Generative Adversarial Network (GAN)

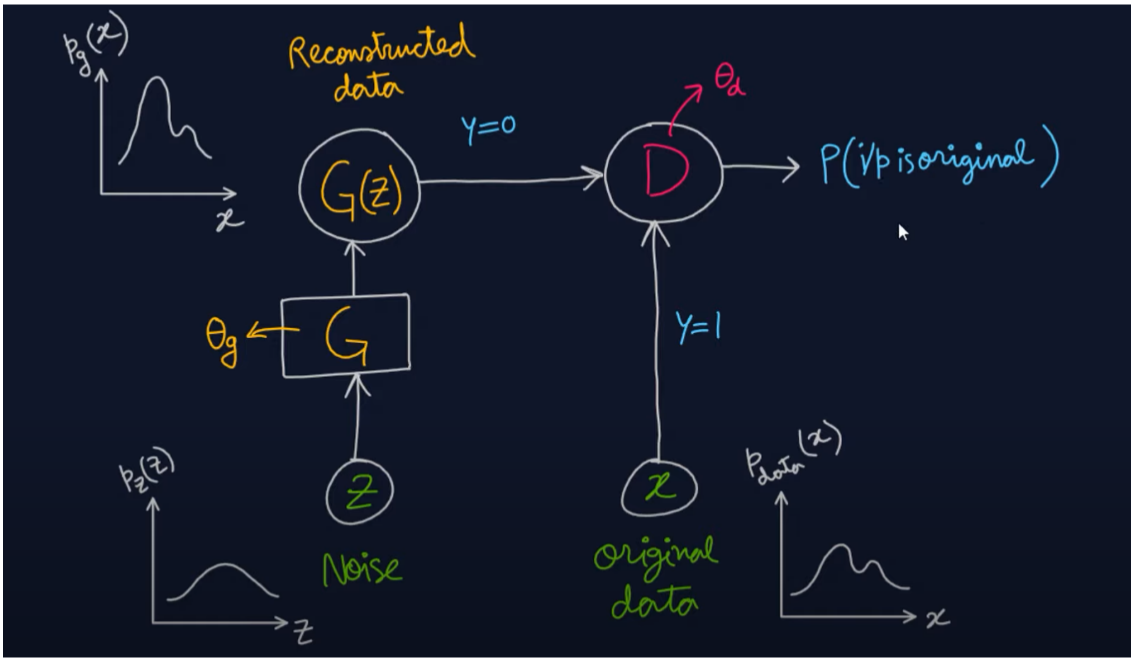
GANs are *generative* models that can create new data instances that resemble the training data. GANs can create images that look like photographs of human faces, even though the faces don't belong to any real person.

It comprises of a generator and a discriminator. The generator tries to fool the discriminator, and the discriminator tries to keep from being fooled.

Generator and Discriminator model resembles a two player min-max game - where each player tries to maximise their chance of winning.

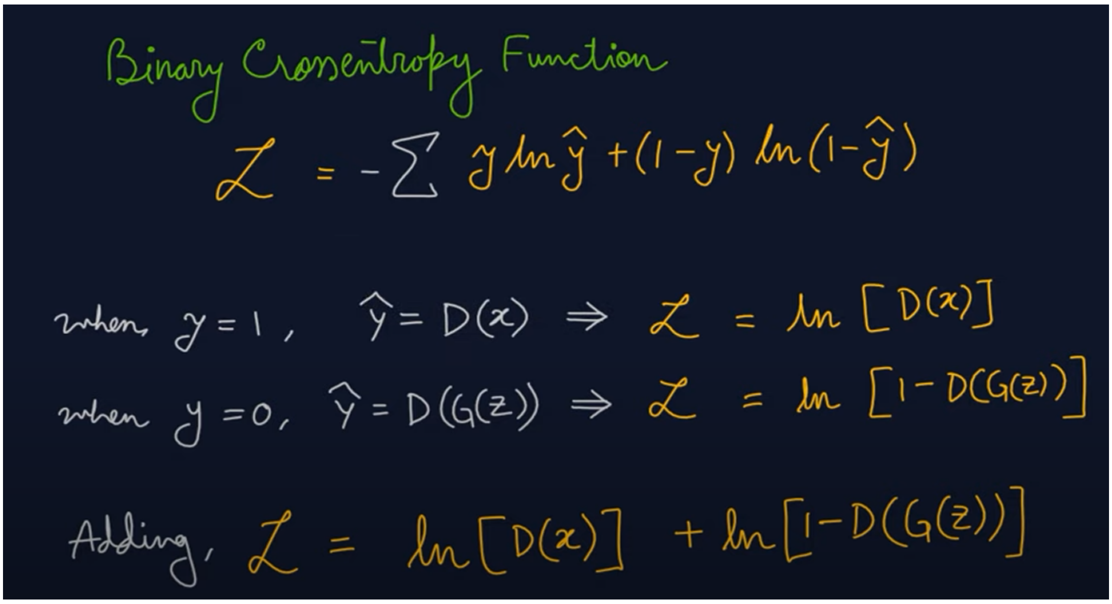
In this project i have implemented a basic model of GAN which can generate dataset similar to that of fashion_mnist dataset

GAN overview

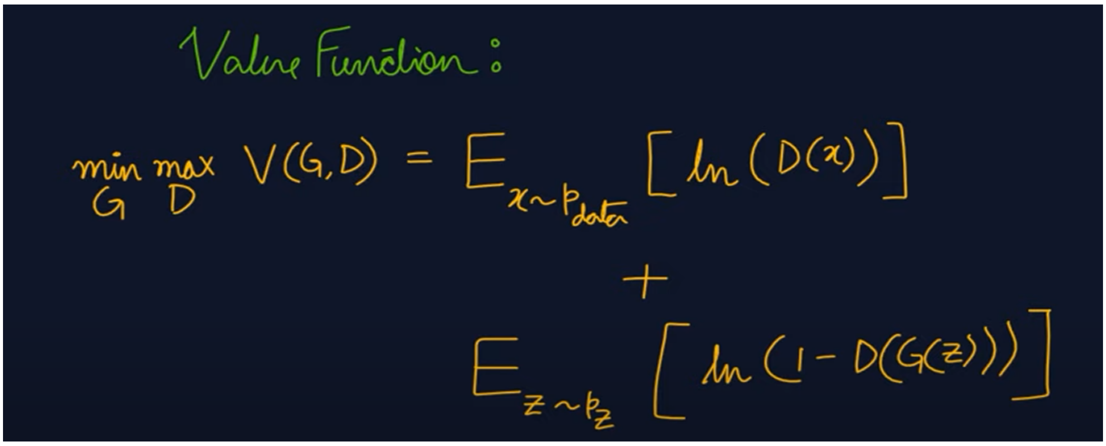


Generator- takes in random values of distribution $P_z(z)$ and converts them to image $P_{gen}(x)$---- tries to analyse the probability distribution function of the original data $P_{data}(x)$.

Discriminator- classifies the generated output $P_{gen}(x)$ as real or fake by comparing it wrt original data $P_{data}(x)$



We prove that using Jensen Shannon- JS Divergence.... To solve the loss fn equation to calculate the minima.



Instead of minimising the likelihood of the discriminator being correct, we try to maximise the likelihood of the discriminator being wrong

FashionGAN

1. Import all necessary libraries and load fashion_mnist dataset

```
] !pip install tensorflow tensorflow-gpu matplotlib tensorflow-datasets ipywidgets
```

```
!pip list
```

```
] import tensorflow as tf
  # If GPUs are enabled-- to limit memory growth
  gpus = tf.config.experimental.list_physical_devices('GPU')
  for gpu in gpus:
      tf.config.experimental.set_memory_growth(gpu, True)
```

```
] # Brining in tensorflow datasets for fashion mnist
  import tensorflow_datasets as tfds
  from matplotlib import pyplot as plt
```

```
ds = tfds.load('fashion_mnist', split='train')

Downloading and preparing dataset 29.45 MiB (download: 29.45 MiB, generated: 36.42 MiB, total: 65.87 MiB) to /root/
Dl Completed...: 100% 4/4 [00:00<00:00, 11.29 url/s]
Dl Size...: 100% 29/29 [00:00<00:00, 53.32 MiB/s]
Extraction completed...: 100% 4/4 [00:00<00:00, 3.75 file/s]
Dataset fashion_mnist downloaded and prepared to /root/tensorflow_datasets/fashion_mnist/3.0.1. Subsequent calls wi
```

```
] ds.as_numpy_iterator().next()['label']

2
```

2. To visualise the dataset– assign an iterator to traverse the dataset– apply some transformations like
   - Squeeze it – 28*28*1 into 28*28
   - Scaling it within [0,1]
   - If we are using conditional GAN model – we include labels
   - Apply data augmentation if needed

```
] import numpy as np
  # Iterator to loop over through each data points
  dataiterator = ds.as_numpy_iterator()
```

