CSE 546 Real-Time Face Swapping Tool using Deepfake Algorithm

Aditya Krishna Sai Pulikonda
Kshitij Jayprakash Sutar
Yash Kotadia

1. Introduction

The aim of this project is to implement a cloud-based real-time face-swapping tool to swap faces in a video. Face swapping is a technique that replaces an original face with another. We use a machine learning algorithm to create a computer-generated version of the subject's face. We have used CNN auto-encoder based deepfake algorithm for face swapping. It involves a complex pipeline of multiple machine learning models namely face detection, landmark detection and autoencoder based swapping. Although the auto-encoder based deep fakes require higher computational power, the advantage of this approach is that the end results are much more realistic than traditional approaches.

Since auto-encoder based face swapping is a computationally expensive operation, one of our goals is to effectively divide computation across multiple instances to obtain a real-time face swapped output feed. Cloud Computing plays a key role in this system as obtaining real time output requires engineering the cloud infrastructure to minimize latency. This can be achieved using the standard cloud computing offerings like load balancing and auto scaling. While there are many people working on building accurate AI models, efficient deployment of these models on the cloud is an important aspect that is relatively less explored. We aim to address this issue by efficiently deploying the deepfake pipeline with an application interface on the cloud.

2. Background

Before the advent of neural network based approaches for face swapping, people have used traditional ways albeit the results were much more realistic with the former. These traditional techniques or methods belong to a larger and broader concept of ideas namely morphing. It is a special effect in motion pictures and animations that changes an image or shape into another through a seamless transition. Traditionally such depiction would be achieved through dissolving techniques on film which are commonly used in the post production process of film editing and video editing. Since the early 1990s, this has been replaced by computer software to create more realistic transitions and since 2018, when the concept of deep fakes was introduced, it is

used to trigger subliminal reactions by applying it to movie recordings that are served in popular online devices.

Long before digital morphing, several techniques namely Tabula Scalata, Mechanical Transformations, Matched Dissolves and animation were used for similar image transformations. Some of these techniques are closer to a matched dissolve - a gradual change between pictures without *warping* (digitally manipulating an image such that any shapes portrayed in the image have been significantly distorted) while others did change the shapes in between the start and end phases of the transformation. In the early 1990s computer techniques that often produced more convincing results began to be widely used. For example, one would morph one face into another by marking key points on the first face, such as the contour of the nose or location of an eye, and mark where these same points existed on the second face. The computer would then distort the first face to have the shape of the second face at the same time that it faded the two faces.

Some of the early examples of digital morphing are the 1986 movie The Golden Child where they implemented rather crude digital morphing effects from animal to human and back and 1988 movie Willow featured a more detailed digital morphing sequence with a person changing into different animals. A similar process was used a year later in the movie Indiana Jones and the Last Crusade. Both the effects were created using grid warping techniques. There have been several applications of morphing in the latter years such as using morphing as a transition technique between one scene and another in television shows, even if the contents of the two images are entirely unrelated and for creating convincing slow motion effects. While perhaps less obvious in the past, morphing is used heavily today and current morphing effects are most often designed to be seamless and invisible to the eye.

Recent advancements in the field of Deep Learning made researchers use various deep learning architectures for face swapping like Convolutional Neural Networks (CNN)s, Recurrent Neural Networks (RNN)s and Generative Adversarial Networks (GAN)s. Although these techniques require huge amounts of training data and take a lot of time to train, they have been proven to deliver highly accurate and realistic results than any of the traditional approaches and the changes made are almost invisible to the human eye. One such modern technique which relies on neural networks that was introduced in early 2018 is the so-called **deep fakes,** which actually morph a person's face to mimic someone else's features, although preserving the original facial expression. These neural network based approaches follow the same process and require three steps - **extraction**, **training** and **creation**.

The extraction process refers to the process of extracting all frames from the video clips or pictures, identifying the faces and aligning them. The alignment is critical, since the neural network that performs the face swap requires all faces to have the same size (usually 256 * 256 pixels) and features aligned. Detecting and aligning faces is a problem that is considered mostly solved, and is done by most applications very efficiently. Training is a technical term borrowed from Machine Learning and refers to the process which allows a neural network to convert a face into another. Although it takes several hours, the training phase needs to be done only

once. Once completed, the model can convert a face from person A into person B. Once the training is complete, the final step is to merge the converted face back into the original frame. This step is where most face-swap applications go wrong as it doesn't use any machine learning and the algorithm that stitches a face back onto an image is hard-coded, and lacks the flexibility of detecting mistakes.

