UC Berkeley MIDS Program
W251 Deep Learning in the Cloud and at the Edge
Fall Semester, 2020

Team:

- Jeremy Fraenkel (jeremyfraenkel@gmail.com)
- Sang Lee (drminix@gmail.com)
- Chris Weyandt (cweyandt@berkeley.edu)
- Ian Anderson (imander@berkeley.edu)

Covid-19 Mask and Fever Detection

Repository: https://github.com/cweyandt/mask-and-fever-detector
Sample images: http://w251-covid-maskdetector.s3-website-us-west-2.amazonaws.com/

Face and Mask detection

Overview

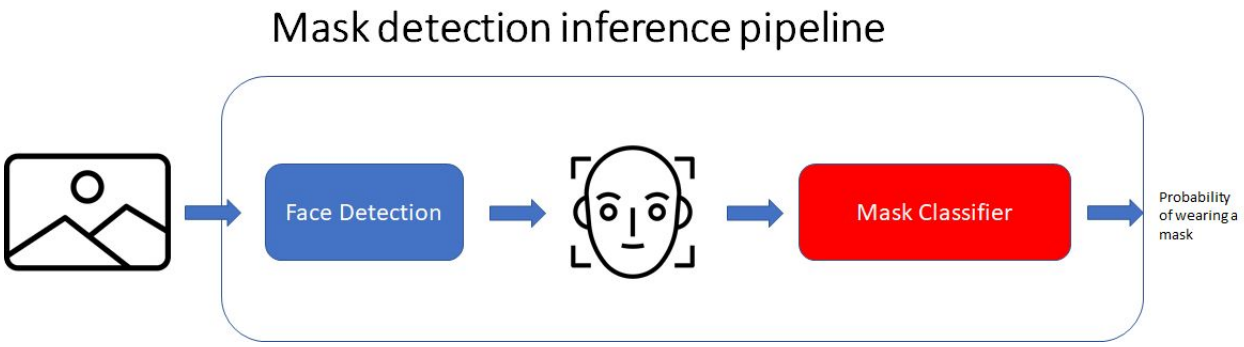


Figure 1. Mask detection inference pipeline

Figure 1 shows the mask detection inference pipeline. It takes a series of live video frames as input, and outputs probabilities of people wearing masks in the given frame. It consists of two modules. These are face detection and mask classification modules. For each input frame, the face detection model detects the bounding boxes of every face. For every detected face, the face detection model resizes and preprocesses it before passing it to the mask classifier. Mask classifier then predicts the probability of a person wearing a mask for each face image.

Face detection model

Three different face detection models were evaluated for this project. These are Haar cascade face detection model[1], YoloFace model[2], Res10-SSD model[3]. Haar cascade face detection model is a simple feature-based cascade classifier objection detection model. It provided the best FPS rate on the edge device but it failed to detect faces especially when there are multiple faces in the same frame. YoloFace is a YoloV3 model trained on WIDER FACE dataset[3]. It provided the best accuracy but it was too slow to process live video streams on NVIDIA Jetson Xavier NX. Res10-SSD model is also a deep-learning based method which uses Single Shot MultiBox Detector framework to effectively detect objects in real-time. Its prediction was not as accurate as the YoloFace model but it processed live videos on NVIDIA Jetson Xavier NX.

