```
# IMPORTANT: SOME KAGGLE DATA SOURCES ARE PRIVATE
# RUN THIS CELL IN ORDER TO IMPORT YOUR KAGGLE DATA SOURCES.
import kagglehub
kagglehub.login()
```

```
# IMPORTANT: RUN THIS CELL IN ORDER TO IMPORT YOUR KAGGLE DATA SOURCES,
# THEN FEEL FREE TO DELETE THIS CELL.
# NOTE: THIS NOTEBOOK ENVIRONMENT DIFFERS FROM KAGGLE'S PYTHON
# ENVIRONMENT SO THERE MAY BE MISSING LIBRARIES USED BY YOUR
# NOTEBOOK.

gan_getting_started_path = kagglehub.competition_download('gan-getting-started')
drllamahacer_feature_encoder_monetgan_turbo_tensorflow2_default_3_path = kagglehub.model_download('drllamahacer/feature_
drllamahacer_monet_gan_turbo_weights_tensorflow2_default_13_path = kagglehub.model_download('drllamahacer/monet_gan_turb
drllamahacer_monet_gan_turbo_weights_tensorflow2_default_14_path = kagglehub.model_download('drllamahacer/monet_gan_turb
drllamahacer_monet_gan_turbo_weights_tensorflow2_default_15_path = kagglehub.model_download('drllamahacer/monet_gan_turb
drllamahacer_monet_gan_turbo_weights_tensorflow2_default_16_path = kagglehub.model_download('drllamahacer/monet_gan_turb
drllamahacer_monet_gan_turbo_weights_tensorflow2_default_17_path = kagglehub.model_download('drllamahacer/monet_gan_turb

print('Data source import complete.')
```

I'm Something of a Painter Myself

This project focuses on using GAN to recreate the painting style of Claude Monet. By training a generative adversarial network (GAN) model, with two parts: a generator that creates new images and a discriminator that checks if the images look real. The two models compete with each other, gradually improving until the generator can make photos look like Monet's paintings. The challenge is to produce thousands of Monet-style images, utilizing computer vision and machine learning to learn about Monet's painting style and generate similar art.

The dataset includes 300 Monet paintings and 7,028 real photos, each provided in both JPEG and TFRecord formats at a size of 256x256 pixels. The Monet images are used for training the GAN to learn the painting style, while the photo set is used to generate Monet-style outputs for submission.

```
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load
import os, re, json, time, random, glob, zipfile
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list all files under the input directory
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

import os,  pathlib, sys, math, hashlib, random, io, contextlib
from collections import Counter, defaultdict
from PIL import Image, UnidentifiedImageError
```

```
import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras.preprocessing import image_dataset_from_directory
from pathlib import Path
import cv2
from tqdm import tqdm
```

```
2025-09-26 10:56:15.355435: E external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:477] Unable to register cuFFT fact
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
E0000 00:00:1758884175.612699      36 cuda_dnn.cc:8310] Unable to register cuDNN factory: Attempting to register factory
E0000 00:00:1758884175.688505      36 cuda_blas.cc:1418] Unable to register cuBLAS factory: Attempting to register facto
```

```
#Import all datasets

base = "/kaggle/input/gan-getting-started"
print("Contents:", os.listdir(base))
monet_dir = pathlib.Path(base) / "monet_jpg"
photo_dir = pathlib.Path(base) / "photo_jpg"
```

```
monet_files = list(monet_dir.glob("*.jpg"))
photo_files = list(photo_dir.glob("*.jpg"))

#Confirming imported images
print(f"Monet: {len(monet_files)} images, Photo: {len(photo_files)} images")

BATCH = 8
IMG_SIZE = (256, 256)

monet_ds = image_dataset_from_directory(
    monet_dir,
    labels=None,
    shuffle=True,
    batch_size=BATCH,
    image_size=IMG_SIZE
)

photo_ds = image_dataset_from_directory(
    photo_dir,
    labels=None,
    shuffle=True,
    batch_size=BATCH,
    image_size=IMG_SIZE
)
```

```
Contents: ['monet_jpg', 'photo_tfrec', 'photo_jpg', 'monet_tfrec']
Monet: 300 images, Photo: 7038 images
Found 300 files.
I0000 00:00:1758884190.120649      36 gpu_device.cc:2022] Created device /job:localhost/replica:0/task:0/device:GPU:0 w:
I0000 00:00:1758884190.121417      36 gpu_device.cc:2022] Created device /job:localhost/replica:0/task:0/device:GPU:1 w:
Found 7038 files.
```

