Deep Learning course project - 3

Title : Style transfer using cycle GAN for MRI images

Sl.no	Name	Roll no.
1	G. Tanmay	B210753EC
2	C. Nithin Bhavan	B210705EC
3	Ch. Karthik	B210711EC

Importing Libraries

In [2]:
```
%pip install git+https://github.com/tensorflow/docs
```

```
Collecting git+https://github.com/tensorflow/docs
  Cloning https://github.com/tensorflow/docs to /tmp/pip-req-build-q0vmnasl
  Running command git clone --filter=blob:none --quiet https://github.com/tensorflow/docs /tmp/pip-r
eq-build-q0vmnasl
  Resolved https://github.com/tensorflow/docs to commit 4d512c2d7c40d69fcb842978aeaa136e19abe2bb
  Preparing metadata (setup.py) ... done
Requirement already satisfied: absl-py in /usr/lib/python3/dist-packages (from tensorflow-docs==202
3.5.26.9808) (0.15.0)
Requirement already satisfied: astor in ./dl/lib/python3.10/site-packages (from tensorflow-docs==202
3.5.26.9808) (0.8.1)
Requirement already satisfied: jinja2 in /usr/lib/python3/dist-packages (from tensorflow-docs==2023.
5.26.9808) (3.0.3)
Requirement already satisfied: nbformat in ./dl/lib/python3.10/site-packages (from tensorflow-docs==
2023.5.26.9808) (5.9.0)
Requirement already satisfied: protobuf>=3.12 in /usr/lib/python3/dist-packages (from tensorflow-doc
s==2023.5.26.9808) (3.12.4)
Requirement already satisfied: pyyaml in /usr/lib/python3/dist-packages (from tensorflow-docs==2023.
5.26.9808) (5.4.1)
Requirement already satisfied: jsonschema>=2.6 in ./dl/lib/python3.10/site-packages (from nbformat->
tensorflow-docs==2023.5.26.9808) (4.17.3)
Requirement already satisfied: traitlets>=5.1 in ./dl/lib/python3.10/site-packages (from nbformat->t
ensorflow-docs==2023.5.26.9808) (5.9.0)
Requirement already satisfied: jupyter-core in ./dl/lib/python3.10/site-packages (from nbformat->ten
sorflow-docs==2023.5.26.9808) (5.3.1)
Requirement already satisfied: fastjsonschema in ./dl/lib/python3.10/site-packages (from nbformat->t
ensorflow-docs==2023.5.26.9808) (2.17.1)
Requirement already satisfied: attrs>=17.4.0 in ./dl/lib/python3.10/site-packages (from jsonschema>=
2.6->nbformat->tensorflow-docs==2023.5.26.9808) (23.1.0)
Requirement already satisfied: pyrsistent!=0.17.0,!=0.17.1,!=0.17.2,>=0.14.0 in ./dl/lib/python3.10/
site-packages (from jsonschema>=2.6->nbformat->tensorflow-docs==2023.5.26.9808) (0.19.3)
Requirement already satisfied: platformdirs>=2.5 in /usr/lib/python3/dist-packages (from jupyter-cor
e->nbformat->tensorflow-docs==2023.5.26.9808) (2.5.1)
Note: you may need to restart the kernel to use updated packages.
```

In [3]:
```
# Library for image processing and visualization
from skimage.transform import resize
from PIL import Image
import imageio
import matplotlib.pyplot as plt
%matplotlib inline

# Library for handling file operations
import os

# Library for handling array operations
import numpy as np

# Library for handling deep learning
import tensorflow as tf
from keras.utils.vis_utils import plot_model
import tensorflow_addons as tfa
autotune = tf.data.AUTOTUNE

# Library for Miscellaneous tasks
import warnings

warnings.filterwarnings("ignore")
```

```
/home/nightfall/Workspace/dl/lib/python3.10/site-packages/tensorflow_addons/utils/tfa_eol_msg.py:23:
UserWarning:

TensorFlow Addons (TFA) has ended development and introduction of new features.
TFA has entered a minimal maintenance and release mode until a planned end of life in May 2024.
Please modify downstream libraries to take dependencies from other repositories in our TensorFlow co
mmunity (e.g. Keras, Keras-CV, and Keras-NLP).

For more information see: https://github.com/tensorflow/addons/issues/2807

  warnings.warn(
```

Dataset and its Discription :

The dataset used is a sample data(provided) and dataset prepared from BraTS2020 Dataset.

Kaggle Link:https://www.kaggle.com/datasets/awsaf49/brats20-dataset-training-validation/code

Here we took some good nummber of samples from T1 and T2 immages from the whole dataset.

T1-weighted MRI(T1) :

T1-weighted images provide good contrast between different tissues based on their intrinsic T1 relaxation times. This sequence is often used to highlight anatomical structures such as gray matter, white matter, and cerebrospinal fluid (CSF). T1-weighted images are typically useful for visualizing the overall structure of the brain and detecting abnormalities such as tumors.

T2-weighted MRI(T2) :

T2-weighted images provide good contrast between different tissues based on their intrinsic T2 relaxation times. This sequence is sensitive to changes in water content and is useful for highlighting areas of inflammation, edema, and some types of pathology. T2-weighted images are often used to complement T1-weighted images in the evaluation of brain tumors and other neurological conditions.

These are commonly used for brain tumor analysis, including segmentation, classification, and treatment planning. The availability of multiple MRI sequences helps radiologists and researchers gain a comprehensive understanding of the brain anatomy and pathology.

There are 43 images under T1 category and 46 images in T2 category

Data loading and preprocessing

In [4]:
```
# Path for images in Sample dataset provided
t1 = "Data/Tr1/TrainT1/"
t2 = "Data/Tr2/TrainT2/"

# Image size
img_height = 128
img_width = 128
```

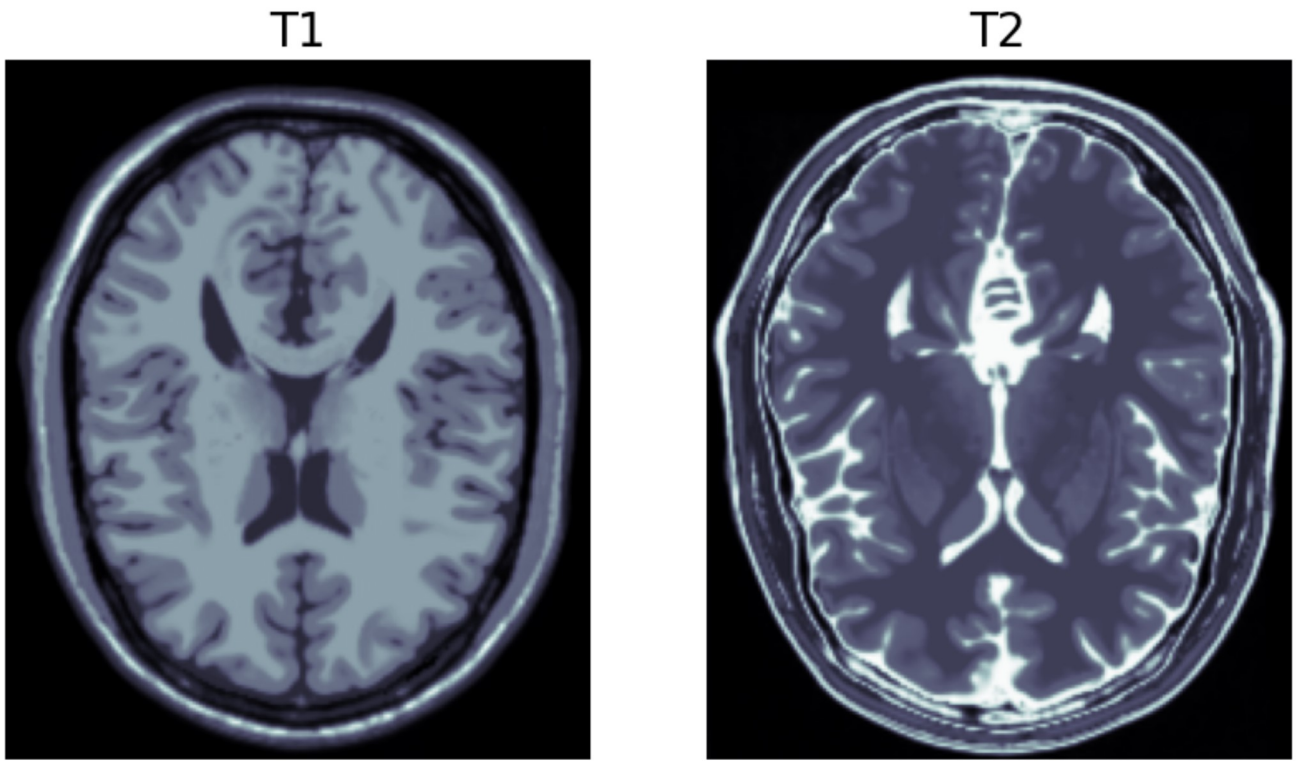
In [5]:
```
# list of images
t1_images = [t1 + i for i in os.listdir(t1)]
t2_images = [t2 + i for i in os.listdir(t2)]
```

In [6]:
```
# Images count for each category
print("\n\nTotal Images in each category:\n")
print(f"t1: {len(t1_images)}")
print(f"t2: {len(t2_images)}")
```

```
Total Images in each category:

t1: 43
t2: 46
```

In [7]:
```python
# Priting sample images from each category
print("\n\nSample Images from each category:\n")
plt.figure(figsize=(10, 10))
plt.subplot(1, 2, 1)
plt.imshow(plt.imread(t1_images[0]), cmap="bone")
plt.title("T1", fontsize=20)
plt.axis("off")
plt.subplot(1, 2, 2)
plt.imshow(plt.imread(t2_images[0]), cmap="bone")
plt.title("T2", fontsize=20)
plt.axis("off")
plt.show()
```

```
Sample Images from each category:
```



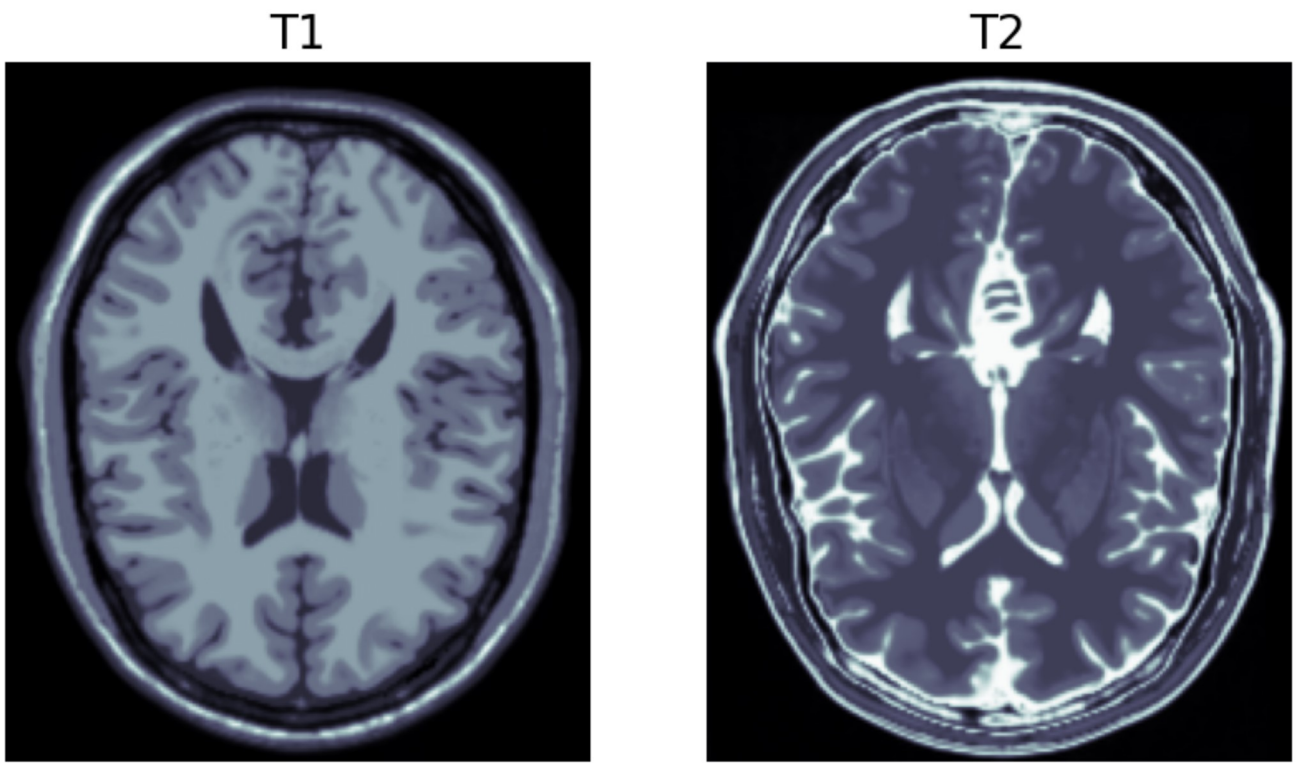
In [8]:
```python
# function to load images and resize
def load_images(img_list):
    img_arr = []
    for img in img_list:
        img = Image.open(img)
        img_arr.append(np.array(img))
    return np.array(img_arr)
```

In [9]:
```python
t1_images = load_images(t1_images)
t2_images = load_images(t2_images)
```

In [10]:
```python
# information of dataset
print("\n\nInformation of dataset:\n")
print(f"t1: {t1_images.shape}")
print(f"t2: {t2_images.shape}")
```

```
Information of dataset:

t1: (43, 217, 181)
t2: (46, 217, 181)
```

In [11]:
```
# Printing sample images from each category after preprocessing
plt.figure(figsize=(10, 10))
plt.subplot(1, 2, 1)
plt.imshow(t1_images[0], cmap="bone")
plt.title("T1", fontsize=20)
plt.axis("off")
plt.subplot(1, 2, 2)
plt.imshow(t2_images[0], cmap="bone")
plt.title("T2", fontsize=20)
plt.axis("off")
plt.show()
```



Data Preprocessing

In [12]:
```
# Function to normalize images
def normalize_img(img):
    img = tf.cast(img, dtype=tf.float32)
    return (img / 127.5) - 1.0
```

