

Configuration Manual

MSc Research Project
Data Analytics

Calista Clifford Gonsalves
Student ID: 22186077

School of Computing
National College of Ireland

Supervisor: Paul Stynes, Musfira Jilani and Mark Cudden

National College of Ireland
MSc Project Submission Sheet



School of Computing

Student Name: Calista Clifford Gonsalves

Student ID: 22186077

Programme: Msc Data Analytics **Year:** 2023-2024

Module: Research Project

Lecturer: Paul Stynes, Musfira Jilani and Mark Cudden

Submission Due Date: 12-08-2024

Project Title: Leveraging Machine Learning and GANs for Parkinson disease detection

Word Count: 948 **Page Count:** 16

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Calista Clifford Gonsalves

Date: 12-08-2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	<input type="checkbox"/>
Attach a Moodle submission receipt of the online project submission , to each project (including multiple copies).	<input type="checkbox"/>
You must ensure that you retain a HARD COPY of the project , both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	<input type="checkbox"/>

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Calista Gonsalves
Student ID:22186077

1 Introduction

The goal of this document is to provide a step-by-step analysis of how the implementation was carried out.

2 System Requirements

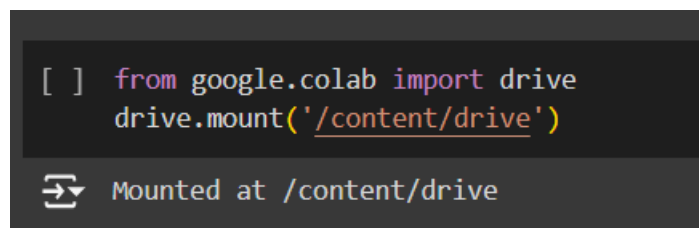
Operating System	Windows 11 Home
Installed Memory	8.00 GB RAM
Processor	Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz
Required memory for execution	Google Colab (TPUv2, 334 GB RAM)

Table 1: System Requirements

Python programming language has been used and the entire code is executed on Google Colab.

3 Project Development

3.1 Setting up of Google Colab



```
[ ] from google.colab import drive
    drive.mount('/content/drive')
```

Mounted at /content/drive

Figure 1: Mounting drive

3.2 Upload original data on drive

The original dataset should be uploaded on drive.

3.3 Importing Libraries

```

import os
import numpy as np
from PIL import Image
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
from torchvision import transforms
from torchvision.utils import save_image
import random

from skimage.metrics import structural_similarity as compare_ssim
import numpy as np

from PIL import Image, ImageOps
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report, recall_score
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet50 import preprocess_input
import tensorflow as tf
import gc
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report, recall_score, precision_score, cohen_kappa_score, f1_score
from PIL import Image, ImageEnhance

```

Figure 2: Importing libraries

3.4 Generating images using Custom GAN

Custom GAN is inspired by SRGAN¹, due to its architecture to generate high resolution images, similarly the Custom GAN architecture makes use of simple architecture to produce images of good quality.

Part 1: Loading images and applying transformation

Before passing images to Generator, it is necessary to ensure that the images are preprocessed.

```

dataset_path = '/content/drive/MyDrive/Dataset'

transform = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.ToTensor()
])

#dataset to include image paths and labels
class CustomImageFolder(Dataset):
    def __init__(self, root, transform=None):
        self.root = root
        self.transform = transform
        self.samples = []

        parkinson_path = os.path.join(root, 'Parkinson')
        healthy_path = os.path.join(root, 'Healthy')

        for label, class_path in enumerate([parkinson_path, healthy_path]):
            for fname in os.listdir(class_path):
                self.samples.append((os.path.join(class_path, fname), label))

    def __len__(self):
        return len(self.samples)

    def __getitem__(self, idx):
        path, label = self.samples[idx]
        image = Image.open(path).convert('RGB')
        if self.transform:
            image = self.transform(image)
        return image, label

def create_dataloader(dataset, label, batch_size=8):
    class_indices = [i for i, (_, lbl) in enumerate(dataset.samples) if lbl == label]
    class_subset = torch.utils.data.Subset(dataset, class_indices)
    return DataLoader(class_subset, batch_size=batch_size, shuffle=True, num_workers=2)

#loading dataset
dataset = CustomImageFolder(root=dataset_path, transform=transform)
batch_size = 8

