CSE 546 — Project Report

Group ID — 10

Project Title — Face Mask Detection

Anirudh Krishna Lakshmanan, Gayathri Alloju, Prateek Srivastava

1. Introduction

One of the major impacts of Covid-19 is felt by businesses and organizations. Even though safety protocols are in place, people are often found to be flouting these norms. One of the common scenarios is a person not wearing his or her face mask properly while entering the premises of restaurants, sports complexes, stores, offices, libraries, etc. While it may be easier to keep a check on individuals when the footfall is low, it can turn into a tedious and inefficient process once the footfall increases.

For example, a sports complex hosting a football match is expected to have spectators entering the stadium in huge numbers and from multiple entry gates. In such a scenario, human monitoring of the crowd is likely to make errors while letting people inside. The problem only amplifies once the person is granted entry for wearing the mask correctly, but later decides to pull it down. In such a case, human monitoring of the entire crowd seems almost impossible.

The problem of monitoring people who are not wearing their masks correctly in a public place is critical to ensure the safety of other individuals who can come in contact with them. The problem needs to be dealt with by businesses and organizations to ensure safety and restore the confidence of the people in them. Once a person feels confident that Covid-19 guidelines are properly implemented at a place, he or she will be willing to go to that place.

2. Background

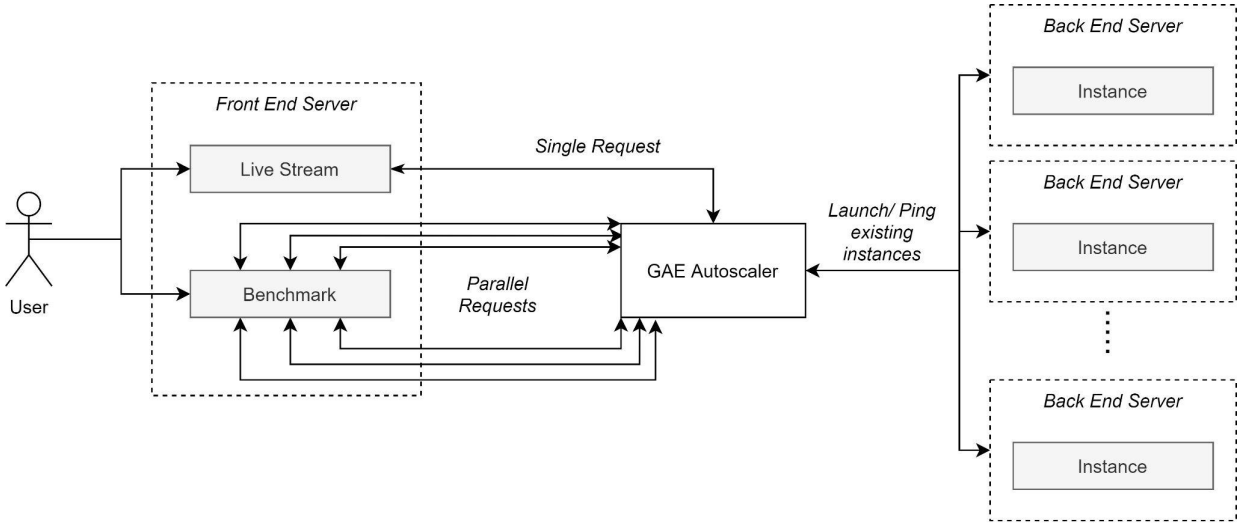
Cloud computing is one of the most relevant parts of the problem. The processing power required to accurately determine whether a person is wearing a mask is not small. This can make low-powered hardware ineffective when solving this problem. With the limited resources of camera hardware, cognitive computation needs to be offset to remote systems, which enables the use of cloud technologies. Auto-scaling ensures that the user can request resources based on the number of active cameras for the task. Hence, cloud computing ensures the solution is able to successfully return the results in a reasonable time, hence making the product useful for businesses.

This problem is important because, without a solution to detect face masks on a large scale, businesses that cater to a large number of people daily cannot follow Covid-19 safety protocols effectively, which can lead to strict actions against them by authorities, low footfall as people are skeptical about their safety, leading to a loss in revenue. Furthermore, it frees up manpower that is allocated to ensure the enforcement of the rules to do other tasks, thereby improving the customer experience.

