Face Mask Detector

Final Report

Amir Fatemi
October 30, 2020

Project Proposal

Problem: Face mask detector or in general object detection is a technology using computer vision and image processing which detects humans and objects and has the ability to classify several objects in an image. It's use cases span a wide range including security industry, self driving cars, face detection to the problem at hand which can detect a facial mask.

Client: The potential clients of a face mask detector can be any organization who would want to make sure that a face mask is properly worn before permission to enter a building, specially at the time of COVID-19.

Data: The data set for this purpose is obtained from Kaggle, which contains 632 images of people wearing face masks, in jpg format, along with the self annotated files in xml format. The data set can be found `here`.

Solution: The approach to this problem is based on utilizing Convolutional Neural Network to build a classification model with the ability of classifying faces with/without mask and combining it with a pre-built face-detection model using OpenCV. In the end, the TensorFlow Object Detection API will be used to come up with a customized Face Mask Detector with labeled images.

Deliverables:

- Codes for:
    1. Building the Deep learning model with Keras and OpenCV
    2. TensorFlow Object Detection API model
- Project report
- Project presentation

Face Mask Detector utilizing Keras

In the first part of this project, I will step by step build a mask/no mask classifier with a combination of famous **ResNet50** deep learning model and several Fully Connected layers (FC) on top of it. After training the model on the training set I use the test set to evaluate the performance of the model. In order to classify the faces with labels, the primary requirement would be to detect the faces inside the picture in the first place. In the end, the performance of this model is contingent upon how good the face detector model can identify the faces in different situations.

Residual Networks (ResNets)

One challenging problem in training deep neural networks, are the vanishing/exploding gradients. ResNets take advantage of skip connections, which allows the model to feed the information from one layer to a layer much deeper in the network.
ResNets are built from smaller units called **Residual Blocks**. As shown in figure 1.

$$z^{[l+1]} = W^{[l+1]}a^{[l]} + b^{[l+1]}$$
$$a^{[l+1]} = g(z^{[l+1]})$$
$$z^{[l+2]} = W^{[l+2]}a^{[l+1]} + b^{[l+2]}$$
$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$$

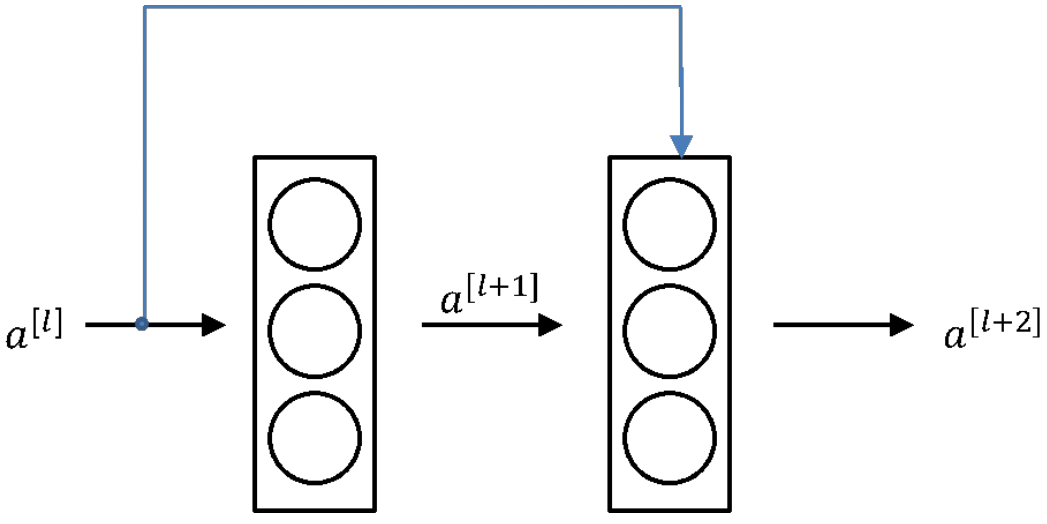

Figure 1: Depiction of one layer of Residual Block, feeding the information from $a^{[l]}$ to two layers further.

One reason for why ResNest work can be demonstrated in the below formulation:

$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]}) = g(W^{[l+2]}a^{[l+1]} + b^{[l+2]} + a^{[l]})$$

when using deep networks with regularization, $W^{l+2} \approx 0$ and the formula above becomes:

$$a^{[l+2]} = g(a^{[l]}) = a^{[l]}$$

so the formula above implies that going deeper in the network would not hurt the performance of the network, and it turns out that it actually helps to build deeper network and achieve better performance. The model I used for this project is the ResNet-50 (figure 2).

