Universidade do Minho
Escola de Engenharia

Sérgio Cristiano Neiva Alves Baixo

3D Facial Recognition using Deep Learning

Dissertação de Mestrado

Engenharia Eletrónica Industrial e Computadores

Automação, Controlo e Robótica

Trabalho efetuado sob a orientação do
Professor Doutor António Fernando Macedo Ribeiro

Novembro de 2021

Acknowledgements

The current dissertation represents the end of my master's degree in Industrial Electronics and Computers Engineering. An extremely demanding challenge whose finale only happened thanks mainly to a lot of hard work, dedication, discipline, consistency, perseverance, and the constant belief and optimistic way of thinking that I am always capable of achieving everything I commit myself to. Having a clear vision of what I wanted to achieve, always helped me to stand up and retry, no matter how many times I failed. Despite this, I would not have done it without the support and incentive of several people along the way.

The most special thanks go to the three most important people in my life, my parents José and Fernanda, and my brother José Pedro for all the love, support and appreciation given, even during those moments when I had to be away from them to dedicate myself to my personal and professional goals and ambitions.

Big thanks to my supervisor, Professor Fernando Ribeiro for the opportunity to work on such a joyful and challenging environment, the Laboratory of Automation and Robotics. I could not be more grateful for all the support and valuable guidance he gave to me, including helpful advices to improve my dissertation. Also thanks to Tiago Ribeiro from the Laboratory for always pointing out the way to go whenever I was having difficulties with the dissertation's work, and for always suggesting new ideas that could eventually contribute to achieve this dissertation's final solution. Big thanks to all the members of the Laboratory in general, specially to Rafael Marques and Bruno Sousa for always putting the moral up during those long car journeys and long hours working in the Laboratory of Automation and Robotics.

Also many thanks to all my hometown friends for always making me "think outside the box" and continuously broaden my standpoint on numerous topics.

Lastly, I want to thank my classmates for all joyful moments we have created together during our time in college. Those moments really helped to withstand the long hours of study and assignment development.

Statement of Integrity

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration. I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

Reconhecimento Facial 3D através de Deep Learning

Com a ascensão de *Deep Learning*, tecnologias de reconhecimento facial baseadas em redes neuronais de convolução, tornaram-se o método de eleição para o desenvolvimento de soluções baseadas em reconhecimento facial. Contudo, apesar do seu grande potencial, o reconhecimento facial apresenta algumas preocupações relativamente à privacidade e segurança quando aplicado. Além disto, os primeiros algoritmos de reconhecimento, a duas dimensões (2D), embora apresentem um desempenho elevado, são influenciados por diversos fatores tais como condições de luz ambiente, a orientação e expressões faciais do sujeito, comprometendo assim a precisão do modelo e, consequentemente, a sua eficácia. Esta tecnologia tem vindo a tornar-se cada vez mais popular tendo sido, inclusive, em diversos aeroportos e, em larga escala, na indústria dos *smartphones*, permitindo assim uma interação mais natural e espontânea entre humano e máquina.

Ao longo desta dissertação foi desenvolvido um sistema de visão por computador, que utilizando técnicas de Inteligência Artificial, nomeadamente *Deep Learning,* é capaz de reconhecer as faces de pessoas a 3 dimensões (3D) em tempo real. Para tal, recorreu-se a imagens RGB e mapas de profundidade capturados em diversas pessoas, implementado assim medidas extra de segurança que permitiram colmatar as falhas adjacentes aos métodos 2D mencionados acima.

Procedeu-se ao desenvolvimento de um algoritmo de captura de imagem de pessoas com as quais se pretendeu realizar o reconhecimento facial. Para tal, foi necessário recorrer a *hardware* específico, nomeadamente uma câmara 3D. Segue-se o processamento individual de cada imagem, e por fim foram desenvolvidas duas redes neuronais de convolução treinadas de forma independente, sendo uma delas especificamente desenvolvida para reconhecimento de imagens RGB enquanto a restante foi desenvolvida com ênfase no reconhecimento de mapas de profundidade. Após a conclusão do treino das redes neuronais, ambas se encontram aptas para reconhecer em tempo real as pessoas onde foi aplicada a captura. O objetivo final passa por aplicar o algoritmo resultante no robô antropomórfico do Laboratório de Automação e Robótica, CHARMIE, com o objetivo de este executar um conjunto de tarefas distintas que variam conforme o utilizador reconhecido pelo sistema de reconhecimento desenvolvido.

Palavras-Chave: 3D, *Deep Learning*, Reconhecimento facial, Visão por computadores

Abstract

3D Facial Recognition using Deep Learning

With the continuous development of Deep Learning, facial recognition technologies based on CNNs (Convolutional Neural Networks) became the main method adopted in the field of facial recognition. However, despite its promising potential, it still presents many concerns regarding privacy and safety when applied. Moreover, the first facial recognition algorithms, the 2D ones, despite having good performance, revealed to suffer from several factors like the environment's lighting conditions, pose and facial expression of the subject in which the recognition was applied, hence compromising the model's accuracy and, consequently, its effectiveness.

This technology became increasingly popular, leading to its use in several environments like in airports and even at a large-scale in the smartphone consumer industry, thus providing a more natural and spontaneous interaction between human and machine.

Throughout this dissertation a computer vision system was developed using Artificial Intelligence techniques, mainly Deep Learning that is able to recognize faces in 3 dimensions (3D) and in real-time. To do so, both RGB and depth images were captured in several subjects, thus implementing extra security measures, that consequently filled in the gaps inherent to 2D methods mentioned above.

An image capture algorithm was developed, which required to use specific hardware, namely a camera that incorporates a 3D sensor. Following this is the image processing of each captured image, and, after this, two CNNs were developed and trained independently, being one of them specifically developed for RGB images, while the other one was developed with emphasis on depth map recognition. After concluding the training process, both networks become suitable to recognize new images from the subjects with which the capture process was taken with, in a real-time scenario. The final goal is to apply the resulting trained models into Laboratory of Automation and Robotics' anthropomorphic robot, CHARMIE, with the purpose of making it to perform different tasks based on the person it has been able to perform the correct recognition.

Keywords: 3D, Computer Vision, Deep Learning, Facial Recognition

Contents

List of Abbreviations

2D – Two-Dimensional

3D – Three-Dimensional

AI – Artificial Intelligence

ANN – Artificial Neural Network

API – Application Programming Interface

CNN – Convolutional Neural Network

CHARMIE – Collaborative Home Assistant Robot by Minho Industrial Electronics

CPU – Central Processing Unit

DL – Deep Learning

DNN – Deep Neural Network

FPS – Frames per Second

GPU – Graphics Processing Unit

GUI – Graphical User Interface

IDE – Integrated Development Environment

LAR – Laboratory of Automation and Robotics

ML – Machine Learning

OS – Operating System

RGB – Red Green Blue

ROI – Region of Interest

List of Figures

List of Tables

1. Introduction

In this introductory section, the contextualization, motivation for choosing the current theme as well as the objectives of this dissertation are going to be presented. The contextualization subsection aims to provide the reader with a contemporary perspective on various subjects that are relevant throughout the rest of this document. Next, the motivations that led to the elaboration of this work are presented, as well as some interesting and notable facts regarding the various topics that helped to sustain the choice for this dissertation's theme.

To achieve valid and plausible results, a need to define objectives arises, so that the research and development follow a systematic set of guidelines to be able to accomplish the goals proposed. It also offers the reader a brief list of the steps that were carried out throughout the present dissertation. Lastly, Document Structure aims to clarify in advance the whole arrangement of the topics of the dissertation

1.1 Contextualization

Vision is perhaps the most important sense that a human being has. The information obtained from the eyes allows humans to have an influence in real-world decision making as well as accomplishing or not, certain tasks, being them real-time ones or not. It is so important and complex, that the human brain cortex has a section exclusively dedicated to visual processing, called the visual cortex. It is responsible for receiving, integrating, and processing the vast amount of information captured by the retina.

Having said this, it would be interesting to replicate the capabilities (or parts of it) of the visual cortex to machines or computers to make them process variable amounts of data. That is where computer vision (and robotic vision) comes in, by focusing on the way to replicate parts of the complexity of the human visual system and enabling computers to identify and process objects in images and videos in the same way humans do.

According to D. Kragic and M. Vincze in [1], computer vision and robot vision are distinct. The first one targets the understanding of a scene mostly from single images or from a fixed camera position. Its methods are tailored for specific applications and research is focused on individual problems and algorithms. On the other hand, the latter requires looking at the system level perspective, where vision is one of several sensory components that work together to fulfil specific tasks, like for example in anthropomorphic robots.

Powered by modern sensor technology, broad access to computing power, and the development of user-friendly programming frameworks, the field of computer vision is experiencing a revitalization that results in ground-breaking modern applications such as autonomous vehicles present in real-life traffic scenarios, medical diagnosis systems that support medical staff in detecting hard-to-find diseases and industrial monitored systems capable of detecting irregularities in manufactured products [2].

Thanks to advances in AI, and innovations in DL and neural networks, both fields have been able to take a great leap forward in recent years and have also been able to surpass humans in some tasks related to the detection and classification of targets/objects with greater accuracy and execution time [3].

Before deep learning, there were some problems associated with computer vision. The tasks that computer vision could perform were very limited and required a lot of manual coding and effort by developers and specialized human operators. For instance, in order to perform facial recognition it was necessary to create a database of images of the subjects to track, then annotate several key data points for every single image, and then capture new images. After all this manual work, the application would finally be able to compare the measurements in the new image with the ones stored in its database and tell if it corresponded with any of the profiles it was tracking. There was very little automation involved and most of the work was done manually. Besides, the error margin was large [3].

With Machine Learning (ML), a whole new different approach to solving computer vision problems was introduced. Instead of relying on hard-coded knowledge to solve problems, machine learning systems allow machines to have the ability to acquire their knowledge, by extracting features from raw data [4]. They then use statistical learning algorithms such as linear regressions or decision trees to detect patterns, classifying images, and detecting objects and subjects in them.

Deep learning provides a fundamentally different approach in doing Machine Learning since it relies on neural networks which in turn enables machines to make accurate decisions without the help of humans. When a neural network is provided with many labelled examples of images, for example, it is able to extract common patterns between those images and transform those into mathematical equations that will help to classify future inputs. When it comes to training the models to get a meaningful accuracy, usually tens of thousands of images are needed, and the more the better.

Deep Learning is a very effective method in dealing with computer vision since it solves the problems mentioned above. In most cases, creating an effective deep learning algorithm comes down to gathering a large amount of labelled training data and fine-tuning the parameters. Deep Learning, along with computer vision, has evolved into practical applications such as cancer detection, self-driving vehicles, and facial recognition too, thanks not only to the unceasing research and contributions of the scientific community but also thanks to new advancements hardware-wise and cloud computing resources.

1.2 Motivation

The global computer vision market size was valued at almost €8.7 billion in 2019 and is expected to grow at a compound annual growth rate of 7.6% from 2020 to 2027 [5]. By 2022, the computer vision and hardware market are estimated to reach €40.2 billion [6]. Driven by the increasing interest in artificial intelligence, but also the rise and continuous research on Deep Learning techniques, computer vision systems have been able to replicate the human vision system, enabling computers and other machines to identify and process regions of interest in images and real-time scenarios in the same way humans do. Disease diagnosis in healthcare, biometric analysis in security, quality inspection in manufacturing, and obstacle avoidance in autonomous driving are some of the most noteworthy applications of computer vision across different industries.

Regarding the field of biometric analysis, face recognition is no more than a natural biometric technique embedded in the everyday lives of all human beings. Amongst all biometric technologies that have been used so far, facial recognition is one of the most widely outspread biometric technologies [7]. 3D face recognition is a very well-known technology that is broadly used by many people since nearly half of the world's population uses a smartphone equipped with some sort of biometric recognition capability. And these values are continuously increasing [8].

Although the accuracy of previous 2D facial recognition algorithms was high, according to J. Luo, *et al.* [9], some factors such as the influence of lighting around the subject, posture, orientation, and the subject's expression compromised the output accuracy, thus, the model's overall effectiveness. To fill in these gaps, 3D face recognition introduces depth maps, a grayscale image that contains information about the distance between the surfaces of objects from a given viewpoint. J. Luo, *et al*, also states that depth information is effective in reducing the impact of illumination, attitude, and expression on face recognition.

This technology is very common in some applications like airport security and has been widely embraced by law enforcement agencies, due to the improving accuracy and availability of the systems and the growing size of face databases. However, they are not yet applied on a complete large scale, since concerns with privacy rights and security are still well-known to the general population.

1.3 Objectives

The goal of this dissertation is to develop an intelligent robot vision system capable of recognizing people in real-time, through the usage of a 3D face recognition algorithm based on two convolutional neural networks (CNNs) and trained using an own made dataset comprised of RGB and depth images, that are taken from a 3D camera. The algorithm must be able to adapt to different scenarios and circumstances like the influence of the environment's lighting conditions and the subject's posture and orientation. To achieve this goal, the problem was structured into the following objectives:

1. Study several CNN architectures, as well as the state-of-the-art CNN face detection and face recognition algorithms to evaluate the most suitable architecture to the final solution.
2. Development of the model's framework, divided into the following sub-tasks:
    - 2.1. Development of an algorithm to manage facial identification and capture of both RGB and depth components of the subjects' faces.
    - 2.2. Development of image processing algorithms.
    - 2.3. Generate two different datasets containing the faces of some students from the Laboratory of Automation and Robotics, one composed of RGB images and the other one by depth images.
    - 2.4. Develop and train two distinct CNN models, one for each component of the subject's face, using the previously generated datasets.
    - 2.5. Fine-tune both networks' parameters to achieve maximum accuracy possible.
3. The final objective is the testing phase, which will be divided into the following sub-tasks:
    - 3.1. Test and improve the model's accuracy, fine-tuning the network's parameters to achieve maximum accuracy possible.
    - 3.2. Test the results with new and unseen dataset images to properly evaluate the model.
    - 3.3. Test the results obtained and apply them to the Laboratory of Automation and Robotics' robot, CHARMIE (Collaborative Home Assistant Robot by Minho Industrial Electronics) so it can perform some of the tasks in the rulebook of the RoboCup@Home [10] competition, which in turn is an international service robots competition.

1.4 Document Structure

After this introductory chapter, a brief research on the literature review is presented. The theoretical aspects essential to this dissertation are detailed, starting with an overview on machine learning as well as neural networks with focus on convolutional neural networks and why this type of neural networks is the most suitable for carrying out the work of this dissertation. Still in the literature review section, some of the convolutional neural network's architectures are described according to their chronological order of appearance. Following this architectures description, there is a brief analysis of the current state-of-the-art methods to deal with face recognition as well as face detection, describing the techniques and architectures commonly used.

Subsequently to the state-of-the-art methods, there is a detailed description of all the methodologies employed to accomplish this dissertation's purpose, from image capture and respective image processing techniques to the development and training of several Neural Networks.

The Tests and Results chapter aims to illustrate the results from the previous chapter's implemented methodologies, mainly the performance of the developed models in real-world scenarios as well as side-by-side comparisons between the distinct techniques employed.

Finally, Conclusions and Further Work aims to summarize all the work done and recap the conclusions taken along the dissertation. Besides, future work ideas are proposed with the intent of making the system even more robust and reliable.

2. Literature Review

Throughout this chapter, an in-depth overview of the various subjects necessary to successfully develop this dissertation is going to be presented. Ranging from the various subfields of Artificial Intelligence all the way to the complex behaviour of deep neural networks, even including the most commonly used CNN architectures, which represents the core development of this dissertation. First, the core concepts of Machine Learning are considered.

2.1 Machine Learning

Machine Learning is a branch of Artificial Intelligence focused on building applications that learn from data and improve their accuracy as they learn over time without the need to program every single action required to do so. In data science, an algorithm is a sequence of statistical processing steps. In machine learning, algorithms are 'trained' to find patterns and features inside massive amounts of data to be able to make decisions and predictions based on new and never seen data. The better the algorithm, the more accurate the decisions and predictions will become as it processes more data. For example, a spam filter is a Machine Learning program that can learn to flag spam given examples of spam emails and examples of regular no spam emails [11].

In [11], A. Géron states that there are many types of Machine Learning systems making it useful to classify them in broad categories based on the following:

- Whether or not they are trained with human supervision (supervised, unsupervised, semi-supervised, and reinforcement learning).
- Whether or not they can learn incrementally on the fly (online versus batch learning).
- Whether the system works by simply comparing new data points to known data points, or instead detect patterns in the training data and build a predictive model (instance-based versus model-based learning).

These criteria are not exclusive, and the categories surely are combined. For example, the spam filter mentioned above may learn on the fly using a DNN model trained using examples of spam. This makes it an online, model-based, supervised learning system. Machine Learning (AI's branch) can be in turn divided into three sub-branches (Figure 1): Supervised Learning, Unsupervised Learning, and Reinforcement Learning, which in turn can also be divided into specific sub-branches.

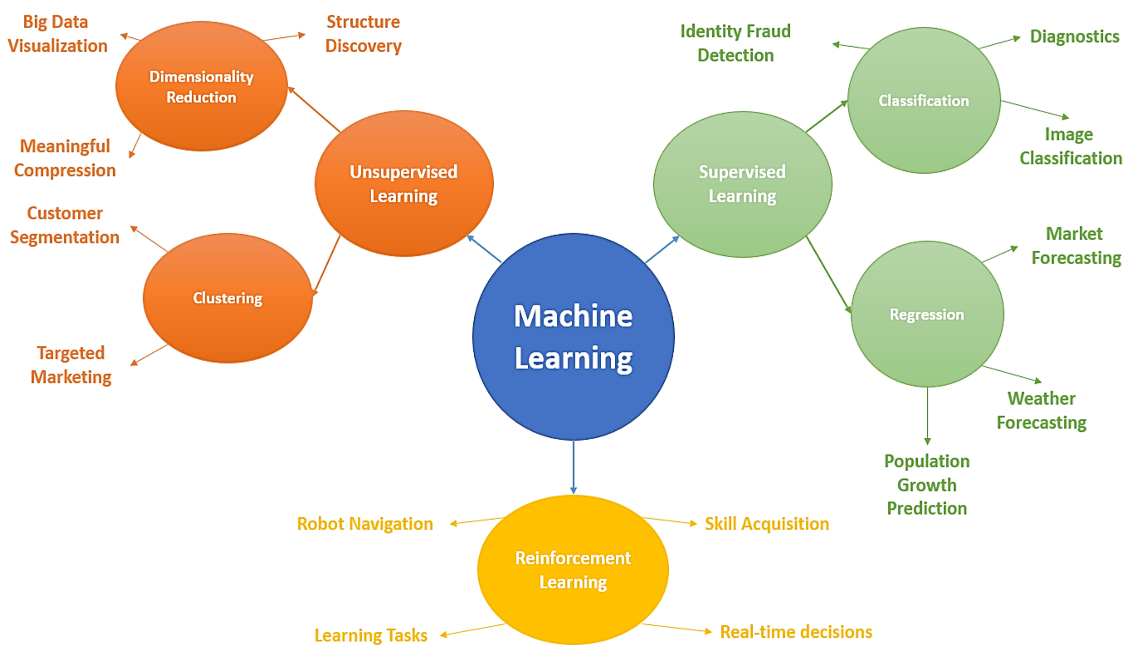


Figure 1- Machine Learning branches and respective sub-branches.

2.1.1 Unsupervised Learning

This sub-branch is where one only has input data (x) and no corresponding output variables. The goal is to model the underlying structure or distribution in the data to learn more about the data. It is called Unsupervised Learning because there are no correct answers and there is no 'teacher'. Algorithms are left on their own to discover and present the interesting structure in the data [12].

Unsupervised Learning scenarios can be further grouped into the following types:

- **Clustering.** Clustering scenarios are those who need to discover the inherent groupings in the data, such as grouping customers by purchasing behaviour. Some of the best-known algorithms for clustering are HCA, which stands for Hierarchical Cluster Analysis, K-means clustering, K-NN (K-nearest neighbours), and PCA (Principal Clustering Analysis).

- **Association.** An association rule learning scenario is where one wants to discover rules that describe large portions of your data, such as people that buy x also tend to buy y.

2.1.2 Supervised Learning

Supervised Learning is defined by its use of labelled data to train algorithms to classify and predict outcomes more accurately, allowing the model to learn over time. The algorithms measure the accuracy through the loss function, adjusting itself until the error has been sufficiently minimized. According to (1), as input data x is fed into the model, it adjusts its weights through a Reinforcement Learning process in order to predict the output variable y, thus ensuring that the model achieves an acceptable level of performance minimizing its loss function ensuring that the model has been fitted appropriately.

$$y = f(x) \tag{1}$$

It is called Supervised Learning because the process of an algorithm learning from the training dataset can be thought of as a teacher supervising the learning process. Assuming that the correct answers are known, the algorithm iteratively makes predictions on the training data and is corrected by the 'teacher', stopping the process when acceptable levels of performance are achieved. Supervised Learning can be further grouped into the following types [13]:

- **Classification** type uses an algorithm to accurately assign test data into specific categories. It recognizes specific entities within the dataset and attempts to draw some conclusions on how those entities should be labelled or defined. The spam filter mentioned in section 2.1 is a good example of a classification problem. It is trained with many examples of emails along with their class (spam or not spam) and it must learn how to classify new incoming emails.

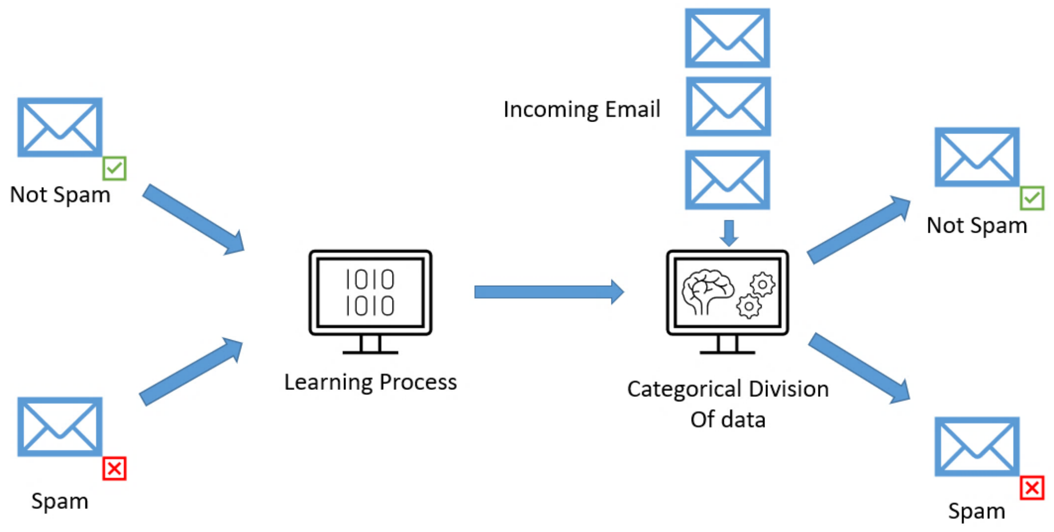


Figure 2- Classification task (spam email filter).

- **Regression** is used to understand the relationship between dependent and independent variables. It is commonly used to make projections, such as for sales revenue for a given business. Linear regression, logistical regression, and polynomial regression are popular regression algorithms. A typical example of a regression problem is to predict the price of a certain car, based on a given set of features like its mileage, age, brand, among others. To train this type of system, namely this example, the model needs to be given many examples of cars, including both their features and their respective labels (i.e., their prices).

2.2 Artificial Neural Networks

As the name suggests, ANNs are modelled based on biological neural networks in the brain, which are made up of cells called neurons, which in turn send signals to each other through connections known as synapses. Neurons transmit electrical signals to other neurons based on the signals they receive from others. An artificial neuron simulates how a biological neuron behaves by adding together the values of the inputs it receives. If this is above a certain threshold, it sends its own signal to its output, which is then received by other neurons. However, a neuron does not always treat each of its inputs with the same weight. Each of its inputs can be adjusted by multiplying it by a weighting factor. For example, if input A is twice as important as input B, then input A would have an input of 2. Weights can also be negative if the value of that input is inversely important [14]. The simplest form of neural networks, often called shallow neural networks are made up of three main layers, being them the input layer, hidden layer, and output layer, as illustrated in Figure 3.

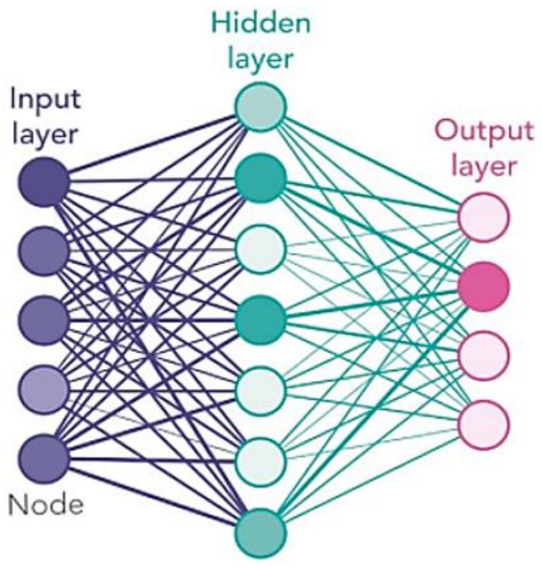


Figure 3- Shallow neural network. Figure from [15] *displayed with the permission of Yang Xiaozhou.*

Each neuron is thus connected to other neurons in the network through the already mentioned synaptic connections, whose values are weighted, and the signals that propagate through the network are strengthened or dampened by these weight values. The process of training involves adjusting these weight values so that the final output of the network gives you the right answer. The first layer is called the input layer. It inputs values, like for example, the pixels of an image. The outputs of this first layer of neurons are connected to a middle layer, called the "hidden" layer. The outputs of the hidden layer are then connected to the final output layer, which in turn gives the final answer to what the network has been trained to do.

For example, a network can be trained to recognize photos of humans and non-humans. Given a dataset of photos labelled with either "human" or "non-human", the network is trained by adjusting its weights so that when it sees a new unlabelled human photo, it outputs if it is either a human or non-human. Neural networks can also classify input data into more than two categories as well, for example, handwritten numerical characters (0-9) like the MNIST dataset. Neural networks offer the following useful properties and capabilities [16]:

- **Nonlinearity** - An artificial neuron can be linear or nonlinear. A neural network, made up of an interconnection of nonlinear neurons, is itself nonlinear. Moreover, the nonlinearity is of a special kind in the sense that it is distributed throughout the network. Nonlinearity is a highly important property, particularly if the underlying physical mechanism responsible for the generation of the input signal (e.g., speech signal) is inherently nonlinear.
- **Input-Output Mapping** - Supervised Learning involves modification of the synaptic weights of a neural network to minimize the difference between the desired response and the actual response of the network produced by the input signal by applying a set of labelled training examples. Each example consists of a unique input signal and a corresponding desired response. Thus, the network learns from the examples by constructing an input-output mapping for the problem at hand.
- **Adaptivity** - Neural networks have a built-in capability to adapt their synaptic weights to changes in the surrounding environment.
- **Evidential Response** - In the context of pattern classification, a neural network can be designed to provide information not only about which pattern to select but also about the confidence in the decision made. This latter information may be used to reject ambiguous patterns, and thereby improve the classification performance of the network.

2.2.1 Deep Neural Networks

Deep Neural Networks (DNNs) are distinguished from the simpler shallow neural networks by their depth, i.e., the number of hidden layers through which data must pass in a multistep process of pattern recognition. Usually, to qualify a neural network as deep, it must consist of at least two hidden layers. Each layer of neurons (or nodes) trains on a distinct set of features based on the previous layer's output. The further the data advances into the network, the more complex the features the neurons can recognize since they aggregate and recombine features from previous layers. This is known as feature hierarchy, and it is a hierarchy of increasing complexity and abstraction. It makes deep neural networks capable of handling very large, high-dimensional datasets with millions of parameters that pass through nonlinear functions [17]. Figure 4 portrays a Deep Neural Network.

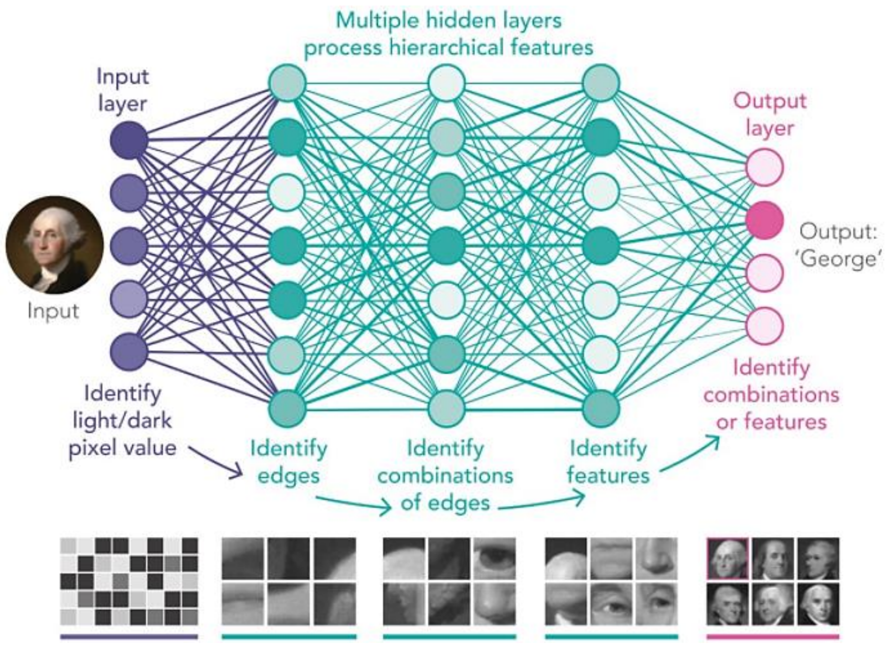


Figure 4- Deep Neural Network. Figure from [15] *displayed with the permission of Yang Xiaozhou.*

Two important operations that make a neural network [15] are the following:

- **Forward Propagation -** this is the prediction step. The network reads the input data, computes its values across the network, and gives a final output value. As mentioned earlier in a shallow neural network, to make a prediction, a vector of numbers is taken as input. Each node in the layer has its own weight. When the input value is passed through the layer, the network computes its weighted sum as shown in Figure 5. This is usually followed by an activation function, as shown in (2):

$$y = f(w.x + b) \tag{2}$$

Where $w.x + b$ is the weighted sum, f is the activation function and y is the output value.

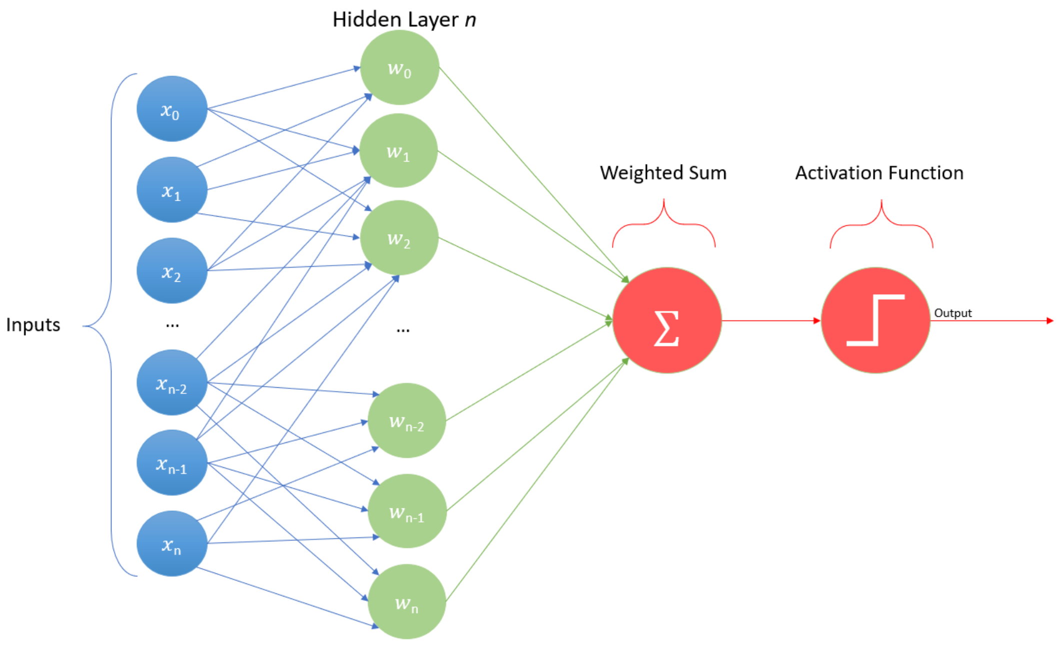


Figure 5- Process of forward propagation.

In a deeper neural network, the process is exactly the same, but the process is repeated for each of the layers.

- **Backpropagation -** this is the essence of training. By comparing the network's predictions/outputs and the ground truth values, i.e., compute loss, the network adjusts its parameters and fine-tunes its weights based on previous epochs (i.e., iterations), improving the performance for the next iterations. Here, the network takes the loss and recursively calculates the loss function's slope with respect to each parameter. An optimization algorithm is then used to update the network's parameters using the gradient information until the performance cannot be improved anymore.

One analogy often used to explain gradient optimization is hiking. Training the network to minimize loss is like getting down to the lowest point on the ground from a mountain. Finding the loss function gradients is like finding the path on the way down. The optimization algorithm is the step where one takes the path and eventually reaches the lowest point.

Although according to Y. LeCun, *et. al.* [18] deep neural networks like the one in Figure 4 are capable of recognizing raw images (usually size-normalized and centred), they have some flaws. For computers, an image is a multiple-dimension array of numbers, and each number represents a pixel value. If it is intended to train a DNN to recognize some object or subject on an image, then each input node should be a pixel. So, for example, if a certain dataset has a set of 30x30 images, then there should be 900 input nodes. If it is a three-channel image, the number of input nodes multiplies by 3. If the input is a high-resolution image, the number of input nodes grows substantially. Hence, the number of parameters in a DNN grows, increasing the computational resources needed to implement and train the network. Nonetheless, the number of parameters may not stand a problem on some modern computers.

According to Y. LeCun, *et al.* [18], the topology represented in Figure 4 has no built-in invariance that uncovers the variations on the input images, namely, translations, scales, and geometric distortions. Moreover, it completely ignores the strong 2D local correlation that images have, i.e., the nearby pixels are highly correlated.

2.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) emerged from the study of the brain's visual cortex, and they have been used in image recognition since the 1980s. In the last few years, thanks to the increase in computational power and the amount of available training data, CNNs have achieved great performance levels on some complex visual tasks. They empower image search services, self-driving vehicles, face recognition systems, among others. Moreover, CNNs are not restricted only to visual perception, they are also successful at other tasks, such as voice recognition or Natural Language Processing [11].

This type of network is specialized in processing data that has a known, grid-like topology. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, and image data, which can be thought of as a 2D grid of pixels. The name "Convolutional Neural Network" indicates that the network employs a mathematical operation called convolution [4].

CNNs combine three architectural ideas to ensure some degree of shift, scale, and distortion invariance, being them, local receptive fields, shared weights, and spatial or temporal subsampling (pooling). The idea of connecting units to local receptive fields on the input goes back to the perceptron in the early 1960s, and it was almost simultaneous with Hubel and Wiesel's discovery of locally sensitive, orientation-selective neurons in the cat's visual system [19]. With local receptive fields, neurons can extract elementary visual features such as oriented edges, endpoints, corners, textures, colours, etc. These features are then combined by the subsequent layers to detect higher-order features. Distortions or shifts of the input can cause the position of salient features to vary. In addition, elementary feature detectors that are useful on one part of the image are likely to be useful across the entire image.

According to D. Rumelhart, Y. Lecun *et. al,* this knowledge can be applied by forcing a set of units, whose receptive fields are located at different places on the image, to have identical weight vectors [20] [21]. Units in a layer are organized in planes within which all the units share the same set of weights. The set of outputs of the units in such a plane is called a feature map. Units in a feature map are constrained to perform the same operation on different parts of the image. A complete convolutional layer is composed of several feature maps so that multiple features can be extracted at each location [18].

CNNs comprise three types of layers. These are convolutional layers, pooling layers, and fully-connected layers. When these layers are stacked, a CNN architecture has been formed. A simplified CNN architecture for MNIST classification is illustrated in Figure 6.

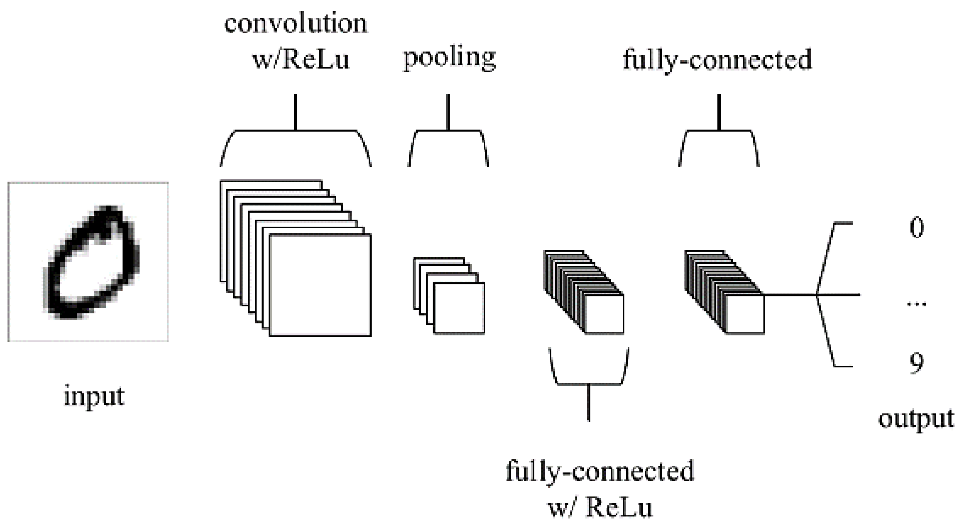


Figure 6- Simple CNN architecture, composed of just five layers. Figure from [22] *displayed with the permission of Keiron O'Shea.*

The basic functionality of the example CNN above can be broken down into the following four key areas:

1. As found in other forms of ANN, the input layer will hold the pixel values of the image.
2. The convolutional layer will determine the output of neurons of which are connected to local regions of the input through the calculation of the scalar product between their weights and the region connected to the input volume. The rectified linear unit (ReLu) aims to apply an 'elementwise' activation function such as sigmoid to the output of the activation produced by the previous layer.
3. The pooling layer will then simply perform down sampling along the spatial dimensionality of the given input, further reducing the number of parameters within that activation.
4. The fully connected layers will then perform the same duties found in standard ANNs and attempt to produce class scores from the activations, to be used for classification. It is also suggested that ReLu may be used between these layers, to improve performance.

Through this simple method of transformation, CNNs are able to transform the original input layer by layer using convolutional and down-sampling techniques to produce class scores for classification and regression purposes.

In the following pages the main blocks of CNNs will be discussed in more detail, as well as some well-known algorithms that use Convolutional Neural Networks.

2.3.1 Convolutional Layers

Before starting to describe what convolutional layers are, it is convenient to explain what a convolution is. A convolution is a mathematical operation that shifts one function over another and measures the integral of their pointwise multiplication. It has deep connections with the Fourier transform and the Laplace transform and is heavily used in signal processing.

The first layer in a CNN is the convolutional layer, which is the main building block and does most of the computational heavy work. The input data is a tensor with shape (number of images, image height, image width, input channels) which is convolved using filters or kernels [23]. These kernels, which are specified with each convolutional layer, are usually small in spatial dimensionality but spread along the entirety of the input depth. When the data reaches a convolutional layer, the layer convolves each filter across the spatial dimensionality of the input to produce a 2D activation map [22].

As the kernel slides through the input, the scalar product is calculated for each value in that kernel. This process involves taking the element-wise product of filters in the image and then summing those specific values for every sliding action as specified in Figure 7.

Supposing one has a certain image as an input and a 3x3 dimension filter. For each 3x3 grid of pixels in the input image, the filter performs an element-wise multiplication as it slides across each 3x3 grid of pixels in the input image, starting on its first pixel. When it reaches the far-right pixels, the filter shifts down one line and continues the iteration through the input image. The result of this element-wise multiplication is the output image.

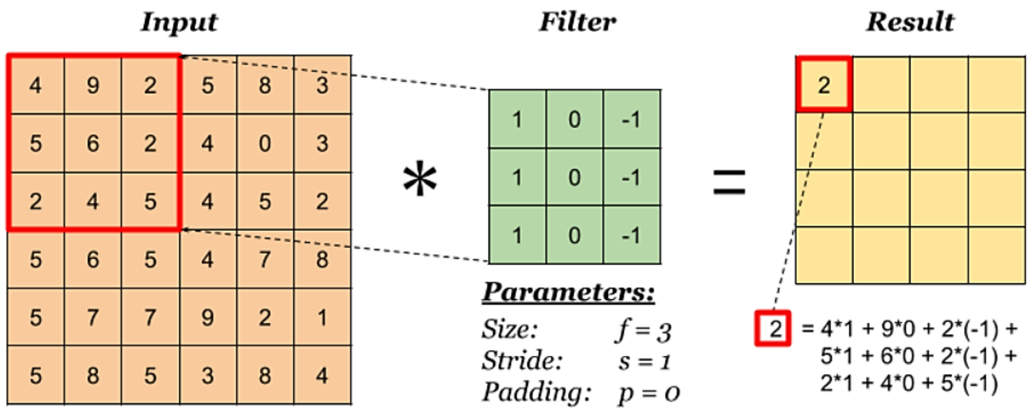


Figure 7- Block of 6x6 pixels (left), random 3x3 filter (centre), output image (right). Figure from [24] *displayed with the permission of Benny Prijono.*

If the original input image is 28x28, the output size is going to be 26x26., after convolving the image, since with the 28x28 image, the 3x3 filter can only fit into 26x26 possible positions, and not all 28x28. Therefore, one gets the resulting 26x26 output. This is due to what happens when the edges of a given image are convolved.

Convolutional layers can significantly reduce the complexity of the model through the optimization of its output. These are optimized through three hyperparameters, being them:

- Depth
- Zero padding
- Stride

The depth of the output volume produced by the convolutional layers can be manually set through the number of neurons within the layer to the same region of the input. This can be seen with other forms of

ANNs, where all neurons in the single hidden layer are directly connected to every single neuron beforehand. Reducing this hyperparameter can significantly minimize the total number of neurons of the network, but it can also significantly reduce the pattern recognition capabilities of the model.

Zero padding occurs when a border of pixels all with values zero is added around the edges of the input images. This adds a kind of padding of zeros around the outside of the image, hence the name zero padding [25]. It allows the use of a convolutional layer without necessarily shrinking the height and width of the volumes. This is important for building deeper networks since otherwise the height/width would shrink as the network gets deeper [24].

