基于 CycleGAN 的图像莫奈风格迁移

目录

1. 任务介绍

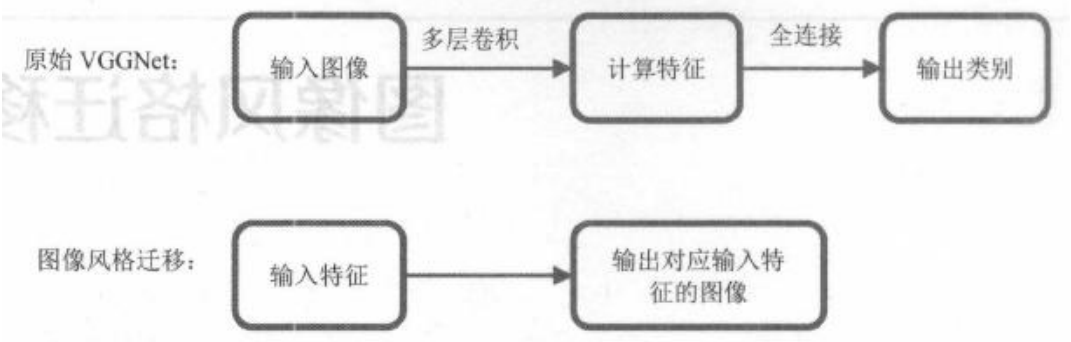
1.1 风格迁移介绍

所谓图像风格迁移，是指利用算法学习著名画作的风格，然后再把这种风格应用到另外一张图片上的技术。著名的国像处理应用 Prisma 是利用风格迁移技术，将普通用户的照片自动变换为具有艺术家的风格的图片。这篇文章会介绍这项技术背后的原理，此外，还会使用 TensorFlow 实现一个快速风格迁移的应用。

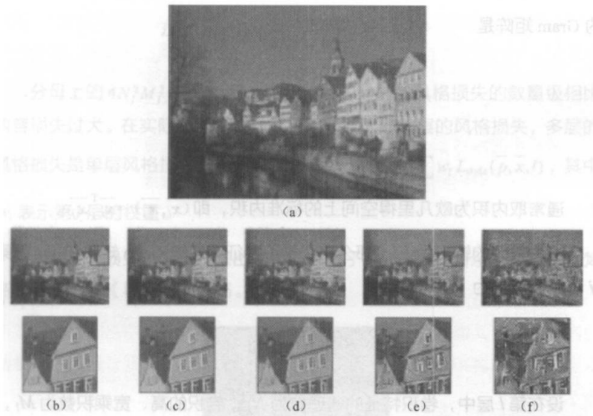
本次实验利用 CycleGAN 架构为照片添加莫奈风格。 在程序中，我们将使用 TFRecord 数据集，导入相关包并将加速器更改为 TPU。

以 ImageNet 图像识别模型 VGGNet 为例： 事实上，可以这样理解 VGGNet 的结构：前面的卷积层是从图像中提取“特征”，而后面的全连接层把图片的“特征”转换为类别概率。其中，VGGNet 中的浅层（如 conv1_1，conv1_2），提取的特征往往是比较简单的（如检测点、线、亮度），VGGNet 中的深层（如 conv5_1、conv5_2），提取的特征往往是比较复杂（如有无人脸或某种特定物体）。 VGGNet 的本意是输入图像，提取特征，并输出图像类别。图像风格迁移正好与其相反，输入特征，输出对应这种特征的图片。

VGGNet 的本意是输入图像，提取特征，并输出图像类别。图像风格迁移正好与其相反，输入特征，输出对应这种特征的图片，如图所示。



具体来说，风格迁移使用卷积层的中间特征还原出对应这种特征的原始图像。如图所示，先选取衣服原始图像，经过 VGGNet 计算后得到各种卷积层特征。接下来，根据这些卷积层的特征，还原出对应这种特征的原始图像。图像 b、c 、d 、e 、f 分别为使用卷积层 conv1_2、conv2_2 、conv3 _2 、conv4_2 、conv5_2 的还原图像。可以发现：浅层的还原效果往往比较好，卷积特征基本保留了所高原始图像中形状、位置、颜色、纹理等信息； 深层对应的还原图像丢失了部分颜色和纹理信息，但大体保留原始图像中物体的形状和位置。



具体的数学原理推导如下：

还原图像的方法是梯度下降法。设原始图像为$\vec{p}$，期望还原的图像为$\vec{x}$（即自动生成的图像）。使用的卷积是第l层，原始图像$\vec{p}$在第l层的特征为P_{ij}^l。i表示卷积的第i个通道，j表示卷积的第j个位置。通常卷积的特征是三维的，三维坐标分别对应（高、宽、通道）。此处不考虑具体的高和宽，只考虑位置j，相当于把卷积"压扁"了。比如一个10x10x32的卷积特征，对应$1 \leqslant i \leqslant 32, 1 \leqslant j \leqslant 100$。对于生成图像$\vec{x}$，同样定义它在$l$层的卷积特征为$F_{ij}^l$。

有了上面这些符号后，可以写出"内容损失"（Content Loss）。内容损失$L_{content}(\vec{p}, \vec{x}, l)$的定义是：
$L_{content}(\vec{p}, \vec{x}, l) = \frac{1}{2}\sum_{i,j}(F_{ij}^l - P_{ij}^l)^2$
$L_{content}(\vec{p}, \vec{x}, l)$描述了原始图像$\vec{p}$和生成图像$\vec{x}$在内容上的"差异"。内容损失越小，说明他们的内容越接近；内容损失越大，说明他们的内容差距也越大。先使用原始图像$\vec{p}$计算出它的卷积特征P_{ij}^l，同时随机初始化$\vec{x}$。接着以内容损失$L_{content}(\vec{p}, \vec{x}, l)$为优化目标，通过梯度下降法逐步改变$\vec{x}$，进过一定的步数后，得到的$\vec{x}$是希望的还原图像了。在这个过程中，内容损失$L_{content}(\vec{p}, \vec{x}, l)$应该是越来越小了。

除了还原图像原本的"内容"之外，另一方面，还希望还原图像的"风格"。那么图像的"风格"应该用什么来表示呢？一种方法是使用图像的卷积层特征的Gram矩阵。

Gram矩阵是关于一组向量的内积的对称矩阵，例如，向量组$\vec{x_1}, \vec{x_2}, ..., \vec{x_n}$的Gram矩阵是：

$$\begin{bmatrix} (\vec{x_1}, \vec{x_1}) & (\vec{x_1}, \vec{x_2}) & ... & (\vec{x_1}, \vec{x_n}) \\ (\vec{x_2}, \vec{x_1}) & (\vec{x_2}, \vec{x_2}) & ... & (\vec{x_2}, \vec{x_n}) \\ ... & ... & ... & ... \\ (\vec{x_n}, \vec{x_1}) & (\vec{x_n}, \vec{x_2}) & ... & (\vec{x_n}, \vec{x_n}) \end{bmatrix}$$

通常取内积为欧几里得空间上的标准内积，即$(\vec{x_i}, \vec{x_j}) = \vec{x_i}^T\vec{x_j}$。

设卷积层的输出为F_{ij}^l，那么这个卷积特征对应的Gram矩阵的第i行第j个元素定义为
$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l$
设在第l层中，卷积特征的通道数为N_l，卷积的高、宽成绩数为M_l，那么F_{ij}^l满足$1 \leqslant i \leqslant N_l, 1 \leqslant i \leqslant M_l$。G实际是向量组$F_1^l, F_2^l, \cdots \cdot F_i^l, \cdots F_{N_l}^l$的Gram矩阵，其中$F_i^l = (F_{i1}^l, F_{i2}^l, \cdots F_{ij}^l, \cdots F_{iM_l}^l)$

此处数学符号特较多，因此再举一个例子来加深读者对此Gram矩阵的理解。假设某一层输出的卷积特征为10x10x32，即它是一个宽、高均为10，通道数为32的张量。F_1^l表示第一个通道的特征，它是一个100维的向量。F_2^l表示第二个通道的特征，他同样是一个100维的向量，它对应的Gram矩阵G是

$$\begin{bmatrix} (F_1^l)^T F_1^l & (F_1^l)^T F_2^l & ... & (F_1^l)^T F_{32}^l \\ (F_2^l)^T F_1^l & (F_2^l)^T F_2^l & ... & (F_2^l)^T F_{32}^l \\ ... & ... & ... & ... \\ (F_{32}^l)^T F_1^l & (F_{32}^l)^T F_2^l & ... & (F_{32}^l)^T F_{32}^l \end{bmatrix}$$

