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([[0.2975, 0.8837, 0.5905],
        [0.6292, 0.2722, 0.2039],
        [0.3615, 0.5387, 0.2503],
        [0.7808, 0.6559, 0.0407],
        [0.8948, 0.8002, 0.5082]])
```

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

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
```

```
mkdir: cannot create directory ‘hw10_data’: File exists
```

```
# 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=transforma
```

```
Files already downloaded and verified
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: 0.004283, Train Accuracy: 0.000125
  Epoch 0 @ step 1000: Train Loss: 1.955173, Train Accuracy: 0.321875
  Epoch 0 @ step 2000: Train Loss: 1.821083, Train Accuracy: 0.350500
  Epoch 0 @ step 3000: Train Loss: 1.799701, Train Accuracy: 0.358750
  Epoch 0 @ step 4000: Train Loss: 1.764672, Train Accuracy: 0.375250
  Epoch 0 @ step 5000: Train Loss: 1.753418, Train Accuracy: 0.374500
  Epoch 0 @ step 6000: Train Loss: 1.743400, Train Accuracy: 0.375375
Epoch 0 Test Loss: 1.723607, Test Accuracy: 0.388700, time: 17.1s
  Epoch 1 @ step 7000: Train Loss: 1.757134, Train Accuracy: 0.371500
  Epoch 1 @ step 8000: Train Loss: 1.724747, Train Accuracy: 0.380750
  Epoch 1 @ step 9000: Train Loss: 1.739099, Train Accuracy: 0.387375
  Epoch 1 @ step 10000: Train Loss: 1.736827, Train Accuracy: 0.383500
  Epoch 1 @ step 11000: Train Loss: 1.739003, Train Accuracy: 0.377250
  Epoch 1 @ step 12000: Train Loss: 1.733101, Train Accuracy: 0.384750
Epoch 1 Test Loss: 1.749516, Test Accuracy: 0.377500, time: 17.3s
  Epoch 2 @ step 13000: Train Loss: 1.720829, Train Accuracy: 0.386875
  Epoch 2 @ step 14000: Train Loss: 1.731586, Train Accuracy: 0.384250
  Epoch 2 @ step 15000: Train Loss: 1.737245, Train Accuracy: 0.376625
  Epoch 2 @ step 16000: Train Loss: 1.759624, Train Accuracy: 0.379250
  Epoch 2 @ step 17000: Train Loss: 1.728732, Train Accuracy: 0.379500
  Epoch 2 @ step 18000: Train Loss: 1.733597, Train Accuracy: 0.386125
Epoch 2 Test Loss: 1.765922, Test Accuracy: 0.361100, time: 18.0s
  Epoch 3 @ step 19000: Train Loss: 1.712870, Train Accuracy: 0.390125
  Epoch 3 @ step 20000: Train Loss: 1.749638, Train Accuracy: 0.375875
  Epoch 3 @ step 21000: Train Loss: 1.722721, Train Accuracy: 0.383125
  Epoch 3 @ step 22000: Train Loss: 1.738242, Train Accuracy: 0.374500
  Epoch 3 @ step 23000: Train Loss: 1.723327, Train Accuracy: 0.385125
  Epoch 3 @ step 24000: Train Loss: 1.726176, Train Accuracy: 0.385250
Epoch 3 Test Loss: 1.736255, Test Accuracy: 0.370300, time: 18.3s
  Epoch 4 @ step 25000: Train Loss: 1.731594, Train Accuracy: 0.386500
  Epoch 4 @ step 26000: Train Loss: 1.728547, Train Accuracy: 0.376500
  Epoch 4 @ step 27000: Train Loss: 1.726254, Train Accuracy: 0.386375
  Epoch 4 @ step 28000: Train Loss: 1.729353, Train Accuracy: 0.385125
  Epoch 4 @ step 29000: Train Loss: 1.732182, Train Accuracy: 0.381500
  Epoch 4 @ step 30000: Train Loss: 1.749423, Train Accuracy: 0.367375
  Epoch 4 @ step 31000: Train Loss: 1.748205, Train Accuracy: 0.374875
Epoch 4 Test Loss: 1.733261, Test Accuracy: 0.376300, time: 17.1s
Final Test Loss: 1.733261, Test Accuracy: 0.376300, Total time: 87.8s
```

