*# import system libs*

**import** os

**import** time

**import** shutil

**import** pathlib

**import** itertools

*# import data handling tools*

**import** cv2

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

**import** pandas **as** pd

**import** seaborn **as** sns

sns**.**set\_style('darkgrid')

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

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

**from** sklearn.metrics **import** confusion\_matrix, classification\_report

*# import Deep learning Libraries*

**import** tensorflow **as** tf

**from** tensorflow **import** keras

**from** tensorflow.keras.models **import** Sequential

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

**from** tensorflow.keras.metrics **import** categorical\_crossentropy

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

**from** tensorflow.keras.layers **import** Conv2D, MaxPooling2D, Flatten, Dense, Activation, Dropout, BatchNormalization

**from** tensorflow.keras **import** regularizers

*# Ignore Warnings*

**import** warnings

warnings**.**filterwarnings("ignore")

print ('modules loaded')

*# Generate data paths with labels*

**def** define\_paths(data\_dir):

filepaths **=** []

labels **=** []

foldsold **=** os**.**listdir(data\_dir)

folds **=** []

folds**.**append(foldsold[0])

folds**.**append(foldsold[2])

print(folds)

**for** fold **in** folds:

foldpath **=** os**.**path**.**join(data\_dir, fold)

filelist **=** os**.**listdir(foldpath)

**for** file **in** filelist:

fpath **=** os**.**path**.**join(foldpath, file)

filepaths**.**append(fpath)

**if** fold **==** 'CT\_COVID':

labels**.**append('CT\_COVID')

**elif** fold **==** 'CT\_NonCOVID':

labels**.**append('CT\_NonCOVID')

**return** filepaths, labels

*# Concatenate data paths with labels into one dataframe ( to later be fitted into the model )*

**def** define\_df(files, classes):

Fseries **=** pd**.**Series(files, name**=** 'filepaths')

Lseries **=** pd**.**Series(classes, name**=**'labels')

**return** pd**.**concat([Fseries, Lseries], axis**=** 1)

*# Split dataframe to train, valid, and test*

**def** create\_df(data\_dir):

*# train dataframe*

files, classes **=** define\_paths(data\_dir)

df **=** define\_df(files, classes)

strat **=** df['labels']

train\_df, dummy\_df **=** train\_test\_split(df, train\_size**=** 0.8, shuffle**=** **True**, random\_state**=** 123, stratify**=** strat)

*# valid and test dataframe*

strat **=** dummy\_df['labels']

valid\_df, test\_df **=** train\_test\_split(dummy\_df, train\_size**=** 0.5, shuffle**=** **True**, random\_state**=** 123, stratify**=** strat)

**return** train\_df, valid\_df, test\_df

**def** create\_gens (train\_df, valid\_df, test\_df, batch\_size):

'''

This function takes train, validation, and test dataframe and fit them into image data generator, because model takes data from image data generator.

Image data generator converts images into tensors. '''

*# define model parameters*

img\_size **=** (224, 224)

channels **=** 3 *# either BGR or Grayscale*

color **=** 'rgb'

img\_shape **=** (img\_size[0], img\_size[1], channels)

*# Recommended : use custom function for test data batch size, else we can use normal batch size.*

ts\_length **=** len(test\_df)

test\_batch\_size **=** max(sorted([ts\_length **//** n **for** n **in** range(1, ts\_length **+** 1) **if** ts\_length**%n** == 0 and ts\_length/n <= 80]))

test\_steps **=** ts\_length **//** test\_batch\_size

*# This function which will be used in image data generator for data augmentation, it just take the image and return it again.*

**def** scalar(img):

**return** img

tr\_gen **=** ImageDataGenerator(preprocessing\_function**=** scalar, horizontal\_flip**=** **True**)

ts\_gen **=** ImageDataGenerator(preprocessing\_function**=** scalar)

train\_gen **=** tr\_gen**.**flow\_from\_dataframe( train\_df, x\_col**=** 'filepaths', y\_col**=** 'labels', target\_size**=** img\_size, class\_mode**=** 'categorical',

color\_mode**=** color, shuffle**=** **True**, batch\_size**=** batch\_size)

valid\_gen **=** ts\_gen**.**flow\_from\_dataframe( valid\_df, x\_col**=** 'filepaths', y\_col**=** 'labels', target\_size**=** img\_size, class\_mode**=** 'categorical',

color\_mode**=** color, shuffle**=** **True**, batch\_size**=** batch\_size)