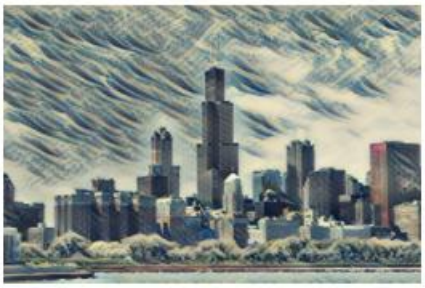
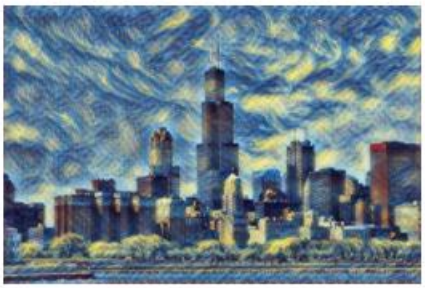
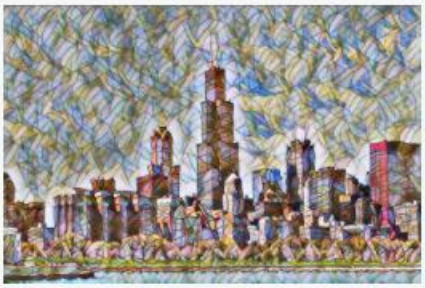
Gram矩阵可以在一定程度上反映原始图像的"风格"。仿照"内容损失"，还可以定义一个"风格损失"（Style Loss）。设原始图像为$\vec{a}$，要还原的风格图像为$\vec{x}$，先计算出原始图像某一层卷积的Gram矩阵为A^l，要还原的图像$\vec{x}$经过同样的计算得到对应卷积层的Gram矩阵是G^l，风格损失定义为

$$L_{style}(\vec{p}, \vec{x}, l) = \frac{1}{4N_l^2 M_l^2}\sum_{i,j}(A_{ij}^l - G_{ij}^l)^2$$

分母上的$4N_l^2 M_l^2$是一个归一化项，目的是防止风格损失的数量级相比内容损失过大。在实际应用中，常利用多层而非一层的风格损失，多层的风格损失是单层风格损失的加权累加，即$L_{style}(\vec{p}, \vec{x}) = \sum_l w_l Lstyle(\vec{p}, \vec{x}, l)$，其中$w_l$表示第$l$层的权重。

该实验建立在另一个开源项目 TensorFlow Slim 的基础上，TensorFlow Slim 是基于 TensorFlow 的一个开源图像分类库，它定义了常用的 ImageNet 模型。而其中的 VGG16 模型正式在定义损失网络时要用到的。

