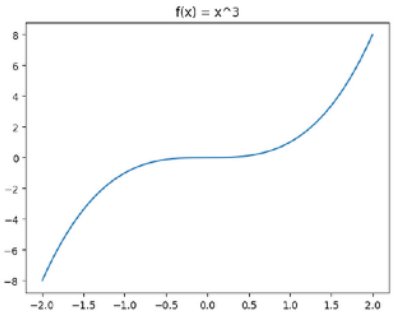
1-1 Deep vs Shallow

Simulate a function



REQUIREMENTS

- *Function needs to be singe-input, single-output*
- *Function needs to be non linear*

FUNCTION 1

- The first function I decided to simulate is $f(x) = x^3$
- I decided to simulate the function from -2 to 2 using 40, 000 points

MODEL 1

For the first model I implement a Neural Network with:

- An input layer (of one element)
- One hidden layer with dimension (1, 128)
- One output layer with dimension (128, 1)

This means that the model has 4 parameters.

I decided to use the *ReLU* function as the activation function of the model.
The model uses a mean squared error (L2 norm) function to calculate the loss.
To update the parameters the model uses the *Stochastic Gradient Descent (SGD)* algorithm.

After comparing the training process with different hyper-parameters the best configuration of the model is with

- A learning rate of 0.01
- 3000 epochs (after that the model does not achieve better results)

MODEL 2

For the second model I implement a Neural Network with a sequential model with:

- An input layer (of one element)
- One hidden layer with dimension (1, 1024)
- One output layer with dimension (1024, 1)

The Sequential model (*nn.Sequential* in pytorch) wraps the layers in the network.
Again, the model has 4 parameters.

The model uses the *ReLU* function as the activation function of the model.
Again, the model uses a mean squared error (L2 norm) function to calculate the loss of our predictions.
Instead of using *Stochastic Gradient Descent* (like in the previous model) this model uses Adam for the optimization algorithm.

After comparing the training process with different hyper-parameters the best configuration of the model is with

- A learning rate of 0.001 (unlike with the previous parameter a better model was achieved with a smaller learning rate)
- 3000 epochs (after that the model does not achieve better results)

MODEL 3 (BONUS)
Since this was a BONUS question I decided to play with this model a try a different configuration.

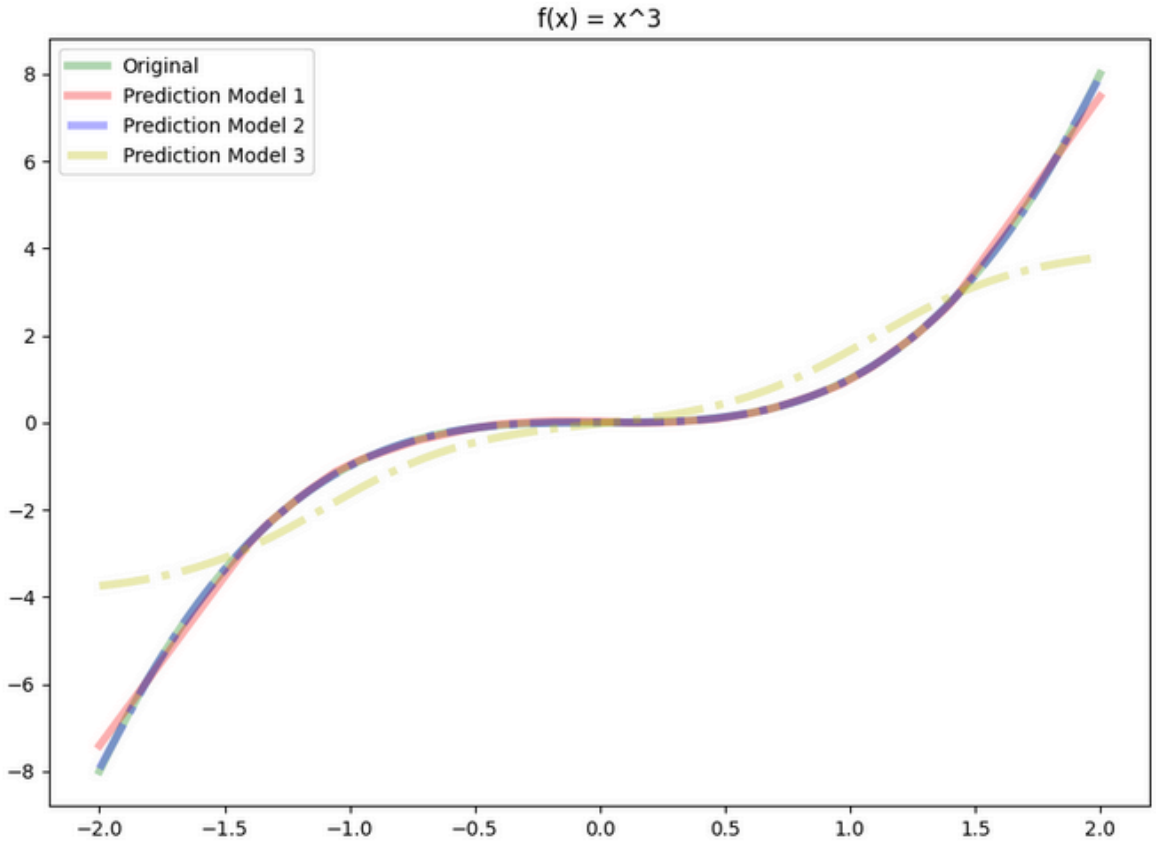
For the third model I implement a Neural Network with a sequential model with:
- An input layer (of one element)
- One hidden layer with dimension (1, 28)
- One output layer with dimension (28, 1)

Again, the model has 4 parameters.

The model uses the *Softmax*(dimension=1) function as the activation function of the model.
To calculate the loss of our predictions, the model uses the *CrossEntropyLoss* function.
The model uses Adam for the optimization algorithm.

