lab8

November 22, 2022

1 ECE 5470 Lab 8 Report

1.0.1 by Cynthia Li

[33]:
```
from v4 import vd
from PIL import Image
```

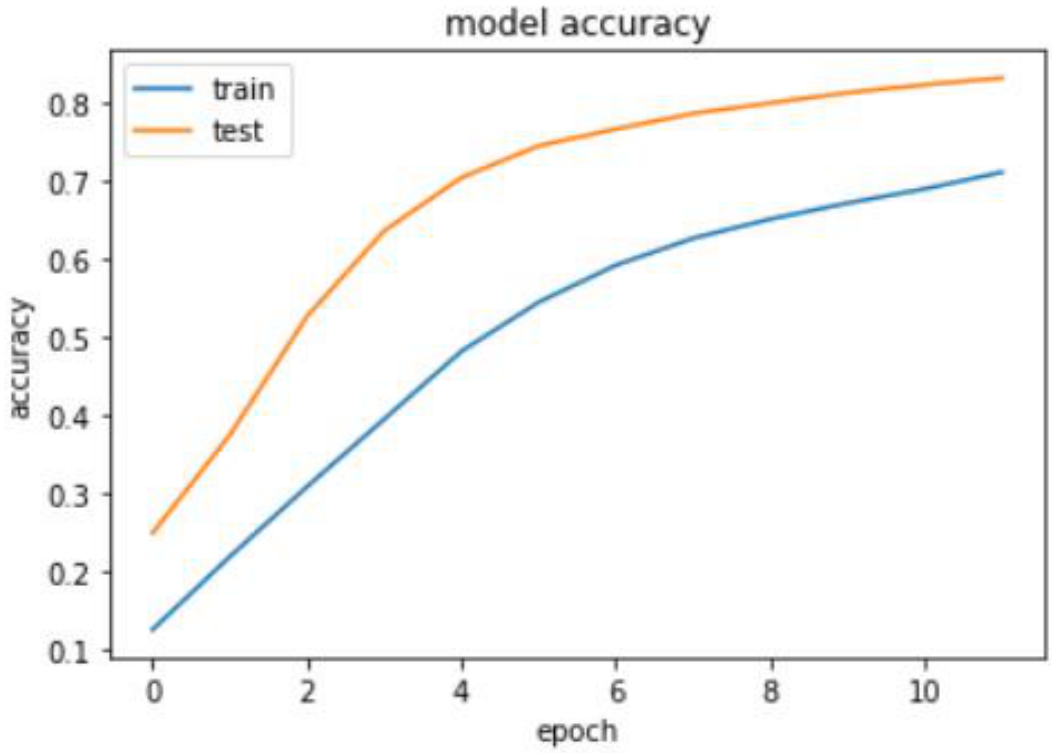
1.1 1. Model evaluation WITH GPU

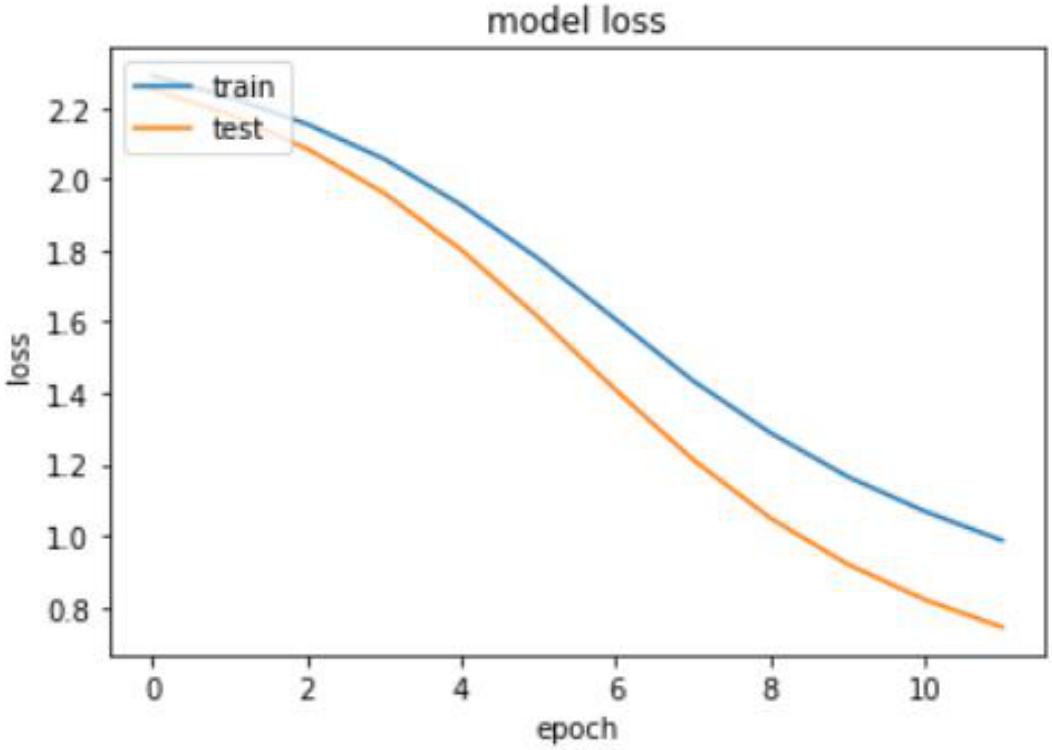
1.1.1 Part 2

Model K1 Epoch 1/12
469/469 [==============================] - 106s 223ms/step - loss: 2.2862 - accuracy: 0.1338 - val_loss: 2.2567 - val_accuracy: 0.2871
Epoch 2/12
469/469 [==============================] - 96s 205ms/step - loss: 2.2395 - accuracy: 0.2413 - val_loss: 2.1999 - val_accuracy: 0.4572
Epoch 3/12
469/469 [==============================] - 94s 201ms/step - loss: 2.1798 - accuracy: 0.3411 - val_loss: 2.1259 - val_accuracy: 0.5565
Epoch 4/12
469/469 [==============================] - 94s 201ms/step - loss: 2.1020 - accuracy: 0.4183 - val_loss: 2.0255 - val_accuracy: 0.6164
Epoch 5/12
469/469 [==============================] - 96s 205ms/step - loss: 1.9948 - accuracy: 0.4855 - val_loss: 1.8892 - val_accuracy: 0.6668
Epoch 6/12
469/469 [==============================] - 96s 204ms/step - loss: 1.8587 - accuracy: 0.5360 - val_loss: 1.7157 - val_accuracy: 0.7119
Epoch 7/12
469/469 [==============================] - 95s 202ms/step - loss: 1.6955 - accuracy: 0.5770 - val_loss: 1.5144 - val_accuracy: 0.7428
Epoch 8/12
469/469 [==============================] - 96s 205ms/step - loss: 1.5219 - accuracy: 0.6122 - val_loss: 1.3105 - val_accuracy: 0.7694
Epoch 9/12
469/469 [==============================] - 95s 203ms/step - loss: 1.3606 - accuracy: 0.6389 - val_loss: 1.1308 - val_accuracy: 0.7878

```
Epoch 10/12
469/469 [==============================] - 96s 204ms/step - loss: 1.2257 -
accuracy: 0.6600 - val_loss: 0.9851 - val_accuracy: 0.8046
Epoch 11/12
469/469 [==============================] - 96s 204ms/step - loss: 1.1159 -
accuracy: 0.6827 - val_loss: 0.8718 - val_accuracy: 0.8162
Epoch 12/12
469/469 [==============================] - 96s 204ms/step - loss: 1.0297 -
accuracy: 0.7007 - val_loss: 0.7848 - val_accuracy: 0.8240
Time Elapsed: 1155.195288 seconds
Test loss: 0.7847607135772705
Test accuracy: 0.8240000009536743
Time Elapsed: 4.101032 seconds
```

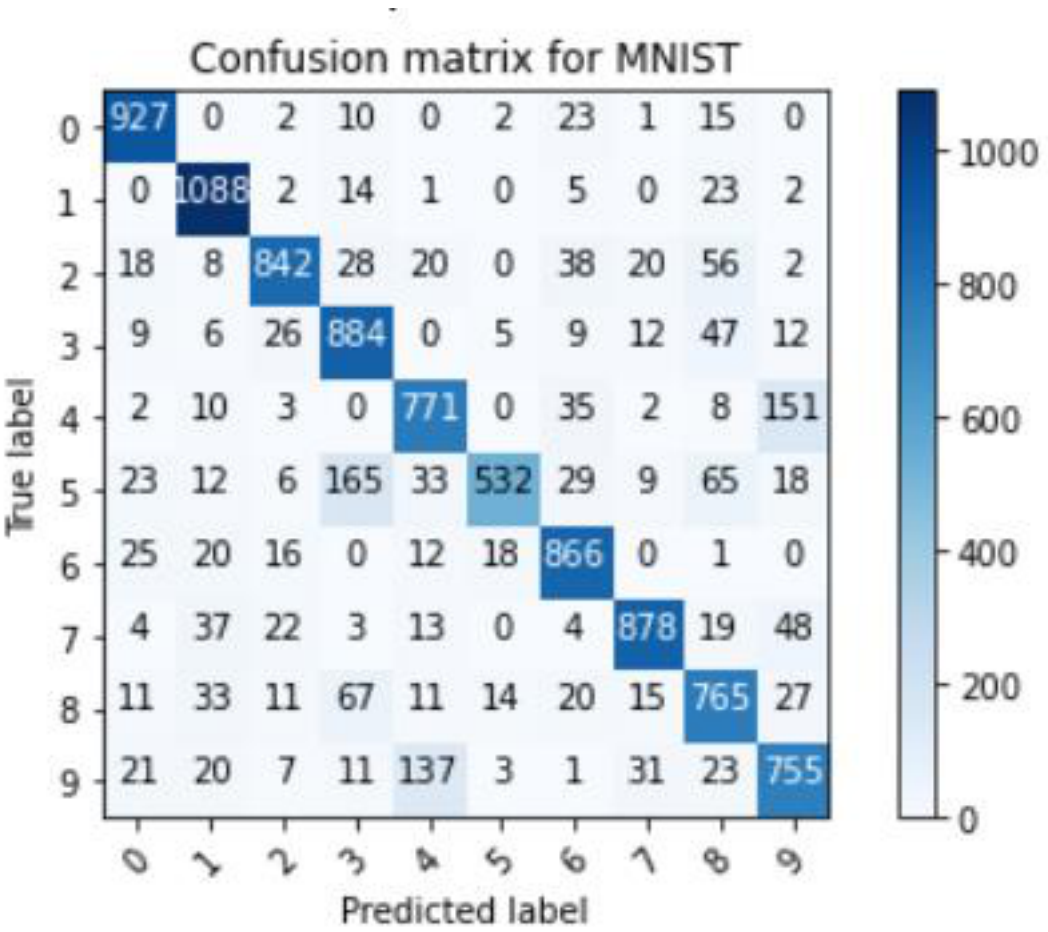
[27]:
```
display(Image.open("./accuracy_epoch.jpg"))
print("Summary of History of Accuracy (Above) Summary of history of Loss␣
  ↪(Below)")
```





Summary of History of Accuracy (Above) Summary of history of Loss (Below)

[28]:
```
display(Image.open("./confusion_matrixK1.jpg"))
print("Confusion Matrix For Keras1 Model on MNIST")
```




```
Confusion Matrix For Keras1 Model on MNIST
```

From the results shown above, we can see that the testing loss and accuracy as 0.7848 and 0.8240. The time elapsed is 1155 seconds. Comparatively to what we have learned from lab 7, this is time consuming (even with the GPU acceleration) with low accuracy and high loss. In fact, when I tried to execute the program without GPU, it took 26414 seconds, which is extremely computationally expensive. From the learning curve plot, we can see that the model does not suffer from overfitting or underfitting. And from the confusion matrix, we can see that the recognition of number 5 is relatively weak, then follows by 4, 8, 9. The best performance is with number 1.

