Continuous Face Detection and Verification During Online Examinations

Dominik Kurasbediani

Warsaw University of Technology

Author Note

[Include any grant/funding information and a complete correspondence address.]

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 serverside 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 some 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 clientside executables to allow easy and seamless process for the students.

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

Motivation

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. The most prevalent one, 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, etc.) and get a passing grade regardless of whether they truly know the material presented during the semester. During the COVID-19 pandemic, when there was no other option other than online education and online exams, the students were incentivized to abuse this system and commit academic offenses and get a passing grade without opening the course textbook once during the semester. This effectively lowers the overall education level, potentially creating severe errors, that 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 (first and last names, 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.

Chapter 1. System Architecture

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:
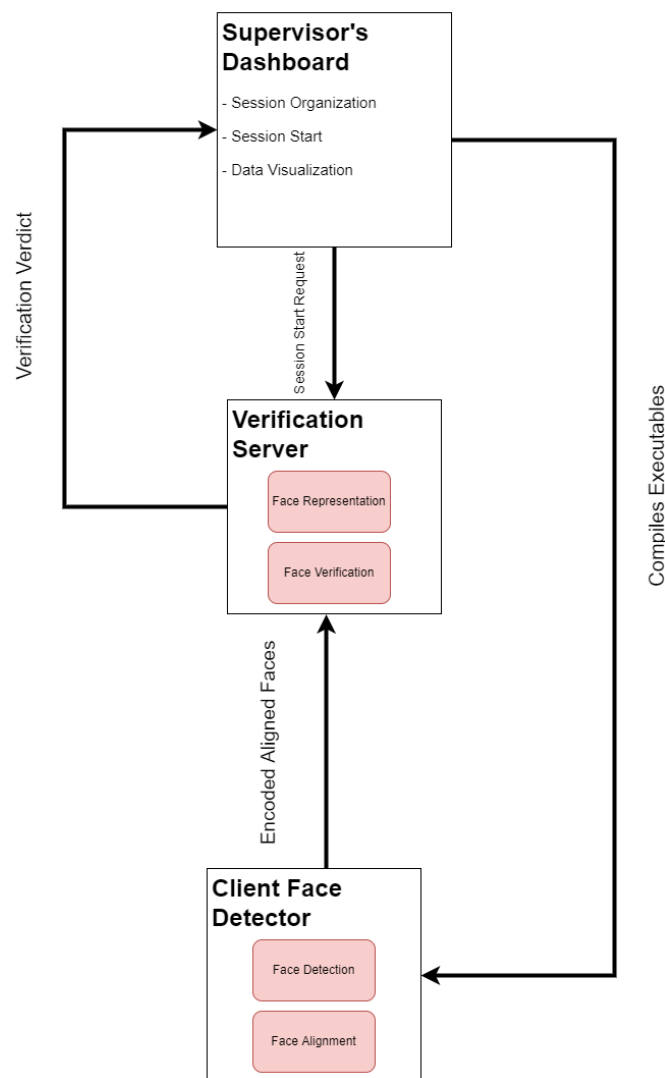

Figure 1. Request flow diagram and the spread of the face recognition pipeline. The red rectangles are the stages of the face recognition pipeline.

1.1 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 limitations of the design, it is impossible to utilize some of the features of React efficiently. Instead of only refreshing the contents when there is change automatically, the contents are manually refreshed with the given frequency. This results in a consistent amount of GET requests to the backend, which is more stable when there is a high volume of clients connected to the session.
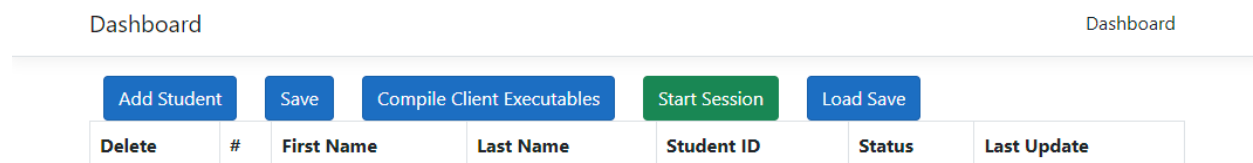

Figure 1. Default UI of the Dashboard

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.

Figure 2. 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.

Figure 3. 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 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. Then 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 is it 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.

1.2 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.
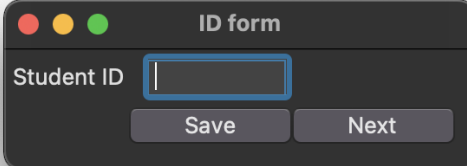

Figure 4. Client Form

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

---

[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.