There are existing solutions that help in the detection of face masks, however, there is no cloud solution that does this. These solutions are trained on the widely available face mask dataset[1]. This means that the existing solutions might be effective in a situation when the number of people coming in is few, but the existing solutions might fail to be effective when the number of people coming in increases, like during an event, or peak office/library hours. This calls for an effective solution that not only helps detect facial masks correctly but also scales up when the number of requests goes up.

3. Design and Implementation

A. Architecture



The image above shows the architecture of the proposed system

The solution consists of two levels of app tiers, one handling a static front end which is in charge of generating requests to a dynamic backend. The front end serves a web application that communicates with the backend using AJAX calls in an asynchronous fashion to avoid any bottlenecks. The front end consists of a live stream page that can mark the detection results in real-time on a live video feed from a webcam connected to the user's system. The other page is a benchmarking utility where the users can upload videos to stress test the application. The application parallelly processes all videos.

The backend consists of a flask server that loads the model and generates predictions for each of the requests coming through. The results are returned as a JSON object which contains the location of each detection as well as the confidence and class of that detection.

B. Cloud Services

The app uses the google app engine, which is a platform as a service (PaaS). Since the app is stateless, there is no need for other solutions which can introduce bottlenecks. This also serves to protect the client's privacy since no data regarding the videos/ streams are stored.

C. Role of GAE

The application consists of two types of servers - The web server that hosts the website for the user to interact with the application, and the application server that contains the deep learning model to detect face masks. Google App Engine provides an easy way to deploy the application on two different servers. The web server is deployed as a separate project, and the application server is deployed as another project.

Google App Engine makes it possible to separate the servers for decoupling and makes it easier to deploy only that server that has changes to be deployed. It also makes it easier for a person wanting to use only the face mask detection model to make an API call to the application server, without having to bother about the webserver.

The most important advantage that Google App Engine provides is autoscaling. Google App Engine saves the hassle of manually controlling the autoscaling, and does scaling in and out on its own as the number of requests goes up. This helps in running the application without any failure and delay, as Google App

Engine scales up and down based on the number of concurrent requests that are sent from the webserver.

D. Autoscaling

The main bottleneck is the machine learning model, which is a faster RCNN mobilenet[2] trained on the Face mask dataset. This can take about half a second to generate a prediction. A single instance of this resource would be ineffective in a real-world scenario where there are multiple streams in parallel requesting the use of the model. Autoscaling solves this problem by automatically scaling the number of model instances based on the load present. This can help users to request more resources during peak sessions while releasing resources in idle sessions. Autoscaling simplifies the logic required to estimate these parameters. Furthermore, it is not necessary for one instance to be allocated to one request source since an instance can handle multiple requests before they time out. This allows for more cost-effective scaling.

While a number of solutions already exist for detecting facial masks on people, the solution proposed in this project uses the power of cloud computing to scale the number of application servers as the number of facial mask recognition requests increase. This ensures that the application can work without any bottlenecks on the number of requests, thus enabling a person using this application to run the face mask detection model parallelly on a number of images. Also, the application scales down, i.e, the number of application server instances goes down once the number of requests goes down, thus preventing the person using this application from incurring charges for resources that he or she is not using.

