

Week 5: GANs Mini-Project

Overview

We recognize the works of artists through their unique style, such as color choices or brush strokes. The “je ne sais quoi” of artists like Claude Monet can now be imitated with algorithms thanks to generative adversarial networks (GANs). In this getting started competition, you will bring that style to your photos or recreate the style from scratch!

Computer vision has advanced tremendously in recent years and GANs are now capable of mimicking objects in a very convincing way. But creating museum-worthy masterpieces is thought of to be, well, more art than science. So can (data) science, in the form of GANs, trick classifiers into believing you’ve created a true Monet? That’s the challenge you’ll take on!

(Please refer to <https://www.kaggle.com/competitions/gan-getting-started/overview> for additional information about this dataset.)

Step 1 Brief Description

1.1 Problem

- **Challenge:** The goal of this challenge is to develop a **Generative Adversarial Network (GAN)** capable of generating between 7,000 to 10,000 Monet-style images. A GAN consists of two competing neural networks: a **generator** and a **discriminator**. The generator is responsible for creating Monet-style images, while the discriminator evaluates whether the generated images are real (from the original Monet dataset) or fake (produced by the generator). These two networks are trained in an adversarial manner—meaning the generator continuously improves to fool the discriminator, and the discriminator learns to better distinguish between real and generated images. The challenge lies in achieving a balance where the generator produces high-quality, realistic Monet-style artwork while ensuring the discriminator does not become too dominant and stifle the learning process.
- **GAN:** For this challenge, I have chosen to implement a **Deep Convolutional GAN (DCGAN)** as the generative model. DCGAN enhances traditional GANs by using convolutional and transposed convolutional layers to generate high-quality images. The **generator** takes random noise as input and upsamples it through convolutional layers with batch normalization and LeakyReLU activation to create Monet-style images. The **discriminator** analyzes input images using convolutional layers to distinguish between real Monet paintings and generated ones. DCGAN’s architectural improvements, such as convolutional layers and batch normalization, help stabilize training and produce visually coherent Monet-style paintings by capturing key artistic features like color tones, brushstrokes, and composition.

1.2 Data

- **Dataset:**
 - **monet_jpg** : Monet paintings used for training to learn and generate Monet-style artworks.
 - **photo_jpg** : Real-world photos intended for style transfer tasks, not used in this project as DCGAN generates images from random noise instead of transforming existing photos.
- **Size and Dimensions:**
 - **Number of images:** 300
 - **Image resolution:** 256 x 256 pixels
 - **File format:** JPEG
 - **Color Channels:** 3 (RGB)

Step 2 EDA

2.1 Data Preparation

- **Loading and Preprocessing the Dataset:** Using TensorFlow’s Keras API to load images and apply essential preprocessing steps, including resizing, normalization, and batching, to prepare the data for model training.

```
In [6]: import tensorflow as tf
import numpy as np
import pandas as pd
import os
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.image import load_img, img_to_array
import warnings
warnings.filterwarnings("ignore")
```

```
In [3]: # Set image size and batch size
IMG_SIZE = (256, 256) # Image size
BATCH_SIZE = 16      # Batch size

# Set data path
MONET_PATH = '/kaggle/input/gan-getting-started/monet_jpg'
```

```
In [4]: # Image preprocessing function
def load_and_preprocess_image(image_path):
    img = load_img(image_path.decode('utf-8'), target_size=IMG_SIZE) # Load image
    img = img_to_array(img) # Convert to array
    img = (img / 127.5) - 1 # Normalize to [-1, 1]
    return img.astype(np.float32)
```

```
In [5]: # Get the list of image files
monet_images = np.array([os.path.join(MONET_PATH, f) for f in os.listdir(MONET_PATH)])

# Create TensorFlow Dataset
def create_dataset(image_list):
    dataset = tf.data.Dataset.from_tensor_slices(image_list)
    dataset = dataset.map(lambda x: tf.numpy_function(func=load_and_preprocess_image, inp=[x], Tout=tf.float32))
    dataset = dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
    return dataset

# Create the dataset
```

```
monet_ds = create_dataset(monet_images)

# Have a Look at the batch shape
for img_batch in monet_ds.take(1):
    print(f"Monet batch shape: {img_batch.shape}")
```

Monet batch shape: (16, 256, 256, 3)

2.2 Visualizations

- **Monet Paintings:** Displaying four Monet paintings from the dataset to visually inspect the data quality and confirm proper preprocessing.

```
In [5]: # Have a Look at Monet paintings
plt.figure(figsize=(16, 4))

for i, image in enumerate(monet_ds.take(1)): # Take one batch
    for j in range(4): # Display the first 4 images
        plt.subplot(1, 4, j + 1) # 1x4 grid display
        plt.imshow((image[j].numpy() + 1) / 2) # Convert back to [0, 1] range for display
        plt.title(f"Monet {j+1}")
        plt.axis('off')

plt.subplots_adjust(wspace=0.1, hspace=0.3)
plt.suptitle("\nSample Monet Images\n", fontsize=15, y=1.1)

plt.show()
```

Sample Monet Images

Monet 1



Monet 2



Monet 3



Monet 4



- **Edge Detection:** Applying Canny edge detection to Monet paintings and visualizing the extracted edges to analyze the structural details and contours in grayscale.

```
In [24]: import cv2

In [33]: plt.figure(figsize=(16, 4))

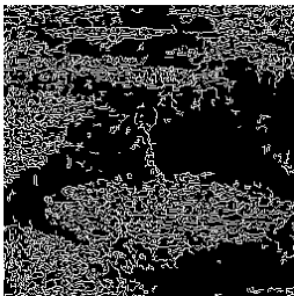
for i, img_path in enumerate(monet_images[:4]):
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE) # Read image in grayscale
    edges = cv2.Canny(img, 100, 200) # Apply Canny edge detection

    plt.subplot(1, 4, i + 1) # 1 row, 4 columns
    plt.imshow(edges, cmap='gray')
    plt.title(f"Monet {i+1}")
    plt.axis('off')

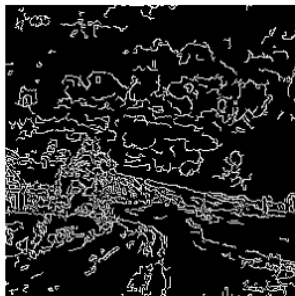
plt.suptitle("\nEdge Detection on Monet Paintings\n", fontsize=15, y=1.1)
plt.subplots_adjust(wspace=0.3) # Adjust spacing between images
plt.show()
```

Edge Detection on Monet Paintings

Monet 1



Monet 2



Monet 3



Monet 4



2.3 Checking for Duplicate

- Counting the total and unique Monet paintings in the dataset to identify any duplicate images.

```
In [30]: unique_images = len(set(monet_images))
total_images = len(monet_images)
print(f"Total images: {total_images}, Unique images: {unique_images}")
if unique_images < total_images:
    print("Warning: Duplicate images found.")
```

Step 3 Model Building and Training

3.1 Building: First DCGAN Model

We will build a basic Deep Convolutional Generative Adversarial Network (DCGAN).

- **Step Overview :**

- **Build the Generator:** The generator is responsible for transforming random noise (a latent vector) into Monet-style images. It consists of a series of transposed convolutional layers (also known as deconvolution layers) that progressively upsample the noise into a structured image. The generator aims to produce images that resemble real Monet paintings as closely as possible, learning the artistic style and color patterns.
- **Build the Discriminator:** The discriminator acts as a binary classifier that distinguishes between real Monet paintings and artificially generated ones. It consists of convolutional layers that downsample the input image, extracting high-level features to assess authenticity. The discriminator's goal is to correctly identify whether an image is real (from the Monet dataset) or fake (created by the generator), thereby providing feedback to improve the generator's performance over time.

Through the adversarial training process, the generator and discriminator are optimized simultaneously in a competitive setup, leading to the production of increasingly realistic Monet-style paintings.