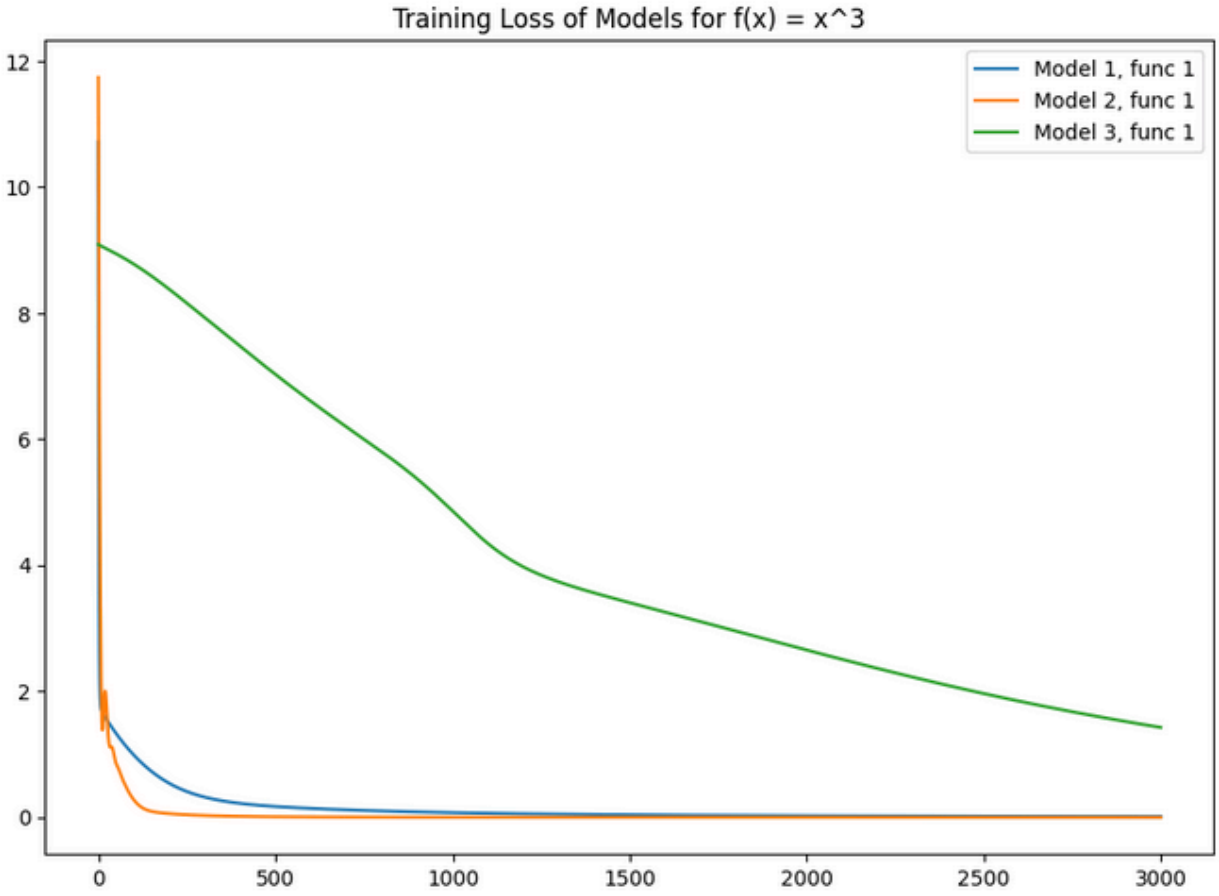
After comparing the training process with different hyper-parameters the best configuration of the model is with
- A learning rate of 0.001
- 3000 epochs (after that the model does not achieve better results)

RESULTS
Ground-truth and predictions by model



From figure 1 we can see that both Model1 and Model 2 simulate the function almost perfectly. Model 3 comes close but is not as good as neither Model 1 or Model 2.

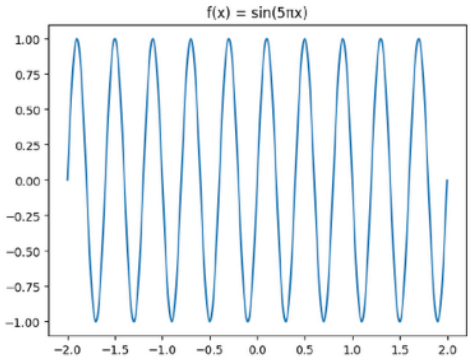
Model Comparison by loss:



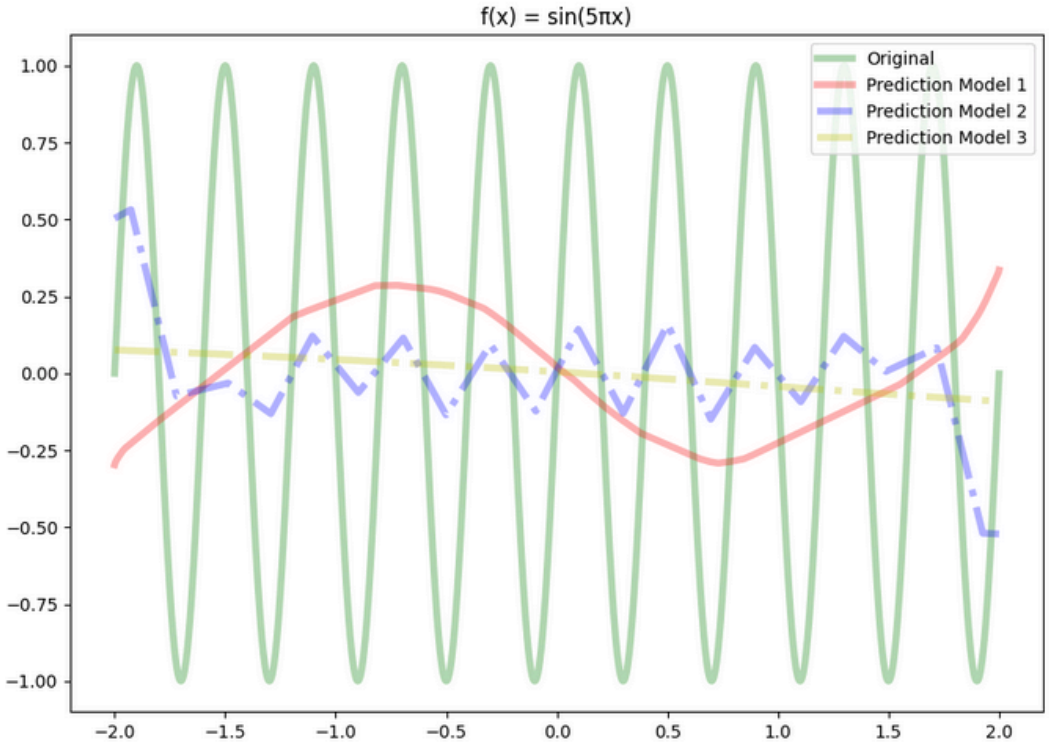
When comparing the loss of the models we can see that both Model 1 and Model 2 decrease and converge rapidly and almost at the same rate, while Model 3 loss decreases at a lower rate. This confirms why Model 1 and Model 2 simulate the function much better than model 3.

FUNCTION 2 (BONUS)

