University of Science and Technology POLITEHNICA Bucharest

Faculty of Electronics, Telecommunications and Information Technology

Facial detection and recognition application

Diploma thesis

submitted in partial fulfillment of the requirements for the Degree of Engineer in the domain *Electronics and Telecommunications*, study program *Technologies and Communications Systems*

Thesis Advisor

Conf. dr. ing. Ionuţ PIRNOG

Student

Robert-Alin ILIE

Year 2024

University "Politehnica" of Bucharest
Faculty of Electronics, Telecommunications and Information Technology
Study program

Anexa 1

DIPLOMA THESIS

of student **ILIE D. Robert-Alin, 441G**

1. Thesis title: Aplicație de detecție și recunoaștere facială

2. The student's original contribution will consist of (not including the documentation part) and design specifications:
Se va implementa o aplicație de detecție și recunoaștere a fețelor din imagini și secvențe video. Aplicația se poate implementa în Matlab, C , Python, Java. Se pot folosi librării specifice și algoritmi dedicați prelucrării imaginilor/video: OpenCV, YOLOv4, Pytorch, Tensorflow, Python Tesseract, etc..

3. Academic courses the thesis is based on::
PDS; POO; TCSM

4. Thesis registration date: 2024-01-29 06:31:19

Thesis advisor(s),
Conf.Dr.Ing. Ionuţ PIRNOG

Student,
ILIE D. Robert-Alin

Departament director,

Dean,
Prof. dr. ing. Mihnea UDREA

Validation code: **90e6c0ec2e**

Annex 5

Statement of Academic Honesty

I hereby declare that the thesis *"Facial detection and recognition application"* submitted to the Faculty of Electronics, Telecommunications and Information Technology in partial fulfillment of the requirements for the degree of Engineer in the domain *Electronics and Telecommunications* study program *Technologies and Communications Systems* is written by myself and was never before submitted to any other faculty or higher learning institution in Romania or any other country.

I declare that all information sources I used, including the ones I found on the Internet, are properly cited in the thesis as bibliographical references. Text fragments cited "as is" or translated from other languages are written between quotes and are referenced to the source. Reformulation using different words of a certain text is also properly referenced. I understand plagiarism constitutes an offence punishable by law.

I declare that all the results I present as coming from simulations or measurements I performed, together with the procedures used to obtain them, are real and indeed come from the respective simulations or measurements. I understand that data faking is an offence punishable according to the University regulations.

Bucharest, *6/25/2024*

Robert-Alin ILIE

(student's signature)

Content

List of figures

List of tables

Introduction

Motivation of the work

Describing the content of images is a relatively easy task for humans and can be done with reasonable accuracy from a very young age. Humans can recognize, distinguish, and describe many categories of objects, scenes, or images, even with various external factors such as changes in lighting, orientation, and posture, which make this task incredibly difficult for machine learning algorithms.

In the context of artificial vision, describing images refers to the automatic generation of essential information associated with digital images. It involves not only recognizing the objects in these images but also describing their attributes, as well as the relationships and interactions between them, all expressed textually in natural language that is syntactically and semantically correct.

In recent years, there have been spectacular advances in deep neural networks, leading to remarkable progress in improving the quality of generated descriptive texts. Convolutional neural networks have been used to obtain essential representations of image features, which are then translated into textual descriptions. Automatic facial recognition has attracted a lot of attention recently due to its wide applicability in various facial analysis tasks. This technology is very important for different real-world applications, such as social understanding on social media platforms, identity verification, video surveillance, crowd behavior analysis, online advertising, product recommendation, and many others. Over time, many studies have been conducted on facial recognition, but with the success of deep learning technologies, recent works specifically use deep neural networks.

The development of this application is not possible without a considerable database. Thus, the increase in the number of images uploaded online has favored the construction of large labeled datasets and, implicitly, applications that recognize human features.

My interest in this current field led me to develop an application specialized in facial recognition. In a world where people spend a large part of their time online, this application aims to reduce barriers in human-machine interaction and improve the user experience through personalized suggestions. By automatically recognizing faces, users can receive relevant recommendations in various areas of interest. Automatic facial recognition has become increasingly important for a growing number of applications, especially with the expansion of social platforms and social media.

The objective of the work

Taking into account the importance of recognizing a human portrait in the current context, I have implemented an application to recognize human faces in photos and videos. For the successful implementation of the application, the following objectives were met:

1. Obtaining a sufficiently large database that includes positive images (those containing human faces) and negative images (those not containing human faces).

2. Designing convolutional neural networks and training the model to recognize human faces.

3. Evaluating the results and performance based on how the values of the hyperparameters and network architecture are varied.

4. Demonstrating the functionality of the application using a custom graphical interface.

Summary of the work

Chapter 1 presents a brief introduction to the theoretical concepts used in the development of the paper. Convolutional neural networks are the main topic of this chapter, accompanied by information about activation functions, optimization algorithms, and transfer learning.

Chapter 2 covers the main technologies and the purpose for which they were used in the paper. Specifically, the technologies presented are: the Python programming language, the Keras, TensorFlow, and OpenCV libraries.

Chapter 3 is dedicated to the implementation of the application, starting with preprocessing and cleaning the dataset. This chapter describes the architectures of the constructed networks, their operation, and the main components. Additionally, the training and testing methods are presented.

Chapter 4 presents the results obtained from training the four models, as well as some experiments conducted to optimize performance. The functioning of the graphical interface is also described to showcase the results.

Chapter 1 Theoretical notions

1.1 Artificial neural networks

1.1.1 Overview

A neural network, or artificial neural network, is a type of computing architecture that is based on a model of how a human brain functions — hence the name "neural."

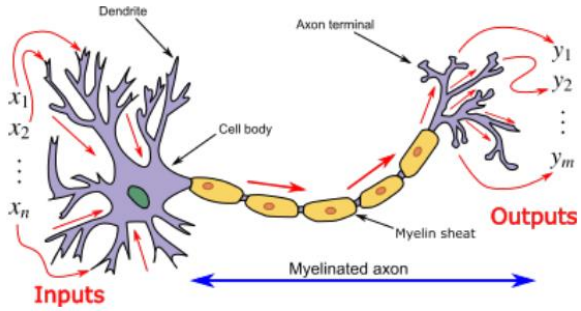

Figure 1.1 – The biological neuron[1]

Neural networks are made up of a collection of processing units called "nodes." These nodes pass data to each other, just like how in a brain, neurons pass electrical impulses to each other. Neural networks are used in machine learning, which refers to a category of computer programs that learn without definite instructions. Specifically, neural networks are used in deep learning — an advanced type of machine learning that can draw conclusions from unlabeled data without human intervention. For instance, a deep learning model built on a neural network and fed sufficient training data could be able to identify items in a photo it has never seen before [2]. Neurons are arranged in layers, where the output of each layer forms the input to the next layer. At the highest level, there are 3 types of layers, sequentially connected to each other:

- The input layer

- The hidden layers

- The output layer

The depth of the network is the number of layers, and the size of the network is the total number of neurons.

The main characteristic of these networks is their ability to learn from examples, using previous experience to improve their performance and make it available for future use. At the input of the network, training examples are provided, which circulate through the network starting from the input layer and moving through the hidden layers to the output layer. This transmission of information is known as forward propagation through the network. Characteristic of this type of artificial neural network is that a neuron receives signals only from neurons in previous layers. Feedforward networks perform data processing sequentially, on each layer, one after another, using activation functions that are differentiable (whose output smoothly transitions from negative to positive rather than abruptly), allowing for training.

This step is followed by a backward propagation stage, aiming to adjust the weights of all connections. The error calculated at the output of the network is sent back from the higher layers to the lower layers, in order to be minimized. It is essentially an iterative process of error optimization,

aiming to decrease the difference between the expected output and the obtained output in each iteration. This difference is estimated by a cost function. In this way, the most relevant features and outputs are highlighted to achieve the desired output. Traversing through a complete forward-backward process for one training example is called an iteration, and going through all training examples is called an epoch. Training is done over multiple epochs to minimize the error.

There are three adaptation paradigms based on: supervised learning, unsupervised learning, and reinforcement learning. In supervised learning, each piece of information transmitted to the model during training is a pair consisting of the input object, also known as the sample, along with the corresponding label or output value. The model establishes a mapping between a given input and the corresponding output based on what it has learned from the labeled training data.

In the case of unsupervised learning, the training data does not come with labels. The goal of these algorithms is to divide the data into "clusters" or groups, based on their attributes through an internal competitive mechanism. Members of such a group must be as similar to each other as possible, and the distance between clusters must be as large as possible.

Reinforcement learning does not benefit from training data accompanied by labels like supervised learning; instead, the data comes with a binary response providing qualitative information (good or bad response) about how well the network has achieved its goal. Thus, the network will know that it has made a mistake but not by how much. The network will be penalized when it produces the same undesired result, and the tendency to repeat mistakes will decrease.

In terms of architecture, artificial neural networks are divided into 2 main categories: feedforward neural networks, which transmit information in one direction (only from input to output), and recursive neural networks, in which information is transmitted from the current time to the past or future time.

1.2 Convolutional neural networks

In deep learning, a convolutional neural network is a class of deep neural networks, most commonly applied to analyze visual imagery. The CNN architecture uses a special technique called Convolution instead of relying solely on matrix multiplications like traditional neural networks. Convolutional networks use convolution, which in mathematics is a mathematical operation on two functions that produces a third function that expresses how the shape of one is modified by the other [3]. Since the foundation of convolutional networks is the use of images as input, one of the main distinctions is the arrangement of neurons in three dimensions: the input's spatial dimensions (height and breadth) and depth. Depth is the third dimension of the activation volume, not the total number of layers in the network. The depth of an RGB color image is three, signifying the three primary colors—red, green, and blue. A primary distinction in convolutional networks is that each layer's neurons are only partially connected to their predecessor's neurons. In practical terms, this means that an input volume with height, width, and depth of 64x64x3 will result in a narrow final output layer with dimensions of 1x1xn, where n is the number of potential classes.

Convolutional neural networks are distinguished from other neural networks by their superior performance with image, speech, or audio signal inputs. They have three main types of layers, which are:

- Convolutional layer
- Pooling layer
- Fully-connected layer

1.2.1 Convolutional layer

The convolutional layer is an important part of convolutional neural networks, with its parameters mostly focused on the utilization of nuclei, also known as filters. The core is a reduced-size two-dimensional matrix of weights that is responsible for identifying recurring patterns in the input data matrix.

The scalar product, which is the result of multiplying the core with a section of the input data matrix that is the same size as the core, is a mathematical operation that yields a single, distinct value. Employing a lower core size compared to the input matrix enables the possibility of performing multiple multiplications at various input locations. The core can identify the target qualities in any position by repeatedly and systematically applying itself to the full input image. The attribute in question is referred to as translation invariance, which has significance due to the shift in focus from the trait's location to its mere presence or absence.

By repeatedly applying the core to the input matrix, we generate scalable products that reflect the values of a two-dimensional matrix. The output matrix generated in this procedure, which represents a two-dimensional arrangement of characters, is referred to as the character map. The dimensions of this map are directly proportional to the number of iterations the filter does to traverse the whole input matrix.

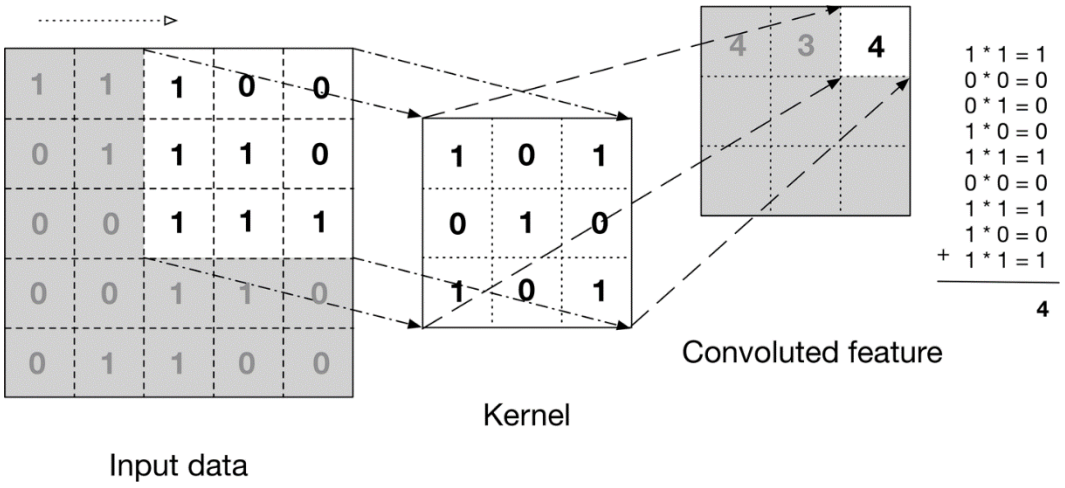

Figure 1.2 – Getting the character map[4]

Convolutional layers optimize the output, which allows them to drastically reduce pattern complexity. The number of neurons in the convolution layer allows the depth of the output volume to be manually adjusted. This significantly reduces the amount of neurons in the convolutional layers, which impacts the pattern's capacity to identify attributes.

1.2.2 Pooling layer

The Pooling Layer is a crucial component in convolutional neural networks that reduces the spatial size of the convolutional feature, reducing computational power and enabling the extraction of rotational and positional invariant features. It performs a down sampling operation, reducing the in-plane dimensionality of feature maps, introducing translation invariance to small shifts and distortions, and reducing the number of learnable parameters. Pooling operations, like convolution operations, include filter size, stride, and padding. It is often inserted between successive convolutional layers to progressively reduce the spatial size of the representation, reducing the number of parameters and computation in the network, and controlling overfitting. There are two

types of Pooling: Max Pooling and Average Pooling. Max Pooling returns the maximum value from the kernel portion, while Average Pooling returns the average of all values. The pooling layer operates independently on every depth slice of the input and resizes it spatially using the max () operation [5].

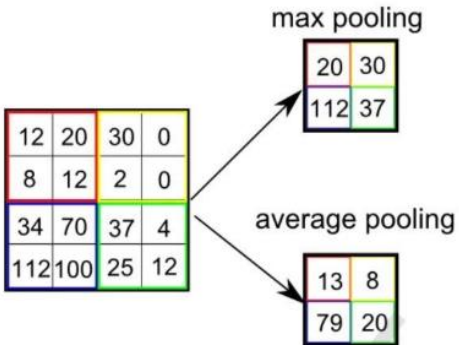

Figure 1.3 – Types of pooling operations

1.2.3 Fully-connected layer

The convolutionary portion of the network is connected to the final classifier that generates the output through the fully connected end layer (Dense). Every activation unit in the following layer is coupled to every input in this layer. This pseudo-strate's function is to convert the three-dimensional output that results from convolutional operations into a one-dimension data type so that it can be fed into the classifier. It does not actually modify the values. The layer offers a number of probabilities equal to the number of classes that were initially created, based on the input vector.

1.3 Activation function

An activation function in an artificial neural network is a function that maps the inputs of a node to their corresponding outputs. The determined weighted sum of each incoming connection for each node in the layer is then provided to an activation function. The activation function applies a non-linear change to the total. The activation function plays a crucial role in determining the neural network's capacity to reach optimal parameters through convergence. Occasionally, the activation function can impede the convergence of the neural network.The selection of an activation function considers both the intended performance and the nature of the problem that needs to be addressed.

1.3.1 Sigmoid

The sigmoid activation function is a nonlinear function that takes values only within the range (0,1). It is especially used for models that predict a probability. A sigmoid function placed as the last layer of a learning model can serve to convert the model result into a probability score, which can be easier to work on and interpret.

