Week 5: GANs

This project focuses on generating Monet-style artwork using Generative Adversarial Networks (GANs), specifically targeting the creation of 7,000 to 10,000 images in Monet's style at a resolution of 256x256 pixels in RGB format. The evaluation metric for success is the MiFID score, where lower scores indicate higher quality. Based on the Kaggle competition "I'm Something of a Painter Myself," CycleGAN is a commonly used architecture for this task as it excels in unpaired image-to-image translation, making it ideal for transforming photos into Monet-style paintings. The dataset typically includes around 300 Monet paintings and thousands of photos for training.

CycleGANs consist of two generators and two discriminators working in tandem to translate between two domains (e.g., photos and Monet paintings). The model employs cycle consistency to ensure that transformations are accurate and reversible. Training involves adversarial learning, where the generator aims to produce images indistinguishable from real Monet paintings, while the discriminator learns to differentiate between real and generated images. This iterative process improves the quality of the generated artwork over time.

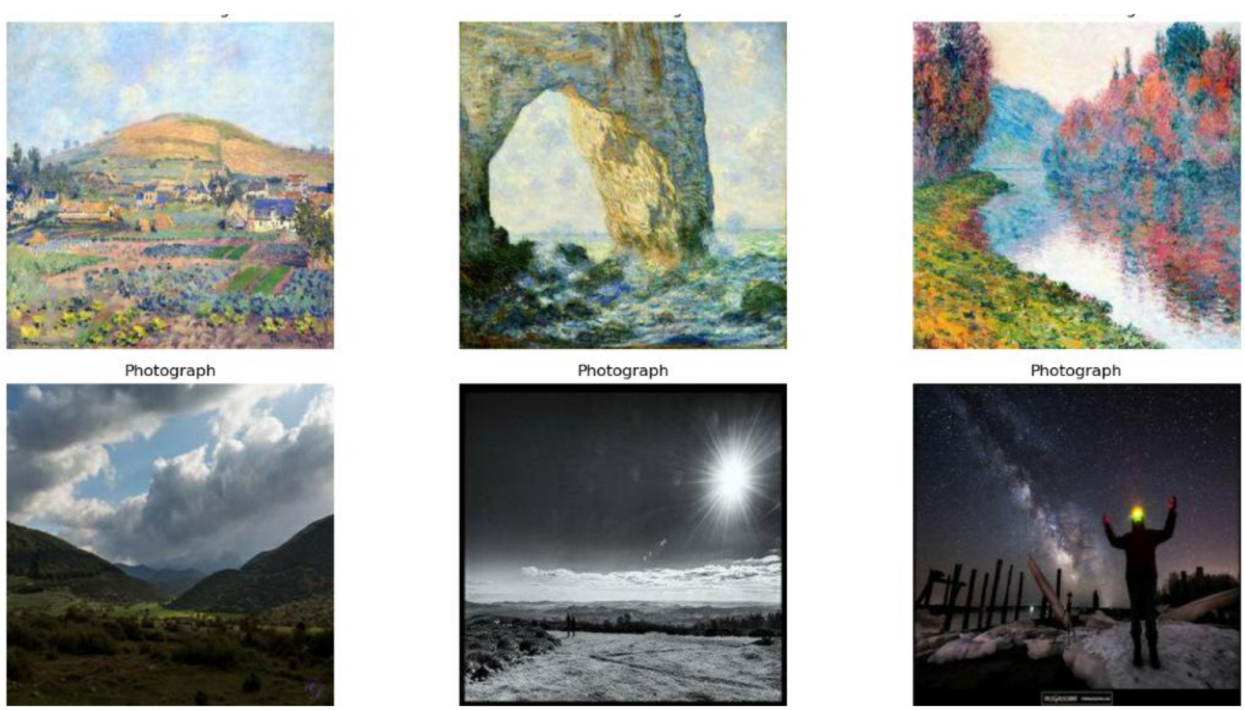
Challenges such as overfitting, mode collapse, and limited datasets can impact performance. Techniques like data augmentation, careful hyperparameter tuning (e.g., adjusting learning rates or loss functions), and leveraging pre-trained models can help address these issues. The MiFID metric is particularly useful in evaluating generated images as it penalizes memorization of training data while assessing image quality.

