COMPUTER VISION - ASSIGNMENT 1 - REPORT

SIMPLE 2-D CLASSIFIER

2.1)
As suggested, I've used the os.path.join() method to create a path in a subfolder. "Os" library was already imported. I have then saved the content of the npz file into the class members samples and annotations. The __getitem__ method recovers a single data sample, creating a tensor out of it.

2.2)
I have added a single linear layer with 2 parameters of input (cartesian coordinates) and 1 output (probability of a point belonging to cluster 1). The activation function scaling the probability in an (0,1) interval is already implemented in the forward method (torch.sigmoid).

2.3)
I have completed the run_training_epoch function. I have then run the script multiple times noticing that the accuracy varies significantly. The accuracy is mostly confined in the interval been 40% and 60%. The majority of the time it stands **between 47% and 53%** on the different epochs.
The network is basically a binary linear classifier, trying to separate the points with a straight line. It is clear that, by tracing a random line separating the data, we will approximately get 50% of the predictions right, since the points seem uniformly distributed around circles. The single layer perceptron is only able to classify linearly separable data.

2.4)
After having built the neural network, I have run the script and obtained an accuracy between **99.5%** and **100%**. A Multi-Layer Perceptron with 3 layers (2 hidden + 1 output) is able to use more complex shapes to separate the data, compared to single layer perceptron. MLP produces way better results dealing with non-linearities.

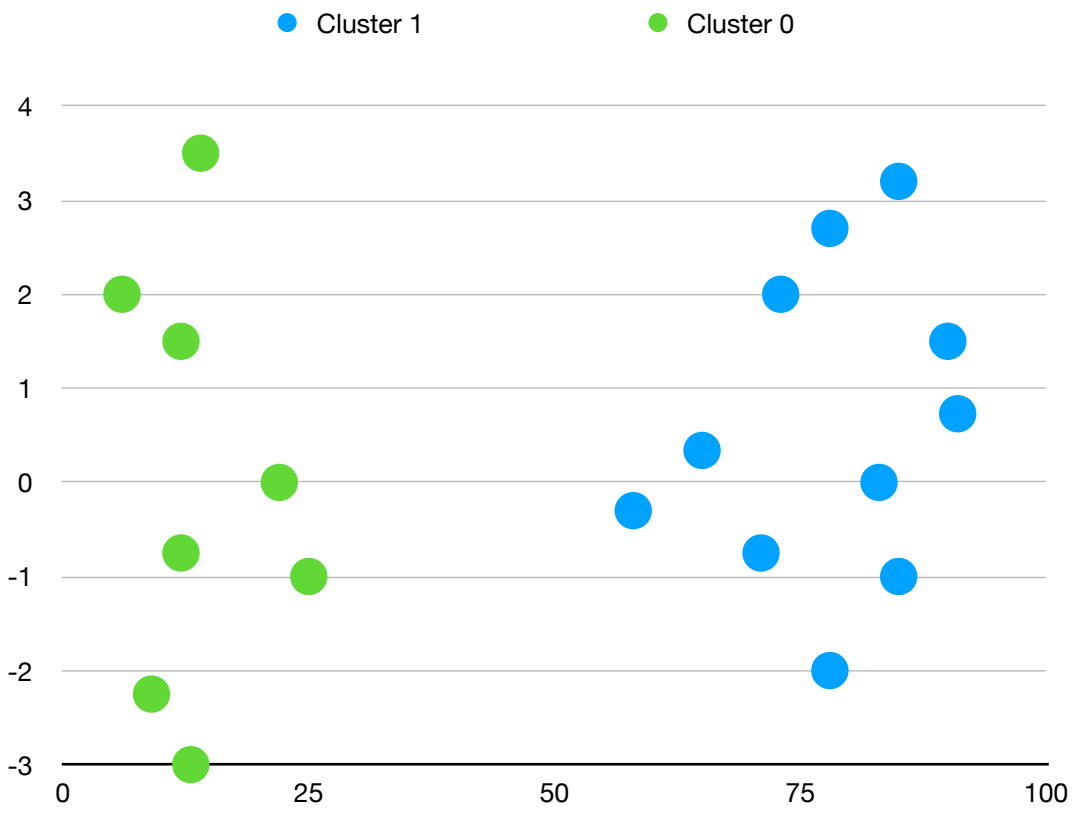
2.5)
Let X1, X2 be the features of the original dataset representing the cartesian coordinates of the point.

I have defined:

X1' = X1**2 + X2**2
X2' = X2/X1

Basically since the classification depends mainly upon the distance between the data point and the origin, I have defined a coordinate X1' that takes into account that distance.
X1' = (distance((X1,X2) ,(0,0)))**2
X2' = tangent of the angle formed by the the distance and the X1-axis.

