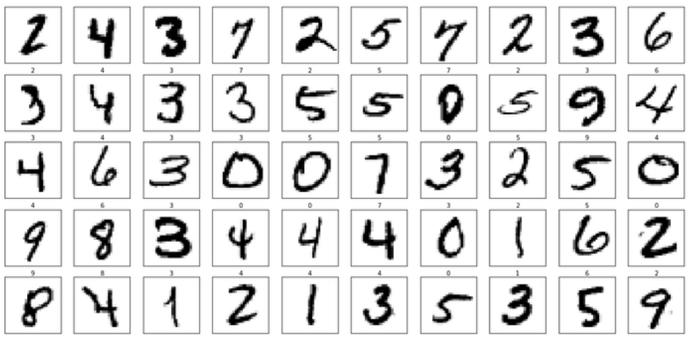
ESIEE Paris

Handwritten digits recognition

E4FI – Group 2



Alex Foulon & Erwan Gautier
28/05/2024

Table of Contents

Introduction

The field of image classification has witnessed significant advancements over the past few decades, driven largely by the development of deep learning techniques. Among these techniques, both multilayer perceptrons (MLPs) and convolutional neural networks (CNNs) have become fundamental tools. This project aims to compare the performance of these two types of neural networks in the context of image classification, using the MNIST dataset as a benchmark.

The MNIST dataset, introduced by Yann LeCun, a notable graduate from ESIEE Paris and his colleagues in the late 1990s, is a widely used dataset consisting of 60,000 training images and 10,000 testing images of handwritten digits, each sized 28x28 pixels. It has become a standard for evaluating the performance of image classification algorithms due to its simplicity and the variety of methods tested on it over the years.

In this study, we will implement and evaluate two models: a multilayer perceptron without convolutional layers and a convolutional neural network. By comparing these models, we aim to understand the strengths and weaknesses of each approach and provide insights into their applicability for different types of image classification tasks. This comparison will help in determining which architecture offers better performance in terms of accuracy and computational efficiency when applied to the MNIST dataset.

The perceptron

The goal of deep learning is to model and make the machine intelligent by drawing inspiration from the functioning of the brain. The first step was to model a neuron, leading to the creation of the perceptron by Frank Rosenblatt in 1957. The perceptron is the basic unit of neural networks. It is a model for binary classification, capable of linearly separating two classes of points. The perceptron can be seen as the simplest type of neural network. This type of neural network does not contain any cycles; it is a feedforward neural network.

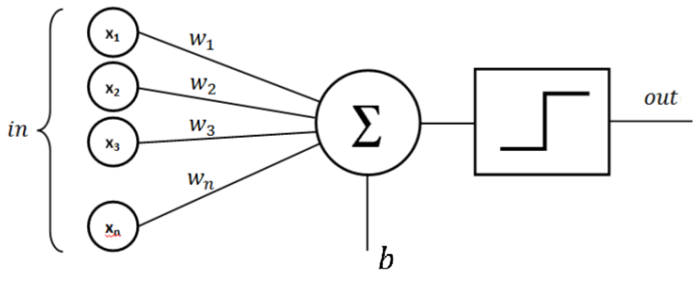


Figure 1 : The perceptron model

To calculate the decision boundary between the two classes, one must first establish a linear combination of the inputs, weighted by their respective weights. The formula for this combination is:

$$z = w1\ x1 + w2\ x2 + \cdots + wn\ xn + b$$

Where $x1$,$x2$,...,xn are the input variables, $w1$,$w2$,...,wn are the weights associated with each variable, and b is a bias term.

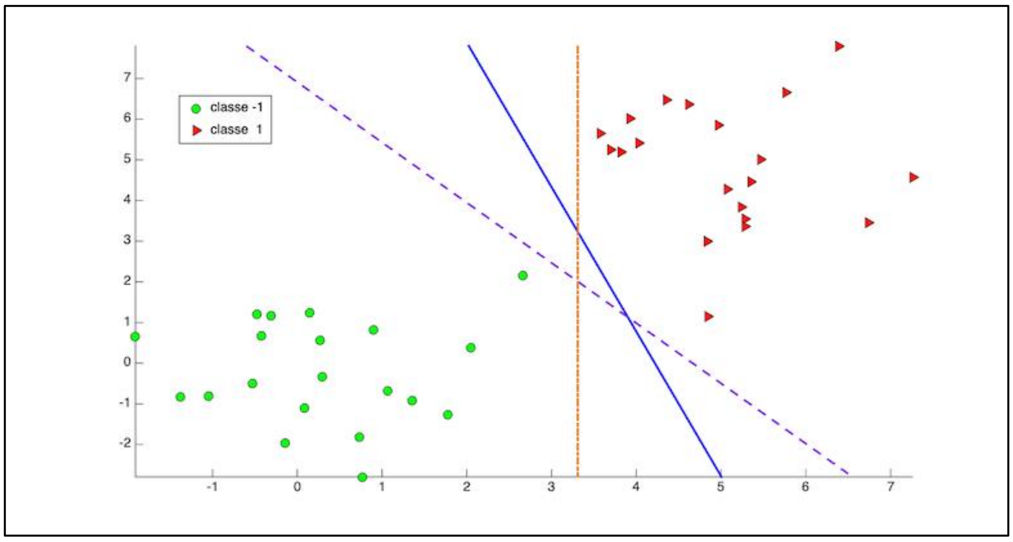


Figure 2 : Linearly separable classes

The hyperplane serving as the decision boundary is defined by the equation where the weighted sum z equals zero:

$$w1\ x1 + w2\ x2 + \cdots + wn\ xn + b = 0$$

This equation represents a hyperplane in an n-dimensional space (where n is the number of input variables). This hyperplane divides the space into two half-spaces. Consequently, one half-space corresponds to one class and the other half-space to the other class. Therefore, if z is negative, our result belongs to the red class, and if z is positive, it belongs to the green class.

