Configuration Manual: Generating MRI images using style transfer learning

MSc Research Project
Master of Science in Data Analytics

Dnyaneshwari Mahajan
Student ID: X20191138

School of Computing
National College of Ireland

Supervisor: Qurrat Ul Ain

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Dnyaneshwari Mahajan
Student ID:	X20191138
Programme:	Master of Science in Data Analytics
Year:	2022-23
Module:	MSc Research Project
Supervisor:	Qurrat UI Ain
Submission Due Date:	15/12/2023
Project Title:	Configuration Manual: Generating MRI images using style transfer learning
Word Count:	633
Page Count:	12

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Dnyaneshwari Mahajan
Date:	24th January 2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual: Generating MRI images using style transfer learning

Dnyaneshwari Mahajan
X20191138

1 Introduction

The configuration manual demonstrates the process used to code the project "Generating MRI images using Style transfer learning". To build a model CycleGAN framework and U-Net-based segmentation of deep learning are used. This manual configuration contains the hardware requirements, software requirements, and steps of implementation.

2 Hardware and software configuration

2.1 Hardware configuration

The below figure.1 shows the hardware configuration used to run the code.

Figure 1: Hardware configuration

2.2 software configuration

The Google colab pro subscription is taken because running the program requires more system RAM, GPU RAM, and disk space.

The below figure.2 shows the software configuration used to run the code.

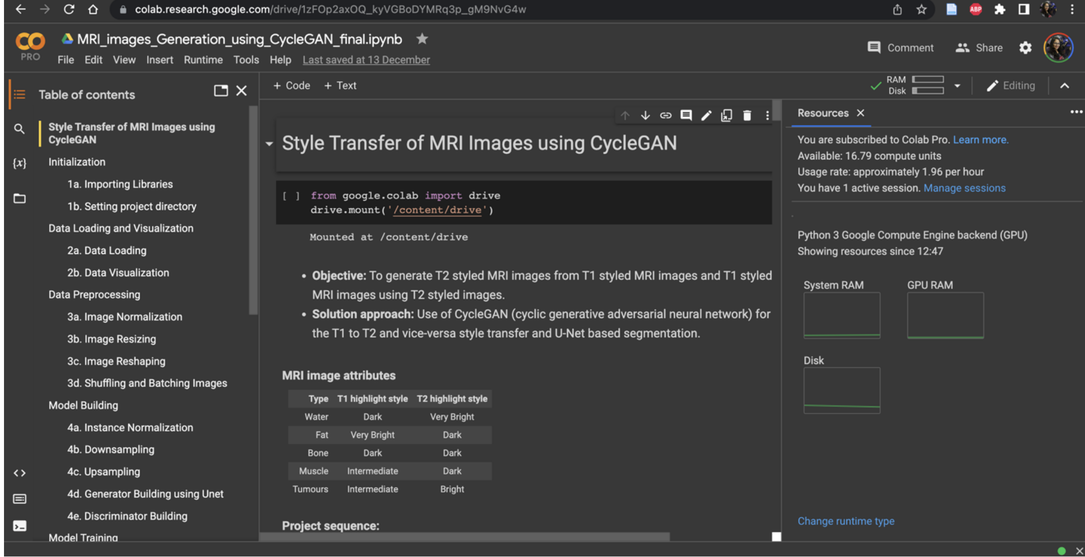

Figure 2: software configuration

3 Data Preparation

The dataset is collected from the GitHub library [1] and was available in.RAR format which is unzipped and uploaded to google drive. The dataset contains 2 sub-datasets of T1-styled MRI images and T2-styled MRI images in .png format. After uploading the dataset, it is divided into train and test datasets as shown in figure.3.

