Fatima Jinnah Women University

Sketch to Digital Image Conversion Using Conditional GANs (pix2pix)

Project Documentation

Submitted to:

Ma'am Mehreen Sirshar

Submitted By:

Hafsa Sheikh	2017-BSE-043
Saman Rani	2017-BSE-061

6th Semester

Department of Software Engineering

1. Introduction:

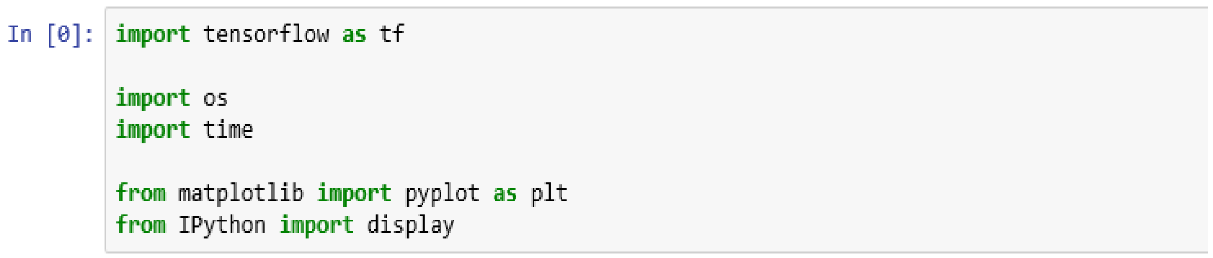
Our project is Sketch to Digital image conversion using pix2pix implemented in Google Colaboratory platform using python language version 3.0. Main aim of the project is to convert sketches into colored image. Using pix2pix we can easily achieve this goal. In the project we use edges2shoes dataset that contains sketch and its corresponding digital image. A generator network use existing data to generate new data. The discriminator network tries to differentiate between the real data and the data generated by the generator network.

2. Process

The first network, the generator, has never seen the real artwork but is trying to create an artwork that look like a real thing. The second network, the discriminator, tries to identify whether an artwork is real or fake. The generator, in turn, tries to fool the discriminator into thinking that it is fake are the real deal by creating more realistic artwork over multiple iterations. The discriminator tries to outwit the generator by continuing to refine its own criteria for determining a fake. They guide each by providing feedback from the successful changes they make in their own process in each iteration. This process is the training of GAN. Ultimately, the discriminator trains the generator to the point at which it can no longer determine which artwork is real and which is fake.

3. Importing libraries:

Import statement is used to **import** a library into the code so that **we** can use it's functionalities in the code. In this code we import different libraries to use the functionality of those libraries in the code



```
In [0]: import tensorflow as tf

        import os
        import time

        from matplotlib import pyplot as plt
        from IPython import display
```

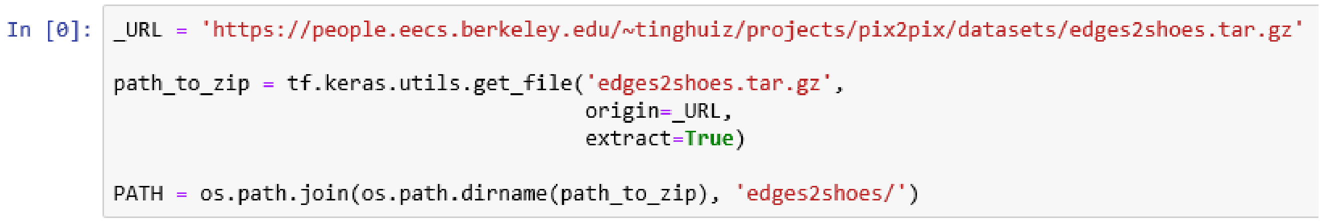
```
In [0]: !pip install -U tensorboard
```

4. Loading Dataset:

We can load the dataset from index of /~tinghuiz/projects/pix2pix/datasets and we apply random jittering and mirroring techniques to the traing dataset.

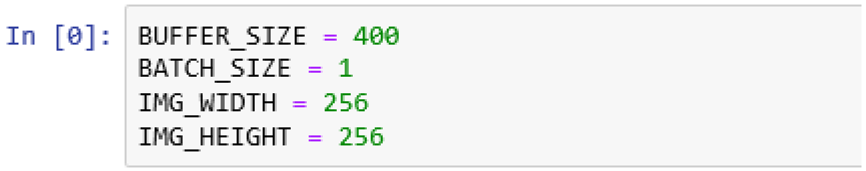
- In random jittering, the image is resized to 286 x 286 and then randomly cropped to 256 x 256
- In random mirroring, the image is randomly flipped horizontally i.e left to right.

In the piece of code we are are loading the dataset from the above mention link.



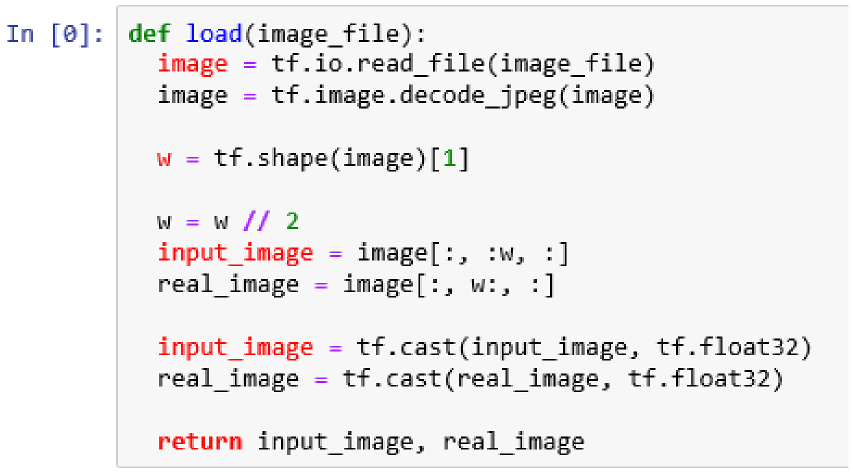
```
In [0]: _URL = 'https://people.eecs.berkeley.edu/~tinghuiz/projects/pix2pix/datasets/edges2shoes.tar.gz'

        path_to_zip = tf.keras.utils.get_file('edges2shoes.tar.gz',
                                              origin=_URL,
                                              extract=True)

        PATH = os.path.join(os.path.dirname(path_to_zip), 'edges2shoes/')
```

In this piece of code defining the different attributes of an image

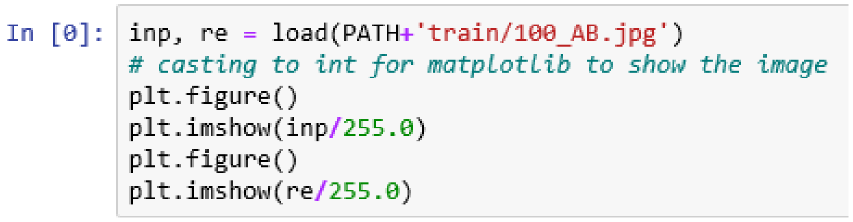


```
In [0]: BUFFER_SIZE = 400
        BATCH_SIZE = 1
        IMG_WIDTH = 256
        IMG_HEIGHT = 256
```

In this code **load function** is used to load an image from the dataset using tensorflow library

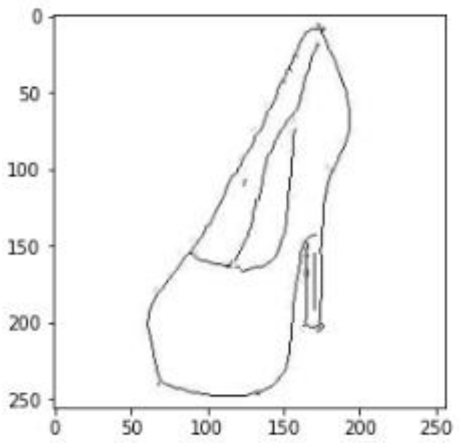


```
In [0]: def load(image_file):
          image = tf.io.read_file(image_file)
          image = tf.image.decode_jpeg(image)

          w = tf.shape(image)[1]

          w = w // 2
          input_image = image[:, :w, :]
          real_image = image[:, w:, :]

          input_image = tf.cast(input_image, tf.float32)
          real_image = tf.cast(real_image, tf.float32)

          return input_image, real_image
```

