IDL Assignment 0

Jasper Mens (s2015242) and Laurens Müller

September 2023

Introduction

This report outlines our approach and discusses our experimental results for assignment 0 of Introduction to Deep Learning. In the first two tasks we concern ourselves with the classification of hand-written digits, and in task 3 we build an XOR neural net.

Contributions: T1 Report & Code: Both, T2 Report & Code: Jasper, T2 Report & Code: Laurens

Task 1: Distance-based classifiers

In this task, we set out to classify a simplified version of the MNIST dataset based on dimensionality reduction, and develop an intuition for point clouds in high-dimensional spaces. The dataset consists of a series of 16×16 images, which we will treat as points in a 256-dimensional space.

1. We can interpret the training images as clouds of points C_d, where d is the digit the images represent. We can use the distances between the centers of these clouds as an indication of which digits may be particularly easy or difficult to distinguish from each other. The centers of the clouds represent the average image representing the digits. Out of interest, we have included these images in figure 2. The distances between each of the centers can be seen in figure 1. We see that the respective centers of the pairs (7,9), (4,9), and (3,5) are closest together, likely making them difficult to separate. The resemblance can also be recognized by eye in figure 2, where for instance 3 and 5 have very similar bottom halves, and the bottom two-thirds of 9 and 4 look quite similar.

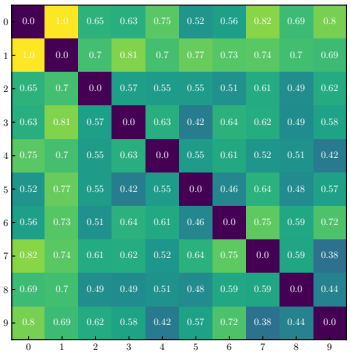

Figure 1: Normalized distances between the centers of C_d. Note the (logical) diagonal symmetry.

2. We now apply three dimensionality reduction algorithms: PCA, U-MAP and T-SNE (from the PyPI packages `scikit-learn` and `umap-learn`). The results are shown in figure 3. We find that PCA fails to separate the digits neatly, apart from picking out C_1. There is some agreement between the degree of overlap and the distances between the means found before, but the spread of the points tends to be larger than the inter-mean distances. Interestingly, the pair of (4,8), whose means are quite well-separated (see figure 1), now shows significant overlap and minimal mean separation. We conclude that, for this particular pairing, PCA only seems to make classification more difficult. The U-MAP-algorithm produces a much cleaner result: there is far less overlap than with PCA, and the (8,9) pair seems to be more separated than in the distance-between-centers approach above. Finally, the T-SNE algorithm perhaps performs slightly worse than the U-MAP (producing looser groupings), while still handily outperforming than PCA. In general, these visualisations agree with our aforementioned intuitions. We see that the problematic pairs mentioned above remain close together through all three reduction algorithms (e.g. C_4, C_7, and C_9 stay close/joined together, while C_0 and C_1 never overlap).

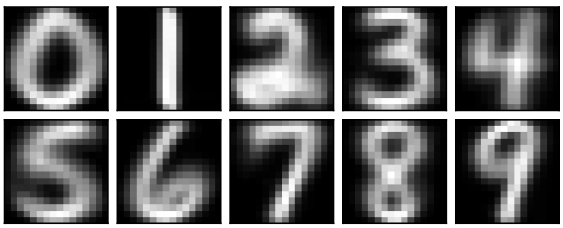

Figure 2: Images representing the centers of the point clouds C_d of digit d, i.e.: the average training image representation of digit d.

3. We now implement a Nearest Mean Classifier (NMC) using our mean images, and evaluate its performance on the test set. The resulting confusion matrix can be seen in figure 4.

   We see that the most problematic pairings are (0,6) and (2,8). This is somewhat unexpected given the above analysis, although the corresponding clouds do appear relatively close together in the U-MAP and T-SNE results.

   The NMC correctly classified 86.4% of the training set, and 80.4% of the testing set.

4. We now implement a K-Nearest-Neighbor (KNN) classifier using the `sklearn` package. We find that the performance of this classifier depends weakly on the number of neighbors (k), but that we get the best results by using $k = 1$. The corresponding confusion matrix can be

