Deep Reinforcement Learning - Handwritten Recognition

Lucas SALI–ORLIANGE

Supervised by René NATOWICZ
André CELA

April 2024

Abstract

In this report, we are looking to understand the creation of two handwritten classifier. Each of them are multi-layers network where one is a simple multi-layers network classifier (***Fully Connected Network, "FCN"***) for handwritten recognition, and the second one add a convolutional dimension in it (" ***CNN"***).

Keywords: Convolutional, Neural, Network, CNN, Multi-layer, Fully, Connected, FCN, Handwritten, Recognition, Classifier

Contents

Introduction **3**

MNIST Data Set **4**
- Loading the Data . . . 4
- Preprocessing the Data - FCN . . . 4
- Preprocessing the Data - CNN . . . 5

Classification by Multi-layers Network (FCN) **6**
- Introduction to the FCN . . . 6
- Creating our FCN . . . 7
- Evaluating our FCN . . . 8
- Testing our FCN . . . 9

Convolutional Neural Network (CNN) **11**
- Introduction to the CNN . . . 11
- Creating our CNN . . . 11
- Evaluating our CNN . . . 12
- Testing our CNN . . . 13

List of Figures

1 MNIST Data Set 4
2 FCN Architecture. Source: [mdpi.com] 6
3 Neuron Architecture. Source: [v7labs.com] 6
4 ReLU & Softmax function. Source: [Medium] 7
5 Training and Validation loss and accuracy performance for 10 epochs 9
6 Training and Validation loss and accuracy performance for 20 epochs 9
7 Example of prediction with input and output 10
8 CNN full architecture 11
9 Training and Validation loss and accuracy performance for 5 epochs 13
10 Training and Validation loss and accuracy performance for 10 epochs 13

Introduction

This project was carried out as the final project of the unit "*Deep Reinforcement Learning*" at ESIEE Paris in 2024.

In this project, from the "*MNIST Data Set*" (stands for **M**odified National **I**nstitute of **S**tandards and **T**echnology), the aim of this plot is to meet several challenges:

- **Multi-Layer Network (FCN):**
  - From the MNIST data set, we are looking to classify the 10 possible numbers through a multi-layer network, also known as FCN.
- **Convolutional Multi-Layer Network (CNN):**
  - From the MNIST data set, we are looking to classify the 10 possible numbers through a convolutional multi-layer network, also known as CNN.

In this report, each model will be explained in their own part, explaining the possible issue encountered, the solutions provided, and the global thinking required to build those models.

MNIST Data Set

Loading the Data

This data set provides us 4 variables which works by pair. One is the training set containing 60 000 images with their associated labels. The second one is the test set with 10 000 pairs. This provides us 70 000 images with their labels. It can be loaded as the following:

```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Each image is made of 28*28 pixels for a total of 784 pixels. Those **images** are encoded as **numpy arrays** (or tensors) and the **labels** are simply an array of digits, ranging from "0" to "9".

Thanks to the ***matplotlib*** and ***numpy*** libraries, we can display the images horizontally and the labels corresponding.

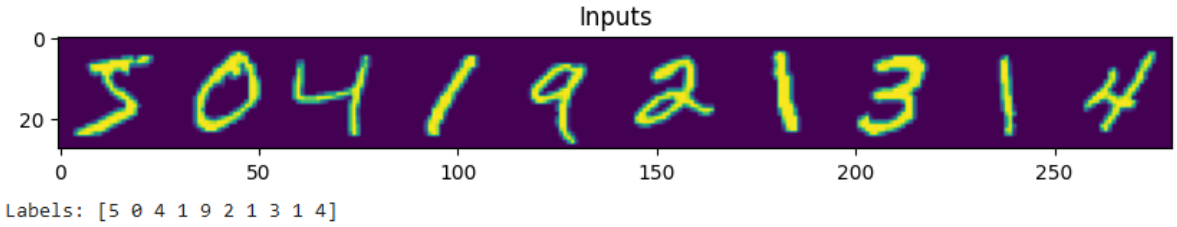


Figure 1: MNIST Data Set

We will train our models with the train set, excluding the test one. We will then evaluate the model by the accuracy of the comparison between the test set prediction and the test set labels.

We aim for a 100% accuracy, even if we keep in mind that the value is unreachable. Otherwise, it would mean, that the set that had been evaluated, had already been used in the training set of our model. It would be a disaster for it, as it will let us think that the model is working well when it's not necessarily the case.

Preprocessing the Data - FCN

Before training, we pre-process our data by reshaping it into the shape that the network expects, and scaling it so that all values are in the [0, 1] interval (by doing this, it will improve the computation time). By the way, this step is called the "**Standardisation**" or "**Normalisation**".

Moreover, our training images are stored in an array of shape (60000, 28, 28) of type uint8 with values in the [0, 255] interval. We already explained that each image contains 28 pixels in width and height explaining this shape (60 000 images of 28 pixels in width and 28 pixels in height). The last interval is matching the colour of the pixel as its value is given in RGB (or grayscale). We then decide to transform it into a float32 array of shape (60000, 28 * 28) with values between 0 and 1. Those lines provide a quick transformation:

```
# Transform the shapes of images of (X, 28, 28) to (X, 28 * 28)
train_images = train_images.reshape((60000, 28 * 28))
test_images = test_images.reshape((10000, 28 * 28))

# Normalize the values from the original values [0, 255] to [0, 1]
train_images = train_images.astype('float32') / 255
test_images = test_images.astype('float32') / 255
```

An other point is to parse our data and labels in the good format to allow our model. We do it once again as it is also required to make our model works:

```
# Convert numpy.ndarray of float32 to categorical
train_labels = to_categorical(train_labels)
test_labels  = to_categorical( test_labels)
```

Preprocessing the Data - CNN

As we did for the FCN, we pre-process our data by reshaping it into the shape that the network expects, and scaling it so that all values are in the [0, 1] interval. However, this time the input format required change as we use a convolutional model. As we are looking for an input with the shape (28 * 28 * 1). Those lines provide the required transformation:

```
# Transform the shapes of images of (X, 28, 28) to (X, 28, 28, 1)
train_images = train_images.reshape((60000, 28, 28, 1))
test_images  =  test_images.reshape((10000, 28, 28, 1))

# Normalize the values from the original values [0, 255] to [0, 1]
train_images = train_images.astype('float32') / 255
test_images  =  test_images.astype('float32') / 255
```

An other point is to parse our data and labels in the good format to allow our model to understand that the prediction is not numerical but categorical. This is ensure with the following code:

```
# Convert numpy.ndarray of float32 to categorical
train_labels = to_categorical(train_labels)
test_labels  = to_categorical( test_labels)
```

Classification by Multi-layers Network (FCN)

Introduction to the FCN

To build our network, we will put 2 "**dense**" hidden layers after the "**input**" layer. A dense layer is a fully connected one such as the following one :

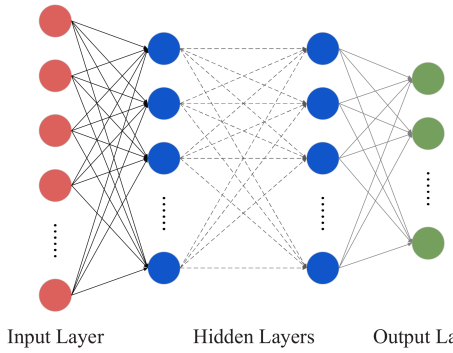


Figure 2: FCN Architecture. Source: [mdpi.com]

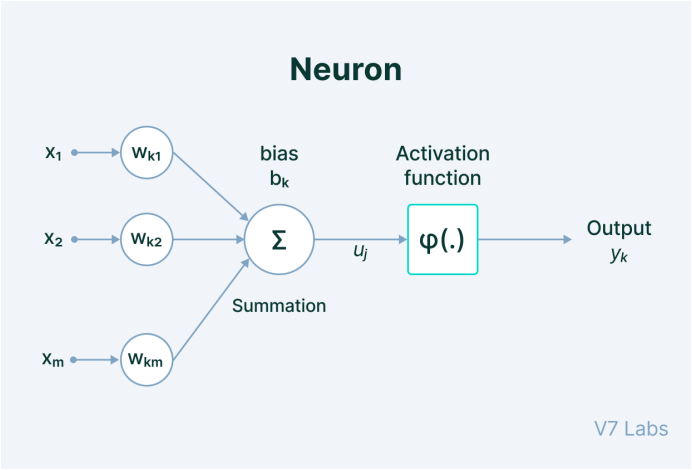


Figure 3: Neuron Architecture. Source: [v7labs.com]

The input layer of the neural network receives pixel values of the input image. Each neuron in this layer corresponds to a pixel in the image.

The first hidden layer is supposed to apply a "Rectified Linear Unit (**ReLU**)" which is an activation function. Its equation is the following one "$f(x) = max(x, 0)$". Then, we apply a second hidden layer which returns the output (label) using the "**softmax**" function useful for multi-class classification.

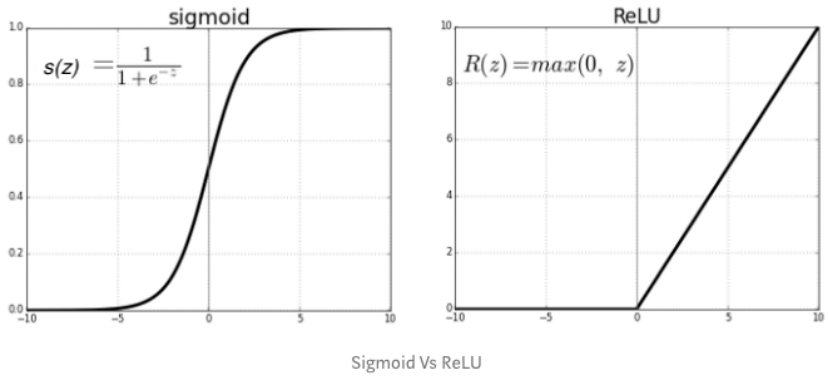


Figure 4: ReLU & Softmax function. Source: [Medium]

Creating our FCN

To realize our networks, we are likely to use the library "***keras***". Indeed, it provides several methods useful for the construction of neural networks, including dense layers, models and related parameters such as the activation function (e.g: ReLu).

The first step to build our model is to define it as a sequential one and then add each required layer. We will add one with a "**ReLu**" activation and then, one with a "Softmax" one. The first one will brings relevant features to build an efficient classifier neural network. Then, the second one is crucial for "multi-class" classification as the output returns the likelihood/confidence of the input belonging to each class. This model will then for each image passed in parameter, return an array of "10" probabilities where each value is the probability that the input belongs to the class "0", "1", ..., "9".

To build our model, we will write the following line of codes. The first parameter for the layers.Dense(), is the number of neurons in the dense layer. With an empirical approach, we define the best value for it. We need to be careful when defining this value as a number below the best one could cause under-fitting (or over-fitting if the value is too much above the best one). In the second layer, "10" is corresponding to the number of outputs possible. In a binary problem, it would be "1" as it will return "0" or "1" on a single output. Nevertheless, as we are looking for the probability of "10" numbers, we are looking for "10" outputs.

```
# Create the network
network = models.Sequential()

# Add a dense layer with "ReLU" activation
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
```

```
# Add a dense layer with "softmax" activation
network.add(layers.Dense(10, activation='softmax'))
```

With the network architecture defined, the parameters of all layers are filled with random values. From now on, we need to gradually adjust these parameters, based on the available data. That's where the "**training_set**" will be useful. To adjust the network we need two more characteristics:

- ***Loss function***: to improve the neural network, we need to measure the difference between the output and the expected one through the "loss function". As we are applying for a multi-class classification, we will go for a "**categorical cross-entropy**".

- ***Optimizer***: to adjust the weights, it will help to use this score as a feedback. It will, in a way, lower the loss score. It implements the **Back-propagation** algorithm.

