ARN Report lab 04

Authors : Junod Arthur & Häffner Edwin

1. Learning Algorithm and Parameters

What is the learning algorithm being used to optimize the weights of the neural networks?

RMSprop aka "Root Mean Square Propagation"

$$E[g^2](t) = \beta E[g^2](t-1) + (1-\beta)(\frac{\partial c}{\partial w})^2$$

$$w_{ij}(t) = w_{ij}(t-1) - \frac{\eta}{\sqrt{E[g^2]}} \frac{\partial c}{\partial w_{ij}}$$

Where we have:

- E[g] as the moving average of squared gradients
- δc/δw as the gradient of the cost function with respect to the weight
- η as the learning rate
- β as the moving average parameter

(source)

What are the parameters (arguments) being used by that algorithm?

From the official documentation of Keras we could find those arguments :

```
keras.optimizers.RMSprop(
    learning_rate=0.001,
    rho=0.9,
    momentum=0.0,
    epsilon=1e-07,
    centered=False,
    weight_decay=None,
    clipnorm=None,
    clipvalue=None,
    global_clipnorm=None,
    use_ema=False,
    ema_momentum=0.99,
    ema_overwrite_frequency=None,
    loss_scale_factor=None,
    gradient_accumulation_steps=None,
    name="rmsprop",
    **kwargs
)
```

What loss function is being used?

The loss function used is named "categorical_crossentropy" The equation is :

$$CE = -\sum_{i}^{C} t_i log(s_i)$$

Where we have:

- s_i as the model's predicted probability distribution over the classes for sample i.
- t_i as the true one-hot encoded class label for sample i.

2. Experiments

Digit recognition from raw data

1. Neural Network Topology

- The neural network topology used in this code is a Multilayer Perceptron (MLP) with the following structure:
  - Input layer: 784 neurons (corresponding to the flattened 28x28 pixel input images)
  - Hidden layer: 512 neurons using ReLU activation
  - Output layer: 10 neurons with Softmax activation (corresponding to the 10 digit classes)

```
model = Sequential()
model.add(Input(shape=(784,)))
model.add(Dense(512, activation='relu'))
model.add(Dense(n_classes, activation='softmax'))
```

Layer (type)	Output Shape	Param #
dense_4 (Dense)	(None, 512)	401,920
dense_5 (Dense)	(None, 10)	5,130
Total params: 407,050 (1.55 MB)		
Trainable params: 407,050 (1.55 MB)		
Non-trainable params: 0 (0.00 B)		

We chose 512 neurons for the hidden layer as there's already a lot of inputs (784). We thought that lowering it too quickly would not yield great results.

2. Model Weights

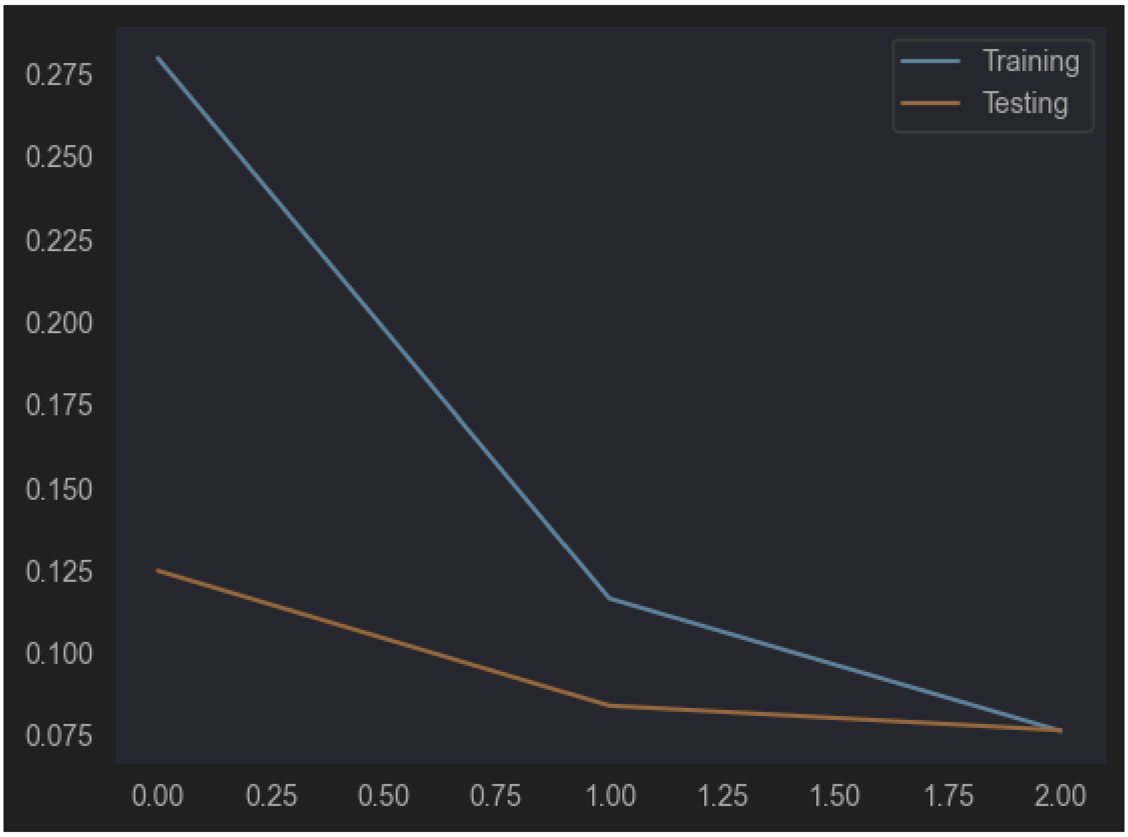
- Here are all the weights linked to this model :
  - Input to Hidden layer weights: 784 (inputs) x 512 (hidden neurons) = 401,408 weights
  - Hidden layer biases: 512 biases
  - Hidden to Output layer weights: 512 (hidden neurons) x 10 (output neurons) = 5,120 weights
  - Output layer biases: 10 biases
  - Total weights: 401,408 + 512 + 5,120 + 10 = 407,050 weights

3. Test Cases

First pass

With our baseline here are our results, we have not changed the base settings that were provided with the notebook :

Test score: 0.08300229161977768 Test accuracy: 0.9747999906539917



Confusion matrix :

	0	1	2	3	4	5	6	7	8	9
0	975	0	0	1	0	2	0	1	1	0
1	0	1117	3	2	0	2	4	3	4	0
2	5	0	1005	7	2	1	1	7	4	0
3	0	0	1	993	0	4	0	5	3	4
4	2	0	2	0	965	0	4	2	1	6
5	2	0	0	11	1	866	5	0	4	3
6	7	2	0	1	5	4	937	1	1	0
7	2	4	9	5	0	0	0	1000	1	7
8	4	0	4	18	3	9	5	5	923	3
9	4	4	0	10	12	3	1	7	1	967

Discussion of the result :

With those very basic settings and only 3 epochs, we're already getting some good results, the accuracy is quite high and we have don't see any overfitting on the graph.

On the other hand, we can see that the class 3 seems to be the one to get the most confused with the other classes while the 1 has only 10 confusions.