Test accuracy = 0.373 Test Loss = 1.71

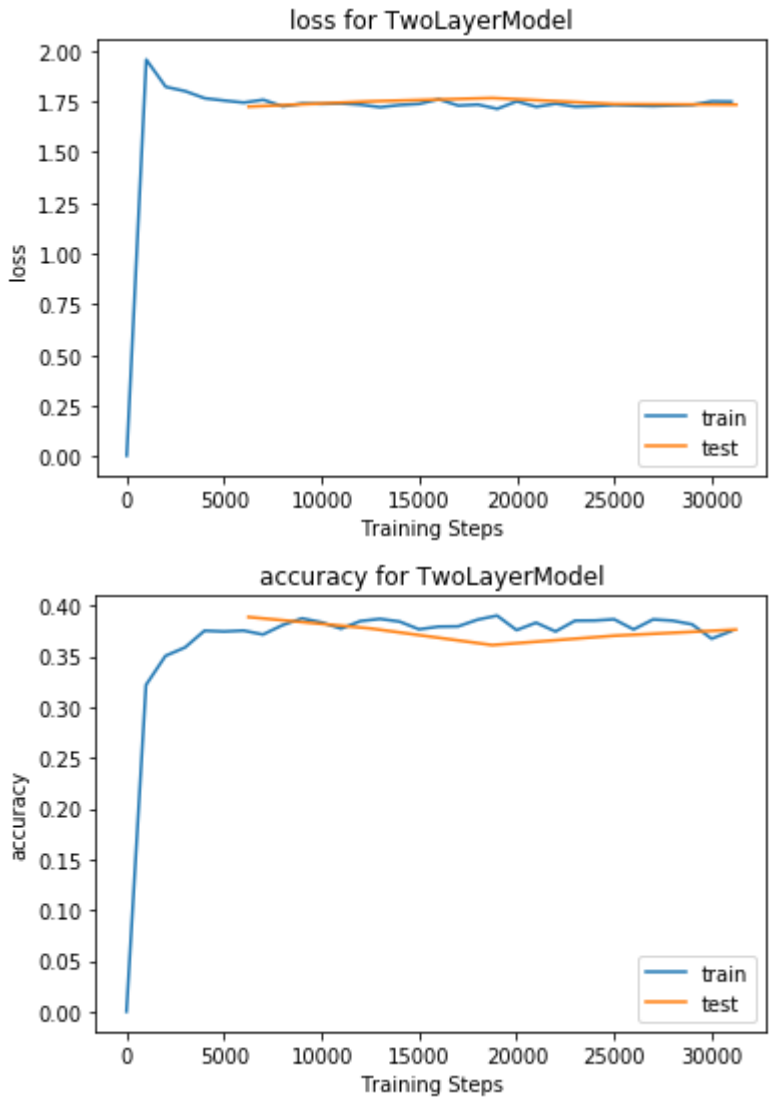
Train accuracy = 0.381 Train Loss = 1.73

