01_deep_convolutional_generative_adversarial_network

September 29, 2021

Deep Convolutional Generative Adversarial Network (DCGAN)

To illustrate the implementation of a GAN using Python, we use the Deep Convolutional GAN (DCGAN) example discussed in the Section 'Evolution of GAN Architectures' to synthesize images from the fashion MNIST dataset.

Adapted from the TensorFlow tutorial on generative modeling with the DCGAN architecture.

1 Imports & Settings

[1]:
```
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
from pathlib import Path
import imageio
import matplotlib.pyplot as plt
from tqdm import tqdm
from time import time

import tensorflow as tf
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.layers import (Dense, Reshape, Flatten, Dropout,
                                     BatchNormalization, LeakyReLU,
                                     Conv2D, Conv2DTranspose)
from tensorflow.keras.models import Sequential
from tensorflow.keras.losses import BinaryCrossentropy
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import plot_model

from IPython import display
```

[3]:
```
gpu_devices = tf.config.experimental.list_physical_devices('GPU')
if gpu_devices:
    print('Using GPU')
    tf.config.experimental.set_memory_growth(gpu_devices[0], True)
else:
    print('Using CPU')
```

```
Using CPU
```

[4]:
```
dcgan_path = Path('dcgan')
img_path = dcgan_path / 'synthetic_images'
if not img_path.exists():
    img_path.mkdir(parents=True)
```

2 Sample Image Saver

[5]:
```
def generate_and_save_images(model, epoch, test_input):
    # Training set to false so that every layer runs in inference mode
    predictions = model(test_input, training=False)

    fig = plt.figure(figsize=(5, 5.2))

    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')

    fig.suptitle(f'Epoch {epoch:03d}', fontsize=14)
    fig.tight_layout()
    fig.subplots_adjust(top=.93)
    fig.savefig(img_path / f'epoch_{epoch:03d}.png', dpi=300)
    plt.show()
```

3 Load and Prepare Data

3.1 Get training images

[6]:
```
# use only train images
(train_images, train_labels), (_, _) = fashion_mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-labels-idx1-ubyte.gz
32768/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-images-idx3-ubyte.gz
26427392/26421880 [==============================] - 1s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-labels-idx1-ubyte.gz
8192/5148 [===============================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [==============================] - 0s 0us/step
```

3.2 Extract image dimensions

[7]:
```
# get dimensionality
n_images = train_images.shape[0]
h = w = 28
```

3.3 Reshape to 4D input format

[8]:
```
train_images = (train_images
                .reshape(n_images, h, w, 1)
                .astype('float32'))
```

3.4 Normalize pixel values

[9]:
```
# Normalize the images in between -1 and 1
train_images = (train_images - 127.5) / 127.5
```

3.5 Create `tf.data.Dataset`

[10]:
```
BUFFER_SIZE = n_images
BATCH_SIZE = 256
```

[11]:
```
train_set = (tf.data.Dataset
             .from_tensor_slices(train_images)
             .shuffle(BUFFER_SIZE)
             .batch(BATCH_SIZE))
```

[12]:
```
train_set
```

[12]:
```
<BatchDataset shapes: (None, 28, 28, 1), types: tf.float32>
```

4 Create DCGAN Architecture

4.1 Build Generator

[13]:
```
def build_generator():
    return Sequential([Dense(7 * 7 * 256,
                             use_bias=False,
                             input_shape=(100,),
                             name='IN'),
                       BatchNormalization(name='BN1'),
                       LeakyReLU(name='RELU1'),
                       Reshape((7, 7, 256), name='SHAPE1'),
                       Conv2DTranspose(128, (5, 5),
                                       strides=(1, 1),
                                       padding='same',
```

```
                                    use_bias=False,
                                    name='CONV1'),
                        BatchNormalization(name='BN2'),
                        LeakyReLU(name='RELU2'),
                        Conv2DTranspose(64, (5, 5),
                                    strides=(2, 2),
                                    padding='same', use_bias=False,
                                    name='CONV2'),
                        BatchNormalization(name='BN3'),
                        LeakyReLU(name='RELU3'),
                        Conv2DTranspose(1, (5, 5),
                                    strides=(2, 2),
                                    padding='same',
                                    use_bias=False,
                                    activation='tanh',
                                    name='CONV3')],
                    name='Generator')
```

