Homework 1 Report CPSC 8430

Andrew Wright

GitHub Link: https://github.com/andrewright17/Homework-1-Deep-Learning

1-1 Deep vs Shallow

Simulate a Function

All models used had an input and output dimension of 1. The models I used are as follows:

- Model 1: DNN with 2 hidden layers of sizes 75 and 25. Total of 2076 parameters
- Model 2: DNN with 5 hidden layers of sizes 17, 26, 25, 23 and 12. Total of 2076 parameters
- Model 3: DNN with 1 hidden layer of size 692. Total of 2077 parameters

All models were trained for 100 epochs and I fine tuned for batch size and learning rate on all models.

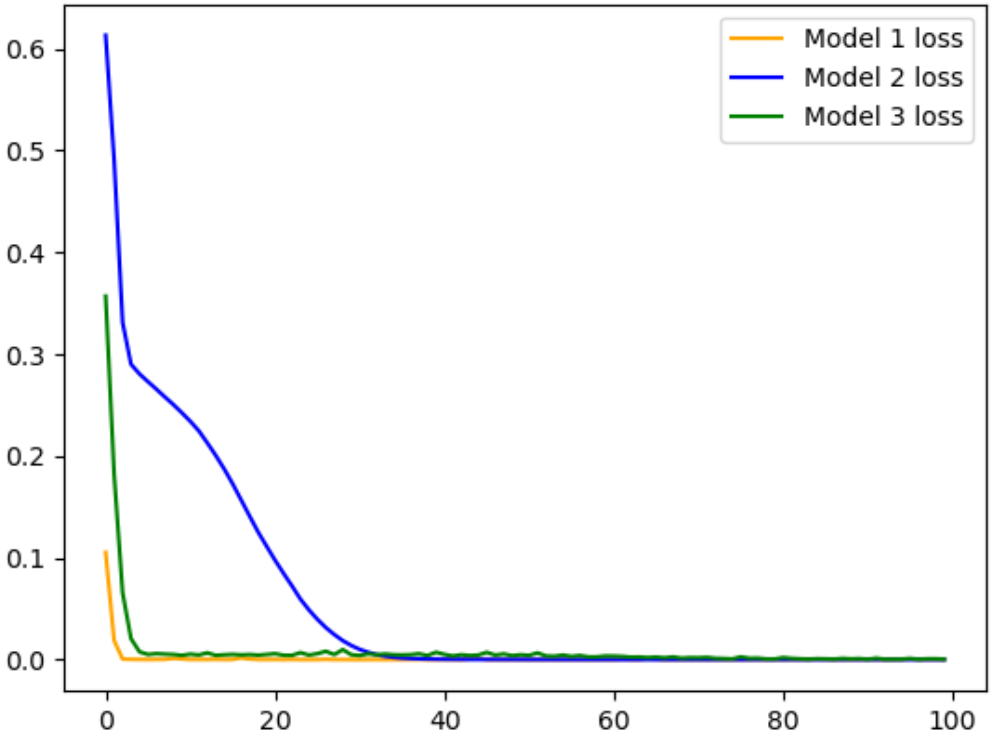
For function 1, we are simulating the function,

$$f(x) = 5\sin^2(\frac{x}{2})$$

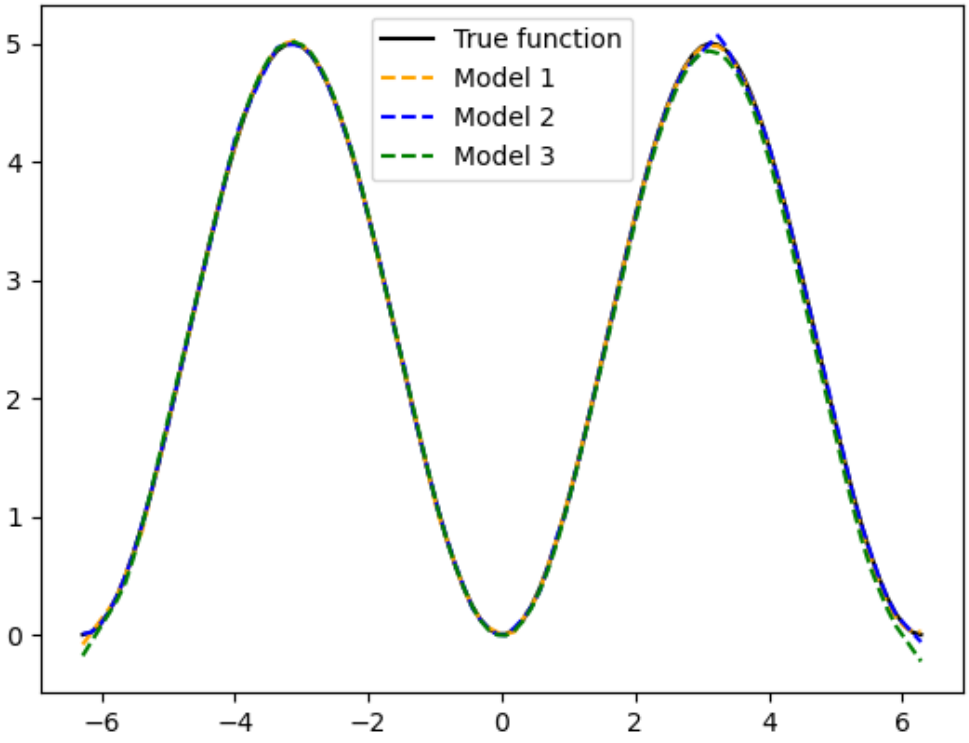
The optimal hyperparameters were:

- Model 1: bactch_size = 32, learning_rate = 0.01
- Model 2: bactch_size = 16, learning_rate = 0.0001
- Model 3: bactch_size = 8, learning_rate = 0.001

The loss for all models are shown in the plot below:



The predicted function curve of all the models is shown in the plot below along with the ground-truth curve:



