Spring 2021

A Low-Cost Face Mask Detection System

Joshua Curtis
The University of Akron, jwc72@uakron.edu

Logan Mospens
The University of Akron, ljm87@uakron.edu

David Nelson
The University of Akron, drn26@uakron.edu

Sumner Wilson
The University of Akron, srw103@uakron.edu

Adam Kilo
The University of Akron, ank80@uakron.edu

FINAL SENIOR DESIGN REPORT

May 2021

Team 22: Mask Detection Vision System

Logan Mospens, David Nelson, Sumner Wilson, Joshua Curtis, Adam Kilo

Advisor:
Dr. Yalin Dong

With the assistance of:
Wenjing Yang

INTRO

In this project we approached the problem of developing a computer-based system which can identify whether people in images are wearing masks or not. To do so, we divided the project into two primary research areas: hardware and software. The hardware team inspected camera solutions and data gathering efforts on campus to assist the software training of the machine learning model. The software team learned how to work with machine learning by using TensorFlow and Python to use transfer learning to develop a model that will quickly detect if someone was wearing a mask or not.

We experienced many roadblocks that we worked around. For example, the hardware team encountered issues with UA IT with establishing a wireless IP camera on campus for data gathering – there being issues with connecting to the actual UA Wi-Fi connection. On the software side of things, the first semester was mostly spent researching machine learning, since no one had previous experience with machine learning. Also, a software issue that arose is that the use of case of machine learning used only detected if an image contained someone wearing a mask or not – it does not detect people who are and aren't wearing masks. Therefore, a middleman solution had to be created to capture faces to then be passed to the mask detector.

Table of Contents

NOTE ON STANDARDS

As this is mostly a software project, there were not many physical standards to abide by. However, one critical aspect of engineering we wanted to abide by were keeping in mind the core tenets of what it means to be an Engineer. In an artificial intelligence project such as this, there is wide potential for gross misuse, especially in the case of working with people's faces. In our project we worked in careful considering of what public resources we were able to use to gather data, as well as consulting the University of Akron for permission to record on campus to gather some training data for the model.

One such Standard reviewed is IEEE-7010-2020, which is titled as "IEEE Recommended Practice for Assessing the Impact of Autonomous and Intelligent Systems on Human Well-Being". This standard outlines a great many examples and definitions in which we can gauge the impact our autonomous system will have. In our case this was critical to this project, due to its involvement in the processing of facial images.

Moving forward with this in mind, we'd like to be firm in stating that this system is in no way intended to or developed to recognize, record, store, or report faces acquired. Our final product is just able to just point out that there is a face with or without a mask on screen. The most that can be made of this project as-is would be a simple statistic program that counts how many individuals are wearing masks versus those that are not.

HARDWARE

After learning the basics of coding in Python, David Nelson decided to move to the hardware team, which was responsible for getting and operating a camera. The first step to finding a suitable camera was to figure out the needs of the group. Price was a factor, so high-end cameras were not recommended. The camera had to be able to communicate with a computer or laptop. At first, this was thought to mean the camera had to operate wirelessly over an internet connection. Cords for the camera was also a consideration, for power and communication with a computer. Battery powered and wall outlet powered cameras were included in the search.

Basler cameras made specifically for machine learning were considered next. The focal length needed to be calculated to determine which lens to buy. Eventually, the Basler camera idea was scrapped for being too expensive and for not being able to work wirelessly. Next, a Reolink brand 5MP security video camera was purchased, as it was advertised to work wirelessly. Special permission had to be obtained through the IT Department on campus, along with campus security, in order to start gathering video footage to train the machine models.

The camera was to be mounted on a wall in the Student Union Center and be able to be accessed remotely. The wall had to be near a wall outlet so that the camera could remain powered on. A special non-intrusive mounting apparatus was constructed such that the mount would not do damage to the wall in the Student Union. This apparatus consisted of screwing the camera into a small painted wooden block, and then having adhesive Velcro-like strips placed on both the wall and the wood block. The adhesive wall strips were rated for 16 pounds, which was more than enough for the small camera.

When the camera was finally ready to be installed, the camera would not turn on. David figured out that the camera needed to be wired to a wireless modem in order to complete setup to the network. Since there were no wireless network modems nearby, the Reolink camera was determined unusable for the project. Search for a second camera had to commence.