However, depending on the neuron's output, it may be beneficial to add an activation function. For example, the sigmoid function transforms activation values into a probability score ranging between 0 and 1. There is also the Heaviside function, seen in figure 1, which does not produce a percentage but rather a binary output (0 or 1). This means that the output does not indicate a degree of certainty or a percentage, but rather a categorical classification.

To quantify errors made by a model, we use a cost function, which helps measure the difference between the model's predictions and the actual observations from the dataset used during training. There are also several other metrics and methods for evaluating a model's performance, such as Mean Squared Error (MSE) and Log Loss, each providing specific insights into different aspects of model accuracy.

Gradient descent involves adjusting the parameters W (representing all the weights) and b to minimize the model's errors, i.e., to minimize the Cost Function. To do this, it is necessary to determine how this function varies with different parameters. This is why we calculate the Gradient (or derivative) of the Cost Function. The derivative indicates how the function changes. Here is the formula for gradient descent:

$$Wt + 1 = Wt - \alpha \frac{\partial Wt}{\partial L}$$

By repeatedly applying this formula, we can reach the minimum for a convex cost function, while in the case of a non-convex function, we can only achieve a local minimum instead of the global minimum of the function. The choice of the learning rate α affects the convergence of the algorithm. A rate that is too high or too low can cause divergence or slow down the learning process, respectively.

We observe in red the minimum of each function, noting that the non-convex function has several, including a local minimum and a global minimum.

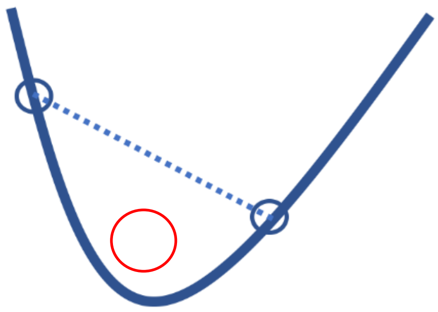


Figure 3: Convex function

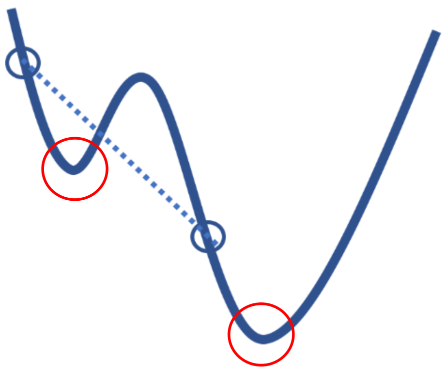


Figure 4: Non-convex function

The perceptron can only linearly separate 2 classes of points. However, the multi-layer perceptron overcomes this limitation.

Multilayer perceptron

In a multilayer perceptron, the neurons of one layer are connected to all the neurons of the following layer, forming a dense network. The weight of each connection is crucial for the network's operation. Optimally configuring these weights is essential for the network to accurately model the problem it is meant to solve.

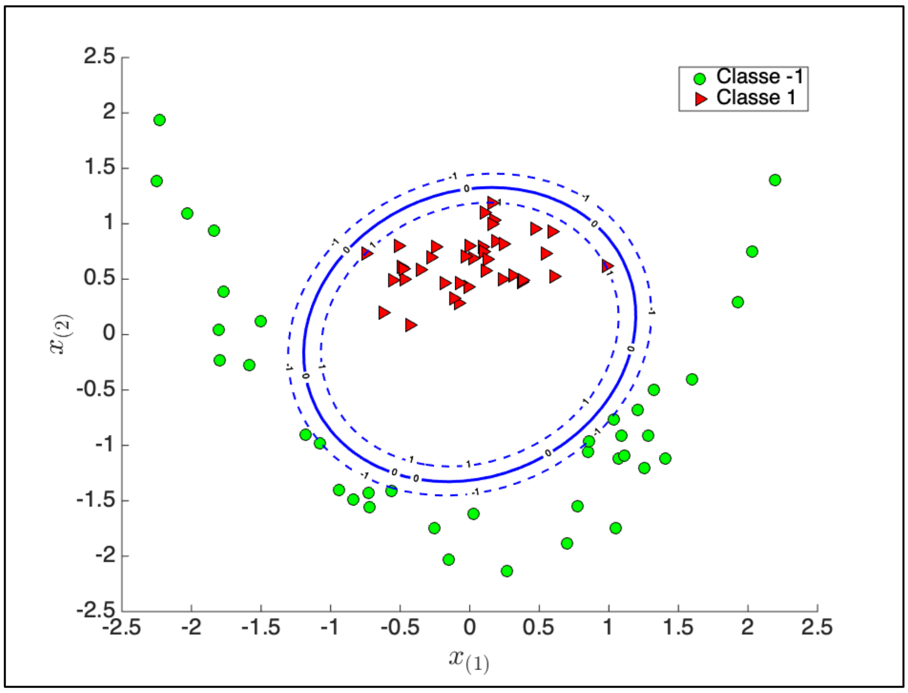


Figure 5 : Non-linearly separable data

The determination of the best weights for the connections is generally done through a backpropagation algorithm. This algorithm involves determining how the network's output varies with the parameters (W, b) in each layer. Using the gradients, one can then update the parameters (W, b) of each layer so that they minimize the error between the model's output and the expected response. Adjustments are made based on the gradient of this cost function, moving from the output back towards the network's inputs, hence the term "backpropagation".

In summary, these steps are repeated in a loop:

1. Forward Propagation
2. Cost Function
3. Backward Propagation
4. Gradient Descent

Evolutions in Deep Learning

Over time, the multilayer perceptron model continued to evolve, notably with the introduction of new activation functions such as the logistic function, the hyperbolic tangent function, and the ReLU function.

In the 1990s, the first variants of the multilayer perceptron began to be developed. Yann LeCun invented the first convolutional neural networks, which are capable of recognizing and processing images, by introducing mathematical convolution and pooling filters at the beginning of these networks. It was also during these years that the first recurrent neural networks appeared, which are another variant of the multilayer perceptron and allow for effectively addressing time series problems such as text reading or voice recognition.

