Report

Machine Learning

May 2024

IORDACHE Paul-Tiberiu 28706827
LUONG Ethan 28627904

Contents

Introduction

This project aims to create a deep learning library named `PyLDL` (Python Lightweight Deep Learning) and to implement neural networks. The implementation is inspired by previous versions of PyTorch and similar implementations that allow for highly modular generic networks. Each layer of the network is seen as a module, and a network is thus composed of a set of modules. In particular, activation functions are also considered as modules.

Through significant experiments, we explored various tasks such as image classification, time series prediction, and training deep neural networks. These experiments allowed us to analyze the impact of architectural choices, optimization techniques, hyperparameters, and hardware constraints on model performance. This project thus contributed to deepening our understanding in the domain of machine learning. The code and further details can be found on our GitHub repository.

Our codebase is organized into two main parts: the core library (`PyLDL`) and the experimental scripts (jupyter notebooks). The core library consists of several key modules that form the backbone of `PyLDL`. These modules include:

- `activations.py` - Implements various activation functions.
- `base.py` - Contains the base classes and utilities.
- `encapsulations.py` - Manages the encapsulation of layers and models.
- `exceptions.py` - Defines custom exceptions for error handling.
- `losses.py` - Provides different loss functions used in training.
- `modules.py` - Contains the main building blocks for neural networks.
- `optimizers.py` - Implements optimization algorithms.
- `utils.py` - Offers various utility functions to support other modules.

The experimental scripts are designed to test and demonstrate the capabilities of the library through various tasks and scenarios. The experiments include:

- `activation_network.ipynb` - Testing activation functions on a binary classification task.
- `clustering.ipynb` - Clustering in latent space for multiclass data by passing through autoencoders.
- `convolutional_network.ipynb` - Implements a convolutional neural network for classification.
- `denoiser.ipynb` - Implements denoising autoencoders to remove noise from images.
- `compression.ipynb` - Implements autoencoders in order to compress, than reproduce images.
- `linear_network.ipynb` - Regression using a basic neural network.
- `losses.ipynb` - Testing our loss functions.
- `multiclass_network.ipynb` - Multiclass classification using neural networks.
- `sequential_network.ipynb` - Experiments on our Sequential and Optim modules.

1 Linear Network

1.1 Linear Module

The Linear module implements a basic linear layer, whose forward pass is a dot product with its parameters (or weights) and the input. It is one of the most fundamental component of any neural network in `PyLDL` (Python Lightweight Deep Learning). It is usually followed by an activation module, and often encapsulated in `Sequential`, section 3.1. The initial weights $(w_{i,j})_{i,j}$ are uniformly chosen using the Xavier initialization to ensure that they do not increase exponentially and the parameters remain stable: $\forall i, j, w_{i,j} \in \left[\frac{-1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$ where n is the number of input features.

1.2 Mean Squared Error

The `MSELoss` simply implements the mean square error loss between the predicted and actual outputs. It is the mean of the squared euclidean norm of the difference of both outputs along the features axis. Instead of returning a vector of these norms, we chose to only return the mean as it is more convenient to plot the loss across the different epochs of the training phase.

The formula of the `MSELoss` is given by the equation 1 and its derivative is given by the equation 2

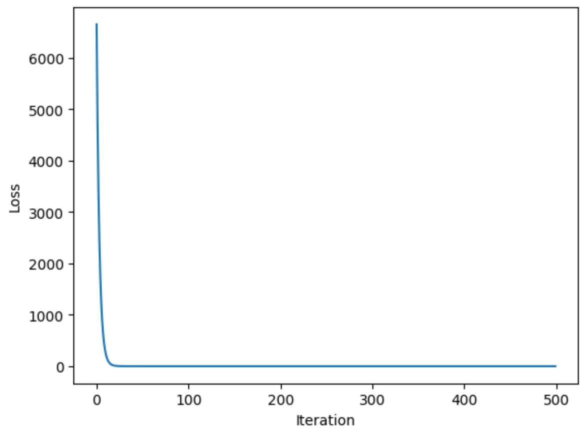
$$\text{Loss}(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} \|y_i - \hat{y}_i\|_2^2 \tag{1}$$

$$\frac{\partial \text{Loss}}{\partial \hat{y}} = -2(y - \hat{y}) \tag{2}$$

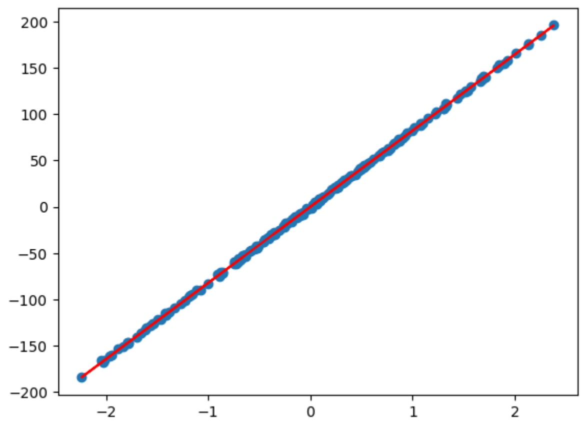
1.3 Experiment

In order to test our linear network, we conducted a gradient descent optimization. It iteratively adjusts the model parameters to minimize the difference between predicted and actual labels using the MSELoss (Subsection 1.2).

Using `scikit-learn`, we generate a dataset with a little bit of gaussian noise to test our `Linear` module. Then, we manually compute the forward and backward passes to do a gradient descent and plot the training loss. As we can see in Figure 1 (a), the training loss seems to converge quickly, so we expect good performances form the network which can be seen in the Figure 1 (b).



(a) Training Loss evolution (MSE)



(b) Result of the linear regression

Figure 1: Linear Regression problem (`n_iter=500`, `gradient_step=1e-4`, `n_samples=1000`)

2 Nonlinear Network

2.1 Activation Functions

Nonlinear modules are basically activation functions which inherit the characteristics of the module `Activation`, available in `pyldl/activations.py`. As such, they only have a forward and a backward pass. They don't have any parameter to update or to reset. Tests can be found in experiments/activation_network.ipynb.

First, we considered the hyperbolic tangent, which is easily implemented thanks to `numpy` built-in function `tanh`:

$$\tanh(X) = \frac{e^X - e^{-X}}{e^X + e^{-X}} \tag{3}$$

$$\tanh'(x) = 1 - \tanh^2(x) \tag{4}$$

We also implemented the sigmoid function:

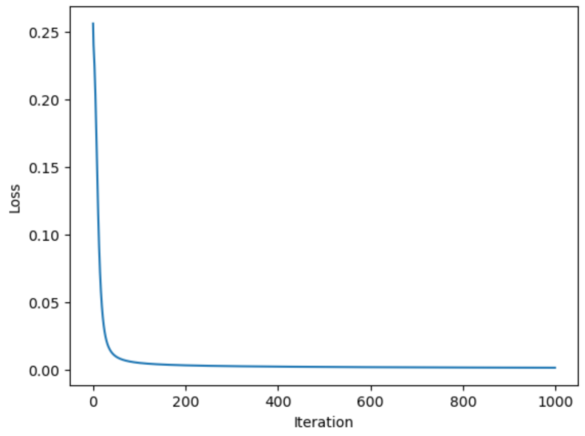
$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{5}$$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{6}$$

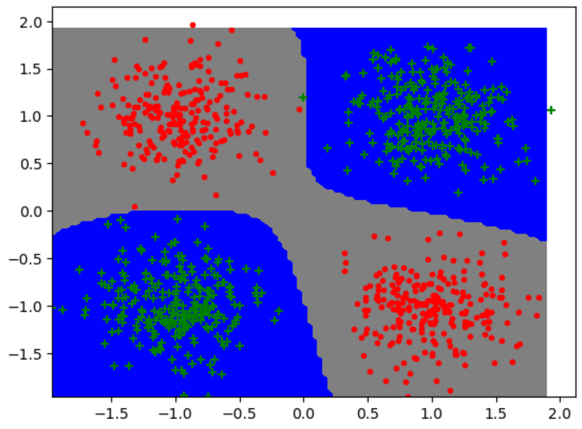
2.2 Experiment

In order to test our nonlinear network, we conducted a gradient descent optimization in order to be able to make a binary classification. The error is equally calculated using the MSELoss. We will setup a two layers linear network with one hyperbolic tangent module between these layers, and one sigmoid module after the last hidden layer of the network. Then, we use a custom dataset that produces 2D points based on gaussian distributions.

Our gradient descent is here entirely handcrafted: we simulate a forward pass by giving each layer output as an input to the following layer. Then, we compute the loss for later and starts the backpropagation and update each module's parameters in reverse order. The training loss seen in the Figure 2 (a) seems great: it reaches values close to zero very quickly (less than 100 iterations), so we expect good performances from the network. The decision boundary seen in the Figure 2 (b) is obviously nonlinear and suits the classification problem. With test samples, we reach an accuracy well over 99%. To conclude, for a simple nonlinear classification problem, our neural network thus yields great results.