Another camera, the Reolink E1 Pro, was purchased, as it could connect wirelessly and be accessed remotely. The camera would not connect to any of the university's wifi networks, so progress was halted. David discovered that the camera could connect to his personal hotspot generated by an iPhone, and setup was completed. The camera was able to record data through a Reolink-brand application on a laptop. The problem now was that the camera could not operate remotely on campus as originally intended.

A third Blink-brand camera was purchased, as it ran on batteries, but this camera was never used for the project. It was intended to be a backup if the second Reolink camera did not work.

The next step was to gather as much data of people wearing face masks. At first, David used his phone camera to take pictures of people with masks, but this was soon found to be subpar for a couple of reasons. First, the camera angles and background would not be consistent from picture to picture. Also, the people were posing for the camera, which would not happen with a real-life security camera. Data needed to be collected in another way.

Eventually, David started bringing the Reolink camera to the Student Union to record people walking by, as mask wearing in the building was mandatory per university policy. Adam assisted in data collection with his camera and tripod. One problem that occurred was the low number of people walking by the camera. Life on campus was less densely populated with students than usual due to the COVID-19 pandemic. Another issue was the distance from the camera to the pedestrians was far enough to create a less-than-ideal camera image which the machine learning program had trouble discerning. There were also long periods of time in the recordings where nobody was walking by the camera, which made the videos cumbersome to use.

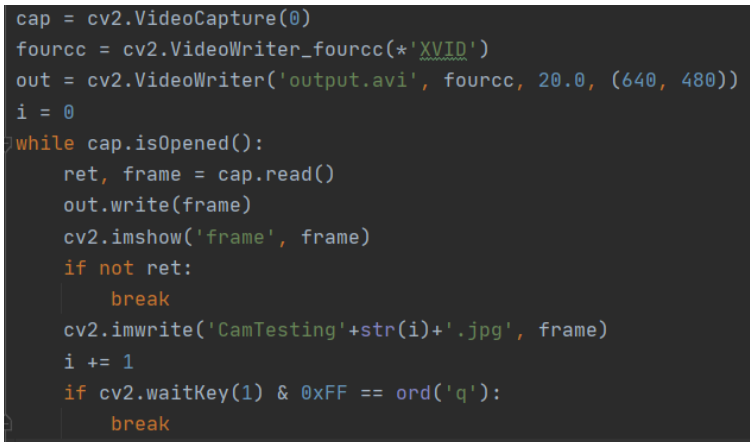
David then spent a week editing the video segments down to only moments with people walking by. This took longer than intended because the lengthy videos had to be played back in real time for the editing software to work. Adam contributed 89 faces to the dataset by cropping footage from campus. The next step was to get the camera to be operated by Python code. The Reolink E1 Pro camera only worked through an app, which could not be bypassed and therefore access via Python was not an option. It was decided to look for a different solution for the camera.

Raspberry Pi was considered due to a couple of reasons. The system was small, relatively portable, and cheap to get. The camera can be interfaced directly with Python code, unlike the other cameras. It also consists of both a computer and camera, so it could run the Python code and capture footage in the same module. The team went ahead with purchasing a Raspberry Pi camera system, and with more time, it would prove a valuable asset to our end goal of creating a vision-based mask detection system.

SOFTWARE

Accessing Hardware

First, when accessing hardware, we had to learn about accessing internal hardware, present in our own computers or laptops. In this case, internal laptop webcams were used. To do this, we worked with an open source computer vision library called OpenCV, which specializes in image and video content. The following code shows an example of how to access such an internal webcam:



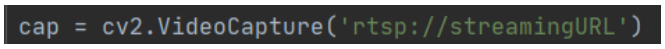
```
cap = cv2.VideoCapture(0)
fourcc = cv2.VideoWriter_fourcc(*'XVID')
out = cv2.VideoWriter('output.avi', fourcc, 20.0, (640, 480))
i = 0
while cap.isOpened():
    ret, frame = cap.read()
    out.write(frame)
    cv2.imshow('frame', frame)
    if not ret:
        break
    cv2.imwrite('CamTesting'+str(i)+'.jpg', frame)
    i += 1
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
```

This code accesses the internal webcam and saves each frame of the video output into a specified folder.