[14]:
```
generator = build_generator()
```

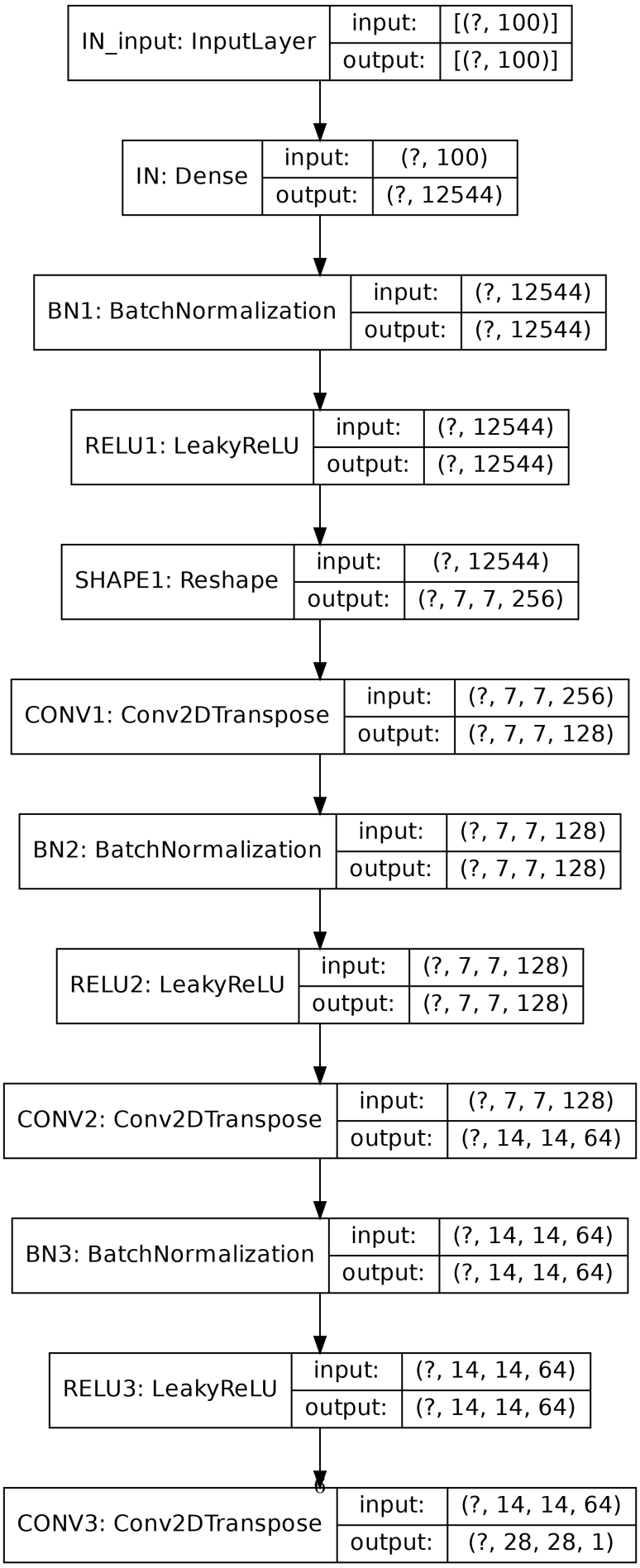
[15]:
```
generator.summary()
```

```
Model: "Generator"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
IN (Dense)                   (None, 12544)             1254400
_________________________________________________________________
BN1 (BatchNormalization)     (None, 12544)             50176
_________________________________________________________________
RELU1 (LeakyReLU)            (None, 12544)             0
_________________________________________________________________
SHAPE1 (Reshape)             (None, 7, 7, 256)         0
_________________________________________________________________
CONV1 (Conv2DTranspose)      (None, 7, 7, 128)         819200
_________________________________________________________________
BN2 (BatchNormalization)     (None, 7, 7, 128)         512
_________________________________________________________________
RELU2 (LeakyReLU)            (None, 7, 7, 128)         0
_________________________________________________________________
CONV2 (Conv2DTranspose)      (None, 14, 14, 64)        204800
_________________________________________________________________
BN3 (BatchNormalization)     (None, 14, 14, 64)        256
_________________________________________________________________
RELU3 (LeakyReLU)            (None, 14, 14, 64)        0
_________________________________________________________________
CONV3 (Conv2DTranspose)      (None, 28, 28, 1)         1600
```

```
=================================================================
Total params: 2,330,944
Trainable params: 2,305,472
Non-trainable params: 25,472
_________________________________________________________________
```

