[[1]](#footnote-1)

Facial Recognition as First Factor for Web Authentication

Sebastian M Makinson

*Abstract*—Using Facial Recognition is it possible to build a good enough authentication system using the user’s face to identify them and a password to verify them? This experiment aims to see the feasibility of not just doing this, but doing it with off-the-shelf models, frameworks, and components.

**TERMS**:

*Off-The-Shelf:* Any components of code that don’t require much, if any, custom code to make it work.

*Web-Based:* A piece of software delivered through the web browser.

# INTRODUCTION

I

DENTIFYING users in a web based system has been a problem since the beginning of the internet. Modern techniques using Machine Learning and recent advancements in web-based machine learning libraries such as tensorflow.js have allowed us to bring modern biometrics to the browser. I am interested in the feasibility of training a model against a known dataset to detect faces and identify them and using that same model in the web browser to perform the identification portion of an authentication system for a website in place of a username. With the prevalence of webcams on nearly all devices now days facial recognition for authentication is more feasible than ever. Using off-the-shelf frameworks, models, and components we should be able to build a strong, and efficient video based facial detection authentication system for any website. This paper will explore the feasibility and pitfalls in finding a series of off-the-shelf components to build a web-based biometric authentication system.

# Previous Work

This experiment relies on the previous work done by the teams at Google who built TensorFlow [1] and the community who has been using it to optimize models for the browser. I built this platform on top of the face-api.js [2] framework. This framework is built on top of Tensorflow.js and uses models trained and designed for that platform. The three models used for this project were MTCNN, Tiny Face Detector, and SSD MobileNet V1. I ended up using pre-trained models and instead focused my attention on evaluating the model’s efficiency for both speed and file size of the resulting model. This project also relies heavily on my prior experience with building web applications and authentication systems.

# Experimental Design

The experiment here is to answer two questions: 1) Is it possible to rely on face detection as a first factor for authentication? 2) Is it possible to do some or all of the work in the browser?

In order to answer these questions, I planned to build a basic web system that first allowed a user to register for the site. This requires the user to give us their face and train it against my model. I will capture their face using facial detection in the browser and send that data to the backend where it will be used to update the model. The user can then sign into the system. In order to sign in they will be required to allow their webcam to capture their face and enter their password. Throughout the process of signing in the user’s image will be captured and sent to the backend for matching. The password and matching username will be used to determine if the user is allowed to sign in or not.

A screenshot of a cell phone

Description automatically generated

[FIG 1. Design Flowchart of the interaction]

The application is split into two pieces: Frontend and Backend. These are designated as *face-detection-frontend* and *face-detection-backend*.

**Face-detection-frontend** is designed using only two pages. Page 1 is the sign-in Page where I do the face detection to find the user’s face and match to a username. Page 2 is the Registration page where I find the user’s face to train the model for that user. The frontend is built using *React.JS* [3]a library from Facebook used for building UIs. I am also using *face-api.js* [2] as the face detection library. Face-api.js is built on top of Tensorflow.js and uses models designed to work with Tensorflow.js [1]. For the sign-in page I constantly capture the user’s image and send the sub image detected as the face to the backend. I crop the image captured by the webcam to the area detected as the face.

In order to detect faces efficiently I needed an algorithm that would have a good tradeoff of speed and accuracy. The tradeoff is more important that a traditional face detection algorithm because I am doing the matching in the web browser which has limited resources. For this problem I was able to find a couple of pre-trained models that worked well with face-api.js and would work in the browser. I evaluated Three different models:

**MTCNN**: First was MTCNN (Multi-task Cascaded Convolutional Neural Networks) [4] which is a 3-stage Cascaded Convolutional Neural Network. This allows it to have a great accuracy with a wide variety of lighting and alignments.

**Tiny Face Detector**: Tiny Face Detector is based off the Tiny Yolo v2 model which is itself based on the Yolo [5] [6] object detection method. The *Tiny Face Detec*tor replaces the regular convolutions of Yolo with depthwise separable convolutions making the model smaller for browsers.

