Technological Institute of the Philippines
938 Aurora Blvd., Cubao, Quezon City

College of Engineering and Architecture
Electronics Engineering Department

Excercise

Generative Adversarial Network

Submitted by:
Gabotero, Clive Jake A.
ECE41S1

Submitted to:
Engr. Christian Lian Paulo P. Rioflorido, MSEE

October 2022

It is the main goal in this activity to show an application of the Generative Adversarial Networks (GANs). Moreover, it is also serves to understand how this neural network works.

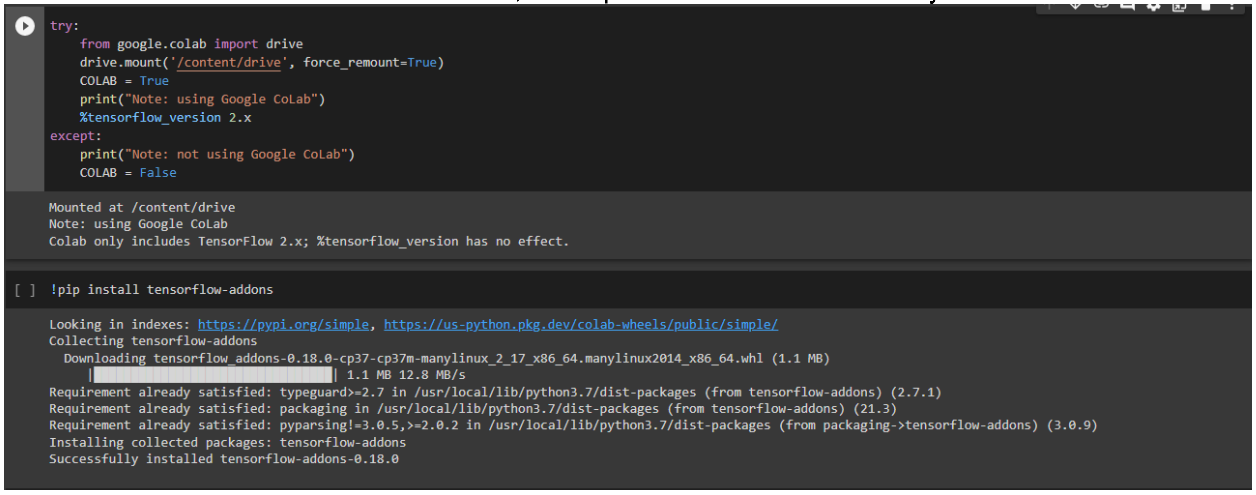
As the name implies, the Generative Adversarial Network is composed of two main parts, namely the generator network and the discriminator network. The generator's role is to generate a fake image out of a random noise initially and then passes it through the discriminator. The discriminator then assesses the image from the generator and decides if it is fake or real. As the evaluation comes out from the discriminator, the feedback is then taken again by the generator for it to be able to generate a better image. This process is called the backpropagation. These processes will continue to happen until the generator is able to produce an image that is deceiving enough to fool the discriminator that this image is real even it is a fake one.

In this activity, an application of Cycle GAN which is the image-to-image translation is demonstrated and images of horses and zebras were used as dataset. Cycle GAN uses the concept of unpaired data, meaning, the input image is not related to the label or the style of what the output would be. An example of this idea is the neural style transfer where an input is an image and the other signifies a style. The idea of using this type of GAN will be further elaborated in this activity.

Below is the code to implement the image-to-image GAN.

Image – to – image GAN:

To start the Generative Adversarial Network, it is important to load the necessary libraries.

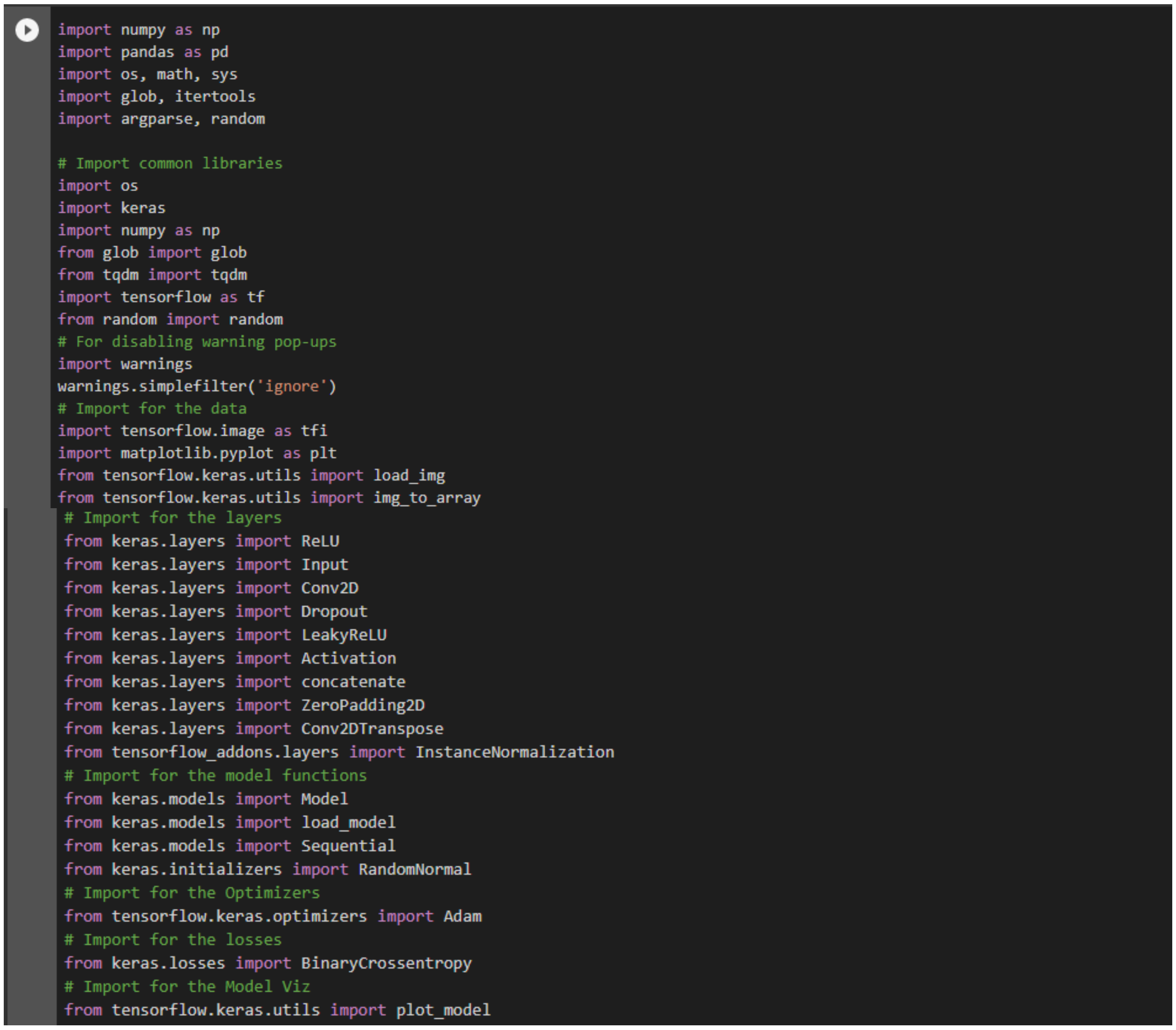


```
try:
    from google.colab import drive
    drive.mount('/content/drive', force_remount=True)
    COLAB = True
    print("Note: using Google CoLab")
    %tensorflow_version 2.x
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

