DTSA 5511 - Introduction to Deep Learning

University of Colorado, Boulder

Master of Science in Data Science

MonetStyle Paintings |Kaggle

1 Introduction

Description

In this challenge, we are given a dataset that contains four directories: monet_tfrec, photo_tfrec, monet_jpg, and photo_jpg. The monet_tfrec and monet_jpg directories contain the same painting images, and the photo_tfrec and photo_jpg directories contain the same photos.(jpeg and TensorFlow records format). We are asked to add Monet-style to these images and submit your generated jpeg images as a zip file. Other photos outside of this dataset can be transformed but keep your submission file limited to 10,000 images.

Evaluation

MiFID Submissions are evaluated on MiFID (Memorization-informed Fréchet Inception Distance), which is a modification from Fréchet Inception Distance (FID). The smaller MiFID is, the better your generated images are.

Data Structure

Files

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

Submission

Your kernel's output must be called images.zip and contain 7,000-10,000 images sized 256x256.

2 Exploratory Data Analysis(EDA)

2.1 Data Loading and Libraries Import

```
import tensorflow as tf
from tensorflow.keras import layers, Model, losses, optimizers,
    Input
import tensorflow_addons as tfa
```

```
MONET_FILENAMES = tf.io.gfile.glob('/kaggle/input/gan-getting-
    started/monet_tfrec/*.tfrec')
PHOTO_FILENAMES = tf.io.gfile.glob('/kaggle/input/gan-getting-
    started/photo_tfrec/*.tfrec')
```

2.2 Setting Up TPUs or GPUs

```
# Set up TPU or GPU/CPU
try:
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver()
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.experimental.TPUStrategy(tpu)
except ValueError:
    strategy = tf.distribute.get_strategy()

print('Number of replicas:', strategy.num_replicas_in_sync)

AUTOTUNE = tf.data.experimental.AUTOTUNE
```

**********extra helper loading functions

```
IMAGE_SIZE = [256, 256]

def decode_image(image):
    image = tf.image.decode_jpeg(image, channels=3)
    image = (tf.cast(image, tf.float32) / 127.5) - 1
    image = tf.reshape(image, [*IMAGE_SIZE, 3])
    return image

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

```
def load_dataset(filenames):
    ignore_order = tf.data.Options()
    ignore_order.experimental_deterministic = False
    dataset = tf.data.TFRecordDataset(filenames)
    dataset = dataset.with_options(ignore_order)
    dataset = dataset.map(read_tfrecord, num_parallel_calls=
       AUTOTUNE)
    return dataset.batch(1)
```

3 Introduction To CycleGANs

Generative Adversarial Networks (GANs) have revolutionized the field of image generation, enabling the creation of highly realistic and artistic images. Among these, Cycle-Consistent Generative Adversarial Networks, or CycleGANs, have emerged as a powerful tool for image-to-image translation tasks without needing paired examples. This capability makes CycleGANs particularly suited for artistic style transfer, such as generating images in the style of famous painters like Claude Monet.

CycleGANs are a type of GAN that enables the transformation of images from one domain to another (e.g., from a photograph to a Monet painting) without direct paired examples. This is achieved through a dual-GAN setup where two generators and two discriminators learn to translate images back and forth between two domains. The key innovation of CycleGAN is the introduction of a cycle consistency loss. This loss ensures that an image can be converted from its original domain to the target domain and back again to the original domain, preserving key characteristics and content of the original image throughout the process.

Unlike traditional methods that require exact before-and-after pairs, CycleGANs can learn to translate between domains with unpaired datasets. This is perfect for art style transfer, where finding exact pairs of real-world scenes and their Monet-style counterparts is impractical. CycleGANs are designed to maintain the integrity of the original image content while altering its style. This ensures that the generated Monet-style images retain the structure and essence of the input photographs, with the distinctive stylistic touches of Monet's painting technique.

4 Defining Generator and Discriminator Models

Generators are neural networks that aim to transform an input image from one domain (e.g., a real photograph) into an output image that looks as though it belongs to another domain (e.g., a Monet painting). The goal of the generator is to create images that are indistinguishable from real images in the target domain.

Discriminators are neural networks that classify images as real (belonging to the dataset of the target domain) or fake (generated by the generator). The discriminator's goal is to guide the generator towards producing more realistic and stylistically accurate images by providing feedback on the authenticity of the generated images.

Generator's Model

- **Input Layer** our generator starts with an input layer that accepts images of size 256x256 pixels with 3 color channels (RGB).
- **Downsampling Layers** model then applies convolutional layers with strides to reduce the spatial dimensions of the image. This downsampling process helps the network to encode the input image into a dense representation, capturing its essential features. Instance normalization is applied after some convolutional layers to stabilize the learning process and help in style transfer.
- **Upsampling Layers** After encoding, the network uses transposed convolutional layers to gradually increase the spatial dimensions of the representation, aiming to reconstruct the image in the target domain's style. Instance normalization is again used here to ensure the style consistency of the generated image.
- **Output Layer** The final layer uses a convolution with a tanh activation function to produce the output image. The tanh function ensures that the output pixel values are normalized between -1 and 1, matching the common preprocessing of images.

Discriminator's Model

- **Input Layer** Similar to the generator, the discriminator accepts 256x256 pixel RGB images as input.
- **Downsampling Layers**It uses convolutional layers with strides for downsampling, allowing the network to focus on the features necessary for distinguishing between real and fake images. LeakyReLU activation functions are used to add nonlinearity, enabling the network to learn more complex patterns.
- **Output Layer** The final convolutional layer outputs a single value, representing the discriminator's assessment of the image's authenticity. A value closer to 1 indicates a real image, while a value closer to 0 indicates a fake image.