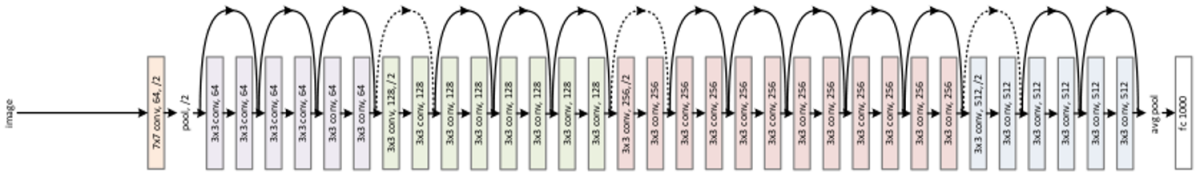

Figure 2: ResNet-50.

Transfer Learning using Pre-trained Models

In most practical applications, obtaining a large number of training data is not necessarily an easy task, so using models already trained on a big data set seems to be a good idea. In this case I used the ResNet-50 model which is pre-trained on ImageNet data. Since ImageNet does not necessarily contain the class of images that we want to use in our project, then most of the time we keep the model and try to keep the already trained weights as much as possible. In this case I froze the layers of the original model and added several fully connected (FC) layers to customize the model. The data set size used for this model is around 700 images of people with facial mask and 700 images of people without mask and 20% of the data has been used to validate the model performance. Also data augmentation techniques has been applied to the data to account for rotations, size, horizontal flips, etc. After training on 10 epochs, the model reaches **97.5%** accuracy on training and **100%** accuracy on the test set. The performance of the model on accuracy and loss are given below (figures 3 and 4).

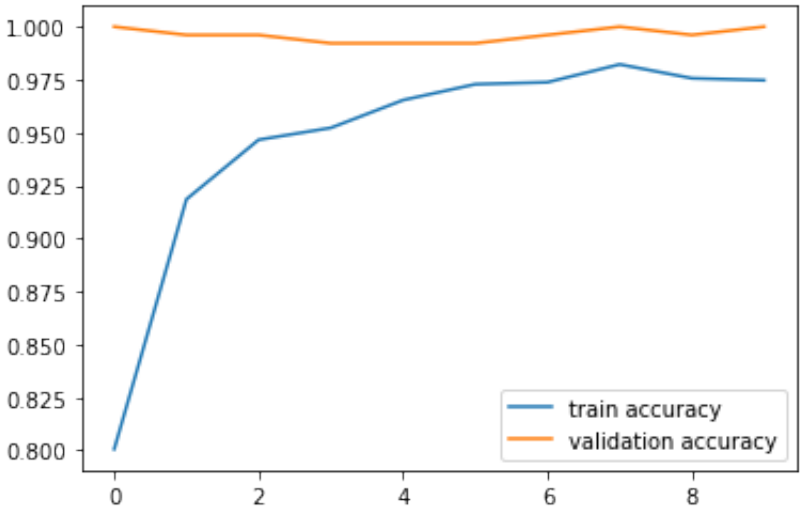

Figure 3: Training and test set accuracy on the data set.

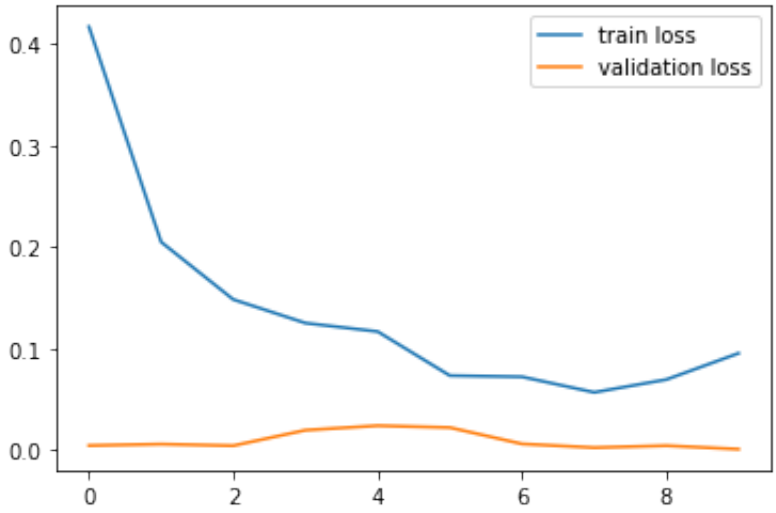

Figure 4: Training and test set loss on the data set.

