Generating Fashion Clothes Using GAN's

Importing required Libraries:

- Install the latest version of tensor flow

```
!pip3 install --upgrade tensorflow
```

```
Requirement already satisfied: tensorflow in /usr/local/lib/python3.10/dist-packages (2.12.0)
Collecting tensorflow
  Downloading tensorflow-2.13.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (524.1 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 524.1/524.1 MB 3.0 MB/s eta 0:00:00
Requirement already satisfied: absl-py>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (1.4.0)
Requirement already satisfied: astunparse>=1.6.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (1.6.3)
Requirement already satisfied: flatbuffers>=23.1.21 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (23.5.26)
Requirement already satisfied: gast<=0.4.0,>=0.2.1 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (0.4.0)
Requirement already satisfied: google-pasta>=0.1.1 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (0.2.0)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (1.57.0)
Requirement already satisfied: h5py>=2.9.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (3.9.0)
Collecting keras<2.14,>=2.13.1 (from tensorflow)
  Downloading keras-2.13.1-py3-none-any.whl (1.7 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.7/1.7 MB 55.3 MB/s eta 0:00:00
Requirement already satisfied: libclang>=13.0.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (16.0.6)
Requirement already satisfied: numpy<=1.24.3,>=1.22 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (1.23.5)
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (3.3.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from tensorflow) (23.1)
Requirement already satisfied: protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<5.0.0dev,>=3.20.3 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (3.20.3)
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-packages (from tensorflow) (67.7.2)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (1.16.0)
Collecting tensorboard<2.14,>=2.13 (from tensorflow)
  Downloading tensorboard-2.13.0-py3-none-any.whl (5.6 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5.6/5.6 MB 86.8 MB/s eta 0:00:00
Collecting tensorflow-estimator<2.14,>=2.13.0 (from tensorflow)
  Downloading tensorflow_estimator-2.13.0-py2.py3-none-any.whl (440 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 440.8/440.8 kB 40.9 MB/s eta 0:00:00
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (2.3.0)
Collecting typing-extensions<4.6.0,>=3.6.6 (from tensorflow)
  Downloading typing_extensions-4.5.0-py3-none-any.whl (27 kB)
Requirement already satisfied: wrapt>=1.11.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (1.14.1)
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (0.33.0)
```

- **numpy**:It is used to perform mathematical operations.
- **tensorflow**: It is used for machine learning and neural network related functions.
- **matplotlib** : It is used for plotting images.
- **Keras:** It is an open-source deep learning API that acts as an interface for TensorFlow and other popular deep learning frameworks

```
import keras

import numpy as np
import tensorflow as tf
print(f"tensorflow version: {tf.__version__}")

from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input,Conv2DTranspose,BatchNormalization,ReLU,Conv2D,LeakyReLU


from IPython import display

import matplotlib.pyplot as plt

%matplotlib inline

import os
from os import listdir
from pathlib import Path
import imghdr

import time
from tqdm.auto import tqdm
```

Mount Google Drive:

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
```

```
Mounted at /content/drive
```

Unzip the dataset folder:

```
!unzip -q '/content/drive/MyDrive/GAN /clothes_data.zip'
```

Creating data Generator:

- **Set the size of all images before going for training:**

```
img_height, img_width, batch_size=64,64,128
```

- **Create a training directory:**

Create a Keras **image_dataset_from_directory** object with a specified image directory (data) and store all these values in the training directory which is **train_ds.**

```
train_ds = tf.keras.utils.image_dataset_from_directory(directory='Data',
                                                       image_size=(img_height, img_width),
                                                       batch_size=batch_size,
                                                       label_mode=None)
```

```
Found 3000 files belonging to 1 classes.
```

Display 5 images from the batch:

- Take the first batch of images for displaying

```
images=train_ds.take(1)
```

- Convert the batch dimension to the indexes to the required indexes that is 64*64.