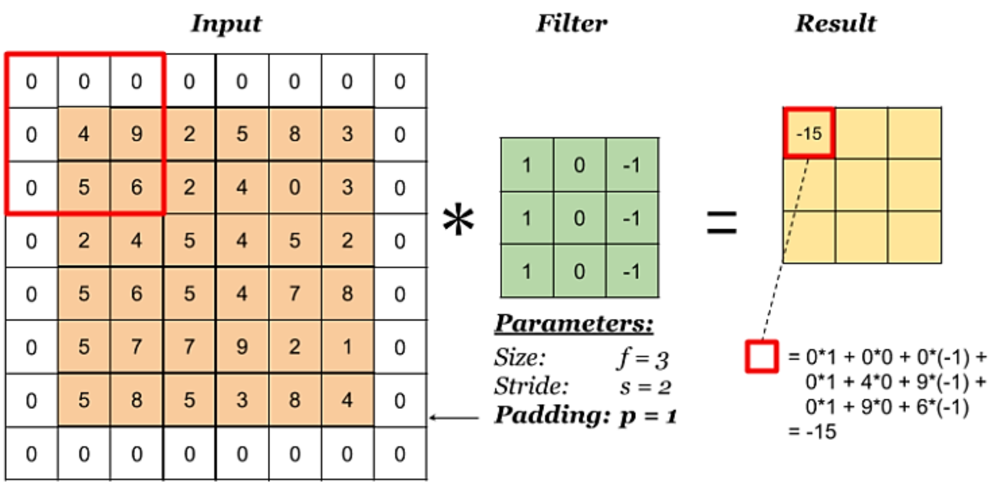


Figure 8- Image with a zero-padding on the borders. Figure from [24] displayed with the permission of Benny Prijono.

The stride is basically how many cells the filter has to move in the input to calculate the next cell in the result. Setting the stride to a greater number will reduce the amount of overlapping and produce an output of lower spatial dimensions [22].

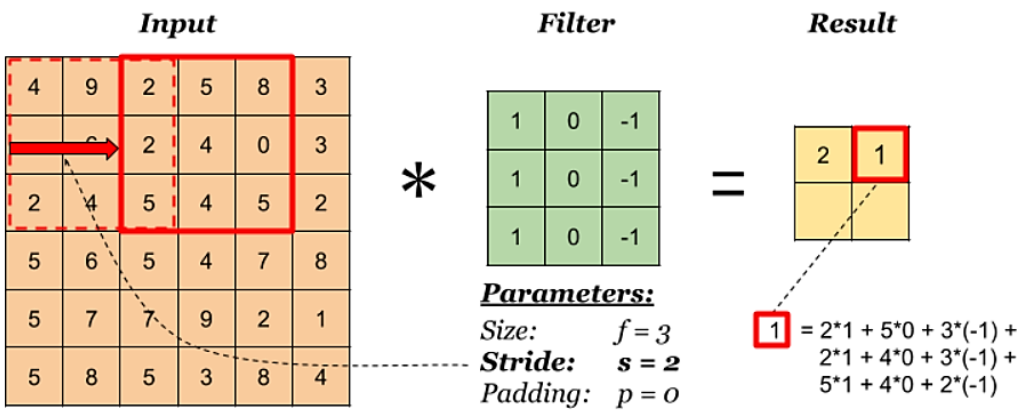


Figure 9- Filter with stride equal to 2. Figure from [24] displayed with the permission of Benny Prijono.

2.3.2 Pooling Layers

Pooling layers aim to gradually reduce the size of the representations, by reducing the number of parameters and the model computational complexity, speeding up calculations, as well as making some of the features it detects a bit more robust. The layer that does the reduction is called the pooling layer or sub-sampling layer. There are two different pooling operations, being them max-pooling and average-pooling, being max-pooling more commonly used [24].

Max-pooling layers operate over each activation map in the input and scale its size using the "MAX" function, with kernels of, for example, 2×2 dimension applied with a stride of 2 along the spatial dimensions of the input. This scales the activation map down to 25% of its original size while maintaining the depth volume to its standard size [22]. The average pooling layer has a similar approach to the max-pooling layer but instead of using a "MAX" function, it uses an average function to scale the size of the activation maps.

Consider the following 4×4 layer. If a 2×2 filter with stride 2 and max-pooling and average-pooling is used, one gets the response in Figure 10-A and Figure 10-B respectively. It is clear that each 2×2 matrix is combined into 1 single number by taking their maximum value (A) and average value (B).

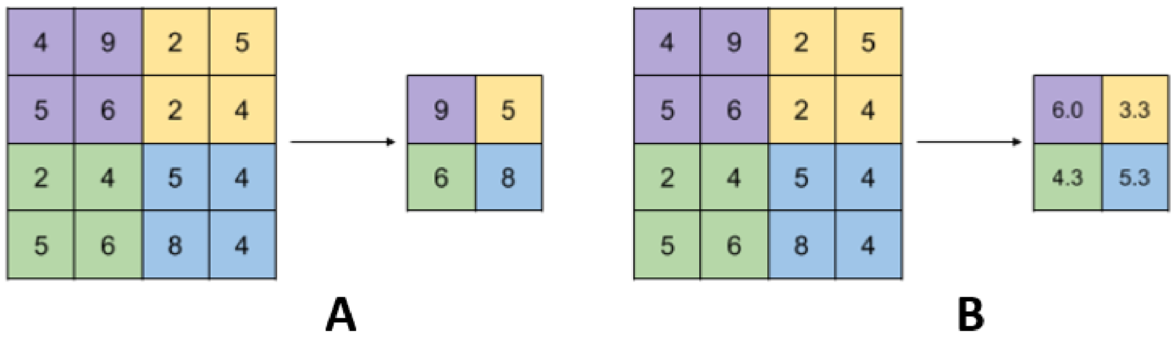


Figure 10- Max-Pooling (A), Average-Pooling (B). Figure from [24] *displayed with the permission of Benny Prijono.*

2.3.3 Fully Connected Layers

Fully-connected layers contain neurons directly connected to the neurons in the two adjacent layers (input and output layers), without being connected to any layers within them. This is analogous to the way neurons arrange themselves in traditional forms of ANN [22] (Figure 3). The fully connected input layer (often called flatten layer) takes the output from the previous layers, “flattens” them, and turns them into a single vector that can be an input for the next stage.

The last fully-connected layer (fully connected output layer) contains as many neurons as the number of classes to be predicted. For example, if a certain network needs to predict 0-9 hand-written digits, the last fully-connected layer will have 10 neurons, since it needs to predict 10 different classes (0-9).

2.4 Convolutional Neural Network’s Architectures

The architecture of a CNN is a key factor in determining its performance and efficiency. The way the layers are structured, which elements are used in each layer, and how they are designed will often affect the performance and accuracy of the model thus causing inconsistent predictions.

2.4.1 VGG-16

This CNN, proposed by K. Simonyan and A. Zisserman from the University of Oxford in “Very Deep Convolutional Networks for Large-Scale Image Recognition” [28]. The researchers investigated the effect of the convolutional network depth on its accuracy in the large-scale image recognition setting. They increased the depth of their architecture to 16 layers (13 convolutional and 3 fully-connected layers), carrying with them the ReLU activation function from AlexNet.

This architecture improves the AlexNet one by replacing large kernel-sized filters (11 and 5 in the first and second convolutional layer, respectively) with multiple 3×3 kernel-sized filters one after another. VGG16 was trained for weeks and was using NVIDIA Titan Black GPU’s [28].

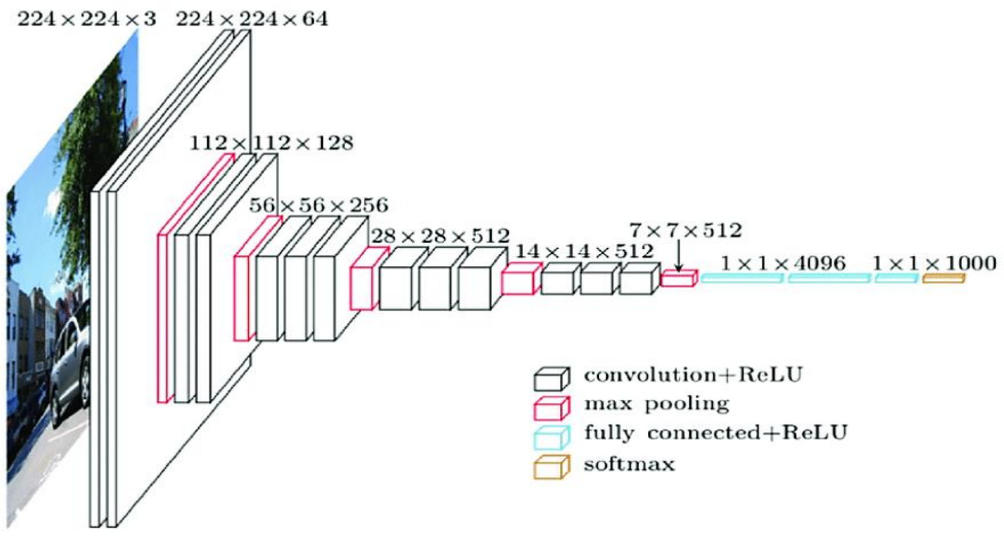


Figure 11- VGG-16 architecture. Figure from [24] *displayed with the permission of Benny Prijono.*

The input for VGG-16 is a 224x224x3 RGB image that passes through first and second convolutional layers with 64 filters of size 3x3 and the same pooling with a stride of 14. The image dimensions changes to 224x224x64. Then a max-pooling layer is applied with a filter size of 3×3 and a stride of 2. The resulting image dimensions will be reduced to 112x112x64.

The third and fourth layers are two convolutional ones with 128 feature maps having a size of 3×3 and a stride of 1. Then there is again a max-pooling layer with filter size 3×3 and a stride of 2. This layer is the same as the previous pooling layer except it has 128 feature maps so the output will be reduced to 56x56x128.

The fifth and sixth layers are convolutional layers with filter size 3x3 and a stride of one. Both used 256 feature maps. The two convolutional layers are followed by a maximum pooling layer with filter size 3x3, a stride of 2, and have 256 feature maps.

Next are the two sets of 3 convolutional layers followed by a max-pooling layer. All convolutional layers have 512 filters of size 3x3 and a stride of one. The final size will be reduced to 7x7x512.

The thirteenth layer is a convolutional layer output that is flattened through a fully connected layer with 25088 feature maps each of size 1×1. The fourteenth and fifteenth ones are two fully connected layers with 4096 units and finally, there is a softmax output layer with 1000 possible values.

2.4.2 Inception V3

The Inception V3 architecture mainly focuses on decreasing computational power by modifying the previous Inception models (V1 and V2). The idea was proposed by C. Szegedy, *et. al.,* in [29]. By comparing the VGG network to the Inception one, the latter has proven to be more computationally efficient in terms of number of parameters generated by the network and the economic cost incurred, such as memory and other physical resources. If any changes are to be made to an Inception Network, they must be carried out cautiously, to make sure that the computational advantages are not lost. Several techniques for optimizing the network have been suggested to lose the constraints for easier model adaptation. These techniques include techniques like factorized convolutions, regularization, dimension reduction, and parallelized computations. This network's architecture is detailed in the following 5 steps.

Factorized Convolutions:

In [29], several alternative ways of factorizing convolutions in different settings were explored to increase the computational efficiency of the solution. Since Inception networks are fully convolutional, each weight corresponds to one multiplication per activation, therefore, any reduction in computational cost results in a reduced number of parameters. This in turn means that with suitable factorization, one can end up with more disentangled parameters and, consequently, faster training. Also, the computational and memory savings can be used to increase the filter-bank sizes of the network while maintaining the ability to train each model replica on a single computer.

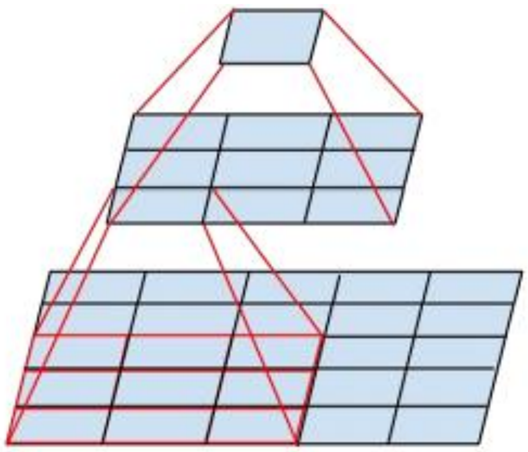


Figure 12- Mini-network replacing the 5x5 convolutions. Figure from [29] *displayed with the permission of Christian Szegedy.*

Smaller Convolutions:

Convolutions with larger spatial filters tend to be excessively expensive in terms of computation. For example, a 5 x 5 convolution with *n* filters over a grid with *m* filters is 2.78 times more computationally expensive than a 3 x 3 convolution with the same number of filters [29]. Of course, a 5 x 5 filter can capture dependencies between signals between activations of units further away in the earlier layers, so a reduction of the geometric size of the filters comes at a large cost of expensiveness. Since this is a vision network, it seems natural to exploit translation invariance again and replace the fully connected layer component with a two-layer convolutional architecture: the first layer is a 3 x 3 convolution, the second is a fully connected layer on top of the 3 x 3 output grid of the first layer. Sliding this small network over the input activation grid boils down to replacing the 5 x 5 convolution layer with two layers of 3 x 3 convolution.

The authors of [29] claim that this setup reduces the parameter count by sharing the weights between adjacent tiles. The illustration of this modification integrated with the adjacent layers can be seen in Figure

13. Several control experiments were carried out and it was concluded that using linear activation functions was always inferior to using rectified linear units in all stages of the factorization.

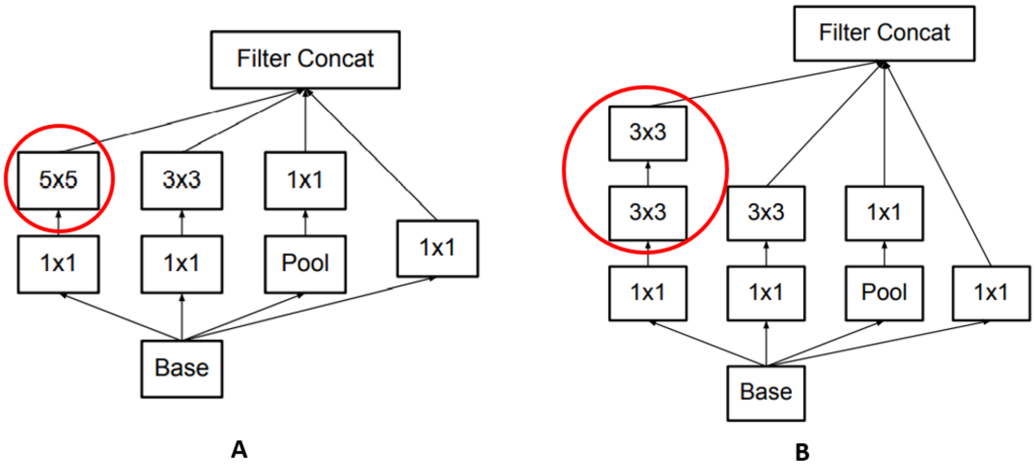


Figure 13- Original Inception module (A). Module after modification suggested (B). Figure from [29] *displayed with the permission of Christian Szegedy.*

Asymmetric Convolutions:

The results obtained in the phase above suggest that convolutions with filters larger than 3 x 3 might not be generally useful, as they can always be reduced into a sequence of 3 x 3 convolutional layers. Given these results, one could consider factorizing into smaller convolutions like for example 2x2. In fact, better results are obtained when asymmetric convolutions are used, i.e., *n* x 1. For instance, using a 3 x 1 convolution followed by a 1 x 3 convolution is equivalent to sliding a two-layer network with the same receptive field as in a 3 x 3 convolution (see Figure 13-B). Despite this, the two-layer solution is 33% cheaper for the same number of output filters, if the number of input and output filters is equal. By comparison, factorizing a 3 x 3 convolution into a two 2 x 2 convolution represents only 11% saving on computation [29].

In theory, one could go even further and argue that any *n* x *n* by a 1 x *n* convolution followed by a *n* x 1 convolution can be replaced, and the computational cost saving increases dramatically as *n* grows. In practice, employing this approach has better results on medium grid sizes (on *m* x *m* feature maps, where *m* ranges between 12 and 20) than on early layers. On that level, very good results can be achieved by using 1 x 7 convolutions followed by 7 x 1 convolutions.

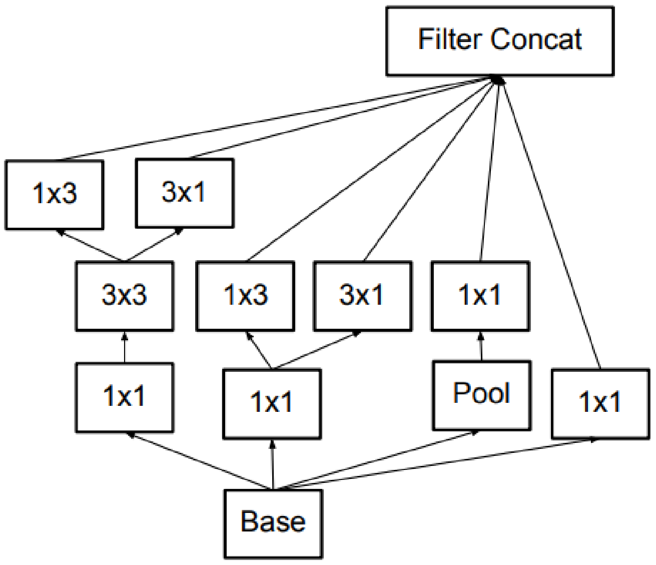


Figure 14- Asymmetric convolution illustration. Figure from [29] *displayed with the permission of Christian Szegedy.*

Auxiliary Classifiers:

An auxiliary classifier is a small CNN inserted between layers during training, and the loss incurred is added to the main network loss. According to [30], auxiliary classifiers, introduced in this same article, improve the convergence of very deep neural networks. By applying auxiliary classifiers early in training did not result in improved convergence. It was not until the end of training that the network with auxiliary branches starts to overtake the accuracy of the network without any auxiliary branch and reaches a slightly higher plateau. In Inception V3, an auxiliary classifier acts as a regularizer.

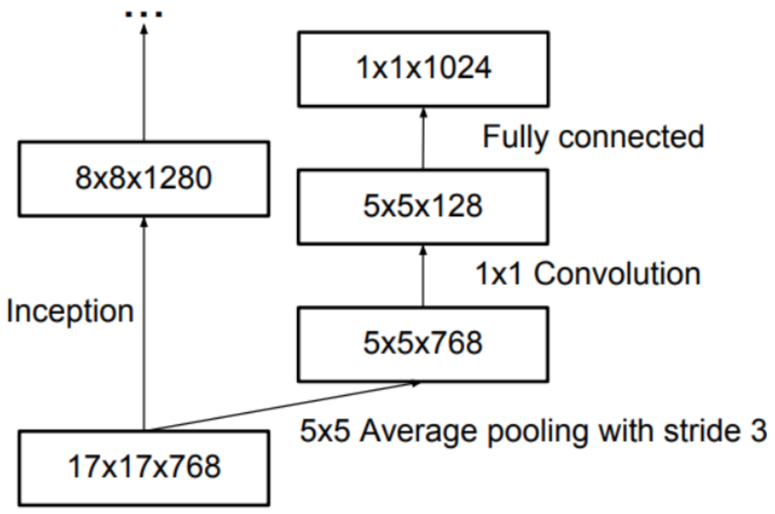


Figure 15- Auxiliary classifier on top of the 17 x 17 layer. Figure from [29] *displayed with the permission of Christian Szegedy.*

Grid size reduction:

Conventionally, convolutional networks used pooling operations to decrease the grid size of the feature maps. To prevent a representational bottleneck, before applying max pooling or average pooling, the activation dimension of the network filters is expanded. For example, starting a *n* x *n* grid with *k* filters, to reach a *n*/2 x *n*/2 grid with 2*k* filters, first, a convolution with a stride of 1 and 2*k* filters needs to be computed and then apply an additional pooling step. This means that the overall computational cost is dominated by the expensive convolution on the larger grid using $2.n^2.k^2$ operations.

The authors suggest a variant that reduces the computational cost even further while removing the representational bottleneck, illustrated in Figure 16.

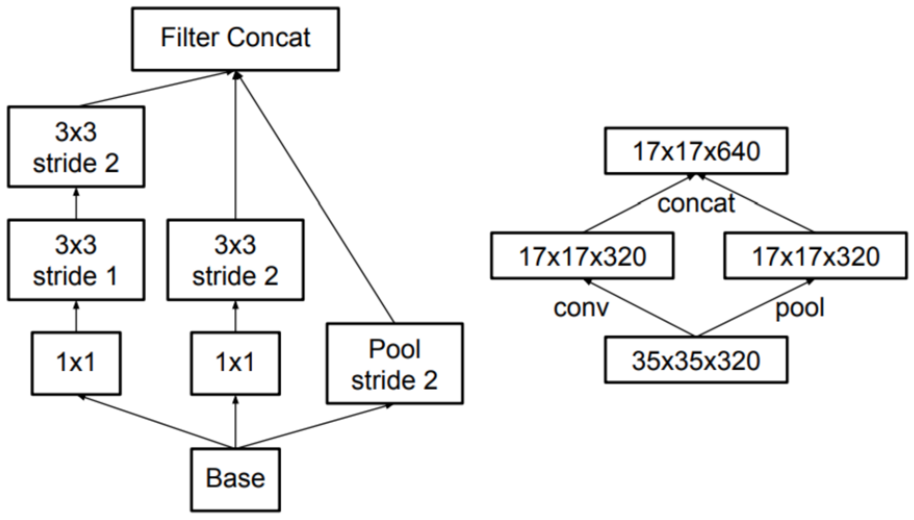


Figure 16- Grid size reduction of the Inception V3. Figure from [29] *displayed with the permission of Christian Szegedy.*

The concepts covered above are illustrated in Figure 17 below.

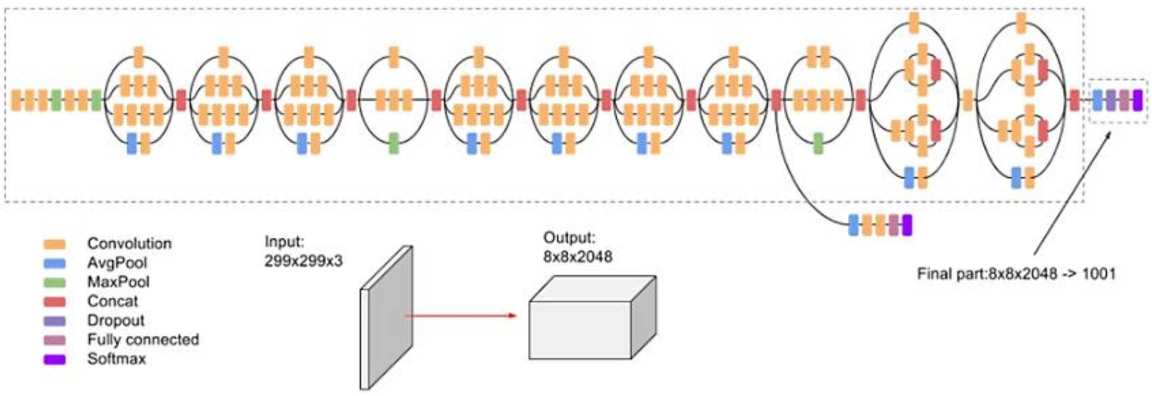


Figure 17- Inception V3 architecture. Figure from [31] *displayed with the permission of Mohammad Shakirul Islam.*

2.4.3 Inception-Resnet V1

Inspired by the performance of Residual Networks (Resnet [32]), a hybrid network incorporating Inception and Resnet modules was proposed in [33] alongside a 4° version of the Inception network. There are two sub-versions of Inception-Resnet, namely V1 and V2. The premise for this proposal is to introduce residual connections that add the output of the convolution operation of the inception module, to the input.

According to [33], Inception-Resnet V1 has similar results compared to Inception V3. Also, the computational cost between these two architectures is very similar, being Inception-Resnet V1 the network with faster training but also the one with slightly worse accuracy. See Figure 18 portraying the full structure of the Inception-Resnet V1.

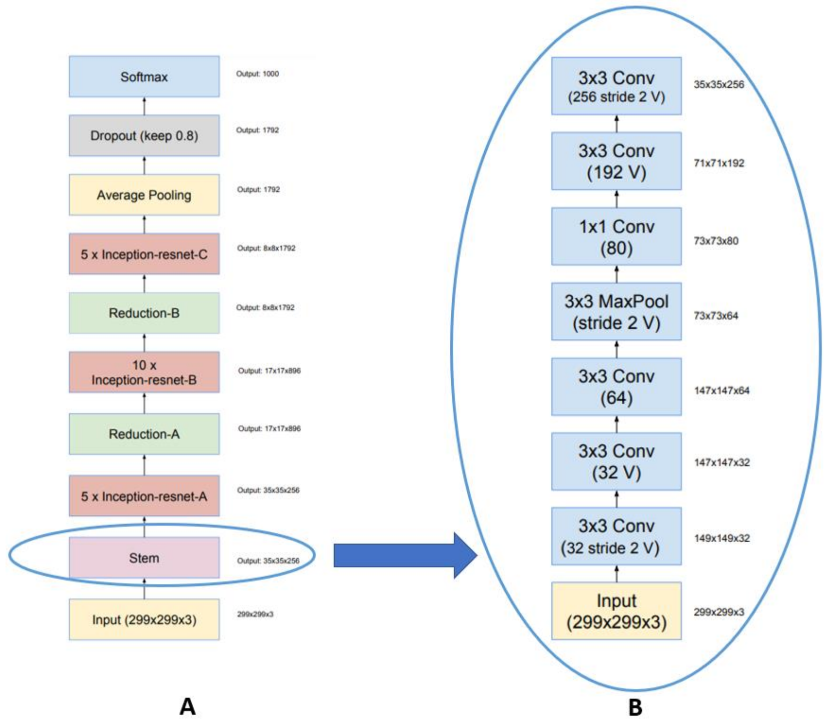


Figure 18- Full structure of the Inception-Resnet V1 (A). Inception-Resnet's Stem. Figure created from [33] *displayed with the permission of Christian Szegedy.*

Inception-Resnet module A, Inception-resnet module B and Inception-resnet module C are the Inception Resnet modules used. In Figure 19, a shortcut connection at the left of each of the modules is observable. This shortcut connection has proved that it is possible to go even deeper in Resnet networks [32].

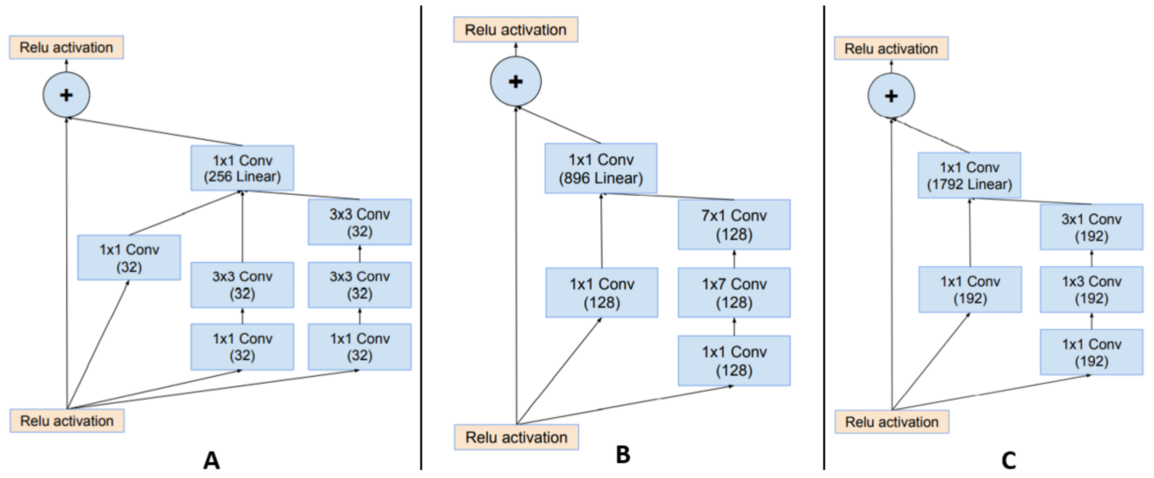


Figure 19- Inception-Resnet modules. Figure created from [33] *displayed with the permission of Christian Szegedy.*

According to the authors, the inception block induces a dimensionality reduction associated. To compensate for this effect, a filter expansion block (1 x 1 convolution layer without activation) is added to scale up the dimensionality of the filter bank before the addition to match the input depth. The pooling operations inside the main inception modules were replaced in favour of the residual connections. However, pooling operations can still be found in the reduction modules, like seen in Figure 20.

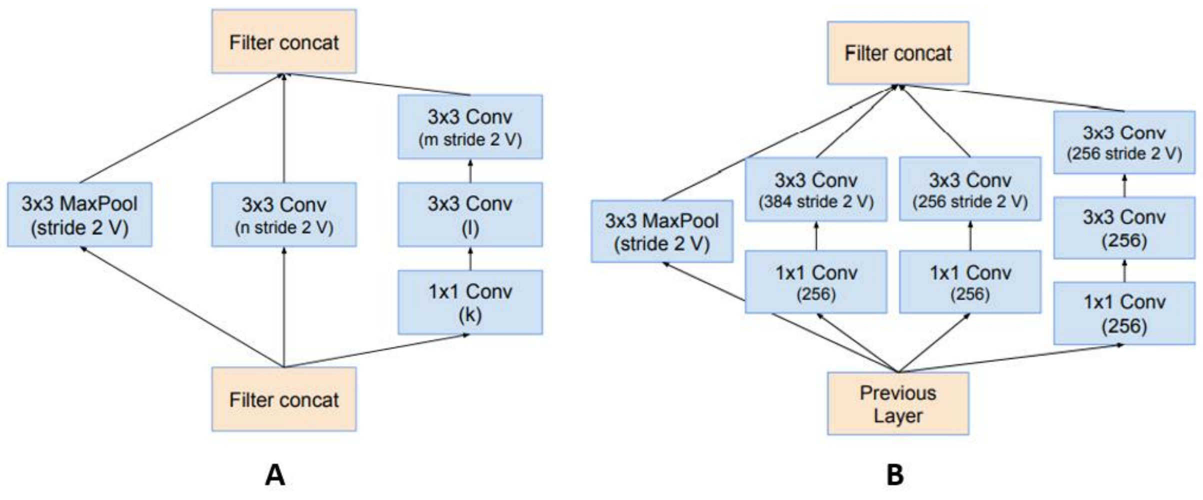


Figure 20- Reduction modules. Figure created from [33] *displayed with the permission of Christian Szegedy.*

In the article, the authors also mentioned that if the number of filters exceeded 1000, the residual variants started to exhibit instabilities and the network would "die" early in training [33], denoting that the last layer before the average pooling started to produce only zeros after a few tens of thousands of iterations. Bearing this in mind, to increase the stability, the authors scaled-down the residuals before adding them to the previous layer activation. The scaling factors, (Figure 21), selected to rescale the residuals were comprehended between 0.1 and 0.3, right before being added to the accumulated layer activations.

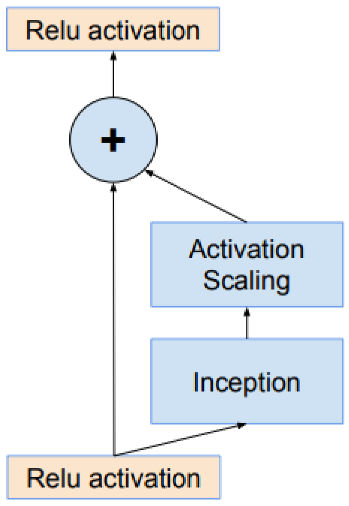


Figure 21- Scaling process. Figure from [33] *displayed with the permission of Christian Szegedy.*

3. State of the Art

Due to the evolution of CNNs and their respective architectures presented above, some CNN-based models developed had outstanding results in the detection field. These models were capable of detecting different breeds of animals, people, different types of plants and objects in a frame, or a sequence of frames (video). This section exhibits several state-of-the-art works and technologies associated with face recognition. It is divided into two subsections: face detection and face recognition, since, to recognize faces, it is necessary to detect them first.

3.1 Face Detection

Face detection is usually the first step towards many face-related technologies. However, face detection itself can have very useful applications like for example detecting faces in frames, in photo-taking, and even guiding the person to align his/her face with the centre of the frame. In the following subsections, two of the most used face detection algorithms are addressed.

3.1.1 Haar Cascade

Object detection using Haar feature-based cascade classifiers is an effective object detection method proposed by P. Viola and M. Jones in their paper, "Rapid Object Detection using a Boosted Cascade of Simple Features" [34] in 2001. It is a ML-based approach where a cascade function is trained from a lot of positive and negative images. It is then used to detect objects or ROIs in other images.

Initially, the algorithm needs a lot of positive images (face images) and negative images (images without faces) to train the classifier and learn what it is supposed to classify and discard. Then, it is necessary to extract features from it. For this purpose, Haar features (Figure 22) are used. The sum of the pixels which lie within the white rectangles is subtracted from the sum of the pixels in the grey rectangles. Two rectangle features are shown in Figure 22-A and Figure 22-B. Figure 22-C shows a three-rectangle feature, and Figure 22-D shows a four-rectangle feature.

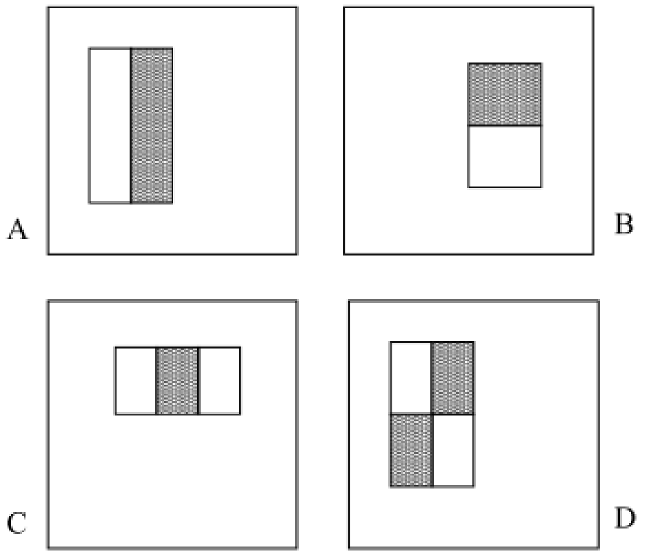


Figure 22- Example of rectangle features shown relative enclosing detection window. Figure from [34] *displayed with the permission of Michael Jones.*

Haar features are similar to a filter in the pooling layer. Each feature is a single value obtained by subtracting the sum of pixels under the white rectangle from the sum of pixels under the black rectangle.

Now all possible sizes and locations of each kernel are used to calculate the features. For example, a 24x24 window can result in over 160000 features. For each feature calculation, one needs to find the sum of pixels under white and black rectangles. To solve this, the authors introduced integral images. These simplify the calculation of the sum of pixels, thus speed the algorithm’s execution time.

But among all the features calculated, most of them are irrelevant. For example, consider Figure 23. The top row shows two good features. The first feature selected seems to focus on the property that the region of the eyes is often darker than the region of the nose and cheeks. The second feature selected relies on the property that the eyes are darker than the bridge of the nose.

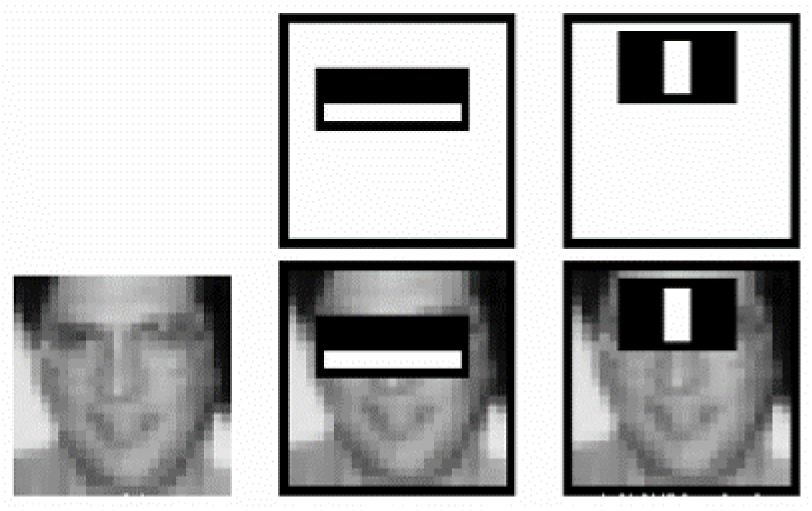


Figure 23- First and second features selected by Adaboost. Figure from [34] *displayed with the permission of Michael Jones.*

For this, each feature is applied to all the training images. For each feature, it finds the best threshold which will classify the faces to positive and negative. But obviously, there will be errors or misclassifications. The features with a minimum error rate must be selected, which means selecting the features that best classify face and non-face images. The process works by giving each image an equal weight in the beginning. After each classification, the weights of misclassified images are increased. Then the process repeats. New error rates are calculated as well as new weights. The process continues until the required accuracy or error rate is achieved or the required number of features is found.

The final classifier is a weighted sum of these weak classifiers. It is called weak because it alone cannot classify the image, but together with other classifiers forms a strong one. According to P. Viola and M. Jones [34], 200 features provide detection with 95% accuracy. Their final setup had around 6000 features, which is a big difference compared to the initial 160000 plus features.

Now applying 6000 features to each 24x24 image, then verifying if it is face or not. To reduce inefficiency and time consumption, the authors proposed a method to check if a window is not a face region, since, in an image, most of the image region is a non-face region. If it is not, it will be discarded and will not be processed again. The focus remains on regions where a face is detected.

To achieve this solution, the authors introduced the concept of Cascade of Classifiers. Instead of applying all the features on a window, features were grouped into different stages of classifiers and applied one by one. If a window fails the first stage, it is discarded. If it passes the second stage of features, the window is applied, and the process continues. The windows that pass all stages are considered a face region.

The authors' detector had 6061 features with 38 stages with 1, 10, 25, 25, and 50 features in the first five stages, respectively. Each classifier in the cascade was trained with the 4916 training faces (plus their vertical mirror images for a total of 9832 training faces) and 10,000 non-face sub-windows using the Adaboost training procedure. According to the authors, on average, 10 features out of 6061 are evaluated per sub-window. This is possible because a large majority of sub-windows are rejected by the first or second layer in the cascade.

Overall, this approach for object detection minimizes computation time while achieving high detection accuracy. The approach used to construct a face detection system was approximately 15 times faster than previous approaches. Some results are shown in Figure 24.

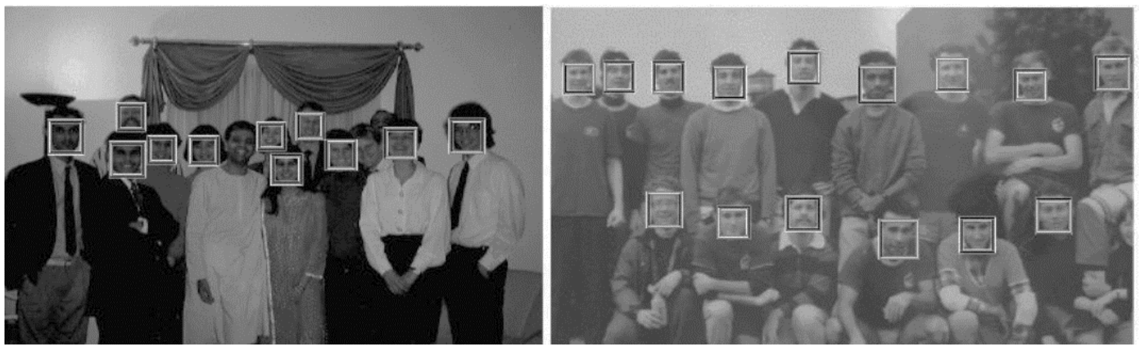


Figure 24- Output of the face detector. Figure from [34] *displayed with the permission of Michael Jones.*

3.1.2 Multi-task Cascaded Convolutional Networks (MTCNN)

MTCNN is a framework developed as a solution for both face detection and face alignment. The process consists of three stages of convolutional networks that are able to recognize faces and landmark locations such as eyes, nose, and mouth.

The article was proposed by K. Zhang, *et al.* [35], as a way to integrate both tasks (recognition and alignment) using multi-task learning. It is divided into three stages. In the first stage, it uses a shallow CNN to quickly produce candidate windows through a fast Proposal Network (P-Net). The second stage refines the proposed candidate windows through a Refinement Network (R-Net), a more complex CNN. And lastly, the third stage, called Output Network (O-Net), uses a third CNN, even more complex than the others, to produce the final bounding box and facial landmark position.

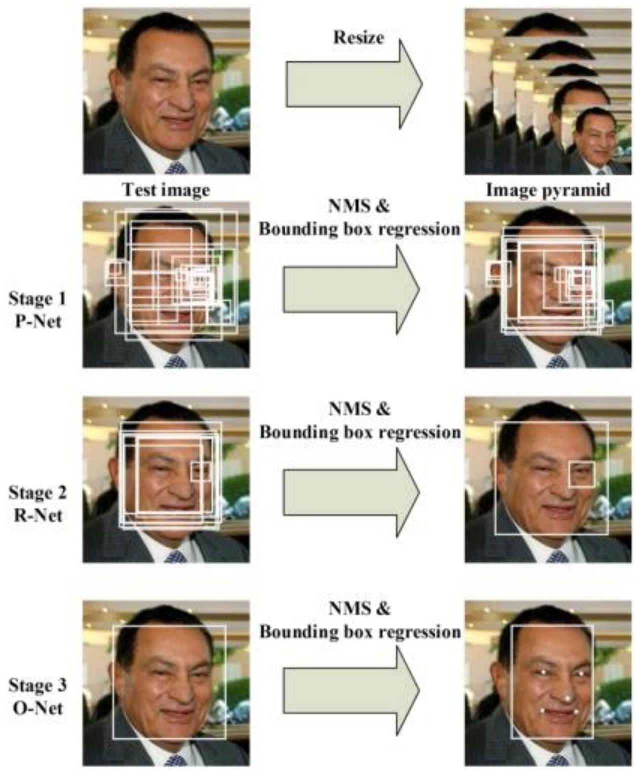


Figure 25- Cascaded framework with three-stage multi-task CNN. Figure from [35] *displayed with the permission of Kaipeng Zhang.*

The first step before stage 1 is to take the image and resize it to different scales in order to build a stacked image pyramid, which is the input of the following three-staged cascaded network.

Stage 1 (P-Net)

This first stage is a fully convolutional network (FCN). The difference between a CNN and FCN is that the latter one does not use a fully connected layer as part of the architecture. This Proposal Network is used to obtain candidate windows and their bounding box regression vectors. Bounding box regression is a popular technique to predict the localization of boxes when the goal is to detect an object of some predefined class, in this case, faces. After obtaining the bounding box vectors, some refinement is done to combine overlapping regions. The final output of this stage is all candidate windows after refinement to downsize the volume of candidates.

Stage 2 (R-Net)

All candidates from the P-Net are fed into the R-Net. This network is a CNN instead of the FCN in stage 1 since there is a fully connected layer at the last stage of the network architecture. The R-Net further reduces the number of candidates, performs calibration with bounding box regression, and employs non-maximum suppression (NMS) to merge overlapping candidates.

The R-Net outputs whether the input is a face or not, a 4-element vector which is the bounding box for the face, and a 10-element vector for facial landmark localization.

Stage 3 (O-Net)

This stage is similar to the R-Net, but this Output Network aims to describe the face in more detail and output the five facial landmarks' positions for eyes, nose, and mouth. Figure 26 illustrates all the 3 stages.

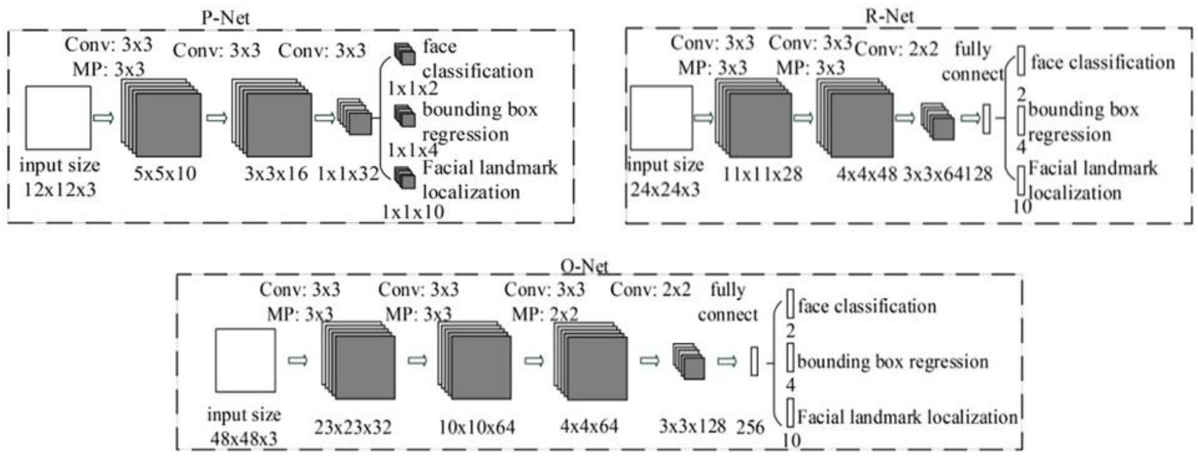


Figure 26- Three stages of MTCNN. Figure from [35] *displayed with the permission of Kaipeng Zhang*

3.2. Face Recognition

Regarding face recognition, some algorithms identify facial features by extracting landmarks, or features, from an image of the subject's face. For example, they may analyse the relative position, size, and/or shape of the eyes, nose, cheekbones, and jaw. These features are then used to search for other images with matching features. As for 3D recognition, the scope of this dissertation, the algorithms associated focus on capturing the depth of the face by using 3D sensors. This is then used to identify distinctive features of the surface of the face. As already mentioned in previous sections, 3D recognition is not affected by changes in lighting or pose, making it a more reliable and robust alternative.

3.2.1 Related Work

Recently, the performance of 2D face recognition systems was boosted significantly with the popularization of CNN's, [36]–[38]. It turns out that methods using CNN feature extractors trained on a massive dataset, outperform conventional methods using hand-crafted feature extractors, such as Local Binary Pattern [39] or Fisher Vectors [40]. Deep Learning approaches require a large dataset to learn a face representation, which is invariant to different factors, such as expressions or poses. Large-scale datasets of 2D face images can be easily obtained from the web. Facenet [38] for example, uses roughly 200 million face images of 8 million independent people as training data. VGG Face [37], assembled a massive training dataset containing 2.6 million face images of over 2700 identities.

There are roughly two kinds of 3D sensors; one is the 3D scanners like the Minolta, producing high-quality results but usually slow and expensive. The other type of 3D cameras are depth ones like the Microsoft Kinect, [41], which although affordable, fast, and compact, have low resolution, low precision, and low reliability, thus providing very noisy results. B. Li, *et al.* [42], used low-quality RGB-D data from a consumer-level sensor to handle face recognition problems under various poses. Both texture image and depth maps were transformed to the frontal view for similarity calculation. However, symmetric filling on texture images involved in the frontalized face images often results in artefacts that may degrade the matching performance. Rather than frontalization on the 2D face image, C. Ciaccio, *et. al.* in [43] generated multiple face images under some predefined poses from 3D face models in the gallery, and the query face compared to all these projected images. Experiments show that the recognition rate was higher than the frontalization based method. To deal with pose variation, the textured images are deformed according to rotation from depth maps and achieved a 69.1% rank-1 recognition rate on the Bosphorus database, [41].

4. Methods and Methodologies

To accomplish the results proposed, primarily it was necessary to ensure correct planning of the tasks to guarantee those same results. The system's prototype development was segmented into six stages, being them the image capture (both RGB and depth components), image processing, dataset processing, the dual model training process, the model's concatenation, and lastly the testing phase. Initially, the development was carried out sequentially in two operating systems, first in Ubuntu 18.04 distribution of Linux and later in Windows 10. This was due to the drivers of the first camera used (the Kinect) only being compatible with Ubuntu. The first code script for capturing the images of the dataset was entirely written using PyCharm IDE in Ubuntu using Python as the programming language. Here, the main libraries used were OpenCV and Freenect. The latter one is responsible for establishing the connection between the Kinect's hardware and the software written, thus aiding in the frames handling from the different camera sensors.

OpenCV on the other hand is used to acquire the subject's region of interest (ROI), as well as handling the necessary image processing to ensure a quality dataset. The code script made in Ubuntu has a guided user-friendly interface to guide a subject to correctly capture his/her face. This was made like this to guarantee a uniform set of quality images throughout the whole dataset.

It is extremely important to point out that a second 3D sensor was used after the Kinect. The Intel RealSense D455 was the camera used to capture the final images of the LAR's subjects, and also it is the camera that will be integrated into CHARMIE the anthropomorphic robot, for test and real-world usage. More on these cameras will be addressed in the upcoming sections.

Upon capturing the images, it is time to start building the neural networks and respective training processes. To make this possible, TensorFlow along with Keras frameworks were used. Both of these are open-source platforms that allow the implementation of AI (ML/DL) solutions. Both contain several built-in functions that allow to conveniently implement deep neural networks through the creation of layers (convolutional, pooling, etc.) with their respective variables or tensors, as well as specifying filter dimensions for the layers and other relevant parameters. In addition, it contains functions that define the methods for the training processes, like the classifiers, optimizers, call-backs, and loss functions. All this information and processes will be detailed in the following sections.

4.1. Model Framework

It was crucial to separate the problem into different steps performed sequentially with some parallelism between them. The steps taken were the image acquisition, image processing, dataset preparation, followed by the individual training process of 2 independent neural networks, model concatenation, and then the final testing/execution stage like shown in Figure 27.

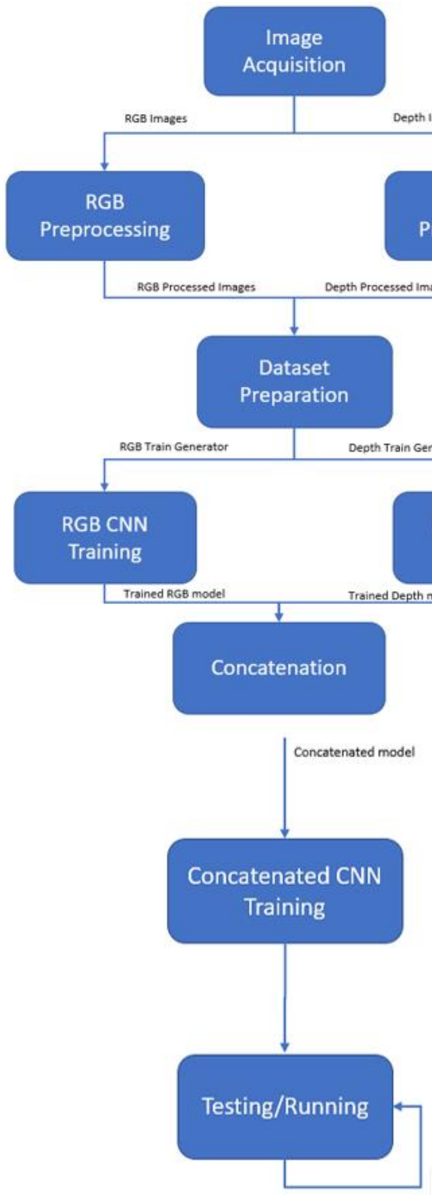


Figure 27- System Framework.

As observed in the image above, the execution of the stages is sequential with some of them being executed in parallel, all the way through the last stage which runs on a continuous loop and it is where the frames are constantly being acquired by the camera and each frame is being passed on to the network trained in the previous stage, to make predictions in real-time. It runs on a continuous loop until the program forcibly ends through a user command. This only happens when the network is already accurately trained to recognize the faces of all the subjects in the dataset, a process that happens during the training phase. As for this stage, the elaboration of the deep learning model is built upon trial and error and by adjusting its hyper parameters according to some crucial results like the accuracy and loss on the validation set and the overall performance on new and unseen images. Regarding the image processing and dataset preparation stages, here is where all the processing such as the isolation of the ROI is going to be applied to the images, as well as to the dataset itself. As for the image acquisition, this is the only stage that will take place in Linux's Ubuntu 18.04 distribution in an initial approach where the Kinect camera was used, and later in Windows 10 O.S. with the acquisition of the overall better-performing Intel RealSense D455 camera. In the following sections, the framework and its corresponding phases will be addressed in more detail.

4.2. Hardware

During the development of the current dissertation, two 3D sensors/cameras were used, to capture both the subject's RGB and depth images to build the dataset. Initially, a Microsoft Kinect 360 (Figure 28) was used. This camera was used to capture the images for the first dataset. Later, having the opportunity to use LAR's Intel RealSense D455 (Figure 31), the development using the Kinect was discontinued, for the reasons revealed in the upcoming sections.

4.2.1. Microsoft Kinect 360

The choice for this camera was obvious. It was the only one available at the time the project began, and since it has a 3D sensor that captures depth data, it is a simple solution to turn to, since it is practical and easy to use alongside the python language.

Unfortunately, it is an outdated camera, released in 2010, meaning it has inferior hardware and other features compared to new and more modern alternatives. This factor reflected not only on the image quality provided by the camera but especially in the amount of processing that had to be applied to the images captured. The weaker the image quality is, the more processing has to be applied to the images. Besides, since the camera is older there is a lack of drivers to establish a connection between software and hardware, suggesting a lack of support by other developers for building software and applications for the Kinect's hardware and subsequently, an absence of functions that provide more interaction with the hardware. The Kinect 360's camera layout can be seen in Figure 28.

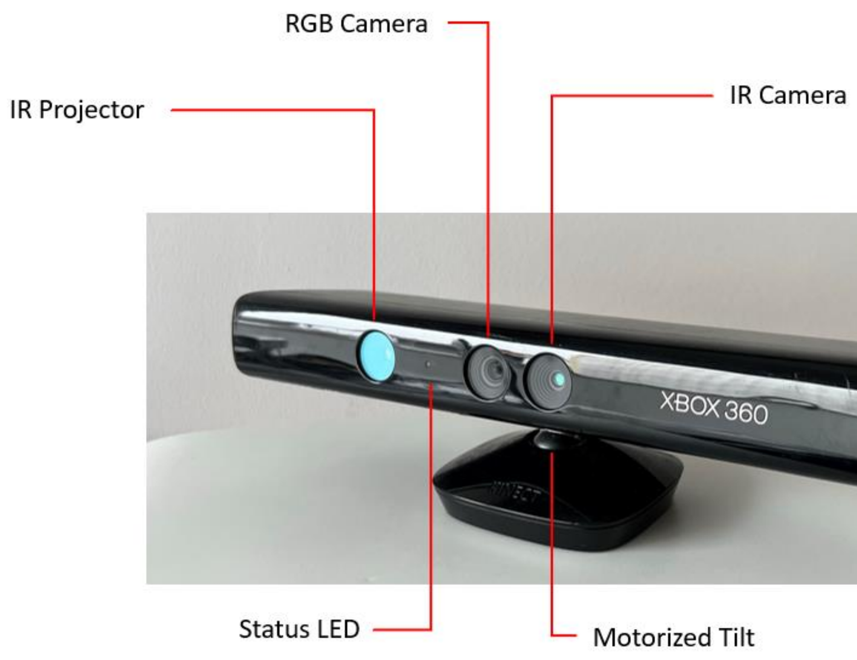


Figure 28- Microsoft Kinect 360.

Some of the Kinect 360's specifications are represented in Table 1 below:

Table 1- Microsoft Kinect Specifications.

RGB Image Resolution	640x480p @30 fps
Depth Image Resolution	320x240p
Min. Depth Distance	≈75 cm
Max. Depth Distance	≈4 m
Depth Resolution	≈1.5mm at 50cm, ≈5cm at 5m
Horizontal Field of View	57°
Vertical Field of View	43°
Latency	≈90 ms with processing
Aspect Ratio	4:3

In order to establish the connection between the camera's hardware and the computer used, it was necessary to install the Libfreenect driver. The installation only took place in the Ubuntu 18.04 distribution due to driver incompatibility with Windows, as mentioned before. The driver's library contains a Python wrapper, (Freenect) that allowed the use of simple functions to extract both RGB and depth components from the camera's sensors. This easily allowed to use the mentioned components with the OpenCV library. Figure 29 represents the raw components of data of one subject from the Kinect dataset, RGB, and depth components. As shown, primarily from Figure 29-A, the actual resolution is quite poor, with a resolution of 640x480 pixels, especially by today's standards. Similarly, the depth component also has a poor resolution. On a depth map of a human face, like the one in Figure 29-B, the pixels are as darker as closer the subject is to the camera, hence, the nose is darker than the eyes. Also, another indication of the lack of quality of the Kinect sensor is the noise introduced in each depth map, represented in Figure 29-B by the white area of pixels surrounding the subject.

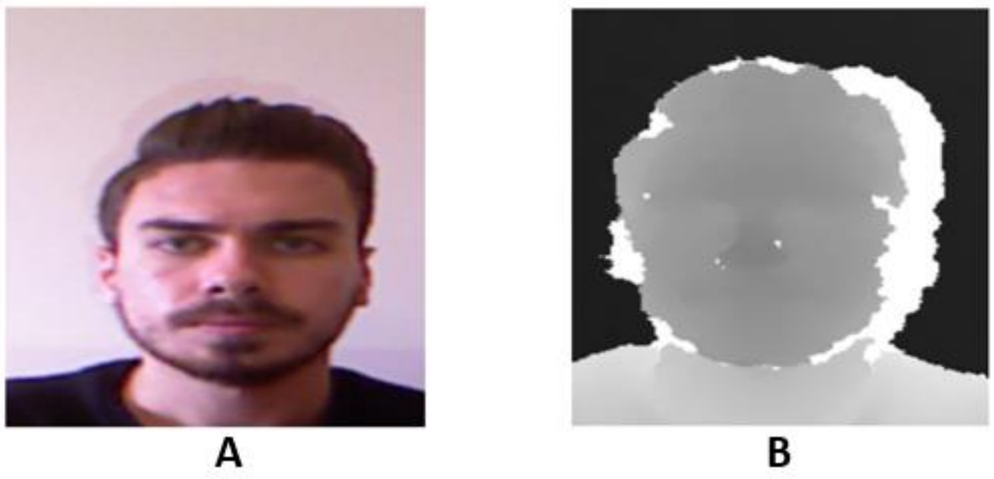


Figure 29- Subject 1 on a straight pose of the Kinect dataset. RGB component (A), Depth component (B).

This always happens when there is some difference between the depths of surfaces from the camera. In this figure, it is about the depth difference between the face and the wall behind. The same applies to Figure 30, an overall inferior image quality on the RGB image and noise introduction in the depth image. These problems were partially solved using image processing.

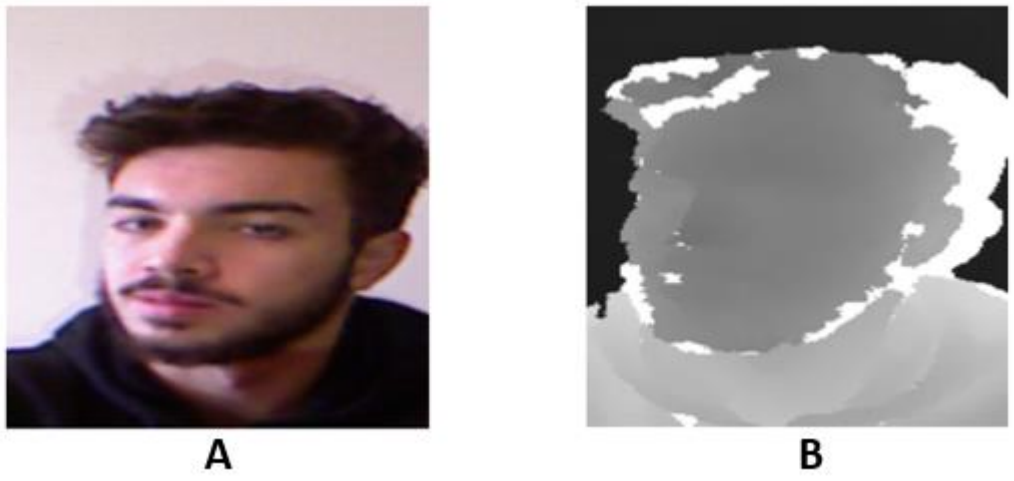


Figure 30- Subject 2 on a left-looking pose of the Kinect dataset. RGB component (A), Depth component (B).

One last important thing to point out regarding the camera is that all images had to be taken at approximately 75 cm distance from the subjects, because the depth sensor has a minimum interval distance until it starts capturing the image with the least possible noise. Also, since the intention is to work with small details, like noses, lips, forehead distance, and eye concavity, the subject's distance relative to the camera cannot be too far away, otherwise these same details would be almost unnoticeable, and the whole face would resemble a uniform grey matter. That could become a problem to the future learning performance of the neural network. The images above were already taken at ≈75 cm distance.

4.2.2. Intel RealSense D455

The camera used to replace the Microsoft Kinect was the Intel RealSense D455. It was acquired by LAR to be used in several computer vision projects. Due to its great image quality, framed in today's modern image quality standards, it was an immense improvement over the Kinect, and the results speak for themselves. The RealSense is more compact than the Kinect, has USB-C connectivity, and can be used in all operating systems.

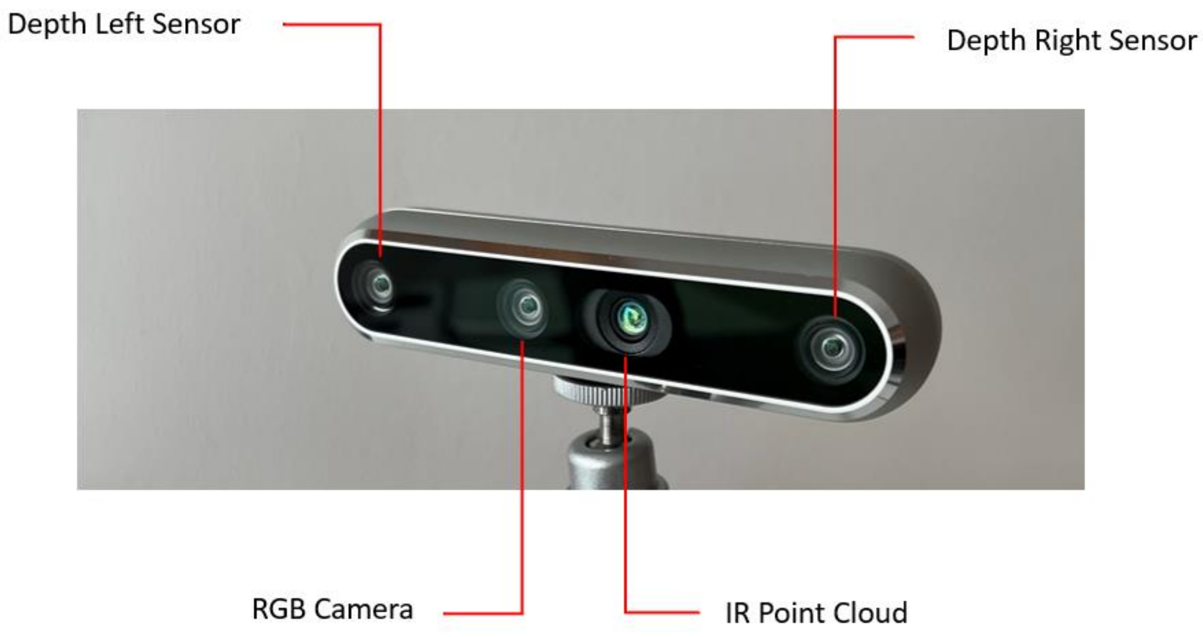


Figure 31- Intel RealSense D455.

Besides, it can be used as an external USB camera without the manual installation of specific drivers which turned out to be a bit tiresome, upon installation in the Kinect. Table 2 shows a set of specifications for the Intel RealSense D455.

Table 2- Intel RealSense D455 Specifications.

RGB Image Resolution	1280x800p @30 fps
Depth Image Resolution	1280x720p @90 fps
Min. Depth Distance	≈52 cm
Max. Depth Distance	≈6 m
Depth Accuracy	<2% at 4m
RGB Horizontal Field of View	90°
RGB Vertical Field of View	65°
Depth Horizontal Field of View	86°
Depth Vertical Field of View	57°
Aspect Ratio	16:9

As previously mentioned, there were no issues with driver compatibility regarding this camera. The drivers are automatically installed the first time the camera is plugged in, regardless of the O.S. This is a big positive aspect since one can use the D455 even in portable development kits like the Raspberry PI, regardless of the O.S. it is running. Besides, it includes various wrappers like C, C++, and Python, among others, and is simple to interact with the OpenCV framework. This allowed keeping a portion of the previous code scripts developed with the Kinect sensor. Figure 32 shows both raw images of a subject captured by the Intel RealSense, RGB and depth images.

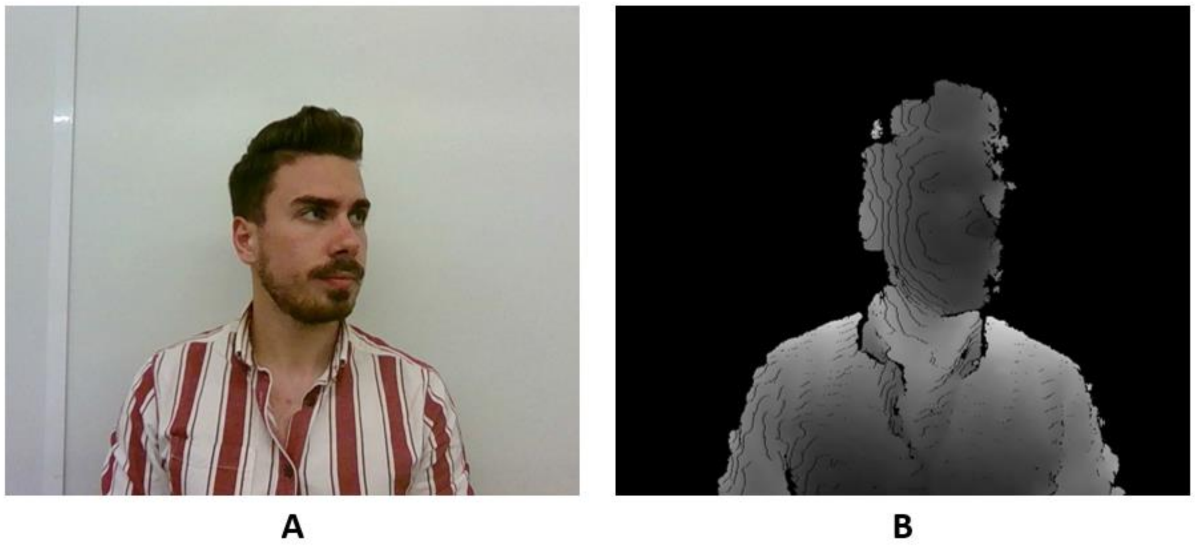


Figure 32- Subject 1 on a right-looking pose of the RealSense dataset. RGB component (A), Depth component (B).

As one can see, the difference in image quality between both cameras is unmistakable. The RGB image does not have a blurry-like effect, and the colours are accurate. As for the depth image, it is clear that it has a good distinction between several face elements through their distance, and there are no signs of white-pixelated areas.

Another peculiarity related to this camera is its ability to accurately measure the distance to a pre-defined target. This revealed very useful to the image capture algorithm since it forced the subject to maintain a certain distance to the camera before making the capture. That same distance, which was 58 cm, is the best possible distance to secure quality depth images across the dataset.

Another subject of the dataset, this time in a different pose from the one in Figure 32 is illustrated in Figure 33. The quality of the images remains consistent between figures.

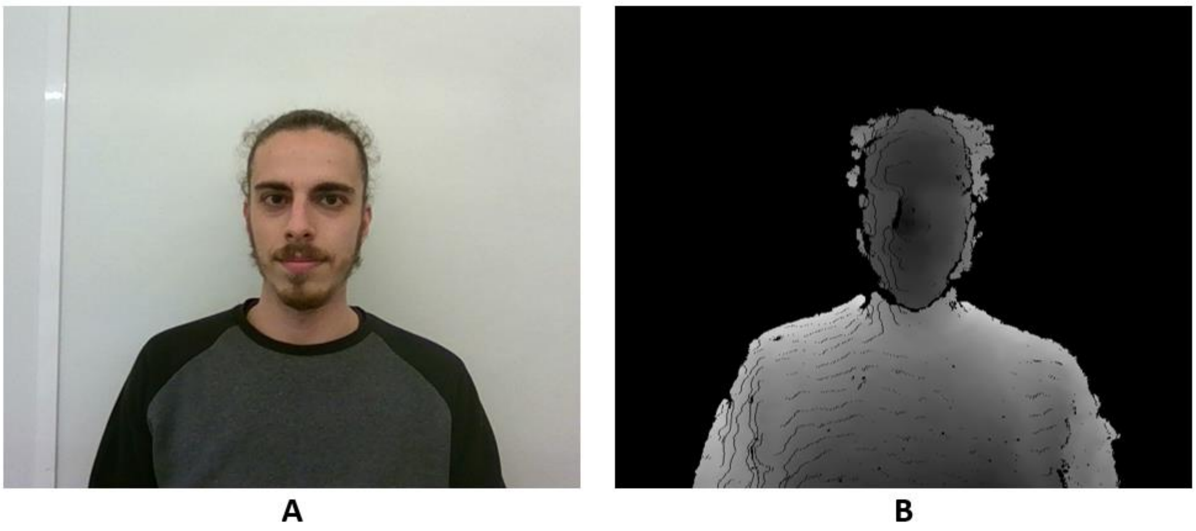


Figure 33- Subject 2 on a straight-looking pose of the RealSense dataset. GB component (A), Depth component (B).

4.3. Image Acquisition

This is the initial stage of the process. The one that captures and saves the raw images of the subject, both RGB and depth in a local storage. This segment of the algorithm was envisioned to be user-friendly, aiming to assist the subjects to capture their faces in several angles and poses using on-screen guidance. There are two code scripts in this stage, the training set capture, and the validation set one.

4.3.1. Training Set

To perform the image acquisition part, OpenCV was mainly applied. This library is widely used since it has support in many operating systems and has a solid follow-up in the online community. It also supports Python, which is the language used throughout the dissertation work development.

The goal is to extract a region of interest (ROI), which is nothing but the subject's entire face. Immediately after starting the program, the user is prompted with a dialog box to type his/her name. The name typed in corresponds to a label or class for the elaboration of the dataset in the following stages. A dialog box provides the user the option to either type his/her name or to cancel the process. A folder with this user's name is automatically created in a local directory.

This folder is intended to store both RGB and depth images, each in a different sub-directory. Besides the creation of the directory, the program checks if the name inputted already corresponds to an existent directory. If that case verifies, all the content in the pre-existent folder is permanently deleted in order to store the newly captured images, each category in its respective sub-directory.

Since one of the cameras lacked the ability to estimate the distance to a given target (Kinect), it was required to overcome this absence through software. Therefore, there were slightly different versions of the algorithm, one for each camera. On both camera's algorithms, immediately after inputting the name in the dialog box, two windows portraying the subject would appear, corresponding to what the camera was capturing. Each of this windows correspond to the RGB and depth components. Although the algorithms had a similar concept of capturing the images, they were developed through distinct approaches, since the RealSense being a more advanced system, allowed to implement a more simple and straightforward solution. For a more detailed description on the image capture process for the training set, see Appendix A.

4.3.2. Validation Set

To train the CNNs it is recommended to use a validation set. According to S. Russel and P. Norvig in [44], a validation set is a subset of the training set that can be used to obtain an early estimate of the model performance. In the words of B. D. Ripley in [45], a validation set is a set of examples used to tune the parameters of a classifier, for example, to choose the number of hidden units in a neural network.

The image capture algorithm for the validation set follows an approach similar to the one used in the training set, but this time it captures fewer images. According to the AI scientific community, the validation set contains approximately 20% of the images of the dataset. Figure 34 shows a sample of the validation set from the Kinect camera.

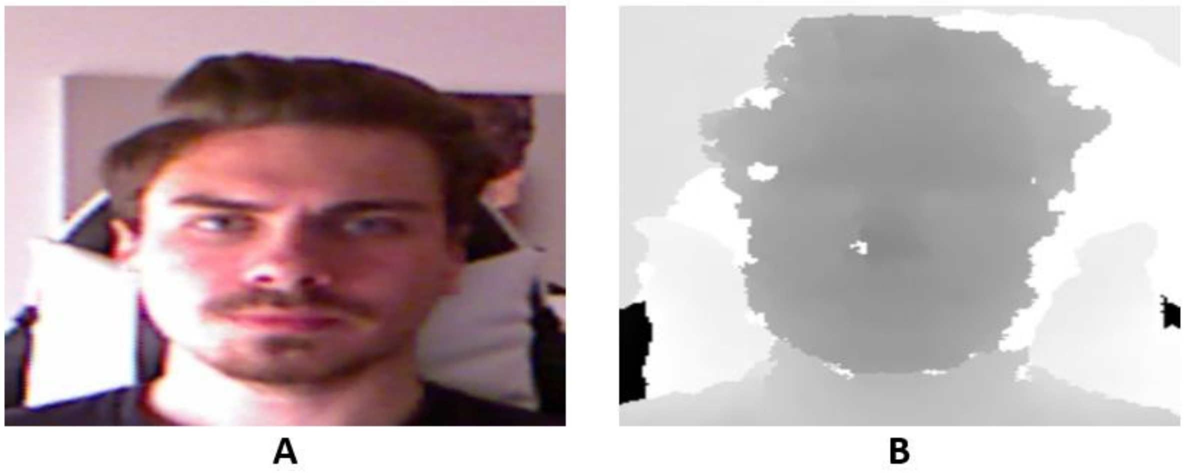


Figure 34- Sample of the validation set taken with the Kinect dataset.

An important note here, this time the images were taken in different environments, hence the multiple distances detected in the depth component. All the iterations used for the training set images were also used for this capture, as well as other functionalities such as the timer, among others. The capture created with the Kinect totalized 228 images for each subject. Therefore, by adding up all the images of the dataset, there are 1712 images, 1256 for training, and 456 for validation, bearing in mind that the Kinect dataset had only two categories, i.e., two subjects.

Regarding the validation set captured by the Intel RealSense, this was also made using fewer images than its respective training set. With this camera, 3 distinct captures were taken for each subject, each one on three different days. This procedure helped to create a larger diversity in the faces of the dataset despite being the same person. An example of this is illustrated in Figure 35, where one can observe distinct images of a subject taken at distinct locations and with different poses while keeping the same distance and height related to the camera. This applies to both RGB and depth components.

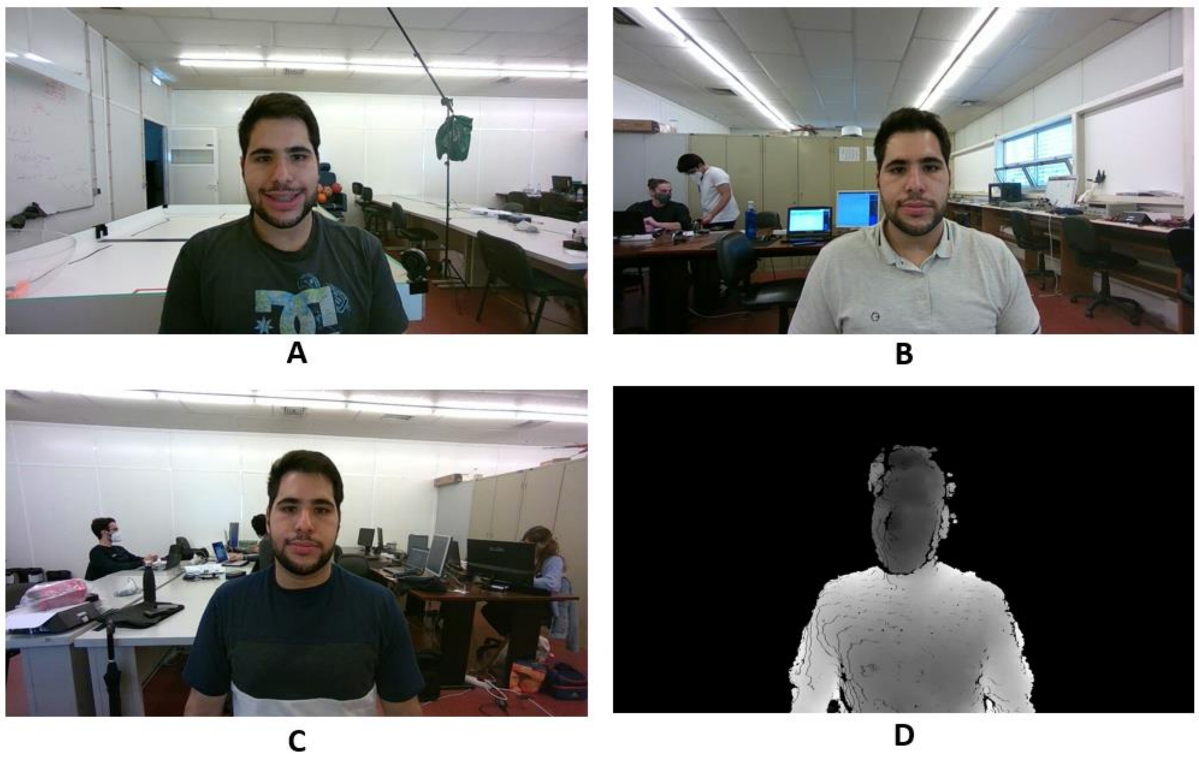


Figure 35- Images of a subject taken on three different days with different poses.

Each day, a subject had 21 different images, two from each of the iterations (1 to 7). This meant a total of 63 images per subject for each component of the image.

The total number of training images for the RGB and depth training sets was 6612, 3306 for each image component. Regarding validation, there were 756 images, 378 for each component of the image. These values were totally influenced by the number of classes or subjects in the dataset. The Intel RealSense had 6 subjects.

4.4. Image Processing

The second stage of the methods used to achieve the purpose of this dissertation is the image processing one. One felt the need to do image pre-processing before leaping to the convolutional neural networks training since the raw images were not suitable for training since their overall quality and noise incidence could heavily compromise the final results. Also, this kind of procedure is largely advisable throughout the AI scientific community.

In this stage, like in the previous one, the type of actions to perform to obtain favourable results was directly related to the camera with which the images from the image acquisition phase were taken. Consequently, there were different types of processing applied to the images, being the Kinect's images the ones that were heavily processed,

Initially, the processing was made using Jupyter Notebook which is an open-source web application that allows to create and share documents that contain live code, equations, visualizations, and narrative text. It can be used in data cleaning, numerical simulation, statistical modelling, data visualization, machine learning, among other areas. Jupyter was launched through Anaconda Navigator, a GUI included in Anaconda distribution that allows to launch applications and easily manage packages, environments, and channels without using command-line commands. Later, the code developed in Jupyter Notebook was transferred to PyCharm IDE just to have all the code in one single IDE.

In both cameras, the processing occurred by opening the directory of either RGB or depth images for a given person, and then for each image apply the necessary processing functions and ending the process by saving the images with a .jpg extension in a different folder.

Concerning the Kinect camera, it was necessary to perform several processing steps since the images had very low quality. Besides this, RGB and depth images were not aligned. This can be observed in Figure 29 and Figure 30, where the depth image looks misaligned relative to its respective RGB image. This had to be manually adjusted along with the remaining processing.

The libraries used were OpenCV and PIL, both for opening and applying filters to the images, and other libraries like Matplotlib, Regex, and Open3D, only for visualization and result gathering purposes.

For the RGB images of the Kinect training set, firstly, a sharpening operation was made using the 'ImageEnhance' module available in the PIL library. This sharpens the images by reducing the blurriness present in the Kinect's images. In addition, a shift in the image was made in order to align it with its respective depth image.

This shift moves all the pixels, in this case to the left, and the area on the right, that previously had had pixels gets filled with black pixels. Besides this shift to the left, a shift upwards was also made. The results before and after applying both the sharpening and the shifting can be seen in the figure below.

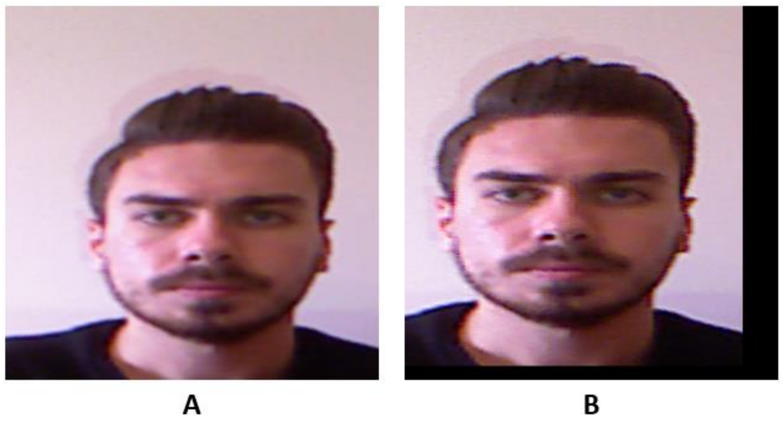


Figure 36- RGB images of a subject before (A) and after (B) applying image processing.

Besides the shift that is clearly visible in Figure 36-B, the sharpening slightly helps with the quality of the image itself even slightly though. The side effect of trying to unblur an image is the introduction of unwanted noise to it.

As for depth images the algorithm started by removing noisy areas like the white zone in Figure 37-A. This reduced the noise introduced by the sensor, thus making a more prominent distinction between close and distant objects/targets. The algorithm made this by examining all pixels. If a pixel had a value over 250, that pixel would be assigned a value of 0, i.e., black.

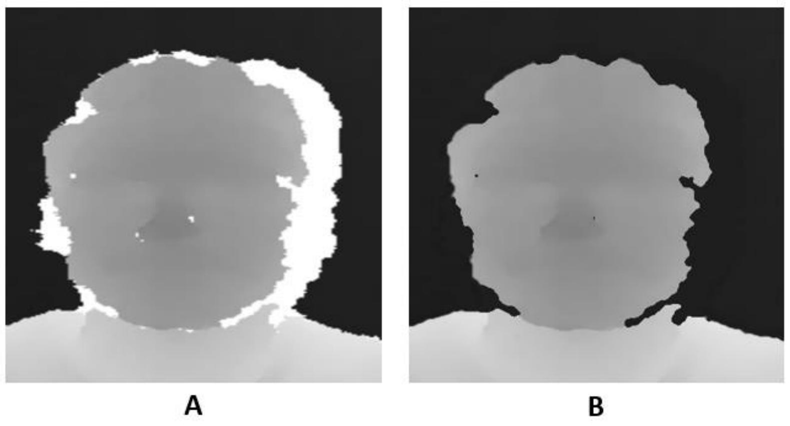


Figure 37- Depth images of a subject before and after applying image processing.

Moreover, an erosion (kernel size of 3) followed by a median blur operation was carried out. This helped to smooth the depth image by eroding the boundaries of a foreground object, i.e., the face, removing small smudges that could appear, and by applying a small blur. The results before and after applying both these operations can be seen in Figure 37.

The processing operations allowed to visualize the point cloud of a subject. A point cloud is a set of data points in space that represent the shape or object in 3D, with a set of x, y, and z coordinates. Considering the 2D image along with a depth image, it is possible to fuse them together and draw the geometries to make a 3D visualization of the face of a given subject. This point cloud was made using the Open3D library. This latter one has methods that require passing to a function, both components of the image, and it returns a real-time point cloud visualization of those components. Below are some illustrations of the point cloud obtained from one subject.

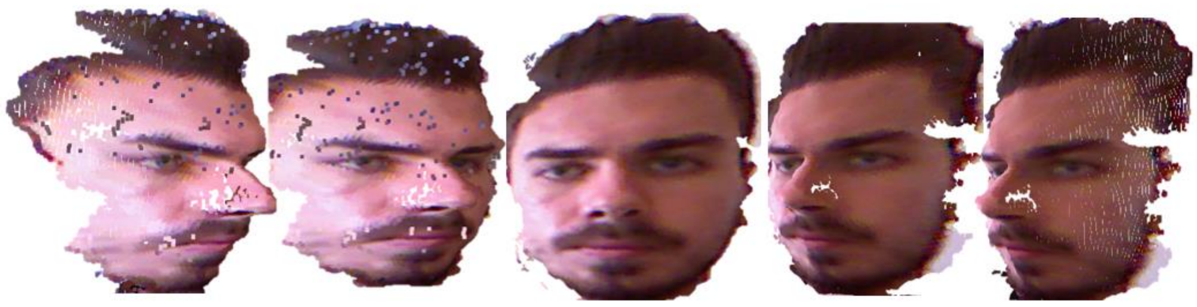


Figure 38- Point cloud of a subject of the dataset.

