Computer Vision and Deep Learning: Lab 02

“Handwritten Digit Classification using ANN and CNN”

Valentin Six

Exercise 1:

Neurons	8	16	32	64	128
Accuracy	92,42 %	94,26 %	95,98 %	97,77 %	98,68 %

Table 1: System performance evaluation for various numbers of neurons on the hidden layer

We can note that the neurons are in the hidden layer, the more accurate our model seems to be. This is quite intuitive: the more complex a model is, the more detailed it is going to be. But accuracy comes at a cost: the more neurons are on the hidden layer, the more time the model will need to converge.

Exercise 2:

Batch size	32	64	128	256	512
Accuracy	93,12 %	92,87 %	92,59 %	92,21 %	91,51 %

Table 2: System performance evaluation for various values of the batch size, for 8 neurons on the hidden layer

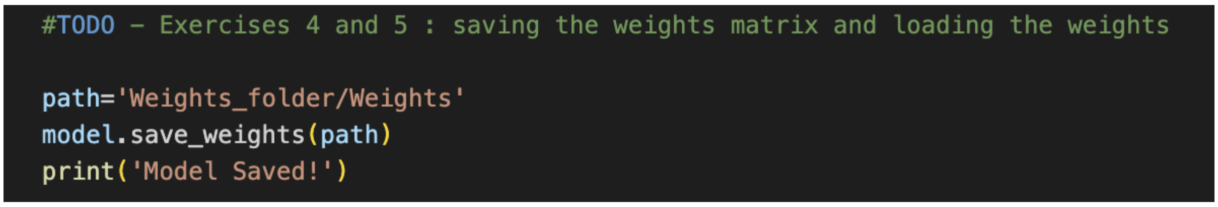
It seems that the bigger batch size we use, the less accurate our model is.

Exercise 3:

When using the mean squared error as the system performance metric, the final loss value (at the end of training) for the training set and validation set is almost equal to the loss value we obtained for the accuracy metric: in both cases the final loss value at the end of training was around 0,27). So, we could say that changing the system performance metric does not change the training process: in the first case the model tries to maximize the accuracy, in the second case the model tries to minimize the MSE, but the process is the same.

Exercise 4:

The following code is used to save the weights:

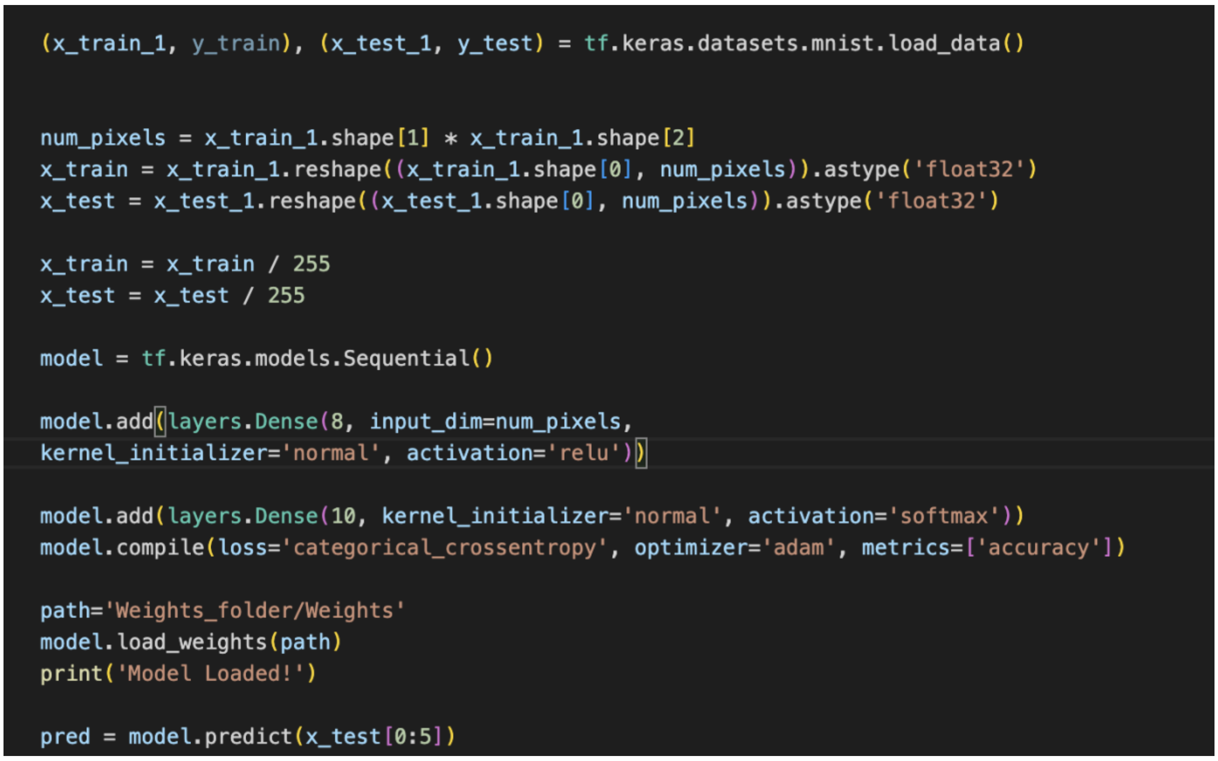


```
#TODO - Exercises 4 and 5 : saving the weights matrix and loading the weights

path='Weights_folder/Weights'
model.save_weights(path)
print('Model Saved!')
```

Figure 1: Code for saving the weights after the training

Exercise 5:

Then we rebuild a model from scratch (using the same architecture as the previous one) and load the weights that were saved in the previous step. After building our model we try to predict the label for the first 5 images in the dataset:



```
(x_train_1, y_train), (x_test_1, y_test) = tf.keras.datasets.mnist.load_data()


num_pixels = x_train_1.shape[1] * x_train_1.shape[2]
x_train = x_train_1.reshape((x_train_1.shape[0], num_pixels)).astype('float32')
x_test = x_test_1.reshape((x_test_1.shape[0], num_pixels)).astype('float32')

x_train = x_train / 255
x_test = x_test / 255

model = tf.keras.models.Sequential()

model.add(layers.Dense(8, input_dim=num_pixels,
kernel_initializer='normal', activation='relu'))

model.add(layers.Dense(10, kernel_initializer='normal', activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

path='Weights_folder/Weights'
model.load_weights(path)
print('Model Loaded!')

pred = model.predict(x_test[0:5])
```

Figure 2: Code for loading the weights and making predictions

We can display the first five images and associate with them the predicted label: we can see that our model predicts the digits correctly for the first 5 images!

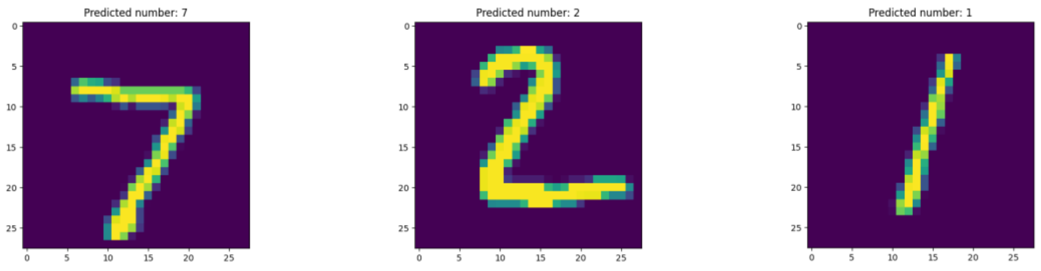


Figure 3: Some displayed images and the predicted label

Exercise 6:

After having built our CCN model, we can test the accuracy of our model with multiple values of filter sizes:

Filter size	1x1	3x3	5x5	7x7	9x9
Accuracy	96.02 %	98.26 %	98.47 %	98.82 %	98.99 %

Table 3: System performance evaluation for various values of the filter size

Exercise 7:

We can also look at the impact of changing the number of neurons in the hidden layer:

Neurons	16	64	128	256	512
Accuracy	96.91 %	97.49 %	98.06 %	98.42 %	98.64 %

Table 4: System performance evaluation for various values of neurons in the hidden layer

Exercise 8:

Finally, we can conjecture that the number of epochs in the system will have an impact on the accuracy of the system. We can check that in the following table:

Epochs	1	2	5	10	20
Accuracy	95.11 %	96.91 %	98.19 %	98.64 %	98.68 %

Table 5: System performance evaluation for various numbers of epochs

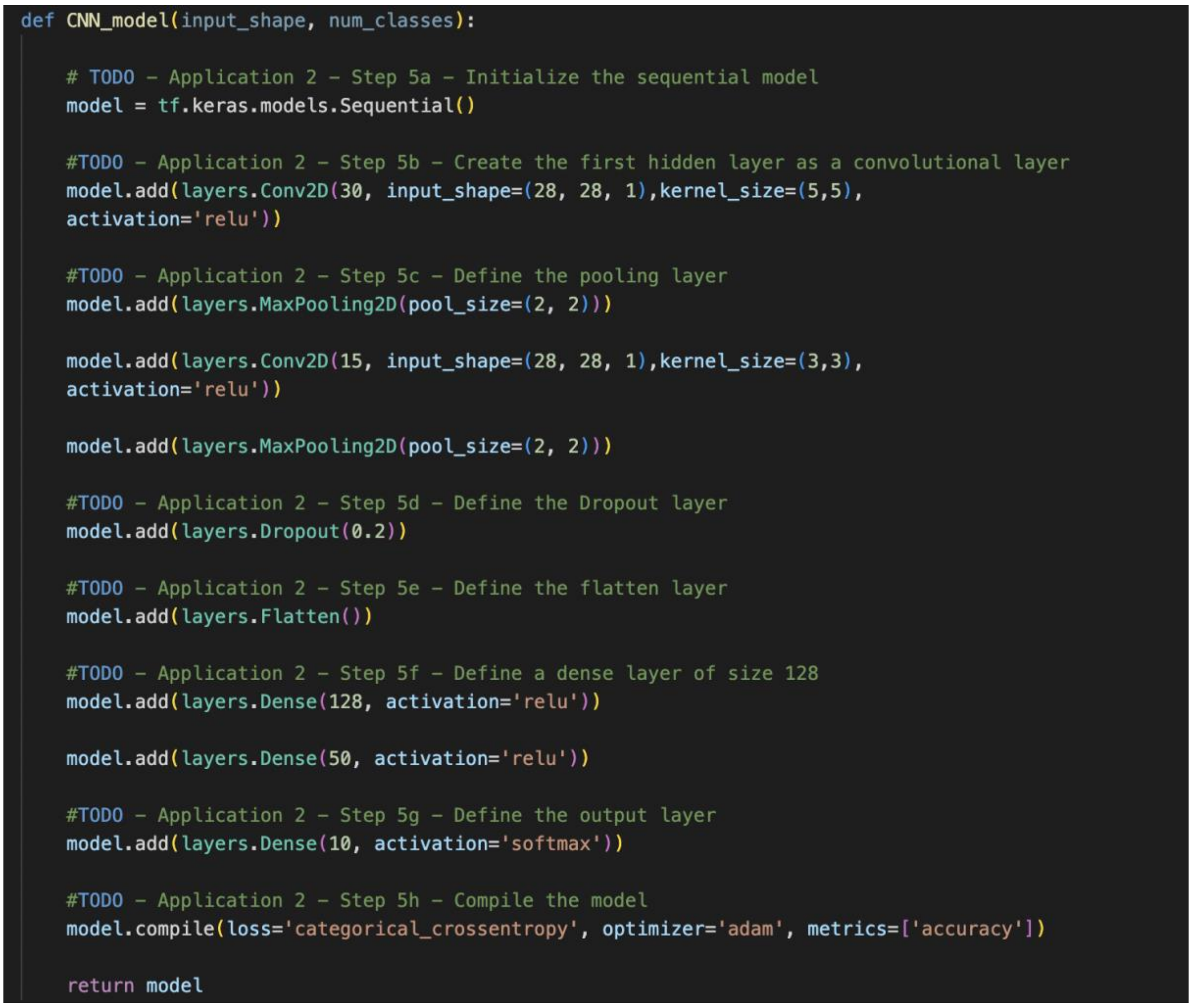
We can note that the training time is directly proportional to the number of epochs we implement: each epoch is around 4 seconds, so the total training time is 4n seconds, n being the number epochs we use in our system.

The system accuracy will usually grow with the number of epochs, but if we use many epochs, the model will tend to "overfit" on the training data (we can see with n=20 that after 17 epochs, the accuracy on the training set continues to grow, but the accuracy on the validation set decreases a little bit).

Exercise 9:

In this part we will add a few more layers (Conv2D, MaxPooling, Dense, Dropout, Flatten…) to our model, in hope to achieve a better accuracy.

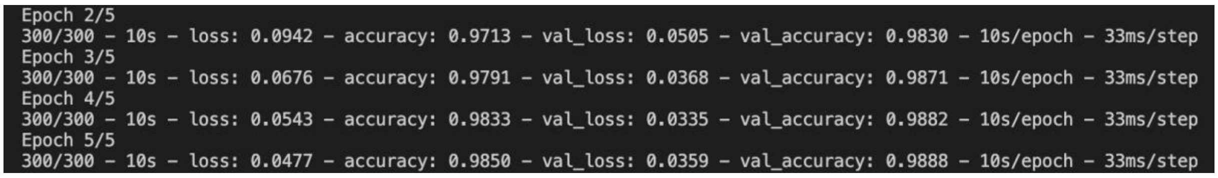
After the modifications, our model looks like this:



```
def CNN_model(input_shape, num_classes):

    # TODO - Application 2 - Step 5a - Initialize the sequential model
    model = tf.keras.models.Sequential()

    #TODO - Application 2 - Step 5b - Create the first hidden layer as a convolutional layer
    model.add(layers.Conv2D(30, input_shape=(28, 28, 1),kernel_size=(5,5),
    activation='relu'))

    #TODO - Application 2 - Step 5c - Define the pooling layer
    model.add(layers.MaxPooling2D(pool_size=(2, 2)))

    model.add(layers.Conv2D(15, input_shape=(28, 28, 1),kernel_size=(3,3),
    activation='relu'))

    model.add(layers.MaxPooling2D(pool_size=(2, 2)))

    #TODO - Application 2 - Step 5d - Define the Dropout layer
    model.add(layers.Dropout(0.2))

    #TODO - Application 2 - Step 5e - Define the flatten layer
    model.add(layers.Flatten())

    #TODO - Application 2 - Step 5f - Define a dense layer of size 128
    model.add(layers.Dense(128, activation='relu'))

    model.add(layers.Dense(50, activation='relu'))

    #TODO - Application 2 - Step 5g - Define the output layer
    model.add(layers.Dense(10, activation='softmax'))

    #TODO - Application 2 - Step 5h - Compile the model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

    return model
```

Figure 4: Code for building the final model (with added layers)

The final accuracy of the model (with 5 epochs) is 98.88 %.



```
Epoch 2/5
300/300 - 10s - loss: 0.0942 - accuracy: 0.9713 - val_loss: 0.0505 - val_accuracy: 0.9830 - 10s/epoch - 33ms/step
Epoch 3/5
300/300 - 10s - loss: 0.0676 - accuracy: 0.9791 - val_loss: 0.0368 - val_accuracy: 0.9871 - 10s/epoch - 33ms/step
Epoch 4/5
300/300 - 10s - loss: 0.0543 - accuracy: 0.9833 - val_loss: 0.0335 - val_accuracy: 0.9882 - 10s/epoch - 33ms/step
Epoch 5/5
300/300 - 10s - loss: 0.0477 - accuracy: 0.9850 - val_loss: 0.0359 - val_accuracy: 0.9888 - 10s/epoch - 33ms/step
```

Figure 5: System accuracy for the final model, with 5 epochs