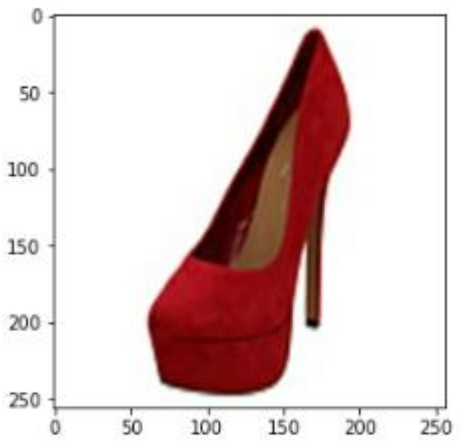


```
In [0]: inp, re = load(PATH+'train/100_AB.jpg')
        # casting to int for matplotlib to show the image
        plt.figure()
        plt.imshow(inp/255.0)
        plt.figure()
        plt.imshow(re/255.0)
```

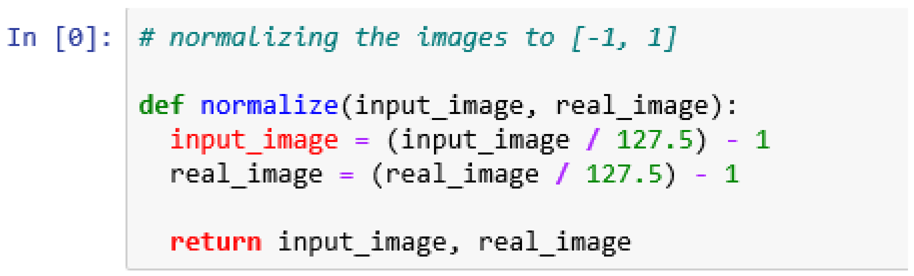
Output is:



0
50
100
150
200
250
0
50
100
150
200
250

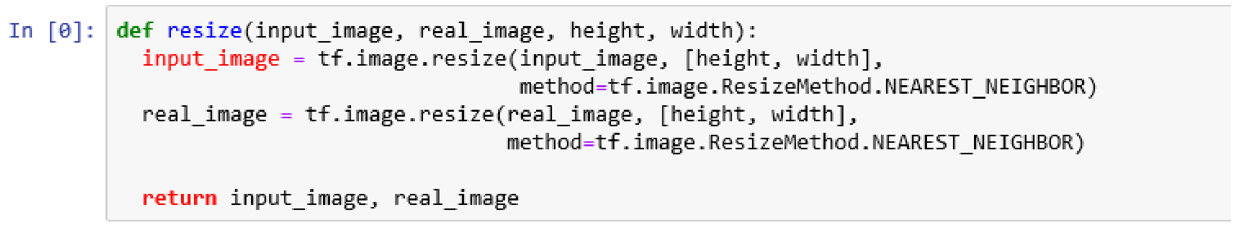


0
50
100
150
200
250
0
50
100
150
200
250

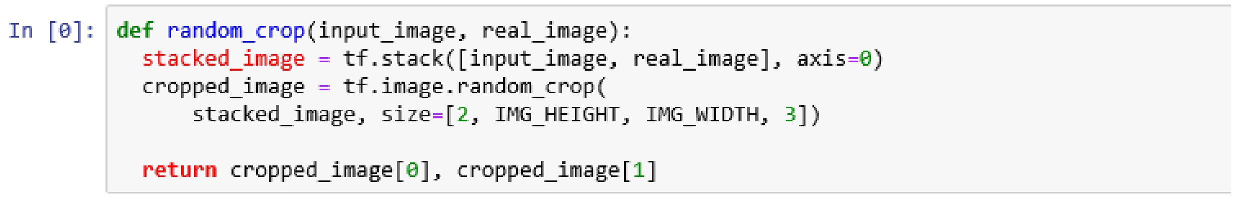


```
In [0]: # normalizing the images to [-1, 1]

        def normalize(input_image, real_image):
          input_image = (input_image / 127.5) - 1
          real_image = (real_image / 127.5) - 1

          return input_image, real_image
```

This code defines the **resize and random_crop functions()** to resizing the image by changing the values of image attributes



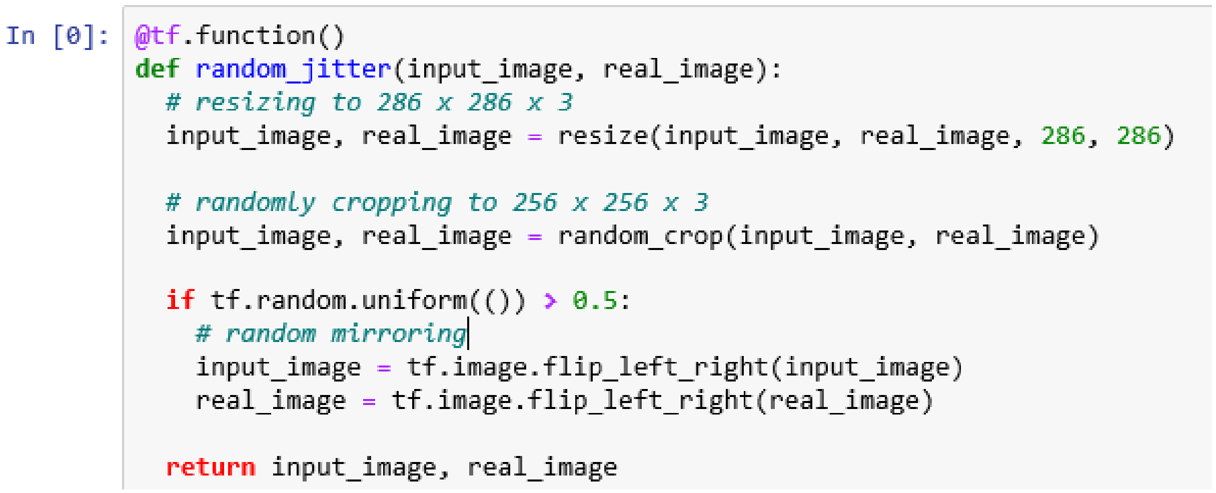
```
In [0]: def resize(input_image, real_image, height, width):
          input_image = tf.image.resize(input_image, [height, width],
                                        method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)
          real_image = tf.image.resize(real_image, [height, width],
                                       method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)

          return input_image, real_image
```



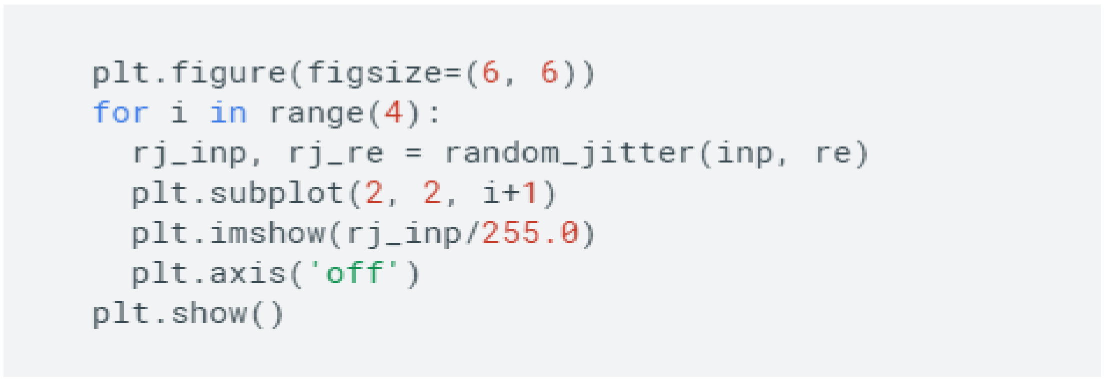
```
In [0]: def random_crop(input_image, real_image):
          stacked_image = tf.stack([input_image, real_image], axis=0)
          cropped_image = tf.image.random_crop(
              stacked_image, size=[2, IMG_HEIGHT, IMG_WIDTH, 3])

          return cropped_image[0], cropped_image[1]
```

Here define **normalize function()** to mormalize an image by changing the range of pixel intensity value.