The above paragraph gives a brief view of how any neural network approach more or less works. In this project, we used **CNN based autoencoders** which encode the original face to produce a lower dimensional representation of original face, sometimes referred to as base vector or latent face. When this latent face is passed through the decoder, it is reconstructed to output the swapped face. Autoencoders are lossy, hence the reconstructed face is unlikely to have the same level of detail that was originally present. The Deepfake algorithm depends on an Ad Hoc algorithm for **face detection**. We are using a Single Shot Multibox Detector(SSD) [8] based pretrained model that can adapt to different poses and is quick. For determining the facial landmarks, we have used the LBF Detector[7] which is a fast and robust model for detecting the facial landmarks.

While the deep fake algorithm does produce very real looking results, people have been skeptical about it since it could potentially malign anyone's reputation. However, it still does have many applications in the film and entertainment industry and these applications signify the importance of this problem. The fact that it takes a lot of computation power and training images still serve as a deterrent against its misuse. Our cloud computing backed tool shall only be used for making real-time inferences, the face-swapping algorithm shall have to be trained before using the tool for making inferences. Thus it still does not make the misuse of deep fakes any easier while serving as a useful tool for many applications. For example, currently, the film industry uses expensive CGIs for scenes where it is not feasible to have the cast play the scene. This application could potentially help bring down the cost of such scenes. One such most popular example is replacing Paul Walker's brother's face by Paul Walker's face in the Fast and Furious 7 movie after the latter's sudden death in a car crash during shooting. Furthermore, there are people who unfortunately have their face disfigured due to accidents, this application along with a trained model could help them regain their confidence at least on online videos. We can also add our application as a plugin to the browsers when we make video calls or during video conferences. Clearly, deep fakes still do have a lot of advantages that could be leveraged using our proposed tool.

With our application, we intend to provide more transparency and better usability around the auto-encoder based deep fake algorithm by making use of the same algorithm, but not compete with it as such. As explained in the project proposal, our goal was to not compete and improve the results produced by deep fakes, but efficiently deploy the model on the cloud. This aspect can be achieved using Google App Engine (Standard and Flexible) and Google AI Platform. Current apps like Doublicat only support face-swapping for gifs and images where a user uploads an image and chooses another face that they want to replace with. Our idea is to achieve real time face-swapping or produce a live feed of the user by swapping his or her face. The

computational power required for few images is significantly less than what is required for a video stream as for the former, the computation can be done on the client side which is not possible for videos. Additionally, video streams have latency constraints that will require additional optimizations to the infrastructure. Real time inferencing can only be achieved when we are doing parallel processing that reduces the latency and can be achieved using standard cloud computing offerings like Auto Scaling and Load Balancing. These features along with the PaaS component of Google Cloud Platform (GCP) and Google AI Platform for deployment are used in our project to achieve our desired goal of doing real time face swapping using deep fakes.

3. Design and Implementation

3.1 System Design

The Deepfake Face swapping application is built on 3 important components, i.e. Client Application, Websocket Server and CNN Autoencoder based on Deepfake algorithm. The Client application is built around a Django framework which is used to send frames and receive frames from the server. We implemented a websocket server which provides a full-duplex communication channel from the client application. The main backend server for our project is the CNN Autoencoder which is responsible for reading these frames sent from the client and converting it into swapped face frames which would be returned to the client through the websocket.These 3 components are deployed with the help of different Google Cloud Platform(GCP) services which provides functionalities such as Autoscaling based on load, Real-time processing, and also Build and run the Deepfake model.