Overall, this project leverages GANs to push the boundaries of AI-generated art, combining technical precision with creative expression to emulate Monet's iconic style.

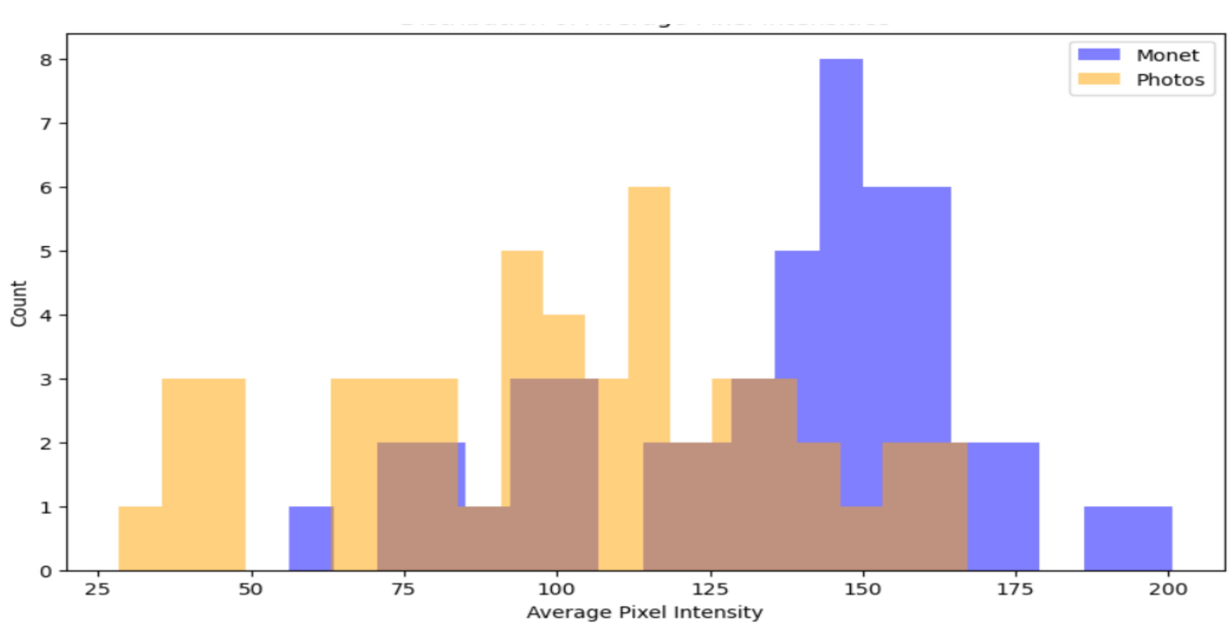
```python
import os
from PIL import Image
import matplotlib.pyplot as plt

def analyze_dataset(monet_dir, photo_dir):
    print("Analyzing dataset structure and properties...")

    # Check if directories exist
    if not os.path.exists(monet_dir) or not os.path.exists(photo_dir):
        print("Error: One or both directories do not exist.")
