**FaceMask Detection code explanation.**

```

import os

import cv2

import random

import pickle

from tqdm import tqdm

import numpy as np

import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow import keras

from keras import layers

from keras.models import load\_model

from keras.preprocessing.image import ImageDataGenerator

from keras.callbacks import ModelCheckpoint

from keras.callbacks import EarlyStopping

%matplotlib inline

```

**This is the initial setup of libraries that are needed for the code.**

```

def create\_dir(path):

if not os.path.exists(path):

os.makedirs(path)

def remove\_dir(path):

if os.path.exists(path):

os.removedirs(path)

```

**These are two functions that are defined for creating and removing directories respectively.**

```

dataset\_dir = "D:/Datasets/Face\_Mask"

train\_dir = os.path.join(dataset\_dir, "train")

test\_dir = os.path.join(dataset\_dir, "test")

create\_dir(train\_dir)

create\_dir(test\_dir)

```

**These lines define the directories where the training and testing data will be stored. `dataset\_dir` is the directory where the dataset is stored. `train\_dir` is the directory where the training data will be stored. `test\_dir` is the directory where the testing data will be stored. `create\_dir` function is called to create the directories if they don't exist.**

```

for i in os.listdir(dataset\_dir):

if i.startswith('new'):

files\_dir = os.path.join(dataset\_dir, i)

files = os.listdir(files\_dir)

random.shuffle(files)

create\_dir(os.path.join(train\_dir, i))

create\_dir(os.path.join(test\_dir, i))

for j in files[:int(len(files) \* 0.8)]:

src\_path = os.path.join(files\_dir, j)

dest\_path = os.path.join(train\_dir, i, j)

os.rename(src\_path, dest\_path)

for j in files[int(len(files) \* 0.8):]:

src\_path = os.path.join(files\_dir, j)

dest\_path = os.path.join(test\_dir, i, j)

os.rename(src\_path, dest\_path)

remove\_dir(files\_dir)

```

**These lines are responsible for splitting the dataset into training and testing data. The dataset directory is iterated, and if the name of the directory starts with 'new', then the contents of that directory are considered for splitting. The files are shuffled and then 80% of the files are moved to the training directory, and the remaining 20% are moved to the testing directory. The original directory is then removed.**

```

IMG\_SIZE = 224

BATCH\_SIZE = 32

SEED = 42

train\_gen = ImageDataGenerator(

rotation\_range = 20,

validation\_split = 0.2,

horizontal\_flip = True,

preprocessing\_function = tf.keras.applications.resnet50.preprocess\_input

)

test\_gen = ImageDataGenerator(

preprocessing\_function = tf.keras.applications.resnet50.preprocess\_input

)

```

**These lines set the image size, batch size, and seed values. `ImageDataGenerator` objects are created for training and testing data with some augmentation parameters. `preprocessing\_function` is set to apply the ResNet50 preprocessing function to the images.**

```

train\_batch = train\_gen.flow\_from\_directory(

directory = train\_dir,

target\_size = (IMG\_SIZE, IMG\_SIZE),

batch\_size = BATCH\_SIZE,

class\_mode = 'sparse',

color\_mode='rgb',

subset = 'training',

shuffle=True,

seed = SEED

)

```

**This line creates a generator that generates batches of augmented training data from**

```

img\_shape = (IMG\_SIZE, IMG\_SIZE, 3)

```

**This line sets the size of the input images to be passed into the neural network. In this case, the input images are resized to IMG\_SIZE x IMG\_SIZE and have 3 color channels (RGB).**

```

base\_resnet = tf.keras.applications.resnet50.ResNet50(input\_shape=img\_shape, include\_top=False, weights='imagenet', pooling='avg')

print(base\_resnet.summary())

```

**This line initializes a pre-trained ResNet50 model from Keras with the specified input shape and weights trained on the ImageNet dataset. The include\_top parameter is set to False, so the final fully connected layer is not included in the model. The pooling parameter is set to "avg" to use average pooling. The summary function is called to print the architecture of the model.**

```

input = base\_resnet.input

x = base\_resnet.output

x = layers.Dense(512, activation='relu')(x)

x = layers.Dropout(0.5)(x)

x = layers.Dense(128, activation='relu')(x)

x = layers.Dropout(0.5)(x)

output = layers.Dense(2, activation='softmax')(x)

model = keras.Model(inputs=input, outputs=output)

print(model.summary())

```

**These lines create a new neural network by adding some fully connected layers on top of the pre-trained ResNet50 model. The input and output layers of the new model are set to the input and output layers of the pre-trained ResNet50 model, respectively. Dense layers with 512 and 128 units are added, and Dropout layers are used for regularization. The output layer has 2 units with a softmax activation function for binary classification. The model summary is printed to show the architecture of the new model.**

```

model.compile(

optimizer = keras.optimizers.Adam(learning\_rate=0.0001),

loss = keras.losses.SparseCategoricalCrossentropy(),

metrics = ['accuracy']

)