First approach : The Multilayer Perceptron

Description of the model

Our first version of the model is a multilayer perceptron (MLP). We built it using the Keras python library. Here is the part where the model is defined :

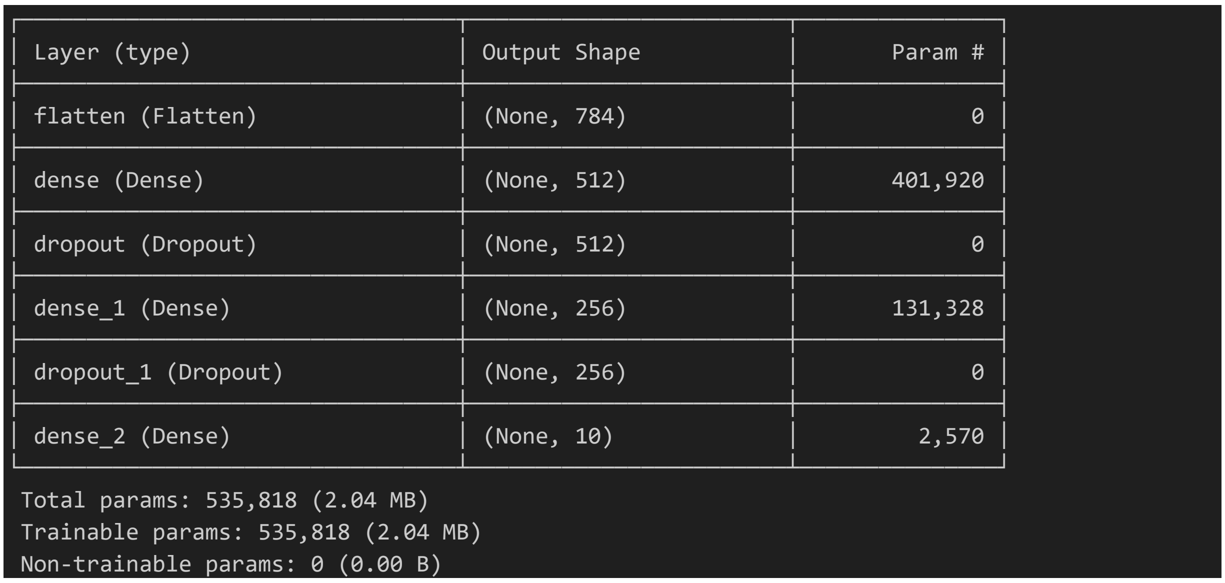
```
num_classes = 10
input_shape = (28, 28, 1)

model = keras.Sequential(
[
    keras.Input(shape=input_shape),
    layers.Flatten(),
    layers.Dense(512, activation="relu"),
    layers.Dropout(0.5),
    layers.Dense(256, activation="relu"),
    layers.Dropout(0.5),
    layers.Dense(num_classes, activation="softmax"),
])
```

It consists of 7 layers, each one having its own use :

1. **Input Layer** : This layer defines the shape of the input data, which in this case is an image of size 28x28 pixels with 1 color channel (grayscale).
2. **Flatten Layer** : This layer flattens the 2D image (28x28) into a 1D vector (784 elements). Dense layers expect 1D input. Flattening converts the 2D pixel grid into a format suitable for dense layers, enabling the network to process the image as a single vector.
3. **First Dense Layer** : This layer consists of 512 neurons, each fully connected to the input features. Dense layers are the core components of an MLP. The ReLU (Rectified Linear Unit) activation function introduces non-linearity, allowing the network to learn complex patterns. With 512 neurons, this layer has a sufficient capacity to capture and process detailed features from the input.
4. **First Dropout Layer** : This layer randomly sets 50% of the input units to 0 at each update during training time. Dropout is a regularization technique used to prevent overfitting. By randomly dropping units, the network becomes less sensitive to specific neuron weights, improving it's generalization capabilities.
5. **Second Dense Layer** : This layer consists of 256 neurons, each fully connected to the previous layer's output. Like the first dense layer, this layer further processes and refines the features learned. With fewer neurons than the previous layer, it acts as a funnel, focusing on the most important features.
6. **Second Dropout Layer** : Similar to the first dropout layer, it sets 50% of the input units to 0 during training, further improving the generalization capabilities of the model.
7. **Output Dense Layer** : This layer consists of 10 neurons in this case (one for each digit), each one representing a class. The softmax activation function converts the output to a probability distribution over the classes, which is essential for classification tasks. The network outputs a probability for each digit, and the digit with the highest probability is chosen as the predicted class.

After the model is built, `model.summary()` gives us this output, which describes the architecture of the model :



```
┌─────────────────────────────────┬────────────────────────┬───────────────┐
│ Layer (type)                    │ Output Shape           │       Param # │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ flatten (Flatten)               │ (None, 784)            │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense (Dense)                   │ (None, 512)            │       401,920 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout (Dropout)               │ (None, 512)            │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_1 (Dense)                 │ (None, 256)            │       131,328 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_1 (Dropout)             │ (None, 256)            │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_2 (Dense)                 │ (None, 10)             │         2,570 │
└─────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 535,818 (2.04 MB)
 Trainable params: 535,818 (2.04 MB)
 Non-trainable params: 0 (0.00 B)
```

We can see the size of each layer, along with the number of parameters that are gonna be fine-tuned during the training process.

