Introduction to Deep Learning Assignment 0

Chloé Tap, Evan Meltz, Giulia Rivetti (Group 36)

September 2023

Contributions

Chloé Tap: Task 1 code, task 1 report.
Evan Metlz: Task 2 code, task 2 report.
Giulia Rivetti: Task 3 code, task 3 report.

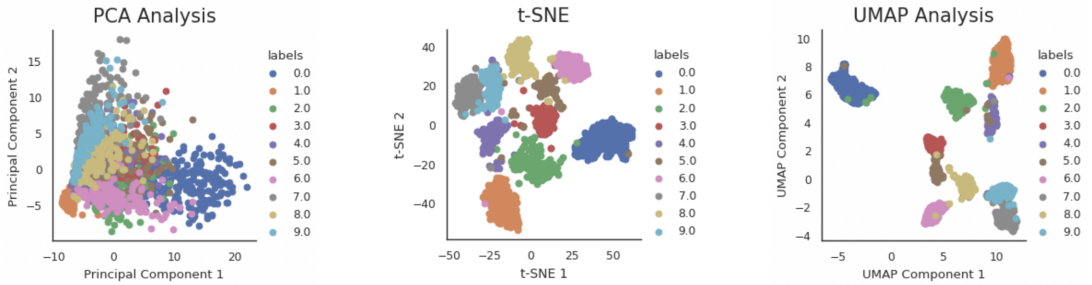
Task 1: Data dimensionality, distance-based classifiers

1. Digit Centres

The goal was to calculate the centres of each digit "cloud" using the training data provided. Using dataframes to store our data, we sorted them by digit label and for each group of vectors corresponding to the same digit, we calculated a mean value for each co-ordinate which corresponded to a 256-dimensional "centre" vector for each digit. The distances between these centres were calculated to determine which digits are most difficult to distinguish. The top 5 resulting pairs were: (7,9), (4,9), (3,5), (8,9) and (5,6). These results were to be expected as by eye their shapes are very similar.

2. PCA, U-MAP, T-SNE

We compared 3 dimensionality-reduction algorithms on our training data: Principal Component Analysis (PCA), t-distributed Stochastic Neighbor Embedding (t-SNE), and Uniform Manifold Approximation and Projection (U-MAP). Each of these algorithms aims to reduce the size of the dataset while preserving the global and local structure of the original data. We constrained our training set dimensions to 2-dimensional (2D) space for each algorithm, i.e. from (1707, 256) to (1707, 2).
PCA converts a set of correlated features in the high dimensional space into a series of uncorrelated features (the principal components) in the low dimensional space. To goal of PCA is to drop the features with lowest variance, so that we end up with only the features that preserve the most information (i.e. those with high variance).
t-SNE aims to reduce dimensionality while also keeping similar instances close and dissimilar instances apart. In particular, it calculates the joint probabilities between the data points for high and low dimensional space and determines a suitable 2D embedding which minimizes the difference between these probabilities using gradient descent. In essence, t-SNE groups together similar instances in order to map to the required dimensionality reduction of our data. U-MAP works similarly to t-SNE, as it attempts to find a low-dimensional representation which preserves relationships between neighbours in high-dimensional space. The embedding in the low-dimensional (2D) space is achieved using stochastic gradient descent, which minimizes the discrepancy between the pairwise distances in the high-dimensional and the low-dimensional space.
The visualisations of each algorithm for the MNIST training data are shown in Figure 1. In all cases, there is a clear overlap of digits 7 & 9, which confirms our top result from part 1. PCA is unable to separate features for high-dimensional data, due to its linearized method. t-SNE and U-MAP are most effective for visualization purposes, where the labels are able to clearly be distinguished from one another.



(a) PCA (b) t-SNE (c) U-MAP

Figure 1: Visualisations of PCA, t-SNE and U-MAP on MNIST training data. PCA uses a linear method which fails to separate the clusters of labels sufficiently, whereas the non-linear t-SNE and U-MAP approaches allow for a more distinct separation of labels as shown.

3. Nearest Mean vs K-Nearest Neighbours

We developed a simple distance based classifier using the "centres" associated with each digit labelled 0-9. Iterating over all vectors and centres, the Euclidean distance between each vector and each centre was evaluated. The shortest distance was found and the label of the centre associated with this distance calculation was assigned to the vector. In addition, we implemented the K Nearest Neighbours (KNN) algorithm from Scikit-Learn. For this classifier, it determines the Euclidean distance from each vector to its "nearest neighbours" from the training data and, based on the training labels, votes on the most popular neighbour label and then assigns it to that vector. In our case, we chose k=3 neighbours as it yielded a reasonably good cross-validation score.

For both classifiers, we compared the training and test set results using confusion matrices which indicated the number of correctly predicted digits by the chosen algorithm. We found that the accuracy of the nearest mean classifier for the training set was ∼86% while that of the test set was ∼80%. As expected, the accuracy is not very high as overlaps of the digit centre "clouds" cause vectors to be easily misclassified as a result of small inter-class separations. For the KNN approach, we found ∼98% of the training data was correctly classified and ∼91% of the test data was correctly classified. This is a substantial improvement on the nearest mean classifier. Both classifiers are very simple in their construction, and manage to classify digits reasonably well. However it was noticed that the digit 5 was most difficult to classify correctly in our test data using both classifiers, with an accuracy of only 67-69%. We also confirmed our finding from part 1 that digits 7 & 9 are very difficult to differentiate in our nearest mean classifier. Hence the classifiers are not very well suited for small inconsistencies such as these.