```

Figure 3: Loading images and applying transformation for Custom GAN

¹ <https://medium.com/analytics-vidhya/super-resolution-gan-srgan-5e10438aec0c>

Part 2: Defining Generator Architecture

The generator class consists of multiple convolutional layers with ReLU activations and batch normalization to refine the image features. The output is an image with enhanced feature.

```
#generator and discriminator architectures
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=9, padding=4),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 3, kernel_size=3, padding=1),
        )

    def forward(self, x):
        return self.model(x)
```

Figure 4: Generator

Part 3: Defining Discriminator Architecture

The discriminator is used to distinguish between real and generated images

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Flatten(),
            nn.Linear(128 * 32 * 32, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.model(x)
```

Figure 5: Discriminator

Part 4: Training the model

```
#initializing optimizers with learning rates
optimizer_G = optim.Adam(generator.parameters(), lr=0.0001, betas=(0.5, 0.999))
optimizer_D = optim.Adam(discriminator.parameters(), lr=0.0001, betas=(0.5, 0.999))

#loss functions
criterion_GAN = nn.BCELoss().to(device)
criterion_content = nn.L1Loss().to(device)

#directories to save results
results_dir_parkinson = '/content/drive/MyDrive/SRGANTRIALWORKSEED/Parkinson'
results_dir_healthy = '/content/drive/MyDrive/SRGANTRIALWORKSEED/Healthy'
os.makedirs(results_dir_parkinson, exist_ok=True)
os.makedirs(results_dir_healthy, exist_ok=True)

# Training function
def train_model(data_loader, results_dir, num_epochs=20, initial_lr=0.0001, lr_decay_epoch=10):
    optimizer_G.param_groups[0]['lr'] = initial_lr #initial LR for generator
    optimizer_D.param_groups[0]['lr'] = initial_lr # initial LR for discriminator

    for epoch in range(num_epochs):
        if epoch > lr_decay_epoch:
            #reducing learning rate after lr_decay_epoch
            new_lr = initial_lr * (0.1 ** ((epoch - lr_decay_epoch) // 10))
            optimizer_D.param_groups[0]['lr'] = new_lr

        for i, (imgs, labels) in enumerate(data_loader):

            imgs_lr = imgs.to(device)
            labels = labels.to(device)

            #training Discriminator
            optimizer_D.zero_grad()

            gen_hr = generator(imgs_lr)

            #calculating discriminator loss with label smoothing
            valid = torch.ones(imgs_lr.size(0), 1, device=device) * 0.9 #labels for real images with label smoothing
            fake = torch.zeros(imgs_lr.size(0), 1, device=device) #labels for fake images

            real_loss = criterion_GAN(discriminator(imgs_lr), valid) #loss on real images
            fake_loss = criterion_GAN(discriminator(gen_hr.detach()), fake) #loss on generated images

            d_loss = (real_loss + fake_loss) / 2 #total discriminator loss

            d_loss.backward()
            optimizer_D.step()

            #training generator
            optimizer_G.zero_grad()

            gen_hr = generator(imgs_lr)

            #calculating generator loss
            g_loss_content = criterion_content(gen_hr, imgs_lr) #content loss

            #adversarial loss
            fake_validity = discriminator(gen_hr)
            g_loss_GAN = criterion_GAN(fake_validity, valid)

            g_loss = g_loss_content + 1e-3 * g_loss_GAN #total generator loss

            g_loss.backward()
            optimizer_G.step()

            #losses printed
            if i % 100 == 0:
                print(f"[Epoch {epoch}/{num_epochs}] [Batch {i}/{len(data_loader)}] [D loss: {d_loss.item():.4f}] [G loss: {g_loss.item():.4f}]")