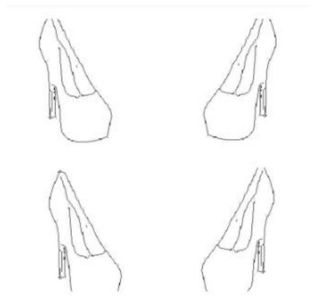
@tf.function() is used to construct a callable that execute a tensorflow graph created by trace-compiling the tensorflow operations in functions and **random_jittering** function is used randomly cropping for the image by Resize an image to bigger height and width. Randomly crop to the target size. Randomly flip the image horizontally. using random jittering the image can be seen as:



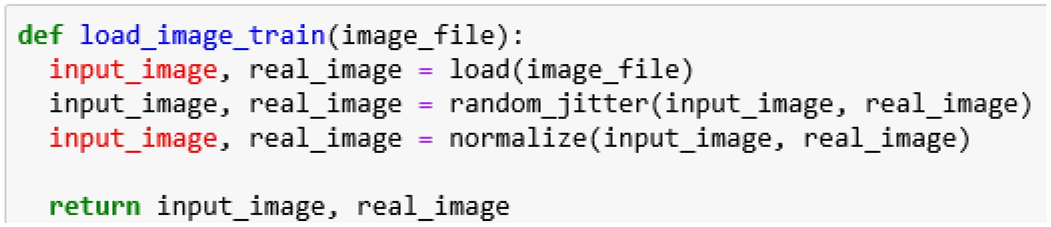
```
In [0]: @tf.function()
        def random_jitter(input_image, real_image):
          # resizing to 286 x 286 x 3
          input_image, real_image = resize(input_image, real_image, 286, 286)

          # randomly cropping to 256 x 256 x 3
          input_image, real_image = random_crop(input_image, real_image)

          if tf.random.uniform(()) > 0.5:
            # random mirroring
            input_image = tf.image.flip_left_right(input_image)
            real_image = tf.image.flip_left_right(real_image)

          return input_image, real_image
```



```
plt.figure(figsize=(6, 6))
for i in range(4):
  rj_inp, rj_re = random_jitter(inp, re)
  plt.subplot(2, 2, i+1)
  plt.imshow(rj_inp/255.0)
  plt.axis('off')
plt.show()
```

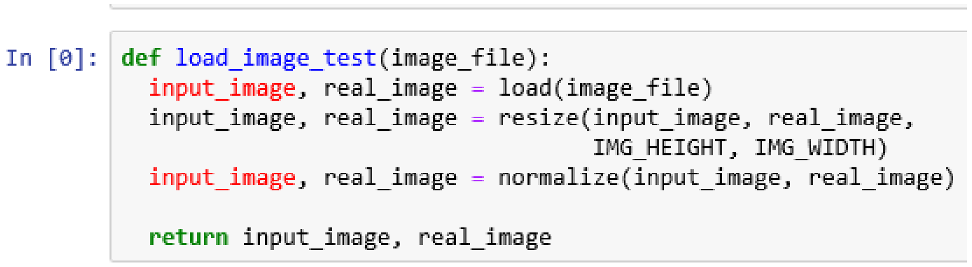


load_image_train() function is used to load image for the training dataset.



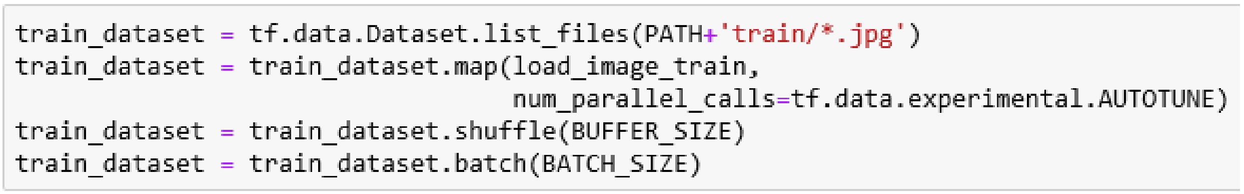
```
def load_image_train(image_file):
  input_image, real_image = load(image_file)
  input_image, real_image = random_jitter(input_image, real_image)
  input_image, real_image = normalize(input_image, real_image)

  return input_image, real_image
```

load_image_test () function is used to load image for the testing dataset.

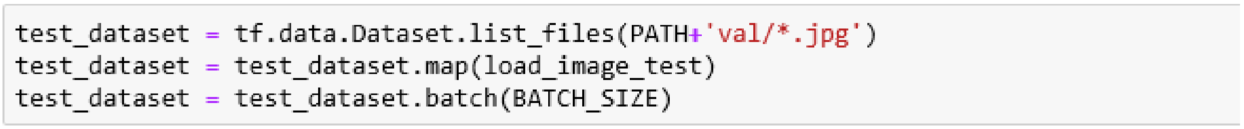


```
In [0]: def load_image_test(image_file):
          input_image, real_image = load(image_file)
          input_image, real_image = resize(input_image, real_image,
                                           IMG_HEIGHT, IMG_WIDTH)
          input_image, real_image = normalize(input_image, real_image)

          return input_image, real_image
```

5. **Input pipeline** is used to chain multiple estimators into one and hence ,automate the machine learning process. This is extremely useful as there are often a fixed sequence of steps in processing the data.
6. **Build a generator:**



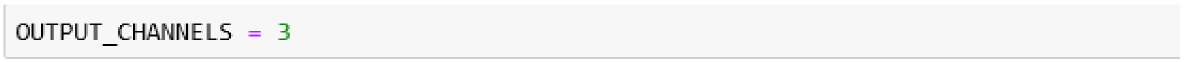
```
train_dataset = tf.data.Dataset.list_files(PATH+'train/*.jpg')
train_dataset = train_dataset.map(load_image_train,
                                  num_parallel_calls=tf.data.experimental.AUTOTUNE)
train_dataset = train_dataset.shuffle(BUFFER_SIZE)
train_dataset = train_dataset.batch(BATCH_SIZE)
```



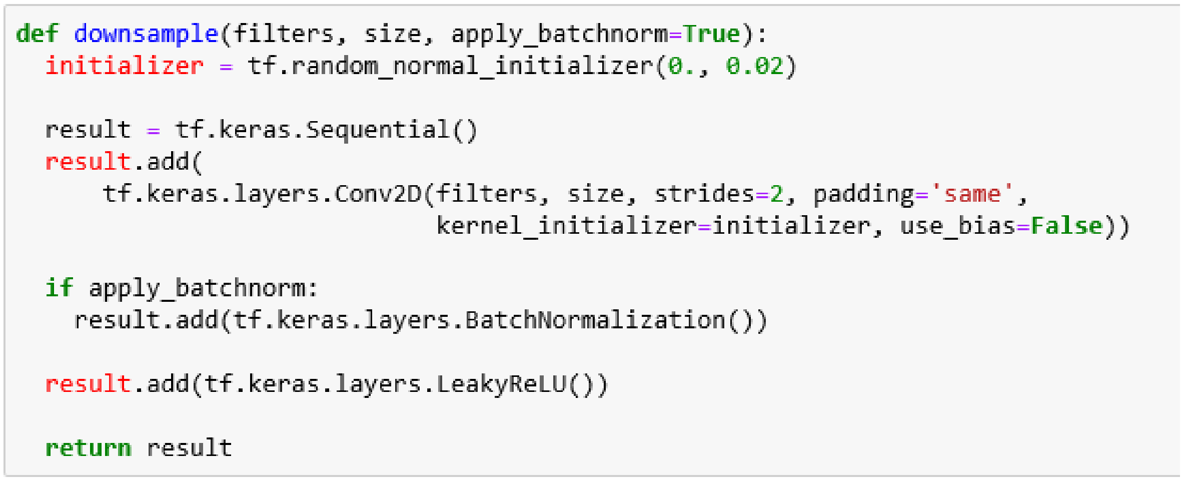
```
test_dataset = tf.data.Dataset.list_files(PATH+'val/*.jpg')
test_dataset = test_dataset.map(load_image_test)
test_dataset = test_dataset.batch(BATCH_SIZE)
```

GANs has two neural network one is a **generator** that synthesizes new samples from scratch.The generator is not trained directly and instead is trained via a discriminator network. Under build a generator following functions are used

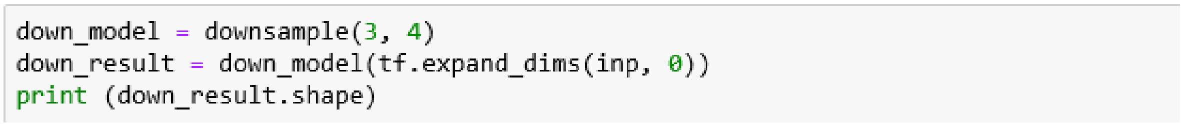
- **Downsample()** will randomly sample a data set so that all classes have the same frequency as the minority class.



```
OUTPUT_CHANNELS = 3
```

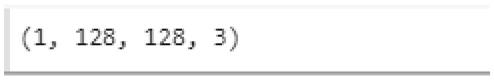


```
def downsample(filters, size, apply_batchnorm=True):
  initializer = tf.random_normal_initializer(0., 0.02)

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

  if apply_batchnorm:
    result.add(tf.keras.layers.BatchNormalization())

  result.add(tf.keras.layers.LeakyReLU())

  return result
```



```
down_model = downsample(3, 4)
down_result = down_model(tf.expand_dims(inp, 0))
print (down_result.shape)
```

Output is :



```
(1, 128, 128, 3)
```

- **Upsample()** samples with replacement to make the class distributions equal.

