TECHNOLOGICAL INSTITUTE OF THE PHILIPPINES
938 Aurora Blvd., Cubao, Quezon City

COLLEGE OF ENGINEERING AND ARCHITECTURE
ELECTRONICS ENGINEERING DEPARTMENT

1ST SEMESTER SY 2022 - 2023

Prediction and Machine Learning

COE 005
ECE41S11

Midterm Exam
Exercise
Generative Adversarial Network (GAN)
Semantic-to-Image-to-Photo Translation

Submitted to:
Engr. Christian Lian Paulo Rioflorido

Submitted on:
10/21/2022

Submitted by:
David Luneta

Data and Results:

In this exersice it will illustrate how to build and train a conditional generative adversarial network (cGAN) in semantic image to photo translation. This simulation used a Index of pix2pix datasets called cityscapes with a size of 99M that later on used in predicting the output of photo realistic image from the semantic layout.

In the image belows I first import the needed libraries for this simulation like TensorFlow

Luneta_GAN.ipynb

File Edit View Insert Runtime Tools Help Last edited on October 20

```
import tensorflow as tf

import os
import pathlib
import time
import datetime

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

In this part of the code, the data was gathered and import from the website efrosgan.eecs.berkeley.edu/pix2pix/datasets/. It has a 6 datasets available in the website the cityscapes, edges2handbags, edges2shoes, facades, maps and night2day. I chose the datasets called cityscapes because it size is only 99M smaller that the other datasets

```
dataset_name = "cityscapes"
```

For this part I copy the link of the datasets I needed for the program and then it will automatically unzip the data I get from the website and download it.

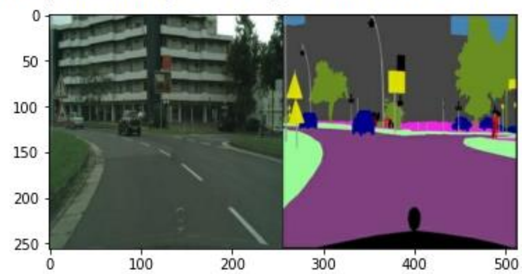
```
_URL = f'http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/cityscapes.tar.gz'

path_to_zip = tf.keras.utils.get_file(
    fname=f"{dataset_name}.tar.gz",
    origin=_URL,
    extract=True)

path_to_zip  = pathlib.Path(path_to_zip)

PATH = path_to_zip.parent/dataset_name
```

```
Downloading data from http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/cityscapes.tar.gz
103441232/103441232 [==============================] - 103s 1us/step
```

It is the non-windows files system paths It reads the entire contents of the datasets images then returns with a bit representation of the image. Then create an image from a 2-dimensional NumPy array. I saw that from the left image it is a photorealist image while compared to the right image it is a sematic layout of the road generate from the datasets

```
[ ] sample_image = tf.io.read_file(str(PATH / 'train/1.jpg'))
    sample_image = tf.io.decode_jpeg(sample_image)
    print(sample_image.shape)

    (256, 512, 3)
```

```
plt.figure()
plt.imshow(sample_image)

<matplotlib.image.AxesImage at 0x7fd748c607d0>
```



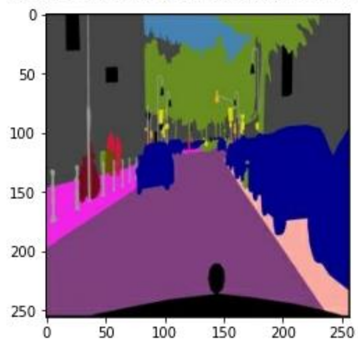
Creating a functions that will reads the image files and returns two image tensors. And it will convert two images to float32 tensors

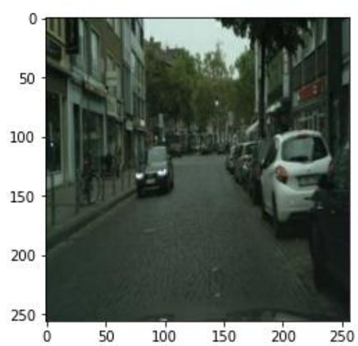
```
#Define a function that loads image files and outputs two image tensors:
def load(image_file):
  # Read and decode an image file to a uint8 tensor
  image = tf.io.read_file(image_file)
  image = tf.io.decode_jpeg(image)

  # Split each image tensor into two tensors:
  # - one with a real building facade image
  # - one with an architecture label image
  w = tf.shape(image)[1]
  w = w // 2
  input_image = image[:, w:, :]
  real_image = image[:, :w, :]

  # Convert both images to float32 tensors
  input_image = tf.cast(input_image, tf.float32)
  real_image = tf.cast(real_image, tf.float32)

  return input_image, real_image
```

Plotting the sample of the input and real images. In the first picture a notice that it is a semantic layout of the car, road and the building with the representation of the real image.

