BODIN Amaury

BERNARD Vincent

Report on digit recognition :

Table des matières

1 Introduction

This is a digit recognition problem, we will have to build some neural networks that can classify between the digits. In this project, we will use the function mnist.load_data() to import 70,000 grayscale images of handwritten numbers from 0 to 9.

Each image has a size of 28 * 28 pixels. These images are stored in Keras, so when we download the images, we put them into 2 tuples for training (X_train, y_train) and testing (X_test, y_test).

X_train and X_test consist of images of handwritten digits from 0 to 9. They are in the shape of a multidimensional array in which each element represents the value of a pixel in the image. On the other hand, y_train and y_test are the labels of the corresponding images. That means, this is the real digit in the image, in a one-dimensional array.

In this project, we will build 2 different types of neural networks, one with Dense layers and another with convolutional layers. Then we will observe the performance of those algorithms during training and prediction.

2 Fully connected neural networks

2.1 The neural network

First of all, we download the training and testing data as follows:

```python
# un dataset de chiffres pré-loadé sur keras
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

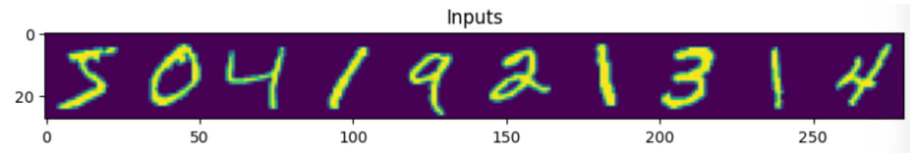
The images are encoded in a NumPy array called a tensor, and the labels are simply an array with digits. The models will learn on the training data and then be tested on the test data.

So, we check the shape of the train_images because we need it in order to create the right network.

```python
train_images.shape
```

```
(60000, 28, 28)
```

So, we have three pieces of information: the first one is that there are 60,000 images of size 28 pixels in terms of height and 28 pixels in terms of width. We want to see how the digits look, so we show a batch of 10 digits and their labels:



```
Labels: [5 0 4 1 9 2 1 3 1 4]
```

As we can see, the digits can have extremely different forms. For example, the two "four" digits and the ones are very different.

Now, it is time for us to create our first neural network. This is a fully connected neural network.

```python
network = models.Sequential() # un réseau séquentiel
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,))) # on
network.add(layers.Dense(64, activation='relu')) # ces lignes permettent de cré
network.add(layers.Dense(32, activation='relu'))
network.add(layers.Dense(16, activation='relu'))
network.add(layers.Dense(10, activation='softmax')) # cette ligne permet d'avoi
```

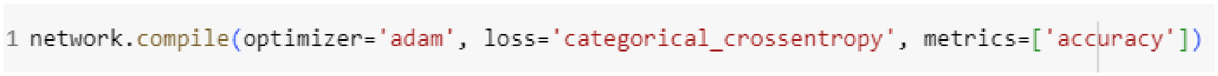
On the first line, we define a sequential model. Then we choose a Dense layer with a number of hidden layers and a ReLU activation to introduce non-linearity so the calculations will be easier, as this is an easy computation to perform.

We tried several numbers of layers and hidden layers, but we decided to keep this one because we obtained better performance. At the end, we choose 10 and a softmax activation. Ten because we have 10 possibilities; the numbers are from 0 to 9. The softmax activation is used to obtain a probability between 0 and 1 (the probability to obtain each number), and the sum of those probabilities is equal to 1.

Now, the neural network is created, so we are ready to move on to the next step.

2.2 The compilation

Compiling a model is an important step in which we can configure some important parameters for the learning process.



```
network.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

We select an optimizer. An optimizer is an algorithm used to adjust the weights of the neural network during training. We use it to minimize the loss function. We chose the Adam optimizer because it is an accelerated version of Stochastic Gradient Descent. This algorithm initializes the weights of the model randomly. And at each epoch (iteration during training), it selects a batch of data randomly for training. It updates the weights in a way that minimizes the loss function (the difference between the true value and the prediction).

We also select the loss function (the difference between the true value and the prediction) in order to minimize this loss. We choose categorical cross-entropy because it is suitable for multi-class classification problems such as ours.

Finally, we select a metric. A metric is used to evaluate the performance of the model during training. In our case, we choose accuracy, which is the number of "correct" predictions divided by the total number of predictions.

