DTSA-5511: Week 5 - Kaggle Mini Project

September 28, 2022

Generative Adversarial Networks

Introduction

This is the Week 5 assignment for DTSA-5511, Introduction to Deep Learning. This project applies a Generative Adversarial Network (GAN) to an image dataset in an attempt to perform a style transfer from photographs to an image that appears as though it was painted by Monet. The dataset comes from a Kaggle Competition and consists of an image set consisting of 7,038 photographs and and image set consisting of 300 Monet paintings. Each of these images is 256x256 pixels and comprised of 3 channels of 8-bits each for the red, green, and blue values (i.e. a 24-bit 256x256 image).

What is a GAN?

A Generative Adversarial Network is a machine learning model consisting of two neural networks which compete against each other to become more accurate in their predictions.[1] One of the neural networks generates "fake" data in attempt to pass it off as real and the other neural network attempts to detect what data is real and what is fake. These two neural networks, the generator and the discriminator respectively, form a feedback loop and in the process each of the two neural networks get better at its individual purpose.

In addition to the image translation being done in this project, other things GAN's have been used for are:

- Filling in images from an outline
- Generating a realistic image from text
- Producing photorealistic depictions of product prototypes
- Converting black and white imagery into color

This Jupyter notebook can be downloaded from https://github.com/clayv/DTSA-5511_Week5.

References:
[1]TechTarget. Generative Adversarial Network. https://www.techtarget.com/searchenterpriseai/definition/generative-adversarial-network-GAN

Kaggle. Monet CycleGAN Tutorial, by Amy Jang.
https://www.kaggle.com/code/amyjang/monet-cyclegan-tutorial/notebook

In [28]:
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import tensorflow_addons as tfa

from kaggle_datasets import KaggleDatasets
import matplotlib.pyplot as plt
import numpy as np
import os
import random

AUTOTUNE = tf.data.AUTOTUNE
RANDOM_SEED = 42

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

```
TensorFlow version: 2.10.0
```

Exploratory Data Analysis

Before exploring the image sets, the data will be loaded and some helper functions for working with images will are created

In [6]:
```
#GLOBALS
IMAGE_SIZE = [256, 256]
OUTPUT_CHANNELS = 3
INITIAL_MEAN = 0.0
INITIAL_STD_DEV = 0.05 #0.02

#DATA_DIR = "./data"
#TEMP_DIR = "./fake_monets"
DATA_DIR = KaggleDatasets().get_gcs_path()
TEMP_DIR = "/kaggle/temp/"

MONET_FILENAMES = tf.io.gfile.glob(os.path.join(DATA_DIR + '/monet_tfrec/*.tfrec'))
print('Monet TFRecord Files:', len(MONET_FILENAMES))
print("Monet images:", sum(1 for _ in tf.data.TFRecordDataset(MONET_FILENAMES)))
```

```
PHOTO_FILENAMES = tf.io.gfile.glob(os.path.join(DATA_DIR + '/photo_tfrec/*.tfrec'))
print('Photo TFRecord Files:', len(PHOTO_FILENAMES))
print("Photo images:", sum(1 for _ in tf.data.TFRecordDataset(PHOTO_FILENAMES)))

KERNEL_INITIALIZER = tf.random_normal_initializer(INITIAL_MEAN, INITIAL_STD_DEV, seed = RANDOM_SEED)
#KERNEL_INITIALIZER = tf.random_uniform_initializer(minval= 0.0, maxval = 1.0, seed = RANDOM_SEED)
GAMMA_INTIALIZER = keras.initializers.RandomNormal(mean = INITIAL_MEAN, stddev = INITIAL_STD_DEV, seed = RANDOM_SEED)
```

