Introduction to Deep Learning

Jyothis Gireesan Mini, Michail Athanasios Kalligeris Skentzos, and Priyanjali Goel

Student Ids: 3777103, 4398831, 4406559
Contribution: Task 1: Jyothis, Task 2: Priyanjali, Task 3: Thanos

1 Introduction

Based on the simplified version of the MNIST data set, a collection of 1707 training digits and 1000 test sets represented by 256 pixel values. The objective is to develop several algorithms for classifying images of handwritten digits and designing our own neural network from scratch for the XOR problem.

2 Distance Based Classifiers

2.1 Class Centers

For the first task we were supposed to experiment with dimensionality reduction techniques to visualize our data and develop a simple algorithm to classify digits (0 through 9) that is based on the distance between the centers of the clouds of digits.

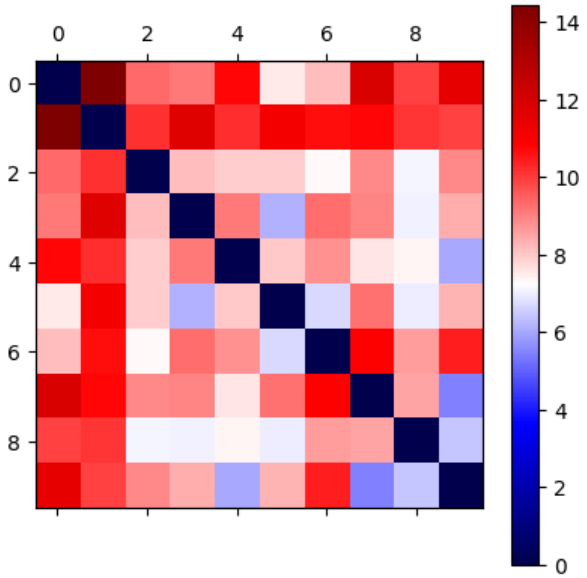

Figure 1: Class Distance Matrix

More specifically we had to calculate the average distances from each center for each given digit and visualise them to gain insight about the dataset. We started by calculating the cloud centers C_d for each digit [by taking the mean of all images labelled as digit d]. We evaluated how close the centers were from each other based on their Euclidean distances. This gave us insight into how the centers for digits like {4, 9} and {7, 9} are very close to each other and could be misidentified and it was surprising to see digits like {2, 6} and {4, 8} could be misidentified as well.

2.2 Dimensionality Reduction

At this point we began to have a basic understanding on what to expect, and wanted to see how well the dimensionality reduction algorithms were able to classify them. PCA, U-MAP, T-SNE were the three algorithms used for this. Please refer the images below 2, 3, and 4. PCA performed as expected with only seperating out 1 and 0 and all others are blended with at least 2 other numbers. Although it was able to classify a single digit together to a point, there was a lot of overlap among the digits. UMAP and T-SNE makes a much better job of gathering most of the items from a class. Some classes are still bounded close together like {3, 8, 5}, {9, 4, 7} the main difference was UMAP is the segregated more spaciously with some overlaps which suggests, it is predicting wrong digits with a higher probability than T-SNE

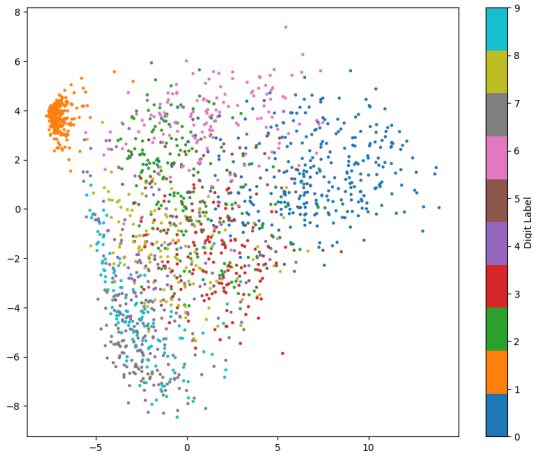

Figure 2: PCA

This showed that PCA is not well suited for high dimensional data such as MNIST with non-linear properties and complex space. It is useful for capturing linear relationships. Whereas UMAP and TSNE was able to do wonders, they properly segregated and classified each of the digits by clustering the training data together by their properties and differentiating it enough from clusters of the other digits. UMAP and TSNE take large computational time being an unsupervised technique. TSNE seems like it has more noise understanding than UMAP. These graphs of the digits, scatter plotted,

seem to correspond with our earlier conclusions obtained using techniques of cloud centers.