            #generated images saved after the 10th epoch
            if epoch >= 10:
                for k in range(imgs_lr.size(0)):
                    save_path = os.path.join(results_dir, f"epoch_{epoch}_batch_{i}_img_{k}.png")
                    save_image(gen_hr[k], save_path, normalize=True)
```

Figure 6: Training function in Custom GAN

Output

```

Training for Parkinson class...
[Epoch 0/20] [Batch 0/204] [D loss: 0.7895] [G loss: 0.7247]
[Epoch 0/20] [Batch 100/204] [D loss: 0.1698] [G loss: 0.0854]
[Epoch 0/20] [Batch 200/204] [D loss: 0.1885] [G loss: 0.0471]
[Epoch 1/20] [Batch 0/204] [D loss: 0.2705] [G loss: 0.0528]
[Epoch 1/20] [Batch 100/204] [D loss: 0.5476] [G loss: 0.0570]
[Epoch 1/20] [Batch 200/204] [D loss: 0.2236] [G loss: 0.0369]

```

Figure 7: Custom GAN Output

Part 5: Checking the total count of original and generated images

The 'SRGANTRIALWORKSEED' folder consists of Custom GAN saved images and 'Dataset' folder consists of original images.

```

import os

def count_images_in_directory(directory):
    image_extensions = ['.jpg', '.jpeg', '.png', '.bmp', '.gif', '.tiff']
    count = 0
    for root, _, files in os.walk(directory):
        for file in files:
            if any(file.lower().endswith(ext) for ext in image_extensions):
                count += 1
    return count

#base directories
base_real_dir = '/content/drive/MyDrive/Dataset'
base_generated_dir = '/content/drive/MyDrive/SRGANTRIALWORKSEED'

subdirectories = ['Parkinson', 'Healthy']

#counting images in each subdirectory
total_images = 0
for base_dir in [base_real_dir, base_generated_dir]:
    for subdirectory in subdirectories:
        dir_path = os.path.join(base_dir, subdirectory)
        count = count_images_in_directory(dir_path)
        total_images += count
        print(f"Number of images in {dir_path}: {count}")

print(f"Total images in both directories: {total_images}")

```

```

Number of images in /content/drive/MyDrive/Dataset/Parkinson: 1632
Number of images in /content/drive/MyDrive/Dataset/Healthy: 1632
Number of images in /content/drive/MyDrive/SRGANTRIALWORKSEED/Parkinson: 16320
Number of images in /content/drive/MyDrive/SRGANTRIALWORKSEED/Healthy: 16320
Total images in both directories: 35904

```

Figure 8: Count of Custom GAN saved images and original images

Part 6: Calculating SSIM

The calculate_batch_ssim function calculates the SSIM score for each batch of real and generated images by calling the calculate_ssim function to compute the SSIM score for individual pairs of images within each batch and then averages these scores to provide an overall SSIM score.

```

def calculate_ssim(images1, images2, win_size=3, data_range=1.0):
    ssim_sum = 0.0
    num_images = min(len(images1), len(images2))

    for i in range(num_images):
        img1 = images1[i].permute(1, 2, 0).numpy() #converting to numpy and reshaping
        img2 = images2[i].permute(1, 2, 0).numpy()
        #print(f"Comparing images of shape: {img1.shape} and {img2.shape}")
        ssim_sum += compare_ssim(img1, img2, multichannel=True, win_size=win_size, channel_axis=-1, data_range=data_range)

    ssim_avg = ssim_sum / num_images
    return ssim_avg

def calculate_batch_ssim(loader1, loader2, win_size=3, data_range=1.0):
    total_ssim = 0.0
    count = 0
    for batch1, batch2 in zip(loader1, loader2):
        ssim = calculate_ssim(batch1, batch2, win_size=win_size, data_range=data_range)
        total_ssim += ssim * len(batch1)
        count += len(batch1)
    return total_ssim / count

base_real_dir = '/content/drive/MyDrive/Dataset'
base_generated_dir = '/content/drive/MyDrive/SRGANTRIALWORKSEED'

