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
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/. 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:

```
import torch
torch.rand(5, 3)
```

```
tensor([[0.1072, 0.1348, 0.5591],
        [0.0972, 0.1583, 0.9011],
        [0.3416, 0.6958, 0.7808],
        [0.5589, 0.8751, 0.1891],
        [0.6782, 0.3040, 0.4816]])
```

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.

```
import numpy as np
import torch
from torch import nn
from torch import optim
from torchsummary import summary

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.

```
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

```
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, training_loss]]))
        all_training_acc = np.concatenate((all_training_acc, [[training_step, training_acc]]))

        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.

```
!mkdir hw10_data
```

```
# 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=transformations)
```

```
  0%|          | 0/170498071 [00:00<?, ?it/s]Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to hw10_dat
170500096it [00:07, 23606943.09it/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.

```
from torch.utils.data import DataLoader

batch_size = 8
train_loader = torch.utils.data.DataLoader(train_set, batch_size, shuffle=True, num_workers=2)
test_loader = torch.utils.data.DataLoader(test_set, batch_size, shuffle=True, num_workers=2)

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

```

```
training_epochs = 5
```

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.

```
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)
```

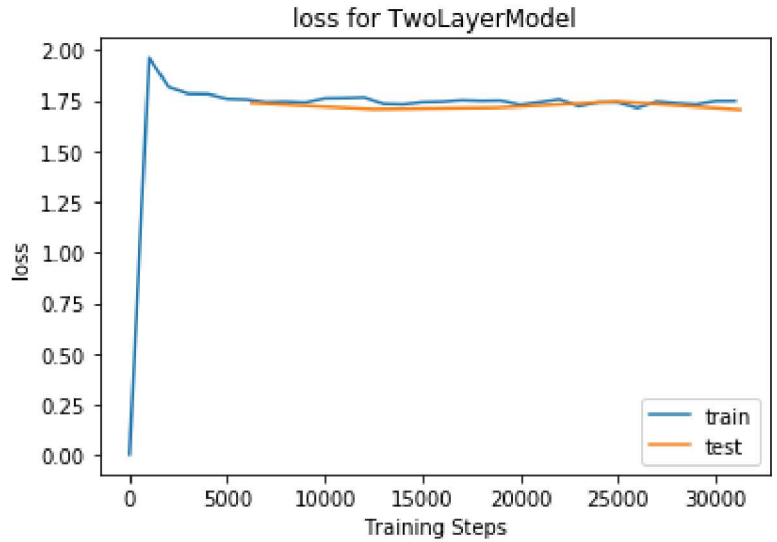
```
  Epoch 0 @ step 0: Train Loss: 0.004301, Train Accuracy: 0.000125
  Epoch 0 @ step 1000: Train Loss: 1.961467, Train Accuracy: 0.320250
  Epoch 0 @ step 2000: Train Loss: 1.817933, Train Accuracy: 0.352000
  Epoch 0 @ step 3000: Train Loss: 1.783904, Train Accuracy: 0.357500
  Epoch 0 @ step 4000: Train Loss: 1.782436, Train Accuracy: 0.366000
  Epoch 0 @ step 5000: Train Loss: 1.758631, Train Accuracy: 0.374250
  Epoch 0 @ step 6000: Train Loss: 1.755314, Train Accuracy: 0.368750
Epoch 0 Test Loss: 1.738084, Test Accuracy: 0.378600, time: 16.6s
  Epoch 1 @ step 7000: Train Loss: 1.742370, Train Accuracy: 0.383625
  Epoch 1 @ step 8000: Train Loss: 1.744828, Train Accuracy: 0.374375
  Epoch 1 @ step 9000: Train Loss: 1.740492, Train Accuracy: 0.374625
  Epoch 1 @ step 10000: Train Loss: 1.761740, Train Accuracy: 0.368875
  Epoch 1 @ step 11000: Train Loss: 1.763021, Train Accuracy: 0.372375
  Epoch 1 @ step 12000: Train Loss: 1.766127, Train Accuracy: 0.367250
Epoch 1 Test Loss: 1.707702, Test Accuracy: 0.386500, time: 17.7s
  Epoch 2 @ step 13000: Train Loss: 1.734238, Train Accuracy: 0.381125
```