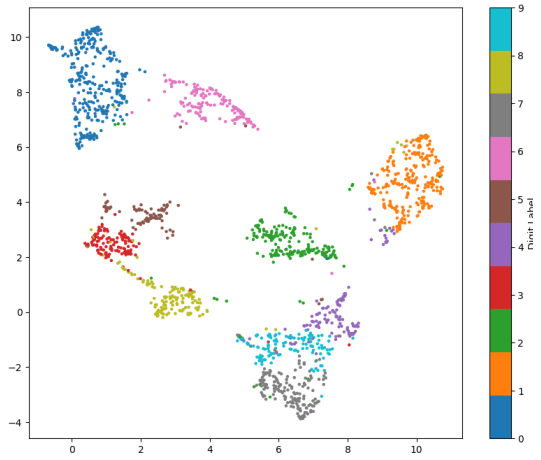

Figure 3: U-MAP

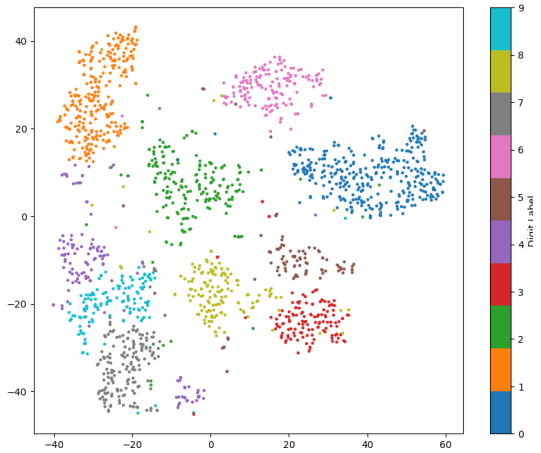

Figure 4: t-SNE

2.3 Nearest Mean Classifier and KNN

Next, our aim is to see how our model can predict well with the test data we are provided. Starting from the nearest mean classifier and comparing it to K-Nearest-Neighbour (KNN) classifier. Using nearest mean we got an accuracy of 86.35% for the training set and 80.40% for the test set. Even though the accuracy rate is not that high, the model seem to hold as it gives an equivalent prediction to the train as well as the test thereby showing it does not overfit the training data. This has confusion for the pairs {0, 6}, {2, 8}, {3, 5} and {4, 9}. It performs worst for the digits 2 and 5. Whereas KNN was able to produce much more accurate results. It produce the best of results for the calues $k < 5$. Figure 5 shows the percentage of accuracy on training as well as test data for both the nearest mean and KNN for k=5. KNN with $k = 1$, gives train accuracy of 100% and test accuracy of 91.50%. Whereas with k=4, it results in train accuracy of 97.89% and test accuracy of 91.40%. KNN generates accurate results across the digits with only digits {5, 0} and {5, 3} have a confusion greater than 10%.

3 Perceptron

For this task we trained a multi class single layer perceptron model that learns to classify the input image among the 10 digit classes provided. The approach utilises supervised learning, leveraging the concept of gradient descent to update the weights iteratively. The main purpose is to minimise the loss function and enhance the model's ability to give correct predictions. To improve the flexibility of the model, we include the bias. For computational convenience, the inputs are appended with 1s. This makes our total input features size as (1076, 257). We understood a crucial point while initializing the weights (257, 10). At first, we took random numbers as multiple of 100 and the training took about 991 epochs as seen in the Figure 6. And when we tweaked it to be distributed randomly between the scale of -1 to 1, it led to a faster convergence , within 311 epochs.

