MNIST Dataset: Handwritten digits recognition.

The project is split into two parts. In the first part we train a typical CNN architecture to classify the MNIST dataset images of handwritten digits. The model architecture, the training process, the differences between the train and the validation loss and the results are discussed in detail. In the second part, the convolutional part of the network is discarded. Image features are no longer automatically computed by the convolutional layers and are manually calculated using the SIFT algorithm. The two cases are compared, and the final conclusions are made.

Part A:

In the first part we construct and train a **convolutional neural network (CNN)** for the classification of images, specifically for the recognition of handwritten digits. This problem is a supervised learning problem since we have a dataset described by labels at our disposal. The dataset used is the famous benchmark **MNIST** dataset. This dataset contains a total of 70,000 samples of handwritten digit images (from 0 to 9, i.e., a total of 10 classes) divided into two subsets. The first subset is the training set, which contains 60,000 images, and the second subset is the test set, which contains the remaining 10,000 images. The images are grayscale and have dimensions of 28x28. Each image is described by a label that indicates which digit it represents.

The first step is to download and load the two data sets (see .py file for details). Below are some randomly selected samples from each class.

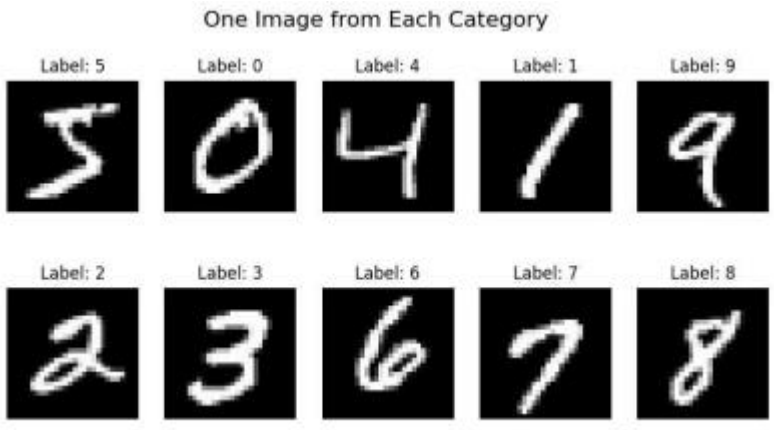

Figure 1The MNIST dataset

Next, we construct the convolutional neural network with the following architecture by defining the custom class CNN() in the manner presented in the corresponding section of the code provided in the repository.

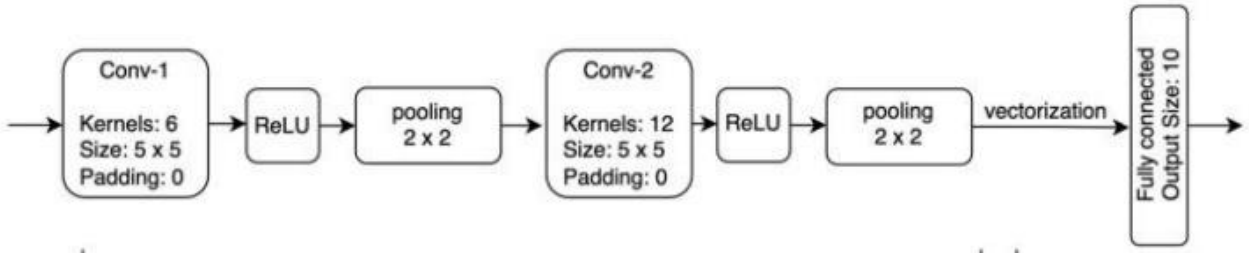

Figure 2 CNN architecture

As we can observe from the above image, the CNN consists of two parts. The first part is characterized by two convolutional layers with ReLU activation functions applied to their outputs, and two pooling layers. This part of the network is responsible for extracting features from the images. The second part of the CNN is a fully connected neural network that performs image classification. This part consists of two linear fully connected layers. Let's examine the structure of the CNN in more detail.

The first layer of the network is the first convolutional layer, characterized by 6 kernels where each kernel is a 5x5 filter since the images are grayscale and have only one channel. Because the number of kernels is 6, the output of conv-1 will be a set of 6 feature maps. Specifically, for an input image to conv-1, each kernel will convolve with it — without applying padding since padding = 0 — thus producing a 2D matrix at the output called a feature map. Due to the lack of padding (stride = 1), the matrix dimensions will be smaller than the original image dimensions. Therefore, while the input to conv-1 was (1, 28, 28), the output will be (6, 24, 24). The output is then fed into a **ReLU** activation function to introduce non-linearity into the network so that it is not limited to learning only linear patterns. Afterward, the ReLU output, which has the same dimensions as its input, passes through the pooling layer. In the pooling layer, downsampling —specifically max pooling— of the 6 input feature maps occurs using 2x2 filters (with padding = 0, stride = 1), resulting in an output of dimensions (6, 23, 23).

