Monet_GAN_Project_Notebook

February 15, 2025

1 Section 1: GAN Monet Project Overview

Kaggle has an open competition which started in 2020 called, “I’m Something of a Painter Myself”. The challenge asked participants to develop a GAN (Generative Adversarial Network) which creates images in the style of Claude Monet, who is a well-known French painter from the Impressionist period.

Generative Adversarial Network (GAN): GANs are comprised of a generator to create the image and a discriminator to identify the fake images. Usage of the discriminator in training the neural network helps to improve the quality of the images generated. Thus, our challenge is to develop a generator which can create realistic Monet images – such that our discriminator cannot discern the difference between the real and fake Monet.

Nowadays, GANs can be used to create synthetic images, videos, songs, etc. – whether these creations are used in the medical industry, entertainment, gaming, marketing or other industries.

Description of Datasets:

- **Monet:** The Monet dataset contains 300 color images which are 256 by 256 pixels of real Monet paintings in JPEG or TFRecord format with an alphanumeric identification number.

- **Photo:** The photo dataset consists of 7038 color photos which are 256 by 256 pixels also in JPEG or TFRecord format with an alphanumeric identification number.

Note: Both JPG and TFRecord formats are provided, but only one format is necessary for the GAN. The alphanumeric identification numbers of the images are not utilized in our models.

Citation: Amy Jang, Ana Sofia Uzsoy, and Phil Culliton. I’m Something of a Painter Myself. https://kaggle.com/competitions/gan-getting-started, 2020. Kaggle.

Project Approach: For this assignment, we are asked to utilize the Monet and photo datasets summarized above to generate fake Monet-stylized images and submit to the competition. To build the GAN, we follow a modified U-Net architecture originally documented in a research paper by Ronnenberg, Fischer and Brox titled, “U-Net: Convolutional Networks for Biomedical Image Segmentation”. This architecture demonstrated good performance with small amounts of labeled images. Since we have 300 Monet images, this architecture appeared to be a good methodology for our project. More details on the architecture are provided below.

The images from our best performing GAN are submitted to the Kaggle competition page, where we screenshot the submission as asked for this project.

Github Link: https://github.com/chiffr3/GAN

1.1 Evaluation Metrics

Our primary task is image generation. The images from our best performing GAN were submitted to the Kaggle competition. The competition asks for submissions to contain 7000 to 10000 images sized 256 by 256 in JPEG format zipped in a file called 'images.zip'.

The Fréchet Inception Distance (FID) was introduced in 2017 as a method of calculating the similarity between real and generated images. Later, the Memorization-Informed Fréchet Inception Distance (MiFID) was introduced with a memorization penalty whereby the FID value is divided by the memorization term. The intention is include a penalty if the generated images are too similar to the training images.

- **MiFID:** The evaluation metric for this competition is MiFID (Memorization-Informed Fréchet Inception Distance). The goal is a smaller MiFID number, which indicates more realistic images being generated. The best performing model is identified, as well as the associated hyperparameters.

1.2 Summary of Results

Here is a summary of the top scores from models on the iterations. We walk through these models below. The results from the top model is shown in blue.

Table 1

Summary of Results for CGAN Models

Model	Monet Generator Loss	MiFID Score	Epochs	Optimizer	Learning Rate	Beta_1
Model 1	2.177	-	25	Adam	2e-4	0.8
Model 2	2.037	60.0	50	Adam	2e-4	0.8

Note: These models also utilized the same pre-processing methods on the images

Highlights from Table 1:

- The lowest loss scores noted in blue above were from "Model 2", which was used for the Kaggle competition
- As noted in Table 1, Model 1 actually contained the same hyperparameters as Model 2. However, Model 2 ran for twice as many epochs.
- The Learning Rate (LR) of 2e-4 was used for the Adam optimizer and generated better results than initial testing with the default of 1e-3 or the reduced LR of 1e-4.
- The execution time varied across these models and hyperparameters.

In summary, the data augmentation methods impacted the results. Therefore, these models were re-run with the same pre-processing methods to investigate the results. Those models performed better on the Monet Generator Loss and, as a result, are shown below.

2 Section 2: Import Packages and Load Data

[4]:
```
# Load packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pydot
import graphviz
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import shutil
import cv2
import re
from PIL import Image
from IPython.display import display
```

[2]:
```
# Import Tensorflow/Keras packages
import tensorflow as tf
from tensorflow import keras
from keras import layers #, ops
from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten,␣
 ↪Conv2D, Conv2DTranspose, MaxPooling2D, BatchNormalization, LeakyReLU, ReLU,␣
 ↪ZeroPadding2D
from tensorflow.keras.models import Sequential
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau,␣
 ↪ModelCheckpoint
from tensorflow.keras.regularizers import l2
from keras.optimizers import Adam, RMSprop #, AdamW
from tensorflow.python.ops.numpy_ops import np_config
np_config.enable_numpy_behavior()
#from keras.models import load_model
```

2.1 Load Data

We have already downloaded the zipped file from Kaggle and unzipped into a folder called GAN. Now, let's access the TFRecord files for the Monet and photo images.

[3]:
```
monet_files = tf.io.gfile.glob(str('GAN/monet_tfrec/*.tfrec'))
print("Size of Monet TFRecord Files:", len(monet_files))

photo_files = tf.io.gfile.glob(str('GAN/photo_tfrec/*.tfrec'))
print("Size of Photo TFRecord Files:", len(photo_files))
```

```
Size of Monet TFRecord Files: 5
Size of Photo TFRecord Files: 20
```

3 Section 3: Data Preparation

Scale Images: From the Kaggle description, we know these images are 256 by 256 pixels and are color or RGB (Red Green Blue) images. Therefore, below we use an image size of 256, 256 with channels set to 3 for RGB and we scale the images to -1, 1.

Data Augmentation: Also, we create a function called "random_jitter" below which resizes, crops and flips the images. Since the Monet dataset only contains 300 images, this random jitter can help augment our data.

[4]:
```
# Set image size to 256 by 256
IMG_SIZE = [256,256]

# create function to decode the image and scale to -1,1
def decode_img(image):
    image = tf.image.decode_jpeg(image, channels = 3)
    image = (tf.cast(image, tf.float32) / 127.5) - 1 #map values to [-1,1]
    image = tf.reshape(image, [*IMG_SIZE, 3])
    return image

# create function for random jitter
@ tf.function()
def random_jitter(input_img):
    # resize
    input_img = tf.image.resize(input_img, [286,286], method = tf.image.
 ↪ResizeMethod.NEAREST_NEIGHBOR)
    # crop back to 256,256
    input_img = tf.image.random_crop(input_img, size = [256, 256, 3])
    # flip images
    if tf.random.uniform(()) > 0.5:
        input_img = tf.image.flip_left_right(input_img)
    return input_img

# create function to read the TFRecord image format
def read_tfrecord(sample):
    tfr_format = {
        "image_name": tf.io.FixedLenFeature([], tf.string),
        "image": tf.io.FixedLenFeature([], tf.string),
        "target": tf.io.FixedLenFeature([], tf.string)
    }
    sample = tf.io.parse_single_example(sample, tfr_format)
    image = decode_img(sample['image'])
    image = random_jitter(image) # add random jitter
    return image
```

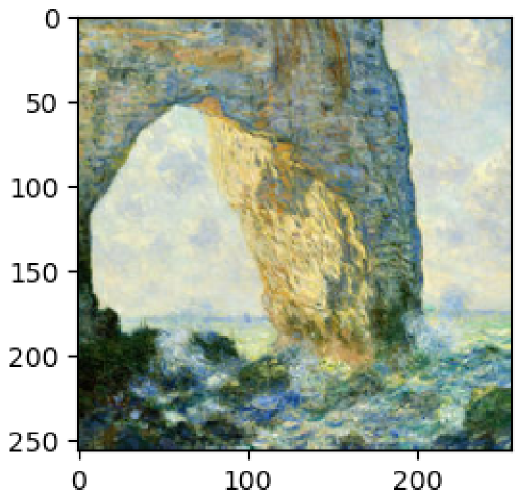
In these models, we do not need to keep the labels for these images. Therefore, we can create another function to just keep the images from our files.

[5]:
```
# Define function to keep the images
AUTOTUNE = tf.data.experimental.AUTOTUNE

def load_img(names, label = True, order = False):
    img_data = tf.data.TFRecordDataset(names)
    img_data = img_data.map(read_tfrecord, num_parallel_calls = AUTOTUNE)
    return img_data
```

Now, we should be ready to load and view a sample of our images.

[6]:
```
# Load images
monet_imgs = load_img(monet_files, label = True).batch(1)
photo_imgs = load_img(photo_files, label = True).batch(1)
```

[7]:
```
# check monet images loaded
ex_monet = next(iter(monet_imgs))
plt.subplot(122)
plt.imshow(ex_monet[0]*0.5+0.5)
plt.show()
```



4 Section 4: Exploratory Data Analysis

To start, let's confirm the count of images loaded into each folder

4.1 Verify Image Counts

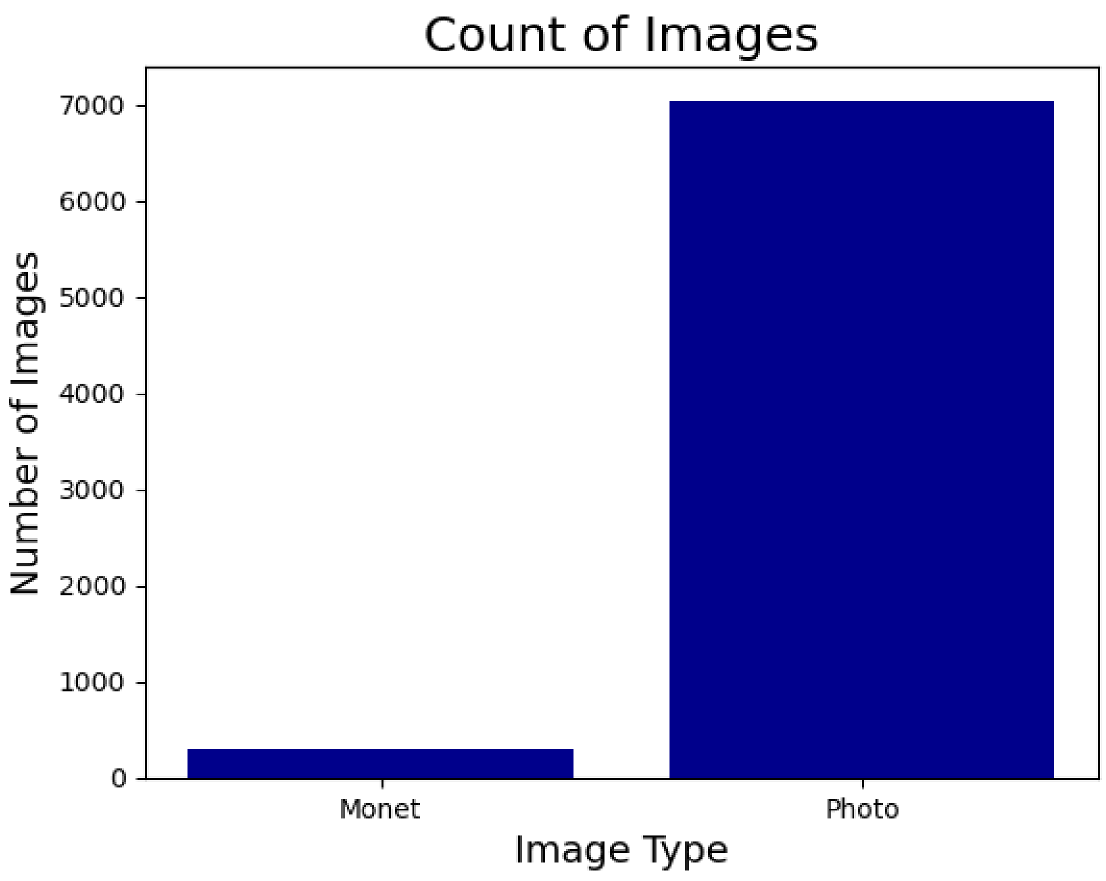
[75]:
```
# Check number of photos in monet_imgs and photo_imgs
num_monet_imgs = 0
num_photo_imgs = 0
```

```
for img in monet_imgs:
    num_monet_imgs += 1

for img in photo_imgs:
    num_photo_imgs += 1
print("Number of Monet Images:", num_monet_imgs, "\nNumber of Photo Images:",␣
 ↪num_photo_imgs)
```

```
Number of Monet Images: 300
Number of Photo Images: 7038
```

Figure 1: Count of Images

```
[80]: # Create dict of counts
count_dict = {'Monet': num_monet_imgs,'Photo': num_photo_imgs}
# Plot count of images
plt.bar(count_dict.keys(), height = count_dict.values(), color = 'darkblue')
plt.title("Count of Images", size = 18)
plt.xlabel("Image Type", size = 14)
plt.ylabel("Number of Images", size = 14)
plt.show()
```



Summary of Image Counts:

Since we have much fewer Monet images, we can use data augmentation methods during our preprocessing to help improve our results. This code was added to the preprocessing in Section 3 above.

