People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

University of M'Hamed BOUGARA – Boumerdes
Institute of Electrical and Electronic Engineering

Department of Electronics

Project Report presented in Partial Fulfillment of the Requirements of the Degree of

MASTER
In Electronics

Option: **Computer Engineering**

Title:

Face Mask Detection Using Convolutional Neural Networks and Haar Cascade Classifiers

Presented By:
BOUTHIBA Mohammed Ramzi
Supervisor:
Mr. A. MOHAMMED-SAHNOUN.

Registration Number: 161632002994

Dedication

This work is dedicated to my beloved mother whom I hope will get better soon.

To my roommate Djamel-Eddine, who has been a great friend during the past 5 years.

Acknowledgments

I would like to thank my fellow student Warda REZIG for her constant help and assistance throughout this project.

Abstract

This project aims to develop a system that relies on face mask detection. It can be used as a method to control access to buildings, offices or any closed facility or public gathering places that promote human interactions. The access control is achieved through the monitoring of the people entering a certain building through a camera and decide whether to grant access or not to the person wishing to enter. The decision is based on whether the person is wearing a mask or not.

The implementation of this system is made possible using two different machine learning techniques, namely: Convolutional Neural Networks and Haar cascade classifiers. The Haar cascade classifier is used to detect faces off frames captured from a video stream. The faces captured by the classifier are then fed to the CNN to classify whether the person is wearing a mask or not.

The CNN architectures used in this project are the MobileNetV2, EfficientNet-B0 and a small custom CNN. These models are evaluated and compared to each other using various metrics in order to pick the one that suits well this specific project.

Table of Contents

Dedication........ i

Acknowledgments........ ii

Abstract........ iii

List of Figures........ vi

List of Tables........ viii

List of Abbreviations........ ix

General Introduction........ 1

Chapter 1: Background and Related Work........ 3

- 1.1 Introduction........ 3
- 1.2 Machine Learning........ 3
- 1.3 Haar Cascade Classifier........ 4
  - 1.3.1 Haar Features........ 5
  - 1.3.2 Integral Image........ 6
  - 1.3.3 Boosting........ 7
  - 1.3.4 Cascading the Classifiers........ 13
- 1.4 Deep Learning........ 14
- 1.5 Artificial Neural Networks........ 15
- 1.6 Convolutional Neural Networks........ 18
  - 1.6.1 Types of CNN Layers........ 19
  - 1.6.2 Training a CNN........ 22
- 1.7 Transfer Learning........ 28
  - 1.7.1 Models Used........ 29
- 1.8 Related Work........ 32
- 1.9 Conclusion........ 33

Chapter 2: Models and the Dataset Used........ 34

- 2.1 Introduction........ 34
- 2.2 Dataset - RMFD........ 34
- 2.3 Preprocessing........ 36

2.4 Haar Cascade Classifier 37
2.5 Models....................37
2.5.1 MobileNetV2 37
2.5.2 EfficientNet-B0....................38
2.5.3 Custom Model....................39
2.6 Tools Used....................41
2.6.1 Python....................41
2.6.2 Keras....................41
2.6.4 Training Environment 41
2.6.5 Testing Environment and Device....................41
2.7 Conclusion 42
Chapter 3: Results and Discussion....................43
3.1 Introduction....................43
3.2 Evaluation Metrics....................43
3.2.1 Confusion Matrix 43
3.2.2 Average Accuracy....................44
3.2.3 Precision 44
3.2.4 Recall....................45
3.2.5 F1-score 45
3.3 Experimental Results 46
3.3.1 MobileNetV2 46
3.3.2 EfficientNet-B0....................48
3.3.3 Custom Model....................50
3.3.3.1 Speed Improvement....................52
3.3.4 Summary and Comparison....................55
3.3.5 Real-Time Detection and Classification....................55
3.4 Conclusion 57
General Conclusion and Future Work....................59
Bibliography

List of Figures

Fig. 1.1: Some machine learning algorithms [1]. 4
Fig. 1.2: Examples of Haar-like features [3]. 5
Fig. 1.3: calculation of the integral image [5]. 6
Fig. 1.4: Examples for Haar features applied to a window [4]. 7
Fig. 1.5: plot of the output weight α_t with respect to the error rate ϵ_t [5]. 9
Fig. 1.6: An illustration presenting the intuition behind the boosting 10
algorithm, consisting of the parallel learners and weighted dataset [6]. 10
Fig. 1.7: plot of the function e^{-x}. 11
Fig. 1.8: AdaBoost algorithm [7]. 12
Fig. 1.9: The overall form of the detection process [4]. 13
Fig. 1.10: Relationship between AI, ML and Deep Learning [8]. 15
Fig. 1.11: Biological neuron [10]. 16
Fig. 1.12: Single artificial neuron (node) with n+1 inputs [9]. 16
Fig. 1.13: A simple DNN structure [9]. 17
Fig. 1.14: A simple convolutional neural network [12]. 18
Fig. 1.15: Convolutional layer with one filter [12]. 20
Fig. 1.16: Max-Pooling and Average Pooling operations [12]. 21
Fig. 1.17: Fully connected layers [14]. 21
Fig. 1.18: summary of the properties of each type of layer [14]. 22
Fig. 1.19: Summary of the ReLU activation function and its variants [14]. 23
Fig. 1.20: Bias-variance trade-off [12]. 28
Fig. 1.21: Transfer learning performance comparison graph [15]. 29
Fig. 1.22: Depthwise separable convolution. 30
Fig. 1.23: Bottleneck Block. 31
Fig. 1.24: Model proposed by paper [21] authors. 32
Fig. 2.1: Masked and normal faces examples from RMFD. 35
Fig. 2.2: Classes distribution after balancing the dataset. 35
Fig. 2.3: MobileNetV2 architecture. 38
Fig. 2.4: EfficientNet-B0 architecture. 39

Fig. 2.5: Custom model architecture...40
Fig. 3.1: Two classes confusion matrix [20]. 43
Fig. 3.2: MobileNetV2 loss and accuracy curves. 47
Fig. 3.3: MobileNetV2 confusion matrix. 48
Fig. 3.4: Efficientnet-B0 loss and accuracy curves. 49
Fig. 3.5: Efficientnet-B0 confusion matrix. 50
Fig. 3.6: Custom model loss and accuracy curves. 51
Fig. 3.7: Custom model confusion matrix...52
Fig. 3.8: Improved custom model loss and accuracy curves...53
Fig. 3.9: Improved custom model confusion matrix..54
Fig. 3.9: Real-time classification flowchart. 56

List of Tables

2.1 Number of samples in each set……..36
3.1 MobileNetV2 hyperparameters……..46
3.2 MobileNetV2 testing results……..47
3.3 MobileNetV2 classification report……48
3.4 Efficientnet-B0 testing results……49
3.5 Efficientnet-B0 classification report……..50
3.6 Custom model testing results…….51
3.7 Custom model classification report…...51
3.8 Improved custom model testing results…….53
3.9 Improved custom model classification report……54
3.10 Summary and comparison of the models…...55
3.11 Real-time execution time of the models…...57

List of Abbreviations

Adam	Adaptive Moment Estimation
ANN	Artificial Neural Network
AI	Artificial Intelligence
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DNN	Deep Neural Network
FC	Fully Connected
GPU	Graphics Processing Unit
ML	Machine Learning
MFDD	Masked Face Detection Dataset
NN	Neural Network
NERCMS	National Engineering Research Center for Multimedia Software
OpenCV	Open Computer Vision
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RMFD	Real-world Masked Face Dataset
SBC	Single Board Computer
SGD	Stochastic Gradient Descent
SMFD	Simulated Masked Face Dataset
VRAM	Video Random Access Memory

General Introduction

During the past year and a half, the world has been suffering from an outbreak of a deadly virus, namely COVID-19. This virus is propagated mainly through human interactions. Millions of people around the world have been affected by this virus. The fact that this virus has a significant death rate especially amongst elderly people, it led the World Health Organization to provide sanitary instructions and safety measures to prevent infections and hinder the spread of the virus.

Many governments around the world have put laws and policies regarding safety measures during this pandemic to contain the spread of the virus. For example, imposing on every citizen to wear face masks when going out in public, when taking public transportation and in the workplace. However, certain people fail to abide by these new laws and policies hence exposing their relative and colleagues to this virus which is a serious problem with dangerous consequences.

The motivation behind this project is to provide a system to control access of people to areas where the spread of the COVID-19 virus is highly probable to spread (e.g., a train station). The decision on whether to allow or deny access is based on whether the person whishing to access that area is wearing a face mask or not.

To develop this system, we had to use two main machine learning approaches: Haar cascade classifiers and Convolutional Neural Networks. Since it is intended to be installed at facilities or building entries, this system is designed to run on low computational power devices such as single board computers. The SBC used in this project is a Raspberry Pi 3 Model B which is used to test the performance of the different model used throughout this project.

The first chapter covers the theoretical background of the machine learning techniques used and the math behind them. We also cover why a Haar cascade classifier is used for the task of detection and why convolutional neural networks are used for the task of classification. We briefly discuss the similarities and the differences between the two techniques and the related work done in the context of this project.

In this second chapter we present the methods and steps followed to carry out the experiments in this project. We first cover the dataset that we used, then we discuss the proposed CNN architectures and the software tools used.

For the last chapter, three different CNN models are presented and discussed: MobileNetV2, Efficientnet-B0 and a custom-built CNN model. These three models are tested and compared to each other in terms of accuracy in classifying images into two classes: "person wearing a mask" and "person not wearing a mask". We also consider the execution time metric since it is an important aspect given the nature of this project.

Finally, we give a general conclusion about the results obtained and how they provide a useful insight in designing CNNs that are intended to run on low computational devices. we also gave suggestion on potential improvements for the systems.

Chapter 1: Background and Related Work

1.1 Introduction

In this chapter we are going to give an overview on deep learning, convolutional neural networks, and Haar Cascade classifiers. We are also going to cover one of the related works done for a similar project.

1.2 Machine Learning

A computer program is said to be learning when its performance on a given task improves with experience. Machine learning algorithms are applied to training data in order to derive patterns and make accurate predictions on new data through trial and error without being explicitly programmed to do so.

There are two ubiquitous types of machine learning algorithms: supervised learning and unsupervised learning.

- **Supervised learning:** The algorithm is exposed to a dataset containing labeled examples, i.e., for each input, there is a corresponding desired output. Using these labeled data, the algorithm learns to predict the correct output of unseen data with great accuracy.

  Supervised learning is mostly used to solve two types of problems: classification problems and regression problems. Classification is further discussed later since it’s our main topic in this project.

- **Unsupervised learning:** Unlike supervised learning, in this case the algorithm is presented with unlabeled data. This type of learning is used when we want to discover patterns in unstructured data.

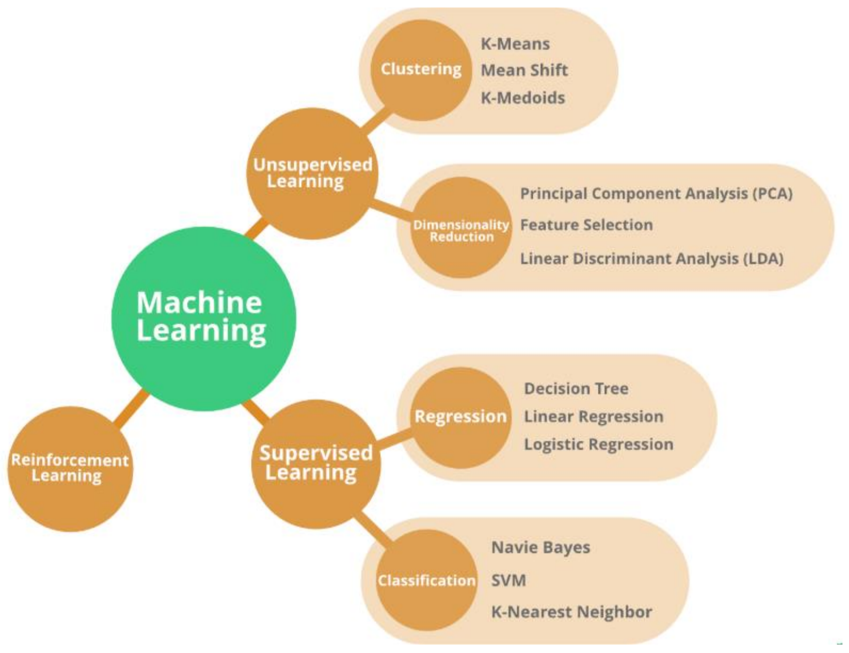


Fig. 1.1: *Some machine learning algorithms [1].*

The first machine learning algorithm that we are going to tackle in this project is the Haar Cascade algorithm.

1.3 Haar Cascade Classifier

Cascading classifiers is a method used to concatenate multiple weak classifiers to obtain at the end a strong classifier. These classifiers are trained on a few hundred samples of positive images (images that we want to detect) and a few hundred of other images that do not contain positive images.

Before cascading the weak classifiers, we need to select them first. A great way to select a small set of classifiers is by using a Boosting technique. Adaptative Boosting or AdaBoost is a meta-algorithm usually used in conjuncture with other learning algorithms. This meta-algorithm tries to solve the problem of constructing a single strong learner from a set of weak learners. A weak learner is defined to be a classifier that is only slightly correlated with the true classification (it can label examples better than random guessing). In contrast, a strong learner is a classifier that is arbitrarily well-correlated with the true classification [2].

1.3.1 Haar Features

Haar-like features are digital image features used in object recognition. These features were used in the first real-time face detector which this project is inspired from.

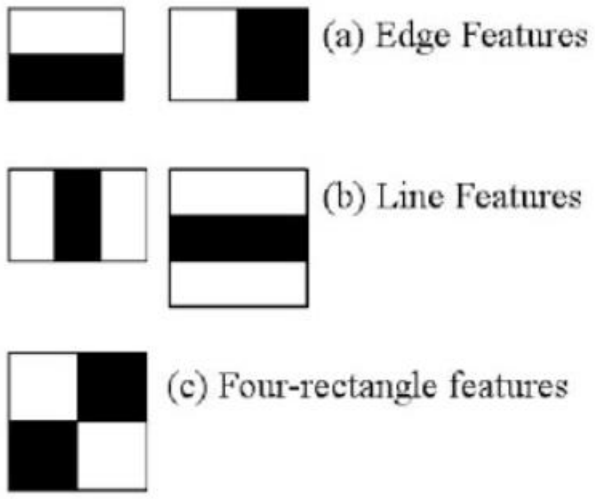


Fig. 1.2: *Examples of Haar-like features [3].*

Before the introduction of Haar features, practitioners used to do image processing with only the pixel values of that image made the task of feature calculation computationally expensive. Paul Viola and Michael Jones [4] adapted the idea of using Haar features. A Haar feature considers adjacent rectangular regions at a specific location in a detection window, sums up the pixel intensities in each region and calculates the difference between these sums. This calculated difference is then used to make a prediction along with multiple other Haar calculations.