*# Note: we will use custom test\_batch\_size, and make shuffle= false*

test\_gen **=** ts\_gen**.**flow\_from\_dataframe( test\_df, x\_col**=** 'filepaths', y\_col**=** 'labels', target\_size**=** img\_size, class\_mode**=** 'categorical',

color\_mode**=** color, shuffle**=** **False**, batch\_size**=** test\_batch\_size)

**return** train\_gen, valid\_gen, test\_gen

**def** show\_images(gen):

'''

This function take the data generator and show sample of the images

'''

*# return classes , images to be displayed*

g\_dict **=** gen**.**class\_indices *# defines dictionary {'class': index}*

classes **=** list(g\_dict**.**keys()) *# defines list of dictionary's kays (classes), classes names : string*

images, labels **=** next(gen) *# get a batch size samples from the generator*

*# calculate number of displayed samples*

length **=** len(labels) *# length of batch size*

sample **=** min(length, 25) *# check if sample less than 25 images*

plt**.**figure(figsize**=** (20, 20))

**for** i **in** range(sample):

plt**.**subplot(5, 5, i **+** 1)

image **=** images[i] **/** 255 *# scales data to range (0 - 255)*

plt**.**imshow(image)

index **=** np**.**argmax(labels[i]) *# get image index*

class\_name **=** classes[index] *# get class of image*

plt**.**title(class\_name, color**=** 'blue', fontsize**=** 12)

plt**.**axis('off')

plt**.**show()

**def** plot\_label\_count(df, plot\_title):

'''

This function take df and plot labels value counts

'''

*# Define needed variables*

vcounts **=** df['labels']**.**value\_counts()

labels **=** vcounts**.**keys()**.**tolist()

values **=** vcounts**.**tolist()

lcount **=** len(labels)

**if** lcount **>** 55:

print('The number of labels is > 55, no plot will be produced')

**else**:

plot\_labels(lcount, labels, values, plot\_title)

**def** plot\_labels(lcount, labels, values, plot\_title):

width **=** lcount **\*** 4

width **=** np**.**min([width, 20])

plt**.**figure(figsize**=** (width, 5))

form **=** {'family': 'serif', 'color': 'blue', 'size': 25}

sns**.**barplot(x**=**labels, y**=**values)

plt**.**title(f'Images per Label in {plot\_title} data', fontsize**=** 24, color**=** 'blue')

plt**.**xticks(rotation**=** 90, fontsize**=** 18)

plt**.**yticks(fontsize**=** 18)

plt**.**xlabel('CLASS', fontdict**=** form)

yaxis\_label **=** 'IMAGE COUNT'

plt**.**ylabel(yaxis\_label, fontdict**=** form)

rotation **=** 'vertical' **if** lcount **>=** 8 **else** 'horizontal'

**for** i **in** range(lcount):

plt**.**text(i, values[i] **/** 2, str(values[i]), fontsize**=** 12,

rotation**=** rotation, color**=** 'yellow', ha**=** 'center')

plt**.**show()

**class** MyCallback(keras**.**callbacks**.**Callback):

**def** \_\_init\_\_(self, model, patience, stop\_patience, threshold, factor, batches, epochs, ask\_epoch):

super(MyCallback, self)**.**\_\_init\_\_()

self**.**model **=** model

self**.**patience **=** patience *# specifies how many epochs without improvement before learning rate is adjusted*

self**.**stop\_patience **=** stop\_patience *# specifies how many times to adjust lr without improvement to stop training*

self**.**threshold **=** threshold *# specifies training accuracy threshold when lr will be adjusted based on validation loss*

self**.**factor **=** factor *# factor by which to reduce the learning rate*

self**.**batches **=** batches *# number of training batch to run per epoch*

self**.**epochs **=** epochs

self**.**ask\_epoch **=** ask\_epoch

self**.**ask\_epoch\_initial **=** ask\_epoch *# save this value to restore if restarting training*