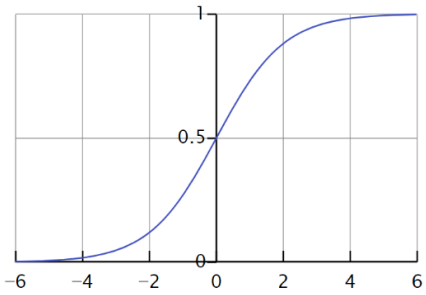

Figure 1.4 - Sigmoid function

$$f(x) = \frac{1}{1 + e^{-x}} \qquad (1.1)$$

1.3.2 Hyperbolic tangent

The hyperbolic tangent activation function is similar to the sigmoid function, has a nonlinear character, but the range of values is, this time, in the range [-1, 1]. Thus, the larger the input (the more positive), the closer the output value will be to 1, while the smaller the input is (the less negative), the nearer will the outputs be to -1. Hyperbolic tangent is preferred to the sigmoid function because it is centered at zero.

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \qquad (1.2)$$

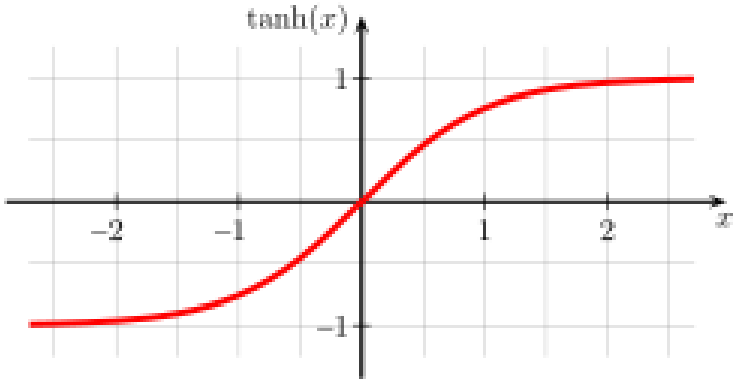

Figure 1.5 – Hyperbolic tangent function

1.3.3 ReLU

Because it has an infinite range of values, the ramp function, also known as the linear rectification function, is the most widely used activation function. If the input value is negative, ReLU returns 0; otherwise, it returns the input. The primary benefit of this function is that it selectively activates neurons based on input values; that is, it does not activate neurons in response to negative values. The ramp function is far more computationally efficient than the sigmoid function and the hyperbolic tangent function since it only activates a specific number of neurons.

$$f(x) = max(0, x) \qquad (1.3)$$

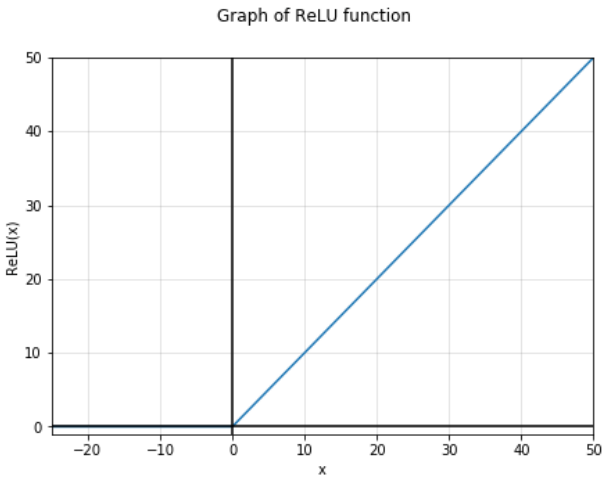

Figure 1.6 – ReLu function

1.3.4 Softmax

The Softmax function is often described as a combination of multiple sigmoids, it takes values between 0 and 1. It is used in multiple class classification problems because it returns a probability vector. Each value in the output of the softmax function is interpreted as the probability of membership for each class.

$$f(z_i) = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}} \qquad (1.4)$$

1.4 Optimization algorithm

Machine learning's primary goal is to train a model that performs well and yields the required outcomes. In order to reduce the cost function and optimize hyperparameters, optimization methods are preferred. Since the cost function describes the discrepancy between the estimated parameter's real value and the value produced by the model, it is crucial to reduce it.

1.4.1 Negative gradient

The most widely used iterative optimization technique for locating the cost function's local minimum is called negative gradient. Since the procedure is first order, it simply takes into account the first derivative when determining new parameter values by employing a constant step with respect to the current values.

Since this is the direction of the most abrupt drop, the idea behind this approach is to take values in the opposite direction to the gradient of the cost function at the present location. The learning rate is represented by the parameters η in the formula below, where the value of θ is modified based on the gradient value at the current point (ΔJ(θ)).

$$\theta = \theta - \eta \Delta J(\theta) \qquad (1.5)$$

As the algorithm iterates to a minimum of the cost function, the step size at each iteration is represented by the learning rate. Selecting the right rate might be difficult since a low value causes the process to converge slowly and take longer to reach the ideal parameter values that minimize losses. An excessive learning rate can impede convergence and lead to variations in the cost function, either approaching or above the minimum. The learning rates that are most in demand are: 0.001, 0.003, 0.01, 0.03, 0.1, and 0.3.

1.4.2 Adam

Adam [6] is an abbreviation for adaptive moment estimate. This is an optimization algorithm that seeks the existence of a separate learning rate for each parameter. The algorithm is an adaptive one, which means that individual learning rates are calculated for different parameters using estimates of the first and second gradient moment, based on previous weight values. The moment of a random variable is defined as the expected value of that variable at the power of n.

$$m_n = E[X^n] \qquad (1.6)$$

The first moment of the gradient is the average, and the second moment is the uncentric variance (the average does not decrease during the calculation of the variance). The gradient of the cost function of the neural network can be considered a random variable, as it is usually evaluated on

a small base of random data. The 2 moments are estimated using moving exponential averages, calculated on a gradient assessed on a small batch of data.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \qquad (1.7)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \qquad (1.8)$$

The usually chosen values for the hyperparameters of the β1, β2 algorithm are β1 = 0.91, β2 = 0.999, so that the vectors of the m and v moving averages are initialized with zeros at the first iteration. Since m and v are estimates of the first and second moment, the following property on expected values of moving averages is desired:

$$E[m_t] = E[g_t] \qquad (1.9)$$

$$E[v_t] = E[g_t^2] \qquad (1.10)$$

The property expressed above is not respected so estimators are not impartial. The vectors m and v are initialized with the value zero and their behavior tends toward zero. Corrections are applied to the first and second moments so that the value of the estimator is the desired one. This step is usually called bias correction. [7] The final formulas for the bias correction estimator for the first and second moment shall be as follows:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad (1.11)$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \qquad (1.12)$$

These moments are used to calculate the learning rate for each parameter. The Adam algorithm updates the weights as follows:

$$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \qquad (1.13)$$

1.5 Regularization methods

Over-training the model is the main issue that comes up when training a neural network. When an analysis fits a certain set of data very well but does not match with other data, it is said to be overtrained and is unable to accurately predict other unknown data. This idea arises when the trained model contains too many parameters or keeps too many characteristics and specifics from the input data set, making it overly complex.

Regularization approaches, which make small adjustments to the learning process to improve the model's ability to generalize unfamiliar input, are employed to lessen overtraining of the model. Furthermore, by eliminating extraneous data, regularization approaches streamline the model and boost its ability to generalize.

1.5.1 L1 & L2 type regularizations

Adding a term to the cost function called a regularization term is the most common regularization method. Following the inclusion of the regularization term, the weight values fall, suggesting a decrease in both the neural network's complexity and the model's overtraining.

Using L2 regularization, a neural network with parameters θ and a cost function L(θ) is encouraged to use all of its inputs in a lower proportion rather than using some entries more frequently than others.

Because L2 adjustment pushes weights to reduce to zero (but not down to zero), it is often referred to as weight breakdown.

$$L(\theta)_{nou} = L(\theta) + \frac{\lambda_2}{2} \cdot \|\theta\|_2^2 \qquad (1.14)$$

By encouraging the network to set the weights of less significant features to zero, the L1 adjustment helps the network to completely remove some features. We penalize the weights' absolute values in this way. For this reason, this method works well for compressing patterns.

$$L(\theta)_{nou} = L(\theta) + \lambda_1 \cdot \|\theta\|_1 \qquad (1.15)$$

λ_1, λ_2 are adjustment terms. Also, L1 / L2 regulation is often implemented.

1.5.2 Regularization by dropout

The new methods showed that Dropout [8] is a regularization approach that compliments the other methods (L1, L2, or max norm) and is very effective and simple. Dropout is introduced during training by randomly chosen neurons that are disregarded. Randomly, they are "dropped-out" (Fig. 1.7). This indicates that on the forward pass, their contribution to the activation of downstream neurons is momentarily eliminated, and on the backward trip, the neuron does not get any weight updates.

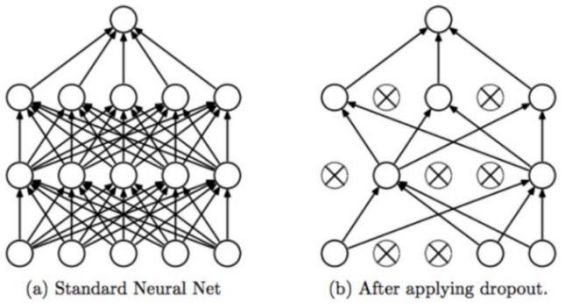

Figure 1.7 – Dropout example[9]

Neuron weights become embedded in their network context when a neural network learns. There is some specialization since the weights of the neurons are adjusted for particular traits. When specialization is overdone, it might lead to a delicate model that is overly dependent on the training set of neurons.

1.6 Data augmentation

Through the creation of modified data from pre-existing data, a technique known as data augmentation can be used to artificially expand the size of a learning data set. Both the issue of overtraining the model and the issue of insufficiently large data sets are resolved by this method. Depending on the demands of the user, this technique might greatly expand the data set by a specific percentage (200%, 500%).

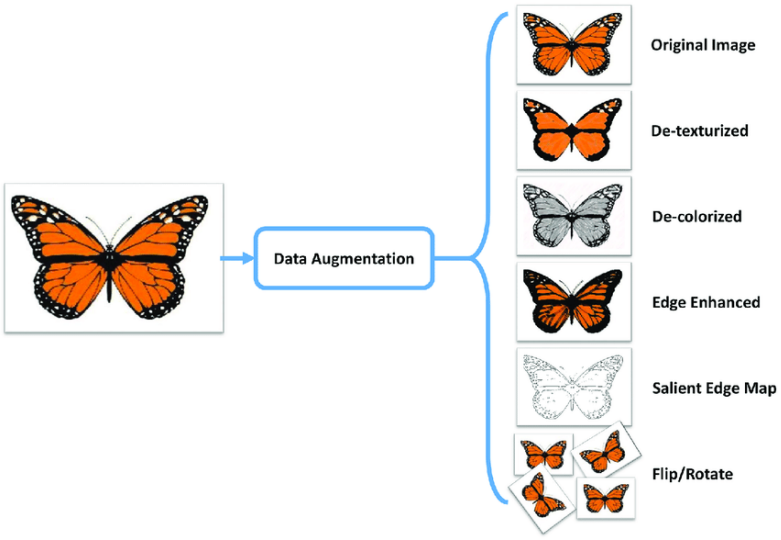

Figure 1.8 – Data augmentation example[10]

When it comes to image manipulation, transformations on data include a variety of operations like rotations, color changes, random deletion, image mixing, geometric transformations, adding Gaussian algae noise, twists, and more. This is because convolutional neural networks are capable of learning features that are independent of their location in images.

1.7 Transfer learning

A machine learning technique called transfer learning uses a model that has been trained for one task to serve as the foundation for another model that completes a related task.

Because developing neural network models for each sort of problem would have required a significant amount of time and computer resources, pre-trained models are employed for a wide range of tasks in this widely used method.

Conventional learning is discrete and solely task-based, meaning that model training and data sets are concentrated on a single issue. Nothing that can be applied to a different model is maintained. Knowledge (traits, weights, etc.) from previously trained models can be transferred and retained for the training of subsequent models through the process of transfer learning. Using smaller data sets for models that inherit this information is a key benefit of transfer learning.

When the first solved problem has a complicated, elaborately labelled data set and the structures of the two problems are comparable, it is helpful to transfer the weights of one trained model to another new model. A few characteristics of the pre-trained models selected for this project are shown in table 1.1.

Model	Size	Conventional Precision	Parameters number	Depth
MobileNet	16 MB	0.704	4,253,864	88
MobileNetV2	14 MB	0.713	3,538,984	88
VGG19	549 MB	0.713	143,667,240	26
InceptionV3	92 MB	0.779	23,851,784	159
ResNet50V2	98 MB	0.760	25,613,800	-
NASNetMobile	23 MB	0.744	5,326,716	-
DenseNet201	80 MB	0.773	20,242,984	201

Table 1.1 - The main features of pre-trained models

1.7.1 MobileNet

Time and accuracy are critical variables in real-world applications like object identification and image categorization. Therefore, in order to be employed in real-time applications, an extremely exact model that requires less complexity and a quicker calculation time is required. In order to address this issue, the MobileNet concept was developed. The model's use of a convolutional neural network with deep distinct convolutions is what makes it special. The network multiplier and the resolution multiplicator are the two parameters that make up the network. While the second parameter is used to lower the image resolution, the first parameter thins the grid uniformly by lowering the number of filters.

1.7.2 MobileNetV2

Based on the concepts of the earlier MobileNet model, MobileNetV2 uses segregated deep convolution as an extremely effective architectural building block. Two new features are added to the architecture by MobileNetV2, though: rapid connections between locks and linear locks between layers.

Across the board, MobileNetV2 models are often faster for the same accuracy. Specifically, the new models achieve higher accuracy while using twice as many processes and 30% fewer parameters than the MobileNet models. Moreover, because it utilizes a lot less memory than earlier versions, this model is appropriate for mobile applications.

1.7.3 VGG19

One of the most often used pre-trained models is VGG19 [11], which achieved an accuracy of 92.7% on one of ImageNet's largest datasets that included over 15 million images from 1000 different classes.

VGG19 is an extremely deep network with 16 weight layers, 13 convolutional layers the size of a 3x3 core, and fully connected layers. In order to do this, the VGG family of networks has shown improved performance with depth. These days, this architecture is frequently employed for the robust generic characteristics that are relevant for applications other than classification and are extracted after the final convolutional layer. This model's main drawback is that because of its tremendous complexity, it takes longer to get the necessary performance.

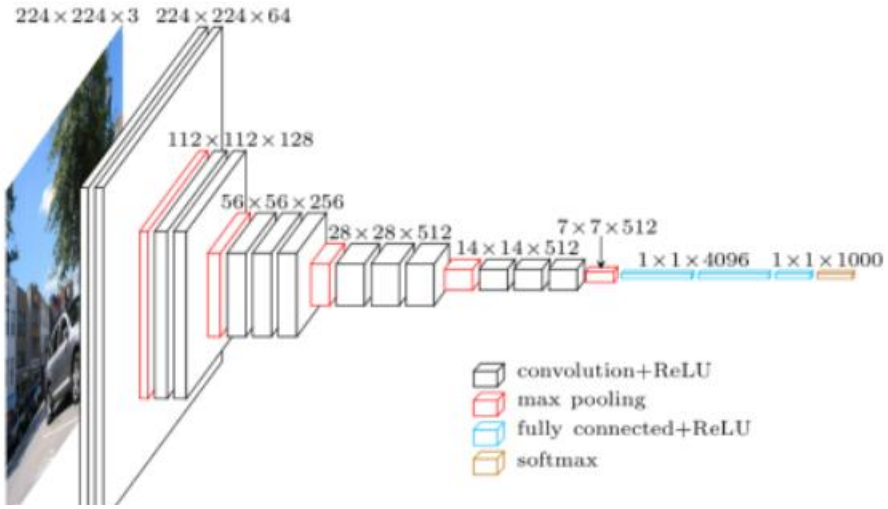

Figure 1.9 – VGG19 architecture[12]

1.7.4 ResNet

One of the newest but still very important architectures is ResNet [13], which addresses the issue of the steep drop of gradients as one advances through the network in the backward propagation phase. The design suggests an approach where a residual block is included, where an input is taken to be added to the output of a block of convolutional straws, for networks with many layers that are challenging to train. The so-called "shortcut" connections between these nodes have made it possible to build far deeper networks with up to 1000 layers. It was possible to reduce the amount of filter per layer and keep a comparatively modest number of traceable parameters by increasing the number of filters.