Results and performance

Let’s take a look at the confusion matrix of the predictions :

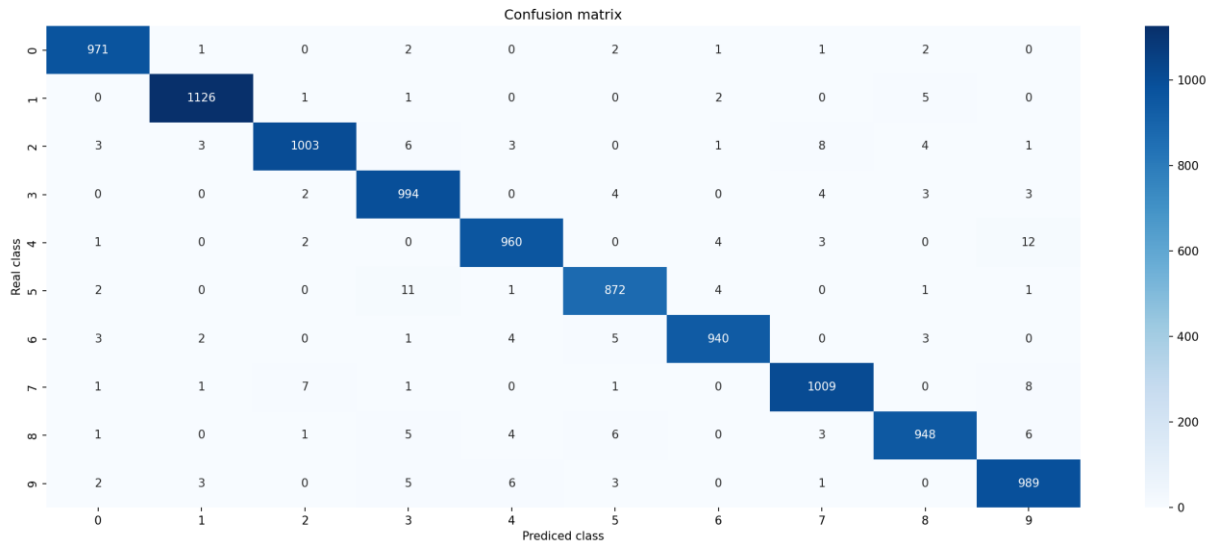


Figure 6 : Confusion matrix, Multilayer Perceptron

The values along the diagonal represent the number of correct predictions for each class, and values off the diagonal represent misclassifications. We can observe that the majority of the predictions are correct as most values are along the diagonal. Misclassifications are minimal and scattered, indicating strong performance.

Now, we will analyse the evolution of the precision and the loss during training and validation :

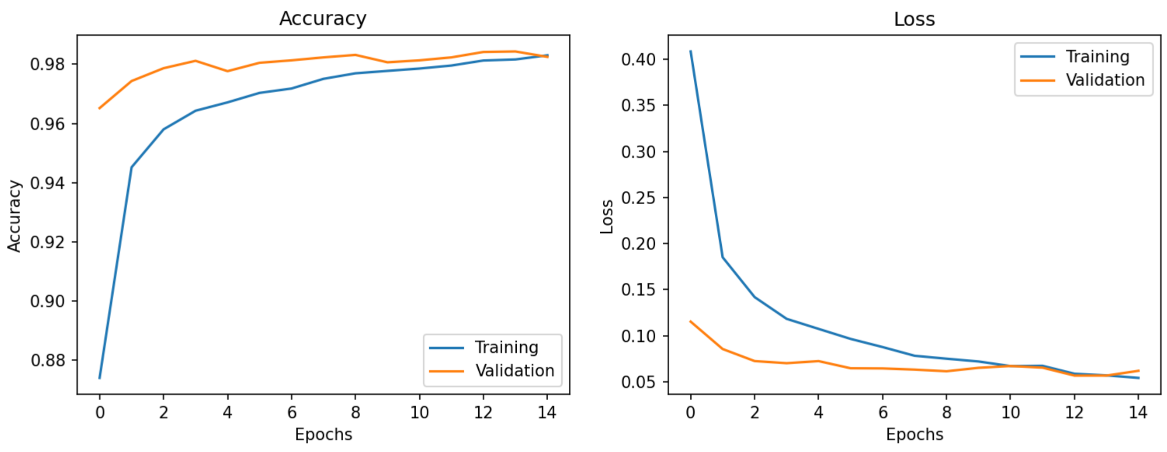
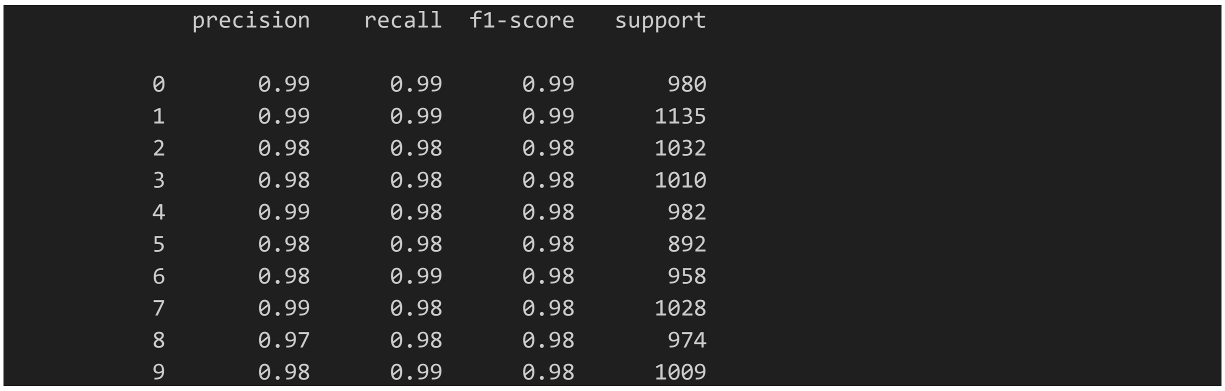


Figure 7 : Accuracy and loss depending on epoch, Multilayer Perceptron

Training Accuracy is increasing steadily and reaches around 98%. Validation Accuracy starts high and remains slightly above training accuracy, indicating good generalization.

Training Loss decreases rapidly initially and then slowly continues to decrease, approaching zero. Validation Loss is decreasing as well, reaching very low values.

Here is the classification report table :



```
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       980
           1       0.99      0.99      0.99      1135
           2       0.98      0.98      0.98      1032
           3       0.98      0.98      0.98      1010
           4       0.99      0.98      0.98       982
           5       0.98      0.98      0.98       892
           6       0.98      0.99      0.98       958
           7       0.99      0.98      0.98      1028
           8       0.97      0.98      0.98       974
           9       0.98      0.99      0.98      1009
```

- Precision is high across all classes (around 0.98-0.99).
- Recall measures the ability to capture all positive samples, and is high across all classes (around 0.98-0.99).
- F1-Score is the harmonic mean of precision and recall. Here, it indicates strong balance between precision and recall for all c lasses (around 0.98-0.99).

