UNIVERSITÀ DI PISA

Computational Intelligence & Deep Learning Project

A.A. 2022/23

Report

Developed by:

- **Domenico Armillotta -** badge : 643020 - email: d.armillotta@studenti.unipi.it
- **Stefano Dugo -** badge : 564370 email: s.dugo@studenti.unipi.it

Link of Google Colab:

https://colab.research.google.com/drive/1HDUn5_bx5Hmeqm3zOZktI_gmiK2pzlLx?usp=sharing

INTRODUCTION

The objective is to create two image classifiers, one developing a CNN from scratch, and the second with a pre-trained keras
model but using feature extraction and fine tuning techniques.
Then compare the results.
If there is time left, we will try an implementation using YOL.

FIRST PART - CNN from Scratch

DATASET

The German Traffic Sign Benchmark is a multi-class, single-image classification challenge held at the International Joint Conference on Neural Networks (IJCNN) 2011.
Characteristic of dataset :

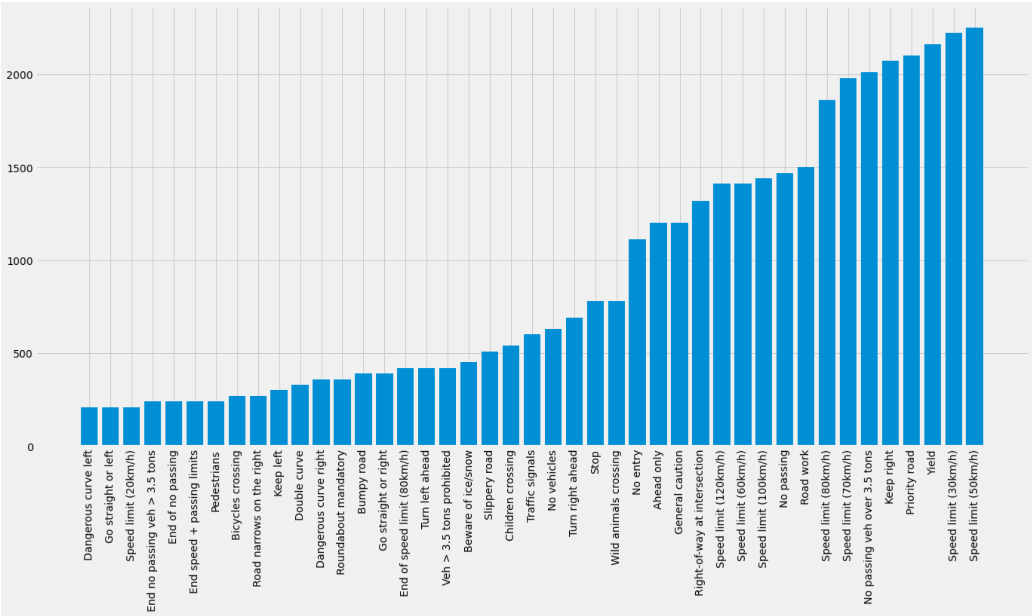
- More than 40 classes
- More than 50,000 images in total
- Large, lifelike database

Link = [GTSRB - German Traffic Sign Recognition Benchmark | Kaggle]()

In our project the dataset has been uploaded to google Drive , so that it can be used on google colab.

