A REPORT

ON

DETECTION OF FACE-MASK WORN BY A PERSON

BY

HITESH ARYAN ACHARYA 2018AAPS0384H

AT

SMART i-ELECTRONICS SYSTEMS, PUNE

A Practice School-1 Station of

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

June, 2020

A REPORT

ON

DETECTION OF FACE-MASK WORN BY A PERSON

BY

HITESH ARYAN ACHARYA 2018AAPS0384H

AT

SMART i-ELECTRONICS SYSTEMS, PUNE

A Practice School-1 Station of

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

June, 2020

ACKNOWLEDGEMENTS

I am grateful to my Institute and the Practice School Division for organizing the Practice School-1 and setting up avenues for the same. I am deeply indebted to the Smart I - Electronics Systems for providing me with an opportunity to work with them and mentoring me. I would like to thank my instructor Prof. Rejesh N.A. for his constant guidance, clarification of my doubts and providing me with lecture videos and other invaluable learning material. I would also like to thank my mentor Mr. Pankaj Zanwar for explaining the details of the problem statement, helping me with the approach and providing me with the required material. Lat but not the least, I would like to thank my family and friends who have supported me throughout.

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI

(RAJASTHAN)

Practice School Division

Station: Smart i-Electronics Systems **Centre: Pune**

Duration: 6 weeks **Date of start: 18th May, 2020**

Date of submission: 26th June, 2020

Title of the project: Detection of a Face-mask worn by a Person

ID No./Name(s)/ Discipline(s)/ of the student(s) **2018AAPS0384H, Hitesh Aryan Acharya, B.E. ECE**

Name(s) and designation(s) of the expert(s): **Mr. Vaibhav, Employee**
Mr. Pankaj Zanwar, Director

Name(s) of the PS Faculty: **Mr. Rejesh N. A.**

Key Words: Neural Networks, Propagation, Dataset, EPOCH, Optimization

Project Areas: Deep Learning, OpenCV

Abstract:

The following project involves building a neural network model which can identify and detect whether a person is wearing a face-mask using concepts like convolutions, data augmentation etc. It also requires understanding of TensorFlow/Keras and OpenCV.

Signature(s) of Student(s) Signature of PS Faculty

Date Date

Table of Contents

CHAPTER 1

Why is face-mask detection required?

Coronavirus as we all are aware is among the most contagious viruses. It is hence required that we all follow certain guidelines like wearing a face mask whenever outside. Social, religious and academic institutions, where there is a large influx of people, requires a robust technique to detect whether all are wearing a face-mask. Hence it is imperative that we develop a computer algorithm to do the same. This eliminates additional manpower and human error.

Why use Deep-Learning?

Deep learning uses multiple layers of neural networks to progressively extract higher level features from the raw input. For example, in image processing, lower layers may identify edges, while higher layers may identify the concepts relevant to a human such as digits or letters or faces.

Algorithms such as deep learning tend to perform better with more data. Living in such day and age where we have an abundance of data it is useful that we develop models which implement machine learning and deep learning techniques. The figure below shows how the performance of DL algorithms increase over amount of data compared to those of traditional learning algorithms.

Implementing a deep-learning algorithm to detect face-masks is very easy and effective. It can work on live-feed images and videos as well as can detect multiple face-masks at a time. It doesn't necessarily require a real dataset and yet still performs to the desired level. Most importantly, the code is very user friendly and can be tweaked to accommodate diverse parameters accordingly.

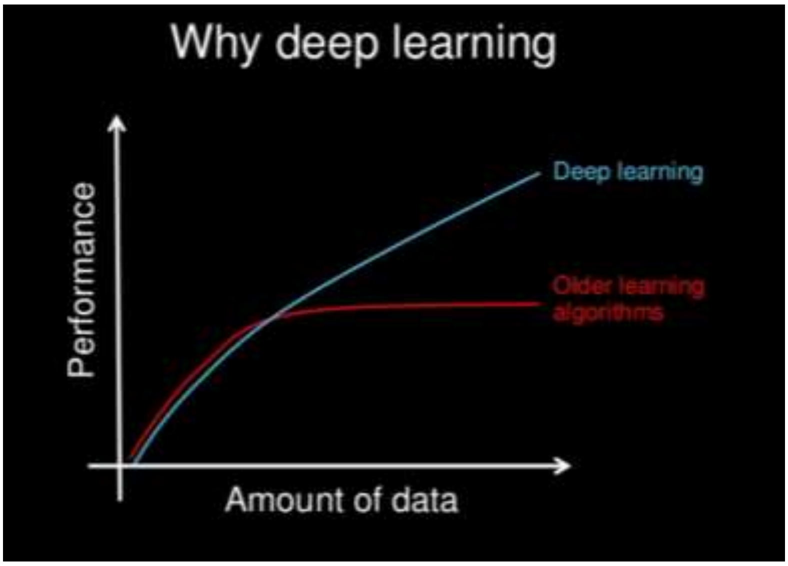


Figure 1- Deep Learning vs Traditional algorithms

CHAPTER 2

Convolutional Neural Networks

What are convolutions?

In image processing, convolutions are basically matrix operations on the pixel values to bring out a desired effect in the image. The name "convolutional neural network" indicates that the network employs a mathematical operation called convolution. Convolution is a specialized kind of linear operation. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

Why do we need convolutions?

Often images in the dataset aren't uniform and the characteristic properties of the image can be placed anywhere in the image. Hence we require a technique which can identify the important features of the image and make the image size uniform. Convolutions help condense the information present and reduce the size of the image at the same time. This pre-processing helps in reducing computation time. Pooling, Receptive Field, Weights etc. are some of the common terms in CNN which will be discussed later.

Operation	Kernel ω	Image result g(x,y)
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Gaussian blur 5 × 5 (approximation)	$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$	

Figure 2- Effect of convolutions on image/Source- Wikipedia

CHAPTER 3

3.1 Logistic Regression and Linear Regression

Linear regression is a linear approach to modelling the relationship between a dependent variable and one or more independent variables. In linear regression we try to fit a straight line which passes through most of the data points**. Logistic Regression** is predominantly used in classification. It is a model used to model the probability of a certain event. But unlike linear regression where we compute the slope of a straight line, in logistic regression we have a S-shaped curve which performs binary classification, i.e. 0 or 1.

In linear regression we compute the slope of the straight line using the least square fit method. Here we measure the distance between the points and the straight line and add the squares of these distances. The minimum value pertaining to this is the best fit straight line.

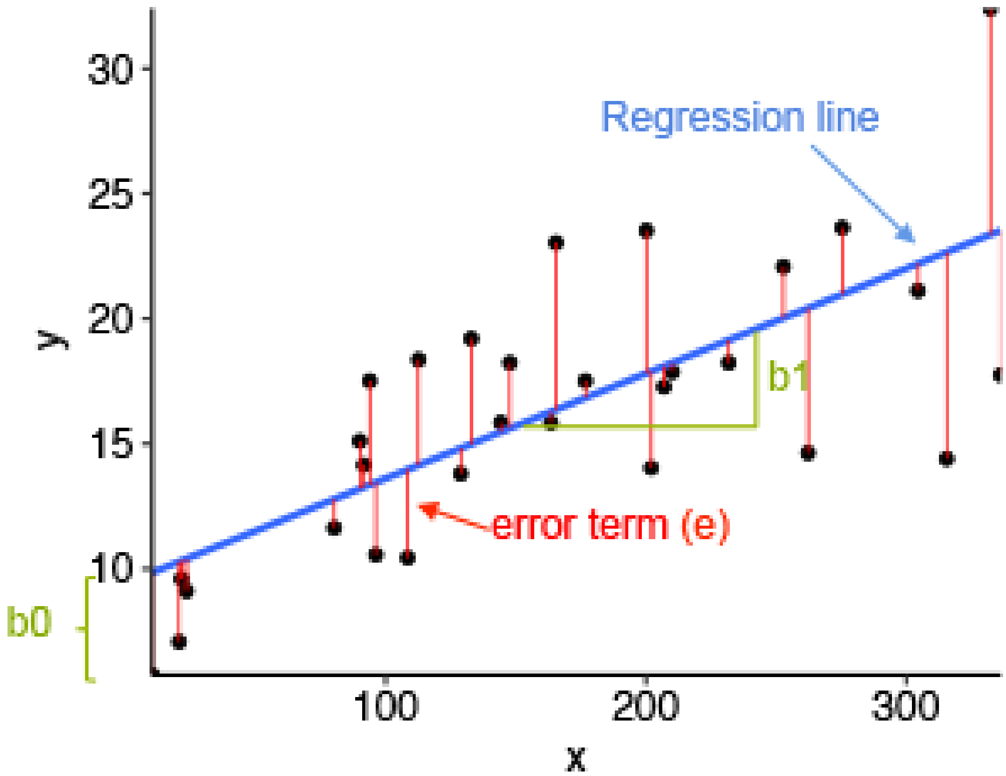


Figure 3- Linear Regression

In logistic regression we try to divide the data into two broad classes using maximum likelihood estimation. In a nutshell we try to pass the majority of the data points though the S-shaped curve.

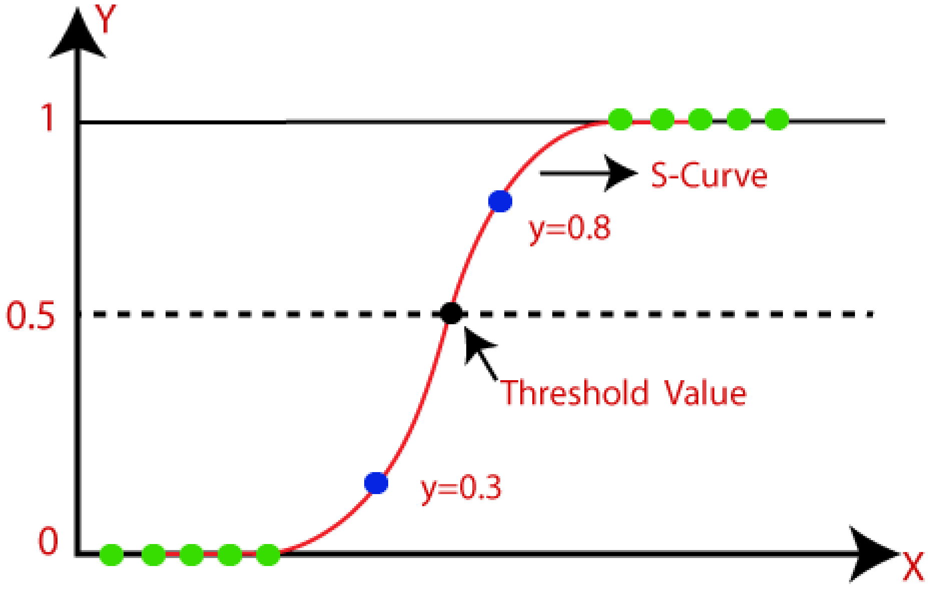


Figure 4- Logistic Regression

Such S-shaped curves are called logistic functions. Popular functions in machine learning include the sigmoid function and the hyperbolic tan.

sigmoid : $A = \frac{1}{1+e^{-x}}$

tanh(x) : $A = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$

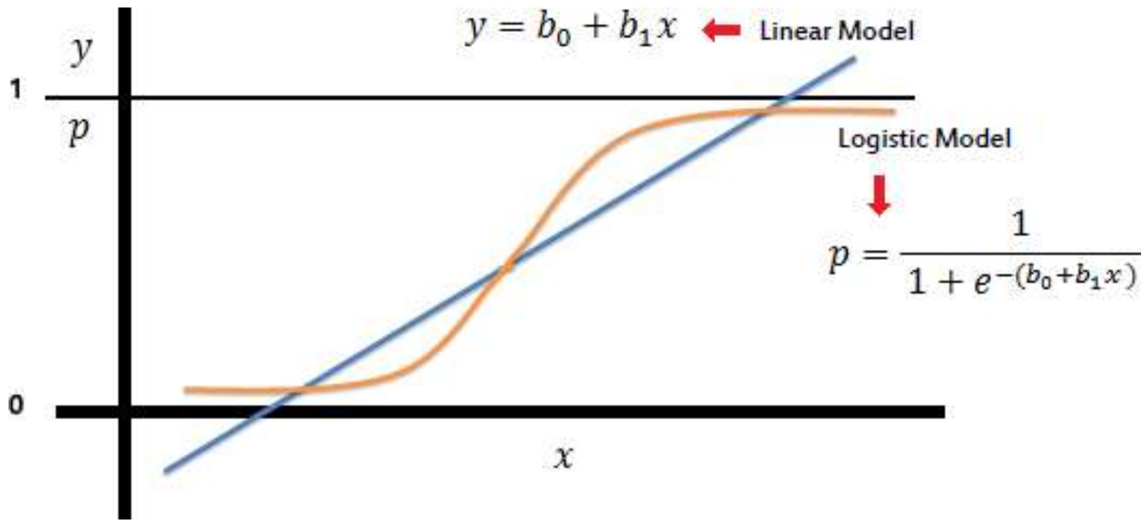


Figure 5- Logistic vs Linear Regression

Here we will be learning to classify whether an image is a cat image or a non-cat image using logistic regression. Cat-image is represented by [1] in the labels' dataset and a non-cat image is represented by [0]. We will be using the sigmoid function.