To summarize the results, the confusion matrix indicates high accuracy with minimal misclassifications. The graphs show effective learning and good generalization with minimal overfitting. The classification report table confirms strong performance across all metrics.

Overall, the model performs excellently on the MNIST dataset, achieving high accuracy and strong generalization capabilities.

Multilayer Convolutional Neural Network

Description of the model

Our second version of the model is a convolutional neural network (CNN). We built it using the Keras Python library. Here is the part where the model is defined:

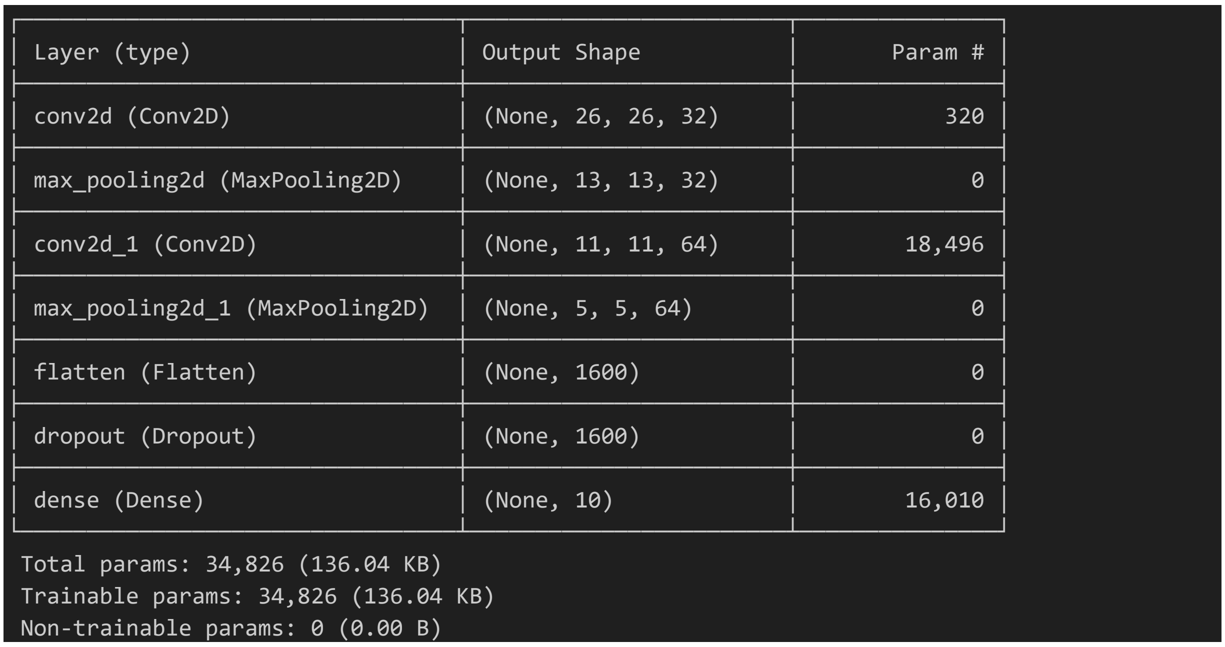
```
num_classes = 10
input_shape = (28, 28, 1)

model = keras.Sequential(
[
    keras.Input(shape=input_shape),
    layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Flatten(),
    layers.Dropout(0.5),
    layers.Dense(num_classes, activation="softmax"),
])
```

It consists of 8 layers, each one having its own use, with some similarities with the previous model :

- **Input Layer** : This layer defines the shape of the input data, which in this case is an image of size 28x28 pixels with 1 color channel (grayscale).
- **First Convolutional Layer** : This layer consists of 32 filters, each of size 3x3. The convolution operation scans the image with these filters to detect local patterns such as edges. The ReLU activation function introduces non-linearity, enabling the network to learn complex features.
- **First MaxPooling Layer** : This layer reduces the spatial dimensions of the feature maps produced by the previous convolutional layer. Using a pool size of 2x2, it downsamples the input, reducing the number of parameters and computational load, which helps in controlling overfitting.
- **Second Convolutional Layer** : This layer consists of 64 filters of size 3x3. It further processes the downsampled feature maps, allowing the network to learn more abstract and complex features.
- **Second MaxPooling Layer** : Similar to the first max-pooling layer, this layer reduces the spatial dimensions of the feature maps generated by the second convolutional layer. This additional downsampling helps in further reducing the model's computational complexity and controlling overfitting.
- **Flatten Layer** : This layer flattens the 3D feature maps (output from the last max-pooling layer) into a 1D vector, enabling the network to process the entire image as a single vector.
- **Dropout Layer** : This layer randomly sets 50% of the input units to 0 at each update during training time, improving its generalization capabilities.
- **Output Dense Layer** : This layer consists of 10 neurons (one for each digit), each one representing a class. The softmax activation function converts the output to a probability distribution over the classes. The digit with the highest probability is chosen as the prediction.

`model.summary()` describes the architecture of the model :



```
┌─────────────────────────────────┬────────────────────────┬───────────────┐
│ Layer (type)                    │ Output Shape           │       Param # │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d (Conv2D)                 │ (None, 26, 26, 32)     │           320 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d (MaxPooling2D)    │ (None, 13, 13, 32)     │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_1 (Conv2D)               │ (None, 11, 11, 64)     │        18,496 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_1 (MaxPooling2D)  │ (None, 5, 5, 64)       │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ flatten (Flatten)               │ (None, 1600)           │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout (Dropout)               │ (None, 1600)           │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense (Dense)                   │ (None, 10)             │        16,010 │
└─────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 34,826 (136.04 KB)
 Trainable params: 34,826 (136.04 KB)
 Non-trainable params: 0 (0.00 B)
```

We can see the size of each layer, along with the number of parameters that are gonna be fine-tuned during the training process. Note that there are a lot less trainable parameters in this model.