batch_size = 8

#real and generated images resized
real_parkinson_dataset = CustomImageFolder(base_real_dir, 'Parkinson', transform)
generated_parkinson_dataset = CustomImageFolder(base_generated_dir, 'Parkinson', transform)
real_healthy_dataset = CustomImageFolder(base_real_dir, 'Healthy', transform)
generated_healthy_dataset = CustomImageFolder(base_generated_dir, 'Healthy', transform)

real_parkinson_loader = DataLoader(real_parkinson_dataset, batch_size=batch_size, shuffle=False, num_workers=0)
generated_parkinson_loader = DataLoader(generated_parkinson_dataset, batch_size=batch_size, shuffle=False, num_workers=0)
real_healthy_loader = DataLoader(real_healthy_dataset, batch_size=batch_size, shuffle=False, num_workers=0)
generated_healthy_loader = DataLoader(generated_healthy_dataset, batch_size=batch_size, shuffle=False, num_workers=0)

```

Figure 9: SSIM code

```

#SSIM for batches
ssim_parkinson = calculate_batch_ssim(real_parkinson_loader, generated_parkinson_loader, win_size=3, data_range=1.0)
ssim_healthy = calculate_batch_ssim(real_healthy_loader, generated_healthy_loader, win_size=3, data_range=1.0)

print(f"SSIM for Parkinson class: {ssim_parkinson:.6f}")
print(f"SSIM for Healthy class: {ssim_healthy:.6f}")

Found directory: /content/drive/MyDrive/Dataset/Parkinson
Found directory: /content/drive/MyDrive/SRGANTRIALWORKSEED/Parkinson
Found directory: /content/drive/MyDrive/Dataset/Healthy
Found directory: /content/drive/MyDrive/SRGANTRIALWORKSEED/Healthy
SSIM for Parkinson class: 0.606148
SSIM for Healthy class: 0.637629

```

Figure 10: SSIM output

Part 7: EDA of generated images and original images

```

#loading images from a specific directory
def load_images_from_subdirectory(base_dir, subdirectory):
    directory = os.path.join(base_dir, subdirectory)
    image_paths = glob(os.path.join(directory, '*.png'))
    images = [cv2.imread(img_path) for img_path in image_paths]
    return images, image_paths

#original images
original_parkinson_images, original_parkinson_paths = load_images_from_subdirectory(base_real_dir, 'Parkinson')
original_healthy_images, original_healthy_paths = load_images_from_subdirectory(base_real_dir, 'Healthy')

#GAN images
gan_parkinson_images, gan_parkinson_paths = load_images_from_subdirectory(base_generated_dir, 'Parkinson')
gan_healthy_images, gan_healthy_paths = load_images_from_subdirectory(base_generated_dir, 'Healthy')

#display images
def display_images(images, title):
    fig, axes = plt.subplots(1, 5, figsize=(15, 5))
    for img, ax in zip(images, axes):
        ax.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
        ax.axis('off')
    plt.suptitle(title)
    plt.show()

display_images(random.sample(original_parkinson_images, 5), 'Original Parkinson Images Samples')
display_images(random.sample(original_healthy_images, 5), 'Original Healthy Images Samples')
display_images(random.sample(gan_parkinson_images, 5), 'GAN Parkinson Images Samples')
display_images(random.sample(gan_healthy_images, 5), 'GAN Healthy Images Samples')

```

Figure 11: Code for EDA of Custom GAN and original images

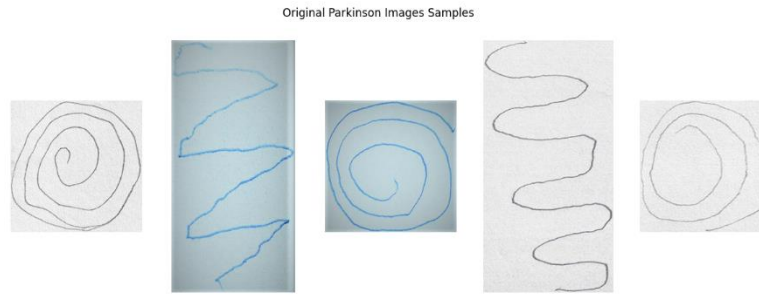


Figure 12: EDA for Custom GAN and original images

3.5 Creating GAN combined data

Here, all images from original dataset are collected and then remaining images are taken from ‘SRGANTRIALWORKSEED’ folder which has the Custom GAN saved images, the total images are restricted to 7000 images due to high computational requirements