CHAPTER 4

LOGISTIC REGRESSION USING NEURAL NETWORK

4.1 IMPORTING THE NECESSARY LIBRARIES

```
import numpy as np
import matplotlib.pyplot as plt
import h5py
import scipy
from PIL import Image
from scipy import ndimage
from lr_utils import load_dataset

%matplotlib inline
```

Figure 6- Importing libraries

- Numpy is one of the most important library in Python. Using numpy we can compute array, vector and matrix operations with ease.
- Matplotlib is used for plotting graphs and images.
- H5py is important for binary data format.
- Python Imaging Library or PIL is a library that adds support for opening, manipulating, and saving many different image file formats.
- SciPy is library which is used to solve scientific and mathematical problems. It is built on the NumPy extension and allows the user to manipulate and visualize data with a wide range of high-level commands.
- %matplotlib inline sets the backend of matplotlib to the 'inline' backend. We use this in Jupyter notebook.

4.2 LOADING THE DATASET

```
# Loading the data (cat/non-cat)
train_set_x_orig, train_set_y, test_set_x_orig, test_set_y, classes = load_dataset()
```

Figure 7- Loading our dataset

We divide our datasets into two classes: the training set and testing set. These sets contain sub-classes with x as being the input and y being the output.

```python
# Example of a picture
index = 20
plt.imshow(test_set_x_orig[index])
print ("y = " + str(train_set_y[:, index])+", it's a '"+classes[np.squeeze(train_set_y[:, index])].decode("utf-8") + "'picture.")
```

Figure 8- Example from the dataset

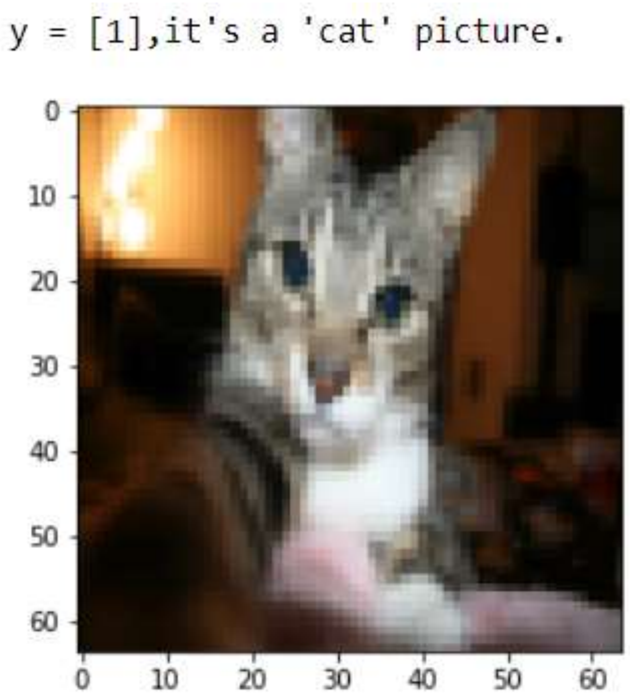


Figure 98- It's a cat picture

This is an example from the training dataset. The output gives [1] if it is a cat picture.

4.3 UNDERSTANDING OUR DATASET

Colored images are 3-dimensional matrices. Pixel intensity across the x and y-axis represent two dimension while RGB makes up the third dimension. We use training examples to train our model.

```
m_train = train_set_x_orig.shape[0]
m_test = test_set_x_orig.shape[0]
num_px = train_set_x_orig.shape[1]

print ("Number of training examples: m_train = " + str(m_train))
print ("Number of testing examples: m_test = " + str(m_test))
print ("Height/Width of each image: num_px = " + str(num_px))
print ("Each image is of size: (" + str(num_px) + ", " + str(num_px) + ", 3)")
print ("train_set_x shape: " + str(train_set_x_orig.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x shape: " + str(test_set_x_orig.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
```

Figure 9- Extracting the shapes of the input images

```
Number of training examples: m_train = 209
Number of testing examples: m_test = 50
Height/Width of each image: num_px = 64
Each image is of size: (64, 64, 3)
train_set_x shape: (209, 64, 64, 3)
train_set_y shape: (1, 209)
test_set_x shape: (50, 64, 64, 3)
test_set_y shape: (1, 50)
```

Figure 10- Dimensions of the input images

We have 209 training and 50 testing examples. The number of pixels in the image is 64*64 with each axis having 64 pixels. The images form our input x. The output y is a row vector denoting [0] or [1] for each of the training example.

CHAPTER 5

General Architecture of the Neural Network

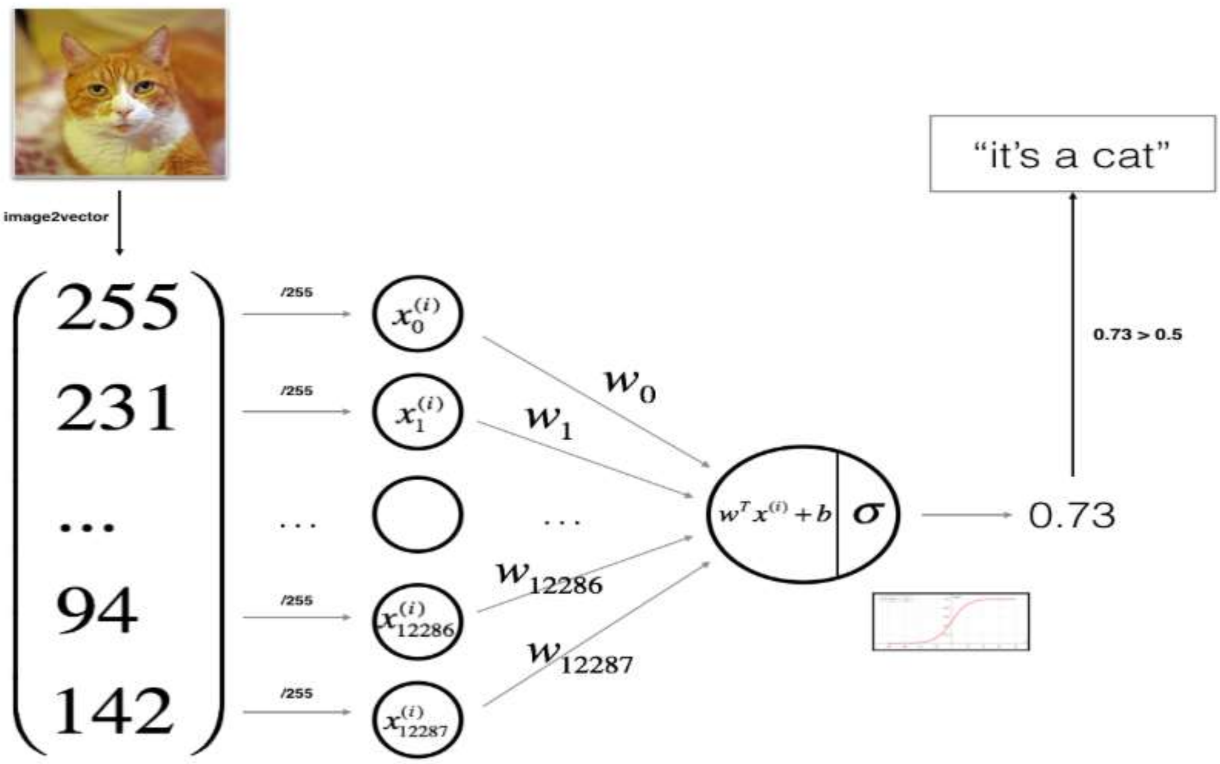


Figure 11- Model of the neural network/Source- Coursera-Introduction to Neural Networks and Deep Learning

Mathematical expression of the algorithm:

For one example $x^{(i)}$, forward propagation:

$$z^{(i)} = w^T.x^{(i)} + b \qquad (1)$$

$$\dot{y}^{(i)} = a^{(i)} = \mathrm{sigmoid}(z^{(i)}) \qquad (2)$$

$$L(a^{(i)}, y^{(i)}) = -y^{(i)} \log(a^{(i)}) - (1-y^{(i)}) \log(1-a^{(i)}) \qquad (3)$$

The cost is then computed by summing over all training examples:

$$J = \frac{1}{m}\sum_{i=1}^{m} L(a^{(i)}, y^{(i)}) \qquad (4)$$

Equations for back-propagation:

$$\frac{\partial J}{\partial w} = \frac{1}{m}X(A-Y)^T \qquad (5)$$

$$\frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(a^{(i)}-y^{(i)}) \qquad (6)$$

As explained before, we first convert the 3D image into a unit dimensional column vector. We then normalize each row value by dividing it by 255. Each input feature is matrix-multiplied by a corresponding weight. We update these weights to get the desired result during back propagation. Since the weight vector has the same dimension as that of the input(X_flatten), we take the transpose of w vector and then perform matrix multiplication. We add a bias b to alter the weight according to our needs. The bias is a scalar parameter and also requires to updated. The resultant is called z. The activation function is the the sigmoid function which takes z as the input and outputs a i.e. the probability. Based on the probability we calculate the loss function. The loss function is based on the maximum likelihood estimation. It tells how much the predicted result is different from the actual result. The cost function is the mean of loss function over the entire training set.

Using the cost function we alter the weights in order to get the desired output. We achieve this by gradient descent which requires the computation of **dJ/dw** and **dJ/db(** denoted dw and db respectively) i.e. the change in L with respect to change in w and b. We go forward and backward till we get an acceptable low cost value.

NOTE :- The superscript (i) in the above equation denotes the i^{th} example. A, Y represent the vectorized version of a, y i.e. the m examples stacked together to get m columns.