This process is repeated one more time, with the only difference being that the second convolutional layer has 12 kernels instead of 6, and each kernel is not just a 5x5 filter but a set of 6 filters of 5x5, as the input to conv-2 has 6 channels instead of 1. The input is again convolved with each kernel or more accurately, each channel of the input is convolved with the corresponding channel of each kernel, and the results of these convolutions are combined to form the output of a kernel. The result of this process is an

output of dimensions (12, 19, 19). After this output passes through the ReLU and the pooling layer, the output of the first part of the network (which has dimensions (12, 18, 18)) is obtained. As previously mentioned, this part is responsible for extracting features from the images to be used as input to the classifier, the second part of the network. Before the output of the second pooling layer is fed into the classifier, vectorization is performed to convert it into the appropriate form, a 1D vector of 12*18*18 values. To be precise, since in each forward pass not just one image but a set of images (the so-called mini-batch) is used, the input to the classifier is not a 1D vector of 12*18*18 values but a 2D matrix of 64 such vectors, one for each image in the mini-batch (each mini-batch consists of 64 images).

Concluding the analysis of the network's structure, let's also study the structure of its second part. The first layer of the classifier consists of 128 neurons, and the second layer, which is the output layer, contains 10 neurons, one for each class. The ReLU activation function is used in the first layer, while the softmax function is used in the second layer to convert the output values of the 10 neurons into probabilities. The network predicts the class described by the highest probability each time. In practice, during the construction of the network, the softmax function is not used as it is integrated into the cost function that we use.

To enable the training of the implemented network, we construct the train_model() function. This function takes as input the model, the set of mini-batches with training images, the set of mini-batches with validation images, the cost function, the optimizer, and the desired number of epochs. For the purposes of this exercise, **cross-entropy** is used as the cost function, and a **SGD** (Stochastic Gradient Descent) algorithm with a learning rate of 0.01 is used as the optimizer. The process implemented by this function is as follows:

Each epoch (considered complete when the network has seen the entire dataset, i.e., all mini-batches) is characterized by two phases: the train phase and the validation or evaluation phase. Initially, the train phase is implemented, during which the mini-batches containing training images sequentially enter the network one after the other. For each mini-batch, two passes occur in the network: the forward pass (left to right) where the training loss for the given mini-batch is calculated by summing the costs for each image in the mini-batch, and the backward pass (right to left) where the SGD optimizer is used to optimize the cost function with respect to the network parameters (the weights of the convolutional layer kernels and the weights and biases of the classifier layer neurons). After the backward pass, the network parameters are updated. Subsequently, we move to the second phase, where the same process occurs, but this time, the mini-batches containing validation images sequentially enter, and only the forward pass is implemented to calculate the validation loss for each mini-batch. This completes one epoch. For each completed epoch, we calculate the training and validation loss. These values are obtained by averaging the cost function values across all training and validation mini-batches,

respectively. The train_model() function returns the model with the highest accuracy based on the validation dataset, and two lists of the training and validation loss values for each epoch.

Having the train_model() function at our disposal, we can train and use the CNN. The model was trained four times, each time using a different percentage of the initially available training dataset (the set of 60,000 images). Specifically, the percentages of 5%, 10%, 50%, and 100% were examined. Each percentage of the initial set is randomly divided into two subsets: the training image subset and the validation image subset (80% and 20% respectively). Each of these subsets is divided into mini-batches of size 64.

The **validation process** is an integral part of a model's evaluation. The model evaluation could be exclusively based on the training loss and the model's accuracy on the training image set, which is not recommended for the following reason. As the model trains, the training loss will certainly decrease as the model learns the training images better and better. However, a good model is not one that can accurately recognize only the images on which it has been trained, but to be considered a truly good model, it must be able to generalize well and accurately recognize images it has not been trained on (images belonging to the training image classes). The more a model is trained, the more likely it is to overfit, meaning it will recognize the training images with high accuracy but fail to generalize. Therefore, if we want to avoid overfitting, we cannot rely on the training loss. For this reason, the validation process is implemented, where a subset of images on which the model has not been trained is used to periodically check its performance. For this reason, the train_model() function selects and returns the model with the highest accuracy on the validation dataset. In conclusion, the validation loss is a better indicator of the network's performance compared to the training loss, which cannot characterize an overfitted model as 'bad' as it should.

Below we can see the training and validation loss curves for the four cases examined.

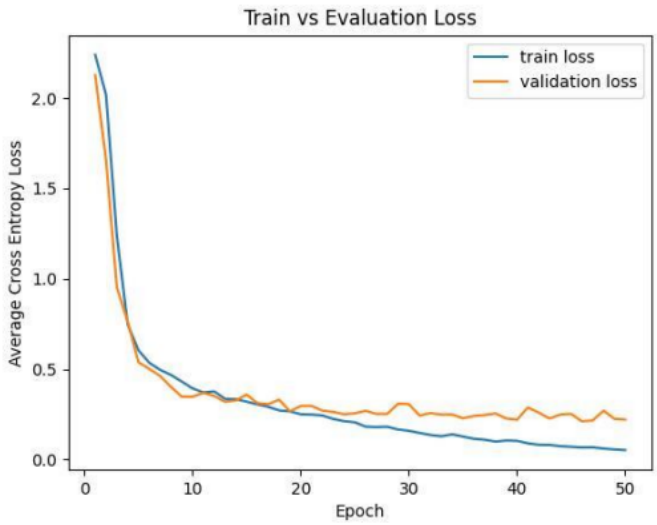

Figure 3 Train loss - Validation loss vs Number of Epochs. 5% of the train set used.

