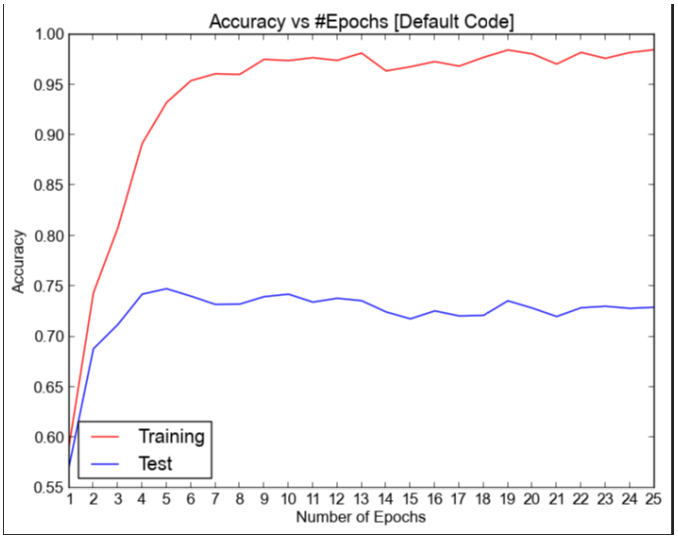
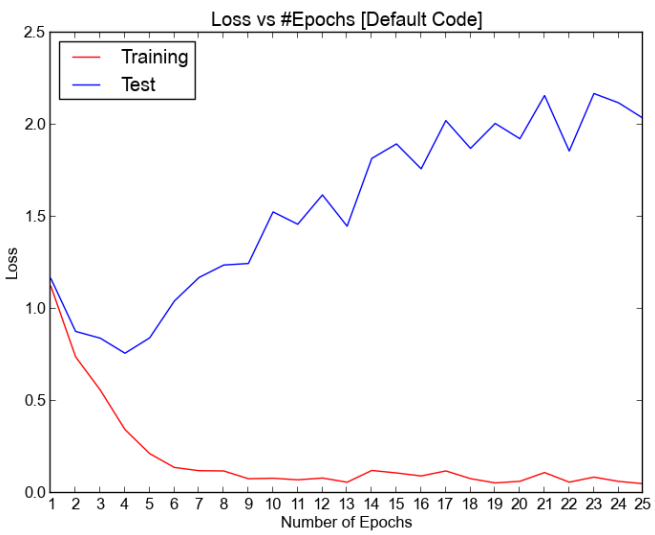
Assignment 3 Report

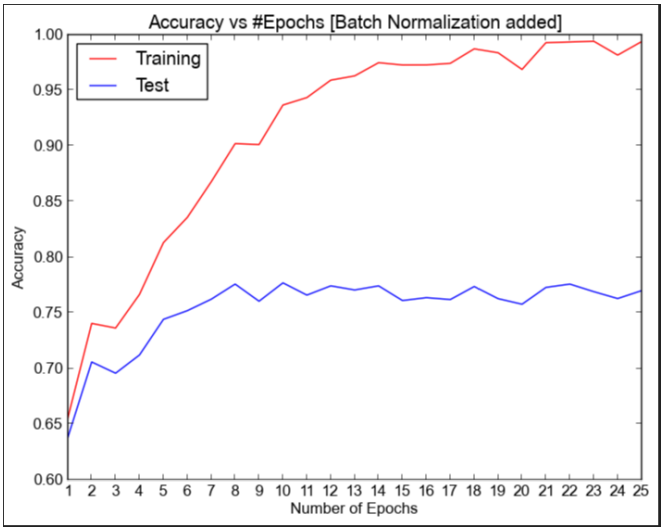
Before completing any part of the assignment, the comparison between training and testing accuracy, training vs testing loss is shown in the following graphs. The epoch was set to 25 from 10. After 25 epochs the training accuracy reached 98.4% and the testing accuracy was 72.9%

