



CONTINUOUS FACE DETECTION AND VERIFICATION DURING ONLINE EXAMINATIONS

Dominik Kurasbediani

01146313@pw.edu.pl

ABSTRACT

I present a prototype of a system used to continuously detect and verify faces of examinees during online examinations, as well as a method of representation of the results of such verifications, and overall architecture of the system. The system consists of three modules: the client-side face detection, the server-side face verification, and the exam supervisor's dashboard. Face detection occurs on each client's computer using *Single Shot Multibox Detection* (SSD). The resulting image is aligned and cropped, then encoded into base-64 format and sent to the verification server via a POST request. Upon receiving such request, the server decodes the base-64 image, feeds it to the *VGG-Face* model to get a vector representation of the face and compares the Euclidean distance between said image and a predefined base image. The results of the last operation are sent to the exam supervisor's dashboard via a POST request. The exam supervisor's dashboard's task is to receive and process such requests and represent the updated data in real time. In addition, the exam supervisor's dashboard is responsible for configuration of the exam session, as well as generation of the client-side executables to allow easy and seamless process for the students.

KEYWORDS: Real-Time Face Recognition, Client-Server Architecture, Online Examination.

STATEMENT ABOUT THE AUTHORSHIP OF THE WORK SIGNED BY THE STUDENT

TABLE OF CONTENTS

ABSTRACT	1
Abstract in Polish	3
Statement about the authorship of the work signed by the student	4
Chapter 1. Introduction	8
Motivation	8
Deficiencies of online examinations	8
Existing Procedures	8
Proposed Procedure	9
Chapter 2. System Architecture	10
2.2 Exam Supervisor's Dashboard	11
Frontend	11
Backend	14
2.2 Client-Side Face Detection	16
ID Form	16
Capture Window	16
2.3 Verification Server	19
File Handling	19
Representation	19
Verification	20
Chapter 3. Detection Solution Study & Analysis	24
Speed	24
Accuracy	25
ROC	25
AUC	26
Results	26
Resources	27
Analysis	27
Conclusion	28
Chapter 3. VGG-Face	29
Network Architecture and Training	29
Learning Face Classifier	29

Triplet Loss.....	30
Architecture	30
Training	31
Evaluation and Results	31
Chapter 4. Single Shot Multibox Detector (SSD)	33
Architecture	33
Training	34
References	36
List of Tables	36
List of figures	38
List of appendices.....	39
Appendices	40
Appendix A. Large Tables	40

CHAPTER 1. INTRODUCTION

MOTIVATION

During the COVID-19 pandemic, there were no alternatives to online education and online examinations. The lockdown prevented people from leaving their houses, so students were forced into the system that was not yet ready. Now that the lockdown is over, many students, international students in particular, have made commitments preventing them from attending in-person classes. While some universities still offer online classes, the online examination protocols are flawed and underdeveloped. The current solutions for online examination are variants of the following approach: the students connect to a third party-provided online meeting (e.g., Zoom, Microsoft Teams), turn on their cameras, proceed to complete the exam. This method is flawed in terms of security against academic offenses.

DEFICIENCIES OF ONLINE EXAMINATIONS

The most prevalent downside of existing protocols, in my experience as a student, is the abuse of the fact that the exam supervisor has never seen students' faces before the exam due to it not being required during the semester, as well as the absence of identity verification as a prerequisite to writing the exam. In other words, students may outsource (ask their friends to write the exam for them, pay a professional to do it, or other) and get a passing grade regardless of whether they truly know the material presented during the semester. Hence, the students were incentivized to abuse this system, commit academic offenses, and get a passing grade without opening the course textbook during the semester. This effectively lowers the overall education level, potentially creating severe errors, which may be lethal in some cases, due to students-turned-professionals being unqualified. One may argue that the proportion of students cheating is marginal at best, however this problem is a hole in a ship that might lead to it sinking.

With an existing state-of-the-art face recognition technology, it is possible to prevent these problems by building and integrating security applications into existing infrastructures of academic institutions.

EXISTING PROCEDURES

Normally, examinations are held in-person, which prevents students from committing academic offenses under the watchful eyes of the exam supervisors and their assistance. The identity of examinees is established and verified either on entrance to the exam room or during the exam itself: the examinees are asked to present a student identification card that contains the image of their face, first and last name(s), and the student ID number. This information is verified against a list of students registered for this exam, and the images in the identification card is matched with the face of the owner of the card. If these conditions are met, the examinee is cleared to write the exam. Examinees are not allowed to leave the exam room until they have turned in their exam papers, or their identity is verified once again on re-entry.

The described above procedure is put in place with security in mind, to prevent the problem mentioned in the previous section. It has been successful in preventing most cases of identity fraud in the context of examinations. However, currently there is no such procedure being used for online examinations.

PROPOSED PROCEDURE

The procedure I use in my application is modeled after the existing in-person exam protocol:

1. Before the start of the exam session, exam supervisors are required to create a list of students allowed to take the exam (first and last names, student numbers, optional base image).
2. The client executables (access point for the students) are generated and distributed to the students.
3. The exam supervisor starts the exam session.
4. The students launch the executables, input their identification data (student number)
5. The system verifies their faces throughout the duration of the whole exam.

The described above procedure replicates the in-person procedure except for the last step. Step 6 covers an edge case that is only possible during online examinations: it is not possible to fake one's identity once it has been verified without leaving the examination room first. However, during online examinations it is possible. By ensuring continuous verification, the application prevents it from happening.

Any face recognition pipeline consists of at least four parts: face detection, alignment, face representation, and verification. The system consists of three modules: client-side face detection, server-side face verification, and exam supervisor's dashboard. The pipeline stages are spread across the three modules to optimally distribute the stress and utilization of computational resources required of each stage. Figure 1 shows the spread of the face recognition pipeline:

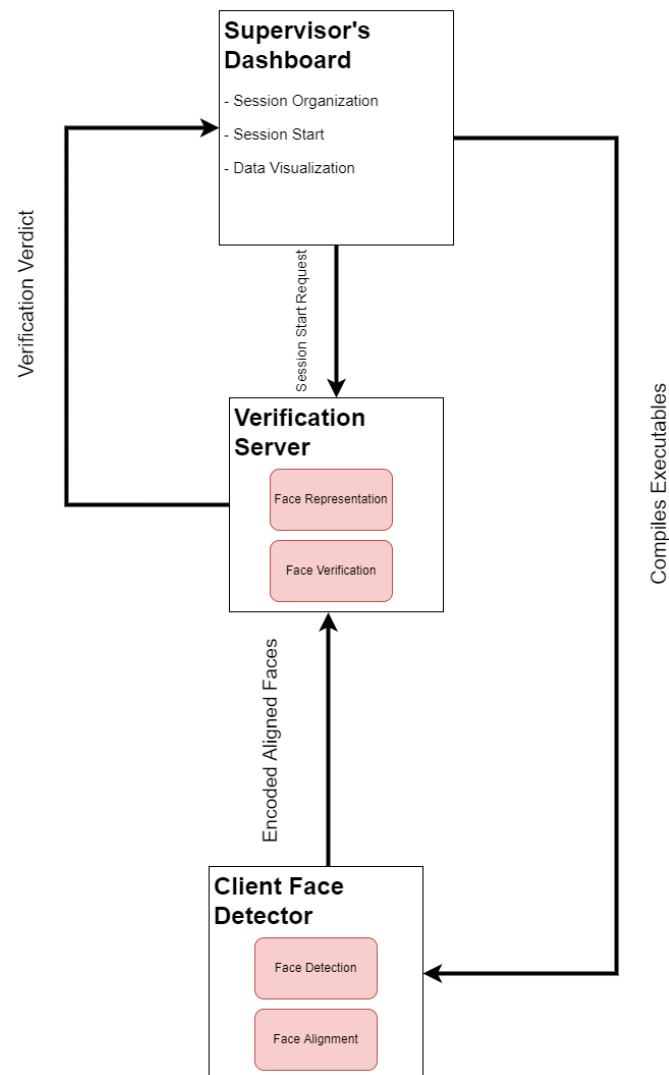


Figure 1. Request flow diagram and the spread of the face recognition pipeline. The red rectangles are the stages of the face recognition pipeline.

