Monet using GAN

Below the prerequisite modules, data and HW are set up.

Note: this notebook has been run on local computer(s) and on Kaggle. Some of the cells made for Kaggle won't work locally and vice versa.

In [2]:
```
import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
        break
```

In [1]:
```
import os
import matplotlib.pyplot as plt
import time
import seaborn as sns
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
# import tensorflow_addons as tfa
# from kaggle_datasets import KaggleDatasets

if os.path.exists(r'C:\Users\kuusnin\tempwork\temp\gan-getting-started'):
    datapath = r'C:\Users\kuusnin\tempwork\temp\gan-getting-started'
elif os.path.exists(r'C:\Users\nikok\Documents\Monet using GAN'):
    datapath = r'C:\Users\nikok\Documents\Monet using GAN'
else:
    datapath = r'/kaggle/input/gan-getting-started'
print(datapath)
```

```
C:\Users\nikok\Documents\Monet using GAN
```

In [4]:
```
tf.__version__
try:
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver('local')
    print('Device:', tpu.master())
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.TPUStrategy(tpu)
except Exception as e:
    print("can't initialize tpu, using default, exception: " + str(e))
    strategy = tf.distribute.get_strategy()
print('Number of replicas:', strategy.num_replicas_in_sync)
```

```
Device:
INFO:tensorflow:Deallocate tpu buffers before initializing tpu system.
INFO:tensorflow:Initializing the TPU system: local
can't initialize tpu, using default, exception: TPUs not found in the cluster. Faile
d in initialization: No OpKernel was registered to support Op 'ConfigureDistributedT
PU' used by {{node ConfigureDistributedTPU}} with these attrs: [embedding_config="",
compilation_failure_closes_chips=false, tpu_embedding_config="", is_global_init=fals
e, enable_whole_mesh_compilations=false, tpu_cancellation_closes_chips=2]
Registered devices: [CPU]
Registered kernels:
  <no registered kernels>

         [[ConfigureDistributedTPU]] [Op:__inference__tpu_init_fn_4]
Number of replicas: 1
```

In [4]:
```
print("Available GPUs:", tf.config.list_physical_devices('GPU'))
strategy = tf.distribute.MirroredStrategy()
```

```
Available GPUs: []
```

Brief description of the problem and data (5 pts)

Briefly describe the challenge problem and NLP. Describe the size, dimension, structure, etc., of the data.

The objective is to generate 7000+ Monet style paintings either by random seed or using the ~7000 photos provided. There are 300 Monet paintings to train the model with.

The data is available in two formats: two sets of regular jpg image files and two sets of tfrec-files. "*The TFRecord format is a simple format for storing a sequence of binary records. Converting your data into TFRecord has many advantages, such as: More efficient storage: the TFRecord data can take up less space than the original data; it can also be partitioned into multiple files.*" (https://keras.io/examples/keras_recipes/creating_tfrecords/)

Since the TFRecord format is more efficient this work is used as an exercise in usage of TFRecord. Below, the TFRecord filenames are saved and counted.

In [5]:
```
monet_files = os.listdir(os.path.join(datapath, 'monet_tfrec'))
photo_files = os.listdir(os.path.join(datapath, 'photo_tfrec'))
monet_filenames = [os.path.join(datapath, 'monet_tfrec', f) for f in monet_files]
photo_filenames = [os.path.join(datapath, 'photo_tfrec', f) for f in photo_files]
print(20*'*', 'Monet paintings', 20*'*')
print('First filename:', monet_files[0], '\nNumber of files:', len(monet_files))
print(20*'*', 'Photos', 20*'*')
print('First filename:', photo_files[0], '\nNumber of files:', len(photo_files))
```

```
******************** Monet paintings ********************
First filename: monet00-60.tfrec
Number of files: 5
******************** Photos ********************
First filename: photo00-352.tfrec
Number of files: 20
```

Figure out the contents of the tfrec files

We do not always know the content of the TFRecord files so a exploratory code below inspects the files.

From the code below we can see that each record/example contains three fields:

- target: label of the image. Not needed in this work
- image_name: name of the image
- image: the actual image data

Both of the data sets appear to have the same structure.

In [6]:
```
# The following code is adapted from an answer from Microsoft Copilot
import tensorflow as tf
from google.protobuf.json_format import MessageToJson
import json

def iterate_record(dataset):
    # Initialize a counter
    record_count = 0
    # Iterate through the dataset and count the records
    for _ in dataset:
        record_count += 1
    print(f'Total number of records: {record_count}')

    # Iterate through the dataset and parse each record
    for raw_record in dataset.take(1):  # Adjust the number to read more records
        example = tf.train.Example()
        example.ParseFromString(raw_record.numpy())
        json_message = MessageToJson(example)
        parsed_record = json.loads(json_message)
        print(json.dumps(parsed_record, indent=2)[0:500])

# Create a TFRecordDataset
print(20*'*', 'Monet paintings', 20*'*')
raw_monet_dataset = tf.data.TFRecordDataset(monet_filenames)
iterate_record(raw_monet_dataset)

print('\n' + 20*'*', 'Photos', 20*'*')
raw_photo_dataset = tf.data.TFRecordDataset(photo_filenames)
iterate_record(raw_photo_dataset)
```

```
******************* Monet paintings *******************
Total number of records: 300
```

```
{
  "features": {
    "feature": {
      "image_name": {
        "bytesList": {
          "value": [
            "MjVjOTkwNDc4Mg=="
          ]
        }
      },
      "image": {
        "bytesList": {
          "value": [
            "/9j/4AAQSkZJRgABAQEBLAEsAAD/2wBDAAIBAQEBAQIBAQECAgICAgQDAgICAgUEBAMEBgU
GBgYFBgYGBwkIBgcJBwYGCAsICQoKCgoKBggLDAsKDAkKCgr/2wBDAQICAgICAgUDAwUKBwYHCgoKCgoKCgo
KCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgr/wAARCAEAAQADAREAAhEBAxEB/8Q
AHgAAAgIDAQEBAQAAAAAAAAA
```


```
******************* Photos *******************
Total number of records: 7038
```

```
{
  "features": {
    "feature": {
      "image_name": {
        "bytesList": {
          "value": [
            "MGI5MWYzNTljNQ=="
          ]
        }
      },
      "image": {
        "bytesList": {
          "value": [
            "/9j/4AAQSkZJRgABAQEBLAEsAAD/2wBDAAIBAQEBAQIBAQECAgICAgQDAgICAgUEBAMEBgU
GBgYFBgYGBwkIBgcJBwYGCAsICQoKCgoKBggLDAsKDAkKCgr/2wBDAQICAgICAgUDAwUKBwYHCgoKCgoKCgo
KCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgoKCgr/wAARCAEAAQADAREAAhEBAxEB/8Q
AHQAAAQUBAQEBAAAAAAAAAAA
```


We can parse the data now since the structure of the tfrecord is known. Also batch size of 32 is chosen.