The GCP services utilized for our project are:

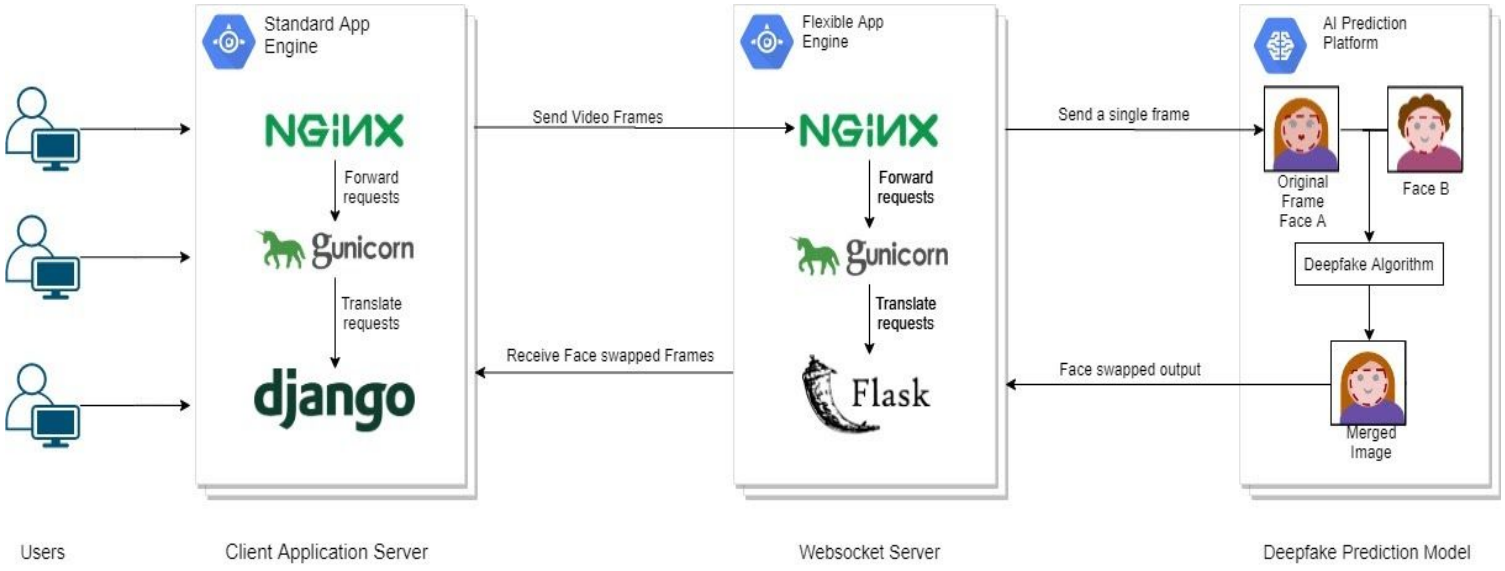
1) Google App Engine(GAE)

   We have used the GAE's PaaS service to deploy our client application and websocket.But, the type of app engine is different for both components, the client application is deployed on the Standard App Engine environment whereas Websocket is deployed on the Flexible App Engine environment.

2) Google AI Platform

   The Deepfake algorithm requires a lot of training data and then building this model is quite expensive in terms of computation resources. AI platform provides the perfect solution to host such a trained model on the cloud, and then use this model to form face swapped frames from the new image data

