UNet.ipynb

import os

import cv2

import numpy as np

from glob import glob

from tqdm import tqdm

import tensorflow as tf

import matplotlib.pyplot as plt

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Conv2D, MaxPooling2D, UpSampling2D, Dropout, concatenate, Conv2DTranspose, Input

from tensorflow.keras.layers import Multiply, Add

from tensorflow.keras.metrics import MeanIoU

from tensorflow.keras.callbacks import Callback, ModelCheckpoint

def load\_image(image\_path, SIZE):

image = cv2.imread(image\_path)

image = cv2.resize(image, (SIZE, SIZE))

image = image.astype(np.float32) / 255.0

return image

def load\_images(paths, SIZE, trim=None, channels=3):

if trim is not None:

paths = paths[:trim]

images = []

for path in paths:

img = load\_image(path, SIZE)

if channels == 1:

img = img[:, :, :1]

images.append(img)

return np.array(images)

def show\_image(image, title=None, cmap=None, alpha=1.0):

plt.imshow(image, cmap=cmap, alpha=alpha)

if title is not None:

plt.title(title)

plt.axis('off')

def show\_mask(image, mask, cmap='jet', alpha=0.2):

show\_image(image)

show\_image(np.squeeze(mask), cmap=cmap, alpha=alpha)

# Model Definition

def conv\_block(inputs=None, n\_filters=32, dropout\_prob=0, max\_pooling=True):

conv = Conv2D(n\_filters, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(inputs)

conv = Conv2D(n\_filters, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(conv)

if dropout\_prob > 0:

conv = Dropout(dropout\_prob)(conv)

if max\_pooling:

next\_layer = MaxPooling2D()(conv)

else:

next\_layer = conv

skip\_connection = conv

return next\_layer, skip\_connection

def upsampling\_block(expansive\_input, contractive\_input, n\_filters=32):

up = Conv2DTranspose(n\_filters, 3, strides=(2, 2), padding='same')(expansive\_input)

merge = concatenate([up, contractive\_input], axis=3)

conv = Conv2D(n\_filters, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(merge)

conv = Conv2D(n\_filters, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(conv)

return conv

def unet\_model(input\_size=(96, 128, 3), n\_filters=32, n\_classes=1):

inputs = Input(input\_size)

cblock1 = conv\_block(inputs, n\_filters)

cblock2 = conv\_block(cblock1[0], n\_filters \* 2)

cblock3 = conv\_block(cblock2[0], n\_filters \* 4)

cblock4 = conv\_block(cblock3[0], n\_filters \* 8, dropout\_prob=0.3)

cblock5 = conv\_block(cblock4[0], n\_filters\*16, dropout\_prob=0.3, max\_pooling=False)

ublock6 = upsampling\_block(cblock5[0], cblock4[1], n\_filters \* 8)

ublock7 = upsampling\_block(ublock6, cblock3[1], n\_filters \* 4)

ublock8 = upsampling\_block(ublock7, cblock2[1], n\_filters \* 2)

ublock9 = upsampling\_block(ublock8, cblock1[1], n\_filters)

conv9 = Conv2D(n\_filters,

3,

activation='relu',

padding='same',

kernel\_initializer='he\_normal')(ublock9)

conv10 = Conv2D(n\_classes, 1, padding='same', activation='sigmoid')(conv9)

model = tf.keras.Model(inputs=inputs, outputs=conv10)

return model

# Data paths

root\_path = "C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/Covid"

image\_path = os.path.join(root\_path, 'images/')

mask\_path = os.path.join(root\_path, 'masks/')

# Load image paths

image\_paths = sorted(glob(os.path.join(image\_path, '\*.png')))

mask\_paths = [path.replace('images', 'masks') for path in image\_paths]

# Load images and masks

images = load\_images(image\_paths, SIZE=256, trim=2000)

masks = load\_images(mask\_paths, SIZE=256, trim=2000, channels=1)

# Display sample

if len(images) > 0 and len(masks) > 0:

show\_mask(images[0], masks[0], alpha=0.2)

else:

print("Images or masks array is empty.")

# # Display sample

# show\_mask(images[0], masks[0], alpha=0.2)

# Model Compilation and Training

img\_height = 256

img\_width = 256

num\_channels = 3

unet = unet\_model((img\_height, img\_width, num\_channels))

unet.compile(optimizer='adam',

loss='binary\_crossentropy',

metrics=['accuracy', MeanIoU(2)])

# Training

BATCH\_SIZE = 8

SPE = len(images) // BATCH\_SIZE

unet.fit(

images, masks,

validation\_split=0.1,

epochs=1,

steps\_per\_epoch=SPE

)

# Display predictions

plt.figure(figsize=(15, 20))

n = 0

for i in range(1, 21):

plt.subplot(5, 4, i)

if n == 0:

id = np.random.randint(len(images))

image, mask = images[id], masks[id]

plt.title("Original Mask")

show\_mask(image, mask)

n += 1

elif n == 1:

pred\_mask = unet.predict(image[np.newaxis, ...])[0]

plt.title("Predicted Mask")

show\_mask(image, pred\_mask)

n += 1

elif n == 2:

id = np.random.randint(len(images))

image, mask = images[id], masks[id]

plt.title("Original Mask")

show\_mask(image, mask)

n += 1

elif n == 3:

pred\_mask = unet.predict(image[np.newaxis, ...])[0]

plt.title("Predicted Mask")

show\_mask(image, pred\_mask)

n = 0

plt.tight\_layout()

plt.show()

output\_path = os.path.join(root\_path, 'segmented/Covid/')

os.makedirs(output\_path, exist\_ok=True)

for i in range(len(images)):

image = images[i]

pred\_mask = unet.predict(image[np.newaxis, ...])[0]

# Save the original image

cv2.imwrite(os.path.join(output\_path, f'original\_{i}.png'), cv2.cvtColor((image \* 255).astype(np.uint8), cv2.COLOR\_RGB2BGR))

# Save the predicted mask

cv2.imwrite(os.path.join(output\_path, f'predicted\_mask\_{i}.png'), (pred\_mask \* 255).astype(np.uint8))

# Data paths

bacterialRootPath = "C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/Bacterial"

bacterialImagePath = os.path.join(bacterialRootPath, 'images/')

bacterialmaskPath = os.path.join(bacterialRootPath, 'masks/')

# Load image paths

bacterialImagePaths = sorted(glob(os.path.join(bacterialImagePath, '\*.png')))

bacterialmaskPaths = [path.replace('images', 'masks') for path in bacterialImagePaths]

# Load images and masks

images = load\_images(bacterialImagePaths, SIZE=256, trim=2000)

masks = load\_images(bacterialmaskPaths, SIZE=256, trim=2000, channels=1)

# Display sample

if len(images) > 0 and len(masks) > 0:

show\_mask(images[0], masks[0], alpha=0.2)

else:

print("Images or masks array is empty.")

# # Display sample

# show\_mask(images[0], masks[0], alpha=0.2)

# Model Compilation and Training

img\_height = 256

img\_width = 256

num\_channels = 3

unet = unet\_model((img\_height, img\_width, num\_channels))

unet.compile(optimizer='adam',

loss='binary\_crossentropy',

metrics=['accuracy', MeanIoU(2)])

BATCH\_SIZE = 8

SPE = len(images) // BATCH\_SIZE

unet.fit(

images, masks,

validation\_split=0.1,

epochs=1,

steps\_per\_epoch=SPE

)

# Display predictions

plt.figure(figsize=(15, 20))

n = 0

for i in range(1, 21):

plt.subplot(5, 4, i)

if n == 0:

id = np.random.randint(len(images))

image, mask = images[id], masks[id]

plt.title("Original Mask")

show\_mask(image, mask)

n += 1

elif n == 1:

pred\_mask = unet.predict(image[np.newaxis, ...])[0]

plt.title("Predicted Mask")

show\_mask(image, pred\_mask)

n += 1

elif n == 2:

id = np.random.randint(len(images))

image, mask = images[id], masks[id]

plt.title("Original Mask")

show\_mask(image, mask)

n += 1

elif n == 3:

pred\_mask = unet.predict(image[np.newaxis, ...])[0]

plt.title("Predicted Mask")

show\_mask(image, pred\_mask)

n = 0

plt.tight\_layout()

plt.show()

output\_path = os.path.join(bacterialRootPath, 'segmented/Bacterial/')

os.makedirs(output\_path, exist\_ok=True)

for i in range(len(images)):

image = images[i]

pred\_mask = unet.predict(image[np.newaxis, ...])[0]

cv2.imwrite(os.path.join(output\_path, f'original\_{i}.png'), cv2.cvtColor((image \* 255).astype(np.uint8), cv2.COLOR\_RGB2BGR))

cv2.imwrite(os.path.join(output\_path, f'predicted\_mask\_{i}.png'), (pred\_mask \* 255).astype(np.uint8))

healthyRootPath = "C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/Healthy"

healthyImagePath = os.path.join(healthyRootPath, 'images/')

healthymaskPath = os.path.join(healthyRootPath, 'masks/')

healthyImagePaths = sorted(glob(os.path.join(healthyImagePath, '\*.png')))

healthymaskPaths = [path.replace('images', 'masks') for path in healthyImagePaths]

images = load\_images(healthyImagePaths, SIZE=256, trim=2000)

masks = load\_images(healthymaskPaths, SIZE=256, trim=2000, channels=1)

# Display sample

if len(images) > 0 and len(masks) > 0:

show\_mask(images[0], masks[0], alpha=0.2)

else:

print("Images or masks array is empty.")