This point cloud was only possible after aligning the RGB with the depth, otherwise, it would not look much like a 3D face, since the image components were shifted from each other. Bearing in mind the poor quality of both image components, it was expectable that the point cloud gathered from the Kinect camera would also have inferior quality, and that can be confirmed in Figure 38. The random pixels on the images on the left of the figure above derive from noise, and even some misalignment between RGB and depth images.

It is important to clarify that this point cloud, and eventually other point clouds that were built, are only illustrated here for visualization purposes besides exhibiting the potentialities of joining an RGB image with its respective depth image. The point clouds were not used to satisfy this dissertation's purpose.

Furthermore, the same image processing algorithm for the Kinect's images was also applied to its validation set, in a similar way the training set ones did.

As for the images captured with the Intel RealSense, these had minimal processing. It was concluded that the raw images were very good, thanks to the superior image quality offered by this camera. This algorithm started by performing a concatenation between the sub-directory of the raw training set images and the images themselves (RGB and depth), similarly to the one used with the previous camera's captured images. This resulted in a list of images with the directory appended. This method was chosen since OpenCV reads images by taking the image directory as an input. This proved to be an easier method to use. The algorithm also checked whether the name of a subject was present in the directory of subjects (). If this condition does not verify, the program returns an error message to alert that the subject's images are not present in the directory. Also, if someone tried to reprocess a certain subject's images, it would send an alert message, apart from terminating the program.

After the first condition mentioned above was verified, the program started by simultaneously reading the first image in each sub-directory. Then, for each RGB image the Haar Cascade algorithm was executed. As referred previously, this returned 4 coordinates based on the detection of a face in a frame. These coordinates form a bounding box surrounding the subject's face. The interior of this bounding box was the ROI. Afterwards, the ROI was cropped, which meant that everything outside it would be discarded.

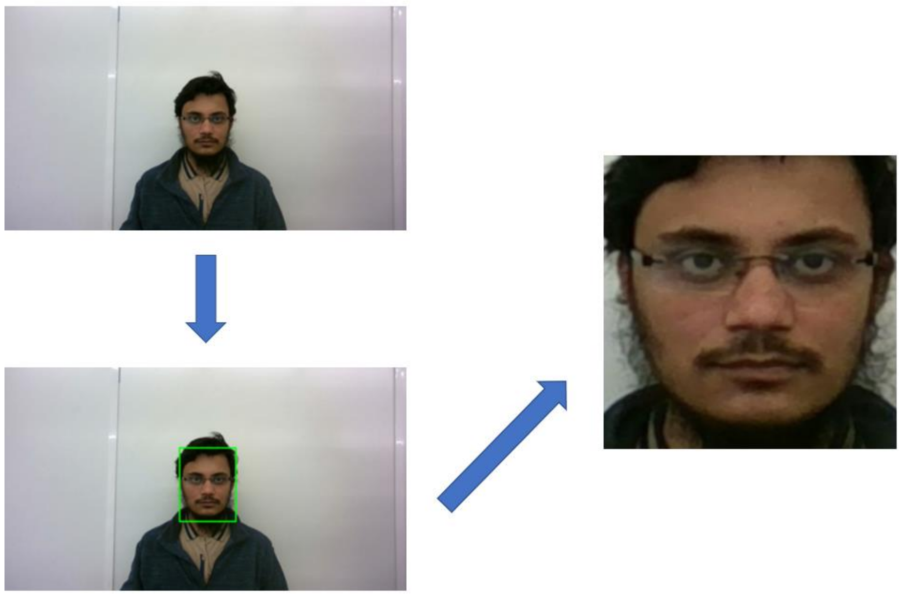


Figure 39- The three steps of the image processing of an RGB image from the RealSense dataset, regarding one subject.

As shown in Figure 39, the Haar Cascade algorithm analyses the raw RGB image of a subject to detect the presence of a face. In case of successful detection, a green rectangle is drawn like in Figure 39. Then, the algorithm crops and stores everything inside the green rectangle. This results in a close-up picture of the subject. The image is then saved in a different directory, with only cropped images. Simultaneously, the respective depth image is cropped using the coordinates that were used for the face detection, and similarly, this one also gets cropped with the same dimensions but with an addition of 25 pixels in all 4 edges of the image. This was made to obtain a slightly clearer perception of the content inside the image.

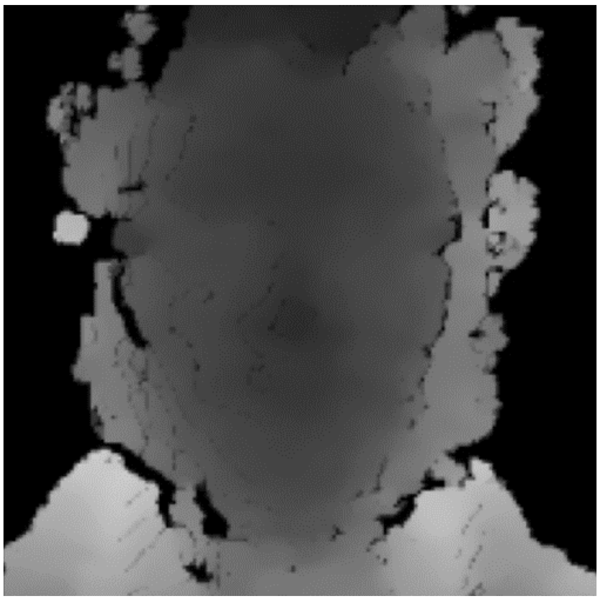


Figure 40- Processed depth image.

Lastly, a dilation operation, having an ellipse shape and a kernel size of 2 as a structuring element, was added to reduce the black lines on the subject face area that all raw depth images had like seen in Figure 32-B. All images are resized with a shape of (175, 175).

Finally, after saving in a depth images dedicated sub-directory , the resulted images would look like the one in Figure 40. The same process repeats across all the subjects' images of the dataset as well as for the ones in the validation set. The processing operation has a duration of roughly 45 seconds per subject.

4.5. Dataset Preparation

The next step consists of preparing the dataset before feeding the data to the convolutional neural networks. This task is critical to ensure the training process is carried out effectively, smoothly, and without causing an impact on memory resources.

This preparation is carried out by creating the proper image generators for the directories of images. An image data generator is a class, available in the Keras API, that allows to load the images according to a stipulated batch size by withdrawing them from a specified directory, applying optional image augmentation, and feeding them to the neural networks. By using this method, system memory is preserved considerably, since images are not being loaded all at once. This is noticeable especially in large datasets, like the one captured with the Intel RealSense. This was used in the datasets of both cameras.

One particular feature of an image generator is that it enables to point it at a certain directory, and the sub-directories inside that directory are assigned as labels for the neural network models. Figure 41 illustrates the structure of the generators used. There are four generators and each one points to each directory. Every sub-directory's name inside this directory, either RGB or depth, is considered as a label/category of the dataset, i.e., Rafael, Nuno, Tiago, etc., and each image inside these sub-directories is loaded and labelled accordingly, regardless of its name. This occurs both in the training set and validation set of either the Kinect or the RealSense datasets.

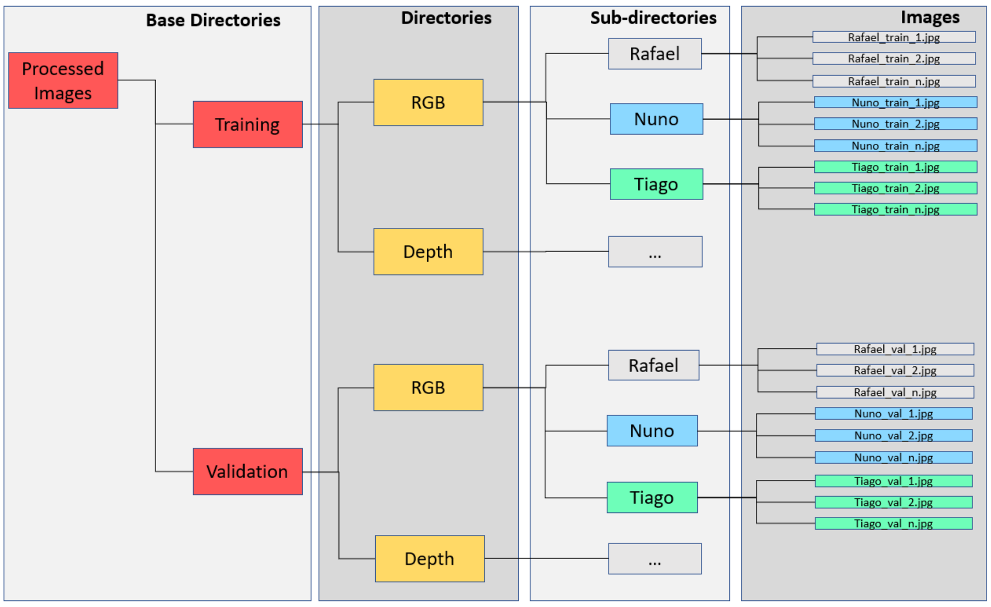


Figure 41- Structure of directories.

A common mistake that might lead to errors related to the generator is pointing it to a sub-directory instead of pointing to a directory. It must always be pointed to the directories that contain sub-directories that contain images. The names of the sub-directories are the labels for the images and the dataset.

By instantiating an object from the class ImageDataGenerator, available in Keras, parameters regarding data normalization and image augmentation can be passed as inputs. Image augmentation is discussed later in more detail.

Data normalization is a scaling technique in which dataset features values are shifted and rescaled so that they end up ranging between 0 and 1. It is also known as Min-Max scaling. The formula for normalization corresponds to (3):

$$X' = \frac{X - X_{Min}}{X_{Max} - X_{Min}} \tag{3}$$

X_{Max} and X_{Min} are the maximum and the minimum values of a feature, respectively. In the case of images, the features are its pixels. When the value of X is the minimum value in the column, the numerator is 0, and hence X' is 0. On the other hand, when the value of X is the maximum value in a certain column of a data frame, the numerator is equal to the denominator and thus the value of X' is 1.

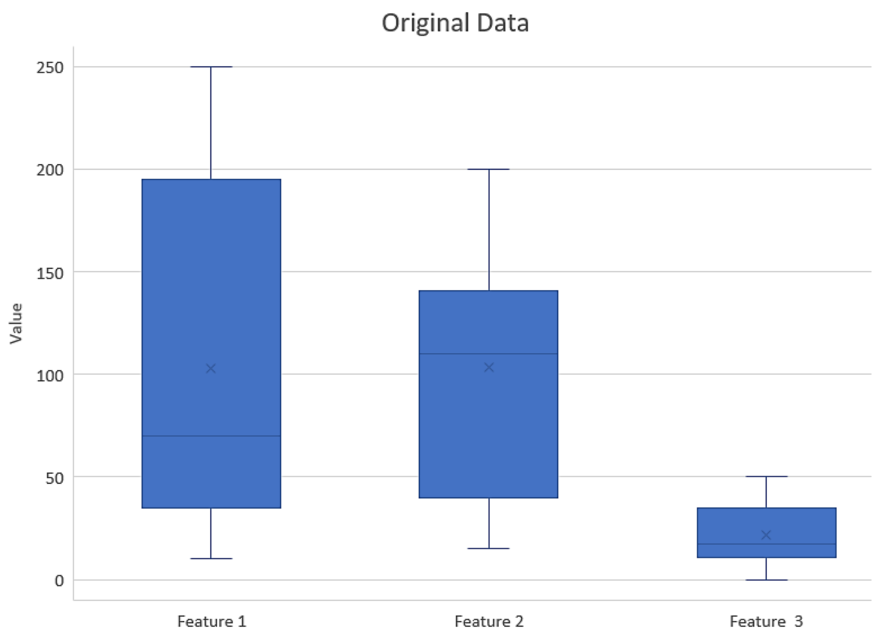


Figure 42- Box-plotted original data.

For example, consider Figure 42, where a dataset containing three features, features 1, 2, and 3, whereas feature 1 ranges between 15 to 250, feature 2 ranges from 20–200, and feature 3 ranges between 0 and 50, meaning that the features are clearly in different ranges. When further analysis is made, like multivariate linear regression, for example, the attributed feature 1 intrinsically influences the result due to its larger value. So, data has to be normalized to bring all the variables to the same scale range, like in Figure 43 below.

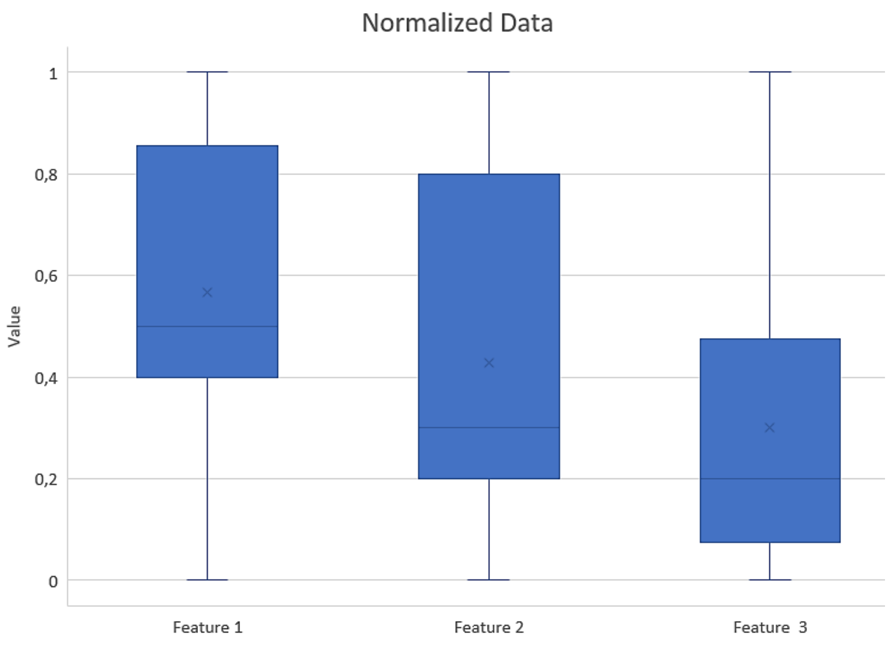


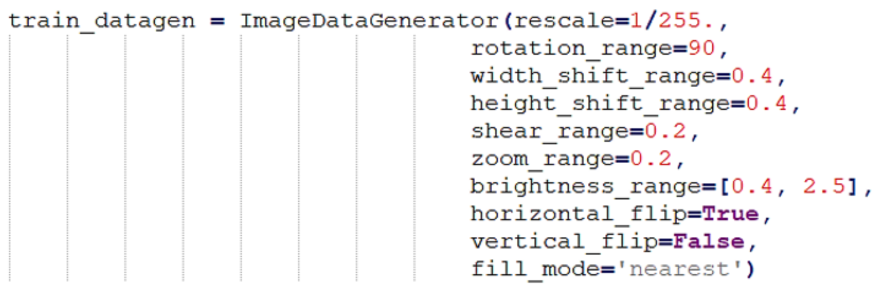
Figure 43- Box-plotted normalized data.

The data was normalized for all the generators used. Besides normalization, ImageDataGenerator class allows resizing of all images as they are loaded. By doing this at runtime, one can experiment with different sizes without impacting the source data. This is useful if a dataset does not have a uniformly sized set of images. In case of both Kinect and RealSense datasets, all the images had already been homogeneously resized in the image processing stage. To improve the efficiency of loading images, they were loaded for training and validation in batches, i.e., a given number of samples that are broadcasted through the network. Another parameter worth mentioning is the class mode. For the purpose of this dissertation, a 'categorical' class mode was used, since there were multiple labels in the dataset, except for the Kinect dataset. Since this dataset had only two labels, a 'binary' class mode was also a viable option to use.

4.5.1. Image Augmentation

When working with deep learning models, an often-encountered problem is that there is not a proper amount of data available to perform training, either because the dataset is small or the access to more data is limited. Abundant data is the key to ensure fairly good performances on model prediction. This is where the concept of image augmentation fits in.

Data augmentation is a technique to artificially create new training data from existing training data. This is carried out by applying transformations to copies of some of the samples from the training data, thus creating new and different training examples. This increases the size of the training set, allowing higher diversity. Besides expanding the size of the training set, this also allows the neural network model to generalize better on new and unseen data, making it more robust in training. These transformations include a range of image manipulations, such as shifts, horizontal and vertical flips, zooms, rotations, changes in brightness, and so on. Figure 44 portrays an implementation example of image augmentation that makes use of some transforms which can be applied to images.



```
train_datagen = ImageDataGenerator(rescale=1/255.,
                                   rotation_range=90,
                                   width_shift_range=0.4,
                                   height_shift_range=0.4,
                                   shear_range=0.2,
                                   zoom_range=0.2,
                                   brightness_range=[0.4, 2.5],
                                   horizontal_flip=True,
                                   vertical_flip=False,
                                   fill_mode='nearest')
```

Figure 44- Implementation of image augmentation inside an image generator.

However, not all transformations are suitable to perform. These depend on the data content. For example, a horizontal flip of a picture of a cat may make sense, since the photo could have been taken from the left or right. However, a vertical flip of a photo of a cat does not make much sense since the model is very unlikely to see a photo of an upside-down cat. In the following pages, some augmentation techniques are explained in more detail, some of which were used later to expand the datasets.

<u>Rotation</u>:

The ImageDataGenerator class allows to randomly rotate images through any degree between 0 and 360, by providing an integer value in the 'rotation_range' argument. When the images are rotated, some pixels might move outside the image and leave an empty area that needs to be filled in. This can be filled in different ways like a constant value or nearest pixel values. This is specified in the 'fill_mode' argument and the default value is "nearest". This simply replaces the empty area with the nearest pixel values, as seen in Figure 45.

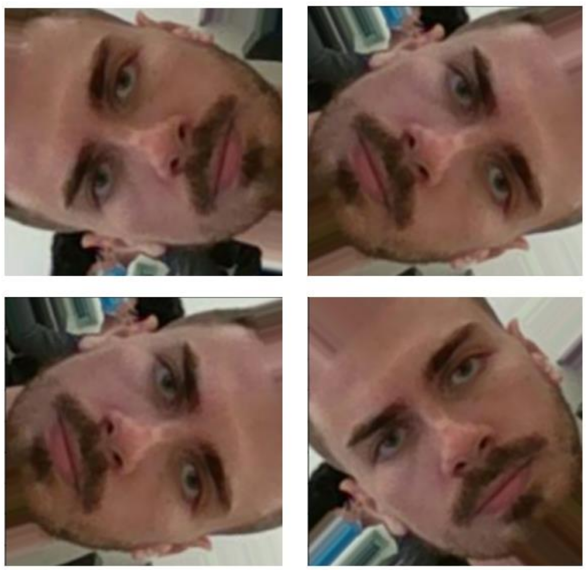


Figure 45- Random Rotation Transformations.

<u>Shift</u>:

The object or subject might be off centre and, to overcome this problem, it is possible to shift the image either horizontally or vertically by adding a certain constant value. ImageDataGenerator class has the argument 'height_shift_range' for a vertical shift of image and 'width_shift_range' for a horizontal shift. The values of these arguments must be constant. Figure 46 portrays several shifted images, some of those are width shifted, while some others are height shifted. The last one is both width and height shifted.

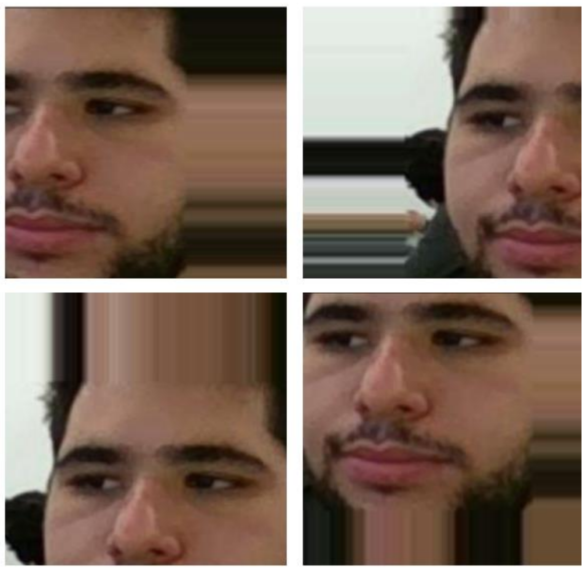


Figure 46- Random Shift Transformations.

<u>Zoom</u>:

This method randomly zooms an image in and out. ImageDataGenerator class takes in a float value or a range of values for zooming in the 'zoom_range' argument. Any value smaller than 1 zooms in , whereas any value greater than 1 zooms out.

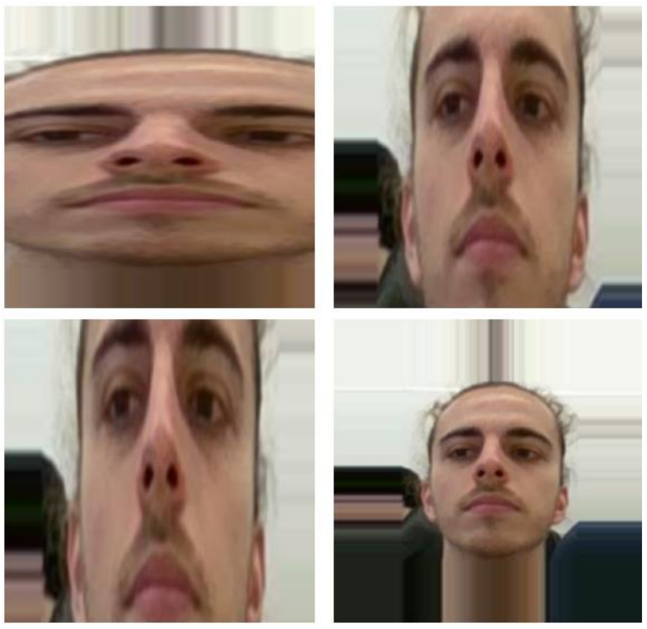


Figure 47- Random Zoom Transformations.

<u>Brightness</u>:

This transformation randomly changes the brightness of an image being a very useful augmentation technique as many times subjects do not stand in perfect lighting conditions. This permits abstraction regarding light conditions on all images. Brightness can be controlled in the ImageDataGenerator class through the 'brightness_range' argument. It accepts a list of two float values and picks a brightness shift value from that range. Values less than 1 darken the image, whereas values above 1 brighten it. See Figure 48 for an example.

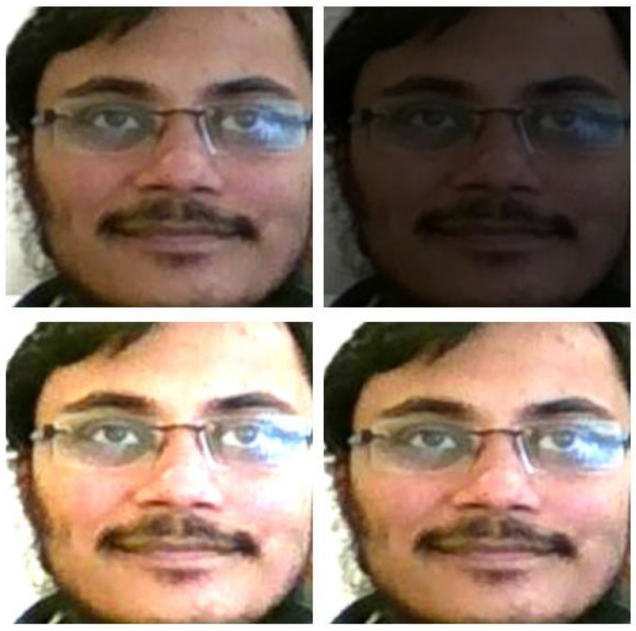


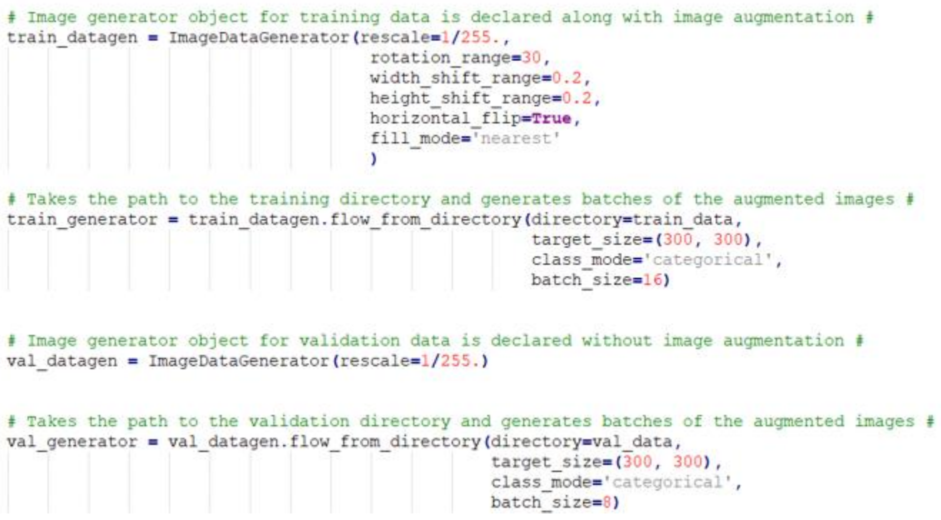
Figure 48- Random Brightness Transformations.

Besides the transformations above, there are several others, but which do not apply to this context like for example, vertical flipping. There is no point in having upside-down images of a person.

Two ImageDataGenerator classes with different arguments for image augmentation were used, one for RGB images and another for depth images. This choice was made since RGB and depth images have different characteristics, hence the need to make two distinct generators with different augmentation parameters.

Additionally, it should be emphasized that image augmentation must only be implemented in the training sets and not on the validation sets. This is due to the validation set's purpose, which is to give an estimate of the model's skill while simultaneously tuning the model's hyper parameters while the training is taking place, thus estimating the error the model has in real-world images. Hence, validation set images should be as close to reality as possible, as it would be if the camera was already mounted and running the final algorithm to recognize and validate the results.

For the Kinect dataset, the image generator objects used for the RGB images are portrayed in Figure 49, as well as the image augmentation parameters used for the images in this training set.



```
# Image generator object for training data is declared along with image augmentation #
train_datagen = ImageDataGenerator(rescale=1/255.,
                                   rotation_range=30,
                                   width_shift_range=0.2,
                                   height_shift_range=0.2,
                                   horizontal_flip=True,
                                   fill_mode='nearest'
                                   )

# Takes the path to the training directory and generates batches of the augmented images #
train_generator = train_datagen.flow_from_directory(directory=train_data,
                                                    target_size=(300, 300),
                                                    class_mode='categorical',
                                                    batch_size=16)


# Image generator object for validation data is declared without image augmentation #
val_datagen = ImageDataGenerator(rescale=1/255.)

# Takes the path to the validation directory and generates batches of the augmented images #
val_generator = val_datagen.flow_from_directory(directory=val_data,
                                                target_size=(300, 300),
                                                class_mode='categorical',
                                                batch_size=8)
```

Figure 49- Image generator with image augmentation implemented for the Kinect RGB training set.

As described in Figure 44 not all parameters were used to expand the Kinect's training set, and the validation generator does not use the image augmentation technique for the reasons stated above. One very important parameter is the 'rescale' one. This normalizes data, hence, it has to be included in all generators.

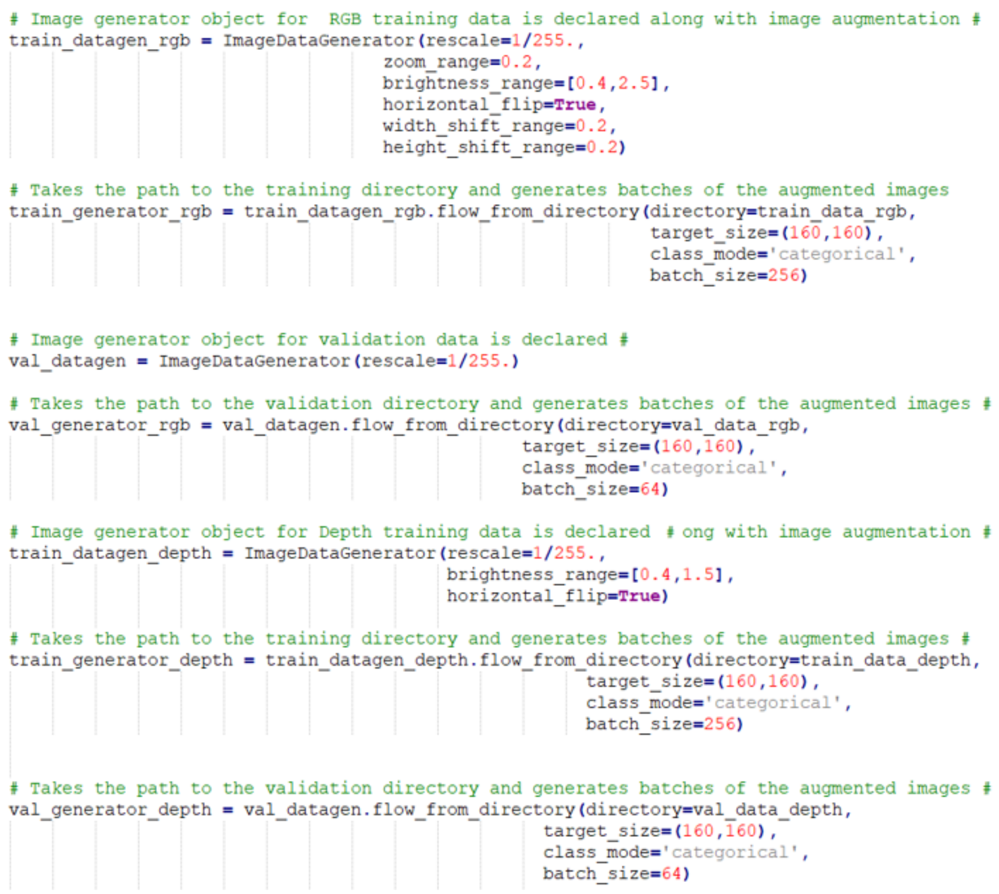
The batch size is dependent on the number of images in the directory specified and represents the number of training samples that is propagated through the network in each iteration. The batch size is comprehended between 1 and the total of training samples. If it is equal to 1, results in so-called stochastic gradient descent, i.e., relies on one image sample per iteration, meaning that the loss changes more drastically between iterations. As opposed to this behaviour, choosing a batch size equal to the total of training samples (batch gradient descent) deals with more memory usage but with smoother and more accurate results.

Another useful aspect of the image generator is that it can resize all the image samples to a specified target size. This resizing occurs in memory, as the images are being loaded by the generator. The target size chosen here was 300x300 pixels, whilst the original image size of all images from the Kinect dataset was 400x400. The parameter 'class_mode' designates the type of classes of the dataset. Since more than two classes exist, the choice was the 'categorical'. Alternatively, the choice could have been 'binary', since, in this first dataset there were two classes.

No image generator was ever created for the depth component of the Kinect training set, since, while this was still under development, the Intel RealSense was acquired, and since it was an overall better camera, all the focus of the work was shifted to this new camera.

As for the RealSense dataset, the image generator for the RGB images can be seen in Figure 50 below.

In this case a more complex set of augmentation parameters was used, since this CNN was more complex and capable of handling more data. The final batch size used was the one portrayed in Figure 50 and it provided the best overall results.



```
# Image generator object for  RGB training data is declared along with image augmentation #
train_datagen_rgb = ImageDataGenerator(rescale=1/255.,
                                   zoom_range=0.2,
                                   brightness_range=[0.4,2.5],
                                   horizontal_flip=True,
                                   width_shift_range=0.2,
                                   height_shift_range=0.2)

# Takes the path to the training directory and generates batches of the augmented images
train_generator_rgb = train_datagen_rgb.flow_from_directory(directory=train_data_rgb,
                                                         target_size=(160,160),
                                                         class_mode='categorical',
                                                         batch_size=256)


# Image generator object for validation data is declared #
val_datagen = ImageDataGenerator(rescale=1/255.)

# Takes the path to the validation directory and generates batches of the augmented images #
val_generator_rgb = val_datagen.flow_from_directory(directory=val_data_rgb,
                                                   target_size=(160,160),
                                                   class_mode='categorical',
                                                   batch_size=64)

# Image generator object for Depth training data is declared  # ong with image augmentation #
train_datagen_depth = ImageDataGenerator(rescale=1/255.,
                                         brightness_range=[0.4,1.5],
                                         horizontal_flip=True)

# Takes the path to the training directory and generates batches of the augmented images #
train_generator_depth = train_datagen_depth.flow_from_directory(directory=train_data_depth,
                                                               target_size=(160,160),
                                                               class_mode='categorical',
                                                               batch_size=256)


# Takes the path to the validation directory and generates batches of the augmented images #
val_generator_depth = val_datagen.flow_from_directory(directory=val_data_depth,
                                                     target_size=(160,160),
                                                     class_mode='categorical',
                                                     batch_size=64)
```

Figure 50- Image generator with image augmentation implementation for the RealSense dataset.

4.6. Training

After applying image augmentation to the training set and creating all image generators, the images are ready to enter the neural networks for training. But first, it is necessary to build the neural networks as well as adjust its parameters.

Since different approaches were implemented accordingly to the type of camera used, different neural networks had to be built for each camera, because the first neural networks that were used to train with the Kinect dataset proved to be useless in Intel RealSense's dataset. This can be explained by the fact that the Kinect dataset had only two classes, while the Intel RealSense had six, hence, more data to train. This could mean the neural network was not able to deal with added data. To undertake this, the neural networks' hyper parameters had to be adjusted. If these adjustments did not work at all, the network had to be entirely replaced with a new one. The new one could either be built from scratch or through the use of transfer learning.

The initial training for the Kinect dataset was carried out using Jupyter Notebook since the image processing was also done previously using this application. An inconvenience in using Jupyter Notebook is that the training is performed locally using the system's CPU. Training neural networks using CPUs is computationally expensive since CPUs only perform sequential calculations. This meant the training process is very slow, making it worse when more image samples are added, image augmentation is used, image size increases, or more layers are added to the network. Given the ineffectiveness of this approach, training should be done by using one or multiple GPUs, since this tactic allows parallel calculations, thus speeding the learning process exponentially. Since the local system did not have a proper GPU card, it was decided to use Google Colab. Regarding the RealSense camera's images, these were always trained on neural networks using the GPUs from the allocated server in Google Colab.

The first CNN implemented was the one in Figure 51. The architecture was entirely built from scratch, using as inputs the RGB images from the Kinect, in batches, coming from the image generator made previously.

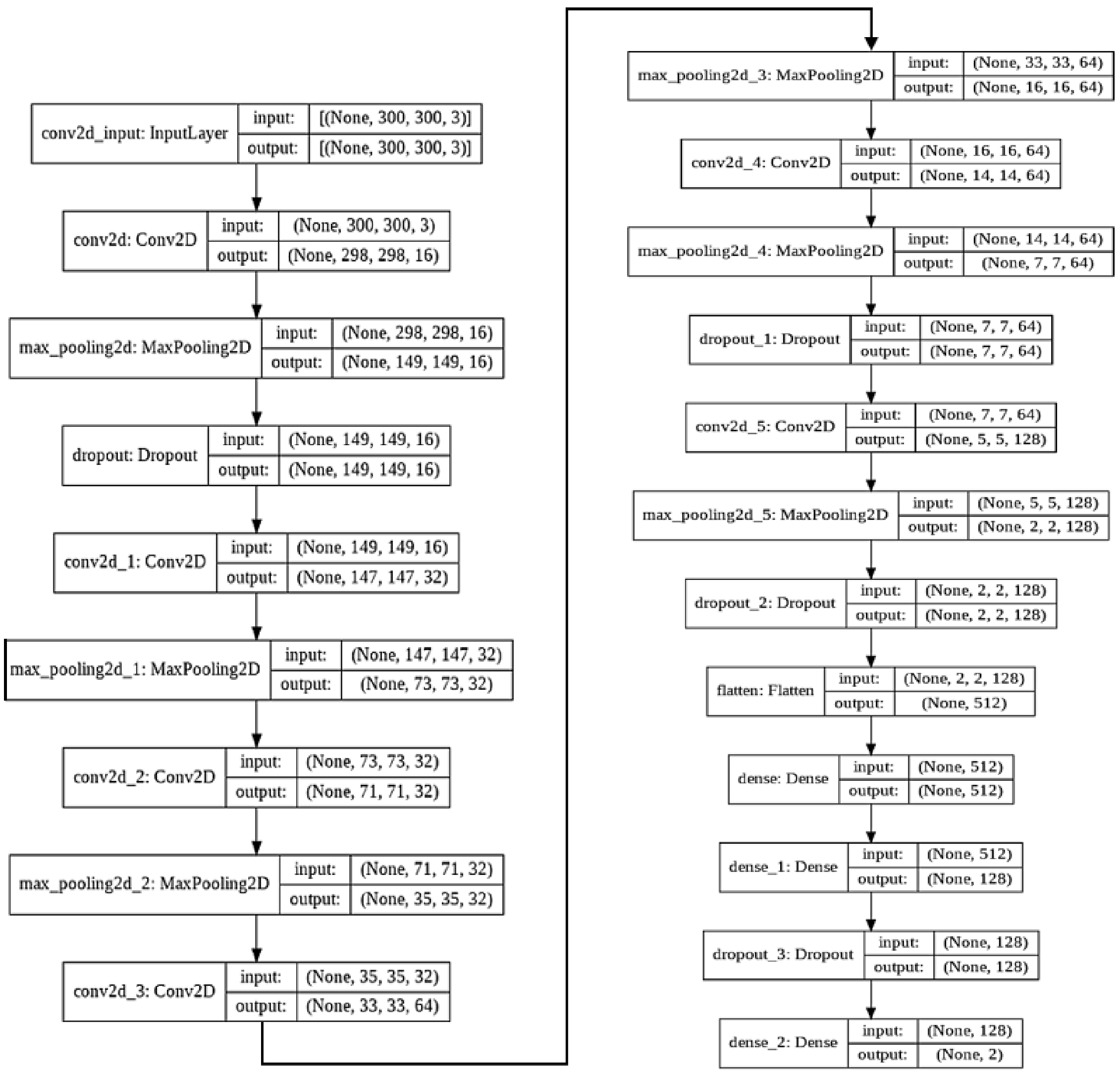

conv2d_input: InputLayer | input: [(None, 300, 300, 3)] | output: [(None, 300, 300, 3)]
conv2d: Conv2D | input: (None, 300, 300, 3) | output: (None, 298, 298, 16)
max_pooling2d: MaxPooling2D | input: (None, 298, 298, 16) | output: (None, 149, 149, 16)
dropout: Dropout | input: (None, 149, 149, 16) | output: (None, 149, 149, 16)
conv2d_1: Conv2D | input: (None, 149, 149, 16) | output: (None, 147, 147, 32)
max_pooling2d_1: MaxPooling2D | input: (None, 147, 147, 32) | output: (None, 73, 73, 32)
conv2d_2: Conv2D | input: (None, 73, 73, 32) | output: (None, 71, 71, 32)
max_pooling2d_2: MaxPooling2D | input: (None, 71, 71, 32) | output: (None, 35, 35, 32)
conv2d_3: Conv2D | input: (None, 35, 35, 32) | output: (None, 33, 33, 64)
max_pooling2d_3: MaxPooling2D | input: (None, 33, 33, 64) | output: (None, 16, 16, 64)
conv2d_4: Conv2D | input: (None, 16, 16, 64) | output: (None, 14, 14, 64)
max_pooling2d_4: MaxPooling2D | input: (None, 14, 14, 64) | output: (None, 7, 7, 64)
dropout_1: Dropout | input: (None, 7, 7, 64) | output: (None, 7, 7, 64)
conv2d_5: Conv2D | input: (None, 7, 7, 64) | output: (None, 5, 5, 128)
max_pooling2d_5: MaxPooling2D | input: (None, 5, 5, 128) | output: (None, 2, 2, 128)
dropout_2: Dropout | input: (None, 2, 2, 128) | output: (None, 2, 2, 128)
flatten: Flatten | input: (None, 2, 2, 128) | output: (None, 512)
dense: Dense | input: (None, 512) | output: (None, 512)
dense_1: Dense | input: (None, 512) | output: (None, 128)
dropout_3: Dropout | input: (None, 128) | output: (None, 128)
dense_2: Dense | input: (None, 128) | output: (None, 2)

Figure 51- Architecture 1 of the RGB model used for the Kinect dataset.

This model was very simple and straightforward. The input images have the size (300,300,3) and are input into the network. Then they get through convolutional and pooling layers as the images undergo consecutive convolutions. As a consequence of the convolution process, the images get smaller as explained in 2.3.1. On the last layer before the Flatten one, the size of the images is (2,2,128). I is virtually impossible to add more convolutional layers since, if that was the case, there would not be enough pixels to perform convolution. Fully connected layers are added since they allow the network to learn non-linear combinations of the features obtained with the convolutional layers. It is important to note that the number of neurons of the output of the last layer is always equal to the number of classes, which in this case was 2. Several Dropout layers were placed throughout the network. This layer aims to reduce the complexity of the model by helping to prevent it from overfitting. The way it works is by randomly deactivating neurons in a layer according to a given probability *p* from a Bernoulli distribution. Since only two labels exist in this dataset, the final layer has only 2 neurons and a ‘softmax’ activation function. Alternatively, it was possible to specify only one neuron at the output, meaning in this case that a ‘sigmoid’ activation function has to be stated. The option to either choose one of these cases is exclusively possible when the dataset has only 2 labels. Note that a softmax function outputs any value between 0 and 1 for each sample it receives. On the other hand, a sigmoid function outputs either 0 or 1.

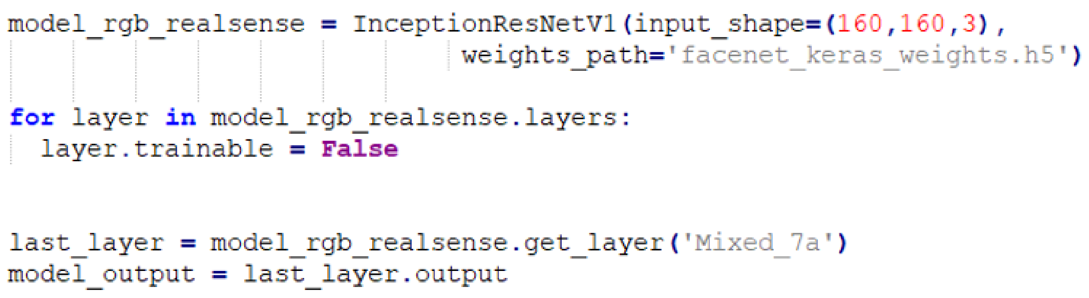
A few optimizers were used with this model, being the final optimizer the one that obtained the best overall results, the Adam optimizer. The purpose is to update the network’s weights according to the loss function response, and in turn, minimize it. Other optimizers such as SGD and RMSprop were also tested. The loss function selected was ‘categorical_crossentropy’ since there were 2 or more classes. Alternatively, it was possible to use ‘binary_crossentropy’ as long as previously one gave only 1 neuron for the last fully-connected layer, since a binary loss function means that the output is either 0 or 1, which in turn corresponds to the neuron either being 0 or 1/deactivated or activated. As for the learning rate selected, it was chosen upon a trial-and-error approach, being 1×10^{-4} the value with the best outcome. The results of selecting each one the optimizers are described in chapter Tests and Results.