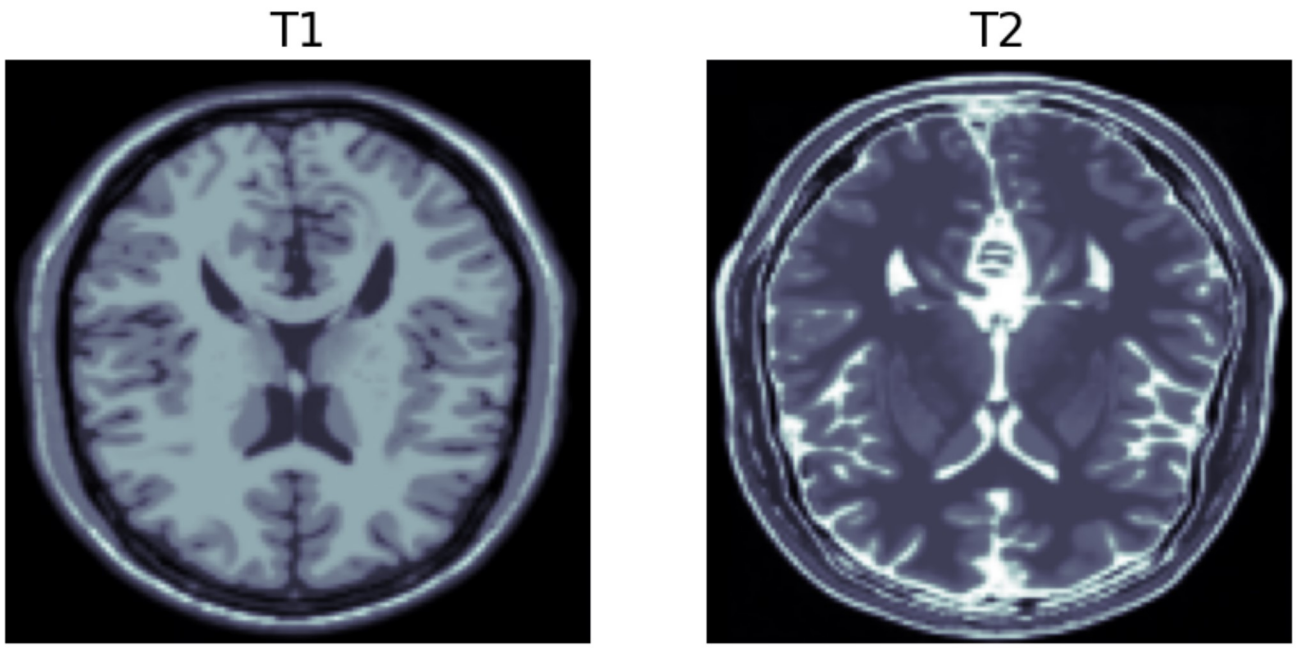
In [13]:
```
# Function to preprocess images
def preprocess_images(img):
    img = tf.image.random_flip_left_right(img)
    img = tf.image.resize(img, [*(img_height, img_width)])
    img = tf.image.random_crop(img, size=[*(img_height, img_width, 1)])
    img = normalize_img(img)
    return img
```

In [14]:
```
# reshaping array to format (height, width, channels) with channels = 1
t1_images = t1_images.reshape(
    t1_images.shape[0], t1_images.shape[1], t1_images.shape[2], 1
).astype("float32")
t2_images = t2_images.reshape(
    t2_images.shape[0], t2_images.shape[1], t2_images.shape[2], 1
).astype("float32")
```

In [16]:
```
t1_images = [preprocess_images(image) for image in t1_images]
t2_images = [preprocess_images(image) for image in t2_images]
```

In [17]:
```
# Printing sample images from each category after preprocessing
plt.figure(figsize=(10, 10))
```

```
plt.subplot(1, 2, 1)
plt.imshow(t1_images[0], cmap="bone")
plt.title("T1", fontsize=20)
plt.axis("off")
plt.subplot(1, 2, 2)
plt.imshow(t2_images[0], cmap="bone")
plt.title("T2", fontsize=20)
plt.axis("off")
plt.show()
```



Data Batch Creation

In [18]:
```
# defining batch and buffer size
buffer_size = 256
batch_size = 1
```

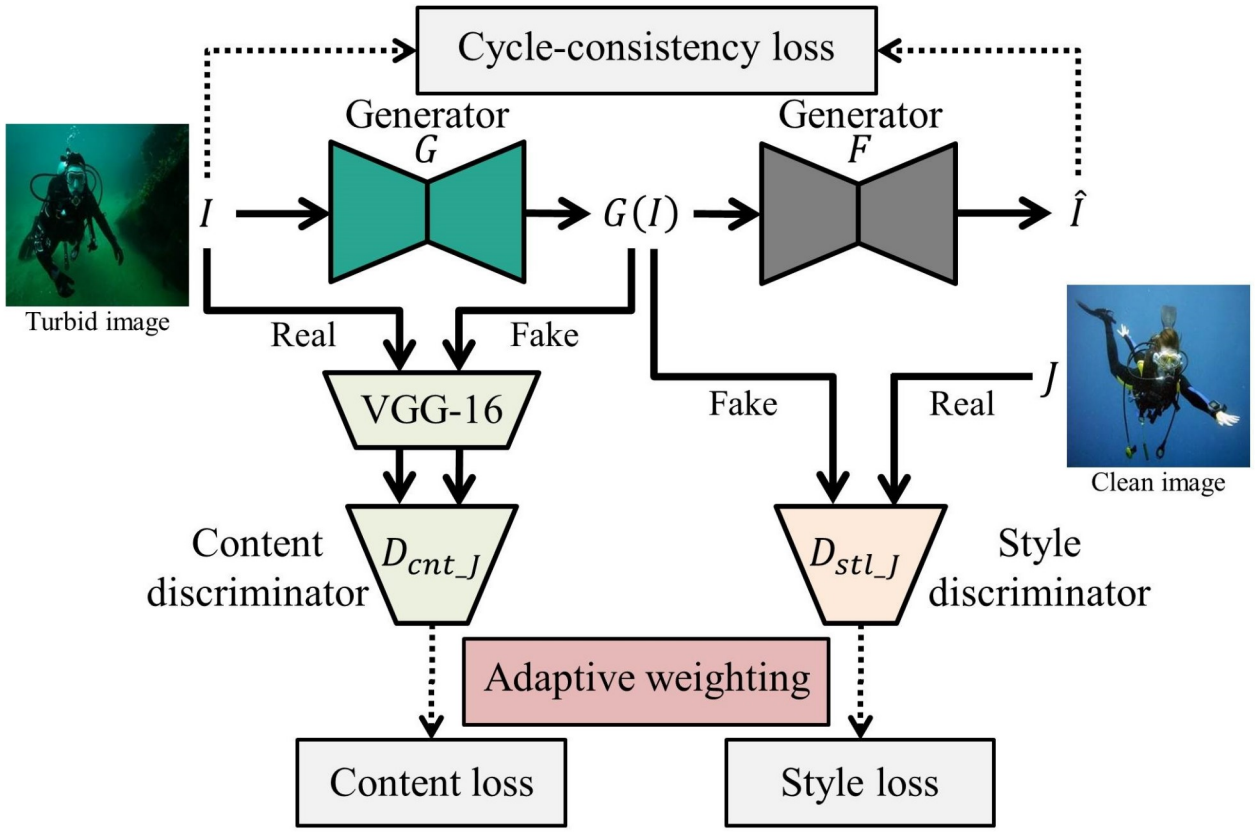
In [19]:
```
t1_images = (
    tf.data.Dataset.from_tensor_slices(t1_images)
    .cache()
    .shuffle(buffer_size)
    .batch(batch_size)
)
t2_images = (
    tf.data.Dataset.from_tensor_slices(t2_images)
    .cache()
    .shuffle(buffer_size)
    .batch(batch_size)
)
```

GAN and Cycle GAN's :

Generative Adversarial Networks (GANs) are deep learning architectures consisting of a generator and a discriminator network, trained in a competitive manner. The generator learns to create synthetic data samples, such as images, from random noise, while the discriminator learns to distinguish between real and fake samples. Through iterative training, the generator aims to produce samples that are indistinguishable from real data, while the discriminator improves at differentiating between real and fake samples.

Cycle Generative Adversarial Networks (CycleGANs) are a variant designed for unpaired image-to-image translation tasks. They consist of two generators and two discriminators, aiming to learn mappings between two domains without paired data. Unlike traditional GANs, CycleGANs enforce cycle consistency by reconstructing the original image from the translated image and vice versa, ensuring meaningful translations

between domains.



We first take an image input (x) and using the generator G to convert into the reconstructed image. Then we reverse this process from reconstructed image to original image using a generator F. Then we calculate the mean squared error loss between real and reconstructed image. The most important feature of this cycle_GAN is that it can do this image translation on an unpaired image where there is no relation exists between the input image and output image.

Creating Generator and Discriminator

In [20]:
```
# Weights initializer for the layers.
kernel_init = tf.keras.initializers.RandomNormal(mean=0.0, stddev=0.02)
# Gamma initializer for instance normalization.
gamma_init = tf.keras.initializers.RandomNormal(mean=0.0, stddev=0.02)
```

In [21]:
```
# Downsampling method
def downsample(filters, size, apply_norm=True):
    initializer = tf.random_normal_initializer(0.0, 0.02)
    result = tf.keras.Sequential()
    result.add(
        tf.keras.layers.Conv2D(
            filters,
            size,
            strides=2,
            padding="same",
            kernel_initializer=initializer,
            use_bias=False,
        )
    )
    if apply_norm:
        result.add(tfa.layers.InstanceNormalization())
    result.add(tf.keras.layers.LeakyReLU())
    return result
```

In [22]:
```
# Upsampling method
def upsample(filters, size, apply_dropout=False):
    initializer = tf.random_normal_initializer(0.0, 0.02)
    result = tf.keras.Sequential()
    result.add(
```

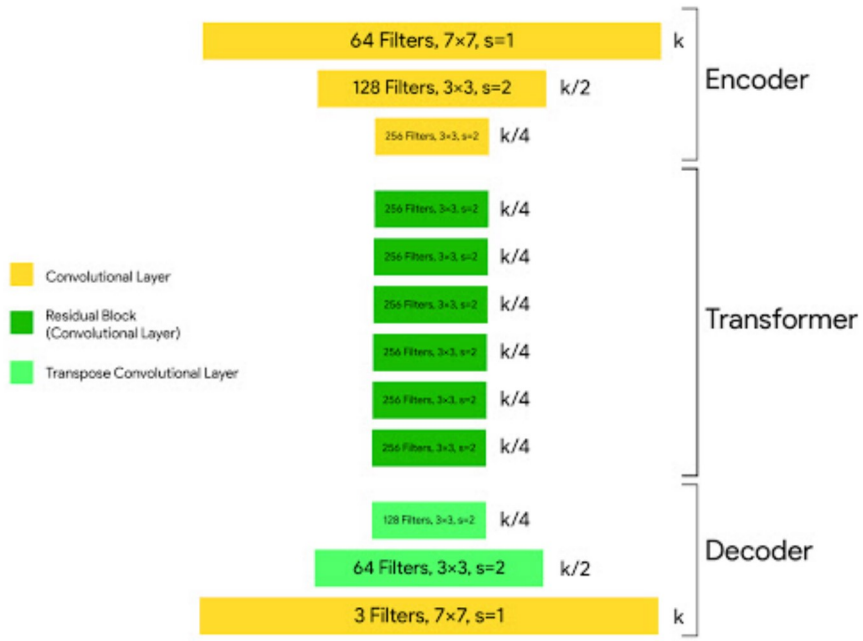
```
        tf.keras.layers.Conv2DTranspose(
            filters,
            size,
            strides=2,
            padding="same",
            kernel_initializer=initializer,
            use_bias=False,
        )
    )
    result.add(tfa.layers.InstanceNormalization())
    if apply_dropout:
        result.add(tf.keras.layers.Dropout(0.5))
    result.add(tf.keras.layers.ReLU())
    return result
```

Generator Architecture:

Each CycleGAN generator has three sections:

- Encoder
- Transformer
- Decoder

The encoder extracts features from the input image by using Convolutions and compressed the representation of image but increase the number of channels. Then the output of encoder after activation function is applied is passed into the transformer. The transformer contains 6 or 9 residual blocks based on the size of input. The output of transformer is then passed into the decoder which uses 2 -deconvolution block of fraction strides to increase the size of representation to original size.



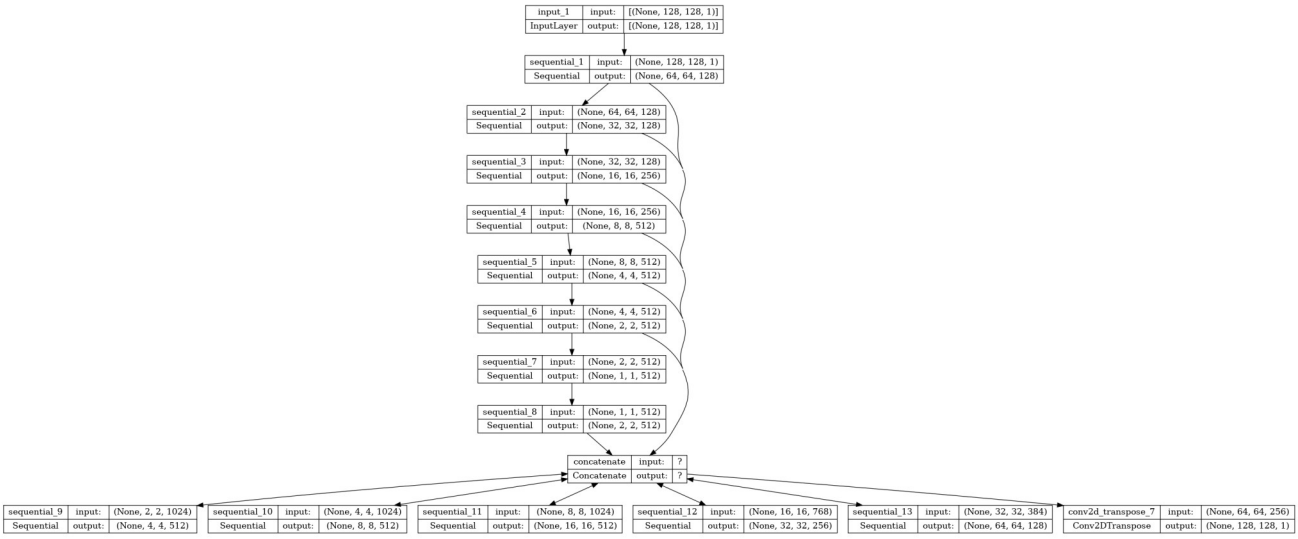
In [23]:
```
def unet_generator():
    down_stack = [
        downsample(128, 4, apply_norm=False),
        downsample(128, 4),
        downsample(256, 4),
        downsample(512, 4),
        downsample(512, 4),
        downsample(512, 4),
        downsample(512, 4),
    ]
    # create a stack of downsample models
    up_stack = [
```

```
        upsample(512, 4, apply_dropout=True),
        upsample(512, 4, apply_dropout=True),
        upsample(512, 4, apply_dropout=True),
        upsample(512, 4),
        upsample(256, 4),
        upsample(128, 4),
        upsample(128, 4),
    ]  # create a stack of upsample models

    initializer = tf.random_normal_initializer(0.0, 0.02)
    last = tf.keras.layers.Conv2DTranspose(
        1,
        4,
        strides=2,
        padding="same",
        kernel_initializer=initializer,
        activation="tanh",
    )
    concat = tf.keras.layers.Concatenate()
    inputs = tf.keras.layers.Input(shape=[128, 128, 1])
    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 = concat([x, skip])
    x = last(x)
    return tf.keras.Model(inputs=inputs, outputs=x)
```

In [24]:
```
# Creating generator models
generator_g = unet_generator()
generator_f = unet_generator()
```