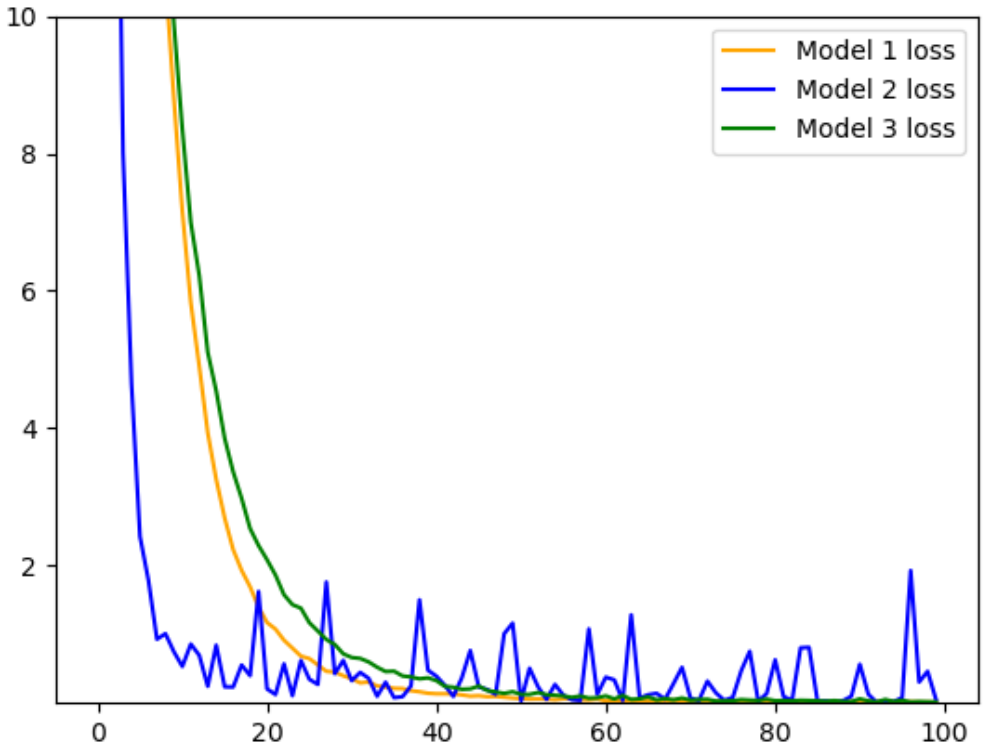
For function 2, we simulated the function,

$$f(x) = \frac{x^3 e^{(x/10)}}{15}$$

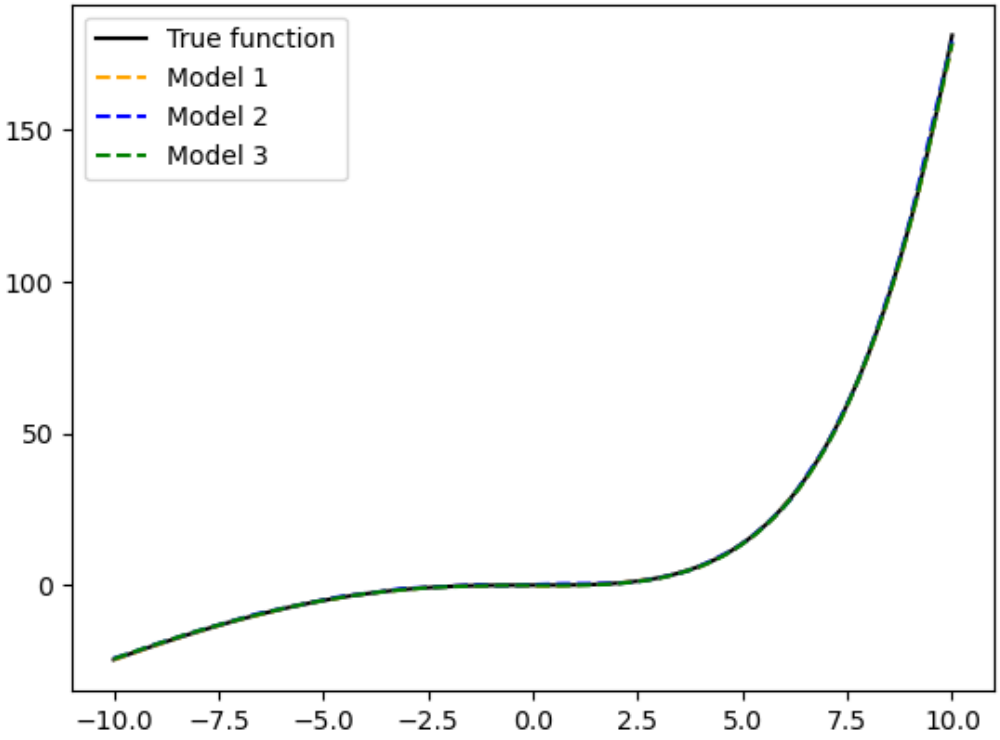
The optimal hyperparameters were:

- Model 1: bactch_size = 16, learning_rate = 0.001
- Model 2: bactch_size = 8, learning_rate = 0.001
- Model 3: bactch_size = 8, learning_rate = 0.001

The loss for all models are shown in the plot below:



The predicted function curve of all the models is shown in the plot below along with the ground-truth curve:



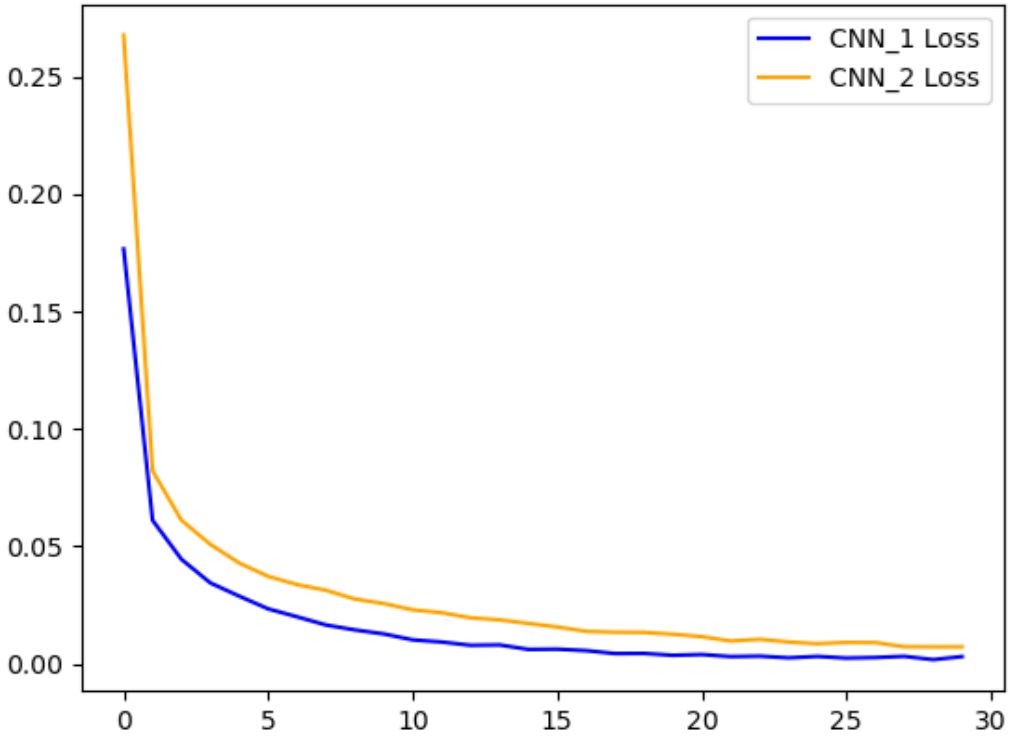
It appears that all models seemed to approximate the function fairly well. However, Model 3 slightly misses the curve at the endpoints of function 1. Model 3 is the shallowest of all the models, so it is intuitive that it could have more error than the other models.