Chapter 2 Technologies used

2.1 Python

Python is referred to as an interpreted language, which simply means that the programmer can use it to run the program immediately without having to compile it into machine language instructions first. Because hardware can understand it, this makes it a competent language to be used with emulators or virtual machines based on the native code of an existent machine.

It's a high-level programming language used in intricate situations. To be more comprehensive and hence be used more frequently, high-level languages handle variables, arrays, objects, complex arithmetic, boolean expressions, and other abstract ideas in computer science.

Additionally, Python is regarded as a general-purpose programming language, indicating that it may be applied to various technologies and fields [14].

One of the key roles of Python in Artificial Intelligence is the support of various powerful libraries and frameworks. Interestingly, these libraries are specially designed for multiple AI and machine learning tasks. Also, they simplify the development process. To name a few libraries – TensorFlow, PyTorch, Keras, etc [15].

These libraries were managed by programs like Pip and Conda, which take care of downloading, installing, and verifying the presence of required dependencies. Installing and managing extra packages that are not included in the standard Python library is possible with Pip, the primary package manager for the Python programming language. The Pip manager, which is incredibly user-friendly and simple to use, makes it easier to access libraries.

Using virtual environments is made possible by the Python programming language. One technique that helps isolate dependencies for every Python project that is developed is the virtual environment. Libraries are not installed at the system level, but rather stored in a virtual environment within the project. Virtual environments offer high mobility and are helpful for projects involving multiple developers. As a result, there is no need to reinstall dependencies when moving the project to a different Python interpreter. Using the package management system and Conda environment, implementing the virtual environment is simple.

The Python programming language was used to handle and process the dataset, build the model, train it, test it, show parameters and results, make a graphical user interface, and find faces in photos and videos. Standard and specialized tools for artificial intelligence and machine learning were used to make all of this possible.

2.2 TensorFlow & Keras

TensorFlow [16] is an open-source library developed by Google primarily for deep learning applications. It also supports traditional machine learning. TensorFlow was originally developed for large numerical computations without keeping deep learning in mind. However, it proved to be very useful for deep learning development as well, and therefore Google open-sourced it.

TensorFlow accepts data in the form of multi-dimensional arrays of higher dimensions called tensors. Multi-dimensional arrays are very handy in handling large amounts of data.

TensorFlow works on the basis of data flow graphs that have nodes and edges. As the execution mechanism is in the form of graphs, it is much easier to execute TensorFlow code in a distributed manner across a cluster of computers while using GPUs.

Deep neural networks can be trained and run in a huge number of areas. Some of the most well-known uses are handwritten number classification, image recognition, word embedding, recurring neural networking, natural language processing, and simulations based on partial differential equations.

The best thing about TensorFlow for the growth of machine learning is that it abstracts things. The way that machine learning and deep learning encoded data used to be a lot more difficult before libraries were made. Instead of figuring out how to link the output of one function to the input of another or figuring out how to code the algorithm in detail, this library lets developers focus on the logic of their app.

Tensorflow serves as a high-level framework, but it can be further abstracted by using the Keras library. Keras [17] is a high-level, deep learning API developed by Google for implementing neural networks. It is written in Python and is used to make the implementation of neural networks easy. It also supports multiple backend neural network computation. Keras is relatively easy to learn and work with because it provides a python frontend with a high level of abstraction while having the option of multiple back-ends for computation purposes. This makes Keras slower than other deep learning frameworks, but extremely beginner-friendly. Another option available in Keras for training is to use a generator-type object. This would involve feeding the model fresh data at the start of each period. Generators split the data into smaller sets that are provided in accordance with the model because, typically, the complete dataset cannot be utilized within a single epoch due to limited resources. This method allows for the potential preprocessing of data sets prior to training, which adds diversity to the model and improves its generalizability.

Both libraries allow the acceleration of operations by using graphics processors that have much higher computing power than conventional processors.

Keras	TensorFlow
Typically used for smaller datasets.	Used for large datasets and high-performance models.
High-level API for ease of use.	Offers both high and low-level APIs.
Ideal for quick prototyping and experimentation.	Better suited for complex tasks.
Keras has a simple architecture and easy to use.	TensorFlow has a complex architecture and not easy to use.
Calling certain functions in an automatic mode at certain stages of training.	Allows quick model training and implementation, regardless of language or platform used.

Table 2.1 – Difference between TensorFlow and Keras [18]

Within this project, the Keras library (which is based on TensorFlow) played an important role in network implementation, model training and evaluation, as well as in data processing.

2.3 Matplotlib

Matplotlib is a Python-based plotting library that is intended to generate static, interactive, and animated visualizations. It is an essential instrument in scientific research, machine learning, and data science. It provides a diverse selection of plot types, such as scatter plots, line plots, bar plots, histograms, and pie charts. Matplotlib[19] is very flexible and customizable for creating plots. It does require a lot of code to make more basic plots with little customizations. When working in a setting where exploratory data analysis is the main goal, requiring many quickly drawn plots without as

much emphasis on aesthetics, the library seaborn is a great option as it builds on top of Matplotlib to create visualizations more quickly.

Matplotlib is an essential tool for machine learning (ML) due to its critical role in data visualization. It is employed to comprehend data, monitor models, and communicate results in a clear and efficient manner. Graphs and diagrams can be generated using Matplotlib, which aids in the identification of patterns, relationships, and anomalies in data. These visualizations are essential for the informed decision-making process regarding data preprocessing and the selection of the most suitable machine learning algorithms. Matplotlib is utilized for real-time monitoring, hyperparameter adjustment, blur adjustment, model interpretation, and the presentation of ML project results. Additionally, it is used for the presentation of results. It is indispensable in machine learning, as it enables the conversion of raw data into valuable and easily reacted-to information, thereby facilitating the optimization of models and the making of decisions.

Matplotlib is essential for data visualization and interpretation in machine learning projects. It provides versatile tools that facilitate data exploration, monitoring model performance, and communicating results. Thus, it significantly contributes to the success and efficiency of ML projects.

2.4 OpenCV

OpenCV (Open Source Computer Vision Library) is an open-source library dedicated to computerized image processing and vision. Originally developed by Intel, it quickly became one of the most used bookstores in the field of computer vision due to its extensive set of functionalities and efficient algorithms. OpenCV supports both static image processing and video streams, being extremely useful in object recognition, face detection, motion analysis, and more. OpenCV [20] is used by many applications, products, and research projects. It is often utilized for product inspection in factories, automated medical image analysis, security video surveillance, human-machine interfaces, and robotic vision. Among its use cases, it is also used for satellite mapping and autonomous vehicles. Its image processing capabilities enable tasks such as video stream processing, image stitching, or camera calibration.

2.4.1 OpenCV in Machine Learning

OpenCV in Machine Learning: OpenCV offers a multitude of functions that are indispensable in machine learning applications. It provides comprehensive tools for image processing, allowing developers to manipulate and analyze visual data effectively. These functions range from basic operations like image reading and writing to more complex tasks such as feature extraction, image transformations, and contour detection. By leveraging OpenCV's extensive library, machine learning practitioners can streamline the preprocessing of images, enhance model training, and improve the overall accuracy and performance of their algorithms.

Data Preprocessing: Data preprocessing is one of the most frequently employed OpenCV functions in machine learning. It is essential to transform and clean data in order to enhance the accuracy and performance of a machine learning model before it is trained. OpenCV facilitates the execution of functions such as image resizing, grayscale transformation, and image normalization.

Object Detection and Recognition: OpenCV offers a variety of algorithms for object detection and recognition. For example, Haar Cascade and LBP (Local Binary Patterns) are well-known methods for detecting features. These algorithms can be incorporated into machine learning models to enhance the detection and recognition capabilities of objects.

Data Visualization: In numerous machine learning projects, data visualization is essential for understanding the model's behavior and performance. OpenCV enables the real-time manipulation of images and prediction results, thereby facilitating the interpretation and adjustment of models.

OpenCV's extensive suite of functions makes it an invaluable tool in the realm of machine learning. From preprocessing and object detection to data visualization, OpenCV streamlines various stages of the machine learning workflow. Its robust capabilities allow practitioners to develop more accurate and efficient models, ultimately advancing the field of machine learning.

2.4.1 Haar Cascade

Haar Cascade [21] is a machine learning-based approach for object detection, introduced by Paul Viola and Michael Jones in their seminal paper in 2001. This method involves training a cascade function with a large number of positive and negative images to detect objects in images or video streams efficiently. The technique uses Haar-like features, which are simple rectangular features that compute the difference in intensity values between adjacent rectangular groups of pixels. These features are then used to build classifiers, which are applied in stages, allowing for rapid and accurate detection of objects such as faces, eyes, and other objects in real-time applications.

In this project, Haar Cascade is used for real-time face detection in both static image files and in the video stream and live video captured from a webcam

For facial detection in static images, after making a prediction using a neural network model to confirm the presence of a face with a sufficiently reliable level, the Haar Cascade classifier is used to locate the face. The image is first converted to gray tones, because the Haar Cascade algorithm requires gray-tones images for processing. If faces are detected, they are framed with rectangles on the original image.

For face detection in video stream and live video, the process is similar, but applied to each frame of the video stream.. After predicting the presence of the face with the neural network model, the Haar Cascade classifier is applied to the current frame of the video. Detected faces are highlighted with rectangles and a label indicating "Face Detected" is displayed.

Thus, Haar Cascade is an essential element of this code for face detection in both static images and real-time video, allowing effective identification of facial regions, which facilitates processing, visualization and subsequent interaction in the context of machine learning.

2.5 Tkinter

Tkinter [22] is the standard library for developing graphical user interfaces (GUI) in Python. It is included in the standard distribution of the Python language, which makes it easily accessible and usable without requiring the installation of additional packages. Tkinter is a port to the Tcl/Tk graphics library and provides a simple and efficient method for developing GUI applications.

One of the main advantages of Tkinter is its simplicity. It is easy to learn and use, making it ideal for beginners. Tkinter offers a variety of graphical components, called widgets, which include buttons, tags, text fields, checkboxes, and more. These widgets can be combined and arranged in various ways to create complex and intuitive user interfaces.

Customizing the appearance of the components is another strength of Tkinter. Developers can change the widget properties, such as color, font, size, and style, to create attractive and user-friendly interfaces. Tkinter supports the event-based programming model, allowing you to associate callback functions with various user events, such as mouse clicks, key presses, and other interactions. This programming model facilitates the development of interactive and responsive applications.

Chapter 3 Implementation Method

3.1 The dataset

In order to realize our facial recognition project, we chose to use a varied and balanced dataset to train a robust and accurate model. Our dataset consists of two distinct collections of images: one containing positive images, representing human faces, and the other containing negative images, which include various objects and animals.

The positive images were selected from the renowned LFW (Labeled Faces in the Wild) dataset, which is widely used in the research community to evaluate facial recognition algorithms. LFW is a vast and diversified dataset, containing images of faces of real people, captured in natural conditions and in various scenarios. For this project, we extracted a subset of 6000 images from the LFW, ensuring that they are representative and cover a wide range of facial variations, including different expressions, illuminations and angles. This diversity is essential for training a model capable of recognizing faces in various real-world conditions.

Figure 3.1 – Positive images

The negative images, on the other hand, come from the Natural Images dataset, which contains images of common objects and animals. We have selected a total of 6000 negative images, which include categories like cats, dogs, cars, planes and fruits. The choice of these categories has been made strategically to ensure sufficient diversity in the negative data set, thereby helping to train a model that can correctly differentiate between human faces and other types of images. This diversity helps reduce the rate of false positive, ensuring that our model does not misclassify non-human objects as faces.

Figure 3.2 – Negative images

3.2 Data preprocessing

In the process of preparing the dataset [23], we allocated the images into three sub-sets: training, validation and testing. For each subset, weined a balance between positive and negative images to prevent any form of bias during the model training. So, for the training set, we selected 60% of the total images, which translates into 7200 images (3600 positive and 3600 negative). The validation set contains 20% of the images, i.e. 2400 images (1200 positive and 1200 negative), and the test set includes the remaining 20% of all 2400 pictures (1200 negative and 1200 positive).

The balanced and diversified distribution of images in the three subsets is crucial for the correct evaluation of the model performance. The validation set is used to monitor performance during training and adjust hyperparameters, while the test set serves the final evaluation of the model, providing an objective estimate of its ability to generalize on invisible data.

In conclusion, our dataset is well structured and diversified, consisting of 6000 positive images and 6000 negative images, coming from reputable sources. This structure ensures the right balance and diversity that are essential for the training and evaluation of a robust and performing facial recognition model.

Image preprocessing also involved data augmentation steps for the training set. Data augmentation is an essential technique in machine learning that helps increase the diversity of the training set without collecting more data. This includes operations such as rotation, translation, zoom and image reversal, thereby helping to improve the model's ability to generalize and cope with variability in input data.

For validation and testing, we only applied image normalization, scaling pixel values to the range [0, 1] to ensure consistency of network inputs. Thus, these sets were not augmented to maintain data integrity and to provide a real evaluation of the model performance. Also, all images were resized to 224x224 pixels, which is the standard input size for many pre-trained convolutional neural networks, including VGG16 and ResNet50.

```python
# Augmentation configuration for training
train_datagen = ImageDataGenerator(
    rescale=1. / 255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)
```

Figure 3.3 – Augmentation

Through these data preprocessing procedures, we have ensured that our facial recognition model is trained, validated and tested on a balanced and diversified data set. This methodology contributes to obtaining a robust model, capable of performing well under varying conditions and generalizing correctly on new, previously unseen data. Resizing images to 224x224 pixels guarantees compatibility with the neural network architectures used, thus facilitating efficient training and high model performance.

3.3 The model

The model used for facial recognition in this project is a Convolutional Neural Network (CNN) built using the TensorFlow and Keras framework. The model architecture is designed to extract and learn relevant features from images, thus facilitating effective face recognition.

The model starts with an input layer that supports images with 224x224 pixels and 3 color channels (RGB). The image size was chosen to be compatible with many pre-trained convolutional neural network architectures, such as VGG16 and ResNet50, thus facilitating the use of learning transfer if necessary.

The model architecture includes the following layers:

1. Convolution Layer 1:
   This layer applies 32 3x3 filters to the input image.
   The ReLU activation function is used to introduce non-linearities, allowing the model to learn complex functions.
   This layer extract basic features such as edges and textures from images.
2. Pooling Layer 1:
   A 2x2 max-pooling layer is used to reduce the spatial dimension of the extracted features while retaining the essential information.
   Pooling helps reduce computational complexity and prevent overfitting.
3. Convolution Layer 2:
   This layer applies 64 3x3 filters, continuing to extract more complex features.
   The ReLU activation function is also used here.
4. Pooling Layer 2:
   Another layer of max-pooling 2x2 is applied to reduce again the spatial dimension of the features.
5. Convolution Layer 3:
   This layer applies 128 3x3 sized filters, extracting even more detailed features from the images.
   The ReLU activation function is used.