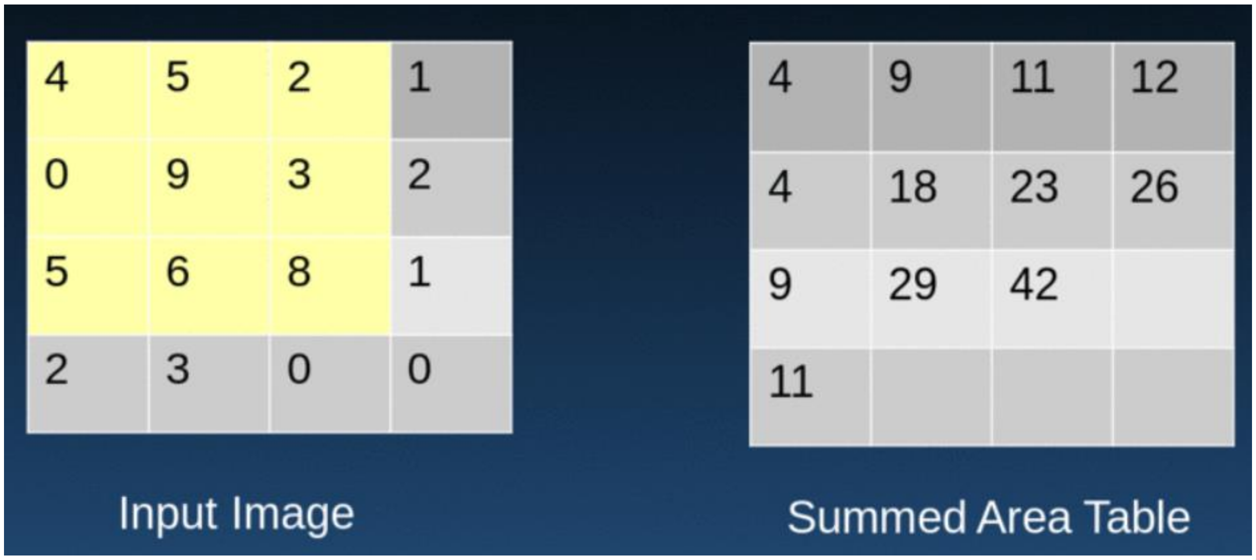
For example, with a human face, it is a common observation that among all faces the region of the eyes is darker than the region of the cheeks. Therefore, a common Haar feature for face detection is a set of two adjacent rectangles that lie above the eye and the cheek region. The position of these rectangles (Haar features) is defined relative to a detection window that acts like a bounding box to the target object (the face in this case).

To calculate the features on an image using the Haar kernels considering all the possible sizes of the kernels and all the possible locations at where they could be computed on the image would be a lot of computations to do. Even a 24×24 window would result in over 160,000 features.

Keeping in mind that we will slide many windows through our image to detect potential objects, that is a huge computational cost that we just cannot afford since our main objective is for our system to be computationally friendly. One of the contributions of Paul Viola and Michael Jones in their 2001 paper cited earlier was to use a concept called integral images.

1.3.2 Integral Image

The integral image also known as the summed area table is an image representation that allows very fast Haar feature evaluation. Once computed, any one of these Haar-like features can be computed at any scale or location at constant time.



Input Image

4	5	2	1
0	9	3	2
5	6	8	1
2	3	0	0

Summed Area Table

4	9	11	12
4	18	23	26
9	29	42	
11			

Fig. 1.3: *calculation of the integral image [5].*

It is clear from Figure 1.3 that the value of the pixels at any location in the integral image is the sum of the pixels in the original image that are in the area defined by the top left corner and the pixel coordinates that we want to calculate its value in the integral image.

The integral image can be calculated in a single pass through the image starting from the top left corner. Once the integral image is calculated, summing the pixel intensities over any rectangular area in the image takes only four addition operations regardless of the size of the rectangle.

1.3.3 Boosting

As stated before, a window as small as 24×24 could result in over 160,000 Haar features. Even though each feature can be computed efficiently since we introduced the concept of integral images, computing every possible Haar feature is computationally expensive for low-power CPUs.

There are many combinations of Haar features. However, we need to pick only some of them which are relevant to our problem. In our case, we need to detect faces, thus we need features which are relevant to detect face features such as eyes, mouths etc. Figure 1.4 below, depicts two useful features in detecting faces. One of them detects the eye region, since the area below the eyes tends to be brighter than the area of the eyes themselves, thus the sum of the difference between these two areas will generally be a net positive which promotes that the candidate sub-window contains a face.

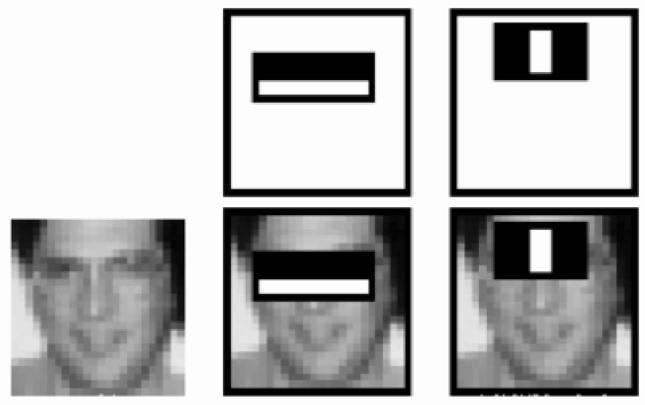


Fig. 1.4: *Examples for Haar features applied to a window [4].*

A solution to select the best Haar features out of +160,000 features is by using a variant of a boosting algorithm such as AdaBoost (Adaptive Boosting).

AdaBoost is a popular boosting technique which helps you construct a linear combination of multiple weighted "weak classifiers" into a single "strong classifier". Every Haar features is considered a weak classifier because, on its own, it cannot determine with great confidence that the sub-window contains a face.

AdaBoost can be applied to any classification algorithm, so it's considered as a meta-algorithm or a technique that builds on top of other classifiers as opposed to being a classifier itself [5]. The output (the strong classifier) of AdaBoost is represented in the equation below:

$$H(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right) \tag{1.1}$$

Where:

- $h_t(x)$ is the prediction made by the weak classifier *t* on the input *x*
- $\alpha_t$is the weight assigned by the AdaBoost algorithm to the classifier *t*
- The final output $H(x)$ is the sign of the linear combination of *T* weak classifiers.

1.3.3.1 Training

The classifiers are trained one at a time. After each classifier is trained, we update the vector D_t which represents a probability distribution of each of the training example appearing in the training set for the next classifier.

The first classifier ($t = 1$) is trained with equal probability given to all training examples. After it's trained, we compute the output weight (alpha) for that classifier [5].

$$\alpha_t = \frac{1}{2} ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right) \tag{1.2}$$

Where:

- The error rate ϵ which is just the number of misclassifications over the training set divided by the training set size.
- ln is the logarithmic function.

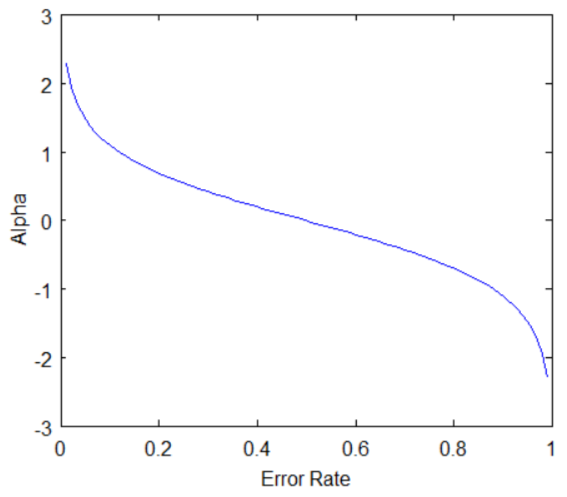


***Fig. 1.5:** plot of the output weight α_t with respect to the error rate ϵ_t [5].*

From the graph we can deduce three key pieces of intuition:

1. The classifier weight increases exponentially as the error rate approaches 0. We can deduce that better classifiers are given a greater weight, therefor it has more influence on the linear combination.

2. When the error rate is 0.5 the classifier's weight is 0. A classifier with 50% accuracy is no better than random guessing so we discard it.

3. We give a negative weight to classifiers with worse than 50% accuracy. The classifier weight grows exponentially negative as the error approaches 1. This means that we want to take the opposite answer of what the classifier suggested.

After computing the weight for the first classifier, we update the training example's probability distribution vector D_t using the following formula:

$$D_{t+1}(i) = \frac{D_t(i)e^{-\alpha_t y_i h_t(x_i)}}{Z_t} \tag{1.3}$$

Where:

- $D_t(i)$ is the old probability of the training example i
- $D_{t+1}(i)$ is the new probability of the training example i
- α_t is the weight of classifier t
- y_i is the ground truth label of the training example i
- $h_t(x_i)$ is the prediction made by classifier t on the training example i

$$Z_t = \sum_{i=1}^{I} D_t(i) \tag{1.4}$$

Where:

- I is the total number of training examples.

D_{t+1} carries a weight for each example in the training set that represents that example's probability to be selected in the next classifier's training round. As mentioned previously, the first classifier is trained with the training data probability being equally distributed.

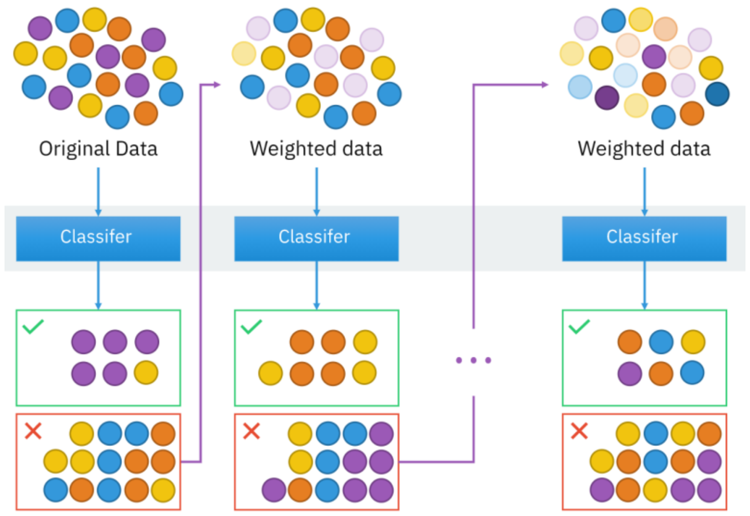


Fig. 1.6: *An illustration presenting the intuition behind the boosting algorithm, consisting of the parallel learners and weighted dataset [6].*

The above-mentioned equation needs to be evaluated for each of the training sample x_i and its corresponding label y_i. Each sample's weight from the previous training round is going to be increased or decreased by this exponential term. To understand how this exponential term behaves, let's look first at how the function e^x behaves [5].

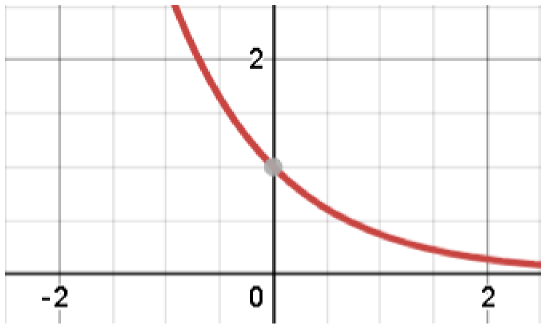


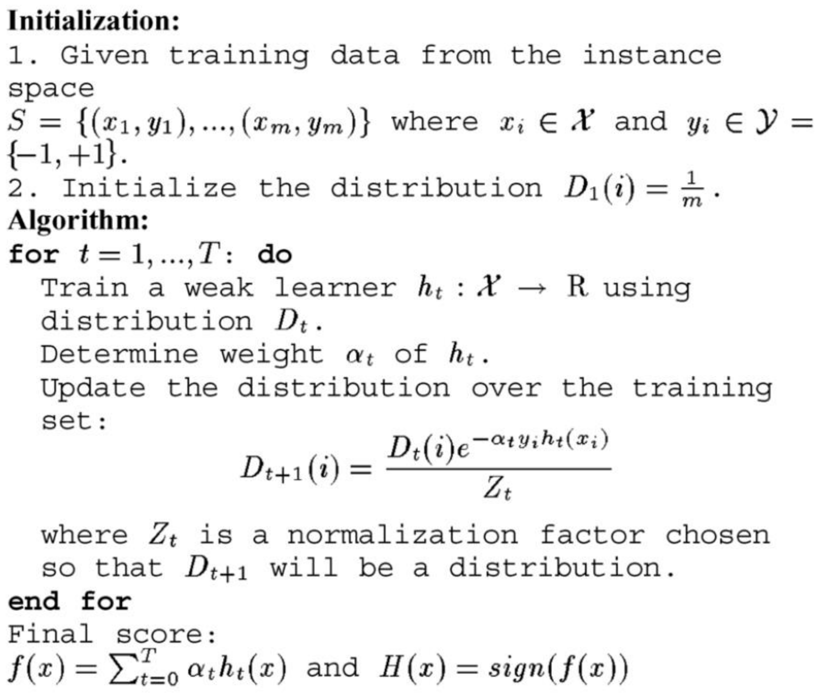
Fig. 1.7: *plot of the function e^{-x}.*

The function e^{-x} will return a fraction for positive values of x, and a value greater than one for negative input values. Thus, the weight for each training sample will be either increased or decreased according to the final sign of the term $-\alpha_t y_i h_t(x_i)$.

For binary classifiers whose output is constrained to either -1 or +1, the terms y_i and $h_t(x_i)$ only contribute to the sign and not the magnitude.

The term y_i represents the correct output for the corresponding training example, and $h_t(x_i)$ is the predicted output by classifier t on this training example. If the predicted and actual output agree, $y_i h_t(x_i)$ will always be +1. If they disagree, $y_i h_t(x_i)$ will be -1.

From the equation we can deduce that if classifier with a positive α coefficient misclassifies a sample x_i, this sample will be given greater weight so that it can be included in the next round of evaluations. Likewise. If a weak classifier misclassifies an input, the consequences are less strict.



```
Initialization:
1. Given training data from the instance
space
$S = \{(x_1, y_1), ..., (x_m, y_m)\}$ where $x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y} =$
$\{-1, +1\}$.
2. Initialize the distribution $D_1(i) = \frac{1}{m}$.
Algorithm:
for $t = 1, ..., T$: do
  Train a weak learner $h_t : \mathcal{X} \rightarrow$ R using
  distribution $D_t$.
  Determine weight $\alpha_t$ of $h_t$.
  Update the distribution over the training
  set:
        $D_{t+1}(i) = \frac{D_t(i)e^{-\alpha_t y_i h_t(x_i)}}{Z_t}$
  where $Z_t$ is a normalization factor chosen
  so that $D_{t+1}$ will be a distribution.
end for
Final score:
$f(x) = \sum_{t=0}^{T} \alpha_t h_t(x)$ and $H(x) = sign(f(x))$
```

Fig. 1.8: *AdaBoost algorithm [7].*

1.3.3.2 Using AdaBoost to Select the Features

As discussed above, in its original form, the AdaBoost learning algorithm is used to only boost the classification performance of weak classifiers. However, in the paper cited earlier [4], the authors used a variant of the AdaBoost both to select a small set of features and to train the classifier.

For this, we apply every feature on all the training images. For each feature, it finds the best threshold which will classify the faces to positive and negative. Obviously, there will be errors or misclassifications. We select the features with minimum error rate, which means they are the features that most accurately classify the face and non-face images. Then we simply add the selected feature to our subset of selected features [3].

Paper [4] says even 200 features provide detection with 95% accuracy. Their final setup had around 6000 features. That is a reduction from +160,000 features to only 6000 features.

However, in an image, most of the image is not a face. One needs to find a way to discard non-promising areas and only focus most of the computation time on promising sub-windows. The authors tackled that problem by coming up with a great contribution which is cascading the classifiers.