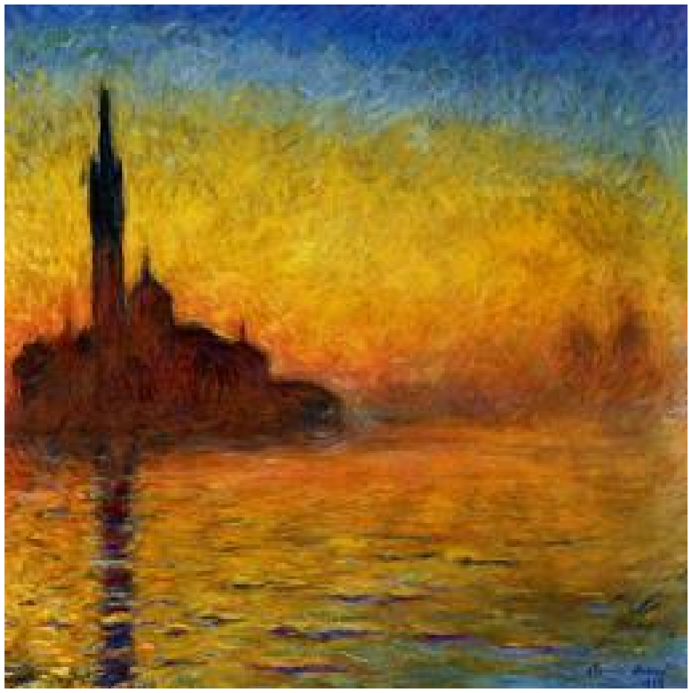
4.2 View Images

Let's view some images for a visualization of the data involved in this project. First, we can view one of Monet's water lily paintings from Giverny.

[8]:
```
# open image
waterlily_img = Image.open('GAN/1994b8d4a2.png')
display(waterlily_img)
```

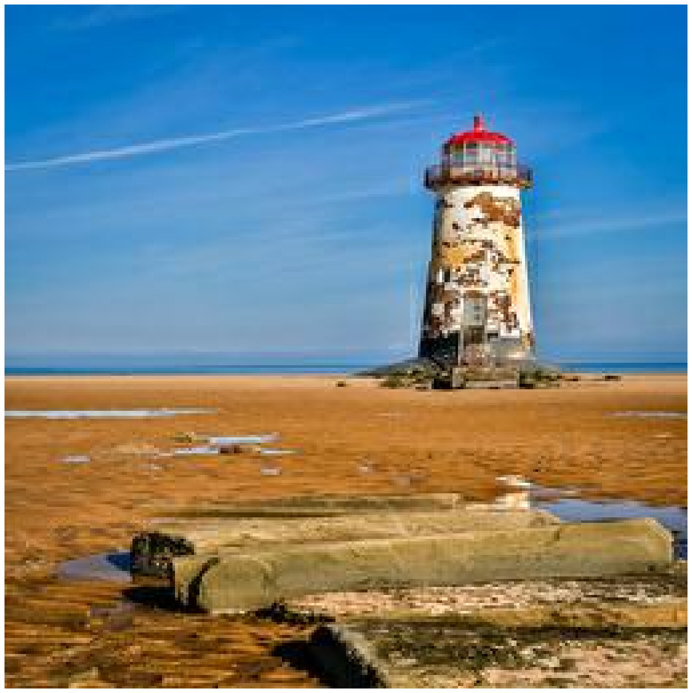
The image below is Monet's San Maggiore at Dusk.

[9]:
```
# open image
sanmag_img = Image.open('GAN/89d970411d.png')
display(sanmag_img)
```



Meanwhile, the following image of a lighthouse was from the photos file.

```
[10]: # open image
      lighthouse_img = Image.open('GAN/0a497f768d.png')
      display(lighthouse_img)
```



This photo of a lighthouse is clearly different from a painting in the Impressionist style of Monet. Therefore, we obviously have some work to do, if we are going to generate images in the style of Monet sufficient to fool the discriminator.

5 Section 5: Building Blocks and Architectures for CGAN

5.1 Build the Generator

For building the generator, the plan is to use a CycleGAN which is a modified version of the U-Net architecture described in the paper, "U-Net Convolutional Networks for Biomedical Image Segmentation". This research described a way to train a network on relatively few images, as opposed to thousands of samples. An example of this U-net architecture visualized in the research paper is shown below.

Let's summarize the key components:

Convolution 2D: The value 3x3 indicates filter size and 64, 128, 256, 512 are the number of filters.

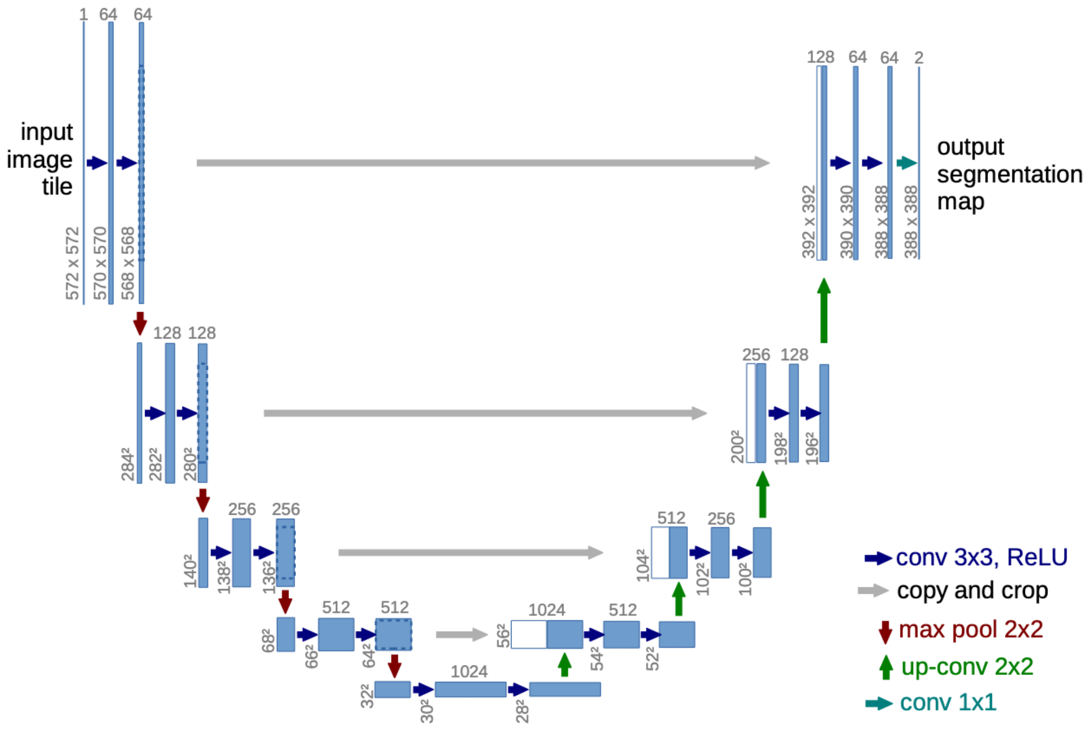
Copy and Crop: Copy and crop was necessary on this architecture because of pixel loss at the borders.

MaxPooling: After each block of Convolution 2D, there is Max Pooling with a stride of 2 for the downsampling layer.

Convolution 2D Transpose: Convolution 2D Transpose is used in the upsampling process.

Activation: The ReLU (Rectified Linear Unit) activation function is used with the Convolution layers.

[11]:
```
# open image
unet_img = Image.open('U-net_architecture.png')
display(unet_img)
```



Summary of U-Net Architecture:

We adopted a modified version of this U-Net architecture below. The image size was maintained throughout. To start, we build our encoder and decoder below.

- **Encoder:** This network architecture depicts an encoder on the left side which downsamples the image with blocks containing Convolution -> Batch Normalization -> Leaky ReLU.
- **Decoder:** Then, on the right, the decoder reverses the process in the sequence of: Transposed Convolution -> Batch Normalization -> Dropout -> ReLU.

5.1.1 Encoder for downsampling the image

- **Convolution 2D:** Since we have images, we use the Convolution 2D package with the varying filter sizes set in our generator and discriminator functions. We also set padding to same, which keeps the same size from input to output.
- **Normalization:** Here we use Batch Normalization after each Conv2D layer to help prevent overfitting.
- **Activation:** For the activation, Leaky ReLU (Leaky Rectified Linear Unit) is used to help avoid overfitting by feeding a small gradient value rather than zero, as in the case of ReLU.

[12]:
```
#hyperparams
channels = 3
strides = 2
```

```
mean = 0.0
std = 0.02

def downsample(filters, size, apply_norm = True):
    initializer = tf.random_normal_initializer(0., 0.02)
    gamma_init = keras.initializers.RandomNormal(mean = mean, stddev = std)

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

    if apply_norm:
        res.add(BatchNormalization(gamma_initializer = gamma_init))

    res.add(LeakyReLU())
    return res
```

5.1.2 Decoder for upsampling the image

- **Convolution 2D Transpose:** For our decoder, we reverse the process and therefore use Convolution 2D Transpose.
- **Normalization:** Here we use Batch Normalization after each Conv2D layer to help prevent overfitting.
- **Dropout:** We also set a drop out of 0.40 for each layer to randomly mask 40% of the data and help prevent overfitting.
- **Activation:** For upsampling, the ReLU (Rectified Linear Unit) activation is used and provides a non-negative output.

[13]:
```
#hyperparams
channels = 3
strides = 2
mean = 0.0
std = 0.02
drop = 0.4

def upsample(filters, size, apply_dropout = False):
    initializer = tf.random_normal_initializer(mean, std)
    gamma_init = keras.initializers.RandomNormal(mean = mean, stddev = std)

    res = keras.Sequential()
    res.add(Conv2DTranspose(filters, size, strides = strides, padding = 'same',␣
 ↪kernel_initializer = initializer, use_bias = False))

    res.add(BatchNormalization())

    if apply_dropout:
        res.add(Dropout(drop))

    res.add(ReLU())
    return res
```

5.1.3 Generator Function

Next, we create the generator utilizing the downsample and upsample functions for encoding and decoding the images. In addition, other values include:

- **Channels:** The channels are set to 3, since we have RGB images.
- **Strides:** The strides are set to 2, which causes the filter to skip and move by 2 pixels each time reducing the output dimension. We can see this result in the generator architecture below with the reduced output size in the layers. The effect is similar to pooling, as shown in the original U-Net architecture.
- **Kernel initializer:** The kernel initializer is using Random Normal initializer value with mean of 0 and standard deviation of 0.02. It sets the initial weights for our model using a random normal distribution.
- **Skip connections:** The skip connections feed input from previous layer to next layers, which helps prevent the gradients hitting zero and the degradation of performance in subsequent layers.