4. Testing and evaluation

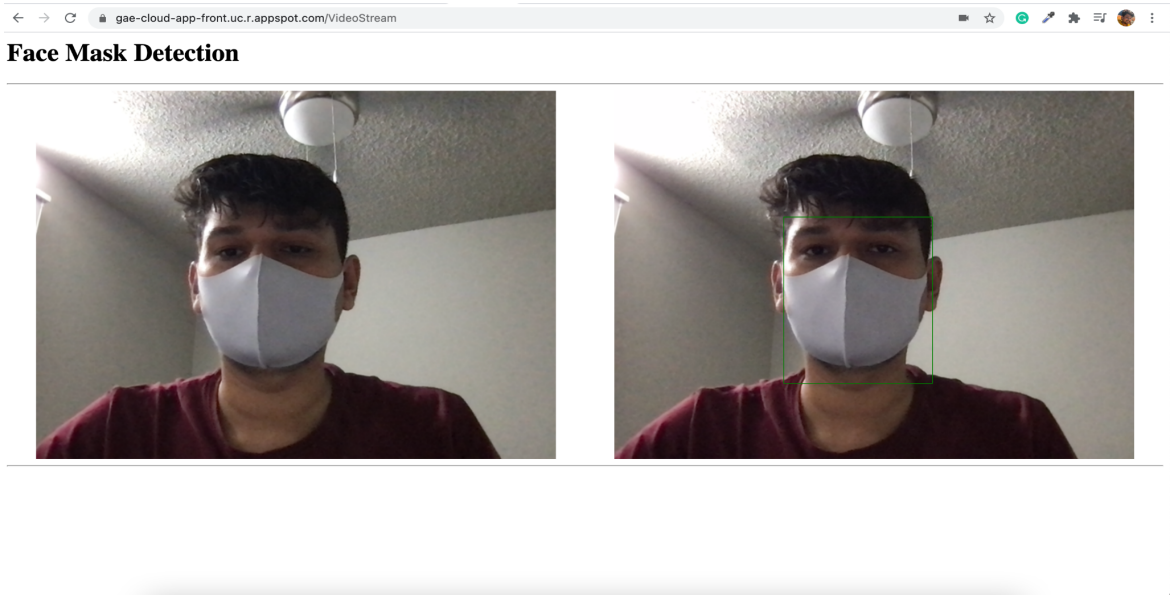
The testing was carried out in three key areas mentioned below-

A. Testing the website

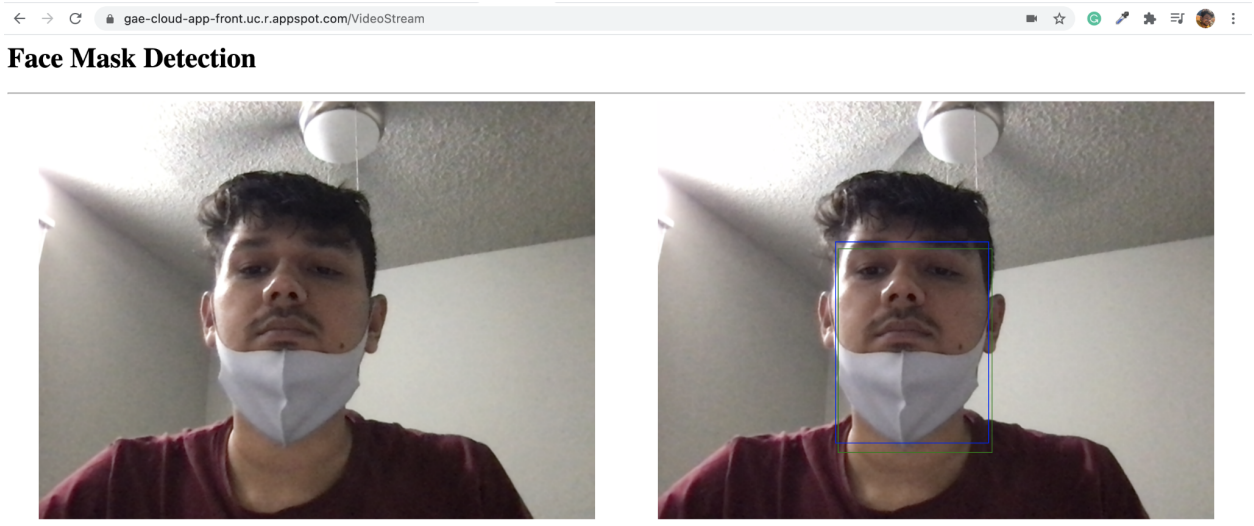
Functional testing of the website was done by deploying the website and ensuring that the landing page showed up correctly, the buttons for the live stream, and for the load testing were functional and were routed to the correct pages. For the landing page, testing was done to ensure that the webcam was able to stream the video properly on different devices and browsers. For the webpage that supports load testing, a number of videos were uploaded ranging from 5 to 20, and testing was done to ensure that the webpage does not timeout or crash. Additionally, UI testing was carried out to validate that images were rendered for their corresponding videos, and the loading gif was displayed then the request was being processed. Also, once the results were returned, testing was done to ensure that facial recognition boxes were drawn correctly on the image.