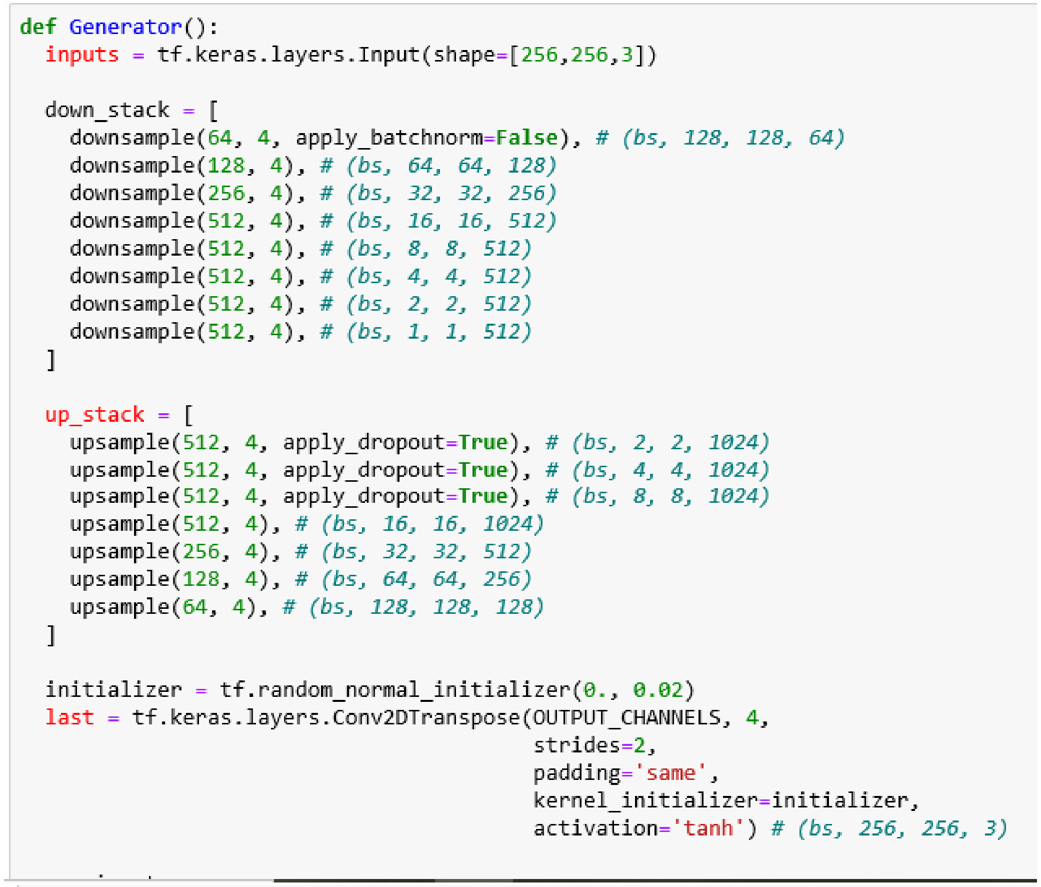
```
def upsample(filters, size, apply_dropout=False):
  initializer = tf.random_normal_initializer(0., 0.02)

  result = tf.keras.Sequential()
  result.add(
    tf.keras.layers.Conv2DTranspose(filters, size, strides=2,
                                    padding='same',
                                    kernel_initializer=initializer,
                                    use_bias=False))

  result.add(tf.keras.layers.BatchNormalization())

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

  result.add(tf.keras.layers.ReLU())

  return result
```

```
up_model = upsample(3, 4)
up_result = up_model(down_result)
print (up_result.shape)
```

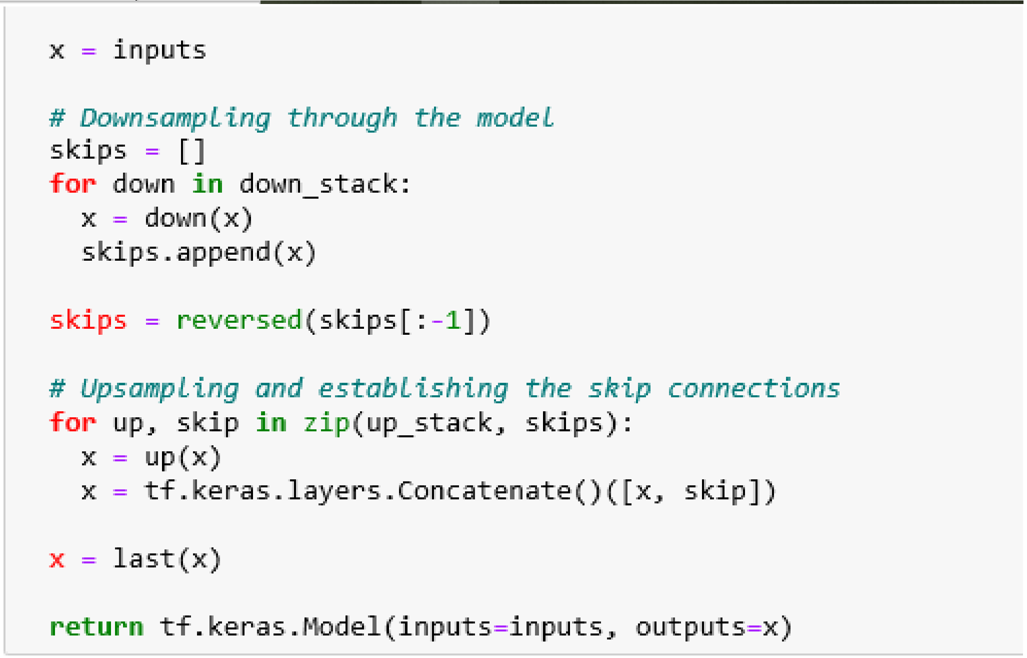
Output is:

```
(1, 256, 256, 3)
```

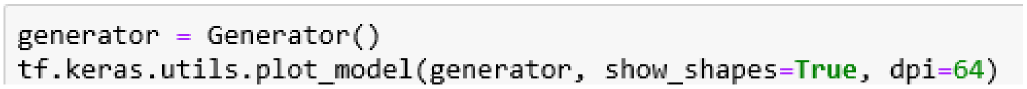
- **Generator()** is a special type of function which does not return a single value, instead it returns an iterator object with a sequence of values. It can be used in a for loop.



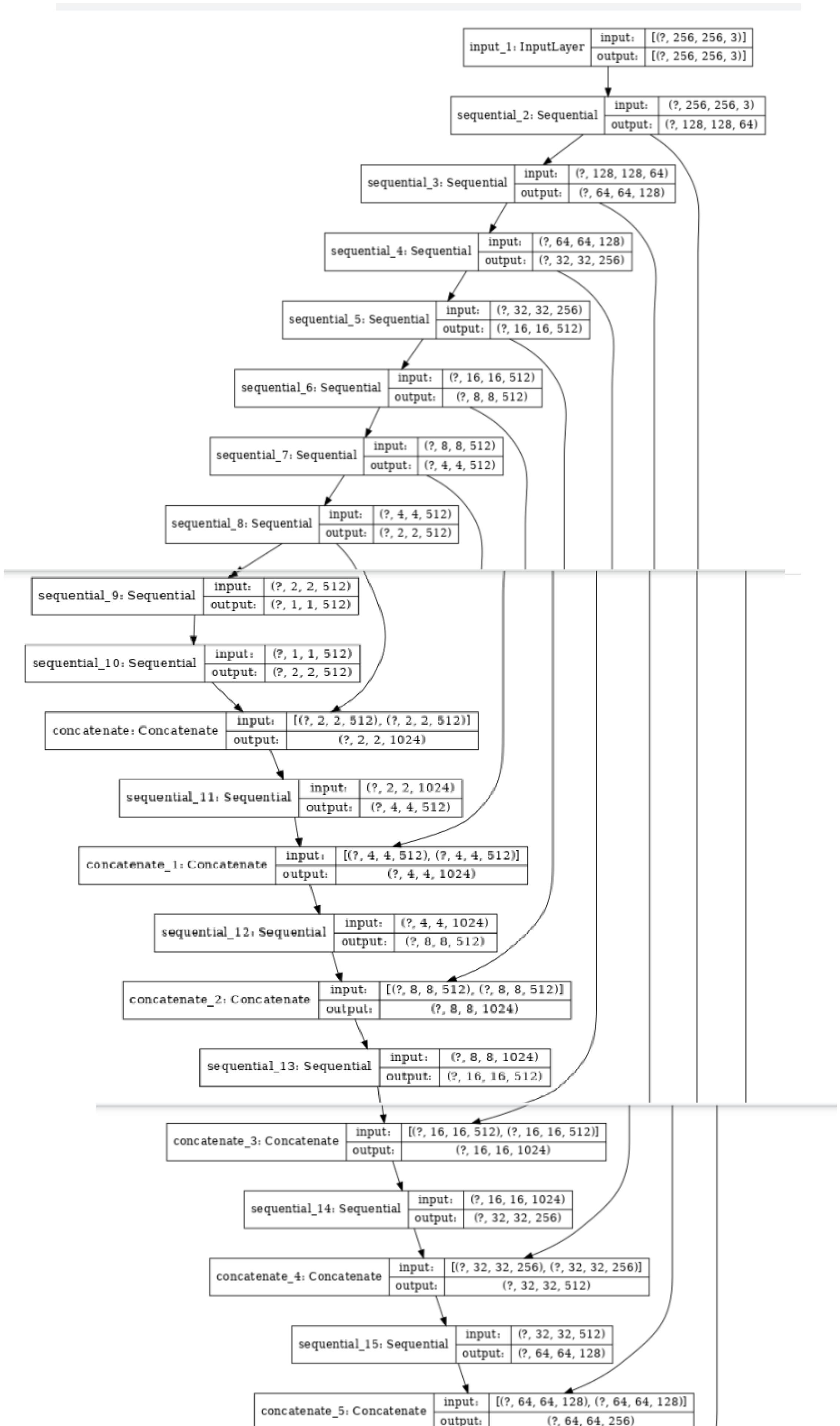
```
def Generator():
  inputs = tf.keras.layers.Input(shape=[256,256,3])

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

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

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