Task 2: Implement a Multi-Class Perceptron Algorithm

In this section we were required to develop from scratch a multi-class perceptron training algorithm which should be used to train a single-layer perceptron. The network will be trained and tested on a simplified version of the MNIST dataset which consists of 1707 training images and 1000 test images. The perceptron will consist of 10 nodes, one for each digit from 0-9 and have 256 inputs, being the data from the trainIn dataset plus a bias of 1. - see Figure 2.

This data is taken from CSV files and stored in matrices. The weights of the network are randomly initialised and stored in a 256x10 sized matrix. A bias of 1 is then appended to each row of input data. Using a nested for loop with the outer loops iterations being set to the number of epochs and the inner loops iteration being set to the number of samples, the dot product of the input data matrix and the weight matrix is then calculated, the argmax() function is then used to then get the output node with the strongest activation and lastly using gradient descent the weights are updated. Once the loop ends the final updated weights are returned.

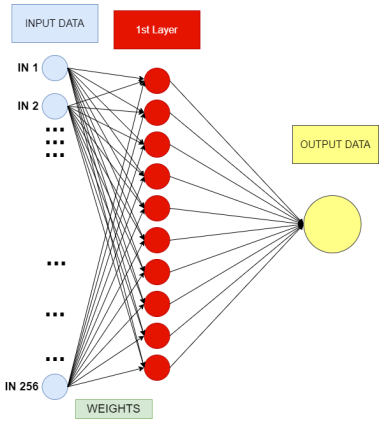
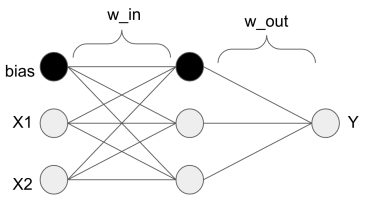


Figure 2: Multi-class Single-Layer Perceptron.

With these weights the predicted output values on the train and test data are calculated. These are then compared with the actual output values and the results are displayed. These results show that using this method yields a 100% accuracy with identifying the train data with at least 100 epochs. However, with the test (unseen) data, the accuracy after several runs of the program peaks at around 87%. As mentioned above the accuracy of the K-Nearest method yielded an overall accuracy of about 91%. So, we can therefore see that while both methods are very close in their accuracy of classifying digits, the K-Nearest method is superior. These findings are not surprising as I believe that using a single-layer perceptron severely restricts its potential for a higher accuracy.

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

In this task we implemented a neural network that computes the XOR function, which takes two binary inputs and return a binary output, as shown in Table 1. In particular, this neural network is characterized by two inputs, two hidden nodes and an output. Each node in the hidden layer has three incoming weights (two for the inputs nodes and one for the bias) and one outgoing weight which is connected to the output node - see Figures (a) and (b). The first step to implement the XOR network is to initialize the weights and biases.



(a) Neural network structure.

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

(b) XOR truth table.

There are several techniques that can be used to do that, and we will explore some of them later; biases are initialized to 1. We employed 5000 iterations in order to properly train the network. While iterating, we applied the forward propagation method, in which the inputs are propagated through the network. At the end of this process we get a prediction y. Then, we compute the mean squared error function, which measures the average squared difference between the observed values and the predicted values. The former corresponds to our target output (0, 1, 1, 0), given as input x (00, 01, 10, 11); the latter represents the predictions computed by our network. At each iteration we compute the mean squared error, storing it in a vector called “losses”. The mean squared error is used to update the values of the weights and biases. This is the concept behind the gradient descent algorithm, whose purpose is to find the input that minimizes the output of the loss function. This can be done by changing the values of the weights and biases based on the following formula:

$$weights = weights - \eta * grdmse(weights)$$

where η is the learning rate, a hyperparameter that establishes the pace at which the network is learning. The $grdmse(weights)$ denotes the function computing the gradient for each mean squared error that we have previously computed. Notice that it's important to keep track of the mean squared error as well as the number of misclassified inputs, because they give us information about the performance of the network. In a properly trained network the mean squared error needs to decrease with the number of iterations and indeed this is what happens with our network.
We observed what happens when using different initialization strategies for the weights and various learning rate values. We tested the network with three different values of leaning rate: 0.1, 0.5 and 0.9, observing that, in general, for a given initialization strategy the network works better with a higher learning value, because in this case the network needs less iterations to be trained. Focusing on the case where the learning rate is set to 0.9, we can observe that the best accuracy for out network can be obtained by employing the random and Xavier initialization strategies. The He initialization technique does not provide a good accuracy because it is meant to be used on larger networks, whereas our neural network only has two inputs. Then, we observed how many iterations are needed so that the network can correctly compute the XOR function. By using a random initialization technique for the weights, we notice that by setting the learning rate at 0.1, we need three iterations to get the correct result of the XOR function. If we increase the learning rate up to 0.5, the function is correctly computed at the first try. Therefore we can deduce that, as also observed before, the random initialization works better with higher values of learning rate. Finally, we employ different activation functions to see how the affect the accuracy of the XOR network and we compare them by using the same weights vector, which has been randomly generated. By running the program, we notice that the best accuracy can be obtained with the tanh activation function, whereas the sigmoid function turns out to be less accurate. ReLU is not the best activation function to use in this case, because it works better with the he initialization technique but, as explained before, that is not the best option with our small neural network.