

Transform Photos to Monet Paintings with CycleGANs

Imagine this: you take a photo of an airplane flying against a clear blue sky. Nice, right? Now, imagine if you could transform that photo into something that looks like it was painted by an artist—like it belongs in a famous art gallery. Instead of just a regular photo, the sky becomes dreamy and swirly, the airplane looks like it's gliding through a magical painting, and everything feels like it came out of a fairy tale. That's what style transfer does!

What's style transfer?

Think of style transfer as teaching a computer to "paint." It takes the style of one image like a famous painting and applies it to the content of another image. It's like asking the computer, "What would this photo look like if Claude Monet or Vincent van Gogh painted it?"

How does this happen?

Here's the cool part: it's not magic—it's technology! There's something called CycleGAN, a type of smart computer model. It's like having two artists that help each other out. One artist learns to turn your photo into a painting, and the other learns to make sure it still looks like the original image. Together, they train each other until they get really, really good.

Why is this special?

Even professional artists might find it hard to perfectly mimic styles like Monet or van Gogh. But with the help of models like CycleGAN, anyone can take their regular photos and turn them into stunning, artistic images. It's like giving your pictures an artistic glow-up!



Setup

For this project, we will be using the following libraries:

- `numpy` for mathematical operations.
- `Pillow` for image processing functions.
- `tensorflow` for machine learning and neural network related functions.
- `matplotlib` for additional plotting tools.

Installing Required Libraries

```
In [1]: # All Libraries required for this lab are listed below.  
!mamba install -qy numpy==1.22.3 matplotlib==3.5.1 tensorflow==2.9.0 skillsr  
  
# Note: If the environment doesn't support "!mamba install", use "!pip insta
```

Could not solve for environment specs

The following packages are incompatible

```

└─ matplotlib 3.5.1 is installable with the potential options
  └─ matplotlib 3.5.1 would require
    └─ matplotlib-base >=3.5.1,<3.5.2.0a0 with the potential options
      └─ matplotlib-base 3.5.1 would require
        └─ python >=3.10,<3.11.0a0 , which can be installed;
      └─ matplotlib-base 3.5.1 would require
        └─ freetype >=2.11.0,<3.0a0 , which can be installed;
      └─ matplotlib-base 3.5.1 would require
        └─ python >=3.8,<3.9.0a0 , which can be installed;
      └─ matplotlib-base 3.5.1 would require
        └─ python >=3.9,<3.10.0a0 , which can be installed;
    └─ pyqt with the potential options
      └─ pyqt [5.15.10|5.15.7|5.9.2] would require
        └─ python >=3.10,<3.11.0a0 , which can be installed;
      └─ pyqt [5.15.10|5.15.7] would require
        └─ python >=3.11,<3.12.0a0 , which can be installed;
      └─ pyqt 5.15.10 would require
        └─ python >=3.12,<3.13.0a0 , which can be installed;
      └─ pyqt 5.15.10 would require
        └─ python_abi 3.13.* *_cp313, which can be installed;
      └─ pyqt [5.15.10|5.15.7|5.9.2] would require
        └─ python >=3.8,<3.9.0a0 , which can be installed;
      └─ pyqt [5.15.10|5.15.7|5.9.2] would require
        └─ python >=3.9,<3.10.0a0 , which can be installed;
      └─ pyqt 5.15.7 would require
        └─ qt-main 5.15.* with the potential options
          └─ qt-main 5.15.2 would require
            └─ icu >=73.1,<74.0a0 , which can be installed;
          └─ qt-main 5.15.2 would require
            └─ icu >=58.2,<59.0a0 , which can be installed;
          └─ qt-main 5.15.2 would require
            └─ openssl >=3.0.10,<4.0a0 , which can be installed;
          └─ qt-main 5.15.2 would require
            └─ openssl >=3.0.9,<4.0a0 , which can be installed;
          └─ qt-main 5.15.2 would require
            └─ openssl >=3.0.15,<4.0a0 , which can be installed;
          └─ qt-webengine 5.15.* with the potential options
            └─ qt-webengine 5.15.9 would require
              └─ qt >=5.15.2,<6 , which can be installed;
            └─ qt-webengine 5.15.9 would require
              └─ qt 5.15.9 , which can be installed;
        └─ pyqt [5.6.0|5.9.2] would require
          └─ python >=2.7,<2.8.0a0 , which can be installed;
        └─ pyqt [5.6.0|5.9.2] would require
          └─ python >=3.5,<3.6.0a0 , which can be installed;
        └─ pyqt [5.6.0|5.9.2] would require
          └─ python >=3.6,<3.7.0a0 , which can be installed;
        └─ pyqt [5.6.0|5.9.2] would require
          └─ qt [5.6.* |5.9.* ] with the potential options
            └─ qt [5.6.2|5.9.4|5.9.5|5.9.6] would require
              └─ openssl 1.0.* , which can be installed;
            └─ qt 5.6.2 would require
              └─ openssl >=1.0.2n,<1.0.3a , which can be installed;
            └─ qt 5.6.3 would require

```



```
import matplotlib.pyplot as plt

%matplotlib inline

import os
from os import listdir
from pathlib import Path
import imgchr

import time
from tqdm.auto import tqdm
import time
from PIL import Image
import random
```

```
2024-12-25 01:04:18.723492: I tensorflow/core/platform/cpu_feature_guard.cc:1
93] This TensorFlow binary is optimized with oneAPI Deep Neural Network Libr
ary (oneDNN) to use the following CPU instructions in performance-critical ope
rations: AVX2 AVX512F AVX512_VNNI FMA
To enable them in other operations, rebuild TensorFlow with the appropriate c
ompiler flags.
2024-12-25 01:04:18.972063: I tensorflow/core/util/port.cc:104] oneDNN custom
operations are on. You may see slightly different numerical results due to fl
oating-point round-off errors from different computation orders. To turn them
off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2024-12-25 01:04:18.984076: W tensorflow/compiler/xla/stream_executor/platfor
m/default/dso_loader.cc:64] Could not load dynamic library 'libcudart.so.11.
0'; dLError: libcudart.so.11.0: cannot open shared object file: No such file
or directory
2024-12-25 01:04:18.984112: I tensorflow/compiler/xla/stream_executor/cuda/cu
dart_stub.cc:29] Ignore above cudart dLError if you do not have a GPU set up
on your machine.
2024-12-25 01:04:20.064867: W tensorflow/compiler/xla/stream_executor/platfor
m/default/dso_loader.cc:64] Could not load dynamic library 'libnvinfer.so.7';
dLError: libnvinfer.so.7: cannot open shared object file: No such file or dir
ectory
2024-12-25 01:04:20.065037: W tensorflow/compiler/xla/stream_executor/platfor
m/default/dso_loader.cc:64] Could not load dynamic library 'libnvinfer_plugi
n.so.7'; dLError: libnvinfer_plugin.so.7: cannot open shared object file: No
such file or directory
2024-12-25 01:04:20.065072: W tensorflow/compiler/tf2tensorrt/utils/py_utils.
cc:38] TF-TRT Warning: Cannot dlopen some TensorRT libraries. If you would li
ke to use Nvidia GPU with TensorRT, please make sure the missing libraries me
ntioned above are installed properly.
tensorflow version: 2.11.0
skillsnetwork version: 0.20.6
```

If you choose to download this notebook and run it in an environment where a TPU accelerator is available, the following code does the initialization work and allows you to implement synchronous distributed training with a TPU `strategy`.

```
In [3]: try:
        tpu = tf.distribute.cluster_resolver.TPUClusterResolver()
```

```

print('Device:', tpu.master())
tf.config.experimental_connect_to_cluster(tpu)
tf.tpu.experimental.initialize_tpu_system(tpu)
strategy = tf.distribute.experimental.TPUStrategy(tpu)
except:
    strategy = tf.distribute.get_strategy()
print('Number of replicas:', strategy.num_replicas_in_sync)

AUTOTUNE = tf.data.experimental.AUTOTUNE

print(tf.__version__)

```

Number of replicas: 1
2.11.0

Defining Helper Functions

```

In [4]: def decode_image(file_path):

        """
        return a decoded tf tensor given an image file path,

        """
        image = tf.io.read_file(file_path)
        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 load_dataset(directory, labeled=True):

        """
        return a tf.data.Dataset consisting of images from directory, decoded by

        """
        dataset = tf.data.Dataset.list_files(directory + "/*.jpg")
        dataset = dataset.map(decode_image, num_parallel_calls=AUTOTUNE)

        return dataset

```

What is Image Style Transfer in Deep learning?

Image Style Transfer is a fascinating concept in deep learning that allows us to take the content of one image and apply the "style" of another image to it. It's like creating a new image that looks like it was painted by an artist but still contains the details of a real photograph. Think of it as combining two images into one in a creative and artistic way.