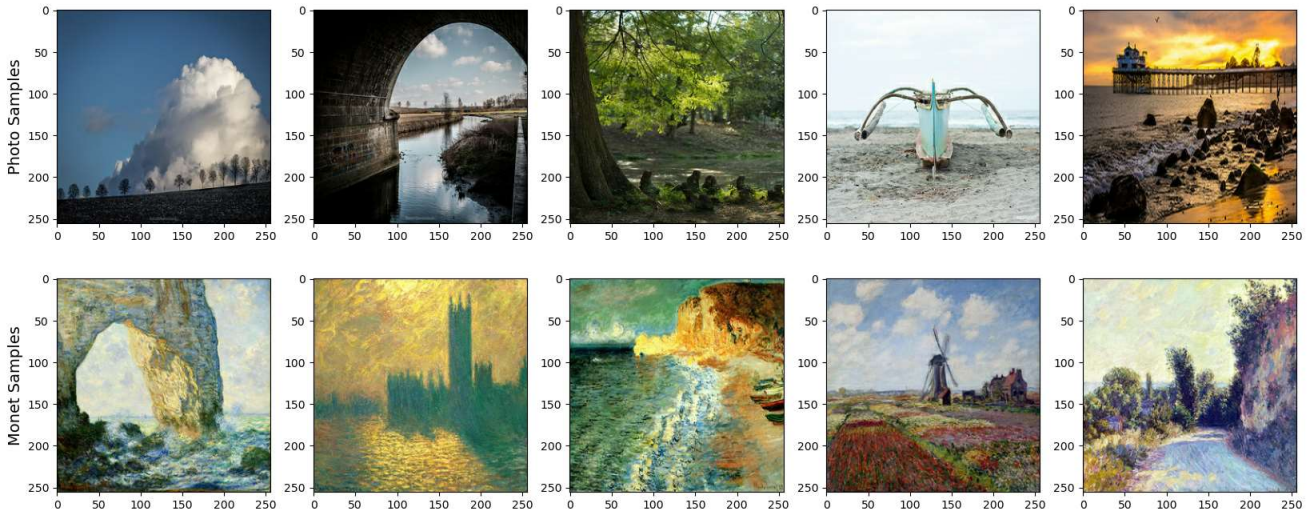
```
Monet TFRecord Files: 5
Monet images: 300
Photo TFRecord Files: 20
Photo images: 7038
```

In [7]:
```
#Image Helper Functions
def encodeImage(image):
    image = tf.image.decode_jpeg(image, channels=3)
    #As the RGB values vary from 0 to 255 inclusive, this next
    #line will alter the values to range from -1 to 1 for training
    image = (tf.cast(image, tf.float32) / 127.5) - 1
    image = tf.reshape(image, [*IMAGE_SIZE, 3])
    return image

def decodeImage(image, maxValue = 1):
    #This converts the -1 to 1 values to 0 to 1 for use with imshow
    image = (image * 0.5) + 0.5
    if maxValue == 255:
        #This converts the now 0 to 1 values to 0 to 255 for use with
        #PIL.Image when saving as a JPEG for submission to Kaggle
        image = (image * 255).astype(np.uint8)

    return image


def read_tfrecord(record):
    tfrecord_format = {
        "image_name": tf.io.FixedLenFeature([], tf.string),
        "image": tf.io.FixedLenFeature([], tf.string),
        "target": tf.io.FixedLenFeature([], tf.string)
    }
    record = tf.io.parse_single_example(record, tfrecord_format)
    image = encodeImage(record['image'])
    return image
```

In [8]:
```
def load_dataset(filenames, labeled = True, ordered = False):
    dataset = tf.data.TFRecordDataset(filenames)
    dataset = dataset.map(read_tfrecord, num_parallel_calls = AUTOTUNE)
    return dataset

monet_ds = load_dataset(MONET_FILENAMES, labeled = True).batch(1)
photo_ds = load_dataset(PHOTO_FILENAMES, labeled = True).batch(1)
```

The code in the following cell takes the first 5 photographs and Monet's and displays them to ensure we are reading the data correctly and see the types of images in the dataset.

In [9]:
```
numImages = 5
def makeImageRow(ax, row, ds, generate = True):
    for i, image in enumerate(ds.take(numImages)):
        #imshow will work with integer values 0 - 255 _or_
        #floats from 0 - 1.  This next line will convert
        #the float values -1 to 1 created in the
        #_encodeImage_ function to a range between 0 - 1
        ax[row, i].imshow(decodeImage(image[0]))
        if generate:
            prediction = m_generator(image, training = False)[0].numpy()
            ax[1, i].imshow(decodeImage(prediction))

def showSamples():
    fig, ax = plt.subplots(2, numImages, figsize = (20, 8))
    fig.suptitle('Sample Images', fontsize = 20)

    makeImageRow(ax, 0, photo_ds, generate = False)
    makeImageRow(ax, 1, monet_ds, generate = False)
    ax[0, 0].set_ylabel('Photo Samples', fontsize='14')
    ax[1, 0].set_ylabel('Monet Samples',  fontsize='14')

    plt.show()

showSamples()
```



The second row displays the Monet's and allows us to see the impressionist style of painting that Monet founded and is known for. As mentioned earlier there are 7,038 photos and 300 Monets in the dataset each one being 256x256 pixels with 3 8-bit color channels.