*# callback variables*

self**.**count **=** 0 *# how many times lr has been reduced without improvement*

self**.**stop\_count **=** 0

self**.**best\_epoch **=** 1 *# epoch with the lowest loss*

self**.**initial\_lr **=** float(tf**.**keras**.**backend**.**get\_value(model**.**optimizer**.**lr)) *# get the initial learning rate and save it*

self**.**highest\_tracc **=** 0.0 *# set highest training accuracy to 0 initially*

self**.**lowest\_vloss **=** np**.**inf *# set lowest validation loss to infinity initially*

self**.**best\_weights **=** self**.**model**.**get\_weights() *# set best weights to model's initial weights*

self**.**initial\_weights **=** self**.**model**.**get\_weights() *# save initial weights if they have to get restored*

*# Define a function that will run when train begins*

**def** on\_train\_begin(self, logs**=** **None**):

msg **=** 'Do you want model asks you to halt the training [y/n] ?'

print(msg)

ans **=** input('')

**if** ans **in** ['Y', 'y']:

self**.**ask\_permission **=** 1

**elif** ans **in** ['N', 'n']:

self**.**ask\_permission **=** 0

msg **=** '{0:^8s}{1:^10s}{2:^9s}{3:^9s}{4:^9s}{5:^9s}{6:^9s}{7:^10s}{8:10s}{9:^8s}'**.**format('Epoch', 'Loss', 'Accuracy', 'V\_loss', 'V\_acc', 'LR', 'Next LR', 'Monitor','% Improv', 'Duration')

print(msg)

self**.**start\_time **=** time**.**time()

**def** on\_train\_end(self, logs**=** **None**):

stop\_time **=** time**.**time()

tr\_duration **=** stop\_time **-** self**.**start\_time

hours **=** tr\_duration **//** 3600

minutes **=** (tr\_duration **-** (hours **\*** 3600)) **//** 60

seconds **=** tr\_duration **-** ((hours **\*** 3600) **+** (minutes **\*** 60))

msg **=** f'training elapsed time was {str(hours)} hours, {minutes:4.1f} minutes, {seconds:4.2f} seconds)'

print(msg)

*# set the weights of the model to the best weights*

self**.**model**.**set\_weights(self**.**best\_weights)

**def** on\_train\_batch\_end(self, batch, logs**=** **None**):

*# get batch accuracy and loss*

acc **=** logs**.**get('accuracy') **\*** 100

loss **=** logs**.**get('loss')

*# prints over on the same line to show running batch count*

msg **=** '{0:20s}processing batch {1:} of {2:5s}- accuracy= {3:5.3f} - loss: {4:8.5f}'**.**format(' ', str(batch), str(self**.**batches), acc, loss)

print(msg, '\r', end**=** '')

**def** on\_epoch\_begin(self, epoch, logs**=** **None**):

self**.**ep\_start **=** time**.**time()

*# Define method runs on the end of each epoch*

**def** on\_epoch\_end(self, epoch, logs**=** **None**):

ep\_end **=** time**.**time()

duration **=** ep\_end **-** self**.**ep\_start

lr **=** float(tf**.**keras**.**backend**.**get\_value(self**.**model**.**optimizer**.**lr)) *# get the current learning rate*

current\_lr **=** lr

acc **=** logs**.**get('accuracy') *# get training accuracy*

v\_acc **=** logs**.**get('val\_accuracy') *# get validation accuracy*

loss **=** logs**.**get('loss') *# get training loss for this epoch*

v\_loss **=** logs**.**get('val\_loss') *# get the validation loss for this epoch*

**if** acc **<** self**.**threshold: *# if training accuracy is below threshold adjust lr based on training accuracy*

monitor **=** 'accuracy'

**if** epoch **==** 0:

pimprov **=** 0.0

**else**:

pimprov **=** (acc **-** self**.**highest\_tracc ) **\*** 100 **/** self**.**highest\_tracc *# define improvement of model progres*

