University of Beira Interior

Master's in Computer Engineering

Machine Learning

Deep Learning Final Project

Authoress:
Rita Correia, M9933

Professor Dr.:
Hugo Proença

July, 2021

Contents

List of Figures

Listings

Acronyms

Adam	Adaptive Moment Estimation
CNN	Convolutional Neural Network
CPU	Central Process Unit
GPU	Graphics Processing Unit
AI	Artificial Intelligence
IDE	Integrated Development Environment
reLU	Rectified Linear Unit
RGB	Red, Green, Blue
RMSProp	Root Mean Square Propagation
SGD	Stochastic Gradient Descent
IT	Information Technologies
UBI	University of Beira Interior
UC	Curricular Unit

Chapter 1

Introduction

1.1 Motivation and Context

The Deep Learning area is related to the application of artificial neural networks in solving complex problems that expand the most advanced computational power. There are several practical applications that have already been built using these techniques, such as autonomous cars, the discovery of new medicine, early diagnosis of diseases, facial recognition, prediction of stock prices, etc. These artificial networks (mainly the CNN) seek to simulate the way the human brain works. For this reason, it is extremely important that Information Technologies (IT) professionals know how to work with these tools, since large companies use their systems, such as Airbus, eBay, Dropbox, Intel, IBM, Uber, Twitter, Snapchat and also Google itself.

Selecting and implementing a practical problem where a Deep Learning classifier can be used to distinguish elements from different classes, was the challenge proposed in this project to students attending the Curricular Unit (UC) of Machine Learning present in the second semester of the first year of the Master's in Computer Engineering, at University of Beira Interior (UBI).

1.2 Objectives

This project has as main objectives:

- Manually acquire a dataset composed of at least 1000 photographs with the same size and color spectrum. Each image must be captured with the greatest possible variability of factors;
- Train and test an appropriate CNN model under the chosen dataset;
- Perform an empirical validation of the model with a test set.

Chapter 2

Convolutional Neural Networks

Convolutional neural networks, initially proposed by Yann LeCun in 1988, are the classic and most popular model of Deep Learning, being developed based on the visual cortex of animals, which is composed of millions of complex cell clusters, sensitive to small sub-regions of the visual field, called receivable fields. These regions, in convolutional neural networks, are also called receptive fields and are formed by selected subsets of the vector of characteristics (numerical representation of the object) to be analyzed. [1].

CNN are typically used in Computer Vision, a sub-area of Artificial Intelligence (AI) that is concerned with the processing of videos and images (for example in robotics projects with object recognition), and which aims to simulate on computers view that the Human Being has of the World.

In the book "*Neural Networks and Learning Machines*" [3], Simon Haykin divides the structure of convolutional networks into three main objectives:

- **Feature extraction**: each neuron receives input signals from a receptive field in the previous layer, enabling the extraction of local features. This extraction of local characteristics makes the exact position of each characteristic (or pixel, in the case of an image), irrelevant, as long as its position about neighboring characteristics is maintained [1].

  In a simpler way, the feature extraction will automatically check and select which are the main characteristics of each class (they can be, for example, colors or shapes), thus not using all the pixels of each class (or image, in this case). In an exemplary way, we can consider that we have two characters, Bart (figure 2.1) and Homer (figure 2.2), each representing, in this case, a distinct class. Characterizing both classes according to their own characteristics, we can see that Bart has an orange shirt, blue shorts and blue shoes. Homer, on the other hand, has a white shirt, blue pants (another blue value) and dark gray shoes.

Figure 2.1: Bart - first class.

Figure 2.2: Homer - second class.

In the image 2.3 we have an example of a matrix of characteristics, for the case of the two classes mentioned above, where it is possible to find the characteristics of the Bart class in the first 3 values, followed by the values corresponding to the characteristics of the Homer class, and finally, the class to which each image belongs. These values are thus extracted using a CNN (figure 2.4) and in this case, the red dots (*input layer*) are the total characteristics (about both Bart and Homer classes), the yellow neurons (represented by dots) are the hidden layers (where image processing and model training are performed) and blue neurons are related to the output layer (in this case we would only need two, as we only have 2 classes).

```
8.97,3.45,2.35,0.0,00.00,0.00,Bart
6.75,0.94,0.52,0.00,0.00,0.00,Bart
9.69,4.10,1.56,0.00,0.00,0.00,Bart
0.00,0.00,0.00,4.68,0.66,0.01,Homer
0.00,0.00,0.00,0.12,2.50,0.03,Homer
0.00,0.00,0.00,5.80,0.50,1.28,Homer
```

Figure 2.3: Characteristics matrix.

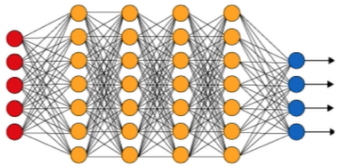

Figure 2.4: Example of CNN.

- **Characteristics mapping**: each layer of the network is composed of several matrices of characteristics (feature maps), which are regions where neurons share the same synaptic weights. These weights are called filters, and give the model robustness, making it capable of handling variations in distortion, rotation and translation in the image. Weight sharing also allows for a huge reduction in the number of parameters to be optimized [1].
- **Subsampling**: After each convolution layer, a subsampling layer is applied, which is nothing more than a collection of samples from each characteristic matrix. These samplings can be performed by obtaining the sum, taking the average and selecting the highest (max pooling) or the lowest (min pooling) value of the region under analysis [1].

The image 2.5, taken from Haykin's book [3], shows a convolutional network applied to a 28x28 pixel image, placing four 24x24 size filters in the first layer, and by making a subsampling. The process continues with the application of new filters and subsamples.

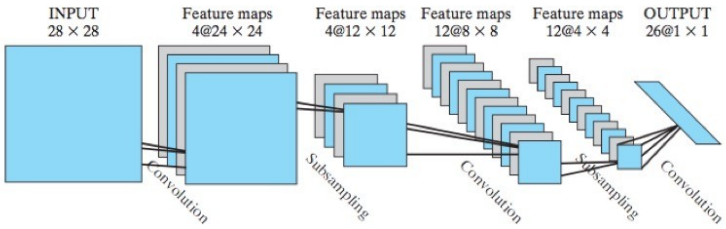

Figure 2.5: Convolutional network.

In the figure 2.6 we have an example of an image represented by a binary matrix (we can consider a black and white image), where we will apply what would be a network layer, that is, perform the convolution and subsampling. In this case, and to make the example simple to understand, we apply only one filter represented by the matrix illustrated in yellow.

We can see that each region in green is a matrix of characteristics, which is obtained by "sliding" the 3x3 filter through the image. The yellow filter is applied to each characteristic matrix, which is actually a matrix multiplication (matrix x filter). This operation is called **convolution**. Finally, to obtain each value of the new 6x6 matrix, we perform a max pooling, that is, we select the largest value of the matrix resulting from the application of the filter in the characteristics matrix [1].

