Overview:

This project focuses on analyzing rat neural data obtained from video recordings. The goal is to understand neural dynamics during specific activities.

Objective:

- **Data Loading:** Load and preprocess rat neural data from various sources.
- **Exploratory Data Analysis (EDA):** Understand the structure and patterns in neural activity.
- **Conditional GAN Implementation:** Implement Conditional Generative Adversarial Networks (cGANs) to generate visual representations from neural data.
- **Visualization and Analysis:** Visualize and interpret the generated data to gain insights into neural dynamics.

Steps:

1. **Data Preparation:** Loading multiple data sources (mat files, video frames).
2. **Exploratory Data Analysis:** Understanding neural activity patterns and relationships.
3. **cGAN Implementation:** Building and training cGAN models on rat neural data.
4. **Data Generation:** Generating visual representations from neural data using cGANs.

Libraries Used:

- `tensorflow`
- `time`
- `os`
- `random`
- `keras.optimizers.Adam`
- `tensorflow.keras.layers`
- `tensorflow.keras.models.Model`
- `numpy`
- `numpy.expand_dims`
- `numpy.zeros`
- `numpy.ones`
- `numpy.randn`
- `numpy.randint`
- `PIL.Image`
- `scipy.io.loadmat`
- `scipy.signal.filtfilt`
- `scipy.signal.windows`
- `warnings`
- `matplotlib.pyplot`
- `keras.layers.Dropout`
- `keras.layers.Concatenate`

- `keras.layers.UpSampling2D`
- `keras.layers.Reshape`
- `keras.layers.Activation`
- `keras.layers.Conv2D`
- `keras.layers.BatchNormalization`
- `keras.layers.LeakyReLU`
- `keras.layers.Input`
- `keras.layers.Flatten`
- `keras.layers.multiply`
- `keras.layers.Dense`
- `keras.layers.Embedding`
- `keras.models.Sequential`

Import Required Libraries

```
import tensorflow as tf
import time
import os
import random
from keras.optimizers import Adam
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential, Model
import numpy as np
from numpy import expand_dims
from numpy import zeros
from numpy import ones
from numpy.random import randn
from numpy.random import randint
from keras.optimizers import Adam
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from PIL import Image
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Dropout
from keras.layers import Embedding
from keras.layers import Concatenate
from scipy.signal import filtfilt, windows
# supress warnings
import warnings
warnings.filterwarnings('ignore')
from matplotlib import pyplot as plt
from scipy.io import loadmat
from keras.layers import Dropout, Concatenate
```

```
from keras.layers import UpSampling2D, Reshape, Activation, Conv2D,
BatchNormalization, LeakyReLU, Input, Flatten, multiply
from keras.layers import Dense, Embedding
from keras.models import Sequential, Model
```

Import Data

```
# Define the file paths for the datasets
file_path_1 = 'C:\\Users\\Chandra\\Desktop\\Model\\data\\
day10_adv_ms.mat'
file_path_2 = 'C:\\Users\\Chandra\\Desktop\\Model\\data\\2011-04-
23_day10.mat'

# Define directories for data access
data_directory_1 = "C:\\Users\\Chandra\\Desktop\\Model\\data\\"
data_directory_2 = "C:\\Users\\Chandra\\Desktop\\Model\\data\\
reach_video_data\\"

# Check if both files exist before proceeding
if os.path.exists(file_path_1) and os.path.exists(file_path_2):
    # Load the datasets if files exist
    day10_adv_ms = loadmat(file_path_1)
    day10 = loadmat(file_path_2)

    # Load multiple files using a loop
    num_files = 65  # Number of files to load (0 to 65)
    loaded_files = []
    for i in range(num_files):
        file_name = f'day10_frame_reach_vid_nofrz_reach{i}.npy'
        file_path = os.path.join(data_directory_2, file_name)
        if os.path.exists(file_path):
            loaded_file = np.load(file_path)
            loaded_files.append(loaded_file)
        else:
            print(f"File '{file_name}' does not exist.")

    # Convert loaded files into a numpy array
    loaded_array = np.array(loaded_files)

    # Additional data loading
    frame_reach_unit = np.load(data_directory_1 +
'day10_frame_reach_unit.npy')
    tr_adv_ms = day10_adv_ms['tr_adv_ms']
    s = day10['s']
else:
    # Handle the case where either or both files do not exist
    print("One or both files do not exist:", file_path_1, file_path_2)
```