3.2 System Architecture Diagram



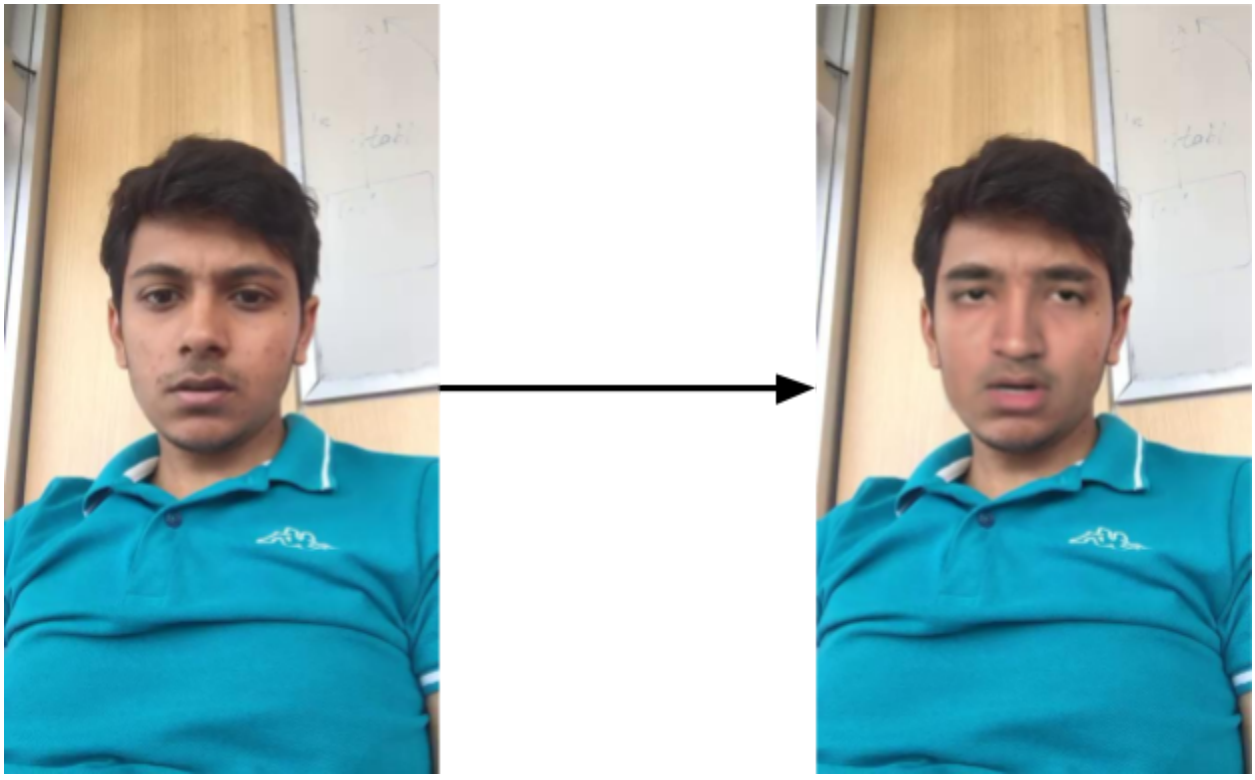
As shown in the above architecture diagram we are using a 3-tier architecture for this project. The client is hosted in a 'Standard App Engine' environment. It is a web application and we have used Django to build this component. The second component is the WebSocket Server which is hosted in the Flexible App Engine environment and as it is a background service we haven't added any UI component and integrated the Python code using Flask. The final component which is the Deepfake Prediction Model is deployed on the AI Prediction Platform which contains our pre-trained and it is used as an API to return the swapped face as output.

3.3 Implementation of Components with Cloud Services

The implementation of components with cloud services on the each of the 3 components is discussed in detail below:

1. Client Application Server

We decided to create a web application for the user to receive the Face swapped video. This web application is built on the Django Python framework. The user can see the original face video preview on the top right side and after starting the face swapped process they can see the video stream of the swapped face on the bottom right side of the screen.



Example of an image where our model swapped one face with the other

The application is hosted using the Google App Engine's(GAE) standard environment. This application is used to send and receive image frames which have been recorded by the webcam.

We have used the standard environment for our client application because the deployment was this type of instance that happens rapidly and the VM instance comes up in seconds in case of autoscaling. The integration of GAE with our current Django application was also seamless. The steps involved in integrating app engine instance with Django application are:

- Creating a project on Google Cloud Console and enabling APIs.
- Creating app.yaml configuration file which contains the processes to run in order to deploy our application.
- Running the command 'python manage.py collectstatic' to Gather all the static content into one folder by moving all of the app's static files into the folder specified by STATIC_ROOT.
- Finally, executing 'gcloud app deploy' to deploy this Django application on the app engine server.

2. Websocket Server

We have implemented a websocket server which listens for requests from the client and forwards it to the backend server, and also reverse i.e from backend server to the client.
The data which is transferred from the client are images encoded in the form of string.

Also, the data from the backend server is in the form of images encoded in string.

We chose to use websockets over HTTP connection as websocket provides a fully-duplex communication whereas HTTP provides half duplex i.e. the sender and receiver can send data and neither of them have to wait for response before sending another packet. This ensures a very low latency compared with HTTP overhead.

This websocket server is also deployed on Google App Engine(GAE) but here we use the flexible environment of the app engine instead of the standard one.

We have used the flexible app engine environment over standard as this type of instance provides ability to handle consistent traffic and support for background tasks such as websockets. The flexible environment runs applications within a Docker container on Compute Engine VMs. As our application needs this component to handle a lot of image frames in the form of requests from both client and server, we have implemented an flexible environment app engine. The steps involved in integrating app engine with the websocket server are:

- Creating a project on Google Cloud Console and enabling APIs.
- Creating app.yaml configuration file which contains the processes to run in order to deploy our application.
- Finally, executing 'gcloud app deploy' to deploy this websocket server with the help of gunicorn which is a Python Web Server Gateway Interface HTTP server.