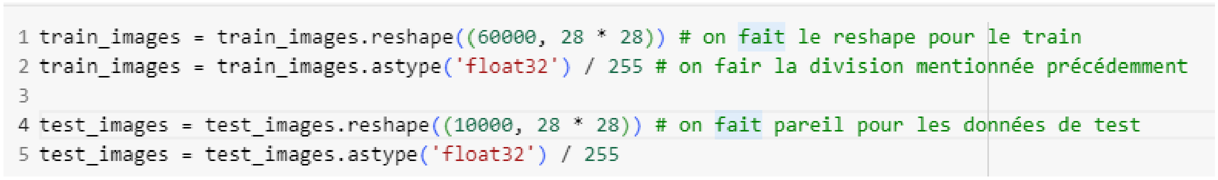
Now, we are almost ready to train our model. Let's meet again in the next part.

2.3 Reshape images and labels

Before even trying to train our model we have to make sure to have the right shape of data for this model. If this is not the case we have to reshape them.

As we saw at the beginning of this report, we had this shape (60 000, 28,28) more over our “values” are between 0 and 255 because the images are levels of gray. So we want them to be between 0 and 1 so we will have to divide them by 255 and we also want our data to be this way : (60 000, 28*28).

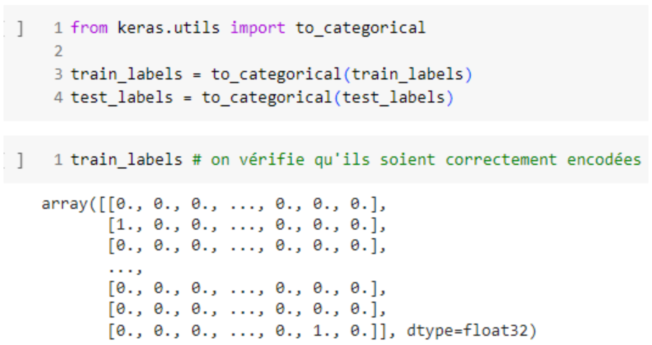
So we will have to reshape our data and perform the division mentioned :



```python
train_images = train_images.reshape((60000, 28 * 28)) # on fait le reshape pour le train
train_images = train_images.astype('float32') / 255 # on fair la division mentionnée précédemment

test_images = test_images.reshape((10000, 28 * 28)) # on fait pareil pour les données de test
test_images = test_images.astype('float32') / 255
```

Furthermore, we also have to encode our labels. Indeed, for multiclass classifications we need them to be one-Hot Encoded :



```python
]   from keras.utils import to_categorical

    train_labels = to_categorical(train_labels)
    test_labels = to_categorical(test_labels)
```

```python
]   train_labels # on vérifie qu'ils soient correctement encodées
```

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 1., 0.]], dtype=float32)
```

2.4 Training of the neural network

We are finally ready to train our model, for that we will use the function “.fit()” that we have in keras. We are free to select the number of batch sizes and epoch that we want.

The batch size and the number of epochs are 2 hyperparameters. The batch size is the number of training examples that are used during one iteration, we chose 500 but it could have been more or less. This means that during each training iteration the weights will be updated using those 500 training examples.

The number of epochs indicates the number of times that the entire training dataset is passed through the model. So, here we choose 10.

Our goal is to increase at the maximum the accuracy and reduce to the minimum the loss. For that, we have to “find” the best combination of epoch and batch size.

Here is the training :

```python
history = network.fit(train_images, train_labels, epochs=10, batch_size=500)
```

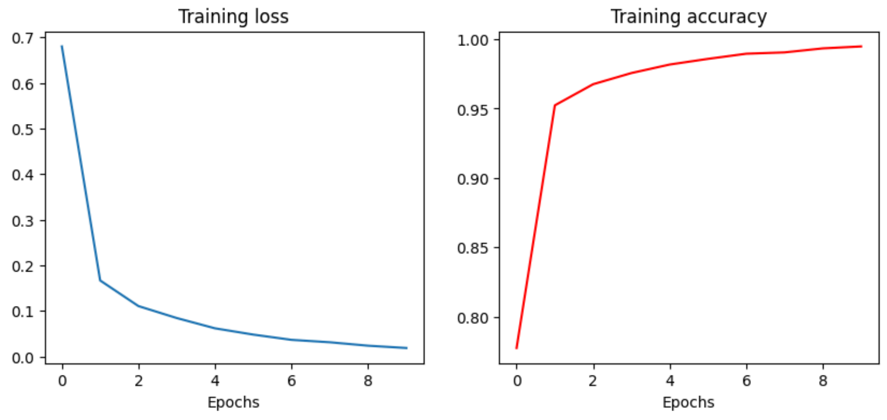
2.5 Learning performance

Here are the results during the training :