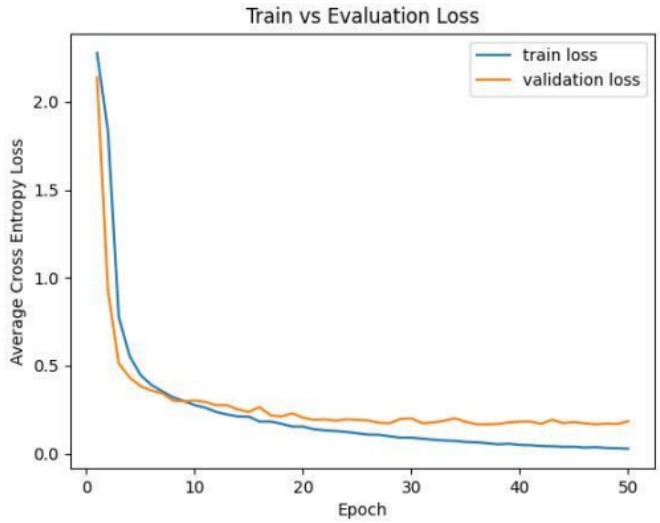

Figure 4 Train loss - Validation loss vs Number of Epochs. 10% of the train set used.

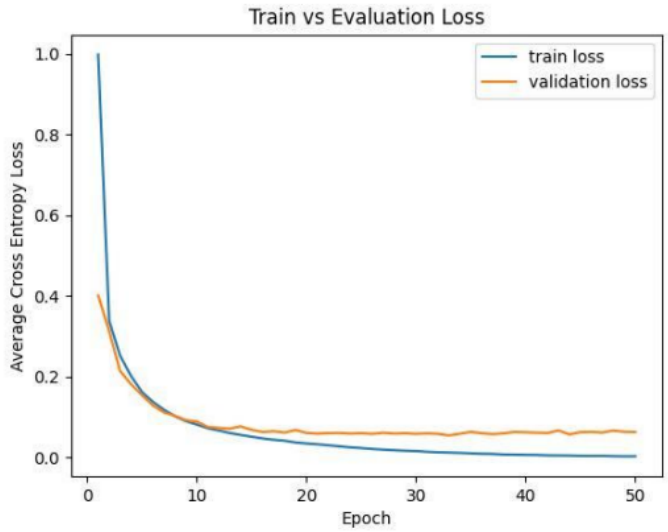

Figure 5 Train loss - Validation loss vs Number of Epochs. 50% of the train set used.

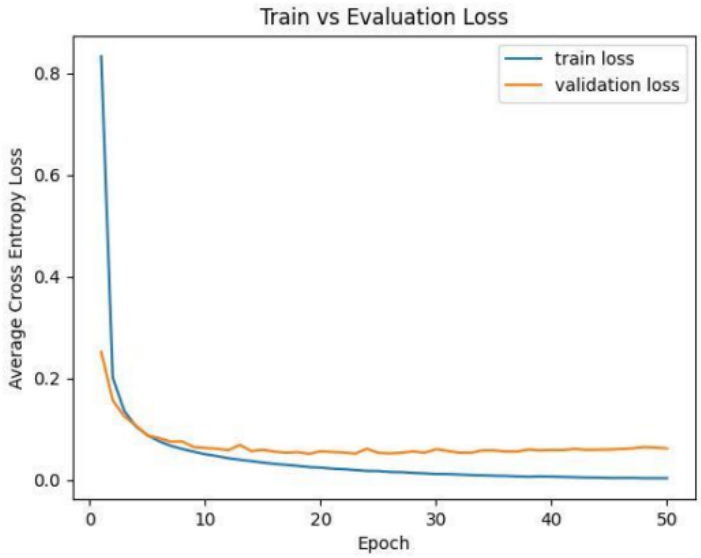

Figure 6 Train loss - Validation loss vs Number of Epochs. 100% of the train set used.

In all four cases, we observe that both errors take on large values at the beginning (with the largest values occurring in the first epoch when the model was initialized with random parameter values), which is logical since, in the initial stages of training, the model is at an early stage, and its performance is poor. As the number of epochs increases, we observe an exponential decrease in both errors in all four graphs. If we focus on the training loss, we notice that it continuously decreases as the number of epochs increases (as the model continues to train) with the corresponding blue curves asymptotically approaching the horizontal axis from a certain point onwards, in all four diagrams. In contrast, we observe that in all four cases, the validation loss initially decreases as the number of epochs increases but then either starts to increase (while the training loss

continues to decrease), indicating that the model is overfitted to the training images, or remains approximately stable, indicating negligible changes in model performance and thus no reason to continue training. To choose the appropriate number of epochs needed to complete the training without overfitting, we rely on the validation loss curve in each case and select the number of epochs corresponding to the minimum validation loss before it starts to increase or, in cases where it does not increase but remains practically stable from one epoch onwards, we choose the epoch up to which a significant error reduction is observed. The above graphs experimentally confirmed what was mentioned in the previous paragraph and the reason why we prefer the validation loss metric over the training loss.

