Problem 3 (Modern Deep Learning Tools: PyTorch)

As you might imagine from the previous problem, actually implementing backprop by hand can be frustrating! Fortunately, having modern automatic differentiation tools means that you will rarely have to do so. In this problem, you will learn how to use PyTorch (the ML library of choice in industry and academia) and implement your own neural networks for image classification.

To begin the assignment, **upload T3_P3.ipynb to Google Colab**. Write and run your code in Colab, and download your Colab notebook as an .ipynb file to submit as a supplemental file. Include your written responses, plots, and required code (as specified) in this LaTeX file.

If you have never used Google Colab before, see the 'HW 3' Addendum Post on Ed, where the staff have compiled resources to help you get started.

You can use the Listings package to display code, as shown below:

```
example_array = np.array([1, 2, 3])
```

1. Please answer Problem 3.1 from the Colab here.
2. Please answer Problem 3.2 from the Colab here.
3. Please answer Problem 3.3 from the Colab here.
4. Please answer Problem 3.4 from the Colab here.
5. Please answer Problem 3.5 from the Colab here.
6. Please answer Problem 3.6 from the Colab here.
7. Please answer Problem 3.7 from the Colab here.

Solution:

1. *Where are gradients stored in PyTorch and how can they be accessed?*

   The gradients of our tensors are stored in a computation graph in PyTorch. This computation graph is constructed implicitly and automatically. This allows Autograd, which implements automatic differentiation, to efficiently perform backpropogation using the chain rule and the gradients stored in the computation graph. For the tensors and gradients to be stored in the computation graph, the `requires_grad` field of the Tensors must be set to true so its gradients will be stored in the computation graph. `Function` objects encode the computation graph, allowing us to access the gradients using the `grad_fn` attribute of Tensor.

2. *In the model defined in Step 2 in the tutorial section, how many weights does the model have, and which layer do they come from?*

   There are four weights for x, x^2, x^3, and the bias term. These weights come from the `torch.nn.Linear(3,1)` layer.

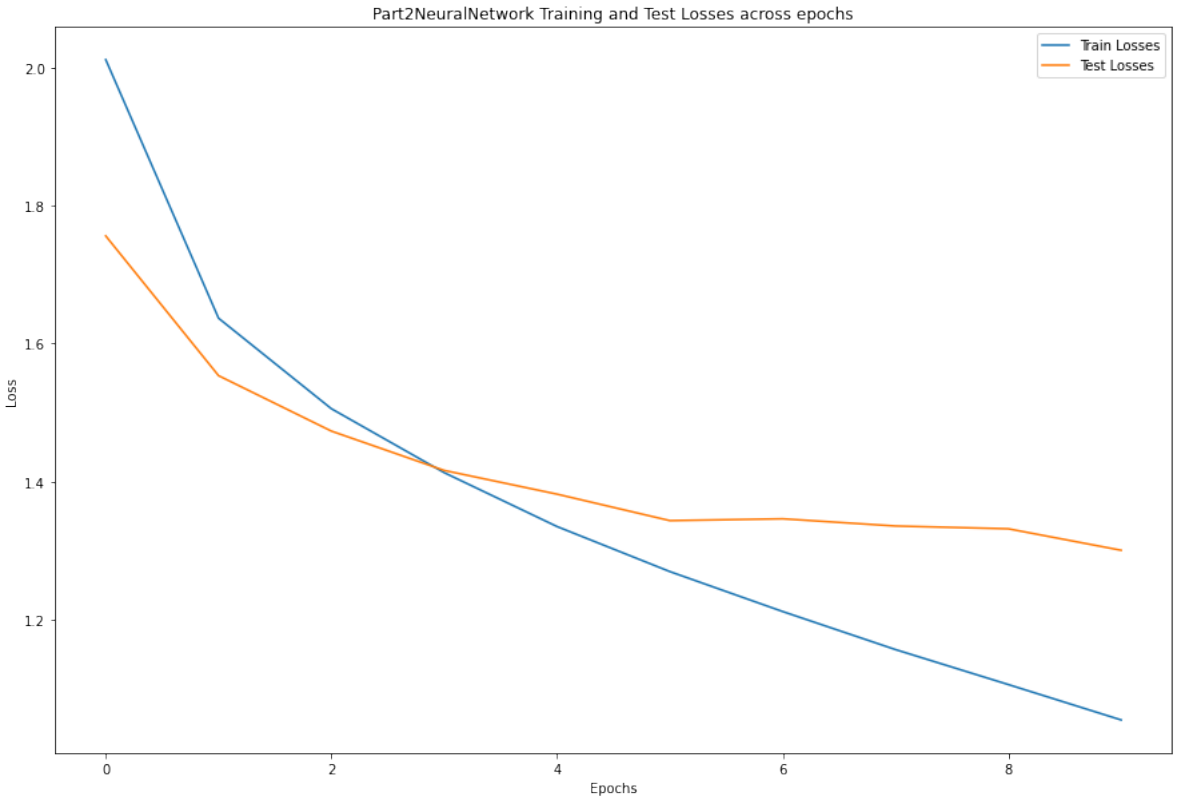
3. *Conceptually, what is a fully-connected layer in a neural network, and how is it represented in PyTorch?*

   A fully-connected layer in a neural network is one where every input node is connected to every output node (activation unit) of that layer. These are represented in PyTorch by `torch.nn.Linear()` in the model's definition.

