# DTSA-5511 Week 5 - I'm Something of a Painter Myself

A Jupyter notebook with a description of the problem/data, exploratory data analysis (EDA) procedure, analysis (model building and training), result, and discussion/conclusion.

## Project description

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.

## Dataset Description

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.

The photo directories contain photos.

### Files

monet_jpg - 300 Monet paintings sized 256x256 in JPEG format monet_tfrec - 300 Monet paintings sized 256x256 in TFRecord format photo_jpg - 7028 photos sized 256x256 in JPEG format photo_tfrec - 7028 photos sized 256x256 in TFRecord format

## 1. Exploratory Data Analysis (EDA)

### 1.1. Importing libraries

Loading libraries required for this mini-project.

In [2]:
```
#utilities
import numpy as np
import random
import re
import pandas as pd
```

```
import PIL

#plots
import matplotlib.pyplot as plt

# modeling
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import tensorflow_addons as tfa

from kaggle_datasets import KaggleDatasets

# Remove warnings
import warnings
warnings.filterwarnings('ignore')

# detect TPU device
try:
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver()
    print('Device:', tpu.master())
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.experimental.TPUStrategy(tpu)
except:
    strategy = tf.distribute.get_strategy()
print('Number of replicas:', strategy.num_replicas_in_sync)

# dynamically tune value at runtime
AUTOTUNE = tf.data.experimental.AUTOTUNE

# tensorflow version
print(tf.__version__)
```

```
Number of replicas: 1
2.6.4
```

## 1.2. TFRecords

Reading the photo and Monet images on the TFRecords files and creating functions for loading those images.

In [3]:

```
IMAGE_SIZE = [256, 256]
BATCH_SIZE = 16

# Load the data
gcs_path = KaggleDatasets().get_gcs_path()

monet_files = tf.io.gfile.glob(str(gcs_path + '/monet_tfrec/*.tfrec'))
print('Monet TFRecord Files:', len(monet_files))

photo_files = tf.io.gfile.glob(str(gcs_path + '/photo_tfrec/*.tfrec'))
print('Photo TFRecord Files:', len(photo_files))
```

```
2022-10-02 22:36:53.417069: W tensorflow/core/platform/cloud/google_auth_provider.cc:18
4] All attempts to get a Google authentication bearer token failed, returning an empty t
oken. Retrieving token from files failed with "Not found: Could not locate the credentia
ls file.". Retrieving token from GCE failed with "Failed precondition: Error executing a
```

```
n HTTP request: libcurl code 6 meaning 'Couldn't resolve host name', error details: Coul
d not resolve host: metadata".
Monet TFRecord Files: 5
Photo TFRecord Files: 20
```

In [4]:
```
# Function to count the number of photo and Monet images
def count_data_items(filenames):
    n = [int(re.compile(r"-([0-9]*)\.").search(filename).group(1)) for filename in file
    return np.sum(n)

n_monet_samples = count_data_items(monet_files)
n_photo_samples = count_data_items(photo_files)
```

In [5]:
```
# Function to decode a JPEG-encoded image to an RGB channel
def decode_image(image):
    image = tf.image.decode_jpeg(image, channels=3)
    image = (tf.cast(image, tf.float32) / 127.5) - 1
    image = tf.reshape(image, [*IMAGE_SIZE, 3])
    return image

# Function to parse a simple example photo to decode the image
def read_tfrecord(example):
    tfrecord_format = {
        "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, tfrecord_format)
    image = decode_image(example['image'])
    return image

# Function to extract image from files
def load_dataset(filenames, labeled=True, ordered=False):
    dataset = tf.data.TFRecordDataset(filenames)
    dataset = dataset.map(read_tfrecord, num_parallel_calls=AUTOTUNE)
    return dataset
```

## 1.3. Image visualization

Visualizing three samples of photo and Monet images.

In [6]:
```
# Loading the Monet images
monet_ds = load_dataset(monet_files, labeled=True).batch(10)
# Loading the photo images
photo_ds = load_dataset(photo_files, labeled=True).batch(10)
```

```
2022-10-02 22:37:02.897438: I tensorflow/core/common_runtime/process_util.cc:146] Creati
ng new thread pool with default inter op setting: 2. Tune using inter_op_parallelism_thr
eads for best performance.
```

In [7]:
```
example_monet = next(iter(monet_ds))
example_photo = next(iter(photo_ds))
```