```
for i,x in enumerate(X[0:5]):
    x=x.numpy()
    max_=x.max()
    min_=x.min()
    xnew=np.uint(255*(x-min_)/(max_-min_))

    plt.axis("off")
    plt.subplot(1,5,i+1)
```

```
X=[x for x in images]
```

- Plot the first five images in the batch using the function **plot_array.**

```
plot_array(X[0])
```

Building a Generator:

The generator is designed to create images from a latent vector of 100 dimensions. The generator is defined as a Sequential model, which means the layers are stacked in sequence .The architecture consists of several convolutional transpose layers (also

known as deconvolutional layers) followed by batch normalization and rectified linear unit (ReLU) activation functions..

- **Input Layer**: The generator takes a latent vector of 100 dimensions as input.
- **Block 1**: A convolutional transpose layer upsamples the input to a larger spatial resolution. It is followed by batch normalization and ReLU activation.
- **Block 2:** Another convolutional transpose layer further upsamples the feature maps. Again, batch normalization and ReLU activation are applied.
- **Block 3:** Similar to Block 2, this block performs additional upsampling using a convolutional transpose layer, followed by batch normalization and ReLU activation.
- **Block 4:** This block continues the upsampling process with another convolutional transpose layer, followed by batch normalization and ReLU activation.
- The last convolutional transpose layer aims to produce the final image. It outputs 3 channels (for RGB images) .

```
def make_generator():

    model=Sequential()

    # input is latent vector of 100 dimensions
    model.add(Input(shape=(1, 1, 100), name='input_layer'))

    # Block 1 dimensionality of the output space  64 * 8
    model.add(Conv2DTranspose(64 * 8, kernel_size=4, strides= 4, padding='same', kernel_initializer=tf.keras.initializers.RandomNormal(mean=0.0, stddev=0.02), use_bias=Fa
    model.add(BatchNormalization(momentum=0.1,  epsilon=0.8, center=1.0, scale=0.02, name='bn_1'))
    model.add(ReLU(name='relu_1'))

    # Block 2: input is 4 x 4 x (64 * 8)
    model.add(Conv2DTranspose(64 * 4, kernel_size=4, strides= 2, padding='same', kernel_initializer=tf.keras.initializers.RandomNormal(mean=0.0, stddev=0.02), use_bias=Fa
    model.add(BatchNormalization(momentum=0.1,  epsilon=0.8, center=1.0, scale=0.02, name='bn_2'))
    model.add(ReLU(name='relu_2'))

    # Block 3: input is 8 x 8 x (64 * 4)
    model.add(Conv2DTranspose(64 * 2, kernel_size=4,strides=  2, padding='same', kernel_initializer=tf.keras.initializers.RandomNormal(mean=0.0, stddev=0.02), use_bias=Fa
    model.add(BatchNormalization(momentum=0.1,  epsilon=0.8,  center=1.0, scale=0.02, name='bn_3'))
    model.add(ReLU(name='relu_3'))
```

```
# Block 4: input is 16 x 16 x (64 * 2)
model.add(Conv2DTranspose(64 * 1, kernel_size=4, strides= 2, padding='same', kernel_initializer=tf.keras.initializers.RandomNormal(mean=0.0, stddev=0.02),
model.add(BatchNormalization(momentum=0.1,  epsilon=0.8,  center=1.0, scale=0.02, name='bn_4'))
model.add(ReLU(name='relu_4'))

model.add(Conv2DTranspose(3, 4, 2,padding='same', kernel_initializer=tf.keras.initializers.RandomNormal(mean=0.0, stddev=0.02), use_bias=False,
                          activation='tanh', name='conv_transpose_5'))

return model
```

```
gen = make_generator()
gen.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv_transpose_1 (Conv2DTr  (None, 4, 4, 1024)        1638400
 anspose)

 bn_1 (BatchNormalization)   (None, 4, 4, 1024)        4096

 relu_1 (ReLU)               (None, 4, 4, 1024)        0

 conv_transpose_2 (Conv2DTr  (None, 8, 8, 256)         4194304
 anspose)

