Mars Deep Neural Network for Classifying Geological Features

Advanced Geocomputing 5543

> The synopsis of this project is to classify geological features on the surface of Mars to using training data from many subsets of Martian satellite imagery. This project was originally intended to classify deepfake satellite imagery to draw out comparisons between EuroSAT and Planet Data. Modeling off the Cheskape Bay Conservancy to find conservation efforts to minimize and reduce environmental degradation and maximize sustainability. Transitioning to using a generative adversarial network (GAN) and deep learning, and template cat pet picture script generator, the script was altered to generate EurotSat data and now Mars geological subset images.
>
> Alexander Danielson, (in collaboration with Jake Ford and Timothy Tran)

In [2]:

```
 #Inspiration: https://www.tensorflow.org/tutorials/generative/dcgan
# https://www.kaggle.com/code/joxcat/simple-cat-generator
# Project worked on by Alexander Danielson, Jake Ford, and Timothy Tran
# Import modules below
# This script uses TensorFlow models to create deepfake satellite imagery data (orginal

import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras import layers
import time
from IPython import display
import os
import PIL

# Parameters to define
# We can change how many epochs, examples to create, etc
img_size = 64
EPOCHS = 1
noise_dim = 100
num_examples_to_generate = 12
BUFFER_SIZE = 40000
BATCH_SIZE = 256

# The strategy below is setting up parallel code for use with TensorFlow
# It is useful the more GPUs we have access to
strategy = tf.distribute.MirroredStrategy()

# Will need to be changed if used on a different machine, like MSI's Mesabi.
mars_dataset = tf.keras.preprocessing.image_dataset_from_directory(
    "C:\Users\Alexander Danielson\Desktop\Fall 2022Spring2023\ArcGIS I\FinalProject\mar
    labels="inferred",
    label_mode="int",
    class_names=['rockfall'],
    color_mode="rgb", # output color
    batch_size=32,
```

```
    image_size=(img_size, img_size),
    shuffle=True,
    seed=None,
    validation_split=None,
    subset=None,
    interpolation="bilinear",
    follow_links=False,
)


# Creating arrays for our images and labels
# To later be formatted into a numpy array
mars_train_labels = []
mars_train_images = []

for images, labels in eu_dataset:
    for i in range(len(images)):
      eu_train_images.append(images[i])
      eu_train_labels.append(labels[i])

# Formatting our dataset into a numpy array
# It has a dimension of 3 because we are working with RGB bands
# The print statements were helpful to keep track of the shape of the data
eu_images = np.array(eu_train_images)
print(eu_images.shape)
eu_images = eu_images.reshape(eu_images.shape[0],img_size,img_size,3)
print(eu_images.shape)

eu_labels = np.array(eu_train_labels)
print(eu_labels.shape)
eu_labels = eu_labels.reshape(eu_labels.shape[0],)
print(eu_labels.shape)

# Uses save from numpy to keep track of the images and labels
from numpy import save
save('images.npy', eu_images)
save('labels.npy', eu_labels)


train_images = eu_images
train_labels = eu_labels

train_images = train_images.reshape(train_images.shape[0], img_size, img_size, 3).astyp
train_images = (train_images - 127.5) / 127.5 # Normalize the images to [-1, 1]

# Define the train dataset model here by slicing the train images and shuffling them
train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).b

# Generator model from TensorFlow GAN tutorial that we are using
# Parameters were changed, like the input size and filters, so the model
# will correctly take our input
# Added strategy.scope() as parallel code
with strategy.scope():
    def make_generator_model():

        model = tf.keras.Sequential()
        model.add(layers.Dense(8*8*256, use_bias=False, input_shape=(100,)))
        model.add(layers.BatchNormalization())
        model.add(layers.LeakyReLU())
```

```
        model.add(layers.Reshape((8, 8, 256)))
        assert model.output_shape == (None, 8, 8, 256) # Note: None is the batch size

        model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', u
        assert model.output_shape == (None, 8, 8, 128)
        model.add(layers.BatchNormalization())
        model.add(layers.LeakyReLU())

        model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', us
        assert model.output_shape == (None, 16, 16, 64)
        model.add(layers.BatchNormalization())
        model.add(layers.LeakyReLU())

        model.add(layers.Conv2DTranspose(3, (10, 10), strides=(4, 4), padding='same', u
        assert model.output_shape == (None, 64, 64, 3) #generator model

        return model