In [7]:
```python
def normalize(image):
    return (tf.cast(image, tf.float32) / 127.5) - 1

def parse_tfrecord_fn(example):
    feature_description = {
        "image_name": tf.io.FixedLenFeature([], tf.string),
        "image": tf.io.FixedLenFeature([], tf.string),
        "target": tf.io.FixedLenFeature([], tf.string)
    }
    example = tf.io.parse_single_example(example, feature_description)
    example["image"] = tf.io.decode_jpeg(example["image"], channels=3)
    example["image"] = normalize(example["image"])
```

```
    return example

def decode_image(tf_image):
    return ((tf_image['image'].numpy() + 1) * 127.5).astype(int)

BATCH_SIZE = 32
monet_dataset = raw_monet_dataset.map(parse_tfrecord_fn, num_parallel_calls=tf.data
photo_dataset = raw_photo_dataset.map(parse_tfrecord_fn, num_parallel_calls=tf.data
```

In [8]:
```
# Pick one example from each dataset
sample_monet = next(iter(monet_dataset))
sample_photo = next(iter(photo_dataset))
```

In [9]:
```
def get_dims(dataset, name):
    dims = []
    for data in dataset:
        dims.append(data['image'].numpy().shape)
    dims = np.array(dims)
    print(f'Number of images in {name} dataset:', np.sum(dims[:,0]))
    print(f'Unique shapes of the {name} data:', np.unique(dims[:,1:], axis=0))

get_dims(monet_dataset, 'Monet')
get_dims(photo_dataset, 'photos')
```

```
Number of images in Monet dataset: 300
Unique shapes of the Monet data: [[256 256   3]]
Number of images in photos dataset: 7038
Unique shapes of the photos data: [[256 256   3]]
```

The code above shows the number of paintings/photos and the size of each image. They are all the same size 256x256 with 3 channels.

In [10]:
```
image_shape = (256, 256, 3)
```

Exploratory Data Analysis (EDA) — Inspect, Visualize and Clean the Data (15 pts)

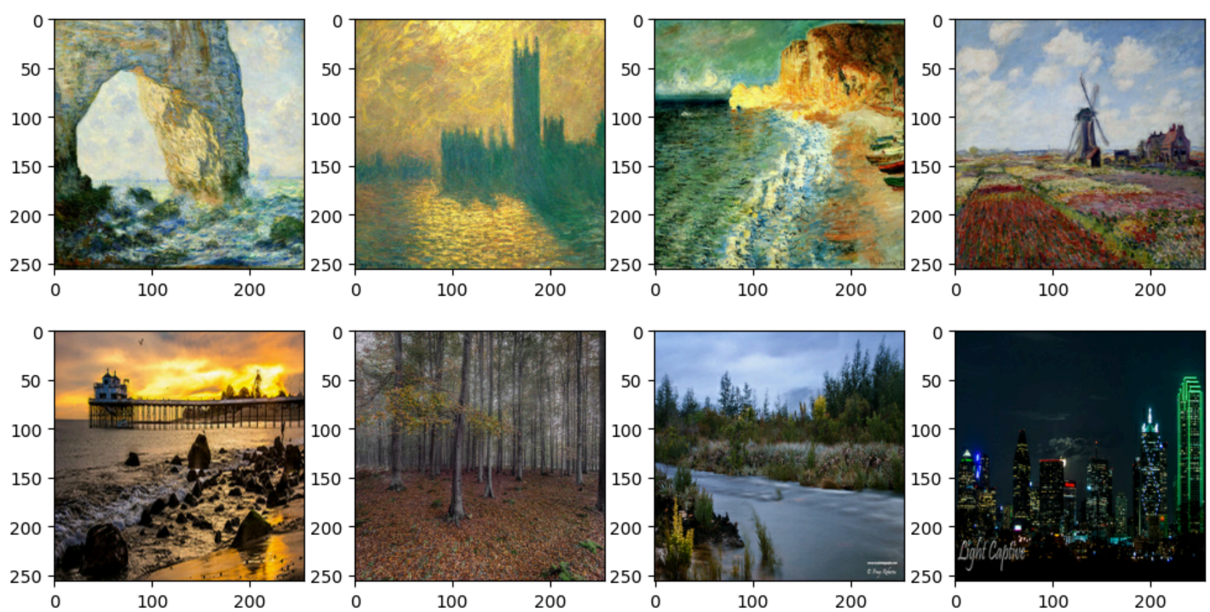
Show a few visualizations like histograms. Describe any data cleaning procedures. Based on your EDA, what is your plan of analysis?

Below a few example images from both datasets are shown.

In [11]:
```
fig = plt.figure(figsize=(12, 6))
i = 0
for i in range(4):
    data = sample_monet['image']
    ax = plt.subplot(2, 4, i + 1)
    plt.imshow(np.array((data[i]+1)*127.5).astype(int))
    i += 1
for data in range(4):
    data = sample_photo['image']
    ax = plt.subplot(2, 4, i + 1)
```

```
    plt.imshow(np.array((data[i]+1)*127.5).astype(int))
    i += 1
fig.suptitle('Examples of Monet and photos')
plt.show()
```