Finally, we evaluate the four models (the models with the highest accuracy returned by the train_model() function in each case, based on the performance on the validation data) on the test set. The confusion matrices and the corresponding accuracy values are presented below.

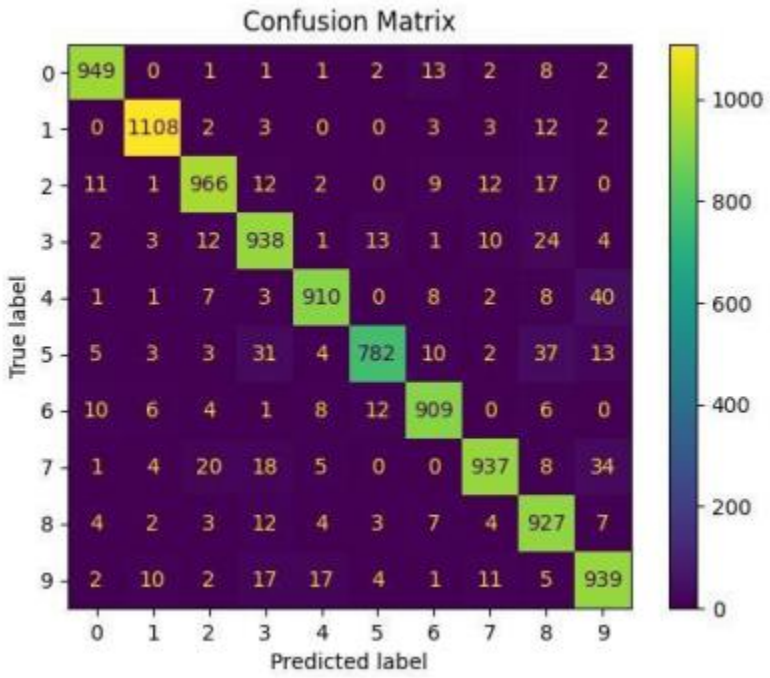

Figure 7 Confusion Matrix. 5% of the train set used

Accuracy on test set: 0.9380008012820513

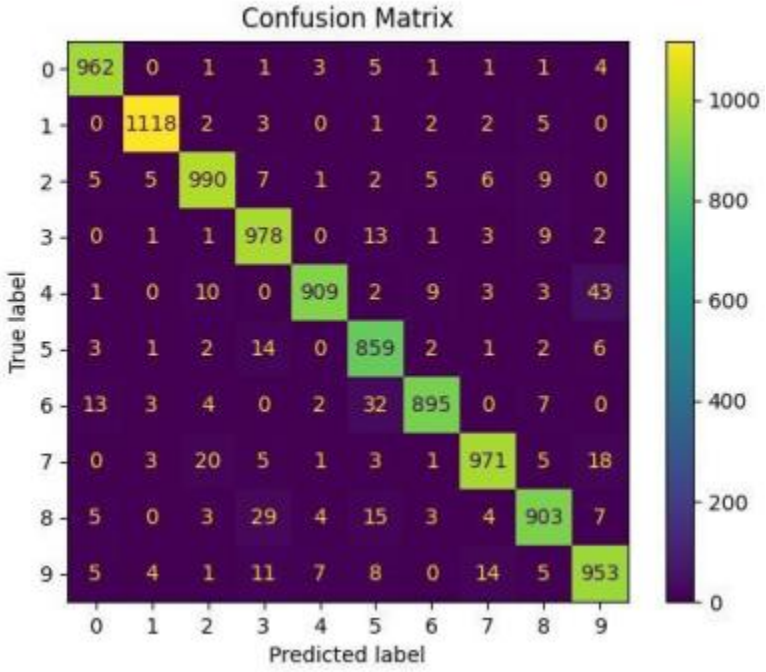

Figure 8 Confusion Matrix. 10% of the train set used

Accuracy on test set: 0.9553285256410257

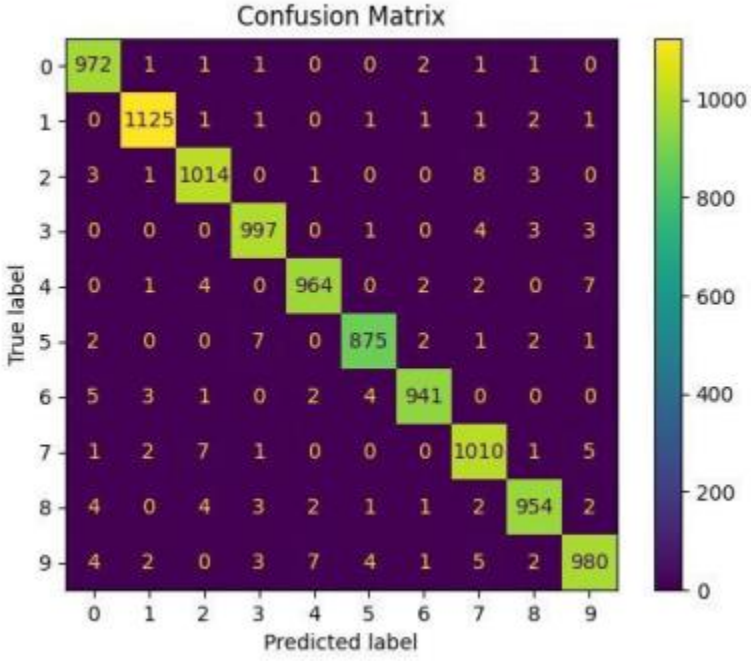

Figure 9 Confusion Matrix. 50% of the train set used

Accuracy on test set: 0.9847756410256411

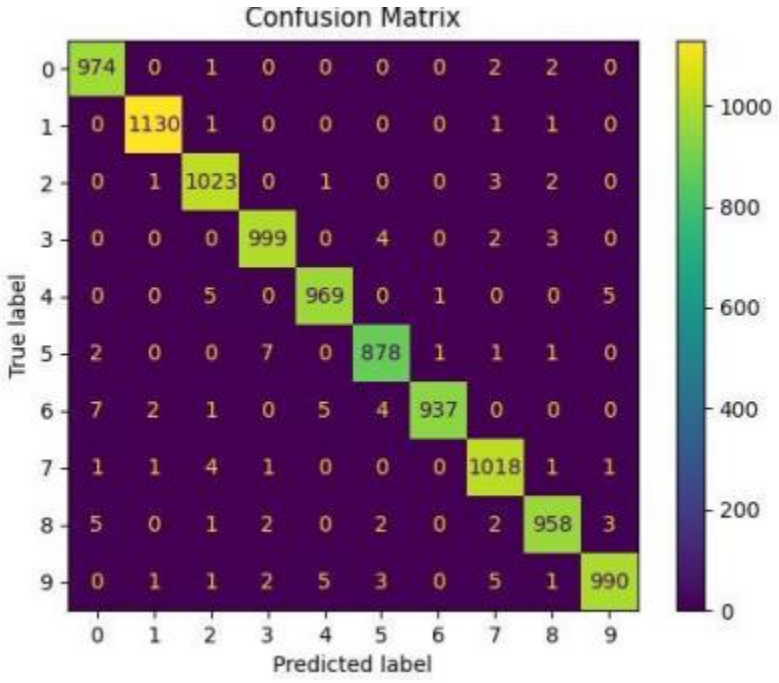

Figure 10 Confusion Matrix. 100% of the train set used

Accuracy on test set: 0.9891826923076923

Comparing the above accuracy values, we conclude that the increase in the available volume of training data has a direct impact on the model's performance. Specifically, the model trained on 5% of the original dataset achieved an accuracy of 93.80%, the one trained on 10% achieved an accuracy of 95.53%, the model trained on 50% achieved an accuracy of 98.48%, and finally, the model trained on the entire available training data achieved an accuracy of 98.92%. It is evident that increasing the volume of data leads to an improvement in the model's performance. This fact highlights one of the main characteristics of neural networks, the need for large volume of training data. The structures of neural networks are quite complex in most cases and are characterized by a vast number of parameters, making them an ideal choice for demanding applications (such as image classification) but simultaneously creating the need for the collection of vast amounts of data for their adequate training.

Part B:

In the second part of the project, the problem remains the same. The difference compared to before is that the first part of the network, i.e., the convolutional part, is removed, leaving only the second part, the classifier. Now, the network does not automatically learn the features of the training images; instead, these features are manually calculated using the **SIFT** algorithm and then fed into the classifier. Therefore, the first step is to compute a feature vector for each available image.

To find the features, the SIFT algorithm is used. SIFT identifies features that are scale, rotation, and illumination invariant. The input to the algorithm is an image, and the output is a set of 128D feature vectors — one vector for each feature point of the image identified by the algorithm. This process is characterized by a significant problem: in general, the SIFT algorithm will detect a different number of feature points for different images, resulting in each image being described by a different number of 128D vectors. Consequently, the input to the classifier will be of different dimensions each time. Obviously, this is not acceptable, as the input to the classifier must always be of specific, fixed dimensions. To achieve this, we apply the following process: we gather all the 128D vectors computed by SIFT for all the detected feature points of all 70,000 available images and create a dataset. We then apply the **KMeans** algorithm to this dataset, aiming to cluster the vectors, selecting the number of clusters to be 50. Each feature vector is assigned to a specific cluster out of the 50 available. Next, for each image, we initialize a histogram consisting of 50 bins, one for each cluster, we check which cluster each feature vector of the image belongs to, and increase the corresponding bar of the histogram by 1. This results in the histogram of each image being formed based on how its feature vectors are classified into the various clusters. The outcome of this process is that each image is described by a histogram — vector of the same dimensions (50D vector). Practically, this implementation is achieved by the custom functions `create_bow()` and `compute_histograms()`. The former takes as input the set of feature vectors and the desired number of clusters, implements the clustering, and returns the k-means model based on which the cluster of a vector is determined. The latter takes as input a list of sets of feature vectors (one set for each image) and the k-means model produced by `create_bow()`, returning a list of feature histograms.