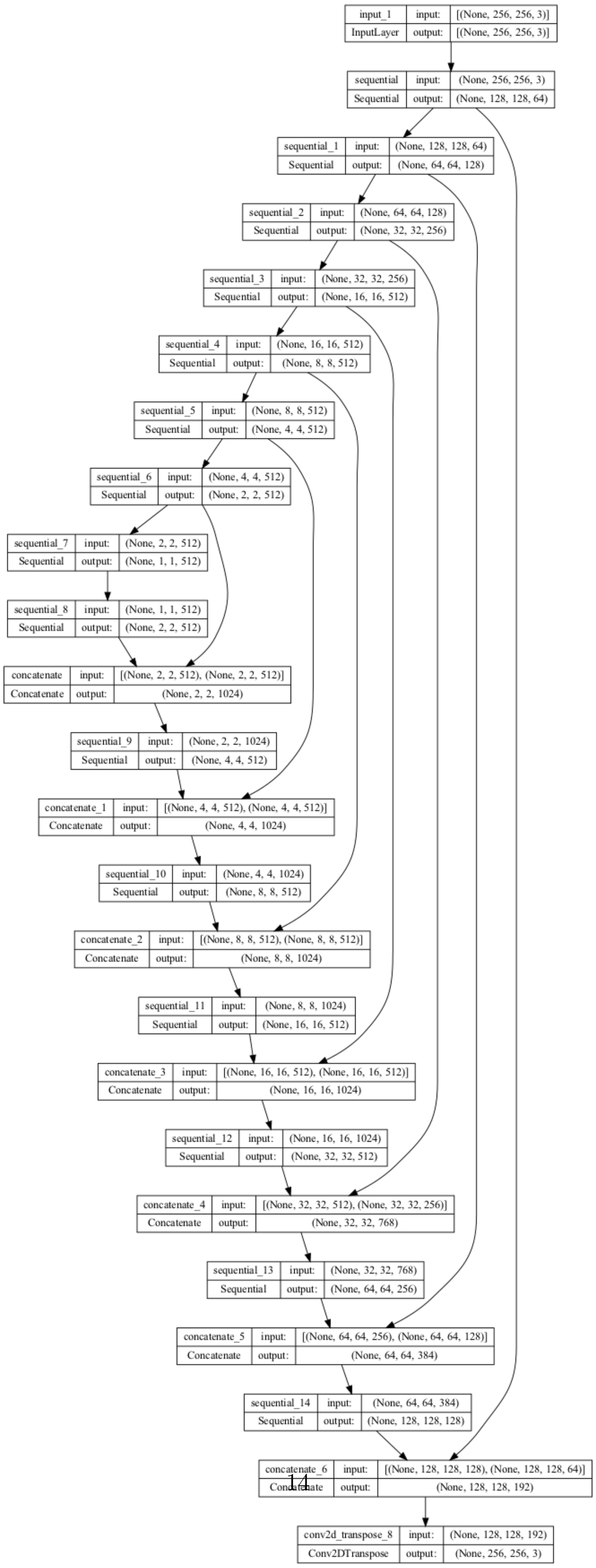
[14]:
```
#hyperparams
channels = 3
strides = 2
mean = 0.0
std = 0.02
drop = 0.4

# Define generator with layers and batch sizes

def Generator():
    inputs = layers.Input(shape = [256, 256, 3])

    down = [
        downsample(64,4, apply_norm = False),  # (batch size, 128, 128, 64)
        downsample(128,4),    # (batch size, 64, 64, 128)
        downsample(256,4),
        downsample(512,4),
        downsample(512,4),
        downsample(512,4),
        downsample(512,4),
        downsample(512,4)]

    up = [
        upsample(512, 4, apply_dropout = True),
        upsample(512, 4, apply_dropout = True),
        upsample(512, 4, apply_dropout = True),
        upsample(512, 4),
        upsample(512, 4),
        upsample(256, 4),
        upsample(128, 4),
        upsample(64, 4)]

    initializer = tf.random_normal_initializer(mean, std)
    trans_out = Conv2DTranspose(channels, 4, strides = 2, padding = 'same',␣
↪kernel_initializer = initializer, activation = 'tanh')

    i = inputs

    # Go down through model and skip connections
    skip_lst = []

    for d in down:
        i = d(i)
        skip_lst.append(i)

    # reverse list
    skip_lst = reversed(skip_lst[:-1])

    # Go up and get the skip connections
    for u, s in zip(up, skip_lst):
        i = u(i)
        i = layers.Concatenate()([i, s])

    i = trans_out(i)

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

5.1.4 Generator Architecture

Here we can visualize the Generator architecture, which shows the input and output shape of each layer throughout the downsampling and upsampling.

[15]:
```
# Visualize the generator architecture
gen = Generator()
tf.keras.utils.plot_model(gen, show_shapes = True, dpi = 64)
```

[15]:


input_1 | input: | [(None, 256, 256, 3)]
InputLayer | output: | [(None, 256, 256, 3)]
sequential | input: | (None, 256, 256, 3)
Sequential | output: | (None, 128, 128, 64)
sequential_1 | input: | (None, 128, 128, 64)
Sequential | output: | (None, 64, 64, 128)
sequential_2 | input: | (None, 64, 64, 128)
Sequential | output: | (None, 32, 32, 256)
sequential_3 | input: | (None, 32, 32, 256)
Sequential | output: | (None, 16, 16, 512)
sequential_4 | input: | (None, 16, 16, 512)
Sequential | output: | (None, 8, 8, 512)
sequential_5 | input: | (None, 8, 8, 512)
Sequential | output: | (None, 4, 4, 512)
sequential_6 | input: | (None, 4, 4, 512)
Sequential | output: | (None, 2, 2, 512)
sequential_7 | input: | (None, 2, 2, 512)
Sequential | output: | (None, 1, 1, 512)
sequential_8 | input: | (None, 1, 1, 512)
Sequential | output: | (None, 2, 2, 512)
concatenate | input: | [(None, 2, 2, 512), (None, 2, 2, 512)]
Concatenate | output: | (None, 2, 2, 1024)
sequential_9 | input: | (None, 2, 2, 1024)
Sequential | output: | (None, 4, 4, 512)
concatenate_1 | input: | [(None, 4, 4, 512), (None, 4, 4, 512)]
Concatenate | output: | (None, 4, 4, 1024)
sequential_10 | input: | (None, 4, 4, 1024)
Sequential | output: | (None, 8, 8, 512)
concatenate_2 | input: | [(None, 8, 8, 512), (None, 8, 8, 512)]
Concatenate | output: | (None, 8, 8, 1024)
sequential_11 | input: | (None, 8, 8, 1024)
Sequential | output: | (None, 16, 16, 512)
concatenate_3 | input: | [(None, 16, 16, 512), (None, 16, 16, 512)]
Concatenate | output: | (None, 16, 16, 1024)
sequential_12 | input: | (None, 16, 16, 1024)
Sequential | output: | (None, 32, 32, 512)
concatenate_4 | input: | [(None, 32, 32, 512), (None, 32, 32, 256)]
Concatenate | output: | (None, 32, 32, 768)
sequential_13 | input: | (None, 32, 32, 768)
Sequential | output: | (None, 64, 64, 256)
concatenate_5 | input: | [(None, 64, 64, 256), (None, 64, 64, 128)]
Concatenate | output: | (None, 64, 64, 384)
sequential_14 | input: | (None, 64, 64, 384)
Sequential | output: | (None, 128, 128, 128)
concatenate_6 | input: | [(None, 128, 128, 128), (None, 128, 128, 64)]
Concatenate | output: | (None, 128, 128, 192)
conv2d_transpose_8 | input: | (None, 128, 128, 192)
Conv2DTranspose | output: | (None, 256, 256, 3)

5.2 Build the Discriminator

Now, we need the discriminator or art critic for this project. The discriminator's role is to detect whether the Monet-stylized images from the generator are real or fake. The output from the discriminator will be a small two-dimensional image where low pixel values indicate a forgery and high pixel values are assigned for images denoted as real.

5.2.1 Discriminator Function

Next, we create the discriminator utilizing the downsample functions for encoding the images. In addition, one other value is:

- **Zero Padding:** Here we add zero padding of the input to our Convolution 2D layer which adds zeros to the border of our images. This padding ensures the pixels near the edges are also used in the training, as well as maintains consistent dimensions in our models.

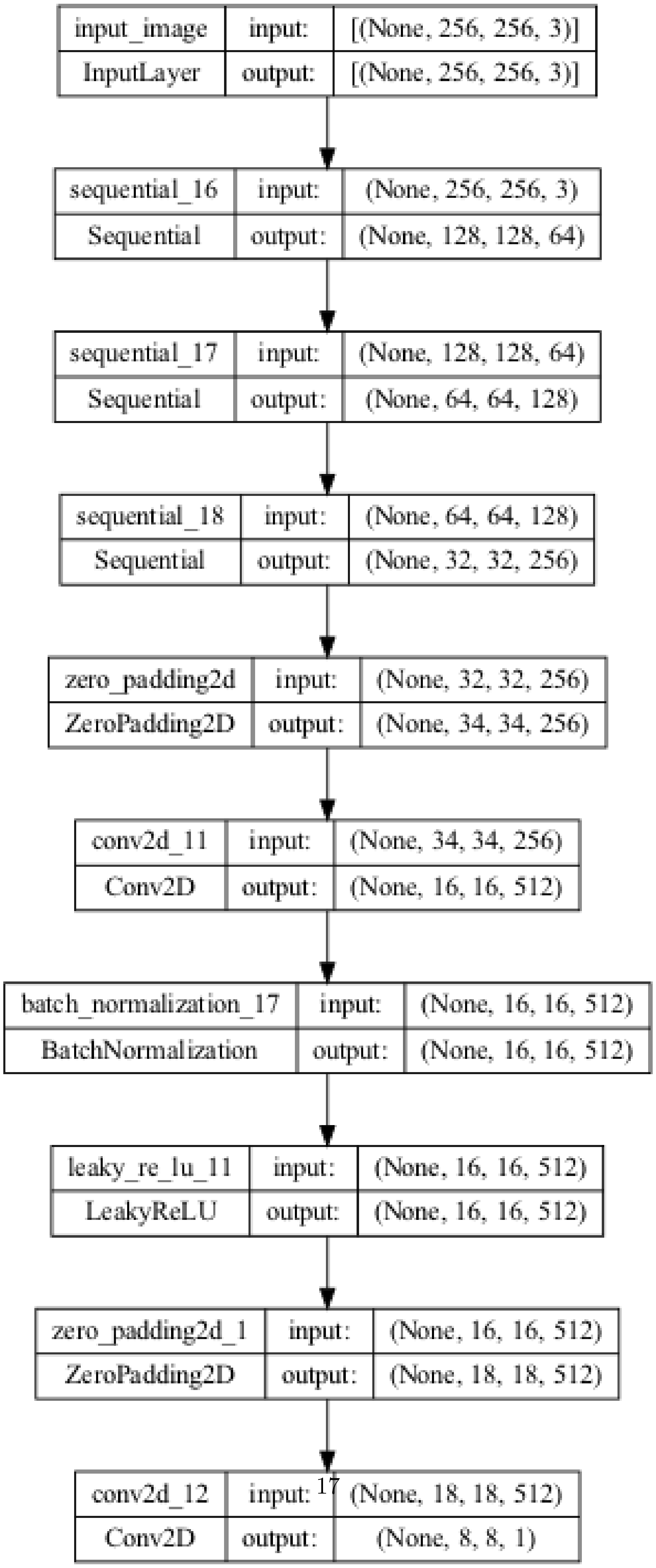
[16]:
```
#hyperparams
channels = 3
strides = 2
mean = 0.0
std = 0.02
drop = 0.4

# build discriminator
def Discriminator():
    initializer = tf.random_normal_initializer(mean, std)
    gamma_init = keras.initializers.RandomNormal(mean = mean, stddev = std)
    img_in = layers.Input(shape = [256,256,3], name = 'input_image')

    i = img_in

    d1 = downsample(64, 4, False)(i)
    d2 = downsample(128, 4)(d1)
    d3 = downsample(256,4)(d2)

    pad1 = ZeroPadding2D()(d3)
    conv2d = Conv2D(512, 4, strides = 2, kernel_initializer = initializer,␣
 ↪use_bias = False)(pad1)
    norm = BatchNormalization(gamma_initializer = gamma_init)(conv2d)
    leaky_relu = LeakyReLU()(norm)
    pad2 = ZeroPadding2D()(leaky_relu)

    trans_out = Conv2D(1,4, strides = strides, kernel_initializer =␣
 ↪initializer)(pad2)

    return tf.keras.Model(inputs = img_in, outputs = trans_out)
```

5.2.2 Discriminator Architecture

Now, we can visualize the Discriminator architecture which shows our layers as well as the shape of input and output.

[17]:
```
# Visualize the discriminator architecture
discrim = Discriminator()
tf.keras.utils.plot_model(discrim, show_shapes = True, dpi = 64)
```

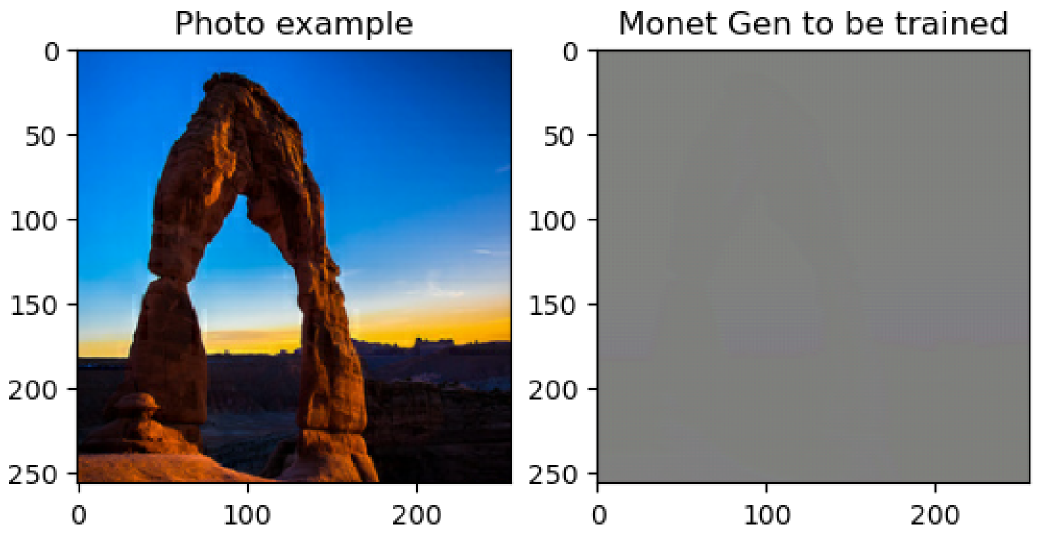
[17]:


input_image | input: | [(None, 256, 256, 3)]
InputLayer | output: | [(None, 256, 256, 3)]
sequential_16 | input: | (None, 256, 256, 3)
Sequential | output: | (None, 128, 128, 64)
sequential_17 | input: | (None, 128, 128, 64)
Sequential | output: | (None, 64, 64, 128)
sequential_18 | input: | (None, 64, 64, 128)
Sequential | output: | (None, 32, 32, 256)
zero_padding2d | input: | (None, 32, 32, 256)
ZeroPadding2D | output: | (None, 34, 34, 256)
conv2d_11 | input: | (None, 34, 34, 256)
Conv2D | output: | (None, 16, 16, 512)
batch_normalization_17 | input: | (None, 16, 16, 512)
BatchNormalization | output: | (None, 16, 16, 512)
leaky_re_lu_11 | input: | (None, 16, 16, 512)
LeakyReLU | output: | (None, 16, 16, 512)
zero_padding2d_1 | input: | (None, 16, 16, 512)
ZeroPadding2D | output: | (None, 18, 18, 512)
conv2d_12 | input: | (None, 18, 18, 512)
Conv2D | output: | (None, 8, 8, 1)

Next, we initialize the generators and discriminators for our Monet images, as well as the photos. Then, we can check an example. Since the Monet generator has not yet been trained, the sample picture should be grayed out.

[18]:
```
# establish generators and discriminators for Monets and photos
monet_gen = Generator()
photo_gen = Generator()

monet_dis = Discriminator()
photo_dis = Discriminator()
```

[19]:
```
# Check current status with generator
ex_photo = next(iter(photo_imgs))
ex_monet = monet_gen(ex_photo)

plt.subplot(1, 2, 1)
plt.title("Photo example")
plt.imshow(ex_photo[0]*0.5+0.5)

plt.subplot(1, 2, 2)
plt.title("Monet Gen to be trained")
plt.imshow(ex_monet[0]*0.5+0.5)
plt.show()
```



Note: Monet Generator needs to be trained. Therefore, the sample picture is grayed out.

6 Section 6: Build the CGAN Model

Once we have our Generator and Discriminator, we can build our Cycle-Consistent Generative Adversarial Network (CGAN or CycleGAN) model.

CGAN Model: We utilize the keras.Model for creating our CGAN model.

train_step: The train_step function below overrides the usual model training step called when fitting the model. This step carries out the predictions, loss and gradient calculations, as well as applying the gradients to our optimizer.

[20]:
```
class CGAN(keras.Model):
    def __init__(self, monet_gen, photo_gen, monet_dis, photo_dis, lambda_cycle␣
 ↪= 10, lambda_identity = 0.5):
        super(CGAN, self).__init__()
        self.m_gen = monet_gen
        self.p_gen = photo_gen
        self.m_dis = monet_dis
        self.p_dis = photo_dis
        self.lambda_cycle = lambda_cycle
        self.lambda_identity = lambda_identity

    def call(self, inputs):
        return (self.m_gen(inputs),
               self.p_gen(inputs),
               self.m_dis(inputs),
               self.p_dis(inputs))

    def compile(self, m_gen_opt, p_gen_opt, m_dis_opt, p_dis_opt, gen_loss_fn,␣
 ↪dis_loss_fn):
        super().compile()
        self.m_gen_opt = m_gen_opt
        self.p_gen_opt = p_gen_opt
        self.m_dis_opt = m_dis_opt
        self.p_dis_opt = p_dis_opt
        self.generator_loss_fn = gen_loss_fn
        self.discriminator_loss_fn = dis_loss_fn
        self.cycle_loss_fn = keras.losses.MeanAbsoluteError()
        self.id_loss_fn = keras.losses.MeanAbsoluteError()

    @tf.function
    def train_step(self, batch_data):
        real_m, real_p = batch_data

        with tf.GradientTape(persistent = True) as tape:
            # from photo to monet then back
            fake_m = self.m_gen(real_p, training = True)
            post_p = self.p_gen(fake_m, training = True)
            # monet to photo then back
            fake_p = self.p_gen(real_m, training = True)
            post_m = self.m_gen(fake_p, training = True)

            # generate the same real monet and photo
            same_m = self.m_gen(real_m, training = True)
            same_p = self.p_gen(real_p, training = True)

            # discriminator checks with real images as input
            dis_real_m = self.m_dis(real_m, training = True)
            dis_real_p = self.p_dis(real_p, training = True)

            # discriminator checks with fake images as input
            dis_fake_m = self.m_dis(fake_m, training = True)
            dis_fake_p = self.p_dis(fake_p, training = True)

            # get generator adversarial loss
            m_gen_loss = self.generator_loss_fn(dis_fake_m)
            p_gen_loss = self.generator_loss_fn(dis_fake_p)

            # Generator cycle loss
            cycle_loss_m = self.cycle_loss_fn(real_m, post_m) * self.
↪lambda_cycle
            cycle_loss_p = self.cycle_loss_fn(real_p, post_p) * self.
↪lambda_cycle

            # Generator identity loss
            id_loss_m = (self.id_loss_fn(real_m, same_m) * self.lambda_cycle *␣
↪self.lambda_identity)
            id_loss_p = (self.id_loss_fn(real_p, same_p) * self.lambda_cycle *␣
↪self.lambda_identity)

            # get Total Generator loss
            total_m_gen_loss = m_gen_loss + cycle_loss_m + id_loss_m
            total_p_gen_loss = p_gen_loss + cycle_loss_p + id_loss_p


            # Get Discriminator loss
            m_dis_loss = self.discriminator_loss_fn(dis_real_m, dis_fake_m)
            p_dis_loss = self.discriminator_loss_fn(dis_real_p, dis_fake_p)

            # Get Gradients for monet and photos, generator and discriminator
            m_gen_grad = tape.gradient(total_m_gen_loss, self.m_gen.
↪trainable_variables)
            p_gen_grad = tape.gradient(total_p_gen_loss, self.p_gen.
↪trainable_variables)
```

```
            m_dis_grad = tape.gradient(m_dis_loss, self.m_dis.
↪trainable_variables)
            p_dis_grad = tape.gradient(p_dis_loss, self.p_dis.
↪trainable_variables)

            # Apply Gradients for each optimizer, zip gradient with variables
            self.m_gen_opt.apply_gradients(zip(m_gen_grad, self.m_gen.
↪trainable_variables))
            self.p_gen_opt.apply_gradients(zip(p_gen_grad, self.p_gen.
↪trainable_variables))

            self.m_dis_opt.apply_gradients(zip(m_dis_grad, self.m_dis.
↪trainable_variables))
            self.p_dis_opt.apply_gradients(zip(p_dis_grad, self.p_dis.
↪trainable_variables))

            return {"Monet_Generator_loss": total_m_gen_loss,
                    "Photo_Generator_loss": total_p_gen_loss,
                    "Monet_Discriminator_loss": m_dis_loss,
                    "Photo_Discriminator_loss": p_dis_loss}
```

6.1 Create the Loss functions

Unlike our previous projects which had training and validation data, we have two different types of images. We need a computational method which enables our models to learn; therefore, we use loss functions.

Two loss functions incorporated in the CGAN model are:

- **Cycle loss function:** Our cycle loss calculation is based on Mean Absolute Error since it measures whether the image returned from a cycle is similar to the original image. For example, if a Monet is converted to an image then back to a Monet, it should resemble the original Monet.
- **Identity loss function:** The identity loss calculation is also based on Mean Absolute Error. For example, this function calculates what happens if a real Monet is fed to the Monet Generator.

To train the CGAN model, we also have loss functions for the generator as well as the discriminator.

- **Discriminator loss function:** The discriminator loss function below averages the real and fake loss calculations using binary cross entropy.
- **Generator loss function:** The generator loss function calculates the loss from our generator models, using binary cross entropy.

Binary cross entropy was used for the adversarial loss function, since we have a binary classification problem with our images being either a Monet or a photo. The discriminator and generator loss calculations are used with our gradient, which is then applied to our optimizer.

```
[21]: # Average the real and fake loss
adv_loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits = True, reduction␣
 ↪= tf.keras.losses.Reduction.NONE)

def discriminator_loss_fn(real, gen):
    real_loss = adv_loss_fn(tf.ones_like(real), real)
    fake_loss = adv_loss_fn(tf.zeros_like(gen), gen)
    avg_dis_loss = (real_loss + fake_loss) * 0.5

    return avg_dis_loss
```

```
[22]: # try to find gen loss
def generator_loss_fn(gen):
    fake_loss = adv_loss_fn(tf.ones_like(gen), gen)
    return fake_loss
```

7 Section 7: Train the CGAN Model(s)

Now that we have all of our building blocks, we can finally compile, fit and train our CGAN model.

7.1 CGAN Model 1 Training

Optimizer: When we compile the model, we select the Adaptive Moment estimation or 'Adam' optimizer which tends to perform well on images and; therefore, is a good starting point.

- **Learning Rate:** Instead of the default learning rate, we use a smaller learning rate of 0.0002 or 2e-4. Initial investigation showed a learning rate of 0.001 or 1e-3 appeared to lead to erratic behavior after approximately a dozen epochs.
- **Beta_1:** This hyperparameter value determines the exponential decay rate of the first moment estimates and values are typically a float value less than one. The beta_1 value is a moving average where smaller values give less weight to the past gradients and more weight to the recent gradients. Instead of the default value, we use a slightly smaller value of 0.80.

```
[30]: #Set optimizers
m_gen_optimizer = Adam(2e-4, beta_1 = 0.8)
p_gen_optimizer = Adam(2e-4, beta_1 = 0.8)

m_dis_optimizer = Adam(2e-4, beta_1 = 0.8)
p_dis_optimizer = Adam(2e-4, beta_1 = 0.8)
```

```
[31]: #Build CGAN model
cgan_model1 = CGAN(monet_gen, photo_gen, monet_dis, photo_dis)
cgan_model1.compile(m_gen_opt = m_gen_optimizer, p_gen_opt = p_gen_optimizer,
                   m_dis_opt = m_dis_optimizer, p_dis_opt = p_dis_optimizer,
                   gen_loss_fn = generator_loss_fn, dis_loss_fn =␣
 ↪discriminator_loss_fn)
```

```
[33]: cgan_model1.fit(tf.data.Dataset.zip((monet_imgs, photo_imgs)), epochs = 25)
```

```
Epoch 1/25
300/300 [==============================] - 506s 2s/step - Monet_Generator_loss:
3.2735 - Photo_Generator_loss: 3.6240 - Monet_Discriminator_loss: 0.6609 -
Photo_Discriminator_loss: 0.6587
Epoch 2/25
300/300 [==============================] - 529s 2s/step - Monet_Generator_loss:
2.5579 - Photo_Generator_loss: 2.6422 - Monet_Discriminator_loss: 0.6442 -
Photo_Discriminator_loss: 0.6546
Epoch 3/25
300/300 [==============================] - 601s 2s/step - Monet_Generator_loss:
2.5605 - Photo_Generator_loss: 2.6432 - Monet_Discriminator_loss: 0.6290 -
Photo_Discriminator_loss: 0.6453
Epoch 4/25
300/300 [==============================] - 718s 2s/step - Monet_Generator_loss:
2.4808 - Photo_Generator_loss: 2.5473 - Monet_Discriminator_loss: 0.6303 -
Photo_Discriminator_loss: 0.6499
Epoch 5/25
300/300 [==============================] - 574s 2s/step - Monet_Generator_loss:
2.3629 - Photo_Generator_loss: 2.4902 - Monet_Discriminator_loss: 0.6310 -
Photo_Discriminator_loss: 0.6440
Epoch 6/25
300/300 [==============================] - 571s 2s/step - Monet_Generator_loss:
2.2923 - Photo_Generator_loss: 2.4430 - Monet_Discriminator_loss: 0.6260 -
Photo_Discriminator_loss: 0.6372
Epoch 7/25
300/300 [==============================] - 576s 2s/step - Monet_Generator_loss:
2.2585 - Photo_Generator_loss: 2.4294 - Monet_Discriminator_loss: 0.6249 -
Photo_Discriminator_loss: 0.6274
Epoch 8/25
300/300 [==============================] - 572s 2s/step - Monet_Generator_loss:
2.2481 - Photo_Generator_loss: 2.4212 - Monet_Discriminator_loss: 0.6269 -
Photo_Discriminator_loss: 0.6361
Epoch 9/25
300/300 [==============================] - 575s 2s/step - Monet_Generator_loss:
2.2498 - Photo_Generator_loss: 2.4718 - Monet_Discriminator_loss: 0.6287 -
Photo_Discriminator_loss: 0.6279
Epoch 10/25
300/300 [==============================] - 572s 2s/step - Monet_Generator_loss:
2.2277 - Photo_Generator_loss: 2.4270 - Monet_Discriminator_loss: 0.6310 -
Photo_Discriminator_loss: 0.6280
Epoch 11/25
300/300 [==============================] - 616s 2s/step - Monet_Generator_loss:
2.2322 - Photo_Generator_loss: 2.4618 - Monet_Discriminator_loss: 0.6296 -
Photo_Discriminator_loss: 0.6324
Epoch 12/25
300/300 [==============================] - 722s 2s/step - Monet_Generator_loss:
```

```
2.2349 - Photo_Generator_loss: 2.4655 - Monet_Discriminator_loss: 0.6266 -
Photo_Discriminator_loss: 0.6222
Epoch 13/25
300/300 [==============================] - 715s 2s/step - Monet_Generator_loss:
2.2138 - Photo_Generator_loss: 2.4575 - Monet_Discriminator_loss: 0.6311 -
Photo_Discriminator_loss: 0.6217
Epoch 14/25
300/300 [==============================] - 703s 2s/step - Monet_Generator_loss:
2.2081 - Photo_Generator_loss: 2.4381 - Monet_Discriminator_loss: 0.6286 -
Photo_Discriminator_loss: 0.6250
Epoch 15/25
300/300 [==============================] - 707s 2s/step - Monet_Generator_loss:
2.2039 - Photo_Generator_loss: 2.4158 - Monet_Discriminator_loss: 0.6303 -
Photo_Discriminator_loss: 0.6394
Epoch 16/25
300/300 [==============================] - 704s 2s/step - Monet_Generator_loss:
2.1923 - Photo_Generator_loss: 2.4006 - Monet_Discriminator_loss: 0.6275 -
Photo_Discriminator_loss: 0.6355
Epoch 17/25
300/300 [==============================] - 708s 2s/step - Monet_Generator_loss:
2.2042 - Photo_Generator_loss: 2.4344 - Monet_Discriminator_loss: 0.6343 -
Photo_Discriminator_loss: 0.6327
Epoch 18/25
300/300 [==============================] - 703s 2s/step - Monet_Generator_loss:
2.1892 - Photo_Generator_loss: 2.4100 - Monet_Discriminator_loss: 0.6302 -
Photo_Discriminator_loss: 0.6277
Epoch 19/25
300/300 [==============================] - 564s 2s/step - Monet_Generator_loss:
2.1968 - Photo_Generator_loss: 2.3800 - Monet_Discriminator_loss: 0.6312 -
Photo_Discriminator_loss: 0.6459
Epoch 20/25
300/300 [==============================] - 561s 2s/step - Monet_Generator_loss:
2.1962 - Photo_Generator_loss: 2.3294 - Monet_Discriminator_loss: 0.6262 -
Photo_Discriminator_loss: 0.6406
Epoch 21/25
300/300 [==============================] - 559s 2s/step - Monet_Generator_loss:
2.1985 - Photo_Generator_loss: 2.4067 - Monet_Discriminator_loss: 0.6333 -
Photo_Discriminator_loss: 0.6352
Epoch 22/25
300/300 [==============================] - 562s 2s/step - Monet_Generator_loss:
2.1961 - Photo_Generator_loss: 2.3376 - Monet_Discriminator_loss: 0.6337 -
Photo_Discriminator_loss: 0.6446
Epoch 23/25
300/300 [==============================] - 553s 2s/step - Monet_Generator_loss:
2.1964 - Photo_Generator_loss: 2.3688 - Monet_Discriminator_loss: 0.6356 -
Photo_Discriminator_loss: 0.6376
Epoch 24/25
300/300 [==============================] - 563s 2s/step - Monet_Generator_loss:
```

```
2.1767 - Photo_Generator_loss: 2.3554 - Monet_Discriminator_loss: 0.6281 -
Photo_Discriminator_loss: 0.6312
Epoch 25/25
300/300 [==============================] - 559s 2s/step - Monet_Generator_loss:
2.2045 - Photo_Generator_loss: 2.4016 - Monet_Discriminator_loss: 0.6464 -
Photo_Discriminator_loss: 0.6479
```

[33]:
```
<keras.callbacks.History at 0x380d8b790>
```

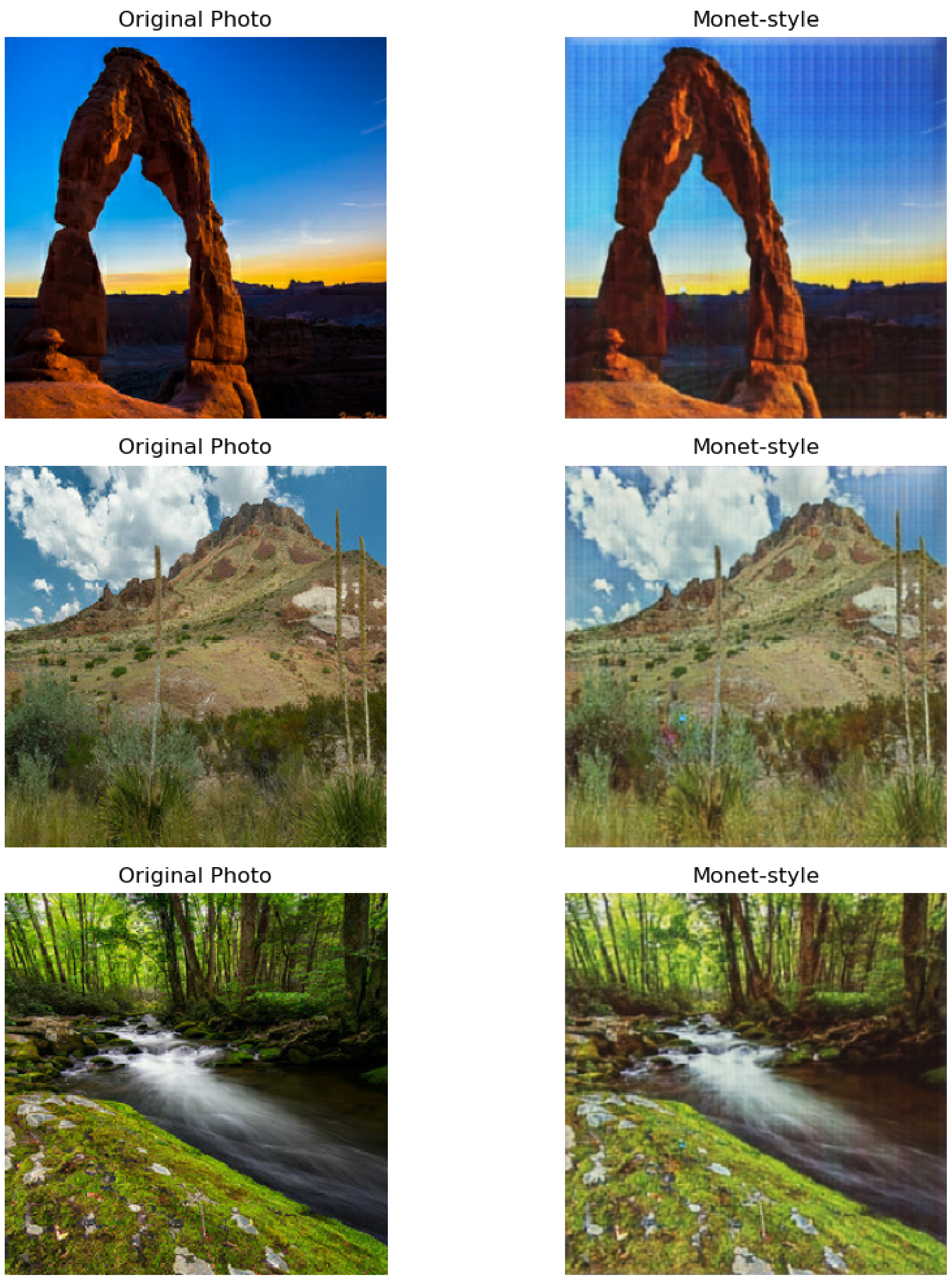
Note: The minimum Monet Generator Loss occurred in epoch 24; therefore, increasing the number of epochs may further reduce our loss.

Let's try a visualization.

7.2 Visualize the Monet-stylized photos

Now that our Monet Generator has been trained, let's view a simple side-by-side comparison of the original photo next to the Monet-stylized version of the same photo.

[59]:
```python
# create view of several images
_, ax = plt.subplots(3,2, figsize = (10,10))
for i, img in enumerate(photo_imgs.take(3)):
    pred = monet_gen(img, training = False)[0].numpy()
    pred = (pred * 127.5 + 127.5).astype(np.uint8)
    img = (img[0] * 127.5 + 127.5).astype(np.uint8)
    ax[i, 0].imshow(img)
    ax[i, 1].imshow(pred)

    ax[i, 0].set_title("Original Photo")
    ax[i, 1].set_title("Monet-style")

    ax[i, 0].axis("off")
    ax[i, 1].axis("off")

plt.tight_layout()
plt.show()
```



Summary: There are some noticeable differences between the original photos and the Monet-style photos; however, the generated photos do not appear very similar to actual Monet paintings. First, let's save this model. Then, we can see if modifying the learning rate and epochs may lead to an improved model.

Save CGAN Model 1:

[69]:
```
# Save model
cgan_model1.save_weights('cgan_model1.h5')
```

Next, let's increase the epochs and see if we can improve our results.

7.3 CGAN Model 2 Training

Epochs: Now, let's increase the number of epochs to 50 try and improve our results.

Note: Originally, we tried a lower Learning Rate (LR) of 1e-4 but that yielded worse results. Therefore, the LR was set back to 2e-4 and re-run.

[34]:
```
#Set optimizers
m_gen_optimizer = Adam(2e-4, beta_1 = 0.8)
p_gen_optimizer = Adam(2e-4, beta_1 = 0.8)

m_dis_optimizer = Adam(2e-4, beta_1 = 0.8)
p_dis_optimizer = Adam(2e-4, beta_1 = 0.8)
```

[35]:
```
#Build CGAN model
cgan_model2 = CGAN(monet_gen, photo_gen, monet_dis, photo_dis)
cgan_model2.compile(m_gen_opt = m_gen_optimizer, p_gen_opt = p_gen_optimizer,
                    m_dis_opt = m_dis_optimizer, p_dis_opt = p_dis_optimizer,
                    gen_loss_fn = generator_loss_fn, dis_loss_fn =␣
 ↪discriminator_loss_fn)
```

[134]:
```
cgan_model2.fit(tf.data.Dataset.zip((monet_imgs, photo_imgs)), epochs = 50)
```

```
Epoch 1/50
300/300 [==============================] - 630s 2s/step - Monet_Generator_loss:
3.5300 - Photo_Generator_loss: 3.8918 - Monet_Discriminator_loss: 0.6597 -
Photo_Discriminator_loss: 0.6613
Epoch 2/50
300/300 [==============================] - 659s 2s/step - Monet_Generator_loss:
2.5594 - Photo_Generator_loss: 2.6522 - Monet_Discriminator_loss: 0.6441 -
Photo_Discriminator_loss: 0.6500
Epoch 3/50
300/300 [==============================] - 657s 2s/step - Monet_Generator_loss:
2.5529 - Photo_Generator_loss: 2.6040 - Monet_Discriminator_loss: 0.6351 -
Photo_Discriminator_loss: 0.6520
Epoch 4/50
300/300 [==============================] - 654s 2s/step - Monet_Generator_loss:
2.4691 - Photo_Generator_loss: 2.5824 - Monet_Discriminator_loss: 0.6367 -
Photo_Discriminator_loss: 0.6492
Epoch 5/50
300/300 [==============================] - 653s 2s/step - Monet_Generator_loss:
```

```
2.3846 - Photo_Generator_loss: 2.5430 - Monet_Discriminator_loss: 0.6383 -
Photo_Discriminator_loss: 0.6502
Epoch 6/50
300/300 [==============================] - 609s 2s/step - Monet_Generator_loss:
2.3009 - Photo_Generator_loss: 2.4846 - Monet_Discriminator_loss: 0.6408 -
Photo_Discriminator_loss: 0.6469
Epoch 7/50
300/300 [==============================] - 516s 2s/step - Monet_Generator_loss:
2.2523 - Photo_Generator_loss: 2.4330 - Monet_Discriminator_loss: 0.6359 -
Photo_Discriminator_loss: 0.6417
Epoch 8/50
300/300 [==============================] - 515s 2s/step - Monet_Generator_loss:
2.2131 - Photo_Generator_loss: 2.4056 - Monet_Discriminator_loss: 0.6379 -
Photo_Discriminator_loss: 0.6387
Epoch 9/50
300/300 [==============================] - 516s 2s/step - Monet_Generator_loss:
2.2101 - Photo_Generator_loss: 2.4246 - Monet_Discriminator_loss: 0.6413 -
Photo_Discriminator_loss: 0.6404
Epoch 10/50
300/300 [==============================] - 517s 2s/step - Monet_Generator_loss:
2.1936 - Photo_Generator_loss: 2.4296 - Monet_Discriminator_loss: 0.6433 -
Photo_Discriminator_loss: 0.6328
Epoch 11/50
300/300 [==============================] - 518s 2s/step - Monet_Generator_loss:
2.1784 - Photo_Generator_loss: 2.4756 - Monet_Discriminator_loss: 0.6513 -
Photo_Discriminator_loss: 0.6291
Epoch 12/50
300/300 [==============================] - 518s 2s/step - Monet_Generator_loss:
2.1641 - Photo_Generator_loss: 2.4628 - Monet_Discriminator_loss: 0.6395 -
Photo_Discriminator_loss: 0.6261
Epoch 13/50
300/300 [==============================] - 518s 2s/step - Monet_Generator_loss:
2.1713 - Photo_Generator_loss: 2.4980 - Monet_Discriminator_loss: 0.6463 -
Photo_Discriminator_loss: 0.6214
Epoch 14/50
300/300 [==============================] - 520s 2s/step - Monet_Generator_loss:
2.1422 - Photo_Generator_loss: 2.4340 - Monet_Discriminator_loss: 0.6414 -
Photo_Discriminator_loss: 0.6294
Epoch 15/50
300/300 [==============================] - 519s 2s/step - Monet_Generator_loss:
2.1807 - Photo_Generator_loss: 2.4728 - Monet_Discriminator_loss: 0.6477 -
Photo_Discriminator_loss: 0.6433
Epoch 16/50
300/300 [==============================] - 521s 2s/step - Monet_Generator_loss:
2.1538 - Photo_Generator_loss: 2.4454 - Monet_Discriminator_loss: 0.6438 -
Photo_Discriminator_loss: 0.6371
Epoch 17/50
300/300 [==============================] - 520s 2s/step - Monet_Generator_loss:
```

```
2.1517 - Photo_Generator_loss: 2.4650 - Monet_Discriminator_loss: 0.6439 -
Photo_Discriminator_loss: 0.6293
Epoch 18/50
300/300 [==============================] - 523s 2s/step - Monet_Generator_loss:
2.1676 - Photo_Generator_loss: 2.4613 - Monet_Discriminator_loss: 0.6455 -
Photo_Discriminator_loss: 0.6327
Epoch 19/50
300/300 [==============================] - 521s 2s/step - Monet_Generator_loss:
2.1740 - Photo_Generator_loss: 2.4402 - Monet_Discriminator_loss: 0.6442 -
Photo_Discriminator_loss: 0.6448
Epoch 20/50
300/300 [==============================] - 523s 2s/step - Monet_Generator_loss:
2.1576 - Photo_Generator_loss: 2.4049 - Monet_Discriminator_loss: 0.6383 -
Photo_Discriminator_loss: 0.6305
Epoch 21/50
300/300 [==============================] - 522s 2s/step - Monet_Generator_loss:
2.1520 - Photo_Generator_loss: 2.4910 - Monet_Discriminator_loss: 0.6477 -
Photo_Discriminator_loss: 0.6411
Epoch 22/50
300/300 [==============================] - 523s 2s/step - Monet_Generator_loss:
2.1565 - Photo_Generator_loss: 2.4983 - Monet_Discriminator_loss: 0.6480 -
Photo_Discriminator_loss: 0.6220
Epoch 23/50
300/300 [==============================] - 524s 2s/step - Monet_Generator_loss:
2.1662 - Photo_Generator_loss: 2.5038 - Monet_Discriminator_loss: 0.6484 -
Photo_Discriminator_loss: 0.6365
Epoch 24/50
300/300 [==============================] - 523s 2s/step - Monet_Generator_loss:
2.1607 - Photo_Generator_loss: 2.5282 - Monet_Discriminator_loss: 0.6510 -
Photo_Discriminator_loss: 0.6386
Epoch 25/50
300/300 [==============================] - 525s 2s/step - Monet_Generator_loss:
2.1367 - Photo_Generator_loss: 2.5039 - Monet_Discriminator_loss: 0.6461 -
Photo_Discriminator_loss: 0.6197
Epoch 26/50
300/300 [==============================] - 522s 2s/step - Monet_Generator_loss:
2.1485 - Photo_Generator_loss: 2.4875 - Monet_Discriminator_loss: 0.6465 -
Photo_Discriminator_loss: 0.6374
Epoch 27/50
300/300 [==============================] - 525s 2s/step - Monet_Generator_loss:
2.1281 - Photo_Generator_loss: 2.5350 - Monet_Discriminator_loss: 0.6595 -
Photo_Discriminator_loss: 0.6354
Epoch 28/50
300/300 [==============================] - 524s 2s/step - Monet_Generator_loss:
2.1380 - Photo_Generator_loss: 2.5833 - Monet_Discriminator_loss: 0.6544 -
Photo_Discriminator_loss: 0.6358
Epoch 29/50
300/300 [==============================] - 524s 2s/step - Monet_Generator_loss:
```

```
2.1208 - Photo_Generator_loss: 2.5488 - Monet_Discriminator_loss: 0.6509 -
Photo_Discriminator_loss: 0.6300
Epoch 30/50
300/300 [==============================] - 524s 2s/step - Monet_Generator_loss:
2.1183 - Photo_Generator_loss: 2.4842 - Monet_Discriminator_loss: 0.6563 -
Photo_Discriminator_loss: 0.6398
Epoch 31/50
300/300 [==============================] - 525s 2s/step - Monet_Generator_loss:
2.1257 - Photo_Generator_loss: 2.5594 - Monet_Discriminator_loss: 0.6557 -
Photo_Discriminator_loss: 0.6304
Epoch 32/50
300/300 [==============================] - 524s 2s/step - Monet_Generator_loss:
2.1467 - Photo_Generator_loss: 2.4500 - Monet_Discriminator_loss: 0.6412 -
Photo_Discriminator_loss: 0.6263
Epoch 33/50
300/300 [==============================] - 525s 2s/step - Monet_Generator_loss:
2.1421 - Photo_Generator_loss: 2.5363 - Monet_Discriminator_loss: 0.6650 -
Photo_Discriminator_loss: 0.6318
Epoch 34/50
300/300 [==============================] - 524s 2s/step - Monet_Generator_loss:
2.1002 - Photo_Generator_loss: 2.5415 - Monet_Discriminator_loss: 0.6664 -
Photo_Discriminator_loss: 0.6293
Epoch 35/50
300/300 [==============================] - 524s 2s/step - Monet_Generator_loss:
2.0843 - Photo_Generator_loss: 2.5402 - Monet_Discriminator_loss: 0.6547 -
Photo_Discriminator_loss: 0.6222
Epoch 36/50
300/300 [==============================] - 524s 2s/step - Monet_Generator_loss:
2.0880 - Photo_Generator_loss: 2.4669 - Monet_Discriminator_loss: 0.6635 -
Photo_Discriminator_loss: 0.6342
Epoch 37/50
300/300 [==============================] - 526s 2s/step - Monet_Generator_loss:
2.1519 - Photo_Generator_loss: 2.5320 - Monet_Discriminator_loss: 0.6613 -
Photo_Discriminator_loss: 0.6363
Epoch 38/50
300/300 [==============================] - 524s 2s/step - Monet_Generator_loss:
2.0975 - Photo_Generator_loss: 2.4772 - Monet_Discriminator_loss: 0.6576 -
Photo_Discriminator_loss: 0.6288
Epoch 39/50
300/300 [==============================] - 524s 2s/step - Monet_Generator_loss:
2.0884 - Photo_Generator_loss: 2.4223 - Monet_Discriminator_loss: 0.6605 -
Photo_Discriminator_loss: 0.6325
Epoch 40/50
300/300 [==============================] - 524s 2s/step - Monet_Generator_loss:
2.0589 - Photo_Generator_loss: 2.3982 - Monet_Discriminator_loss: 0.6578 -
Photo_Discriminator_loss: 0.6240
Epoch 41/50
300/300 [==============================] - 524s 2s/step - Monet_Generator_loss:
```

```
2.0472 - Photo_Generator_loss: 2.4223 - Monet_Discriminator_loss: 0.6597 -
Photo_Discriminator_loss: 0.6292
Epoch 42/50

300/300 [==============================] - 524s 2s/step - Monet_Generator_loss:
2.0735 - Photo_Generator_loss: 2.4501 - Monet_Discriminator_loss: 0.6544 -
Photo_Discriminator_loss: 0.6188
Epoch 43/50
300/300 [==============================] - 524s 2s/step - Monet_Generator_loss:
2.0785 - Photo_Generator_loss: 2.4323 - Monet_Discriminator_loss: 0.6646 -
Photo_Discriminator_loss: 0.6228
Epoch 44/50
300/300 [==============================] - 524s 2s/step - Monet_Generator_loss:
2.0499 - Photo_Generator_loss: 2.4715 - Monet_Discriminator_loss: 0.6542 -
Photo_Discriminator_loss: 0.6167
Epoch 45/50
300/300 [==============================] - 524s 2s/step - Monet_Generator_loss:
2.0471 - Photo_Generator_loss: 2.3848 - Monet_Discriminator_loss: 0.6422 -
Photo_Discriminator_loss: 0.6100
Epoch 46/50
300/300 [==============================] - 524s 2s/step - Monet_Generator_loss:
2.0649 - Photo_Generator_loss: 2.5019 - Monet_Discriminator_loss: 0.6558 -
Photo_Discriminator_loss: 0.6206
Epoch 47/50
300/300 [==============================] - 523s 2s/step - Monet_Generator_loss:
2.0665 - Photo_Generator_loss: 2.4164 - Monet_Discriminator_loss: 0.6489 -
Photo_Discriminator_loss: 0.6220
Epoch 48/50
300/300 [==============================] - 524s 2s/step - Monet_Generator_loss:
2.0788 - Photo_Generator_loss: 2.4642 - Monet_Discriminator_loss: 0.6597 -
Photo_Discriminator_loss: 0.6197
Epoch 49/50
300/300 [==============================] - 523s 2s/step - Monet_Generator_loss:
2.0881 - Photo_Generator_loss: 2.3823 - Monet_Discriminator_loss: 0.6549 -
Photo_Discriminator_loss: 0.6132
Epoch 50/50
300/300 [==============================] - 523s 2s/step - Monet_Generator_loss:
2.0374 - Photo_Generator_loss: 2.4338 - Monet_Discriminator_loss: 0.6544 -
Photo_Discriminator_loss: 0.6202
```

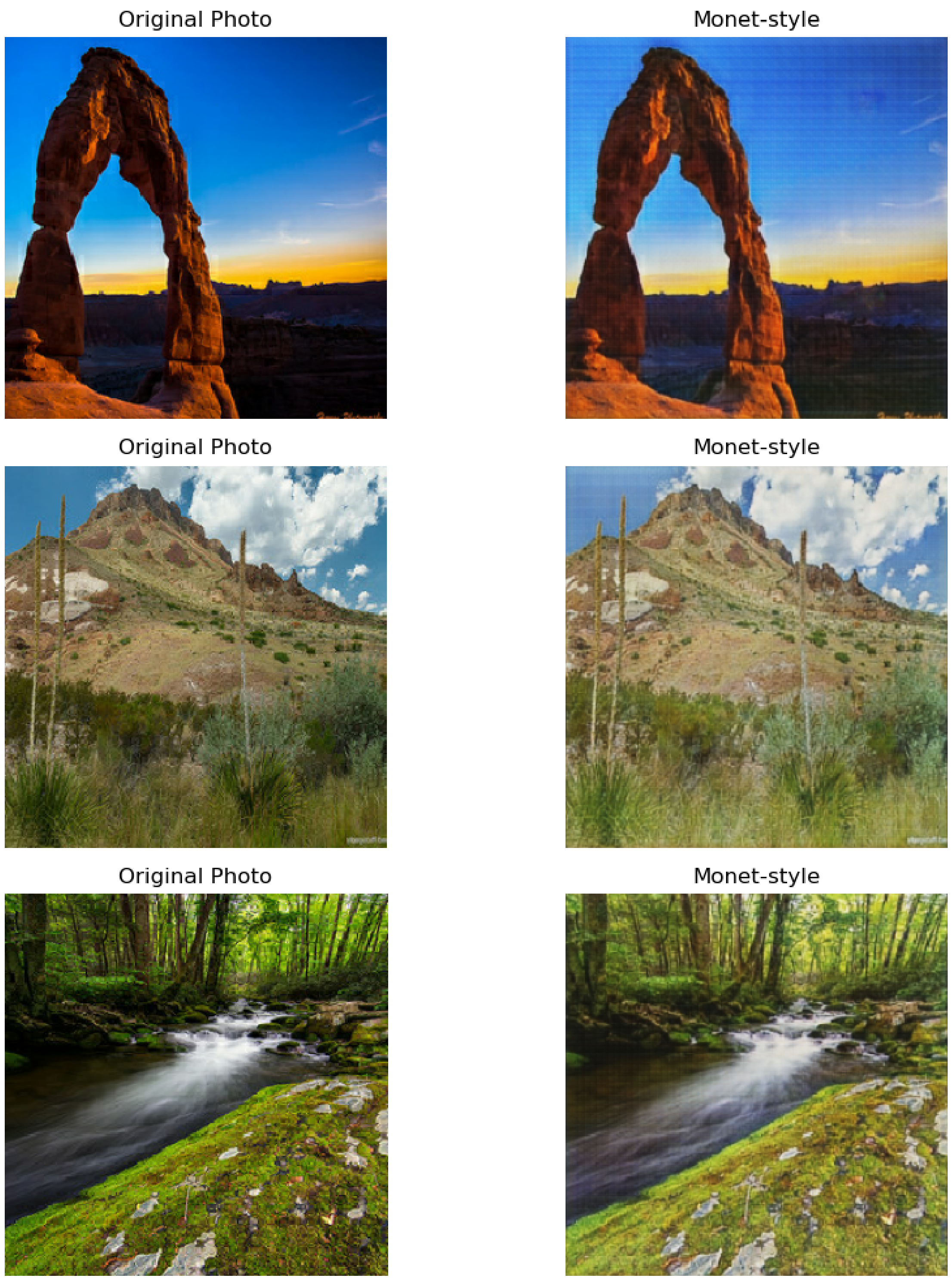
```
[134]: <keras.callbacks.History at 0x3660e1d10>
```

Note: We see the Monet Generator loss dropped with the additional epochs.

Let's try a visualization.

7.4 Visualize the Monet-stylized photos

[135]:
```
# create view of several images
_, ax = plt.subplots(3,2, figsize = (10,10))
for i, img in enumerate(photo_imgs.take(3)):
    pred = monet_gen(img, training = False)[0].numpy()
    pred = (pred * 127.5 + 127.5).astype(np.uint8)
    img = (img[0] * 127.5 + 127.5).numpy().astype(np.uint8)

    ax[i, 0].imshow(img)
    ax[i, 1].imshow(pred)

    ax[i, 0].set_title("Original Photo")
    ax[i, 1].set_title("Monet-style")

    ax[i, 0].axis("off")
    ax[i, 1].axis("off")

plt.tight_layout()
plt.show()
```



Summary: There are some noticeable differences between the original photos and the Monet-style photos; however, it looks like more epochs would be required to create images that truly resemble Monet paintings. Let's save this model.

Save CGAN Model 2:

```
[136]: # Save model
cgan_model2.save_weights('cgan_model2.h5')
```

7.5 Directory and Images for Kaggle submissions

Now we can create a directory for storing our generated images and submit to Kaggle to check our models. The model weights and architecture were loaded into Kaggle with images generated on Kaggle for the submission. The Kaggle screenshot and score are included below.

```
[78]: # Create images directory
#os.mkdir(images)
```

```
[ ]: # Check folders exist
#print("Image_dir folders:", os.listdir('images'))
```

7.5.1 Generate Images from CGAN Model 2

This code is shown for display purposes but the code is not run, since we do not want to re-generate the images.

```
[140]: # Generate images and store in img_directory
#i = 1
#for img in photo_imgs:
    #pred = monet_gen(img, training = False)[0].numpy()
    #pred = (pred * 127.5 + 127.5).astype(np.uint8)
    #im = Image.fromarray(pred)
    #im.save("images/" + str(i) + ".jpg")
    #i+=1
```

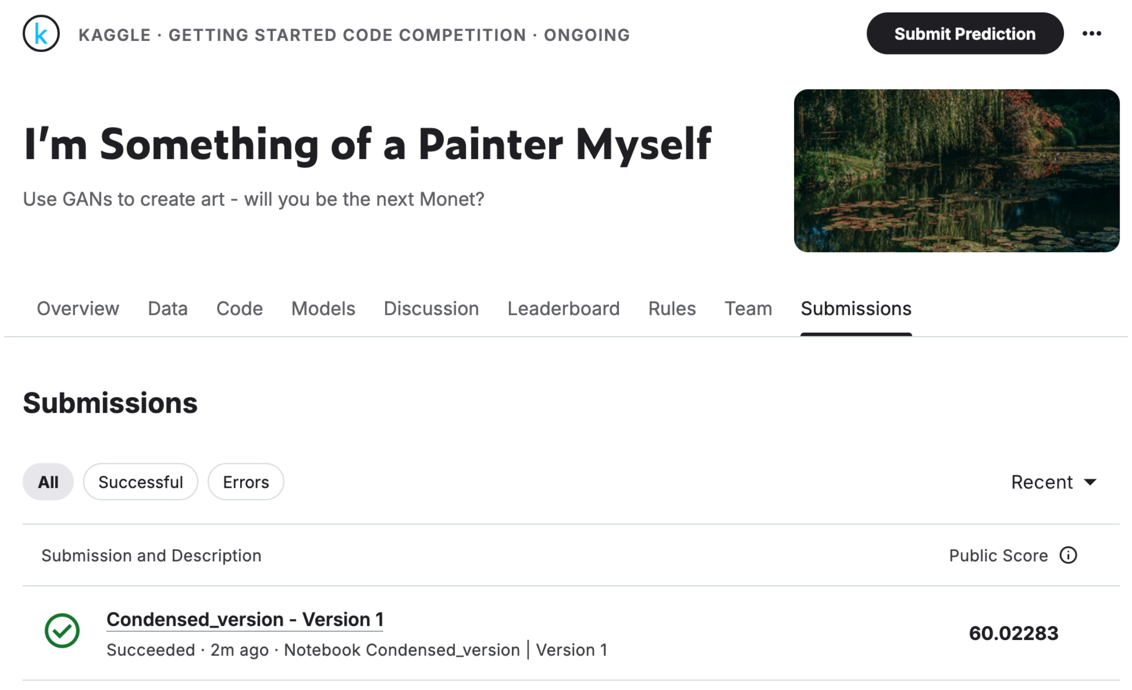
```
[148]: # Check number of files generated
print("Number of files in image directory:", len([img for img in os.
 ↪listdir('images')]), "images")
```

```
Number of files in image directory: 7038 images
```

```
[ ]: # Create zip file of images
#shutil.make_archive('images', format = 'zip', root_dir = ".", base_dir =␣
 ↪'images')
```

7.5.2 Kaggle screenshot of model submission

```
[4]: # open image
kaggle_cgan_img = Image.open('Kaggle_CGAN_submission.png')
display(kaggle_cgan_img)
```



I'm Something of a Painter Myself

Use GANs to create art - will you be the next Monet?

Submissions

Submission and Description	Public Score
Condensed_version - Version 1 Succeeded · 2m ago · Notebook Condensed_version | Version 1	60.02283

Summary:

Here we see the Kaggle Public Score of 60.02 rounded to two decimal places.

8 Section 8: Compare Results Across Models

Below we have a visualization of the Monet Generator Loss scores across the two models.

There is also a table comparing the hyperparameters for the models, along with their score.

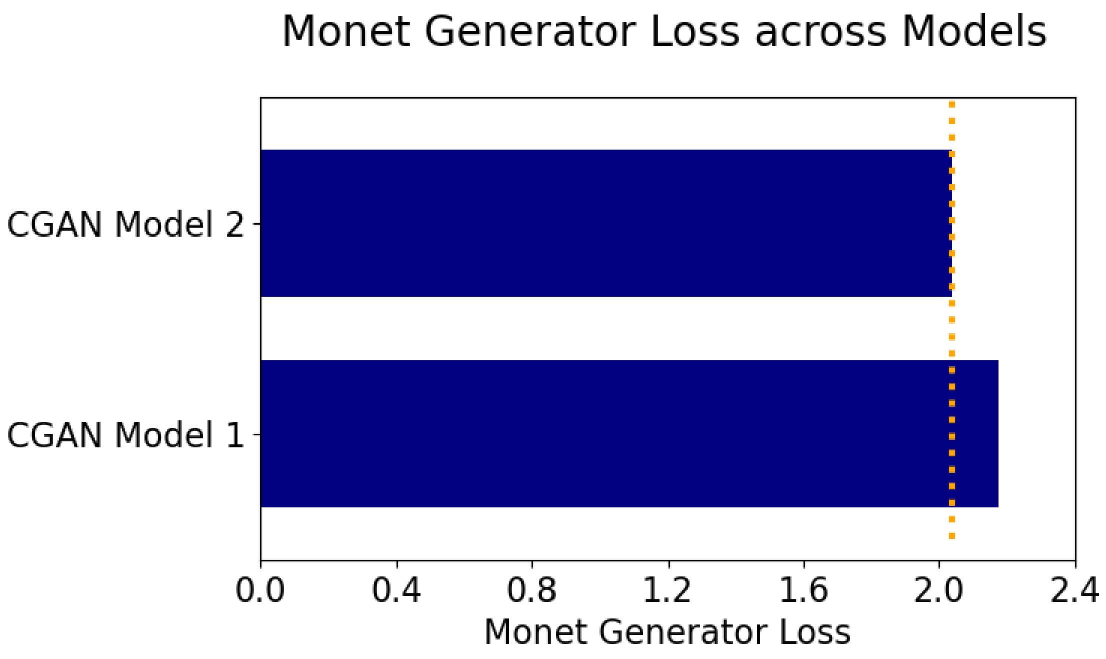
Figure 2: Monet Generator Loss across Models

```
[5]: # Extract the accuracy scores for each model and create list of scores and␣
     ↪models
     mon_gen_loss1, mon_gen_loss2 = 2.1767, 2.0374

     gen_loss_scores = [mon_gen_loss1, mon_gen_loss2]
     model_names = ["CGAN Model 1", "CGAN Model 2"]
```

```
[6]: # Create plot of Accuracy and model names
     plt.figure()
     plt.figure(figsize = (7,4))
     plt.suptitle("Monet Generator Loss across Models", fontsize = 20, x = 0.51, y =␣
     ↪1.02)

     acc_plot = pd.Series(gen_loss_scores,
                          index = model_names).sort_values(ascending = False)
```

```
ax = acc_plot.plot.barh(width = 0.7, color = 'navy')
plt.vlines(x = 2.04, ymin = -0.5, ymax = 200, color = "orange", ls = 'dotted',␣
  ↪lw = 3)
plt.xticks(np.arange(0, 2.60, step= 0.4), fontsize = 16)
plt.yticks(fontsize = 16)
plt.xlabel("Monet Generator Loss", fontsize = 16)
plt.show()
```

```
<Figure size 640x480 with 0 Axes>
```



Summary of Monet Generator Loss across Models:

Some items noted above:

- The best results were from Model 2 with 50 epochs.
- Models 1 and 2 had the same hyperparameters – the key difference was the number of epochs.
- Doubling the number of epochs reduced loss approximately 6.4%. Ideally, if we expect generative images similar enough to Monet that it would fool an art critic, the loss needs to be closer to zero.

In summary, further training and investigation is needed to improve these models.

8.1 Summary of Models and Hyperparameters

In conclusion, here is the summary of the top scores from models, as well as some key hyperparameters. The results from the top model is shown in blue, whereas, the score in red reflects a model with a worse (higher) loss. Since Model 2 yielded a lower Monet Generator loss, the Monet-generated images submitted to Kaggle utilized Model 2. Therefore, Model 1 did not receive a MiFID score.

We will walk through some of the Key Learnings and Takeaways for these iterations below, including certain functions and hyperparameters.

Table 2

Summary of Results for top CGAN Models

Model	Monet Generator Loss	MiFID Score	Epochs	Optimizer	Learning Rate	Beta_1
Model 1	2.177	-	25	Adam	2e-4	0.8
Model 2	2.037	60.0	50	Adam	2e-4	0.8

Note: These models also utilized the same pre-processing methods on the images

Summary of Results:

- **Learning Rate:** For the original models, the default Learning Rate (LR) of 0.001 was used. However, the models demonstrated erratic behavior after a dozen epochs. Therefore, a LR of 2e-4 or 0.0002 was subsequently utilized. Model 2 originally used a LR of 1e-4, which led to a higher loss than Model 1 – therefore, Model 2 was re-run with a LR of 2e-4. More experimentation with the Learning Rates could lead to a more optimal result.
- **Beta_1:** Futher experimentation with the beta_1 values could also alter results. In order to see the impact from increased epochs, this value was maintained at 0.8. The default beta_1 value is 0.9. The value of 0.8 was utilized to give reduced weight to the past gradients and more weight to the recent gradients.
- **Epochs:** The key difference between Models 1 and 2 was the number of epochs. The loss declined by increasing the epochs. However, if we expect generative images similar enough to Monet that it would fool an art critic, the loss needs to be closer to zero. Thus, there is definitely room for improvement either with additional epochs or other model adjustments.

Next, let's wrap up with the Conclusion and highlight the key learnings.

9 Section 9: Conclusion

Based on our metrics and results, which model would we select?

In summary, the top model we would select for analyzing and predicting with a lower loss and MiFID score is CGAN Model 2.

However, the Monet-stylized images generated above do not resemble real Monets to even an amateur art enthusiast. Therefore, there is definitely room for improving the model.

Key Learnings and Takeaways

- **Data Augmentation:** Preliminary training of models without data augmentation yielded worse (higher) loss values. Therefore, image rotation, resizing and cropping was added to the preprocessing steps. This data augmentation led to lower loss values.

- **Number of Epochs:** The additional training epochs in CGAN Model 2 helped reduce our loss value (and presumably lower the MiFID score, if images from model 1 were submitted to Kaggle). Thus, further increasing the number of epochs may improve our results.
- **Learning Rates:** Modifying the default Learning Rate for the Adam optimizer appeared to avoid some erratic behavior in preliminary training efforts on the images. Better results were attained with the Learning Rate set to 2e-4. The Learning Rate was reduced further (1e-4) and the loss increased despite adding epochs – in which case, further experimentation is possible.
- **Loss Function:** The loss function was important for tracking the performance throughout the epochs. Different loss functions yield different results. Therefore, further experimentation could yield improved results.

9.1 Other factors to consider

9.1.1 Possible bias or factors that might impact results

Differences in the pre-processing and preparation of images could impact the results. Since this data contained relatively few Monet images compared to photos, we utilized methods for data augmentation such as resizing, cropping and rotating the images. A different approach or method could yield different results.

Another factor that could impact results is the system used for building these models. The iterations can be fairly time consuming and better hardware with more GPUs could process the sames models in reduced time – or, allow the execution of more complex models.

9.1.2 Areas for consideration

Some areas that would be interesting to explore further might include:

- Experimentation with different Learning Rates and Learning Rate Schedules
- Evaluating how Pretrained Models from KerasHub performed on this data
- Using different customized Keras methods
- Employing other machine learning models

Overall, it was interesting to see how Generative Adversarial Networks performed on this data.

Thank you for reading!

10 Section 10: Resources

Websites:

- https://keras.io/examples/generative/cyclegan/
- https://www.tensorflow.org/tutorials/generative/cyclegan
- https://arxiv.org/abs/1505.04597
- https://arxiv.org/pdf/1606.03498
- https://blog.keras.io/building-autoencoders-in-keras.html

- https://keras.io/api/layers/convolution_layers/convolution2d/
- https://keras.io/api/layers/normalization_layers/batch_normalization/
- https://keras.io/api/layers/pooling_layers/max_pooling2d/
- https://keras.io/api/layers/regularization_layers/dropout/
- https://keras.io/api/optimizers/
- https://www.kaggle.com/code/amyjang/monet-cyclegan-tutorial
- https://jonathan-hui.medium.com/gan-cyclegan-6a50e7600d7
- https://jonathan-hui.medium.com/gan-a-comprehensive-review-into-the-gangsters-of-gans-part-2-73233a670d19

Github Link: https://github.com/chiffr3/GAN

```
[ ]:
```