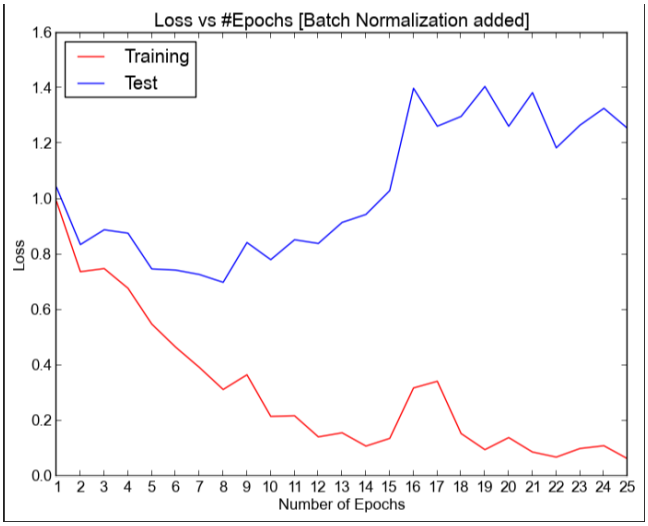




Part 1:

We added batch normalization after our 1st fully connected layer using BatchNorm1d(). Then we ran the code and the 2 figures: training and testing accuracy, training vs testing loss is given below:





From the images we can see that because of batch normalization training accuracy reaches 95% after 12 epochs whereas it reaches the same accuracy without batch normalization after just 7 epochs. Also applying batch normalization, we reached maximum testing accuracy of 77.670% at 24th epoch.

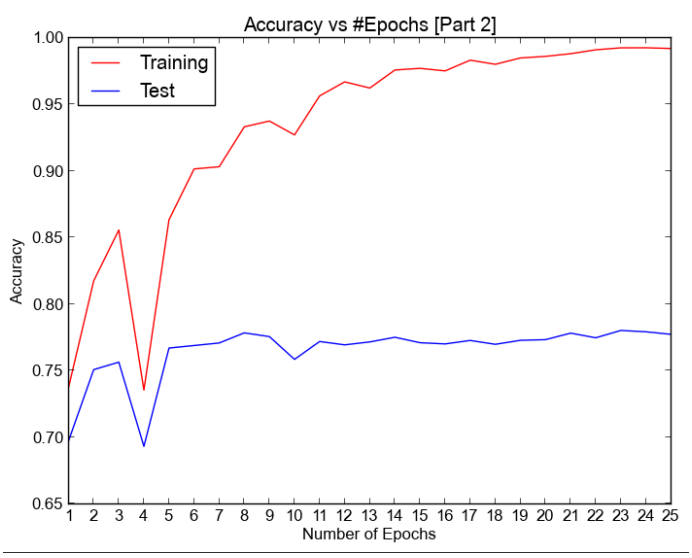
Batch normalization works differently during training phase and evaluation phase. net.train() and net.eval() automatically switches the batch normalization function depending which phase it's in. And the model after batch normalization was saved to 'mytraining.pth' file (state of the parameters after 10 epochs were stored).

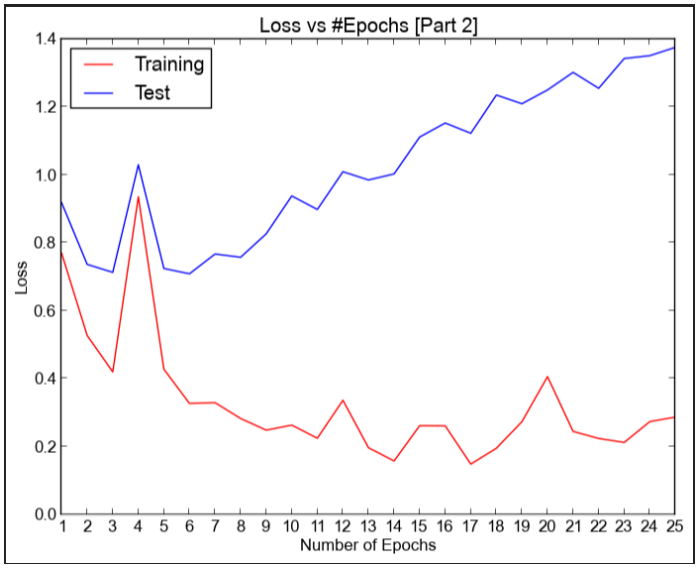
Part 2:

For part 2 we added a new fully connected layer of 512 units after our 1st fully connected layer using nn.Linear(512, 512). Then we also used mytraining.pth to load pretrained weights of our parameters using the following lines of code.

```
pretrained_dict = torch.load('mytraining.pth')
net = Net().cuda()
net.load_state_dict(pretrained_dict, strict = False)
```

As mytraining.pth did not contain weights for the newly added fully connected layer, we needed to add strict = false. The following figures show training and testing accuracy, training vs testing loss after completion of part 2.





We can see from the figure that the main difference of using pretrained weights are from the starting of epoch 1 we are getting very good training accuracy (74% compared to 63%) of part 2 and testing accuracy (70% compared to 62%) of part 2. Also, after only 6 epochs the testing accuracy gets saturated to 77~78%. We achieved highest testing accuracy of 78.020% in epoch 21 and got above 99% training accuracy after 22 epochs.

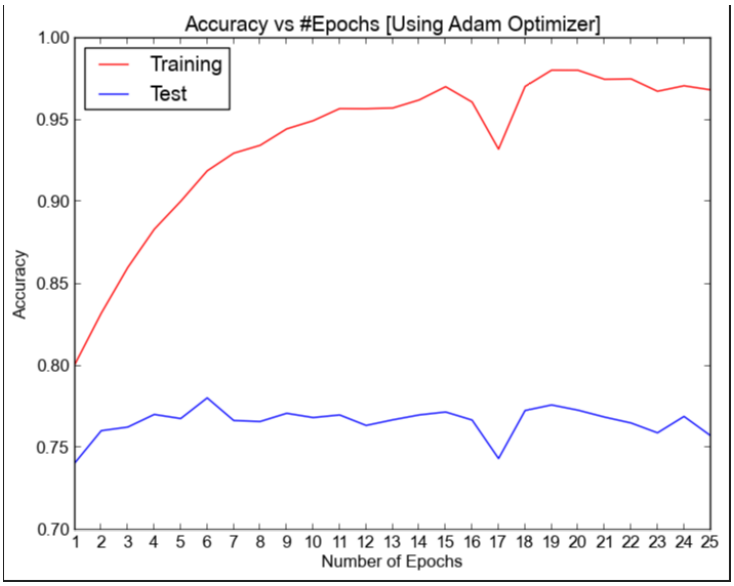
We trained and saved our new model of part 3 using torch.save(net.state_dict(), 'newmodelparameters2.pth') into newmodelparameters2.pth file.

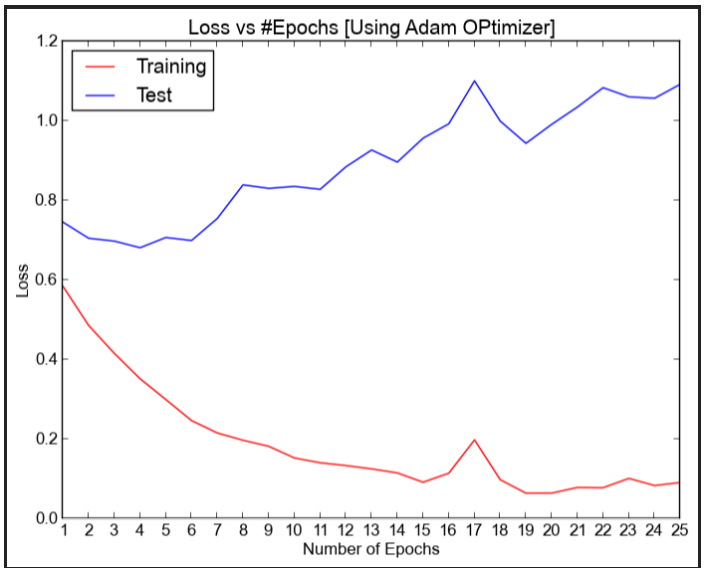
Part 3:

For part 3 we used the Adam optimizer.