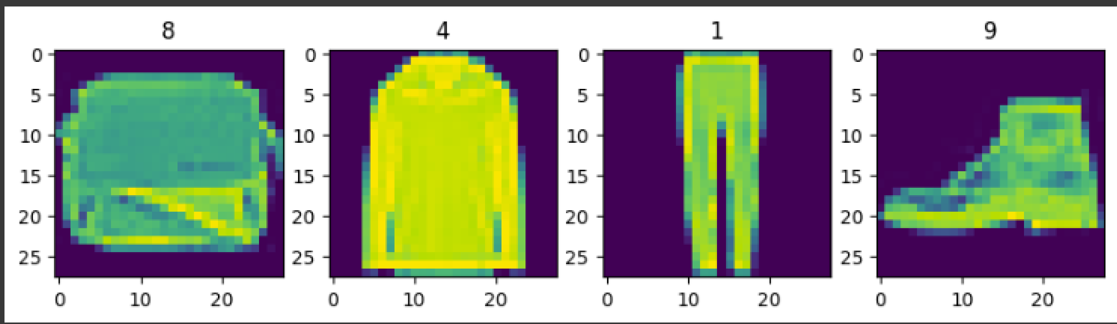
```
# Getting random sample data out of the pipeline- to visualize it
dataiterator.next()['image']

array([[[  0],
        [  0],
        [  0],
        [  0],
        [  0],
        [  0],
        [  0],
        [  0],
        [  0],
        [ 18],
        [ 77],
        [227],
```

```
np.squeeze(dataiterator.next()['image']).shape # Converts 28*28*1 into 28*28
```

```
(28, 28)
```

```
# Setup the subplot formatting
fig, ax = plt.subplots(ncols=4, figsize=(10,10))
# Loop four times and get images
for idx in range(4):
    sample = dataiterator.next()
    # Plot the image using a specific subplot
    ax[idx].imshow(np.squeeze(sample['image']))
    # Appending the image label as the plot title
    ax[idx].title.set_text(sample['label'])
```




```
# Scale and return images -- for faster training of the model
def scale_images(data):
    image = data['image']
    return image / 255
# If we are using any cGAN, then we may need the labels also

# We could apply data augmentation also --if needed for other models
```

3. Create data pipeline for the model using these operations
   - Map
   - Cache
   - Shuffle
   - Batch - 128
   - Prefetch – to optimise the flow of data between storage and processing units, resulting in improved performance, reduced latency, and better resource utilisation during model training or inference.

```
''' Creating a data pipeline--- Common for any tensorflow model
    1.map 2.cache 3.shuffle 4.batch 5.prefetch '''

# Running the dataset through the scale_images preprocessing step
ds = ds.map(scale_images)
# Cache the dataset for that batch
ds = ds.cache()
# Shuffle it up
ds = ds.shuffle(60000)
# Batch into 128 images per sample
ds = ds.batch(128)
# Reduces the likelihood of bottlenecking
ds = ds.prefetch(64)
```

```
'prefetch() --While the model is busy processing one batch of data, the next batch is being fetched and prepared in the background'
```

```
ds.as_numpy_iterator().next().shape
```

```
(128, 28, 28, 1)
```

4. Build Generator and Discriminator model

Dense layer / fully connected layer - matrix representing that each neuron in a layer is related to all neurons in the prev layer by some weight. The number of neurons in a fully connected layer determines the dimensionality of the output which is the product of the number of neurons in the current layer and the number of neurons in the previous layer, including the bias terms.

Activation functions such as Softmax, ReLU or leakyReLU introduce non-linearities, enabling the network to learn complex relationships between inputs and outputs.