2.2 EXAM SUPERVISOR'S DASHBOARD

The Dashboard module is responsible for configuration of the session, executing compilation scripts, starting the verification server, and presentation of the verification results. It is a web application that the examination supervisor exclusively will have access to. The module consists of two components: the frontend and the backend.

FRONTEND

This component is responsible for the visual representation of the data and allowing the supervisor to utilize all the tools provided efficiently and with ease. The User Interface (UI) is designed to be as minimalistic and intuitive as possible to minimize the training required to become acclimated to the system. It is built with React JS, which is a JavaScript framework. Due to the limitation imposed by the `USEEFFECT` hook in React JS, it is impossible to use it to automatically refresh the contents of the page whenever there is a change in data. Instead, the contents are refreshed with a set frequency. This results in a consistent amount of GET requests to the backend, which is more stable when there is a large number of clients connected to the session.

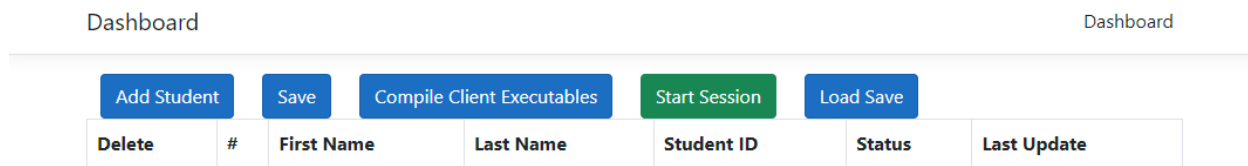


Figure 2. Default UI of the Dashboard

ADD STUDENT

This button redirects the user to a separate page that contains a form that allows for addition of new students to the database. The required components are the first and last names, and the student identifier. There is also an optional field to add a base image of the student. This base image is uploaded to the backend server and saved locally for the verification purposes. The 'Cancel' button redirects the user back to the previous page.

Dashboard

First Name

Last Name

Student ID

Base Image

Figure 3. Add Student Form

When the form is submitted, the contents are sent to the backend via a POST request, where it is processed further. After this step, the student is added to the table on the previous page and to the database.

Dashboard

Dashboard

<input type="button" value="Add Student"/>	<input type="button" value="Save"/>	<input type="button" value="Compile Client Executables"/>	<input type="button" value="Start Session"/>	<input type="button" value="Load Save"/>		
Delete	#	First Name	Last Name	Student ID	Status	Last Update
<input type="button" value="Delete"/>	1	Dominik	Kurasbediani	302155	false	0001-01-01T00:00:00

Figure 4. Table after a new examinee has been added.

The ‘Delete’ button deletes the given student from the database.

SAVE

The ‘Save’ button saves the current state of the database, serializes it into .json format and the supervisor can download it to their computer.

COMPILE CLIENT EXECUTABLES

This button starts the compilation of the executables used by clients (examinees) to send the detected faces to the verification server. In case the supervisor does not have the executables, or the configuration of the component, or the component itself, has changed between sessions, it is required to compile new client executables. In the prototype application, this process can take up to approximately 2 hours to finish, and once started the supervisor is asked not to close or refresh the page until the process is finished.

START SESSION

This button starts the verification server effectively starting the session. Shortly after this step, the supervisor can alert the examinees that the session has begun and ask them to run the distributed executables.

LOAD SAVE

The 'Load Save' is the counterpart to the 'Save' button. The supervisor can use this button to upload a previously saved session instead of starting a new one.

BACKEND

This component is responsible for all the data management, script execution and operation of the dashboard. It is a REST API built with C# and ASP.NET Core.

DATABASE

The module is implemented in a way that omits configuration and maintenance of dedicated databases and instead utilizes an in-memory database built in runtime. This solution reduces the cost of maintenance in the long-term at the cost of building the database every session (which is compensated by the ability of the user to export the session data and load sessions from a .json file). The solution also decreases the amount of time required for the changes to occur due to the lack of the need to send and receive data from a dedicated database. However, it is configured in a way that would not take a significant amount of development time to implement support of a dedicated solution due to utilization of .NET's Entity Framework.

ADD STUDENT

Entity Framework and ASP.NET provide a simple and well-documented framework for database operations such as adding, editing, and deleting new entries (POST, PUT and DELETE requests, respectively.) The form sent along with the POST request contains serialized text information regarding the name, surname, and student identifier. The latter is the primary key.

The form may optionally contain an image. If there is an image attached, the server will rename the image to the student identifier the image belongs to postfixed with the extension of the image file and save it locally. The location of the saved image is also recorded in the database to avoid iterating through the directory to delete the image when deleting a student, as well as to supplement session save-load system.

SAVING

When the server receives a request to save the current session, it serializes the list of students currently in the database into a .json file (name, surname, student identifier, status, last status update and path to the base image). The file is saved locally to the dashboard's host machine to provide an additional fail-safe to user errors. Then the resulting file is read as an array of bytes. After that, the server returns a file with the specified serialized session (Status200OK), the Content-Type of 'application/octet-stream', and 'SaveData.json' as the name of the file. This file is then sent to the user and is downloaded.

LOADING

Upon receiving a given POST request to load the session, the server assumes an attached session data file previously generated with the 'Save' button. The file is then saved locally on the server's host machine to allow for diagnostics in case an error occurs. The contents of the file are then deserialized and the list of examinees is added to the existing database.

All the steps are performed inside try-catch blocks to avoid fatal errors due to dealing with file systems. In addition to the assumption stated above, there are null checks at the deserialization step to handle the case where the submitted .json file would contain serialization errors. If the file submitted is empty, the user will receive a Status-400 Bad Request response.

Upon successfully adding all the examinees to the database, the server returns a Status-200 OK response.

COMPILATION SCRIPT EXECUTION

To maintain modularity and customizability of the system, the Dashboard relies on Windows Shell or Bash (depending on the host OS) to execute the compilation process. The API itself only starts said scripts. For the prototype, the script first attempts to install all the Python dependencies required by the client. It then attempts to install PyIntsaller – a Python package that allows to bundle the client application and all its dependencies into a single package. The packages that are already present are skipped to avoid redundancies. Once dependency installation has been concluded, the client application is compiled using PyInstaller. The compilation time depends on the technical specifications of the machine it is executed on. However, the first time this process is executed on a machine, it takes a significant amount of time to finish. After the process is finished, PyInstaller does not remove the intermediate files, so further compilation times will take significantly less time. Once the compilation has finished, the resulting files are zipped and sent to the requester.

START SESSION

The system design expects the verification server and the dashboard to be launched from the same machine. The main reason is session directory preparation. The verification server verifies the incoming faces against their respective base images that are prepared in this step.

Once the dashboard's server receives a request to start a session, before running any scripts, it starts preparing the directories and file system hierarchy expected by the verification server. First, it deletes the remainder directories from the previous session. Then, it creates a directory for each examinee with the student identifier as the name.

Like the 'Compilation Script Execution' section, this procedure uses custom Windows Shell or Bash scripts to allow for customizability and expandability. In the prototype, the script iterates through the base images uploaded through the 'Add Student' step, puts them into their respective directory and renames them to 'base' followed by the extension of the file. Then the script installs the Python dependencies required by the verification server and starts the server.