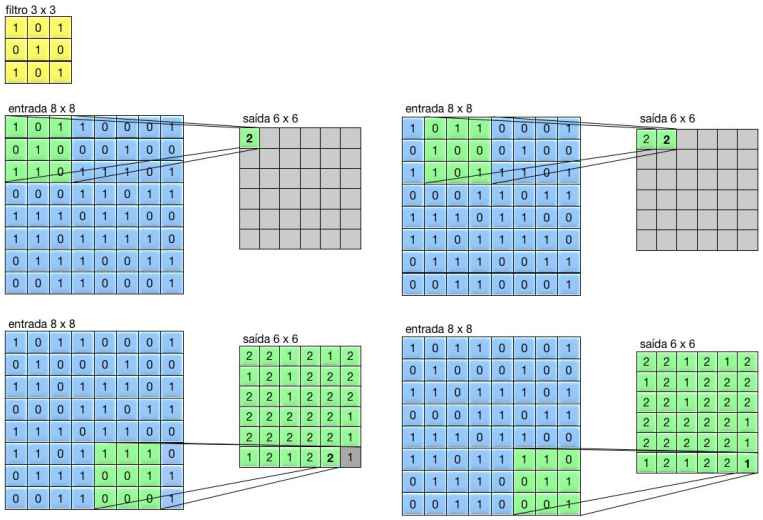

Figure 2.6: Convolution operation.

The filter values are therefore the weights of the network. These weights are learned during the training process, usually using the back-propagation technique.

For classification problems, in the last layer of the network we can apply a softmax function, in order to obtain the probability that a given image belongs to any of the possible classes present in the problem [1].

Chapter 3

Development and implementation

3.1 IDE and language

The project was developed in the *Visual Studio Code* Integrated Development Environment (IDE). This software is a text editor that offers a wide range of features. It is also multi-platform, free and aimed at those looking for a simple IDE, light but at the same time quite complete and with a wide range of extensions. The programming language used was Python, in version 3.6.10, under the Anaconda distribution.

3.2 CPU vs GPU

Initially, the Central Process Unit (CPU) of the local machine was being used to process the training of the model created through the CNN. However, each *epoch* (iteration) took more than 40 seconds to execute, which gives a total of approximately 1.5 hours for 100 iterations. Thus, the development, testing and *debug* process was a super slow and time-consuming process.

For this reason, and with the Anaconda distribution, the Graphics Processing Unit (GPU) was used, by creating a virtual environment "PythonGPU" and installing the `tensorflow` dependencies there for this purpose (*tensorflow-gpu*). It was also necessary to download and install the CUDA toolkit, which is an NVIDIA API that provides a development environment for creating high-performance applications to run on the GPU. This toolkit also includes libraries for CPU use, *debugging* and optimization tools, a C/C++ compiler and a runtime library [4].

In the figure 3.1 we can observe the use of the GPU when training the model.

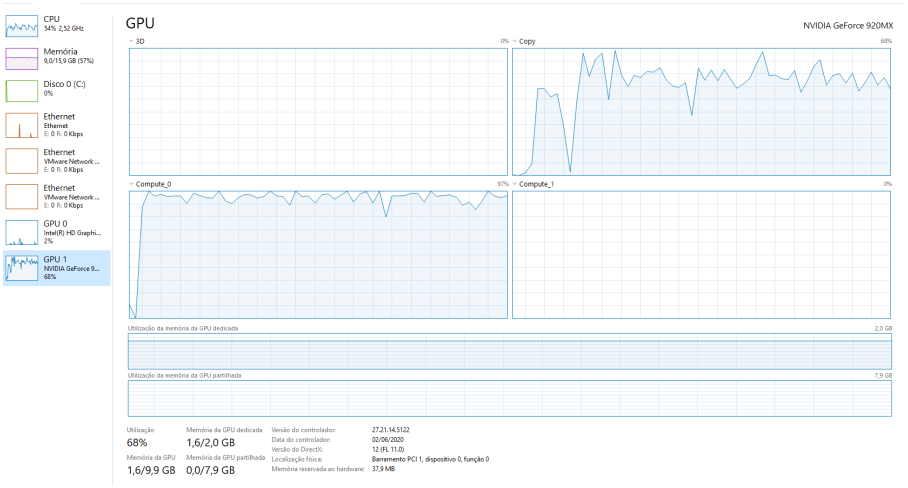

Figure 3.1: Use of GPU in training the model.

3.3 Dataset

The problem that I explored in this project is the identification and distinction between two classes composed of information and danger traffic signs, respectively.

All images were captured with the same mobile device, in the same format (.jpg), in the same color spectrum (Red, Green, Blue (RGB)) and with only one instance of the object to be distinguished by image. The photographs were also captured at different times of the day (from morning to night), in different perspectives and distances, with different degrees of sharpness, and there were also some partially cropped signs in the images.

The dataset has a total of 1604 photographs, 320 of which (about 20 % of the total) are in the test set, with the same number of photographs in both classes of signals. The remaining 1284 are present in the training set, about 20 % of which were used for validation.

In the figures 3.2 to 3.6 are shown some examples of images, referring to both classes, present in the dataset, taken from both the training and test sets.

Figure 3.2: Danger traffic sign - example 1.

Figure 3.3: Danger traffic sign - example 2.

Figure 3.4: Danger traffic sign - example 3.

Figure 3.5: Danger traffic sign - example 4.

Figure 3.6: Information traffic sign - example 1.

Figure 3.7: Information traffic sign - example 2.

Figure 3.8: Information traffic sign - example 3.

Figure 3.9: Information traffic sign - example 4.

3.4 Imports

For the implementation of the program it was necessary to make several imports (a total of 19), namely:

- **cv2**: also called OpenCV, it is an image and video processing library available in Python and many other high-level programming languages. It is used for all types of image and video analysis, such as facial recognition and detection, plate reading, photo editing, advanced robotic vision, optical character recognition and more. It was used in the project to read and resize the images;
- **matplotlib.pyplot and matplotlib.image**: the ".pyplot" is a plotting

library for Python, which can be used to plot lines, graphs and histograms. With the ".image" module it is also possible to show images;

- **os**: it is a built-in python module for accessing the local machine and its file system. It can be used to display content in directories, create new folders and exclude folders;
- **gc**: short for garbage collector, it is an important tool for manually cleaning and deleting unnecessary variables;
- **train_test_split**: it is a method of sklearn.model_selection and was used to divide the training and validation set;
- **layers**: it is a class in the `Keras` library that contains different types of layers used in deep learning, such as: Convolutional layer (the most used in computer vision), Pooling layer, Recurrent layer, Embedding layers (usually used in natural language processing), Normalization layers, among others;
- **models**: it is a class from the `Keras` library that contains two types of models: Sequential model (which we use in this project) and the model with a functional API;
- **optimizers**: it is a module from the `Keras` library that contains several optimizers, that is, types of back propagation algorithms, to train the model, such as: Root Mean Square Propagation (RMSProp), SGD, Adam , adagrad, adadelta, etc;
- **ImageDataGenerator**: it is a class from the `Keras` library that is used when working with reduced size datasets.