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.

```
Epoch 2 Test Loss: 1.715050, Test Accuracy: 0.387000, time: 16.8s
```

```
1 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)

```
layer1_in_channels = 3
layer1_output_channels = 16

```

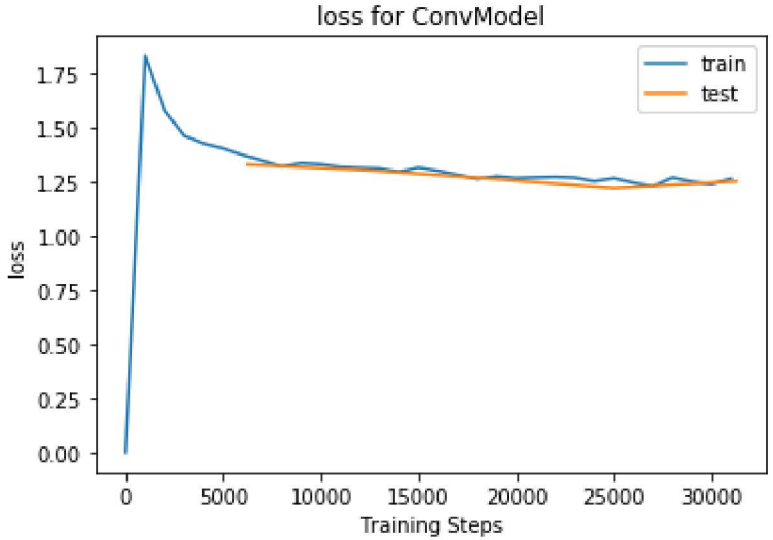
```
layer2_in_channels = layer1_output_channels
layer2_output_channels = 16

layer3_out = 64
kernel_size = 3
nun_labels = 10

padding_size=1
stride_size=1

class ConvModel(nn.Module):
  # Your Code Here
  def __init__(self):
        super().__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(layer1_in_channels, layer1_output_channels, kernel_size, stride=stride_size, padding=padding_size),
            nn.ReLU(),
            nn.MaxPool2d(2))

        self.layer2 = nn.Sequential(
            nn.Conv2d(layer2_in_channels, layer2_output_channels, kernel_size, stride=stride_size, padding=padding_size),
            nn.ReLU(),
            Flatten())


        self.layer3 = nn.Sequential(
            nn.Linear(4096, layer3_out),
            nn.ReLU())

        self.fc = nn.Linear(layer3_out, nun_labels)


  def forward(self, x):
      out = self.layer1(x)
      out = self.layer2(out)
      out = self.layer3(out)
      out = out.reshape(out.size(0), -1)
      return self.fc(out)



print(ConvModel())

model = ConvModel().to(device)

summary(model, (3,32,32))

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)
```

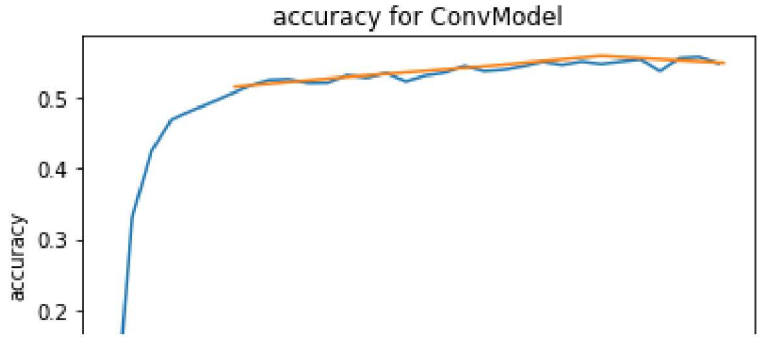
```
55
```

