Homework 10 - CIFAR10 Image Classification with PyTorch

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.

Dev Environment

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 (https://colab.research.google.com/drive)](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**.

Installing PyTorch and Dependencies

The instructions for installing and setting up PyTorch can be found at [https://pytorch.org/get-started/locally/ (https://pytorch.org/get-started/locally/)](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([[0.6727, 0.6112, 0.5345],
        [0.7707, 0.0481, 0.9676],
        [0.3093, 0.4211, 0.1461],
        [0.6003, 0.3805, 0.9057],
        [0.9781, 0.3022, 0.5507]])
```

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 [0]:
```
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 [3]:
```
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)
```

```
cuda:0
```

Training Code

In [0]:
```
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() # set previos gradients zero

      # forward > loss > backward > update
      preds = model(images)
      loss = loss_func(preds, labels)
      loss.backward()
      opt.step()

      # calculate train loss & accuracy
      training_loss += loss.item()
      training_acc += (torch.argmax(preds, dim=1)==labels).float().mean()

      # log & print train loss & accuracy
      if training_step % print_every == 0:
        training_loss /= print_every
        training_acc /= print_every

        all_training_loss = np.concatenate((all_training_loss, [[training_step, training
_loss]]))
        all_training_acc = np.concatenate((all_training_acc, [[training_step, training_a
cc]]))

        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

    # calculate validation loss & accuracy on test data
    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 [0]:

```
!mkdir hw10_data
```

In [6]:

```
# 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=transformations
)
test_set = datasets.CIFAR10(root='hw10_data', download=True, train=False, transform=tran
sformations)
```

```
  0%|          | 0/170498071 [00:00<?, ?it/s]

Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to hw10_data/cifar-1
0-python.tar.gz

100%|██████████| 170393600/170498071 [00:37<00:00, 3442228.26it/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 [0]:
```
from torch.utils.data import DataLoader

batch_size = 32
train_loader = torch.utils.data.DataLoader(train_set, batch_size, shuffle=True, num_work
ers=2)
test_loader = torch.utils.data.DataLoader(test_set, batch_size, shuffle=True, num_worker
s=2)

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

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.

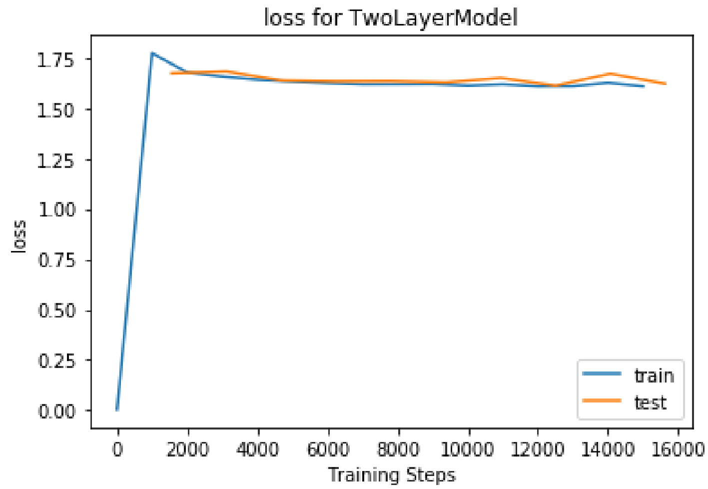
In [8]:
```
class TwoLayerModel(nn.Module):
  def __init__(self):
    super(TwoLayerModel, self).__init__()
    self.net = nn.Sequential(
      Flatten(),
      nn.Linear(input_dim, 64),
      nn.ReLU(),
      nn.Linear(64, 10))

