Supervised Deep Learning

Assignment 1

Francesco Manzali

20/06/2021

Two simple neural network models are developed for solving two supervised problems: function approximation (regression) and image recognition (multi-class classification). Generalization performance during training is evaluated through cross-validation, and improved by using an extensive random grid-search for hyperparameters, both regarding the learning process and the model's architecture.

Introduction

Supervised Learning is a framework of machine learning in which a model learns "by example", inferring useful patterns from a set of inputs (features) paired with their desired output (labels).
Formally, given a *labelled dataset* $D = \{(\boldsymbol{x_i}, \boldsymbol{y_i})\}_{i=1,\dots,N}$, supervised learning trains a model $f\colon \boldsymbol{x} \to \boldsymbol{y}$ such that $f(\boldsymbol{x_i}) \approx \boldsymbol{y_i}\, \forall i = 1, \dots, N$. This is done by iteratively minimizing a *loss function*, which quantifies the difference (error) between predictions $f(\boldsymbol{x_i})$ and known labels $\boldsymbol{y_i}$. For neural networks, the minimization is achieved by the *backpropagation* algorithm, along with an *optimizer* such as Stochastic Gradient Descent.

In this assignment, two supervised learning tasks are examined:

1. **Regression** (sec. 1): 100 samples (x, y) from an unknown scalar function $y = g(x)$ of one variable are given, and supervised learning is used to estimate the form of $g(x)$.
2. **Classification** (sec. 2): 60 000 small (28×28 pixels) labelled images of digits from the MNIST dataset [1] are used to build a classifier, effectively performing a limited optical character recognition.

1 Regression

Learning a single smooth scalar function is usually an easy task. In this case, there are a few specifics of the dataset that make it more challenging:

- Samples are few (only 100)
- The training dataset is biased, in the sense that it does not cover uniformly the function's domain. In fact, two regions around $x = \pm 2$ are left out (fig. 2, pag. 4).

1.1 Methods

Since the dataset is small and the problem is simple, overfitting is highly probable. Thus, it would be best to consider "small" neural networks, i.e. with *few* hidden layers (≤ 5) and *few* neurons per layer (≤ 15).

The exact architecture, along with all the learning hyperparameters and the choice of the optimizer, is found by performing a random grid-search. To mitigate the dataset's small size, cross-validation with $k = 5$ folds is used to estimate generalization performance.

The algorithm is as follows:

1. A set of hyperparameters is sampled.
2. The training dataset is divided in k random subsets of equal size. One is put aside, and the whole training algorithm is run by using just the remaining $k - 1$ sets as a labelled dataset. Then, generalization performance is evaluated on the set that was put aside. This procedure is repeated k times, choosing a different set to put aside at each iteration. Finally, scores (e.g. the *loss* values) are averaged over the k repetitions.

Hyperparameters are used also to choose the model's architecture. The network is always a sequence of fully-connected *Linear* layers, with a non-linear activation in-between. The Mean Squared Error (MSE) is used as the loss function.

Three "schemes" are devised for constructing the network:

- *same*: all layers have the same number of neurons (a random integer in $[2, 15]$). E.g. $(5, 5, 5)$ neurons for 3 hidden layers.
- *rise*: the first hidden layer has 2 to 4 neurons, and each subsequent layer has double the size of the previous one. E.g. $(3, 6, 12)$ neurons for 3 hidden layers.
- *rise-decrease*: same as *rise*, but the "width" of layers increases up to the "middle" layer in the sequence, and then stays the same/decreases by half at each of the deeper layers. E.g. $(4, 8, 8, 4)$ neurons for 4 hidden layers.

The same non-linear activation function, chosen randomly between *ReLU* ($f(x) = \max(0, x)$), *LeakyReLU* ($f(x) = \max(0, x) + 0.01 \min(0, x)$), *Tanh* and *Sigmoid* ($f(x) = [1 + \exp(-x)]^{-1}$), is inserted after each hidden layer. No activation is needed for the output.

The hyperparameters used for learning are the following:

- **Optimizer**: random choice between SGD (with Nesterov momentum), Adam and RMSProp. The last two are more advanced, since they can "adapt" their learning rate automatically.
- (Initial) **learning rate**: a random value λ between 10^{-4} and 1, chosen so that $\log \lambda$ follows a uniform distribution.
- **Batch size**: either 5, 7, 10 or 15.