4.
```
# TODO - Complete Part2NeuralNetwork. Include class into your PDF submission!

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

class Part2NeuralNetwork(nn.Module):
    def __init__(self):
        super(Part2NeuralNetwork, self).__init__()
        ## TODO: Define your neural network layers here!
        ## Importantly, any modules which initialize weights should be initialized
        ## as member variables here.
        ## Note: Keep track of the shape of your input tensors in the training
        ## and test sets, because it affects how you define your layers!
        ## You might find this resource helpful:
        ## https://towardsdatascience.com/pytorch-layer-dimensions-what-sizes-should-they-be-
        self.flatten = nn.Flatten()
        self.layer1 = nn.Linear(3072,1000)
        self.layer2 = nn.Linear(1000,1000)
        self.layer3 = nn.Linear(1000,10)


    def forward(self, x):
        ## TODO: This is where you should apply the layers defined in the __init__
        ## method and the ReLU activation functions to the input x.
        #x = x.view(32,-1)
        #x = x.flatten()
        x = self.flatten(x)
        x = F.relu(self.layer1(x))
```

```
        x = F.relu(self.layer2(x))
        return F.relu(self.layer3(x))

# Display model architecture
model = Part2NeuralNetwork()
model.to(device)
```

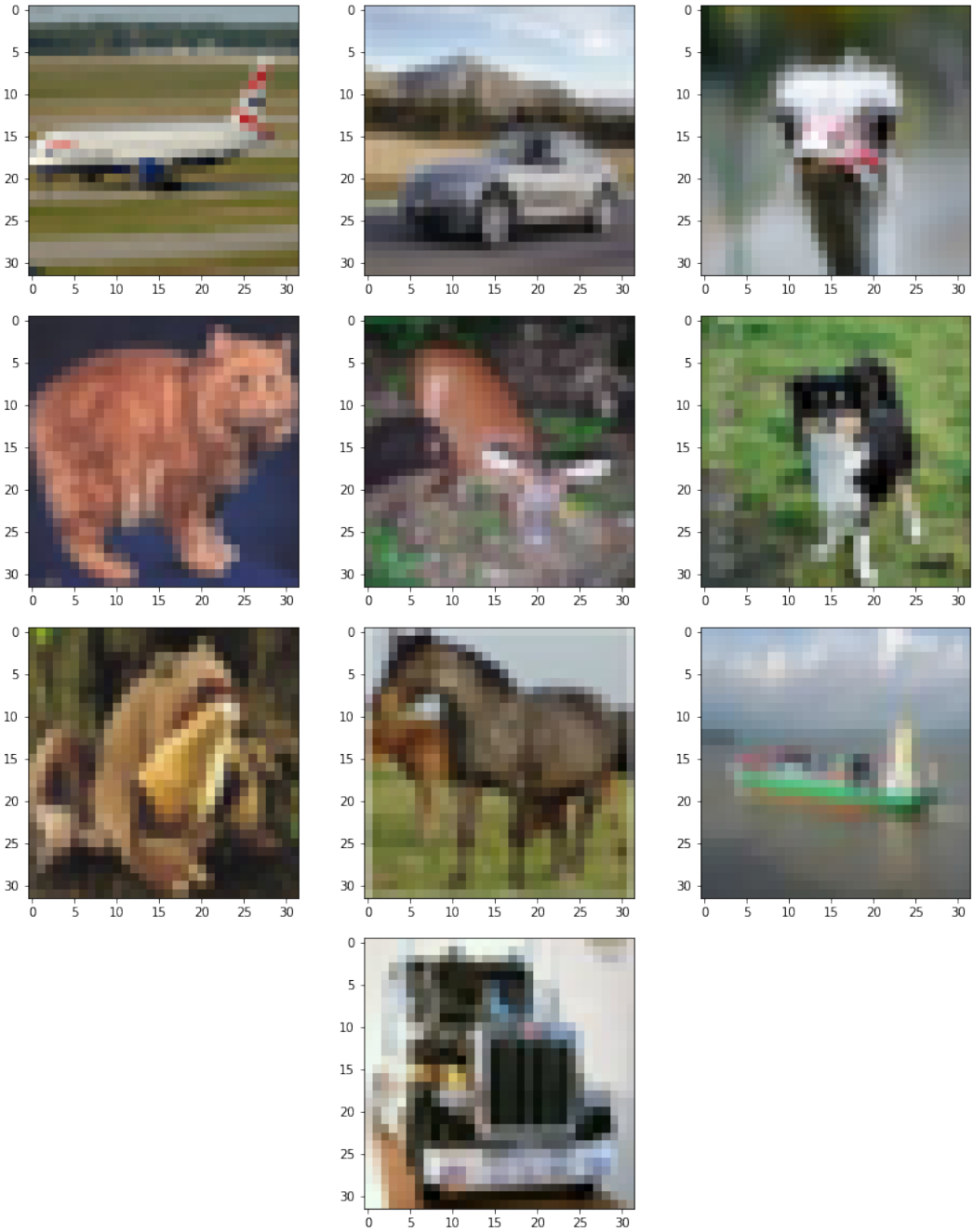
5. The plot of the train and test loss vs. number of epochs is shown below:



6. The Train accuracy of our model was 67%. The Test accuracy of our model was 54%. The train/test precision/recall are reported in the table below:

Class	Train Precision	Train Recall	Test Precision	Test Recall
0 (plane)	0.7702	0.628	0.6695	0.551
1 (car)	0.6932	0.8642	0.6035	0.723
2 (bird)	0.5941	0.505	0.4667	0.393
3 (cat)	0.5491	0.4656	0.3739	0.329
4 (deer)	0.566	0.6514	0.4477	0.505
5 (dog)	0.6296	0.5532	0.4773	0.432
6 (frog)	0.6837	0.7436	0.5714	0.624
7 (horse)	0.7825	0.7582	0.6274	0.596
8 (ship)	0.7475	0.803	0.6492	0.705
9 (truck)	0.6894	0.7444	0.5447	0.597

Below are images of each of the ten classes:



The ship and plane class had strong scores for precision and recall on both the train and test set. This makes sense, since none of the other eight classes look similar to a plane or a ship. Moreover, it's easier to classify these two classes since they both tend to be on a blue background (the sky or the water) and they are typically different shapes. Notice, however, that the plane class does only moderately well on a Test Recall basis. This could potentially be due to variance in the dataset, since the CIFAR images contain many different angles of the plane. In contrast, the CIFAR images tended to include the entire ship in the image, so the ship class had less variance in the images and therefore has a very high score on a Test Recall basis.

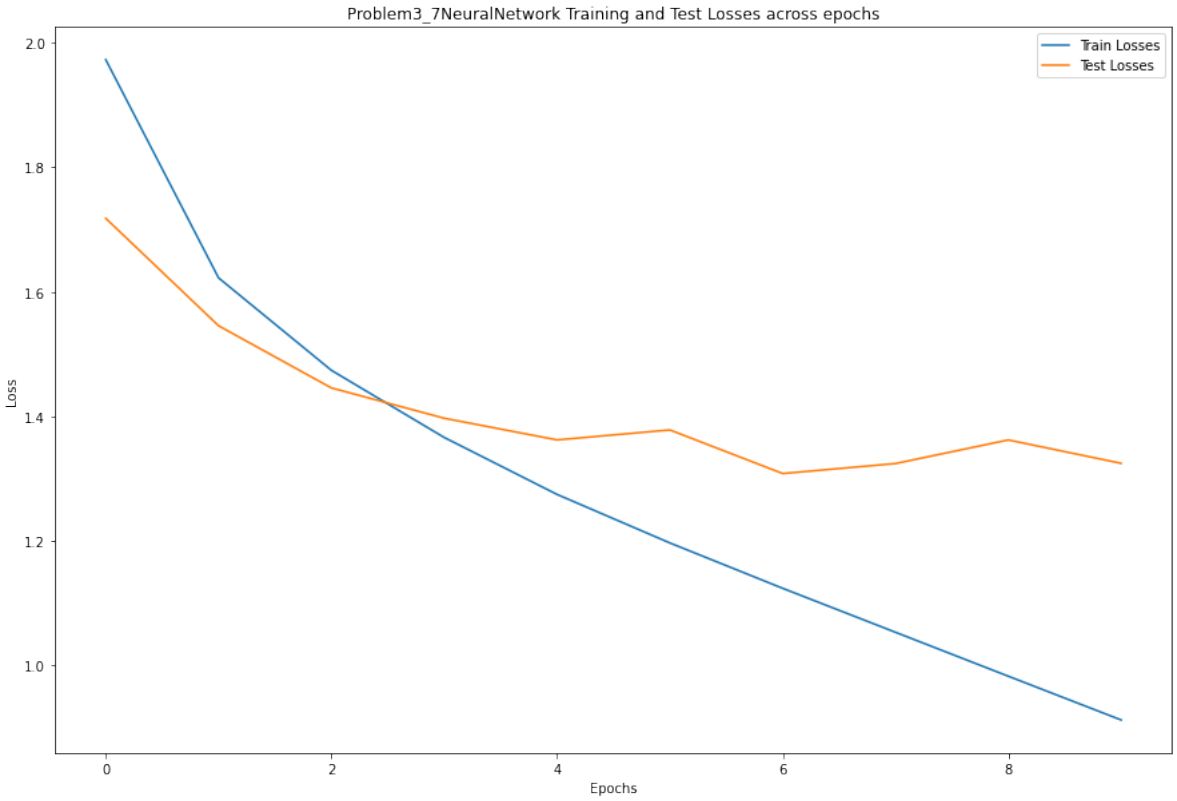
Notice that the bird and cat classes have the worst metrics for Test Precision and Test Recall. For the bird class, there are many different types of birds that are included in the CIFAR dataset. This variation in the images make it difficult for our model to classify the birds class accurately. The variation gives the bird class a very low Test Recall. The example bird image shown above is a good example of why the model performs poorly on a Test Precision basis. The example bird image is just the head of an ostrich, and our model likely misclassified this image as one of the other classes (likely another animal class) since most of the images of birds in the CIFAR data are of smaller birds and of the entire bird. Moreover, a bird in flight has a very different shape from a bird that is standing with its wings closed, which is another reason our model has difficult precisely classifying images of birds. The similar intuition can also be applied to the cat class, since the CIFAR images of cats include cats of many different colors and many different perspectives of cats. In particular, images of a cat curled and laying down are a very different shape from images of a cat standing up (such as the example cat image shown above). In particular, it's likely that the model misclassifies many of the cat images as one of the other classes (likely the dog class).