Dropout - used for regularisation

UpSampling2D - Upsamples the generated image by adding extra space to image

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dense, Flatten, Reshape, LeakyReLU, Dropout, UpSampling2D
```

5. Generator

- generator_Model()   // adding a layer at a time
- test_model = generator_Model()
- test_model.summary

By this way can check what happens at each step when we add a new layer to our model

```
def build_generator():
    model = Sequential()

    # Takes in random values and reshape it to 7x7x128
    # Beginning of a generated image
    model.add(Dense(7*7*128, input_dim=128))  # input_dim -- controls generation of image ; 7*7 -- gives shape
    model.add(LeakyReLU(0.2))
    model.add(Reshape((7,7,128)))   # Finally we have to convert 7*7*128 into 28*28*1

    # Upsampling block 1
    model.add(UpSampling2D())   #14*14*128
    model.add(Conv2D(128, 5, padding='same'))
    model.add(LeakyReLU(0.2))

    # Upsampling block 2
    model.add(UpSampling2D())   #28*28*128
    model.add(Conv2D(128, 5, padding='same'))
    model.add(LeakyReLU(0.2))

    # Convolutional block 1
    model.add(Conv2D(128, 4, padding='same'))
    model.add(LeakyReLU(0.2))

    # Convolutional block 2
    model.add(Conv2D(128, 4, padding='same'))
    model.add(LeakyReLU(0.2))

    # Conv layer to get to one channel
    model.add(Conv2D(1, 4, padding='same', activation='sigmoid'))

    return model
```

```
generator = build_generator()
```

model.add(Dense(7*7*128, input_dim=128)) -> input_dim -- controls generation of image ;
7*7 -- gives shape

Conv2D and UpSampling2D can be used to reduce this random values into 28*28*1
model.add(UpSampling2D()) -> 7*7 converted to 14*14
Similarly for subsequent layers

model.add(Conv2D(128, 5, padding='same')) -> 128- for the output dim ; 5*5 kernel is used
Like this we convert the img to 28*28 size
We add few bunch of layers like this, to get a sophisticated model.

In the final layer, we use
model.add(Conv2D(1, 4, padding='same', activation='sigmoid')) -> to get output dim-1 ;
4*4 kernel sigmoid is used

```
generator.summary()

Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 6272)              809088

 leaky_re_lu (LeakyReLU)     (None, 6272)              0

 reshape (Reshape)           (None, 7, 7, 128)         0

 up_sampling2d (UpSampling2D  (None, 14, 14, 128)      0
 )

 conv2d (Conv2D)             (None, 14, 14, 128)       409728

 leaky_re_lu_1 (LeakyReLU)   (None, 14, 14, 128)       0

 up_sampling2d_1 (UpSampling  (None, 28, 28, 128)      0
 2D)

 conv2d_1 (Conv2D)           (None, 28, 28, 128)       409728

 leaky_re_lu_2 (LeakyReLU)   (None, 28, 28, 128)       0

 conv2d_2 (Conv2D)           (None, 28, 28, 128)       262272

 leaky_re_lu_3 (LeakyReLU)   (None, 28, 28, 128)       0

 conv2d_3 (Conv2D)           (None, 28, 28, 128)       262272

 leaky_re_lu_4 (LeakyReLU)   (None, 28, 28, 128)       0

 conv2d_4 (Conv2D)           (None, 28, 28, 1)         2049

=================================================================
Total params: 2,155,137
Trainable params: 2,155,137
Non-trainable params: 0
_________________________________________________________________
```

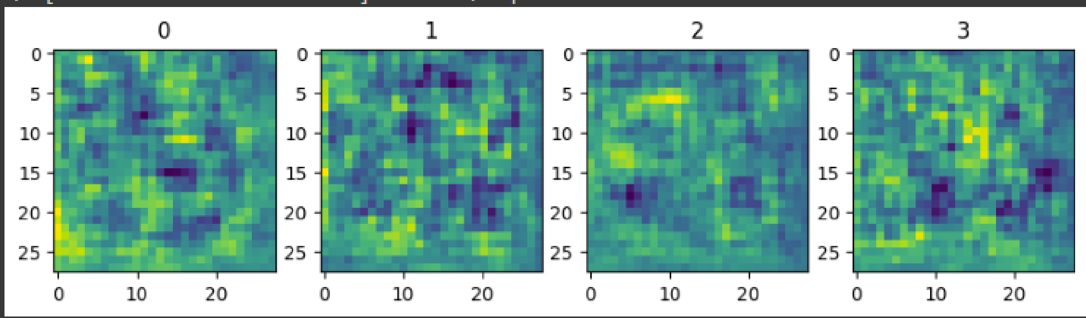
Visualise it — Using enumerate- to be able to get value and index of itr in loop

```
img = generator.predict(np.random.randn(4,128,1))    # 4--images ; 128-- random values
img.shape
```

```
1/1 [==============================] - 8s 8s/step
(4, 28, 28, 1)
```

```
# Generate new fashion
img = generator.predict(np.random.randn(4,128,1))
fig, ax = plt.subplots(ncols=4, figsize=(10,10))
# Loop 4 times and get images
for idx, img in enumerate(img):
    ax[idx].imshow(np.squeeze(img))
    # Appending the image label as the plot title
    ax[idx].title.set_text(idx)
```

```
1/1 [==============================] - 0s 25ms/step
```



6. Discriminator

- discriminator_Model() # adding a layer at a time
- test_model = discriminator_Model()
- test_model.summary

By this way can check what happens at each step when we add a new layer to our model

```
def build_discriminator():
    model = Sequential()

    # First Conv Block
    model.add(Conv2D(32, 5, input_shape = (28,28,1)))
    model.add(LeakyReLU(0.2))
    model.add(Dropout(0.4))

    # Second Conv Block
    model.add(Conv2D(64, 5))    # need not mention input_shape except for the first layer
    model.add(LeakyReLU(0.2))
    model.add(Dropout(0.4))

    # Third Conv Block
    model.add(Conv2D(128, 5))
    model.add(LeakyReLU(0.2))
    model.add(Dropout(0.4))

    # Fourth Conv Block
    model.add(Conv2D(256, 5))
    model.add(LeakyReLU(0.2))
    model.add(Dropout(0.4))

    # Flatten then pass to dense layer
    model.add(Flatten())
    model.add(Dropout(0.4))
    model.add(Dense(1, activation='sigmoid'))    # Fake -1; Real -0

    return model
```

```
discriminator = build_discriminator()
```

model.add(Conv2D(32, 5, input_shape = (28,28,1))) -> using 32 filters of 5*5 ; since padding is not same – the output decreases in size ;input_shape = (28,28,1) – same as output of generator
For further layers,
model.add(Conv2D(64, 5)) -> we dropped input_shape ; increased filters used
model.add(Dropout(0.4))-> for regularisation

Finally we Flatten it.
model.add(Flatten())

```
discriminator.summary()

Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_14 (Conv2D)          (None, 24, 24, 32)        832

 leaky_re_lu_14 (LeakyReLU)  (None, 24, 24, 32)        0

 dropout_5 (Dropout)         (None, 24, 24, 32)        0

 conv2d_15 (Conv2D)          (None, 20, 20, 64)        51264

 leaky_re_lu_15 (LeakyReLU)  (None, 20, 20, 64)        0

 dropout_6 (Dropout)         (None, 20, 20, 64)        0

 conv2d_16 (Conv2D)          (None, 16, 16, 128)       204928

 leaky_re_lu_16 (LeakyReLU)  (None, 16, 16, 128)       0

 dropout_7 (Dropout)         (None, 16, 16, 128)       0

 conv2d_17 (Conv2D)          (None, 12, 12, 256)       819456

 leaky_re_lu_17 (LeakyReLU)  (None, 12, 12, 256)       0

 dropout_8 (Dropout)         (None, 12, 12, 256)       0

 flatten_1 (Flatten)         (None, 36864)             0

 dropout_9 (Dropout)         (None, 36864)             0

 dense_3 (Dense)             (None, 1)                 36865

=================================================================
Total params: 1,113,345
Trainable params: 1,113,345
Non-trainable params: 0
_________________________________________________________________
```

7. Training loop

Traning loop :

* fix the learning of G *

Inner loop for D :

- -take m data samples & m fake data samples
- update θ_d by grad. ascent

$$\frac{\partial}{\partial \theta_d} \frac{1}{m} \left[\ln[D(x)] + \ln[1 - D(G(z))] \right]$$

* fix the learning of D *

take m fake data samples

update θ_g by grad. descent

$$\frac{\partial}{\partial \theta_g} \frac{1}{m} \left[\ln[1 - D(G(z))] \right]$$

In general we

- Create a model
- Compile model- to assign los function and optimizer
- Fit model - trains the model
-

In GANs, we keep a slower learning rate for discriminator; so that it wouldn't dominate the generator. So the discriminator doesn't learn too fast and destroy generator.

Adam - Optimizer
Binary cross entropy - Loss function
g_opt = Adam(learning_rate=0.0001)
d_opt = Adam(learning_rate=0.00001)

Setup Losses and Optimizers

```
# Adam is going to be the optimizer for both
from tensorflow.keras.optimizers import Adam
# Binary cross entropy is going to be the loss for both
from tensorflow.keras.losses import BinaryCrossentropy
```