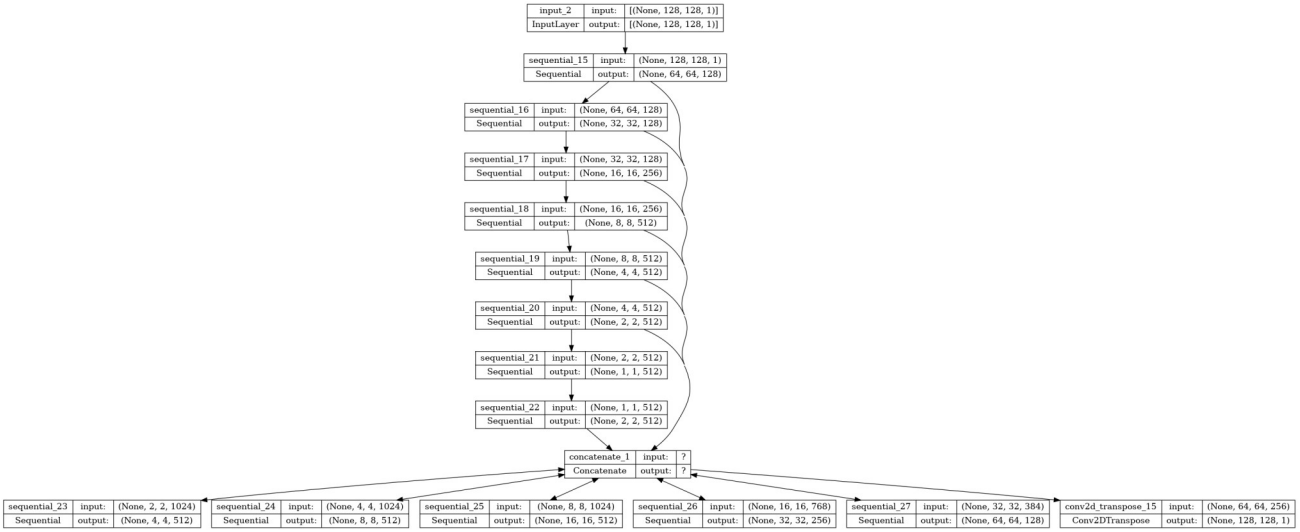
In [25]:
```
# Printing summary of generator g
plot_model(generator_g, show_shapes=True)
```

Out[25]:



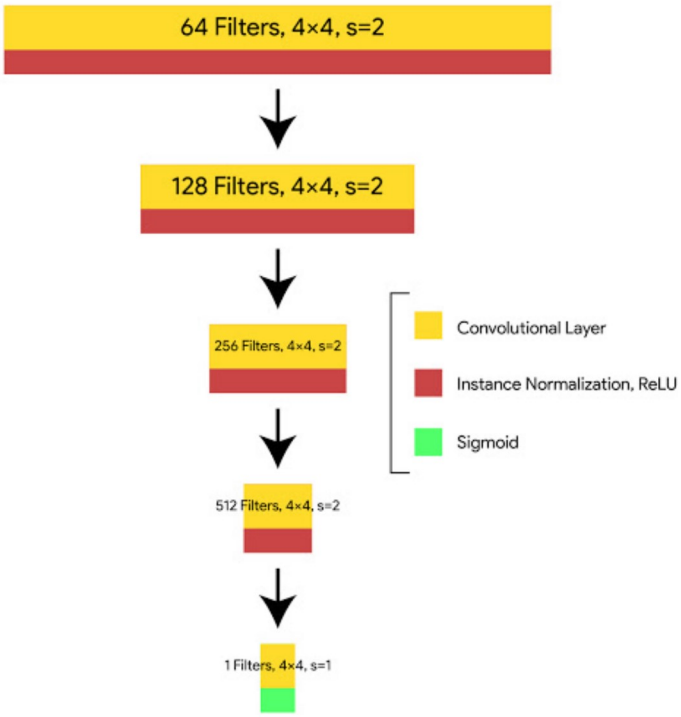
In [26]:
```
# Printing summary of generator f
plot_model(generator_f, show_shapes=True)
```

Out[26]:



Discriminator architecture :

In discriminator of a cycle gan we use PatchGAN discriminator. The difference between a PatchGAN and regular GAN discriminator is that rather the regular GAN maps from a 256×256 image to a single scalar output, which signifies “real” or “fake”, whereas the PatchGAN maps from MxM to an NxN array of outputs X, where each Xij signifies whether the patch ij in the image is real or fake.



PatchGAN discriminator evaluates whether local patches of the generated images match corresponding patches in the real images. By examining local patches, the PatchGAN discriminator can provide fine-grained feedback to the generator, facilitating more detailed and realistic image translations.

The use of PatchGAN discriminator in CycleGANs allows for more precise discrimination and enables the generator to focus on generating high-quality images at the patch level.

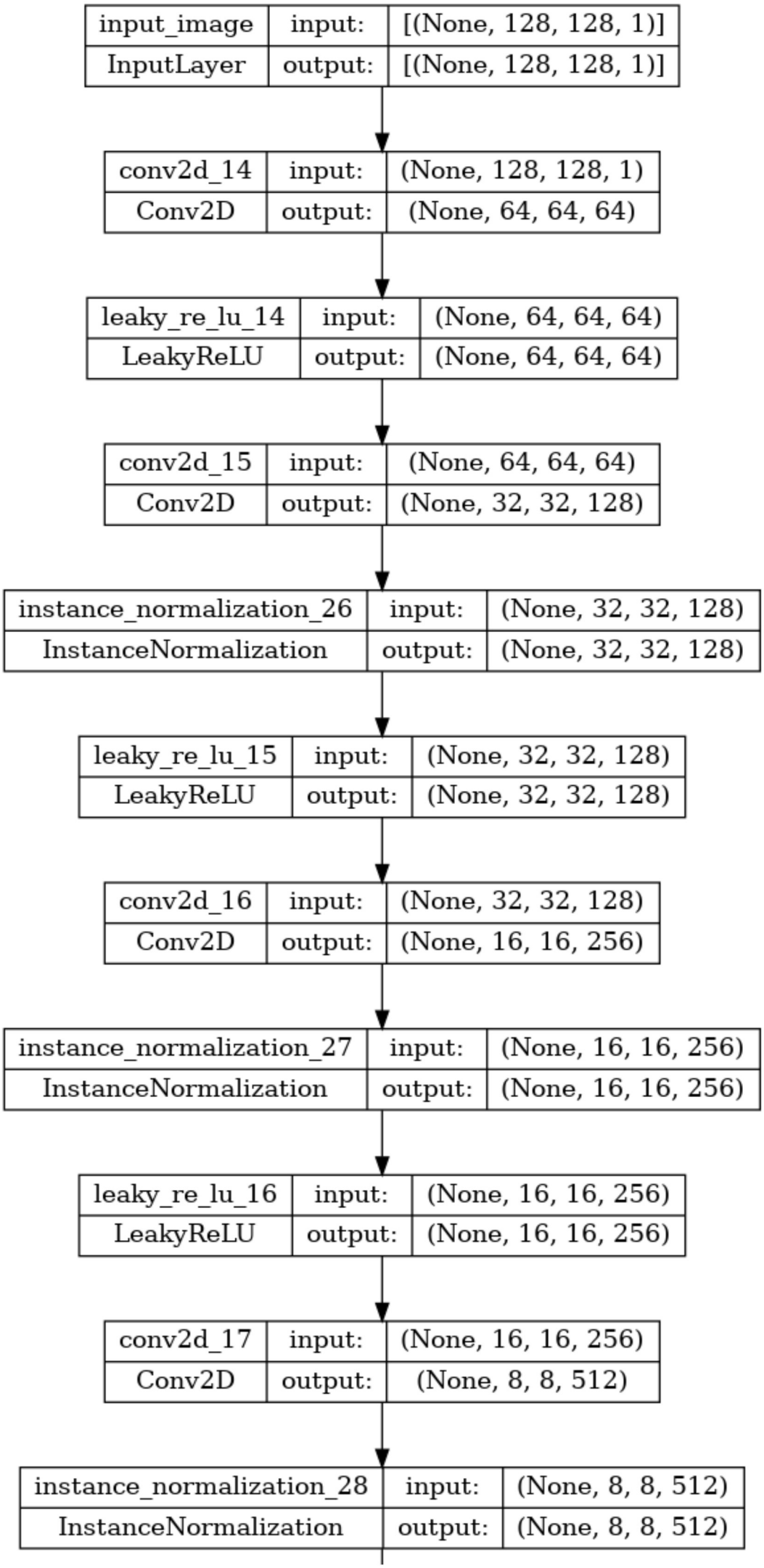
In [27]:
```
# function for discriminator
def discriminator():
    in_image = tf.keras.layers.Input(shape=[128, 128, 1], name="input_image")
    d = tf.keras.layers.Conv2D(
        64, (4, 4), strides=(2, 2), padding="same", kernel_initializer=kernel_init
    )(in_image)
    d = tf.keras.layers.LeakyReLU(alpha=0.2)(d)
    d = tf.keras.layers.Conv2D(
        128, (4, 4), strides=(2, 2), padding="same", kernel_initializer=kernel_init
```

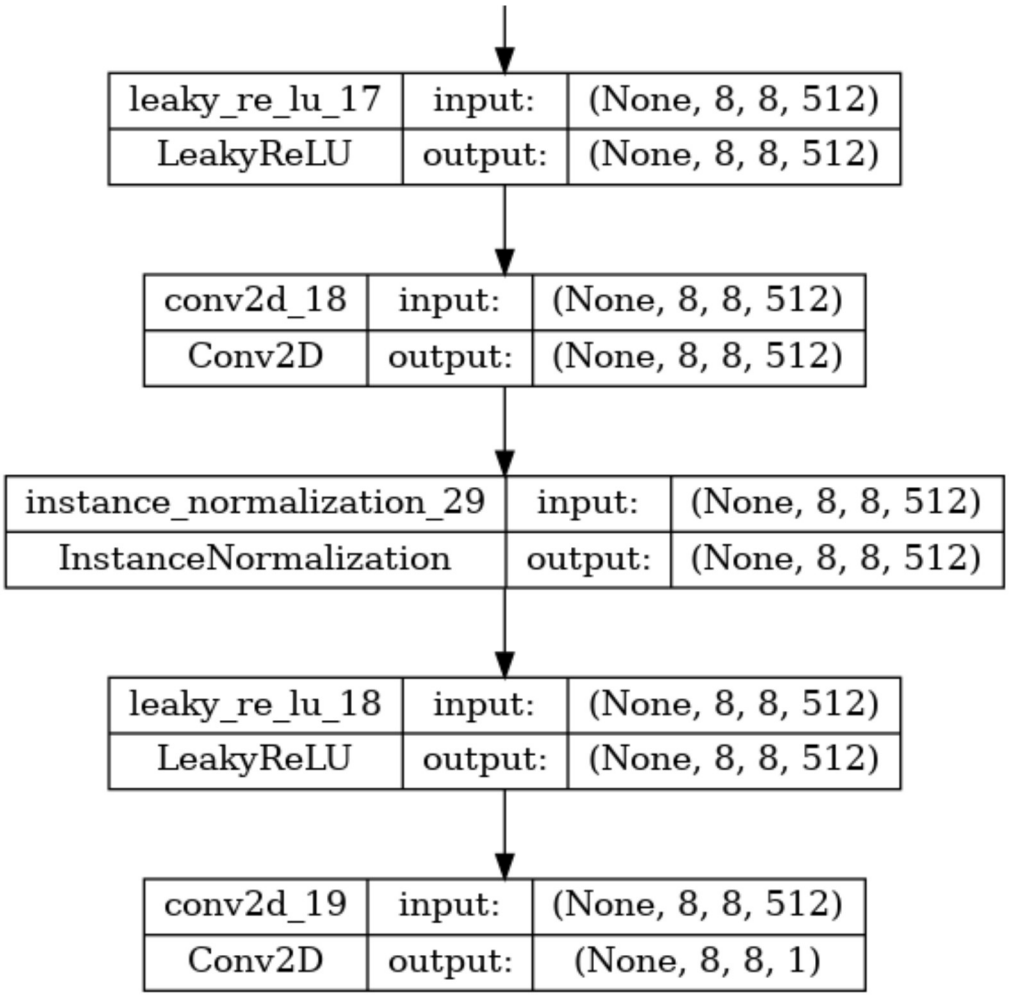
```
    )(d)
    d = tfa.layers.InstanceNormalization(axis=-1)(d)
    d = tf.keras.layers.LeakyReLU(alpha=0.2)(d)
    d = tf.keras.layers.Conv2D(
        256, (4, 4), strides=(2, 2), padding="same", kernel_initializer=kernel_init
    )(d)
    d = tfa.layers.InstanceNormalization(axis=-1)(d)
    d = tf.keras.layers.LeakyReLU(alpha=0.2)(d)
    d = tf.keras.layers.Conv2D(
        512, (4, 4), strides=(2, 2), padding="same", kernel_initializer=kernel_init
    )(d)
    d = tfa.layers.InstanceNormalization(axis=-1)(d)
    d = tf.keras.layers.LeakyReLU(alpha=0.2)(d)
    d = tf.keras.layers.Conv2D(
        512, (4, 4), padding="same", kernel_initializer=kernel_init
    )(d)
    d = tfa.layers.InstanceNormalization(axis=-1)(d)
    d = tf.keras.layers.LeakyReLU(alpha=0.2)(d)
    patch_out = tf.keras.layers.Conv2D(
        1, (4, 4), padding="same", kernel_initializer=kernel_init
    )(d)
    return tf.keras.models.Model(in_image, patch_out)
```

In [28]:
```
# Creating generator models
discriminator_x = discriminator()
discriminator_y = discriminator()
```

In [29]:
```
# Printing summary of generator g
plot_model(discriminator_x, show_shapes=True)
```

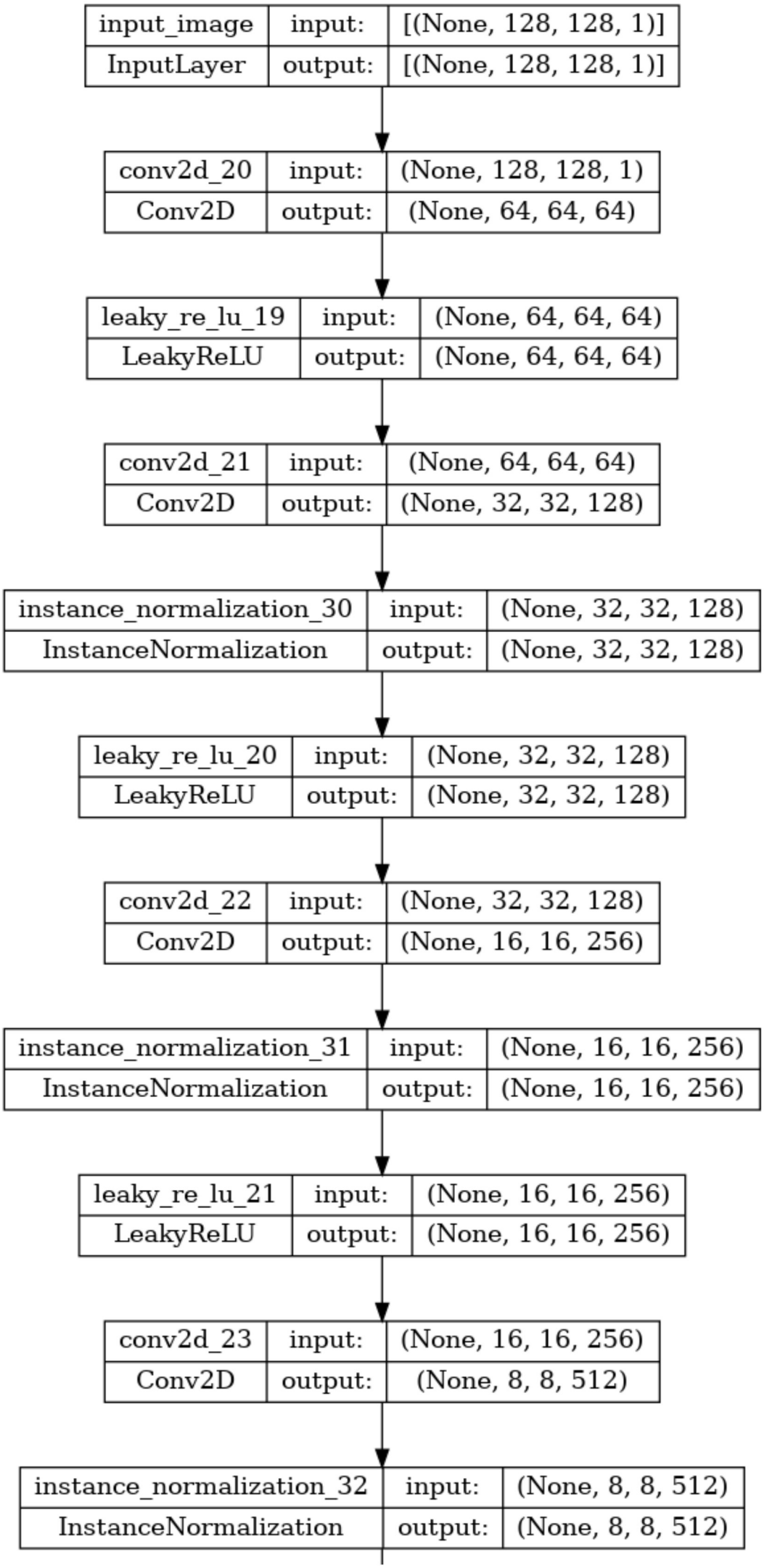
Out[29]:

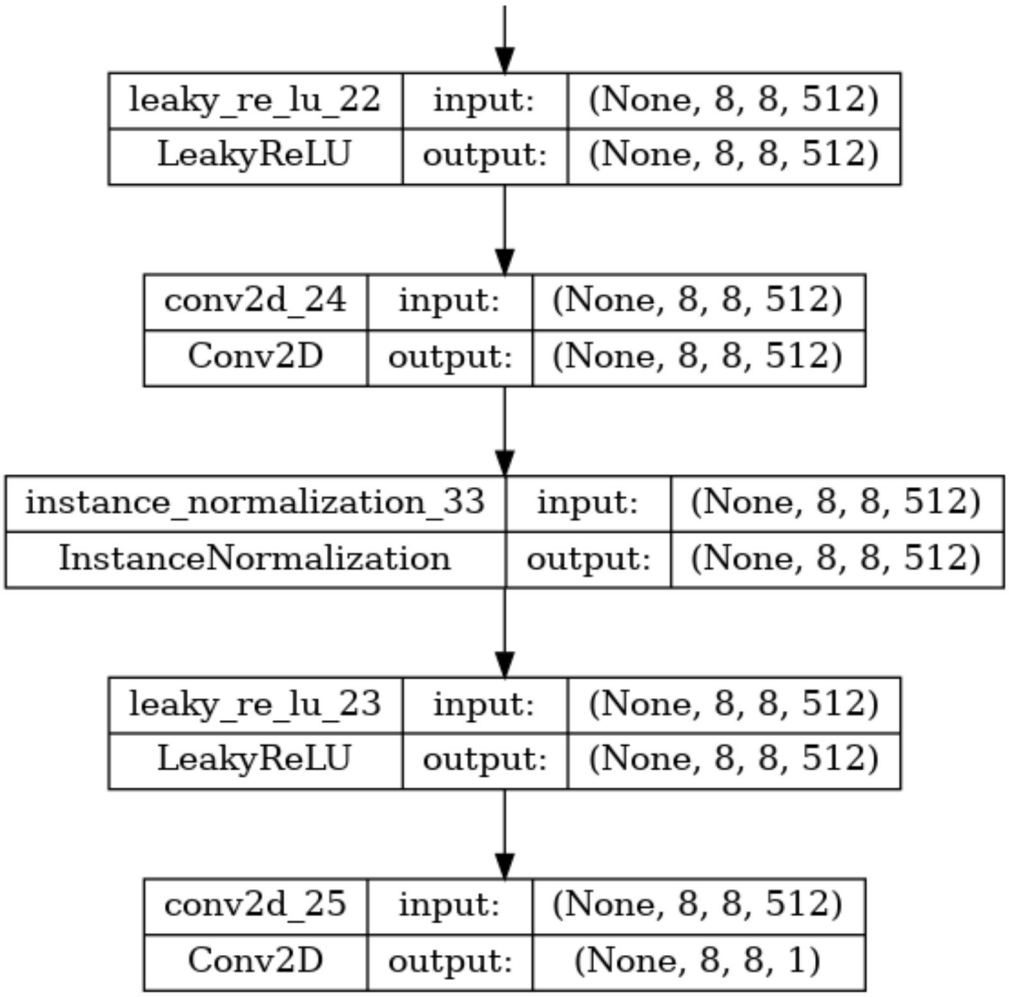

input_image | input: | [(None, 128, 128, 1)]
InputLayer | output: | [(None, 128, 128, 1)]
conv2d_14 | input: | (None, 128, 128, 1)
Conv2D | output: | (None, 64, 64, 64)
leaky_re_lu_14 | input: | (None, 64, 64, 64)
LeakyReLU | output: | (None, 64, 64, 64)
conv2d_15 | input: | (None, 64, 64, 64)
Conv2D | output: | (None, 32, 32, 128)
instance_normalization_26 | input: | (None, 32, 32, 128)
InstanceNormalization | output: | (None, 32, 32, 128)
leaky_re_lu_15 | input: | (None, 32, 32, 128)
LeakyReLU | output: | (None, 32, 32, 128)
conv2d_16 | input: | (None, 32, 32, 128)
Conv2D | output: | (None, 16, 16, 256)
instance_normalization_27 | input: | (None, 16, 16, 256)
InstanceNormalization | output: | (None, 16, 16, 256)
leaky_re_lu_16 | input: | (None, 16, 16, 256)
LeakyReLU | output: | (None, 16, 16, 256)
conv2d_17 | input: | (None, 16, 16, 256)
Conv2D | output: | (None, 8, 8, 512)
instance_normalization_28 | input: | (None, 8, 8, 512)
InstanceNormalization | output: | (None, 8, 8, 512)



In [30]:
```
# Printing summary of generator f
plot_model(discriminator_y, show_shapes=True)
```

Out[30]:


input_image | input: | [(None, 128, 128, 1)]
InputLayer | output: | [(None, 128, 128, 1)]
conv2d_20 | input: | (None, 128, 128, 1)
Conv2D | output: | (None, 64, 64, 64)
leaky_re_lu_19 | input: | (None, 64, 64, 64)
LeakyReLU | output: | (None, 64, 64, 64)
conv2d_21 | input: | (None, 64, 64, 64)
Conv2D | output: | (None, 32, 32, 128)
instance_normalization_30 | input: | (None, 32, 32, 128)
InstanceNormalization | output: | (None, 32, 32, 128)
leaky_re_lu_20 | input: | (None, 32, 32, 128)
LeakyReLU | output: | (None, 32, 32, 128)
conv2d_22 | input: | (None, 32, 32, 128)
Conv2D | output: | (None, 16, 16, 256)
instance_normalization_31 | input: | (None, 16, 16, 256)
InstanceNormalization | output: | (None, 16, 16, 256)
leaky_re_lu_21 | input: | (None, 16, 16, 256)
LeakyReLU | output: | (None, 16, 16, 256)
conv2d_23 | input: | (None, 16, 16, 256)
Conv2D | output: | (None, 8, 8, 512)
instance_normalization_32 | input: | (None, 8, 8, 512)
InstanceNormalization | output: | (None, 8, 8, 512)



Cost functions

Adversarial Loss:

We apply adversarial loss to both our mappings of generators and discriminators. This adversary loss is written as

$$Loss_{advers}\left(F, D_x, Y, X\right) = \frac{1}{m}\sum\left(1 - D_x\left(F\left(y\right)\right)\right)^2$$

Cycle consistancy loss :

Given a random set of images adversarial network can map the set of input image to random permutation of images in the output domain which may induce the output distribution similar to target distribution. Thus adversarial mapping cannot guarantee the input xi to yi . For this to happen the author proposed that process should be cycle-consistent. This loss function used in Cycle GAN to measure the error rate of inverse mapping G(x) -> F(G(x)). The behavior induced by this loss function cause closely matching the real input (x) and F(G(x))

$$Loss_{cyc}\left(G, F, X, Y\right) = \frac{1}{m}\left[\left(F\left(G\left(x_i\right)\right) - x_i\right) + \left(G\left(F\left(y_i\right)\right) - y_i\right)\right]$$

Cost function we used is the sum of adversarial loss and cyclic consistent loss :

$$L\left(G, F, D_x, D_y\right) = L_{advers}\left(G, D_y, X, Y\right) + L_{advers}\left(F, D_x, Y, X\right) + \lambda L_{cycl}\left(G, F, X, Y\right)$$

In [31]:
```
# Binary cross entropy loss function
bce_loss = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

In [32]:
```
# Define the loss function for the generators
def generator_loss(generated):
```

```
    return bce_loss(tf.ones_like(generated), generated)
```

In [33]:
```
# function for discriminator loss
def discriminator_loss(real, generated):
    real_loss = bce_loss(tf.ones_like(real), real)
    generated_loss = bce_loss(tf.zeros_like(generated), generated)
    return (real_loss + generated_loss) * 0.5
```

In [34]:
```
# Cycle consistency loss function
def calc_cycle_loss(real_image, cycled_image):
    return tf.reduce_mean(tf.abs(real_image - cycled_image))
```

In [35]:
```
# Identity loss function
def identity_loss(real_image, same_image):
    return tf.reduce_mean(tf.abs(real_image - same_image))
```

Parameters for Loss Function defined in CycleGAN Model class (10,0.5)

Defining Optimizers

- Cyclegan uses instance normalization instead of batch normalization.

In [36]:
```
# Leveraging Adam Optimizer to smoothen gradient descent
generator_g_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
generator_f_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)

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

CycleGAN Model

In [37]:
```
class CycleGan(tf.keras.Model):
    def __init__(
        self,
        generator_G,
        generator_F,
        discriminator_X,
        discriminator_Y,
        lambda_cycle=10.0,
        lambda_identity=0.5,
    ):
        super().__init__()
        self.gen_G = generator_G
        self.gen_F = generator_F
        self.disc_X = discriminator_X
        self.disc_Y = discriminator_Y
        self.lambda_cycle = lambda_cycle
        self.lambda_identity = lambda_identity

    def call(self, inputs):
        return (
            self.disc_X(inputs),
            self.disc_Y(inputs),
            self.gen_G(inputs),
            self.gen_F(inputs),
        )

    def compile(
        self,
        gen_G_optimizer,
        gen_F_optimizer,
        disc_X_optimizer,
        disc_Y_optimizer,
        gen_loss_fn,
        disc_loss_fn,
        cycle_loss_fn,
```

```
        identity_loss_fn,
    ):
        super().compile()
        self.gen_G_optimizer = gen_G_optimizer
        self.gen_F_optimizer = gen_F_optimizer
        self.disc_X_optimizer = disc_X_optimizer
        self.disc_Y_optimizer = disc_Y_optimizer
        self.generator_loss_fn = gen_loss_fn
        self.discriminator_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_x, real_y = batch_data
        with tf.GradientTape(persistent=True) as tape:
            fake_y = self.gen_G(real_x, training=True)
            fake_x = self.gen_F(real_y, training=True)

            cycled_x = self.gen_F(fake_y, training=True)
            cycled_y = self.gen_G(fake_x, training=True)

            # Identity mapping
            same_x = self.gen_F(real_x, training=True)
            same_y = self.gen_G(real_y, training=True)

            # Discriminator output
            disc_real_x = self.disc_X(real_x, training=True)
            disc_fake_x = self.disc_X(fake_x, training=True)

            disc_real_y = self.disc_Y(real_y, training=True)
            disc_fake_y = self.disc_Y(fake_y, training=True)

            # Generator adversarial loss
            gen_G_loss = self.generator_loss_fn(disc_fake_y)
            gen_F_loss = self.generator_loss_fn(disc_fake_x)

            # Generator cycle loss
            cycle_loss_G = self.cycle_loss_fn(real_y, cycled_y) * self.lambda_cycle
            cycle_loss_F = self.cycle_loss_fn(real_x, cycled_x) * self.lambda_cycle

            # Generator identity loss
            id_loss_G = (
                self.identity_loss_fn(real_y, same_y)
                * self.lambda_cycle
                * self.lambda_identity
            )
            id_loss_F = (
                self.identity_loss_fn(real_x, same_x)
                * self.lambda_cycle
                * self.lambda_identity
            )

            # Total generator loss
            total_loss_G = gen_G_loss + cycle_loss_G + id_loss_G
            total_loss_F = gen_F_loss + cycle_loss_F + id_loss_F

            # Discriminator loss
            disc_X_loss = self.discriminator_loss_fn(disc_real_x, disc_fake_x)
            disc_Y_loss = self.discriminator_loss_fn(disc_real_y, disc_fake_y)

        # Get the gradients for the generators
        grads_G = tape.gradient(total_loss_G, self.gen_G.trainable_variables)
        grads_F = tape.gradient(total_loss_F, self.gen_F.trainable_variables)

        # Get the gradients for the discriminators
        disc_X_grads = tape.gradient(disc_X_loss, self.disc_X.trainable_variables)
        disc_Y_grads = tape.gradient(disc_Y_loss, self.disc_Y.trainable_variables)

        # Update the weights of the generators
```

```
        self.gen_G_optimizer.apply_gradients(
            zip(grads_G, self.gen_G.trainable_variables)
        )
        self.gen_F_optimizer.apply_gradients(
            zip(grads_F, self.gen_F.trainable_variables)
        )

        # Update the weights of the discriminators
        self.disc_X_optimizer.apply_gradients(
            zip(disc_X_grads, self.disc_X.trainable_variables)
        )
        self.disc_Y_optimizer.apply_gradients(
            zip(disc_Y_grads, self.disc_Y.trainable_variables)
        )

        return {
            "G_loss": total_loss_G,
            "F_loss": total_loss_F,
            "D_X_loss": disc_X_loss,
            "D_Y_loss": disc_Y_loss,
            "Total_Generator_Loss": total_loss_G+total_loss_F,
        }
```

CallBacks defined

In [38]:
```
# Getting Sample Image
t1_sample = t1_images.shuffle(buffer_size).as_numpy_iterator().next()
t2_sample = t2_images.shuffle(buffer_size).as_numpy_iterator().next()


class GANMonitor(tf.keras.callbacks.Callback):
    def __init__(self, num_img=4):
        self.num_img = num_img

    def on_train_begin(self, epoch, logs=None):
        prediction1 = self.model.gen_G.predict(t1_sample)
        prediction2 = self.model.gen_F.predict(t2_sample)
        fig = plt.figure(figsize=(8, 4))
        display_list = [t1_sample[0], prediction1[0], t2_sample[0], prediction2[0]]
        title = ["Input Image", "Generated Image", "Input Image", "Generated Image"]
        for i in range(4):
            plt.subplot(1, 4, i + 1)
            plt.title(title[i])
            plt.imshow(display_list[i][:, :, 0], cmap="gray")
            plt.axis("off")
        fig.suptitle("CycleGAN Sample Output Before Training", fontsize=16)
        plt.savefig("IMG/image_at_epoch_0.png")
        plt.show()
        plt.close()

    def on_epoch_end(self, epoch, logs=None):
        prediction1 = self.model.gen_G.predict(t1_sample)
        prediction2 = self.model.gen_F.predict(t2_sample)
        fig = plt.figure(figsize=(8, 4))
        display_list = [t1_sample[0], prediction1[0], t2_sample[0], prediction2[0]]
        title = ["Input Image", "Generated Image", "Input Image", "Generated Image"]
        for i in range(4):
            plt.subplot(1, 4, i + 1)
            plt.title(title[i])
            plt.imshow(display_list[i][:, :, 0], cmap="gray")
            plt.axis("off")
        fig.suptitle(f"CycleGAN Sample Output After epoch {epoch+1}", fontsize=16)
        plt.savefig(f"IMG/image_at_epoch_{epoch+1}.png")
        plt.show()
        plt.close()
```