  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, num_epochs=10)
```

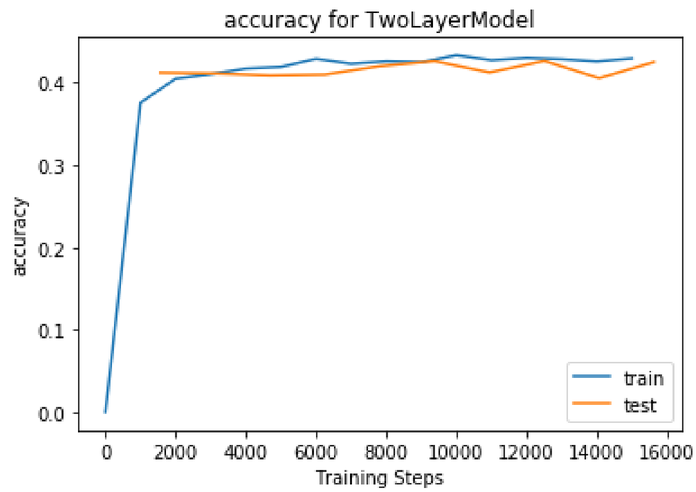
```
  Epoch 0 @ step 0: Train Loss: 0.004287, Train Accuracy: 0.000125
  Epoch 0 @ step 1000: Train Loss: 1.772654, Train Accuracy: 0.375625
Epoch 0 Test Loss: 1.672711, Test Accuracy: 0.412041, time: 5.7s
  Epoch 1 @ step 2000: Train Loss: 1.676743, Train Accuracy: 0.404875
  Epoch 1 @ step 3000: Train Loss: 1.656414, Train Accuracy: 0.410250
Epoch 1 Test Loss: 1.681461, Test Accuracy: 0.411242, time: 5.7s
  Epoch 2 @ step 4000: Train Loss: 1.640921, Train Accuracy: 0.417125
Epoch 2 Test Loss: 1.636957, Test Accuracy: 0.408846, time: 5.4s
  Epoch 3 @ step 5000: Train Loss: 1.632107, Train Accuracy: 0.419094
  Epoch 3 @ step 6000: Train Loss: 1.624097, Train Accuracy: 0.428875
Epoch 3 Test Loss: 1.632707, Test Accuracy: 0.409844, time: 5.5s
  Epoch 4 @ step 7000: Train Loss: 1.618297, Train Accuracy: 0.422906
Epoch 4 Test Loss: 1.633762, Test Accuracy: 0.420028, time: 5.7s
  Epoch 5 @ step 8000: Train Loss: 1.618586, Train Accuracy: 0.425875
  Epoch 5 @ step 9000: Train Loss: 1.619208, Train Accuracy: 0.424844
Epoch 5 Test Loss: 1.627341, Test Accuracy: 0.426118, time: 5.4s
  Epoch 6 @ step 10000: Train Loss: 1.611559, Train Accuracy: 0.433313
Epoch 6 Test Loss: 1.649090, Test Accuracy: 0.412440, time: 5.3s
  Epoch 7 @ step 11000: Train Loss: 1.618335, Train Accuracy: 0.427250
  Epoch 7 @ step 12000: Train Loss: 1.607949, Train Accuracy: 0.430031
Epoch 7 Test Loss: 1.611719, Test Accuracy: 0.426318, time: 5.3s
  Epoch 8 @ step 13000: Train Loss: 1.608835, Train Accuracy: 0.428469
  Epoch 8 @ step 14000: Train Loss: 1.625498, Train Accuracy: 0.425719
Epoch 8 Test Loss: 1.670497, Test Accuracy: 0.405551, time: 5.3s
  Epoch 9 @ step 15000: Train Loss: 1.608230, Train Accuracy: 0.429375
Epoch 9 Test Loss: 1.620814, Test Accuracy: 0.425020, time: 5.4s
Final Test Loss: 1.620814, Test Accuracy: 0.425020, Total time: 54.9s
```

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 [9]:
```
plot_graphs("TwoLayerModel", metrics)
```





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

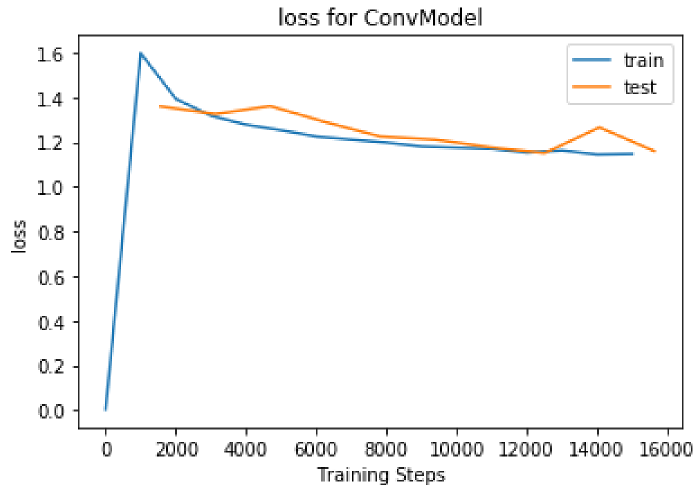
You will have to figure out the input sizes of the first fully connnected layer based on the previous layer sizes. Note that you also need to fill those in the report section (see report section in the notebook for details)

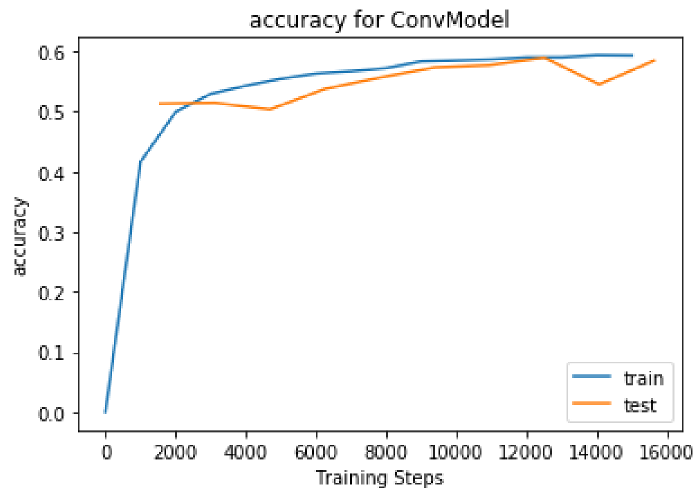
In [0]:
```
import torch.nn.functional as F
class ConvModel(nn.Module):
  # Your Code Here
  def __init__(self):
    super(ConvModel, self).__init__()

