Assignment 1

Overview

Asif Maidan, 209009612

Yariv Tachnai, 208513671

NN Architecture

The network consists of L fully connected layers, each Layer but the last is defined as:

$$L = tanh(WX + b)$$

The last layer is defined as:

$$L = softmax(X^T W + b)$$

where the number of layers L and the width of the hidden layers are user parameters.

ResNN Architecture

The residual network consists of L-1 reidual layers defined as

$$L = X + W_2(tanh(W_1X + b))$$

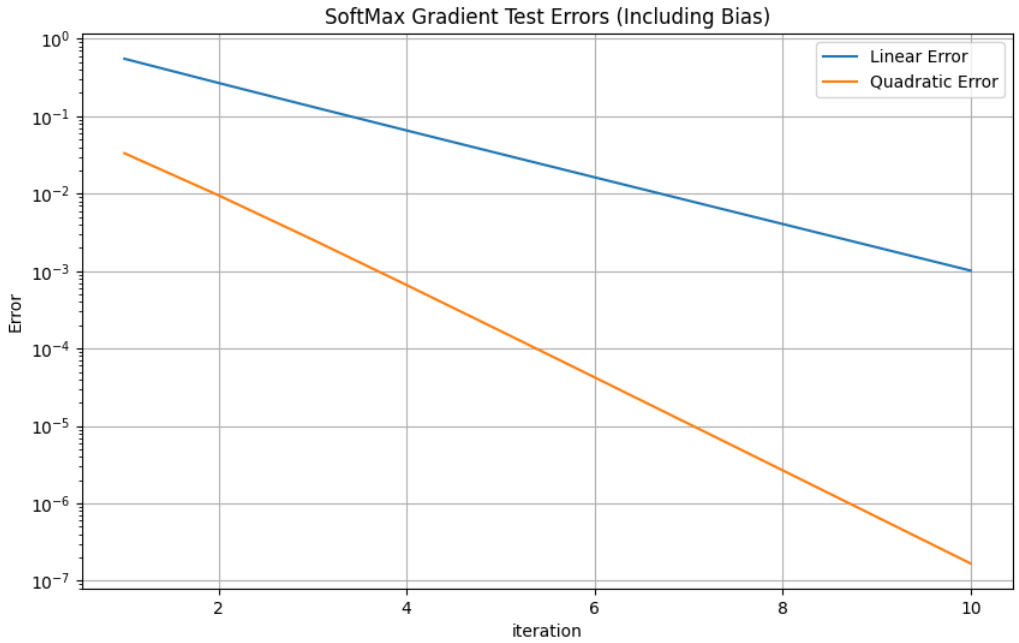
And a final softmax layer similar to the regular NN

Part I

▼ Loss Function and Gradient

We wrote the functions for the computation of the softmax, its loss and the gradient of its loss, and used the grad test as shown in the class notes to check for mistakes.

The graph verifies the accuracy of the gradient calculation, showing that the linear error decreases at a rate of approximately 0.5, while the quadratic error decreases at a rate of approximately 0.25. Given that our epsilon sequence follows 0.5^n, the decrease rate of the linear error can be characterized as $O(\epsilon)$, and that of the quadratic error as $O(\epsilon^2)$



▼ SGD Optimizer

We examined the Stochastic Gradient Descent (SGD) algorithm as outlined in the optimization booklet and implemented it with slight modifications

Algorithm: Stochastic Gradient Descent

 # Input: Objective: $F(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} f_i(\mathbf{w})$ *to minimize.*

for $k = 1, ..., maxIter$ **do**

#Loop over epochs

Divide the data indices $\{1, ..., m\}$ into random mini-batches (mb) $\{\mathcal{S}_j\}_{j=1}^{\#mb}$.

Denote $\mathbf{w}^{(k,1)} = \mathbf{w}^{(k)}$.

for $j = 1, ..., \#mb$ **do**

Compute the mini-batch gradient: $\mathbf{g}^{(k,j)} = \frac{1}{|\mathcal{S}_j|}\sum_{i\in\mathcal{S}_j} \nabla f_i(\mathbf{w}^{(k,j)})$.

Choose a step-length $\alpha^{(k,j)}$ (a "learning-rate")

Apply a step:

$$\mathbf{w}^{(k,j+1)} \leftarrow \mathbf{w}^{(k,j)} + \alpha^{(k,j)}\mathbf{g}^{(k,j)}.$$

end

Denote $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k,\#mb+1)}$ *# may be replaced by averaging over j*

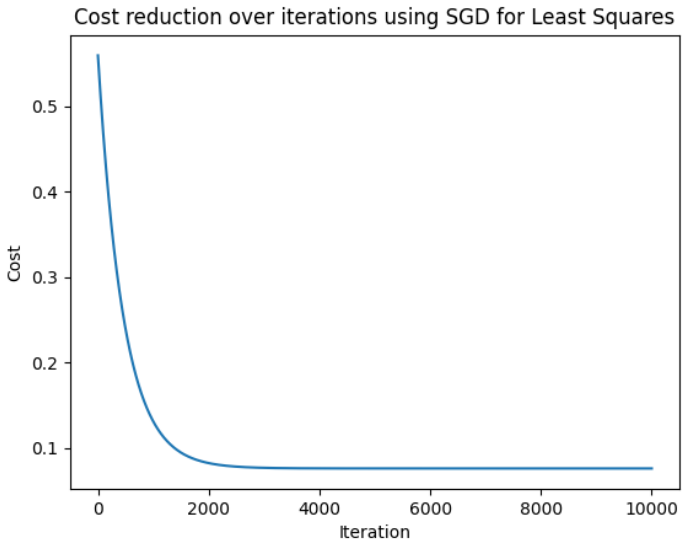
 # Check convergence by some criterion.