```
Mounted at /content/drive
Note: using Google CoLab
Colab only includes TensorFlow 2.x; %tensorflow_version has no effect.
```

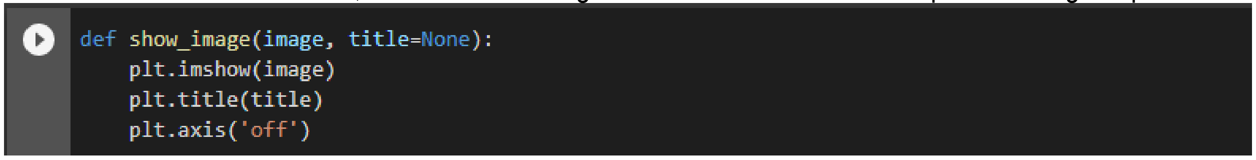
```
[ ] !pip install tensorflow-addons
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting tensorflow-addons
  Downloading tensorflow_addons-0.18.0-cp37-cp37m-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.1 MB)
     |████████████████████████████████| 1.1 MB 12.8 MB/s
Requirement already satisfied: typeguard>=2.7 in /usr/local/lib/python3.7/dist-packages (from tensorflow-addons) (2.7.1)
Requirement already satisfied: packaging in /usr/local/lib/python3.7/dist-packages (from tensorflow-addons) (21.3)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in /usr/local/lib/python3.7/dist-packages (from packaging->tensorflow-addons) (3.0.9)
Installing collected packages: tensorflow-addons
Successfully installed tensorflow-addons-0.18.0
```



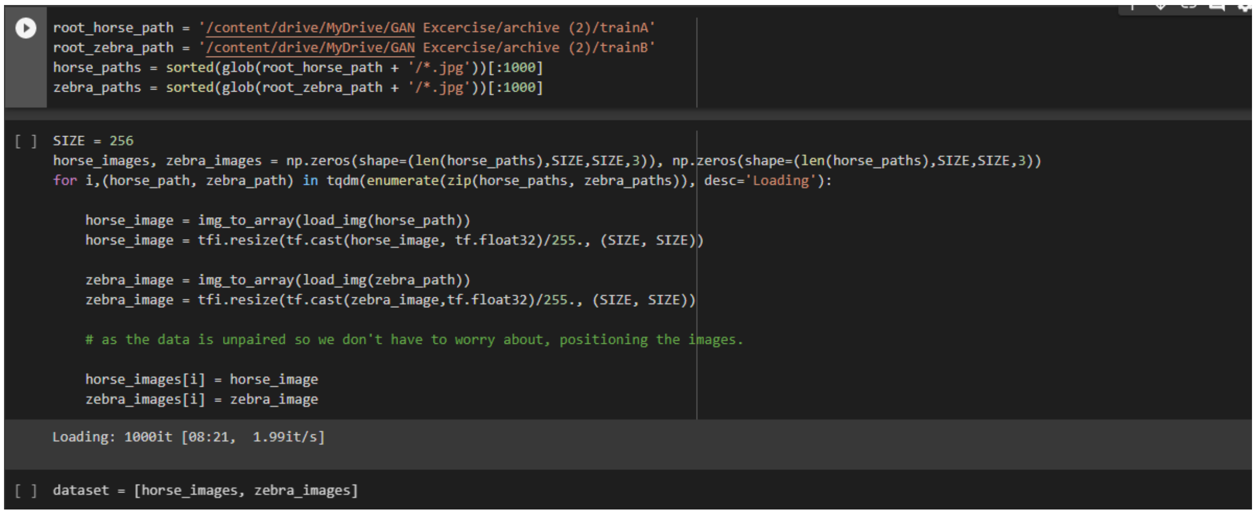
```python
import numpy as np
import pandas as pd
import os, math, sys
import glob, itertools
import argparse, random

# Import common libraries
import os
import keras
import numpy as np
from glob import glob
from tqdm import tqdm
import tensorflow as tf
from random import random
# For disabling warning pop-ups
import warnings
warnings.simplefilter('ignore')
# Import for the data
import tensorflow.image as tfi
import matplotlib.pyplot as plt
from tensorflow.keras.utils import load_img
from tensorflow.keras.utils import img_to_array
# Import for the layers
from keras.layers import ReLU
from keras.layers import Input
from keras.layers import Conv2D
from keras.layers import Dropout
from keras.layers import LeakyReLU
from keras.layers import Activation
from keras.layers import concatenate
from keras.layers import ZeroPadding2D
from keras.layers import Conv2DTranspose
from tensorflow_addons.layers import InstanceNormalization
# Import for the model functions
from keras.models import Model
from keras.models import load_model
from keras.models import Sequential
from keras.initializers import RandomNormal
# Import for the Optimizers
from tensorflow.keras.optimizers import Adam
# Import for the losses
from keras.losses import BinaryCrossentropy
# Import for the Model Viz
from tensorflow.keras.utils import plot_model
```

In these next lines of code, functions are being customized so that it can be plotted using Matplotlib.



```python
def show_image(image, title=None):
    plt.imshow(image)
    plt.title(title)
    plt.axis('off')
```

Since Google Colab is being used in this activity, the dataset is uploaded to google drive where it can be imported using the next lines of codes.



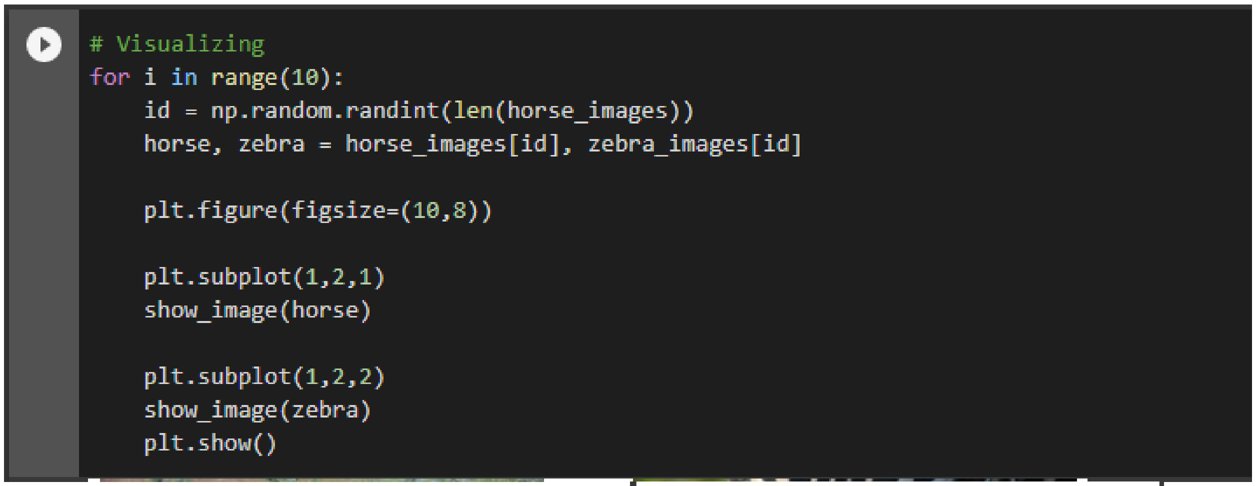
```
root_horse_path = '/content/drive/MyDrive/GAN Excercise/archive (2)/trainA'
root_zebra_path = '/content/drive/MyDrive/GAN Excercise/archive (2)/trainB'
horse_paths = sorted(glob(root_horse_path + '/*.jpg'))[:1000]
zebra_paths = sorted(glob(root_zebra_path + '/*.jpg'))[:1000]
```