```
# Adjust the network with an optimizer and a loss function
network.compile(optimizer='adam', loss='categorical_crossentropy',
    metrics=['accuracy'])
```

Evaluating our FCN

As we have built the multi-layers network with the hidden layers and the input layer, we can now train it on the "***train_images***" and "***train_labels***". The "***epochs***" are the number of iteration where the model will be trained, and the "***batch_size***" is the number of sample from the "***train_set***". It means that for "**10 epochs**" and a value of "**1 000 batch_size**", the training set will be split into 1 000 samples (60 images per sample in our base case) and train 10 times. To reduce over-fitting the, training set will be split according to 70/30 split. Indeed, we assume this ratio has a good trade-off to let the model learn and keep enough inputs to test it on a decent variety of inputs. To retrieve an understandable plot to understand the evolution of the model performance, we provide the code below:

```
# Retrieve information on the built network
history = network.fit(train_images, train_labels, epochs=10, batch_size=1000,
    validation_split=0.3)

# Create the graph cells and the size of it
f, (ax1, ax2) = plt.subplots(1,2, figsize=(10,4))

# Define the training and validation loss according to the epochs
ax1.plot(history.history['loss'], label='Training Loss')
ax1.plot(history.history['val_loss'], label='Validation Loss')
ax1.set_title('Validation and Training loss')
ax1.set_xlabel('Epochs')
ax1.legend()

# Define the training accurarcy according to the epochs
ax2.plot(history.history['accuracy'], label='Training Accuracy')
ax2.plot(history.history['val_accuracy'], label='Validation Accuracy')
ax2.set_title('Validation and Training accuracy')
ax2.set_xlabel('Epochs')
ax2.legend()
```

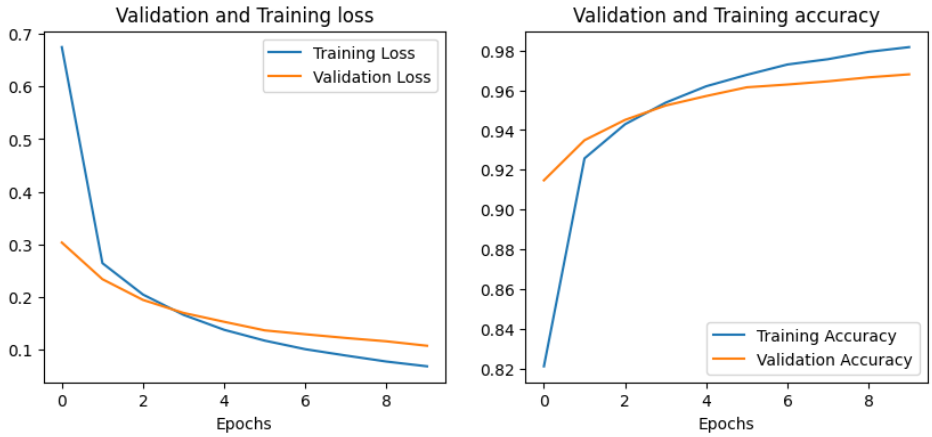


Figure 5: Training and Validation loss and accuracy performance for 10 epochs

Figure 6: Training and Validation loss and accuracy performance for 20 epochs

As we can see, across the epochs, our models look to converge. However, it is because the parameters are filled to keep a good trade-off between performance and over-fitting. The over-fitting describes a model too much trained on the training set and losing a general accuracy for a specific one (focusing on too much specificity and not fitting a general case). As we can see for more epochs, the validation accuracy is not improving that much after the 10th epochs while the training is. We can conclude that the 10th one is the frontier that when crossed lead to over-fitting. As we want to avoid it, we cap the value of the epochs to 10.

Testing our FCN

As we have created and evaluated the performance of our model, we can now try to predict our test set. We are looking for good results as our validation accuracy is hitting "**0.95**" or more. It is already a very good result. To realize our prediction, we compile the following code. It also provides us some example of the input and the array returned:

```
# Realise the prediction with our "network" and the "test_images"
predictions = network.predict(test_images)
```

```
# Define the value to begin a ten values interval
n = 0

# Retrieve the real format of the input and the matching output
inputs  = test_images[n*batch_size : (n+1)*batch_size,:].reshape(
    batch_size, 28, 28)
outputs = predictions[n*batch_size : (n+1)*batch_size,:]

# Create the graph cells and the size of it
f, ax = plt.subplots(batch_size, 2, figsize=(10, 4 * batch_size))

# For the number of sample, plot the input and the graph of the prediction
for i in range(batch_size):
    ax[i,0].imshow(inputs[i,:,:])
    ax[i,0].set_title("Digit")
    ax[i,1].bar(range(10), outputs[i,:])
    ax[i,1].set_title("Prediction (probabilities)")
    ax[i,1].set_xticks(range(10))
    ax[i,1].set_ylim((0,1.1))

# Retrieve the test loss and accurary
test_loss, test_acc = network.evaluate(test_images, test_labels)
```

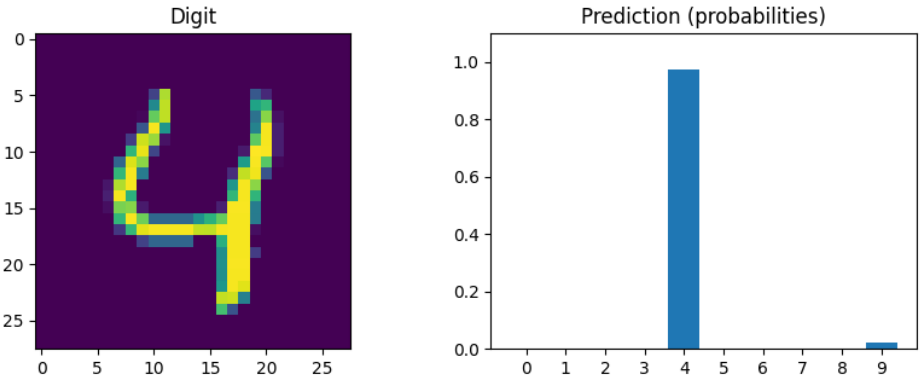


Figure 7: Example of prediction with input and output

The validation accuracy was "**0.975**" and the loss was "**0.877**". For our testing set, we reach respectively "**0.979**" and "**0.072**". This are very good results as our accuracy improved and stands between the validation and training one. This can be explained by enough neurons per layer avoiding under-fitting and allowing enough units to learn the main features, but also avoid the over-fitting by adding too much units per layers. Also, the parameters such as the loss function, the optimizer, the activation function and so on... have been chosen greatly to classify our inputs.

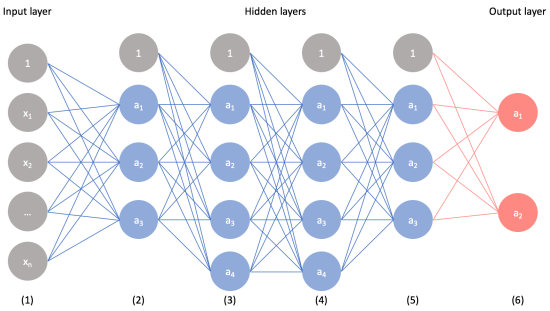
Convolutional Neural Network (CNN)

Introduction to the CNN

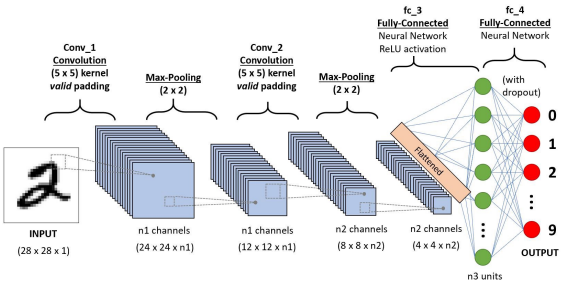
To build our convolutional neural network, we will modify our "**FCN**" one and implement a new technique that will surely improve our model, preventing over-fitting.

ConvNets take advantage of the fact that the input consists of images, and they constrain the architecture in a more sensible way. To do so, they make use of convolutional layers. The fundamental difference is this: fully-connected layers learn global patterns in the inputs, whereas convolution layers learn local patterns in small 2D windows of the inputs. More specifically, a convolution layer operates over 3D tensors with two spatial axes and a depth axis (height, width, channels). The convolution operation extracts patches from its input, and applies the same transformation to all of these patches. The output is still a 3D tensor, but its dimensions depend on the layer's hyper-parameters, specified by the kernel size and the number of kernels.

ConvNets mainly use three types of layers: convolutional (CONV), pooling (POOL), fully-connected (FC). The figure below shows a concrete example of ConvNet architecture. The first layer (left) stores the raw image pixels, whereas and the last layer (right) stores the class probabilities. The activation of each hidden layer along the processing path is shown as a column.



(a) CNN Architecture. Source: [jeremyjordan.me]



(b) Convolutional Network Architecture. Source: [analyticsvidhya.com]

Figure 8: CNN full architecture

Creating our CNN

In the same way, we first define the structure of the model and then add some layers in it.