5.1 INPUT TO THE NEURAL NETWORK

In order to input our images, we require to squeeze the 3 dimensions to a single dimension. This can be done by taking each pixel value in an orderly fashion and stacking it as a row vector. This way all pixel values are stacked one over the other making it a one-dimensional matrix. Further, each training example has to be computed independently. This can be done by taking an explicit for-loop and incrementing the loop to 50. But this is a very long process. To avoid such time consumption, we vectorize the input to a 2- dimensional input layer with each row representing the corresponding pixel of the input image and each column denoting the different examples. So the new dimensions of the input are now (64*64*3, 50).

A trick when you want to flatten a matrix X of shape (a,b,c,d) to a matrix X_flatten of shape (b∗c∗d, a) is to use:

```
X_flatten = X.reshape(X.shape[0], -1).T
```

```
# Reshape the training and test examples
train_set_x_flatten = train_set_x_orig.reshape(train_set_x_orig.shape[0], -1).T
test_set_x_flatten = test_set_x_orig.reshape(test_set_x_orig.shape[0], -1).T
print ("train_set_x_flatten shape: " + str(train_set_x_flatten.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x_flatten shape: " + str(test_set_x_flatten.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
print ("sanity check after reshaping: " + str(train_set_x_flatten[0:5,0]))
```

Figure 12- Reshaping the dimensions of our input

```
train_set_x_flatten shape: (12288, 209)
train_set_y shape: (1, 209)
test_set_x_flatten shape: (12288, 50)
test_set_y shape: (1, 50)
sanity check after reshaping: [17 31 56 22 33]
```

Figure 13- Dimensions after flattening

Each pixel value in our flattened image is ranging from 0-255. We normalize it by dividing each pixel by 255.

```
train_set_x = train_set_x_flatten/255.
test_set_x = test_set_x_flatten/255.
```

Figure 14- Normalizing the input

CHAPTER 6

Parameter Operations during Forward and Backward Propagations

6.1 SIGMOID FUNCTION

```
def sigmoid(z):

    s = 1/(1+np.exp(-z))

    return s
```

Figure 15- Defining the sigmoid function

6.2 INITIALIZING THE PARAMETERS

```
def initialize_with_zeros(dim):

    w = np.zeros([dim,1])
    b = 0

    assert(w.shape == (dim, 1))
    assert(isinstance(b, float) or isinstance(b, int))

    return w, b
```

Figure 16- Initializing the weights and bias

```
dim = 2
w, b = initialize_with_zeros(dim)
print ("w = " + str(w))
print ("b = " + str(b))
```

Figure 17- Calling the function

```
w = [[ 0.]
 [ 0.]]
b = 0
```

Figure 18-Initial value of the parameters

We initialize the weights and bias to zero. Since w is a matrix of the same dimension as the input, we multiply the transpose of w with X. The assert keyword is used to confirm the dimensions of the parameters. It gives an error if the dimensions are not matched. The bias b is arbitrary and hence we assign it with value 0.

6.3 FORWARD & BACKWARD PROPAGATION

FORWARD PROPAGATION

- The first layer consists of input features. The input features have to be a row vector.
- The second layer consists of two computations. They are pre-activation and activation respectively.
- Weights(w) are multiplied with the input features in the pre-activation. A bias(b) is also added to their product accordingly. It is denoted by z.
- The activation computes the probability of activation of the corresponding neuron. z is the input and the g(z) is the output where g(x) is the activation function.
- We often use the ReLU (Rectified Linear Unit) as the activation function in the hidden layers and Sigmoid function in the output layer.
- The maximum probability of activation in the output layer is usually the answer of our problem.

BACKWARD PROPAGATION

- During training, once the output activation is achieved, loss is calculated using the loss function. The loss is respect to the labelled data.
- We need to find the change in the loss function with respect to change in the weights and biases.
- We are essentially finding the minima of the loss function using a technique known as stochastic gradient descent.
- Stochastic gradient descent is a an optimizer algorithm which involves hyperparameters like the learning rate.

```python
def propagate(w, b, X, Y):

    m = X.shape[1]

    A = sigmoid(np.dot(w.T, X) + b)
    cost = -np.sum((np.multiply(Y, np.log(A)) + (np.multiply(1-Y, np.log(1-A)))))/m

    dw = np.dot(X, (A-Y).T)/m
    db = np.sum(A-Y)/m


    assert(dw.shape == w.shape)
    assert(db.dtype == float)
    cost = np.squeeze(cost)
    assert(cost.shape == ())

    grads = {"dw": dw,
             "db": db}

    return grads, cost
```

Figure 19- Computing cost during forward and backward propagation

```python
w, b, X, Y = np.array([[1.],[2.]]), 2., np.array([[1.,2.,-1.],[3.,4.,-3.2]]), np.array([[1,0,1]])
grads, cost = propagate(w, b, X, Y)
print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
print ("cost = " + str(cost))
```

Figure 20- Performing forward and backward propagation and storing the values in a dictionary

```
dw = [[ 0.99845601]
 [ 2.39507239]]
db = 0.00145557813678
cost = 5.80154531939
```

Figure 21- Change in cost function with respect to the weight and bias

For the forward propagation we calculate the activation and the cost functions. The equations have been defined above. 'm' represents the number of training examples. Since X is a 2 dimensional matrix, the columns denoting the examples hence m = X.shape[1].

For the backward propagation the parameters dw and db are calculated. The equations have been defined above. Both w and dw have the same dimensions. Similarly for b and db. np.squeeze () is a numpy function used to do away with redundant dimensions.

Finally we define a dictionary grads containing dw and db. The function returns cost and grads.

NOTE:- The above function takes in sample arguments for the sake of checking the correctness of the function. We will input the original arguments shortly.

6.4 UPDATING THE PARAMETERS

```
def optimize(w, b, X, Y, num_iterations, learning_rate, print_cost = False):

    costs = []

    for i in range(num_iterations):

        grads, cost = propagate(w, b, X, Y)

        dw = grads["dw"]
        db = grads["db"]

        w -= learning_rate*dw
        b -= learning_rate*db

        if i % 100 == 0:
            costs.append(cost)

        if print_cost and i % 100 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

    params = {"w": w,
              "b": b}

    grads = {"dw": dw,
             "db": db}

    return params, grads, costs
```

Figure 22- Updating the weight and bias

```
params, grads, costs = optimize(w, b, X, Y, num_iterations= 100, learning_rate = 0.009, print_cost = False)

print ("w = " + str(params["w"]))
print ("b = " + str(params["b"]))
print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
```

Figure 23- Extracting the value of parameters after updating

```
w = [[ 0.19033591]
 [ 0.12259159]]
b = 1.92535983008
dw = [[ 0.67752042]
 [ 1.41625495]]
db = 0.219194504541
```

Figure 24- Value of parameters

Here we are updating the parameters using gradient descent. First we retrieve the parameters dw and db from the grads dictionary. Using the hyperparameter learning_rate = 0.009, we compute w and b using the above equations. We store the updated parameters in dictionaries params and grads. The function returns params, grads and costs.

We have additionally defined a costs array. The cost gets updated after every iteration. We append the value of the cost variable in the costs array after every 100 iterations. Using this array we will plot the learning curve.

NOTE:- The above function takes in sample arguments for the sake of checking the correctness of the function. We will input the original arguments shortly.

CHAPTER 7

7.1 BINARY CLASSIFICATION

```
def predict(w, b, X):

    m = X.shape[1]
    Y_prediction = np.zeros((1,m))
    w = w.reshape(X.shape[0], 1)

    A = sigmoid(np.dot(w.T, X) + b)

    for i in range(A.shape[1]):

        if(A[0][i] <= 0.5):
            Y_prediction[0][i] = 0

        else:
            Y_prediction[0][i] = 1
        pass

    assert(Y_prediction.shape == (1, m))

    return Y_prediction
```

Figure 25- Predicting an example

```
w = np.array([[0.1124579],[0.23106775]])
b = -0.3
X = np.array([[1.,-1.1,-3.2],[1.2,2.,0.1]])
print ("predictions = " + str(predict(w, b, X)))
```

Figure 26- Testing an example on the predict function

```
predictions = [[ 1.  1.  0.]]
```

Figure 27- Output for our test example

Here we classify the image as a cat image or a non-cat image based on the probability computed by the activation function. If the probability is greater than 0.5 then it is classified as a cat image and a non-cat image otherwise. The dimensions of Y_prediction is the same as A, i.e. (1, m). It classifies each example as a cat or a non-cat image.

NOTE:- The above function takes in sample arguments for the sake of checking the correctness of the function. We will input the original arguments shortly.

7.2 THE BINARY CLASSIFIER

```python
def model(X_train, Y_train, X_test, Y_test, num_iterations = 2000, learning_rate = 0.5, print_cost = False):

    w, b = np.zeros([X_train.shape[0], 1]), 0

    parameters, grads, costs = optimize(w, b, X_train, Y_train, num_iterations, learning_rate, print_cost = False)

    w = parameters["w"]
    b = parameters["b"]

    Y_prediction_test = predict(w, b, X_test)
    Y_prediction_train = predict(w, b, X_train)

    print("train accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_train - Y_train)) * 100))
    print("test accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_test - Y_test)) * 100))


    d = {"costs": costs,
         "Y_prediction_test": Y_prediction_test,
         "Y_prediction_train" : Y_prediction_train,
         "w" : w,
         "b" : b,
         "learning_rate" : learning_rate,
         "num_iterations": num_iterations}

    return d
```

Figure 28- Linking all the functions and making a final model

```python
d = model(train_set_x, train_set_y, test_set_x, test_set_y, num_iterations = 2000, learning_rate = 0.005, print_cost = True)
```

Figure 29- Setting the hyperparameters and providing the training and testing dataset

```
train accuracy: 99.04306220095694 %
test accuracy: 70.0 %
```

Figure 30- Displaying the accuracy

Here we build the neural network that we intended from the beginning. We pass the original parameters. All the functions defined before are called through cascading. Finally we test the accuracy of our model by comparing Y_prediction with train_set_y and test_set_y. We see the training accuracy to be much greater than the testing accuracy. This is a case of overfitting. But the model performs pretty good considering the network uses no hidden layers.

7.3 SAMPLE TEST

```
index = 49
plt.imshow(test_set_x[:,index].reshape((num_px, num_px, 3)))
print("y ="+str(test_set_y[0,index])+",you predicted that it is a\""+classes[d["Y_prediction_test"][0,index]].decode("utf-8")
```

Figure 31- Displaying a result from testing

```
y = 0, you predicted that it is a "non-cat" picture.
```

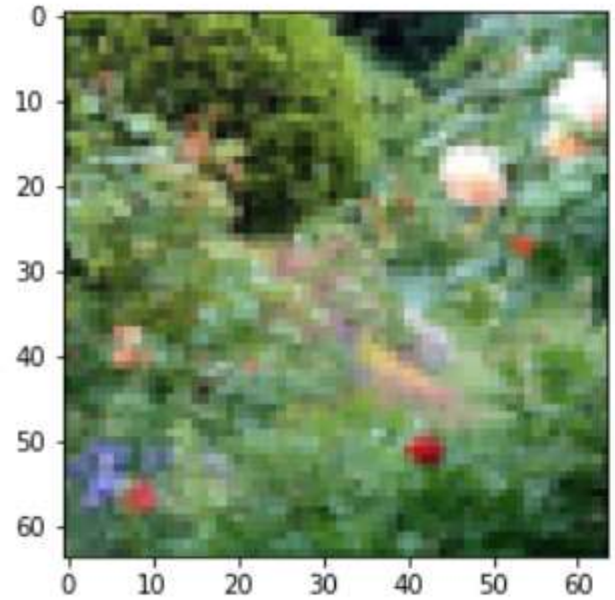


Figure 32- The desired result

This is a test example. We see the neural network predicts the desired result.