**if** acc **>** self**.**highest\_tracc: *# training accuracy improved in the epoch*

self**.**highest\_tracc **=** acc *# set new highest training accuracy*

self**.**best\_weights **=** self**.**model**.**get\_weights() *# training accuracy improved so save the weights*

self**.**count **=** 0 *# set count to 0 since training accuracy improved*

self**.**stop\_count **=** 0 *# set stop counter to 0*

**if** v\_loss **<** self**.**lowest\_vloss:

self**.**lowest\_vloss **=** v\_loss

self**.**best\_epoch **=** epoch **+** 1 *# set the value of best epoch for this epoch*

**else**:

*# training accuracy did not improve check if this has happened for patience number of epochs*

*# if so adjust learning rate*

**if** self**.**count **>=** self**.**patience **-** 1: *# lr should be adjusted*

lr **=** lr **\*** self**.**factor *# adjust the learning by factor*

tf**.**keras**.**backend**.**set\_value(self**.**model**.**optimizer**.**lr, lr) *# set the learning rate in the optimizer*

self**.**count **=** 0 *# reset the count to 0*

self**.**stop\_count **=** self**.**stop\_count **+** 1 *# count the number of consecutive lr adjustments*

self**.**count **=** 0 *# reset counter*

**if** v\_loss **<** self**.**lowest\_vloss:

self**.**lowest\_vloss **=** v\_loss

**else**:

self**.**count **=** self**.**count **+** 1 *# increment patience counter*

**else**: *# training accuracy is above threshold so adjust learning rate based on validation loss*

monitor **=** 'val\_loss'

**if** epoch **==** 0:

pimprov **=** 0.0

**else**:

pimprov **=** (self**.**lowest\_vloss **-** v\_loss ) **\*** 100 **/** self**.**lowest\_vloss

**if** v\_loss **<** self**.**lowest\_vloss: *# check if the validation loss improved*

self**.**lowest\_vloss **=** v\_loss *# replace lowest validation loss with new validation loss*

self**.**best\_weights **=** self**.**model**.**get\_weights() *# validation loss improved so save the weights*

self**.**count **=** 0 *# reset count since validation loss improved*

self**.**stop\_count **=** 0

self**.**best\_epoch **=** epoch **+** 1 *# set the value of the best epoch to this epoch*

**else**: *# validation loss did not improve*

**if** self**.**count **>=** self**.**patience **-** 1: *# need to adjust lr*

lr **=** lr **\*** self**.**factor *# adjust the learning rate*

self**.**stop\_count **=** self**.**stop\_count **+** 1 *# increment stop counter because lr was adjusted*

self**.**count **=** 0 *# reset counter*

tf**.**keras**.**backend**.**set\_value(self**.**model**.**optimizer**.**lr, lr) *# set the learning rate in the optimizer*

**else**:

self**.**count **=** self**.**count **+** 1 *# increment the patience counter*

**if** acc **>** self**.**highest\_tracc:

self**.**highest\_tracc **=** acc

msg **=** f'{str(epoch **+** 1):^3s}/{str(self**.**epochs):4s} {loss:^9.3f}{acc **\*** 100:^9.3f}{v\_loss:^9.5f}{v\_acc **\*** 100:^9.3f}{current\_lr:^9.5f}{lr:^9.5f}{monitor:^11s}{pimprov:^10.2f}{duration:^8.2f}'

print(msg)

**if** self**.**stop\_count **>** self**.**stop\_patience **-** 1: *# check if learning rate has been adjusted stop\_count times with no improvement*

msg **=** f' training has been halted at epoch {epoch **+** 1} after {self**.**stop\_patience} adjustments of learning rate with no improvement'

print(msg)

self**.**model**.**stop\_training **=** **True** *# stop training*

**else**:

**if** self**.**ask\_epoch **!=** **None** **and** self**.**ask\_permission **!=** 0:

**if** epoch **+** 1 **>=** self**.**ask\_epoch:

msg **=** 'enter H to halt training or an integer for number of epochs to run then ask again'

print(msg)

ans **=** input('')

**if** ans **==** 'H' **or** ans **==** 'h':

msg **=** f'training has been halted at epoch {epoch **+** 1} due to user input'

print(msg)

self**.**model**.**stop\_training **=** **True** *# stop training*

**else**:

**try**:

ans **=** int(ans)

self**.**ask\_epoch **+=** ans

msg **=** f' training will continue until epoch {str(self**.**ask\_epoch)}'

print(msg)

msg **=** '{0:^8s}{1:^10s}{2:^9s}{3:^9s}{4:^9s}{5:^9s}{6:^9s}{7:^10s}{8:10s}{9:^8s}'**.**format('Epoch', 'Loss', 'Accuracy', 'V\_loss', 'V\_acc', 'LR', 'Next LR', 'Monitor', '% Improv', 'Duration')

print(msg)

**except** Exception:

print('Invalid')

**def** plot\_training(hist):

'''

This function take training model and plot history of accuracy and losses with the best epoch in both of them.