```
inp, re = load(str(PATH / 'train/100.jpg'))
# Casting to int for matplotlib to display the images
plt.figure()
plt.imshow(inp / 255.0)
plt.figure()
plt.imshow(re / 255.0)
```

```
<matplotlib.image.AxesImage at 0x7fd748e5b210>
```





Setting the image to size of 256x256 so that it will not have a problem in executing the coding in later on.

```
BUFFER_SIZE = 400
# The batch size of 1 produced better results for the U-Net in the original pix2pix experiment
BATCH_SIZE = 1
# Each image is 256x256 in size
IMG_WIDTH = 256
IMG_HEIGHT = 256
```

From the original image size to resize image

```
[ ] 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
```

Crop the image with the set image width and height 256x256 and coordinates the width and height of the image to [-1,1]

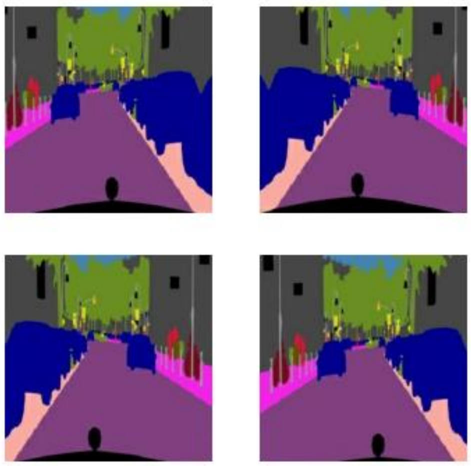
```
[ ] def random_crop(input_image, real_image):
      stacked_image = tf.stack([input_image, real_image], axis=0)
      cropped_image = tf.image.random_crop(
          stacked_image, size=[2, IMG_HEIGHT, IMG_WIDTH, 3])

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

```
[ ] # 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
```

```
@tf.function()
def random_jitter(input_image, real_image):
  # Resizing to 286x286
  input_image, real_image = resize(input_image, real_image, 286, 286)

  # Random cropping back to 256x256
  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
```

Setting the figure size into 6, 6. Allowing us to specify the width and height of the figure in unit inches. We can now see the preprocess visualize data of the cityscapes from our datasets

```
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()
```



Building an input pipeline with tf.data that will distribute file system and apply random perturbation to each image, and merge randomly selected image into a batch training.

```
#Build an input pipeline with tf.data
train_dataset = tf.data.Dataset.list_files(str(PATH / 'train/*.jpg'))
train_dataset = train_dataset.map(load_image_train,
                                  num_parallel_calls=tf.data.AUTOTUNE)
train_dataset = train_dataset.shuffle(BUFFER_SIZE)
train_dataset = train_dataset.batch(BATCH_SIZE)
```

```
try:
  test_dataset = tf.data.Dataset.list_files(str(PATH / 'test/*.jpg'))
except tf.errors.InvalidArgumentError:
  test_dataset = tf.data.Dataset.list_files(str(PATH / 'val/*.jpg'))
test_dataset = test_dataset.map(load_image_test)
test_dataset = test_dataset.batch(BATCH_SIZE)
```

After importing the necessary modules, setting the image size, visualizing the data, setting the necessary code needed. We now proceed to building the generator network for the pix2pix cGAN. Now we should define the encoder from the U-Net.

```
#Build the generator
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
```

After defining the decoder we next to define the encoder

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

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

```
[ ] #Define the upsampler (decoder):
    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)
```

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

+ Code + Text

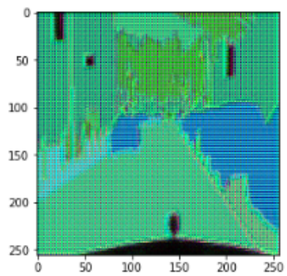
After setting the downsampler(encoder) and upsampler(decoder) we now define the generator this generator takes random noise as input and tries to recreate the images from the data sets. After defining the generator it will now proceed to downsampling through the model and Upsampling and establishing the skip connections

```
[ ] #Define the generator with the downsampler and the upsampler:
    def Generator():
      inputs = tf.keras.layers.Input(shape=[256, 256, 3])

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

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

The image below shows a visualization of the generator model architecture and will test the generator. The coloration of the generator model has new synthetic images as shown in the image below

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

```
WARNING:matplotlib.image: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 0x7fd6e4248890>
```



Define the generator loss in which the distance of the data generated by the GAN and the distribution of the real data of the cityscapes