3. Prediction Model

The face prediction model is the main backend for this project which does the processing part for the input images from the client. This is performed using the Deepfake algorithm which helps with extracting the face from the image, trains the models on extracted faces and converts the images with this model.

As the models used for this purpose require high computational resources, for this project we are using the trained model which can be used to predict the swapped images. Then this model returns the result back to the websocket.

We have deployed this prediction model using the Custom Prediction routines provided by the Google AI Prediction Platform. This allowed us to write our own Python code as routine to tell the server what exactly to do on the input image. So this allowed us to get the input image and preprocess this image to detect the face and apply the face prediction model on the extracted face from the input image. The steps used to deploy this is given below:

- Create Prediction.py, which contains the methods predict and from_path, the predict function is used to write the custom routine required to do the processing of the image and the from_path function is used to load an instance of our prediction model.
- Use the command 'gcloud ai-platform models create Prediction' to create a model resource on the Google cloud.

Our project implements auto scaling feature by using the Google App Engine resources, in the case of Client application server, the standard environment is capable of scaling up and down according to the number of users accessing the application. The websocket is also able to scale but in a different way compared to the client. It has a flexible environment as there are consistent requests from the user so the scaling up and down is done gradually compared to client where it is done rapidly.

3.4 Benefits of Proposed Solution

The proposed solution works on building an application that can be used to video stream real-time face swapping using google cloud resources. The important aspect of our project is that a user can obtain a live feed of his face being swapped with another person rather than he/she providing an image and getting the swapped face. This aspect is achieved by using the Google Cloud Resources such as App Engine(standard and flexible) and Google AI Platform. The existing solutions provide these functionalities on images and gifs which might not suit few users who want video stream, so we have built a solution which could provide such functionality with the essence of real time.

4. Testing and Evaluation

As explained in section 3, the proposed architecture consists of 3 tiers. We shall first discuss the performance of each of the servers individually and then evaluate the performance of the entire system as a whole.

4.1) Application Tier

The application server provides an entry point for the client. This server serves the HTML page which acts as an interface to the application. It also provides the relevant scripts that are required to set up the websocket communication between the client and the websocket server. We evaluate the server based on the availability and ability to scale to multiple clients.