end

Return $\mathbf{w}^{(k+1)}$ as the solution. *# may be replaced by some averaging*

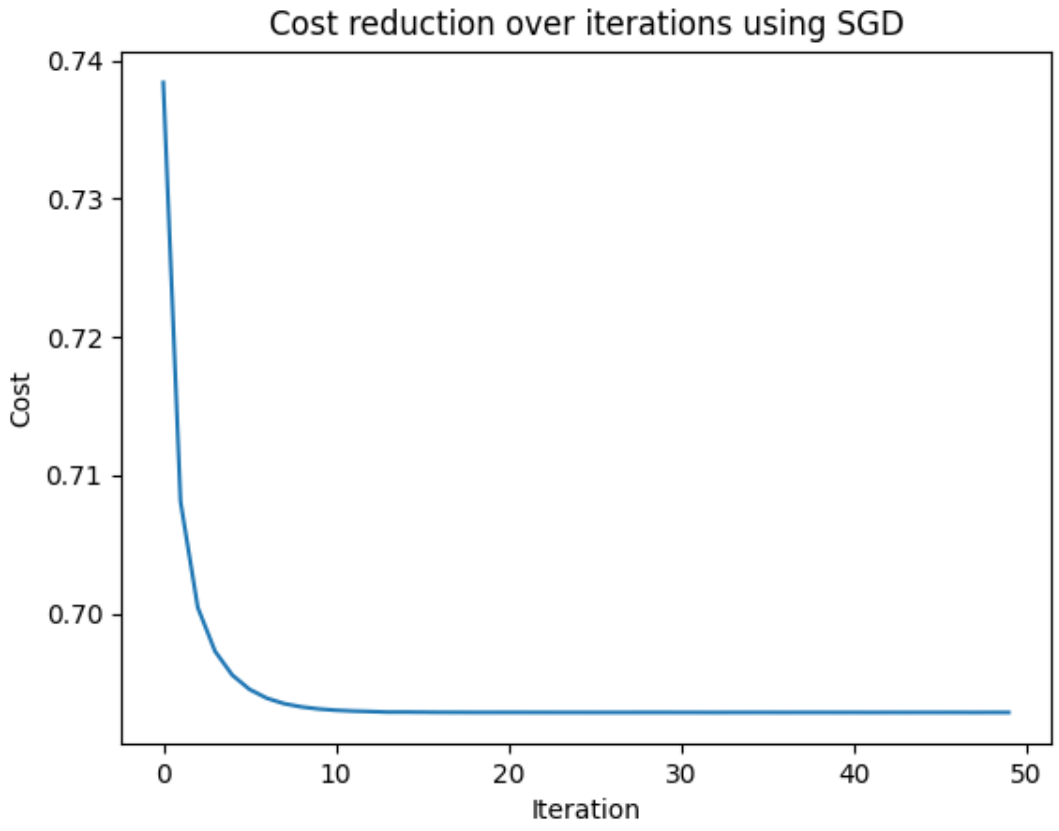
Algorithm 4: Stochastic Gradient Descent

As we can see in the adjacent graph, over the course of 10000 iterations, the cost function decreases into a plateau at around 0.04 , a desirable outcome that confirms the effectiveness of the optimizer.



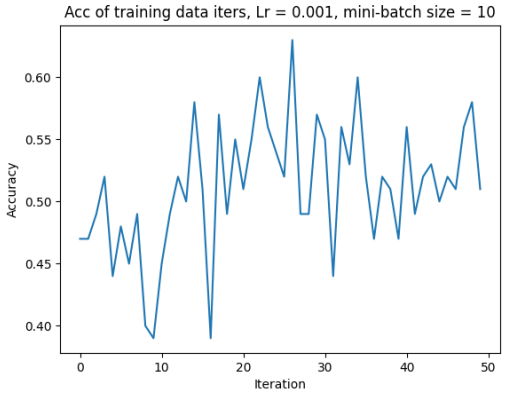
▼ Softmax function Minimization

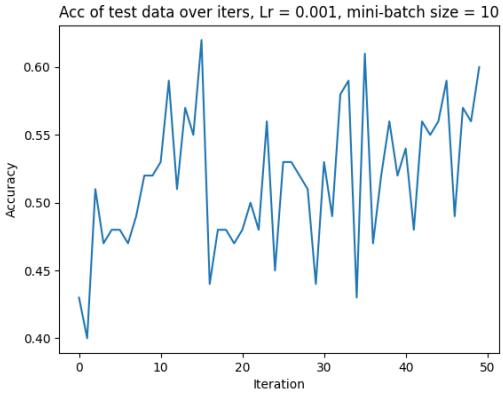
We applied our variant of the Stochastic Gradient Descent algorithm to optimize the previously developed softmax function. The graph below demonstrates that the cost decreases and stabilizes at a plateau, approximately at 0.69:



This suggests that although the gradients are becoming almost negligible no actual learning occurs,

We see that when we plot the accuracy of the model over the training data and the test data:





Adjusting the learning rate and mini-batch size did not result in a significant change in accuracy, which remained around 50%. Given that the dataset used was SweetRollData, which contains two classes, this level of accuracy is essentially equivalent to random guessing.

however when we increased the learning rate above 0.5, we found out that the cost function fluctuates and doesn’t get reduced as much.