(a) Training Loss evolution (MSE)



(b) Result of the linear regression

Figure 2: Nonlinear classification problem (`n_iter_max=1000`, `gradient_step=1e-3`, `n_samples=500`)

3 Encapsulation and Optimization

Encapsulation is a fundamental concept in the design of the `PyLDL` library, ensuring that each component of the neural network is modular and self-contained. In the context of `PyLDL`, encapsulation refers to the way different parts of the network, such as layers and optimization processes.

3.1 Sequential Network

The `Sequential` class, located in `pyldl/layers.py`, serves as a core component of the `PyLDL` library, representing a neural network. It encapsulates a list of Module objects, each of which defines a layer of the network. The forward pass of a `Sequential` network involves sequentially executing the forward pass of each layer, where the input to each layer is the output from the previous layer. The backward pass, which is essential for training the network, is conducted in a similar sequence but in reverse order. The `Sequential` class ensures that backpropagation is properly managed through the `backward()` and `update_parameters()` methods, facilitating efficient gradient descent optimization.

Experiments demonstrating the effectiveness of this library can be found in the notebook `experiments/sequential_network.ipynb`. Notably, the tasks and random seed settings in this notebook are consistent with those in two previous notebooks, ensuring that the results are reproducible and should match exactly across these experiments. It was done by setting the random seed in said notebooks and comparing the different results and visualizations.

3.2 Optimization

The optimization process in `PyLDL` is also encapsulated within specific classes and functions. The `Optim` class in `pyldl/optimizers.py` encapsulates the entire training step of a network, including the forward pass, backpropagation, and parameter updates. By bundling these steps into a single class, `PyLDL` ensures that the optimization process is handled consistently and efficiently across different networks.

In the early-stage development of `PyLDL`, the forward pass and the backpropagation were manually computed by giving the output of a layer to the following layers, and updating the parameters in reverse order. The SGD function, also in `pyldl/optimizers.py`, encapsulates the stochastic gradient descent training process. This function iteratively trains the network over a defined number of epochs, utilizing minibatch gradient descent. Depending on the batch size, the descent can be stochastic or full-batch, providing flexibility in the training process. The encapsulated design of SGD allows users to easily modify training parameters and apply the same optimization routine to different networks.

4 Multiclassification

4.1 Softmax

For multi-class classification problems, the network needs to decide which class is the most likely given an example. To compare the outputs for each class, we need to ensure that they follow a probability distribution. We thus implemented a `Softmax` module. The forward pass is given by:

$$\text{Softmax}_i(x) = \frac{e^{x_i}}{\sum_k e^{x_k}} \tag{7}$$

$$\text{Softmax}_i'(x) = \text{Softmax}_i(x) \cdot (1 - \text{Softmax}_i(x)) \tag{8}$$

In practical use, we don't use the Softmax on the output of the network. Instead, the Cross Entropy loss (Subsection 4.2) which is usually used in multiclassification problems instead of the MSE loss (Subsection 1.2) already transforms the output.

4.2 Cross Entropy Loss

The `CrossEntropyLoss` implements the cross-entropy loss between the predicted and actual outputs. It first applies the softmax function to the predicted outputs to obtain the probability distribution over the classes, ensuring numerical stability by subtracting the maximum predicted value. The loss is then calculated as the mean of the negative log likelihood of the true class probabilities. By averaging over all samples, the returned value provides a single scalar loss, which is more convenient to plot across different epochs during the training phase.

The gradient computation involves the difference between the predicted probabilities and the actual one-hot encoded labels, facilitating the optimization process in training neural networks. This loss usually performs better than the MSE which is more likely to smooth the output than extracting one definite class.

Equation 9 defines the cross-entropy loss, and the equation 10 gives its derivative.

$$\text{Loss}(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} \left(-\sum_{c=1}^{C} y_{i,c} \hat{y}_{i,c} + \log \sum_{c=1}^{C} \exp(\hat{y}_{i,c}) \right) \tag{9}$$

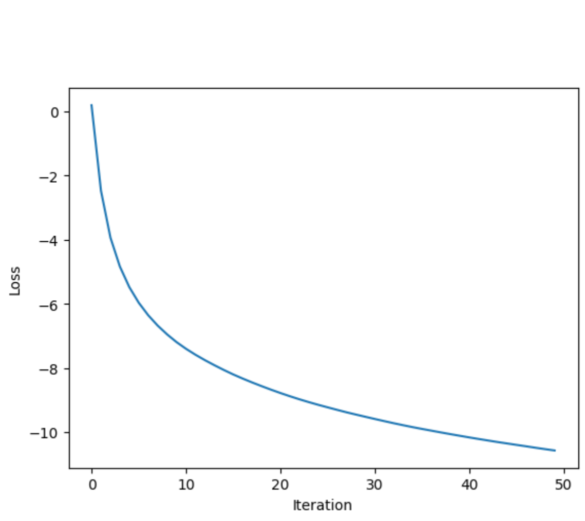
$$\frac{\partial \text{Loss}}{\partial \hat{y}} = \frac{\exp(\hat{y}_{i,c} - \max(\hat{y}_i))}{\sum_{c=1}^{C} \exp(\hat{y}_{i,c} - \max(\hat{y}_i))} - y_{i,c} \tag{10}$$

4.3 Experiment

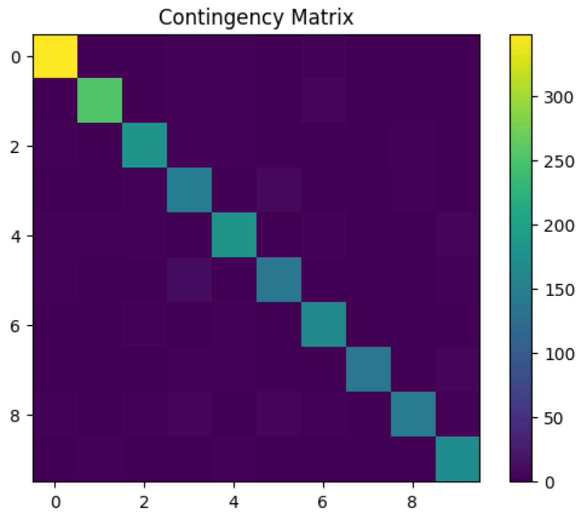
To effectively test our multiclassification-related modules, we consider a digit multiclassification problem from `scikit-learn`: given black and white digit images, the network should predict what digit - between 0 and 9 - is represented without using one-versus-all. In order to get proper outputs of size `n_classes = 10`, we need to transform the labels using one-hot encoding. For example, we get the following pairs (label, one-hot encoded vector):

$$0 \rightarrow \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$
$$4 \rightarrow \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$
$$7 \rightarrow \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

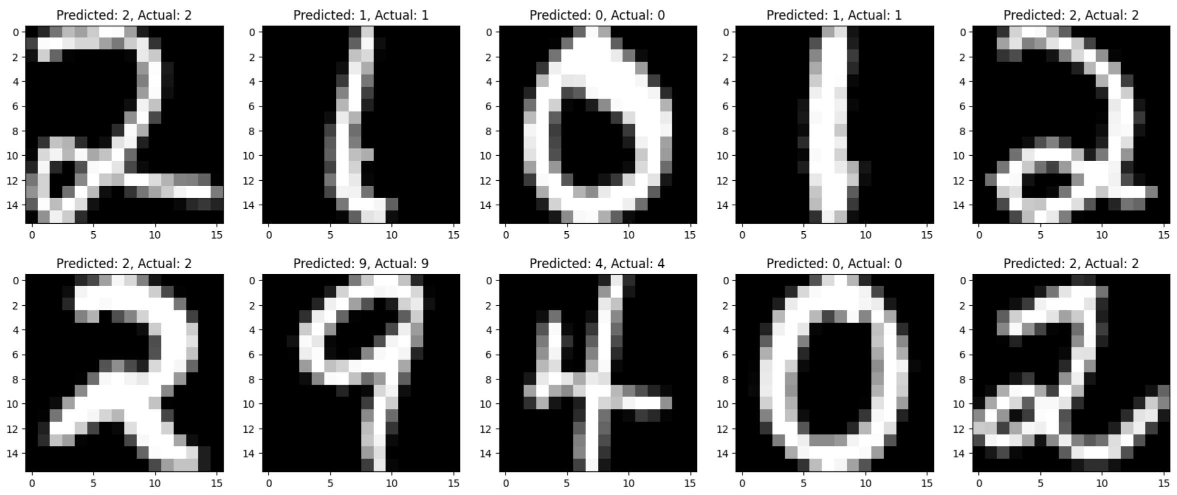
While it is not as fast as formerly encountered problems (regression, binary classification), the loss is converging well. Plus, the network performs very well and reaches a high accuracy on the test dataset (over 97%). The contingency matrix also shows us the digits that are commonly mistaken for each other (e.g. nines may be mistaken for fives). It could be because in the examples, these digits look very similar.



(a) Training Loss evolution (Cross Entropy)



(b) Contingency Matrix



(c) Some examples of classified digits

Figure 3: Digits Multiclassification problem (`n_iter=1000`, `gradient_step=1e-3`, `batch_size=64`)

5 Test Campaigns

5.1 Autoencoder

An autoencoder is a specialized type of neural network designed for unsupervised learning tasks, particularly for tasks that involve data compression and reconstruction. The `AutoEncoder` class in `PyLDL`, also located in `pyldl/layers.py`, encapsulates this concept. This architecture will be used in the following test campaigns.