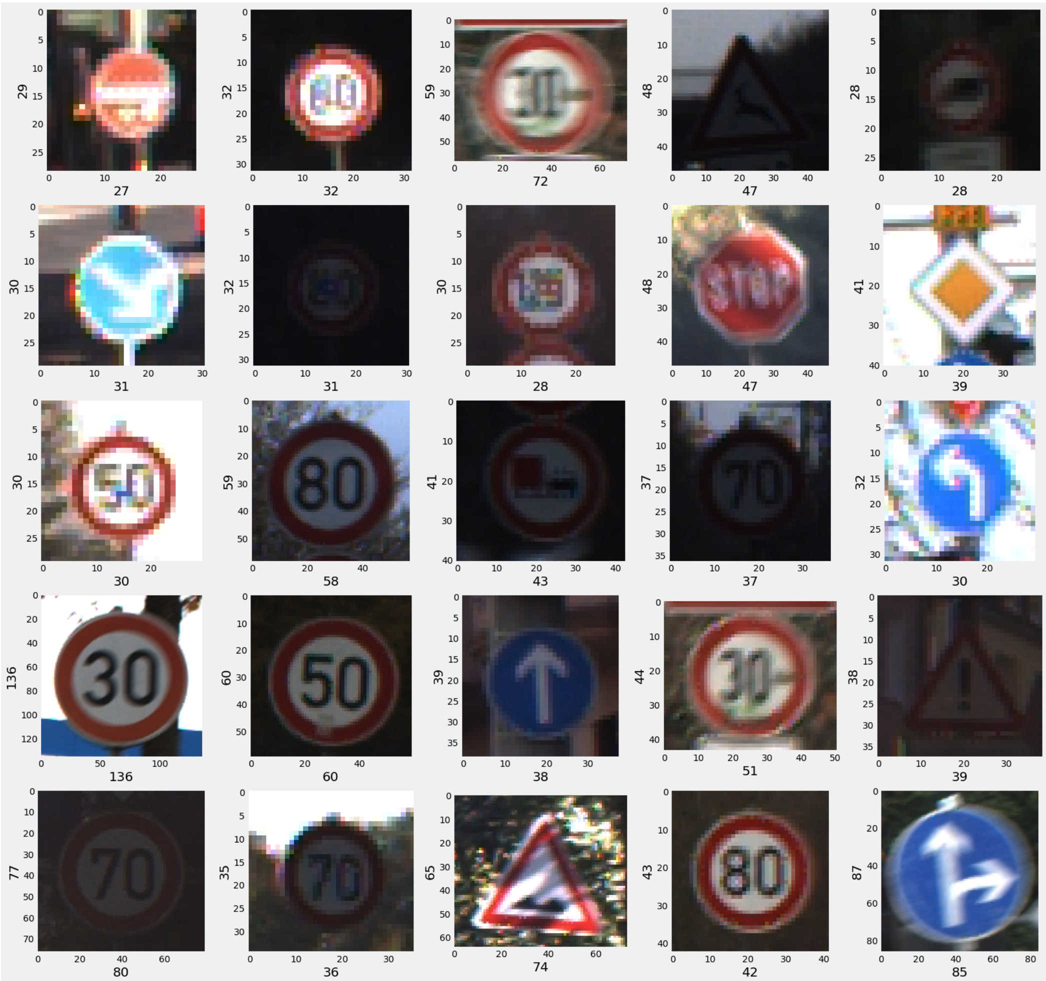
Prepare the train and test set

We assigned a textual label to each class.
The plot of the distribution of the data we have is as follows:



The images are organised in folders indicating their class, each one has a different size, but will be resized later.

Here are some examples of street sign images:



We can see that there are images with different gloss, different rotation and so on.
Once we have placed the images and labels in a Nunpy array, we split the data in the Train folder into Training set and Validation Set according to the 80/20 proportions, while the images in the Test folder we will use to compose the Test set.
For labels we use the One Hot Encoding notation.
In the end we will have :
Training = 32k
Validation = 7k
Testing = 18k

Augmentation:

According to keras documentation when you don't have a large image dataset, it's a good practice to artificially introduce sample diversity by applying random yet realistic transformations to the training images
This helps expose the model to different aspects of the training data while slowing down overfitting.
According to the keras documentation, there are two methods to carry out augmentation, we have chosen to implement the second strategy, i.e. to apply it directly on the train dataset. With this option, your data augmentation will happen on CPU, asynchronously, and will be buffered before going into the model

The parameters chosen for Augmentation are as follows:

```
rotation_range=10,
zoom_range=0.15,
width_shift_range=0.1,
height_shift_range=0.1,
shear_range=0.15,
horizontal_flip=False,
vertical_flip=False,
```

Handle Imbalanced Dataset

One way to address class imbalance in a convolutional neural network (CNN) is to use class weights. Class weights are used to adjust the loss function of the model in order to give more weight to the examples in the minority classes, which can help the model to better learn from these classes and improve its overall performance.

In the Keras library, you can specify class weights when compiling a model by using the class_weight argument in the compile() function. The class weights should be specified as a dictionary that maps class labels to weights.

Develop the AlexNet Model based Architecture:

Architecture:

This is very similar to the architectures that Alex Krizhevsky advocated in the 2012 for image classification

Dropout:

Dropout technique works by randomly reducing the number of interconnecting neurons within a neural network. At every training step, each neuron has a chance of being left out, or rather, dropped out of the collated contributions from connected neurons.

In the original paper is placed at the end with 0.5 value but More recent research has shown some value in applying dropout also to convolutional layers, although at much lower levels: p=0.1 or 0.2. Dropout was used after the activation function of each convolutional layer: CONV->RELU->DROP.

Convolutional layer:

A convolution is a mathematical term that describes a dot product multiplication between two sets of elements. Within deep learning the convolution operation acts on the filters/kernels and image data array within the convolutional layer. Therefore a convolutional layer is simply a layer the houses the convolution operation that occurs between the filters and the images passed through a convolutional neural network.

Batch Normalisation layer:

Batch Normalization is a technique that mitigates the effect of unstable gradients within a neural network through the introduction of an additional layer that performs operations on the inputs from the previous layer. The operations standardize and normalize the input values, after that the input values are transformed through scaling and shifting operations.

MaxPooling layer:

Max pooling is a variant of sub-sampling where the maximum pixel value of pixels that fall within the receptive field of a unit within a sub-sampling layer is taken as the output. The max-pooling operation below has a window of 2x2 and slides across the input data, outputting an average of the pixels within the receptive field of the kernel.

Flatten layer:

Takes an input shape and flattens the input image data into a one-dimensional array.

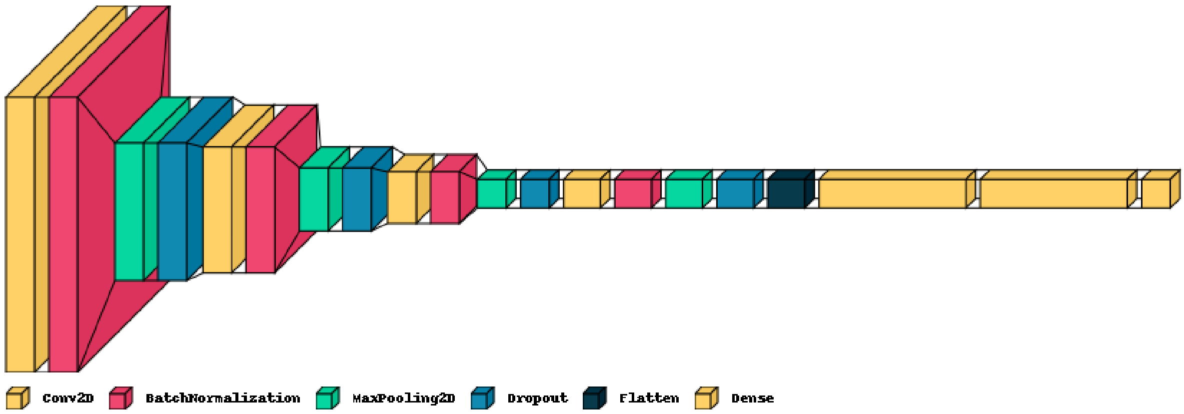
Dense Layer:

A dense layer has an embedded number of arbitrary units/neurons within. Each neuron is a perceptron.

Softmax Activation Function:

A type of activation function that is utilized to derive the probability distribution of a set of numbers within an input vector. The output of a softmax activation function is a vector in which its set of values represents the probability of an occurrence of a class or event. The values within the vector all add up to 1.

Neural network architectures:




```
Total params: 1,747,179
Trainable params: 1,746,219
Non-trainable params: 960
```

Training Phase:

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

Being a multiclass problem, we used the loss function : “cateorical_crossentropy”.
We have trained the model in 15 epochs, but this is a hyper-parameter that we will tune to in the future

Early stopping callback function:

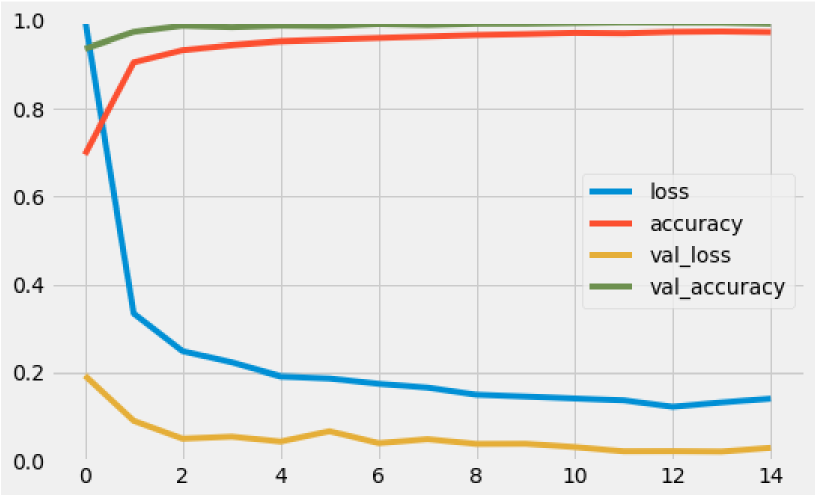
We used the early stopping that is a form of regularization that can be used to prevent overfitting. Is implemented by specifying a callback function in the .fit function that monitors the performance of the model

on a validation dataset and interrupts the training process when the performance stops improving.
We set also the patience parameters = 10.

```
early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor='val_prc',
    verbose=1,
    patience=10,
    mode='max',
    restore_best_weights=True)
```

Evaluate the AlexNet model :

Metrics extrapolated from **Model** Training **History :**



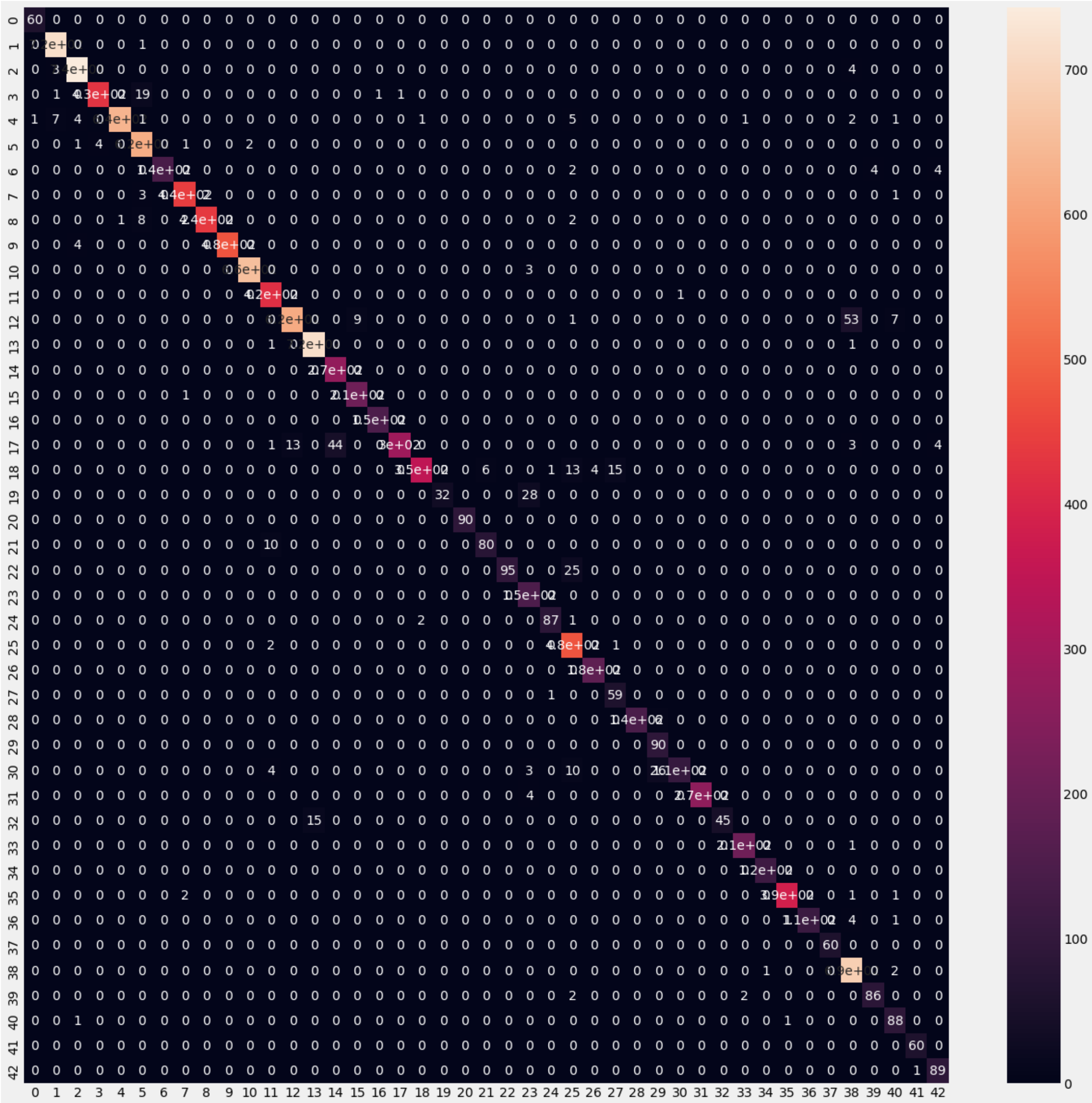
We can see that the model is not in overfitting , because don't have a low error in the training set and a higher error in the testing set.
An attempt was made to combat overfitting by inserting Dropout layers in the model and implementing augmentation of the training set and the early stopping condition.
So we tried to use all the technique to prevent overfitting in our model.

Overall accuracy on the Test Set:

```
395/395 [==============================] - 21s 54ms/step
Test Data accuracy:  96.56373713380839
```

Confusion Matrix :

Elaborated by submitting the test set



Report of the model:

For a more detailed report on each class, this graph shows the precision/recall and f1 measure for each model class.

	precision	recall	f1-score	support
0	0.98	1.00	0.99	60
1	0.98	1.00	0.99	720
2	0.99	0.99	0.99	750
3	0.99	0.95	0.97	450
4	1.00	0.97	0.98	660
5	0.95	0.99	0.97	630
6	1.00	0.93	0.97	150
7	0.99	0.99	0.99	450
8	1.00	0.97	0.98	450
9	1.00	0.99	1.00	480
10	1.00	1.00	1.00	660
11	0.96	1.00	0.98	420
12	0.98	0.90	0.94	690
13	0.98	1.00	0.99	720
14	0.86	1.00	0.92	270
15	0.96	1.00	0.98	210
16	0.99	1.00	1.00	150
17	1.00	0.82	0.90	360
18	0.99	0.90	0.94	390
19	1.00	0.53	0.70	60
20	1.00	1.00	1.00	90
21	0.93	0.89	0.91	90
22	1.00	0.79	0.88	120
23	0.80	1.00	0.89	150
24	0.98	0.97	0.97	90
25	0.89	0.99	0.94	480
26	0.98	1.00	0.99	180
27	0.79	0.98	0.87	60
28	1.00	0.96	0.98	150
29	0.74	1.00	0.85	90
30	0.99	0.71	0.83	150
31	1.00	0.99	0.99	270
32	1.00	0.75	0.86	60
33	0.99	1.00	0.99	210
34	0.99	1.00	1.00	120
35	0.99	0.99	0.99	390
36	1.00	0.95	0.97	120
37	1.00	1.00	1.00	60
38	0.91	1.00	0.95	690
39	0.96	0.96	0.96	90
40	0.87	0.98	0.92	90
41	0.98	1.00	0.99	60
42	0.92	0.99	0.95	90
accuracy			0.97	12630
macro avg	0.96	0.95	0.95	12630
weighted avg	0.97	0.97	0.97	12630

Visual Network Test:

25 random images of the dataset were put to the model test, and we observe how they are classified


Actual=16 || Pred=16
Actual=1 || Pred=1
Actual=38 || Pred=38
Actual=33 || Pred=33
Actual=11 || Pred=11
Actual=38 || Pred=38
Actual=18 || Pred=18
Actual=12 || Pred=12
Actual=25 || Pred=25
Actual=35 || Pred=35
Actual=12 || Pred=12
Actual=7 || Pred=7
Actual=23 || Pred=23
Actual=7 || Pred=7
Actual=4 || Pred=4
Actual=9 || Pred=9
Actual=21 || Pred=21
Actual=20 || Pred=20
Actual=27 || Pred=27
Actual=38 || Pred=38
Actual=4 || Pred=4
Actual=33 || Pred=33
Actual=9 || Pred=9
Actual=3 || Pred=3
Actual=1 || Pred=1

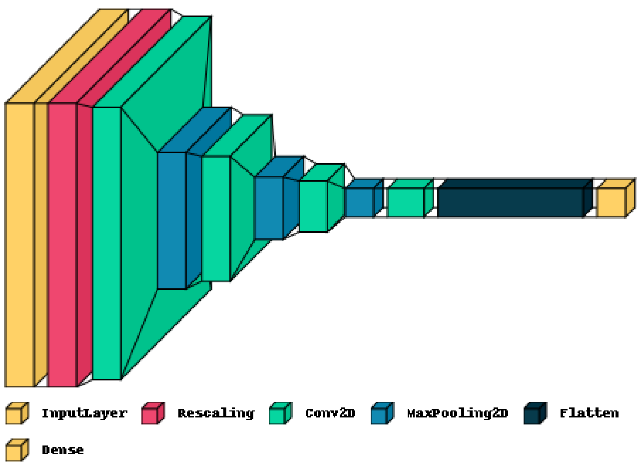
Develop the MNIST like Model based architecture:

Architecture:

We will reuse the same general structure of MNIST: our convnet will be a stack of alternated Conv2D (with relu activation) and MaxPooling2D layers.

However, since we are dealing with bigger images and a more complex problem, we will make our network accordingly larger: it will have two more Conv2D + MaxPooling2D stage. This serves both to augment the capacity of the network, and to further reduce the size of the feature maps, so that they aren't overly large when we reach the Flatten layer.

Note that the depth of the feature maps is progressively increasing in the network (from 32 to 256), while the size of the feature maps is decreasing. This is a pattern that you will see in almost all convnets.




```
Total params: 616,291
Trainable params: 616,291
Non-trainable params: 0
```

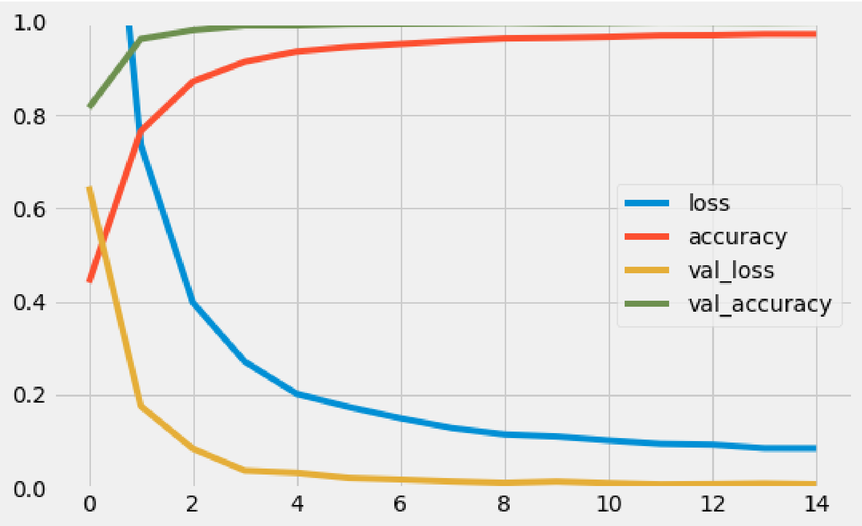
Training Phase:

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

We have trained the model in 15 epochs, but this is a hyper-parameter that we will tune to in the future

Evaluate the Mnist model :

Metrics extrapolated from **Model** Training **History** :



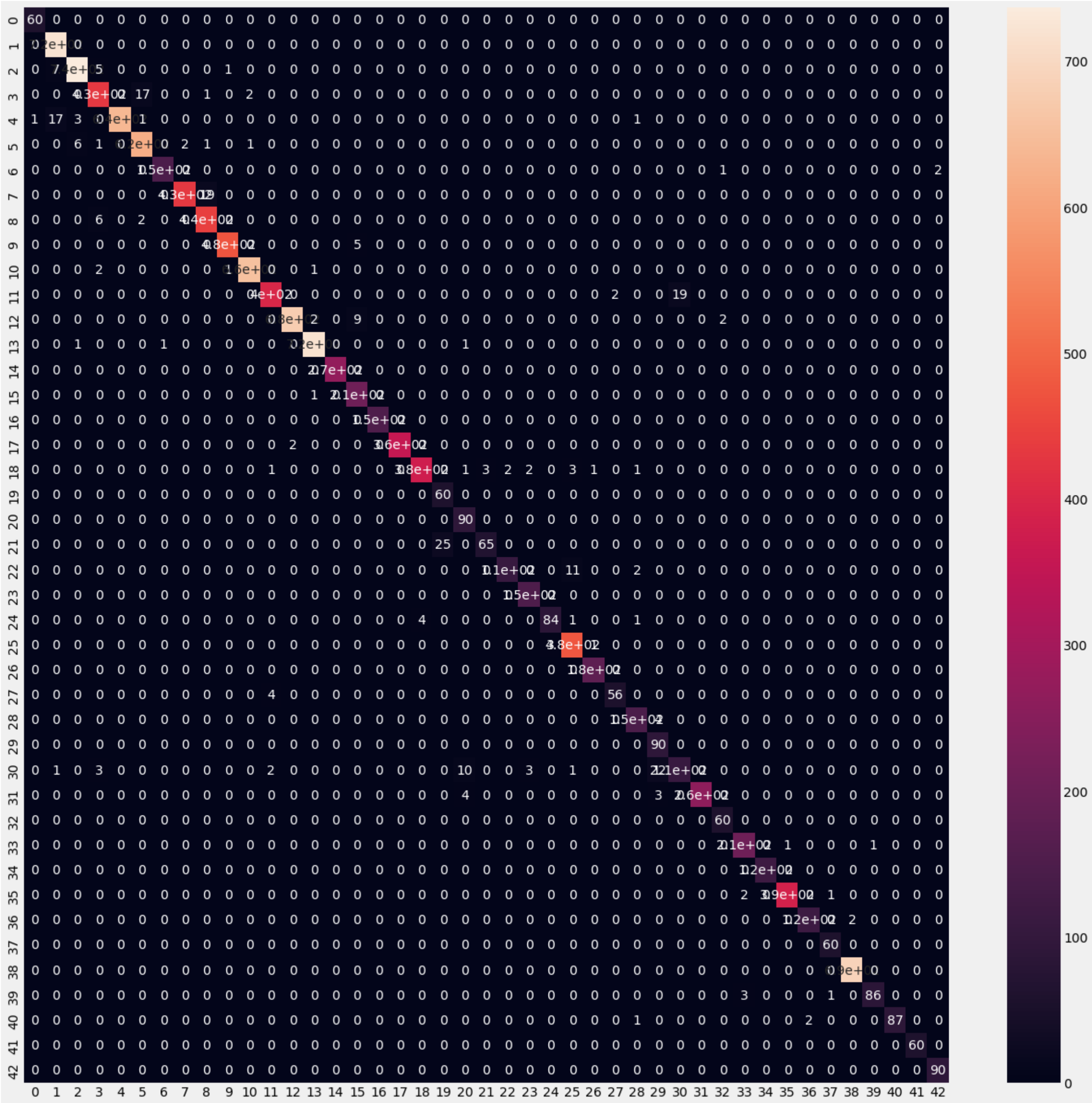
We can see that the model is not in overfitting , because don't have a low error in the training set and a higher error in the testing set.
An attempt was made to combat overfitting by inserting Dropout layers in the model and implementing augmentation of the training set.

Overall accuracy on the Test Set:

```
395/395 [==============================] - 16s 41ms/step
Test Data accuracy:  97.79097387173397
```

Confusion Matrix :

Elaborated by submitting the test set



Report of the model:

For a more detailed report on each class, this graph shows the precision/recall and f1 measure for each model class.

	precision	recall	f1-score	support
0	0.98	1.00	0.99	60
1	0.97	1.00	0.98	720
2	0.99	0.98	0.98	750
3	0.96	0.96	0.96	450
4	1.00	0.97	0.98	660
5	0.97	0.98	0.98	630
6	0.99	0.98	0.99	150
7	1.00	0.96	0.98	450
8	0.95	0.98	0.97	450
9	1.00	0.99	0.99	480
10	1.00	0.99	0.99	660
11	0.98	0.95	0.97	420
12	1.00	0.98	0.99	690
13	0.99	1.00	1.00	720
14	1.00	1.00	1.00	270
15	0.94	1.00	0.97	210
16	1.00	1.00	1.00	150
17	1.00	0.99	1.00	360
18	0.99	0.96	0.98	390
19	0.71	1.00	0.83	60
20	0.85	1.00	0.92	90
21	0.96	0.72	0.82	90
22	0.98	0.89	0.93	120
23	0.97	1.00	0.98	150
24	0.97	0.93	0.95	90
25	0.97	0.99	0.98	480
26	0.99	1.00	0.99	180
27	0.97	0.93	0.95	60
28	0.96	0.97	0.97	150
29	0.76	1.00	0.86	90
30	0.85	0.72	0.78	150
31	1.00	0.97	0.99	270
32	0.95	1.00	0.98	60
33	0.98	0.99	0.98	210
34	1.00	1.00	1.00	120
35	1.00	0.99	0.99	390
36	0.98	0.98	0.98	120
37	0.97	1.00	0.98	60
38	1.00	1.00	1.00	690
39	0.99	0.96	0.97	90
40	1.00	0.97	0.98	90
41	1.00	1.00	1.00	60
42	0.98	1.00	0.99	90
accuracy			0.98	12630
macro avg	0.96	0.97	0.97	12630
weighted avg	0.98	0.98	0.98	12630

Visual Network Test:

25 random images of the dataset were put to the model test, and we observe how they are classified


Actual=16 || Pred=16
Actual=1 || Pred=1
Actual=38 || Pred=38
Actual=33 || Pred=33
Actual=11 || Pred=11
Actual=38 || Pred=38
Actual=18 || Pred=18
Actual=12 || Pred=12
Actual=25 || Pred=25
Actual=35 || Pred=35
Actual=12 || Pred=12
Actual=7 || Pred=7
Actual=23 || Pred=23
Actual=7 || Pred=7
Actual=4 || Pred=4
Actual=9 || Pred=9
Actual=21 || Pred=21
Actual=20 || Pred=20
Actual=27 || Pred=27
Actual=38 || Pred=38
Actual=4 || Pred=4
Actual=33 || Pred=33
Actual=9 || Pred=9
Actual=3 || Pred=3
Actual=1 || Pred=1

Hyper - Parameter Tuning

We tried to fine tuning the hyperparameters in order to improve performance using the Keras library called “keras Tuner”.

This process was only carried out for Mnist, as it was faster to train, alternatively on AlexNet it would have been more or less the same steps.
The optimized parameters were:
the number of training epochs, the learning rate, and the number of units in the dense layers of the network.
All other parameters remained unchanged
This are the result of the search:

```
Best epoch: 37

The optimal number of units in the first densely-connected layer is 288

the optimal learning rate for the optimizer is 0.001
```

Evaluate Hyper Mnist Model :

As can be seen from the graphs, performance has improved slightly, which is due to fine-tuning.
So you can see that it is a very useful step, especially in situations with very large networks
Here are the results:

SECOND PART - CNN USING VCC19

We used the VGG19 architecture to test a pre-trained network on our dataset. For leveraging the network we exploited fine-tuning: we instantiated the vgg19 convolutional base and unfreezed the last 4 layers to perform fine-tuning on them.

COMPARISON : CNN from Scratch VS CNN with VCC19