2.2 Client-Side Face Detection

Once the compilation of client executables is done, the supervisor may distribute them to the examinees. Each examinee gets one executable file for their operating system (.exe for Windows, .app for Mac OS).¹ They will launch this executable right before attempting the exam. The module is responsible for detecting the faces of each examinee. It also crops and aligns the face to the size required and sends the image to the verification server, where it is processed further. The module is written in Python. The Module is divided into two components: the ID form and the capture window.

ID Form

This component is responsible for the presentation of the form the examinees must complete for the detection process to begin. It consists of only one field – the student identifier.

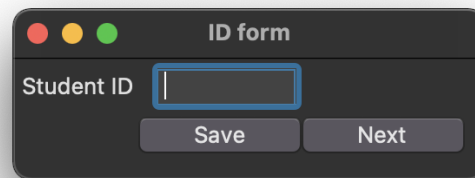


Figure 5. Client Form

Once completed, the user presses the corresponding buttons, and the identifier entered is saved and is sent with every payload for the verification server to identify the sender. Once the ‘Next’ button is pressed, the form window closes, and the capture window opens starting the face detection process.

CAPTURE WINDOW

The opened window shows the camera-feed with a rectangle around the detected face (if any) and a number in the top left corner. The number represents the time left before the detection is “unfrozen” (more on this in the backend section)

¹ The prototype is limited to only producing executables for the host machine’s OS. It can be expanded using Docker or utilizing a different bundler.

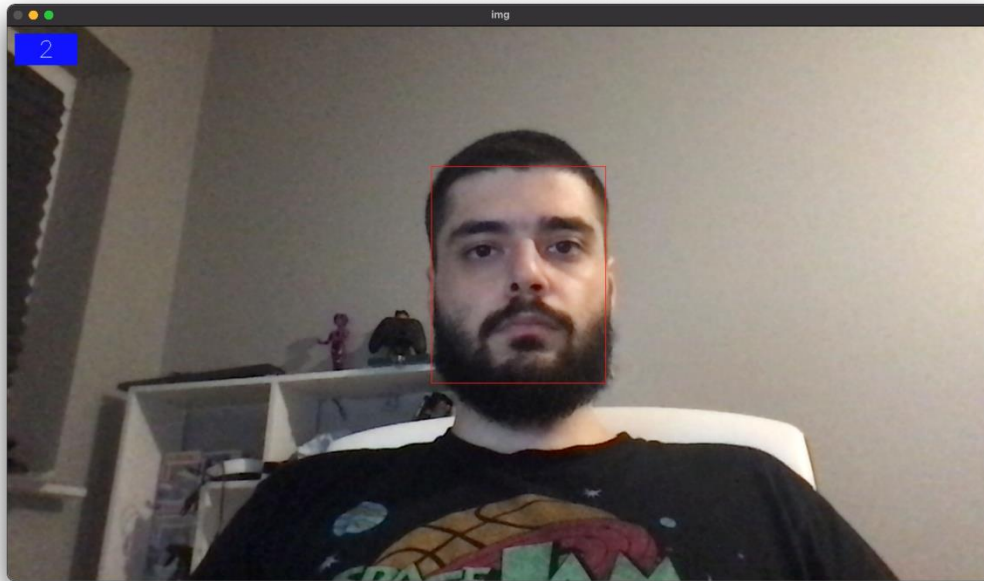


Figure 6. Capture window.

This component is responsible for face detection, alignment, cropping and sending the captured images to the verification server.

FACE DETECTION

The face detection solution uses ResNet SSD model (Single Shot Multi-Box Detection, more on SSD in Chapter 2) and its pre-trained weights provided by OpenCV community. However, OpenCV's deep neural network module can load external caffe models, so the solution is expandable, and the model chosen is interchangeable.

Since the model expects a certain sized input, the images taken from the camera are scaled down to that size. Most of the time models require low resolutions (300x300 in case of ResNet SSD), so to maintain modularity and customizability of the system, the images sent to the verification server are scaled back to the original resolution.

The scaled down images are then fed forward to the caffe model. The resulting matrix is then filtered by face features.² This produces a set of coordinates that are then scaled up to the original resolution, and the result is a set of coordinates that define the bounding box containing the detected face.

PROCEDURES & ERROR PREVENTION

² The naming for these features will also differ depending on the model used, so filtering depends on the choice of the model.

To prevent false positives and false negatives, as well as a waste of verification server's resources, the client will only send the detected face to the verification server if the face has been detected in a set number of consecutive frames. This amount is customizable, and the prototype runs at five consecutive frames.

Since the faces are detected each frame, which can happen more than once per second, sending an image each time requires considerable transmission bandwidth. In addition, the verification server might get overloaded and crash during sessions with a large number of students. To prevent both, after sending an image to the server, the client stops the detection and the process of sending a face for a set time. This time is customizable, and the prototype uses five seconds. During the 'freeze,' the image captured in the last frame is cropped to the size of the bounding box so that it only contains the contents of the box. The resulting image is then encoded into a base-64 format and sent to the verification server along with the examinee identifier in the body of a POST request.

2.3 VERIFICATION SERVER

This module is responsible for representation and verification of faces. The start of the verification server signifies the start of the session. Once the session starts, the verification server will receive POST requests containing the cropped images of faces to verify from the clients. Upon receiving such request and handling the resulting files, the server feeds the image to the VGG-Face based CNN to receive a vector representation of the face. It then verifies it against an existing base image that is set during session configuration. The result of the comparison is then sent to the exam supervisor's dashboard via a POST request, body of which contains the student identifier and the result of verification. The module contains the representation and verification units of the pipeline. It is implemented in Python with the use of Flask framework to manage receiving and sending requests.

FILE HANDLING

Upon receiving a POST request from a client, the server notes the request number and the examinee identifier of the request's author. First, the server must check whether an examinee that authored the request is registered for the session. It does so by checking the existence of a directory named after the examinee identifier configured via the supervisor's dashboard prior to the start of the session. While inefficient, it allows for modularity and component flexibility. If it does not, the server ignores the request, otherwise it may proceed further. The server then decodes the received image from base-64. The server then checks whether a base image for the examinee exists. If it does not, it uses the first image received during the session as a base image for that examinee. The server then saves the image locally in the directory of the base image of the examinee. Then, the server cleans the directory so that up to nine last images and the base image remain. Saving the images to the hard drive is also a measure to handle the case of server crashing, as there are no measures preventing the server from crashing. After that, the representation phase of the pipeline begins.

REPRESENTATION

The aim of this phase is to transform the received image to vector form, so that the verification is possible. This is done by feeding the image to the VGG-Face based CNN (more on this in Chapter 3) to receive a vector representation of the face. The model itself is state-of-the-art and produces results with 98.95% accuracy when evaluated on LFW (Labeled Faces in the Wild) dataset. [13] The model performs better than most state-of-the-art alternatives, including Google's FaceNet (without alignment 98.87% accuracy) and Facebook's DeepFace (97.35% accuracy) while requiring less data and using simpler network architecture. [13] However, the choice of the model used is customizable and may also be changed to a different one.

Since the image is already aligned and cropped, there is no need for extra face detections and any other pre-processing. Upon receiving the vector representation of an image, the server checks whether the same image already exists, and if it does, the verification result will return 'false,' meaning the examinee is cheating. It does so by finding the difference between the vector representation of each image in the directory dedicated to the examinee. If the server did not stop here, it caches the vector representation to memory to optimize the performance. Also, it releases the memory taken by the previous allocation.

The number of allocations is customizable due to the limitations caused by the size of the vector representation. VGG-Face produces a vector of length 2622, so in the prototype the number of caches is set to three per examinee, otherwise there is a risk of exhausting the RAM.