```
def make_generator_model():
    model = tf.keras.Sequential([
        # Input layer
        Input(shape=(256, 256, 3)),
        # Downsampling
        layers.Conv2D(64, (4, 4), strides=(2, 2), padding='same
            '),
        layers.LeakyReLU(),
        layers.Conv2D(128, (4, 4), strides=(2, 2), padding='
            same'),
        tfa.layers.InstanceNormalization(),
        layers.LeakyReLU(),
        # Upsampling
        layers.Conv2DTranspose(128, (4, 4), strides=(2, 2),
            padding='same'),
        tfa.layers.InstanceNormalization(),
        layers.ReLU(),
        layers.Conv2DTranspose(64, (4, 4), strides=(2, 2),
            padding='same'),
        tfa.layers.InstanceNormalization(),
        layers.ReLU(),
        # Output layer
        layers.Conv2D(3, (7, 7), padding='same', activation='
            tanh')
    ])
    return model

def make_discriminator_model():
    model = tf.keras.Sequential([
        # Input layer
        Input(shape=(256, 256, 3)),
        # Downsampling
        layers.Conv2D(64, (4, 4), strides=(2, 2), padding='same
            '),
        layers.LeakyReLU(),
        layers.Conv2D(128, (4, 4), strides=(2, 2), padding='
            same'),
        layers.LeakyReLU(),
        # Output layer
        layers.Conv2D(1, (4, 4), strides=(1, 1), padding='same'
            )
    ])
    return model
```

5 Defining Generator Discriminator Loss and The Training Step

Discriminator Loss
The discriminator's goal is to correctly classify real images as real and generated (fake) images as fake. The discriminator loss consists of two parts:

- **Real Loss** This is calculated by comparing the discriminator's predictions on real images (real_output) to a tensor of ones, representing the "real" label. The closer the discriminator's predictions are to 1 (indicating "real"), the lower this part of the loss will be.
- **Fake Loss** This is calculated by comparing the discriminator's predictions on generated images (fake_output) to a tensor of zeros, representing the "fake" label. The closer the predictions are to 0, the lower the fake loss.

The total discriminator loss is the sum of the real loss and the fake loss. By minimizing this total loss, the discriminator learns to better distinguish between real and generated images.

Generator Loss
The generator's goal is to create images that the discriminator will classify as real. The generator loss is calculated by comparing the discriminator's predictions on the generated images (fake_output) to a tensor of ones, indicating the generator is aiming for its outputs to be classified as "real".
Minimizing the generator loss encourages the generator to produce images that are indistinguishable from real images, according to the discriminator's judgement.

Training Step
The training step involves updating both the generator and discriminator models based on their respective loss functions. Here's how it works:

- **Forward Pass**
  - Generate images (generated_y) from real images (real_x) using the generator.
  - Pass both real images (real_y) and generated images (generated_y) through the discriminator to obtain predictions on real and generated images, respectively.
- **Losses Calculations**

  - Calculate the generator loss using the discriminator's predictions on the generated images.
  - Calculate the discriminator loss using its predictions on both real and generated images.

- **Backward Pass Update**

  - Calculate gradients of the losses with respect to the models' parameters using gradient tape.
  - Apply these gradients to the models' parameters using their respective optimizers, updating the generator and discriminator in the direction that will reduce their losses.

This process repeats for each training step, iteratively improving the generator's ability to produce realistic images and the discriminator's ability to classify images correctly. The use of @tf.function decorator on the train_step function compiles it into a high-performance callable, which can significantly improve training speed by optimizing the execution graph.