Mask Detection Model

Overview

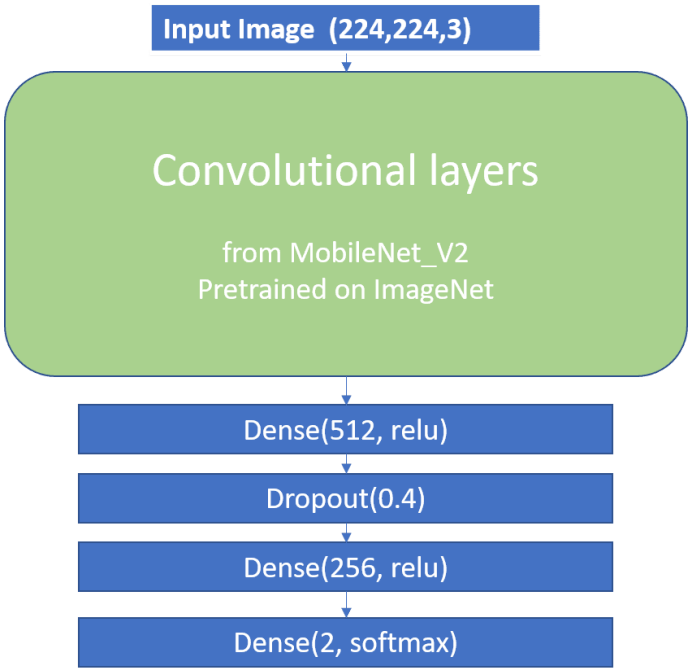


Figure 2. Mask detection model architecture

Figure 2 shows the network architecture of the mask detection model. It consists of convolutional layers that were extracted from MobileNetV2 [4] and a simple feedforward network. Convolutional layers from pretrained MobilNetV2 were adopted and used as a feature extractor. The output of the convolutional layers is then passed to the feedforward network which computes the probabilities of a person wearing a mask. Data augmentation and Dropout are applied as regularizor to prevent overfitting. MobileNetV2 was chosen since it achieves optimal performance on embedded devices with limiting resources [4].

Deep learning framework

For deep learning framework, Tensorflow2.0/Keras was chosen for its simplicity and usability. Other deep learning frameworks and models such as TensorRT with ONNX could have been adopted to achieve better performance on edge devices.

Data sources

600K face and mask-ed face images were gathered from multiple sources. These are public social media dataset[5], Kaggle mask detection dataset[6].