# show\_mask(images[0], masks[0], alpha=0.2)

img\_height = 256

img\_width = 256

num\_channels = 3

unet = unet\_model((img\_height, img\_width, num\_channels))

unet.compile(optimizer='adam',

loss='binary\_crossentropy',

metrics=['accuracy', MeanIoU(2)])

# Training

BATCH\_SIZE = 8

SPE = len(images) // BATCH\_SIZE

unet.fit(

images, masks,

validation\_split=0.1,

epochs=1,

steps\_per\_epoch=SPE

)

# Display predictions

plt.figure(figsize=(15, 20))

n = 0

for i in range(1, 21):

plt.subplot(5, 4, i)

if n == 0:

id = np.random.randint(len(images))

image, mask = images[id], masks[id]

plt.title("Original Mask")

show\_mask(image, mask)

n += 1

elif n == 1:

pred\_mask = unet.predict(image[np.newaxis, ...])[0]

plt.title("Predicted Mask")

show\_mask(image, pred\_mask)

n += 1

elif n == 2:

id = np.random.randint(len(images))

image, mask = images[id], masks[id]

plt.title("Original Mask")

show\_mask(image, mask)

n += 1

elif n == 3:

pred\_mask = unet.predict(image[np.newaxis, ...])[0]

plt.title("Predicted Mask")

show\_mask(image, pred\_mask)

n = 0

plt.tight\_layout()

plt.show()

output\_path = os.path.join(healthyRootPath, 'segmented/Healthy/')

os.makedirs(output\_path, exist\_ok=True)

for i in range(len(images)):

image = images[i]

pred\_mask = unet.predict(image[np.newaxis, ...])[0]

# Save the original image

# cv2.imwrite(os.path.join(output\_path, f'original\_{i}.png'), cv2.cvtColor((image \* 255).astype(np.uint8), cv2.COLOR\_RGB2BGR))

# Save the predicted mask

cv2.imwrite(os.path.join(output\_path, f'predicted\_mask\_{i}.png'), (pred\_mask \* 255).astype(np.uint8))

# Data paths

viralRootPath = "C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/Viral"

viralImagePath = os.path.join(viralRootPath, 'images/')

viralmaskPath = os.path.join(viralRootPath, 'masks/')

# Load image paths

viralImagePaths = sorted(glob(os.path.join(viralImagePath, '\*.png')))

viralmaskPaths = [path.replace('images', 'masks') for path in viralImagePaths]

# Load images and masks

images = load\_images(viralImagePaths, SIZE=256, trim=2000)

masks = load\_images(viralmaskPaths, SIZE=256, trim=2000, channels=1)

# Display sample

if len(images) > 0 and len(masks) > 0:

show\_mask(images[0], masks[0], alpha=0.2)

else:

print("Images or masks array is empty.")

# # Display sample

# show\_mask(images[0], masks[0], alpha=0.2)

# Model Compilation and Training

img\_height = 256

img\_width = 256

num\_channels = 3

unet = unet\_model((img\_height, img\_width, num\_channels))

unet.compile(optimizer='adam',

loss='binary\_crossentropy',

metrics=['accuracy', MeanIoU(2)])

# Training

BATCH\_SIZE = 8

SPE = len(images) // BATCH\_SIZE

unet.fit(

images, masks,

validation\_split=0.1,

epochs=1,

steps\_per\_epoch=SPE

)

# Display predictions

plt.figure(figsize=(15, 20))

n = 0

for i in range(1, 21):

plt.subplot(5, 4, i)

if n == 0:

id = np.random.randint(len(images))

image, mask = images[id], masks[id]

plt.title("Original Mask")

show\_mask(image, mask)

n += 1

elif n == 1:

pred\_mask = unet.predict(image[np.newaxis, ...])[0]

plt.title("Predicted Mask")

show\_mask(image, pred\_mask)

n += 1

elif n == 2:

id = np.random.randint(len(images))

image, mask = images[id], masks[id]

plt.title("Original Mask")

show\_mask(image, mask)

n += 1

elif n == 3:

pred\_mask = unet.predict(image[np.newaxis, ...])[0]

plt.title("Predicted Mask")

show\_mask(image, pred\_mask)

n = 0

plt.tight\_layout()

plt.show()

output\_path = os.path.join(viralRootPath, 'segmented/Viral/')

os.makedirs(output\_path, exist\_ok=True)

for i in range(len(images)):

image = images[i]

pred\_mask = unet.predict(image[np.newaxis, ...])[0]

# Save the original image

# cv2.imwrite(os.path.join(output\_path, f'original\_{i}.png'), cv2.cvtColor((image \* 255).astype(np.uint8), cv2.COLOR\_RGB2BGR))

# Save the predicted mask

cv2.imwrite(os.path.join(output\_path, f'predicted\_mask\_{i}.png'), (pred\_mask \* 255).astype(np.uint8))

import tensorflow as tf

# Create the U-Net model

model = unet\_model()

# Save the model

model.save("./unet\_model.h5")

Enhancement.ipynb

import os

import cv2

import matplotlib.pyplot as plt

import random

def apply\_clahe(input\_folder, output\_folder, clip\_limit=2.0, grid\_size=(8, 8)):

# Create output folder if it doesn't exist

os.makedirs(output\_folder, exist\_ok=True)

# Get a list of image files in the input folder

image\_files = [f for f in os.listdir(input\_folder) if f.lower().endswith(('.png', '.jpg', '.jpeg'))]

# Apply CLAHE to each image

for image\_file in image\_files:

# Read the image

image\_path = os.path.join(input\_folder, image\_file)

img = cv2.imread(image\_path, cv2.IMREAD\_COLOR)

# Convert the image to LAB color space

lab = cv2.cvtColor(img, cv2.COLOR\_BGR2LAB)

# Split the LAB image into L, A, and B channels

l, a, b = cv2.split(lab)

# Apply CLAHE to the L channel

clahe = cv2.createCLAHE(clipLimit=clip\_limit, tileGridSize=grid\_size)

cl = clahe.apply(l)

# Merge the enhanced L channel with the original A and B channels

enhanced\_lab = cv2.merge((cl, a, b))

# Convert the LAB image back to BGR color space

enhanced\_img = cv2.cvtColor(enhanced\_lab, cv2.COLOR\_LAB2BGR)

# Save the enhanced image to the output folder

output\_path = os.path.join(output\_folder, f'enhanced\_{image\_file}')

cv2.imwrite(output\_path, enhanced\_img)

clip\_limit = 2.0

grid\_size = (8, 8)

covidIP = "C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/Covid/segmented/Covid"

covidOP ="C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/Covid/segmented/enhanced"

# Apply CLAHE to the images in the input folder and save the enhanced images to the output folder

apply\_clahe(covidIP, covidOP, clip\_limit=clip\_limit, grid\_size=grid\_size)

bacterialIP = "C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/Bacterial/segmented/Bacterial"

bacterialOP ="C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/Bacterial/segmented/enhanced"

apply\_clahe(bacterialIP, bacterialOP, clip\_limit=clip\_limit, grid\_size=grid\_size)

HealthyIP = "C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/Healthy/segmented/Healthy"

HealthyOP ="C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/Healthy/segmented/enhanced"

apply\_clahe(HealthyIP, HealthyOP, clip\_limit=clip\_limit, grid\_size=grid\_size)

ViralIP = "C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/Viral/segmented/Viral"