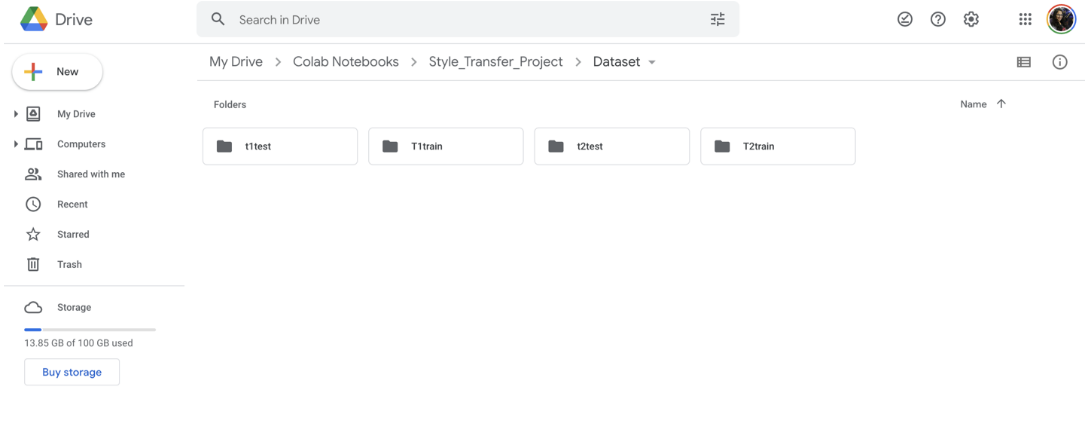

Figure 3: Dataset

- Importing Libraries: First, the required libraries are imported as shown in the figure.4.
- Setting project directory and Data Loading: The drive is then mounted in the program then Changing the working directory to target_path (Mode_Training folder) and then datasets are loaded in respective variables as shown in figure.5,figure.6.

[1]https://github.com/hackassin/Brain-MRI-Style-Transfer-With-GAN

1. Initialization

1a. Importing Libraries
1b. Setting project directory

1a. Importing Libraries

```
[ ] # Importing required libraries
    import numpy as np
    import tensorflow as tf

    import matplotlib.pyplot as plt
    %matplotlib inline
    import imageio
    from skimage.transform import resize
    from keras.utils.vis_utils import plot_model

    import os
    import shutil
    from glob import glob
```

```
[ ] # Installing tensorflow
    !pip install git+https://github.com/tensorflow/docs

    Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
```

Figure 4: Importing Libraries

1b. Setting project directory

This project is run on Google Colab

```
[ ] # Mounting google drive - required if code is run on Google Colab
    from google.colab import drive
    drive.mount('/content/drive')

    Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
[ ] # Defining target path
    target_path = '/content/drive/MyDrive/Colab Notebooks/Style_Transfer_Project/Model_Training/'

    # Defining version folder
    version_folder = 'V5D' #version of experiment
```

```
[ ] # Changing the working directory to target_path (Model_Training folder)
    os.chdir(target_path)

    # Checking working directory after change
    !pwd

    /content/drive/MyDrive/Colab Notebooks/Style_Transfer_Project/Model_Training
```

```
[ ] # Changing working directory to version folder
    if os.path.isdir(version_folder) == True:
      shutil.rmtree(os.path.join(target_path, version_folder))
      os.mkdir(version_folder)
      os.chdir(os.path.join(target_path, version_folder))

    else:
      os.mkdir(version_folder)
      os.chdir(os.path.join(target_path, version_folder))

    # Checking the current working directory after change
    !pwd
```

Figure 5: Setting project directory

2a. Data Loading

```
# Creating a function to load images from the specified folder path
def image_loader(folderpath):
  images_array = []
  for filepath in glob(folderpath + '*'):
    img = imageio.imread(filepath)
    images_array.append(img)
  images_array = np.array(images_array)
  return images_array
```

```
# Defining path for loading T1 and T2 images
T1_train_path = '/content/drive/MyDrive/Colab Notebooks/Style_Transfer_Project/Dataset/T1train/' #T1 images folderpath
T2_train_path = '/content/drive/MyDrive/Colab Notebooks/Style_Transfer_Project/Dataset/T2train/' #T2 images folderpath
T1_test_path = '/content/drive/MyDrive/Colab Notebooks/Style_Transfer_Project/Dataset/t1test/' #T1 images folderpath
T2_test_path = '/content/drive/MyDrive/Colab Notebooks/Style_Transfer_Project/Dataset/t2test/' #T2 images folderpath
```

```
# Loading images
T1_trimages = image_loader(T1_train_path) #Loading T1 train images
T1_teimages = image_loader(T1_test_path) #Loading T1 train images
T2_trimages = image_loader(T2_train_path) #Loading T2 train images
T2_teimages = image_loader(T2_test_path) #Loading T2 test images
```

```
# Checking shapes of the loaded image datasets
print('Shape of T1 training images: ', T1_trimages.shape)
print('Shape of T2 training images: ', T2_trimages.shape)
print('Shape of T1 testing images: ', T1_teimages.shape)
print('Shape of T2 testing images: ', T2_teimages.shape)
```

```
Shape of T1 training images:  (33, 217, 181)
Shape of T2 training images:  (36, 217, 181)
Shape of T1 testing images:  (9, 217, 181)
Shape of T2 testing images:  (9, 217, 181)
```

Figure 6: Data Loading

4 Data Preprocessing

In this step, data normalization is shown in the figure.7, image resizing, image reshaping, batching and shuffling of the image are done.