6. Pooling Layer 3:
   Max-pooling 2x2 is used to reduce the size of the features.
7. Convolution Layer 4:
   This layer applies another 128 3x3 filters, continuing to extract complex features.
   The ReLU activation function is used.
8. Pooling Layer 4:
   A last layer of max-pooling 2x2 is applied to obtain an additional reduction in the size of the features.
9. Layer of Flatten:
   This layer transforms the 2D characteristics matrix into a 1D vector, preparing the data for dense layers.
10. Layer of Dropout:
    A drop-out layer with a rate of 0.5 is used to prevent overfitting, randomly disabling 50% of neurons during training.
11. Dense Layer:
    A fully connected layer of 512 neurons is applied using the ReLU activation function.
    L2 adjustment is used to penalize high coefficients and to prevent overfitting.
12. Layer of Dropout:
    Another layer of dropout with a rate of 0.5 is used to prevent overfitting.
13. Output Layer:
    Final layer fully connected with 1 neuron and sigmoid activation function, which produces a probability of belonging to the positive (face) or negative (non-face) class

The model is compiled using the binary_crossentropy loss function, suitable for binary classification problems. The Adam optimizer is used with a learning rate of 0.0001 to adjust the model weights during training. The L2 regulation method is applied to penalize high coefficients, helping to prevent overfitting and improve model generalization.

Layer type	**Configuration**	**Output shape**	**Parameters**
Input	Images (224, 224, 3)	(224, 224, 3)	0
Conv2D	32 filters, 3x3 kernel, ReLU	(222, 222, 32)	896
MaxPooling2D	2x2 kernel, stride=2	(111, 111, 32)	0
Conv2D_1	64 filters, 3x3 kernel, ReLU	(109, 109, 64)	18496
MaxPooling2D_1	2x2 kernel, stride=2	(54, 54, 64)	0
Conv2D_2	64 filters, 3x3 kernel, ReLU	(52, 52, 128)	73856
MaxPooling2D_2	2x2 kernel, stride=2	(26, 26, 128)	0
Conv2D_3	64 filters, 3x3 kernel, ReLU	(24, 24, 128)	147584
MaxPooling2D_3	2x2 kernel, stride=2	(12, 12, 128)	0
Flatten	-	18432	0
Dropout	Rate = 0.5	18432	0
Dense	512 neurons, ReLU, L2(0.001)	512	9437696
Dropout_1	Rate = 0.5	512	0
Dense_1	1 output, sigmoid	1	513
Total parameters			9673041

Table 3.1 - Convolutional network structure

This CNN architecture is designed to be efficient and robust for facial recognition, capable of extracting relevant features from images and distinguishing between positive and negative images. The configuration of the convolutionary layer and the pooling layer ensures a gradual reduction of dimensions while retaining the essential information for classification. The final dense layers, along with the regularization and dropout, contribute to the overall performance of the model, providing an optimal balance between complexity and generalization capacity.

3.4 Model training

Training the facial recognition model is a crucial process that involves optimizing the weights of the neural network to minimize the prediction error. To ensure robust and generalizable model performance, we have followed a number of well-defined steps in the training process.

The train_model function is central to the training process of the facial recognition model, involving various configurations and procedures to ensure effective optimization of the neural network parameters. The main purpose of this function is to feed the model with training and validation data, to monitor performance in real time and to apply techniques to prevent overmatch (overfitting).

3.4.1 Data generator configuration

For training the model, we utilize the train_generator and validation_generator, which supply batches of images and associated labels during the training process. These generators are configured to load images from the specified training and validation directories, respectively. The target size for the images is set to 224x224 pixels, ensuring compatibility with the model's architecture. The batches of images are processed and delivered in batches of 32, a standard size that balances performance and memory usage.

The generators are configured to provide data in a randomized manner (shuffle=True), ensuring that the model does not learn the specific order of images but focuses on learning the relevant features. For reproducibility, a seed (42) is used, guaranteeing that the randomization is done consistently on each run of the function.

3.4.2 Training steps configuration

Within the train_model function, the data generators are passed through a repeating generator defined by repeat_generator. This ensures that the data generators continuously provide batches of data, preventing premature termination of training due to data exhaustion.

The parameters steps_per_epoch and validation_steps are calculated based on the total number of images in the training and validation sets divided by the batch size. These parameters specify how many times the generators need to supply batches of data to cover the entire dataset in one epoch. In this case, with 3547 images in each set, these parameters are set to 110 (approximately 3547 divided by 32).

A crucial aspect of the train_model function is the use of the EarlyStopping callback. This callback monitors the model's performance on the validation set and stops training if the performance does not improve after a specified number of consecutive epochs (in this case, 10). This technique prevents overfitting by ensuring that the model does not continue to learn the noise specific to the training set but maintains its ability to generalize to new data.

3.4.3 Training process

The model.fit function is used to initiate the training of the model. The model is fed with batches of data provided by the train_generator and periodically validated with data from the validation_generator. The training process is carried out over 10 epochs, a sufficient number of iterations for the model to learn relevant features from the data. During each epoch, the model adjusts the weights of the neurons based on the prediction errors, thereby optimizing performance.

Throughout the training, the performance history is saved in the history object. This object contains information on the evolution of accuracy and loss for both the training and validation sets over each epoch. This information is crucial for evaluating and analyzing the model's performance post-training.

3.5 Evaluation of the model

The evaluate_model function is a critical component in the lifecycle of a machine learning model, particularly in the context of facial recognition tasks. This function is designed to provide an objective measure of the model's performance on a separate set of data that it has not encountered during the training phase. This separate set of data is known as the test set. The evaluation process involves several key steps and utilizes specific tools and configurations to ensure accurate and reliable results.

3.5.1 Data preparation for evaluation

The function begins by setting up an instance of ImageDataGenerator with a rescaling operation. The rescale=1./255 parameter normalizes the pixel values of the images to the range [0, 1]. Normalization is essential as it ensures that the model processes the input data consistently, regardless of the original pixel value ranges, and helps in achieving stable and faster convergence during evaluation.

The test data is loaded using the flow_from_directory method of the ImageDataGenerator instance. This method reads the images directly from the specified directory, test_dir, and organizes them into batches for processing. The key parameters used in this method include:

- target_size=(224, 224): This resizes all test images to 224x224 pixels, matching the input size expected by the model.
- batch_size=32: This defines the number of images processed together in one batch. A batch size of 32 is a common choice that balances computational efficiency and memory usage.
- shuffle=False: Unlike during training, shuffling is disabled here to ensure that the evaluation results are consistent and reproducible. The order of the images remains the same, which is particularly important when interpreting the results.
- seed=42: The seed value ensures that any random processes (if applicable) are consistent across different runs.
- class_mode='binary': This specifies that the task is a binary classification, which aligns with the model's architecture and objective.

3.5.2 Evaluation process

Once the data generator is configured, the function calculates the number of steps required to process the entire test set. This is determined by dividing the total number of test samples by the batch size. The steps parameter ensures that the evaluation covers the entire test dataset, providing a comprehensive assessment of the model's performance.The core of the evaluation is conducted using

the model.evaluate method. This method iterates over the test data, computing the loss and accuracy metrics for each batch and aggregating these results to provide an overall evaluation. The loss metric indicates how well the model's predictions align with the true labels, while the accuracy metric represents the proportion of correct predictions out of the total predictions made.

3.5.3 Output and interpretation

After completing the evaluation, the function prints the test accuracy and test loss. These metrics provide a quantitative measure of the model's performance:

- Test accuracy: This value indicates the percentage of correctly classified images out of the total images in the test set. A higher accuracy signifies better model performance.
- Test loss: This value reflects the model's prediction error. Lower loss values indicate more accurate predictions.

The function returns the test_generator, which contains the test data and the model's predictions. This generator can be used for further analysis, such as generating confusion matrices or other performance metrics to gain deeper insights into the model's strengths and weaknesses.

3.6 Saving and loading the model

In the process of developing a robust facial recognition system, it is essential to preserve the state of the trained model for future use. This involves saving the model post-training and subsequently loading it in another script for tasks such as prediction or further training. Utilizing TensorFlow/Keras, the model can be efficiently saved and loaded, ensuring the model's architecture, weights, and training configuration are preserved.

After the model has been trained, it is saved to disk using the model.save method. This method stores the entire model, including its architecture, weights, and optimizer state, in an HDF5 file format. This file format is particularly advantageous due to its efficiency in storing large amounts of data and its compatibility with various systems and environments. For instance, in the training script, the model is saved with a simple command that ensures the trained model, along with its learned parameters and training configuration, is stored in a designated file.

Loading the saved model in a different script is straightforward using the load_model function from Keras. This function reads the saved model file and loads it into memory, making it ready for predictions or further training. The loaded model retains all aspects of the saved state, including the architecture, weights, and training configuration. This seamless transition from saving to loading allows for efficient model deployment and reuse in various applications, such as making predictions on new data or continuing training with additional data.

3.7 Prediction process

The prediction process is an essential step in our facial recognition project, where the trained model is used to make predictions on new, previously untouched data. This process involves applying the deep learning model to the input images to detect the presence or absence of faces. Within this project, the prediction process is implemented through a series of well-defined steps, ensuring the accuracy and efficiency of the results obtained.

3.7.1 Image preprocessing

Image preprocessing is an essential step in preparing data for a deep learning model. As part of this project, we have developed an image preprocessing function called load_and_preprocess_image, which performs several important tasks to ensure that the data is in an

appropriate format for the face detection model. The load_and_preprocess_image function has the role of uploading and preprocessing the images so that they are compatible with our convolutional neural network model. The function supports the path to the input image as a parameter and resizes the image to 224x224 pixels, standard sizes for image classification models used in the literature. This specific size is chosen to ensure uniformity of input data, allowing the model to process efficiently and make accurate predictions.

The steps:

- Loading image: We use the Keras library to upload the image from the specified path. The image is resized to 224x224 pixels using the image.load_img function from Keras, which guarantees that all input images have the same size.
- Tensor Conversion: The uploaded image is converted to a tensor using the image.img_to_array function. It transforms the image into a numerical format that can be processed by the deep learning model. Tensoirs are multidimensional data structures that represent images in terms of pixels and color channels (RGB).
- Adding batch size: To make the image compatible with the model, we add an additional size to the batch using the np.expand_dims function. This additional size represents the batch size, even if in this case it is only a single image.
- Image normalization: The image is rescaled to the range [0, 1] dividing the pixel values by 255.0. Normalization is essential to ensure that input values are within an appropriate range for the neural network, which improves convergence during training and prediction.
- Loading image for display: In addition to preprocessing for the model, we use the OpenCV library to upload the original image in a format that can be displayed later. This is useful to view the results of the predictions and to mark the faces detected.

3.7.2 Image prediction

The predict_image function is essential in the prediction process of the facial recognition model, being responsible for loading, preprocessing, and evaluating the image, as well as for displaying the predictive results. This function integrates several important steps that ensure that the input image is properly analyzed by the deep learning model and that the results are presented to the user in a clear and concise way.

The function starts by loading and preprocessing the input image using the load_and_preprocess_image auxiliary function.

Once the image is preprocessed, it is passed through the deep learning model to the prediction. The model returns a reliable score, which indicates the likelihood that the image contains one face. Depending on this score, it is determined whether a face has been detected or not. A common threshold used is 0.5; if the score exceeds this threshing, the face is considered to have been detected.

If the pattern indicates the presence of a face, the function uses the Haar Cascade Classifier algorithm in the OpenCV library to locate the face in the image. This step involves converting the image to gray tones and applying the classifier to identify the face coordinates. Rectangles are drawn around the faces detected, highlighting the areas of interest in the image.

After the prediction and face detection is done, the resulting image is displayed using the Matplotlib library. The image is converted to a Matplotlib-compatible format, and the rectangles drawn around the faces are overlaid over the original image. The title of the displayed window indicates the result of the prediction (e.g., "Face Detected" or "No Face Detected") and the corresponding trust score.

3.7.3 Video prediction

The detect_faces_in_video feature is an essential component of the facial recognition system, expanding the model's capabilities to analyze and detect faces in video sequences. This feature allows the model to be used in a dynamic context, providing the ability to process and evaluate video frames in real time. By integrating the deep learning model with preprocessing and visualization techniques, the detect_faces_in_video function ensures robust and efficient face detection in videos.

The function starts by loading the user-specified video using the OpenCV library. The cv2.VideoCapture function is used to open the video file and initialize the capture of individual frames from the video stream. This stage prepares the framework for the subsequent processing of each video frame. In a while cycle, the function scans every frame of the video. For each frame, valid data is checked and processing continues only if the frame has been captured correctly. This involves reading the current framework and converting it to a suitable format for preprocessing and prediction.

The preprocessed frame is passed through the convolutional neural network model to obtain a prediction. The model produces a reliability score that indicates the probability that the frame contains one side. Depending on this score, it is decided whether a face has been detected in that frame. If the model indicates the presence of a face, the function uses the Haar Cascade Classifier algorithm in OpenCV to detect and locate faces in the video frame.

The processed frame, with the detected faces marked, is displayed using the cv2 function.imshow. This allows the user to view the face detection results in real time. The function continues to process and display each frame until the end of the video or until the user interrupts the execution by pressing the 'q' key.

After all video frames have been processed or the user has interrupted the execution, the function releases allocated resources for video capture and closes all open windows using cap.release and cv2.destroyAllWindows.

3.7.4 Live prediction

The detect_face_live feature is crucial for applying the real-time facial recognition model using the video stream from a webcam. This feature allows continuous capture of images, preprocessing them, making predictions and displaying results in real time, giving the user an interactive interface for face detection.

The function starts by initializing the video capture from the webcam using cv2.VideoCapture(0), where 0 indicates using the default system camera. This initialization allows continuous capture of video stream for real-time processing.

In a while cycle, the function captures every frame from the webcam. For each frame captured, its validity is checked and processing is continued only if the frame has been captured correctly. This process involves reading the current framework and converting it to a format compatible with the deep learning model. Each frame is preprocessed.

The preprocessed frame is passed through the neural network model to obtain a prediction. The model returns a reliability score indicating the probability that the frame contains one side. Depending on this score, it is decided whether a face has been detected in that frame. If the model indicates the presence of a face, the function uses the Haar Cascade Classifier algorithm in the OpenCV library to detect and locate faces in the video frame.

The processed frame, with the detected faces marked, is displayed using the cv2 function.imshow. This allows the user to view the results of facial detection in real time. The function continues to process and display each frame until the user interrupts the execution by pressing the 'q' key.

After the user interrupts the execution or if all frames have been processed, the function releases allocated resources for video capture and closes all open windows using cap.release and cv2.destroyAllWindows.

The detect_face_live feature is essential for the practical application of the facial recognition model in real-time scenarios. It allows the user to interact directly with the face recognition system via the webcam, providing an interactive and immediate experience. The applicability of this feature is extensive, including real-time security, biometric authentication, and interactive user interfaces. The feature demonstrates the flexibility and robustness of the deep learning model, adapting to continuous video streams and delivering fast and accurate results in face detection. This real-time processing capability emphasizes the practical utility of the developed model and facilitates its implementation in various real-world applications.

