Generative Adversarial Networks (GAN) is a type of neural network architecture for generative modeling. Generative modeling is the process of using a model to create new examples that logically result from a distribution of samples that already exists. For example, generating new photographs that are similar to but clearly distinct from a dataset of existing photographs is an example of generative modeling.

Two neural network models are used to train the generative algorithm (GAN). One model that learns to provide fresh, convincing samples is known as the "generator" or "generative network" model. The other model, commonly known as a "discriminator" or "discriminative network," is trained to distinguish between examples that are created and ones that are real.

The generative model can then be utilized to generate fresh plausible samples whenever needed after training. Generative Adversarial Networks have very specific applications and, in this task, I utilized it to generate new images from a set of portraits.

```
In [1]: import random
        import math
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        import seaborn as sns
        import PIL
        from PIL import Image
        import tensorflow  as tf
        from tensorflow.keras import layers
        from tensorflow.keras.models import Sequential
        from tensorflow.keras.layers import Dense, Reshape, UpSampling2D, Conv2D, BatchNormalization
        from tensorflow.keras.layers import LeakyReLU, Dropout, ZeroPadding2D, Flatten, Activation
        from tensorflow.keras.optimizers import Adam
        import tqdm
        import warnings
        warnings.filterwarnings("ignore")
        sns.set(rc={"axes.facecolor":"#EDE9DE","figure.facecolor":"#D8CA7E"})
```

Figure 1: Importing Libraries

These lines of codes show the libraries that were needed to complete this task.

```
In [2]: data_path = "Portraits"
        batch_s = 64
        data = tf.keras.preprocessing.image_dataset_from_directory(data_path, label_mode = None, image_size = (64,64), batch_size = batch

        Found 1000 files belonging to 1 classes.
```

Figure 2: Loading the Data

These lines of codes show how the data is loaded. This loads the data from the directory to prepare for training.

```
In [3]: latent_dim = 100
        g_resolution=2

        generator = Sequential()
        generator.add(Dense(4*4*256,activation="relu",input_dim=latent_dim))
        generator.add(Reshape((4,4,256)))
        generator.add(UpSampling2D())
        generator.add(Conv2D(256,kernel_size=3,padding="same"))
        generator.add(BatchNormalization(momentum=0.8))
        generator.add(Activation("relu"))
        generator.add(UpSampling2D())
        generator.add(Conv2D(256,kernel_size=3,padding="same"))
        generator.add(BatchNormalization(momentum=0.8))
        generator.add(Activation("relu"))
        generator.add(UpSampling2D())
        generator.add(Conv2D(256,kernel_size=3,padding="same"))#
        generator.add(BatchNormalization(momentum=0.8))
        generator.add(Activation("relu"))
        generator.add(UpSampling2D())
        generator.add(Conv2D(128,kernel_size=3,padding="same"))
        generator.add(BatchNormalization(momentum=0.8))
        generator.add(Activation("relu"))
        generator.add(Conv2D(3,kernel_size=3,padding="same"))
        generator.add(Activation("tanh"))

        generator.summary()
```

Figure 3: Building a Generator

These lines of codes start the building of the GAN. This is the generator network that will produce new plausible samples.

```
In [4]: seed = tf.random.normal([1, latent_dim])
        Generated_Portrait = generator(seed, training=False)
        plt.imshow(Generated_Portrait[0, :, :, 0])
        plt.axis("off")
```

Figure 4: Creating Random Output from the Generator

These codes are for visualization of the generated random seed. Let us see the output of the generator without training.

Out[4]: (-0.5, 63.5, 63.5, -0.5)

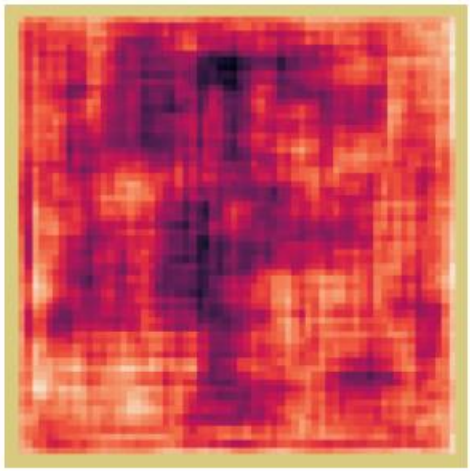

Figure 5: Generated Portrait Without Training

This image shows the output random seed from the untrained generator. Clearly, the output only contains noise since the generator is not yet trained.