3. Data Preprocessing

3a. Image Normalization
3b. Image Resizing
3c. Image Reshaping
3d. Shuffling and Batching images

3a. Image Normalization

```
[ ] # Normalizing images between -1 and +1

    # Normalizing T1 images
    T1_trimages_norm = (T1_trimages/127.5) - 1.0
    T1_teimages_norm = (T1_teimages/127.5) - 1.0
    # Normalizing T2 images
    T2_trimages_norm = (T2_trimages/127.5) - 1.0
    T2_teimages_norm = (T2_teimages/127.5) - 1.0
```

```
[ ] # Visualizing T1 and T2 weighted images - after normalization
    fig, ax = plt.subplots(1, 2, figsize = [8, 8])

    fig.subplots_adjust(left = 0.1,
                        bottom = 0.1,
                        right = 0.9,
                        top = 0.9,
                        wspace = 0.4,
                        hspace = 0.4)

    #T1-weighted image - after normalization
    ax[0].set_title('T1-weighted MRI scan')
    ax[0].imshow(T1_trimages[0], cmap = 'gray')

    #T2-weighted image - after normalization
    ax[1].set_title('T2-weighted MRI scan')
    ax[1].imshow(T2_trimages[0], cmap = 'gray')

    plt.show()
```

Figure 7: Data Normalization

5 Model Building

After the data pre-processing is done the model for generating MRI images is built using U-Net segmentation by using Upsampling (Transposed Convolutional Neural Network)and Downsampling(Convolutional Neural Network) layers as shown in figure.8

After this step building of the Discriminator is also performed with only Downsampling (Convolutional Neural Network) layers as shown in figure.9.

- Defining Losses: After building a model all Losses are defined as required for training a model as shown in figure.10.

- Checkpoints Initialization: To save the model during training flow checkpoints are stored on the drive and Adam optimizer is also applied to the model as shown in figure.11.

```
# Defining a function for Unet generator
def unet_generator():
  #Downsampling
  down_stack = [
              downsample(64, 4, False), #I/P: (bs, 256, 256, 1), bs: batch_size; O/P: (bs, 128, 128, 64)
              downsample(128, 4), #O/P: (bs, 64, 64, 128)
              downsample(256, 4), #O/P: (bs, 32, 32, 256)
              downsample(256, 4), #O/P: (bs, 16, 16, 256)
              downsample(256, 4), #O/P: (bs, 8, 8, 256)
              downsample(512, 4), #O/P: (bs, 4, 4, 512)
              downsample(512, 4), #O/P: (bs, 2, 2, 512)
              downsample(512, 4) #independent bottleneck layer at the middle; O/P: (bs, 1, 1, 512)
               ]#For every downsampling layer, there will be an upsampling counterpart
  #Upsampling
  up_stack = [
             upsample(512, 4, True), #I/P: (bs, 1, 1, 512), bs: batch_size; O/P: (bs, 2, 2, 512)
             upsample(512, 4, True), #O/P: (bs, 4, 4, 512)
             upsample(256, 4), #O/P: (bs, 8, 8, 256)
             upsample(256, 4), #O/P: (bs, 16, 16, 256)
             upsample(256, 4), #O/P: (bs, 32, 32, 256)
             upsample(128, 4), #O/P: (bs, 64, 64, 128)
             upsample(64, 4) #(bs, 128, 128, 64)
  ]
  initializer = tf.random_normal_initializer(0., 0.02)
  #Layer at the last of upsampling
  last = tf.keras.layers.Conv2DTranspose(1, #1 channel output (grayscale)
                                         4, strides = 2, #strides indicate number of pixesls shift over input matrix
                                         padding = 'same',
                                         kernel_initializer = initializer,
                                         activation = 'tanh') #I/P: (bs, 128, 128, 64), bs: batch_size; O/P: (bs, 256, 256, 1)
  #Downsample and upsample layers must be connected together by bypass skip connections
  concat = tf.keras.layers.Concatenate()
  inputs = tf.keras.layers.Input(shape = [resize_dim_1, resize_dim_2, 1]) #input 1 grayscale image of depth 1
  x = inputs
  #Downsampling through the model
  skips = []
  for down in down_stack:
    x = down(x)
    skips.append(x)
  skips = reversed(skips[:-1]) #skips[:-1] leaves out the bottleneck layer from skips listf
  #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)
```

Figure 8: U-Net based generator model

4e. Discriminator Building

```
[ ] # Discriminators are image classifiers
    # Discriminators contain only Convolutional Layers and no Transposed Convolutional layers
    # Discriminator has only downsampling layers
```

```
# Defining a function for Discriminator Model

def discriminator():
  initializer = tf.random_normal_initializer(0., 0.02)

  #Adding input layer of size (resize_dim_1, resize_dim_2,1)
  inp = tf.keras.layers.Input(shape = [resize_dim_1, resize_dim_2, 1],
                              name = 'input_image')
  x = inp

  #Adding downsampling step
  down1 = downsample(64, 4, False)(x) #I/P: (bs, 256, 256, 1), bs: batch_size; O/P: (bs, 128, 128, 64)
  down2 = downsample(128, 4)(down1) #O/P: (bs, 64, 64, 128)
  down3 = downsample(256, 4)(down2) #O/P: (bs, 32, 32, 256)
  #Adding a padding (zero) layer
  zero_pad1 = tf.keras.layers.ZeroPadding2D()(down3) #O/P: (bs, 34, 34, 256)

  #Implementing a concrete downsampling layer
  conv = tf.keras.layers.Conv2D(512, 4,
                                strides = 1,
                                kernel_initializer = initializer,
                                use_bias = False)(zero_pad1) #I/P: (bs, 34, 34, 256), bs: batch_size; O/P: (bs, 31, 31, 512)
  norm1 = InstanceNormalization()(conv) #Applying instance normalization
  leaky_relu = tf.keras.layers.LeakyReLU()(norm1) #Applying activation

  #Applying zero padding layer
  zero_pad2 = tf.keras.layers.ZeroPadding2D()(leaky_relu) #O/P: (bs, 33, 33, 512)

  #Add a last pure 2D convolution layer
  last = tf.keras.layers.Conv2D(1, 4, strides = 1, kernel_initializer = initializer)(zero_pad2) #O/P: (bs, 30, 30, 1)

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

Figure 9: U-Net based Discriminator model

5b. Calculating Discriminator Loss

Discriminator losses:

- Loss on Real data
- Loss on Fake/Generated data

```
[ ] # Defining a function for discriminator loss

    def discriminator_loss(real_data, fake_data):
      real_loss = loss_func(tf.ones_like(real_data), real_data) #Discriminator loss on real data (misclassification of real as fake)
      fake_loss = loss_func(tf.zeros_like(fake_data), fake_data) #Discriminator loss on fake data (misclassification of fake as real)
      total_disc_loss = real_loss + fake_loss #Total discriminator loss
      return total_disc_loss * 0.5 #weighted loss
```

5c. Calculating Generator Loss

- Loss on generated/fake data

```
[ ] # Defining a function for generator loss

    def generator_loss(fake_data):
       gg=loss_func(tf.ones_like(fake_data), fake_data) #no weights associated with loss
       return gg
```

5d. Calculating Cycle Consistency Loss

- Cycle Loss

```
[ ] # Cycle loss: L1 distance between the real and the recycled image
```

```
[ ] # Defining a function for cycle loss

    def cycle_loss(real_image, cycled_image):
      #Pixel-wise difference between the real and cycled image is taken first
      #Then the absolute value of such difference is taken : L1 loss
      #tf.reduce_mean (computes mean of elements across dimensions of a tensor) takes the average of such losses
      cyc_loss = tf.reduce_mean(tf.abs(real_image - cycled_image))
      return 10.0 * cyc_loss #weighted loss
```

Figure 10: Losses

```
[ ] # Initializing Adam optimizer

    # For Generators
    generator_g_optimizer = tf.keras.optimizers.Adam(2e-4, #learning rate (2e-4 = 0.0002)
                                                     beta_1 = 0.5 #exponential decay rate for 1st moment estimates
                                                     )

    generator_f_optimizer = tf.keras.optimizers.Adam(2e-4,
                                                     beta_1 = 0.5)

    # For Discriminators
    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)
```

5g. Checkpoint Initialization

- Checkpoints initialized to save models during training flow

```
[ ] # Defining the Checkpoint path
    checkpoint_path = './Trained_Model'

    # Defining the Checkpoint
    ckpt = tf.train.Checkpoint(generator_g = generator_g,
                               generator_f = generator_f,
                               discriminator_x = discriminator_x,
                               discriminator_y = discriminator_y,
                               generator_g_optimizer = generator_g_optimizer,
                               generator_f_optimizer = generator_f_optimizer,
                               discriminator_x_optimizer = discriminator_x_optimizer,
                               discriminator_y_optimizer = discriminator_y_optimizer)

    # Setting the Checkpoint Manager
    ckpt_manager = tf.train.CheckpointManager(ckpt,
                                              checkpoint_path,
                                              max_to_keep = 5)

    # To restore the latest checkpoint if a checkpoint exists
    if ckpt_manager.latest_checkpoint:
      ckpt.restore(ckpt_manager.latest_checkpoint)
      print('Latest checkpoint restored')
```

Figure 11: Checkpoints Initialization and applying Optimizer

6 Model Training

The function is defined for the training of a single batch of data using the CycleGAN framework which contains 2 generators and 2 discriminators as shown in figure.12.

```
# Defining a function for training a single batch of data
@tf.function
def train_step(real_x, real_y):
  with tf.GradientTape(persistent = True) as tape:
    #Persistent = True because tape is used more than once to calculate the gradients
    #Generator G translates from X to Y
    #Generator F translates from Y to X
    fake_y = generator_g(real_x, training = True)
    cycled_x = generator_f(fake_y, training = True)

    fake_x = generator_f(real_y, training = True)
    cycled_y = generator_g(fake_x, training = True)
    #Generating same_x and same_y to calculate identity loss
    same_x = generator_f(real_x, training = True)
    same_y = generator_g(real_y, training = True)
    #Finding discriminator predictions on real and fake data
    disc_real_x = discriminator_x(real_x, training = True)
    disc_real_y = discriminator_y(real_y, training = True)
    disc_fake_x = discriminator_x(fake_x, training = True)
    disc_fake_y = discriminator_y(fake_y, training = True)
    #Calculating losses
    #Calculating loss on generators
    gen_g_loss = generator_loss(disc_fake_y)
    gen_f_loss = generator_loss(disc_fake_x)
    #Calculating total cycle loss
    total_cycle_loss = cycle_loss(real_x, cycled_x) + cycle_loss(real_y, cycled_y)
    #Calculating the total generator loss
    #Total generator loss = BCE loss + cycle loss + identity loss
    total_gen_g_loss = gen_g_loss + total_cycle_loss + identity_loss(real_y, same_y)
    total_gen_f_loss = gen_f_loss + total_cycle_loss + identity_loss(real_x, same_x)
    #Discriminator losses
    disc_x_loss = discriminator_loss(disc_real_x, disc_fake_x)
    disc_y_loss = discriminator_loss(disc_real_y, disc_fake_y)
  #Calculating gradients for generator and discriminator
  generator_g_gradients = tape.gradient(total_gen_g_loss,
                                        generator_g.trainable_variables)
  generator_f_gradients = tape.gradient(total_gen_f_loss,
                                        generator_f.trainable_variables)

  discriminator_x_gradients = tape.gradient(disc_x_loss,
                                            discriminator_x.trainable_variables)
  discriminator_y_gradients = tape.gradient(disc_y_loss,
                                            discriminator_y.trainable_variables)
  #Applying gradients to optimizer
  generator_g_optimizer.apply_gradients(zip(generator_g_gradients,
                                            generator_g.trainable_variables))
  generator_f_optimizer.apply_gradients(zip(generator_f_gradients,
                                            generator_f.trainable_variables))

  discriminator_x_optimizer.apply_gradients(zip(discriminator_x_gradients,
                                                discriminator_x.trainable_variables))
  discriminator_y_optimizer.apply_gradients(zip(discriminator_y_gradients,
                                                discriminator_y.trainable_variables))
```

Figure 12: Function defination of model training

```
[ ] # Setting number of epochs for model training
    num_epochs = 100
```

6. Visualizing Style Transfer

```
[ ] from keras.utils.image_utils import save_img
    # Calling the train_step function for model training over a specified number of epochs
    import matplotlib.pyplot as plt
    for epoch in range(1, num_epochs + 1):
      for image_x, image_y in tf.data.Dataset.zip((T1_train, T2_train)):
       train_step(image_x, image_y)
      generate_images(generator_g, sample_T1_train,
                      generator_f, sample_T2_train)


      ckpt_save_path = ckpt_manager.save()
      print('Saving checkpoint for epoch', epoch,'at', ckpt_save_path)


      c=cycle_loss(image_x, image_y)
      i=identity_loss(image_x, image_y)
      d=discriminator_loss(image_x, image_y)
      g=generator_loss(image_y)

      print('losses','Cyclic consistancy loss: {:1.2f}'.format(c.numpy()))
      print('identity_loss: {:1.2f}'.format(i.numpy()))
      print('discriminator_loss: {:1.2f}'.format(d.numpy()))
      print('generator_loss: {:1.2f}'.format(g.numpy()))
```

Input Image-T1 Generated Image-T2 Input Image-T2 Generated Image-T1

```
Saving checkpoint for epoch 96 at ./Trained_Model/ckpt-96
losses Cyclic consistancy loss: 3.90
identity_loss: 0.20
discriminator_loss: 0.74
generator_loss: 0.92
```

Figure 13: Model Training

7 Image Generation and Model Evaluation:

The function is created that takes the input image and generates the output MRI respectively as shown in the figure.14, Also in this function definition evaluation code is included which prints the SSIM and PSNR score. Whenever this function is called it generates the input image and also prints the SSIM and PSNR score to evaluate a respective model.

After the function is defined it is called to generate 4 types of images whose output contains generated image and the SSIM and PSNR score of the generated image respectively.

```python
# Creating a function to take input image and provide generated image
from skimage.metrics import structural_similarity as ssim
from scipy import mean
from collections import OrderedDict
from math import log10, sqrt
import math
def image_generator(input_image, generator):
  ##Pre-processing input image
  #Normalizing input between -1 and +1
  preprocess_inp = (input_image / 127.5) - 1
  #Resizing input image
  resize_dim_1 = 256
  resize_dim_2 = 256
  preprocess_inp = resize(preprocess_inp, (resize_dim_1, resize_dim_2))
  #Reshaping input image
  preprocess_inp = preprocess_inp.reshape(1, resize_dim_1, resize_dim_2, 1)
  #Converting datatype to float32
  preprocess_inp = preprocess_inp.astype('float32')
  ##Generating image out of preprocessed input using generator
  generated_image = generator(preprocess_inp)
  ##Post-processing output image
  #Converting output image from TF format to numpy array format
  postprocess_out = generated_image.numpy()
  #Reshaping output image
  postprocess_out = postprocess_out.reshape(resize_dim_1, resize_dim_2)
  #Resizing output image to dimensions of input image
  postprocess_out = resize(postprocess_out, (input_image.shape[0],nput_image.shape[1]))      i
  #De-normalizing output image
  postprocess_out = (postprocess_out + 1) * 127.5
  #Converting datatype of output image to that of input image
  postprocess_out = postprocess_out.astype(str(input_image.dtype))
  ##Visualizing results
  plt.figure(figsize = [16, 16])
  plt.subplot(1, 2, 1)
  plt.title('Original image')
  plt.imshow(input_image, cmap = 'gray')
  plt.subplot(1, 2, 2)
  plt.title('Generated image')
  plt.imshow(postprocess_out, cmap = 'gray')
  plt.savefig('generated image_{:04d}.png'.format(epoch))
  s = ssim(input_image, postprocess_out)
  plt.show()
  print('SSIM SCORE',s)
  mse = mean((input_image - postprocess_out) ** 2)
  max_pixel = 255.0