**SSD MobileNet V1**: SSD MobileNetV1 is a SSD (Single Shot Multibox Detector) [7] based on MobileNet v1 [8]. The focus on this model is efficiently obtaining bounding boxes for the faces.

My methodology for testing these different models was to test them in the real world scenario. So I implemented the sign-in page and built it so I could easily swap out the face detection models. The results are as follows after 160 detections:

|  |  |  |  |
| --- | --- | --- | --- |
| Model | Average Match Time | Average Score | Model Size |
| MTCNN | 411.44ms | 0.9978 | 6400kb |
| Tiny | 61.63ms | 0.82 | 5600kb |
| SSD | 106.70ms | 0.75 | 160kb |

[Fig 2. Average Match and Score after 160 detections]

Using the results above I then calculated a score of the effectiveness of the model for my use case. The equation to calculate that is:

mintime = Minimum Average Match Time (Tiny)

maxscore = Maximum Average Score

I also calculate a weighted score where I downgrade the score based on the size of the model:

This gives us the following scores:

|  |  |  |
| --- | --- | --- |
| Model | Score | Weighted Score |
| MTCNN | 0.149 | 0.004 |
| Tiny | 1.216 | 1.216 |
| SSD | 0.766 | 0.022 |

[Fig 3. Scores and Weighted Scores based on above algorithm]

Based on these results the clear winner was Tiny Face Detector. This gave me a clear advantage in the speed in which I could detect the faces being almost twice as fast as any of the other models. It was middle of the road when it came to accuracy but was nearly 40 times smaller output model compared to the largest which was MTCNN. This meant that the model could load faster and provide a better user experience for the end user.

Now that I was able to extract faces using my frontend models, I am able to then do the matching. I take the bounding box that is determined by Tiny and cut out the face from the webcam image. I use the HTML5 Canvas API to copy the byte data from the current video frame into a canvas, I then take that canvas and crop it to the bounding box that is returned from the model. I then export the image as a Base64 encoded string of a JPEG. I also allow the browser to lower the quality of the image to 50% to save on bandwidth. There was also no noticeable difference in the results between 100% and 50% image quality.

Once I extract the faces and send the image to the backend I wait for a response with a list of potential matches and their scores. After I receive the list of potential matches, I pick the one with the highest score, keep track of that score, and save the username. If one of the matches is a lower score than a previously saved match, then I throw it away. This means that the highest match is only ever used as the user’s username.

**Frontend Registration:** One of the major design decisions that had to be made was how to register the user. For the purposes of this experiment I decided to keep the solution as simple as possible. This involved the following process:

First, I ask the user their username. This is sent to the backend to create an entry for that user. It also clears out any existing training data for that user.

Second, I do a little training game to train the user’s model. Because the YOLO algorithm and by extension the Tiny algorithm works best when there is a good variety of training data, I decided to create a small game to allow the user to train their face at different angles.

**A group of people posing for a photo

Description automatically generated**

[Fig 4. Face Detection Grid from Registration]

The game has the user look at a series of dots as they move around the screen. I then capture the face at 9 different angles which gives me a good variety of faces to work with. An area of further study on this would be to figure out the optimal number of angles to capture during authentication and how the change in number of captures effects the resulting model. It would also be interesting to study if the different angles make any difference in the results, or if I could use this angle technique to infer 3D capture data [9]. It could be useful to generate a 3D model of the face using this technique to create a depth map that I could train my model against.

Lastly, I ask the user for their password. Once they give me their password, I finalize the training set and re-train my authentication model using the nine images that were captured.

**Face-detection-Backend:** The backend for the face recognition is a simple NodeJS server using Tensorflow.js and face-api.js to train and run my models.

The backend is split into three layers.

First, I have the API layer. This layer is responsible for communicating with the frontend and handling the WebSocket connections.