1.3.4 Cascading the Classifiers

In order to improve the efficiency of the classifiers, one method is to arrange them in a cascade manner. The idea is to place the most efficient classifiers at the beginning of the cascade so that we can reject most negative sub-windows while keeping almost all potential positive sub-windows. As the cascade goes on, the classifiers get more and more complex allowing to extract with great accuracy the relevant features that allow to make the final decision on the classification.

A positive result from a classifier at any point of the cascade triggers the evaluation of the classifier after it which is usually a more complex one. This process is followed throughout the whole cascade architecture. A negative result from any classifier at any point would result in discarding that whole sub-window.

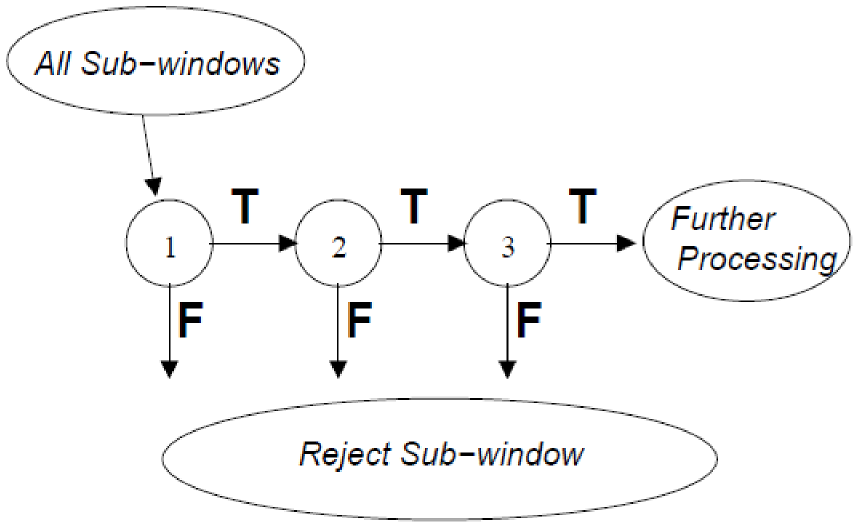


Fig. 1.9: *The overall form of the detection process [4].*

As an example, if we wanted to make an excellent first stage classifier, we would pick one that has the highest true positives. This allows to keep all potential positive sub-windows and not risk throwing away true positive in the early stages of the classifier. First stage classifiers have the luxury of having a relatively high false positive rate since it will be taken care of in the following stages of the cascade.

When training a cascade, we have to make a trade-off between speed and accuracy. Classifiers with more features tend to make better predictions, however, they are slower to run. In the training step we generally have to optimize the quantities: the number of classifiers in each stage, the number of features in each stage and the threshold. The lower the threshold of the stage the lower the false negatives are in that stage.

These three quantities are traded off in order to minimize the expected number of evaluated features. Finding the best trade-off is a very hard problem; thus, in practice, each stage in the cascade reduces the false positive rate and decreases the detection rate. Target values for the minimum reduction of the false positive rate maximum decrease in detection are set in the beginning of the training and the training stops when these values are met.

The complete face detection cascade presented in paper [4] has 38 stages with over 6000 features. Nevertheless, the cascade structure results in fast average detection times. On a difficult dataset, containing 507 faces and 75 million sub-windows, faces are detected using an average of 10 feature evaluations per sub-window. That is a decrease from having to evaluate 6000+ features for each sub-window to only about an average of 10 features.

1.4 Deep Learning

Deep learning is subset of machine learning which is an approach to achieve artificial intelligence and attempt to mimic the human brain.

Neural networks make up the backbone of deep learning models. Basically, any neural network with more than three layers is considered a deep neural network. Theoretically, the more layers a neural network has, the better its performance will be; since the additional layers will only optimize and refine its accuracy.

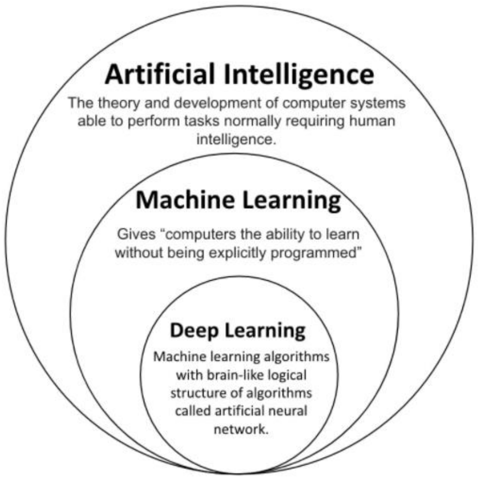

Artificial Intelligence
The theory and development of computer systems able to perform tasks normally requiring human intelligence.
Machine Learning
Gives "computers the ability to learn without being explicitly programmed"
Deep Learning
Machine learning algorithms with brain-like logical structure of algorithms called artificial neural network.

Fig. 1.10: *Relationship between AI, ML and Deep Learning [8].*

Deep learning has been taking off since the last decade. This is due to three main components: the availability of huge amounts of data, a lot of algorithmic innovations that made neural networks run much faster along with the availability of huge computational powers.

1.5 Artificial Neural Networks

ANNs are a ML model inspired by the biological nervous systems that are composed of interconnected neurons, hence the name neural networks. Each biological neuron is made of dendrites, an axon, and synapses. The neuron collects the input signals from its dendrites and produces spikes of electrical activity to be transmitted via the axon which is divided into thousands of branches. At the end of each branch, a structure called a synapse connects it to the dendrite of another neuron. Synapses mainly amplify or inhibit the signal coming out of the axon depending on the signals fed from the dendrites [9].

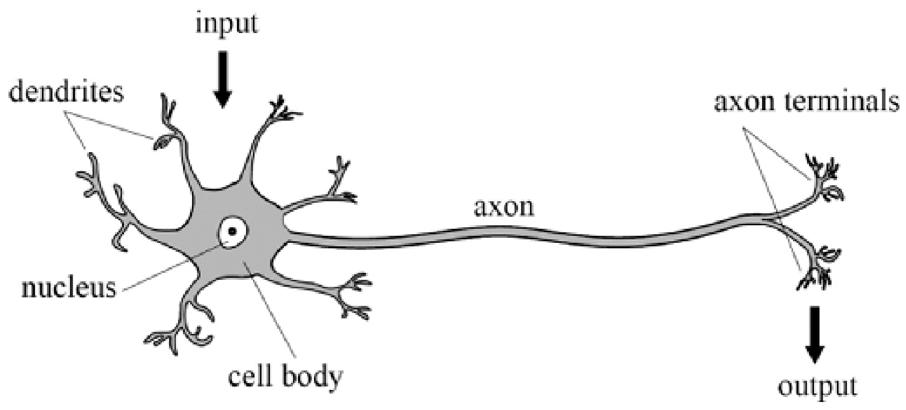


Fig. 1.11: *Biological neuron [10].*

Similarly, ANNs are also made up of a set of interconnected neurons. Neurons are the simplest processing units that make up an ANN. Each neuron takes a set of inputs (e.g., word frequency), each input x_n is associated with a synaptic weight w_n. The weights are multiplied by the corresponding inputs and accumulated together then biased, as depicted in figure 1.11. The result is used to calculate the output of the activation function as shown in equation below [9].

$$a = g\left(\sum_{n=1}^{N} x_n . w_n + b\right) \tag{1.5}$$

Where g is the activation function and N is the number of inputs.

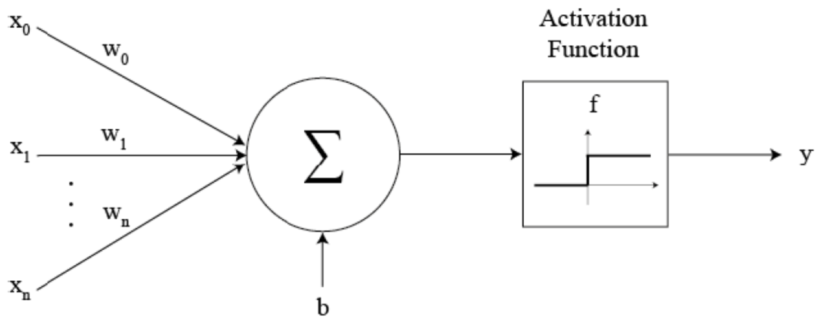


Fig. 1.12: *Single artificial neuron (node) with n+1 inputs [9].*

Typical ANNs are made up of three groups of successive layers: an input layer, hidden layer, and output layer – this type of neural network is referred to as Deep Neural Networks (DNNs). An example of a simple DNN is shown in figure 1.12.

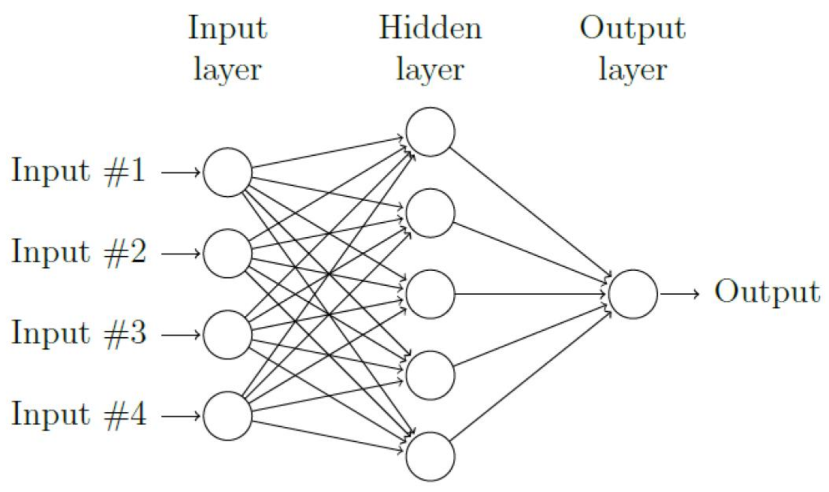


Fig. 1.13: *A simple DNN structure [9].*

In most cases, the training of ANN classifiers is done in a supervised manner forward propagation and backward propagation.
The neurons in neural networks are grouped into layers. Each neuron in the network produces an output according to the function shown in Figure 1.12. the value calculated by each neuron is passed to all the neurons in the next layer of the network. This process is called forward propagation. The output of the neuron in the last layer of the network is the output value of the whole network.

Another process called back propagation used in conjuncture with forward propagation uses error optimization algorithms such as gradient descent to calculate errors in predictions and then adjusts the weights and biases of the neurons by moving backwards through the layers. Together, forward propagation and backpropagation allow a neural network to minimize the errors it makes in each classification, thus resulting in a more accurate result.

The above describes in simple terms how simple deep neural networks operate to make intelligent decisions. However, DNN's algorithms can get very complex quickly, and there are many types of neural networks to address different problems.

One of the types that derives from simple neural network architecture are Convolutional neural networks. This type of network is used primarily in computer vision and image classification applications, can detect features and patterns within an image, enabling tasks, like object detection or recognition. In 2015, a CNN bested a human in an object recognition challenge for the first time [11].

1.6 Convolutional Neural Networks

Convolutional neural networks are the best type of neural networks for image classification and detection problems because of they provide the highest accuracy amongst all the types of neural networks. They were first proposed by computer scientist Yann LeCun in the late 90s; after he was inspired from the human visual perception of recognizing things. The basic building block of a CNN is the convolutional layers. at the end of the convolutional layers sequence, we find a few fully connected layers like the one found in simple feed forward neural networks.

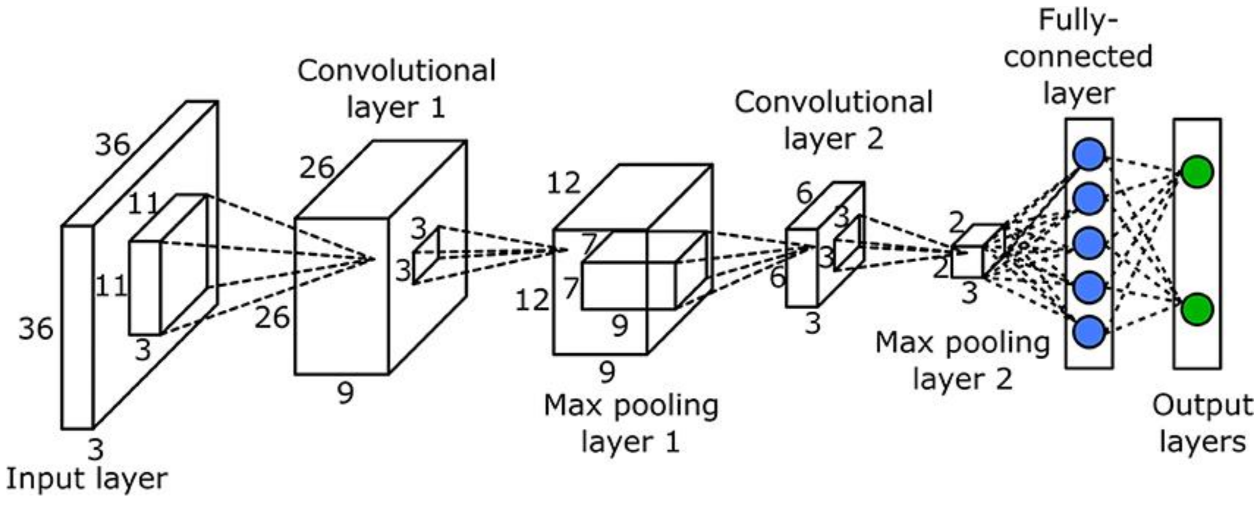


Fig. 1.14: A simple convolutional neural network [12].

Why choose CNNs over feedforward NN for image data

One approach to attempt classification on image data is to use feedforward neural networks like the one depicted in Figure 1.13. For example, we could flatten the image into a vector (24×24 image into a 576×1 vector) and feed it as an input to a feedforward neural network.

In case of extremely basic binary images, the above approach has been found to show an average score when performing classification. However, the approach had little to no accuracy when performing on complex images having different pixel dependencies across the image.

A CNN network can be trained to understand the sophistication of the image better. The reason for this is the nature of the architecture allows for a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights.

Furthermore, unlike fully connected layers of feedforward networks, convolutional networks have sparse connections. In each layer, each output value depends only on a small number of input values. This promotes far less computations through the network allowing for faster forward and backwards propagation routines.

1.6.1 Types of CNN Layers

In order to understand the overall architecture of CNNs and how they operate, we must first familiarize ourselves with the different types of layers a CNN usually consists of.

1. **Convolutional Layers**

   This type of layer is the core building block of a CNN that does most of the computational heavy lifting. It is the first layer to extract features from the input image (generally RGB). The kernel superposed over the input image starting from the top left corner. It starts shifting according to the stride value of that layer, performing a matrix multiplication at each step. After the convolution, another image is obtained with a different height, width, and depth [13].

As depicted in Figure 1.15 below, each conv layer must have at least one filter associated with it. The number of filters in each layer corresponds to the number of channels in the result image of that layer (e.g., for the example in Figure 1.15 below, the layer has one filter thus the result block has only one channel).

An important aspect to keep in mind is that the number of channels n_c in each kernel in a particular layer must be equal to the number of channels of the input image of that layer.

A padding value in CNN layers refers to the number of pixels added to the input image prior to being processed by a kernel. The two most commonly used types of padding in CNN layers are same padding and valid padding.