The architecture of the model used in this part of is different from the one used previously; thus, we remove the convolutional and pooling layers from the model—class `CNN()`—resulting in the model `FCNN()` (see code for more details), which is essentially just the classifier. The only difference compared to the previous classifier is that the input to the first layer is of dimension 50 and not 12*18*18. The `train_model()` function is again used to train the model.

The errors during the training of the new model are as follows:

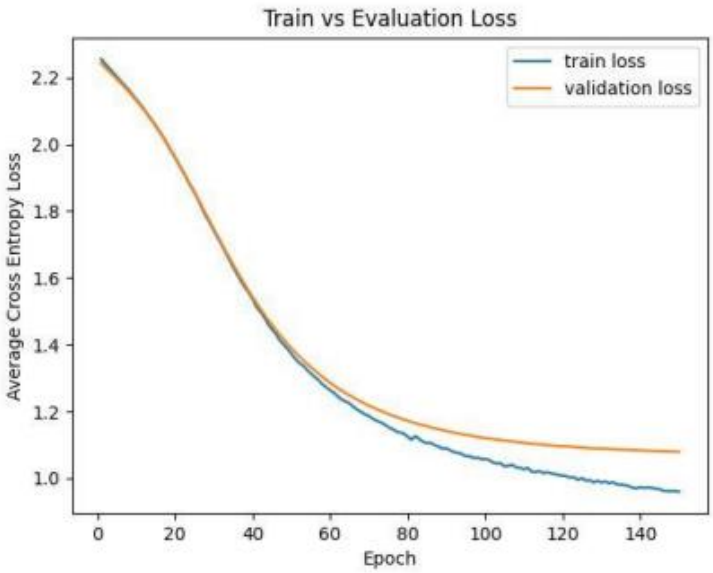

Figure 11Train loss - Validation loss vs Number of Epochs. 5% of the train set used.

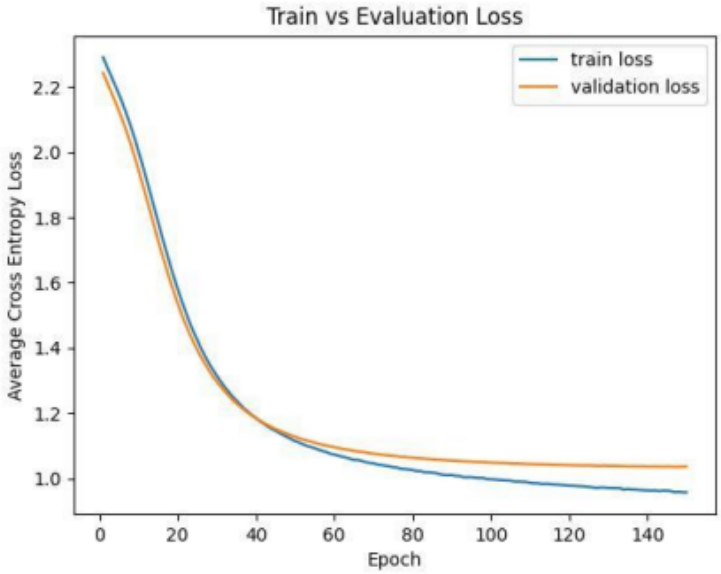

Figure 12Train loss - Validation loss vs Number of Epochs. 10% of the train set used.

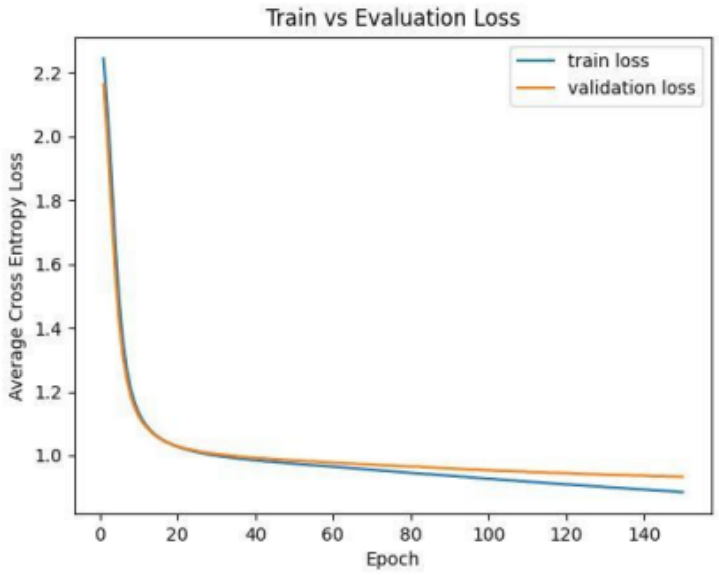

Figure 13 Train loss - Validation loss vs Number of Epochs. 50% of the train set used.