Image Style Transfer allows us to creatively combine the content of one image like a photo of a landscape, a person, or a horse with the style of another such as a famous painting, a season, or an animal's skin. It's like taking the essence of one image and dressing it up in the look of another.

You can use this technique in many fun and practical ways. For instance, it can turn a regular photo into something that looks like a famous artist's painting, change the season in a photo, or make animals look different without having to take new pictures. It's a powerful tool used in various fields, from art creation to virtual reality and more!

CycleGANs

Understanding Vanilla GANs

The term "vanilla" GANs refers to the simplest, most basic version of Generative Adversarial Networks (GANs), introduced by Ian Goodfellow and colleagues in 2014 in their groundbreaking paper "Generative Adversarial Nets." In everyday language, "vanilla" is often used to describe something plain, fundamental, or without any extra features or complexities. Similarly, vanilla GANs are the foundation of all GANs without any modifications or advanced enhancements.

What Makes Vanilla GANs "Vanilla"?

Basic Architecture:

A vanilla GAN has two networks:

1. A Generator (G): Creates fake data from random noise.
2. A Discriminator (D): Distinguishes between real and fake data.

These two networks work in opposition, improving each other.

Simple Loss Function:

The loss function in vanilla GANs is straightforward: The Generator tries to maximize the likelihood of fooling the Discriminator. The Discriminator tries to maximize its ability to distinguish real from fake. No advanced tricks or techniques are used for stability.

No Special Modifications:

Vanilla GANs do not include techniques like:

1. Wasserstein loss (used in WGANs for stability).
2. Gradient penalties.

3. Conditional inputs (used in Conditional GANs for specific outputs like labeled images).
4. Feature matching or architectural enhancements like attention mechanisms.

Standard Training Process:

The Generator and Discriminator are trained alternately, iterating in a straightforward adversarial loop.

Why Start with Vanilla GANs?

Vanilla GANs are like the "101" or foundation of GANs. Understanding them helps us:

- Grasp the core idea of adversarial learning.

See the challenges, like:

1. Training instability: Sometimes, one network (G or D) gets too strong, making training hard.
2. Mode collapse: The Generator may start producing limited or repetitive outputs instead of diverse ones.
3. Build the groundwork for understanding advanced GANs, like WGANs, DCGANs, and StyleGANs.

Analogy: Vanilla Ice Cream

Think of "vanilla" in GANs as the vanilla flavor in ice cream: It's the basic, original flavor. It doesn't have toppings or extra ingredients like chocolate chips or caramel swirls (which would represent modifications in advanced GANs). While plain, it's essential because other flavors build on top of it.

Similarly, vanilla GANs are the plain, unadorned version that serves as the base for all the advanced GAN architectures.

Think of it as two teams playing a game where each tries to outsmart the other. These two teams are the Generator and the Discriminator. Together, they form the GAN system.

The Two Players: Generator (G) and Discriminator (D)

- Generator (G): The Artist Trying to Create Fakes

Imagine a budding artist trying to paint landscapes. At first, their paintings might not look real—they may lack proper colors, depth, or details. The Generator is like this artist.

Its job is to create fake images (or data) that are similar to real images, but it starts out knowing nothing.

- Discriminator (D): The Art Critic

Now, think of an art critic who inspects paintings and decides whether they're real masterpieces or amateurish fakes. The Discriminator is this critic. It examines images and determines whether they're real from real-world data or fake created by the Generator.

- The Learning Game: G and D Compete

The Generator and the Discriminator are locked in a constant game:

The Generator tries to fool the Discriminator by creating fake images that look like real ones. The Discriminator tries to catch the fakes and become better at identifying fake images. This competition keeps both improving:

The Generator learns to make better and better fake images. The Discriminator learns to get better at spotting fakes.

The Process in Detail

Step 1: Start with Noise

The Generator starts with some random input called noise (like random scribbles on a canvas). It tries to turn this noise into an image.

Step 2: Real Images vs. Fake Images

The Discriminator is shown two types of images: Real images (actual photos from the dataset, like pictures of cats, dogs, or landscapes). Fake images (created by the Generator from random noise). The Discriminator then guesses whether each image is real or fake.

Step 3: Feedback Loop

The Generator gets feedback from the Discriminator: If the Discriminator catches the fake, it tells the Generator, "Your fake is bad—try harder!" If the Discriminator gets fooled, it adjusts itself to improve for next time. The Discriminator also gets feedback to improve its ability to spot fakes.

The Training Objective

- For the Generator (G):

Its goal is to make images so realistic that the Discriminator gets confused and labels them as "real." In simple terms, the Generator keeps asking itself: "How can I improve so that my fakes look real enough to fool the critic?"

- For the Discriminator (D):

Its goal is to correctly classify real images as "real" and fake images as "fake." In other words, the Discriminator keeps asking itself: "How can I catch even the tiniest flaws in the fake images?"

- Let's imagine this as a real-world scenario:

The Generator is a counterfeiter trying to create fake money. The Discriminator is a bank employee trained to spot counterfeit bills.

At first, the Generator's fake money is terrible—maybe it's just a blank piece of paper. The bank employee (Discriminator) easily catches it and rejects it. The Generator learns from this failure and makes better fakes—maybe it starts printing bills with colors and numbers. The bank employee also improves, learning to check for more subtle details like watermarks or texture. This back-and-forth continues until the Generator creates fake money so realistic that even the bank employee struggles to tell if it's fake or real.

Key Idea: Learning by Comparison

GANs work because both networks (Generator and Discriminator) keep pushing each other to improve:

The Generator improves its ability to create realistic images. The Discriminator sharpens its skills to catch the fakes. It's like a competitive game where both players get better over time.

Example: Generating Cat Images

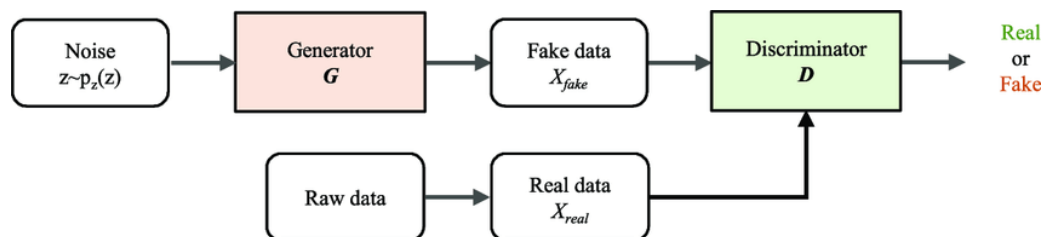
Let's say we want a GAN to generate images of cats. Here's how it works:

We have a dataset of real cat images. The Generator starts with random scribbles and tries to create something that looks like a cat. The Discriminator looks at these images and says:

"This is a real cat!" (if it's from the real dataset).

"This is a fake cat!" (if it's from the Generator).

Over time, the Generator's fake cats start looking more and more like real cats.



Visualization

The picture in the original post shows this back-and-forth process:

The Generator takes random input and creates fake images. The Discriminator compares the fake images with real ones and provides feedback. Both networks improve together, making the Generator's images better and the Discriminator's judgment sharper.

Why is this Powerful?

GANs are powerful because they don't need labeled data. They just need real-world examples (like photos of cats) to train on. This makes GANs useful for tasks like:

1. Creating realistic images.
2. Generating new artwork.
3. Simulating real-world data.

What is CycleGAN, and How is It Different?

Imagine you want to create a program that can take a winter photo and turn it into a summer photo, or take a picture of a horse and make it look like a zebra. Most machine learning models that do this kind of image translation need paired data—for example, a winter photo and its exact summer version taken at the same spot. But CycleGANs don't need such paired data.

Instead, CycleGANs can learn to make the transformations using two separate collections of images:

A bunch of winter photos. A bunch of summer photos.

Why Is This Important?

For many tasks, it's hard (or even impossible) to get paired data.

Example:

If you want to convert old black-and-white photos into color photos, you likely won't have the exact colored versions of those old photos. If you want to translate paintings into realistic photos (e.g., Van Gogh's style into modern landscapes), paired examples don't exist. CycleGAN solves this problem by using a new way of training, including something called Cycle Consistency Loss.

What Is Forward-Backward Consistency?

Let's imagine we are converting units of measurement. Suppose you start with a distance in miles, convert it to kilometers, and then convert it back to miles. we'd expect to end up with the same number of miles we started with.

Example: Start with 10 miles. Convert it to kilometers: $10 \text{ miles} \times 1.60934 = 16.0934$ kilometers. Convert it back to miles: $16.0934 \text{ kilometers} \div 1.60934 = 10 \text{ miles}$. This ability to go forward (miles to kilometers) and then backward (kilometers to miles) while preserving the original value is an example of forward-backward consistency.