    self.net = nn.Sequential(
        nn.Conv2d(3,16,3,1,1),
        nn.ReLU(),
        nn.MaxPool2d(2,2),
        nn.Conv2d(16,16,3,1,1),
        nn.ReLU(),
        Flatten(),
        nn.Linear(4096, 64),
        nn.ReLU(),
        nn.Linear(64, 10)
    )

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

In [11]:
```
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, num_epochs=10)
```

```
  Epoch 0 @ step 0: Train Loss: 0.004307, Train Accuracy: 0.000094
  Epoch 0 @ step 1000: Train Loss: 1.597196, Train Accuracy: 0.416281
Epoch 0 Test Loss: 1.358723, Test Accuracy: 0.512979, time: 7.3s
  Epoch 1 @ step 2000: Train Loss: 1.392143, Train Accuracy: 0.499594
  Epoch 1 @ step 3000: Train Loss: 1.317306, Train Accuracy: 0.529344
Epoch 1 Test Loss: 1.324693, Test Accuracy: 0.514377, time: 7.5s
  Epoch 2 @ step 4000: Train Loss: 1.277293, Train Accuracy: 0.542969
Epoch 2 Test Loss: 1.359997, Test Accuracy: 0.503594, time: 7.5s
  Epoch 3 @ step 5000: Train Loss: 1.252705, Train Accuracy: 0.554594
  Epoch 3 @ step 6000: Train Loss: 1.224263, Train Accuracy: 0.563219
Epoch 3 Test Loss: 1.289326, Test Accuracy: 0.537640, time: 7.5s
  Epoch 4 @ step 7000: Train Loss: 1.210711, Train Accuracy: 0.567156
Epoch 4 Test Loss: 1.224806, Test Accuracy: 0.556510, time: 8.2s
  Epoch 5 @ step 8000: Train Loss: 1.197062, Train Accuracy: 0.572344
  Epoch 5 @ step 9000: Train Loss: 1.180133, Train Accuracy: 0.583531
Epoch 5 Test Loss: 1.210565, Test Accuracy: 0.573482, time: 7.5s
  Epoch 6 @ step 10000: Train Loss: 1.174629, Train Accuracy: 0.585219
Epoch 6 Test Loss: 1.176760, Test Accuracy: 0.577276, time: 7.5s
  Epoch 7 @ step 11000: Train Loss: 1.169524, Train Accuracy: 0.586844
  Epoch 7 @ step 12000: Train Loss: 1.152071, Train Accuracy: 0.590156
Epoch 7 Test Loss: 1.150153, Test Accuracy: 0.589657, time: 7.5s
  Epoch 8 @ step 13000: Train Loss: 1.161712, Train Accuracy: 0.590313
  Epoch 8 @ step 14000: Train Loss: 1.143967, Train Accuracy: 0.593813
Epoch 8 Test Loss: 1.265373, Test Accuracy: 0.545327, time: 7.5s
  Epoch 9 @ step 15000: Train Loss: 1.146474, Train Accuracy: 0.593313
Epoch 9 Test Loss: 1.159240, Test Accuracy: 0.584864, time: 7.5s
Final Test Loss: 1.159240, Test Accuracy: 0.584864, Total time: 75.7s
```

In [12]:
```
plot_graphs("ConvModel", metrics)
```





Do you notice the improvement over the accuracy compared to that in Part 1?

Part 1: Test Accuracy: **0.425020**

Part 2: Test Accuracy: **0.584864**

Yes, the accuracy of the model improved by adding convolutional layers.