Part II

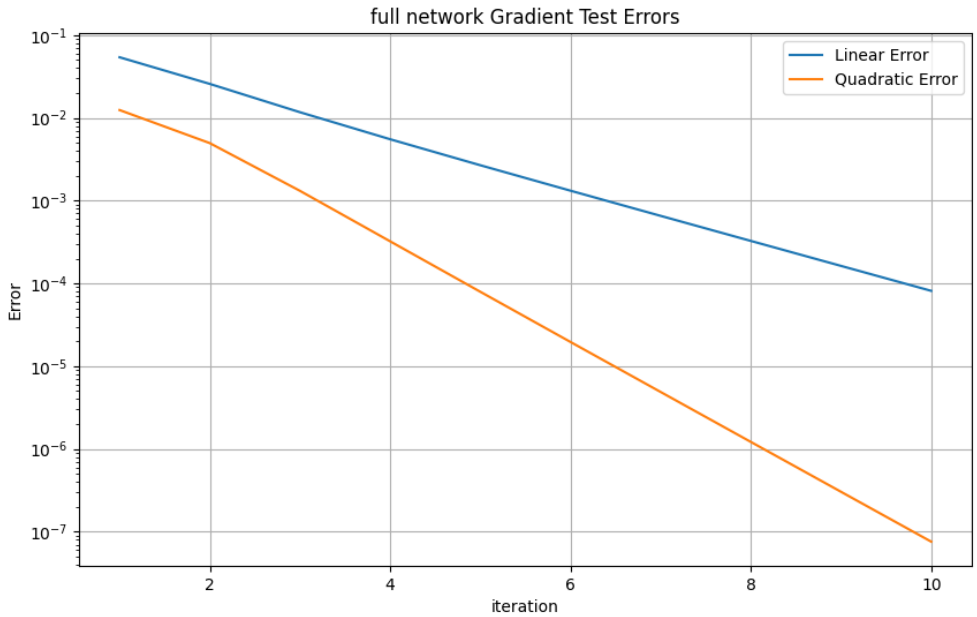
▼ Standard Neural Network

We wrote the code for the neural network, the initialization, the forward pass and the backword pass.

we initialized weights and biases for each layer

1. The first layer the weights dimensions were: input_features X Hidden_layer_size, they were randomly generated and normalized
   biases are: Hidden_layer_size X 1, initiated with zeros
2. The hidden layer's weights dimensions were: Hidden_layer_size X Hidden_layer_size normalized, they were randomly generated and normalized
   biases are: Hidden_layer_size X 1, initiated with zeros
3. The last layer's weights dimensions were: Hidden_layer_size X Classes normalized, they were randomly generated and normalized
   biases are: 1 X Classes, initiated with zeros

We performed a gradient check over the entire network:

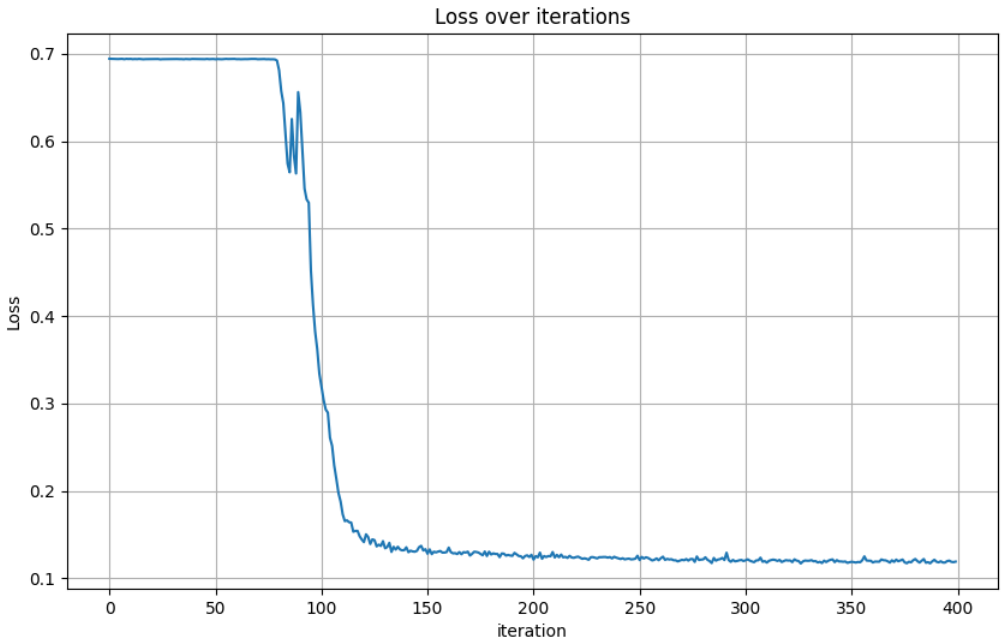


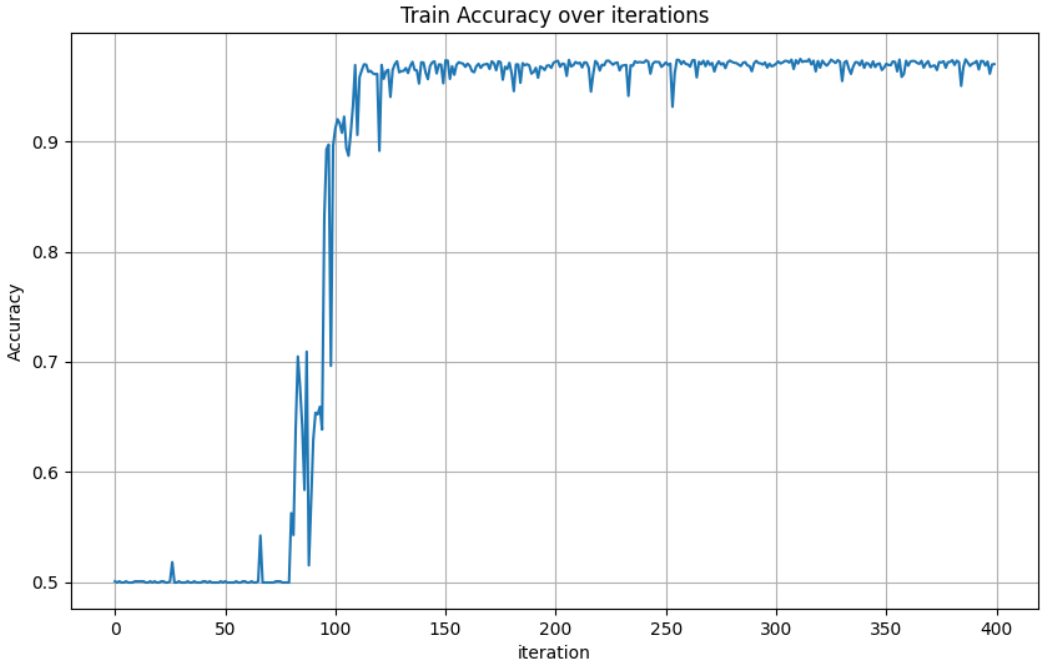
Once more, employing epsilons in the form of 0.5^n we observed the desired outcome, which validates the accuracy of the gradient calculations.

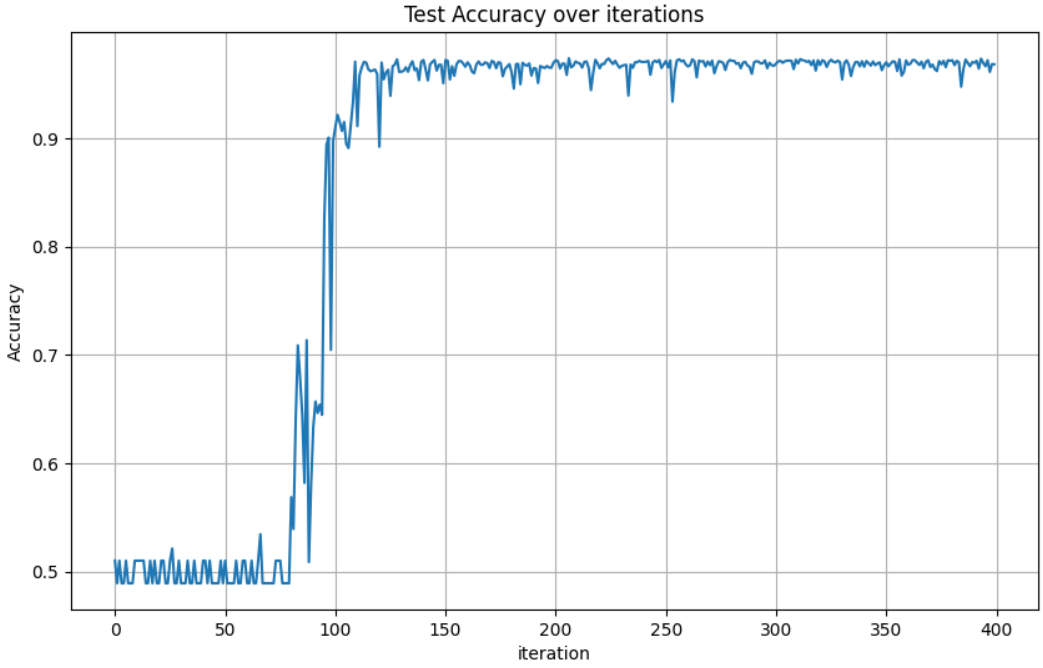
Next, we tested the combination of our NN with the optimizer we built earlier (we modified the optimizer so it would integrate well with the network interface).

After experimenting with various hyperparameters, we identified our optimal settings using the SwissRollData dataset (with other datasets yielding similar outcomes). The chosen hyperparameters were:

- layer width = 9
- layer number = 5
- learning rate = 0.2
- iterations = 400
- mini batch size = 50




Train Accuracy over iterations
Accuracy
iteration
0.5
0.6
0.7
0.8
0.9
0
50
100
150
200
250
300
350
400


Test Accuracy over iterations
Accuracy
iteration
0.5
0.6
0.7
0.8
0.9
0
50
100
150
200
250
300
350
400

The data indicates a reduction in loss to approximately 0.12 across 400 iterations, with little change observed post iteration 150, while accuracy increases to roughly 97% for both training and testing datasets. Notably, a spike is observed in all graphs between epochs 75 and 125, suggesting that the bulk of learning transpires within this epoch range.

we conducted a search for the best hyper-parameters and we want to share some snippets of the resulting excel file:

The best results:

	Hyperparameters- layer width, layer number, Lr, epochs, batch size	Loss	Accuracy
2	(9, 5, 0.21000000000000002, 400, 50)	0.10846597	97.36
3	(9, 5, 0.31000000000000005, 200, 30)	0.113840055	97.36
4	(6, 5, 0.01, 400, 10)	0.117243744	97.34
5	(9, 4, 0.51, 300, 110)	0.114917355	97.32
6	(9, 5, 0.11, 300, 50)	0.112618232	97.3

Some results in the middle:

1694	(3, 3, 0.21000000000000002, 400, 170)	0.49818479982453157	72.34
1695	(3, 3, 0.51, 400, 70)	0.484818413	72.34
1696	(9, 5, 0.7100000000000001, 100, 10)	0.5686705080426819	72.32
1697	(9, 3, 0.21000000000000002, 200, 110)	0.5220241370245221	72.3
1698	(3, 3, 0.11, 400, 90)	0.48715559561055655	72.28
1699	(3, 3, 0.7100000000000001, 200, 70)	0.5070555931890487	72.28
1700	(3, 4, 0.21000000000000002, 400, 70)	0.4724288004776738	72.24000000000001
1701	(9, 4, 0.51, 100, 130)	0.4822744483620528	72.24000000000001
1702	(3, 4, 0.6100000000000001, 400, 190)	0.5298726483254795	72.22