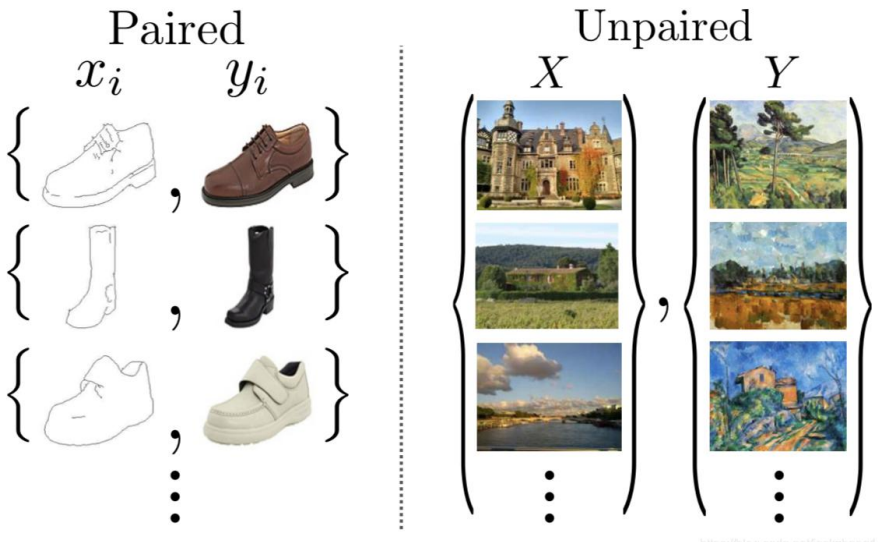
七个预训练模型及其风格化图片效果见表：

模型名称	来源介绍	原始风格图像	风格化后的图像
wave	葛饰北斋的著名画作《神奈川冲浪里》		
cubist	现代艺术图片		
denoised_starry	梵高的著名画作《星空》		
mosaic	镶嵌玻璃装饰图		
scream	爱德华·蒙克著名画作《呐喊》		

除此之外，训练时最先关心的应该是损失下降的情况。损失主要由风格损失、内容损失两项构成 center_loss 和 style_loss 分别对应了内容损失和风格损失，中间的 regularizer_loss 可以暂时不用理会。最理想的情况是 content_loss 和 style_loss 随着训练地不断下降。在训练的初期可能会出现只有 style_loss 下降而 content_loss 上升的情况，不过这是暂时的，最后两个损失都会出现较为稳定的下降。 当训练新的“风格”时，再时可能还会需要调整配置文件中的 content_ weight 和 style_ weight 。当 content_weight 过大时，观察到的 generated 图像会非常接近原始的 origin 图像。而 style_ weight 过大时，会导致图像过于接近原始的风格图像。在训练时，需要合理调整 style_weight 和 content_weight 的比重。

1.2 CycleGAN 介绍

在 CycleGAN 出现之前，pix2pix 网络在处理 image-image translation 问题上比较 state-of-the-art。但是 pix2pix 需要利用成对（pair）的数据进行模型训练，如下图所示：



成对的数据在自然界中是非常稀有的，因此 pix2pix 对数据的要求很高，一般而言不具备通用性。CycleGAN 的出现可以解决这一问题，也就是说，CycleGAN 可利用 unpaired 数据，在 source domain X XX 和 target domain Y YY 之间建立一个映射：G : X → Y G:X ->Y G:X→Y，从而使得源域 X XX 的图像转化为与目标域 Y YY 分布相似的图像。也就是说，图像 G (X) G(X)G(X) 无法被分辨出是从 Y YY 中采样的还是由 G GG 生成的。

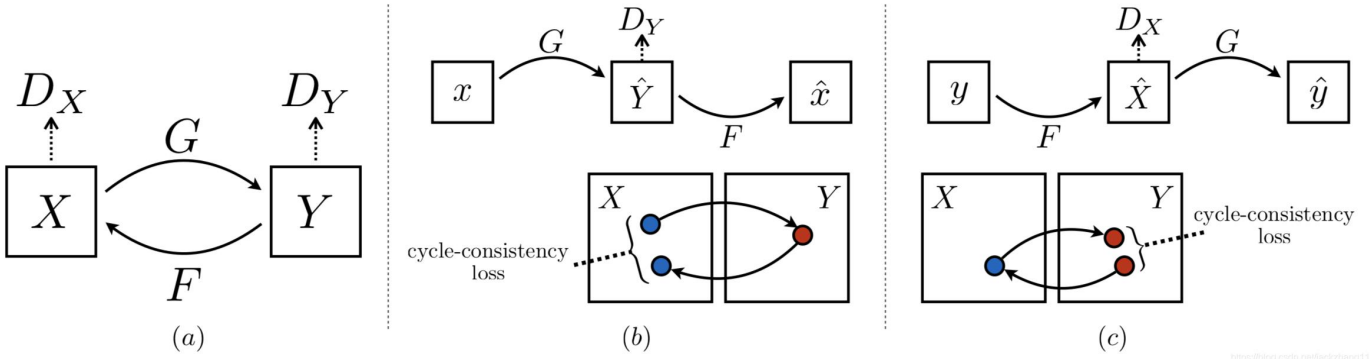
通过这个操作，我们就可以将自然图像转化为具有莫奈风格的图像，可以将斑马转变成普通的骏马，将冬天转变为夏天等等。也就是说，CycleGAN 可以实现图像的风格迁移，更广义地来说，实现了图像间的翻译。