```
Epoch 1/10
120/120 [==============================] - 5s 25ms/step - loss: 0.6799 - accuracy: 0.7775
Epoch 2/10
120/120 [==============================] - 3s 22ms/step - loss: 0.1667 - accuracy: 0.9524
Epoch 3/10
120/120 [==============================] - 3s 22ms/step - loss: 0.1105 - accuracy: 0.9676
Epoch 4/10
120/120 [==============================] - 4s 33ms/step - loss: 0.0843 - accuracy: 0.9757
Epoch 5/10
120/120 [==============================] - 3s 23ms/step - loss: 0.0617 - accuracy: 0.9818
Epoch 6/10
120/120 [==============================] - 3s 23ms/step - loss: 0.0480 - accuracy: 0.9859
Epoch 7/10
120/120 [==============================] - 4s 30ms/step - loss: 0.0365 - accuracy: 0.9896
Epoch 8/10
120/120 [==============================] - 5s 41ms/step - loss: 0.0312 - accuracy: 0.9905
Epoch 9/10
120/120 [==============================] - 3s 28ms/step - loss: 0.0236 - accuracy: 0.9934
Epoch 10/10
120/120 [==============================] - 3s 26ms/step - loss: 0.0186 - accuracy: 0.9948
```

As we can see, during the process of learning we can visualize the accuracy and the loss of the model. We observe that the accuracy fastly converges to 100% and the loss to 0 (without reaching it obviously).

Let’s plot those the loss and accuracy :



We see in the previous graphs, the loss is described progressively over the epochs while the precision increases. This is indeed normal because as training progresses the model improves. However, we must be careful about overfitting.

2.6 Prediction performance

Once we have analyzed the performance during the training this is important to make sure that the performance of the accuracy is the same on the test data. So we perform the prediction and hope that we will obtain good results.

```
predictions = network.predict(test_images)
```

```
13/313 [==============================] - 2s
```

As you will see below, the prediction is made on each class. Obviously, we keep the one with the highest probability. For example, if I have a probability of 80% to obtain the number 5 so the rest of the probability will be lower so we consider that it will be the number 5.

Here is an example on how it looks like :

```
predictions[0]
```

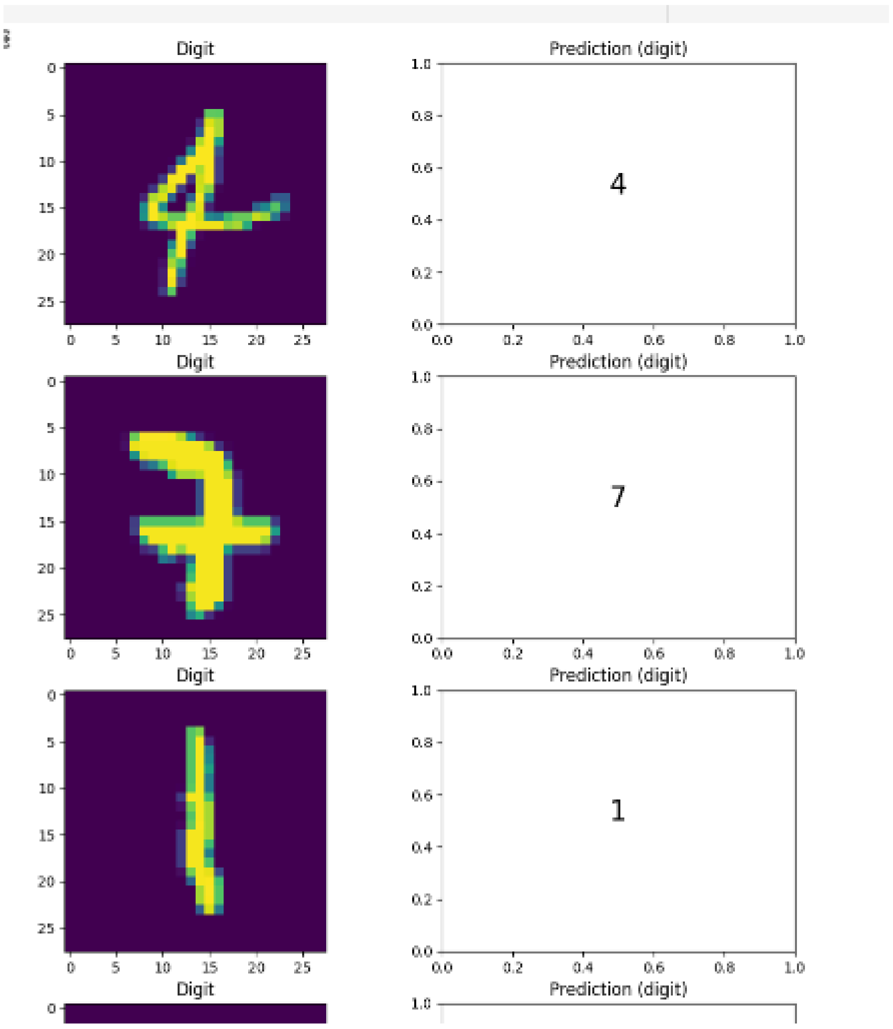
```
array([1.7346186e-06, 7.2427065e-07, 7.7200777e-05, 8.1140381e-05,
       3.1915764e-07, 4.5135931e-07, 2.0246911e-10, 9.9972427e-01,
       2.0642960e-06, 1.1205630e-04], dtype=float32)
```