In the context of images and CycleGANs, the idea is similar: If you transform a winter photo into a summer photo and then transform it back, it should look like the original winter photo. This ensures that the transformations are meaningful and consistent.

Why Is Forward-Backward Consistency Important in Images?

When translating images, without forward-backward consistency, the translation might not make sense, because:

Let's say we have two groups of photos:

Photos of Mountains (Domain X) Photos of Beaches (Domain Y) Now, the Generator in a CycleGAN learns how to turn a mountain photo into a beach photo (called mapping $X \rightarrow Y$) and vice versa. But here's the problem:

If there's no forward-backward consistency, the mountain might turn into a random beach photo without retaining any of the original features of the mountain.

Example: A snowy mountain might turn into a sandy beach. But when you convert that beach back into a mountain, it might no longer look like the original snowy mountain—it could turn into something random. This is why CycleGANs enforce forward-backward consistency. The model learns to transform a mountain into a beach and then back into the same mountain, ensuring the transformation is meaningful and consistent.

How Does CycleGAN Achieve This?

CycleGAN introduces two main Generators:

Generator G: Translates images from Domain X (mountains) to Domain Y (beaches).

Generator F: Translates images from Domain Y (beaches) back to Domain X (mountains).

It also introduces Cycle Consistency Loss, which measures how close the images are after completing the full cycle:

Start with a mountain image (x). Convert it into a beach image using $G(x)$. Convert the beach image back into a mountain using $F(G(x))$. Compare the final mountain image ($F(G(x))$) to the original mountain (x). If the two are very different, the model is penalized, and it learns to improve. The same happens in the reverse direction (beaches to mountains).

Example with Cats and Dogs Now lets say we are converting:

Photos of cats into dogs. Photos of dogs back into cats. Without forward-backward consistency, you might start with a photo of a white cat:

The model might turn it into a black Labrador retriever. When you convert it back into a cat, it might come out as a striped tiger!

Cycle Consistency Loss ensures this doesn't happen. It forces the model to retain key features (like the white fur of the original cat) through the transformations.

Why Is This So Powerful?

CycleGAN's ability to work without paired data makes it incredibly versatile. You can use it for:

Artistic transformations e.g., turning photos into paintings or vice versa. Style changes e.g., summer to winter landscapes. Domain adaptations e.g., turning sketches into realistic photos. And all of this is done without requiring exact "before and after" examples, which are hard to collect in most cases.

Key Takeaways

1. Paired data isn't required: You just need two sets of images e.g., winter photos and summer photos.
2. Forward-backward consistency: The model ensures that transformations are meaningful by comparing original images to those that have gone through a full cycle of transformations.
3. Cycle Consistency Loss: A unique loss function that makes sure the transformations don't lose key details of the original image.

By enforcing these ideas, CycleGAN allows us to do complex image translations in a way that's simple, efficient, and practical for real-world applications.

Data Loading

The unpaired dataset comes from a Kaggle competition called [I'm Something of a Painter Myself](#). The original dataset contains around 400MB of images, but for this project we will only use 300 Monet paintings and 300 photos for training the CycleGAN. The following cell downloads the zipped dataset.

```
In [5]: await skillsnetwork.prepare("https://cf-courses-data.s3.us.cloud-object-storage.com/output/gan_getting_started_300.zip")
Downloading gan_getting_started_300.zip: 0%|          | 0/9155521 [00:00<?, ?it/s]
0%|          | 0/605 [00:00<?, ?it/s]
Saved to '.'
```

In the current working directory, there is a folder called `gan_getting_started_300` which contains two subfolders: `monet_jpg_300` and `photo_jpg_300`. Each of the subfolders contains 300 JPG format images:

```
In [6]: IMG_PATH = "gan_getting_started_300"

MONET_FILENAMES = tf.io.gfile.glob(str(IMG_PATH + '/monet_jpg_300/*.jpg'))
print('Monet JPG Files:', len(MONET_FILENAMES))

PHOTO_FILENAMES = tf.io.gfile.glob(str(IMG_PATH + '/photo_jpg_300/*.jpg'))
print('Photo JPG Files:', len(PHOTO_FILENAMES))

Monet JPG Files: 300
Photo JPG Files: 300
```

The `load_dataset` helper function utilizes the `Dataset` API from `tf.data`, which allows us to build asynchronous, highly optimized data pipelines. The `.batch(n)` combines consecutive elements of a dataset into batches. We create two `tf.data.Dataset` objects for feeding our Monet painting and photo data into the CycleGAN in a streaming fashion.

```
In [7]: IMAGE_SIZE = [256, 256]

monet_ds = load_dataset("gan_getting_started_300/monet_jpg_300", labeled=True)
```

```
photo_ds = load_dataset("gan_getting_started_300/photo_jpg_300", labeled=True)
monet_ds, photo_ds
```

```
2024-12-25 01:04:45.590897: W tensorflow/compiler/xla/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'libcuda.so.1'; dlerror: libcuda.so.1: cannot open shared object file: No such file or directory
2024-12-25 01:04:45.590951: W tensorflow/compiler/xla/stream_executor/cuda/cuda_driver.cc:265] failed call to cuInit: UNKNOWN ERROR (303)
2024-12-25 01:04:45.590992: I tensorflow/compiler/xla/stream_executor/cuda/cuda_diagnostics.cc:156] kernel driver does not appear to be running on this host (jupyterlab-deepthiskp1): /proc/driver/nvidia/version does not exist
2024-12-25 01:04:45.591428: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations: AVX2 AVX512F AVX512_VNNI FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
```

```
Out[7]: (<BatchDataset element_spec=TensorSpec(shape=(None, 256, 256, 3), dtype=tf.float32, name=None)>,
        <BatchDataset element_spec=TensorSpec(shape=(None, 256, 256, 3), dtype=tf.float32, name=None)>)
```

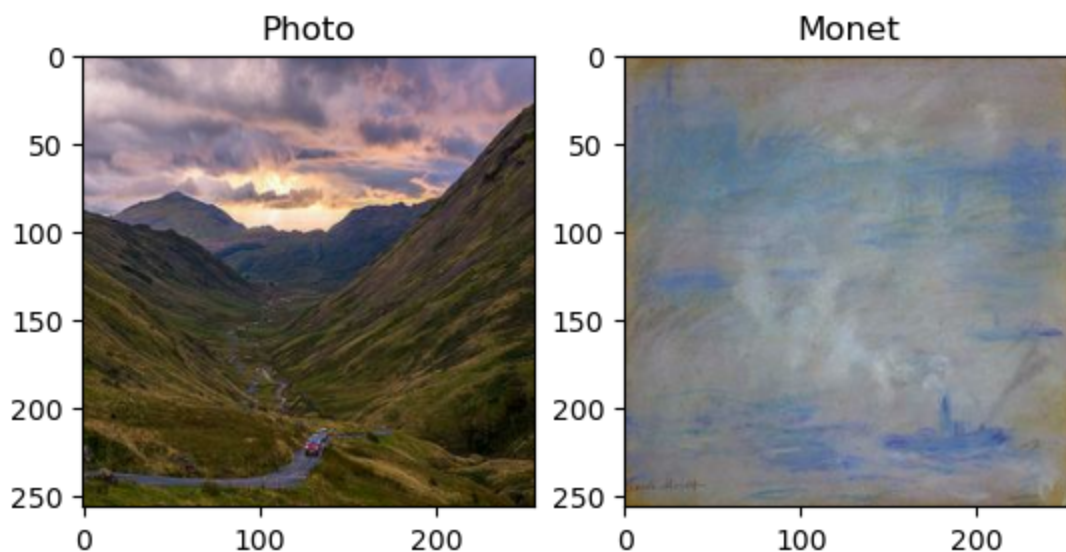
Let's visualize a photo example and a Monet example:

```
In [8]: example_monet = next(iter(monet_ds))
        example_photo = next(iter(photo_ds))
```

```
In [9]: 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)
```

```
Out[9]: <matplotlib.image.AxesImage at 0x7fa94c716210>
```

Building the Generator

Understanding the Downsampling Block

When building a CycleGAN generator, the first step is encoding the input image. This is done using downsampling blocks, which gradually reduce the dimensions of the image while preserving important features.

What Is a Downsampling Block?

Imagine you have a large, detailed map of a city. To carry it in your pocket, you fold it multiple times. Each fold makes the map smaller, but the critical details—like landmarks and roads—are still there, just in a more compact form.