And the worst results:

3423	(3, 3, 0.01, 100, 130)	0.6925688645061481	51.04
3424	(3, 3, 0.01, 100, 190)	0.6925368066146774	51.04
3425	(3, 4, 0.01, 100, 70)	0.6927297025588041	51.04
3426	(3, 4, 0.01, 100, 150)	0.692704508	51.04
3427	(3, 4, 0.01, 100, 190)	0.6927324014301636	51.04
3428	(3, 4, 0.01, 200, 190)	0.6926312915646792	51.04
3429	(3, 4, 0.41000000000000003, 300, 90)	0.7012375449413646	51.04
3430	(3, 4, 0.51, 200, 190)	0.69305267	51.04
3431	(3, 4, 0.6100000000000001, 300, 70)	0.6941291945799066	51.04

The observed outcomes underscore the critical role of selecting appropriate hyperparameters, as evidenced by the variability in accuracy rates. It is observed that increasing the number of layers and their width generally correlates with improved performance. Additionally, it is noted that certain hyperparameter configurations yield results comparable to random guessing.

Concluding our experimentation and analysis, we successfully trained our neural network to classify three distinct datasets with acceptable accuracy. This underscores the critical influence of initial parameter selection and hyperparameter optimization on the network's performance. Our findings reveal that the manner in which weights are initialized can significantly impact the learning process and outcome. Specifically, initializing weights to ones, rather than adopting a randomized approach, led to suboptimal results, highlighting the delicate balance required in parameter initialization for effective network training.

Furthermore, our search for the optimal hyperparameters further illustrates the pivotal role these settings play in neural network efficiency. The process of identifying the right combination of layer width, number of layers, learning rate, iteration count, and mini-batch size was instrumental in maximizing the network's accuracy.

Our experiments demonstrate that the effectiveness of a neural network is not solely dependent on its architectural design but is also significantly influenced by the initial conditions and training configurations. The right initialization strategy, coupled with a well-considered selection of hyperparameters, can dramatically improve a network's ability to classify data accurately.

▼ Residual Neural Network

We wrote the code for the neural network, the initialization, the forward pass and the backword pass.

we initialized weights and biases for each layer:

1. In each hidden layer, W_1 and W_2 have dimensions of: input_features X input_features, they were randomly generated and normalized
   the biases have dimensions of: input_features X 1, initiated with zeros
2. in the last layer W has the dimensions: input_features X Classes and are normalized, they were randomly generated and normalized
   the biases are: 1 X Classes initiated with zeros

The gradients of the Residual network

We structured the Jacobian transpose multiplied by vector to mirror the form used in the standard network notes.

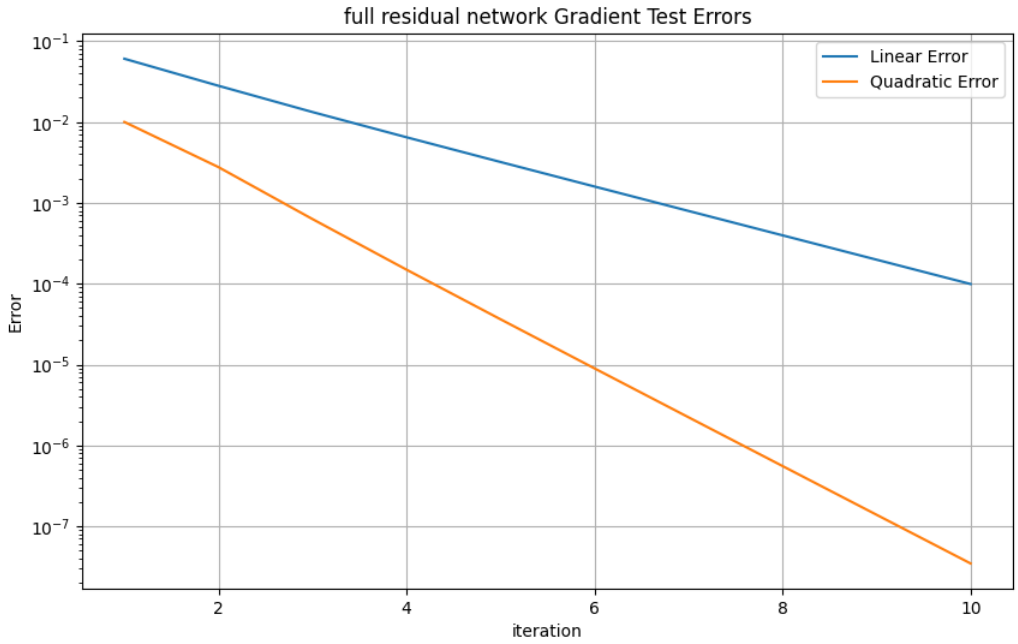
$$(\frac{\partial Loss}{\partial W_1})^T v = (\sigma'(W_1 X + b) \odot W_2^T v) X^T$$

$$(\frac{\partial Loss}{\partial W_2})^T v = v\sigma(W_1 X + b)$$

$$(\frac{\partial Loss}{\partial b})^T v = (\sigma'(W_1 X + b) \odot W_2^T v).sumOverCollums()$$

$$(\frac{\partial Loss}{\partial X})^T v = v + (W_1^T(\sigma'(W_1 X + b) \odot W_2^T v))$$