```
ConvModel(
  (layer1): Sequential(
    (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer2): Sequential(
    (0): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Flatten()
  )
  (layer3): Sequential(
    (0): Linear(in_features=4096, out_features=64, bias=True)
    (1): ReLU()
  )
  (fc): Linear(in_features=64, out_features=10, bias=True)
)
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 16, 32, 32]             448
              ReLU-2          [-1, 16, 32, 32]               0
         MaxPool2d-3          [-1, 16, 16, 16]               0
            Conv2d-4          [-1, 16, 16, 16]           2,320
              ReLU-5          [-1, 16, 16, 16]               0
           Flatten-6                [-1, 4096]               0
            Linear-7                  [-1, 64]         262,208
              ReLU-8                  [-1, 64]               0
            Linear-9                  [-1, 10]             650
================================================================
Total params: 265,626
Trainable params: 265,626
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.38
Params size (MB): 1.01
Estimated Total Size (MB): 1.40
----------------------------------------------------------------
Exception ignored in: <bound method _DataLoaderIter.__del__ of <torch.utils.data.dataloader._DataLoaderIter object at 0x7fe6
Traceback (most recent call last):
  File "/usr/local/lib/python3.6/dist-packages/torch/utils/data/dataloader.py", line 717, in __del
```

```
Traceback (most recent call last):
    self._shutdown_workers()
Exception ignored in: <bound method _DataLoaderIter.__del__ of <torch.utils.data.dataloader._DataLoaderIter object at 0x7fe6
Traceback (most recent call last):
  File "/usr/local/lib/python3.6/dist-packages/torch/utils/data/dataloader.py", line 713, in _shutdown_workers
  File "/usr/lib/python3.6/multiprocessing/queues.py", line 240, in _feed
    send_bytes(obj)
    w.join()
  File "/usr/lib/python3.6/multiprocessing/process.py", line 122, in join
  File "/usr/lib/python3.6/multiprocessing/connection.py", line 200, in send_bytes
    self._send_bytes(m[offset:offset + size])
    assert self._parent_pid == os.getpid(), 'can only join a child process'
  File "/usr/lib/python3.6/multiprocessing/connection.py", line 404, in _send_bytes
    self._send(header + buf)
AssertionError: can only join a child process
  File "/usr/lib/python3.6/multiprocessing/queues.py", line 240, in _feed
    send_bytes(obj)
Traceback (most recent call last):
  File "/usr/lib/python3.6/multiprocessing/connection.py", line 368, in _send
    n = write(self._handle, buf)
BrokenPipeError: [Errno 32] Broken pipe
  File "/usr/local/lib/python3.6/dist-packages/torch/utils/data/dataloader.py", line 717, in __del__
  File "/usr/lib/python3.6/multiprocessing/connection.py", line 200, in send_bytes
    self._send_bytes(m[offset:offset + size])
  File "/usr/lib/python3.6/multiprocessing/connection.py", line 404, in _send_bytes
    self._send(header + buf)
    self._shutdown_workers()
  File "/usr/local/lib/python3.6/dist-packages/torch/utils/data/dataloader.py", line 713, in _shutdown_workers
    w.join()
  File "/usr/lib/python3.6/multiprocessing/connection.py", line 368, in _send
    n = write(self._handle, buf)
  File "/usr/lib/python3.6/multiprocessing/process.py", line 122, in join
```

```
1 plot_graphs("ConvModel", metrics)
```





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

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**.

```
# You Awesome Super Best model code here
layer1_in_channels = 3
layer1_output_channels = 128

layer2_in_channels = layer1_output_channels
layer2_output_channels = 128

layer3_in_channels = layer2_output_channels
layer3_output_channels =128

layer4_in_channels = layer3_output_channels
layer4_output_channels = 128

kernel_size = 3
nun_labels = 10


class AwesomeModel(nn.Module):
  def __init__(self):
        super().__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(layer1_in_channels, layer1_output_channels, kernel_size, padding=1),
            nn.ReLU(),
            #nn.Conv2d(layer1_output_channels, layer1_output_channels, kernel_size, padding=1),
            #nn.ReLU(),
            nn.MaxPool2d(2))

        self.layer2 = nn.Sequential(
            nn.Conv2d(layer2_in_channels, layer2_output_channels, kernel_size, padding=1),
            nn.ReLU(),
            #nn.Conv2d(layer2_output_channels, layer2_output_channels, kernel_size, padding=1),
            #nn.ReLU(),
            #nn.Conv2d(layer2_output_channels, layer2_output_channels, kernel_size, padding=1),
            #nn.ReLU(),
            nn.MaxPool2d(2))