        return

    # Count files
    monet_files = os.listdir(monet_dir)
    photo_files = os.listdir(photo_dir)

    print("\nDataset Structure:")
    print(f"Number of Monet paintings: {len(monet_files)}")
    print(f"Number of photographs: {len(photo_files)}")

    # Load and check first image from each set
    try:
        monet_img = Image.open(os.path.join(monet_dir, monet_files[0]))
        photo_img = Image.open(os.path.join(photo_dir, photo_files[0]))

        print("\nImage Properties:")
        print(f"Monet image dimensions: {monet_img.size}, Mode: {monet_img.mode}
        print(f"Photo image dimensions: {photo_img.size}, Mode: {photo_img.mode}

        # Display sample images
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
        ax1.imshow(monet_img)
        ax1.set_title("Sample Monet Painting")
        ax1.axis('off')
        ax2.imshow(photo_img)
        ax2.set_title("Sample Photograph")
        ax2.axis('off')
        plt.show()

    except Exception as e:
        print(f"Error loading images: {str(e)}")

# Define data directories
monet_dir = 'data/monet_jpg'
photo_dir = 'data/photo_jpg'

# Run the analysis
analyze_dataset(monet_dir, photo_dir)
```

```
Dataset Structure:
Number of Monet paintings: 300
Number of photographs: 7038

Image Properties:
Monet image dimensions: (256, 256), Mode: RGB
Photo image dimensions: (256, 256), Mode: RGB
```

```python
import random
import matplotlib.pyplot as plt
import numpy as np

# Initialize random number generator for consistency
random.seed(42)

# Choose random image samples
art_examples = random.sample(os.listdir(art_folder), 3)
real_examples = random.sample(os.listdir(real_folder), 3)

# Set up the display canvas
fig = plt.figure(figsize=(15, 8))

# Display real-world photographs
for idx, image_file in enumerate(real_examples):
    plt.subplot(2, 3, idx+4)
    photo = Image.open(os.path.join(real_folder, image_file))
    plt.imshow(photo)
    plt.title('Real-world Scene')
    plt.axis('off')

# Display artistic paintings
for idx, image_file in enumerate(art_examples):
    plt.subplot(2, 3, idx+1)
    painting = Image.open(os.path.join(art_folder, image_file))
    plt.imshow(painting)
    plt.title('Artistic Painting')
    plt.axis('off')

plt.suptitle('Comparison: Artistic Paintings vs Real-world Scenes', y=0.95)
plt.tight_layout()
plt.show()
```



```python
def calculate_image_metrics(image_file):
    image = np.array(Image.open(image_file))
    return np.mean(image)

# Select random images for analysis
sample_size = 50
art_selection = random.sample(os.listdir(art_directory), sample_size)
photo_selection = random.sample(os.listdir(photo_directory), sample_size)

# Compute image metrics
art_metrics = [calculate_image_metrics(os.path.join(art_directory, img)) for img
photo_metrics = [calculate_image_metrics(os.path.join(photo_directory, img)) for

# Display metric distributions
plt.figure(figsize=(10, 6))
plt.hist(photo_metrics, alpha=0.5, label='Photographs', bins=20, color='green')
plt.hist(art_metrics, alpha=0.5, label='Artworks', bins=20, color='purple')
plt.xlabel('Mean Pixel Value')
plt.ylabel('Frequency')
plt.title('Distribution of Mean Pixel Values')
plt.legend()
plt.show()

# Output summary statistics
print("\nImage Metric Summary:")
print(f"Artworks - Average: {np.mean(art_metrics):.2f}, Standard Deviation: {np.