ViralOP ="C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/Viral/segmented/enhanced"

apply\_clahe(ViralIP, ViralOP, clip\_limit=clip\_limit, grid\_size=grid\_size)

import numpy as np

# Replace this with your array of class values

class\_values = ['Covid', 'Bacterial', 'Healthy', 'Viral']

# Save the array to a .npy file

np.save('classes.npy', class\_values)

CXR-Net.ipynb:

import tensorflow as tf

from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D, concatenate, GlobalAveragePooling2D

def conv\_block(inputs, filters, kernel\_size=3, dropout\_prob=0.0):

conv = Conv2D(filters, kernel\_size, activation='relu', padding='same')(inputs)

conv = Conv2D(filters, kernel\_size, activation='relu', padding='same')(conv)

if dropout\_prob > 0.0:

conv = tf.keras.layers.Dropout(dropout\_prob)(conv)

return conv

def cxr\_net\_feature\_extraction(input\_size=(256, 256, 3), n\_filters=32, dropout\_prob=0.0):

inputs = Input(input\_size)

# Encoder

encoder\_layers = []

for i in range(5):

conv = conv\_block(inputs if i == 0 else encoder\_layers[-1], n\_filters \* (2\*\*i), dropout\_prob=dropout\_prob)

encoder\_layers.append(conv)

encoder\_layers.append(MaxPooling2D(pool\_size=(2, 2))(encoder\_layers[-1]))

# Bottleneck

bottleneck = conv\_block(encoder\_layers[-1], n\_filters \* 32, dropout\_prob=dropout\_prob)

# Feature extraction block

feature\_output = GlobalAveragePooling2D()(bottleneck)

model = tf.keras.Model(inputs=inputs, outputs=feature\_output)

return model

# Create the CXR-NET for feature extraction model

cxr\_net\_feature\_extraction\_model = cxr\_net\_feature\_extraction(input\_size=(256, 256, 3), n\_filters=32, dropout\_prob=0.3)

# Print the model summary

cxr\_net\_feature\_extraction\_model.summary()

import tensorflow as tf

from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D, concatenate, GlobalAveragePooling2D

def conv\_block(inputs, filters, kernel\_size=3, dropout\_prob=0.0):

conv = Conv2D(filters, kernel\_size, activation='relu', padding='same')(inputs)

conv = Conv2D(filters, kernel\_size, activation='relu', padding='same')(conv)

if dropout\_prob > 0.0:

conv = tf.keras.layers.Dropout(dropout\_prob)(conv)

return conv

def cxr\_net\_feature\_extraction(input\_size=(256, 256, 3), n\_filters=32, dropout\_prob=0.0):

inputs = Input(input\_size)

# Encoder

encoder\_layers = []

for i in range(6): # Increased encoder layers to 6

conv = conv\_block(inputs if i == 0 else encoder\_layers[-1], n\_filters \* (2\*\*i), dropout\_prob=dropout\_prob)

encoder\_layers.append(conv)

encoder\_layers.append(MaxPooling2D(pool\_size=(2, 2))(encoder\_layers[-1]))

# Bottleneck

bottleneck = conv\_block(encoder\_layers[-1], n\_filters \* 64, dropout\_prob=dropout\_prob)

# Feature extraction block

feature\_output = GlobalAveragePooling2D()(bottleneck)

# Decoder

decoder\_layers = []

for i in range(5, -1, -1): # Increased decoder layers to 6

upsample = UpSampling2D(size=(2, 2))(bottleneck if i == 5 else decoder\_layers[-1])

concat = concatenate([encoder\_layers[i\*2], upsample], axis=-1)

conv = conv\_block(concat, n\_filters \* (2\*\*i), dropout\_prob=dropout\_prob)

decoder\_layers.append(conv)

# Output

output = Conv2D(1, 1, activation='sigmoid')(decoder\_layers[-1])

model = tf.keras.Model(inputs=inputs, outputs=[output, feature\_output])

return model

# Create the CXR-NET for feature extraction model

cxr\_net\_feature\_extraction\_model = cxr\_net\_feature\_extraction(input\_size=(256, 256, 3), n\_filters=32, dropout\_prob=0.3)

# Print the model summary

cxr\_net\_feature\_extraction\_model.summary()

import tensorflow as tf

from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D, concatenate, GlobalAveragePooling2D

def conv\_block(inputs, filters, kernel\_size=3, dropout\_prob=0.0):

conv = Conv2D(filters, kernel\_size, activation='relu', padding='same')(inputs)

conv = Conv2D(filters, kernel\_size, activation='relu', padding='same')(conv)

if dropout\_prob > 0.0:

conv = tf.keras.layers.Dropout(dropout\_prob)(conv)

return conv

def cxr\_net\_feature\_extraction(input\_size=(256, 256, 3), n\_filters=32, dropout\_prob=0.0):

inputs = Input(input\_size)

# Encoder

encoder\_layers = []

for i in range(6): # Increased encoder layers to 6

conv = conv\_block(inputs if i == 0 else encoder\_layers[-1], n\_filters \* (2\*\*i), dropout\_prob=dropout\_prob)

encoder\_layers.append(conv)

encoder\_layers.append(MaxPooling2D(pool\_size=(2, 2))(encoder\_layers[-1]))

# Bottleneck

bottleneck = conv\_block(encoder\_layers[-1], n\_filters \* 64, dropout\_prob=dropout\_prob)

# Decoder

decoder\_layers = []

for i in range(5, -1, -1): # Increased decoder layers to 6

upsample = UpSampling2D(size=(2, 2))(bottleneck if i == 5 else decoder\_layers[-1])

concat = concatenate([encoder\_layers[i\*2], upsample], axis=-1)

conv = conv\_block(concat, n\_filters \* (2\*\*i), dropout\_prob=dropout\_prob)

decoder\_layers.append(conv)

# Additional Encoder Layer

additional\_encoder = conv\_block(decoder\_layers[-1], n\_filters \* 32, dropout\_prob=dropout\_prob)

encoder\_layers.append(additional\_encoder)

encoder\_layers.append(MaxPooling2D(pool\_size=(2, 2))(encoder\_layers[-1]))

# Feature extraction block

feature\_output = GlobalAveragePooling2D()(encoder\_layers[-1])

# Output

output = Conv2D(1, 1, activation='sigmoid')(decoder\_layers[-1])

model = tf.keras.Model(inputs=inputs, outputs=[output, feature\_output])

return model

# Create the CXR-NET for feature extraction model

cxr\_net\_feature\_extraction\_model = cxr\_net\_feature\_extraction(input\_size=(256, 256, 3), n\_filters=32, dropout\_prob=0.3)

# Print the model summary

cxr\_net\_feature\_extraction\_model.summary()

import os

import numpy as np

from tensorflow.keras.preprocessing import image

from tqdm import tqdm

# Set the path to the folder containing images

input\_folder\_path = "C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/Bacterial/segmented/Bacterial"

# Set the path to save the .npy files

output\_folder\_path = "C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Bacterial"

# Create the output folder if it doesn't exist

os.makedirs(output\_folder\_path, exist\_ok=True)

# Get a list of image files in the input folder

image\_files = [f for f in os.listdir(input\_folder\_path) if f.endswith(('.png', '.jpg', '.jpeg'))]

# Load the CXR-NET feature extraction model

cxr\_net\_feature\_extraction\_model = cxr\_net\_feature\_extraction(input\_size=(256, 256, 3), n\_filters=32, dropout\_prob=0.3)

# Adjust the number of epochs

num\_epochs = 50 # Set the desired number of epochs

# Iterate through each image and extract features

for img\_file in tqdm(image\_files, desc='Extracting Features'):

img\_path = os.path.join(input\_folder\_path, img\_file)

# Load and preprocess the image

img = image.load\_img(img\_path, target\_size=(256, 256))

img\_array = image.img\_to\_array(img)

img\_array = np.expand\_dims(img\_array, axis=0)

# Predict using the CXR-NET feature extraction model

predictions = cxr\_net\_feature\_extraction\_model.predict(img\_array)

# Get the features and save as .npy file

features = predictions[1] # Assuming the feature output is at index 1

feature\_file\_path = os.path.join(output\_folder\_path, f"{os.path.splitext(img\_file)[0]}\_features.npy")

np.save(feature\_file\_path, features)

import os

import numpy as np

from tensorflow.keras.preprocessing import image

from tqdm import tqdm

# Set the path to the folder containing images

input\_folder\_path = "C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/Covid/segmented/Covid"

