UNIVERSITÀ DI PISA

Intelligent Systems

Convolutional neural network for traffic signs classification

Tommaso Bertini

Giovanni Marrucci

ANNO ACCADEMICO 2022/2023

Contents

1 Introduction **1**

2 Object detection overview **1**
2.1 Limitations 1
2.2 Road sign detection 1

3 Dataset **2**
3.1 Chinese dataset processing 2
3.2 Italian dataset processing 3
3.3 Final processing and augmentation 4

4 CNN from scratch **5**
4.1 First model 5
4.2 Second model - Augmentation 6
4.3 Third model - Dropout 9
4.4 Fourth model - Adding Layers 11
4.5 Fifth model - Adaptive Learning Rate 13

5 Pre-trained CNN **15**

6 Comparison between fifth model and VGG19 **17**

7 Conclusions **19**

1 Introduction

In the ever-evolving field of computer vision and deep learning, the development and evaluation of neural network models continue to be at the forefront of innovation. Our project represents an exploration into the world of Convolutional Neural Networks (CNNs) with a specific focus on road sign classification.

The primary objective of this project was to design and implement a CNN from scratch capable of classifying road signs into four distinct categories.

Secondly, we harnessed the capabilities of a pretrained CNN model, which are designed for general-purpose image classification, originally developed and fine-tuned on extensive image datasets, to assess its performance in comparison to our custom model. The analysis encompassed aspects such as model accuracy, training efficiency, and adaptability to the nuances of road sign classification.

2 Object detection overview

2.1 Limitations

Because of the limitations of Google Colab, creating an architecture from scratch capable of performing multiple object detection was not feasible, because it could have taken too many resources in terms of training time and capacity of the network. The idea was instead to perform single object detection, where the assumption is that the images fed to the network contain a single traffic sign, and for each each sign there was a txt file with the coordinate of the bounding box.

2.2 Road sign detection

To perform this task, instead of relying on a complex object detection architecture, we adopted a simple multi-headed CNN to perform feature extraction, with a regression head in charge of locating the sign inside the image (outputting the coordinates of the bounding boxes) and a standard classification head to perform classification.
The boxes generated by the first are in the YOLO format (x_center, y_center, width, height), while the latter generates a number between 0 and 3 corresponding to four different classes of road signs (we will discuss about them later).

3 Dataset

We used two dataset for this project:

- **Chinese traffic signs** composed by 57 classes.
  Available at this link: `https://www.kaggle.com/datasets/dmitryyemelyanov/chinese-traffic-signs`
- **Italian traffic signs** composed by 87 classes.
  Available at this link: `https://github.com/marcomoauro/traffic-sign-dataset`

For our neural network model, we decided to simplify the classification by using the **four main categories** of road signs: *mandatory*, *prohibitory*, *warning*, and *indication* signs. Upon reviewing the datasets, we observed that the number of samples for indication signs was significantly lower compared to the other three classes. Consequently, we decided to replace the indication class with the *speed limit* class. Thus, our dataset will be divided into the following classes with these identification codes:

- **Mandatory** $\Rightarrow$ 0
- **Prohibition** $\Rightarrow$ 1
- **Speed Limit** $\Rightarrow$ 2
- **Warning** $\Rightarrow$ 3

In addition to the images, there are also files containing coordinates for the bounding boxes of traffic signs.

(Note: "Bounding box" is a technical term that refers to an imaginary rectangle that surrounds an object, often used in computer vision to define the position of objects within an image.)

3.1 Chinese dataset processing

This dataset was clean and almost every image had only one sign in it. The only problem that we encountered was with the format of the bounding boxes. These were represented as follows: (x1, y1, x2, y2) , but we needed them to be in the YOLO format. Therefore we extracted the coordinates from the csv file, converted and saved them in a txt file with the name equal to the image it belonged to.
The images were divided, as previously mentioned, in 58 categories. To identify the photos' classes, the name of each has a number from 000 to 057. We decided to take the ones with the following numbers:

- **Mandatory**: 020-024, 026, 027, 030
- **Prohibition**: 054-055
- **Speed Limit**: 002-007
- **Warning**: 032, 034-051

After this division we simply added to each txt the identification code of the class it belonged to.

Figure 1: Raw image

3.2 Italian dataset processing

In this dataset, we have many images with multiple road signs, and consequently, their respective annotation files contain multiple bounding boxes. To create a simpler model, we have opted to crop the images with the following constraints:

- Each image should feature only one road sign.
- The crop should encompass a random portion of the background surrounding the bounding box
- The cropped regions should have similar dimensions.
- Cropped images with dimensions less than 50px are excluded.