```
g_opt = Adam(learning_rate=0.0001)
d_opt = Adam(learning_rate=0.00001)
g_loss = BinaryCrossentropy()
d_loss = BinaryCrossentropy()
```

```
] # Importing the base model class to subclass our training step
  from tensorflow.keras.models import Model
```

```
tf.random.normal((6, 128, 1))
```

```
<tf.Tensor: shape=(6, 128, 1), dtype=float32, numpy=
array([[[ 1.45390391e+00],
        [ 5.02592146e-01],
        [-4.73166734e-01],
        [-1.16095236e-02],
```

At the end of the process. I.e. at global minima …we have $P_{data}(x)$ equal to $P_{gen}(x)$.... Gen is trying to attain the state of data

7.1. Build subclassed model

These has to be defined first

- __init__()
- compile()
- train_step()

def __init__(self, generator, discriminator, *args, **kwargs): -> instantiating subclass model of gen and disc

In Python, __init__ is a constructor that is automatically called when an object of a class is created. It is used to initialise the attributes or state of the object.

*args, **kwargs -> to pass arguments

def compile(self, g_opt, d_opt, g_loss, d_loss, *args, **kwargs): -> optimizers and losses are passed

def train_step(self, batch): needs batch step -128

```
class FashionGAN(Model):
    def __init__(self, generator, discriminator, *args, **kwargs):
        # Pass through args and kwargs to base class
        super().__init__(*args, **kwargs)

        # Create attributes for gen and disc
        self.generator = generator
        self.discriminator = discriminator

    def compile(self, g_opt, d_opt, g_loss, d_loss, *args, **kwargs):
        # Compile with base class
        super().compile(*args, **kwargs)

        # Create attributes for losses and optimizers
        self.g_opt = g_opt
        self.d_opt = d_opt
        self.g_loss = g_loss
        self.d_loss = d_loss

    def train_step(self, batch):
        # Get the data
        real_images = batch
        fake_images = self.generator(tf.random.normal((128, 128, 1)), training=False)

        # Train the discriminator
        with tf.GradientTape() as d_tape:
            # Pass the real and fake images to the discriminator model
            yhat_real = self.discriminator(real_images, training=True)
            yhat_fake = self.discriminator(fake_images, training=True)
            yhat_realfake = tf.concat([yhat_real, yhat_fake], axis=0)

            # Create labels for real and fakes images
            y_realfake = tf.concat([tf.zeros_like(yhat_real), tf.ones_like(yhat_fake)], axis=0)

            # Add some noise to the TRUE outputs
            noise_real = 0.15*tf.random.uniform(tf.shape(yhat_real))
            noise_fake = -0.15*tf.random.uniform(tf.shape(yhat_fake))
            y_realfake += tf.concat([noise_real, noise_fake], axis=0)

            # Calculate loss - BINARYCROSS
            total_d_loss = self.d_loss(y_realfake, yhat_realfake)

        # Apply backpropagation - nn learn
        dgrad = d_tape.gradient(total_d_loss, self.discriminator.trainable_variables)
        self.d_opt.apply_gradients(zip(dgrad, self.discriminator.trainable_variables))
```

y_realfake = tf.concat([tf.zeros_like(yhat_real), tf.ones_like(yhat_fake)], axis=0)

Concatenate initial zeroes and ones, which are later used noise to produce images

```
# Add some noise to the TRUE outputs
noise_real = 0.15*tf.random.uniform(tf.shape(yhat_real))
noise_fake = -0.15*tf.random.uniform(tf.shape(yhat_fake))
y_realfake += tf.concat([noise_real, noise_fake], axis=0)
```

Since fake was declared as 1- we reduce its value by adding some -ve noise, similarly noise is added to real which is 0 here.

```python
        # Train the generator
        with tf.GradientTape() as g_tape:
            # Generate some new images
            gen_images = self.generator(tf.random.normal((128,128,1)), training=True)

            # Create the predicted labels
            predicted_labels = self.discriminator(gen_images, training=False)

            # Calculate loss - trick to training to fake out the discriminator
            total_g_loss = self.g_loss(tf.zeros_like(predicted_labels), predicted_labels)

        # Apply backprop
        ggrad = g_tape.gradient(total_g_loss, self.generator.trainable_variables)
        self.g_opt.apply_gradients(zip(ggrad, self.generator.trainable_variables))

        return {"d_loss":total_d_loss, "g_loss":total_g_loss}
```

with tf.GradientTape() as d_tape: -> starts calculating our gradient