3.5 Separation of training and test set

To separate the dataset into the training and test sets, two variables were initially created "train_dir" and "test_dir" both containing the path for the training and testing directories, respectively. Then, with a seed of 3, 160 images of information traffic signs (20% of the total existing information traffic signs) and 160 images of danger traffic signs were selected in a pseudo-random way, as we can see in the excerpt 3.1.

```
random.seed(3)
#info signs for testing
info_moved = random.sample(glob.glob("C:\\Users\\ritac\\Desktop\\
    TP5_ML\\train\\train\\info *.jpg"), 160)
#danger signs for testing
perigo_moved = random.sample(glob.glob("C:\\Users\\ritac\\Desktop\\
    TP5_ML\\train\\train\\perigo *.jpg"), 160)
```

Listing 3.1: Separation of images for the test set.

If the test directory is empty, it will move the previously selected photos from both classes to the test set, using the `move()` method in the `shutil` library.

3.6 Image storage

As we can see in the excerpt 3.2, the program will find all the danger and information images, in the training set, saving them into 2 different variables (`train_info` and `train_perigo`). The images in the test set will also be saved in a variable called `test_imgs`).

```
# info signs
train_info = ['train/train/{}'.format(i) for i in os.listdir(
    train_dir) if 'info' in i] #all imgs with the word "info" (642
    images)
# danger signs
train_perigo = ['train/train/{}'.format(i) for i in os.listdir(
    train_dir) if 'perigo' in i] #all imgs with the word "perigo"
    (642 images)
# all the test images
test_imgs = ['test/test/{}'.format(i) for i in os.listdir(test_dir)
    ]
```

Listing 3.2: Image storage.

Then, the information and danger images in the training set are stored in a single variable, called `train_imgs`, and both training and test sets randomize the order of the photographs, using the `shuffle` method from the `random` library. In order not to run out of memory when training the model, we also clean up the variables that are no longer needed (`train_info` and `train_perigo`) using the `collect()` method from the `gc` library. To check if the first 4 images were being passed to the program correctly, a small `plot` was given to them using the `mpimg` module, as we can see in the figure 3.10.

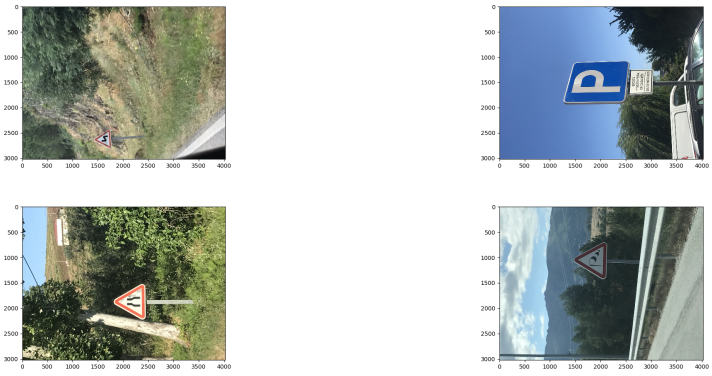

Figure 3.10: Some of the stored images.

3.7 Image and class processing

This section will explain how the images stored in the previous section were processed. For this purpose, the variables that will contain the dimensions to

be applied to all images were initially declared. The variables "`nrows`" and "`ncolumns`", both with a value of 150, determine the height and width of the images, respectively, and the variable `channels` with a value of 3, determines the number of channels in the images. A color image consists of 3 channels corresponding to the red, green and blue pixels (RGB). We could also use 1 channel that would, in this case, read the images in the *gray-scale* format, that is, in black and white.

To read and process the images with the above mentioned dimensions, the function `read_and_process (images)` was created. This function is presented in the excerpt 3.3.

```
def read_and_process(images):
    X = [] #images
    y = [] #labels
    for i in images:
        X.append(cv2.cv2.resize(cv2.cv2.imread(i, cv2.cv2.
            IMREAD_COLOR), (nrows, ncolumns), interpolation = cv2.
            cv2.INTER_CUBIC))
        if 'info' in i:
            y.append(1)
        elif 'perigo' in i:
            y.append(0)
    return X,y
```

Listing 3.3: Reading and processing images.

This function will return two variables: X and y. The variable X will contain the training set, while the variable y will contain the labels. Then, with the `OpenCV` library (cv2), we will read and size all the images with the previously defined values, saving them into the X array. Finally, we fill the vector y with the labels. If the image contains the word "info" in the name, the value "1" will be added to the array, otherwise "0" will be added.

To check if the images are in fact being processed, and after applying the previous function, I did a `print()` to the first position of X and y, having obtained the results presented in the figures 3.11 and 3.12.

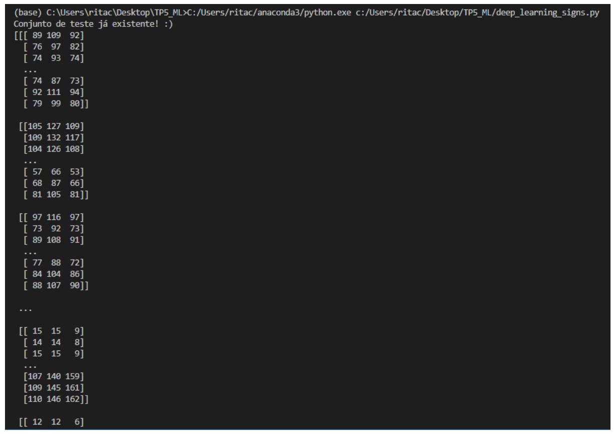

Figure 3.11: Representation of the first image.

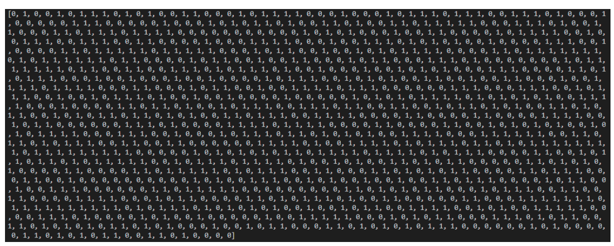

Figure 3.12: Image classes.

In this way, it was possible to understand how the computer reads each image (pixel by pixel), thus reaching the conclusion that in the first image it is a danger sign (because the label in the first position is 0). To check the state of the images after processing them, a `plot` was given to the first 5 figures present in the vector X, as we can see in the figure 3.13. In this case, it is not possible to display these images with the `mpimg` module from the `matplotlib` library as they are now interpreted as pixel arrays and not pure .jpg files. Therefore, it was necessary to use the `imshow()` module.

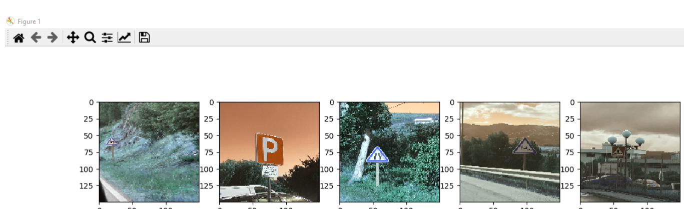

Figure 3.13: Preview of the first 5 images.

