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
Generative Adversarial Networks (GANs)

Submitted to:
Engr. Christian Lian Paulo Rioflorido

Submitted on:
OCTOBER 22, 2022

Submitted by:
RHAN VINCENT M. AUSTRIA

Introduction

A Generative Adversarial Network, or GAN, is a type of neural network architecture for generative modeling. Generative modeling involves using a model to generate new examples that plausibly come from an existing distribution of samples, such as generating new photographs that are similar but specifically different from a dataset of existing photographs.

A GAN is a generative model that is trained using two neural network models. One model is called the "generator" or "generative network" model that learns to generate new plausible samples. The other model is called the "discriminator" or "discriminative network" and learns to differentiate generated examples from real examples.

CycleGAN uses a cycle consistency loss to enable training without the need for paired data. In other words, it can translate from one domain to another without a one-to-one mapping between the source and target domain.

This opens up the possibility to do a lot of interesting tasks like photo-enhancement, image colorization, style transfer, etc. All you need is the source and the target dataset (which is simply a directory of images).

Code

Imports

```
[ ]  import tensorflow as tf
```

```
[ ]  import tensorflow_datasets as tfds
     from tensorflow_examples.models.pix2pix import pix2pix

     import os
     import time
     import matplotlib.pyplot as plt
     from IPython.display import clear_output

     AUTOTUNE = tf.data.AUTOTUNE
```

We first import tensorflow. This will help implement best practices for data automation, model tracking, performance monitoring, and model retraining. Proceeding to import os. Importing os allows us to use our operating system and provides or allows functions to create, move, and/or remove the current directory.

Next, is the time. Time function takes seconds passed since epoch as an argument and returns a string representing local time. The pyplot as plt gives an unfamiliar reader a hint that pyplot is a module, rather than a function which could be incorrectly assumed from the first form.

```
[ ] dataset, metadata = tfds.load('cycle_gan/horse2zebra',
                                  with_info=True, as_supervised=True)

    train_horses, train_zebras = dataset['trainA'], dataset['trainB']
    test_horses, test_zebras = dataset['testA'], dataset['testB']
```

```
[ ] BUFFER_SIZE = 1000
    BATCH_SIZE = 1
    IMG_WIDTH = 256
    IMG_HEIGHT = 256
```

```
[ ] def random_crop(image):
      cropped_image = tf.image.random_crop(
          image, size=[IMG_HEIGHT, IMG_WIDTH, 3])

      return cropped_image
```

```
[ ] # normalizing the images to [-1, 1]
    def normalize(image):
      image = tf.cast(image, tf.float32)
      image = (image / 127.5) - 1
      return image
```

We then proceed on getting our data. As we proceed to getting our dataset, we then assign them. Lastly, we then format the images to make it the same size.

```
[ ] def random_jitter(image):
      # resizing to 286 x 286 x 3
      image = tf.image.resize(image, [286, 286],
                              method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)

      # randomly cropping to 256 x 256 x 3
      image = random_crop(image)

      # random mirroring
      image = tf.image.random_flip_left_right(image)

      return image
```

```
[ ] def preprocess_image_train(image, label):
      image = random_jitter(image)
      image = normalize(image)
      return image
```

```
[ ] def preprocess_image_test(image, label):
      image = normalize(image)
      return image
```

Next, we then add noise or jitter, while we resize, and process the image.

```
train_horses = train_horses.cache().map(
    preprocess_image_train, num_parallel_calls=AUTOTUNE).shuffle(
    BUFFER_SIZE).batch(BATCH_SIZE)

train_zebras = train_zebras.cache().map(
    preprocess_image_train, num_parallel_calls=AUTOTUNE).shuffle(
    BUFFER_SIZE).batch(BATCH_SIZE)

test_horses = test_horses.map(
    preprocess_image_test, num_parallel_calls=AUTOTUNE).cache().shuffle(
    BUFFER_SIZE).batch(BATCH_SIZE)

test_zebras = test_zebras.map(
    preprocess_image_test, num_parallel_calls=AUTOTUNE).cache().shuffle(
    BUFFER_SIZE).batch(BATCH_SIZE)
```

```
sample_horse = next(iter(train_horses))
sample_zebra = next(iter(train_zebras))
```

```
plt.subplot(121)
plt.title('Horse')
plt.imshow(sample_horse[0] * 0.5 + 0.5)

plt.subplot(122)
plt.title('Horse with random jitter')
plt.imshow(random_jitter(sample_horse[0]) * 0.5 + 0.5)
```

```
plt.subplot(121)
plt.title('Zebra')
plt.imshow(sample_zebra[0] * 0.5 + 0.5)

plt.subplot(122)
plt.title('Zebra with random jitter')
plt.imshow(random_jitter(sample_zebra[0]) * 0.5 + 0.5
```