```
[ ] #Define the generator loss
    LAMBDA = 100
```

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

```
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
```

Building the discriminator so that the system will distinguish the real data created by the generator

```
[ ] #Build the discriminator
    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])  # (batch_size, 256, 256, channels*2)

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

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

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

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

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

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

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

While defining the discriminator loss can help the system to classify the real image or correctly label the fake image that comes from the generator

```
[ ] 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
```

```
[ ] #Define the optimizers and a checkpoint-saver
    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)
```

By building the Generator and Discriminator network it is now ready for the training of predicting the image from a semantic layout.

```
[ ] #Training
    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, step):
      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))

      with summary_writer.as_default():
        tf.summary.scalar('gen_total_loss', gen_total_loss, step=step//1000)
        tf.summary.scalar('gen_gan_loss', gen_gan_loss, step=step//1000)
        tf.summary.scalar('gen_l1_loss', gen_l1_loss, step=step//1000)
        tf.summary.scalar('disc_loss', disc_loss, step=step//1000)
```

```
def fit(train_ds, test_ds, steps):
  example_input, example_target = next(iter(test_ds.take(1)))
  start = time.time()

  for step, (input_image, target) in train_ds.repeat().take(steps).enumerate():
    if (step) % 1000 == 0:
      display.clear_output(wait=True)

      if step != 0:
        print(f'Time taken for 1000 steps: {time.time()-start:.2f} sec\n')

      start = time.time()

      generate_images(generator, example_input, example_target)
      print(f"Step: {step//1000}k")

    train_step(input_image, target, step)

    # Training step
    if (step+1) % 10 == 0:
      print('.', end='', flush=True)


    # Save (checkpoint) the model every 5k steps
    if (step + 1) % 5000 == 0:
      checkpoint.save(file_prefix=checkpoint_prefix)
```

In the image below is a output of the training data with the steps of 40000 so that it will have a better predicted output of the semantic layout, I tried the 7000steps for fitting the data but the predict result is blurry than the 40000steps. I'm thinking if I will to too 100K of steps but I'm worry for my pc that might hang.

```
[ ] fit(train_dataset, test_dataset, steps=40000)
```

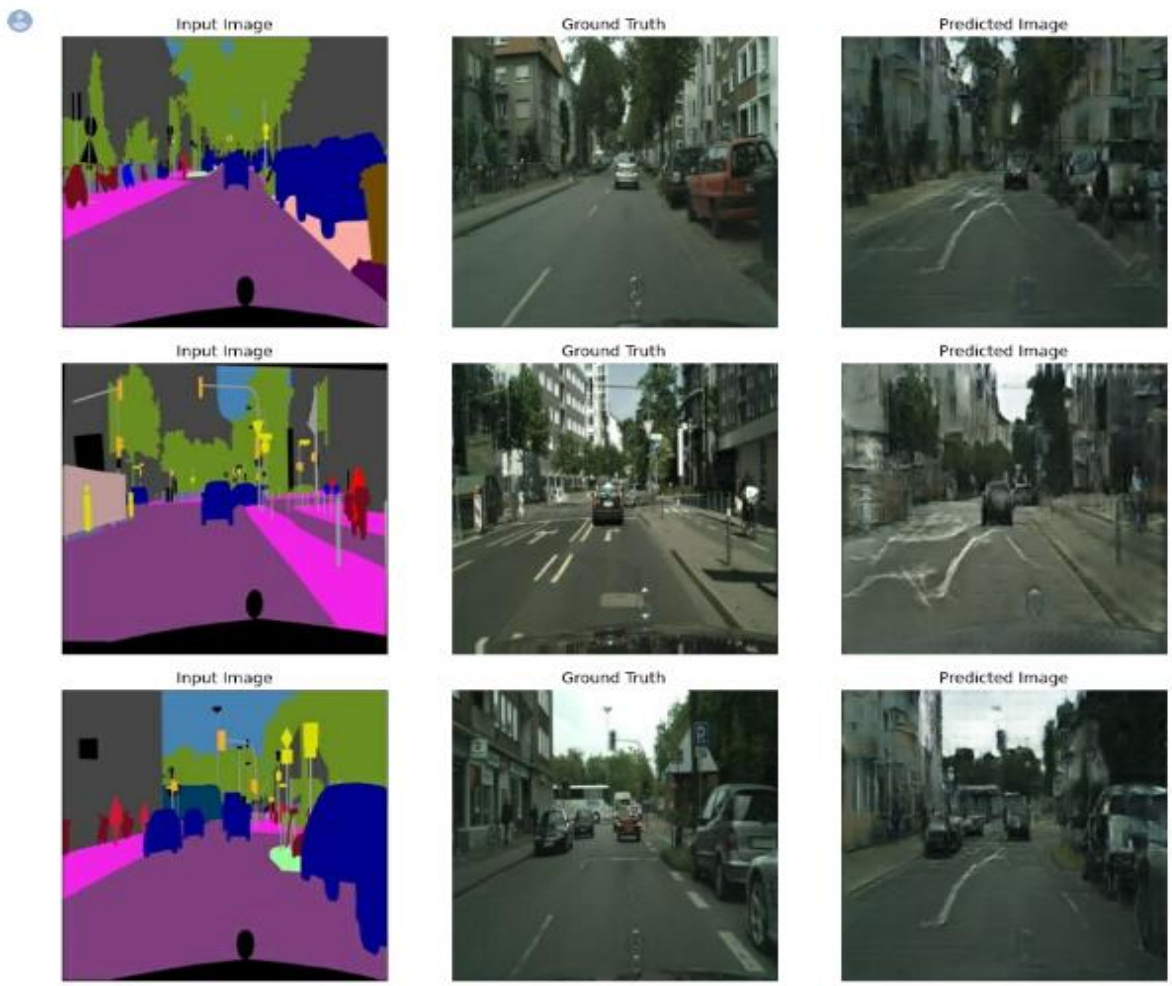
```
Time taken for 1000 steps: 94.74 sec
```

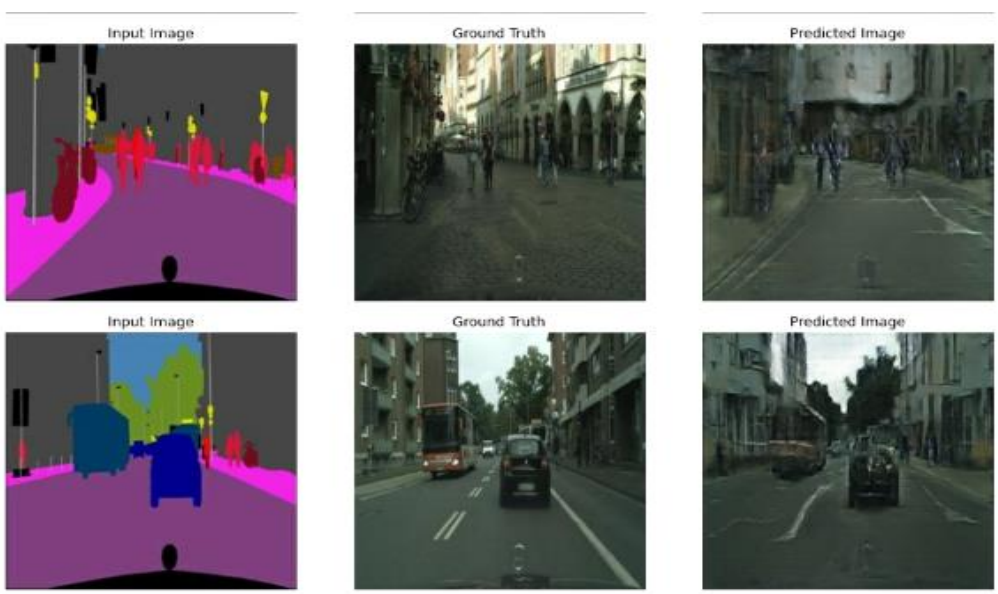

```
Step: 39k
.......................................................................................................
```

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

In conclusion the one of the applications of GAN in semantic to photo translation shows that it obtain the structural correlations between the image with the help of comparison of the image from the semantic input image, ground truth and the predicted output image. To sum it up GAN is a powerful model for image generation as seen in the simulation of the cityscapes.

The Final output or the generated images using the test set of semantic to photo translation in random data sets of cityscapes


Input Image
Ground Truth
Predicted Image
Input Image
Ground Truth
Predicted Image
Input Image
Ground Truth
Predicted Image


Input Image
Ground Truth
Predicted Image
Input Image
Ground Truth
Predicted Image

References:

Liu, J., Zou, Y., & Yang, D. (2020, May 1). Semanticgan: Generative Adversarial Networks For Semantic Image To Photo-Realistic Image Translation. IEEE Xplore. https://doi.org/10.1109/ICASSP40776.2020.9053087

Nair, A. (2019, August 13). How To Build A Generative Adversarial Network In 8 Simple Steps. Analytics India Magazine. https://analyticsindiamag.com/how-to-build-a-generative-adversarial-network-in-8-simple-steps/

Pix2Pix | TensorFlow Core. (n.d.). TensorFlow. https://www.tensorflow.org/tutorials/generative/pix2pix

Dataset: http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/