Exploratory Data Analysis

```
loaded_array.shape
all_frames = np.transpose(loaded_array, (0, 3, 1, 2))
all_frames.shape
```

```
(65, 820, 299, 299)
```

The `all_frames` shape represents the loaded rat neural video data, comprising:

- **65 reaches:** These are distinct instances or events observed in the neural data.
- **820 frames:** Each reach contains 820 fram.h.
- **299 x 299:** The frames have a spatial resolution of 299 x 299 pixels (height x widtdata.

```
# Create an empty array to store resized images with shape (65, 820,
80, 80)
resized_images = np.zeros((65, 820, 80, 80), dtype=np.uint8)

# Loop through each day and each image in a day for resizing
for i in range(all_frames.shape[0]):  # Loop through each reach
    for j in range(all_frames.shape[1]):  # Loop through each image in
a reach
        # Resize each image to 80x80 while maintaining aspect ratio
directly using NumPy
        resized_img = np.array(Image.fromarray(all_frames[i,
j]).resize((80, 80), Image.LANCZOS))
        # Store the resized image in the new array
        resized_images[i, j] = resized_img

# The variable resized_images now contains the resized images with
shape (65, 820, 80, 80).
# Here, images originally sized 299x299 pixels are resized to 80x80
pixels to facilitate efficient processing and analysis.

#copying the data to all_frames
all_frames =resized_images

# Create an empty list to store extracted frames representing specific
instances relative to each reach
all_image = []

# Loop through each reach in the dataset
for i in range(all_frames.shape[0]):
    # Extract frames representing different instances relative to each
reach:
    # - Frame at index 410: Before the reach
    # - Frame at index 510: During the reach
    # - Frame at index 610: After the reach
    sub_frames_before = all_frames[i, 410, :, :]  # Extract frame
before reach (at index 410)
    sub_frames_during = all_frames[i, 510, :, :]  # Extract frame
```

```
during reach (at index 510)
    sub_frames_after = all_frames[i, 610, :, :]   # Extract frame
after reach (at index 610)

    # Append the extracted frames to the list 'all_image'
    all_image.append(sub_frames_before)  # Append frame before reach
to 'all_image'
    all_image.append(sub_frames_during)  # Append frame during reach
to 'all_image'
    all_image.append(sub_frames_after)   # Append frame after reach to
'all_image'

# The 'all_image' list now contains frames extracted at specific
instances relative to each reach:
# - Frames at index 410 represent frame before the reach.
# - Frames at index 510 represent frame during the reach.
# - Frames at index 610 represent frame after the reach.

# Convert the list 'all_image' containing extracted frames into a
NumPy array
all_frames = np.array(all_image)

# Tile the 'all_frames' array to duplicate its content for further
processing
# This creates a new array by repeating 'all_frames' twice along the
first dimension
# The resulting shape becomes (2 * number of reaches, number of
instances per reach, frame dimensions)
all_frames = np.tile(all_frames, (2, 1, 1))

# List containing specific labels representing instances relative to
each reach
all_sublabels = [410, 510, 610]

# Create a list 'newlabels' to store labels for instances relative to
each of the 65 reaches
newlabels = []

# Iterate through the range of reaches (65 in this case) and append
the 'all_sublabels' list to 'newlabels'
# Each entry in 'newlabels' will contain the same set of sublabels
[410, 510, 610]
for i in range(65):
    newlabels.append(all_sublabels)

# Convert the list 'newlabels' containing sublabels for each reach
into a NumPy array
# The resulting 'all_labels' array holds the labels for instances
relative to each reach
```

```python
# It will have a shape (65, number of instances per reach)
all_labels = np.array(newlabels)

# Repeat the 'all_sublabels' array for all 65 reaches
# This creates a new array ('repeated_array') by repeating the
'all_sublabels' array 65 times
repeated_array = np.tile(all_sublabels, 65)
# Repeat the 'repeated_array' content twice along the first dimension
# This creates a new array ('all_labels') by duplicating the
'repeated_array' content twice
all_labels = np.tile(repeated_array, 2)
```