但是如果仅有 G : X → Y G:X->YG:X→Y，显然是不能完成这一任务的。因为这个映射只能确保 G (X) G(X)G(X) 神似目标域 Y YY 中的样本，并不能确保它与生成前的图像是对应的。举个不太恰当例子，现在的源域 X XX 表示“中文”，目标域 Y YY 表示“英文”。从 X XX 中采样一个样本「你好吗」，经过 G : X → Y G:X->YG:X→Y 得到的 G (X) G(X)G(X) 理应是「How are you」。但是，由于该映射只是希望 G (X) G(X)G(X) 拥有目标域“英文”的特征，所以它可以是任意一句英文，如「I’m fine」。这样就失去了 translation 的意义。同时，GAN 网络为了保证最小化 Loss，宁愿所有样本的都生成同一个输出，也不会冒险去生成多样的结果，这就造成了 Mode Collapse.

为了解决这个问题，CycleGAN 中引入了循环一致性损失。仅仅有 G : X → Y G:X->Y G:X→Y 是不够的，还需要再引入一个映射 F : Y → X F:Y-> X F:Y→X，以将 G (X) G(X)G(X) 重新映射回源域 X XX，并衡量 F (G (X)) F(G(X))F(G(X)) 与 X XX 之间的差距，希望这个差距越小越好。这种思想相当于是一种 Autoencoder，G (X) G(X)G(X) 相当于 AE 中的编码，如果这个编码能够还原出与 X XX 相似的 F (G (X)) F(G(X))F(G(X))，那么就可以认为 G (X) G(X)G(X) 虽然接近目标域分布 Y YY，但原始图像上的语义特征并没丢失。

有了这一重约束，上面例子中的「你好吗」就不会翻译成「I’m fine」，因为「I’m fine」的语义特征和「你好吗」不同，不易从「I’m fine」还原到「你好吗」。

因此，整个网络的模型架构如下图所示：



图（a）展示的是$G:X\rightarrow Y$和$F:Y\rightarrow X$两个生成器映射过程，D_Y鉴别的是$G(X)$属于目标域分布Y的确定性，和GAN中的鉴别器原理一致；D_X与之同理；图（b）展示的正向循环一致性损失，即$x\rightarrow G(x)\rightarrow F(G(x))\approx x$；图（c）展示的反向循环一致性损失，即$y\rightarrow F(y)\rightarrow G(F(y))\approx y$。

因此整个训练过程的损失函数定义如下：

$$L(G,F,D_X,D_Y)=L_{GAN}(G,D_Y,X,Y)+L_{GAN}(F,D_X,Y,X)+\lambda L_{cyc}(G,F)$$

其中前两项是常规生成对抗网络中的损失函数，最后一项是循环一致性损失函数，λ是一个调节系数。对于前两项，这里采用了LSGAN的思想，可以进一步表达为：

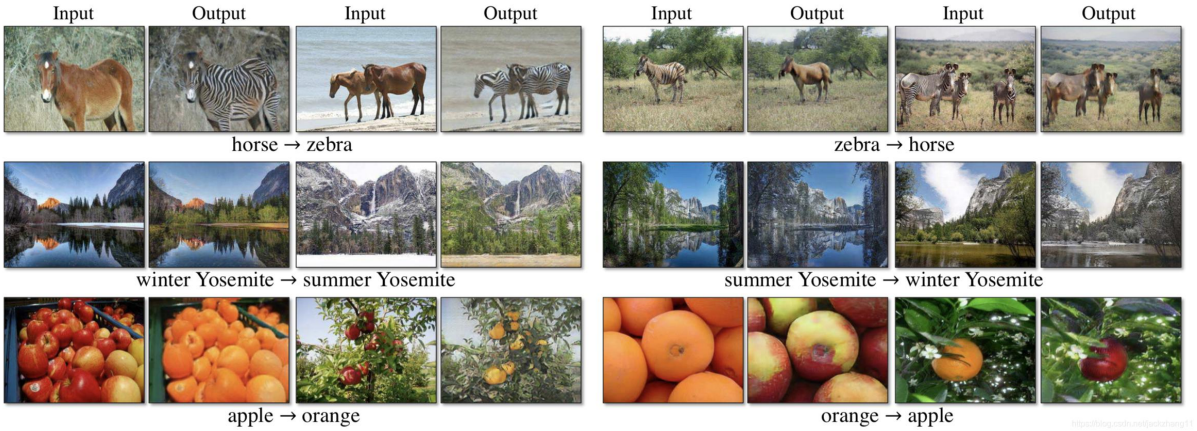
$$L_{LSGAN}(G,D_Y,X,Y)=E_{y\sim p_{data}(y)}[(D_Y(y)-1)^2]+E_{x\sim p_{data}(x)}[D_Y(G(x))^2]$$