```
  x = inputs

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

  skips = reversed(skips[:-1])

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

  x = last(x)

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

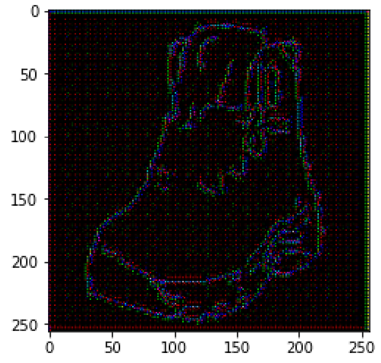


```
generator = Generator()
tf.keras.utils.plot_model(generator, show_shapes=True, dpi=64)
```


input_1: InputLayer | input: | [(?, 256, 256, 3)] | output: | [(?, 256, 256, 3)]
sequential_2: Sequential | input: | (?, 256, 256, 3) | output: | (?, 128, 128, 64)
sequential_3: Sequential | input: | (?, 128, 128, 64) | output: | (?, 64, 64, 128)
sequential_4: Sequential | input: | (?, 64, 64, 128) | output: | (?, 32, 32, 256)
sequential_5: Sequential | input: | (?, 32, 32, 256) | output: | (?, 16, 16, 512)
sequential_6: Sequential | input: | (?, 16, 16, 512) | output: | (?, 8, 8, 512)
sequential_7: Sequential | input: | (?, 8, 8, 512) | output: | (?, 4, 4, 512)
sequential_8: Sequential | input: | (?, 4, 4, 512) | output: | (?, 2, 2, 512)
sequential_9: Sequential | input: | (?, 2, 2, 512) | output: | (?, 1, 1, 512)
sequential_10: Sequential | input: | (?, 1, 1, 512) | output: | (?, 2, 2, 512)
concatenate: Concatenate | input: | [(?, 2, 2, 512), (?, 2, 2, 512)] | output: | (?, 2, 2, 1024)
sequential_11: Sequential | input: | (?, 2, 2, 1024) | output: | (?, 4, 4, 512)
concatenate_1: Concatenate | input: | [(?, 4, 4, 512), (?, 4, 4, 512)] | output: | (?, 4, 4, 1024)
sequential_12: Sequential | input: | (?, 4, 4, 1024) | output: | (?, 8, 8, 512)
concatenate_2: Concatenate | input: | [(?, 8, 8, 512), (?, 8, 8, 512)] | output: | (?, 8, 8, 1024)
sequential_13: Sequential | input: | (?, 8, 8, 1024) | output: | (?, 16, 16, 512)
concatenate_3: Concatenate | input: | [(?, 16, 16, 512), (?, 16, 16, 512)] | output: | (?, 16, 16, 1024)
sequential_14: Sequential | input: | (?, 16, 16, 1024) | output: | (?, 32, 32, 256)
concatenate_4: Concatenate | input: | [(?, 32, 32, 256), (?, 32, 32, 256)] | output: | (?, 32, 32, 512)
sequential_15: Sequential | input: | (?, 32, 32, 512) | output: | (?, 64, 64, 128)
concatenate_5: Concatenate | input: | [(?, 64, 64, 128), (?, 64, 64, 128)] | output: | (?, 64, 64, 256)

```
gen_output = generator(inp[tf.newaxis,...], training=False)
plt.imshow(gen_output[0,...])
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
<matplotlib.image.AxesImage at 0x7fd03a255160>
```



Generator loss:

GANs try to replicate a probability distribution, therefore use loss function that reflect the distance between the distribution of the data generated by the GAN and the distribution of the real data.It is a sigmoid cross entropy loss of the generated images and an **array of ones**. This allows the generated image to become structurally similar to the target image.

```
LAMBDA = 100
```

```
def generator_loss(disc_generated_output, gen_output, target):
  gan_loss = loss_object(tf.ones_like(disc_generated_output), disc_generated_output)

  # mean absolute error
  l1_loss = tf.reduce_mean(tf.abs(target - gen_output))

  total_gen_loss = gan_loss + (LAMBDA * l1_loss)

  return total_gen_loss, gan_loss, l1_loss
```

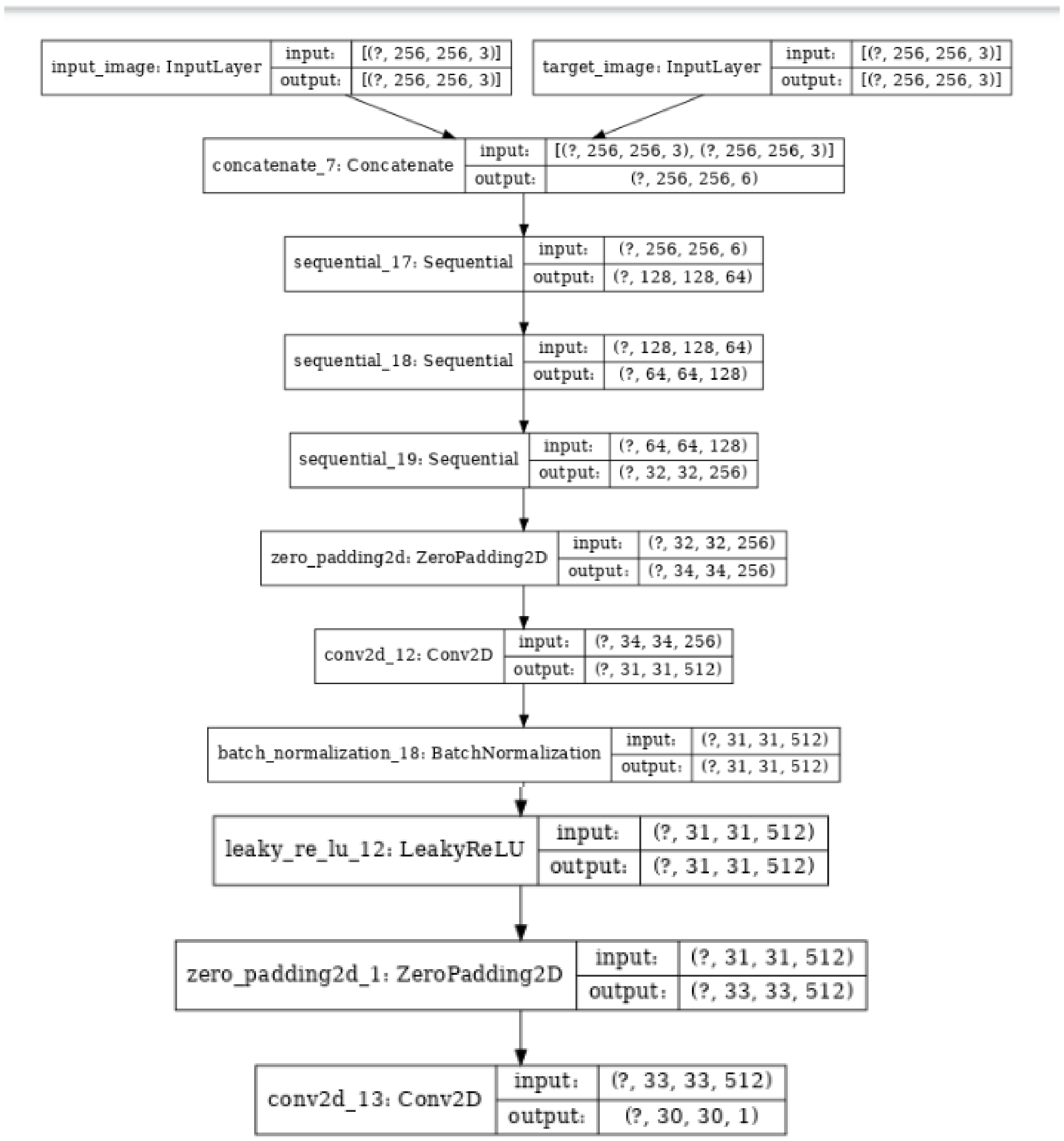
7. **Build a Discriminator:**

The discriminator is trained directly on real an generated image and is responsible for classifying images as real or fake. So the **Discriminator()** receives 2 inputs.