        self.layer3 = nn.Sequential(
```

```
            nn.Conv2d(layer3_in_channels, layer3_output_channels, kernel_size),
            nn.BatchNorm2d(layer3_output_channels),
            nn.ReLU(),
            nn.MaxPool2d(2))

        self.dropout = nn.Dropout2d(p=0.2)

        self.layer4 = nn.Sequential(
            nn.Conv2d(layer4_in_channels, layer4_output_channels, kernel_size, padding=1),
            nn.BatchNorm2d(layer4_output_channels),
            nn.ReLU(),
            nn.MaxPool2d(2))

        self.fc = nn.Linear(layer4_output_channels, nun_labels)

  def forward(self, x):
      out = self.layer1(x)

      out = self.layer2(out)

      out = self.layer3(out)

      out = self.dropout(out)

      out = self.layer4(out)

      out = out.reshape(out.size(0), -1)
      return self.fc(out)




model = AwesomeModel().to(device)

loss = nn.CrossEntropyLoss()
# optimizer = optim.RMSprop(model.parameters(), lr=0.001, weight_decay=0.01)
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

summary(model, (3,32,32))
metrics = train(model, train_loader, test_loader, loss, optimizer, 10)
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 128, 32, 32]           3,584
              ReLU-2          [-1, 128, 32, 32]               0
         MaxPool2d-3          [-1, 128, 16, 16]               0
            Conv2d-4          [-1, 128, 16, 16]         147,584
              ReLU-5          [-1, 128, 16, 16]               0
         MaxPool2d-6            [-1, 128, 8, 8]               0
            Conv2d-7            [-1, 128, 6, 6]         147,584
       BatchNorm2d-8            [-1, 128, 6, 6]             256
              ReLU-9            [-1, 128, 6, 6]               0
        MaxPool2d-10            [-1, 128, 3, 3]               0
        Dropout2d-11            [-1, 128, 3, 3]               0
           Conv2d-12            [-1, 128, 3, 3]         147,584
      BatchNorm2d-13            [-1, 128, 3, 3]             256
             ReLU-14            [-1, 128, 3, 3]               0
        MaxPool2d-15            [-1, 128, 1, 1]               0
           Linear-16                  [-1, 10]           1,290
================================================================
Total params: 448,138
Trainable params: 448,138
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 2.96
Params size (MB): 1.71
Estimated Total Size (MB): 4.68
----------------------------------------------------------------
  Epoch 0 @ step 0: Train Loss: 0.004442, Train Accuracy: 0.000125
  Epoch 0 @ step 1000: Train Loss: 1.760698, Train Accuracy: 0.350500
  Epoch 0 @ step 2000: Train Loss: 1.466611, Train Accuracy: 0.469375
  Epoch 0 @ step 3000: Train Loss: 1.319344, Train Accuracy: 0.527750
  Epoch 0 @ step 4000: Train Loss: 1.225576, Train Accuracy: 0.568000
  Epoch 0 @ step 5000: Train Loss: 1.152384, Train Accuracy: 0.594125
  Epoch 0 @ step 6000: Train Loss: 1.083020, Train Accuracy: 0.613125
Epoch 0 Test Loss: 0.952849, Test Accuracy: 0.659100, time: 31.8s
  Epoch 1 @ step 7000: Train Loss: 1.030204, Train Accuracy: 0.640500
  Epoch 1 @ step 8000: Train Loss: 0.981465, Train Accuracy: 0.657250
  Epoch 1 @ step 9000: Train Loss: 0.962273, Train Accuracy: 0.666125