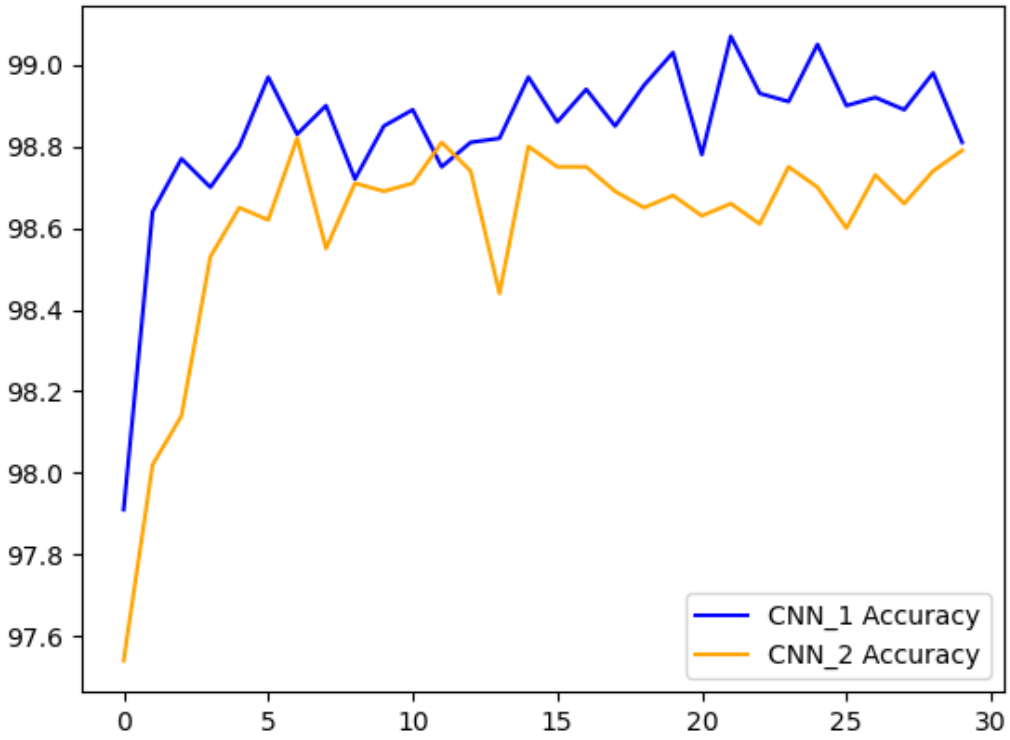
Train on Actual Task

For traininig on an acutal task, I chose to train on the MNIST dataset. The model architecture was similar for both models. I used CNNs with 2 convolutional layers, kernal size of 3 and padding of 1 in each layer, and max pooling with a kernal size of 2 after each Conv2d. Finally, there is a fully connected layer at the end.

- Model 1: CNN, first convolutional layer with 16 output channels, second convolutional layer with 32 output channels
- Model 2: CNN, first convolutional layer with 8 output channels, second convolutional layer with 16 output channels

The plots of the loss and accuracy are below:



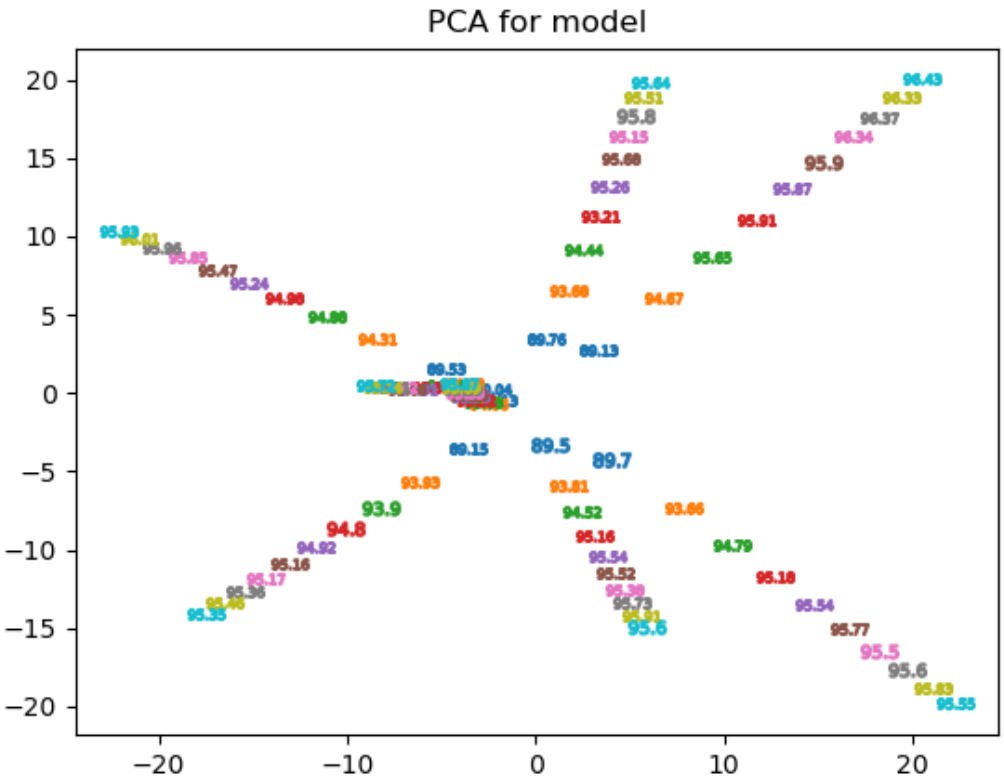


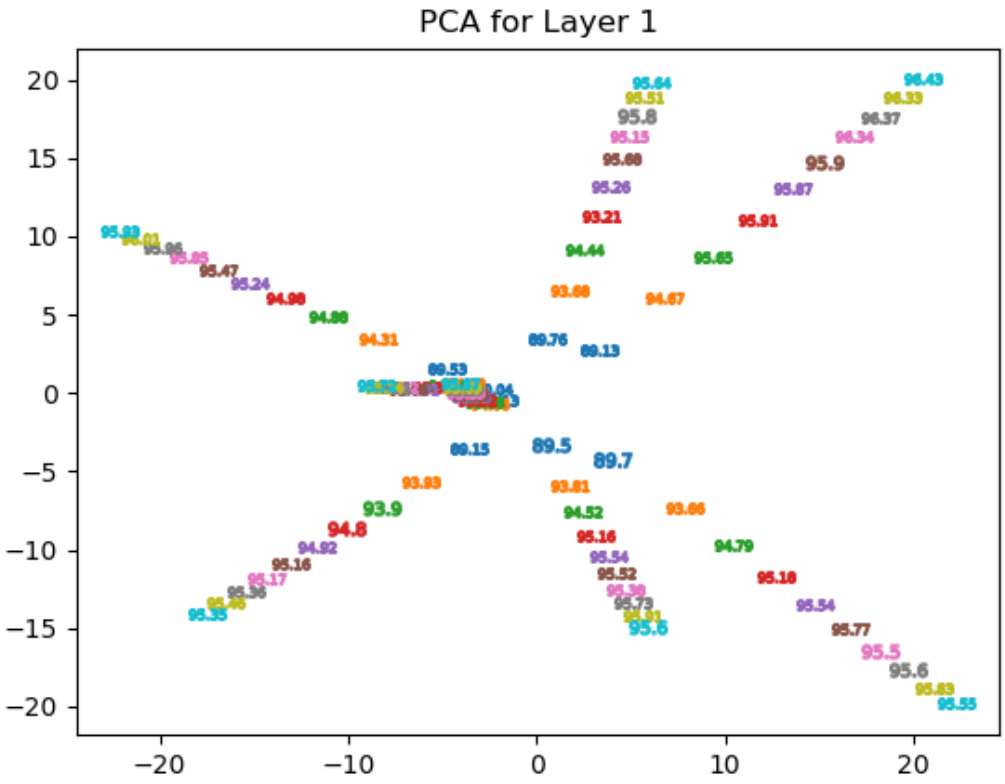
1-2 Optimization

Visualize Optimization Process

To visualize the optimization process, I trained a DNN with hidden layers of sizes 17, 26, 25, 23, and 12 on the MNIST dataset. During the training process, I collected the weights of the model parameters at each epoch. These weights were stored in a dataframe where the granularity was 1 row per epoch per training. The model trained 8 times for 30 epochs each.

The plots below are the PCA for the whole model, followed by the PCA for the first layer. Weights were sampled for every 3 epochs.



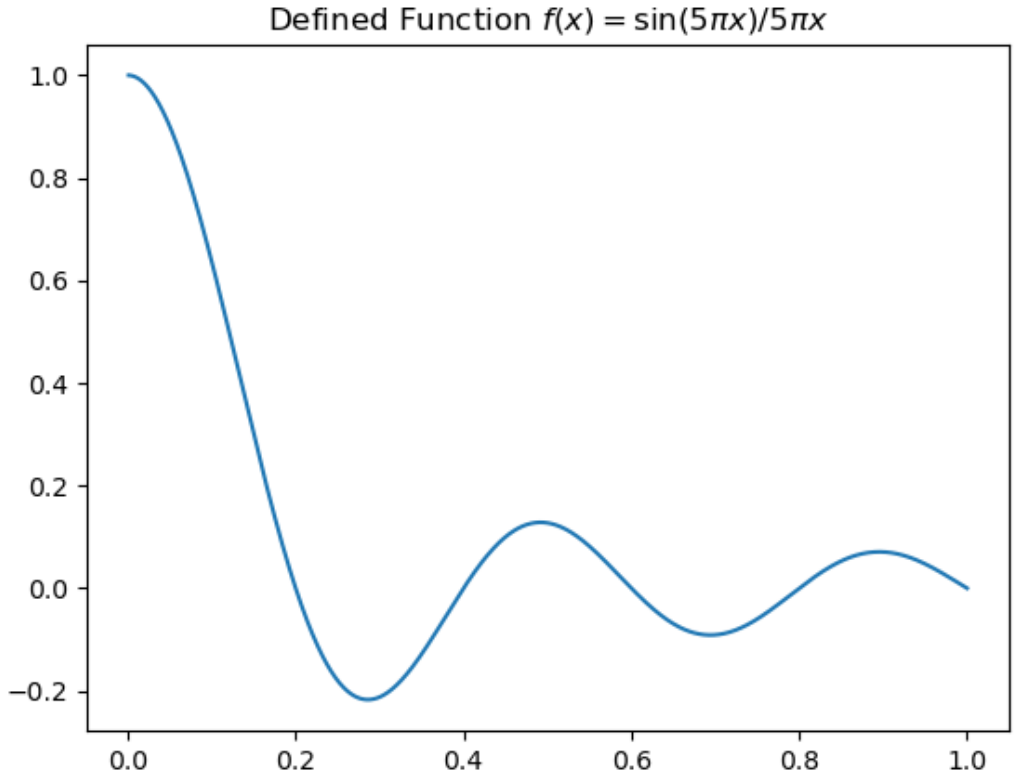


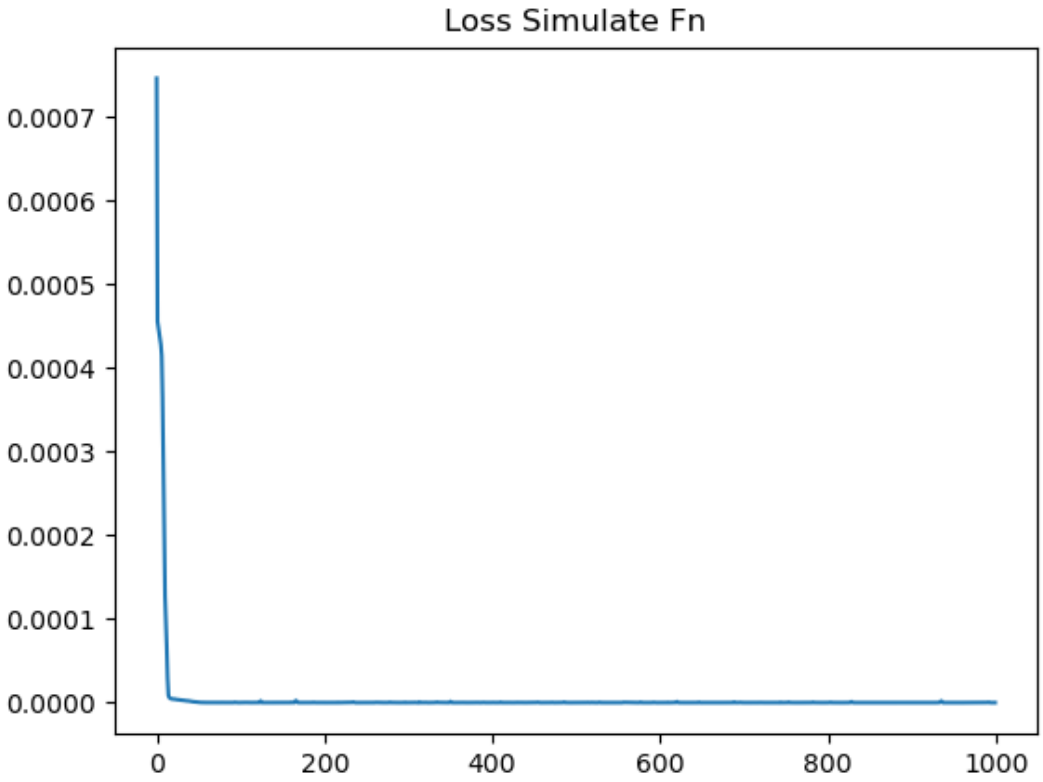
You can see the parameters dispersing throughout the training process. The accuracy of the model—indicated by the points of the plots—increases as we move toward the edges.