As this is a curated dataset, no data cleanup is necessary.

Additionally, this project requires the use of a CycleGAN which measures loss in the cycle of translating from one image to another and back again. As such, the CycleGAN Tutorial will be used as a base model to work with.

In examining the model in that tutorial, the generator function would down sample an image to a single 1x1 pixel with 512 nuerons. As Monet was an impressionist and did not use pointillism that seemed too extreme and consequently the last down sampling and first upsampling layers in the generator were commented out of that model.

Model Architecture

The CycleGAN Tutorial was be used as a base model to work with in this project. However, as mentioned previously, some modifications were made to the model as well as extensive code cleaning.

In addition the removal of the last down sampling and first up sampling layers in the generator, a modifiaction was made use an Exponential Decay Learning Rate Scheduler instead of fixed a fixed rate of .0002.

Another major modification to the code is so that progress is displayed at intervals while the training is taking place. This is essential with GAN's as loss curves are not useful when training GAN's.

In [10]:
```
def downsample(filters, size, apply_instancenorm = True):
    layerDownSample = keras.Sequential()
    layerDownSample.add(layers.Conv2D(filters, size, strides = 2, padding = 'same',
        kernel_initializer = KERNEL_INITIALIZER, use_bias = False)
    )

    if apply_instancenorm:
        layerDownSample.add(tfa.layers.InstanceNormalization(gamma_initializer = GAMMA_INTIALIZER))

    layerDownSample.add(layers.LeakyReLU())

    return layerDownSample

def upsample(filters, size, apply_dropout = False):
    layerUpSample = keras.Sequential()
    layerUpSample.add(layers.Conv2DTranspose(filters, size, strides = 2, padding = 'same',
        kernel_initializer = KERNEL_INITIALIZER, use_bias = False)
    )

    layerUpSample.add(tfa.layers.InstanceNormalization(gamma_initializer = GAMMA_INTIALIZER))

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

    layerUpSample.add(layers.ReLU())

    return layerUpSample
```

In [11]:
```
def Generator():
    inputs = layers.Input(shape = [*IMAGE_SIZE, 3])
```

```
    down_stack = [
        downsample(64, 4, apply_instancenorm = False),
        downsample(128, 4),
        downsample(256, 4),
        downsample(512, 4),
        downsample(512, 4),
        downsample(512, 4),
        downsample(512, 4),
#        downsample(512, 4),
    ]

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

    last = layers.Conv2DTranspose(OUTPUT_CHANNELS, 4, strides = 2,
         padding = 'same', kernel_initializer = KERNEL_INITIALIZER, activation = 'tanh'
    )

    outputs = inputs

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

    skips = reversed(skips[:-1])

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

    outputs = last(outputs)

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

In [12]:
```
def Discriminator():
    inputLayer = layers.Input(shape = [*IMAGE_SIZE, 3], name = 'input_image')

    down1 = downsample(64, 4, False)(inputLayer)
    down2 = downsample(128, 4)(down1)
    down3 = downsample(256, 4)(down2)

    zero_pad1 = layers.ZeroPadding2D()(down3)
    conv = layers.Conv2D(512, 4, strides=1, kernel_initializer = KERNEL_INITIALIZER, use_bias = False)(zero_pad1)

    normalized = tfa.layers.InstanceNormalization(gamma_initializer = GAMMA_INTIALIZER)(conv)
    leaky_relu = layers.LeakyReLU()(normalized)
    zero_pad2 = layers.ZeroPadding2D()(leaky_relu)
    last = layers.Conv2D(1, 4, strides = 1, kernel_initializer = KERNEL_INITIALIZER)(zero_pad2)

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

In [13]:
```
m_generator = Generator() #Generates Monet's from photos
p_generator = Generator() #Generates photos from Monets <- Used in cycling back to photo for loss calculations
m_discriminator = Discriminator() #Discriminates real Monet's from fake ones
```

The following cell shows the training progress at the end of the passed epoch. But as the model has begun getting trained yet, it displays the progress at "Epoch: 0", which is what simply the photographs in their initial state. For the "keen-eyed" reader, you may notice that the initial images below show more detail that the ones in the sample tutorial. There is a reason for this and it will be covered later in this notebook.