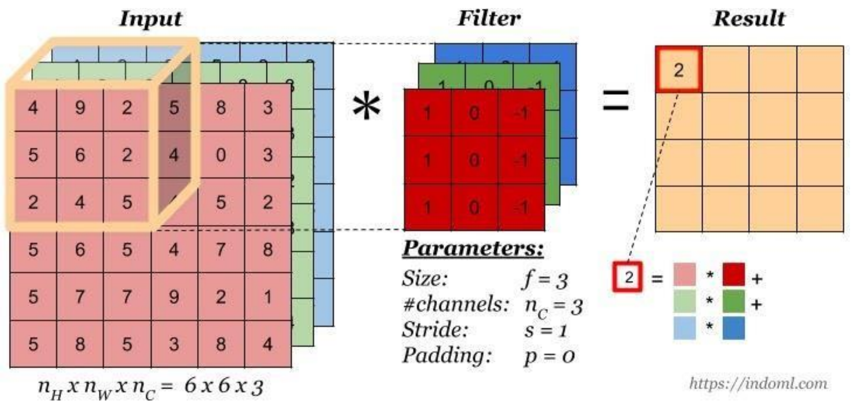


Fig. 1.15: *Convolutional layer with one filter [12].*

The bias is another quantity associated with each filter in each layer. The bias is added to each element of the output image after the filter has gone through all the input image. If a layer has 32 filters, that layer would have 32 bias parameters associated with each filter.

Also, after adding a bias, we apply an activation function on the computed matrix. This is done to introduce non-linearity to the network.

2. **Pooling Layers**

This layer performs a down-sampling operation, typically applied after a convolution layer, which does some spatial invariance. In particular, max and average pooling are special kinds of pooling where the maximum and average value is taken, respectively.

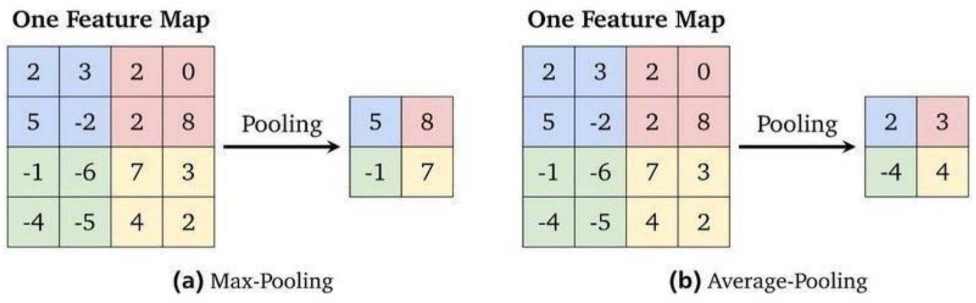


Fig. 1.16: *Max-Pooling and Average Pooling operations [12].*

3. **Fully Connected Layers (for Classification)**

The fully connected layer (FC) operates on a flattened input where each input is connected to all neurons. If present, FC layers are usually found towards the end of CNN architectures and can be used to optimize objectives such as class scores.

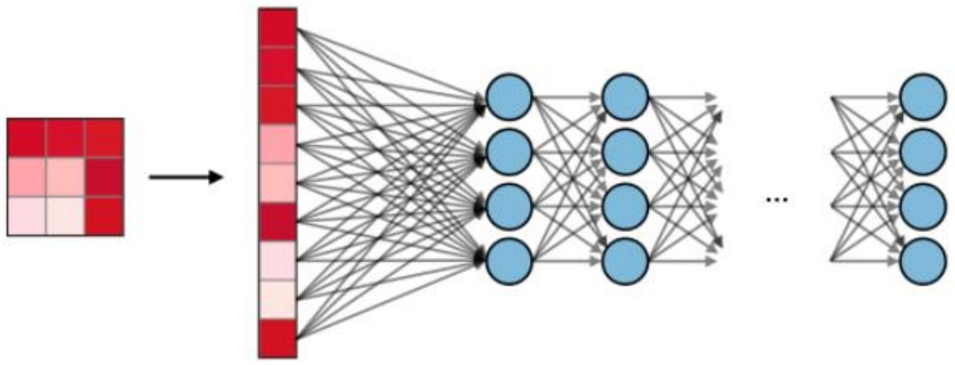
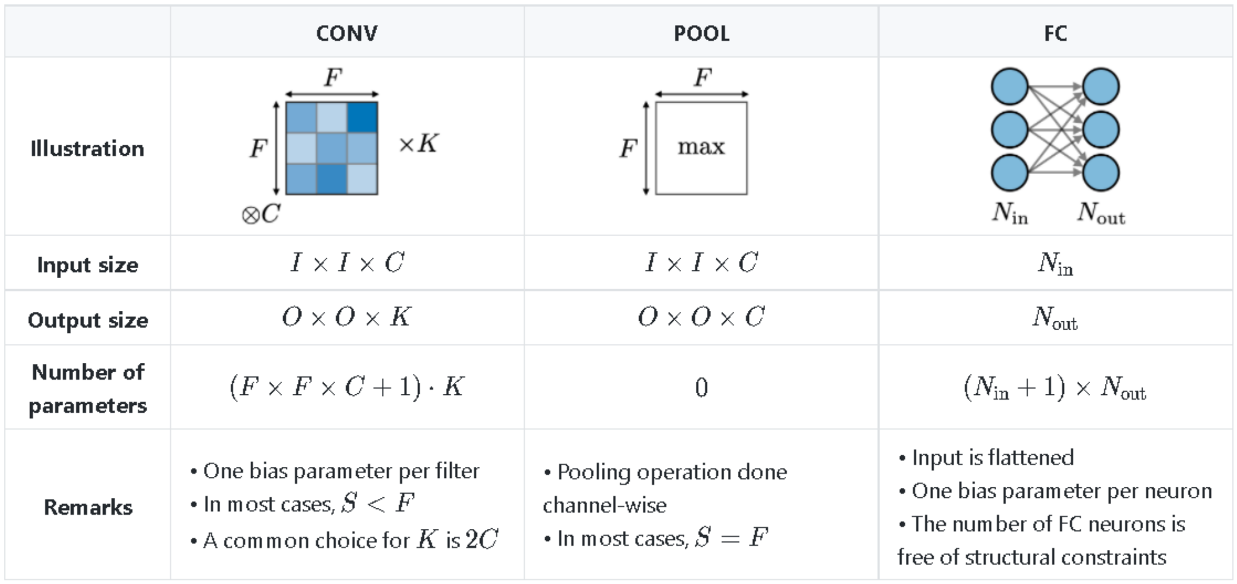


Fig. 1.17: *Fully connected layers [14].*



	CONV	POOL	FC
Illustration	F, F, $\times K$, $\otimes C$	F, F, max	N_{in}, N_{out}
Input size	$I \times I \times C$	$I \times I \times C$	N_{in}
Output size	$O \times O \times K$	$O \times O \times C$	N_{out}
Number of parameters	$(F \times F \times C + 1) \cdot K$	0	$(N_{\text{in}} + 1) \times N_{\text{out}}$
Remarks	• One bias parameter per filter • In most cases, $S < F$ • A common choice for K is $2C$	• Pooling operation done channel-wise • In most cases, $S = F$	• Input is flattened • One bias parameter per neuron • The number of FC neurons is free of structural constraints

***Fig. 1.18:** summary of the properties of each type of layer [14].*

Note: In figure 1.18, *O* denotes the output size, and it has been shown previously in this report how to calculate it. *S* represents the stride of the filters (kernels).

1.6.2 Training a CNN

At first our model is initialized with random parameters (filters values and biases in each layer). The appropriate parameters result in a minimum error or loss when evaluating the examples in the training dataset. Those parameters are found by training the model.

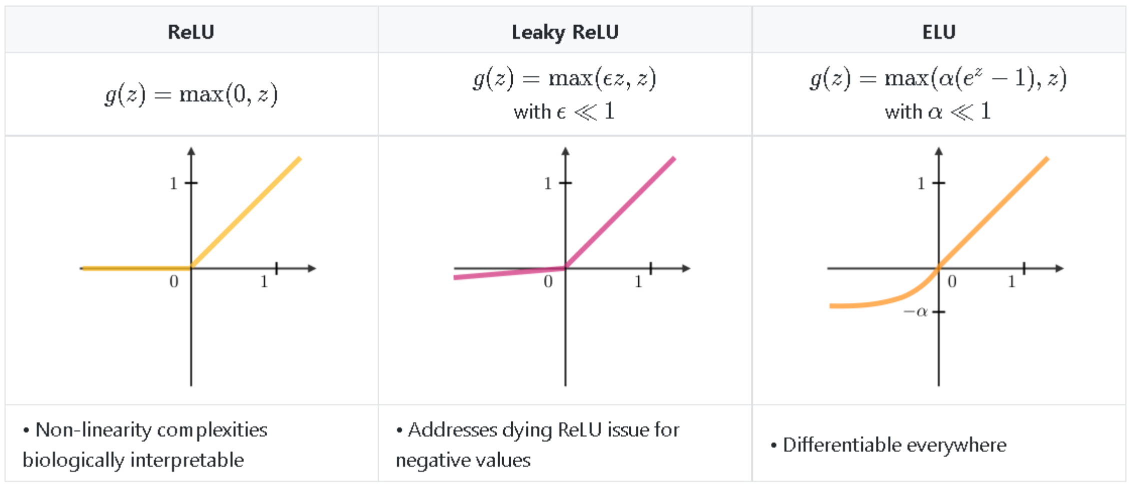
1.6.2.1 Activation Functions

An important aspect about any CNN model is the activation function that model uses. Activation functions are used to introduce non-linearities to the network. Without them, the network would perform linear combinations of the features which would lead to no better result than using a simple linear regression algorithm.

The choice of the activation function impacts the performance of the neural network. All hidden layers typically use the same activation function; however, the output layer uses a different one that depends on the type of prediction performed by the model. The most ubiquitous activation functions in practice are the following:

1. **Rectified Linear Unit (ReLU) – typically used in the conv layers**

   It is an activation function that is used on all elements of the volume. Also, one of its variants may be used in practice.



ReLU	Leaky ReLU	ELU
$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$	$g(z) = \max(\alpha(e^z - 1), z)$ with $\alpha \ll 1$
• Non-linearity complexities biologically interpretable	• Addresses dying ReLU issue for negative values	• Differentiable everywhere

Fig. 1.19: *Summary of the ReLU activation function and its variants [14].*

2. **Softmax – used in the output layer**

   the softmax function takes as input an n-dimensional vector *x* and output a similar vector p of outputs probability. i.e., it assigns a probability to each entry of the input vector.

$$p = \begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix} \tag{1.6}$$

$$\text{where } p_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{1.7}$$

1.6.2.2 Cost Function

The cost function calculates how much our predicted result deviates from the actual label that we want to predict. This is done by averaging the results of a given loss function across the entire dataset (in case of using batch gradient descent) or the entire batch (in case of using mini-batch gradient descent). The most used loss function in practice for outputs that have a probability value between 0 and 1 is the cross-entropy loss function.

$$L(y,p) = -\sum_{i=1}^{C} y_i log(p_i) \tag{1.8}$$

The above loss function is evaluated for every training example when training our model. Where y_i is the label vector for the example i and p_i is our prediction vector for the same example calculated by the softmax function. *c* defines the number of classes in our dataset.

1.6.2.3 Gradient Descent

The gradient descent algorithm calculates how much we need to change our model's parameters to minimize the cost function. To start finding the right values, we initialize our parameters randomly, then we apply gradient descent iteratively to get closer to the minimum value of the cost function at each iteration.

There are three main gradient descent algorithms used in practice. They differ on when they update the values of the parameters. We are going to cover two of them:

1. **Batch Gradient Descent**

   calculates the error for each example within the training dataset, but only after all training examples have been evaluated does the model get updated. This whole process is like a cycle, and it's called a training epoch (more on this later). The formula used to update the parameters if the following:

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_1, \dots, \theta_k) \tag{1.9}$$

where: j is the index of the parameter that we want to update.
α is the learning rate (more on this later).
k is the number of parameters we have in our model.
J is the cost function.

2. **Mini-Batch Gradient Descent**

Mini-batch gradient descent is the one that we used in this project. This algorithm splits the training dataset into small batches of randomly picked training examples and performs an update for each of those batches.

Common mini-batch sizes range between 50 and 256, but like any other machine learning technique, there is no clear rule because it varies for different applications.

1.6.2.4 Backward Propagation

The backward propagation algorithm allows the information from the cost to then flow backwards through the network, to compute the gradient of the loss function with respect to the model's parameters. The gradient is then used by an optimization algorithm such as the gradient descent to update the model's parameters.

The algorithm involves the recursive application of the chain rule from calculus that is used to calculate the derivative of a sub-function given the derivative of the parent function for which the derivative is known.

The term backward propagation is often misunderstood as meaning the whole learning algorithm for multi-layer neural networks. Backward propagation refers only to the method for computing the gradient, while another algorithm, such as mini-batch gradient descent, is used to perform learning using this gradient.

1.5.2.5 Hyperparameters tuning

Tuning hyperparameters for deep neural network is difficult as it is slow to train a deep neural network and there are numerous parameters to configure. In this part, we briefly cover the most common hyperparameters found in a CNN.

1. **Learning Rate**

   Learning rate controls how much to update the weight in the optimization algorithm. We can use fixed learning rate, gradually decreasing learning rate, momentum-based methods, or adaptive learning rates, depending on our choice of optimizer such as Adam.

2. **Number of Epochs**

   Number of epochs is the number of times the entire training set pass through the neural network. We should increase the number of epochs until we see a small gap between the test error and the training error.

3. **Batch size**

   A hyperparameter of gradient descent that controls the number of training samples to work through before the model's internal parameters are updated. Mini batch is usually preferable in the learning process of CNNs. A range of 16 to 128 is a good choice to test with.

4. **Number of convolutional layers, filters, filter sizes**

   It is usually good to add more layers until the test error no longer improves. The tradeoff is that it is computationally expensive to train the network. Having a small number of units may lead to underfitting while having more units are usually not harmful with appropriate regularization.

1.6.2.6 Overfitting and Underfitting

Overfitting occurs when our model performs well on our training set but poorly on our testing set. This can be seen as our model memorizing the training examples and failing to generalize to new examples. When a model is overfitting, we say it has high variance. In machine learning, the difference in fits between datasets is called variance.

Overfitting can be reduced using regularization techniques such as L2 Regularization and Dropout.

- **L2 Regularization**

for this type of regularization we need to add the following value to our loss function L:

$$L(y,p) = -\sum_{i=1}^{C} y_i log(p_i) + \lambda \sum_{j=1}^{k} {\theta_j}^2 \tag{1.10}$$

As stated before, θ represents our parameters i.e., our filters' elements' values and there are k of them. λ is set before initiating the training.

If we have overfitting, we can reduce the weight that some of the terms in our model carry by increasing their cost. If the value of lambda is large enough, the parameters will be decreased because the added L2 expression will penalize our parameters. Thus, reducing the influence that some parameters have on the decision, therefore simplifying our model.

- **Dropout**

The key idea is to randomly drop units (along with their connections) from the neural network during training. This means that their contribution to the activation of downstream neurons is temporarily removed on the forward pass and any weight updates are not applied to the neuron on the backward pass [12].

Contrary to overfitting, underfitting occurs when our model fails to classify the examples it was trained on. In machine learning, when our model fails to capture the true relationship among our input features it is said to have a high bias. This problem usually happens, when we don't have enough data, when we over-simplify our model or when we don't train our model for enough epochs.