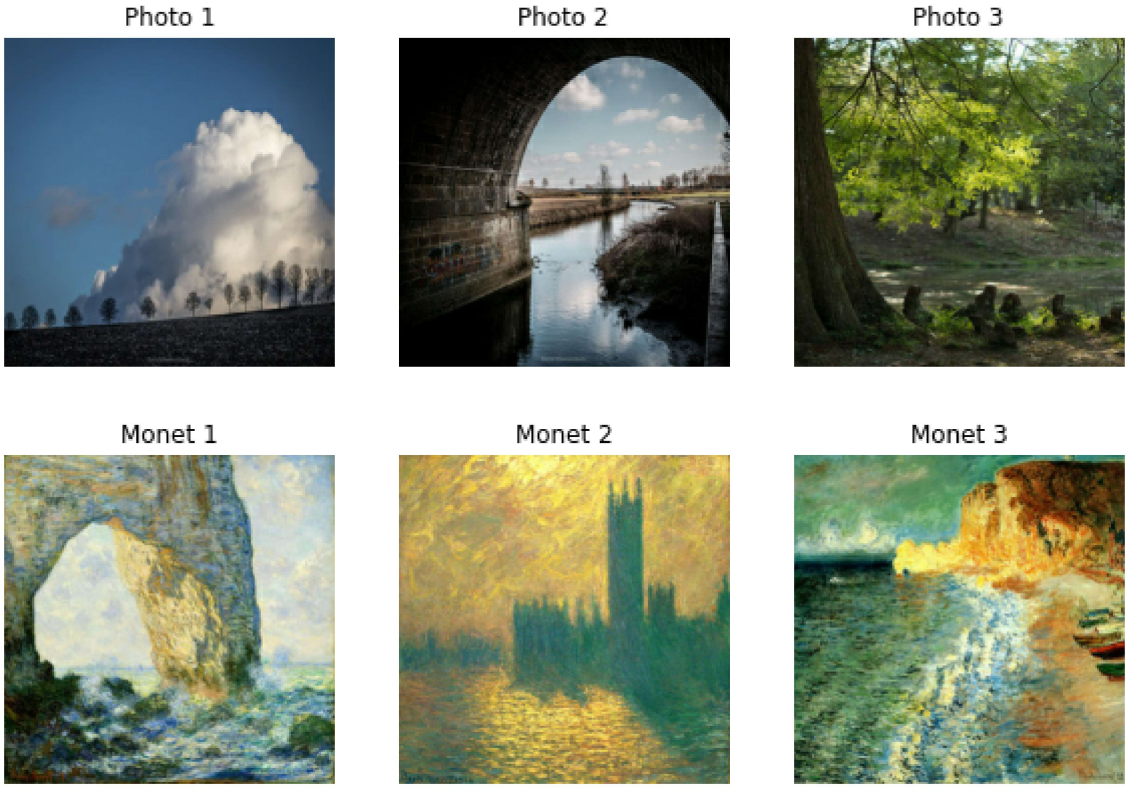
```
2022-10-02 22:37:09.660580: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:1
```

85] None of the MLIR Optimization Passes are enabled (registered 2)

In [8]:

```
#number = random.randint(0, 9)
plt.figure(figsize=(10, 7))

plt.subplot(231)
plt.title('Photo 1')
plt.axis('off')
plt.imshow(example_photo[0] * 0.5 + 0.5)
plt.subplot(232)
plt.axis('off')
plt.title('Photo 2')
plt.imshow(example_photo[1] * 0.5 + 0.5)
plt.subplot(233)
plt.axis('off')
plt.title('Photo 3')
plt.imshow(example_photo[2] * 0.5 + 0.5)

plt.subplot(234)
plt.title('Monet 1')
plt.axis('off')
plt.imshow(example_monet[0] * 0.5 + 0.5)
plt.subplot(235)
plt.title('Monet 2')
plt.axis('off')
plt.imshow(example_monet[1] * 0.5 + 0.5)
plt.subplot(236)
plt.title('Monet 3')
plt.axis('off')
plt.imshow(example_monet[2] * 0.5 + 0.5)
```

Out[8]: <matplotlib.image.AxesImage at 0x7f46582fb5d0>



# 2. CycleGAN

Implementing CycleGAN because it is a very popular GAN architecture used to learn transformation between images of different styles.

## 2.1. Building RestNet architecture

This architecture introduces the concept of Residual Blocks to solve the problem of vanishing/exploring gradient.