Figure 2: Raw image

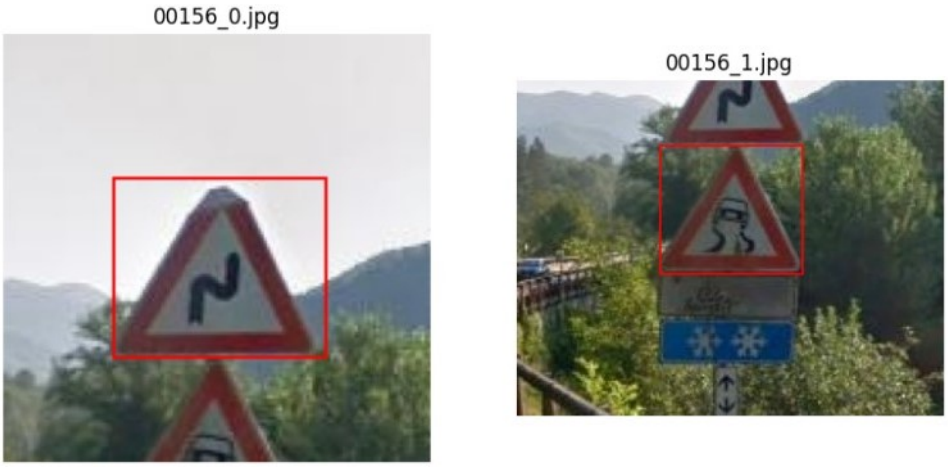


Figure 3: Cropped images

3.3 Final processing and augmentation

The bounding boxes in this dataset are in YOLO format, which means they have coordinates normalized relative to the image size. From the example below, we can see that the values contained in the text file represent the sign class, the center coordinates, the width, and the height, respectively.

Here an example of the content in a txt file:

```
3 0.746134 0.298828 0.090206 0.230469
```

The number of samples that we had available were the following:

- mandatory: 1070
- prohibition: 1149
- speed limit: 1498
- warning: 1815

As it can be seen, we had a different number of images for each category. This wasn't optimal for the training of the neural network. Therefore we opted for the usage of data augmentation. To do so we used the library "Albumentations" which let us scale, rotate and translate the images in various ways. Using this technique we balanced all the classes to have 2000 images each.

4 CNN from scratch

To implement the following we built a stack of convolutional layers and pooling layers with the purpose of creating the architecture for the feature extraction. At the end of these layers we added two heads: classification_head and regression_head. The first one is in charge of classifying the sign, while the latter is in charge of calculating the position of the regression box in the image. As we will see each model is better than the previous one,this was achieved by adding layers and modifying hyper-parameters.

4.1 First model

The first experiment we made consisted of a simple convolutional neural network with two convolutional layers and two pooling layers, a flatten layer and the two output heads. The losses used are the categorical cross entropy for the classification head and the mean squared error on the regression head. Below is shown the summary of the model:

Layer (type)	Output Shape	Param #	Connected to
input_2 (InputLayer)	[(None, 64, 64, 3)]	0	[]
conv2d_2 (Conv2D)	(None, 63, 63, 16)	208	['input_2[0][0]']
max_pooling2d_2 (MaxPooling2D)	(None, 31, 31, 16)	0	['conv2d_2[0][0]']
conv2d_3 (Conv2D)	(None, 30, 30, 32)	2080	['max_pooling2d_2[0][0]']
max_pooling2d_3 (MaxPooling2D)	(None, 15, 15, 32)	0	['conv2d_3[0][0]']
flatten_1 (Flatten)	(None, 7200)	0	['max_pooling2d_3[0][0]']
classification (Dense)	(None, 4)	28804	['flatten_1[0][0]']
regression (Dense)	(None, 4)	28804	['flatten_1[0][0]']

Total params: 59896 (233.97 KB)
Trainable params: 59896 (233.97 KB)
Non-trainable params: 0 (0.00 Byte)

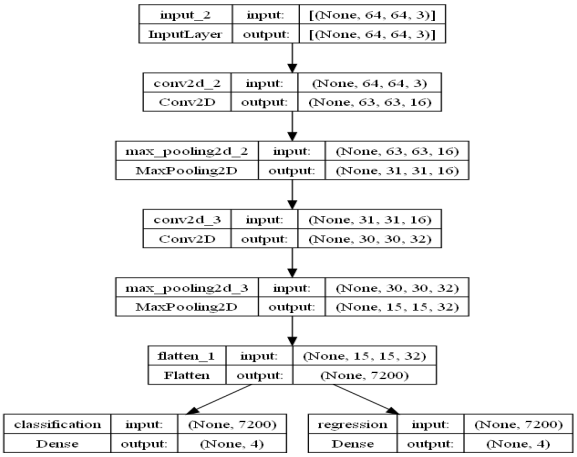
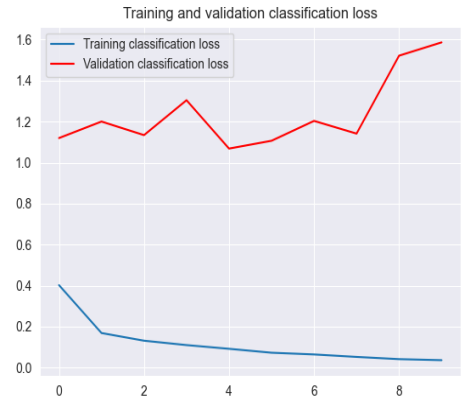


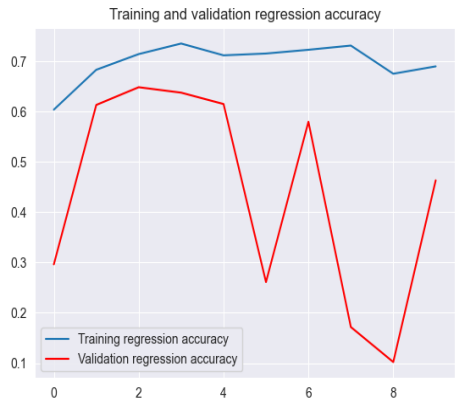
Figure 4: Model summary

The network was trained for 10 epochs and showed very good results both for labeling and detecting correctly the images of the training set, reaching an almost 100% accuracy for the classification and 70% for the regression. On the other hand it performed poorly on the validation set in both cases, probably due to overfitting.
This is plausible since we used the dataset without the augmentation of the images. Furthermore we didn't divide equally the images of each class in validation and training, leading to an unbalanced input. Below are shown the accuracy and regression graphs:

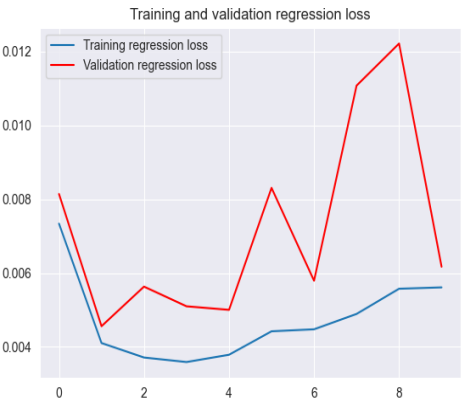
(a) Classification accuracy



(b) Classification loss



(c) Regression accuracy



(d) Regression loss

Figure 5: First model graphs

4.2 Second model - Augmentation

After observing that the performance of our initial model was far from our ideal values on the validation side, one of the first approaches we adopted was the use of data augmentation. Data augmentation is a technique used to artificially increase the size of a dataset by applying various transformations to the existing data. These transformations can include random rotations, flips, scaling, cropping, and other image manipulations. Using data augmentation we expected to have a better dataset, which means enhancing the diversity of the data, making the model more robust.

With this approach, as we wrote before, we expanded our dataset to 2000 samples per class. We split the samples in training, validation, and test sets, choosing a 80/10/10 ratio.

It's important to underline that the data generated through various augmentation transformations wasn't used for the validation and test sets, but exclusively for the training data. Therefore, for each class, we randomly selected 200 samples from the not augmented dataset for both the validation and test sets. Augmented samples were added to the remaining samples until reaching 1600 samples for the training set.
We used the same structure as the first model to show how a well-tought dataset can make a difference. Below is shown, again, the summary of the model:

Layer (type)	Output Shape	Param #	Connected to
input_2 (InputLayer)	[(None, 64, 64, 3)]	0	[]
conv2d_2 (Conv2D)	(None, 63, 63, 16)	208	['input_2[0][0]']
max_pooling2d_2 (MaxPooling2D)	(None, 31, 31, 16)	0	['conv2d_2[0][0]']
conv2d_3 (Conv2D)	(None, 30, 30, 32)	2080	['max_pooling2d_2[0][0]']
max_pooling2d_3 (MaxPooling2D)	(None, 15, 15, 32)	0	['conv2d_3[0][0]']
flatten_1 (Flatten)	(None, 7200)	0	['max_pooling2d_3[0][0]']
classification (Dense)	(None, 4)	28804	['flatten_1[0][0]']
regression (Dense)	(None, 4)	28804	['flatten_1[0][0]']

Total params: 59896 (233.97 KB)
Trainable params: 59896 (233.97 KB)
Non-trainable params: 0 (0.00 Byte)

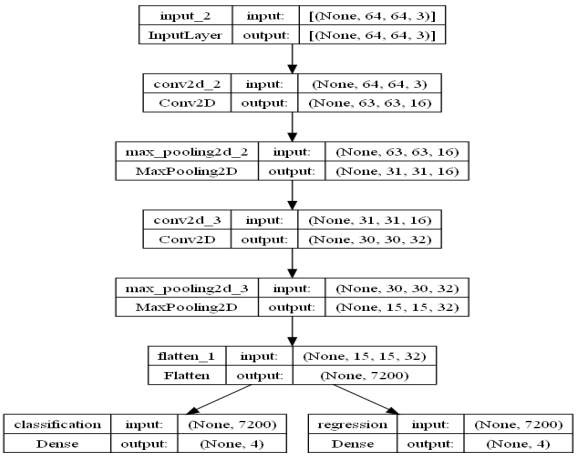
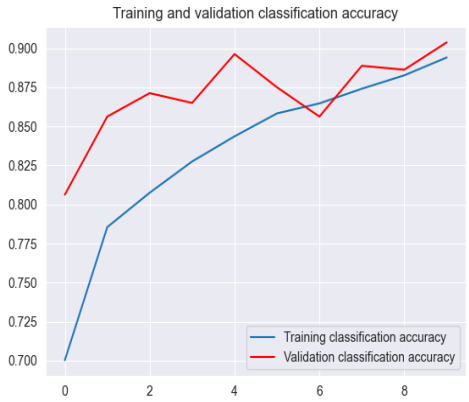
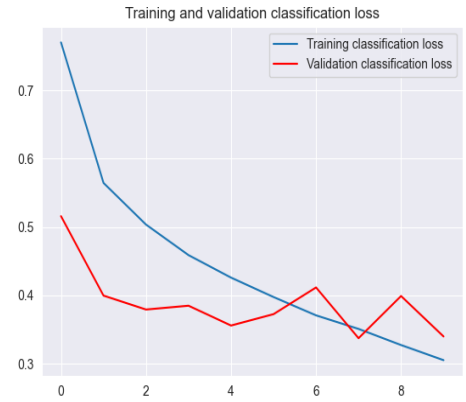


Figure 6: Model summary

The network was trained for 10 epochs and showed good results both for labeling and detecting correctly the images of the training set. These were not as good as the previous model for the classification but slightly better from a regression point of view, reaching 90% accuracy for the classification and 73% for the regression. For the validation set we had good improvements in both cases, but still some uncertainty on the results both from an accuracy and loss point of view.
As expected the results significantly improved for the validation set. Anyways we can clearly see some fluctuation of the values, especially for the regression head. To fix it we will gradually improve the CNN with some techniques.
Below are shown the accuracy and regression graphs:



(a) Classification accuracy



(b) Classification loss

(c) Regression accuracy

(d) Regression loss

Figure 7: Second model graphs

Here, we have presented the confusion matrix for the test set.