1. "cap=cv2.VideoCapture(0)": This line specifies the location of the webcam and sets its output to be captured.
2. "out=cv2.VideoWriter('output.avi"…": This line specifies the output of the captured video footage to be a .avi file, called "output.avi". The remaining parameters in the line specify the conditions of the output, such as the resolution.
3. "while cap.isOpened():": This line begins a continuous loop which will run as long as the code has been started. The following lines are all contingent on this condition being fulfilled. The video input is read and captured, the output is displayed, and the frames are set to be written, or saved to a file folder called "CamTesting", with file names corresponding to the frame number, beginning with 0.
4. "if cv2.waitKey(1) & 0xFF == ord('q')": This line sets the conditions for ending the video capture. In this case, if the "q" key is pressed on the computer, the video capture and frame saving will cease.

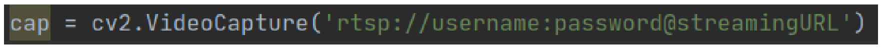
Accessing IP Camera Equipment

Once a camera had been acquired, we needed to develop a code which could connect to it remotely. As mentioned in the hardware section, our first solution was to purchase an IP camera. OpenCV has a built-in functionality which allows users to access a camera through an RTSP (Real-Time Streaming Protocol) streaming URL. Barring our issues with connecting our equipment to the wi-fi at UA, the following line of code would substitute in to access an IP camera:



```
cap = cv2.VideoCapture('rtsp://streamingURL')
```

Since many IP cameras also require a username and password, the previous line of code can be modified to:



```
cap = cv2.VideoCapture('rtsp://username:password@streamingURL')
```

The rest of the code from the previous section would still be applicable, regardless of whether an internal or external camera is being used.

Accessing Raspberry Pi Camera Equipment

The second solution we considered was to integrate a Raspberry Pi camera module to the project, and use it as an IP camera. As mentioned in the hardware section, the same issues regarding internet access were present, but outside of that, the code for accessing the video from this camera is slightly different.

```
camera.start_preview()
camera.start_recording('/home/pi/Desktop/output.h264')
```

Similar camera operations would then take place, following the required pipeline.

Learning Machine Learning

To begin the work on the machine learning aspect of the software, we first had to learn to work with machine learning. To start this, we worked with TensorFlow's official documentation, as it had many example projects to start with. This was especially helpful considering their tutorials surrounding using Convolution Neural Networks (CNNs) to create image classifiers. Some of the tutorial projects include: a clothing classifier (shirt, shoes, dresses, socks, etc.); a cat versus dog image classifier, and an NIST handwritten digit classifier (recognizes written numbers 0-9).

These tutorials helped introduce many important topics of machine learning, including but not limited to data augmentation, convolutional layers, pooling layers, and more. These strategies helped greatly in learning more about machine learning down the line, as well as helping our training process using data augmentation.

Data Augmentation

Data augmentation is essentially modifying the training data in realistic ways before training to strengthen the validation and real-world accuracy of the model. One can do many things to add a step of data augmentation, but the following figure shows many of the operations that can be randomly done to a training image (next page):

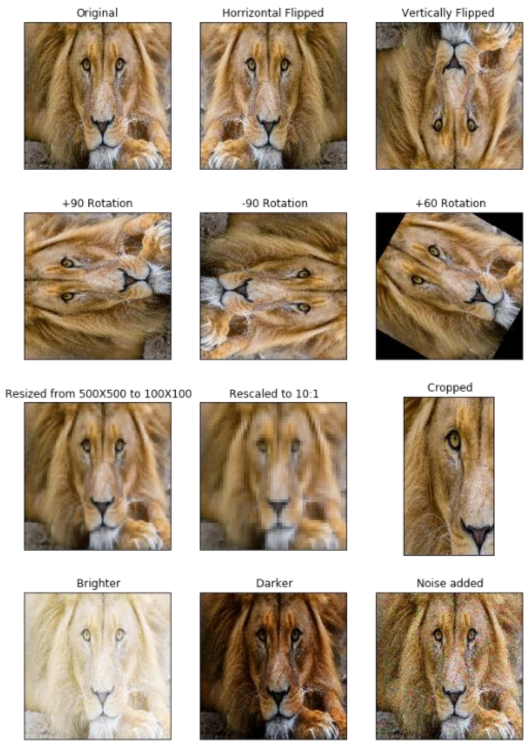


Figure 1: Possible Data Augmentation operations.
Source: https://www.mygreatlearning.com/blog/understanding-data-augmentation/

Dataset for Training

To teach the machine learning model how to recognize a masked face and an unmasked face, we require a large amount of data. After gathering data mostly online through publicly licensed means and on-campus gathering methods we were cleared for, we ended up with over 2,200 total images to use to train our machine learning models.

