TALLINNA TEHNIKAÜLIKOOL
INSENERITEADUSKOND

Elektroenergeetika ja mehhatroonika instituut

NÄOTUVASTUST RAKENDAV PROTOTÜÜP TURVASÜSTEEM KONTORILE

A prototype office security system using facial identification

BAKALAUREUSETÖÖ

Üliõpilane:	Erki Meinberg
Üliõpilaskood:	164390
Juhendaja:	Dmitry Shvarts, teadur

Tallinn 2019

(Tiitellehe pöördel)

AUTORIDEKLARATSIOON

Olen koostanud lõputöö iseseisvalt.
Lõputöö alusel ei ole varem kutse- või teaduskraadi või inseneridiplomit taotletud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on viidatud.

“.......” .... 201…..

Autor: ....
/ allkiri /

Töö vastab bakalaureusetöö/magistritööle esitatud nõuetele

“.......” .... 201…..

Juhendaja: ....
/ allkiri /

Kaitsmisele lubatud

“.......”...................201… .

Kaitsmiskomisjoni esimees ....
/ nimi ja allkiri /

Elektroenergeetika ja mehhatroonika instituut

LÕPUTÖÖ ÜLESANNE

Üliõpilane:	Erki Meinberg, 164390
Õppekava, peaeriala:	MAHB02/13 - mehhatroonika
Juhendaja(d):	Teadur, Dmitry Shvarts, 58509613

Lõputöö teema:

(eesti keeles)	Näotuvastust rakendav prototüüp turvasüsteem kontorile
(inglise keeles)	A prototype office security system using facial identification

Lõputöö põhieesmärgid:
1. Luua prototüüp turvasüsteemist kontorile, mis rakendab näotuvastust kasutajate tuvastamiseks.

Lõputöö etapid ja ajakava:

Nr	**Ülesande kirjeldus**	**Tähtaeg**
1.	Uurida ja tuvastada olemasolevaid näotuvastus algoritme.	01.04.2019
2.	Võrrelda olemasolevaid näotuvastus algoritme ning nende sobivust töö eesmärgiks.	14.04.2019
3.	Prototüüpsüsteemile lõpliku algoritmi ja arhitektuuri loomine.	28.04.2019
4.	Prototüüpsüsteemi arhitektuuri rakendamine ja testimine.	12.05.2019
5.	Töö vormistamine, köitmine.	21.05.2019

Töö keel: inglise **Lõputöö esitamise tähtaeg:** “21” 05 2019 a

Üliõpilane: Erki Meinberg .... “15” 03 2019 a

/allkiri/

Juhendaja: Dmitry Shvarts .... “15” 03 2019 a

/allkiri

Kinnise kaitsmise ja/või avalikustamise piirangu tingimused formuleeritakse pöördel

CONTENTS

PREFACE

The subject of this thesis was originally proposed by research scientist Dmitry Shvarts, of the Tallinn University of Technology.

The author of this thesis would like to thank his instructor, research scientist Dmitry Shvarts, for the positive feedback and assistance while the thesis was being worked on. The author would also like to thank his colleagues over at the TUT Robotics Club, who provided excellent moral support for the duration of the work.

List of abbreviations and symbols

ARM	Acorn RISC Machine
CNN	Convolutional neural network
GUI	Graphical user interface

SISSEJUHATUS

Närvivõrkude rakendamine näotuvastuses on mitmeid aastakümneid juba uuritud probleem olnud [1]. Alates konvolutsiooniliste närvivõrkude kasutuselevõtmisest on neid rakendavate näotuvastussüsteemide täpsus hüppeliselt tõusnud [2]. Sellest tulenevalt on närvivõrkudel baseeruvaid isikutuvastussüsteeme hakatud aina rohkem tarbesüsteemides rakendama.

Käesoleva lõputöö eesmärk on koostada algoritm näotuvastust rakendava turvasüsteemi jaoks. Lõputöö teema on välja käidud teadur Dmitry Shvartsi poolt ning oli algselt mõeldud kui ühe komponendina suuremasse, nö. „tarka“ turvasüsteemi, mis rakendaks pilditöötlust ka muudel eesmärkidel.

Töö teiseks eesmärgiks on ka välja töötatud algoritm implementeerida. Seda tehtakse Python programmeerimiskeeles, kasutades laialt levinud teeke, nagu OpenCV ja NumPy. See võimaldab süsteemi kerged liidestamist teiste süsteemidega ning ka kerget laiendamist ja muutmist.

Lõputöö esimeses peatükis antakse ülevaade olemasolevast teadmusest närvivõrkude rakendamisel näotuvastuse eesmärkidel. Antakse üldise tausta ülevaade, kirjeldatakse täpsemalt eesmärgiks kasutatavate närvivõrkude struktuure ja omadusi ning tuuakse välja metoodika närvivõrkude rakendamisel näotuvastuseks.

Lõputöö teises peatükis teostatakse võrdlus kahe erineva näotuvastusalgoritmi vahel. Mõlemad meetodid baseeruvad esimeses peatükis välja toodud alustel. Võrreldakse algoritmide täpsust, töökindlust ning töökiirust. Peatüki alusel valitakse töö käigus koostatud prototüüpsüsteemi jaoks sobiv algoritm.

Kolmas peatükk defineerib ära prototüüpsüsteemi nõuded. Samuti kirjeldatakse ära prototüüpsüsteemi arhitektuuriline lahendus, mis arendati vastavalt nendele nõuetele. Arhitektuuris rakendatakse eelnevas peatükis valitud metoodikat.

Neljas peatükk kirjeldab kolmandas peatükis esitletud arhitektuuri implementeerimist Python programmeerimiskeeles. Täpsustatakse paari tähtsama detaili rakendust, näiteks närvivõrgu rakendust, katsetatakse süsteemi vastavust nõuetele ning tuuakse välja ka võimalusi edasiseks arenduseks.

Lõputöö lisadena on toodud viimases peatükis kirjeldatud implementatsiooni lähtekood Python programmeerimiskeeles.

INTRODUCTION

The use of neural networks for the purposes of facial identification has been the subject of research for many decades [1]. The introduction of convolutional neural networks provided a noticeable jump in the capabilities of neural networks when applied to image processing, and this includes their application for facial identification [2].

With that in mind, goal of the present work is to develop a prototype security system that utilizes facial identification. An example use-case for such a system would be to secure an office or a room, as a replacement for other methods that may rely on passcodes or physical devices, such as RFID cards. The topic was proposed by research scientist Dmitry Shvarts, of the Tallinn University of Technology. The system was originally proposed as a key component of a smart security system, which relied on image processing for various tasks, one of which was the identification of people in a secured area.

The software architecture developed over the course of this work will also be implemented as a prototype using the Python programming language. The implementation will be evaluated once complete, with shortcomings and points for future development identified. The Python programming language will be used to write most of the software in question, along with the use of common libraries for image processing and the use of neural networks, such as OpenCV, NumPy, and dlib.

To achieve these two goals, the first chapter of the work will provide a review of associated literature, with the intent to identify existing methods for applying neural networks for facial identification. We will determine contemporary approaches toward the goal and select methodologies for evaluation and application in the later chapters. This includes providing an overview of neural network architectures and how they are applied.

The second chapter will conduct the testing of various methods for facial identification identified in the first chapter. The objective is to identify which contemporary neural network architectures are best for creating a secure and reliable system for use in the application described.

In the third chapter, we will be outlining the requirements for the system to be created and creating the system architecture based off that. We will be integrating the most appropriate pipeline, as identified in the second chapter, and working off the methods explored in the first chapter. The outcome of this chapter will be the general architecture of a system to fulfil the main goal of the work.

The fourth chapter will concern the implementation of the system as a prototype, and evaluation of it. The system will be evaluated with a live camera feed. Shortcomings will be identified, as well as the potential for future use and deployment.

The appendices of the work will include the source code produced while implementing the architecture described in the third chapter of the work.

1. INTRODUCTION

Facial recognition from both static images and live camera feeds has been the subject of research for many years. Older approaches tend to be procedural, such as processing of a face into a set of eigen-vectors, known as eigenfaces; or the matching of graphs [3]. And in recent years, the application of neural networks has become highly researched and optimal for both general image classification, as well as for facial recognition in specific [4] [5]. As such, we will be picking them as the main form of algorithm for handling facial recognition in this work. An investigation for their application for this specific task follows in this chapter.

1.1 Application of neural networks in facial recognition

As per [1], the application of neural networks for the purposes of machine face recognition has been investigated for close to 40 decades now. In this time, multiple different techniques have been used and error rates under controlled conditions have reached 1 % or less by the year 2006 [6]. Considering this, more recent research, such as [7], has been conducted with the intent of making facial identification via neural networks more robust and thus, usable under wider circumstances.

A common modern approach to solve the issue of facial identification has been using convolutional neural networks [5]. Works such as [4] have iterated or specialized these designs. In parallel, CNN architectures using deeper construction (more layers) have also been employed, to further improve accuracy over a normal CNN [5]. The deeper networks are generally more difficult to train, however, which prompted the application of residual layers in addition to a deep CNN architecture. This makes training easier and allows for the minimization of error rates with deeper architectures, in comparison to thinner networks [8].

1.1.1 Convolutional neural networks

Due to the prolific use of CNNs in the field of facial recognition [4] [5], we will be taking a closer look at them for use in this work. CNNs generally use multiple convolutional layers, which reduce the input image's dimensionality, extract specific features, and generate a linear output vector based on the extracted features. Typically, multiple convolutional layers are utilized. [3]

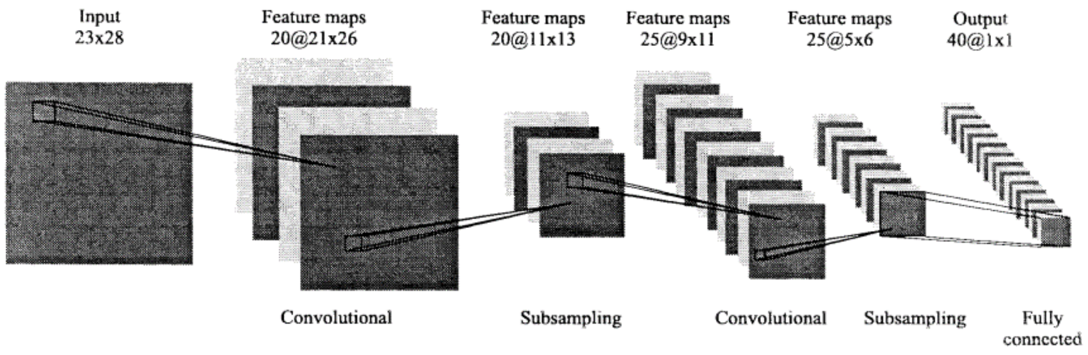


Figure 1.1. A typical convolutional network [3].

The pipeline for employing CNNs for facial recognition is described in Figure 1.2. Depending on the structure of the neural network, it may be that instead of raw images being given to the neural network, an already processed set of descriptors is provided. Such descriptors include histograms of oriented gradients (HOG), and are commonly generated by handcrafted algorithms. This is common for shallow neural networks. Deep neural networks usually manage feature extraction themselves. [5]

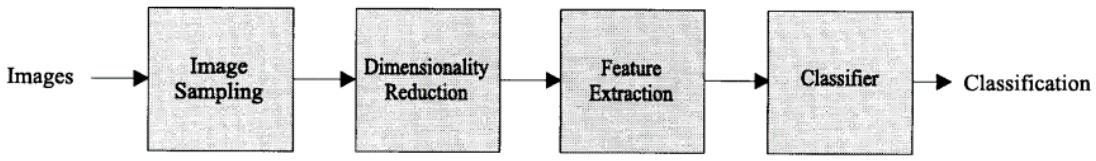
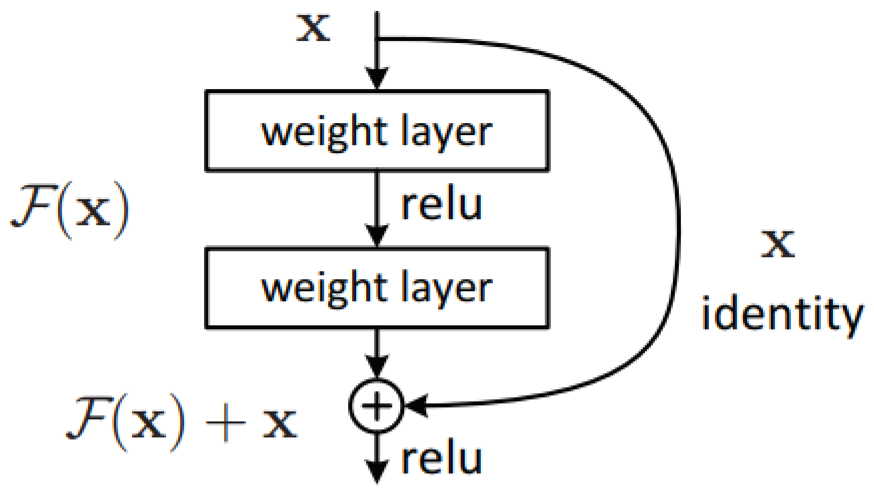


Figure 1.2. An example pipeline for using neural networks for face recognition as seen in [3].

