Assignment 1
Introduction to Deep Learning

Peter Breslin - s3228282 Louis Siebenaler - s3211126

October 2021

Introduction

The MNIST dataset is a modified collection of handwritten digits often used in the training and testing of classification algorithms. A simplified MNIST set of 2707 images of digits represented by 256-dimensional vectors and associated labels is used for the development and evaluation of various classification algorithms for handwritten digits. Further exploration of neural networks was undertook and applied to the XOR problem of predicting the outputs of XOR logic gates given two binary inputs.

Task 1: Data dimensionality, distance-based classifiers

The first task involved the treatment of information stored in high-dimensional spaces and the use of dimensionality reduction techniques for data visualisation purposes. Moreover, two distance-based classifiers of varying complexity were developed and compared.
The images of digits used in this assignment depicted integer values from 0 to 9. Each of these 10 digits can be represented by a cloud of points in 256 dimensional space, each cloud consisting of the vectors of training images corresponding to each digit. By finding the center of each cloud, the distance from the vector that represents this image to each of the 10 centers can then be calculated. The center that is closest to this vector defines the label of the image, thereby classifying the digit. This procedure characterises the distance-based classifier algorithm and is outlined below:

```
d = np.arange(0,10)                          #digit d
Cd = np.zeros((10, 256))                     #Cd = cloud of points in 256-D space
for i in d:                                  #finding the center of each cloud
    ind = np.where(train_labels == i)        #identifying the vectors of training
    cd = train_set[ind]                       #images for given digit
    mean = cd.sum(axis=0) / len(ind[0])      #mean of each pixel coordinate
    Cd[i] = mean                             #building cloud of mean vector images
```

The distances between each cloud of mean centers were found for every digit and stored in a 10x10 array. The shortest distance between a given vector image to each of these clouds defines which classification the digit is most likely to belong to. Viewing the entries of our array can give us an idea of the expected accuracy of the algorithm. The shorter the distance between clouds, the more challenging it is to distinguish which cloud the digit should be classified to. For each digit, there are some distances that are relatively short compared to others. This indicates that our algorithm will have some difficulty differentiating between certain digits, meaning that it would be impossible to have an classification accuracy of 100%.
Digits 7 and 9 were found to be the most difficult for the algorithm to distinguish between. Intuitively, this makes sense as these digits can look very similar when handwritten. There was

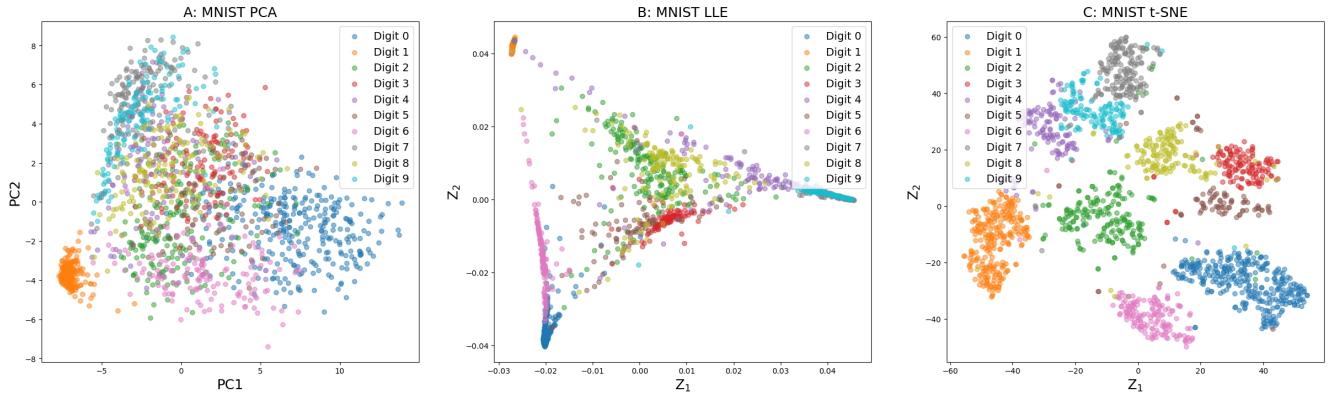

Figure 1: Visualisations using dimensionality reduction algorithms: The Principle Component Analysis (PCA) is shown in Plot A (*left*), the Locally Linear Embedding (LLE) method in Plot B (*middle*), and the t-distributed Stochastic Neighbour Embedding (t-SNE) approach in Plot C (*right*).

also difficulty found in separating digits 3, 5 and 8. Again, this is understandable as these digits have a similar curvatures when handwritten.