Model Training and Deployment

Overview

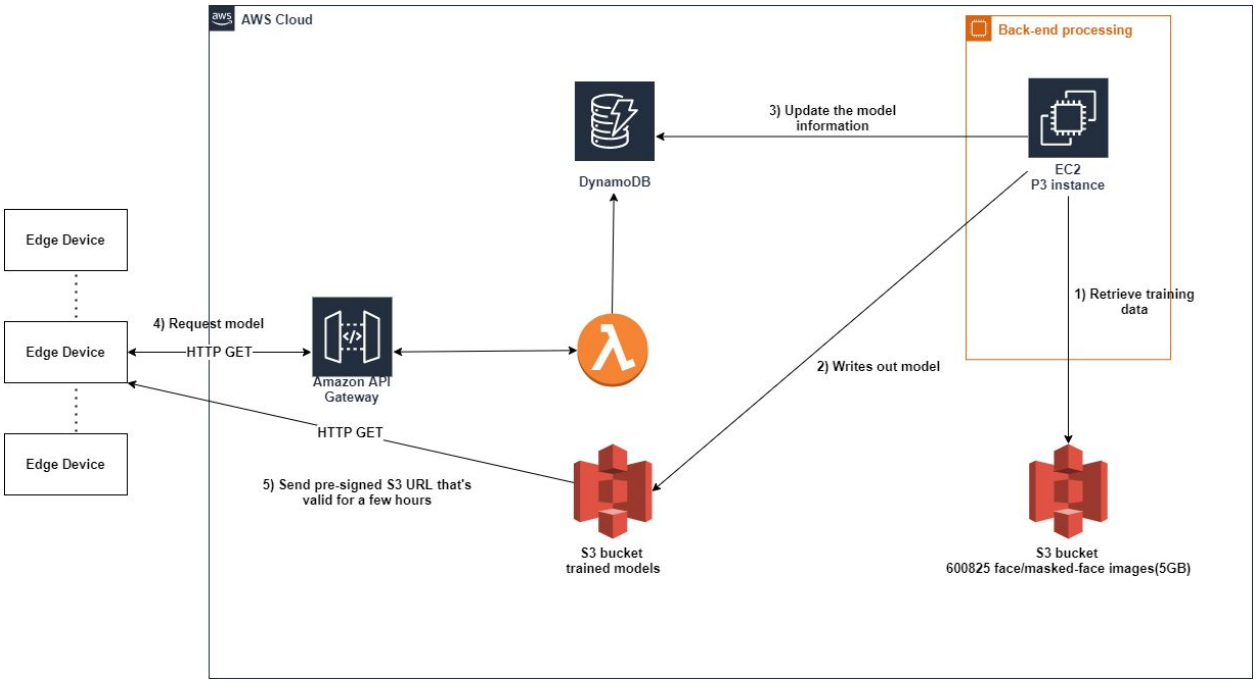


Figure 3. Mask classifier training and deployment

Figure 3 shows the mask model training and deployment architecture. Mask detection model training was done in the cloud. Face and masked face images are stored in the input S3 bucket. A P3 instance was configured to read the data from the input S3 bucket and train the model. Once the training is done, the final model is copied to the output S3 bucket and the training status and model information is updated in the dynamodb database.

Model deployment

For deployment, a serverless architecture is adopted to reduce the cost while maximizing the scalability. To update the mask detection model, edge devices connect to the API gateway over REST API. Lambda function then queries dynamodb database to check if any new models are ready for deployment. If so, lambda function sends back a new pre-signed S3 URL that's valid for a short period of time. Edge devices can download updated models using the provided URL.

Infrastructure

Architecture

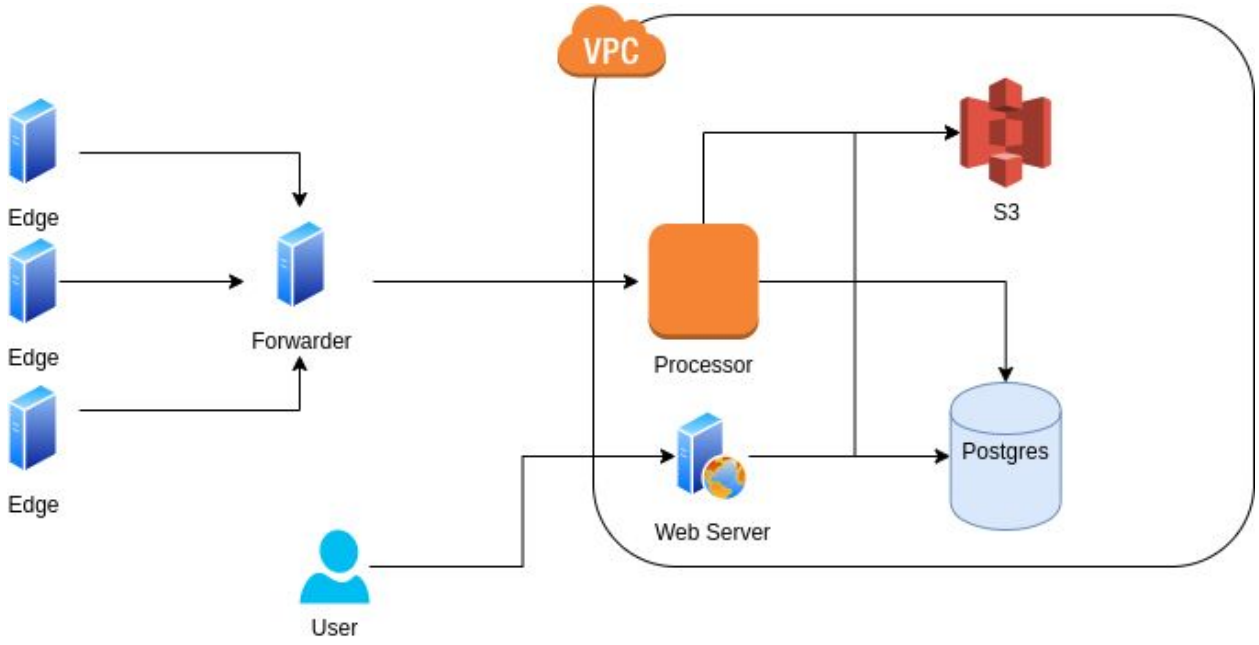


Figure 4. System architecture

Figure 4 shows the system architecture. There are four components: edge device, forwarder, processor, and web server. Edge device performs facial detection, mask detection, and temperature detection. It also sends gathered face images to the forwarder. Upon receiving face images, the forwarder sends a message to the cloud over MQTT protocol. Similar images are discarded to conserve network bandwidth/storage. Processor receives the messages over MQTT protocol and saves detected images into S3 static site and mask detection statistics into postgres database. Web server provides two separate end-points that allow users to either view raw data captures or visualize aggregate statistics.