```
base_real_dir = '/content/drive/MyDrive/Dataset'
base_generated_dir = '/content/drive/MyDrive/SRGANTRIALWORKSEED'
subdirectories = ['Parkinson', 'Healthy']

#destination directory
dest_dir = '/content/drive/MyDrive/SelectedImageFINAL'

#collecting images separately for each category
category_images = {'Parkinson': [], 'Healthy': []}
for subdirectory in subdirectories:
    #collecting images from real directory
    real_dir_path = os.path.join(base_real_dir, subdirectory)
    real_images = get_images_from_directory(real_dir_path)
    category_images[subdirectory].extend(real_images)

    #collecting images from generated directory
    generated_dir_path = os.path.join(base_generated_dir, subdirectory)
    generated_images = get_images_from_directory(generated_dir_path)
    category_images[subdirectory].extend(generated_images)

#ensuring enough images in each category
total_images_needed = 7000
images_per_category = total_images_needed // 2
for category, images in category_images.items():
    if len(images) < images_per_category:
        raise ValueError(f"Not enough images in the {category} category to select {images_per_category}. Only found {len(images)} images.")

selected_images = {'Parkinson': [], 'Healthy': []}

for category in subdirectories:
    if category == 'Parkinson':
        #getting all real Parkinson images first
        real_images = [img for img in category_images[category] if base_real_dir in img]

        #selecting all real Parkinson images
        selected_images[category] = real_images[:images_per_category]

    #checking remaining space for Parkinson images
    remaining_space = images_per_category - len(selected_images[category])
```

Figure 13: Creating GAN combined data (part 1)

```

if remaining_space > 0:
    #getting all generated Parkinson images
    generated_images = [img for img in category_images[category] if base_generated_dir in img]

    #appending generated Parkinson images up to the remaining space
    selected_images[category].extend(generated_images[:remaining_space])

    print(f"Used {len(generated_images[:remaining_space])} GAN images for the Parkinson category.")

elif category == 'Healthy':
    #getting all real Healthy images first
    real_images = [img for img in category_images[category] if base_real_dir in img]

    #selecting all real Healthy images
    selected_images[category] = real_images[:images_per_category]

    #checking remaining space for Healthy images
    remaining_space = images_per_category - len(selected_images[category])

    if remaining_space > 0:
        #getting all generated Healthy images
        generated_images = [img for img in category_images[category] if base_generated_dir in img]

        #appending generated Healthy images up to the remaining space
        selected_images[category].extend(generated_images[:remaining_space])

        print(f"Used {len(generated_images[:remaining_space])} GAN images for the Healthy category.")

copy_images(selected_images, dest_dir)

print(f"Copied {total_images_needed} images to {dest_dir} under their respective folders.")

```

Used 1868 GAN images for the Parkinson category.
 Used 1868 GAN images for the Healthy category.
 Copied 7000 images to /content/drive/MyDrive/SelectedImageFINAL under their respective folders.

Figure 14: Creating GAN combined data (part 2)

The 'SelectedImageFINAL' folder contains the GAN combined data and is used further for the research, for the modelling part.

3.6 Modelling

3.6.1 ResNet50 + KNN classifier

The images are preprocessed and then passed to ResNet50 for feature extraction and then passed to KNN classifier for classification. The images are normalized and enhanced by increasing the brightness

```

parent_dataset_folder = "/content/drive/MyDrive/SelectedImageFINAL"
subdirectories = ['Parkinson', 'Healthy']

#resizing
target_size = (128, 128)

#preprocessing images
def preprocess_image(image_path):
    image = Image.open(image_path)

    if image.mode == 'RGBA':
        image = image.convert('RGB')

    image = image.resize(target_size)

    #contrast
    enhancer = ImageEnhance.Brightness(image)
    image = enhancer.enhance(1.2)

    #normalizing the image array
    image_array = np.array(image) / 255.0

    return image_array

preprocessed_images = []
labels = []

#iterating through each class folder
for class_folder in subdirectories:
    class_folder_path = os.path.join(parent_dataset_folder, class_folder)
    if os.path.isdir(class_folder_path):
        #each image file in the class folder
        for filename in os.listdir(class_folder_path):
            if filename.endswith(".png"): #images in PNG format
                image_path = os.path.join(class_folder_path, filename)

```

Figure 15: Preprocessing

After preprocessing, the images are splitted into train and test in a ratio of 80:20.