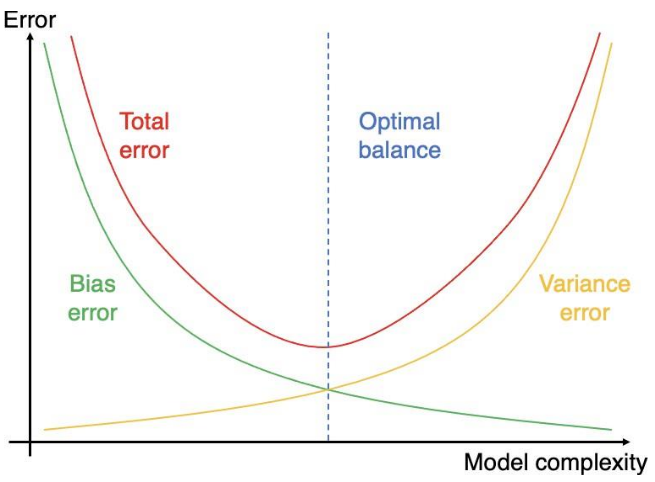


Fig. 1.20: *Bias-variance trade-off [12].*

1.7 Transfer Learning

Transfer learning is a technique where rather than training the weights from scratch, we download the weights that someone else has trained on that same network architecture and use that as starting point for our specific problem. The starting point weights used in this project are the weights that were trained on the *ImageNet* dataset.
For transfer learning to work well, we need to pick models that are pre-trained on a type of data that is similar to the type of data that we are using. For example, in this project we are working

with image data, thus, the pre-trained models we need to pick need to be trained on image data as well.

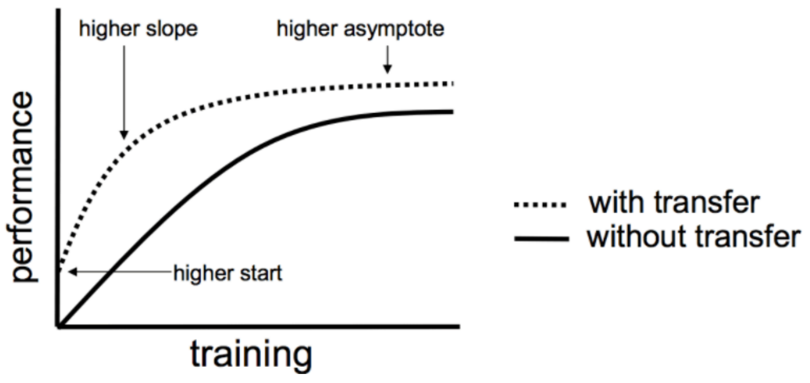


Fig. 1.21: *Transfer learning performance comparison graph [15].*

1.7.1 Models Used

For the purpose of this project, we are going to gather and use low-computation CPU friendly pre-trained models trained on image data. Also, we will use simple models created from scratch.

1.7.1.1 MobileNetV2

MobileNetV2 is a convolutional neural network architecture that seeks to perform well and have low latency on mobile and embedded devices. It is an improvement upon the original MobileNetV1 architecture. The key idea was to use depthwise-separable convolutions instead of normal convolutions.

Depthwise-separable convolutions

This type of convolutions does the computations in two steps: depthwise convolutions, then pointwise convolutions. This technique was proven to reduce the computation time by a factor of 10 in typical CNN layers.

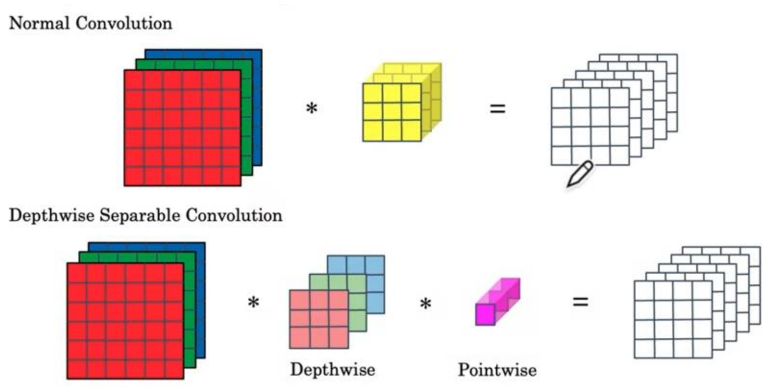


Fig. 1.22: *Depthwise separable convolution.*

Considering the above figure, in the normal convolution we would have to make 2160 multiplications to obtain our result. However, when using the depthwise separable convolution technique, the computation is reduced to only 672 multiplications.

The authors of the MobileNet paper showed that in general, the ratio of the cost of the depthwise separable convolution compared to the normal convolution is determined using this equation:

$$r = \frac{1}{n_c} + \frac{1}{f^2} \tag{1.11}$$

Where n_c is the number of channels in the filter and f is the filter size.

Inverted Residuals and Linear Bottleneck Block

In MobileNet v2, there are two main changes. One is the addition of a residual connection. This residual connection or skip connection, takes the input from the previous layer, and passes it directly to the next layer, does allow gradients to propagate backward more efficiently.

The second change is the addition of an expansion layer before the depthwise convolution, followed by the pointwise convolution, which we're going to call projection in a point-wise convolution.

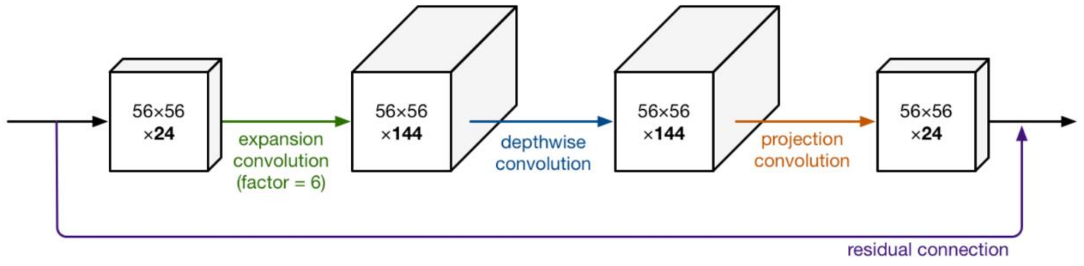


Fig. 1.23: *Bottleneck Block.*

Residual connections connect the beginning and end of a convolutional block with a skip connection. By adding these two states the network has the opportunity of accessing earlier activations that weren't modified in the convolutional block. This approach turned out to be essential in order to build networks of great depth.

Advantages of the Bottleneck Block

The bottleneck block accomplishes two things, One, by using the expansion operation, it increases the size of the representation within the bottleneck block. This allows the neural network to learn a richer function.

The bottleneck block uses the pointwise convolution or the projection operation in order to project it back down to a smaller set of values, so that when you pass this the next block, the amount of memory needed to store these values is reduced back down. This is very useful when deploying the model on memory restrained mobile and embedded devices

That's why the MobileNet v2 can get a better performance than MobileNet v1, while still continuing to use only a modest amount of compute and memory resources.

1.7.1.2 EfficientNet

First introduced in 2019 by Tan and Le, EfficientNet is among the most efficient models (i.e., requiring least floating-point operations per second for inference) that reaches state-of-the-art accuracy on both the *ImageNet* dataset and common image classification transfer learning tasks.

To scale a CNN, we could either: scale the resolution of the input image *r*, adjust the depth of the model *d*, or adjust the width of the model *w*. The authors of EfficientNet gave a method to find the optimal trade-off between those three values in order to get the best possible performance within a given computational budget.

1.8 Related Work

Motivated by the tedious task of manual observation of the face mask in crowded places, researchers have been seeking to automate the task of face mask detection via an AI system.

In this paper [16], the authors presented a MobileNet architecture with a global pooling block for face mask detection. The proposed model employs a global pooling layer to perform a flatten of the feature vector. A fully connected dense layer associated with the softmax layer has been utilized for classification. The proposed model outperforms existing models on two publicly available face mask datasets in terms of vital performance metrics.

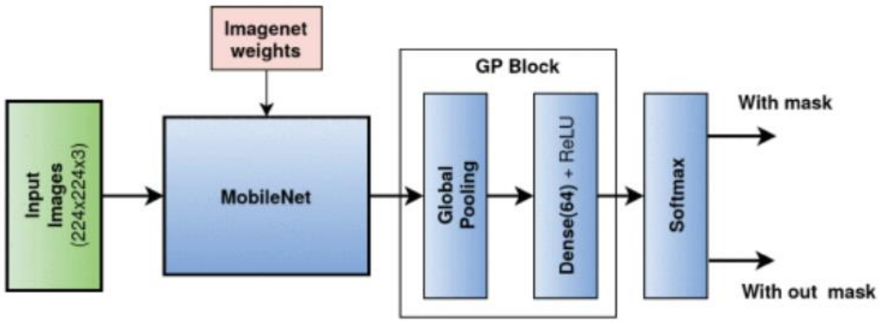


Fig. 1.24: *Model proposed by paper [21] authors.*

The authors trained the model for 50 epochs for a batch size of 8 examples. They set the initial learning rate to 0.0001 and used the Adam optimizer. Their proposed model records best perform of 99.56%.

1.9 Conclusion

Convolutional neural networks have been the most prominent type of machine learning techniques used in image data classification tasks. Haar cascade classifiers have been shown to be amongst the most efficient type of classifiers for computationally restrained devices such as mobile and embedded devices. In this chapter, all the machine learning methods that are going to be used in this project have been thoroughly explained with detailed examples. Thus, in this project it has been decided to use both Haar cascade classifiers and CNNs to carry out the main task of this project.

Chapter 2: Models and the Dataset Used

2.1 Introduction

In this chapter, the methods used for the implementation of the proposed system will be presented. The implementation follows four major steps: exploratory data analysis, preprocessing, face detection, face mask detection. Since the choice of the data is important to neural networks, it additionally involved the search for proper data sets. The dataset suggested in some of the consulted state-of-the-art papers [17], is used as a baseline for the evaluation of CNNs.

2.2 Dataset - RMFD

The dataset used in this project is known as the Real-world Masked Face Dataset (RMFD). It was created by researchers from Wuhan University, and it was sponsored by the National Engineering Research Center for Multimedia Software (NERCMS) and the School of Computer Science of Wuhan University.

This dataset was primarily developed for masked face recognition purposes. i.e., developing a technique to recognize people even when they are wearing face masks.

The researchers provided two three types of masked faces datasets, including Masked Face Detection Dataset (MFDD), Real-world Masked Face Recognition Dataset (RMFD) and Simulated Masked Face Recognition Dataset (SMFD).

Given that we are not concerned with doing recognition in this project, the most suitable type of dataset for our project would be the MFDD variant. However, after extensive research online for this dataset, it was nowhere to be found. Thus, we settled with the RMFD dataset and manipulated it to fit the needs of our project.

RMFD is a dataset that contains two classes: masked faces and normal faces. Given that this dataset was intended for recognition purposes. Each classes contains a folder for every person used in the dataset. In total, this dataset contains 5,000 masked faces of 525 people and 90,000 normal faces.

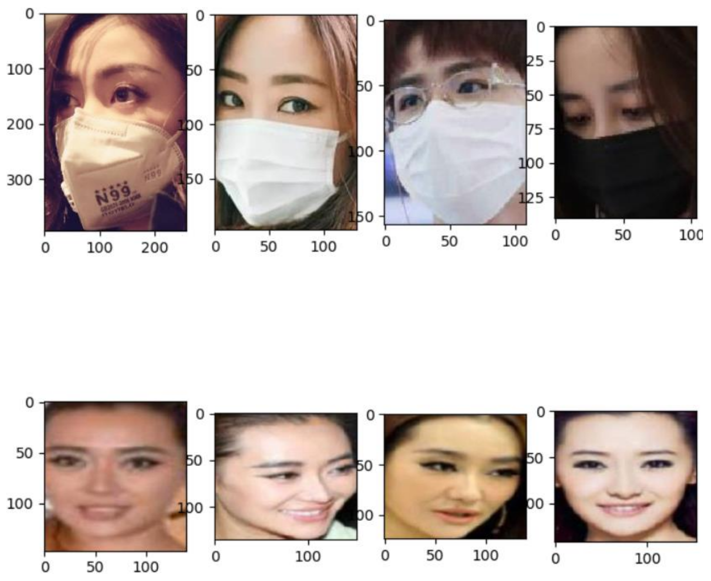


Fig. 2.1: *Masked and normal faces examples from RMFD.*

After downloading the dataset, it was found that it contains only 2,203 masked faces. Thus, the dataset was balanced to also have 2,203 normal faces. i.e., 88,265 images of normal faces were removed.

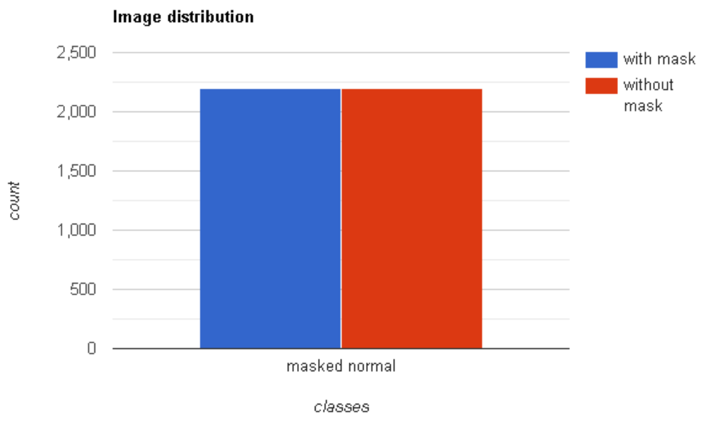


Fig. 2.2: *Classes distribution after balancing the dataset.*

2.3 Preprocessing

In order to train our convolutional neural networks, we need to preprocess the images that we gathered from our dataset in a way that is convenient for our models to consume. There are certain criteria our images must meet in order for them to be accepted.

The image size that is most commonly accepted by the pre-trained models used in this project is 224×224 pixels. Thus, the images were resized to fit the aforementioned size.

To make our model converge faster, we have to normalize the pixel values in each picture. That is, pixel values in our pictures must range from -1 to +1.

Also, our images are labeled with their respective class names. Since our models will be using the softmax output function, and given that we have two classes, then our data to be accepted in our models we need to encode those alphabetic names into vectors using one-hot encoding. This resulted in encoding the *with_mask* class into the vector $\begin{bmatrix}1\\0\end{bmatrix}$ and the *without_mask* class into the vector $\begin{bmatrix}0\\1\end{bmatrix}$.

The data is then shuffled and split into a training set and a testing set. The training set make up 72% of the dataset, the testing set represents 20% and the validation set is 08%. It is also worth mentioning that the split was stratified. i.e., the percentage of each class in our sets is preserved.

Table 2.1: Number of samples in each set.

Class	*With mask*	*Without mask*	*Percentage*
Training set	1586	1585	72%
Validation set	176	177	08%
Testing set	441	441	20%
Total	2203	2203	100%

After all the preprocessing is done, we resulted in six arrays containing our data. These arrays are stored on disk to avoid doing the preprocessing every time we need to train a model.

2.4 Haar Cascade Classifier

Since we are going to classify images coming from the camera in real-time, we need to detect the faces before doing the classification. The detection of the faces is done by a publicly available Haar cascade classifier. It is a popular classifier used in many projects. This classifier contains 25 stages. It is provided in an *.xml* file and can be found in the OpenCV data folder.