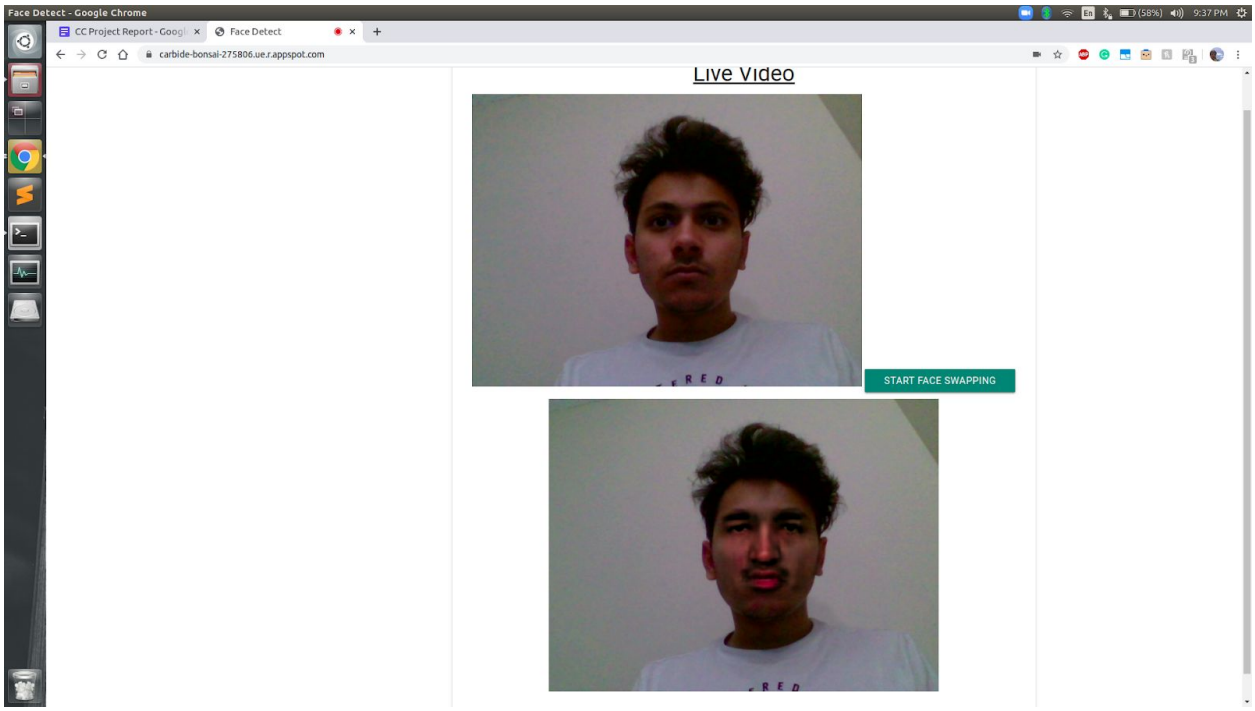


Figure x: Interface provided by the Application Server

The main webpage loads within a maximum of 2 seconds. Since our app is deployed on the auto scalable app engine platform, the application server can be scaled on-demand. In future, if the app is to be used by people from multiple regions, the app engine shall be deployed to serve from data centres closed to the client.

4.2) Websocket Tier

The websocket server acts as a mediator between the AI Platform and the client. The important criteria for the websocket tier is that it should scale for multiple users. It should ideally accept multiple frames from the client and make the relevant API calls to the AI Platform. Based on the tests conducted, the websocket server indeed does scale depending on the number of users attempting to connect. However, an important diagnosis was that it performs blocking API calls

for each frame it receives. This leads to serial processing of the frames which is an issue that needs to be addressed. Thus, due to the performance of the websocket tier, our application experiences significant latency and loss in throughput which shall be discussed subsequently.

4.3) AI Platform Tier

Considering the real-time constraint of the application and the complexity of the deepfake models, it is imperative to have a separate tier to handle the complex operations. The model that has been currently deployed only uses CPU machines. While it helps in significantly reducing the costs, considering the real-time nature of the application, in future it has to be deployed on GPU based servers. The only important metric for evaluating this tier is the time taken by the model for each API call. Based on our observations, for each frame the API call takes around 2 seconds. This accounts for the deepfake inferencing as well as the communication overhead. By the use of GPUs, this time can be significantly reduced. It is important to note that since the AI Platform Prediction scales on-demand, higher time for API call only increases the latency and ideally does not impact the FPS.

4.4) System Performance

In the previous sections, we have discussed the performance of each of the tiers individually, however, the efficiency of the entire system as a whole is the actual functional requirement of the users. The key metrics for evaluating the entire system are latency and throughput. The current latency of the application is around 2 seconds and is caused due to the bottleneck in the websocket server as discussed previously. The throughput, which we measure in terms of Frames per Second is 0.6 currently. For the application to work realtime, the FPS should be atleast 20 frames per second. We discuss the methods of overcoming these bottlenecks in the conclusion.

5. Code

5.1 Installation of the Project

The installation of Client Application Server and Websocket is similar as both are hosted on the App Engine. The Deepfake model is pre-trained and loaded and used in order to get the merged image. The steps included in installing each component is given below, these instructions are given for Ubuntu 18.04:

1) Client Application Server
    a) Install all the required python library using 'pip install -r requirements'
    b) To test the client application locally, we used the command 'python manage.py runserver'
    c) To run the client on the cloud we need to collect the static files in STATIC_ROOT folder using 'python manage.py collectstatic' and run 'gcloud app deploy' to start deployment.
2) Websocket Server
    a) Install all the required python library using 'pip install -r requirements'
    b) To test the websocket locally, we used the command, 'gunicorn -b 127.0.0.1:8001 -k flask_sockets.worker main:app'.
    c) To run the websocket on the cloud we used 'gcloud app deploy'.
3) Deepfake Prediction model
    a) The deepfake prediction model has to be deployed as a custom routine on the AI Platform as a prediction routine.
    b) Run setup.py to package the custom routine and the additional dependencies.
    c) Create a Model on AI Platform and deploy the created package as a version. It is important to specify Pediction.ModelPrediction as the main class which instructs the routine about loading and inferencing from the model.