$$L_{LSGAN}(F,D_X,X,Y)=E_{x\sim p_{data}(x)}[(D_X(x)-1)^2]+E_{y\sim p_{data}(y)}[D_X(G(y))^2]$$

对于循环一致性损失，其相当于自动编码器的重构误差，可以表达为：

$$L_{cyc}(G,F)=E_{x\sim p_{data}(x)}[||F(G(x))-x||_1]+E_{y\sim p_{data}(y)}[||G(F(y))-y||_1]$$

CycleGAN的理论到这里就差不多了，可以看出理论并不是很复杂，而是非常精妙。巧妙的引入了循环一致性改变了网络的结构，不再需要成对的数据，从而使得CycleGAN非常的generalized。下面是论文中CycleGAN的一些实验结果：



根据上图，CycleGAN 可以实现斑马和马，冬天夏天即橙子苹果间的转换，论文中还做了很多有趣的实验，如图像增强，将随机拍摄的自然图像转换为专业的摄影图片，或融入莫奈的画风。

2. 数据获取及预处理

对于大数据而言，TensorFlow 推荐使用自带的 tfrcords 文件。tfrecords 文件是以二进制进行存储的，适合以串行的方式读取大批量的数据。对于训练数据而言，我们可以编写程序将普通的训练数据保存为 tfrecords 数据格式。

```
GCS_PATH = KaggleDatasets().get_gcs_path()
```

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

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

```
Monet TFRecord Files: 5
Photo TFRecord Files: 20
2021-11-09 00:58:08.316774: I tensorflow/core/platform/cloud/google_auth_provider.cc:180] Attempting an empty bearer token since no token was retrieved from files, and GCE metadata check was skipped.
2021-11-09 00:58:08.392115: I tensorflow/core/platform/cloud/google_auth_provider.cc:180] Attempting an empty bearer token since no token was retrieved from files, and GCE metadata check was skipped.
```

编写函数将 tfrcords 文件转化为图片格式的文件：

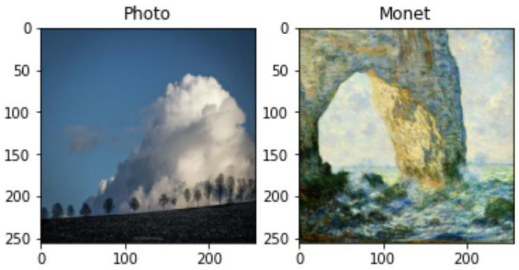
```
IMAGE_SIZE = [256, 256]

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
```

数据集中所有图片尺寸均已达到 256x256。由于这些图像是 RGB 图像，将通道设置为 3。此外，需要将图像缩放到[-1,1]比例，构建一个生成模型，不需要标签或图像 id，所以只从 TFRecord 返回图像。

然后，我们可以选取一组数据集中的照片与莫奈风格画可视化并观察

```
plt.subplot(121)
plt.title('Photo')
plt.imshow(example_photo[0] * 0.5 + 0.5)

plt.subplot(122)
plt.title('Monet')
plt.imshow(example_monet[0] * 0.5 + 0.5)
```

: <matplotlib.image.AxesImage at 0x7fbf8c4dbc50>



3. 设计生成器

我们将在 CycleGAN 中使用 UNET 架构。为了构建生成器，让我们首先定义下采样和上采样方法。顾名思义，下采样通过步幅减少了图像的二维尺寸、宽度和高度。步幅是过滤器采取的步长。由于步幅为 2，因此过滤器将每隔一个像素应用一次，从而将重量和高度减少 2。我们将使用实例规范化而不是批处理规范化。由于实例规范化在 TensorFlow API 中不是标准的，我们将使用 TensorFlow 附加组件中的层。