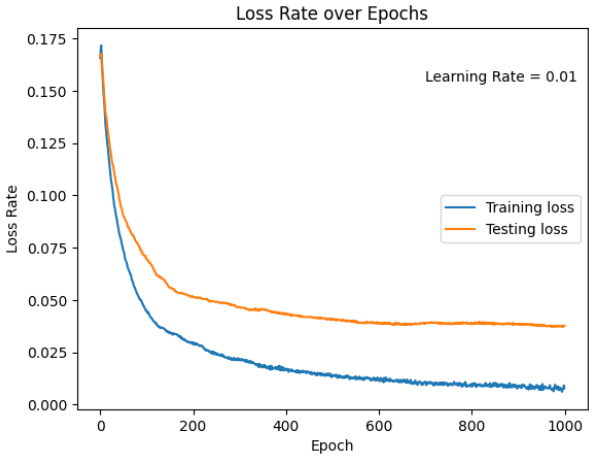

Figure 6: Training performance initiated with Random weights as multiple of 100

We tried with all initial weights as 0s and observed that the overlap between the training and testing accuracy was less and the test accuracy dropped much earlier while the training accuracy was achieved way earlier. This shows that the zeros initlization can lead to same kind of learning for all neurons, leading to poor generalisation and high overfitting. We tried checking the scaling factor, in case of inputs too, but for this particular set we could not find any significant difference as we are dealing with just a single layer perceptron.

Another interesting thing was observed while changing the learning rate - a factor that determines the step size of updating the weights. The higher the learning rate, the convergence speed will be slower and hence, finding a stable learning rate is important. As seen in the Figure 7, it took about 400 epochs when initialised as 0.9. Otherwise, it took about an average of 270 epochs when value was 0.01. In case of even more complex data, the high learning rate might even skip global minima and get stuck with different local minima , also leading to exploding gradient problems.

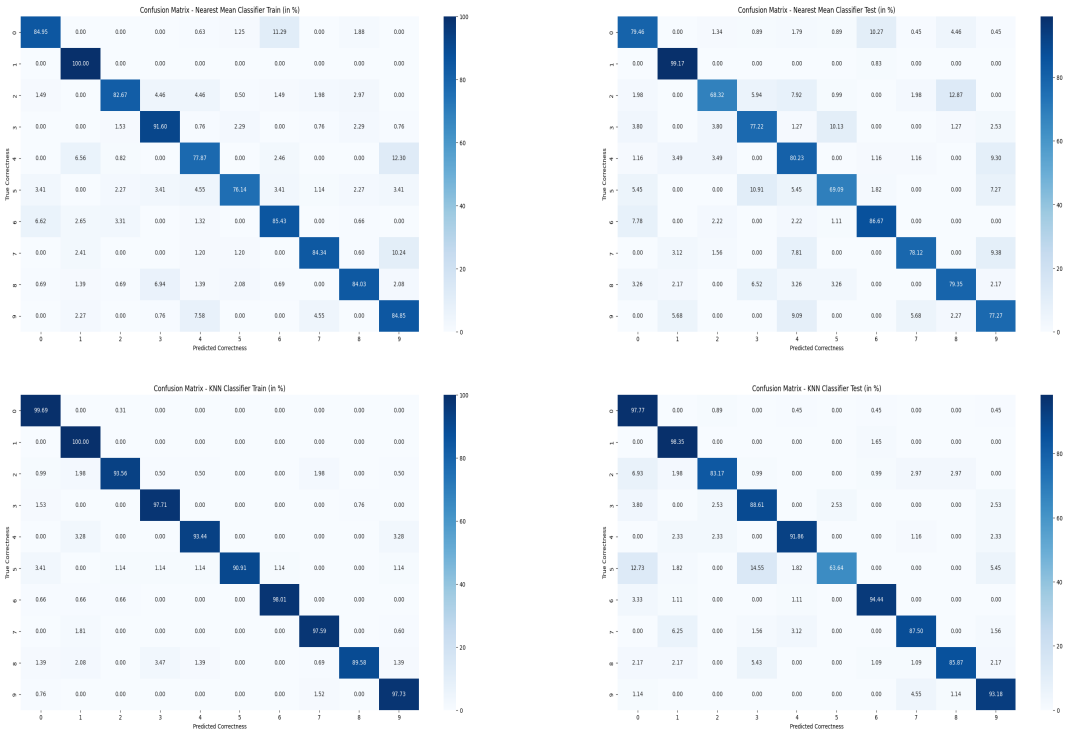

Figure 5: Confusion Matrix for Nearest Mean Neighbor and KNN for both train and test data.