Model P1 EPOCH [1/12]. Running Loss: 180.14 Progress: 42.67 %
EPOCH [1/12]. Running Loss: 233.03 Progress: 85.33 %
Train Avg. Loss: [0.2292] Acc: 0.8843 on 60000.0 images
Validating…
Val Avg. Loss: [0.0813] Acc: 0.9618 on 10000.0 images

EPOCH [2/12]. Running Loss: 31.3 Progress: 42.67 %
EPOCH [2/12]. Running Loss: 55.36 Progress: 85.33 %
Train Avg. Loss: [0.1667] Acc: 0.967 on 60000.0 images
Validating…
Val Avg. Loss: [0.0189] Acc: 0.9736 on 10000.0 images

EPOCH [3/12]. Running Loss: 19.39 Progress: 42.67 %
EPOCH [3/12]. Running Loss: 36.85 Progress: 85.33 %
Train Avg. Loss: [0.0714] Acc: 0.9756 on 60000.0 images
Validating…
Val Avg. Loss: [0.0072] Acc: 0.9798 on 10000.0 images

EPOCH [4/12]. Running Loss: 14.4 Progress: 42.67 %
EPOCH [4/12]. Running Loss: 28.52 Progress: 85.33 %
Train Avg. Loss: [0.0191] Acc: 0.9804 on 60000.0 images
Validating…
Val Avg. Loss: [0.0052] Acc: 0.982 on 10000.0 images

EPOCH [5/12]. Running Loss: 11.58 Progress: 42.67 %
EPOCH [5/12]. Running Loss: 23.63 Progress: 85.33 %
Train Avg. Loss: [0.0513] Acc: 0.9834 on 60000.0 images
Validating…
Val Avg. Loss: [0.004] Acc: 0.9855 on 10000.0 images

EPOCH [6/12]. Running Loss: 10.13 Progress: 42.67 %
EPOCH [6/12]. Running Loss: 20.53 Progress: 85.33 %
Train Avg. Loss: [0.0422] Acc: 0.9856 on 60000.0 images
Validating…
Val Avg. Loss: [0.0031] Acc: 0.9854 on 10000.0 images

EPOCH [7/12]. Running Loss: 8.87 Progress: 42.67 %
EPOCH [7/12]. Running Loss: 18.27 Progress: 85.33 %
Train Avg. Loss: [0.0218] Acc: 0.9873 on 60000.0 images
Validating…
Val Avg. Loss: [0.0014] Acc: 0.9862 on 10000.0 images

EPOCH [8/12]. Running Loss: 8.36 Progress: 42.67 %
EPOCH [8/12]. Running Loss: 16.21 Progress: 85.33 %
Train Avg. Loss: [0.0248] Acc: 0.9888 on 60000.0 images
Validating…
Val Avg. Loss: [0.0016] Acc: 0.9877 on 10000.0 images

EPOCH [9/12]. Running Loss: 7.59 Progress: 42.67 %
EPOCH [9/12]. Running Loss: 14.89 Progress: 85.33 %
Train Avg. Loss: [0.0529] Acc: 0.9897 on 60000.0 images
Validating…
Val Avg. Loss: [0.0013] Acc: 0.9883 on 10000.0 images

EPOCH [10/12]. Running Loss: 6.67 Progress: 42.67 %
EPOCH [10/12]. Running Loss: 13.53 Progress: 85.33 %
Train Avg. Loss: [0.0344] Acc: 0.9908 on 60000.0 images

Validating...
Val Avg. Loss: [0.0011] Acc: 0.9882 on 10000.0 images

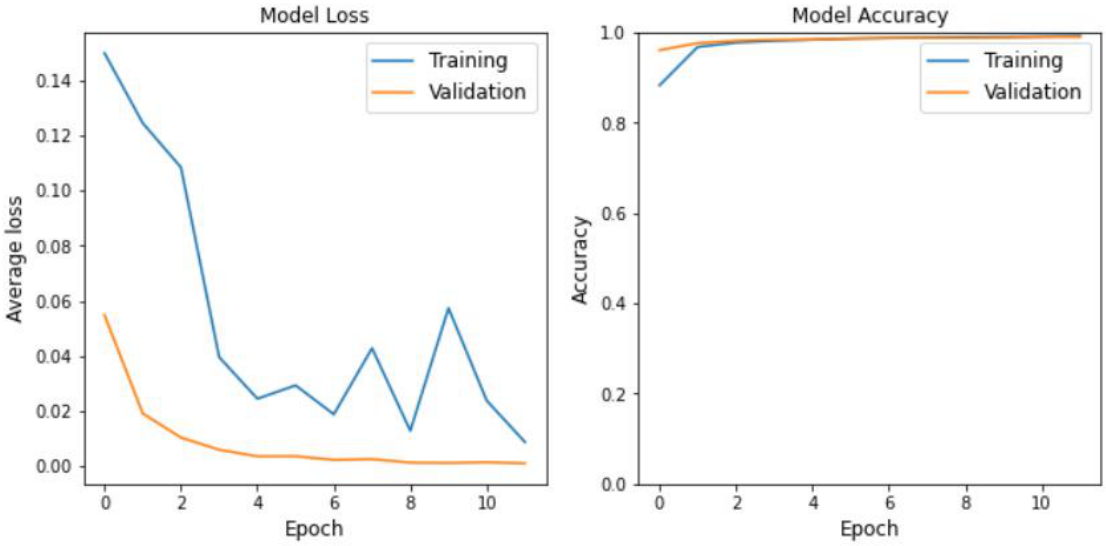
EPOCH [11/12]. Running Loss: 5.91 Progress: 42.67 %
EPOCH [11/12]. Running Loss: 12.19 Progress: 85.33 %
Train Avg. Loss: [0.0087] Acc: 0.9919 on 60000.0 images
Validating...
Val Avg. Loss: [0.0012] Acc: 0.9885 on 10000.0 images

EPOCH [12/12]. Running Loss: 5.49 Progress: 42.67 %
EPOCH [12/12]. Running Loss: 11.32 Progress: 85.33 %
Train Avg. Loss: [0.044] Acc: 0.9922 on 60000.0 images
Validating...
Val Avg. Loss: [0.0008] Acc: 0.9886 on 10000.0 images

Time Elapsed: 105.965357 seconds

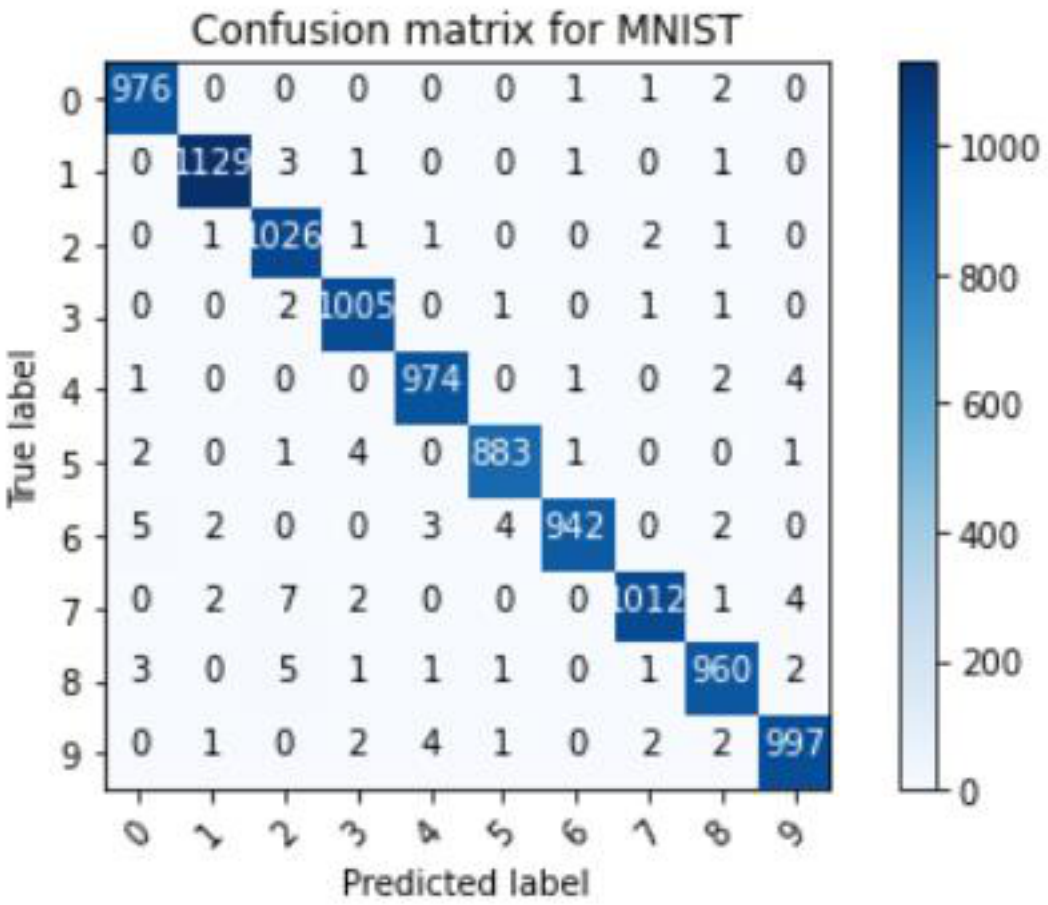
Test Accuracy of the model on the 10000 test images: 99.04 %
Time Elapsed: 20.390108 seconds

[18]:
```
display(Image.open("./loss_accuracyP2.jpg"))
print("Learning curves: Model Loss (Left) Model Accuracy (Right)")
```



Learning curves: Model Loss (Left) Model Accuracy (Right)

[19]:
```
display(Image.open("./confusion_matrixP1.jpg"))
print("Confusion Matrix For Pytorch Model2 on MNIST")
```



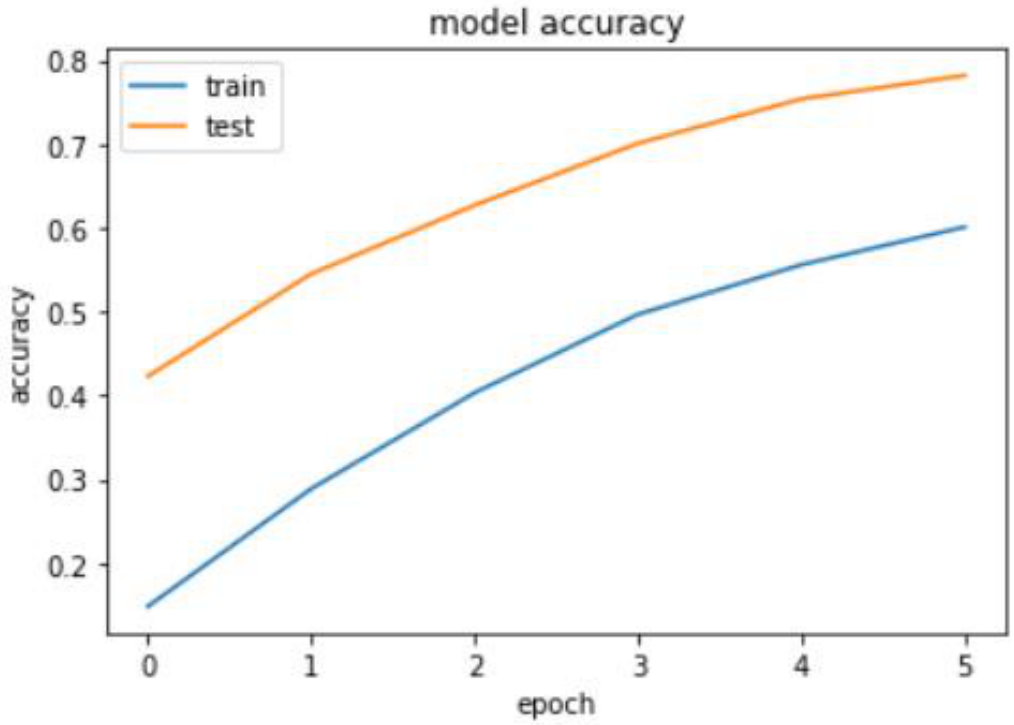
`Confusion Matrix For Pytorch Model2 on MNIST`

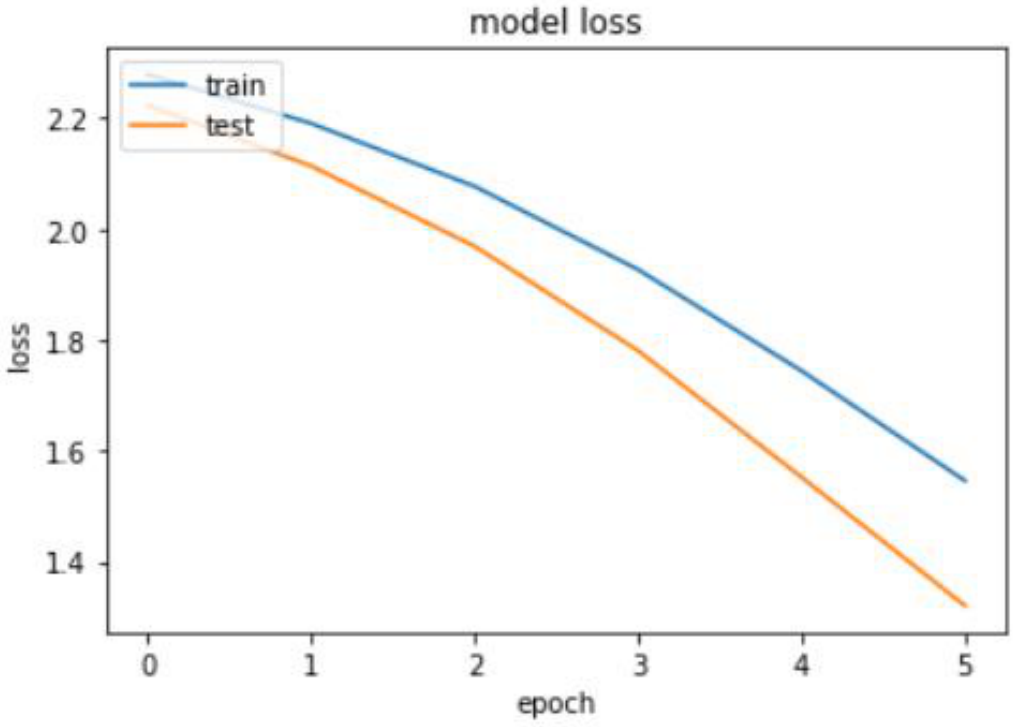
From the results shown above, we can see that the testing loss and accuracy as 0.044 and 0.9922. The time elapsed is 106.0 seconds for training and 20.4s for testing. Comparatively to what we have learned from lab 7 and kera model, this is still time consuming than PCA version of KNN and perceptron even with the GPU acceleration, but made an improvement than regular full set KNN and the kera model. Pytorch provides the highest accuracy and lowest loss than any algorithm or cnn models we've used. From the learning curve plot, we can see that the model does not suffer from overfitting or underfitting. And from the confusion matrix, we can see that the recognition of number 5 is relatively weak, but overall the accuracy is very high.