The output of a CNN is a classification. Depending on how the neural network is specified, the classifier may be a confidence value or a more generalized output describing features of the input. In the case of facial recognition, in order to avoid the issue of having to retrain the network every time a new face is introduced, a more generalized classification approach is sought after. [2] One such method, suitable for the purposes of this thesis, will be described in section 1.2.

1.1.2 Residual neural networks

With the application of deeper CNNs in the field of image processing, it was noted that training them is more difficult than regular CNNs. [8] points out that a 20-layer CNN for general image recognition can more easily achieve a lower test error than a 56-layer deep CNN. As a solution, [8] proposes the usage of residual layers to help better train the deeper CNNs.



x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 1.3. The principle behind deep residual learning [8].

As shown in Figure 1.3, the premise behind residual learning layers is the passing of an identity x via a so-called shortcut connection to a deeper layer’s input. This allows the solver functions used in training the CNN to more easily adjust the deeper layers of the neural network, without incurring the problems of extreme gradients while learning. [8]

1.2 Facial embedding encoding

The system being proposed in this thesis must be capable of matching an unknown face against a known set of faces within some database. There must also be some method of knowing that the face is not present within the database. [2] and [5] propose the usage of facial embeddings: the mapping of a face’s feature set into an d-dimensional space.

The general process can be described by the expression $f(x) \in \mathbb{R}^d$, wherein the function $f(x)$ maps the image x into the d-dimensional space $\mathbb{R}$. The function itself is realized through the application of neural networks [2]. This method can achieve accuracy of well over 90 %, as per the results of [5].

Such a method also avoids the issue of one-shot training [9]. The issue would arise if we were to use a neural network that is simply trained to identify a specific set of faces from a camera image. In which case, every new face would incur the retraining of the network. The application of embeddings allows us to assign a specific identity to any given face, without the need to retrain the network.

The resulting embeddings are also relatively compact in terms of memory utilization. The method outlined in [2] can produce a vector that is only 128-bytes long. This is a very good quality to have for a system that runs on limited hardware, such as an embedded security system. [2] Makes the additional note that a larger embedding does not necessarily improve the accuracy of the neural network. Though, as they note, this may be since a larger embedding requires more training or a deeper structure of the CNN.

1.2.1 Usage of facial embedding encoding to recognize a known face

As pointed out by [2], the usage of such embeddings reduces the task of verifying a face (knowing whether or not the face on a given image is the same as another face) to finding the Euclidian distance between two embeddings and comparing that to a threshold. Two identical embeddings represent a completely identical face, and embeddings which are close to each other in the multidimensional space represent similar faces.

The act of recognizing a face, that is, figuring out who is present on the picture, is a k-nearest neighbour problem using embeddings, wherein the known embedding nearest to the unknown embedding is most likely to be the face of the subject [2]. This result must also be verified as per the previous paragraph.

As per the example [10], a neural network may also be applied to this problem. Doing so is a method of solving the k-nearest neighbour problem. It replaces the distance threshold with a confidence threshold against which the output of the evaluating neural network is compared. However, this would introduce the one-shot training problem defined in section 1.2, and is not suitable for our purposes.

1.3 Conclusion

From this chapter, we have determined that one of the most common contemporary neural network architectures to apply for the purpose of facial identification is the convolutional neural network. With the deeper variants of it, including residual networks, further increasing their capabilities. Based on this, we will be focusing on these architectures in our following chapters, with the goal being to test and evaluate existing implementations of them for use in the system that will be outlined later in this work.

We also provided an overview of the idea of using facial embeddings to allow for the identification of any given face, as outlined in [2]. This method will be utilized by the system as the primary means of facial identification within this work.

The following chapters will now provide an overview and comparison of actual implementations of these ideas. With the final implementation detailed later in this work utilizing these implementations.

2. COMPARISON OF SOME AVAILABLE FACIAL RECOGNITION ALGORITHMS

For the purposes of this work, ready-made facial recognition systems were selected and analysed. The purpose of this analysis was to learn about the different practical methods for applying neural networks in facial recognition. The results of this comparison will be taken into account in the design of the prototype system, with the prototype system's architecture being described in chapter 3 of this work.

The two examples being evaluated were [10] and [11]. Both are written in the Python programming language, using publicly available libraries and pre-trained neural networks. This makes the testing and further development of the two systems relatively easy, and allows for the rapid prototyping of our own system.

Both example systems also output a 128-dimensional vector, much like was described in [2]. This allows us to use the same custom verification system applied to the output of either system.

2.1 Methodology

Both systems will be tested with the same pre-recorded input and using the same output verification method. The testing framework can be described with the following drawing:

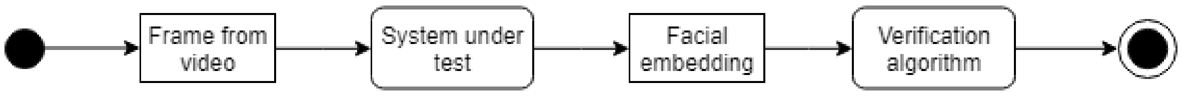


Figure 2.1. An UML acitivity diagram describing the framework to be used for testing.

The pre-recorded video stream is that of the author presenting his face at various angles to the camera. The video stream is 422 frames long, lasts 13 seconds, and is recorded at a resolution of 1280x720.

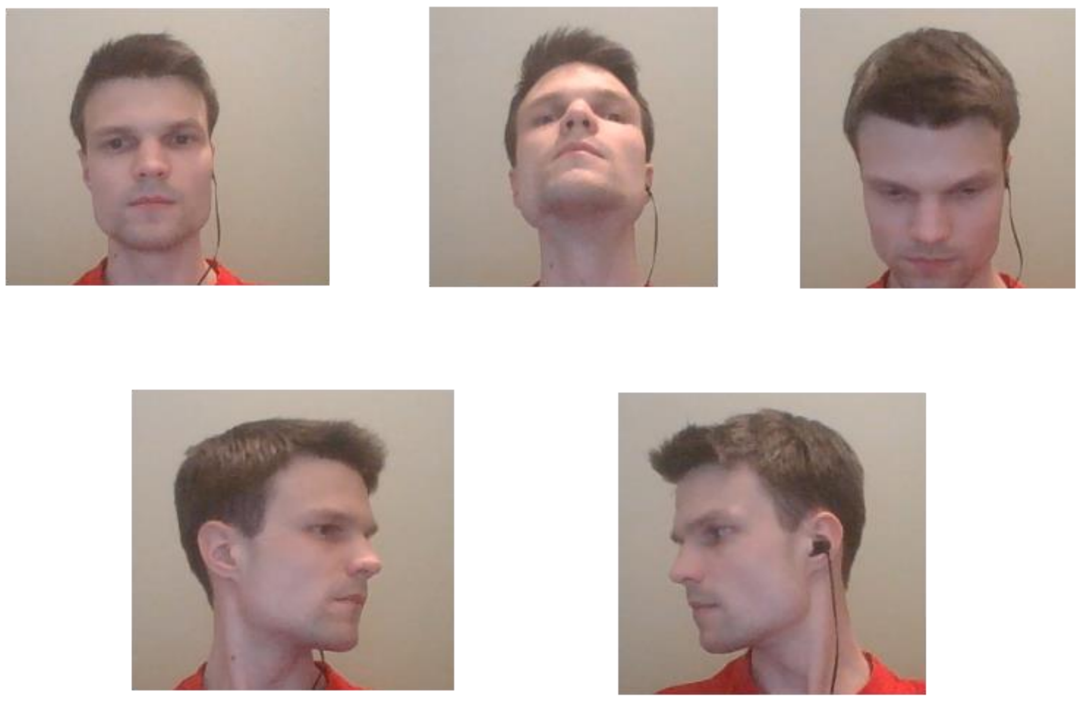


Figure 2.2. Examples of frames from the test footage, indicating the face to be identified at different angles.

2.1.1 Embedding verification

The algorithm used for facial verification is described more in-depth in item 3.4.2 of this work. As a short overview, however, we use a store of known embeddings that are pre-calculated with the neural network currently being tested. The Euclidean distance between the unknown embedding and each of the known embeddings is calculated. The first known embedding for which the distance is below a selected threshold is selected as a positive identification. In the case of these tests, the threshold distance was set to 0.6. The recommendation was based off of the documentation in both examples, [10] [11].

For the purposes of potentially catching false-positive identifications, the faces of 5 different people were included in the data store.

2.1.2 Identification accuracy

In the tests, we are concerned with the two following quantities: error rate and processing speed. Due to the nature of the system being developed, error rate is our first and foremost concern. In our case, error rate is defined as the number of erroneous identifications over the number of total identifications that pass the validator's threshold:

$$x_{error} = \frac{n_{neg}}{n_{det}} \cdot 100\ \% \tag{2.1}$$

where: x_{error} is the accuracy of the system,

$n_{negative}$ is the number of erroneous identifications recorded over the course of a test,

n_{det} is the number of frames wherein the system detected a face.

This measurement assists us in gauging and verifying the accuracy of both the system under test, as well as our identity verification method.

Of interest, also, is the detection rate of the system, calculated as follows:

$$x_{detection} = \frac{n_{det}}{422} \cdot 100\ \% \tag{2.2}$$

Where: $x_{detection}$ is the detection rate of the system,

n_{det} is the number of frames wherein the system detected a face.

And finally, we will also be comparing the reliability of the system by calculating the positive identification rate:

$$x_{reliability} = \frac{n_{pos}}{422} \cdot 100\ \% \tag{2.3}$$

Where: $x_{reliability}$ is the detection rate of the system,

n_{pos} is the number of positive identifications recorded over the course of a test.

2.1.3 Processing speed

The second quantity being measured, as already mentioned, is the processing speed of the system under test. Processing speed is an important variable to ensure in the responsiveness of the system. In each test, the processing time of each component is recorded per frame processed, and an average processing time per frame is calculated from the total frames processed. This will give us an understanding of what components of the system are likely to produce a bottle neck, while also allowing us to determine if the system being tested is fast enough for real time application.

It should be noted that during some tests, not all frames were fully processed. Specifically, in cases where the facial detection algorithm fails to identify a valid face, the rest of the algorithm are skipped for the duration of that frame. As such, while calculating average processing speed of various components, only the frames that those components processed were considered. This is also the reason why a whole system frames per second indicator is not taken into consideration: a system which identifies the subject's presence in fewer frames is likely to speed up due to the facial identification algorithms being skipped, thus increasing the general frames per second count.

2.2 Overview of systems under test

2.2.1 Libraries used

These examples required the following Python libraries to be installed using the pip library manager:

- opencv-python, version 4.0.0.21;
- dlib, version 19.17.0;
- imutils, version 0.5.2;
- numpy, 1.16.2.

2.2.2 Example 1, using OpenFace with OpenCV

The first system evaluated was presented as an example in [10]. As per the article, the example uses an implementation of the FaceNet paper [2], called OpenFace [12], as its core component for producing facial embeddings. The whole process flow is as follows:

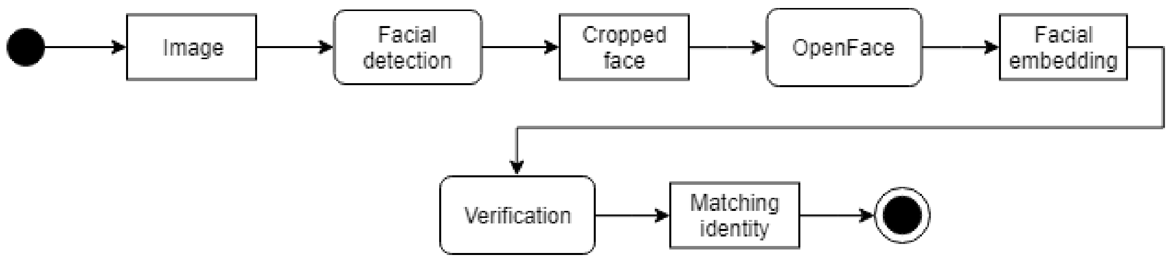


Figure 2.3. An UML activity diagram describing the general process of the OpenFace based example system.