Examples of Monet and photos



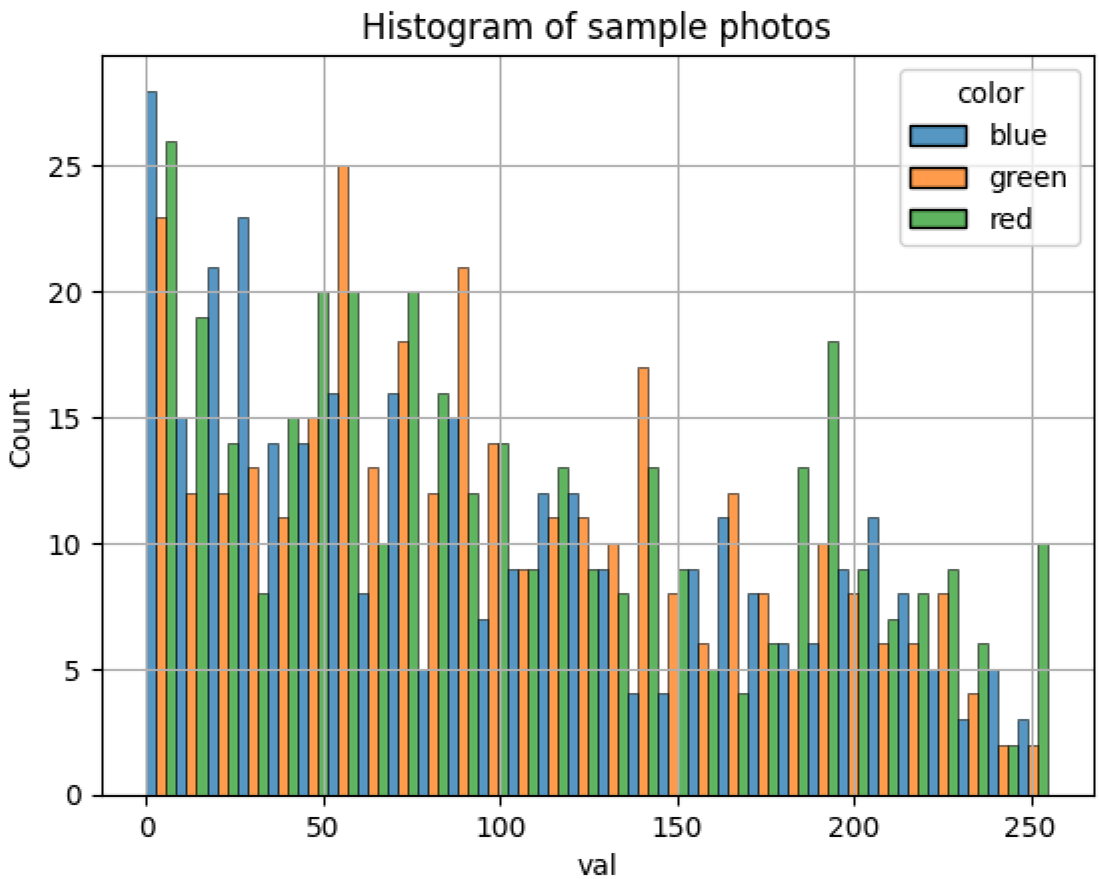
Next, let's check distribution of real image vs Monet paintings.

In [12]:
```
def plot_histogram(dataset, title):
    rgb = []
    rgb = [decode_image(d) for d in dataset.take(9)]
    # for data in dataset:
    #     rgb.append(decode_image(data))
    rgb = np.array(rgb)
    # print(rgb.shape)
    pic_channels = ['red', 'green', 'blue']
    rgb_df = pd.DataFrame()
    for i, clr in enumerate(pic_channels):
        df = pd.DataFrame(rgb[:,:,:,:,i].ravel(), columns=['val'])
        df['color'] = clr
        rgb_df = pd.concat([rgb_df, df])
    print('Minimum and maximmum values in ' + title + ' data: ', min(rgb_df.val), '
    sns.histplot(rgb_df.sample(1000), x='val', hue='color', bins=30, multiple='dodg
    plt.grid()
    plt.title('Histogram of ' + title)
    plt.show()
```

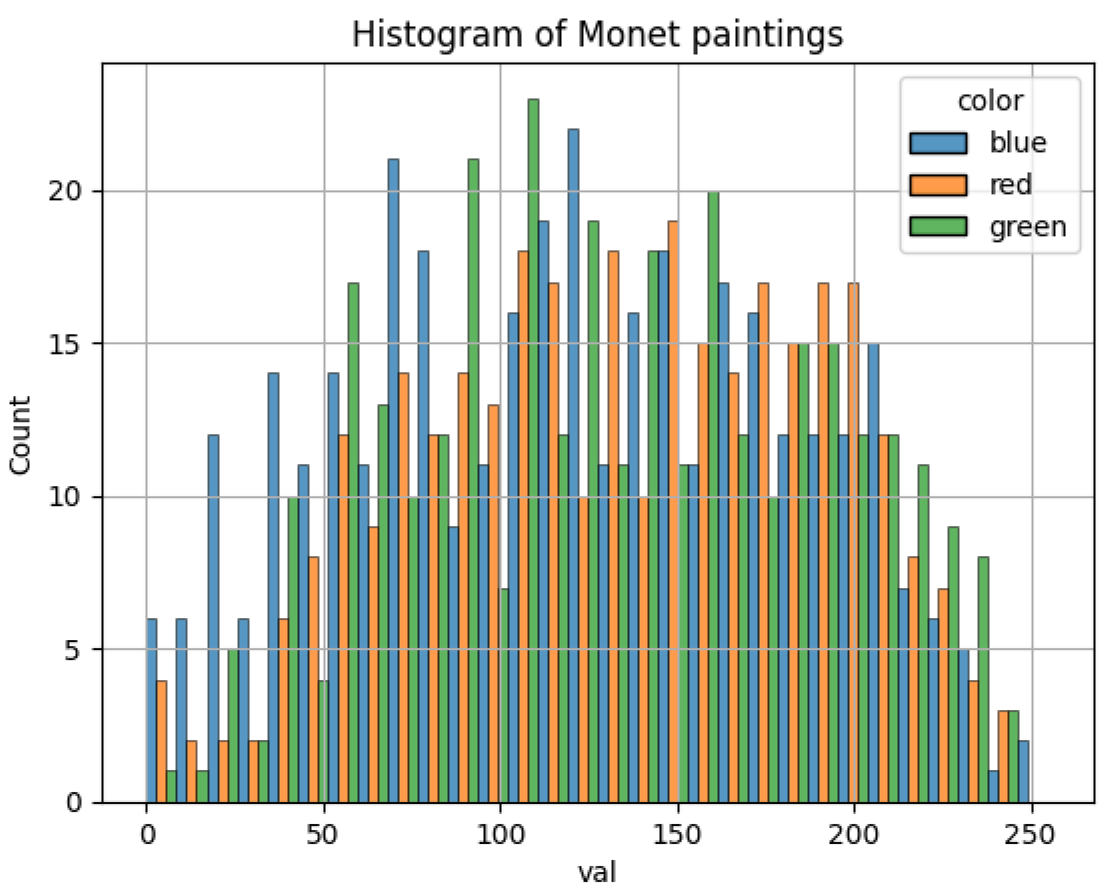
From the histograms we can see how the intensity distibution of the photos is fairly uniform. Where as the Monet paintings are more Gaussian-like distributed. That makes intuitively sense since the Monet painting style is 'blended' and soft compared to the photos.

In [13]:
```
plot_histogram(photo_dataset, title='sample photos')
plot_histogram(monet_dataset, title='Monet paintings')
```

```
Minimum and maximmum values in sample photos data:  0 & 255
```




```
Minimum and maximmum values in Monet paintings data:  0 & 255
```

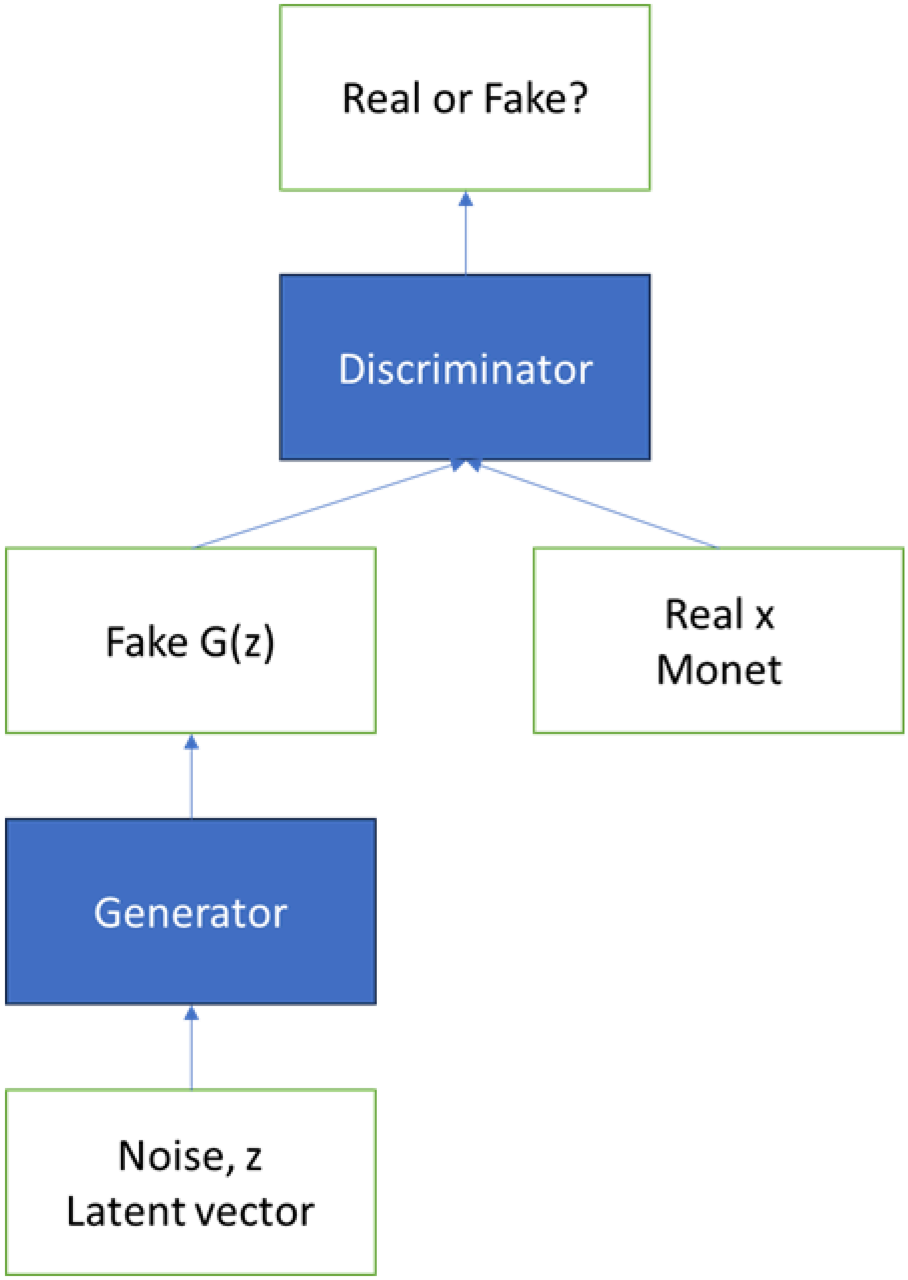


Model Architecture (25 pts)

Describe your model architecture and reasoning for why you believe that specific architecture would be suitable for this problem.

GAN, Generative adversarial network, is suitable for this kind of problem where the objective is to transform items between two domains. The source material for transformation cen either be generated or existing data can be used.

GAN has two main components, Generator and Discriminator. The discriminator tries to distinguish between fake and real object and the generator tries to create real looking objects. The generator and discriminator compete against each other; generator tries to fool discriminator and the discriminator tries to spot the fakes. In this work, the generator tries to create Monet style paintings and the discriminator tries to determine if an image is Monet or not.



Picture adapted from

By Zhang, Aston and Lipton, Zachary C. and Li, Mu and Smola, Alexander J. - https://github.com/d2l-ai/d2l-en, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=152265649

Discriminator

The first attempt to create a discriminator is below. It has three convolutional neural networks (CNN) and the output of each of them is activated by a leaky ReLU. The final layer uses a sigmoid activation to give a probability value between 0 and 1.

Source: https://keras.io/examples/generative/dcgan_overriding_train_step/

In [15]:
```
my_discriminator = keras.Sequential(
    [
        keras.Input(shape=(256, 256, 3)),
        layers.Conv2D(int(64), kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(negative_slope=0.2),
        layers.Conv2D(int(128), kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(negative_slope=0.2),
        layers.Conv2D(int(128), kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(negative_slope=0.2),
        layers.Flatten(),
        layers.Dropout(0.2),
        layers.Dense(1, activation="sigmoid"),
    ],
    name="discriminator",
)
my_discriminator.summary()
```

Model: "discriminator"

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 128, 128, 64)	3,136
leaky_re_lu_3 (LeakyReLU)	(None, 128, 128, 64)	0
conv2d_4 (Conv2D)	(None, 64, 64, 128)	131,200
leaky_re_lu_4 (LeakyReLU)	(None, 64, 64, 128)	0
conv2d_5 (Conv2D)	(None, 32, 32, 128)	262,272
leaky_re_lu_5 (LeakyReLU)	(None, 32, 32, 128)	0
flatten_1 (Flatten)	(None, 131072)	0
dropout_1 (Dropout)	(None, 131072)	0
dense_1 (Dense)	(None, 1)	131,073

Total params: 527,681 (2.01 MB)

Trainable params: 527,681 (2.01 MB)

Non-trainable params: 0 (0.00 B)

The discriminator above did not very well so the improved version below was created. It has now five CNN layers, each with added batch normalization and leaky ReLU stays the same. Number of filters in CNNs is higher. Initially increasing number from 32 to 512, and from 128 to 512 in the final version.

In [17]:
```
with strategy.scope():
    my_discriminator2 = keras.Sequential(
        [
            keras.Input(shape=(256, 256, 3)),
```

```
            layers.Conv2D(int(128), kernel_size=3, strides=2, padding="same"), # wa
            layers.BatchNormalization(),
            layers.LeakyReLU(negative_slope=0.2),
            layers.Conv2D(int(128), kernel_size=3, strides=2, padding="same"), # wa
            layers.BatchNormalization(),
            layers.LeakyReLU(negative_slope=0.2),
            layers.Conv2D(int(128), kernel_size=3, strides=2, padding="same"),
            layers.BatchNormalization(),
            layers.LeakyReLU(negative_slope=0.2),
            layers.Conv2D(int(256), kernel_size=3, strides=2, padding="same"),
            layers.BatchNormalization(),
            layers.LeakyReLU(negative_slope=0.2),
            layers.Conv2D(int(512), kernel_size=3, strides=2, padding="same"),
            layers.BatchNormalization(),
            layers.LeakyReLU(negative_slope=0.2),
            layers.Flatten(),
            layers.Dropout(0.2),
            layers.Dense(1, activation="sigmoid"),
        ],
        name="discriminator",
    )
    my_discriminator2.summary()
```

Model: "discriminator"

Layer (type)	Output Shape	Param #
conv2d_11 (Conv2D)	(None, 128, 128, 128)	3,584
batch_normalization_5 (BatchNormalization)	(None, 128, 128, 128)	512
leaky_re_lu_11 (LeakyReLU)	(None, 128, 128, 128)	0
conv2d_12 (Conv2D)	(None, 64, 64, 128)	147,584
batch_normalization_6 (BatchNormalization)	(None, 64, 64, 128)	512
leaky_re_lu_12 (LeakyReLU)	(None, 64, 64, 128)	0
conv2d_13 (Conv2D)	(None, 32, 32, 128)	147,584
batch_normalization_7 (BatchNormalization)	(None, 32, 32, 128)	512
leaky_re_lu_13 (LeakyReLU)	(None, 32, 32, 128)	0
conv2d_14 (Conv2D)	(None, 16, 16, 256)	295,168
batch_normalization_8 (BatchNormalization)	(None, 16, 16, 256)	1,024
leaky_re_lu_14 (LeakyReLU)	(None, 16, 16, 256)	0
conv2d_15 (Conv2D)	(None, 8, 8, 512)	1,180,160
batch_normalization_9 (BatchNormalization)	(None, 8, 8, 512)	2,048
leaky_re_lu_15 (LeakyReLU)	(None, 8, 8, 512)	0
flatten_3 (Flatten)	(None, 32768)	0
dropout_3 (Dropout)	(None, 32768)	0
dense_3 (Dense)	(None, 1)	32,769

Total params: 1,811,457 (6.91 MB)

Trainable params: 1,809,153 (6.90 MB)

Non-trainable params: 2,304 (9.00 KB)

Generator

Similarly as for discriminator, the first attempt to create a generator is below. It has three 'reverse' convolutional neural networks (CNN) and the output of each of them is activated by a leaky ReLU. The final layer uses a sigmoid activation to give a probability value between 0 and 1. It takes in a latend dimension vector and using the transposed CNN starts to build a image. Finally, a CNN layer is used to map the values into image dimensions (256x256x3).

Source: https://keras.io/examples/generative/dcgan_overriding_train_step/

In [18]:
```
latent_dim = 100
```

In [20]:
```
my_generator = keras.Sequential(
    [
        keras.Input(shape=(latent_dim,)),
        layers.Dense(8 * 8 * 128),
        layers.Reshape((8, 8, 128)),
        layers.Conv2DTranspose(int(128), kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(negative_slope=0.2),
        layers.Conv2DTranspose(int(256), kernel_size=4, strides=4, padding="same"),
        layers.LeakyReLU(negative_slope=0.2),
        layers.Conv2DTranspose(int(512), kernel_size=4, strides=4, padding="same"),
        layers.LeakyReLU(negative_slope=0.2),
        layers.Conv2D(3, kernel_size=5, padding="same", activation="sigmoid"),
    ],
    name="generator",
)
my_generator.summary()
```

Model: "generator"

Layer (type)	Output Shape	Param #
dense_5 (Dense)	(None, 8192)	827,392
reshape_1 (Reshape)	(None, 8, 8, 128)	0
conv2d_transpose_3 (Conv2DTranspose)	(None, 16, 16, 128)	262,272
leaky_re_lu_19 (LeakyReLU)	(None, 16, 16, 128)	0
conv2d_transpose_4 (Conv2DTranspose)	(None, 64, 64, 256)	524,544
leaky_re_lu_20 (LeakyReLU)	(None, 64, 64, 256)	0
conv2d_transpose_5 (Conv2DTranspose)	(None, 256, 256, 512)	2,097,664
leaky_re_lu_21 (LeakyReLU)	(None, 256, 256, 512)	0
conv2d_17 (Conv2D)	(None, 256, 256, 3)	38,403

Total params: 3,750,275 (14.31 MB)

Trainable params: 3,750,275 (14.31 MB)

Non-trainable params: 0 (0.00 B)

The reason why the first set of generators and discriminators did not work well could be because of the sigmoid activation in the end of the generator produces 0 to 1 values whereas the images were coded with noramlized values between -1 and 1.

The Second iteration of the generator below. Now with four transposed convolution layers and last activation was set to tanh to give numbers in -1...1 range.