- Input image and the target image, which it should classify as real.
- Input image and the generated image (output of generator), which it should classify as fake.
- We concatenate these 2 inputs together in the code (tf.concat([inp, tar], axis=-1)

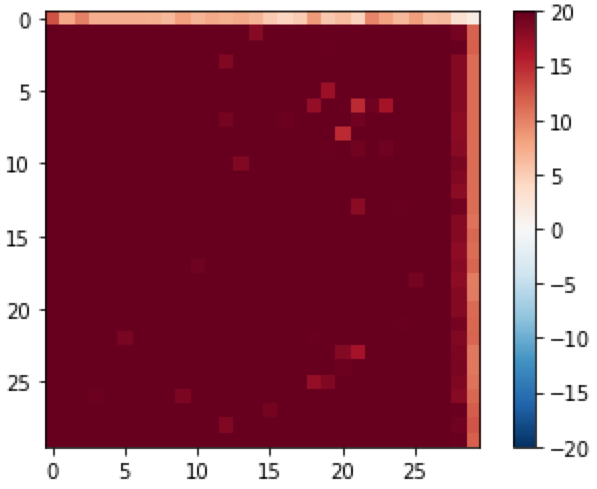
```
def Discriminator():
  initializer = tf.random_normal_initializer(0., 0.02)

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

  x = tf.keras.layers.concatenate([inp, tar]) # (bs, 256, 256, channels*2)

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

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

  batchnorm1 = tf.keras.layers.BatchNormalization()(conv)

  leaky_relu = tf.keras.layers.LeakyReLU()(batchnorm1)

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

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

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

```
discriminator = Discriminator()
tf.keras.utils.plot_model(discriminator, show_shapes=True, dpi=64)
```


input_image: InputLayer
input: [(?, 256, 256, 3)]
output: [(?, 256, 256, 3)]
target_image: InputLayer
input: [(?, 256, 256, 3)]
output: [(?, 256, 256, 3)]
concatenate_7: Concatenate
input: [(?, 256, 256, 3), (?, 256, 256, 3)]
output: (?, 256, 256, 6)
sequential_17: Sequential
input: (?, 256, 256, 6)
output: (?, 128, 128, 64)
sequential_18: Sequential
input: (?, 128, 128, 64)
output: (?, 64, 64, 128)
sequential_19: Sequential
input: (?, 64, 64, 128)
output: (?, 32, 32, 256)
zero_padding2d: ZeroPadding2D
input: (?, 32, 32, 256)
output: (?, 34, 34, 256)
conv2d_12: Conv2D
input: (?, 34, 34, 256)
output: (?, 31, 31, 512)
batch_normalization_18: BatchNormalization
input: (?, 31, 31, 512)
output: (?, 31, 31, 512)
leaky_re_lu_12: LeakyReLU
input: (?, 31, 31, 512)
output: (?, 31, 31, 512)
zero_padding2d_1: ZeroPadding2D
input: (?, 31, 31, 512)
output: (?, 33, 33, 512)
conv2d_13: Conv2D
input: (?, 33, 33, 512)
output: (?, 30, 30, 1)

```
disc_out = discriminator([inp[tf.newaxis,...], gen_output], training=False)
plt.imshow(disc_out[0,...,-1], vmin=-20, vmax=20, cmap='RdBu_r')
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x7fd03a2c9d68>
```

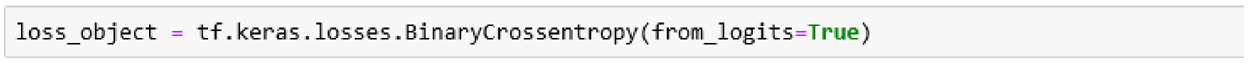


Discriminator loss:

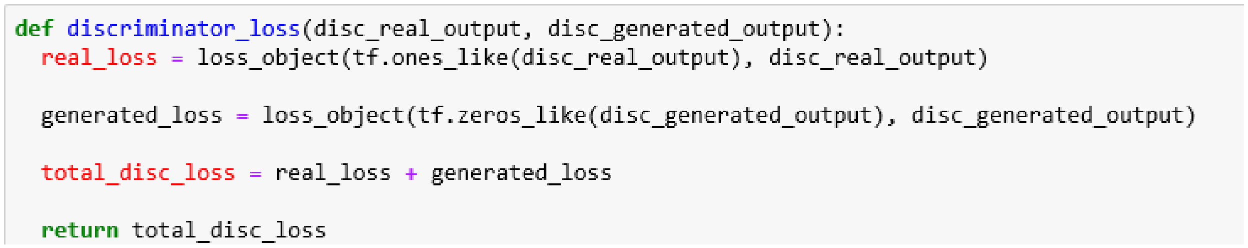
As the generator as loss function as well as discriminator also has loss function The discriminator loss function takes 2 inputs; real images, generated images real_loss is a sigmoid cross entropy loss of the real images and an array of ones(since these are the real images) generated_loss is a sigmoid cross entropy loss of the generated images and an array of zeros(since these are the fake images). Then the total_loss is the sum of real_loss and the generated_loss.

8. Optimizers and Checkpoint-saver:

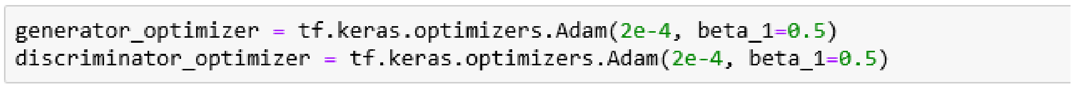
Optimizers are the extended class, which include added information to train a specific model. It is used for improving speed and performance for training a specific model.Checkpoint are binary files in a proprietary format which map variable names to tensor values. The best way to examine the contents of a checkpoint is to load it using a server. Server can automatically numbers checkpoint filenames with counters.



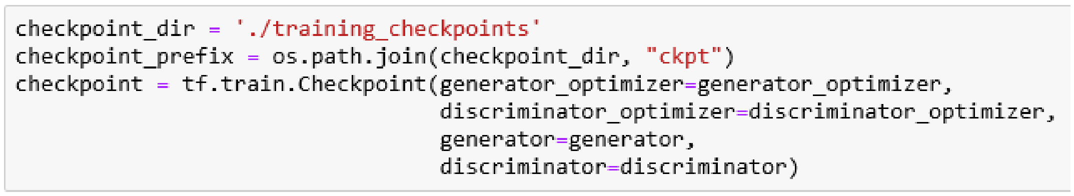
```
loss_object = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```



```
def discriminator_loss(disc_real_output, disc_generated_output):
  real_loss = loss_object(tf.ones_like(disc_real_output), disc_real_output)

  generated_loss = loss_object(tf.zeros_like(disc_generated_output), disc_generated_output)

  total_disc_loss = real_loss + generated_loss

  return total_disc_loss
```