```
SIZE = 256
horse_images, zebra_images = np.zeros(shape=(len(horse_paths),SIZE,SIZE,3)), np.zeros(shape=(len(horse_paths),SIZE,SIZE,3))
for i,(horse_path, zebra_path) in tqdm(enumerate(zip(horse_paths, zebra_paths)), desc='Loading'):

    horse_image = img_to_array(load_img(horse_path))
    horse_image = tfi.resize(tf.cast(horse_image, tf.float32)/255., (SIZE, SIZE))

    zebra_image = img_to_array(load_img(zebra_path))
    zebra_image = tfi.resize(tf.cast(zebra_image,tf.float32)/255., (SIZE, SIZE))

    # as the data is unpaired so we don't have to worry about, positioning the images.

    horse_images[i] = horse_image
    zebra_images[i] = zebra_image
```

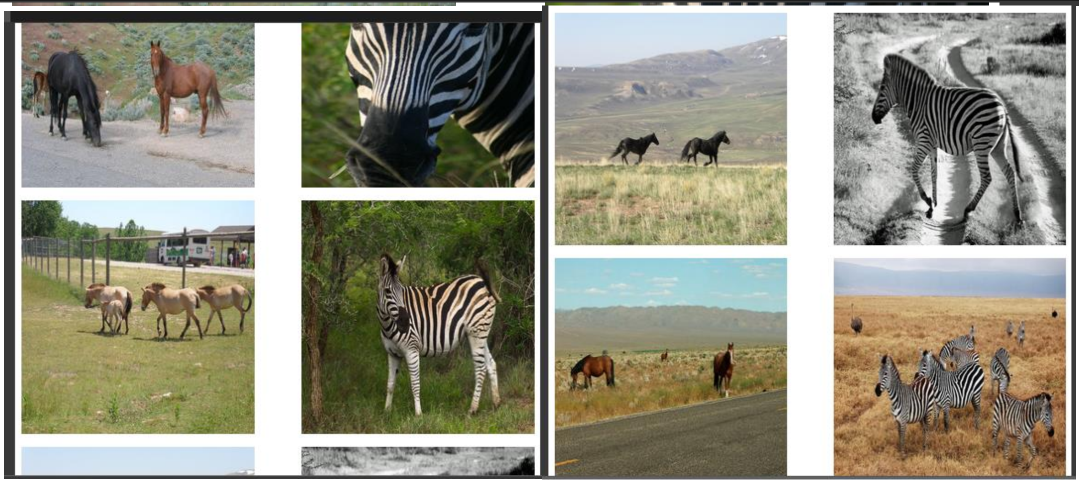
```
Loading: 1000it [08:21,  1.99it/s]
```

```
dataset = [horse_images, zebra_images]
```

Now we visualize the images to see what they look like.

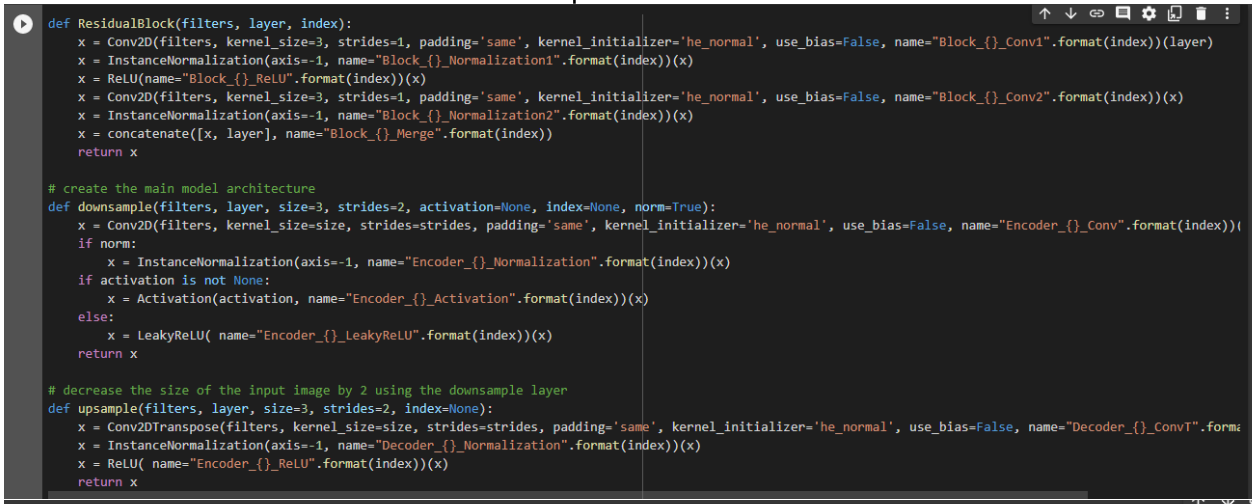