Total time: 87.8s

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.

```
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

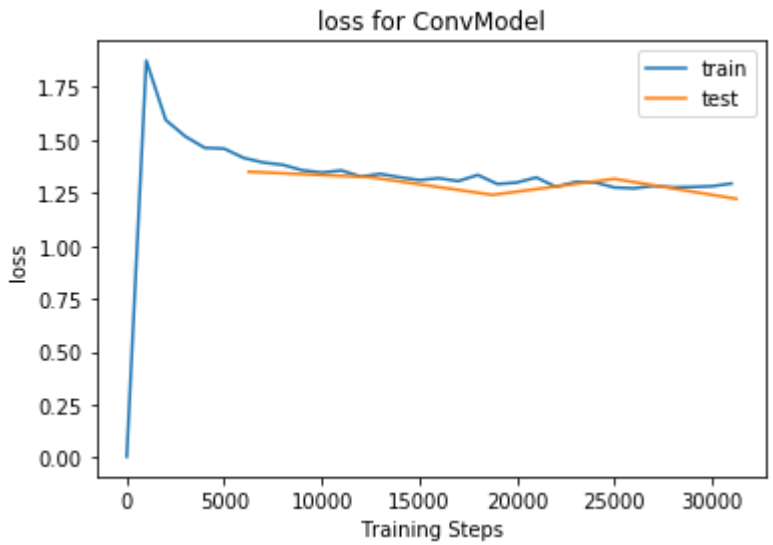
You will have to figure out the input sizes of the first fully connnected layer based on the previous layer sizes. Note that you also need to fill those in the report section (see report section in the notebook for details)

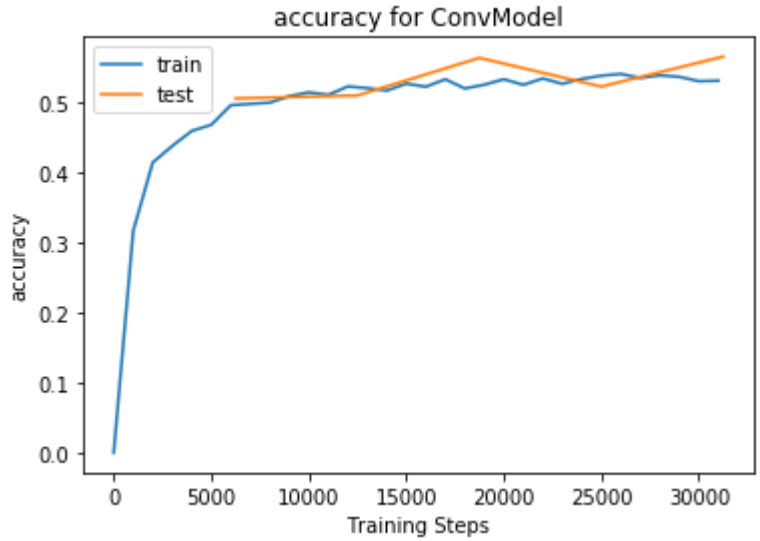
```
import torch.nn.functional as F
class ConvModel(nn.Module):
  # Your Code Here
  def __init__(self):
    super(ConvModel, self).__init__()
    self.net = nn.Sequential(
        nn.Conv2d(3, 16, kernel_size=3, stride = 1, padding=1),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Conv2d(16, 16, kernel_size=3, stride = 1, padding=1),
        nn.ReLU(),
        Flatten(),
        nn.Linear(4096, 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: 0.004281, Train Accuracy: 0.000250
  Epoch 0 @ step 1000: Train Loss: 1.875209, Train Accuracy: 0.317125
  Epoch 0 @ step 2000: Train Loss: 1.594658, Train Accuracy: 0.414625
  Epoch 0 @ step 3000: Train Loss: 1.517218, Train Accuracy: 0.438250
  Epoch 0 @ step 4000: Train Loss: 1.463042, Train Accuracy: 0.459750
  Epoch 0 @ step 5000: Train Loss: 1.459179, Train Accuracy: 0.468500
  Epoch 0 @ step 6000: Train Loss: 1.415305, Train Accuracy: 0.496500
Epoch 0 Test Loss: 1.350353, Test Accuracy: 0.506200, time: 23.4s
  Epoch 1 @ step 7000: Train Loss: 1.393922, Train Accuracy: 0.498375
  Epoch 1 @ step 8000: Train Loss: 1.382849, Train Accuracy: 0.500125
  Epoch 1 @ step 9000: Train Loss: 1.357306, Train Accuracy: 0.509375
  Epoch 1 @ step 10000: Train Loss: 1.346191, Train Accuracy: 0.514875