7.4 COST

```
# Plot learning curve (with costs)
costs = np.squeeze(d['costs'])
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations (per hundreds)')
plt.title("Learning rate =" + str(d["learning_rate"]))
plt.show()
```

Figure 33- Plot learning curve

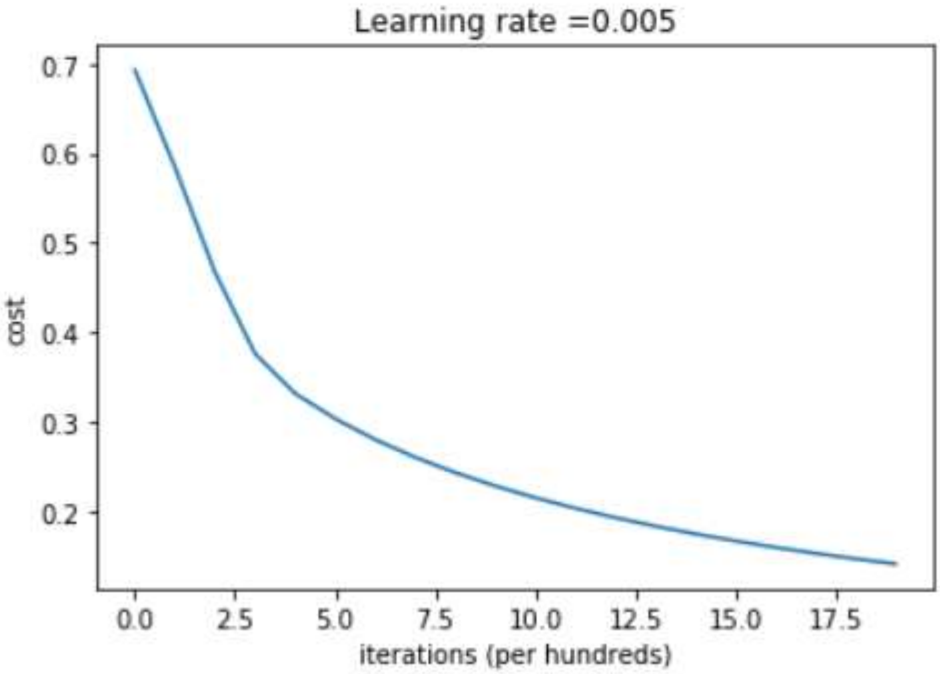


Figure 34- The cost function vs no. of iterations with a learning rate = 0.005

Here we plot the cost as a function of number of iterations. Here we have hard coded the learning rate. We see the cost decreases with increase in the number of iterations.

CHAPTER 8

8.1 Why we use Tensorflow and Keras

TensorFlow is a library package in python which is a symbolic math library and has extensive use in the fields of data science and machine learning. The in-built functions make the code reusable and efficient.

Keras is an open-source neural network library written in python which is used on top of TensorFlow. It focuses on being user-friendly, modular and extensible.

- In the following program we use pooling alongside convolutions. Pooling further reduces the size of the image.
- In MaxPooling it retains the maximum value of the specified neighborhood and discards the other pixel values.
- In AveragePooling it averages the values in the specified neighborhood.

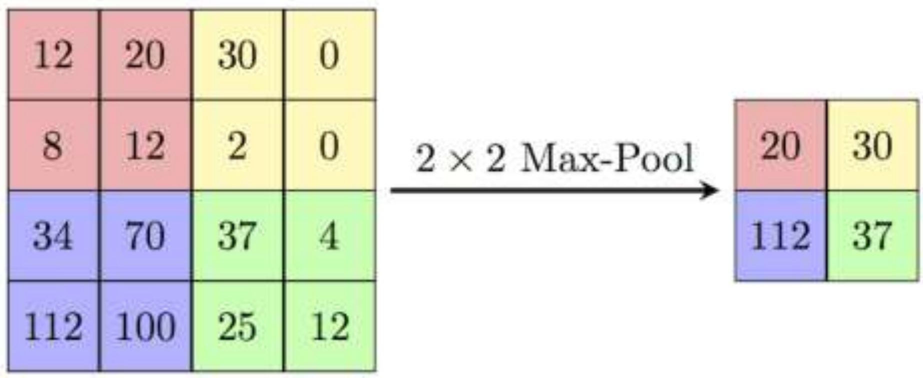


Figure 35- Max Pooling

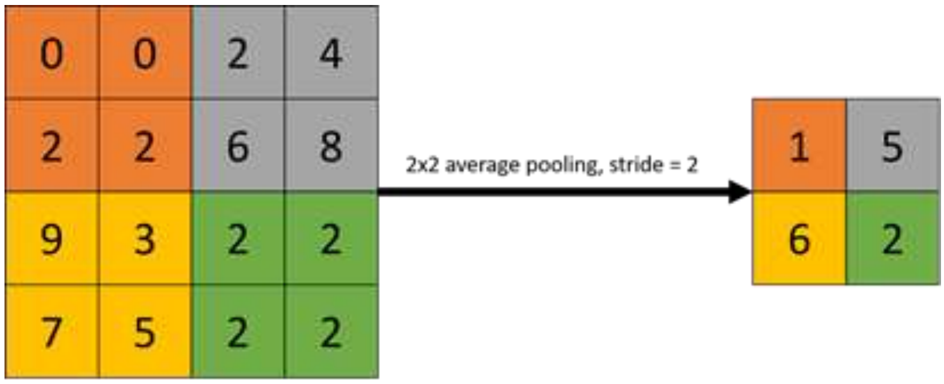


Figure 36- Average Pooling

8.2 ImageDataGenerator

Another feature in TensorFlow is its ability to label the datasets based on the sub-directories. Most of the times, the dataset generated are not labelled and it is a very time-consuming activity to label each and every image. In TensorFlow there is already a in-built function ***ImageDataGenerator*** which label the datasets.

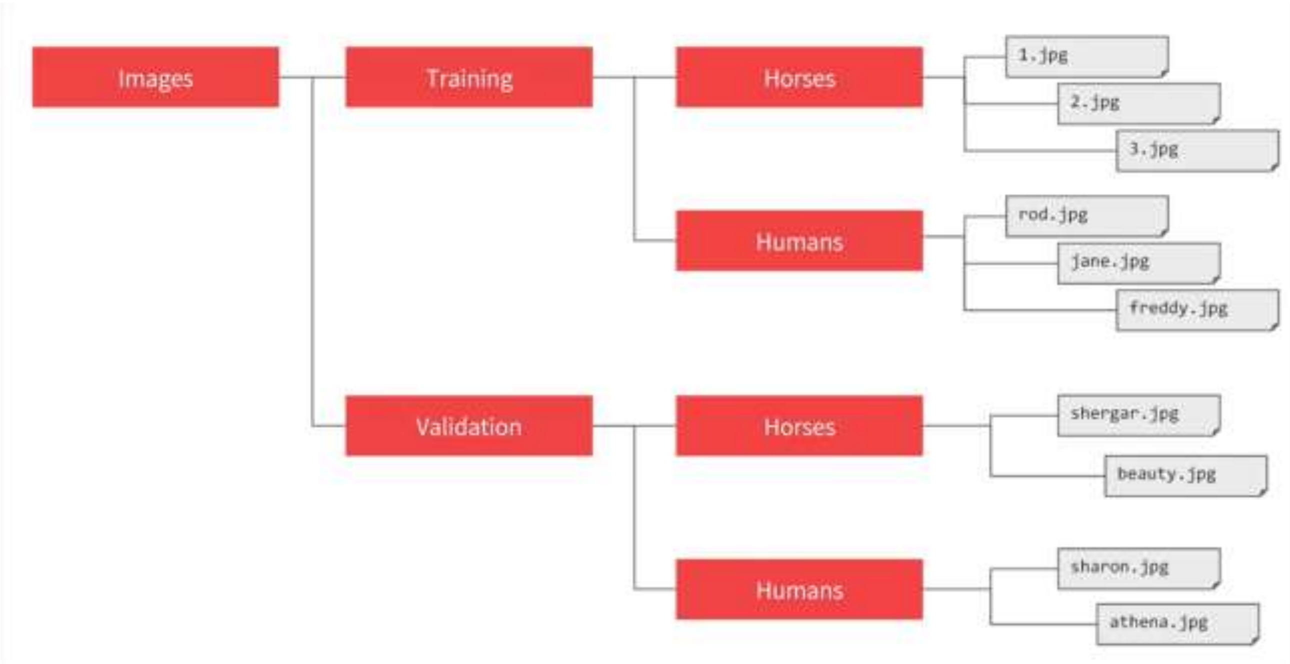


Figure 37- Coursera, Introduction to TensorFlow for Artificial Intelligence, Machine Learning, and Deep Learning

8.3 Cat Detection using TensorFlow

Here I train a similar neural network used for detecting images of cats using TensorFlow and Keras.

```
import tensorflow as tf
import os
import zipfile
from os import path, getcwd, chdir

path = f"{getcwd()}/../tmp2/happy-or-sad.zip"

zip_ref = zipfile.ZipFile(path, 'r')
zip_ref.extractall("/tmp/h-or-s")
zip_ref.close()
```

Figure 38- Importing zip files and extracting them

```
def train_cat_not_model():

    DESIRED_ACCURACY = 0.999

    class myCallback(tf.keras.callbacks.Callback):

        def on_epoch_end(self, epoch, logs = {}):

            if(logs.get('acc') > DESIRED_ACCURACY):
                print("Reached 99.9% accuracy so cancelling training!")
                self.model.stop_training = True

    callbacks = myCallback()
```

Figure 39- This function trains the model till it achieves the desired accuracy

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(16, (3,3), activation = 'relu', input_shape = (150, 150, 3)),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(32, (3,3), activation = 'relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(32, (3,3), activation = 'relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation = 'relu'),
    tf.keras.layers.Dense(1, activation = 'sigmoid')
])
```

Figure 40- Applying convolutions and pooling to reduce computation time. The ‘dense’ keyword defines the number of neurons in the layer.

```python
from tensorflow.keras.optimizers import RMSprop

model.compile(loss = 'binary_crossentropy', optimizer = RMSprop(lr = 0.001), metrics = ['accuracy'])

from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(rescale = 1./255)

train_generator = train_datagen.flow_from_directory(
    '/tmp/h-or-s',
    target_size = (150, 150),
    batch_size = 10,
    class_mode = 'binary')
```

Figure 41- Propagating through the neural network and generating labels for the dataset. The rescale normalizes the pixel values

```python
history = model.fit_generator(
      train_generator, steps_per_epoch = 8, epochs = 15, verbose = 1, callbacks = [callbacks])

return history.history['acc'][-1]
```

Figure 42- Training the model till the desired accuracy

```python
train_cat_not_model()
```

Figure 43- Calling the function

OUTPUT:

```
Found 80 images belonging to 2 classes.
Epoch 1/15
8/8 [==============================] - 5s 638ms/step - loss: 1.0556 - acc: 0.5875
Epoch 2/15
8/8 [==============================] - 1s 71ms/step - loss: 0.7459 - acc: 0.7625
Epoch 3/15
8/8 [==============================] - 1s 64ms/step - loss: 0.2174 - acc: 0.9125
Epoch 4/15
8/8 [==============================] - 1s 73ms/step - loss: 0.1534 - acc: 0.9500
Epoch 5/15
8/8 [==============================] - 1s 63ms/step - loss: 0.1707 - acc: 0.9250
Epoch 6/15
8/8 [==============================] - 1s 63ms/step - loss: 0.0388 - acc: 0.9875
Epoch 7/15
7/8 [=========================>....] - ETA: 0s - loss: 0.0333 - acc: 1.0000Reached 99.9% accuracy so cancelling training!
8/8 [==============================] - 1s 73ms/step - loss: 0.0376 - acc: 1.0000
```

Figure 44- Training till the model achieves 99.9% accuracy, i.e., till the 7th epoch

CHAPTER 9

9.1 Training the Neural Network for Face Mask Detection

1) Importing Libraries and Packages

```
# 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
```

- We import all the necessary dependencies from TensorFlow and Keras. Some of the functions have already been discussed.
- ImageDataGenerator is used in data augmentation and class labelling.
- MobileNetV2 is a neural network architecture optimised for mobile devices.
- AveragePooling, Dropout, Flatten Dense, Input are the usual functions that are required to build a neural network.
- We'll be using Adam optimizer in our deep net.