In [21]:
```
with strategy.scope():
    latent_dim = latent_dim
    my_generator2 = keras.Sequential(
        [
            keras.Input(shape=(latent_dim,)),
            layers.Dense(16 * 16 * 512),
            layers.Reshape((16, 16, 512)),
            layers.Conv2DTranspose(int(256), kernel_size=3, strides=2, padding="sam
            layers.BatchNormalization(),
            layers.LeakyReLU(negative_slope=0.2),
            layers.Conv2DTranspose(int(128), kernel_size=3, strides=2, padding="sam
            layers.BatchNormalization(),
            layers.LeakyReLU(negative_slope=0.2),
            layers.Conv2DTranspose(int(128), kernel_size=3, strides=2, padding="sam
            layers.BatchNormalization(),
            layers.LeakyReLU(negative_slope=0.2),
            layers.Conv2DTranspose(int(128), kernel_size=3, strides=2, padding="sam
            layers.BatchNormalization(),
            layers.LeakyReLU(negative_slope=0.2),
            layers.Conv2D(3, kernel_size=5, padding="same", activation="tanh"),
        ],
        name="generator",
    )
    my_generator2.summary()
```

```
Model: "generator"
```

Layer (type)	Output Shape	Param #
dense_6 (Dense)	(None, 131072)	13,238,272
reshape_2 (Reshape)	(None, 16, 16, 512)	0
conv2d_transpose_6 (Conv2DTranspose)	(None, 32, 32, 256)	1,179,904
batch_normalization_10 (BatchNormalization)	(None, 32, 32, 256)	1,024
leaky_re_lu_22 (LeakyReLU)	(None, 32, 32, 256)	0
conv2d_transpose_7 (Conv2DTranspose)	(None, 64, 64, 128)	295,040
batch_normalization_11 (BatchNormalization)	(None, 64, 64, 128)	512
leaky_re_lu_23 (LeakyReLU)	(None, 64, 64, 128)	0
conv2d_transpose_8 (Conv2DTranspose)	(None, 128, 128, 128)	147,584
batch_normalization_12 (BatchNormalization)	(None, 128, 128, 128)	512
leaky_re_lu_24 (LeakyReLU)	(None, 128, 128, 128)	0
conv2d_transpose_9 (Conv2DTranspose)	(None, 256, 256, 128)	147,584
batch_normalization_13 (BatchNormalization)	(None, 256, 256, 128)	512
leaky_re_lu_25 (LeakyReLU)	(None, 256, 256, 128)	0
conv2d_18 (Conv2D)	(None, 256, 256, 3)	9,603

Total params: 15,020,547 (57.30 MB)

Trainable params: 15,019,267 (57.29 MB)

Non-trainable params: 1,280 (5.00 KB)

In order to further improve the score an additional generator below was created. It is otherwise the same as the 2nd one but add another layer of transposed CNN with 512 filters. Strides of the last layer were adjusted to produce desired image size.

In [22]:
```
with strategy.scope():
    latent_dim = latent_dim
    my_generator3 = keras.Sequential(
        [
```

```
            keras.Input(shape=(latent_dim,)),
            layers.Dense(16 * 16 * 512),
            layers.Reshape((16, 16, 512)),
            layers.Conv2DTranspose(int(512), kernel_size=3, strides=2, padding="sam
            layers.BatchNormalization(),
            layers.LeakyReLU(negative_slope=0.2),
            layers.Conv2DTranspose(int(256), kernel_size=3, strides=2, padding="sam
            layers.BatchNormalization(),
            layers.LeakyReLU(negative_slope=0.2),
            layers.Conv2DTranspose(int(128), kernel_size=3, strides=2, padding="sam
            layers.BatchNormalization(),
            layers.LeakyReLU(negative_slope=0.2),
            layers.Conv2DTranspose(int(64), kernel_size=3, strides=2, padding="same
            layers.BatchNormalization(),
            layers.LeakyReLU(negative_slope=0.2),
            layers.Conv2DTranspose(int(32), kernel_size=3, strides=1, padding="same
            layers.BatchNormalization(),
            layers.LeakyReLU(negative_slope=0.2),
            layers.Conv2D(3, kernel_size=5, padding="same", activation="tanh"),
        ],
        name="generator",
    )
    my_generator3.summary()
```

Model: "generator"

Layer (type)	Output Shape	Param #
dense_7 (Dense)	(None, 131072)	13,238,272
reshape_3 (Reshape)	(None, 16, 16, 512)	0
conv2d_transpose_10 (Conv2DTranspose)	(None, 32, 32, 512)	2,359,808
batch_normalization_14 (BatchNormalization)	(None, 32, 32, 512)	2,048
leaky_re_lu_26 (LeakyReLU)	(None, 32, 32, 512)	0
conv2d_transpose_11 (Conv2DTranspose)	(None, 64, 64, 256)	1,179,904
batch_normalization_15 (BatchNormalization)	(None, 64, 64, 256)	1,024
leaky_re_lu_27 (LeakyReLU)	(None, 64, 64, 256)	0
conv2d_transpose_12 (Conv2DTranspose)	(None, 128, 128, 128)	295,040
batch_normalization_16 (BatchNormalization)	(None, 128, 128, 128)	512
leaky_re_lu_28 (LeakyReLU)	(None, 128, 128, 128)	0
conv2d_transpose_13 (Conv2DTranspose)	(None, 256, 256, 64)	73,792
batch_normalization_17 (BatchNormalization)	(None, 256, 256, 64)	256
leaky_re_lu_29 (LeakyReLU)	(None, 256, 256, 64)	0
conv2d_transpose_14 (Conv2DTranspose)	(None, 256, 256, 32)	18,464
batch_normalization_18 (BatchNormalization)	(None, 256, 256, 32)	128
leaky_re_lu_30 (LeakyReLU)	(None, 256, 256, 32)	0
conv2d_19 (Conv2D)	(None, 256, 256, 3)	2,403

Total params: 17,171,651 (65.50 MB)

Trainable params: 17,169,667 (65.50 MB)

Non-trainable params: 1,984 (7.75 KB)

Training

Loss functions are defined below. The models use binary cross entropy.