In a CycleGAN generator, the downsampling block does something similar:

It "folds" the image (reducing its width and height) by applying a convolutional layer with strides (steps). While reducing the size, it keeps the most important features, such as edges, shapes, and patterns. This compact version of the image is easier for the generator to process and transform into the desired style (e.g., winter to summer).

Why Use Instance Normalization Instead of Batch Normalization?

Imagine you're in a room with different lighting conditions. You want to compare two photos—one taken in bright daylight and another taken at dusk. To make this comparison

easier, you adjust the brightness of both photos to a neutral level. This adjustment is like normalization, which helps remove unnecessary information like contrast.

1. Batch Normalization:

This adjusts the brightness or features based on the entire group of photos in the batch. Works well when you have a large batch size (many photos being processed together).

2. Instance Normalization:

This adjusts the brightness or features for each photo independently. Perfect for situations where you are working with small batch sizes (like 1-2 images at a time). By focusing on individual images, it removes unnecessary contrast while preserving the important content of each image.

Why Instance Normalization Improves Results

Imagine we're training a model to transform landscape photos into paintings. Each photo might have different lighting (day, night, foggy) or contrast. With instance normalization, the model treats each photo independently and focuses on its unique style and features. This avoids "blurring" the differences between photos in the batch and results in better, sharper generated images.

How It All Fits Together

In the downsampling block:

Convolution extracts key features while reducing the image size. Instance Normalization ensures each image is normalized independently, making it easier for the generator to focus on the task of style transformation. This combination helps the CycleGAN generator learn efficiently and produce realistic and meaningful images, even with a small dataset or varying image styles.

```
In [10]: 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(tf.nn.InstanceNormalization(gamma_initializer=gamma_

result.add(layers.LeakyReLU())

return result
```

The defined downsampling block provides an efficient way for us to add multiple **Convolution-InstanceNorm-LeakyReLU** layers with variable number of filters and filter sizes.

Defining the Upsampling Block

Once the generator has compressed the input image into a smaller, feature-rich version using downsampling, it needs to "rebuild" or expand the image to its original size—but in the new style (e.g., winter to summer). This process of increasing the dimensions of an image is called upsampling.

What is Upsampling?

Imagine you have a tiny, pixelated version of an image (like an old-school video game graphic). Upsampling is like gradually adding more details to make the image larger and clearer, eventually reaching its full size.

Example: Start with a small 10x10 pixel grid (a simplified version of the image). Gradually increase it to 20x20, then 40x40, and finally to 100x100 pixels.

Upsampling doesn't just stretch the image; it intelligently "fills in" missing details to make the larger version look natural and realistic.

How Does Upsampling Work in the Generator?

In a CycleGAN generator, upsampling blocks are used to rebuild the compressed image into its full-sized output. To achieve this, we use transpose convolution, also known as "deconvolution."

Transpose Convolution vs. Normal Convolution

1. Normal Convolution (Downsampling):

Think of using a cookie cutter on a large piece of dough. Each cut (convolution) reduces the dough's size, leaving only the important features (the cookies). Example: Start with a large photo and shrink it by keeping only key features.

2. Transpose Convolution (Upsampling):

Imagine taking the cookies and placing them back on a larger tray. As you do this, you fill in the gaps to recreate a full-sized tray of dough. Example: Start with a small, simplified photo and expand it back to its original size, filling in the missing details. Building the Upsampling Block

To build an upsampling block, we use the following components:

1. Transpose Convolution (Conv2DTranspose):

This increases the width and height of the image while learning to add meaningful details. Example: A small sketch of a tree is expanded into a larger, more detailed tree.

2. Instance Normalization:

This step ensures that the brightness and contrast of the image are adjusted at every stage so that no one feature (e.g., light or dark areas) dominates the output.

3. ReLU Activation:

After upsampling and normalizing, this adds non-linearity to the process, helping the generator learn complex details like textures and patterns (e.g., making snowflakes look realistic on a winter photo).

Everyday Analogy for Upsampling

Imagine you are baking a pizza: When you stretch the pizza dough (upsampling), you're making it larger. But simply stretching it isn't enough; you also need to add toppings (details) to make it look and taste like a proper pizza! Similarly, the upsampling block expands the image and adds meaningful features like edges, colors, and textures to ensure the output looks natural.

Summary

- An upsampling block:

Increases the size of the compressed image (Transpose Convolution). Normalizes each step to maintain consistent brightness and contrast (Instance Normalization). Adds complexity with ReLU to capture fine details like patterns and textures. By combining

multiple upsampling blocks, the generator creates a full-sized, realistic image that matches the desired style or domain (e.g., turning a winter photo into a summer photo).

```
In [11]: def upsample(filters, size, apply_dropout=False):

    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.Conv2DTranspose(filters, size, strides=2,
                                      padding='same',
                                      kernel_initializer=initializer,
                                      use_bias=False))

    result.add(tfa.layers.InstanceNormalization(gamma_initializer=gamma_init))

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

    result.add(layers.ReLU())

    return result
```

Assembling the Generator

The generator in a CycleGAN is like an artist that takes an input image (e.g., a winter landscape) and creates an output image in a different style (e.g., a summer version of the same scene). To do this, it goes through three main stages:

1. Encoding the input image using downsampling blocks.
2. Transforming the image using ResNet blocks.
3. Reconstructing the output image using upsampling blocks.
4. Encoding the Input Image (Downsampling)

This is the first step, where the generator compresses the image to capture its most important features while discarding unnecessary details.

Imagine taking a large, high-resolution photo and shrinking it into a small thumbnail that still conveys the main idea (e.g., you can recognize that it's a landscape or a portrait, even at a smaller size).

The goal here is to reduce the image dimensions so that the model can focus on extracting essential patterns, like edges, textures, and colors.

2. Transforming the Image (ResNet Blocks)

This is where the "magic" happens. The generator applies residual network (ResNet) blocks to modify the compressed image and transform it into the desired style (e.g., adding bright greens and sunny tones to a winter landscape).

What is a ResNet Block?

Think of it as a mini transformation unit with a skip connection. This skip connection is like taking a shortcut by directly adding the input to the output of the block.

Why Skip Connections?

Skip connections help solve the vanishing gradient problem, which can occur when training deep neural networks. Imagine writing an essay and making edits to it. If you keep the original sentences (input) alongside your edits (output), you'll always retain the core meaning of the essay. That's what skip connections do—they ensure that the important information from earlier layers isn't lost during the transformation. The ResNet blocks allow the generator to make detailed, fine-grained changes to the image without forgetting the important features captured in the encoding step.

3. Reconstructing the Output Image (Upsampling)

After transforming the image, the generator needs to rebuild it into its original size using upsampling blocks. This step ensures that the output image looks complete and realistic in its new style.

Think of an artist sketching a rough outline (encoded image) and then gradually adding layers of color, shading, and details to create the final masterpiece (output image). The upsampling blocks expand the compressed image and add all the necessary details, like colors, textures, and shapes, to make the output image look natural.

How ResNet Blocks Connect Downsampling and Upsampling

The ResNet blocks in the middle act as a bridge between the downsampling and upsampling stages. They concatenate or link the compressed features (from downsampling) with the expanded features (from upsampling) in a symmetrical way.

Imagine building a bridge between two mountains (downsampling on one side and upsampling on the other). The ResNet blocks ensure that the information flows smoothly across the bridge, so no important details are left behind.

Key Benefits of the Generator Design

- Encoding: Focuses on capturing the most important features of the input image.
- ResNet Blocks: Ensures smooth and meaningful transformations with skip connections.
- Upsampling: Reconstructs the image to its original size with added details in the desired style.

The generator works like a well-organized assembly line:

- It starts by compressing the input image into a simplified form (downsampling).
- It then applies transformations using ResNet blocks to change the image's style.
- Finally, it reconstructs the full-sized output image with realistic details (upsampling).