```
# We build our CNN as Layer - Pooling - Layer - Pooling - Layer - Flatten -
# Neural Network as we already dit
model_cnn = models.Sequential()

# First couple layer/pooling with 32 neurons (our images (28*28 pixels) will
# fit the format (28, 28, channel (= 1 for the input layer)))
model_cnn.add(layers.Conv2D(32, (3, 3), activation='relu',
    input_shape=(28, 28, 1)))
model_cnn.add(layers.MaxPooling2D((2, 2)))

# Second couple layer/pooling with 64 neurons
model_cnn.add(layers.Conv2D(64, (3, 3), activation='relu'))
model_cnn.add(layers.MaxPooling2D((2, 2)))

# Last couple so layer then flatten it to send it in a fully connected neural
# network
model_cnn.add(layers.Conv2D(64, (3, 3), activation='relu'))
model_cnn.add(layers.Flatten())

# Fully connected neural network that returns an array of 10 probabilities
model_cnn.add(layers.Dense(64, activation='relu'))
model_cnn.add(layers.Dense(10, activation='softmax'))
```

We repeat the same process for the "***optimizer***" and the "***loss function***":

```
# Define the optimizer and the loss function
model_cnn.compile(optimizer='adam', loss='categorical_crossentropy',
    metrics=['accuracy'])
```

Evaluating our CNN

For the evaluation, the process is the same, with the same reason:

```
# Retrieve information on the built network
history_cnn = model_cnn.fit(train_images, train_labels, epochs = 5, batch_size = 64,
    validation_split = 0.3)

# Create the graph cells and the size of it
f, (ax1, ax2) = plt.subplots(1,2, figsize=(10,4))

# Define the training and validation loss according to the epochs
ax1.plot(history.history['loss'], label='Training Loss')
ax1.plot(history.history['val_loss'], label='Validation Loss')
ax1.set_title('Validation and Training loss')
ax1.set_xlabel('Epochs')
ax1.legend()

# Define the training accurarcy according to the epochs
ax2.plot(history.history['accuracy'], label='Training Accuracy')
ax2.plot(history.history['val_accuracy'], label='Validation Accuracy')
ax2.set_title('Validation and Training accuracy')
```