In [14]:
```
def showProgress(epoch = 0):
    fig, ax = plt.subplots(2, numImages, figsize = (20, 8))
    fig.suptitle('Progress at Epoch: {0}'.format(epoch), fontsize = 20)

    makeImageRow(ax, 0, photo_ds, generate = True)
    ax[0, 0].set_ylabel('Original', fontsize='14')
    ax[1, 0].set_ylabel('Monet-esque',  fontsize='14')

    plt.show()

showProgress()
print("The second row shows what the Monet generator is creating prior to _any_ training. It is the 'initialized' image.")
```

The second row shows what the Monet generator is creating prior to _any_ training. It is the 'initialized' image.

The cell below contains the custom training cycle which will be called during the model fit. A custom training cycle is required to enable the feedback loop between the generator and discriminator. The train_step function itself has been modified from the tutorial to eliminate all code which trains the ability of the model to convert Monet paintings to photos.

In [15]:
```
class PaintingCycleGan(keras.Model):
    def __init__(self, m_generator, p_generator, m_discriminator, lambda_cycle=10):
        super(PaintingCycleGan, self).__init__()
        self.m_gen = m_generator
        self.p_gen = p_generator
        self.m_disc = m_discriminator
        self.lambda_cycle = lambda_cycle

    def compile(self, m_gen_optimizer, m_disc_optimizer, gen_loss_fn, disc_loss_fn, cycle_loss_fn, identity_loss_fn):
        super(PaintingCycleGan, self).compile()
        self.m_gen_optimizer = m_gen_optimizer
        self.m_disc_optimizer = m_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

    @tf.function
    def train_step(self, batch_data):
        real_monet, real_photo = batch_data

        with tf.GradientTape(persistent = True) as tape:
            #Cycle a photo into a Monet and then back to a photo
            fake_monet = self.m_gen(real_photo, training = True)
            cycled_photo = self.p_gen(fake_monet, training = True)

            #Try to generate a Monet from a Monet
            same_monet = self.m_gen(real_monet, training = True)

            #Call monet discriminator with a _real_ Monet
            disc_real_monet = self.m_disc(real_monet, training = True)

            #Call monet discriminator with a _fake_ Monet
            disc_fake_monet = self.m_disc(fake_monet, training = True)

            #Get generator loss on fake Monet
            monet_gen_loss = self.gen_loss_fn(disc_fake_monet)

            #Get loss on the photo that was turned into a Monet and back
            total_cycle_loss = self.cycle_loss_fn(real_photo, cycled_photo, self.lambda_cycle)

            #Add previous two losses to the loss on the Monet generated from a Monet
            total_monet_gen_loss = monet_gen_loss + total_cycle_loss + \
                self.identity_loss_fn(real_monet, same_monet, self.lambda_cycle)

            #Get discriminator loss between real Monet and fake one
            monet_disc_loss = self.disc_loss_fn(disc_real_monet, disc_fake_monet)

        # Calculate the gradients for generator and discriminator
        m_generator_gradients = tape.gradient(total_monet_gen_loss, self.m_gen.trainable_variables)
        m_discriminator_gradients = tape.gradient(monet_disc_loss, self.m_disc.trainable_variables)
```

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

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

        return {"monet_gen_loss": total_monet_gen_loss,  "monet_disc_loss": monet_disc_loss}
```

The various loss functions are defined in the next cell and will be passed into the *compile* function of the PaintingCycleGan class defined in the cell direct above this one.

In [16]:
```
def generator_loss(generated):
    return tf.keras.losses.BinaryCrossentropy(from_logits = True,
        reduction = tf.keras.losses.Reduction.NONE)(tf.ones_like(generated), generated)

def discriminator_loss(real, generated):
    real_loss = tf.keras.losses.BinaryCrossentropy(from_logits = True,
        reduction = tf.keras.losses.Reduction.NONE)(tf.ones_like(real), real)

    generated_loss = tf.keras.losses.BinaryCrossentropy(from_logits = True,
        reduction = tf.keras.losses.Reduction.NONE)(tf.zeros_like(generated), generated)

    return (real_loss + generated_loss) * 0.5

def calc_cycle_loss(real_image, cycled_image, LAMBDA):
    cycleLoss = tf.reduce_mean(tf.abs(real_image - cycled_image))
    return LAMBDA * cycleLoss

def identity_loss(real_image, same_image, LAMBDA):
    identLoss = tf.reduce_mean(tf.abs(real_image - same_image))
    return LAMBDA * 0.5 * identLoss
```

The Exponential Decay learning scheduler is defined in the next cell. However, note that is *not* being used to initialize the Adam optimizer. Again, there is a reason for this and it will be covered later in this notebook.

In [17]:
```
lr_fn = tf.keras.optimizers.schedules.ExponentialDecay(
    1e-4,
    decay_steps = 300,
    decay_rate = 0.96,
    staircase = True)