```
#preprocess image
preprocessed_image = preprocess_image(image_path)

preprocessed_images.append(preprocessed_image)

labels.append(class_folder)

#converting to numpy arrays
preprocessed_images = np.array(preprocessed_images)
labels = np.array(labels)

#encoding the labels
label_encoder = LabelEncoder()
encoded_labels = label_encoder.fit_transform(labels)

X_train, X_test, y_train, y_test = train_test_split(preprocessed_images, encoded_labels, test_size=0.2, random_state=42)

#free memory
del preprocessed_images, labels
gc.collect()

#ResNet50 model for feature extraction
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(128, 128, 3))
feature_extractor = tf.keras.Model(inputs=base_model.input, outputs=base_model.output)

#extracting features from the training and test images
def extract_features(images, batch_size=32):
    num_samples = images.shape[0]
    features_list = []
    for i in range(0, num_samples, batch_size):
        batch_images = images[i:i + batch_size]
        features = feature_extractor.predict(batch_images, batch_size=batch_size)
        features_flattened = features.reshape(features.shape[0], -1)
        features_list.append(features_flattened)
    features_full = np.vstack(features_list)
    return features_full

X_train_features = extract_features(X_train)
```

Figure 16: ResNet50 for feature extraction

Grid search is set up for KNN classifier for parameters like, n_neighbors, weights and metrics. The best parameter found is used for prediction of test set.

```
X_train_features = extract_features(X_train)
X_test_features = extract_features(X_test)

#free memory
del X_train, X_test
gc.collect()

#GridSearchCV
param_grid = {
    'n_neighbors': [3, 5],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan']
}

knn = KNeighborsClassifier()

grid_search = GridSearchCV(knn, param_grid, cv=5, n_jobs=-1, verbose=1)

grid_search.fit(X_train_features, y_train)

#best parameters found by GridSearchCV
print("Best parameters found: ", grid_search.best_params_)

#training the kNN classifier with the best parameters
best_knn = grid_search.best_estimator_

#predicting labels for the test set using the best kNN classifier
y_test_pred_labels = best_knn.predict(X_test_features)

test_accuracy_manual = accuracy_score(y_test, y_test_pred_labels)
print("Manual accuracy on test data:", test_accuracy_manual)

conf_matrixrk = confusion_matrix(y_test, y_test_pred_labels)
print("Confusion Matrix:")
print(conf_matrixrk)
```

Figure 17: ResNet50 with KNN classifier

3.6.2 InceptionV3 +KNN

The images are preprocessed and then passed to InceptionV3 for feature extraction and then passed to KNN classifier for classification

```
#InceptionV3 model for feature extraction
base_model = InceptionV3(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
feature_extractor = tf.keras.Model(inputs=base_model.input, outputs=base_model.output)

#extracting features from the training and test images
def extract_features(images):
    features = feature_extractor.predict(images)
    #flatten the features
    features_flattened = features.reshape(features.shape[0], -1)
    return features_flattened

X_train_features = extract_features(X_train)
X_test_features = extract_features(X_test)

#parameter grid for hyperparameter tuning
param_grid = {
    'n_neighbors': [3, 5],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan']
}

#kNN classifier
knn = KNeighborsClassifier()

grid_search = GridSearchCV(knn, param_grid, cv=5, scoring='accuracy', n_jobs=-1)

grid_search.fit(X_train_features, y_train)

#best hyperparameters
best_params = grid_search.best_params_
print("Best hyperparameters:", best_params)