In [39]:
```
# Checkpoint callback
checkpoint_filepath = (
```

```
    "./Trained_Model/cyclegan_checkpoints.{epoch:03d}"
)
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    monitor="Total_Generator_Loss",filepath=checkpoint_filepath, save_weights_only=True, save_freq=
)
```

Model Training

In [40]:
```
# Defining epochs
epochs = 300
```

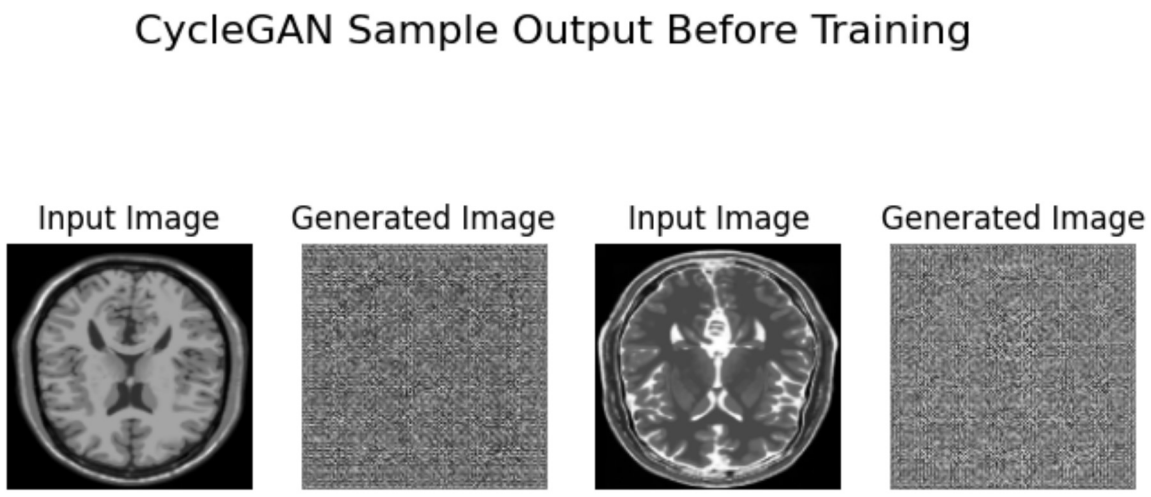
In [41]:
```
# create a CycleGAN model
cycle_gan_model = CycleGan(
    generator_F=generator_f,
    generator_G=generator_g,
    discriminator_X=discriminator_x,
    discriminator_Y=discriminator_y,
)
```

In [42]:
```
# compile the model
cycle_gan_model.compile(
    gen_G_optimizer=generator_g_optimizer,
    gen_F_optimizer=generator_f_optimizer,
    disc_X_optimizer=discriminator_x_optimizer,
    disc_Y_optimizer=discriminator_y_optimizer,
    gen_loss_fn=generator_loss,
    disc_loss_fn=discriminator_loss,
    cycle_loss_fn=calc_cycle_loss,
    identity_loss_fn=identity_loss,
)
```

In [43]:
```
# Callbacks
plotter = GANMonitor()
```

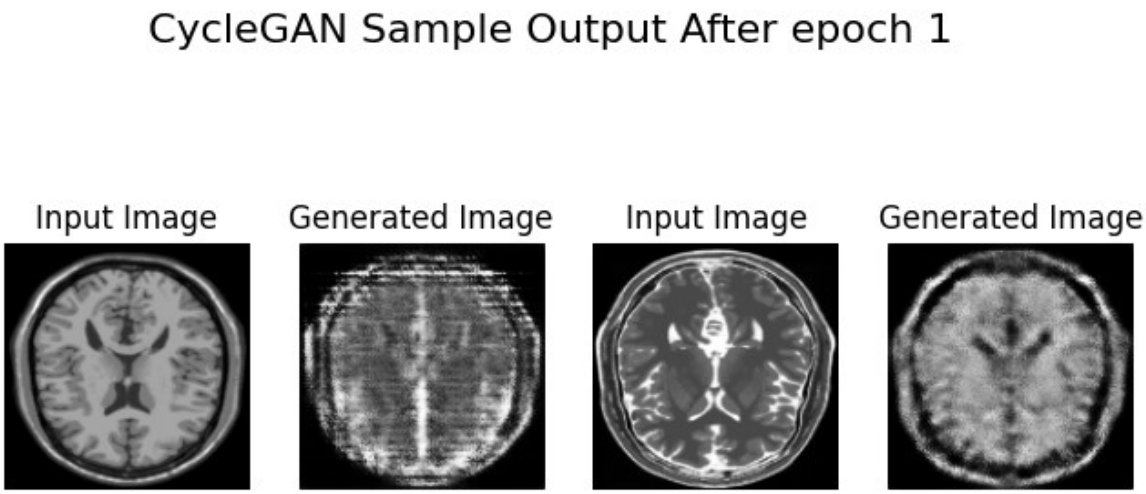
In [44]:
```
# Fit the model
history = cycle_gan_model.fit(
    tf.data.Dataset.zip((t1_images, t2_images)),
    epochs=epochs,
    callbacks=[plotter],
)
```

```
2023-06-23 19:26:49.681368: I tensorflow/compiler/xla/stream_executor/cuda/cuda_dnn.cc:428] Loaded c
uDNN version 8700
1/1 [==============================] - 3s 3s/step
1/1 [==============================] - 0s 359ms/step
```

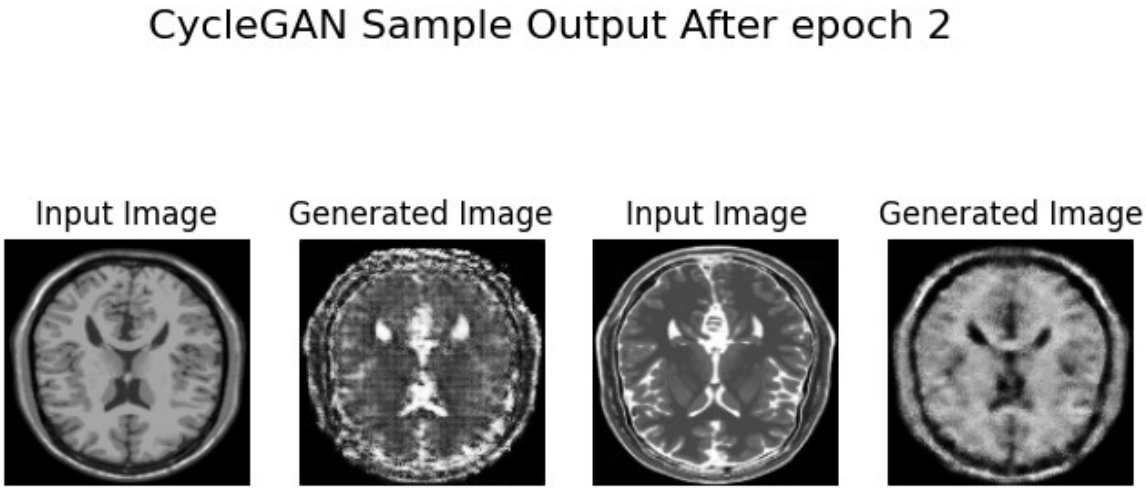



```
Epoch 1/300
```

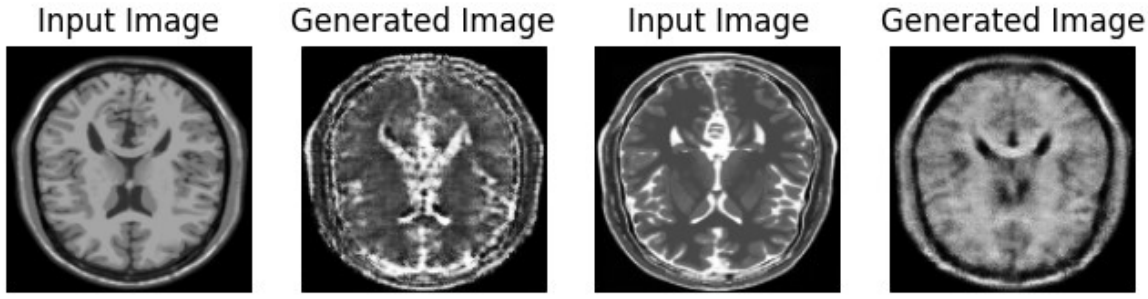
```
2023-06-23 19:27:10.692995: E tensorflow/core/grappler/optimizers/meta_optimizer.cc:954] layout fail
ed: INVALID_ARGUMENT: Size of values 0 does not match size of permutation 4 @ fanin shape inmodel/se
quential_8/dropout/dropout/SelectV2-2-TransposeNHWCToNCHW-LayoutOptimizer
2023-06-23 19:27:14.508670: I tensorflow/compiler/xla/service/service.cc:173] XLA service 0x7f899c00
a620 initialized for platform CUDA (this does not guarantee that XLA will be used). Devices:
2023-06-23 19:27:14.508726: I tensorflow/compiler/xla/service/service.cc:181]   StreamExecutor devic
e (0): NVIDIA GeForce RTX 3060 Ti, Compute Capability 8.6
2023-06-23 19:27:14.513057: I tensorflow/compiler/mlir/tensorflow/utils/dump_mlir_util.cc:268] disab
ling MLIR crash reproducer, set env var `MLIR_CRASH_REPRODUCER_DIRECTORY` to enable.
2023-06-23 19:27:14.592127: I tensorflow/compiler/jit/xla_compilation_cache.cc:477] Compiled cluster
using XLA!  This line is logged at most once for the lifetime of the process.
1/1 [==============================] - 0s 17ms/stepG_loss: 4.7778 - F_loss: 3.8402 - D_X_loss: 0.635
7 - D_Y_loss: 0.6117 - Total_Generator_Loss: 8.61
1/1 [==============================] - 0s 20ms/step
```



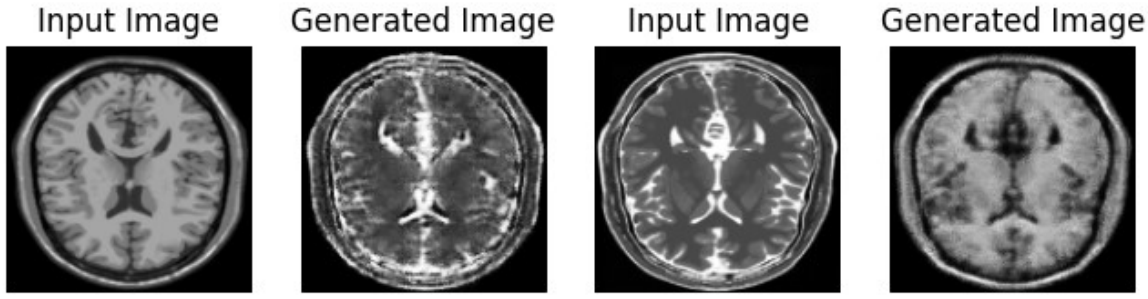

```
86/86 [==============================] - 45s 210ms/step - G_loss: 4.7769 - F_loss: 3.8258 - D_X_loss
: 0.6344 - D_Y_loss: 0.6102 - Total_Generator_Loss: 8.6027
Epoch 2/300
1/1 [==============================] - 0s 18ms/stepG_loss: 4.1601 - F_loss: 2.6853 - D_X_loss: 0.644
7 - D_Y_loss: 0.5526 - Total_Generator_Loss: 6.84
1/1 [==============================] - 0s 15ms/step
```



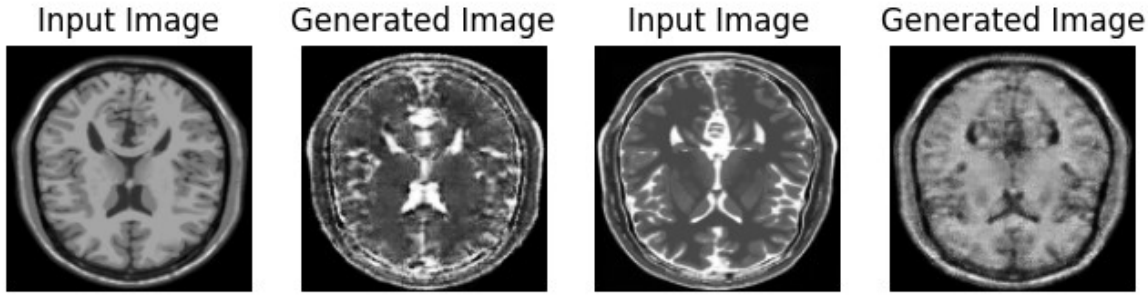

```
86/86 [==============================] - 19s 220ms/step - G_loss: 4.1624 - F_loss: 2.6857 - D_X_loss
: 0.6437 - D_Y_loss: 0.5515 - Total_Generator_Loss: 6.8481
Epoch 3/300
1/1 [==============================] - 0s 16ms/stepG_loss: 3.8613 - F_loss: 2.5609 - D_X_loss: 0.618
9 - D_Y_loss: 0.5341 - Total_Generator_Loss: 6.42
1/1 [==============================] - 0s 15ms/step
```



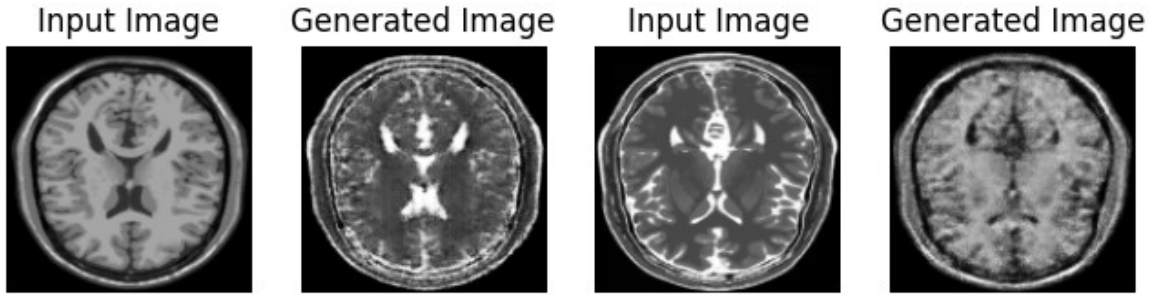

```
86/86 [==============================] - 18s 215ms/step - G_loss: 3.8548 - F_loss: 2.5555 - D_X_loss
: 0.6214 - D_Y_loss: 0.5333 - Total_Generator_Loss: 6.4102
Epoch 4/300
1/1 [==============================] - 0s 16ms/stepG_loss: 3.3925 - F_loss: 2.2832 - D_X_loss: 0.677
0 - D_Y_loss: 0.5868 - Total_Generator_Loss: 5.67
1/1 [==============================] - 0s 15ms/step
```



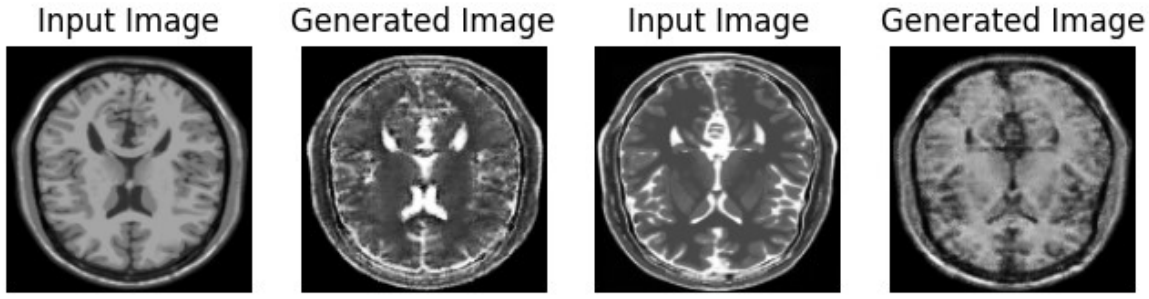

```
86/86 [==============================] - 18s 212ms/step - G_loss: 3.3868 - F_loss: 2.2779 - D_X_loss
: 0.6763 - D_Y_loss: 0.5888 - Total_Generator_Loss: 5.6647
Epoch 5/300
1/1 [==============================] - 0s 16ms/stepG_loss: 3.1415 - F_loss: 2.1748 - D_X_loss: 0.661
3 - D_Y_loss: 0.5609 - Total_Generator_Loss: 5.31
1/1 [==============================] - 0s 15ms/step
```



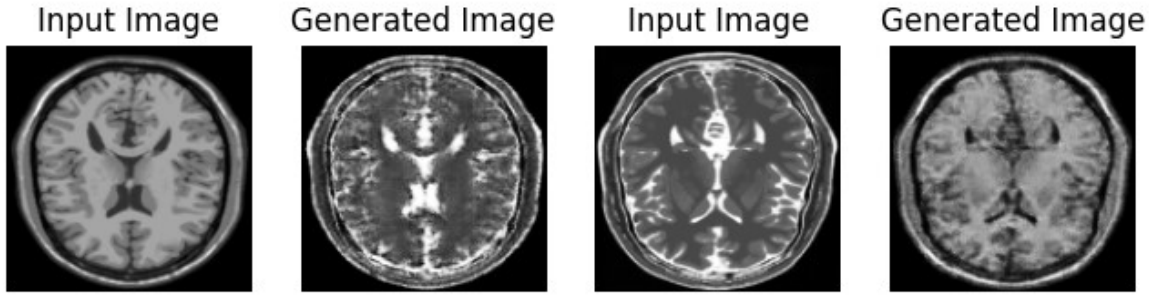

```
86/86 [==============================] - 18s 212ms/step - G_loss: 3.1354 - F_loss: 2.1739 - D_X_loss
: 0.6614 - D_Y_loss: 0.5601 - Total_Generator_Loss: 5.3093
Epoch 6/300
1/1 [==============================] - 0s 19ms/stepG_loss: 2.9984 - F_loss: 2.0426 - D_X_loss: 0.662
9 - D_Y_loss: 0.5743 - Total_Generator_Loss: 5.04
1/1 [==============================] - 0s 16ms/step
```



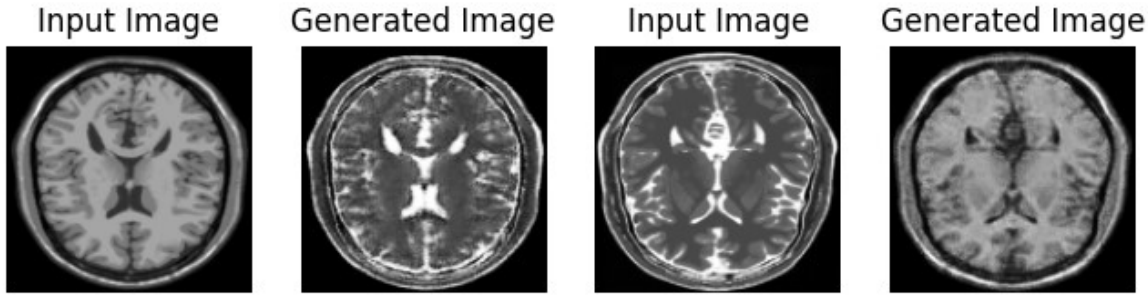

```
86/86 [==============================] - 18s 212ms/step - G_loss: 2.9992 - F_loss: 2.0393 - D_X_loss
: 0.6622 - D_Y_loss: 0.5762 - Total_Generator_Loss: 5.0385
Epoch 7/300
1/1 [==============================] - 0s 17ms/stepG_loss: 3.0107 - F_loss: 1.9900 - D_X_loss: 0.654
4 - D_Y_loss: 0.5112 - Total_Generator_Loss: 5.00
1/1 [==============================] - 0s 16ms/step
```



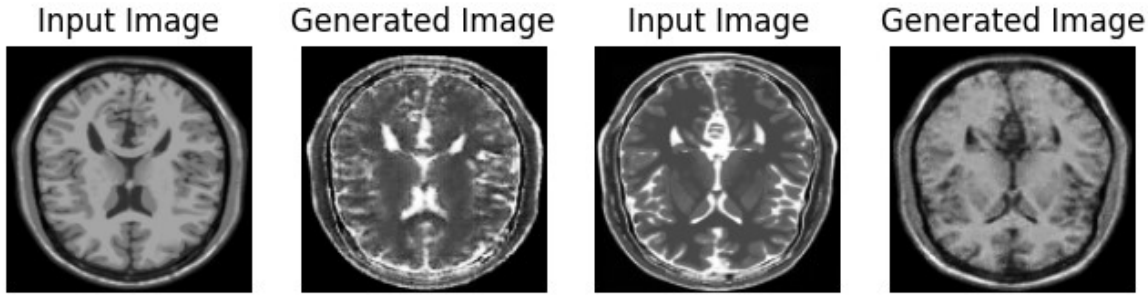

```
86/86 [==============================] - 19s 215ms/step - G_loss: 3.0087 - F_loss: 1.9857 - D_X_loss
: 0.6542 - D_Y_loss: 0.5097 - Total_Generator_Loss: 4.9944
Epoch 8/300
1/1 [==============================] - 0s 17ms/stepG_loss: 2.9544 - F_loss: 1.9785 - D_X_loss: 0.656
0 - D_Y_loss: 0.5397 - Total_Generator_Loss: 4.93
1/1 [==============================] - 0s 15ms/step
```



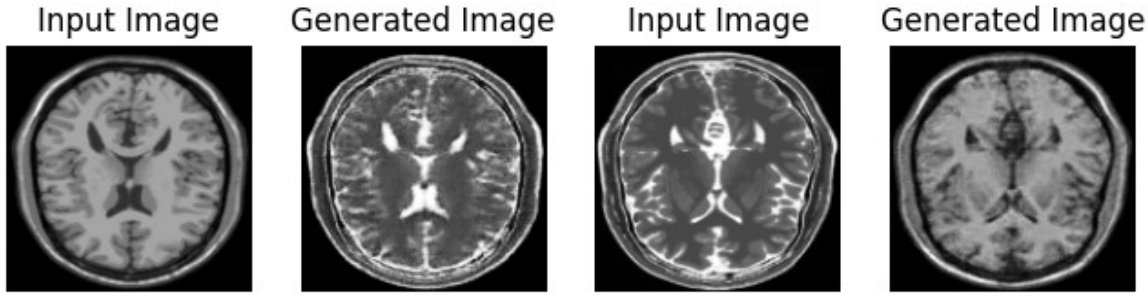

```
86/86 [==============================] - 19s 222ms/step - G_loss: 2.9470 - F_loss: 1.9813 - D_X_loss
: 0.6543 - D_Y_loss: 0.5398 - Total_Generator_Loss: 4.9283
Epoch 9/300
1/1 [==============================] - 0s 19ms/stepG_loss: 2.7452 - F_loss: 1.8464 - D_X_loss: 0.676
6 - D_Y_loss: 0.5874 - Total_Generator_Loss: 4.59
1/1 [==============================] - 0s 20ms/step
```



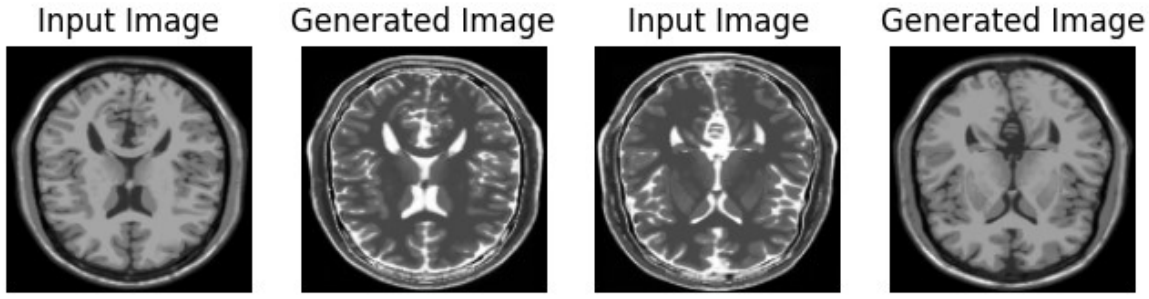

```
86/86 [==============================] - 18s 211ms/step - G_loss: 2.7405 - F_loss: 1.8432 - D_X_loss
: 0.6763 - D_Y_loss: 0.5871 - Total_Generator_Loss: 4.5837
Epoch 10/300
1/1 [==============================] - 0s 17ms/stepG_loss: 2.6173 - F_loss: 1.7244 - D_X_loss: 0.687
8 - D_Y_loss: 0.5863 - Total_Generator_Loss: 4.34
1/1 [==============================] - 0s 20ms/step
```



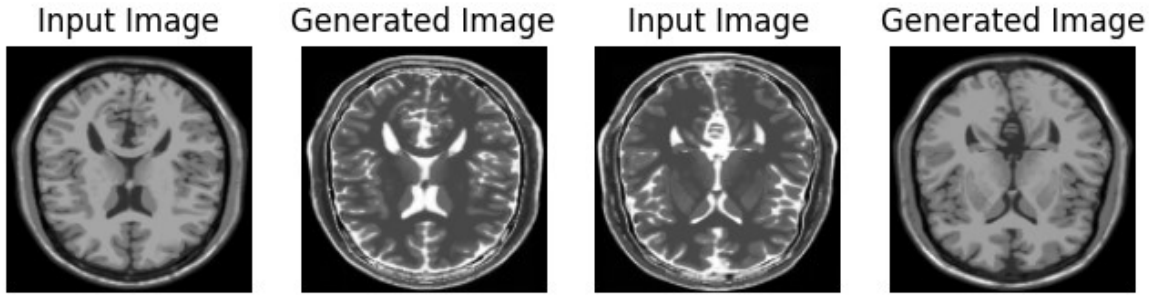

```
86/86 [==============================] - 18s 212ms/step - G_loss: 2.6196 - F_loss: 1.7209 - D_X_loss
: 0.6885 - D_Y_loss: 0.5854 - Total_Generator_Loss: 4.3405
Epoch 11/300
1/1 [==============================] - 0s 16ms/stepG_loss: 2.3101 - F_loss: 1.5513 - D_X_loss: 0.710
1 - D_Y_loss: 0.6416 - Total_Generator_Loss: 3.86
1/1 [==============================] - 0s 15ms/step
```



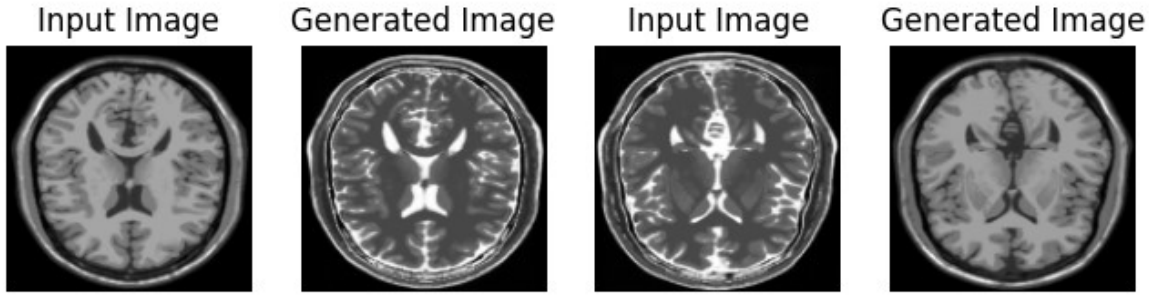

```
86/86 [==============================] - 18s 213ms/step - G_loss: 2.3070 - F_loss: 1.5514 - D_X_loss
: 0.7102 - D_Y_loss: 0.6411 - Total_Generator_Loss: 3.8584
Epoch 12/300
1/1 [==============================] - 0s 16ms/stepG_loss: 2.1209 - F_loss: 1.4583 - D_X_loss: 0.711
7 - D_Y_loss: 0.6425 - Total_Generator_Loss: 3.57
1/1 [==============================] - 0s 15ms/step
```



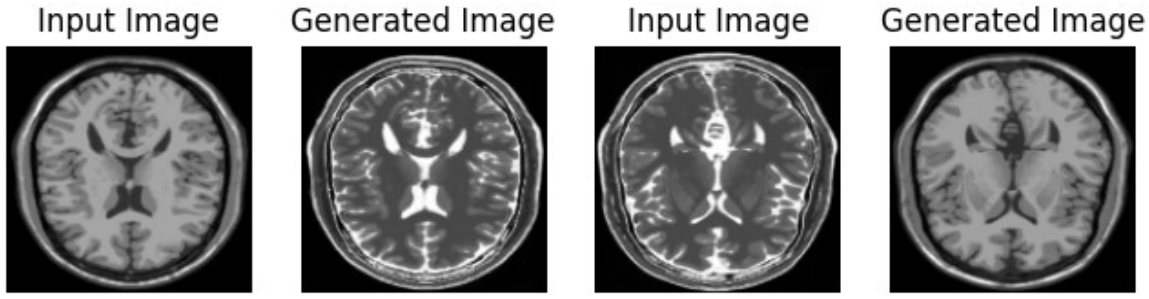

```
86/86 [==============================] - 19s 224ms/step - G_loss: 0.9518 - F_loss: 0.8219 - D_X_loss
: 0.6861 - D_Y_loss: 0.6733 - Total_Generator_Loss: 1.7738
Epoch 295/300
1/1 [==============================] - 0s 23ms/stepG_loss: 0.9546 - F_loss: 0.8250 - D_X_loss: 0.685
2 - D_Y_loss: 0.6757 - Total_Generator_Loss: 1.77
1/1 [==============================] - 0s 19ms/step
```



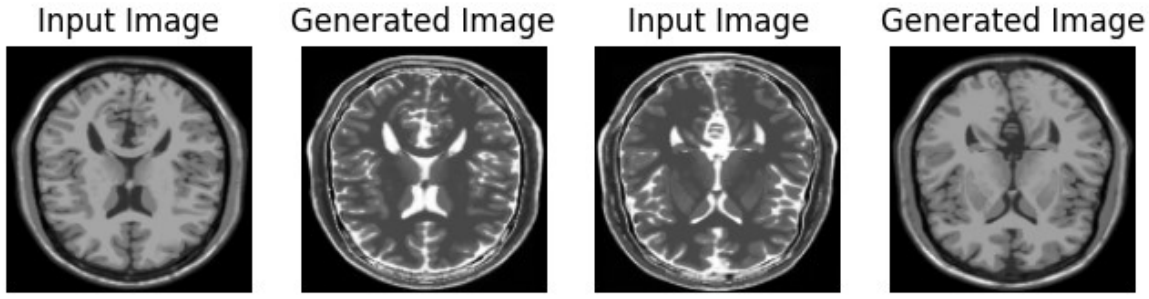

```
86/86 [==============================] - 19s 217ms/step - G_loss: 0.9532 - F_loss: 0.8266 - D_X_loss
: 0.6843 - D_Y_loss: 0.6749 - Total_Generator_Loss: 1.7798
Epoch 296/300
1/1 [==============================] - 0s 24ms/stepG_loss: 0.9428 - F_loss: 0.8324 - D_X_loss: 0.683
6 - D_Y_loss: 0.6819 - Total_Generator_Loss: 1.77
1/1 [==============================] - 0s 23ms/step
```



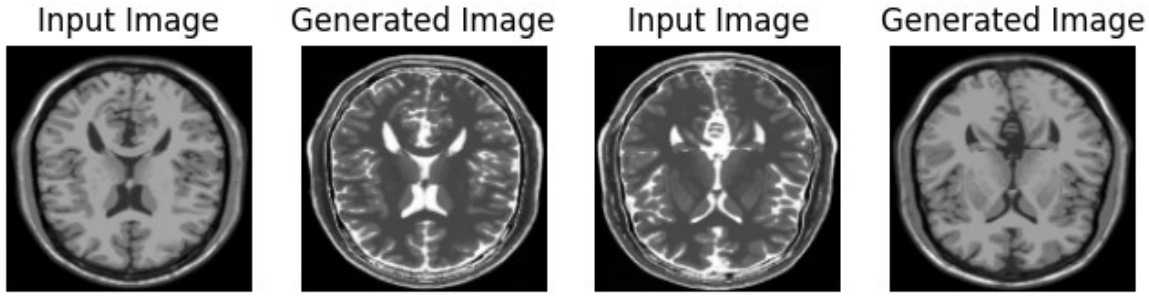

```
86/86 [==============================] - 19s 216ms/step - G_loss: 0.9417 - F_loss: 0.8342 - D_X_loss
: 0.6829 - D_Y_loss: 0.6805 - Total_Generator_Loss: 1.7759
Epoch 297/300
1/1 [==============================] - 0s 23ms/stepG_loss: 0.9578 - F_loss: 0.8170 - D_X_loss: 0.686
3 - D_Y_loss: 0.6703 - Total_Generator_Loss: 1.77
1/1 [==============================] - 0s 16ms/step
```



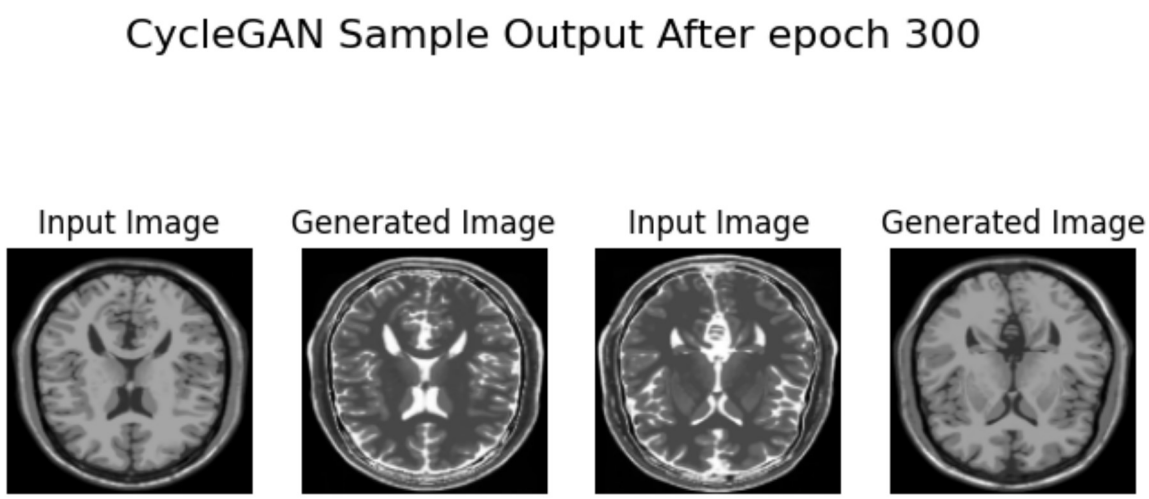

```
86/86 [==============================] - 18s 213ms/step - G_loss: 0.9557 - F_loss: 0.8150 - D_X_loss
: 0.6876 - D_Y_loss: 0.6713 - Total_Generator_Loss: 1.7707
Epoch 298/300
1/1 [==============================] - 0s 18ms/stepG_loss: 0.9608 - F_loss: 0.8333 - D_X_loss: 0.683
2 - D_Y_loss: 0.6705 - Total_Generator_Loss: 1.79
1/1 [==============================] - 0s 16ms/step
```




```
86/86 [==============================] - 18s 213ms/step - G_loss: 0.9663 - F_loss: 0.8312 - D_X_loss
: 0.6828 - D_Y_loss: 0.6711 - Total_Generator_Loss: 1.7975
Epoch 299/300
1/1 [==============================] - 0s 16ms/stepG_loss: 0.9735 - F_loss: 0.8234 - D_X_loss: 0.683
1 - D_Y_loss: 0.6670 - Total_Generator_Loss: 1.79
1/1 [==============================] - 0s 15ms/step
```

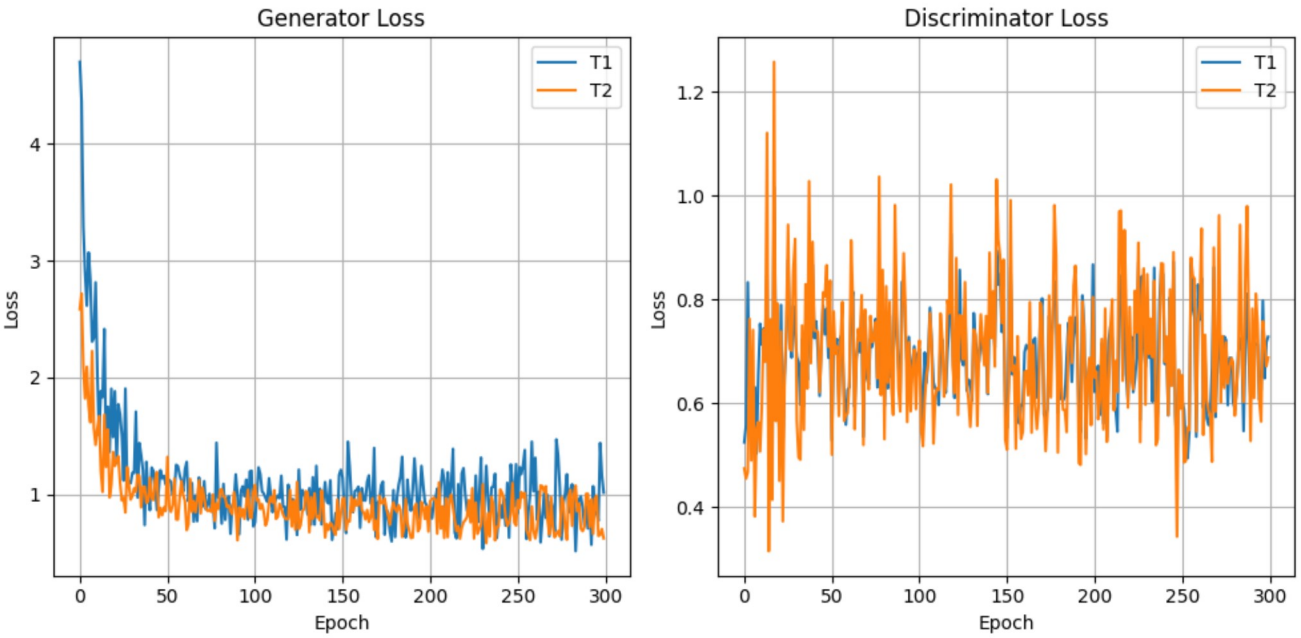



```
86/86 [==============================] - 18s 212ms/step - G_loss: 0.9757 - F_loss: 0.8220 - D_X_loss
: 0.6835 - D_Y_loss: 0.6670 - Total_Generator_Loss: 1.7977
Epoch 300/300
1/1 [==============================] - 0s 16ms/stepG_loss: 0.9634 - F_loss: 0.8193 - D_X_loss: 0.686
8 - D_Y_loss: 0.6729 - Total_Generator_Loss: 1.78
1/1 [==============================] - 0s 16ms/step
```




```
86/86 [==============================] - 18s 209ms/step - G_loss: 0.9640 - F_loss: 0.8171 - D_X_loss
: 0.6872 - D_Y_loss: 0.6731 - Total_Generator_Loss: 1.7811
```

In [45]:
```python
def plot_hist(hist):
    plt.figure(figsize=(10, 5))
    plt.subplot(121)
    plt.plot(hist.history["G_loss"])
    plt.plot(hist.history["F_loss"])
    plt.legend(["T1", "T2"])
    plt.title("Generator Loss")
    plt.ylabel("Loss")
    plt.xlabel("Epoch")
    plt.grid()
    plt.subplot(122)
    plt.plot(hist.history["D_X_loss"])
    plt.plot(hist.history["D_Y_loss"])
    plt.title("Discriminator Loss")
    plt.ylabel("Loss")
    plt.xlabel("Epoch")
    plt.grid()
    plt.legend(["T1", "T2"])
    plt.tight_layout()
    plt.show()


plot_hist(history)
```



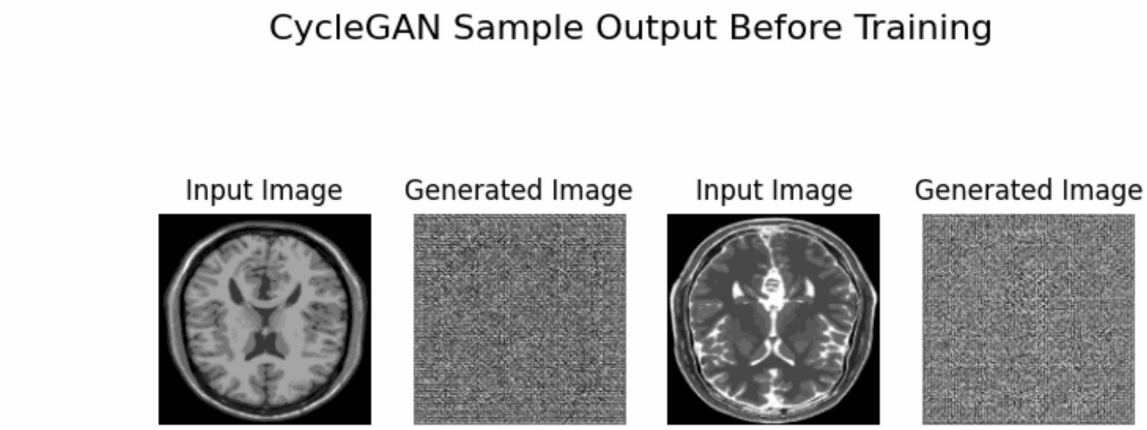
Checking the output of the model

In [47]:
```python
anim_file = "cyclegan_mri_pre.gif"
```

```
pro_img_path = "./IMG/"
filenames = [pro_img_path + i for i in os.listdir(pro_img_path)]
filenames = sorted(filenames, key=os.path.getmtime)
img_list = [imageio.imread(filename) for filename in filenames]
imageio.mimsave(anim_file, img_list, duration=2, loop=1)
```

In [48]:
```
import tensorflow_docs.vis.embed as embed

embed.embed_file(anim_file)
```

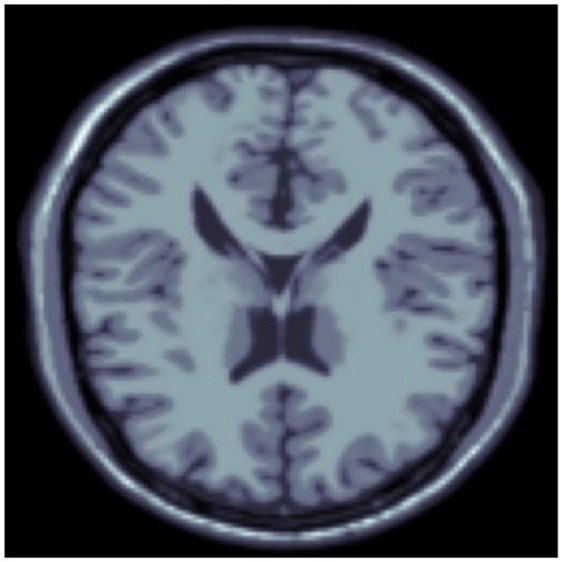
Out[48]:



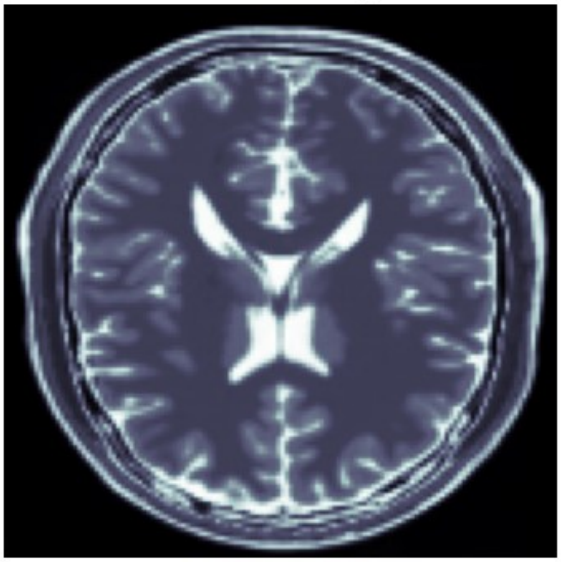
Above plot is output of **one epoch**

In [49]:
```
# evaluate the model for T1 to T2
_, ax = plt.subplots(4, 2, figsize=(10, 15))
for i, img in enumerate(t1_images.take(4)):
    prediction = cycle_gan_model.gen_G(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)

    ax[i, 0].imshow(img, cmap="bone")
    ax[i, 1].imshow(prediction, cmap="bone")
    ax[i, 0].set_title("Input image")
    ax[i, 0].set_title("Input image")
    ax[i, 1].set_title("Translated image")
    ax[i, 0].axis("off")
    ax[i, 1].axis("off")

    prediction = tf.keras.utils.array_to_img(prediction)
plt.tight_layout()
plt.show()
```

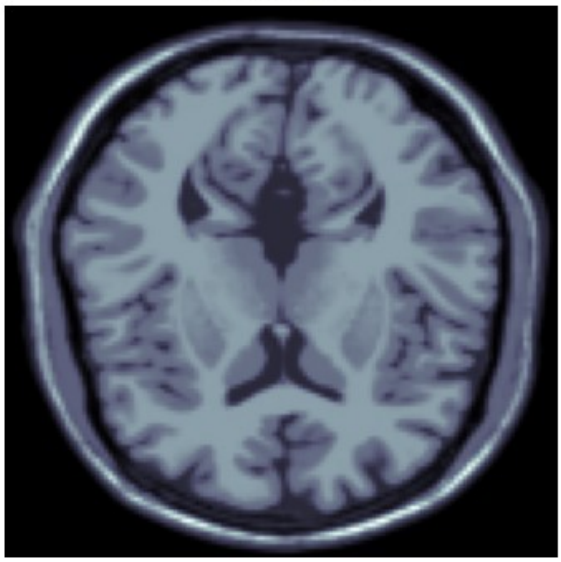
Input image



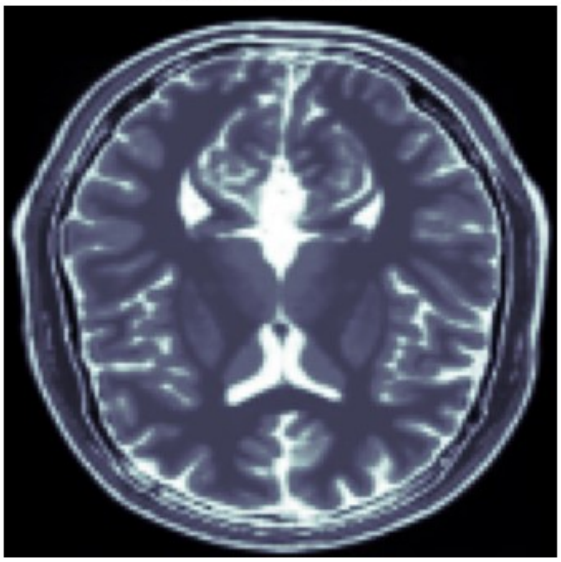
Translated image



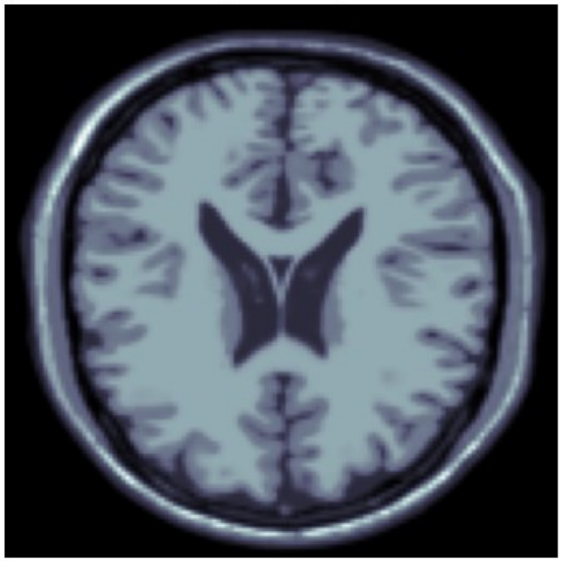
Input image



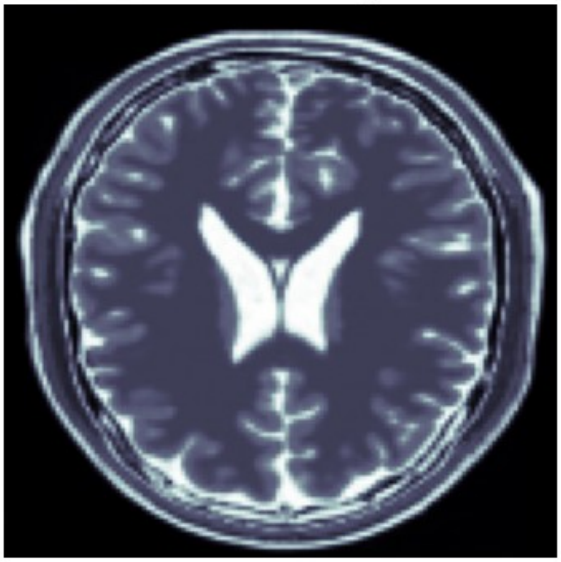
Translated image



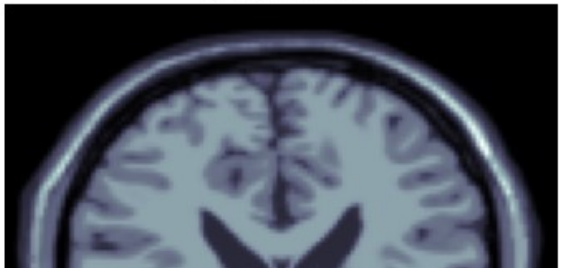
Input image



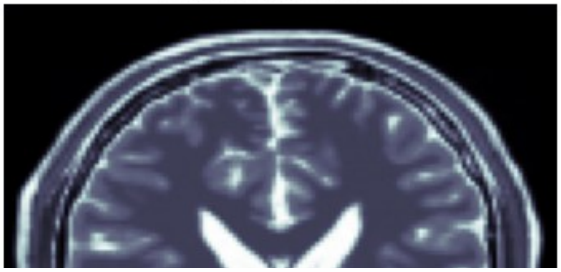
Translated image

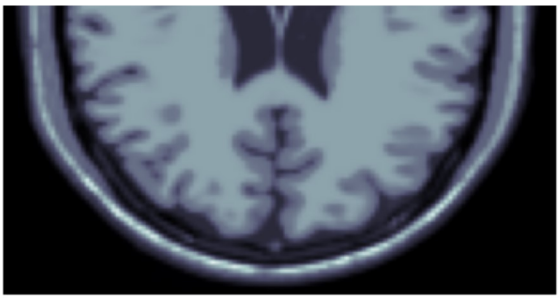


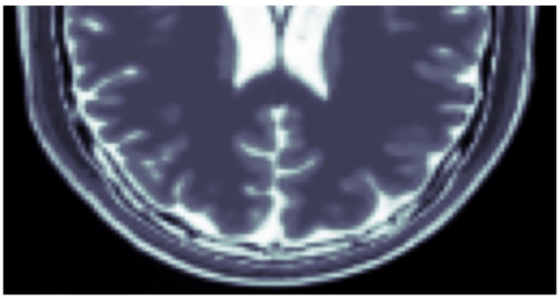
Input image



Translated image

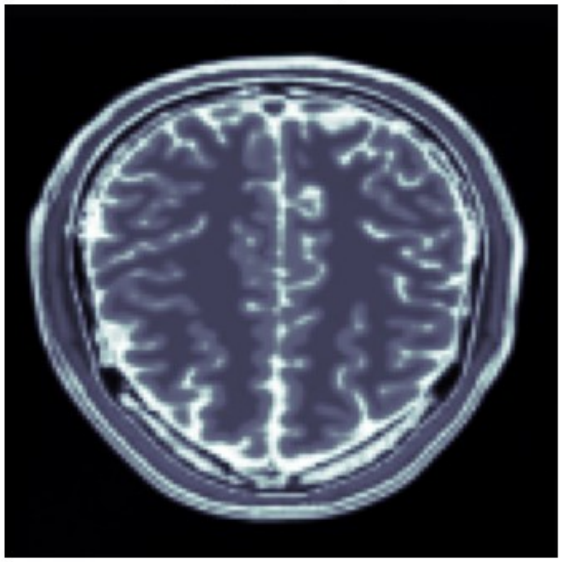




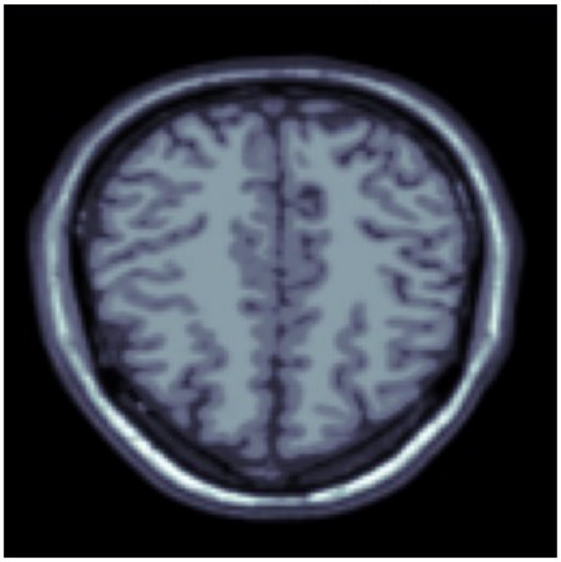


In [50]:
```
# evaluate the model for T2 to T1
_, ax = plt.subplots(4, 2, figsize=(10, 15))
for i, img in enumerate(t2_images.take(4)):
    prediction = cycle_gan_model.gen_F(img, training=False)[0].numpy()
    prediction = (prediction * 127.5 + 127.5).astype(np.uint8)
    img = (img[0] * 127.5 + 127.5).numpy().astype(np.uint8)

    ax[i, 0].imshow(img, cmap="bone")
    ax[i, 1].imshow(prediction, cmap="bone")
    ax[i, 0].set_title("Input image")
    ax[i, 0].set_title("Input image")
    ax[i, 1].set_title("Translated image")
    ax[i, 0].axis("off")
    ax[i, 1].axis("off")

    prediction = tf.keras.utils.array_to_img(prediction)
plt.tight_layout()
plt.show()
```

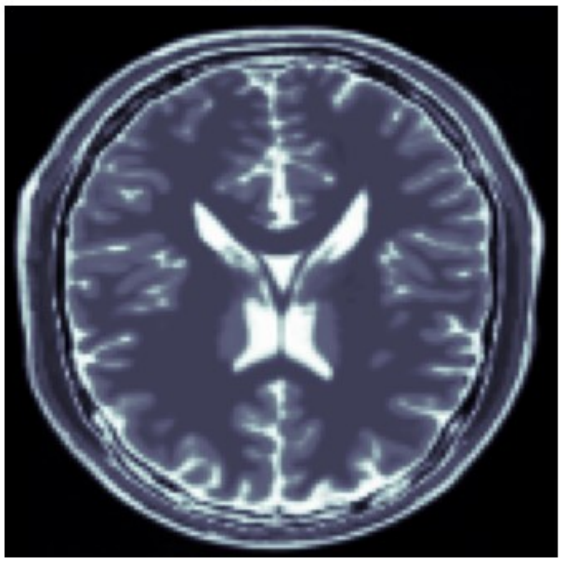
Input image



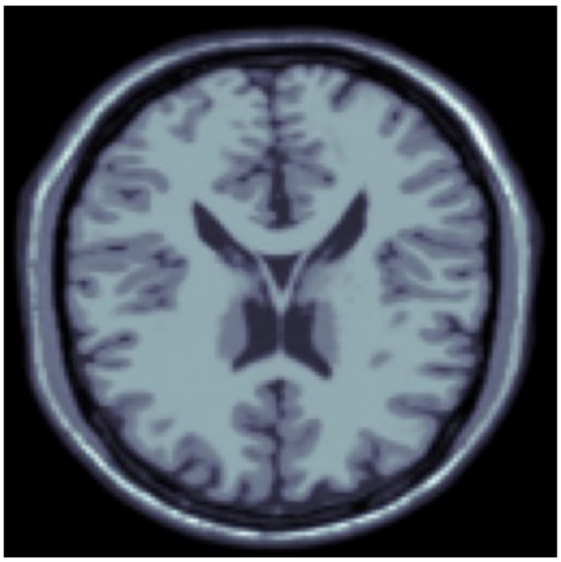
Translated image



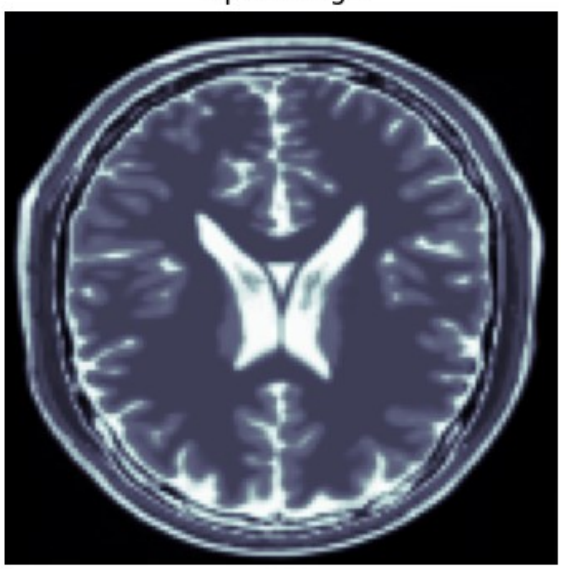
Input image



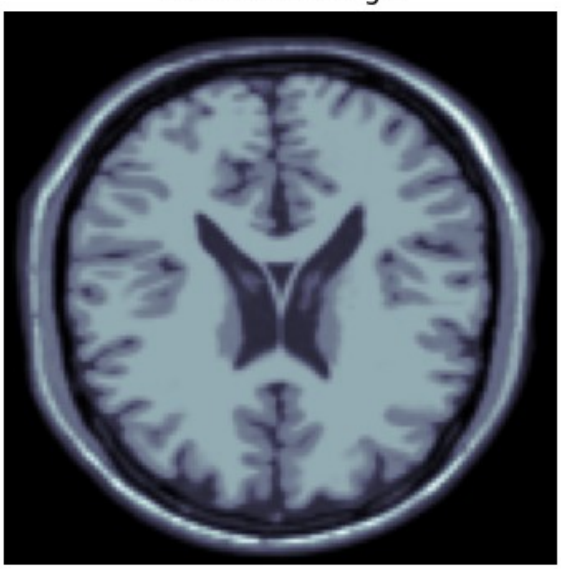
Translated image



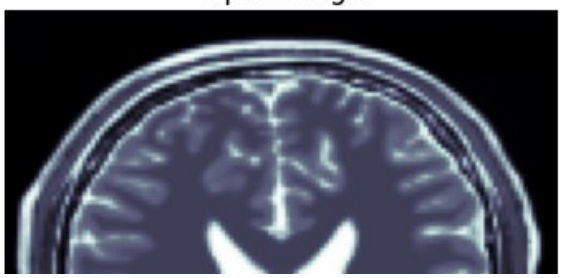
Input image



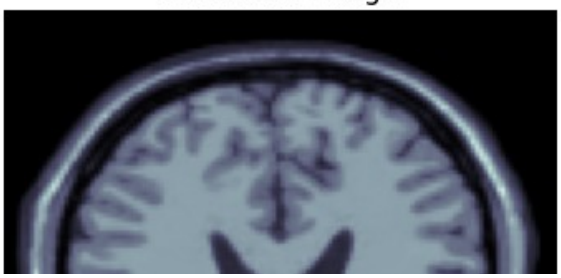
Translated image

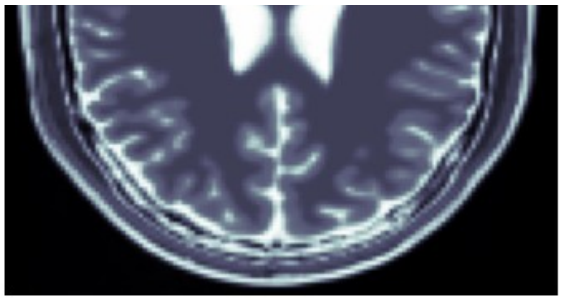


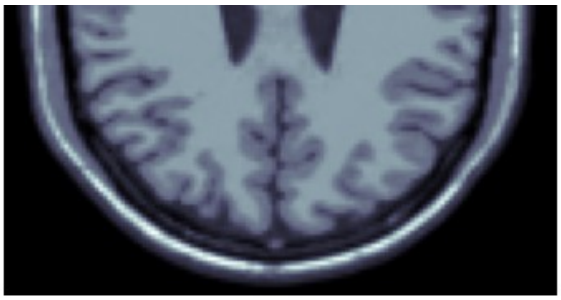
Input image



Translated image







https://www.mdpi.com/2077-1312/7/7/200

Conclusion :

This CycleGANs architectur has been effective in translating MRI images from T1 to T2 modalities without the need for paired data.

This demonstrates the model's ability to learn meaningful mappings between unpaired image domains, facilitating the synthesis of realistic T2-like images from T1 inputs and vice versa.

The translated T2 images exhibit a high degree of structural fidelity, indicating the successful preservation of anatomical features during the translation process.

References :

- Dataset : https://www.kaggle.com/datasets/awsaf49/brats20-dataset-training-validation/code
- https://www.mdpi.com/2077-1312/7/7/200
- https://www.geeksforgeeks.org/cycle-generative-adversarial-network-cyclegan-2/
- Most of the code is inspired from https://github.com/Hitha83/MRI-styletransfer-CycleGAN/tree/main