```
ax2.set_xlabel('Epochs')
ax2.legend()
```

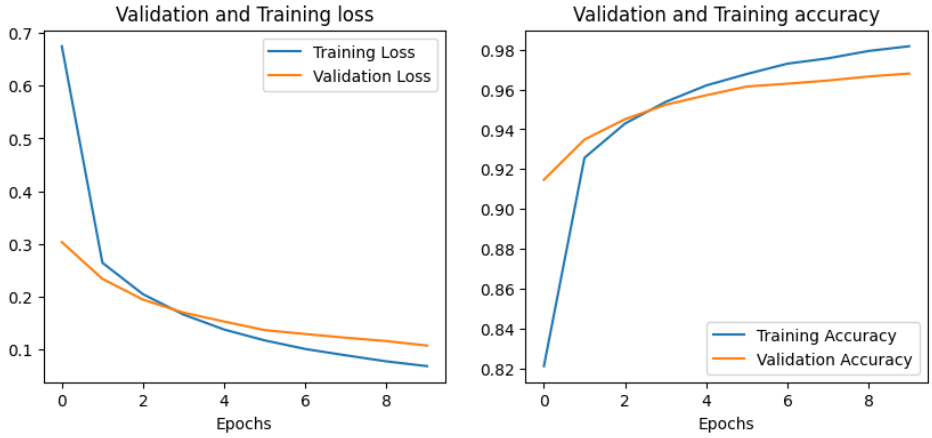


Figure 9: Training and Validation loss and accuracy performance for 5 epochs

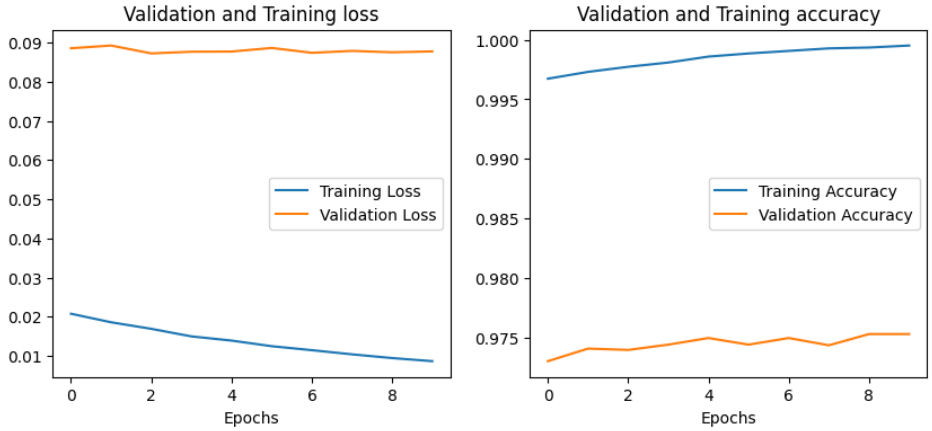


Figure 10: Training and Validation loss and accuracy performance for 10 epochs

As we can see, across the epochs, our models converge once again.We observe, for more epochs, the validation accuracy is not improving that much after the 5th epochs while the training is. We can conclude that the 5th one is the frontier that when crossed lead to over-fitting. As we want to avoid it, we cap the value of the epochs to 5. In the terminal outputs, we can see that the val_acc of the 5th epoch is "0.9881" while the 6th one is "0.9869".

Testing our CNN

We can finally test our model on the test set and see if the accuracy and loss are better for this new model!

```
# Realise the prediction with our "network" and the "test_images"
predictions_cnn = model_cnn.predict(test_images)
```

```
# Retrieve the test loss and accurary
test_loss, test_acc = model_cnn.evaluate(test_images, test_labels)
```

The validation accuracy was "**0.9848**" and the loss was "**0.0498**". For our testing set, we reach respectively "**0.988**" and "**0.036**". This are very good results. We have to note that both values are better than the FCN and the global accuracy is really good. We removed some over-fitting through the pooling realized after some dense layers and we reached a better model.