In [5]:
```
discriminator = Sequential()
discriminator.add(Conv2D(32, kernel_size=3, strides=2, input_shape=(64,64,3), padding="same"))
discriminator.add(LeakyReLU(alpha=0.2))
discriminator.add(Dropout(0.25))
discriminator.add(Conv2D(64, kernel_size=3, strides=2, padding="same"))
discriminator.add(ZeroPadding2D(padding=((0,1),(0,1))))
discriminator.add(BatchNormalization(momentum=0.8))
discriminator.add(LeakyReLU(alpha=0.2))
discriminator.add(Dropout(0.25))
discriminator.add(Conv2D(128, kernel_size=3, strides=2, padding="same"))
discriminator.add(BatchNormalization(momentum=0.8))
discriminator.add(LeakyReLU(alpha=0.2))
discriminator.add(Dropout(0.25))
discriminator.add(Conv2D(256, kernel_size=3, strides=1, padding="same"))
discriminator.add(BatchNormalization(momentum=0.8))
discriminator.add(LeakyReLU(alpha=0.2))
discriminator.add(Dropout(0.25))
discriminator.add(Conv2D(512, kernel_size=3, strides=1, padding="same"))
discriminator.add(BatchNormalization(momentum=0.8))
discriminator.add(LeakyReLU(alpha=0.2))
discriminator.add(Dropout(0.25))
discriminator.add(Flatten())
discriminator.add(Dense(1, activation="sigmoid"))

discriminator.summary()
```

Figure 6: Building a Discriminator

These lines of codes show the building of the discriminator network that will differentiate generated examples from real examples.

```
In [6]: Discriminator_Verdict = discriminator(Generated_Portrait)
        print (Discriminator_Verdict)

        tf.Tensor([[0.4999984]], shape=(1, 1), dtype=float32)
```

Figure 7: Verdict Without Training

This shows the verdict the discriminator network has for the previously generated image that contains random noise. The verdict says that there is almost a fifty-fifty chance of the image being real. This is due to the discriminator being untrained.

```
In [7]: class GAN(tf.keras.Model):
            def __init__(self, discriminator, generator, latent_dim):
                super(GAN, self).__init__()
                self.discriminator = discriminator
                self.generator = generator
                self.latent_dim = latent_dim

            def compile(self, d_optimizer, g_optimizer, loss_fn):
                super(GAN, self).compile()
                self.d_optimizer = d_optimizer
                self.g_optimizer = g_optimizer
                self.loss_fn = loss_fn
                self.d_loss_metric = tf.keras.metrics.Mean(name="d_loss")
                self.g_loss_metric = tf.keras.metrics.Mean(name="g_loss")

            @property
            def metrics(self):
                return [self.d_loss_metric, self.g_loss_metric]
```

Figure 8: Defining the GAN Training

```
    @property
    def metrics(self):
        return [self.d_loss_metric, self.g_loss_metric]

    def train_step(self, real_images):
        batch_size = tf.shape(real_images)[0]
        seed = tf.random.normal(shape=(batch_size, self.latent_dim))
        generated_images = self.generator(seed)
        combined_images = tf.concat([generated_images, real_images], axis=0)
        labels = tf.concat([tf.ones((batch_size, 1)), tf.zeros((batch_size, 1))], axis=0)
        labels += 0.05 * tf.random.uniform(tf.shape(labels))
        with tf.GradientTape() as tape:
            predictions = self.discriminator(combined_images)
            d_loss = self.loss_fn(labels, predictions)
        grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
        self.d_optimizer.apply_gradients(zip(grads, self.discriminator.trainable_weights))

        seed = tf.random.normal(shape=(batch_size, self.latent_dim))

        misleading_labels = tf.zeros((batch_size, 1))

        with tf.GradientTape() as tape:
            predictions = self.discriminator(self.generator(seed))
            g_loss = self.loss_fn(misleading_labels, predictions)
        grads = tape.gradient(g_loss, self.generator.trainable_weights)
        self.g_optimizer.apply_gradients(zip(grads, self.generator.trainable_weights))

        self.d_loss_metric.update_state(d_loss)
        self.g_loss_metric.update_state(g_loss)
        return {"d_loss": self.d_loss_metric.result(), "g_loss": self.g_loss_metric.result()}
```