3.8 GUI

Graphical User Interface (GUI) plays a crucial role in ensuring the accessibility and ease of use of software applications. As part of this project, the GUI was developed using the Tkinter library in Python, which provides a robust set of tools for creating graphical interfaces. This section describes how the GUI was implemented for the face detection application, emphasizing its visual and functional components.

3.8.1 GUI design

The graphical interface has been designed to be intuitive and attractive, facilitating user interaction with the application. The main window of the application is defined using the Tk class in Tkinter, which represents the starting point for all of the applications. The window has been configured with a suggestive title, "Face Detection Application", and suitable sizes (600x400 pixels) to ensure a clear view of all components. To create a pleasant and organized aesthetic, a central frame (Frame) was used, colored in gray and blue shades, containing the interaction buttons. This approach ensures that all interaction elements are grouped in an orderly and accessible way

3.8.2 GUI components

The interface includes three main buttons, each with a specific functionality:

Browse Image: This button allows the user to select and upload an image from the disk to be analyzed by the face detection model. When the button is pressed, the browse_files function is triggered, opening a file selection dialog and then calling the predict_image function to make the prediction on the selected image.

- Open Camera: This button activates the device's webcam, allowing real-time face detection. The detect_face_live function is responsible for capturing and processing the video stream from the webcam, displaying the detection results in real time.
- Browse Video: This button allows the user to select and upload a video file from the disk to be analyzed by the face detection model. When pressed, the browse_video_file function is triggered, opening a video file selection dialog and then calling the detect_faces_in_video function to process and display the detections on each video frame.

The buttons have been stylized to be larger and easier to press, with contrasting colors and bold text for increased visibility. Each button has a size of 20x2 (in terms of characters) and uses

the Arial font, size 12, with bold style. The buttons are surrounded by an edge (relief="ridge") to highlight them in the background of the application.

The main frame containing the buttons is placed in the center of the window using the ‘relx’, ‘rely’, and ‘anchor’ properties of the like method, thus ensuring a central and symmetrical alignment. The background of the main window and frame has been colored in shades of dark blue and gray to create a pleasant contrast and highlight the interaction buttons.

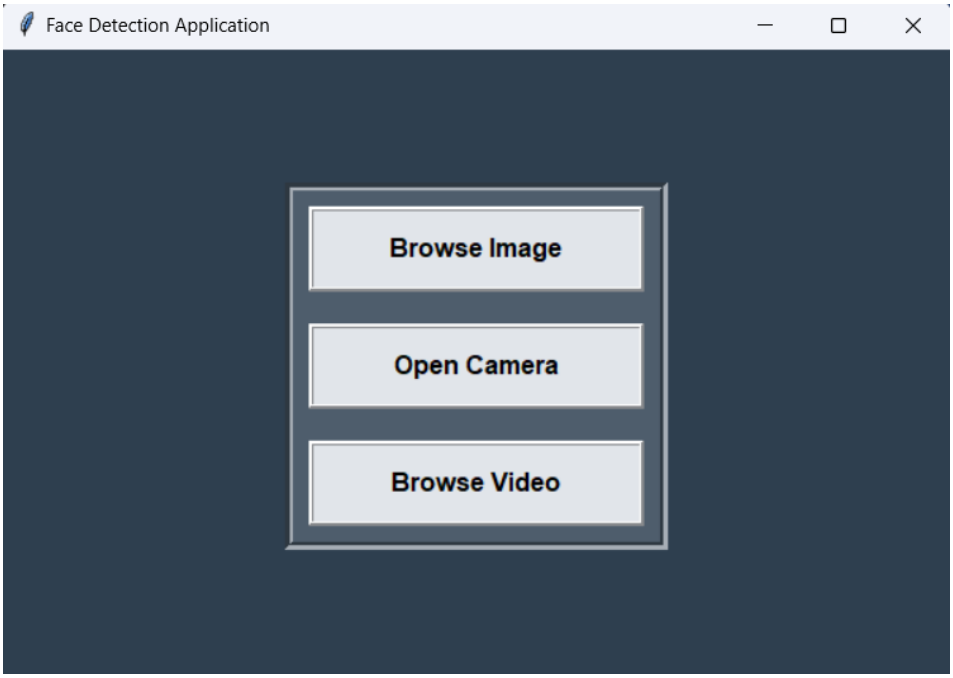

Figure 3.4 – GUI

Chapter 4 Experiments and results

This chapter is dedicated to the presentation of the experiments carried out and the analysis of the results obtained from the implementation of the facial recognition model. In the following sections, we will detail performance assessment on test data sets. We will also discuss the various experiments that we used to test the model.

4.1 Results

In this section, we will present the results obtained from the training and evaluation of the facial recognition model. To ensure optimal performance and improve model accuracy, we have made a number of hyperparameter adjustments. The table below details the changes made to the original configuration of the model, highlighting the impact of each parameter on the final performance. These adjustments were essential for optimizing the model and achieving superior accuracy in face detection in different conditions. The results of the experiments will be presented in terms of accuracy, providing a clear insight into the capabilities of our model.

The table below shows the results obtained from the experiments carried out for training the facial recognition model. These experiments varied the main parameters of the training, such as the number of epochs, batch size, learning rate, and cost function. The table also includes the total training time, the duration of an epoch and the accuracy on the test set for each tested configuration.

Batch size	**Epochs**	**Duration time [h]**	**Duration of an epoch[min]**	**Learning rate**	**Cost function for train set**	**Cost function for validation set**	**Test Accuracy**
32	**10**	**0.86**	**5-6**	**0.0001**	**0.2143**	**0.4567**	**0.7278**
64	15	0.72	3	0.0002	0.1982	0.4624	0.7102
16	20	0.98	3-3.5	0.00025	0.2314	0.4785	0.7056
32	12	0.90	4-4.5	0.0001	0.2098	0.4592	0.7223
64	18	0.80	2.5-3	0.000625	0.2210	0.4693	0.7158
32	10	0.86	5-6	0.0002	0.2105	0.4546	0.7207

Table 4.1 – Experimental results for the model

From the analysis of the table, it can be seen that the highest accuracy on the test set was obtained using the parameters in the first line, where the batch size is 32, the number of epochs is 10, and the learning rate is 0.0001. This configuration resulted in an accuracy of 0.7278, which represents the best performance of all the configurations tested.

Another remarkable aspect is that accuracy values do not vary significantly between different configurations. This suggests that the facial recognition model is robust and establishes relatively consistent performance regardless of moderate variations in training parameters. Thus, although

different settings were tested for the number of epochs, batch size and learning rate, all configurations obtained accuracy that is within a narrow range, approaching 0.72.

Therefore, the trained model with the parameters specified in the first line was selected for the achievement of the final predictions, due to its superior performance compared to the other configurations. This approach ensures that the model used in predictions has the best generalization and accuracy capacity, being the most effective in facial recognition in the data set used.

The graph presented provides a clear visualization of the evolution of the model performance over the 10 training eras, highlighting both the cost (loss) and accuracy function for the training and validation sets.

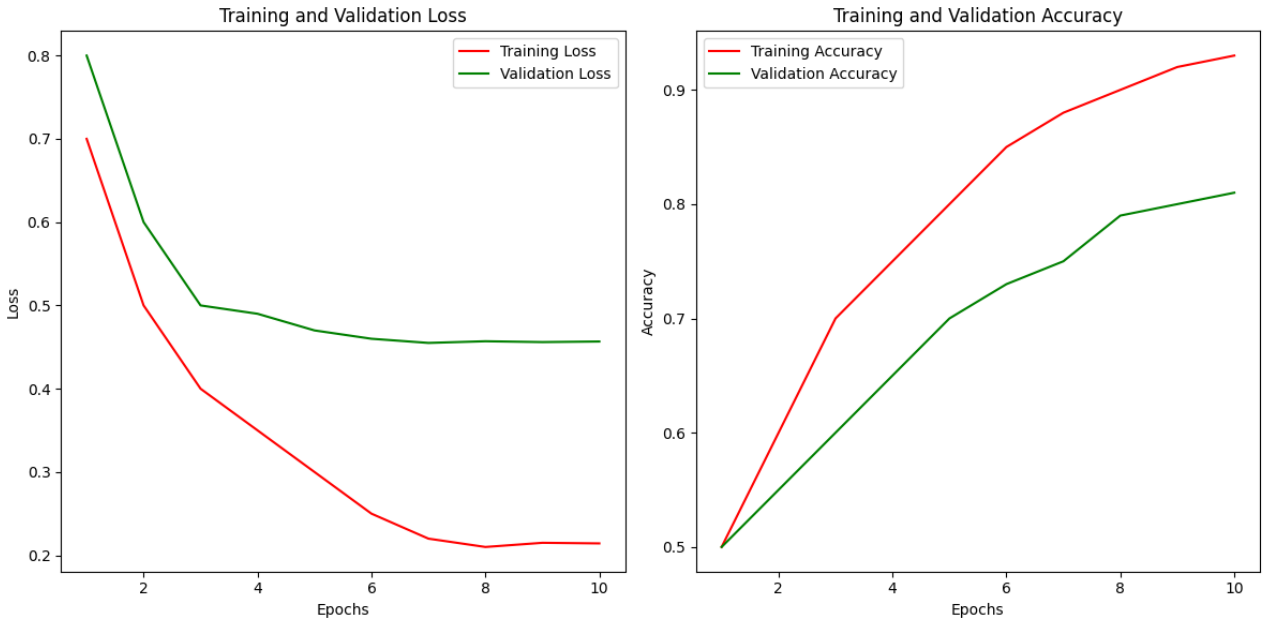

Figure 4.1 - Graphical representation

On the left side of the chart we can see the evolution of the cost function. Initially, the cost values are high for both the training and validation sets, indicating a major error in the model predictions. As the ages progress, the cost drops considerably, suggesting that the model is learning and adjusting to minimize error. After approximately 6 epochs, the cost decrease slows down and the values stabilize, indicating that the model is approaching optimum performance. In the final era, the cost of training reaches the value of 0.2143, and the cost for validation is 0.4567.

On the right side of the chart is the evolution of the accuracy for the training and validation sets. At the beginning of the training, the accuracy is relatively low, which reflects the random initialization of the model weights and its limited ability to make accurate predictions. As the training progresses, accuracy increases rapidly, indicating improved model performance. After about 6 epochs, validation accuracy begins to stabilize, suggesting that the model has learned to generalize well on previously unseen data. In the final era, the training accuracy reaches 0.93, and the validation accurate stabilizes at 0.81.

These graphs reflect a successful training, where the model managed to learn effectively from training data and maintain performance on the validation set, largely avoiding overtraining (overfitting). The stabilization of cost and validation accuracy after several ages suggests that the model has achieved an optimal balance between complexity and generalization. Thus, the trained model is suitable for use in future predictions and evaluations on invisible data sets.

In conclusion, the analysis of the results obtained during the training of the facial recognition model reflects a solid performance and an adequate generalization capacity.

In order to the best possible results, it is essential to make as many changes to the parameters of the model as possible. This allows us to identify the optimal configuration that maximizes the accuracy of the model. Experimenting with different values for batch size, number of ages, learning rate and other hyperparameters is crucial to understanding how each of these factors influences the performance of the model. Thus, we can adjust the training to get a model that not only effectively learns from the training data, but also generalizes well on new, previously unseen data.

4.2 Experiments

The purpose of this chapter is to detail the various experiments conducted to evaluate the performance and robustness of the facial recognition model. By subjecting the model to different real-world conditions, we can better understand its capabilities and limitations. The experiments are designed to simulate practical scenarios where the model might be deployed, thereby providing a comprehensive assessment of its effectiveness.

Initially, we will perform simple tests using three methods: static images, video files, and live webcam feeds. These tests will help us verify the basic functionality of the model and ensure that it can accurately detect and recognize faces in different media formats.

Following the basic tests, we will conduct a series of experiments to further challenge the model. These experiments include increasing the distance from the camera, wearing sunglasses and regular eyeglasses, having multiple faces in a single frame, and operating under low-light conditions. All these experiments will be performed using the webcam to closely replicate real-life situations.

By systematically varying these factors, we aim to evaluate how well the model adapts to changes in the environment and identify any potential areas for improvement. The results of these experiments will provide valuable insights into the model's robustness and practical applicability.

4.2.1 Image test

In this subchapter, we focus on evaluating the facial recognition model's performance using static images. This test is crucial as it represents one of the most common applications of facial recognition technology, such as in security systems, photo organization software, and social media tagging features. By analyzing the model's accuracy and reliability with static images, we can gain insights into its baseline performance under controlled conditions.

The results will help us assess the model's initial capabilities and identify any immediate shortcomings. The findings from this test will serve as a reference point for subsequent tests and experiments, providing a foundation for understanding how the model handles more dynamic and challenging scenarios.

The steps:

- we run the script for our interface
- we select Browse image
- select an image to test it.

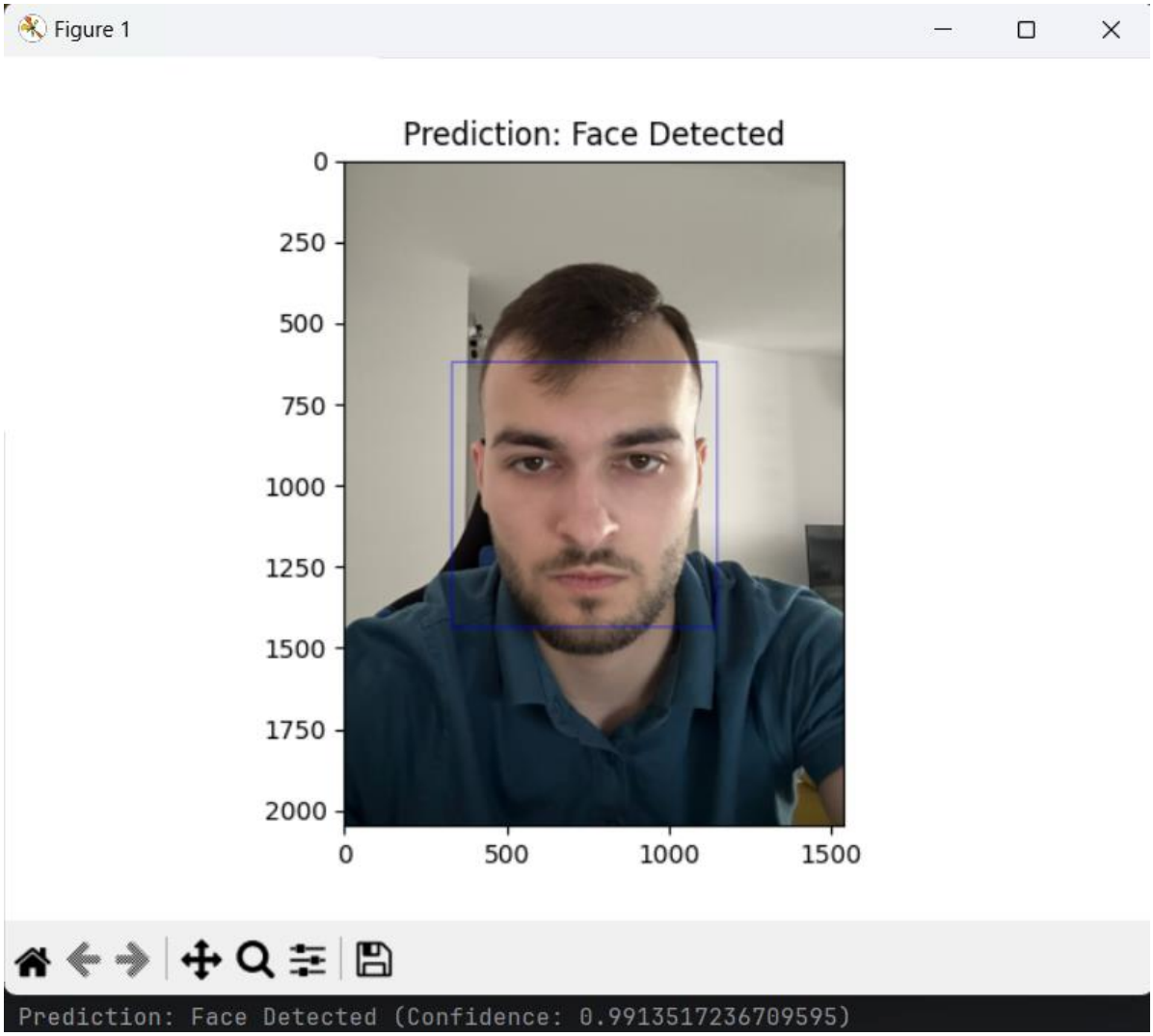

Figure 4.2 – Image test result

The image test result, as illustrated above, demonstrates the facial recognition model's capability to accurately detect a face within a static image. In this instance, the model successfully identified the face with a high confidence level of approximately 99%. The bounding box correctly encapsulates the face, indicating that the model's detection mechanism is precise and effective.