Figure 8: Confusion Matrix

4.3 Third model - Dropout

In the third experiment, we introduced a modification to the model by incorporating the dropout layer. This adjustment was made in response to the observed fluctuations, which in the previous model. The aim was to add regularization and improve model generalization. Moreover, a dropout layer can reduce overfitting which is one of the causes of the fluctuations.
Below is shown the summary of the model:

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 64, 64, 3)]	0	[]
conv2d (Conv2D)	(None, 63, 63, 16)	208	['input_1[0][0]']
max_pooling2d (MaxPooling2D)	(None, 31, 31, 16)	0	['conv2d[0][0]']
conv2d_1 (Conv2D)	(None, 30, 30, 32)	2080	['max_pooling2d[0][0]']
max_pooling2d_1 (MaxPooling2D)	(None, 15, 15, 32)	0	['conv2d_1[0][0]']
flatten (Flatten)	(None, 7200)	0	['max_pooling2d_1[0][0]']
dropout (Dropout)	(None, 7200)	0	['flatten[0][0]']
classification (Dense)	(None, 4)	28804	['dropout[0][0]']
regression (Dense)	(None, 4)	28804	['dropout[0][0]']

Total params: 59896 (233.97 KB)
Trainable params: 59896 (233.97 KB)
Non-trainable params: 0 (0.00 Byte)

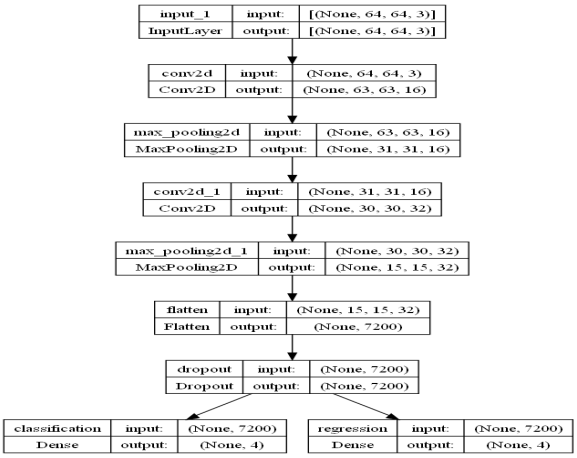
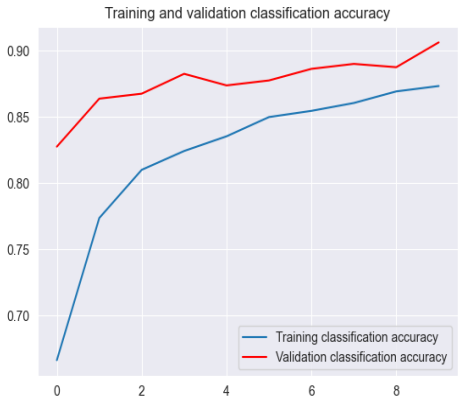
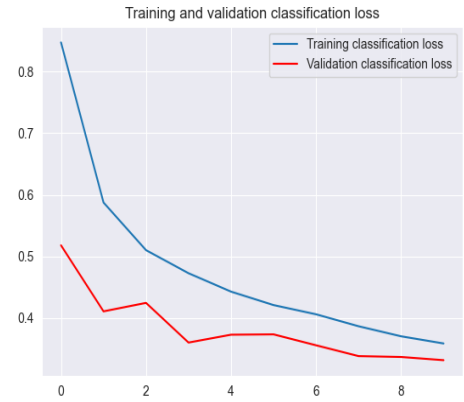


Figure 9: Model summary

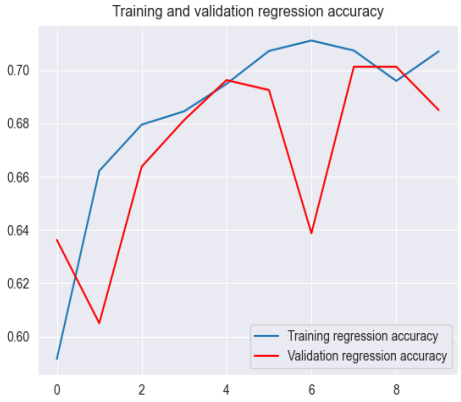
Keras suggests dropout layer values ranging from 0.5 to 0.8. We experimented with training our neural network model using dropout rates of 0.5, 0.65, and 0.8, comparing their performance and selecting the one that performed best for our CNN.
After comparing the performance of the three scenarios, we observed that our neural model performed better when using a dropout value of 0.5. Therefore, we retained this configuration for subsequent tests. As we can see the curves fluctuate a lot less and we have the same performance for the training set with a significant improvement on the validation side.
Below, we have provided the graphs depicting the best configuration of the model:



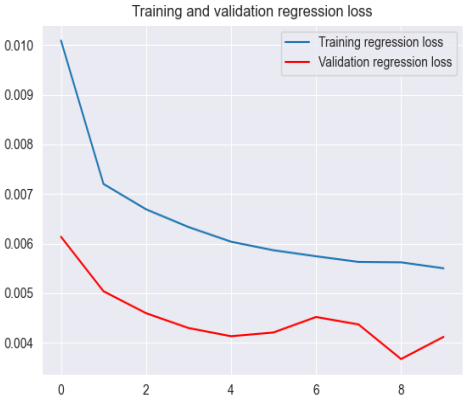
(a) Classification accuracy



(b) Classification loss



(c) Regression accuracy



(d) Regression loss

Figure 10: Third model graphs

Here, we have presented the confusion matrix for the test set.