```
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
```

- Pre-processing the input is required in neural networks especially when using a system with low computation power.
- Img_to_array is used to convert the multi-dimensional input image to a single dimension.
- To_categorical and LabelBinarizer are ways to label an output. It makes the training easier and efficient. These functions have been explained in Appendix-D.
- Sklearn is an additional machine learning library. It features various classification, regression and clustering algorithms including support vector machines etc.

The complete code without explanation is available in Appendix-A.

- Imutils is an important convenience package. Paths is used to store the path of the dataset and other testing examples for the model to use.
- Matplotlib, NumPy, argparse and os have additional utilities as we'll see later.

2) Argparsing Arguments

```
# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-d", "--dataset", required=True,
    help="path to input dataset")
ap.add_argument("-p", "--plot", type=str, default="plot.png",
    help="path to output loss/accuracy plot")
ap.add_argument("-m", "--model", type=str,
    default="mask_detector.model",
    help="path to output face mask detector model")
args = vars(ap.parse_args())
```

- Argparsing has been discussed in detail in Appendix A.
- Dataset argument is required to train the images. Dataset contains all our training and testing examples. This is a necessary argument.
- The plot displays the training and validation loss and accuracy over number of iterations.
- The neural network model is stored in mask_detector.model as default. If necessary we can change the default parameter.
- We store the parse arguments in a dictionary args. This has been discussed in Appendix-A.

3) Hyperparameter Initialization and Loading Dataset

```
# initialize the initial learning rate, number of epochs to train for,
# and batch size
INIT_LR = 1e-4
EPOCHS = 20
BS = 32
```

- We store the values of the hyper-parameters. Note that the learning rate is hard-coded for the time being. As we will see later, we add a decay to the learning rate. BS denotes Batch-size.

```
# 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...")
imagePaths = list(paths.list_images(args["dataset"]))
data = []
labels = []
```

- We create a list containing all the dataset images. We use the function paths.list_images to get the path of our dataset. Args is the dictionary we defined during argparsing.
- We create two lists – one for the data and the other for the labels.

```
# loop over the image paths
for imagePath in imagePaths:
    # 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)
```

- We loop over the imagePaths list and extract the class label from the filename. imagePath.split is a list containing the names of the directories that make up the path. The second last element of the list is the name of the labels' file.
- We perform some operations on the input image. This includes converting the 3D image to a single dimensional array and pre-processing it. Pre-processing has been discussed in Appendix-C.
- The lists – label and data are being appended after each iteration of the loop.
- We convert these lists to NumPy arrays for the ease of operations.

4) Processing the Dataset

```
# perform one-hot encoding on the labels
lb = LabelBinarizer()
labels = lb.fit_transform(labels)
labels = to_categorical(labels)
```

- LabelBinarizer, fit_transform and to_categorical have been discussed in Appendix-D.

```
# 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)
```

- We split our dataset using the train_test_split function. It splits the data and labels according to the proportion assigned to test_size. Stratify ensures that the training and testing samples contain an equal measure of dichotomy.
- Random_state = 42 ensures that the division between test and train samples are the same every time the program is run. Random_state = 0 would make the partition random at every run.

The complete code without explanation is available in Appendix-A.

5) Data Augmentation

```
# construct the training image generator for data augmentation
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")
```

- We create an instance of data augmentation using the ImageDataGenerator class. We will apply this to our dataset later.

6) Building the Neural Network

```
# load the MobileNetV2 network, ensuring the head FC layer sets are
# left off
baseModel = MobileNetV2(weights="imagenet", include_top=False,
    input_tensor=Input(shape=(224, 224, 3)))
```

- We create a base model which will be the input layer for our neural network. The function arguments have their usual meanings. MobileNetV2 architecture has been discussed in detail in Appendix-D.

```
# construct the head of the model that will be placed on top of the
# the base model
headModel = baseModel.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)
```

- Further layers of the neural network are defined.
- These layers are placed on top of the base model, i.e., the output from the base model will be loaded into the head model.
- We apply average pooling with (7,7) blocks. This is similar to max pooling with the difference being we take the average of the (7,7) block instead of taking the max value.
- We define a layer with 128 units and relu activation function.
- We apply dropout with the threshold value being 0.5.
- The output layer has 2 units – with mask or without mask classes. The output activation is the softmax function.

```
# place the head FC model on top of the base model (this will become
# the actual model we will train)
model = Model(inputs=baseModel.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 baseModel.layers:
    layer.trainable = False
```

- We construct the model using Model function. Note that the output of the base model is the input to the head model.
- We freeze the layers of the base model. This ensures that the parameters are not updated during the first iteration of training.

```
# 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"])

```

- We compile the model and define the hyper-parameters. The decay is also defined which is essentially reducing the value of the learning rate.
- The loss is computed using binary cross entropy.

7) Training the Neural Network

```
# train the head of the network
print("[INFO] training head...")
H = model.fit(
    aug.flow(trainX, trainY, batch_size=BS),
    steps_per_epoch=len(trainX) // BS,
    validation_data=(testX, testY),
    validation_steps=len(testX) // BS,
    epochs=EPOCHS)
```

- Now we train the neural network using our training dataset.
- Our validation dataset is same as the testing dataset. It validates the test dataset after each propagation and compares it with test labels.

The complete code without explanation is available in Appendix-A.

8) Validation and Saving the Model

```
# make predictions on the testing set
print("[INFO] evaluating network...")
predIdxs = 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
predIdxs = np.argmax(predIdxs, axis=1)

# show a nicely formatted classification report
print(classification_report(testY.argmax(axis=1), predIdxs,
    target_names=lb.classes_))
```

- We test our trained model and store the values in a variable predIdxs.
- We use argmax function to get the predicted results. Argmax gives out the input corresponding to the max value of the function. In this case, if the maximum probability corresponds to an image with mask, the argmax function will give the output – with_mask.

```
# serialize the model to disk
print("[INFO] saving mask detector model...")
model.save(args["model"], save_format="h5")

```

- We save the model in the h5 format.

9) Plotting the different Parameters

```
# 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")
plt.savefig(args["plot"])
```

- We plot the training accuracy, training loss, validation accuracy and validation loss over iterations. Finally we save the plot with .png extension.

The complete code without explanation is available in Appendix-A.

9.2 Testing of our Model on Live Examples

1) Importing the Libraries

```
# python detect_mask_image.py --image examples/example_01.png

# 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
import numpy as np
import argparse
import cv2
import os

```

- Import the necessary packages and dependencies.
- cv2 is the OpenCV library and is necessary for face detections on live examples.

2) Argparsing Arguments

```
# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required=True,
    help="path to input image")
ap.add_argument("-f", "--face", type=str,
    default="face_detector",
    help="path to face detector model directory")
ap.add_argument("-m", "--model", type=str,
    default="mask_detector.model",
    help="path to trained face mask detector model")
ap.add_argument("-c", "--confidence", type=float, default=0.5,
    help="minimum probability to filter weak detections")
args = vars(ap.parse_args())
```

- We parse arguments. The image argument is necessary to parse. The model will try to detect the face mask on this image. We give the path to the image.
- We create other default arguments. These will be used in our program.
- We have discussed about confidence and confidence threshold in detail. The confidence here is 0.5.
- We create a dictionary args.

3) Loading the Face Detection Model

```
# load our serialized face detector model from disk
print("[INFO] loading face detector model...")
prototxtPath = os.path.sep.join([args["face"], "deploy.prototxt"])
weightsPath = os.path.sep.join([args["face"],
    "res10_300x300_ssd_iter_140000.caffemodel"])
net = cv2.dnn.readNet(prototxtPath, weightsPath)
```

- This is a built in function in OpenCV to detect faces in an image. Args[“face”] returns face_detector since during arparsing we defined its default value the same.

The complete code without explanation is available in Appendix-A.

- The net variable contains the information on how to detect a face in an image.

```
# load the face mask detector model from disk
print("[INFO] loading face mask detector model...")
model = load_model(args["model"])
```

- We load the trained model from the disk. Note that model was created during argparsing.

4) Face Detection in the Input Image

```
# load the input image from disk, clone it, and grab the image spatial
# dimensions
image = cv2.imread(args["image"])
orig = image.copy()
(h, w) = image.shape[:2]

# construct a blob from the image
blob = cv2.dnn.blobFromImage(image, 1.0, (300, 300),
    (104.0, 177.0, 123.0))

# pass the blob through the network and obtain the face detections
print("[INFO] computing face detections...")
net.setInput(blob)
detections = net.forward()
```

- We create an instance of the image, clone it and grab the spatial dimensions of the image.
- We construct a blob of the image. The function blobFromImage has been discussed in detail in Appendix-C.
- We input the blob to our face detection model which is net. It then detects the face and the information is then stored in another variable detections.

5) Building a Confidence Threshold for Face Detection

```
# 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 > args["confidence"]:
        # 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")
```

- Only images where face detection is above a certain confidence threshold are selected.
- After they pass the confidence threshold, we grab the spatial dimensions of the face. We ensure dimensions are integer type by using astype("int").

The complete code without explanation is available in Appendix-A.

```
        # 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))
```

- Here we check the validity of the box dimensions by ensuring they are inside the image spatial dimensions.

6) Pre-processing the Input Image

```
        # extract the face ROI, convert it from BGR to RGB channel
        # ordering, resize it to 224x224, and preprocess it
        face = image[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)
        face = np.expand_dims(face, axis=0)
```

- The image's region of interest is extracted, i.e., the face. We then apply other operations on it.
- We change the order of the colour channels and resize it and then pre-process it.
- Expand_dims is a NumPy function which inserts an additional axis at the specified position. This axis stores the probability of the person wearing a mask or not.

7) Display the Result on the Screen

```
        # pass the face through the model to determine if the face
        # has a mask or not
        (mask, withoutMask) = model.predict(face)[0]

        # 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)

```

- The model predicts whether the person is wearing a face mask or not.
- We compare the probabilities and build a label which will be displayed on the screen.

```
        # 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
        cv2.putText(image, label, (startX, startY - 10),
            cv2.FONT_HERSHEY_SIMPLEX, 0.45, color, 2)
        cv2.rectangle(image, (startX, startY), (endX, endY), color, 2)

# show the output image
cv2.imshow("Output", image)
cv2.waitKey(0)
```

- We display a rectangular box around the face with a label – mask or no mask. There is also a confidence label which displays how much the model is confident of the result out of 100.
- The interface waits for the user to press any key to exit runtime.

The complete code without explanation is available in Appendix-A.

9.3 Detection of Face Mask during a Live Video Stream

1) Importing Libraries

```python
# 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
```

- Import the necessary packages and dependencies.
- For using the webcam we require VideoStream. This allows us to capture the frames.
- We also require the time library.

2) Processing the Video frame

```python
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))
```

- We create a function with arguments:
  i) Frame: The particular frame captured by our webcam.
  ii) faceNet: Face detection.
  iii) maskNet: Mask detection.
- We grab the spatial dimensions of the frame.
- We create a blob as discussed before.

3) Face Detections

```python
    # 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 = []
    preds = []
```

- We obtain the face detection by passing it through the network.
- We initialize list of faces, their corresponding locations, and the list of predictions from face mask network.

The complete code without explanation is available in Appendix-A.

4) Building Confidence Threshold

```
# 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 > args["confidence"]:
        # 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")
```

- As discussed before in 'Building a Confidence Threshold for Face Detection' (page 39) we let go of images with confidence lower than the threshold.
- We create a box with the specified dimensions.

5) Pre-processing the Input Frames

```
        # 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)
        face = np.expand_dims(face, axis=0)

```

- The same operations as in 'Pre-processing the Input Image' (page 40).