  Epoch 1 @ step 10000: Train Loss: 0.941776, Train Accuracy: 0.672875
  Epoch 1 @ step 11000: Train Loss: 0.909894, Train Accuracy: 0.685375
```

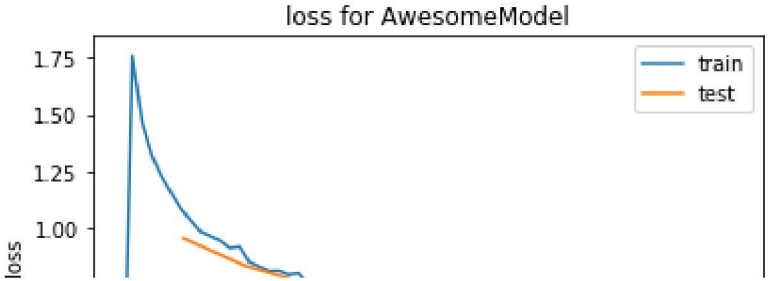
```
  Epoch 1 @ step 12000: Train Loss: 0.915352, Train Accuracy: 0.677375
Epoch 1 Test Loss: 0.831166, Test Accuracy: 0.716500, time: 33.0s
  Epoch 2 @ step 13000: Train Loss: 0.845838, Train Accuracy: 0.705250
  Epoch 2 @ step 14000: Train Loss: 0.827135, Train Accuracy: 0.712000
  Epoch 2 @ step 15000: Train Loss: 0.807027, Train Accuracy: 0.723375
  Epoch 2 @ step 16000: Train Loss: 0.808400, Train Accuracy: 0.723375
  Epoch 2 @ step 17000: Train Loss: 0.791600, Train Accuracy: 0.723875
  Epoch 2 @ step 18000: Train Loss: 0.798014, Train Accuracy: 0.722125
Epoch 2 Test Loss: 0.754116, Test Accuracy: 0.739400, time: 32.4s
  Epoch 3 @ step 19000: Train Loss: 0.749551, Train Accuracy: 0.741000
  Epoch 3 @ step 20000: Train Loss: 0.698005, Train Accuracy: 0.754625
  Epoch 3 @ step 21000: Train Loss: 0.717129, Train Accuracy: 0.745750
  Epoch 3 @ step 22000: Train Loss: 0.699480, Train Accuracy: 0.757875
  Epoch 3 @ step 23000: Train Loss: 0.703705, Train Accuracy: 0.753625
  Epoch 3 @ step 24000: Train Loss: 0.702818, Train Accuracy: 0.757000
Epoch 3 Test Loss: 0.686003, Test Accuracy: 0.763200, time: 32.4s
  Epoch 4 @ step 25000: Train Loss: 0.697754, Train Accuracy: 0.757625
  Epoch 4 @ step 26000: Train Loss: 0.613823, Train Accuracy: 0.783250
  Epoch 4 @ step 27000: Train Loss: 0.607760, Train Accuracy: 0.787625
  Epoch 4 @ step 28000: Train Loss: 0.629644, Train Accuracy: 0.783375
  Epoch 4 @ step 29000: Train Loss: 0.618953, Train Accuracy: 0.781750
  Epoch 4 @ step 30000: Train Loss: 0.626694, Train Accuracy: 0.780875
  Epoch 4 @ step 31000: Train Loss: 0.635129, Train Accuracy: 0.774750
Epoch 4 Test Loss: 0.673252, Test Accuracy: 0.767200, time: 32.3s
  Epoch 5 @ step 32000: Train Loss: 0.559631, Train Accuracy: 0.806500
  Epoch 5 @ step 33000: Train Loss: 0.550793, Train Accuracy: 0.810750
  Epoch 5 @ step 34000: Train Loss: 0.548881, Train Accuracy: 0.808500
  Epoch 5 @ step 35000: Train Loss: 0.565127, Train Accuracy: 0.801000
  Epoch 5 @ step 36000: Train Loss: 0.557797, Train Accuracy: 0.806125
  Epoch 5 @ step 37000: Train Loss: 0.577192, Train Accuracy: 0.798625
Epoch 5 Test Loss: 0.643921, Test Accuracy: 0.780900, time: 32.1s
  Epoch 6 @ step 38000: Train Loss: 0.496895, Train Accuracy: 0.821250
  Epoch 6 @ step 39000: Train Loss: 0.488642, Train Accuracy: 0.836000
  Epoch 6 @ step 40000: Train Loss: 0.510050, Train Accuracy: 0.820750
  Epoch 6 @ step 41000: Train Loss: 0.512778, Train Accuracy: 0.821250
  Epoch 6 @ step 42000: Train Loss: 0.528616, Train Accuracy: 0.816250
  Epoch 6 @ step 43000: Train Loss: 0.531179, Train Accuracy: 0.812250
Epoch 6 Test Loss: 0.603606, Test Accuracy: 0.796800, time: 32.8s
  Epoch 7 @ step 44000: Train Loss: 0.496106, Train Accuracy: 0.827375
  Epoch 7 @ step 45000: Train Loss: 0.443340, Train Accuracy: 0.844500
  Epoch 7 @ step 46000: Train Loss: 0.431364, Train Accuracy: 0.850875
  Epoch 7 @ step 47000: Train Loss: 0.464086, Train Accuracy: 0.837125
```