Facial detection is accomplished via a smaller pretrained neural network, utilizing a pretrained Caffe deep learning model. The model itself is provided by the OpenCV library. The purpose of this smaller deep learning model is to locate and extract the locations of all faces within a frame, so that they may be cropped out and provided as an input to the OpenFace model. [10]

During general testing, it should be noted that the facial detection model utilized by this example had an issue of falsely detecting certain geometry as faces. This would result in the OpenFace model processing data that it was not trained to classify, and thus providing an unknown output.

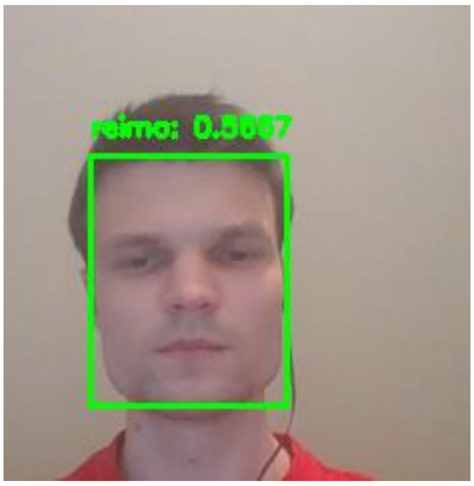


Figure 2.4. A false positive identification. The name indicated above the identification rectangle is incorrect.

OpenFace, the model used for calculating the facial embeddings, is a deep CNN, based on the model proposed in [2], as already noted. According to the implementation's own description at [12], the accuracy of these models ranges from 76.12 % to 92.92 %. The model specifically used in this example, "nn4.small2.v1", is cited as having an accuracy of 92.92 % [13]. The output of the network is a 128-dimensional facial embedding [10] [12].

Identification within the example was handled by a custom trained SVM model [10]. However, this required that the model be retrained every time a new face was added to the database or removed. As such, the SVM model was removed in favour of the custom verification process.

2.2.3 Example 2, using ResNet with dlib

The second system to be evaluated is based on the ResNet model, originally described in [8]. The original implementation of this example can be found in [11]. The example was originally implemented as only working on static images, so some refactoring was required to make it also function on a video stream. The general process flow of this example is as follows:

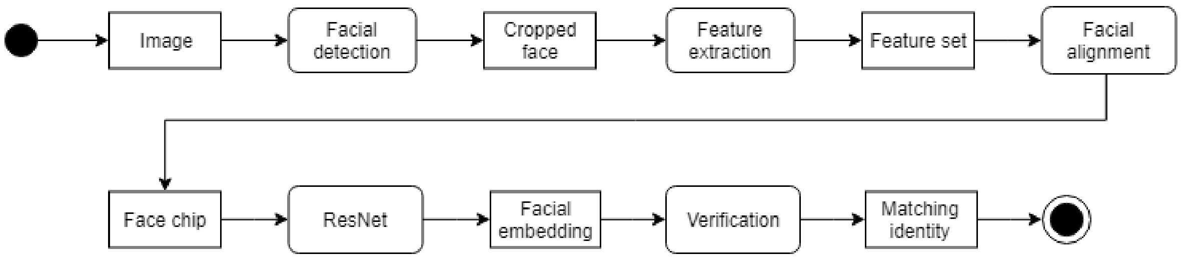


Figure 2.5. An UML activity diagram showcasing the operating logic of the second example.

Facial detection is accomplished by using the frontal face detector embedded into the dlib library. As per [14], the detector utilizes HOG features which are then classified using a linear classifier with extra post-processing.

Feature extraction in the case of this model is done outside of the main classifier. In this case, a technique described in [15]. The model uses an ensemble of regression trees to detect up to 194 landmarks from a provided input face. From this, we can also recognize the pose (orientation) of the face, which is used for aligning the face to generate what the example refers to as a face chip.

The ResNet model then takes the extracted and aligned output of the feature extraction phase, the face chip, and computes from it a 128-dimensional vector. The ResNet model is a residual deep CNN. According to the example description [11], the pretrained model has an accuracy of 99.38 % when ran against the LFWB dataset.

2.3 Overview of results

Both systems were subjected to testing, as per the methodology described in section 2.1. The results are displayed and compared in the following section.

2.3.1 Identification accuracy

The identification specific statistics are listed in the following table. As noted previously, the footage itself lasted for 422 frames, and at all times, a face was present in the frame, though perhaps rotated at odd angles.

Table 2.1. Accuracy and identification statistics of the tests.

	OpenFace	**ResNet**
No. frames with a face detected, n_{det}	422	361
No. positive identifications, n_{pos}	133	343
No. false-positive identifications, n_{neg}	47	0

As can immediately be observed, the OpenFace based system produced 47 false identifications: meaning that it incorrectly identified the person in frame to be someone else.

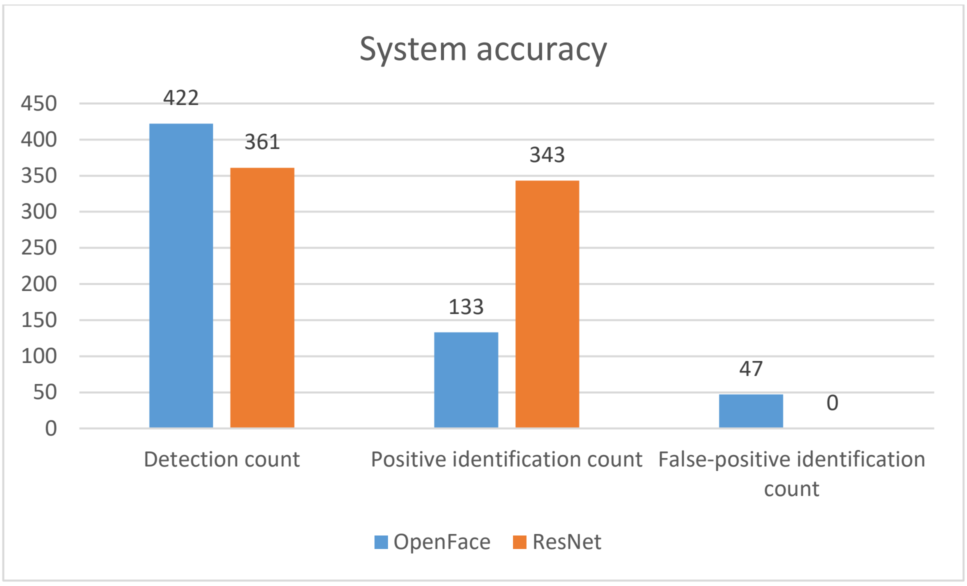


Figure 2.6. Comparative look at the accuracy of the two systems.

From these statistics, we derive the following chart, using the equations specified in item 2.1.2:

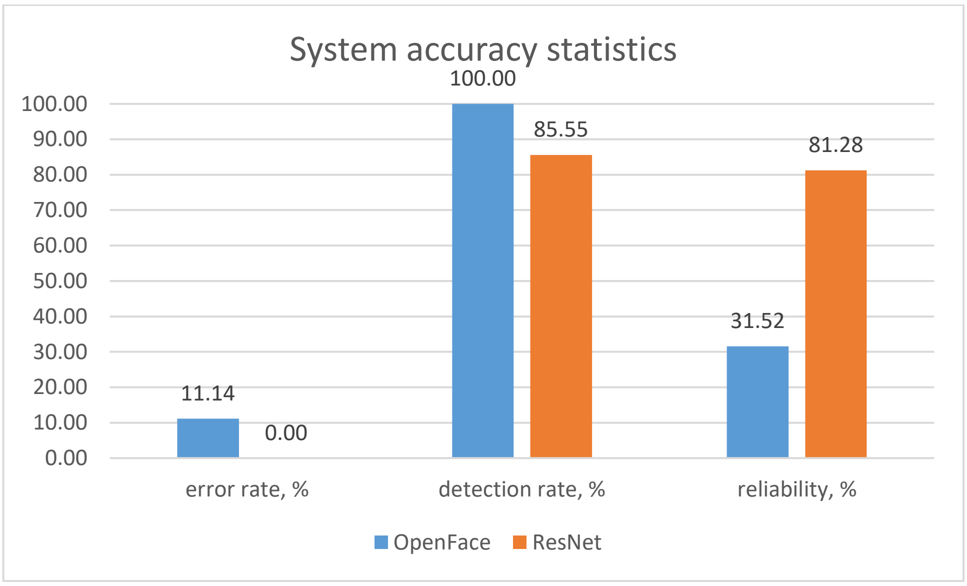


Figure 2.7. Aggregated system accuracy statistics.

The most important statistics to make note of here is the fact that the OpenFace based system incorrectly identified the person from the footage 11,14 % of the time. Of further note is the low reliability, wherein only 31,52 % of the frames were correctly identified as the person required.

2.3.2 Processing speed

As already noted, both systems were tested against the same pre-recorded video stream. The results of profiling measurements were as follows:

Table 2.2. Component execution time profiling results. As per chapter 2.2.3, the chip creation process is only present in the second example.

	OpenFace	**ResNet**
Avg. face detection time, t_{det} ms	49,0	700,0
Avg. chip creation time, t_{chip} ms	N/A	2,8
Avg. embedding creation time, t_{embed} ms	11,0	481,0
Total test duration, t_{total} s	27,86	472,94

As can be observed, the ResNet based example is at least 16 times slower in its execution. The comparison between the average time-per-frame processing can be highlighted as such:

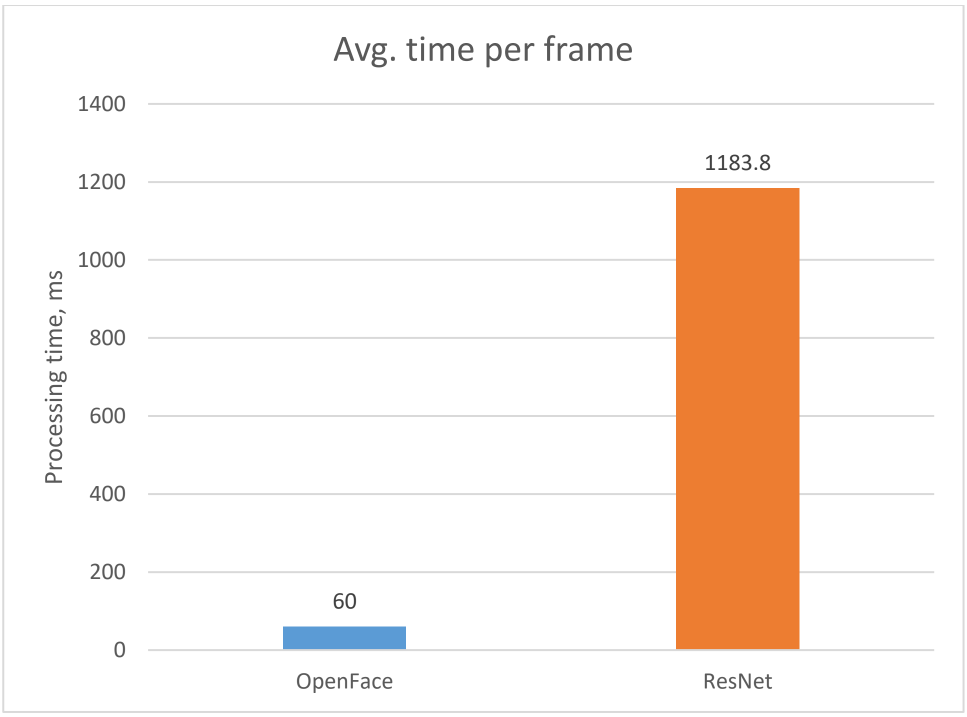


Figure 2.8. A comparison of the average processing time per frame. Calculated as $t = t_{det} + t_{chip} + t_{embed}$.

The ResNet based system ended up being roughly 20 times slower than OpenFace based one. When we compare based on the components, it can be seen that both the face detection model, as well as the embedding creation are multiple times slower for the ResNet based system than for the OpenFace based system.