Coordinate X2' takes inspiration from the polar coordinates.



The data within the plane (X1',X2') should look something like that: they are now linearly separable. Cluster 1 have a bigger X1' coordinate than Cluster 0 points.

After having run the script multiple times, with the single linear classifier I have obtained an accuracy in the last epoch **over 95%**. However, it might take a few epochs to get to that level of accuracy.

MNIST - CLASSIFIER

3.1)

The **normalize** function uses the following formula to normalize between [-1,1]:

Newsample = -1 + (sample * 2) / 255

where sample ranges from 0 to 255

3.2)
Run-training epoch has been implemented using the cross entropy instead of binary cross entropy. We have now a multi-class classification problem.

3.3)
Remark:
The code implementing the linear classifier is marked by #PROVISIONAL 3.3

The code implementing the MLP classifier with the hidden layer is marked by #FINAL 3.3

In order to run the script correctly, one should carefully choose which lines to uncomment.

PERFORMANCE:

The linear classifier with a single layer manages to reach an accuracy of around **90-91 %**

The MLP classifier with one hidden layer performs slightly better with an accuracy ranging **between 92% and 95%**, respectively in the first and in the last epoch.

3.4)

The testing accuracy of the convolutional neural network improves to **97 - 98 %**

3.5)

Number of Parameters for MLP with one hidden layer:

Firs of all, the ReLU activation function does not feature any training parameter. Only the 2 linear layers have a positive number of parameters.

For any linear, fully connected layer:

Let:

- **IN_Neurons** define the number of input neurons
- **OUT_Neurons** define the number of output neurons

Then, for any linear, fully connected layer the number of parameters is computed as:

#PARAMETERS_LC = OUT_Neurons * (IN_Neurons + 1)

Each of the input neurons is connected to all output neurons (every connection has to be counted as a **weight**). Moreover for each output neuron there is a **bias** adding to the linear combination of inputs and weights. That explains the previous formula.
According to the previous formula, the total number of parameters for each linear layer is:

Param_LC1 = 32 * (784 + 1) = 25120
Param_LC2 = 10 * (32 + 1 = 330

Thus, the total number of parameters is:

#Parameters = Param_LC1 + Param_LC2 = 25120 + 330 = 25450

So **25450** is the number of parameters of the MLP with one hidden layer.

Number of Parameters for the Convolutional Neural Network:

First of all, only the 3 convolutional layers as well as the final linear classifier feature training parameters. Neither the ReLU activation function, nor the MaxPooling operation feature any training parameter.

For a certain convolutional layer:

Let:
- **K** define the width/length of a square kernel
- **Cin** define the number of input channels
- **Cout** define the number of output channels

Then, for every convolutional layer the number of parameters is counted as:

Parameters_CONV = (K * K * Cin + 1) * Cout

Every input channel is convolved with a number of kernel equal to the number of output channels. Moreover there is one bias for every output channel. That explains the previous formula.
According to the previous formula, the total number of parameters for each convolutional layer is:

Param_Conv1 = (3 * 3 * 1 + 1) * 8 = 80
Param_Conv2 = (3 * 3 * 8 + 1) * 16 = 1168
Param_Conv3 = (3 * 3 * 16 + 1) * 32 = 4640

To complete the computation we must now consider the number of parameters in the final linear classifier:

Since there are 32 input neurons and 10 output neurons:

Param_LC = 10 * (32 + 1) = 330

Thus, the total number of parameters is:

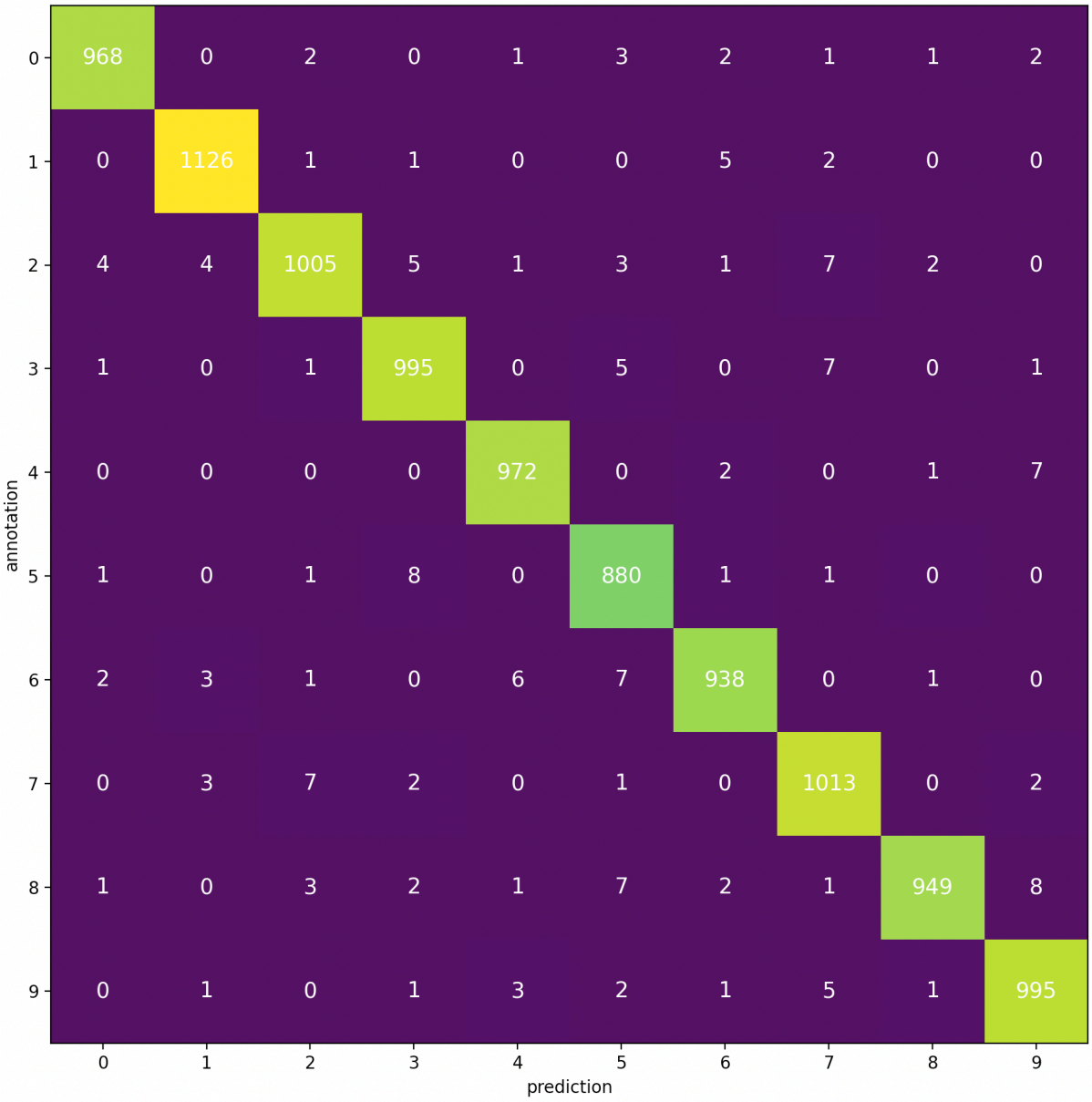
#Parameters = Param_Conv1 + Param_Conv2 + Param_Conv3 + Param_LC =

80 +1168+4640+330 = 6218

So **6218** is the number of parameters of the convolutional neural network.

3.6) CONFUSION MATRIX

Here's a screenshot of the confusion matrix, related to the Convolutional neural network.



The confusion matrix is almost diagonal, as we have not reached a 100% accuracy. The confusion matrix can help us define how accurate our model can baec for a specific output category.

For a certain digit X we can, for example, compute:

- **TRUE POSITIVES :** the number of digits X which where correctly classified. I evaluate them just by considering the numbers on the diagonal.

- **FALSE POSITIVES:** the number of “not X” digits which were incorrectly classified as X. I evaluate them by summing the elements on the X column, except the element on the diagonal.

- **FALSE NEGATIVES:** the number of digits X which our model failed to classify as X. I evaluate them by summing the elements on the X row, except the element on the diagonal.

Computing those parameters for the reported confusion matrix:

DIGIT	TRUE POSITIVES	FALSE POSITIVES	FALSE NEGATIVES
0	968	9	12
1	1126	11	9
2	1005	16	27
3	995	19	15
4	972	12	10
5	880	28	12
6	938	14	20
7	1013	24	15
8	949	6	25
9	995	20	14
TOTAL	9841	159	159

Of course the number of total false positives has to equal the number of total false negatives.

The **accuracy** is calculated by:

TOTAL TRUE POSITIVES / (TOTAL TRUE POSITIVES + FALSE POSITIVES) =

9841 / (9841 + 159) = 9841 / 10000 = **98.41 %**

After having plotted multiple confusion matrices, upon different trainings, I have observed that the network often fail to distinguish between “4” and “9”. Moreover “5” is often mistaken with “3” and “6”.