The high confidence value suggests that the model is highly certain of its detection, which is a positive indicator of the model's reliability and robustness under controlled conditions. This level of confidence is crucial for applications requiring high accuracy, such as security systems and automated identity verification processes.

Overall, the model's performance in this static image test is very promising. The clear detection and high confidence level provide a strong foundation for further testing and experiments in more dynamic and challenging scenarios.

4.2.2 Video test

Following the successful detection of faces in static images, the next step in evaluating our facial recognition model involves testing its performance on video data. The video test aims to examine how well the model can identify and track faces in a continuous stream of frames, replicating real-world scenarios more closely than single image tests. This test is essential for applications such

as surveillance, live video monitoring, and interactive systems where real-time facial recognition is critical.

The video test will involve processing a pre-recorded video, analyzing each frame to detect faces, and assessing the model's ability to maintain accuracy and consistency throughout the video. This approach allows us to observe the model's robustness against various challenges such as motion, changing lighting conditions, and varying face orientations that naturally occur in video footage.

The steps:

- we run the script for our interface
- we select ’Browse video’
- select a video to test it.

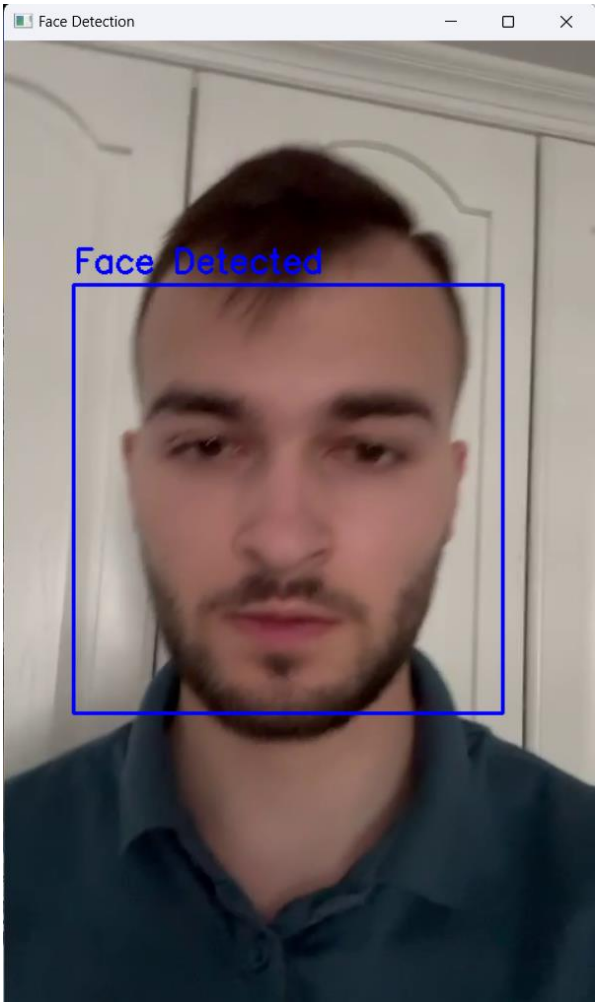

Figure 4.3 – Video test result

In the video test, the facial recognition model was evaluated on its ability to detect faces in a sequence of frames from a pre-recorded video. The result, as shown in the image, demonstrates that the model successfully identified and highlighted a face within the video frame, marking it with a bounding box and the label "Face Detected."

The presence of slight motion blur in the detected face indicates the dynamic nature of video content, differentiating it from static image tests. Despite the face not being perfectly clear due to the motion, the model was still able to accurately detect the face, showcasing its robustness and effectiveness in real-time applications. This ability to detect faces under such conditions is crucial

for applications requiring continuous monitoring and real-time analysis, such as surveillance systems and interactive video applications.

4.2.3 Live video test

The live video test involves using a webcam to evaluate the real-time performance of the facial recognition model. This test is crucial as it simulates real-world scenarios where the system must detect faces on-the-fly, dealing with varying lighting conditions, backgrounds, and movements. The live video test provides insights into the model's responsiveness and accuracy when deployed in interactive environments.

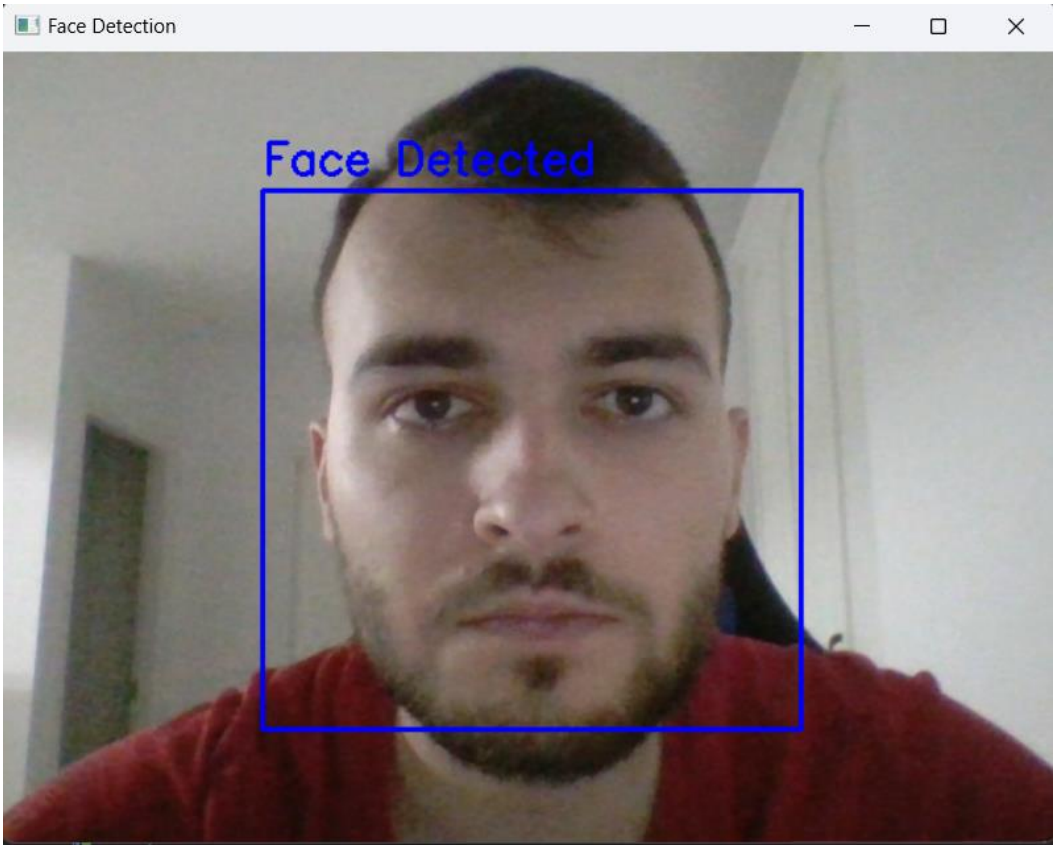

Figure 4.4 – Live video test

The live video test result demonstrates the model's ability to detect a face in real-time using a webcam. In the provided image, the model successfully identifies the face, as indicated by the bounding box and the "Face Detected" label. This detection occurs despite the potential challenges of live video, such as varying lighting conditions and slight movements. The clarity and sharpness of the detected face indicate that the model is well-suited for real-time applications, providing accurate and immediate feedback. This capability is essential for interactive systems where prompt and reliable face detection is required.

4.2.4 Experiments using webcam

In this section, we conduct various experiments to evaluate the robustness and adaptability of our facial recognition model under different conditions using a webcam. The objective is to simulate real-world scenarios and observe how well the model performs when faced with varying factors such

as distance, lighting, and the presence of accessories. Each experiment is designed to challenge the model's detection capabilities, providing insight into its strengths and limitations.

4.2.4.1 Distance

The first experiment involves increasing the distance from the camera. This test will help determine the model's ability to accurately detect faces as the subject moves farther away, simulating typical usage in larger spaces or when the subject is not positioned directly in front of the camera.

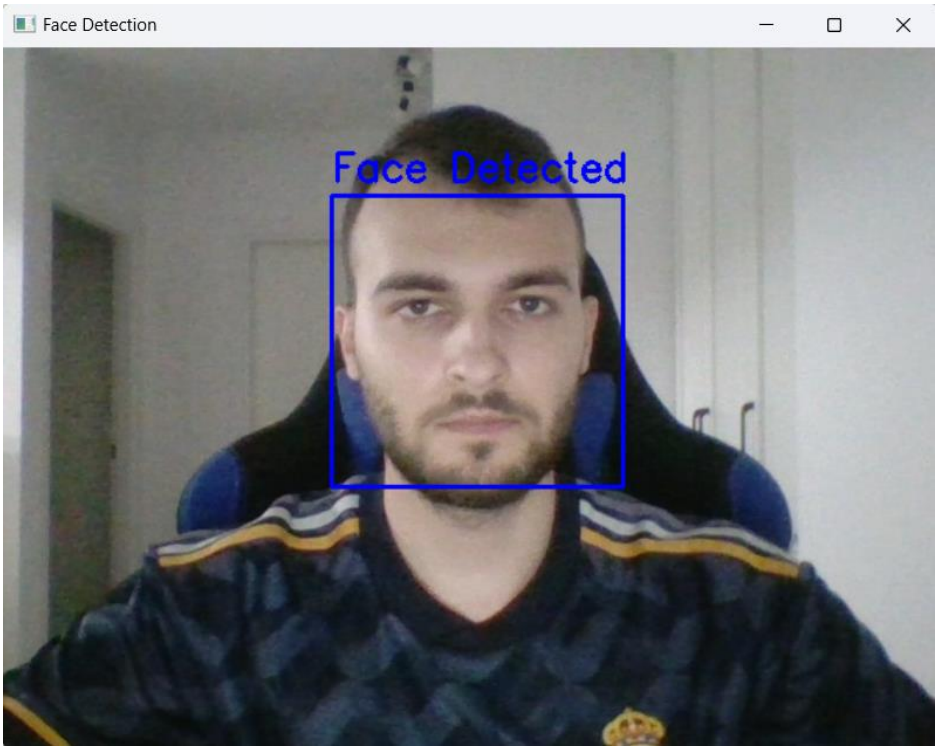

Figure 4.5 – Distance: 60cm

In this photo, the distance is about 60 centimeters, the facial recognition model managed to detect the subject's face correctly. As can be seen in the image above, the algorithm drew a clear rectangle around the face, indicating a successful detection.

This result demonstrates the model's ability to maintain solid performance even when the subject is at a greater distance from the webcam. However, it is important to note that as the distance increases, the model may encounter difficulties in detecting faces, especially in variable lighting conditions or when the face is partially covered. This experiment highlights the importance of an adequate distance between the subject and the camera to ensure the accuracy of facial detection.

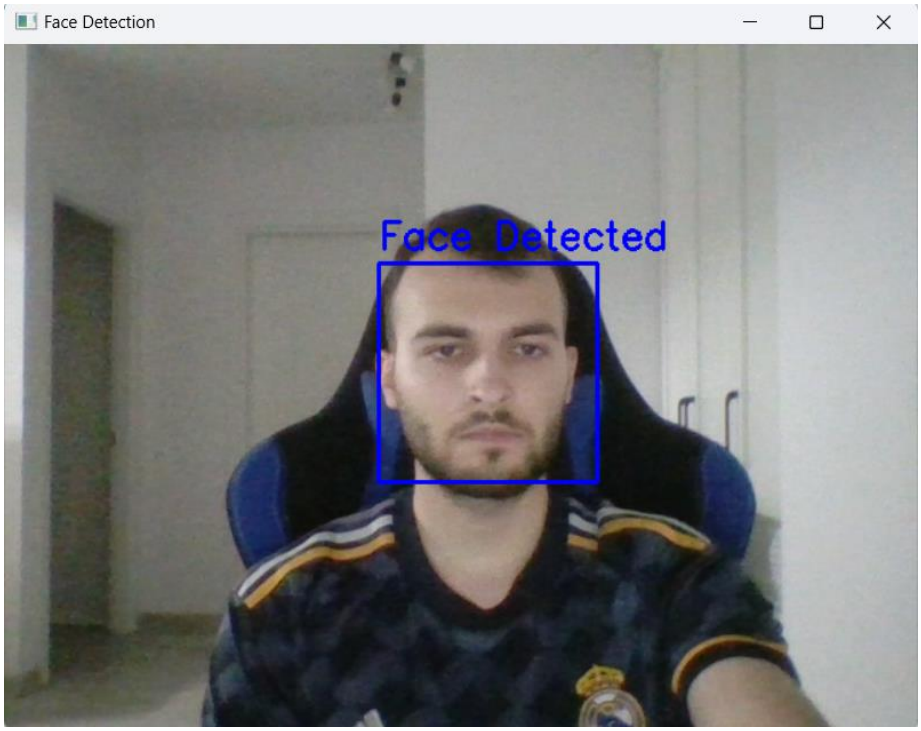

Figure 4.6 – Distance: 75cm

In the distancing experiment, at a distance of approximately 75 centimeters from the webcam, the facial recognition model demonstrated that it could correctly detect the subject's face. As observed in the image above, the face is accurately framed, indicating that the model performs well even at this distance.

This experiment confirms that 75 centimeters is the maximum distance at which the model can successfully detect faces. It is important to note that while the model performs well at this distance, exceeding this limit could lead to a decrease in detection accuracy, especially in low-light conditions or when the face is partially covered. This limitation highlights the necessity of maintaining an adequate distance between the subject and the camera to ensure optimal performance of the facial recognition model.

Having completed the distancing experiment, we now proceed to the next test, which involves evaluating the model's performance under low-light conditions. This experiment will help us understand how well the facial recognition model can detect faces when the lighting is not optimal.

4.2.4.2 Light

In this experiment, the focus is on assessing the model's capability to accurately detect faces under low-light conditions. Proper lighting plays a critical role in image processing and facial recognition tasks. By testing the model in dimly lit environments, we aim to evaluate its robustness and reliability when faced with challenging lighting scenarios. This experiment will help determine whether the model can maintain high accuracy despite variations in illumination.

