Introduction to Deep Learning

Homework #1

Elvis Somers; Felix Wente; Yuze Zhang
Student Numbers: S3007146; S3337731; S2819368

Problem 1

The purpose of the first task was to develop some intuitions of data points in high dimensional spaces. This was done by (1) developing a straightforward distance-based algorithm for classifying handwritten digits and comparing it to the K-nearest neighbours algorithm,which is also distance-based; and (2) using dimensionality reduction techniques to visualise the high-dimensional data.
In the first subtask the goal was to set up the simple algorithm, such that it can later be used to classify digits. This was done by separating the data into 10 distinct clouds representing the 10 different digits in the 256-dimensional (pixel) space. The clouds were created by extracting the indices of the data points belonging to a certain number from the train_out data and filtering the train_in data by those indices. The filtered data points (clouds) were then assigned to a dictionary separated, where the keys are the ten distinct digits and the values are the data points (clouds) belonging to a certain digit. The choice of a dictionary was twofold: (1) one can easily index and loop through the different clouds by calling the different keys of the dictionary and (2) Dictionaries use hash lookup, while lists require walking through the list until it finds the result, which make dictionaries much faster.
Next the centroids of those clouds were computed as they will be used as the distance measure of how far a particular data point is from the 10 different data clouds. The smallest distance to a cloud determines to which cloud a new data point would belong, i.e., classifying it to one of the 10 digits. The centroids is just a 256-dimensional vector of means over all coordinates of vectors belonging to a certain cloud, which were again stored under different keys within a dictionary. To say something about the expected accuracy of the classifier, the (euclidian) distances between the centroids were computed. The centroids closest together (Distance = 5.43) were the ones of number seven and nine. Those will likely be the numbers hardest to separate. On the other hand number zero and one had their centroids furthest apart(14.45). Those will be the numbers easiest to distinguish by the classifier.
The Figures 1 2 and 3 Represent the dimensionality reduction algorithms: PCA, UMAP, T-SNE respectively. In general, one can see that the separation is much more clearer for UMAP and t-SNE than in PCA. However, they all agree which numbers are closer together, i.e., are more similar in the data space and which are further apart. They agree with our centroid model that numbers seven and nine are closest together, i.e., most similar, while numbers zero and one have the clearest distinction and are, therefore, furthest apart.
In order to use the algorithm for classification we now loop through all the images in the data and compute the distances of that image to all our previously calculated centroids. The label (number) assigned to that image will be the one that belongs to the centroid with the smallest distance to that image. To get the accuracy of the classifier the number of correct classifications are counted and divided over the total number classifications. This procedure separately for both the training and test data. For the training data the classifier yielded an accuracy of 86%, for the test data it yielded an accuracy of 80%.
For comparison, the K-nearest neighbours algorithm,a less naive classifier yielded an accuracy of 94% for the training data and 88% for the test data. When looking at the confusion matrices both classifiers indicate that number five had the fewest correct classifications. Numbers seven and nine

Problem 1 continued on next page. . .

got the most misclassifications between each other, as was predicted, since their centroids are the closest together. This, however, is only true for the training data but not for the test data. This might be due to two reasons: (1) due to the lower sample size in the test set the trend would only be visible with a larger number of pictures in the test set. (2) It just happened to be the case that the centroids of 7 and 9 were closer in the training data but this was rather due to noise than an actual trend, which didn't carry over to the test set. The second reason is supported by the fact that the confusion matrix for both the test and training data of the knn classifier as well as the confusion matrix for the test data of the simple classifier agree that 4 and 9 got the most misclassification. This would also make sense intuitively since these numbers can be written fairly similarly.