3.1.1 Generator

- **Input Layer:** Takes a 1D random noise vector of size `latent_dim` (128).
- **Dense Layer:** Expands the noise vector to a feature map of size `16x16x512`.
- **Reshape Layer:** Converts the flattened vector into a 3D feature map (16x16x512).
- **Transposed Convolutions Layers (Upsampling):**
 - `16x16 → 32x32` (256 filters)
 - `32x32 → 64x64` (128 filters)
 - `64x64 → 128x128` (64 filters)
 - `128x128 → 256x256` (3 output channels)
- **Activation and Normalization:** Each transposed convolution is followed by `Batch Normalization` and `LeakyReLU` activation (except the final layer which uses `tanh` activation).
- **Output Layer:** Produces a final `256x256x3` image with pixel values normalized to the range [-1, 1]

```
In [6]: from tensorflow.keras.layers import Dense, Reshape, Conv2DTranspose, BatchNormalization, LeakyReLU, Input
from tensorflow.keras.models import Model
```

```
In [7]: def build_generator(latent_dim):
    inputs = Input(shape=(latent_dim,)) # Receive random noise vector
    x = Dense(16 * 16 * 512, use_bias=False)(inputs)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Reshape((16, 16, 512))(x) # Reshape into 16x16x512 feature map

    x = Conv2DTranspose(256, kernel_size=4, strides=2, padding="same", use_bias=False)(x) # Upsample to 32x32
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)

    x = Conv2DTranspose(128, kernel_size=4, strides=2, padding="same", use_bias=False)(x) # Upsample to 64x64
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)

    x = Conv2DTranspose(64, kernel_size=4, strides=2, padding="same", use_bias=False)(x) # Upsample to 128x128
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)

    # Ensure output is 256x256x3, which is the final upsampling
    x = Conv2DTranspose(3, kernel_size=4, strides=2, padding="same", activation="tanh")(x) # Upsample to 256x256

    model = Model(inputs, x, name="generator")
    return model

latent_dim = 128 # Dimension of random noise
generator = build_generator(latent_dim)
generator.summary()
```

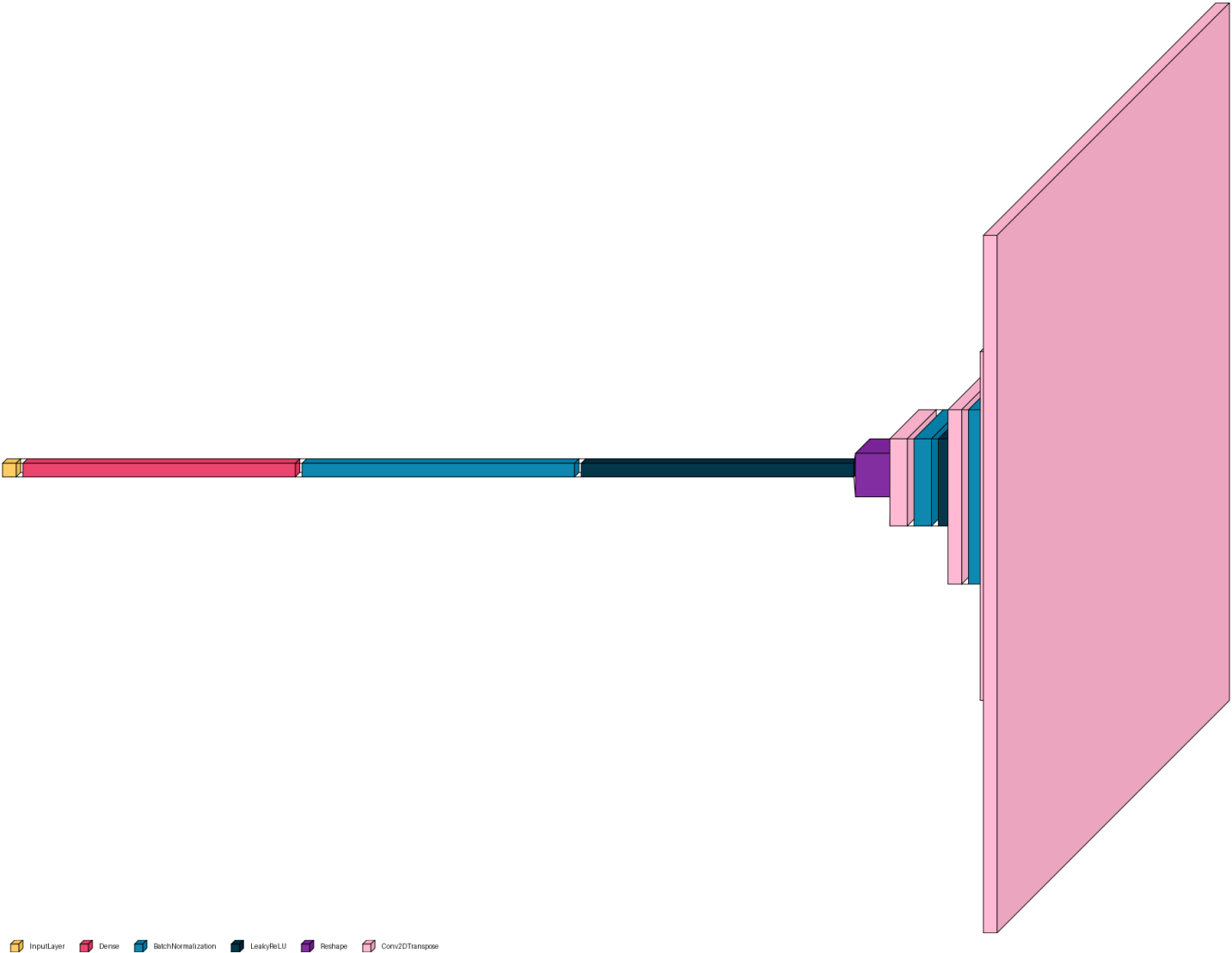
Model: "generator"

| Layer (type) | Output Shape | Param # |
|--|----------------------|------------|
| input_layer (InputLayer) | (None, 128) | 0 |
| dense (Dense) | (None, 131072) | 16,777,216 |
| batch_normalization (BatchNormalization) | (None, 131072) | 524,288 |
| leaky_re_lu (LeakyReLU) | (None, 131072) | 0 |
| reshape (Reshape) | (None, 16, 16, 512) | 0 |
| conv2d_transpose (Conv2DTranspose) | (None, 32, 32, 256) | 2,097,152 |
| batch_normalization_1 (BatchNormalization) | (None, 32, 32, 256) | 1,024 |
| leaky_re_lu_1 (LeakyReLU) | (None, 32, 32, 256) | 0 |
| conv2d_transpose_1 (Conv2DTranspose) | (None, 64, 64, 128) | 524,288 |
| batch_normalization_2 (BatchNormalization) | (None, 64, 64, 128) | 512 |
| leaky_re_lu_2 (LeakyReLU) | (None, 64, 64, 128) | 0 |
| conv2d_transpose_2 (Conv2DTranspose) | (None, 128, 128, 64) | 131,072 |
| batch_normalization_3 (BatchNormalization) | (None, 128, 128, 64) | 256 |
| leaky_re_lu_3 (LeakyReLU) | (None, 128, 128, 64) | 0 |
| conv2d_transpose_3 (Conv2DTranspose) | (None, 256, 256, 3) | 3,075 |

Total params: 20,058,883 (76.52 MB)
Trainable params: 19,795,843 (75.52 MB)
Non-trainable params: 263,040 (1.00 MB)

```
In [9]: os.system('pip install visualkeras')
import visualkeras
visualkeras.layered_view(generator, legend=True)
```

Out[9]:



InputLayer Dense BatchNormalization LeakyReLU Reshape Conv2DTranspose

3.1.2 Discriminator

- Input Layer: Takes an image of shape (256, 256, 3) (height, width, and RGB channels).
- Convolutional Layers (Downsampling):
 - 1st Layer: 64 filters, kernel size 4x4, stride 2 (output: 128x128x64).

- **2nd Layer:** 128 filters, kernel size 4x4, stride 2 (output: 64x64x128).
 - **3rd Layer:** 256 filters, kernel size 4x4, stride 2 (output: 32x32x256).
- **Activation and Regularization:**
 - **LeakyReLU Activation:** With an alpha of 0.2 to allow small negative values.
 - **Dropout Layer:** With a rate of 0.3 to reduce overfitting by randomly deactivating neurons during training.
 - **Flattening Layer:** Converts the final 3D feature map into a 1D vector for classification.
 - **Output Layer:** A fully connected (Dense) layer with a single neuron using sigmoid activation, which outputs a probability score indicating whether the input image is real (close to 1) or fake (close to 0).

```
In [11]: from tensorflow.keras.layers import Conv2D, Flatten, Dropout
```

```
In [12]: def build_discriminator(img_shape):
    inputs = Input(shape=img_shape) # Input: 256x256x3 image

    x = Conv2D(64, kernel_size=4, strides=2, padding="same")(inputs) # Downsamples to 128x128x64
    x = LeakyReLU(alpha=0.2)(x)
    x = Dropout(0.3)(x)

    x = Conv2D(128, kernel_size=4, strides=2, padding="same")(x) # Downsamples to 64x64x128
    x = LeakyReLU(alpha=0.2)(x)
    x = Dropout(0.3)(x)

    x = Conv2D(256, kernel_size=4, strides=2, padding="same")(x) # Downsamples to 32x32x256
    x = LeakyReLU(alpha=0.2)(x)
    x = Dropout(0.3)(x)

    x = Flatten()(x) # Flatten the feature maps to a 1D vector for classification
    x = Dense(1, activation="sigmoid")(x) # Output Layer

    model = Model(inputs, x, name="discriminator")
    return model

discriminator = build_discriminator((256, 256, 3))
discriminator.summary()
```

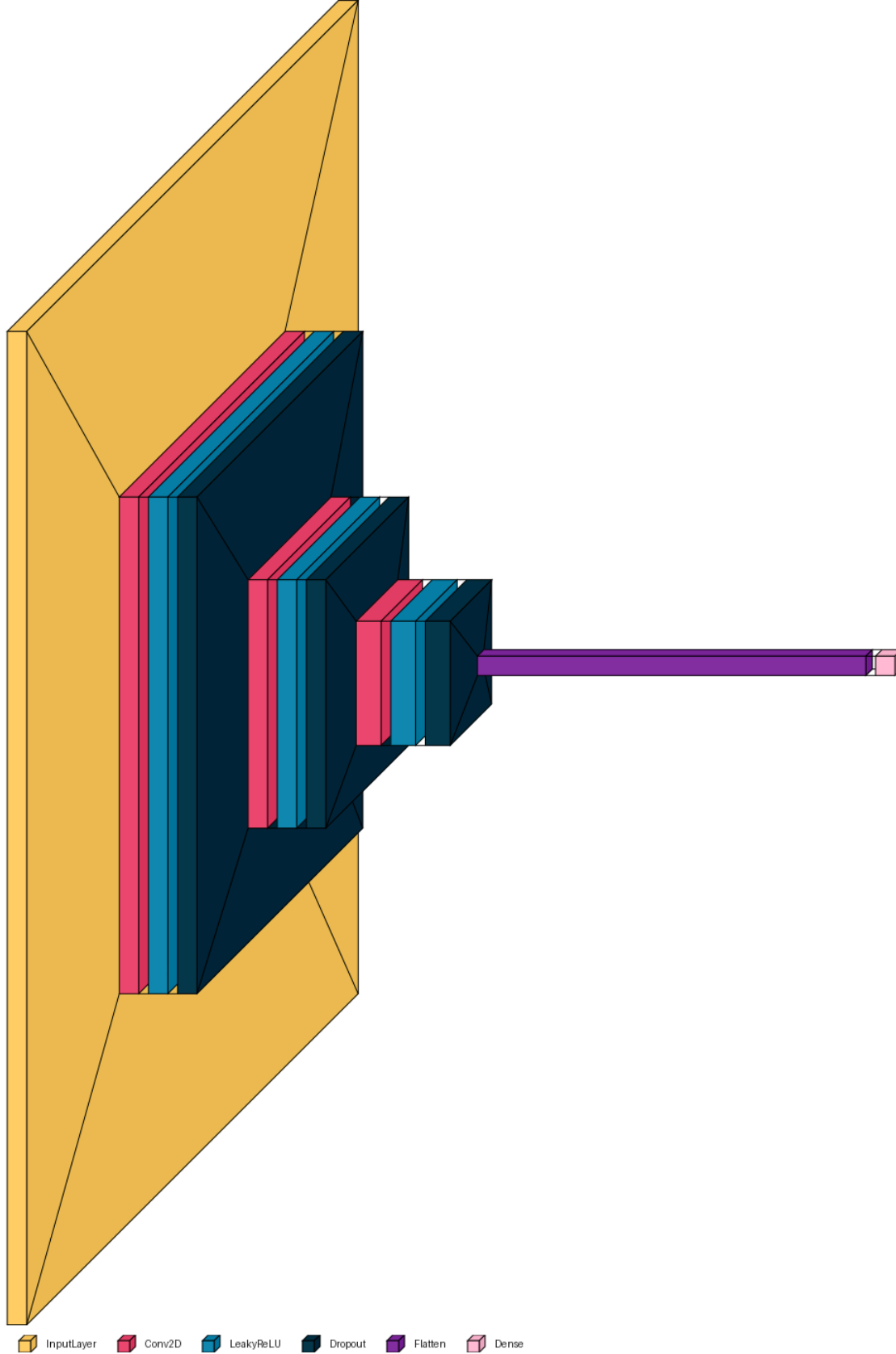
Model: "discriminator"

| Layer (type) | Output Shape | Param # |
|----------------------------|----------------------|---------|
| input_layer_1 (InputLayer) | (None, 256, 256, 3) | 0 |
| conv2d (Conv2D) | (None, 128, 128, 64) | 3,136 |
| leaky_re_lu_4 (LeakyReLU) | (None, 128, 128, 64) | 0 |
| dropout (Dropout) | (None, 128, 128, 64) | 0 |
| conv2d_1 (Conv2D) | (None, 64, 64, 128) | 131,200 |
| leaky_re_lu_5 (LeakyReLU) | (None, 64, 64, 128) | 0 |
| dropout_1 (Dropout) | (None, 64, 64, 128) | 0 |
| conv2d_2 (Conv2D) | (None, 32, 32, 256) | 524,544 |
| leaky_re_lu_6 (LeakyReLU) | (None, 32, 32, 256) | 0 |
| dropout_2 (Dropout) | (None, 32, 32, 256) | 0 |
| flatten (Flatten) | (None, 262144) | 0 |
| dense_1 (Dense) | (None, 1) | 262,145 |

Total params: 921,025 (3.51 MB)
 Trainable params: 921,025 (3.51 MB)
 Non-trainable params: 0 (0.00 B)

```
In [13]: visualkeras.layered_view(discriminator, legend=True)
```

Out[13]:



3.1.3 Loss Function & Optimizers

- Loss Function:

- `BinaryCrossentropy` is used as the loss function to measure the accuracy of real vs. generated images.
- It evaluates how well the discriminator distinguishes real from fake images by comparing predicted and actual labels.