Face Detection model

Now that the ResNet-50 model has been trained to classify masked faces from non-masked ones, we yet need another model to detect the faces and gives us the location of the faces in the image. I have used OpenCV api and utilized it's already trained face detection models to use in this assignment. the model I used is called Single Shot Detection (SSD) model to detect the faces.
SSD model builds a feature map of the image and in each graph generates boxes with different aspect ratios and during the prediction, the model scores different boxes for the presence of object class instances in those boxes in the graph depending on the orientaiton and location of the object in the image. SSD is also a very fast model that can perform well in real time (figure 5).

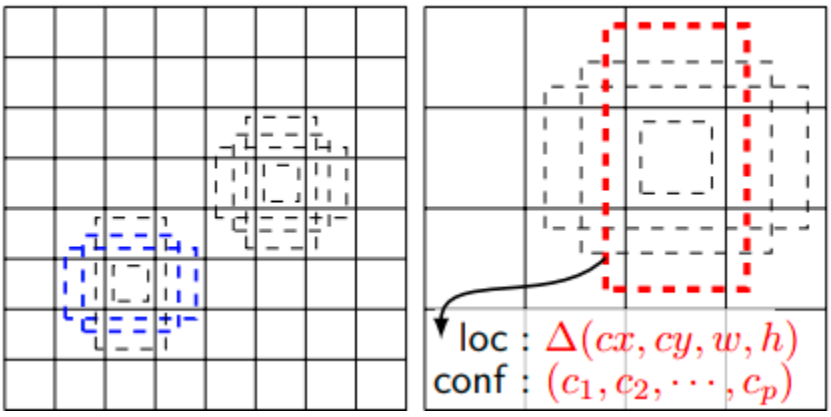

Figure 5: SSD, 8×8 and 4×4 feature maps.

There exists other ready-to-use models for face detection like face detection using Haar features, which can be used for similar projects as well.
In the end, the detected face is given as input to the ResNet-50 classification model that we built previously and the output will be either face with mask or without mask. Below is the end result of the model. One thing worth mentioning is that by using more classification images, the binary classification model can be improved, but the model works as long as the face detection model is able to detect the faces.

Figure 6: Both faces have been detected and classified correctly.

Figure 7: One of the faces has not been detected and the other is classified incorrectly.

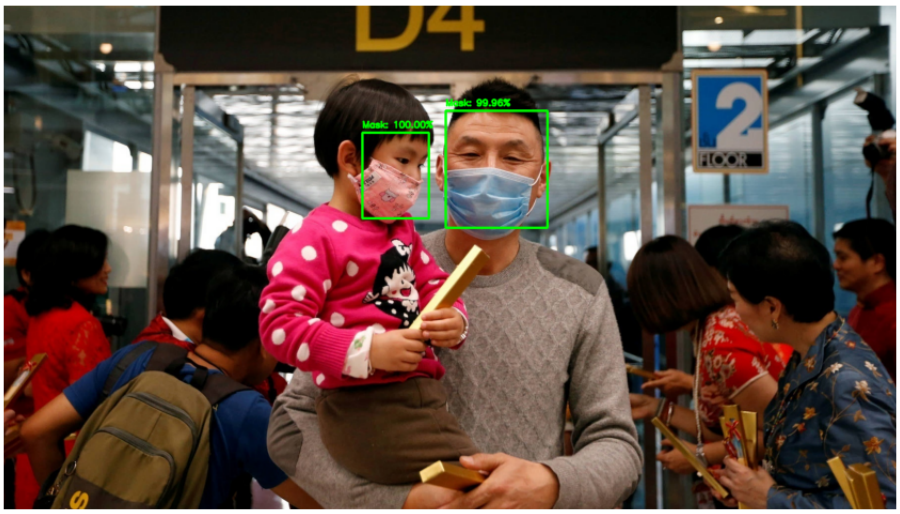

Figure 8: Both faces have been detected and classified correctly.

Figure 9: In this image of a crowded place, the model was incapable of detecting any faces.

Figure 10: Model detected the masked face correctly.