VERIFICATION

Once a vector representation of two images is obtained, it is possible to compare the two. The assumption is that images of the face belonging to the same person will have similar vector representations, within a set margin of error. There are two ways to find how similar the vector representations are considered for this purpose.

EUCLIDEAN DISTANCE

The Euclidean distance between two vectors may be represented by the following formula:

$$ED = \sum_{i=0}^n \sqrt{(x_i - y_i)^2}, \quad (1)$$

where n represents the length of the vector representation of an image.

The verdict on whether the face shown on the first image is the same face shown in the second, may be determined by comparing the obtained in (1) Euclidean Distance to a set threshold value:

$$Verdict = ED < threshold$$

In the prototype, Euclidean distance is the verification method of choice, and the threshold is set to 0.55 as recommended by the authors of the model. It is completely customizable, and for stricter results one may want to decrease it.

COSINE DISTANCE

Let a and b be vectors, and θ be the angle between them.

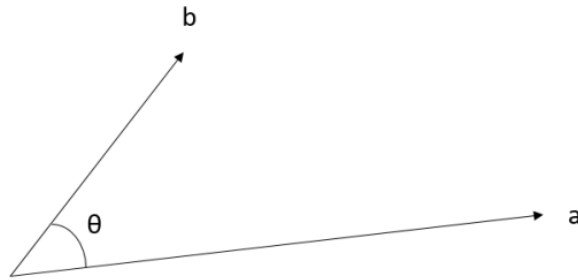


Figure 7. Vectors a and b .

Let vector c be equal to $a - b$. Vectors a , b and c create a valid triangle.

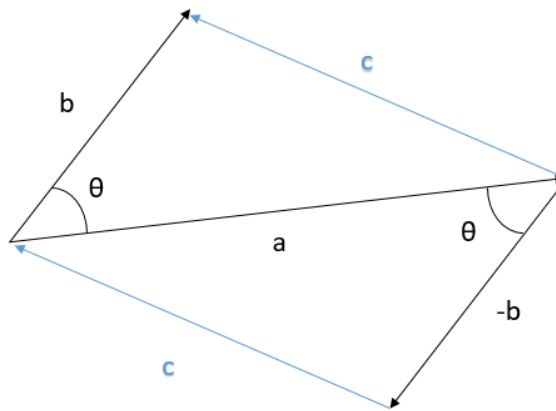


Figure 8. Vectors a , b , and c .

Therefore, the law of cosines states that:

$$\|c\|^2 = \|a\|^2 + \|b\|^2 - 2\|a\|\|b\|\cos\theta,$$

where $\|a\|$, $\|b\|$ and $\|c\|$ are the vector lengths of a , b and c respectively.

Then,

$$\|c\|^2 = c \cdot c = (a - b)(a - b) = a \cdot a - a \cdot b - b \cdot a + b \cdot b = \|a\|^2 + \|b\|^2 - a \cdot b - b \cdot a$$

(1)

$$\|c\|^2 = \|a\|^2 + \|b\|^2 - 2a \cdot b \quad (2)$$

$$\|c\|^2 = \|a\|^2 + \|b\|^2 - 2\|a\|\|b\|\cos\theta = \|a\|^2 + \|b\|^2 - 2a \cdot b \quad (3)$$

$$\|a\|\|b\|\cos\theta = a \cdot b \quad (4)$$

$$a_1b_1 + a_2b_2 + \dots + a_nb_n = \|a\|\|b\|\cos\theta \quad (5)$$

$$\cos\theta = \frac{a_1b_1 + a_2b_2 + \dots + a_nb_n}{\|a\|\|b\|} \quad (6)$$

And, from Pythagorean theorem, the length of a vector is:

$$\|V\| = \sqrt{\sum_{i=1}^n V_i^2} \quad (7)$$

Then, by combining (6) and (7), cosine similarity can be defined as:

$$\text{cosine similarity} = \frac{a_1b_1 + a_2b_2 + \dots + a_nb_n}{\sqrt{\sum_{i=1}^n a_i^2} \sqrt{\sum_{i=1}^n b_i^2}} = \frac{a^T b}{\sqrt{a^T a} \sqrt{b^T b}}$$

In this way, similar vectors will produce comparable results. Cosine distance is then defined by:

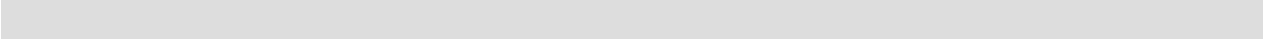
$$\text{cosine distance} = 1 - \text{cosine similarity}$$

Like in the case of Euclidean distance, the verdict of whether the face in the first image is the same as the same in the second image is determined by comparing the calculated distance to a set threshold:

$$\text{verdict} = \text{cosine distance} < \text{threshold}$$

VERDICT

Once a verdict is obtained through comparing Euclidean or Cosine distance to a set threshold, it is packed into the body of a POST request, along with the examinee identifier. The POST request is then sent to the supervisor's dashboard, where the status of that examinee is updates, along with the last update request time.



CHAPTER 3. DETECTION SOLUTION STUDY & ANALYSIS

When choosing a solution for face detection, there are three main factors to consider: speed, accuracy and resources required. To choose the optimal detector, a study was conducted. The face detectors considered are:

1. Haar Cascade, offered by OpenCV.
2. Single Shot Multibox Detector (SSD), offered by OpenCV.
3. Histogram of Oriented Gradients (HOG), offered by Dlib.
4. Max-Margin Object Detection (MMOD), offered by Dlib.
5. Multi-task Cascaded Convolutional Network (MTCNN), offered by its own library.

The detectors listed above contain both legacy (Haar Cascade, HOG) and state-of-the-art solutions. The aim of the study is to describe and compare the listed above face detection methods in terms of speed, accuracy and computational resources required. The study is conducted on a 2.6 GHz 6-Core Intel i7 machine with 16 GB 2667 MHz DDR 4 RAM.

SPEED

Each of the above detectors is used to find a face in 6600 consecutive frames containing a single face. The time required to detect the face is recorded. The Activity Monitor is monitored throughout the experiment to find the relative resources taken up by the calculations for each detector (CPU Load & Memory Load).

The results are presented in Table 1 below, Table 6 (in the appendix section), and Figure 9. MMOD is the slowest detector, while Haar Cascade is the fastest on average, with SSD being the close second.

Face Detection Speed Comparison
Vertical axis - logarithmic time

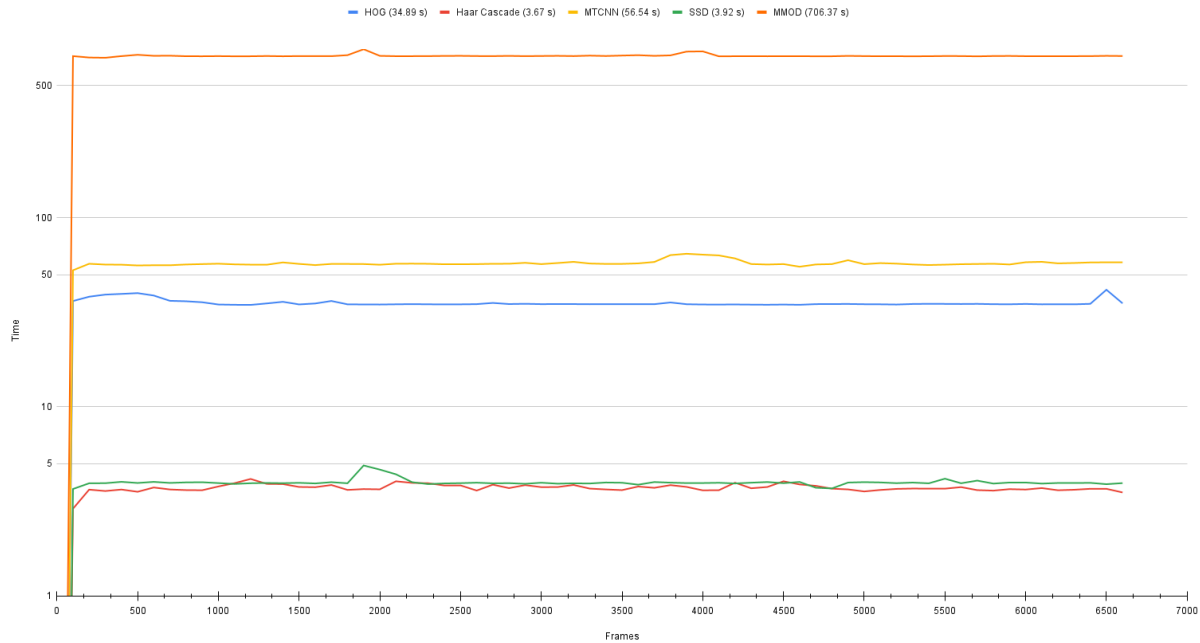


Figure 9. Chart representing the speed of detection of different face detectors. The values are taken from Table 1.