For the classes where the Test Precision is much greater than the Test Recall, for example the plane class, this implies that when the model does predict that class, it is fairly certain about its prediction, but its lower Test Recall implies that for many of the images of that class the model is very uncertain and misclassifies the image as another class. We can interpret these classes as having a small decision region. We can also consider the converse for classes where the Test Recall is much greater than the Test Precision. We can interpret theses classes as having a large decision region, as a high Test Recall means it is classifying many of the images of this class correctly, but its low Test Precision implies that many other images of other classes are misclassified as being this class.

7. The neural network code is below:

```
class Problem3_7NeuralNetwork(nn.Module):
    def __init__(self):
        super(Problem3_7NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.layer1 = nn.Linear(3072,4000)
        self.layer2 = nn.Linear(4000,5000)
        self.layer3 = nn.Linear(5000,5000)
        self.layer4 = nn.Linear(5000,10)


    def forward(self, x):
        x = self.flatten(x)
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        x = F.relu(self.layer3(x))
        return F.relu(self.layer4(x))

# Display model architecture
model = Problem3_7NeuralNetwork()
model.to(device)
```

The model had a train accuracy of 72% and a test accuracy of 54%.
The graph of the train/test loss is below:



The Train/Test Precision/Recall are gven in the table below:

Class	Train Precision	Train Recall	Test Precision	Test Recall
0 (plane)	0.8672	0.6046	0.6848	0.454
1 (car)	0.8575	0.8824	0.6715	0.648
2 (bird)	0.6399	0.6024	0.4545	0.444
3 (cat)	0.521	0.6462	0.3635	0.454
4 (deer)	0.673	0.6834	0.4923	0.479
5 (dog)	0.7481	0.4836	0.5406	0.353
6 (frog)	0.7337	0.8468	0.5598	0.669
7 (horse)	0.8274	0.8064	0.6353	0.608
8 (ship)	0.6643	0.917	0.5409	0.774
9 (truck)	0.865	0.7968	0.5912	0.554

We find in our four metrics (loss, accuracy, precision, and recall) that this model performs significantly better on the training dataset, but performs roughly the same the test dataset as the model from Question 3.4. I tried multiple combinations of adding or removing layers and increasing or decreasing the number of nodes in each hidden layer. I was unable to find any combination that improved the test accuracy above 54%, however, this model is the only one that performed the same on the test dataset as the model from Question 3.4, and also performed better on the training dataset. The model architecture was constructed by adding one more additional hidden layer and increasing the number of nodes in each hidden layer to either 4000 or 5000. Recall from lecture that fully connected neural networks don't suffer from overfitting as much as our other statistical models that we've seen in class. The intuition for this is that some weights can just be set to approximately 0 if the model is overfitting. Given this fact, it is expected that this new model performs at least as well on the test dataset as the

old model. It also makes intuitive sense that if this model is overfitting, then it will overfit the training dataset and therefore increase the training accuracy.