# Set the path to save the .npy files

output\_folder\_path = "C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Covid"

# Create the output folder if it doesn't exist

os.makedirs(output\_folder\_path, exist\_ok=True)

# Get a list of image files in the input folder

image\_files = [f for f in os.listdir(input\_folder\_path) if f.endswith(('.png', '.jpg', '.jpeg'))]

# Load the CXR-NET feature extraction model

cxr\_net\_feature\_extraction\_model = cxr\_net\_feature\_extraction(input\_size=(256, 256, 3), n\_filters=32, dropout\_prob=0.3)

# Adjust the number of epochs

num\_epochs = 50 # Set the desired number of epochs

# Iterate through each image and extract features

for img\_file in tqdm(image\_files, desc='Extracting Features'):

img\_path = os.path.join(input\_folder\_path, img\_file)

# Load and preprocess the image

img = image.load\_img(img\_path, target\_size=(256, 256))

img\_array = image.img\_to\_array(img)

img\_array = np.expand\_dims(img\_array, axis=0)

# Predict using the CXR-NET feature extraction model

predictions = cxr\_net\_feature\_extraction\_model.predict(img\_array)

# Get the features and save as .npy file

features = predictions[1] # Assuming the feature output is at index 1

feature\_file\_path = os.path.join(output\_folder\_path, f"{os.path.splitext(img\_file)[0]}\_features.npy")

np.save(feature\_file\_path, features)

import os

import numpy as np

from tensorflow.keras.preprocessing import image

from tqdm import tqdm

# Set the path to the folder containing images

input\_folder\_path = "C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/Healthy/segmented/Healthy"

# Set the path to save the .npy files

output\_folder\_path = "C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Healthy"

# Create the output folder if it doesn't exist

os.makedirs(output\_folder\_path, exist\_ok=True)

# Get a list of image files in the input folder

image\_files = [f for f in os.listdir(input\_folder\_path) if f.endswith(('.png', '.jpg', '.jpeg'))]

# Load the CXR-NET feature extraction model

cxr\_net\_feature\_extraction\_model = cxr\_net\_feature\_extraction(input\_size=(256, 256, 3), n\_filters=32, dropout\_prob=0.3)

# Adjust the number of epochs

num\_epochs = 50 # Set the desired number of epochs

# Iterate through each image and extract features

for img\_file in tqdm(image\_files, desc='Extracting Features'):

img\_path = os.path.join(input\_folder\_path, img\_file)

# Load and preprocess the image

img = image.load\_img(img\_path, target\_size=(256, 256))

img\_array = image.img\_to\_array(img)

img\_array = np.expand\_dims(img\_array, axis=0)

# Predict using the CXR-NET feature extraction model

predictions = cxr\_net\_feature\_extraction\_model.predict(img\_array)

# Get the features and save as .npy file

features = predictions[1] # Assuming the feature output is at index 1

feature\_file\_path = os.path.join(output\_folder\_path, f"{os.path.splitext(img\_file)[0]}\_features.npy")

np.save(feature\_file\_path, features)

import os

import numpy as np

from tensorflow.keras.preprocessing import image

from tqdm import tqdm

# Set the path to the folder containing images

input\_folder\_path = "C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/Viral/segmented/Viral"

# Set the path to save the .npy files

output\_folder\_path = "C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Viral"

# Create the output folder if it doesn't exist

os.makedirs(output\_folder\_path, exist\_ok=True)

# Get a list of image files in the input folder

image\_files = [f for f in os.listdir(input\_folder\_path) if f.endswith(('.png', '.jpg', '.jpeg'))]

# Load the CXR-NET feature extraction model

cxr\_net\_feature\_extraction\_model = cxr\_net\_feature\_extraction(input\_size=(256, 256, 3), n\_filters=32, dropout\_prob=0.3)

# Adjust the number of epochs

num\_epochs = 50 # Set the desired number of epochs

# Iterate through each image and extract features

for img\_file in tqdm(image\_files, desc='Extracting Features'):

img\_path = os.path.join(input\_folder\_path, img\_file)

# Load and preprocess the image

img = image.load\_img(img\_path, target\_size=(256, 256))

img\_array = image.img\_to\_array(img)

img\_array = np.expand\_dims(img\_array, axis=0)

# Predict using the CXR-NET feature extraction model

predictions = cxr\_net\_feature\_extraction\_model.predict(img\_array)

# Get the features and save as .npy file

features = predictions[1] # Assuming the feature output is at index 1

feature\_file\_path = os.path.join(output\_folder\_path, f"{os.path.splitext(img\_file)[0]}\_features.npy")

np.save(feature\_file\_path, features)

Classification.ipynb:

import os

import numpy as np

# Set the paths to the folders

folder\_paths = [

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Bacterial",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Covid",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Healthy",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Viral"

]

all\_data = []

all\_labels = []

# Initialize an empty list to store the fused data

fused\_data = []

# Loop through each folder

for folder\_path in folder\_paths:

# Get a list of all files in the folder

files = os.listdir(folder\_path)

# Sort the files to maintain order if needed

files.sort()

# Initialize an empty list to store data from the current folder

folder\_data = []

# Loop through each file in the folder

for file\_name in files:

# Construct the full path to the file

file\_path = os.path.join(folder\_path, file\_name)

# Load data from the Numpy file

data = np.load(file\_path)

# Append data to the list for the current folder

folder\_data.append(data)

# Concatenate the data from the current folder and append to the fused\_data list

fused\_data.extend(folder\_data)

# Convert the list to a Numpy array if needed

fused\_data\_array = np.array(fused\_data)

# Now, 'fused\_data\_array' contains the concatenated data from all four folders

# Assuming fused\_data\_array contains the fused data from all four folders

# Initialize an empty list to store labels

label\_of\_fused\_model = []

# Number of classes and samples per class

num\_classes = 4

samples\_per\_class = 26

# Loop through each class and assign labels

for class\_label in range(num\_classes):

label\_of\_fused\_model.extend([class\_label] \* samples\_per\_class)

# Convert the list to a Numpy array if needed

label\_of\_fused\_model = np.array(label\_of\_fused\_model)

# all\_features = np.array(fused\_data\_array)

# all\_labels = np.array(label\_of\_fused\_model)

import numpy as np

import tensorflow as tf

from tensorflow.keras import layers, models

# Assuming the length of each feature vector is 1024

input\_shape = (1024,)

num\_classes = 4 # Replace with your actual number of classes

def create\_model(input\_shape, num\_classes):

model = models.Sequential([

layers.Input(shape=input\_shape),

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

layers.Dropout(0.5),

layers.Dense(256, activation='relu'),

layers.Dropout(0.5),

layers.Dense(num\_classes, activation='softmax')

])

return model

# Create an instance of the model

model = create\_model(input\_shape, num\_classes)

# Print model summary

model.summary()

model.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

# Assuming all\_features and all\_labels are your feature and label lists

# Preprocess the data if needed

# Convert to NumPy arrays or TensorFlow tensors

all\_features\_processed = np.squeeze(all\_features, axis=1) # Assuming the shape is (num\_samples, 1, 1024)

all\_features\_tensor = tf.convert\_to\_tensor(all\_features\_processed, dtype=tf.float32)

all\_labels\_tensor = tf.convert\_to\_tensor(all\_labels, dtype=tf.int64)

# Create a TensorFlow Dataset

dataset = tf.data.Dataset.from\_tensor\_slices((all\_features\_tensor, all\_labels\_tensor))

# Train the model

batch\_size = 32

epochs = 10

# Create a TensorFlow Dataset

dataset = tf.data.Dataset.from\_tensor\_slices((all\_features\_tensor, all\_labels\_tensor))

# Train the model

batch\_size = 32

epochs = 10

# Use the TensorFlow Dataset for training

train\_dataset = dataset.shuffle(buffer\_size=10000).batch(batch\_size).prefetch(buffer\_size=tf.data.AUTOTUNE)

# Fit the model

model.fit(train\_dataset, epochs=epochs)

# Save the trained model if needed

model.save('trained\_classifier\_model.h5').