5.2 Functionalities of the Program

Client Application Server

The client contains 3 important files i.e. app.yaml, static files and facedetect files. The functionalities of each file is given below:

- App.yaml contains the configuration of the app's setting. It tells the URL how to respond to handlers and static files. In our case we load the static html files from the static folder using app.yaml.
- Static files contain the HTML and CSS code for our web application which enables the users to access the face swapping video stream.
- The facedetect folder contains the Django default files generated while initializing the Django framework. The views.py is responsible for the request handling of the image. Urls.py redirects the application according to the path provided. Settings.py holds the configuration values required for the web application to run.

Websocket Server

The Websocket server similar to client server has app.yaml, and also as it is a background task we have no static files and the functions to be executed in main.py which is deployed via Flask application.

- App.yaml contains the entrypoint for the application which is given as a gunicorn command which redirects it to main.py.
- Main.py receives the client frames, makes an api call to the deepfake model and returns the results to the client.

Deepfake Prediction Model

The Deepfake Prediction model is deployed on the AI Platform which provides a PaaS infrastructure to provide machine learning inferencing as a service. The websocket server sends API requests and based on the volume of these requests and current resources, the AI Platform provides automatic scaling. The details of the underlying algorithm are as explained in the background. Following is a description of the files it contains:

- Prediction.py contains the ModelPrediction class which is defined based on the custom routine format. It defines the routines for loading the deepfake model and for performing inferencing.

6. Conclusions

We have developed a 3-tier architecture to enable efficient deployment of deepfake based face swapping. The proposed 3-tier architecture could potentially be used for deploying many other real time applications which rely on complex machine learning models. The Application server serves as an entry point for the user. This provides the user an interface to the underlying cloud infrastructure. The user is then connected to the WebSocket server for real-time duplex communication. Finally, we have an autoscaling ML inferencing server which does the heavy-lifting.

The key learning objective of this project was to understand the challenges involved in deploying a machine learning model to the cloud and come up with solutions for it. An important takeaway we learnt while deploying the machine learning model on the AI Platform is that it is important for machine learning engineers to modularize their code at a higher level to provide decoupled modules for 1) loading the weights of the models and 2) performing inferencing. It is important that the loading is performed only once per instance lifecycle since it takes time.

As discussed in section 4, the key metrics for evaluating our application is latency and FPS. The current latency of the app around 2s and FPS is 0.6. While the latency might be acceptable in some cases, clearly the FPS needs at least a 40x speedup. The reason for the low FPS according to our diagnosis is 1) The websocket server is not working asynchronously and 2) The AI Platform Custom Routine currently doesn't support GPUs. The websockets currently supported by App Engine only allow using Flask. Due to minimal support from Google to provide websockets as a part of App Engine, it would be beneficial to move the websocket server from App Engine to Compute Engine, this would allow more control over the libraries that can be used. The second issue can be addressed by incorporating the entire deepfake pipeline in a single tensorflow graph. This would then enable us to directly deploy the models on AI Platform GPUs, significantly throttling the FPS and reducing the latency.

7. Individual Contribution - Yash Kotadia

The main contributions are summarized in sections as below:

Deployed the Deepfake Model on AI Platform

The Deepfake model lies at the core of the functionality our application aims to offer. While our goal wasn't to modify the deepfake algorithm, significant modifications were required in the code to satisfy the format of the custom routine call in the AI Platform.

The primary reason for the custom routines to have the particular format for deploying the models is that it needs to separate the logic of loading the model weights from the actual inferencing. This is important because ideally the instance should load the model only once in its entire lifecycle to reduce the overheads of loading the model. Thus, the deepfake algorithm had to be packaged according to these specifications.

Specifically, the deepfake model has three models. The face detector, landmark detector and the CNN autoencoder for face-swapping. Each of these models have their own set of weights that they require to perform inferencing. On instantiation, an instance first loads the weights of all the three models. Then during the inferencing, these three models are pipelined to output the desired image.

While all the operations that are performed are to be done on an image, the interface provided by GCP does not allow to directly pass an image as an input for the inferencing. This image has to be first serialized before sending it over the communication channel. Furthermore, based on the specifications provided by GCP for the interfacing, certain modifications were required to communicate the requests and predictions between the websocket server and the AI Platform.