An autoencoder network is divided into two symmetric parts: the encoder and the decoder. The encoder transforms the input data into a compact representation in a latent space with a lower dimensionality than the original input space. For instance, an encoder might consist of layers [Linear(64,16), Linear(16,2)], reducing the input dimension from 64 to 2 through successive transformations.

Conversely, the decoder reconstructs the original data from this compact representation. It mirrors the structure of the encoder but in reverse order, with layers such as [Linear(2,16), Linear(16,64)]. This symmetry ensures that the decoder effectively learns to reconstruct the input data from the encoded representation, striving for minimal loss of information. The `AutoEncoder` class in `PyLDL` thus provides a robust framework for implementing and training autoencoder networks.

By providing detailed encapsulation of network creation, training, and optimization processes, as well as specialized structures like autoencoders, the `PyLDL` library offers a comprehensive toolkit for developing and experimenting with neural networks in a structured and efficient manner.

5.2 Binary Cross Entropy

The `BCELoss` implements the binary cross-entropy loss between the predicted and actual outputs. The loss is computed using the 11 formula. This loss is decent when handling image reconstruction tasks. Compared to the MSE loss (Subsection 1.2), whose outputs are smoothed, the outputs of the BCE loss are more extreme and either close to 0 or 1. In the case of white and black digit images, it makes the white digits sharper, contrasting with the black background.

$$\text{Loss}(y, \hat{y}) = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i + \epsilon) + (1 - y_i)\log(1 - \hat{y}_i + \epsilon)\right] \tag{11}$$

where ϵ is a small value to avoid logarithm of zero. The derivative of the binary cross-entropy is given by:

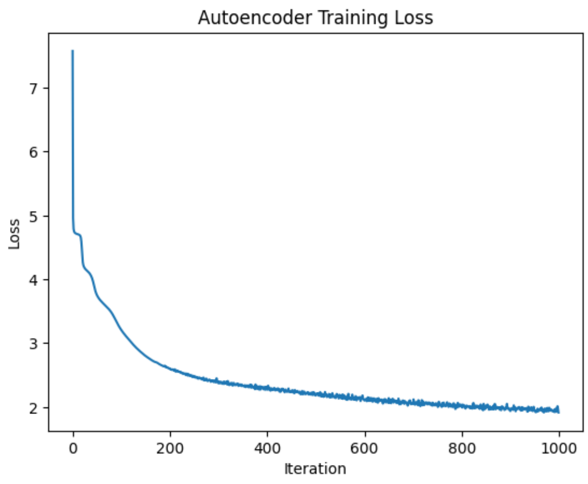
$$\frac{\partial \text{Loss}}{\partial \hat{y}} = \frac{\hat{y} - y}{(\hat{y} + \epsilon)(1 - \hat{y} + \epsilon)N} \tag{12}$$

which accounts for the difference between the predicted probabilities and the actual labels, scaled by the predicted probabilities and the number of samples.

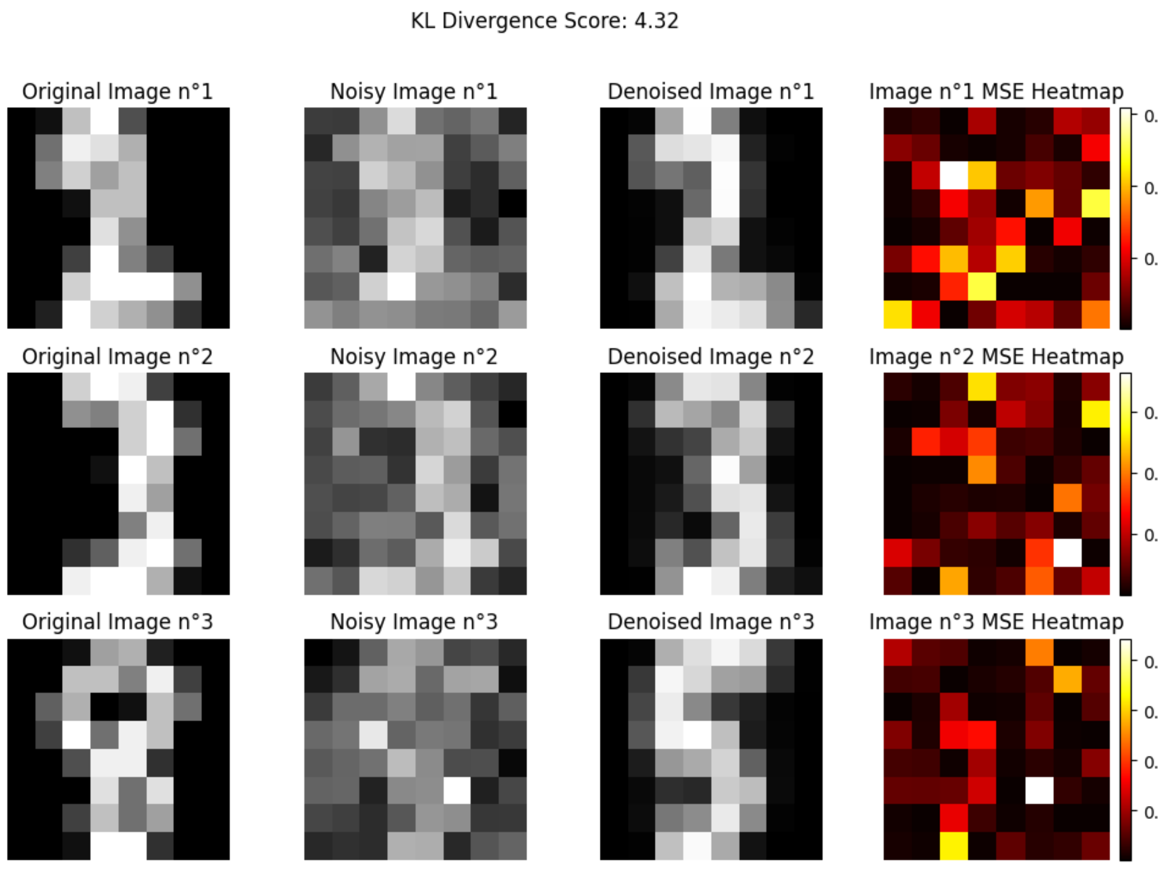
5.3 Denoiser

This part outlines a comprehensive experiment designed to evaluate the effectiveness of an autoencoder-based denoiser on the different datasets. The process begins with loading and normalizing the dataset, followed by adding noise to the data. The dataset is then split into training and testing sets. An autoencoder is constructed with a specific encoder-decoder architecture to learn a compressed representation of the noisy data and subsequently reconstruct the original data. The autoencoder is trained using stochastic gradient descent (SGD, Subsection 3.2) with mean squared error (MSE, Subsection 1.2) loss. The performance is visualized through loss plots and evaluated using the Kullback-Leibler (KL) divergence score between the original and denoised images. The effectiveness of the autoencoder in improving classification accuracy is tested by comparing the performance of a neural network classifier on original, noisy, and denoised images. The results are summarized with accuracy scores and visualized using contingency matrices.

For our first test, we chose the digits dataset and an autoencoder is defined and trained using our SGD. The encoder is: `Sequential(Linear(X_train.shape[1], 32), Tanh(), Linear(32, 16), Tanh(), Linear(16, 2), Tanh())` and the loss used is the MSE loss. After training, the good results show us that we can expect great performances for image reconstruction tasks of the networks we will develop. Indeed, the training loss seems to converge at last even though it is slightly unstable and well over 0. Plus, even though the reconstructed images are not as close to the original images due to the MSE loss and our network, they are still quite accurate. The errors shown on the heatmap seem to be very localized.



(a) Training Loss evolution (MSE)



(b) Reconstructed Images and Heatmaps

Figure 4: Image Reconstruction with an AutoEncoder (`n_iter=1000`, `gradient_step=1e-3`, `batch_size=64`)

We then studied the accuracy of each type of image inputs. The original images have a 97% accuracy. On the other hand, the accuracy using noisy images is 78% and the one using denoised images is 82%. While the accuracy using the original images is similar to the one we got with our standard multiclassification network, the accuracies with the noisy and denoised images are much lower. We can still notice that using the denoised images yields slightly better results.