We performed a gradient check over the entire residual network:



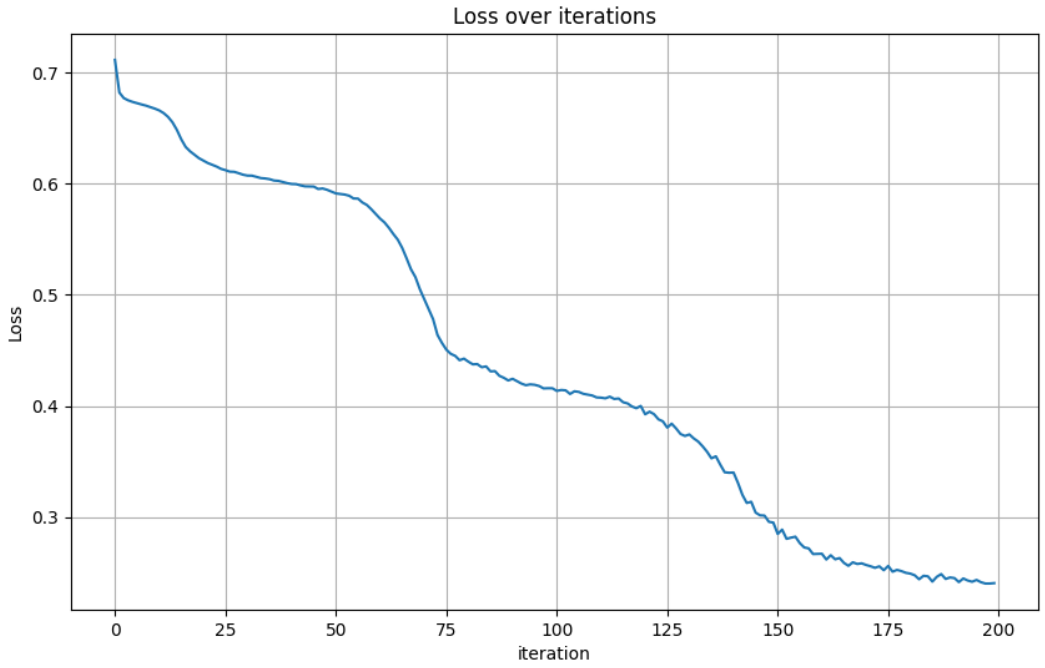
We receive similar results to both the regular NN and the first Gradient test we performed, again validating the gradient calculations

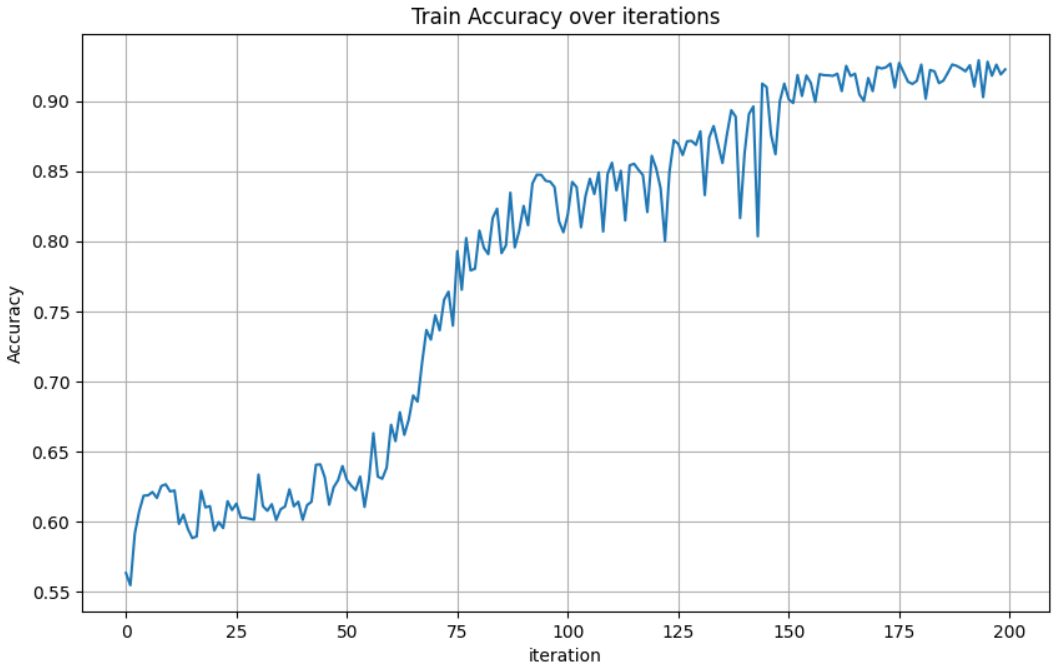
We also tested the residual network with our SGD variant to obtain a classifying network.