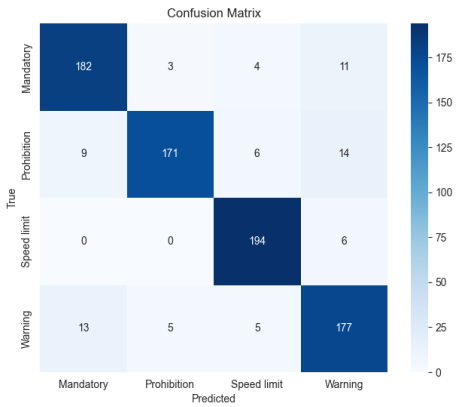


Figure 11: Confusion Matrix

4.4 Fourth model - Adding Layers

As a fourth approach, we increased the complexity of the network to enhance training. We are aware that increasing the complexity of a neural network model allows it to extract more complex features and, therefore, improves its learning capacity. However, excessive complexity can weigh down the model and increase the risk of overfitting. Therefore, we opted to add just one additional convolutional block with a higher number of filters in the main structure. Moreover, we add some complexity to the regression head adding 2 dense layers.
Below is shown the summary of the model:

Layer (type)	Output Shape	Param #	Connected to
input_2 (InputLayer)	[(None, 64, 64, 3)]	0	[]
conv2d_3 (Conv2D)	(None, 63, 63, 16)	208	['input_2[0][0]']
max_pooling2d_3 (MaxPooling2D)	(None, 31, 31, 16)	0	['conv2d_3[0][0]']
conv2d_4 (Conv2D)	(None, 30, 30, 32)	2080	['max_pooling2d_3[0][0]']
max_pooling2d_4 (MaxPooling2D)	(None, 15, 15, 32)	0	['conv2d_4[0][0]']
conv2d_5 (Conv2D)	(None, 14, 14, 64)	8256	['max_pooling2d_4[0][0]']
max_pooling2d_5 (MaxPooling2D)	(None, 7, 7, 64)	0	['conv2d_5[0][0]']
flatten_1 (Flatten)	(None, 3136)	0	['max_pooling2d_5[0][0]']
dropout_1 (Dropout)	(None, 3136)	0	['flatten_1[0][0]']
dense_2 (Dense)	(None, 64)	200768	['dropout_1[0][0]']
dense_3 (Dense)	(None, 32)	2080	['dense_2[0][0]']
classification (Dense)	(None, 4)	12548	['dropout_1[0][0]']
regression (Dense)	(None, 4)	132	['dense_3[0][0]']

Total params: 226072 (883.09 KB)
Trainable params: 226072 (883.09 KB)
Non-trainable params: 0 (0.00 Byte)

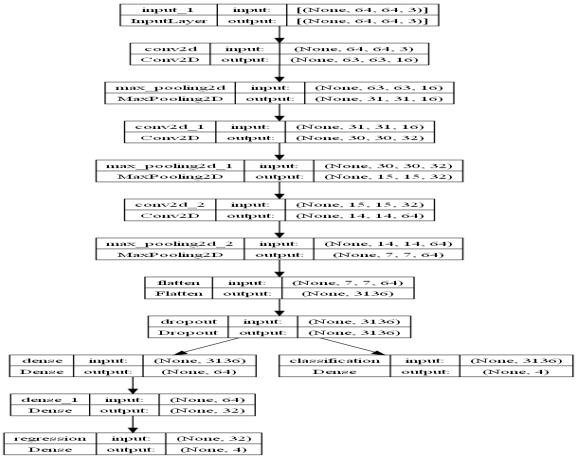
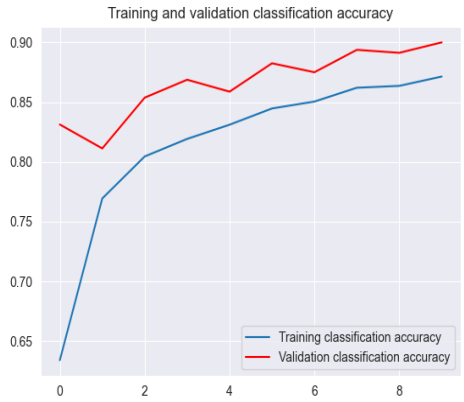
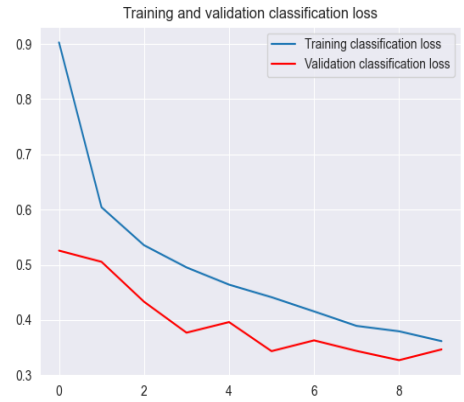


Figure 12: Model summary

Observing the graphs we can see a big improvement for all the curves, especially for the regression accuracy and loss.
Below are shown the accuracy and regression graphs:



(a) Classification accuracy



(b) Classification loss

(c) Regression accuracy

(d) Regression loss

Figure 13: Fourth model graphs

Here, we have presented the confusion matrix for the test set.