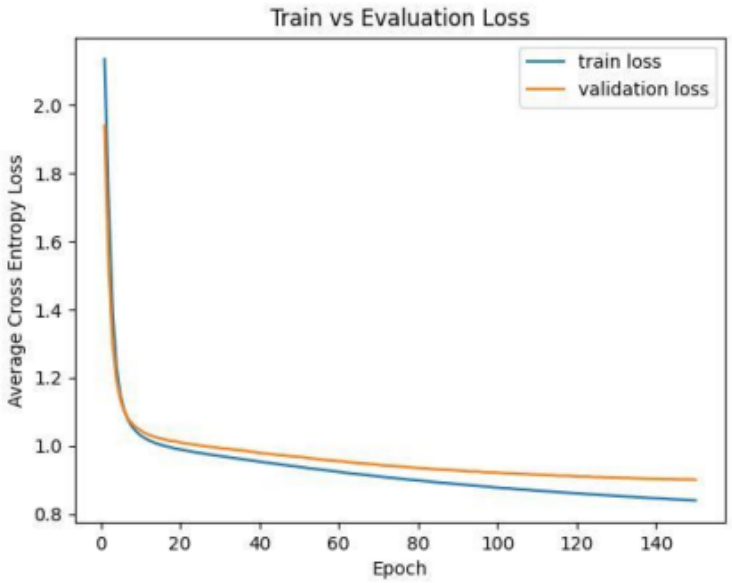

Figure 14 Train loss - Validation loss vs Number of Epochs. 100% of the train set used

In all four diagrams, we observe similar phenomena to those seen in the case of the convolutional network. Both the training error and the validation error reach their maximum values at the beginning of the training, when the network has not yet had time to "learn" the images. As the number of epochs increases, the training loss continuously decreases and eventually asymptotically approaches zero. Meanwhile, the validation loss also initially decreases, but after a certain point, the change becomes negligible and remains practically constant.

In the case of the convolutional network, some diagrams showed an increase in the validation loss, or in other words, overfitting of the model after a certain epoch. However, this is not observed here, even though the number of epochs is three times greater than before. This happens because overfitting does not depend solely on the number of epochs during training but also on the complexity of the model. A more complex model is more likely to be overfitted compared to a less complex one, even if trained for fewer epochs. This is exactly what we observe in our case. By removing the convolutional part of the network, we have reduced the complexity of the model, resulting in no overfitting even after 150 epochs of training. Obviously, we will not train the model for 150 epochs since, after a certain point, the validation loss is practically constant, and continuing the training would not make sense as the difference in accuracy would be negligible. For example, for the model resulting from using 10% of the training data, training is essentially complete after ~ 60 epochs.

The four models returned by the `train_model()` function were evaluated on the test set, and the results obtained are as follows:

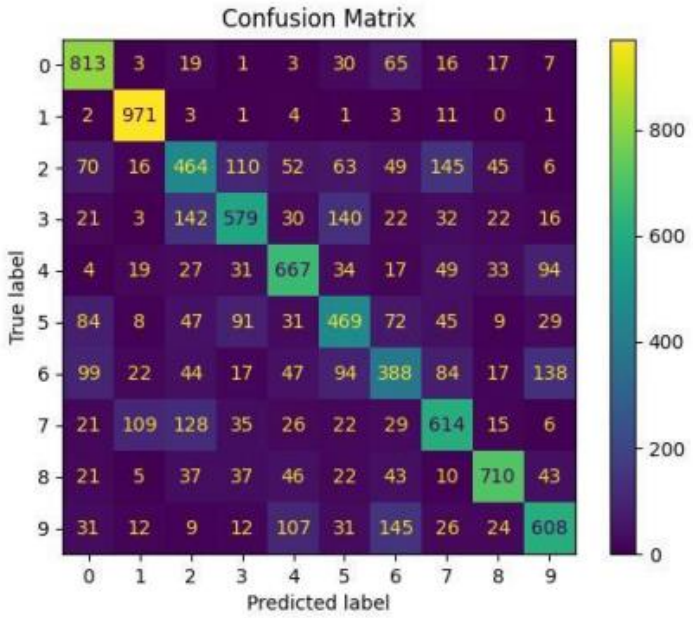

Figure 15 Confusion Matrix. 5% of the train set used

Accuracy on test set: 0.6416462418300654

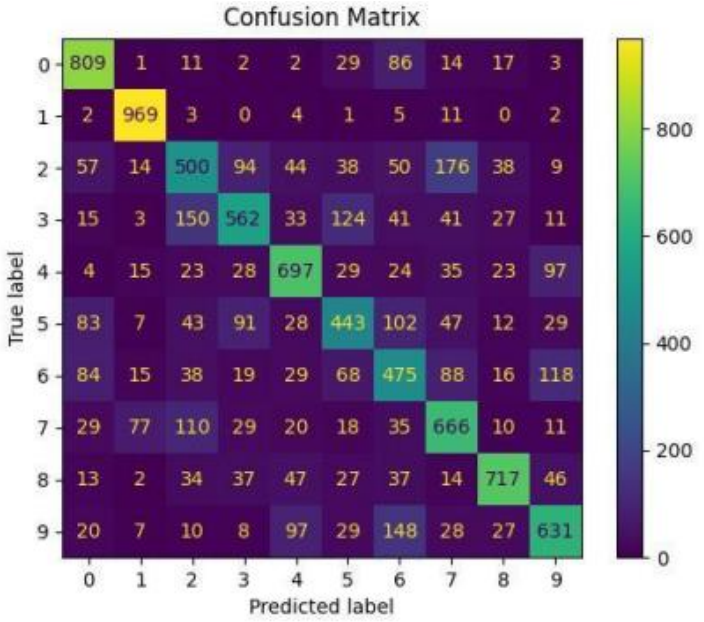

Figure 16 Confusion Matrix. 10% of the train set used

Accuracy on test set: 0.6606413398692811

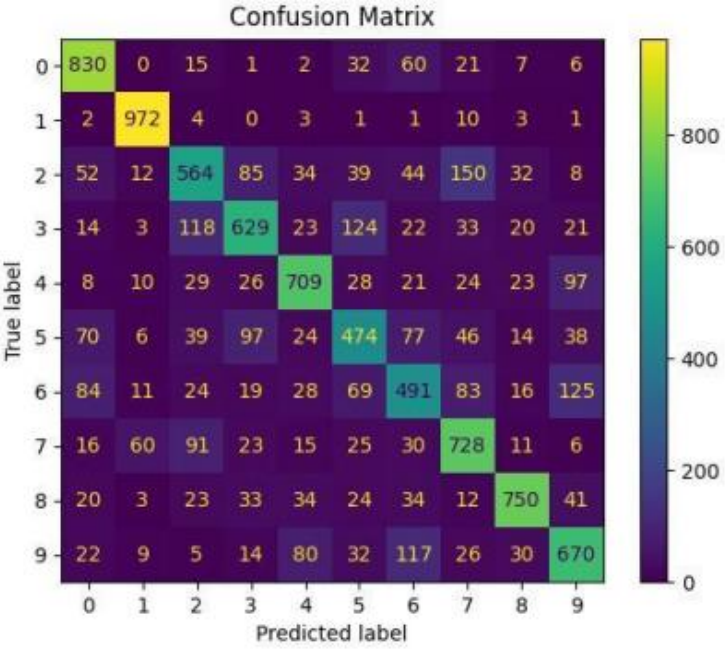

Figure 17 Confusion Matrix. 50% of the train set used

Accuracy on test set 0.5: 0.6961805555555556

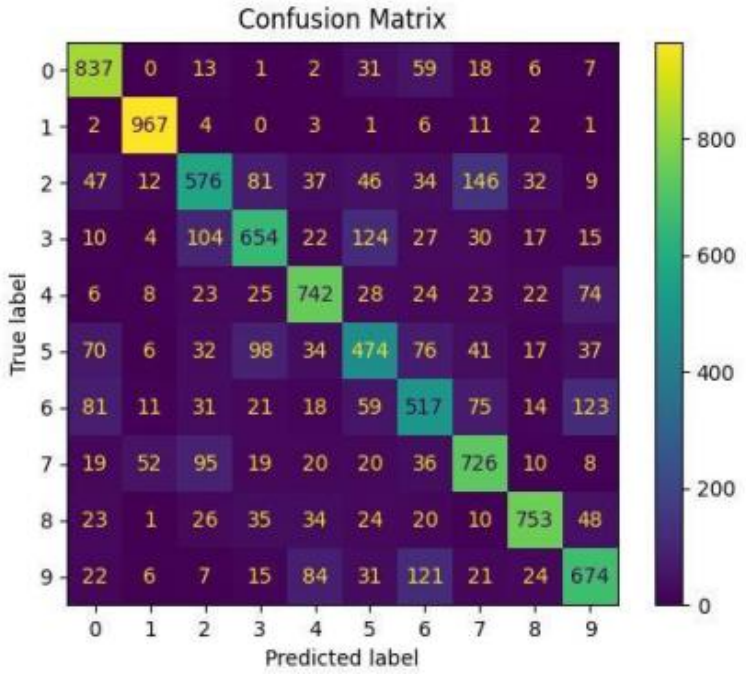

Figure 18 Confusion Matrix. 100% of the train set used

Accuracy on test set: 0.7066993464052288

Once again, increasing the percentage of training data used leads to an increase in the model's accuracy. The convolutional networks from the first part of the exercise exhibit significantly higher accuracy compared to the non-convolutional networks from the second part. For example, the convolutional model trained on 100% of the data achieves an accuracy of 98.92% on the test set, while the non-convolutional model, also trained on 100% of the data, has an accuracy of 70.67%, showing a difference of 28.25%!

This difference might be reduced by exploring different feature extraction methods, such as HoG, SURF, or even by parameterizing the process used in this exercise. For instance, we could examine different values for the number of clusters during k-means clustering or try an entirely different method for extracting feature vectors of the same length, based on the vectors produced by SIFT.

In conclusion, we have proven by comparing the two cases convolutional networks characterized by their automatic feature extraction capabilities, are preferred compared to simple fully connected networks and manual feature extraction, for applications related to image processing and analysis.