Here's few settings we need to change :

- Number of epochs
  - With only three epochs, it's hard to realize if we're overfitting or not
- The amound of hidden layers and maybe try to use a dropOut layer

Second pass

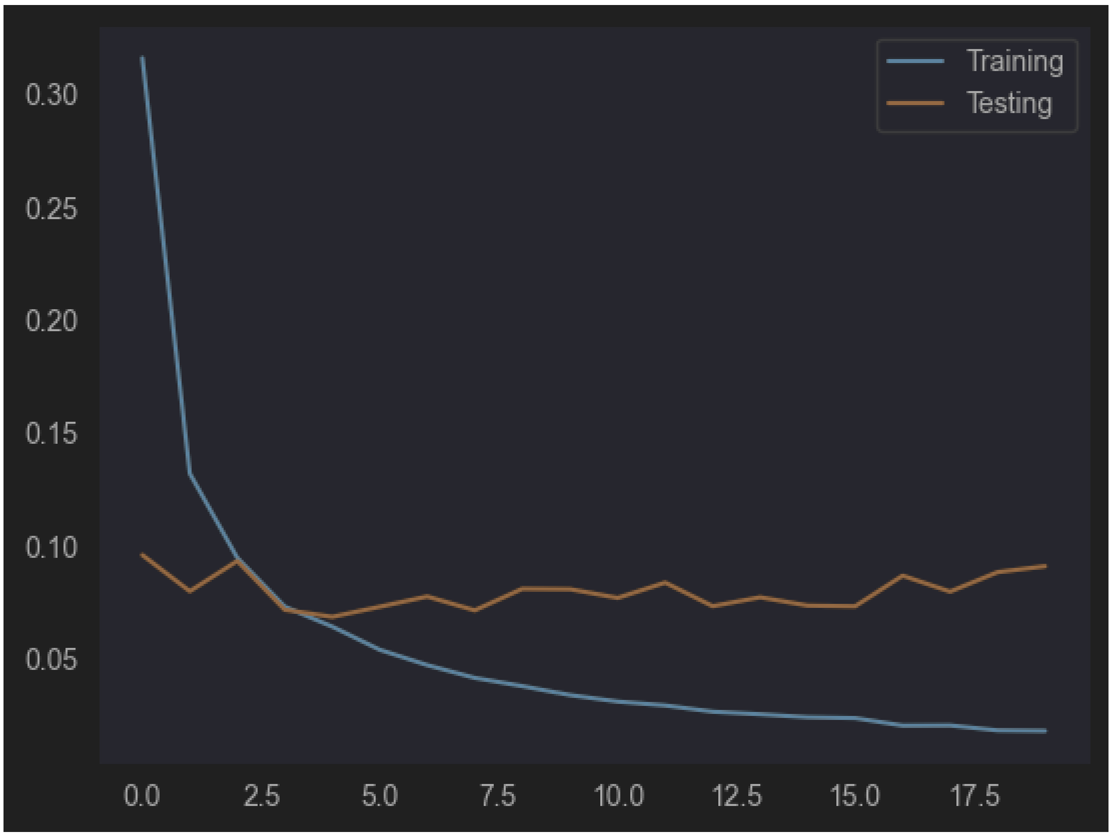
For our second try, we used these layers :

```
model = Sequential()
model.add(Input(shape=(784,)))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2)) #0.5 felt too harsh, bad results with it
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))

model.add(Dense(n_classes, activation='softmax'))
```

Layer (type)	Output Shape	Param #
dense_13 (Dense)	(None, 512)	401,920
dropout_4 (Dropout)	(None, 512)	0
dense_14 (Dense)	(None, 256)	131,328
dropout_5 (Dropout)	(None, 256)	0
dense_15 (Dense)	(None, 128)	32,896
dropout_6 (Dropout)	(None, 128)	0
dense_16 (Dense)	(None, 10)	1,290
Total params: 567,434 (2.16 MB)		
Trainable params: 567,434 (2.16 MB)		
Non-trainable params: 0 (0.00 B)		

And here are our results with the original parameters but 20 epochs :

Test score: 0.07446609437465668 Test accuracy: 0.9854000210762024



Confusion matrix:

	0	1	2	3	4	5	6	7	8	9
0	974	1	0	1	0	0	3	1	0	0
1	0	1128	0	0	0	1	2	1	3	0
2	0	2	1014	2	2	0	2	6	4	0
3	0	0	4	996	0	4	0	2	2	2
4	2	1	0	0	968	0	6	1	1	3
5	1	0	0	5	1	880	1	1	2	1
6	2	1	1	0	2	4	948	0	0	0
7	2	1	7	0	0	0	0	1012	1	5
8	6	1	3	4	3	6	2	3	943	3
9	0	2	0	1	7	3	1	4	0	991

Discussion of the result :

In this model, we tried to use a tapered structure to force the network to learn to compact the data.

As we can see, there's clear overfitting with those layers and parameters.

But it still performs better than the first pass with a lower test score and higher accuracy.

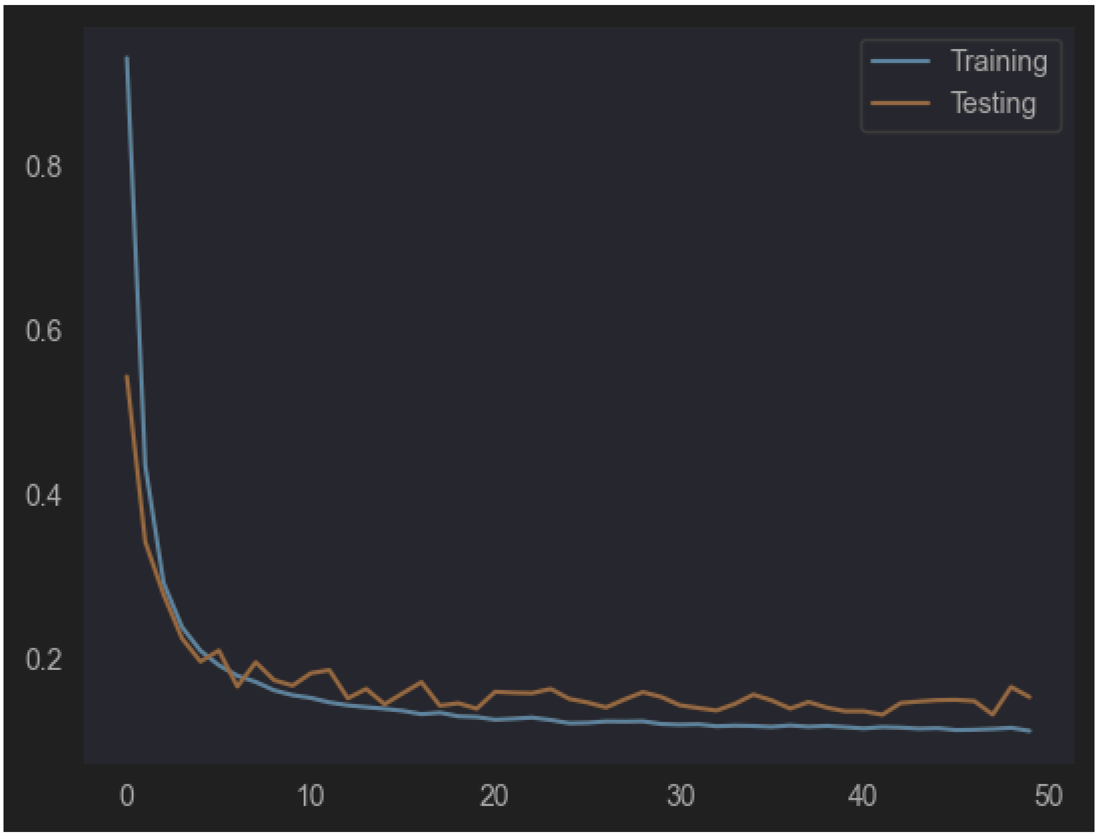
Third pass

```
model = Sequential()
model.add(Input(shape=(784,)))
model.add(Dense(512, activation='relu', kernel_regularizer=keras.regularizers.l2(0.001)))
model.add(BatchNormalization())
model.add(Dense(256, activation='relu', kernel_regularizer=keras.regularizers.l2(0.001)))
model.add(BatchNormalization())
model.add(Dropout(0.2))
model.add(Dense(128, activation='relu', kernel_regularizer=keras.regularizers.l2(0.001)))
model.add(BatchNormalization())
model.add(Dropout(0.2))
model.add(Dense(n_classes, activation='softmax'))
```

Layer (type)	Output Shape	Param #
dense_29 (Dense)	(None, 512)	401,920
batch_normalization_7 (BatchNormalization)	(None, 512)	2,048
dense_30 (Dense)	(None, 256)	131,328
batch_normalization_8 (BatchNormalization)	(None, 256)	1,024
dropout_13 (Dropout)	(None, 256)	0
dense_31 (Dense)	(None, 128)	32,896
batch_normalization_9 (BatchNormalization)	(None, 128)	512
dropout_14 (Dropout)	(None, 128)	0
dense_32 (Dense)	(None, 10)	1,290
Total params: 571,018 (2.18 MB)		
Trainable params: 569,226 (2.17 MB)		
Non-trainable params: 1,792 (7.00 KB)		

In the parameters we added some weight_decay of 1e-4 and effectued 50 epochs, here are our results :

Test score: 0.16643929481506348 Test accuracy: 0.968999981880188



Confusion matrix :

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	966	1	1	0	3	1	4	1	3	0
1	0	1121	1	2	1	1	2	2	3	2
2	3	3	995	13	1	1	1	6	9	0
3	0	0	0	984	0	9	0	5	4	8
4	0	2	5	1	965	0	1	1	4	3
5	3	0	0	7	3	865	7	2	4	1
6	4	2	1	1	14	7	924	0	5	0
7	1	3	8	3	15	0	0	990	5	3
8	1	4	0	4	7	3	1	5	948	1
9	3	1	0	3	56	5	0	5	4	932

Discussion of the result :

We tried to use the L2 regularization and batch normalization layers to reduce the overfitting done by our previous model.

The overfitting seemed to have been reduced by a bit, but it produced a model with worse performances than the previous one on the second pass.

We can see in the confusion matrix that the 4 is often confused for a 9 by a large margin (the most in any other model: 56 !). And thus the class 4 is the one that is the most often confused for another number.

We believe that the small reduction of overfitting isn't worth the pretty significant reduction of performances.

Discussion of digit recognition from raw data

The best performing model we produced is the second pass one with few confusions classes by classes (not one above 7).

The 1 seems to be the most recognizable class with only 16 confusions for the highest value (third pass model) and 9 for the lowest (second pass model).

3 and 4 look to be hard to differientate from other classes, especially 4 that seems to be often mixed with 9.

Digit recognition from features of the input data

1. Neural Network Topology

- The neural network topology used is a Multilayer Perceptron (MLP) with the following structure:
  - Input layer: hog_size (392) neurons (corresponding to the flattened HOG feature vector)
  - Hidden layer: 64 neurons with ReLU activation
  - Output layer: 10 neurons with Softmax activation (corresponding to the 10 digit classes)

```
model = Sequential()
model.add(Input(shape=(hog_size,)))
model.add(Dense(64, activation='relu'))
model.add(Dense(n_classes, activation='softmax'))
```

We're also using the base values of n_orientations (8) and pix_p_cell (4).

The input layer has hog_size neurons, which corresponds to the length of the HOG feature vector computed from the 28x28 pixel input images.

Layer (type)	Output Shape	Param #
dense_6 (Dense)	(None, 64)	25,152
dense_7 (Dense)	(None, 10)	650
Total params: 25,802 (100.79 KB)		
Trainable params: 25,802 (100.79 KB)		
Non-trainable params: 0 (0.00 B)		

Since the hog size is much smaller than what we had with the raw data, we decided to start with just 64 neurons for the hidden layer.

2. Model Weights

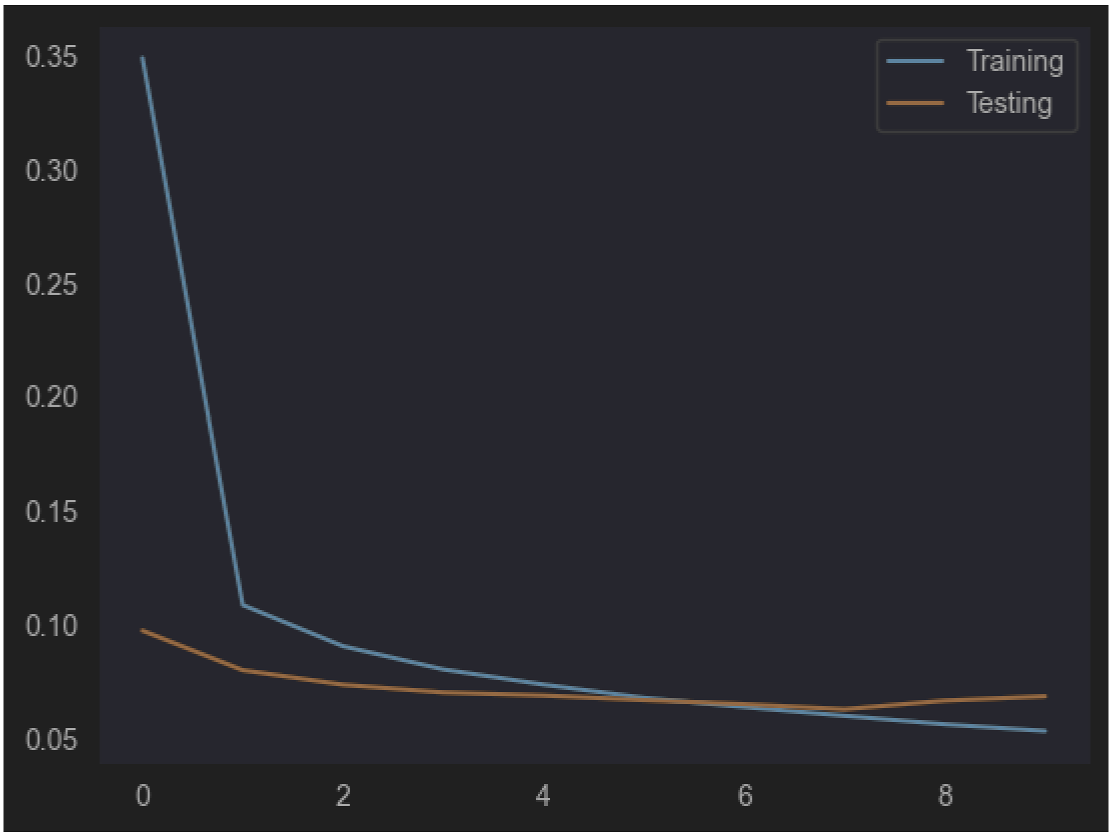
- Here are all the weights linked to this model :
  - Input to Hidden layer weights: 392 x 64 (hidden neurons) weights = 25’088 weights
  - Hidden layer biases: 64 biases
  - Hidden to Output layer weights: 64 (hidden neurons) x 10 (output neurons) = 640 weights
  - Output layer biases: 10 biases
  - Total weights: 25’088 + 64 + 640 + 10 = 25’802 weights

3. Test Cases

First pass

With our base settings and 10 epochs, here are the results :

Test score: 0.07196466624736786 Test accuracy: 0.977400004863739



and the confusion matrix :

	0	1	2	3	4	5	6	7	8	9
0	968	0	3	0	0	2	5	1	0	1
1	1	1126	2	1	0	0	2	1	2	0
2	2	2	1016	1	2	0	2	3	4	0
3	0	0	7	988	0	5	0	3	6	1
4	1	2	3	1	959	0	1	2	2	11
5	2	1	0	15	0	866	5	0	2	1
6	4	2	2	0	5	2	943	0	0	0
7	0	6	10	3	3	0	0	993	4	9
8	5	0	5	11	1	1	4	5	934	8
9	0	3	1	5	7	1	0	7	4	981

Discussion of the result :

With these base values, we're having slightly better results than the first pass of the MLP from raw data. What's notable though is that we're using substancely less layers and weight. After getting those results, we did try the raw data with the same layer composition and we're indeed getting worse result than with the hog. (Test score: 0.08612820506095886,Test accuracy: 0.9732999801635742) It's interesting to see that in this case, HOG is better than raw data while being more efficient !

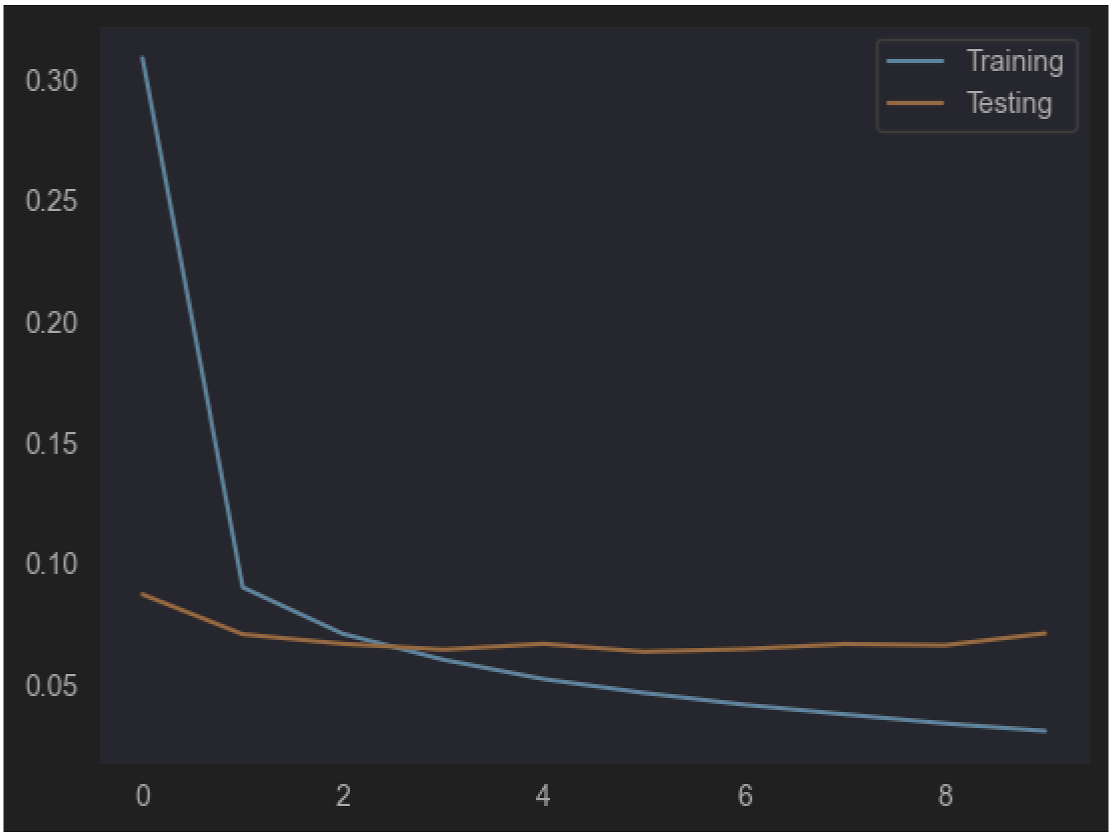
But there's a drawback, computing the HOG takes a lot of time and CPU.

Second pass

On this second pass, we're just going to try to change the HOG's parameters to see the changes they provide :

By doubling the number of orientations from 8 to 16 and changing nothing else we're getting these results :

Test score: 0.06866113841533661 Test accuracy: 0.9783999919891357



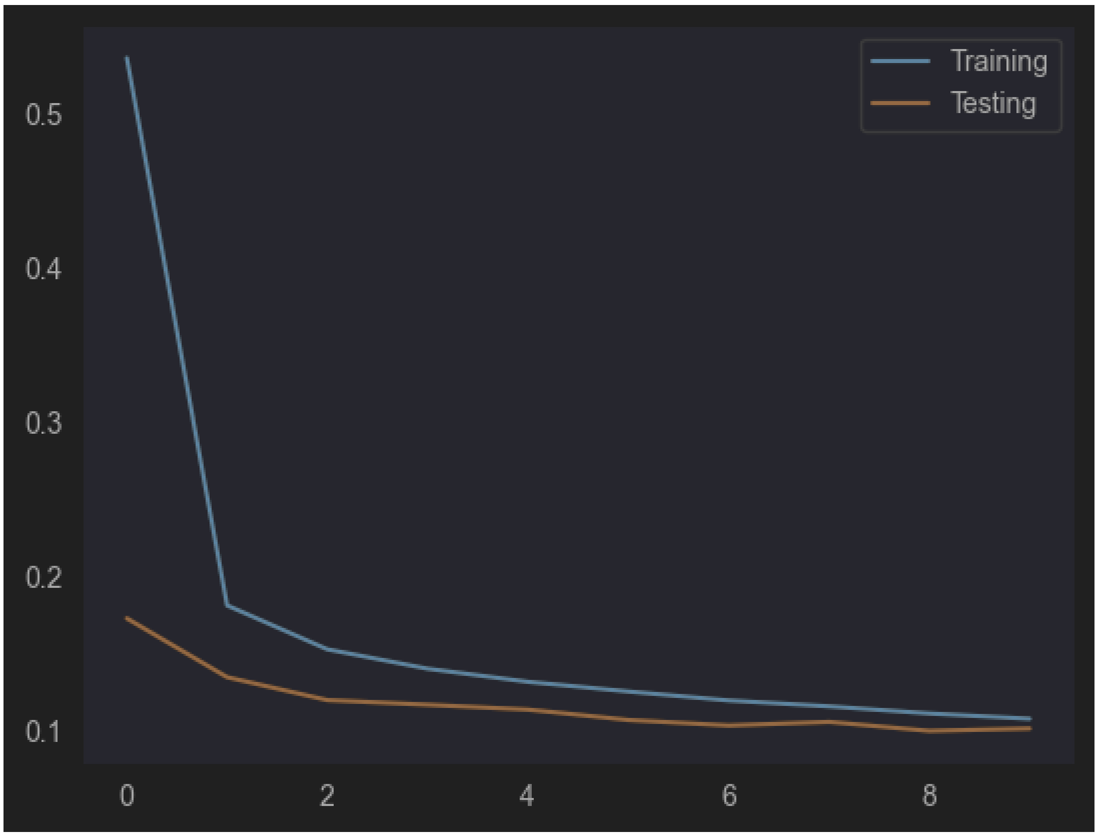
Confusion matrix :

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	966	0	2	0	1	2	5	1	2	1
1	1	1123	3	3	1	0	2	1	1	0
2	3	1	1013	2	1	0	2	5	5	0
3	0	1	5	984	0	5	0	4	11	0
4	0	1	1	0	973	0	1	0	3	3
5	1	1	1	13	0	867	4	1	4	0
6	2	3	1	1	6	2	943	0	0	0
7	0	3	10	3	6	0	0	1000	2	4
8	3	0	2	6	2	1	1	5	950	4
9	0	3	1	3	23	2	0	7	5	965

It's slightly better than with 8 !

Now we're going to change from 4 to 7 for the pix_p_cell, keeping the same amount of orientation (16) :

Test score: 0.11152961105108261 Test accuracy: 0.963699996471405



Confusion matrix :

	0	1	2	3	4	5	6	7	8	9
0	968	1	0	1	0	1	4	0	3	2
1	1	1114	3	4	2	0	6	2	3	0
2	3	2	995	12	4	1	1	7	7	0
3	1	1	6	967	0	17	0	5	10	3
4	0	2	3	0	933	0	5	8	4	27
5	1	0	0	7	1	874	3	0	6	0
6	7	2	0	0	4	9	931	0	3	2
7	0	3	17	1	5	1	0	975	3	23
8	2	0	7	15	4	12	6	2	915	11
9	0	4	2	6	8	4	0	10	10	965

Which is way worse...

Discussion of the result :

We see a slight improvement in the values from the first pass when we increase the number of rotations. These additional rotations help with the detection of the number 4.

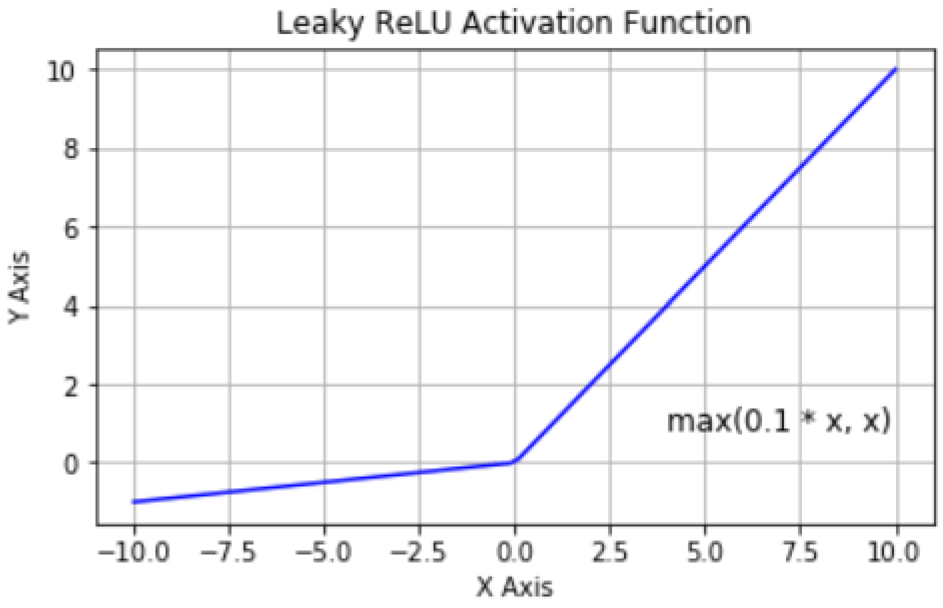
The number 8 is often mixed up with the other classes and espacially with the number 3.

We can also see that increasing the amount of pixel that are taken into account simplifies the model too much and gives us worse result when testing...

Third pass

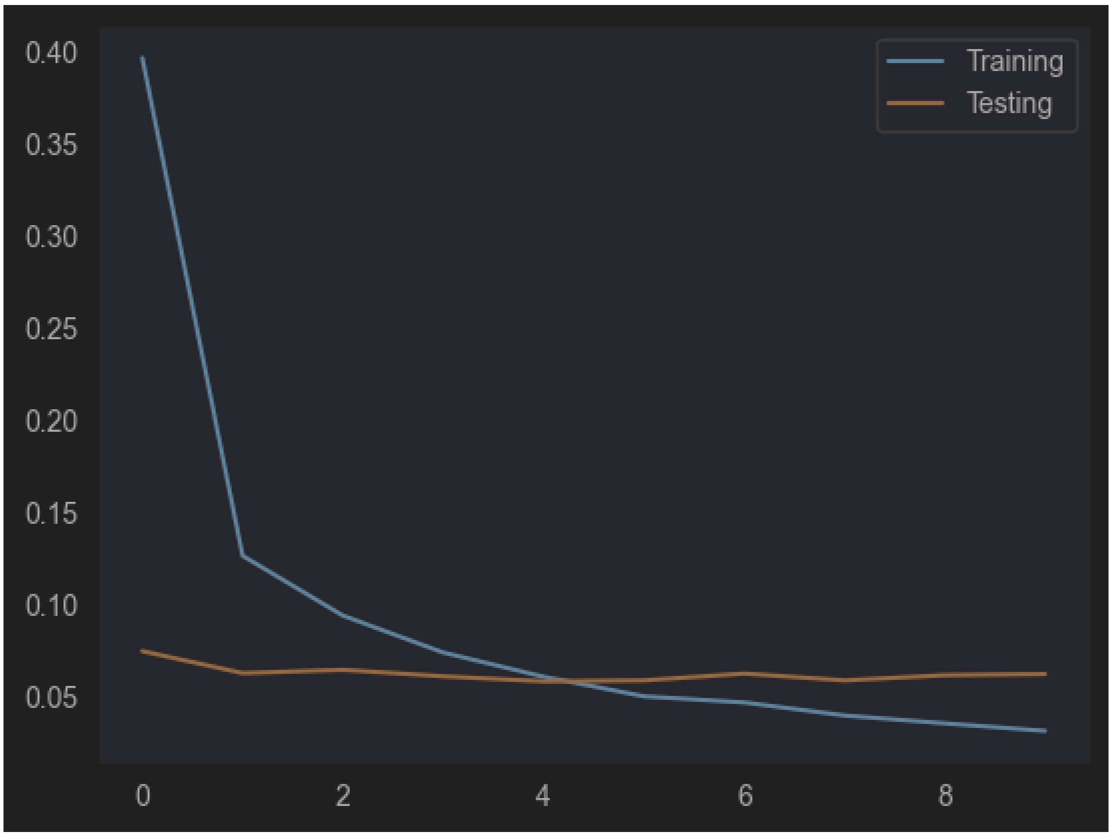
```
model = Sequential()
model.add(Input(shape=(hog_size,)))
model.add(Dense(256, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.3))
model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.3))
model.add(Dense(64, activation='leaky_relu'))
model.add(BatchNormalization())
model.add(Dropout(0.3))
model.add(Dense(32, activation='leaky_relu'))
model.add(BatchNormalization())
model.add(Dropout(0.3))
model.add(Dense(n_classes, activation='softmax'))
```

We used a Leaky ReLU activation function in the last two dense layers.



We also put some dropout layers between them as well as batch normalization layers.

Layer (type)	Output Shape	Param #
dense_60 (Dense)	(None, 256)	200,960
batch_normalization_12 (BatchNormalization)	(None, 256)	1,024
dropout_30 (Dropout)	(None, 256)	0
dense_61 (Dense)	(None, 128)	32,896
batch_normalization_13 (BatchNormalization)	(None, 128)	512
dropout_31 (Dropout)	(None, 128)	0
dense_62 (Dense)	(None, 64)	8,256
batch_normalization_14 (BatchNormalization)	(None, 64)	256
dropout_32 (Dropout)	(None, 64)	0
dense_63 (Dense)	(None, 32)	2,080
batch_normalization_15 (BatchNormalization)	(None, 32)	128
dropout_33 (Dropout)	(None, 32)	0
dense_64 (Dense)	(None, 10)	330
Total params: 246,442 (962.66 kB)		
Trainable params: 245,482 (958.91 kB)		
Non-trainable params: 960 (3.75 kB)		



Confusion matrix :

	0	1	2	3	4	5	6	7	8	9
0	976	0	0	0	1	1	1	1	0	0
1	0	1129	2	2	0	0	1	1	0	0
2	1	1	1021	1	2	0	0	4	2	0
3	0	1	1	1000	0	4	0	2	2	0
4	1	0	0	0	968	0	2	3	0	8
5	1	0	1	9	0	875	2	1	3	0
6	5	3	0	0	3	3	943	0	1	0
7	0	2	6	2	5	0	0	1007	2	4
8	5	0	2	7	1	1	0	2	951	5
9	0	4	0	2	5	1	0	10	4	983

Discussion of the result :

After mix matching what we used in the raw data testing and trying out different activations, we arrived to this pretty good results.

We just wonder why the testing starts so low ?

Discussion of digit recognition from features of the input data

The best model we have is the last one (third pass) where we used what we discovered during the testing on the first experiment. We have pretty good results across all the classes and they all are recognized at pretty much the same rate.

Convolutional neural network digit recognition

Based on the provided notebook, we can address the two requested items as follows:

1. Neural Network Topology

The neural network topology used in this notebook is a Convolutional Neural Network (CNN).

- The network architecture consists of the following layers:
  - Input Layer: Accepts the 28x28x1 grayscale image tensor.
  - Convolutional Layer 1 (l1): 32 filters of size 2x2 with ReLU activation.
  - Max Pooling Layer 1 (l1_mp): 2x2 max pooling.
  - Convolutional Layer 2 (l2): 32 filters of size 2x2 with ReLU activation.
  - Max Pooling Layer 2 (l2_mp): 2x2 max pooling.
  - Convolutional Layer 3 (l3): 32 filters of size 2x2 with ReLU activation.
  - Max Pooling Layer 3 (l3_mp): 2x2 max pooling.
  - Flatten Layer (flat): Flattens the output of the convolutional layers.
  - Dense Layer (l4): 32 units with ReLU activation.
  - Output Layer (l5): 10 units with softmax activation (one for each digit class).

2. Model Weights

- Here are all the weights linked to this model :

1. Convolutional Layer 1 (l1):
   - Filters: 32 (output channels)
   - Filter size: 2x2 (height x width)
   - Input channels: 1 (grayscale image)
   - Weights: (2 x 2 x 1 + 1) x 32 = 160 (filter weights + biases)
2. Convolutional Layer 2 (l2):
   - Filters: 32 (output channels)
   - Filter size: 2x2 (height x width)
   - Input channels: 32 (from previous layer)
   - Weights: (2 x 2 x 32 + 1) x 32 = 4'128 (filter weights + biases)
3. Convolutional Layer 3 (l3):
   - Filters: 32 (output channels)
   - Filter size: 2x2 (height x width)
   - Input channels: 32 (from previous layer)
   - Weights: (2 x 2 x 32 + 1) x 32 = 4'128 (filter weights + biases)
4. Dense Layer (l4):
   - Units: 32

- Input size: 3x3x32 = 288
- Weights: 288 x 32 + 32 = 9'248 (weights + biases)

5. Output Layer (l5):

- Units: 10 (one for each digit class)
- Input size: 32 (from the previous dense layer)
- Weights: 32 x 10 + 10 = 330 (weights + biases)

Total number of weights in the model: 160 + 4'128 + 4'128 + 9'248 + 330 = 17'994 weights

Layer (type)	Output Shape	Param #
l0 (InputLayer)	(None, 28, 28, 1)	0
l1 (Conv2D)	(None, 28, 28, 32)	160
l1_mp (MaxPooling2D)	(None, 14, 14, 32)	0
l2 (Conv2D)	(None, 14, 14, 32)	4,128
l2_mp (MaxPooling2D)	(None, 7, 7, 32)	0
l3 (Conv2D)	(None, 7, 7, 32)	4,128
l3_mp (MaxPooling2D)	(None, 3, 3, 32)	0
flat (Flatten)	(None, 288)	0
l4 (Dense)	(None, 32)	9,248
l5 (Dense)	(None, 10)	330
Total params: 17,994 (70.29 KB)		
Trainable params: 17,994 (70.29 KB)		
Non-trainable params: 0 (0.00 B)		

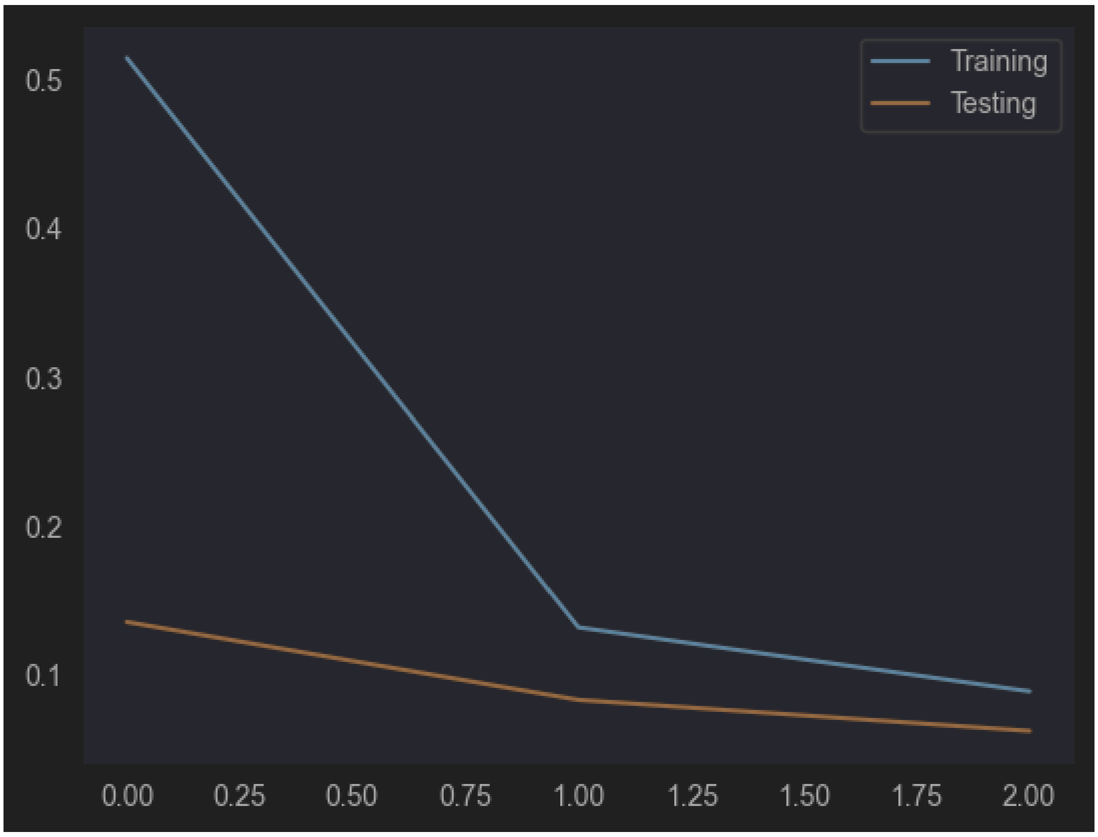
We choose the initial values to be 32 instead of 2 since it was giving us horrendous results.

3. Test Cases

First pass

With the model presented just earlier, we got those results :

Test score: 0.058950554579496384 Test accuracy: 0.9803000092506409



Confusion matrix :

	0	1	2	3	4	5	6	7	8	9
0	971	0	3	0	0	0	2	2	2	0
1	0	1132	1	0	0	0	0	2	0	0
2	1	1	1011	2	1	0	0	13	3	0
3	0	0	3	989	0	4	0	8	3	3
4	0	0	1	0	970	0	0	4	0	7
5	1	0	1	11	0	867	2	1	7	2
6	8	3	0	2	3	3	938	0	1	0
7	0	2	4	0	0	0	0	1020	2	0
8	5	3	7	3	3	3	1	9	933	7
9	2	6	0	4	5	2	0	15	3	972

Discussion of the result :

We have pretty good results for a first try.

The number 7 gets frequently mixed up with other classes (54 confusions!) and especially with the number 9 and 2 (respectively 15 and 13 confusions).

We aso only have 5 confusions for the number 6 which is low.

Second pass

On this second pass, we tried to implement the pyramidal sructure we used in the MLP notebooks. After going for the compacting route and seeing how slow it was to do each epoch (around 50 seconds per epochs), we looked online and saw that it may be better to start with lower values and then build up to bigger ones.

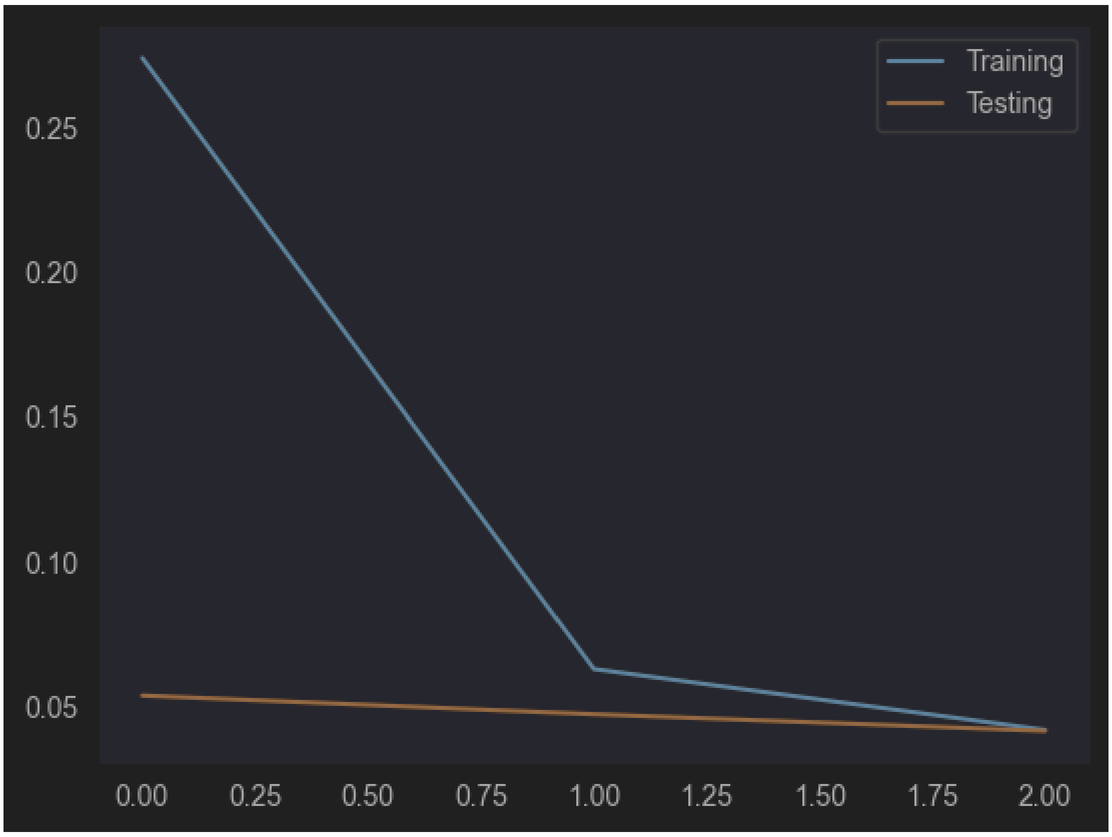
It's better to start with fewer filters and gradually increase the number of filters in deeper layers. This approach allows the network to learn low-level features first and then build upon them to learn more complex features.

```
l0 = Input(shape=(height, width, 1), name='l0')

l1 = Conv2D(32, (2, 2), padding='same', activation='relu', name='l1')(l0)
l1_mp = MaxPooling2D(pool_size=(2, 2), name='l1_mp')(l1)

l2 = Conv2D(64, (2, 2), padding='same', activation='relu', name='l2')(l1_mp)
l2_mp = MaxPooling2D(pool_size=(2, 2), name='l2_mp')(l2)

l3 = Conv2D(128, (2, 2), padding='same', activation='relu', name='l3')(l2_mp)
l3_mp = MaxPooling2D(pool_size=(2, 2), name='l3_mp')(l3)

flat = Flatten(name='flat')(l3_mp)

l4 = Dense(256, activation='relu', name='l4')(flat)
l5 = Dense(n_classes, activation='softmax', name='l5')(l4)

model = Model(inputs=l0, outputs=l5)
```

Layer (type)	Output Shape	Param #
l0 (InputLayer)	(None, 28, 28, 1)	0
l1 (Conv2D)	(None, 28, 28, 32)	160
l1_mp (MaxPooling2D)	(None, 14, 14, 32)	0
l2 (Conv2D)	(None, 14, 14, 64)	8,256
l2_mp (MaxPooling2D)	(None, 7, 7, 64)	0
l3 (Conv2D)	(None, 7, 7, 128)	32,896
l3_mp (MaxPooling2D)	(None, 3, 3, 128)	0
flat (Flatten)	(None, 1152)	0
l4 (Dense)	(None, 256)	295,168
l5 (Dense)	(None, 10)	2,570
Total params: 339,050 (1.29 MB)		
Trainable params: 339,050 (1.29 MB)		
Non-trainable params: 0 (0.00 B)		

It made each epochs 5 times faster and yielded similar results :

Test score: 0.04089416190981865 Test accuracy: 0.9872999787330627



Confusion matrix :

	0	1	2	3	4	5	6	7	8	9
0	974	0	0	0	1	0	1	1	1	2
1	0	1134	0	0	0	0	0	1	0	0
2	0	1	1023	0	0	0	0	7	1	0
3	0	0	1	1007	0	0	0	0	0	2
4	0	0	0	0	976	0	0	0	1	5
5	1	1	0	12	0	872	1	1	0	4
6	7	6	0	0	9	6	925	0	4	1
7	0	3	3	0	0	0	0	1019	1	2
8	3	1	2	3	2	3	0	4	945	11
9	0	0	0	1	5	0	0	3	2	998

Discussion of the result :

Now we're getting really good results, it's the best we've gotten from all the notebooks.

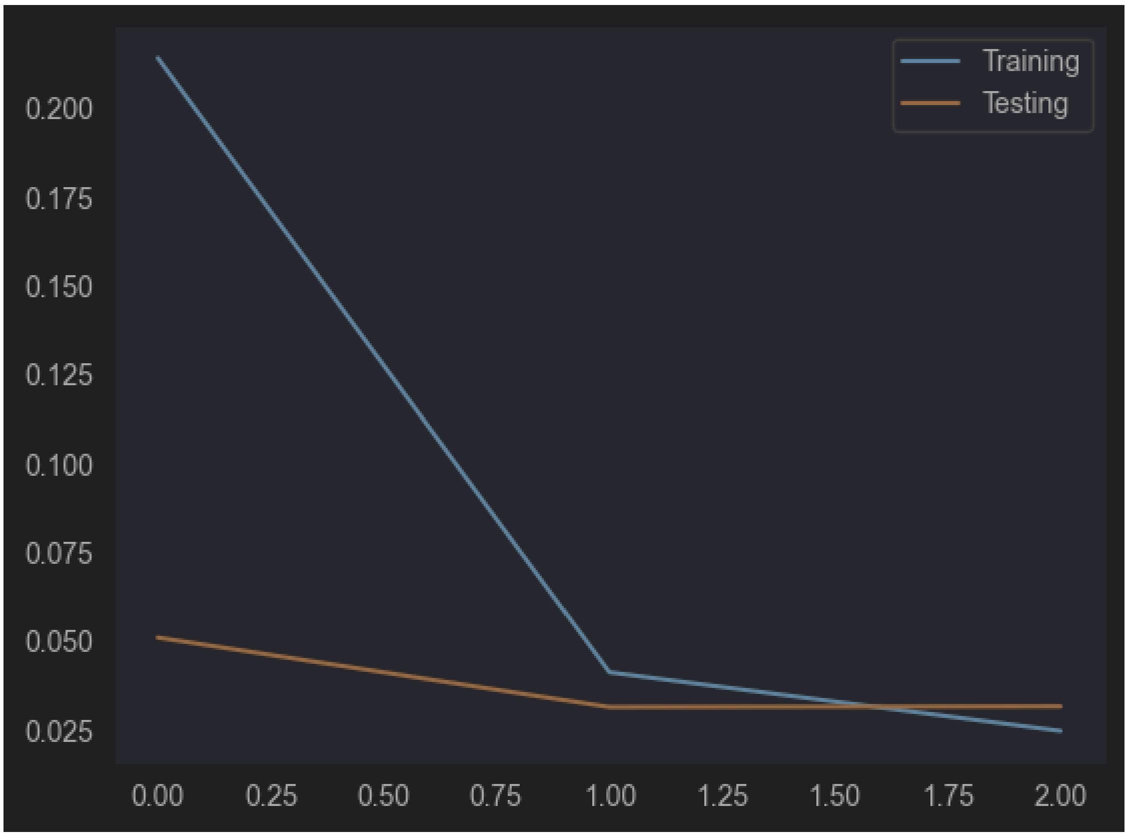
The highest number of confusion is for the number 9 (27) and only have 2 confusions for the number 6 !

Third pass

The last thing we're using to try to improve our model is changing the filter's size. Originally it was at 2,2 but we've tried with 3,5,7 and 9 to see which size would yield the best result. Each augmentation of the size meant an increase in processing time for each epoch as the amount of weight would increase exponentially, (9,9) as the size would give us 1'129'994 total parameters...

After doing lots of test, the best size was (7,7), giving us those results :

Test score: 0.026562191545963287 Test accuracy: 0.9919000267982483



Confusion matrix :

	0	1	2	3	4	5	6	7	8	9
0	972	0	0	0	0	0	5	1	2	0
1	0	1134	0	1	0	0	0	0	0	0
2	0	1	1019	3	0	0	0	7	2	0
3	0	0	0	1000	0	8	0	0	2	0
4	0	0	0	0	977	0	1	0	0	4
5	1	0	0	2	0	888	1	0	0	0
6	1	3	0	0	2	1	949	0	2	0
7	0	4	2	2	1	0	0	1014	1	4
8	0	0	1	0	0	1	0	0	972	0
9	0	0	0	1	3	4	0	3	4	994

Discussion of the result :

Very good results from this model.

We have a really good recognition for the number 0 and 2 and while the number 6 gets confused more often than in the second pass, the number of confusions globally went down.

The most confused number in this model is the number 5 with only 14 confusions !

Discussion of digit recognition from features of the input data :

The best model we have is the last one. We could reach better results with CNN compared to the MLP models we used.

3. CNN Models

The CNNs models are deeper (have more layers), do they have more weights than the shallow ones? explain with one example.

Just like with MLP, if a model has more layers, it will induce more weights.

We can just look at our own CNN model to see that more layers = more weights. In the original model we had this :

1. Convolutional Layer 1 (l1):
   - Filters: 32 (output channels)
   - Filter size: 2x2 (height x width)
   - Input channels: 1 (grayscale image)
   - Weights: (2 x 2 x 1 + 1) x 32 = 160 (filter weights + biases)
2. Convolutional Layer 2 (l2):
   - Filters: 32 (output channels)
   - Filter size: 2x2 (height x width)
   - Input channels: 32 (from previous layer)
   - Weights: (2 x 2 x 32 + 1) x 32 = 4'128 (filter weights + biases)
3. Convolutional Layer 3 (l3):
   - Filters: 32 (output channels)
   - Filter size: 2x2 (height x width)
   - Input channels: 32 (from previous layer)
   - Weights: (2 x 2 x 32 + 1) x 32 = 4'128 (filter weights + biases)
4. Dense Layer (l4):
   - Units: 32
   - Input size: 3x3x32 = 288
   - Weights: 288 x 32 + 32 = 9'248 (weights + biases)
5. Output Layer (l5):
   - Units: 10 (one for each digit class)
   - Input size: 32 (from the previous dense layer)
   - Weights: 32 x 10 + 10 = 330 (weights + biases)

Total number of weights in the model: 160 + 4'128 + 4'128 + 9'248 + 330 = 17'994 weights

If we remove the convolutional Layer 3, we would remove 4'128 weights from this model. Therefore it proves that more layers = more weights.

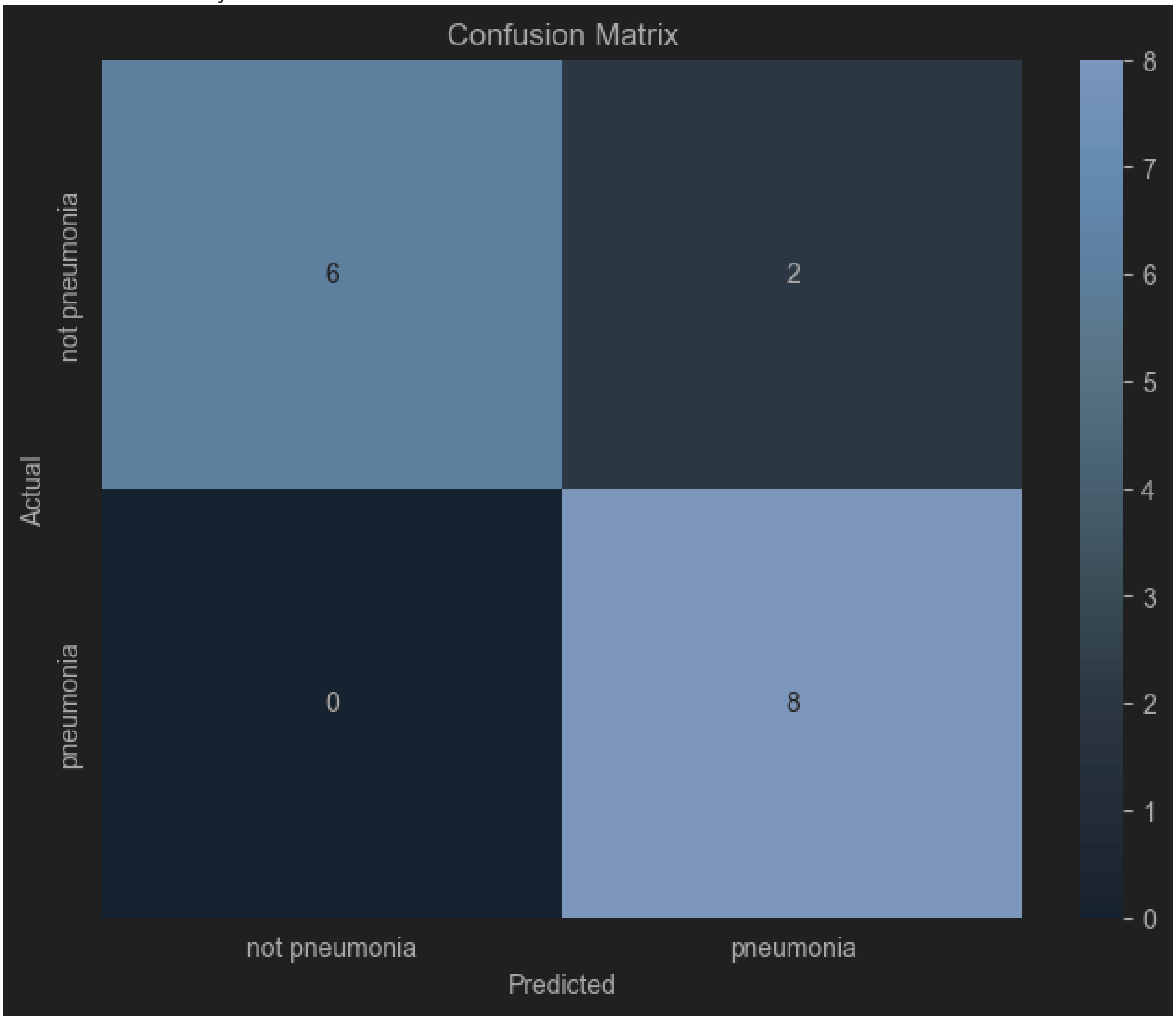
4. Chest X-ray Pneumonia Recognition

Train a CNN for the chest x-ray pneumonia recognition.

Results :

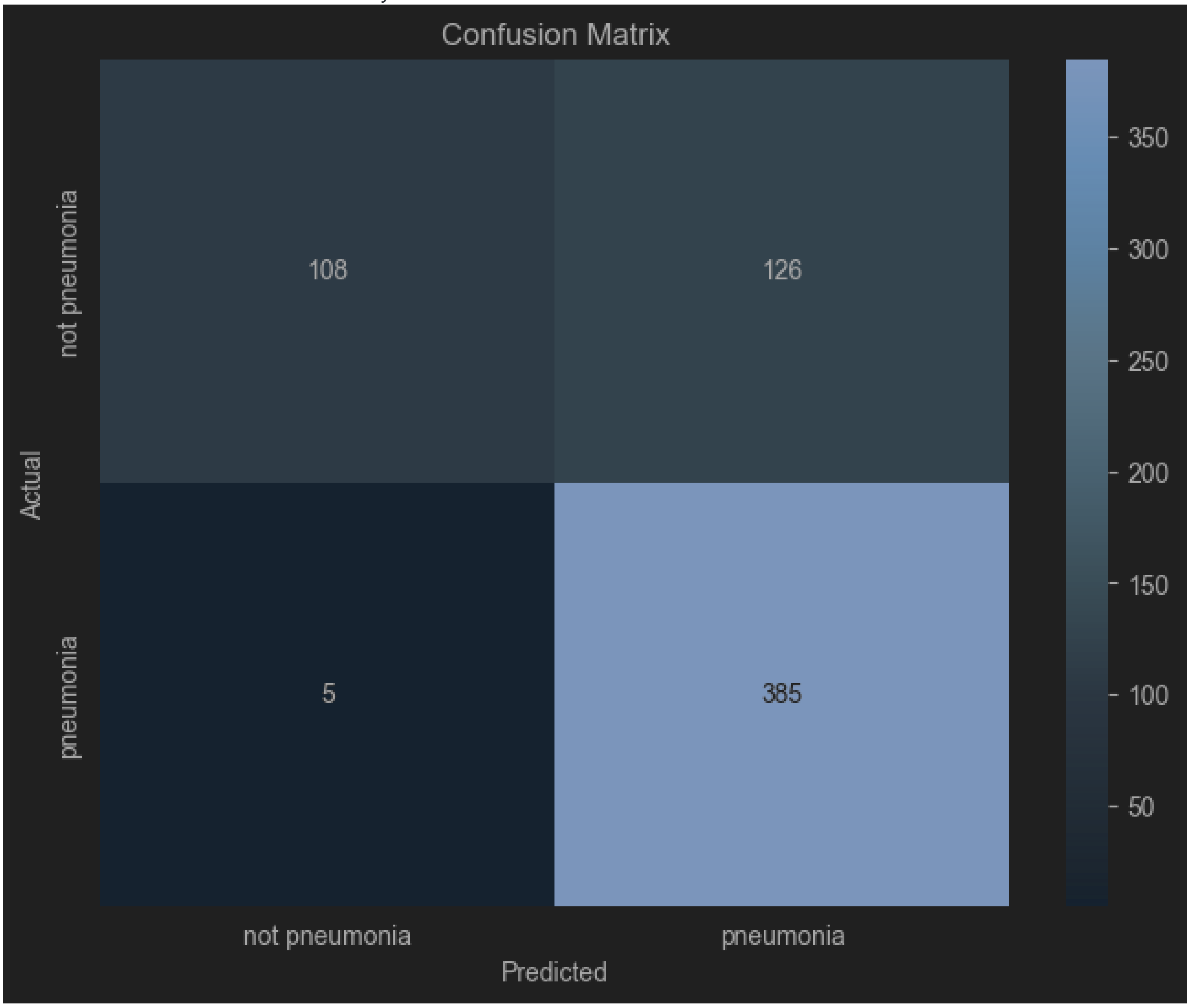
Validation :

f1 score : 0.89 accuracy : 0.875



Test :

```
f1 score : 0.8546059933407325 accuracy : 0.7900641025641025
```



Discussion :

We have false positive in both the validation and the test results and we prefer to have a false positive rather than a false negative.

Sadly on 624 test, we have 5 false negatives, which means that our model could be dangerous for 0.801% of the people we analyze. So if we use this on the population of Switzerland (8'776 millions), ~70'320 peoples could have a dangerous result.

For a real application this percentage would be too high...