Assignment 3:

Behnam Saeedi
(Saeedib@oregonstate.edu)
CS535: Deep Learning
Due March 8th 11:59pm
Winter 2019

1 THE NETWORK

The libraries needed to be imported. The original network provided in sample code contain the following layers:

- 4 Convolution layers. 3 to 32, 32 to 32, 32 to 64 and 64 to 64 convolutions.
- Pooling layer.
- two fully connected layer with linear transformation

ReLU is used on the convolutions and connected layers.

```
super(Net, self).__init__()
self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
self.conv2 = nn.Conv2d(32, 32, 3, padding=1)
self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
self.conv4 = nn.Conv2d(64, 64, 3, padding=1)
self.pool = nn.MaxPool2d(2, 2)
self.fc1 = nn.Linear(64 * 8 * 8, 512)
self.fc2 = nn.Linear(512, 10)
```

The base performance of the network is as follows:

Figure 1. Base NN performance

```
EPOCH: 1 train_loss: 1.09929 train_acc: 0.60774 test_loss: 1.14584 test_acc 0.59050
EPOCH: 2 train_loss: 0.81836 train_acc: 0.71658 test_loss: 0.95353 test_acc 0.66920
EPOCH: 3 train_loss: 0.53262 train_acc: 0.81338 test_loss: 0.81432 test_acc 0.71570
EPOCH: 4 train_loss: 0.37711 train_acc: 0.88052 test_loss: 0.76586 test_acc 0.73740
EPOCH: 5 train_loss: 0.20018 train_acc: 0.93782 test_loss: 0.80206 test_acc 0.74940
EPOCH: 6 train_loss: 0.17408 train_acc: 0.94122 test_loss: 0.97919 test_acc 0.72990
EPOCH: 7 train_loss: 0.12818 train_acc: 0.95890 test_loss: 1.08368 test_acc 0.72770
EPOCH: 8 train_loss: 0.12102 train_acc: 0.95906 test_loss: 1.25812 test_acc 0.72480
EPOCH: 9 train_loss: 0.09325 train_acc: 0.96934 test_loss: 1.32113 test_acc 0.73780
```

The network behaves as expected. We can see that the loss on training slowly reduces and the accuracy increases. The Testing accuracy however, increases for a while and then decreases again. This is generally a sign of overfitting and we can see that in the given number of epochs the testing accuracy is not increasing after the 4th

epoch.

---

2 QUESTION 1

The firs portion requires addition of a batch normalization layer after the first fully connected layer:

```
self.bn1 = nn.BatchNorm1d(512, affine=False)
```

The performance of the network for first section is as follows:

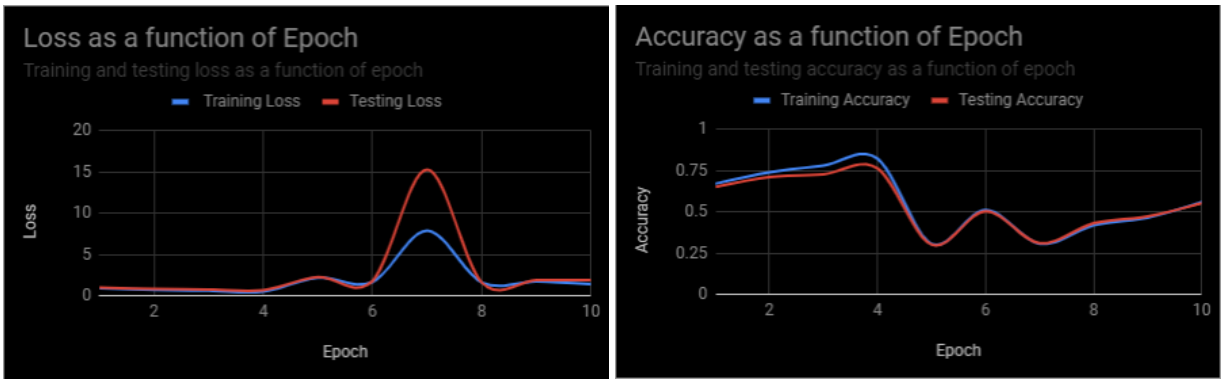

Figure 2. Base NN performance

```
EPOCH: 1 train_loss: 0.94718 train_acc: 0.67036 test_loss: 0.99536 test_acc 0.65040
EPOCH: 2 train_loss: 0.74429 train_acc: 0.73948 test_loss: 0.83676 test_acc 0.71000
EPOCH: 3 train_loss: 0.63229 train_acc: 0.77978 test_loss: 0.78018 test_acc 0.72720
EPOCH: 4 train_loss: 0.52012 train_acc: 0.82184 test_loss: 0.69449 test_acc 0.76250
EPOCH: 5 train_loss: 2.21552 train_acc: 0.30630 test_loss: 2.30058 test_acc 0.30140
EPOCH: 6 train_loss: 1.67069 train_acc: 0.51152 test_loss: 1.79972 test_acc 0.50370
EPOCH: 7 train_loss: 7.89614 train_acc: 0.30654 test_loss: 15.28391 test_acc 0.31020
EPOCH: 8 train_loss: 1.61872 train_acc: 0.41822 test_loss: 1.60967 test_acc 0.43120
EPOCH: 9 train_loss: 1.74386 train_acc: 0.46510 test_loss: 1.89703 test_acc 0.47240
EPOCH: 10 train_loss: 1.42278 train_acc: 0.55942 test_loss: 1.92227 test_acc 0.55230
```

There is an odd and inconsistent result with this run. We can see that there is a spike in loss at epoch 7. This could be due a variety of reasons and difficult to pin point what exactly caused this. Further more, we can see that the network starts to recover afterwards, but in 10 epochs it could not recover fully. What is troubling is that this is a rather unexpected behavior for a term that is inherently designed to decrease over-fitting. The loss spike suggests that the epoch 7 might introduce some very large weights in this particular case.

---

3 QUESTION 2

The task for this section is to add a 512 node fully connected layer. This is done by using the following code:

```
self.fc1_2 = nn.Linear(512, 512)
```

The performance of the network for second section is as follows:

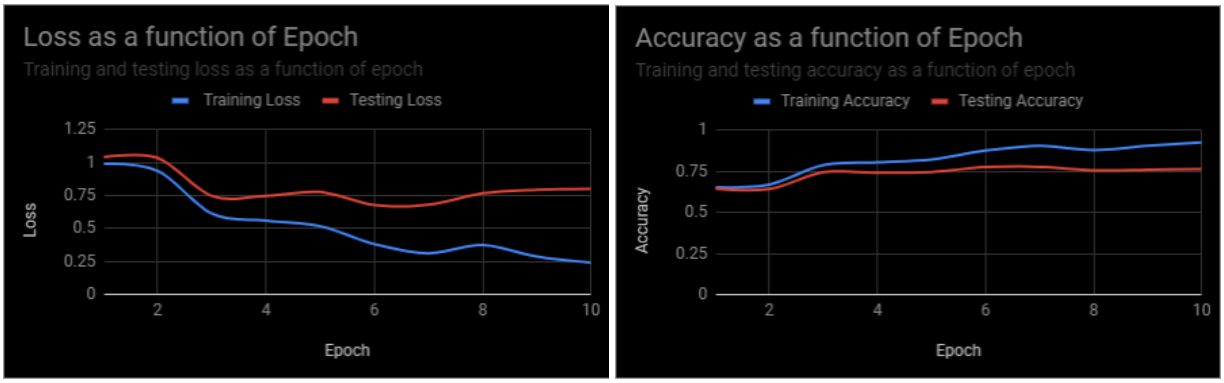

Figure 3. Base NN performance

```
EPOCH: 1 train_loss: 0.99066 train_acc: 0.65342 test_loss: 1.04069 test_acc 0.64450
EPOCH: 2 train_loss: 0.93611 train_acc: 0.66928 test_loss: 1.03319 test_acc 0.64190
EPOCH: 3 train_loss: 0.61189 train_acc: 0.78872 test_loss: 0.74566 test_acc 0.74480
EPOCH: 4 train_loss: 0.55911 train_acc: 0.80362 test_loss: 0.74720 test_acc 0.74010
EPOCH: 5 train_loss: 0.51787 train_acc: 0.82144 test_loss: 0.77675 test_acc 0.74560
EPOCH: 6 train_loss: 0.38210 train_acc: 0.87526 test_loss: 0.67766 test_acc 0.77580
EPOCH: 7 train_loss: 0.31298 train_acc: 0.90428 test_loss: 0.67935 test_acc 0.77680
EPOCH: 8 train_loss: 0.37575 train_acc: 0.87810 test_loss: 0.76663 test_acc 0.75620
EPOCH: 9 train_loss: 0.28800 train_acc: 0.90494 test_loss: 0.79246 test_acc 0.75890
EPOCH: 10 train_loss: 0.24265 train_acc: 0.92614 test_loss: 0.79996 test_acc 0.76370
```

In this model we can see that the loss is continuously decreasing and the accuracy is increasing. In the 10 epochs we can see a significant improvement in the training set. The test set however, does not improve as fast as training and at epoch 10 we can see a rise n the loss again. This could be just an anomaly or a result of an over-fit. This could simple be an anomaly for this particular example and random selection of initial weights since we do not see this repeating in future runs. This particular example helped me understand better how batch normalization behaves.