Deployment scripting

Makefile

Once our application grew beyond a single container it was clear that a means for standardizing the build and deployment process was necessary. A Makefile located in the top folder of our repository is configured with many targets that script common actions taken throughout the development of the project. These targets cover multi-architecture docker image builds with

automatic pushing to DockerHub, AWS service initialization and destruction, edge device provisioning and deployment, and website deployment.

An environment (.env) file is used to pass user-defined parameters into the build process, such as DockerHub repository location, webcam and thermal camera configuration parameters, MQTT configuration, AWS parameters and edge device IP addresses.

Terraform for AWS service management

Terraform is used to script the deployment and configuration of AWS resources. Upon initialization the terraform infrastructure will utilize stored AWS credentials (from AWS CLI) to create the cloud VPC server and s3 bucket. Security credentials are automatically configured to limit IP traffic to the predefined edge devices and the controller (computer where terraform is running). Terraform collects all of the necessary information such as VPC public IP addresses and passes them back to the host environment for exporting to edge device environments.

Ansible for application deployment

Ansible is used to perform scripted actions on all of the target machines. For example, after provisioning AWS resources with terraform, ansible performs remote installation of packages, copies resource files to the cloud and edge devices, pulls appropriate container images from DockerHub to the target machines and executes them. It is also used for updating remote websites, and for stopping, and destroying services. Ansible can be used to perform many scripted tasks on remote targets with near effortless scalability.

Websites

Two websites are deployed in the cloud. The first is a simple image browser is loaded directly onto the AWS s3 bucket and can be accessed at the root of the bucket URL with the following syntax:

```
http://<bucketname>.s3-website-<region>.amazonaws.com
```

This website is a simple javascript interface that reads all of the folders in the s3 bucket and creates buttons on the web page for each folder. When a folder button is clicked the web frame is updated with a grid containing all of the images in the s3 bucket folder, with each image also serving as a hyperlink to download the original image. Basic CSS is used to adjust the number of columns in the image grid when the window is resized.

The second website is an interface to mask detection statistics deployed on the AWS EC2 instance over port 8080. This Metabase site aggregates metadata from the Postgres database running in a different container on the same EC2 instance. This database includes image detection statistics from each edge device. The Metabase dashboard shows detection classes as percentages and as time-series statistics.

Data Transfer

Simple MQTT brokers sit at the edge and in the cloud. At the edge, the broker collects images and metadata from the mask detector and delivers them to the message forwarder. In the cloud, the broker accepts messages from one or more edge devices and hands them off to the message processor.

To limit unnecessary data transfer and storage, each face-containing frame output from the mask detector is compared to the last 100 frames the forwarder has received. A histogram-based similarity measure is conducted against each of the frames in the buffer and the data is discarded if the similarity is too high. This process dramatically decreases the number of images uploaded to the cloud server, while also having a short enough memory that it will not miss important changes in the scene.

User Interface mode

While the edge device is intended to operate in a headless state with no user interface, a Qt5 application was developed to display the resulting images directly on-screen in the Jetson gui. The UI populates a window that displays the post-processed video streams with detection results imposed on the video frames. The UI mode can be run on the Jetson simply by executing the docker-compose.ui_mode file at the top of our github repository[1] in a graphical interface. Images for the mask detector will be automatically downloaded from DockerHub run the mask detection models in UI mode without having to clone the repository or build any images.

Imaging

Standard video streaming

The application framework was first built upon the standard openCV VideoCapture class which allows the detection service to be run with a basic webcam feed. The detector will run as fast as the hardware will allow, capturing a new RGB image frame as soon as the detection process is complete. It is not necessary or desirable to process every single frame at the camera's default framerate, so capture threading is not required.