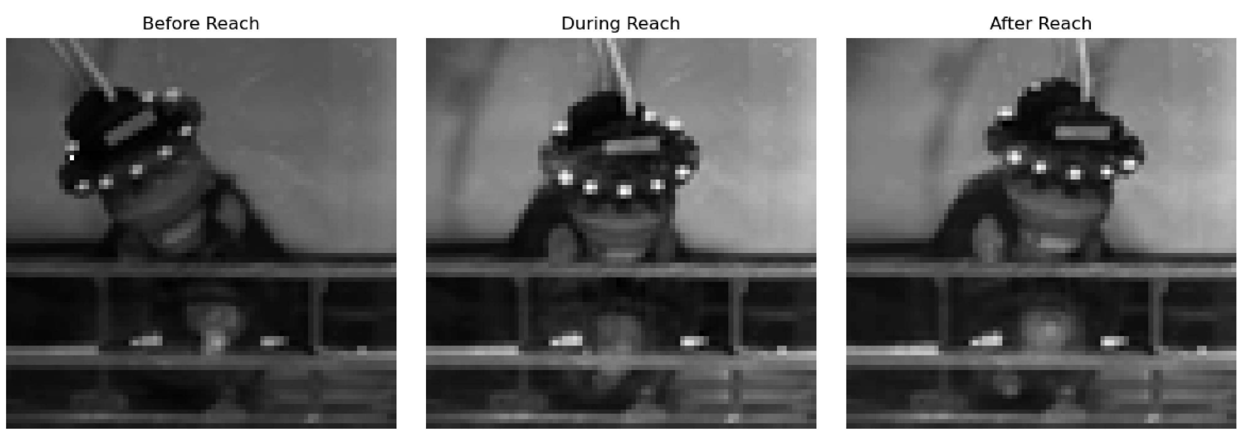
Data Visualization

```python
# Assuming all_frames contains images corresponding to before, during,
and after reach
num_images = 3  # Number of images to display
timing_labels = ['Before Reach', 'During Reach', 'After Reach']  #
Labels for timing context

# Create a figure and subplots to display images side by side
fig, axes = plt.subplots(1, num_images, figsize=(12, 4))  # Adjust
figsize as needed

# Loop through the images and display them in separate subplots with
timing labels
for i in range(num_images):
    axes[i].imshow(all_frames[i], cmap='gray',
interpolation='nearest')
    axes[i].set_title(timing_labels[i])  # Set timing label as subplot
title
    axes[i].axis('off')  # Turn off axis labels (optional)

plt.tight_layout()  # Adjust layout for better visualization
plt.show()
```



```
all_labels.shape
```

```
(390,)
```

```
all_frames.shape
```

```
(390, 80, 80)
```

- 'all_labels' has a shape of (390,) indicating 390 instances where each label corresponds to an image frame.
- 'all_frames' has a shape of (390, 80, 80), where:
  - The first dimension aligns with 'all_labels', ensuring a one-to-one relationship between labels and frames.
  - Each frame has dimensions 80x80 pixels, forming the visual data associated with each label.

This aligned structure allows convenient pairing of labels and frames for analyses or models where timing context (before reach, during reach, after reach) represented by the labels can be associated with the respective images. mages.

```
# Define the discriminator model architecture capable of classifying
images
def define_discriminator(in_shape=(80, 80, 1), n_classes=3):
    """
    Discriminator model:
    - Inputs: Image and corresponding label.
    - Outputs: Binary classification (real or fake) of the input
image-label pair.

    Arguments:
    - in_shape: Tuple specifying input image shape (height, width,
channels).
    - n_classes: Number of classes/categories for label embedding.

    Architecture:
    - Embeds categorical input label into image-sized representation.
    - Concatenates label representation as an additional channel to
the input image.
    - Utilizes Convolutional Neural Network (CNN) layers for feature
extraction.
    - Employs Leaky ReLU activation for feature map enhancement.
    - Uses Dropout to prevent overfitting.
    - Produces a binary classification output (real/fake) using a
sigmoid activation.

    Parameters:
    - in_label: Input layer for the label.
    - in_image: Input layer for the image.
    - li: Embedding layer to represent the input label categorically.
```

```
    - merge: Concatenation of image and label embeddings as an input
for the CNN layers.
    - fe: Feature maps extracted by CNN layers.
    - out_layer: Output layer predicting the authenticity of the
image-label pair.

    Compilation:
    - Utilizes binary cross-entropy loss and Adam optimizer for
training.
    - Learning rate set to 0.0002 and beta_1 to 0.5 for Adam
optimizer.

    Returns:
    - Discriminator model compiled and ready for training.
    """

    # Define label input
    in_label = Input(shape=(1,))
    # Embed categorical input label
    li = Embedding(n_classes, 50)(in_label)
    # Scale up to image dimensions with linear activation
    n_nodes = in_shape[0] * in_shape[1]
    li = Dense(n_nodes)(li)
    # Reshape to an additional channel
    li = Reshape((in_shape[0], in_shape[1], 1))(li)
    # Define image input
    in_image = Input(shape=in_shape)
    # Concatenate label as a channel to the image
    merge = Concatenate()([in_image, li])
    # Downsample using convolutional layers with LeakyReLU activation
    fe = Conv2D(128, (3, 3), strides=(2, 2), padding='same')(merge)
    fe = LeakyReLU(alpha=0.2)(fe)
    fe = Conv2D(128, (3, 3), strides=(2, 2), padding='same')(fe)
    fe = LeakyReLU(alpha=0.2)(fe)
    fe = Conv2D(128, (3, 3), strides=(2, 2), padding='same')(fe)
    fe = LeakyReLU(alpha=0.2)(fe)
    fe = Conv2D(128, (3, 3), strides=(2, 2), padding='same')(fe)
    fe = LeakyReLU(alpha=0.2)(fe)
    # Flatten feature maps
    fe = Flatten()(fe)
    # Apply dropout to prevent overfitting
    fe = Dropout(0.4)(fe)
    # Output layer for binary classification (real/fake)
    out_layer = Dense(1, activation='sigmoid')(fe)
    # Define the discriminator model
    model = Model([in_image, in_label], out_layer)
    # Compile the model with binary cross-entropy loss and Adam
optimizer
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt,
```

```
metrics=['accuracy'])
    return model

# Define the generator model responsible for generating images from
random noise and labels
def define_generator(latent_dim, n_classes=3):
    """
    Generator model:
    - Inputs: Random noise and corresponding label.
    - Outputs: Synthetic image corresponding to the input noise-label
pair.

    Arguments:
    - latent_dim: Dimension of the input random noise vector.
    - n_classes: Number of classes/categories for label embedding.

    Architecture:
    - Embeds categorical input label into an image-sized
representation.
    - Combines random noise with dense layers to form an initial image
foundation.
    - Uses Convolutional Neural Network (CNN) layers for upsampling
and image generation.
    - Employs Leaky ReLU activation for feature map enhancement.