This thoughtful design ensures that the generator can create high-quality, style-translated images while maintaining important features of the original input.

```
In [12]: 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])

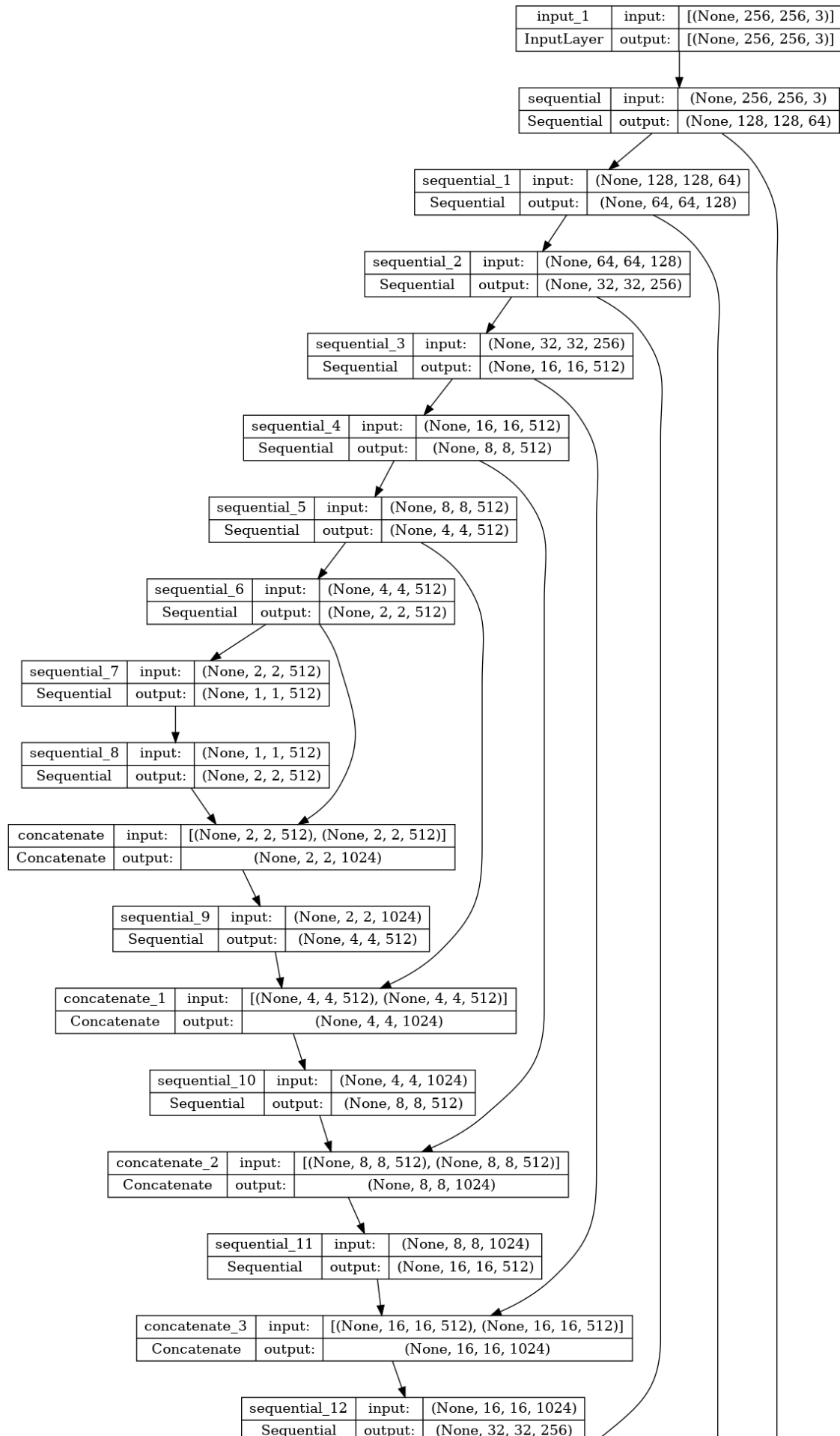
x = last(x)

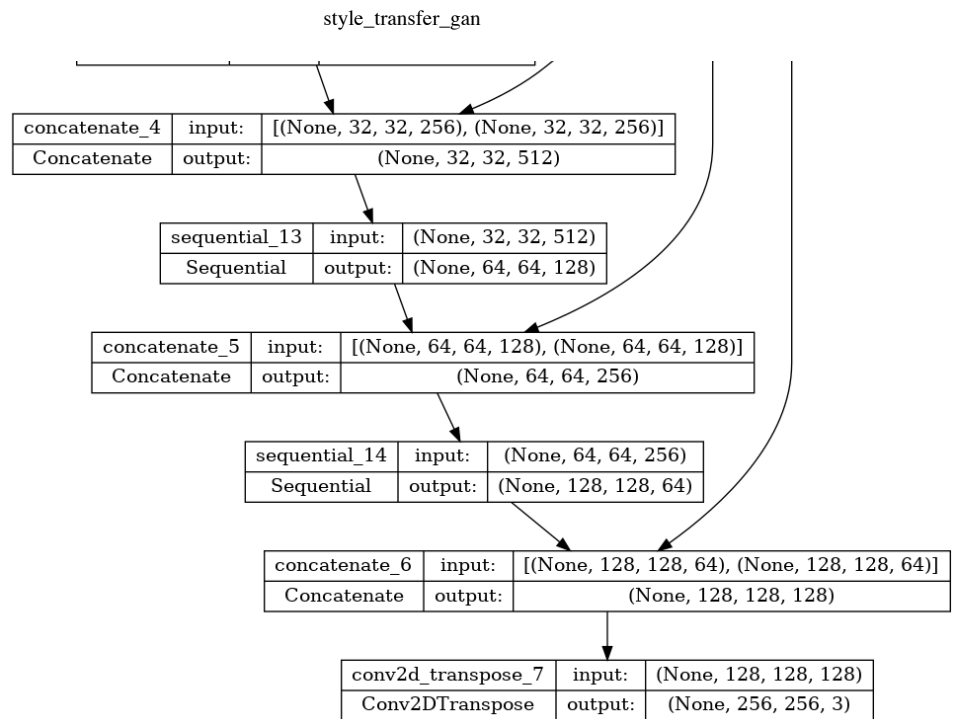
return keras.Model(inputs=inputs, outputs=x)
```

Now we're visualizing our Generator model's architecture using the `plot_model` function from `keras.utils.vis_utils`.

```
In [13]: gen = Generator()
plot_model(gen, show_shapes=True, show_layer_names=True)
```


Out [13]:





By looking at the visualization, we can see very clearly how the skip connections operation works and that all the concatenations are done symmetrically.

Also, it is to be noted that it's a fairly complicated architecture and one Generator alone has 54M parameters to train!

Building the Discriminator

In the CycleGAN architecture, the discriminator is like a detective. Its job is to look at an image and decide whether it's real (a genuine image from the dataset) or fake (an image created by the generator). It takes an input image of size 256×256 and gives a single number as output—a "realness" score.

Working

1. Input to the Discriminator

The input to the discriminator is a color image with dimensions 256×256 pixels. This can be either:

1. A real image: An actual image from the training dataset.
2. A fake image: An image created by the generator model.

The discriminator's task is to classify these images correctly. If the image is real, it should output "real"; if it's fake, it should output "fake."

2. The Layers in the Discriminator

The discriminator is built using a combination of three types of layers:

1. Convolutional Layers: These extract features like edges, textures, and patterns from the image.
2. Instance Normalization: This is used to standardize the feature maps, ensuring stability during training.
3. Leaky ReLU Activation: This introduces non-linearity, which helps the model learn complex patterns while preventing issues like the "dying neuron" problem.

Why Instance Normalization?

Instance normalization is often used in generator models, but it works well in discriminator models too. It ensures the model focuses on the structure and patterns in individual images rather than being influenced by the overall dataset.

3. How the Layers are Stacked

The discriminator uses 4 sets of Convolution-InstanceNorm-LeakyReLU layers, with increasing filter sizes in each layer.

Layer 1:

Filters: 64 filters Purpose: Extracts basic features like edges and small patterns. Think of this as a magnifying glass zooming in on the image's simplest details.

Layer 2:

Filters: 128 filters Purpose: Captures more detailed patterns, like textures and shapes. Like identifying the texture of grass or the outline of a tree.

Layer 3:

Filters: 256 filters Purpose: Detects more abstract features, like clusters of objects or patterns in color. Recognizing whether the image represents a forest or a cityscape.

Layer 4:

Filters: 512 filters Purpose: Focuses on the most complex features and relationships in the image. Spotting finer details like reflections in water or intricate patterns in a building.

4. The Output Layer

After the 4 main layers, the discriminator applies a final convolutional layer to produce a 1-dimensional output.

What does this mean?

The output is a score indicating whether the image is real or fake.

1. Score close to 1: The image is likely real.
2. Score close to 0: The image is likely fake.

Imagine a teacher grading an assignment. A perfect score is 10/10 means the work is real, while a failing score means it's fake.

How Does It Work in Practice?

The discriminator is shown a mix of real images from the dataset and fake images generated by the generator. It learns to classify these images by analyzing the features in each one.

- During training:

If the discriminator correctly identifies a fake image, it's rewarded. If it misclassifies an image, it's penalized and adjusts its parameters to improve.

Why Does the Discriminator Need to Be Strong?

The discriminator pushes the generator to improve. If the discriminator is very good at spotting fakes, the generator will have to work harder to create realistic images. This "competition" drives both models to get better over time.

