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

GAN Exercise: Semantic-Image-to-Photo Translation
Submitted to:
Engr. Christian Lian Paulo Riofloridо

Submitted on:
10/22/2022

Submitted by:
John Michael Valad-on

Selected GAN Application: ***Semantic-Image-to-Photo Translation***

(Results)

Data and Results:

The Experiment used cityscapes datasets to show the semantic image to photo translation. The input image will process into segmentation model to get corresponding semantic label map. This focuses on semantic understanding of urban street scenes, providing a semantic label and corresponding photo.

jupyter Valad-on_gan Last Checkpoint: 4 hours ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Run Code

```
In [1]: import tensorflow as tf

        import os
        import pathlib
        import time
        import datetime

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

(Import Necessary Libraries)

```
In [2]: dataset_name = "cityscapes"

In [3]: _URL = f'http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/cityscapes.tar.gz'

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

        path_to_zip  = pathlib.Path(path_to_zip)

        PATH = path_to_zip.parent/dataset_name

In [4]: list(PATH.parent.iterdir())

Out[4]: [WindowsPath('C:/Users/jmval/.keras/datasets/cityscapes'),
         WindowsPath('C:/Users/jmval/.keras/datasets/cityscapes.tar.gz')]
```

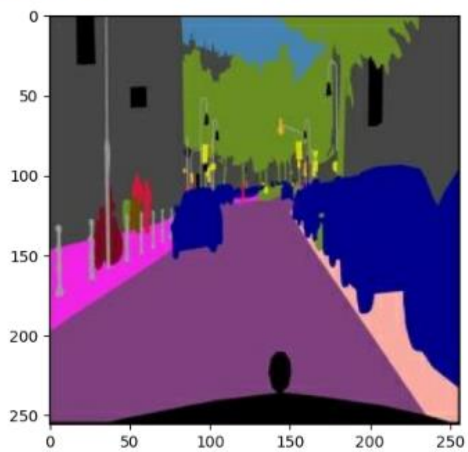
(Gathering and importing of Data)

The data was able to obtain through the website name efrosgans. It consists of images of cityscapes which will be our data and import it by directly downloading it using tf.keras.utils.get_file. The data pre-processing can start by unzipping the downloaded file.

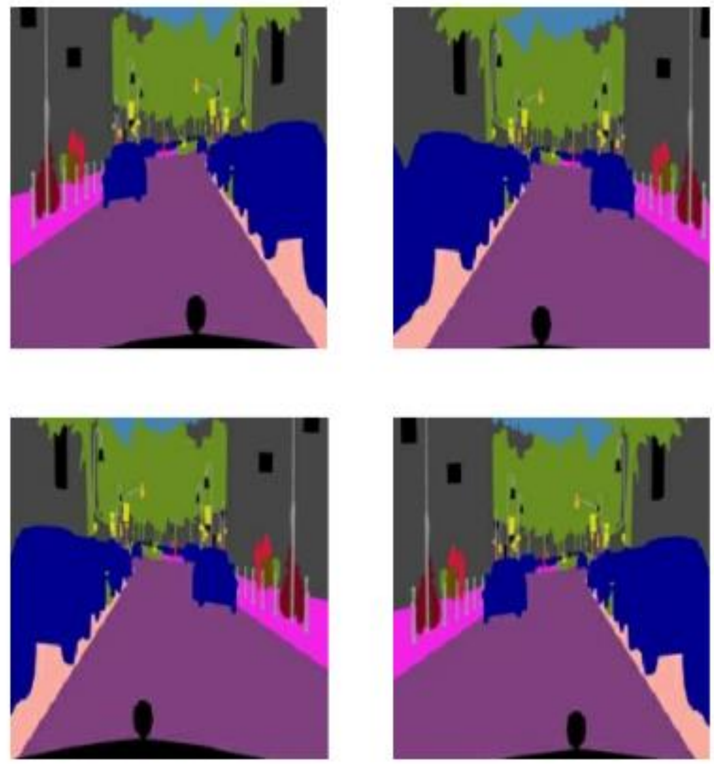
```python
In [7]: def load(image_file):
          # Read and decode an image file to a uint8 tensor
          image = tf.io.read_file(image_file)
          image = tf.io.decode_jpeg(image)

          # 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
```

```python
In [8]: 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)
```

Out[8]: <matplotlib.image.AxesImage at 0x2843a246310>




```python
In [14]: 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()
```



(Visualization of the data)

In this code, we are able to split the image of the real cityscapes from the semantic label image by defining the two image tensors. Hence, the image here is our input data with semantic labeling.