```
np.argmax(predictions[0])
```

```
7
```

So here we predict a seven because it has the highest probability.

Bellow, we decide to predict an entire batch (not all of them in this picture) and to visualize the numbers and the predictions :



As we know the real "values" we can compare our predictions with the y_test.

For that, we use the function evaluate and we plot the loss and the accuracy on the test data.

```python
loss, acc = network.evaluate(test_images, test_labels)

print("loss : ", loss)
print("acc : ", acc)
```

```
313/313 [==============================] - 2s 7ms/step - loss: 0.0720 - accuracy: 0.9785
loss :  0.07203495502471924
acc :  0.9785000085830688
```

As we can see now, we have a lower accuracy than expected during the training process and we also have a higher loss which is not so good. Fortunately, this result is satisfying regarding the number of data and the accuracy ~0.98.

3 Multi-layer model with convolution

The process is almost the same in this part so we will not reexplain what was already explained before.

So, once again we will use the MNIST digits from KERAS.

We will also have to reshape the data because this network does not take into account the same shapes for some reasons that will be explained later.

3.1 The neural network

The main difference between a fully-connected network (Dense layers) and the ConvNet is the fact that for Dense layers, they learn global patterns in 3D while convolutional networks learn local patterns in 2D.

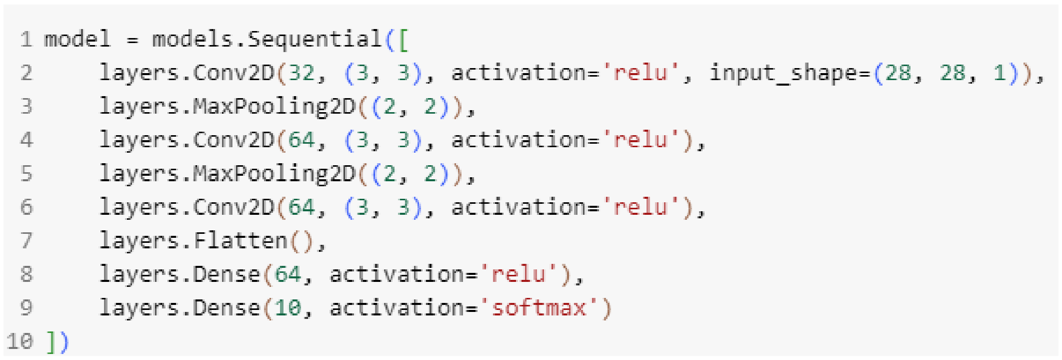
So here, we create a CNN model, with only 3 layers accompanied by maxpooling, which allows us to reduce the dimensionality. And then we put 2 Dense layers anyway because we need them for classification.

The output of a Conv2D and MaxPooling2D is a 3D tensor. This is why we have to flatten before using the Dense layers to get back to a vector 1D. We chose to put a Dense layer of 64 hidden layers and then a layer with 10 to predict the output (digits from 0 to 9 => 10 classes) using softmax activation, for the same reason as before, to obtain a prediction between 0 and 1 for those classes.

The first parameter of a layer.Conv2D is the number of filters. If the number of filters is extremely high the model will have more ability to extract information from the image, the problem is that it increases the number of parameters and leads to a higher risk of overfitting.

The second parameter defines the size of the core of convolution, this is a “tool” that is used to extract information locally.

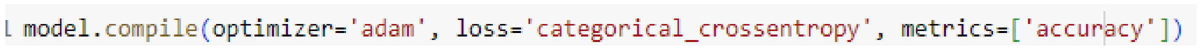
The layer.Maxpooling2D is used to define the size of the “tool” mentioned before and its “moving” step. The pooling is used to downsample which reduces the dimensionality of an image but keeping the essential part of the information. We also use ReLu activation to bring non-linearity in the model.