```

**This line compiles the model and sets the optimizer, loss function, and evaluation metric for training. The Adam optimizer is used with a learning rate of 0.0001. SparseCategoricalCrossentropy loss function is used for classification problems with more than two classes. The metric used to evaluate the model during training is accuracy.**

```

checkpoint = ModelCheckpoint(

filepath = 'D:/Github/CVPR/Finalterm/checkpoints/resnet50\_checkpoint\_{epoch:02d}.h5',

monitor='val\_loss',

save\_best\_only=True,

mode='auto',

save\_freq = 'epoch'

)

early\_stp = EarlyStopping(monitor='val\_loss', min\_delta=0, patience=5, restore\_best\_weights=True)

```

**These lines create two callbacks, ModelCheckpoint and EarlyStopping, to be used during training. ModelCheckpoint saves the model weights after each epoch if the validation loss improves. EarlyStopping stops training if the validation loss does not improve after a certain number of epochs (patience) and restores the weights of the best performing epoch.**

```

hist = model.fit(

train\_batch,

validation\_data = valid\_batch,

epochs = 50,

batch\_size=BATCH\_SIZE,

callbacks = [checkpoint, early\_stp]

)

```

**This line trains the model on the training dataset for 50 epochs with a batch size of BATCH\_SIZE. The validation dataset is used for validation during training. The ModelCheckpoint and EarlyStopping callbacks are used during training.**

```

test\_loss, test\_acc = model.evaluate(test\_batch, verbose=1)

print('Test loss:', test\_loss)

print('\nTest accuracy:', test\_acc)

```

**The code is testing the trained model on a test dataset and saving the model as a .h5 file. It then loads the saved model and uses it to perform real-time face mask detection on a live video stream.**

```

test\_loss, test\_acc = model.evaluate(test\_batch, verbose=1)

print('Test loss:', test\_loss)

print('\nTest accuracy:', test\_acc)

```

**The `model.evaluate()` function is used to evaluate the model on the test data. The test loss and accuracy are printed to the console.**

```

model.save("D:/Github/CVPR/Finalterm/mask\_detector.h5")

model = load\_model("D:/Github/CVPR/Finalterm/mask\_detector.h5")

```

**The `model.save()` function is used to save the trained model as a .h5 file. The `load\_model()` function is then used to load the saved model from the file.**

```

def preprocess\_f(frame):

frame = cv2.resize(frame, (IMG\_SIZE, IMG\_SIZE))

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

frame = np.expand\_dims(frame, axis=0)

frame = tf.keras.applications.resnet50.preprocess\_input(frame)

return frame

```

**The `preprocess\_f()` function is defined to preprocess each frame of the live video stream before passing it to the model for prediction. The function resizes the frame to the specified IMG\_SIZE, converts it from BGR to RGB color space, adds an extra dimension to the frame, and applies the ResNet50 preprocess\_input() function to the frame.**

```

capture = cv2.VideoCapture(0)

if not capture.isOpened():

raise IOError("cannot open camera")

while(capture.isOpened()):

ret, frame = capture.read()

if ret:

frame\_copy = frame.copy()

frame\_copy = preprocess\_f(frame\_copy)

preds = model.predict(frame\_copy)

label = np.argmax(preds)

if label == 0:

cv2.putText(frame, "Mask Detected", (30, 30), cv2.FONT\_HERSHEY\_SIMPLEX, 0.8, (0, 255, 0), 2)

elif label == 1:

cv2.putText(frame, "No Mask Detected", (30, 30), cv2.FONT\_HERSHEY\_SIMPLEX, 0.8, (100, 100, 255), 2)

cv2.imshow('Mask Detection', frame)

c = cv2.waitKey(3)

if c == 27: # ASCII of Esc

break

else:

break

capture.release()

cv2.destroyAllWindows()

```

**The code captures a live video stream from the default camera (specified by the value 0). If the camera cannot be opened, an IOError is raised. The code then enters a loop to read each frame of the video stream until the user presses the Esc key. Each frame is copied and preprocessed using the `preprocess\_f()` function. The preprocessed frame is then passed to the model for prediction using the `model.predict()` function. The label with the highest probability is determined using `np.argmax()`, and a corresponding text label is added to the frame using `cv2.putText()`. The frame with the text label is displayed using `cv2.imshow()`. If the user presses the Esc key, the loop is exited and the video stream is released and the windows are closed using `capture.release()` and `cv2.destroyAllWindows()`.**