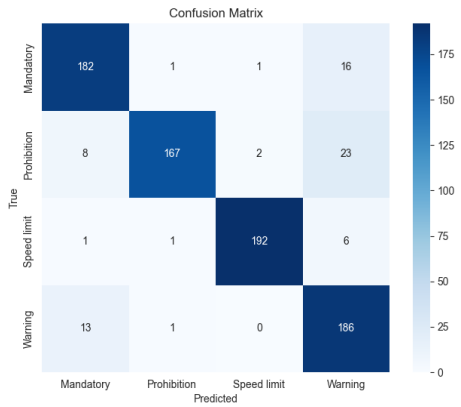


Figure 14: Confusion Matrix

4.5 Fifth model - Adaptive Learning Rate

As a fifth and final approach, we opted to apply an adaptive learning rate, specifically using the **Adam optimizer** because it is one of the most commonly employed optimizers for CNN classification tasks. The aim of Adam optimization is to adjust the learning rate adaptively for each parameter in the model based on the history of gradients calculated for that parameter. This helps the optimizer converge faster and more accurately than fixed learning rate methods.
Additionally, we reduced the batch size from 32 to 16 samples to increase the network's adaptability and reduce even more the risk of overfitting. (This was tested and verified performing a training in both the situations)
Since we only modified the previously mentioned hyperparameters the structure of the network is the same as before:

Layer (type)	Output Shape	Param #	Connected to
input_7 (InputLayer)	[(None, 64, 64, 3)]	0	[]
conv2d_18 (Conv2D)	(None, 63, 63, 16)	208	['input_7[0][0]']
max_pooling2d_18 (MaxPooling2D)	(None, 31, 31, 16)	0	['conv2d_18[0][0]']
conv2d_19 (Conv2D)	(None, 30, 30, 32)	2080	['max_pooling2d_18[0][0]']
max_pooling2d_19 (MaxPooling2D)	(None, 15, 15, 32)	0	['conv2d_19[0][0]']
conv2d_20 (Conv2D)	(None, 14, 14, 64)	8256	['max_pooling2d_19[0][0]']
max_pooling2d_20 (MaxPooling2D)	(None, 7, 7, 64)	0	['conv2d_20[0][0]']
flatten_6 (Flatten)	(None, 3136)	0	['max_pooling2d_20[0][0]']
dropout_6 (Dropout)	(None, 3136)	0	['flatten_6[0][0]']
dense_12 (Dense)	(None, 64)	200768	['dropout_6[0][0]']
dense_13 (Dense)	(None, 32)	2080	['dense_12[0][0]']
classification (Dense)	(None, 4)	12548	['dropout_6[0][0]']
regression (Dense)	(None, 4)	132	['dense_13[0][0]']

Total params: 226072 (883.09 KB)
Trainable params: 226072 (883.09 KB)
Non-trainable params: 0 (0.00 Byte)

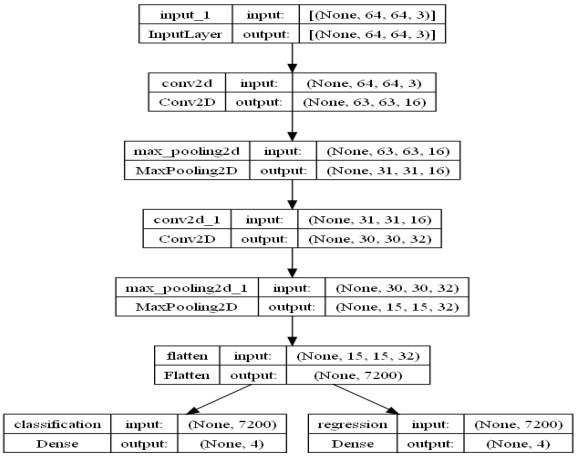
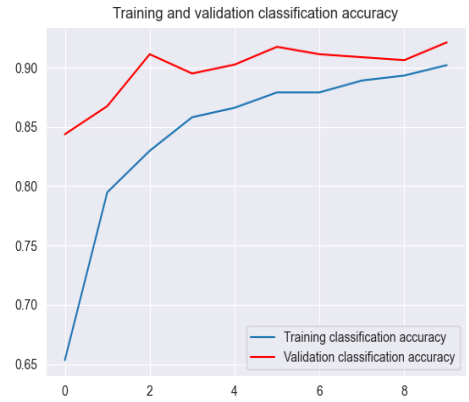
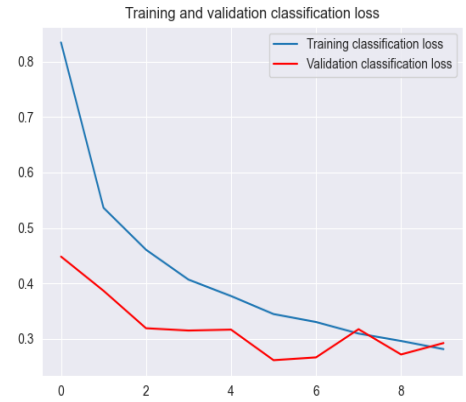


Figure 15: Model summary

In this case the general performance improved, leading to a better accuracy for both the heads of classification and regression. These hyperparameters improved the loss too, leading to a lower loss in both cases.
Below, we have included the graphs depicting the performance of this final model:



(a) Classification accuracy



(b) Classification loss

(c) Regression accuracy

(d) Regression loss

Figure 16: Fifth model graphs

Here, we have presented the confusion matrix for the test set:

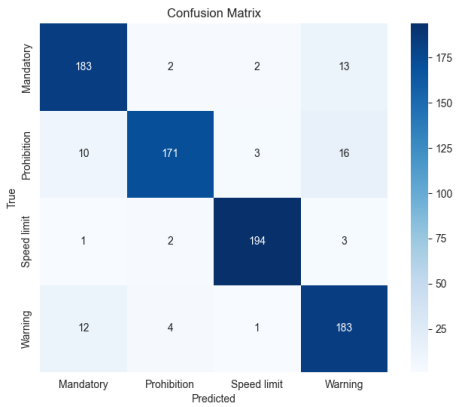


Figure 17: Confusion Matrix

5 Pre-trained CNN

For the comparison with a pretrained model we choose to use VGG19.
VGG19 is a well-established convolutional neural network architecture widely recognized for its strengths in computer vision tasks. Here are our motivations for the choice:

- **Strong Feature Extractor**: VGG19's architecture is designed with multiple convolutional layers, making it an excellent feature extractor.
- **Pretrained on Large Datasets**: it is pretrained on the ImageNet dataset, which contains millions of images from thousands of diverse classes.
- **Proven Performance**: VGG19 has a strong track record in the field of computer vision.

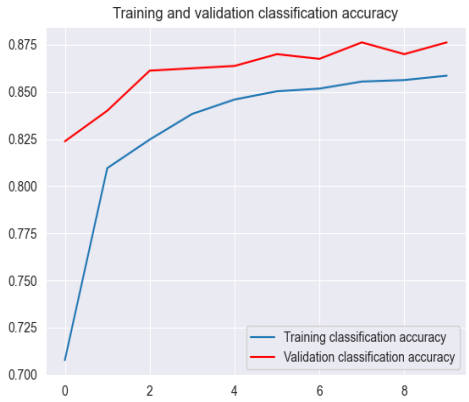
This is the summary of the model:

Layer (type)	Output Shape	Param #	Connected to
input_3 (InputLayer)	[(None, 64, 64, 3)]	0	[]
block1_conv1 (Conv2D)	(None, 64, 64, 64)	1792	['input_3[0][0]']
block1_conv2 (Conv2D)	(None, 64, 64, 64)	36928	['block1_conv1[0][0]']
block1_pool (MaxPooling2D)	(None, 32, 32, 64)	0	['block1_conv2[0][0]']
block2_conv1 (Conv2D)	(None, 32, 32, 128)	73856	['block1_pool[0][0]']
block2_conv2 (Conv2D)	(None, 32, 32, 128)	147584	['block2_conv1[0][0]']
block2_pool (MaxPooling2D)	(None, 16, 16, 128)	0	['block2_conv2[0][0]']
block3_conv1 (Conv2D)	(None, 16, 16, 256)	295168	['block2_pool[0][0]']
block3_conv2 (Conv2D)	(None, 16, 16, 256)	590080	['block3_conv1[0][0]']
block3_conv3 (Conv2D)	(None, 16, 16, 256)	590080	['block3_conv2[0][0]']
block3_conv4 (Conv2D)	(None, 16, 16, 256)	590080	['block3_conv3[0][0]']

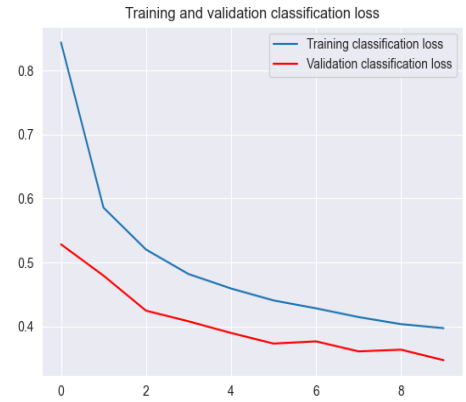
...
Trainable params: 37096 (144.91 KB)
Non-trainable params: 20024384 (76.39 MB)

Figure 18: VGG19 summary

Having said so we set the model as not trainable and we used the model.fit() to draw the classification and regression graphs. The following graphs show that our model was slightly better in all the occasions. This is probably due to the fact that almost all the weights were set.
Below, the graphs of the VGG19:



(a) Classification accuracy



(b) Classification loss

(c) Regression accuracy

(d) Regression loss

Figure 19: VGG19 graphs

Here, we have presented the confusion matrix for the test set:

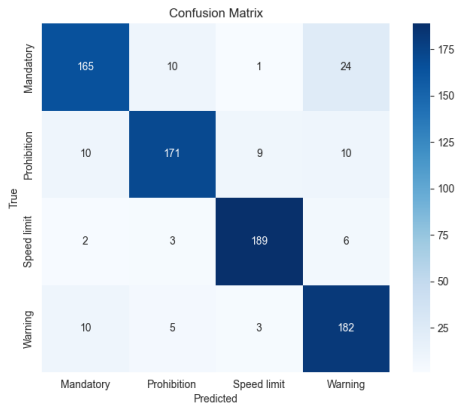
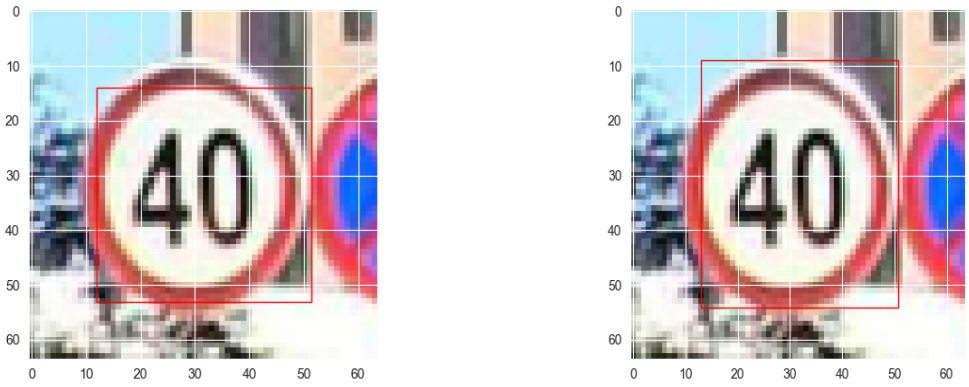


Figure 20: Confusion Matrix

6 Comparison between fifth model and VGG19

We reported a image comparison for each class.