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 [0]:
```
# You Awesome Super Best model code here

# Used model structure from this post and changed a little bit
# https://github.com/BeierZhu/CIFAR-10_Pytorch/blob/master/Vgg.py

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

    self.net = nn.Sequential(

        # stage 1
        nn.Conv2d(3, 64, 3, padding=1),
        nn.BatchNorm2d(64),
        nn.ReLU(),
        nn.Dropout2d(p=0.3),

        nn.Conv2d(64,64, 3, padding=1),
        nn.BatchNorm2d(64),
        nn.ReLU(),
        nn.MaxPool2d(2, 2),

        # stage 2
        nn.Conv2d(64, 128, 3, padding=1),
        nn.BatchNorm2d(128),
        nn.ReLU(),
        nn.Dropout2d(p=0.4),

        nn.Conv2d(128, 128, 3, padding=1),
        nn.BatchNorm2d(128),
        nn.ReLU(),
        nn.MaxPool2d(2, 2),

        # Stage 3
        nn.Conv2d(128, 256, 3, padding=1),
        nn.BatchNorm2d(256),
        nn.ReLU(),
        nn.Dropout2d(p=0.4),

        nn.Conv2d(256, 256, 3, padding=1),
        nn.BatchNorm2d(256),
        nn.ReLU(),
        nn.MaxPool2d(2, 2),

        # Stage 4
        nn.Conv2d(256, 512, 3, padding=1),
        nn.BatchNorm2d(512),
        nn.ReLU(),
        nn.Dropout2d(p=0.4),

        nn.Conv2d(512, 512, 3, padding=1),
        nn.BatchNorm2d(512),
        nn.ReLU(),
        nn.MaxPool2d(2, 2),

        nn.Conv2d(512, 512, 3, padding=1),
        nn.BatchNorm2d(512),
        nn.ReLU(),
        nn.MaxPool2d(2, 2),

        # stage 5

        Flatten(),
```

```
        nn.Dropout(p=0.5),
        nn.Linear(512, 512),
        nn.ReLU(),

        nn.Dropout(p=0.4),
        nn.Linear(512, 256),
        nn.ReLU(),

        nn.Dropout(p=0.3),
        nn.Linear(256, 10),
    )

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

In [0]:
```
model = AwesomeModel().to(device)
```

In [15]:
```
loss = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001,momentum=0.9)