Results and performance

Let’s take a look at the confusion matrix of the predictions :

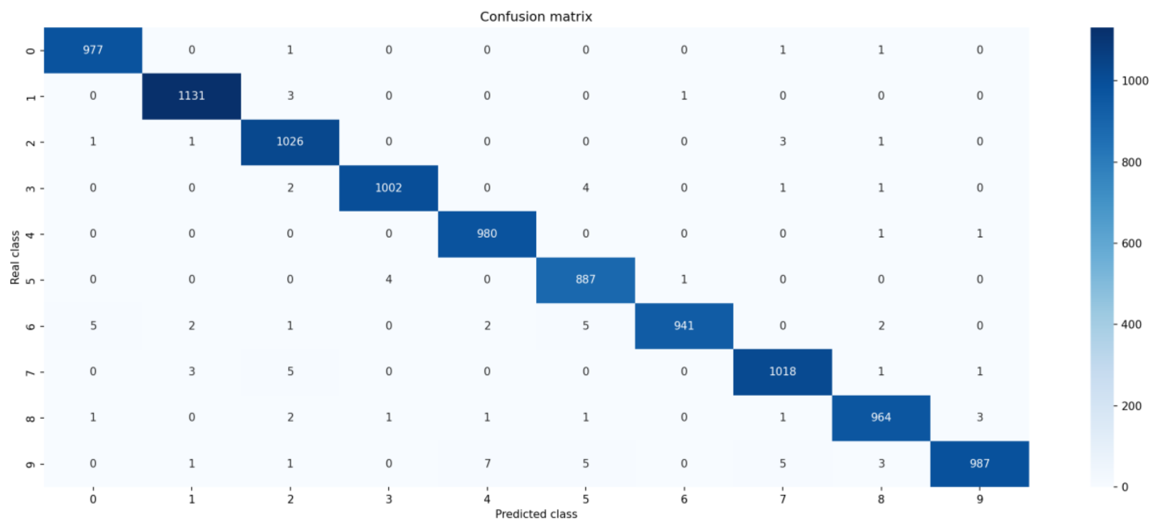


Figure 8 : Confusion matrix, Convolutional Neural Network

As with the first model, we can observe that the majority of the predictions are correct as most values are along the diagonal. Misclassifications are minimal and scattered, indicating strong performance.

Now, we will analyse the evolution of the precision and the loss during training and validation :

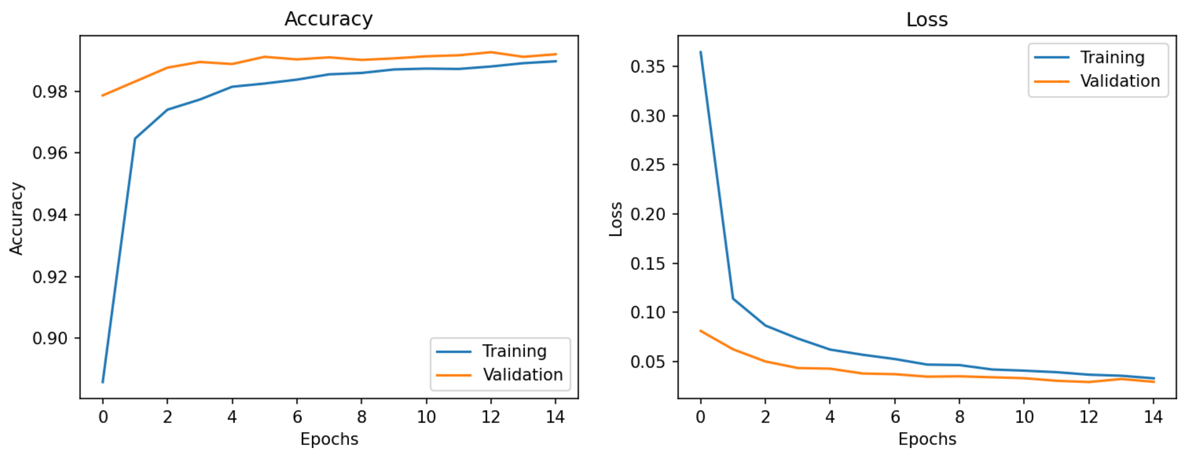
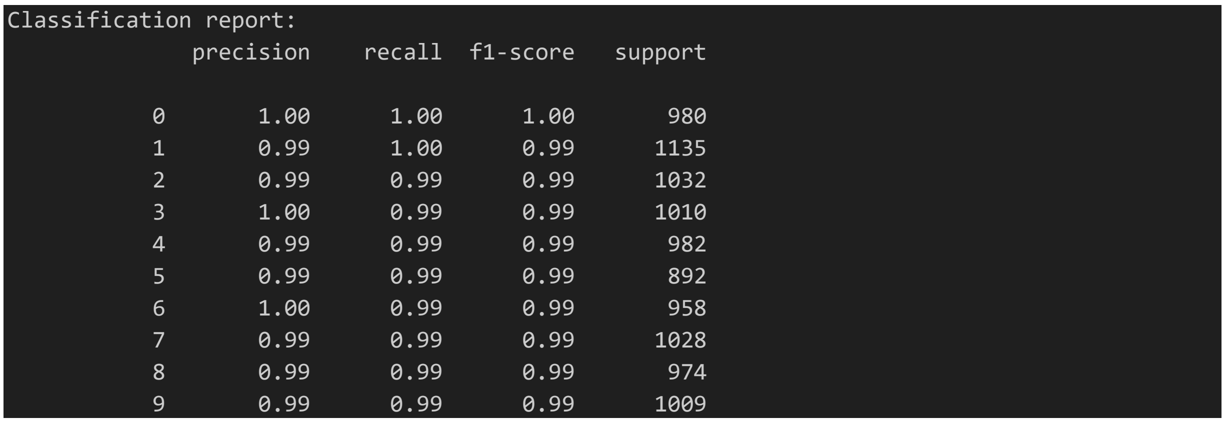


Figure 9 : Accuracy and loss depending on epoch, Convolutional Neural Network

Training Accuracy is increasing steadily and reaches 99%. Validation Accuracy starts high and remains slightly above training accuracy, indicating good generalization.