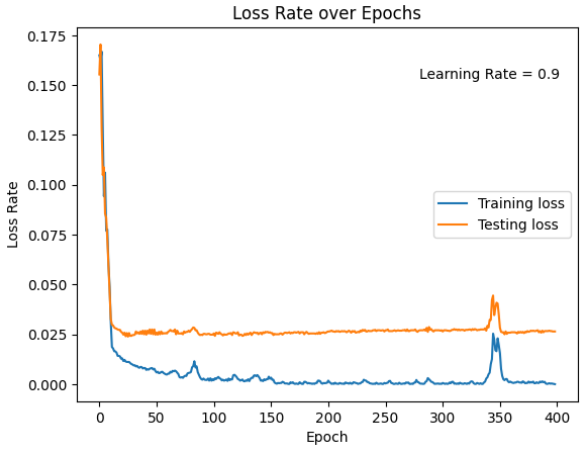

Figure 7: Training performance when started with lr = 0.9

One of the other key takeaways was the importance of using one hot encoding after applying argmax in classification tasks. If predictions are compared directly as numerical values it can lead to larger error magnitude and it will never lead to any convergence. The model would mistake it for learning about numerical values rather than categorical match.

Overall, it appears that the single-layer multiclass perceptron model has overfitted to the training data, as test accuracy attained its peak at approximately 88.9% around epoch 60, despite the loss continuing to decrease and eventually reaching zero after another 200 epochs. This pattern suggests that while the model can perfectly fit the training data (resulting in zero loss), it fails to generalize well to unseen data, leading to stagnation or even degradation in test performance. The model was unable to learn hierarchical representations of data like edges, textures etc.

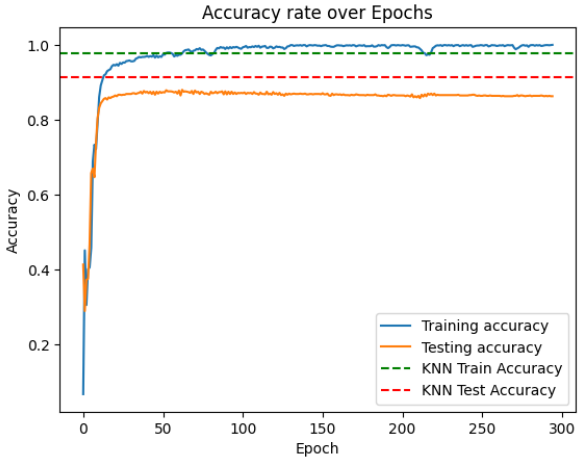

Figure 8: KNN and Single layer Perceptron Performance

In conclusion, KNN is more suited for such classification tasks ,see Figure 8 because it does not assume a fixed boundary like linear classifier to separate the classes, rather it focuses on the distance proximity within the features. For data like MNIST where other features of image like lighting, noise, scale are not a concern, KNN can be more useful.

4 Neural Network

In task 3 we created a neural networks that solves the XOR problem. We implemented the mean squared error function for loss and also wrote functions to calculate the gradients for all the weights in the network and the backpropagation algorithm from scratch. Finally, we also implemented a "lazy randomised approach", and experimented with Tanh and ReLU activation functions.

4.1 Method

We began task 3 by creating functions for sigmoid and its derivative using numpy so that we can base our solution on vectors. We randomly initialised two numpy arrays of dimensions (2, 3) and (1, 3) to be used as weights and also copied those values to separate variables so as to always use them as starting points in our training. We also generated two numpy arrays for the data with dimensions (4, 2) and (4, 1) and filled them with the appropriate values. Afterwards we wrote functions to perform the network inference, and calculate the error and the gradients for each weight in a vectorized manner. The initial approach was with sigmoid as activation and a learning rate of 10, then we implemented it for variable activation functions and gradients. Then we implemented the backpropagation algorithm and ran it until the error became less than 0.01.