```python
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

3.2 The compilation

Once again we have to compile our model using the loss “categorical cross-entropy” because this is a multi-class problem. We also use the metric “accuracy” and the Adam optimizer as before.



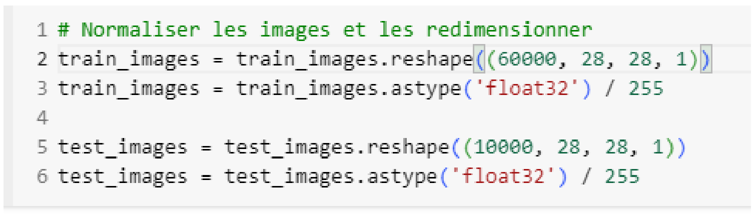
```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

3.3 Reshape images and labels

As we did during the first part, we will once again normalize the data and reshape them. We have to normalize the data in order to make sure that they are in a range from 0 to1, which makes the data more easily comparable and easier to process. So we have to divide them by 255 because the images are grayscale with 0 meaning black and 255 meaning white.

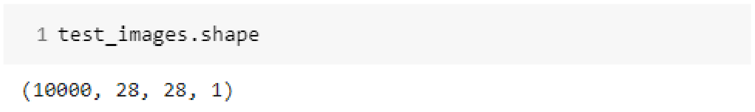
The convolutional network is expecting to receive a certain shape of data under the form of tensors with the following dimensions : width, heights , "color".

As we saw before, the images are 28 * 28 and the color is 1 because no RGB but greyscale. So we can reshape them to : (60 000, 28,28,1) with 60 000 the number of images.



```
# Normaliser les images et les redimensionner
train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255
```

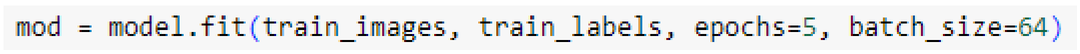


```
test_images.shape
```

```
(10000, 28, 28, 1)
```

3.4 Training of the neural network

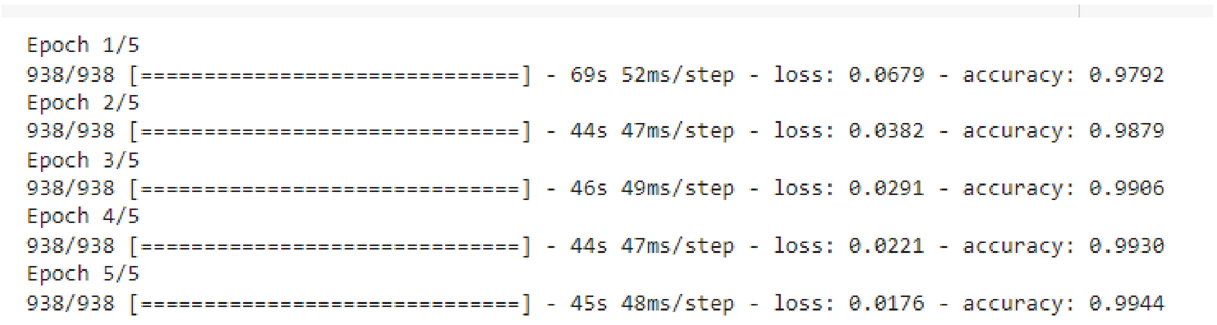
Now, we have made everything right so we can fit the model.



