**from** **google.colab** **import** drive

drive.mount('/content/MyDrive')

path = '/content/MyDrive/My Drive/Colab Notebooks/Face-Mask-Detection-master/'

zip\_path = path + 'face-mask-detector.zip'

**from** **zipfile** **import** ZipFile

**with** ZipFile(zip\_path, 'r') **as** z:

z.extractall()

print("Training zip extraction done!")

%cd /content/face-mask-detector

!ls

*# USAGE*

*# python train\_mask\_detector.py --dataset dataset*

*# import the necessary packages*

**from** **tensorflow.keras.preprocessing.image** **import** ImageDataGenerator

**from** **tensorflow.keras.applications** **import** MobileNetV2

**from** **tensorflow.keras.layers** **import** AveragePooling2D

**from** **tensorflow.keras.layers** **import** Dropout

**from** **tensorflow.keras.layers** **import** Flatten

**from** **tensorflow.keras.layers** **import** Dense

**from** **tensorflow.keras.layers** **import** Input

**from** **tensorflow.keras.models** **import** Model

**from** **tensorflow.keras.optimizers** **import** Adam

**from** **tensorflow.keras.applications.mobilenet\_v2** **import** preprocess\_input

**from** **tensorflow.keras.preprocessing.image** **import** img\_to\_array

**from** **tensorflow.keras.preprocessing.image** **import** load\_img

**from** **tensorflow.keras.utils** **import** to\_categorical

**from** **sklearn.preprocessing** **import** LabelBinarizer

**from** **sklearn.model\_selection** **import** train\_test\_split

**from** **sklearn.metrics** **import** classification\_report

**from** **imutils** **import** paths

**import** **matplotlib.pyplot** **as** **plt**

**import** **numpy** **as** **np**

**import** **argparse**

**import** **os**

*# initialize the initial learning rate, number of epochs to train for,*

*# and batch size*

INIT\_LR = 1e-4

EPOCHS = 20

BS = 32

*# grab the list of images in our dataset directory, then initialize*

*# the list of data (i.e., images) and class images*

print("[INFO] loading images...")

image\_paths = list(paths.list\_images('/content/face-mask-detector/dataset'))

data = []

labels = []

*# loop over the image paths*

**for** imagePath **in** image\_paths:

*# extract the class label from the filename*

label = imagePath.split(os.path.sep)[-2]

*# load the input image (224x224) and preprocess it*

image = load\_img(imagePath, target\_size=(224, 224))

image = img\_to\_array(image)

image = preprocess\_input(image)

*# update the data and labels lists, respectively*

data.append(image)

labels.append(label)

*# convert the data and labels to NumPy arrays*

data = np.array(data, dtype="float32")

labels = np.array(labels)

*# perform one-hot encoding on the labels*

l\_bin= LabelBinarizer()

labels = l\_bin.fit\_transform(labels)

labels = to\_categorical(labels)

*# partition the data into training and testing splits using 75% of*

*# the data for training and the remaining 25% for testing*

(trainX, testX, trainY, testY) = train\_test\_split(data, labels,

test\_size=0.20, stratify=labels, random\_state=42)

*# construct the training image generator for data augmentation*

data\_aug= ImageDataGenerator(

rotation\_range=20,

zoom\_range=0.15,

width\_shift\_range=0.2,

height\_shift\_range=0.2,

shear\_range=0.15,

horizontal\_flip=**True**,

fill\_mode="nearest")

*# load the MobileNetV2 network, ensuring the head FC layer sets are*

*# left off*

Model\_base = MobileNetV2(weights="imagenet", include\_top=**False**,

input\_tensor=Input(shape=(224, 224, 3)))

*# construct the head of the model that will be placed on top of the*

*# the base model*

headModel = Model\_base.output

headModel = AveragePooling2D(pool\_size=(7, 7))(headModel)

headModel = Flatten(name="flatten")(headModel)

headModel = Dense(128, activation="relu")(headModel)

headModel = Dropout(0.5)(headModel)

headModel = Dense(2, activation="softmax")(headModel)

*# place the head FC model on top of the base model (this will become*

*# the actual model we will train)*

model = Model(inputs= Model\_base.input, outputs=headModel)

*# loop over all layers in the base model and freeze them so they will*

*# \*not\* be updated during the first training process*

**for** layer **in** Model\_base.layers:

layer.trainable = **False**

*# compile our model*

print("[INFO] compiling model...")

opt = Adam(lr=INIT\_LR, decay=INIT\_LR / EPOCHS)

model.compile(loss="binary\_crossentropy", optimizer=opt,

metrics=["accuracy"])

*# train the head of the network*

print("[INFO] training head...")