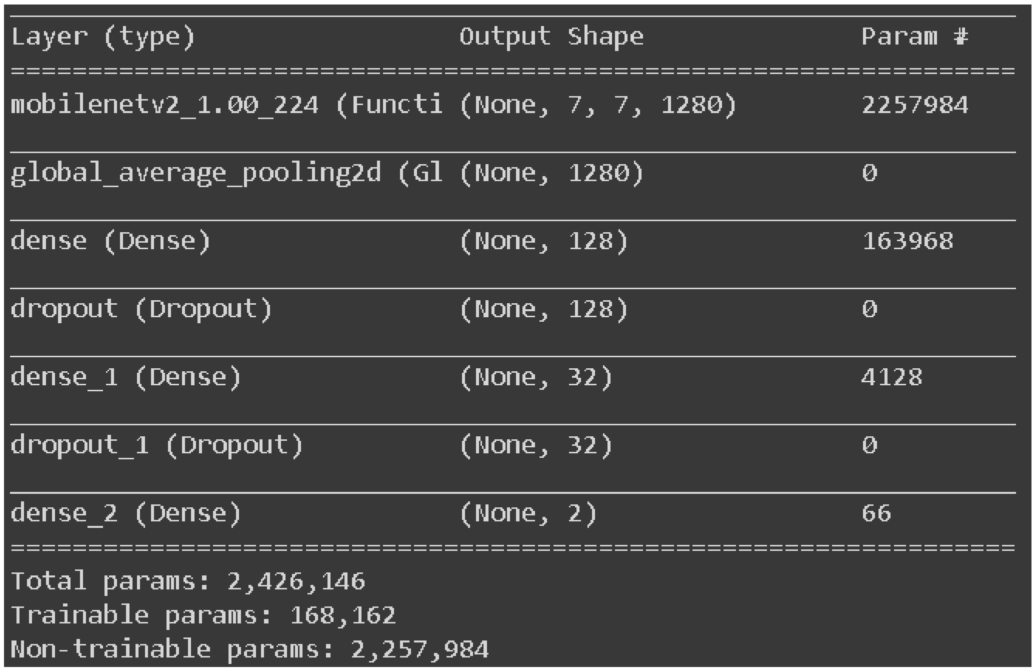
2.5 Models

Given the low computational budget nature of our project, the most computationally efficient pre-trained models have been picked, namely: MobileNetV2 and EfficientNetB0. Furthermore, to experiment more with custom CNN models built from scratch, a simple CNN model was created to compare it against the pre-trained models that we picked.

2.5.1 MobileNetV2

The model accepts the input size of 224×244. The input size perfectly fits our preprocessed dataset. Thus, we don't need to resize our images.
As a whole, the architecture of MobileNetV2 contains the initial fully convolution layer with 32 filters, followed by 19 residual bottleneck layers.

To benefit from transfer learning, the architecture of this model was imported from *Keras* (a *python* library that runs on top of *TensorFlow*) and was initialized with weights that were already trained on the *ImageNet* dataset. This dataset contains 1000 classes. Thus, the classification layers were removed from the architecture and were replaced by our own classification layers as depicted in the figure below.



```
Layer (type)                 Output Shape              Param #
=================================================================
mobilenetv2_1.00_224 (Functi (None, 7, 7, 1280)        2257984

global_average_pooling2d (Gl (None, 1280)              0

dense (Dense)                (None, 128)               163968

dropout (Dropout)            (None, 128)               0

dense_1 (Dense)              (None, 32)                4128

dropout_1 (Dropout)          (None, 32)                0

dense_2 (Dense)              (None, 2)                 66
=================================================================
Total params: 2,426,146
Trainable params: 168,162
Non-trainable params: 2,257,984
```

Fig. 2.3: *MobileNetV2 architecture.*

In total, 6 layers were added. The dropout layers were used to avoid overfitting.

2.5.2 EfficientNet-B0

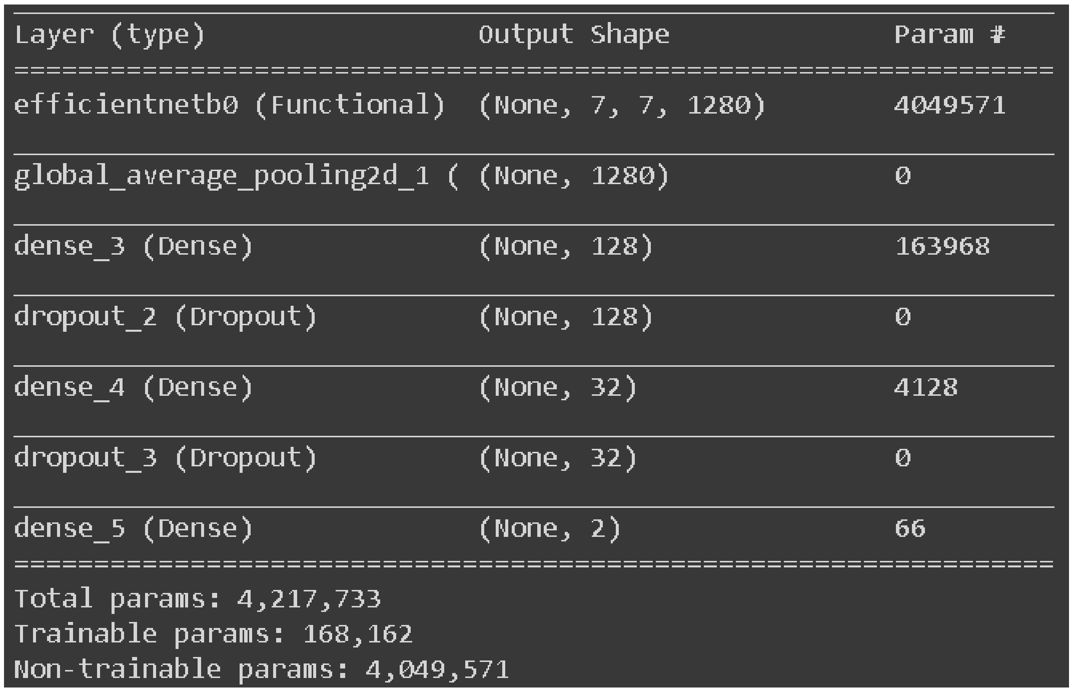
This architecture is from the EfficientNet family of models. The *B0* variant was found to accept an input size of 224×224. Thus, this variant was chosen for the sake of this project.

Taking B0 as a baseline model, the authors developed a full family of EfficientNet from B1 to B7 which achieved state of the art accuracy on ImageNet while being very efficient to its competitors.

The EfficientNet-B0 architecture wasn't developed by engineers but by the neural network itself. They developed this model using a multi-objective neural architecture search that optimizes both accuracy and floating-point operations.

Taking B0 as a baseline model, the authors developed a full family of EfficientNet from B1 to B7 which achieved state of the art accuracy on ImageNet while being very efficient to its competitors.

Similar to the previous pre-trained model discussed, this model was imported from the *Keras* library along with the *ImageNet* weights. The classification layers were left off and we added the same classification layers as the previous model for a fair comparison. Figure 2.5 below shows the parameters and the layers of the final model.



```
Layer (type)                 Output Shape              Param #
=================================================================
efficientnetb0 (Functional)  (None, 7, 7, 1280)        4049571
_________________________________________________________________
global_average_pooling2d_1 ( (None, 1280)              0
_________________________________________________________________
dense_3 (Dense)              (None, 128)               163968
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_4 (Dense)              (None, 32)                4128
_________________________________________________________________
dropout_3 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_5 (Dense)              (None, 2)                 66
=================================================================
Total params: 4,217,733
Trainable params: 168,162
Non-trainable params: 4,049,571
```

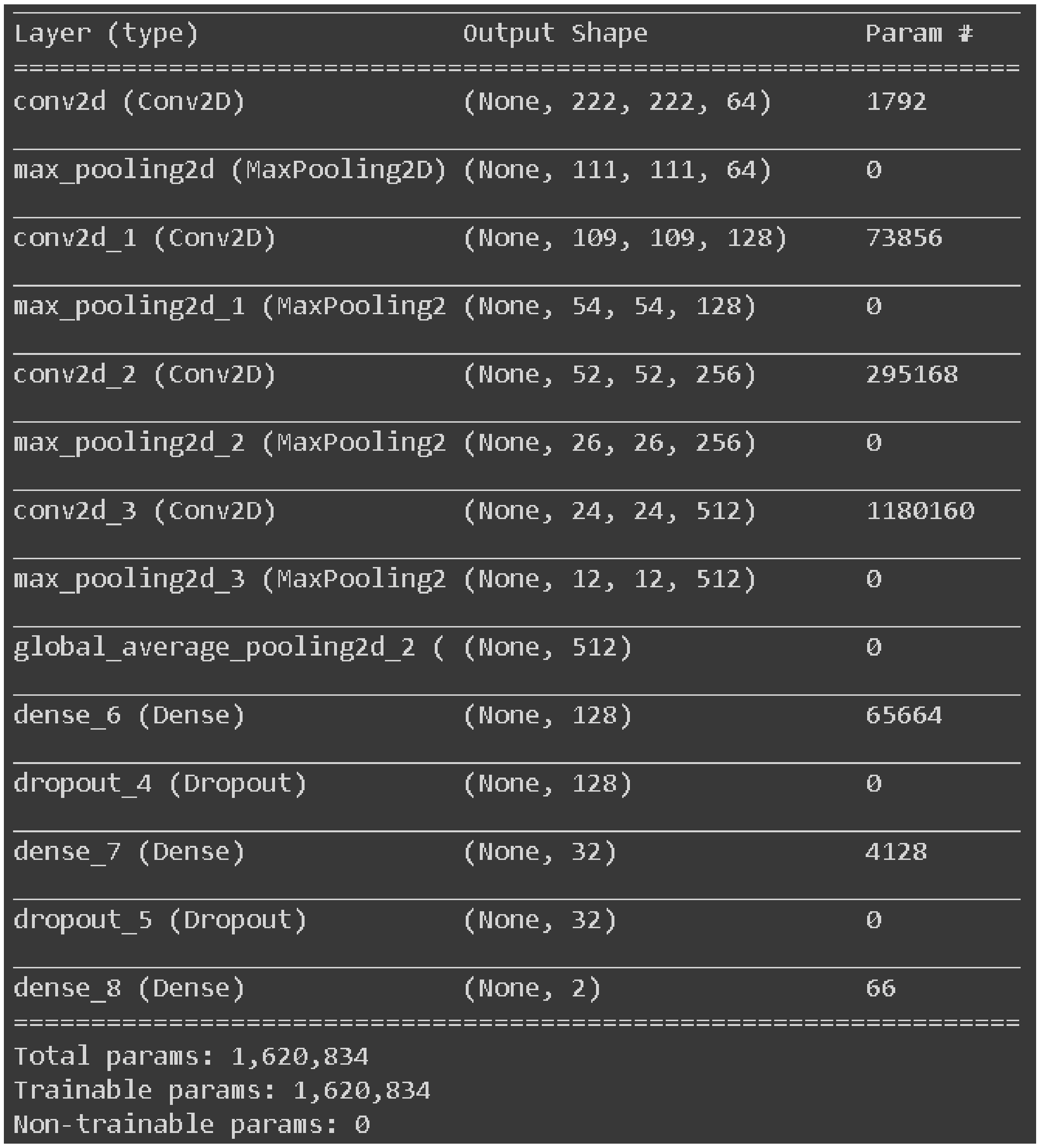
Fig. 2.4: *EfficientNet-B0 architecture.*

2.5.3 Custom Model

For experimental reasons, we wanted to compare the performance and execution speed of a custom simple CNN model built from scratch to the pre-trained models used.

The architecture of the model is depicted in Figure 2.6 below. This model has significantly less parameters than the previous models. But this does not guarantee that it has less floating-point operations per second (which is an indirect metric for a model’s execution speed).