Dimensionality reduction algorithms

Data reduction algorithms allows us to more easily visualise high dimensional data and reduce the complexity of the model. In our case, this allows us to visualise the 256 dimensional MNIST training data in just 2-Dimensions. The Principal Component Analysis (PCA) scheme linearly transforms the data into its components of greatest variance. The Locally Linear Embedding (LLE) method measures how each training instance linearly relates to its nearest neighbours and then searches for a low-dimensional representation of the data where these local relationships are best preserved. The t-distributed Stochastic Neighbour Embedding (t-SNE) approach transforms similarities between data points to probabilities and reduces dimensionality by keeping similar instances close and dissimilar instances apart.

The visualisation results from applying these different dimensionality reduction algorithms to our MNIST dataset are shown in Figure 1. The strongly correlated orange digit 1 cluster observed in our PCA analysis illustrates the precision our algorithm has for classifying this digit. The distance between this cluster and other data points is relatively large, furthering the point that the algorithm distinguishes this digit more easily than others. A weaker correlation is seen with the remaining digits, particularly with digit 4. These data points are quite spread out, implying that this data is unstructured and similar to other digits. Furthermore, digits 5, 3 and 8 strongly overlap with each other, agreeing with earlier comments regarding the physical similarities between these handwritten images.

In our LLE analysis, strong correlations are observed for a number of digits. These are conveyed by the tight clustering of data points (such as those relating to digit 0 and 1), as well as the straight line clustering (e.g. digit 6). The further the data points are from one another the more dissimilar the images are to each other, meaning that the algorithm will likely not mix those digits up. The overlapping structure of points observed in the center indicates that the algorithm will have difficulty in distinguishing these digits.

The result of our t-SNE analysis conveys the effectiveness of this algorithm for dimensionality reduction and visualisation purposes. The data for each digit is well clustered, illustrating the algorithm's capability for separating the images. Moreover, the distribution agrees with our distance

matrix in that the cluster of data points for digit 9 is observed to be closest to that for digits 4 and 7. This highlights that our classifier will find distinguishing between these digits challenging, once more agreeing with our intuition regarding the similar shapes of these handwritten digits.

Distance-based and K-Nearest-Neighbour (KNN) classifiers

A simple distance-based classifier (DBC) was implemented and applied to all points in our set of training data. The percentage of correctly classified digits was found to explore the accuracy of this approach, resulting in an accuracy of 86.4%. As expected, this classifier is not completely accurate due to the similarities of data between some digits. When generalising the classifier by implementing it on the test dataset, an accuracy of 80.4% was obtained. This suggests that our algorithm slightly overfitted the data.

KNN predicts classifications by computing the distances between the training points and labels (i.e. the potential classes) and selecting the K number of points which are nearest to the labels. The algorithm then finds the probabilities that given labels belongs to given groups of K data points and assigns a classification based on the highest probability calculated. This algorithm was implemented using the *sklearn* package and accuracies of 96.6% and 90.8% were found for the training and test data, respectively. The greater accuracy of this algorithm compared to our simple DBC is expected as KNN better filters out false classifications by grouping and averaging the data points suspected to belong to a given label.

Confusion matrices were constructed for the simple distance-based and KNN classifiers for both the training and test datasets. These matrices summarise the number of correct and incorrect predictions made for each digit. From the number of incorrect classifications in our confusion matrices, it was seen that the simple DBC found digit 5 (76.1% accurate) most challenging to classify when using the training data and digit 2 (68.3% accurate) when running with the test data. In comparison, the KNN classifier found digit 8 (89.6% accurate) most difficult to classify when run with the training data and digit 5 (63.6% accurate) when run with the test data. This agrees with our previous comments regarding the difficulty faced in distinguishing these digits due to their physical similarities when handwritten.