 bn_2 (BatchNormalization)   (None, 8, 8, 256)         1024

 relu_2 (ReLU)               (None, 8, 8, 256)         0

 conv_transpose_3 (Conv2DTr  (None, 16, 16, 256)       1048576
 anspose)

 bn_3 (BatchNormalization)   (None, 16, 16, 256)       1024

 relu_3 (ReLU)               (None, 16, 16, 256)       0

 conv_transpose_4 (Conv2DTr  (None, 32, 32, 128)       524288
 anspose)

 bn_4 (BatchNormalization)   (None, 32, 32, 128)       512

 relu_4 (ReLU)               (None, 32, 32, 128)       0

 conv_transpose_5 (Conv2DTr  (None, 64, 64, 3)         6144
 anspose)

_________________________________________________________________
=================================================================
Total params: 7418368 (28.30 MB)
Trainable params: 7415040 (28.29 MB)
Non-trainable params: 3328 (13.00 KB)
_________________________________________________________________
```

Building a Discriminator:

The Discriminator has five convolution layers.

- The first and final Conv2D layers have Batch Normalization, since directly applying batchnorm to all layers could result in sample oscillation and model instability;
- The first four Conv2D layers use the Leaky-Relu activation with a slope of 0.2.

- Lastly, instead of a fully connected layer, the output layer has a convolution layer with a Sigmoid activation function.

```
def make_discriminator():

    model=Sequential()

    # Block 1: input is 64 x 64 x (3)
    model.add(Input(shape=(64, 64, 3), name='input_layer'))
    model.add(Conv2D(64, kernel_size=4, strides= 2, padding='same', kernel_initializer=tf.keras.initializers.RandomNormal(mean=0.0, stddev=0.02), use_b
    model.add(LeakyReLU(0.2, name='leaky_relu_1'))

    # Block 2: input is 32 x 32 x (64)
    model.add(Conv2D(64 * 2, kernel_size=4, strides= 2, padding='same', kernel_initializer=tf.keras.initializers.RandomNormal(mean=0.0, stddev=0.02), u
    model.add(BatchNormalization(momentum=0.1,  epsilon=0.8, center=1.0, scale=0.02, name='bn_1'))
    model.add(LeakyReLU(0.2, name='leaky_relu_2'))

    # Block 3
    model.add(Conv2D(64 * 4, 4, 2, padding='same', kernel_initializer=tf.keras.initializers.RandomNormal(mean=0.0, stddev=0.02), use_bias=False, name='
    model.add(BatchNormalization(momentum=0.1,  epsilon=0.8, center=1.0, scale=0.02, name='bn_2'))
    model.add(LeakyReLU(0.2, name='leaky_relu_3'))


    #Block 4
    model.add(Conv2D(64 * 8, 4, 2, padding='same', kernel_initializer=tf.keras.initializers.RandomNormal(mean=0.0, stddev=0.02), use_bias=False, name='
    model.add(BatchNormalization(momentum=0.1,  epsilon=0.8, center=1.0, scale=0.02, name='bn_3'))
    model.add(LeakyReLU(0.2, name='leaky_relu_4'))


    #Block 5
    model.add(Conv2D(1, 4, 2,padding='same', kernel_initializer=tf.keras.initializers.RandomNormal(mean=0.0, stddev=0.02), use_bias=False,
                     activation='sigmoid', name='conv_5'))
```

