4 times, and then repeated a new sequence of Fully connected, Relu, Drop out, Batch normalization another 4 times. Our intuition is that with more than more nerons and layers, it is more accurate to delineate a class.

Test (on validation set) Accuracy (5 Points): 0.798600

Test loss (5 Points): 0.675462

Training time (5 Points): 7685.2s

Plots:

- Plot a graph of accuracy on validation set vs training steps (5 Points)
- Plot a graph of loss on validation set vs training steps (5 Points)

See figure of AwesomeModel above.

10 bonus points will be awarded to top 3 scorers on leaderboard (in case of tie for 3rd position everyone tied for 3rd position will get the bonus)