    - Produces a synthetic image output using the tanh activation
function.

    Parameters:
    - in_label: Input layer for the label.
    - in_lat: Input layer for the random noise vector.
    - li: Embedding layer to represent the input label categorically.
    - gen: Initial image foundation derived from random noise.
    - merge: Concatenation of label representation and image
foundation as an input for CNN layers.

    Upsampling:
    - Utilizes Conv2DTranspose layers to upsample the image
progressively to the desired dimensions (80x80).

    Output:
    - Synthetic image output of the desired dimensions and
characteristics.

    Returns:
    - Generator model ready for generating synthetic images based on
noise and labels.
    """

    # Define label input
    in_label = Input(shape=(1,))
```

```
    # Embed categorical input label
    li = Embedding(n_classes, 50)(in_label)
    # Linear multiplication for reshaping
    n_nodes = 5 * 5
    li = Dense(n_nodes)(li)
    # Reshape label representation to an additional channel
    li = Reshape((5, 5, 1))(li)
    # Define input layer for random noise
    in_lat = Input(shape=(latent_dim,))
    # Foundation for a 5x5 image from random noise
    n_nodes = 128 * 5 * 5
    gen = Dense(n_nodes)(in_lat)
    gen = LeakyReLU(alpha=0.2)(gen)
    gen = Reshape((5, 5, 128))(gen)
    # Merge label representation with the generated image foundation
    merge = Concatenate()([gen, li])
    # Upsample progressively to reach 80x80 image dimensions
    gen = Conv2DTranspose(128, (4, 4), strides=(2, 2), padding='same')
(merge)
    gen = LeakyReLU(alpha=0.2)(gen)
    gen = Conv2DTranspose(128, (4, 4), strides=(2, 2), padding='same')
(gen)
    gen = LeakyReLU(alpha=0.2)(gen)
    gen = Conv2DTranspose(128, (4, 4), strides=(2, 2), padding='same')
(gen)
    gen = LeakyReLU(alpha=0.2)(gen)
    gen = Conv2DTranspose(128, (4, 4), strides=(2, 2), padding='same')
(gen)
    gen = LeakyReLU(alpha=0.2)(gen)
    # Output layer for generating synthetic images
    out_layer = Conv2D(1, (5, 5), activation='tanh', padding='same')
(gen)
    # Define the generator model
    model = Model([in_lat, in_label], out_layer)
    return model

# Define the combined Generator and Discriminator model used for
training the Generator
def define_gan(g_model, d_model):
    """
    GAN (Generative Adversarial Network) model:
    - Combines the Generator and Discriminator models for updating the
Generator.

    Arguments:
    - g_model: Generator model used for generating synthetic images.
    - d_model: Discriminator model used for classifying real and
synthetic images.

    Architecture:
```

```
    - Freezes the weights of the Discriminator to avoid updating
during GAN training.
    - Utilizes Generator inputs (noise and label) to generate
synthetic images.
    - Connects the Generator output and label input to the
Discriminator for classification.
    - Generates a GAN output representing the Discriminator's
classification of the generated images.

    Parameters:
    - gen_noise: Input layer for random noise in the Generator.
    - gen_label: Input layer for labels in the Generator.
    - gen_output: Output of the Generator model, representing
synthetic images.
    - gan_output: Output representing the Discriminator's
classification of synthetic images.

    Compilation:
    - Compiles the GAN model using binary cross-entropy loss and Adam
optimizer.

    Returns:
    - GAN model ready for training the Generator by adversarial
learning.
    """

    # Make Discriminator weights non-trainable to prevent updates
during GAN training
    d_model.trainable = False
    # Get noise and label inputs from the Generator model
    gen_noise, gen_label = g_model.input
    # Get image output from the Generator model
    gen_output = g_model.output
    # Connect Generator output and label input to the Discriminator
for classification
    gan_output = d_model([gen_output, gen_label])
    # Define GAN model to take noise and label inputs and output a
classification
    model = Model([gen_noise, gen_label], gan_output)
    # Compile the GAN model using binary cross-entropy loss and Adam
optimizer
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt)
    return model

def load_real_samples():
    # Load dataset (all_frames: image frames, all_labels:
corresponding labels)
    (trainX, trainy) = all_frames, all_labels
```

```
    # Expand image dimensions to include the channel dimension
    X = expand_dims(trainX, axis=-1)

    # Convert image data type from integers to floats for
normalization
    X = X.astype('float32')

    # Normalize pixel values from [0, 255] to [-1, 1]
    X = (X - 127.5) / 127.5

    # Return preprocessed image data and corresponding labels
    return [X, trainy]

def generate_real_samples(dataset, n_samples):
    # Split dataset into images and labels
    images, labels = dataset

    # Choose random instances from the dataset
    ix = randint(0, images.shape[0], n_samples)

    # Select images and their corresponding labels based on the random
instances
    X, labels = images[ix], labels[ix]

    # Generate class labels indicating the samples are real
    y = ones((n_samples, 1))

    # Return selected real image samples with labels and class labels
    return [X, labels], y


# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples, n_classes=3):
    # generate points in the latent space
    x_input = randn(latent_dim * n_samples)
    # reshape into a batch of inputs for the network
    z_input = x_input.reshape(n_samples, latent_dim)
    # Define timing context choices (before reach, during reach, after
reach)
    choices = [410,510,610]
    # Generate random labels representing timing contexts for each
sample
    labels = np.random.choice(choices, size = n_samples)
    # Return generated latent points and corresponding random labels
    return [z_input, labels]

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):
     # generate points in latent space
     z_input, labels_input = generate_latent_points(latent_dim,
n_samples)
```

```
      # predict outputs
      images = generator.predict([z_input, labels_input])
      # Generate class labels indicating the samples are fake
      y = zeros((n_samples, 1))

      return [images, labels_input], y


# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim,
n_epochs=1000, n_batch=128):
      bat_per_epo = int(dataset[0].shape[0] / n_batch)
      half_batch = int(n_batch / 2)
      # manually enumerate epochs
      for i in range(n_epochs):
            # enumerate batches over the training set
            for j in range(bat_per_epo):
                  # get randomly selected 'real' samples
                  [X_real, labels_real], y_real =
generate_real_samples(dataset,half_batch)
                  # update discriminator model weights
                  d_loss1, _ = d_model.train_on_batch([X_real,
labels_real], y_real)
                  # generate 'fake' examples
                  [X_fake, labels], y_fake =
generate_fake_samples(g_model, latent_dim, half_batch)
                  # update discriminator model weights
                  d_loss2, _ = d_model.train_on_batch([X_fake, labels],
y_fake)
                  # prepare points in latent space as input for the
generator
                  [z_input, labels_input] =
generate_latent_points(latent_dim, n_batch)
                  # create inverted labels for the fake samples

                  y_gan = ones((n_batch, 1))
                  # update the generator via the discriminator's error
                  g_loss = gan_model.train_on_batch([z_input,
labels_input], y_gan)
                  # summarize loss on this batch
                  print('>%d, %d/%d, d1=%.3f, d2=%.3f g=%.3f' %
                        (i+1, j+1, bat_per_epo, d_loss1, d_loss2,
g_loss))
      # save the generator model
      g_model.save('NeuroVidGenModel.h5')


# size of the latent space
latent_dim = 100
# create the discriminator
```

```python
d_model = define_discriminator()
# create the generator
g_model = define_generator(latent_dim)