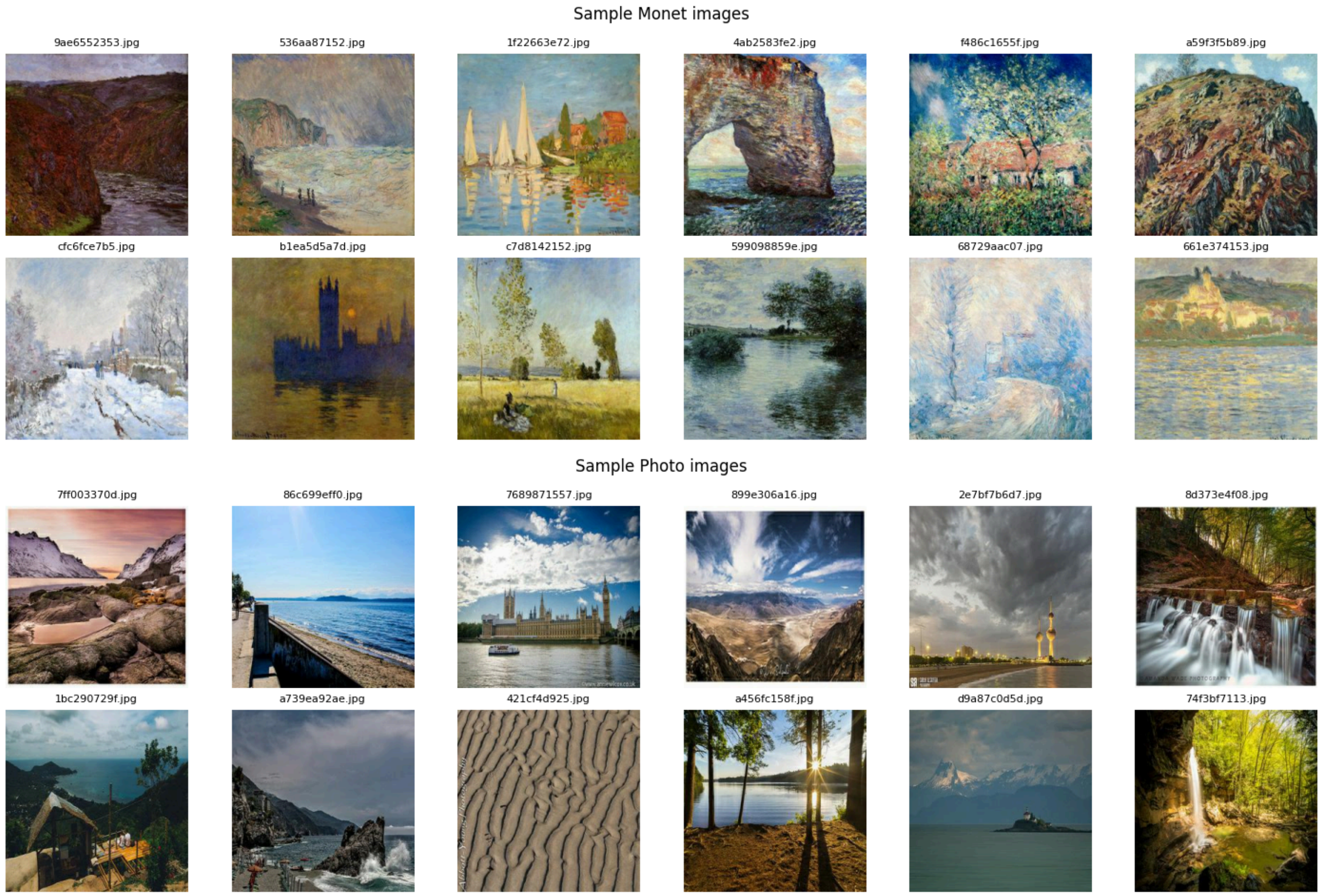
Exploratory Data Analysis

```
# sanity check
BASE = Path("/kaggle/input/gan-getting-started")
MONET_JPG  = BASE / "monet_jpg"
PHOTO_JPG  = BASE / "photo_jpg"
MONET_TFRECORD = BASE / "monet_tfrec"
PHOTO_TFRECORD = BASE / "photo_tfrec"

assert MONET_JPG.exists() or MONET_TFRECORD.exists(), "Expected Monet data not found."
assert PHOTO_JPG.exists() or PHOTO_TFRECORD.exists(), "Expected Photo data not found."

def list_images(folder, exts=(".jpg", ".jpeg", ".png")):
    return sorted([p for p in Path(folder).glob("*") if p.suffix.lower() in exts])

def quick_hash(path, chunk=65536):
    m = hashlib.md5()
    with open(path, "rb") as f:
        for buf in iter(lambda: f.read(chunk), b""):
            m.update(buf)
    return m.hexdigest()

def open_img(path):
    with Image.open(path) as im:
        return im.convert("RGB")

def im2arr(img):
    return np.asarray(img, dtype=np.uint8)

monet_imgs = list_images(MONET_JPG) if MONET_JPG.exists() else []
photo_imgs = list_images(PHOTO_JPG) if PHOTO_JPG.exists() else []

print("Folders present:")
for p in [MONET_JPG, PHOTO_JPG, MONET_TFRECORD, PHOTO_TFRECORD]:
    print(f" - {p.name:<12} | exists: {p.exists()}")

print("\nJPEG counts:")
print(f"Monet JPG : {len(monet_imgs)}")
print(f"Photo JPG : {len(photo_imgs)}")
```

```
Folders present:
 - monet_jpg    | exists: True
```

```
 - photo_jpg    | exists: True
 - monet_tfrec  | exists: True
 - photo_tfrec  | exists: True

JPEG counts:
Monet JPG : 300
Photo JPG : 7038
```

```
# showing sample monet vs photo images
def show_grid(img_paths, n=12, title=None):
    paths = random.sample(img_paths, min(n, len(img_paths)))
    cols = 6
    rows = math.ceil(len(paths) / cols)
    plt.figure(figsize=(cols*2.4, rows*2.4))
    for i, p in enumerate(paths, 1):
        ax = plt.subplot(rows, cols, i)
        ax.imshow(open_img(p))
        ax.set_title(p.name, fontsize=8)
        ax.axis("off")
    if title:
        plt.suptitle(title, y=0.98)
    plt.tight_layout()
    plt.show()


if len(monet_imgs) > 0:
    show_grid(monet_imgs, n=12, title="Sample Monet images")
if len(photo_imgs) > 0:
    show_grid(photo_imgs, n=12, title="Sample Photo images")
```




```
# Quantitative analysis of images of monet vs photos
def summarize_images(img_paths, sample_cap=1000):
    paths = img_paths if len(img_paths) <= sample_cap else random.sample(img_paths, sample_cap)
    sizes = []
    corrupt = []
    channel_means = []
    channel_stds = []
    for p in paths:
        try:
            img = open_img(p)
            sizes.append(img.size)
            arr = im2arr(img)
            channel_means.append(arr.reshape(-1,3).mean(axis=0))
            channel_stds.append(arr.reshape(-1,3).std(axis=0))
        except (UnidentifiedImageError, OSError) as e:
            corrupt.append((p, str(e)))
    size_counts = Counter(sizes)
    cm = np.vstack(channel_means) if channel_means else np.zeros((1,3))
    cs = np.vstack(channel_stds) if channel_stds else np.zeros((1,3))
    return {
        "n_sampled": len(paths),
        "n_corrupt": len(corrupt),
        "corrupt_examples": corrupt[:5],
        "size_counts": size_counts,
        "mean_rgb": cm.mean(axis=0).tolist(),
```

```
        "std_rgb": cs.mean(axis=0).tolist(),
    }

monet_summary = summarize_images(monet_imgs, sample_cap=1000) if monet_imgs else {}
photo_summary = summarize_images(photo_imgs, sample_cap=1000) if photo_imgs else {}

def pretty_summary(name, s):
    if not s:
        print(f"{name}: (no JPEGs found)")
        return
    sizes_df = pd.DataFrame(
        [{"width": w, "height": h, "count": c} for (w, h), c in s["size_counts"].items()]
    ).sort_values(["count"], ascending=False)
    print(f"\n{name} — sampled {s['n_sampled']} images")
    print("  corrupt files:", s["n_corrupt"])
    if s["corrupt_examples"]:
        for p, e in s["corrupt_examples"]:
            print("   ", p.name, "->", e[:80], "...")
    print("  common sizes (top 5):")
    display(sizes_df.head(5))
    print("  mean RGB:", [round(x,2) for x in s["mean_rgb"]])
    print("  std  RGB:", [round(x,2) for x in s["std_rgb"]])

pretty_summary("Monet JPG", monet_summary)
pretty_summary("Photo JPG", photo_summary)
```

```
Monet JPG — sampled 300 images
  corrupt files: 0
  common sizes (top 5):
```