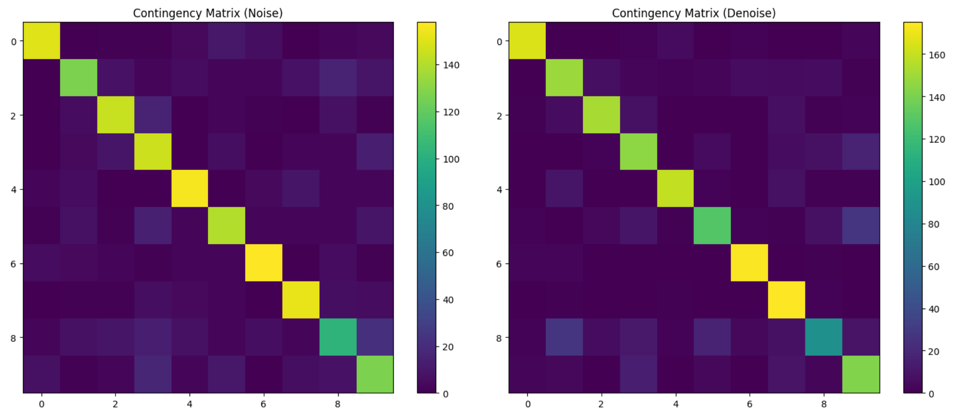


Figure 5: Denoiser Contingency Matrix

The contingency matrixes are very similar with the noisy and denoised images. However, with the noisy images, it seems that for a given digit, the range of digits mistaken by the network is wider than with the denoised images. In order to find a better model configuration, we perform a grid search over various hyperparameters. It results in the following parameters:
`{'activation_functions': 'Sigmoid', 'batch_size': 32, 'gradient_step': 0.01, 'loss': 'BCELoss', 'n_epochs': 500, 'n_neurons_per_layer': [64, 32]}`

The training loss has an odd shape at the beginning but it still seems to converge towards close-to-0 values. However, compared to the model we had initially, the reconstructed images are much better. The autoencoder seems to do a great work as it is difficult to recognize the noisy images even for us. Plus, the low Kullback-Leibler Divergence score indicates that the reconstructed images are close to the original ones.

We then measured the accuracy for each type of image, and our performances seem to grow. The original images have a 96% accuracy. On the other hand, the accuracy using noisy images is 80% and the one using denoised images grows to 85% from our previous model. The accuracies with the noisy and denoised images are again, lower than the original ones. However, we sligthly increased our accuracy using denoised images and we can see in the Figure 7 (b), our denoised images are very close to the original ones. We can also see, in Figure 7 (b), that the MSE Heatmaps seem to be more stable than our previous model, however we can still notice errors with a higher value.

Regarding the contingency matrixes (Figure 6), we can still say that they are quite similar with the noisy and denoised images. Also, we can still admit that with the noisy images, for a given digit, the range of digits mistaken by the network is wider than with the denoised images. So a similar behaviour to our previous model is observed.

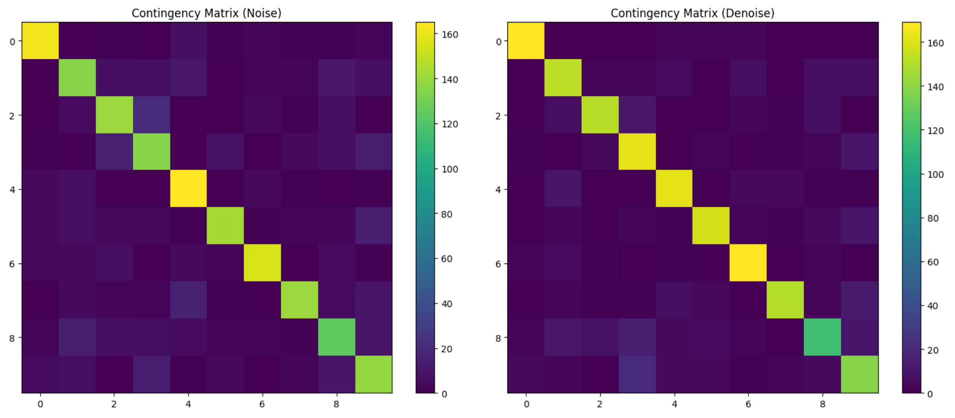
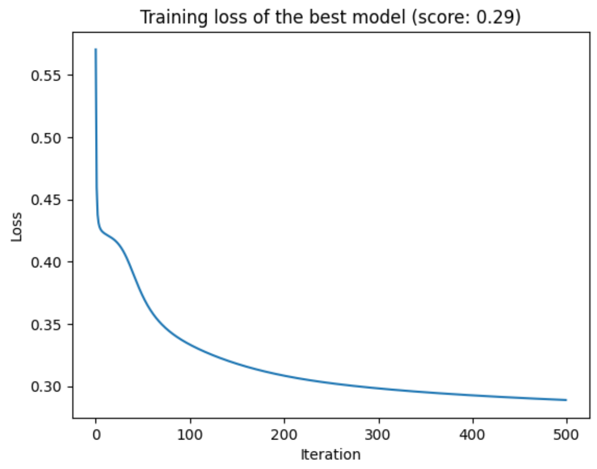
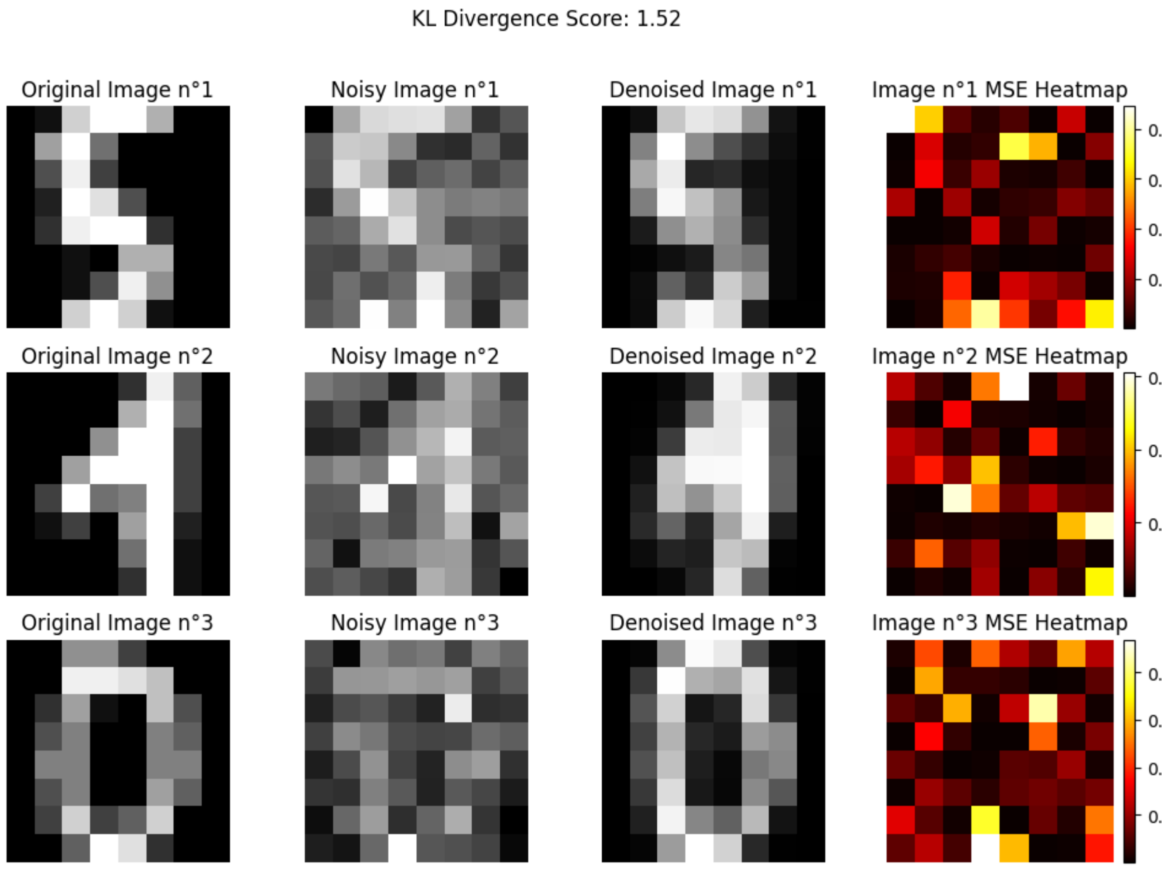


Figure 6: Denoiser Contingency Matrix (Digits)



(a) Training Loss evolution (MSE)

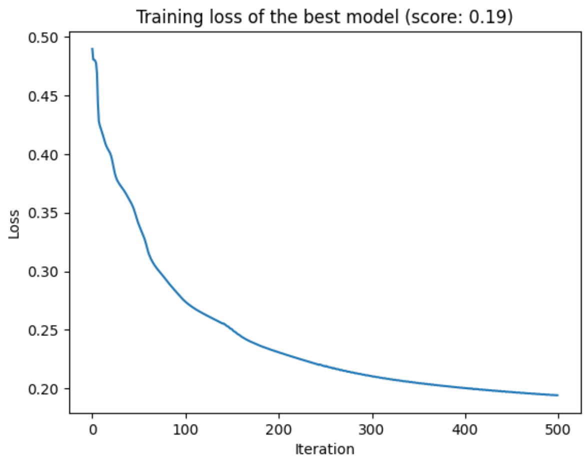


(b) Reconstructed Images and Heatmaps

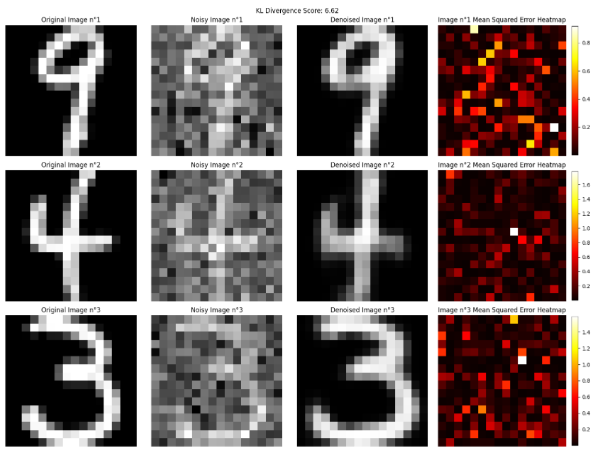
Figure 7: Digits Image Reconstruction with an AutoEncoder (`n_iter=500`, `gradient_step=1e-2`, `batch_size=32`)

We then proceeded to study another digits dataset: USPS, which has bigger images. The same pre-processing is applied. This time, we directly started by the grid search, and the best hyperparameters we found are: `{'activation_functions': 'Sigmoid', 'batch_size': 32, 'gradient_step': 0.01, 'loss': 'BCELoss', 'n_epochs': 500, 'n_neurons_per_layer': [256, 128, 32]}`

We still get a model that has a training loss that converged, as seen in the Figure 8. We can also say that the denoised images look very close to the original ones, however small differences can be seen. Looking than at thea heatmaps, we can admit that the small differences are normal, because we still have higher valued errors. Regarding the accuracies of our model and hyperparameters, we got 96.13% for the original images, 91% for the noisy images, and 96.06% using our denoised images, which is a great result, being very close to the original images and the gap between the accuracy of the denoised images and the noised images increased significantly comparing to our previous models.