```
mod = model.fit(train_images, train_labels, epochs=5, batch_size=64)
```

3.5 Learning performance

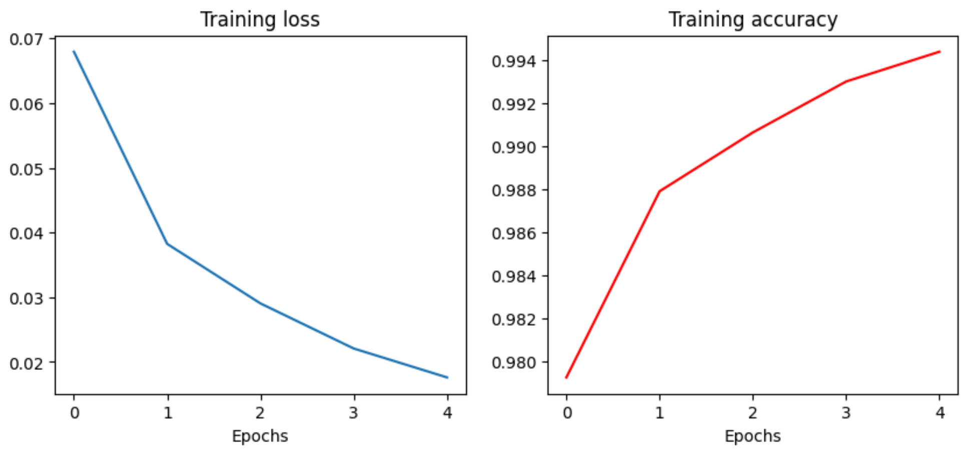
Here is the result of the performance during the training process :



```
Epoch 1/5
938/938 [==============================] - 69s 52ms/step - loss: 0.0679 - accuracy: 0.9792
Epoch 2/5
938/938 [==============================] - 44s 47ms/step - loss: 0.0382 - accuracy: 0.9879
Epoch 3/5
938/938 [==============================] - 46s 49ms/step - loss: 0.0291 - accuracy: 0.9906
Epoch 4/5
938/938 [==============================] - 44s 47ms/step - loss: 0.0221 - accuracy: 0.9930
Epoch 5/5
938/938 [==============================] - 45s 48ms/step - loss: 0.0176 - accuracy: 0.9944
```

As we can see, the accuracy is very high at the end of the first epoch but it can't really improve during the next epochs.

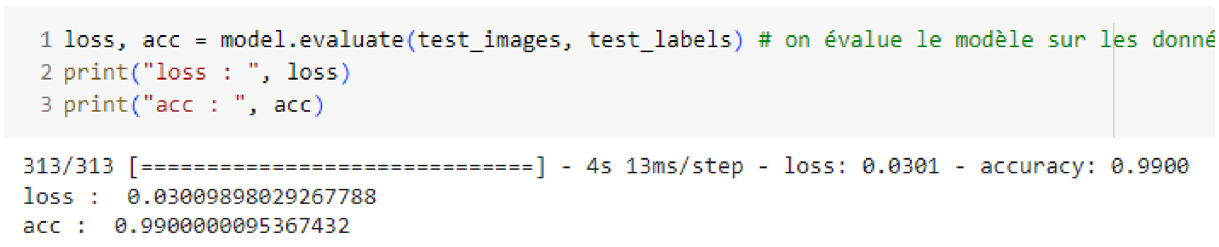
We plot those results to visualize them :



As we can observe, we obtained a final loss 2 times lower than the previous model and an accuracy a little bit higher.

3.6 Prediction performance

We perform the prediction on the test data with the function evaluate and print les accuracy and the loss :



```
loss, acc = model.evaluate(test_images, test_labels) # on évalue le modèle sur les donné
print("loss : ", loss)
print("acc : ", acc)
```

```
313/313 [==============================] - 4s 13ms/step - loss: 0.0301 - accuracy: 0.9900
loss :  0.03009898029267788
acc :  0.9900000095367432
```

As we can see we obtained better results than during the first part of this project.

4 Conclusion

In this project, we faced the digit recognition problem using two approaches: a fully connected neural network and a convolutional layer model.

Firstly, we built a fully connected neural network with dense layers and softmax activation for classification. We compiled the model with Adam optimizer, categorical cross-entropy loss, and accuracy metric. After reshaping the data, we trained the model and evaluated its performance. While achieving high accuracy on the training data, we observed slight overfitting on the test data.

Next, we constructed a convolutional layer model to learn local patterns in the images. This model comprised convolutional and max-pooling layers followed by dense layers for classification. We compiled and trained the model similarly, noting slightly improved performance compared to the fully connected neural network.

In conclusion, both approaches yielded satisfactory results for digit recognition, with the convolutional layer model showing a slight improvement over the fully connected neural network. Further refinement and exploration of techniques could enhance performance even more.