4.6.1 Transfer Learning

Besides the neural network used above for the Kinect dataset made from scratch, all the other neural networks used were developed using transfer learning. Through this method, the best possible results were achieved in less training time, i.e., in fewer epochs.

Transfer learning is a method that allows to reuse an already trained model (pre-trained model) and consequently, its respective weights and learned features, and apply it to a new situation or problem. It is a very popular technique in the DL community since it allows to train any sort of DNN in a short period and with a lower amount of data. For example, if someone has trained a CNN model to classify breeds of dogs, another person could benefit from this training model by applying its weights to a CNN model to classify breeds of cats, since the first CNN had already learned the median-level features and, most importantly, the low-level ones, like lines or shapes, which are common in these two animals.

Similarly, for the RGB component of the face recognition problem, one used a pre-trained model that was previously trained using a dataset containing faces of people, extracted the resulting weights of that model, and applied them in another model with the same architecture. This pre-trained model was implemented based on Facenet [38], using 200 million face images from 8 million people. It is a 2D face recognition algorithm trained using an Inception-Resnet V1 architecture (see section 2.4.3). For this, it is required to declare an object of the class InceptionResnetV1, which builds the whole architecture. Then, it is essential to load the model with the weights file developed when training with the 200 million faces dataset. Then, the input shape of the images is specified, which for this case was (160,160,3), and then it is required to “lock” all the layers from the imported model, since there is no intention to train these layers anymore as they are already trained. To manage this, it is required to iterate through each of the layers and signal them to false, as shown in Figure 52.



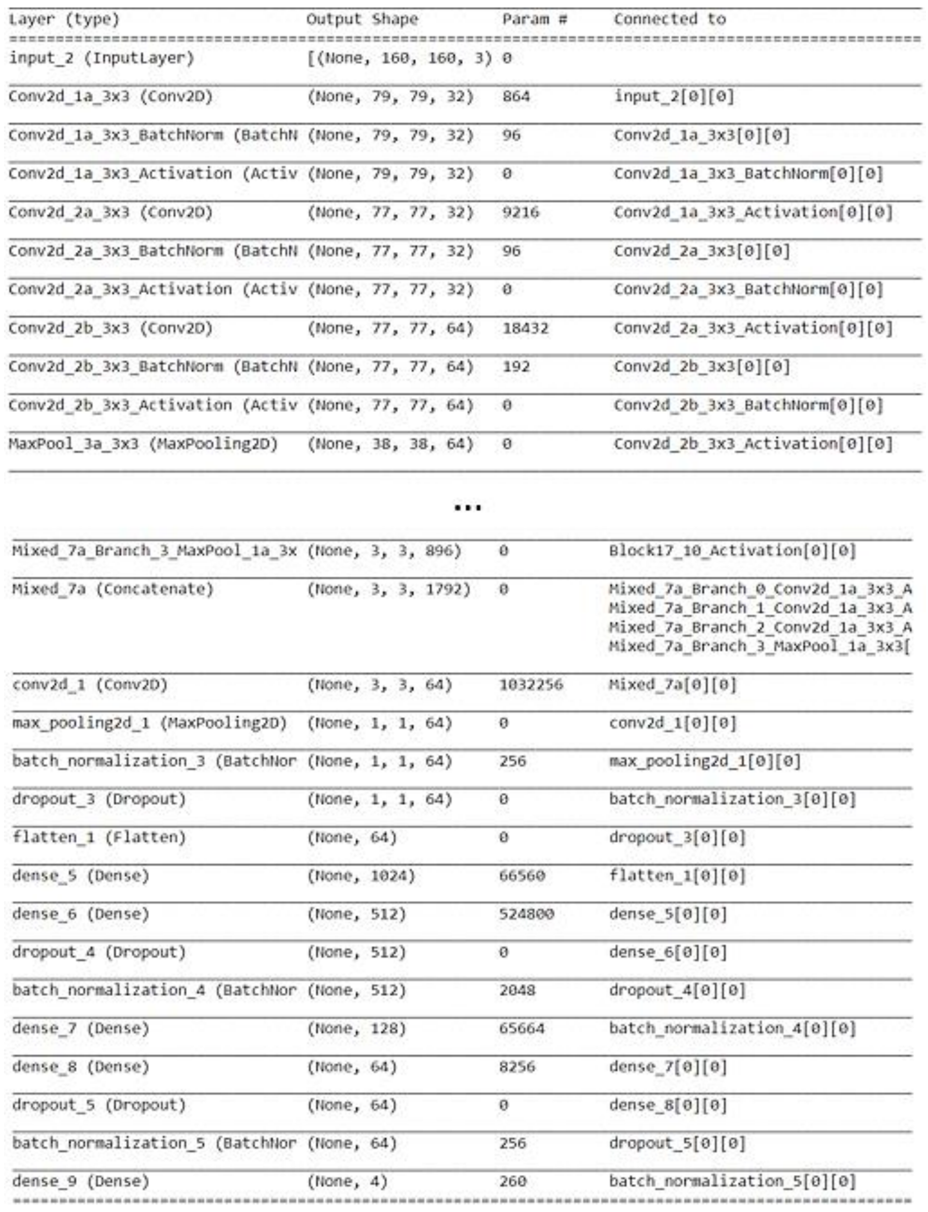
```
model_rgb_realsense = InceptionResNetV1(input_shape=(160,160,3),
                                        weights_path='facenet_keras_weights.h5')

for layer in model_rgb_realsense.layers:
    layer.trainable = False


last_layer = model_rgb_realsense.get_layer('Mixed_7a')
model_output = last_layer.output
```

Figure 52- Object Inception-Resnet V1, with weight loading.

As stated, the first step is to import the model’s architecture and load it with the pre-trained weights.. After signalling the layers to false, the ‘Mixed_7a’ layer was extracted, which means that all the layers above were discarded, i.e., the architecture starts in its input and extends all the way to ‘Mixed_7a’. This layer becomes the model output before adding more layers.

In order to extract more features from the training sets made, extra convolutional and pooling layers were added before flattening and introducing fully connected layers, i.e., the top of the model. This model has 356 layers deep. Figure 53 illustrates the first and last layers of the RealSense RGB model.

Similar to the Kinect RGB model, the last fully-connected layer of the model has several neurons that correspond to the number of classes/categories of the dataset. Regarding the optimizer, the one that obtained the best results was the Adam one, although others were also used to compare results. The optimizer’s learning rate was chosen upon a trial-and-error approach, being 4×10^{-5} the value that provided the best overall outcome. Since this dataset had 6 classes, ‘categorical_crossentropy’ was assigned as loss function and the activation function selected was ‘softmax’.



```
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_2 (InputLayer)            [(None, 160, 160, 3) 0
__________________________________________________________________________________________________
Conv2d_1a_3x3 (Conv2D)          (None, 79, 79, 32)   864         input_2[0][0]
__________________________________________________________________________________________________
Conv2d_1a_3x3_BatchNorm (BatchN (None, 79, 79, 32)   96          Conv2d_1a_3x3[0][0]
__________________________________________________________________________________________________
Conv2d_1a_3x3_Activation (Activ (None, 79, 79, 32)   0           Conv2d_1a_3x3_BatchNorm[0][0]
__________________________________________________________________________________________________
Conv2d_2a_3x3 (Conv2D)          (None, 77, 77, 32)   9216        Conv2d_1a_3x3_Activation[0][0]
__________________________________________________________________________________________________
Conv2d_2a_3x3_BatchNorm (BatchN (None, 77, 77, 32)   96          Conv2d_2a_3x3[0][0]
__________________________________________________________________________________________________
Conv2d_2a_3x3_Activation (Activ (None, 77, 77, 32)   0           Conv2d_2a_3x3_BatchNorm[0][0]
__________________________________________________________________________________________________
Conv2d_2b_3x3 (Conv2D)          (None, 77, 77, 64)   18432       Conv2d_2a_3x3_Activation[0][0]
__________________________________________________________________________________________________
Conv2d_2b_3x3_BatchNorm (BatchN (None, 77, 77, 64)   192         Conv2d_2b_3x3[0][0]
__________________________________________________________________________________________________
Conv2d_2b_3x3_Activation (Activ (None, 77, 77, 64)   0           Conv2d_2b_3x3_BatchNorm[0][0]
__________________________________________________________________________________________________
MaxPool_3a_3x3 (MaxPooling2D)   (None, 38, 38, 64)   0           Conv2d_2b_3x3_Activation[0][0]
__________________________________________________________________________________________________

                                          ...

Mixed_7a_Branch_3_MaxPool_1a_3x (None, 3, 3, 896)    0           Block17_10_Activation[0][0]
__________________________________________________________________________________________________
Mixed_7a (Concatenate)          (None, 3, 3, 1792)   0           Mixed_7a_Branch_0_Conv2d_1a_3x3_A
                                                                 Mixed_7a_Branch_1_Conv2d_1a_3x3_A
                                                                 Mixed_7a_Branch_2_Conv2d_1a_3x3_A
                                                                 Mixed_7a_Branch_3_MaxPool_1a_3x3[
__________________________________________________________________________________________________
conv2d_1 (Conv2D)               (None, 3, 3, 64)     1032256     Mixed_7a[0][0]
__________________________________________________________________________________________________
max_pooling2d_1 (MaxPooling2D)  (None, 1, 1, 64)     0           conv2d_1[0][0]
__________________________________________________________________________________________________
batch_normalization_3 (BatchNor (None, 1, 1, 64)     256         max_pooling2d_1[0][0]
__________________________________________________________________________________________________
dropout_3 (Dropout)             (None, 1, 1, 64)     0           batch_normalization_3[0][0]
__________________________________________________________________________________________________
flatten_1 (Flatten)             (None, 64)           0           dropout_3[0][0]
__________________________________________________________________________________________________
dense_5 (Dense)                 (None, 1024)         66560       flatten_1[0][0]
__________________________________________________________________________________________________
dense_6 (Dense)                 (None, 512)          524800      dense_5[0][0]
__________________________________________________________________________________________________
dropout_4 (Dropout)             (None, 512)          0           dense_6[0][0]
__________________________________________________________________________________________________
batch_normalization_4 (BatchNor (None, 512)          2048        dropout_4[0][0]
__________________________________________________________________________________________________
dense_7 (Dense)                 (None, 128)          65664       batch_normalization_4[0][0]
__________________________________________________________________________________________________
dense_8 (Dense)                 (None, 64)           8256        dense_7[0][0]
__________________________________________________________________________________________________
dropout_5 (Dropout)             (None, 64)           0           dense_8[0][0]
__________________________________________________________________________________________________
batch_normalization_5 (BatchNor (None, 64)           256         dropout_5[0][0]
__________________________________________________________________________________________________
dense_9 (Dense)                 (None, 4)            260         batch_normalization_5[0][0]
==================================================================================================
```

Figure 53- First layers and last layers of the RealSense RGB model and respective inputs.

An Early Stopping function to halt the training was added, with the purpose of automatically halt the training once the model's performance stops improving on the validation set. A problem with training CNNs is the choice of the best suitable number for training epochs. Too many epochs can lead to overfitting problems, whereas too few of them can result in an underfit model. Using an Early Stopping provides a balance in the proper number of epochs and avoids overfitting. Usually, only two parameters are specified to the Early Stopping function: the monitor value and the patience value. The first one relates to which metric should be supervised. Usually, this metric is the validation loss since according to the DL community, it provides a better estimate of the real-world performance of a model. The patience value relates to the number of epochs with no improvement after which training must stop. For example, supposing that patience is equal to 15. If at a certain epoch *n* the validation loss reaches a certain value and, after *n*+15 epochs that same validation loss does not decrease, the training process stops, whereas if it reaches a new minimum value, the training continues until it cannot go below that same value in the next 15 epochs.

Lastly, the 'fit' function needs to be called to start the training. This function takes as input, the training and validation generators, number of epochs, early stop functions declared, steps per epoch (the number of batch iterations before a training epoch is considered finished), and validation steps, which is the same as steps per epoch but applied to the validation set. To calculate the steps per epoch parameter, the following equation is used:

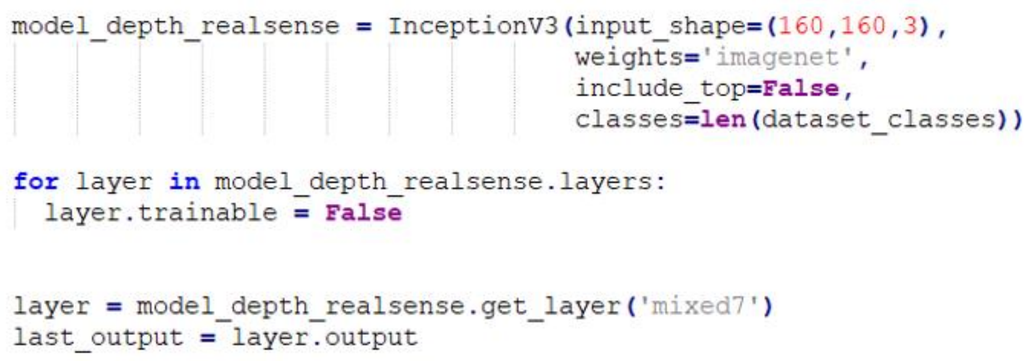
$$ST = floor(ceil(training\_samples)\ /\ training\_sample\_size) \tag{4}$$

Similarly to the equation above, to calculate the same parameter but applied to the validation set, the following formula must be used instead:

$$SV = floor(ceil(validation\_samples)\ /\ validation\_sample\_size) \tag{5}$$

The ceil function returns the smallest possible integer value which is greater than or equal to the given argument, whilst the floor function returns the largest integer less than or equal to the given argument.

Regarding the RealSense dataset depth component, transfer learning was also used to train the model with depth images, since the results obtained with the training of the RGB model turned out to be quite promising. However, for depth images a different architecture was implemented. An Inception V3 network was used (see section 2.4.2), with the built-in weights pre-trained on the ImageNet dataset. ImageNet is a very large dataset with a collection of human-annotated images designed by academics for developing computer vision algorithms. It contains thousands of classes and is trained using hundreds of thousands of images. To import this architecture, the same principle used above for the RGB model was used (see Figure 54 for illustration purposes).

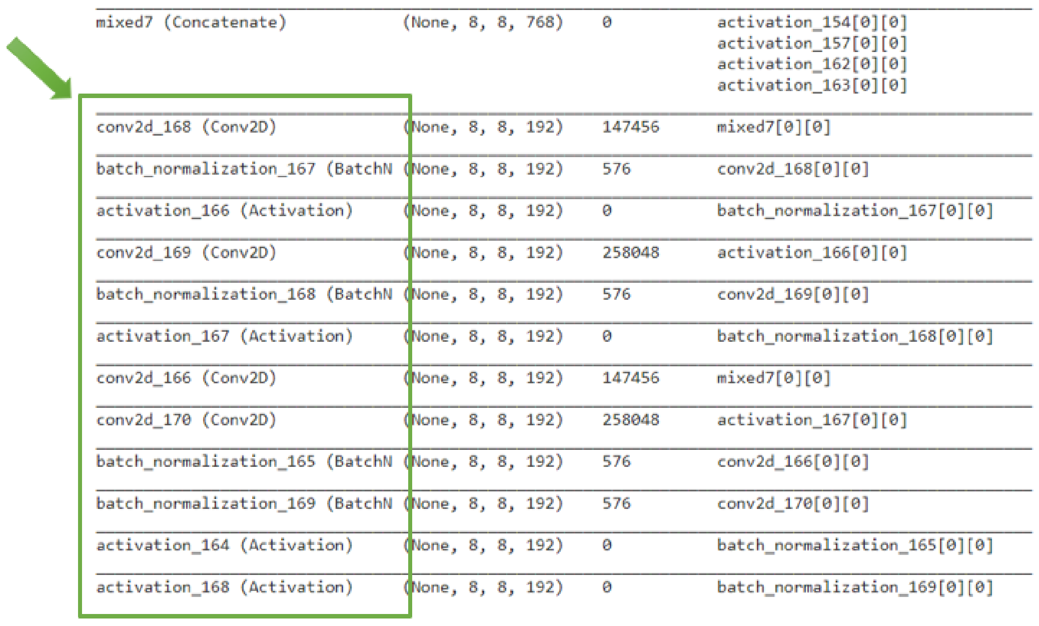


```
model_depth_realsense = InceptionV3(input_shape=(160,160,3),
                                   weights='imagenet',
                                   include_top=False,
                                   classes=len(dataset_classes))

for layer in model_depth_realsense.layers:
  layer.trainable = False


layer = model_depth_realsense.get_layer('mixed7')
last_output = layer.output
```

Figure 54- Object Inception V3.

This time around, one does not specify the directory of the weights because the intention is to use the ImageNet dataset. Instead, it is necessary to specify the 'imagenet' command in the 'weights' parameter. The 'include_top' boolean variable is also set to 'False' since one does not want to include the top fully connected layers of the pre-trained model. Lastly, the total number of classes in the dataset is specified.

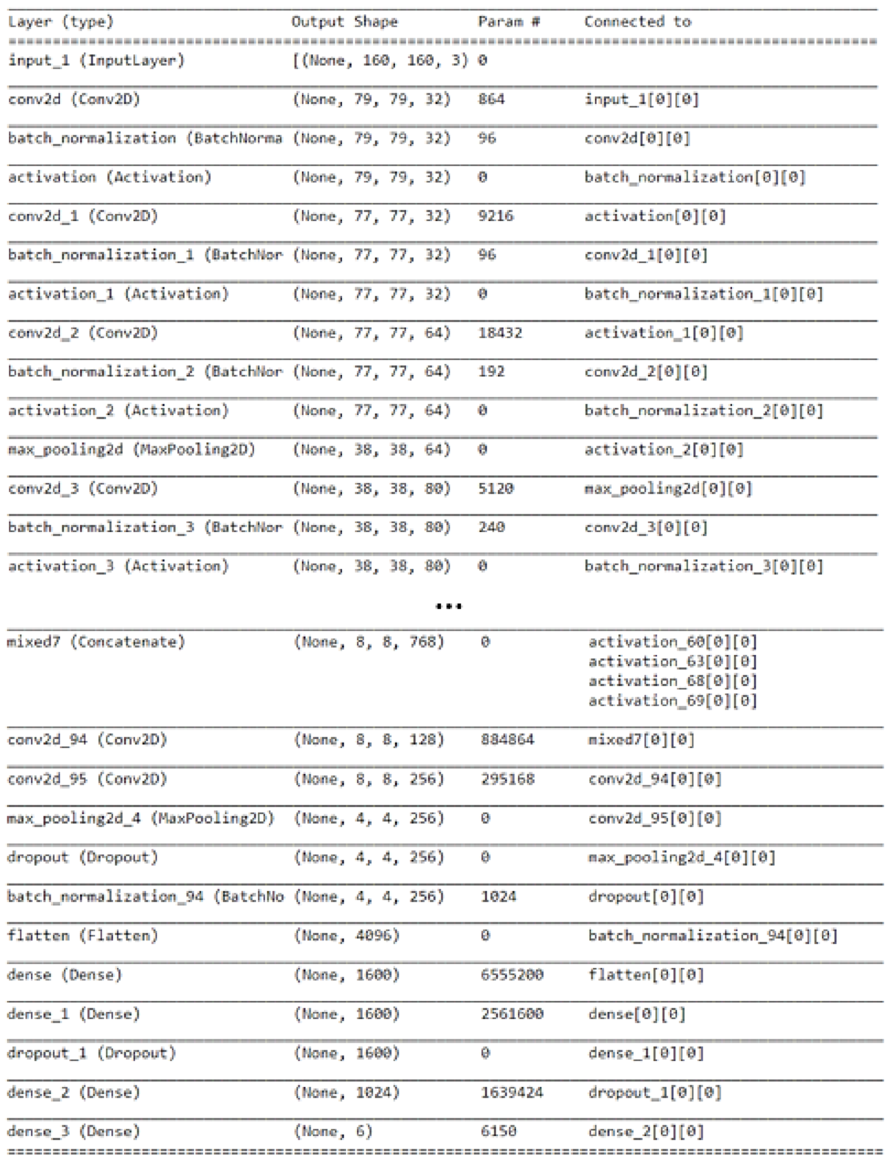
Analogously to the RGB model, it is desirable to prevent the existing layers from being trained again, since that would compromise the weights already stored for this model. Then, the 'mixed7' layer becomes the new output of the model before adding more layers. This means that all the layers above 'mixed7', like convolution ones, activation, and even 'mixed8' and 'mixed9' are discarded. Some of these layers can be seen highlighted in Figure 55.



```
mixed7 (Concatenate)            (None, 8, 8, 768)    0           activation_154[0][0]
                                                                 activation_157[0][0]
                                                                 activation_162[0][0]
                                                                 activation_163[0][0]

conv2d_168 (Conv2D)             (None, 8, 8, 192)    147456      mixed7[0][0]
batch_normalization_167 (BatchN (None, 8, 8, 192)    576         conv2d_168[0][0]
activation_166 (Activation)     (None, 8, 8, 192)    0           batch_normalization_167[0][0]
conv2d_169 (Conv2D)             (None, 8, 8, 192)    258048      activation_166[0][0]
batch_normalization_168 (BatchN (None, 8, 8, 192)    576         conv2d_169[0][0]
activation_167 (Activation)     (None, 8, 8, 192)    0           batch_normalization_168[0][0]
conv2d_166 (Conv2D)             (None, 8, 8, 192)    147456      mixed7[0][0]
conv2d_170 (Conv2D)             (None, 8, 8, 192)    258048      activation_167[0][0]
batch_normalization_165 (BatchN (None, 8, 8, 192)    576         conv2d_166[0][0]
batch_normalization_169 (BatchN (None, 8, 8, 192)    576         conv2d_170[0][0]
activation_164 (Activation)     (None, 8, 8, 192)    0           batch_normalization_165[0][0]
activation_168 (Activation)     (None, 8, 8, 192)    0           batch_normalization_169[0][0]
```

Figure 55- Discarded layers from the Inception V3 network.

Like in the previous architecture, two more convolution layers were added as well as a pooling layer to introduce the depth images into the pre-trained network and extract features within these images. Then, the network was flattened and fully-connected layers were added. The last fully-connected layer has the same number of neurons as the number of classes in the dataset. Other parameters are very similar to the previous model, like the optimizer, the loss function, and the Earlystop function inside the fit function. The parameter that differs here from the previous model is the learning rate, which for this model was set to 6×10^{-5}. This learning rate established the most favourable overall results, as can be seen in section 5.



```
______________________________________________________________________________________________
Layer (type)                   Output Shape         Param #     Connected to
==============================================================================================
input_1 (InputLayer)           [(None, 160, 160, 3) 0

______________________________________________________________________________________________
conv2d (Conv2D)                (None, 79, 79, 32)   864         input_1[0][0]
______________________________________________________________________________________________
batch_normalization (BatchNorma (None, 79, 79, 32)  96          conv2d[0][0]
______________________________________________________________________________________________
activation (Activation)        (None, 79, 79, 32)   0           batch_normalization[0][0]
______________________________________________________________________________________________
conv2d_1 (Conv2D)              (None, 77, 77, 32)   9216        activation[0][0]
______________________________________________________________________________________________
batch_normalization_1 (BatchNor (None, 77, 77, 32)  96          conv2d_1[0][0]
______________________________________________________________________________________________
activation_1 (Activation)      (None, 77, 77, 32)   0           batch_normalization_1[0][0]
______________________________________________________________________________________________
conv2d_2 (Conv2D)              (None, 77, 77, 64)   18432       activation_1[0][0]
______________________________________________________________________________________________
batch_normalization_2 (BatchNor (None, 77, 77, 64)  192         conv2d_2[0][0]
______________________________________________________________________________________________
activation_2 (Activation)      (None, 77, 77, 64)   0           batch_normalization_2[0][0]
______________________________________________________________________________________________
max_pooling2d (MaxPooling2D)   (None, 38, 38, 64)   0           activation_2[0][0]
______________________________________________________________________________________________
conv2d_3 (Conv2D)              (None, 38, 38, 80)   5120        max_pooling2d[0][0]
______________________________________________________________________________________________
batch_normalization_3 (BatchNor (None, 38, 38, 80)  240         conv2d_3[0][0]
______________________________________________________________________________________________
activation_3 (Activation)      (None, 38, 38, 80)   0           batch_normalization_3[0][0]

                                              ...
______________________________________________________________________________________________
mixed7 (Concatenate)           (None, 8, 8, 768)    0           activation_60[0][0]
                                                                activation_63[0][0]
                                                                activation_68[0][0]
                                                                activation_69[0][0]
______________________________________________________________________________________________
conv2d_94 (Conv2D)             (None, 8, 8, 128)    884864      mixed7[0][0]
______________________________________________________________________________________________
conv2d_95 (Conv2D)             (None, 8, 8, 256)    295168      conv2d_94[0][0]
______________________________________________________________________________________________
max_pooling2d_4 (MaxPooling2D) (None, 4, 4, 256)    0           conv2d_95[0][0]
______________________________________________________________________________________________
dropout (Dropout)              (None, 4, 4, 256)    0           max_pooling2d_4[0][0]
______________________________________________________________________________________________
batch_normalization_94 (BatchNo (None, 4, 4, 256)   1024        dropout[0][0]
______________________________________________________________________________________________
flatten (Flatten)              (None, 4096)         0           batch_normalization_94[0][0]
______________________________________________________________________________________________
dense (Dense)                  (None, 1600)         6555200     flatten[0][0]
______________________________________________________________________________________________
dense_1 (Dense)                (None, 1600)         2561600     dense[0][0]
______________________________________________________________________________________________
dropout_1 (Dropout)            (None, 1600)         0           dense_1[0][0]
______________________________________________________________________________________________
dense_2 (Dense)                (None, 1024)         1639424     dropout_1[0][0]
______________________________________________________________________________________________
dense_3 (Dense)                (None, 6)            6150        dense_2[0][0]
==============================================================================================
```

Figure 56- First layers and last layers of the RealSense depth model and respective inputs.

Besides training both models independently, an approach to train them together took place. To do this both outputs of the models were concatenated. Model concatenation is very useful when two or more distinct datasets exist and different data features to train and has two or more neural network models. In theory, the two models portrayed above could perfectly be concatenated since they have different data.

A two-layer concatenation is described in Figure 57. RGB and depth model's layers converge into a single concatenation layer, and then the resulting model gets flattened before adding fully-connected layers. This means that it is possible to have 2 or more different inputs to form one single architecture. The last remaining steps are to compile the model using an optimizer with a selected learning rate and loss function and to fit the concatenated model. For this, the junction of the two generators is used.

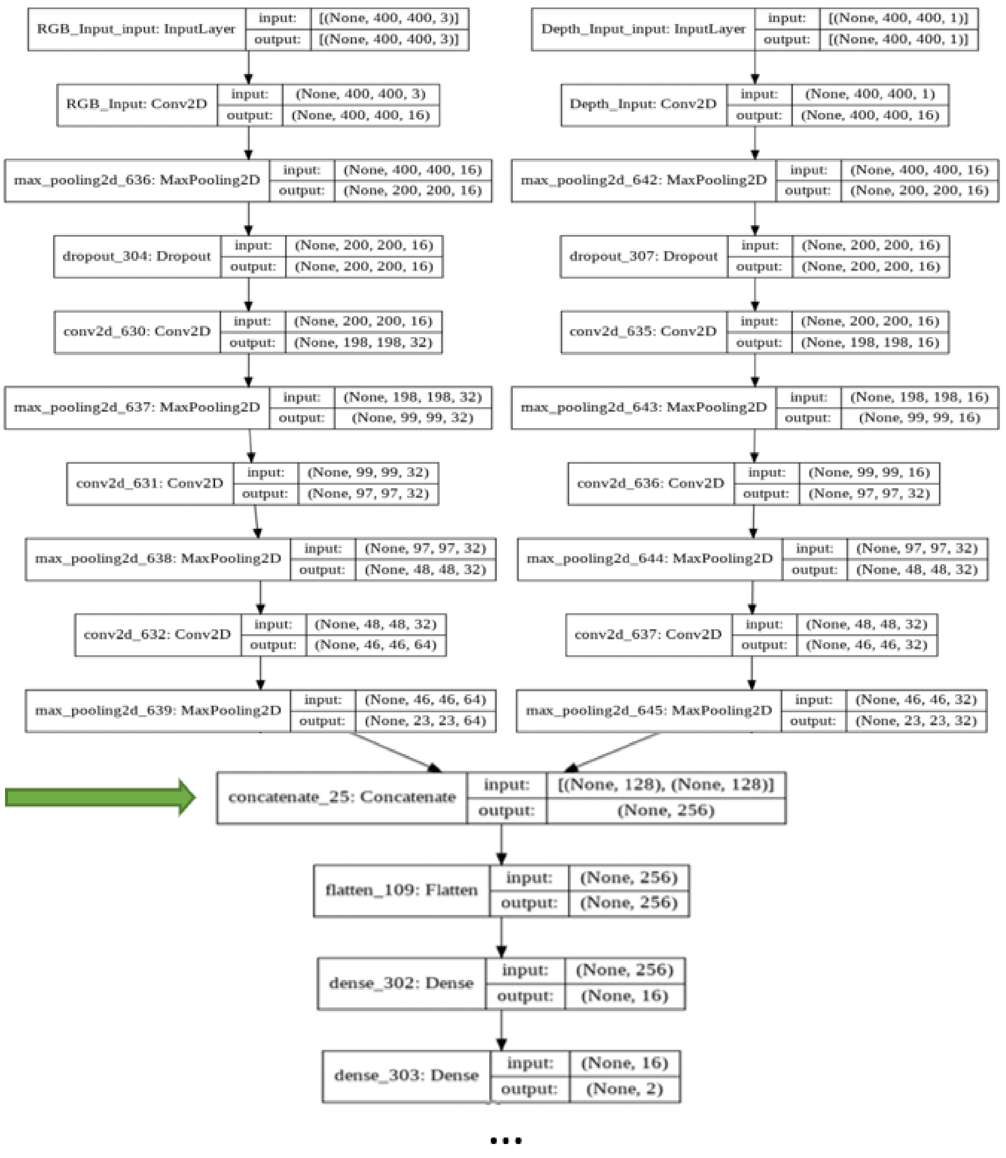

RGB_Input_input: InputLayer | input: [(None, 400, 400, 3)] | output: [(None, 400, 400, 3)]
Depth_Input_input: InputLayer | input: [(None, 400, 400, 1)] | output: [(None, 400, 400, 1)]
RGB_Input: Conv2D | input: (None, 400, 400, 3) | output: (None, 400, 400, 16)
Depth_Input: Conv2D | input: (None, 400, 400, 1) | output: (None, 400, 400, 16)
max_pooling2d_636: MaxPooling2D | input: (None, 400, 400, 16) | output: (None, 200, 200, 16)
max_pooling2d_642: MaxPooling2D | input: (None, 400, 400, 16) | output: (None, 200, 200, 16)
dropout_304: Dropout | input: (None, 200, 200, 16) | output: (None, 200, 200, 16)
dropout_307: Dropout | input: (None, 200, 200, 16) | output: (None, 200, 200, 16)
conv2d_630: Conv2D | input: (None, 200, 200, 16) | output: (None, 198, 198, 32)
conv2d_635: Conv2D | input: (None, 200, 200, 16) | output: (None, 198, 198, 16)
max_pooling2d_637: MaxPooling2D | input: (None, 198, 198, 32) | output: (None, 99, 99, 32)
max_pooling2d_643: MaxPooling2D | input: (None, 198, 198, 16) | output: (None, 99, 99, 16)
conv2d_631: Conv2D | input: (None, 99, 99, 32) | output: (None, 97, 97, 32)
conv2d_636: Conv2D | input: (None, 99, 99, 16) | output: (None, 97, 97, 32)
max_pooling2d_638: MaxPooling2D | input: (None, 97, 97, 32) | output: (None, 48, 48, 32)
max_pooling2d_644: MaxPooling2D | input: (None, 97, 97, 32) | output: (None, 48, 48, 32)
conv2d_632: Conv2D | input: (None, 48, 48, 32) | output: (None, 46, 46, 64)
conv2d_637: Conv2D | input: (None, 48, 48, 32) | output: (None, 46, 46, 32)
max_pooling2d_639: MaxPooling2D | input: (None, 46, 46, 64) | output: (None, 23, 23, 64)
max_pooling2d_645: MaxPooling2D | input: (None, 46, 46, 32) | output: (None, 23, 23, 32)
concatenate_25: Concatenate | input: [(None, 128), (None, 128)] | output: (None, 256)
flatten_109: Flatten | input: (None, 256) | output: (None, 256)
dense_302: Dense | input: (None, 256) | output: (None, 16)
dense_303: Dense | input: (None, 16) | output: (None, 2)
...

Figure 57- Layer concatenation.

The concatenation carried out is not the one illustrated in Figure 57. This image serves only as an example. The original concatenated model is too extensive to portray here since it is basically the connexion of two already vast models. The performance of this concatenated model while making predictions is shown in section 5.

4.6.2 Haar Cascade Classifier Development

A Haar Cascade classifier is an object detection algorithm that can be used to detect anything, including faces and other patterns. It uses positive and negative images to train the classifier. In this dissertation, a Haar Cascade classifier was built to check and detect if a subject appeared in front of the camera. This became a major security measure of the work, and its responsibility was to prevent an outsider from tricking the system by making it think that the outsider was a known person in the dataset.

For example, in old 2D face recognition systems, security was not so well employed as it was in 3D systems, meaning that someone could pretend to be someone else by showing to the camera a picture of a known subject of the dataset. The result was that the same picture was validated as a known subject, even though it was an outsider that was showing the picture. Needless to say that this event caused a security breach and if, there was confidential data involved, it would have become compromised.

To assure this does not happen by any means, a Haar Cascade was built to detect if a subject was standing in front of the camera. This detection checks for the depth of a subject, and if it detected a non-face depth map, it would not run the predictions. Therefore, when a picture of someone is shown on a smartphone, the system would consider it as noise, and would not proceed with the predictions of both RGB and depth models. The final result of this implementation is portrayed in Figure 58.

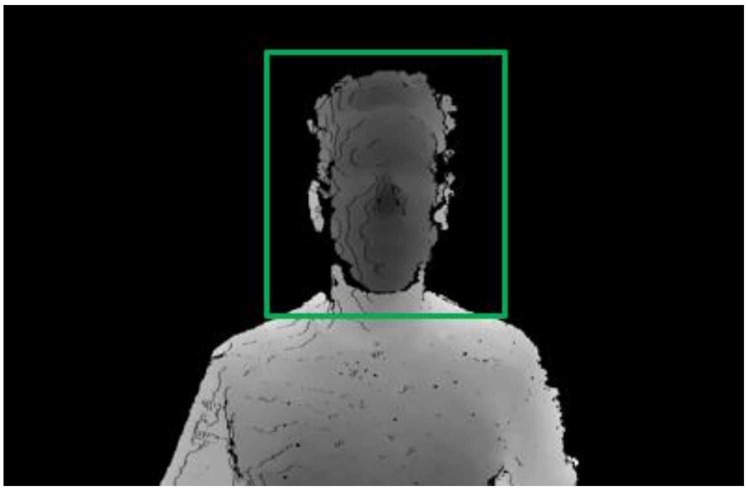
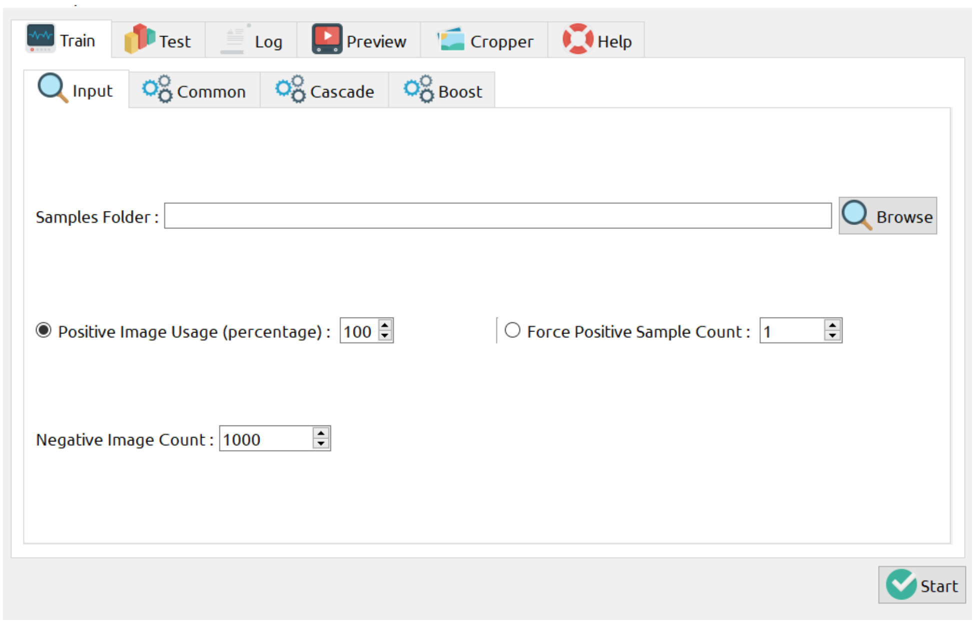


Figure 58- Face depth map detection.

To build the classifier, the program Cascade Trainer GUI was used. This program can be used to train, test, and improve cascade classifier models. It uses a GUI to set the parameters and simplifies the use of OpenCV tools for training and testing classifiers. The program interface is illustrated in Figure 59.



Samples Folder :

Positive Image Usage (percentage) : 100

Force Positive Sample Count : 1

Negative Image Count : 1000

Figure 59- Cascade Trainer GUI.

But before opening the program, two folders in a newly created directory must be created, one for positive images and another one for negative images. The first folder must contain only pictures of the target for the classifier to detect and identify, in this case, face depth maps. Moreover, one wants to detect face depth maps in different poses, so one adds multiple images with multiple poses to the positive folder as shown in Figure 60. The order of the images is irrelevant, as well as the naming scheme for each image. The positive folder has a total of 165 images, and it contains several images from the image capture phase. Images taken in multiple poses were added to increase the detection complexity and to allow the model to detect the face in multiple positions. By doing this, one ensures that the model not only recognizes people showing the front of the face but also the side of it.

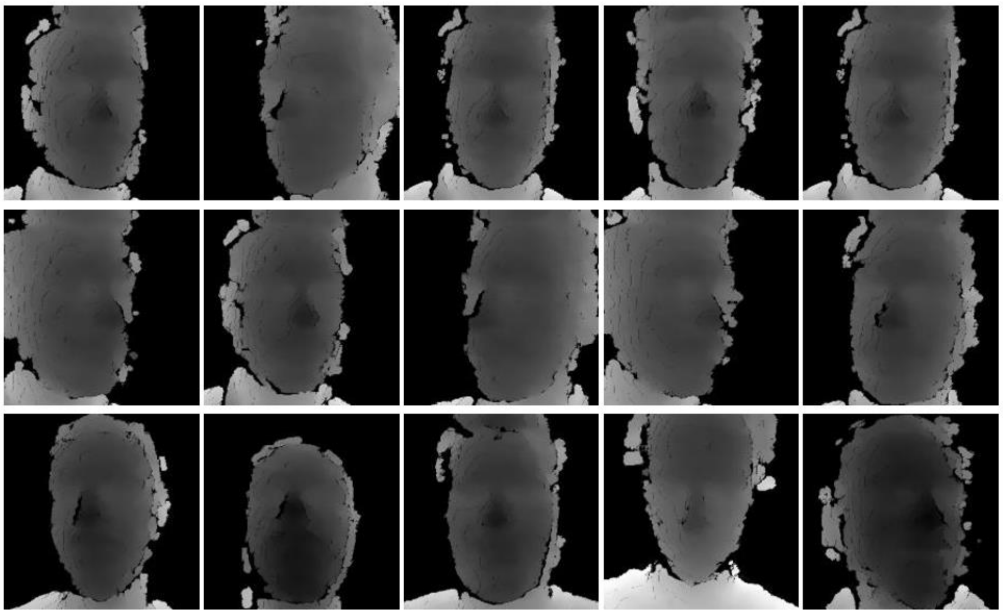


Figure 60- Positive Images.

As for the negative folder, it includes the images that are designated as negative. Negative images are the opposite of positive ones, i.e., any image that does not contain a face depth map. Therefore, for this folder, it is appropriate to include non-face depth maps that could eventually appear next to a face depth map in real-world face recognition, like for example, shoulder depth maps, pitch-black images, white images, and noisy images. Also, depth maps of other objects and even full-body depth maps could also be included. It is recommended to have up to 4x the number of positive images in the negative folder, thus, the total number of negative images is 919.