Sources:

https://www.kaggle.com/code/thuylinh225/generate-monet-images-using-dcgan

https://keras.io/examples/generative/dcgan_overriding_train_step/

In [23]:
```
# create loss function for the generator
with strategy.scope():
    def generator_loss(fake_output):
        cross_entropy = tf.keras.losses.BinaryCrossentropy(reduction=tf.keras.losse
        return cross_entropy(tf.ones_like(fake_output), fake_output)

    # create loss function for the discriminator
    def discriminator_loss(real_output, fake_output):
        cross_entropy = tf.keras.losses.BinaryCrossentropy(reduction=tf.keras.losse
        real_loss = cross_entropy(tf.ones_like(real_output), real_output)
        fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
        total_loss = real_loss + fake_loss
        return total_loss
```

Callbacks

Both, generator and discriminator, use Adam optimizer with learning rate of 0.0002.

Source: https://www.kaggle.com/code/thuylinh225/generate-monet-images-using-dcgan

In [24]:
```
# Create two separate optimizers for the generator and discriminator
with strategy.scope():
    generator_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5
    discriminator_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1
```

In [25]:
```
# Set the hyperparameters to be used for training
EPOCHS = 1000
BATCH_SIZE = BATCH_SIZE
noise_dim = latent_dim
print('Batch size:', BATCH_SIZE)
print('Latent vector size (same as noise vector size):', noise_dim)
```

```
Batch size: 32
Latent vector size (same as noise vector size): 100
```

The code for the model and training is below.

- The generator creates images from noise vector.
- The discriminator evaluates the images created by the generator along with real Monet paintings.
- Losses and gradients are calculated and weights are updated.

In [26]:
```
class DCGAN_model:
    def __init__(self, noise_dim, EPOCHS, BATCH_SIZE, generator, discriminator, dat
```

```
        self.noise_dim = noise_dim
        self.EPOCHS = EPOCHS
        self.BATCH_SIZE = BATCH_SIZE
        self.generator = generator
        self.discriminator = discriminator
        self.dataset = dataset

    @tf.function
    def train(self, images):

    # Create random noise vector
        noise = tf.random.normal([images.shape[0], noise_dim])

        with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:

        # generate images use random noise vector
            generated_images = self.generator(noise, training=True)

            # use discriminator to evaluate the real and fake images
            real_output = self.discriminator(images, training=True)
            fake_output = self.discriminator(generated_images, training=True)

            # compute generator loss and discriminator loss
            gen_loss = generator_loss(fake_output)
            disc_loss = discriminator_loss(real_output, fake_output)

            # Compute gradients
            gradients_of_generator = gen_tape.gradient(gen_loss, self.generator.tra
            gradients_of_discriminator = disc_tape.gradient(disc_loss, self.discrim

            # Update optimizers
            generator_optimizer.apply_gradients(zip(gradients_of_generator, self.ge
            discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,

        return (gen_loss + disc_loss) * 0.5

    @tf.function
    def distributed_train(self, images):
        per_replica_losses = strategy.run(self.train, args=(images,))
        return strategy.reduce(tf.distribute.ReduceOp.MEAN, per_replica_losses, axi

    def generate_images(self):
        noise = tf.random.normal([self.BATCH_SIZE, self.noise_dim])
        predictions = self.generator.predict(noise)
        return predictions

    def generate_and_plot_images(self):
        image = self.generate_images()
        gen_imgs = 0.5 * image + 0.5
        fig = plt.figure(figsize=(10, 10))
        for i in range(25):
            plt.subplot(5, 5, i+1)
            plt.imshow(gen_imgs[i, :, :, :])
            plt.axis('off')
        plt.show()
```

```
    def train_loop(self):
        e_ls = []
        mean_ls = []
        for epoch in range(self.EPOCHS):
            start = time.time()

            total_loss = 0.0
            num_batches = 0

            for image_batch in self.dataset:
                loss = self.distributed_train(image_batch['image'])
                total_loss += tf.reduce_mean(loss)
                num_batches += 1
            mean_loss = total_loss / num_batches

            if (epoch+1) % 200 == 0:
                print ('Time for epoch {} is {} sec, mean loss is {}'.format(epoch
                self.generate_and_plot_images()

                e_ls.append(epoch+1)
                mean_ls.append(mean_loss)
        print("\nMean Loss for every 200 epochs: \n")
        table = pd.DataFrame({"Epoch": e_ls, "Mean Loss": np.array(mean_ls)})
        return table
```

Training

The training is executed in the cell below.

Source: https://www.kaggle.com/code/thuylinh225/generate-monet-images-using-dcgan

In [ ]:
```
# train, visualize and print out the result for DCGAN model
gan1 = DCGAN_model(noise_dim, EPOCHS, BATCH_SIZE, my_generator2, my_discriminator2,
res1 = gan1.train_loop()
res1
```

Results and Analysis (35 pts)

Run hyperparameter tuning, try different architectures for comparison, apply techniques to improve training or performance, and discuss what helped.

Includes results with tables and figures. There is an analysis of why or why not something worked well, troubleshooting, and a hyperparameter optimization procedure summary.

The first generator/discriminator did not produce reasonable looking Monet paintings. I suspect that either they just did not have enough layers or the sigmoid activation in the generator did not work. The 2nd version of generator and discriminator have one additional layer (total of four) and batch normalization was added, and the generator had tanh activation. These were used as a new baseline. They produced reasonably good result of 100 with ~45min of GPU time.

Increasing the number of filters in the CNN layers to minimum number of 128 (some layers up from 32 and 64) improved the score a little, to ~95, but increased the GPU training time to 1h and 45min.

Leaky ReLU with the negative slope of 0.1 was also tried but the results were worse (value of 0.2 is widely used in GAN models).