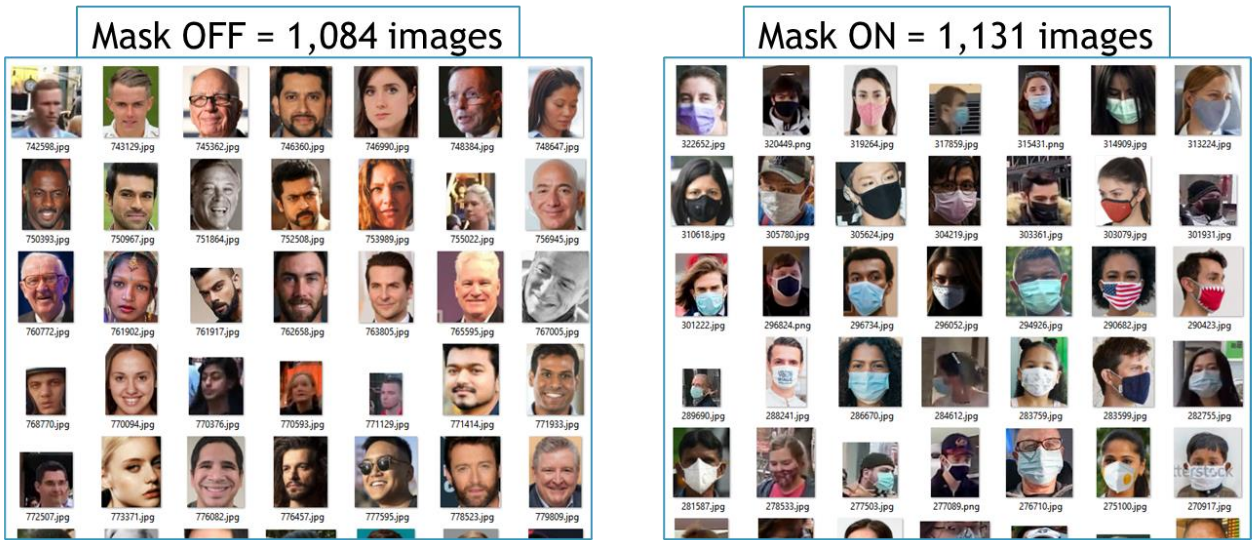


Figure 2: Examples of images used for training and validation

Our image sources include the following:

- Kaggle, Rajtilak Pal (rajtlakls2510) "Face Mask Detection"
- Kaggle, Shanmukh (shanmukh05) "Mask_Detection"
- GitHub (MIT License), Chandrika Deb (chandrikadeb7) "Face-Mask-Detection"

From these sources, we then post processed the images to make sure they fit our use case and cropped many outliers to focus on the faces. And in the cases of large datasets, we did not blindly copy the images, but instead tried to keep only the best images to reduce the amount of poor resulting biases that would be inflicted on the resulting model. We also made use of public YouTube videos, where people would walk through cities, and from there I would screenshot faces with and without masks to add into our dataset.

The following public YouTube videos were used in this way:

- "Paris Evening Walk and Bike Ride - 4K - With Captions!", March 20, 2021
- "RIO DE JANEIRO Downtown, Walking Tour Rio City Center — BRAZIL Walk (Narrated)【4K】BR", December 22, 2019
- "Rome, Italy - 4K Virtual Walking Tour around the City - Travel Guide", August 23, 2019
- "【4K】Walking New York City: The Flatiron District in Lower Manhattan", March 5, 2021

Transfer Learning

After getting a grasp on the fundamentals of working with an image classifier we began research into how to apply transfer learning to this project to expedite and strengthen our final model. To do so was quite daunting at first, as many tutorials and instructional material incorporate somewhat hacky methods of doing so, involving manually freezing all layers, and ripping off the classification head of an already-trained model. This method works just fine if you are using a very specific pre-existing model, but in our case, we were able to make use of the Tensor Flow Hub (TFHub) to simply retrieve the feature vector layers of MobileNetV2 models.