# Assuming 'all\_features' is your input data

# Preprocess the data if needed

# For example, if 'all\_features' has shape (num\_samples, 1, 1024), reshape it to (num\_samples, 1024)

all\_features\_processed = np.squeeze(all\_features, axis=1)

# Convert to TensorFlow tensor

all\_features\_tensor = tf.convert\_to\_tensor(all\_features\_processed, dtype=tf.float32)

# Use the model to predict labels

predictions = model.predict(all\_features\_tensor)

# Get the predicted labels

predicted\_labels = np.argmax(predictions, axis=1)

# Now 'predicted\_labels' contains the predicted class labels for each input sample

print(predicted\_labels)

from sklearn.metrics import accuracy\_score

accuracy = accuracy\_score(all\_labels, predicted\_labels)

print("Accuracy:", accuracy)

import os

import numpy as np

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten, Dropout

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

from sklearn.preprocessing import LabelEncoder

# Load features and labels from all .npy files in a folder

def load\_data(folder\_path):

data = []

labels = []

for file\_name in os.listdir(folder\_path):

if file\_name.endswith('.npy'):

features = np.load(os.path.join(folder\_path, file\_name))

label = folder\_path.split(os.path.sep)[-1] # Assumes folder name is the label

data.append(features)

labels.extend([label] \* features.shape[0])

data = np.concatenate(data, axis=0)

return data, labels

# Define VGG16-based model

def create\_vgg16\_model(input\_shape, num\_classes):

model = Sequential()

model.add(Dense(256, activation='relu', input\_shape=input\_shape))

model.add(Flatten())

model.add(Dropout(0.5))

model.add(Dense(num\_classes, activation='softmax'))

return model

folder\_paths = [

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Bacterial",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Covid",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Healthy",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Viral"

]

all\_data = []

all\_labels = []

for folder\_path in folder\_paths:

folder\_data, folder\_labels = load\_data(folder\_path)

all\_data.append(folder\_data)

all\_labels.extend(folder\_labels)

data = np.concatenate(all\_data, axis=0)

# Encode labels

label\_encoder = LabelEncoder()

encoded\_labels = label\_encoder.fit\_transform(all\_labels)

num\_classes = len(label\_encoder.classes\_)

# Split data into train and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(data, encoded\_labels, test\_size=0.2, random\_state=42)

# Create and compile the model

input\_shape = X\_train.shape[1:] # Assuming X\_train has shape (number\_of\_samples, feature\_dim)

VGGmodel = create\_vgg16\_model(input\_shape=input\_shape, num\_classes=num\_classes)

VGGmodel.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

# Train the model

VGGmodel.fit(X\_train, y\_train, epochs=100, batch\_size=32, validation\_split=0.2)

# Evaluate the model on the test set

accuracy = VGGmodel.evaluate(X\_test, y\_test)[1]

print(f'Test accuracy: {accuracy}')

VGGmodel.save('vgg16\_model.h5')

from tensorflow.keras.applications import InceptionV3

from tensorflow.keras.layers import Input

# Create the InceptionV3 model

input\_tensor = Input(shape=(299, 299, 3))

inception\_model = InceptionV3(input\_tensor=input\_tensor, weights=None, include\_top=True)

# Print model summary

inception\_model.summary()

# image\_classifier.py

import os

import numpy as np

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Dense, GlobalAveragePooling2D

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

from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import accuracy\_score

# Function to load and preprocess features

def load\_and\_preprocess\_data(folder\_path, label):

features = []

labels = []

for filename in os.listdir(folder\_path):

if filename.endswith(".npy"):

file\_path = os.path.join(folder\_path, filename)

feature = np.load(file\_path)

features.append(feature)

labels.append(label)

if not features:

print(f"No features found in folder: {folder\_path}")

return np.array(features), np.array(labels)

def main():

# Paths to your folders

folder\_paths = [

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Bacterial",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Covid",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Healthy",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Viral"

]

labels = ['Covid', 'Bacterial', 'Healthy', 'Viral']

# Load and preprocess features from each folder

all\_features = []

all\_labels = []

for folder\_path, label in zip(folder\_paths, labels):

features, labels = load\_and\_preprocess\_data(folder\_path, label)

all\_features.append(features)

all\_labels.append(labels)

# Concatenate features and labels from all folders

X = np.concatenate(all\_features)

y = np.concatenate(all\_labels)

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Convert labels to numerical values

label\_encoder = LabelEncoder()

y\_train\_encoded = label\_encoder.fit\_transform(y\_train)

y\_test\_encoded = label\_encoder.transform(y\_test)

# Assume the features have the shape (your\_feature\_shape,)

input\_shape = X\_train.shape[1:]

# Create a simple model

input\_tensor = Input(shape=input\_shape)

x = Dense(256, activation='relu')(input\_tensor)

predictions = Dense(len(labels), activation='softmax')(x)

model = Model(inputs=input\_tensor, outputs=predictions)

# Compile the model

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

# Train the model

history = model.fit(X\_train, y\_train\_encoded, epochs=10, batch\_size=32, validation\_split=0.1)

# Evaluate the model on the test set

loss, accuracy = model.evaluate(X\_test, y\_test\_encoded)

print(f"Test Accuracy: {accuracy \* 100:.2f}%")

train\_accuracy = history.history['accuracy'][-1]

print(f"Train Accuracy: {train\_accuracy \* 100:.2f}%")

main()

# Accurate Inception v3

import os

import numpy as np

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten, Dropout

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

from sklearn.preprocessing import LabelEncoder

# Load features and labels from all .npy files in a folder

def load\_data(folder\_path):

data = []

labels = []

for file\_name in os.listdir(folder\_path):

if file\_name.endswith('.npy'):

features = np.load(os.path.join(folder\_path, file\_name))

label = folder\_path.split('/')[-1] # Assumes folder name is the label

data.append(features)

labels.extend([label] \* features.shape[0])

data = np.concatenate(data, axis=0)

return data, labels

# Define Inception model

def create\_inception\_model(input\_shape):

model = Sequential()

model.add(Flatten(input\_shape=input\_shape))

model.add(Dense(256, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(num\_classes, activation='softmax'))

return model

# Load data from four folders

folder\_paths = [

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Bacterial",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Covid",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Healthy",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Viral"

]

all\_data = []

all\_labels = []

for folder\_path in folder\_paths:

folder\_data, folder\_labels = load\_data(folder\_path)

all\_data.append(folder\_data)

all\_labels.extend(folder\_labels)

data = np.concatenate(all\_data, axis=0)

# Encode labels

label\_encoder = LabelEncoder()

encoded\_labels = label\_encoder.fit\_transform(all\_labels)

num\_classes = len(label\_encoder.classes\_)

# Split data into train and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(data, encoded\_labels, test\_size=0.2, random\_state=42)

# Create and compile the model

model = create\_inception\_model(input\_shape=data.shape[1:])

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

# Train the model

model.fit(X\_train, y\_train, epochs=10, batch\_size=32, validation\_split=0.2)

# Evaluate the model on the test set

accuracy = model.evaluate(X\_test, y\_test)[1]

print(f'Test accuracy: {accuracy\*100}')

# Optionally, save the model for later use

model.save('inception\_model.h5')

import tensorflow as tf

from tensorflow.keras import layers, models

# Define the AlexNet model architecture

def alexnet\_model(input\_shape=(227, 227, 3), num\_classes=1000):

model = models.Sequential()

# Layer 1

model.add(layers.Conv2D(96, (11, 11), strides=(4, 4), activation='relu', input\_shape=input\_shape))

model.add(layers.BatchNormalization())

model.add(layers.MaxPooling2D((3, 3), strides=(2, 2)))

# Layer 2

model.add(layers.Conv2D(256, (5, 5), padding='same', activation='relu'))

model.add(layers.BatchNormalization())

model.add(layers.MaxPooling2D((3, 3), strides=(2, 2)))

# Layer 3

model.add(layers.Conv2D(384, (3, 3), padding='same', activation='relu'))

# Layer 4

model.add(layers.Conv2D(384, (3, 3), padding='same', activation='relu'))

# Layer 5

model.add(layers.Conv2D(256, (3, 3), padding='same', activation='relu'))

model.add(layers.MaxPooling2D((3, 3), strides=(2, 2)))

# Flatten and fully connected layers

model.add(layers.Flatten())

model.add(layers.Dense(4096, activation='relu'))

model.add(layers.Dropout(0.5))

model.add(layers.Dense(4096, activation='relu'))

model.add(layers.Dropout(0.5))

model.add(layers.Dense(num\_classes, activation='softmax'))

return model

# Create the AlexNet model

model = alexnet\_model()

# Display the model summary

model.summary()

import tensorflow as tf

from tensorflow.keras import layers, models

def build\_adeco\_cnn(input\_shape):