```python
# Create instance of subclassed model
fashgan = FashionGAN(generator, discriminator)
```

```python
# Compile the model
fashgan.compile(g_opt, d_opt, g_loss, d_loss)
```

Compile the model

3 Build Callback

```python
import os
# import random
from tensorflow.keras.preprocessing.image import array_to_img
from tensorflow.keras.callbacks import Callback
```

```python
class ModelMonitor(Callback):
    def __init__(self, num_img=3, latent_dim=128):
        self.num_img = num_img
        self.latent_dim = latent_dim

    def on_epoch_end(self, epoch, logs=None):
        random_latent_vectors = tf.random.uniform((self.num_img, self.latent_dim,1))
        generated_images = self.model.generator(random_latent_vectors)
        generated_images *= 255
        generated_images.numpy()
        for i in range(self.num_img):
            img = array_to_img(generated_images[i])
            img.save(os.path.join('/content/drive/MyDrive/FashionGAN/images',f'generated_img_{epoch}_{i}.jpg'))
```

Callbacks- allows us to incorporate features such as advanced logging, model saving, and early stopping.

Training the model and saving it under hist - to plot it later

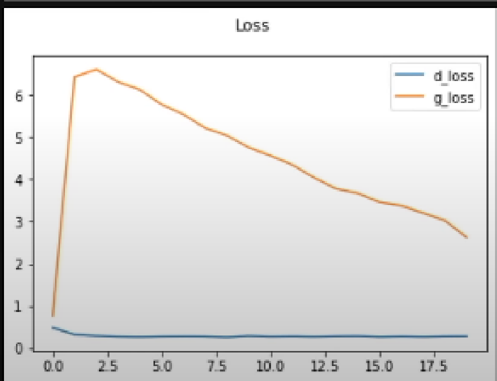
```
# Recommended 2000 epochs
hist = fashgan.fit(ds, epochs=30, callbacks=[ModelMonitor()])
```

```
469/469 [==============================] - 82s 175ms/step - d_loss: 0.3698 - g_loss: 2.9796
Epoch 3/30
469/469 [==============================] - 84s 179ms/step - d_loss: 0.2769 - g_loss: 2.6009
Epoch 4/30
469/469 [==============================] - 85s 181ms/step - d_loss: 0.2739 - g_loss: 2.6323
Epoch 5/30
469/469 [==============================] - 85s 181ms/step - d_loss: 0.2732 - g_loss: 2.6404
Epoch 6/30
469/469 [==============================] - 84s 180ms/step - d_loss: 0.2714 - g_loss: 2.6583
Epoch 7/30
469/469 [==============================] - 84s 180ms/step - d_loss: 0.2711 - g_loss: 2.6472
Epoch 8/30
469/469 [==============================] - 85s 181ms/step - d_loss: 0.5675 - g_loss: 1.4329
Epoch 9/30
469/469 [==============================] - 85s 181ms/step - d_loss: 0.4770 - g_loss: 0.1971
Epoch 10/30
469/469 [==============================] - 86s 183ms/step - d_loss: 0.3372 - g_loss: 0.0656
```

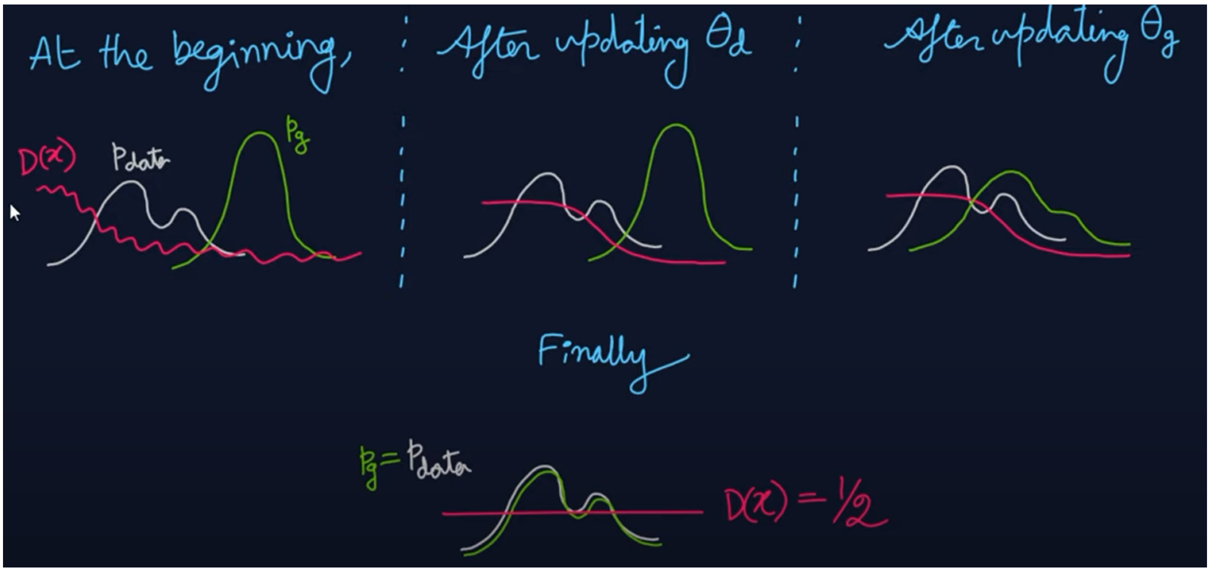
Final output

4.4 Review Performance