In [28]:
```
pd.DataFrame(np.array([['v7','v8','v9','v10','v11','v12'],
              ['Baseline', 'Latent dimension 200', 'Leaky ReLU with 0.1 slope', 'Ad
               'Up to 2000 epochs', 'Min. 128 filters in CNN'],
               [100.9, 124.3, 125, 101.9, 116.7, 94.5]]).transpose(),
               columns=['Version', 'Description', 'Score'])
```

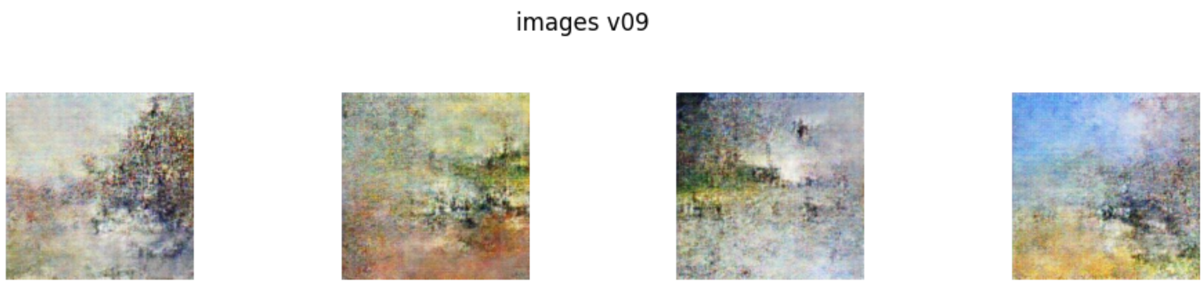
Out[28]:

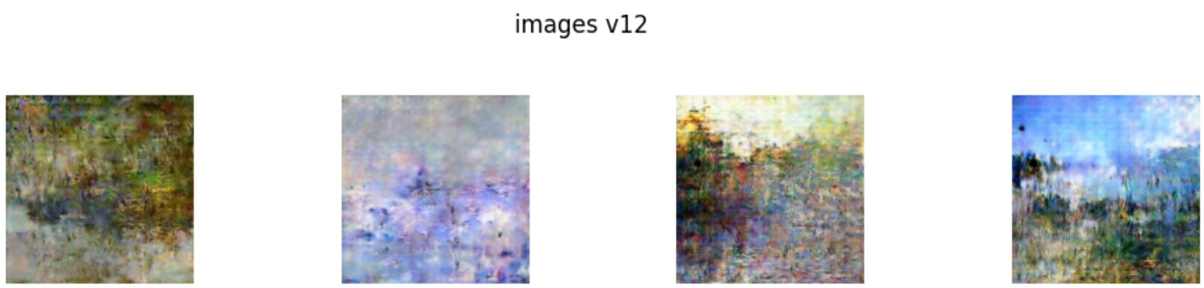
	Version	Description	Score
0	v7	Baseline	100.9
1	v8	Latent dimension 200	124.3
2	v9	Leaky ReLU with 0.1 slope	125
3	v10	Additional hidden CNN layer in the generator	101.9
4	v11	Up to 2000 epochs	116.7
5	v12	Min. 128 filters in CNN	94.5

Sample 'Monets' from some of the models are shown below.

In [19]:
```
from PIL import Image
import zipfile
import random
import io

zips = ["images v07.zip", "images v09.zip", "images v12.zip"]
ids = random.sample(range(7000), 4)
files_to_extract = ['image_{}.jpg'.format(s) for s in ids]
for zip in zips:
    fig = plt.figure(figsize=(12, 6))
    i = 0
    with zipfile.ZipFile(os.path.join(datapath, zip), 'r') as zipf:
        for file_name in files_to_extract:
            if file_name in zipf.namelist():
                with zipf.open(file_name) as file:
                    # Read the image into a Pillow Image object
                    image = Image.open(io.BytesIO(file.read()))
                    # Show the image
                    ax = plt.subplot(3, 4, i + 1)
                    i += 1
                    plt.imshow(image)
                    plt.axis('off')
    plt.suptitle(zip[0:-4])
    plt.show()
```





Submission

The code for file submission in Kaggle is below.

In [27]:
```
from PIL import Image
import shutil
from keras.saving import load_model
import zipfile
import io
```

In [ ]:
```
no_of_pics = 7050
with strategy.scope():
    with zipfile.ZipFile("images.zip", "w") as zipf:
        p = 1
        for i in range(int(np.ceil(no_of_pics/BATCH_SIZE))):
            print(i, end='\r')
            noise = tf.random.normal([BATCH_SIZE, noise_dim])
            img_batch = my_generator2.predict(noise)
            for img in img_batch:
                image = ((img + 1) * 127.5).astype('uint8')
                # print(image.shape)
                # print(np.min(image.ravel()), np.max(image.ravel()))
                image = Image.fromarray(image)
                # Save the image to an in-memory buffer
                buffer = io.BytesIO()
                image.save(buffer, format="JPEG")
```

```
                buffer.seek(0)  # Move to the beginning of the buffer
                # Write the image to the ZIP file
                zipf.writestr(f"image_{p}.jpg", buffer.getvalue())
                p += 1
        plt.imshow(image)
```

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

for dirname, _, filenames in os.walk('/kaggle/working'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
        break
for dirname, _, filenames in os.walk('/kaggle/output'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
        break

file_size_bytes = os.path.getsize('/kaggle/working/images.zip')
print(f"File size (bytes): {file_size_bytes}")
# Convert to other units (optional)
file_size_kb = file_size_bytes / 1024
file_size_mb = file_size_kb / 1024
print(f"File size (KB): {file_size_kb:.2f}")
print(f"File size (MB): {file_size_mb:.2f}")
```

Conclusion (15 pts)

Discuss and interpret results as well as learnings and takeaways. What did and did not help improve the performance of your models? What improvements could you try in the future?

Looks like just adding more parameters into the model improved the score the best.

Getting the TPU working with the code was challenging, and in the end I ended up using GPU for training.

Could try to improve the score by changing parameters of the worse than baseline models to opposite direction (i.e., increase leaky ReLU slope, decrease latent dimension, etc.) and try to combine all the improvements into one model. Additionally, monitoring the loss while training could help find the optimal epoch count. Adjusting the learning rates could also improve the model.

From architecture perspective, using CycleGAN type of model could be beneficial.

Versions

Version 2: Baseline, 100 latent dimension, 1000 epochs.

Version 3: Same, problem with output files.

Version 4: Same, problem with output files. Save directly into zip-file.

Version 5: Same, problem with output files. Don't save the generator (only one file saved). Save as .jpg, not .jpeg.

Version 5: Same, problem with output files. Accidentaly overwrote images.zip.

Version 6: Success, score ~100.

Version 7: Increase latent dimension from 100 to 200.

Version 9: Adjust leaky ReLU to 0.1.

Version 10: Add another layer into generator ("my_generator3).

Version 11: Increase epochs from 1000 to 2000.

Version 12: Add more filters, minimum of 128 per layer.

Sources

https://www.kaggle.com/code/amyjang/monet-cyclegan-tutorial

https://keras.io/examples/generative/dcgan_overriding_train_step

https://www.kaggle.com/code/thuylinh225/generate-monet-images-using-dcgan