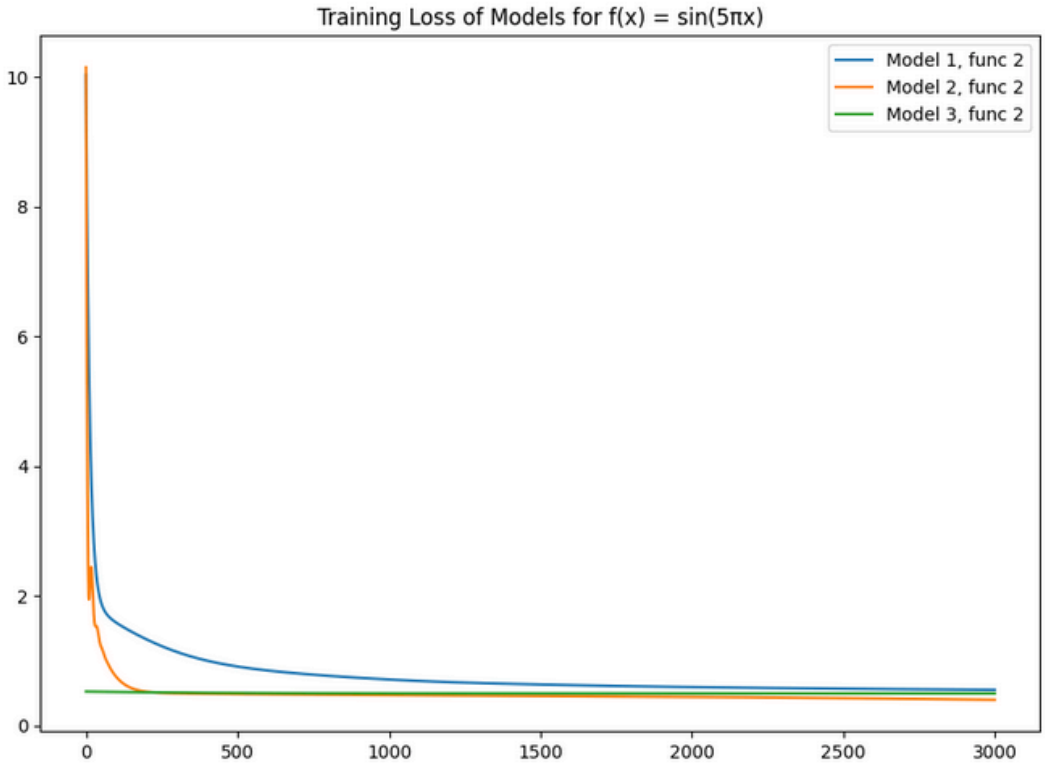
- The second function I decided to simulate is f(x) = sin(5πx)
- Again, I decided to simulate the function from -2 to 2 using 40, 000 points



I used the same 3 models for the second function:



Because the second function is more “complex” the three models fail to simulate it as well as the first function. From the figure we can see that the model that best simulates the function is Model 2. This is confirmed when comparing the loss of the models:



Train on actual task using shallow and deep models

REQUIREMENTS

- *Use MNIST or CIFAR-10*
- *Use CNN or DNN*
- *Visualize the training process by showing both loss and accuracy on two charts*

MNIST

- I decided to use the MNIST dataset to train a DNN

MODEL 1

For the first model I implement a Neural Network with:

- An input layer of size 28 * 28 = 784
  - Each image is made of 28 rows and 28 columns (a matrix of 28x28) with a value between 0 and 255 that represents a black and white color scale (0 = black and 255 = white)
- One hidden layer of dimension (784, 128)
- One output layer with dimension (128, 10)
  - The output is a layer of 10 because each index (0 to 9) represents the possible labels that the image represents

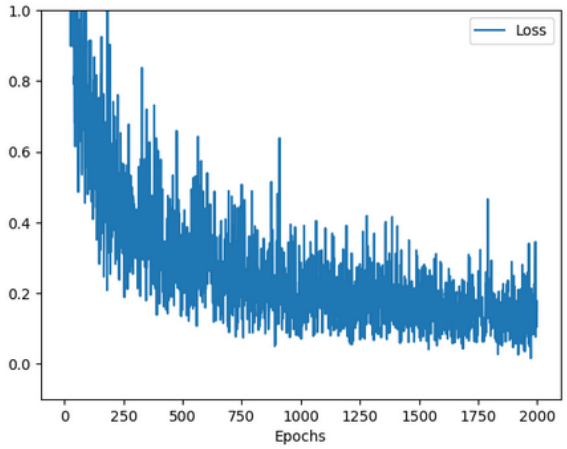
- The model uses a ReLU activation function.
- I decided to use the *CrossEntropyLoss*() for the loss function . *CrossEntropyLoss*() is very useful in training multiclass classification problems. In this problem we have 9 possible classes (integers form 0 to 9).
- I decided to use the Stochastic Gradient Descent for the optimization algorithm although Adam also performed well in the different tests I ran.

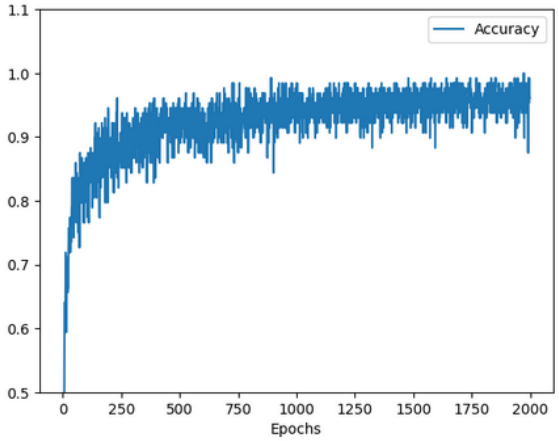
The model perform best when the batch size is 128 and these samples are “shuffled” randomly.

After comparing the training process with different hyper-parameters the best configuration of the model is with

- A learning rate of 0.001
- 3000 epochs (after that the model does not achieve better results)

Here are the results:





We can see that this model performs really well, as the loss decreases, the accuracy increases and both converge quickly.

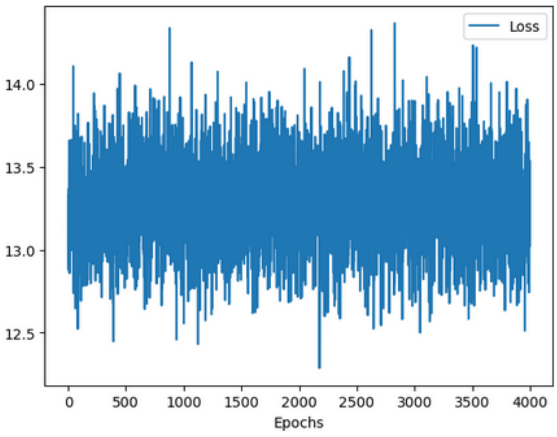
MODEL 2

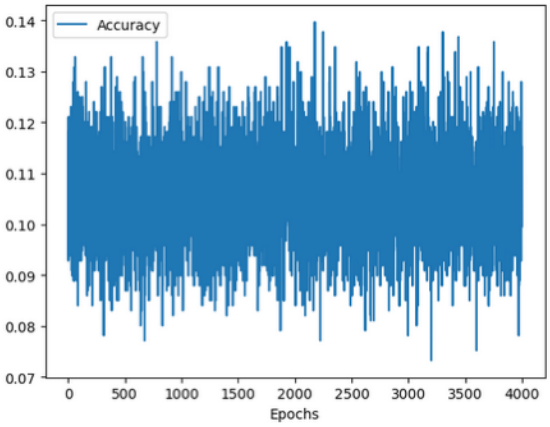
For the second model I implement a Neural Network with a sequential model with:

- An input layer of size 28 * 28 = 784
  - Each image is made of 28 rows and 28 columns (a matrix of 28x28) with a value between 0 and 255 that represents a black and white color scale (0 = black and 255 = white)
- Two hidden layers
  - (784, 128) with ReLU activation function
  - (128, 64) with ReLU activation function
- One output layer with dimension (64, 10)
  - The output is a layer of 10 because each index (0 to 9) represents the possible labels that the image represents
  - Uses *softmax* activation function (*softmax* are used with classification problems)

I found the best results for this model were achieved when it used the whole train set (no sampling)