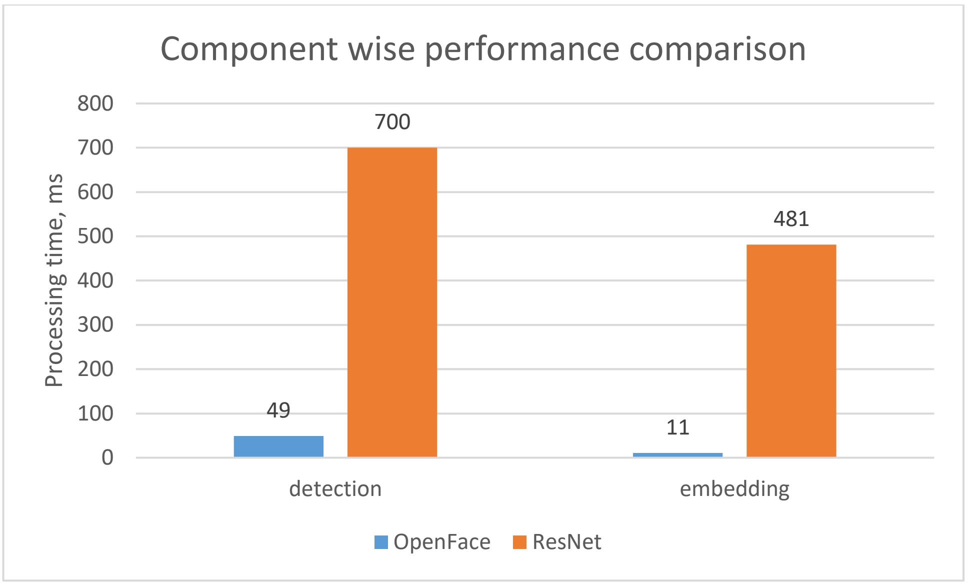


Figure 2.9. A component wise comparison of the average performance of the systems under test.

Component wise, it can be observed that both of the key components of the ResNet based system are multiple times slower, than the OpenFace based system. This is most likely due to the more complex models being employed in the former system.

2.4 Conclusion

There were severe differences in both the accuracy and processing speed of the two examples. As far as accuracy is concerned, the ResNet based system was both more reliable, at 81,28 % reliability, and had no false detections. As opposed to the OpenFace based system, which falsely identified the subject on 11,14 % of the time, and positively identified the subject on only 31,52 % of the frames where a face was detected.

Though this came clearly at the cost of speed. The ResNet based system is slower by a factor of 16. With both the face detection model and embedding calculation model being considerably slower than the simpler OpenFace based ones.

For a security-based application, the error rate is of much more importance than processing speed. As such, for the purposes of this work, the ResNet based model is a lot more suitable. Further, the processing speed approximately 1,2 seconds per frame is not all that detrimental, especially when the higher identification reliability is taken into consideration.

3. PROTOTYPE SYSTEM ARCHITECTURE

With the methods for facial identification investigated and tested, we can now begin to build the architecture around this key component. The following chapter will describe the system architecture produced while composing this work, to accomplish the goals of the thesis. This architecture will then be implemented in chapter 4 of the work.

3.1 System requirements

The general requirements for the system are defined as the following:

- the system must operate on a real time video feed from a camera;
- the system must compare embeddings in a manner as described in [2];
- the system must hold its known embeddings in a datastore;
- the system must output a decision based on whether or not the unknown face is positively identified.

Within the confines of this work, the last aspect of the system is realized as a decision as to whether or not the individual stood before the camera is a known person and should be granted entry to the guarded area. Though in reality, this specific action could be anything, from logging, to unlocking a computer or a device, or whatever.

3.2 System architecture overview

Based on the requirements, a general system drawing was compiled. Ideally, the general architecture is implementation neutral. Meaning that, as long as the data structures within the system remain the same, the method of generating these data structures is irrelevant. This allows for later experimentation with other algorithms, neural networks, and database drivers.

As per the requirements, the input of the algorithm is a camera image from the live camera feed. And the output is either a matching embedding, or a decision made based on that matching embedding. In the case of this example system, the decision is whether or not we can grant entry to a person to a specified area.

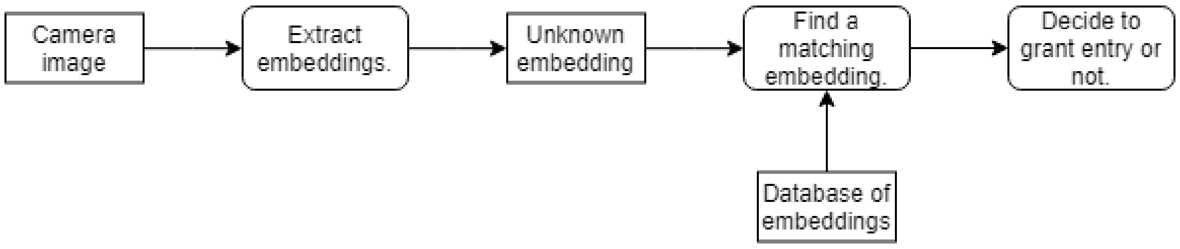


Figure 3.1. A high-level overview of the proposed system.

3.2.1 Internal state machine

Due to the fact that we need to retain state, at least in the case for when the system is "open", a small state machine is also used when implementing the system. The state machine has three states, with their interaction being as follows:

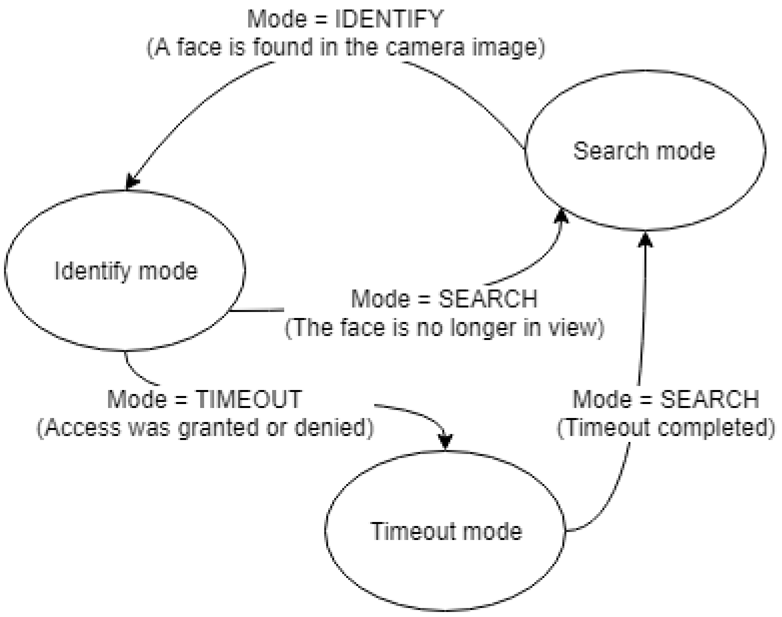


Figure 3.2. System state machine overview.

3.3 System search mode

Search mode is the "at rest" component of the program. It is meant to do minimal processing, only identifying the presence of a face in the camera image, and nothing else. It is expected that the program spends most of its time within this state. The conclusion of the search mode should always be the successful detection of a face from within the current camera frame, at which point identification of the detected face in the identification mode can proceed.

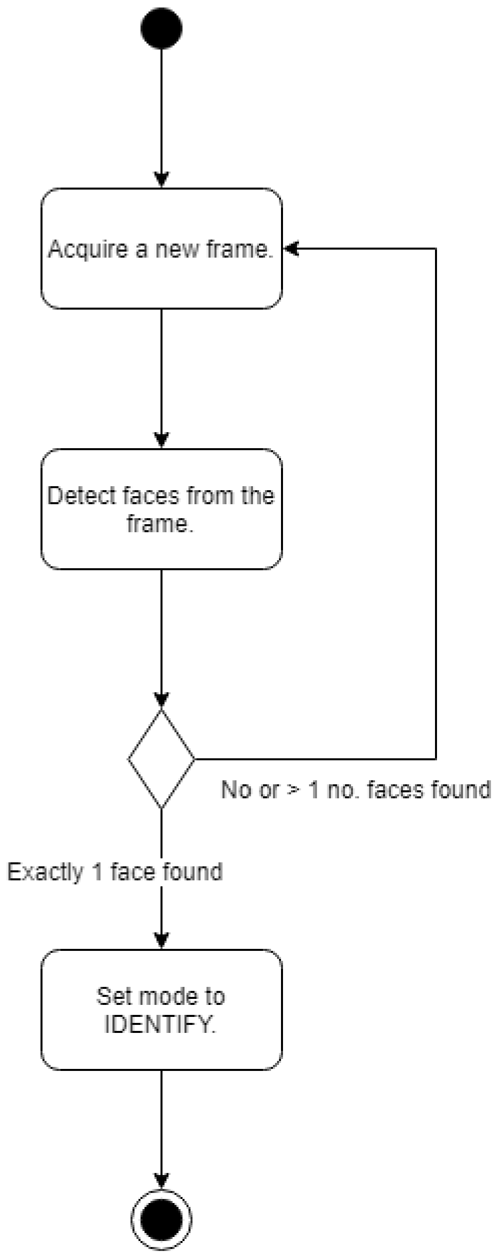


Figure 3.3. An outline of the system's search mode.

The face detection is done with a method described in chapter 2, and the output has to be the faces identified. A minimal neural network can be used to locate and enumerate them.

As can be seen in Figure 3.3, the condition for the system proceeding is set to only one face being present in the camera frame. The reason for restricting the system to just one face in view, is for the sake of security. A situation where multiple faces are permitted to be in frame for identification will cause an indeterminate situation as to which face should be checked for identification.

3.4 System identification mode

Identification mode can be described as the main “work” mode of the program. This mode will require addressing of the embedding database, and is responsible for identifying a face. The conclusion of this state will either the be identification of the face, regardless of whether or not we identify it positively, at which point the program will proceed into the timeout mode; or the loss of the face from the camera frame during the identification process, at which point the program will transition back into the search mode.

The “working set” mentioned in Figure 3.4 is a collection of embeddings extracted during the current identification cycle, to be used for filtering. Filtering, both the necessity for it and the potential methods for going about it, are described in item 3.4.1. It is also necessary that the working set be cleared after identification.

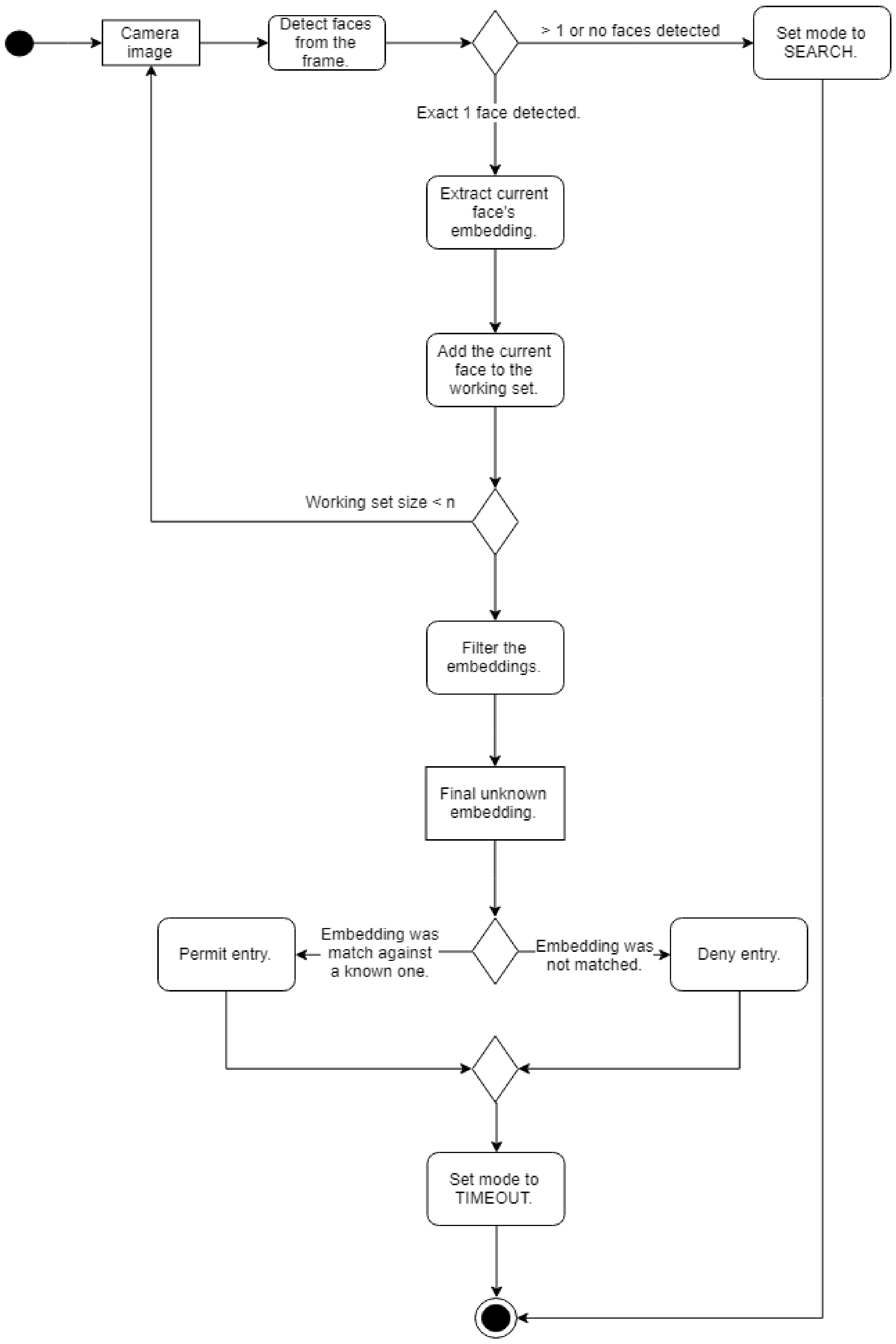


Figure 3.4. An outline of the program's identify mode.