metrics = train(model, train_loader, test_loader, loss, optimizer, num_epochs=60)
```

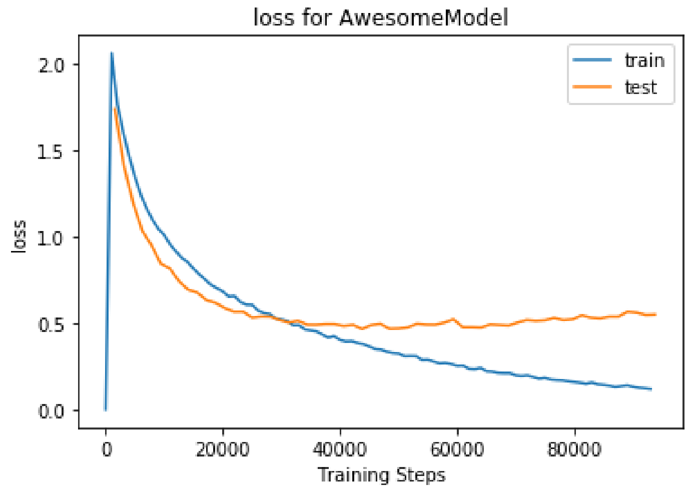
```
  Epoch 0 @ step 0: Train Loss: 0.004316, Train Accuracy: 0.000094
  Epoch 0 @ step 1000: Train Loss: 2.063875, Train Accuracy: 0.195219
Epoch 0 Test Loss: 1.739872, Test Accuracy: 0.328974, time: 38.0s
  Epoch 1 @ step 2000: Train Loss: 1.766375, Train Accuracy: 0.315219
  Epoch 1 @ step 3000: Train Loss: 1.607049, Train Accuracy: 0.389719
Epoch 1 Test Loss: 1.411071, Test Accuracy: 0.461262, time: 38.5s
  Epoch 2 @ step 4000: Train Loss: 1.468229, Train Accuracy: 0.452188
Epoch 2 Test Loss: 1.195680, Test Accuracy: 0.567392, time: 38.1s
  Epoch 3 @ step 5000: Train Loss: 1.345738, Train Accuracy: 0.505906
  Epoch 3 @ step 6000: Train Loss: 1.243075, Train Accuracy: 0.549656
Epoch 3 Test Loss: 1.035183, Test Accuracy: 0.625499, time: 38.4s
  Epoch 4 @ step 7000: Train Loss: 1.164685, Train Accuracy: 0.582969
Epoch 4 Test Loss: 0.953198, Test Accuracy: 0.663538, time: 38.3s
  Epoch 5 @ step 8000: Train Loss: 1.099700, Train Accuracy: 0.605750
  Epoch 5 @ step 9000: Train Loss: 1.047218, Train Accuracy: 0.626906
Epoch 5 Test Loss: 0.847232, Test Accuracy: 0.696086, time: 38.2s
  Epoch 6 @ step 10000: Train Loss: 1.011963, Train Accuracy: 0.636438
Epoch 6 Test Loss: 0.819198, Test Accuracy: 0.710064, time: 38.3s
  Epoch 7 @ step 11000: Train Loss: 0.957615, Train Accuracy: 0.661875
  Epoch 7 @ step 12000: Train Loss: 0.916926, Train Accuracy: 0.677094
Epoch 7 Test Loss: 0.745702, Test Accuracy: 0.736522, time: 38.4s
  Epoch 8 @ step 13000: Train Loss: 0.880312, Train Accuracy: 0.693156
  Epoch 8 @ step 14000: Train Loss: 0.855204, Train Accuracy: 0.701875
Epoch 8 Test Loss: 0.696890, Test Accuracy: 0.757987, time: 38.2s
  Epoch 9 @ step 15000: Train Loss: 0.816427, Train Accuracy: 0.715844
Epoch 9 Test Loss: 0.680104, Test Accuracy: 0.768171, time: 38.3s
  Epoch 10 @ step 16000: Train Loss: 0.784167, Train Accuracy: 0.732125
  Epoch 10 @ step 17000: Train Loss: 0.753756, Train Accuracy: 0.742656
Epoch 10 Test Loss: 0.636115, Test Accuracy: 0.782947, time: 38.2s
  Epoch 11 @ step 18000: Train Loss: 0.723888, Train Accuracy: 0.749125
Epoch 11 Test Loss: 0.620387, Test Accuracy: 0.789237, time: 38.2s
  Epoch 12 @ step 19000: Train Loss: 0.704943, Train Accuracy: 0.760344
  Epoch 12 @ step 20000: Train Loss: 0.686425, Train Accuracy: 0.767750
Epoch 12 Test Loss: 0.590109, Test Accuracy: 0.801218, time: 38.2s
  Epoch 13 @ step 21000: Train Loss: 0.657173, Train Accuracy: 0.775781
Epoch 13 Test Loss: 0.568881, Test Accuracy: 0.805312, time: 38.3s
  Epoch 14 @ step 22000: Train Loss: 0.660146, Train Accuracy: 0.776031
  Epoch 14 @ step 23000: Train Loss: 0.625189, Train Accuracy: 0.787781
Epoch 14 Test Loss: 0.569213, Test Accuracy: 0.809405, time: 38.3s
  Epoch 15 @ step 24000: Train Loss: 0.609768, Train Accuracy: 0.795938
  Epoch 15 @ step 25000: Train Loss: 0.608535, Train Accuracy: 0.793094
Epoch 15 Test Loss: 0.535032, Test Accuracy: 0.821585, time: 38.3s
  Epoch 16 @ step 26000: Train Loss: 0.575955, Train Accuracy: 0.806438
Epoch 16 Test Loss: 0.541370, Test Accuracy: 0.817192, time: 38.2s
  Epoch 17 @ step 27000: Train Loss: 0.560903, Train Accuracy: 0.810250
  Epoch 17 @ step 28000: Train Loss: 0.555373, Train Accuracy: 0.808531
Epoch 17 Test Loss: 0.540128, Test Accuracy: 0.821286, time: 38.3s
  Epoch 18 @ step 29000: Train Loss: 0.530582, Train Accuracy: 0.820813
Epoch 18 Test Loss: 0.520931, Test Accuracy: 0.831470, time: 38.2s
  Epoch 19 @ step 30000: Train Loss: 0.523688, Train Accuracy: 0.823656
  Epoch 19 @ step 31000: Train Loss: 0.515785, Train Accuracy: 0.826625
Epoch 19 Test Loss: 0.506369, Test Accuracy: 0.831070, time: 38.2s
  Epoch 20 @ step 32000: Train Loss: 0.490486, Train Accuracy: 0.834781
Epoch 20 Test Loss: 0.516765, Test Accuracy: 0.829972, time: 38.2s
  Epoch 21 @ step 33000: Train Loss: 0.491086, Train Accuracy: 0.833125
  Epoch 21 @ step 34000: Train Loss: 0.466275, Train Accuracy: 0.840031
Epoch 21 Test Loss: 0.493606, Test Accuracy: 0.838758, time: 38.2s
  Epoch 22 @ step 35000: Train Loss: 0.460906, Train Accuracy: 0.842906
Epoch 22 Test Loss: 0.493132, Test Accuracy: 0.837859, time: 38.3s
  Epoch 23 @ step 36000: Train Loss: 0.455245, Train Accuracy: 0.845813
  Epoch 23 @ step 37000: Train Loss: 0.436713, Train Accuracy: 0.850125
Epoch 23 Test Loss: 0.497351, Test Accuracy: 0.839856, time: 38.5s
  Epoch 24 @ step 38000: Train Loss: 0.419437, Train Accuracy: 0.855906
```