Model K2 Epoch 1/6
600/600 [==============================] - 152s 250ms/step - loss: 2.2843 - accuracy: 0.1415 - val_loss: 2.2495 - val_accuracy: 0.3249
Epoch 2/6
600/600 [==============================] - 144s 240ms/step - loss: 2.2242 - accuracy: 0.2525 - val_loss: 2.1724 - val_accuracy: 0.4908
Epoch 3/6

```
600/600 [==============================] - 144s 240ms/step - loss: 2.1423 -
accuracy: 0.3555 - val_loss: 2.0653 - val_accuracy: 0.6203
Epoch 4/6
600/600 [==============================] - 144s 240ms/step - loss: 2.0259 -
accuracy: 0.4505 - val_loss: 1.9153 - val_accuracy: 0.7062
Epoch 5/6
600/600 [==============================] - 144s 239ms/step - loss: 1.8720 -
accuracy: 0.5196 - val_loss: 1.7189 - val_accuracy: 0.7593
Epoch 6/6
600/600 [==============================] - 144s 240ms/step - loss: 1.6874 -
accuracy: 0.5734 - val_loss: 1.4897 - val_accuracy: 0.7857
Time Elapsed: 872.331653 seconds
Test loss: 1.4896653890609741
Test accuracy: 0.7857000231742859
Time Elapsed: 5.966268 seconds
```

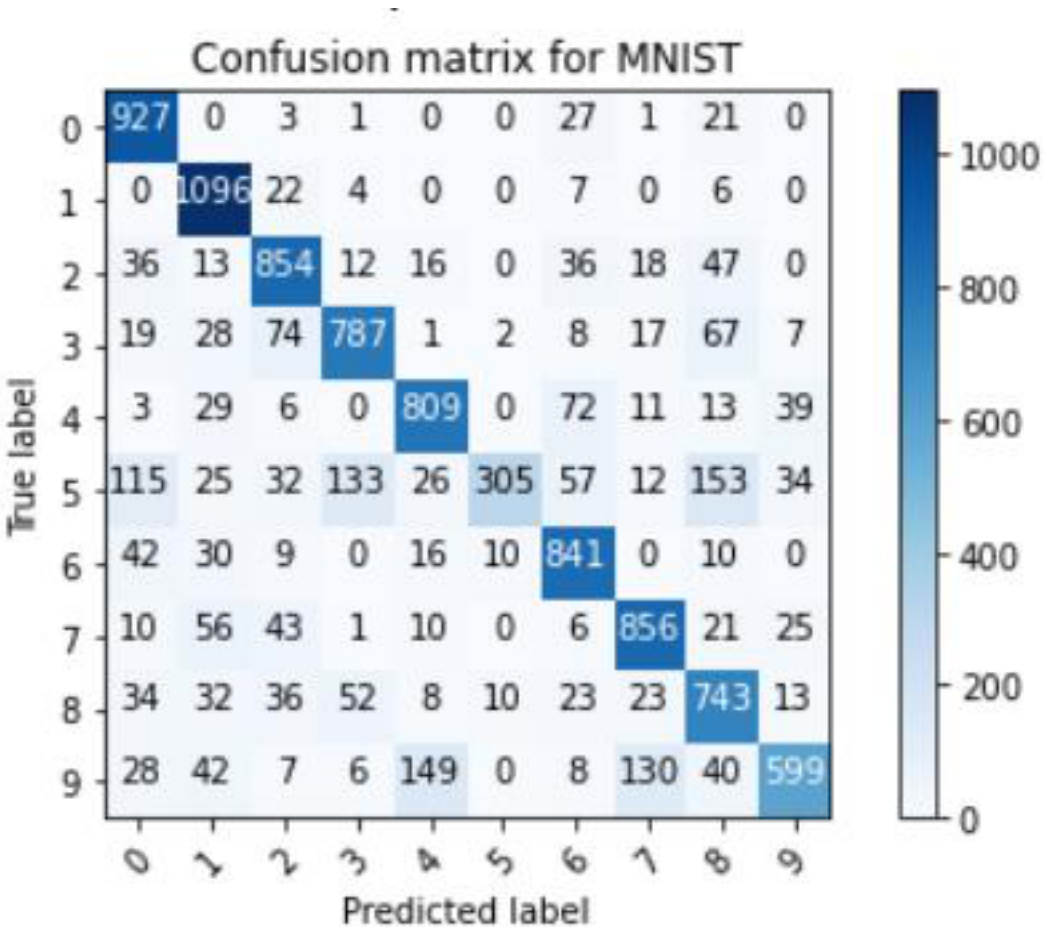
[20]:
```
display(Image.open("./loss_accuracyK2.jpg"))
print("Learning curves: Model Accuracy (Up) Model Loss (Down)")
```





Learning curves: Model Accuracy (Up) Model Loss (Down)

[21]:
```
display(Image.open("./confusion_matrixK2.jpg"))
print("Confusion Matrix For Pytorch Model2 on MNIST")
```




```
Confusion Matrix For Pytorch Model2 on MNIST
```

From the results shown above, we can see that the testing loss and accuracy as 1.4897 and 0.7857. The time elapsed is 872 seconds. Clearly using fewer epoch can reduce the time consumption for training and testing while compromising the accuracy and loss. But still, compare to what we have learned from lab 7 and the pytorch model, this is time consuming (even with the GPU acceleration) with low accuracy and high loss. From the learning curve plot, we can see that the model does not suffer from overfitting or underfitting. And from the confusion matrix, we can see that the recognition of number 5 is very weak, then follows by 3, 8, 9. The best performance is with number 1.

Model P2 EPOCH [1/6]. Running Loss: 188.86 Progress: 33.33 %
EPOCH [1/6]. Running Loss: 246.79 Progress: 66.67 %
EPOCH [1/6]. Running Loss: 282.66 Progress: 100.0 %
Train Avg. Loss: [0.1601] Acc: 0.8936 on 60000.0 images

Validating…
Val Avg. Loss: [0.1884] Acc: 0.964 on 10000.0 images

EPOCH [2/6]. Running Loss: 28.36 Progress: 33.33 % EPOCH [2/6]. Running Loss: 51.76 Progress: 66.67 %
EPOCH [2/6]. Running Loss: 71.21 Progress: 100.0 %
Train Avg. Loss: [0.0483] Acc: 0.97 on 60000.0 images
Validating…
Val Avg. Loss: [0.1075] Acc: 0.9761 on 10000.0 images

EPOCH [3/6]. Running Loss: 17.55 Progress: 33.33 %
EPOCH [3/6]. Running Loss: 33.6 Progress: 66.67 %
EPOCH [3/6]. Running Loss: 48.35 Progress: 100.0 %
Train Avg. Loss: [0.1417] Acc: 0.9784 on 60000.0 images
Validating…
Val Avg. Loss: [0.0699] Acc: 0.9811 on 10000.0 images

EPOCH [4/6]. Running Loss: 13.44 Progress: 33.33 %
EPOCH [4/6]. Running Loss: 25.6 Progress: 66.67 %
EPOCH [4/6]. Running Loss: 38.0 Progress: 100.0 %
Train Avg. Loss: [0.0522] Acc: 0.9829 on 60000.0 images
Validating…
Val Avg. Loss: [0.0464] Acc: 0.9829 on 10000.0 images

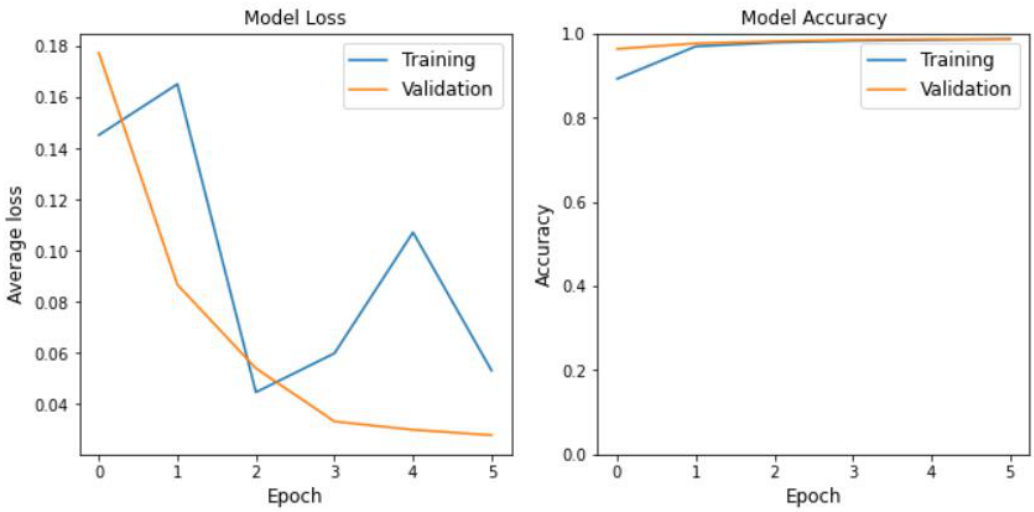
EPOCH [5/6]. Running Loss: 10.36 Progress: 33.33 %
EPOCH [5/6]. Running Loss: 20.96 Progress: 66.67 %
EPOCH [5/6]. Running Loss: 31.44 Progress: 100.0 %
Train Avg. Loss: [0.0827] Acc: 0.9855 on 60000.0 images
Validating…
Val Avg. Loss: [0.0328] Acc: 0.9849 on 10000.0 images

EPOCH [6/6]. Running Loss: 9.25 Progress: 33.33 %
EPOCH [6/6]. Running Loss: 18.25 Progress: 66.67 %
EPOCH [6/6]. Running Loss: 26.98 Progress: 100.0 %
Train Avg. Loss: [0.0631] Acc: 0.9875 on 60000.0 images
Validating…
Val Avg. Loss: [0.038] Acc: 0.9861 on 10000.0 images

Time Elapsed: 305.764682 seconds

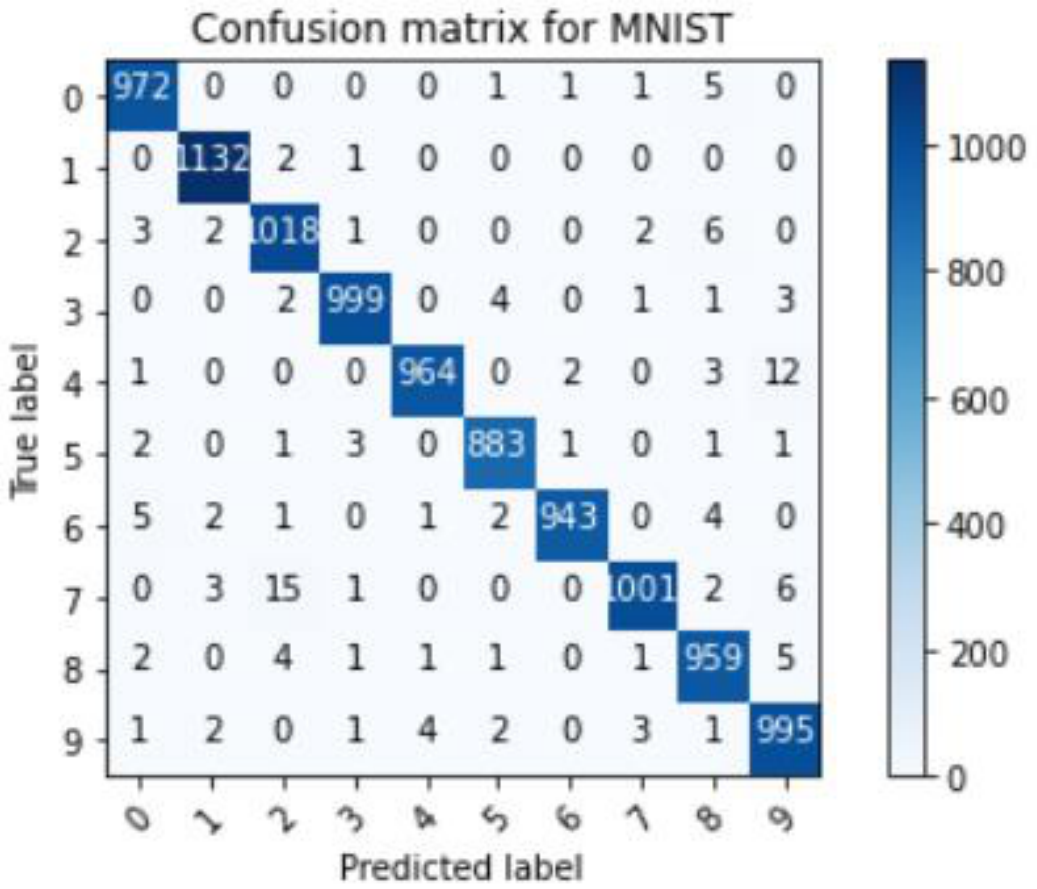
Test Accuracy of the model on the 10000 test images: 98.61 %
Time Elapsed: 3.435184 seconds

[15]:
```
display(Image.open("./loss_accuracy.jpg"))
print("Learning curves: Model Loss (Left) Model Accuracy (Right)")
```



Learning curves: Model Loss (Left) Model Accuracy (Right)

[26]:
```
display(Image.open("./confusion_matrixP2.jpg"))
print("Confusion Matrix For Pytorch Model on MNIST")
```



`Confusion Matrix For Pytorch Model on MNIST`

From the results shown above, we can see that the testing loss and accuracy as 0.038 and 0.9861. The time elapsed is 305 for training and 3.43 for testing. That is to say reducing number of epoch increased the training time for pytorch but reduced the testing time for it while slightly compromising the accuracy and the loss. Overall it still has a very high accuracy. Therefore, as discussed before, compare to what we have learned from lab 7 and kera model, this is still time consuming than PCA version of KNN and perceptron even with the GPU acceleration, but made an improvement than regular full set KNN and the kera model. From the learning curve plot, we can see that the model does not suffer from overfitting or underfitting. And from the confusion matrix, we can see that the recognition of number 5 is relatively weak, but overall the accuracy is very high.

Discussion on GPU I have in fact tested the K1 model with and without GPU. It turns out using GPU can save about 90% time while not compromising the training and testing. This makes a lot of sense because graphic processing unit can efficiently accelerate the creation and rendering of images, videos, animations, etc. by performing fast math calculation and freeing CPU for other tasks. However, just like CPU, each time we run the program, the run time is dependent on the loading of the GPU as well. So even if we run the same program twice at different loading, we would still get very different runtime (when the GPU is highly occupied, the run time is longer).