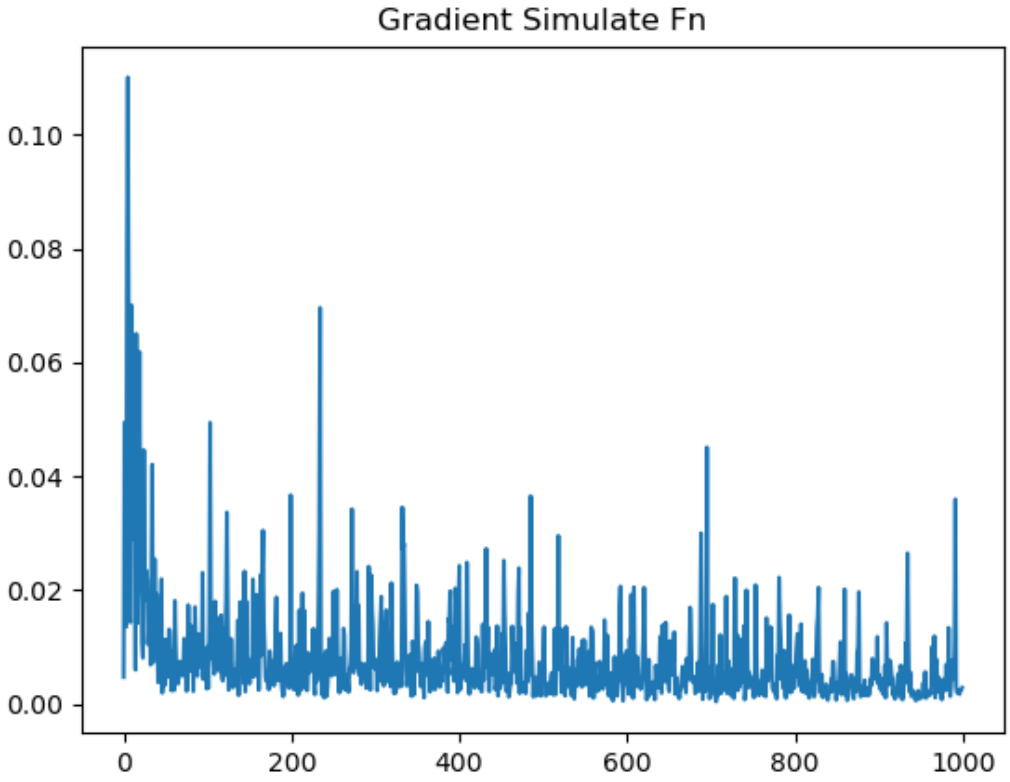
Observe Gradient Norm During Training

Below are the plots for the loss function and the gradient norm during training. I trained the same model as above (with different input and output sizes) on the function,

$$f(x) = \frac{\sin(5\pi x)}{5\pi x}$$


Defined Function $f(x) = \sin(5\pi x)/5\pi x$
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0


Loss Simulate Fn
0.0007
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
0
200
400
600
800
1000

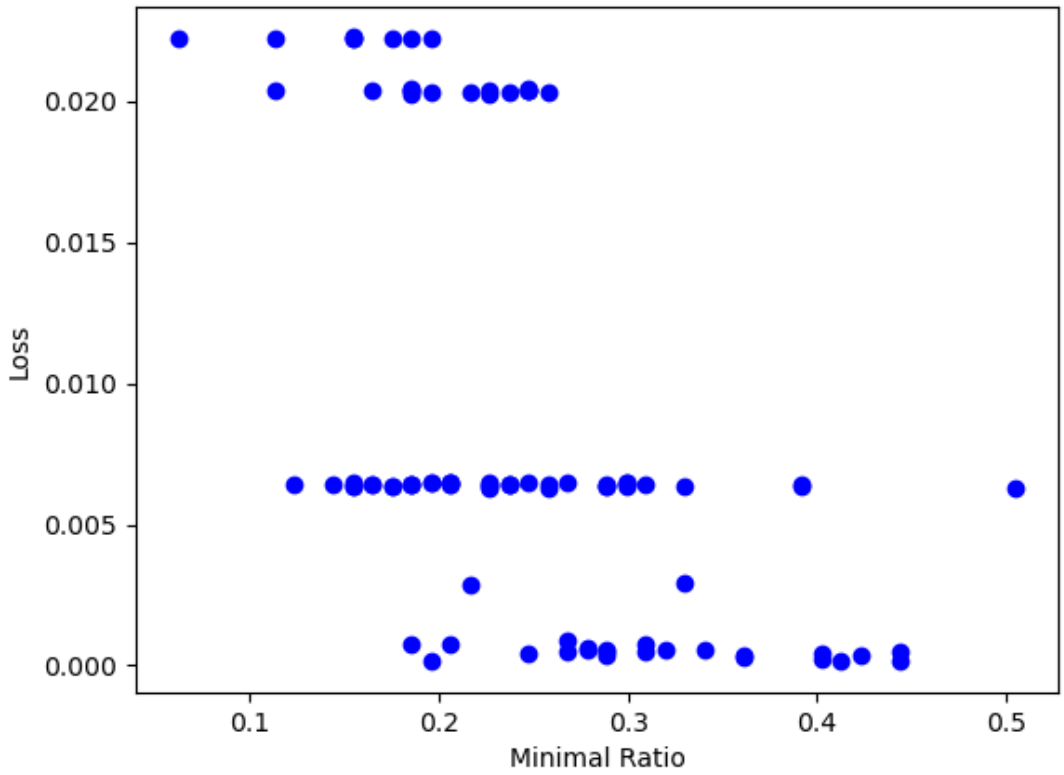


You'll notice that the loss converges rather quickly (less than 100 epochs), but the gradient norm takes much longer to converge. Although, you can see the variability decrease as training progresses.

What Happens When Gradient is Close to Zero

In order to calculate the minimal ratio, I used the Hessian matrix approach which involved calculating the eigenvalues. The minimal ratio definition for this problem is the proportion of eigenvalues greater than 0. The way I got the weight which was equal to 0 is by setting a threshold of 0.001. When the gradient norm was less than 0.001, training was stopped and that was considered to be 0 for practical purposes.

Below is a plot of the minimal ratio to the loss:



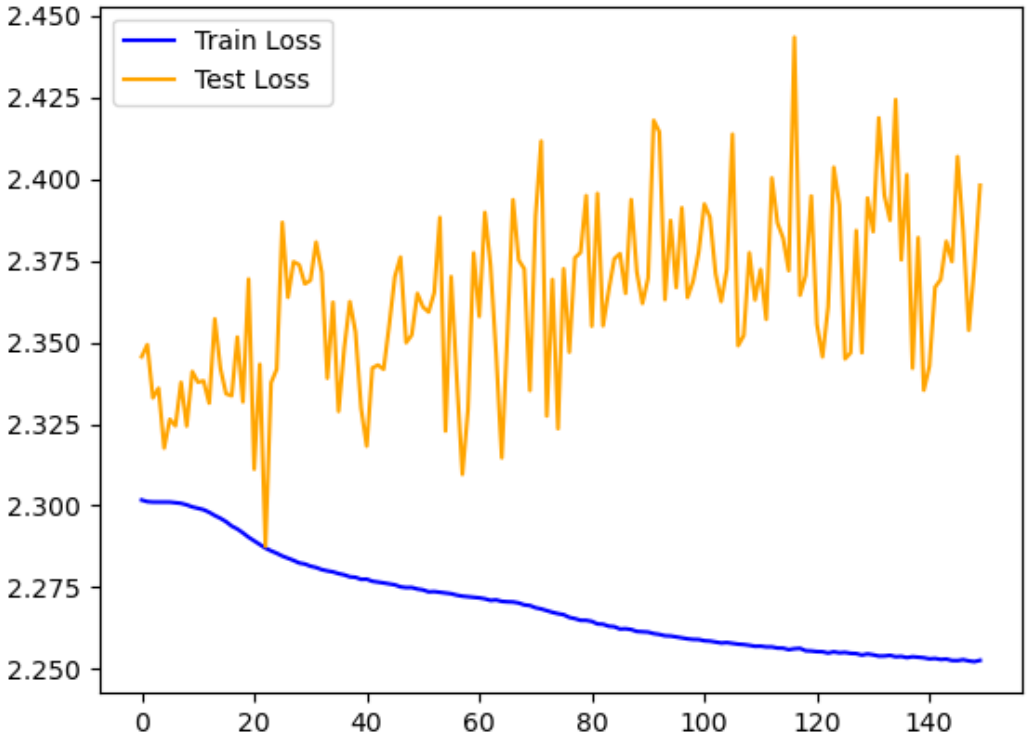
Notice that the minimal ratio greater than roughly 0.25 leads to a smaller loss.

1-3 Generalization

Fitting Random Labels

To fit a model on random labels, I used a CNN with the exact same specifications as Model 1 in 1-1 pt 2 above. This model was trained for 150 epochs on the MNIST dataset. Hyperparameters were set to a batch_size of 64 and learning_rate of 0.001. The optimizer used was Adam.

To make the MNIST labels random, I used random.shuffle() in the training and testing dataloaders. The loss and accuracy are plotted below.