- Optimizers:

- The `Adam` optimizer is chosen to optimize the weights of both the generator and discriminator.
- A learning rate of `0.0002` ensures gradual updates for stable learning.
- `beta_1=0.5` is used to reduce oscillations and improve convergence stability.

```
In [10]: # Define the loss function for the generator and discriminator
loss_fn = tf.keras.losses.BinaryCrossentropy()

# Optimizer for the generator
generator_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)

# Optimizer for the discriminator
discriminator_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)
```

3.1.4 Training

The training process involves iterating over the dataset for a specified number of epochs. In each step, random noise is fed into the generator to produce fake images, which are then evaluated by the discriminator alongside real images. The generator is optimized to produce images that the discriminator classifies as real, while the discriminator is trained to distinguish real from fake images accurately. Gradients are computed and applied to update the weights of both models. Throughout training, the generator and discriminator losses are recorded and displayed, helping to monitor the progress and performance of the DCGAN.

```
In [11]: # Record Losses
gen_losses = []
disc_losses = []

# Define a single training step, using @tf.function to optimize execution speed
@tf.function
def train_step(real_images):
    # Generate random noise as input to the generator
    noise = tf.random.normal([BATCH_SIZE, latent_dim])

    # Track gradients
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        # Use the generator to create fake images
        generated_images = generator(noise, training=True)

        # Discriminator's output for real images
        real_output = discriminator(real_images, training=True)

        # Discriminator's output for fake (generated) images
        fake_output = discriminator(generated_images, training=True)

        # Compute generator loss
        gen_loss = loss_fn(tf.ones_like(fake_output), fake_output)

        # Compute discriminator loss
        disc_loss_real = loss_fn(tf.ones_like(real_output), real_output)
        disc_loss_fake = loss_fn(tf.zeros_like(fake_output), fake_output)
        disc_loss = disc_loss_real + disc_loss_fake

    # Calculate gradients
    gen_gradients = gen_tape.gradient(gen_loss, generator.trainable_variables)
    disc_gradients = disc_tape.gradient(disc_loss, discriminator.trainable_variables)

    # Use optimizers to update the weights of the generator and discriminator
    generator_optimizer.apply_gradients(zip(gen_gradients, generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(disc_gradients, discriminator.trainable_variables))

    return gen_loss, disc_loss

# Define the overall training process
def train(dataset, epochs=50):
    for epoch in range(epochs):
        for image_batch in dataset:
            gen_loss, disc_loss = train_step(image_batch)

            gen_losses.append(gen_loss.numpy())
            disc_losses.append(disc_loss.numpy())

        print(f"Epoch {epoch+1}, Generator Loss: {gen_loss:.4f}, Discriminator Loss: {disc_loss:.4f}")

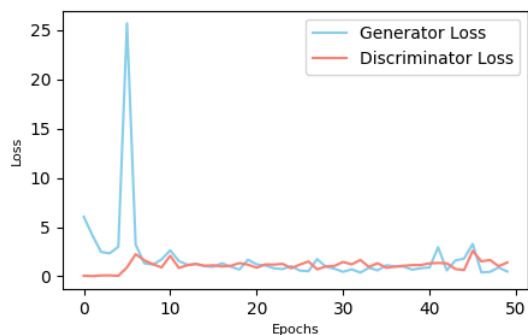
# Start training, running for 50 epochs
train(monet_ds, epochs=50)
```

```
Epoch 1, Generator Loss: 6.0504, Discriminator Loss: 0.0599
Epoch 2, Generator Loss: 4.1659, Discriminator Loss: 0.0311
Epoch 3, Generator Loss: 2.4785, Discriminator Loss: 0.0893
Epoch 4, Generator Loss: 2.3366, Discriminator Loss: 0.1025
Epoch 5, Generator Loss: 3.0087, Discriminator Loss: 0.0511
Epoch 6, Generator Loss: 25.6728, Discriminator Loss: 0.9115
Epoch 7, Generator Loss: 3.2392, Discriminator Loss: 2.2481
Epoch 8, Generator Loss: 1.3261, Discriminator Loss: 1.6336
Epoch 9, Generator Loss: 1.1974, Discriminator Loss: 1.2369
Epoch 10, Generator Loss: 1.7158, Discriminator Loss: 0.9075
Epoch 11, Generator Loss: 2.6453, Discriminator Loss: 2.0893
Epoch 12, Generator Loss: 1.5582, Discriminator Loss: 0.8547
Epoch 13, Generator Loss: 1.1762, Discriminator Loss: 1.1354
Epoch 14, Generator Loss: 1.2316, Discriminator Loss: 1.2743
Epoch 15, Generator Loss: 1.0310, Discriminator Loss: 1.0734
Epoch 16, Generator Loss: 0.9658, Discriminator Loss: 1.1265
Epoch 17, Generator Loss: 1.3326, Discriminator Loss: 1.0046
Epoch 18, Generator Loss: 0.9830, Discriminator Loss: 1.0687
Epoch 19, Generator Loss: 0.6905, Discriminator Loss: 1.3493
Epoch 20, Generator Loss: 1.6877, Discriminator Loss: 1.1873
Epoch 21, Generator Loss: 1.2163, Discriminator Loss: 0.8776
Epoch 22, Generator Loss: 1.1106, Discriminator Loss: 1.2155
Epoch 23, Generator Loss: 0.8195, Discriminator Loss: 1.1928
Epoch 24, Generator Loss: 0.7376, Discriminator Loss: 1.2734
Epoch 25, Generator Loss: 1.0431, Discriminator Loss: 0.8213
Epoch 26, Generator Loss: 0.5824, Discriminator Loss: 1.1965
Epoch 27, Generator Loss: 0.5260, Discriminator Loss: 1.5089
Epoch 28, Generator Loss: 1.7538, Discriminator Loss: 0.7138
Epoch 29, Generator Loss: 0.9919, Discriminator Loss: 1.0060
Epoch 30, Generator Loss: 0.7749, Discriminator Loss: 1.0595
Epoch 31, Generator Loss: 0.4623, Discriminator Loss: 1.4626
Epoch 32, Generator Loss: 0.7107, Discriminator Loss: 1.2203
Epoch 33, Generator Loss: 0.3838, Discriminator Loss: 1.6638
Epoch 34, Generator Loss: 0.8801, Discriminator Loss: 0.9595
Epoch 35, Generator Loss: 0.6054, Discriminator Loss: 1.3324
Epoch 36, Generator Loss: 1.1257, Discriminator Loss: 0.8717
Epoch 37, Generator Loss: 1.0144, Discriminator Loss: 0.9813
Epoch 38, Generator Loss: 0.9960, Discriminator Loss: 1.0767
Epoch 39, Generator Loss: 0.6824, Discriminator Loss: 1.1428
Epoch 40, Generator Loss: 0.8339, Discriminator Loss: 1.1570
Epoch 41, Generator Loss: 0.8953, Discriminator Loss: 1.3185
Epoch 42, Generator Loss: 2.9501, Discriminator Loss: 1.3626
Epoch 43, Generator Loss: 0.6044, Discriminator Loss: 1.3262
Epoch 44, Generator Loss: 1.6135, Discriminator Loss: 0.7456
Epoch 45, Generator Loss: 1.8025, Discriminator Loss: 0.6416
Epoch 46, Generator Loss: 3.2938, Discriminator Loss: 2.5914
Epoch 47, Generator Loss: 0.4029, Discriminator Loss: 1.5187
Epoch 48, Generator Loss: 0.4626, Discriminator Loss: 1.6637
Epoch 49, Generator Loss: 0.8984, Discriminator Loss: 1.0280
Epoch 50, Generator Loss: 0.4863, Discriminator Loss: 1.4144
```

3.1.5 Visualize Results

```
In [12]: # Visualize Loss
plt.figure(figsize=(5, 3))
plt.plot(gen_losses, label='Generator Loss', color='skyblue')
plt.plot(disc_losses, label='Discriminator Loss', color='salmon')
plt.xlabel('Epochs', fontsize=8)
plt.ylabel('Loss', fontsize=8)
plt.title('\nTraining Loss\n', fontsize=12)
plt.legend()
plt.show()
```

Training Loss

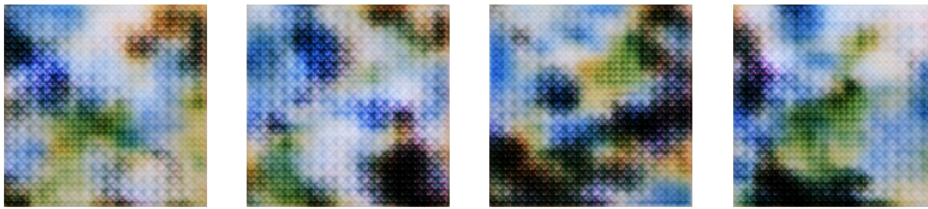


```
In [13]: # Generate and display images
def generate_and_show_images(generator, num_images=4):
    noise = tf.random.normal([num_images, latent_dim]) # Generate random noise as input
    generated_images = generator(noise, training=False) # Generate images using the generator

    plt.figure(figsize=(10, 3))
    for i in range(num_images):
        plt.subplot(1, num_images, i + 1)
        plt.imshow((generated_images[i].numpy() + 1) / 2) # Convert back to [0,1] range for display
        plt.axis('off')
    plt.suptitle("\nGenerated Monet-style Images\n", fontsize=12)
    plt.show()

generate_and_show_images(generator)
```


Generated Monet-style Images



3.2 Hyperparameter Tuning: Second DCGAN Model

- **Training Duration:** Increase the number of `epochs` from 50 to 200 to allow for more thorough training and better convergence.
- **Learning Rate:** Reduce the generator's `learning_rate` to 0.0001 to balance training dynamics and prevent the discriminator from overpowering the generator.
- **Momentum:** Adjust the `beta_1` parameter to 0.4 to stabilize training and improve the Adam optimizer's momentum behavior.
- **Label Smoothing:** Implement label smoothing, replacing the real label value of 1 with a random value between 0.9 and 1.0, to enhance stability and prevent overfitting.
- **Training Frequency:** Train the generator twice per batch to help reduce mode collapse and improve the diversity of generated images.