In [9]:

```
OUTPUT_CHANNELS = 3

# Reducing the features of an array or an image.
def downsample(filters, size, apply_instancenorm=True):
    initializer = tf.random_normal_initializer(0., 0.02)
    gamma_init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)

    result = keras.Sequential()
    result.add(layers.Conv2D(filters, size, strides=2, padding='same',
                             kernel_initializer=initializer, use_bias=False))

    if apply_instancenorm:
        result.add(tfa.layers.InstanceNormalization(gamma_initializer=gamma_init))

    result.add(layers.LeakyReLU())

    return result

# Inserting null-values between original values to increase the sampling rate
def upsample(filters, size, apply_dropout=False):
    initializer = tf.random_normal_initializer(0., 0.02)
    gamma_init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)

    result = keras.Sequential()
    result.add(layers.Conv2DTranspose(filters, size, strides=2,
                                      padding='same',
                                      kernel_initializer=initializer,
                                      use_bias=False))

    result.add(tfa.layers.InstanceNormalization(gamma_initializer=gamma_init))

    if apply_dropout:
        result.add(layers.Dropout(0.5))

    result.add(layers.ReLU())

    return result

# Generating ResNet
def ResNetGenerator():
    inputs = layers.Input(shape=[256,256,3])

    # bs = batch size
    down_stack = [
        downsample(64, 4, apply_instancenorm=False), # (bs, 128, 128, 64)
        downsample(128, 4), # (bs, 64, 64, 128)
```

```
        downsample(256, 4), # (bs, 32, 32, 256)
        downsample(512, 4), # (bs, 16, 16, 512)
        downsample(512, 4), # (bs, 8, 8, 512)
        downsample(512, 4), # (bs, 4, 4, 512)
        downsample(512, 4), # (bs, 2, 2, 512)
        downsample(512, 4), # (bs, 1, 1, 512)
    ]

    up_stack = [
        upsample(512, 4, apply_dropout=True), # (bs, 2, 2, 1024)
        upsample(512, 4, apply_dropout=True), # (bs, 4, 4, 1024)
        upsample(512, 4, apply_dropout=True), # (bs, 8, 8, 1024)
        upsample(512, 4), # (bs, 16, 16, 1024)
        upsample(256, 4), # (bs, 32, 32, 512)
        upsample(128, 4), # (bs, 64, 64, 256)
        upsample(64, 4), # (bs, 128, 128, 128)
    ]

    initializer = tf.random_normal_initializer(0., 0.02)
    last = layers.Conv2DTranspose(OUTPUT_CHANNELS, 4,
                                  strides=2,
                                  padding='same',
                                  kernel_initializer=initializer,
                                  activation='tanh') # (bs, 256, 256, 3)

    x = inputs

    # Downsampling through the model
    skips = []
    for down in down_stack:
        x = down(x)
        skips.append(x)

    skips = reversed(skips[:-1])

    # Upsampling and establishing the skip connections
    for up, skip in zip(up_stack, skips):
        x = up(x)
        x = layers.Concatenate()([x, skip])

    x = last(x)

    return keras.Model(inputs=inputs, outputs=x)
```

In [10]:

```
#Building the ResNet and printing the model summary
generator = ResNetGenerator()
generator.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         [(None, 256, 256, 3)      0         
_________________________________________________________________
sequential (Sequential)      (None, 128, 128, 64)      3072      
```

| | | | |
|---|---|---|---|
| sequential_1 (Sequential) | (None, 64, 64, 128) | 131328 | sequential[0][0] |
| sequential_2 (Sequential) | (None, 32, 32, 256) | 524800 | sequential_1[0][0] |
| sequential_3 (Sequential) | (None, 16, 16, 512) | 2098176 | sequential_2[0][0] |
| sequential_4 (Sequential) | (None, 8, 8, 512) | 4195328 | sequential_3[0][0] |
| sequential_5 (Sequential) | (None, 4, 4, 512) | 4195328 | sequential_4[0][0] |
| sequential_6 (Sequential) | (None, 2, 2, 512) | 4195328 | sequential_5[0][0] |
| sequential_7 (Sequential) | (None, 1, 1, 512) | 4195328 | sequential_6[0][0] |
| sequential_8 (Sequential) | (None, 2, 2, 512) | 4195328 | sequential_7[0][0] |
| concatenate (Concatenate) | (None, 2, 2, 1024) | 0 | sequential_8[0][0]<br>sequential_6[0][0] |
| sequential_9 (Sequential) | (None, 4, 4, 512) | 8389632 | concatenate[0][0] |
| concatenate_1 (Concatenate) | (None, 4, 4, 1024) | 0 | sequential_9[0][0]<br>sequential_5[0][0] |
| sequential_10 (Sequential) | (None, 8, 8, 512) | 8389632 | concatenate_1[0][0] |
| concatenate_2 (Concatenate) | (None, 8, 8, 1024) | 0 | sequential_10[0][0]<br>sequential_4[0][0] |
| sequential_11 (Sequential) | (None, 16, 16, 512) | 8389632 | concatenate_2[0][0] |
| concatenate_3 (Concatenate) | (None, 16, 16, 1024) | 0 | sequential_11[0][0]<br>sequential_3[0][0] |
| sequential_12 (Sequential) | (None, 32, 32, 256) | 4194816 | concatenate_3[0][0] |
| concatenate_4 (Concatenate) | (None, 32, 32, 512) | 0 | sequential_12[0][0]<br>sequential_2[0][0] |
| sequential_13 (Sequential) | (None, 64, 64, 128) | 1048832 | concatenate_4[0][0] |

```
__________
concatenate_5 (Concatenate)     (None, 64, 64, 256)  0           sequential_13[0][0]
                                                                 sequential_1[0][0]
__________
sequential_14 (Sequential)      (None, 128, 128, 64) 262272      concatenate_5[0][0]
__________
concatenate_6 (Concatenate)     (None, 128, 128, 128 0           sequential_14[0][0]
                                                                 sequential[0][0]
__________
conv2d_transpose_7 (Conv2DTrans (None, 256, 256, 3)  6147        concatenate_6[0][0]
==================================================================================================
Total params: 54,414,979
Trainable params: 54,414,979
Non-trainable params: 0
__________
```

