CODE

import tensorflow as tf

import os

import pathlib

import time

import datetime

from matplotlib import pyplot as plt

from IPython import display

# Load dataset

dataset\_name = "facades"

\_URL = f'http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/{dataset\_name}.tar.gz'

path\_to\_zip = tf.keras.utils.get\_file(f"{dataset\_name}.tar.gz", origin=\_URL, extract=True)

path\_to\_zip = pathlib.Path(path\_to\_zip)

PATH = path\_to\_zip.parent/dataset\_name

# Helper function to load and process images

def load(image\_file):

image = tf.io.read\_file(image\_file)

image = tf.io.decode\_jpeg(image)

w = tf.shape(image)[1] // 2

input\_image = tf.cast(image[:, w:, :], tf.float32)

real\_image = tf.cast(image[:, :w, :], tf.float32)

return input\_image, real\_image

# Image preprocessing functions

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

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]

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

def random\_jitter(input\_image, real\_image):

input\_image, real\_image = resize(input\_image, real\_image, 286, 286)

input\_image, real\_image = random\_crop(input\_image, real\_image)

if tf.random.uniform(()) > 0.5:

input\_image = tf.image.flip\_left\_right(input\_image)

real\_image = tf.image.flip\_left\_right(real\_image)

return input\_image, real\_image

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

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

# Parameters

BUFFER\_SIZE = 400

BATCH\_SIZE = 1

IMG\_WIDTH = 256

IMG\_HEIGHT = 256

OUTPUT\_CHANNELS = 3

# Prepare datasets

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).batch(BATCH\_SIZE)

test\_dataset = tf.data.Dataset.list\_files(str(PATH / 'test/\*.jpg'))

test\_dataset = test\_dataset.map(load\_image\_test)

test\_dataset = test\_dataset.batch(BATCH\_SIZE)

# Downsample block

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

# Upsample block

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

# Generator

def Generator():

inputs = tf.keras.layers.Input(shape=[IMG\_HEIGHT, IMG\_WIDTH, 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

skips = []

for down in down\_stack:

x = down(x)

skips.append(x)

skips = reversed(skips[:-1])

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()

# Discriminator

def Discriminator():

initializer = tf.random\_normal\_initializer(0., 0.02)

inp = tf.keras.layers.Input(shape=[IMG\_HEIGHT, IMG\_WIDTH, 3], name='input\_image')

tar = tf.keras.layers.Input(shape=[IMG\_HEIGHT, IMG\_WIDTH, 3], name='target\_image')

x = tf.keras.layers.concatenate([inp, tar])

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)

discriminator = Discriminator()

# Loss functions

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)

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

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

# Optimizers

generator\_optimizer = tf.keras.optimizers.Adam(2e-4, beta\_1=0.5)

discriminator\_optimizer = tf.keras.optimizers.Adam(2e-4, beta\_1=0.5)

# Checkpoints

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)

# Training step function

@tf.function

def train\_step(input\_image, target):

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))

return gen\_total\_loss, disc\_loss

# Training loop

def fit(train\_ds, test\_ds, steps):

for step, (input\_image, target) in train\_ds.take(steps).enumerate():

if (step) % 10 == 0:

display.clear\_output(wait=True)

for example\_input, example\_target in test\_ds.take(1):

generate\_images(generator, example\_input, example\_target)

print(f"Step: {step // 1000}k")

gen\_total\_loss, disc\_loss = train\_step(input\_image, target)

if (step + 1) % 1000 == 0:

checkpoint.save(file\_prefix=checkpoint\_prefix)

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])

plt.imshow(display\_list[i] \* 0.5 + 0.5)

plt.axis('off')

plt.show()

# Start training

fit(train\_dataset, test\_dataset, steps=40000)

OUTPUT