```python
In [17]: 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)
```

```python
In [18]: 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)
```

(Build an input pipeline with tf.data)

```
In [24]: 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)

In [25]: generator = Generator()
         tf.keras.utils.plot_model(generator, show_shapes=True, dpi=64)
```

(Build the model)

Generator model generates new synthetic images. it aids in gathering the structural correlation data between textures, which directs the generator to create satisfying textures.

```
In [27]: LAMBDA = 100

In [28]: loss_object = tf.keras.losses.BinaryCrossentropy(from_logits=True)

In [29]: 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
```

(Define generator loss)

The generated image can resemble the target image structurally thanks to the generator loss, which is a sigmoid cross-entropy loss of the generated images and an array of ones.

```
In [30]: 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)
```

```
In [31]: discriminator = Discriminator()
         tf.keras.utils.plot_model(discriminator, show_shapes=True, dpi=64)
```

(Build Discriminator)

The discriminator classifies each image whether it is real or not real. Hence, discriminator have two inputs.

```
In [33]: 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 Discriminator loss)

A sigmoid cross-entropy loss of the real images and an array of ones is called real loss, while a loss of the generated images and an array of zeros is called generated loss (since these are the fake images). Finally, the total loss is calculated as the sum of the genuine loss and the created loss.

```
In [39]: @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)
```

```
In [40]: 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 [42]: fit(train_dataset, test_dataset, steps=40000)

Time taken for 1000 steps: 187.60 sec
```


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

(Training of data)

In this code, we train the data for each input which generates the predicted image. it also shows at the real photo which will be our basis to our predicted image. in the training process, the input image and the created image serve as the discriminator's first input. The target image and input image make up the second input. It Calculate the discriminator and generator losses. The gradients of loss are then computed with respect to the generator and discriminator variables (inputs) and applied to the optimizer.

```
In [46]: # 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)
```

(Generated images results)

<u>Discussion</u>

A specific kind of generative adversarial network is the cGAN. Two models that compete with one another make up this kind of architecture. the discriminator network, which tries to discern between authentic and fraudulent images, and the generator model, which creates fresh artificial images. The two models compete with one another because the generator seeks to produce images that are convincing enough to trick the discriminator network while the discriminator functions as a loss function, comparing the created image to an ever-improving evaluator rather than the original source data. As a result, the two models are trained concurrently through an adversarial process, making it difficult to distinguish between real and fake images as the networks are taught.

In conclusion, a realistic image can be produced using conditional GANs from an input semantic drawing. These networks learn a loss function to train in addition to learning the mapping from input image to output image. This method is useful for a variety of applications including cityscape, apartments, human face, scenic environments, colorizing photographs, constructing objects from edge maps, and synthesizing photos from label maps and vehicles whose photorealistic translations can be generated with the semantic input provided.

Reference:

[1] Tsang, S.-H. (2020, December 25). *[review] pix2pix: Image-to-image translation with conditional adversarial networks (GAN)*. Medium. Retrieved October 21, 2022, from https://sh-tsang.medium.com/review-pix2pix-image-to-image-translation-with-conditional-adversarial-networks-gan-ac85d8ecead2

[2] Colab, S. van. (2020, April 17). *Our take on image-to-image translation with conditional adversarial networks*. Medium. Retrieved October 21, 2022, from https://medium.com/@sigurdcolab/our-take-on-image-to-image-translation-with-conditional-adversarial-networks-e870c6b33a48