## 2.2. Building the discriminator

The discriminator is used to distinguish real data from the data created by the generator.

In [11]:

```
# Function to create the discriminator
def Discriminator():
    initializer = tf.random_normal_initializer(0., 0.02)
    gamma_init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)

    inp = layers.Input(shape=[256, 256, 3], name='input_image')

    x = inp

    down1 = downsample(64, 4, False)(x) # (bs, 128, 128, 64)
    down2 = downsample(128, 4)(down1) # (bs, 64, 64, 128)
    down3 = downsample(256, 4)(down2) # (bs, 32, 32, 256)

    zero_pad1 = layers.ZeroPadding2D()(down3) # (bs, 34, 34, 256)
    conv = layers.Conv2D(512, 4, strides=1,
                         kernel_initializer=initializer,
                         use_bias=False)(zero_pad1) # (bs, 31, 31, 512)

    norm1 = tfa.layers.InstanceNormalization(gamma_initializer=gamma_init)(conv)

    leaky_relu = layers.LeakyReLU()(norm1)

    zero_pad2 = layers.ZeroPadding2D()(leaky_relu) # (bs, 33, 33, 512)

    last = layers.Conv2D(1, 4, strides=1,
                         kernel_initializer=initializer)(zero_pad2) # (bs, 30, 30, 1)

    return tf.keras.Model(inputs=inp, outputs=last)
```

In [12]:

```
# Generating the discriminator
discriminator = Discriminator()
```

```
discriminator.summary()
```

```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_image (InputLayer)     [(None, 256, 256, 3)]     0
_________________________________________________________________
sequential_15 (Sequential)   (None, 128, 128, 64)      3072
_________________________________________________________________
sequential_16 (Sequential)   (None, 64, 64, 128)       131328
_________________________________________________________________
sequential_17 (Sequential)   (None, 32, 32, 256)       524800
_________________________________________________________________
zero_padding2d (ZeroPadding2 (None, 34, 34, 256)       0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 31, 31, 512)       2097152
_________________________________________________________________
instance_normalization_16 (I (None, 31, 31, 512)       1024
_________________________________________________________________
leaky_re_lu_11 (LeakyReLU)   (None, 31, 31, 512)       0
_________________________________________________________________
zero_padding2d_1 (ZeroPaddin (None, 33, 33, 512)       0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 30, 30, 1)         8193
=================================================================
Total params: 2,765,569
Trainable params: 2,765,569
Non-trainable params: 0
_________________________________________________________________
```

In [13]:

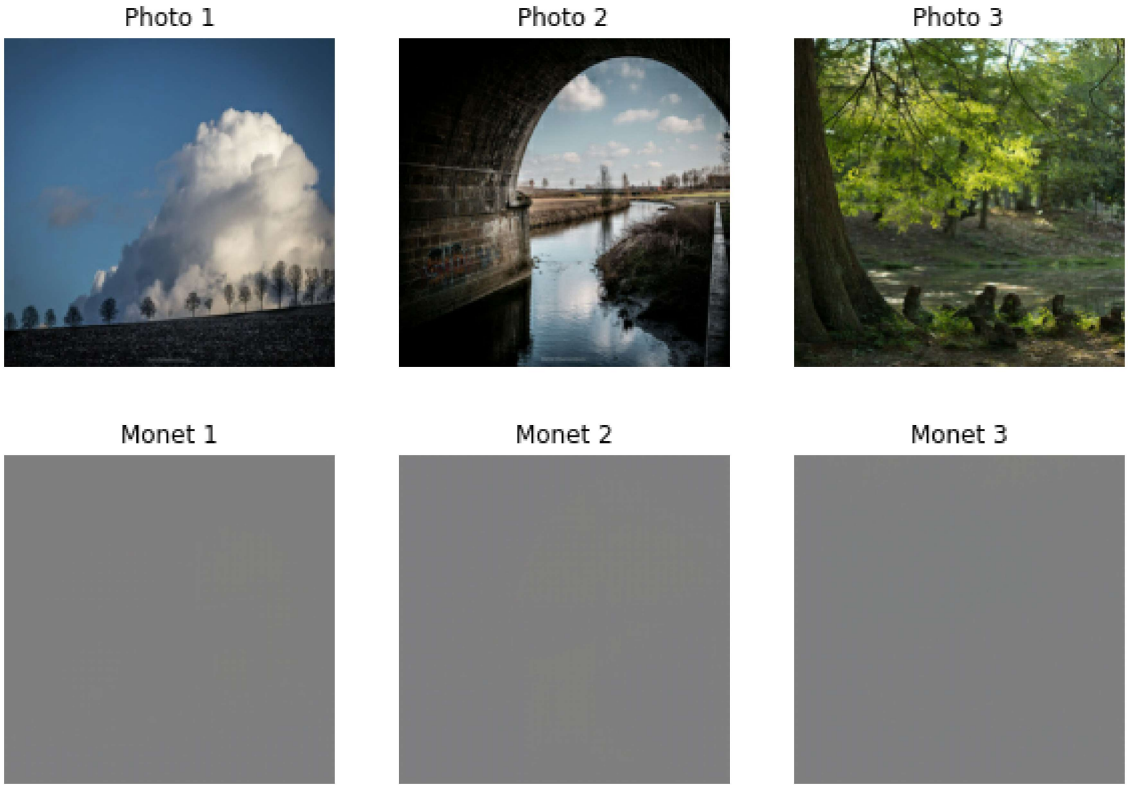
```
with strategy.scope():
    monet_generator = ResNetGenerator() # transforms photos to Monet-esque paintings
    photo_generator = ResNetGenerator() # transforms Monet paintings to be more like ph

    monet_discriminator = Discriminator() # differentiates real Monet paintings and gen
    photo_discriminator = Discriminator() # differentiates real photos and generated ph
```

In [14]:

```
# Visualization
plt.figure(figsize=(10, 7))
to_monet = monet_generator(example_photo)

plt.subplot(231)
plt.title('Photo 1')
plt.axis('off')
plt.imshow(example_photo[0] * 0.5 + 0.5)
plt.subplot(232)
plt.axis('off')
plt.title('Photo 2')
plt.imshow(example_photo[1] * 0.5 + 0.5)
plt.subplot(233)
plt.axis('off')
plt.title('Photo 3')
plt.imshow(example_photo[2] * 0.5 + 0.5)

plt.subplot(234)
plt.title('Monet 1')
plt.axis('off')
```

```
plt.imshow(to_monet[0] * 0.5 + 0.5)
plt.subplot(235)
plt.title('Monet 2')
plt.axis('off')
plt.imshow(to_monet[1] * 0.5 + 0.5)
plt.subplot(236)
plt.title('Monet 3')
plt.axis('off')
plt.imshow(to_monet[2] * 0.5 + 0.5)
```

Out[14]: <matplotlib.image.AxesImage at 0x7f46483de9d0>




Since our generators are not trained yet, the generated Monet-esque photo does not show what is expected at this point.

## 2.3. Building the cGAN model

Overriding the train_step() method of the Model class for training via fit().

In [15]:

```
# CycleGan class
class CycleGan(keras.Model):
    def __init__(
        self,
        monet_generator,
        photo_generator,
        monet_discriminator,
        photo_discriminator,
        lambda_cycle=10,
    ):
        super(CycleGan, self).__init__()
        self.m_gen = monet_generator
        self.p_gen = photo_generator
```

```
        self.m_disc = monet_discriminator
        self.p_disc = photo_discriminator
        self.lambda_cycle = lambda_cycle

    def compile(
        self,
        m_gen_optimizer,
        p_gen_optimizer,
        m_disc_optimizer,
        p_disc_optimizer,
        gen_loss_fn,
        disc_loss_fn,
        cycle_loss_fn,
        identity_loss_fn
    ):
        super(CycleGan, self).compile()
        self.m_gen_optimizer = m_gen_optimizer
        self.p_gen_optimizer = p_gen_optimizer
        self.m_disc_optimizer = m_disc_optimizer
        self.p_disc_optimizer = p_disc_optimizer
        self.gen_loss_fn = gen_loss_fn
        self.disc_loss_fn = disc_loss_fn
        self.cycle_loss_fn = cycle_loss_fn
        self.identity_loss_fn = identity_loss_fn

    def train_step(self, batch_data):
        real_monet, real_photo = batch_data

        with tf.GradientTape(persistent=True) as tape:
            # photo to monet back to photo
            fake_monet = self.m_gen(real_photo, training=True)
            cycled_photo = self.p_gen(fake_monet, training=True)

            # monet to photo back to monet
            fake_photo = self.p_gen(real_monet, training=True)
            cycled_monet = self.m_gen(fake_photo, training=True)

            # generating itself
            same_monet = self.m_gen(real_monet, training=True)
            same_photo = self.p_gen(real_photo, training=True)

            # discriminator used to check, inputing real images
            disc_real_monet = self.m_disc(real_monet, training=True)
            disc_real_photo = self.p_disc(real_photo, training=True)

            # discriminator used to check, inputing fake images
            disc_fake_monet = self.m_disc(fake_monet, training=True)
            disc_fake_photo = self.p_disc(fake_photo, training=True)

            # evaluates generator loss
            monet_gen_loss = self.gen_loss_fn(disc_fake_monet)
            photo_gen_loss = self.gen_loss_fn(disc_fake_photo)

            # evaluates total cycle consistency loss
            total_cycle_loss = self.cycle_loss_fn(real_monet, cycled_monet, self.lambda

            # evaluates total generator loss
            total_monet_gen_loss = monet_gen_loss + total_cycle_loss + self.identity_lo
            total_photo_gen_loss = photo_gen_loss + total_cycle_loss + self.identity_lo
```

```
            # evaluates discriminator loss
            monet_disc_loss = self.disc_loss_fn(disc_real_monet, disc_fake_monet)
            photo_disc_loss = self.disc_loss_fn(disc_real_photo, disc_fake_photo)

        # Calculate the gradients for generator and discriminator
        monet_generator_gradients = tape.gradient(total_monet_gen_loss,
                                                  self.m_gen.trainable_variables)
        photo_generator_gradients = tape.gradient(total_photo_gen_loss,
                                                  self.p_gen.trainable_variables)

        monet_discriminator_gradients = tape.gradient(monet_disc_loss,
                                                      self.m_disc.trainable_variables)
        photo_discriminator_gradients = tape.gradient(photo_disc_loss,
                                                      self.p_disc.trainable_variables)

        # Apply the gradients to the optimizer
        self.m_gen_optimizer.apply_gradients(zip(monet_generator_gradients,
                                                 self.m_gen.trainable_variables))

        self.p_gen_optimizer.apply_gradients(zip(photo_generator_gradients,
                                                 self.p_gen.trainable_variables))

        self.m_disc_optimizer.apply_gradients(zip(monet_discriminator_gradients,
                                                  self.m_disc.trainable_variables))

        self.p_disc_optimizer.apply_gradients(zip(photo_discriminator_gradients,
                                                  self.p_disc.trainable_variables))

        return {
            "monet_gen_loss": total_monet_gen_loss,
            "photo_gen_loss": total_photo_gen_loss,
            "monet_disc_loss": monet_disc_loss,
            "photo_disc_loss": photo_disc_loss
        }
```

# 3. Training

Fitting the best weights and biases to CGAN model to minimize the loss function over prediction range

In [16]:

```
with strategy.scope():
    # Returns the gradients of the loss with respect to the learnable parameters in the
    def discriminator_loss(real, generated):
        real_loss = tf.keras.losses.BinaryCrossentropy(from_logits=True, reduction=tf.k

        generated_loss = tf.keras.losses.BinaryCrossentropy(from_logits=True, reduction

        total_disc_loss = real_loss + generated_loss

        return total_disc_loss * 0.5

with strategy.scope():
    # Goes through the discriminator and gets classified as either “Real” or “Fake” bas
    def generator_loss(generated):
        return tf.keras.losses.BinaryCrossentropy(from_logits=True, reduction=tf.keras.
```

```
with strategy.scope():
    # Calculate the cycle consistency loss be finding the average of their difference.
    def calc_cycle_loss(real_image, cycled_image, LAMBDA):
        loss1 = tf.reduce_mean(tf.abs(real_image - cycled_image))

        return LAMBDA * loss1

with strategy.scope():
    # Compares the input with the output of the generator.
    def identity_loss(real_image, same_image, LAMBDA):
        loss = tf.reduce_mean(tf.abs(real_image - same_image))
        return LAMBDA * 0.5 * loss
```

In [17]:

```
with strategy.scope():
    monet_generator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
    photo_generator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)

    monet_discriminator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
    photo_discriminator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
```

In [18]:

```
with strategy.scope():
    cycle_gan_model = CycleGan(
        monet_generator, photo_generator, monet_discriminator, photo_discriminator
    )

    cycle_gan_model.compile(
        m_gen_optimizer = monet_generator_optimizer,
        p_gen_optimizer = photo_generator_optimizer,
        m_disc_optimizer = monet_discriminator_optimizer,
        p_disc_optimizer = photo_discriminator_optimizer,
        gen_loss_fn = generator_loss,
        disc_loss_fn = discriminator_loss,
        cycle_loss_fn = calc_cycle_loss,
        identity_loss_fn = identity_loss
    )
```

# 4. Prediction

Generating the Monet images

In [19]:

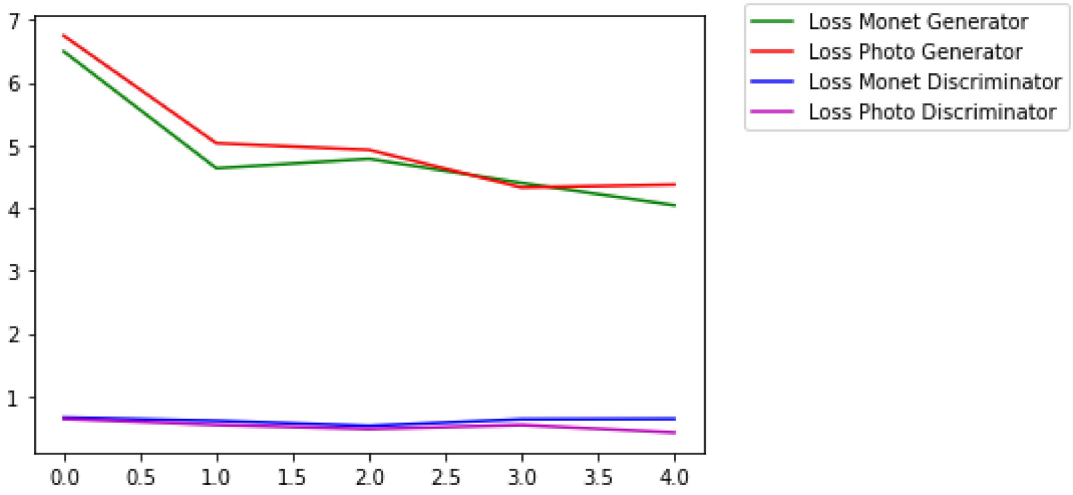
```
# Train the model
history1 = cycle_gan_model.fit(
    tf.data.Dataset.zip((monet_ds, photo_ds)),
    epochs=5
).history
```

```
Epoch 1/5
30/30 [==============================] - 1180s 37s/step - monet_gen_loss: 10.2063 - phot
o_gen_loss: 10.6039 - monet_disc_loss: 0.6622 - photo_disc_loss: 0.6612
Epoch 2/5
30/30 [==============================] - 1122s 37s/step - monet_gen_loss: 5.3387 - photo
_gen_loss: 5.4785 - monet_disc_loss: 0.6023 - photo_disc_loss: 0.5889
Epoch 3/5
30/30 [==============================] - 1119s 37s/step - monet_gen_loss: 5.2045 - photo
```

```
_gen_loss: 5.3080 - monet_disc_loss: 0.5204 - photo_disc_loss: 0.5182
Epoch 4/5
30/30 [==============================] - 1123s 37s/step - monet_gen_loss: 4.9569 - photo
_gen_loss: 5.0516 - monet_disc_loss: 0.5451 - photo_disc_loss: 0.5379
Epoch 5/5
30/30 [==============================] - 1119s 37s/step - monet_gen_loss: 4.5191 - photo
_gen_loss: 4.7154 - monet_disc_loss: 0.5969 - photo_disc_loss: 0.5141
```

In [20]:

```
# Chart to display the loss generator and loss discriminator for photo and Monet images
loss_results_df = pd.DataFrame(history1)
loss_results_df = loss_results_df.applymap(np.mean)

plt.plot(loss_results_df.index, loss_results_df['monet_gen_loss'], color='g', label='Lo
plt.plot(loss_results_df.index, loss_results_df['photo_gen_loss'], color='r', label='Lo
plt.plot(loss_results_df.index, loss_results_df['monet_disc_loss'], color='b', label='L
plt.plot(loss_results_df.index, loss_results_df['photo_disc_loss'], color='m', label='L
plt.legend(loc='upper center', bbox_to_anchor=(1.3, 1.05))
plt.show()
```




In [21]:

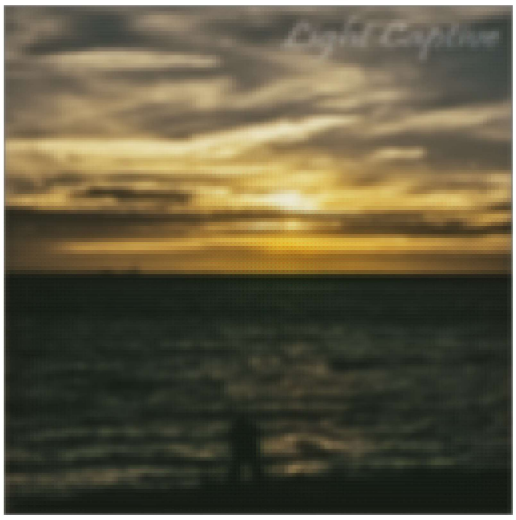
```
ds_iter = iter(photo_ds)
for n_sample in range(4):
        example_sample = next(ds_iter)
        generated_sample = monet_generator(example_sample)

        f = plt.figure(figsize=(10, 10))

        plt.subplot(121)
        plt.title('Input image')
        plt.imshow(example_sample[0] * 0.5 + 0.5)
        plt.axis('off')

        plt.subplot(122)
        plt.title('Generated image')
        plt.imshow(generated_sample[0] * 0.5 + 0.5)
        plt.axis('off')
        plt.show()
```

Input image

Generated image

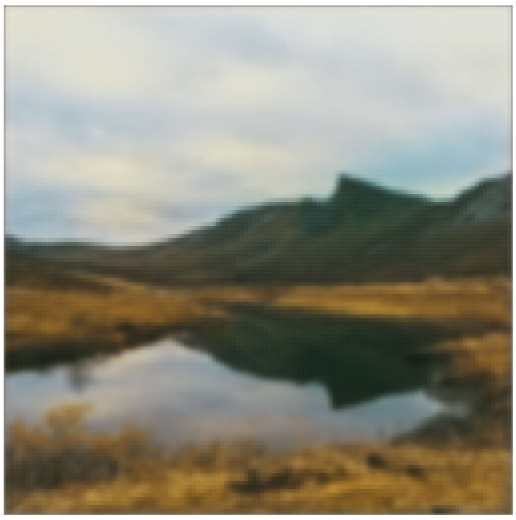
Input image

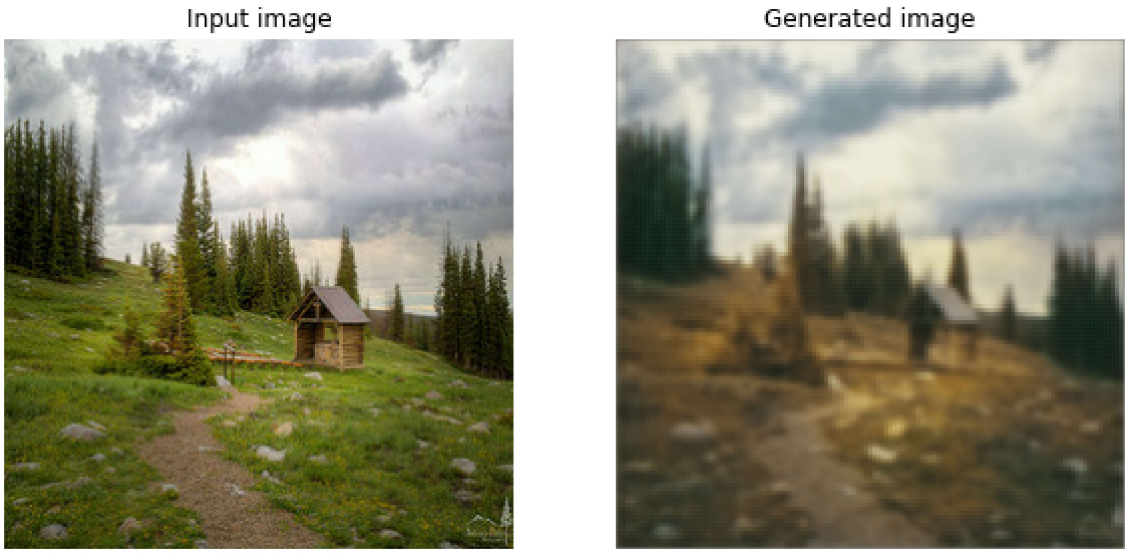

Generated image



Input image

Generated image






# 5. Submission

Generate the submission

In [24]:
```
! mkdir ../images
```

In [35]:
```
i = 1
for img in load_dataset(photo_files, labeled=True).batch(1):
    prediction = monet_generator(img, training=False)[0].numpy()
    prediction = (prediction * 127.5 + 127.5).astype(np.uint8)
    im = PIL.Image.fromarray(prediction)
    im.save("../images/" + str(i) + ".jpg")
    i += 1
print(i)
```

```
7039
```

In [37]:
```
import shutil
shutil.make_archive("/kaggle/working/images", 'zip', "/kaggle/images")
```

Out[37]:
```
'/kaggle/working/images.zip'
```

# 5. Discussion

## 5.1. Conclusion

CycleGAN is the type of GAN selected to convert photos to Monet images because this architecture is used to map between artistic and realistic images. I wanted to use initially 30 epochs but I chose 5 epochs because it takes long time to process. In addition, the 5 epochs used worked well to generate the Monet images when I checked visually.

## 5.2. Further improvement

For further improvement, I could increase the number of epochs and add Differentiable Augmentation (DiffAugment). DiffAug enables the gradients to be propagated through the augmentation back to the generator, regularizes the discriminator without manipulating the target distribution, and maintains the balance of training dynamics. Three choices of transformation I would like to experiments which are translation, cutOut, and color.