Three types of regularization are considered, with at most one applied to any given network:

- **Dropout**: a percentage p of neurons from all layers (except the last) are randomly "switched off" during each forward pass of the training procedure, with p between 10^{-4} and 10^{-1} (log-uniform). This reduces the "specialization" of neurons, distributing the computation and making the network more robust. During evaluation, all neurons are activated.
- **L1**: a term proportional to the sum of the absolute values of all weights is added to the loss function. This forces the weights to be smaller, and removes unimportant connections, making the network "sparser". The proportionality factor is chosen between 10^{-5} and 10^{-1} (log-uniform).
- **L2**: a term proportional to the sum of weights squared is added to the loss function, forcing all weights to be much smaller. The proportionality factor is chosen between 10^{-5} and 10^{-1} (log-uniform).

In total, two random-grid searches (500+ trials) are executed: the first over all the search space, and the second over a smaller set of possibilities encompassing the best combinations found during the previous search.

1.2 Results

The final architecture consists of 3 hidden layers with $(8, 16, 8)$ neurons each, a *LeakyReLU* activation function, and the following hyperparameters for learning:

- **Optimizer**: Adam, with 0.0083 as initial learning rate
- No regularization
- Batch size of 7

After 1000 epochs of training (fig. 1), the final training MSE is 0.19 ± 0.02, while the validation MSE is 0.3 ± 0.2.

Then, the model is re-trained over the whole 100 samples of the training dataset for 1000 epochs, and finally evaluated on a new "test" dataset, which was never used before. The results are plotted in fig. 2. The performance is very good, with the test MSE (~ 0.1) lying at the lower end of the MSE estimated during validation (0.3 ± 0.2), probably due to the fact that the test dataset seems to be "less noisy" than the training one. Moreover, the two regions with no training samples, which contain two local maxima, are well-reconstructed. The estimation, however, is significantly lower than the ground-truth in these subsets, because it is biased to follow the neighboring (lower) points.

The output of each neuron is shown in fig. 3. Note how the complexity is higher the deeper is the layer. At the beginning, neurons can only differentiate low/high inputs, but already at the second layer they begin to specialize to model local maxima. In layer 3 the output of several neurons already looks like the fitted function, while others do not activate at all. This opens the possibility of removing neurons, "pruning" the network, and reducing its computational cost while not affecting its performance. A histogram of all the weights of each layer is reported in the appendix (fig. 7, pag. 7).

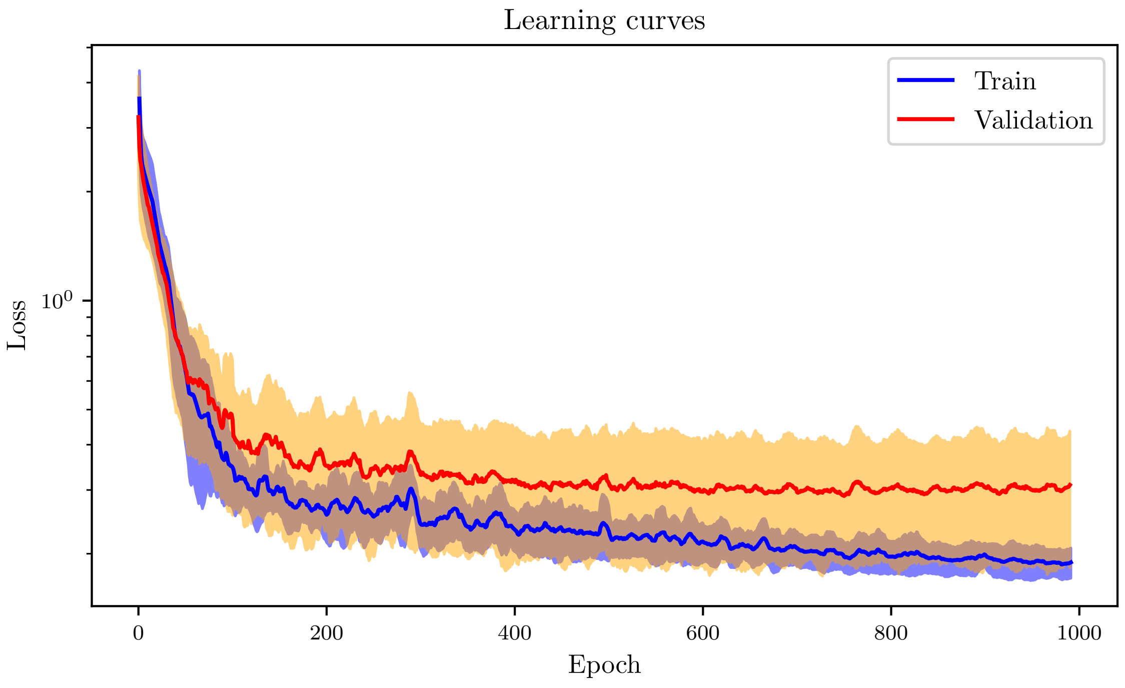

Figure (1) – Learning curves for the regression task, with the best hyperparameters found by the random grid-search. Training and validation losses are averaged over a 5-fold cross-validation, with the filled areas representing ± 1 std variation. Note that the model converges around 400 epochs, with the validation loss remaining stationary thereafter, and the training loss slightly decreasing (but remaining still within ± 1 std of the validation one).