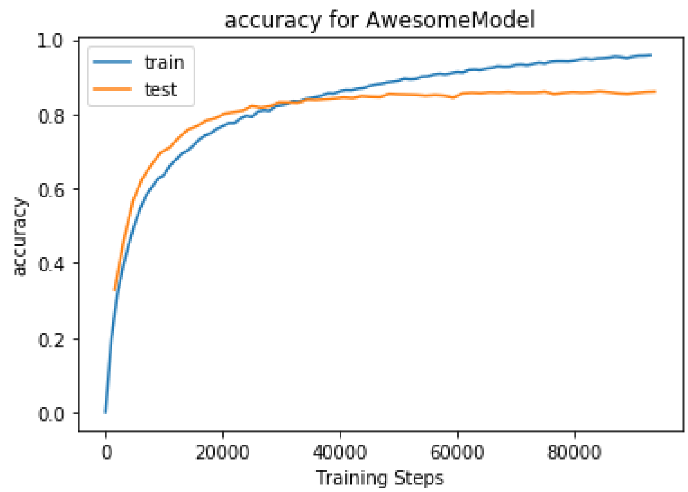
```
  Epoch 24 @ step 39000: Train Loss: 0.427867, Train Accuracy: 0.855250
Epoch 24 Test Loss: 0.497224, Test Accuracy: 0.841753, time: 38.2s
  Epoch 25 @ step 40000: Train Loss: 0.405917, Train Accuracy: 0.861000
Epoch 25 Test Loss: 0.486178, Test Accuracy: 0.844649, time: 38.3s
  Epoch 26 @ step 41000: Train Loss: 0.397232, Train Accuracy: 0.864531
  Epoch 26 @ step 42000: Train Loss: 0.398492, Train Accuracy: 0.864094
Epoch 26 Test Loss: 0.492924, Test Accuracy: 0.842851, time: 38.3s
  Epoch 27 @ step 43000: Train Loss: 0.387656, Train Accuracy: 0.867938
Epoch 27 Test Loss: 0.469240, Test Accuracy: 0.848343, time: 38.2s
  Epoch 28 @ step 44000: Train Loss: 0.381261, Train Accuracy: 0.869719
  Epoch 28 @ step 45000: Train Loss: 0.367959, Train Accuracy: 0.875000
Epoch 28 Test Loss: 0.491518, Test Accuracy: 0.846546, time: 38.2s
  Epoch 29 @ step 46000: Train Loss: 0.353058, Train Accuracy: 0.879000
Epoch 29 Test Loss: 0.498740, Test Accuracy: 0.845747, time: 38.5s
  Epoch 30 @ step 47000: Train Loss: 0.349537, Train Accuracy: 0.881063
  Epoch 30 @ step 48000: Train Loss: 0.337567, Train Accuracy: 0.884719
Epoch 30 Test Loss: 0.470260, Test Accuracy: 0.854233, time: 38.5s
  Epoch 31 @ step 49000: Train Loss: 0.330268, Train Accuracy: 0.887156
  Epoch 31 @ step 50000: Train Loss: 0.327298, Train Accuracy: 0.889344
Epoch 31 Test Loss: 0.471880, Test Accuracy: 0.852835, time: 38.3s
  Epoch 32 @ step 51000: Train Loss: 0.311669, Train Accuracy: 0.895438
Epoch 32 Test Loss: 0.478210, Test Accuracy: 0.852137, time: 38.1s
  Epoch 33 @ step 52000: Train Loss: 0.312826, Train Accuracy: 0.893500
  Epoch 33 @ step 53000: Train Loss: 0.312008, Train Accuracy: 0.894719
Epoch 33 Test Loss: 0.499167, Test Accuracy: 0.851338, time: 38.3s
  Epoch 34 @ step 54000: Train Loss: 0.289930, Train Accuracy: 0.900219
Epoch 34 Test Loss: 0.494047, Test Accuracy: 0.849241, time: 38.1s
  Epoch 35 @ step 55000: Train Loss: 0.291349, Train Accuracy: 0.901125
  Epoch 35 @ step 56000: Train Loss: 0.280548, Train Accuracy: 0.905813
Epoch 35 Test Loss: 0.493511, Test Accuracy: 0.851238, time: 38.1s
  Epoch 36 @ step 57000: Train Loss: 0.270425, Train Accuracy: 0.908281
Epoch 36 Test Loss: 0.505886, Test Accuracy: 0.849541, time: 38.2s
  Epoch 37 @ step 58000: Train Loss: 0.272724, Train Accuracy: 0.906438
  Epoch 37 @ step 59000: Train Loss: 0.266223, Train Accuracy: 0.909531
Epoch 37 Test Loss: 0.525338, Test Accuracy: 0.844050, time: 38.2s
  Epoch 38 @ step 60000: Train Loss: 0.255988, Train Accuracy: 0.912938
Epoch 38 Test Loss: 0.479198, Test Accuracy: 0.855531, time: 38.2s
  Epoch 39 @ step 61000: Train Loss: 0.257142, Train Accuracy: 0.911688
  Epoch 39 @ step 62000: Train Loss: 0.238708, Train Accuracy: 0.918750
Epoch 39 Test Loss: 0.479320, Test Accuracy: 0.857129, time: 38.2s
  Epoch 40 @ step 63000: Train Loss: 0.236176, Train Accuracy: 0.919875
  Epoch 40 @ step 64000: Train Loss: 0.243539, Train Accuracy: 0.918469
Epoch 40 Test Loss: 0.477517, Test Accuracy: 0.856030, time: 38.2s
  Epoch 41 @ step 65000: Train Loss: 0.225087, Train Accuracy: 0.922188
Epoch 41 Test Loss: 0.494937, Test Accuracy: 0.858427, time: 38.1s
  Epoch 42 @ step 66000: Train Loss: 0.222748, Train Accuracy: 0.924219
  Epoch 42 @ step 67000: Train Loss: 0.215543, Train Accuracy: 0.928125
Epoch 42 Test Loss: 0.491899, Test Accuracy: 0.857728, time: 38.1s
  Epoch 43 @ step 68000: Train Loss: 0.214034, Train Accuracy: 0.926750
Epoch 43 Test Loss: 0.488777, Test Accuracy: 0.859425, time: 38.1s
  Epoch 44 @ step 69000: Train Loss: 0.214166, Train Accuracy: 0.927094
  Epoch 44 @ step 70000: Train Loss: 0.201180, Train Accuracy: 0.932063
Epoch 44 Test Loss: 0.506589, Test Accuracy: 0.857428, time: 38.2s
  Epoch 45 @ step 71000: Train Loss: 0.197636, Train Accuracy: 0.932906
Epoch 45 Test Loss: 0.521037, Test Accuracy: 0.857528, time: 38.2s
  Epoch 46 @ step 72000: Train Loss: 0.201111, Train Accuracy: 0.931031
  Epoch 46 @ step 73000: Train Loss: 0.191837, Train Accuracy: 0.934406
Epoch 46 Test Loss: 0.515992, Test Accuracy: 0.857628, time: 38.2s
  Epoch 47 @ step 74000: Train Loss: 0.182353, Train Accuracy: 0.937844
  Epoch 47 @ step 75000: Train Loss: 0.186912, Train Accuracy: 0.935719
Epoch 47 Test Loss: 0.519650, Test Accuracy: 0.860024, time: 38.2s
  Epoch 48 @ step 76000: Train Loss: 0.176767, Train Accuracy: 0.940531
Epoch 48 Test Loss: 0.533460, Test Accuracy: 0.853435, time: 38.1s
  Epoch 49 @ step 77000: Train Loss: 0.173800, Train Accuracy: 0.941844
```

