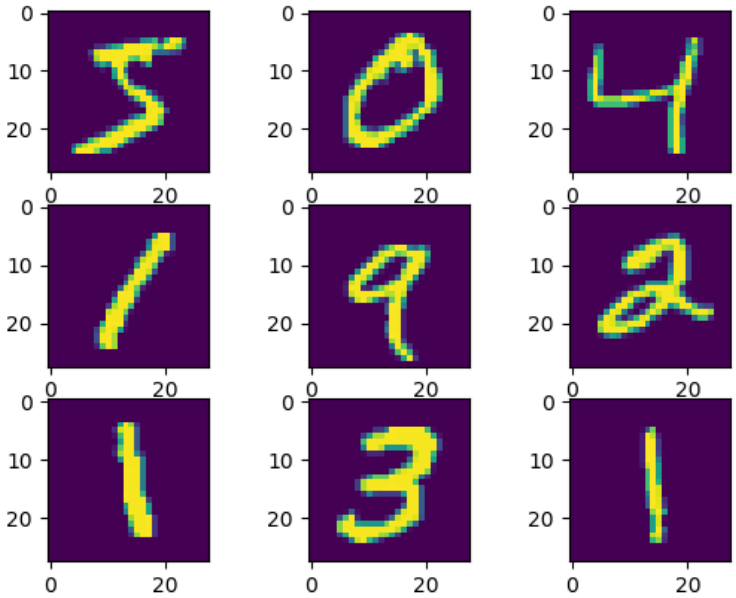
Intelligent Systems

Recognizing Digits using Neural Networks

ABOUELOULA Ayman BENSRHIER Nabil ZRIRA Ayoub

April 02, 2020



Contents

Chapter 1

Introduction

Digit and handwriting recognition is one of the most difficult problems of pattern recognition and artificial intelligence, for example:

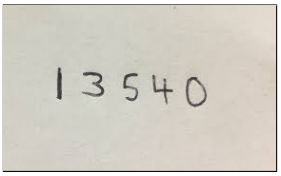


Figure 1.1: **example**

Most people recognize those digits as 13540 because our brain contains millions of neurons, with billions of connections between them, is like we carry a computer in our head trained by over millions of years.
there are two solutions to deal with the problem:

- trying to write a program that describes the form of each digit, but the exception numbers and special cases are very large which makes the algorithmic solution difficult if not impossible to implement
- Using Neural networks: The idea is to take a large number of handwritten digits, and try to develop a system which can learn from those examples.

In the next chapters we are going to design and evaluate a neural network architectures that can recognize hand-drawn digits using the gray-scale MNIST images, ANN, and CNN
The program in the next chapter can recognize digits with an accuracy over 95 percent. Furthermore, in later chapters we'll develop ideas which can improve accuracy to over 99 percent using CNN.
The next chapters will also develop many ideas about neural networks,like the stochastic gradient descent. and we will focus on justifying our choices especially in choosing some specific parameters like number of epochs, batch size, number of layers, non-linear activation functions, loss functions, types of pooling, spatial-extent (size) and number of filters

Chapter 2

Artificial Neural Network

2.1 Steps to build an ANN

- Step1: Randomly initialise the weights to small numbers close to 0(but not 0)
- Step2: Input the first observation of your dataset in the input layer, each feature in one input node
- step3: Forward-propagation: from the left to the right, the neurons are activated in a way that the impact of each neuron's activation is limited by the weigths. Propagate the activations until getting the predicted result y.
- step4: compare the result to the actual one and measure the error
- step5: Back-prpagation: from the right to the left to update the weights according to how much they are responsible for the error.
- step6: repeat steps 1 to 5 and update the weights after each observation(reinforcement learning). Or after a batch of observations(Batch learning).
- step7: when the whole training set passed through the ANN, that makes an epoch. Make more epochs

2.2 Building an ANN

2.2.1 importing libraries and packages

```
import numpy as np
import mnist
import matplotlib.pyplot as plt
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten
from keras.utils import to_categorical
from sklearn.metrics import classification_report
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
```

2.2.2 Importing the MNIST Dataset

The dataset we will be using in this Project is called the MNIST dataset. This dataset is made up of images of handwritten digits, 28x28 pixels in size.

```
#ANN model
#importing data
train_images = mnist.train_images()
train_y = mnist.train_labels()
test_images = mnist.test_images()
test_y = mnist.test_labels()
#normalizing data
train_x = (train_images/255) - 0.5
test_x = (test_images/255) - 0.5
train_x = train_x.reshape((-1, 784)) # 28*28 = 784
test_x = test_x.reshape((-1, 784))
```

2.2.3 Defining the Neural Network Architecture

The architecture of the neural network refers to elements such as the number of layers in the network, the number of units in each layer and the activation function used.
we initialized our artificial Artificial neural network by defining it as a sequence of layers, and we added 2 hidden layers and an output layer to our ANN.
Then in the forward propagation step, the neurons are activated by the activation function, in such a way that the higher the value of the activation function is for the neuron,the more impact this neuron is going to have in the network.
there are several activation function and the best one based on experiment and on research is the rectifier function(relu) for the hidden layers and softmax for the output layer
The question now is how to choose the number of nodes in the hidden layers?usually we choose the average of the number of nodes in the input layer and the number of nodes in the output layer, and this rule is based on experiment and not on theory :

$$(784 + 10)/2 = 397$$

```
#initialising the ANN
model= tf.keras.Sequential()
#adding the input layer and the first hidden layer
model.add( tf.keras.layers.Dense(397, activation = 'relu', input_dim = 784))
#adding the second hidden layer
model.add(tf.keras.layers.Dense(397, activation = 'relu'))
#adding the output layer
model.add(tf.keras.layers.Dense(10, activation = 'softmax'))
```

2.2.4 compiling the ANN

After adding all the layers, we are going to compile the whole artificial neural network and that means applying stochastic gradient descent.
To do that we are going to use compile method that contains several parameters, like the optimizer which is the algorithm you want to use to find the optimal set of weights in the NN, and this algorithm is going to be nothing else than the stochastic gradient descent algorithm
A very efficient one is "ADAM" ,but in the following sections we will see how to choose these parameters in order to increase the accuracy.

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

2.2.5 Fitting the ANN to the training set

Until now we have only built the ANN without making the connection to the training set, so to make this connection we are going to use fit method
As explained in step 6 , the stochastic gradient descent algorithm, can choose update their weight either after each observation or after a batch of observations, so we choose batch-size = 32, and the training part consist to apply those steps over many epochs so we choose nb-epoch = 6.
we will see in the following sections how to choose these parameters in order to increase the accuracy.

```
model.fit(train_x, to_categorical(train_y), epochs = 6, batch_size = 32)
```

2.2.6 results

```
Epoch 1/6
1875/1875 [==============================] - 11s
    6ms/step - loss: 0.2660 - accuracy: 0.9175
Epoch 2/6
1875/1875 [==============================] - 10s
    5ms/step - loss: 0.1331 - accuracy: 0.9579
Epoch 3/6
1875/1875 [==============================] - 11s
    6ms/step - loss: 0.1020 - accuracy: 0.9679
Epoch 4/6
1875/1875 [==============================] - 10s
    5ms/step - loss: 0.0841 - accuracy: 0.9732
Epoch 5/6
1875/1875 [==============================] - 9s
    5ms/step - loss: 0.0691 - accuracy: 0.9783
Epoch 6/6
1875/1875 [==============================] - 9s
    5ms/step - loss: 0.0632 - accuracy: 0.9798
```

2.3 Evaluating and improving the ANN

In the previous section we split our data in two parts the training set and the test set and we trained our model and test it on them, that is a correct way of evaluating but not the best one.
So judging the model only by one accuracy on one model is not relevant, that is why we will use an other technique called K-fold Cross Validation.
this method split the training set into 10 folds, k =10, and we train our model on 9-folds and test it on the remaining one ,so we can make 10 different combinations to train and test our model