  psnr = 20 * log10(max_pixel / sqrt(mse))
  print('PSNR SCORE',psnr,'db')
```

Figure 14: Function for Image Generation and Model Evaluation

7.1 Generating Data By Providing Test dataset

The four results are generated such as T1styled image to T1Styled image as shown in figure.15,T2-Styled image to T2-styled image as shown in figure.16, T2-styled image to T1-Styled image as shown in figure.17, T1-styled image to T2-Styled image as shown in figure.18.

These four figure shows the results of generated images using 2 types of a generator on the test dataset which is converted into TensorFlow format.

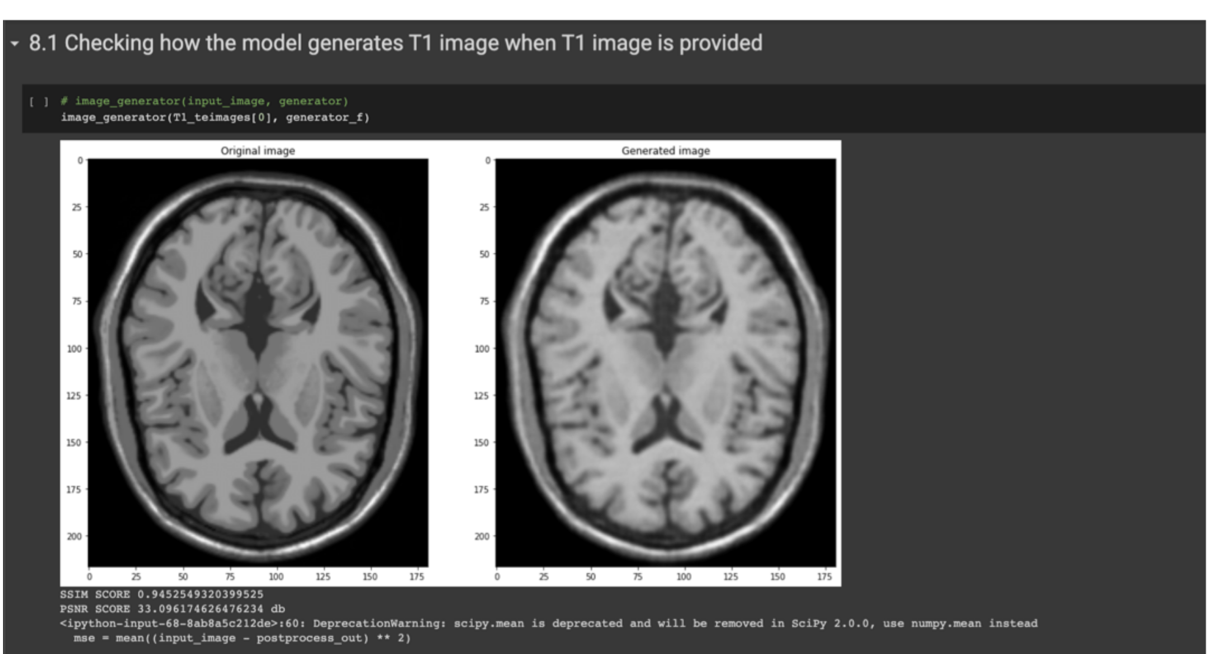

Figure 15: T1-styled image to T1-Styled image

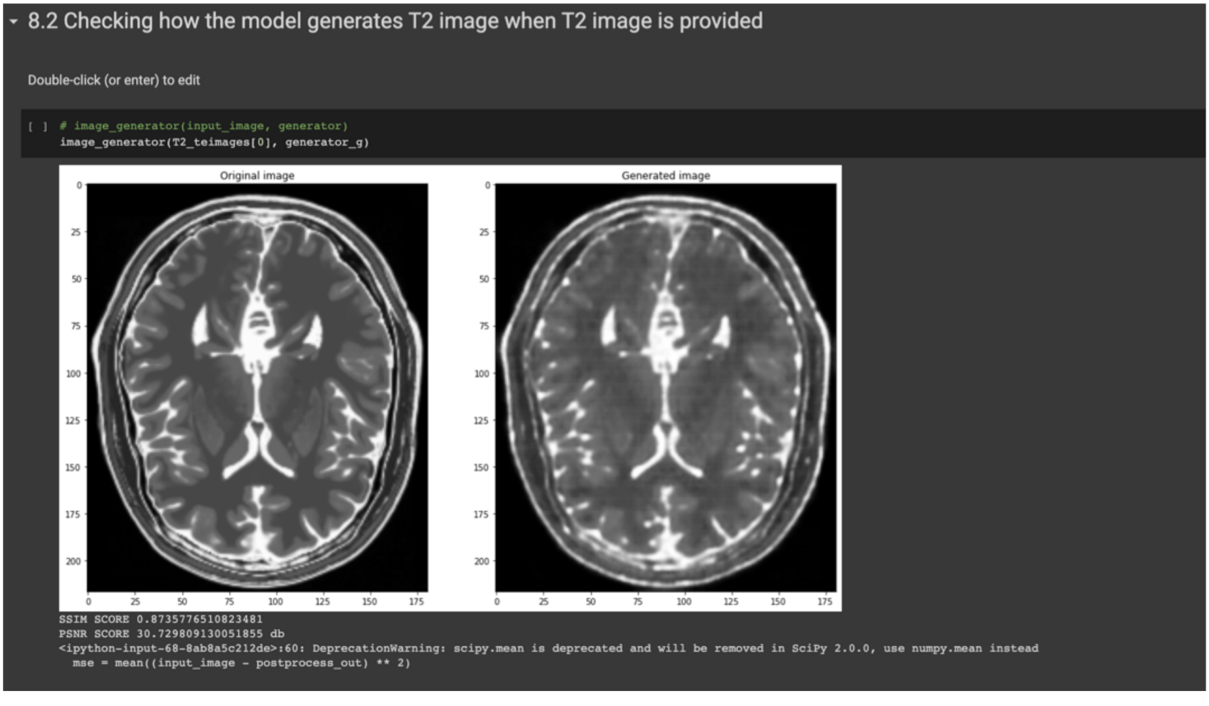

Figure 16: T2-Styled image to T2-styled image

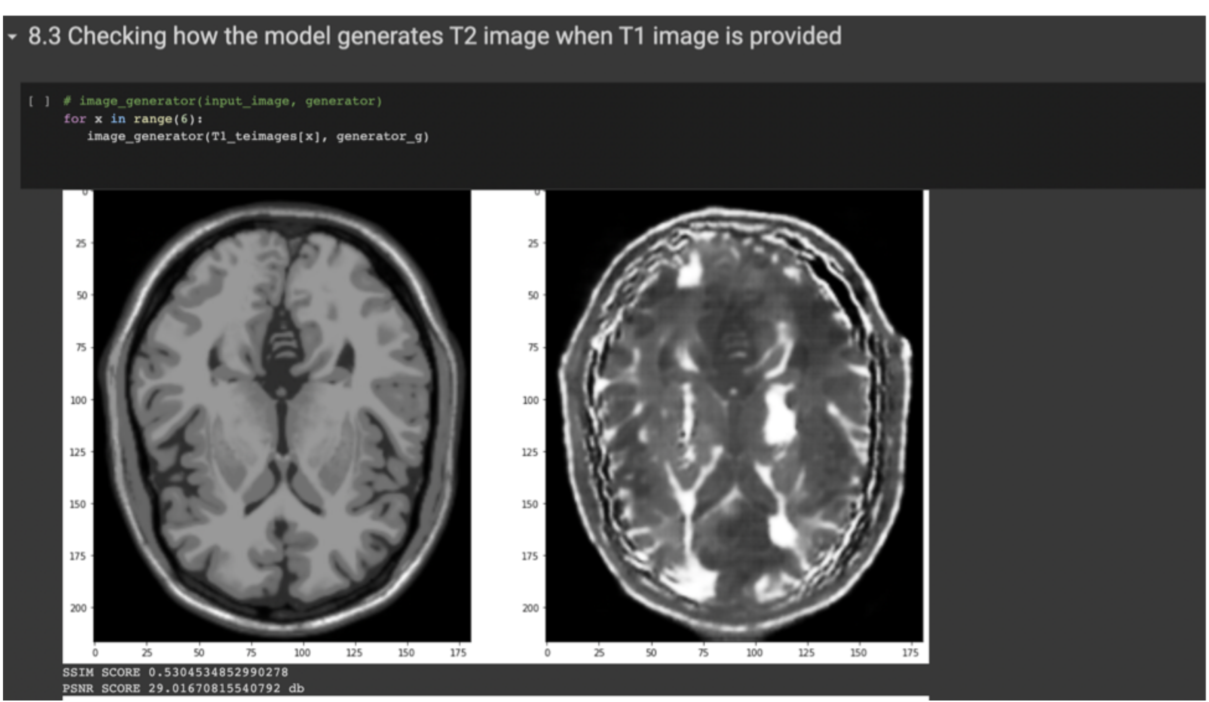

Figure 17: T2-styled image to T1-Styled image

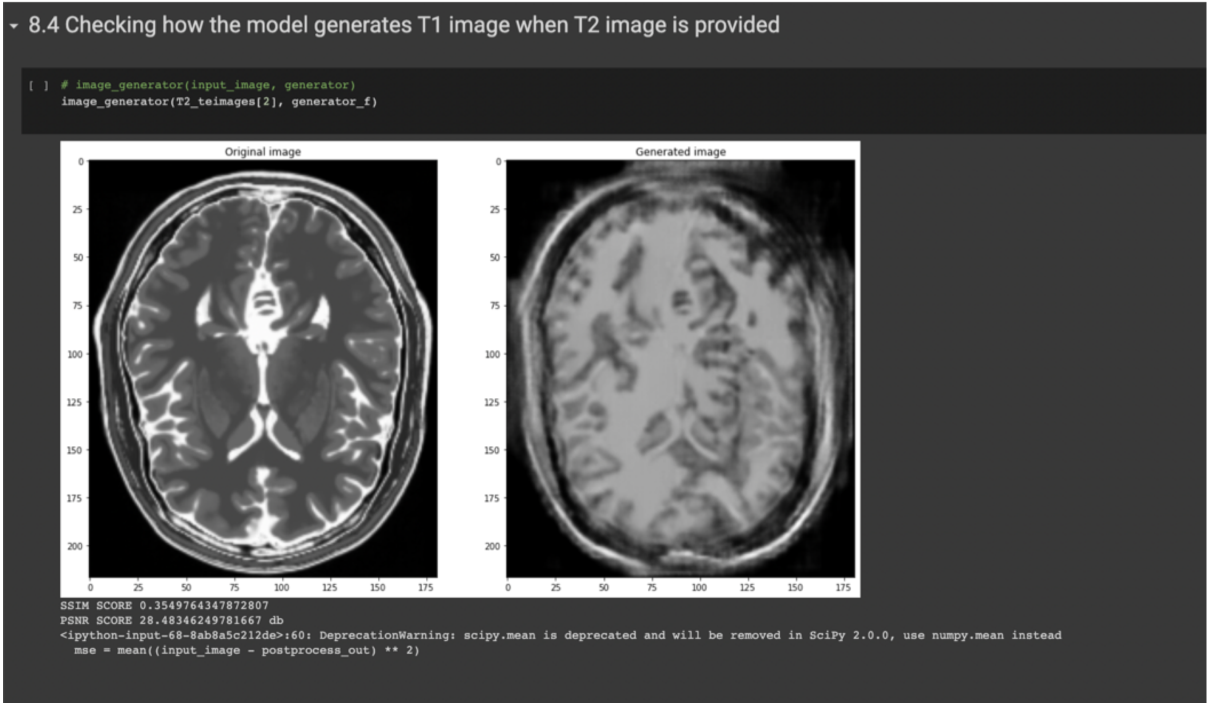

Figure 18: T1-styled image to T2-Styled image