```
plt.suptitle('Loss')
plt.plot(hist.history['d_loss'], label='d_loss')
plt.plot(hist.history['g_loss'], label='g_loss')
plt.legend()
plt.show()
```



The complete Training Process in nutshell



At the beginning, neither the Gen, Disc have any idea what they are doing … Neither the classifier is doing anything.

When the disc is trained by updating θ_d, the classifier distinguishes fake and real data

When the gen learns something by updating θ_g , the gen tries to match distribution of gen and data with the classifier doing its job.

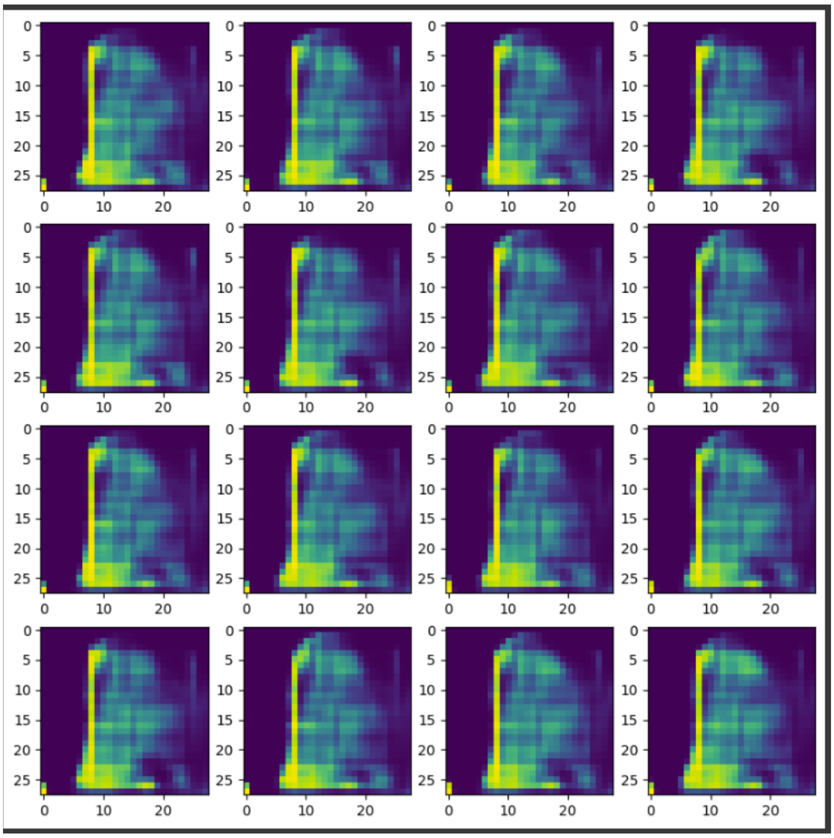
At the end when the minima of value function has been attained, the gen has successfully represented the data point

8. Testing the model

```
imgs = generator.predict(tf.random.normal((16, 128, 1)))

1/1 [==============================] - 0s 18ms/step

fig, ax = plt.subplots(ncols=4, nrows=4, figsize=(10,10))
for r in range(4):
    for c in range(4):
        ax[r][c].imshow(imgs[(r+1)*(c+1)-1])
```



This was the output which I could generate after training my model for 30 epochs, since the model is a heavy one train.
Further training this model could produce images like the fashion_mnist dataset.