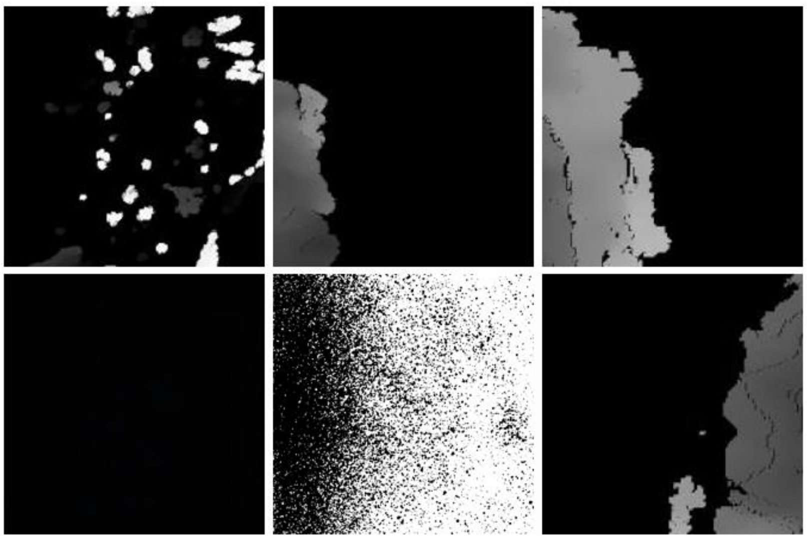
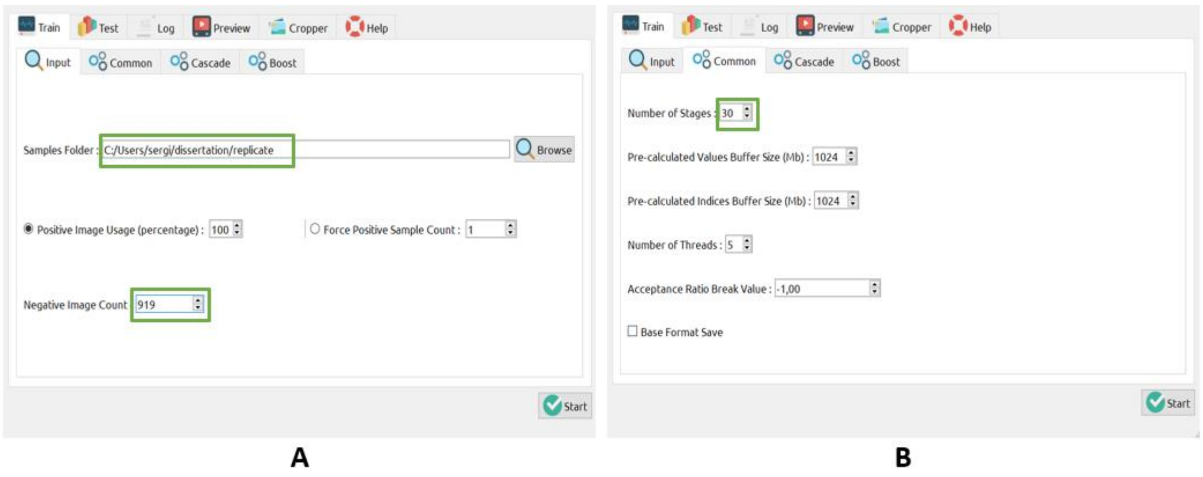


Figure 61- Negative Images.

Having all the images prepared and included in their respective folder, the training of the classifier is ready to start. To do this, in the main tab of the application, “Train”, and in the sub-tab “Input”, it is required to specify the directory which contains both positive and negative folders. The total number of negative images is also required and has to be specified (Figure 62-A).



Samples Folder : C:/Users/sergi/dissertation/replicate

Positive Image Usage (percentage) : 100

Force Positive Sample Count : 1

Negative Image Count 919

Number of Stages : 30

Pre-calculated Values Buffer Size (Mb) : 1024

Pre-calculated Indices Buffer Size (Mb) : 1024

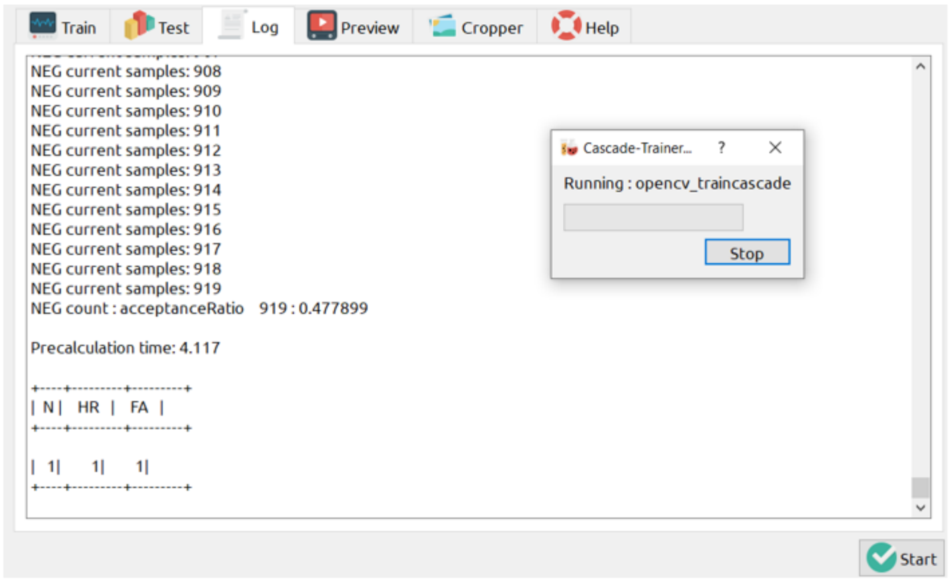
Number of Threads : 5

Acceptance Ratio Break Value : -1,00

Base Format Save

Figure 62- Parameters required for the Cascade Trainer GUI.

The ‘Number of Stages’ parameter also has to be specified. This parameter regards the number of iterations that the classifier uses to learn the images. Here, the final number of stages was 30, but a value of 10 was also used previously. It is important to note that the higher the number of stages is, the more accurate the classifier will turn out to be.



```
NEG current samples: 908
NEG current samples: 909
NEG current samples: 910
NEG current samples: 911
NEG current samples: 912
NEG current samples: 913
NEG current samples: 914
NEG current samples: 915
NEG current samples: 916
NEG current samples: 917
NEG current samples: 918
NEG current samples: 919
NEG count : acceptanceRatio    919 : 0.477899

Precalculation time: 4.117

+----+---------+---------+
| N |   HR  |   FA  |
+----+---------+---------+

|  1|      1|      1|
+----+---------+---------+
```

Running : opencv_traincascade

Figure 63- Training of the classifier.

The downside here is that the time it takes to train these classifiers is proportional to the number of stages. There is no need to modify the remaining parameters. The start button initializes training and prompts what is illustrated in Figure 63. When the training ends, the classifier file with a .xml extension is added to the previously specified directory, like shown in Figure 64. Then, similarly to the Haar Cascade file used to detect RGB faces in the image capture stage, the resulting file can also be used alongside OpenCV to detect face depth maps.

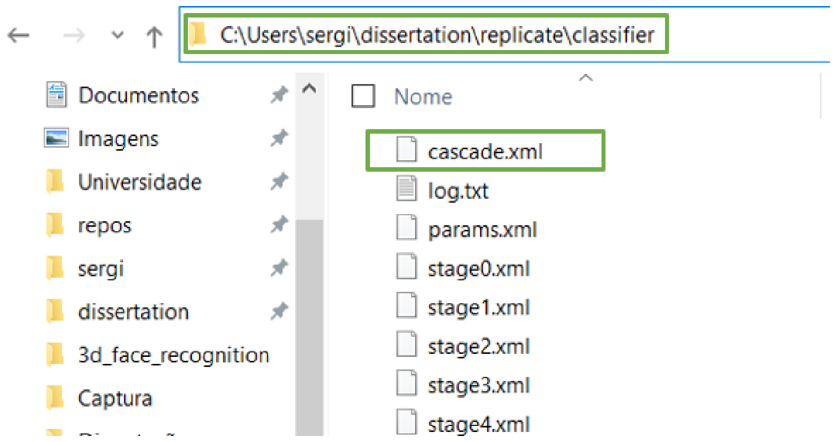


Figure 64- .xml file that is created after the final stage of the training.

5. Tests and Results

In this section, all the results from the methods covered above are presented and demonstrated. The main emphasis is to provide multiple tests and results primarily from the training methods, but also from other relevant aspects that indirectly influence the overall training process, like performance comparison using different platforms as well as performance comparison using different technologies and approaches. The tests appear by order of their relevance throughout the development of the dissertation.

5.1 Haar Cascade vs MTCNN detection algorithms

Both these algorithms perform face detection using the camera's RGB frames. The detection is made for each frame immediately after initializing the camera. Unlike Haar Cascade where the .xml file has to be imported from a specific folder containing this same file, to use the MTCNN algorithm, the entire library or specific functions of the MTCNN library have to be imported to PyCharm.

Supposing both algorithms are executed simultaneously and independently from each other, the results should be similar to the ones portrayed in Figure 65. The top images correspond to the results of the MTCNN algorithm, whilst the ones below correspond to the Haar Cascade.

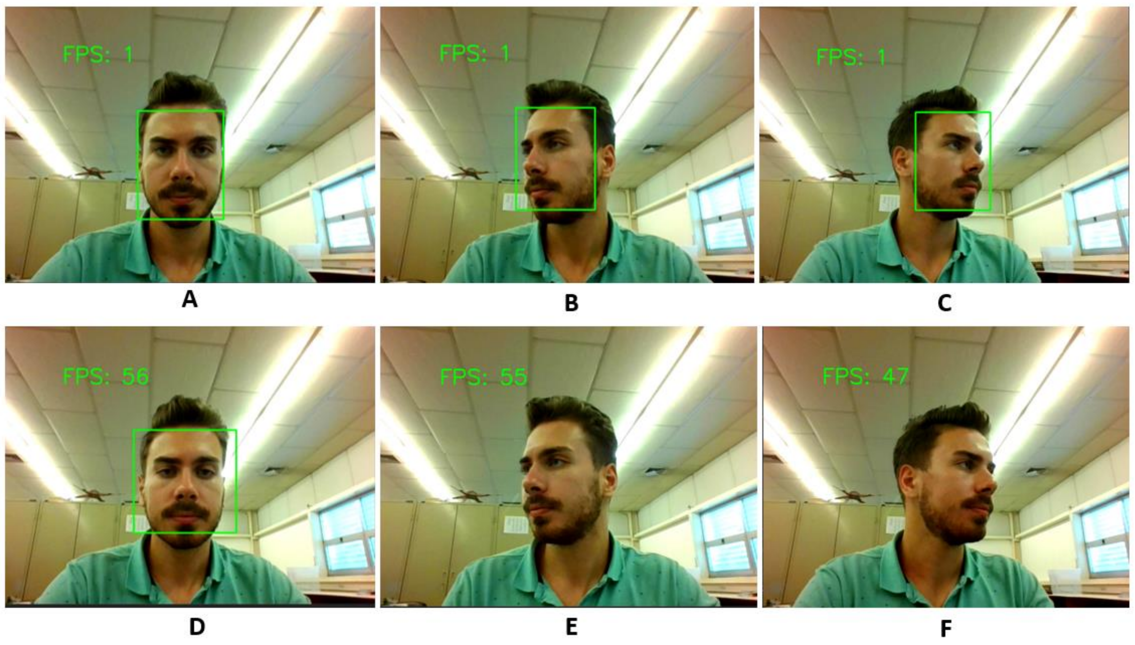


Figure 65- MTCNN (A,B,C) vs Haar Cascade (D,E,F).

MTCNN has a rather poor performance regarding frame rate since it only maxed out at 1 FPS. Haar Cascade, on the other hand, runs faster, i.e., it is faster at making detections, enabling a higher frame rate, averaging at around 50 FPS, which is about 50x more than the MTCNN algorithm. This result, in practice, makes a significant difference when performing the capture procedure, since it is easier for a person to capture its face with almost zero-latency feedback from the camera.

Regarding frame rate, Haar Cascade is clearly better, but concerning detection in multiple poses, it is noticeable that MTCNN is more capable of detecting faces sideways, as shown in Figure 65. It is also slightly better at obtaining the coordinates, since it does a better job surrounding the face's edges, leaving out background details, namely visible in a "look straight" pose. Despite this, the fact that it does not allow more than 1 FPS is enough to abandon the MTCNN method since it becomes a very dull process to perform image capture with.

Since performance and optimization were prioritized, the algorithm used in the final script to perform image capture, and later in the script to perform predictions through the usage of face detection was the Haar Cascade one, for the reasons described above.

5.2 Google Colab vs Jupyter Notebook

To measure the difference in performance of both Google Colab and Jupyter Notebook platforms, training was performed using the architecture defined in Figure 51. This test was performed on both platforms using the same model, the same image generators with equal image augmentation parameters, the same hyperparameters applied for the architecture, and the same fit function. Table 3 illustrates the time in seconds it took to run 10 epochs on each platform. Note that this training was only executed to extract the training time per epoch, hence, the accuracy results were irrelevant. The parameters used to train this network regarding image augmentation were the ones shown in Figure 44. The input shape of the images introduced to the network in both platforms was (300, 300, 3), and the batch size used for the training set was set to 64, while the batch size used for the validation set was 32. The optimizer used was Adam, with a learning rate of 0.0001.

Table 3- Training execution time per epoch between Google Colab and Jupyter Notebook.

Epoch \ Environment	Jupyter Notebook	Google Colab
1	34 s	20 s
2	35 s	18 s
3	36 s	18 s
4	35 s	19 s
5	35 s	19 s
6	39 s	18 s
7	38 s	19 s
8	39 s	18 s
9	41 s	18 s
10	42 s	18 s

The results show the clear advantage of using Google Colab. The duration of each epoch is smaller and more consistent throughout the whole training when compared to Jupyter Notebook's training. Even though the difference between time per epoch is approximately 20 seconds, this value changes if the number of epochs is superior to 10, as well as it would increase if more samples were added or even the number of classes in the dataset increases.

It is also important to note that there is a correlation between the size of the network and time of training, i.e., the bigger the architecture, the longer it takes to complete the training. These results represented the first turning point of this dissertation when the training time was revealed to be much quicker using Google Colab, making henceforth trainings to always run on this platform.

5.3 Kinect dataset parameter fine-tuning

The neural network developed in Figure 51 was subjected to several tests. The first ones were executed to compare the difference in using different learning rates. For this set of tests, the architecture of the neural network remained the same, as well as the image augmentation parameters, number of layers, and optimizer. The number of epochs varied according to the Earlystop function, thus for some tests the number of epochs was slightly superior to other ones. Four different learning rates were tested, 0.001, 0.0001, 0.00001. and 0.000001 since it is part of the DL convention to test with this set of values. Other values in between these could also be used, as well as other positive learning rates like 0.1 and 1, although learning rates greater than 1 are not recommended.10

All the learning rate tests used the same image augmentation parameters, portrayed in Figure 44. The 'target_size' parameter, which was set to (300,300,3) in the previous test, remains the same, as well as the batch size values for the training set and validation set, 64 and 32, respectively. Besides these, the optimizer used in the image below was Adam. The results of this test are visible below in Figure 66.

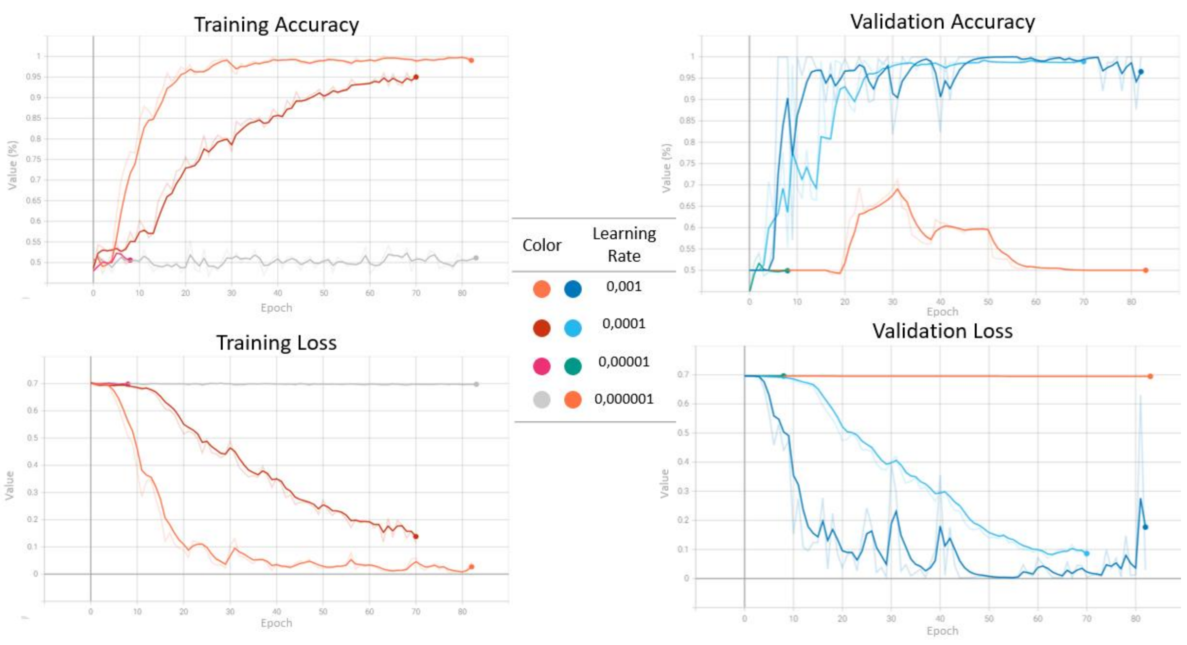


Figure 66- Different learning rate analysis with Kinect dataset.

According to the image, the only training in which the model properly converged was the training with the learning rate set to 0.001, finishing training at the end of epoch 82 with an accuracy value of 0.9905, a loss value of 0.04349, and total training time of 27 minutes and 48 seconds. However, the 0.0001 learning rate also converged, in fewer epochs but maxing out at 95% accuracy and a loss of 0.15. Both the learning and loss curves of the 0.001 learning rate are the steepest amongst the other learning rate tests, meaning that this model learns quicker even though it took longer to finish training. As for the 0.0001 learning rate, both its resulting slopes are not as inclined as the 0.001 one, denoting that this model took longer to learn and to converge. The remaining learning rates showed odd behaviours, in a way that either stopped too early in training or became stuck with a high training error.

As for the performance in the validation set, it is possible to view an expected behaviour according to the training performance. 0.001 converged revealing fluctuation, especially in the middle of training. In practice, this means that, in the validation set, a portion of the samples are being classified randomly, hence the fluctuation observable. As the training evolves, this fluctuation is gradually reduced. The second learning rate has a better performance on the validation set with significantly less fluctuation, and it converges more smoothly. As for the smaller learning rates, they are too small for the model to even learn. Between the first and the second learning rates, it is clear that 0.0001 is more stable especially in the validation set, thus revealing to be the most appropriate choice to use. Based on these results, a learning rate of 0.0001 is the one to use for the following tests.

For this test, 4 different optimizers were tested, being them Adadelta, Adam, Stochastic Gradient Descent (SGD), and RMSprop. The 3 latter ones are amongst the most widely used optimizers in the scientific community. The tests were executed very similarly to the previous ones, this time having a fixed learning rate of 0.0001. The results on the performance of the model in the training set are portrayed in Figure 67.

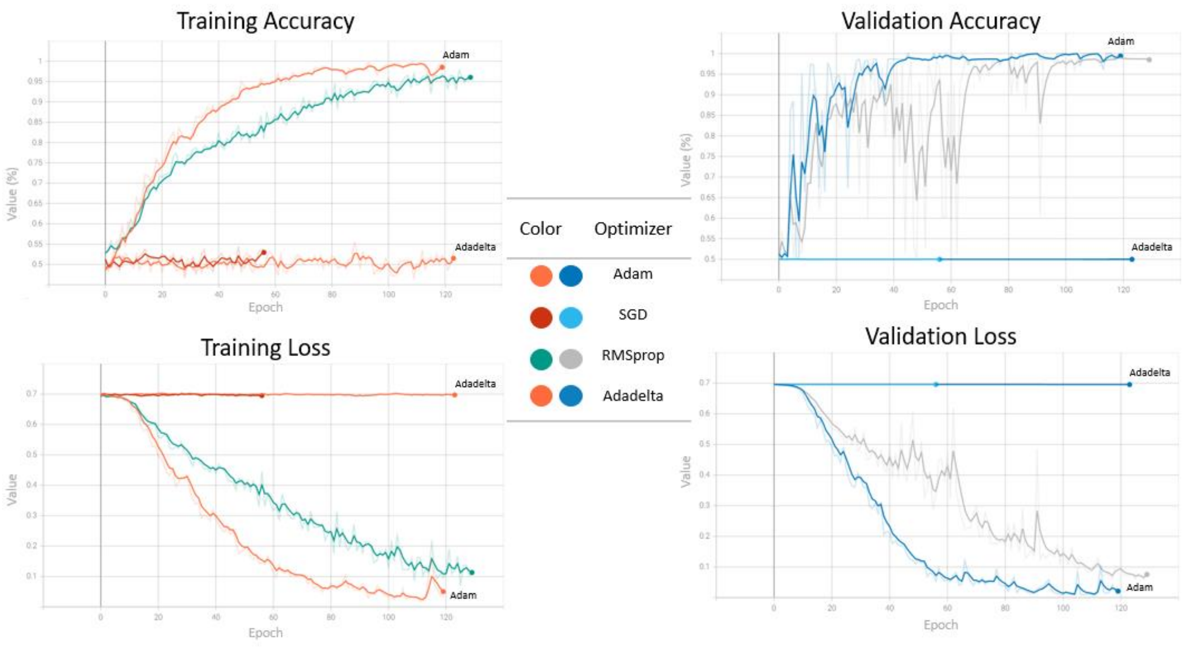


Figure 67- Different optimizer analysis with Kinect dataset.

By analysing the image above, it is safe to say that the Adam and RMSprop optimizers had the best outcome while SGD and Adadelta did not evolve whatsoever through the course of the training process. Regarding loss, Adam had the lowest value (0.05034) while RMSprop's minimum loss was 0.1132. The total time it took to train, was approximately 43 minutes for both optimizers. Training performance metrics, both accuracy, and loss should not be the decisive factors to either choose an optimizer nor even any other hyperparameter. The performance of a model in the validation set must be analysed since the validation set is composed of a set of new data samples that are new and were never seen by the model, thus being validation metrics the most important ones to evaluate how good a model is.

As expected, SGD and Adadelta's validation results did match its respective training results, meaning that neither model learned the data. In contrast, the remaining optimizers did learn the data with Adam converging quicker and with less fluctuation when compared with RMSprop. Notably, the validation accuracy is bigger than the training accuracy. Normally, training accuracy should have the highest value since training data is something with which the model is already familiar with. In this case, since the training used image augmentation (according to the ones in Figure 49) and the validation did not use any type of image augmentation, hence, there is more data diversity to learn than in the validation set, which in turn takes longer to train to achieve higher accuracy rates. Taking into consideration the results obtained, the Adam optimizer is clearly the most suitable choice according to the data and model architecture used.

For the following tests, the influence of using the architectures stated in chapter 2.4, through the use of transfer learning is evaluated. All the tests were performed using the Adam optimizer with a learning rate of 0.0001, a batch size of 16 in the training set and a batch size of 8 in the validation set. The Earlystop function used a 'patience' parameter of 15.

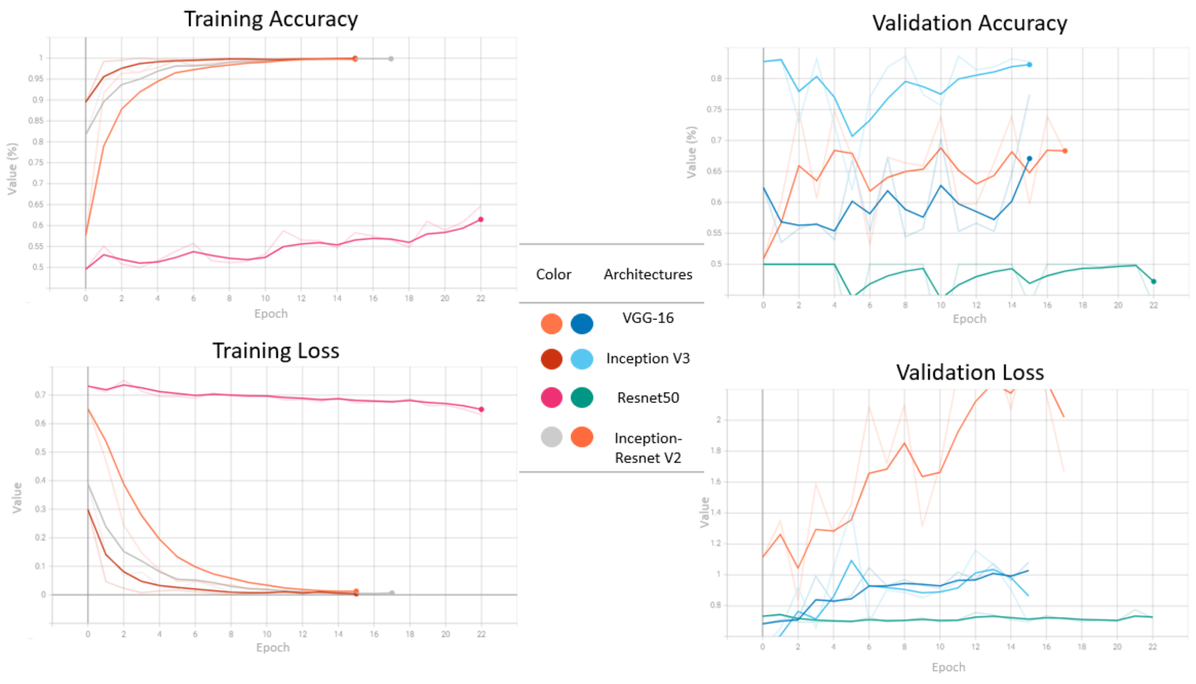


Figure 68- Different architectures used with the Kinect dataset.

It is possible to conclude that at first glance, VGG-16, Inception V3, and Inception-Resnet V2 learned the data, since throughout the analysis of the performance on the training set, all of the 3 converged with accuracy values of around 0.99 and a loss value very close to 0. As for Resnet50, this one barely surpassed 60% accuracy before the Earlystop function decided to halt its training.

However the outcome is completely different when analysing the performance on the validation set. As shown in Figure 68, none of the 4 models performed decently in the validation set. This can indicate that the model completely memorized the samples of the training set, indicating a clear overfitting problem.

There is a possibility that these architectures are way too complex to deal with the current data since the number of images for this dataset is not particularly large, despite having image augmentation. To solve this, one could easily change the image augmentation function or even add more samples to the training set, although that would mean taking a step back in the training process to re-capture more images for both the training set and validation set.

The test phase (or prediction phase) is where the ability of the developed neural network(s) to make predictions on data that the model has never seen is tested. According to the results, the model built from scratch, without transfer learning, is the one to choose to make the predictions. The results of the predictions with RGB images made by the Kinect model are portrayed in Figure 69. Regarding depth, no network was ever built since in the meantime the Intel RealSense was acquired.

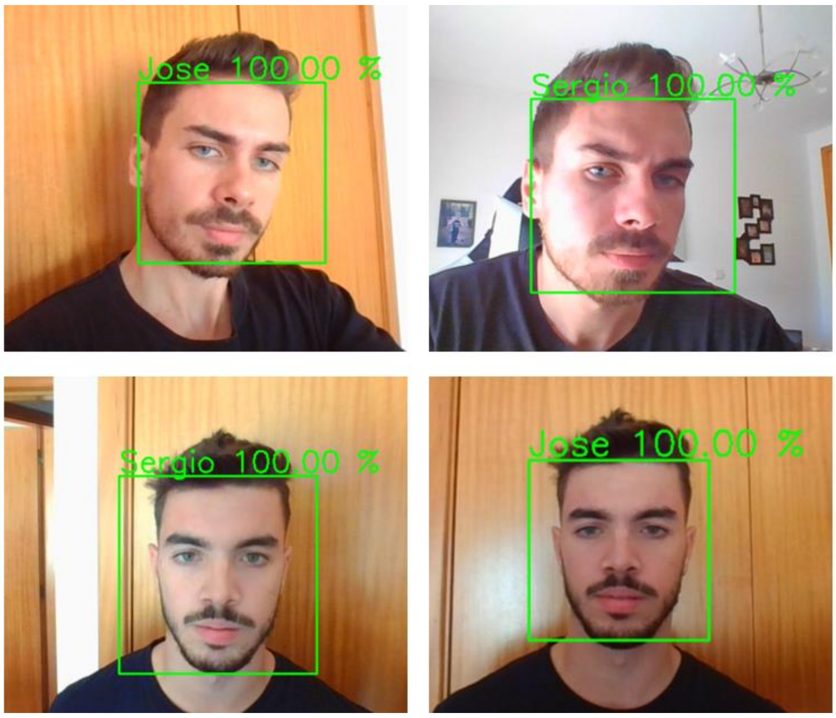
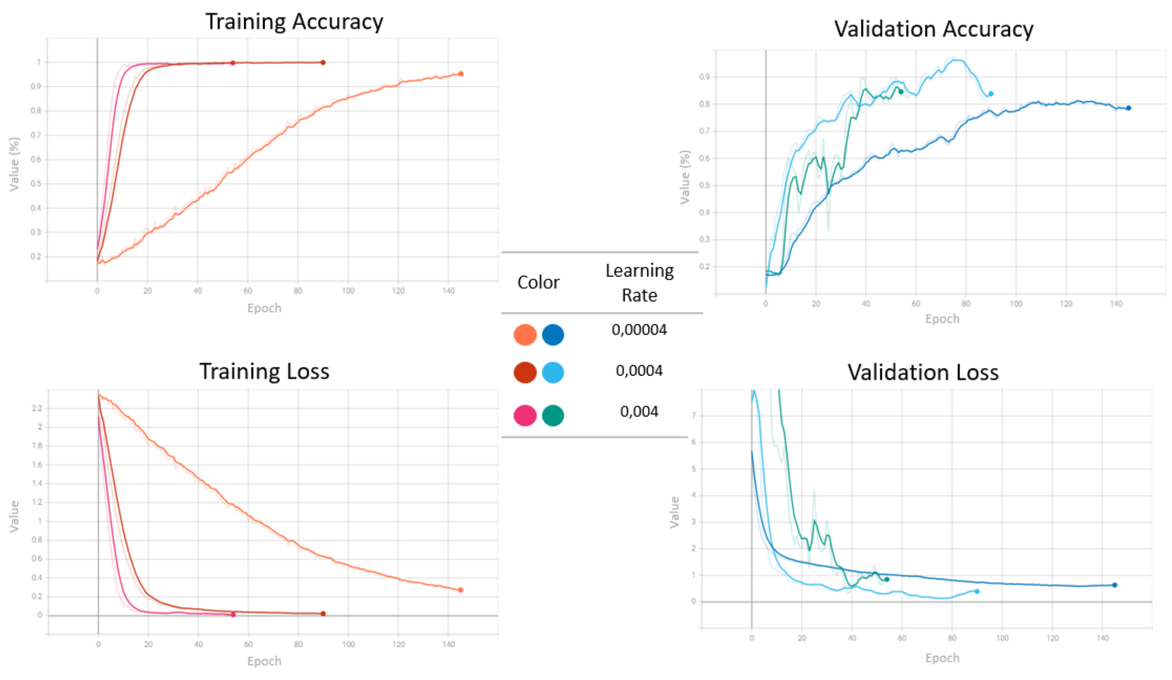


Figure 69- Kinect prediction results.

The results show that this model does not perform well on real-time predictions. It often mistakes both subjects, and the confidence is constantly at 100%, despite the wrong prediction. This means that this model is not suitable for this face recognition task due to its poor performance. To solve this inability to predict correctly, one could try to rethink the whole architecture of the model, or even perform some more changes to the dataset, despite the poor image quality. However, making these changes does not guarantee the success of the predictions with this model.

5.4 RealSense RGB dataset parameter fine-tuning using transfer learning

Since an Intel RealSense D455 was acquired, it was decided that the development with the Kinect would cease. This decision was made upon the reasons pointed out in 4.2.2. In this segment, the tests are performed using transfer learning. Initially, all the weights were imported from ImageNet. The first architecture used was a VGG-16 with 3 different learning rates to train the model. The results are portrayed in Figure 70. The highest learning rates converge quickly on the training set, in a smaller number of epochs. In contrast, the behaviour changes in the validation set, being noticeable that the lowest learning rate (0.00004) has smoother learning curves.



Color	Learning Rate
● ●	0,00004
● ●	0,0004
● ●	0,004

Figure 70- Training results of VGG-16 on RGB RealSense dataset using transfer learning.

Besides the curves, the smallest learning also converges to about 80% on validation and a loss of nearly 0.64. The model with a learning rate of 0.0004 also shows some fairly good behaviour on the validation set, whereas the remaining model shows some fluctuations associated.

Nonetheless, testing with new and unseen data outside the training process must still be done. An additional function was developed to quickly test and analyse the results of the trained model. However, these tests were not performed in real-time, instead, they were always performed by fetching each new image from a given directory, do the required processing and normalization for each image and lastly, make a prediction and return its result. For reference, Figure 71 (Top), portrays the correct name (class) for each of the subjects in the RealSense dataset. Figure 71 (Bottom) has 3 unknown subjects to the trained models. Note that neither datasets used a class named "Unknown", although this could be used here, since it is in fact used in some DL projects. Instead, in this work, an unknown prediction is a prediction whose percentage of confidence is inferior to 85% (at the time of the prediction of these tests). The percentage of confidence is, like the name says, the percentage of how sure the model was with the result of the prediction it made.

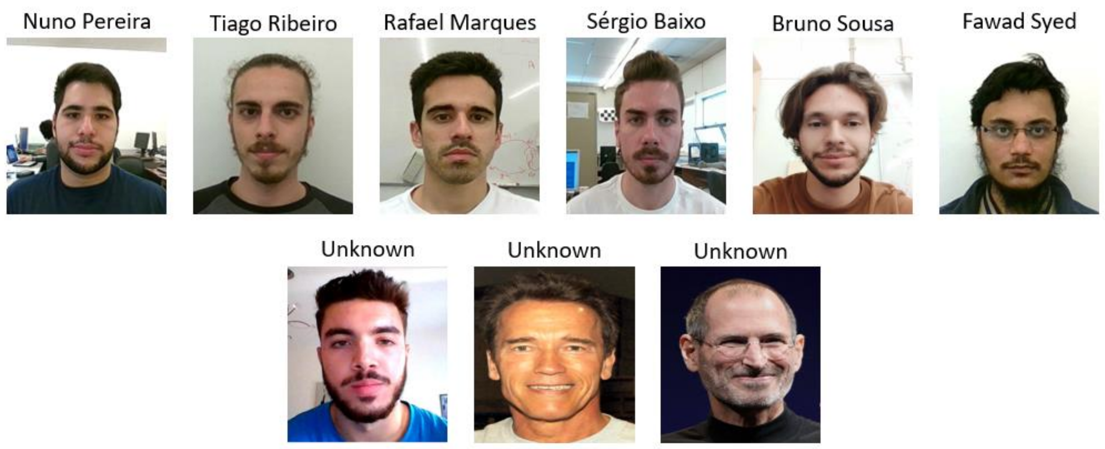


Figure 71- Illustration of the 6 classes of the RealSense dataset (Above). 3 Unknown subjects to the model (Below).

By performing predictions with the VGG-16 model trained previously, in Figure 72 in some of the images, the model made the right prediction, while in others it predicted them wrong. This set of predictions used a learning rate of 0.00004. The worst-case scenario is observable here which is when the model thinks that an unknown person is a subject of the dataset. This event happened in 3 predictions, and all of them with very high confidence scores. Besides this, the model also failed to make the correct prediction in several other images, which makes this model, having a learning rate of 0.00004, unreliable and untrustworthy in real-time image prediction.

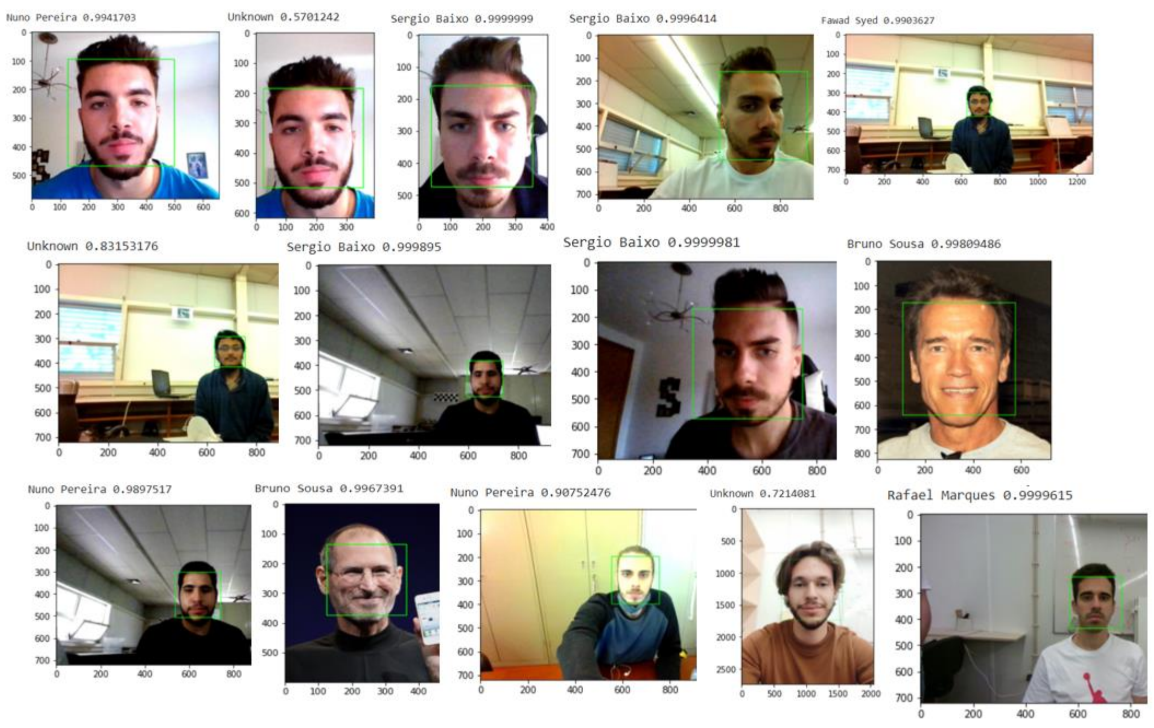


Figure 72- Prediction results of the VGG-16 model with a learning rate of 0.00004.

The results were very similar by changing the learning rate to either 0.00004 or 0.004, with only some changed results in one image or another but, the model still made several mistakes, leading to the conclusion that this model is not suitable to deal with the RealSense's dataset. Therefore, another architecture was tested to see if it uncovers more acceptable results.

The next architecture used was the Inception V3, detailed in 2.4.2. Similar to the previous tests using VGG-16, the test with this architecture generated very similar results, at least when comparing the record of training as shown in Figure 73. All models with different learning rates have also converged, indicating that the models have learned the data from the dataset, but in practice, the results on new data may be completely different, like previously seen.

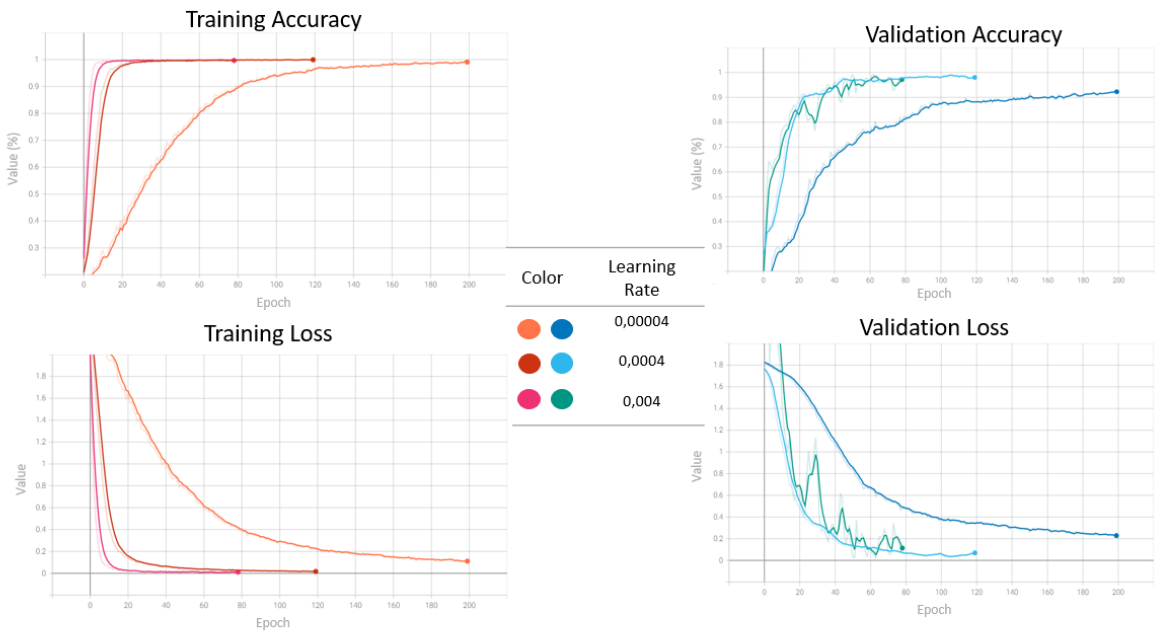


Figure 73- Results of Inception V3 on RGB RealSense dataset using transfer learning.

Figure 74 shows only some prediction results from all 3 models obtained by training with different learning rates, 0.00004, 0.0004, and 0,004 (see Figure 73). It is noticeable that the results obtained are not consistent, since on some occasions all 3 models may or may not get the correct prediction.

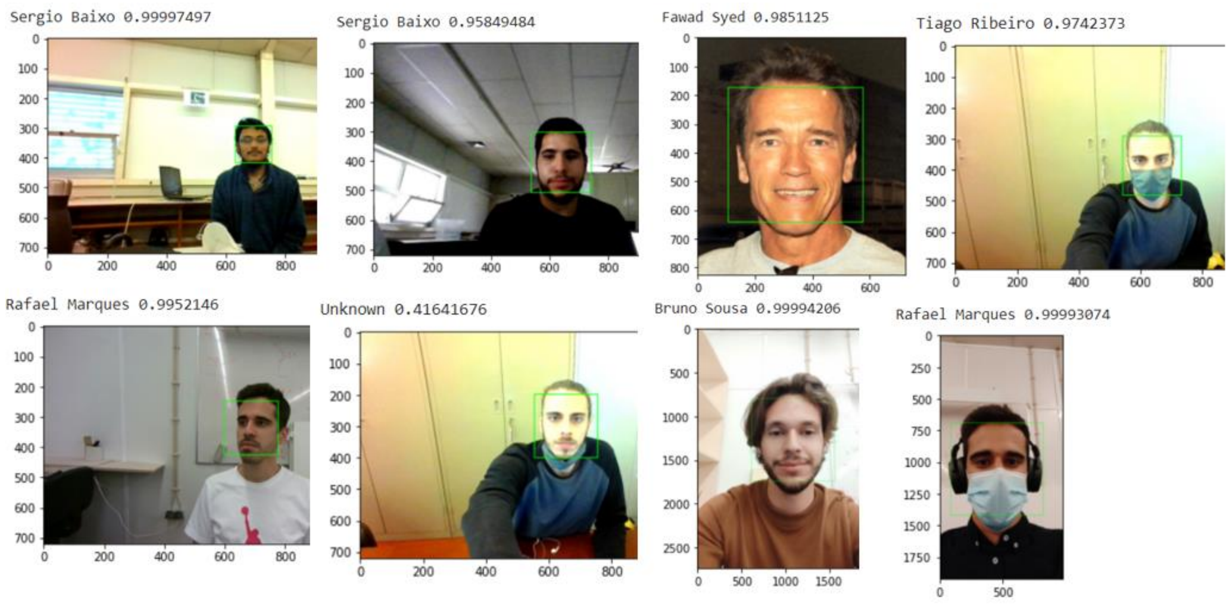


Figure 74- Prediction results of the Inception V3 model using several learning rates.