But overall, GPU highly accelerates the CNN model training and testing process.

1.2 2. New Model Design

In the first two trials, I want to explore the effect of learning rate on the model. As we know, when we choose a large learning rate, the model would converge very fast to a suboptimal solution and result in undesirable divergent behavior in the loss function, but when the learning rate is too small ends up with tiny changes in the weight changes and therefore, very slow computation progress.

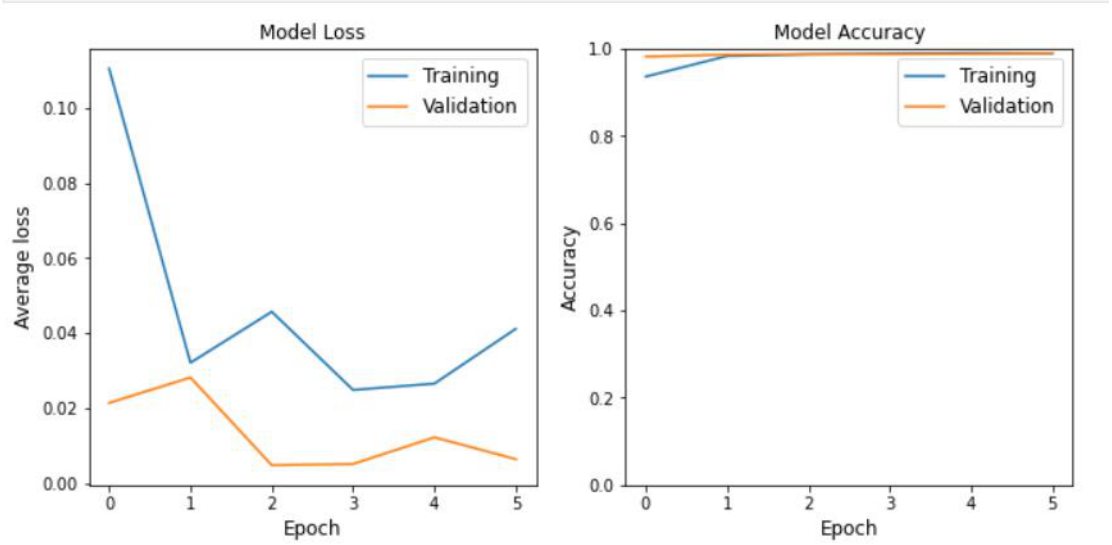
I decided that the original learning rate of 0.002 is too small, making the and I simply changed the learning rate to 0.02, and then to 0.2 to see how big impact it has on both computation time and the accuracy.

Below is the result with Original Learning Rate=0.002 Train Avg. Loss: [0.0958] Acc: 0.9933 on 60000.0 images
Validating…
Val Avg. Loss: [0.0058] Acc: 0.9855 on 10000.0 images
Time Elapsed: 339.236808 seconds
Test Accuracy of the model on the 10000 test images: 98.55 %
Time Elapsed: 3.965011 seconds

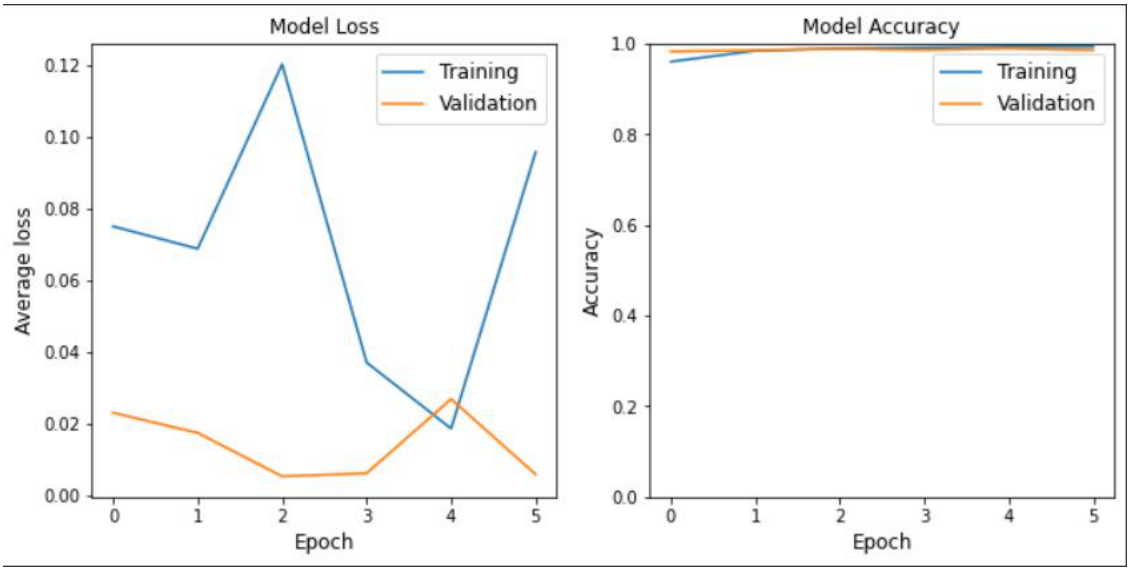
Below is the result for learning rate=0.02: Train Avg. Loss: [0.0077] Acc: 0.9908 on 60000.0 images
Validating…
Val Avg. Loss: [0.0093] Acc: 0.991 on 10000.0 images
Time Elapsed: 341.546268 seconds
Test Accuracy of the model on the 10000 test images: 99.1 %
Time Elapsed: 3.847924 seconds

And below is the result for learning rate=0.2 Train Avg. Loss: [0.1276] Acc: 0.9234 on 60000.0 images
Validating…
Val Avg. Loss: [0.176] Acc: 0.9636 on 10000.0 images
Time Elapsed: 286.054918 seconds
Test Accuracy of the model on the 10000 test images: 96.44 %
Time Elapsed: 3.165447 seconds

[23]:
```
display(Image.open("./plot_lr02.jpg"))
print("Learning rate = 0.02: Pytorch Model Loss (Left) Model Accuracy (Right)")

display(Image.open("./plot_lr2.jpg"))
print("Learning rate = 0.2: Pytorch Model Loss (left) Model Accuracy (Right)")
```

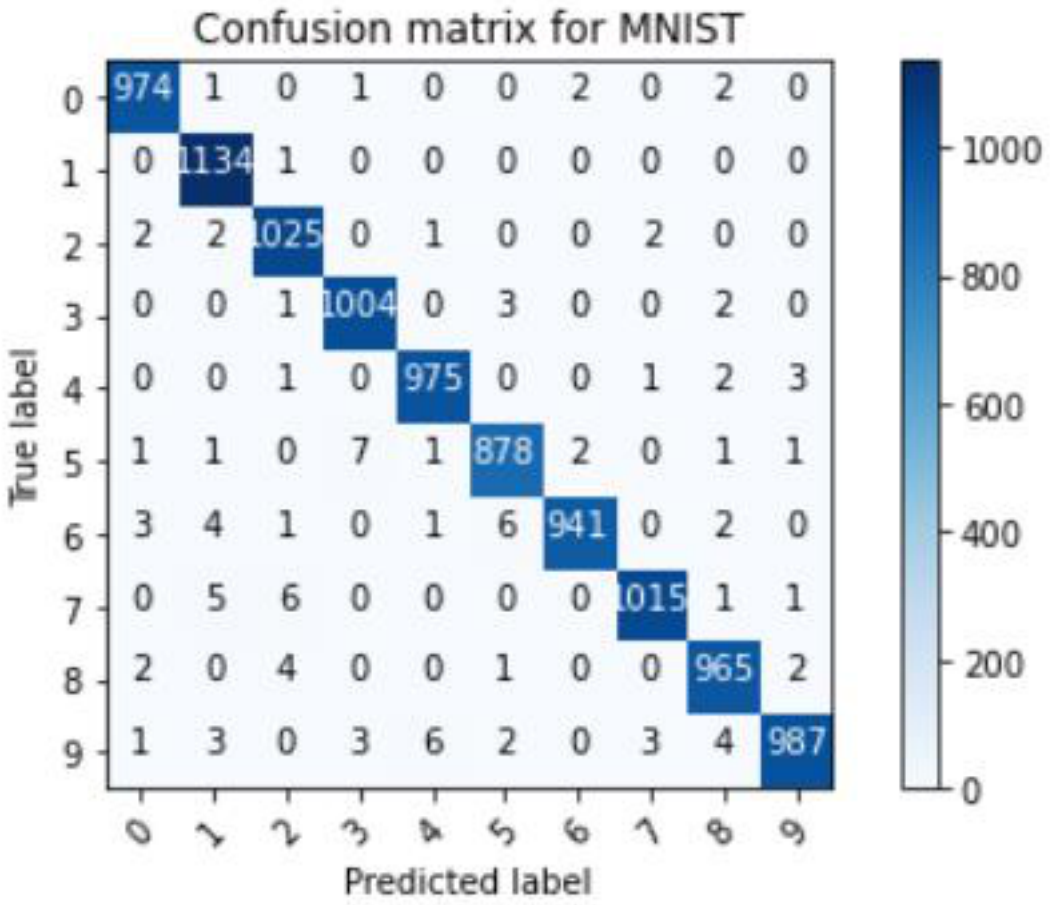


Learning rate = 0.02: Pytorch Model Loss (Left) Model Accuracy (Right)

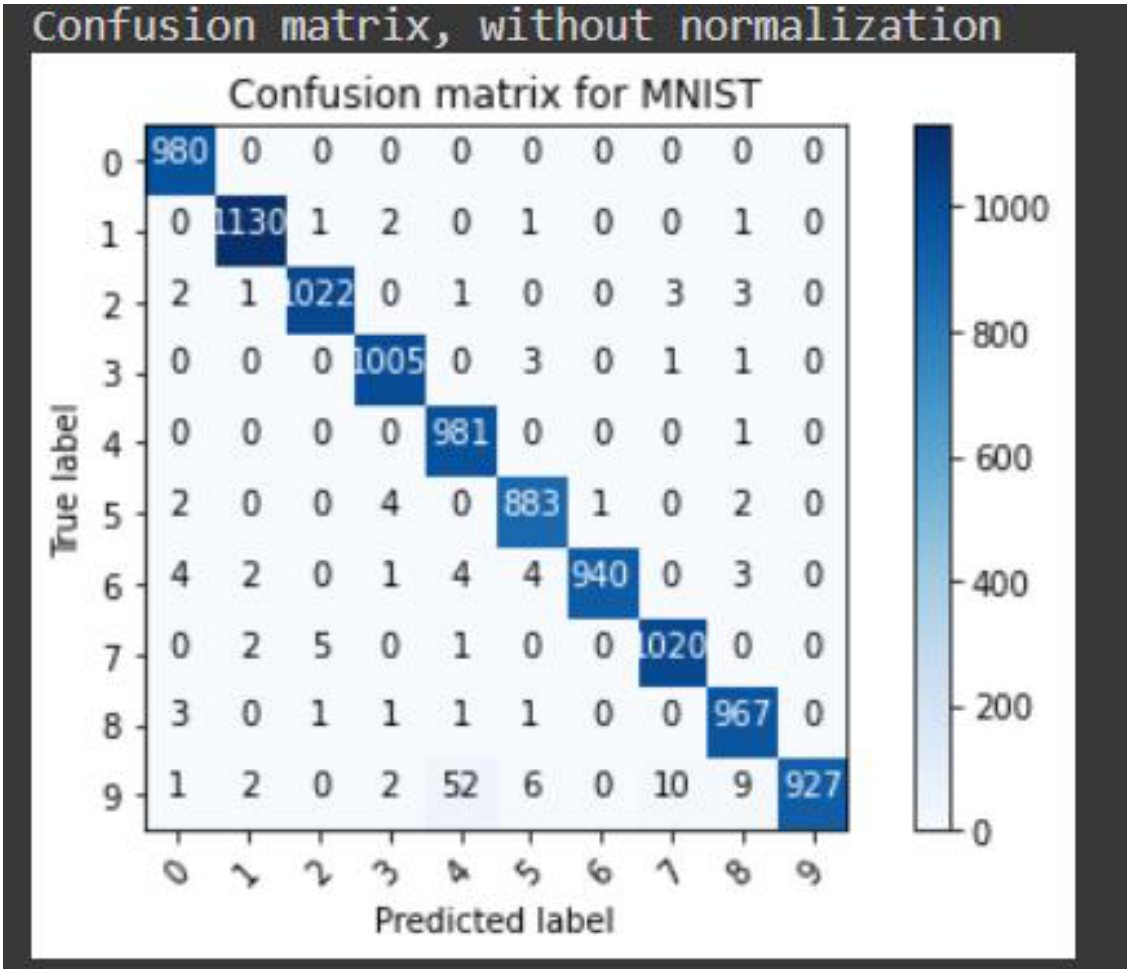


Learning rate = 0.2: Pytorch Model Loss (left) Model Accuracy (Right)

[24]:
```
display(Image.open("./cm_lr02.jpg"))
print("comfusion matrix For Pytorch Model on MNIST with lr=0.02")

display(Image.open("./cm_lr2.jpg"))
print("Confusion Matrix For Pytorch Model on MINIS with lr=0.2")
```



comfusion matrix For Pytorch Model on MNIST with lr=0.02



`Confusion Matrix For Pytorch Model on MINIS with lr=0.2`

From the above result of testing, we can see that using learning rate of 0.002 and 0.02 both work pretty well on the training and testing set. We can see that changing the learning rate does not affect the training and testing time that much with GPU accUsing learning rate of 0.2 tends to result in overfitting as we can see from the learning curve plot and it has reduced the accuracy and has a irregular loss. But overall, lr=0.02 would be more optimal. ##### Therefore, from now on, we will use a learning rate of 0.02.

In the next step, I want to test the effect of using different number of neurons.

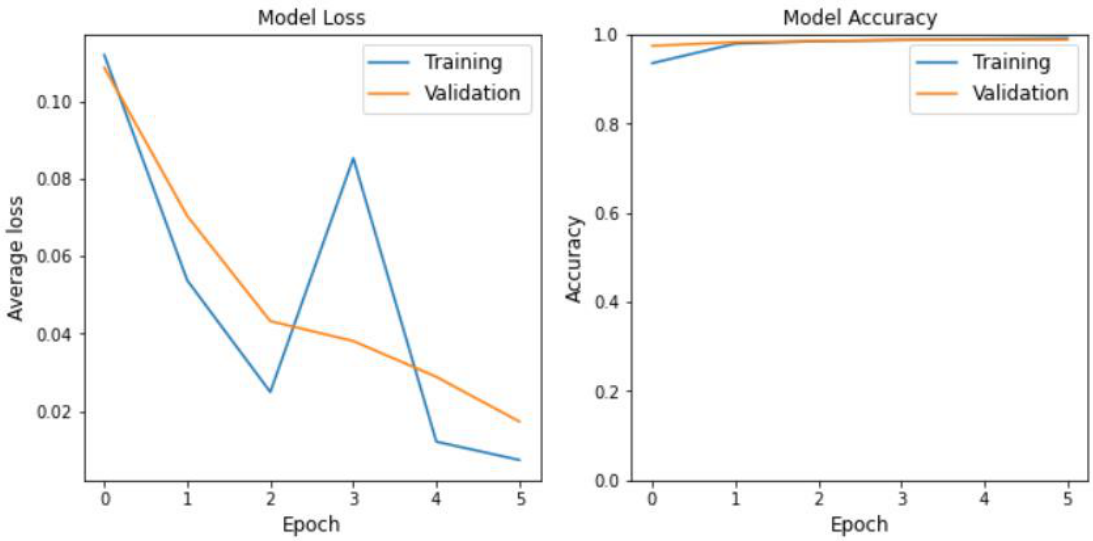
Below are results for 2-layer pytorch with neurons of 1>16>32 Train Avg. Loss: [0.0077] Acc: 0.9908 on 60000.0 images
Validating…
Val Avg. Loss: [0.0093] Acc: 0.991 on 10000.0 images
Time Elapsed: 341.546268 seconds
Test Accuracy of the model on the 10000 test images: 99.1 %
Time Elapsed: 3.847924 seconds

Below are results for 2-layer pytorch with neurons of 1>32>64 Train Avg. Loss: [0.0223] Acc: 0.9918 on 60000.0 images
Validating…
Val Avg. Loss: [0.0127] Acc: 0.9891 on 10000.0 images
Time Elapsed: 533.516725 seconds
Test Accuracy of the model on the 10000 test images: 99.04 %
Time Elapsed: 4.386131 seconds

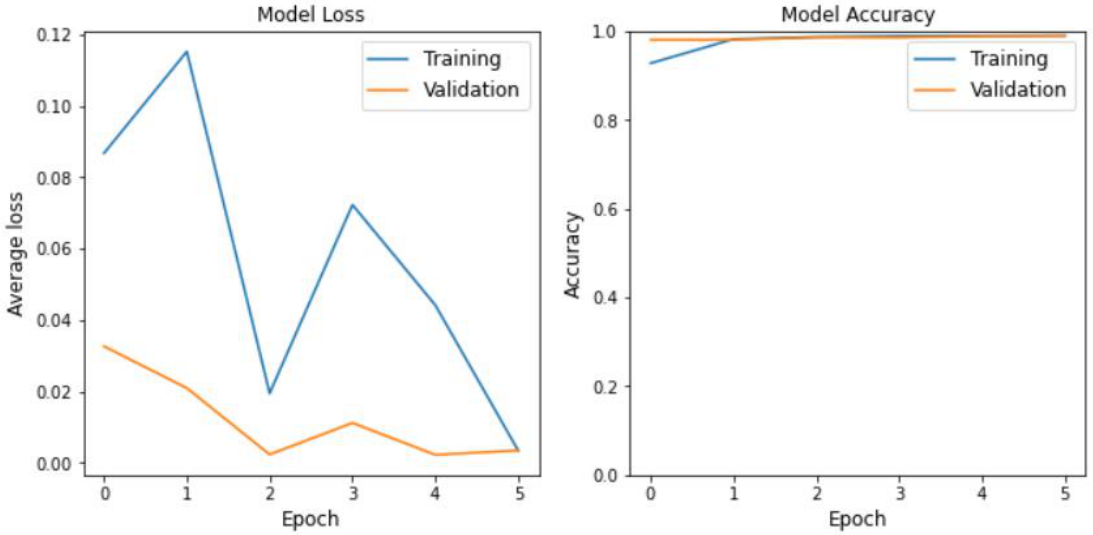
Below are results for 2-layer pytorch with neurons of 1>48>96 Train Avg. Loss: [0.0035] Acc: 0.9908 on 60000.0 images
Validating…
Val Avg. Loss: [0.0035] Acc: 0.9899 on 10000.0 images
Time Elapsed: 672.369639 seconds
Test Accuracy of the model on the 10000 test images: 98.51 %
Time Elapsed: 4.898453 seconds

Below are results for 2-layer pytorch with neurons of 1>64>128 Train Avg. Loss: [0.0087] Acc: 0.9864 on 60000.0 images
Validating…
Val Avg. Loss: [0.0148] Acc: 0.984 on 10000.0 images
Time Elapsed: 816.105477 seconds
Test Accuracy of the model on the 10000 test images: 97.98 %
Time Elapsed: 5.115823 seconds

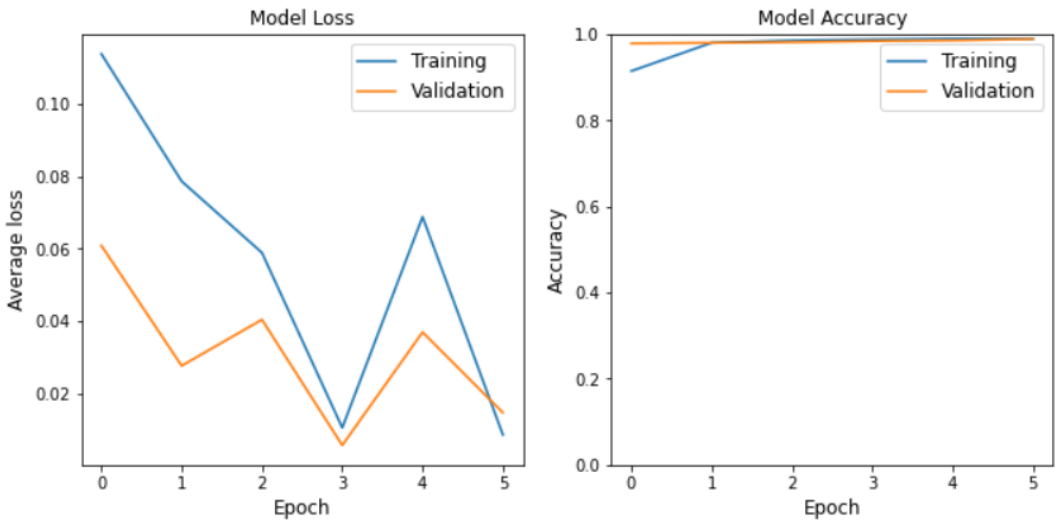
[16]:
```
display(Image.open("./plot_nr32.jpg"))
print("Neuron 1>32>64: Pytorch Model Loss (Left) Model Accuracy (Right)")

display(Image.open("./plot_nr48.jpg"))
print("Neuron 1>48>96: Pytorch Model Loss (left) Model Accuracy (Right)")

display(Image.open("./plot_nr64.png"))
print("Neuron 1>64>128: Pytorch Model Loss (left) Model Accuracy (Right)")
```



Neuron 1>32>64: Pytorch Model Loss (Left) Model Accuracy (Right)

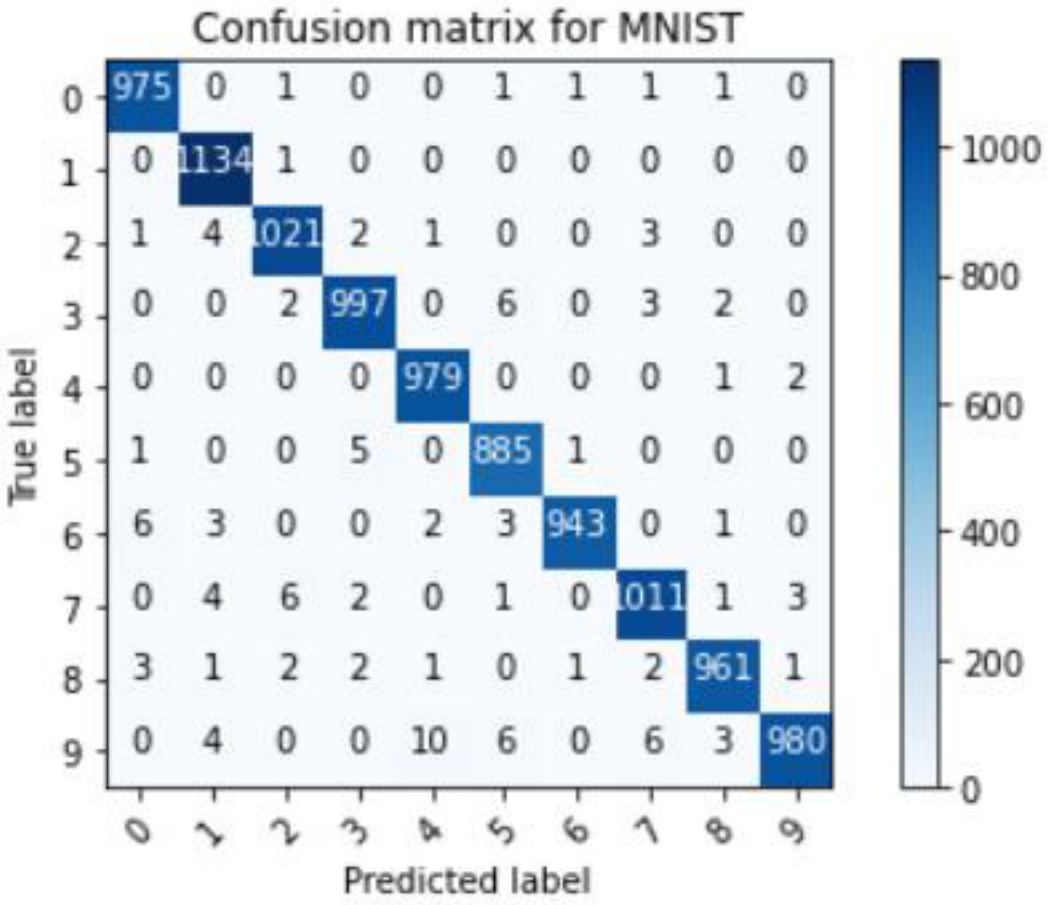


Neuron 1>48>96: Pytorch Model Loss (left) Model Accuracy (Right)

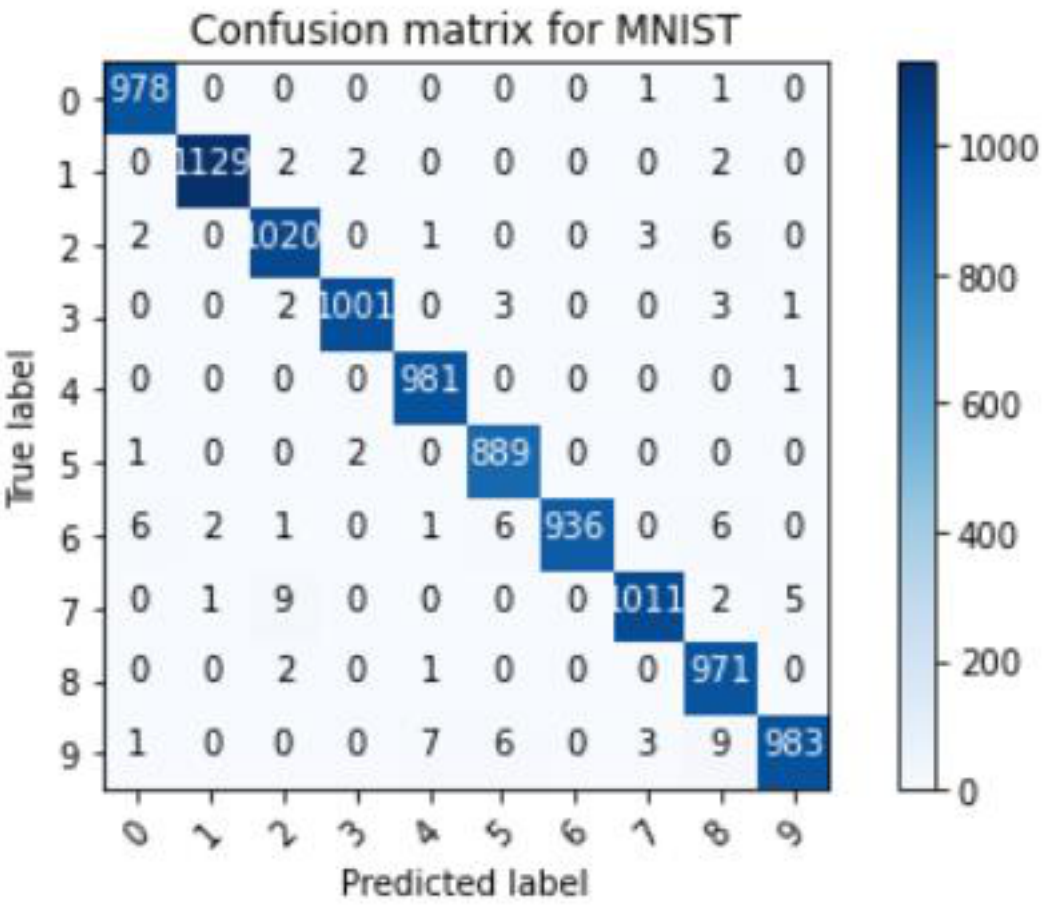


Neuron 1>64>128: Pytorch Model Loss (left) Model Accuracy (Right)

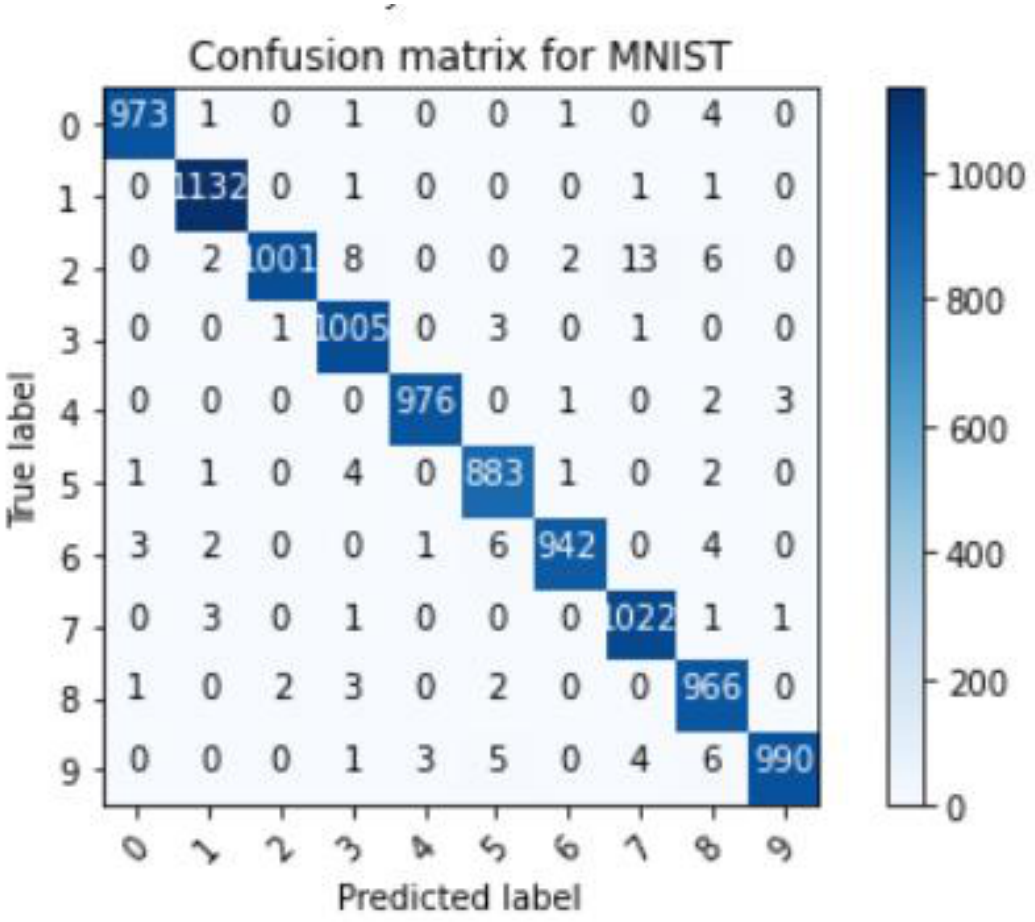
[17]:
```
display(Image.open("./cm_nr32.jpg"))
print("comfusion matrix For Pytorch Model on MNIST with Neuron 1>32>64")

display(Image.open("./cm_nr48.jpg"))
print("Confusion Matrix For Pytorch Model on MINIS with Neuron 1>48>96")

display(Image.open("./cm_nr64.jpg"))
print("Confusion Matrix For Pytorch Model on MINIS with Neuron 1>64>128")
```



comfusion matrix For Pytorch Model on MNIST with Neuron 1>32>64



Confusion Matrix For Pytorch Model on MINIS with Neuron 1>48>96



`Confusion Matrix For Pytorch Model on MINIS with Neuron 1>64>128`

From the results above, we can see that increasing the number of neurons in each layer would increase the training time. The accuracy seems pretty nice as results of 1>16>32, 1>32>64, and 1>48>96 whereas increasing the neurons to 1>64>128 showed a little decrease in accuracy because of overfitting shown by the learning curve plot, which shows that the training accuracy is higher than the testing accuracy and the training loss is higher than the validation loss. However, it is also worth noticing that the testing accuracy has been improved from 1>16>32 to 1>32>64, and then dropped again after increasing the neurons in the first layer to 48.

Therefore, I decided to use a 1>32>64 model.

Then I tried to add dropouts and softmax layers as instructed in class.

The results are shown below: ##### dropout = 0.02: Train Avg. Loss: [1.4855] Acc: 0.9888 on 60000.0 images
Validating…
Val Avg. Loss: [1.4721] Acc: 0.9887 on 10000.0 images
Time Elapsed: 52.806295 seconds
Test Accuracy of the model on the 10000 test images: 98.87 %

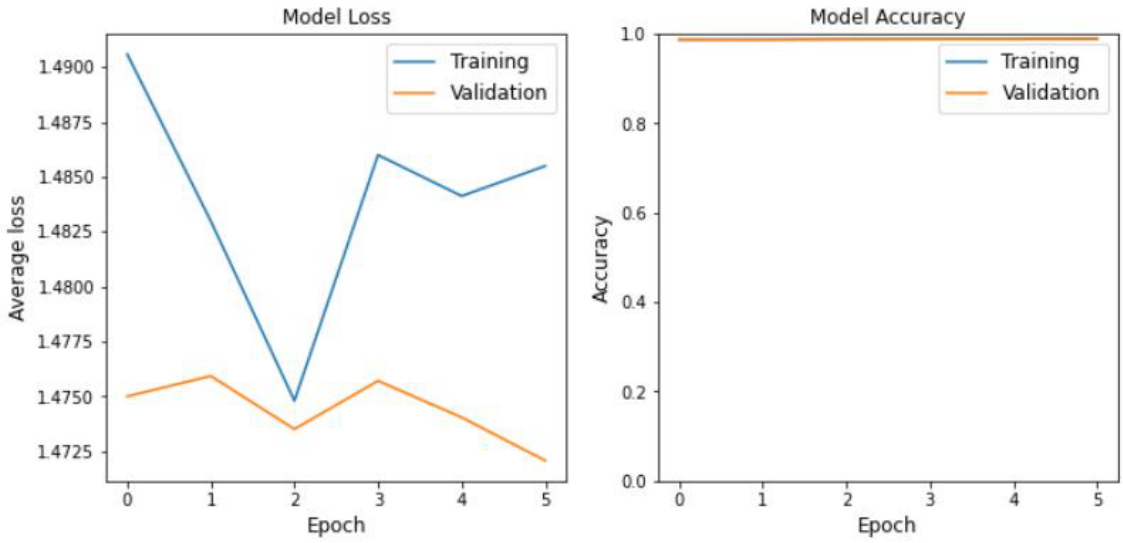
Time Elapsed: 1.512871 seconds

dropout =0.05: Train Avg. Loss: [1.4905] Acc: 0.9758 on 60000.0 images
Validating…
Val Avg. Loss: [1.4762] Acc: 0.9868 on 10000.0 images
Time Elapsed: 53.149858 seconds
Test Accuracy of the model on the 10000 test images: 98.68 %
Time Elapsed: 0.956252 seconds

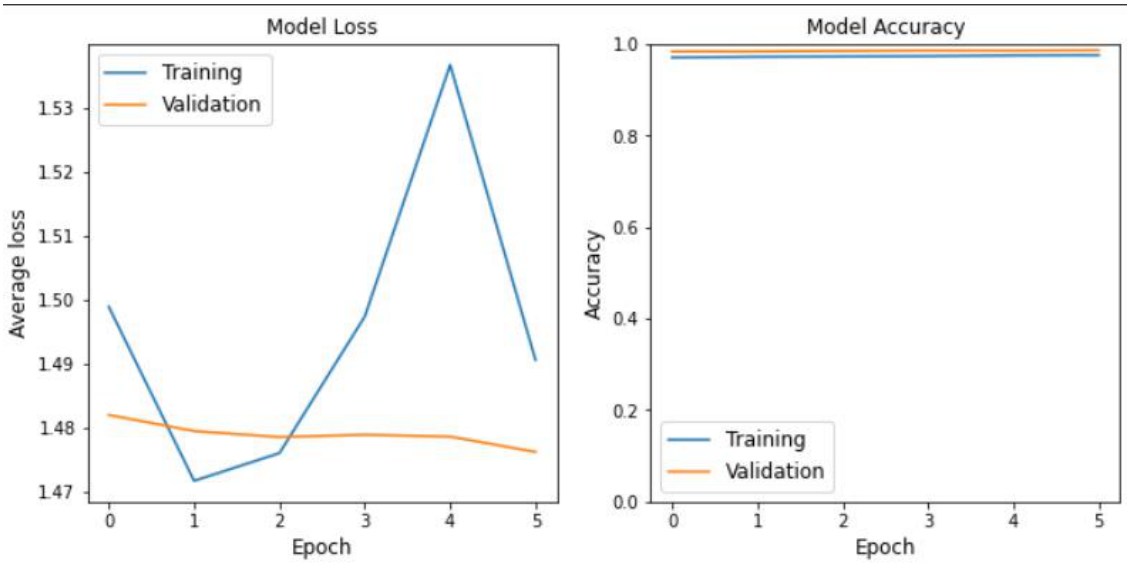
dropout = 0.15: Train Avg. Loss: [1.5675] Acc: 0.9099 on 60000.0 images
Validating…
Val Avg. Loss: [1.4824] Acc: 0.9822 on 10000.0 images
Time Elapsed: 53.777991 seconds
Test Accuracy of the model on the 10000 test images: 98.22 %
Time Elapsed: 0.975473 seconds

dropout=0.3 Train Avg. Loss: [1.6079] Acc: 0.7768 on 60000.0 images
Validating…
Val Avg. Loss: [1.4849] Acc: 0.9805 on 10000.0 images
Time Elapsed: 52.552638 seconds
Test Accuracy of the model on the 10000 test images: 98.05 %
Time Elapsed: 0.981333 seconds

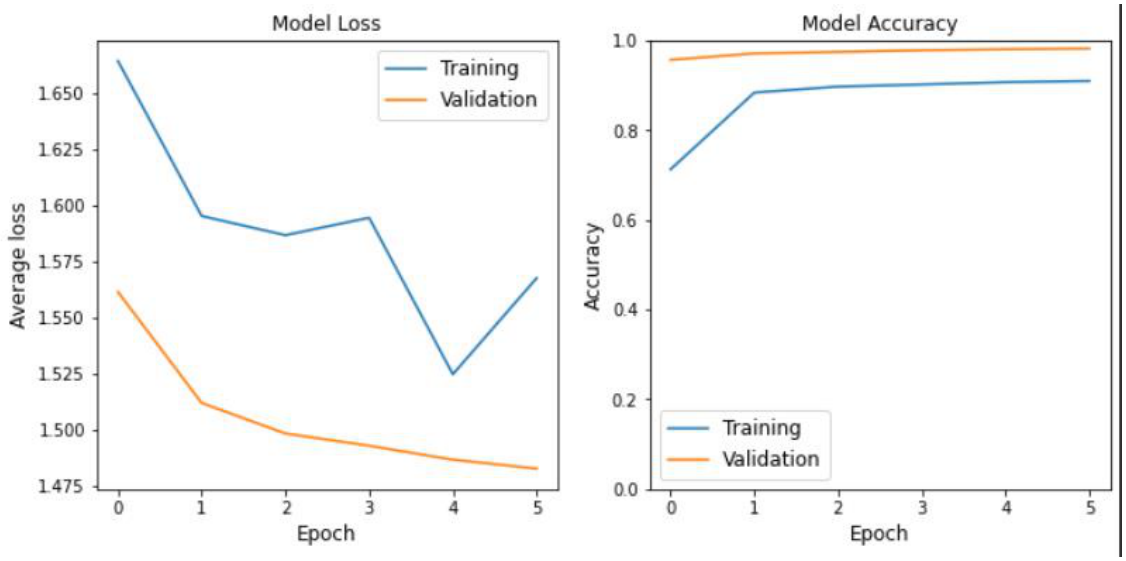
[44]:
```
display(Image.open("./plot_dr02.jpg"))
print("Pytorch Model Loss (Left) Model Accuracy (Right) with dropout=0.15\n")
display(Image.open("./plot_dr05.jpg"))
print("Pytorch Model Loss (Left) Model Accuracy (Right) with dropout=0.05\n")
display(Image.open("./plot_dr25.jpg"))
print("Pytorch Model Loss (Left) Model Accuracy (Right) with dropout=0.15\n")
display(Image.open("./plot_dr30.png"))
print("Pytorch Model Loss (Left) Model Accuracy (Right) with dropout=0.30\n")

display(Image.open("./cm_dr02.jpg"))
print("Confusion Matrix For Pytorch Model with dropout=0.02")
display(Image.open("./cm_dr05.jpg"))
print("Confusion Matrix For Pytorch Model with dropout=0.05")
display(Image.open("./cm_dr25.jpg"))
print("Confusion Matrix For Pytorch Model with dropout=0.15")
display(Image.open("./cm_dr30.jpg"))
print("Confusion Matrix For Pytorch Model with dropout=0.30")
```



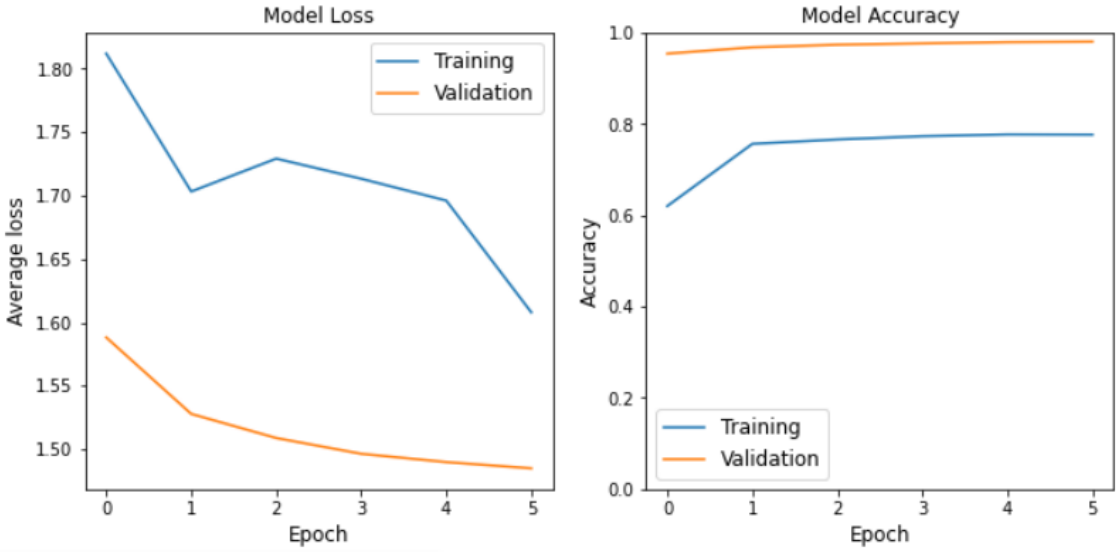
Pytorch Model Loss (Left) Model Accuracy (Right) with dropout=0.15



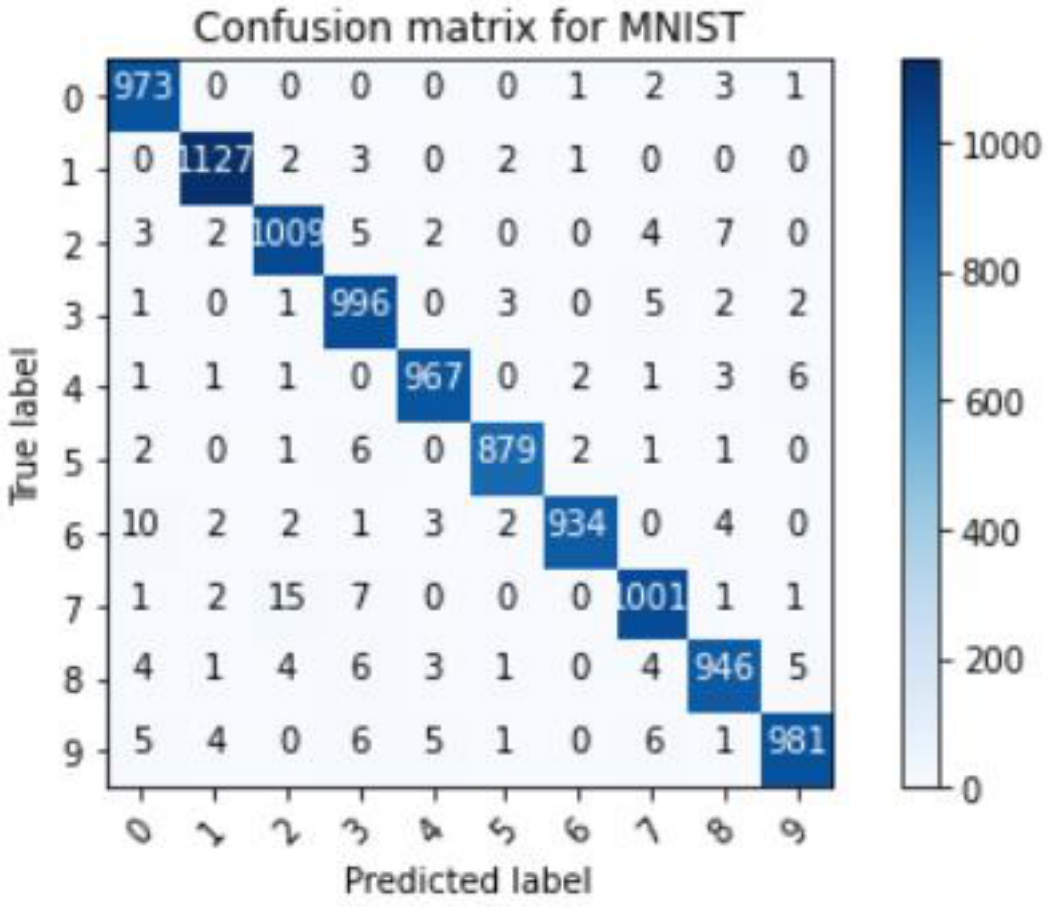
Pytorch Model Loss (Left) Model Accuracy (Right) with dropout=0.05



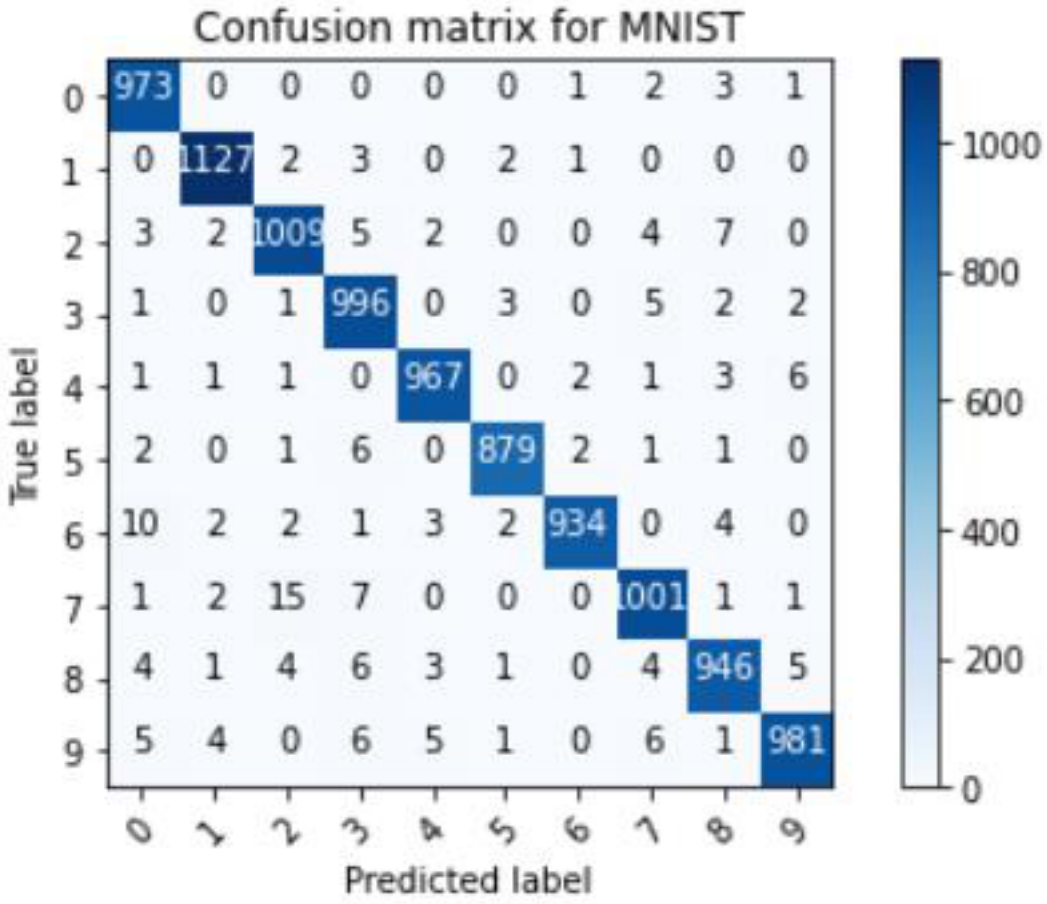
Pytorch Model Loss (Left) Model Accuracy (Right) with dropout=0.15



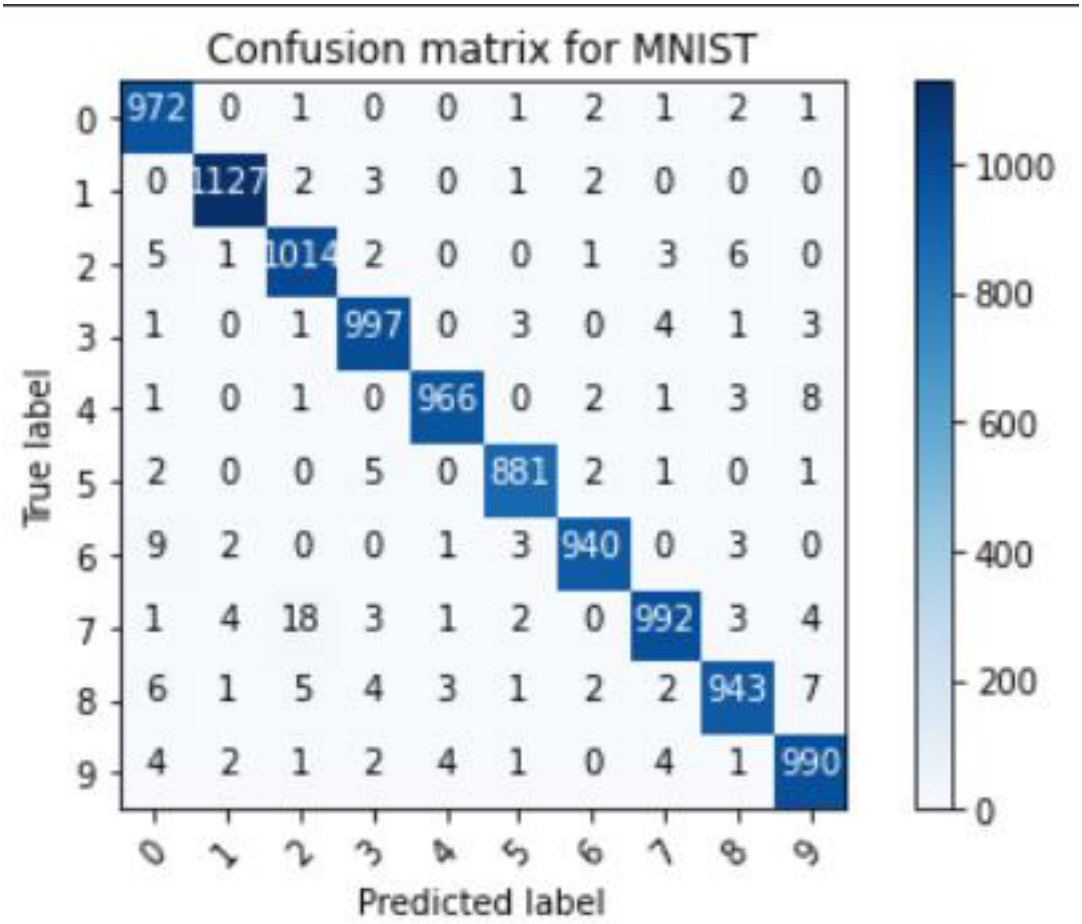
Pytorch Model Loss (Left) Model Accuracy (Right) with dropout=0.30



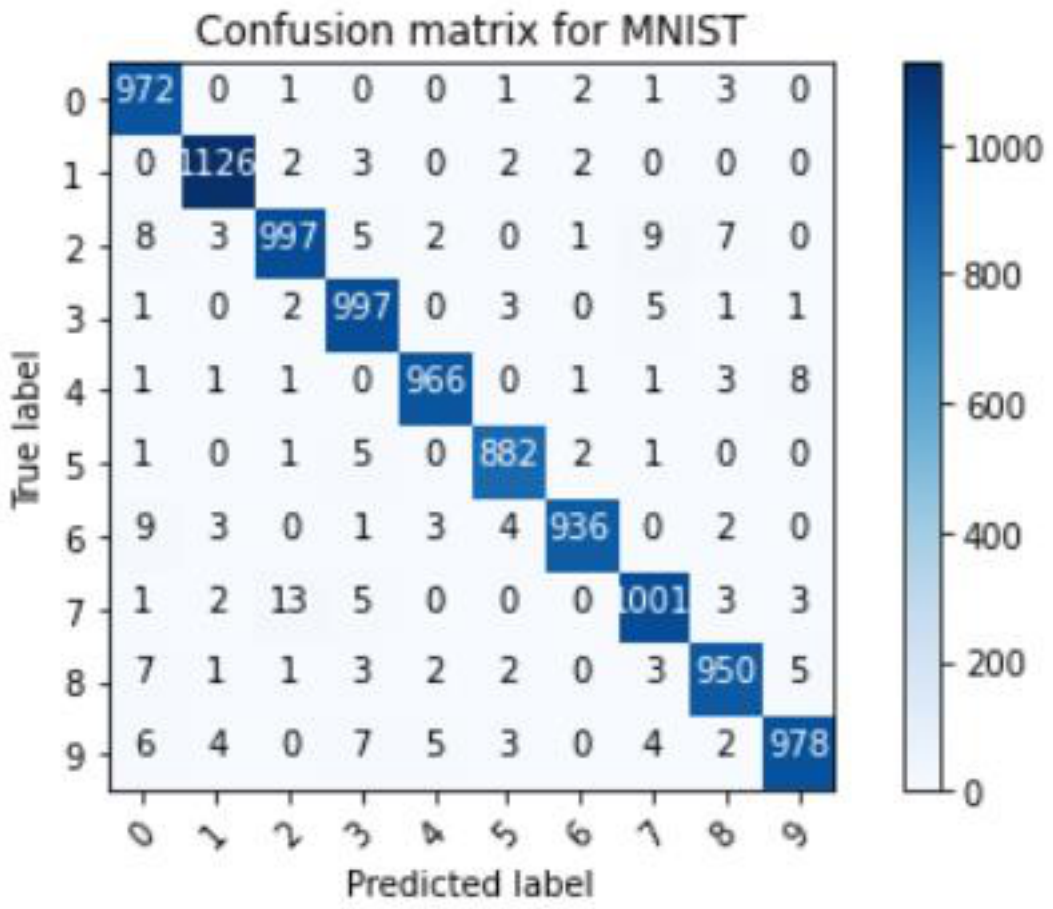
Confusion Matrix For Pytorch Model with dropout=0.02



Confusion Matrix For Pytorch Model with dropout=0.05



Confusion Matrix For Pytorch Model with dropout=0.15



`Confusion Matrix For Pytorch Model with dropout=0.30`

From the above results, we can see that adding dropout layer and softmax layer will highly reduce the training and testing time. The training time is less than 10% of before and the testing time is less than 20% of before. This is lower dropout parameter can preserve the accuracy while not increasing the training/testing time. From the learning curve plot, we can also see that non of the model suffers from overfitting. Therefore, I finally decided to employ the dropout=0.02 to best preserve the accuracy while the highly reducing the computational cost for the best performance.

1.2.1 Changes made for performance improvement

1. Change learning rate to 0.02

2. change the number of neurons in the layer to 1>32>64

3. add a dropout layer = 0.02 and add an 1 dimension and softmax layer The code is shown below:

```
[ ]: # Model
# Convolutional neural network (two convolutional layers)
class ConvNet(nn.Module):
    def __init__(self, num_classes=10):
        super(ConvNet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
            ) # The channel output size is changed from 16 to 32
        self.layer2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.fc1 = nn.Sequential(
            nn.Linear(7*7*64, 256),
            nn.ReLU()) # Changed the output into 256 instead of number of␣
 ↪classes, becase we wil use 2 dense layers
        self.fc2 = nn.Linear(256, num_classes) # Second dense layer added,␣
 ↪taking in 256 as inputs and outputing the number of classes 10.
        self.drop_out = nn.Dropout(p=0.3) # Created a dropout layer to prevent␣
 ↪overfitting
        self.flat=nn.Flatten()
        self.soft_max = nn.Softmax(dim=1) # Created a softmax layer

    def forward(self, x): # The final structure of the CNN model
        out = self.layer1(x)
        out = self.layer2(out)

        out = out.reshape(out.size(0), -1)
        out = self.fc1(out)
        out = self.fc2(out)
        out = self.drop_out(out)
        out = self.flat(out)
        out = self.soft_max(out)
        return out

model = ConvNet(num_classes).to(device)
```

Since the original model is already very accurate with 98%-99% accuracy, there is not much to improve in the accuracy. The main improvement I made to the model is to reduce the computation time while preserving the accuracy.

1.3 3. Results on own Data

In this section, I used the original CNN pytorch model (P2) to test the thickened hand written dataset that I created in Lab 7. The data set is under folder idata2. I changed the section that obtain data following the pytorch tutorial that instructs us to using the SimpleDataset to get our own data and convert them to tensor for more efficient training. An interesting thing I noticed during debug process was that we need to change the special characters (51-60th character that are not numbers) to the label class 10 and change the number of classes accordingly. Then we can follow the training as we usually do. To plot the confusion matrix, we need to make subtle changes since it's not a square matrix anymore, and we change the number of classes in the model and the hyperparameter accordingly. The results are shown below.

EPOCH [1/6]. Running Loss: 188.04 Progress: 33.33 %
EPOCH [1/6]. Running Loss: 245.08 Progress: 66.67 %
EPOCH [1/6]. Running Loss: 282.32 Progress: 100.0 %
Train Avg. Loss: [0.1247] Acc: 0.8945 on 60000.0 images
Validating…
Val Avg. Loss: [1.6798] Acc: 0.6833333333333333 on 60.0 images

EPOCH [2/6]. Running Loss: 27.38 Progress: 33.33 %
EPOCH [2/6]. Running Loss: 51.54 Progress: 66.67 %
EPOCH [2/6]. Running Loss: 71.41 Progress: 100.0 %
Train Avg. Loss: [0.1652] Acc: 0.9693 on 60000.0 images
Validating…
Val Avg. Loss: [1.7102] Acc: 0.75 on 60.0 images

EPOCH [3/6]. Running Loss: 17.54 Progress: 33.33 %
EPOCH [3/6]. Running Loss: 33.47 Progress: 66.67 %
EPOCH [3/6]. Running Loss: 48.66 Progress: 100.0 %
Train Avg. Loss: [0.197] Acc: 0.9781 on 60000.0 images
Validating…
Val Avg. Loss: [1.7813] Acc: 0.7833333333333333 on 60.0 images

EPOCH [4/6]. Running Loss: 13.29 Progress: 33.33 %
EPOCH [4/6]. Running Loss: 26.43 Progress: 66.67 %
EPOCH [4/6]. Running Loss: 38.83 Progress: 100.0 %
Train Avg. Loss: [0.0871] Acc: 0.9824 on 60000.0 images
Validating…
Val Avg. Loss: [1.8328] Acc: 0.7833333333333333 on 60.0 images

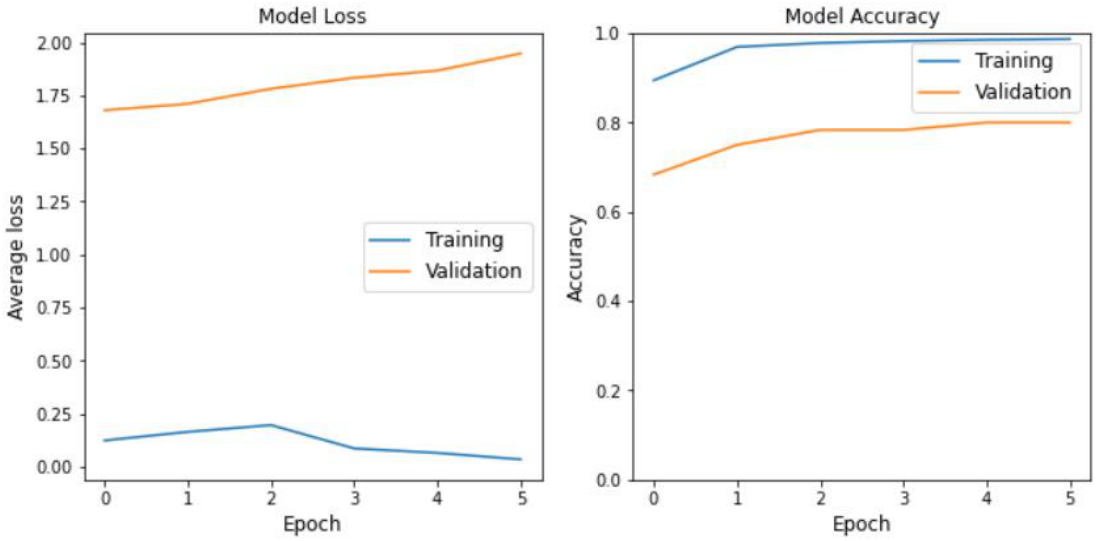
EPOCH [5/6]. Running Loss: 11.19 Progress: 33.33 %
EPOCH [5/6]. Running Loss: 22.54 Progress: 66.67 %
EPOCH [5/6]. Running Loss: 32.6 Progress: 100.0 %
Train Avg. Loss: [0.066] Acc: 0.9852 on 60000.0 images
Validating…
Val Avg. Loss: [1.8668] Acc: 0.8 on 60.0 images

EPOCH [6/6]. Running Loss: 9.67 Progress: 33.33 %
EPOCH [6/6]. Running Loss: 19.02 Progress: 66.67 %
EPOCH [6/6]. Running Loss: 28.66 Progress: 100.0 %
Train Avg. Loss: [0.0355] Acc: 0.987 on 60000.0 images

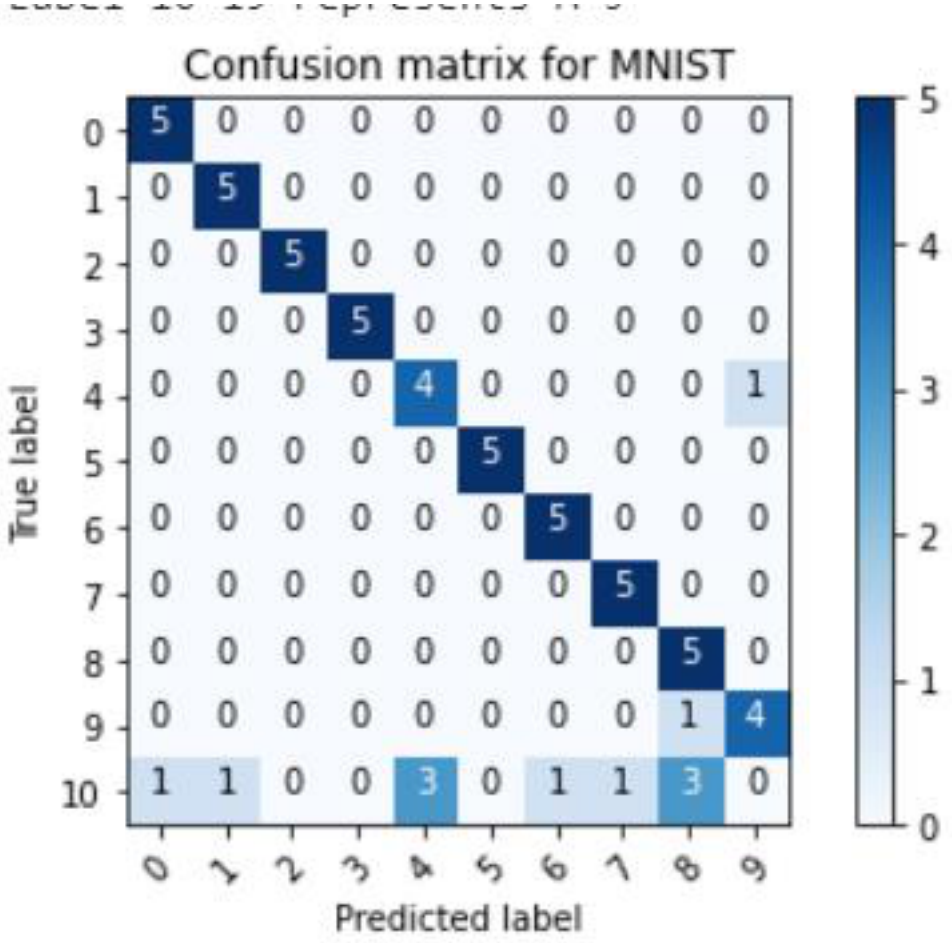
Validating...
Val Avg. Loss: [1.9472] Acc: 0.8 on 60.0 images

Time Elapsed: 1936.860053 seconds
Test Accuracy of the model on the 10000 test images: 80.0 %

[32]:
```
display(Image.open("./plot_self.jpg"))
print("Pytorch Model Loss (Left) Model Accuracy (Right) on iData set")

display(Image.open("./cm_self.jpg"))
print("Confusion Matrix For Pytorch Model on iData set")
```




```
Pytorch Model Loss (Left) Model Accuracy (Right) on iData set
```



`Confusion Matrix For Pytorch Model on iData set`

Train Avg. Loss: [0.0355] Acc: 0.987 on 60000.0 images Validating… Val Avg. Loss: [1.9472] Acc: 0.8 on 60.0 images

Time Elapsed: 1936.860053 seconds Test Accuracy of the model on the 10000 test images: 80.0 % In this part, I did not use GPU acceleration so the training time was 1936 seconds. This is in fact a shorter time compared to what generally take to run pytorch with 6 epoch (on my laptop it generally takes 2200-2800 seconds). From the results above, we can see a loss of 1.9472 and accuracy of 0.8. This accuracy has improved by 35% from lab 7 where we only have accuracy of 0.45. And again, because we have 10 special characters that must be misclassified, the highest accuracy we can get is 83.33% and therefore, the 80% accuracy is extremely high, again proving the high performance of pytorch. From the confusion matrix, we can see this as well since only class 10 and 1 of 9 is misclassified. However, if we look at the learning curve plot, we can see that the model seems to be suffering from overfitting. If we exclude class 10 which is deliberately designed to be misclassified, the accuracy is in fact 98%, which means that this is not underfitting and works fine.

2 4. Summary

2.0.1 Comparison of approaches

Model

Accuracy(%)

Test Loss

TrainingTime

Epochs

TestingTime

TrainingAccuracy(%)

Keras Tutorial

83.88

0.7024

1957

12

ModelK1

82.40

0.7848

1155

12

4.10

70.07

ModelP1

99.04

0.0008

105.96

12

20.39

99.22

ModelK2

78.57

1.4897

872.33

6

5.97

57.34

ModelP2

98.61

0.038

305.76

6

3.43

98.75

Custom Model

Custom data

80.00

1.9472

1936.86

6

252.67

0.987

Note that for section 3, I run the program in jupyter notebook instead of colab so the training and testing time is longer than others.

2.0.2 Discusssion

In this lab, we explored the differences between kera and pytorch models and tested different parameters with pytorch. Here are some findings of mine as a short conclusion: 1. It's actually a surprise to me that kera is so computationally expensive yet has such a low accuracy. 2. Increasing the number of neurons would increase the training time and testing time accordingly but not increasing it in a linear correlation. And adding too many neurons tends to lead to overfitting. 3. Adding dropout layer and softmax layers does not affect the accuracy very much but would highly reduce the training and testing time. 4. Pytorch has really good performance on my own dataset that which had very low accuracy in lab 7 but tends to result in overfitting, true application needs revisement.