'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)
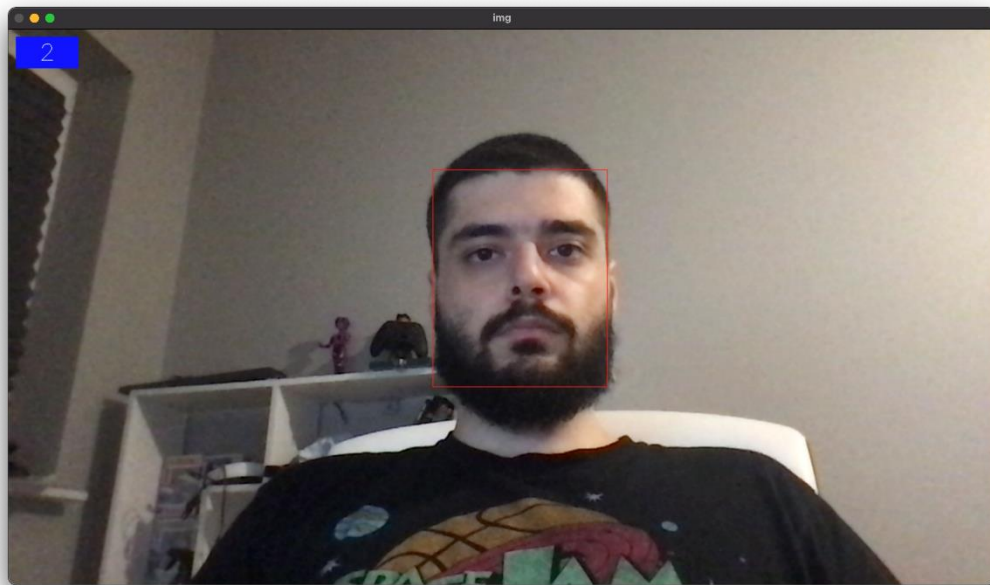
Fig. 5. Capture window.

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 in order 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.

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 most 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.

---

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

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 3, Table 4 and Figure 6. MMOD is the slowest detector, while Haar Cascade is the fastest on average, with SSD being the close second.
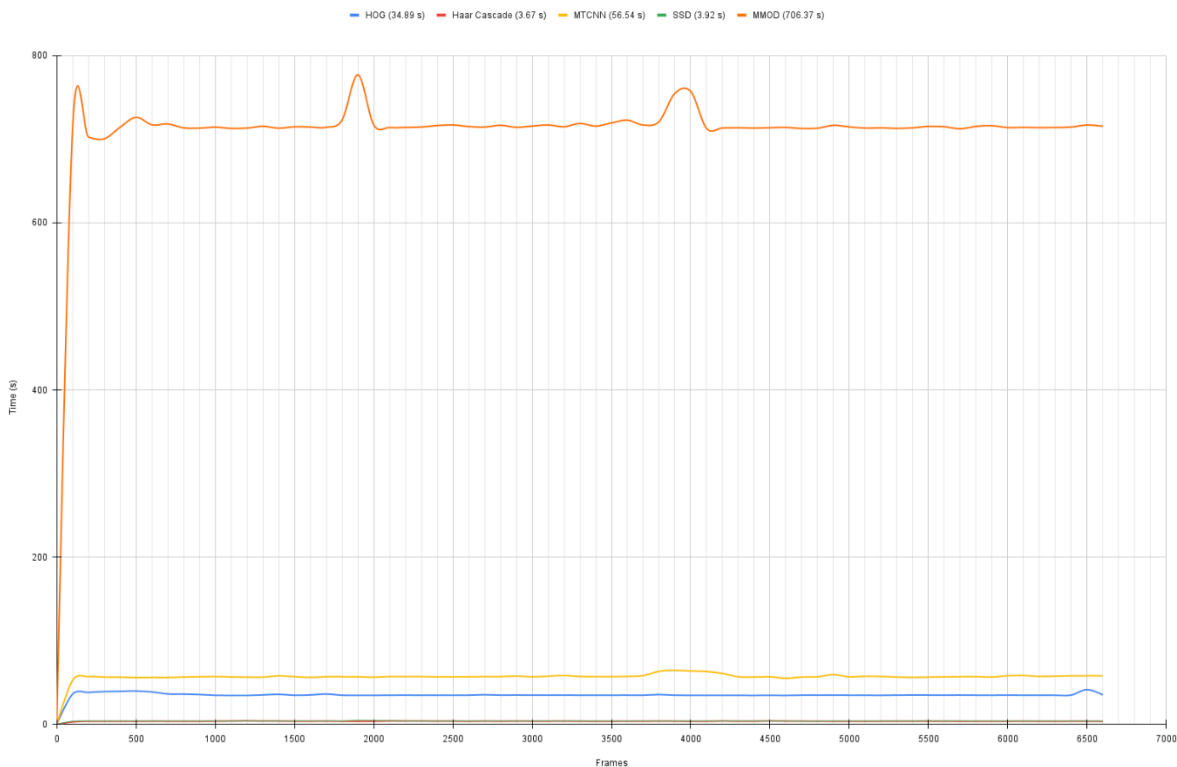

Figure 6. Chart representing the speed of detection of different face detectors. The values are taken from Table 4.

Accuracy

The accuracy experiments were conducted on the FDDB (Jain, 2010) dataset. In descending order of accuracy:

1. MTCNN – ~90% (Wang, 2018)

2. SSD – ~88% (Granger, 2017)

3. MMOD – ~70% (Cheney, 2015)

4. HOG – ~60% (Cheney, 2015)

5. Haar Cascade – ~50% (Pattarapongsin, 2020)

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 the Tables section in Table 1 and Table 2 respectively.

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 12. It also uses the most resources and would not be feasible in a commercial application – 3 GB of RAM.

SSD seems to be 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 relatively 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 accuracy.

Procedures & Error Prevention

In order 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 uses a considerable amount of traffic. In addition, the verification server might get overloaded and crash during sessions with many students. In order 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.

1.3 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 number of the student the verification result belongs to 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. 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 then cleans the directory so that up to nine last images and the base image remain. 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 done by feeding the image to the VGG-Face based CNN (more on this in Chapter 2) to receive a vector representation of the face. The model itself is state-of-the-art and produces results with 98.95% accuracy when tested on LFW (Labeled Faces in the Wild) dataset. (Parkhi, 2015) 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. (Parkhi, 2015) However, 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 in order 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:

$$ED = \sum_{i=0}^{n} \sqrt{(x_i - y_i)^2},$$ (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.
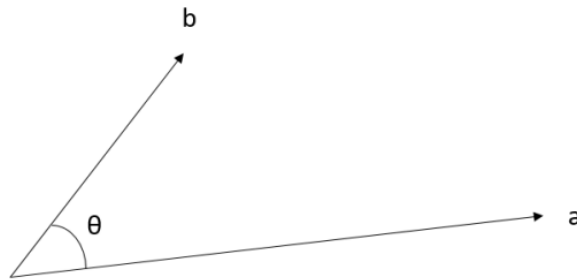
Figure 7. Vectors a and b.

Let vector c be equal to $a - b$. Vectors a, b and c create a valid triangle.
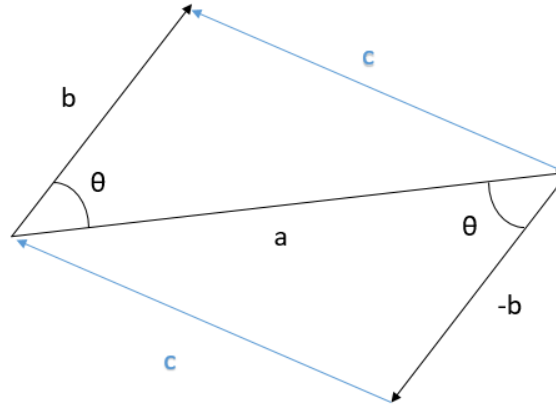
Figure 8. Vectors a, b and c.

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,

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

$b \cdot a$ \hfill (1)

$$\|c\|^2 = \|a\|^2 + \|b\|^2 - 2a \cdot b \tag{2}$$

$$\|c\|^2 = \|a\|^2 + \|b\|^2 - 2\|a\|\|b\|cos\theta = \|a\|^2 + \|b\|^2 - 2a \cdot b \tag{3}$$

$$\|a\|\|b\|cos\theta = a \cdot b \tag{4}$$

$$a_1b_1 + a_2b_2 + \cdots + a_nb_n = \|a\|\|b\|cos\theta \tag{5}$$

$$cos\theta = \frac{a_1b_1 + a_2b_2 + \cdots + a_nb_n}{\|a\|\|b\|} \tag{6}$$

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

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

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

$$cosine\ similarity = \frac{a_1 b_1 + a_2 b_2 + \cdots + a_n b_n}{\sum_{i=1}^{n} \sqrt{a_i^2} \ \sum_{i=1}^{n} \sqrt{b_i^2}} = \frac{a^T b}{\sqrt{a^T a}\sqrt{b^T b}}$$

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

$$cosine\ distance = 1 - cosine\ similarity$$

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:

$$verdict = cosine\ distance < threshold$$

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.

References

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.

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.

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.

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.

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.

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

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.

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.

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.

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.

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).

Jain, V., & Learned-Miller, E. (2010). Fddb: A benchmark for face detection in unconstrained settings (Vol. 2, No. 6). UMass Amherst technical report.

Parkhi, O. M., Vedaldi, A., & Zisserman, A. (2015). Deep Face Recognition. British Machine Vision Conference.

Tables

Table 1

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 (%)	0.52	27.63	12.39	28.05	12.48
Difference (%)	6.96.	16.08	9.92	16.57	10.27

Table 2

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

Table 3

Detector Speed Comparison

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

Table 4

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

Frames	HOG (34.89 s)	Haar Cascade (3.67 s)	MTCNN (56.54 s)	SSD (3.92 s)	MMOD (706.37 s)
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
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

Frames	HOG (34.89 s)	Haar Cascade (3.67 s)	MTCNN (56.54 s)	SSD (3.92 s)	MMOD (706.37 s)
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

Frames	HOG (34.89 s)	Haar Cascade (3.67 s)	MTCNN (56.54 s)	SSD (3.92 s)	MMOD (706.37 s)
6100	34.81871414	3.716593266	58.40186977	3.925348997	714.1170731
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