Then, a new memory cleaning was performed, and the variables X and y, which were in list format, were changed to a numpy array in order to insert them into the model. After making sure that the images were being correctly processed and stored, it was also prudent to check the labels/classes. If both classes are being well stored in the respective vector (y), there must be 642 zeros and 642 ones, that is, 642 information signs and the same number of danger signs. For this purpose, the labels `plot` was also performed, in a histogram format, using the `seaborn` library.

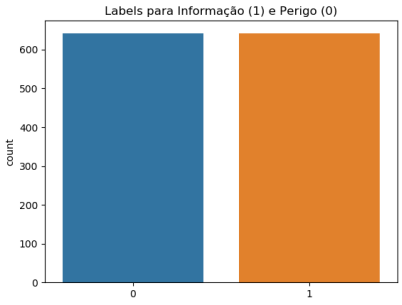

Figure 3.14: Visualization of the number of classes.

After making sure that we actually have two classes with 642 signs each, we can also observe the shape of our data. The obtained results are shown in the figure 3.15.

```
Shape of train images is:  (1284, 150, 150, 3)
Shape of labels is:  (1284,)
```

Figure 3.15: X and y shape.

We can conclude that the training set consists of a 4-dimensional array - 1284x150x150x3 - corresponding to the batch_size (dataset size), height, width and channels of the images, respectively. The array shape of images is important for the model in `Keras` that will be built, as it receives as input a vector with height, width and channels.

3.8 Division of training and validation set

One of the most important things to do before building and training the model, is to divide the remaining dataset into a training and validation set. For this division, the `train_test_split` method of the `sklearn.model_selection` library was used, as we can see in the excerpt 3.4.

```
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size =
    0.20, random_state=2)
```

Listing 3.4: Division into training and validation sets.

In this case, 80% of the dataset was assigned to the training set and 20% to the validation set.

After this division, we checked the shape of the dataset again. The result is shown in the figure 3.16.

```
Shape of train imgs:  (1027, 150, 150, 3)
Shape of validation imgs:  (257, 150, 150, 3)
Shape of train lables  (1027,)
Shape of validation imgs:  (257,)
```

Figure 3.16: X and y shape after splitting the test and validation sets.

After a new memory cleaning of X and y variables, we saved into two variables (`ntrain` and `nval`) the size of the training and validation sets, respectively, for later use. A 32 batch size has also been defined.

3.9 CNN model architecture

In this section we will discuss the architecture of the implemented model, which refers to the way in which the convolutional layers are inserted. A popular, effective and simple architecture called VGGnet was used. This architecture was proposed by the *Visual Geometry Group*, a group at the University of Oxford, in 2014.

This model was created through the `Keras` library. This is a library of open source neural networks written in Python. It is capable of running on *TensorFlow*, *Microsoft Cognitive Toolkit* or *Theano*. `Keras` was designed to allow rapid experimentation with deep neural networks, thus focusing on easy, modular and extensible use.

A small VGGnet was used, but we can see that the size of the filter increases as the layers increase. The architecture used is summarized in: 32 $\rightarrow$ 64 $\rightarrow$ 128 $\rightarrow$ 512 - and the final layer is 1. This is presented in the excerpt 3.5.

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3,3), activation='relu', input_shape
    =(150,150,3)))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(64, (3,3), activation='relu'))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(128, (3,3), activation='relu'))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(128, (3,3), activation='relu'))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
```

```
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Listing 3.5: CNN model architecture.

Initially, a sequential layers model is created. We then create a first layer by calling the function `.add()` in the model created and then passing the type of layer that we want, in this case, a `Conv2D` layer. This first layer is called input layer and has some important parameters, such as:

- **filters**: this is the size of the output dimension, that is, the number of output filters in the convolution. In this case, we start with 32;
- **kernel_size**: specifies the height and width of the 2D convolution window, in this case 3x3;
- **activation**: it refers to the activation function to be used in the neural network. In this case we use Rectified Linear Unit (reLU), which is the most common activation function used today. This function improves the neural networks as it speeds up the training process. The gradient calculation is very simple (0 or 1, depending on the x signal). In addition, the computational steps of the reLU function are simple: any negative element is defined as 0.0 - without exponentials and without multiplication or division operations [5];
- **input_shape**: it refers to the dimensions with which we resize our images and their the channel, in this case is [150,150,3].

Next up is the `MaxPooling2D` layer. The task of this layer, as already discussed in the chapter 2, is to reduce the spatial size of the resources received and, therefore, to reduce the number of parameters and the computation in the model. Thus, this layer helps to reduce over-adaptation (overfitting). The overfitting happens when our model memorizes the training data. In this way, the model would perform excellently in training time, but would fail in test time.

Then, another convolutional layer was added, this time with 64 filters and the respective `MaxPooling2D` was made. The same happened for the remaining layers, both with 128 filters.

After that, the `Flatten layer` was declared. The `Conv2D` layers extract and learn spatial resources, which are passed to a dense layer after being "flattened". Basically, this layer transforms a two-dimensional array of resources into a vector that can be fed into a fully connected neural network classifier, as we can see in the figure 3.17.

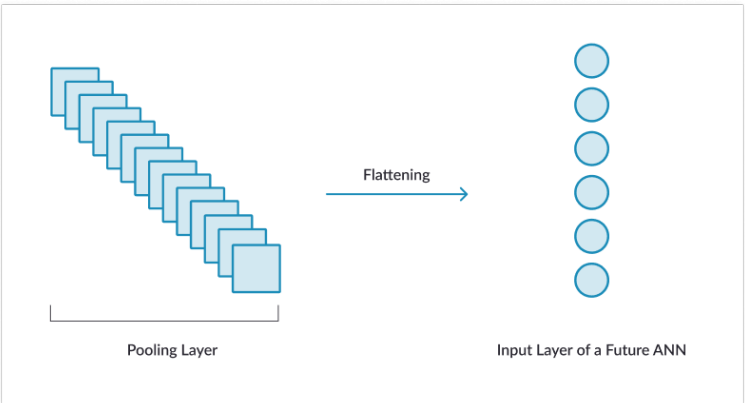

Figure 3.17: Flatten layer.

Next, we add a `Dropout` layer with a value of 0.5. This layer randomly discards some layers in the neural networks and learns from the reduced network. In this way, the network learns to be independent and unreliable in a single layer. This layer also assists in overfitting. The 0.5 value means randomly discarding half the layers.

Then a dense layer was added, that is, a fully connected layer with 512 inputs.

The last layer, also dense, has an output size of 1 and an activation function called `sigmoid`. This is because we are trying to detect whether an image is an information or danger traffic signal, that is, we want the model to show a probability of how certain an image is an information signal and not a danger signal. For this purpose, we intend a probabilistic value, in which higher values mean that the classifier believes that the image is an information signal and lower values a danger image. The `sigmoid` function is ideal for this task, as it receives a set of numbers and returns a probability distribution in a range from 0 to 1.