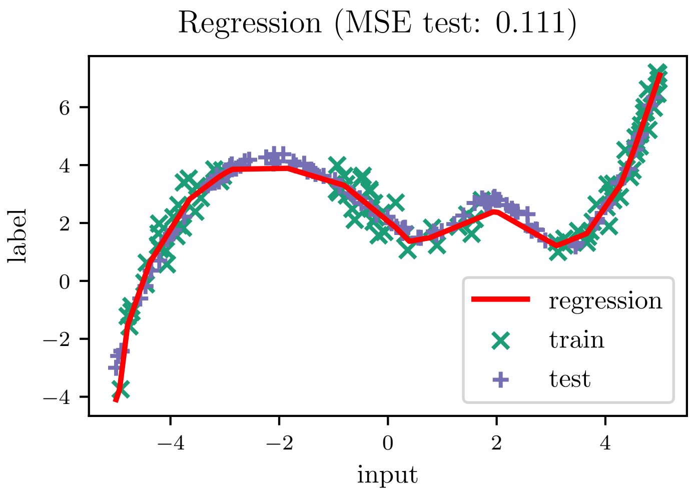

Figure (2) – Final results for the regression task.

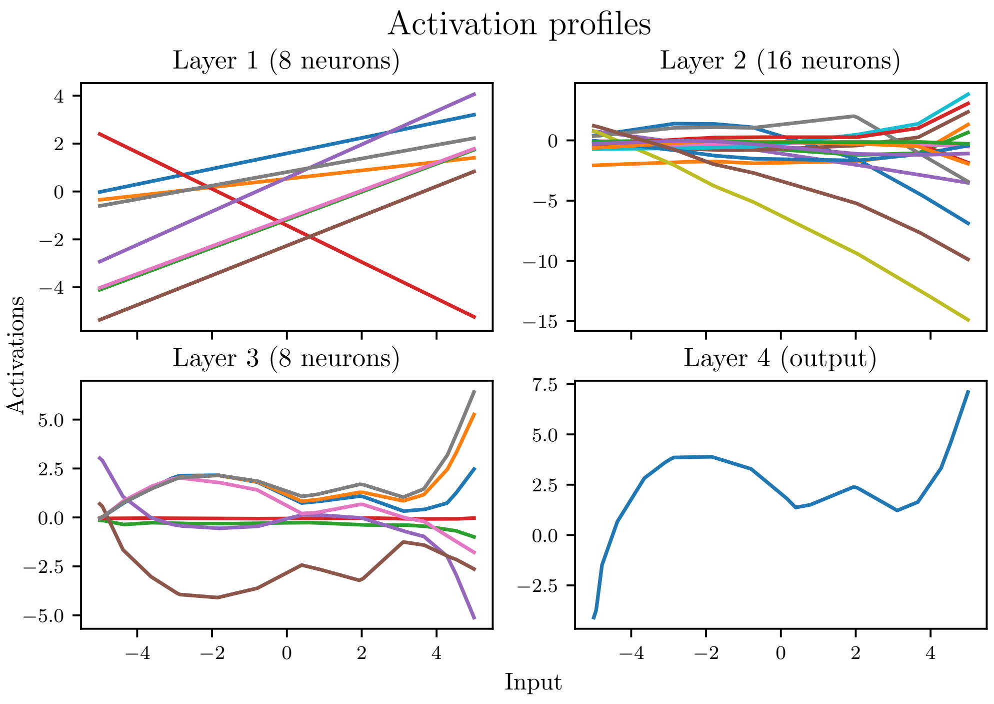

Figure (3) – Activation profiles for all layers in the network.

2 Classification

2.1 Methods

MNIST is a classic dataset for image recognition, consisting of 70 000 images of digits of 28×28 pixels in size. Only a subset of it is used for training a multi-class classifier, while the rest is reserved for estimating the generalization performance. Specifically:

- The first 60 000 samples are used during training. 80% of these (48 000) are actually shown to the network to optimize its weights (*training dataset*), while the remaining 20% (12 000) are used to estimate the performance during hyperparameter optimization (*validation dataset*).
- The remaining 10 000 samples form a *test dataset* used only at the very end to estimate the generalization performance of the model.

