### Importing the necessary libraries

import numpy as np

import pandas as pd

import seaborn as sns

import tensorflow as tf

import matplotlib.pyplot as plt

import os

from distutils.dir\_util import copy\_tree, remove\_tree

from PIL import Image

from random import randint

from imblearn.over\_sampling import SMOTE

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

from sklearn.metrics import matthews\_corrcoef as MCC

from sklearn.metrics import balanced\_accuracy\_score as BAS

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

import tensorflow\_addons as tfa

from keras.utils.vis\_utils import plot\_model

from tensorflow.keras import Sequential, Input

from tensorflow.keras.layers import Dense, Dropout

from tensorflow.keras.layers import Conv2D, Flatten

from tensorflow.keras.callbacks import EarlyStopping

from tensorflow.keras.applications.inception\_v3 import InceptionV3

from tensorflow.keras.preprocessing.image import ImageDataGenerator as IDG

from tensorflow.keras.layers import SeparableConv2D, BatchNormalization, MaxPool2D

print("TensorFlow Version:", tf.\_\_version\_\_)

### Data Pre-Processing

base\_dir = "/kaggle/input/alzheimers-dataset-4-class-of-images/Alzheimer\_s Dataset/"

root\_dir = "./"

test\_dir = base\_dir + "test/"

train\_dir = base\_dir + "train/"

work\_dir = root\_dir + "dataset/"

if os.path.exists(work\_dir):

remove\_tree(work\_dir)

os.mkdir(work\_dir)

copy\_tree(train\_dir, work\_dir)

copy\_tree(test\_dir, work\_dir)

print("Working Directory Contents:", os.listdir(work\_dir))

WORK\_DIR = './dataset/'

CLASSES = [ 'NonDemented',

'VeryMildDemented',

'MildDemented',

'ModerateDemented']

IMG\_SIZE = 176

IMAGE\_SIZE = [176, 176]

DIM = (IMG\_SIZE, IMG\_SIZE)

#Performing Image Augmentation to have more data samples

ZOOM = [.99, 1.01]

BRIGHT\_RANGE = [0.8, 1.2]

HORZ\_FLIP = True

FILL\_MODE = "constant"

DATA\_FORMAT = "channels\_last"

work\_dr = IDG(rescale = 1./255, brightness\_range=BRIGHT\_RANGE, zoom\_range=ZOOM, data\_format=DATA\_FORMAT, fill\_mode=FILL\_MODE, horizontal\_flip=HORZ\_FLIP)

train\_data\_gen = work\_dr.flow\_from\_directory(directory=WORK\_DIR, target\_size=DIM, batch\_size=6500, shuffle=False)

def show\_images(generator,y\_pred=None):

"""

Input: An image generator,predicted labels (optional)

Output: Displays a grid of 9 images with lables

"""

# get image lables

labels =dict(zip([0,1,2,3], CLASSES))

# get a batch of images

x,y = generator.next()

# display a grid of 9 images

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

if y\_pred is None:

for i in range(9):

ax = plt.subplot(3, 3, i + 1)

idx = randint(0, 6400)

plt.imshow(x[idx])

plt.axis("off")

plt.title("Class:{}".format(labels[np.argmax(y[idx])]))

else:

for i in range(9):

ax = plt.subplot(3, 3, i + 1)

plt.imshow(x[i])

plt.axis("off")

plt.title("Actual:{} \nPredicted:{}".format(labels[np.argmax(y[i])],labels[y\_pred[i]]))

# Display Train Images

show\_images(train\_data\_gen)

#Retrieving the data from the ImageDataGenerator iterator

train\_data, train\_labels = train\_data\_gen.next()

#Getting to know the dimensions of our dataset

print(train\_data.shape, train\_labels.shape)

#Performing over-sampling of the data, since the classes are imbalanced

sm = SMOTE(random\_state=42)

train\_data, train\_labels = sm.fit\_resample(train\_data.reshape(-1, IMG\_SIZE \* IMG\_SIZE \* 3), train\_labels)

train\_data = train\_data.reshape(-1, IMG\_SIZE, IMG\_SIZE, 3)

print(train\_data.shape, train\_labels.shape)

#Splitting the data into train, test, and validation sets

train\_data, test\_data, train\_labels, test\_labels = train\_test\_split(train\_data, train\_labels, test\_size = 0.2, random\_state=42)

train\_data, val\_data, train\_labels, val\_labels = train\_test\_split(train\_data, train\_labels, test\_size = 0.2, random\_state=42)

### Constructing a Convolutional Neural Network Architecture

def conv\_block(filters, act='relu'):

"""Defining a Convolutional NN block for a Sequential CNN model. """

block = Sequential()

block.add(Conv2D(filters, 3, activation=act, padding='same'))

block.add(Conv2D(filters, 3, activation=act, padding='same'))

block.add(BatchNormalization())

block.add(MaxPool2D())

return block

def dense\_block(units, dropout\_rate, act='relu'):