We then input the jitter, and on the train and test horse, zebra. Then we visualize the outcome.

```
OUTPUT_CHANNELS = 3

generator_g = pix2pix.unet_generator(OUTPUT_CHANNELS, norm_type='instancenorm')
generator_f = pix2pix.unet_generator(OUTPUT_CHANNELS, norm_type='instancenorm')

discriminator_x = pix2pix.discriminator(norm_type='instancenorm', target=False)
discriminator_y = pix2pix.discriminator(norm_type='instancenorm', target=False)
```

```
to_zebra = generator_g(sample_horse)
to_horse = generator_f(sample_zebra)
plt.figure(figsize=(8, 8))
contrast = 8

imgs = [sample_horse, to_zebra, sample_zebra, to_horse]
title = ['Horse', 'To Zebra', 'Zebra', 'To Horse']

for i in range(len(imgs)):
  plt.subplot(2, 2, i+1)
  plt.title(title[i])
  if i % 2 == 0:
    plt.imshow(imgs[i][0] * 0.5 + 0.5)
  else:
    plt.imshow(imgs[i][0] * 0.5 * contrast + 0.5)
plt.show()
```

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

plt.subplot(121)
plt.title('Is a real zebra?')
plt.imshow(discriminator_y(sample_zebra)[0, ..., -1], cmap='RdBu_r')

plt.subplot(122)
plt.title('Is a real horse?')
plt.imshow(discriminator_x(sample_horse)[0, ..., -1], cmap='RdBu_r')

plt.show()
```

We then import and reuse the Pix2Pix models. Pix2Pix model is used for synthesizing photos from label maps, generating colorized photos from black and white images, turning Google Maps photos into aerial images, and even transforming sketches into photos.

The Loss Function

```
[ ]  LAMBDA = 10
```

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

```
[ ]  def discriminator_loss(real, generated):
       real_loss = loss_obj(tf.ones_like(real), real)

       generated_loss = loss_obj(tf.zeros_like(generated), generated)

       total_disc_loss = real_loss + generated_loss

       return total_disc_loss * 0.5
```

```
[ ]  def generator_loss(generated):
       return loss_obj(tf.ones_like(generated), generated)
```

```
[ ]  def calc_cycle_loss(real_image, cycled_image):
       loss1 = tf.reduce_mean(tf.abs(real_image - cycled_image))

       return LAMBDA * loss1
```

```
[ ]  def identity_loss(real_image, same_image):
       loss = tf.reduce_mean(tf.abs(real_image - same_image))
       return LAMBDA * 0.5 * loss
```

The loss function is a method of evaluating how well specific algorithm models the given data. If predictions deviate too much from actual results, loss function would cough up a very large number. Gradually, with the help of some optimization function, loss function learns to reduce the error in prediction.

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

     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)
```

We the use optimizers for our generators and discriminators. Optimizers are algorithms or methods used to change the attributes of your neural network such as weights and learning rate in order to reduce the losses. Optimizers help to get results faster.

```
[ ] checkpoint_path = "./checkpoints/train"

    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)

    ckpt_manager = tf.train.CheckpointManager(ckpt, checkpoint_path, max_to_keep=5)

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

We use checkpoint because it will help and may be used directly or as the starting point for a new run, picking up where it left off.

```
[ ] EPOCHS = 20
```

```
[ ] def generate_images(model, test_input):
      prediction = model(test_input)

      plt.figure(figsize=(12, 12))

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

      for i in range(2):
        plt.subplot(1, 2, 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()
```

Now, we proceed to train our algorithm.

```
@tf.function
def train_step(real_x, real_y):
  # persistent is set to True because the tape is used more than
  # once to calculate the gradients.
  with tf.GradientTape(persistent=True) as tape:
    # Generator G translates X -> Y
    # Generator F translates Y -> 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)

    # same_x and same_y are used for identity loss.
    same_x = generator_f(real_x, training=True)
    same_y = generator_g(real_y, training=True)

    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)

    # calculate the loss
    gen_g_loss = generator_loss(disc_fake_y)
    gen_f_loss = generator_loss(disc_fake_x)

    total_cycle_loss = calc_cycle_loss(real_x, cycled_x) + calc_cycle_loss(real_y, cycled_y)
```

```
    # Total generator loss = adversarial loss + cycle 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)

    disc_x_loss = discriminator_loss(disc_real_x, disc_fake_x)
    disc_y_loss = discriminator_loss(disc_real_y, disc_fake_y)

  # Calculate the 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)

  # Apply the gradients to the 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))
```

```
for epoch in range(EPOCHS):
  start = time.time()

  n = 0
  for image_x, image_y in tf.data.Dataset.zip((train_horses, train_zebras)):
    train_step(image_x, image_y)
    if n % 10 == 0:
      print ('.', end='')
    n += 1

  clear_output(wait=True)
  # Using a consistent image (sample_horse) so that the progress of the model
  # is clearly visible.
  generate_images(generator_g, sample_horse)

  if (epoch + 1) % 5 == 0:
    ckpt_save_path = ckpt_manager.save()
    print ('Saving checkpoint for epoch {} at {}'.format(epoch+1,
                                                         ckpt_save_path))

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

After training our algorithm, we now look at the results.