H = model.fit(

aug.flow(trainX, trainY, batch\_siz=BS),

steps\_per\_epoch=len(trainX) // BS,

validation\_data=(testX, testY),

validation\_steps=len(testX) // BS,

epochs=EPOCHS)

*# make predictions on the testing set*

print("[INFO] evaluating network...")

pred\_Indx = model.predict(testX, batch\_size=BS)

*# for each image in the testing set we need to find the index of the*

*# label with corresponding largest predicted probability*

pred\_Indx = np.argmax(pred\_Indx , axis=1)

*# show a nicely formatted classification report*

print(classification\_report(testY.argmax(axis=1), pred\_Indx ,

target\_names=lb.classes\_))

*# plot the training loss and accuracy*

N = EPOCHS

plt.style.use("ggplot")

plt.figure()

plt.plot(np.arange(0, N), H.history["loss"], label="train\_loss")

plt.plot(np.arange(0, N), H.history["val\_loss"], label="val\_loss")

plt.plot(np.arange(0, N), H.history["accuracy"], label="train\_acc")

plt.plot(np.arange(0, N), H.history["val\_accuracy"], label="val\_acc")

plt.title("Training Loss and Accuracy")

plt.xlabel("Epoch #")

plt.ylabel("Loss/Accuracy")

plt.legend(loc="lower left")

*# plot the training loss and accuracy*

N = EPOCHS

plt.style.use("ggplot")

plt.figure()

plt.plot(np.arange(0, N), H.history["loss"], label="train\_loss")

plt.plot(np.arange(0, N), H.history["val\_loss"], label="val\_loss")

plt.plot(np.arange(0, N), H.history["accuracy"], label="train\_acc")

plt.plot(np.arange(0, N), H.history["val\_accuracy"], label="val\_acc")

plt.title("Training Loss and Accuracy")

plt.xlabel("Epoch #")

plt.ylabel("Loss/Accuracy")

plt.legend(loc="lower left")

*# USAGE*

*# python detect\_mask\_video.py*

*# import the necessary packages*

**from** **tensorflow.keras.applications.mobilenet\_v2** **import** preprocess\_input

**from** **tensorflow.keras.preprocessing.image** **import** img\_to\_array

**from** **tensorflow.keras.models** **import** load\_model

**from** **imutils.video** **import** VideoStream

**import** **numpy** **as** **np**

**import** **argparse**

**import** **imutils**

**import** **time**

**import** **cv2**

**import** **os**

**def** detect\_and\_predict\_mask(frame, faceNet, maskNet):

*# grab the dimensions of the frame and then construct a blob*

*# from it*

(h, w) = frame.shape[:2]

blob = cv2.dnn.blobFromImage(frame, 1.0, (300, 300),

(104.0, 177.0, 123.0))

*# pass the blob through the network and obtain the face detections*

faceNet.setInput(blob)

detections = faceNet.forward()

*# initialize our list of faces, their corresponding locations,*

*# and the list of predictions from our face mask network*

faces = []

locs = []

predicts = []

*# loop over the detections*

**for** i **in** range(0, detections.shape[2]):

*# extract the confidence (i.e., probability) associated with*

*# the detection*

confidence = detections[0, 0, i, 2]

*# filter out weak detections by ensuring the confidence is*

*# greater than the minimum confidence*

**if** confidence > 0.5:

*# compute the (x, y)-coordinates of the bounding box for*

*# the object*

box = detections[0, 0, i, 3:7] \* np.array([w, h, w, h])

(startX, startY, endX, endY) = box.astype("int")

*# ensure the bounding boxes fall within the dimensions of*

*# the frame*

(startX, startY) = (max(0, startX), max(0, startY))

(endX, endY) = (min(w - 1, endX), min(h - 1, endY))

*# extract the face ROI, convert it from BGR to RGB channel*

*# ordering, resize it to 224x224, and preprocess it*

face = frame[startY:endY, startX:endX]

face = cv2.cvtColor(face, cv2.COLOR\_BGR2RGB)

face = cv2.resize(face, (224, 224))

face = img\_to\_array(face)

face = preprocess\_input(face)

*# add the face and bounding boxes to their respective*

*# lists*

faces.append(face)

locs.append((startX, startY, endX, endY))

*# only make a predictions if at least one face was detected*

**if** len(faces) > 0:

*# for faster inference we'll make batch predictions on \*all\**

*# faces at the same time rather than one-by-one predictions*

*# in the above `for` loop*

faces = np.array(faces, dtype="float32")

predicts = maskNet.predict(faces, batch\_size=32)

*# return a 2-tuple of the face locations and their corresponding*

*# locations*

**return** (locs, predicts)

faceNet=cv2.dnn.readNet('/content/face-mask-detector/face\_detector/deploy.prototxt','/content/face-mask-detector/face\_detector/res10\_300x300\_ssd\_iter\_140000.caffemodel')

**import** **base64**

**import** **html**

**import** **io**

**import** **time**

**from** **IPython.display** **import** display, Javascript

**from** **google.colab.output** **import** eval\_js

**import** **numpy** **as** **np**

**from** **PIL** **import** Image

**import** **cv2**

**def** start\_input():

js = Javascript('''

var video;

var div = null;

var stream;

var captureCanvas;

var imgElement;

var labelElement;

var pendingResolve = null;

var shutdown = false;

function removeDom() {

stream.getVideoTracks()[0].stop();

video.remove();

div.remove();

video = null;

div = null;

stream = null;

imgElement = null;

captureCanvas = null;

labelElement = null;