# Define convolutional layers

model = tf.keras.Sequential([

layers.Conv2D(32, kernel\_size=(3, 3), activation='relu', input\_shape=input\_shape),

layers.MaxPooling2D(pool\_size=(2, 2)),

layers.Conv2D(64, kernel\_size=(3, 3), activation='relu'),

layers.MaxPooling2D(pool\_size=(2, 2)),

layers.Conv2D(128, kernel\_size=(3, 3), activation='relu'),

layers.MaxPooling2D(pool\_size=(2, 2)),

layers.Flatten(),

# Flatten and connect to dense layers

layers.Dense(256, activation='relu'),

layers.Dropout(0.5),

layers.Dense(1, activation='sigmoid') # Sigmoid for binary classification

])

model.compile(optimizer='adam',

loss='binary\_crossentropy',

metrics=['accuracy'])

return model

# Assuming input image dimensions are 120x120 pixels with 3 color channels

input\_shape = (120, 120, 3)

# Create the ADECO-CNN model

adeco\_cnn\_model = build\_adeco\_cnn(input\_shape)

# Display the model architecture

adeco\_cnn\_model.summary()

import os

import numpy as np

import tensorflow as tf

from tensorflow.keras import layers, models

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

from sklearn.preprocessing import LabelEncoder

# Load features and labels from all .npy files in a folder

def load\_data(folder\_path):

data = []

labels = []

for file\_name in os.listdir(folder\_path):

if file\_name.endswith('.npy'):

features = np.load(os.path.join(folder\_path, file\_name))

label = folder\_path.split('/')[-1] # Assumes folder name is the label

data.append(features)

labels.extend([label] \* features.shape[0])

data = np.concatenate(data, axis=0)

return data, labels

# Define ADECO-CNN model architecture

def build\_adeco\_cnn(input\_shape, num\_classes):

model = tf.keras.Sequential([

layers.Conv2D(32, kernel\_size=(3, 3), activation='relu', input\_shape=input\_shape),

layers.MaxPooling2D(pool\_size=(2, 2)),

layers.Conv2D(64, kernel\_size=(3, 3), activation='relu'),

layers.MaxPooling2D(pool\_size=(2, 2)),

layers.Conv2D(128, kernel\_size=(3, 3), activation='relu'),

layers.MaxPooling2D(pool\_size=(2, 2)),

layers.Flatten(),

layers.Dense(256, activation='relu'),

layers.Dropout(0.5),

layers.Dense(num\_classes, activation='softmax')

])

return model

folder\_paths = [

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Bacterial",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Covid",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Healthy",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Viral"

]

all\_data = []

all\_labels = []

for folder\_path in folder\_paths:

folder\_data, folder\_labels = load\_data(folder\_path)

all\_data.append(folder\_data)

all\_labels.extend(folder\_labels)

data = np.concatenate(all\_data, axis=0)

# Encode labels

label\_encoder = LabelEncoder()

encoded\_labels = label\_encoder.fit\_transform(all\_labels)

num\_classes = len(label\_encoder.classes\_)

# Split data into train and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(data, encoded\_labels, test\_size=0.2, random\_state=42)

# Reshape the data to represent images

input\_shape = (32, 32, 1) # Assuming grayscale images with dimensions 32x32

X\_train = X\_train.reshape(-1, 32, 32, 1)

X\_test = X\_test.reshape(-1, 32, 32, 1)

# Create and compile the ADECO-CNN model

model = build\_adeco\_cnn(input\_shape, num\_classes)

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

# Train the model

model.fit(X\_train, y\_train, epochs=10, batch\_size=32, validation\_split=0.2)

# Evaluate the model on the test set

accuracy = model.evaluate(X\_test, y\_test)[1]

print(f'Test accuracy: {accuracy\*100}')

# Optionally, save the model for later use

model.save('adeco\_cnn\_model.h5')

from keras.applications import ResNet50

from keras.layers import Input

from keras.models import Model

from keras import models

from keras.layers import GlobalAveragePooling2D, Dense

# Assuming the original images are 32x32 grayscale

input\_shape = (32, 32, 3)

# Create the ResNet50 model with the correct input shape and pre-trained weights

base\_model = ResNet50(weights='imagenet', include\_top=False, input\_shape=input\_shape)

# Add your custom layers on top of the base model

x = base\_model.output

x = GlobalAveragePooling2D()(x)

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

predictions = Dense(num\_classes, activation='softmax')(x)

# Combine the base model with your custom layers

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

# Compile the model

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

# Print model summary

model.summary()

from keras.applications import ResNet50

from keras.layers import Input

from keras.models import Model

from keras import models

from keras.layers import GlobalAveragePooling2D, Dense

# Assuming the original images are 32x32 grayscale

input\_shape = (32, 32, 3)

# Create the ResNet50 model with the correct input shape and pre-trained weights

base\_model = ResNet50(weights='imagenet', include\_top=False, input\_shape=input\_shape)

# Add your custom layers on top of the base model

x = base\_model.output

x = GlobalAveragePooling2D()(x)

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

predictions = Dense(num\_classes, activation='softmax')(x)

# Combine the base model with your custom layers

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

# Compile the model

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

# Print model summary

model.summary()

from keras.applications import vgg16, densenet, xception, nasnet, efficientnet

from keras.models import load\_model

densenet\_model = densenet.DenseNet121(weights='imagenet', include\_top=True)

xception\_model = xception.Xception(weights='imagenet', include\_top=True)

nasnet\_model = nasnet.NASNetLarge(weights='imagenet', include\_top=True)

efficientnet\_model = efficientnet.EfficientNetB7(weights='imagenet', include\_top=True)

from keras.models import load\_model

# Load the models

loaded\_densenet\_model = load\_model('densenet\_model.h5')

loaded\_xception\_model = load\_model('xception\_model.h5')

loaded\_nasnet\_model = load\_model('nasnet\_model.h5')

loaded\_efficientnet\_model = load\_model('efficientnet\_model.h5')

print("DenseNet121 model summary:")

print(loaded\_densenet\_model.summary())

print("Xception model summary:")

print(loaded\_xception\_model.summary())

print("NASNetLarge model summary:")

print(loaded\_nasnet\_model.summary())

print("EfficientNetB7 model summary:")

print(loaded\_efficientnet\_model.summary())

import os

import numpy as np

from tensorflow.keras import layers, models

from tensorflow.keras.applications import VGG19

from tensorflow.keras.preprocessing.image import ImageDataGenerator

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

from sklearn.preprocessing import LabelEncoder

# Load features and labels from all .npy files in a folder

def load\_data(folder\_path):

data = []

labels = []

for file\_name in os.listdir(folder\_path):

if file\_name.endswith('.npy'):

features = np.load(os.path.join(folder\_path, file\_name))

label = folder\_path.split('/')[-1] # Assumes folder name is the label

data.append(features)

labels.extend([label] \* features.shape[0])

data = np.concatenate(data, axis=0)

return data, labels

# Load data from four folders

folder\_paths = [

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Bacterial",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Covid",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Healthy",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Viral"

]

all\_data = []

all\_labels = []

for folder\_path in folder\_paths:

folder\_data, folder\_labels = load\_data(folder\_path)

all\_data.append(folder\_data)

all\_labels.extend(folder\_labels)

data = np.concatenate(all\_data, axis=0)

# Encode labels

label\_encoder = LabelEncoder()

encoded\_labels = label\_encoder.fit\_transform(all\_labels)

num\_classes = len(label\_encoder.classes\_)

# Split data into train and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(data.reshape(-1, 32, 32, 1), encoded\_labels, test\_size=0.2, random\_state=42)

# Convert grayscale images to RGB

X\_train = np.repeat(X\_train, 3, axis=-1)

X\_test = np.repeat(X\_test, 3, axis=-1)

# Reshape the data to represent RGB images

input\_shape = (32, 32, 3) # Assuming the original images are 32x32 RGB

X\_train = X\_train.reshape(-1, 32, 32, 3)

X\_test = X\_test.reshape(-1, 32, 32, 3)

# Create the VGG19 model with the correct input shape and pre-trained weights

base\_model = VGG19(weights='imagenet', include\_top=False, input\_shape=input\_shape)

# Add custom layers on top of the base model

x = base\_model.output

x = layers.Flatten()(x)

x = layers.Dense(256, activation='relu')(x)

predictions = layers.Dense(num\_classes, activation='softmax')(x)

# Combine the base model with the custom layers

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

# Compile the model

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

# Train the model

model.fit(X\_train, y\_train, epochs=10, batch\_size=32, validation\_split=0.2)