```
return model
```

```
disc = make_discriminator()
disc.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv_1 (Conv2D)             (None, 32, 32, 64)        3072

 leaky_relu_1 (LeakyReLU)    (None, 32, 32, 64)        0

 conv_2 (Conv2D)             (None, 16, 16, 128)       131072

 bn_1 (BatchNormalization)   (None, 16, 16, 128)       512

 leaky_relu_2 (LeakyReLU)    (None, 16, 16, 128)       0

 conv_3 (Conv2D)             (None, 8, 8, 256)         524288

 bn_2 (BatchNormalization)   (None, 8, 8, 256)         1024

 leaky_relu_3 (LeakyReLU)    (None, 8, 8, 256)         0

 conv_4 (Conv2D)             (None, 4, 4, 512)         2097152

 bn_3 (BatchNormalization)   (None, 4, 4, 512)         2048

 leaky_relu_4 (LeakyReLU)    (None, 4, 4, 512)         0

 conv_5 (Conv2D)             (None, 2, 2, 1)           8192

=================================================================
Total params: 2767360 (10.56 MB)
Trainable params: 2765568 (10.55 MB)
Non-trainable params: 1792 (7.00 KB)
_________________________________________________________________
```

Define Loss function in generator and discriminator:

The loss functions is used for both the generator and the discriminator. The generator loss encourages the generator to produce more realistic samples, while the discriminator loss guides the discriminator to accurately distinguish between real and generated samples.

- **cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True):** This line initializes a binary cross-entropy loss function using TensorFlow's Keras library.

- **def generator_loss(Xhat):** The generator_loss takes a single argument Xhat (fake images), which represents the generated data samples from the generator.

- **return cross_entropy(tf.ones_like(Xhat), Xhat):** In the generator_loss function, this line calculates the loss for the generator. It compares the generated samples (Xhat) to a tensor of ones (indicating real data labels) using the previously defined binary cross-entropy loss. The goal is to encourage the generator to produce samples that are more similar to real data, hence minimizing this loss.

- **def discriminator_loss(X, Xhat):** The discriminator_loss function takes two arguments: X, representing the real data samples, and Xhat, representing the generated data samples.

- **real_loss = cross_entropy(tf.ones_like(X), X**): In the discriminator_loss function, this line calculates the loss for the discriminator based on real data samples.

- **fake_loss = cross_entropy(tf.zeros_like(Xhat), Xhat):** This line calculates the loss for the discriminator based on the generated data samples..

- **total_loss = 0.5*(real_loss + fake_loss):** This line computes the total loss for the discriminator. The total loss is the average of the losses calculated for both the real and fake data samples. The factor of 0.5 ensures that both components contribute equally to the total loss.

- **return total_loss:** The discriminator_loss function returns the computed total loss for the discriminator.

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

```
def generator_loss(Xhat):
    return cross_entropy(tf.ones_like(Xhat), Xhat)
```

```
def discriminator_loss(X, Xhat):
    real_loss = cross_entropy(tf.ones_like(X), X)
    fake_loss = cross_entropy(tf.zeros_like(Xhat), Xhat)
    total_loss = 0.5*(real_loss + fake_loss)
    return total_loss
```

Define optimizer in generator and discriminator:

- Create two Adam optimizers, one for the generator and another for the discriminator, both with the same learning rate of 0.0002.
- The beta parameters beta coefficients β_1=**0.5 and** β_2=**0.999,** which are responsible for computing the running averages of the gradients during backpropagation.

```
learning_rate = 0.0002

generator_optimizer = tf.keras.optimizers.Adam(lr = 0.0002, beta_1 = 0.5, beta_2 = 0.999 )