3.4.1 Embedding filtering

Figure 3.4 outlines the creation of a filter that is to be applied to the embeddings. Filtering of the recovered embeddings can be used to minimize user annoyance while using the system. If the camera providing a feed to the program is at an odd angle, then it may well be that the first frame where a face is detected is not the best candidate for selection, due to the face being skewed on it. If the program was to run identification purely off of that one capture, then the system may well output an unknown identification and lock the user out temporarily.

In the implementation presented in chapter 4 of this work, the filter used is a simple averaging over three subsequent embeddings. Another idea to implement as a filter would be to remove the frame with the highest standard deviation from the rest, counting it as erroneous.

3.4.2 Embedding identification

The identification of the embeddings is done on the count of already found embeddings, according to the principles described in [2]. The basic process is to iterate over every known embedding, and to recover the first one where the Euclidian distance to the unknown embedding is below a selected threshold. The methodology itself is described in chapter 1.2.1 of this work. It is expected that the embeddings are held within a database, or in memory, and can be accessed on demand.

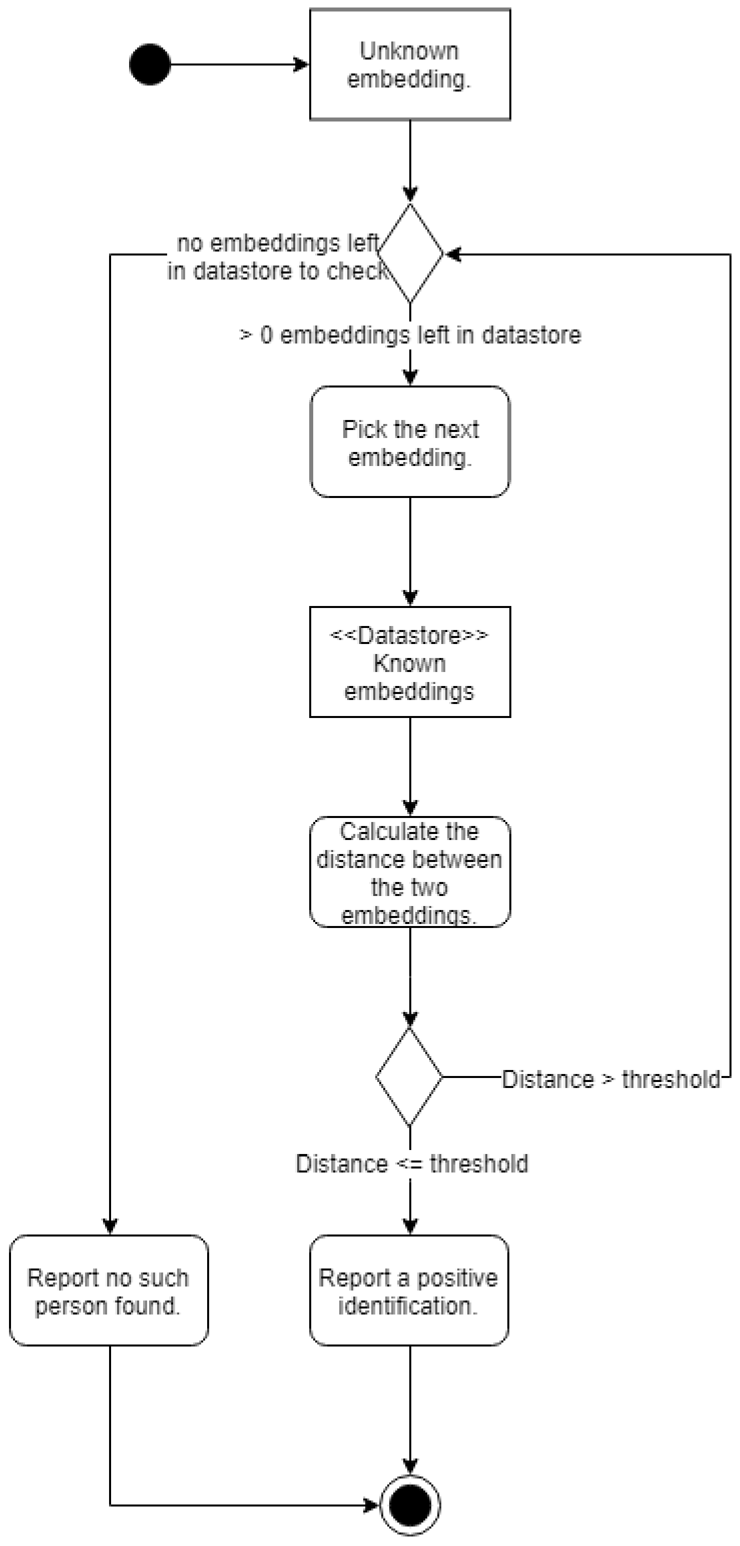

Unknown embedding.
no embeddings left in datastore to check
> 0 embeddings left in datastore
Pick the next embedding.
<<Datastore>> Known embeddings
Calculate the distance between the two embeddings.
Distance > threshold
Distance <= threshold
Report no such person found.
Report a positive identification.

Figure 3.5. The algorithm for matching an unknown embedding from the datastore of known embeddings.

It should be noted that the worst-case complexity of this type of search is $O(n)$, wherein n is equal to the number of embeddings within the database. This could, theoretically, bottle-neck the system when a large number of people are registered to within the system. However, due to the read-only nature of the operation, this operation is also highly scalable with the use of multiple threads.

3.5 System timeout mode

The timeout mode is the conclusion of identification. Its primary function is to implement application specific logic. For example, if this program is applied to control an electronic door lock, then this mode would unlock the door upon positive identification, and keep the door unlocked for a short time, until the user has been able to pass through the door. In the case of a negative identification, the system would temporarily lock out for the user to leave and another user to enter the frame. After the timeout mode, the program proceeds back into the search mode.

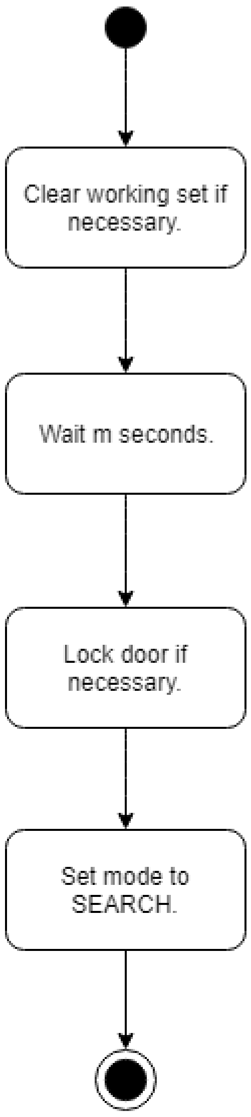


Figure 3.6. The system timeout procedure.

4. IMPLEMENTATION OF THE PROTOTYPE SYSTEM

The final part of this thesis is to write up an example implementation of the algorithms described in chapter 3, while also applying what has been learned in chapters 1 and 2. The outcome of this chapter will be the implementation of a system which corresponds to the requirements listed in point 3.1.

4.1 Libraries and development environment

The system architecture was implemented using the Python programming language [16], specifically Python version 3.7.3. The use of Python allows for relatively easy experimentation with various libraries and methodologies.

The OpenCV [17] library was used as the primary image processing and hardware camera handling. The library is relatively well known, and has Python bindings through the use of the opencv-python package. A library dependant on this, the Imutils library [18], helped further abstract the process of capturing frames from a generic camera device. OpenCV was also used to provide a simple GUI for the project.

NumPy [19] was used to provide helpers for mathematical operations. There is also a high degree of interoperability between NumPy and the various machine learning and neural network libraries such as dlib. Meaning that for experimentation, it is very easy to swap out the machine learning library used in favour of another one.

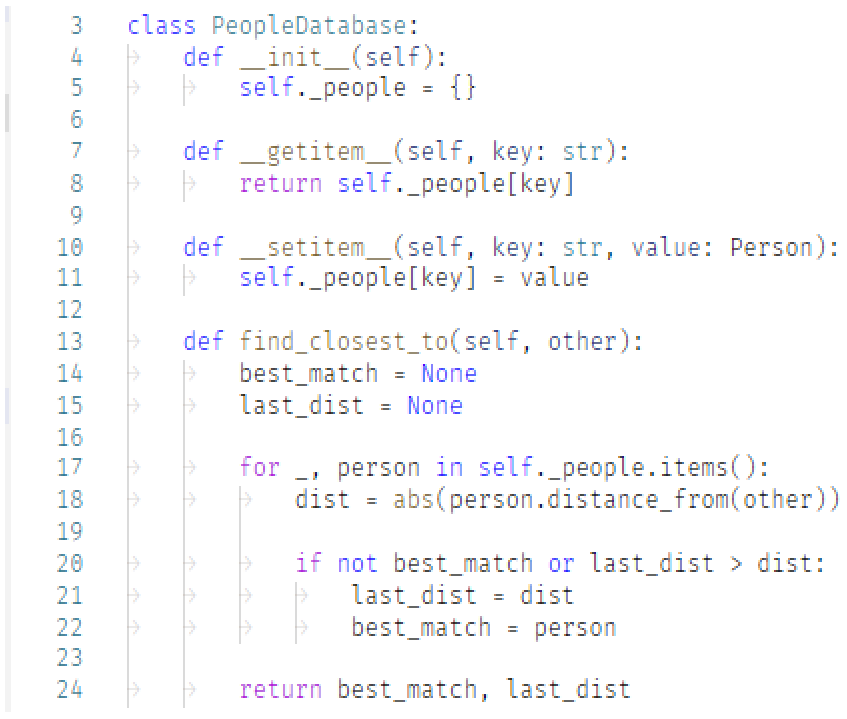
And as the key component of the system, to implement facial identification itself and the neural network operations required for it, the dlib library [20] was used. Much like is the case with OpenCV, dlib is originally a C++ library which has Python bindings, using the dlib package.

4.2 Implementation of the prototype system architecture

4.2.1 Storage of facial encodings

The fulfil the requirement of persistent data storage, the embeddings of the known faces were pre-calculated using the applied neural network and stored on disk. For the purposes of this system, the Python object serialization framework pickle [21] was used. This allows for the storage of a complete data object on disk, and the later recovery of it.

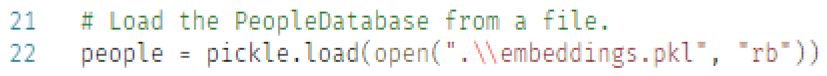
To represent an identity tied to an embedding, the Person class is used. It has simply two variables: the person’s name and their corresponding identity, which is precalculated. All identities known to the system are held in a PeopleDatabase class, which is the object serialized onto disk as previously described. A helper method, find_closest_to, is used to find the Person instance stored within a PeopleDatabase instance where the Euclidean distance between the unknown embedding is minimal. After that, a thresholding operation can be applied, as described in chapter 3.4.2.



```python
class PeopleDatabase:
    def __init__(self):
        self._people = {}

    def __getitem__(self, key: str):
        return self._people[key]

    def __setitem__(self, key: str, value: Person):
        self._people[key] = value

    def find_closest_to(self, other):
        best_match = None
        last_dist = None

        for _, person in self._people.items():
            dist = abs(person.distance_from(other))

            if not best_match or last_dist > dist:
                last_dist = dist
                best_match = person

        return best_match, last_dist
```

Figure 4.1. The PeopleDatabase class, along with the implementation of the find_closest_to method in Python.