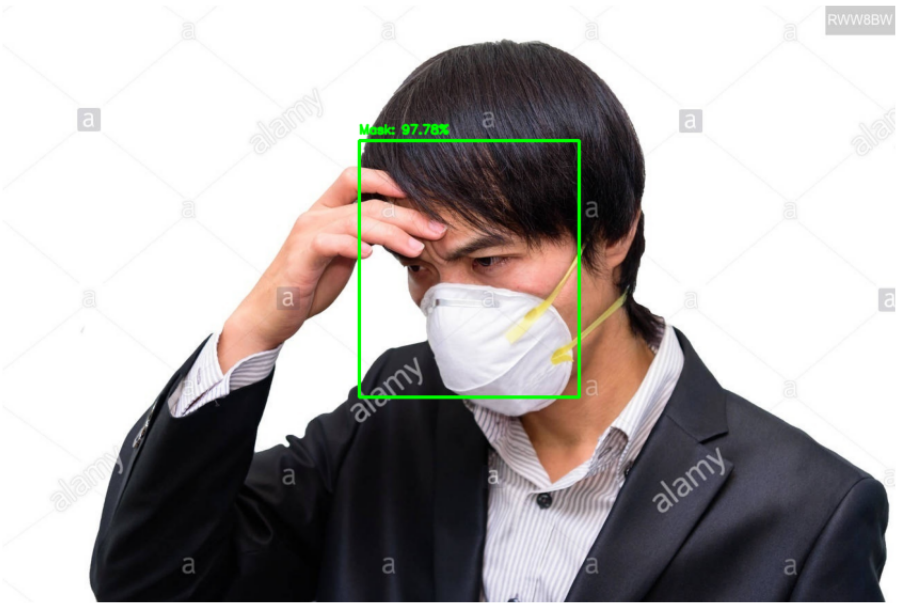

Figure 11: The masked face has been detected correctly.

This model is somewhat simplistic cause it relies heavily on the ability of face detection model to detect the masked faces, which is not always the case and has it's shortcomings. In the next chapter I will use another approach which will give a more robust answer to the same problem.

TensorFlow Object Detection API

I have used TensorFlow2 Object Detection API for this task. there are generally several steps to prepare our own customized Object Detection model:

- Preparing and labeling the data set
- Generating TensorFlow records
- Training the model

Preparing and Labeling the Data Set

The bulk of the data I used in this task can be found Here.
The data set I used to train the model contains 165 images of peopel with or without face mask, and the annotating of the images is done using Labelimg software. The outcome of the image labeling is an xml file for each image which contains the label and coordinates of each face in the image.

Figure 12: Image annotation using labelimg.

After annotating the images, it is time to build a file named label_map · pbtxt which can be found

in **workspace\training_demo\annotations** and contains the names of the two classes we are trying to detect (mask and no_mask in this case).

Generating TensorFlow Records

Now that the annotation files, we can use the script in **scripts\preprocessing** to convert our annotated files to TFRecord format.

Training the Model

There are many pre-trained models we can choose from and they can be found in TensorFlow Model Zoo. The model I used is called SSD MobileNet V2 FPNLite 640×640. The model can be found in **workspace\training_demo\pre-trained-models**. Another thing that remains is to create a directory called **models** and copy the pipeline $\cdot$ config file from pre-trained-models into it.

Make sure to change the config file according to the needs of the question at hand. now to train the model, we only need to run the file model_main_tf2 $\cdot$ py which is located in **training_demo** folder.

Now that the training is finished, in order to use the model we need to export it first, using the file exporter_main_v2 $\cdot$ py which is also located in **training_demo** folder. I tried not to go in too much detail here but all the instructions can be found in TensorFlow 2 Object Detection API tutorial . To end this report, I will show several images and how the model performed on them compare to the former model I built using Keras.

Figure 13: Correctly detected masked face.

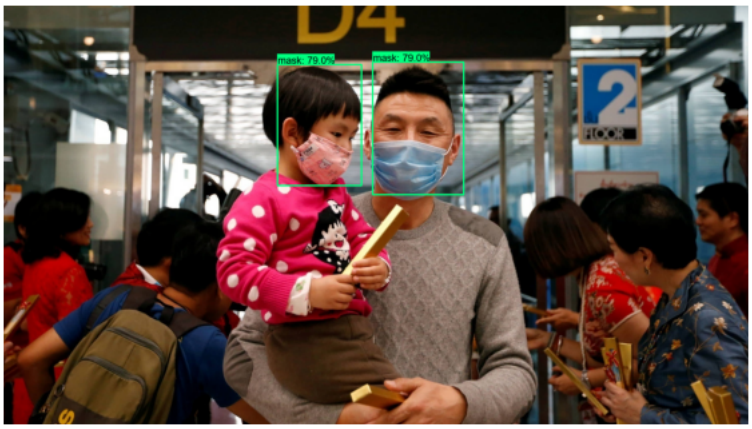

Figure 14: Correctly detected masked faces.