#m_generator_optimizer = tf.keras.optimizers.Adam(learning_rate = lr_fn, beta_1 = 0.5, beta_2 = 0.999, epsilon=0.0001)
#m_discriminator_optimizer = tf.keras.optimizers.Adam(learning_rate = lr_fn, beta_1 = 0.5, beta_2 = 0.999, epsilon=0.0001)
m_generator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1 = 0.5)
m_discriminator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1 = 0.5)
```

The code in following cell defines a progress callback so that the while the model is training we can visually see if the model is converging. Again this is because loss curves are not useful when training GAN's. With code as it is currently written a sample of 5 images is displayed every 5th epoch.

In [18]:
```
progressInterval = 5

class ProgresssCallback(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        if (epoch + 1) % progressInterval == 0:
            showProgress(epoch + 1)
```

Finally an instance of the model is created, initialized, and compiled.

In [19]:
```
cycle_gan_model = PaintingCycleGan(m_generator, p_generator, m_discriminator)

cycle_gan_model.compile(
    m_gen_optimizer = m_generator_optimizer,
    m_disc_optimizer = m_discriminator_optimizer,
    gen_loss_fn = generator_loss,
    disc_loss_fn = discriminator_loss,
    cycle_loss_fn = calc_cycle_loss,
    identity_loss_fn = identity_loss
)
```

The TensorFlow documentation states that the shuffle argument is ignored when the data being trained on is an object of tf.data.Dataset. As this is the case here, the Monet images are shuffled once before beginning training. Changing "reshuffle_each_iteration" to True visually seemed cause the generator to perform poorly, so it was set to False.

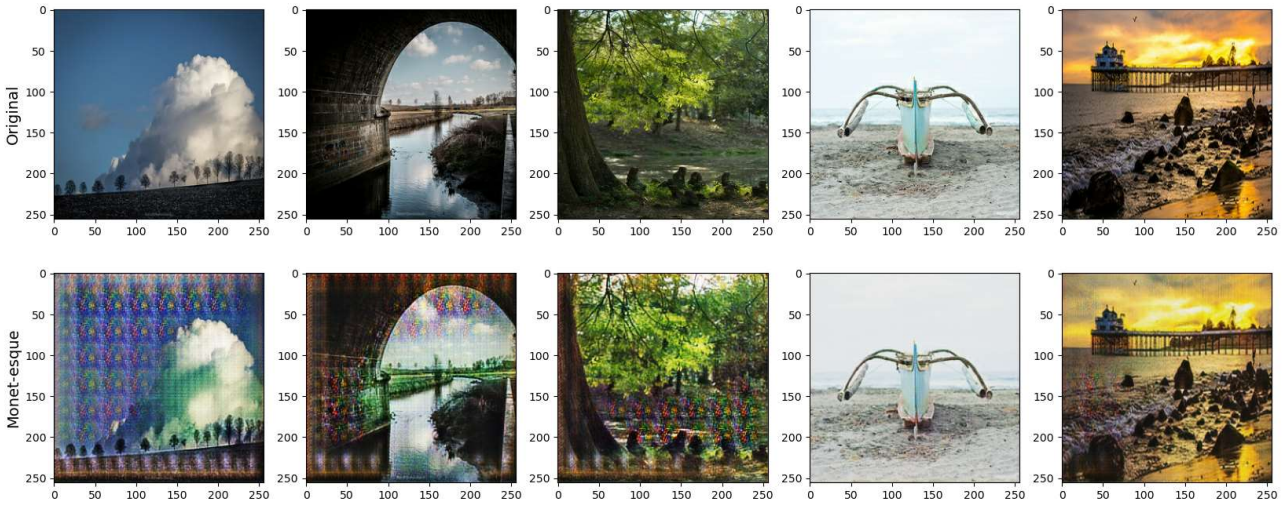
Finally the model is trained for 25 epochs and the progress every 5 epochs is displayed.

In [20]:
```
numEpochs = 25
monet_ds = monet_ds.shuffle(5, seed = RANDOM_SEED, reshuffle_each_iteration = False)