Our Model

The TFHub Python module has extremely helpful methods that allow for a simple function call to download a TFHub model feature vector for easy use within a new sequential model. For example, the following code snippet shows an example of how we used the TFHub module to download a MobileNetV2 variant's feature vector:

```
tf.keras.Sequential([
    tf.keras.layers.experimental.preprocessing.Rescaling(1. / 255),
    tfhub.KerasLayer("https://tfhub.dev/google/imagenet/mobilenet_v2_075_96/feature_vector/5",
                     trainable=True),
    tf.keras.layers.Dense(256, activation="relu"),
    tf.keras.layers.Dense(2, activation="softmax")
])
```

This above snippet is the final body of the model we used. As you can see, it's extremely simple at only 6 lines of code. What this model does can be boiled down to the following steps:

1. "Rescaling": The given 96x96 image is converted into black and white (so each pixel is a value between 0 and 1, not an RGB array of three 0-255 values.) this is required for the MobileNetV2 model.
2. "tfhub.KerasLayer": The image is passed through the chosen TFHub model, and then a huge array of features is returned. The specific "mobilenet_v2_075_96" we use returns 1280 feature matrices.
3. "Dense 256": This layer takes the 1280 features and condenses them between 256 neurons. Each one of these neural nodes takes input from all the previous features, meaning each of the 256 neurons is influenced by all the previous 1280 features.
4. "Dense 2": This is the final layer of the model. This dense layer works the same as the previous dense layer, but instead results in the final binary prediction between "mask on" and "mask off" using the previous 256 neurons. The softmax activation output of this 2 neuron dense layer results in the "normalized probability distribution" of the prediction, resulting in a confidence value for the prediction.

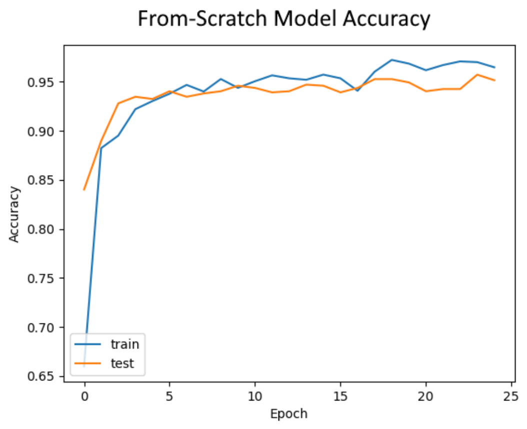
Performance

Transfer learning has two massive upsides that we took advantage of: first, the massively assisted training process of the model. Transfer learning allows us to use an already-finished complex set of convolution layers that have already been optimized and prepared by Google (in this case of using MobileNetV2). I saw this effect very clearly when training many models. To capture the best functioning model, I would train about 10 models at a time, and select the best performing one from the training session. Its important to note here that I did not focus solely on the training accuracy of these models, because validation accuracy shows the actual performance of the model with images it has never seen before. This process was greatly assisted by transfer learning because training a new iteration of the model only took approximately a minute and a half for 4 or 5 epochs.

Training Strategy	Epochs Needed	Training Time (seconds)	Time per Epoch (seconds/epoch)
From-Scratch Model	25	90	3.6
Transfer Learning	4	30	7.5

Figure 3: Training time comparison between model strategies

Next, using transfer learning greatly assisted in rapidly increasing the accuracy of the model after only a few epochs. For comparison, when I was experimenting with a from-scratch sequential model I designed, transfer learning takes only approximately 4 or 5 epochs to stop progressing in accuracy, whereas a from-scratch model takes approximately 25 to 30 epochs to reach a stop in accuracy gain. Its also worth noting that the time spent per epoch of the transfer learning model is much higher, but that very negligible downside is very well worth the performance increase when the result still predicts whether a face is wearing mask in approximately 30ms.



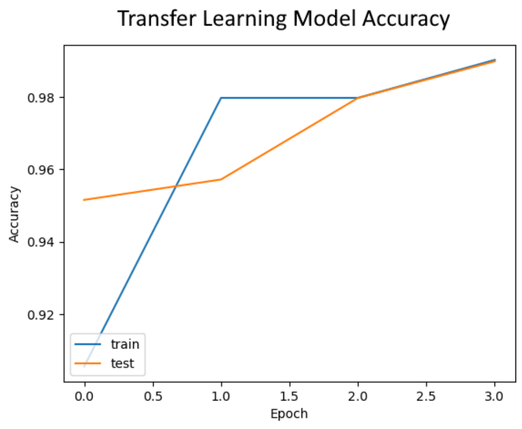


*Figure 4: Comparing accuracy of model strategies (Note that accuracy axes are not equal, Transfer learning performs **extremely well**)*

To clarify, in this above figure, the “train” accuracy is the accuracy in which the still-learning model was able to predict whether a face was masked or not. This “train” set contains a huge majority of our dataset’s images and is what is being used to teach and improve the model. However, the “test” accuracy, is the evaluation of the remaining images from our dataset that the model has never seen before. So, in our case, we chose a 60/40 ratio for testing and validation, meaning that 880 images are evaluated for the “test” AKA “validation” testing of the model. This “test” accuracy is important since the model never gets to learn from these images, meaning that it’s a true look at the current performance of the model.