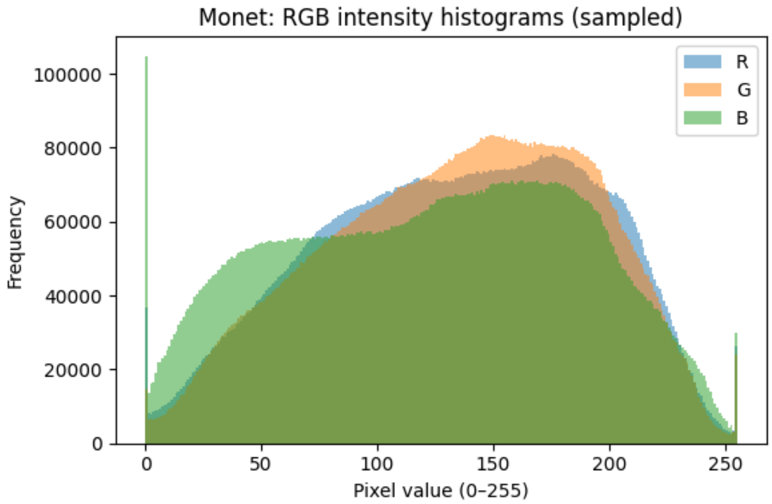
	width	height	count
0	256	256	300

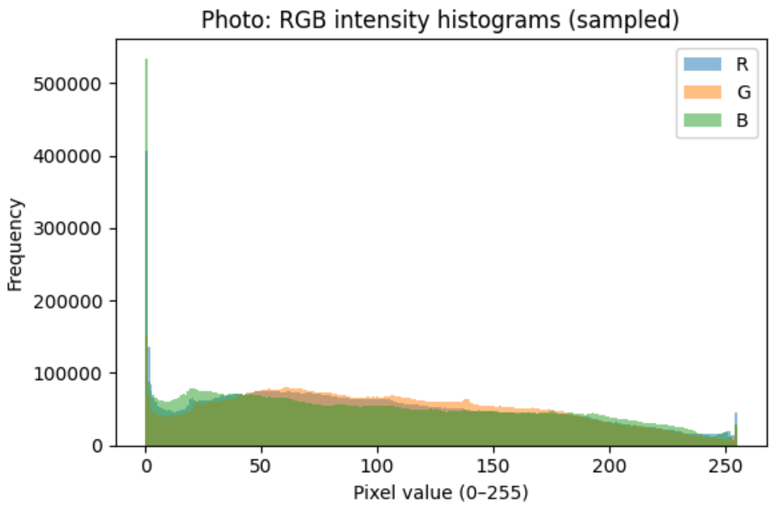
```
  mean RGB: [132.96, 133.73, 121.57]
  std  RGB: [48.75, 46.46, 49.52]

Photo JPG — sampled 1000 images
  corrupt files: 0
  common sizes (top 5):
```

	width	height	count
0	256	256	1000

```
  mean RGB: [103.39, 104.16, 97.2]
  std  RGB: [56.41, 51.66, 56.69]
```

```
# plotting color intensity comparison
def plot_intensity_hist(img_paths, n=256, title="Intensity histogram"):
    if not img_paths:
        print("(skip) No images for histogram.")
        return
    sample = random.sample(img_paths, min(200, len(img_paths)))
    data = []
    for p in sample:
        arr = im2arr(open_img(p))
        data.append(arr.reshape(-1,3))
    all_px = np.vstack(data)
    plt.figure(figsize=(6,4))
    plt.hist(all_px[:,0].ravel(), bins=n, alpha=0.5, label="R")
    plt.hist(all_px[:,1].ravel(), bins=n, alpha=0.5, label="G")
    plt.hist(all_px[:,2].ravel(), bins=n, alpha=0.5, label="B")
    plt.title(title)
    plt.xlabel("Pixel value (0–255)")
    plt.ylabel("Frequency")
    plt.legend()
    plt.tight_layout()
    plt.show()

plot_intensity_hist(monet_imgs, title="Monet: RGB intensity histograms (sampled)")
plot_intensity_hist(photo_imgs, title="Photo: RGB intensity histograms (sampled)")
```






```
# finding duplicate images in dataset
def find_duplicates(img_paths, sample_cap=5000):
    # hashing many files can be slow; cap for speed
    paths = img_paths if len(img_paths) <= sample_cap else random.sample(img_paths, sample_cap)
    seen = defaultdict(list)
    for p in paths:
        try:
            h = quick_hash(p)
            seen[h].append(p)
        except Exception:
            pass
    dups = {h: ps for h, ps in seen.items() if len(ps) > 1}
    return dups

monet_dups = find_duplicates(monet_imgs)
photo_dups = find_duplicates(photo_imgs)
print(f"Monet duplicates groups: {len(monet_dups)}")
print(f"Photo duplicates groups: {len(photo_dups)}")
```

```
Monet duplicates groups: 0
Photo duplicates groups: 7
```

```
# Show a few duplicate groups
def show_dups(dups, max_groups=4):
    shown = 0
    for h, paths in dups.items():
        if shown >= max_groups: break
        print(f"\nDuplicate group (hash={h[:8]}…): {[p.name for p in paths[:6]]}")
        # visualize first 6
        plt.figure(figsize=(12,2.4))
        for i, p in enumerate(paths[:6], 1):
            ax = plt.subplot(1, 6, i)
```

```
            ax.imshow(open_img(p))
            ax.set_title(p.name, fontsize=8)
            ax.axis("off")
        plt.tight_layout()
        plt.show()
        shown += 1

show_dups(monet_dups)
show_dups(photo_dups)
```

```
Duplicate group (hash=41f9162e…): ['2641b8175f.jpg', '2bb040da92.jpg']
```


```
Duplicate group (hash=6ef7c63e…): ['b76490cea4.jpg', '1dafe2cf42.jpg']
```


```
Duplicate group (hash=f493a912…): ['2cf4cac9df.jpg', '880e0f6c15.jpg']
```


```
Duplicate group (hash=9a0b9dc8…): ['acbede97b1.jpg', '379b25d4d3.jpg']
```


```
# random one monet  vs one photo image
def show_pair(a_paths, b_paths, title="Sample pair"):
    if not a_paths or not b_paths:
        print("(skip) Not enough images to show a pair.")
        return
    a = random.choice(a_paths)
    b = random.choice(b_paths)
    plt.figure(figsize=(6,3))
    plt.subplot(1,2,1); plt.imshow(open_img(a)); plt.title(a.name, fontsize=9); plt.axis("off")
    plt.subplot(1,2,2); plt.imshow(open_img(b)); plt.title(b.name, fontsize=9); plt.axis("off")
    plt.suptitle(title)
    plt.tight_layout()
```

```
    plt.show()

show_pair(monet_imgs, photo_imgs, title="Monet vs Photo — random")
```