3.2.1 Loss Function & Optimizers

```
In [14]: loss_fn = tf.keras.losses.BinaryCrossentropy()
generator_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001, beta_1=0.4)
discriminator_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.4)
```

3.2.2 Training

```
In [15]: gen_losses = []
disc_losses = []

@tf.function
def train_step(real_images):
    noise = tf.random.normal([BATCH_SIZE, latent_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(real_images, training=True)
        fake_output = discriminator(generated_images, training=True)

        # Label Smoothing: Replace real label 1 with 0.9, keeping fake label as 0
        real_labels = tf.random.uniform(shape=tf.shape(real_output), minval=0.9, maxval=1.0) # Smoothed positive labels
        fake_labels = tf.zeros_like(fake_output) # Fake labels remain 0

        # Compute Losses
        gen_loss = loss_fn(tf.ones_like(fake_output), fake_output) # Generator tries to create real-looking images
        disc_loss_real = loss_fn(real_labels, real_output) # Using smoothed positive labels
        disc_loss_fake = loss_fn(fake_labels, fake_output) # Fake labels remain 0
        disc_loss = disc_loss_real + disc_loss_fake

    gen_gradients = gen_tape.gradient(gen_loss, generator.trainable_variables)
    disc_gradients = disc_tape.gradient(disc_loss, discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gen_gradients, generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(disc_gradients, discriminator.trainable_variables))

    return gen_loss, disc_loss

def train(dataset, epochs=200):
    for epoch in range(epochs):
        for image_batch in dataset:
            for _ in range(2): # Train the generator twice
                gen_loss, disc_loss = train_step(image_batch)

            gen_losses.append(gen_loss.numpy())
            disc_losses.append(disc_loss.numpy())

        print(f"Epoch {epoch+1}, Generator Loss: {gen_loss:.4f}, Discriminator Loss: {disc_loss:.4f}")

# Start training, running for 200 epochs
train(monet_ds, epochs=200)
```

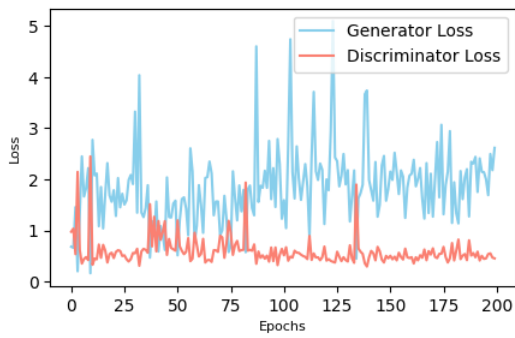
Epoch 1, Generator Loss: 0.6827, Discriminator Loss: 0.9757
Epoch 2, Generator Loss: 0.6623, Discriminator Loss: 1.0363
Epoch 3, Generator Loss: 1.4577, Discriminator Loss: 0.5377
Epoch 4, Generator Loss: 0.2031, Discriminator Loss: 2.1459
Epoch 5, Generator Loss: 1.3261, Discriminator Loss: 0.6248
Epoch 6, Generator Loss: 2.4488, Discriminator Loss: 0.3542
Epoch 7, Generator Loss: 1.6655, Discriminator Loss: 0.4439
Epoch 8, Generator Loss: 1.8324, Discriminator Loss: 0.4815
Epoch 9, Generator Loss: 2.2200, Discriminator Loss: 0.4212
Epoch 10, Generator Loss: 0.1651, Discriminator Loss: 2.4482
Epoch 11, Generator Loss: 2.7721, Discriminator Loss: 0.3297
Epoch 12, Generator Loss: 2.0771, Discriminator Loss: 0.4515
Epoch 13, Generator Loss: 2.1134, Discriminator Loss: 0.4365
Epoch 14, Generator Loss: 1.0840, Discriminator Loss: 0.7255
Epoch 15, Generator Loss: 1.8505, Discriminator Loss: 0.4670
Epoch 16, Generator Loss: 1.0408, Discriminator Loss: 0.7201
Epoch 17, Generator Loss: 1.7154, Discriminator Loss: 0.6063
Epoch 18, Generator Loss: 2.3176, Discriminator Loss: 0.3749
Epoch 19, Generator Loss: 1.6882, Discriminator Loss: 0.5381
Epoch 20, Generator Loss: 1.5619, Discriminator Loss: 0.5664
Epoch 21, Generator Loss: 1.7873, Discriminator Loss: 0.4617
Epoch 22, Generator Loss: 1.2843, Discriminator Loss: 0.5815
Epoch 23, Generator Loss: 2.0251, Discriminator Loss: 0.6153
Epoch 24, Generator Loss: 1.4539, Discriminator Loss: 0.6018
Epoch 25, Generator Loss: 1.7191, Discriminator Loss: 0.5013
Epoch 26, Generator Loss: 1.5078, Discriminator Loss: 0.5149
Epoch 27, Generator Loss: 1.5916, Discriminator Loss: 0.4389
Epoch 28, Generator Loss: 1.9870, Discriminator Loss: 0.3940
Epoch 29, Generator Loss: 2.0866, Discriminator Loss: 0.4415
Epoch 30, Generator Loss: 1.9087, Discriminator Loss: 0.5602
Epoch 31, Generator Loss: 3.3239, Discriminator Loss: 0.5551
Epoch 32, Generator Loss: 1.3482, Discriminator Loss: 0.6425
Epoch 33, Generator Loss: 4.0356, Discriminator Loss: 0.3110
Epoch 34, Generator Loss: 1.3510, Discriminator Loss: 0.5915
Epoch 35, Generator Loss: 1.2620, Discriminator Loss: 0.6412
Epoch 36, Generator Loss: 1.4400, Discriminator Loss: 0.6221
Epoch 37, Generator Loss: 1.8863, Discriminator Loss: 0.5645
Epoch 38, Generator Loss: 0.4696, Discriminator Loss: 1.5147
Epoch 39, Generator Loss: 1.1835, Discriminator Loss: 0.6891
Epoch 40, Generator Loss: 0.8923, Discriminator Loss: 1.2686
Epoch 41, Generator Loss: 1.5678, Discriminator Loss: 0.5922
Epoch 42, Generator Loss: 0.5890, Discriminator Loss: 1.1865
Epoch 43, Generator Loss: 1.0874, Discriminator Loss: 0.8185
Epoch 44, Generator Loss: 0.9198, Discriminator Loss: 0.9631
Epoch 45, Generator Loss: 0.6206, Discriminator Loss: 1.1833
Epoch 46, Generator Loss: 2.0380, Discriminator Loss: 0.5234
Epoch 47, Generator Loss: 1.2623, Discriminator Loss: 0.8362
Epoch 48, Generator Loss: 1.2543, Discriminator Loss: 0.6588
Epoch 49, Generator Loss: 1.5270, Discriminator Loss: 0.6332
Epoch 50, Generator Loss: 1.5834, Discriminator Loss: 0.6057
Epoch 51, Generator Loss: 0.5149, Discriminator Loss: 1.1981
Epoch 52, Generator Loss: 1.1970, Discriminator Loss: 0.6914
Epoch 53, Generator Loss: 1.6140, Discriminator Loss: 0.6041
Epoch 54, Generator Loss: 1.6486, Discriminator Loss: 0.5373
Epoch 55, Generator Loss: 1.4277, Discriminator Loss: 0.5842
Epoch 56, Generator Loss: 0.9069, Discriminator Loss: 0.8513
Epoch 57, Generator Loss: 2.6067, Discriminator Loss: 0.4009
Epoch 58, Generator Loss: 2.1521, Discriminator Loss: 0.4479
Epoch 59, Generator Loss: 0.7734, Discriminator Loss: 0.9564
Epoch 60, Generator Loss: 1.0910, Discriminator Loss: 0.7205
Epoch 61, Generator Loss: 1.8875, Discriminator Loss: 0.4930
Epoch 62, Generator Loss: 1.5032, Discriminator Loss: 0.5415
Epoch 63, Generator Loss: 0.9552, Discriminator Loss: 0.8374
Epoch 64, Generator Loss: 2.0259, Discriminator Loss: 0.3746
Epoch 65, Generator Loss: 2.0451, Discriminator Loss: 0.4219
Epoch 66, Generator Loss: 2.3485, Discriminator Loss: 0.4259
Epoch 67, Generator Loss: 2.1163, Discriminator Loss: 0.3863
Epoch 68, Generator Loss: 1.2979, Discriminator Loss: 0.6156
Epoch 69, Generator Loss: 1.5673, Discriminator Loss: 0.6132
Epoch 70, Generator Loss: 1.5672, Discriminator Loss: 0.5829
Epoch 71, Generator Loss: 0.9105, Discriminator Loss: 0.8975
Epoch 72, Generator Loss: 1.0018, Discriminator Loss: 0.8684
Epoch 73, Generator Loss: 1.6787, Discriminator Loss: 0.4665
Epoch 74, Generator Loss: 1.2229, Discriminator Loss: 0.7459
Epoch 75, Generator Loss: 0.5783, Discriminator Loss: 1.1899
Epoch 76, Generator Loss: 0.7020, Discriminator Loss: 0.9589
Epoch 77, Generator Loss: 2.1575, Discriminator Loss: 0.5242
Epoch 78, Generator Loss: 1.6365, Discriminator Loss: 0.7203
Epoch 79, Generator Loss: 1.1979, Discriminator Loss: 0.7977
Epoch 80, Generator Loss: 1.8816, Discriminator Loss: 0.5992
Epoch 81, Generator Loss: 1.3747, Discriminator Loss: 0.6164
Epoch 82, Generator Loss: 1.7945, Discriminator Loss: 0.6232
Epoch 83, Generator Loss: 0.5756, Discriminator Loss: 1.9377
Epoch 84, Generator Loss: 1.8048, Discriminator Loss: 0.6054
Epoch 85, Generator Loss: 1.8798, Discriminator Loss: 0.6228
Epoch 86, Generator Loss: 1.4375, Discriminator Loss: 0.6044
Epoch 87, Generator Loss: 1.2973, Discriminator Loss: 0.7289
Epoch 88, Generator Loss: 4.5997, Discriminator Loss: 0.3502
Epoch 89, Generator Loss: 1.5592, Discriminator Loss: 0.5891
Epoch 90, Generator Loss: 1.8747, Discriminator Loss: 0.4662
Epoch 91, Generator Loss: 1.8348, Discriminator Loss: 0.5148
Epoch 92, Generator Loss: 2.1685, Discriminator Loss: 0.4494
Epoch 93, Generator Loss: 1.8344, Discriminator Loss: 0.5088
Epoch 94, Generator Loss: 2.7520, Discriminator Loss: 0.3912
Epoch 95, Generator Loss: 1.6130, Discriminator Loss: 0.6675
Epoch 96, Generator Loss: 2.2194, Discriminator Loss: 0.4019
Epoch 97, Generator Loss: 1.4606, Discriminator Loss: 0.6682
Epoch 98, Generator Loss: 2.7884, Discriminator Loss: 0.3235
Epoch 99, Generator Loss: 2.2937, Discriminator Loss: 0.5247
Epoch 100, Generator Loss: 1.2300, Discriminator Loss: 0.6415
Epoch 101, Generator Loss: 1.5931, Discriminator Loss: 0.5153
Epoch 102, Generator Loss: 1.0470, Discriminator Loss: 0.6726
Epoch 103, Generator Loss: 2.6861, Discriminator Loss: 0.4094
Epoch 104, Generator Loss: 4.7390, Discriminator Loss: 0.4898
Epoch 105, Generator Loss: 2.1437, Discriminator Loss: 0.4703
Epoch 106, Generator Loss: 1.6257, Discriminator Loss: 0.6055
Epoch 107, Generator Loss: 2.6448, Discriminator Loss: 0.5671
Epoch 108, Generator Loss: 2.0128, Discriminator Loss: 0.5622
Epoch 109, Generator Loss: 1.5983, Discriminator Loss: 0.5282
Epoch 110, Generator Loss: 1.9452, Discriminator Loss: 0.5013

| | | | | |
|------------|-----------------|---------|---------------------|--------|
| Epoch 111, | Generator Loss: | 2.2988 | Discriminator Loss: | 0.4843 |
| Epoch 112, | Generator Loss: | 1.8595, | Discriminator Loss: | 0.4412 |
| Epoch 113, | Generator Loss: | 0.8689, | Discriminator Loss: | 0.8939 |
| Epoch 114, | Generator Loss: | 2.4595, | Discriminator Loss: | 0.4239 |
| Epoch 115, | Generator Loss: | 3.7107, | Discriminator Loss: | 0.5516 |
| Epoch 116, | Generator Loss: | 2.1611, | Discriminator Loss: | 0.4735 |
| Epoch 117, | Generator Loss: | 1.9878, | Discriminator Loss: | 0.4766 |
| Epoch 118, | Generator Loss: | 2.3092, | Discriminator Loss: | 0.4188 |
| Epoch 119, | Generator Loss: | 2.1890, | Discriminator Loss: | 0.4790 |
| Epoch 120, | Generator Loss: | 1.1262, | Discriminator Loss: | 0.6871 |
| Epoch 121, | Generator Loss: | 1.9988, | Discriminator Loss: | 0.4119 |
| Epoch 122, | Generator Loss: | 1.7880, | Discriminator Loss: | 0.4409 |
| Epoch 123, | Generator Loss: | 2.9325, | Discriminator Loss: | 0.4065 |
| Epoch 124, | Generator Loss: | 5.0936, | Discriminator Loss: | 0.3882 |
| Epoch 125, | Generator Loss: | 2.4274, | Discriminator Loss: | 0.3739 |
| Epoch 126, | Generator Loss: | 2.3564, | Discriminator Loss: | 0.5858 |
| Epoch 127, | Generator Loss: | 1.8634, | Discriminator Loss: | 0.4889 |
| Epoch 128, | Generator Loss: | 2.0673, | Discriminator Loss: | 0.4020 |
| Epoch 129, | Generator Loss: | 2.4960, | Discriminator Loss: | 0.4782 |
| Epoch 130, | Generator Loss: | 1.9019, | Discriminator Loss: | 0.4220 |
| Epoch 131, | Generator Loss: | 2.1790, | Discriminator Loss: | 0.3943 |
| Epoch 132, | Generator Loss: | 1.1766, | Discriminator Loss: | 0.7112 |
| Epoch 133, | Generator Loss: | 1.6658, | Discriminator Loss: | 0.4521 |
| Epoch 134, | Generator Loss: | 2.1741, | Discriminator Loss: | 0.3688 |
| Epoch 135, | Generator Loss: | 0.4493, | Discriminator Loss: | 1.8980 |
| Epoch 136, | Generator Loss: | 1.6306, | Discriminator Loss: | 0.6480 |
| Epoch 137, | Generator Loss: | 1.7307, | Discriminator Loss: | 0.5864 |
| Epoch 138, | Generator Loss: | 1.8941, | Discriminator Loss: | 0.5434 |
| Epoch 139, | Generator Loss: | 3.6632, | Discriminator Loss: | 0.3593 |
| Epoch 140, | Generator Loss: | 3.7379, | Discriminator Loss: | 0.2957 |
| Epoch 141, | Generator Loss: | 1.9870, | Discriminator Loss: | 0.4773 |
| Epoch 142, | Generator Loss: | 1.7827, | Discriminator Loss: | 0.6018 |
| Epoch 143, | Generator Loss: | 1.5838, | Discriminator Loss: | 0.5561 |
| Epoch 144, | Generator Loss: | 2.3393, | Discriminator Loss: | 0.4161 |
| Epoch 145, | Generator Loss: | 1.2692, | Discriminator Loss: | 0.6801 |
| Epoch 146, | Generator Loss: | 1.5004, | Discriminator Loss: | 0.5966 |
| Epoch 147, | Generator Loss: | 2.2747, | Discriminator Loss: | 0.4318 |
| Epoch 148, | Generator Loss: | 2.4583, | Discriminator Loss: | 0.3752 |
| Epoch 149, | Generator Loss: | 1.5829, | Discriminator Loss: | 0.5045 |
| Epoch 150, | Generator Loss: | 1.8118, | Discriminator Loss: | 0.4567 |
| Epoch 151, | Generator Loss: | 2.1526, | Discriminator Loss: | 0.5066 |
| Epoch 152, | Generator Loss: | 1.9900, | Discriminator Loss: | 0.4369 |
| Epoch 153, | Generator Loss: | 2.5198, | Discriminator Loss: | 0.4883 |
| Epoch 154, | Generator Loss: | 2.1415, | Discriminator Loss: | 0.4162 |
| Epoch 155, | Generator Loss: | 1.7125, | Discriminator Loss: | 0.5655 |
| Epoch 156, | Generator Loss: | 2.1750, | Discriminator Loss: | 0.4819 |
| Epoch 157, | Generator Loss: | 2.0510, | Discriminator Loss: | 0.4330 |
| Epoch 158, | Generator Loss: | 1.2465, | Discriminator Loss: | 0.6459 |
| Epoch 159, | Generator Loss: | 1.8413, | Discriminator Loss: | 0.4769 |
| Epoch 160, | Generator Loss: | 2.0723, | Discriminator Loss: | 0.4614 |
| Epoch 161, | Generator Loss: | 2.1114, | Discriminator Loss: | 0.4792 |
| Epoch 162, | Generator Loss: | 2.3824, | Discriminator Loss: | 0.3555 |
| Epoch 163, | Generator Loss: | 1.8640, | Discriminator Loss: | 0.4773 |
| Epoch 164, | Generator Loss: | 2.0098, | Discriminator Loss: | 0.4169 |
| Epoch 165, | Generator Loss: | 2.2214, | Discriminator Loss: | 0.5170 |
| Epoch 166, | Generator Loss: | 1.6243, | Discriminator Loss: | 0.4670 |
| Epoch 167, | Generator Loss: | 1.2333, | Discriminator Loss: | 0.6294 |
| Epoch 168, | Generator Loss: | 2.3173, | Discriminator Loss: | 0.4184 |
| Epoch 169, | Generator Loss: | 1.3418, | Discriminator Loss: | 0.5987 |
| Epoch 170, | Generator Loss: | 2.1107, | Discriminator Loss: | 0.5295 |
| Epoch 171, | Generator Loss: | 1.2707, | Discriminator Loss: | 0.6521 |
| Epoch 172, | Generator Loss: | 2.0150, | Discriminator Loss: | 0.4836 |
| Epoch 173, | Generator Loss: | 2.7290, | Discriminator Loss: | 0.4604 |
| Epoch 174, | Generator Loss: | 1.6488, | Discriminator Loss: | 0.5433 |
| Epoch 175, | Generator Loss: | 3.0670, | Discriminator Loss: | 0.5463 |
| Epoch 176, | Generator Loss: | 1.3180, | Discriminator Loss: | 0.6798 |
| Epoch 177, | Generator Loss: | 1.8847, | Discriminator Loss: | 0.4830 |
| Epoch 178, | Generator Loss: | 2.1516, | Discriminator Loss: | 0.3748 |
| Epoch 179, | Generator Loss: | 2.9440, | Discriminator Loss: | 0.4564 |
| Epoch 180, | Generator Loss: | 1.1473, | Discriminator Loss: | 0.7546 |
| Epoch 181, | Generator Loss: | 1.9428, | Discriminator Loss: | 0.4146 |
| Epoch 182, | Generator Loss: | 1.3361, | Discriminator Loss: | 0.6063 |
| Epoch 183, | Generator Loss: | 1.1388, | Discriminator Loss: | 0.8237 |
| Epoch 184, | Generator Loss: | 2.1624, | Discriminator Loss: | 0.4406 |
| Epoch 185, | Generator Loss: | 1.6137, | Discriminator Loss: | 0.5086 |
| Epoch 186, | Generator Loss: | 2.0343, | Discriminator Loss: | 0.5398 |
| Epoch 187, | Generator Loss: | 2.1990, | Discriminator Loss: | 0.4198 |
| Epoch 188, | Generator Loss: | 1.2744, | Discriminator Loss: | 0.8091 |
| Epoch 189, | Generator Loss: | 2.3434, | Discriminator Loss: | 0.5103 |
| Epoch 190, | Generator Loss: | 2.3104, | Discriminator Loss: | 0.5580 |
| Epoch 191, | Generator Loss: | 2.4409, | Discriminator Loss: | 0.4778 |
| Epoch 192, | Generator Loss: | 1.7530, | Discriminator Loss: | 0.5868 |
| Epoch 193, | Generator Loss: | 2.4102, | Discriminator Loss: | 0.4136 |
| Epoch 194, | Generator Loss: | 2.1466, | Discriminator Loss: | 0.5052 |
| Epoch 195, | Generator Loss: | 2.1394, | Discriminator Loss: | 0.4460 |
| Epoch 196, | Generator Loss: | 1.9988, | Discriminator Loss: | 0.4549 |
| Epoch 197, | Generator Loss: | 1.6921, | Discriminator Loss: | 0.5398 |
| Epoch 198, | Generator Loss: | 2.4960, | Discriminator Loss: | 0.5534 |
| Epoch 199, | Generator Loss: | 2.1795, | Discriminator Loss: | 0.4702 |
| Epoch 200, | Generator Loss: | 2.6159, | Discriminator Loss: | 0.4551 |