# Setting the model into a variable so it will be used throughout the code
generator = make_generator_model()

# This shows us what the model is creating everytime we run it
# Makes the noise random so it shows us what its creating
noise = tf.random.normal([1, 100])
generated_image = generator(noise, training=False)

# The code below just shows us what the generated image looks like, but we
# don't need it in our case.
#plt.imshow(generated_image[0, :, :, 0], cmap='gray')

# This is the discriminator model, derived from the inspiration listed
# above. Some parameters had to be changed in order to take our eurosat input
with strategy.scope():
    def make_discriminator_model():
        model = tf.keras.Sequential()
        model.add(layers.Conv2D(64, (10, 10), strides=(2, 2), padding='same',
                                         input_shape=[64, 64, 3]))
        model.add(layers.LeakyReLU())
        model.add(layers.Dropout(0.3))

        model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
        model.add(layers.LeakyReLU())
        model.add(layers.Dropout(0.3))

        model.add(layers.Flatten())
        model.add(layers.Dense(1))

        return model

# Creating the discriminator model
discriminator = make_discriminator_model()
decision = discriminator(generated_image)
print(decision)

cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
```

```
    return total_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)


generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

# We are able to save checkpoints so that can pick up right where we left off
# It saves every 15 epochs and can be called later on in case it gets interrupted
# Please make sure that the directory is the same as the dataset if run elsewhere
checkpoint_dir = '../cat_gan' # Should be the same directory as the dataset
checkpoint_prefix = os.path.join(checkpoint_dir, 'ckpt')
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                 discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)


# Creates the random seed to generate
seed = tf.random.normal([num_examples_to_generate, noise_dim])


@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
      generated_images = generator(noise, training=True)

      real_output = discriminator(images, training=True)
      fake_output = discriminator(generated_images, training=True)

      gen_loss = generator_loss(fake_output)
      disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_

    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminat


# The actual function that takes out dataset and epochs to train our model
def train(dataset, epochs):
    for epoch in range(epochs):
        start = time.time()

        for image_batch in dataset:
            train_step(image_batch)

    # This saves the image at each epoch for later reference
        display.clear_output(wait=True)
        generate_and_save_images(generator,
                                 epoch + 1,
                                 seed)

    # Save the model every 15 epochs
    if (epoch + 1) % 15 == 0:
```

```
        checkpoint.save(file_prefix = checkpoint_prefix)

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

# Generate after the final epoch
    display.clear_output(wait=True)
    generate_and_save_images(generator,
                             epochs,
                             seed)


# A function created to save the images that the model outputs into the root
# directory of where the program is stored.

def generate_and_save_images(model, epoch, test_input):
  # Notice `training` is set to False.
  # This is so all layers run in inference mode (batchnorm).
  predictions = model(test_input, training=False)

  # This says it's never used, but it actually helps format the final pictures
  # so that they don't look so far apart from each other
  fig = plt.figure(figsize=(4,4))

  for i in range(predictions.shape[0]):
      plt.subplot(4, 4, i+1)
      plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
      plt.axis('off')

  plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
  plt.show()

# A function below to display created images at certain epochs
def display_image(epoch):
    return PIL.Image.open('image_at_epoch_{:04d}.png'.format(epoch))

# A call to run the script with how many epochs we have set
train(train_dataset, EPOCHS)

# Restores a saved model, the checkpoint, with the following code
#checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))
```

```
  File "/tmp/ipykernel_91/526942852.py", line 33
    labels="inferred",
    ^
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in position 2-3: truncated \UXXXXXXXX escape
```

In [ ]: