GR 5241 Final Project Milestone II Report

Contents

Side Note:

As mentioned in Ed that this report will be focusing on Milestone II

Milestone II: Deep Learning

In this part, I had implemented two different artificial neural network structures (ANN and CNN) with the MNIST training and test sets.

A brief introduction about MNIST dataset: The MNIST database of handwritten digits is one of the most used datasets for training various image processing systems and machine learning algorithms. MNIST has a training set of 60,000 examples, and a test set of 10,000 examples.

3. Train with ANN

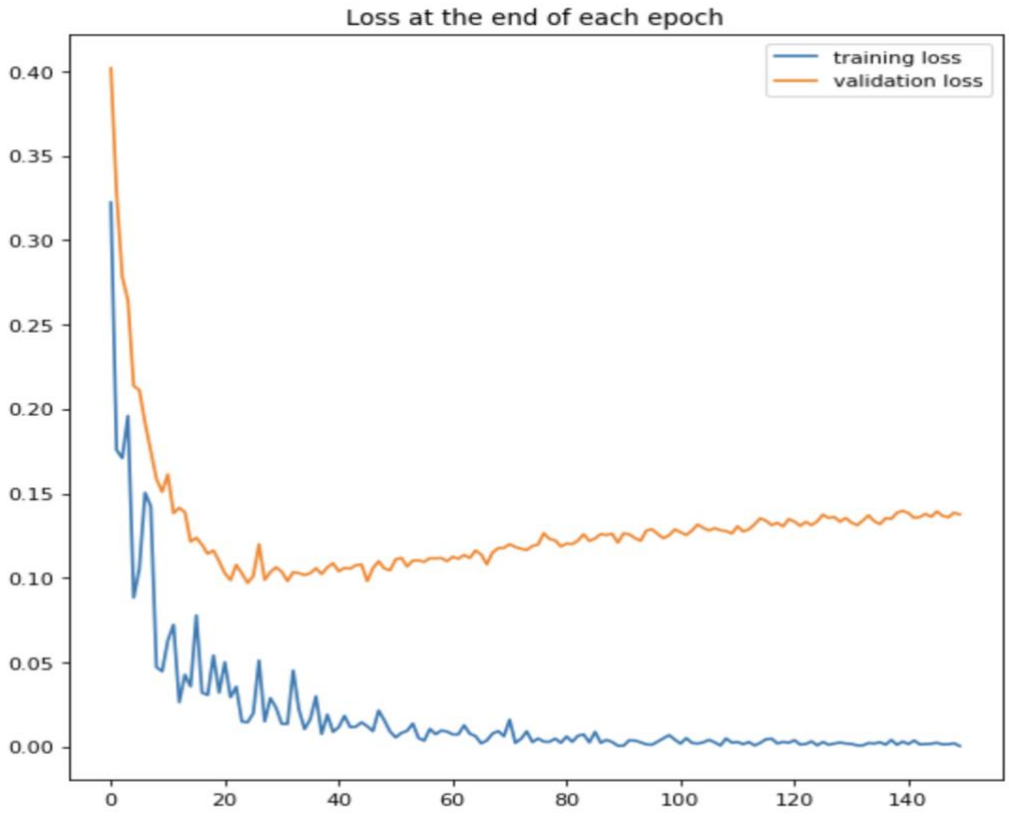
In this part I have constructed my model as required: a single layer neural network with 100 hidden units that is with architecture (input features: 784, hidden layer units: 100 and output features: 10). Below is the screenshot of my model and this model remains valid until **Train with CNN**.

```
class Model(nn.Module):

    def __init__(self, in_features = 784, h1 = 100, out_features = 10):
        # one single layer neural network
        super().__init__()
        self.fc1 = nn.Linear(in_features,h1)
        self.out = nn.Linear(h1,out_features)

    def forward(self,x):

        x = F.relu(self.fc1(x))
        x = self.out(x)

        return F.log_softmax(x, dim=1)
```

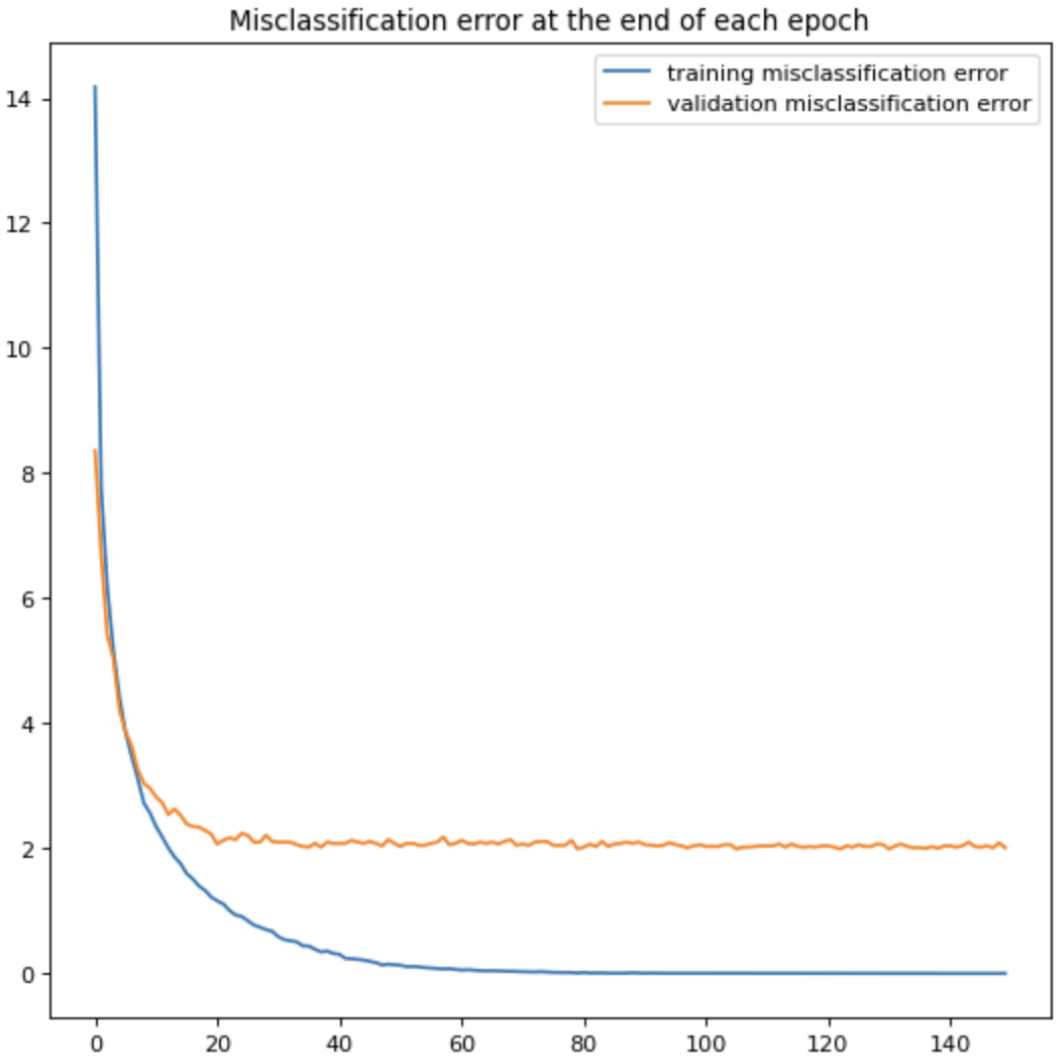
Firstly, for the default setup, I choose SGD as my optimizer with learning rate = 0.1 as our baseline model. Then I have trained this model 6 times with 150 epochs each time, and based on the results for the validation curve, the misclassification error converges during the first

training at the 35th epochs with a peak test accuracy 98.03% and so is the loss error reaching the lowest point at the 35th epochs with a lowest value around 0.10 and starts to increase after that point. As the training process keeps going, we notice that for the validation curve the loss error keeps increasing until it converges to around 0.14 in the 5th training and the misclassification error tends to be stable during the rest trainings. Unsurprisingly, for the training curve, the loss error keeps having the decreasing trends and the misclassification error goes to 0 around the 80th epochs. Since here we are using the SGD optimizer, the training curve for loss error has fluctuations instead of having the smooth curve. Below are the loss error / misclassification error plots for the first training:

(a) Cross-Entropy Error Plot



(b) Misclassification Error Plot

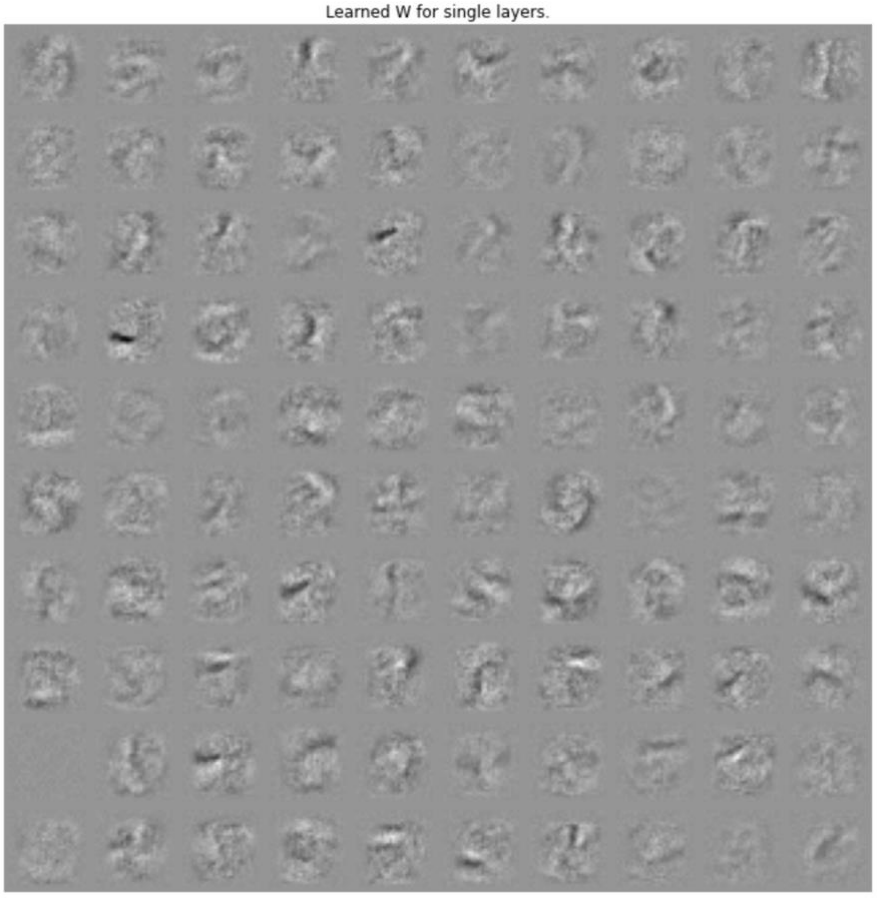


By comparing to those two different types of plots we can find out that for plots in (a), the validation loss starts to increase after a point. In my model at learning rate 0.1, it starts the increasing trend in the middle of first training at the 35th epoch and loss gradually getting more and more during the rest training.

For the plots in (b), we will find out that the misclassification error (in percentage) eventually converged to a specific percentage, in this model it's around 2%, and it does not change much as we keep repeating more trainings.

As a result, this model obtains its best prediction in the first training. Since the ANN model reaches its best performance at the 35th epochs with test accuracy of 98.03%, the below visualization of my best results of the learned weight as one hundred 28 by28 images will be the result from the first training.

(c) Best Result Visualization



The ANN takes flattened tabular inputs and therefore loses the information stored in the spatial features. Based on the plot above, it is hard to recognize any patterns or structures by human eyes.

(d) Different Hyperparameters

Concerned with the computational cost, I did not train the rest model for more than 5 times as it is a result based on the findings from part a and b, the ANN model is able to converge during the first training.

In this section I have tried 6 different hyperparameters: learning rate of 0.01, 0.2 and 0.5; momentum of 0.0, 0.5 and 0.9. As a conclusion, we can find out that as the learning rate increases, both of our errors (loss, misclassification) will have a faster rate to converge. For example, at learning rate 0.01, it takes more than 150 epochs for our model to reach the convergence and yet for learning rate 0.2 and 0.5, our model takes about 10 epochs and 4 epochs to be able to converge. After the convergent point, loss error starts to increase, and misclassification error remains stable. For the momentum, we can find out it plays the same role as learning rate does, that is the increment of momentum brings a faster rate for our model to touch the converge point.

4. Train with CNN

In this part I have constructed my model as required: one 2-D convolutional layer, Relu activation function and Maxpooling with appropriate hyperparameters, below is the screenshot of my CNN model.

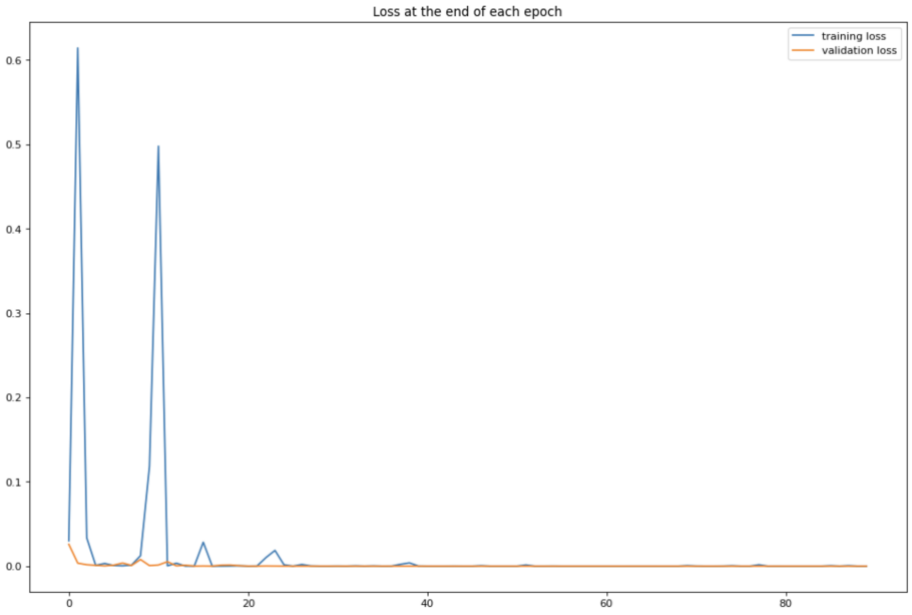
```python
class ConvolutionalNetwork(nn.Module):
    def __init__(self):
        # one 2-D convolutional layers
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, 3, 1)
        self.out = nn.Linear(13*13*16,10)

    def forward(self, X):
        # one 2-D convolutional layers -> Relu activation -> Maxpooling
        X = F.relu(self.conv1(X))
        X = F.max_pool2d(X, 2, 2)
        X = X.view(-1,13*13*16)
        X = self.out(X)
        return F.log_softmax(X, dim=1)
```

The CNN model performs better than my previous ANN model and yet it is more computationally expensive hence I only trained CNN model once and with 90 epochs per time. For the default setup, I choose SGD as my optimizer with learning rate = 0.10 as my baseline model.

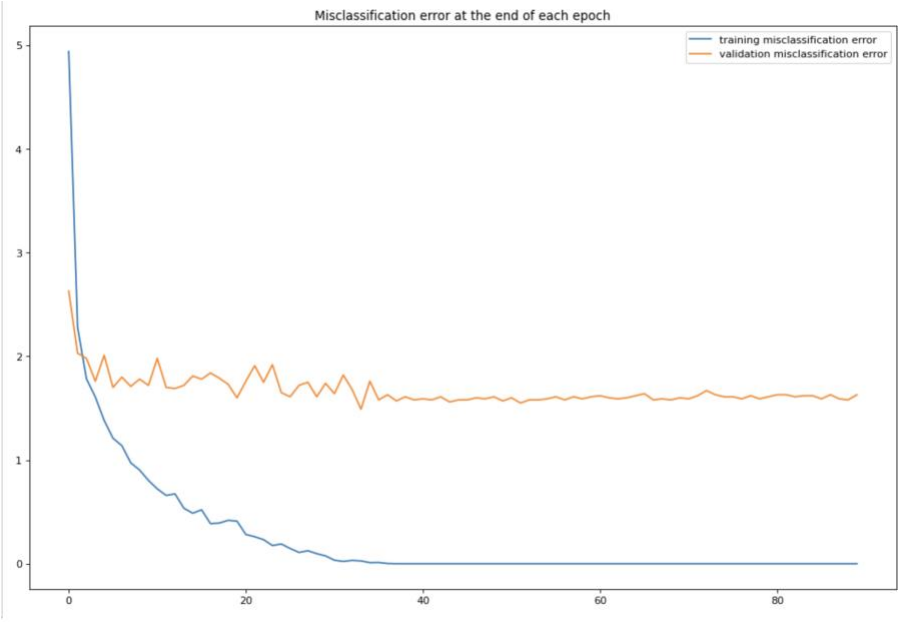
ANN uses weights and an activation function for the bulk of its method as it reconstructs how a brain's neural network works. Comparatively, there is no neuron or weights in CNN. CNN instead casts multiple layers on images and uses filtration to analyze image inputs.

By comparing to plots from ANN model, CNN model takes less epochs to converge and it has a better performance than ANN's first training. From the plots, CNN seems to converge in the very beginning of the training process that it for the training set it converges around the 20^{th} epochs and for the testing set it converges around the 10^{th} epochs. Also, as a comparison to the ANN model: the misclassification error for the first training of ANN is slight above 2% and yet the misclassification error for the first training of CNN is slight below 2%.

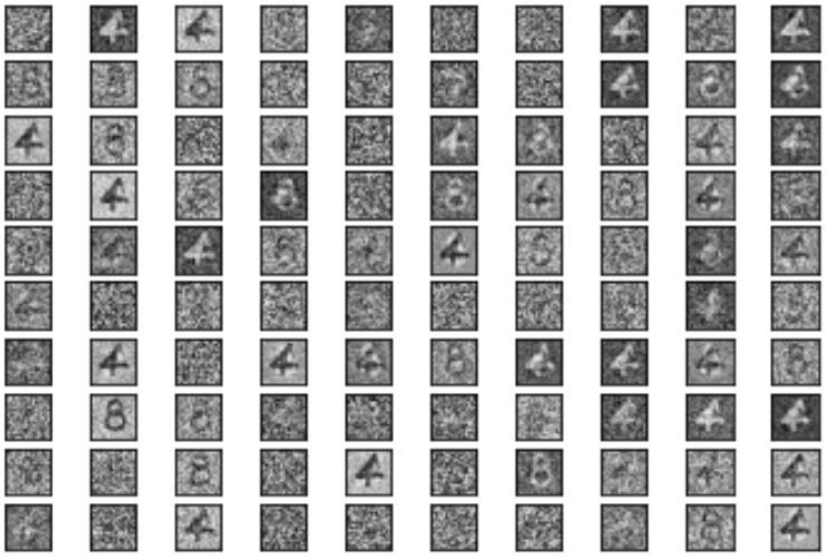
(a) Cross-Entropy Error Plot



(b) Misclassification Error Plot



(c) Best Result Visualization



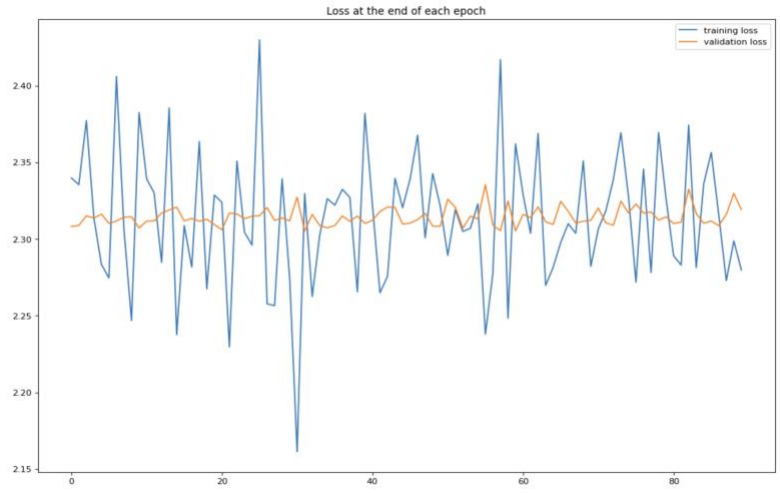
When using CNN, these spatial features are extracted from image input. This makes the CNN model ideal when thousands of features need to be extracted. Instead of having to measure each individual feature, and so CNN gathers these features on its own. Based on the plot above, we can find out that CNN exhibits learned features with some structures.

(d) Different Hyper-parameters

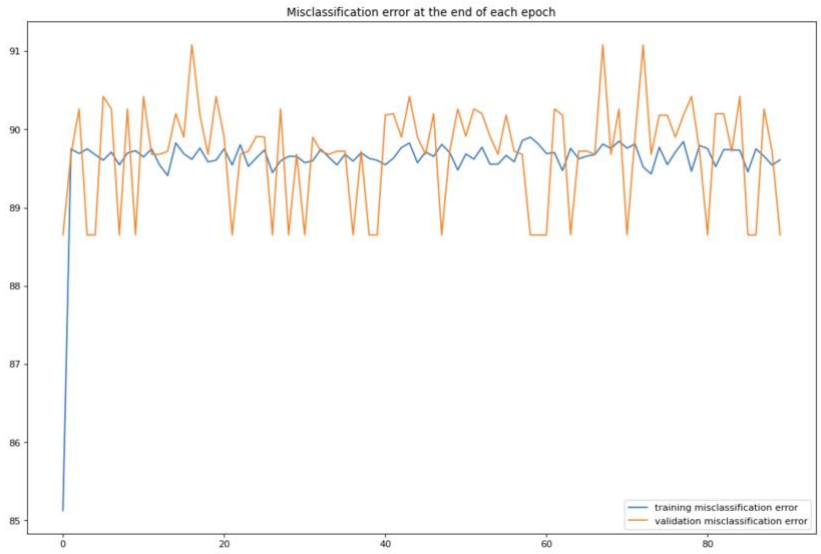
In this section I have also tried 6 different hyperparameters: learning rate of 0.01, 0.2 and 0.5; momentum of 0.0, 0.5 and 0.9. There are two differences between CNN and ANN towards parameters tuning. First, CNN is able to converge at learning rate of 0.01 while ANN could not, and CNN touches the convergent point around 30^{th} epochs. On top of that, CNN shows the same trend as ANN does within a certain extent and yet its performance drops drastically when the learning rate or the momentum is bigger than a threshold. That is, we find out as the learning rate / momentum increases, both of our errors (loss, misclassification) will be unable to reach the

convergence and those two errors fluctuate around some values. For example, at learning rate 0.5, the validation loss error is bouncing around 2.3 and training loss error is bouncing between 2.15 and 2.45. And it shows any sign to converge even till the last epochs. Besides, the misclassification error is high for both training and validation curve that training curve is bouncing around 90% and validation is bouncing between 88% to 91%. Also, it shows no sign to converge. Below are the error plots at learning rate of 0.5 for CNN model:

Loss error (learning rate 0.5)

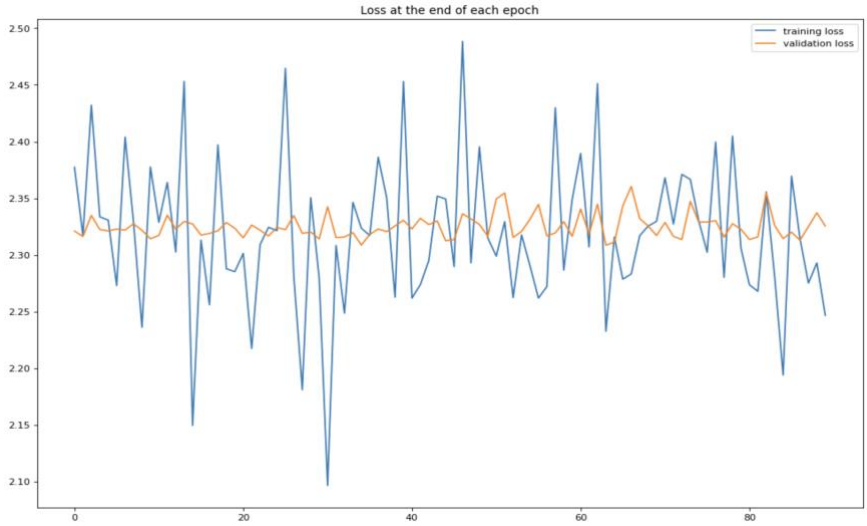


Misclassification error (learning rate 0.5)

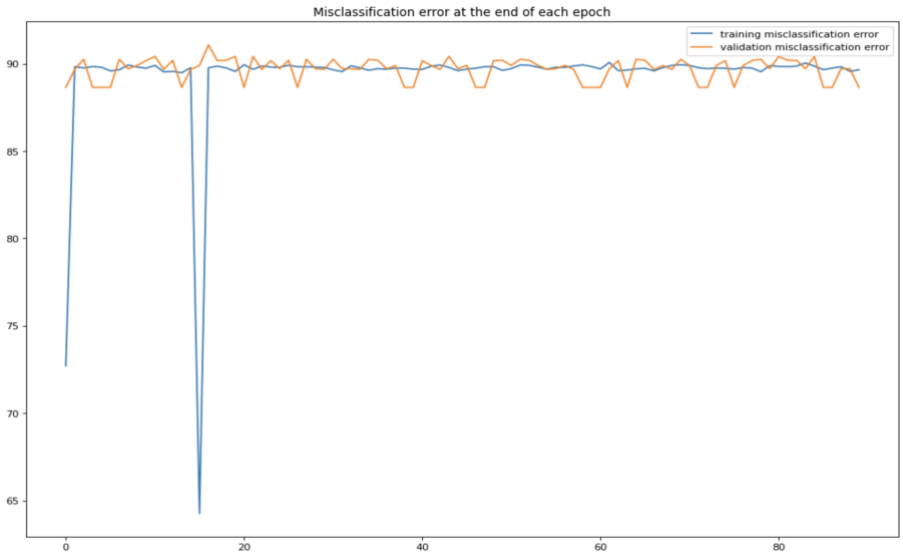


The same phenomenon shows in the momentum too that for the momentum of 0.9 the validation loss error is bouncing around 2.33 and training loss error is bouncing between 2.15 and 2.50. And it shows any sign to converge even till the last epochs. Besides, the misclassification error is high for both training and validation curve that both curves are bouncing around 90%. Below are the error plots at momentum of 0.9 for CNN model:

Loss error (momentum: 0.9)



Misclassification error (momentum: 0.9)



5. Favorite Deep Learning Architecture

The main goal in this step is to beat the performance of SVM with Gaussian Kernel, that is to have a test error rate lower than 1.4%. Here, I construct two CNN models with different output channels towards convolutional layers and each of them have two convolutional layers, two pulling layers and two fully connected layers.

The first one:

```
## two 2-D convolutional layers,two fully connected hidden layers

class ConvolutionalNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, 3, 1)
        self.conv2 = nn.Conv2d(16, 25, 3, 1)
        self.fc1 = nn.Linear(5*5*25, 200)
        self.fc2 = nn.Linear(200, 100)
        self.out = nn.Linear(100,10)

    def forward(self, X):
        X = F.relu(self.conv1(X))
        X = F.max_pool2d(X, 2, 2)
        X = F.relu(self.conv2(X))
        X = F.max_pool2d(X, 2, 2)
        X = X.view(-1, 5*5*25)
        X = F.relu(self.fc1(X))
        X = F.relu(self.fc2(X))
        X = self.out(X)
        return F.log_softmax(X, dim=1)
```

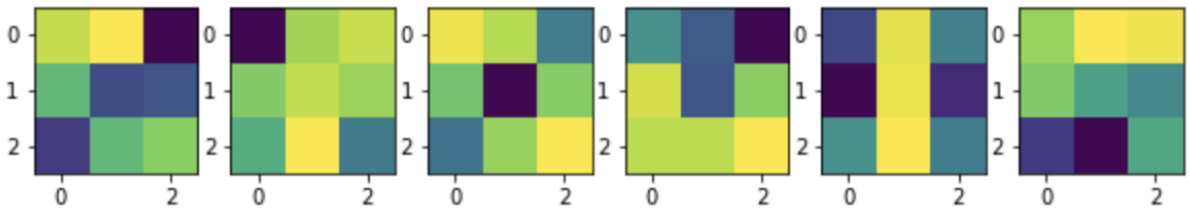
The second one:

```
class ConvolutionalNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 6, 3, 1)
        self.conv2 = nn.Conv2d(6, 16, 3, 1)
        self.fc1 = nn.Linear(5*5*16, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84,10)

    def forward(self, X):
        X = F.relu(self.conv1(X))
        X = F.max_pool2d(X, 2, 2)
        X = F.relu(self.conv2(X))
        X = F.max_pool2d(X, 2, 2)
        X = X.view(-1, 5*5*16)
        X = F.relu(self.fc1(X))
        X = F.relu(self.fc2(X))
        X = self.fc3(X)
        return F.log_softmax(X, dim=1)
```

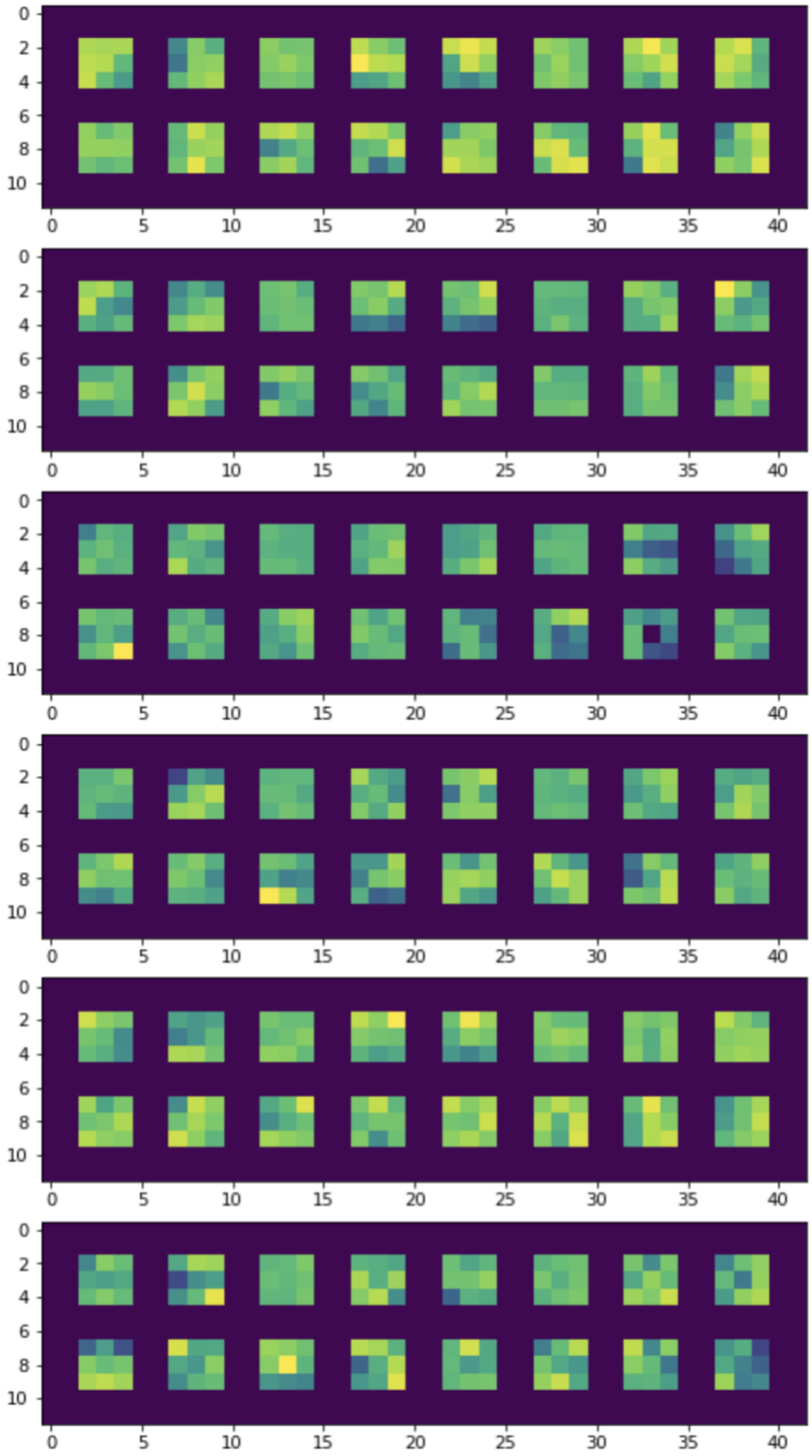
Turns out the second model has a better performance than the first model and it only takes about 3 epochs to converge as its loss error is practically 0 and the final test accuracy is 1.33% which is smaller than 1.4%.

Since for the new construction I have used multilayer Convolutional Network, below I will attach how the filter looks like in the CNN model.

First filters:



Second filters:



More about Deep Learning

In this part, we need to train our customized dataset through at least two of my favorite Deep Learning Algorithms, here is my choice: ANN and CNN with optim.Adam as my optimizer. The difference between this part and previous part is that I am using Adam as my optimizer and using accuracy for my plots instead of using misclassification error (in percentage). Therefore, I did not test how momentum will affect my model as Adad don't take momentum as one of its parameters.

As a brief introduction about our customized dataset, there are 3 txt files and have been split into 3 parts: train set contains 20,000 lines and validation set and test set are both containing 5000 lines in the same format. Each line contains 1569 coordinates, with the first 784 real-valued numbers correspond to the 784-pixel values for the first digit, and the next 784 real valued numbers correspond to the pixel values for the second digit.

A brief description about my workflow:

First of all, I define two customized dataset functions in order to turn the txt file into a similar dataset structure like MNIST dataset. As ANN and CNN takes different structured data inputs, that is ANN deals with flattened tabular data and CNN deals with data which keeps its spatial features, thus my two customized dataset functions are defined differently.

On top of that I build my ANN model and CNN model to train the dataset respectively and test my model with different parameters (learning rate of 0.2 and 0.5). Since here we are using the validation set instead of the test set like we did before, the validation curve below then stands for the test results for the validation set and test accuracy is calculated based on the test set.

Customized dataset function ANN

```
class CustomerDataset(Dataset):
    def __init__(self, file_path):
        file_out = pd.read_table(file_path,sep=',',names=range(1569))
        X = file_out.iloc[:,:1568].values
        y = file_out.iloc[:,1568].values
        # Standardlize y
        sc = StandardScaler()
        X = sc.fit_transform(X)
        # convert to tensor
        self.X = torch.tensor(X,dtype=torch.float32)
        self.y = torch.tensor(y)

    def __len__(self):
        return len(self.y)

    def __getitem__(self,idx):
        return self.X[idx],self.y[idx]


```

```
train_q5 = CustomerDataset(train_path)
val_q5 = CustomerDataset(val_path)
test_q5 = CustomerDataset(test_path)
```

```
train_loader = DataLoader(train_q5, batch_size=100, shuffle=True)
val_loader = DataLoader(val_q5,batch_size=100, shuffle=True)
test_loader = DataLoader(test_q5, batch_size=500, shuffle=False)
```

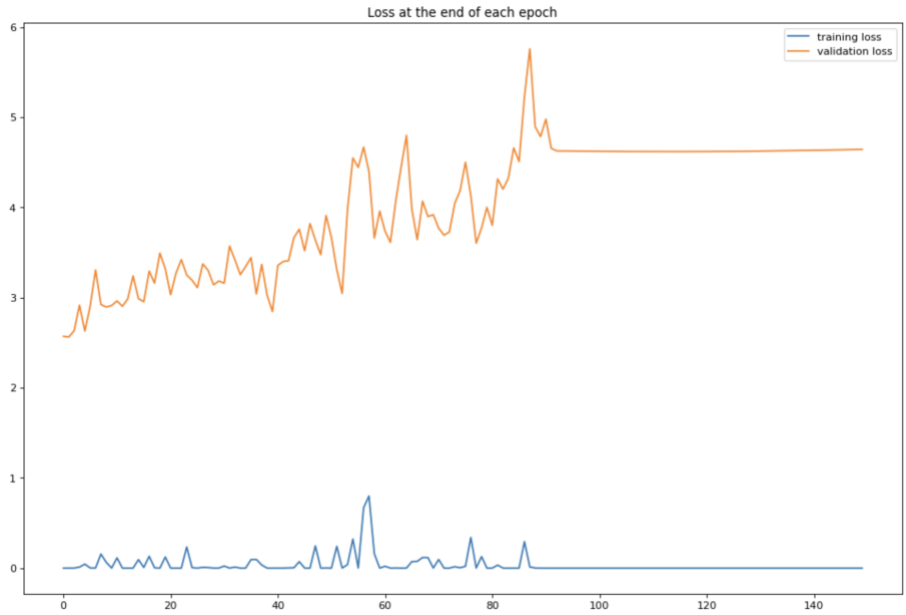
Starting with the ANN model, in this architecture, I have used 3 hidden layers with 784, 120, 84 units respectively and set up learning rate of 0.001 as my baseline model with 150 epochs per training.

Model structure

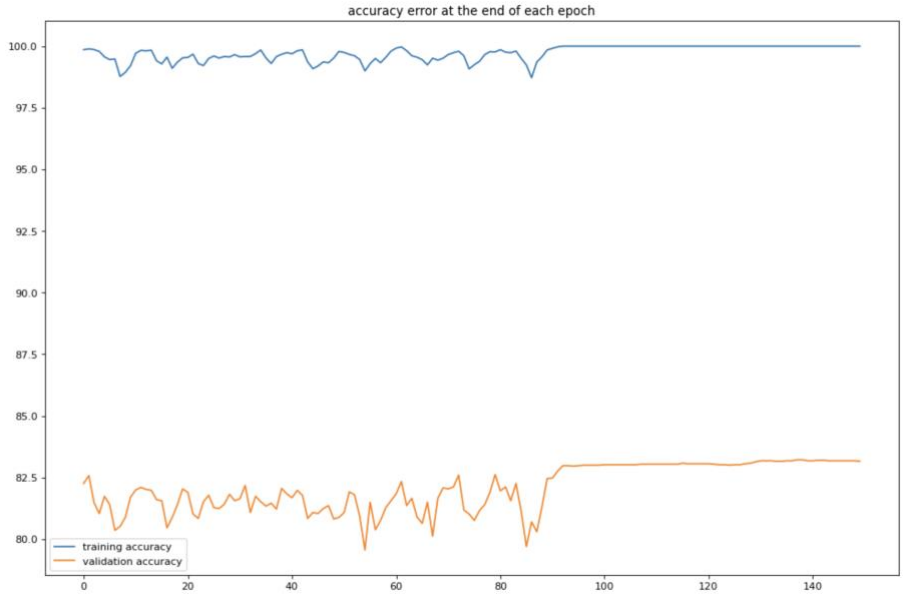
```
class MultilayerPerceptron(nn.Module):
    def __init__(self, in_sz=1568, out_sz=19, layers=[784,120,84]):
        super().__init__()
        self.fc1 = nn.Linear(in_sz,layers[0])
        self.fc2 = nn.Linear(layers[0],layers[1])
        self.fc3 = nn.Linear(layers[1],layers[2])
        self.fc4 = nn.Linear(layers[2],out_sz)

    def forward(self,X):
        X = F.relu(self.fc1(X))
        X = F.relu(self.fc2(X))
        X = F.relu(self.fc3(X))
        X = self.fc4(X)
        return F.log_softmax(X, dim=1)
```

At learning of 0.001, the model fails to converge at the first training, and it converges during the second training around the 85th epochs with highest test accuracy 83.16%

Loss Error (Second training)

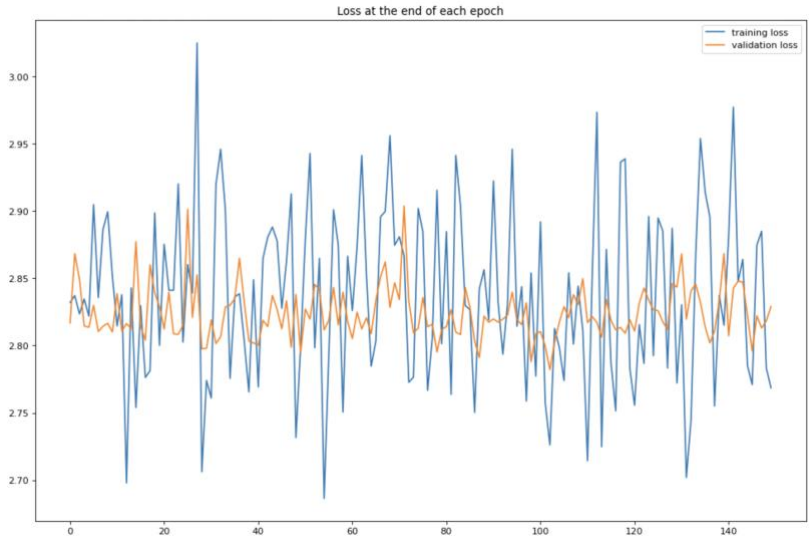


Accuracy (Second training)

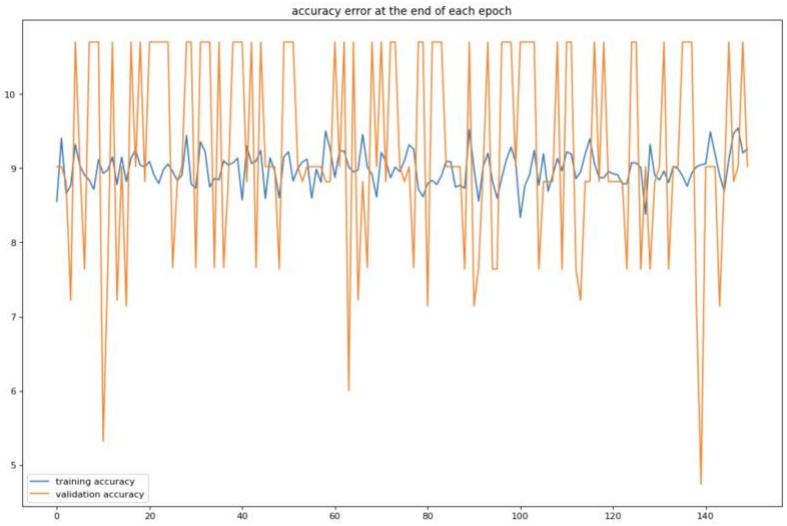


In this part, for both ANN and CNN big learning rate has a strong bad influence towards the training results and to certain extent, big learning rate will result in very bad results and leave the training performance looks unreasonable. For the ANN model we shall observe such phenomenon at learning rate of 0.5.

Loss Error (Learning rate 0.5)



Accuracy (Learning rate 0.5)



Then comes to the CNN model. The idea of creating the customized dataset function is to reshape the flattened data into a rectangular shape, that is restore the data into two 28 by 28 matrix and then concatenate those two matrices into one by the row.

Customized dataset function

```
class CustomerDataset(Dataset):
    def __init__(self, file_path):
        file_out = pd.read_table(file_path,sep=',',names=range(1569))
        X = np.array(file_out.iloc[:,:1568].values).reshape(file_out.shape[0],28,56)
        y = file_out.iloc[:,1568].values
        # convert to tensor
        self.X = torch.tensor(X,dtype=torch.float32)
        self.y = torch.tensor(y)

    def __len__(self):
        return len(self.y)

    def __getitem__(self,idx):
        return self.X[idx],self.y[idx]

```

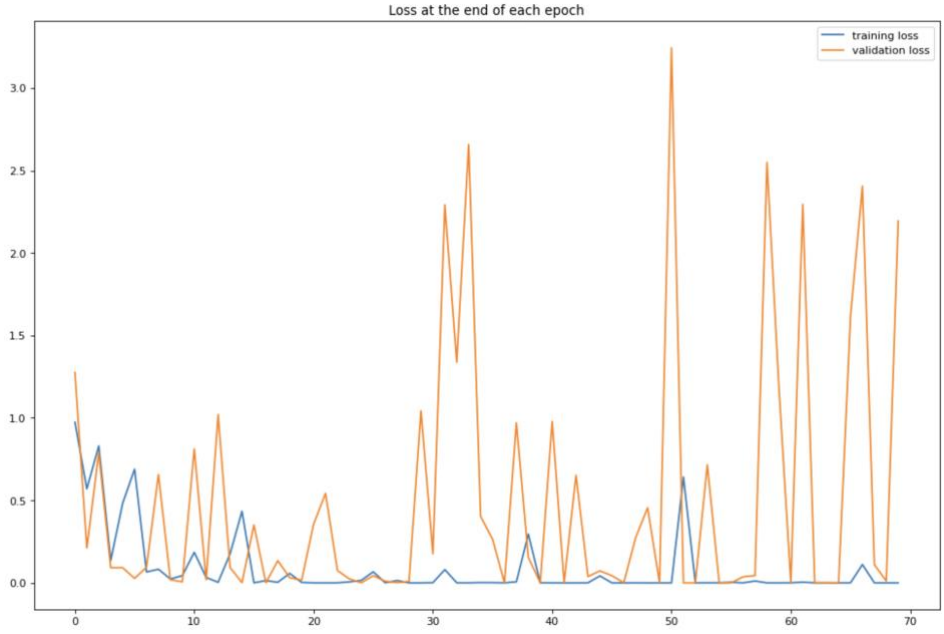
For the CNN model I have chosen the same architecture as what I did in the previous part: **Favorite Deep Learning Architecture.** Since the shape of data is a rectangular instead of a square and so the model has been modified slightly different. In here I also choose to set up learning rate of 0.001 as my baseline model with 70 epochs per training.

Model structure

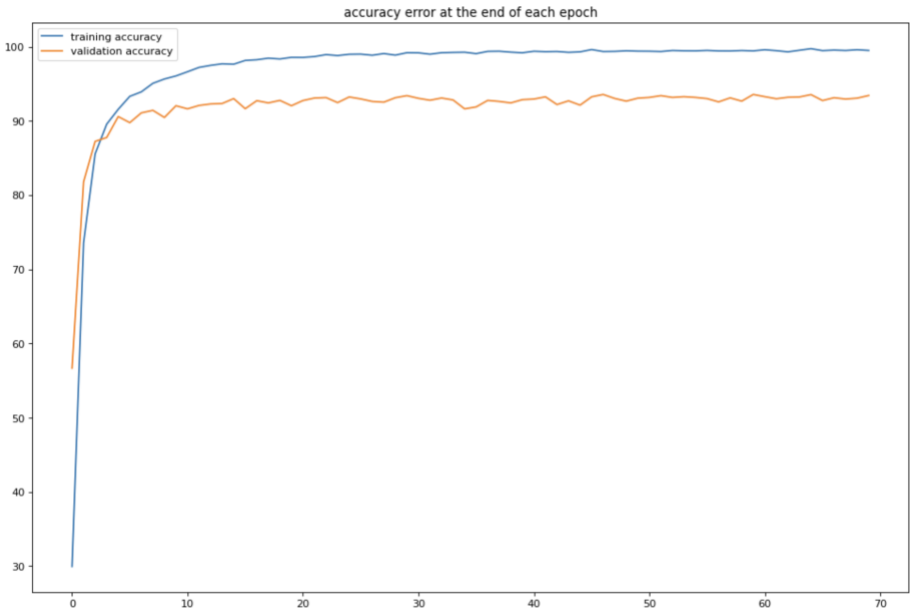
```
class ConvolutionalNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 6, 3, 1)
        self.conv2 = nn.Conv2d(6, 16, 3, 1)
        self.fc1 = nn.Linear(5*12*16, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84,19)

    def forward(self, X):
        X = F.relu(self.conv1(X))
        X = F.max_pool2d(X, 2, 2)
        X = F.relu(self.conv2(X))
        X = F.max_pool2d(X, 2, 2)
        X = X.view(-1, 5*12*16)
        X = F.relu(self.fc1(X))
        X = F.relu(self.fc2(X))
        X = self.fc3(X)
        return F.log_softmax(X, dim=1)
```

At learning of 0.001, the CNN model is able to converge at the 46th epochs during the first training with highest test accuracy 93.96%.

Loss Error

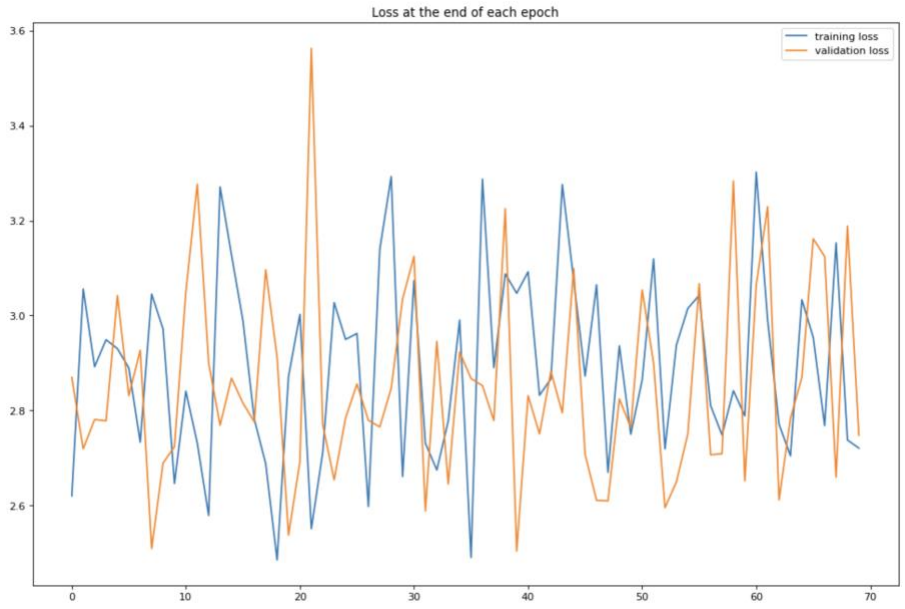


Accuracy

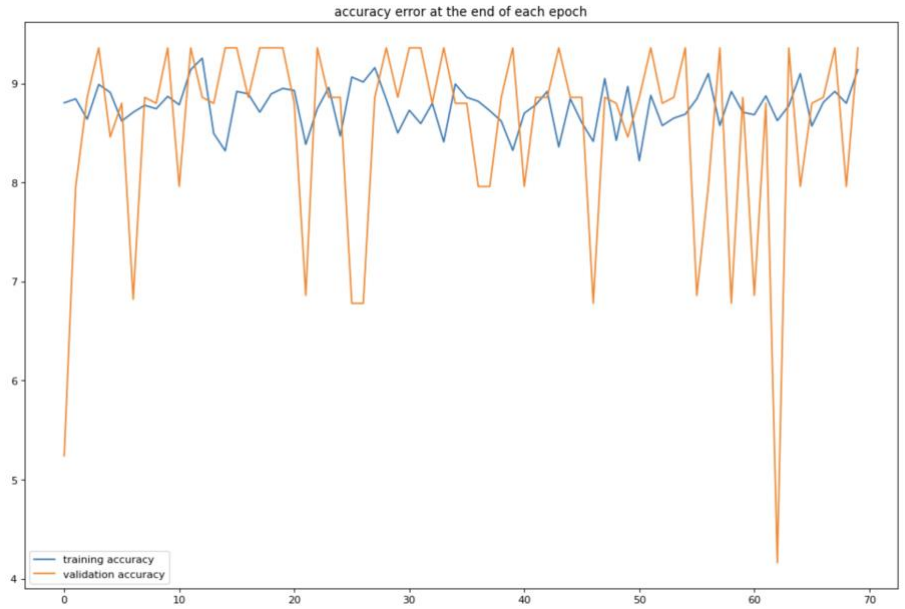


As mentioned before, big learning rate has even a stronger influence towards the CNN model. The result shows that for both learning rate of 0.2 and 0.5, the performance of CNN drops drastically. Below are the plots from learning rate of 0.5.

Loss Error (learning rate 0.5)



Accuracy (learning rate 0.5)



Hence the best performance is performed by CNN. Compared to the MNIST, the customized dataset has a poorer performance as the test accuracy is a lot less than what we got in MNIST dataset.

Reasons why for my model the test misclassification could not go blew 1% based on variance perspective and bias perspective.

Handling variance

1. From the plot we can see that the training accuracy is practically 100% and yet the validation accuracy is not which means our model has some overfitting problems and so we might want to add some regulation terms in our convolutional neural network.
2. Theoretically speaking, some of the images in the training set might be randomly rotated in both directions by some degrees or zoomed by some precents or shifted in different directions.
3. For my model, I did not add any dropout layers. If I could add some dropout layers after the pooling layers and some fully connected layers, the performance will probably get better.
4. The model did not use dynamical learning rate. Dynamically reducing the learning rate helps the algorithm to converge faster and reach closer to the global minima, since in this case the CNN model is very sensitive to the learning rate, hence I would consider using dynamical learning rate for future models and trainings.

Handling bias

1. The model did not have enough layers. The number of hidden in the dense layers were also increased to accommodate the larger input due to the increase in the volume

of the convolutional layers since the customized dataset has more information contained than the MNIST dataset.

2. The number of filters is insufficient. By increasing the number of layers and filters, we will obtain a deeper and denser network which allows the model to learn more complex features of the handwritten digits from our customized dataset.