B. Testing the correctness of the deep learning model

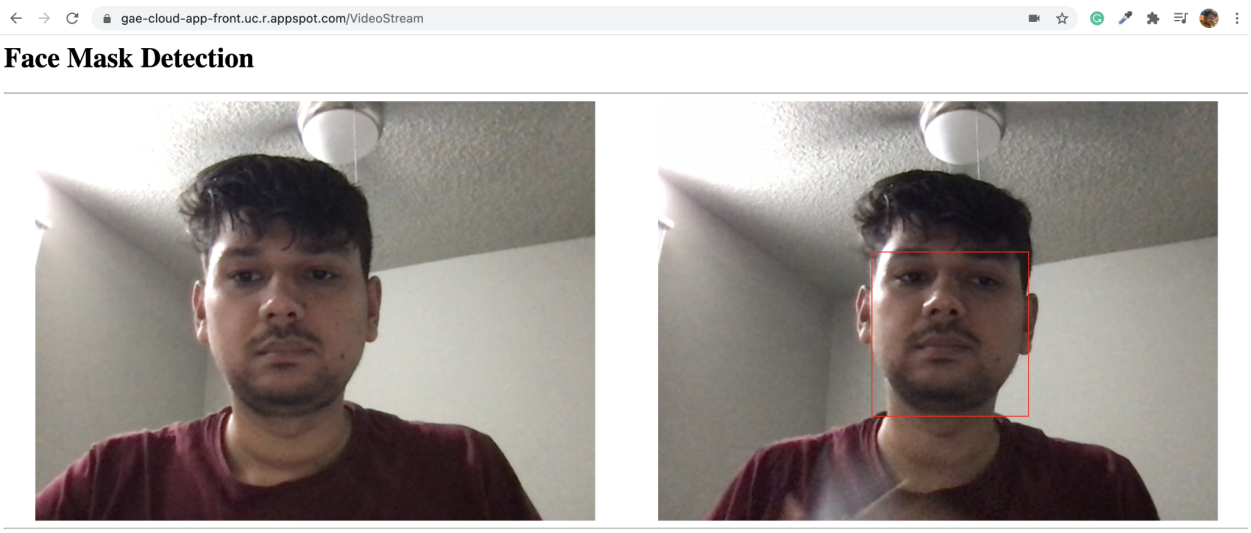
The deep learning model used to detect the correctness of facial masks is supposed to differentiate between a person wearing the mask correctly and a person not wearing the mask correctly or not wearing it at all. Testing was done on a number of videos, consisting of different people, with one or multiple people in some of the videos, to make sure that the results were correct. The deep learning model was found to be largely effective in delivering the results correctly, in all the above-mentioned scenarios. There was no impact on the correctness based on the resolution of the video and the lighting in the background. Furthermore, results with low confidence are filtered out. This typically corresponds to results with less than 50% score.



The above image shows the live stream returning the result of the deep learning model in the form of a green box, which denotes that the person is wearing the mask correctly.