#training the kNN classifier with the best hyperparameters
knn_best = KNeighborsClassifier(**best_params)
knn_best.fit(X_train_features, y_train)

#predicting labels for the test set using the kNN classifier
y_test_pred_labels = knn_best.predict(X_test_features)
```

Figure 18: InceptionV3 with KNN classifier

3.6.3 Inceptionv3

InceptionV3 is used and additional layers² are added and top 20 layers are unfrozen for fine tuning. The model is trained for 10 epochs using Adam optimizer and loss function.

² <https://medium.com/@armielynobinguar/simple-implementation-of-inceptionv3-for-image-classification-using-tensorflow-and-keras-6557feb9bf53>

```

#tensorflow datasets
batch_size = 16
train_ds = tf.data.Dataset.from_tensor_slices((X_train, y_train)).batch(batch_size).prefetch(tf.data.AUTOTUNE)
test_ds = tf.data.Dataset.from_tensor_slices((X_test, y_test)).batch(batch_size).prefetch(tf.data.AUTOTUNE)

#InceptionV3 model
base_model = InceptionV3(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
base_model.trainable = True

#unfreezing top layers of the model for finetuning
for layer in base_model.layers[-20:]:
    layer.trainable = True

model = models.Sequential([
    base_model,
    layers.GlobalAveragePooling2D(),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(len(subdirectories), activation='softmax')
])

model.compile(optimizer=optimizers.Adam(learning_rate=1e-4),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(train_ds, epochs=10)

test_loss, test_accuracyinc = model.evaluate(test_ds)
print(f"Accuracy on test data: {test_accuracyinc}")

y_test_pred = model.predict(test_ds)
y_test_pred_labels = np.argmax(y_test_pred, axis=1)

print("Manual accuracy on test data:", accuracy_score(y_test, y_test_pred_labels))
print("Confusion Matrix:")
conf_matrixinc = confusion_matrix(y_test, y_test_pred_labels)
print(conf_matrixinc)

```

Figure 19: InceptionV3

3.6.4 ResNet50

ResNet50 is used as a base model and few layers are added on top of the model.³

```

#tensorflow datasets
batch_size = 16
train_ds = tf.data.Dataset.from_tensor_slices((X_train, y_train)).batch(batch_size).prefetch(tf.data.AUTOTUNE)
test_ds = tf.data.Dataset.from_tensor_slices((X_test, y_test)).batch(batch_size).prefetch(tf.data.AUTOTUNE)

base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
base_model.trainable = True

#unfreezing the top layers of the model for fine-tuning
for layer in base_model.layers[-20:]:
    layer.trainable = True

model = models.Sequential([
    base_model,
    layers.GlobalAveragePooling2D(),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(len(subdirectories), activation='softmax')
])

model.compile(optimizer=optimizers.Adam(learning_rate=1e-4),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(train_ds, epochs=10)

test_loss, test_accuracyres = model.evaluate(test_ds)
print(f"Accuracy on test data: {test_accuracyres}")

y_test_pred = model.predict(test_ds)

```

Figure 20: ResNet50

³ <https://medium.com/@nitishkundu1993/exploring-resnet50-an-in-depth-look-at-the-model-architecture-and-code-implementation-d8d8fa67e46f>

3.7 EDA (ACCURACY)

```
import matplotlib.pyplot as plt

metrics = ['ResNetKNN', 'InceptionKNN', 'InceptionV3', 'Resnet50']
values = [test_accuracy_manual, test_accuracyik, test_accuracyinc, test_accuracyres]

colors = ['#1f77b4', '#aec7e8', '#6baed6', '#08306b']

plt.figure(figsize=(10, 6))
bars = plt.bar(metrics, values, color=colors)

plt.xlabel('Accuracy Metrics')
plt.ylabel('Accuracy Value')
plt.title('Comparison of Different Accuracy Metrics')
plt.ylim([0, 1.1])

for bar in bars:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval + 0.02, f'{yval:.2f}',
             ha='center', va='bottom', fontsize=10)