hist = cycle_gan_model.fit(
```

```
    tf.data.Dataset.zip((monet_ds, photo_ds)),
    epochs = numEpochs, verbose = True, callbacks = [ProgresssCallback()]
)

if numEpochs % progressInterval != 0:
    showProgress(numEpochs)
```

```
Epoch 1/25
300/300 [==============================] - 74s 202ms/step - monet_gen_loss: 5.6665 - monet_disc_loss: 0.6203
Epoch 2/25
300/300 [==============================] - 60s 200ms/step - monet_gen_loss: 5.5757 - monet_disc_loss: 0.6118
Epoch 3/25
300/300 [==============================] - 60s 201ms/step - monet_gen_loss: 5.5160 - monet_disc_loss: 0.6353
Epoch 4/25
300/300 [==============================] - 60s 199ms/step - monet_gen_loss: 5.4814 - monet_disc_loss: 0.6344
Epoch 5/25
300/300 [==============================] - ETA: 0s - monet_gen_loss: 5.4492 - monet_disc_loss: 0.6350
```

Progress at Epoch: 5




```
300/300 [==============================] - 61s 204ms/step - monet_gen_loss: 5.4489 - monet_disc_loss: 0.6359
Epoch 6/25
300/300 [==============================] - 60s 200ms/step - monet_gen_loss: 5.4235 - monet_disc_loss: 0.6343
Epoch 7/25
300/300 [==============================] - 60s 200ms/step - monet_gen_loss: 5.3940 - monet_disc_loss: 0.6400
Epoch 8/25
300/300 [==============================] - 60s 199ms/step - monet_gen_loss: 5.3520 - monet_disc_loss: 0.6392
Epoch 9/25
300/300 [==============================] - 60s 200ms/step - monet_gen_loss: 5.3088 - monet_disc_loss: 0.6467
Epoch 10/25
300/300 [==============================] - ETA: 0s - monet_gen_loss: 5.3148 - monet_disc_loss: 0.6409
```

Progress at Epoch: 10

```
300/300 [==============================] - 61s 204ms/step - monet_gen_loss: 5.3119 - monet_disc_loss: 0.6419
Epoch 11/25
300/300 [==============================] - 60s 200ms/step - monet_gen_loss: 5.2907 - monet_disc_loss: 0.6439
Epoch 12/25
300/300 [==============================] - 60s 200ms/step - monet_gen_loss: 5.2502 - monet_disc_loss: 0.6479
Epoch 13/25
300/300 [==============================] - 60s 200ms/step - monet_gen_loss: 5.2385 - monet_disc_loss: 0.6448
Epoch 14/25
300/300 [==============================] - 60s 200ms/step - monet_gen_loss: 5.2819 - monet_disc_loss: 0.6400
Epoch 15/25
300/300 [==============================] - ETA: 0s - monet_gen_loss: 5.2813 - monet_disc_loss: 0.6407
```

Progress at Epoch: 15


```
300/300 [==============================] - 61s 204ms/step - monet_gen_loss: 5.2788 - monet_disc_loss: 0.6414
Epoch 16/25
300/300 [==============================] - 60s 199ms/step - monet_gen_loss: 5.2207 - monet_disc_loss: 0.6411
Epoch 17/25
300/300 [==============================] - 60s 200ms/step - monet_gen_loss: 5.2011 - monet_disc_loss: 0.6419
Epoch 18/25
300/300 [==============================] - 60s 200ms/step - monet_gen_loss: 5.1941 - monet_disc_loss: 0.6415
Epoch 19/25
300/300 [==============================] - 60s 200ms/step - monet_gen_loss: 5.1917 - monet_disc_loss: 0.6420
Epoch 20/25
300/300 [==============================] - ETA: 0s - monet_gen_loss: 5.1836 - monet_disc_loss: 0.6409
```

Progress at Epoch: 20