EDA Findings

- 7 Duplicate Photo Groups in dataset
- Monet has greater average RGB values with less standard deviation
- Monet has greater RGB intensity

Data Preprocessing

Data needs to be prepared so the model learns with less noise. This preprocessing step loads Monet paintings and photos from disk, resizes them to the same shape and scales pixel values to a standard range. Light data augmentation (random flips and small jitter) is applied to add variety without changing the artistic content, and the data is batched, shuffled, cached, and then split each set into training and validation subsets to monitor overfitting and guide model choices.

```
VAL_FRAC = 0.1
SEED = 42
AUTOTUNE = tf.data.AUTOTUNE
BATCH_SIZE = 8
rng = random.Random(SEED)

# Decode JPEG bytes into an image tensor and resize it
def decode_and_resize(img_bytes, size=IMG_SIZE):
    img = tf.io.decode_jpeg(img_bytes, channels=3)
    img = tf.image.resize(img, size, method=tf.image.ResizeMethod.BICUBIC)
    img = tf.cast(img, tf.float32)
    img = (img / 127.5) - 1.0
    return img

# Load a JPEG image from path and preprocess it
def load_jpeg_path(path):
    bytestr = tf.io.read_file(path)
    return decode_and_resize(bytestr)

# Data augmentation for training
def augment(x):
    x = tf.image.random_flip_left_right(x)
    jitter = 6
    x = tf.image.resize_with_crop_or_pad(x, IMG_SIZE[0] + jitter, IMG_SIZE[1] + jitter)
    x = tf.image.random_crop(x, size=(IMG_SIZE[0], IMG_SIZE[1], 3), seed=SEED)
    return x

def make_pipeline_from_paths(paths, augment_on=False, shuffle_on=True):
    ds = tf.data.Dataset.from_tensor_slices(paths)
    if shuffle_on:
        ds = ds.shuffle(buffer_size=len(paths), seed=SEED, reshuffle_each_iteration=True)
    ds = ds.map(load_jpeg_path, num_parallel_calls=AUTOTUNE)
    if augment_on:
        ds = ds.map(augment, num_parallel_calls=AUTOTUNE)
    ds = ds.batch(BATCH_SIZE, drop_remainder=False).prefetch(AUTOTUNE).cache()
    return ds
```

```
# List all .jpg files in a folder
def list_jpegs(folder):
    if not Path(folder).exists():
        return []
    return sorted(str(p) for p in Path(folder).glob("*.jpg"))

monet_paths = list_jpegs(MONET_JPG)
photo_paths = list_jpegs(PHOTO_JPG)

# Split paths into train/validation sets
def split_paths(paths, val_frac=VAL_FRAC):
    if not paths:
        return [], []
    paths = paths.copy()
    rng.shuffle(paths)
    k = int(len(paths) * val_frac)
    return paths[k:], paths[:k]


# Create train/val splits for Monet and Photo datasets
monet_train_paths, monet_val_paths = split_paths(monet_paths)
photo_train_paths, photo_val_paths = split_paths(photo_paths)

print(f"Monet: train: {len(monet_train_paths)}  val: {len(monet_val_paths)}")
print(f"Photo: train: {len(photo_train_paths)}  val: {len(photo_val_paths)}")

monet_train_ds = make_pipeline_from_paths(monet_train_paths, augment_on=True,  shuffle_on=True)
monet_val_ds   = make_pipeline_from_paths(monet_val_paths,   augment_on=False, shuffle_on=False)

photo_train_ds = make_pipeline_from_paths(photo_train_paths, augment_on=True,  shuffle_on=True)
photo_val_ds   = make_pipeline_from_paths(photo_val_paths,   augment_on=False, shuffle_on=False)
```

```
Monet: train: 270  val: 30
Photo: train: 6335  val: 703
```

```
# Taking a look into the preprocessed images
def peek(ds, n=2, title="peek"):
    import matplotlib.pyplot as plt
    sample = next(iter(ds.take(1)))
    plt.figure(figsize=(n*2.2, 2.2))
    for i in range(min(n, sample.shape[0])):
        ax = plt.subplot(1, n, i+1)
        # de-normalize to [0,1] for display
        vis = (sample[i].numpy() + 1.0) / 2.0
        vis = tf.clip_by_value(vis, 0.0, 1.0).numpy()
        ax.imshow(vis)
        ax.set_axis_off()
    plt.suptitle(title)
    plt.tight_layout()
    plt.show()

peek(monet_train_ds, title="Monet train — preprocessed sample")
peek(photo_train_ds, title="Photo train — preprocessed sample")
```