By looking at Figure 74, it is clear that neither model has scored a perfect prediction using the Inception V3 architecture. Like with the VGG-16, this can indicate that the dataset is not appropriate for this architecture, or simply needs more time to absorb and learn the data. But one has to be cautious, since to learn more data, more training epochs are needed, and this can eventually lead to overfitting problems.

Since results with both architectures were not acceptable at all, maybe the best strategy would be to slightly change the approach. So for the next tests, instead of using ImageNet, like previously, a pre-trained model already trained with human faces was imported into an Inception-Resnet V1 architecture, since this pre-trained model, trained with human faces, was previously trained on this same architecture.

To perform this training, the weights file needs to be locally downloaded. This file was generated by the researchers that performed extensive and costly training on Inception-Resnet V1. After downloading this weights file it must be specified when instantiating the network. This creates the network with the specified weights pre-loaded.

Figure 75 shows the theoretical results across 3 different trainings, each with 3 different learning rates, the ones used previously. It is clear that the results show great performance across all trainings with all of them converging very quickly and maintaining a number of epochs almost inferior to 100.

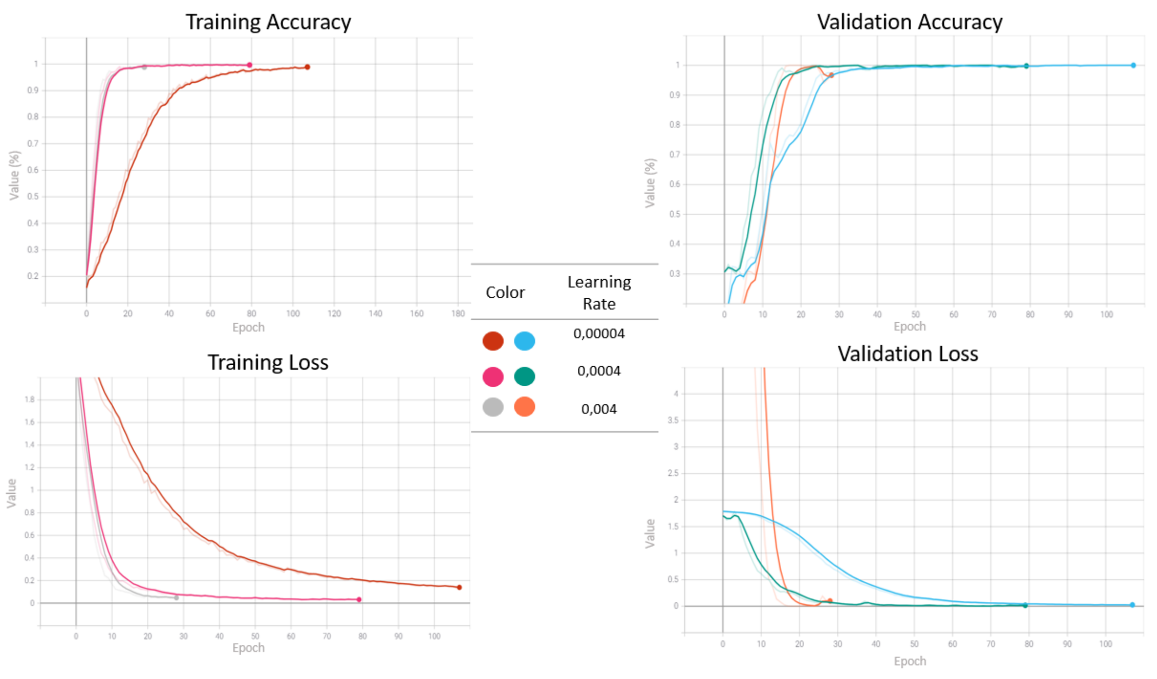


Figure 75- Results of Inception-Resnet V1 with the RealSense dataset using transfer learning and pre-loaded with a weights file from face images.

To determine how the models perform in real-world scenarios, images that were never seen before by the model are given to it to make predictions, just like in previous tests. This time, the performance with new images is remarkably superior to the previously trained models. Considering all 31 images in the test set, the model with a learning rate of 0.0004 misidentified only 2 faces, while the model with a learning rate of 0.004 misidentified 3 faces. On the other hand, the 0.00004 learning rate did exceptionally well since it correctly identified all the 31 test images. Figure 76 shows these misidentified images along with the respective learning rate.

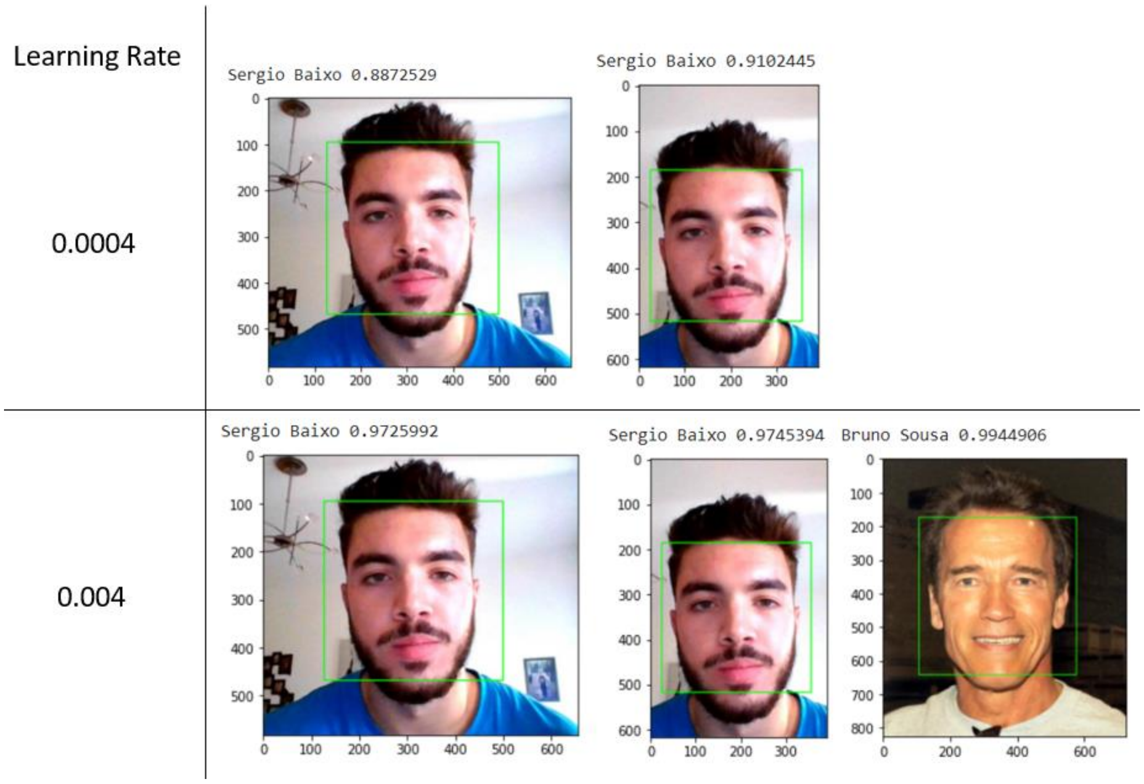


Figure 76- Wrongly predicted images.

It seems that the higher the learning rate, the greater the tendency for the models to make mistakes. This statement can be explained since the 0.004 learning rate did one additional bad prediction and the confidence percentage for the wrong prediction is higher, which is bad since the model is highly sure the image corresponds to its prediction, even though it made a wrong estimate. Using a pre-trained model on faces seemed to make a big difference, and it can be explained by the fact that the model has previously seen faces before, therefore, it already knows shapes, edges, and other face's characteristics. This is also associated with the slightly inferior number of epochs in which the training took place, in comparison with previous training attempts.

Not only the 0.00004 learning rate performed well on closed distance and full shown faces, but it also performed well on extreme conditions, like further away faces, side faces, and faces with a mask on. Figure 77 shows some of these extreme conditions. The model still recognizes Nuno Pereira and Fawad Syed despite they were standing at a longer distance, and even though Fawad was looking slightly to the side. The model also provides low confidence predictions to unknown subjects (around 50%). Besides all of these, the model's most remarkable feat is the ability to correctly predict when the subject is wearing a mask, and it does it with high accuracy. Tiago Ribeiro's and Bruno Sousa's confidence percentages with a mask are very similar to their corresponding images without a mask.

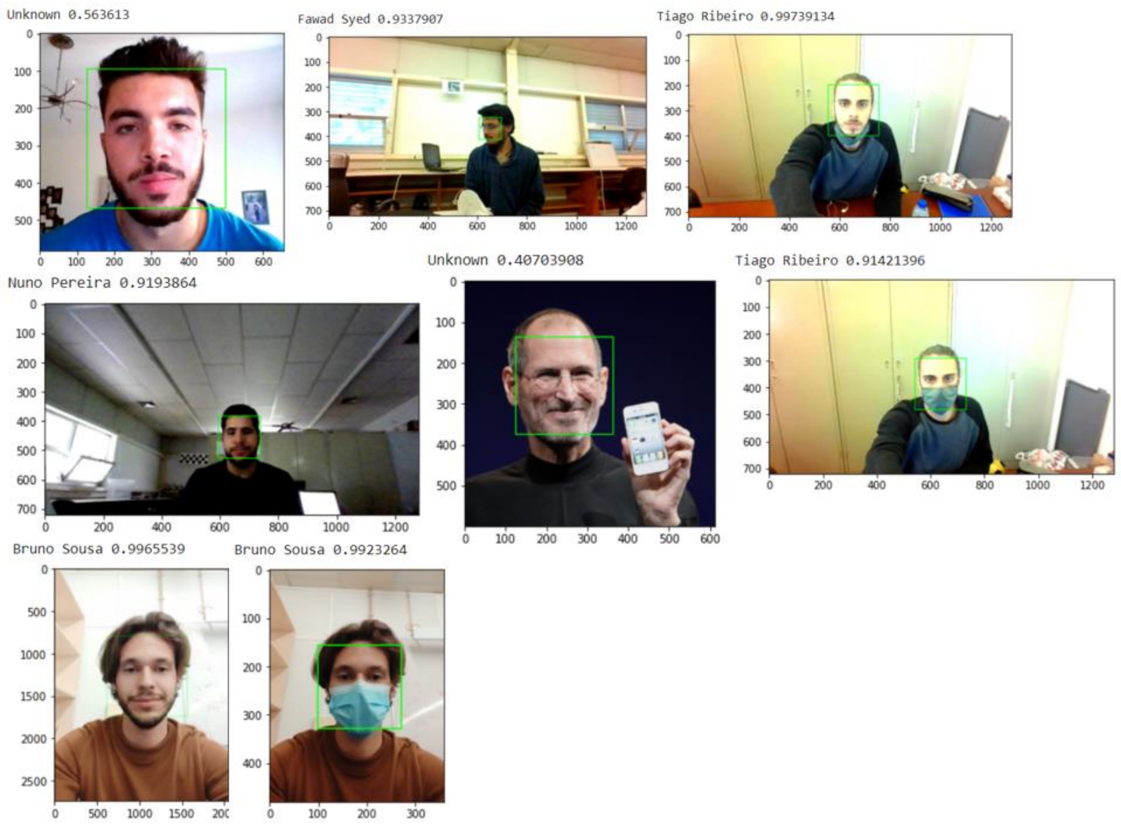


Figure 77- Correctly predicted images (RGB model).

Considering that the model with a learning rate of 0.00004 correctly predicted all the tests, this is the final model selected for the real-time implementation of the system to serve its final purpose. The next step is to test the depth model to see if it also has good performances on depth data like the previous RGB model did.

5.5 RealSense Depth dataset parameter fine-tuning using transfer learning

For the depth dataset, the tests are similar to previous ones. The architecture used here was the Inception V3. This time, the weights were loaded from ImageNet, since there was not any weights file that resulted from the training of depth faces in existence, therefore, the ImageNet one has to be used. As previously seen, this action can impact the result of the predictions later on. The network used is detailed in Figure 56.

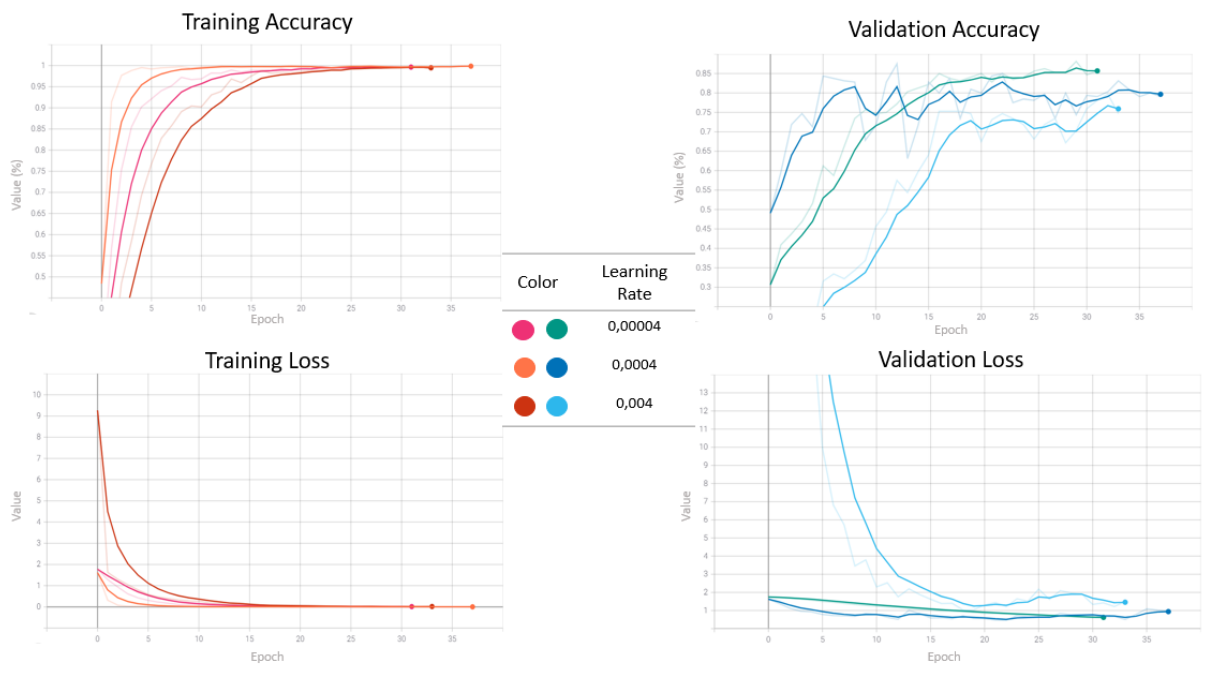


Figure 78- Results of Inception V3 on Depth RealSense dataset using transfer learning.

Having higher learning rates can lead to more fluctuation, since they can cause undesirable divergent behaviour in the loss function. In contrast, having low learning rates may cause a very slow training process, since the model is making very little updates to its weights during training.

Judging by the results in Figure 78, it is clear that the model with a 0.00004 learning rate has a more stable learning process. Overall, with all the tests performed so far, the lowest learning rate has always had the smoothest learning process. It is possible that if the learning rate was lowered even more, maybe to 0.000004 for example, one could end up with a longer training process since it takes more time to converge. To test the capability of the model, the process used with previous models repeats. Figure 79 portrays the results of the predictions.

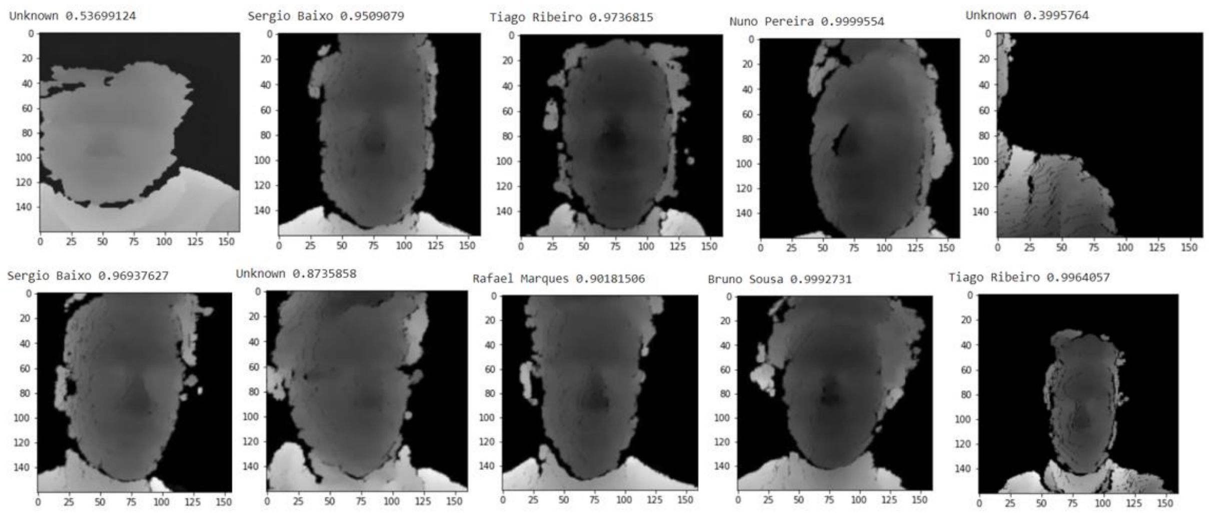


Figure 79- Prediction results of Inception V3 with depth dataset model with a learning rate of 0.00004.

The model correctly predicts the majority of the faces, as shown above, while some others are mistaken either with another face or none at all. The results are similar to those of the remaining learning rates, being the smallest one slightly better.

The conclusion reached when training Deep Learning models with depth maps, particularly face depth maps, is that the results can easily differ since these images can be quite ambiguous. Even for humans, it is not obvious at first glance which depth face image belongs to which subject, therefore, one should not rely exclusively on depth face recognition especially if there are matters of security on the line, as these can be at risk.

The following step is to merge RGB and Depth models as mentioned on page 68. To do so, some changes had to be made to the image generators, both the training and validation ones. The image generators were adjusted so they pass one image at a time to its respective network. The model takes one RGB image from the RGB training set and one depth image from the depth training set simultaneously and pushes them to their respective architecture (if it is an RGB or a depth one). In addition to this modification, the architecture structure was also affected. The full architecture of Inception-Resnet V1 and Inception V3 remains the same used before, but without their tops, i.e., without fully-connected layers, including the last fully-connected layer whose number of neurons corresponds to the number of classes. Then their output, i.e., their last layer is concatenated together, flattened, and some fully-connected and dropout layers are added on top. Lastly, this new model gets compiled using the Adam optimizer and a learning rate of 0.00004. Following this, a normal training occurs, passing the newly created image generators described above.

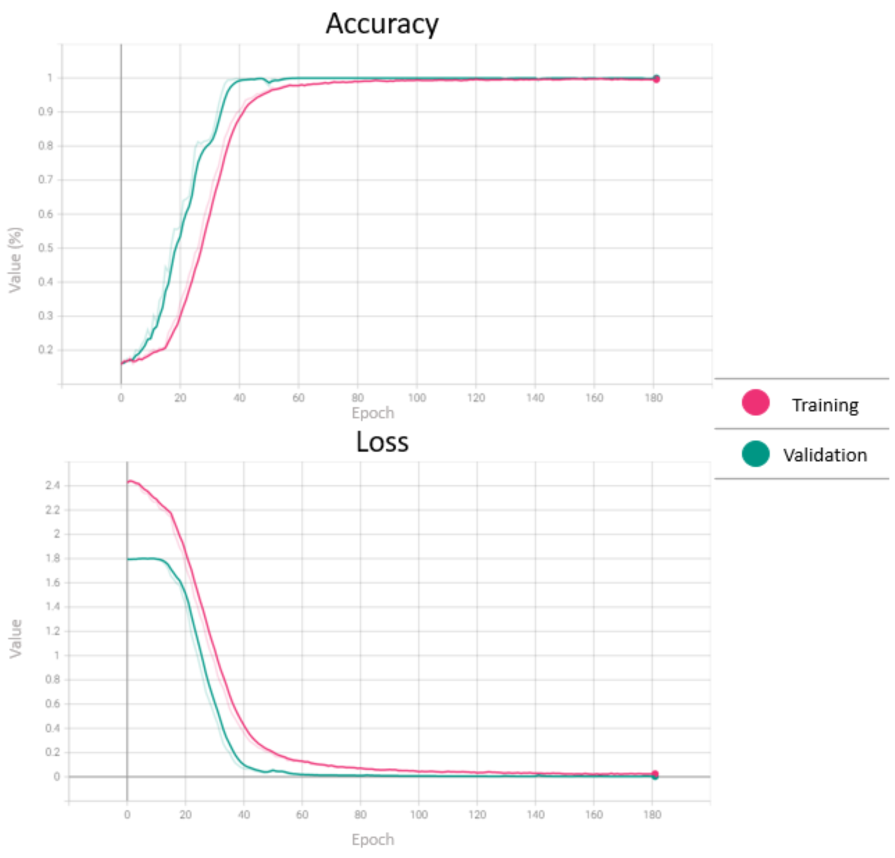


Figure 80- Training results of the merged model.

Figure 80 illustrates the progression of the merged model with a 0.00004 learning rate. By comparing these results with previous ones, it is possible to see that the merged model takes 2 to 3 times longer to train, supposedly since it is training on two distinct architectures before its merging. Besides, the results showed on the graphics are pretty decent. Unfortunately, the results on the predictions made here reveal a worse performance in comparison with the previous models, especially with depth images. The previous RGB model was able to correctly predict images of subjects with a mask on and was able to do so with substantially high accuracy values. Through observation of Figure 81, the model outputs slightly inferior accuracy values regarding RGB images. Besides this, the model has wrongly predicted several depth images. Due to poor performance on the predictions, this merged model is discarded, and the final models to use are the Inception-Resnet V1 for RGB images and the Inception V3 for depth images.

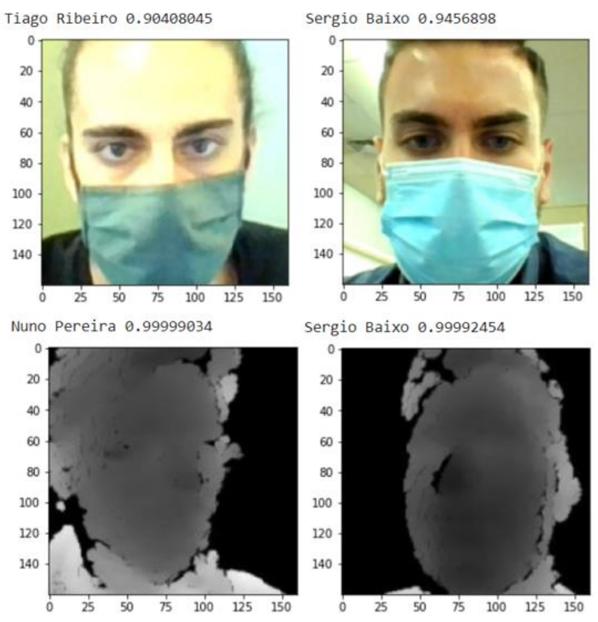


Figure 81- Sample of bad results obtained from the merged model.

The entire architecture, including the weights, can be loaded into a file for future use. Given that the concatenation has failed, the system framework illustrated in Figure 27 becomes obsolete and a new system framework looks like the one in Figure 82.

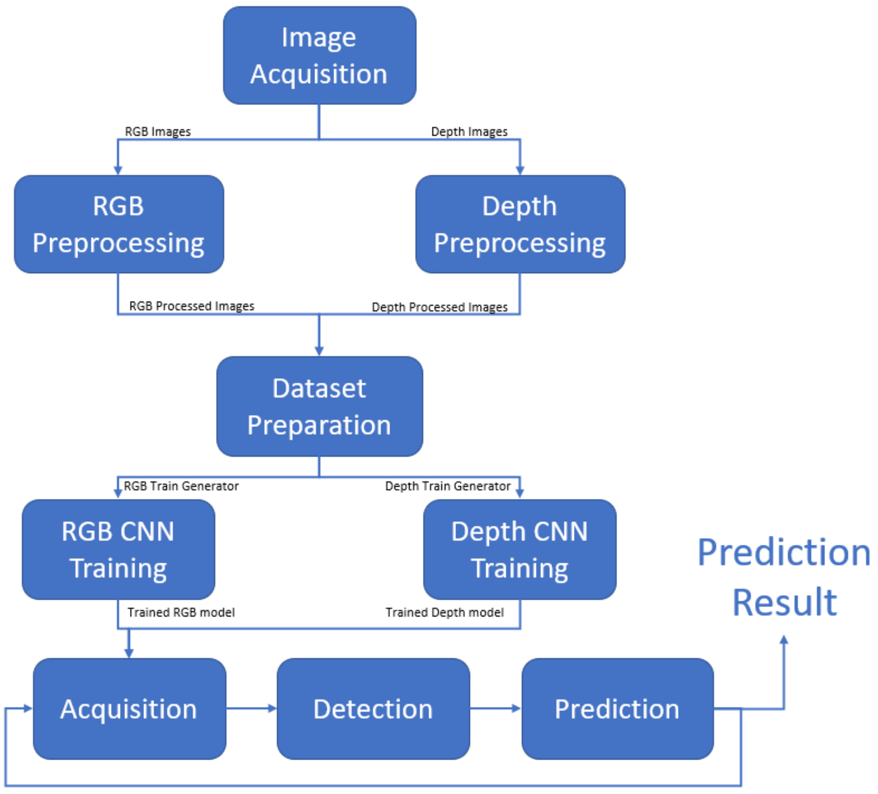


Figure 82- Final system framework.

So far there is a model capable of handling RGB images, and another distinct model capable of handling depth images. Each of these models makes its predictions independently, but the final algorithm to deal with the final real-time image prediction must be elaborated. The final concept is presented on the following pages. One of the ambitions that emerged at the beginning of the dissertation was the need for the system to know if it is being fooled by a picture on a smartphone or even by a printed picture. Thanks mainly to the developed Haar Cascade classifier described in 4.6.2, this was achieved, and one of the purposes for which 3D face recognition was created has been fulfilled, which is the addition of an extra layer of security on top of the recognition process. The developed classifier only recognizes depth faces and fails when it is confronted with depth maps regarding other objects or noise. On top of that, it still maintains two distinct points of prediction (RGB and depth images).

5.6 The Final Concept

In this section, the final prototype is presented. It is the collection of all the successful results performed previously, from the image capturing and processing to the DNNs that performed better. As said earlier, the Haar Cascade classifier is also present to validate the presence of a person in front of the camera. Both this detection and the two predictions are made with the two different neural networks function almost simultaneously.

The program starts by loading both CNN models built for RGB and depth images into the source code. The Haar Cascade classifier file created is also loaded into the code as a .xml file (see Figure 64). Also, the Haar Cascade classifier for detecting regular 2D faces is loaded, followed by a configuration of the Intel RealSense. This includes the establishment of required parameters to deal with the rest of the algorithm, including resolution, FPS, and other internal ones. After this is done, the camera immediately starts acquiring the frames and real live footage from the camera can be seen.

Figure 83 shows an instance of the footage captured when the subjects are not being detected since they are too far away from the camera. In the top image, the depth Haar Cascade is not detecting the depth face because the image does not look much like a depth face, whilst in the bottom image, the face is too close, resulting in noise being introduced by the camera, which in turn is discarded by the classifier.

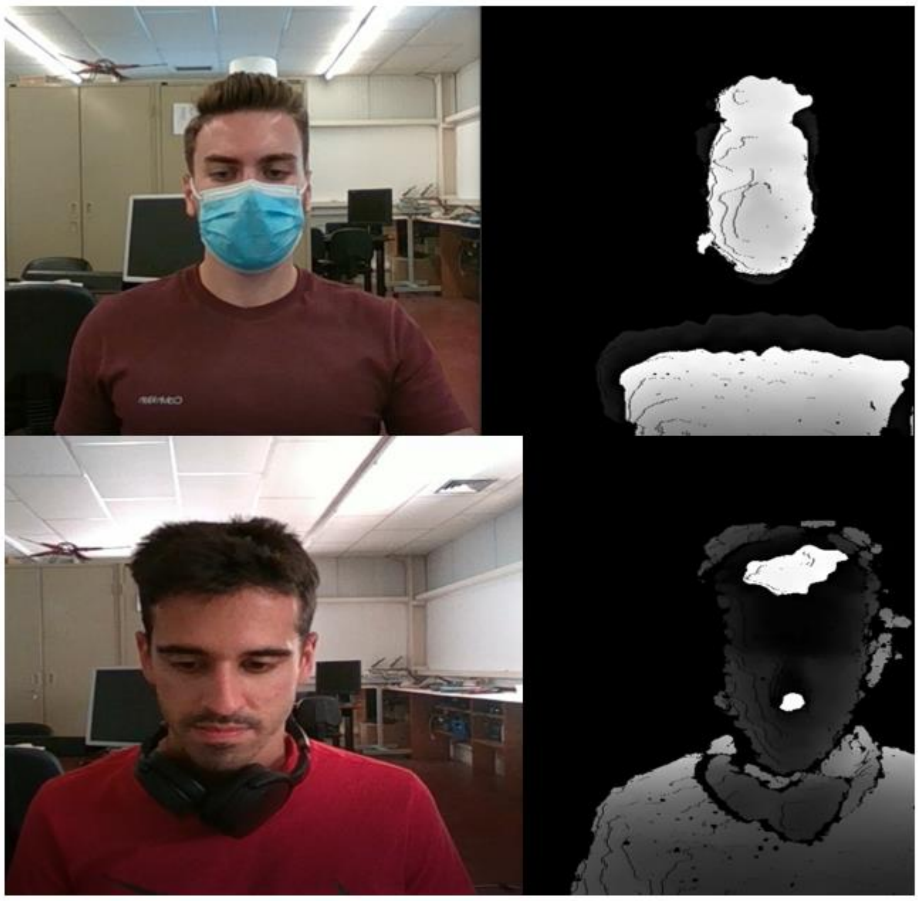


Figure 83- Footage of the camera when subjects are not being detected.

After the parameter establishment, both Haar Cascade classifiers are called to check the presence of a person. Again, one of the cascades checks the RGB frame, while the other checks the depth frame. If none detects a face, the loop continues until a detection occurs. Otherwise, if both of them detect a face, then it enters a loop where some attributes like the distance of the person to the camera are computed. After this, the squares surrounding the face both in RGB and depth frames are drawn. When a face is detected, the algorithm extracts it according to its coordinates on the screen, then applies the different image processing already mentioned above according to each image component. Then, the already processed images enter their respective network and the output of this is the result of the prediction. Figure 84 contains the detection of a subject and respective prediction.

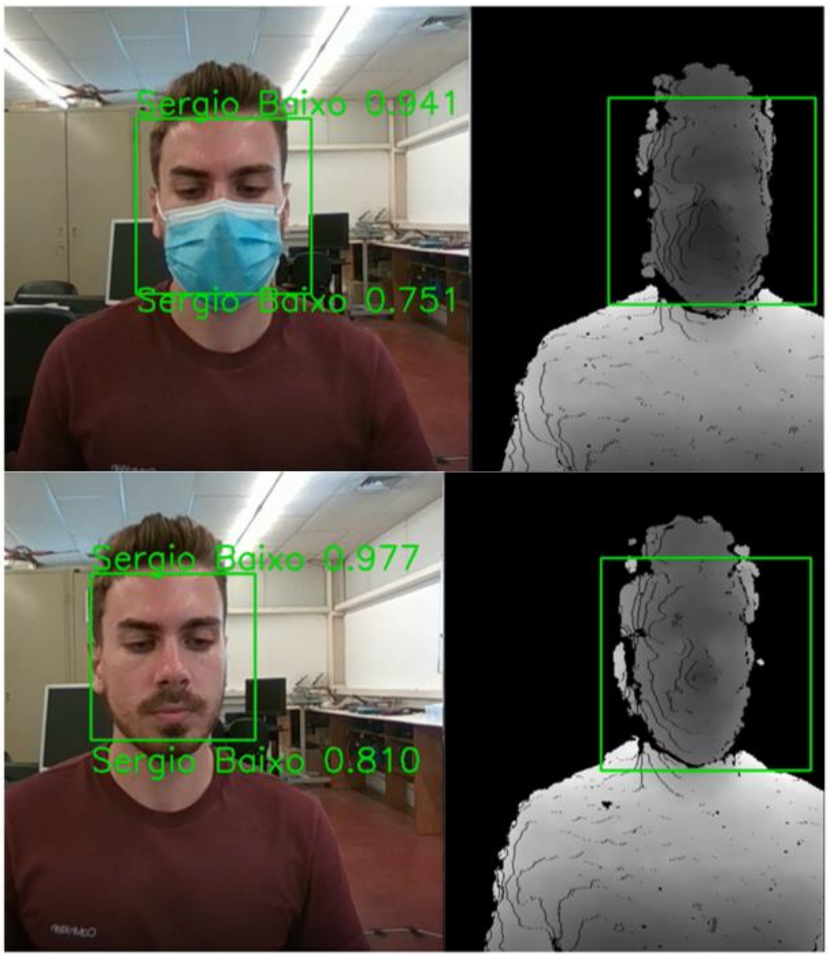


Figure 84- Detection and correct prediction with and without a mask.

The result of the prediction made is composed of the name of the subject predicted and the respective percentage of confidence. The values next to the name of the subject correspond to the percentage of confidence and each model has its own. The top prediction regards the one made by the RGB model whilst the bottom one corresponds to the one made by the depth model.

It is possible to observe an increase in the percentage of confidence when the subject takes the mask off in both models. This behaviour is expected since when the mask is off, there is a lot more information that the model can use to make predictions. Besides, during training, all the images were taken without a mask, meaning that the model has never seen a subject with a mask on until the predictions.

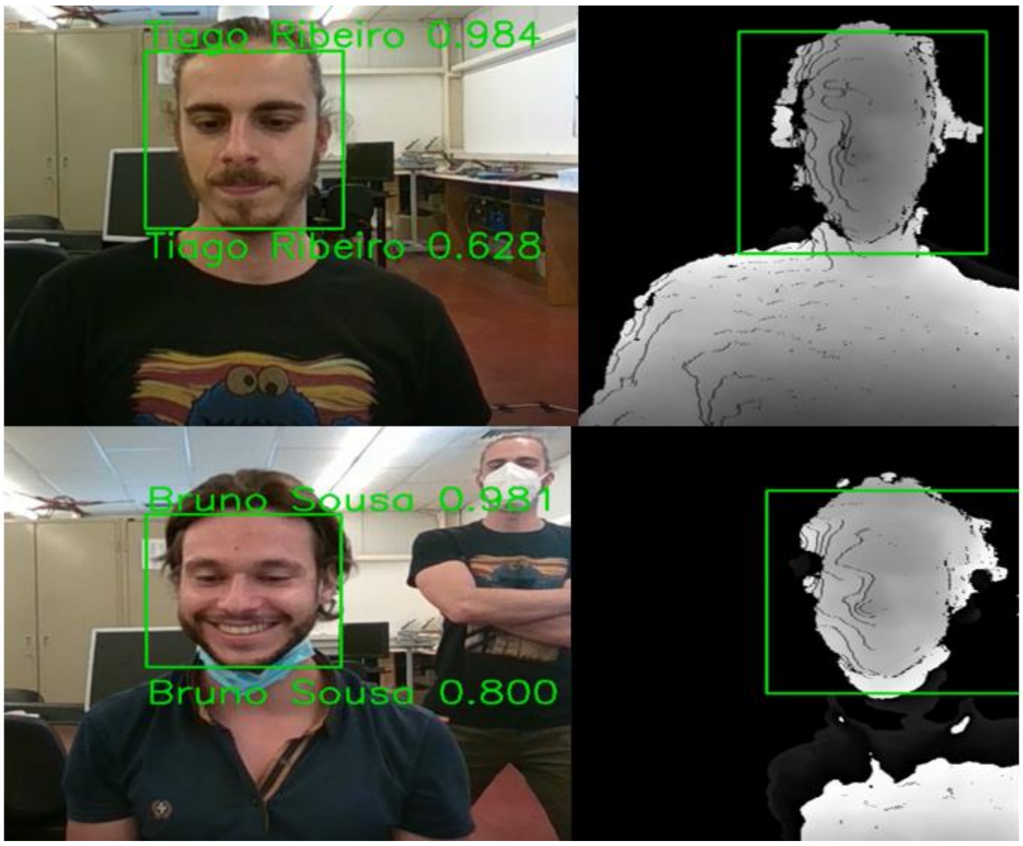


Figure 85- Detection and correct prediction of Bruno Sousa and Tiago Ribeiro.

In Figure 85 two more correct predictions are portrayed, even though depth face maps between these two subjects are quite similar. Bear in mind that in Bruno Sousa's prediction, Tiago Ribeiro, which is behind Bruno is not being detected since the algorithm was configured to discard anything that stands beyond a certain distance.

It is important to note that, despite the algorithm being only developed to make predictions of subjects within a certain distance to the camera, it only shows the result of those predictions if the respective percentage of confidence is at least 87% for the RGB model and 54% for the depth model. This ensures higher security in subject validation.

Figure 86 shows Rafael Marques in two irregular poses, looking slightly sideways with a mask on and another one with the eyes almost closed and in a dark environment. Even in difficult conditions, the RGB model does not struggle at all when making predictions. In contrast, the depth model does struggle, resulting in inferior confidence percentages in comparison with the RGB model.

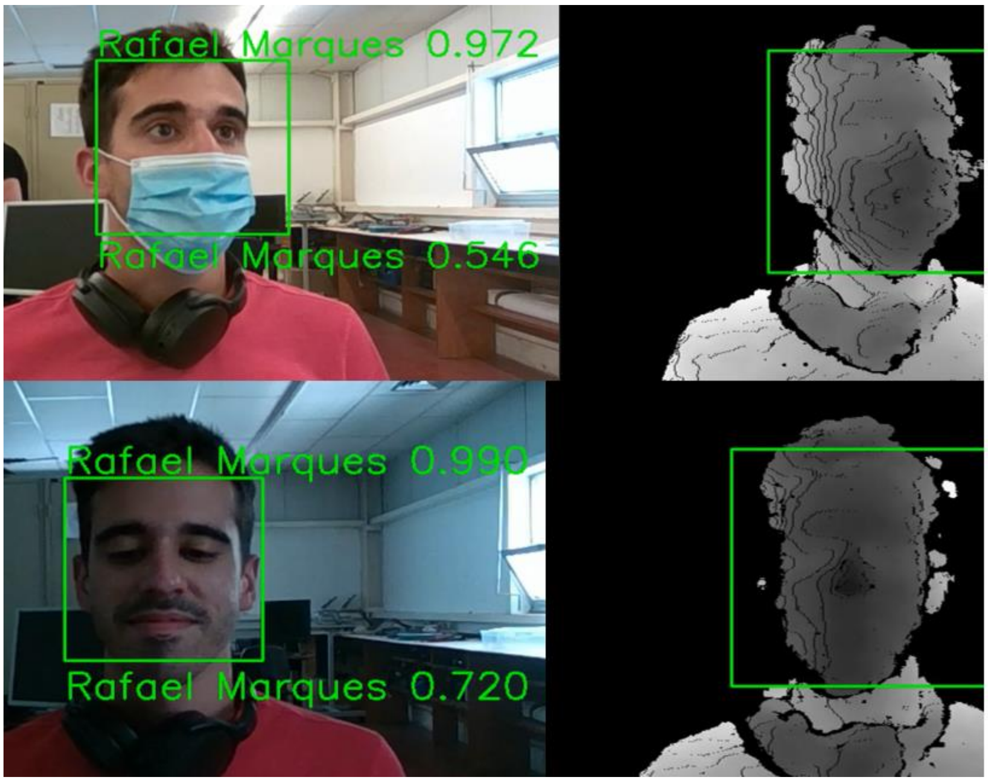


Figure 86- Detection and correct prediction of Rafael Marques.

Figure 87 portrays another atypical pose performed by Rafael Marques. It is important to remind that the model was not trained using images of subjects that were smiling or even wearing a mask. It also did not contain images with strange poses like the one in Figure 87. Images like this one serve to prove once again, that the models are quite robust to pose changes, besides also being robust to changes in lighting conditions.

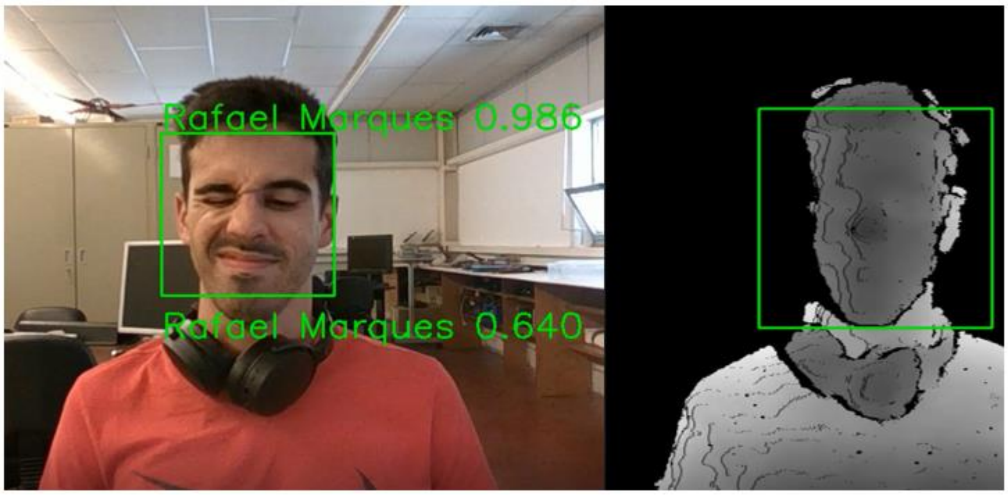


Figure 87- Strange pose illustration by Rafael Marques.