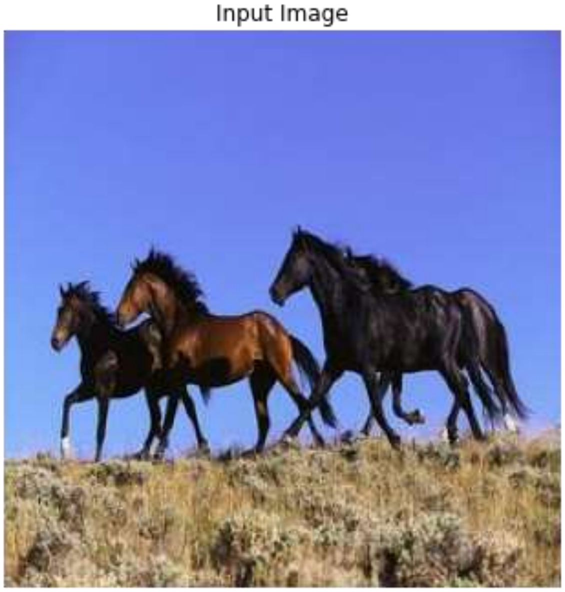
```
[ ] # Run the trained model on the test dataset
    for inp in test_horses.take(5):
      generate_images(generator_g, inp)
```

Results

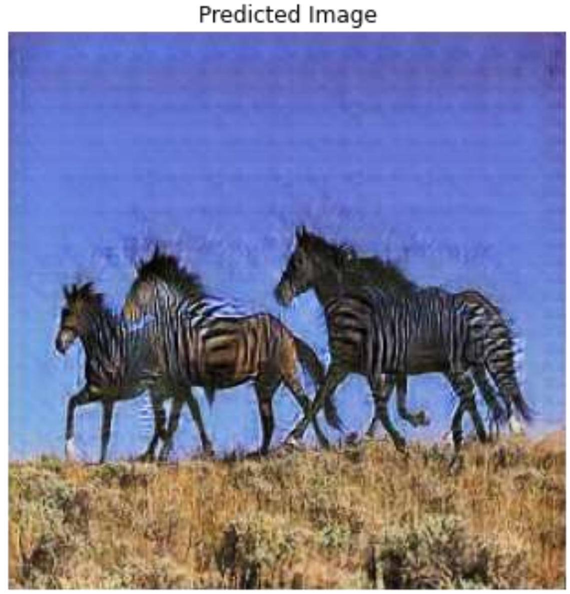
Input Image

Predicted Image

Input Image



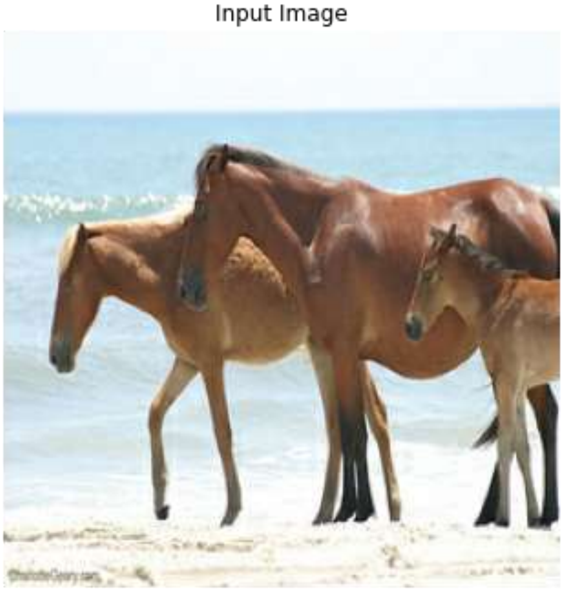
Predicted Image



Input Image

Predicted Image

Input Image



Predicted Image

Input Image

Predicted Image

Conclusion

As you can see on the results, it is incomplete. Take note that we used 20 epochs rather than 200. I've lessened the epoch due to google colab only being able to run 12 hours a day and it will take a power gpu and a longer run time to complete the 200 epochs training. Since we reduced the epoch there will be incomplete results. However, as we can see the generated images are able to mimic or transfer zebra stripes into the horse, I could say that the algorithm did well. It generated and transfer zebra strips accurately from the horse's body while reducing the loss of the horse itself or the environment. It is a little blurry because the transfer is not fully completed. In order to complete the images, it would need a longer training and runtime.

Reference(s):

- https://machinelearningmastery.com/impressive-applications-of-generative-adversarial-networks/
- https://www.tensorflow.org/
- https://www.tensorflow.org/tutorials/generative/cyclegan
- https://stackoverflow.com/questions/30558087/is-from-matplotlib-import-pyplot-as-plt-import-matplotlib-pyplot-as-plt
- https://www.tensorflow.org/tutorials/generative/pix2pix
- https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23
- https://towardsdatascience.com/optimizers-for-training-neural-network