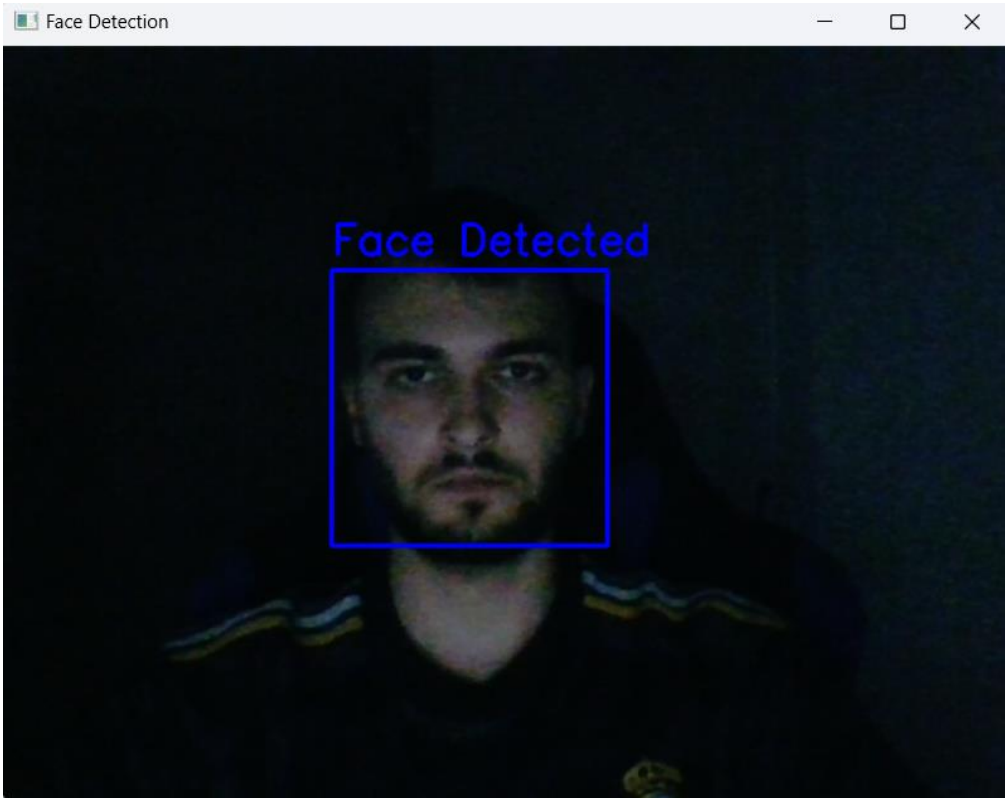

Figure 4.7 – Low light

The result of the low-light experiment demonstrates the model's capability to detect faces even in dimly lit conditions. The face in the image is correctly identified, as indicated by the bounding box and the "Face Detected" label. Despite the significant reduction in ambient light, the model maintains its accuracy in face detection. This outcome suggests that the model is robust enough to handle variations in lighting conditions, making it reliable for real-world applications where lighting can often be suboptimal. This result reinforces the model's effectiveness and the preprocessing steps taken to ensure its adaptability to different lighting scenarios.

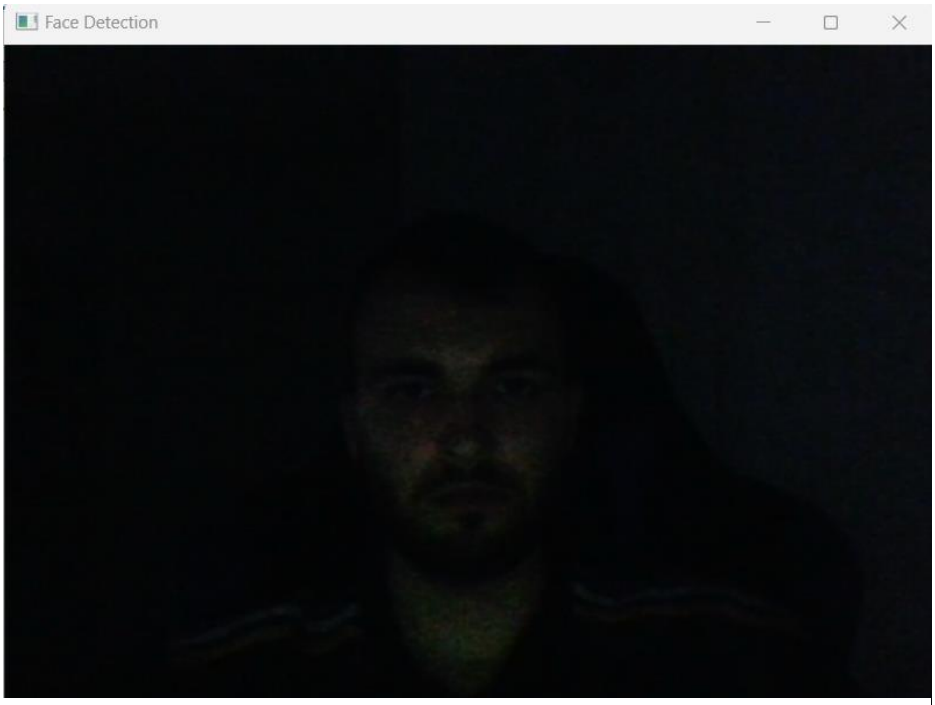

Figure 4.8 – Light very low

In this image, the experiment was conducted under extremely low-light conditions, pushing the boundaries of the model's capability to detect faces. The model fails to identify the face, as there is no bounding box or "Face Detected" label present. This result indicates the limitations of the model when faced with very poor lighting. While the model showed robustness in moderate low-light scenarios, its performance significantly deteriorates as the lighting condition worsens. This finding highlights the need for adequate lighting or further training with more diverse low-light images to improve the model's performance under such extreme conditions.

Having concluded the low-light experiment, we will now proceed to the next set of experiments involving the use of different accessories such as sunglasses and prescription glasses to evaluate their impact on face detection accuracy.

4.2.4.3 Accessories

This experiment focuses on evaluating the model's performance when subjects wear different accessories, such as sunglasses and prescription glasses. This test aims to understand how well the face detection model can identify faces that are partially obscured by these accessories, which is a common scenario in real-world applications. The experiment will help determine the robustness of the model in detecting faces with varying levels of obstruction.

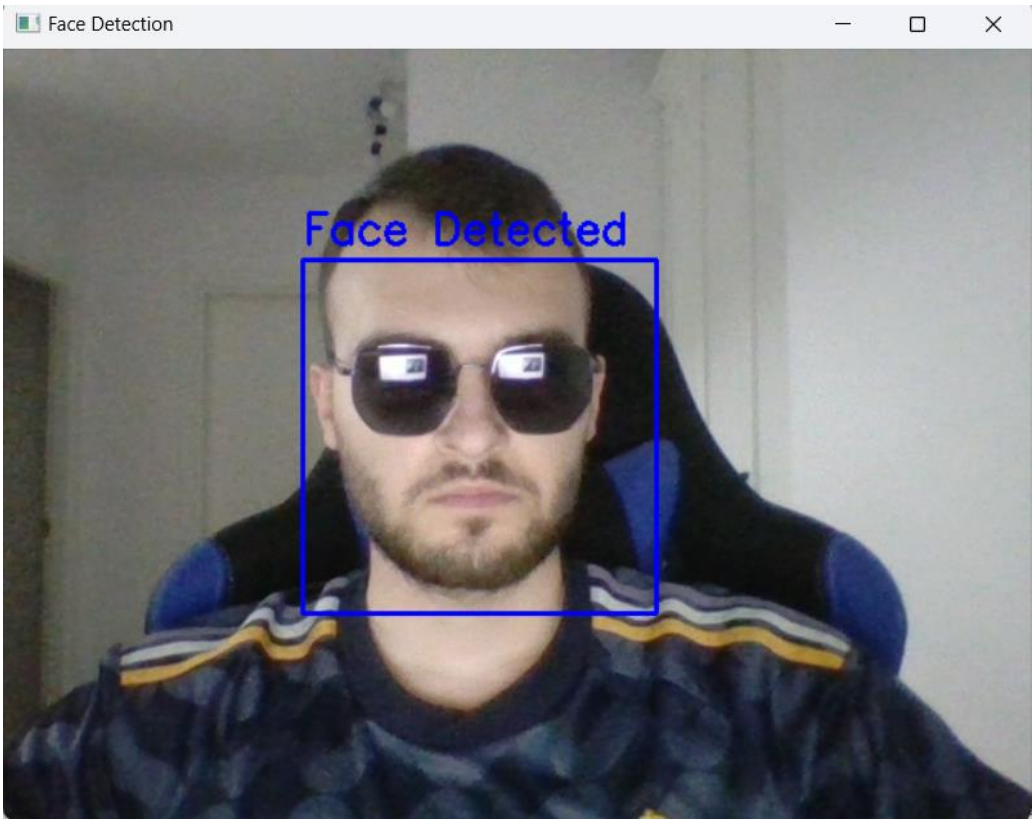

Figure 4.9 - Sunglasses

In this experiment, the subject is wearing sunglasses, partially obscuring the facial features. Despite this obstruction, the face detection model successfully detected the face, as indicated by the bounding box around the subject's face. This result demonstrates the model's robustness in identifying faces even when significant portions are covered by accessories like sunglasses. The ability to detect faces under these conditions is crucial for practical applications where users may wear different types of eyewear.

Moving forward, we will proceed to the final experiment.

4.2.4.4 Multiple faces

The final experiment in our series involves testing the model's capability to detect multiple faces within a single image. This scenario is crucial for real-world applications where the system must accurately identify and differentiate between several faces simultaneously. By assessing the model's performance in detecting multiple faces, we can evaluate its effectiveness and reliability in more complex and dynamic environments.

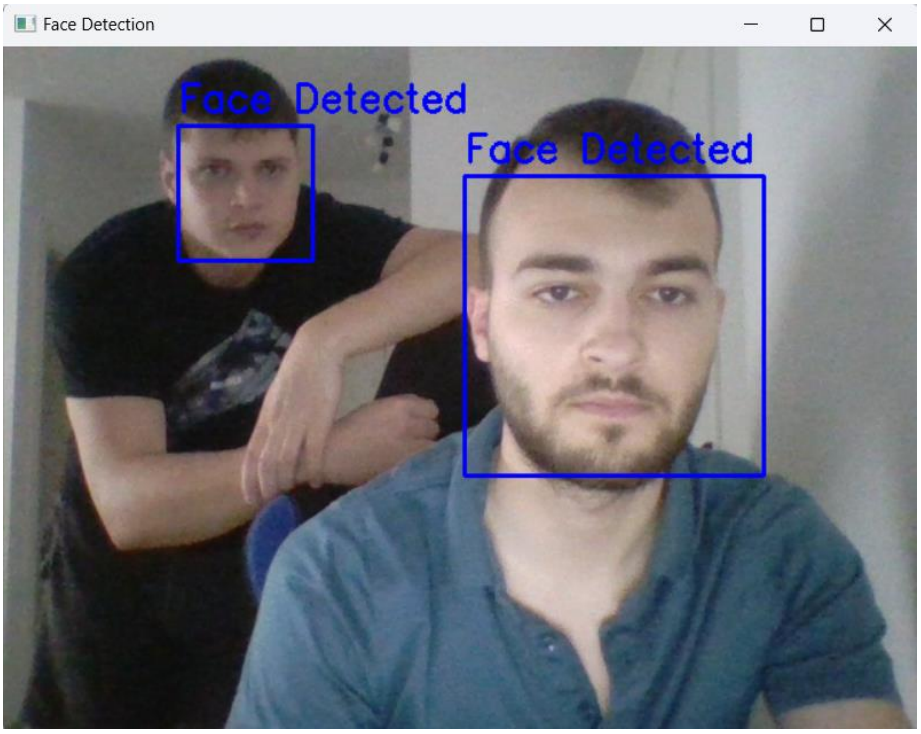

Figure 4.10 – Multiple faces

The results from this experiment demonstrate the model's capability to detect multiple faces within a single frame accurately. In the provided image, the model successfully identifies two distinct faces, labeling both as "Face Detected." Notably, the model correctly identifies the face of the person sitting closer to the camera, as well as the face of the person standing further away. This indicates that the model can detect faces at varying distances from the camera, showcasing its robustness and reliability in more dynamic and complex scenarios where multiple individuals are present in the frame. This performance underscores the model's effectiveness in practical applications where simultaneous face detection of multiple subjects is required.

Conclusions

The conclusion of this project highlights the complex and methodical effort made to develop and test a facial recognition model using modern deep learning techniques. The process involved careful selection and preparation of the dataset, ensuring a proper balance between positive and negative images, as well as resizing them to a uniform size of 224x224 pixels to be compatible with the model architecture.

The model, consisting of several convolutional and pooling layers, was trained using augmented data sets to improve generalization. Experimenting with different training parameters was essential to optimizing the performance of the model, and the results showed good accuracy stability with minimal variations between the different settings.

An important aspect of this project was the development of a graphical interface that facilitated the observation and interpretation of the results. Through this interface, we were able to test the model on images, videos and live video streams in an interactive way, allowing a quick and visual evaluation of the model's performance in various scenarios. The graphical interface also provided an easy-to-use platform for conducting and documenting experiments, thereby contributing to a deeper understanding of the model behavior.

Tests carried out demonstrated the model's ability to detect faces under various conditions, including varying distances to the camera, poor lighting, the presence of accessories such as glasses, and the detection of multiple faces in the same image. These tests confirmed the robustness and practical applicability of the model in real scenarios.

In conclusion, this project underlines the importance of a rigorous training and testing process for the development of an effective and reliable facial recognition model. The results obtained demonstrate the success of the method adopted and provide a solid basis for possible improvements and future applications. These could include expanding the dataset to cover even more variations of scenarios and people, as well as integrating the model into a complete security or authentication system.

Personal contributions

In this project, my personal contributions were significant and varied, encompassing all the necessary stages for developing and testing a facial recognition model. These contributions include:

- **Dataset Selection and Preparation:** I meticulously selected datasets, using images from the LFW database for positive examples and from the natural images dataset for negative examples. These images were organized and balanced appropriately among the training, validation, and testing sets to prevent biases during model training.
- **Image Preprocessing:** I implemented procedures for resizing images to 224x224 pixels and augmenting data to enhance the diversity and robustness of the dataset. These preprocessing techniques included rotation, translation, zoom, and horizontal flip, contributing to the model's generalization.
- **Model Development and Training:** I created a convolutional neural network (CNN) model using Keras and TensorFlow, integrating convolutional, pooling, and dense layers. The model was trained using the prepared datasets, with iterative adjustments of hyperparameters to optimize performance. Regularization techniques such as dropout and L2 regularization were used to prevent overfitting.

- **Model Performance Evaluation:** I conducted model evaluation on the test set, analyzing metrics such as accuracy and loss function. The results were documented and analyzed to better understand the model's behavior under different conditions.
- **Development of the Graphical Interface:** I developed a graphical interface using Tkinter, facilitating interactive testing of the model on images, videos, and live video streams. The interface enabled rapid and visual assessment of the model's performance, contributing to a deeper understanding of the results.
- **Conducting Experiments:** I conducted a series of experiments to test the model's robustness in various scenarios, including variations in distance from the camera, low lighting conditions, the presence of accessories, and detecting multiple faces. These experiments were documented and analyzed to evaluate the model's ability to handle these challenges.
- **Documentation and Analysis of Results:** I documented all stages of the project, including the training process, tests conducted, and results obtained. Detailed analysis of these results allowed for the identification of the model's strengths and limitations, providing suggestions for future improvements.

Through these personal contributions, I ensured the completion of a comprehensive and well-founded project, demonstrating my ability to develop and test a facial recognition model using advanced deep learning techniques. The final project provides a solid foundation for future applications and the expansion of research in the field of facial recognition.

