GAN Kaggle Mini-Project

GAN - Generative Adversarial Network

Citation: https://arxiv.org/abs/1406.2661

We propose a new framework for estimating generative models via an adversarial process, in which we simultaneously train two models: a generative model G that captures the data distribution, and a discriminative model D that estimates the probability that a sample came from the training data rather than G. The training procedure for G is to maximize the probability of D making a mistake. This framework corresponds to a minimax two-player game. In the space of arbitrary functions G and D, a unique solution exists, with G recovering the training data distribution and D equal to 1/2 everywhere. In the case where G and D are defined by multilayer perceptrons, the entire system can be trained with backpropagation.

Problem / Kaggle Competition: "I'm Something of a Painter Myself"

Citation: Kaggle - https://www.kaggle.com/competitions/gan-getting-started/overview

A GAN consists of at least two neural networks: a generator model and a discriminator model. The generator is a neural network that creates the images. For our competition, you should generate images in the style of Monet. This generator is trained using a discriminator.
The two models will work against each other, with the generator trying to trick the discriminator, and the discriminator trying to accurately classify the real vs. generated images.
Your task is to build a GAN that generates 7,000 to 10,000 Monet-style images.

Kaggle score: https://www.kaggle.com/competitions/gan-getting-started/leaderboard

- FID measures how close your generated images are to real images.
- Lower FID = better quality
- 0 means perfect match (never happens in practice)
- So on the leaderboard: Lower score = Better

Data

Citation: Kaggle - https://www.kaggle.com/competitions/gan-getting-started/data

Dataset
The dataset contains four directories: monet_tfrec, photo_tfrec, monet_jpg, and photo_jpg. The monet_tfrec and monet_jpg directories contain the same painting images, and the photo_tfrec and photo_jpg directories contain the same photos.
The monet directories contain Monet paintings. Use these images to train your model.
The photo directories contain photos. Add Monet-style to these images and submit your generated jpeg images as a zip file. Other photos outside of this dataset can be transformed but keep your submission file limited to 10,000 images.

Plan

- Import Libraries
- Setup configuration
- Check the data
- Define Functions
- Load the data
- EDA
- Model Setup
- Model Training
- Generate file for Kaggle submission
- Submission
- Conclusions
- Discussion
- Link GitHub Repository
- Citation
- AI Acknowledgements

Notebook

- The Notebook is setup to run in the Kaggle environment.
- Set "Accelerator" to "GPU T4x2" in the Kaggle environment.
- A similar notebook was used in Kaggle to submit to Kaggle as "notebook03b8491926 - Version3"

```python
# Libraries
import os
import re
import zipfile
from pathlib import Path
import matplotlib.pyplot as plt
import random
from PIL import Image
from tqdm import tqdm

import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from kaggle_datasets import KaggleDatasets
```

```python
# Config

print("TensorFlow version:", tf.__version__)

# Disable XLA (can be unstable on Kaggle GPU combos)
tf.config.optimizer.set_jit(False)
print("XLA disabled.")

AUTO = tf.data.AUTOTUNE

# Use default strategy
strategy = tf.distribute.get_strategy()
print("Using default strategy")
print("Replicas in sync:", strategy.num_replicas_in_sync)

IMAGE_SIZE = (256, 256)
IMG_HEIGHT, IMG_WIDTH = IMAGE_SIZE
CHANNELS = 3

BATCH_SIZE = 1 * strategy.num_replicas_in_sync     # Up to 4
EPOCHS = 10
BUFFER_SIZE = 1024

LR = 2e-4
LAMBDA_CYCLE = 10.0
LAMBDA_ID = 0.5 * LAMBDA_CYCLE

N_GENERATED_IMAGES = 7000

COMPETITION_NAME = "gan-getting-started"
GCS_PATH = KaggleDatasets().get_gcs_path(COMPETITION_NAME)
```

```
TensorFlow version: 2.18.0
XLA disabled.
Using default strategy
Replicas in sync: 1
```