"""Defining a Dense NN block for a Sequential CNN model. """

block = Sequential()

block.add(Dense(units, activation=act))

block.add(BatchNormalization())

block.add(Dropout(dropout\_rate))

return block

def construct\_model(act='relu'):

"""Constructing a Sequential CNN architecture for performing the classification task. """

model = Sequential([

Input(shape=(\*IMAGE\_SIZE, 3)),

Conv2D(16, 3, activation=act, padding='same'),

Conv2D(16, 3, activation=act, padding='same'),

MaxPool2D(),

conv\_block(32),

conv\_block(64),

conv\_block(128),

Dropout(0.2),

conv\_block(256),

Dropout(0.2),

Flatten(),

dense\_block(512, 0.7),

dense\_block(128, 0.5),

dense\_block(64, 0.3),

Dense(4, activation='softmax')

], name = "cnn\_model")

return model

#Defining a custom callback function to stop training our model when accuracy goes above 99%

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

def on\_epoch\_end(self, epoch, logs={}):

if logs.get('val\_acc') > 0.99:

print("\nReached accuracy threshold! Terminating training.")

self.model.stop\_training = True

my\_callback = MyCallback()

#EarlyStopping callback to make sure model is always learning

early\_stopping = EarlyStopping(monitor='val\_loss', patience=2)

#Defining other parameters for our CNN model

model = construct\_model()

METRICS = [tf.keras.metrics.CategoricalAccuracy(name='acc'),

tf.keras.metrics.AUC(name='auc'),

tfa.metrics.F1Score(num\_classes=4)]

CALLBACKS = [my\_callback]

model.compile(optimizer='adam',

loss=tf.losses.CategoricalCrossentropy(),

metrics=METRICS)

model.summary()

### Training & Testing the Model

#Fit the training data to the model and validate it using the validation data

EPOCHS = 100

history = model.fit(train\_data, train\_labels, validation\_data=(val\_data, val\_labels), callbacks=CALLBACKS, epochs=EPOCHS)

#Plotting the trend of the metrics during training

fig, ax = plt.subplots(1, 3, figsize = (30, 5))

ax = ax.ravel()

for i, metric in enumerate(["acc", "auc", "loss"]):

ax[i].plot(history.history[metric])

ax[i].plot(history.history["val\_" + metric])

ax[i].set\_title("Model {}".format(metric))

ax[i].set\_xlabel("Epochs")

ax[i].set\_ylabel(metric)

ax[i].legend(["train", "val"])

#Evaluating the model on the data

#train\_scores = model.evaluate(train\_data, train\_labels)

#val\_scores = model.evaluate(val\_data, val\_labels)

test\_scores = model.evaluate(test\_data, test\_labels)

#print("Training Accuracy: %.2f%%"%(train\_scores[1] \* 100))

#print("Validation Accuracy: %.2f%%"%(val\_scores[1] \* 100))

print("Testing Accuracy: %.2f%%"%(test\_scores[1] \* 100))

#Predicting the test data

pred\_labels = model.predict(test\_data)

#Print the classification report of the tested data

#Since the labels are softmax arrays, we need to roundoff to have it in the form of 0s and 1s,

#similar to the test\_labels

def roundoff(arr):

"""To round off according to the argmax of each predicted label array. """

arr[np.argwhere(arr != arr.max())] = 0

arr[np.argwhere(arr == arr.max())] = 1

return arr

for labels in pred\_labels:

labels = roundoff(labels)

print(classification\_report(test\_labels, pred\_labels, target\_names=CLASSES))

#Plot the confusion matrix to understand the classification in detail

pred\_ls = np.argmax(pred\_labels, axis=1)

test\_ls = np.argmax(test\_labels, axis=1)

conf\_arr = confusion\_matrix(test\_ls, pred\_ls)

plt.figure(figsize=(8, 6), dpi=80, facecolor='w', edgecolor='k')

ax = sns.heatmap(conf\_arr, cmap='Greens', annot=True, fmt='d', xticklabels=CLASSES, yticklabels=CLASSES)

plt.title('Alzheimer\'s Disease Diagnosis')

plt.xlabel('Prediction')

plt.ylabel('Truth')

plt.show(ax)

#Printing some other classification metrics

print("Balanced Accuracy Score: {} %".format(round(BAS(test\_ls, pred\_ls) \* 100, 2)))

print("Matthew's Correlation Coefficient: {} %".format(round(MCC(test\_ls, pred\_ls) \* 100, 2)))

#Saving the model for future use

model\_dir = work\_dir + "alzheimer\_cnn\_model"

model.save(model\_dir, save\_format='h5')

os.listdir(work\_dir)

pretrained\_model = tf.keras.models.load\_model(model\_dir)

#Check its architecture

plot\_model(pretrained\_model, to\_file=work\_dir + "model\_plot.png", show\_shapes=True, show\_layer\_names=True)