Introduction to Deep Learning: Assignment 1

Ioannis Koutalios (s3365530)
Daan Planken (s1838547)
Jelle Pleunes (s2443341)

October 9, 2024

Introduction

For this assignment we use different approaches (distance-based classifiers and multi-class perceptrons) for classifying images of handwritten digits from the well-known MNIST data set [1]. The training set consists of 1707 images and the test set consists of 1000 images. Additionally, we are implementing and training a multi-layer neural network that implements the XOR function.

Task 1: Data dimensionality, distance-based classifiers

1.1

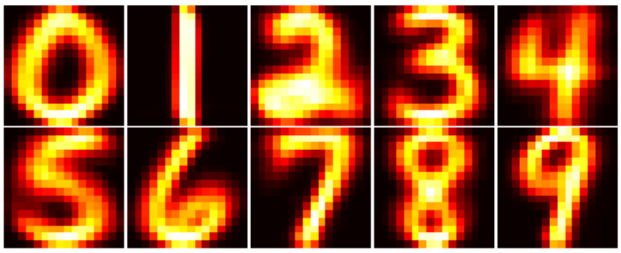

Figure 1: The average over the training set of each of the digits

A relatively simple way to classify the digits, is to compare them to the average images of each digit in the training set, and pick whichever is closest. Here 'close' and 'average' refer to the images when viewed as 256-dimensional vectors. The first step is then to calculate the average image of each of the digits, which can be seen in Figure 1.

Next, in order to get a feel for the effectiveness of this classifier, we plot the distances between each of the averages, resulting in the table seen in Figure 2.

[0.0	14.4	9.3	9.1	10.8	7.5	8.2	11.9	9.9	11.5]
[0.0	0.0	10.1	11.7	10.2	11.1	10.6	10.7	10.1	9.9]
[0.0	0.0	0.0	8.2	7.9	7.9	7.3	8.9	7.1	8.9]
[0.0	0.0	0.0	0.0	9.1	6.1	9.3	8.9	7.0	8.4]
[0.0	0.0	0.0	0.0	0.0	8.0	8.8	7.6	7.4	6.0]
[0.0	0.0	0.0	0.0	0.0	0.0	6.7	9.2	7.0	8.3]
[0.0	0.0	0.0	0.0	0.0	0.0	0.0	10.9	8.6	10.4]
[0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	8.5	5.4]
[0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	6.4]
[0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0]

Figure 2: The distances between each of the averages. Values a_{ij} with $i \geq j$ have been set to zero. Otherwise the matrix would be mirrored in the diagonal.

[1] http://yann.lecun.com/exdb/mnist/

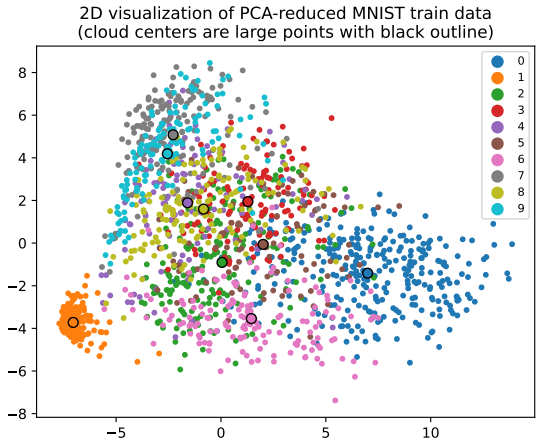

Figure 3: PCA

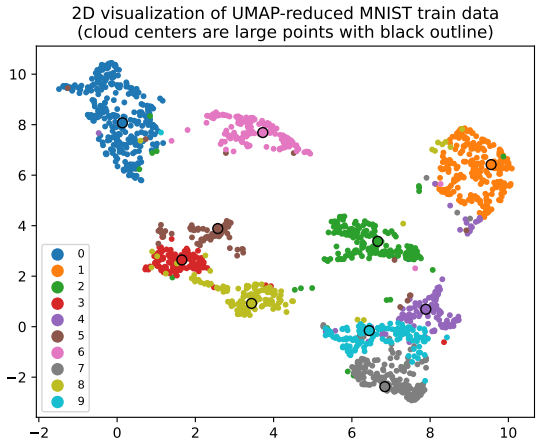

Figure 4: UMAP

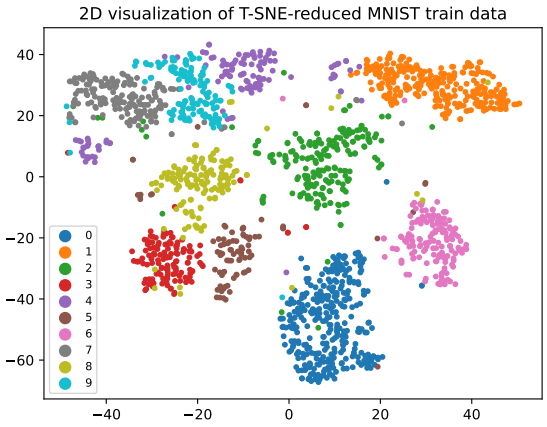

Figure 5: T-SNE

The pair that is closest together seems to be "seven" and "nine", with a distance of around 5.4. This makes sense, since we can see in Figure 1 that "seven" and "nine" seem to overlap quite a lot. This would then indicate that this classifier would be at least reasonably accurate, since it is able to pick up on the "closeness" of "seven" and "nine", as well as other digits you would expect to be close to each other, such as "four" and "nine". Therefore it is also likely that the classifier would classify a digit correctly if it is "close" to any of the averages.

1.2

Clearly, we can see from Figure 3, Figure 4 and Figure 5 that each of the visualisations seem to match our intuition: 7 is close to 9, as is 4, while for example, 0 is always very far away from 1, just as we would expect.

1.3

On the training set, the algorithm classified 1474 of the digits correctly, and 233 incorrectly, for a percentage of 86.35%. On the test set, the algorithm classified 804 digits correctly, and 196 incorrectly, for a percentage of 80.4%. Therefore, the classifier is reasonably accurate, but far from perfect.

1.4

Using the KNN classifier (with $k = 3$), we classified 1671 of the training set correctly, and 36 incorrect for a percentage of 97.89%. On the testing data, we classified 914 correctly, 86 incorrectly, for a percentage of 91.4%.

In order to get an understanding of the digits that get misclassified, we can consider the so-called confusion matrix. In this confusion matrix, the element $a_{ij}, 0 \leq i, j \leq 9$ represents the number of digits i that get classified

```
[318   0   1   0   0   0   0   0   0   0]
[  0 252   0   0   0   0   0   0   0   0]
[  1   0 195   0   0   0   0   4   1   1]
[  1   0   1 129   0   0   0   0   0   0]
[  0   2   0   0 117   0   0   0   0   3]
[  3   0   0   0   0  84   1   0   0   0]
[  0   1   0   0   1   0 149   0   0   0]
[  0   0   0   0   1   0   0 164   0   1]
[  1   2   0   4   1   0   0   0 136   0]
[  1   0   0   1   0   0   0   3   0 127]
```

Figure 6: The KNN confusion matrix on the train set

```
[219   0   2   0   1   0   1   0   0   1]
[  0 119   0   0   0   0   2   0   0   0]
[  6   2  87   1   0   0   1   1   3   0]
[  4   0   1  70   0   2   0   0   0   2]
[  0   2   1   0  78   1   0   1   0   3]
[  5   0   0   9   1  37   0   0   0   3]
[  2   1   0   0   1   0  86   0   0   0]
[  0   3   0   1   3   0   0  56   0   1]
[  1   2   0   4   1   0   1   2  79   2]
[  1   0   0   0   0   0   0   3   1  83]
```

Figure 7: The KNN confusion matrix on the test set

as j. So a_{33} is the number of "threes" that get correctly classified, and a_{12} is the number of "ones" classified as a "two".

However, from the confusion matrices Figure 6 and Figure 7, we see that the digits that are most often confused are not the ones we would expect given our visualisation: on the test set, only 3 of the "four" digits get classified as a "nine", while not a single "nine" gets classified as a "four". Meanwhile, 9 "fives" get classified as a "three", while "three" and "five" were not very close according to the visualization.

One possible explanation for this discrepancy is that there are large differences between the total number of each digit in both the train and the test set. In the training set, there are > 300 "zeros", but < 90 "fives". This could then bias our classifier towards "zeros", since the probability that a "zero" is accidentally one of the nearest neighbours is fairly large, while the probability that a "five" is correctly the nearest neighbour is small, since there are comparatively few "fives".

This would explain why so many digits get incorrectly classified as a "zero", and why so few digits get classified as a "five". In order to circumvent this problem, one might normalize the number of instances for each digit, in order to more accurately determine which digits are difficult to distinguish, and possibly to improve the correctness of the KNN classifier, but this is outside the scope of this report.

Task 2: Implement a multi-class perceptron algorithm

For this task we created a single-layer perceptron with 10 nodes using the multi-class perceptron training algorithm. The inputs for this node were 256 + 1 bias (which is equal to 1). The output has 10 values between 0 and 1, each one representing one digit. The index of the maximum value of the output determines the digit that our perceptron predicts for each input.

We implemented two ways of training the perceptron. The first one uses the whole training data set for calculating one update for the weights. The second one calculates an update for the weights with each individual input from the training data set one by one and then repeats the process until the perceptron is trained. In Table 1 we compare how efficiently each of the two implementations trained the perceptron.

The weights are initialized using `np.random.RandomState`, where a seed is set to make results reproducible. All of the results can be generalized with small differences for any initialization of the weights, which is something that was tested by changing the seed for the random state. The weights have a shape of 257×10 because we have 257 inputs (256 + 1 bias) and 10 nodes to our single-layer perceptron.

The way the weights are updated is by using the "Adaline learning rule". At first we calculate the dot product of our inputs with the weights. We then use the activation function over the result of this multiplication to get values between 0 and 1. The activation function in our case is a sigmoid with the formula: $f(x) = \frac{1}{1+e^{-x}}$, which always gives a value between 0 and 1. We then update the weights by using the formula

$$\Delta W = \eta \cdot x \cdot (d - out)$$

where η is the learning rate, d is the correct output, out is the result of the activation function over the multiplication result and x is the input we used.

One issue we faced while developing the algorithm was that we were unable to implement the "Neuron learning rule" to update the weights by making use of the derivative of the activation function over the output. The formula for this is: $\Delta W = \eta \cdot x \cdot (d - out) \cdot out'(x)$ where $out'(x) = out(x) \cdot (1 - out(x))$. When using this learning rule the weights were not getting updated correctly and the algorithm never converged, giving us very low accuracy.

The stopping condition for both training methods is to stop when the accuracy over the training set is equal to 1 or until the number of consecutive loops over the whole training set with no improvement is 15. The number 15 was selected after trying different numbers for multiple initialization of the weights. The learning

rate was also tested for different initializations of the weights and it was found that $\eta = 0.2$ was able to train the perceptron every time and was faster than smaller values.

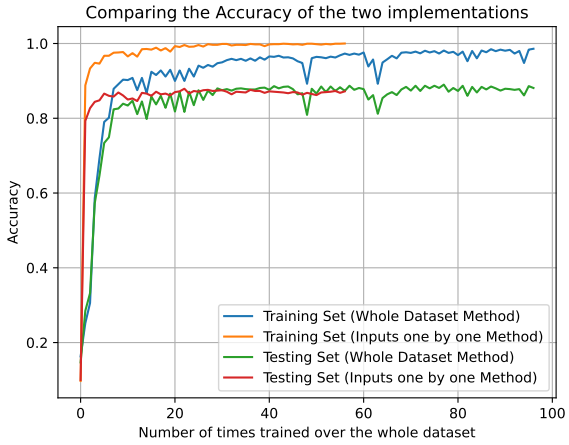

Figure 8: The accuracy of both the training and testing sets for the two different implementations of the perceptron training. We can clearly see that the method of training using the inputs one by one requires less loops over the whole training set, it is however much slower as compared with the method of using the whole data set at once.

We can clearly see in both Figure 8 and Table 1that both methods have trained the perceptron and the algorithm converged. We achieve perfect accuracy on the training set meaning that no cases were misclassified. For the testing set the accuracy falls to around 88% for both cases (a bit lower for the one by one implementation but almost insignificant). The real difference between the two comes at the time they required in order to converge. When we used the whole data set instead of training for each input the training time reduced significantly (by a factor of around 2.5). Although in Figure 8 we can see that this approach uses the training data set more times in order to reach a plateau, doing one matrix multiplication instead of 1707(the number of inputs in the training set) requires less time, reducing the overall amount of time and thus making the algorithm more efficient.

Method	Training Set Accuracy	Testing Set Accuracy	Running Time
Whole data set	0.986	0.881	2.869
One by one	1.000	0.872	6.729

Table 1: The results of the accuracy of the multi-class perceptron on the training and testing data set with the running times of the training process.

Lastly we print the confusion matrices for both training methods. The two matrices for the testing set are shown in Table 2, the left table representing the whole data set approach and the right table the one by one approach. The matrices for the training set are not in the report because the accuracy is almost or equal to 1, meaning that the matrices are almost completely diagonal and there is no more information to be obtained from the confusion matrix. From the tables we can see that both perceptrons had some difficulties classifying the digits 4 and 5 (accuracy is around 0.75 compared to the 0.88 for all the digits) and did exceptionally well in classifying the digits 0, 1 and 6 (accuracy over 0.92). Both perceptrons confused inputs of the digit 2 with the digit 8. As we see in the left table 6 inputs of the digit 2 were classified as 8 and 8 cases on the right table.

When comparing the perceptrons accuracy to the distance-based classifier based on centroids and the KNN method from the task 1, the KNN method is the one achieving the highest accuracy on the testing set. The distance-based classifier based on centroids also outperforms the perceptron. The results are presented in Table 3. The perceptron is limited by the fact that it has only a single layer, which may be the reason why it is not able to compete with simple distance-based classifiers.

Task 3: Implement the XOR network and the Gradient Descent Algorithm

Our task is to implement a multi-layer neural network, for which the weights are adjusted (trained) using a gradient descent algorithm.

	0	1	2	3	4	5	6	7	8	9
0	214	0	4	1	3	0	0	0	1	1
1	0	117	0	0	1	0	2	0	0	1
2	2	0	83	2	5	0	1	2	6	0
3	1	0	2	63	0	5	0	2	4	2
4	2	1	2	1	66	2	2	4	1	5
5	3	0	0	5	2	41	1	1	0	2
6	3	0	0	0	2	1	84	0	0	0
7	0	0	1	1	3	0	0	56	0	3
8	3	0	1	4	1	4	1	1	76	1
9	0	1	0	0	1	0	0	3	2	81

	0	1	2	3	4	5	6	7	8	9
0	215	0	2	0	3	0	2	0	1	1
1	0	117	0	0	1	0	2	0	0	1
2	1	0	83	4	3	0	1	1	8	0
3	2	0	3	67	0	1	0	1	3	2
4	3	2	4	0	63	3	3	1	1	6
5	3	0	0	3	3	41	0	2	0	3
6	1	0	0	0	2	4	83	0	0	0
7	0	1	1	0	4	0	0	55	0	3
8	3	2	2	4	2	1	1	3	72	2
9	0	2	1	0	2	0	0	5	2	76

Table 2: The confusion matrices for the testing data set for the two different implementations of the perceptron training. On the left is for the perceptron trained with the whole training set each time, while on the right for the perceptron trained with the inputs given one by one.

Method	Training Set Accuracy	Testing Set Accuracy
Perceptron (Whole data set)	0.99	0.88
Perceptron (One by one)	1.00	0.87
distance-based classifier (based on centroids)	0.86	0.80
KNN ($k = 3$)	0.98	0.91

Table 3: The results of the accuracy of the multi-class perceptron on the training and testing data set compared with the distance-based classifier and the KNN method from Task 1

3.1

Our function $xor\_net(inputs, weights)$ simulates the network with two inputs, two hidden nodes and one output node. The first three elements of the *weights* vector correspond to the incoming weights for the first hidden nodes. The next three elements correspond to the incoming weights for the second hidden node. The final three elements correspond to the incoming weights for the output node. The sigmoid activation function is used by all non-input nodes

3.2

The $mse(weights)$ function iterates over all possible inputs of the network $inp \in \{0, 1\}^2$ and adds the squared norm of each difference vector to the result sum. Finally, the result sum divided by the number of inputs (4) is returned.

3.3

The $grdmse(weights)$ function calculates the update of the weights using the Generalized Delta Rule:

$$\Delta W_{ji} = \eta \delta_j y_i, \text{ where } \delta_j = \begin{cases} \varphi'(v_j)(d_j - y_j) & \text{if } j \text{ is an output node} \\ \varphi'(v_j) \sum_{k \text{ of next layer}} \delta_k w_{kj} & \text{if } j \text{ is a hidden node} \end{cases} \quad (1)$$

with w_{ji} the weight from node j to node i, η the learning rate, y_j the output of node j, and v_j the activation of node j.

The calculation of ΔW is done using back propagation, meaning that it starts with calculating the three incoming weights of the output node, after which the six total incoming weights of the two hidden nodes are computed. ΔW is calculated for all four possible input-output pairs, and then the averaged vector (with dimension 9, like the *weights* vector) is returned.

3.4

The gradient descent algorithm starts by initializing *weights* to a vector of 9 normally distributed (mean 0, standard deviation 1) random values. Then the algorithm iteratively updates *weights* using $weights := weights + \eta \cdot grdmse(weights)$ until $xor\_net(inputs, weights)$ is correct for all input-output pairs (where an output value greater than 0.5 is interpreted as "1" and other values are interpreted as "0").

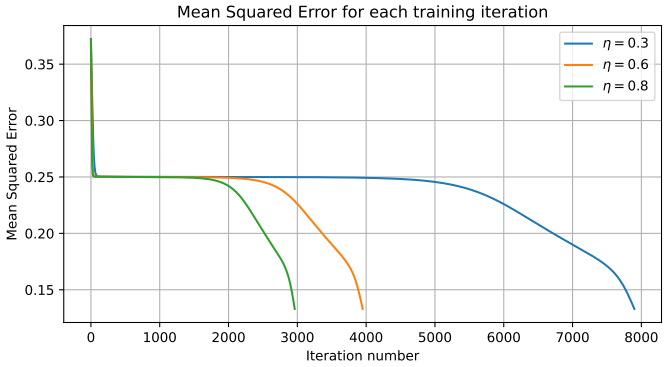

Figure 9: MSE value across gradient descent iterations, for different learning rates η.

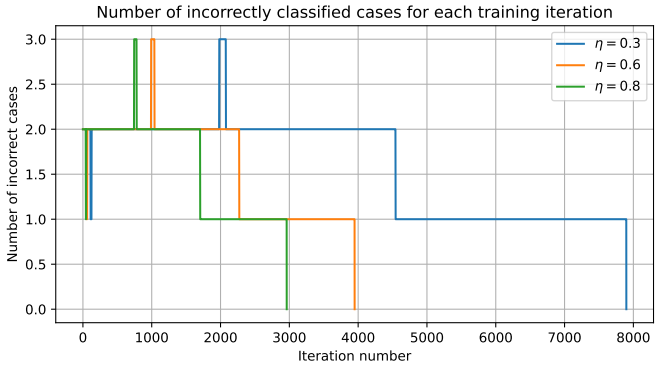

Figure 10: Number of incorrectly classified inputs across gradient descent iterations, for different learning rates η.

Results

We start by training the network on the XOR input-output pairs using different values for the learning rate η. Here the same initial weights were used across training runs. Figure 9 shows the evolution of the MSE during training. In this example, a higher learning rate leads to faster convergence. This does not always hold, and in our experiments we saw cases where $\eta = 0.6$ took fewer iterations than $\eta = 0.8$, for instance. A possible cause could be that with a higher learning rate the algorithm more easily 'overshoots' desired weight values. Figure 10 shows number of incorrectly classified inputs during training. In this example the pattern is visually similar for all three values for η, and we can see that the number of incorrectly classified inputs decreases earlier for higher η.

A different initialization strategy was also used for the weights. Instead of sampling from the $\mathcal{N}(0, 1)$ distribution we sample from the $\mathcal{U}(-1, 1)$ distribution. Figure 11 and Figure 12 show that sampling from a

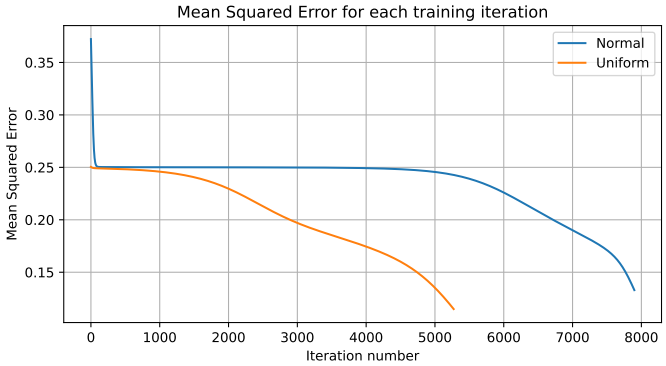

Figure 11: MSE value across gradient descent iterations, for initial weights sampled from a normal distribution and for weights sampled from a uniform distribution.

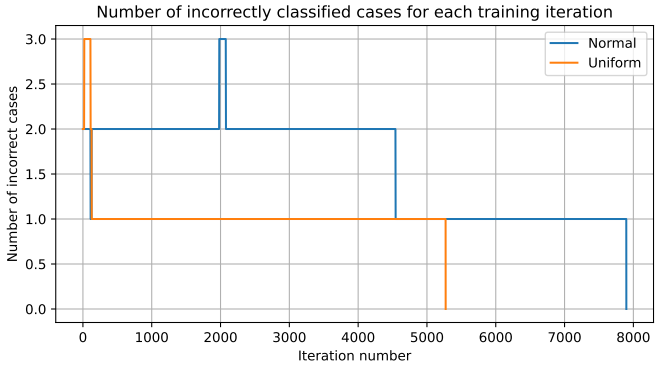

Figure 12: Number of incorrectly classified inputs across gradient descent iterations, for initial weights sampled from a normal distribution and for weights sampled from a uniform distribution.

uniform distribution leads to faster convergence for our example. Learning rate $\eta = 0.3$ was used here. An explanation for this could be that the sampled values are not as close to 0 when using the uniform distribution, which means that a given input may lead to more extreme activations (farther from 0.5) using the initial weights. These activations could mean that the initial network is closer to an XOR network in its behavior.

Finally, the "lazy approach" for finding weights was used. A vector of 9 normally distributed weights was repeatedly generated, until a set of weights implementing the XOR network was found. This experiment was repeated for multiple runs, and the results can be found in Table 4. On average, 131,891 sets of weights needed to be generated, but there is a large variance with outliers as small as 8,992 and as large as 393,348. We can conclude that the "lazy approach" is inefficient and inconsistent in the number of loop iterations required.

Run	#Loop iterations
1	52,980
2	46,931
3	393,348
4	87,585
5	43,016
6	8,992
7	271,175
8	151,100

Table 4: Number of loop iterations (sets of weights) needed to find weights for the xor net, using the lazy approach.

Conclusions

The distance-based classifier, the KNN classifier and the single-layer perceptron were all implemented successfully for use with the MNIST data set. KNN showed the highest accuracy by a small margin among these classifiers, followed by the distance-based classifier based on centroids. The single-layer perceptron showed the lowest accuracy. Finally, the XOR network was trained successfully using back propagation for different learning rate values and different weight initialization strategies.

Contributions

Report task 1: Daan Planken
Report task 2: Ioannis Koutalios
Report task 3: Jelle Pleunes
Code: everyone