To visualize the architecture and parameter size of the implemented CNN, we can use the `summary()` method of `Keras` on the model object.

In the figure 3.18, we can see the number of parameters we intend to train, in this case more than 3 million, and the general architecture of the different layers.

Then, it was necessary to compile the model with the `compile()` method of `Keras`, as we can see in the excerpt 3.6.

```
model.compile(loss='binary_crossentropy', optimizer = optimizers.
    RMSprop(lr=1e-4), metrics= ['acc'])
```

Listing 3.6: Model compilation.

In this method, three parameters are passed:

- **loss**: we specify a loss function that our optimizer will minimize. In this

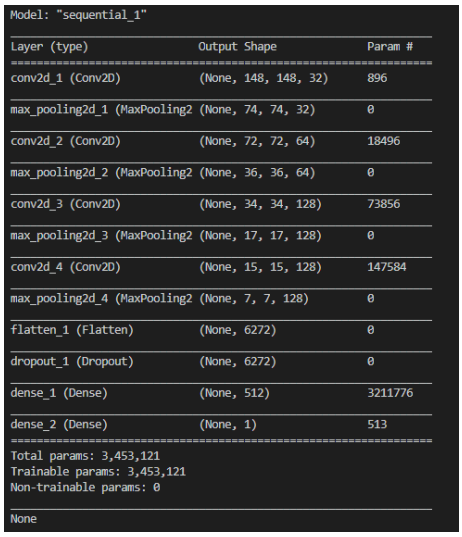

Figure 3.18: Model summary.

case, as we are working with a model with only two classes, we can use the binary cross entropy loss (`binary_crossentropy`);

- **optimizer**: in this case we use the RMSProp optimizer with a learning rate of 0.0001. This optimizer is part of a process called hyperparameters adjustment, which can be the difference between a real model and a more naive one. Hyperparameters are all parameters of a model that are not updated during learning and are used to configure the model (for example, the size of the hash space, number of decision trees and their depth, number of layers of a deep neural network, etc.) or the algorithm used to decrease the cost function (algorithm learning rate, etc.). Thus, hyperparameter adjustment consists of the problem of automatically choosing a set of ideal hyperparameters for a learning algorithm;

- **metrics**: this is where we specify which metric we want to use to measure our model's performance after training. As we are dealing with a classification problem, the precision metric (`acc`) is a good option. This metric is used to measure the performance of the model, which depends on the type of problem we are dealing with [2].

3.10 Normalization

Finally, and before we start training the model, we need to perform some normalization, such as, for example, scaling our pixel values of the images to obtain a unit standard deviation and an average of 0. For this purpose, the

`ImageDataGenerator` module from `Keras` was used, which performs some important functions when inserting images into the model during training.

According to the creator of `Keras`, François Chollet, the `ImageDataGenerator` method of this library allows us to quickly configure generators that automatically transform image files into pre-processed tensors, which can be directly embedded on models during the training phase. This method includes several features, such as:

- Decode the .jpg content into RGB pixel grids;
- Convert these grids to floating point tensors;
- Dimension the pixel values (between 0 and 255) for the interval [0.1] (the neural networks perform better with normalized data);
- Helps to easily enlarge images (data augmentation). This is an important resource that will be used, as our training dataset is small.

For this purpose, two generators were created, one for the training set and another for the validation set, as we can see in the excerpt 3.7.

```
train_datagen = ImageDataGenerator(rescale=1./255, rotation_range
    =40, width_shift_range=0.2, height_shift_range=0.2, shear_range
    =0.2, zoom_range=0.2, horizontal_flip = True, )
val_datagen = ImageDataGenerator(rescale=1./255)
```

Listing 3.7: Images normalization.

As we can see, we initially passed the rescale parameter to the `ImageDataGenerator` object. The option rescale = 1./255 is a very important parameter, as it normalizes the pixel values of the image by inserting them in the range [0,1]. This helps the model to learn and update its general parameters more efficiently. The following parameters are related to data augmentation. These tell the `ImageDataGenerator` to randomly apply some transformation to the images. This will help to increase our data set and improve generalization, also in order to preventing overfitting.

In the `val_datagen` variable we also created an `ImageDataGenerator` object for our validation set, however, in this case, we didn't perform data augmentation, implementing only the resizing operation.

After having the `ImageDataGenerator` objects initialized, we effectively created the generators from them, passing on our training and validation sets, as we can see in the excerpt 3.8.

```
train_generator = train_datagen.flow(X_train, y_train, batch_size=
    batch_size)
val_generator = val_datagen.flow(X_val, y_val, batch_size=
    batch_size)
```

Listing 3.8: Creation of generators.

As you we see, the `flow()` method was called in the previously created data generators, passing the X and y sets: X_train and y_train for training and X_val and y_val for validation. The batch size tells the data generator to take only the specified size (32 in our case) of images, at a time.

3.11 Model training

The training of the model was done by evoking the `fit()` method of `Keras`, and passing some parameters, as we can see in the excerpt 3.9.

```
history = model.fit_generator(train_generator,
                              steps_per_epoch=ntrain // batch_size,
                              epochs=100,
                              validation_data=val_generator,
                              validation_steps=nval // batch_size)
```

Listing 3.9: Model training.

The first parameter refers to the `ImageDataGenerator` object of the training set, which is called `train_generator`;

In the second parameter (`steps_per_epoch`) the number of steps per iteration is specified. This tells the model how many images we want to process before we update the gradient to our cost function. In this case, a total of 1027 images divided by the size of the batch set to 32, will take approximately 10 steps. This means that we will make a total of 10 gradient updates to the model, per iteration.

In the third parameter, the number of epochs (iterations) is defined. An iteration is a complete cycle throughout the training set. In our case, a epoch is achieved when we do 10 gradient updates, as specified by the parameter `steps_per_epoch`. In this case, 100 iterations were defined.

Then, in the last two parameters, we indicate the validation set and the size of the steps, which in this case is the size of the vector `X_val`.

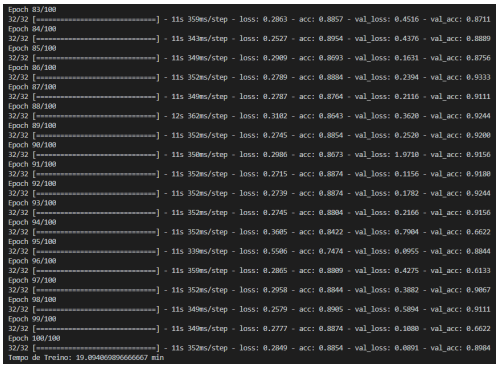

Figure 3.19: Model training at the terminal.

In the figure 3.19, we can see the training process printed in the terminal of the used IDE.

After the training is completed, the model weights are saved using the `save_weights()` method of `Keras` as like as the model itself, using the `save()` method from the same library.

After training a model using `Keras`, the metrics that were specified when compiling the model are always calculated and saved in a variable called `history`.

It is possible to extract these values, showing them graphically through the `matpltlib` library. This `history` object contains all the updates that happened during the training process.