Table 1. Detector Speed Comparison

Detector	HOG	Haar Cascade	MTCNN	SSD	MMOD
Avg. per 100 frames (s)	34.89	3.67	56.54	3.92	706.37
Avg. per 1 frame (s)	0.35	0.04	0.57	0.04	7.06
Avg. frames per second:	2.87	27.22	1.77	25.50	0.14

ACCURACY

The accuracy of a detector is determined by calculating the Area Under Curve (AUC) of the corresponding Receiver Operating Characteristic Curve (ROC).

ROC

An ROC curve is a graph showing the performance of a classification model at all classification parameters: True Positive Rate and False Positive Rate.

True Positive Rate (**TPR**) is a synonym for recall and is defined as follows: (“Classification: ROC Curve and AUC - Google Developers”)

$$TPR = \frac{TP}{TP + FN},$$

where TP is the number of True Positive results, and FN is the number of False Negative results.

False Positive Rate (**FPR**) is defined as follows:

$$FPR = \frac{FP}{FP + TN},$$

where FP is the number of False Positives, and TN is the number of True Negatives.

An ROC curve plots TPR over FPR at different classification thresholds. Lowering the classification threshold classifies more items as positive. Therefore, it increases both FPR and TPR.

AUC

Area Under the ROC curve measures the entire two-dimensional area underneath the ROC curve from (0, 0) to (1, 1). The values range from 0 to 1. A detector whose predictions are always wrong across a dataset will have an AUC of 0.0, while a detector whose predictions are always right will have an AUC of 1.0. This metric is desirable for two reasons: it is scale-invariant (measures how well predictions are ranked as opposed to their absolute values), and it is classification-threshold-invariant (measures the quality of the model’s predictions regardless of the classification threshold chosen).

RESULTS

The accuracy experiments were conducted on the Fddb [12] dataset and the percentages shown are based on AUC values.³ In descending order of accuracy:

1. MTCNN – ~90% [9]
2. SSD – ~88% [10]
3. MMod – ~70% [8]
4. HOG – ~60% [8]
5. Haar Cascade – ~50% [11]

³ The accuracies are taken from their respective references

RESOURCES

The observations are made on the data produced by the Activity Monitor that creates a new output entry every ten seconds. Results are the outputs of the Activity Monitor regarding the average CPU Load and RAM usage over the period of computation, presented in Table 2 and Table 3, respectively.

Table 2. CPU Load Comparison by Detectors

Detector	HOG	Haar Cascade	MTCNN	SSD	MMOD
CPU Load Before (%)	2.56	11.55	2.47	11.48	2.21
CPU Load After (%)	9.52	27.63	12.39	28.05	12.48
Difference (%)	6.96.	16.08	9.92	16.57	10.27

Table 3. RAM Use Comparison by Detector

Detector	HOG	Haar Cascade	MTCNN	SSD	MMOD
RAM Before (GB)	7.8	6.2	10.7	6.2	10.9
RAM During (GB)	8.2	6.8	11.1	6.7	13.9
Difference (GB)	0.4	0.6	0.4	0.5	3.0

ANALYSIS

While Haar Cascade is one of the fastest methods (avg. 27.22 frames per second), it has the lowest accuracy out of all tested (approx. 50%). Using Haar Cascade may lead to production of inaccurate data even with the error prevention procedures (described in the next section) in place.

HOG is another legacy method that does not compete with the state-of-the-art methods. Not only is the accuracy just approximately 10% better than Haar Cascade but it is also much slower (avg. 2.87 frames per second), despite loading the CPU the least, while using as much RAM as MTCNN.

MTCNN overperforms all tested methods in terms of accuracy (approx. 90%). However, the speed at which it can detect faces (avg. 1.77 frames per second) makes it inapplicable in real-time applications. However, this method is the most efficient in terms of resources used over accuracy.

While MMOD reaches the state-of-the-art status due to its accuracy (approx. 70%), the detection speed is the slowest out of all considered detectors. It is slower than MTCNN by a factor of twelve. It also uses the most resources and would not be feasible in a commercial application – 3 GB of RAM.

SSD is the perfect detector for real time applications due to its fast speed of detection (avg. 25.5 frames per second), high accuracy (approx. 88%) and low resources requirements – lower than Haar Cascade in terms of RAM used and approximately the same CPU Load.

CONCLUSION

While all the detectors have their advantages and disadvantages, Single Shot Multibox Detector (SSD) is the perfect detector for real time applications with its low resource use, high speed, and a

CHAPTER 3. VGG-FACE

Pre-trained VGG-Face and its derivative (ResNet) models are used during the detection and representation stages of the face recognition pipeline in the prototype. These models are CNN architectures for face identification and verification. Compared to other state-of-the-art models, VGG-Face has a much simpler architecture that achieves state-of-the-art results on several image and video face recognition benchmarks. [13] This chapter covers the network architecture, its training, as well as evaluation of the results against existing datasets.

NETWORK ARCHITECTURE AND TRAINING

This section describes the process of training the CNN, as well as its structure. Due to the unavailability of large quantities of training data to the public, the training process was omitted in the implementation of the prototype and instead a model with pretrained weights (provided by the OpenCV community) is used. The data set used to train the model contains 2622 identities with up to one thousand images per identity. [13]

LEARNING FACE CLASSIFIER

Let ϕ denote the deep architecture considered. Initially, they are bootstrapped by structuring the problem of recognizing $N = 2622$ individuals as a N-ways classification problem. This improves the training speed, as well as makes the training simpler. [13]

Let $l_t, t = 1, \dots, T$ denote each training image. The CNN associates each training image one score vector per identity with the use of final fully connected layer containing N linear predictors $W \in \mathbb{R}^{N \times D}$, $b \in \mathbb{R}^N$. The score vectors $x_t = W\phi(l_t) + b$ are compared to the ground-truth class identity $c_t \in \{1, \dots, N\}$. It is accomplished by computing the softmax log-loss:

$$E(\phi) = - \sum_{t=1}^T \log \left(\frac{e^{e_{c_t} \cdot x_t}}{\sum_{q=1}^N e^{e_q \cdot x_t}} \right),$$

where e_c denotes the one-hot vector of class c .

After learning, the classifier layer can be removed, and the score vectors can be used for face identity verification. This corresponds to the pre-trained weights in the prototype. The identity verification can be performed by finding the Euclidean distance as described in Chapter 1. However, the scores can be further improved by adjusting them for verification in Euclidean space using *triplet loss* training scheme.