'''

*# Define needed variables*

tr\_acc **=** hist**.**history['accuracy']

tr\_loss **=** hist**.**history['loss']

val\_acc **=** hist**.**history['val\_accuracy']

val\_loss **=** hist**.**history['val\_loss']

index\_loss **=** np**.**argmin(val\_loss)

val\_lowest **=** val\_loss[index\_loss]

index\_acc **=** np**.**argmax(val\_acc)

acc\_highest **=** val\_acc[index\_acc]

Epochs **=** [i**+**1 **for** i **in** range(len(tr\_acc))]

loss\_label **=** f'best epoch= {str(index\_loss **+** 1)}'

acc\_label **=** f'best epoch= {str(index\_acc **+** 1)}'

*# Plot training history*

plt**.**figure(figsize**=** (20, 8))

plt**.**style**.**use('fivethirtyeight')

plt**.**subplot(1, 2, 1)

plt**.**plot(Epochs, tr\_loss, 'r', label**=** 'Training loss')

plt**.**plot(Epochs, val\_loss, 'g', label**=** 'Validation loss')

plt**.**scatter(index\_loss **+** 1, val\_lowest, s**=** 150, c**=** 'blue', label**=** loss\_label)

plt**.**title('Training and Validation Loss')

plt**.**xlabel('Epochs')

plt**.**ylabel('Loss')

plt**.**legend()

plt**.**subplot(1, 2, 2)

plt**.**plot(Epochs, tr\_acc, 'r', label**=** 'Training Accuracy')

plt**.**plot(Epochs, val\_acc, 'g', label**=** 'Validation Accuracy')

plt**.**scatter(index\_acc **+** 1 , acc\_highest, s**=** 150, c**=** 'blue', label**=** acc\_label)

plt**.**title('Training and Validation Accuracy')

plt**.**xlabel('Epochs')

plt**.**ylabel('Accuracy')

plt**.**legend()

plt**.**tight\_layout

plt**.**show()

**def** plot\_confusion\_matrix(cm, classes, normalize**=** **False**, title**=** 'Confusion Matrix', cmap**=** plt**.**cm**.**Blues):

'''

This function plot confusion matrix method from sklearn package.

'''

plt**.**figure(figsize**=** (10, 10))

plt**.**imshow(cm, interpolation**=** 'nearest', cmap**=** cmap)

plt**.**title(title)

plt**.**colorbar()

tick\_marks **=** np**.**arange(len(classes))

plt**.**xticks(tick\_marks, classes, rotation**=** 45)

plt**.**yticks(tick\_marks, classes)

**if** normalize:

cm **=** cm**.**astype('float') **/** cm**.**sum(axis**=** 1)[:, np**.**newaxis]

print('Normalized Confusion Matrix')

**else**:

print('Confusion Matrix, Without Normalization')

print(cm)

thresh **=** cm**.**max() **/** 2.

**for** i, j **in** itertools**.**product(range(cm**.**shape[0]), range(cm**.**shape[1])):

plt**.**text(j, i, cm[i, j], horizontalalignment**=** 'center', color**=** 'white' **if** cm[i, j] **>** thresh **else** 'black')

plt**.**tight\_layout()

plt**.**ylabel('True Label')

plt**.**xlabel('Predicted Label')

data\_dir **=** '/kaggle/input/covidct'

**try**:

*# Get splitted data*

train\_df, valid\_df, test\_df **=** create\_df(data\_dir)

*# Get Generators*

batch\_size **=** 40

train\_gen, valid\_gen, test\_gen **=** create\_gens(train\_df, valid\_df, test\_df, batch\_size)

**except**:

print('Invalid Input')