These values were extracted as shown in the excerpt 3.10.

```
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1) #tamanho da epoch a partir do
    numero de valores na lista "acc"
```

Listing 3.10: Extraction of metrics in model training.

3.12 Model testing

Finally, the model was tested with the images present in the test set.

For this purpose, the same pre-processing that was applied to the training and validation sets was done, as we can see in the excerpt 3.11.

```
X_test, y_test = read_and_process(test_imgs[49:64])
x = np.array(X_test)
test_datagen = ImageDataGenerator(rescale=1./255)
```

Listing 3.11: Pre-processing in the test set.

Initially we read and convert the set of images between positions 49 and 64 of the test set into a list of arrays using the previously defined function `read_and_process()`. In this case, the `y_test` will be empty because the test set does not have defined labels. We then convert that list into a numpy array and finally, we initialize an `ImageDataGenerator` object from that test set, applying only the image pixel normalization operation. After this pre-processing, we create a loop, in this case a `for` cycle, which will iterate the generator images and, with these, make the predictions and display the results. We can see this process in the excerpt 3.12.

```
i=0
text_labels = [] #will contain the labels that will be predicted
plt.figure(figsize=(30,20))

for batch in test_datagen.flow(x, batch_size=1):
    pred = model.predict(batch)
    if pred > 0.5:
        text_labels.append('info sign')
    else:
        text_labels.append('danger sign')
    plt.subplot(5 / columns +2, columns, i+1)
    plt.title('Isto e um ' + text_labels[i])
    imgplot = plt.imshow(batch[0])
    i+=1
    if i % 15 == 0:
        break
```

```
plt.show()
```

Listing 3.12: Predictions and display of the test images.

The prediction is made in each image of the test set passed by the object `ImageDataGenerator`, through the `predict()` method of `Keras`. The variable `pred` represents a probability of how certain the model is that the current image is a danger or information traffic signal. As we assign information signals labels a value of 1, that is, a high probability (at least greater than 0.5), this means that the model is quite confident that the image would be, in this case, a information sign, otherwise it is interpreted as a danger sign.

Thus, for this purpose, a `if-else` instruction was created that appends the string "danger sign" if the probability is smaller than 0.5, otherwise, appends "information sign" to the `text_label` variable. This process is done in order to add a title to each image when we display them through the `matplotlib` library.

Chapter 4

Obtained results and experimental tests

After the implementation and training of the model, several results were graphically obtained through the `matplotlib.pyplot` library. Some tests were also carried out, varying the number of epochs, the batch size and the optimizer.

4.1 Obtained results

The project was carried out, initially, with a `batch_size` of 32, with 100 iterations, using the RMSProp algorithm and the set between image 50 to 65.

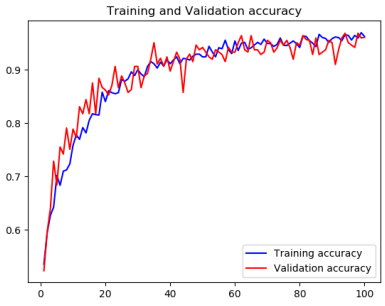

Figure 4.1: Training and validation accuracy.

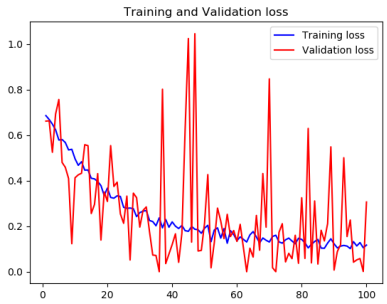

Figure 4.2: Training and validation cost.

The first thing that can be observed from the graphs present in the figures 4.1 and 4.2, is that overfitting is not happening at all, because the training accuracy and validation are relatively close. It is also worth noting that the

precision increases continuously as the iterations increase, suggesting that if we increase the number of iterations, it will probably give an even higher accuracy. Regarding the cost graph, it is also not doing overfitting because this value is decreasing, however there are more fluctuations. The cost is likely to decrease if we increase the number of iterations.

This test was performed for the set between the first image and the fifteenth. The results are present in the figure 4.3.

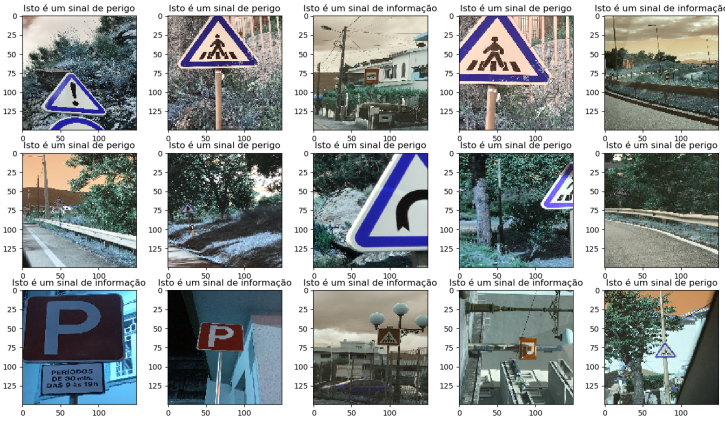

Figure 4.3: Figures 50 to 65 - obtained results.

In this case, and making an empirical evaluation of the obtained images and the corresponding labels, we were able to observe that the model was correct in all 15 figures tested, assigning the correct class (danger (*perigo*) or information (*informação*)) to each one.

4.2 Experimental tests

4.2.1 Fewer number of iterations

In this sub-section we will demonstrate the obtained results, with the same parameters as in the section 4.1, except for the number of iterations that has been reduced to just 20.

In the graphs presented in the figures 4.4 and 4.5, it is possible to verify that there is much less precision, therefore there is not such a linearity in the validation function in relation to the training one. The same applies for the cost graph, which has more fluctuations.

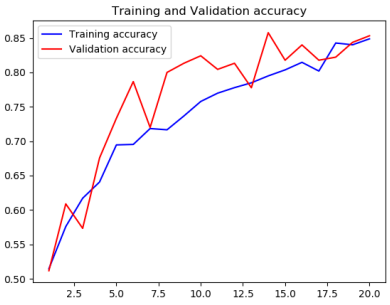

Figure 4.4: Training and validation accuracy - 20 iterations.

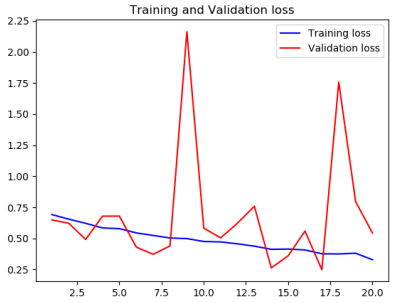

Figure 4.5: Training and validation cost - 20 iterations.

Figure 4.6: Image 50 to 65 - 20 iterations.

In this case, and making an empirical evaluation of the results obtained, we were able to notice that the model failed 2 images out of 15 for the set of images in question, thus concluding that it needs more iterations to learn more accurately.

4.2.2 Highest number of iterations

In this sub-section we will demonstrate the obtained results, with the same parameters as in the section 4.1, with the exception of the number of iterations that has been increased to 200. The set of tested photos has also been changed to between positions 150 and 165, since the model got all the signals right during the 100 iterations tested.

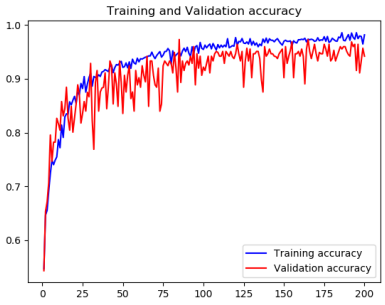

Figure 4.7: Training and validation accuracy - 200 iterations.

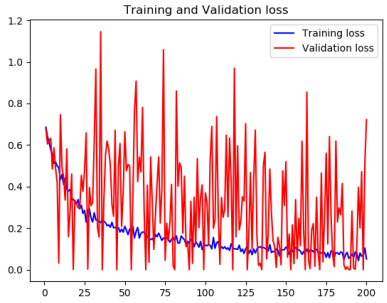

Figure 4.8: Training and validation cost - 200 iterations.

In the graphs presented in the figures 4.7 and 4.8, it is possible to verify that there is a great precision of the validation set compared to the test set, therefore, there is no *overfitting*. In the graph that represents the value of cost per iteration, we can see that the validation set has a lot of fluctuations in relation to the training, however, both started in an identical cost position and ended in the same way.

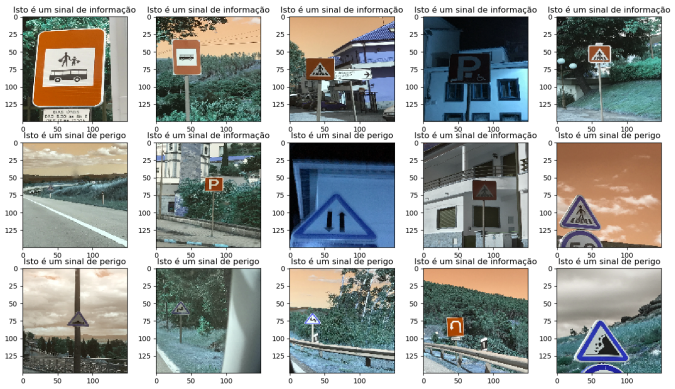

Figure 4.9: Image 150 to 165 - 200 iterations.

According to the results presented in figure 4.9, and also making an empirical evaluation of the results obtained, we can observe that the model was correct in all 15 figures tested, assigning the correct class (danger or information) to each one.

4.2.3 Smallest batch size

In this sub-section we will demonstrate the obtained results, with the same parameters as in the section 4.1, except for the size of the batch which was reduced to 10. The set of photos tested was also changed for between positions 304 and 319.

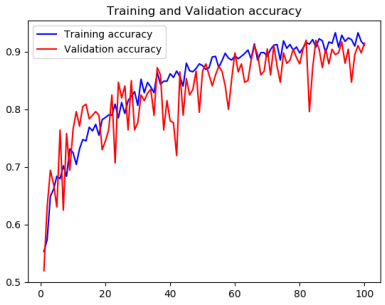

Figure 4.10: Training and validation accuracy - batch size of 10.

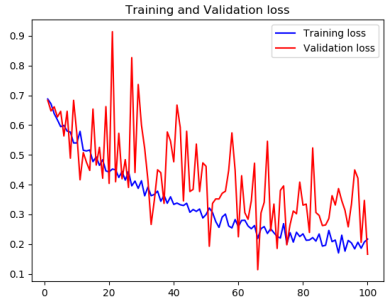

Figure 4.11: Training and validation cost - batch size of 10.

In the graphs presented in the figures 4.10 and 4.11, it is possible to verify that we obtained, in general, better results, in relation to the results initially obtained in the section 4.1. There is no overfitting in any of the graphs, and the validation and training sets have relatively close values.

Figure 4.12: Image 304 to 319 - batch size of 10.

According to the results presented in figure 4.12, and also making an empirical evaluation of the results obtained for the set of images in question, we were able to observe that the model was wrong on 1 image out of 15.

4.2.4 Largest batch size

In this sub-section we will demonstrate the obtained results, with the same parameters as in the 4.1 section, except for the size of the batch which has been increased to 100. In parallel to the previous sub-section, the set of photos tested was also changed to between positions 304 and 319.

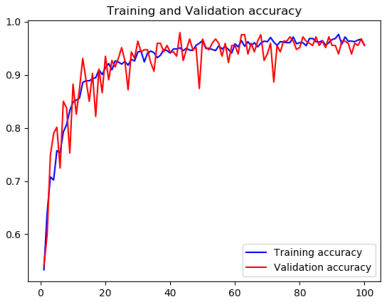

Figure 4.13: Training and validation accuracy - batch size of 100.

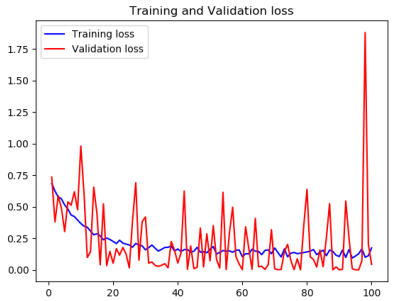

Figure 4.14: Training and validation cost - batch size of 100.

In the graphs presented in the figures 4.13 and 4.14, it is possible to verify that we obtained better results in relation to the graphs referring to a batch size of 10, with in general a great similarity between the values obtained in training and validation sets, so there is no overfitting.

Figure 4.15: Image 304 to 319 - batch size of 100.

According to the results presented in figure 4.15, and also making an empirical evaluation of the results obtained for the set of images in question, we were able to observe that the model failed in 1 image out of 15, having failed the same image that it did with a batch size of 10.

4.2.5 Adam optimizer

In this sub-section we will demonstrate the obtained results, with the same parameters as in the section 4.1, except for the optimizer that was changed to Adam. The set of tested photos has also been changed to between positions 185 and 200.

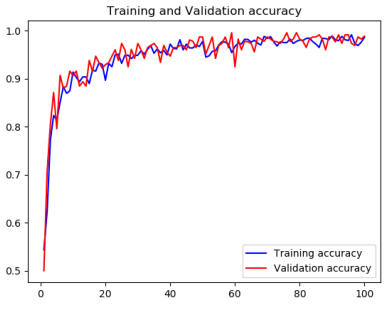

Figure 4.16: Training and validation accuracy - Adam optimizer.

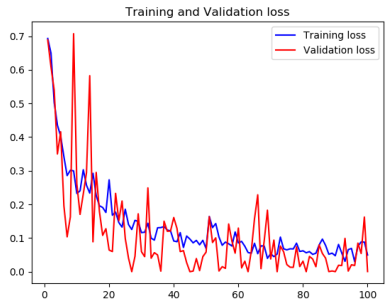

Figure 4.17: Training and validation cost - Adam optimizer.

In the graphs presented in the figures 4.16 and 4.17, it is possible to verify that we obtained better results in relation to the results initially obtained in the section 4.1, with the training and validation sets having very close values, mainly in the graph that shows the accuracy value by iteration. Looking at the graphs above, it is possible to conclude that the Adam optimizer is generally more stable and learns more quickly, compared to the optimizer initially used, the RMSProp.

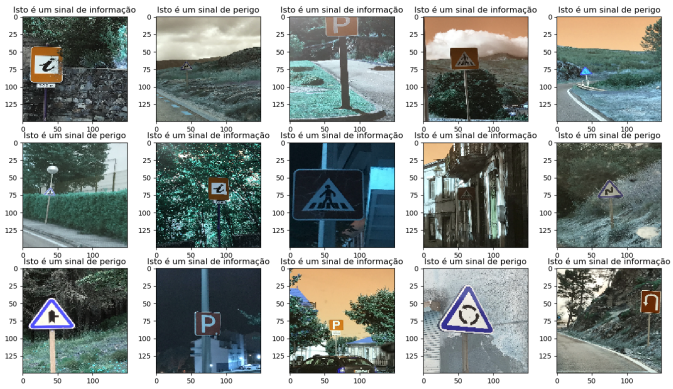

Figure 4.18: Image 185 to 200 - Adam optimizer.

According to the results presented in figure 4.18, and also making an empirical evaluation of the results obtained, we can observe that the model was correct in all 15 figures tested, assigning the correct class (danger or information) to each one.

4.2.6 acSGD optimizer

In this sub-section we will demonstrate the obtained results, with the same parameters as in the section 4.1, except for the optimizer that was changed to the SGD. The set of tested photos has also been changed to between positions 215 and 230.

In the graphs presented in the figures 4.19 and 4.20, we can see that we obtained slightly worse results in the accuracy graph in relation to the results initially obtained in the section 4.1 and in the previous sub-section. The training and validation sets still have relatively close values. Looking at the graphs above, it is possible to conclude that the SGD optimizer is generally less stable and learns more slowly, compared to the optimizer initially used (RMSProp) and the optimizer tested in the previous sub-section (Adam). However, and still compared to the RMSProp optimizer, the graph of the cost functions with SGD shows less oscillations between both training and validation sets, with a much

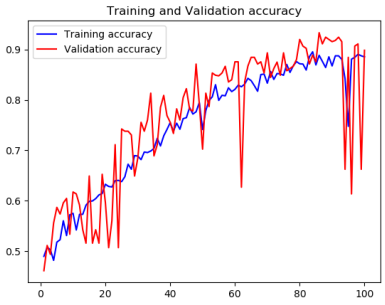

Figure 4.19: Training and validation accuracy - SGD optimizer.

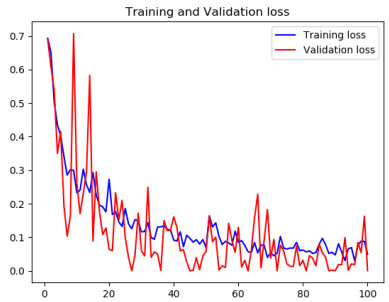

Figure 4.20: Training cost and validation - SGD optimizer.

more pronounced drop in cost. None of the graphics are doing overfitting.

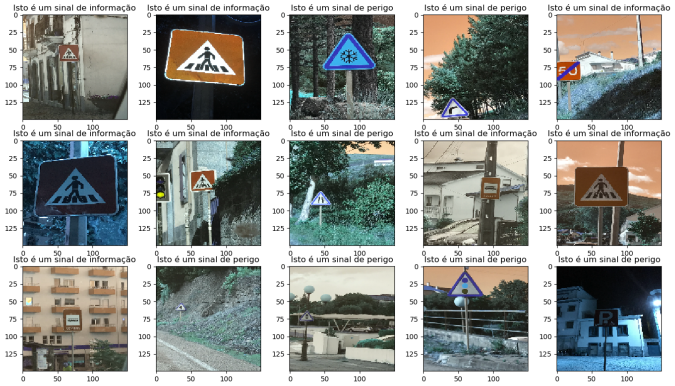

Figure 4.21: Image 215 to 230 - SGD optimizer.

According to the results presented in figure 4.21, and also making an empirical evaluation of the results obtained for the set of images in question, we were able to observe that the model was wrong in 1 image out of 15.

Chapter 5

Conclusion and future work

AI has witnessed a giant growth between the capabilities of Humans and machines. One of the main objectives in the field of Computer Vision, is to allow machines to perceive the World in a manner as similar as possible to Human Beings, using this knowledge for various tasks, such as systems of recognition, analysis and classification of images and videos, processing natural language, etc. The advances in this domain, with the help of Deep Learning, were built and improved over time, mainly through a specific algorithm - a CNN.

This report initially gave an introduction about the CNN operation mode. We are thus able to understand that a CNN receives a certain input (usually an image), passes it through a series of filters and layers, thus seeking to obtain a output that can be classified.

This work was very important to learn and understand in much more detail how a CNN works in its several optimization processes, from the importance of pre-processing the images, to the final model that can be classified.

It is important to underline the positive impact that the use of the GPU, compared to the CPU, had on the model's training time. If the used GPU was more powerful, it was possible to train the model in a matter of minutes, as well as add a larger dataset without losing performance at the runtime level.

The obtained results were also shown and several tests were performed that varied parameters such as the number of iterations, batch size and the optimizer used. Out of these, we came to the conclusion that the best optimizer, for the problem in question, is Adam for presenting a higher learning speed, for the same learning rate value, in relation to the others. We also came to the conclusion that the ideal is to have at least 100 iterations, so that the model is as accurate as possible. As for the batch size, we obtained similar results in the performed tested cases, although the results for a batch size of 100 being slightly better than the rest.

Therefore, I conclude that all the objectives of the present work have been successfully completed and, as a future work, I would very much like to extend the model to more classes of traffic signs, possibly even covering all the existing ones.

Bibliography

[1] Alex Fernandes Mansano. O que é uma Rede Neural Convolucional?, 2017. [Online] https://pt.linkedin.com/pulse/o-que-%C3%A9-um-rede-neural-convolucional-alex-fernandes-mansano. Último acesso a 2 de Julho de 2020.

[2] Alois Bissuel. Hyper-parameter optimization algorithms: a short review, 2019. [Online] https://medium.com/criteo-labs/hyper-parameter-optimization-algorithms-2fe447525903. Último acesso a 3 de Julho de 2020.

[3] Simon Haykin. *Neural Networks and Learning Machines*. ISBN: 978-0-13-147139-9. Pearson Education, Inc., 3 edition, 2009.

[4] NVIDIA. CUDA Toolkit. [Online] https://developer.nvidia.com/cuda-toolkit. Último acesso a 2 de Julho de 2020.

[5] Sycorax. Why do we use ReLU in neural networks and how do we use it?, 2018. [Online] https://stats.stackexchange.com/questions/226923/why-do-we-use-relu-in-neural-networks-and-how-do-we-use-it. Último acesso a 3 de Julho de 2020.