  Epoch 1 @ step 11000: Train Loss: 1.356274, Train Accuracy: 0.511250
  Epoch 1 @ step 12000: Train Loss: 1.327677, Train Accuracy: 0.523250
Epoch 1 Test Loss: 1.324304, Test Accuracy: 0.510400, time: 25.4s
  Epoch 2 @ step 13000: Train Loss: 1.339970, Train Accuracy: 0.520875
  Epoch 2 @ step 14000: Train Loss: 1.323935, Train Accuracy: 0.517375
  Epoch 2 @ step 15000: Train Loss: 1.310562, Train Accuracy: 0.527250
  Epoch 2 @ step 16000: Train Loss: 1.319552, Train Accuracy: 0.522875
  Epoch 2 @ step 17000: Train Loss: 1.306008, Train Accuracy: 0.533500
  Epoch 2 @ step 18000: Train Loss: 1.334976, Train Accuracy: 0.520375
Epoch 2 Test Loss: 1.241016, Test Accuracy: 0.564100, time: 24.8s
  Epoch 3 @ step 19000: Train Loss: 1.292326, Train Accuracy: 0.525750
  Epoch 3 @ step 20000: Train Loss: 1.299196, Train Accuracy: 0.533500
  Epoch 3 @ step 21000: Train Loss: 1.323126, Train Accuracy: 0.525625
  Epoch 3 @ step 22000: Train Loss: 1.279118, Train Accuracy: 0.534625
  Epoch 3 @ step 23000: Train Loss: 1.301730, Train Accuracy: 0.527000
  Epoch 3 @ step 24000: Train Loss: 1.300097, Train Accuracy: 0.534000
Epoch 3 Test Loss: 1.316565, Test Accuracy: 0.523100, time: 27.7s
  Epoch 4 @ step 25000: Train Loss: 1.275744, Train Accuracy: 0.538625
  Epoch 4 @ step 26000: Train Loss: 1.271970, Train Accuracy: 0.541125
  Epoch 4 @ step 27000: Train Loss: 1.283183, Train Accuracy: 0.535375
  Epoch 4 @ step 28000: Train Loss: 1.276901, Train Accuracy: 0.539125
  Epoch 4 @ step 29000: Train Loss: 1.278390, Train Accuracy: 0.537000
  Epoch 4 @ step 30000: Train Loss: 1.281800, Train Accuracy: 0.530750
  Epoch 4 @ step 31000: Train Loss: 1.294403, Train Accuracy: 0.531500
Epoch 4 Test Loss: 1.222404, Test Accuracy: 0.565900, time: 24.8s
```