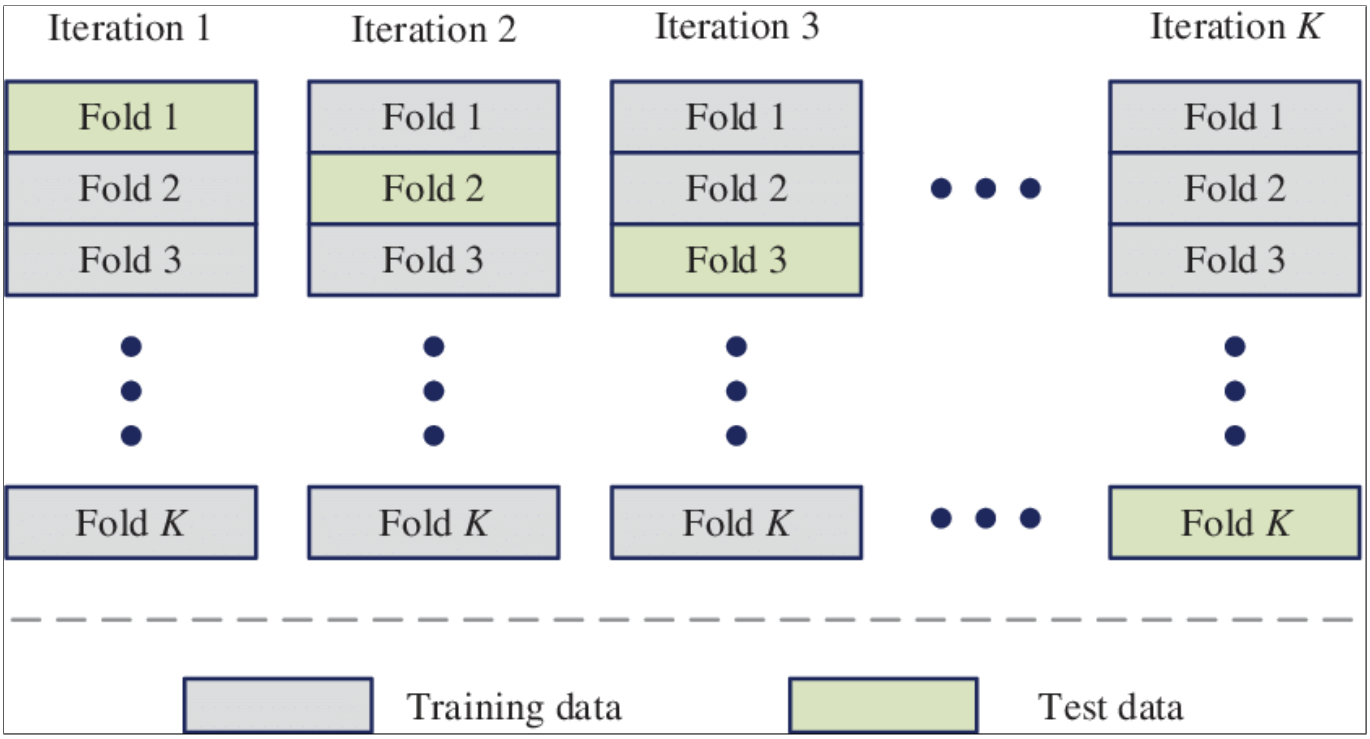


Figure 2.1: **K-fold cross-validation method**

```
def build_classifier():
  model= tf.keras.Sequential()
  model.add( tf.keras.layers.Dense(397, kernel_initializer = 'uniform', activation =
      'relu', input_dim = 784))
  model.add(tf.keras.layers.Dense(397, kernel_initializer = 'uniform', activation =
      'relu'))
  model.add(tf.keras.layers.Dense(10, kernel_initializer = 'uniform', activation =
      'softmax'))
  model.compile(optimizer='adam', loss = 'categorical_crossentropy', metrics =
      ['accuracy'])
  return model
classifier = KerasClassifier(build_fn=build_classifier, batch_size = 32, nb_epoch = 6)
accuracies = cross_val_score(estimator = classifier, X = train_x, y = train_y, cv = 10)
mean = accuracies.mean()
print(mean)
```

The relevant accuracy will be to take the mean of all the accuracies, which is 94.7 and that the relevance accuracy to evaluate our models, also the variance is very low 0.1
now we will try to improve these accuracy by making a better ANN

2.3.1 Improving the model

there is 2 types of parameters: the ones that are learnt from the model and these are the weights and hyper parameters such as epochs, batch size ...
so to improve the model we should give those parameters other values to augment the accuracy
That basically will test several combinations and return the best selection

```
#Evaluating and improving the model
def build_classifier(optimizer, nodes, layer):
  model= tf.keras.Sequential()
  model.add( tf.keras.layers.Dense(layer, kernel_initializer =
      'uniform', activation = 'relu', input_dim = 784))
  for i in range(nodes):
    model.add(tf.keras.layers.Dense(layer, kernel_initializer =
        'uniform', activation = 'relu'))
  model.add(tf.keras.layers.Dense(10, kernel_initializer = 'uniform',
      activation = 'softmax'))
  model.compile(optimizer=optimizer, loss = 'categorical_crossentropy',
      metrics = ['accuracy'])
  return model

model = KerasClassifier(build_fn=build_classifier)
parameters = {'batch_size':[32, 64],
              'nb_epoch':[6, 10], 'nodes' : [1, 2, 3],
              'layer' : [64, 397], 'optimizer' : ['adam', 'rmsprop']}
grid_search = GridSearchCV(estimator = model, param_grid = parameters,
                           scoring = 'accuracy', cv = 10)
grid_search = grid_search.fit(train_x, train_y)
best_parameters = grid_search.best_params_
best_acc = grid_search.best_score_
print(best_acc, best_parameters)
```

The best model is(program may take a while to run):

```
model= tf.keras.Sequential()
model.add( tf.keras.layers.Dense(397, activation = 'relu', input_dim =
    784))
model.add(tf.keras.layers.Dense(397, activation = 'relu'))
model.add(tf.keras.layers.Dense(10, activation = 'softmax'))
model.compile(optimizer='adam', loss = 'categorical_crossentropy',
    metrics = ['accuracy'])
model.fit(train_x, to_categorical(train_y), epochs = 10, batch_size =
    32)
```

2.3.2 ROC curve

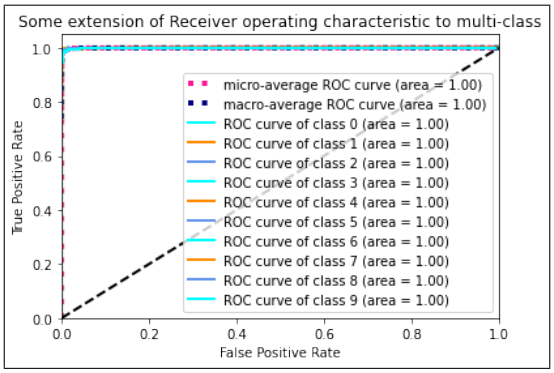


Figure 2.2: **ROC curves**

The ROC curves visualizes the quality of our model on a test set, and as we can see for each class, the ROC is near to the 1. This gives us a good indication of how good our model is at classifying digits.
We can say that we built a good model since it has good measure of separability.

Chapter 3

Convolutional Neural Network

Deep Learning is becoming a popular subfield of machine learning thanks to its high level of performance. The best way to use deep learning to classify images is to build a convolutional neural network using the Keras library in Python.
The source code that created this report can be found **here**.
Now let's get started!

3.1 Image Classification

Consider a 256 x 256 image. The previous ANN see all these pixels as independent variables, and we are asked to predict the class of the image using more parameters and having more data. As a result, there are a variety of problems and challenges associeted with that task.
The big idea behind CNN is that pixels in images are usually related, that means that a local understanding of an image is good enough to make the best prediction.
The benefit is the fact that having fewer nodes improves the time it takes to learn as well as reducing the amount of data.
For instance we get a 100 x 100 image instead of the first one.

So the steps that we're going to be going through with these images, before the input of our first ANN, are:

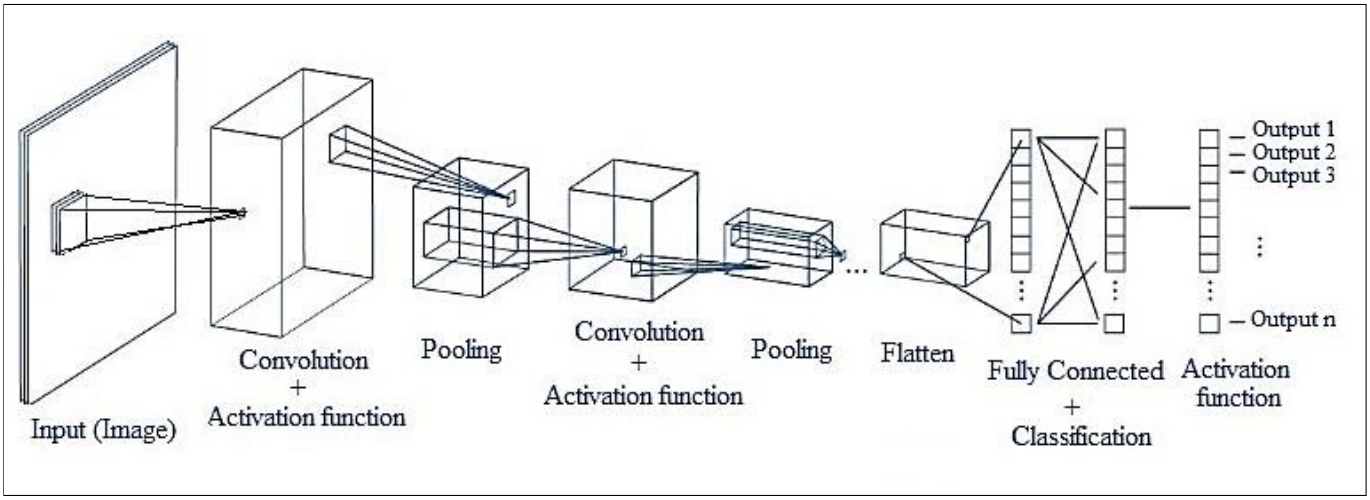


Figure 3.1: **Steps to build a Convolutional Neural Network**

3.1.1 Image Preprocessing

MNIST dataset is available in keras library: (**see code**)

```
import numpy as np
import matplotlib.pyplot as plt
from keras.utils.np_utils import to_categorical
from keras.datasets import mnist
```

We need to reshape the independent variables into 2D arrays.
So that all input features will be black and white pixels:

```
# Importing data
(train_images, train_y) ,(test_images, test_y) = mnist.load_data()

# Encoding dependent variable to categorical
train_y = to_categorical(train_y)
test_y = to_categorical(test_y)

# Scaling data
train_x = (train_images/255)
test_x = (test_images/255)

# Reshaping sets to [observations][width][height][image_version]
train_images = train_x.reshape(train_images.shape[0],28,28,1)
test_images = test_x.reshape(test_images.shape[0],28,28,1)

# Converting pixels from int to float
train_images = train_images.astype('float32')
test_images = test_images.astype('float32')
```

We use matplotlib library to plot a number randomly from the data set:

```
from random import randint
digit = randint(0, 60000)
plt.imshow(train_images[digit].reshape(28, 28), cmap=plt.get_cmap('gray'))
plt.show()
```

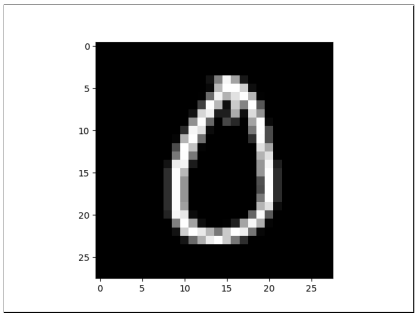


Figure 3.2: **Digit 0**

3.1.2 Convolution

The Convolution is an operation between the **input image** and a **feature detector**, and the result of this operation is the **feature map**.
In fact, the feature map contains the highest number that the feature detector could detect in the input image.

Do we lose information while applying the Feature Detector ?

Of course we are losing some information, because we have less values in our resulting matrix. But at the same time the purpose of the Feature Detector is to detect certain features, certain parts of the image that are integral.

Feature Detector

En general, we use (32, 64, .. 2^n) feature Detectors, which are **three-by-three matrix**, but other people as **Alex-Net** uses 7x7, even others use 5x5.

Activation function

For the ANN we used the activation function to activate the neurons in the neural network. Here we are using this activation function to make sure that we don't have any negative pixel values in our feature maps. So we use **the rectifier activation function** in order to have non-linearity in our convolutional neural network.

There is a paper written by **C.C.Jay Kuo**, from the University of California, he explains why a non-linear activation function is essential at the filter output of all intermediate layers.
Also **Kaiming He**, from Microsoft Research, proposed a different type of Rectified Linear Unit function, And he argued that it delivers better results without sacrificing performance.

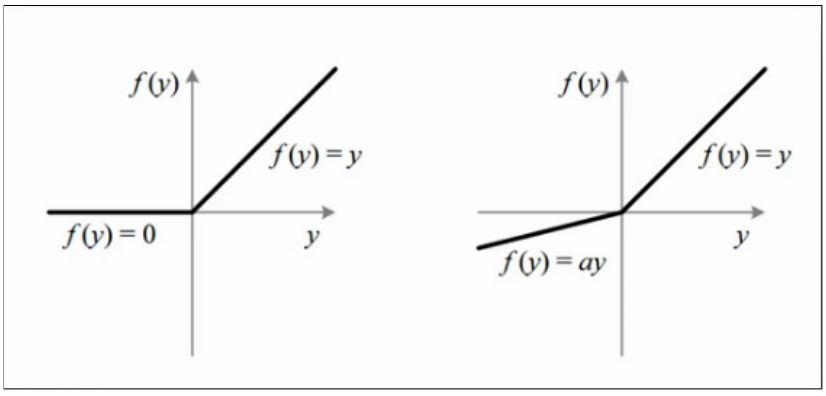


Figure 3.3: **the Rectifier activation functions**

3.1.3 Pooling

The function of this step is to progressively reduce the spatial size of the feature map to reduce the amount of computation and parameters in the network.
Pooling layer operates on each feature map independently. And the most common approach used in pooling is **max pooling** with a size of 2x2 which is a common choice.

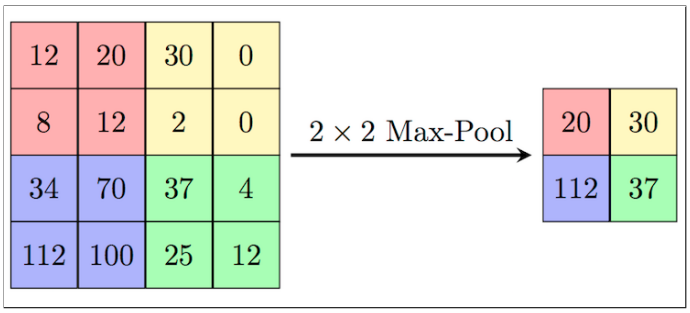


Figure 3.4: **Max-Pooling**

3.1.4 Flattening

After we apply the convolution to the input image and then we apply Pooling to the feature map, we have to flatten the pooled map into a column. Basically, we take the numbers row by row, and put them in one long column. The reason for that is because we want to later input this into an artificial neural network for further processing.
So, this is what it looks like :

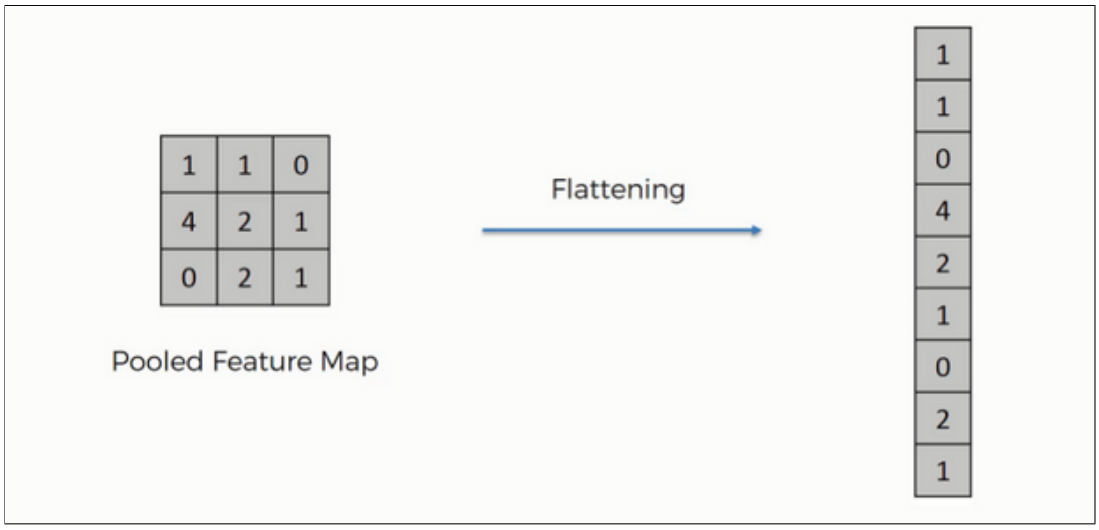


Figure 3.5: **Flattening**

3.1.5 Full Connection

In this step we are going to add a whole artificial neural network to our convolutional neural network.

First CNN:

Layer (type)	Output Shape	Param #
conv2d_9 (Conv2D)	(None, 24, 24, 32)	832
average_pooling2d_2 (Average	(None, 12, 12, 32)	0
flatten_4 (Flatten)	(None, 4608)	0
dense_9 (Dense)	(None, 20)	92180
dense_10 (Dense)	(None, 10)	210

Total params: 93,222
Trainable params: 93,222
Non-trainable params: 0

Figure 3.6: **Architecture of our first model**

We train this model using the **fit** function of Keras.
Then we evaluate it and we get an accuracy of :

98.62%

So this accuracy is a first sign that this architecture is better than a simple ANN.

Now the next step is to optimize this simple model and make it powerful by exploring the effects of variations of hyper-parameters.

3.2 Optimization

There are so many ways to optimize the CNN architecture. So how can we do that ?
Since we have a lot of parameters the **GridSearchCV** function will take a lot of time in order to find the best combination of parameters.
As a result, the best way is the simplest, we will find the most accurate CNN architecture by running some experiments.(**see code**)

3.2.1 Spatial-extent

The spatial-extent is the size of the feature detector or the kernel (filter) size.
Now let's see whether 3x3, 5x5, or 7x7 is the best. (**see code**)

```
for i in [3, 5, 7]:
        classifier = Sequential()
        classifier.add(Conv2D(32, (i, i), input_shape = (28, 28, 1)))
        classifier.add(Activation('relu'))
                                    ...
                                    ...
        classifier.compile(optimizer = 'adam', loss = 'categorical_crossentropy',
            metrics = ['accuracy'])
```

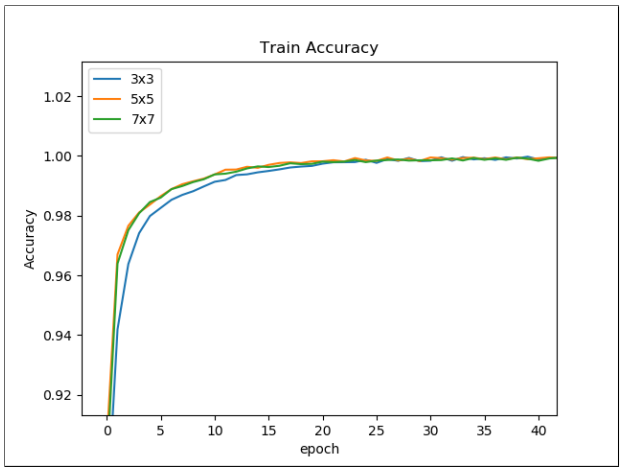


Figure 3.7: **Variations of the kernel size**

Results :

From the curve beside, it seems that 5x5 and 7x7 convolution are slightly better than 3x3 at the first epochs, but from a certain rank of epochs the accuracy will be the same as other choices. However for efficiency, convolutional layers work better because they are lighter, so instead of 5x5 or 7x7 filter we can stack 2 small 3x3 feature detector.

Interpretaion :

The best option is the 3x3 filter, may be because convolution is an interpolation from the given pixels to a center pixel. So we cannot interpolate to a center pixel using an other sized filter.

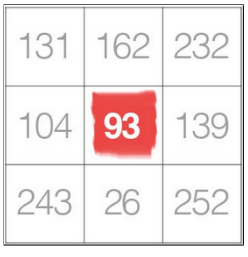


Figure 3.8: **3x3 filter is the popular choice**

3.2.2 Variations in learning rate

Deep learning classifiers are trained by a stochastic gradient descent optimizer. There are many variations of optimizer such as Adam, RMSProp, Adagrad, etc.
The learning rate is an optimizer parameter that describes how far to adjust the weights in the direction opposite of the gradient.
In order to find the best learning rates, we will train our model with 6 different optimizers, and each optimizer will be trained with learning rates from 0.000001 to 100.(**see code**)

```
for classifier in classifiers:
    lr_finder = LRFinder(classifier)
    lr_finder.find(train_x, train_y, start_lr=0.0000001, end_lr=100,
                   batch_size=512, epochs=50)
    lr_finder.plot_loss(n_skip_beginning=20, n_skip_end=5)
```

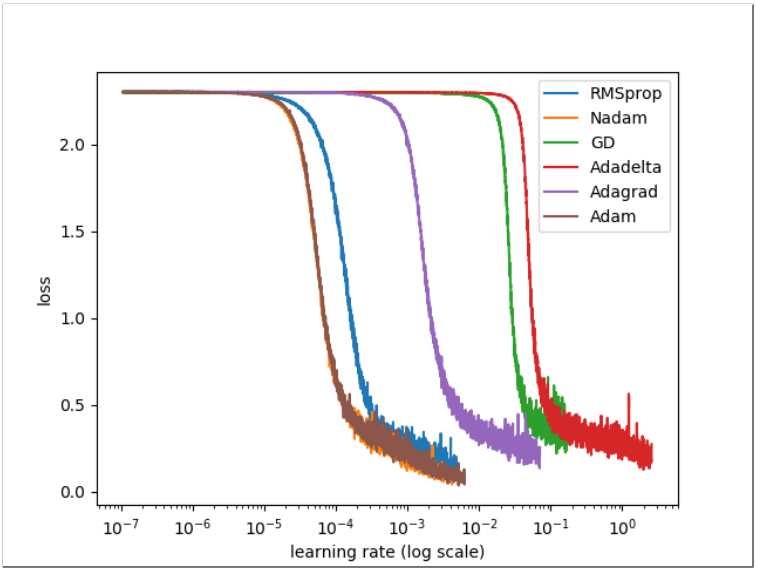


Figure 3.9: **The learning rates for each optimizer**

Comment :

The loss improves slowly with low learning rates, then training accelerates until the learning rate becomes too large and loss reaches a minimum, then the training process diverges.
So, we must select the minimum point on the graph with the fastest decrease for each optimizer.

Now after identifying the best learning rates for each optimizer, let's compare their performance training with the best learning rate found.(**see code**)

```
for opt in [RMSprop(learning_rate=0.01), Nadam(learning_rate=0.01),
            SGD(learning_rate=0.1), Adadelta(learning_rate=1),
            Adagrad(learning_rate=0.1), Adam(learning_rate=0.01)]:
                                ...
    classifier.compile(optimizer = opt, loss = 'categorical_crossentropy',
                       metrics = ['accuracy'])
```

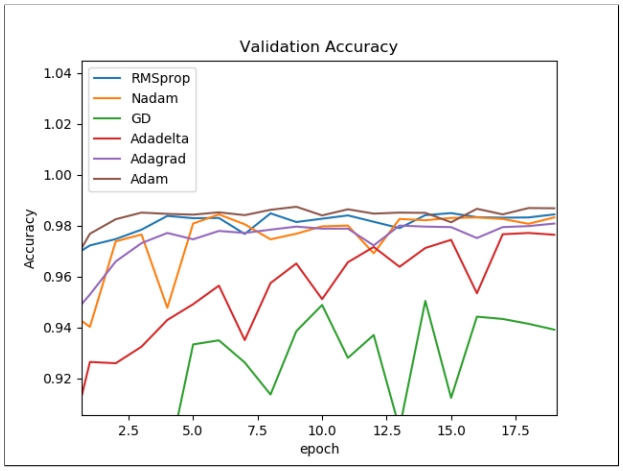


Figure 3.10: **Validation accuracy over epochs**

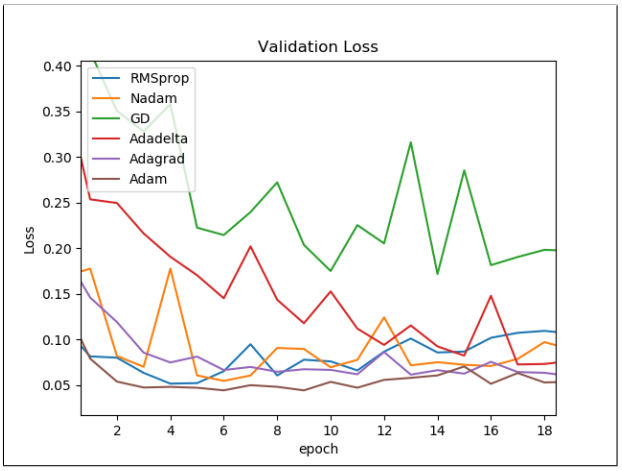


Figure 3.11: **Validation loss over epochs**

Comment

As we notice **Adam (Learning rate = 0.01)** learns the fastest and it is more stable than the others, also it doesn't suffer any major decreases in accuracy.

Adam is the best choice for this model of the six optimizers.

3.2.3 Loss functions

Our problem is a multi-class classification, that means we are predicting more than two classes, each class is related to a unique integer value from 0 to 9.
The problem is implemented as predicting the probability of belonging to each known class.
So the loss functions, that are appropriate for these predictive modeling problems:

1) categorical_crossentropy : it calculates a minimized value that summarizes the average difference between the real and predicted probability distributions for the 10 classes in the problem.
To ensure that each image has an expected probability of 1 or 0 for the actual class value, the dependent variable must **be one hot encoded**, and this can be done using **to_categorical()** Keras function.

2) sparse_categorical_crossentropy : it is used in classification problems with a large number of labels by performing the same as cross-entropy, without requiring that the target variable be one hot encoded.

3) kullback_leibler_divergence : its behavior is very similar to cross-entropy. It computes how much information is lost **in terms of bits** . So it requires that the target variable must be one hot encoded.

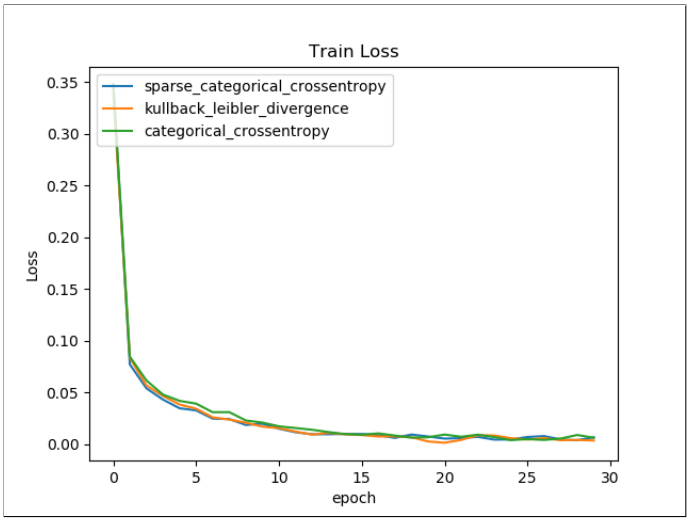


Figure 3.12: **Train loss of each loss function**(see code)

So we will choose **cross-entropy** as a loss function for the coming experiments of this model since the performance of each loss function is similar to the results seen with cross-entropy.

3.2.4 Batch Size

A batch size of **N** means that N samples from the training set will be used to estimate the gradient error before the model weights are updated.
One epoch means that the learning has made one pass through the training data, where images were separated into randomly selected **N** groups.
So what is the best value of N for our model ?
There is no general rule to respond to this question, but we can iterate with different batch size in order to compare their performance.
No let's see which size performs better than the others :(see code)

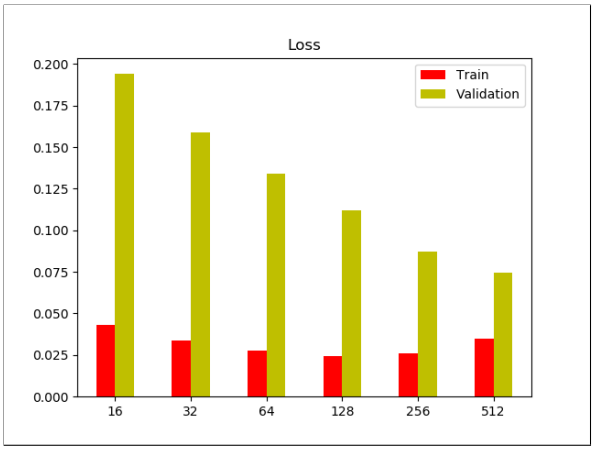


Figure 3.13: **Train/Validation Loss for each batch size**

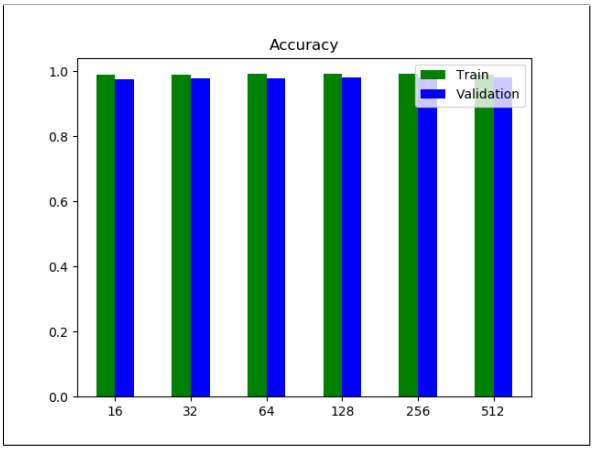


Figure 3.14: **Train/Validation accuracy for each batch size**

Results :

From the experiment above, it appears that **256 batch size** is the best choice, this is due to the fact that the accuracies are almost the same, and 512 performs slightly better.

3.2.5 Type of Pooling

Pooling requires selecting a pooling operation, exactly like a filter to be applied to feature maps. It is always 2x2 pixels applied with two common functions **Average Pooling** and **Maximum Pooling**. **Result:** It appears that MaxPooling2D works better than the AveragePooling2D.(see code)

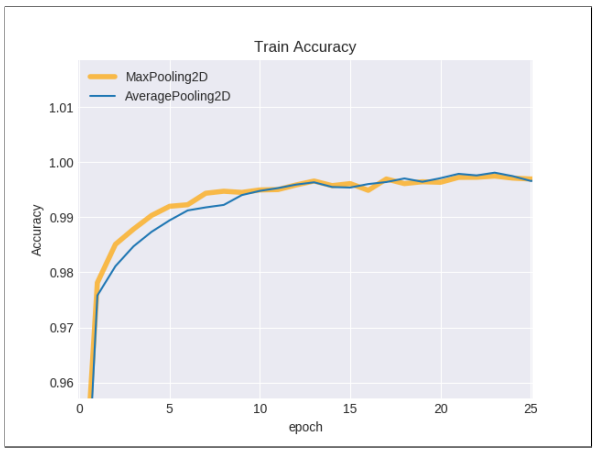


Figure 3.15: **Train accuracy of different types of pooling**

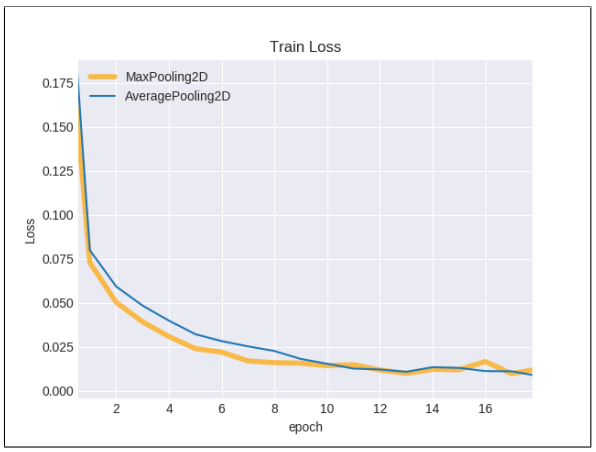


Figure 3.16: **Train loss of different types of pooling**

3.2.6 Number of convolutional layers

Let's see whether one, two, or three pairs of convolution-samplings is the best. We are not going to create four pairs since the shape of the image will be reduced to (1, 1) before the fourth one.

```
Training accuracy :
Conv :  1  :  0.9925100013613701
Conv :  2  :  0.9916629165410995
Conv :  3  :  0.9869608327746391

Validation accuracy :
Conv :  1  :  0.981754994392395
Conv :  2  :  0.9862325027585029
Conv :  3  :  0.9811849951744079

Training loss :
Conv :  1  :  0.02401898671020684
Conv :  2  :  0.02728282084865517
Conv :  3  :  0.04178412406994752

Validation loss :
Conv :  1  :  0.11735451000923194
Conv :  2  :  0.071662528568618
Conv :  3  :  0.08598275788797276
```

Figure 3.17: **Summary of different Conv layers**

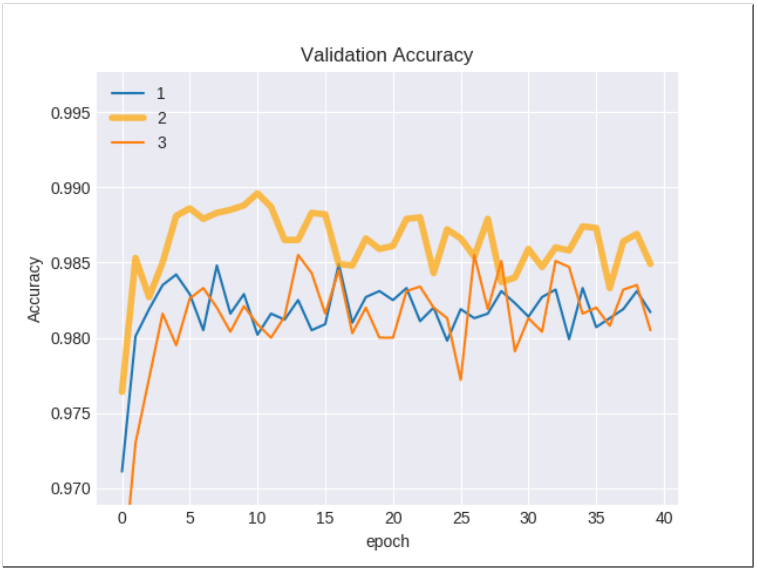


Figure 3.18: **Validation accuracy of different models(see code)**

Comment :

As we said before, 2 stacked small 3x3 feature detectors perform perfectly than 1 convolutional layer. Moreover, the experiment shows that the third layer is the worst. I think this is due to the fact that we lose more information, because the size of the feature map is divided by 2 after each Pool.

3.2.7 Number of filters

In the previous experiment, we proved that two pairs of convolution-subsambling is sufficient.
But how many filters should we include ? (see code)

For example, we can choose between:

- (28, 28, 1) -> (8C3)(MP2) -> (16C3)(MP2)
- (28, 28, 1) -> (16C3)(MP2) -> (32C3)(MP2)
- (28, 28, 1) -> (32C3)(MP2) -> (64C3)(MP2)
- **(28,28,1)->(64C3)(MP2)->(128C3)(MP2)**
- (28, 28, 1) -> (128C3)(MP2) -> (256C3)(MP2)
- (28, 28, 1) -> (256C3)(MP2) -> (512C3)(MP2)

Remark :

- **8C3** means a convolution layer with **8** feature maps using a **3x3** filter.
- **MP2** means max pooling using **2x2** filter.

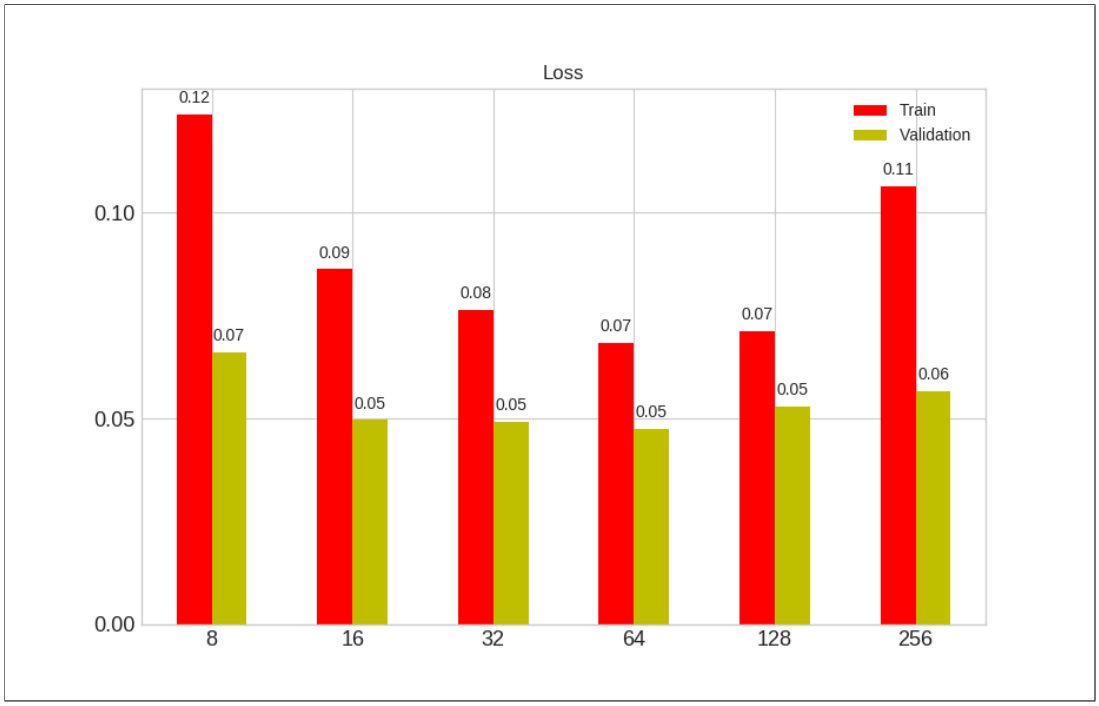


Figure 3.19: **Train/validation loss of different kernels**

Results :

It appears that 64 maps in the first convolutional layer and 128 maps in the second convolutional layer is the best combination since it keeps the loss to the minimum.

3.2.8 Number of dense layers

We compare the performance of the best model deducted from the previous experiments, but this time with different number of dense layers.(**see code**)

Result:

It appears that only **one dense layer** shows the best results, because its average of accuracies over epochs is higher than the accuracies given by more layers. So the model is not worth the additonal computation cost.

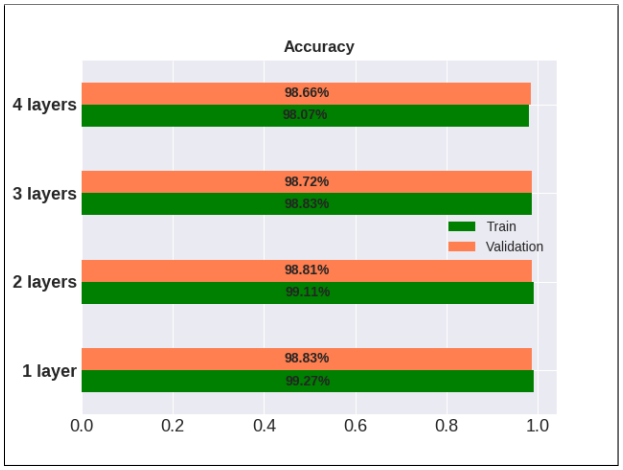


Figure 3.20: **Train/Validation Accuracy of 4 models**

3.2.9 Number of nodes in a hidden layer

The number of nodes in each hidden layer is a hyperparameter that we must specify.
So we evaluate the same model with different dense units(**see code**):

Result:

From this experiment, it clear that **128 nodes** is the best choice.
In fact, its related model does not suffer any bad decreases. So it is more stable than other choices.

3.2.10 Dropout & Batch normalisation

1) Technically, **dropout** refers to the fact that we ignore some nodes during the training phase, exactly during a particular forward or backward pass.
A fully connected NN contains a lot of parameters, that means more dependencies between neurons, and perhaps this may lead to the **overfitting** of training data. So, in order to help our network generalize better, we need to dropout some nodes.

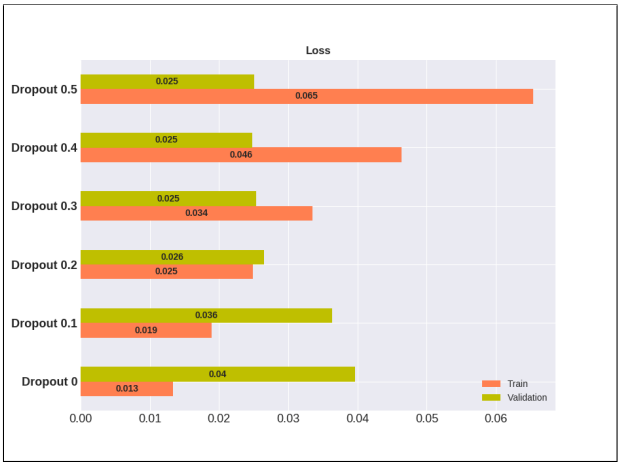
2) **Batch Normalisation** is a technique for improving the stability and performance of a neural model. The idea is to normalise the inputs of each layer, and this helps the model learn faster.

Remark : We always keep in mind that we normalise the output of each layer before applying the activation function, then we dropout if necessary.

Now let's see how much dropout we should add after each layer:(**see code**)

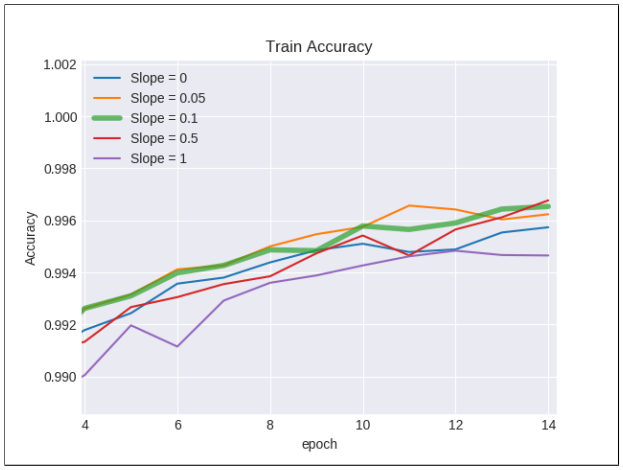
Results:

We notice that the difference, between the error found for validation that of training, is large if we dropout nothing or if we dropout more nodes. Therefore, this is a sign of overfitting, so the best choice is **Dropout=0.2** where the difference is small.



3.2.11 Non-linear activation functions

The role of the rectifier activation function is to have non-linearity in our CNN. That is why we use the **Relu** function as a first model. So once a neuron gets negative inputs, it will always output zero, and it will be inactive forever. This phenomenon is called **Dying Relu**.
So, how can we resolve this problem ?(**see code**)



Solution:

Well, **Leaky ReLU** is the most common and effective method to prevent a dying ReLU by adding a slight slope in the negative range.
So, we trained our model with different slopes, then we find that all slopes between **0.05** and **0.1** are the best choice. We choose then **slope=0.08**.

3.2.12 Image Augmentation

Image Data augmentation is a wide range of techniques to artificially generate new training samples from the existing ones by creating random perturbations and ensuring at the same time that the class features are not changed. This technique involves generating transformed versions of images so as to increase the generalizability of the model.(**see code**)

RK : We have to keep in mind that this operation is not **"additive"**. It is not returning both the transformed data and the original one. Instead, it returns only the randomly transformed data. There are two reasons for this approach :

1 - Different Styles:

That's right thanks to data augmentation we can apply a slight amount of transformation to an input image without changing its class label.
So the purpose is to create images with different styles, because in the real world data rarely follow the same structure and the model can not predict the class of and image it does not see before.

This operation is applied thanks to **Keras ImageDataGenerator** , an object that is used to apply data augmentation, randomly translating, rotating, rescaling, resizing, shearing, flipping ...

```
def augmentation(shear, wshift, rot, zoom):
  train_datagen = ImageDataGenerator(
                                  shear_range = shear,
                                  width_shift_range=wshift,
                                  rotation_range=rot,
                                  zoom_range = zoom,
                                  fill_mode="nearest"
                                  )
  training_set = train_datagen.flow(train_x, train_y, batch_size = 256)
  return training_set
```

Listing 3.1: Function returns randomly transformed data

We choose just small values because our goal is to create just a slight perturbation. However, we test without augmentation, that means we evaluate our network on the unmodified testing data.

```
# unmodified testing data
test_datagen = ImageDataGenerator()
test_set = test_datagen.flow(test_x, test_y, batch_size = 256)
# transformed training data
training_set = augmentation(shear=0.1, wshift=0.02, rot=8, zoom=0.08)
```

Listing 3.2: Data augmentation parameters

We perform the image by applying different geometric transformations:
For example, we can obtain augmented data from the original images:

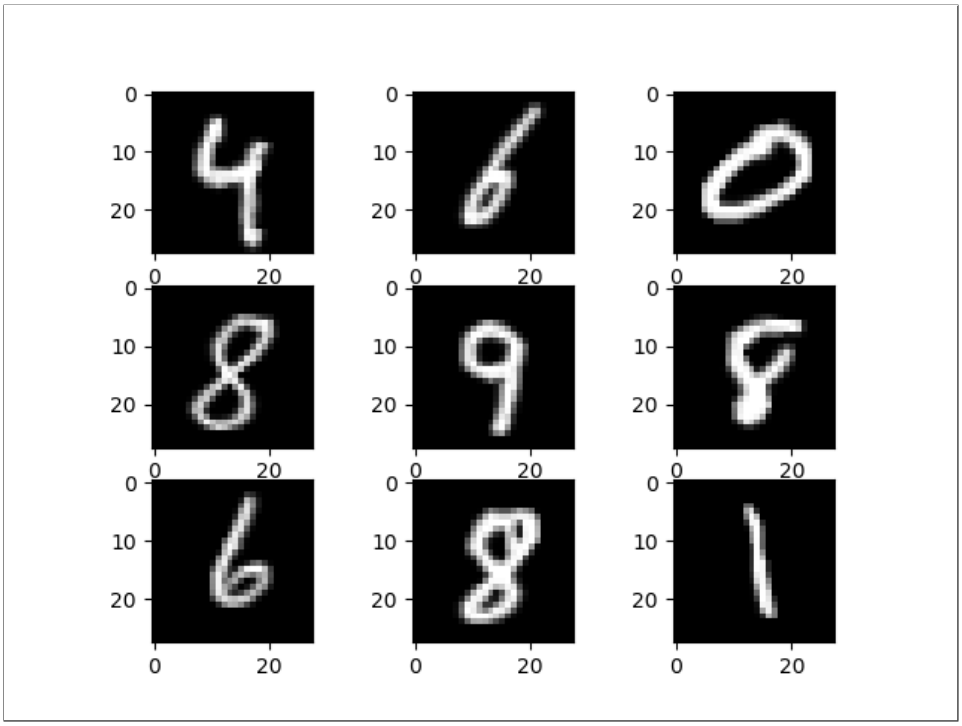


Figure 3.21: **Transformed images**

2 - It can reduce Overfitting:

Here we start our last experiment by training our previous model without using data augmentation:

```
classifier.fit(train_x, train_y, validation_data= (test_x, test_y),
               batch_size = 256, epochs = 50)
```

Listing 3.3: **Classifier without augmentation**

```
def plot_history():
    plt.plot(history.history['loss'], marker='', linewidth=4, alpha=0.7)
    plt.plot(history.history['val_loss'], marker='', linewidth=4, alpha=0.7)
    plt.title('Model␣Loss', fontweight="bold")
    plt.ylabel('Loss', fontweight="bold")
    plt.xlabel('epoch', fontweight="bold")
    plt.legend(['train_loss', 'test_loss'], loc='upper␣left', prop={"weight":"bold"})
    plt.show()
```

Listing 3.4: Function that plots the loss of each set

So let's take a look at the plot associated to our model:

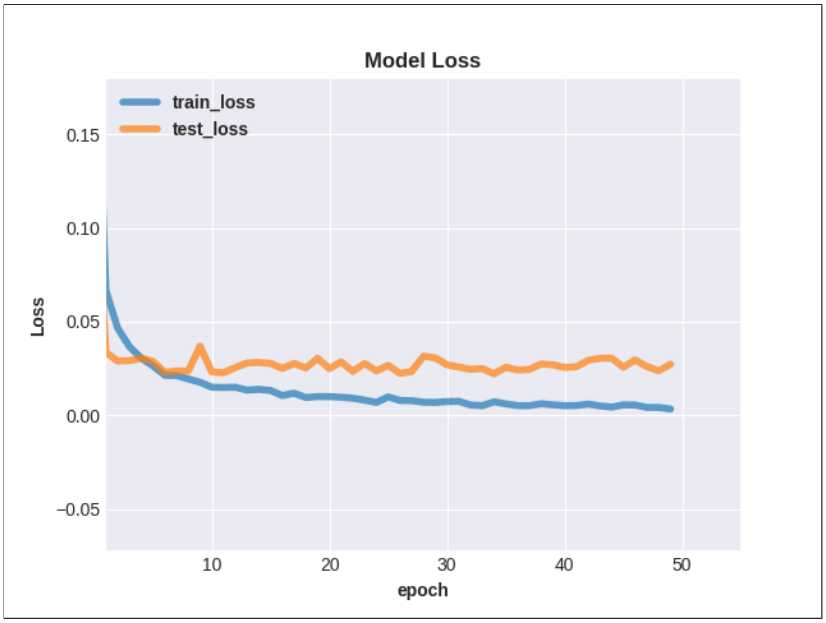


Figure 3.22: **Strong indications of overfitting**

Comment :

At approximately epoch 5 we see our validation loss starts to stabilize while training loss continues to fall. So this behavior is an indicative of overfitting.

Now let's now investigate how data augmentation can act as a form of solution:

```
classifier.fit_generator(training_set,
                         steps_per_epoch = 60000//256,
                         epochs =50,
                         validation_data = test_set,
                         validation_steps = 10000//256)
```

Listing 3.5: **Classifier using augmentation**

We fit the transformed data to the Convolutional Neural Network that we created above using **fit_genator** (This is for a practical purpose, it is used to avoid duplicate data when using multi-processing and when we have large dataset which is the case here).
This function will return a history object; By storing the result of this function in a variable, we can use it to plot the accuracy and loss function plots between training and validation which will help us to analyze our model's performance visually.

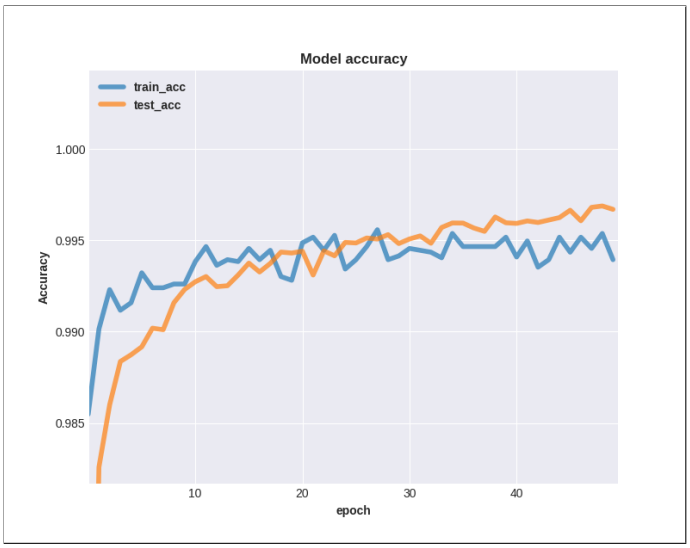


Figure 3.23: **Train/Validation accuracy**

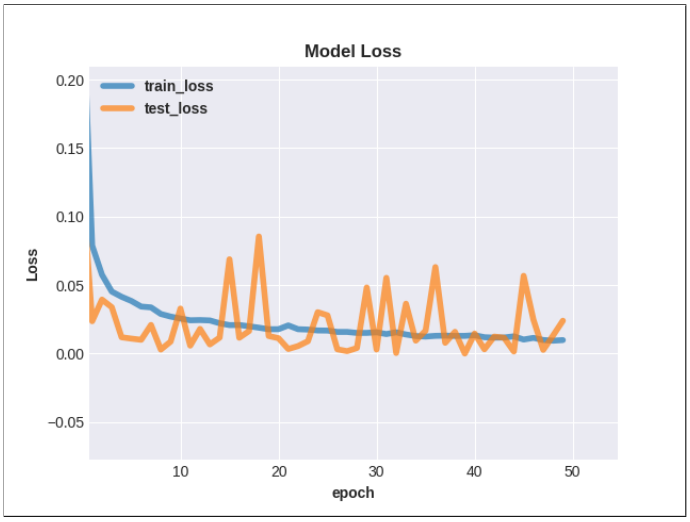


Figure 3.24: **Train/Validation Loss**

Results :

It looks like the validation loss oscillates around the training loss, and also there is not much gap between validation and training accuracy. This gives an intuition that the network has memorized the training data very well and is also guaranteed to work on unseen data.
Therefore, we can say that our model's generalization capability became much better. So the solution to prevent overfitting is by performing regularization, that means by applying data augmentation only for training data.

Chapter 4

Conclusion

4.1 Final architecture

From the previous experiments, we list all features that improve the accuracy of our CNN model:

* **Spatial-extent:**	**3x3**
* **Stochastic gradient descent optimizer:**	**Adam**(**Learning rate** = 0.01)
* **Loss-function:**	**categorical_crossentropy**
* **Batch size:**	**256**
* **Type of pooling:**	**Maximum Pooling 2D**
* **Number of convolutional layers:**	**Two** pairs
* **Number of filters:**	(**64**C3)(MP2)->(**128**C3)(MP2)
* **Number of dense layers:**	**One** dense layer
* **Number of nodes:**	**128** nodes
* **Dropout:**	**0.2**
* **Non-linear activation function:**	**LeakyRelu**(**alpha**=0.08)
* **Augmentation:**	(**shear**=0.1, **wshift**=0.02, **rot**=8, **zoom**=0.08)

Now we gather all these features in one model that we build below:

```
# initializing the neural network
classifier = Sequential()

# the first convolutional layer
classifier.add(Conv2D(64, (3, 3), input_shape = (28, 28, 1)))
BatchNormalization(axis=-1)
classifier.add(LeakyReLU(0.08))
BatchNormalization(axis=-1)
classifier.add(MaxPooling2D(pool_size = (2, 2)))
BatchNormalization(axis=-1)
classifier.add(Dropout(0.2))
BatchNormalization(axis=-1)

# the second convolutional layer
classifier.add(Conv2D(128, (3, 3)))
BatchNormalization(axis=-1)
classifier.add(LeakyReLU(0.08))
BatchNormalization(axis=-1)
classifier.add(MaxPooling2D(pool_size = (2, 2)))
classifier.add(Dropout(0.2))

# flattening
classifier.add(Flatten())
classifier.add(BatchNormalization())

# full connection
classifier.add(Dense(units = 128, kernel_initializer = 'uniform'))
classifier.add(BatchNormalization())
classifier.add(Activation('relu'))
classifier.add(Dropout(0.2))
classifier.add(BatchNormalization())

# the output layer
classifier.add(Dense(units = 10, kernel_initializer = 'uniform'))
classifier.add(BatchNormalization())
classifier.add(Activation('softmax'))

# compiling the CNN
classifier.compile(optimizer = Adam(learning_rate=0.01),
                   loss = 'categorical_crossentropy',
                   metrics = ['accuracy'])
```

Listing 4.1: **The final architecture of our CNN**

After building the CNN, let's visualize the architecture that we created in the above page by using the **summary** function. This will show some extra parameters such as weights and biases.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 64)        640
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 26, 26, 64)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 13, 13, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 128)       73856
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 11, 11, 128)       0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 5, 5, 128)         0
_________________________________________________________________
dropout_2 (Dropout)          (None, 5, 5, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 3200)              0
_________________________________________________________________
batch_normalization_7 (Batch (None, 3200)              12800
_________________________________________________________________
dense_1 (Dense)              (None, 128)               409728
_________________________________________________________________
batch_normalization_8 (Batch (None, 128)               512
_________________________________________________________________
activation_1 (Activation)    (None, 128)               0
_________________________________________________________________
dropout_3 (Dropout)          (None, 128)               0
_________________________________________________________________
batch_normalization_9 (Batch (None, 128)               512
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1290
_________________________________________________________________
batch_normalization_10 (Batc (None, 10)                40
_________________________________________________________________
activation_2 (Activation)    (None, 10)                0
=================================================================
Total params: 499,378
Trainable params: 492,446
Non-trainable params: 6,932
```

4.2 Results of performance (see code)

It's finally time to train our model with Keras **fit_generator()** function seen at the previous section.

```
classifier.fit_generator(training_set, steps_per_epoch = 60000//256, epochs =20)
```

Listing 4.2: **Training the final model**

In order to save our time, let's save this model so that we can directly load it without training it again.

```
classifier.save("cnn.h5py")
```

Listing 4.3: **Saving the model in a file**

Now let's evaluate the performance of our model on the test set and see how it performs:

```
Y_pred = classifier.predict(test_x)
```

Listing 4.4: **Predicting Labels**

Since the predictions we get are float values, it will not be feasible to compare the predicted classes with true test classes. That is why we will use **np.argmax()** to select the index number which has a higher value in the row.

```
Y_pred_classes = np.argmax(Y_pred,axis = 1)
Y_true = np.argmax(test_y,axis = 1)
```

Classification Report

Classification report will help us ti identify the misclassified classes in more detail.
We will be able to observe for which class the classifier performed bad out of the given ten classes.

```
from sklearn.metrics import classification_report
dict_characters = {0: '0', 1: '1', 2: '2', 3: '3', 4: '4',
                   5: '5', 6: '6', 7:'7', 8: '8', 9: '9'}

print(classification_report(Y_true, Y_pred_classes,
       target_names=list(dict_characters.values())), sep='')
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	980
1	0.99	1.00	1.00	1135
2	1.00	1.00	1.00	1032
3	0.99	1.00	1.00	1010
4	0.99	1.00	0.99	982
5	0.99	0.99	0.99	892
6	1.00	0.99	0.99	958
7	0.99	1.00	1.00	1028
8	1.00	1.00	1.00	974
9	1.00	0.99	0.99	1009
accuracy			1.00	10000
macro avg	1.00	1.00	1.00	10000
weighted avg	1.00	1.00	1.00	10000

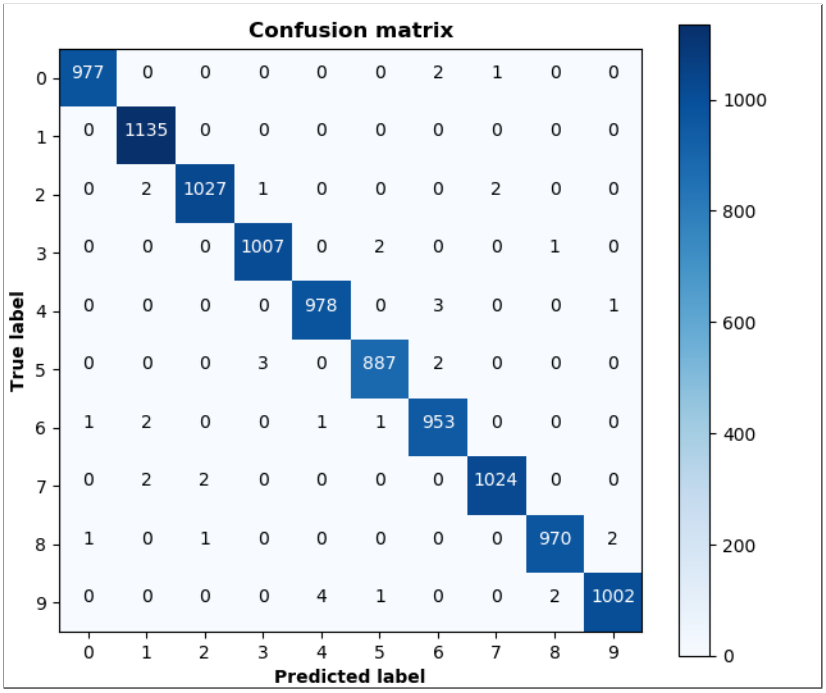
We can see that the model is perfectly performing for all classes regarding both precision and recall.

Confusion Matrix

A confusion matrix is a performance measurement technique for classification. It is a kind of table which helps us to the know the performance of our classification model on a set of test data for that the true values are known. Now let's plot this matrix:

```
import pandas as pd
import seaborn as sn
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(Y_true, Y_pred_classes)
cm = pd.DataFrame(cm, index = [i for i in "0123456789"],
                  columns = [i for i in "0123456789"])
plt.figure(figsize = (5,5))
sn.heatmap(cm, annot=True)
```



This matrix is extremely useful for measuring Recall, Precision, Accuracy and most importantly AUC-ROC Curve. ROC shows how much the network is capable of distinguishing between digits.

ROC curves & Precision-Recall plots

So, let's plot the ROC graph that summarizes the performance of our classifier over all possible thresholds:

```
from sklearn.metrics import roc_curve, precision_recall_curve, auc
fpr, tpr, roc_auc = dict(), dict(), dict()
for i in range(10):
    fpr[i], tpr[i], _ = roc_curve(test_y[:, i], Y_pred[:, i])
    rec[i], pre[i], _ = precision_recall_curve(test_y[:, i], Y_pred[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])
# Plotting ROC curves
for i in range(10):
    plt.plot(fpr[i], tpr[i], lw=2,
             label='Digit␣{}␣(area␣=␣{})'.format(i,format(roc_auc[i], '0.4f')))
    plt.show()
# Plotting Precision-Recall plots
for i in range(10):
    plt.plot(rec[i], pre[i], lw=2, label='Digit␣{}'.format(i))
    plt.show()
```

Here are the results:

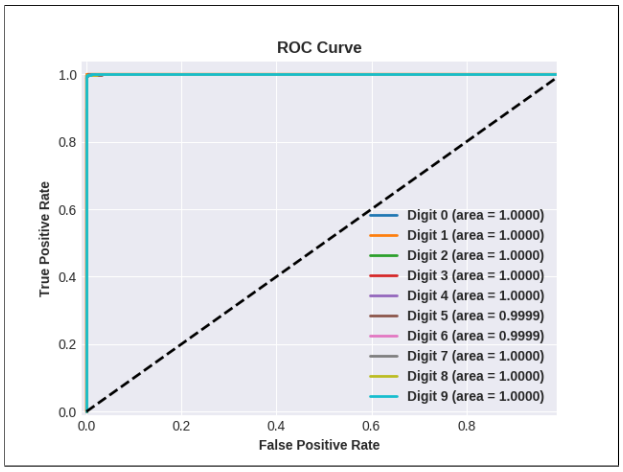


Figure 4.1: **ROC curves**

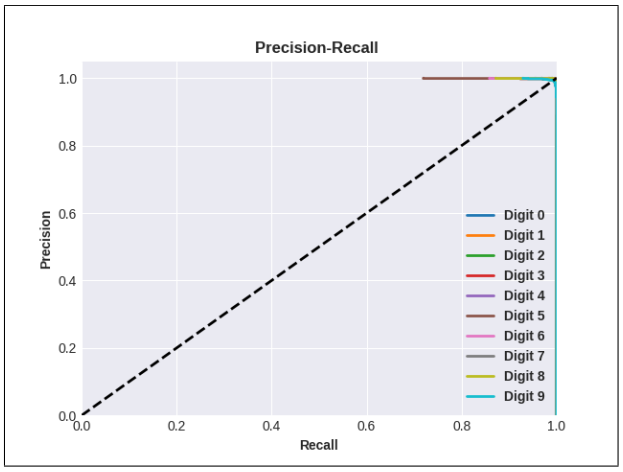


Figure 4.2: **Precision-Recall plots**

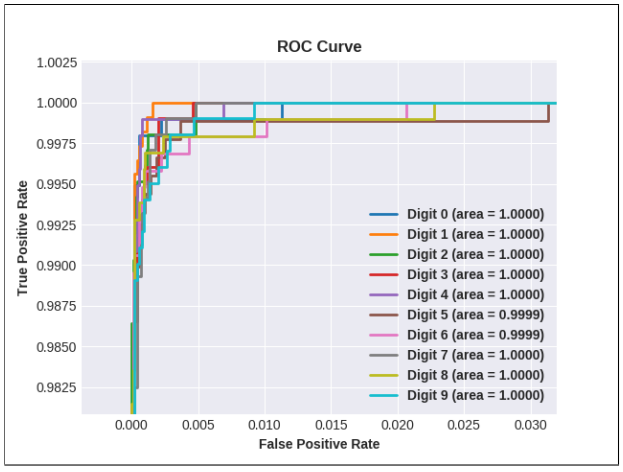


Figure 4.3: **ROC curves (zoomed in at top left)**

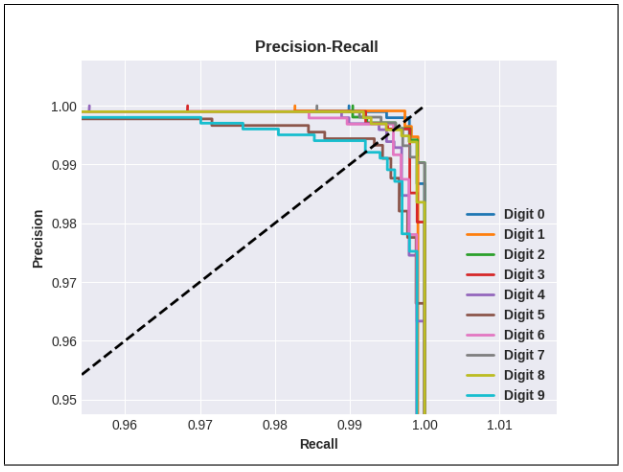


Figure 4.4: **Precision-Recall plots (zoomed in at top right)**

Results

The ROC curves visualizes the quality of our model on a test set, and as we can see for each class, their ROC and AUC values are slightly different and are at the same time near to the 1. This gives us a good indication of how good our model is at classifying digits.
We can say that we built an excellent model since it has good measure of separability.

Extra Test (see code)

Now, we're going to do something very fun, something very exciting, we're going to experiment with a tool where we can draw our own test (See website).
So here we need to draw a number, so let's say we draw a number 4.

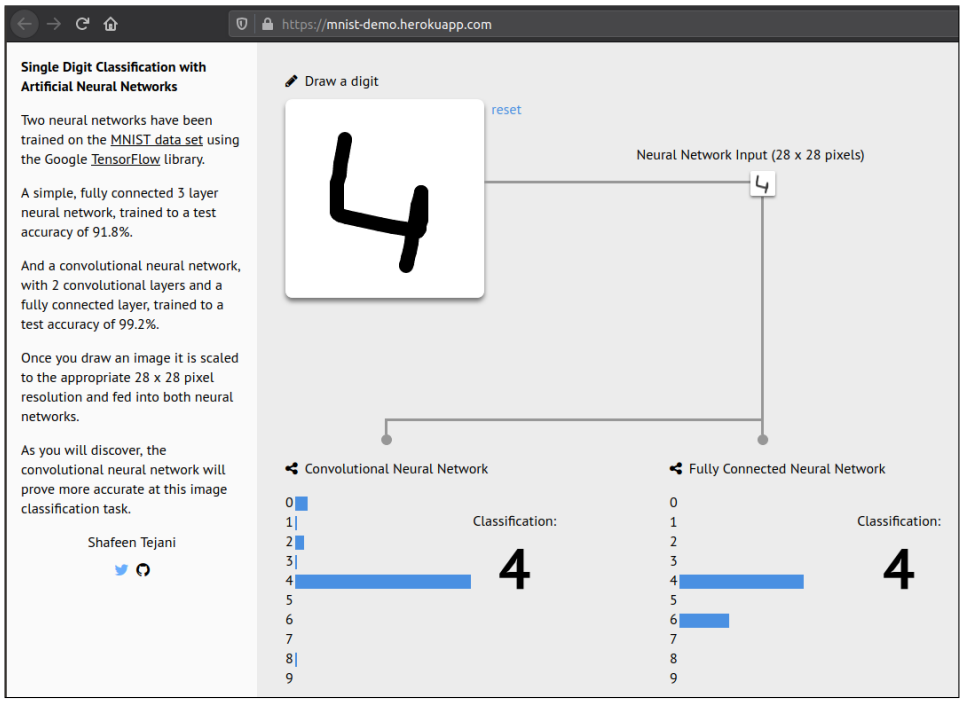


Figure 4.5: **Website that enables users to hand-draw a digit**

Then we take screenshot to that image as an input of our convolutional neural network.
Since the number is black and the background is white, we must permute these colors so as to have the same features of MNSIT data. Now, let's predict this number with our model.

```
# load the model
from keras.models import load_model
classifier = load_model("cnn.h5py")
# import the input image
from keras.preprocessing import image
import cv2
ii = cv2.imread("4.png")
ii = cv2.resize(ii, (28, 28))
# convert the image to gray one
test_image = cv2.cvtColor(ii, cv2.COLOR_BGR2GRAY)
test_image = image.img_to_array(test_image)
# permuting black and white pixels & normalizing
test_image = test_image.astype('float32')
test_image = (255 - test_image)/255
# reshape to (28, 28, 1)
test_image = np.expand_dims(test_image, axis = 0)
# predict the digit
result = classifier.predict(test_image)
# plot the image with the predicted digit
plt.imshow(test_image.reshape(28,28))
plt.title("␣The␣predicted␣digit␣is␣:␣" +
          str(np.argmax(result,axis = 1)[0]),
          fontweight="bold")
plt.show()
```

Well, here is the result :

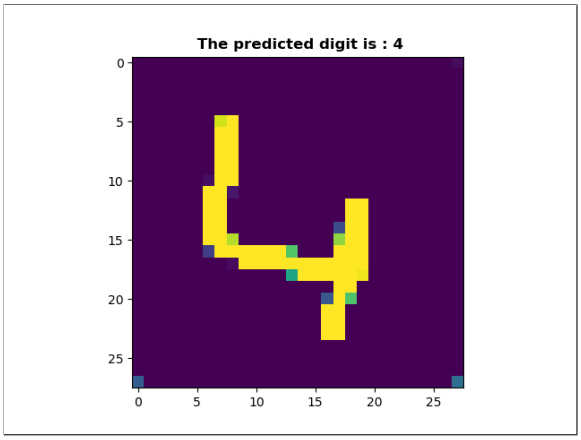


Figure 4.6: **4 is the output of our cnn**

Summary

The performance of Convolutional Neural Networks depends on multiple hyperparameters. Thus, it's advisable to tune the model by conducting lots of experiments. Once the right hyperparameters are found, the classifier should be trained with a larger number of epochs.

Thank you for reading and I hope you found this report beautiful.
Don't forget to see the source code here.

The End
* *
*

Deep Learning.