(a) Training Loss evolution (MSE)



(b) Reconstructed Images and Heatmaps

Figure 8: USPS Image Reconstruction with an AutoEncoder (`n_iter=500`, `gradient_step=1e-2`, `batch_size=32`)

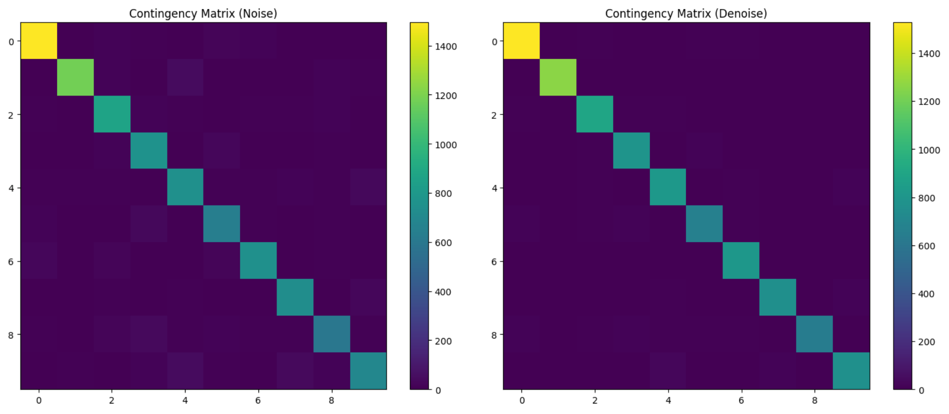
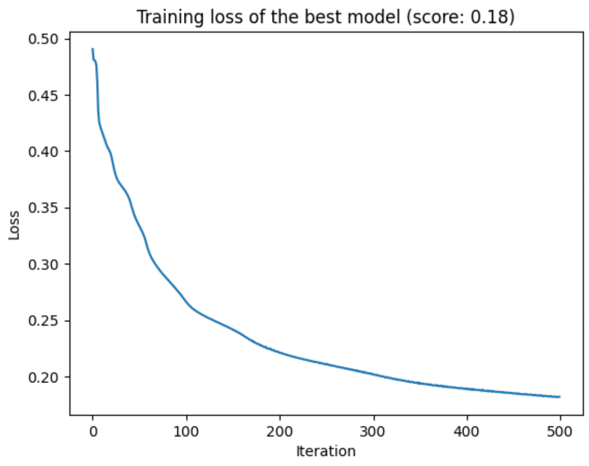


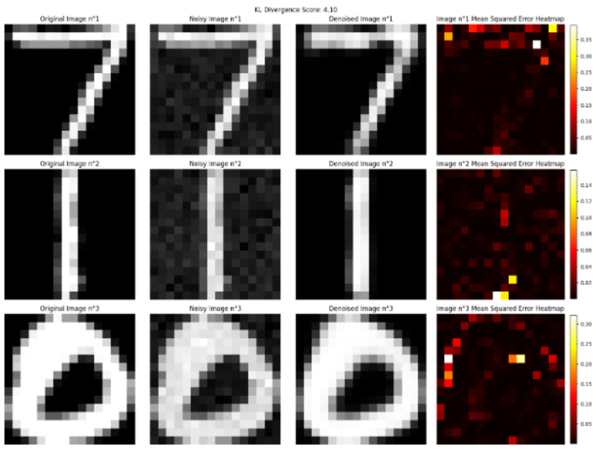
Figure 9: Denoiser Contingency Matrix (USPS)

Overall, the contingency matrixes have much less inaccurate data compared to the previous dataset. It is also more visible that the denoised data yields better results, this being equally seen in the accuracies given during our tests. However, we did previous tests with smaller noise values and the results are quite interesting. In this part, we will refer to the USPS dataset only, but similar results were found for the smaller digits dataset.

After performing a grid search, we can see that the loss converged (Figure 10). The best hyperparameters found are:
`{'activation_functions': 'Sigmoid', 'batch_size': 32, 'gradient_step': 0.01, 'loss': 'BCELoss', 'n_epochs': 500, 'n_neurons_per_layer': [256, 128, 32]}`.



(a) Training Loss evolution (MSE)



(b) Reconstructed Images and Heatmaps

Figure 10: Digits Image Reconstruction (Smaller Noise) with an AutoEncoder (`n_iter=500`, `gradient_step=1e-2`, `batch_size=32`)

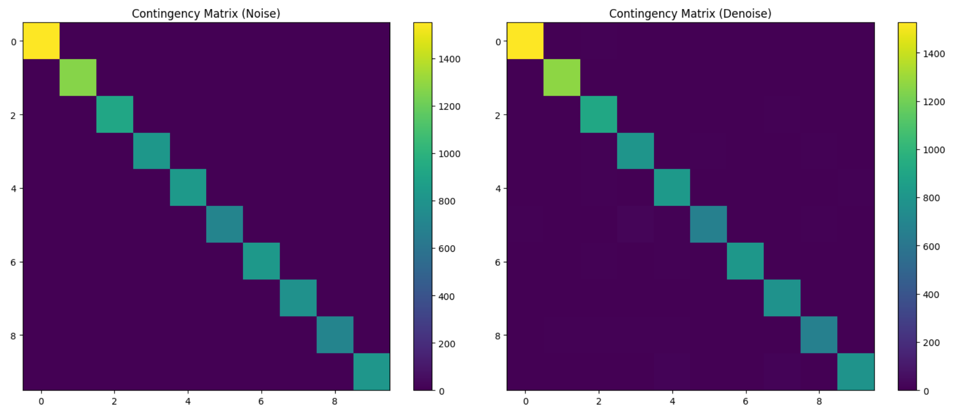


Figure 11: Denoiser Contingency Matrix (Digits, Smaller Noise)

As we can see in Figures 9 and 10, both the noisy and denoised images close to the original which is expected due to the small value of noise we chose. Also, the MSELoss Heatmaps showcase some higher value errors on plenty of the significant pixels, but the results are quite good. On the other hand, the accuracies seem quite high, but also strange. With the original images we have a 96.6% accuracy, with the noised images 99%, and with the denoised 97%. It seems rather strange to perform better with the noised images than with the original and the denoised ones. Regarding the contingency matrixes, both seem to be stable and very similar, this being equally reflected in the accuracies.

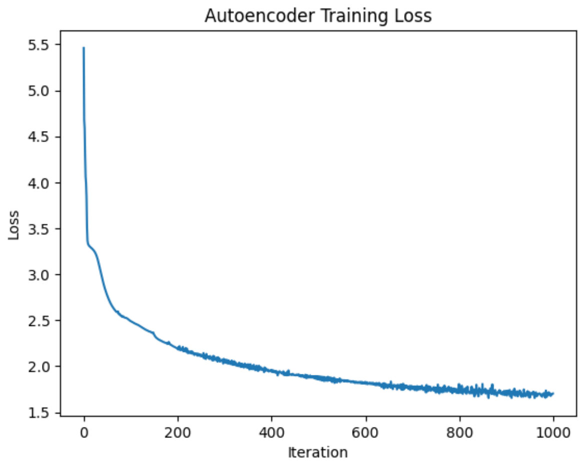
We wanted to include these results with smaller noise because we think they are rather interesting. Our hypothesis is that, using a smaller noise and a split of 0.8 and 0.2 for the train and the test images, it is normal for the noised images to perform rather well as the performances of the denoiser don't outweigh the recognition abilities of the network.