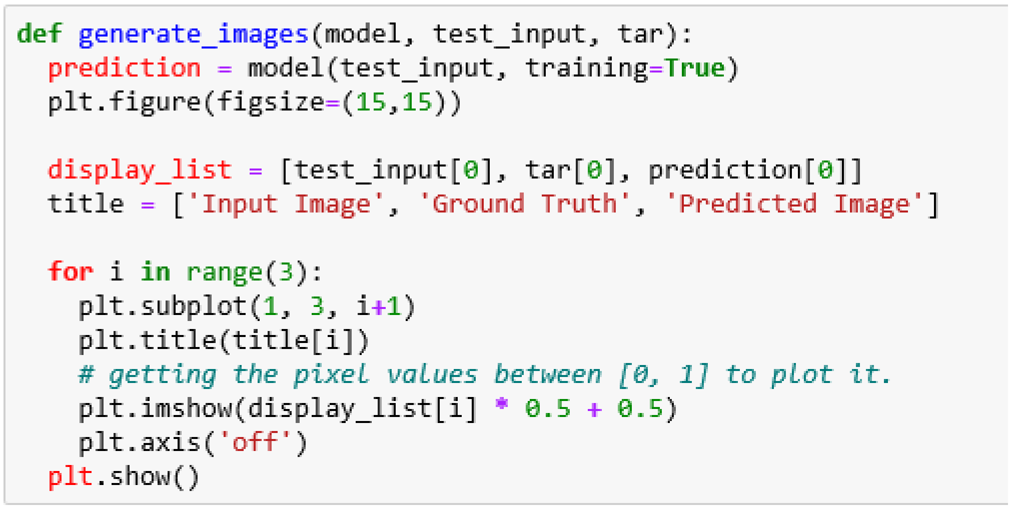
```
generator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
discriminator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
```



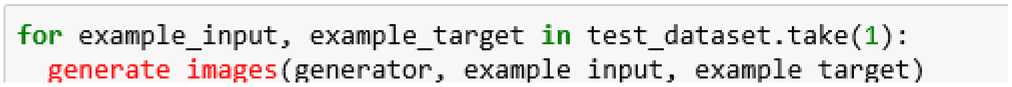
```
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                 discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)
```

9. Generate Images:

Generate_image() function is used to generate images during training. For this we pass images from the test dataset to the generator. The generator will then translate the input image into the output. Last step is to plot the predictions and **voila.**

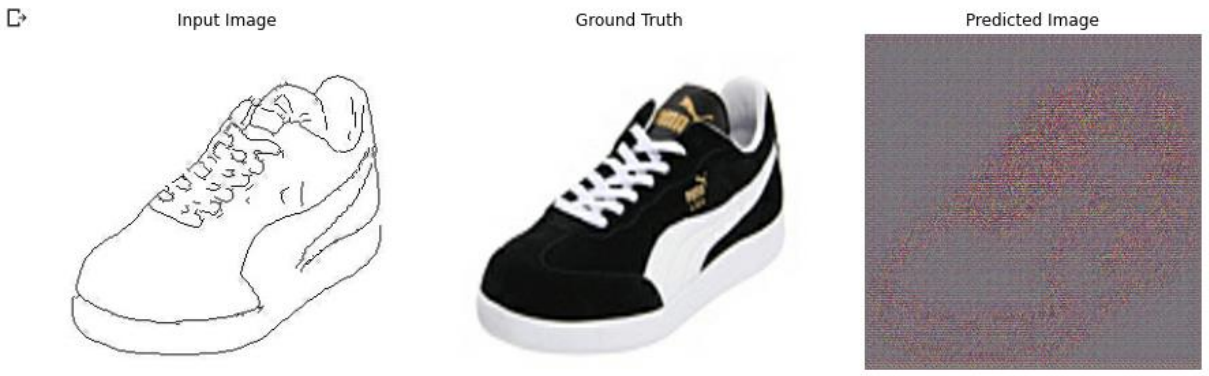


```
def generate_images(model, test_input, tar):
  prediction = model(test_input, training=True)
  plt.figure(figsize=(15,15))

  display_list = [test_input[0], tar[0], prediction[0]]
  title = ['Input Image', 'Ground Truth', 'Predicted Image']

  for i in range(3):
    plt.subplot(1, 3, i+1)
    plt.title(title[i])
    # getting the pixel values between [0, 1] to plot it.
    plt.imshow(display_list[i] * 0.5 + 0.5)
    plt.axis('off')
  plt.show()
```



```
for example_input, example_target in test_dataset.take(1):
  generate_images(generator, example_input, example_target)
```

Output is:

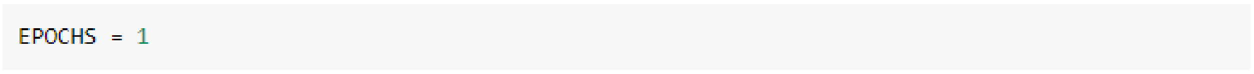


10. Training:

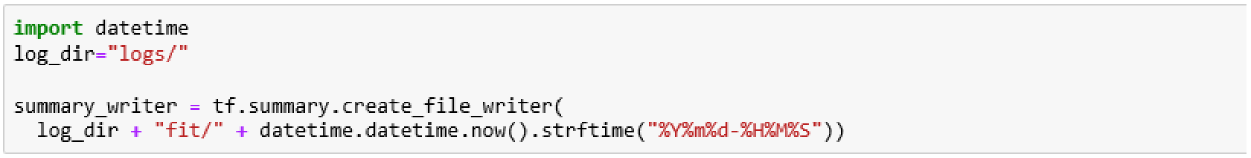
To train our dataset discriminator will be used and do the following steps:

- The discriminator receives the input_image and the generated image as the first input. The second input is the input_image and the target_image.
- Next, we calculate the generator and the discriminator loss.
- Then, we calculate the gradients of loss with respect to both the generator and the discriminator variables(inputs) and apply those to the optimizer.

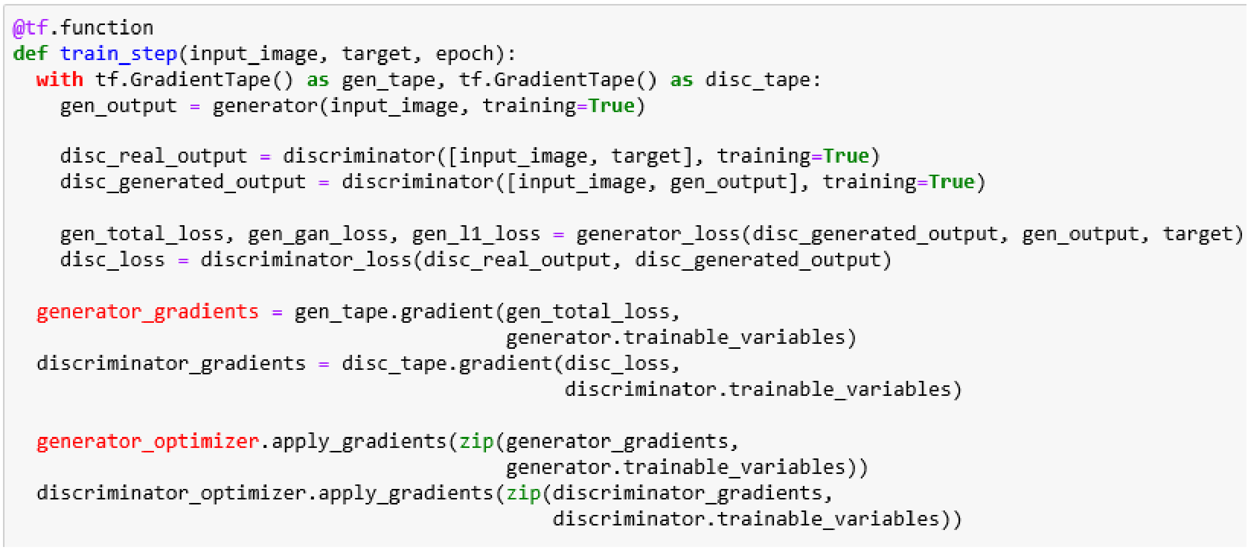
- Then log the losses to TensorBoard.