# Evaluate the model on the test set

accuracy = model.evaluate(X\_test, y\_test)[1]

print(f'Test accuracy: {(accuracy)\*100}')

# Optionally, save the model for later use

model.save('vgg19\_model.h5')

from tensorflow.keras.applications import DenseNet121

# Assuming the original images are 32x32 RGB

input\_shape = (32, 32, 3)

# Create the DenseNet121 model with the correct input shape and pre-trained weights

base\_model = DenseNet121(weights='imagenet', include\_top=False, input\_shape=input\_shape)

# Add custom layers on top of the base model

x = base\_model.output

x = layers.GlobalAveragePooling2D()(x) # Change from Flatten to GlobalAveragePooling2D for DenseNet

x = layers.Dense(256, activation='relu')(x)

predictions = layers.Dense(num\_classes, activation='softmax')(x)

# Combine the base model with the custom layers

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

# Compile the model

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

# Train the model

model.fit(X\_train, y\_train, epochs=10, batch\_size=32, validation\_split=0.2)

# Evaluate the model on the test set

accuracy = model.evaluate(X\_test, y\_test)[1]

print(f'Test accuracy: {(accuracy)\*100}')

# Optionally, save the model for later use

model.save('densenet121\_model.h5')

from tensorflow.keras.applications import EfficientNetB0

# Assuming the original images are 32x32 RGB

input\_shape = (32, 32, 3)

# Create the EfficientNetB0 model with the correct input shape and pre-trained weights

base\_model = EfficientNetB0(weights='imagenet', include\_top=False, input\_shape=input\_shape)

# Add custom layers on top of the base model

x = base\_model.output

x = layers.GlobalAveragePooling2D()(x) # Change from Flatten to GlobalAveragePooling2D for EfficientNet

x = layers.Dense(256, activation='relu')(x)

predictions = layers.Dense(num\_classes, activation='softmax')(x)

# Combine the base model with the custom layers

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

# Compile the model

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

# Train the model

model.fit(X\_train, y\_train, epochs=10, batch\_size=32, validation\_split=0.2)

# Evaluate the model on the test set

accuracy = model.evaluate(X\_test, y\_test)[1]

print(f'Test accuracy: {(accuracy)\*100}')

# Optionally, save the model for later use

model.save('efficientnetb0\_model.h5')

XAI.ipynb

import cv2

import numpy as np

import matplotlib.pyplot as plt

def apply\_clahe\_to\_single\_image(input\_image, clip\_limit=2.0, grid\_size=(8, 8)):

# Convert the image to LAB color space

lab = cv2.cvtColor(input\_image, cv2.COLOR\_BGR2LAB)

# Split the LAB image into L, A, and B channels

l, a, b = cv2.split(lab)

# Apply CLAHE to the L channel

clahe = cv2.createCLAHE(clipLimit=clip\_limit, tileGridSize=grid\_size)

cl = clahe.apply(l)

# Merge the enhanced L channel with the original A and B channels

enhanced\_lab = cv2.merge((cl, a, b))

# Convert the LAB image back to BGR color space

enhanced\_img = cv2.cvtColor(enhanced\_lab, cv2.COLOR\_LAB2BGR)

return enhanced\_img

def calculate\_enhancement\_percentage(original\_hist, enhanced\_hist):

original\_bins = np.arange(256)

enhanced\_bins = np.arange(256)

original\_total\_pixels = np.sum(original\_hist)

enhanced\_total\_pixels = np.sum(enhanced\_hist)

original\_cumulative\_sum = np.cumsum(original\_hist)

enhanced\_cumulative\_sum = np.cumsum(enhanced\_hist)

original\_cdf\_normalized = original\_cumulative\_sum / original\_total\_pixels

enhanced\_cdf\_normalized = enhanced\_cumulative\_sum / enhanced\_total\_pixels

diff\_cdf = np.abs(original\_cdf\_normalized - enhanced\_cdf\_normalized)

max\_diff = np.max(diff\_cdf)

enhancement\_percentage = (1 - max\_diff) \* 100

return enhancement\_percentage

# Set the input image path

input\_image\_path = "./Normal-34.png"

# Read the input image

input\_image = cv2.imread(input\_image\_path)

# Apply CLAHE to the single image

enhanced\_image = apply\_clahe\_to\_single\_image(input\_image, clip\_limit=2.0, grid\_size=(8, 8))

# Convert images to grayscale for histogram calculation

input\_gray = cv2.cvtColor(input\_image, cv2.COLOR\_BGR2GRAY)

enhanced\_gray = cv2.cvtColor(enhanced\_image, cv2.COLOR\_BGR2GRAY)

# Calculate histograms

original\_hist = cv2.calcHist([input\_gray], [0], None, [256], [0, 256])

enhanced\_hist = cv2.calcHist([enhanced\_gray], [0], None, [256], [0, 256])

# Calculate enhancement percentage

enhancement\_percentage = calculate\_enhancement\_percentage(original\_hist, enhanced\_hist)

# Display the original and enhanced images

cv2.imshow("Original Image", input\_image)

cv2.imshow("Enhanced Image", enhanced\_image)

cv2.waitKey(0)

cv2.destroyAllWindows()

# Plot histograms

plt.plot(original\_hist, color='blue', label='Original')

plt.plot(enhanced\_hist, color='red', label='Enhanced')

plt.title('Histogram')

plt.xlabel('Pixel Value')

plt.ylabel('Frequency')

plt.legend()

plt.show()

print(f"Enhancement Percentage: {enhancement\_percentage:.2f}%")

import os

import numpy as np

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten, Dropout

from transformers.utils import GENERATION\_CONFIG\_NAME

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

from sklearn.preprocessing import LabelEncoder

import shap

# Load features and labels from all .npy files in a folder

def load\_data(folder\_path):

data = []

labels = []

for file\_name in os.listdir(folder\_path):

if file\_name.endswith('.npy'):

features = np.load(os.path.join(folder\_path, file\_name))

label = folder\_path.split(os.path.sep)[-1] # Assumes folder name is the label

data.append(features)

labels.extend([label] \* features.shape[0])

data = np.concatenate(data, axis=0)

return data, labels

folder\_paths = [

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Bacterial",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Covid",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Healthy",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Viral"

]

all\_data = []

all\_labels = []

# Define VGG16-based model

def create\_vgg16\_model(input\_shape, num\_classes):

model = Sequential()

model.add(Dense(256, activation='relu', input\_shape=input\_shape))

model.add(Flatten())

model.add(Dropout(0.5))

model.add(Dense(num\_classes, activation='softmax'))

return model

for folder\_path in folder\_paths:

folder\_data, folder\_labels = load\_data(folder\_path)

all\_data.append(folder\_data)

all\_labels.extend(folder\_labels)

data = np.concatenate(all\_data, axis=0)

# Encode labels

label\_encoder = LabelEncoder()

encoded\_labels = label\_encoder.fit\_transform(all\_labels)

num\_classes = len(label\_encoder.classes\_)

# Split data into train and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(data, encoded\_labels, test\_size=0.2, random\_state=42)

# Create and compile the model

input\_shape = X\_train.shape[1:] # Assuming X\_train has shape (number\_of\_samples, feature\_dim)

VGGmodel = create\_vgg16\_model(input\_shape=input\_shape, num\_classes=num\_classes)

VGGmodel.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

VGGmodel.fit(X\_train, y\_train, epochs=10, batch\_size=32, validation\_split=0.2)

explainer = shap.Explainer(VGGmodel, masker=shap.maskers.Independent(X\_train))

shap\_values = explainer.shap\_values(X\_test)

accuracy = VGGmodel.evaluate(X\_test, y\_test)[1]

print(f'Test accuracy: {accuracy}')

shap.summary\_plot(shap\_values, X\_test)

import os

import numpy as np

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten, Dropout

from transformers.utils import GENERATION\_CONFIG\_NAME

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

from sklearn.preprocessing import LabelEncoder

import shap

# Load features and labels from all .npy files in a folder

def load\_data(folder\_path):

data = []

labels = []

for file\_name in os.listdir(folder\_path):

if file\_name.endswith('.npy'):

features = np.load(os.path.join(folder\_path, file\_name))

label = folder\_path.split(os.path.sep)[-1] # Assumes folder name is the label

data.append(features)

labels.extend([label] \* features.shape[0])

data = np.concatenate(data, axis=0)

return data, labels

folder\_paths = [

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Bacterial",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Covid",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Healthy",

"C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Dataset/masked/COVID-19\_Radiography\_Dataset/ExtractedFeatures/Viral"