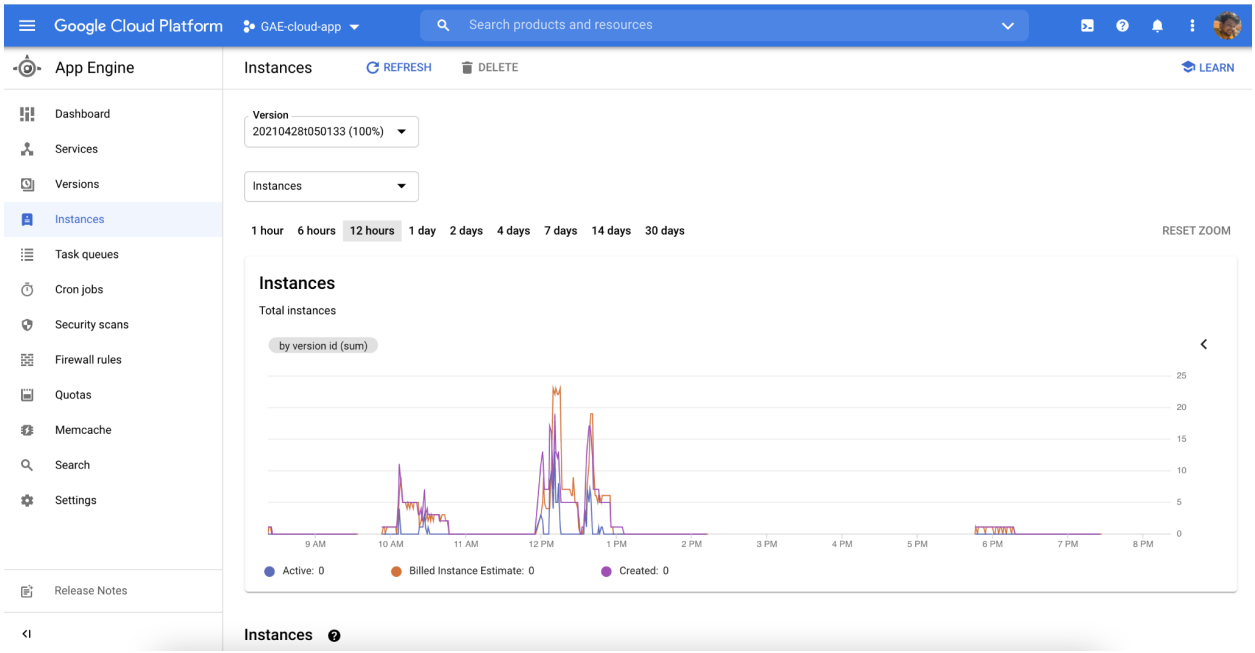
The image above shows that the live stream returns the result of the deep learning model in the form of a blue box, which denotes that the person is not wearing the mask in a correct manner.



The image above shows that the live stream returns the result of the deep learning model in the form of a red box, which denotes that the person is not wearing any mask at all.

C. Testing Auto-Scaling using Google App Engine

Using the multiple video upload functionality of the website, a number of videos were uploaded and their image snapshots were sent as data to the application server concurrently. Using the App Engine's dashboard, it was possible to confirm that when the number of requests went up to the application server, additional instances were spun up, and when the number of requests went down, the number of instances scaled-down.



The image above shows the auto-scaling done by Google App Engine, on the application server over a period of 12 hours. We can see that the number of instances goes up as a certain number of requests are sent to the server, and later goes down as the results are returned and no new requests are sent.

5. Code

Both the front end server and backend server can be easily deployed on GAE by cloning the code and running gcloud deploy command[3].
NOTE: The backend server uses F4_1G resources (due to memory requirements) which are billable.

A. The front end server

The front-end server is present in the 'web_server' folder. This contains a flask server that serves a webpage consisting of the live streaming and benchmarking options. The benchmark page creates asynchronous ajax requests in parallel for each video uploaded. The uploaded videos are played in a loop to mimic the continuous nature of a live streaming system. The results returned are drawn on top of the screenshot of the image taken and the box color indicates the label of prediction. The library requirements needed to run this program are included in the requirements.txt file.

B. The back end server

The backend server is present in the 'application_server' folder**.** This contains a flask server that processes API calls, which are in the format of HTML post requests. This route accepts an image file and returns the result in a JSON object format. The result contains predictions which include box locations, scores (confidence), and labels. The library requirements needed to run this program are included in the requirements.txt file.

C. Training the model

The python notebook is placed inside the 'face_detection_notebook' folder. The python notebook 'Face_Detection.ipynb' contains the resources to train the model. The code is written to be used in a local system but can easily be configured to run on colab by reintroducing the mount drive code block into the code. The model used is a faster rcnn mobilenet but this can be adjusted by changing the get_model function. Finally, the code saves the model at the end of each epoch and prints the loss. The code uses the pytorch vision library[4].

6. Conclusions

The project successfully detects a face mask on a person and determines if the mask is worn correctly. Using Google App Engine, the project is able to auto-scale based on the number of requests that are coming in from various live video streams (emulated using multiple video uploads in the project).