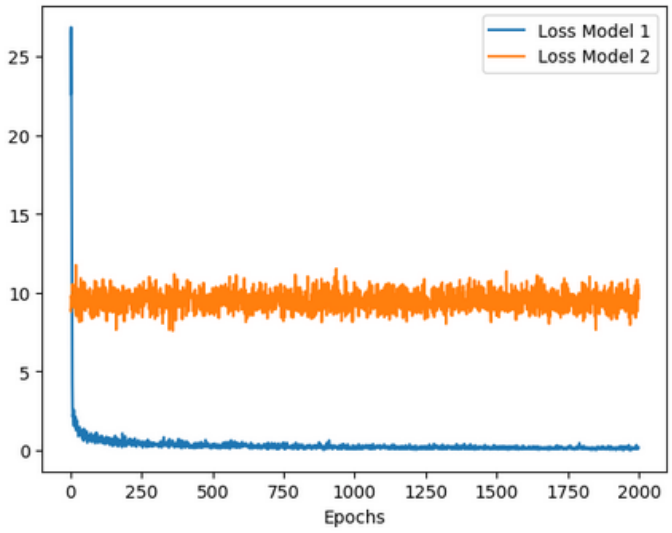
To measure the loss of our predictions this model uses *NLLLoss* or the negative log likelihood loss. This function is useful to train classification problem (just like this one)

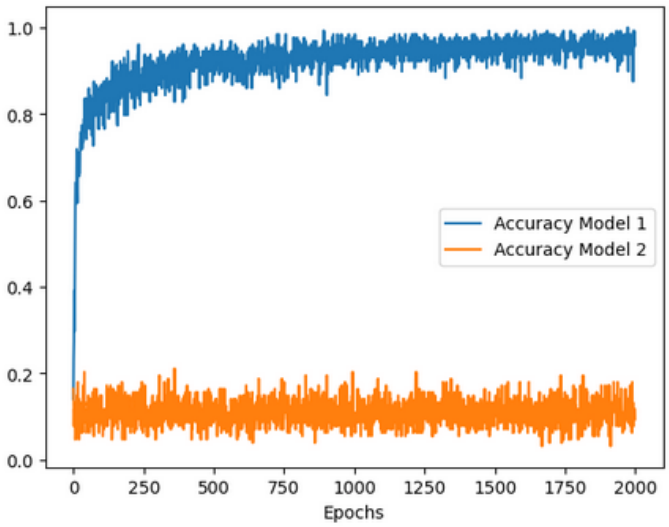




COMPARING MODELS

Model 1 is clearly superior to model 2 both in terms of loss and accuracy. It has more accuracy when predicting the values that the images (and therefore less loss)





https://github.com/David-FR/8430_HMW1.git

1-2 Optimization

Visualize the optimization process

REQUIREMENTS

- *Collect weights of the model every n epochs.*
- *Also collect the weights of the model of different training events.*
- *Record the accuracy (loss) corresponding to the collected parameters.*

MODEL

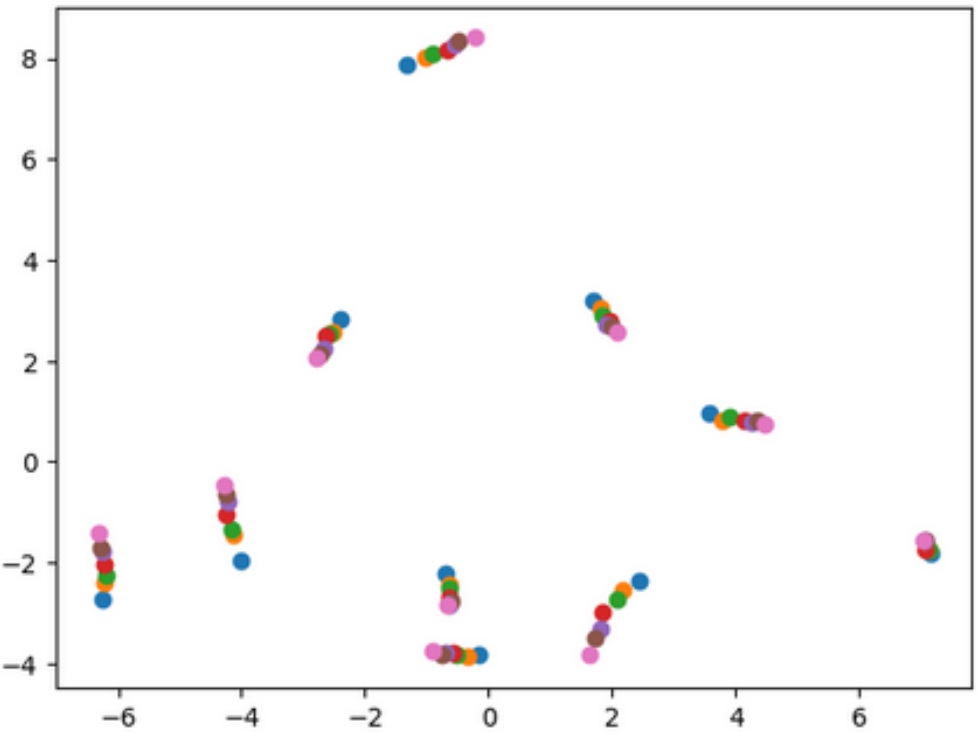
To visualize the optimization process I used the model 1 of last section (Model 1):

- An input layer of size 28 * 28 = 784
  - Each image is made of 28 rows and 28 columns (a matrix of 28x28) with a value between 0 and 255 that represents a black and white color scale (0 = black and 255 = white)
- One hidden layer of dimension (784, 128)
- One output layer with dimension (128, 10)
  - The output is a layer of 10 because each index (0 to 9) represents the possible labels that the image represents

- The model uses a ReLU activation function.
- I decided to use the *CrossEntropyLoss*() for the loss function
- Stochastic Gradient Descent for the optimization algorithm
- A learning rate of 0.001
- 3000 epochs

I decided to collect the weights of the second hidden layer every 3 epochs while training the model 8 times (recommendations from instructions). I got similar results when visualizing the optimization process of the whole model

VISUALIZATION OF THE OPTIMIZATION PROCESS:



Observe gradient norm during training

REQUIREMENTS

- *Record the gradient norm and the loss during training.*
- *Plot them on one figure.*