```
Test accuracy = 0.564 Test Loss = 1.217

Train accuracy = 0.542 Train Loss = 1.264


Output dimension after 1st conv layer: [4096 * 64]

Output dimension after 1st max pooling: [8,16,16,16]

Output dimension after 2nd conv layer: [2048*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]

Training time: 126.2 s
```

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





Do you notice the improvement over the accuracy compared to that in Part 1? yes the accuracy increased from 0.37 to 0.56

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

class myConvModel(nn.Module):
  # Your Code Here
  def __init__(self):
    super(myConvModel, self).__init__()
    self.net = nn.Sequential(
      nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, stride = 1, padding=1),
      nn.BatchNorm2d(16),
      nn.ReLU(),
      nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride = 1, padding=1),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2, stride=2),
      nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride = 1, padding=1),
      nn.BatchNorm2d(64),
      nn.ReLU(),
      nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride = 1, padding=1),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2, stride=2),
      nn.Dropout2d(),
      nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride = 1, padding=1),
      nn.BatchNorm2d(256),
      nn.ReLU(),
      nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, stride = 1, padding=1),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2, stride=2),
      Flatten(),
      nn.Dropout(),
      nn.Linear(4096, 1024),
      nn.ReLU(),
      nn.Linear(1024, 512),
      nn.ReLU(),
      nn.Linear(512, 256),
      nn.ReLU(),
      nn.Dropout(),
      nn.Linear(256, 128),
      nn.ReLU(),
      nn.Dropout(),
      nn.Linear(128, 10))

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

model = myConvModel().to(device)

loss = nn.CrossEntropyLoss()

training_epochs = 60

batch_size = 64
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]

optimizer = optim.RMSprop(model.parameters(), lr=0.001, weight_decay=0.0005)

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