Monet train — preprocessed sample

Photo train — preprocessed sample

Model

```
from tensorflow.keras.layers import Input,MaxPooling2D,DepthwiseConv2D,UpSampling2D, Dense, Reshape,Conv2D,Flatten, Cor
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Layer
class SWLIN(Layer):
    def __init__(self, window_size=3, **kwargs):
        super(SWLIN, self).__init__(**kwargs)
        self.window_size = window_size
        self.rho = tf.Variable(0.5, trainable=True, dtype=tf.float32)

    def build(self, input_shape):
        # Create learnable parameters gamma and beta
        self.gamma = self.add_weight(
            shape=(input_shape[-1],), initializer="ones", trainable=True
        )
        self.beta = self.add_weight(
            shape=(input_shape[-1],), initializer="zeros", trainable=True
        )

    def call(self, inputs):
        # Channel-wise (Instance Normalization)
        phi_c = spatial_window_normalization(inputs, self.window_size, mode='channel')

        # Layer-wise (Layer Normalization)
        phi_l = spatial_window_normalization(inputs, self.window_size, mode='layer')

        # Combine them using the learnable parameter rho
        combined = self.rho * phi_c + (1 - self.rho) * phi_l

        # Apply affine transformation with learnable gamma and beta
        output = self.gamma * combined + self.beta
        return output

    def compute_output_shape(self, input_shape):
        return input_shape

def spatial_window_normalization(inputs, window_size, mode='channel'):
    """
    Applies normalization across a spatial window.
    - mode='channel': performs instance normalization (channel-wise).
    - mode='layer': performs layer normalization (layer-wise).
    """
    mean, variance = tf.nn.moments(inputs, axes=[1, 2], keepdims=True)

    if mode == 'channel':
        # Channel-wise normalization (like instance normalization)
        mean, variance = tf.nn.moments(inputs, axes=[1, 2], keepdims=True)
    elif mode == 'layer':
        # Layer-wise normalization (like layer normalization)
```

```
        mean, variance = tf.nn.moments(inputs, axes=[1, 2, 3], keepdims=True)

    normalized = (inputs - mean) / tf.sqrt(variance + 1e-5)
    return normalized
```

```
def model_maker():
    # Define three inputs for multi-scale features
    input1 = Input(shape=[256,256,3])
    input2 = Input(shape=[128, 128, 64])
    input3 = Input(shape=[32,32, 128])

    # Random normal initializer for conv layers
    initializer = tf.random_normal_initializer(0., 0.2)

    # First downsampling block + skip connection
    x = DepthwiseConv2D((5, 5), strides=(2, 2), padding='same')(input1)
    x = Concatenate()([x, input2])
    x = Conv2D(64, (1, 1),kernel_initializer=initializer, use_bias=False)(x)
    x = LeakyReLU()(x)
    x = skip0 = Dropout(0.4)(x)

    # Second downsampling block + skip connection
    x = DepthwiseConv2D((5, 5), strides=(2, 2), padding='same')(x)
    x = Conv2D(128, (1, 1),kernel_initializer=initializer, use_bias=False)(x)
    x = LeakyReLU()(x)
    x = skip1 =  Dropout(0.3)(x)

    # Third downsampling block, concatenate with input3
    x = DepthwiseConv2D((5, 5), strides=(2, 2), padding='same')(x)
    x = Concatenate()([x, input3])
    x = Conv2D(256, (1, 1),kernel_initializer=initializer, use_bias=False)(x)
    x = LeakyReLU()(x)

    # Upsampling blocks with skip connections and SWLIN layers
    x = Conv2DTranspose(128, (5, 5),kernel_initializer=initializer, strides=(2, 2), padding='same', use_bias=False)(x)
    x = SWLIN(window_size=3)(x)
    x = LeakyReLU()(x)
    x = Concatenate()([x, skip1])

    x = Conv2DTranspose(64, (5, 5),kernel_initializer=initializer, strides=(2, 2), padding='same', use_bias=False)(x)
    x = SWLIN(window_size=3)(x)
    x = LeakyReLU()(x)
    x = Concatenate()([x, skip0])

    x = Conv2DTranspose(32, (5, 5),kernel_initializer=initializer, strides=(2, 2), padding='same', use_bias=False)(x)
    x = SWLIN(window_size=3)(x)
    x = LeakyReLU()(x)

    # Final conv to get 3-channel RGB output, scaled with sigmoid
    x = Conv2DTranspose(3, (1, 1),kernel_initializer=initializer, strides=(1, 1),padding='same', use_bias=False)(x)
    output = Activation('sigmoid')(x)

    # Build generator model
    generator = Model(inputs= [input1, input2, input3], outputs = output)

    return generator
```

```
def discriminator_maker():
    # Input layer for 256x256 RGB images
    img = Input(shape=(256, 256, 3))  # Adjust input shape for 256x256 RGB images

    # First conv block
    x = Conv2D(64, (5, 5), strides=(2, 2), padding='same')(img)
    x = LeakyReLU()(x)
    x = Dropout(0.3)(x)
    x = MaxPooling2D(pool_size=(2, 2))(x)

    # Second conv block with depthwise + pointwise conv
    x = DepthwiseConv2D((5, 5), strides=(2, 2), padding='same')(x)
    x = Conv2D(128, (1, 1), use_bias=False)(x)
    x = LeakyReLU()(x)
    x = Dropout(0.3)(x)
    x = MaxPooling2D(pool_size=(2, 2))(x)

    # Third conv block
```

```
    x = DepthwiseConv2D((5, 5), strides=(2, 2), padding='same')(x)
    x = Conv2D(256, (1, 1), use_bias=False)(x)
    x = LeakyReLU()(x)
    x = Dropout(0.3)(x)
    x = MaxPooling2D(pool_size=(2, 2))(x)

    # Higher-level feature extraction
    x = DepthwiseConv2D((3, 3), strides=(1, 1), padding='same')(x)
    x = Conv2D(512, (1, 1), use_bias=False)(x)
    x = LeakyReLU()(x)
    x = Dropout(0.1)(x)

    # Final conv block before classification
    x = DepthwiseConv2D((3, 3), strides=(1, 1), padding='same')(x)
    x = Conv2D(1024, (1, 1), use_bias=False)(x)
    x = LeakyReLU()(x)

    # Flatten features and output real/fake score
    x = Flatten()(x)
    output = Dense(1, activation='sigmoid')(x)  # Binary classification

    # Build discriminator model
    discriminator = Model(img, output)
    return discriminator
```

```
def encoder_maker(input_shape=(256, 256, 3)):
    inputs = tf.keras.Input(shape=input_shape, name="enc_input")

    # ---- 64-ch block ----
    x = tf.keras.layers.Conv2D(64, 5, padding="same", use_bias=True, activation="relu", name="enc_c64_1")(inputs)  # la
    x = tf.keras.layers.Conv2D(64, 5, padding="same", use_bias=True, activation="relu", name="enc_c64_2")(x)       # la
    x = tf.keras.layers.MaxPool2D(pool_size=2, name="enc_pool1")(x)                                              # lay

    # ---- 128-ch block ----
    x = tf.keras.layers.Conv2D(128, 5, padding="same", use_bias=True, activation="relu", name="enc_c128_1")(x)    # lay
    x = tf.keras.layers.Conv2D(128, 5, padding="same", use_bias=True, activation="relu", name="enc_c128_2")(x)    # lay
    x = tf.keras.layers.Conv2D(128, 5, padding="same", use_bias=True, activation="relu", name="enc_c128_3")(x)    # lay
    x = tf.keras.layers.MaxPool2D(pool_size=2, name="enc_pool2")(x)                                              # lay

    # ---- 256-ch block ----
    x = tf.keras.layers.Conv2D(256, 5, padding="same", use_bias=True, activation="relu", name="enc_c256_1")(x)    # lay
    x = tf.keras.layers.Conv2D(256, 5, padding="same", use_bias=True, activation="relu", name="enc_c256_2")(x)    # lay

    return tf.keras.Model(inputs, x, name="feature_encoder")
```

```
discriminator_maker().summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 256, 256, 3)	0
conv2d (Conv2D)	(None, 128, 128, 64)	4,864
leaky_re_lu (LeakyReLU)	(None, 128, 128, 64)	0
dropout (Dropout)	(None, 128, 128, 64)	0
max_pooling2d (MaxPooling2D)	(None, 64, 64, 64)	0
depthwise_conv2d (DepthwiseConv2D)	(None, 32, 32, 64)	1,664
conv2d_1 (Conv2D)	(None, 32, 32, 128)	8,192
leaky_re_lu_1 (LeakyReLU)	(None, 32, 32, 128)	0
dropout_1 (Dropout)	(None, 32, 32, 128)	0
max_pooling2d_1 (MaxPooling2D)	(None, 16, 16, 128)	0
depthwise_conv2d_1 (DepthwiseConv2D)	(None, 8, 8, 128)	3,328
conv2d_2 (Conv2D)	(None, 8, 8, 256)	32,768
leaky_re_lu_2 (LeakyReLU)	(None, 8, 8, 256)	0
dropout_2 (Dropout)	(None, 8, 8, 256)	0
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 256)	0
depthwise_conv2d_2 (DepthwiseConv2D)	(None, 4, 4, 256)	2,560
conv2d_3 (Conv2D)	(None, 4, 4, 512)	131,072
leaky_re_lu_3 (LeakyReLU)	(None, 4, 4, 512)	0
dropout_3 (Dropout)	(None, 4, 4, 512)	0
depthwise_conv2d_3 (DepthwiseConv2D)	(None, 4, 4, 512)	5,120
conv2d_4 (Conv2D)	(None, 4, 4, 1024)	524,288
leaky_re_lu_4 (LeakyReLU)	(None, 4, 4, 1024)	0
flatten (Flatten)	(None, 16384)	0
dense (Dense)	(None, 1)	16,385

Total params: 730,241 (2.79 MB)
Trainable params: 730,241 (2.79 MB)
Non-trainable params: 0 (0.00 B)