Key Takeaways

1. Convolutional Layers: Extract features from the input image, layer by layer.
2. Instance Normalization: Ensures consistent feature extraction by normalizing individual images.
3. Leaky ReLU Activation: Adds non-linearity, helping the model learn complex patterns.

4. Final Output: A score that indicates whether the image is real or fake.
5. The discriminator plays a critical role in the GAN system, ensuring the generator keeps improving until it produces images that are indistinguishable from real ones!

```
In [14]: 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 = tf.layers.InstanceNormalization(gamma_initializer=gamma_init)(conv)

    leaky_relu = layers.LeakyReLU()(norm1)

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

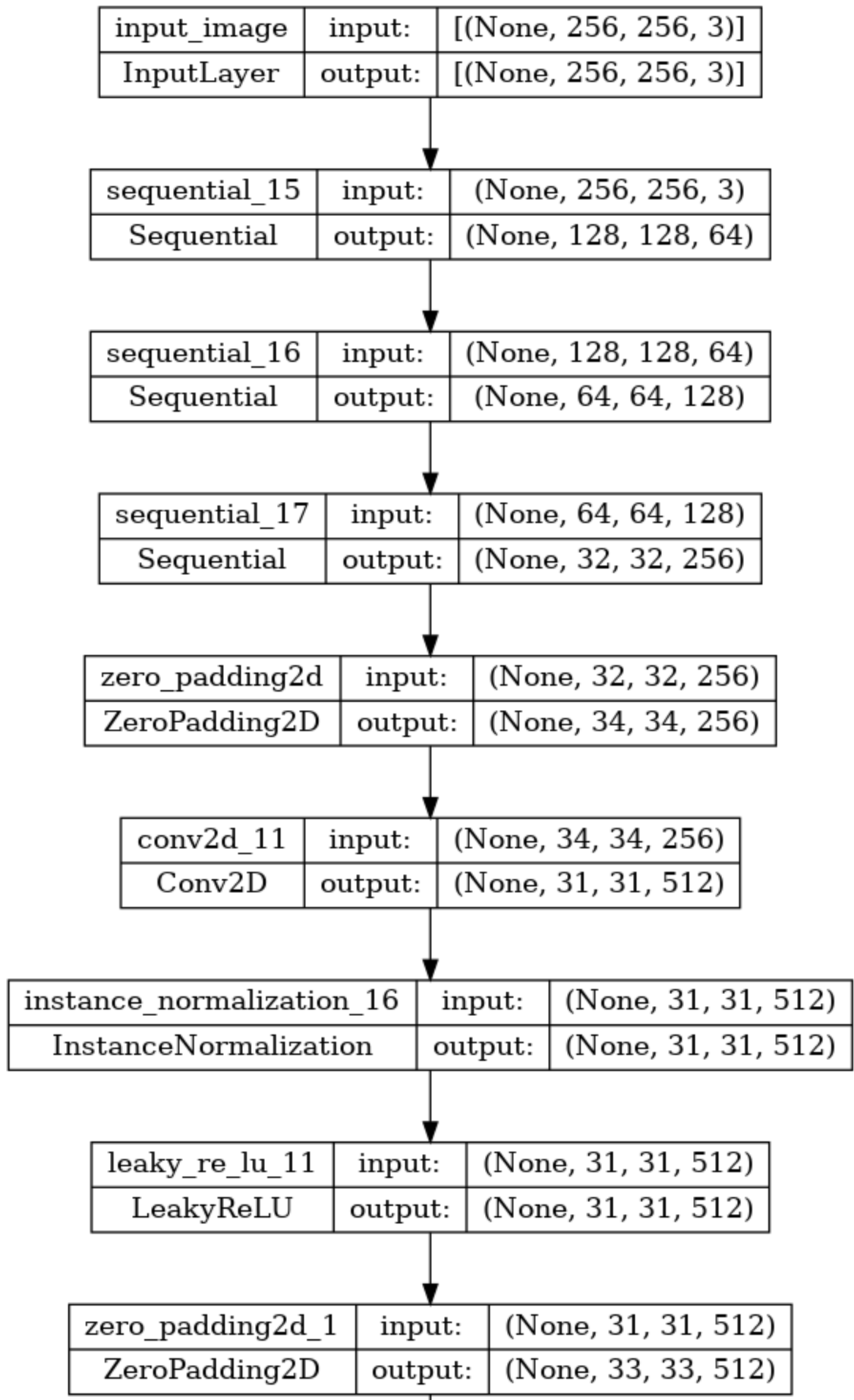
    last = layers.Conv2D(1, 4, strides=1,
                        kernel_initializer=initializer)(zero_pad2) # (bs, 30, 30, 1)


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

Similarly, we can visualize the architecture of the Discriminator model:

```
In [15]: disc = Discriminator()
plot_model(disc, show_shapes=True, show_layer_names=True)
```

Out[15]:





conv2d_12	input:	(None, 33, 33, 512)
Conv2D	output:	(None, 30, 30, 1)

Why Does the Discriminator Look Simpler?

At first glance, the discriminator model might seem less complicated than the generator model. That's because the discriminator's primary job is downsampling—it reduces the image's size step by step to extract meaningful features and decide whether the image is real or fake.

While it may appear simpler, if we look at its internal structure by running the `disc.summary()` command (in programming tools like TensorFlow or PyTorch), we'll notice something which is, the discriminator has over 2 million trainable parameters.

What Are Parameters?

A parameter in a neural network is like a tunable setting. These parameters are adjusted during training to help the model make better predictions.

In the discriminator model: Parameters are mainly the weights and biases in the convolutional layers that extract features from images.

Why so many parameters?

Even though the discriminator focuses on downsampling, it uses multiple filters (with increasing sizes) in each convolutional layer, resulting in millions of parameters. Imagine we're training to recognize counterfeit currency. Even if the task is "just identifying fakes," we need to learn to notice millions of tiny details—like the texture of the paper, the printing technique, holograms, and watermarks. Similarly, the discriminator needs millions of parameters to learn the subtle features that distinguish real images from fake ones.

Why Does Training the Discriminator Take Effort?

Even though the discriminator has fewer layers compared to the generator, training it isn't easy, because:

1. The Discriminator's Job Is Critical

The discriminator is like the critic in a GAN (Generative Adversarial Network). It needs to be very sharp and detail-oriented to spot even the slightest imperfections in the fake images generated by the generator. If the discriminator isn't strong enough, the generator won't improve because it won't face any real challenges.

2. Downsampling Isn't Simple

Downsampling may sound straightforward (reducing the image size), but the model is actually learning to preserve important details while compressing the image. This involves analyzing patterns, textures, and fine details. Imagine shrinking a high-resolution photo to a small thumbnail while still being able to identify what's in the image. It's a complex task.

3. Millions of Parameters

With over 2 million parameters, the discriminator has to adjust all these weights and biases through training. This requires time, data, and computational power. Training the discriminator is like teaching a detective to recognize counterfeit money. Even though their job is "just identifying fake bills," they need to study millions of fine details like ink patterns, paper texture, and watermarks.

How Does This Compare to the Generator?

The generator might seem more complicated because it involves downsampling, transforming, and upsampling, which means it's actively creating new images.

The generator tries to "fool" the discriminator by learning how to create images that look real. The discriminator, on the other hand, simply learns to spot fakes. While this might seem easier, the volume of details it needs to learn makes it equally challenging.

Why Both Models Are Important

The Generator and Discriminator Are Partners in a Game The generator creates fake images, while the discriminator evaluates them. This process repeats until the generator becomes so good that its images are indistinguishable from real ones. Imagine a chef creating a new dish (generator) and a food critic (discriminator) tasting it. Over time, the chef improves their recipe based on the critic's feedback.

Why the Discriminator Must Be Strong

If the discriminator isn't trained well, it will fail to catch fake images, and the generator won't learn anything meaningful. A well-trained discriminator forces the generator to get better.

- Now that we have both the generator and discriminator ready, it's time to combine them into the CycleGAN model.
- The CycleGAN works by having two generators and two discriminators, working together in a loop. This allows the model to translate images between two different styles e.g., turning photos into paintings and back into photos.

Building the CycleGAN Model

When working with a CycleGAN, we're essentially creating a system that can translate between two different types of images—for example, converting photos into Monet-style paintings and vice versa.

But how do we ensure the translation is accurate and meaningful?

That's where forward-backward consistency comes in.

Forward-Backward Consistency in Simple Terms

If we take a photo and convert it into a Monet-style painting, we should be able to take that Monet-style painting and convert it back into the original photo. Similarly, if we take a Monet painting and turn it into a photo, we should be able to turn it back into the same Monet painting. This process of going forward and then backward ensures the model doesn't just generate random transformations. Instead, the transformations make sense and are consistent.

Why Do We Need Two Generators?

To enforce this forward-backward consistency, we need two generators:

1. Photo to Monet Generator (`monet_generator`)

This takes a normal photo and transforms it into a painting in the style of Monet. Imagine taking a photo of a garden and turning it into an artistic Monet-like painting.

2. Monet to Photo Generator (photo_generator)

This does the opposite. It takes a Monet painting and turns it into something that looks like a real photo. A Monet painting of a lily pond gets transformed into a lifelike photograph of an actual pond. Each generator is responsible for one specific direction of transformation.

Why Do We Need Two Discriminators?

For each generator, we need a discriminator to check the quality of its work. These discriminators serve as critics, evaluating whether the output looks real or fake.

1. Monet Discriminator (monet_discriminator)

This evaluates whether a Monet-style image is real (from the dataset) or fake (generated by the photo-to-Monet generator). Imagine it's a Monet art critic saying, "Does this look like a genuine Monet painting?"

2. Photo Discriminator (photo_discriminator)

This evaluates whether a photo is real (from the dataset) or fake (generated by the Monet-to-photo generator). It's like a photography critic saying, "Does this look like a real photo?"

- Two Generators: One transforms photos into Monet-style paintings, and the other transforms Monet paintings into photos.
- Two Discriminators: One judges Monet-style images, and the other judges photo-style images.
- This means our CycleGAN model will have four components:
 1. monet_generator: Turns photos into Monet paintings.
 2. photo_generator: Turns Monet paintings into photos.
 3. monet_discriminator: Judges if Monet-style images are real or fake.
 4. photo_discriminator: Judges if photo-style images are real or fake.

Imagine we're running an art competition between two painters: Painter A specializes in creating Monet-style paintings from photos. Painter B specializes in creating realistic photos from Monet paintings.

we also hire two judges: Judge 1 is an expert in Monet's work and evaluates Painter A's Monet-style paintings. Judge 2 is a photography expert and evaluates Painter B's

realistic photos. Each painter is trying to fool their respective judge into believing their work is authentic, while the judges get better at spotting fakes. This dynamic competition improves both the painters (generators) and the judges (discriminators) over time.