Next, I have the Training and Detection Layer. This layer is responsible for doing the model training and matching of faces passed from the backend.

Last, I have the Data Layer. This layer is the database that holds the Images used for training, the models that are used to extract faces, and the models that are trained from the user’s images. The image database holds the 9 images that are captured during registration. I store those images with the key *username-index.jpeg*. Once the user registers and gives me their password I load the 9 images and do the feature extraction and landmark detection. Because I am doing this on my own servers I am not as concerned with speed or model size. But because of the efficiency trade-offs that are made by Tiny Face Detector and the fairly good detection accuracy I decided to use that one on the backend as well.

A screenshot of a cell phone

Description automatically generated

[Fig 6. Backend Layered Architecture]

Further areas for exploration are into other models that can give me a better accuracy and fairly good speed. This will help us increase the security of the system. I could even use the images that my users give me to train a new model that can be used. If I am able to in the future it would be nice to create 3D models from the user’s face on the frontend and send that 3D data to the backend where I can match it using a 3D face model for the matching. [10] [11] [12]

The API is responsible for maintaining the WebSocket connection with the backend. The API has 4 possible methods: *train, register, auth-attempt, match.* They have the following responsibilities:

**Train:** Train takes in the image data from the frontend. It saves the image to the image database and starts a background job to re-train the user model. The re-training happens based on a queue as to not hold up the frontend. The training happens in the Training and Detection Layer.

A picture containing person, indoor, man, wall

Description automatically generated

[Fig 7. Registration Game]

**Register:** Register takes in the user’s password and username and creates their entry in the database.

**Auth-Attempt:** Auth-Attempt looks up the user by their username that was matched by the matching model and the password. In a full-fledged system this would handle actually authenticating the user.

**Match:** Match is the method that receives image data from the frontend and attempts to return a matching username to the frontend. The Match API method passes the image data along to the Training and Detection layer which is responsible for doing the actual matching.

The Training and Detection Layer is responsible for all the matching and training that happens with the backend. This layer is separated as to avoid deep dependency with the frontend. This way the API layer can transform its data and interact with the Training and Detection layer in a predictable and consistent way.

The Data Layer is one that would need to change as this experiment were implemented and scaled. The image Database would need to be changed from what it currently does, which is store the images in the filesystem to storing them in a database that is more easily searched and indexed. The detection models are the same models that are used on the frontend (SSD, MTCNN, and Tiny) but are not limited to those. I could train new models to take advantage of the increased power of a server as opposed to the browser. This would be an area for further exploration.

In the real-world when testing this framework, I found that with only 1 or 2 users in the database there was a very high rate of False Positives. This was happening because every user was being fit into 1 of the only a couple users in the system. To mitigate this, I used the FERET [13] dataset to add a large number of users to the system. There were a little over 1200 distinct individuals in the FERET dataset. Using these faces I was able to load the large number of users into my system which allowed me to get more accurate.

The process for converting the Color FERET dataset to something usable and importing them into my model went as follows:

First, I had to unzip all of the files using the *bunzip2* tool included with the dataset. This left us with 11338 files in .ppm format. My Nodejs application required my files to be in .jpg format and saved with the name *username-index.jpg*. So, I build a script that copied the all of the files from the FERET dataset using the *convert*imagemagick tool using the command line. This allowed me to copy them in .ppm format and output them in .jpg format. During the copy process I generate a new unique username for every image in the dataset. The files are saved in a seed directory with the new username and index.

Next, after copying them into the correct directory I run a script that does the model creation by computing the face descriptors for every image. It then puts these into a database that is used to bootstrap the model when the server starts up. This is the same database that is used to generate new models when a new user register.

After doing this seeding of the FERET users I went from a high FMR to a much more reasonable FMR. Something that would be nice to do with more time would be to do actual quantification of the FMR and FRR to determine the effectiveness of the matches.

# Results

After all this work I think we were able to create a unique and interesting technical demo by using facial recognition as the first factor for authentication into a web-based system. There were quite a few challenges getting this to work. I was required to research quite a few different techniques and existing models in order to implement this. There was quite a learning curve to learn how to do TensorFlow within the browser and get models to work efficiently.

This technique proved very effective to offload some of the heavy lifting of detecting the faces and cropping the main image on the browser side. This allowed the backend to focus on things that were more important like training the model and matching the faces. There is still a lot of work to do regarding benchmarking and checking the efficiency of the algorithms. It would also be helpful to check the efficiency of the sending images to the backend at scale. My biggest concern is the amount of bandwidth required for this authentication method.

Another interesting future enhancement could be to use fingerprint instead of password as the second factor.

A screen shot of a person

Description automatically generated

[Fig 8. Authentication Page with debug mode]

One interesting factor of this experiment was noticing that it did not require a nearly as accurate model when using a quicker model. This showed us that we could make a good tradeoff between speed and accuracy. It would be very interesting to explore more the actual difference between different speeds and accuracies and how they effected the FMR and FRR of the authentication system.

# Conclusion

In Conclusion, while this was an interesting experiment there is nothing new that hasn’t been done. This is a huge testament to the massive amount of work done by the industry and researchers in this subject. It is now at the point where complex and efficient authentication systems can be built using biometrics with off-the-shelf models, frameworks, and data. We are able to use tools like Tensorflow.js and pre-trained models done by the community to achieve fairly good facial detection at around 15 matches per second in the browser.

With more time to research this topic there is no reason you couldn’t enhance this experiment with techniques such as 3D or 2.5D matching or building custom models to better enhance the speed and accuracy of this technique.

# References

|  |  |
| --- | --- |
| [1] | Google Inc, "Tensorflow," Google, [Online]. Available: https://www.tensorflow.org. [Accessed August 2019]. |
| [2] | V. Mühler, "Github," [Online]. Available: https://github.com/justadudewhohacks/face-api.js?files=1. [Accessed August 2019]. |
| [3] | Facebook, "React.JS," [Online]. Available: https://reactjs.org/. [Accessed August 2019]. |
| [4] | K. Zhang, Z. Zhang, Z. Li and Y. Qiao, "Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks," *IEEE,* vol. 23, no. 10, pp. 1499 - 1503, 2016. |
| [5] | J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," *CoRR,* vol. abs/1506.02640, 2015. |
| [6] | J. Redmon and A. Farhadi, "YOLO: Real-Time Object Detection," [Online]. Available: https://pjreddie.com/darknet/yolov2/. |
| [7] | W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu and A. C. Berg, "SSD: Single Shot MultiBox Detector," *CoRR,* vol. abs/1512.02325, 2015. |
| [8] | A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *CoRR,* vol. abs/1704.04861, 2017. |
| [9] | T. Hara, H. Kubo, A. Maejima and S. Morishima, "Fast-accurate 3D face model generation using a single video camera," in *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*, Tsukuba, Japan, 2012. |
| [10] | S. Berrettia, N. Werghib, A. d. Bimboa and P. Palaa, "Matching 3D face scans using interest points and local histogram descriptors," *Computers & Graphics,* vol. 37, no. 5, pp. 509-525, 2013. |
| [11] | X. Lu, A. Jain and D. Colbry, "Matching 2.5D face scans to 3D models," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 28, no. 1, pp. 31-43, 2005. |
| [12] | N. Werghi, S. Berretti, A. D. Bimbo and P. Pala, "Local descriptors matching for 3D face recognition," in *2013 IEEE International Conference on Image Processing*, Melbourne, VIC, Australia, 2013. |
| [13] | NIST, "color FERET Database," [Online]. Available: https://www.nist.gov/itl/iad/image-group/color-feret-database. [Accessed Aug 2019]. |

1. [↑](#footnote-ref-1)