The serialization and deserialization itself can be done with only a few lines of code, as shown below.



```python
# Load the PeopleDatabase from a file.
people = pickle.load(open(".\\embeddings.pkl", "rb"))
```

Figure 4.2. Loading of the PeopleDatabase instance from disk, using the pickle package.

It should be noted that there are some shortcomings to this method of storing data. And if this system is to be considered for deployment, then an alternate method for storing data should be considered. Due to the nature of the embedding within NumPy, as an array, it is likely possible to serialize it as a byte array for use with various database engines, such as MySQL.

4.2.2 Facial identification

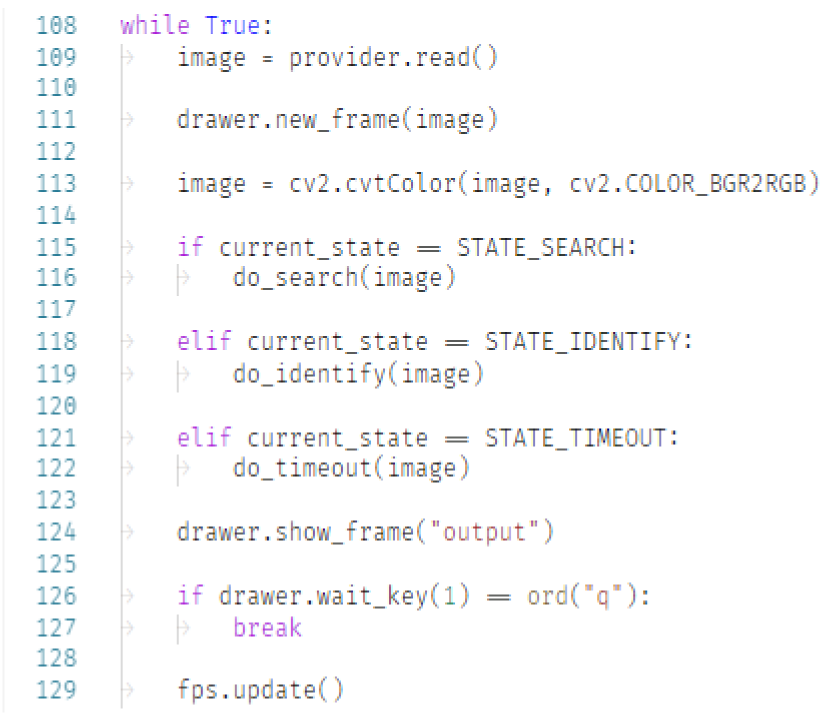
Based on the results of the experimentation done in chapter 2, the ResNet architecture and pipeline was selected for use within this implementation. The full pipeline can be seen in Figure 2.5. The library required to implement this is dlib.

Multiple helper classes were created to spread out and compartmentalize the various sub activities of the process of creating facial embeddings: the FaceChipper, FaceDettection, and FaceSummarizer classes. The latter class, while also generating the embeddings, handles the filtering of the embedding, as noted in point 3.4.1. The filtering is a very simple averaging operation done on the saved points using NumPy's average method.

Once the summarizer produces the averaged embedding during the identification cycle, it is matched with an entry from the PeopleDatabase described in the previous point, and a threshold operation is applied to the resultant distance.

4.2.3 The main cycle

The cycle is what implements the state machine outlined in point 3.2.1 of this work. It includes the camera image acquisition, the required pre-processing, and the post actions of the system, such as updating the GUI.



```python
while True:
    image = provider.read()

    drawer.new_frame(image)

    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    if current_state == STATE_SEARCH:
        do_search(image)

    elif current_state == STATE_IDENTIFY:
        do_identify(image)

    elif current_state == STATE_TIMEOUT:
        do_timeout(image)

    drawer.show_frame("output")

    if drawer.wait_key(1) == ord("q"):
        break

    fps.update()
```

Figure 4.3. The program's main loop as implemented in the Python programming language.

Notice the usage of OpenCV to convert between the BGR and RGB colour spaces, since the Imutils library retrieves a camera image in the former.

All the program's state functions are implemented independently, and their full implementation can be found in the appendices.

4.2.4 The graphical user interface

As noted in the beginning of this chapter, a minimal GUI was implemented to provide the user with feedback. The GUI is written using OpenCV, and plays back the currently processed frame, along with information about the current state that the program is in.

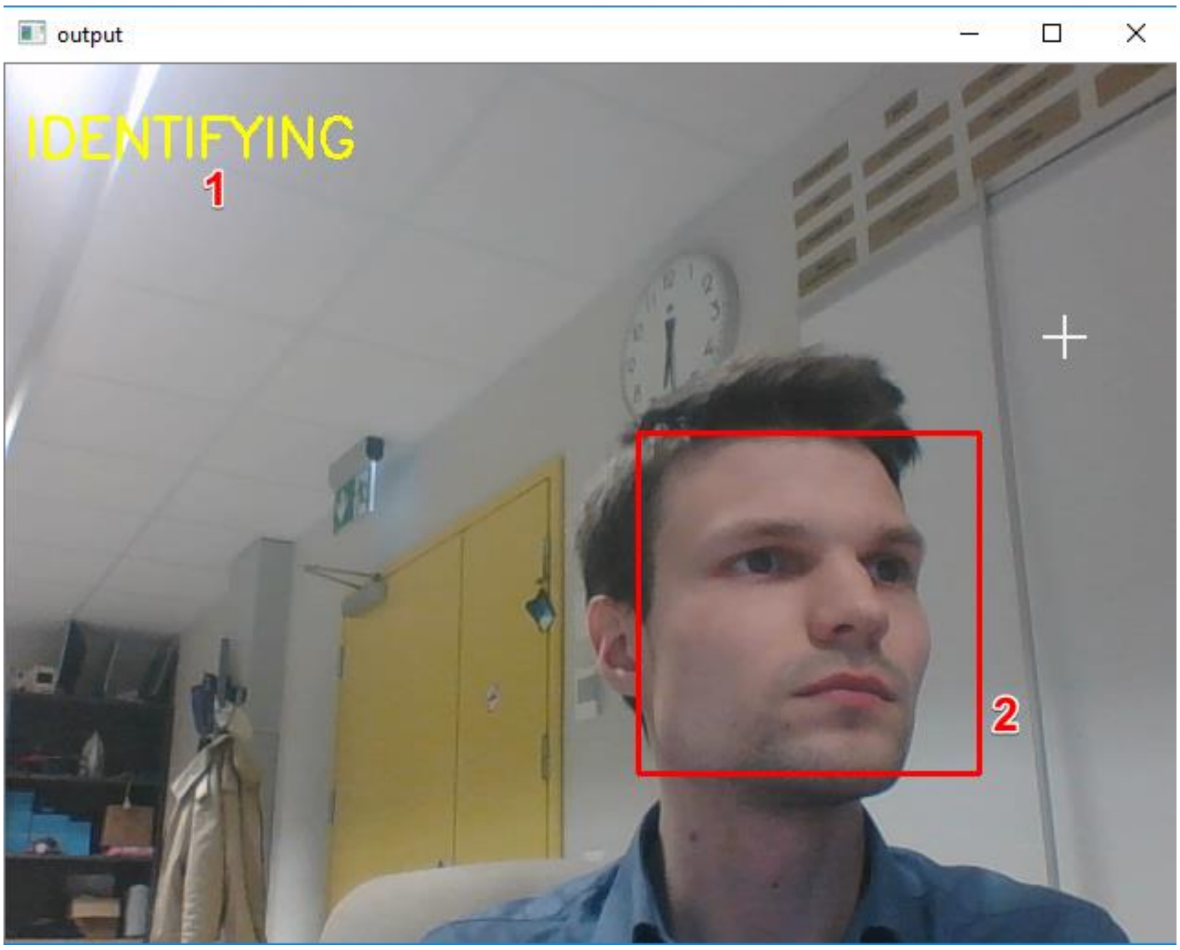


Figure 4.4. An overview of the minimalistic graphical user interface. 1: the current state of the program. 2: a rectangle to indicate a face identified from the current frame.

4.3 Pre-calculating embeddings

Due to the nature of the program, we need to pre-calculate all the facial embeddings before the system is used to identify. A parallel program for this was created, which used pictures of the subjects filed into a specified folder.

The pipeline used was like the one in the main program, with different inputs and outputs. The input is a series of pictures in a specific folder, with the folder named after the person whose identity we are saving. And the output is a compiled serialized Python object, a PeopleDatabase instance, which can then be used as an input for the primary program.

4.4 Testing of the prototype system

With the implementation completed, the system was tested on a laptop using the laptop's integrated camera. The system was provided a set of 5 faces and tested with colleagues to acquire examples of both successful positive and negative identifications.

As demonstrated in the following images, the system is able to recognize and assign a name to a previously recorded face (Figure 4.5), and to deny a face not present in the database (Figure 4.6). This fulfils the basic requirements outlined for it.

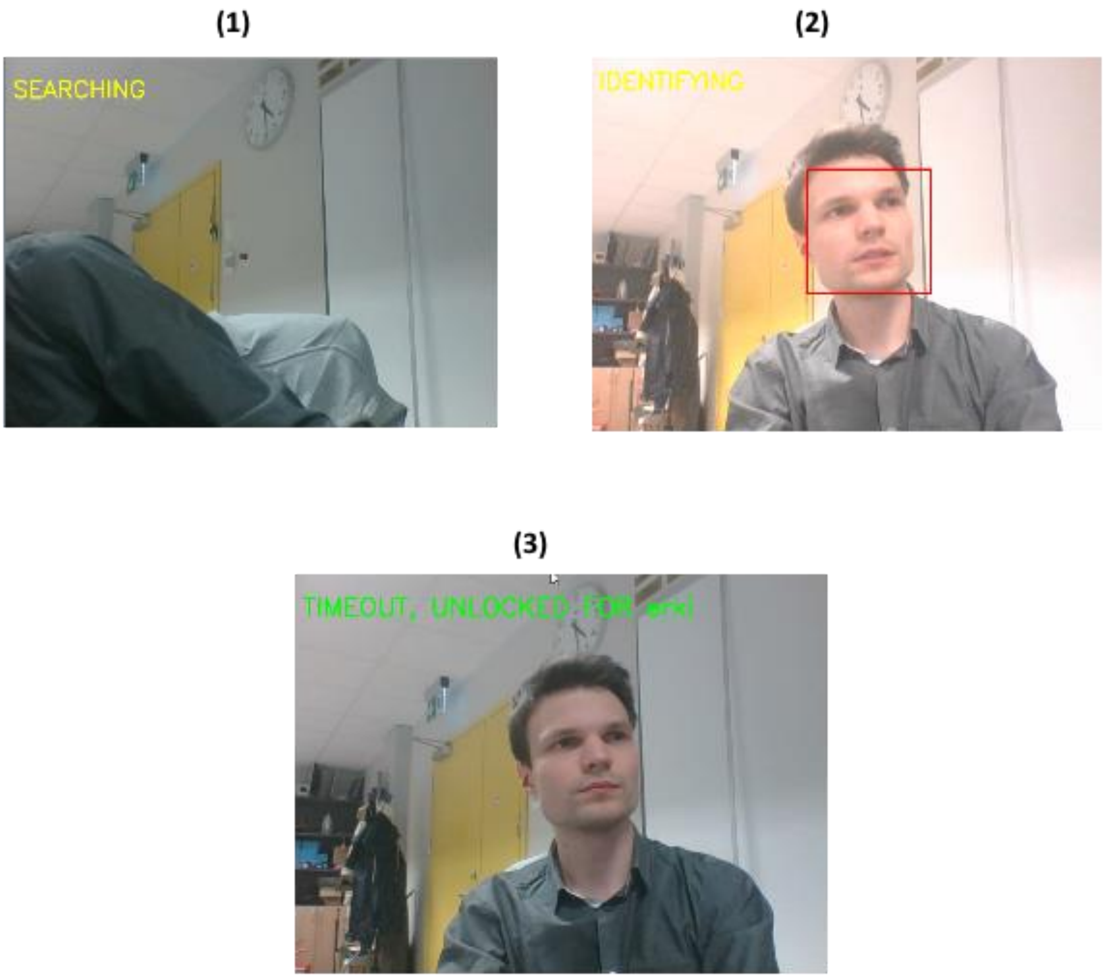


Figure 4.5. Output from the final prototype program. (1) displays the program in the searching state. (2) displays the program in the identification state, note the red rectangle around the face being identified. (3) displays the program after a positive identification, with the name associated with the face displayed in the top left of the screen.