Figure 15: Correctly detected masked faces with low probability.

Figure 16: Wrongly detected masked and non-masked faces.

Figure 17: One masked face has been detected correctly.

Figure 18: Masked face is detected correctly.

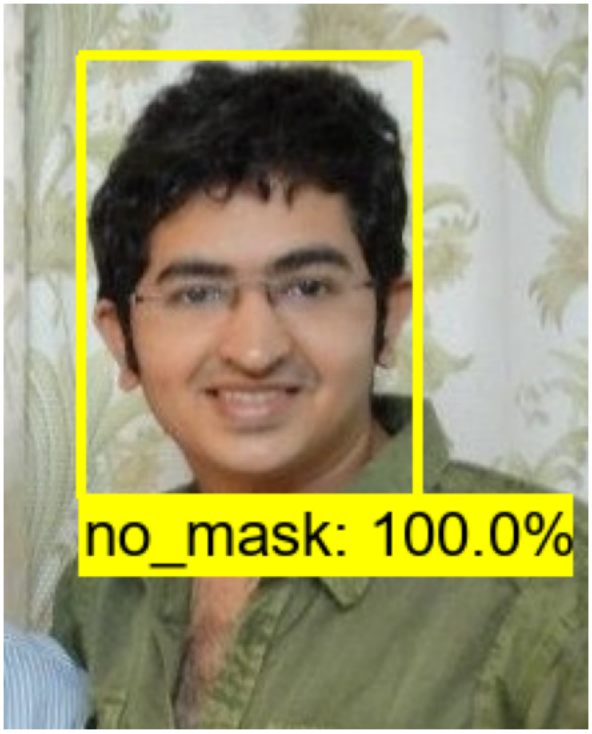

Figure 19: Non-masked face has been detected correctly.

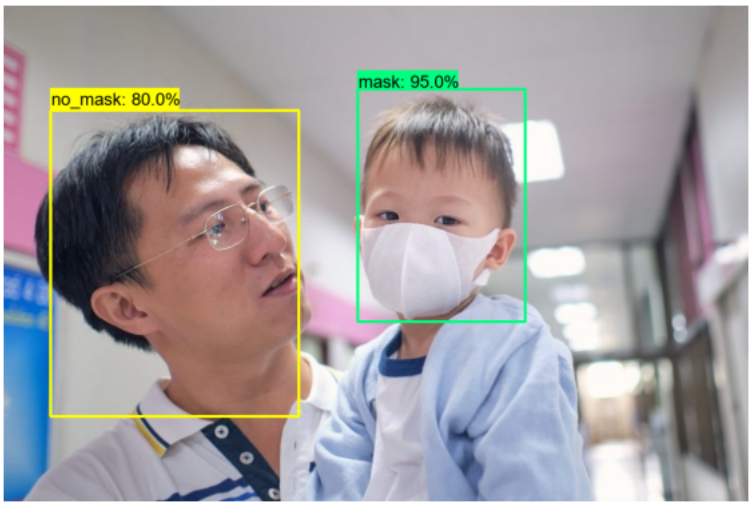

Figure 20: Both masked and non-masked faces are detected correctly.

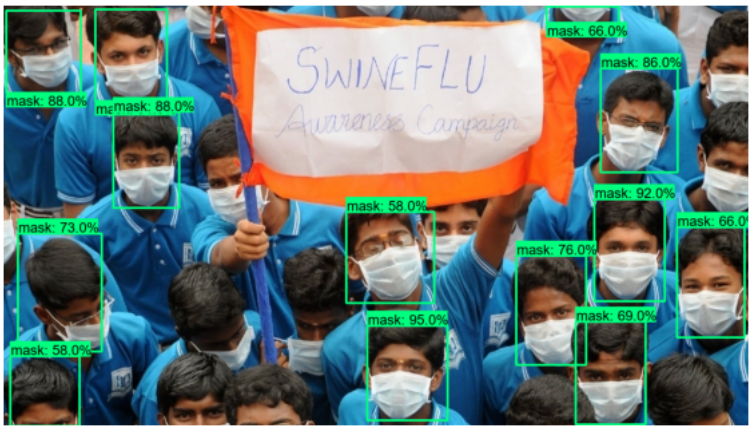

Figure 21: Almost all the masked faces have been detected with high probability.

In order to improve the performance of the model, one could simply add more and more images and include images with different and different face orientation.