After verifying the results for the sigmoid function we experimented with choosing the weights by ourselves, then with a randomised approach for weights in the (-10, 10) range and also used Tanh and ReLU as activation functions. We had to use lr=0.1 for these otherwise the algorithms wouldn't converge. To understand what decisions the algorithms were learning to make and also for easier debugging we plotted the decision boundaries of the models for a 2d input space. We know this won't be available in datasets with more than two input variables but it was really helpful for debugging the neural network implementation. These plots can be seen in Figure 9.

For the manual approach we tried to "catch" the cases where both inputs are 0 or 1 with the first layer and use that in the second layer as if we were constructing logic gates. To do that we chose the weights -12 (bias) and 20, 20 for the first and 26 (bias), -20, -20 for the second neuron. This way the first neuron will have an input of -20 for both 0, 12 in the XOR case and 28 when both are one. So when inputs are the same the neuron will be 0, otherwise after the sigmoid activation. The other neuron will be 26, 6, -14, so 1, 1, 0 so it's 0 only when both inputs are 1. So basically we created $NAND(x1, x2)$, $NAND(x1', x2')$ and combined them with weights -6,5,5 so if they are both 0 we get a -6 input $\rightarrow$ 1, which is larger than $\frac{1}{2}$ so we get a 1 and if either of the neurons activate input goes to -1 so after sigmoid output drops to $\frac{1}{2}$ so we get a 0.

4.2 Results

All of the approaches were able to approximate the xor function and yield results. Obviously the manual approach did not need any iterations but the time it took to translate the logic gates into perceptron networks was a lot longer than running all the other approaches.

For the randomised we got an error below our target, 0.01 after 75130 attempts. Our initial attempt yielded an error of about 0.5 after 100.000.000 iterations but we were using weights in the (0, 10) range so we were limiting the model.

In comparison sigmoid needed only 418 attempts, Tanh 1589 and ReLU 298. The only noticeable problem that we had was that initially ReLU could not converge to a satisfying result. We suppose this was because it had difficulty handling the 0 values so we mapped the inputs from 0, 1 to -1, 1 when training with ReLU and it worked just fine.

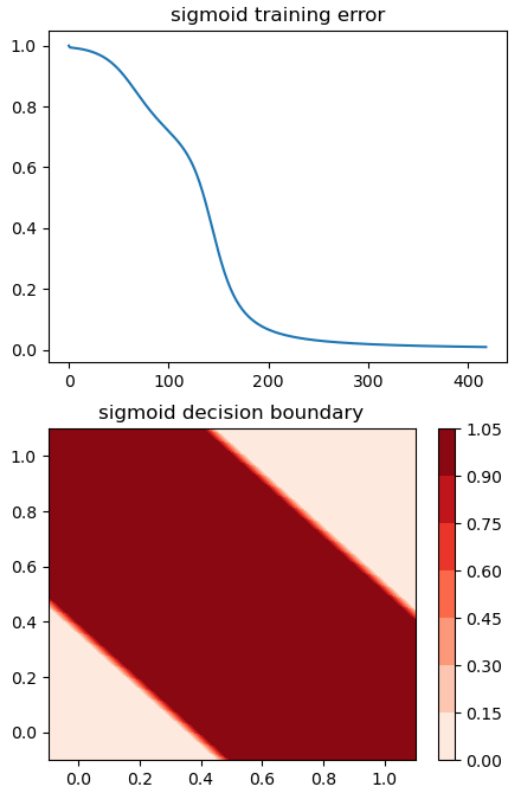

Figure 9: Sigmoid training error, decision boundary

4.3 XOR Conclusion

In the end all implementations were able to converge with error values less than 0.01. Sigmoid and ReLU activations resulted in faster convergence than Tanh and the lazy approach was off the scale even with only having to guess 9 weights.