[16]:
```
plot_model(generator,
           show_shapes=True,
           dpi=300,
           to_file=(dcgan_path / 'generator.png').as_posix())
```

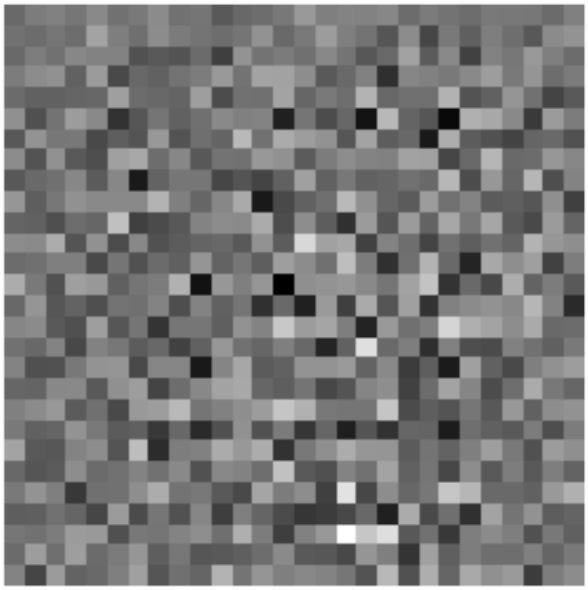
[16]:


IN_input: InputLayer
input: [(?, 100)]
output: [(?, 100)]
IN: Dense
input: (?, 100)
output: (?, 12544)
BN1: BatchNormalization
input: (?, 12544)
output: (?, 12544)
RELU1: LeakyReLU
input: (?, 12544)
output: (?, 12544)
SHAPE1: Reshape
input: (?, 12544)
output: (?, 7, 7, 256)
CONV1: Conv2DTranspose
input: (?, 7, 7, 256)
output: (?, 7, 7, 128)
BN2: BatchNormalization
input: (?, 7, 7, 128)
output: (?, 7, 7, 128)
RELU2: LeakyReLU
input: (?, 7, 7, 128)
output: (?, 7, 7, 128)
CONV2: Conv2DTranspose
input: (?, 7, 7, 128)
output: (?, 14, 14, 64)
BN3: BatchNormalization
input: (?, 14, 14, 64)
output: (?, 14, 14, 64)
RELU3: LeakyReLU
input: (?, 14, 14, 64)
output: (?, 14, 14, 64)
CONV3: Conv2DTranspose
input: (?, 14, 14, 64)
output: (?, 28, 28, 1)

4.1.1 Visualize initial image quality

[17]:
```
noise = tf.random.normal([1, 100])
generated_img = generator(noise, training=False)

plt.imshow(generated_img[0, :, :, 0], cmap='gray')
plt.axis('off');
```



4.2 Build Discriminator

Both the discriminator and generator use a deep CNN architecture, wrapped in a function:

[18]:
```
def build_discriminator():
    return Sequential([Conv2D(64, (5, 5),
                              strides=(2, 2),
                              padding='same',
                              input_shape=[28, 28, 1],
                              name='CONV1'),
                       LeakyReLU(name='RELU1'),
                       Dropout(0.3, name='DO1'),
                       Conv2D(128, (5, 5),
                              strides=(2, 2),
                              padding='same',
                              name='CONV2'),
```

```
                        LeakyReLU(name='RELU2'),
                        Dropout(0.3, name='DO2'),
                        Flatten(name='FLAT'),
                        Dense(1, name='OUT')],
                       name='Discriminator')
```