The final test carried out was to test if one could fool the system, by showing to the camera an image of a subject, instead of the real subject. As said many times in previous chapters, one of the main purposes of 3D face recognition was to implement additional security measures to prevent the system from being fooled, thus making it more reliable and robust. This was implemented in this environment through the use of the Haar Cascade classifier for depth images. This classifier was trained to detect depth faces and discard noisy images or any other irrelevant object present in the line of sight of the camera at a short distance. If a depth face is detected, that means that a "real" person is standing in front of the camera, thus it is possible to validate a legitimate person and not a printed image of a face. This is only possible thanks to depth since, a regular face has multiple depth measures, like camera to the nose, camera to eyes, etc., making the face a non-uniform surface. On the other hand, both a smartphone and a printed sheet of paper have a uniform and completely flat surface, i.e., regardless of their position, any of their points will have the same distance to the camera, meaning that only one measure of distance exists.

Supposing hypothetically that someone, an unknown person, somehow gained access to a picture of someone included in the dataset. If that stranger showed that picture to the camera, what would happen in that case was similar to the situation illustrated in Figure 88. Even though the subject on the image is in the dataset, it is not being predicted by the CNNs since the depth classifier does not detect a face in the first place. The reason for this, as previously said, is because the classifier is seeing noise, hence preventing the system from completing the prediction.

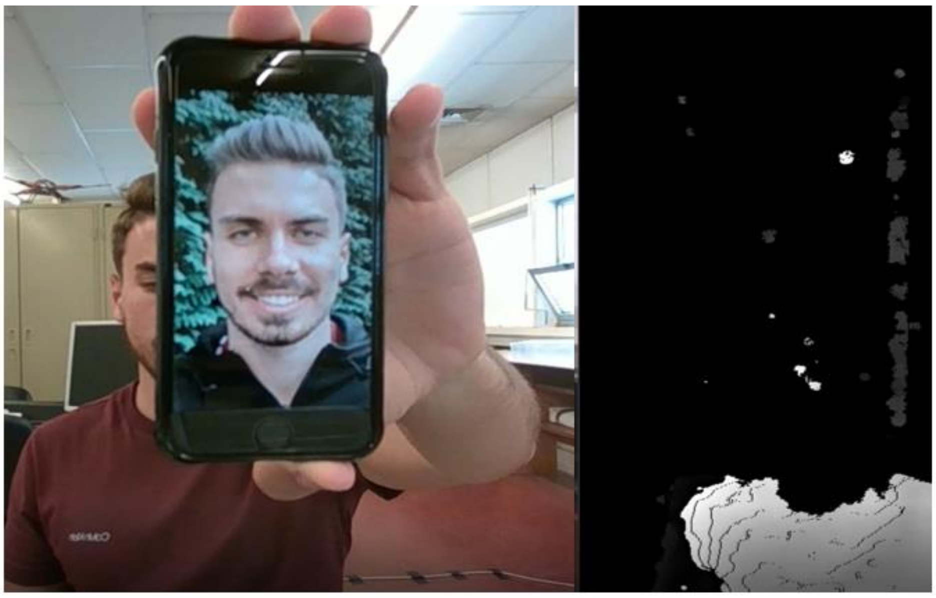


Figure 88- Attempt to fool the algorithm.

By removing the smartphone from the camera's viewing area, the algorithm immediately detects a depth face and an RGB face as before, and automatically does the predictions and outputs the prediction result on the screen.

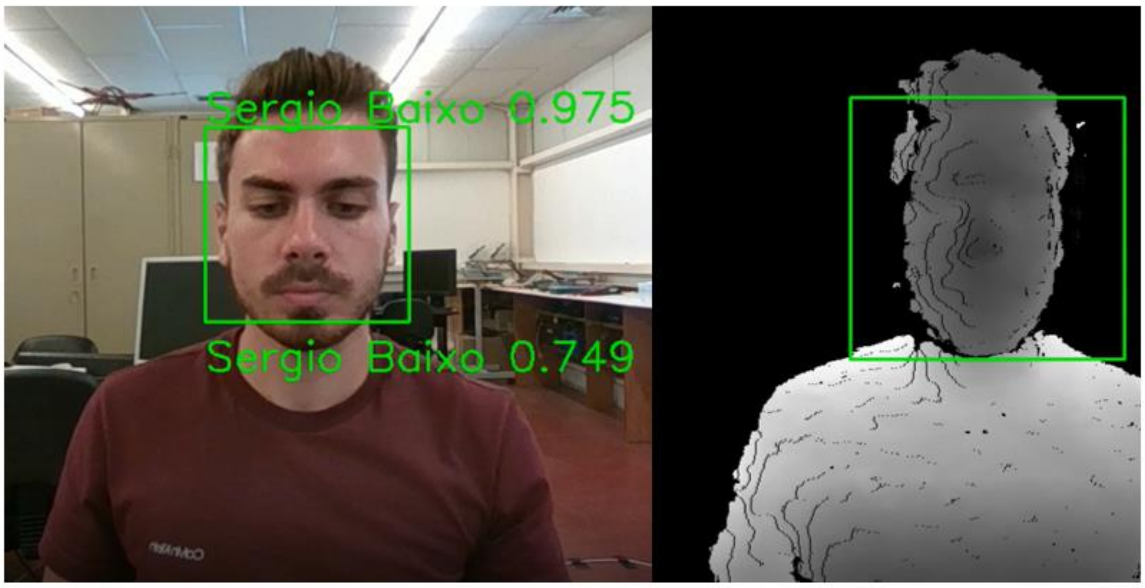


Figure 89- Successful detection after the attempt to fool the system.

If the stranger continues with the attempt to fool the system, the same as before happens, it detects noise and therefore, the algorithm will not proceed to make predictions.

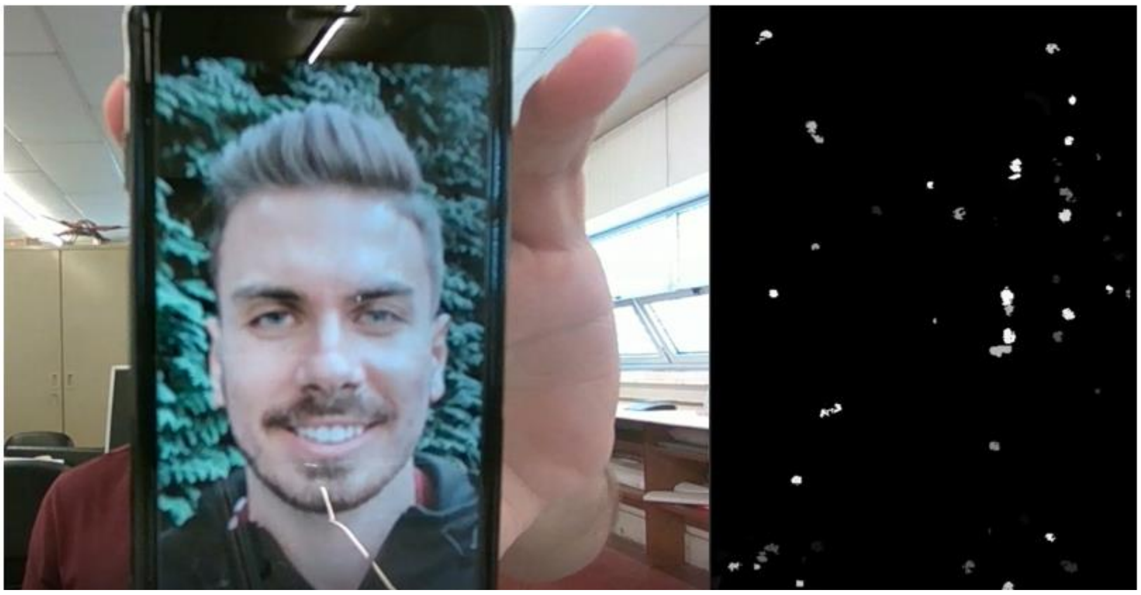


Figure 90- Second attempt to fool the algorithm.

Having this in mind, the conclusion here is that the depth classifier acts as a verifier or validation element since it always checks for the presence of a person before the CNNs start to make predictions.

It is important to mention the limitations associated with this system. Regarding depth, Table 2 refers to a maximum depth distance of 6m which would be fine if the intentions for this dissertation were to detect whole silhouettes, or pets, or even cars since they have unique shapes, but since one is working with faces, which contain several depth measures and small details, as previously mentioned, as the subject moves away from the camera, the camera will gradually lose its capability to acquire depth details and the whole face will look like an uniform blob. Also in Table 2, there is a hardware constraint regarding minimum distance in the Intel RealSense stipulating a minimum depth distance of 52 cm. This means that for distances between 2m and 6m, the depth model would be not feasible since it would not properly detect and recognize a depth face. On the other hand, the RGB model is perfectly capable of performing well, despite these distances, and will make predictions until the Haar Cascade classifier detects RGB faces, regardless of the distance between the camera and the face.

6. Conclusions and Further Work

The work presented throughout this dissertation was written with the intention to provide an accessible approach to the development of a 3D face recognition system. The methods implemented and detailed throughout this dissertation, allow to straightforwardly replicate a similar system, as long as the required hardware and software conditions are satisfied, thus allowing anyone, with or without any knowledge in Deep Learning concepts, to apply it to a personal or professional use case.

Regarding face detection, two major algorithms were addressed, being them Haar Cascade and Multi-task Cascaded Convolutional Networks (MTCNN). As for face recognition, one concluded that the most critical aspect to guarantee success in face recognition tasks is to ensure that a vast amount of data is available for training, in this case, images, since to have better performance on real-world scenarios, a model needs to learn, and it learns better as it contains more images. The problems associated with pose variation in 2D systems were approached, how 3D tried to solve these problems, and how this pose variation can heavily reduce the performance of a face recognition system.

The Methods and Methodologies section aimed to detail all the steps performed in order to produce results. One saw that different cameras can provide distinct results based on their specifications. Software capabilities have been more accessible on the Intel RealSense, since this camera has substantially more contributions from the developer's community. This seriously helped when dealing with depth and depth alignment, along with other features embedded with this camera, thus contributing to the fulfilment of this work. Besides this, one very important aspect associated with the dataset is the use of image augmentation. Here, image augmentation proved to be a valuable implementation, since it delivered more diverse data to both datasets, which consequently caused a positive impact on how later, the models performed in real-time prediction.

In the Tests and Results chapter, one mostly studied the impact of several CNN architectures during training. It was clear that the architecture built from scratch failed sorely, whilst the use of transfer learning previously trained using images of faces, revealed to be the ideal solution, according to the results portrayed in 5.6. These last results heavily relied on the fine-tuning of the hyperparameters, mainly the optimizer, learning rate, and most importantly, the architecture of the model itself. Regarding RGB images, excellent results were obtained in prediction, thanks mainly to the Inception-Resnet V1 architecture already pre-trained using face images from other people. As for depth images, the conclusion reached when training DL models with face depth maps, is that the results can easily differ since these images can be quite ambiguous. Even for humans, it is not obvious at first sight which depth face image belongs to which subject, therefore, one should not rely exclusively on depth face recognition especially if there are matters of security on the line, as these can be at risk.

All the images portrayed in 5.6 prove the reliability of the system developed, making it a well-suited system to eventually be implemented on medium-security facilities. As proven, the developed system is capable of precisely recognize subjects within the dataset, and classify, as unknown subjects, people that do not belong to the dataset. Besides this classification, the system has demonstrated the ability to detect when it is being fooled by a smartphone or a printed sheet of paper containing the face of a person, and thus, this 3D face recognition system can be considered as reliable and secure.

Regarding further work, the addition of some changes could be considered in order to make this system even more robust. Adding more subjects to the dataset, depending on the use case is a way to make the system more suitable to that same use case. However, one must bear in mind that the more subjects are added, the more it takes to complete training since the network has to absorb and deal with more data. Related to the process of adding subjects, there is also the chance to improve on the image capture algorithm, as this can be a bit tedious and extensive due to having various poses captured and since the subjects are required to stand still while obeying a pre-determined distance to the camera in order to maintain a uniform dataset. Besides the hyperparameters analysed in previous chapters, there are a lot more that one can modify to obtain more suitable networks to the datasets used. In this work, the focus was to have the proper balance between hyperparameter tweaking, speed of training, and overall accuracy values. The same can be said for the Haar Cascade depth classifier. Since it was trained only in 30 stages, as mentioned earlier, presumably if this number was set to 60, one could end up with double precision in making the face detection.

At the beginning of this dissertation, the goal of applying the functioning face recognition system to CHARMIE was proposed. This goal has not been accomplished yet, due only to the fact that CHARMIE has not been fully built yet. However, once the anthropomorphic robot finally gets assembled, it will have the capabilities of performing real-time face detection and recognition, taking advantage of this ability to perform certain tasks according to the person it recognizes, providing personal assistance in carrying out his/her daily tasks as well as providing numerous other life conveniences.

7. References

[1] D. Kragic and M. Vincze, "Vision for Robotics," *Found. Trends Robot.*, 2009.

[2] C. Sager, C. Janiesch, and P. Zschech, "A survey of image labelling for computer vision applications," 2021.

[3] I. Mihajlovic, "Computer Vision." https://towardsdatascience.com/everything-you-ever-wanted-to-know-about-computer-vision-heres-a-look-why-it-s-so-awesome-e8a58dfb641e (accessed Dec. 11, 2020).

[4] I. Goodfellow, Y. Bengio, and A. Courville, "Deep Learning." Accessed: Dec. 20, 2020. [Online]. Available: www.deeplearningbook.org.

[5] "Computer Vision Market Size & Share Report." https://www.grandviewresearch.com/industry-analysis/computer-vision-market (accessed Nov. 21, 2020).

[6] BitRefine, "Computer Vision Market by 2022." https://bitrefine.group/11-blog/120-establishing-your-brand-on-college-campuses (accessed Nov. 21, 2020).

[7] N. Zaeri, "3D Face Recognition," in *New Approaches to Characterization and Recognition of Faces*, InTech, 2011.

[8] Statista, "Smartphone users 2020." https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/ (accessed Jan. 09, 2021).

[9] J. Luo, F. Hu, and R. Wang, "3D Face Recognition Based on Deep Learning," in *Proceedings of 2019 IEEE International Conference on Mechatronics and Automation, ICMA 2019*, 2019, pp. 1576–1581.

[10] "RoboCup@Home." https://athome.robocup.org/ (accessed Dec. 09, 2020).

[11] A. Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly.

[12] J. Brownlee, "Supervised and Unsupervised Machine Learning Algorithms," *Machine Learning Mastery*. https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/ (accessed Dec. 09, 2020).

[13] IBM, "What is Supervised Learning." https://www.ibm.com/cloud/learn/supervised-learning (accessed Dec. 09, 2020).

[14] H. Hsu, "How Do Neural Network Systems Work?" https://computerhistory.org/blog/how-do-neural-network-systems-work/ (accessed Feb. 20, 2021).

[15] Y. Xiaozhou, "Convolutional Neural Network: How is it different from the other networks?," *Xiaozhou's Notes*. https://yangxiaozhou.github.io/data/2020/09/24/intro-to-cnn.html (accessed Mar. 05, 2021).

[16] S. Haykin *et al.*, *Neural Networks and Learning Machines Third Edition*. Pearson Prentice Hall, 2009.

[17] Pathmind, "A Beginner's Guide to Neural Networks and Deep Learning." https://wiki.pathmind.com/neural-network.

[18] Y. Lecun, E. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," 1998.

[19] D. H. Hubel and T. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," *J. Physiol.*, pp. 106–154, 1962.

[20] D. Rumelhart, G. Hinton, and R. Williams, "Learning Internal Representations by Error Propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*, MIT Press, 1987, p. 567.

[21] Y. Le Cun, "Generalization and Network Design Strategies," 1989.

[22] K. O'shea and R. Nash, "An Introduction to Convolutional Neural Networks."

[23] S. Saha, "A Comprehensive Guide to Convolutional Neural Networks," *Towards Data Science*. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (accessed Feb. 23, 2021).

[24] Belajar Pembelajaran Mesin Indonesia, "Convolutional Neural Networks (CNN)." https://indoml.com/2018/03/07/student-notes-convolutional-neural-networks-cnn-introduction/ (accessed Jan. 21, 2021).

[25] Deeplizard, "Zero Padding in Convolutional Neural Networks." https://deeplizard.com/learn/video/qSTv_m-KFk0 (accessed Jan. 15, 2021).

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks."

[27] O. Russakovsky *et al.*, "ImageNet Large Scale Visual Recognition Challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, 2015.

[28] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks For Large-Scale Image Recognition," 2015.

[29] C. Szegedy, V. Vanhoucke, S. Ioffe, and J. Shlens, "Rethinking the Inception Architecture for Computer Vision."

[30] C. Szegedy *et al.*, "Going Deeper with Convolutions."

[31] M. Shakirul Islam, A. Foysal, N. Neehal, E. Karim, and S. A. Hossain, "InceptB: A CNN Based Classification Approach for Recognizing Traditional Bengali Games," 2017.

[32] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition."

[33] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning."

[34] P. Viola and M. Jones, "Rapid Object Detection using a Boosted Cascade of Simple Features," 2001.

[35] K. Zhang, Z. Zhang, Z. Li, and Y. Qiao, "Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks," *IEEE Signal Process. Lett.*, pp. 1499–1503, 2016.

[36] Y. Taigman, M. Y. Marc', A. Ranzato, and L. Wolf, "DeepFace: Closing the Gap to Human-Level Performance in Face Verification."

[37] O. M. Parkhi, A. Vedaldi, and A. Zisserman, "Deep Face Recognition."

[38] F. Schroff and J. Philbin, "FaceNet: A Unified Embedding for Face Recognition and Clustering."

[39] T. Ahonen, A. Hadid, and M. Pietikäinen, “Face recognition with local binary patterns,” pp. 469–481, 2004.

[40] K. Simonyan, O. M. Parkhi, A. Vedaldi, and A. Zisserman, “Fisher Vector Faces in the Wild.”

[41] G. Sang, J. Li, and Q. Zhao, “Pose-Invariant Face Recognition via RGB-D Images,” 2016.

[42] B. Y. L. Li, A. S. Mian, W. Liu, and A. Krishna, “Using Kinect for face recognition under varying poses, expressions, illumination and disguise,” *Proc. IEEE Work. Appl. Comput. Vis.*, pp. 186–192, 2013.

[43] C. Ciaccio, L. Wen, and G. Guo, “Face recognition robust to head pose changes based on the RGB-D sensor,” *IEEE 6th Int. Conf. Biometrics Theory, Appl. Syst. BTAS 2013*, 2013.

[44] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards, “Artificial Intelligence A Modern Approach,” 1995.

[45] B. D. Ripley, *Pattern Recognition and Neural Networks*. 2011.

A. Appendix A

A1 Image Capture Procedure

For the Kinect camera, a static red square was constantly being drawn on the centre of each frame of the image. In contrast, another square, the green one, was constantly being drawn dynamically according to the position of the subject’s face. For this, it was used the Haar Cascade algorithm, already mentioned in 3.1.1. OpenCV has built-in functions to deal with .xml extension files, adding them as input to a cascade classifier function and simply returning four coordinates of the subject’s face when one is detected. An .xml file capable of detecting RGB faces was imported to OpenCV.

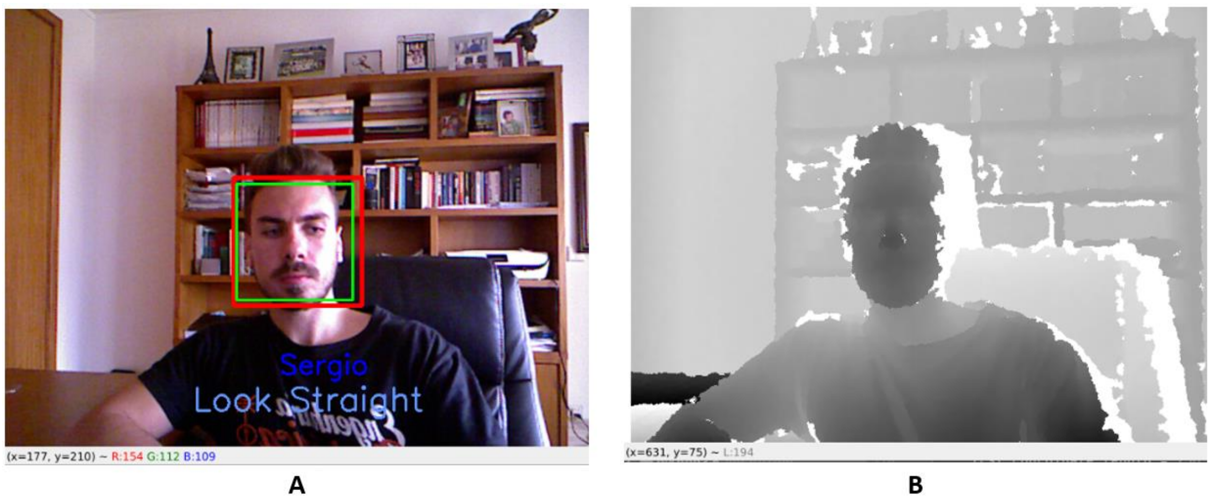


Figure 91- Image acquisition with indication to look straight. RGB (A), Depth (B).

Then, it is possible to draw a rectangle with these four coordinates. The subject must position himself in a way that the green square is contained inside the red one, making the borders of the squares almost coincident with each other. After this, a timer starts. When this timer ends, the exact frame that coincides with zero is the frame that is captured and stored in the created directory, both RGB and depth frames, in its respective sub-directory.

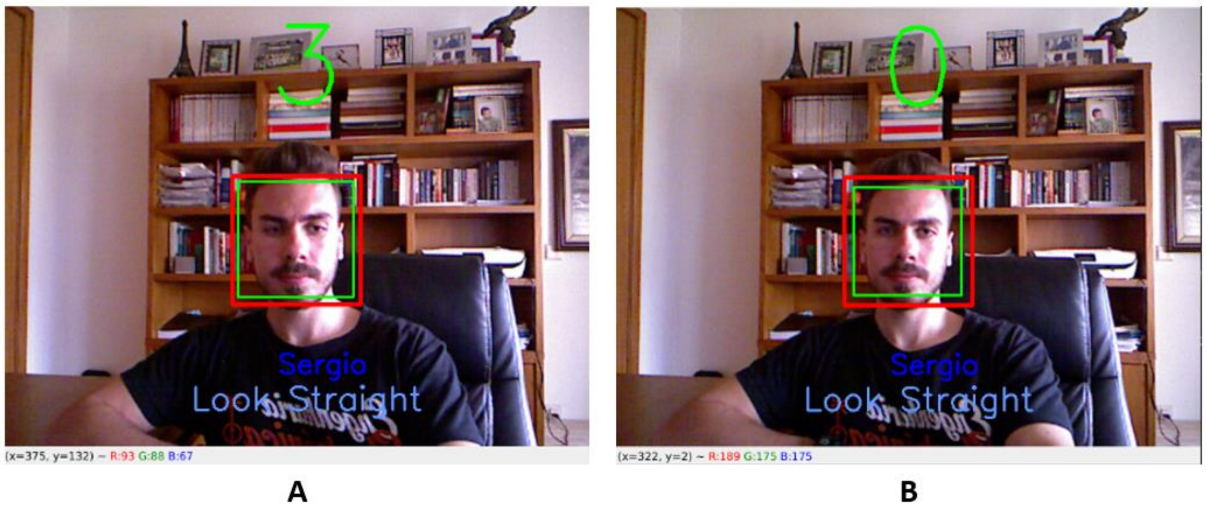

3
Sergio
Look Straight
(x=375, y=132) ~ R:93 G:88 B:67
A
0
Sergio
Look Straight
(x=322, y=2) ~ R:189 G:175 B:175
B

Figure 92- Iteration 1: Timer's countdown. Start (A). End (B).

Figure 92 shows an example of the timer performing. If the subject moves his head resulting in the green square being outside the red square, the timer stops and resets to zero. Once the subject is ready and centered with the green square inside the red one, the timer restarts. Also, if the green square has a small area, which in practice means that the subject is too far away from the camera, the timer does not start. Figure 93 shows a case where the timer does not start because the conditions were not satisfied.

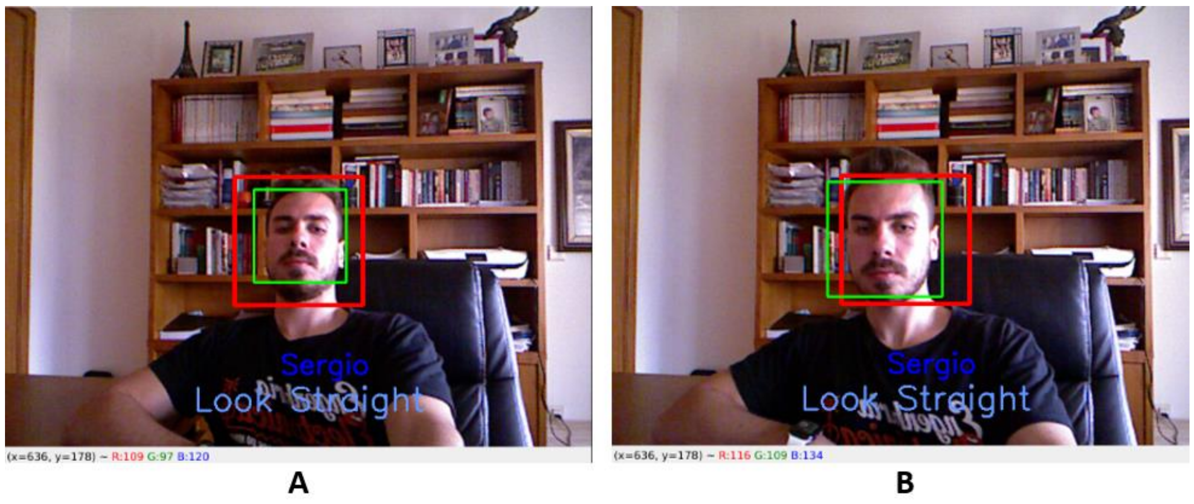

Sergio
Look Straight
(x=636, y=178) ~ R:109 G:97 B:120
A
Sergio
Look Straight
(x=636, y=178) ~ R:116 G:109 B:134
B

Figure 93- Situation where the conditions for the timer to run were not satisfied.

Iteration 1 assists the subjects to look straight to the camera, like in Figure 92. Once the conditions above are satisfied, the timer reaches zero, and the images are stored in the created folder, a second message appears on the screen. This time, the subject is told to look to the left (Figure 94-A). This is the beginning of iteration 2. And the same process as before repeats.

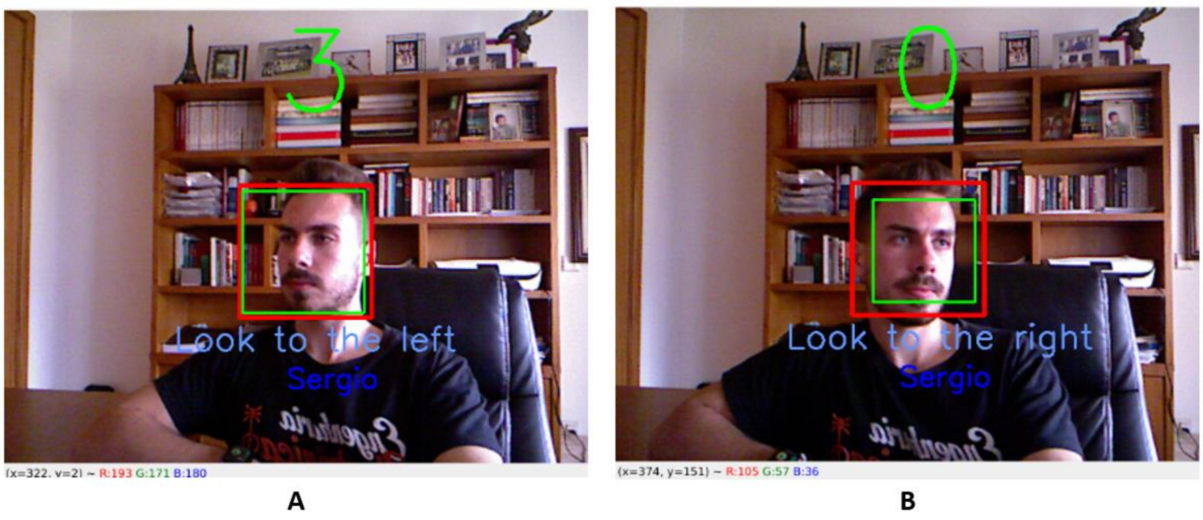

3
Look to the left
Sergio
(x=322, y=21) ~ R:193 G:171 B:180
A
0
Look to the right
Sergio
(x=374, y=151) ~ R:105 G:57 B:36
B

Figure 94- Iteration 2: Look to the left pose (A). Iteration 3: Look to the right pose (B).

The next iteration, the third one regards a look to the right pose (Figure 94-B). The process is the same as in previous iterations.

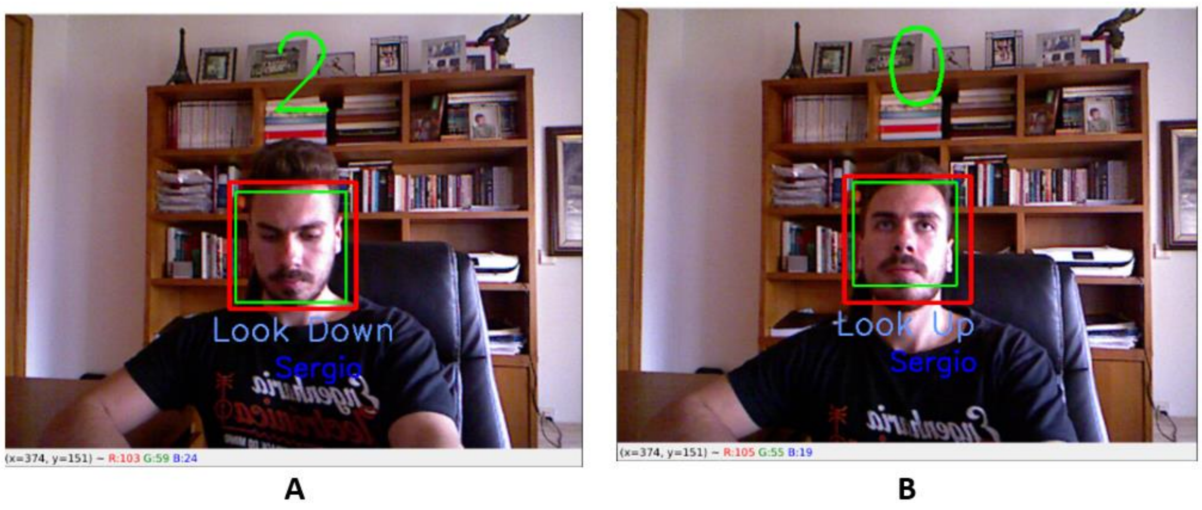

2
Look Down
Sergio
(x=374, y=151) ~ R:103 G:59 B:24
A
0
Look Up
Sergio
(x=374, y=151) ~ R:105 G:55 B:19
B

Figure 95- Iteration 4: Look down pose (A). Iteration 5: Look up pose (B).

Following the left and right-looking poses, the subject had to position himself/herself on a looking down pose and later in a looking up pose. The same process repeats once again. The frame captured is replicated creating multiple images. Having a lot of images is a typical procedure when building datasets, since it strongly helps to ensure good performances on neural network training.

Finally, the last two iterations are slightly different from the previous ones. This time, the subject is required primarily to align the face with the centre of the camera, at a certain distance from it, just like in the previous iterations of capture. After the subject is ready, he/she has to press and hold the keyboard space key. If that key is being pressed, the program continuously starts to acquire frames and stores them. Figure 96 shows an illustration of what is being described.

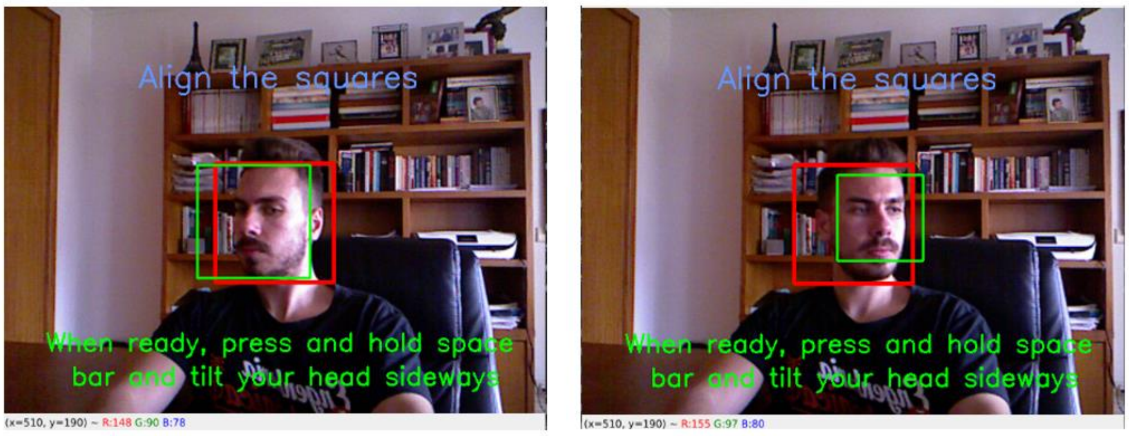

Align the squares
When ready, press and hold space bar and tilt your head sideways
(x=510, y=190) ~ R:148 G:90 B:78
Align the squares
When ready, press and hold space bar and tilt your head sideways
(x=510, y=190) ~ R:155 G:97 B:80

Figure 96- Iteration 6.

For these two iterations the timer is deactivated, and it is no longer required to align the squares inside each other. The only requirement is that the key is being pressed. In the sixth iteration, the subject has to tilt his/her head sideways, and in the seventh, has to tilt his/her head up and down. This approach clearly provides a wider variety of images to the dataset.

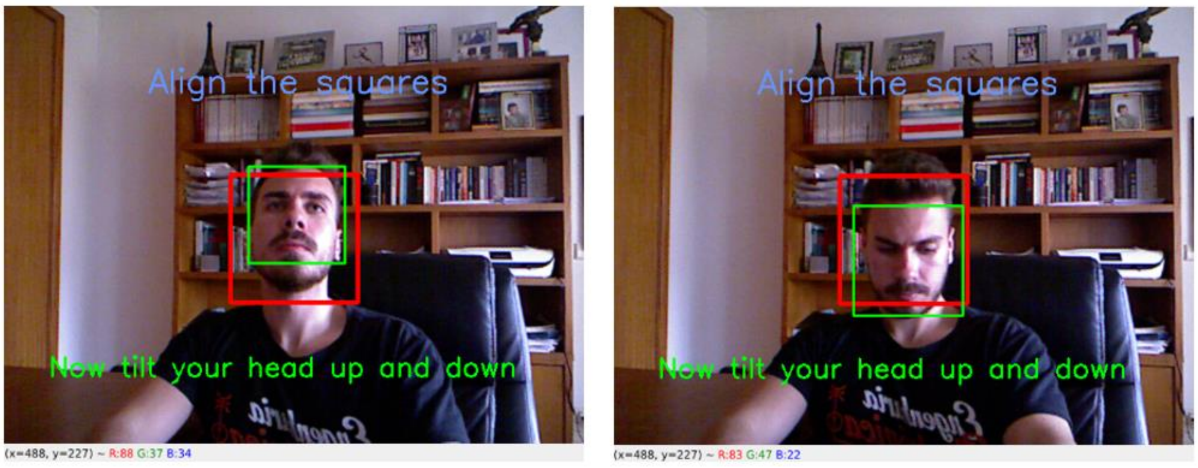

Align the squares
Now tilt your head up and down
(x=488, y=227) ~ R:88 G:37 B:34
Align the squares
Now tilt your head up and down
(x=488, y=227) ~ R:83 G:47 B:22

Figure 97- Iteration 7.

After a certain time, the seventh iteration comes to an end and the window closes. The image acquisition process for the Kinect camera is completed, and the captured images are stored in subject_name/depth or subject_name/rgb, depending on whether they are RGB or depth images. Below is the pseudocode that represents the iterations described above.

Algorithm 1- Image capture and storage

```
BEGIN
    Timer=True
    Set needed parameters
    If Timer
        Loop
            Draw red rectangle
            Draw green rectangle
            If green inside red and green rectangle area<red rectangle area
                Start timer
                    If the timer reaches zero
                        Capture RGB image
                        Capture Depth image
                        Store images in the directory
                        Reset timer
                        Increment iteration
                    If iteration is equal to 4
                        Timer=False

        End Loop
    If !Timer
        Loop
            Draw red rectangle
            Draw green rectangle
            Loop While green inside red and green rectangle area<red rectangle area

                Loop while the key is pressed and
                 number of images less or equal to total images/2
                    Capture RGB image
                    Capture Depth image
                    Store images in the directory
                    Increment number of images
                    If the number of images is equal to total images/2
                        Increment iteration
```

```
                                        If the number of images is equal to the total of images
                                   End Loop
                            End Loop
                     End Loop
END
```

The second algorithm used to execute on the Intel RealSense D455 was a lot simpler. It required the subject to only adjust the distance to the camera, without the need to align squares. The algorithm was possible to be simplified since the RealSense has the functionality to calculate the distance to a pre-established set of coordinates (x, y). Besides the use of this function, everything else remained the same.

Similar to the previous description, the program initializes with the dialog box to input the subject's name, like in . After entering the name, a directory is immediately created with the name of the subject, as well as two sub-directories inside it destined to hold RGB and depth images, separately.

The iterations of the capture with the RealSense, similarly to the iterations on the Kinect, have a timer. But, this time, the timer only starts decreasing when the subject positions himself/herself at 58 cm distance from the camera. This is required only to start the timer since it continues to decrement if the subject maintains the distance between 57 cm and 59 cm. Any value lower than 57 cm or higher than 59 cm will halt the timer and reset it. The procedure was made like this in order to ensure consistency in the quality of depth maps across all images of all subjects.

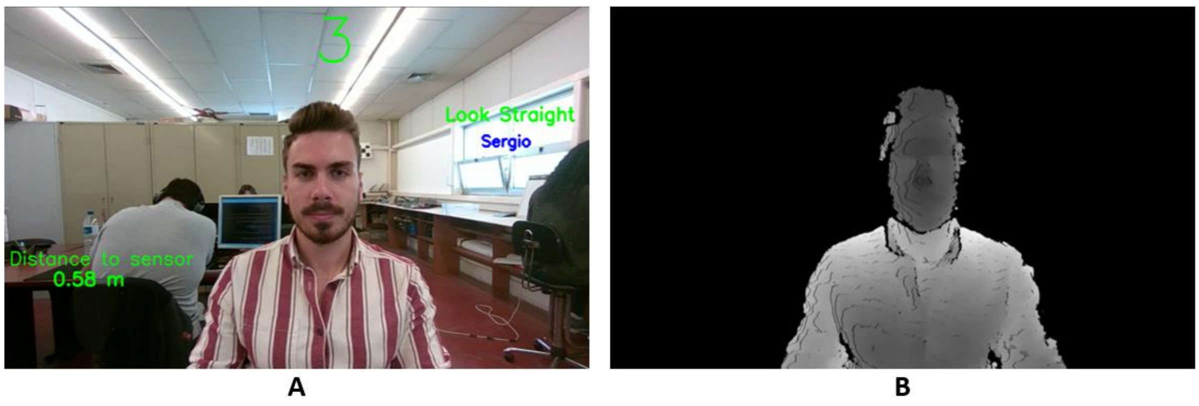


Figure 98- Iteration 1 of the RealSense training set. RGB (A), Depth (B).

As portrayed, the timer only appears when the distance to the camera/depth sensor is 58 cm. An important detail to notice is that both RGB and depth images are aligned. This was easily implemented only using the RealSense, since to tackle this misalignment in the Kinect camera, image processing had to be applied after the capture was complete. With the RealSense this alignment is made automatically, if enabled.

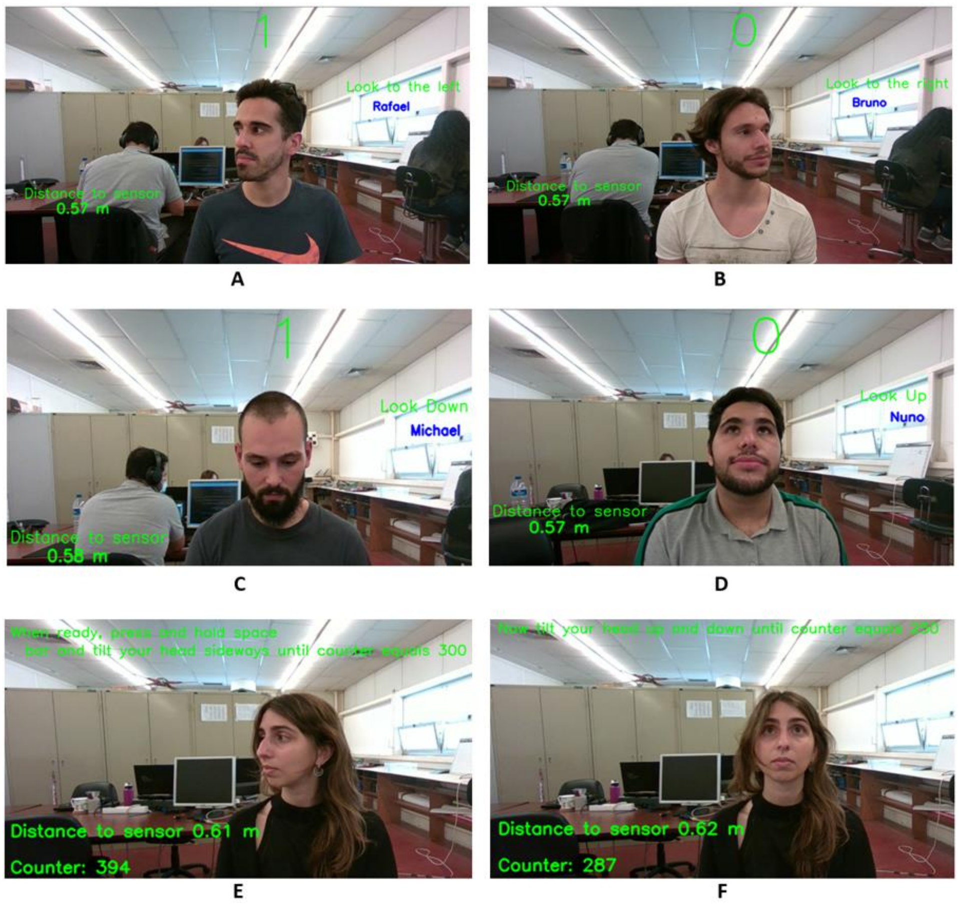


Figure 99- Iterations 2 to 7 of the RealSense training set.

In Figure 99 most of the iterations of the whole image capture process are portrayed. The last two iterations are similar to the two last ones in the Kinect sensor, but this time, the subject has visual feedback of how many images remain until the capture stops.