```
with strategy.scope():
    monet_generator = make_generator_model() # Transforms
       photos to Monet-esque paintings
    photo_generator = make_generator_model() # Transforms Monet
        paintings to be more like photos (not used in this
       example)

    monet_discriminator = make_discriminator_model() #
       Differentiates real Monet paintings and generated Monet
       paintings
    photo_discriminator = make_discriminator_model() #
       Differentiates real photos and generated photos (not
       used in this example)

    generator_optimizer = optimizers.Adam(2e-4, beta_1=0.5)
    discriminator_optimizer = optimizers.Adam(2e-4, beta_1=0.5)

    loss_obj = losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    real_loss = loss_obj(tf.ones_like(real_output), real_output
       )
    fake_loss = loss_obj(tf.zeros_like(fake_output),
       fake_output)
```

```
    total_disc_loss = real_loss + fake_loss
    return total_disc_loss

def generator_loss(fake_output):
    return loss_obj(tf.ones_like(fake_output), fake_output)

@tf.function
def train_step(real_x, real_y):
    # real_x is photos, real_y is Monet paintings
    with tf.GradientTape() as gen_tape, tf.GradientTape() as
       disc_tape:
        generated_y = monet_generator(real_x, training=True)

        real_y_output = monet_discriminator(real_y, training=
           True)
        generated_y_output = monet_discriminator(generated_y,
           training=True)

        gen_loss = generator_loss(generated_y_output)
        disc_loss = discriminator_loss(real_y_output,
           generated_y_output)

    gradients_of_generator = gen_tape.gradient(gen_loss,
       monet_generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
       monet_discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(
       gradients_of_generator, monet_generator.
       trainable_variables))
    discriminator_optimizer.apply_gradients(zip(
       gradients_of_discriminator, monet_discriminator.
       trainable_variables))
```

6 Generating MonetStyle Images and Saving In The Kernel's Output

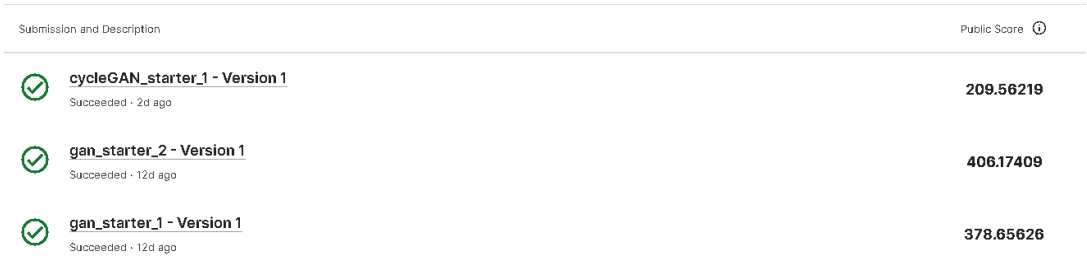
Here, we set the epochs and apply the train_step functions to our images and save the generated images in the kernel's output directory for submission.

```
# Load datasets
with strategy.scope():
    monet_ds = load_dataset(MONET_FILENAMES)
    photo_ds = load_dataset(PHOTO_FILENAMES)

# Training loop
EPOCHS = 25

for epoch in range(EPOCHS):
    for image_x, image_y in tf.data.Dataset.zip((photo_ds,
       monet_ds)):
        train_step(image_x, image_y)
    print(f'Epoch {epoch+1} completed')

# Save the generated images or further process as needed


import os
import PIL

# Create a directory to save generated images
output_dir = '/kaggle/working/images'
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# Generate and save images
def generate_and_save_images(model, test_input, save_path,
   start_idx=1):
    # Note: `test_input` is the dataset from which you want to
       generate images
    # `start_idx` is the starting index for naming the image
       files
    i = start_idx
    for img in test_input.take(10000): # Adjust based on how
       many images you need
        prediction = model(img, training=False)[0].numpy()
        prediction = (prediction * 127.5 + 127.5).astype(np.
```

```
            uint8) # Convert from [-1,1] to [0,255]
        im = PIL.Image.fromarray(prediction)
        im.save(os.path.join(save_path, f'{i}.jpg'))
        i += 1

# Assuming photo_ds is your photo dataset
generate_and_save_images(monet_generator, photo_ds, output_dir)

import shutil

shutil.make_archive('/kaggle/working/images', 'zip', output_dir
    )
```

7 Conclusion and Future Work

Below is my different models performance(MiFID scores):



7.1 Future Work

If One decides to improve upon the performance of the CycleGANs, they should consider:

- **Incorporating More Complex Network Architectures**Explore deeper and more complex generator and discriminator architectures, with careful attention to the balance between model capacity and overfitting risk.
- **Enhancing the Loss Function** Implement additional loss components, such as perceptual loss or feature matching loss, to guide the model towards generating more realistic and aesthetically pleasing images.
- **Experimenting with Training Techniques**Utilize techniques like progressive growing of GANs, where the model starts training with low-resolution images and gradually increases resolution as training progresses.

- **Optimizing the Training Pipeline** Further optimize the data loading and preprocessing steps to reduce bottlenecks and improve the efficiency of model training, especially when scaling to larger datasets.

8 References

[1] Amy Jang, Ana Sofia Uzsoy, Phil Culliton. (2020). I'm Something of a Painter Myself. Kaggle. https://kaggle.com/competitions/gan-getting-started
[2] https://www.kaggle.com/code/amyjang/monet-cyclegan-tutorial
[3] https://www.kaggle.com/code/iosifcovasan/photo-to-monet-using-cyclegan