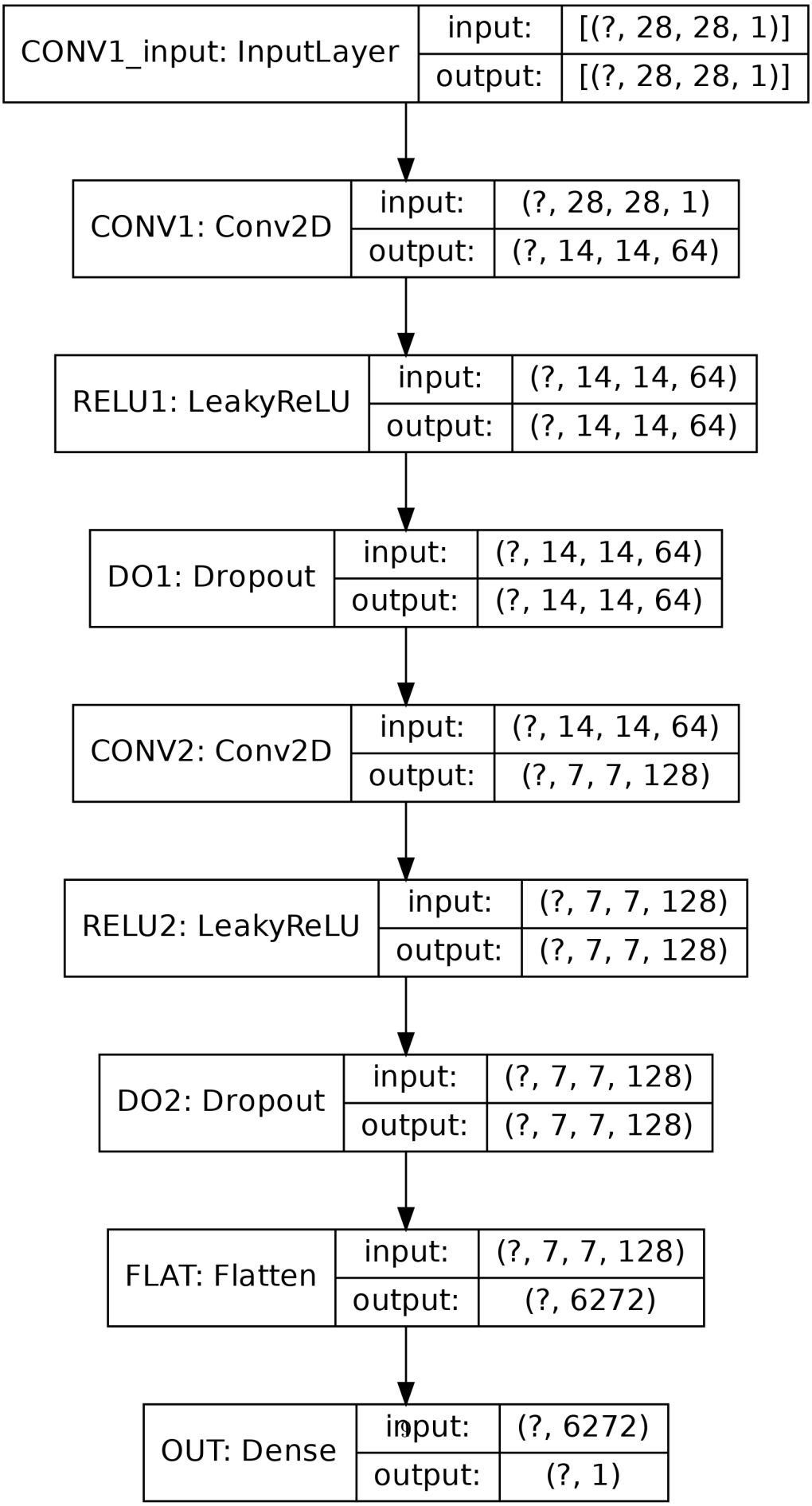
[19]:
```
discriminator = build_discriminator()
```

[20]:
```
discriminator.summary()
```

```
Model: "Discriminator"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
CONV1 (Conv2D)               (None, 14, 14, 64)        1664
_________________________________________________________________
RELU1 (LeakyReLU)            (None, 14, 14, 64)        0
_________________________________________________________________
DO1 (Dropout)                (None, 14, 14, 64)        0
_________________________________________________________________
CONV2 (Conv2D)               (None, 7, 7, 128)         204928
_________________________________________________________________
RELU2 (LeakyReLU)            (None, 7, 7, 128)         0
_________________________________________________________________
DO2 (Dropout)                (None, 7, 7, 128)         0
_________________________________________________________________
FLAT (Flatten)               (None, 6272)              0
_________________________________________________________________
OUT (Dense)                  (None, 1)                 6273
=================================================================
Total params: 212,865
Trainable params: 212,865
Non-trainable params: 0
_________________________________________________________________
```

[21]:
```
plot_model(discriminator,
           show_shapes=True,
           dpi=300,
           to_file=(dcgan_path / 'discriminator.png').as_posix())
```

[21]:


CONV1_input: InputLayer
input: [(?, 28, 28, 1)]
output: [(?, 28, 28, 1)]
CONV1: Conv2D
input: (?, 28, 28, 1)
output: (?, 14, 14, 64)
RELU1: LeakyReLU
input: (?, 14, 14, 64)
output: (?, 14, 14, 64)
DO1: Dropout
input: (?, 14, 14, 64)
output: (?, 14, 14, 64)
CONV2: Conv2D
input: (?, 14, 14, 64)
output: (?, 7, 7, 128)
RELU2: LeakyReLU
input: (?, 7, 7, 128)
output: (?, 7, 7, 128)
DO2: Dropout
input: (?, 7, 7, 128)
output: (?, 7, 7, 128)
FLAT: Flatten
input: (?, 7, 7, 128)
output: (?, 6272)
OUT: Dense
input: (?, 6272)
output: (?, 1)

4.2.1 Show discriminator decision output

[22]:
```
discriminator(generated_img).numpy()
```

[22]:
```
array([[-0.00111907]], dtype=float32)
```

5 Adversarial Training

[23]:
```
cross_entropy = BinaryCrossentropy(from_logits=True)
```

5.1 Generator Loss

[24]:
```
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

5.2 Discriminator Loss

[25]:
```
def discriminator_loss(true_output, fake_output):
    true_loss = cross_entropy(tf.ones_like(true_output), true_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    return true_loss + fake_loss
```

5.3 Optimizers

[26]:
```
gen_optimizer = Adam(1e-4)
dis_optimizer = Adam(1e-4)
```

5.4 Checkpoints

[27]:
```
checkpoints_dir = dcgan_path / 'training_chpk'
checkpoint_prefix = checkpoints_dir / 'ckpt'
checkpoint = tf.train.Checkpoint(gen_optimizer=gen_optimizer,
                                 dis_optimizer=dis_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)
```

5.5 Training Parameters

[28]:
```
EPOCHS = 100
noise_dim = 100
```

[29]:
```
# for gif generation
num_ex_to_gen = 16
seed = tf.random.normal([num_ex_to_gen, noise_dim])
```

The training begins with generator given a random seed as an input which is used to generate an image. The role of discriminator is to compare true image from training set with the fake image from generator. The loss is then calculated for both the models and weights are then updated through gradient descent. As we proceed further we will be having an efficient generator that is able to generate fake images close to the true images and an accurate discriminator that is able to accurately distinguish fake images from true images.

5.6 Training Step

[31]:
```
@tf.function
def train_step(images):
    # generate the random input for the generator
    noise = tf.random.normal([BATCH_SIZE, noise_dim])
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:

        # get the generator output
        generated_img = generator(noise, training=True)

        # collect discriminator decisions regarding real and fake input
        true_output = discriminator(images, training=True)
        fake_output = discriminator(generated_img, training=True)

        # compute the loss for each model
        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(true_output, fake_output)

    # compute the gradients for each loss with respect to the model variables
    grad_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    grad_discriminator = disc_tape.gradient(disc_loss, discriminator.
↪trainable_variables)

    # apply the gradient to complete the backpropagation step
    gen_optimizer.apply_gradients(zip(grad_generator, generator.
↪trainable_variables))
    dis_optimizer.apply_gradients(zip(grad_discriminator, discriminator.
↪trainable_variables))
```

5.7 Training Loop

[32]:
```
def train(dataset, epochs, save_every=10):
    for epoch in tqdm(range(epochs)):
        start = time()
```

```
        for img_batch in dataset:
            train_step(img_batch)

        # produce images for the GIF as we go
        display.clear_output(wait=True)
        generate_and_save_images(generator, epoch + 1, seed)

        # Save the model every 10 EPOCHS
        if (epoch + 1) % save_every == 0:
            checkpoint.save(file_prefix=checkpoint_prefix)

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

[33]:
```
train(train_set, EPOCHS)
```

5.8 Restore last checkpoint

[34]:
```
checkpoint.restore(tf.train.latest_checkpoint(checkpoints_dir));
```

6 Generate animated GIF

[35]:
```
out_file = dcgan_path / 'fashion_mnist.gif'
```

[36]:
```
with imageio.get_writer(out_file, mode='I') as writer:
    filenames = sorted(list(img_path.glob('*.png')))

    last = -1
    for i, filename in enumerate(filenames):
        frame = 2 * (i ** 0.5)
        if round(frame) > round(last):
            last = frame
        else:
            continue
        image = imageio.imread(filename)
        writer.append_data(image)
    image = imageio.imread(filename)
    writer.append_data(image)
```

[37]:
```
display.Image(filename=out_file)
```

[37]:
```
<IPython.core.display.Image object>
```