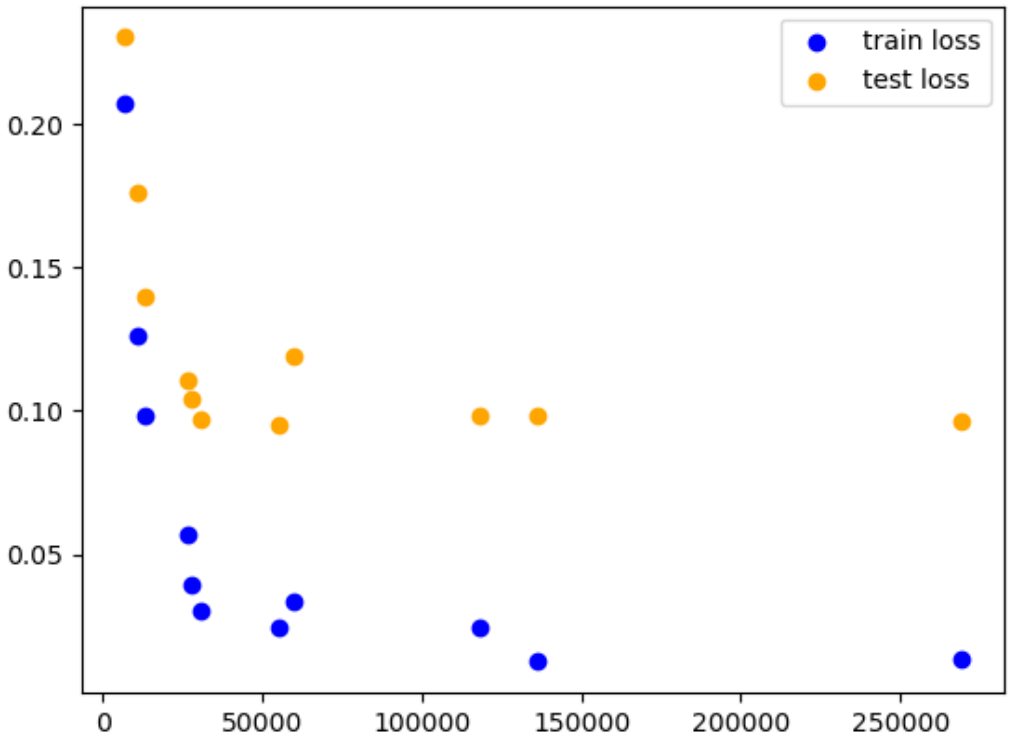


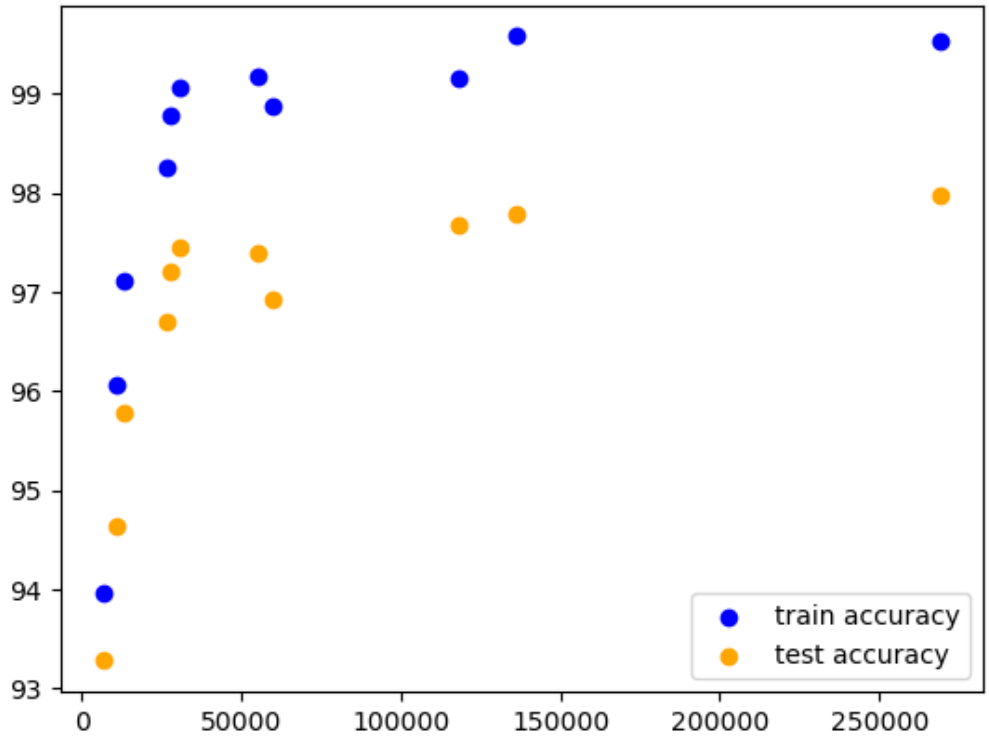
Number of Parameters vs Generalization

For this part, I trained once again on the MNIST dataset. The model structures I chose were DNNs with 2 hidden layers that varied in sizes. The hidden layer sizes were:

- model1 had hidden sizes of 8 & 16
- model2 had hidden sizes of 16 & 32
- model3 had hidden sizes of 32 & 32
- model4 had hidden sizes of 32 & 64
- model5 had hidden sizes of 64 & 64
- model6 had hidden sizes of 64 & 128
- model7 had hidden sizes of 128 & 128
- model8 had hidden sizes of 128 & 256
- model9 had hidden sizes of 256 & 256
- model10 had hidden sizes of 32 & 128
- model11 had hidden sizes of 8 & 256

The plots of the accuracy and loss for both training and testing are below.





Notice that as the parameters increase, we see generally both a decrease in loss and an increase in accuracy for training and testing. This is incredibly interesting because it seems

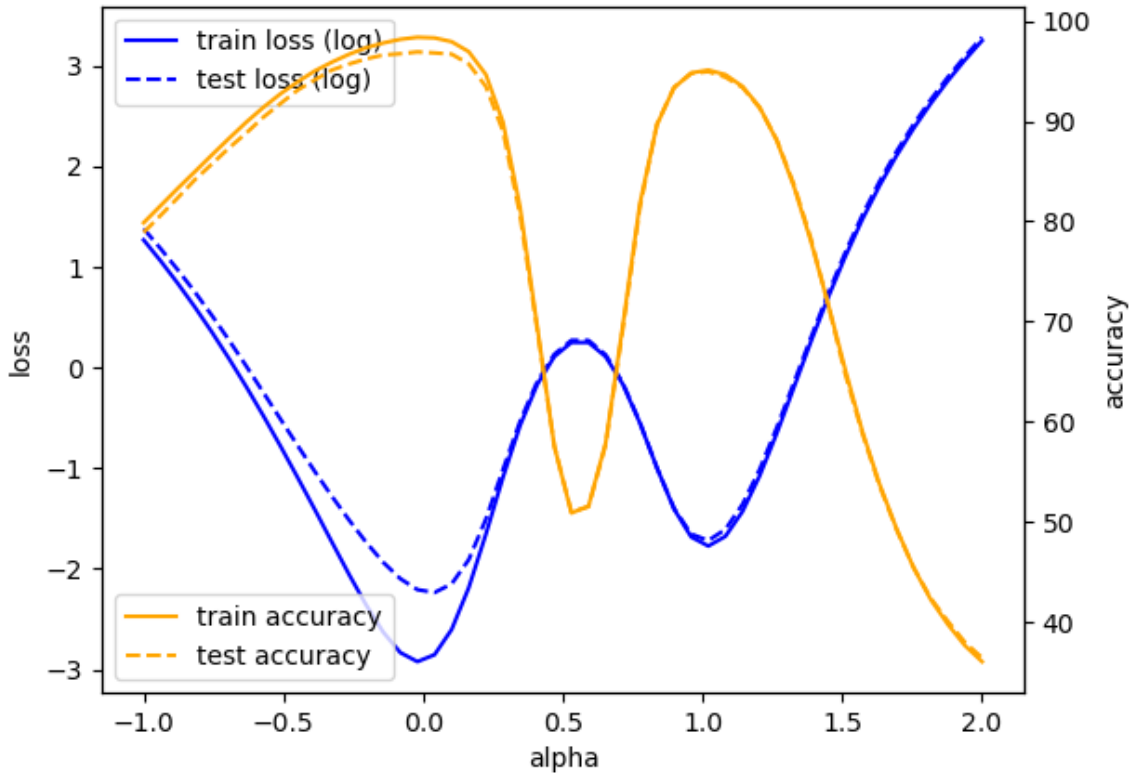
to show us that bigger is better when it comes to the size and performance of models. However, notice that performance improvements are much more gradual as the models get very large (i.e. 150,000 parameters), but that increases in parameters for models under 50,000 parameters or so show significant performance improvement. In other words, for smaller models, we are likely to see a noticeable performance improvement if we increase the size of the model even a relatively small amount.

Flatness vs Generealization

Part 1

For part 1, I used two different batch_sizes. m1 had a batch_size of 64 and m2 had a batch_size of 1024. Both models were trained on the MNIST dataset. Both models were DNNs with two hidden layers of sizes 32 and 64.

Below is the plot of training and testing for both loss and accuracy including the interpolation parameter.