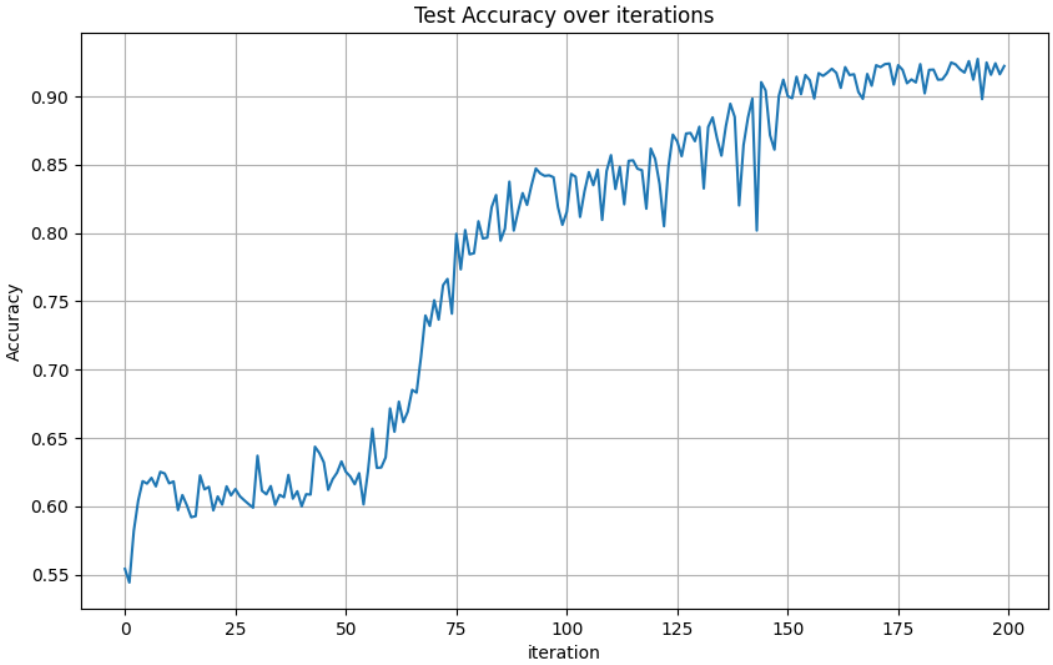
After conducting a search for optimal hyper-parameters yet again, we ran the net with these parameters:

- layer number = 12
- learning rate = 0.004
- iteration_number = 200
- mini batch size = 40

Here are the results:







Once again, we observe a decrease in loss that eventually plateaus at around 0.10 (though the plateau is not depicted in the graph), Additionally, the graph exhibits a steadier progression compared to the loss reduction observed in a standard neural network, suggesting that the majority of learning occurs throughout the entire duration of the training period instead of a few dozens of epochs. Concurrently with the loss, the accuracy for both training and test data increases, stabilizing at approximately 96%. Interestingly, outcomes of this caliber were only attained with a significantly increased number of layers (three times more than the standard network configuration), whereas configurations with fewer layers yielded impractical results.

Next we want to explore some of the results we received over different hyper-parameter combinations:

The best results:

	Hyperparameters - layers, Lr, epochs, batch size	Loss	Accuracy
2	(10, 0.011, 200, 30)	0.14649593935847793	96.28
3	(12, 0.007, 200, 20)	0.1798895496449712	95.32000000000001
4	(13, 0.013000000000000001, 200, 20)	0.17276330152440408	95.1
5	(12, 0.013000000000000001, 200, 30)	0.18247504844768345	95.08
6	(13, 0.008, 200, 20)	0.1730700972081897	94.94
7	(13, 0.009000000000000001, 200, 30)	0.1987241136109619	94.89999999999999

Some results from the middle:

1520	(9, 0.012, 150, 30)	0.5677784483056273	66.53999999999999
1521	(8, 0.005, 150, 20)	0.5985048509752747	66.53999999999999
1522	(8, 0.011, 200, 20)	0.6240637257930001	66.52
1523	(13, 0.008, 200, 40)	0.6281242054726729	66.52
1524	(6, 0.011, 150, 30)	0.5904946038458115	66.5
1525	(6, 0.011, 200, 40)	0.6084075771033479	66.47999999999999

Bad results:

4025	(13, 0.001, 200, 20)	0.6383555348242228	51.559999999999995
4026	(11, 0.004, 100, 40)	0.6392170533681139	51.5
4027	(8, 0.003, 200, 30)	0.6928955429381618	51.06
4028	(9, 0.001, 100, 30)	0.692925021	51.04
4029	(8, 0.009, 100, 30)	0.6928766083242938	51.04
4030	(8, 0.001, 150, 40)	0.692878327	51.04

We receive another assurance to the relevance and importance of the hyper-parameter selection, fewer layers and iterations generally corelates to poorer results.