3.2.3 Visualize Results

```
In [16]: # Visualize Loss
plt.figure(figsize=(5, 3))
plt.plot(gen_losses, label='Generator Loss', color='skyblue')
plt.plot(disc_losses, label='Discriminator Loss', color='salmon')
plt.xlabel('Epochs', fontsize=8)
plt.ylabel('Loss', fontsize=8)
plt.title('\nTraining Loss\n', fontsize=12)
plt.legend()
plt.show()
```

Training Loss

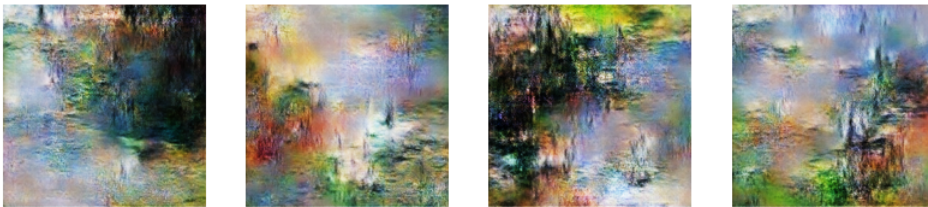


```
In [17]: # Generate and display images
def generate_and_show_images(generator, num_images=4):
    noise = tf.random.normal([num_images, latent_dim])
    generated_images = generator(noise, training=False)

    plt.figure(figsize=(10, 3))
    for i in range(num_images):
        plt.subplot(1, num_images, i + 1)
        plt.imshow((generated_images[i].numpy() + 1) / 2)
        plt.axis('off')
    plt.suptitle("\nGenerated Monet-style Images\n", fontsize=12)
    plt.show()

generate_and_show_images(generator)
```