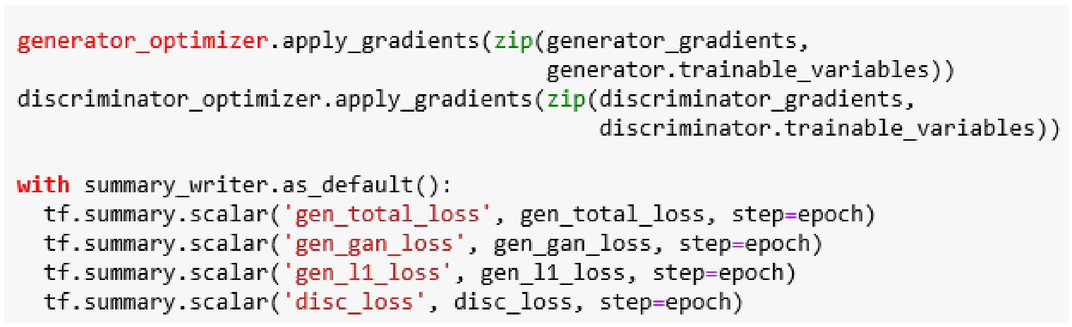
```
EPOCHS = 1
```



```
import datetime
log_dir="logs/"

summary_writer = tf.summary.create_file_writer(
  log_dir + "fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
```



```
@tf.function
def train_step(input_image, target, epoch):
  with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
    gen_output = generator(input_image, training=True)

    disc_real_output = discriminator([input_image, target], training=True)
    disc_generated_output = discriminator([input_image, gen_output], training=True)

    gen_total_loss, gen_gan_loss, gen_l1_loss = generator_loss(disc_generated_output, gen_output, target)
    disc_loss = discriminator_loss(disc_real_output, disc_generated_output)

  generator_gradients = gen_tape.gradient(gen_total_loss,
                                          generator.trainable_variables)
  discriminator_gradients = disc_tape.gradient(disc_loss,
                                               discriminator.trainable_variables)

  generator_optimizer.apply_gradients(zip(generator_gradients,
                                          generator.trainable_variables))
  discriminator_optimizer.apply_gradients(zip(discriminator_gradients,
                                              discriminator.trainable_variables))
```



```
generator_optimizer.apply_gradients(zip(generator_gradients,
                                        generator.trainable_variables))
discriminator_optimizer.apply_gradients(zip(discriminator_gradients,
                                            discriminator.trainable_variables))

with summary_writer.as_default():
  tf.summary.scalar('gen_total_loss', gen_total_loss, step=epoch)
  tf.summary.scalar('gen_gan_loss', gen_gan_loss, step=epoch)
  tf.summary.scalar('gen_l1_loss', gen_l1_loss, step=epoch)
  tf.summary.scalar('disc_loss', disc_loss, step=epoch)
```

The actual training loop:

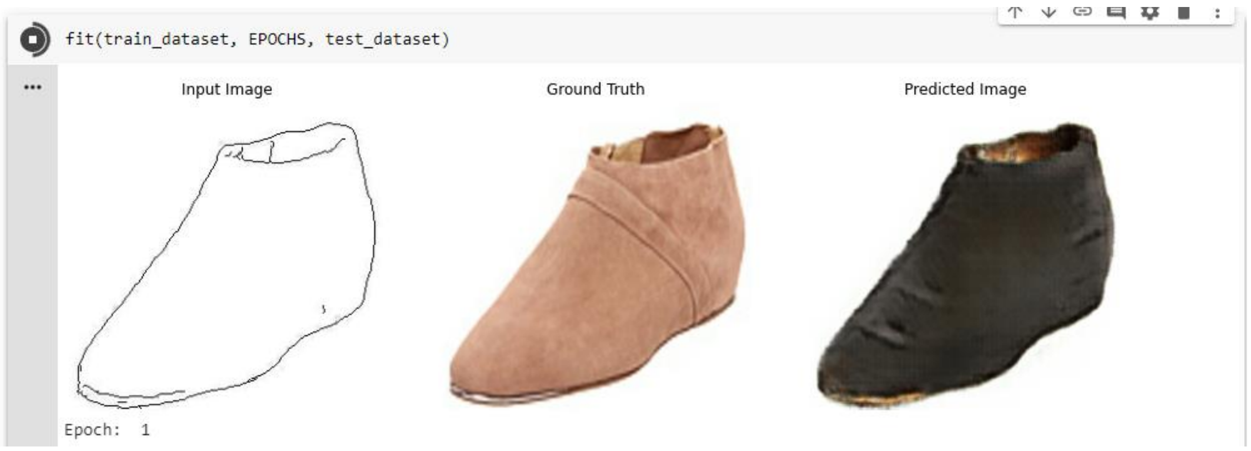
- Iterates over the number of epochs.
- On each epoch it clears the display, and runs generate_images to show it's progress.
- On each epoch it iterates over the training dataset, printing a '.' for each example.
- It saves a checkpoint every 20 epochs.

```
def fit(train_ds, epochs, test_ds):
  for epoch in range(epochs):
    start = time.time()

    display.clear_output(wait=True)

    for example_input, example_target in test_ds.take(1):
      generate_images(generator, example_input, example_target)
    print("Epoch: ", epoch)

    # Train
    for n, (input_image, target) in train_ds.enumerate():
      print('.', end='')
      if (n+1) % 100 == 0:
        print()
      train_step(input_image, target, epoch)
    print()

    # saving (checkpoint) the model every 20 epochs
    if (epoch + 1) % 20 == 0:
      checkpoint.save(file_prefix = checkpoint_prefix)

    print ('Time taken for epoch {} is {} sec\n'.format(epoch + 1,
                                                        time.time()-start))
  checkpoint.save(file_prefix = checkpoint_prefix)
```

```
:  #docs_infra: no_execute
   %load_ext tensorboard
   %tensorboard --logdir {log_dir}
```

Now run the training loop:

```
:  fit(train_dataset, EPOCHS, test_dataset)
```



```
fit(train_dataset, EPOCHS, test_dataset)
```

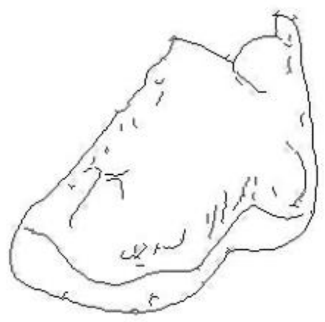
```
Epoch:  1
```

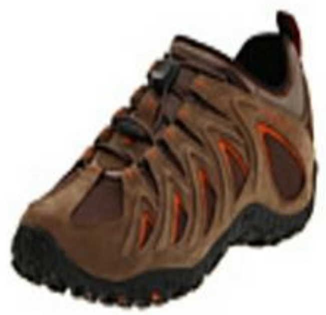
11. **Restore the latest checkpoint and test:**

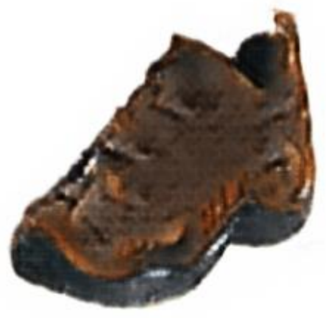
```
# restoring the latest checkpoint in checkpoint_dir
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))
```

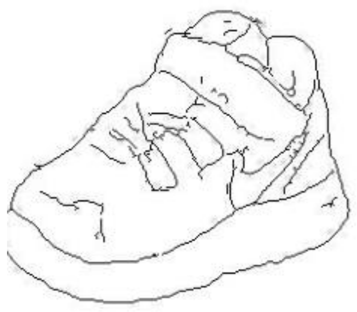
```
# Run the trained model on a few examples from the test dataset
for inp, tar in test_dataset.take(5):
  generate_images(generator, inp, tar)
```

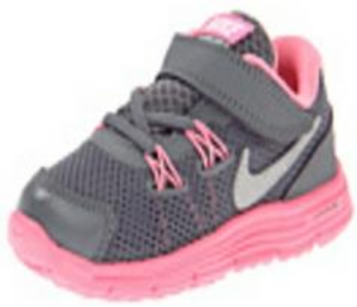
12. **Generate using test dataset**

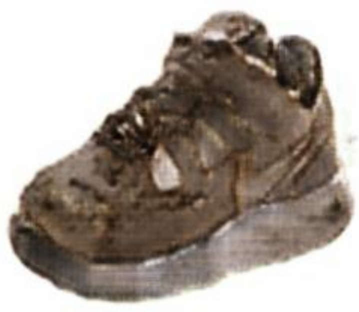

Input Image

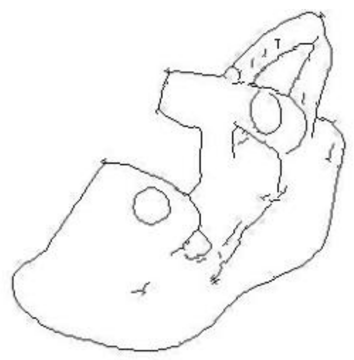

Ground Truth

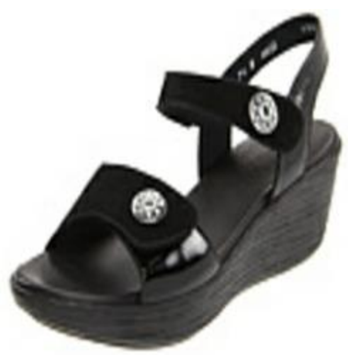

Predicted Image

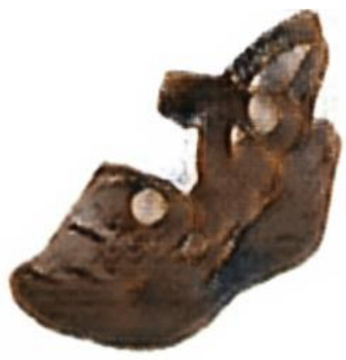

Input Image


Ground Truth


Predicted Image


Input Image


Ground Truth


Predicted Image