```
        # add the face and bounding boxes to their respective
        # lists
        faces.append(face)
        locs.append((startX, startY, endX, endY))

```

- Since there are multiple frames in a video, we append each frame's operation in the lists we initialized before.

6) Face mask Detection and Prediction

```python
    # 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
        preds = maskNet.predict(faces)

    # return a 2-tuple of the face locations and their corresponding
    # locations
    return (locs, preds)
```

- We make a prediction of face-mask detection only if there were one or more face detections. We use the maskNet network.
- We return the location of the faces and the prediction based on it.

7) Argparsing Arguments

```python
# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-f", "--face", type=str,
    default="face_detector",
    help="path to face detector model directory")
ap.add_argument("-m", "--model", type=str,
    default="mask_detector.model",
    help="path to trained face mask detector model")
ap.add_argument("-c", "--confidence", type=float, default=0.5,
    help="minimum probability to filter weak detections")
args = vars(ap.parse_args())
```

- Argument parsing like discussed earlier. The arguments parsed are the same as before.

8) Load the Face and Face Mask Detection Models

```python
# load our serialized face detector model from disk
print("[INFO] loading face detector model...")
prototxtPath = os.path.sep.join([args["face"], "deploy.prototxt"])
weightsPath = os.path.sep.join([args["face"],
    "res10_300x300_ssd_iter_140000.caffemodel"])
faceNet = cv2.dnn.readNet(prototxtPath, weightsPath)

# load the face mask detector model from disk
print("[INFO] loading face mask detector model...")
maskNet = load_model(args["model"])
```

- We load the face detector and face-mask detector models.

The complete code without explanation is available in Appendix-A.

9) Start the Video Stream through WebCam

```
# initialize the video stream and allow the camera sensor to warm up
print("[INFO] starting video stream...")
vs = VideoStream(src=0).start()
time.sleep(2.0)
```

- We start our video stream through our webcam with delay of 2 seconds.

```
# loop over the frames from the video stream
while True:
    # grab the frame from the threaded video stream and resize it
    # to have a maximum width of 400 pixels
    frame = vs.read()
    frame = imutils.resize(frame, width=400)

    # detect faces in the frame and determine if they are wearing a
    # face mask or not
    (locs, preds) = detect_and_predict_mask(frame, faceNet, maskNet)
```

- We grab the frame from the threaded video stream and resize it to have a maximum width of 400 pixels.
- We call the function detect_and_predict_mask that we defined earlier and store the return values in locs and preds.

10) Prediction for Face Mask

```
    # loop over the detected face locations and their corresponding
    # locations
    for (box, pred) in zip(locs, preds):
        # 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)

```

- We do the same operations as we did face_mask_detection with the only difference here being that we are operating on a single frame of a video rather than an image.
- We define a label as we did before.

The complete code without explanation is available in Appendix-A.

```
            # display the label and bounding box rectangle on the output
            # frame
            cv2.putText(frame, label, (startX, startY - 10),
                cv2.FONT_HERSHEY_SIMPLEX, 0.45, color, 2)
            cv2.rectangle(frame, (startX, startY), (endX, endY), color, 2)

        # show the output frame
        cv2.imshow("Frame", frame)
        key = cv2.waitKey(1) & 0xFF

        # if the `q` key was pressed, break from the loop
        if key == ord("q"):
            break

    # do a bit of cleanup
    cv2.destroyAllWindows()
    vs.stop()
```

- Like before the label and the prediction accuracy is displayed alongside the rectangle.
- The output frame is shown in real time. As soon as the ‘q’ key Is pressed the video stream stops and we exit runtime.

The complete code without explanation is available in Appendix-A.

CHAPTER 10

Loopholes in the Model

- Our model is not able to detect faces which are largely covered with masks. The mask obscures with the face-detection process and hence it is not able to pass the confidence threshold.
- The model is only able to detect light-coloured masks and fails to detect other dark colours. This is due to the training dataset which only contains light-coloured masks.
- The model is susceptible to over-fitting. This happens when the model is generalized to the training dataset and fails to classify the testing dataset correctly. This happens when the dataset is not large.

How can the Model be Improved

- Insert more layers into the neural network. More layers will help to recognize parts of the face better and hence the model would be able to have high confidence value on the face detections.
- Improve the dataset by increasing the diversity of the training images. Artificial datasets could hamper the performance of the neural network.
- Increase the number of training data and apply fine-tuning. DropOuts, Regularization are some of the techniques through which we could fit our data better in the model.

Conclusion

Using the above techniques and functions we can train a deep neural net that can detect whether a person is wearing a face mask or not.

The complete code without explanation is available in Appendix-A.

APPENDIX-A

train_mask_detector.py

```python
# 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

# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-d", "--dataset", required=True,
    help="path to input dataset")
ap.add_argument("-p", "--plot", type=str, default="plot.png",
    help="path to output loss/accuracy plot")
ap.add_argument("-m", "--model", type=str,
    default="mask_detector.model",
    help="path to output face mask detector model")
args = vars(ap.parse_args())
```

The complete code without explanation is available in Appendix-A.

```python
# 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...")
imagePaths = list(paths.list_images(args["dataset"]))
data = []
labels = []

# loop over the image paths
for imagePath in imagePaths:
    # 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
lb = LabelBinarizer()
labels = lb.fit_transform(labels)
labels = to_categorical(labels)
```

The complete code without explanation is available in Appendix-A.

```python
# 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
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
baseModel = 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 = baseModel.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=baseModel.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 baseModel.layers:
    layer.trainable = False
```

The complete code without explanation is available in Appendix-A.

```python
# 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_size=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...")
predIdxs = 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
predIdxs = np.argmax(predIdxs, axis=1)

# show a nicely formatted classification report
print(classification_report(testY.argmax(axis=1), predIdxs,
    target_names=lb.classes_))

# serialize the model to disk
print("[INFO] saving mask detector model...")
model.save(args["model"], save_format="h5")

# 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")
plt.savefig(args["plot"])
```

The complete code without explanation is available in Appendix-A.

detect_mask_image.py

```python
# 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
import numpy as np
import argparse
import cv2
import os

# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required=True,
    help="path to input image")
ap.add_argument("-f", "--face", type=str,
    default="face_detector",
    help="path to face detector model directory")
ap.add_argument("-m", "--model", type=str,
    default="mask_detector.model",
    help="path to trained face mask detector model")
ap.add_argument("-c", "--confidence", type=float, default=0.5,
    help="minimum probability to filter weak detections")
args = vars(ap.parse_args())

# load our serialized face detector model from disk
print("[INFO] loading face detector model...")
prototxtPath = os.path.sep.join([args["face"], "deploy.prototxt"])
weightsPath = os.path.sep.join([args["face"],
    "res10_300x300_ssd_iter_140000.caffemodel"])
net = cv2.dnn.readNet(prototxtPath, weightsPath)

# load the face mask detector model from disk
print("[INFO] loading face mask detector model...")
model = load_model(args["model"])

# load the input image from disk, clone it, and grab the image spatial
# dimensions
image = cv2.imread(args["image"])
orig = image.copy()
(h, w) = image.shape[:2]

```

The complete code without explanation is available in Appendix-A.

```python
# construct a blob from the image
blob = cv2.dnn.blobFromImage(image, 1.0, (300, 300),
    (104.0, 177.0, 123.0))

# pass the blob through the network and obtain the face detections
print("[INFO] computing face detections...")
net.setInput(blob)
detections = net.forward()

# 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 > args["confidence"]:
        # 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 = image[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)
        face = np.expand_dims(face, axis=0)
```

```
        # pass the face through the model to determine if the face
        # has a mask or not
        (mask, withoutMask) = model.predict(face)[0]

        # 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
        cv2.putText(image, label, (startX, startY - 10),
            cv2.FONT_HERSHEY_SIMPLEX, 0.45, color, 2)
        cv2.rectangle(image, (startX, startY), (endX, endY), color, 2)

# show the output image
cv2.imshow("Output", image)
cv2.waitKey(0)
```

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 = []
    preds = []
```

```python
    # 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 > args["confidence"]:
            # 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)
            face = np.expand_dims(face, axis=0)

            # 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
        preds = maskNet.predict(faces)

    # return a 2-tuple of the face locations and their corresponding
    # locations
    return (locs, preds)

# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-f", "--face", type=str,
    default="face_detector",
    help="path to face detector model directory")
ap.add_argument("-m", "--model", type=str,
    default="mask_detector.model",
    help="path to trained face mask detector model")
ap.add_argument("-c", "--confidence", type=float, default=0.5,
    help="minimum probability to filter weak detections")
args = vars(ap.parse_args())

# load our serialized face detector model from disk
print("[INFO] loading face detector model...")
prototxtPath = os.path.sep.join([args["face"], "deploy.prototxt"])
weightsPath = os.path.sep.join([args["face"],
    "res10_300x300_ssd_iter_140000.caffemodel"])
faceNet = cv2.dnn.readNet(prototxtPath, weightsPath)

# load the face mask detector model from disk
print("[INFO] loading face mask detector model...")
maskNet = load_model(args["model"])

# initialize the video stream and allow the camera sensor to warm up
print("[INFO] starting video stream...")
vs = VideoStream(src=0).start()
time.sleep(2.0)
```

```python
# loop over the frames from the video stream
while True:
    # grab the frame from the threaded video stream and resize it
    # to have a maximum width of 400 pixels
    frame = vs.read()
    frame = imutils.resize(frame, width=400)

    # detect faces in the frame and determine if they are wearing a
    # face mask or not
    (locs, preds) = detect_and_predict_mask(frame, faceNet, maskNet)

    # loop over the detected face locations and their corresponding
    # locations
    for (box, pred) in zip(locs, preds):
        # 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
        cv2.putText(frame, label, (startX, startY - 10),
            cv2.FONT_HERSHEY_SIMPLEX, 0.45, color, 2)
        cv2.rectangle(frame, (startX, startY), (endX, endY), color, 2)

    # show the output frame
    cv2.imshow("Frame", frame)
    key = cv2.waitKey(1) & 0xFF

    # if the `q` key was pressed, break from the loop
    if key == ord("q"):
        break

# do a bit of cleanup
cv2.destroyAllWindows()
vs.stop()
```

APPENDIX-B

Argparse and Command Line Arguments in Python

Command line arguments are flags given to a program/script at runtime. They contain additional information for our program so that it can execute. This makes the program more user interactive and hence it is used very often. This allows us to give our program different input on the fly without changing the code.