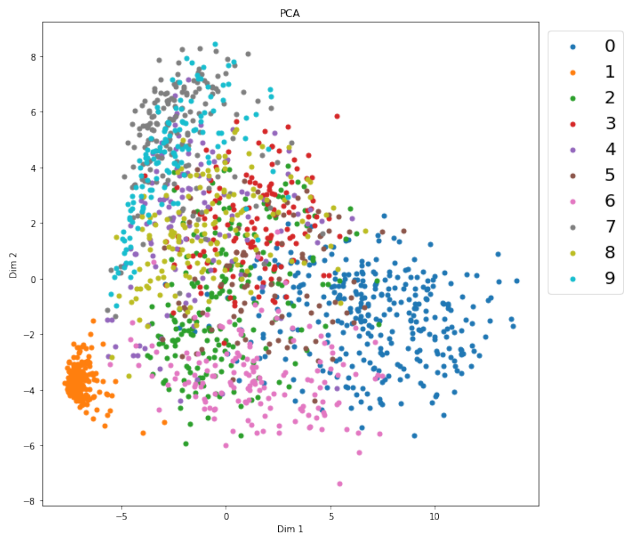

Figure 1: PCA

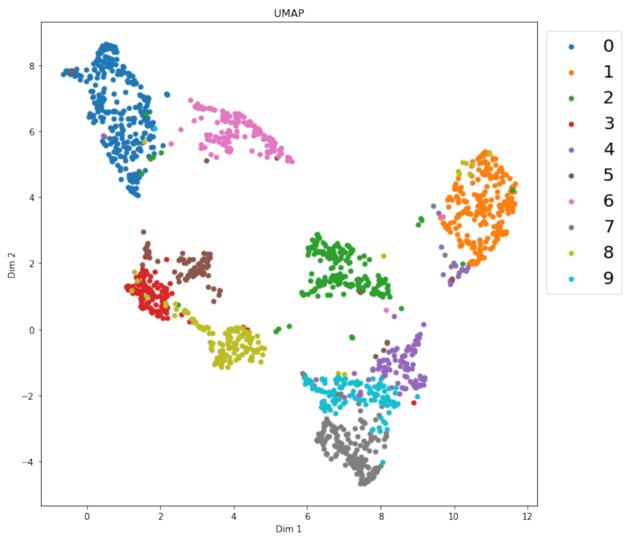

Figure 2: UMAP

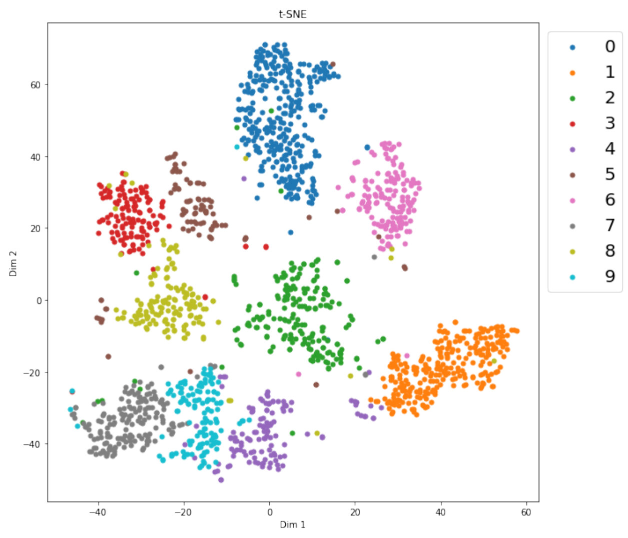

Figure 3: t-SNE

Problem 2

Implementing a Multi-Class Perceptron

A Perceptron network is the simplest version of a neural network - it uses only a single layer of "cells", to which we refer as a "perceptron". In essence, this cell is nothing other than a mathematical formula. The mathematical formulation of the perceptron can be stated as follows:
Given a vector of inputs $\mathbf{x}$, return an output $\hat{Y} \in \{0, 1\}$ given by

$$\hat{Y} = f\left(\sum_i w_i x_i + b\right),$$

where x_i are the elements of the input vector $\mathbf{x}$, and f, w_i and b are aspects of the perceptron. These aspects are referred to as the *activation function*, *weights* and *bias*, respectively. Generally, the activation function is taken to be the unit step function.
Here, we have a *labeled* set of inputs, which is referred to as a *training set*. The training set consists of n (here, $n = 1704$) input vectors $\mathbf{x}_j$, each with a corresponding label Y_j. Our goal in training the network then, is to make the predictions $\hat{Y}_j$ given by our network correspond with the actual labels Y_j as well as possible. In order to do so, we update the weights and biases of our perceptrons after every labeled data point $\{\mathbf{x}_j, Y_j\}$ we evaluate. In order to do so, we use the following *update rule*:

$$\Delta w = \eta(Y_j - \hat{Y}_j)x_j,$$
$$\Delta b = \eta(Y_j - \hat{Y}_j),$$

where η is referred to as the *learning rate* of the perceptron.
A problem with this approach is that the prediction given by the perceptron is a binary one; $\hat{Y} \in \{0, 1\}$. However, we are looking to identify handwritten digits, which is a classification problem with multiple classes - ten, to be precise. In order to resolve this problem, we define ten different perceptrons P_i. Each of these perceptrons makes a binary classification of the following form: $P_i : \hat{Y} \in \{i, \neg i\}$. So we have one perceptron identifying the digits as "zero or not-zero", one identifying them as "one or not-one", et cetera.
Then to obtain a final prediction for each given digit we have to pick the perceptron that is "most sure" of its prediction. Of course, in order to do this, we must pick another activation function than the unit step function. If we use the unit step function, we might obtain 1 as a prediction for multiple digits, or 0 for all of them. In these cases there would be no clear decision procedure for identifying the digit. Instead of the step function, we could use the so called ReLU function: $f(x) = x$ if $x > 0$, $f(x) = 0$ otherwise. Alternatively, we could use the "empty" activation function, $f(x) = x$. This is equivalent to using no activation function at all.
Testing out a few different approaches for defining the attributes of the perceptron network, we found the highest test set accuracy when using the empty activation function and using a random uniform distribution on the interval $[0, 1]$ for the initial weights and biases. Using these approach, we consistently obtained a test set accuracy of $\sim 87\%$.

Problem 3

Implement the XOR network and the Gradient Descent Algorithm

Part 1:

First, let us look at what we are trying to classify. The four points (`[0,0]; [0,1]; [1,0]; [1,1]`) produce corresponding results (`0; 1; 1; 0`). Since the two groups (i.e., 1 and 0) among the four points cannot be separated linearly. Hence, we need a multiple-layer network with multiple perceptrons to help with the classification.

For each point, there are two digits, x_1 and x_2. For the first layer, for a single point (x_1, x_2), the equation we are using is the same as for a single perceptron. However, we need four weights, two biases, and two outputs, which correspond to two hidden nodes:

$$\begin{bmatrix} w_{00} & w_{01} \\ w_{10} & w_{11} \end{bmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} + \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} = \begin{pmatrix} w_{00}x_1 + w_{01}x_1 + b_0 \\ w_{10}x_0 + w_{11}x_1 + b_1 \end{pmatrix}$$

The activation function f is then applied to each element in the resulting matrix to obtain the input of the second layer, which is also the output of the first layer:

$$\begin{pmatrix} f(w_{00}x_1 + w_{01}x_1 + b_0) \\ f(w_{10}x_0 + w_{11}x_1 + b_1) \end{pmatrix} = \begin{pmatrix} h_0 \\ h_1 \end{pmatrix}$$

For the second layer, we have the "point" (h_0, h_1), and we apply the same operation as above to this "point". This time, we only need two weights, one bias, and one output to correspond to the classified result in the end:

$$\begin{bmatrix} w'_{00} & w'_{01} \end{bmatrix} \begin{pmatrix} h_0 \\ h_1 \end{pmatrix} + \mathrm{b}' = \mathrm{w}'_{00}\mathrm{h}_0 + \mathrm{w}'_{01}\mathrm{h}_1 + \mathrm{b}'$$

$$f(w'_{00}h_0 + w'_{01}h_1 + b') = o$$

Based on the description above, there are nine weights in total (including biases), and for each node except input nodes, there are three weights (two weights and one bias) assigned to each node. A map of the network is shown below in Fig. 4.

To construct the function `xor_net(inputs, weights)`, we only need to assign each of the nine weights to each element of the matrices mentioned above, and apply the equations above to the inputs, which is (x_1, x_2). The result gives the predicted output based on the given weights and biases. This is also called forward propagation.

Part 2:

First, we need to find the equation for mean squared error:

$$\mathrm{MSE} = \frac{1}{\mathrm{n}} \sum_{\mathrm{i}} (\mathrm{y_i} - \mathrm{d_i})^2$$

,

Problem 3 continued on next page. . .

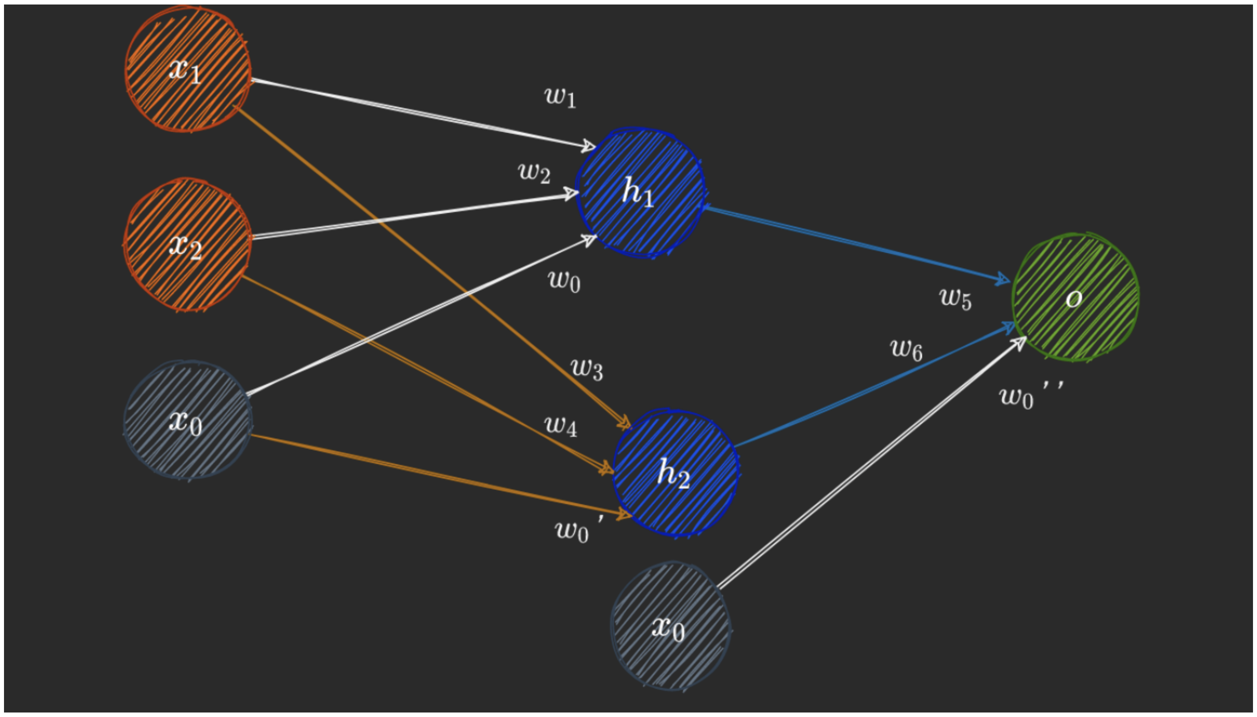

Figure 4: A map of the network with two inputs, two hidden nodes, and one output. Weights w_1, w_2, w_3, w_4, w_5, and w_6 corresponds to w_{00}, w_{01}, w_{10}, w_{11}, w'_{00}, and w'_{01}. w_0, w'_0 and w''_0 are the biases corresponding to b_0, b_1, and b'. Here, 1 and 2 represent nodes/inputs while 0 represents bias "nodes".

where each i corresponds to one prediction y and one expected result d. There are four data points, and we just need to use the equation above over four predictions and expected results to get the MSE.

Part 3:

In this question, we need to calculate the change in weights based on the difference between the predicted result and the actual result. We need to implement the chain rule, $\frac{df(g(x))}{dx} = \frac{df}{dg(x)}\frac{dg(x)}{dx}$. It would also be helpful to see the dependence of the predicted result on other parameters:

$$o = f(w'_{00}f(w_{00}x_0 + w_{01}x_1 + b_0) + w'_{01}f(w_{10}x_0 + w_{11}x_1 + b_1) + b') = f(w'_{00}h_0 + w'_{01}h_1 + b')$$

It is easier to formulate it backwards. The first step is to take the difference between the predicted result and the true result, and it is followed by taking the derivative of the activation function. For the change in b' we simply have:

$$db' = do\left[\frac{df}{d(w'_{00}h_0 + w'_{01}h_1 + b')}\right]$$

To calculate the change in one of the weights in the hidden layer, we have:

$$dw' = do\left[\frac{df}{d(w'_{00}h_0 + w'_{01}h_1 + b')}\frac{d(w'_{00}h_0 + w'_{01}h_1 + b')}{dh}\right]$$

where h is each of the h in the hidden layer. To calculate the change in one of the biases in the input layer, we have:

$$db = do\left[\frac{df}{d(w'_{00}h_0 + w'_{01}h_1 + b')}\frac{d(w'_{00}h_0 + w'_{01}h_1 + b')}{dh}\frac{df}{d(w_0x_0 + w_1x_1 + b)}\right]$$

where h and $w_0x_0 + w_1x_1 + b$ are for each node in the hidden layer. Similarly, to get the change in weights in the input layer, we have:

$$dw = do\left[\frac{df}{d(w'_{00}h_0 + w'_{01}h_1 + b')}\frac{d(w'_{00}h_0 + w'_{01}h_1 + b')}{dh}\frac{df}{d(w_0x_0 + w_1x_1 + b)}\frac{d(w_0x_0 + w_1x_1 + b)}{dx}\right]$$

Again, h and $w_0x_0 + w_1x_1 + b$ are for each node in the hidden layer, and x represents each of the inputs.
Now, we only need to calculate the final predicted result first in the `grdmse(weights)` function using the `xor_net(inputs, weights)` function to get the difference do. Then, we use each weight value and the derivative of the activation function to get the change in weights with respect to the difference in the results. This is also called backward propagation.

