Learning with Artificial Neural Networks

Practical Work 04 – Deep Neural Networks

Francesco Monti

13.05.2022

Contents

Introduction **3**

Report **3**

1. RMSprop . . . 3
    RMSprop parameters . . . 4
2. Model complexity . . . 4
    MLP from raw data . . . 4
    MLP from features . . . 5
    CNN . . . 6
3. Shallow and Deep Neural networks . . . 7
4. Notebooks testing . . . 8
    MLP raw . . . 8
    MLP with features . . . 11
    MNIST Digits CNN . . . 15
    MNIST Fashion CNN . . . 18
    Remarks . . . 22
5. MNIST Fashion CNN . . . 23
    Model 1 . . . 23
    Model 2 . . . 24
    Model 3 . . . 26

Conclusion **31**

Introduction

In this PW we will explore different ways to classify images of digits using a *MLP*, a *MLP from Histogram of Gradients* and a *CNN*. The goals of this PW are to have a better understanding of the differences between a shallow and a deep neural network, to understand the fundamentals of convolutional neural netwrks and to learn the basics of the *Keras* framework.

We will use the *MNIST* digit dataset which is formed of 28x28 pixels images of digits ranging from 0 to 9.

Report

1. RMSprop

The optimization algorithm used in this case is called Root Mean Square Propagation (RMSprop) and is a gradient descent algorithm. This algorithm is similar to the Adaptive Gradient algorithm (AdaGrad) but improves on the decay of the learning rate, preventing it to converge too quickly. It's equations are the following :

$$s = \beta s + (1 - \beta)\nabla_\theta J(\theta) \odot \nabla_\theta J(\theta) \tag{1}$$

$$\theta = \theta - \eta \nabla_\theta J(\theta) \oslash \sqrt{s + \epsilon} \tag{2}$$

When using the *RMSprop* optimizer in *Keras* a number of parameters can be defined. The class constructor has this signature:

```
tf.keras.optimizers.RMSprop(
    learning_rate=0.001,
    rho=0.9,
    momentum=0.0,
    epsilon=1e-07,
    centered=False,
    name="RMSprop",
    **kwargs
)
```

RMSprop parameters

- `learning_rate`: the initial learning rate of the model, will be adapted further
- `rho`: the discounting factor for the coming/past gradient. Prevents the learning rate to converge too quickly by decaying the contribution of old gradients
- `momentum`: the given momentum for the learning
- `epsilon`: a small constant for stability, it mainly prevents a division by 0
- `centered`: a boolean setting the normalization mode. Setting to `True` may help with training but hurts the performances a bit
- `name`: optional prefix for naming the operations on the gradients

The loss function used for this model is the *categorical cross-entropy*, computing the sum of each target value multiplied with the logarithm of it's predicted value.

2. Model complexity

MLP from raw data

The chosen topology is a sequential model with 3 layers, a 256-neurons input layer, with *200'960* weights, a hidden dropout layer with 256 neurons and *0* weights and finally a 10-neurons output layer with *2570* weights. Both input and ouptut layers are fully-connected layers. The total number of weights for this model is *203'530* as given by the function `model.summary()` provided by *Keras*.

```
Model: "MLP-raw"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 Input (Dense)               (None, 256)               200960

 dropout_6 (Dropout)         (None, 256)               0

 Output (Dense)              (None, 10)                2570

=================================================================
Total params: 203,530
Trainable params: 203,530
Non-trainable params: 0
_________________________________________________________________
```

MLP from features

The chosen topology is also a sequential model with 3 layers, a 256-neurons input layer with *100'608* weights, a hidden dropout layer with no weights and a 10-neurons output layer with *2570* weights. Both input and ouptut layers are fully-connected layers. Like the previous model, the total number of weight is computed by `model.summary()`:

```
Model: "MLP-features"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 Input (Dense)               (None, 256)               100608

 dropout_1 (Dropout)         (None, 256)               0

 Output (Dense)              (None, 10)                2570

=================================================================
Total params: 103,178
Trainable params: 103,178
Non-trainable params: 0
_________________________________________________________________
```

CNN

This topology uses the `FunctionalAPI` of *Keras* to define the model. It provides more flexibility for deep learning models. It consists of a 28x28x1 input layer and a fully-connected 10-neurons output layer with *260* weights, as well as 8 hidden layers.

There are 3 *2D Max Pooling* layers with no weigths, 3 *2D Convolution* layers with *234*, *2034* and *1312* weights, a *Flatten* layer that transforms the (3, 3, 16) input to a (144) output with no weigths and finally a fully-connected *Dense* layer with *3625* weights.

The total amount of weights for this model is *7465* weights, as provided by `model.summary()` as always:

```
Model: "CNN"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 l0 (InputLayer)             [(None, 28, 28, 1)]       0

 l1 (Conv2D)                 (None, 28, 28, 9)         234

 l1_mp (MaxPooling2D)        (None, 14, 14, 9)         0

 l2 (Conv2D)                 (None, 14, 14, 9)         2034

 l2_mp (MaxPooling2D)        (None, 7, 7, 9)           0

 l3 (Conv2D)                 (None, 7, 7, 16)          1312

 l3_mp (MaxPooling2D)        (None, 3, 3, 16)          0

 flat (Flatten)              (None, 144)               0

 l4 (Dense)                  (None, 25)                3625

 l5 (Dense)                  (None, 10)                260

=================================================================
Total params: 7,465
```

```
Trainable params: 7,465
Non-trainable params: 0
_________________________________________________________________
```

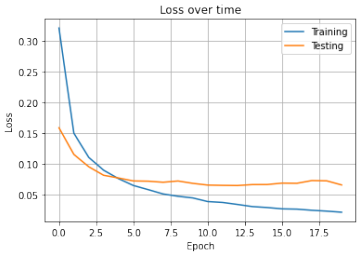
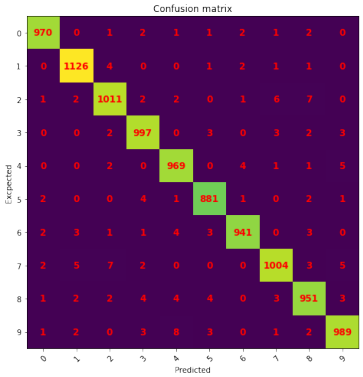
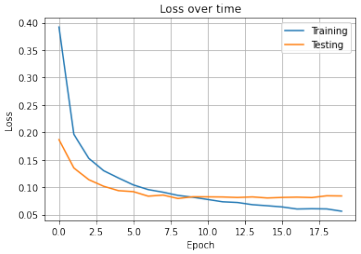
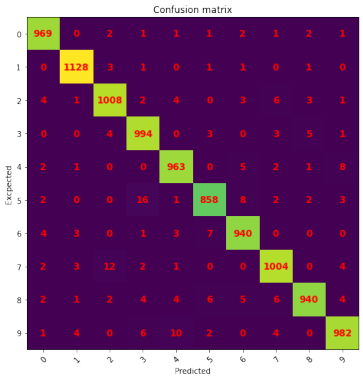
3. Shallow and Deep Neural networks

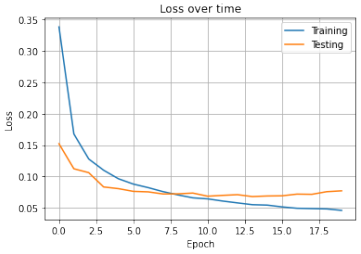
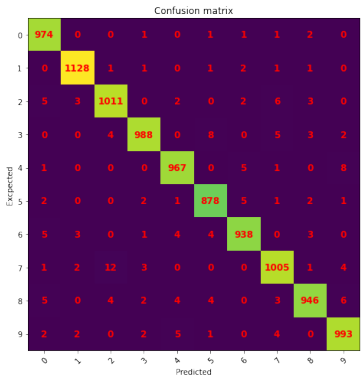
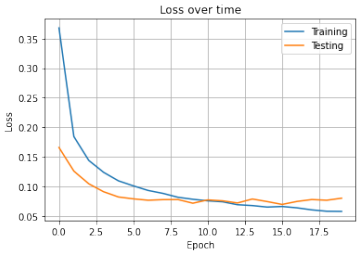
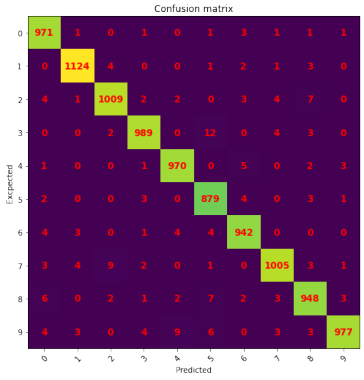
Despite the increased complexity of deep neural networks, their capacity is not bigger than shallow ones. This is mainly because shallow neural networks are composed of *Dense* fully-connected layers. And it is known that *Dense* layers add a lot of capacity to models. This is because each neuron is connected to all the neurons of the next and previous layer (hence the term *fully-connected*).

For instance, the first two experiments have more than *100'000* weights with only 2 layers (the hidden dropout layer doesn't have any weights). In comparison we can observe that the model for the MNIST digits dataset have "only" *7'500* weights, despite having much more layers. Convolutional layers do not need so much weights to work.

4. Notebooks testing

MLP raw

Hyperparameters	Loss	Confusion matrix	Score
256 neurons hidden layer, 0.25 dropout			Score: 0.0662 Accuracy: 98.39%
128 neurons hidden layer, 0.3 dropout			Score: 0.0841 Accuracy: 97.86%

Hyperparameters	Loss	Confusion matrix	Score
384 neurons hidden layer, 0.5 dropout			Score: 0.077 Accuracy: 98.28%
300 neurons hidden layer, 0.5 dropout			Score: 0.0801 Accuracy: 98.14%%

We can see that in this case, the best model we found is the one with 256 neurons in the hidden layer and a layer with a dropout of 0.25:

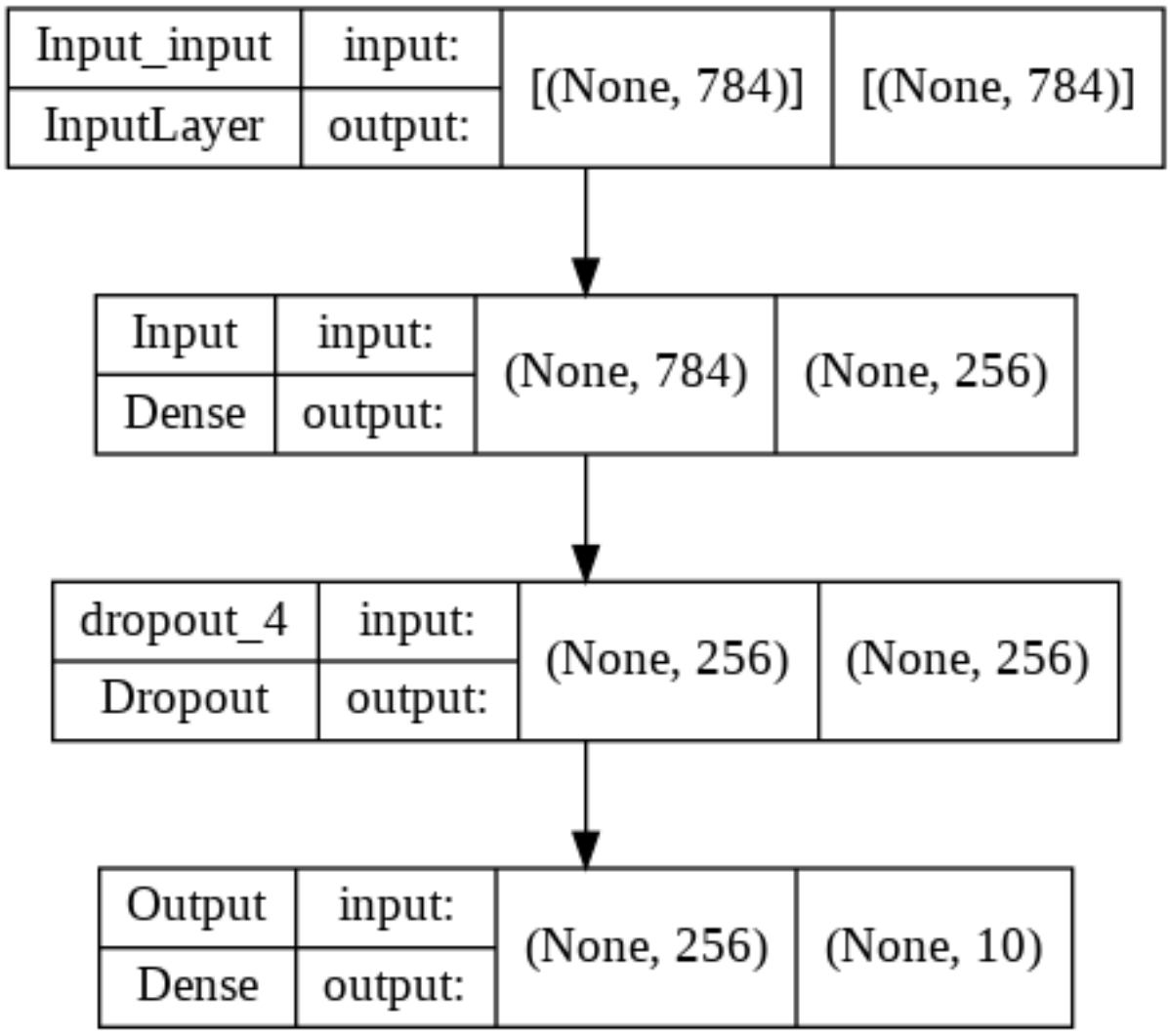
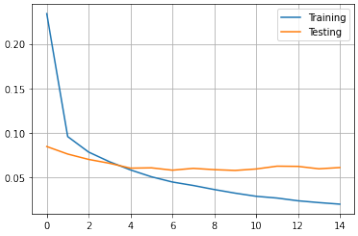
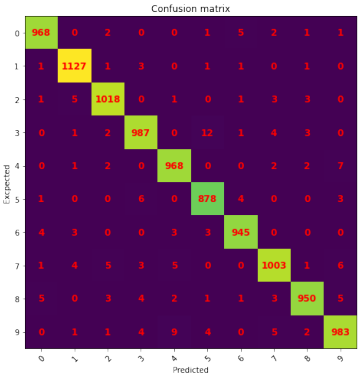
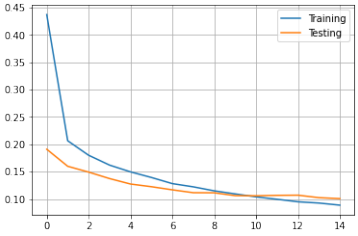
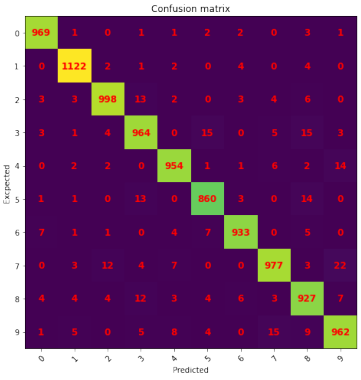
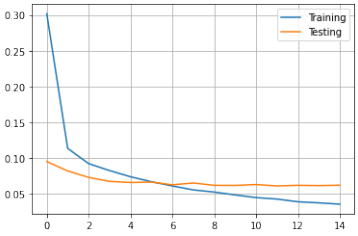
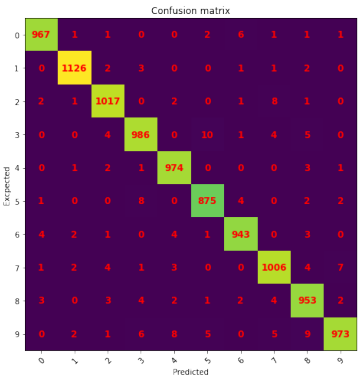
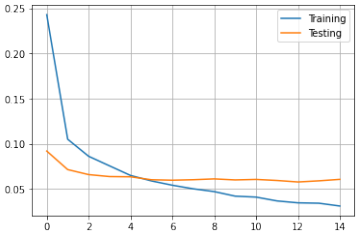
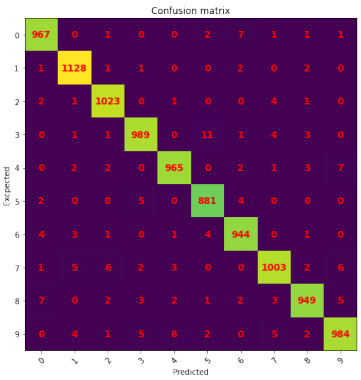
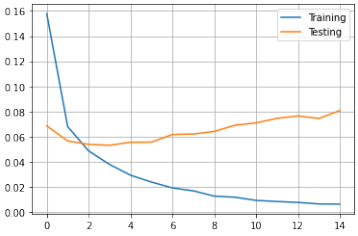
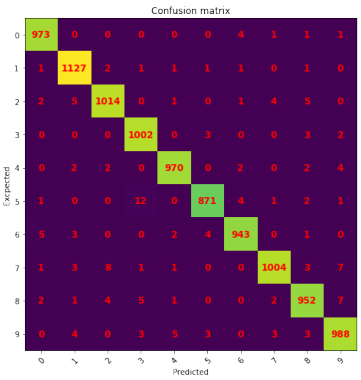


Figure 1: MLP with Raw Data Model

MLP with features

Hyperparameters	Loss	Confusion matrix	Score
8 orientations, 4 pix/cell, 256-neurons hidden layer			Score: 0.0613 Accuracy: 98.27%
8 orientations, 7 pix/cell, 256-neurons hidden layer			Score: 0.1008 Accuracy: 96.66%

Hyperparameters	Loss	Confusion matrix	Score
8 orientations, 4 pix/cell, 128-neurons hidden layer			Score: 0.0622 Accuracy: 98.2%
8 orientations, 4 pix/cell, 128-neurons hidden layer			Score: 0.0607 Accuracy: 98.33%

Hyperparameters	Loss	Confusion matrix	Score
8 orientations, 2 pix/cell, 384-neurons hidden layer			Score: 0.0808 Accuracy: 98.44%

We can see that among these tests, the best model is the one with 8 orientations, 4 pixels per cell and 128 neurons in the hidden layer, with a dropout of 0.25:

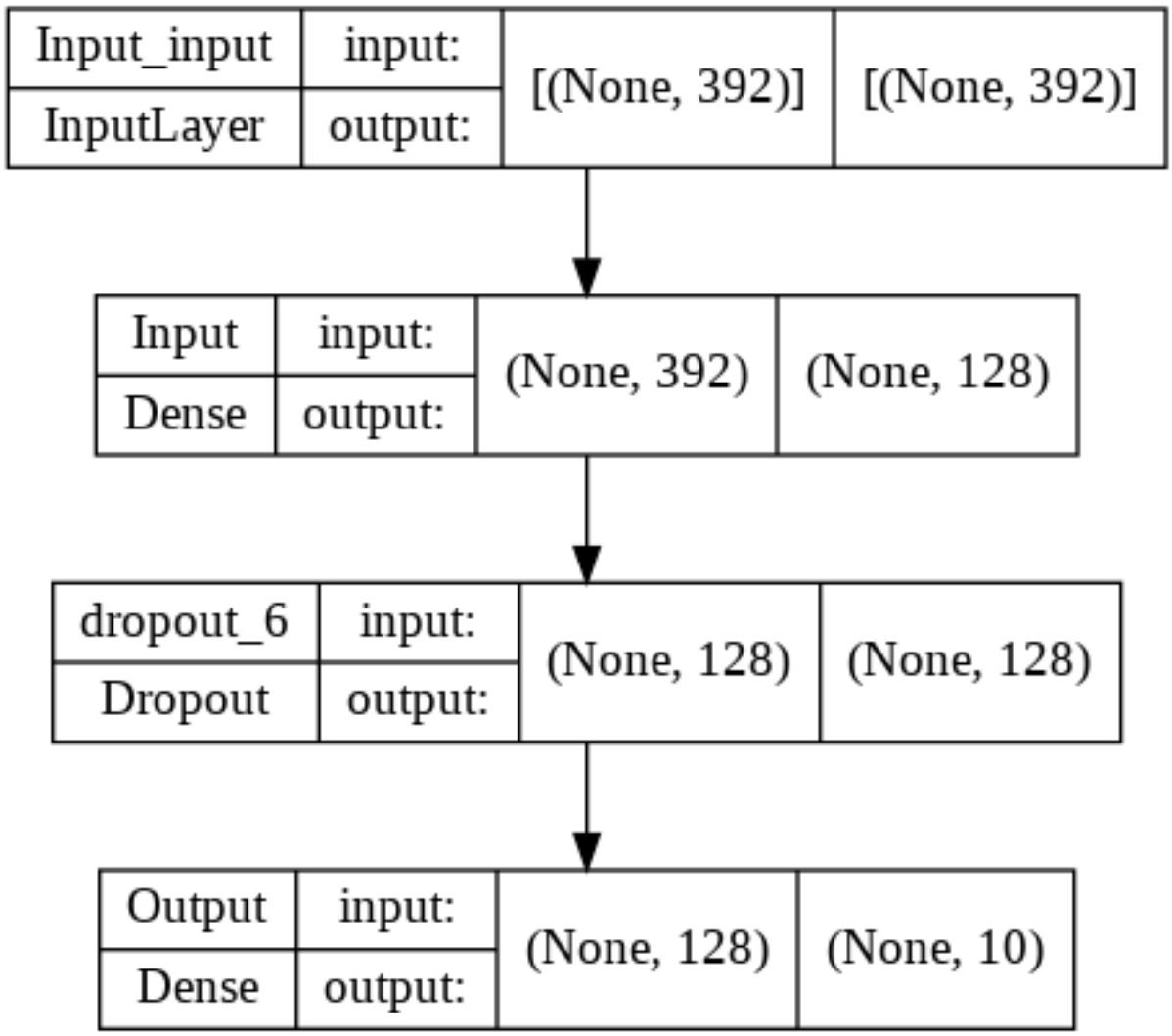
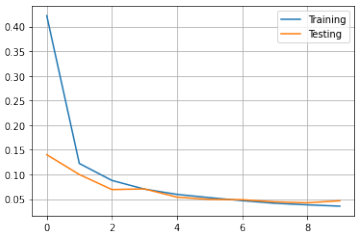
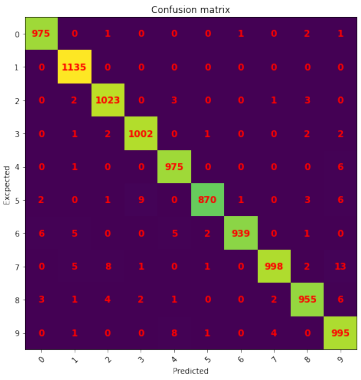
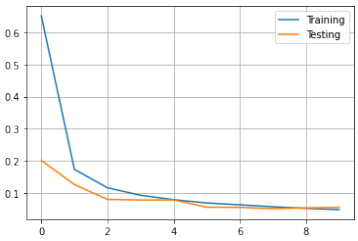
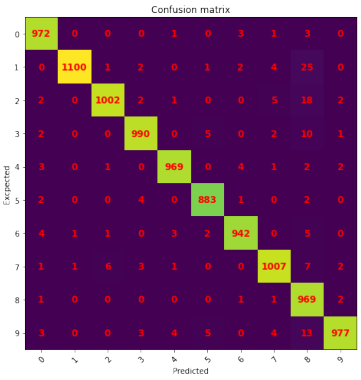
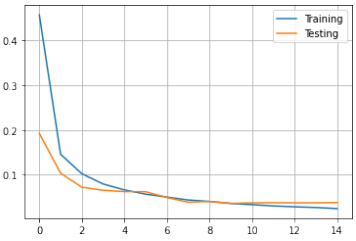
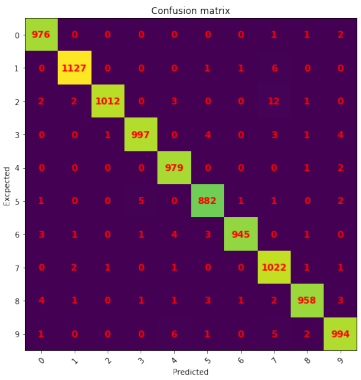
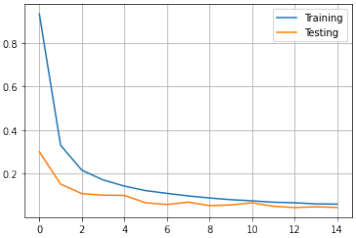
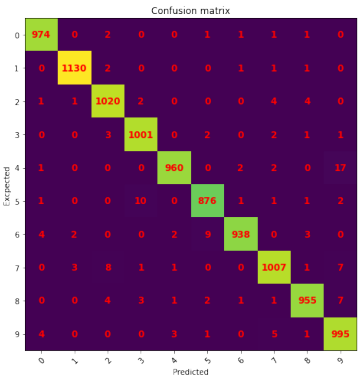


Figure 2: MLP with HOG Model

MNIST Digits CNN

Hyperparameters	Loss	Confusion matrix	Score
2 *Conv2D* (5x5x9) filters, 1 *Conv2D* (3x3x18) filter, each with *MaxPooling*, a *Flatten* layer, 25-neurons hidden layer			Score: 0.0465 Accuracy: 98.67%
2 *Conv2D* (5x5x9) filters, 1 *Conv2D* (3x3x18) filter, each with *MaxPooling*, a *Flatten* layer, 8-neurons hidden layer			Score: 0.0564 Accuracy: 98.11%

Hyperparameters	Loss	Confusion matrix	Score
2 *Conv2D* (5x5x9) filters, 1 *Conv2D* (3x3x18) filter, each with *MaxPooling*, a *Flatten* layer, 32-neurons hidden layer			Score: 0.0379 Accuracy: 98.92%
2 *Conv2D* (5x5x9) filters, 1 *Conv2D* (3x3x18) filter, each with *MaxPooling*, a *Flatten* layer,16-neurons layer, *Dropout* layer of 0.3, 8-neurons hidden layer			Score: 0.0451 Accuracy: 98.56%

We can see that among these tests, the best model is the one with 32-neurons in the hidden layer and the three *Conv2D* filters with *MaxPooling* before that:

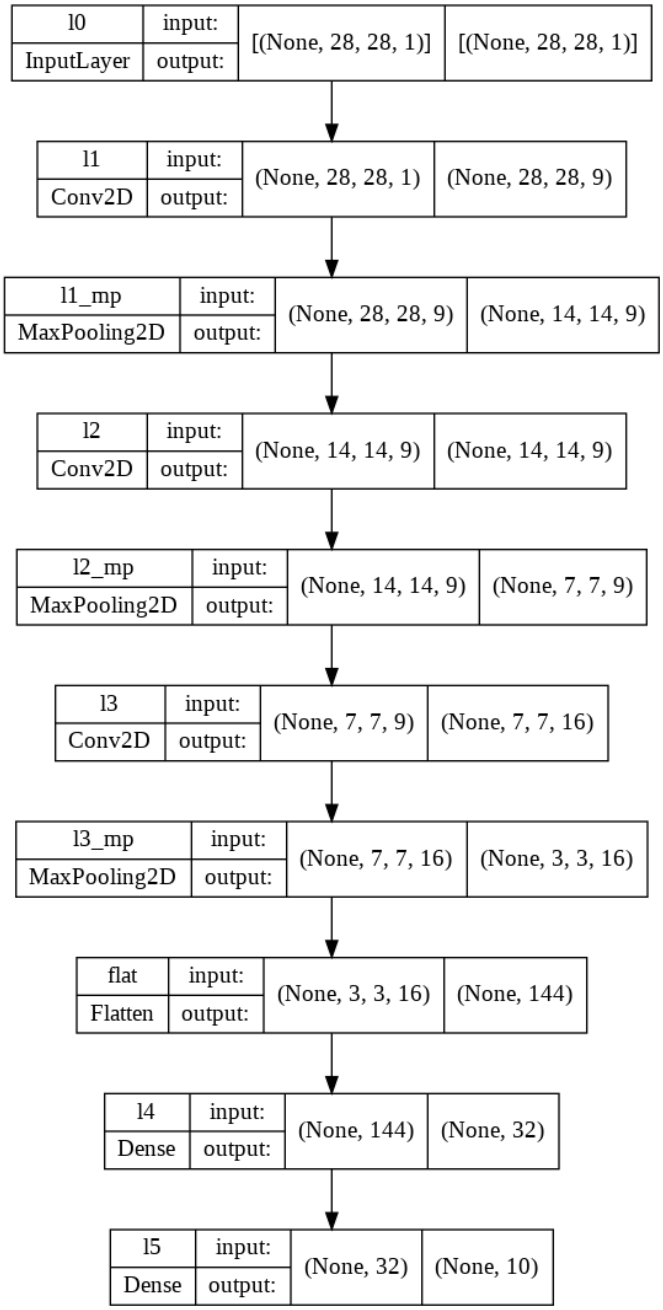
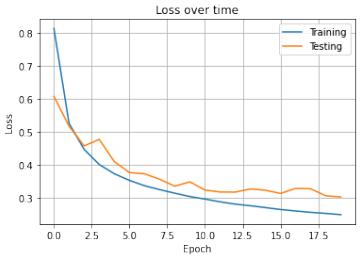
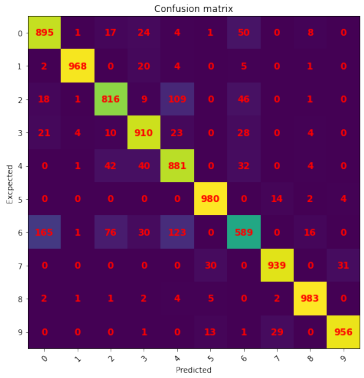
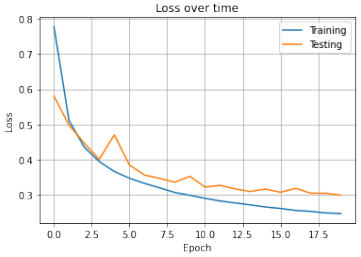
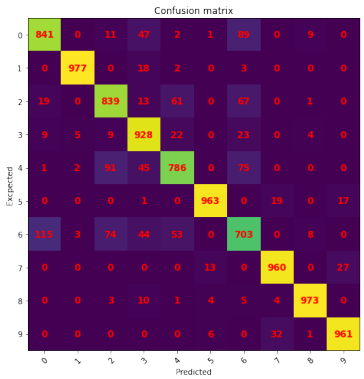
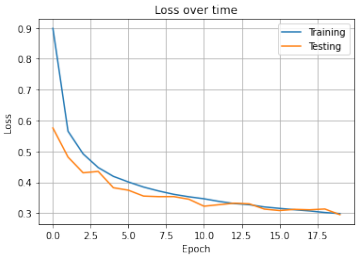
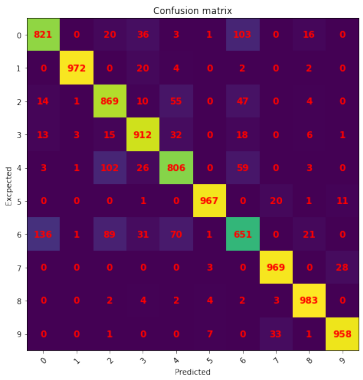
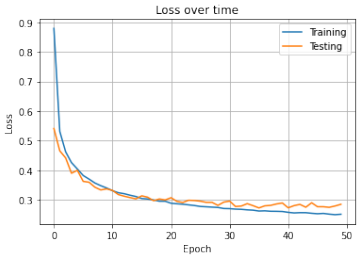
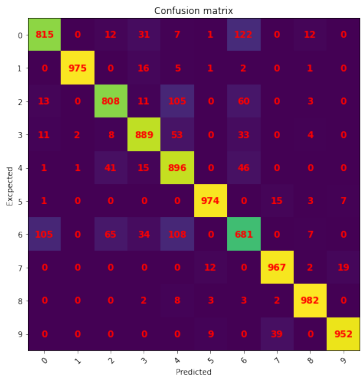


Figure 3: CNN Model

MNIST Fashion CNN

Hyperparameters	Loss	Confusion matrix	Score
2 *Conv2D* (5x5x9) filters, 1 *Conv2D* (3x3x18) filter, each with *MaxPooling*, a *Flatten* layer, 25-neurons hidden layer			Score: 0.3018 Accuracy: 89.17%
2 *Conv2D* (5x5x9) filters, 1 *Conv2D* (3x3x18) filter, each with *MaxPooling*, a *Flatten* layer, 32-neurons hidden layer			Score: 0.2981 Accuracy: 89.31%

Hyperparameters	Loss	Confusion matrix	Score
2 *Conv2D* (3x3) filters, 1 *Conv2D* (5x5) filter, 1 *Conv2D* (7x7) filter each with *MaxPooling*, a *Dropout* layer of 0.3, a *Flatten* layer, 16-neurons hidden layer			Score: 0.2951 Accuracy: 89.08%

Hyperparameters	Loss	Confusion matrix	Score
2 *Conv2D* (3x3) filters, 1 *Conv2D* (5x5) filter, 1 *Conv2D* (7x7) filter each with *MaxPooling*, a *Dropout* layer of 0.3, a *Flatten* layer, 32-neurons hidden layer			Score: 0.2731 Accuracy: 90.27%

We can immediatly see that our previous best model is struggling a lot more with this dataset. After a bit of trial-and-error we found that the model with a (7x7x9) *Conv2D* filter with *MaxPooling*, a (5x5) *Conv2D* filter with *MaxPooling*, 2 (3x3) *Conv2D* filter with *MaxPooling*, a *Dropout* layer of 0.3, a *Flatten* layer and a hidden *Dense* 32-neurons layer can do a pretty good job. There is the more convoluted plot of the model:

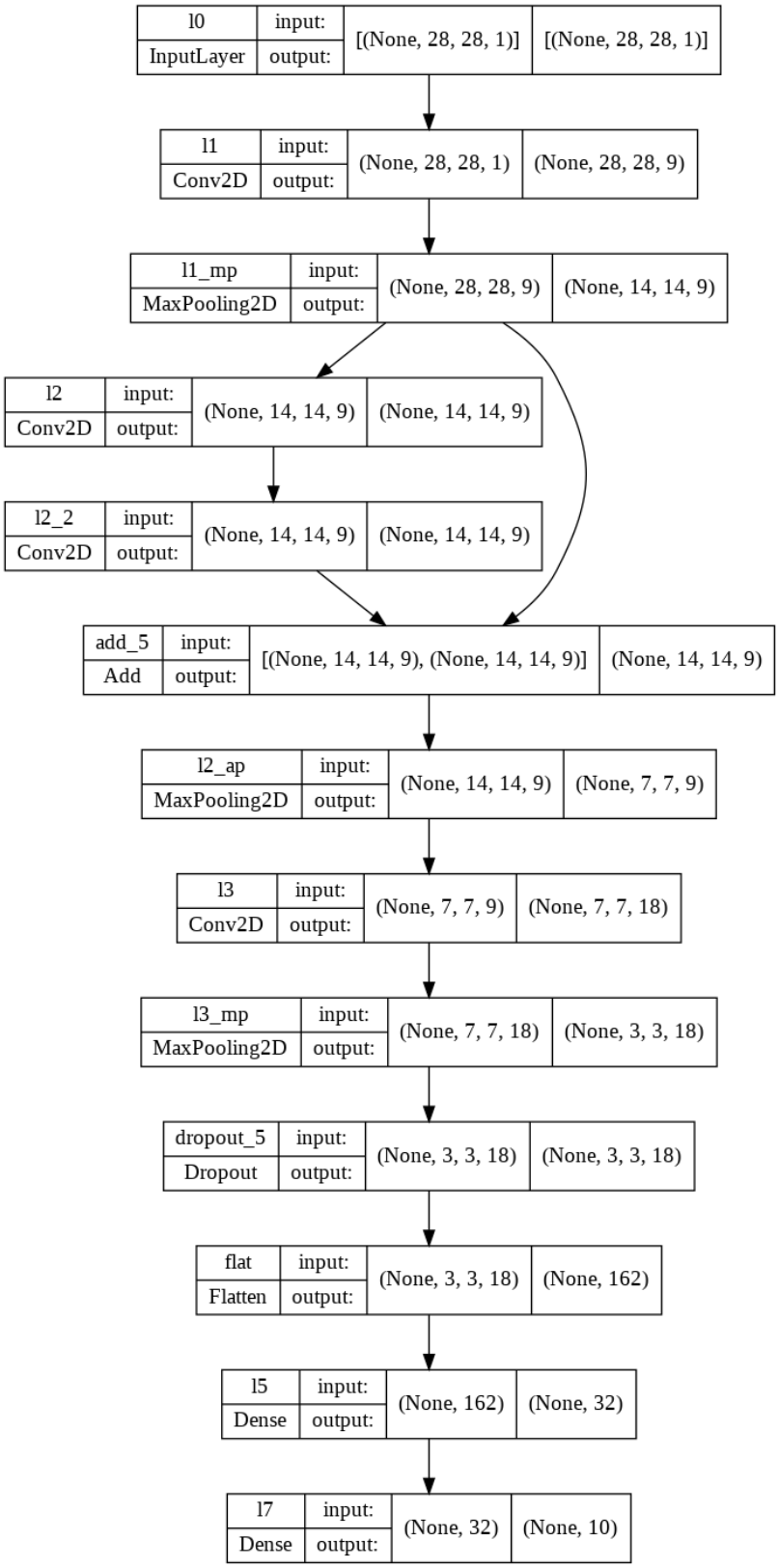


Figure 4: CNN-F Model

Remarks

We can see that the CNN model is better at identifying the digits than the MLP models. In our tests, we do not have significants digits mismatches or frequents confusions.

For the MNIST Fashion dataset we can see that we often mis-identify the sixth class (*shirt*), but we know that the dataset is harder to get absolutely right. We think that our custom model did a pretty good job.

5. MNIST Fashion CNN

To model and train a new CNN to use with the MNIST Fashion dataset, we can take the previous notebook for the “classic” MNIST dataset and start from here.

Model 1

At first we tried to lower the amount of layers present in the model but increase the number of filters and the number of neurons in the hidden dense layer. The model looks like this :

```
Model: "CNN-fashion"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 l0 (InputLayer)             [(None, 28, 28, 1)]       0

 l1 (Conv2D)                 (None, 28, 28, 32)        320

 l1_mp (MaxPooling2D)        (None, 14, 14, 32)        0

 flat (Flatten)              (None, 6272)              0

 l6 (Dense)                  (None, 100)               627300

 l5 (Dense)                  (None, 10)                1010

=================================================================
Total params: 628,630
Trainable params: 628,630
Non-trainable params: 0
_________________________________________________________________
```

We can see that of the 3 *Conv2D* layers we only have one, but with 32 filters. The *l6* layer also has 100 neurons compared to the previous 25. After training and evaluation we can see that the performances aren’t good at all :

```
Test score: 0.445
Test accuracy: 91.8%
```

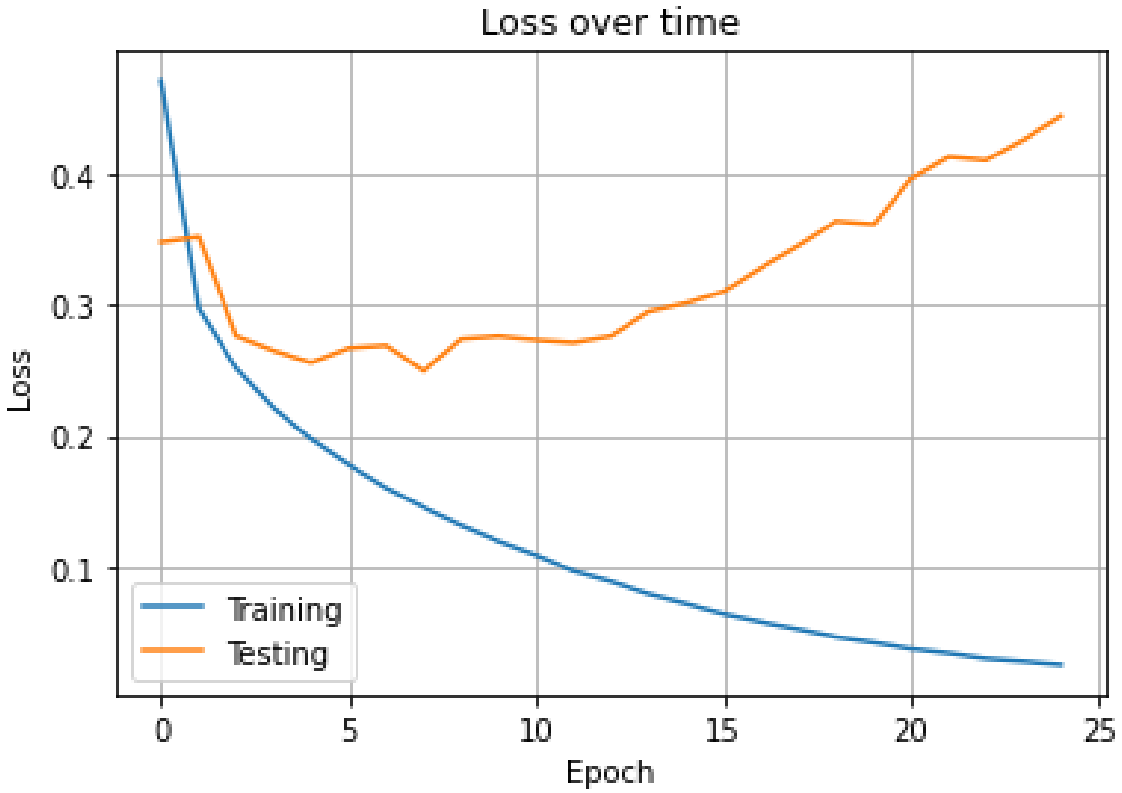


Figure 5: Model 1

Model 2

We can try to add some hidden layers like another *Conv2D* layer with a 5x5 kernel and 8 filters and reduce the number of filters of the 3x3 *Conv2D* layer to 16. We can also reduce the number of neurons in the hidden *Dense* layer to 16 :

```
Model: "CNN-fashion"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 l0 (InputLayer)             [(None, 28, 28, 1)]       0

 l1 (Conv2D)                 (None, 28, 28, 8)         208

 l1_mp (MaxPooling2D)        (None, 14, 14, 8)         0
```

```
 l2 (Conv2D)                 (None, 14, 14, 16)        1168

 l2_mp (MaxPooling2D)        (None, 7, 7, 16)          0

 flat (Flatten)              (None, 784)               0

 l6 (Dense)                  (None, 16)                12560

 l5 (Dense)                  (None, 10)                170

=================================================================
Total params: 14,106
Trainable params: 14,106
Non-trainable params: 0
_________________________________________________________________
```

After training and evaluation we have a better result than before, but it's not quite what we want :

```
Test score: 0.2801
Test accuracy: 90.3%
```

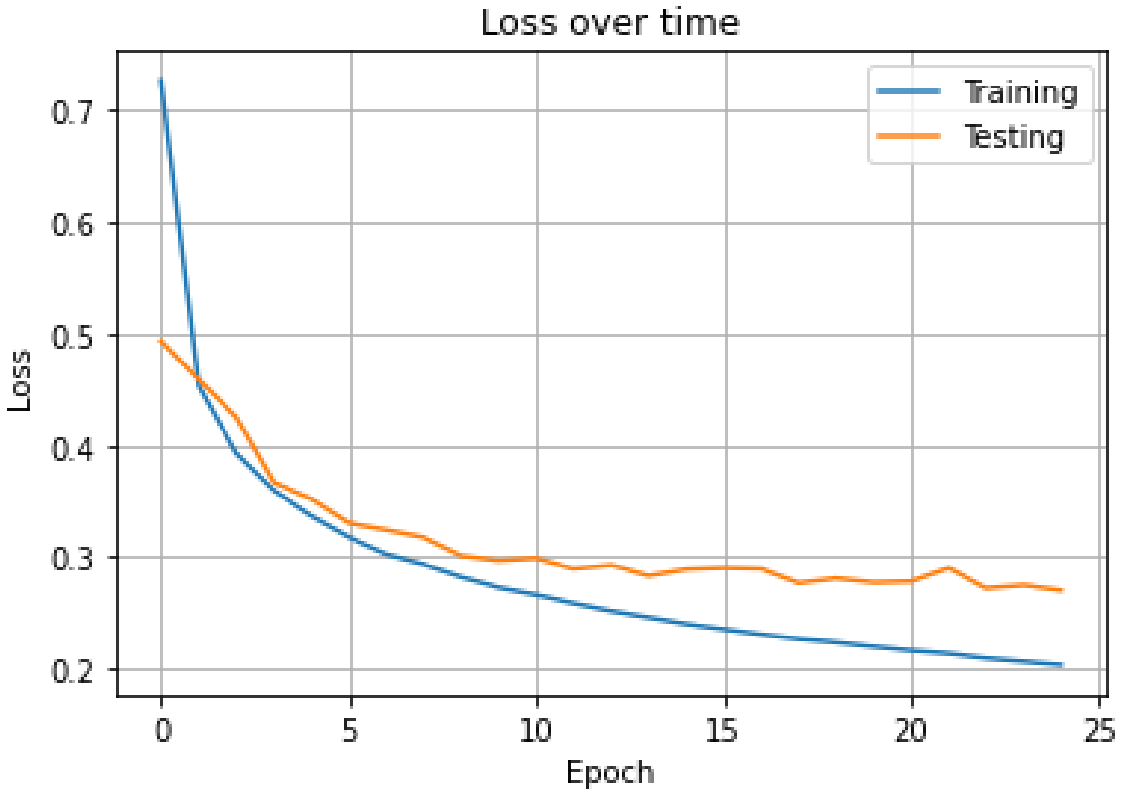


Figure 6: Model 2

Model 3

After that we went a bit crazy with the model. We took inspiration from existing CNNs like ResNet and we tried to mimick some of the behaviours, like the bypasses and the $x \to F(x) + x$ mapping. That's the reason for the much more complex model :

```
Model: "CNN-fashion"
__________________________________________________________________________________
 Layer (type)                Output Shape              Param #    Connected to
==================================================================================
 l0 (InputLayer)             [(None, 28, 28, 1)]       0          []

 l1 (Conv2D)                 (None, 28, 28, 9)         450        ['l0[0][0]']

 l1_mp (MaxPooling2D)        (None, 14, 14, 9)         0          ['l1[0][0]']
```

```
 l2 (Conv2D)                 (None, 14, 14, 9)   2034      ['l1_mp[0][0]']

 l2_2 (Conv2D)                (None, 14, 14, 9)   738       ['l2[0][0]']

 add_5 (Add)                  (None, 14, 14, 9)   0         ['l2_2[0][0]',
                                                             'l1_mp[0][0]']

 l2_ap (MaxPooling2D)         (None, 7, 7, 9)     0         ['add_5[0][0]']

 l3 (Conv2D)                 (None, 7, 7, 18)    1476      ['l2_ap[0][0]']

 l3_mp (MaxPooling2D)         (None, 3, 3, 18)    0         ['l3[0][0]']

 dropout_5 (Dropout)          (None, 3, 3, 18)    0         ['l3_mp[0][0]']

 flat (Flatten)              (None, 162)         0        ['dropout_5[0][0]']

 l5 (Dense)                   (None, 32)          5216      ['flat[0][0]']

 l7 (Dense)                   (None, 10)          330       ['l5[0][0]']

==================================================================================
Total params: 10,244
Trainable params: 10,244
Non-trainable params: 0
----------------------------------------------------------------------------------
```

After the training we can see that the added complexity did not add much more accuracy, but it's better than the previous attempt :

```
Test score: 0.2731
Test accuracy: 90.27%
```

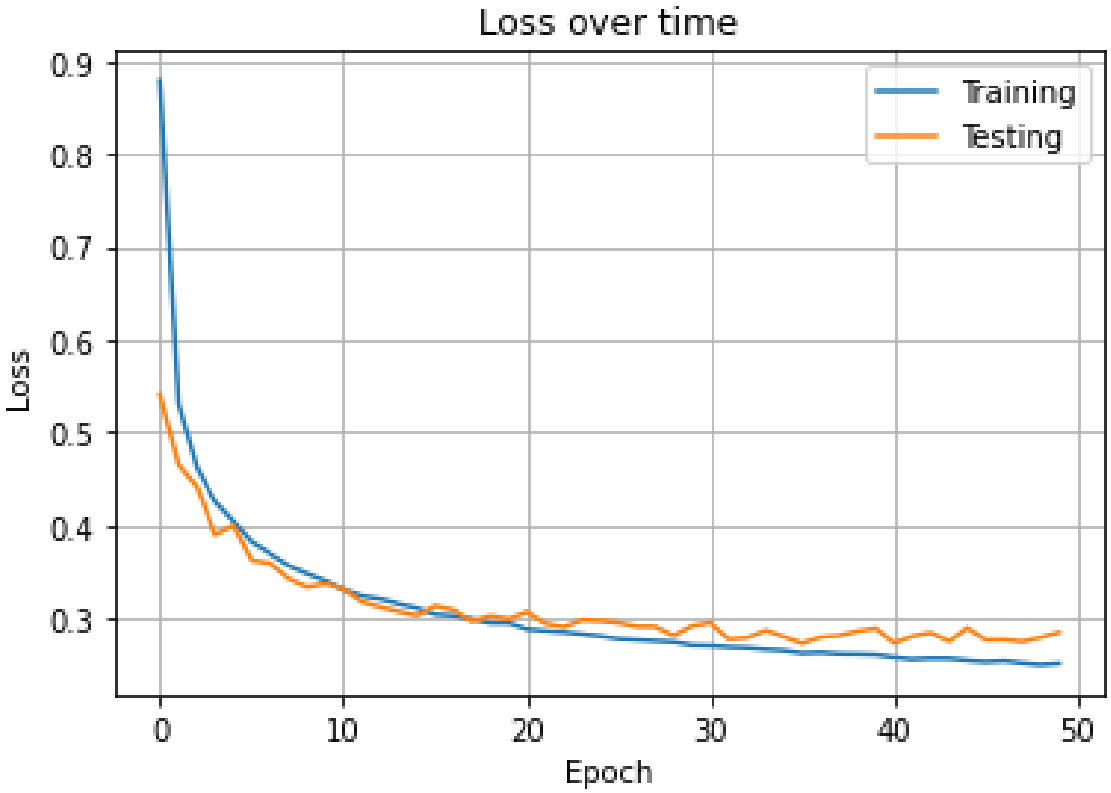


Figure 7: Model 3

We can see by the confusion matrix that we often confuse the *Shirt* class with other classes like the *T-Shirt* and the *Coat*. Which is understandable to some extent.

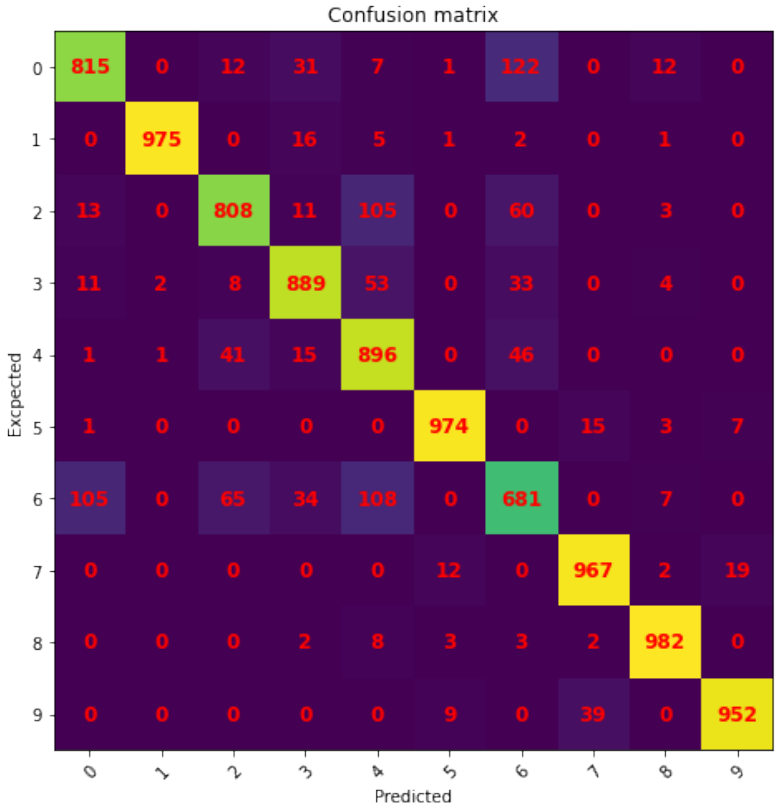


Figure 8: Model 3 - Confusion Matrix

The F1-scores are as following:

```
F1-Score for class "T-shirt/top": 0.84
F1-Score for class "Trouser": 0.98
F1-Score for class "Pullover": 0.83
F1-Score for class "Dress": 0.9
F1-Score for class "Coat": 0.82
F1-Score for class "Sandal": 0.97
F1-Score for class "Shirt": 0.7
F1-Score for class "Sneaker": 0.96
F1-Score for class "Bag": 0.98
F1-Score for class "Ankle boot": 0.96
```

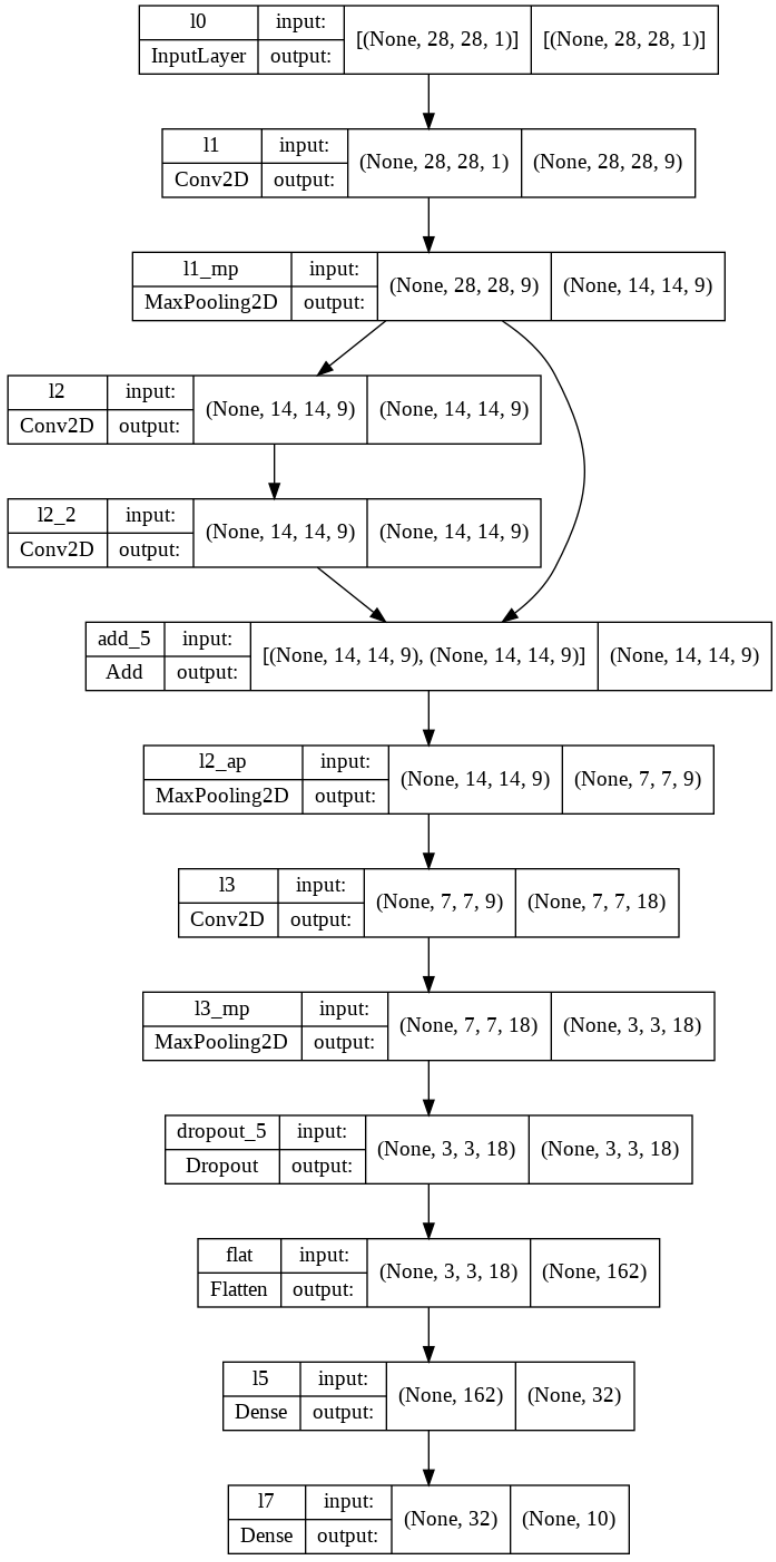


Figure 9: Model 3 - Plot

Conclusion

In the end the result we obtained for the MNIST Fashion dataset is not the best we can find but I think it's enough for now. It was very interesting to learn how *CNNs* work and to experiment with the `Keras` framework to try different approaches and models. One thing that stood out during this practical work was the time it took to compute and train the models. On my laptop it took almost 15 minutes to train the *MLP with HOG*, which quickly hinders the progression and reduces the trials we can make. One solution was to take advantage of Google Colab and its free tier, which provided a decent GPU. As `Keras` is built on top of `Tensorflow` and the latter is built with GPU and multi-threading support, it sped up the process by a lot (~2 min instead of 15). It allowed me to experiment with more convoluted and complex models (even if they were awful), which in term helped me understand better *CNNs*.