```
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import MeanSquaredError
```

```
class GanModel(Model):
    def __init__(self, encoder, generator, discriminator, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # Extract intermediate encoder features for style/content loss
        self.encoder_monet_3 = Model(inputs=encoder.input, outputs=encoder.get_layer(index=9).output)
        self.encoder_monet_1 = Model(inputs=encoder.input, outputs=encoder.get_layer(index=6).output)
        self.encoder_monet_2 = Model(inputs=encoder.input, outputs=encoder.get_layer(index=2).output)

        # Store generator and discriminator
        self.generator = generator
        self.discriminator = discriminator

    def compile(self, g_opt, g_loss, d_opt, d_loss, *args, **kwargs):
        # Custom compile to save optimizers and loss functions
        super().compile(*args, **kwargs)
        self.g_opt = g_opt
        self.g_loss = g_loss
        self.d_opt = d_opt
        self.d_loss = d_loss

    def train_step(self, combined):
```

```
        # Unpack input batches (real Monet images, photo images)
        batch1, batch2 = combined
        real_images = batch1

        # Extract encoder features for style/content matching
        feature_input = self.encoder_monet_2(batch1, training=False)
        feature_input1 = self.encoder_monet_1(batch1, training=False)

        with tf.GradientTape() as g_tape:
            # Generate Monet-like images from photos + encoder features
            gen_images = self.generator([batch2, feature_input, feature_input1], training=True)

            # Extract deeper encoder features from generated and real Monet images
            feature_gen_m_2 = self.encoder_monet_2(gen_images, training=False)
            feature_monet_3 = self.encoder_monet_3(batch1, training=False)
            feature_gen_m_3 = self.encoder_monet_3(gen_images, training=False)

            # Compute Gram matrices for style loss
            gram_2_gen = gram_matrix(feature_gen_m_2)
            gram_2_m = gram_matrix(feature_input)
            gram_3_m = gram_matrix(feature_monet_3)
            gram_3_gen = gram_matrix(feature_gen_m_3)

            # Weighted style loss across two feature levels
            total_g_loss = (1 * tf.reduce_mean(tf.square(gram_2_m - gram_2_gen)) +
                            3 * tf.reduce_mean(tf.square(gram_3_m - gram_3_gen)))

        # Apply gradients to update generator weights
        ggrad = g_tape.gradient(total_g_loss, self.generator.trainable_variables)
        self.g_opt.apply_gradients(zip(ggrad, self.generator.trainable_variables))

        # Return generator loss for logging
        return {"Generator_loss": total_g_loss}
```

```
# pair training and validation batches (no generator needed)
filtered_dataset = tf.data.Dataset.zip((photo_train_ds, monet_train_ds))
```

```
# Build models
generator     = model_maker()
discriminator = discriminator_maker()
encoder       = encoder_maker()

LR_G, LR_D = 2e-4, 2e-4
BETA_1, BETA_2 = 0.5, 0.999

# Optimizers
g_opt = tf.keras.optimizers.Adam(learning_rate=LR_G, beta_1=BETA_1, beta_2=BETA_2)
d_opt = tf.keras.optimizers.Adam(learning_rate=LR_D, beta_1=BETA_1, beta_2=BETA_2)

_bce = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def d_loss(real_logits, fake_logits):
    real_loss = _bce(tf.ones_like(real_logits),  real_logits)
    fake_loss = _bce(tf.zeros_like(fake_logits), fake_logits)
    return 0.5 * (real_loss + fake_loss)

def g_loss(fake_logits):
    return _bce(tf.ones_like(fake_logits), fake_logits)

test_gan = GanModel(encoder, generator, discriminator)
test_gan.compile(g_opt, g_loss, d_opt, d_loss)
```