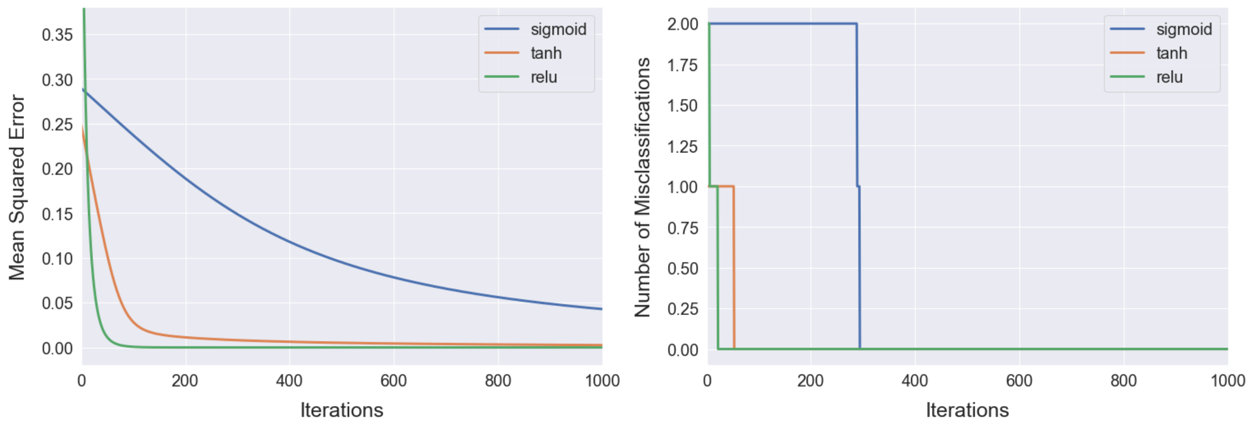

Figure 5: The figures for MSE and the number of misclassifications over the number of iterations. The left panel shows the plot for MSE versus the number of iterations, and the right panel shows the plot for the number of misclassifications versus the number of iterations.

Part 4:
In this part, we need to build a `for` loop of n iterations to make the predicted results converge on the true results (or convergence of MSE to 0). For each iteration, we use `xor_net(inputs, weights)` to get the predicted results for all four points, and we use `grdmse(weights)` to calculate the change of weights for each point, followed by the updates of weights using:

Problem 3 continued on next page. . .

$$w_{new} = w_{old} + lr(dw)$$

and we use these new weights to calculate the next prediction for each point. At the same time, we also record the MSE for each iteration after the predicted results have been calculated. Before the loop, we also create weights using `np.random.uniform` as initial values. We also built a function `classify` to classify the results bigger than and equal to 0.5 to 1 and smaller than 0.5 to 0. The number of misclassifications was also recorded for each iteration. All the aforementioned steps were combined into one training function. The plots for MSE and the ratio of misclassification are shown above in Fig. 5.

Additional Comments and Work:

As can be seen in Fig. 5, after approximately 300 iterations, the correct results were able to be predicted by the network for all three different activation functions. The MSEs of all three different activation functions all dropped below a reasonable threshold after the same number of iterations. Among the three different activation functions, the linear rectifier offers the best performance, while sigmoid offers the worst. However, at the beginning of the iterations, sigmoid and hyperbolic tangent give better predictions, based on observation of the MSE plot.

A possible reason that the linear rectifier appears to be more efficient is mainly due to the fact that this function provides even results for numbers bigger and smaller than 0.5. In this case, it classifies more effectively. In comparison, sigmoid and hyperbolic tangent functions are both concave down above 0, and they might favor certain predicted values more than others. Another thing to notice is that the linear rectifier is a linear activation function, while sigmoid and hyperbolic tangent are non-linear. Regardless of the number of data points, the linear activation function works best for classifications.

Additionally, a `for` loop was also constructed to generate random weights before correct classification. For each iteration, we first generate random weights centered around a mean of 0 and standard deviation of 80 to ensure the possible correct classification without training. Then, we use `xor_net` function to calculate the predicted results. The loop was designed to terminate once the correct results were found and record the number of iterations. We ran the loop 100 times and averaged over number of iterations. The average number of iterations for correct classification is around 2150. However, without the network, it is impossible to make the correct classification for all four data points using weights between 0 and 1.

Contribution of Group Members:

Felix: Final responsibility task 1; Collaborating on tasks 2 and 3.
Elvis: Final responsibility task 2; Collaborating on tasks 1 and 3.
Yuze: Final responsibility task 3; Collaborating on tasks 1 and 2.