```
# Visualizing
for i in range(10):
    id = np.random.randint(len(horse_images))
    horse, zebra = horse_images[id], zebra_images[id]

    plt.figure(figsize=(10,8))

    plt.subplot(1,2,1)
    show_image(horse)

    plt.subplot(1,2,2)
    show_image(zebra)
    plt.show()
```

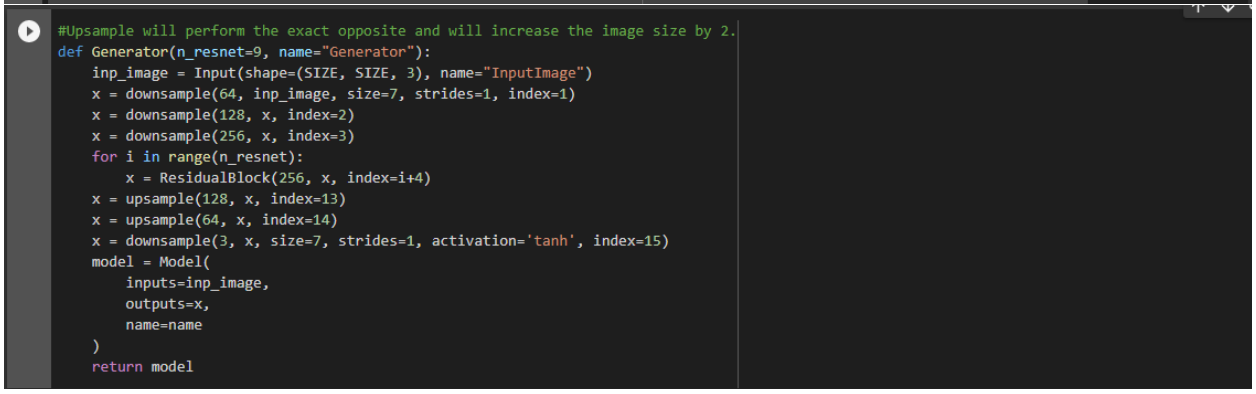


From the lines of codes above, we have successfully plotted and checked the images that we have loaded. The images plotted above shows that there are no direct relations between the appearances of horses and zebras.

The next lines of codes below will be for the generator network. Here, the Generator network is built using residual blocks. This residual block is being designed so that the network will be able to learn the features of the input image. The residual block will contain 2 set convolutional layer, instance normalization, and a skip connector.

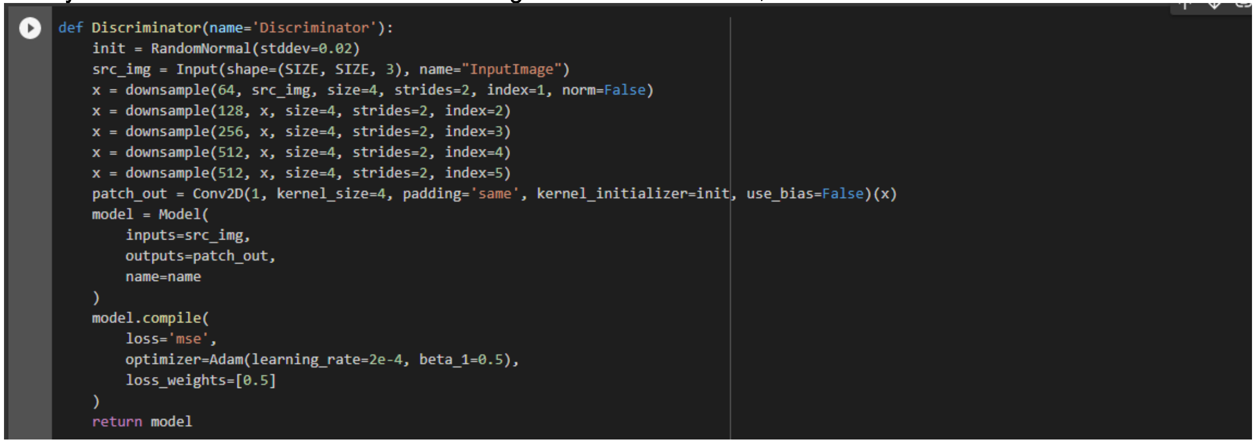


```python
def ResidualBlock(filters, layer, index):
    x = Conv2D(filters, kernel_size=3, strides=1, padding='same', kernel_initializer='he_normal', use_bias=False, name="Block_{}_Conv1".format(index))(layer)
    x = InstanceNormalization(axis=-1, name="Block_{}_Normalization1".format(index))(x)
    x = ReLU(name="Block_{}_ReLU".format(index))(x)
    x = Conv2D(filters, kernel_size=3, strides=1, padding='same', kernel_initializer='he_normal', use_bias=False, name="Block_{}_Conv2".format(index))(x)
    x = InstanceNormalization(axis=-1, name="Block_{}_Normalization2".format(index))(x)
    x = concatenate([x, layer], name="Block_{}_Merge".format(index))
    return x

# create the main model architecture
def downsample(filters, layer, size=3, strides=2, activation=None, index=None, norm=True):
    x = Conv2D(filters, kernel_size=size, strides=strides, padding='same', kernel_initializer='he_normal', use_bias=False, name="Encoder_{}_Conv".format(index))(
    if norm:
        x = InstanceNormalization(axis=-1, name="Encoder_{}_Normalization".format(index))(x)
    if activation is not None:
        x = Activation(activation, name="Encoder_{}_Activation".format(index))(x)
    else:
        x = LeakyReLU( name="Encoder_{}_LeakyReLU".format(index))(x)
    return x