Deployed the Websocket Server

Our current architecture deploys a websocket server for the websocket communication between the client and the server side. While the websockets could have been provided as a part of the application server itself, we chose to deploy a separate websocket server for the following reasons:

- It is generally advisable to abstract the websocket server from the application server since the load requirements of the two significantly differ. Deploying a separate websocket server ensures better load distribution.
- It also prevents the user from directly accessing the expensive AI Platform API. The keys to the API are not required to be exposed with the websocket server.
- The Application server was deployed on the standard app engine since it has the ability to scale down to 0 instances when required. However, the standard app engine does not support websocket communication, thus, we had to default to the Flexible App Engine for providing the websocket functionality.

Individual Contribution - Kshitij Sutar

Designing the Architecture

The architecture for our project was an important aspect as we had to decide on how the requests from the user would be sent to the Deepfake model and returned back to the user with minimum latency. This task required me to understand the important components in this project. So I decided to send the clients requests on the Websocket server instead of the HTTP server as the websocket provided fully duplex communication. Now the main Deepfake model needed a Google resource which would be capable of generating images at a quick rate and this needed high computational resources which were provided by Google AI Platform. Then finally, the decision was made to design a 3-tier architecture where the Client Application server would send requests to Websocket server and websocket server would asynchronously forward these requests to the Google AI Platform for processing the image frame and return the face swapped image in the same fashion. Other detail was that the client application was deployed on the Standard App engine and the Websocket server was deployed on the Flexible App engine. As the websocket required less scalability in comparison to the client as the number of requests received by the websockets were way more than the requests for clients thus we learned that the Flexible App engine supports background tasks better by using the Dockerfile to create a container environment.

Deploying the Client Application Server

The client application was decided to be a Web application, so I decided to make a web application with Django framework as it enabled me to deploy this application on the app engine seamlessly. This web application consisted of a single web page which showed the live video preview of the user from their webcam and once the user clicks 'Start Face Swapping' the user would receive the video frames returned from the AI Prediction Platform.

The tasks which were performed while developing this application are as follows:

- The first task was to build a working web page which displayed the webcam video preview and initially the output was grayscale video preview instead of face swapped video. This was then deployed on the Django python framework and tested on localhost.
- Then, the next step was to build a connection to the Websocket server and send the request in the form of images encoded in the string variable. This was done with the help of the python library called as flask_sockets which enabled the web application to send and receive requests to and from the websockets.
- Then, the client application was to be deployed on the Google App Engine, which required me to understand how to deploy Django on App Engine. The 'manage.py' is where the changes had to be made so that it would make a directory STATIC_ROOT which contains the static files of the web application.
- The final task was to display the face swapped frames received from the websocket server on the clients web page. For this purpose, I had to convert the encoded string variable back to image format and display the image as it was received from the Websocket server.

8. References

1. Feng Y, Wu F, Shao X, Wang Y, Zhou X.” Joint 3d face reconstruction and dense alignment with position map regression network. “ In Proceedings of the European Conference on Computer Vision (ECCV) 2018 (pp. 534-551).

2. Chen, Dongyue, Qiusheng Chen, Jianjun Wu, Xiaosheng Yu, and Tong Jia. "Face swapping: Realistic image synthesis based on facial landmarks alignment." *Mathematical Problems in Engineering* 2019 (2019).

3. Bitouk, Dmitri, Neeraj Kumar, Samreen Dhillon, Peter Belhumeur, and Shree K. Nayar. "Face swapping: automatically replacing faces in photographs." In *ACM SIGGRAPH 2008 papers*, pp. 1-8. 2008.

4. Ko H. “Unsupervised Generation and Synthesis of Facial Images via an Auto-Encoder-Based Deep Generative Adversarial Network.” Applied Sciences. 2020 Jan.

5. Korshunov, Pavel, and Sébastien Marcel. "Deep Fakes: a new threat to face recognition? assessment and detection." *arXiv preprint arXiv:1812.08685* (2018).

6. Pu, Yunchen, Zhe Gan, Ricardo Henao, Xin Yuan, Chunyuan Li, Andrew Stevens, and Lawrence Carin. "Variational autoencoder for deep learning of images, labels and captions." In *Advances in neural information processing systems*, pp. 2352-2360. 2016.

7. S. Ren, X. Cao, Y. Wei, and J. Sun. Face alignment at 3000fps via regressing local binary features. In Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on, pages 1685–1692, June 2014.

8. W. Liu, D. Anguelov, D. Erhan, C. Szegedy, and S. E. Reed. SSD: single shot multibox detector. CoRR, abs/1512.02325, 2015.