Detecting People

A resulting problem we faced was that during our learning and application of machine learning, we developed a classification model instead of an object detection model. The below image shows the difference between the two concepts:

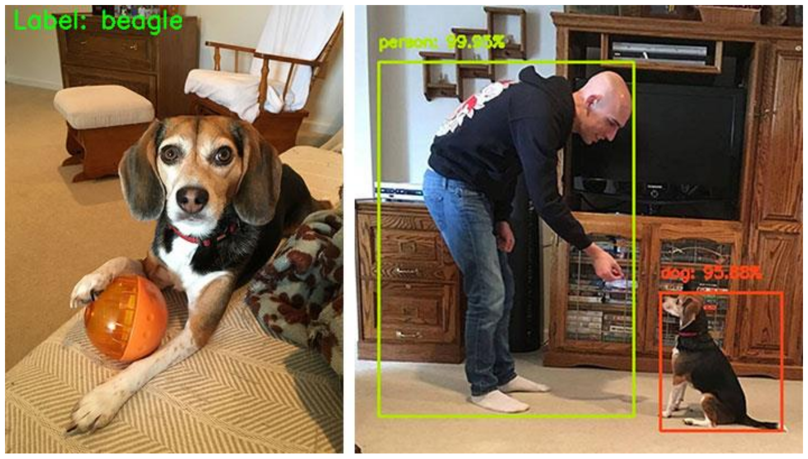


Figure 5: Difference between an image classifier (left) and an object detector (right)
Source: https://www.pyimagesearch.com/2020/06/22/turning-any-cnn-image-classifier-into-an-object-detector-with-keras-tensorflow-and-opencv/

Essentially, the key difference here is that an image classifier takes an image input, and then processes it to detect which applicable label applies to it. But an object detector finds many objects that fit the labels that a model is capable of detecting. From the above image, you can see that the first image detects the beagle, and the second finds a beagle and a person. If the second image instead used a classifier, the **whole image** would instead most likely be detected as a person, leaving out the entire dog.

This is where our detection problem began. Our transfer learned trained model is only capable of classification. Due to the long period of time we had to spend just to learn machine learning and transfer learning, we did not also have the time to learn how to apply transfer learning to a object detection model, so instead we pursued methods to capture the faces or heads of people detected in frame. To do so, we researched four main methods to do so: SSD person detection, Haar Cascades, MTCNN face detection, pose detection, and HOD (Histograms of Oriented Gradients). The following figure shows a summarized breakdown of each of these methods:

Method	Fast?	Accurate?	Ease of Use	What it does
MTCNN	3	1	1	Machine learning model detects faces
SSD person	2	3	1	Machine learning model quickly detects people, and we assume the head to be the top third
Haar Cascade	1	4	1	Facial geometry is quickly detected. MANY false positives.
Pose Detection	4	3	4	Machine learning model detects pose points, create square between shoulders to capture head
HOD	2	3	2	Detects people through histograms of gradients containing pixel vectors

Figure 6: Facial detection strategy matrix

To summarize, let's discuss the three best methods in depth.

HOD (Histograms of Oriented Gradients)

This first strategy is based off of histograms generated from 2D vector gradients. These gradients are generated based on the color difference in the cell. These gradients are stored in 1D histograms, where the values correspond to the angles of the gradients, and the weights correspond to the magnitudes of the gradients. The process pipeline for this strategy is as follows:

1. The frame is scanned with a set detection window.
2. The detection window is subdivided into cells, which contain a number of pixels.
3. For each cell, a gradient is calculated for each pixel. These gradients are used to fill a histogram.
4. The histograms of each cell are combined and sent to a machine learning program to decide whether they correspond to a person or not.

Because the histogram weights correspond to the magnitudes of the gradients, sharper contrasts, typically found on the edges of objects, or in this case people, will have a greater impact on the histograms. The machine learning program used to decide whether the histograms correspond to a person is a Support Vector Machine, which is built into the functionality of OpenCV. While this strategy does detect a person at a distance, it does not perform as well when a person is up close. Also, due to the nature of this method, it can be harder to detect people if their clothing blends into the background of the frame.