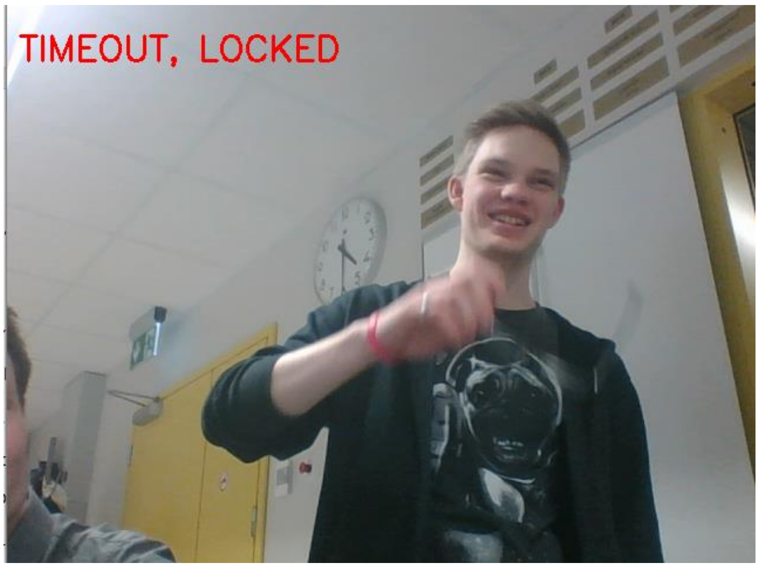


Figure 4.6. The prototype system recognizing an unknown individual and forbidding entry in the timeout state.

During testing, it was also observed that the system is able to identify a face from a relatively large distance away. Figure 4.7 demonstrates a positive identification retrieved from up to 3 meters away from the camera.

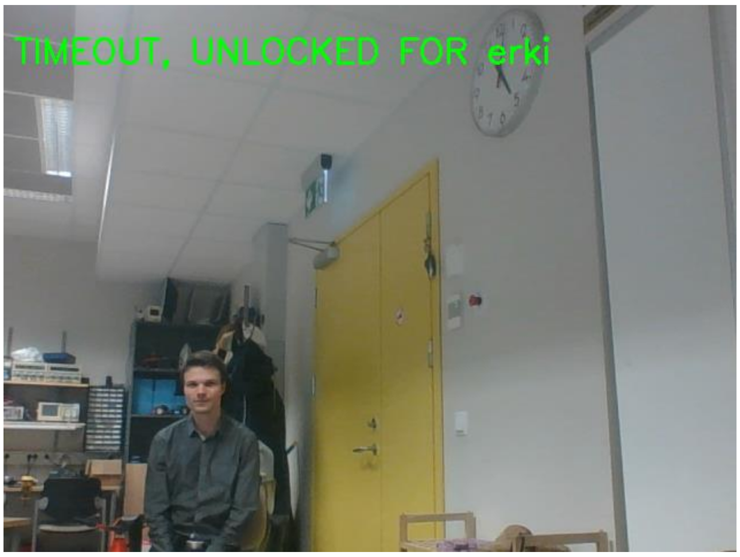


Figure 4.7. A positive identification of a face retrieved from roughly 3 meters away.

4.5 Identified shortcomings

The main shortcoming that was identified with the system was the fact that it could be tricked using a static image of a known face. For example, directing the camera providing the feed towards a monitor on which a known face was displayed was followed by the system positively identifying the known person from the displayed image. This would potentially allow for unsecured access to the are secured by this system.

A possible solution to this problem would be to include another method to confirm that there is indeed a human face present on camera feed. Infrared imaging, for example. Or a more general depth image, to counter the use of a display or a printed sheet of paper.

4.6 Prospects for future deployment and development

Since the Python programming language was used, this example system can be transferred over to the Raspberry Pi, or other ARM processors with a Python interpreter available. This would allow for the deployment of this system as an embedded security measure, to control physical access to any given area.

Figure 4.8 describes such a deployment, where a relatively small and non-complex security system based on the Raspberry Pi would be used to control a door locking mechanism which may, for example, provide entry to a secure area of an office. The central database could be used to synchronize the known and authorized embeddings, so that multiple such security systems could grant access to the same people.

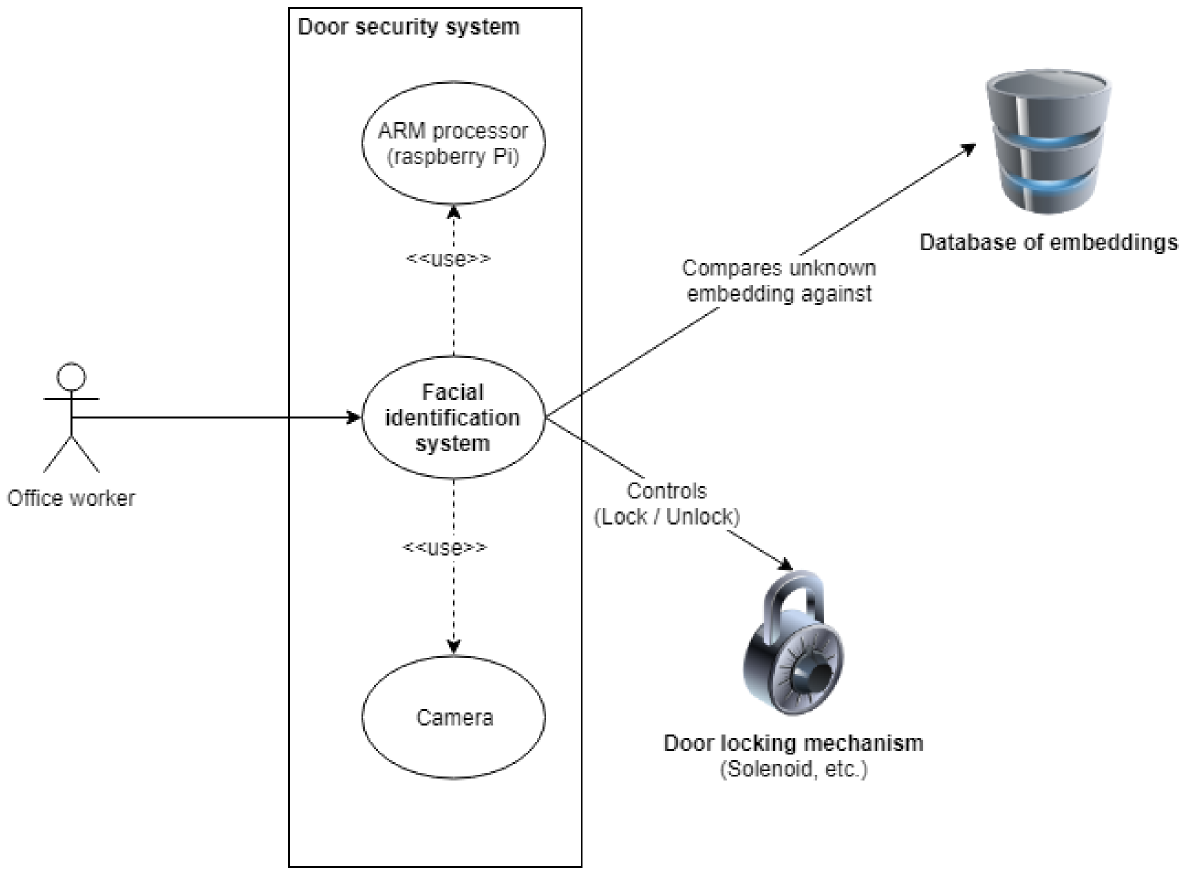

Door security system
ARM processor (raspberry Pi)
<<use>>
Facial identification system
<<use>>
Camera
Office worker
Compares unknown embedding against
Database of embeddings
Controls (Lock / Unlock)
Door locking mechanism (Solenoid, etc.)

Figure 4.8. A use case diagram describing a potential deployment of the system prototype.

Another scenario for deployment would be the adaptation of the architecture to allow for multiple input streams, with a centralized, more capable server handling the image processing and face identification.

KOKKUVÕTE

Lõputöös teostati taustauuring tänapäevastesse lahendustesse närvivõrkude rakendamisel näotuvastuse teostamiseks. Toodi välja olemasolev meetod nägude eristamiseks ja nende profiilide salvestamiseks kasutades närvivõrke ning ka kaks laialt levinud süvanärvivõrgu arhitektuuri antud metoodika rakendamiseks: konvolutsionaalsed närvivõrgud ning residuaalsed konvolutsionaalsed närvivõrgud.

Võrreldi kahte valmistehtud algoritmi, mis rakendasid eelnevalt mainitud metoodikaid: OpenFace närvivõrku ja ResNet närvivõrku. Võrdluse tulemusel järeldati, et kuigi OpenFace on oma töö poolest kiirem, siis selle täpsus ja töökindlus jääb alla ResNet arhitektuurile. Sellest tulenevalt valiti ResNet arhitektuur töö käigus koostatud prototüüpsüsteemi tuumaks.

Pärast põhikomponendi valikut teostati ülesande analüüs, määrati ära töö kaigus teostatava prototüüpsüsteemi nõuded ning koostati nõuetele vastav arhitektuur. Antud arhitektuur rakendab [2] kirjeldatud metoodikat nägude talletamiseks ja võrdlemiseks närvivõrkude abil. Meetod näeb ette näopildi teisendamise n-mõõtmelisse ruumi ning saadud punkti koordinaatide võrdlemist teadaolevate inimeste nägudest koostatud koordinaatidega. Sama inimese näod paiknevad tavaliselt antud n-mõõtmelises ruumis üksteisele lähedal.

Arhitektuuriga kirjeldatud süsteem ka realiseeriti. Tarkvara kirjutati kasutades Python programmeerimiskeelt ning on võimeline eelnevalt salvestatud nägusi ära tundma kaamerapildist. Kaamerapildist tuvastatud näo põhjal on programm võimeline tegema otsuse, kas tuvastatud isik on tema andmevaramus või ei.

Edasiarendusena on võimalik antud programmi rakendada ka riistvaralises lahenduses, kasutades näiteks Raspberry Pi platvormi. Samuti on võimalik täiustada andmevaramu kasutust, viies selle võrku nii, et mitme kaamera pilti saab ühe andmevaramu põhjal töödelda.

Puudujääkidena antud süsteemis jäid ResNet-i implementatsiooni töökiirus ning ka võimalus seda nö. petta. Esimese vajakajäägi likvideerimiseks oleks tarvis uurida lähemalt närvivõrkude kasutust ning optimeerimist, teise jaoks aga lisada näiteks mingi teine, kaamerapilti toetav inimtuvastussüsteem.

Üldiselt aga süsteem toimis ning seda annab rakendada sissejuhatuses mainitud targa turvasüsteemi edasiseks arendamiseks.

SUMMARY

The thesis provided an overview of the modern approaches that are applied to facial identification. We presented existing methods for facial identification by method of feature extraction and mapping them to embeddings via the use of neural networks. We also identified and provided an overview of two prominent neural network architecture for conducting this embedding creation: deep convolutional neural networks, and residual neural networks.

With these methods and algorithms identified, we conducted a comparison of two available implementations of the algorithms: OpenFace, a CNN, and ResNet, a residual neural network. The comparison found that, though OpenFace is a lot faster than ResNet, ResNet was able to provide a lot more reliability, and key of all, accuracy. As such, the ResNet based pipeline examined as used as the core component of our prototype system.

Following the selection of the core pipeline, the rest of the system was detailed. Starting with the establishment of system requirements. With the requirements in place, the system's architecture was detailed around it. The mapping of a face to a facial embedding, a point in an n-dimensional space, as described in [2], was used as the main device for conducting for facial identification. The ResNet neural network was used to produce said embeddings from a camera feed, which could then be compared against known embeddings for identification.

With the architecture defined, the system was implemented. The implementation was completed using the Python programming language, with common libraries used for image processing and neural network application: OpenCV, NumPy, dlib. The implementation successfully used an on-disk datastore to hold a selection of known faces and was able to identify known faces from a live camera image.

At the conclusion of the work, the system that was implemented could easily be integrated into a more complex "smart" security system. Due to the libraries used, the implementation can also easily be applied to a capable ARM processor, such as the ones found on a Raspberry Pi.

The main shortcoming of the system was the ability to cheat the system via the use of a previously capture image. Already realized solutions exist for this, as can be observed from the mobile market with facial identification on certain mobile phones.