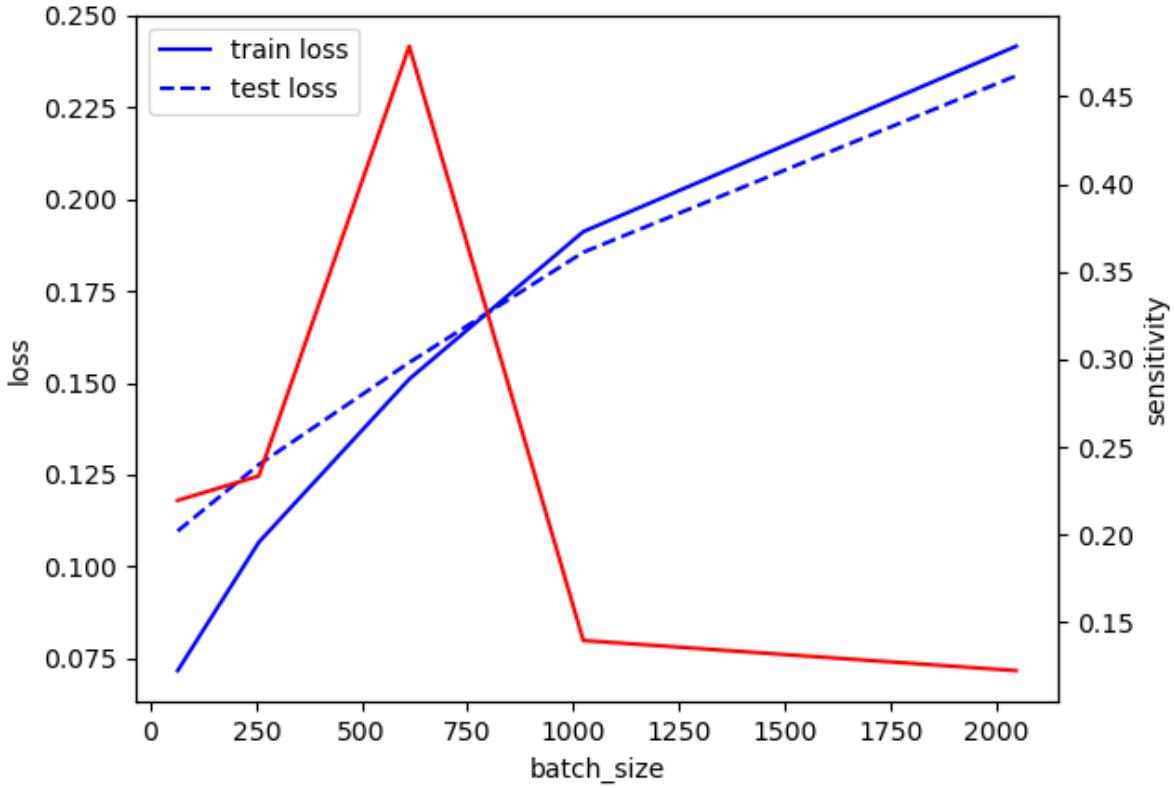


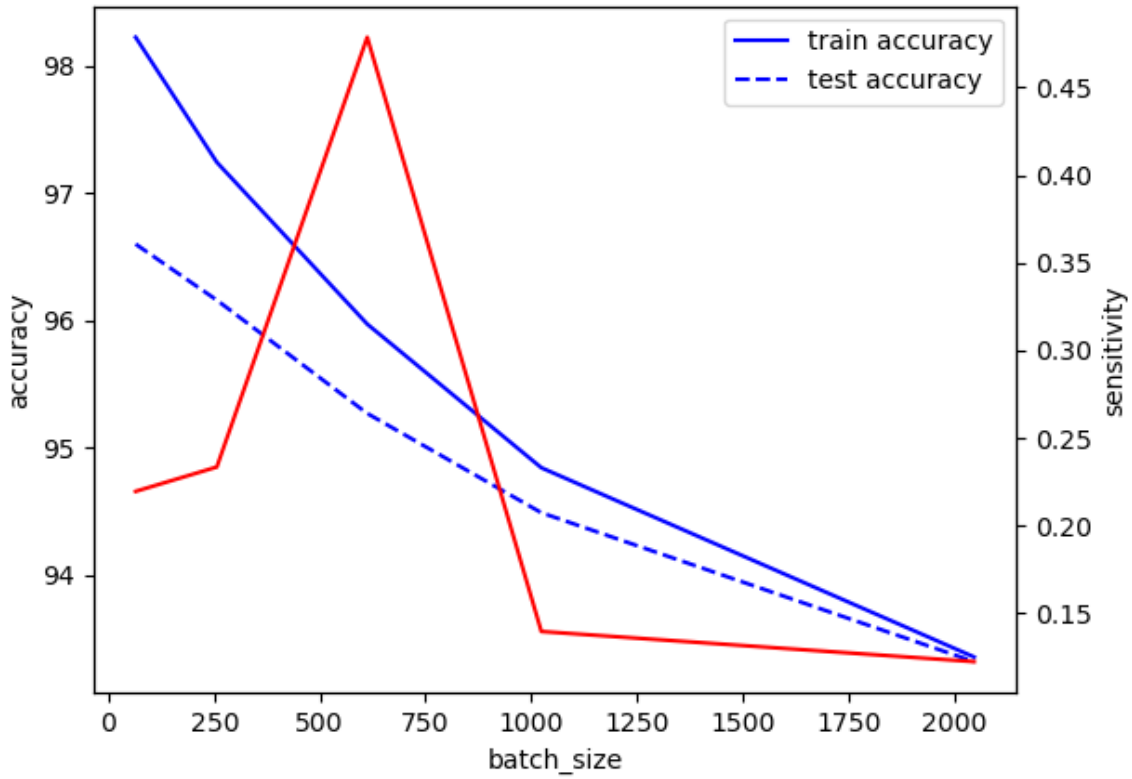
Notice that α gives minimum loss and maximum accuracy close to 0 and 1. At these points, the parameters are either entirely from model 1 or entirely from model 2. This is interesting as it suggests training a single model may yeild better results than a combination of two models with only changes in training approach.

Part 2

For part 2, I also trained on the MNIST dataset. All 5 models were DNNs with two hidden layers of sizes 32 and 64. The training approach we used was to adjust the batch_size for ech model. The batch_sizes used were 64, 256, 612, 1024, and 2048.

Below are the plots of training and testing for both loss and accuracy including sensitivity.





Notice that the model performance––in terms of loss, accuracy, and sensitivity––decreases as the batch_size increases. This is what we would expect as we know that a smaller batch_size is usually better, but comes at the cost of computational time.