plt.show()
```

Figure 21: EDA of accuracy metric for all 4 models

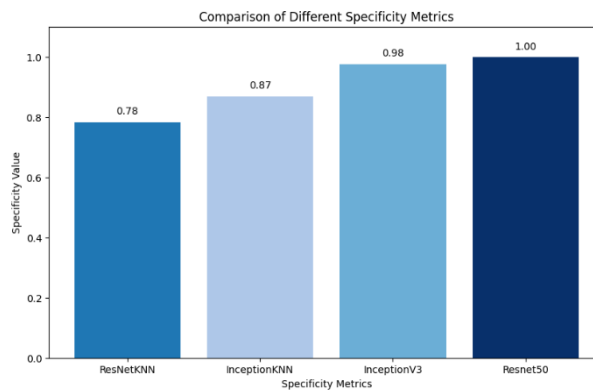


Figure 22: Accuracy graph for all 4 models

4 Replicating the base paper

(Kumar and Bansal, 2023) was chosen as the base paper, because the dataset used in this paper is taken as the original dataset in the current research.

```

import numpy as np
import tensorflow as tf
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.layers import Dense, Dropout, AveragePooling2D, Flatten
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
import os
import random

np.random.seed(42)
tf.random.set_seed(42)
random.seed(42)

parent_dataset_folder = "/content/drive/MyDrive/Dataset"

target_size = (224, 224)

def preprocess_image(image_path):
    image = tf.keras.preprocessing.image.load_img(image_path, target_size=target_size)
    image_array = tf.keras.preprocessing.image.img_to_array(image)
    #normalizing pixel values
    image_array = image_array / 255.0
    return image_array

def load_and_preprocess_images(directory):
    images = []
    labels = []
    for class_folder in os.listdir(directory):
        class_folder_path = os.path.join(directory, class_folder)
        if os.path.isdir(class_folder_path):
            for filename in os.listdir(class_folder_path):
                if filename.endswith(".jpg") or filename.endswith(".png"):
                    image_path = os.path.join(class_folder_path, filename)
                    image = preprocess_image(image_path)
                    images.append(image)
                    labels.append(class_folder)
    return np.array(images), np.array(labels)

preprocessed_images, labels = load_and_preprocess_images(parent_dataset_folder)

preprocessed_images, labels = load_and_preprocess_images(parent_dataset_folder)

#encode labels
label_binarizer = LabelBinarizer()
labels_encoded = label_binarizer.fit_transform(labels)

X_train, X_test, y_train_encoded, y_test_encoded = train_test_split(preprocessed_images, labels_encoded, test_size=0.2, random_state=42)

base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

x = base_model.output
x = AveragePooling2D(pool_size=(7, 7))(x)
x = Flatten()(x)
x = Dense(128, activation='relu')(x)
x = Dropout(0.5)(x)
predictions = Dense(2, activation='softmax', name='dense_1_of_2')(x) #output layer with 2 neurons

model = Model(inputs=base_model.input, outputs=predictions)

#freezing the layers of the base model
for layer in base_model.layers:
    layer.trainable = False

model.compile(optimizer=Adam(lr=0.001), loss='sparse_categorical_crossentropy', metrics=['accuracy'])

model.summary()

history = model.fit(X_train, y_train_encoded, batch_size=44, epochs=74, validation_data=(X_test, y_test_encoded))

loss, accuracy = model.evaluate(X_test, y_test_encoded)
print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy:.4f}")

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_224_no_top.h5
9406464/9406464 [=====] - 0s 0us/step
WARNING:absl:'lr' is deprecated in Keras optimizer, please use 'learning_rate' or use the legacy optimizer, e.g., tf.keras.optimizers.legacy.Adam.
Model: "model"

```

Layer (type)	Output Shape	Param #	Connected to
=====			

Figure 23: Base paper

References

Kumar, B. A. and Bansal, M. (2023) ‘A transfer learning approach with MobileNetV2 for Parkinson’s disease detection using hand-drawings’, in *Proceedings of the 14th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. Delhi, India, 6-8 July 2023, pp.1-8. doi: 10.1109/ICCCNT56998.2023.10307641