Generated Monet-style Images



3.3 Submit Images

- Generate 7,000 Monet-style images using the **generator**.
- Save the images as **JPG** files in the output directory.
- **Compress** the images into a **ZIP** archive.
- **Clean up temporary files** by removing the generated image folder.
- **Final output** consists of the **ZIP** file containing all the generated images.

```
In [21]: from tensorflow.keras.preprocessing.image import array_to_img
import zipfile
import shutil
```

```
In [22]: # Define paths
output_dir = "/kaggle/working/generated_images"
zip_file_path = "/kaggle/working/images.zip"

# Ensure output directory exists
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# Number of images to generate
num_images = 7000
latent_dim = 128

# Function to generate images without printing progress
def generate_and_save_images(generator, num_images, output_dir):
    for i in range(num_images):
        noise = tf.random.normal([1, latent_dim]) # Generate random noise
        generated_image = generator(noise, training=False) # Generate image

        # Convert from [-1, 1] to [0, 255]
        img = (generated_image[0].numpy() + 1) * 127.5
        img = img.astype(np.uint8)

        # Save image as JPG
        img_path = os.path.join(output_dir, f"monet_{i+1}.jpg")
        array_to_img(img).save(img_path)

    print("All images generated successfully.")

# Generate and save images
generate_and_save_images(generator, num_images, output_dir)



# Compress images to a zip file without printing progress
def zip_images(output_dir, zip_file_path):
    with zipfile.ZipFile(zip_file_path, 'w') as zipf:
        for root, dirs, files in os.walk(output_dir):
            for file in files:
                zipf.write(os.path.join(root, file), arcname=file)
```

```
# Create ZIP archive
zip_images(output_dir, zip_file_path)

# Clean up by removing the generated images folder
shutil.rmtree(output_dir)

print("Image generation and compression complete. The only output is 'images.zip'.")
```

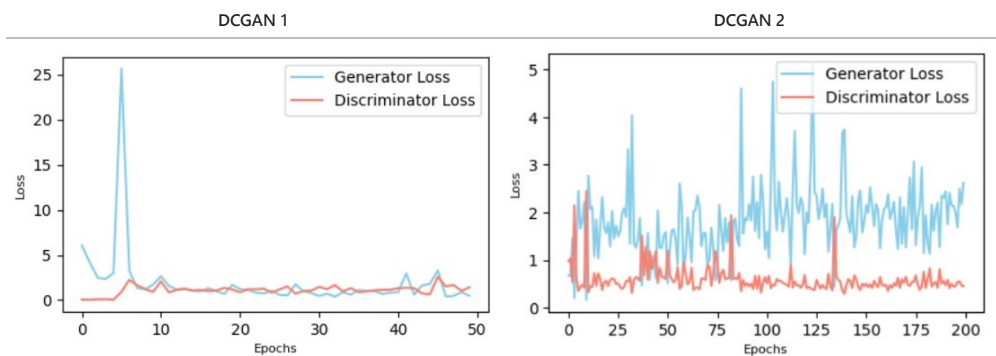
All images generated successfully.
Image generation and compression complete. The only output is 'images.zip'.

| Submission and Description | Public Score  |
|---|--|
| <div> w5_kaggle - Version 2</div> <div>Succeeded · 4m ago</div> | 121.23453 |

Step 4 Results and Analysis

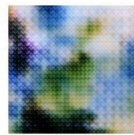
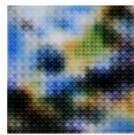
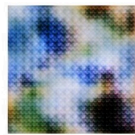
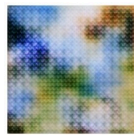
4.1 Visualization Analysis

- Loss Trend:
 - DCGAN 1 (epochs=50):** The training loss analysis over 50 epochs shows that the generator initially struggled, with a significant spike at **epoch 6**, but both generator and discriminator losses **gradually stabilized** afterward, indicating improved balance. The generator loss fluctuated but did not collapse to zero, suggesting that mode collapse was likely avoided, while the discriminator maintained a loss around 1, meaning it remained effective in distinguishing real from fake images. By epoch 50, the generator loss decreased to **0.4863**, and the discriminator loss was **1.4144**, showing a slight advantage for the discriminator. The **mean generator loss of 1.9031** indicates ongoing learning, but further improvements are needed to produce more realistic images. A **mean discriminator loss of 1.1156** suggests balanced competition, as values around **1.0** are ideal for training stability. To optimize performance, fine-tuning learning rates and training frequency may be required. Overall, the training is progressing well, but further refinements and qualitative evaluations are needed to ensure high-quality results.
 - DCGAN 2 (epochs=200):** The training loss analysis over 200 epochs indicates that the generator loss fluctuated, with several peaks and valleys, suggesting challenges in learning but overall improvement. The **mean generator loss of 1.8341** shows that the generator is performing better compared to earlier stages, but still requires further optimization to generate high-quality images consistently. Meanwhile, the **discriminator loss averaged at 0.7039**, indicating that it is becoming increasingly confident in distinguishing real from generated images, which might lead to challenges for the generator in future training. The training process shows that while the generator continues to learn, the discriminator is becoming too strong, which could hinder generator performance. A lower generator loss is ideal, but if the discriminator loss drops too much, it may indicate that it is overpowering the generator. Overall, the model shows promising progress, but maintaining an optimal balance between the two networks remains crucial to achieving high-quality results.



- Images Quality:
 - DCGAN 1 (epochs=50):**
 - Lack of Details:** The images are **blurry**, indicating the generator hasn't yet learned Monet's fine-grained features. More training is needed to capture intricate textures.
 - Color Distribution:** While the colors resemble Monet's style, they are **overly blended** without clear separation, requiring further refinement for better contrast and realism.
 - Repetitive Patterns:** **Grid-like artifacts** suggest issues such as insufficient data variation, architectural limitations, or imbalanced loss between the generator and discriminator.
 - Mode Collapse:** The images appear **similar**, indicating mode collapse, where the generator lacks diversity. More training and learning rate adjustments could help.
 - DCGAN 2 (epochs=200):**
 - Improved Details:** The images now have **finer textures** and **better-defined elements**, though some areas remain **blurry**, requiring further training.
 - Better Color Distribution:** Colors are more **vibrant** and **well-separated**, capturing Monet's style more effectively, but some areas are still **overly smooth**.
 - Reduced Artifacts:** The earlier grid-like artifacts are largely gone, with more **natural patterns**, though occasional inconsistencies remain.
 - Increased Diversity:** The outputs show **greater variation**, reducing mode collapse, but further structural refinement is needed.

DCGAN 1



DCGAN 2



4.2 Comparison Table

| Model | Epoch | Generator Mean Loss | Discriminator Mean Loss | Generated Image | | | |
|---------|-------|---------------------|-------------------------|-----------------|--|--|--|
| DCGAN 1 | 50 | 1.9031 | 1.1156 | | | | |
| DCGAN 2 | 200 | 1.8341 | 0.7039 | | | | |

Step 5 Conclusion

5.1 Takeaways

- **DCGAN 2 performed better**, producing images that are closer to real Monet-style paintings.
- A **lower generator loss** in DCGAN 2 indicates improved image quality, while a **lower discriminator loss** suggests it became better at distinguishing fake from real images.
- Longer training (200 epochs) resulted in **smoother textures, better color distribution, and reduced artifacts**, compared to the 50-epoch model.

5.2 Improvements

- **What Helped:** Increasing the training epochs to 200 helped the generator capture finer details and better color variations, leading to improved image quality. The balanced losses ensured **better convergence** and **diverse outputs**, while fine-tuning hyperparameters like the **learning rate** and **batch size** made training more stable.
- **What Did Not Help:** Training for only **50 epochs** was insufficient for learning complex features, leading to blurry and repetitive patterns. The low discriminator loss in DCGAN 2 suggests the generator may struggle against an overly confident discriminator, and remaining artifacts indicate a need for further architecture optimization to enhance texture realism.
- **Suggestions for Improvement:** Extending training to **more than 200 epochs**, adjusting the **learning rate scheduling**, using **data augmentation**, and applying **regularization techniques** like dropout can further improve image quality and model generalization.

Overall, DCGAN 2 demonstrated significant improvements over DCGAN 1, producing images that are closer to real Monet-style paintings. However, further fine-tuning and longer training may be necessary to achieve even higher realism and diversity in generated outputs. Balancing the generator and discriminator through careful parameter adjustments will be key to achieving optimal results.

GitHub Repository Link

https://github.com/d93xup60126/GAN_Monet_Paintings