This project aims to classify images into two categories: **Demented** and **Non-Demented** using transfer learning with **InceptionV3** (a pre-trained convolutional neural network).

**1.CODE**

import tensorflow as tf

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

from tensorflow.keras.preprocessing.image import ImageDataGenerator

from tensorflow.keras.layers import Dense, GlobalAveragePooling2D

from tensorflow.keras.models import Model

**EXPLANATION**

 **TensorFlow**: A machine learning framework.

 **InceptionV3**: A pre-trained deep learning model optimized for image classification.

 **ImageDataGenerator**: Helps preprocess images

 **Dense**: Fully connected layers to add to the network.

 **GlobalAveragePooling2D**: Reduces spatial dimensions before the dense layers.

 **Model**: Combines layers into a complete model.

**2.CODE**

base\_model = InceptionV3(weights='imagenet', include\_top=False, input\_shape=(299, 299, 3))

**EXPLANATION**

 **weights='imagenet'**: Loads weights from the ImageNet dataset.

 **include\_top=False**: Excludes the pre-trained classification head to customize for our problem.

 **input\_shape=(299, 299, 3)**: Specifies input image size as 299x299 with 3 color channels (RGB).

**3.CODE**

x = base\_model.output

x = GlobalAveragePooling2D()(x)

x = Dense(1024, activation='relu')(x)

predictions = Dense(1, activation='sigmoid')(x)

model = Model(inputs=base\_model.input, outputs=predictions)

**EXPLANATION**

* **base\_model.output**: Extracts features from the pre-trained model.
* **GlobalAveragePooling2D**: Reduces the feature map to a vector by averaging.
* **Dense(1024, activation='relu')**: Adds a dense layer with 1024 neurons and ReLU activation to learn complex patterns.
* **Dense(1, activation='sigmoid')**: Outputs a single value (0 or 1) using a sigmoid activation for binary classification.
* **Model**: Combines the base model with the custom head into a complete model.

**4.CODE**

train\_datagen = ImageDataGenerator(

rescale=1./255,

shear\_range=0.2,

zoom\_range=0.2,

horizontal\_flip=True

)

validation\_datagen = ImageDataGenerator(rescale=1./255)

**EXPLANATION**

ImageDataGenerator is a tool provided by TensorFlow/Keras to preprocess IMAGE DATA

 **Data Normalization:** Adjusting pixel values so the model trains better.

 **Data Augmentation:** Modifying images in different ways to artificially increase the size of the training dataset and make the model generalize better.

train\_datagen is a variable where you define how to preprocess and augment **training data**.

**Normalization:**

* Images in datasets often have pixel values ranging from 0 to 255.
* Neural networks perform better when input values are scaled to a small range (e.g., 0 to 1).
* rescale=1./255 divides every pixel value by 255, converting the range from [0, 255] to [0, 1].

**Data Augmentation:** -

Data augmentation creates slightly altered copies of images to:

* Increase dataset size.
* Improve the model’s ability to generalize (reduce overfitting).

The following are data augmentation techniques used here:

* **shear\_range=0.2:**
  + Shearing distorts the image by slanting it. Imagine pulling the image diagonally, like pushing a rectangle into a parallelogram.
  + A value of 0.2 means the image can be sheared up to 20%.

** zoom\_range=0.2:**

* + Randomly zooms into the image by up to 20%. This simulates seeing the image closer.
* **horizontal\_flip=True:**
  + Randomly flips images horizontally (left to right). For example, a picture of a person facing right becomes one of them facing left

**Why Augmentation for Training Data?**

* Training a neural network requires as much variety as possible. Augmented images act as new, slightly different training examples, helping the model learn better.

### Why is validation\_datagen different?

Validation data is used to check how well the model is performing **without altering the original images.**

* For validation, you only normalize the pixel values (rescale=1./255) without augmentation.
* The reason is that you want to evaluate the model on unaltered images, as this better reflects its real-world performance.

**EXAMPLE**

Imagine you're teaching a model to recognize dogs in images:

* **Training data:** You show the model images of dogs, but you change their angles (shear), sizes (zoom), and flip them. This is like preparing the dog to recognize its owner in any situation.
* **Validation data:** You just show the dog normal pictures to check if it recognizes its owner without confusion.

**5. CODE**

**Loading the Dataset**

train\_generator = train\_datagen.flow\_from\_directory(

'Dataset/train',

target\_size=(299, 299),

batch\_size=32,

class\_mode='binary'

)

validation\_generator = validation\_datagen.flow\_from\_directory(

'Dataset/test',

target\_size=(299, 299),

batch\_size=32,

class\_mode='binary'

)

**EXPLANATION**

**flow\_from\_directory**: Loads images from folders.

* **target\_size=(299, 299)**: Resizes images to match the input shape of InceptionV3.
* **batch\_size=32**: Processes 32 images at a time.
* **class\_mode='binary'**: Indicates binary classification.

**CODE**

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

**EXPLANATION**

* **optimizer='adam'**: Optimizes the weights to minimize the loss.
* **loss='binary\_crossentropy'**: Measures the error for binary classification.
* **metrics=['accuracy']**: Tracks the accuracy during training.

**TRANING THE MODEL**

**CODE**

model.fit(

train\_generator,

steps\_per\_epoch=train\_generator.samples // train\_generator.batch\_size,

validation\_data=validation\_generator,

validation\_steps=validation\_generator.samples // validation\_generator.batch\_size,

epochs=20

)

**EXPLANATION**

**fit**: Trains the model on the training data.

* **steps\_per\_epoch**: Number of batches to process in each epoch.
* **validation\_data**: Uses validation data to evaluate performance during training.
* **epochs=20**: Number of training iterations.

Great question! Let’s break down **why we divide** in the steps\_per\_epoch and validation\_steps parameters in simple terms so you can understand it deeply.

### Understanding steps\_per\_epoch and validation\_steps

1. **steps\_per\_epoch:**
   * During training, the model processes the training data in batches.
   * This parameter tells the model **how many batches to process in one epoch**.
2. **validation\_steps:**
   * Similarly, this parameter tells the model **how many batches to process in one validation pass** (during evaluation after each epoch).

### What is a Batch?

A **batch** is a small subset of your dataset that the model processes at one time.

* Imagine you have 1,000 training images and you set batch\_size=32. This means:
  + The model processes **32 images at a time** instead of all 1,000 at once (which would be computationally expensive).
  + These 32 images form **one batch**.
  + After processing one batch, the model moves to the next batch, and so on, until it has seen all the training data once.

**Why Division (//)?**

* It divides the total number of images by the batch size to determine how many batches are needed to process the entire dataset.
* The batch size defines **how many images are processed in one go**.
* For example:
  + If the **batch\_size = 32**, the model will study batches of 32 images.
  + After processing one batch, it moves to the next.
* If the entire dataset can be evenly divided into batches of 32, then theoretically, **we know the number of steps required** to process the full dataset.

For example:

* If you have 1,000 training images:
  + steps=1000/32=31.25
  + steps=1000/32=31.25
  + Here, you **know** it needs 31 complete steps and 1 partial step.

.