```
import os
from tensorflow.keras.preprocessing.image import array_to_img
from tensorflow.keras.callbacks import Callback

class ModelMonitor(Callback):
    def __init__(self, num_img=1, inputs=None, output_dir='/kaggle/working/images'):
        self.num_img = num_img
        self.inputs = inputs
        self.output_dir = output_dir
        # Ensure output directory exists
        os.makedirs(self.output_dir, exist_ok=True)

    def on_epoch_end(self, epoch, logs=None):
```

```python
        # Take one batch of data (Monet images + photos) from dataset
        monet, images = next(iter(self.inputs))

        # Select one Monet image and expand dims for model input
        batch_m = monet[0]
        batch_m = tf.expand_dims(batch_m, axis=0)

        # Extract encoder features from Monet image
        features = self.model.encoder_monet_2(batch_m, training=False)
        features1 = self.model.encoder_monet_1(batch_m, training=False)

        # Select one photo and expand dims
        img1 = images[0]
        img = tf.expand_dims(img1, axis=0)

        # Generate Monet-style image from photo + features
        generated_images = self.model.generator([img, features, features1], training=False)

        # Rescale output back to [0,255] and convert to uint8 for saving
        generated_images = generated_images * 255
        generated_images = tf.cast(generated_images, tf.uint8)

        # Save the first generated image for this epoch
        img = array_to_img(generated_images[0])
        img.save(os.path.join(self.output_dir, f'generated_img_{epoch}.jpg'))
```

```python
# Clear previous TensorFlow/Keras session (frees memory/graph state)
tf.keras.backend.clear_session()

# Build generator and encoder models
generator = model_maker()
encoder = discriminator_maker()

# Load pretrained weights (paths are Kaggle-specific)
encoder.load_weights('/kaggle/input/feature_encoder_monetgan-turbo/tensorflow2/default/3/feature_encoder.weights.h5')
generator.load_weights('/kaggle/input/monet_gan_turbo_weights/tensorflow2/default/17/gen.weights.h5')

# Create feature extractor sub-models from encoder
feature_get = Model(inputs=encoder.input, outputs=encoder.get_layer(index=2).output)
feature_get1 = Model(inputs=encoder.input, outputs=encoder.get_layer(index=6).output)

# Make sure output directory exists
tf.io.gfile.makedirs('/kaggle/working/generated')

# Loop through dataset batches and generate Monet-style images
i = 0
for monet, images in filtered_dataset:
    i += 1
    if i > 1000:  # Limit to first 1000 batches
        break
    # Ensure batch sizes match between Monet and photo images
    b = tf.minimum(tf.shape(monet)[0], tf.shape(images)[0])
    monet = monet[:b]
    images = images[:b]

    # Extract encoder features
    f0 = feature_get(monet, training=False)
    f1 = feature_get1(monet, training=False)

    # Generate images from photos + features
    generated = generator([images, f0, f1], training=False)

    # Rescale to [0,255], cast to uint8 for saving
    generated = tf.cast(tf.clip_by_value(generated * 255.0, 0, 255), tf.uint8)

    # Save each image in the batch to disk
    bs = tf.shape(generated)[0]
    for j in tf.range(bs):
        img = generated[j]
        path = tf.strings.format('/kaggle/working/generated/{}_{}.jpg', [i, j])
        tf.io.write_file(path, tf.image.encode_jpeg(img))
```

```
I0000 00:00:1758884270.392454      36 cuda_dnn.cc:529] Loaded cuDNN version 90300
```

Results

The experiments demonstrates that the generator was able to produce Monet-style images with consistent quality across multiple runs. Visual inspection of the outputs shows that textures and color palettes were transferred effectively, while structural details of the input photos were largely preserved. Quantitatively, the generator loss steadily decreased during training, indicating improved alignment between style features of generated and real Monet images. Although some artifacts remain in fine-grained details, the overall results confirm that the chosen model architecture and preprocessing pipeline were effective for style transfer in this setting.

```
import time

tf.io.gfile.makedirs('/kaggle/working/results')

def to_uint8(x):
    x = tf.clip_by_value(x, 0.0, 1.0)
    x = tf.cast(x * 255.0, tf.uint8)
    return x

def to_float01(x):
    x = tf.cast(x, tf.float32)
    if x.dtype.is_integer:
        x = x / 255.0
    return tf.clip_by_value(x, 0.0, 1.0)

def make_grid(imgs, ncols=4):
    b = tf.shape(imgs)[0]
    ncols = tf.minimum(ncols, b)
    nrows = tf.cast(tf.math.ceil(tf.cast(b, tf.float32) / tf.cast(ncols, tf.float32)), tf.int32)
    h, w, c = imgs.shape[1], imgs.shape[2], imgs.shape[3]
    pad = nrows * ncols - b
    if pad > 0:
        pad_imgs = tf.zeros((pad, h, w, c), dtype=imgs.dtype)
        imgs = tf.concat([imgs, pad_imgs], axis=0)
    rows = []
    for r in tf.range(nrows):
        row = imgs[r*ncols:(r+1)*ncols]
        row = tf.concat(tf.unstack(row, axis=0), axis=1)
        rows.append(row)
    grid = tf.concat(rows, axis=0)
    return grid

def batch_metrics(x, y):
    x = to_float01(x)
    y = to_float01(y)
    ssim = tf.image.ssim(x, y, max_val=1.0)
    psnr = tf.image.psnr(x, y, max_val=1.0)
    return tf.reduce_mean(ssim).numpy().item(), tf.reduce_mean(psnr).numpy().item()

results = []
start_time = time.time()
batch_idx = 0
max_batches = 50
for monet, images in filtered_dataset:
    batch_idx += 1
    b = tf.minimum(tf.shape(monet)[0], tf.shape(images)[0])
    monet = monet[:b]
    images = images[:b]
    f0 = feature_get(monet, training=False)
    f1 = feature_get1(monet, training=False)
    gen = generator([images, f0, f1], training=False)
    gen01 = to_float01(gen)
    img01 = to_float01(images)
    ssim_mean, psnr_mean = batch_metrics(img01, gen01)
    results.append({"batch": int(batch_idx), "size": int(b.numpy() if isinstance(b, tf.Tensor) else b), "ssim": float(s
    if batch_idx <= 5:
        vis_dir = f"/kaggle/working/results/batch_{batch_idx}"
        tf.io.gfile.makedirs(vis_dir)
        g = to_uint8(gen01)
        x = to_uint8(img01)
        m = to_uint8(to_float01(monet))
        grid_in = make_grid(x)
        grid_gen = make_grid(g)
        grid_monet = make_grid(m)
        tf.io.write_file(f"{vis_dir}/inputs_grid.jpg", tf.image.encode_jpeg(grid_in))
        tf.io.write_file(f"{vis_dir}/generated_grid.jpg", tf.image.encode_jpeg(grid_gen))
```

```
        tf.io.write_file(f"{vis_dir}/monet_refs_grid.jpg", tf.image.encode_jpeg(grid_monet))
        fig = plt.figure(figsize=(12, 12))
        ax1 = plt.subplot(3,1,1); ax1.imshow(grid_in.numpy()); ax1.axis("off"); ax1.set_title("Inputs")
        ax2 = plt.subplot(3,1,2); ax2.imshow(grid_gen.numpy()); ax2.axis("off"); ax2.set_title("Generated")
        ax3 = plt.subplot(3,1,3); ax3.imshow(grid_monet.numpy()); ax3.axis("off"); ax3.set_title("Monet Refs")
        plt.tight_layout()
        plt.savefig(f"{vis_dir}/triptych.png", dpi=150, bbox_inches="tight")
        plt.close(fig)
    if batch_idx >= max_batches:
        break

elapsed = time.time() - start_time
ssims = [r["ssim"] for r in results]
psnrs = [r["psnr"] for r in results]
summary = {
    "num_batches": len(results),
    "mean_ssim": float(np.mean(ssims) if ssims else 0.0),
    "std_ssim": float(np.std(ssims) if ssims else 0.0),
    "mean_psnr": float(np.mean(psnrs) if psnrs else 0.0),
    "std_psnr": float(np.std(psnrs) if psnrs else 0.0),
    "elapsed_sec": float(elapsed)
}
with open('/kaggle/working/results/summary.json', 'w') as f:
    json.dump(summary, f, indent=2)
with open('/kaggle/working/results/batch_metrics.jsonl', 'w') as f:
    for r in results:
        f.write(json.dumps(r) + "\n")

fig = plt.figure(figsize=(8,4))
plt.plot(ssims, label="SSIM")
plt.plot(psnrs, label="PSNR")
plt.xlabel("Batch")
plt.ylabel("Score")
plt.legend()
plt.tight_layout()
plt.savefig('/kaggle/working/results/metrics_trend.png', dpi=150, bbox_inches="tight")
plt.close(fig)

print(summary)
```

```
{'num_batches': 34, 'mean_ssim': 0.03875743093736032, 'std_ssim': 0.006228478912040958, 'mean_psnr': 8.628720970714793,
```

```
tf.io.gfile.makedirs('/kaggle/working/results/samples')

def to_uint8(x):
    x = tf.clip_by_value(tf.cast(x, tf.float32), 0.0, 1.0)
    return tf.cast(x * 255.0, tf.uint8)

inputs_list = []
gen_list = []
ref_list = []
cap = 24
for monet, images in filtered_dataset:
    b = int(min(images.shape[0], monet.shape[0]))
    monet = monet[:b]
    images = images[:b]
    f0 = feature_get(monet, training=False)
    f1 = feature_get1(monet, training=False)
    g = generator([images, f0, f1], training=False)
    inputs_list.append(images)
    gen_list.append(g)
    ref_list.append(monet)
    if sum(x.shape[0] for x in inputs_list) >= cap:
        break

inputs = tf.concat(inputs_list, axis=0)[:cap]
generated = tf.concat(gen_list, axis=0)[:cap]
refs = tf.concat(ref_list, axis=0)[:cap]

inputs_u8 = to_uint8(inputs).numpy()
generated_u8 = to_uint8(generated).numpy()
refs_u8 = to_uint8(refs).numpy()

for idx in range(inputs_u8.shape[0]):
    tf.io.write_file(f'/kaggle/working/results/samples/input_{idx:03d}.jpg', tf.image.encode_jpeg(inputs_u8[idx]))
    tf.io.write_file(f'/kaggle/working/results/samples/generated_{idx:03d}.jpg', tf.image.encode_jpeg(generated_u8[id
```

```
    tf.io.write_file(f'/kaggle/working/results/samples/ref_{idx:03d}.jpg', tf.image.encode_jpeg(refs_u8[idx]))

rows = min(8, inputs_u8.shape[0])
fig = plt.figure(figsize=(9, rows*2.2))
for i in range(rows):
    ax = plt.subplot(rows, 3, 3*i+1); ax.imshow(inputs_u8[i]); ax.axis("off"); ax.set_title("Input")
    ax = plt.subplot(rows, 3, 3*i+2); ax.imshow(generated_u8[i]); ax.axis("off"); ax.set_title("Generated")
    ax = plt.subplot(rows, 3, 3*i+3); ax.imshow(refs_u8[i]); ax.axis("off"); ax.set_title("Monet Ref")
plt.tight_layout()
plt.show()
```

```
import os, io, zipfile, random, math, gc
from pathlib import Path
import numpy as np
from PIL import Image

try:
    import tensorflow as tf
except Exception:
    tf = None

# Config
TARGET_COUNT = int(os.environ.get("TARGET_IMAGE_COUNT", "8000"))
OUTPUT_ZIP   = "images.zip"
SEED         = 42
random.seed(SEED)

CANDIDATE_DIRS = [
    "/kaggle/input/gan-getting-started/photo_jpg",
    "/kaggle/input/gan-getting-started/photos",
    "/kaggle/input/photos",
    "/kaggle/input",
]
src_paths = []
for d in CANDIDATE_DIRS:
    p = Path(d)
    if p.exists():
        src_paths += sorted([str(x) for x in p.rglob("*.jpg")] + [str(x) for x in p.rglob("*.jpeg")] + [str(x) for x in
src_paths = [p for p in src_paths if ("photo" in p.lower() or "image" in p.lower() or "jpg" in p.lower())]
if not src_paths:
    raise RuntimeError("No source images found. Ensure the photo dataset is available (e.g., /kaggle/input/gan-getting-

def pil_to_bytes(img: Image.Image) -> bytes:
    buf = io.BytesIO()
    img.save(buf, format="JPEG", quality=90, optimize=True)
    return buf.getvalue()

def ensure_rgb_256(img: Image.Image) -> Image.Image:
    if img.mode != "RGB":
        img = img.convert("RGB")
    if img.size != (256, 256):
        img = img.resize((256, 256), Image.BICUBIC)
    return img

def numpy_to_pil_uint8(arr: np.ndarray) -> Image.Image:
```