# decrease the size of the input image by 2 using the downsample layer
def upsample(filters, layer, size=3, strides=2, index=None):
    x = Conv2DTranspose(filters, kernel_size=size, strides=strides, padding='same', kernel_initializer='he_normal', use_bias=False, name="Decoder_{}_ConvT".forma
    x = InstanceNormalization(axis=-1, name="Decoder_{}_Normalization".format(index))(x)
    x = ReLU( name="Encoder_{}_ReLU".format(index))(x)
    return x
```



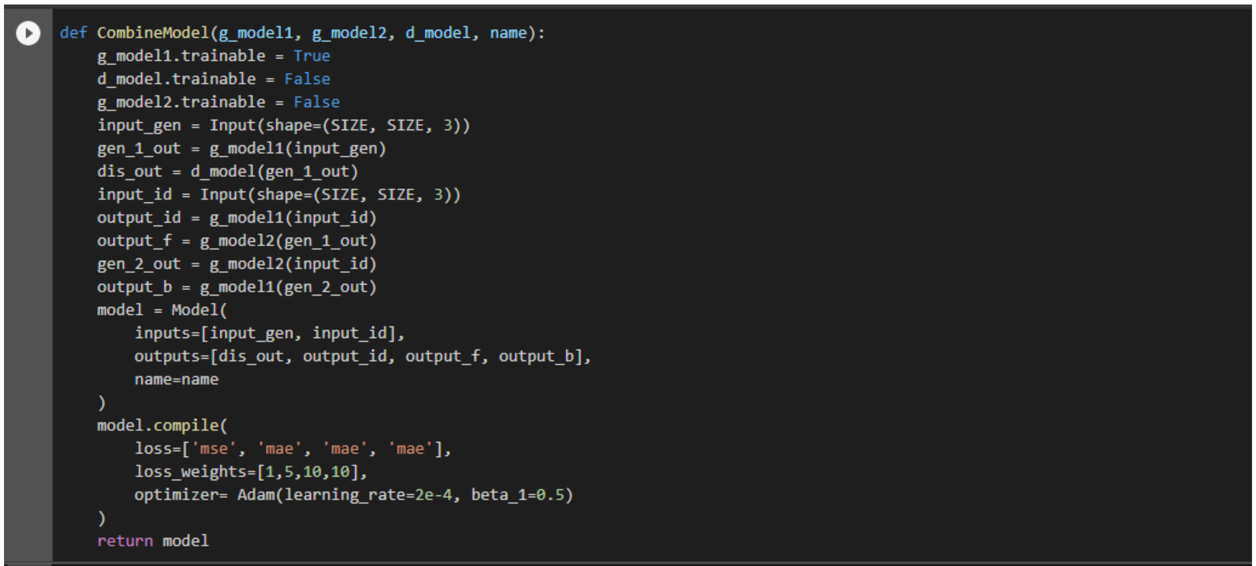
```python
#Upsample will perform the exact opposite and will increase the image size by 2.
def Generator(n_resnet=9, name="Generator"):
    inp_image = Input(shape=(SIZE, SIZE, 3), name="InputImage")
    x = downsample(64, inp_image, size=7, strides=1, index=1)
    x = downsample(128, x, index=2)
    x = downsample(256, x, index=3)
    for i in range(n_resnet):
        x = ResidualBlock(256, x, index=i+4)
    x = upsample(128, x, index=13)
    x = upsample(64, x, index=14)
    x = downsample(3, x, size=7, strides=1, activation='tanh', index=15)
    model = Model(
        inputs=inp_image,
        outputs=x,
        name=name
    )
    return model
```

Moving to the discriminator network, steps are exactly the same with that of the generator network. The only difference now is that instead of using batch normalization, we need to use instance normalization.

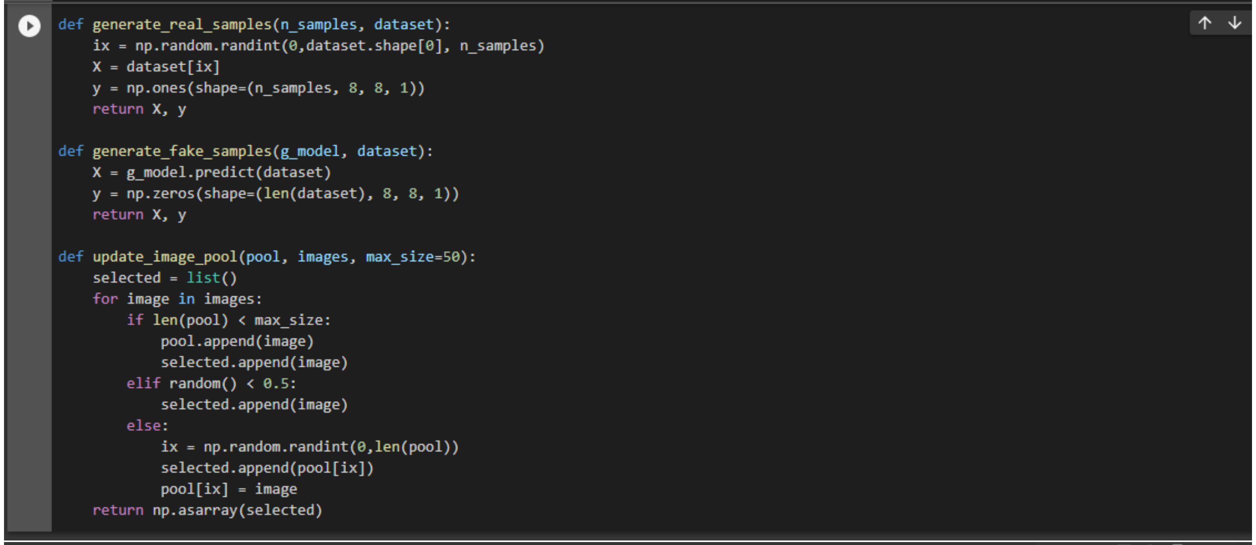


```python
def Discriminator(name='Discriminator'):
    init = RandomNormal(stddev=0.02)
    src_img = Input(shape=(SIZE, SIZE, 3), name="InputImage")
    x = downsample(64, src_img, size=4, strides=2, index=1, norm=False)
    x = downsample(128, x, size=4, strides=2, index=2)
    x = downsample(256, x, size=4, strides=2, index=3)
    x = downsample(512, x, size=4, strides=2, index=4)
    x = downsample(512, x, size=4, strides=2, index=5)
    patch_out = Conv2D(1, kernel_size=4, padding='same', kernel_initializer=init, use_bias=False)(x)
    model = Model(
        inputs=src_img,
        outputs=patch_out,
        name=name
    )
    model.compile(
        loss='mse',
        optimizer=Adam(learning_rate=2e-4, beta_1=0.5),
        loss_weights=[0.5]
    )
    return model
```

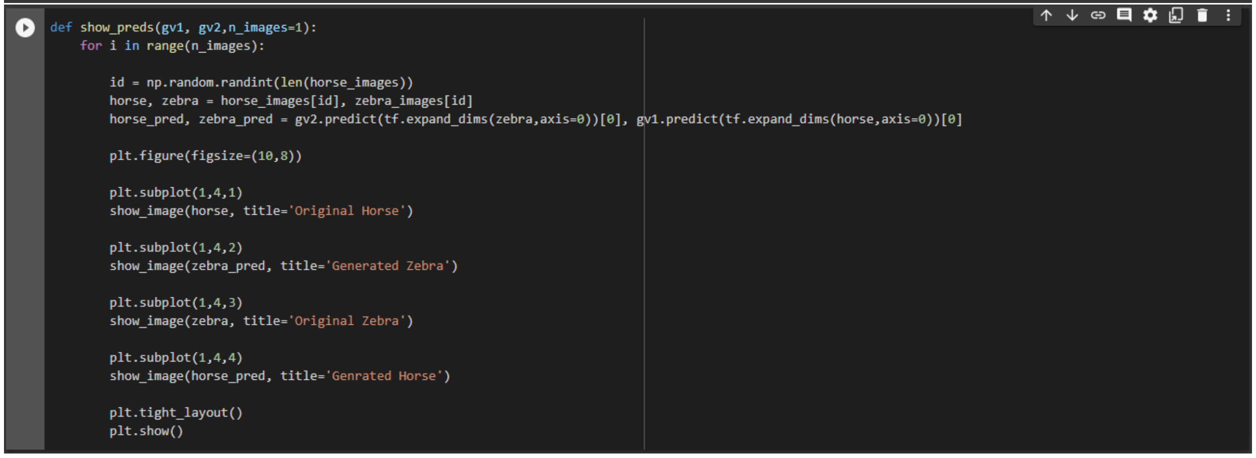
The next processes combines the generator network and the discriminator network which will be essential in training the whole model.



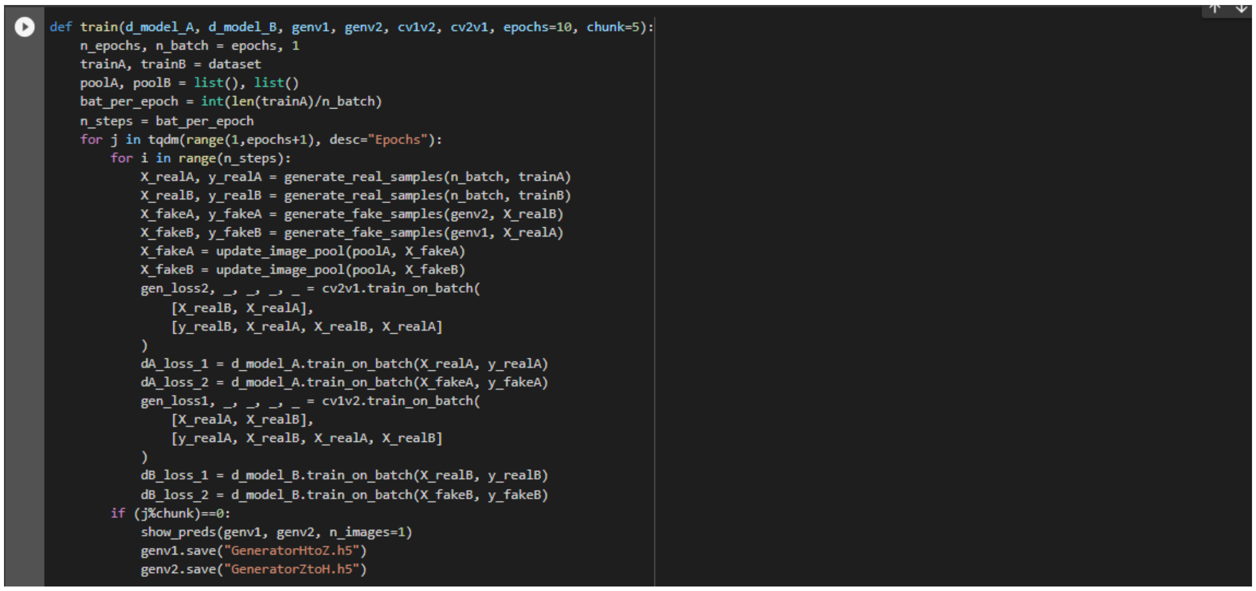
```python
def CombineModel(g_model1, g_model2, d_model, name):
    g_model1.trainable = True
    d_model.trainable = False
    g_model2.trainable = False
    input_gen = Input(shape=(SIZE, SIZE, 3))
    gen_1_out = g_model1(input_gen)
    dis_out = d_model(gen_1_out)
    input_id = Input(shape=(SIZE, SIZE, 3))
    output_id = g_model1(input_id)
    output_f = g_model2(gen_1_out)
    gen_2_out = g_model2(input_id)
    output_b = g_model1(gen_2_out)
    model = Model(
        inputs=[input_gen, input_id],
        outputs=[dis_out, output_id, output_f, output_b],
        name=name
    )
    model.compile(
        loss=['mse', 'mae', 'mae', 'mae'],
        loss_weights=[1,5,10,10],
        optimizer= Adam(learning_rate=2e-4, beta_1=0.5)
    )
    return model
```



```python
def generate_real_samples(n_samples, dataset):
    ix = np.random.randint(0,dataset.shape[0], n_samples)
    X = dataset[ix]
    y = np.ones(shape=(n_samples, 8, 8, 1))
    return X, y

def generate_fake_samples(g_model, dataset):
    X = g_model.predict(dataset)
    y = np.zeros(shape=(len(dataset), 8, 8, 1))
    return X, y

def update_image_pool(pool, images, max_size=50):
    selected = list()
    for image in images:
        if len(pool) < max_size:
            pool.append(image)
            selected.append(image)
        elif random() < 0.5:
            selected.append(image)
        else:
            ix = np.random.randint(0,len(pool))
            selected.append(pool[ix])
            pool[ix] = image
    return np.asarray(selected)
```

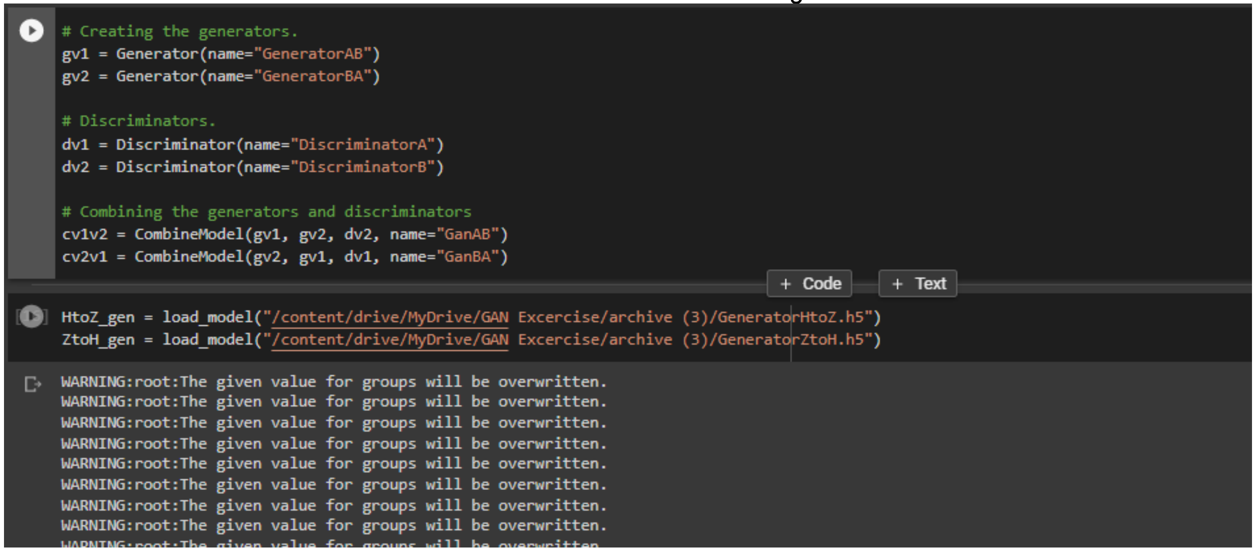


```python
def show_preds(gv1, gv2,n_images=1):
    for i in range(n_images):

        id = np.random.randint(len(horse_images))
        horse, zebra = horse_images[id], zebra_images[id]
        horse_pred, zebra_pred = gv2.predict(tf.expand_dims(zebra,axis=0))[0], gv1.predict(tf.expand_dims(horse,axis=0))[0]

        plt.figure(figsize=(10,8))

        plt.subplot(1,4,1)
        show_image(horse, title='Original Horse')

        plt.subplot(1,4,2)
        show_image(zebra_pred, title='Generated Zebra')

        plt.subplot(1,4,3)
        show_image(zebra, title='Original Zebra')

        plt.subplot(1,4,4)
        show_image(horse_pred, title='Genrated Horse')

        plt.tight_layout()
        plt.show()
```



```
def train(d_model_A, d_model_B, genv1, genv2, cv1v2, cv2v1, epochs=10, chunk=5):
    n_epochs, n_batch = epochs, 1
    trainA, trainB = dataset
    poolA, poolB = list(), list()
    bat_per_epoch = int(len(trainA)/n_batch)
    n_steps = bat_per_epoch
    for j in tqdm(range(1,epochs+1), desc="Epochs"):
        for i in range(n_steps):
            X_realA, y_realA = generate_real_samples(n_batch, trainA)
            X_realB, y_realB = generate_real_samples(n_batch, trainB)
            X_fakeA, y_fakeA = generate_fake_samples(genv2, X_realB)
            X_fakeB, y_fakeB = generate_fake_samples(genv1, X_realA)
            X_fakeA = update_image_pool(poolA, X_fakeA)
            X_fakeB = update_image_pool(poolA, X_fakeB)
            gen_loss2, _, _, _, _ = cv2v1.train_on_batch(
                [X_realB, X_realA],
                [y_realB, X_realA, X_realB, X_realA]
            )
            dA_loss_1 = d_model_A.train_on_batch(X_realA, y_realA)
            dA_loss_2 = d_model_A.train_on_batch(X_fakeA, y_fakeA)
            gen_loss1, _, _, _, _ = cv1v2.train_on_batch(
                [X_realA, X_realB],
                [y_realA, X_realB, X_realA, X_realB]
            )
            dB_loss_1 = d_model_B.train_on_batch(X_realB, y_realB)
            dB_loss_2 = d_model_B.train_on_batch(X_fakeB, y_fakeB)
        if (j%chunk)==0:
            show_preds(genv1, genv2, n_images=1)
            genv1.save("GeneratorHtoZ.h5")
            genv2.save("GeneratorZtoH.h5")
```

Lastly, these next lines of codes will process the image translation after the training. The flow of processes will loop through the model since it is characterized by the generator and the discriminator networks being combined to make the whole final algorithm.



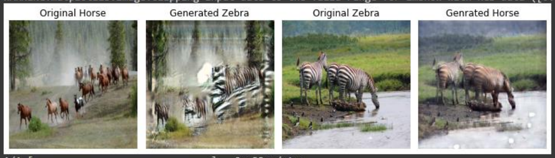
```
# Creating the generators.
gv1 = Generator(name="GeneratorAB")
gv2 = Generator(name="GeneratorBA")

# Discriminators.
dv1 = Discriminator(name="DiscriminatorA")
dv2 = Discriminator(name="DiscriminatorB")

# Combining the generators and discriminators
cv1v2 = CombineModel(gv1, gv2, dv2, name="GanAB")
cv2v1 = CombineModel(gv2, gv1, dv1, name="GanBA")
```

```
HtoZ_gen = load_model("/content/drive/MyDrive/GAN Excercise/archive (3)/GeneratorHtoZ.h5")
ZtoH_gen = load_model("/content/drive/MyDrive/GAN Excercise/archive (3)/GeneratorZtoH.h5")
```

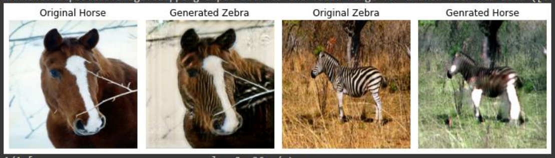
```
WARNING:root:The given value for groups will be overwritten.
WARNING:root:The given value for groups will be overwritten.
WARNING:root:The given value for groups will be overwritten.
WARNING:root:The given value for groups will be overwritten.
WARNING:root:The given value for groups will be overwritten.
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.
```

```
show_preds(HtoZ_gen, ZtoH_gen, n_images=5)
```

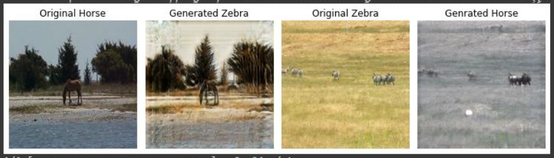
```
1/1 [==============================] - 11s 11s/step
1/1 [==============================] - 1s 881ms/step
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```



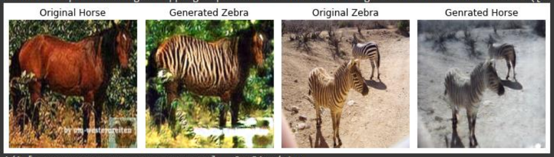
```
1/1 [==============================] - 0s 33ms/step
1/1 [==============================] - 0s 19ms/step
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```



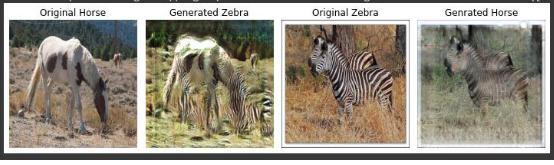
```
1/1 [==============================] - 0s 36ms/step
1/1 [==============================] - 0s 19ms/step
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```



```
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 20ms/step
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```

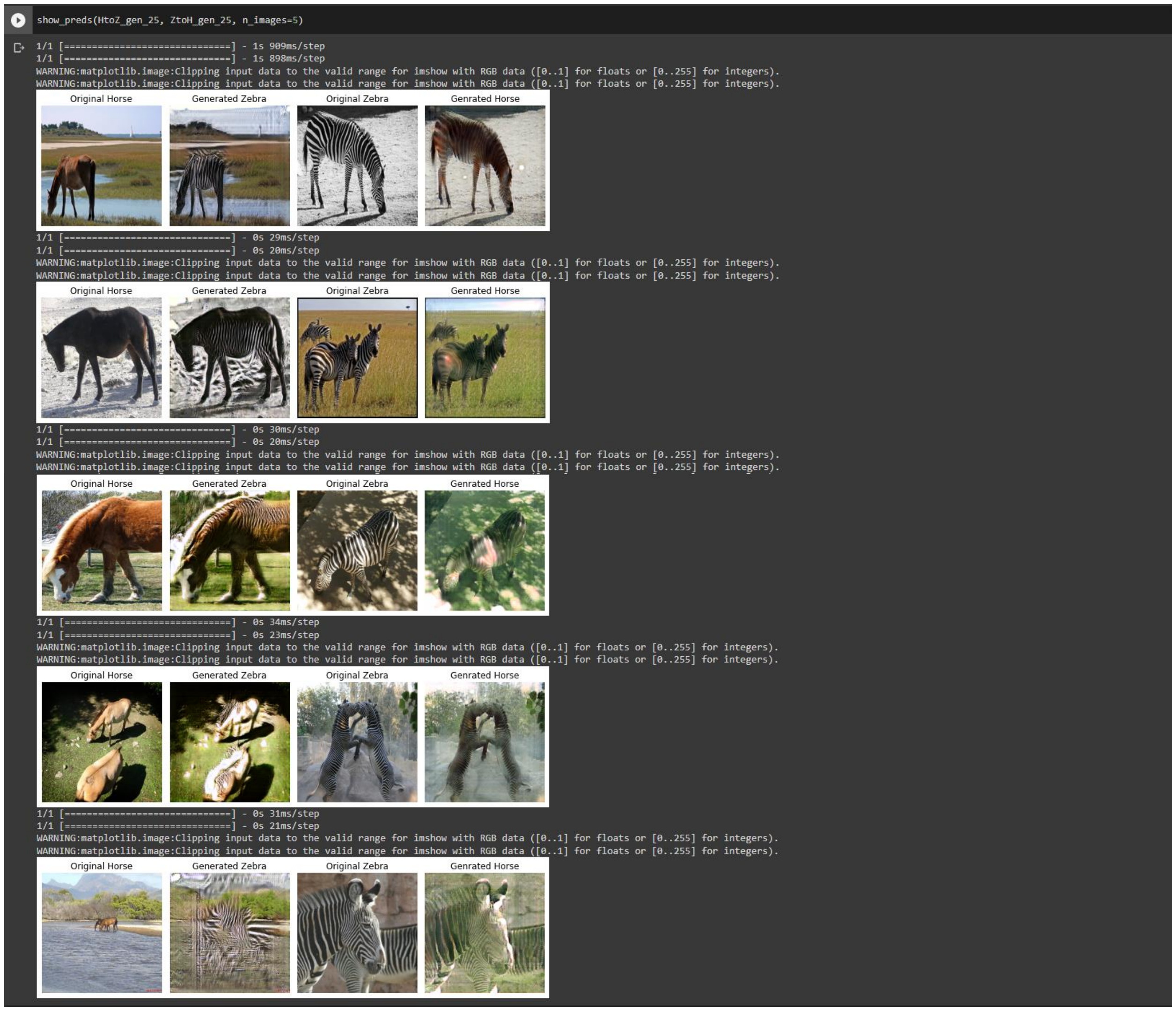


```
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 19ms/step
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```



```
HtoZ_gen_25 = load_model("/content/drive/MyDrive/GAN Excercise/archive (3)/GeneratorHtoZ_25.h5")
ZtoH_gen_25 = load_model("/content/drive/MyDrive/GAN Excercise/archive (3)/GeneratorZtoH_25.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: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.
```



```
show_preds(HtoZ_gen_25, ZtoH_gen_25, n_images=5)
```

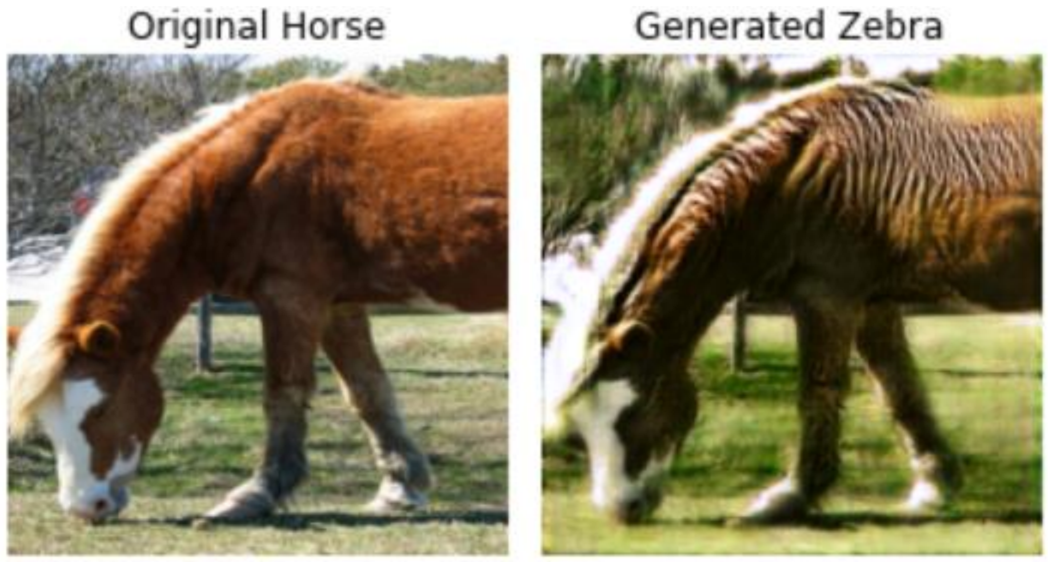
```
1/1 [==============================] - 1s 909ms/step
1/1 [==============================] - 1s 898ms/step
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 20ms/step
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 20ms/step
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 23ms/step
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 21ms/step
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```

Conclusion:

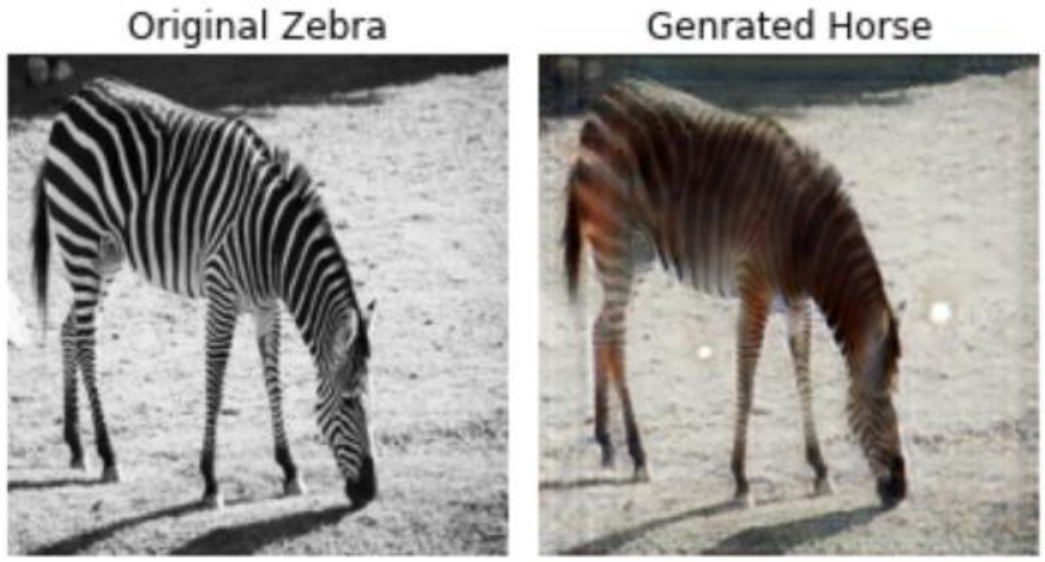
The application of the Cycle GAN in this activity helped me understand ways how wide can machine learning be applied specially in images. Through this, images can be edited to anything or any style for it to be implemented. In this activity, the model was able to make a horse look like a zebra, and a zebra to look like a horse, though the processing of the images were not that good. Still, it is a remarkable experience to observe how neural networks can process anything.

The images shown below are some of the outputs that were obtained from this activity.

Horse into Zebra



Zebra into Horse



References:

1. https://www.kaggle.com/code/utkarshsaxenadn/image-to-image-translation-cycle-gan/notebook - Code reference and dataset
2. https://stackoverflow.com/ - Errors, Buggs, and Fixes.