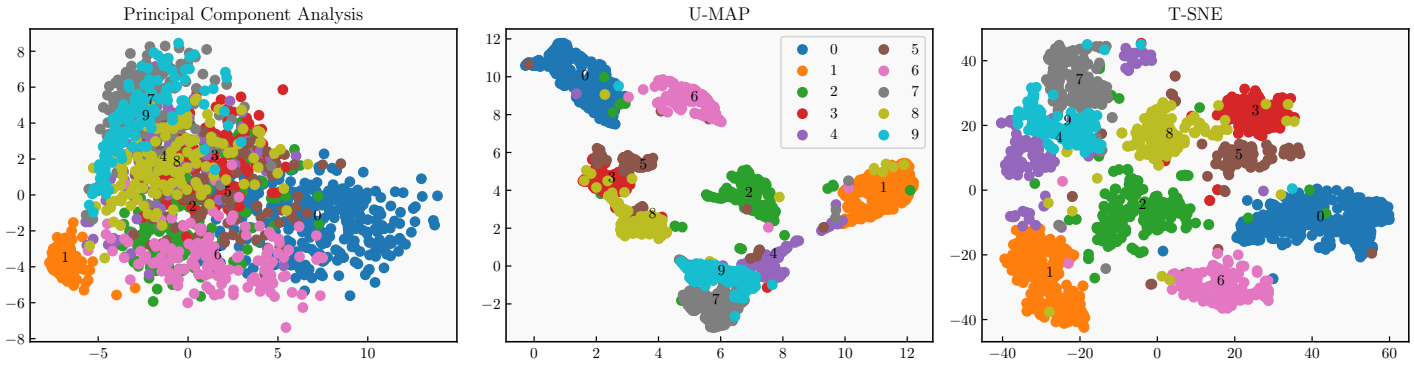

Figure 3: Dimensionality reduction output: PCA (left), U-Map (center) t-SNE (right). The cloud centers are marked with the corresponding digits.

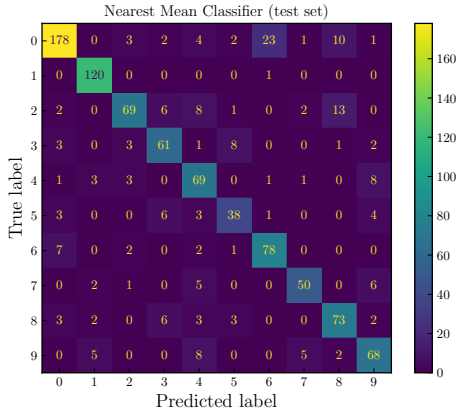

Figure 4: Confusion matrix for the nearest mean classifier: element (i,j) is number of images of digit i classified as digit j. Note that the matrix is not symmetric. For instance, a 2 is more likely to be labeled as an 8 than vice versa. This makes intuitive sense.

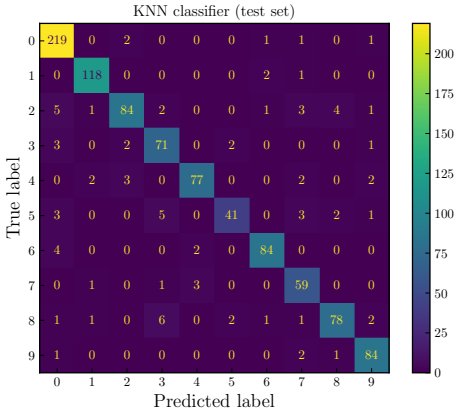

Figure 5: Confusion matrix k-nearest mean classifier: element (i,j) is number of images of digit i classified as digit j

seen in figure 5. We see that the 'hotspots' of the NMC are absent here, and that the overall performance seems to have increased. And indeed, the overall accuracy is higher, with the KNN correctly classifying 91.4% of the test set.

Task 2: The Multi-class Perceptron

The objective of this task is to write and train a basic single-layer multi-class perceptron to classify the same dataset as in Task 1.

We use 10 nodes, corresponding to the possible output classes (digits 0 through 9). Each node has 256 pixel inputs and 1 bias input (set to unity), and, accordingly, 257 weights. The dot product of the input and the weights produces a floating-point activation value that can be interpreted as a measure of 'confidence' the node has that the image is a member of its class. Passing an image through all ten nodes, the output of the perceptron is the class whose node has the strongest activation.

We start with weights drawn from a uniform random distribution and begin training, following the standard perceptron training scheme.

Letting our perceptron train until its training set accuracy reaches 100% (which is attainable for this dataset, as it consists of linearly separable patterns) takes around 2000 epochs, and less than five seconds on Jasper's laptop. As there is some degeneracy in terms of weight combinations capable of producing 100% training accuracy, we repeat the experiment a few times (with different random seeds for the weights) to gauge the reliability of the performance. The results can be seen in table 1.

Perceptron	Epochs	Test accuracy
1	2059	87.8%
2	2108	87.9%
3	2116	88.2%

Table 1: Perceptron performance on the test set for different initial weights.

Figure 6 shows the confusion matrix for Perceptron 1. Once again, it is difficult to make out any particular features.

We can see that neither the test set accuracy nor the confusion matrix show much improvement when compared to the methods applied in Task 1. In fact, the perceptron performs slightly worse than the KNN scheme.