The model starts by 4 convolutional layers. Each convolutional layer is followed by a max pooling layer with a stride of 2. These pooling layers reduce the dimension of the input by a factor of 2 (as shown in Figure 2.6). then, the same classification layers as the previous models were added in order to account for a fair comparison between the models.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 222, 222, 64)      1792
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 111, 111, 64)      0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 109, 109, 128)     73856
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 54, 54, 128)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 52, 52, 256)       295168
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 26, 26, 256)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 24, 24, 512)       1180160
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 12, 12, 512)       0
_________________________________________________________________
global_average_pooling2d_2 ( (None, 512)               0
_________________________________________________________________
dense_6 (Dense)              (None, 128)               65664
_________________________________________________________________
dropout_4 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_7 (Dense)              (None, 32)                4128
_________________________________________________________________
dropout_5 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_8 (Dense)              (None, 2)                 66
=================================================================
Total params: 1,620,834
Trainable params: 1,620,834
Non-trainable params: 0
```

Fig. 2.5: *Custom model architecture.*

2.6 Tools Used

2.6.1 Python

Python is an interpreted high-level general-purpose programming language. This language is a great choice for machine learning project because it is intuitive. contains many open-source libraries for machine learning and data science. The version of python used in this project is python 3.7.2

2.6.2 Keras

Keras is an open-source software library that runs on top of *TensorFlow*. Is provides all the tools needed to create and develop neural networks. The version of keras used in this project is 2.4.0

2.6.4 Training Environment

The preprocessing and the training of the models was done on the Google Colab Pro platform. This platform requires no allows anyone to write and run the Python code through the browser. It is an environment particularly suited to machine learning and data analysis.

The environment provided us with 25GB of RAM and a Tesla T4 GPU with 13.5GB of VRAM. This makes the training process of our models very fast.

2.6.5 Testing Environment and Device

To test and evaluate our models we had to set up a Raspberry Pi 3 Model B with Raspberry Pi OS which comes with python installed. We then installed *TensorFlow* 2.4.0 along with *Keras* 2.4.0 and a few other libraries.

2.7 Conclusion

In this chapter, a detailed explanation of the methods used in order to detect and classify whether people are wearing face masks was given. First, a description of the dataset was provided where we explored the structure of the data its distribution and how we preprocessed it to make it ready for training and testing. Next, we discussed the method used to detect the faces in each image. We presented three different models that are going to be compared to each other in this project. Finally, we presented the tools used to carry out this project.

Chapter 3: Results and Discussion

3.1 Introduction

In this chapter we will present the metrics used to evaluate the performance of each model. We will also present the results achieved and discuss them.

3.2 Evaluation Metrics

For this project, we will evaluate our models using four metrics: accuracy, precision, recall and F-1 score.

3.2.1 Confusion Matrix

A confusion matrix is a way to visualize how well our model performed on the test set. For each class, it depicts the number of correctly predicted samples versus the number of incorrectly predicted samples. The number of rows in a confusion matrix is defined by the number of classes. Fig. 3.1 shows a general structure of a confusion matrix.

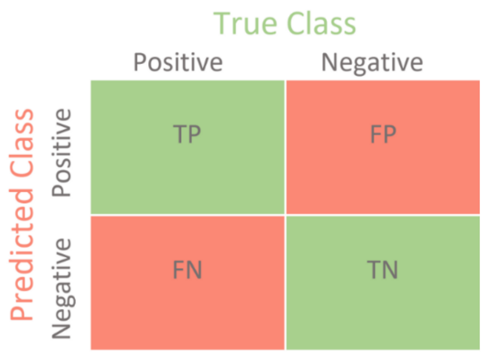


Fig. 3.1: *Two classes confusion matrix [20].*

From the confusion matrix, we can derive four important values for each class. Those four values are used to compute the metrics mentioned before. If we have N classes and we consider class i to be positive and the other N − 1 classes to be negative, then:

- **True positive** (*tp_i*): the number of predictions where the classifier correctly predicts the input sample as belonging to class i.

- **True negative** (*tn_i*): the number of predictions where the classifier correctly predicts the negative samples as negative.

- **False positive** (*fp_i*): the number of predictions where the classifier incorrectly predicts the other N − 1 class as class i.

- **False negative** (*fn_i*): the number of predictions where the classifier incorrectly predicts class i classes as one of the N − 1.

3.2.2 Average Accuracy

It gives the overall accuracy of the model, meaning the fraction of the total samples that were correctly classified by the classifier:

$$accuracy = \frac{1}{N} \sum_{i=1}^{N} \frac{tp_i + tn_i}{tp_i + tn_i + fp_i + fn_i} \tag{3.1}$$

3.2.3 Precision

It tells us what fraction of predictions as a positive class were actually positive. To calculate precision, we use:

$$precision = \frac{tp_i}{tp_i + fp_i} \tag{3.2}$$

3.2.4 Recall

It is the ability of the model to find correct predictions per class.

$$recall = \frac{tp_i}{tp_i + fn_i} \tag{3.3}$$

3.2.5 F1-score

The harmonic average of the precision and recall, it measures the effectiveness of identification when just as much importance is given to recall as to precision for each class.

$$F1_{score} = 2 \times \frac{precision \times recall}{precision + recall} \tag{3.4}$$

3.2.6 average execution time

In order to test the average execution time of our models, we did the test on a Raspberry Pi 3 Model B. the specifications of this device are as follows:

- Quad Core 1.2GHz Broadcom BCM2837 64bit CPU
- 1GB RAM

To calculate the average execution time, we took the mean of 100 different classification prediction operations when running on the aforementioned device.

$$art = \frac{\sum_{i=1}^{N=100} prediction(i)}{N} \tag{3.5}$$

3.3 Experimental Results

The experiments we did in this project consists of training three different CNN models: two models loaded with pre-trained parameters (MobileNetV2 and EfficientNet-B0) trained using transfer learning technique, and one model built from scratch and trained all the parameters.

We are going to start by presenting the results of the pre-trained models first, then we are going to present the results of our custom model that was built from scratch. Then, we are going to compare the models.

3.3.1 MobileNetV2

This model was trained using the hyperparameters shown in Table 3.1. These parameters were chosen because they are the most commonly used when first training a CNN:

Table 3.1: MobileNetV2 hyperparameters.

Initial learning rate	**Batch size**	**Epochs**	**Classification layers**

0.001	32	10	Dense = 128
			Dropout = 0.5
			Dense = 32
			Dropout = 0.5
			Dense = 2

After training the proposed model for 10 epochs on the dataset presented earlier, which contains 3171 training examples and 351 cross-validation examples, it resulted in the following accuracy and loss curves:

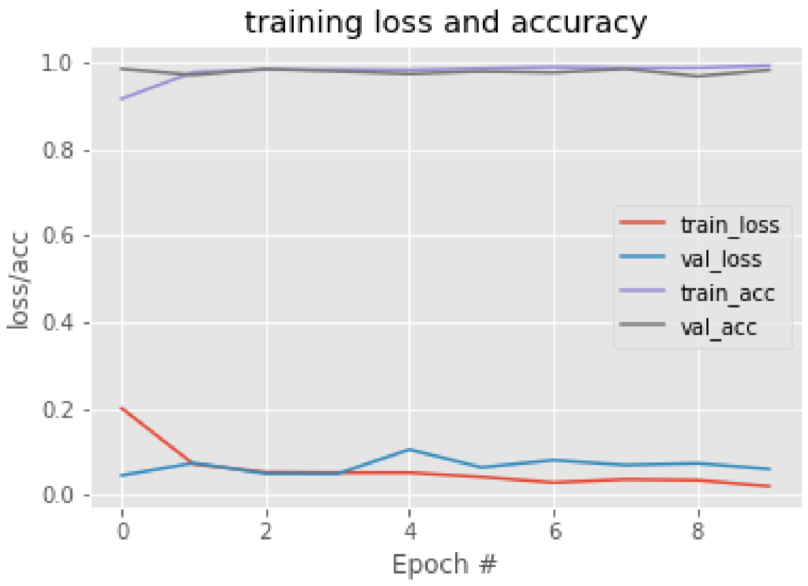


Fig. 3.2: *MobileNetV2 loss and accuracy curves.*

As shown in figure 3.2, the model converged rapidly. Thus, we did not need to train the model for further epochs.

When testing the model on the testing set, which contains 882 examples, the following loss and accuracy were obtained:

Table 3.2: MobileNetV2 testing results.

Loss	**Accuracy**
0.0249	0.9909

We obtained great results on the testing set, which means that our model did not overtrain, thus the model did not overfit the training data and it generalized on the hold-out set pretty well.

We also derived the metrics mentioned earlier in this chapter using a function from the *scikit-learn* library. The following results were obtained:

Table 3.3: MobileNetV2 classification report.

Class	Precision	Recall	F1-score	Accuracy
With mask	0.99	1.00	0.99	0.99
Without mask	1.00	0.99	0.99	

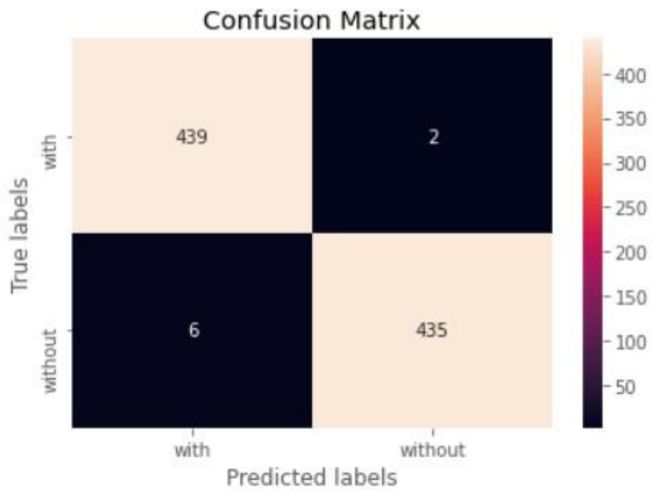


Fig. 3.3: MobileNetV2 confusion matrix.

When the model was tested for execution speed on our testing device (Raspberry Pi 3 Model B), we got the following results:

Execution time
1.19 seconds/prediction

3.3.2 EfficientNet-B0

The hyperparameters used in the training of this model are the same used for the previous model. The same training examples and cross-validation examples were also used. The experiment resulted in the following accuracy and loss curves:

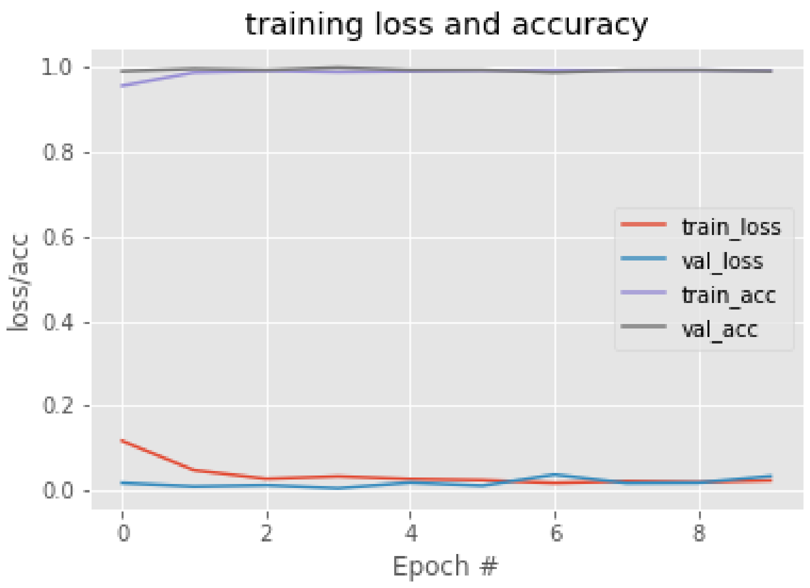


Fig. 3.4: *Efficientnet-B0 loss and accuracy curves.*

As shown in the figure above, the model started to converge starting from the 2nd epoch. The model performed great on the validation set, which indicates the absence of overfitting. The loss value is low which means the error value in each example is small.

When testing the model on the testing set, which contains 882 examples, the following loss and accuracy were obtained:

Table 3.4: Efficientnet-B0 testing results.

Loss	**Accuracy**
0.0451	0.9875

The table above shows that this model did a great job in classifying the examples in the testing set.

Table 3.5: Efficientnet-B0 classification report.

Class	Precision	Recall	F1-score	Accuracy
With mask	0.98	1.00	0.99	0.99
Without mask	1.00	0.98	0.99	

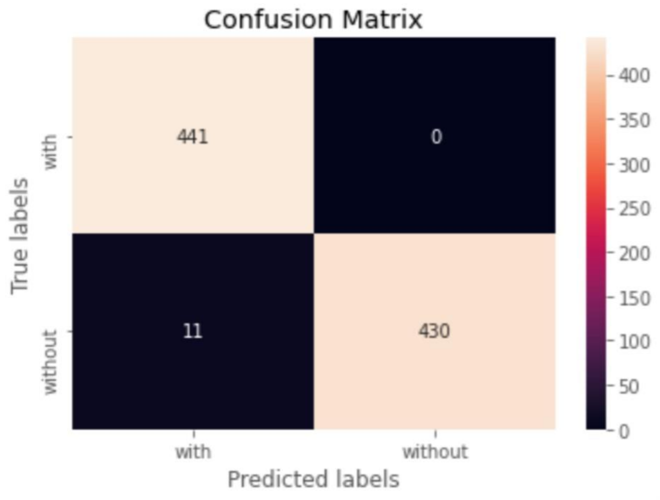


Fig. 3.5: *Efficientnet-B0 confusion matrix.*

When the model was tested for execution speed on our testing device (Raspberry Pi 3 Model B), we got the following results:

Execution time
1.62 seconds/prediction

3.3.3 Custom Model

Unlike the other models, this one got trained for 15 epochs. The learning rate and the batch size have the same values as the previous models.

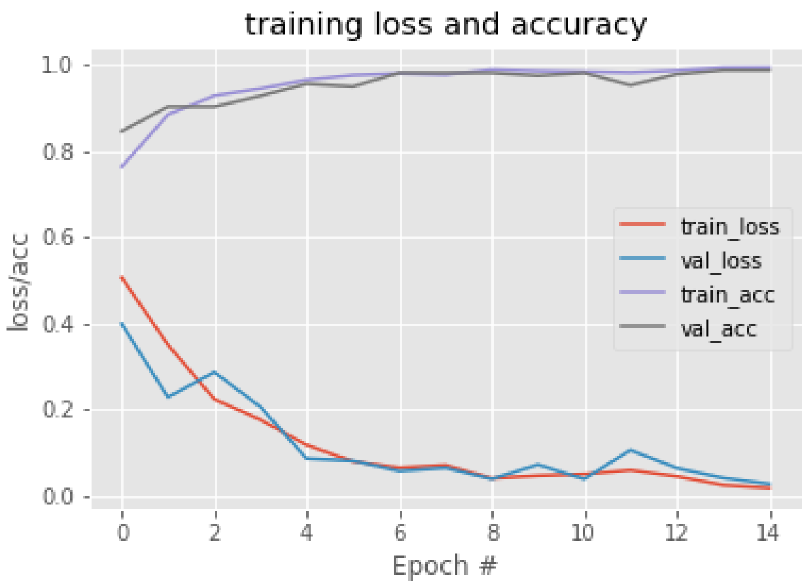


Fig. 3.6: *Custom model loss and accuracy curves.*

As shown in the figure above, the loss values for this model started at a higher value than the previous pre-trained models. The accuracy values started at around 0.8 which is a good value but not as good as the previous two models. The loss values quickly decreased as the training went on and they converged by the last epoch. As for the accuracy values, they also converged quickly.

Table 3.6: custom model testing results.

Loss	**Accuracy**
0.0148	0.9921

The table above shows that this model did a great job in classifying the examples in the testing set.

Table 3.7: Custom model classification report.

Class	**Precision**	**Recall**	**F1-score**	**Accuracy**
With mask	1.00	0.99	0.99	0.99
Without mask	0.99	1.00	0.99	

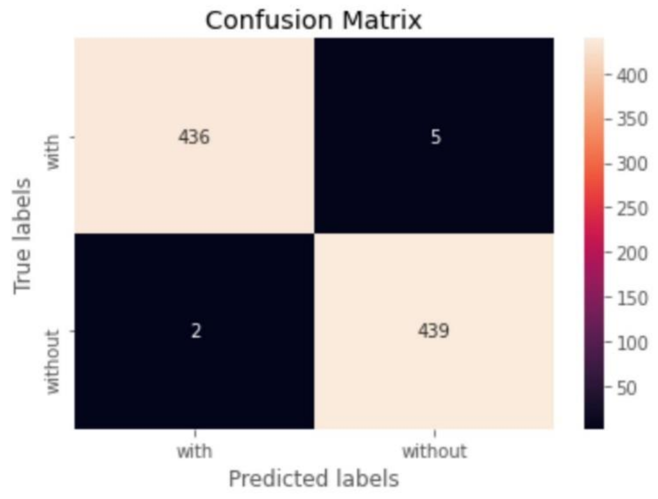


Fig. 3.7: *Custom model confusion matrix.*

When the model was tested for execution speed on our testing device (Raspberry Pi 3 Model B), we got the following results:

Execution time
1.72 seconds/prediction

Although this model has less parameters than the previous pre-trained models, its execution speed is the slowest amongst the three. This is due to the use of depth-wise separable convolution in the pre-trained models which reduces the calculations time drastically.

3.3.3.1 Speed Improvement

In an attempt to reduce the execution speed of this model, the last convolutional layer *conv2d_3* (512 filters, filter size = 3×3, parameters = 1,180,160) was removed from the architecture along with its pooling layer.

The resulting model had 407,906 parameters. This model was trained for 30 epochs and the following results were obtained:

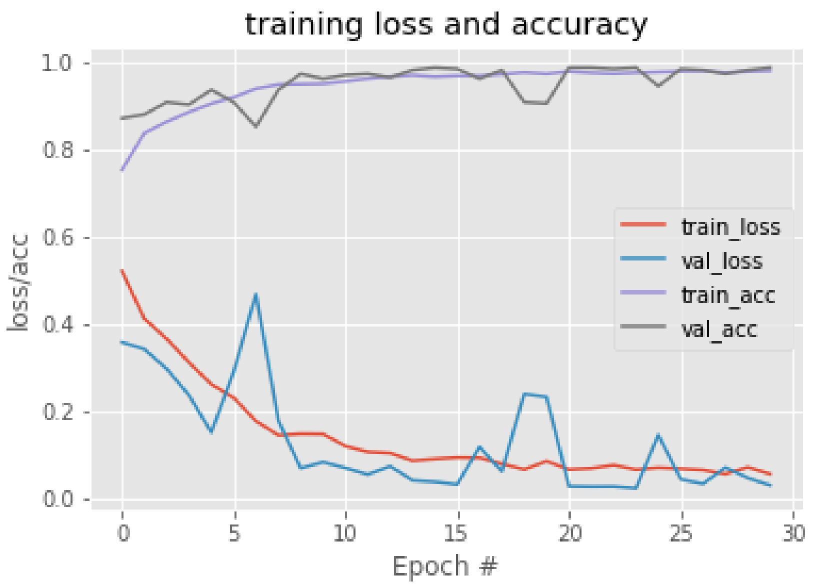


Fig. 3.8: *Improved custom model loss and accuracy curves.*

The number of epochs this model took to converge was higher than before removing the last convolutional layer. This is because it is harder to extract meaningful features with a smaller number of layers.

Table 3.8: Improved custom model testing results.

Loss	Accuracy
0.0195	0.9955

Although this model has the smallest number of parameters, it got the best performance results on the testing set.

Table 3.9: Improved custom model classification report.

Class	Precision	Recall	F1-score	Accuracy
With mask	1.00	1.00	1.00	1.00
Without mask	1.00	1.00	1.00	

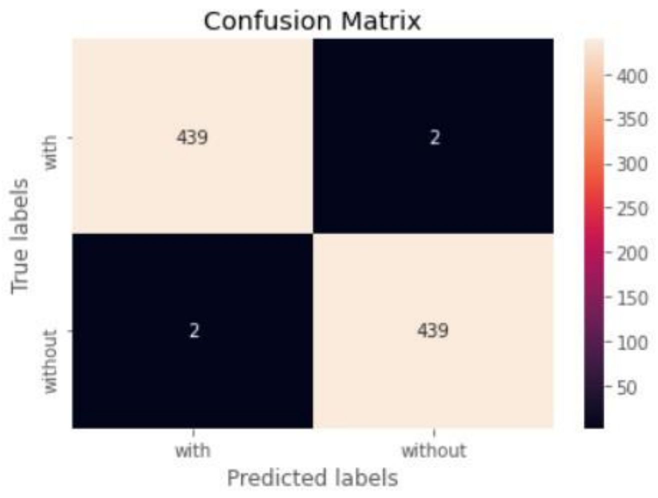


Fig. 3.9: *Improved custom model confusion matrix.*

The evaluation metrics results of this model shown in Table 3.6 are perfect. Also, the confusion matrix showed the smallest number of errors amongst all the models.

When testing the execution speed again, with a lot less parameters than before the improvement, we got the following result:

Execution time
0.82 seconds/prediction

The experiment yielded a 53% improvement on the execution speed while maintaining the level of accuracy.

3.3.4 Summary and Comparison

The following table represents a summary of the results obtained by each of the models proposed in this project:

Table 3.10: Summary and comparison of the models.

Model	Accuracy	Exec. time (s/prediction)
MobilenetV2	0.99	1.19
Efficientnet-B0	0.99	1.62
Improved custom model	1.00	0.82

From the table above we can clearly derive that the improved custom model is the best model since it has the highest accuracy score and the lowest execution time. This was not anticipated in the beginning of the experiment since.

3.3.5 Real-Time Detection and Classification

After training the models, we put them into practice to see how they perform in real-time using live video feed from a camera. We run a python script on a Raspberry Pi 3 Model B equipped with the Raspberry Pi Camera Module v2. The v2 Camera Module has a Sony IMX219 8-megapixel sensor and it can be used to take high-definition video, as well as stills photographs. It attaches via a 15cm ribbon cable to the CSI port on the Raspberry Pi.

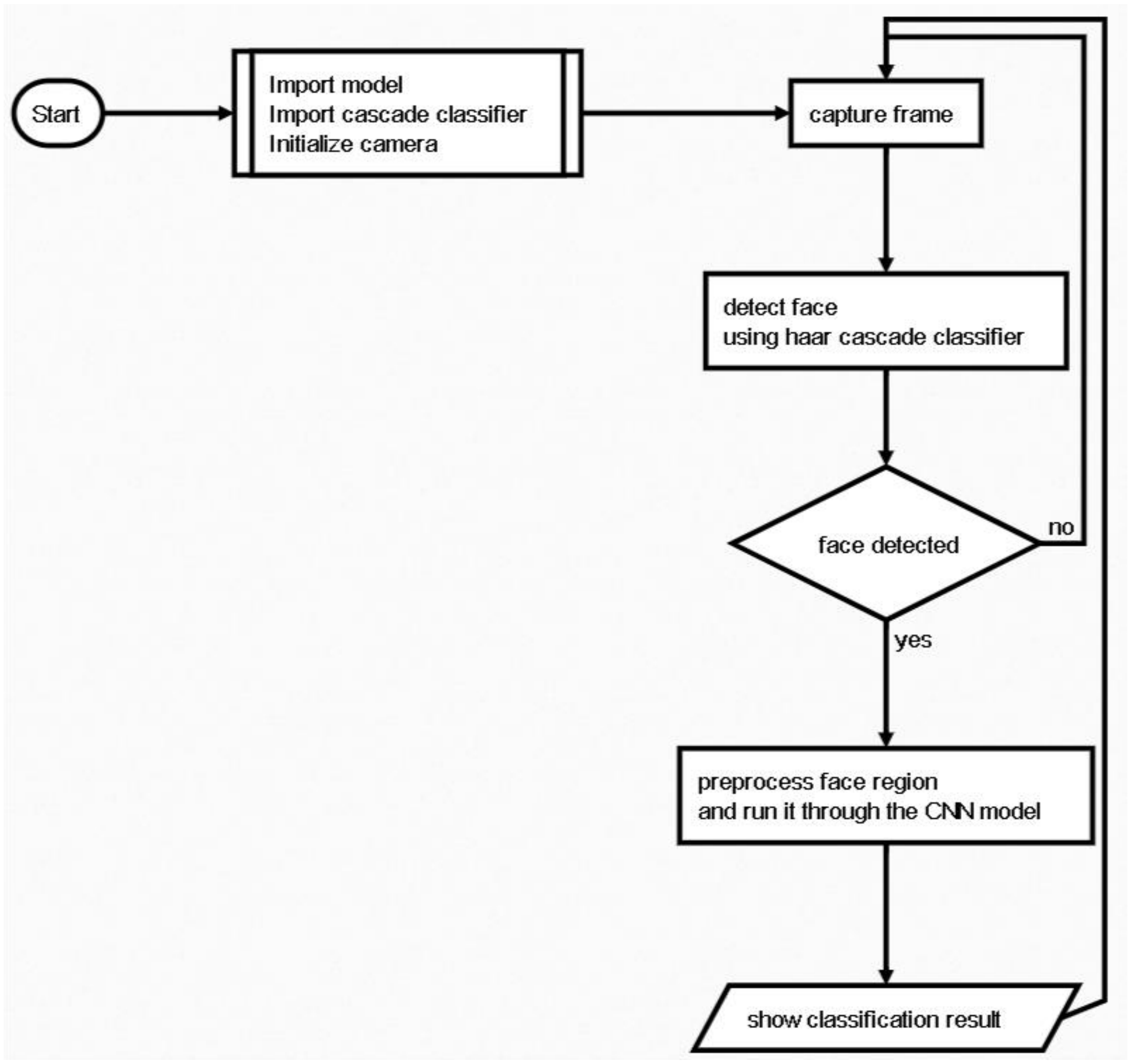

Start
Import model
Import cascade classifier
Initialize camera
capture frame
detect face
using haar cascade classifier
face detected
no
yes
preprocess face region
and run it through the CNN model
show classification result

Fig. 3.9: *Real-time classification flowchart.*

Figure 3.9 illustrates a flowchart of how our real-time detection script runs. After setting up the camera, a few tries were done using the three different proposed models. We recorded the real-time execution time of this algorithm in order to compare which model runs the fastest.

This real-time execution time metric was calculated by taking the mean execution time of 20 consecutive frame during the execution of the routine shown in Figure 3.9.

Table 3.11: real-time execution time of the models.

Model	Average face detection time (s)	Average real-time execution time (s)	Real-time difference (s)	Algorithm execution time
MobileNetV2	0.82	1.4	-0.22	2.22
EfficientNet-B0	0.85	2.14	+0.42	2.99
Custom model	0.89	1.3	+0.48	2.19

When testing the models, all three of them showed great accuracy in detecting whether the mask was worn or not. However, when testing the models for their real-time performance. They showed different results regarding their execution time.

As shown in Table 3.8 the average execution time for the models differs from the values obtained when we ran the average execution time metric presented at the beginning of the chapter.

The table above demonstrates how the execution time of the models changed when ran in a real-time environment. This is probably due to the system running different operating system utilities in parallel which impedes the execution of our models' prediction functions.

Also, the time it takes for the Haar cascade classifier to detect a face in a frame varies a lot from frame to frame. This is because the harder it is for the classifier to decide whether the sub-window contains a face or not the longer it will take the classifier to discard that candidate sub-window which means longer processing time. However, the detection time doesn't deviate much.

3.4 Conclusion

In this chapter, we presented the different evaluation metrics used to evaluate the proposed models. We then proceeded to show the results obtained in the different experiments that we carried.

We started first by training two CNN architectures loaded with pre-trained weights on the *ImageNet* dataset. Afterwards, we excluded the classification layers that come included in those architectures and added our own dense layers and trained those dense layers while freezing all the other layers. This technique training technique is called transfer learning.

We then proposed a CNN architecture and trained it on the same dataset. Although this proposed CNN model had less parameters than the other two, it ran slower than its pre-trained counterparts. In an attempt to improve its execution time, we removed the convolutional layer with the most parameters (the last convolutional layer of the architecture) and trained the model again. The execution time improved drastically after doing this modification.

Finally, we presented our real-time detection system that was intended for use on low-power CPUs. We demonstrated how it works using a flowchart and showed the results we got when running the system on our testing device.

General Conclusion and Future Work

The results obtained in the last chapter were very intriguing and not anticipated. We have seen that a simple 3 convolutional layer had better accuracy than the pre-trained models with transfer learning.

The main takeaway from the experiments is that even though mobile oriented CNN architecture such as MobileNetV2 and EfficientNet-B0 are advertised as the go-to architecture when running image classification on low computational power devices. they are often not the best options in term of efficiency; especially when the features to be learned are not very complex since in our case face mask detection doesn't require a lot of features to be extracted. We have seen that even a 3 convolutional layers network had a near perfect performance on the test dataset while running much faster than the other models. Our best model which was the custom model achieved an accuracy of 99.55% which a great accuracy score.

the most important takeaway from these experiments is that it would be much wiser when developing a CNN model for classification purposes to start with a simple custom built CNN architecture with a small number of convolutional and pooling layer and increment the number of layers until we meet the desired accuracy or until we exceed the execution time budget.

Future Work

Although all of the proposed models showed near perfect results, there is still room for improving the overall system.

A great way to improve the running time of these models is by converting them into the *TensorFlow-Lite* format. This format allows for faster execution speed on mobile and IoT devices.

Another great contribution would be to find a way to run Haar cascade classifiers faster on low computation power devices, since this task takes considerable amount of time during real-time detection and classification. This would make a great research endeavor.

Since, we improved our custom model by removing a convolutional layer and did not hurt the performance of the model, a good problem to work on would be to find a good trade-off between the number of parameters of the model and the accuracy. This will improve the execution time even further.

Lastly, since the act of wearing a face mask in public in not going away anytime soon, it would be a great contribution to work on developing a Haar cascade classifier that is trained on faces of people wearing face masks. This would improve the false negative rate of the classifier used in this project. It is also worth noting that an attempt to develop such a classifier was made during the completion of this project, but due to low computational resources and time constraints, the developed classifier had poor performance compared to the used/existing one.

Bibliography

[1] harkiran78, "Geeksforgeeks," 20 08 2019. [Online]. Available: https://www.geeksforgeeks.org/top-10-algorithms-every-machine-learning-engineer-should-know/. [Accessed 21 08 2021].

[2] Wikipedia, "Wikipedia," 12 07 2021. [Online]. Available: https://en.wikipedia.org/wiki/Boosting_(machine_learning). [Accessed 24 08 2021].

[3] OpenCV, "Cascade Classifier," [Online]. Available: https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html. [Accessed 24 08 2021].

[4] M. J. Paul Viola, "Rapid Object Detection using a Boosted Cascade of Simple," in *COMPUTER VISION AND PATTERN RECOGNITION I-511- I-518 vol.1*, 2001.

[5] C. McCormick, "AdaBoost Tutorial," 13 12 2013. [Online]. Available: https://mccormickml.com/2013/12/13/adaboost-tutorial/. [Accessed 25 08 2021].

[6] Sirakorn, "Wikipedia," 14 01 2020. [Online]. Available: https://en.wikipedia.org/wiki/Boosting_(machine_learning)#/media/File:Ensemble_Boosting.svg. [Accessed 25 08 2021].

[7] G. Tur, "Research Gate," 03 2010. [Online]. Available: https://www.researchgate.net/figure/AdaBoost-algorithm-for-the-binary-classification-task_fig7_224567440. [Accessed 25 08 2021].

[8] A. Wolfewicz, "Levity," 20 07 2021. [Online]. Available: https://levity.ai/blog/difference-machine-learning-deep-learning. [Accessed 28 08 2021].

[9] A. Bouhamadouche, "Emotion Detection Implementation using SVM and Embedding-LSTM," Boumerdes, 2020.

[10] R. E. Schapire, "The Boosting Approach to Machine Learning: An Overview. In: Denison D.D., Hansen M.H., Holmes C.C., Mallick B., Yu B. (eds). Lecture Notes in Statistics, vol 171. Springer. https://doi.org/10.1007/978-0-387-21579-2_9," in *Nonlinear Estimation and Classification*, New York, NY, 2003.

[11] IBM, "Deep Learning," IBM, 01 05 2020. [Online]. Available: https://www.ibm.com/cloud/learn/deep-learning. [Accessed 20 08 2021].

[12] B. M. REZIG Warda, "GPU-Based COVID-19 and other Lung Infections Detector from Chest X-Rays using Deep Convolutional Neural Networks.," Boumerdes, 2021.

[13] S. Saha, "Towards Data Science," 15 12 2018. [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. [Accessed 28 08 2021].

[14] S. A. a. A. Amidi, "Stanford University, Convolutional Neural Networks cheatsheet," [Online]. Available: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks. [Accessed 29 08 2021].

[15] J. Brownlee, "Machine Learning Mastery," 20 12 2017. [Online]. Available: https://machinelearningmastery.com/transfer-learning-for-deep-learning/. [Accessed 02 09 2021].

[16] I. B. V. e. al., "Face mask detection using MobileNet and Global Pooling Block," in *IEEE 4th Conference on Information & Communication Technology (CICT) DOI: 10.1109/CICT51604.2020.9312083*, Chennai, India, 2020.

[17] G. W. e. al., "Masked Face Recognition Dataset and Application," Guangcheng Wang, Wuhan, China, 2020.

[18] J. Mohajon, "Towards Data Science," 29 05 2020. [Online]. Available: https://towardsdatascience.com/confusion-matrix-for-your-multi-class-machine-learning-model-ff9aa3bf7826. [Accessed 11 09 2021].

[19] C. o. D. C. a. Prevention, "Covid Data Tracker Weekly Report," 08 2021. [Online]. Available: https://www.cdc.gov/coronavirus/2019-ncov/covid-data/covidview/index.html. [Accessed 20 08 2021].

[20] "World Health Organization," 01 05 2021. [Online]. Available: https://www.who.int/emergencies/diseases/novel-coronavirus-2019/advice-for-public/myth-busters. [Accessed 20 08 2021].

[21] K. Maladkar, "Analytics India Magazine," 25 01 2018. [Online]. Available: https://analyticsindiamag.com/convolutional-neural-network-image-classification-overview/. [Accessed 29 08 2021].

[22] N. Donges, "Built In," 23 07 2021. [Online]. Available: https://builtin.com/data-science/gradient-descent. [Accessed 31 08 2021].

[23] C.-F. Wang, "Towards Data Science," 04 08 2018. [Online]. Available: https://towardsdatascience.com/whats-the-difference-between-haar-feature-classifiers-and-convolutional-neural-networks-ce6828343aeb. [Accessed 02 09 2021].

[24] "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Python_(programming_language). [Accessed 08 09 2021].

[25] "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Keras. [Accessed 08 09 2021].