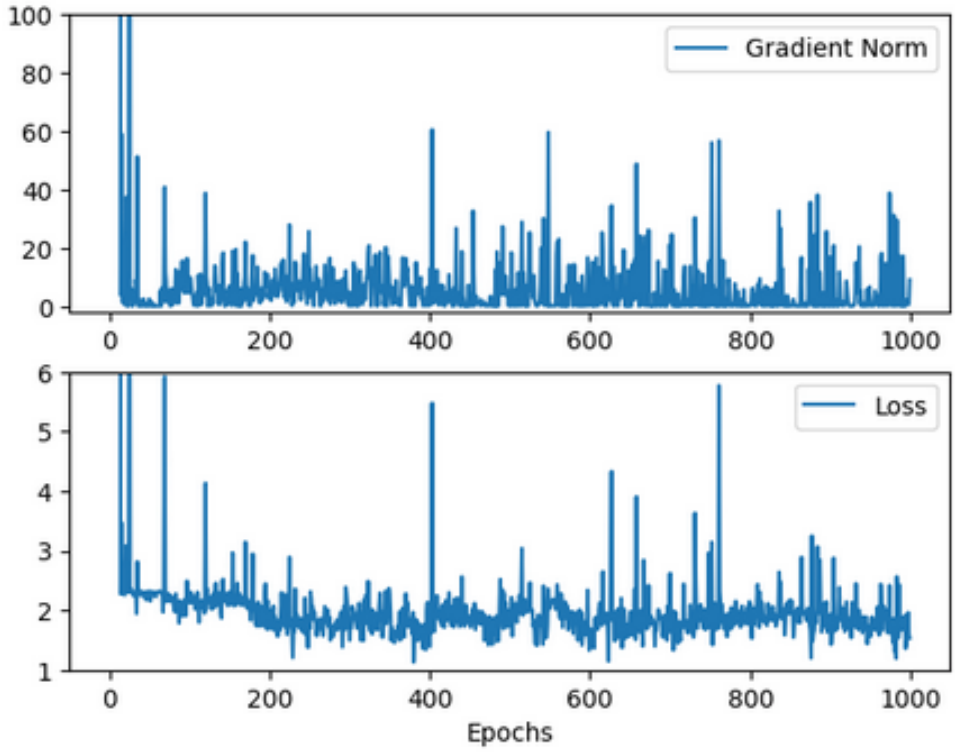
I decided to record the gradient norm for the MNIST database.

Here is the model that I used to record the gradient norm:

- An input layer of size 28 * 28 = 784
  - Each image is made of 28 rows and 28 columns (a matrix of 28x28) with a value between 0 and 255 that represents a black and white color scale (0 = black and 255 = white)
- One hidden layer of dimension (784, 128)
- One output layer with dimension (128, 10)
  - The output is a layer of 10 because each index (0 to 9) represents the possible labels that the image represents

- The model uses a ReLU activation function.
- I decided to use the *CrossEntropyLoss*() for the loss function
- Stochastic Gradient Descent (SGD) for the optimization algorithm
- A learning rate of 0.01
- A batch size of 32 samples
- 1000 epochs

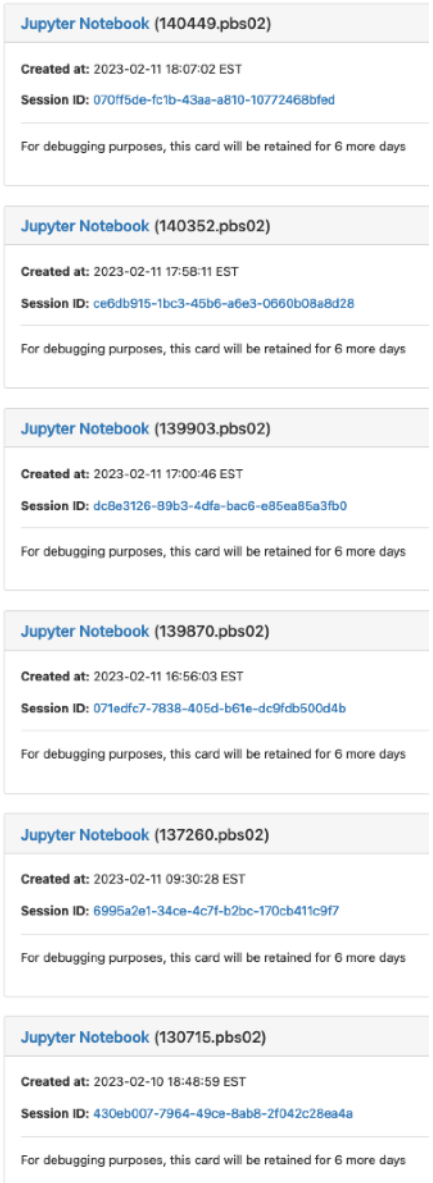
Here is the figure with the gradient norm and loss for the training process.



We can see that as the number of epochs increases the gradient norm decreases at a very slow rate during the first 200 iteration and later stays pretty consistent.

What happens when gradient is almost zero

* *This was an INSANE QUESTION*
* *COMPUTING THE HESSIAN IS A MESS!!!*
  * *From everything that I read ("Hessian based approaches aren't efficient for DNN" I think this is because when you calculate the Hessian it recomputes the model output and loss) and boy is it inefficient, every time I tried to compute the Hessian using the pytorch's stack overflow method (link 2 in your announcement) my Palmetto Instance crashed (58 cores and 62 gb of mem), which made it almost impossible to debug the code: just look how many instances crashed —> —> —>*



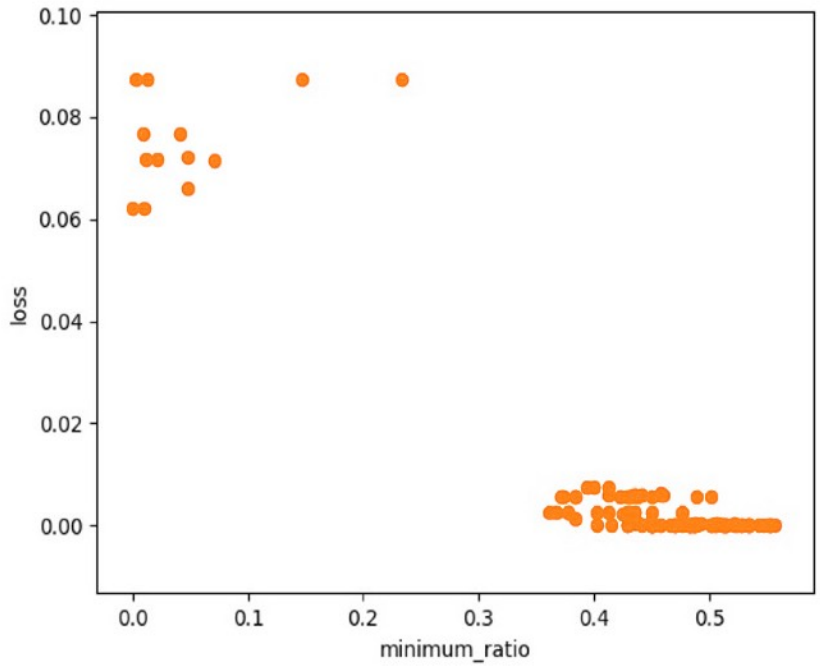
To compute the minimal ratio I used the Levenberg-Marquardt algorithm:

```
def LM(model,loss,n_iter=3):

    alpha=1e-3
    loss_hist=[]
    for i in range(n_iter):
        model.train()
        out=model(X).unsqueeze(1)
        loss_out=loss(out)
        prev_loss=loss_out.item()
        gradients=torch.autograd.grad(loss_out, model.parameters(), create_graph=True)

        model.eval()
        Hessian, g_vector=eval_hessian(gradients, model)

        dx=-1(alpha*torch.eye(Hessian.shape[-1]).cuda()+Hessian).inverse().mm(g_vector).detach()

        cnt=0
        model.zero_grad()

        for p in model.parameters():

            mm=torch.Tensor([p.shape]).tolist()[0]
            num=int(functools.reduce(lambda x,y:x*y,mm,1))
            p.requires_grad=False
            p+=dx[cnt:cnt+num,:].reshape(p.shape)
            cnt+=num
            p.requires_grad=True


        out=model(X).unsqueeze(1)
        loss_out=loss(out)

        if loss_out<prev_loss:
            print("Successful iteration")
            loss_hist.append(loss_out)
            alpha/=10
        else:
            print("Augmenting step size")
            alpha*=10
            cnt=0
            for p in model.parameters():

                mm=torch.Tensor([p.shape]).tolist()[0]
                num=int(functools.reduce(lambda x,y:x*y,mm,1))
                p.requires_grad=False
                p-=dx[cnt:cnt+num,:].reshape(p.shape)
                cnt+=num
                p.requires_grad=True

    return loss_hist
```

https://github.com/David-FR/8430_HMW1.git

Finally we know stat the minimal ratio is the proportion of eigenvalues that are greater than zero.