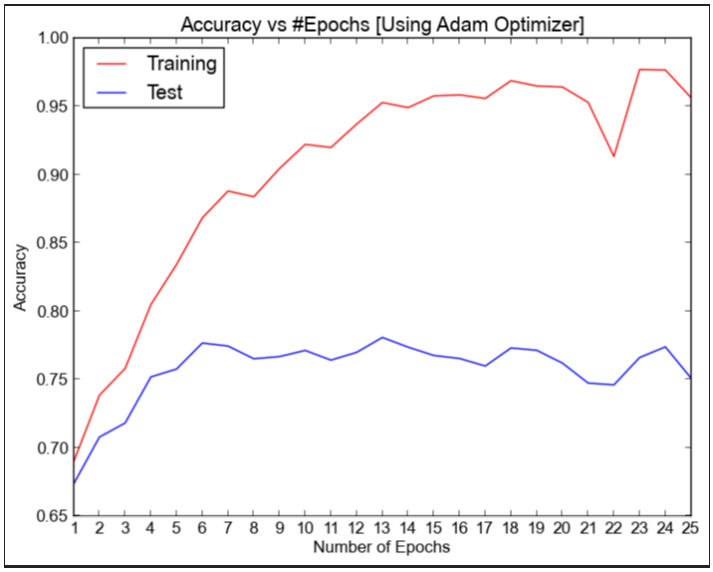
optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)

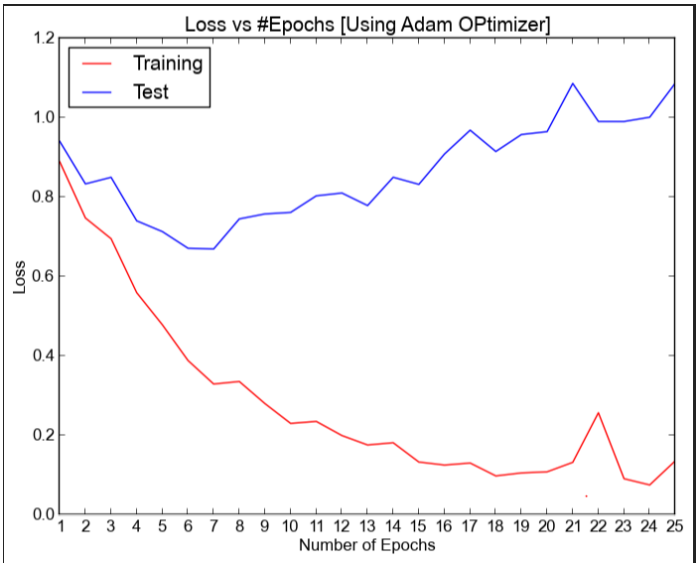
In part 3 using the pretrained weights (mytraining.pth) we got the following figures:





And without using the pretrained weights we got the following figures:





We achieved our best test accuracy of 78.040% using Adam optimizer on pretrained weights just only after 6 epochs.

Part 4:

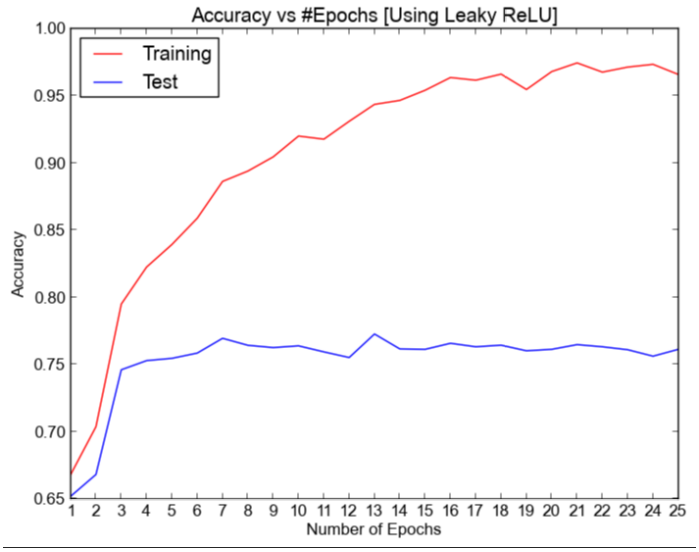
For the part 4 no pretrained weights were used. Parameters were randomly initialized.