```
  Epoch 0 @ step 0: Train Loss: 0.004303, Train Accuracy: 0.000141
Epoch 0 Test Loss: 1.906121, Test Accuracy: 0.213873, time: 14.0s
  Epoch 1 @ step 1000: Train Loss: 2.097556, Train Accuracy: 0.186281
Epoch 1 Test Loss: 1.641510, Test Accuracy: 0.370621, time: 13.9s
  Epoch 2 @ step 2000: Train Loss: 1.640497, Train Accuracy: 0.364094
Epoch 2 Test Loss: 1.858847, Test Accuracy: 0.349920, time: 14.1s
  Epoch 3 @ step 3000: Train Loss: 1.389625, Train Accuracy: 0.493656
Epoch 3 Test Loss: 1.577019, Test Accuracy: 0.477110, time: 14.3s
Epoch 4 Test Loss: 1.005117, Test Accuracy: 0.645701, time: 14.2s
  Epoch 5 @ step 4000: Train Loss: 1.204296, Train Accuracy: 0.575500
Epoch 5 Test Loss: 1.392627, Test Accuracy: 0.560311, time: 14.3s
  Epoch 6 @ step 5000: Train Loss: 1.087381, Train Accuracy: 0.623125
Epoch 6 Test Loss: 1.320950, Test Accuracy: 0.552846, time: 14.4s
  Epoch 7 @ step 6000: Train Loss: 0.996420, Train Accuracy: 0.656984
Epoch 7 Test Loss: 1.130363, Test Accuracy: 0.612560, time: 14.4s
  Epoch 8 @ step 7000: Train Loss: 0.945223, Train Accuracy: 0.678109
Epoch 8 Test Loss: 0.874523, Test Accuracy: 0.695561, time: 14.4s
Epoch 9 Test Loss: 0.825625, Test Accuracy: 0.715764, time: 14.4s
  Epoch 10 @ step 8000: Train Loss: 0.891640, Train Accuracy: 0.703313
Epoch 10 Test Loss: 0.755676, Test Accuracy: 0.742934, time: 14.5s
  Epoch 11 @ step 9000: Train Loss: 0.846765, Train Accuracy: 0.716266
Epoch 11 Test Loss: 0.806042, Test Accuracy: 0.718252, time: 14.6s
  Epoch 12 @ step 10000: Train Loss: 0.807531, Train Accuracy: 0.729703
Epoch 12 Test Loss: 0.933709, Test Accuracy: 0.692277, time: 14.6s
Epoch 13 Test Loss: 0.723837, Test Accuracy: 0.755971, time: 14.5s
  Epoch 14 @ step 11000: Train Loss: 0.776092, Train Accuracy: 0.740484
Epoch 14 Test Loss: 1.938973, Test Accuracy: 0.532743, time: 14.4s
  Epoch 15 @ step 12000: Train Loss: 0.748618, Train Accuracy: 0.753406
Epoch 15 Test Loss: 0.726074, Test Accuracy: 0.761644, time: 14.5s
  Epoch 16 @ step 13000: Train Loss: 0.727951, Train Accuracy: 0.759969
Epoch 16 Test Loss: 0.765890, Test Accuracy: 0.752090, time: 14.6s
  Epoch 17 @ step 14000: Train Loss: 0.703123, Train Accuracy: 0.767453
Epoch 17 Test Loss: 1.049686, Test Accuracy: 0.688396, time: 14.5s
Epoch 18 Test Loss: 0.810010, Test Accuracy: 0.748806, time: 14.5s
  Epoch 19 @ step 15000: Train Loss: 0.684931, Train Accuracy: 0.774844
Epoch 19 Test Loss: 0.817552, Test Accuracy: 0.727607, time: 14.5s
  Epoch 20 @ step 16000: Train Loss: 0.666505, Train Accuracy: 0.781266
Epoch 20 Test Loss: 0.683586, Test Accuracy: 0.768412, time: 14.4s
  Epoch 21 @ step 17000: Train Loss: 0.661091, Train Accuracy: 0.783563
Epoch 21 Test Loss: 0.613804, Test Accuracy: 0.800756, time: 14.4s
Epoch 22 Test Loss: 0.723523, Test Accuracy: 0.756369, time: 14.5s
  Epoch 23 @ step 18000: Train Loss: 0.646232, Train Accuracy: 0.787750
Epoch 23 Test Loss: 0.796744, Test Accuracy: 0.738953, time: 14.4s
  Epoch 24 @ step 19000: Train Loss: 0.629539, Train Accuracy: 0.793156
Epoch 24 Test Loss: 0.672113, Test Accuracy: 0.775378, time: 14.5s
  Epoch 25 @ step 20000: Train Loss: 0.623748, Train Accuracy: 0.794828
Epoch 25 Test Loss: 0.647053, Test Accuracy: 0.792197, time: 14.5s
  Epoch 26 @ step 21000: Train Loss: 0.621406, Train Accuracy: 0.798344
Epoch 26 Test Loss: 0.649907, Test Accuracy: 0.780852, time: 14.4s
Epoch 27 Test Loss: 0.554604, Test Accuracy: 0.818471, time: 14.5s
  Epoch 28 @ step 22000: Train Loss: 0.609869, Train Accuracy: 0.803109
Epoch 28 Test Loss: 0.618049, Test Accuracy: 0.796278, time: 14.4s
  Epoch 29 @ step 23000: Train Loss: 0.592381, Train Accuracy: 0.806922
Epoch 29 Test Loss: 0.652882, Test Accuracy: 0.787420, time: 14.5s
  Epoch 30 @ step 24000: Train Loss: 0.592925, Train Accuracy: 0.806594
Epoch 30 Test Loss: 0.602714, Test Accuracy: 0.798567, time: 14.5s
  Epoch 31 @ step 25000: Train Loss: 0.587965, Train Accuracy: 0.808094
```