Check the data

```python
# Check data
def count_data_items(filenames):
    counts = []
    for f in filenames:
        m = re.search(r"-([0-9]+)\.tfrec$", f)
        if m:
            counts.append(int(m.group(1)))
    if counts:
        return int(np.sum(counts))
    return len(filenames)

MONET_TFRECS = tf.io.gfile.glob(os.path.join(GCS_PATH, "monet_tfrec", "*.tfrec"))
PHOTO_TFRECS = tf.io.gfile.glob(os.path.join(GCS_PATH, "photo_tfrec", "*.tfrec"))

print("Monet TFRecord files:", len(MONET_TFRECS))
print("Photo TFRecord files:", len(PHOTO_TFRECS))

if len(MONET_TFRECS) == 0 or len(PHOTO_TFRECS) == 0:
    raise FileNotFoundError(
        "TFRecord files not found. Make sure the 'gan-getting-started' "
        "dataset is attached to this Kaggle notebook."
    )

n_monet = count_data_items(MONET_TFRECS)
n_photo = count_data_items(PHOTO_TFRECS)

print("Approx Monet images:", n_monet)
print("Approx Photo images:", n_photo)
```

```
Monet TFRecord files: 5
Photo TFRecord files: 20
Approx Monet images: 300
Approx Photo images: 7038
```

Functions

```python
def decode_image(image_bytes):
    image = tf.image.decode_jpeg(image_bytes, channels=CHANNELS)
    image = tf.image.convert_image_dtype(image, tf.float32)
    image = image * 2.0 - 1.0
    image = tf.reshape(image, [IMG_HEIGHT, IMG_WIDTH, CHANNELS])
    return image

def read_tfrecord(example_proto):
    feature_description = {
        "image_name": tf.io.FixedLenFeature([], tf.string),
        "image": tf.io.FixedLenFeature([], tf.string),
        "target": tf.io.FixedLenFeature([], tf.string),
    }
    example = tf.io.parse_single_example(example_proto, feature_description)
    image = decode_image(example["image"])
    return image

def augment_image(image):
    image = tf.image.random_flip_left_right(image)
    new_size = tf.random.uniform([], minval=256, maxval=286, dtype=tf.int32)
    image = tf.image.resize(image, [new_size, new_size])
    image = tf.image.random_crop(image, [IMG_HEIGHT, IMG_WIDTH, CHANNELS])
    return image

def load_dataset(filenames, augment=False, shuffle=False, repeat=False, batch_size=1):
    ds = tf.data.TFRecordDataset(filenames, num_parallel_reads=AUTO)
    ds = ds.map(read_tfrecord, num_parallel_calls=AUTO)
    if augment:
        ds = ds.map(augment_image, num_parallel_calls=AUTO)
    if shuffle:
        ds = ds.shuffle(BUFFER_SIZE, reshuffle_each_iteration=True)
    if repeat:
        ds = ds.repeat()
    ds = ds.batch(batch_size, drop_remainder=True)
    ds = ds.prefetch(AUTO)
    return ds
```

Load the Data

```python
monet_ds_train = load_dataset(
    MONET_TFRECS,
    augment=True,
    shuffle=True,
    repeat=True,
    batch_size=BATCH_SIZE,
)
photo_ds_train = load_dataset(
    PHOTO_TFRECS,
    augment=True,
    shuffle=True,
    repeat=True,
    batch_size=BATCH_SIZE,
)

train_ds = tf.data.Dataset.zip((photo_ds_train, monet_ds_train))

STEPS_PER_EPOCH = min(n_monet, n_photo) // BATCH_SIZE
print("Steps per epoch:", STEPS_PER_EPOCH)
if STEPS_PER_EPOCH == 0:
    raise ValueError("STEPS_PER_EPOCH is 0. Reduce BATCH_SIZE or check data.")

photo_ds_inference = load_dataset(
    PHOTO_TFRECS,
    augment=False,
    shuffle=False,
    repeat=False,
    batch_size=1,
)
```