Here is a visual representation of the minimal ratio after training the model 100 times:



1-3 Generalization

Can network fit random labels?

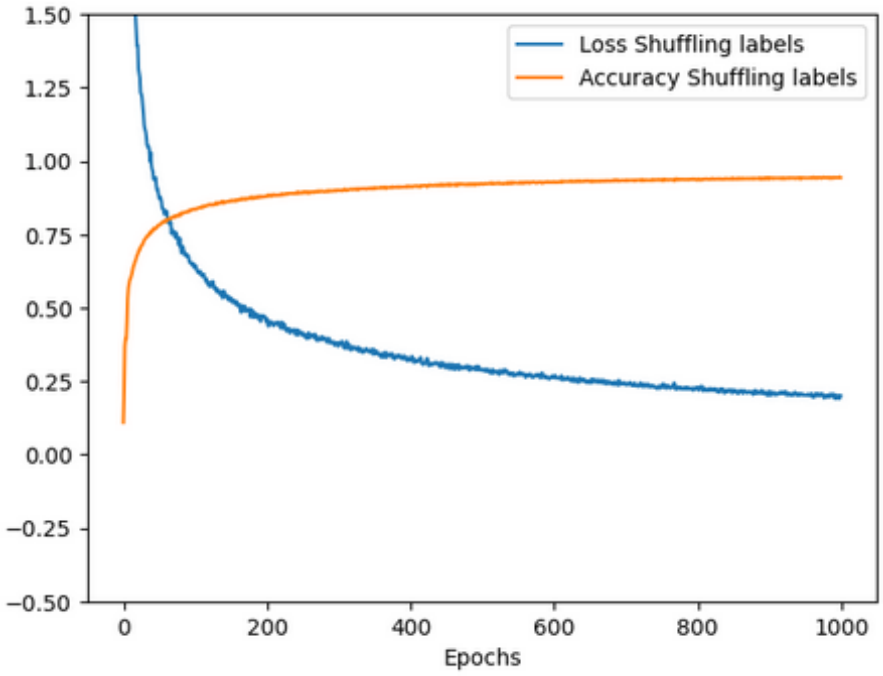
I decided to train a model to see if the model could predict the labels of the MNIST dataset:
MODEL

- An input layer of size 28 * 28 = 784
  - Each image is made of 28 rows and 28 columns (a matrix of 28x28) with a value between 0 and 255 that represents a black and white color scale (0 = black and 255 = white)
- One hidden layer of dimension (784, 128)
- One output layer with dimension (128, 10)
  - The output is a layer of 10 because each index (0 to 9) represents the possible labels that the image represents

- The model uses a ReLU activation function.
- I decided to use the *CrossEntropyLoss*() for the loss function
- Stochastic Gradient Descent for the optimization algorithm
- A learning rate of 0.001
- 3000 epochs

Before training I randomly shuffle the training set:

```
for epoch in range(epochs):
  # SHUFFLE LABELS BEFORE TRAINING
  shuffled_sample_index = np.random.randint(0, X_train.shape[0], size=(len(X_train)))

  X = X_train[shuffled_sample_index]
  X = torch.tensor(X.reshape((-1, 28*28))).float()
  Y = torch.tensor(Y_train[shuffled_sample_index])
```



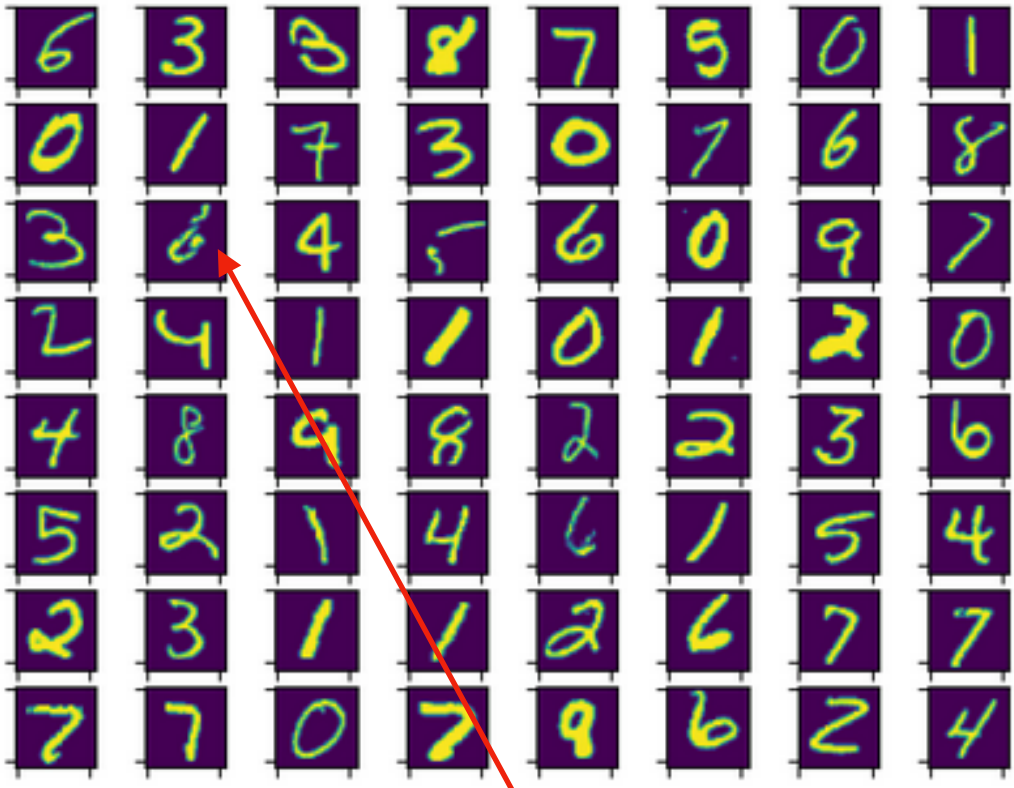
** This was a confusing question (like most of them!!!) the question says "try to fit these images" how do we get THOSE EXACT IMAGES FROM DB (DO I NEED TO LOOK FOR THOSE NUMBERS!!?????!!!!)???*

** Since we have the the MNIST database loaded why not try with random images from the test set??? That is a far better test than 64 arbitrary images anyways.*

Instead of testing my model with an arbitrary number of images I select 64 (could be any number) images from the test set and see how well my model predicts the values of those images.