discriminator_optimizer = tf.keras.optimizers.Adam(lr = 0.0002, beta_1 = 0.5, beta_2 = 0.999 )
```

Create Trained Step Function:

- First, for z, a batch of noise vectors from a normal distribution and feed it to the Generator.
- The Generator produces generated or "fake" images and store it in **xhat.**
- Feed real images **x** and fake images **xhat** to the Discriminator to obtain **real_output** and **fake_output** respectively as the scores.
- We calculate Generator loss **gen_loss** using the **fake_output** from Discriminator since we want the fake images to fool the Discriminator as much as possible.
- We calculate Discriminator loss **disc_loss** using both the real output and **fake_output** since we want the Discriminator to distinguish the two as much as possible.
- We calculate **gradient_of_generator** and **gradient_of_discriminator** based on the losses obtained.

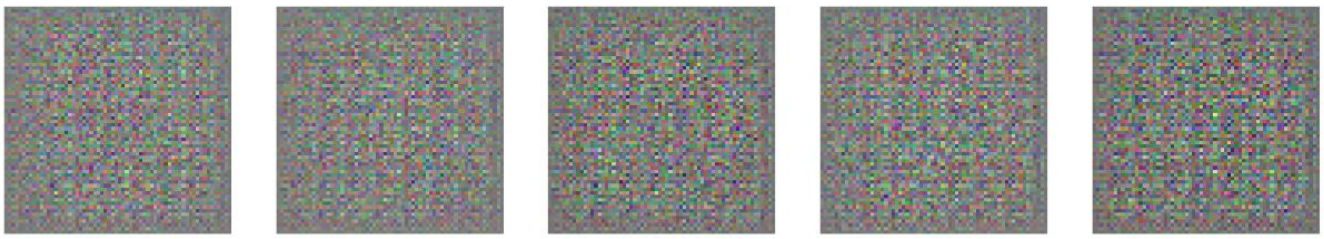
- Finally, we update the Generator and Discriminator by letting their respective optimizers apply the processed gradients on the trainable model parameters.

```
@tf.function

def train_step(X):

    #random samples it was found if you increase the  stander deviation, you get better results
    z= tf.random.normal([BATCH_SIZE, 1, 1, latent_dim])
      # needed to compute the gradients for a list of variables.
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        #generated sample
        xhat = generator(z, training=True)
        #the output of the discriminator for real data
        real_output = discriminator(X, training=True)
        #the output of the discriminator for fake data
        fake_output = discriminator(xhat, training=True)

        #loss for each
        gen_loss= generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)
      # Compute the gradients for gen_loss and generator

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    # Compute the gradients for gen_loss and discriminator
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)
    # Ask the optimizer to apply the processed gradients
    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))
```

As the generator is not trained yet, the output generated images have noise.

```
generator= make_generator()
BATCH_SIZE=128

latent_dim=100
noise = tf.random.normal([BATCH_SIZE, 1, 1, latent_dim])
Xhat=generator(noise,training=False)
plot_array(Xhat)
```

Noisy images



Trained the model to remove noise from the generated images:

- Trained the model for one epoch and then use the generator to produce artificial images.

```
epochs=1

discriminator=make_discriminator()

generator= make_generator()


for epoch in range(epochs):

    #data for the true distribution of your real data samples training ste
    start = time.time()
    i=0
    for X in tqdm(normalized_ds, desc=f"epoch {epoch+1}", total=len(normalized_ds)):

        i+=1
        if i%1000:
            print("epoch {}, iteration {}".format(epoch+1, i))

        train_step(X)


    noise = tf.random.normal([BATCH_SIZE, 1, 1, latent_dim])
    Xhat=generator(noise,training=False)
    X=[x for x in normalized_ds]
    print("orignal images")
    plot_array(X[0])
    print("generated images")
    plot_array(Xhat)
    print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))
```

```
epoch 1: 100% ████████████████████ 24/24 [05:21<00:00, 10.78s/it]
epoch 1, iteration 1
/usr/local/lib/python3.10/dist-packages/keras/src/backend.py:5805: UserWarning: "`binary_crossentropy
  output, from_logits = _get_logits(
epoch 1, iteration 2
epoch 1, iteration 3
epoch 1, iteration 4
epoch 1, iteration 5
epoch 1, iteration 6
epoch 1, iteration 7
epoch 1, iteration 8
epoch 1, iteration 9
epoch 1, iteration 10
epoch 1, iteration 11
epoch 1, iteration 12
epoch 1, iteration 13
epoch 1, iteration 14
epoch 1, iteration 15
epoch 1, iteration 16
epoch 1, iteration 17
epoch 1, iteration 18
epoch 1, iteration 19
epoch 1, iteration 20
epoch 1, iteration 21
epoch 1, iteration 22
epoch 1, iteration 23
epoch 1, iteration 24
```

Results: