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 project aims to create an image classifier that recognizes traffic signs , given an image.
This type of system is particularly useful in applications of self-driving cars or driver assistance in driving, a phenomenon that has been evolving in recent decades.
In fact, this specific application is part of ADAS systems that are more generally able to detect, for example, speed limits, but also access and overtaking bans.
In addition to increasing safety in the vehicle, they are very useful in emergency situations.
At the same rate as the human eye , these systems are much more responsive; in fact we talk about a recognition speed of 28 ms (35 fps on the most modern phones using Yolo v3), compared to the speed of the human eye which is 40 ms (25 fps) .
Speaking of accuracy , it stands at 98% , similar to the human eye , which can make mistakes due to low lighting, fog, excessive speed or fatigue , while an ADAS system is always alert, and in low light and/or blurry conditions can rely on filters that improve feature extraction.

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 YOLO.

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

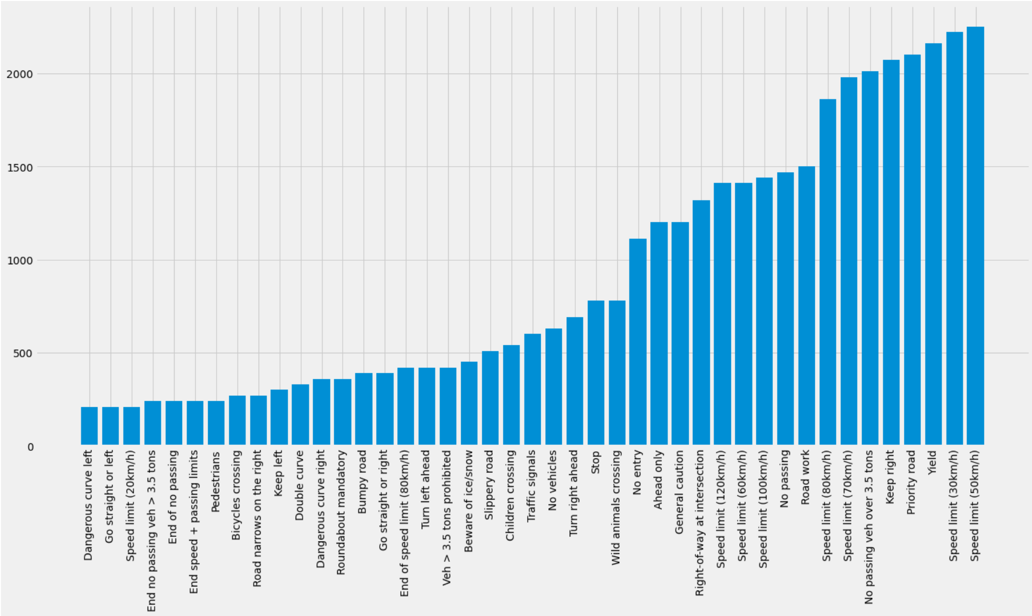
- More than 50,000 images in total
- Large, lifelike database

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 training 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 organized 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 it 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.

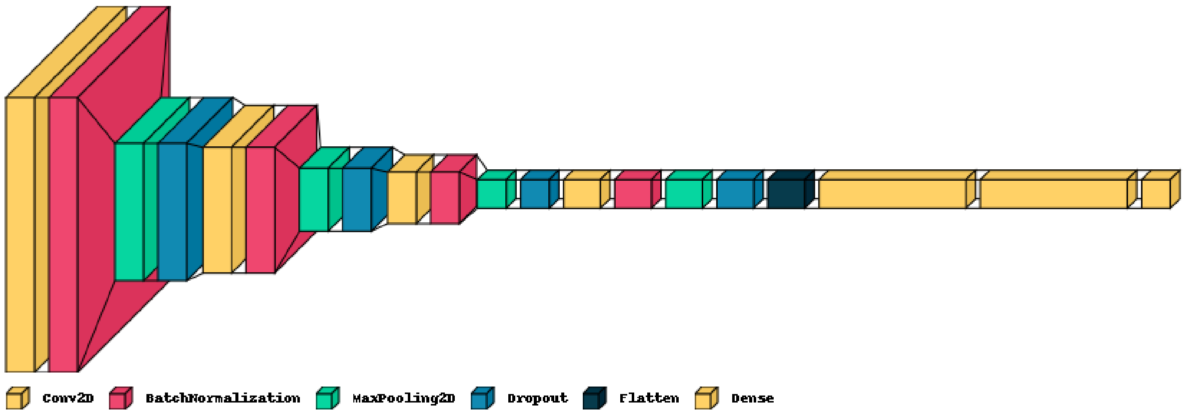
Dense Layer:

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

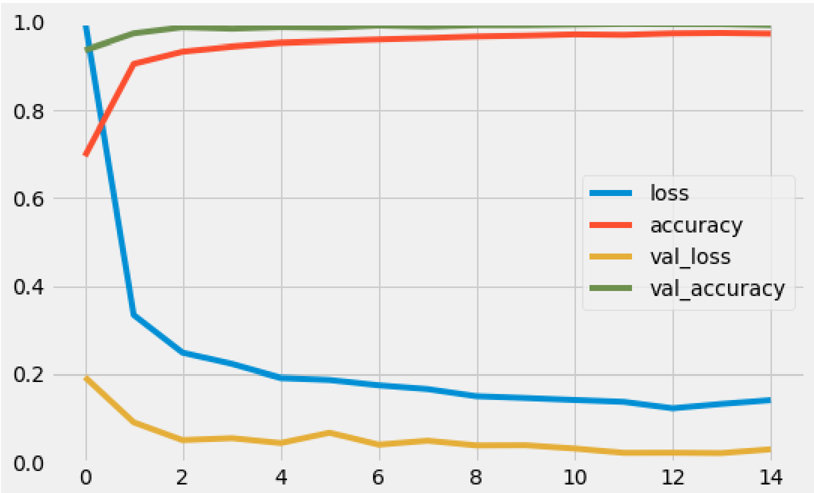
Early stopping callback function:

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 also set 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 :**



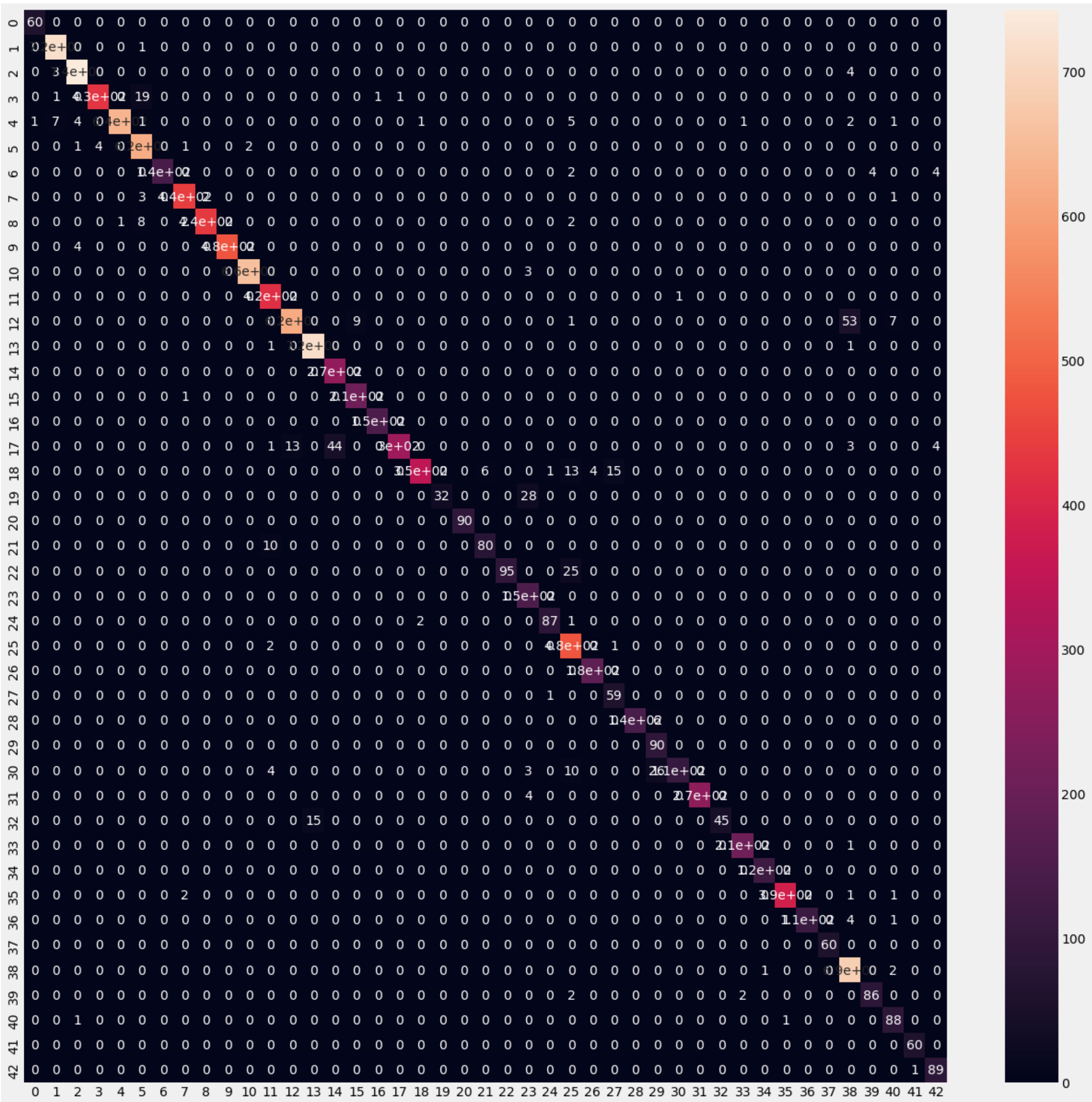
We can see that the model is not overfitting , because we don't have a low error in the training set and a higher error in the testing set.
An attempt was made to combat overfitting by inserting Dropout layers in the model and implementing augmentation of the training set and the early stopping condition.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

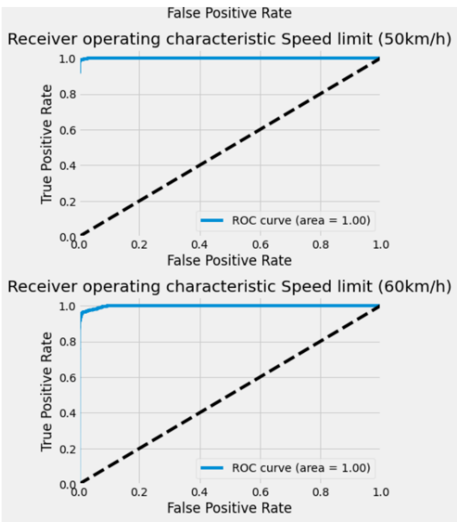


Report of the model:
For a more detailed report on each class, this graph shows the precision/recall and f1 measure for each model class.

	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

ROC Curves (only 2 shown for example):



Develop the MNIST like Model based architecture:

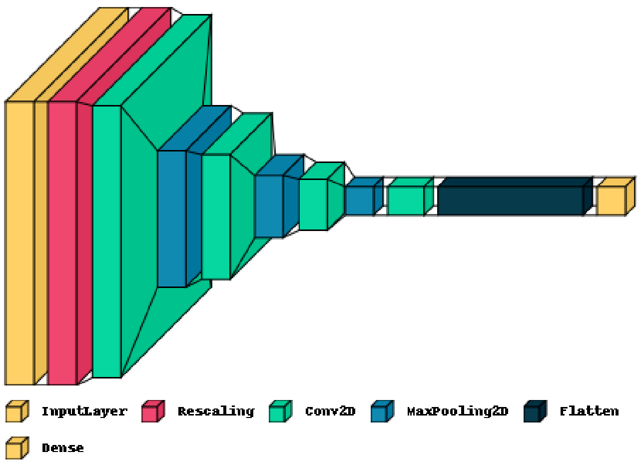
Architecture:

We will reuse the same general structure of MNIST: our convnet will be a stack of alternated Conv2D (with ReLu activation function) 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 stages. 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.

We have a final dropout layer of probability = 0.5 to reduce the overfitting.

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
```

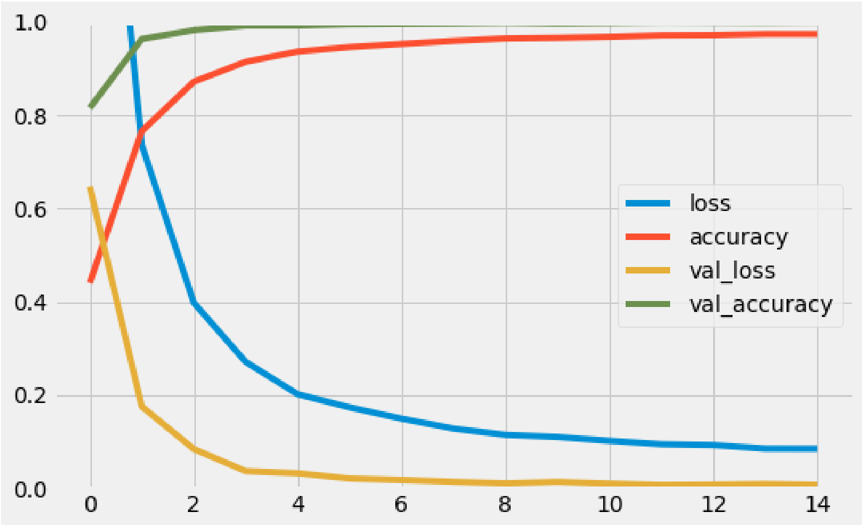
Training Phase:

```
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 :**



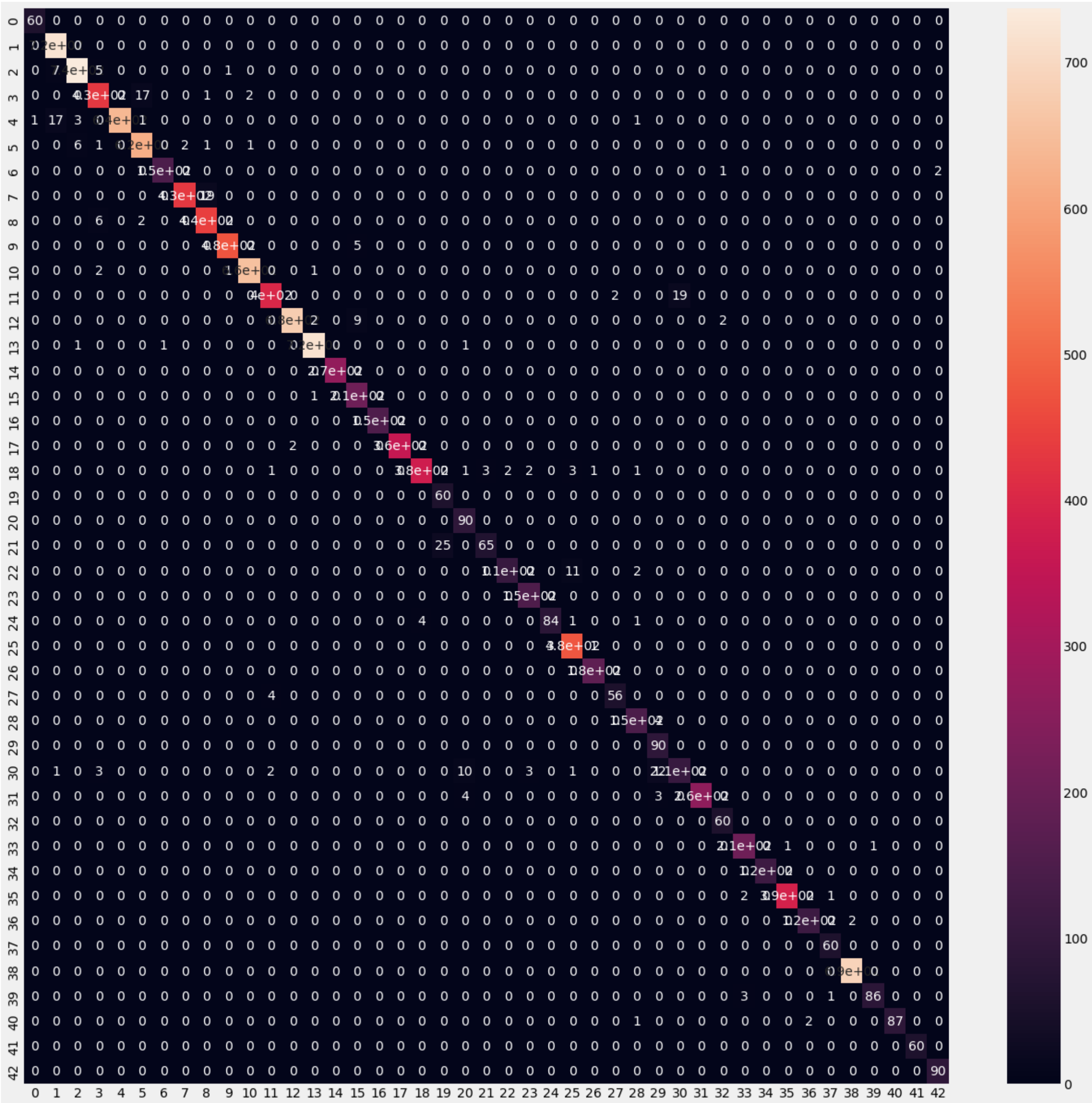
We can see that the model is not in overfitting , because don't have a low error in the training set and a higher error in the testing set.
An attempt was made to combat overfitting by inserting Dropout layers in the model and implementing augmentation of the training set.

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:

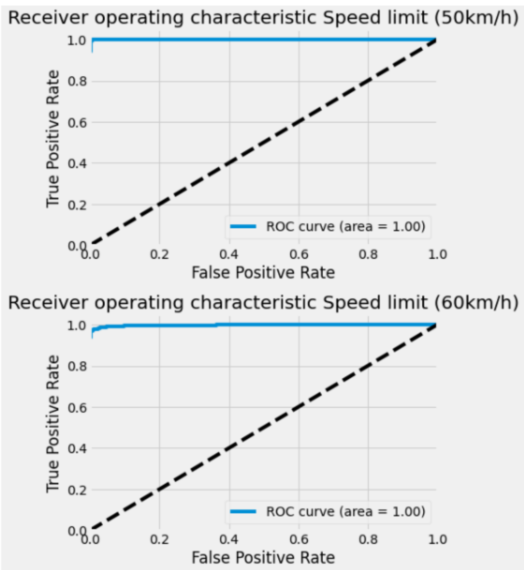
For a more detailed report on each class, this graph shows the precision/recall and f1 measure for each model class.

	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

ROC Curves (only 2 shown for example):



Hyper - Parameter Tuning

We tried to fine-tune 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
```

Applying fine tuning to the mnist-based model led to a performance improvement.
So we can see that it is a very useful step, especially in situations with very large networks

Related works

The results found on the internet regarding the GTSRB dataset, showed similar performance to ours.
On kaggle many of the papers analyzed are superficial on some aspects such as balancing training, or comparing multiple classifiers.
While on github more complete works are present:

1	**CNN with 3 Spatial Transformers**	99.71%	×	Deep neural network for traffic sign recognition systems: An analysis of spatial transformers and stochastic optimisation methods			2018
2	**Sill-Net**	99.68%	×	Sill-Net: Feature Augmentation with Separated Illumination Representation			2021
3	**MicronNet** (fp16)	98.9%	×	MicronNet: A Highly Compact Deep Convolutional Neural Network Architecture for Real-time Embedded Traffic Sign Classification			2018
4	**SEER** (RegNet10B)	90.71%	✓	Vision Models Are More Robust And Fair When Pretrained On Uncurated Images Without Supervision			2022

SECOND PART - CNN USING VCC19

Approach

We used the VGG19 architecture to test a pre-trained network on our dataset through transfer learning. VGG19 is a convolutional neural network (CNN) trained on the ImageNet dataset for image classification. It consists of 19 layers and has been trained to recognize 1000 different classes of objects. For leveraging the network we exploited both feature extraction and fine-tuning.
For the first approach we instantiated the VGG19 convolutional base and then we applied a densely connected classifier on top of it.
We first exploited fast feature extraction without data augmentation: we applied the preprocessing of VGG19 to the input data and then predicted with the convolutional base, then used the densely connected classifier to train, validate and test the model.

```
Model: "vgg19"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 50, 50, 3)]       0

 block1_conv1 (Conv2D)       (None, 50, 50, 64)        1792

 block1_conv2 (Conv2D)       (None, 50, 50, 64)        36928

 block1_pool (MaxPooling2D)  (None, 25, 25, 64)        0

 block2_conv1 (Conv2D)       (None, 25, 25, 128)       73856

 block2_conv2 (Conv2D)       (None, 25, 25, 128)       147584

 block2_pool (MaxPooling2D)  (None, 12, 12, 128)       0

 block3_conv1 (Conv2D)       (None, 12, 12, 256)       295168

 block3_conv2 (Conv2D)       (None, 12, 12, 256)       590080

 block3_conv3 (Conv2D)       (None, 12, 12, 256)       590080

 block3_conv4 (Conv2D)       (None, 12, 12, 256)       590080

 block3_pool (MaxPooling2D)  (None, 6, 6, 256)         0

 block4_conv1 (Conv2D)       (None, 6, 6, 512)         1180160

 block4_conv2 (Conv2D)       (None, 6, 6, 512)         2359808

 block4_conv3 (Conv2D)       (None, 6, 6, 512)         2359808

 block4_conv4 (Conv2D)       (None, 6, 6, 512)         2359808

 block4_pool (MaxPooling2D)  (None, 3, 3, 512)         0

 block5_conv1 (Conv2D)       (None, 3, 3, 512)         2359808

 block5_conv2 (Conv2D)       (None, 3, 3, 512)         2359808

 block5_conv3 (Conv2D)       (None, 3, 3, 512)         2359808

 block5_conv4 (Conv2D)       (None, 3, 3, 512)         2359808

 block5_pool (MaxPooling2D)  (None, 1, 1, 512)         0

=================================================================
Total params: 20,024,384
Trainable params: 20,024,384
Non-trainable params: 0
_________________________________________________________________
```

We also use feature extraction with a data augmentation layer, freezing all the layers before doing that, and applying the VGG19 convolutional base directly to our model (also applying input scale with *vgg19.preprocess_input*).

```
#freezing all the layers
conv_base.trainable = False
#Setting trainable to False empties the list of trainable weights of the layer or model
print("Trainable weights after freezing", len(conv_base.trainable_weights))

conv_base.summary()
```

```
Trainable weights after freezing 0
Model: "vgg19"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 50, 50, 3)]       0

 block1_conv1 (Conv2D)       (None, 50, 50, 64)        1792

 block1_conv2 (Conv2D)       (None, 50, 50, 64)        36928

 block1_pool (MaxPooling2D)  (None, 25, 25, 64)        0

 block2_conv1 (Conv2D)       (None, 25, 25, 128)       73856

 block2_conv2 (Conv2D)       (None, 25, 25, 128)       147584

 block2_pool (MaxPooling2D)  (None, 12, 12, 128)       0

 block3_conv1 (Conv2D)       (None, 12, 12, 256)       295168

 block3_conv2 (Conv2D)       (None, 12, 12, 256)       590080

 block3_conv3 (Conv2D)       (None, 12, 12, 256)       590080

 block3_conv4 (Conv2D)       (None, 12, 12, 256)       590080

 block3_pool (MaxPooling2D)  (None, 6, 6, 256)         0

 block4_conv1 (Conv2D)       (None, 6, 6, 512)         1180160

 block4_conv2 (Conv2D)       (None, 6, 6, 512)         2359808

 block4_conv3 (Conv2D)       (None, 6, 6, 512)         2359808

 block4_conv4 (Conv2D)       (None, 6, 6, 512)         2359808

 block4_pool (MaxPooling2D)  (None, 3, 3, 512)         0

 block5_conv1 (Conv2D)       (None, 3, 3, 512)         2359808

 block5_conv2 (Conv2D)       (None, 3, 3, 512)         2359808

 block5_conv3 (Conv2D)       (None, 3, 3, 512)         2359808

 block5_conv4 (Conv2D)       (None, 3, 3, 512)         2359808

 block5_pool (MaxPooling2D)  (None, 1, 1, 512)         0

=================================================================
Total params: 20,024,384
Trainable params: 0
Non-trainable params: 20,024,384
_________________________________________________________________
```

For exploiting fine-tuning, we instantiated the vgg19 convolutional base and unfreezed the last 5 layers, then applied the densely connected classifier on top of the network (same model as the feature extraction with data augmentation case). It's not convenient to fine tune all the layers; earlier layers are more generic,while higher layers are more specialized; also if we fine tune too many layers we risk overfitting.

```
[ ] # Freezing all layers until the fifth from the last
    conv_base.trainable = True
    for layer in conv_base.layers[:-5]:
     layer.trainable = False
```

```
[ ] conv_base.summary()
```

```
Model: "vgg19"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 50, 50, 3)]       0

 block1_conv1 (Conv2D)       (None, 50, 50, 64)        1792

 block1_conv2 (Conv2D)       (None, 50, 50, 64)        36928

 block1_pool (MaxPooling2D)  (None, 25, 25, 64)        0

 block2_conv1 (Conv2D)       (None, 25, 25, 128)       73856

 block2_conv2 (Conv2D)       (None, 25, 25, 128)       147584

 block2_pool (MaxPooling2D)  (None, 12, 12, 128)       0

 block3_conv1 (Conv2D)       (None, 12, 12, 256)       295168

 block3_conv2 (Conv2D)       (None, 12, 12, 256)       590080

 block3_conv3 (Conv2D)       (None, 12, 12, 256)       590080

 block3_conv4 (Conv2D)       (None, 12, 12, 256)       590080

 block3_pool (MaxPooling2D)  (None, 6, 6, 256)         0

 block4_conv1 (Conv2D)       (None, 6, 6, 512)         1180160

 block4_conv2 (Conv2D)       (None, 6, 6, 512)         2359808

 block4_conv3 (Conv2D)       (None, 6, 6, 512)         2359808

 block4_conv4 (Conv2D)       (None, 6, 6, 512)         2359808

 block4_pool (MaxPooling2D)  (None, 3, 3, 512)         0

 block5_conv1 (Conv2D)       (None, 3, 3, 512)         2359808

 block5_conv2 (Conv2D)       (None, 3, 3, 512)         2359808

 block5_conv3 (Conv2D)       (None, 3, 3, 512)         2359808

 block5_conv4 (Conv2D)       (None, 3, 3, 512)         2359808

 block5_pool (MaxPooling2D)  (None, 1, 1, 512)         0

=================================================================
Total params: 20,024,384
Trainable params: 9,439,232
Non-trainable params: 10,585,152
_________________________________________________________________
```

Parameters

Here we highlight some of the parameters used for the network in the different cases.

Loss function

The loss function we decided to use is the categorical cross entropy.

Optimizers

As optimizer we tried *rmsprop* and the *adam* optimizer:

- RMSProp: RMSprop is an optimization algorithm that can be used to train neural networks. It is a variant of stochastic gradient descent that uses moving averages of the squared gradients to scale the learning rates of each parameter. The idea behind RMSprop is to divide the learning rate for a weight by a running average of the magnitudes of the recent gradients for that weight, which can help to prevent the learning rate from becoming too large and causing the optimization process to diverge.
- Adam: The Adam optimizer is a popular choice for training neural networks because it is computationally efficient, has good convergence properties, and can be used with little hyperparameter tuning. Adam is a variation of stochastic gradient descent. It uses moving averages of the parameters to provide an running estimate of the second raw moments of the gradients; the term adaptive in the name refers to the fact that the algorithm "adapts" the learning rates of each parameter based on the historical gradient information. This can help the optimization process converge more quickly and with better stability compared to other optimization algorithms.

We also used the *ReduceLROnPlateau* callback to reduce the learning rate of the model since we noticed the validation loss didn't not improve for a number of epochs. This can be useful since we needed to prevent overfitting, as a constantly decreasing learning rate can help the model to converge on a solution; also this can help the model to find a good balance between underfitting and overfitting, as it allows the model to continue learning at a slower pace when the validation loss plateaus.
Here we didn't exploit early stopping; in some of our tests we noticed that the training stopped too early so we wanted to see the full training.

Batch Normalization layer

Batch normalization is a technique used to normalize the inputs of a layer in a neural network. It helps to stabilize the learning process and improve the generalization ability of the model. During training, the mean and standard deviation of the activations of a layer are calculated for each mini-batch and used to normalize the activations. This helps to ensure that the distribution of the activations remains consistent, even as the weights of the network are updated. This can help to reduce the internal covariate shift, which is the change in the distribution of the activations of a layer due to the change in the weights.

Dropout layer

We used a Dropout layer for regularization in order to reduce overfitting; we tried different values for this layer and at the end we chose a small value because the size of the dataset was already large enough for our network so if we increased it the performances would decrease; in fact we observed that with larger value for Dropout layers we had a lower training accuracy but also a lower validation accuracy, so it didn't help preventing overfitting. The value we used after some experiments is 0.2.

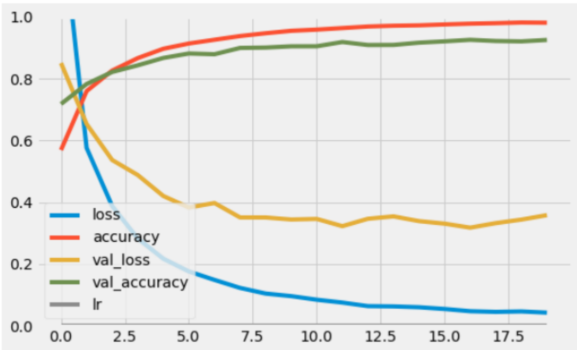
Dense Layer

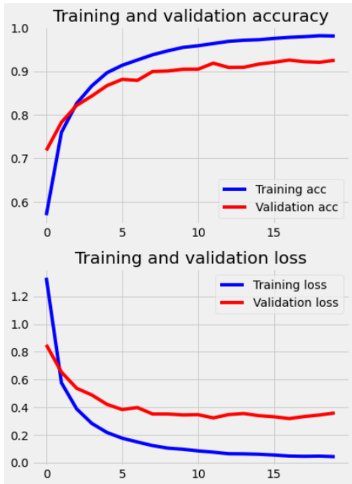
In the densely connected classifier built on top of the VGG19 convolutional base,we used the ReLU activation function in the hidden layers. The ReLU (Rectified Linear Unit) activation function is defined as: $f(x) = \max(0, x)$. This means that the output of the ReLU function is the input value x if x is positive, and 0 if x is negative. The ReLU activation function has the advantage of being computationally efficient, as well as having a non-saturating gradient, which can facilitate the training process. In the hidden layers of the densely connected classifier, the ReLU activation function can help introduce non-linearity into the model, which can improve its ability to learn complex patterns in the data. Therefore before the output layer, which has a softmax activation function, we decided to use fully connected (dense) layers with the ReLU activation function. The number of input units in this layer was chosen experimentally, starting with low values and at the end we decided to use 1024 units.

Performances

For each approach we are going to show the training phase plots, the test performances and two of the roc curves as an example. The accuracy on training and validation sets with fast feature extraction were the following:

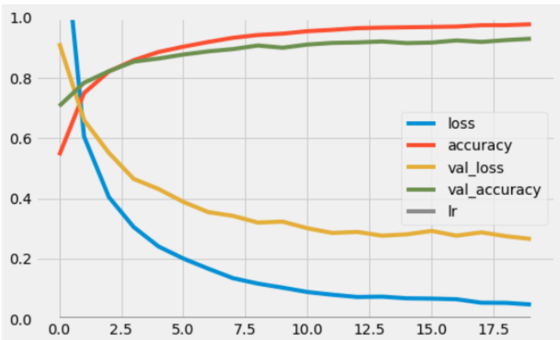
- ***RMSprop optimizer***

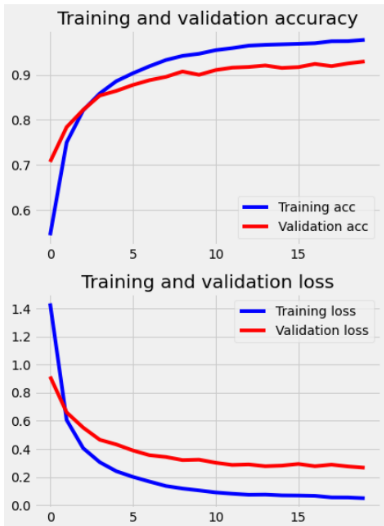





```
- 1s 7ms/step - loss: 0.0431 - accuracy: 0.9808 - val_loss: 0.3584 - val_accuracy: 0.9251 - lr: 0.0010
```

- ***Adam optimizer***



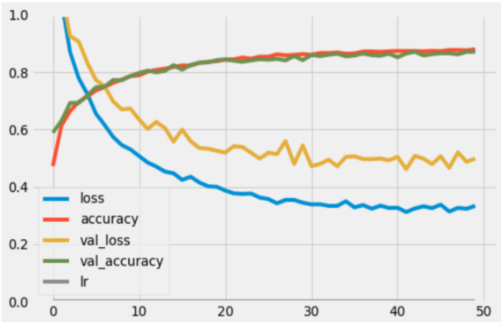


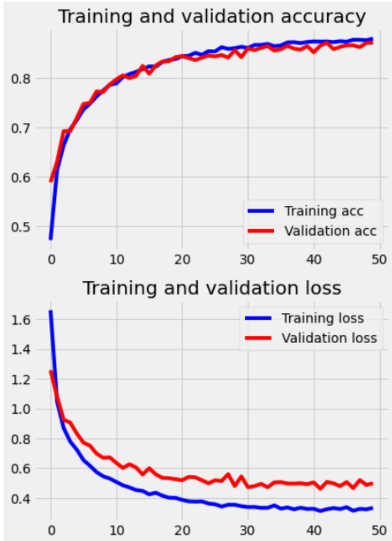

```
- 1s 5ms/step - loss: 0.0487 - accuracy: 0.9778 - val_loss: 0.2658 - val_accuracy: 0.9297 - lr: 0.0010
```

Here we don’t insert the test performances since it’s a solution without data augmentation so the performances are not that good.

The accuracy on training and validation sets with fast feature extraction were the following:

- *RMSprop optimizer*

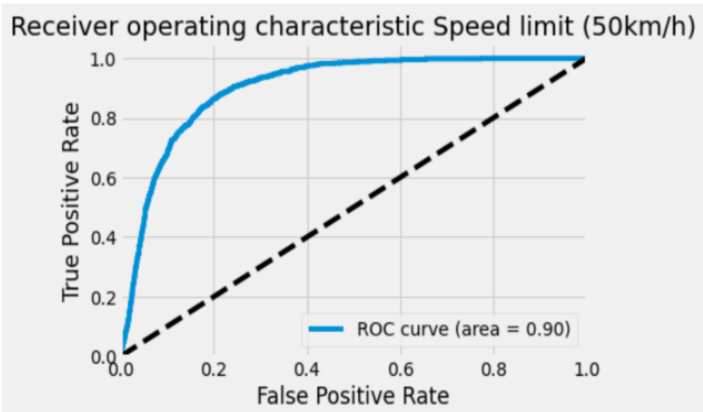


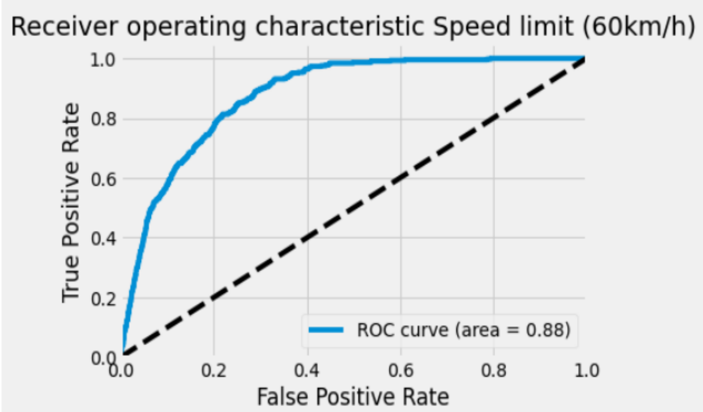



```
loss: 0.3338 - accuracy: 0.8794 - val_loss: 0.4988 - val_accuracy: 0.8711 - lr: 0.0010
```

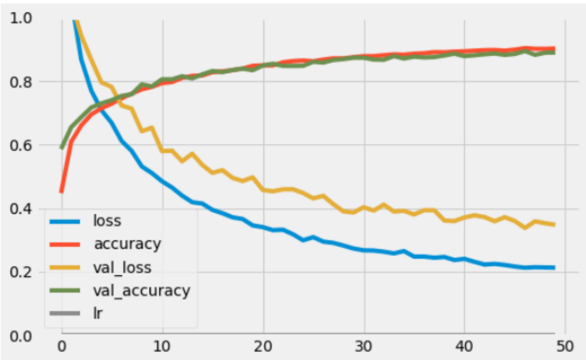
Test set performances:

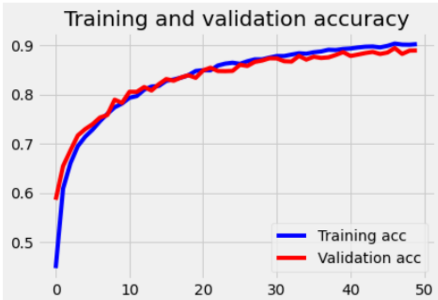
```
395/395 [==============================] - 6s 15ms/step
Test Data accuracy:  54.53681710213777
```





- *Adam optimizer*

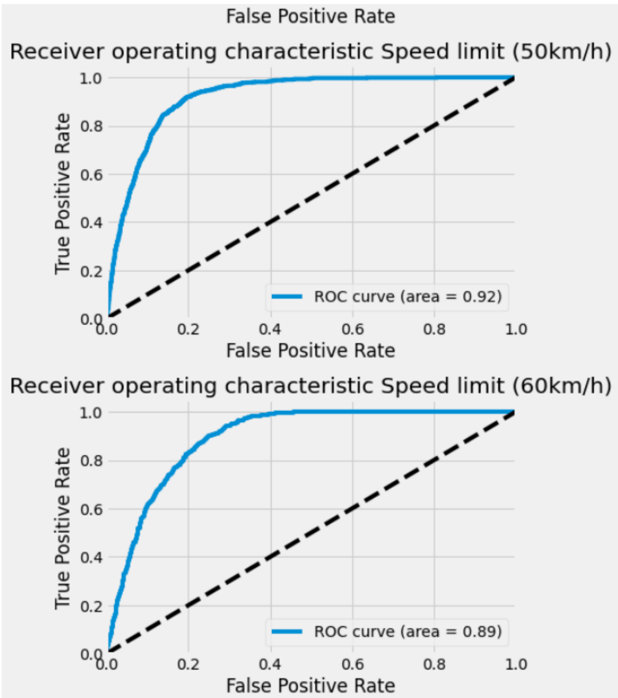





```
- loss: 0.2131 - accuracy: 0.9023 - val_loss: 0.3480 - val_accuracy: 0.8893 - lr: 0.0010
```

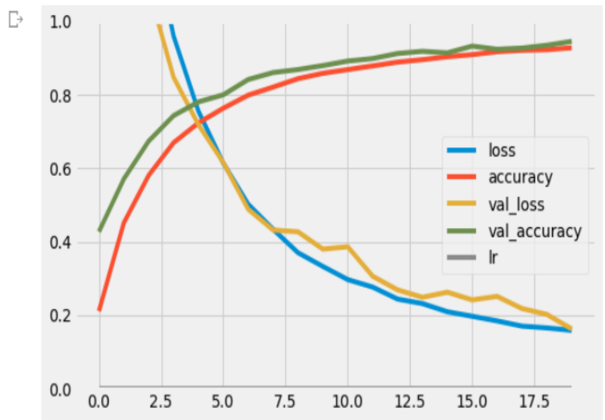
Test set performances:

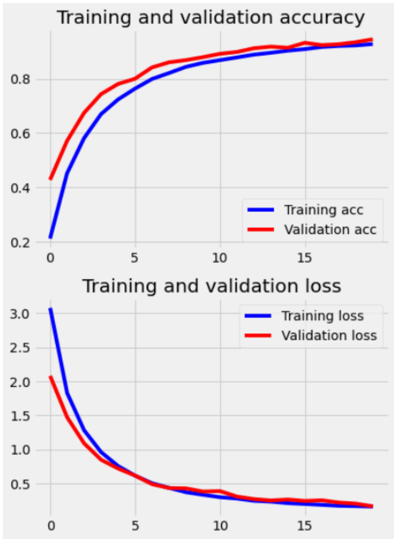
```
395/395 [==============================] - 6s 15ms/step
Test Data accuracy:  53.34125098970704
```



The accuracy on training and validation sets with fast feature extraction were the following:

- *RMSprop optimizer*

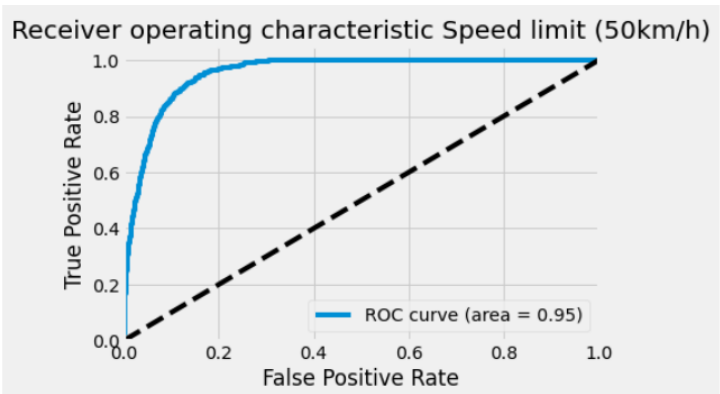


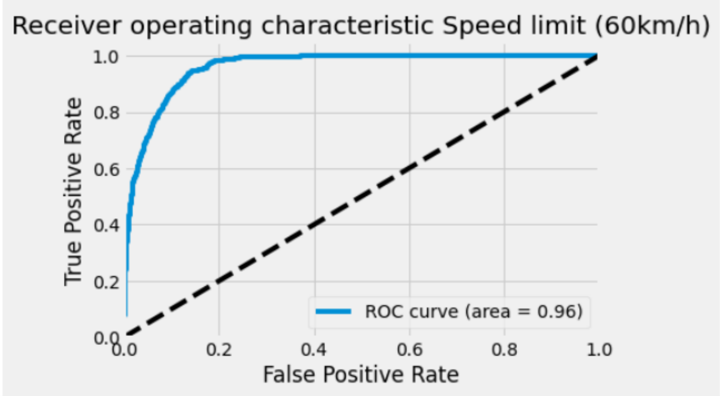



```
- 71s 72ms/step - loss: 0.1591 - accuracy: 0.9277 - val_loss: 0.1630 - val_accuracy: 0.9452 - lr: 1.0000e-05
```

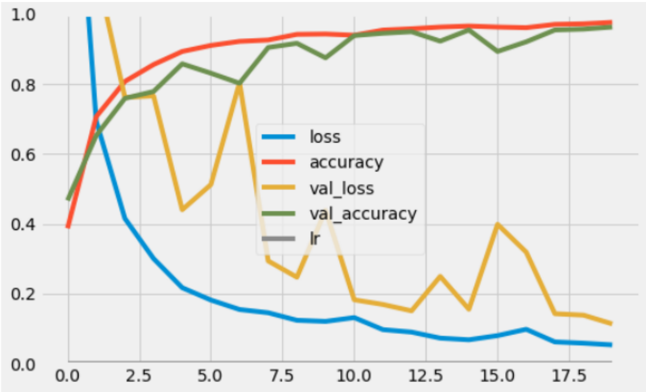
Test set performances:

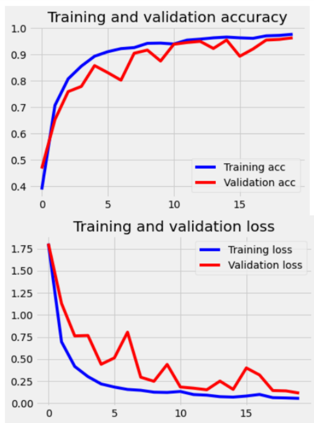
```
395/395 [==============================] - 12s 28ms/step
Test Data accuracy:  76.70625494853523
```





- *Adam optimizer*

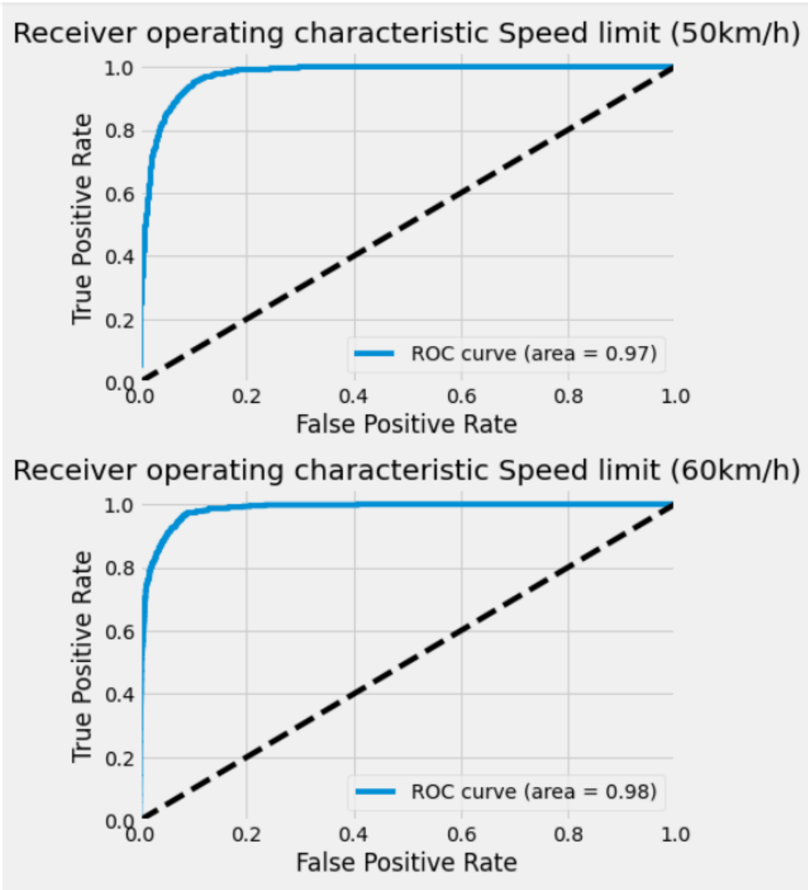





```
- loss: 0.0541 - accuracy: 0.9763 - val_loss: 0.1141 - val_accuracy: 0.9629 - lr: 0.0010
```

Test set performances:

```
395/395 [==============================] - 6s 15ms/step
Test Data accuracy:  82.32779097387174
```



COMPARISON : CNN from Scratch VS CNN with VCC19

As we can see from the previously shown performances, the highest training accuracy we obtained with the pretrained VGG19 model was with the fast feature extraction approach, but it also was the most sensitive to overfitting and so the test performances were not satisfying; this is also true for the feature extraction with data augmentation.
Even though it is faster to train, feature extraction was not a good approach for this task with the dataset we tried to use.
With fine tuning we obtained slightly better performances; using the RMSprop optimizer we obtained a decent accuracy, but with the Adam optimizer we improved the performances even though, as we can see from the validation loss, it was more sensitive to overfitting. For the classification purpose, as we can see from the roc curves (as explained earlier we showed only two classes for making this comparison), the fine tuning with the Adam optimizer gave the best results in exploiting VGG19.

Our conclusions are that within the tested approach the best one for this task are the ones with the AlexNet and the Mnist based architectures; we could try to improve the approach through transfer learning using different pre-trained models and trying a different test set or a different data augmentation layer. From the experiments we did, changing the hyperparameters with vgg19 led to slight changes in the training and test accuracy, but it is possible to try to improve them by performing hyperparameter optimization.

YOLO

One of the initial objectives of our project was to implement Yolo in order to try to solve our image classification task. But due to time limitations and other issues we didn't manage to fully develop this approach.
In fact, we encountered two main problems in the training phase:

1. The labels had to have an edge, and this is usually done with external programmes such as CVAT, LabelImg, VoTT in manual mode, so it would have taken a long time to create a sufficient number of images for our training, validation and test sets.

2. It is generally better to have images of the entire environment rather than just the object itself, especially if you want the model to be able to classify the object in a realistic setting. This is because the context and background of the image can provide valuable information for the model to make an accurate classification.
For example, if you are trying to classify cars in images, it would be helpful for the model to see the car in the context of a street or a parking lot, rather than just a close-up of the car itself. This is because the model can use the surrounding context to understand the shape, size, and appearance of the car, as well as its relationship to other objects in the scene.
So our database was not good, and it was necessary to take a dataset manually or to take screens on google maps.

CONCLUSIONS

Image classification is a task that can be carried out with decent performances exploiting CNNs, like it's vastly described in the literature. With our project we wanted to show how CNNs can be used to correctly classify german traffic signs even though the quality of the image is not that high, and we have seen that there are different approaches that we can use to correctly classify them.

We also highlighted different problems that we can encounter trying the different possible solutions that we analyzed.

We can conclude that it would be interesting for this task to try to improve the performances with different techniques, different models and different approaches, in order to see how much performances can be improved with respect to the currently most adopted approaches for this kind of task.