5.4 Clustering

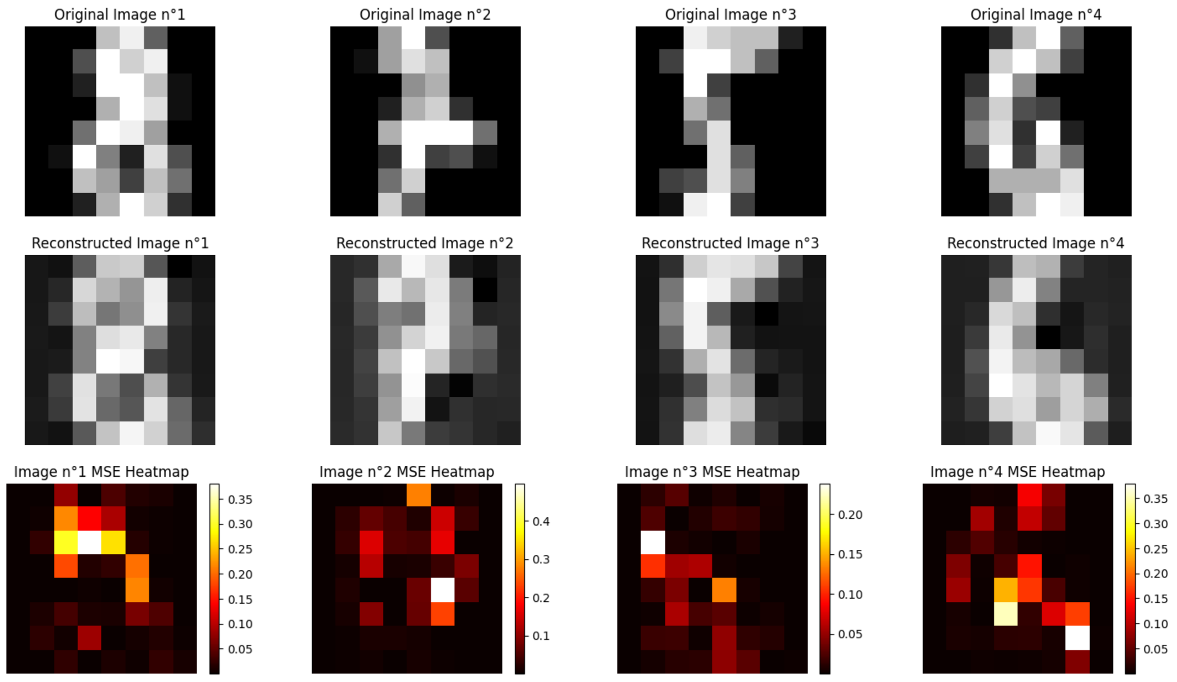
This section demonstrates experiments involving the use of an autoencoder for clustering in the latent space on multiclass datasets. It starts by loading and preprocessing the `scikit-learn` dataset, normalizing the data using Min-Max scaling, and splitting it into training and testing sets. An autoencoder is then defined and trained with a specified architecture and parameters using Stochastic

Gradient Descent (SGD). The encoder is: `Sequential(Linear(X_train.shape[1], 32), Tanh(), Linear(32, 16), Tanh(), Linear(16, 2), Tanh())` and the loss used is the MSE loss.

After training, the good results it yields shows us that we can expect great performances for image reconstruction tasks of the networks we will develop. Indeed, the training loss seems to converge at last even though it is slightly unstable and well over 0. Plus, even though the reconstructed images are a tad blurry due to the MSE loss, they are still quite accurate. The errors shown on the heatmap seem to be very localized.



(a) Training Loss evolution (MSE)



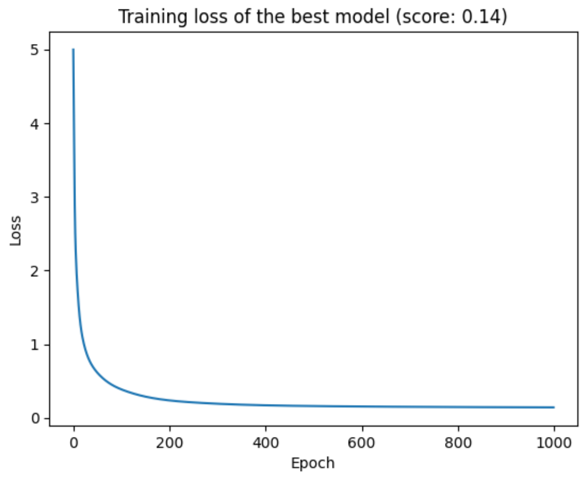
(b) Reconstructed Images and Heatmaps

Figure 12: Image Reconstruction with an AutoEncoder (`n_iter=1000`, `gradient_step=1e-3`, `batch_size=64`)

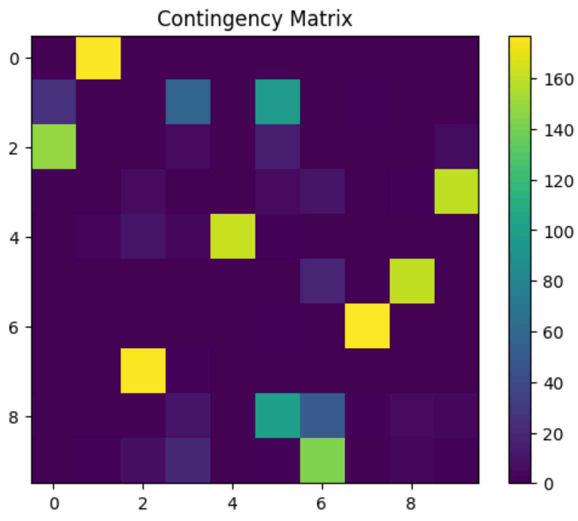
In order to find a better model configuration, we perform a grid search over various hyperparameters. It results in the following parameters:
`{'activation_functions': 'Sigmoid', 'batch_size': 32, 'gradient_step': 0.01, 'loss': 'MSELoss', 'n_epochs': 1000, 'n_neurons_per_layer': [64, 32]}`

The training loss has a great shape, and has converged. The latent representations obtained from the autoencoder are clustered using KMeans, and the clustering performance is evaluated using the cluster purity. Finally, the clustering results are visible through contingency matrices, providing an understanding of how an autoencoder can be used to transform data into a latent space suitable for clustering and how effective different configurations are in achieving meaningful clusters.

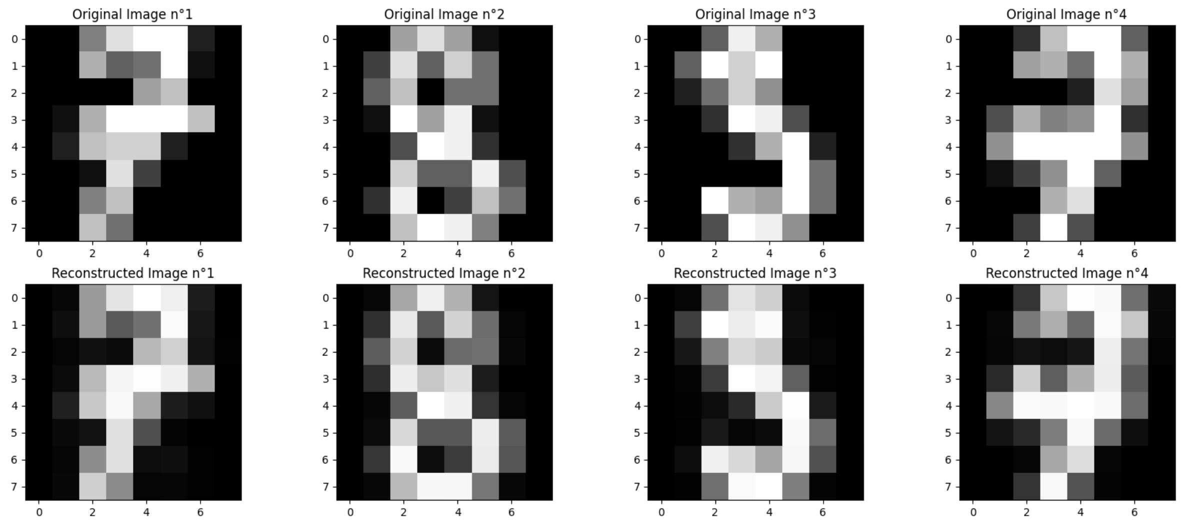
Most of the classes have high purity clusters (over 80%, meaning that most of the clustered digits are the same. The network is thus able to discriminate most classes well. However a few of the classes (mostly for 1 and 8) have a rather low purity (between 40% and 50%). It could be because they have several similar-looking digits (e.g. 8 is similar to 0 and 6).



(a) Training Loss evolution (MSE)



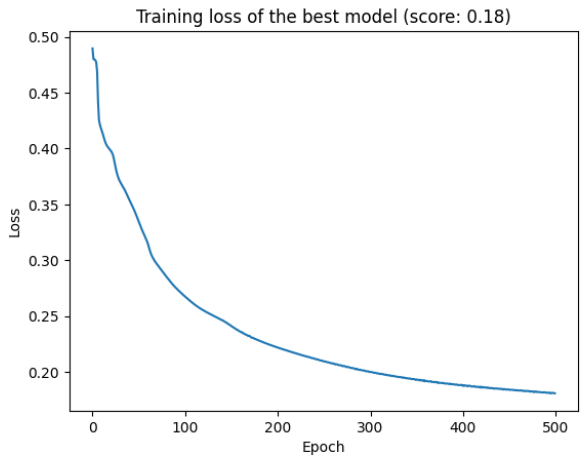
(b) Clustering Contingency Matrix



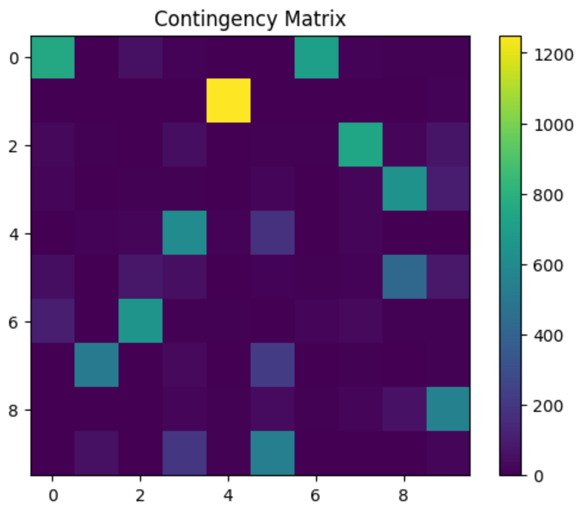
(c) Reconstructed Images and Heatmaps

Figure 13: Image Reconstruction and Clustering (Digits Dataset)

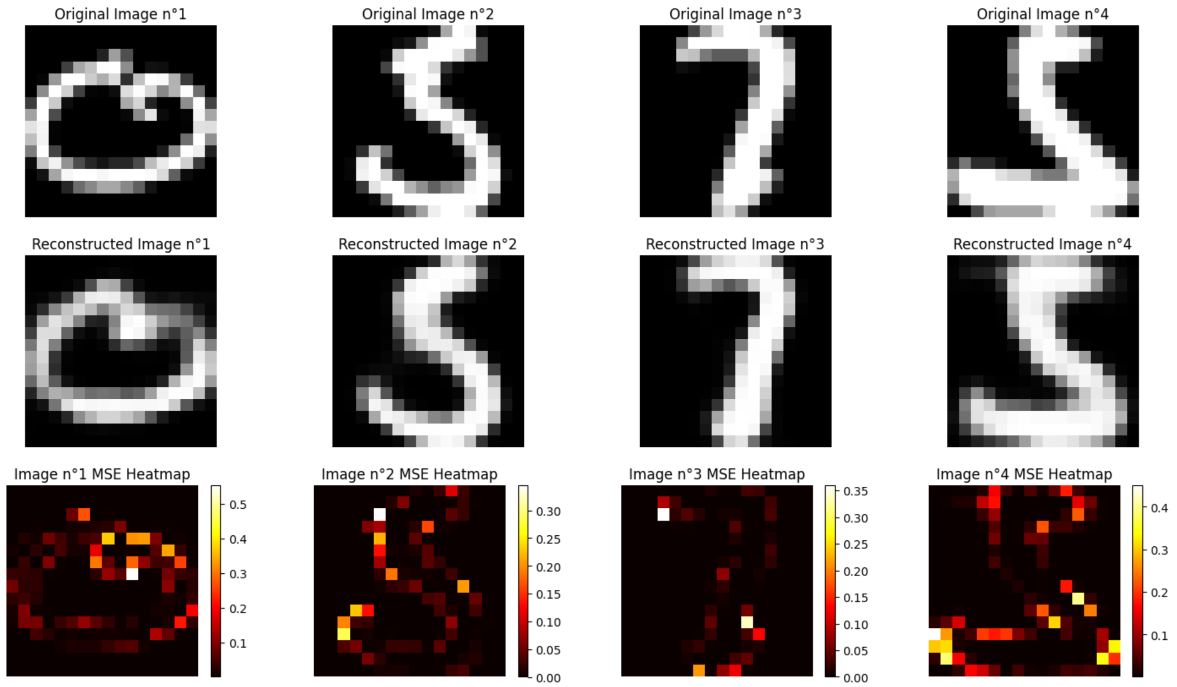
We then proceed to study another digits dataset: USPS. The same preprocessing is applied. The space search and resulting parameters are here different but we still get a model with a training loss that seems to have converged. The reconstructed images also look great. While the heatmaps look very much like the original and reconstructed images, this is likely due to the intensity of the pixels being is either lower or higher than the original ones because the BCE loss sets the pixels intensities to the extreme (close to 0 or 1). This time however, the network has a bit more trouble discriminating the possible digits as the purities are overall lower. However, we can still derive the major classes from the clustering in the latent space.



(a) Training Loss evolution (MSE)



(b) Clustering Contingency Matrix



(c) Reconstructed Images and Heatmaps

Figure 14: Image Reconstruction and Clustering (USPS Dataset)

5.5 Compression

In this section, we are experimenting with the visualization of reconstructed images after significant compression on the USPS digits dataset. One of the objectives of the compression is to save memory space. We use an autoencoder to achieve this compression, reducing the images to a much smaller latent representation before reconstructing them. This approach allows us to evaluate how well the model preserves the important features of the digits while significantly reducing the amount of storage required. By analyzing the reconstructed images, we can assess the trade-offs between compression efficiency and the quality of the visual output.

In order to do this, we proceed the same for the previous sections by implementing a grid search. However, for our hidden layers, we chose to go down to 4 for our last layer. After performing the grid search, our best parameters were: `{'activation_functions': 'Sigmoid', 'batch_size': 64, 'gradient_step': 0.01, 'loss': 'BCELoss', 'n_epochs': 1000, 'n_neurons_per_layer': [256, 100, 4]}`.

After our training process, the training loss (Figure 16) has a nice curve that is steadily converging. So, there should be no major issue with the image reconstruction. Also, the best model achieves a compression rate of 0.02. This means that the original images are compressed by 98% in the latent space which is great.

As expected and seen in the Figure 15, the reconstructed images are close to the originals (as evidenced by the relatively low Kullback-Leibler Divergence score). However, they struggle with the edges, as shown by the heat maps, where the pixel intensity is too diminished compared to the original images (hence the blurring effect). Finally, when we represent the compressed images, we obtain a 2x2 grayscale square, which saves 98% of storage space. While it would be impossible to guess the original digit from this compressed form, once decompressed, the semantics of the image are conveyed: the represented digit is easily discernible, although the image is somewhat blurry.

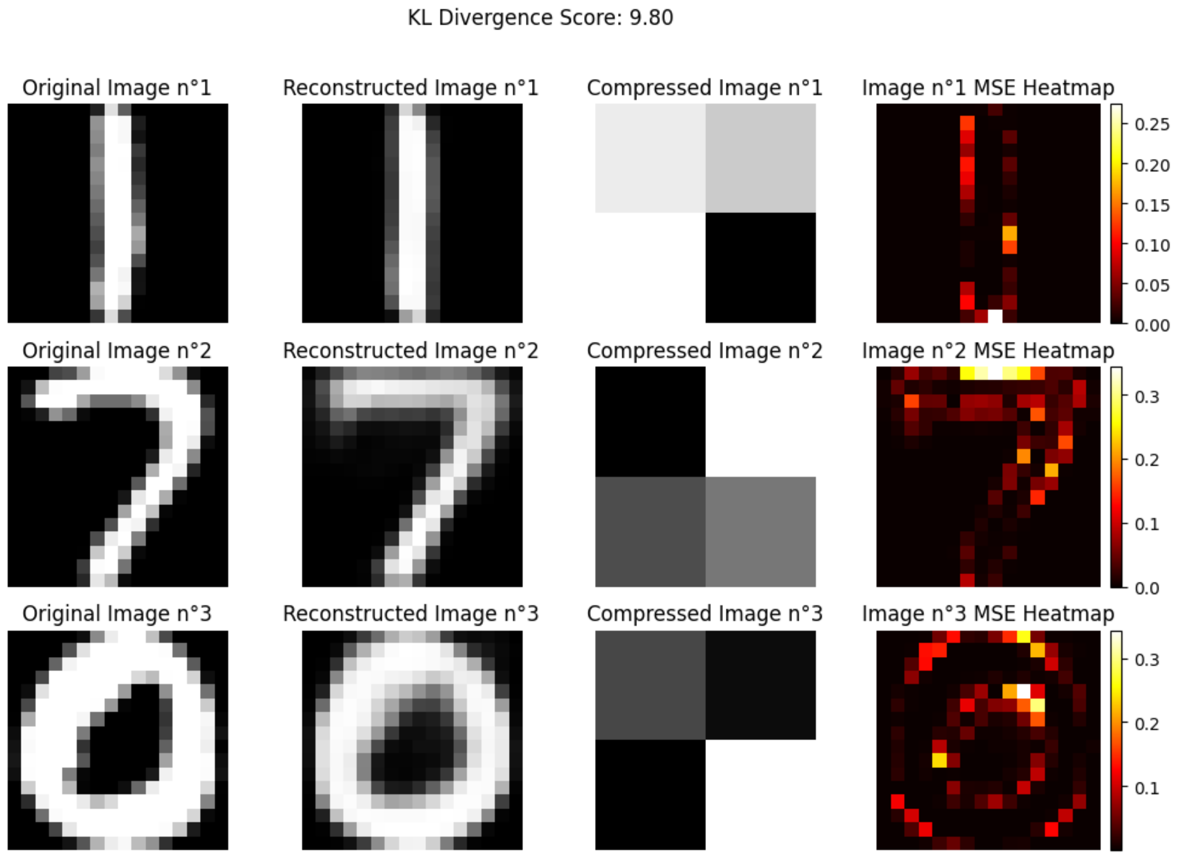


Figure 15: Image Reconstruction, Compression and Heatmaps (USPS Dataset)

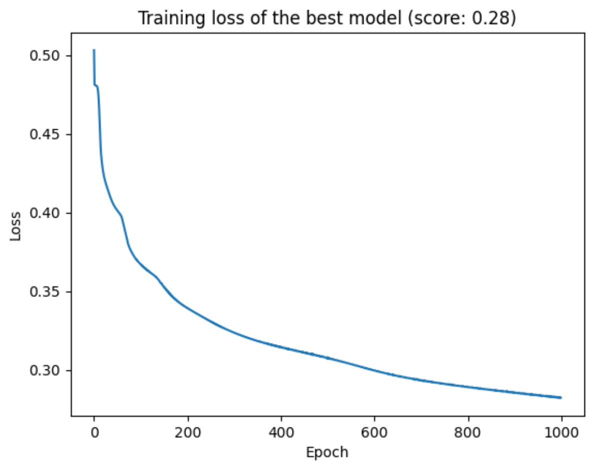


Figure 16: Training loss evolution for compression (USPS Dataset)

6 Convolutional Network

The use of Convolutional Neural Networks (CNNs) offers numerous advantages for image processing. Convolution allows capturing local patterns, reducing the number of learnable parameters, achieving translation invariance, and reducing data dimensionality. This makes it particularly suitable for images, temporal sequences, or signals.

We have successfully implemented one-dimensional convolution as well as a max-pooling layer. The implementation has been optimized by minimizing loops and leveraging advanced functionalities of the `numpy` library (only one loop present in convolution, over the kernel size). Our reasoning is based on using numpy's `np.lib.stride_tricks.sliding_window_view` and `np.einsum` and it can be seen in our code. The main idea is to create a view of the input tensor with a sliding window mechanism, which is a memory-efficient way to access the data. So, for each element in the tensor, the mechanism creates a window of size `k_size` along the second axis (length dimension) and `chan_in` along the third axis (channel dimension).

6.1 Conv1D

The module `Conv1D(`$k\_size, chan\_in, chan\_out, stride$`)` contains a parameter matrix having the size $(k\_size, chan\_in, chan\_out)$, which corresponds to $chan\_out$ filters of size $(k\_size, chan\_in)$. Its forward method takes as input a batch of size $(batch, length, chan\_in)$ and outputs a matrix of size $(batch, \frac{(length-k\_size)}{stride}+1, chan\_out)$.

6.2 MaxPool1D

The `MaxPool1D` module, devoid of learnable parameters, operates with a kernel size (k_size) and a stride length ($stride$). Its forward method accepts input batches of size $(batch, length, chan\_in)$ and yields an output matrix of dimensions $(batch, \frac{(length-k\_size)}{stride}+1, chan\_in)$.

6.3 Flatten

The `Flatten` module transforms input batches of size $(batch, length, chan\_in)$ into a matrix of dimensions $(batch, length \times chan\_in)$.

6.4 ReLU

The ReLU activation function is very useful, especially for convolutional networks because it can speed up the training phase by preventing the exponential growth in the computation required to operate the network. Its expression is simple and applied element-wise:

$$\text{ReLU}(x) = \max(0, x) \tag{13}$$

Its derivative is thus either 0 (if x is negative or zero) or 1.

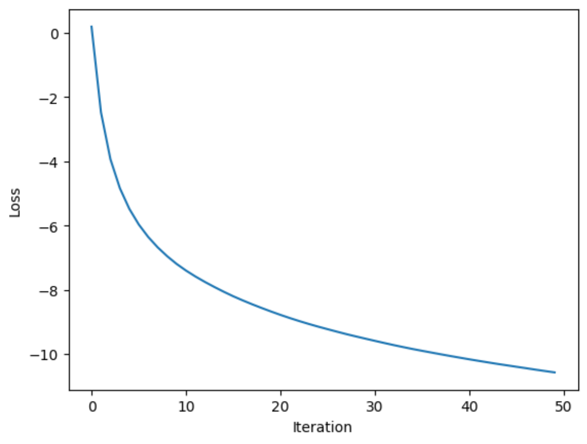
6.5 Experiment

A network using a single convolutional layer of the form `Conv1D(3,1,32)` $\rightarrow$ `MaxPool1D(2,2)` $\rightarrow$ `Flatten()` $\rightarrow$ `Linear(4064,100)` $\rightarrow$ `ReLU()` $\rightarrow$ `Linear(100,10)` performs exceptionally well in digit recognition tasks. However, the training phase is much longer than the other networks.

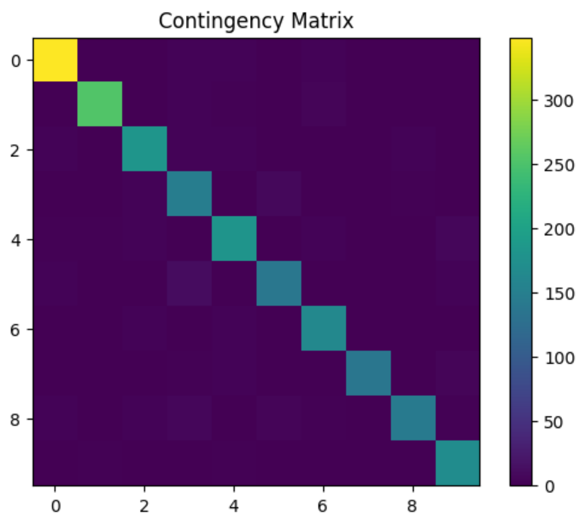
To test the capabilities of such network, we have considered a 64x64 digits dataset. The goal is to identify the digit between 0 and 9. While it is important to one-hot encode the labels, it is also paramount to transform the images in order to match the expected shape. The images only having shades of grey, they only have one color channel. In addition, we performed normalization to ensure that our network is stable.

As the training is significantly longer than previously used networks, we have only trained it over 50 epochs. The resulting training loss (Figure 17) has a great shape and seems to converge quickly despite the large size of the images and the small number of epochs.

In addition, the network recognizes the digits pretty well with an accuracy over 90%. The contingency matrix shows us that errors are mostly due to similar-looking digits (e.g. 3 with 5). It performs well considering that the images are quite big (64x64) and it has only been trained on a few iterations. These results are depicted in Figure 18.



(a) Training Loss evolution (Cross Entropy)



(b) Contingency Matrix

Figure 17: Loss and contingency matrix for CNN and Multiclassification (`n_iter=50`, `gradient_step=1e-4`, `batch_size=32`)

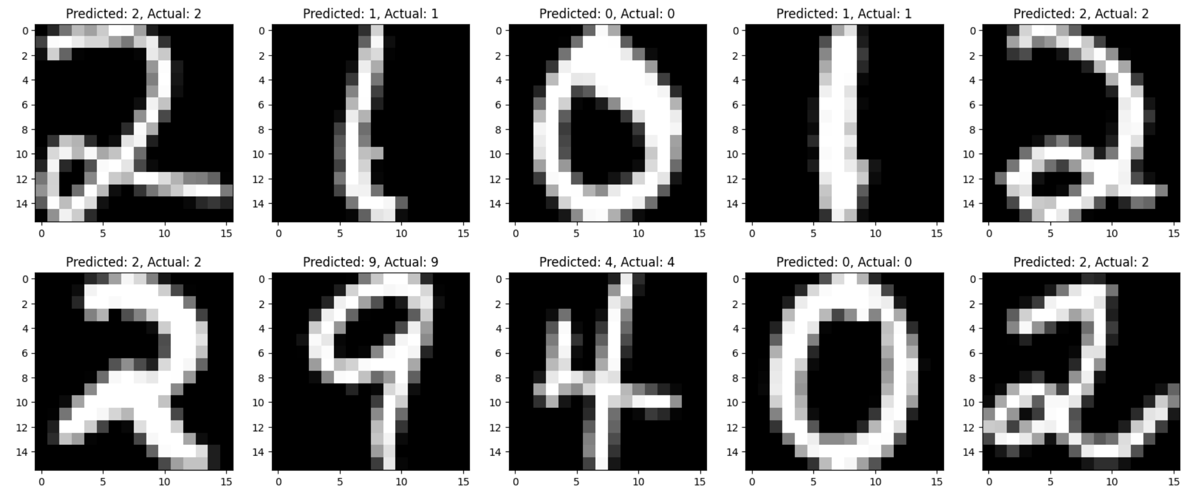


Figure 18: Some examples of classified digits for CNN and Multiclassification (`n_iter=50`, `gradient_step=1e-4`, `batch_size=32`)

Conclusion

Implementing our own deep learning library has been an enriching experience, offering us an in-depth understanding of algebraic concepts and the beauty of backpropagation. Although we explored various tasks such as image reconstruction, we unfortunately did not have time to implement more advanced features and to do further testing.

However, we acknowledge the importance of hardware optimization, such as using GPUs and compilation to improve our models' performance. This could have accelerated computations and allowed us to explore more complex architectures and larger datasets that are much more interesting and fun to use (such as the Olivetti Faces dataset).

In conclusion, although our exploration was limited by time constraints, this experience enabled us to better understand the fundamentals of deep learning and realize the importance of architectural choices, hardware optimization, and implementing advanced features to achieve better performance and explore broader domains.