```
In [16]: with strategy.scope():  
  
    monet_generator = Generator() # transforms photos to Monet-esque paintin  
    photo_generator = Generator() # transforms Monet paintings to be more li  
  
    monet_discriminator = Discriminator() # differentiates real Monet painti  
    photo_discriminator = Discriminator() # differentiates real photos and g
```

All 4 networks that we defined here will be used to compile a CycleGAN.

What is the CycleGan class?

It's a subclass of `tf.keras.Model`, which means it inherits all the handy functionalities of Keras models, like `.fit()` for training. This structure makes it easier to organize everything related to the

What does the CycleGan model need?

It needs 4 optimizers because there are 4 networks in the CycleGAN:

1. Photo-to-Monet generator
 2. Monet-to-Photo generator
 3. Monet discriminator
 4. Photo discriminator
- It also needs 4 loss functions, one for each network, so we can guide their learning properly.

What happens during a training step (train_step)?

A real photo is passed through the Photo-to-Monet generator, turning it into a Monet-style painting. This Monet-style painting is then passed back through the Monet-to-Photo generator, transforming it back into a photo. Similarly, a real Monet painting is passed through the Monet-to-Photo generator, turning it into a photo, which is then passed back through the Photo-to-Monet generator, transforming it back into a Monet

painting. This forward and backward process enforces the cycle consistency we talked about earlier.

What is the cycle consistency loss?

The idea here is to measure how much information gets lost during these transformations. Specifically, the model compares the original input (real photo or Monet painting) with the cycled output (after two transformations). The loss is the absolute difference between the original image and the cycled image. The smaller this difference, the better the model is at retaining the original image's features.

Why is this important?

Without cycle consistency loss, the generators might just create random images that don't make sense. This ensures that the transformations stay meaningful and accurate.

```
In [17]: 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

@tf.function
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,
                                                real_photo, cycled_photo)

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

        # 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
}

```

What is @tf.function?

It's a decorator in TensorFlow, meaning you place it on top of a function to modify its behavior.

When you use @tf.function?

TensorFlow converts the Python function into a TensorFlow computation graph. This is like creating a blueprint that shows how data flows and what operations are applied, step by step.

Why is it useful?

A computation graph is much faster because it removes the overhead of repeatedly interpreting Python code during training. Instead, TensorFlow directly executes the graph.

By storing the flow of operations and variables, it avoids reinitializing computations every time the function is called, which can significantly speed up training.

How does it help in training models?

Imagine we're running a repetitive task, like adding numbers. Without the graph, TensorFlow reinterprets the Python code every single time we add two numbers, which is slow.

With `@tf.function`, TensorFlow pre-compiles the entire addition process into a single, efficient workflow, and just runs it repeatedly, saving time. Think of it as baking cookies. Normally, we'd read the recipe, measure ingredients, and mix every single time you bake (like executing Python functions repeatedly).

Using `@tf.function` is like creating a pre-made cookie dough. Now you just cut the shapes and bake, making the process much faster because the setup work is already done.

Defining Loss Functions

What is the discriminator_loss?

The discriminator is responsible for classifying images as real or fake (generated by the generator). A perfect discriminator should output:

1. All 1s for a real image, meaning it recognizes it as real.
2. All 0s for a fake image, meaning it identifies it as fake.

The `discriminator_loss` measures how far the discriminator's predictions are from these ideal outputs (1s and 0s).

How is the loss calculated?

To quantify this difference, Binary Cross Entropy Loss is used. Binary Cross Entropy compares the predicted values (output of the discriminator) with the expected values (1s for real, 0s for fake). Example: If the discriminator predicts 0.8 for a real image, the loss penalizes it for not predicting 1.0. If it predicts 0.2 for a fake image, the loss penalizes it for not predicting 0.0.

Why divide the loss by half?

The CycleGAN paper recommends halving the discriminator loss during training. This is because the discriminator, being a simpler task (real vs. fake), can quickly outperform the generator. If the discriminator improves too quickly, the generator will struggle to learn how to create better fake images. By slowing down the updates to the

discriminator (dividing its loss by 2), we maintain a balance between the two networks, allowing both to improve together over time. Imagine a teacher (discriminator) and a student (generator) in a classroom. The teacher's job is to grade the student's work (real or fake images). If the teacher is too harsh or improves too quickly, the student won't have time to learn and will give up.

To balance this, the teacher is asked to grade less harshly (divide loss by half) so the student has a fair chance to catch up and improve their work.

This balance is critical for the success of the CycleGAN training, ensuring both the generator and discriminator evolve together.

```
In [18]: with strategy.scope():

    def discriminator_loss(real, generated):
        real_loss = tf.keras.losses.BinaryCrossentropy(from_logits=True, reduction=tf.keras.losses.Reduction.NONE)(real_logits, real_labels)
        generated_loss = tf.keras.losses.BinaryCrossentropy(from_logits=True, reduction=tf.keras.losses.Reduction.NONE)(generated_logits, generated_labels)
        total_disc_loss = real_loss + generated_loss

        return total_disc_loss * 0.5
```

A perfect generator should fool the discriminator by generating images that the discriminator predicts as real or all 1s. Thus, the `generator_loss` compares the discriminator's prediction for a generated image to a matrix of 1s.

```
In [19]: with strategy.scope():

    def generator_loss(generated):
        return tf.keras.losses.BinaryCrossentropy(from_logits=True, reduction=tf.keras.losses.Reduction.NONE)(generated_logits, generated_labels)
```

Here is the `calc_cycle_loss`, or the Cycle Consistency Loss, which is the average of the absolute differences between a real image and a cycled (twice transformed) image.

```
In [20]: #We want our original photo and the twice transformed photo to be similar to each other
#Thus, we can calculate the cycle consistency loss by finding the average of the absolute differences between a real image and a cycled (twice transformed) image.

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
```

Lastly, we have the `identity_loss`. It penalizes large difference between a real image and an image generated by the Generator based on the real image, as a perfect Generator should not have disparate input and output.


```
In [21]: 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
```

Model Training

we will now compile and train a CycleGAN. Since our networks have too many parameters (one generator alone has 54M and a discriminator has 2M) and Skills Network Labs currently doesn't have any GPUs available, it will take hours to train a sufficient number of epochs with a CPU such that the model can do a decent job in transferring image styles.

Therefore, we will train the model for **one epoch** in this lab, which includes 300 iterations. However, we can also choose to run this notebook locally where we can use GPUs or TPUs, then we are free to increase the number of epochs!

Training the CycleGAN

As mentioned before, a CycleGAN is compiled with 4 loss functions and 4 optimizers. Now we're define the optimizer for each network and then we're compiling the model.

```
In [22]: 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)
```

```
In [23]: 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
    )
```

we may remove the "#" signs below to train the model! (Since this is a fairly large model and the Skills Network Labs environment currently only has CPUs, training the large model might cause the issue of a dead kernel. **You may also consider to skip the cell below as we are going to provide you with a pre-trained model in the next section!**

```
In [ ]: # history = cycle_gan_model.fit(  
#       tf.data.Dataset.zip((monet_ds, photo_ds)),  
#       epochs=1)
```

Now that we had a taste of training the mighty CycleGAN, remember that our original task is to transform photos to Monet-esque paintings and we would need a fully-trained `monet_generator` to do the job.

To save us from having to wait for hours in the Skills Network Labs environment, we had trained a CycleGAN beforehand with all the same configurations we had above for 25 epochs. Now we just have to import the weights of the **Monet Generator network** of the pre-trained CycleGAN to help us complete the Style Transfer task!

Visualize our Monet-esque photos

Loading Pre-trained Weights

Before we load the trained `monet_generator`, the cognitive class has presented us a screenshot of the CycleGAN's training history. Even though only the monet generator is needed for the style transfer task, but we can see that the losses of all 4 networks: monet generator, monet discriminator, photo generator, and photo discriminated, were being optimized.

```

history = cycle_gan_model.fit(
    tf.data.Dataset.zip((monet_ds, photo_ds)),
    epochs=25,
    callbacks=[time_callback, cp_callback]
)

Epoch 1/25
300/unknown - 139s 463ms/step - monet_gen_loss: 3.7744 - photo_gen_loss: 3.8493 - monet_disc_loss: 0.6441 - photo_disc_loss: 0.6388
Epoch 1: saving model to training_25/cp.ckpt
300/300 [=====] - 145s 484ms/step - monet_gen_loss: 3.7726 - photo_gen_loss: 3.8465 - monet_disc_loss: 0.6448 - photo_disc_loss: 0.6397
Epoch 2/25
300/300 [=====] - ETA: 0s - monet_gen_loss: 3.6306 - photo_gen_loss: 3.7895 - monet_disc_loss: 0.6522 - photo_disc_loss: 0.5978
Epoch 2: saving model to training_25/cp.ckpt
300/300 [=====] - 146s 485ms/step - monet_gen_loss: 3.6284 - photo_gen_loss: 3.7875 - monet_disc_loss: 0.6525 - photo_disc_loss: 0.5982
Epoch 3/25
300/300 [=====] - ETA: 0s - monet_gen_loss: 3.5084 - photo_gen_loss: 3.6445 - monet_disc_loss: 0.6380 - photo_disc_loss: 0.6156
Epoch 3: saving model to training_25/cp.ckpt
300/300 [=====] - 146s 485ms/step - monet_gen_loss: 3.5060 - photo_gen_loss: 3.6419 - monet_disc_loss: 0.6384 - photo_disc_loss: 0.6160
Epoch 4/25
300/300 [=====] - ETA: 0s - monet_gen_loss: 3.4001 - photo_gen_loss: 3.5082 - monet_disc_loss: 0.6225 - photo_disc_loss: 0.6093
Epoch 4: saving model to training_25/cp.ckpt
300/300 [=====] - 145s 485ms/step - monet_gen_loss: 3.3975 - photo_gen_loss: 3.5060 - monet_disc_loss: 0.6230 - photo_disc_loss: 0.6097
Epoch 5/25
300/300 [=====] - ETA: 0s - monet_gen_loss: 3.2985 - photo_gen_loss: 3.4214 - monet_disc_loss: 0.6155 - photo_disc_loss: 0.5957
Epoch 5: saving model to training_25/cp.ckpt
300/300 [=====] - 146s 485ms/step - monet_gen_loss: 3.2961 - photo_gen_loss: 3.4191 - monet_disc_loss: 0.6161 - photo_disc_loss: 0.5959
Epoch 6/25
300/300 [=====] - ETA: 0s - monet_gen_loss: 3.2463 - photo_gen_loss: 3.3393 - monet_disc_loss: 0.6135 - photo_disc_loss: 0.6015
Epoch 6: saving model to training_25/cp.ckpt
300/300 [=====] - 145s 485ms/step - monet_gen_loss: 3.2442 - photo_gen_loss: 3.3371 - monet_disc_loss: 0.6141 - photo_disc_loss: 0.6021
Epoch 7/25
300/300 [=====] - ETA: 0s - monet_gen_loss: 3.2296 - photo_gen_loss: 3.3132 - monet_disc_loss: 0.6261 - photo_disc_loss: 0.6147
Epoch 7: saving model to training_25/cp.ckpt
300/300 [=====] - 145s 484ms/step - monet_gen_loss: 3.2279 - photo_gen_loss: 3.3112 - monet_disc_loss: 0.6266 - photo_disc_loss: 0.6150
Epoch 8/25
300/300 [=====] - ETA: 0s - monet_gen_loss: 3.2010 - photo_gen_loss: 3.2859 - monet_disc_loss: 0.6123 - photo_disc_loss: 0.6068
Epoch 8: saving model to training_25/cp.ckpt
300/300 [=====] - 145s 484ms/step - monet_gen_loss: 3.1999 - photo_gen_loss: 3.2844 - monet_disc_loss: 0.6130 - photo_disc_loss: 0.6071
Epoch 9/25
300/300 [=====] - ETA: 0s - monet_gen_loss: 3.1742 - photo_gen_loss: 3.2493 - monet_disc_loss: 0.6160 - photo_disc_loss: 0.6120
Epoch 9: saving model to training_25/cp.ckpt
300/300 [=====] - 145s 484ms/step - monet_gen_loss: 3.1731 - photo_gen_loss: 3.2479 - monet_disc_loss: 0.6164 - photo_disc_loss: 0.6121
Epoch 10/25
300/300 [=====] - ETA: 0s - monet_gen_loss: 3.1098 - photo_gen_loss: 3.1929 - monet_disc_loss: 0.6175 - photo_disc_loss: 0.6086

```

Downloading the weights of `monet_generator` :

```
In [24]: await skillsnetwork.prepare("https://cf-courses-data.s3.us.cloud-object-storage.com/output_bucket/monet_generator_50_epochs.tgz",
                                     overwrite=True)
```

```

Downloading monet_generator_50_epochs.tgz: 0%|          | 0/202293808 [00:00<?, ?it/s]
0%|          | 0/14 [00:00<?, ?it/s]
Saved to '.'
```

Creating the `monet_generator_model` using the `load_model` API from Keras:

```
In [25]: # Ignore the warnings of this cell, as we don't need the compilation info to
monet_generator_model = tf.keras.models.load_model("monet_generator_50_epochs.h5")
```

```

WARNING:root:The given value for groups will be overwritten.
WARNING:root:The given value for groups will be overwritten.
WARNING:root:The given value for groups will be overwritten.
WARNING:root:The given value for groups will be overwritten.
WARNING:root:The given value for groups will be overwritten.
WARNING:root:The given value for groups will be overwritten.
WARNING:root:The given value for groups will be overwritten.
WARNING:root:The given value for groups will be overwritten.
WARNING:root:The given value for groups will be overwritten.
WARNING:root:The given value for groups will be overwritten.
WARNING:root:The given value for groups will be overwritten.
WARNING:root:The given value for groups will be overwritten.
WARNING:root:The given value for groups will be overwritten.
WARNING:tensorflow:No training configuration found in save file, so the model
was *not* compiled. Compile it manually.

```

WARNING:tensorflow:No training configuration found in save file, so the model was *not* compiled. Compile it manually.

Visualizing Style Transfer Output

Finally! We will visualize the style transfer output produced by

`monet_generator_model`. We take 5 sample images that are photos of beautiful landscapes in the original dataset and feed them to the model.

```
In [26]: for i in range(5):

    # randomly draw a photo from PHOTO_FILENAMES
    rand_ind = random.randint(0,299)
    img_path = PHOTO_FILENAMES[rand_ind]
    img = decode_image(img_path)
    img = tf.expand_dims(img, axis=0)

    prediction = monet_generator_model(img, training=False)[0].numpy()
    prediction = (prediction * 127.5 + 127.5).astype(np.uint8)
    img = (img[0] * 127.5 + 127.5).numpy().astype(np.uint8)

    fig = plt.figure()
    fig.add_subplot(1,2,1)
    if i == 0:
        plt.title("Input Photos")
    plt.imshow(img)
    plt.axis("off")

    fig.add_subplot(1,2,2)
    if i == 0:
        plt.title("Monet-esque Paintings")
    plt.imshow(prediction)
    plt.axis("off")
```

Input Photos



Monet-esque Paintings