```
Epoch 31 Test Loss: 0.847310, Test Accuracy: 0.739152, time: 14.5s
Epoch 32 Test Loss: 0.543395, Test Accuracy: 0.822552, time: 14.5s
  Epoch 33 @ step 26000: Train Loss: 0.571947, Train Accuracy: 0.813797
Epoch 33 Test Loss: 0.642782, Test Accuracy: 0.793093, time: 14.3s
  Epoch 34 @ step 27000: Train Loss: 0.572295, Train Accuracy: 0.817281
Epoch 34 Test Loss: 0.597109, Test Accuracy: 0.803941, time: 14.5s
  Epoch 35 @ step 28000: Train Loss: 0.566811, Train Accuracy: 0.814859
Epoch 35 Test Loss: 0.627422, Test Accuracy: 0.798169, time: 14.4s
Epoch 36 Test Loss: 0.653705, Test Accuracy: 0.784634, time: 14.5s
  Epoch 37 @ step 29000: Train Loss: 0.566453, Train Accuracy: 0.817313
Epoch 37 Test Loss: 0.532900, Test Accuracy: 0.819666, time: 14.4s
  Epoch 38 @ step 30000: Train Loss: 0.552832, Train Accuracy: 0.818531
Epoch 38 Test Loss: 0.641723, Test Accuracy: 0.788913, time: 14.3s
  Epoch 39 @ step 31000: Train Loss: 0.553680, Train Accuracy: 0.820391
Epoch 39 Test Loss: 0.644886, Test Accuracy: 0.778165, time: 14.5s
  Epoch 40 @ step 32000: Train Loss: 0.545426, Train Accuracy: 0.823406
Epoch 40 Test Loss: 0.665622, Test Accuracy: 0.780951, time: 14.4s
Epoch 41 Test Loss: 0.532114, Test Accuracy: 0.824940, time: 14.5s
  Epoch 42 @ step 33000: Train Loss: 0.542353, Train Accuracy: 0.823875
Epoch 42 Test Loss: 0.526727, Test Accuracy: 0.830115, time: 14.5s
  Epoch 43 @ step 34000: Train Loss: 0.535528, Train Accuracy: 0.825438
Epoch 43 Test Loss: 0.607848, Test Accuracy: 0.810410, time: 14.3s
  Epoch 44 @ step 35000: Train Loss: 0.528425, Train Accuracy: 0.828109
Epoch 44 Test Loss: 0.602699, Test Accuracy: 0.802150, time: 14.5s
Epoch 45 Test Loss: 0.592680, Test Accuracy: 0.809315, time: 14.3s
  Epoch 46 @ step 36000: Train Loss: 0.536234, Train Accuracy: 0.827500
Epoch 46 Test Loss: 0.536912, Test Accuracy: 0.826732, time: 14.4s
  Epoch 47 @ step 37000: Train Loss: 0.525357, Train Accuracy: 0.827781
Epoch 47 Test Loss: 0.817978, Test Accuracy: 0.755175, time: 14.5s
  Epoch 48 @ step 38000: Train Loss: 0.517549, Train Accuracy: 0.832750
Epoch 48 Test Loss: 0.590293, Test Accuracy: 0.804439, time: 14.3s
  Epoch 49 @ step 39000: Train Loss: 0.525741, Train Accuracy: 0.829266
Epoch 49 Test Loss: 0.600800, Test Accuracy: 0.804638, time: 14.4s
Epoch 50 Test Loss: 0.524428, Test Accuracy: 0.828424, time: 14.4s
  Epoch 51 @ step 40000: Train Loss: 0.519982, Train Accuracy: 0.833422
Epoch 51 Test Loss: 0.575192, Test Accuracy: 0.819068, time: 14.4s
  Epoch 52 @ step 41000: Train Loss: 0.515094, Train Accuracy: 0.833609
Epoch 52 Test Loss: 0.490214, Test Accuracy: 0.837082, time: 14.5s
  Epoch 53 @ step 42000: Train Loss: 0.518034, Train Accuracy: 0.833188
Epoch 53 Test Loss: 0.664562, Test Accuracy: 0.792396, time: 14.3s
  Epoch 54 @ step 43000: Train Loss: 0.509185, Train Accuracy: 0.834859
Epoch 54 Test Loss: 0.545063, Test Accuracy: 0.824841, time: 14.4s
Epoch 55 Test Loss: 0.518590, Test Accuracy: 0.832604, time: 14.4s
  Epoch 56 @ step 44000: Train Loss: 0.504160, Train Accuracy: 0.834969
Epoch 56 Test Loss: 0.496053, Test Accuracy: 0.838973, time: 14.4s
  Epoch 57 @ step 45000: Train Loss: 0.504964, Train Accuracy: 0.837828
Epoch 57 Test Loss: 0.665454, Test Accuracy: 0.786127, time: 14.5s
  Epoch 58 @ step 46000: Train Loss: 0.501946, Train Accuracy: 0.838484
Epoch 58 Test Loss: 0.554067, Test Accuracy: 0.823945, time: 14.3s
Epoch 59 Test Loss: 0.555756, Test Accuracy: 0.814988, time: 14.4s
Final Test Loss: 0.555756, Test Accuracy: 0.814988, Total time: 865.0s
```

Results:

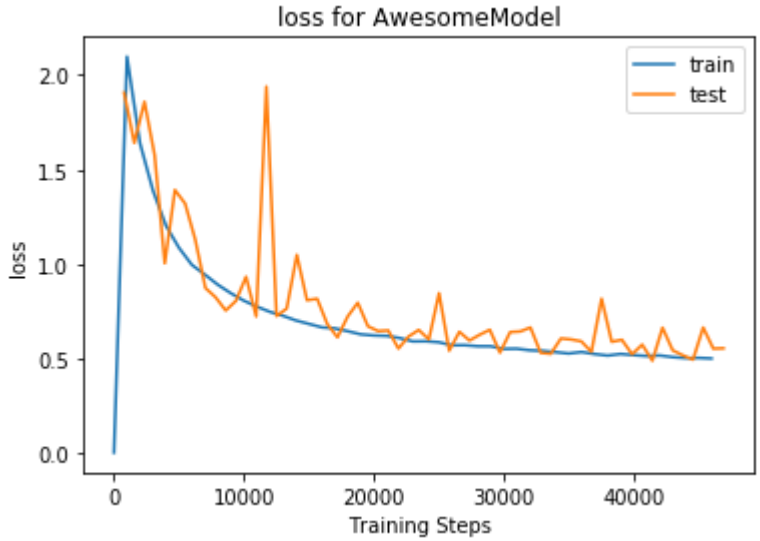
What changes did you make to improve your model? I have added a third and fourth layer, Dropout(), BatchNorm2d, higher epoch, lower weight_decay, and four conv. I think by adding another layer more feature can be found and trained, adding Dropout will prevent overfitting, add more epoch will allow the model to weighting the features with more backward/forward calculations and batchnorm normalize the inputs of each layerto preventt the internal covariate shift problem

Test accuracy = 0.815 Test Loss = 0.555

Train accuracy = 0.838 Train Loss = 0.501

Total time: 865.0s