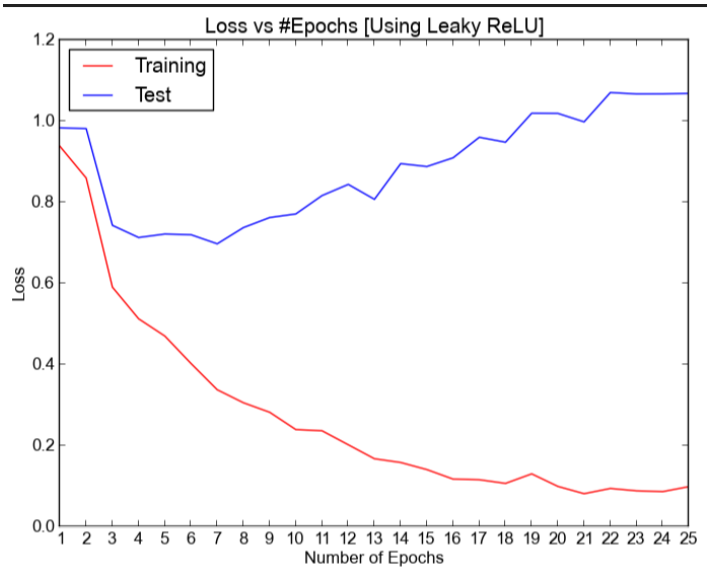
Tuning the network changing the activation function:

By changing all the activation function from ReLU to LeakyReLU I tried to tune the network.

```
def forward(self, x):
    x = F.leaky_relu(self.conv1(x))
    x = F.leaky_relu(self.conv2(x))
    x = self.pool(x)
    x = F.leaky_relu(self.conv3(x))
    x = F.relu(self.conv4(x))
    x = self.pool(x)
    x = x.view(-1, self.num_flat_features(x))
    x = F.leaky_relu(self.fc1(x))
    x = self.bn1(x)
    x = self.newfc(x)
    x = self.fc2(x)
    return x
```

After the change the following figures are generated





Then I changed the number of hidden units in our 1st fully connect layer from 512 to 1024. It changed the number of parameters of the 2nd layer in other parts (512 * 512) to (1024 * 512). I also added l2 regularization to the optimizer using weight_decay = 1e-4. Maximum training accuracy of 78.3% was reached using regulaization and batch normalization. The figures comparing training and testing loss, training and testing accuracy are also given below:

```python
class Net(nn.Module):
    def __init__(self):
        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, 1024)
        self.bn1 = nn.BatchNorm1d(1024)
        #self.newfc = nn.Linear(512, 512)
        #self.fc2 = nn.Linear(512, 10)
        self.fc2 = nn.Linear(1024, 512)
        self.bn2 = nn.BatchNorm1d(512)
        self.fc3 = nn.Linear(512, 10)

    def forward(self, x):
        x = F.leaky_relu(self.conv1(x))
        x = F.leaky_relu(self.conv2(x))
        x = self.pool(x)
        x = F.leaky_relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = self.pool(x)
        x = x.view(-1, self.num_flat_features(x))
        x = F.leaky_relu(self.fc1(x))
        x = self.bn1(x)
        x = self.fc2(x)
        x = self.bn2(x)
        x = self.fc3(x)
        #x = self.newfc(x)
        #x = self.fc2(x)
        return x
```