Command line arguments can be inculcated using argparsing. Argparsing is used a lot in computer vision and deep learning. We'll be using argparse in the following and hence it is imperative that we understand about it.

```

#demo to illustarte function of argparse
import argparse

ap = argparse.ArgumentParser()
ap.add_argument('-n', '--name', type = str, metavar = '', required = True, help = 'Enter your name')
args = vars(ap.parse_args())

print("Hi there {}. Nice to meet you!".format(args["name"]))

```

Figure 45- Example illustrating Argument Parsing

- We first import the library argparse.
- We add argument using the in-built function add_argument. '-n', '—name' is the short and long notation for the argument.
- We define the parameters. The type here is 'string' which means the argument to be parsed is of the type string.
- 'required = True' means the argument is necessary. If the user runs the program without the argument it'll show an error.
- The help option can be used by the user to see the structure of argument parsing. Below we use the '-h' command to call help.

- We create a dictionary ‘args’ which stores all the arguments parsed by the user. It is helpful when there are multiple arguments.

```
C:\WINDOWS\system32\cmd.exe

C:\Users\Hitesh Aryan Acharya\Desktop>python argparse_demo.py -h
usage: argparse_demo.py [-h] -n

optional arguments:
  -h, --help    show this help message and exit
  -n , --name   Enter your name

C:\Users\Hitesh Aryan Acharya\Desktop>python argparse_demo.py --name Aryan
Hi there Aryan. Nice to meet you!
```

Figure 46- Parsing my name as an argument

APPENDIX-C

Pre-processing the image Input and MobileNetV2 architecture

Whenever working with neural networks it is imperative that we pre-process the input. Data pre-processing aims to facilitate the training/testing process by appropriately transforming and scaling the entire dataset. Pre-processing is necessary before training the machine learning models. Pre-processing removes outliers and scales the features to an equivalent range. In our deep learning model as well, we will be pre-processing our dataset. Pre-processing increases efficiency and reduces time consumption.

blobFromImage

In the context of deep learning and image classification, these pre-processing tasks normally involve:

1. Mean subtraction
2. Scaling
3. Optionally channel swapping

All these processes can be implemented using the function cv2.dnn.blobFromImage. This is an in-built function in the OpenCV library and is used extensively during object detection. We will go through each technique.

Mean Subtraction and Scaling: Mean subtraction is used to help combat illumination changes in the input images in our dataset. We can therefore view mean subtraction as a technique used to aid our Convolutional Neural Networks.

We basically compute the mean across all the colour channels in our input image. The mean is stored in a python tuple (three entries, one each for red, blue and green). The standard deviation or the scaling factor is also calculated across the entire image. After this we perform the following operation:

1. $R = (R-\mu)/\sigma$
2. $B = (B-\mu)/\sigma$
3. $G = (G-\mu)/\sigma$

We have hence normalized the image. We do all this using a simple function,

blob = cv2.dnn.blobFromImage(image, scalefactor=1.0, size, mean, swapRB=True)

--**image**: This is the image we want to preprocess.

--**scalefactor**: σ in the above equations is the scalefactor.

--**size**: Here we supply the spatial size that the Convolutional Neural Network expects.

--**mean**: This is the tuple containing the means of the three channels.

--**swapRB**: OpenCV assumes images are in BGR channel order; however, the `mean` value assumes we are using RGB order. To resolve this discrepancy we can swap the R and B channels in image by setting this value to `True`. By default OpenCV performs this channel swapping for us.

The cv2.dnn.blobFromImage function returns a blob which is our input image after mean subtraction, normalizing, and channel swapping.

MobileNetV2 architecture

In 2017 Google introduced MobileNetV1, a family of general purpose computer vision neural networks designed with mobile devices in mind to support classification, detection and more. The ability to run deep networks on personal mobile devices improves user experience, offering anytime, anywhere access, with additional benefits for security, privacy, and energy consumption. MobileNetV2 is a significant improvement over MobileNetV1 and pushes the state of the art for mobile visual recognition including classification, object detection and semantic segmentation.

So in a nutshell MobilenetV2 is a highly efficient architecture that can be applied to embedded devices with limited computational efficiency (e.g. Raspberry Pi, NVIDIA Jetson Nano etc.).

Deploying our face mask detector to embedded devices could reduce the cost of manufacturing such face mask detection systems, hence why we choose to use this architecture.

APPENDIX-D

LabelBinarizer, fit_transform and to_categorical

1. These concepts are best described using examples. LabelBinarizer is a technique to classify the labelled data as seen from the example below:

```
from numpy import array
from sklearn.preprocessing import LabelBinarizer
# define example
data = ['cold', 'cold', 'warm', 'cold', 'hot', 'hot', 'warm', 'cold',
'warm', 'hot']
values = array(data)
print "Data: ", values
#Binary encode
lb = LabelBinarizer()
print "Label Binarizer:", lb.fit_transform(values)
```

Figure 47- Example illustrating LabelBinarizer

```
Data:  ['cold' 'cold' 'warm' 'cold' 'hot' 'hot' 'warm' 'cold' 'warm' 'hot']
Label Binarizer: [[1 0 0]
 [1 0 0]
 [0 0 1]
 [1 0 0]
 [0 1 0]
 [0 1 0]
 [0 0 1]
 [1 0 0]
 [0 0 1]
 [0 1 0]]
```

Figure 48- The format of the labelled output has changed

2. to_categorical function converts a class vector(integers) to binary class matrix.

```
>>> a = tf.keras.utils.to_categorical([0, 1, 2, 3], num_classes=4)
>>> a = tf.constant(a, shape=[4, 4])
>>> print(a)
tf.Tensor(
  [[1. 0. 0. 0.]
   [0. 1. 0. 0.]
   [0. 0. 1. 0.]
   [0. 0. 0. 1.]], shape=(4, 4), dtype=float32)
```

Figure 49- Using the to_categorical function

3. To center the data (make it have zero mean and unit standard error), we subtract the mean and then divide the result by the standard deviation:

$$x' = \frac{(x - \mu)}{\sigma}$$

We do that on the training set of data. But then we have to apply the same transformation to our testing set (e.g. in cross-validation), or to newly obtained examples before forecast. But we have to use the exact same two parameters μ and σ (values) that we used for centering the training set.

Hence, sklearn's (library) transform's fit() just calculates the parameters and saves them as an internal object's state. Afterwards, we can call its transform() method to apply the transformation to any particular set of examples.

fit_transform() joins these two steps and is used for the initial fitting of parameters on the training set x, while also returning the transformed x'. Internally, the transformer object just calls first fit() and then transform() on the same data.

APPENDIX-E

How to Run the Face-Mask Detector Model on Windows Terminal

Install Python

- On the terminal pass the command python --version. Any version above python 3.5 is good enough to run the program.
- Next pass the command pip --version. The version of pip alongside the python version will be displayed if pip was installed as a package. Pip 20.0.2 is the latest version of pip (as of 13 Jun. 20).
- If you have an older version of python installed, it is recommended that you uninstall the python package and proceed to download the latest version from the net.
- To uninstall, go to Control Panel→Programs→Programs and Features→Uninstall a program. Choose the python package you wantto uninstall.
- Go to https://www.python.org/downloads/ and click on Download Python 3.8.3 (as of 13 June 2020).
- Follow the instructions and click on install.
- The latest version of python alongside the latest pip will be installed on your desktop.
- Now go to terminal and repeat the first step. You'll see the latest version of python and pip installed.

```
Microsoft Windows [Version 10.0.18362.900]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Hitesh Aryan Acharya>python --version
Python 3.7.3

C:\Users\Hitesh Aryan Acharya>pip --version
pip 20.0.2 from C:\Users\Hitesh Aryan Acharya\Anaconda3\lib\site-packages\pip (python 3.7)
```

Figure 50- Installing the latest python version

Importing the necessary Libraries and Packages

- The code involves TensorFlow and keras and other important math packages. We first need to use pip to install these packages.
- To install TensorFlow, pass the command:

  pip install --upgrade tensorflow

- Now we need to install some python dependencies. Pass the following commands:

  pip install numpy scipy

  pip install scikit-learn

  pip install pillow

  pip install h5py

  pip install imutils

- Now to install keras, pass the following command:

  pip install keras

Extracting the Files

- Extract all the files from the zip folder. The extracted folder will be named as face_mask_detector.
- Copy the extracted folder to desktop.
- Go to terminal and pass the command cd desktop/face_mask_detector.

Name	Date modified	Type	Size
dataset	12-06-2020 17:06	File folder	
examples	23-05-2020 19:05	File folder	
face_detector	23-05-2020 19:05	File folder	
detect_mask_image	23-05-2020 19:05	JetBrains PyCharm	4 KB
detect_mask_video	23-05-2020 19:05	JetBrains PyCharm	5 KB
model	12-06-2020 18:06	File	11,215 KB
plot	12-06-2020 18:07	PNG File	24 KB
train_mask_detector	06-06-2020 00:06	JetBrains PyCharm	6 KB

Figure 51- face_mask_detector directory

- The directory contains the above files.
- The dataset folder contains the training and testing examples (images) for the neural network.

- Examples folder contains the live examples we'll be using to test the neural network.
- The face_detector folder contains the face classifier which will be used to detect faces in images.
- detect_mask_image, detect_mask_video and train_mask_detector are python files which we will be running on our terminal.
- model is a text file describing the model of the deep learning structure.
- plot is a image file depicting the training and validation loss and accuracy of the model (during train_mask_detector).

Running the Face-Mask Detector

- On the terminal pass the command:

  python train_mask_detector.py --dataset dataset

The above command runs the python code in train_mask_detector using the dataset folder.

- After the training is completed, we can test our network using the examples in the examples folder. To do so pass the command:

  python detect_mask_image.py –image
  examples/example_01.png

This will display a picture with a man wearing a face-mask. As we can see our model is able to identify the face mask. The classifier also labels the image correctly.

- Similarly we can check for other examples. Pass the command:

  python detect_mask_image.py –image
  examples/example_02.png python
  detect_mask_image.py –image
  examples/example_03.png

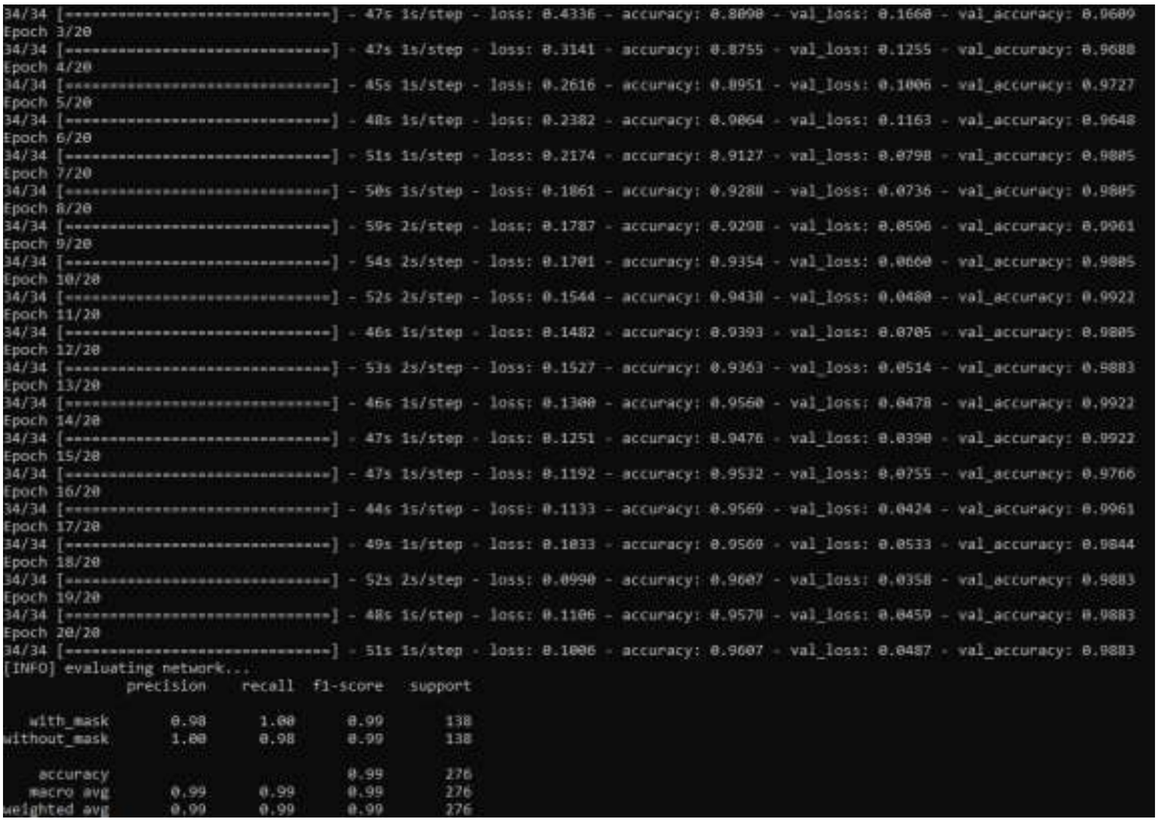
- We can also use the existing model for face-mask detection during a live video stream. To do this pass the command:

  python detect_mask_video.py

Your webcam will now start up. With good accuracy the model will predict whether you're wearing a face mask or not. Good lighting is required for optimal performance.