Each RGB is sent to the face detection service where it is resized, converted to grayscale, and then processed. If faces are detected, the original image will have a timestamp and face boxes with detected class (mask or no_mask) and associated confidence level drawn on it before displaying on screen in the UI and/or transmitting to the forwarder for further processing.

[1] https://github.com/cweyandt/mask-and-fever-detector/blob/main/docker-compose.ui_mode

Thermal video streaming

The application is capable of measuring surface temperature of identified subjects using a long-wavelength infrared camera. The Lepton 3.5 camera[2] by FLIR was chosen for it's exceptional balance between cost and quality. The $199 module is small enough to fit in a smartphone while providing 160x120 pixel resolution in which each pixel represents the absolute temperature of the imaged surface.

The Lepton camera module is packaged for on-circuit mounting and communicates over an SPI interface. To simplify connectivity to a wide variety of edge devices, an additional capture board was used to provide camera module mounting and a USB interface. The PureThermal2[3] interface allows developers to stream pre-processed images from the FLIR camera over a standard v4l2 pipeline, yielding standard RGB images that have been converted to full color to represent the relative temperatures of objects in the frame. Using the thermal camera in this mode is not advantageous because it does not include raw pixel-by-pixel radiometric (thermal) information. Instead, a special fork of the UVC capture class[4] is used to obtain raw radiometric data from the camera with the help of a python wrapper[5] for the C libraries.

The user can enable the THERMAL_ACTIVE=True flag to the .env file. Doing so will trigger use of the custom PureThermalCapture class which captures synchronized frames from both the standard webcam feed and the Lepton IR camera.

Radiometric data processing

When in RAW mode, the PureThermal2 capture interface transmits arrays of radiometric (thermal) data. The array contains temperature measurements in degrees Kelvin for each pixel of the 160x120 image. Any pixel can be read and converted into degrees Fahrenheit or degree Celsius through a simple linear transformation. If plotted, however, the image will be unrecognizable because the range of data is not within proper range. Processing the thermal measurement array into a user-viewable image requires downsampling from 14-bit radiometric data to 8-bits and normalizing over the range of data in the frame. The result is a grayscale image in which the pixel intensity is relative to the surface temperature measured, however it is no longer possible to extract the respective measurements from each pixel. For this reason, both the thermal array and the grayscale image are transmitted through the pipeline for further processing.

[2] https://www.flir.com/products/lepton/
[3] https://groupgets.com/manufacturers/getlab
[4] https://github.com/groupgets/libuvc
[5] https://github.com/groupgets/purethermal1-uvc-capture

Capture threading

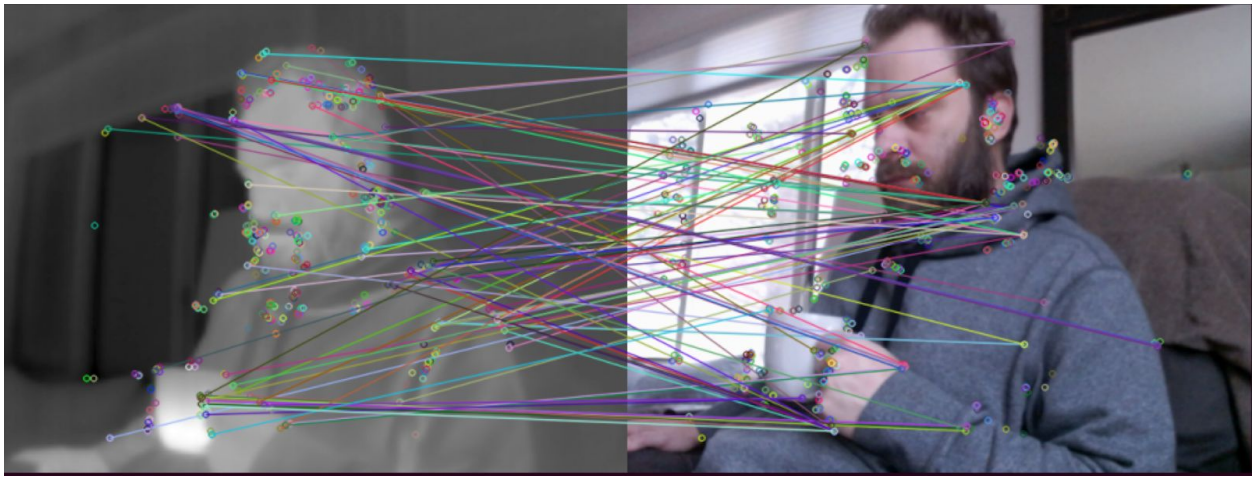
Although standard USB webcams are capable of frame rates in the tens-per-second range, the FLIR Lepton 3.5 maxes out at 9 fps. During initial testing it was immediately apparent that the frames of the RGB camera and the IR camera need to be carefully synchronized to account for moving subjects. Even at 9 fps the thermal camera could likely be capturing images faster than the detection process can run, so it was necessary to create a buffer holding a copy of the most recent frame capture.