SSD Person Detection

In this strategy we used a currently extremely popular deep learning network known as an SSD (Single Shot Multi-Box Detector). This kind of model takes an image as input, and then will output the regression of location and score about the target objects. In our case, to extract the faces of people in an image, we attempted to use a pre-trained MobileNet SSD model to detect anything that matched the "person" label in an image. After the SSD model runs, you are given a large list of the outputs and what each output corresponds to. The initial problem here is that an extremely in-depth model like MobileNet has a huge number of outputs. In the model we used, it detects well over 90 different classes of objects in images (including: cars, boats, birds, cats, dogs, traffic lights, person, and more). Filtering the detections that were not "person" is an easy task and takes a simple conditional if-statement.

In every detection, this model also provides you with the coordinates of the detections "bounding box". This allows for the drawing of boxes over the detections, as well as any further processing. In our case, once we had a "person" detected, we also needed to isolate the face. After this detection, we assumed that the head would be in the top third of the "person" detection, and simply dropped a square down from the top of the detection, where each side of the square was equal to the width of the detection.

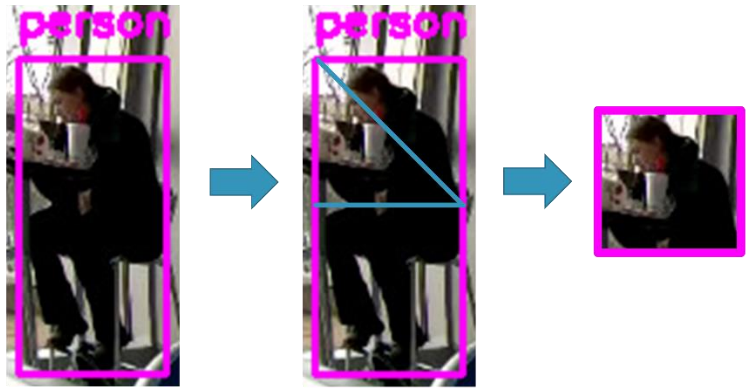


Figure 7: squaring process to go from a "person" detection to a "head" for the machine learning model

This was the most robust way to approach this problem, but even then, it resulted in many issues. For example, during walking movement, the "person" detection bounding box grows and shrinks resulting head detection we create from the person detection. This creates points of inaccuracy, since our model is trained from faces, not from larger pictures that potentially capture more than one face. This also creates issues where you can get very poor processed video where you are left with the resulting quickly growing and shrinking head detections. Finally, this also does not guarantee that the face is captured, since a "person" is detected from all angles, including from behind. The following image highlights these weaknesses:

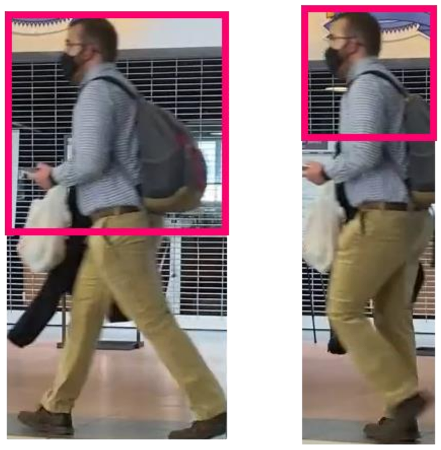


Figure 8: The stride problem with the SSD strategy

MTCNN Facial Detection

The final method we selected was this MTCNN facial detection strategy. In this method, we simply made use of the pretrained MTCNN machine learning model, which is trained to detect faces.

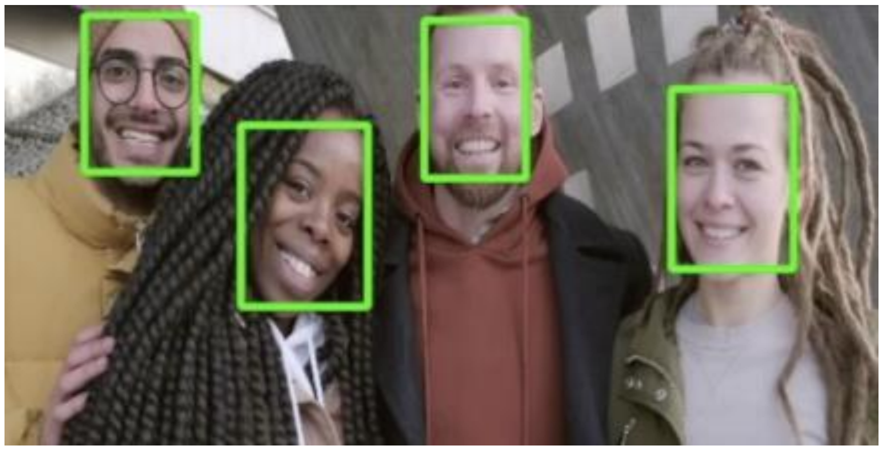


Figure 9: MTCNN facial detection output
Source: https://towardsdatascience.com/9fa646ad7c31