]

all\_data = []

all\_labels = []

def create\_inception\_model(input\_shape):

model = Sequential()

model.add(Flatten(input\_shape=input\_shape))

model.add(Dense(256, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(num\_classes, activation='softmax'))

return model

for folder\_path in folder\_paths:

folder\_data, folder\_labels = load\_data(folder\_path)

all\_data.append(folder\_data)

all\_labels.extend(folder\_labels)

data = np.concatenate(all\_data, axis=0)

# Encode labels

label\_encoder = LabelEncoder()

encoded\_labels = label\_encoder.fit\_transform(all\_labels)

num\_classes = len(label\_encoder.classes\_)

# Split data into train and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(data, encoded\_labels, test\_size=0.2, random\_state=42)

# Create and compile the model

input\_shape = X\_train.shape[1:] # Assuming X\_train has shape (number\_of\_samples, feature\_dim)

incmodel = create\_inception\_model(input\_shape=input\_shape)

incmodel.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

incmodel.fit(X\_train, y\_train, epochs=10, batch\_size=32, validation\_split=0.2)

explainer = shap.Explainer(incmodel, masker=shap.maskers.Independent(X\_train))

shap\_values = explainer.shap\_values(X\_test)

accuracy = incmodel.evaluate(X\_test, y\_test)[1]

print(f'Test accuracy: {accuracy}')

shap.summary\_plot(shap\_values, X\_test)

shap.initjs()

print(f"Current label Shown: {encoded\_labels}")

shap.force\_plot(

base\_value=explainer.expected\_value[encoded\_labels[0]],

shap\_values=shap\_values[encoded\_labels[0]],

features=data[0:50, :],

)

Main.py

import streamlit as st

import cv2

import numpy as np

import tensorflow as tf

import matplotlib.pyplot as plt

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Conv2D, MaxPooling2D, UpSampling2D, Dropout, concatenate, Conv2DTranspose, Input

from tensorflow.keras.layers import Multiply, Add

from tensorflow.keras.metrics import MeanIoU

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Conv2D, MaxPooling2D, UpSampling2D, Dropout, concatenate, Conv2DTranspose, Input, Dense, Flatten, Dropout

from tensorflow.keras.models import Sequential

from sklearn.preprocessing import LabelEncoder

# CLAHE function

def apply\_clahe\_to\_single\_image(input\_image, clip\_limit=2.0, grid\_size=(8, 8)):

lab = cv2.cvtColor(input\_image, cv2.COLOR\_BGR2LAB)

l, a, b = cv2.split(lab)

clahe = cv2.createCLAHE(clipLimit=clip\_limit, tileGridSize=grid\_size)

cl = clahe.apply(l)

enhanced\_lab = cv2.merge((cl, a, b))

enhanced\_img = cv2.cvtColor(enhanced\_lab, cv2.COLOR\_LAB2BGR)

return enhanced\_img

# Function to load a single image

def load\_image(image\_path, SIZE):

image = cv2.imread(image\_path)

image = cv2.resize(image, (SIZE, SIZE))

image = image.astype(np.float32) / 255.0

return image

def load\_mask(mask\_path, SIZE):

mask = cv2.imread(mask\_path, cv2.IMREAD\_GRAYSCALE)

mask = cv2.resize(mask, (SIZE, SIZE))

mask = mask.astype(np.float32) / 255.0

mask = np.expand\_dims(mask, axis=-1)

return mask

def show\_image\_with\_mask(image, mask, title=None):

plt.imshow(image)

plt.imshow(mask, cmap='gray', alpha=0.2) # Use 'gray' colormap

if title is not None:

plt.title(title)

plt.axis('off')

def conv\_block(inputs=None, n\_filters=32, dropout\_prob=0, max\_pooling=True):

conv = Conv2D(n\_filters, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(inputs)

conv = Conv2D(n\_filters, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(conv)

if dropout\_prob > 0:

conv = Dropout(dropout\_prob)(conv)

if max\_pooling:

next\_layer = MaxPooling2D()(conv)

else:

next\_layer = conv

skip\_connection = conv

return next\_layer, skip\_connection

def upsampling\_block(expansive\_input, contractive\_input, n\_filters=32):

up = Conv2DTranspose(n\_filters, 3, strides=(2, 2), padding='same')(expansive\_input)

merge = concatenate([up, contractive\_input], axis=3)

conv = Conv2D(n\_filters, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(merge)

conv = Conv2D(n\_filters, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(conv)

return conv

def unet\_model(input\_size=(96, 128, 3), n\_filters=32, n\_classes=1):

inputs = Input(input\_size)

cblock1 = conv\_block(inputs, n\_filters)

cblock2 = conv\_block(cblock1[0], n\_filters \* 2)

cblock3 = conv\_block(cblock2[0], n\_filters \* 4)

cblock4 = conv\_block(cblock3[0], n\_filters \* 8, dropout\_prob=0.3)

cblock5 = conv\_block(cblock4[0], n\_filters\*16, dropout\_prob=0.3, max\_pooling=False)

ublock6 = upsampling\_block(cblock5[0], cblock4[1], n\_filters \* 8)

ublock7 = upsampling\_block(ublock6, cblock3[1], n\_filters \* 4)

ublock8 = upsampling\_block(ublock7, cblock2[1], n\_filters \* 2)

ublock9 = upsampling\_block(ublock8, cblock1[1], n\_filters)

conv9 = Conv2D(n\_filters,

3,

activation='relu',

padding='same',

kernel\_initializer='he\_normal')(ublock9)

conv10 = Conv2D(n\_classes, 1, padding='same', activation='sigmoid')(conv9)

model = tf.keras.Model(inputs=inputs, outputs=conv10)

return model

def load\_classification\_model():

inceptionv3\_model\_path = "C:/Users/HP/Desktop/Taboo/Final Year Project/Final-Year-Project/Semester VIII/inception\_model.h5" # Replace with your actual path

inceptionv3\_model = tf.keras.models.load\_model(inceptionv3\_model\_path)

chosen\_layer\_name = 'dense\_40'

num\_classes = 4

inception\_base\_model = Model(inputs=inceptionv3\_model.input, outputs=inceptionv3\_model.get\_layer(chosen\_layer\_name).output)

classification\_model = Sequential()

classification\_model.add(inception\_base\_model)

classification\_model.add(Dense(256, activation='relu'))

classification\_model.add(Dropout(0.5))

classification\_model.add(Dense(num\_classes, activation='softmax'))

classification\_model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

return classification\_model

from sklearn.decomposition import PCA

def extract\_features(image):

inceptionv3\_model = tf.keras.applications.InceptionV3(include\_top=False, weights='imagenet', input\_shape=(299,299,3))

preprocessed\_image = tf.keras.applications.inception\_v3.preprocess\_input(image)

features = inceptionv3\_model.predict(preprocessed\_image[np.newaxis, ...])

features = features.flatten()

features = features[:1024]

return features

unet = unet\_model((256, 256, 3))

unet.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy', MeanIoU(2)])

def main():

st.title("Segmentation and Classification")

uploaded\_image = st.file\_uploader("Upload an enhanced lung image", type=["jpg", "png", "jpeg"])

uploaded\_mask = st.file\_uploader("Upload the corresponding mask", type=["jpg", "png", "jpeg"])

if uploaded\_mask is not None:

mask\_bytes = np.asarray(bytearray(uploaded\_mask.read()), dtype=np.uint8)

mask\_image = cv2.imdecode(mask\_bytes, 0)

mask\_resized = cv2.resize(mask\_image, (256, 256))

mask\_resized = mask\_resized.astype(np.float32) / 255.0

if uploaded\_image is not None:

file\_bytes = np.asarray(bytearray(uploaded\_image.read()), dtype=np.uint8)

enhanced\_image = cv2.imdecode(file\_bytes, 1)

enhanced\_image = apply\_clahe\_to\_single\_image(enhanced\_image, clip\_limit=2.0, grid\_size=(8, 8))

features = extract\_features(enhanced\_image)

classification\_model = load\_classification\_model()

predicted\_class\_label = np.argmax(classification\_model.predict(features.reshape(1, -1)))

classes = ['Covid', 'Bacterial', 'Normal', 'Viral']

label\_encoder = LabelEncoder()

label\_encoder.fit(classes)

predicted\_class\_name = label\_encoder.classes\_[predicted\_class\_label]

st.image(enhanced\_image, caption="Predicted Class: " + predicted\_class\_name, use\_column\_width=True)

main()