The lib-uvc capture class uses a ctypes pointer attached to a callback function that is executed each time a new radiometric frame is captured. The callback function pulls a quick RGB frame from the standard webcam, takes a timestamp, calculates min and max temperatures within the frame, and then bundles that information to store in the class buffer. The class is intended only to support the return of the most current pair of RGB + Thermal frames so unused frames are dropped from the buffer when a new capture is successful.

Image registration

When running two cameras side by side it is necessary to correct for differences in the angle and field of view. For standard images this process is relatively straightforward using openCV libraries for motion translation and image homography. Both of these methods attempt to align images based by performing feature extraction and then comparing the similarities between the features in both images. Unfortunately this technique does not work when aligning radiometric images with standard images, because the pixel intensities do not have the same meaning. In a standard image the intensity of each pixel is a measure of the light reflected off of that surface, whereas in a radiometric image each pixel is a measure of the temperature of the surface. Although the edges in these images are often similar, the extracted features rarely align.

Extracted edges from a radiometric image (left) and a standard image (right)



Feature matches between radiometric image (left) and standard image (right)

The above image shows the feature matching pair for the thermal and standard images. The images are already relatively closely aligned through physical orientation of the cameras, yet many of the pairs are connected to drastically different areas of the opposing image. You can see that the windows in the background are black (cold) in the thermal image, while they are white (bright) and contain usable features in the RGB image. This causes significant differences in the features that are extracted from each image. The resulting transformation of either image leads to a completely warped perspective.

Translated thermal image based on feature matching (left) and standard image (right)

While several potential modifications to this method are apparent such as limiting the pixel-to-pixel distance for feature matches (i.e. prune matches by distance instead of by feature similarity measures) or aligning images based on extracted edges, this has not yet been implemented. For prototyping, a small ball-mount was attached to the webcam to hold the thermal camera in a precisely aligned position. Future enhancements would include a camera-calibration mode in the mask detector UI that allowed the user to perform a static calibration of the camera offsets by clicking common points in each image.

A standard webcam with ball-mount for thermal camera

Future Work

- Use more efficient C++ based deep learning framework such as TensorRT.
- Rather than using a two stage approach, use one-shot masked-face detection model.
- Improve mask detection model performance.
    - Gives false positive for people with beard
- Determine effective method for performing image alignment between radiometric and rgb images. Perform fever detection within each face bounding box only, instead of measuring across the entire frame.
- Determine correlation between facial surface temperature and fever conditions. We do not know what measured facial temperature should indicate a fever, and it is likely affected by many conditions.

References

[1] Viola, Paul, and Michael Jones. "Rapid object detection using a boosted cascade of simple features." *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*. Vol. 1. IEEE, 2001.

[2] Chen, Weijun, et al. "YOLO-face: a real-time face detector." *The Visual Computer* (2020): 1-9.

[3] https://github.com/chuanqi305/MobileNet-SSD

[4] Sandler, Mark, et al. "Mobilenetv2: Inverted residuals and linear bottlenecks." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018.

[5] https://github.com/X-zhangyang/Real-World-Masked-Face-Dataset

[6] https://www.kaggle.com/andrewmvd/face-mask-detection