上采样”与“下采样”相反，并增加图像的尺寸。Conv2DTranspose 基本上与 Conv2D 层相反。

```
OUTPUT_CHANNELS = 3

def downsample(filters, size, apply_instancenorm=True):
    initializer = tf.random_normal_initializer(0., 0.02)
    gamma_init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)

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

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

    result.add(layers.LeakyReLU())

    return result
```

设计生成器，生成器首先对输入图像进行下采样，然后在建立长跳跃连接时进行上采样。跳过连接是一种通过将一个层的输出连接到多个层而不是仅一个层来帮助绕过消失梯度问题的方法。在这里，我们以对称方式将下采样层的输出连接到上采样层。

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

    # bs = batch size
    down_stack = [
        downsample(64, 4, apply_instancenorm=False), # (bs, 128, 128, 64)
        downsample(128, 4), # (bs, 64, 64, 128)
        downsample(256, 4), # (bs, 32, 32, 256)
        downsample(512, 4), # (bs, 16, 16, 512)
        downsample(512, 4), # (bs, 8, 8, 512)
        downsample(512, 4), # (bs, 4, 4, 512)
        downsample(512, 4), # (bs, 2, 2, 512)
        downsample(512, 4), # (bs, 1, 1, 512)
    ]

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

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

    x = inputs

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

    skips = reversed(skips[:-1])

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

    x = last(x)

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

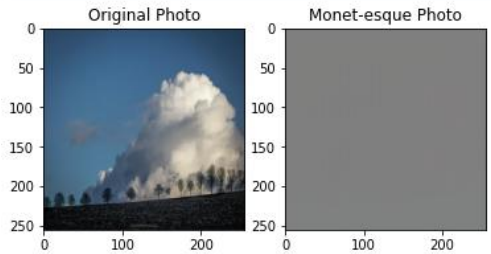
4. 设计判别器

鉴别器接收输入图像并将其分类为真图像或假图像（生成）。鉴别器不输出单个节点，而是输出较小的 2D 图像，其中较高的像素值指示真实分类，较低的值指示假分类。

```
def Discriminator():
    initializer = tf.random_normal_initializer(0., 0.02)
    gamma_init = keras.initializers.RandomNormal(mean=0.0, stddev=0.02)

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

    x = inp

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

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

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

    leaky_relu = layers.LeakyReLU()(norm1)

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

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

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

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

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

```
to_monet = monet_generator(example_photo)

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

plt.subplot(1, 2, 2)
plt.title("Monet-esque Photo")
plt.imshow(to_monet[0] * 0.5 + 0.5)
plt.show()
```



5. 搭建 CycleGan 模型

我们将对 tf.keras.Model 进行子类化，以便稍后运行 fit（）来训练我们的模型。在训练步骤中，模型将照片转换为莫奈绘画，然后再转换回照片。原始照片和两次变换照片之间的差异是周期一致性损失。我们希望原始照片和两次变换的照片彼此相似，之后再定义损失。

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

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

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

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

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

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

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

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

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

            # evaluates total cycle consistency loss
            total_cycle_loss = self.cycle_loss_fn(real_monet, cycled_monet, self.lambda_cycle) + self.cycle_loss_fn(real_photo, cycled_photo, self.lambda_cycle)

            # evaluates total generator loss
            total_monet_gen_loss = monet_gen_loss + total_cycle_loss + self.identity_loss_fn(real_monet, same_monet, self.lambda_cycle)
            total_photo_gen_loss = photo_gen_loss + total_cycle_loss + self.identity_loss_fn(real_photo, same_photo, self.lambda_cycle)

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

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

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

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

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

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

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

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

6. 设计损失函数

下面的鉴别器损失函数将真实图像与 1 的矩阵进行比较，将假图像与 0 的矩阵进行比较。完美鉴别器将输出真实图像的所有 1，假图像的所有 0。鉴别器损耗输出实际损耗和生成损耗的平均值。

```
with strategy.scope():
    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)

        total_disc_loss = real_loss + generated_loss

        return total_disc_loss * 0.5
```

生成器想愚弄鉴别器，使其认为生成的图像是真实的。完美的发生器只有 1s 的鉴别器输出。因此，它将生成的图像与 1 的矩阵进行比较，以找出损失。

```
with strategy.scope():
    def generator_loss(generated):
        return tf.keras.losses.BinaryCrossentropy(from_logits=True, reduction=tf.keras.losses.Reduction.NONE)(tf.ones_like(generated), generated)
```

我们希望我们的原始照片和两次转换的照片彼此相似。因此，我们可以通过求其差值的平均值来计算周期一致性损失。

```
with strategy.scope():
    def calc_cycle_loss(real_image, cycled_image, LAMBDA):
        loss1 = tf.reduce_mean(tf.abs(real_image - cycled_image))

        return LAMBDA * loss1
```

身份丢失会将图像与其生成器（即照片与照片生成器）进行比较。如果给定一张照片作为输入，我们希望它生成与原始照片相同的图像。标识丢失将生成器的输入与输出进行比较。

```
with strategy.scope():
    def identity_loss(real_image, same_image, LAMBDA):
        loss = tf.reduce_mean(tf.abs(real_image - same_image))
        return LAMBDA * 0.5 * loss
```

7. 训练 CycleGAN

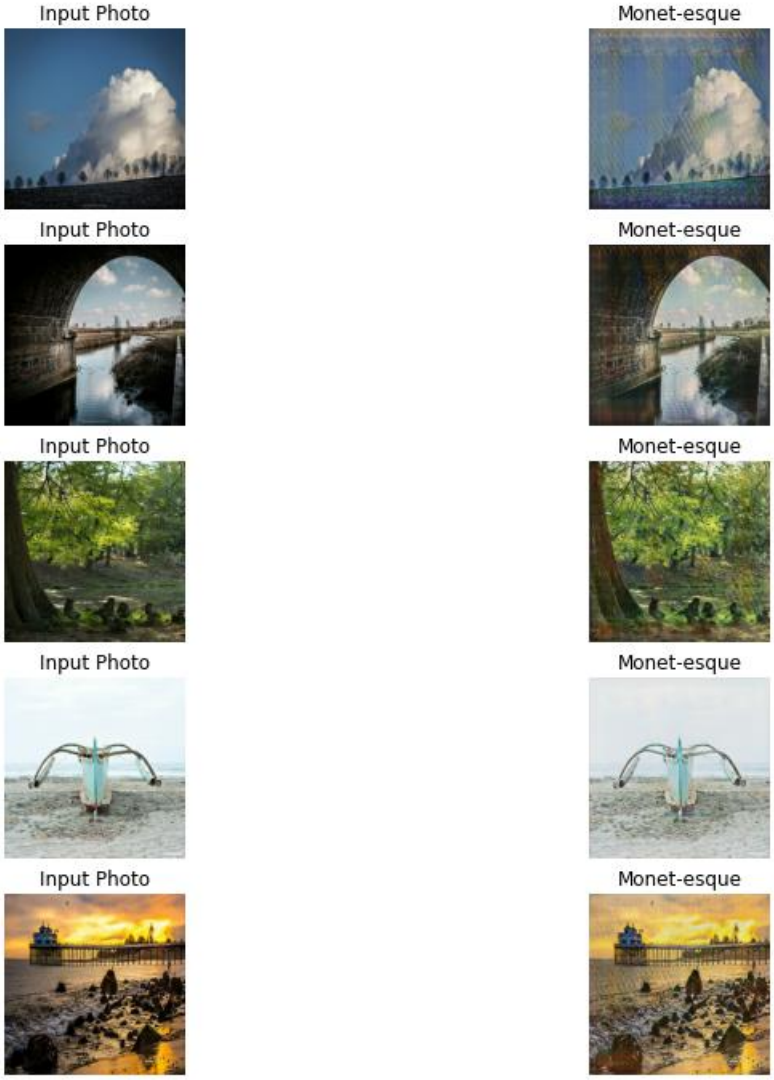
开始编译 CycleGAN 的模型。由于我们使用 tf.keras.Model 来构建我们的 CycleGAN，因此我们可以使用拟合函数来训练我们的模型。

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

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

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

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

8. 结果可视化

可视化输出的莫奈风格的照片：



最后将输出的莫奈风格的图像保存在 output 文件夹下。