```
  Epoch 49 @ step 78000: Train Loss: 0.172286, Train Accuracy: 0.941906
Epoch 49 Test Loss: 0.520330, Test Accuracy: 0.856629, time: 38.2s
  Epoch 50 @ step 79000: Train Loss: 0.166942, Train Accuracy: 0.941469
Epoch 50 Test Loss: 0.525385, Test Accuracy: 0.858926, time: 38.2s
  Epoch 51 @ step 80000: Train Loss: 0.162864, Train Accuracy: 0.943469
  Epoch 51 @ step 81000: Train Loss: 0.158696, Train Accuracy: 0.946625
Epoch 51 Test Loss: 0.547873, Test Accuracy: 0.857728, time: 38.2s
  Epoch 52 @ step 82000: Train Loss: 0.152656, Train Accuracy: 0.948438
Epoch 52 Test Loss: 0.534700, Test Accuracy: 0.858726, time: 38.0s
  Epoch 53 @ step 83000: Train Loss: 0.159697, Train Accuracy: 0.946531
  Epoch 53 @ step 84000: Train Loss: 0.149814, Train Accuracy: 0.949063
Epoch 53 Test Loss: 0.530289, Test Accuracy: 0.861022, time: 38.3s
  Epoch 54 @ step 85000: Train Loss: 0.146403, Train Accuracy: 0.950031
Epoch 54 Test Loss: 0.540495, Test Accuracy: 0.858626, time: 38.1s
  Epoch 55 @ step 86000: Train Loss: 0.141641, Train Accuracy: 0.951594
  Epoch 55 @ step 87000: Train Loss: 0.134346, Train Accuracy: 0.954781
Epoch 55 Test Loss: 0.540200, Test Accuracy: 0.855531, time: 38.2s
  Epoch 56 @ step 88000: Train Loss: 0.139033, Train Accuracy: 0.953125
  Epoch 56 @ step 89000: Train Loss: 0.144132, Train Accuracy: 0.949938
Epoch 56 Test Loss: 0.569438, Test Accuracy: 0.853934, time: 38.1s
  Epoch 57 @ step 90000: Train Loss: 0.135469, Train Accuracy: 0.954250
Epoch 57 Test Loss: 0.564087, Test Accuracy: 0.856729, time: 38.2s
  Epoch 58 @ step 91000: Train Loss: 0.129687, Train Accuracy: 0.956656
  Epoch 58 @ step 92000: Train Loss: 0.126419, Train Accuracy: 0.956875
Epoch 58 Test Loss: 0.549876, Test Accuracy: 0.859225, time: 38.2s
  Epoch 59 @ step 93000: Train Loss: 0.122108, Train Accuracy: 0.958094
Epoch 59 Test Loss: 0.552209, Test Accuracy: 0.860224, time: 38.0s
Final Test Loss: 0.552209, Test Accuracy: 0.860224, Total time: 2293.3s
```

What changes did you make to improve your model?

In [16]:
```
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 [18]:
```
!wget http://courses.engr.illinois.edu/cs498aml/sp2019/homeworks/test_file.zip
!unzip test_file.zip
```

```
--2019-04-26 04:35:49--  http://courses.engr.illinois.edu/cs498aml/sp2019/homeworks/test
_file.zip
Resolving courses.engr.illinois.edu (courses.engr.illinois.edu)... 130.126.151.9
Connecting to courses.engr.illinois.edu (courses.engr.illinois.edu)|130.126.151.9|:80...
connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://courses.engr.illinois.edu/cs498aml/sp2019/homeworks/test_file.zip [fol
lowing]
--2019-04-26 04:35:49--  https://courses.engr.illinois.edu/cs498aml/sp2019/homeworks/tes
t_file.zip
Connecting to courses.engr.illinois.edu (courses.engr.illinois.edu)|130.126.151.9|:44
3... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3841776 (3.7M) [application/x-zip-compressed]
Saving to: ‘test_file.zip’

test_file.zip       100%[===================>]   3.66M  12.1MB/s    in 0.3s

2019-04-26 04:35:50 (12.1 MB/s) - ‘test_file.zip’ saved [3841776/3841776]

Archive:  test_file.zip
  inflating: test_file.pt
```