```
300/300 [==============================] - 61s 204ms/step - monet_gen_loss: 5.1816 - monet_disc_loss: 0.6417
Epoch 21/25
300/300 [==============================] - 60s 200ms/step - monet_gen_loss: 5.2500 - monet_disc_loss: 0.6359
Epoch 22/25
300/300 [==============================] - 60s 201ms/step - monet_gen_loss: 5.1952 - monet_disc_loss: 0.6387
Epoch 23/25
300/300 [==============================] - 60s 201ms/step - monet_gen_loss: 5.1907 - monet_disc_loss: 0.6386
Epoch 24/25
300/300 [==============================] - 60s 200ms/step - monet_gen_loss: 5.1946 - monet_disc_loss: 0.6361
Epoch 25/25
300/300 [==============================] - ETA: 0s - monet_gen_loss: 5.1941 - monet_disc_loss: 0.6370
```


```
300/300 [==============================] - 61s 204ms/step - monet_gen_loss: 5.1916 - monet_disc_loss: 0.6379
```

These next two cells output the final fake Monet's and then create a ZIP archive file for submission to Kaggle.

In [21]:
```
import PIL

try:
    os.mkdir(TEMP_DIR)
except:
    pass

for i, img in enumerate(photo_ds):
    prediction = m_generator(img, training = False)[0].numpy()
    prediction = decodeImage(prediction, maxValue = 255)
    im = PIL.Image.fromarray(prediction)
    im.save(os.path.join(TEMP_DIR, str(i) + ".jpg"))
```

In [20]:
```
import shutil
shutil.make_archive("/kaggle/working/images", "zip", TEMP_DIR)
```

Out[20]:
```
'/kaggle/working/images.zip'
```

Hyperparameter Tuning

One of the hyperparameters tuned was already described when it was decided to set "reshuffle_each_iteration" to False. However, during this project lots of tests were performed and hyperparameters tried.

One of these was the learning rate. As noted above, and Eponential Decay learning rate was applied. The learning rate scheduler was introduced because when I ran the tutorial's model originally, the seemed to be too much color "blooms" in the fake Monet's. To combat this I theorized that as the learning rate was not decreasing over time that is what was causing these blooms from the model continually overshooting it target. Various initial learning rates were tried, and finally a value of 2e-5 was settled on with the number of steps set to 50.

It is worth mentioning, that this was done when testing on a small subset of 10 photos. With 7,038 photos in the dataset I choose to try testing on a smaller subset to speed up the training time. However, I noticed when expanding to the full dataset the the progress was visually different - which in hindsight makes sense... a GAN creates a feedback loop so if the number of photos is reduced, so is the nature of the feedback. After figuring this out, from that time forward I only trained on the full photo image set. But, I didn't change the number of steps in the Exponential Decay scheduler. Because of the increase in photos trained the number of steps increased to 300 per epoch which meant my learning rate was decaying much faster that previously so the later epochs were showing almost no change.

After adjusting the steps for the learning rate and with what I thought visually would score well, I attempted to submit my image.zip file to Kaggle. It was at this time that I discovered that the notebook used to create the images must be created and run on the Kaggle website and it is not possible to upload an image.zip file. So I imported my notebook into Kaggle and ran it there. Once doing that though, I noticed that the progress images where running on Kaggle looked different than when running the notebook locally. So any previous hyperparameter tuning was now invalid and had to be done again. Now running on Kaggle, the learning rate was again adjusted until I settled on an intial learning rate of 5e-6 (down from 2e-5) and 300 steps, which would cause at decay at the end of each epoch.

Additional hyperparameter tuning included experiments with different kernel initializers, including random_uniform_initializer (-1 to 1) and LecunNormal, but in the end the original random_normal_initializer seemed to give the best visual results. The model was then submitted and received a score of 100.69810.

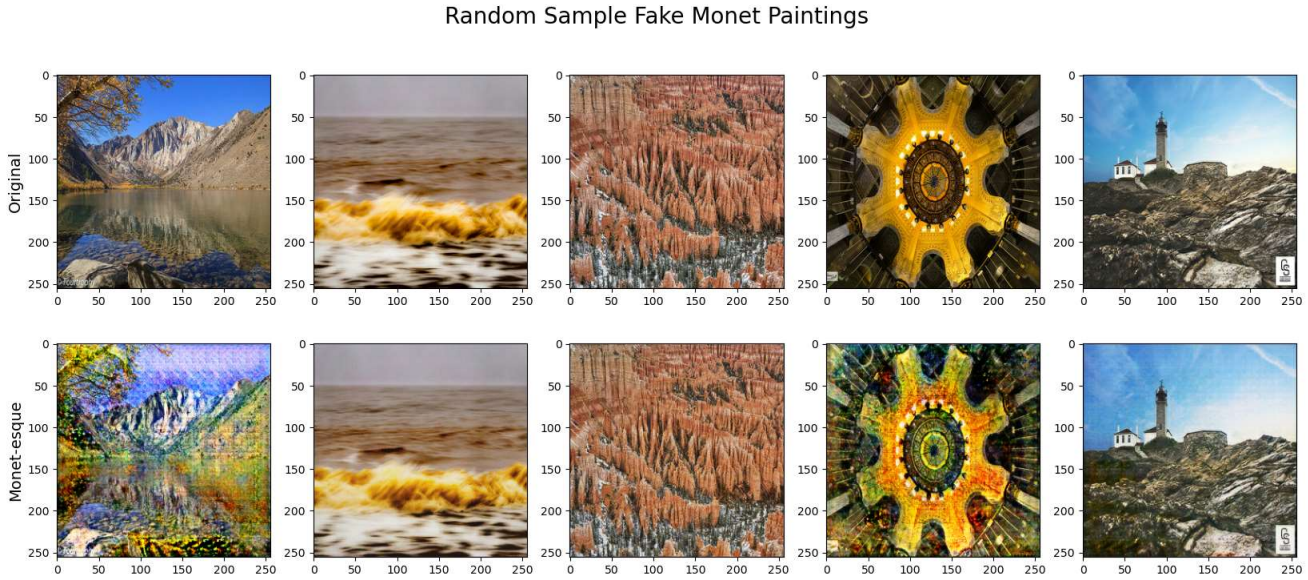
I was not pleased with this score and attempted additional hyperparameter tuning. This went through several more iterations, when on a lark I decided to go back to an fixed learning rate of 2e-4 in the Adam optimizer and instead change the kernel and gamma initializers to use a standard deviation of 0.05 instead of 0.02. I ran the model and it seemed to produce good results visually, so I submitted it and received a score of 61.14871 (in this competition, a lower score is better).

It was this model that I decided to finalize this project with and that is the learning rate function is not used and the "Epoch: 0" images look different.

Results and Analysis

The cell below displays 5 random original photos from the photos dataset and their "Monet-esque" versions.

In [90]:
```
index = [random.randint(0, 7039) for i in range(numImages)]