---

4 QUESTION 3

The task for this section is to use an optimizer and adaptive scheduling in order to tune the network. This task could be done by using any of the following packages:

- RMSprop
- Adagrad
- Adamax
- SGD

And many more. The optimization I used was Adamax.

```
optimizer = torch.optim.Adamax(net.parameters(), lr=0.001, weight_decay=0.0001)
```

The performance of the network for third section is as follows:

Figure 4. Base NN performance

```
EPOCH: 1 train_loss: 0.88633 train_acc: 0.68904 test_loss: 0.95678 test_acc 0.65920
EPOCH: 2 train_loss: 0.66534 train_acc: 0.76736 test_loss: 0.78794 test_acc 0.72710
EPOCH: 3 train_loss: 0.54207 train_acc: 0.81308 test_loss: 0.73174 test_acc 0.74390
EPOCH: 4 train_loss: 0.47223 train_acc: 0.83778 test_loss: 0.74291 test_acc 0.74880
EPOCH: 5 train_loss: 0.39003 train_acc: 0.86338 test_loss: 0.76654 test_acc 0.74570
EPOCH: 6 train_loss: 0.23211 train_acc: 0.92556 test_loss: 0.71557 test_acc 0.77140
EPOCH: 7 train_loss: 0.16797 train_acc: 0.94278 test_loss: 0.78188 test_acc 0.77660
EPOCH: 8 train_loss: 0.11391 train_acc: 0.96304 test_loss: 0.85010 test_acc 0.76860
EPOCH: 9 train_loss: 0.11595 train_acc: 0.96210 test_loss: 0.95165 test_acc 0.75840
EPOCH: 10 train_loss: 0.06750 train_acc: 0.97782 test_loss: 1.01847 test_acc 0.77010
```

The performance over the 10 epochs are very similar to the last case. However, it appears that the curve for loss and accuracy are both much smoother than question 2. Another thing to note here is that we can actually see the the training loss gets very close to 0 and the test loss fluctuates. However, it looks like the testing accuracy continues to increase to epoch 10. The fluctuation we observe is very consistent with examples of optimizers we have seen.

---

5 QUESTION 4

The task for the question four is to change the network model in 2 different way.

5.1 1

The first way I modified the model was by adding a bias to the final fully connected layer and use Adamax optimization

```
self.fc2 = nn.Linear(512, 10,bias=True)
# and
optimizer = torch.optim.Adamax(net.parameters(), lr=0.001, weight_decay=0.00001)
```

The performance of the network for fourth-one section is as follows:

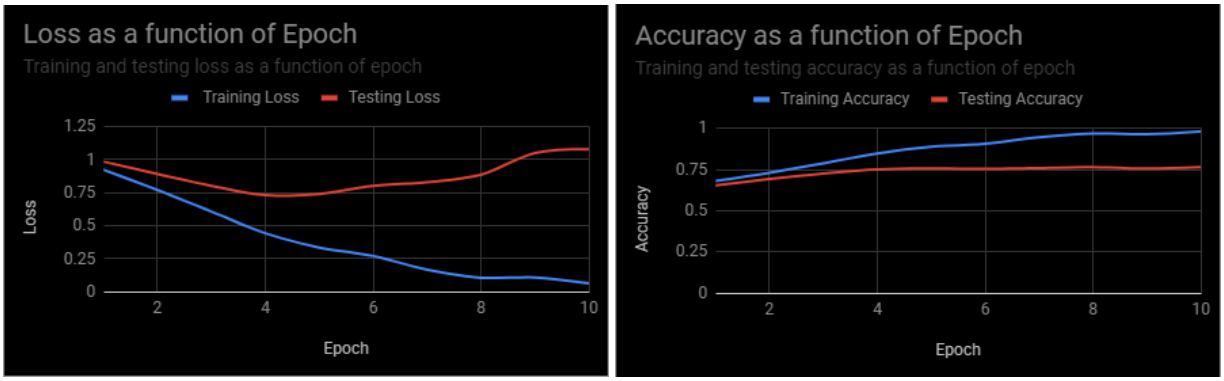

Figure 5. Base NN performance

```
EPOCH: 1 train_loss: 0.92284 train_acc: 0.67996 test_loss: 0.98525 test_acc 0.65390
EPOCH: 2 train_loss: 0.77036 train_acc: 0.72980 test_loss: 0.89148 test_acc 0.69150
EPOCH: 3 train_loss: 0.60752 train_acc: 0.78702 test_loss: 0.80159 test_acc 0.72530
EPOCH: 4 train_loss: 0.44322 train_acc: 0.84744 test_loss: 0.73110 test_acc 0.75050
EPOCH: 5 train_loss: 0.33486 train_acc: 0.88780 test_loss: 0.74054 test_acc 0.75630
EPOCH: 6 train_loss: 0.27081 train_acc: 0.90554 test_loss: 0.80134 test_acc 0.75390
EPOCH: 7 train_loss: 0.16732 train_acc: 0.94556 test_loss: 0.82942 test_acc 0.75860
EPOCH: 8 train_loss: 0.10637 train_acc: 0.96720 test_loss: 0.88640 test_acc 0.76370
EPOCH: 9 train_loss: 0.10819 train_acc: 0.96330 test_loss: 1.05054 test_acc 0.75520
EPOCH: 10 train_loss: 0.06391 train_acc: 0.97980 test_loss: 1.07900 test_acc 0.76420
```

This tuning gave a very good results for the model on training. The test loss and accuracy are fairly good as well. It does have a similar behavior to other networks with respect test loss and accuracy. Originally we tried to use Adagrad for this approach. The result for Adagrad was very poor which raised reason to believe the problem that we are trying to solve is non-convex. Therefore we used Adamax instead.

5.2 2

The second way I modified the model was by Adding bias to all convolutional layers and adding a new batch normalization layer before the second fully connected layer.

```
self.conv1 = nn.Conv2d(3, 32, 3, padding=1, bias=True)
self.conv2 = nn.Conv2d(32, 32, 3, padding=1, bias=True)
self.conv3 = nn.Conv2d(32, 64, 3, padding=1, bias=True)
self.conv4 = nn.Conv2d(64, 64, 3, padding=1, bias=True)
# and
self.bn2 = nn.BatchNorm1d(512, affine=True)
```

The performance of the network for fourth-two section is as follows:

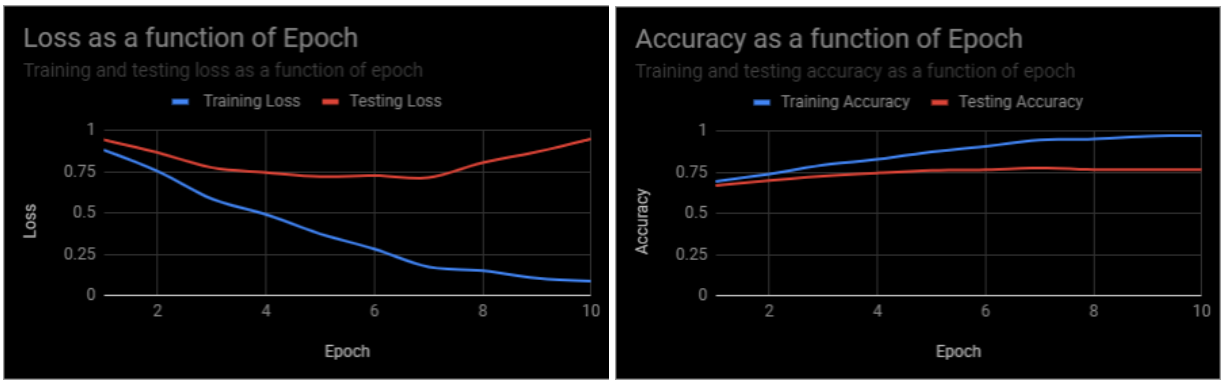

Figure 6. Base NN performance

```
EPOCH: 1 train_loss: 0.88162 train_acc: 0.69450 test_loss: 0.94334 test_acc 0.66960
EPOCH: 2 train_loss: 0.75255 train_acc: 0.73964 test_loss: 0.86545 test_acc 0.70030
EPOCH: 3 train_loss: 0.58522 train_acc: 0.79420 test_loss: 0.77483 test_acc 0.72650
EPOCH: 4 train_loss: 0.49125 train_acc: 0.82856 test_loss: 0.74492 test_acc 0.74590
EPOCH: 5 train_loss: 0.37371 train_acc: 0.87354 test_loss: 0.71989 test_acc 0.76080
EPOCH: 6 train_loss: 0.28221 train_acc: 0.90680 test_loss: 0.72683 test_acc 0.76510
EPOCH: 7 train_loss: 0.17402 train_acc: 0.94622 test_loss: 0.71416 test_acc 0.77570
EPOCH: 8 train_loss: 0.15099 train_acc: 0.95138 test_loss: 0.80516 test_acc 0.76610
EPOCH: 9 train_loss: 0.10550 train_acc: 0.96882 test_loss: 0.86933 test_acc 0.76600
EPOCH: 10 train_loss: 0.08771 train_acc: 0.97268 test_loss: 0.94846 test_acc 0.76480
```

We can see that the network seems to decrease loss and improve accuracy very fast. This normalization is more in line with what you expect from regularization of the model.