Here are the results:

```
Images to test:
<Figure size 100x100 with 0 Axes>
```



```
Predictions:
[[6 3 5 8 7 3 0 1]
 [0 1 7 3 0 7 6 8]
 [3 2 4 5 6 0 9 7]
 [6 4 1 1 0 1 2 0]
 [4 8 9 8 6 2 3 6]
 [5 2 1 4 6 1 5 4]
 [2 3 1 1 2 6 7 7]
 [7 7 0 7 9 6 2 4]]
```

We can see that the model predicts the number in the labels pretty well.

If we examine the classification “errors” we can see that they are actually pretty acceptable.

For example, even though the model predicts incorrectly this image as a 2 even to the human eye (me) it is difficult to classify correctly (is it a 6 or 0??)

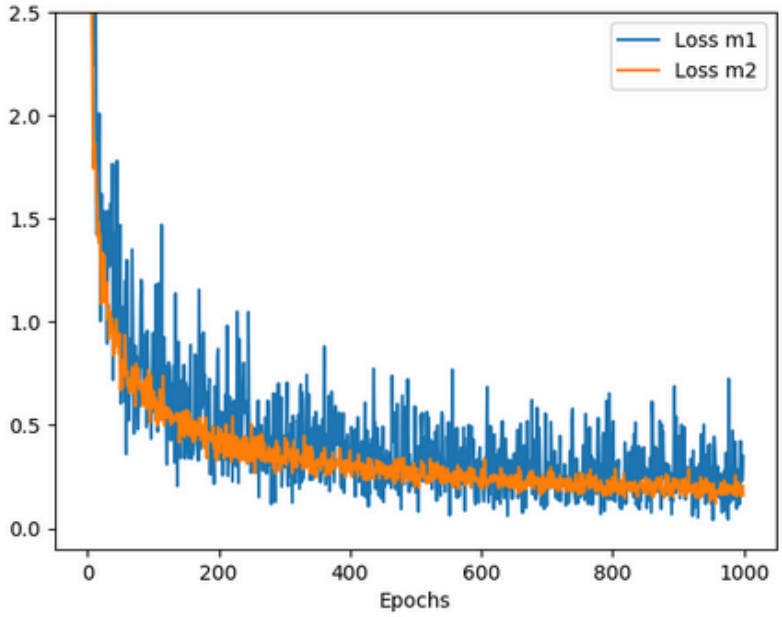
Number of parameters Vs Generalization Part 1

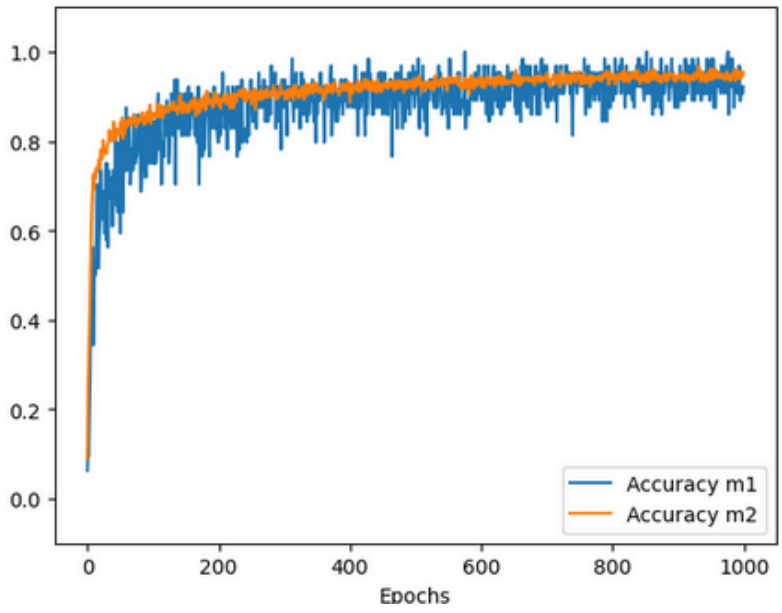
REQUIREMENTS

- *Train two models (m1 and m2) with different training approach. (e.g. batch size 64 and 1024)*
- *Record the loss and accuracy of the model which is linear interpolation between m1 and m2.*

Here are the batch sizes that I decided to use for the models:
- M1 = 64
- M2 = 1024

Here are the comparison of both models:





Even though both models have a good performance we can see that Model 2 (using a batch size of 1024 elements) gives us less variation in both the loss and the accuracy of the model.

Increasing the batch size further does not gives us considerable improvements and only slows down the training process.

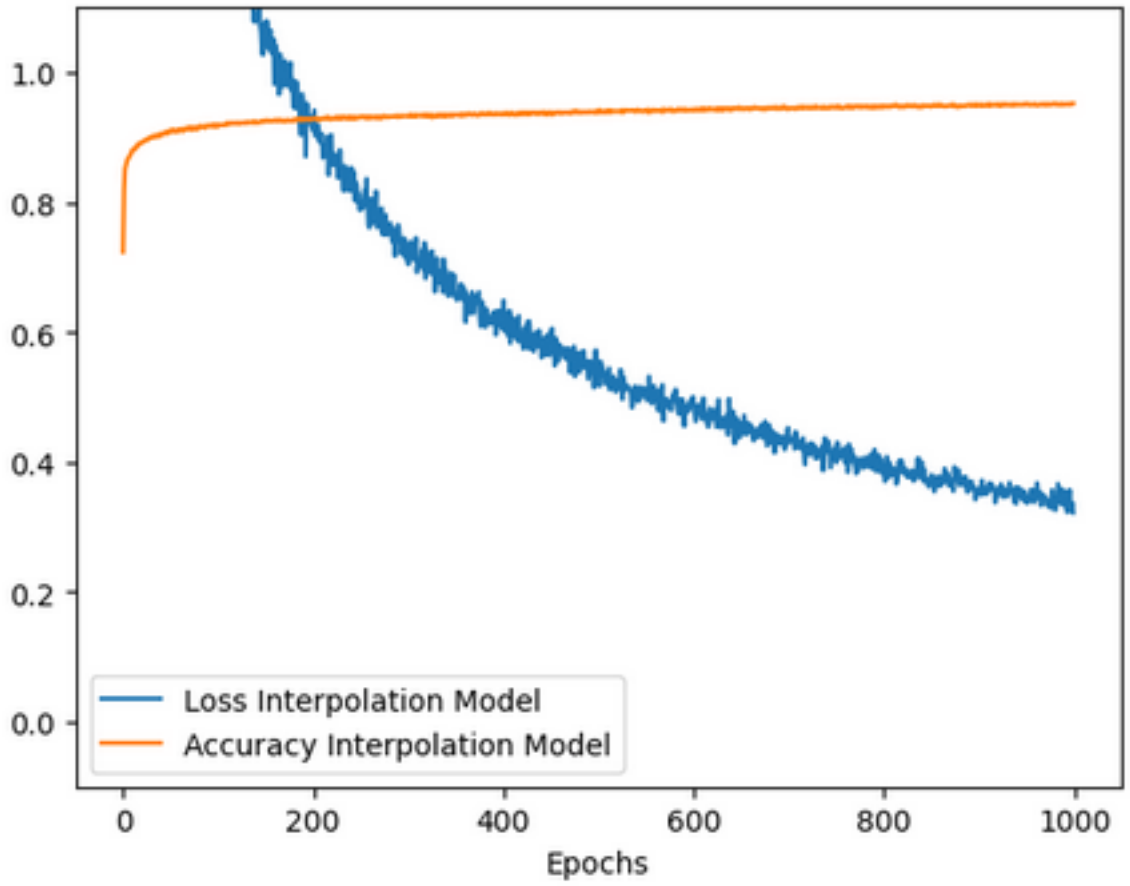
Flatness Vs Generalization Part 1

To compare the cross_entropy and the accuracy as the value of α changes for the model that is a linear interpolation between m1 and m2:

$$\boldsymbol{\theta}_{\boldsymbol{\alpha}} = (1 - \alpha)\,\boldsymbol{\theta}_{\mathbf{1}} + \alpha\boldsymbol{\theta}_{\mathbf{2}}$$