TRIPLET LOSS

The goal of this training is to improve the learning scores by comparing face descriptors in Euclidean space. The output pretrained $\phi(l_t) \in \mathbb{R}^D$ is l^2 -normalized and projected to a $L \ll D$ dimensional space using

$$x_t = \frac{W' \phi(l_t)}{\|\phi(l_t)\|_2},$$

where $W' \in \mathbb{R}^{L \times D}$.

Note that unlike in the linear predictor from the previous section, $L \neq D$ is not equal to the number of class identities but is the size of the descriptor embedding (the authors of VGG-Face set $L = 1024$). Also, the projection W' is trained to minimize *triplet loss*:

$$E(W') = \sum_{(a,b,c) \in T} \max \{0, \alpha - \|x_a - x_c\|_2^2 + \|x_a - x_b\|_2^2\},$$

where $x_i = \frac{W' \phi(l_i)}{\|\phi(l_i)\|_2}$, $\alpha \geq 0$ is a fixed scalar called *learning margin*, and T is a collection of *learning triplets*.

The learning bias is also canceled in the differences in the above equation. From the learning triplet (a, b, c) , a is the *anchor* face image, $b \neq a$ and c are positive and negative examples of anchor's identity, respectively.

ARCHITECTURE

The detailed architecture of the CNN is given in Table 7. It comprises of eleven blocks, a linear operator followed by one or more non-linearities (ReLU or max pooling). (“Comparative Study of Human Age Estimation Based on Hand-Crafted and ...”) The first eight blocks are exclusively linear convolutions, whereas the remaining three are Fully Connected (FC) – linear convolutions where the size of the filters matches the size of the input data. The first two FC layers output a 4096-dimensional vector, while the last FC layer has $N = 2622$ dimensions. The resulting vector then passes through a softmax layer to compute the class posterior probabilities.

The input is a face image of size 224×224 with the average face image subtracted. The average face image is computer from the from the training set.

TRAINING

According to [13], learning the N-way face classifier follows the steps described by [14] with modifications described by [6]. The goal of training is to find parameters of the CNN to minimize the average prediction log-loss after the softmax layer. The optimization happens through stochastic gradient descent (SGD) in batches of 64 samples and momentum coefficient of 0.9, and the model is regularized using dropout and weight decay. The model was trained using three decreasing learning rates. [13]

During training, the network is fed with random 224×224 pixel patches cropped from the images from the training set, with crops changing every time an image was sampled. The color channel augmentations mentioned in [14] and [6] were not applied, whereas the images were flipped horizontally with 50% probability. [13]

For learning using triplet loss, the CNN, except for the last Fully Connected layer, is frozen. The FC layer is then learnt for ten epochs using SGD with a fixed learning rate of 0.25. An epoch contains all positive pairs (a, b) , where a is the anchor image and b is its paired positive example. To ensure balanced learning, each pair (a, b) is extended to a triplet (a, b, n) , where n is selected at random, such that it violates triplet loss margin. This avoids training with examples that are too hard. [13]

EVALUATION AND RESULTS

While the training was done on an original dataset assembled by [13], to allow for comparison to other state-of-the-art models, evaluation is performed on existing benchmark datasets – LFW and YTF.

Table 4 and Table 5 below compare the accuracy results of the VGG-Face model to other state-of-the-art models using the LFW and YFD benchmark datasets, respectively. The accuracy is calculated the same way it is described in Chapter 2.

Table 4. LFW unrestricted settings.

No	Method	Images	Networks	Accuracy
1	Fisher Vector Faces	-	-	93.10
2	DeepFace	4M	3	97.35
3	Fusion	500M	5	98.37
4	DeepID-2,3		200	99.47
5	FaceNet	200M	1	98.87
6	FaceNet + Alignment	200M	1	99.63
7	VGG-Face	2.6M	1	98.95

Table 5. YFD unrestricted settings. K indicates the number of faces used to represent each video.⁴

No	Method	Images	Networks	Accuracy
1	Video Fisher Vector Faces	-	-	83.8
2	DeepFace	4M	1	91.4
3	DeepID-2,2+,3	-	200	93.2
4	FaceNet + Alignment	200M	1	95.1
5	VGG-Face (K=100)	2.6M	1	91.6
6	VGG-Face (K=100) + Embedding learning	2.6	1	97.3

⁴ All the data is taken from [13].

SSD is a method for detecting objects in images using a single neural network. Compared to methods that require object proposals, it is easy to train because it does not rely on proposal generation or subsequent pixel (or feature) resampling and encapsulates all computation in a single network. [1] Hence, integration into face recognition pipelines does not entail a cost that would normally appear when switching detectors. SSD is the first deep learning-based method that does not resample pixels or features for bounding box hypothesis and is as accurate as the methods that do while maintaining higher speeds of detection.

This section covers the detection method in detail. Some experiments regarding comparison to other state-of-the-art solutions have already been conducted and analyzed in Chapter 2.

ARCHITECTURE

The method relies on feed-forward CNN to produce a fixed-size collection of bounding boxes and scores, which used to determine the presence of instances of a set object class on the boxes. The method uses ResNet (a network based on VGG-16 up to layer *conv5_3*, described in Chapter 3), further referred to as *base network*. [1] add auxiliary structure that produce detections with multi-scale feature maps, convolutions predictors, and default boxes and aspect ratios.

To support the multi-scale feature maps feature, [1] add convolutional feature layers to the end of the base network, which progressively decrease in size allowing predictions of detections with multiple scales. The detection prediction model is unique to each feature layer. Each extra feature layer may use a set of convolutional filters (denoted in the figure below) to produce those detection predictions.

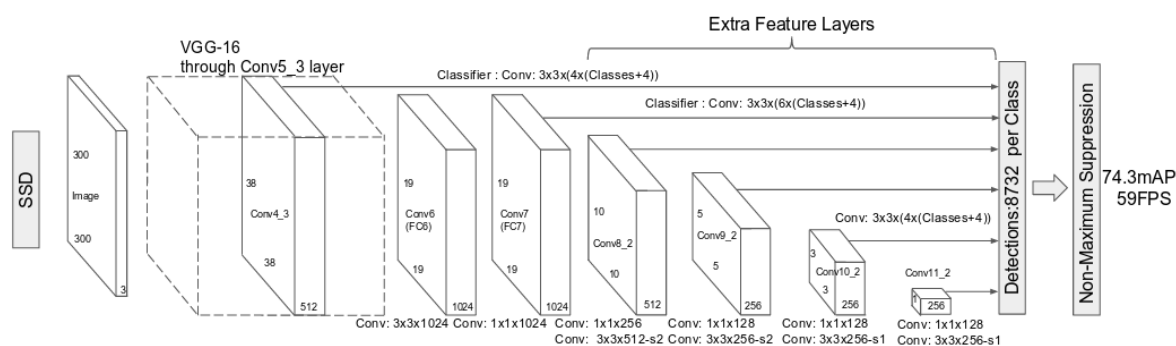


Figure 10. ResNet SSD network structure.

Let $m \times n$ denote the size of a feature layer and p denote the number of channels in that feature layer. A potential prediction's parameters are predicted with a filter of size $3 \times 3 \times p$, which produces a score for a category (or an offset relative to the default box coordinates).

For multiple feature maps at the top of the network, a default bounding box is associated for each cell. The default boxes are positioned in a way that ensures that the position of each box relative to its corresponding cell is fixed. At each cell, the offsets relative to the default box shapes in the cell are predicted, as well as the scores per class that signify the presence of an instance of that class in each of those boxes. For an $m \times n$ feature map, for each default box out of k at a given location, c class scores are

computed along with four offsets relative to the original default box shape. These yield $(c + 4)kmn$ outputs with $(c + 4)k$ filters. By applying default boxes to several feature maps at different resolutions, the model efficiently approximates the space of possible output box shapes.