fig, ax = plt.subplots(2, numImages, figsize = (20, 8))
fig.suptitle('Random Sample Fake Monet Paintings', fontsize = 20)

j = 0
for i, image in enumerate(photo_ds):
    if i in index:
        ax[0, j].imshow(decodeImage(image[0]))
        prediction = m_generator(image, training = False)[0].numpy()
        ax[1, j].imshow(decodeImage(prediction))
        j += 1

ax[0, 0].set_ylabel('Original', fontsize='14')
ax[1, 0].set_ylabel('Monet-esque',  fontsize='14')

plt.show()
```



Visually the images look perhaps a bit "over"-trained, but time constraints prevented continued experimentation.

On tuning the hyperparameters, the final model did not vary much from the tutorial sample, the sole exceptions being the change to the standard deviation for the kernel and gamma intializers from 0.02 to 0.05 and the removal of the last down sample and first up sample layers in the generator.

The change in the standard deviation gave the best score increase and I believe this is due to the giving slightly more color information that when it was set at 0.02. Larger numbers, e.g. 0.1, were tried but this resulted in too much color blooming. This blooming might be counteracted by training for fewer epochs though.

Conclusion

Overall the "Monet-esque" images came out well and I was pleased with the competition score of 61.14871. This project cemented several ideas though: (1) training GAN's can take a long time (2) small changes in initialization can have a major impact on final results and (3) parameterized learning schedules can be impacted by the amount of training data.

An unfortunate downside of training GAN's is that they really must be trained on the full dataset as the feedback loop will be altered when going from a subset to the full dataset, negating any previous work on hyperparameter tuning.

To improve the model further I think my first step would be to implement the MiFID (Memorization-informed FID) Kaggle uses to score this competition. By doing this I could get my score more quickly than with a submission, which are also limited to 5 per day. Another area for improvement would be to implement a "dual head" discriminator. I saw this in a few of the top scoring submssions, including this one, and it is a technique I was not aware of prior to participating in this competition.

Lastly, I'd like to try submitting this exact same model and hyperparameter tuning but after only 20 epochs as I think those results are more visually pleasing, but would like to see how the Kaggle scoring algorithm rates them.