LIST OF REFERENCES

[1] R. Chellappa, C. L. Wilson ja S. Sirohey, „Human and Machine Recognition of Faces: A Survey," *Proceedings of the IEEE,* kd. 83, nr 5, pp. 705-740, 1995.

[2] D. K. J. P. Florian Schroff, „FaceNet: A Unified Embedding for Face Recognition and Clustering," IEEE Xplore, 2015.

[3] S. Lawrence, C. L. Giles, A. C. Tsoi ja A. D. Back, „Face Recognition: A Convolutional," *IEEE TRANSACTIONS ON NEURAL NETWORKS,* kd. 8, nr 1, pp. 98-113, 1997.

[4] M. Z. B. C. D. K. W. W. T. W. M. A. H. A. Andrew G. Howard, „MobileNets: Efficient Convolutional Neural Networks for Mobile Vision," 2017.

[5] A. V. A. Z. Omkar M. Parkhi, „Deep Face Recognition," %1 *British Machine Vision Conference*, Oxford, 2015.

[6] J. R. B. B. A. D. G. G. A. J. O. P. Jonathon Phillips, „An Introduction to the Good, the Bad, & the Ugly Face Recognition," *IEEE Int. Conf. on Automatic Face & Gesture Recognition and Workshops,* pp. 346-353, 2011.

[7] Y. R. C. L. X. T. C. C. L. Kaidi Cao, „Pose-Robust Face Recognition via Deep Residual Equivariant Mapping," %1 *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, New York, 2018.

[8] X. Z. S. R. J. S. Kaiming He, „Deep Residual Learning for Image Recognition," %1 *IEEE Conference on Computer Vision and Pattern Recognition*, New York, 2016.

[9] R. F. P. P. Li Fei-Fei, „A Bayesian Approach to Unsupervised One-Shot Learning of Object Categories," %1 *9th IEEE International Conference on Computer Vision*, Nice, 2003.

[10] A. Rosebrock, „OpenCV Face Recognition," 24 september 2018. [Online]. Available: https://www.pyimagesearch.com/2018/09/24/opencv-face-recognition/. [Accessed 1 May 2019].

[11] D. E. K. e. al., „Github.com - dlib - face recognition example," 26 veebruar 2019. [Online]. Available: https://github.com/davisking/dlib/blob/master/python_examples/face_recognition.py. [Accessed 4 May 2019].

[12] B. Amos, „OpenFace,“ 19 jaanuar 2016. [Online]. Available: https://cmusatyalab.github.io/openface/. [Accessed 5 May 2019].

[13] B. Amos, „Models and Accuracies - OpenFace,“ 9 august 2016. [Online]. Available: https://cmusatyalab.github.io/openface/models-and-accuracies/. [Accessed 5 May 2019].

[14] A. E. King, „dlib.net/face_detector.py.html,“ [Online]. Available: http://dlib.net/face_detector.py.html. [Accessed 5 May 2019].

[15] J. S. Vahid Kazemi, „One Millisecond Face Alignment with an Ensemble of Regression Trees,“ %1 *IEEE Conference on Computer Vision and Pattern Recognition*, Columbus, 2014.

[16] Python Software Foundation, „Welcome to Python.org,“ 2019. [Online]. Available: https://www.python.org/. [Accessed 12 May 2019].

[17] OpenCV team, „OpenCV,“ 2019. [Online]. Available: https://opencv.org/. [Accessed 12 May 1029].

[18] A. Rosebrock, „Imutils on Github,“ 2019. [Online]. Available: https://github.com/jrosebr1/imutils. [Accessed 12 May 2019].

[19] NumPy developers, „NumPy - NumPy,“ 2019. [Online]. Available: https://www.numpy.org/. [Accessed 12 May 2019].

[20] D. E. King, „dlib C++ Library,“ 10 märts 2019. [Online]. Available: http://dlib.net/. [Accessed 12 May 2019].

[21] Python Software Foundation, „pickle -- Python object serialization -- Python 3.7.3 documentation,“ 12 mai 2019. [Online]. Available: https://docs.python.org/3/library/pickle.html. [Accessed 12 May 2019].

APPENDICES

Appendix 1 Main program code in the Python programming language

```
import time
import imutils.video
import cv2
import pickle

from face_scanner.face_chipper import FaceChipper
from face_scanner.face_detector import FaceDetector
from face_scanner.face_summarizer import FaceSummarizer
from face_scanner.frame_drawer import FrameDrawer
from face_scanner.person_holder import *

# Initialize the video streams.
provider = imutils.video.VideoStream()
provider.start()

#
detector = FaceDetector()
chipper = FaceChipper("./shape_predictor_5_face_landmarks.dat")
descriptor = FaceSummarizer("./dlib_face_recognition_resnet_model_v1.dat")

# Load the PeopleDatabase from a file.
people = pickle.load(open(".\\embeddings.pkl", "rb"))

drawer = FrameDrawer()

fps = imutils.video.FPS()
fps.start()

STATE_SEARCH = 0
STATE_IDENTIFY = 1
STATE_TIMEOUT = 2

TEXT_CORNER = (10, 50)

current_state = STATE_SEARCH
timeout_started = None
person_unlocked_for = None

def do_search(frame):
  global current_state, drawer, chipper, descriptor, detector

  detections = detector.get_detections(image)

  drawer.add_text("SEARCHING", TEXT_CORNER, drawer.COLOR_YELLOW)
  for _, detection in enumerate(detections):
    drawer.add_rectangle(detection, drawer.COLOR_YELLOW)

  if len(detections) == 1:
    current_state = STATE_IDENTIFY

def do_identify(frame):
  global current_state, drawer, chipper, descriptor, detector

  detections = detector.get_detections(image)

  for _, detection in enumerate(detections):
    drawer.add_rectangle(detection, drawer.COLOR_YELLOW)

  drawer.add_text("IDENTIFYING", TEXT_CORNER, drawer.COLOR_YELLOW)

  if len(detections) != 1:
    current_state = STATE_SEARCH

    descriptor.reset_embeddings()
  else:
    detections = detector.get_detections(image)
    detection = detections[0]

    drawer.add_rectangle(detection, drawer.COLOR_RED)

    chip = chipper.get_face_chip(image, detection)

    descriptor.add_embedding(chip)

    if len(descriptor.embeddings) == 3:
      current_state = STATE_TIMEOUT

def do_timeout(frame):
  global current_state, drawer, chipper, descriptor, detector, person_unlocked_for, people,
timeout_started

  if timeout_started is None:
    averaged_embedding = descriptor.embedding_avg_position()

    person, dist = people.find_closest_to(averaged_embedding)

    timeout_started = time.time()

    if dist < 0.5:
      drawer.add_text(f"UNLOCKED; WELCOME {person.name}", TEXT_CORNER,
drawer.COLOR_GREEN)
      person_unlocked_for = person.name
      return
```

```
        else:
            drawer.add_text(f"LOCKED; NO MATCH", TEXT_CORNER, drawer.COLOR_RED)
            return

    if time.time() - timeout_started > 4:
        person_unlocked_for = None
        descriptor.reset_embeddings()
        timeout_started = None

        current_state = STATE_SEARCH
    elif person_unlocked_for:
        drawer.add_text(f"TIMEOUT, UNLOCKED FOR {person_unlocked_for}", TEXT_CORNER,
drawer.COLOR_GREEN)
    else:
        drawer.add_text("TIMEOUT, LOCKED", TEXT_CORNER, drawer.COLOR_RED)

while True:
    image = provider.read()

    drawer.new_frame(image)

    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    if current_state == STATE_SEARCH:
        do_search(image)

    elif current_state == STATE_IDENTIFY:
        do_identify(image)

    elif current_state == STATE_TIMEOUT:
        do_timeout(image)

    drawer.show_frame("output")

    if drawer.wait_key(1) == ord("q"):
        break

    fps.update()

fps.stop()
print("[INFO] elasped time: {:.2f}".format(fps.elapsed()))
print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))

cv2.destroyAllWindows()
provider.stop()
```

Appendix 2 The Person class code used by the main program

```
import numpy as np
import dataclasses

@dataclasses.dataclass
class Person:
    name: str
    position: np.array

    def distance_from(self, other):
        return np.linalg.norm(self.position - other)

class PersonConstructor:
    def __init__(self, name: str):
        self.name = name
        self.embeddings = None

    def append_embedding(self, embedding: np.array):
        if self.embeddings is None:
            self.embeddings = embedding
        else:
            self.embeddings = np.vstack((self.embeddings, embedding))

    def std_dev(self):
        return np.std(self.embeddings, axis=0)

    def average_position(self):
        return np.average(self.embeddings, axis=0)

    def to_person(self):
        return Person(self.name, self.average_position())
```

Appendix 3 The PeopleDatabase class code used by the main program

```
from .person import Person, PersonConstructor

class PeopleDatabase:
  def __init__(self):
    self._people = {}

  def __getitem__(self, key: str):
    return self._people[key]

  def __setitem__(self, key: str, value: Person):
    self._people[key] = value

  def find_closest_to(self, other):
    best_match = None
    last_dist = None

    for _, person in self._people.items():
      dist = abs(person.distance_from(other))

      if not best_match or last_dist > dist:
        last_dist = dist
        best_match = person

    return best_match, last_dist

class PeopleConstructor:
  def __init__(self):
    self._people = {}

  def __getitem__(self, key: str):
    if key not in self._people:
      self._people[key] = PersonConstructor(key)

    return self._people[key]

  def for_all(self):
    for key, value in self._people.items():
      yield key, value

  def to_people_database(self):
    people = PeopleDatabase()

    for name, constructor in self._people.items():
      person = constructor.to_person()
      people[name] = person

    return people
```

Appendix 3 The FaceDetector class used in the main program

```
import dlib

class FaceDetector:
  def __init__(self):
    self._detector = dlib.get_frontal_face_detector()

  def get_detections(self, image):
    """
    Takes an image as input. Returns a set of frontal facial detections.
    """
    return self._detector(image, 1)
```

Appendix 4 The FaceChipper class used by the main program

```
import dlib

class FaceChipper:
    def __init__(self, face_predictor_path):
        self._predictor_path = face_predictor_path
        self._face_predictor = dlib.shape_predictor(self._predictor_path)

    def get_face_chip(self, image, detection):
        """
        Takes an image and a detection, cuts out the detection from it, and returns
        the appropriate face chip.
        """

        shape = self._face_predictor(image, detection)
        return dlib.get_face_chip(image, shape)
```

Appendix 5 The FaceSummarizer class used by the main program

```python
import numpy as np
import dlib

class FaceSummarizer:
  def __init__(self, face_model_path):
    self._model_path = face_model_path
    self._recognizer = dlib.face_recognition_model_v1(face_model_path)

    self.embeddings = None

  def reset_embeddings(self):
    """
    Resets the current HOG embedding array.
    """
    self.embeddings = None

  def add_embedding(self, chip):
    """
    Takes a face chip, turns it into a 256d HOG embedding, and stores it in the
    current array.

    Returns the newly acquired descriptor.
    """
    descriptor = self._recognizer.compute_face_descriptor(chip)

    if self.embeddings is None:
      self.embeddings = descriptor
    else:
      self.embeddings = np.vstack((self.embeddings, descriptor))

    return descriptor

  def embedding_std_dev(self):
    return np.std(self.embeddings, axis=0)

  def embedding_avg_position(self):
    return np.average(self.embeddings, axis=0)
```

Appendix 6 The FrameDrawer class used by the main program

```
import cv2

class FrameDrawer:
    COLOR_GREEN = (0, 255, 0)
    COLOR_RED = (0, 0, 255)
    COLOR_BLUE = (255, 0, 0)
    COLOR_YELLOW = (0, 255, 255)

    def __init__(self):
        self._current_frame = None

    def new_frame(self, frame):
        self._current_frame = frame

    def add_rectangle(self, rectangle, color):
        """
        Adds a rectangle to be drawn.
        """
        if self._current_frame is None:
            return

        start_x = rectangle.tl_corner().x
        start_y = rectangle.tl_corner().y
        end_x = rectangle.br_corner().x
        end_y = rectangle.br_corner().y

        cv2.rectangle(self._current_frame, (start_x, start_y), (end_x, end_y), color, 2)

    def add_text(self, string, coordinates, color):
        if self._current_frame is None:
            return

        cv2.putText(self._current_frame, string, coordinates, cv2.FONT_HERSHEY_SIMPLEX, 1, color,
2)

    def show_frame(self, frame_name):
        cv2.imshow(frame_name, self._current_frame)

    def wait_key(self, timeout):
        return cv2.waitKey(timeout) & 0xFF
```