```
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
```

```
--2019-04-30 17:05:03--  http://courses.engr.illinois.edu/cs498aml/sp2019/homeworks/te
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.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://courses.engr.illinois.edu/cs498aml/sp2019/homeworks/test_file.zip [f
--2019-04-30 17:05:04--  https://courses.engr.illinois.edu/cs498aml/sp2019/homeworks/t
Connecting to courses.engr.illinois.edu (courses.engr.illinois.edu)|130.126.151.9|:443
HTTP request sent, awaiting response... 200 OK
Length: 3841776 (3.7M) [application/x-zip-compressed]
Saving to: ‘test_file.zip.1’

test_file.zip.1     100%[===================>]   3.66M  1.07MB/s    in 3.4s

2019-04-30 17:05:08 (1.07 MB/s) - ‘test_file.zip.1’ saved [3841776/3841776]

Archive:  test_file.zip
replace test_file.pt? [y]es, [n]o, [A]ll, [N]one, [r]ename: y
  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, 1, shuffle=False, num_workers=2)




for ele in test_loader:
    x = ele[0]

    # Fill your code here
    output = model(x.reshape(-1,3,32,32).to(device))
    _, predicted = torch.max(output, 1)
    #print(predicted.cpu().numpy()[0])
    pyval = predicted.cpu().numpy()[0].item()
    f_pred.write(str(pyval))
    f_pred.write('\n')

f_pred.close()
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):

Test loss (5 Points):

Training time (5 Points):

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)

Test accuracy = 0.373 Test Loss = 1.71

Train accuracy = 0.381 Train Loss = 1.73

Total time: 87.8s

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:

Output dimension after 1st max pooling:

Output dimension after 2nd conv layer:

Output dimension after flatten layer:

Output dimension after 1st fully connected layer:

Output dimension after 2nd fully connected layer:

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

Test loss (5 Points):

Training time (5 Points):

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)

Test accuracy = 0.564 Test Loss = 1.217

Train accuracy = 0.542 Train Loss = 1.264

Output dimension after 1st conv layer: [4096 * 64]

Output dimension after 1st max pooling: [8,16,16,16]

Output dimension after 2nd conv layer: [2048*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]

Training time: 126.2 s

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):

Test loss (5 Points):

Training time (5 Points):

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)

Test accuracy = 0.815 Test Loss = 0.555

Train accuracy = 0.838 Train Loss = 0.501

Total time: 865.0s

What changes did you make to improve your model? I have added a third and fourth layer, Dropout(), BatchNorm2d, higher epoch, lower weight_decay, and four conv. I think by adding another layer more feature can be found and trained, adding Dropout will prevent overfitting, add more epoch will allow the model to weighting the features with more backward/forward calculations and batchnorm normalize the