It should be noted that the amount of data for each digit varies. This may introduce a bias and lead to poor generalisation due to a lack of data available for a given digit. This explains why the KNN classifier had a lower accuracy rate than the simple DBC when classifying their 'most difficult' digit (2 and 5, respectively).

Task 2: Multi-class perceptron algorithm

It is possible to improve the classification of handwritten digits by implementing a multi-class perceptron algorithm and use this to train a single layer perceptron. The choice of a single layer perceptron is motivated by the fact that this simplified version of the MNIST dataset consists of points that are linearly separable. As a result, the generalised perceptron algorithm will converge after a finite number of iterations and separate all digits.

In our implementation, the perceptron consists of 10 output nodes, each having 257 inputs and 1 output. The inputs correspond to the 256 dimension coordinates of a digit vector and 1 input bias. The output node with the strongest activation (i.e highest output magnitude) yields the classification of a digit vector. In our case, we have assigned the n^{th} node to the digit n. For example, if node 5 has the strongest activation, the classification for the handwritten digit is 5. The learning algorithm is outlined on the next page:

```
z = np.dot(w,y_in)+b   #w includes the weights, b includes the biases
                        #and y_in is the input
y_out = np.argmax(z)
if y_out != label:      #checks whether there has been a misclassification
    i_up = np.where(z > z[int(label)])   #selects the nodes that have been
                                          #activated too much
    w[i_up] -=  y_in
    w[int(label)] += y_in
```

In summary, the perceptron algorithm decreases the weights of the nodes that exhibit too strong of an activation and increases the weights of the node that should have the strongest activation. This learning process is repeated for each training digit until no training examples are misclassified anymore. In our case, this was achieved after approximately 40 - 60 epochs, where a epoch corresponds to one training iteration through the entire training set. The range in convergence time is a consequence of the random initialisation of the input weights. Figure 2a shows the training evolution of one of our single layer perceptrons. This illustrates that the perceptron algorithm quickly converges and is capable of reaching an accuracy of 100% on the training set.
After training the perceptron, its performance on the test set of digits was assessed. The accuracy on this varies within the range of 86 - 88 %. This is better than the naive distance-based approach, however slightly worse to the KNN classifier. One likely reason for the performance on the test set being worse is that the perceptron has overfitted the training set. As a result it exhibits a deteriorated performance on data that it has not seen yet and does not generalise perfectly. Studying the confusion matrix on the test set classification reveals that similar to the previous classifiers, the perceptron has difficulties classifying the digit 5 which is due to its lack of data in the training set.
It should be noted that by adding more layers to the perceptron, one should expect the performance on the test set to improve. However, this is associated with a computational time cost.

Task 3: The XOR Network and Gradient Descent Algorithm

A simple neural network consisting of one hidden layer has been trained to emulate a XOR gate. The possible inputs for this gate are the vectors (0, 0), (0, 1), (1, 0), (1, 1) which have corresponding targets 0, 1, 1, 0. Thus, the training data for this neural network consists of these 4 input vectors. In our implementation, the architecture of the network is as follows: 2 input nodes, 1 hidden layer consisting of two nodes, 1 output node. Each non-input node has three incoming weights, this includes the bias weight which is initialised to a value of 1. We initially use a sigmoid activation function ϕ at all non-input nodes. This choice is motivated by both the function varying smoothly in a range from [0, 1] which includes all target output values of a XOR gate and its simple derivative which is defined by $\phi' = \phi(1 - \phi)$. Saturation of the sigmoid function which occurs around -7 and 7 will not be an issue since we decide to interpret any output above 0.5 as 1 and below 0.5 as 0. The mean squared error (MSE) is employed as the error function in our network which is required for the gradient descent algorithm, the learning method of the network. The general formula for a gradient descent step, i.e the update rule for the weights is:

$$w = w - \eta \nabla_w \text{MSE}(w), \tag{1}$$

where w is a weight value of the network and η is the learning rate. For each training iteration Equation 1 is applied to every weight.

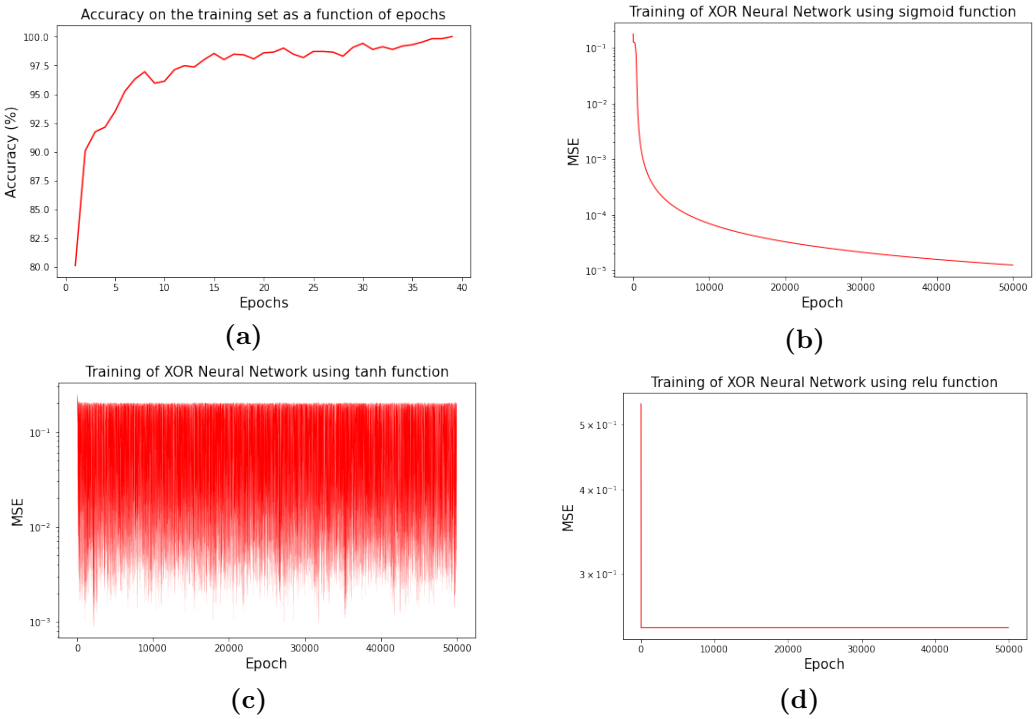

Figure 2: Plot **(a)** shows the single layer perceptron training evolution. Plots **(b)**, **(c)**, and **(d)** show the mean-squared error as a function of time for a XOR network using 3 different activation functions.

Figure 2b shows the training over 50000 epochs of a XOR neural network that performs very well. For this, the weights were randomly initialised and a learning rate of $\eta = 1$ was applied. Throughout the training, the percentage of misclassified inputs was only 0.30%. It can be recognised that the MSE rapidly decreases initially and has not converged yet after 50000 epochs. However, after 50000 epochs the network is perfectly able to emulate a XOR gate, hence further training is unnecessary. We have experimented with various learning rates η and we found that $\eta = 1$ works very well for this problem. It is important to emphasise that some random initialisation of weights and strategies involving only zeros or ones for weights did not converge towards a XOR function. Thus, choosing the right set of initial weights is crucial when training the network.
The 'lazy' approach was also employed in which random weights are generated until the network is able to compute the XOR function. Repeating this approach a few times, we found that the number of random weight reinitialisations varies between 60000 - 400000. While this approach was fast (speed comparable to the that of gradient descent) for this simple problem, it is strongly advised to use gradient descent as the learning algorithm for more complex problems.
Lastly, alternative activation functions (tanh and relu) were tested for the XOR network by applying them to each non-input node. It was found that both tanh and relu are not suitable for emulating a XOR function, as can be observed in Figures 2c and 2d. The training using the tanh function is very unstable and does not converge. This is related to the fact that the derivative of tanh quickly saturates hence preventing the weights to update when necessary (this is also referred to as a *vanishing gradient problem*). Furthermore, the range of tanh is [-1, 1] which is undesirable given that the target outputs are 0 or 1. The MSE remains constant throughout the training when using the relu function (expect for the first training iteration). Applying the relu function to all non-input nodes causes the network to behave as a linear function as relu is a linear function itself. However, the XOR problem is not linearly separable and as a result cannot be solved by such a network. Thus, it does not make sense to apply the relu function to every node.