EDA

- Visial inspection of samples
- Check width distribution
- Show pixel-intensity histograms

```
# Visual Inspection Samples
def show_samples(paths, title, n=6):
    plt.figure(figsize=(12, 6))
    chosen = random.sample(paths, min(n, len(paths)))
    for i, p in enumerate(chosen):
        img = Image.open(p)
        plt.subplot(2, 3, i+1)
        plt.imshow(img)
        plt.axis("off")
        plt.title(os.path.basename(p))
    plt.suptitle(title, fontsize=16)
    plt.show()

monet_files = sorted(tf.io.gfile.glob("/kaggle/input/gan-getting-started/monet_jpg/*.jpg"))
photo_files = sorted(tf.io.gfile.glob("/kaggle/input/gan-getting-started/photo_jpg/*.jpg"))
show_samples(monet_files, "Random Monet Images")
show_samples(photo_files, "Random Photo Images")

# Metadata Statistics
def compute_metadata(paths, sample_size=200):
    widths, heights = [], []
    means = []
    stds = []

    sample = random.sample(paths, min(sample_size, len(paths)))

    for p in tqdm(sample, desc="Metadata"):
        img = np.array(Image.open(p).convert("RGB"))
        h, w, _ = img.shape
        heights.append(h)
        widths.append(w)
        means.append(img.mean())
        stds.append(img.std())

    return widths, heights, means, stds

monet_w, monet_h, monet_mean, monet_std = compute_metadata(monet_files)
photo_w, photo_h, photo_mean, photo_std = compute_metadata(photo_files)

# Plot dimension distributions
plt.figure(figsize=(12,5))
plt.hist(monet_w, bins=20, alpha=0.5, label="Monet width")
plt.hist(photo_w, bins=20, alpha=0.5, label="Photo width")
plt.legend()
plt.title("Image Width Distribution")
plt.show()

plt.figure(figsize=(12,5))
plt.hist(monet_mean, bins=20, alpha=0.5, label="Monet pixel mean")
plt.hist(photo_mean, bins=20, alpha=0.5, label="Photo pixel mean")
plt.legend()
plt.title("Pixel Intensity Means")
plt.show()
```

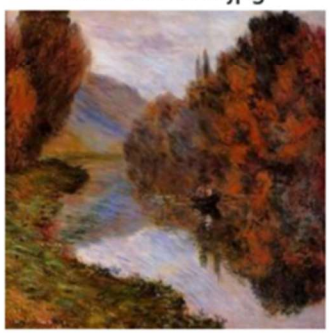
Random Monet Images

dcf5ea1040.jpg

23b07c3769.jpg

c2576267d4.jpg

b3adc75e7d.jpg



cfc6fce7b5.jpg

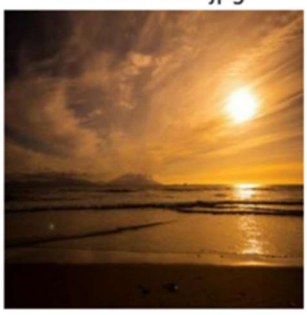
d5b0c260a0.jpg

Random Photo Images

ab808be34d.jpg

276c73bead.jpg

b11afb309d.jpg

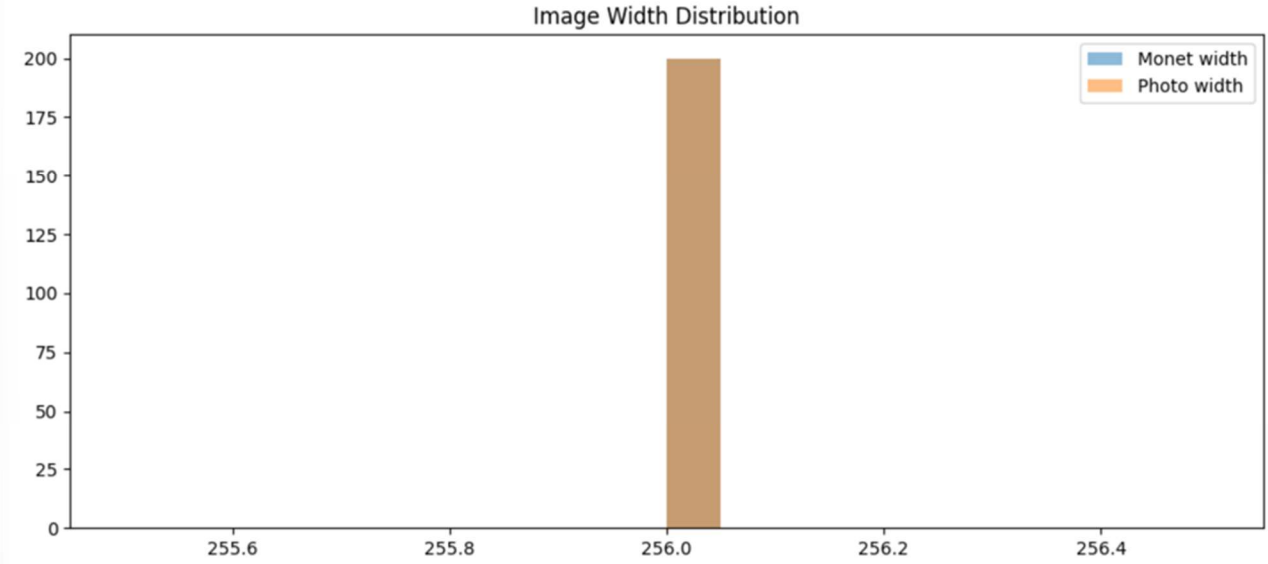


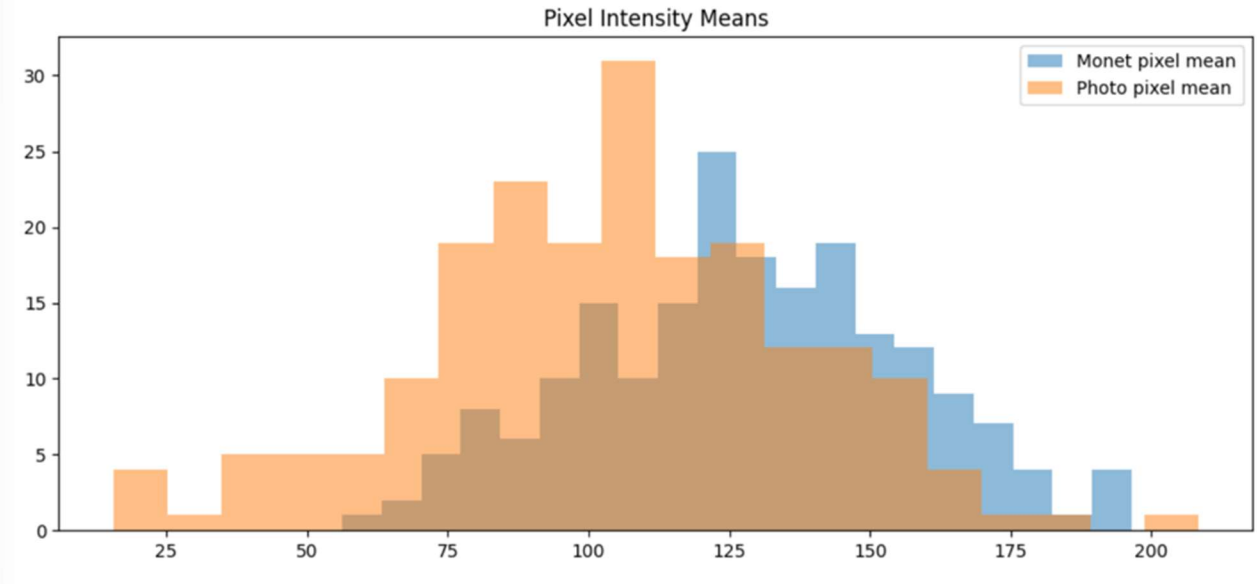
a8e62405d3.jpg

6b04d109cf.jpg

4ca019a3d6.jpg

```
Metadata: 100%|██████████| 200/200 [00:00<00:00, 455.54it/s]
Metadata: 100%|██████████| 200/200 [00:00<00:00, 476.22it/s]
```





Insights

- The width distribution indicates that all images are uniformly sized, so no resizing issues are present.
- Pixel-intensity histograms reveal that Monet images tend to have slightly higher and more widely spread brightness values. The photos cluster is more tightly at lower intensities, reflecting their more balanced, natural lighting.

Model Setup

- InstanceNorm: Normalizes each sample independently across spatial dimensions
- Residual Block: A pair of convolutional layers with a skip connection that lets the input flow directly to the output
- ResNet Generator: A generator built from several residual blocks, allowing it to learn complex transformations
- Discriminator: A CNN that judges whether an image is real or generated, guiding the generator by providing feedback on how realistic its outputs appear

```python
# InstanceNorm
class InstanceNormalization(layers.Layer):
    def __init__(self, epsilon=1e-5, **kwargs):
        super().__init__(**kwargs)
        self.epsilon = epsilon

    def build(self, input_shape):
        channels = input_shape[-1]
        self.gamma = self.add_weight(
            shape=(channels,), initializer="ones", trainable=True, name="gamma"
        )
        self.beta = self.add_weight(
            shape=(channels,), initializer="zeros", trainable=True, name="beta"
        )
        super().build(input_shape)

    def call(self, x):
        mean, var = tf.nn.moments(x, axes=[1, 2], keepdims=True)
        inv = tf.math.rsqrt(var + self.epsilon)
        x_norm = (x - mean) * inv
        return self.gamma * x_norm + self.beta
```

```python
# Residual Block
def residual_block(x, filters, use_norm=True, name=None):
    init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)
    skip = x

    y = layers.Conv2D(
        filters, 3, strides=1, padding="same",
        kernel_initializer=init, use_bias=not use_norm,
        name=None if name is None else name + "_conv1",
    )(x)
    if use_norm:
        y = InstanceNormalization(name=None if name is None else name + "_in1")(y)
    y = layers.ReLU()(y)

    y = layers.Conv2D(
        filters, 3, strides=1, padding="same",
        kernel_initializer=init, use_bias=not use_norm,
        name=None if name is None else name + "_conv2",
    )(y)
    if use_norm:
        y = InstanceNormalization(name=None if name is None else name + "_in2")(y)

    out = layers.Add(name=None if name is None else name + "_add")([skip, y])
    return out
```

```python
# ResNet Generator
def build_resnet_generator(name="generator", n_res_blocks=9):
    init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)
    inputs = keras.Input(shape=(IMG_HEIGHT, IMG_WIDTH, CHANNELS))

    x = layers.Conv2D(
        64, 7, strides=1, padding="same",
        kernel_initializer=init, use_bias=False,
    )(inputs)
    x = InstanceNormalization()(x)
    x = layers.ReLU()(x)

    for filters in [128, 256]:
        x = layers.Conv2D(
            filters, 3, strides=2, padding="same",
            kernel_initializer=init, use_bias=False,
        )(x)
        x = InstanceNormalization()(x)
        x = layers.ReLU()(x)

    for i in range(n_res_blocks):
        x = residual_block(x, 256, name=f"res{i+1}")

    for filters in [128, 64]:
        x = layers.Conv2DTranspose(
            filters, 3, strides=2, padding="same",
            kernel_initializer=init, use_bias=False,
        )(x)
        x = InstanceNormalization()(x)
        x = layers.ReLU()(x)

    x = layers.Conv2D(
        CHANNELS, 7, strides=1, padding="same",
        kernel_initializer=init,
    )(x)
    outputs = layers.Activation("tanh")(x)
    return keras.Model(inputs, outputs, name=name)
```

```python
# Discriminator
def build_discriminator(name="discriminator"):
    init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)
    inputs = keras.Input(shape=(IMG_HEIGHT, IMG_WIDTH, CHANNELS))

    def disc_block(x, filters, stride, use_norm=True):
        x = layers.Conv2D(
            filters, 4, strides=stride, padding="same",
            kernel_initializer=init, use_bias=not use_norm,
        )(x)
        if use_norm:
            x = InstanceNormalization()(x)
        x = layers.LeakyReLU(0.2)(x)
        return x

    x = disc_block(inputs, 64, stride=2, use_norm=False)
    x = disc_block(x, 128, stride=2)
    x = disc_block(x, 256, stride=2)
    x = disc_block(x, 512, stride=1)
    x = layers.Conv2D(1, 4, strides=1, padding="same", kernel_initializer=init)(x)
    return keras.Model(inputs, x, name=name)

with strategy.scope():
    G_photo_to_monet = build_resnet_generator(name="G_photo_to_monet")
    G_monet_to_photo = build_resnet_generator(name="G_monet_to_photo")

    D_monet = build_discriminator(name="D_monet")
    D_photo = build_discriminator(name="D_photo")

    mse_loss = keras.losses.MeanSquaredError()
    mae_loss = keras.losses.MeanAbsoluteError()

    def generator_adversarial_loss(fake_logits):
        return mse_loss(tf.ones_like(fake_logits), fake_logits)

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

    def cycle_consistency_loss(real, cycled):
        return mae_loss(real, cycled)

    def identity_loss(real, same):
        return mae_loss(real, same)

    gen_G_optimizer = keras.optimizers.Adam(LR, beta_1=0.5, beta_2=0.999)
    gen_F_optimizer = keras.optimizers.Adam(LR, beta_1=0.5, beta_2=0.999)
    disc_monet_optimizer = keras.optimizers.Adam(LR, beta_1=0.5, beta_2=0.999)
    disc_photo_optimizer = keras.optimizers.Adam(LR, beta_1=0.5, beta_2=0.999)
```

```
I0000 00:00:1764191472.260430     522 gpu_device.cc:2022] Created device /job:localhost/replica:0/task:0/device:GPU:0 with 13942 M
B memory:  -> device: 0, name: Tesla T4, pci bus id: 0000:00:04.0, compute capability: 7.5
I0000 00:00:1764191472.261035     522 gpu_device.cc:2022] Created device /job:localhost/replica:0/task:0/device:GPU:1 with 13942 M
B memory:  -> device: 1, name: Tesla T4, pci bus id: 0000:00:05.0, compute capability: 7.5
Steps per epoch: 300
```

Model Training

```python
# Training
def train_step(batch):
    real_photo, real_monet = batch

    with tf.GradientTape(persistent=True) as tape:
        fake_monet = G_photo_to_monet(real_photo, training=True)
        fake_photo = G_monet_to_photo(real_monet, training=True)

        cycled_photo = G_monet_to_photo(fake_monet, training=True)
        cycled_monet = G_photo_to_monet(fake_photo, training=True)

        same_monet = G_photo_to_monet(real_monet, training=True)
        same_photo = G_monet_to_photo(real_photo, training=True)

        disc_real_monet = D_monet(real_monet, training=True)
        disc_fake_monet = D_monet(fake_monet, training=True)
        disc_real_photo = D_photo(real_photo, training=True)
        disc_fake_photo = D_photo(fake_photo, training=True)

        gen_G_adv = generator_adversarial_loss(disc_fake_monet)
        gen_F_adv = generator_adversarial_loss(disc_fake_photo)

        cycle_loss_photo = cycle_consistency_loss(real_photo, cycled_photo)
        cycle_loss_monet = cycle_consistency_loss(real_monet, cycled_monet)
        total_cycle_loss = cycle_loss_photo + cycle_loss_monet

        id_loss_monet = identity_loss(real_monet, same_monet)
        id_loss_photo = identity_loss(real_photo, same_photo)

        total_gen_G_loss = gen_G_adv + LAMBDA_CYCLE * total_cycle_loss + LAMBDA_ID * id_loss_monet
        total_gen_F_loss = gen_F_adv + LAMBDA_CYCLE * total_cycle_loss + LAMBDA_ID * id_loss_photo

        disc_monet_loss = discriminator_loss(disc_real_monet, disc_fake_monet)
        disc_photo_loss = discriminator_loss(disc_real_photo, disc_fake_photo)

    gen_G_gradients = tape.gradient(total_gen_G_loss, G_photo_to_monet.trainable_variables)
    gen_F_gradients = tape.gradient(total_gen_F_loss, G_monet_to_photo.trainable_variables)
    disc_monet_gradients = tape.gradient(disc_monet_loss, D_monet.trainable_variables)
    disc_photo_gradients = tape.gradient(disc_photo_loss, D_photo.trainable_variables)

    gen_G_optimizer.apply_gradients(zip(gen_G_gradients, G_photo_to_monet.trainable_variables))
    gen_F_optimizer.apply_gradients(zip(gen_F_gradients, G_monet_to_photo.trainable_variables))
    disc_monet_optimizer.apply_gradients(zip(disc_monet_gradients, D_monet.trainable_variables))
    disc_photo_optimizer.apply_gradients(zip(disc_photo_gradients, D_photo.trainable_variables))

    del tape

    return {
        "gen_G_loss": float(total_gen_G_loss.numpy()),
        "gen_F_loss": float(total_gen_F_loss.numpy()),
        "disc_monet_loss": float(disc_monet_loss.numpy()),
        "disc_photo_loss": float(disc_photo_loss.numpy()),
    }
```

```python
# Training Loop
from time import time

history = {
    "gen_G_loss": [],
    "gen_F_loss": [],
    "disc_monet_loss": [],
    "disc_photo_loss": [],
}

print("Starting training...")

for epoch in range(1, EPOCHS + 1):
    start_time = time()
    epoch_metrics = {
        "gen_G_loss": 0.0,
        "gen_F_loss": 0.0,
        "disc_monet_loss": 0.0,
        "disc_photo_loss": 0.0,
    }
    for step, batch in enumerate(train_ds.take(STEPS_PER_EPOCH)):
        metrics = train_step(batch)
        for k in epoch_metrics:
            epoch_metrics[k] += metrics[k]

        if (step + 1) % 10 == 0 or (step + 1) == STEPS_PER_EPOCH:
            print(
                f"Epoch [{epoch}/{EPOCHS}] "
                f"Step [{step+1}/{STEPS_PER_EPOCH}] "
                f"G_G: {metrics['gen_G_loss']:.2f} "
                f"G_F: {metrics['gen_F_loss']:.2f} "
                f"D_M: {metrics['disc_monet_loss']:.2f} "
                f"D_P: {metrics['disc_photo_loss']:.2f}",
                end="\r",
            )

    epoch_time = time() - start_time
    for k in epoch_metrics:
        epoch_metrics[k] /= STEPS_PER_EPOCH
        history[k].append(epoch_metrics[k])

    print(
        f"\nEpoch {epoch}/{EPOCHS} time: {epoch_time:.1f}s | "
        f"G_G: {epoch_metrics['gen_G_loss']:.3f} "
        f"G_F: {epoch_metrics['gen_F_loss']:.3f} "
        f"D_M: {epoch_metrics['disc_monet_loss']:.3f} "
        f"D_P: {epoch_metrics['disc_photo_loss']:.3f}"
    )

print("Training finished.")
```

```
Starting training...
I0000 00:00:1764271172.031451     220 cuda_dnn.cc:529] Loaded cuDNN version 90300
Epoch [1/10] Step [300/300] G_G: 8.00 G_F: 8.31 D_M: 0.26 D_P: 0.2141
Epoch 1/10 time: 854.5s | G_G: 8.259 G_F: 8.467 D_M: 0.419 D_P: 0.498
Epoch [2/10] Step [300/300] G_G: 6.27 G_F: 6.44 D_M: 0.19 D_P: 0.18
Epoch 2/10 time: 848.1s | G_G: 7.261 G_F: 7.270 D_M: 0.264 D_P: 0.269
Epoch [3/10] Step [300/300] G_G: 5.35 G_F: 5.63 D_M: 0.25 D_P: 0.28
Epoch 3/10 time: 846.1s | G_G: 6.823 G_F: 6.832 D_M: 0.242 D_P: 0.261
Epoch [4/10] Step [300/300] G_G: 4.94 G_F: 4.51 D_M: 0.16 D_P: 0.1613
Epoch 4/10 time: 846.9s | G_G: 6.573 G_F: 6.582 D_M: 0.247 D_P: 0.252
Epoch [5/10] Step [300/300] G_G: 6.57 G_F: 6.22 D_M: 0.21 D_P: 0.19
Epoch 5/10 time: 841.2s | G_G: 6.173 G_F: 6.294 D_M: 0.229 D_P: 0.241
Epoch [6/10] Step [300/300] G_G: 4.99 G_F: 4.96 D_M: 0.38 D_P: 0.24
Epoch 6/10 time: 838.9s | G_G: 5.908 G_F: 6.036 D_M: 0.219 D_P: 0.234
Epoch [7/10] Step [300/300] G_G: 5.19 G_F: 5.35 D_M: 0.10 D_P: 0.13
Epoch 7/10 time: 839.3s | G_G: 5.574 G_F: 5.619 D_M: 0.225 D_P: 0.231
Epoch [8/10] Step [300/300] G_G: 4.28 G_F: 4.44 D_M: 0.26 D_P: 0.39
Epoch 8/10 time: 843.8s | G_G: 5.330 G_F: 5.360 D_M: 0.227 D_P: 0.227
Epoch [9/10] Step [300/300] G_G: 4.84 G_F: 4.75 D_M: 0.14 D_P: 0.40
Epoch 9/10 time: 840.1s | G_G: 5.134 G_F: 5.183 D_M: 0.225 D_P: 0.222
Epoch [10/10] Step [300/300] G_G: 4.51 G_F: 4.37 D_M: 0.18 D_P: 0.17
Epoch 10/10 time: 838.6s | G_G: 5.133 G_F: 5.206 D_M: 0.227 D_P: 0.229
Training finished.
```

Generate file for Kaggle competition

```
# Generate Monet-style images into images.zip
OUTPUT_DIR = Path("/kaggle/working")
ZIP_PATH = OUTPUT_DIR / "images.zip"

def denormalize_to_uint8(image):
    image = (image + 1.0) * 0.5
    image = tf.clip_by_value(image, 0.0, 1.0)
    image = tf.image.convert_image_dtype(image, dtype=tf.uint8)
    return image

print("Generating Monet-style images into images.zip ...")

with zipfile.ZipFile(ZIP_PATH, mode="w", compression=zipfile.ZIP_DEFLATED) as zf:
    idx = 0
    for batch in photo_ds_inference:
        if idx >= N_GENERATED_IMAGES:
            break
        photo = batch
        fake_monet = G_photo_to_monet(photo, training=False)[0]
        img_uint8 = denormalize_to_uint8(fake_monet)
        jpg_bytes = tf.io.encode_jpeg(img_uint8).numpy()

        filename = f"image_{idx:05d}.jpg"
        zf.writestr(filename, jpg_bytes)
        idx += 1

        if idx % 100 == 0:
            print(f"Generated {idx}/{N_GENERATED_IMAGES} images...", end="\r")

print(f"\nDone. Wrote {idx} images to {ZIP_PATH}.")
```

```
Generating Monet-style images into images.zip ...
Generated 7000/7000 images...
Done. Wrote 7000 images to /kaggle/working/images.zip.
```

Submission Results

Kaggle score "notebook03b8491926 - Version3": 84.30637

Conclusions

- The GAN pipeline successfully generated Monet-style images and achieved a competitive score on the Kaggle leaderboard.
- The training process demonstrated stable adversarial learning.
- The results suggest that further gains are possible through hyperparameter tuning—such as adjusting learning rates, experimenting with alternative GAN losses, or increasing model capacity.

Discussion

- The notebook is running in the Kaggle environment. It is not necessary to download any files.
  You can download the files if you want to create a notebook which is running locally on your PC.
- Hyperparameter Tuning which could be done:
  - More epochs
  - Different Learning Rates for Generator and/or Discrimninator
  - Larger batches
  - Other loss functions
  - Different number of filters
  - Image Augmentation

Link GitHub Repository

https://github.com/Oliver-VG/GAN-Monet-Kaggle-Mini-Project

Citation / References

- Kaggle competition: https://www.kaggle.com/competitions/gan-getting-started/overview
- https://arxiv.org/abs/1406.2661

AI Acknowledgement

ChatGPT-5.1 (OpenAI, 2025) was used to assist in proofreading and improving the grammatical accuracy of the Markdown content. No substantive changes to the original ideas or analysis were made by the model.