The training dataset is *augmented* by adding a random simple affine transformation to each batch during learning (random rotations of up to ±10°, translations of up to ±10% and scaling of up to ±10%). This means that, effectively, the model always sees different samples (dynamic data augmentation). Both the validation and test datasets are not affected in any way.

The examined model is a Convolutional Neural Network with the fixed architecture shown in fig. 4, chosen for its simplicity.

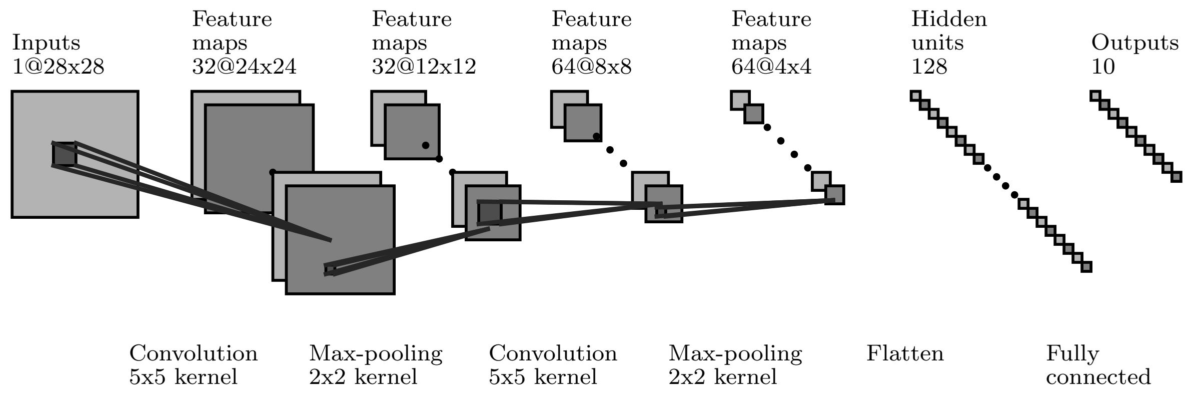

Figure (4) – CNN Architecture. Image drawn with gwding/draw_convnet.

A random grid search is used for choosing the hyperparameters for learning:

- **Optimizer**: either Adam or SGD (with Nesterov momentum)
- **Learning rate**: a value in $[10^{-4}, 10^{-2}$ (log-uniform)
- **Batch size**: either 64, 128 or 256
- **Activation function**: either *ReLU* or *LeakyReLU*.
- **Regularization**: either *dropout* (in $[.05, .65]$, log-uniform, added after each hidden layer), *L1* (in $[10^{-6}, 10^{-4}]$, log-uniform), *L2* (in $[10^{-6}, 10^{-4}]$, log-uniform) or *none*.

2.2 Results

The best hyperparameters are found to be: Adam with $\lambda = 6 \times 10^{-4}$, ReLU activations, batch size of 64 and no regularization. However, note that the grid-search evaluates each choice of hyperparameters over just 10 epochs, to avoid taking too long. Since overfitting is not likely to happen immediately, the search suggests not to use regularization. But, at the end, one would like to train the model for longer, to maximize its performance. In that case, some amount of regularization must be considered.

Specifically, a small dropout value of 0.3 is added, and the network is trained following all the other hyperparameters for 100 epochs. Moreover, if the validation loss does not decrease for 10 consecutive epochs, training is stopped prematurely. This idea, known as "early stopping", helps to prevent overfitting.

After the final training, the model reaches 99.73% of accuracy on the validation dataset, and 99.56% on the test dataset. The learning curves are shown in the appendix (fig. 9, pag. 8), along with the histograms of the weights (fig. 8), the confusion matrix (fig. 11) and some examples of misclassifications (fig. 12).

To see what the network is doing, it is useful to plot some filters. For instance, fig. 5 shows the kernels of the first convolutional layer. These appear to be mostly edge detectors: filter 24 detects horizontal lines, filter 15 oblique ones, and filter 17 vertical ones. Their effect is more apparent when they are applied to a sample, as shown in fig. 6.