Figure 9: Defining the GAN Training

Figures 8 and 9 show the compilation of the GAN to prepare for training.

```
In [8]: epochs = 100
        discriminator_opt = tf.keras.optimizers.Adamax(1.5e-4,0.5)
        generator_opt = tf.keras.optimizers.Adamax(1.5e-4,0.5)
        loss_fn = tf.keras.losses.BinaryCrossentropy()

        model = GAN(discriminator=discriminator, generator=generator, latent_dim=latent_dim)

        model.compile(d_optimizer=discriminator_opt, g_optimizer=generator_opt, loss_fn=loss_fn)

        history = model.fit(data, epochs=epochs)
```

```
16/16 [==============================] - 125s 8s/step - d_loss: -22444.8301 - g_loss: 2012494.6250
Epoch 92/100
16/16 [==============================] - 125s 8s/step - d_loss: -26322.5430 - g_loss: 2340417.5000
Epoch 93/100
16/16 [==============================] - 125s 8s/step - d_loss: -30504.1406 - g_loss: 2705058.0000
Epoch 94/100
16/16 [==============================] - 125s 8s/step - d_loss: -33306.2461 - g_loss: 3101579.0000
Epoch 95/100
16/16 [==============================] - 144s 9s/step - d_loss: -38857.3789 - g_loss: 3537119.7500
Epoch 96/100
16/16 [==============================] - 142s 9s/step - d_loss: -49190.3008 - g_loss: 4120048.0000
Epoch 97/100
16/16 [==============================] - 144s 9s/step - d_loss: -54181.0898 - g_loss: 4779692.0000
Epoch 98/100
16/16 [==============================] - 147s 9s/step - d_loss: -61090.2305 - g_loss: 5449711.0000
Epoch 99/100
16/16 [==============================] - 143s 9s/step - d_loss: -70195.7344 - g_loss: 6176433.0000
Epoch 100/100
16/16 [==============================] - 142s 9s/step - d_loss: -76300.5938 - g_loss: 6939270.0000
```

Figure 10: Training the Model

Training the model, we defined the number of epochs, the optimizers for generator and discriminator networks, and the cross-entropy loss.

GAN Training has two sections:

1. The discriminator is trained while the generator is idle. The discriminator is trained using real images and random noise from the still untrained generator. This trains it to tell between fake and real. This accommodates the discriminator to predict as fakes.
2. The generator is trained while the discriminator is idle. In this section, the generator is trained. After training the discriminator, this step uses the predictions from the discriminator. To level up the contest, the generator needs to adjust the weights in an attempt to deceive the discriminator.

The process is repeated based on the number of epochs.

```
In [10]: def Show_Img(data):
             plt.figure(figsize=(15,15))
             for images in data.take(1):
                 for i in range(18):
                     ax = plt.subplot(6, 6, i + 1)
                     ax.imshow(images[i].numpy().astype("uint8"))
                     ax.axis("off")

         Show_Img(data)
```

Figure 11: Visualizing the Output

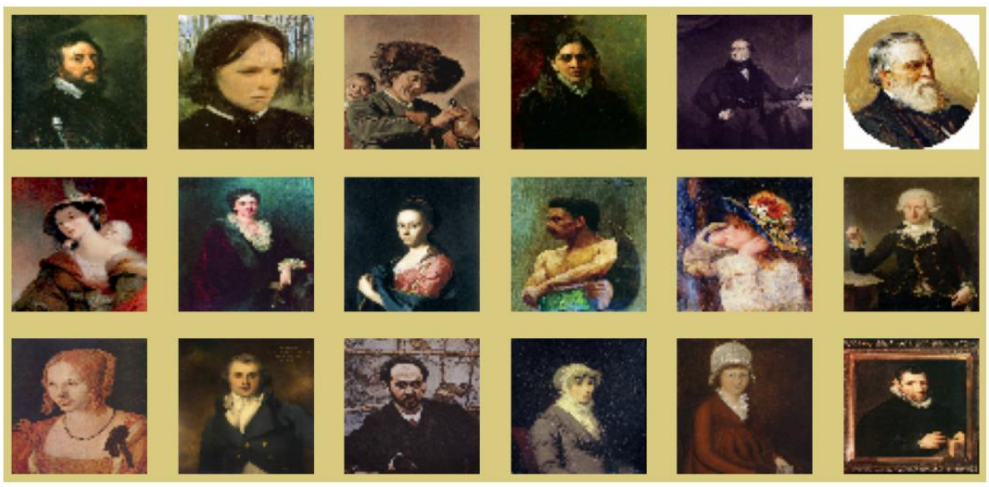

Figure 12: Generated Output

Figure 12 shows the generated output of the GAN. As seen from the figure, I defined the number of images to be generated by the GAN and it showed 18 new images. From a dataset of 1000 images, I can say that the GAN picked up the patterns in the portraits in able to generate new set of portraits.