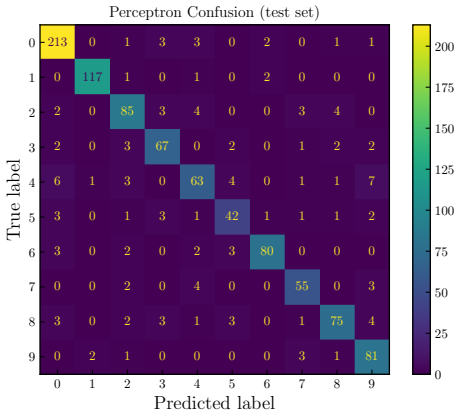

Figure 6: Confusion matrix for perceptron 1 (after training).

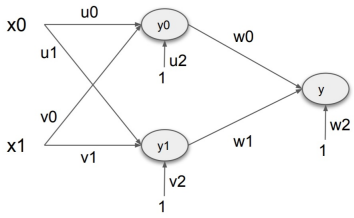

Figure 7: Naming convention of inputs, nodes and weights for the XOR-network

Task 3: XOR Network

xor_net

For this task we implement an XOR network using the naming conventions in figure 7. We begin by inmplementing the function `xor_net(weights, input)`, wich simulates a network with two inputs, two hidden nodes, and one output node, and returns the activation of the output node. We compute the node activations in the separate function `activations(weigths, inputs)`.

Error function

Next, we implement the error function. Since both the mean square error (MSE) and the number of misclassified inputs are used to study the gradient descent algorithm below, the function `mse(weights, inputs, targets)` returns both. This function calculates the mean error over all four possible inputs of the XOR-function.

Gradient of Error function

The next step is to find the gradient of the error function. We need a function that returns a vector of the same dimension as the weights input, using the following partial derivatives:

- $\frac{\partial E}{\partial w_i} = (y-d)y'\frac{\partial net}{\partial w_i}$
- $\frac{\partial E}{\partial u_i} = (y-d)y'y_0'w_0\frac{\partial net_0}{\partial u_i}$
- $\frac{\partial E}{\partial v_i} = (y-d)y'y_1'w_1\frac{\partial net_1}{\partial v_i}$

In these expressions, d is the target value, net the total input for y, net_0 that for y_0 and net_1 for y_1. The resulting function `grdmse(weights, inputs, targets)` also needs input and target data next to the weight parameter, and uses the aforementioned `activations(weights, inputs)` function. Furthermore, it uses the function `deractfun(activation)` to compute the derivative of the (sigmoid, tanh or ReLU) activation function:

- Sigmoid: $y'(net) = y(net)(1-y(net))$
- Tanh: $y'(net) = 1-y^2(net)$
- ReLU: $y'(net) = 1$ (if net ≥ 0) or 0 (if net < 0)

Gradient descent algorithm

Finally, we combine all of these functions and implement the gradient descent algorithm: `gradec(weights, inputs, targets)`. As discussed, `mse()` returns both the mean squared error and the number of misclassified inputs, which are both printed at the end of the simulation.

The amount of training data is constrained by the fact that the xor-operation only has four distinct inputs, so our program trains the network by repeatedly running the algorithm on those inputs. The algorithm stops training as soon as the MSE is below a threshold of 0.1, or, when training takes too long, more than 1000 training cycles). Finally, we specify the learning rate `eta`, activation function `actyp`, initialization type `inityp`, and learning strategy `typ` in the main function.

We conduct experiments by varying four aspects:

1. *Learning Rate* As expected, the program needs more training cycles when the learning rate is smaller (e.g. 0.1). Interestingly, our program does occasionally produce a sufficiently small MSE and sufficiently high accuracy while using a higher learning rate (10), but these nets tend to employ very large weights (range -10 to 10).

2. *Initialization* The program-function `initweights()` initializes weights with either a uniform or a normal distribution. Using a normal distribution tends to lead to slower training (more cycles) than when using a uniform one. The algorithm often even fails to find good weights (with both low MSE and high accuracy) within the maximum training time. One reason for this is that the normal distribution also produces strong outlier weights, which take more training cycles to be adjusted.

3. *Random trial-and-error* As an alternative to training, we have also tried random trial-and-error to find the desired weights. Here, weights are repeatedly randomized until the desired performance is reached. Unsurprisingly, we find that this method generally fails to reach sufficiently good weights in time. The gradient descent algorithm seems to perform much better.

4. *Activation functions* In addition to the sigmoid activation function, we have also tried to use tanh- and relu-activation functions. With both alternatives, the training fails to reach the desired performance, with large and extremely large weights respectively. The sigmoid function is the only one that achieves that goal.