Training Loss decreases rapidly initially and then slowly continues to decrease, approaching zero. Validation Loss is decreasing as well, reaching very low values.

Here is the classification report table :



```
Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       980
           1       0.99      1.00      0.99      1135
           2       0.99      0.99      0.99      1032
           3       1.00      0.99      0.99      1010
           4       0.99      0.99      0.99       982
           5       0.99      0.99      0.99       892
           6       1.00      0.99      0.99       958
           7       0.99      0.99      0.99      1028
           8       0.99      0.99      0.99       974
           9       0.99      0.99      0.99      1009
```

- Precision is near perfect across all classes (0.99 - 1.00).
- Recall is near perfect across all classes (0.99 - 1.00).
- F1-Score indicates strong balance between precision and recall for all classes (0.99 - 1.00).

To summarize the results, the confusion matrix indicates high accuracy with minimal misclassifications. The graphs show effective learning and good generalization with minimal overfitting. The classification report table shows near perfect performance across all metrics.

Overall, the model performs excellently on the MNIST dataset, achieving high accuracy and strong generalization capabilities.

Training and prediction time

In this section, we benchmark the training times of the two different neural network architectures on a laptop with an RTX 4050 and a Ryzen 9 7940HS. The benchmark focuses on the relationship between accuracy and loss over time during the training process, and includes prediction time.

Multilayer Perceptron :

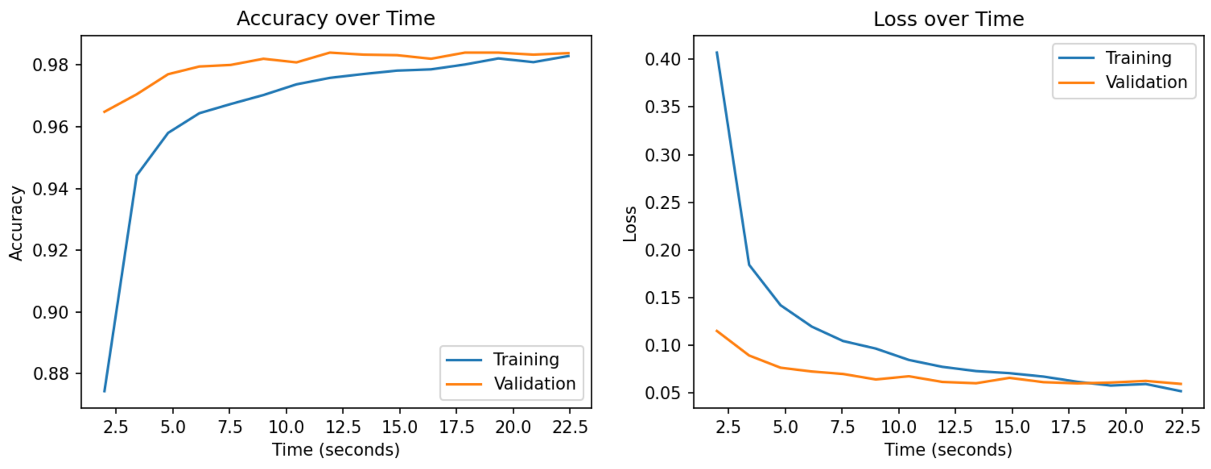


Figure 10 : Accuracy and loss depending on training time, Multilayer Perceptron

The training accuracy of the MLP rapidly increases within the first 5 seconds, reaching around 96%. The accuracy continues to improve gradually, stabilizing at approximately 98% after 20 seconds. Validation accuracy starts high and consistently remains slightly above the training accuracy.

The training loss decreases sharply within the first few seconds, from around 0.4 to below 0.1. The loss continues to decline slowly, approaching near-zero values by the end of the training period. Validation loss remains lower than training loss.

We benchmarked prediction time as well, and got a result of 0.5071 seconds to predict 10000 images.

Convolutional Neural Network :

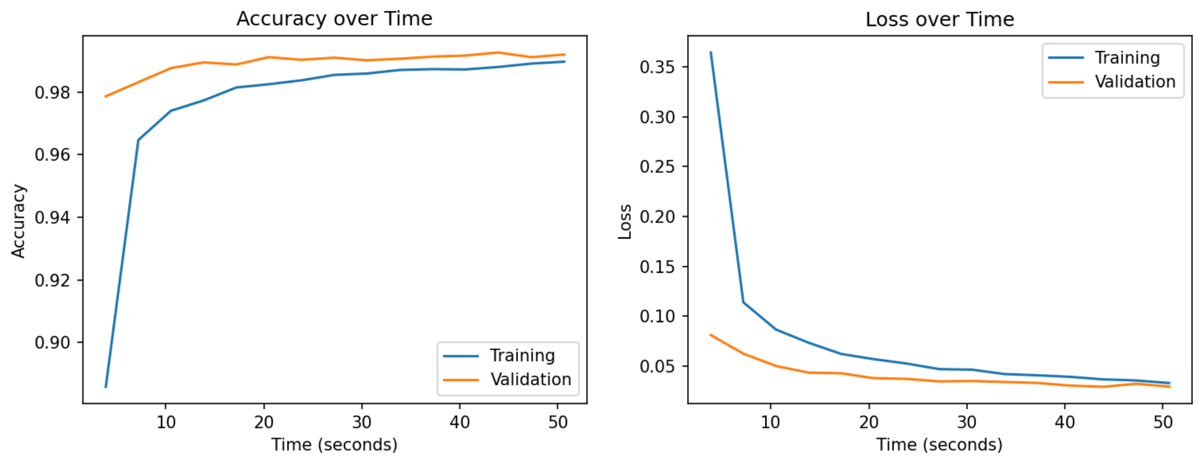


Figure 21 : Accuracy and loss depending on training time, Convolutional Neural Network

The CNN's training accuracy also shows a rapid increase, but it takes longer to stabilize compared to the MLP.It reaches around 97% within the first 20 seconds and gradually approaches 98% by 50 seconds. Validation accuracy starts high and remains slightly above training accuracy, similar to the MLP.

The training loss for the CNN drops quickly from 0.35 to around 0.1 within the first 15 seconds. The loss continues to decrease steadily, nearing zero by the 50-second mark. Validation loss stays consistently below the training loss as well.

As for prediction time, the model needs 0.7936 seconds to predict 10000 images.

Comparative Insights

On our hardware, the MLP converges faster than the CNN, reaching its near-optimal accuracy and loss values in about 20 seconds, whereas the CNN takes around 50 seconds. This difference is expected because MLPs generally have simpler architectures compared to CNNs, which involve more complex computations due to convolutional and pooling operations.

Both models show good generalization, with validation accuracy and loss closely tracking their respective training metrics. The validation metrics for both models are slightly better than the training metrics, indicating effective regularization and no overfitting.

The CNN takes longer to train due to its increased computational complexity from convolutional layers, which process spatial hierarchies in the data. Despite the longer training time, CNNs are typically more effective for image data due to their ability to capture spatial features, as evidenced by the higher final accuracy compared to the MLP.

The CNN is 56.49% slower than the MLP when predicting a bulk of 10000 images. Which is also explained by the increased complexity of the model, leading to higher computation times.

The MLP model trains and predicts faster, achieving high accuracy and low loss in a shorter amount of time. However, while the CNN is slower, it ultimately delivers better performance, achieving a final accuracy of 99%. This higher accuracy results in half the number of errors compared to the MLP, which reaches 98% accuracy.

Hand testing the models

We built an UI that enables manual testing of the models using a drawing canvas.

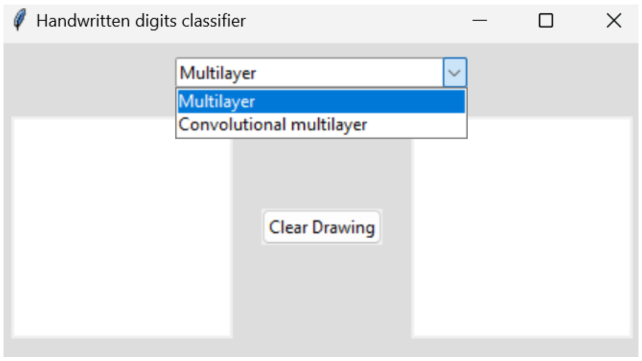
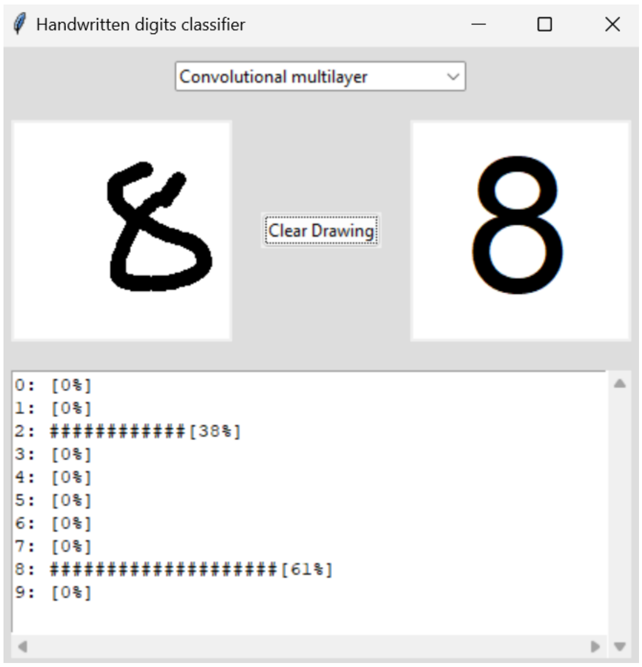


Figure 12 : Selecting the model on the UI

After choosing the right model, we can draw on the left canvas and let the trained neural network predict the number we wrote :



```
0:  [0%]
1:  [0%]
2: ############[38%]
3:  [0%]
4:  [0%]
5:  [0%]
6:  [0%]
7:  [0%]
8: ####################[61%]
9:  [0%]
```

Figure 13 : Testing the model on the UI

The model makes predictions in real time, displaying its confidence for each digit. We owe thanks to Alexis Vapaille and Adrien Pelfresne for suggesting the idea of plotting individual confidences in real time.

The results are quite good. Of course, they don't match the 99% accuracy from our previous benchmarks, since this metric was based on the MNIST dataset symbols, not the ones written by our hands. Still, the results are impressive, especially on the convolutional model.

Conclusion

In this study, we compared the performance of Multilayer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs) for handwritten digit recognition using the MNIST dataset. The MLP, with its simpler architecture, trained quickly and achieved high accuracy in a shorter time. It reached 98% accuracy rapidly, making it efficient for scenarios where training time is critical.

On the other hand, the CNN took longer to train due to its complex architecture involving convolutional and pooling layers. This complexity allowed the CNN to capture intricate spatial patterns, achieving a superior accuracy of 99%. The increased training time and computational cost are balanced by its higher accuracy and fewer errors compared to the MLP.

Our analysis also showed that the CNN was 56.49% slower in predicting 10,000 images than the MLP. Despite this, the CNN's ability to reduce errors significantly makes it a better choice for tasks requiring high precision in image classification.

In summary, while MLPs are suitable for applications requiring quicker training or prediction times, the superior accuracy of CNNs makes them the clear choice for high-precision image classification tasks.

References

https://en.wikipedia.org/wiki/Perceptron

https://en.wikipedia.org/wiki/Multilayer_perceptron

https://en.wikipedia.org/wiki/Linear_separability

https://en.wikipedia.org/wiki/Deep_learning

https://openclassrooms.com/fr/courses/5264666-appliquez-l-apprentissage-statistique-aux-objets-connectes/5361336-realisez-les-machines-a-vaste-marge-svm-pour-la-classification

https://www.youtube.com/playlist?list=PLO_fdPEVlfKoanjvTJbIbd9V5d9Pzp8Rw