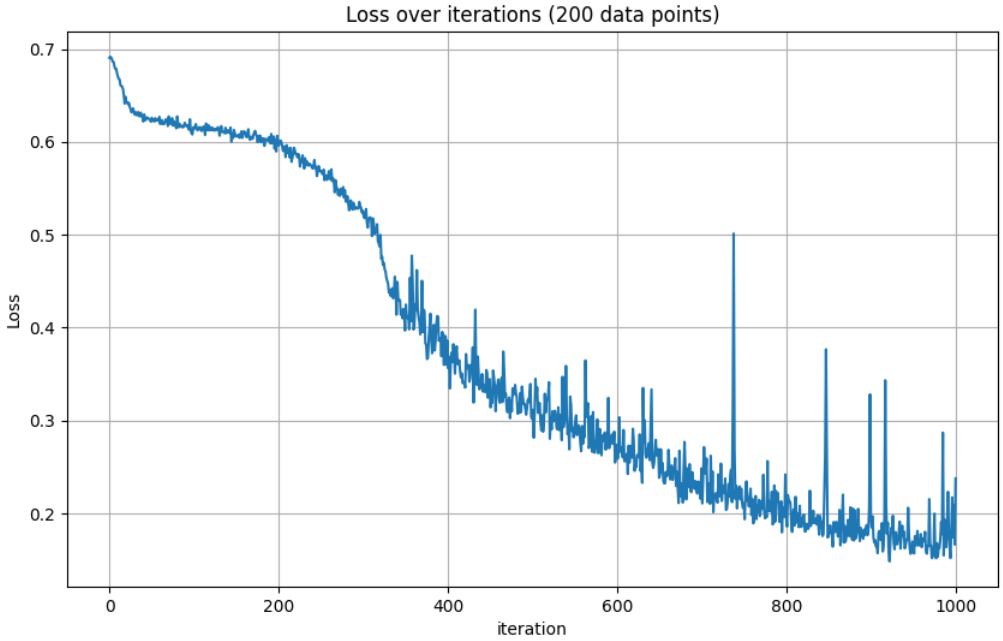
▼ Reduced Data Experiment

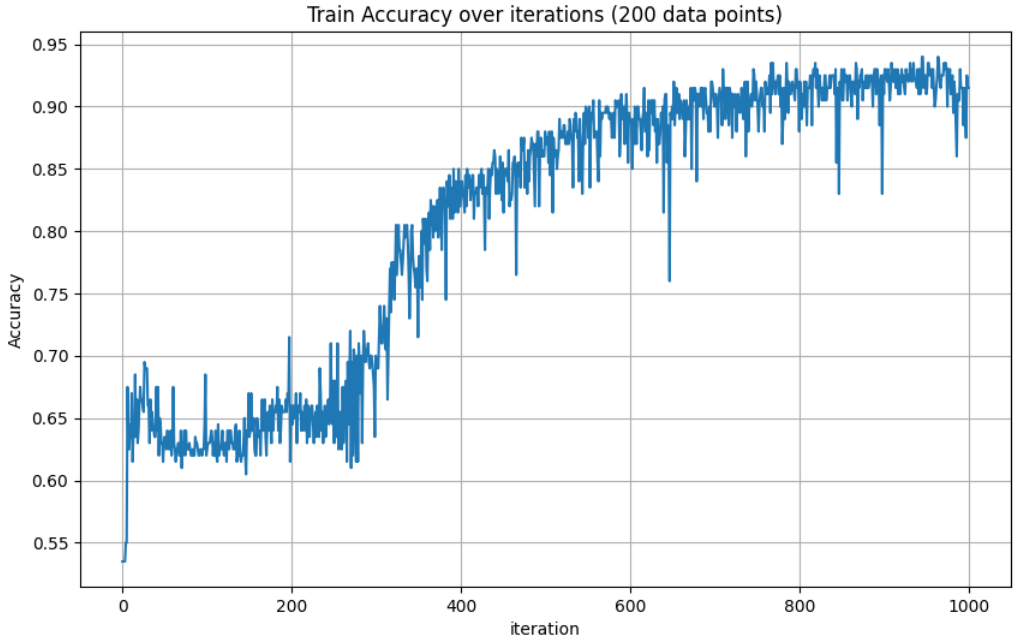
Finally, we aim to evaluate our network using a random subsample of just 200 data points to understand the implications, specifically, how the quantity of data available influences the network's performance outcome.

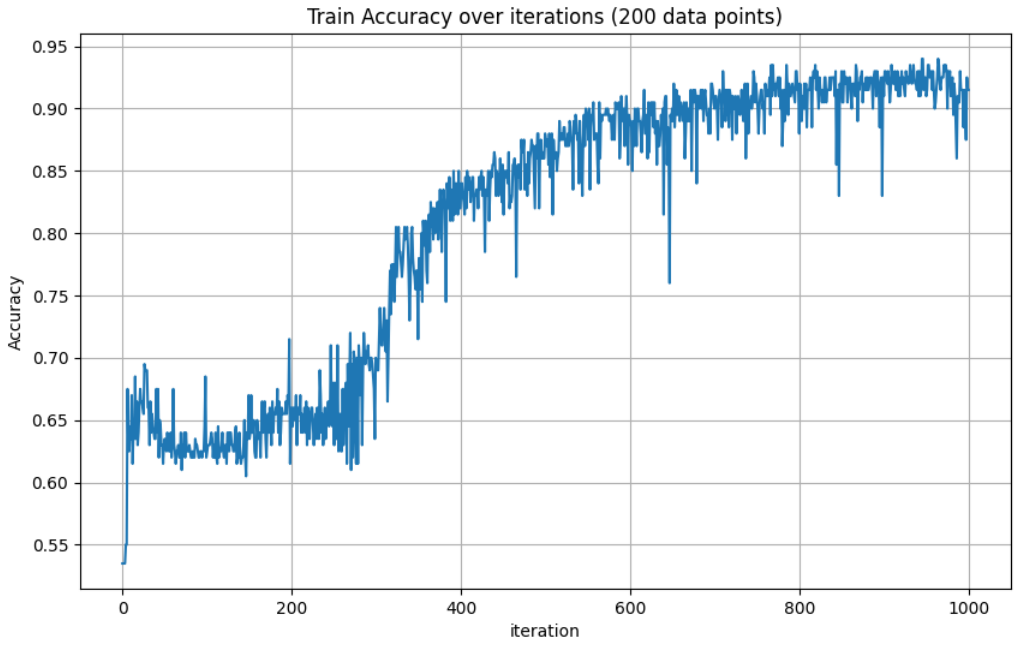
Initially, we applied the optimal hyper-parameter settings identified from the full dataset. However, this approach yielded minimal improvement in the network's accuracy, with performance closely mirroring that of random guesses.

In response, we significantly augmented the number of iterations, multiplying the original count by five to reach a total of 1000 iterations. The outcomes,

detailed below, were unexpectedly favorable.







The loss graph, despite some fluctuations, demonstrates an overall consistent decrease, stabilizing at approximately 0.2. Likewise, there is a noticeable improvement in the network's accuracy across both the training and test datasets.

These findings indicate that although the reduced data volume led to inconsistent enhancements, there is a general trend of improvement over iterations. This may suggest that while the quantity of data impacts the pace and magnitude of learning, given sufficient time, the training is likely to converge towards similar outcomes.