Here is how the model looks in terms of loss and accuracy with an interpolation ratio of 0.5 and a learning rate of 0.01

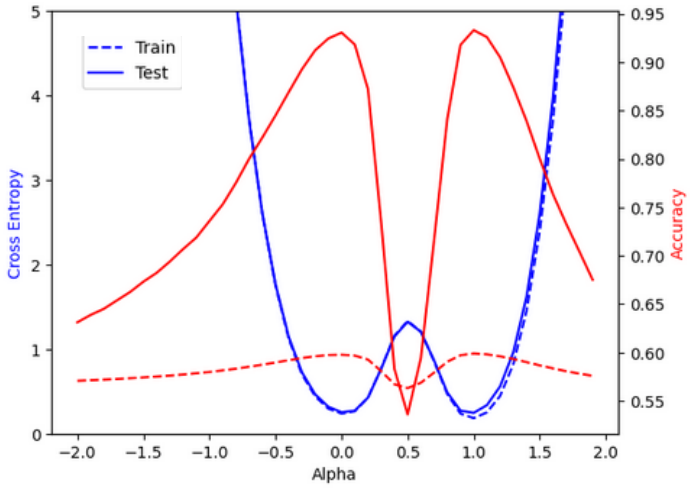
Note that the three models (m1, m2 and the linear interpolation model) use 4 parameters.



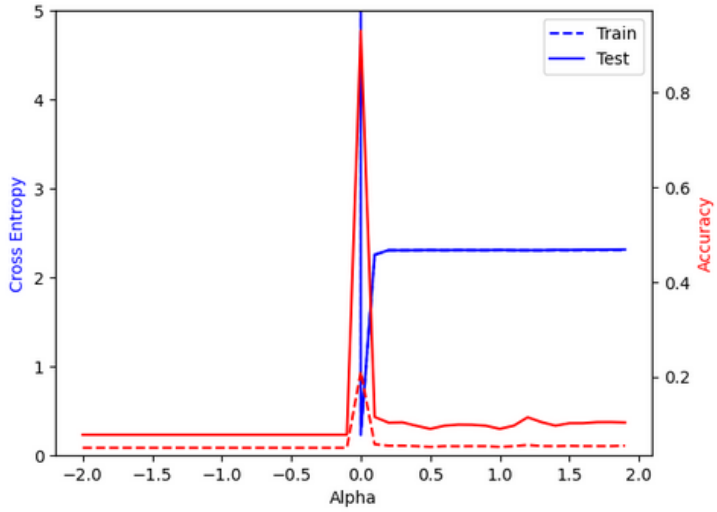
Since the model is a linear interpolation of two very good models it will still be very accurate and coverage (in terms of accuracy) after just 500 interaction. Notice that even thought the loss keeps decreasing after the 500th iteration, the accuracy no longer increases.

I decided to test 40 values (or 40 models) of alpha (or interpolation ratio) form -2 to 2. Here are the results:

Next, here are the results using a learning rate of 0.01



Finally, here are the results using a learning rate of 0.1



We can see that with a learning rate of 0.01 the accuracy is the lowest when the interpolation ratio between the two models is 50% however, when the learning rate increases to 0.1 the accuracy is the highest when the ratio is 50%.

Flatness Vs Generalization Part 2

To compare the sensitivity of the model I decided to calculate and compare the sensitivity of 6 models with different batch sizes. Here are the different batch sizes that I decided to use:

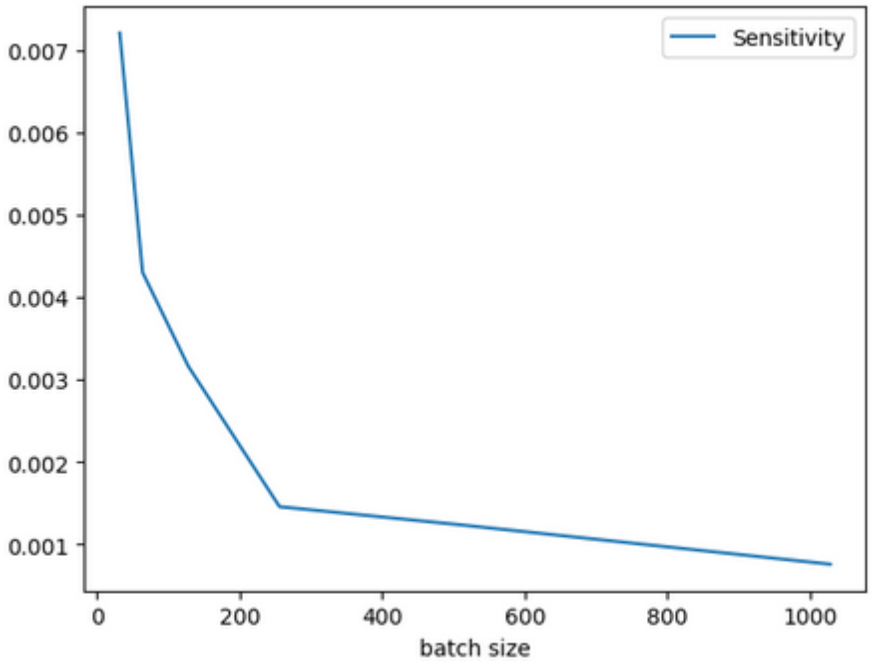
```
training_approaches = [32, 64, 128, 256, 512, 1028]
```

Here is the model that I used:

- An input layer of size 28 * 28 = 784
  - Each image is made of 28 rows and 28 columns (a matrix of 28x28) with a value between 0 and 255 that represents a black and white color scale (0 = black and 255 = white)
- One hidden layer of dimension (784, 128)
- One output layer with dimension (128, 10)
  - The output is a layer of 10 because each index (0 to 9) represents the possible labels that the image represents

- The model uses a ReLU activation function.
- I decided to use the *CrossEntropyLoss*() for the loss function
- Stochastic Gradient Descent for the optimization algorithm
- A learning rate of 0.001
- 500 epochs (as opposed to 3000 to speed up the training process, this process need to run 6 times)

Here are the sensitivities that my model recorded:



We can clearly see that the sensitivity (or the forbenius norm of gradients “forb”) decreases as the batch size increases. This is consistent with what we expected. With more values (or bigger batch size) the model converges and thus less sensitive.