MTCNN is a well-trained model that is available to be simply installed as a Python module. After easily configuring the module, we can call the MTCNN class in our code and search an image for any faces. Like the SSD, you are then given a list of the detections, their confidence levels, and finally the bounding boxes for each detection. After looping through each detection, if a detection passes our set minimum level of confidence (approx. 80%) it is then passed onto the mask classifier to see if the face is wearing a mask.

One problem is that even though this detects faces extremely well, when someone wears a mask, they cover up many critical features of their face. This leads to MTCNN having a lot of difficulty when it comes to detecting masked faces. But one thing we noticed is that when an image has higher resolution, this downside is mostly neglected due to other facial features being more recognizable, like the eyes.

However, another main issue with this strategy is the time it takes to detect faces. MTCNN is not very efficient, especially compared to the SSD MobileNet model we used in the previous strategy. While running in a video, MTCNN takes up to almost 300ms to process a frame for faces. Our machine learning model can determine if faces are wearing a mask in about 30ms (it's worth noting that this speed is the same for processing up to 5 faces at the same time). This causes a huge delay in processing since in total, 330ms results in a speed of 3 frames per second. While MTCNN works decently, this speed in our final product is not very good.

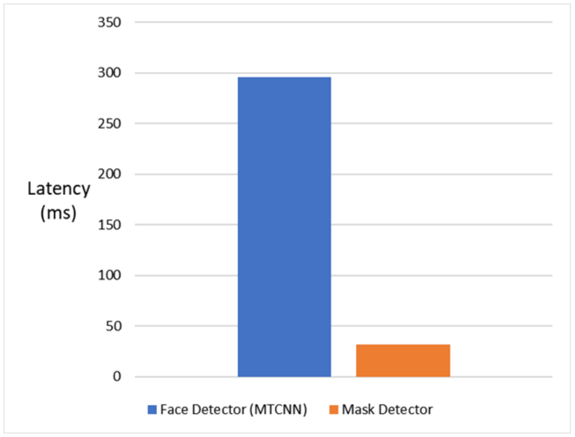


Figure 10: Comparing face detection, and mask detection time

CONCLUSION

After all our efforts we were able to progress this overall project significantly. After starting with no knowledge of machine learning and wireless camera streaming, we came a long way to learn more about both these topics and will hopefully be able to maybe leverage these machine learning skills further down the line. Although our final product performs relatively slowly to what we'd like to, we were able to build a firm groundwork for a solution in this domain. Unfortunately, due to having to work under the constrains of COVID-19, we were unable to hit key landmarks in producing an optimized final product. But, after our efforts we have arrived to what we believe to be the best answers for furthering both hardware and software aspects of the project.

For the software end, it seems the best course of action would be to investigate training our very own SSD model. We already have to data available to begin this, from the over 2,220 images we've collected. By pursuing this strategy, we'll be able to put together everything we have learned so far. There are methods available to transfer learn pre-existing SSD models, and by pursuing this strategy, the need for a middleman step for capturing faces will no longer be required. Assumedly though, this newly trained SSD model will have its own latency that will most likely be a bit higher than the current mask classifier. This is to be expected since an SSD model will have to rapidly scan an entire image for any possible masked or unmasked faces in the image.

As for hardware, our research has led us through many possible options to pursue. One idea that fits excellently with the project is the use of a Raspberry Pi. Using a Raspberry Pi would be a great choice due to its low cost compared to other commercial cameras. Also, there is growing support for embedding machine learning models onto Raspberry Pi. So, in an ideal scenario, a possible future final product could include our entire program on a Raspberry Pi camera system.

REFERENCES

"IEEE Recommended Practice for Assessing the Impact of Autonomous and Intelligent Systems on Human Well-Being," in IEEE Std 7010-2020 , vol., no., pp.1-96, 1 May 2020, doi: 10.1109/IEEESTD.2020.9084219.