}

function onAnimationFrame() {

if (!shutdown) {

window.requestAnimationFrame(onAnimationFrame);

}

if (pendingResolve) {

var result = "";

if (!shutdown) {

captureCanvas.getContext('2d').drawImage(video, 0, 0, 512, 512);

result = captureCanvas.toDataURL('image/jpeg', 0.8)

}

var lp = pendingResolve;

pendingResolve = null;

lp(result);

}

}

async function createDom() {

if (div !== null) {

return stream;

}

div = document.createElement('div');

div.style.border = '2px solid black';

div.style.padding = '3px';

div.style.width = '100%';

div.style.maxWidth = '600px';

document.body.appendChild(div);

const modelOut = document.createElement('div');

modelOut.innerHTML = "<span>Status:</span>";

labelElement = document.createElement('span');

labelElement.innerText = 'No data';

labelElement.style.fontWeight = 'bold';

modelOut.appendChild(labelElement);

div.appendChild(modelOut);

video = document.createElement('video');

video.style.display = 'block';

video.width = div.clientWidth - 6;

video.setAttribute('playsinline', '');

video.onclick = () => { shutdown = true; };

stream = await navigator.mediaDevices.getUserMedia(

{video: { facingMode: "environment"}});

div.appendChild(video);

imgElement = document.createElement('img');

imgElement.style.position = 'absolute';

imgElement.style.zIndex = 1;

imgElement.onclick = () => { shutdown = true; };

div.appendChild(imgElement);

const instruction = document.createElement('div');

instruction.innerHTML =

'<span style="color: red; font-weight: bold;">' +

'When finished, click here or on the video to stop this demo</span>';

div.appendChild(instruction);

instruction.onclick = () => { shutdown = true; };

video.srcObject = stream;

await video.play();

captureCanvas = document.createElement('canvas');

captureCanvas.width = 512; //video.videoWidth;

window.requestAnimationFrame(onAnimationFrame);

return stream;

}

async function takePhoto(label, imgData) {

if (shutdown) {

removeDom();

shutdown = false;

return '';

}

var preCreate = Date.now();

stream = await createDom();

var preShow = Date.now();

if (label != "") {

labelElement.innerHTML = label;

}

if (imgData != "") {

var videoRect = video.getClientRects()[0];

imgElement.style.top = videoRect.top + "px";

imgElement.style.left = videoRect.left + "px";

imgElement.style.width = videoRect.width + "px";

imgElement.style.height = videoRect.height + "px";

imgElement.src = imgData;

}

var preCapture = Date.now();

var result = await new Promise(function(resolve, reject) {

pendingResolve = resolve;

});

shutdown = false;

return {'create': preShow - preCreate,

'show': preCapture - preShow,

'capture': Date.now() - preCapture,

'img': result};

}

''')

display(js)

**def** take\_photo(label, img\_data):

data\_img = eval\_js('takePhoto("**{}**", "**{}**")'.format(label, img\_data))

**return** data\_img

**def** js\_reply\_to\_image(js\_reply):

*"""*

*input:*

*js\_reply: JavaScript object, contain image from webcam*

*output:*

*image\_array: image array RGB size 512 x 512 from webcam*

*"""*

jpeg\_bytes\_imgweb = base64.b64decode(js\_reply['img'].split(',')[1])

imageweb\_PIL = Image.open(io.BytesIO(jpeg\_bytes\_imgweb))

imageweb\_array = np.array(imageweb\_PIL)

**return** imageweb\_array

**import** **imutils**

start\_input()

label\_html = 'Capturing...'

imgweb\_data = ''

count = 0

**from** **google.colab.patches** **import** cv2\_imshow

**while** **True**:

js\_reply = take\_photo(label\_html, imgweb\_data)

**if** **not** js\_reply:

**break**

image = js\_reply\_to\_image(js\_reply)

*# grab the frame from the threaded video stream and resize it*

*# to have a maximum width of 400 pixels*

frame = image

v=**True**

**if** v == **True**:

frame = imutils.resize(frame, width=400)

*# detect faces in the frame and determine if they are wearing a*

*# face mask or not*

(locs, predicts) = detect\_and\_predict\_mask(frame, faceNet, model)

**for** (box, pred) **in** zip(locs, predicts):

*# unpack the bounding box and predictions*

(startX, startY, endX, endY) = box

(mask, withoutMask) = pred

*# determine the class label and color we'll use to draw*

*# the bounding box and text*

label = "Mask" **if** mask > withoutMask **else** "No Mask"

color = (0, 255, 0) **if** label == "Mask" **else** (0, 0, 255)

*# include the probability in the label*

label = "**{}**: **{:.2f}**%".format(label, max(mask, withoutMask) \* 100)

*# display the label and bounding box rectangle on the output*

*# frame*

frame=cv2.putText(frame, label, (startX, startY - 10),cv2.FONT\_HERSHEY\_SIMPLEX, 0.45, color, 2)

frame=cv2.rectangle(frame, (startX, startY), (endX, endY), color, 2)

*# show the output frame*

cv2\_imshow(frame)