```
  Epoch 7 @ step 48000: Train Loss: 0.470960, Train Accuracy: 0.836750
  Epoch 7 @ step 49000: Train Loss: 0.476336, Train Accuracy: 0.832500
Epoch 7 Test Loss: 0.616699, Test Accuracy: 0.798900, time: 32.6s
  Epoch 8 @ step 50000: Train Loss: 0.453773, Train Accuracy: 0.838500
  Epoch 8 @ step 51000: Train Loss: 0.391436, Train Accuracy: 0.860125
  Epoch 8 @ step 52000: Train Loss: 0.423008, Train Accuracy: 0.849625
  Epoch 8 @ step 53000: Train Loss: 0.406135, Train Accuracy: 0.859375
  Epoch 8 @ step 54000: Train Loss: 0.401967, Train Accuracy: 0.858375
  Epoch 8 @ step 55000: Train Loss: 0.436225, Train Accuracy: 0.847875
  Epoch 8 @ step 56000: Train Loss: 0.450351, Train Accuracy: 0.841125
Epoch 8 Test Loss: 0.615750, Test Accuracy: 0.796400, time: 33.0s
```

What changes did you make to improve your model?

```
  Epoch 9 @ step 59000: Train Loss: 0.382001, Train Accuracy: 0.879000
```

```
1 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.

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

```
--2019-05-02 04:53:55--  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 [following]
--2019-05-02 04:53:56--  https://courses.engr.illinois.edu/cs498aml/sp2019/homeworks/test_file.zip
Connecting to courses.engr.illinois.edu (courses.engr.illinois.edu)|130.126.151.9|:443... 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  5.06MB/s    in 0.7s

2019-05-02 04:53:56 (5.06 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.

```
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 = torch.utils.data.DataLoader(torch_dataset, batch_size, shuffle=False, num_workers=2)

model.eval()
with torch.no_grad():
  for ele in test_loader:
      x = ele[0]
      image = x.reshape(-1,3,32,32)
      image = image.to(device)
      preds = model(image)

      pred_labels = torch.argmax(preds, dim=1)
      for pred_label in pred_labels:
        f_pred.write(str(pred_label.item()))
        f_pred.write('\n')

f_pred.close()
```

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): 0.39

Test loss (5 Points): 1.70

Training time (5 Points): 85.6s

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: (8,16, 32, 32)

Output dimension after 1st max pooling: (8, 16, 16, 16)

Output dimension after 2nd conv layer: (8,16, 16, 16)

Output dimension after flatten layer: (8, 4096)

Output dimension after 1st fully connected layer: (8,64)

Output dimension after 2nd fully connected layer: (8,10)

Test (on validation set) Accuracy (5 Points): 0.55

Test loss (5 Points): 1.25

Training time (5 Points): 128.4s

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):

We experimented with various parameters and concluded a final model:

TEST MODEL1: Added 2 additional convolutional layers(total 4) with 16 output channels each because it will extract more subtle features from the image. Added maxpool(2) after each convolutional layer, dropout with probability of 0.2 after the last convolutional layer and batch normalization. Increased epochs to 10. Changed Optimizer from RMSprop to SGD. This gave the test accuracy ~ 64%

TEST MODEL 2: Modified TEST MODEL 1 by changing output channels to 32 in all convolutional layers. This gave the test accuracy ~ 72%

TEST MODEL 3: Modified TEST MODEL 2 dropout rate from 0.2 to 0.4. This gave the test accuracy ~ 71%

FINAL MODEL: Modified TEST MODEL 2 by changing output channels to 128. This gave the test accuracy ~ 79%

Test (on validation set) Accuracy (5 Points): 0.79

Test loss (5 Points): 0.62

Training time (5 Points): 324.5s

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)

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)

```
1
```