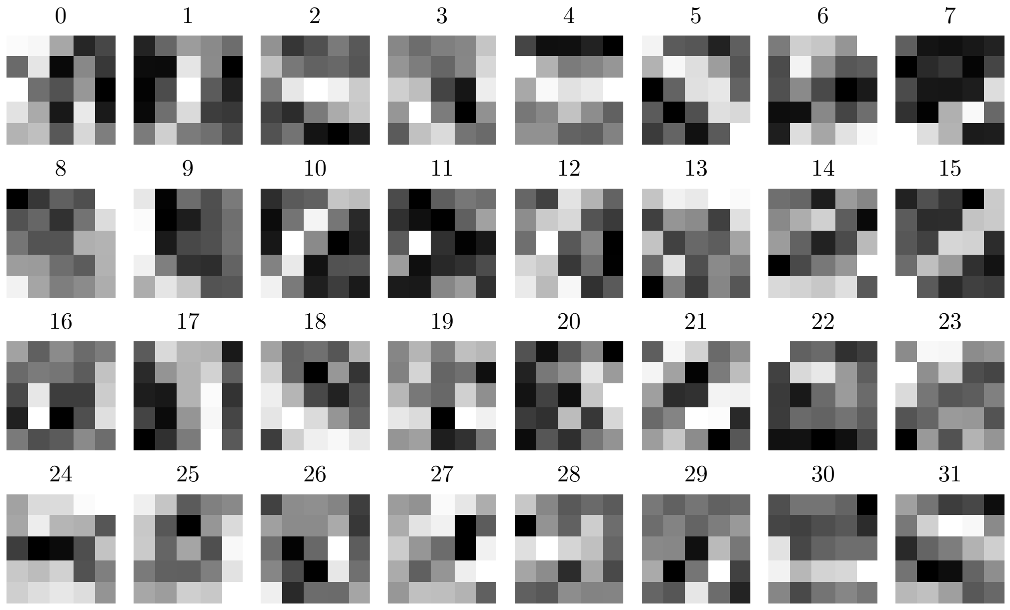

Figure (5) – Plot of the weights from the filters in the first convolutional layer.

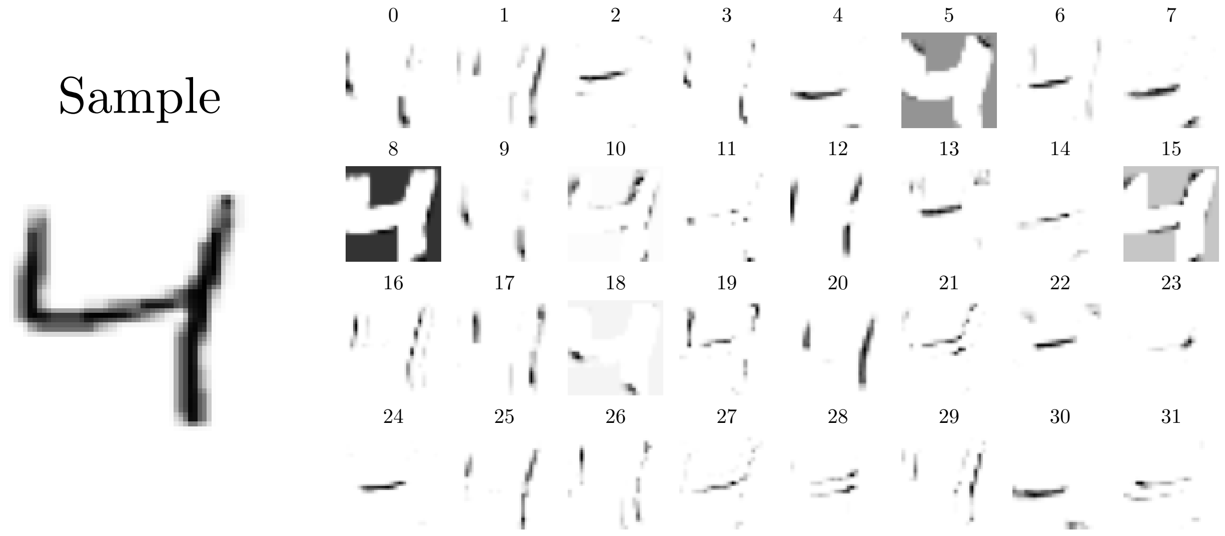

Figure (6) – Feature maps $0 - 31$ obtained by applying the filters from fig. 5 to the sample shown to the left.

References

[1] Y. Lecun et al. "Gradient-Based Learning Applied to Document Recognition". *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324. ISSN: 1558-2256. DOI: 10.1109/5.726791.

Appendix

This section contains some additional figures, showing a few more glimpses on the performance of the models and their inner workings.

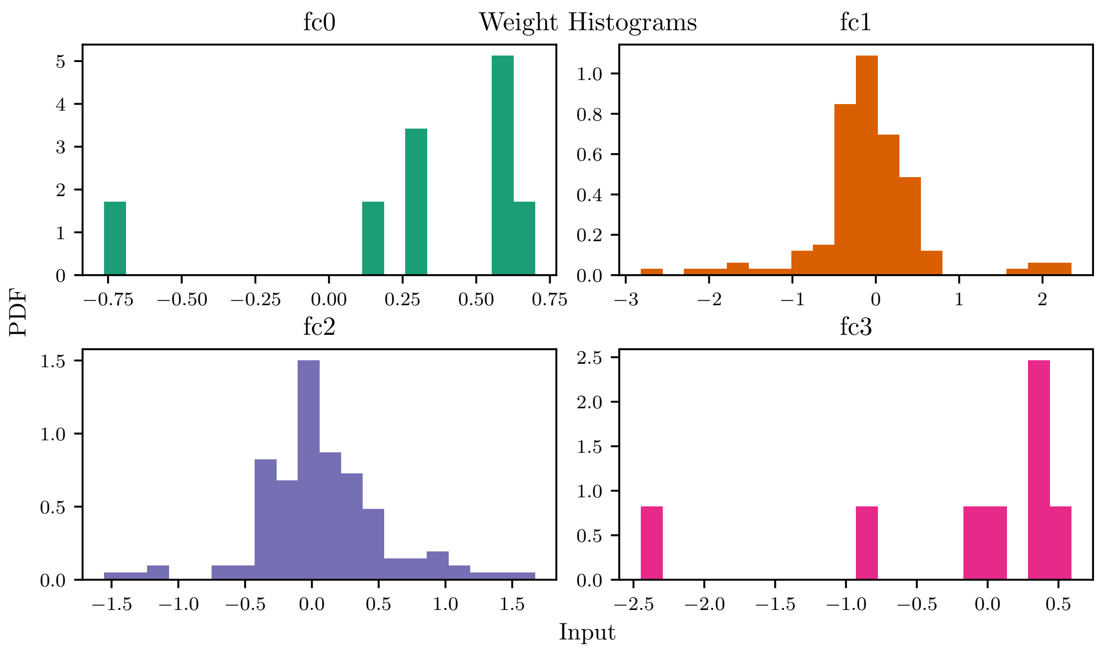

Figure (7) – Histograms of weights for the neural network used in the regression task.

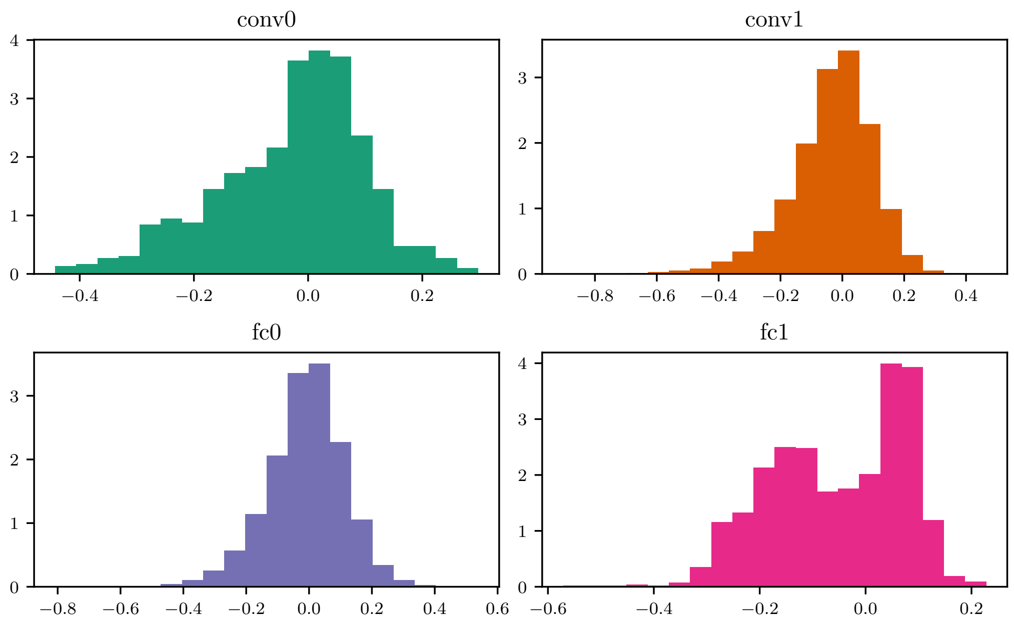

Figure (8) – Histogram of weights for the neural network used in the classification task.