The current solution can be improved in the following key areas -

1. An image recognition model can be used at the webserver in order to filter out relevant images which need to be passed on to the application server. As of now, all the captured frames are sent to the application server, which means there will be an unnecessary load on the server even when there is no person present in front of the camera. Using solutions like OpenCV and HarrCascade, we can capture only those frames that have a person in them, and send that frame back to the application server. This can speed up the solution and reduce the load on the backend server thereby reducing the cost.
2. The deep learning model used to detect the face masks' correctness can be improved by training the model on more and varied datasets. Currently, a faster RCNN mobilenet model is used due to the memory limitations of a standard GAE server. However, upgrading to flex servers can allow the use of other models which have higher accuracy, such as a faster RCNN resnet. However, these may require a vGPU as well to enable real-time processing, which is not supported by GAE at the moment.
3. Instead of using videos to check auto-scaling, a better way to live stream multiple videos on devices can be found.

7. Individual contributions

Student name - Prateek Srivastava

Contribution to the design of the project

The project consists of two major parts, a webserver, and an application server. I contributed to the design of the webserver. The webserver hosts the website that takes a user to the dashboard of the application. The dashboard includes two key components. The first component is a live stream video feed to test the application as it would be presented to the user in real-time. The second component is used to test the auto-scaling of the application server by uploading multiple videos and confirming that the required number of app server instances are created and terminated as the number of requests from the webserver increases and decreases respectively.
The live stream is designed to allow live webcam video on the client-side, take image snapshots from the video, and send the data to the deep learning model running on the app server. Once the app server returns the results, the live stream uses the data returned from the app server to draw a red, blue, or green box on the face of the person to represent the correct results.
For the auto-scaling component, the design lets a user upload multiple videos concurrently. Image frames are captured from all of the videos parallelly. The image frames are then sent to the app server as data in parallel requests. This component is designed to emulate the live video stream running concurrently on multiple devices as multiple video uploads. Thus, auto-scaling can be tested for the application without actually running multiple live video streams.

Contribution to the implementation of the above design -

I implemented the website using Python 3.8.5 on the Flask web server framework. The web pages are implemented using HTML, CSS, JavaScript, and Jquery. The flask server renders the 'index.html' as the dashboard for the website. The dashboard consists of two HTML buttons, one for the live stream video feed and the other for the upload videos functionality to test the autoscaling of app servers when the number of requests increases or decreases from the webserver.
The live stream video feed is implemented using client-side technologies, which are HTML, CSS, JavaScript, and Jquery. A user's webcam is used to stream the video using HTML's 'getUserMedia' functionality. Once the video is streaming, an HTML 'canvas' is drawn over the video to capture an image frame. The image frame is converted into a blob object, which is then posted to the app server by making a JQuery AJAX request to the app server's cloud URL. The results are returned as a JSON object which contains the location of each detection as well as the confidence and class of that detection. Using this data, a colored box is drawn over the face of the person, where green indicates the correct usage of the face mask, blue indicates that the face mask is not worn properly, and red indicates that the person is not wearing a face mask.
For the auto-scaling component, an HTML form is used to upload the videos on the webserver, and these videos are displayed using HTML video frames. For each video frame, a dynamically generated HTML image tag is generated, which is used to display the corresponding images with their results.

Contribution to the testing of the project -

Testing was done from my end to ensure that the website's frontend functionalities were working as expected. Apart from this, testing of the live video stream was done in the following scenarios to check the correctness of the deep learning model- live video streaming of a person with a face mask worn properly, a person without a face mask, a person with a face mask but the mask not being worn correctly, a person with sunglasses and a mask, a person with a cap, sunglasses, and a mask. These test cases were carried out on different users on the live feed. Testing was also done for the multiple video uploads functionality to confirm that the website does not timeout when the webpage is loaded with multiple videos.

8. References

[1] Face mask detection dataset (Mask Dataset)
[2] Faster RCNN MobileNet paper (Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks)
[3] GAE Documentation (Quickstart for Python 3 in the App Engine Standard Environment)
[4] Pytorch library (Pytorch)