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 [0]:
```
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_loader_2 = torch.utils.data.DataLoader(torch_dataset, batch_size=8, shuffle=False,
num_workers=2)
```

In [0]:
```
output_file=np.array([])
for ele in test_loader_2:
    x = ele[0]
    x=x.reshape(-1,3,32,32)
    x=x.type(torch.cuda.FloatTensor)
    # Fill your code here
    outputs = model.net(x)
    _, predicted = torch.max(outputs, 1)
    output_file=np.append(output_file,  torch.Tensor.cpu(predicted))
np.savetxt('submission.txt', output_file, delimiter='\n', fmt='%d')
from google.colab import files
files.download('submission.txt')
```

Report

Part 0: Imports and Basic Setup (5 Points)

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

Part 1: Fully connected neural networks (25 Points)

Test (on validation set) accuracy (5 Points): **0.425020**

Test loss (5 Points): **1.620814**

Training time (5 Points): **54.9 sec**

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)

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: **16x32x32**

Output dimension after 1st max pooling: **16x16x16**

Output dimension after 2nd conv layer: **16x16x16**

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): **0.584864**

Test loss (5 Points): **1.159240**

Training time (5 Points): **75.7 sec**

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)

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):

Test (on validation set) Accuracy (5 Points): **0.860224**

Test loss (5 Points): **0.552209**

Training time (5 Points): **2293.3 sec**

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)

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)

In [0]:

```

```