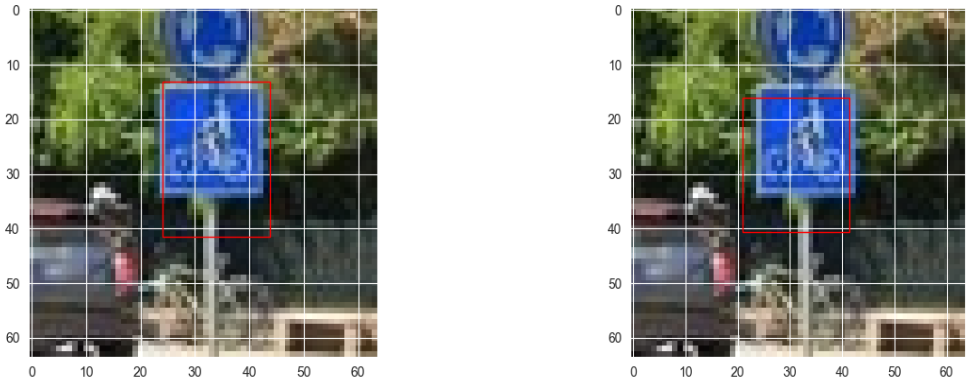
- Target class: 2
- Fifth Model predict: 2
- VGG19 predict: 2



(a) Fifth model (b) VGG19 model

Figure 21: image example 1

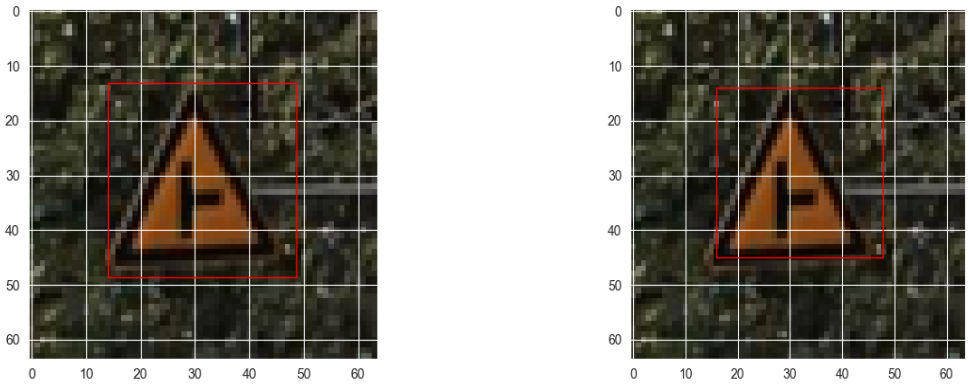
- Target class: 0
- Fifth Model predict: 0
- VGG19 predict: 3



(a) Fifth model (b) VGG19 model

Figure 22: image example 2

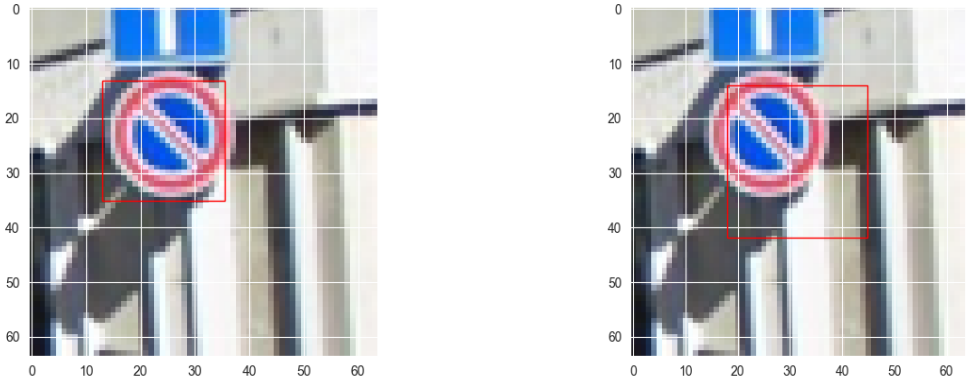
- Target class: 3
- Fifth Model predict: 3
- VGG19 predict: 3



(a) Fifth model (b) VGG19 model

Figure 23: image example 3

- Target class: 1
- Fifth Model predict: 3
- VGG19 predict: 1



(a) Fifth model (b) VGG19 model

Figure 24: image example 4

7 Conclusions

Impressively we got better results using our model than the pretrained one. This was probably due to the fact that our model was specifically made for this type of classification and detection, while the other was more general purpose.
Moreover our network was faster since the number of parameters was a lot lower than the pretrained one. The difference was highly evident during the model.fit() in which the "From scratch" network spent 6 seconds, whereas the VGG 19 almost one full minute.