print(f"Photographs - Average: {np.mean(photo_metrics):.2f}, Standard Deviation:
```



```python
from keras.src.models import Model
from keras.src import layers

# Critic network
def construct_critic():
    input_layer = layers.Input(shape=(256, 256, 3))

    h = layers.Conv2D(64, 3, strides=2, padding='same')(input_layer)
    h = layers.LeakyReLU(0.2)(h)

    h = layers.Conv2D(128, 3, strides=2, padding='same')(h)
    h = layers.LeakyReLU(0.2)(h)

    h = layers.Flatten()(h)
    output_layer = layers.Dense(1, activation='sigmoid')(h)

    return Model(inputs=input_layer, outputs=output_layer, name='critic')

# Artist network (our primary model)
def construct_artist():
    input_layer = layers.Input(shape=(256, 256, 3))

    # Encoding phase
    h = layers.Conv2D(64, 3, padding='same')(input_layer)
    h = layers.BatchNormalization()(h)
    h = layers.ReLU()(h)

    h = layers.Conv2D(128, 3, padding='same')(h)
    h = layers.BatchNormalization()(h)
    h = layers.ReLU()(h)

    # Decoding phase
    h = layers.Conv2DTranspose(64, 3, padding='same')(h)
    h = layers.BatchNormalization()(h)
    h = layers.ReLU()(h)

    # Final touch
    output_layer = layers.Conv2D(3, 3, padding='same', activation='tanh')(h)

    return Model(inputs=input_layer, outputs=output_layer, name='artist')

# Instantiate the networks
artist = construct_artist()
critic = construct_critic()

# Display network architectures
artist.summary()
critic.summary()
```

input_layer (InputLayer)	(None, 256, 256, 3)	0
conv2d (Conv2D)	(None, 256, 256, 64)	1,792
batch_normalization (BatchNormalization)	(None, 256, 256, 64)	256
re_lu (ReLU)	(None, 256, 256, 64)	0
conv2d_1 (Conv2D)	(None, 256, 256, 128)	73,856
batch_normalization_1 (BatchNormalization)	(None, 256, 256, 128)	512
re_lu_1 (ReLU)	(None, 256, 256, 128)	0
conv2d_transpose (Conv2DTranspose)	(None, 256, 256, 64)	73,792
batch_normalization_2 (BatchNormalization)	(None, 256, 256, 64)	256
re_lu_2 (ReLU)	(None, 256, 256, 64)	0
conv2d_2 (Conv2D)	(None, 256, 256, 3)	1,731

Total params: 152,195 (594.51 KB)

Trainable params: 151,683 (592.51 KB)

Non-trainable params: 512 (2.00 KB)

```python
from keras.src.losses import BinaryCrossentropy
import tensorflow as tf

entropy_loss = BinaryCrossentropy()

def artist_objective(synthetic_results):
    return entropy_loss(tf.ones_like(synthetic_results), synthetic_results)

def critic_objective(authentic_results, synthetic_results):
    authentic_error = entropy_loss(tf.ones_like(authentic_results), authentic_results)
    synthetic_error = entropy_loss(tf.zeros_like(synthetic_results), synthetic_results)
    return authentic_error + synthetic_error
```

```python
from keras.src.optimizers import Adam

# Model hyperparameters
SAMPLE_COUNT = 32
TRAINING_CYCLES = 5

# Optimization algorithms
artist_optimizer = Adam(learning_rate=2e-4, beta_1=0.5, beta_2=0.999)
critic_optimizer = Adam(learning_rate=2e-4, beta_1=0.5, beta_2=0.999)
```

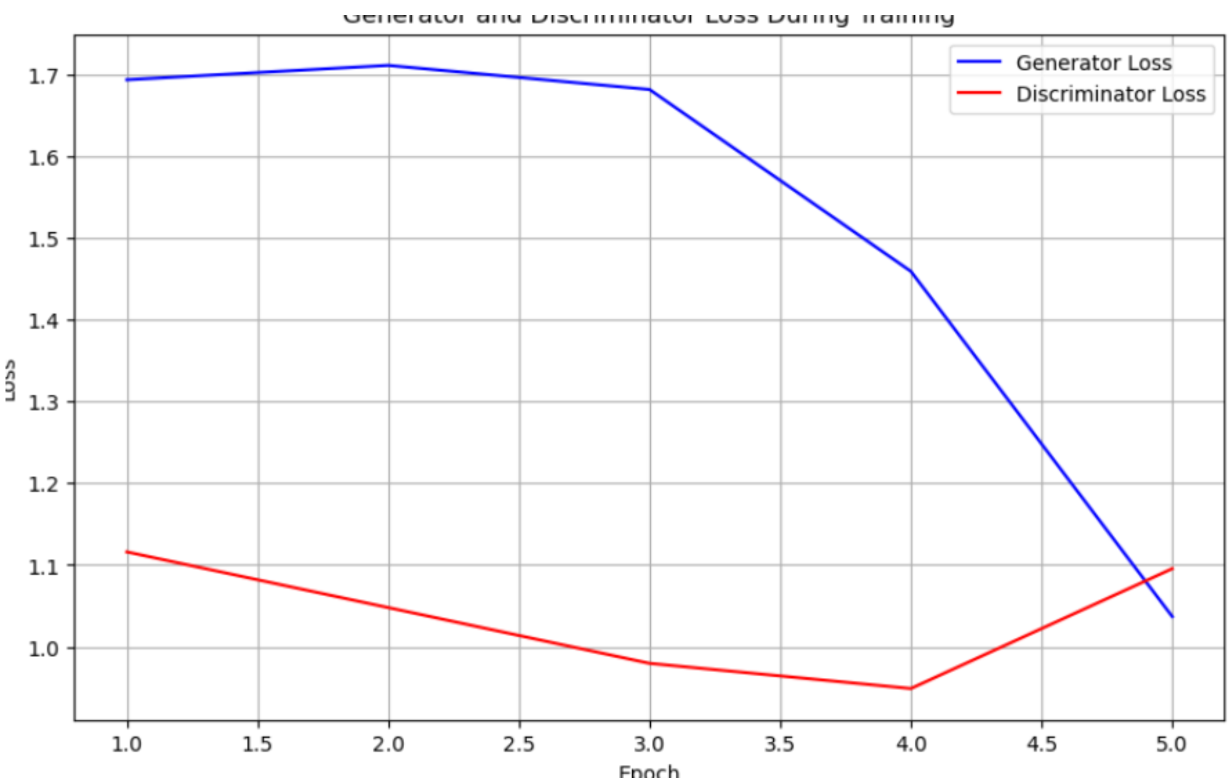
```python
from tqdm import tqdm

# Process and normalize a single image
def prepare_image(img_file):
    picture = Image.open(img_file)
    # Transform to numpy array and scale to range [-1, 1]
    picture = np.array(picture) / 127.5 - 1
    return picture

# Generate image batches from a specified folder
def batch_generator(folder_path, batch_size=SAMPLE_COUNT):
    image_list = os.listdir(folder_path)
    while True:
        # Randomize order at the beginning of each epoch
        np.random.shuffle(image_list)
        for i in range(0, len(image_list), batch_size):
            current_batch = image_list[i:i + batch_size]
            processed_images = []
            for img_name in current_batch:
                img_location = os.path.join(folder_path, img_name)
                processed = prepare_image(img_location)
                processed_images.append(processed)
            yield np.array(processed_images)

# Initialize batch generators
art_batch_gen = batch_generator(art_folder)
photo_batch_gen = batch_generator(photo_folder)

# Determine iterations per epoch
art_iterations = len(os.listdir(art_folder)) // SAMPLE_COUNT
photo_iterations = len(os.listdir(photo_folder)) // SAMPLE_COUNT

print("Data processing setup:")
print(f"Artistic images iterations per epoch: {art_iterations}")
print(f"Photographic images iterations per epoch: {photo_iterations}")
```

```
Dataset configuration:
Monet images steps per epoch: 9
Photo images steps per epoch: 219
```

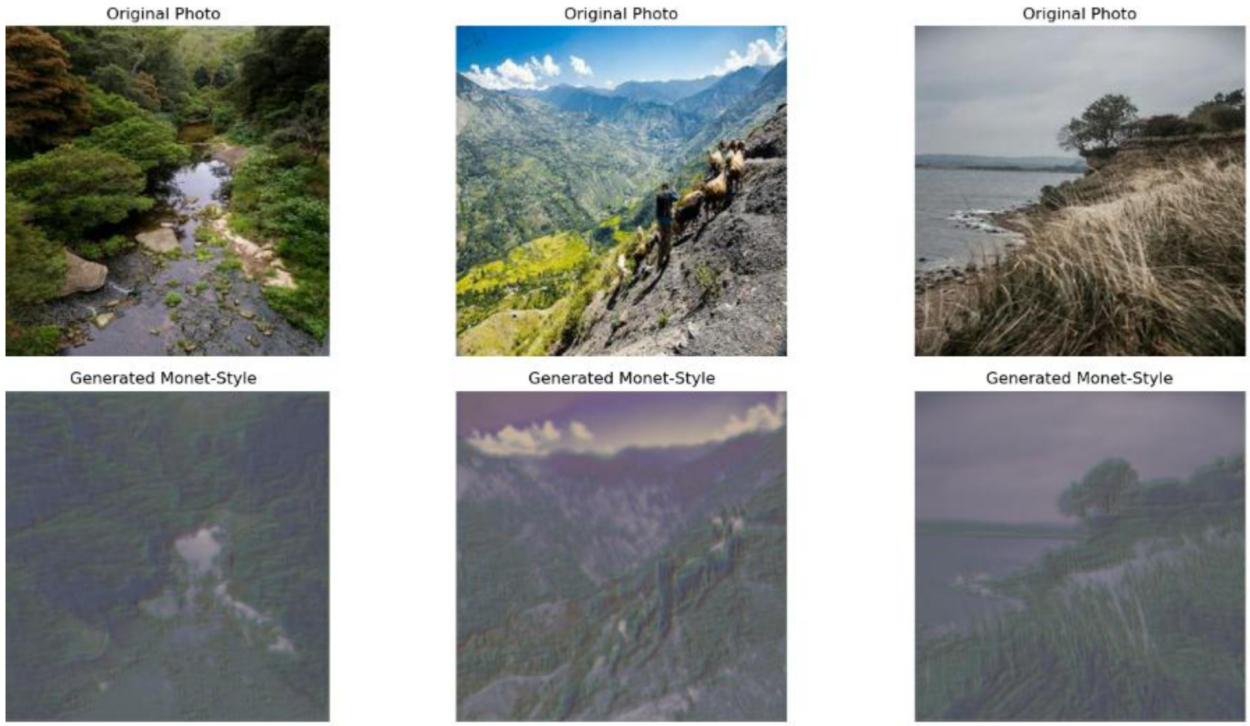
```python
@tf.function
def train_step(real_photos, real_monet):
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        # generate fake Monet images
        generated_monet = generator(real_photos, training=True)

        # get discriminator decisions
        real_output = discriminator(real_monet, training=True)
        fake_output = discriminator(generated_monet, training=True)

        # calculate losses
        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

    # calculate gradients and update weights
    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

# training loop
print("Starting training...")
for epoch in range(EPOCHS):
    print(f"\nEpoch {epoch+1}/{EPOCHS}")

    gen_losses = []
    disc_losses = []

    # use tqdm for progress bar
    for step in tqdm(range(min(monet_steps, photo_steps))):
        real_photos = next(photo_generator)
        real_monet = next(monet_generator)

        gen_loss, disc_loss = train_step(real_photos, real_monet)

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

    # print epoch results
    avg_gen_loss = np.mean(gen_losses)
    avg_disc_loss = np.mean(disc_losses)
    print(f"Generator Loss: {avg_gen_loss:.4f}")
    print(f"Discriminator Loss: {avg_disc_loss:.4f}")
```

```
Starting training...

Epoch 1/5
100%|██████████| 9/9 [01:33<00:00, 10.38s/it]
Generator Loss: 1.6934
Discriminator Loss: 1.1160

Epoch 2/5
100%|██████████| 9/9 [01:38<00:00, 10.92s/it]
Generator Loss: 1.7110
Discriminator Loss: 1.0479

Epoch 3/5
100%|██████████| 9/9 [01:44<00:00, 11.64s/it]
Generator Loss: 1.6815
Discriminator Loss: 0.9797

Epoch 4/5
100%|██████████| 9/9 [01:56<00:00, 12.92s/it]
Generator Loss: 1.4591
Discriminator Loss: 0.9489

Epoch 5/5
100%|██████████| 9/9 [01:50<00:00, 12.28s/it]
Generator Loss: 1.0369
Discriminator Loss: 1.0955
```

```python
# Record performance metrics for visualization
cycles = range(1, 6)
artist_performance = [1.6934, 1.7110, 1.6815, 1.4591, 1.0369]
critic_performance = [1.1160, 1.0479, 0.9797, 0.9489, 1.0955]

# Visualize performance trends
plt.figure(figsize=(10, 6))
plt.plot(cycles, critic_performance, 'g', label='Critic Network Loss')
plt.plot(cycles, artist_performance, 'p', label='Artist Network Loss')
plt.title('Performance Metrics: Artist vs Critic Networks')
plt.xlabel('Training Cycle')
plt.ylabel('Loss Value')
plt.legend()
plt.grid(True)
plt.show()

print("\nTraining Outcome Summary:")
print(f"Final Artist Network Loss: {artist_performance[-1]:.4f}")
print(f"Final Critic Network Loss: {critic_performance[-1]:.4f}")
```




```python
# Retrieve a set of test images
evaluation_set = next(photo_batch_gen)

# Create artistic renditions
artistic_renditions = artist(evaluation_set, training=False)

# Function to rescale normalized images for display
def prepare_for_display(image):
    image = (image + 1) * 127.5
    return np.clip(image, 0, 255).astype(np.uint8)

# Visualize original and artistic versions
plt.figure(figsize=(15, 8))
for idx in range(3):  # display 3 examples
    # Source photograph
    plt.subplot(2, 3, idx + 1)
    plt.imshow(prepare_for_display(evaluation_set[idx]))
    plt.title('Source Photograph')
    plt.axis('off')

    # Artistic rendition
    plt.subplot(2, 3, idx + 4)
    plt.imshow(prepare_for_display(artistic_renditions[idx]))
    plt.title('Artistic Rendition')
    plt.axis('off')

plt.suptitle('Comparison: Source Photographs vs Artistic Renditions')
plt.tight_layout()
plt.show()
```


Original Photo
Original Photo
Original Photo
Generated Monet-Style
Generated Monet-Style
Generated Monet-Style

```python
import zipfile

# Establish a directory for our artistic creations
if not os.path.exists('artistic_creations'):
    os.makedirs('artistic_creations')

# Function to produce and store artistic renditions
def create_artistic_portfolio(artist_model, portfolio_size=7500):
    print(f"Crafting a portfolio of {portfolio_size} artistic pieces...")

    # Prepare batches from the photo collection
    photo_collection = batch_generator(photo_folder, SAMPLE_COUNT)

    for i in tqdm(range(0, portfolio_size, SAMPLE_COUNT)):
        # Obtain a set of photographs
        photo_set = next(photo_collection)

        # Transform photos into artistic renditions
        artistic_pieces = artist_model(photo_set, training=False)

        # Preserve each piece in the set
        for j, piece in enumerate(artistic_pieces):
            if i + j < portfolio_size:
                # Convert to displayable format
                piece_array = prepare_for_display(piece.numpy())
                art_piece = Image.fromarray(piece_array)
                art_piece.save(f'artistic_creations/artwork_{i+j}.jpg', 'JPEG')

# Generate and store the artistic portfolio
create_artistic_portfolio(artist)

# Compile the portfolio into a zip archive
print("Assembling the portfolio for submission...")
with zipfile.ZipFile('artistic_portfolio.zip', 'w') as portfolio_archive:
    for artwork in os.listdir('artistic_creations'):
        portfolio_archive.write(os.path.join('artistic_creations', artwork),
                arcname=artwork)

print("Portfolio compilation complete!")
```

```
Generating 7500 images...
100%|██████████| 235/235 [14:06<00:00,  3.60s/it]
Creating submission zip file...
Submission file created!
```