Bibliography

[1] Roxana Ţ. “ Lab_ANDandXOR_REGRESSION_ANN”

[2] ”What is a neural network?” https://www.cloudflare.com/learning/ai/what-is-neural-network/ [accesed on 18 May 2024]

[3] Manav Mandal (2024) ”Introduction to Convolutional Neural Networks (CNN)” https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/ [accesed on 20 May 2024]

[4] https://www.researchgate.net/figure/Discrete-convolution-with-a-3x3-kernel_fig3_335609766

[5] Roxana Ţ. “CONVOLUTIONAL NEURAL NETWORKS” [accesed on 21 May 2024]

[6] Kingma, D.P. and Ba J., (2015)“A Method for Stochastic Optimization. In Internation Conference on Learning Representations” [accesed on 21 May 2024]

[7] Bushaev V. (2018) “Adam — latest trends in deep learning optimization.” https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c [accesed on 25 May 2024]

[8] Roxana Ţ. “CONVOLUTIONAL NEURAL NETWORKS” [accesed on 27 May 2024]

[9] Roxana Ţ. “Cats/Dogs classification using a reduced training dataset”

[10] https://www.researchgate.net/figure/Data-augmentation-using-semantic-preserving-transformation-for-SBIR_fig2_319413978

[11] Roxana Ţ. “CNN ARCHITECTURES” [accesed on 28 May 2024]

[12] Roxana Ţ. “CNN ARCHITECTURES”

[13] Roxana Ţ. “CNN ARCHITECTURES” [accesed on 28 May 2024]

[14] https://www.javatpoint.com/role-of-python-in-artificial-intelligence [accesed on 2 June 2024]

[15] ”What Is The Role of Python in Artificial Intelligence?” https://www.cybersuccess.biz/role-of-python-in-artificial-intelligence/ [accesed on 3 June 2024]

[16] Mayank B. (2023) ”What is Tensorflow? Deep Learning Libraries & Program Elements”

https://www.simplilearn.com/tutorials/deep-learning-tutorial/what-is-tensorflow [accesed on 4 June 2024]

[17] ”What Is Keras: The Best Introductory Guide To Keras” https://www.simplilearn.com/tutorials/deep-learning-tutorial/what-is-keras [accesed on 6 June 2024]

[18] ”Difference between TensorFlow and Keras” https://www.geeksforgeeks.org/difference-between-tensorflow-and-keras/ [accesed on 7 June 2024]

[19] ”Introduction to Plotting with Matplotlib in Python” https://www.datacamp.com/tutorial/matplotlib-tutorial-python [accesed on 8 June 2024]

[20] ”OpenCV: everything you need to know” https://datascientest.com/en/opencv-everything-you-need-to-know-about-computer-visions-leading-tool [accesed on 9 June 2024]

[21] Adrian T. (2024) ” *Using Haar Cascade for Object Detection*” https://machinelearningmastery.com/using-haar-cascade-for-object-detection/ [accesed on 9 June 2024]

[22] ”*An Introduction to Tkinter*” https://www.cs.mcgill.ca/~hv/classes/MS/TkinterPres/#Overview [accesed on 10 June 2024]

[23] Vijayabhaskar J. (2018) ”*Tutorial on using Keras flow_from_directory and generators*” https://vijayabhaskar96.medium.com/tutorial-image-classification-with-keras-flow-from-directory-and-generators-95f75ebe5720 [accesed on 11 June 2024]

The Annex

Preparing dataset function

```
def prepare_dataset(original_directory, base_directory):
    #If the folder already exists, remove everything
    if os.path.exists(base_directory):
        shutil.rmtree(base_directory)

    #Recreate the base folder
    os.mkdir(base_directory)

    #Create the training folder in the base directory
    train_directory = os.path.join(base_directory, 'train')
    os.mkdir(train_directory)

    #Create the validation folder in the base directory
    validation_directory = os.path.join(base_directory, 'validation')
    os.mkdir(validation_directory)

    #Create the test folder in the base directory
    test_directory = os.path.join(base_directory, 'test')
    os.mkdir(test_directory)

    #Create the positive/negative folders in training/validation directories
    train_positive_directory = os.path.join(train_directory, 'positive')
    os.mkdir(train_positive_directory)

    train_negative_directory = os.path.join(train_directory, 'negative')
    os.mkdir(train_negative_directory)

    validation_positive_directory = os.path.join(validation_directory,
'positive')
    os.mkdir(validation_positive_directory)

    validation_negative_directory = os.path.join(validation_directory,
'negative')
    os.mkdir(validation_negative_directory)

    test_directory = os.path.join(test_directory, 'test_folder')
    os.mkdir(test_directory)

    #Shuffle and split the positive files
    positive_files = os.listdir(os.path.join(original_directory, 'positive'))
    random.shuffle(positive_files)
    num_positive_files = len(positive_files)
    num_train_positive_files = int(0.6 * num_positive_files)
    num_validation_positive_files = int(0.2 * num_positive_files)

    for fname in positive_files[:num_train_positive_files]:
        src = os.path.join(original_directory, 'positive', fname)
        dst = os.path.join(train_positive_directory, fname)
        shutil.copyfile(src, dst)

    for fname in
positive_files[num_train_positive_files:num_train_positive_files +
num_validation_positive_files]:
        src = os.path.join(original_directory, 'positive', fname)
        dst = os.path.join(validation_positive_directory, fname)
        shutil.copyfile(src, dst)

    for fname in positive_files[num_train_positive_files +
num_validation_positive_files:]:
```

```
        src = os.path.join(original_directory, 'positive', fname)
        dst = os.path.join(test_directory, fname)
        shutil.copyfile(src, dst)

    #Shuffle and split the negative files
    negative_files = os.listdir(os.path.join(original_directory, 'negative'))
    random.shuffle(negative_files)
    num_negative_files = len(negative_files)
    num_train_negative_files = int(0.6 * num_negative_files)
    num_validation_negative_files = int(0.2 * num_negative_files)

    for fname in negative_files[:num_train_negative_files]:
        src = os.path.join(original_directory, 'negative', fname)
        dst = os.path.join(train_negative_directory, fname)
        shutil.copyfile(src, dst)

    for fname in
negative_files[num_train_negative_files:num_train_negative_files +
num_validation_negative_files]:
        src = os.path.join(original_directory, 'negative', fname)
        dst = os.path.join(validation_negative_directory, fname)
        shutil.copyfile(src, dst)

    for fname in negative_files[num_train_negative_files +
num_validation_negative_files:]:
        src = os.path.join(original_directory, 'negative', fname)
        dst = os.path.join(test_directory, fname)
        shutil.copyfile(src, dst)

    #Print the number of images in each directory
    print('Total training positive images:',
len(os.listdir(train_positive_directory)))
    print('Total training negative images:',
len(os.listdir(train_negative_directory)))
    print('Total validation positive images:',
len(os.listdir(validation_positive_directory)))
    print('Total validation negative images:',
len(os.listdir(validation_negative_directory)))
    print('Total test images:', len(os.listdir(test_directory)))

    return
```

Model function

```
def create_model():
    model = models.Sequential([
        layers.Conv2D(32, (3, 3), activation='relu', input_shape=(224, 224, 3)),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(128, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(128, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(512, activation='relu',
kernel_regularizer=regularizers.l2(0.001)),  # L2 regularization
        layers.Dropout(0.5),
        layers.Dense(1, activation='sigmoid')
    ])
```

```
    model.compile(loss='binary_crossentropy',
                  optimizer=Adam(learning_rate=1e-4),
                  metrics=['accuracy'])
    return model
```

Train model function

```
def train_model(model, train_dir, validation_dir):
    # Augmentation
    train_datagen = ImageDataGenerator(
        rescale=1. / 255,
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest'
    )

    validation_datagen = ImageDataGenerator(rescale=1. / 255)

    train_generator = train_datagen.flow_from_directory(
        train_dir,
        target_size=(224, 224),
        batch_size=32,
        shuffle=True,
        seed=42,
        class_mode='binary')

    validation_generator = validation_datagen.flow_from_directory(
        validation_dir,
        target_size=(224, 224),
        batch_size=32,
        shuffle=True,
        seed=42,
        class_mode='binary')

    train_generator = repeat_generator(train_generator)
    validation_generator = repeat_generator(validation_generator)

    steps_per_epoch = int(3547/32)
    validation_steps = int(3547/32)

    early_stopping = EarlyStopping(monitor='val_loss', patience=10,
restore_best_weights=True)

    history = model.fit(
        train_generator,
        steps_per_epoch=steps_per_epoch,
        epochs=10,
        validation_data=validation_generator,
        validation_steps=validation_steps,
        callbacks=[early_stopping])

    return model, history
```

Evaluate model and save model functions

```
def evaluate_model(model, test_dir):
    test_datagen = ImageDataGenerator(rescale=1. / 255)
    test_generator = test_datagen.flow_from_directory(
        test_dir,
        target_size=(224, 224),
        batch_size=32,
        shuffle=False,
        seed=42,
        class_mode='binary')

    test_steps = int(test_generator.samples / test_generator.batch_size)

    test_loss, test_accuracy = model.evaluate(test_generator,
                                              steps=test_steps)
    print(f'Test accuracy: {test_accuracy:.4f}')
    print(f'Test loss: {test_loss:.4f}')

    return test_generator

def save_model(model, model_path):
    model.save(model_path)
```

Main

```
def main():
    original_directory = 'faces_all'
    base_directory = 'faces_prepared'
    model_path = 'My_model.h5'

    prepare_dataset(original_directory, base_directory)

    model = create_model()

    train_dir = os.path.join(base_directory, 'train')
    validation_dir = os.path.join(base_directory, 'validation')
    test_dir = os.path.join(base_directory, 'test')

    model, history = train_model(model, train_dir, validation_dir)
    save_model(model, model_path)

    acc = history.history['accuracy']
    val_acc = history.history['val_accuracy']
    loss = history.history['loss']
    val_loss = history.history['val_loss']

    epochs = range(len(acc))
    plt.plot(epochs, acc, 'r', label='Training accuracy')
    plt.plot(epochs, val_acc, 'g', label='Validation accuracy')
    plt.title('Training and validation accuracy')
    plt.legend()
    plt.figure()
    plt.plot(epochs, loss, 'r', label='Training loss')
    plt.plot(epochs, val_loss, 'g', label='Validation loss')
    plt.title('Training and validation loss')
    plt.legend()
    plt.show()

    # Evaluate the model on the test set
    evaluate_model(model, test_dir)
```

Load and preprocess image function

```
def load_and_preprocess_image(img_path, target_size=(224, 224)):
    # Preprocessing the image for the model
    img_keras = image.load_img(img_path, target_size=target_size)
    img_tensor = image.img_to_array(img_keras)  # Convert to tensor
    img_tensor = np.expand_dims(img_tensor, axis=0)  # Add batch dimension
    img_tensor /= 255.0  # Rescale to [0, 1]

    # Load the original image for displaying
    img_cv2 = cv2.imread(img_path)

    return img_cv2, img_tensor
```

Predict image function

```
def predict_image(model, img_path):
    original_img, img_tensor = load_and_preprocess_image(img_path)

    # Make prediction
    prediction = model.predict(img_tensor)
    confidence = prediction[0][0]
    print(f"Prediction: {'Face Detected' if confidence > 0.5 else 'No Face
Detected'} (Confidence: {confidence})")

    if confidence > 0.5:
        # Use Haar Cascade to draw a rectangle around the face
        gray_img = cv2.cvtColor(original_img, cv2.COLOR_BGR2GRAY)
        faces = face_cascade.detectMultiScale(gray_img, scaleFactor=1.1,
minNeighbors=5, minSize=(30, 30))

        for (x, y, w, h) in faces:
            cv2.rectangle(original_img, (x, y), (x+w, y+h), (255, 0, 0), 2)

    # Convert image back to uint8 for display with cv2 and matplotlib
    original_img_uint8 = original_img.astype(np.uint8)

    # Display the image with matplotlib
    plt.imshow(cv2.cvtColor(original_img_uint8, cv2.COLOR_BGR2RGB))
    plt.title(f"Prediction: {'Face Detected' if confidence > 0.5 else 'No Face
Detected'}")
    plt.show()
```

Detect face live function

```
def detect_face_live():
    stop_camera = False
    cap = cv2.VideoCapture(0)

    while not stop_camera:
        ret, frame = cap.read()
        if not ret:
            break

        frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        frame_resized = cv2.resize(frame_rgb, (224, 224))
        frame_tensor = image.img_to_array(frame_resized)
        frame_tensor = np.expand_dims(frame_tensor, axis=0)
        frame_tensor /= 255.0
```

```
        # Predict using the model
        prediction = face_detection_model.predict(frame_tensor)
        confidence = prediction[0][0]

        if confidence > 0.5:
            gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray_frame, scaleFactor=1.1,
minNeighbors=5, minSize=(30, 30))

            for (x, y, w, h) in faces:
                cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 0, 0), 2)
                label = "Face Detected"
                cv2.putText(frame, label, (x, y-10), cv2.FONT_HERSHEY_SIMPLEX,
0.9, (255, 0, 0), 2)
        else:
            label = "No Face Detected"

        # Display the frame
        cv2.imshow("Face Detection", frame)

        # Break the loop if 'q' is pressed
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cap.release()
    cv2.destroyAllWindows()
```

Detect faces in video function

```
def detect_faces_in_video(video_path):
    cap = cv2.VideoCapture(video_path)
    while cap.isOpened():
        ret, frame = cap.read()
        if not ret:
            break

        frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        frame_resized = cv2.resize(frame_rgb, (224, 224))
        frame_tensor = image.img_to_array(frame_resized)
        frame_tensor = np.expand_dims(frame_tensor, axis=0)
        frame_tensor /= 255.0

        # Predict using the model
        prediction = face_detection_model.predict(frame_tensor)
        confidence = prediction[0][0]

        if confidence > 0.5:
            gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray_frame, scaleFactor=1.1,
minNeighbors=5, minSize=(30, 30))

            for (x, y, w, h) in faces:
                cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 0, 0), 2)
                label = "Face Detected"
                cv2.putText(frame, label, (x, y-10), cv2.FONT_HERSHEY_SIMPLEX,
0.9, (255, 0, 0), 2)
        else:
            label = "No Face Detected"

        # Display the frame
        cv2.imshow("Face Detection", frame)
```

```
        # Break the loop if 'q' is pressed
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cap.release()
    cv2.destroyAllWindows()
```

GUI functionality

```
# Load the pre-trained model
face_detection_model = load_model('GPT_model.h5')

# Create GUI
root = Tk()
root.title("Face Detection Application")
root.geometry("600x400")
root.configure(bg="#2e3f4f")

frame = Frame(root, bg="#4e5d6c", bd=5, relief="groove")
frame.place(relx=0.5, rely=0.5, anchor="center")

button_browse = Button(frame, text="Browse Image", command=browse_files,
width=20, height=2, bg="#e1e5ea", fg="black", font=("Arial", 12, "bold"), bd=3,
relief="ridge")
button_browse.grid(row=0, column=0, padx=10, pady=10)

button_camera = Button(frame, text="Open Camera", command=detect_face_live,
width=20, height=2, bg="#e1e5ea", fg="black", font=("Arial", 12, "bold"), bd=3,
relief="ridge")
button_camera.grid(row=1, column=0, padx=10, pady=10)

button_video = Button(frame, text="Browse Video", command=browse_video_file,
width=20, height=2, bg="#e1e5ea", fg="black", font=("Arial", 12, "bold"), bd=3,
relief="ridge")
button_video.grid(row=2, column=0, padx=10, pady=10)

root.mainloop()
```