TRAINING

While SSD is comparatively easier to train than other object detectors, the problem of the availability of annotated datasets of faces described in the previous chapter persists, which forced the use of a pre-trained model in the prototype. Nevertheless, it is important to note that the training process of SSD differs from other detectors that use region proposals. Namely, the ground truth information must be assigned to specific outputs in the fixed set of detector outputs. A prerequisite to training the model is choosing a set of default boxes and scales for detection. It is also necessary to choose hard negative mining and data augmentation strategies.

Each ground truth box, selected from a set of default boxes of varying locations, scaling and aspect ratios, is matched to the default box with the best Jaccard overlap. This is done to determine which default boxes correspond to ground truth during training. Then, default boxes that have a Jaccard overlap with any ground truth higher than a set threshold, are matched. This avoids making the network to pick only one default box with maximum overlap, allowing the network to pick multiple overlapping boxes, which simplifies the learning problem.

The training objective of SSD is the extended version of MultiBox. [1] Let $x_{ij}^p \in \{1, 0\}$ be the result of matching the i -th default box to j -th ground truth in category p . The objective loss function is defined as follows:

$$L(x, c, l, g) = \frac{1}{N} (L_{conf}(x, c) + \alpha L_{loc}(x, l, g)),$$

where N is the number of matched default boxes, loc is localization loss (Smooth L1 loss between predicted box l and ground truth box g), $conf$ is confidence loss, c is classes confidence, and α is the weight term, which is set to 1 by cross validation. The offsets for the bounding box center are regressed for its width and height.

To improve and smoothen the semantic segmentation results, [1] use feature maps from both upper and lower layers of the network. This helps capture more details from the input. Increasing the number of feature maps does not cause a huge increase of computational overhead. Moreover, the default boxes are designed in a way to allow specific feature maps learn to respond to certain scales of objects. The scale of default boxes for each feature map is computed using the formula:

$$s_k = s_{\min} + \frac{s_{\max} - s_{\min}}{m - 1} (k - 1), \quad k \in [1, m]$$

where m is the number of feature maps used for prediction, s_{\min} is the scale of the lowest layer, and s_{\max} is the scale of the highest layer. It is also assumed that all layers are spaces regularly. Let the set of different aspect ratios is applied to the default boxes be denoted as $a_r \in \{1, 2, 3, \frac{1}{2}, \frac{1}{3}\}$. The width and the height of the default boxes is computed using the following formulas, respectively:

$$w_k^a = s_k \sqrt{a_r}$$

$$h_k^a = \frac{s_k}{\sqrt{a_r}}$$

Also, in the case of $a_r = 1$, and additional default box with $s'_k = \sqrt{s_k s_{k+1}}$ is added, resulting in six default boxes per feature map location. Each default box's center is set to $(\frac{i+0.5}{|f_k|}, \frac{j+0.5}{|f_k|})$, where $|f_k|$ is the size of the k -th square feature map, and $i, j \in [0, |f_k|]$.

The combination predictions for all default boxes with different scales and aspect ratios achieves a diverse set of predictions, which covers a vast amount of input object sizes and shapes. This, however, leads to most default boxes being negative matches. In order to balance the positive and negative examples during training, the negative examples are sorted by descending confidence loss for each default box and the first ones are picked in a fashion that preserves the ratio of 3:1 of negatives to positives. *W. Liu et al. (2016)* suggest that this leads to faster optimization and a more stable training.

REFERENCES

1. Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C. Y., & Berg, A. C. (2016, October). Ssd: Single shot multibox detector. In European conference on computer vision (pp. 21-37). Springer, Cham.
2. Zhang, K., Zhang, Z., Li, Z., & Qiao, Y. (2016). Joint face detection and alignment using multitask cascaded convolutional networks. *IEEE signal processing letters*, 23(10), 1499-1503.
3. Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28.
4. Gao, X., Xu, J., Luo, C., Zhou, J., Huang, P., & Deng, J. (2022). Detection of Lower Body for AGV Based on SSD Algorithm with ResNet. *Sensors*, 22(5), 2008.
5. Serengil, S. I., & Ozpinar, A. (2020, October). Lightface: A hybrid deep face recognition framework. In 2020 innovations in intelligent systems and applications conference (ASYU) (pp. 1-5). IEEE.
6. Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.
7. Owusu, E., Abdulai, J. D., & Zhan, Y. (2019). Face detection based on multilayer feed - forward neural network and haar features. *Software: Practice and Experience*, 49(1), 120-129.
8. Cheney, J., Klein, B., Jain, A. K., & Klare, B. F. (2015, May). Unconstrained face detection: State of the art baseline and challenges. In 2015 International Conference on Biometrics (ICB) (pp. 229-236). IEEE.
9. Ma, Mei, and Jianji Wang. "Multi-view face detection and landmark localization based on MTCNN." In 2018 Chinese Automation Congress (CAC), pp. 4200-4205. IEEE, 2018.
10. Granger, E., Kiran, M., & Blais-Morin, L. A. (2017, November). A comparison of CNN-based face and head detectors for real-time video surveillance applications. In 2017 Seventh International Conference on Image Processing Theory, Tools and Applications (IPTA) (pp. 1-7). IEEE.
11. Pattarapongsin, P., Neupane, B., Vorawan, J., Sutthikulsombat, H., & Horanont, T. (2020, July). Real-time drowsiness and distraction detection using computer vision and deep learning. In Proceedings of the 11th International Conference on Advances in Information Technology (pp. 1-6).
12. Jain, V., & Learned-Miller, E. (2010). Fddb: A benchmark for face detection in unconstrained settings (Vol. 2, No. 6). UMass Amherst technical report.
13. Parkhi, O. M., Vedaldi, A., & Zisserman, A. (2015). Deep Face Recognition. *British Machine Vision Conference*.
14. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25.

List of Tables

Table 1. Detector Speed Comparison	25
Table 2. CPU Load Comparison by Detectors	27
Table 3. RAM Use Comparison by Detector	27
Table 4. LFW unrestricted settings.	32
Table 5. YFD unrestricted settings. K indicates the number of faces used to represent each video.	32
Table 6. Time It Took to Detect a Face In 100 Frames for Each Detector in The Study	40
Table 7. Network Configuration.....	43

List of figures

Figure 1. Request flow diagram and the spread of the face recognition pipeline. The red rectangles are the stages of the face recognition pipeline.	10
Figure 2. Default UI of the Dashboard.....	11
Figure 3. Add Student Form.....	12
Figure 4. Table after a new examinee has been added.	12
Figure 5. Client Form	16
Figure 6. Capture window.....	17
Figure 7. Vectors a and b.	21
Figure 8. Vectors a, b, and c.....	21
Figure 9. Chart representing the speed of detection of different face detectors.....	25
Figure 10. ResNet SSD network structure.	33

List of appendices

Appendix A. Large Tables	40
--------------------------------	----

APPENDIX A. LARGE TABLES

Table 6. Time It Took to Detect a Face In 100 Frames for Each Detector in The Study

Frames	HOG (34.89 s)	Haar Cascade (3.67 s)	MTCNN (56.54 s)	SSD (3.92 s)	MMOD (706.37 s)
0	1.91E-06	3.10E-06	1.19E-06	9.54E-07	9.54E-07
100	36.23554897	2.892754078	52.70409131	3.675975084	714.192678
200	38.19530511	3.647857904	57.03780794	3.940686941	702.5152059
300	39.1812408	3.586294889	56.46708608	3.945479155	700.4858158
400	39.52439189	3.651924849	56.37909412	4.010371923	714.3655293
500	39.90351486	3.553075314	55.89740419	3.955664158	726.1441998
600	38.74442697	3.743636131	56.03834772	4.008642197	717.1490119
700	36.33512497	3.65500474	56.02677226	3.957942724	718.219789
800	36.13481593	3.626913071	56.51862001	3.983809948	713.509789
900	35.70137191	3.620682955	56.81792307	3.991145372	713.1944649
1000	34.73823833	3.790052176	57.09005785	3.95042491	714.4203029
1100	34.61590981	3.94590497	56.62220502	3.909925938	712.8579521
1200	34.57207179	4.153652906	56.41286898	3.943305969	713.2627251
1300	35.18846607	3.911190033	56.37266803	3.955844879	715.407809
1400	35.89384604	3.903182983	57.96661687	3.945614815	713.1372221
1500	34.77288485	3.768427134	56.94543505	3.962347984	714.8470218
1600	35.16177893	3.756734133	56.12602091	3.933098078	714.520968
1700	36.26053715	3.859287024	56.91828895	3.993356705	714.2139311
1800	34.80420804	3.635421991	56.9134059	3.941020012	722.8619452
1900	34.76336527	3.672314167	56.83629584	4.894211054	776.919234
2000	34.73393321	3.660866022	56.36047101	4.656308174	716.854419
2100	34.82744288	4.037606955	57.07335114	4.396603107	713.9079881
2200	34.90236497	3.958940029	57.12043285	3.994015932	714.0851352
2300	34.8298192	3.943880081	57.02983022	3.901252985	714.5670102
2400	34.78930902	3.836150885	56.69914913	3.934653044	716.274045
2500	34.81329298	3.83989501	56.71609712	3.947287083	716.9648211
2600	34.89078975	3.60651803	56.802001	3.969969034	715.0454102
2700	35.40051413	3.874823809	57.01909995	3.938131094	714.5149717
2800	34.92322326	3.715693951	57.05367613	3.946982861	716.5775499

Frames	HOG (34.89 s)	Haar Cascade (3.67 s)	MTCNN (56.54 s)	SSD (3.92 s)	MMOD (706.37 s)
2900	35.02890015	3.854228973	57.70291305	3.915117741	714.1904569
3000	34.89464712	3.760348797	56.80151296	3.96803093	715.6364722
3100	34.92685199	3.763174057	57.54318213	3.922150135	716.953058
3200	34.91502595	3.861320972	58.38510585	3.938748837	714.8684709
3300	34.86225367	3.693748951	57.2086637	3.930501938	718.7691603
3400	34.87888098	3.656505823	56.935462	3.978216887	715.604208
3500	34.86684799	3.625830889	56.95947814	3.966948032	719.419198
3600	34.88854241	3.786928892	57.3007431	3.874152899	722.7068
3700	34.86988211	3.728536129	58.29726315	3.999593973	716.939121
3800	35.60413408	3.853372812	63.3768971	3.971475124	721.0257149
3900	34.85610509	3.770243168	64.4227891	3.950376272	754.906004
4000	34.73978376	3.616067171	63.72983575	3.9523561	756.9659011
4100	34.72175217	3.62292695	63.14247298	3.965316057	712.736068
4200	34.75443697	3.969269991	60.87343502	3.932665825	713.4003789
4300	34.6889441	3.708190918	56.85196495	3.969374895	713.6921852
4400	34.63854909	3.766145945	56.5027101	4.000520945	713.2438259
4500	34.71834493	4.037469149	56.80987072	3.946408033	713.7064109
4600	34.61712503	3.885704041	55.02467823	4.004537821	714.096406
4700	34.90109587	3.82043314	56.53544235	3.733859777	712.7653639
4800	34.93319106	3.691066027	56.81591392	3.700518847	713.1198988
4900	34.95742893	3.65775609	59.53966475	3.975763083	716.5449371
5000	34.86180305	3.567481041	56.77645206	3.996060848	714.6663461
5100	34.822685	3.633549213	57.48884678	3.981528044	713.2522249
5200	34.72906613	3.679870844	57.1327889	3.94652009	713.5582352
5300	34.95375896	3.695036888	56.53270698	3.974726915	712.8458319
5400	35.0191102	3.691630125	56.1580832	3.938524246	713.5904951
5500	34.988446	3.690860987	56.40633416	4.167500019	715.3053761
5600	34.94446492	3.760370016	56.73422885	3.941513062	714.889977
5700	34.99821973	3.625473976	56.89035892	4.070861816	712.391938
5800	34.8733511	3.603440046	57.03578591	3.926305056	715.3215871
5900	34.83911896	3.669696093	56.52842903	3.976537704	716.0530481
6000	34.96892524	3.651811123	58.10961103	3.976406097	713.9290781
6100	34.81871414	3.716593266	58.40186977	3.925348997	714.1170731

Frames	HOG (34.89 s)	Haar Cascade (3.67 s)	MTCNN (56.54 s)	SSD (3.92 s)	MMOD (706.37 s)
6200	34.84648705	3.619926929	57.31867075	3.955195189	713.8299489
6300	34.82126904	3.642750978	57.5792222	3.954037189	713.993417
6400	35.00901389	3.68410778	57.96823192	3.961057186	714.4916289
6500	41.56679916	3.683916092	58.05325794	3.897565842	716.917402
6600	35.2322619	3.524402857	58.05020332	3.948808908	715.40253

Table 7. Network Configuration

Layer	0	1	2	3	4	5	6	7	8
Type	Input	conv	relu	Conv	Relu	Mpool	Conv	Relu	Conv
Name	-	Conv1_1	Relu1_1	Conv1_2	Relu1_2	Pool1	Conv2_1	Relu2_1	Conv2_2
Support	-	3	1	3	1	2	3	1	3
Filt dim	-	3	-	64	-	-	64	-	128
Num filts	-	64	-	64	-	-	128	-	128
Stride	-	1	1	1	1	2	1	1	1
Pad	-	1	0	1	0	0	1	0	1

Layer	9	10	11	12	13	14	15	16	17
Type	Relu	Mpool	Conv	Relu	Conv	Relu	Conv	Relu	Mpool
Name	Relu2_2	Pool2	Conv3_1	Relu3_1	Conv3_2	Relu3_2	Conv3_3	Relu3_3	Pool3
Support	1	2	3	1	3	1	3	1	2
Filt dim	-	-	128	-	256	-	256	-	-
Num filts	-	-	256	-	256	-	256	-	-
Stride	1	2	1	1	1	1	1	1	2
Pad	0	0	1	0	1	0	1	0	0

Layer	18	19	20	21	22	23	24	25	26
Type	Conv	Relu	Conv	Relu	Conv	Relu	Mpool	Conv	Relu
Name	Conv4_1	Relu4_1	Conv4_2	Relu4_2	Conv4_3	Relu4_3	Pool4	Conv5_1	Relu5_1
Support	3	1	3	1	3	1	2	3	1
Filt dim	256	-	512	-	512	-	-	512	-
Num filts	512	-	512	-	256	-	-	512	-
Stride	1	1	1	1	1	1	2	1	1
Pad	1	0	1	0	1	0	0	1	0

Layer	27	28	29	30	31	32	33	34	35
Type	Conv	Relu	Conv	Relu	Mpool	Conv	Relu	Conv	Relu
Name	Conv5_2	Relu5_2	Conv5_3	Relu5_3	Pool5	Fc6	Relu6	Fc7	Relu7
Support	3	1	3	1	2	7	1	1	1

Layer	27	28	29	30	31	32	33	34	35
Filt dim	512	-	512	-	-	512	-	4096	-
Num filts	512	-	512	-	-	4096	-	4096	-
Stride	1	1	1	1	2	1	1	1	1
Pad	1	0	1	0	1	0	0	0	0

Layer	36	37
Type	Conv	Softmx
Name	Fc8	Prob
Support	1	1
Filt dim	4096	-
Num filts	1024	-
Stride	1	1
Pad	0	0