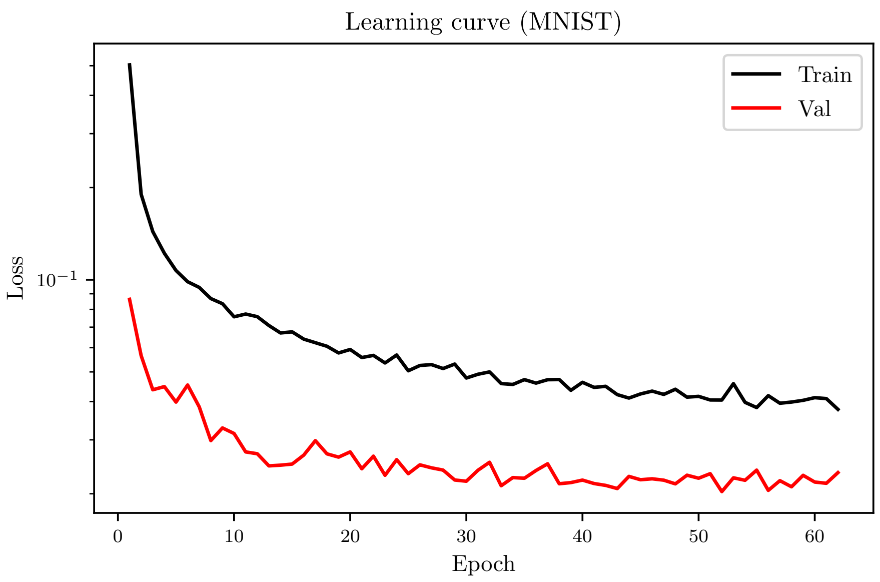

Figure (9) – Learning curve for the final training of the MNIST classifier. Note that the validation curve is significantly lower than the training one. This is because the model uses dropout for regularization, which is active just for the training part. Moreover, training samples are always different at each epoch, due to the random transformations of the augmentation procedure, while the validation dataset is always the same.

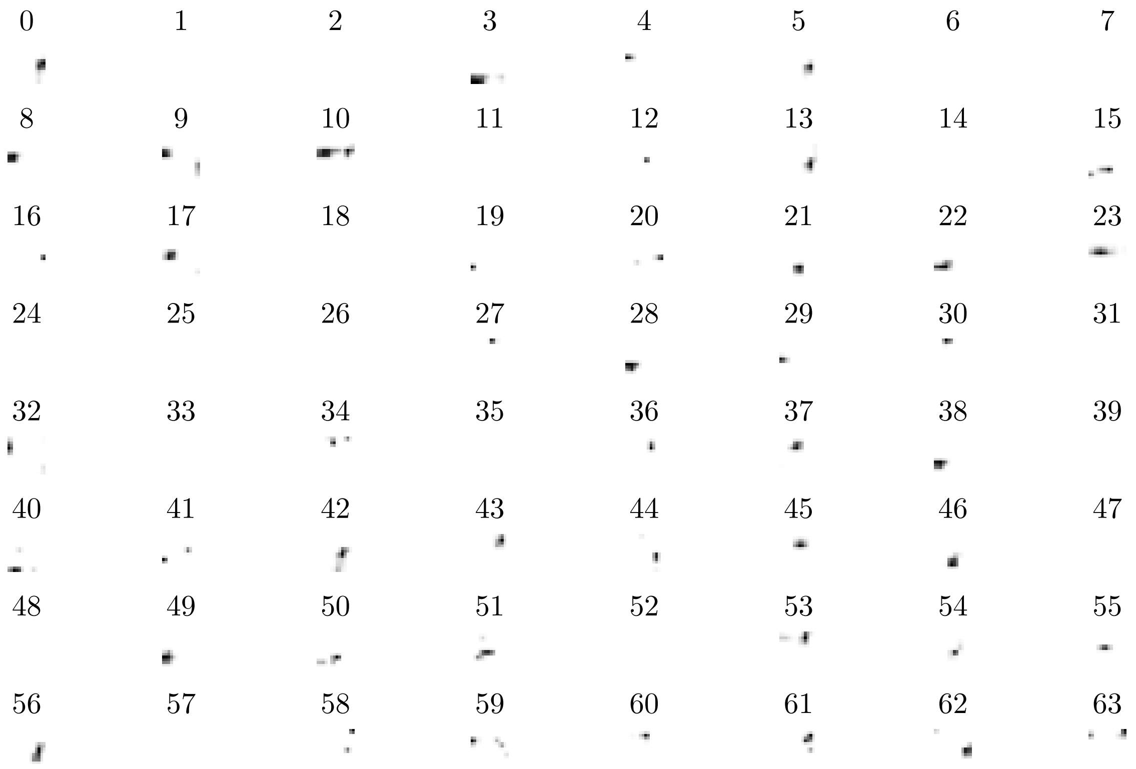

Figure (10) – Feature maps of the second convolutional layer of the CNN used for the classification task, with the same sample from fig. 6. Since this layer is preceded by a maxpool operation, the maps are much smaller.

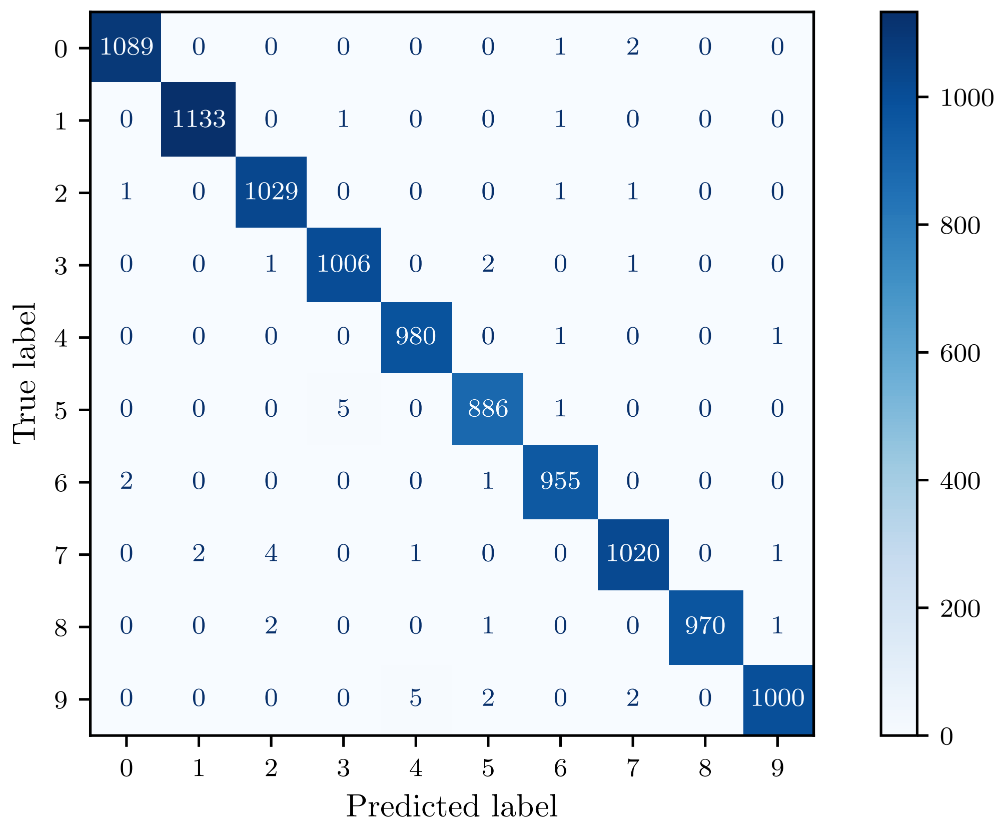

Figure (11) – Confusion matrix for the MNIST classifier.

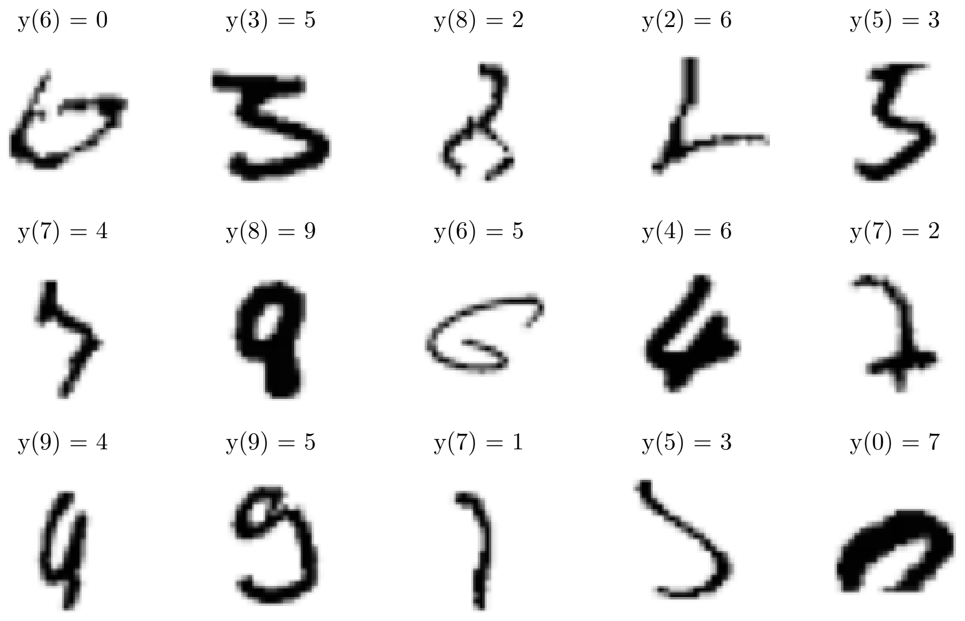

Figure (12) – Some of the samples from the test dataset that are misclassified by the CNN model, labelled as y(true_label) = wrong_predicted_label.