APPENDIX-F

Some snips during runtime



```
34/34 [==============================] - 47s 1s/step - loss: 0.4336 - accuracy: 0.8090 - val_loss: 0.1660 - val_accuracy: 0.9609
Epoch 3/20
34/34 [==============================] - 47s 1s/step - loss: 0.3141 - accuracy: 0.8755 - val_loss: 0.1255 - val_accuracy: 0.9688
Epoch 4/20
34/34 [==============================] - 45s 1s/step - loss: 0.2616 - accuracy: 0.8951 - val_loss: 0.1006 - val_accuracy: 0.9727
Epoch 5/20
34/34 [==============================] - 48s 1s/step - loss: 0.2382 - accuracy: 0.9064 - val_loss: 0.1163 - val_accuracy: 0.9648
Epoch 6/20
34/34 [==============================] - 51s 1s/step - loss: 0.2174 - accuracy: 0.9127 - val_loss: 0.0798 - val_accuracy: 0.9805
Epoch 7/20
34/34 [==============================] - 50s 1s/step - loss: 0.1861 - accuracy: 0.9288 - val_loss: 0.0736 - val_accuracy: 0.9805
Epoch 8/20
34/34 [==============================] - 59s 2s/step - loss: 0.1787 - accuracy: 0.9298 - val_loss: 0.0596 - val_accuracy: 0.9961
Epoch 9/20
34/34 [==============================] - 54s 2s/step - loss: 0.1701 - accuracy: 0.9354 - val_loss: 0.0660 - val_accuracy: 0.9805
Epoch 10/20
34/34 [==============================] - 52s 2s/step - loss: 0.1544 - accuracy: 0.9438 - val_loss: 0.0480 - val_accuracy: 0.9922
Epoch 11/20
34/34 [==============================] - 46s 1s/step - loss: 0.1482 - accuracy: 0.9393 - val_loss: 0.0705 - val_accuracy: 0.9805
Epoch 12/20
34/34 [==============================] - 53s 2s/step - loss: 0.1527 - accuracy: 0.9363 - val_loss: 0.0514 - val_accuracy: 0.9883
Epoch 13/20
34/34 [==============================] - 46s 1s/step - loss: 0.1300 - accuracy: 0.9560 - val_loss: 0.0478 - val_accuracy: 0.9922
Epoch 14/20
34/34 [==============================] - 47s 1s/step - loss: 0.1251 - accuracy: 0.9476 - val_loss: 0.0398 - val_accuracy: 0.9922
Epoch 15/20
34/34 [==============================] - 47s 1s/step - loss: 0.1192 - accuracy: 0.9532 - val_loss: 0.0755 - val_accuracy: 0.9766
Epoch 16/20
34/34 [==============================] - 44s 1s/step - loss: 0.1133 - accuracy: 0.9569 - val_loss: 0.0424 - val_accuracy: 0.9961
Epoch 17/20
34/34 [==============================] - 49s 1s/step - loss: 0.1033 - accuracy: 0.9569 - val_loss: 0.0533 - val_accuracy: 0.9844
Epoch 18/20
34/34 [==============================] - 52s 2s/step - loss: 0.0990 - accuracy: 0.9607 - val_loss: 0.0358 - val_accuracy: 0.9883
Epoch 19/20
34/34 [==============================] - 48s 1s/step - loss: 0.1106 - accuracy: 0.9579 - val_loss: 0.0459 - val_accuracy: 0.9883
Epoch 20/20
34/34 [==============================] - 51s 1s/step - loss: 0.1006 - accuracy: 0.9607 - val_loss: 0.0487 - val_accuracy: 0.9883
[INFO] evaluating network...
              precision    recall  f1-score   support

   with_mask       0.98      1.00      0.99       138
without_mask       1.00      0.98      0.99       138

    accuracy                           0.99       276
   macro avg       0.99      0.99      0.99       276
weighted avg       0.99      0.99      0.99       276
```

Figure 52- Training our neural network over 20 EPOCHS

Figure 53- Detecting face-mask in an image

Figure 54- Here the model is not able to detect the face mask but correctly detects the non-masked faces

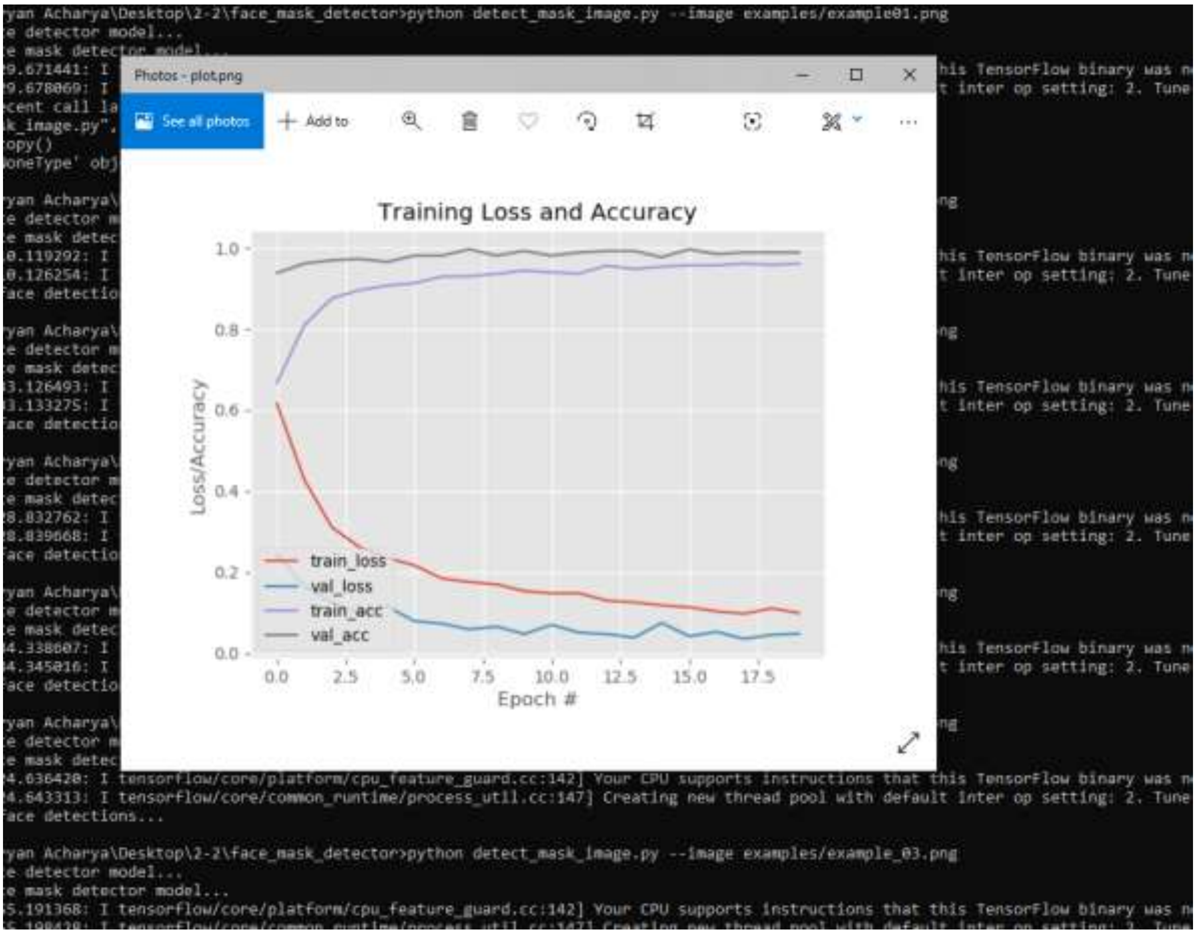


Figure 55- The plot of training and validation parameters

Figure 56- The network correctly detects my green mask during video stream

Figure 57- But it doesn't detect my dark blue mask

References

- COVID-19: Face Mask Detector with OpenCV, Keras/TensorFlow, and Deep Learning (https://www.pyimagesearch.com/2020/05/04/covid-19-face-mask-detector-with-opencv-keras-tensorflow-and-deep-learning/)
- Deep learning: How OpenCV's blobFromImage works (https://www.pyimagesearch.com/2017/11/06/deep-learning-opencvs-blobfromimage-works/)
- TensorFlow Documentation (https://www.tensorflow.org/api_docs/python/tf/keras/Model)
- Keras Documentation (https://keras.io/documentation/)
- Coursera- Neural Networks and Deep Learning (https://www.coursera.org/learn/neural-networks-deep-learning/home/welcome)
- Coursera- Introduction to TensorFlow for Artificial Intelligence, Machine Learning, and Deep Learning (https://www.coursera.org/learn/introduction-tensorflow/home/welcome)
- StackOverflow- Scikit-learn's LabelBinarizer (https://stackoverflow.com/questions/50473381/scikit-learns-labelbinarizer-vs-onehotencoder)
- StackExchange Data Science- fit_transform (https://stackoverflow.com/questions/50473381/scikit-learns-labelbinarizer-vs-onehotencoder)
- Confidence Threshold (https://blog.zenggyu.com/en/post/2018-12-16/an-introduction-to-evaluation-metrics-for-object-detection)
- MobileNetV2 architecture (https://analyticsindiamag.com/why-googles-mobilenetv2-is-a-revolutionary-next-gen-on-device-computer-vision-network/)
- Data Augmentation (https://www.pyimagesearch.com/2019/07/08/keras-imagedatagenerator-and-data-augmentation/)

Glossary:

- Activation: The probability of a neuron being in the on-state.
- Classifier: Categorizing an image into a class.
- Convolution: matrix multiplication of a kernel with the pixel value to get a desired effect.
- Dataset: The input features on which the neural network has to be trained.
- Gradient: The direction along maximum change.
- Hyperparameters: The hard-coded parameters in a neural network. These do not change with time.
- Overfitting: The phenomenon where the model is generalized to the training dataset and does not perform satisfactorily in testing.
- Pooling: A method where the maximum value in a neighbourhood is chosen to reduce the size of the image.
- Regression: A measure of the relation between the mean value of one variable and corresponding values of other variables.
- Vectorization: A method where numbers are stacked in the form of a matrix for the ease of computation.