# create the gan
gan_model = define_gan(g_model, d_model)
# load image data
dataset = load_real_samples()
# train model
train(g_model, d_model, gan_model, dataset, latent_dim)
```

```
2/2 [==============================] - 1s 600ms/step
>1, 1/3, d1=0.703, d2=0.695 g=0.692
2/2 [==============================] - 0s 424ms/step
>1, 2/3, d1=0.630, d2=0.695 g=0.692
2/2 [==============================] - 0s 428ms/step
>1, 3/3, d1=0.553, d2=0.698 g=0.690
2/2 [==============================] - 0s 429ms/step
>2, 1/3, d1=0.454, d2=0.706 g=0.682
2/2 [==============================] - 0s 429ms/step
>2, 2/3, d1=0.335, d2=0.729 g=0.660
2/2 [==============================] - 0s 428ms/step
>2, 3/3, d1=0.207, d2=0.787 g=0.611
2/2 [==============================] - 0s 438ms/step
>3, 1/3, d1=0.123, d2=0.921 g=0.537
2/2 [==============================] - 0s 428ms/step
>3, 2/3, d1=0.100, d2=1.119 g=0.468
2/2 [==============================] - 0s 428ms/step
>3, 3/3, d1=0.138, d2=1.277 g=0.453
2/2 [==============================] - 0s 429ms/step
>4, 1/3, d1=0.228, d2=1.250 g=0.494
2/2 [==============================] - 0s 429ms/step
>4, 2/3, d1=0.362, d2=1.071 g=0.564
2/2 [==============================] - 0s 427ms/step
>4, 3/3, d1=0.471, d2=0.895 g=0.641
2/2 [==============================] - 0s 426ms/step
>5, 1/3, d1=0.544, d2=0.783 g=0.698
2/2 [==============================] - 0s 426ms/step
>5, 2/3, d1=0.582, d2=0.706 g=0.760
2/2 [==============================] - 0s 442ms/step
>5, 3/3, d1=0.603, d2=0.655 g=0.832
2/2 [==============================] - 0s 424ms/step
>6, 1/3, d1=0.621, d2=0.601 g=0.898
2/2 [==============================] - 0s 426ms/step
>6, 2/3, d1=0.624, d2=0.563 g=0.980
2/2 [==============================] - 0s 430ms/step
>6, 3/3, d1=0.614, d2=0.521 g=1.080
2/2 [==============================] - 0s 426ms/step
>7, 1/3, d1=0.619, d2=0.451 g=1.216
2/2 [==============================] - 0s 427ms/step
>7, 2/3, d1=0.611, d2=0.409 g=1.329
```

```
2/2 [==============================] - 1s 578ms/step
>997, 3/3, d1=0.526, d2=0.549 g=1.600
2/2 [==============================] - 1s 610ms/step
>998, 1/3, d1=0.517, d2=0.497 g=1.604
2/2 [==============================] - 1s 594ms/step
>998, 2/3, d1=0.561, d2=0.448 g=1.596
2/2 [==============================] - 1s 594ms/step
>998, 3/3, d1=0.427, d2=0.463 g=1.526
2/2 [==============================] - 1s 594ms/step
>999, 1/3, d1=0.441, d2=0.482 g=1.598
2/2 [==============================] - 1s 578ms/step
>999, 2/3, d1=0.411, d2=0.405 g=1.492
2/2 [==============================] - 1s 594ms/step
>999, 3/3, d1=0.517, d2=0.498 g=1.662
2/2 [==============================] - 1s 594ms/step
>1000, 1/3, d1=0.508, d2=0.460 g=1.552
2/2 [==============================] - 1s 578ms/step
>1000, 2/3, d1=0.469, d2=0.461 g=1.543
2/2 [==============================] - 1s 594ms/step
>1000, 3/3, d1=0.416, d2=0.425 g=1.673
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics
have yet to be built. `model.compile_metrics` will be empty until you
train or evaluate the model.
```

```python
def generate_and_save_images(generator, latent_dim, n_samples=10,
n_classes=3, save_path='NeuroVidGen_samples'):
    os.makedirs(save_path, exist_ok=True)  # Create the folder if it
doesn't exist

    # Generate random points in the latent space
    latent_points = np.random.randn(n_samples, latent_dim)
    # Generate random class labels
    choices = [410,510,610]
    labels = np.random.choice(choices, size = n_samples)
    # Generate images using the generator model
    generated_images = generator.predict([latent_points, labels])

    # Save the generated images
    for i in range(n_samples):
        image = generated_images[i].reshape(80, 80)
        label = labels[i]
        filename =
f"{save_path}/generated_image_{i}_class_{label}.png"
        plt.imsave(filename, image, cmap='gray')

# Assuming 'g_model' is the trained generator model and 'latent_dim'
is defined
generate_and_save_images(g_model, latent_dim)
```

```
1/1 [==============================] - 0s 20ms/step
```