

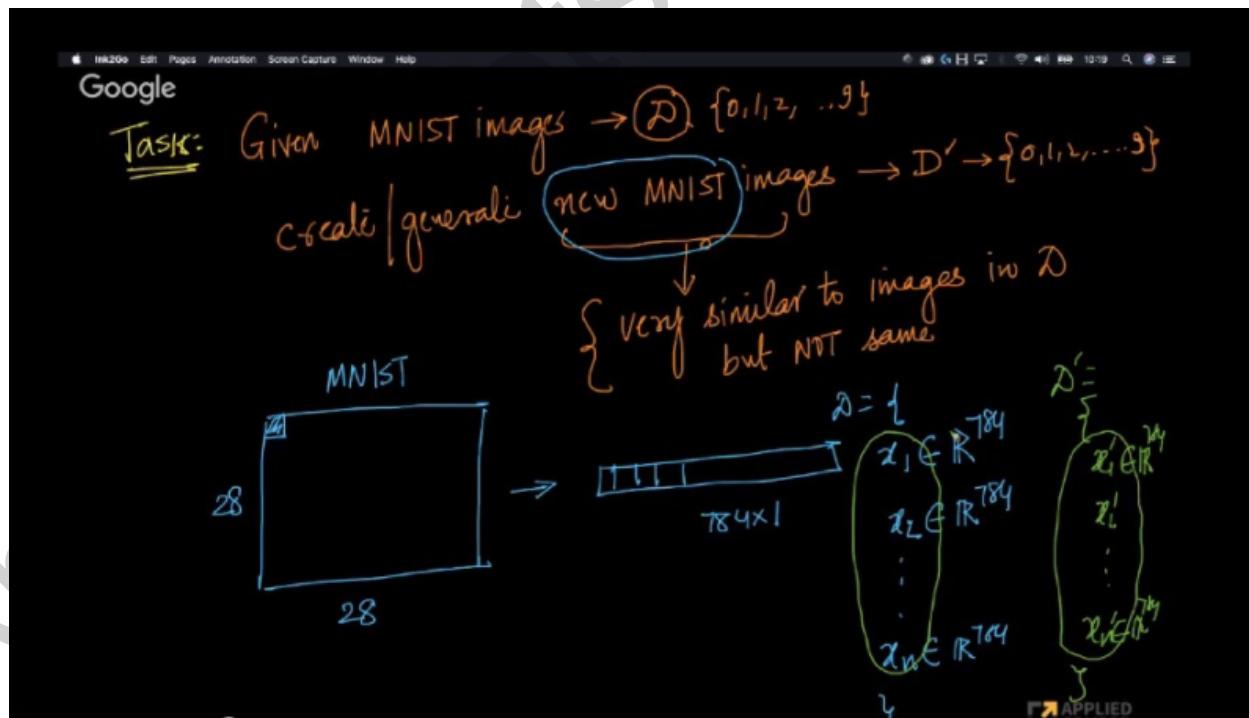
## 63.1 Generative Adversarial Networks (GAN)

Let's dive into the topic. We already know what a network is. It's nothing but a neural network which is either deep or shallow. There are two new terms i.e, generative and adversarial which needs to be addressed.

We will basically start from scratch in understanding the topic.

### What GAN will help us achieve ?

Given a set of images which is from the MNIST dataset, we need to devise an algorithm which generates new images which are similar to it. Basically, the images from the MNIST dataset are drawn from a certain distribution. Let's call it D. Our task is to sample some more images from the D which is not part of the set we have. For this, we need to understand the distribution D.



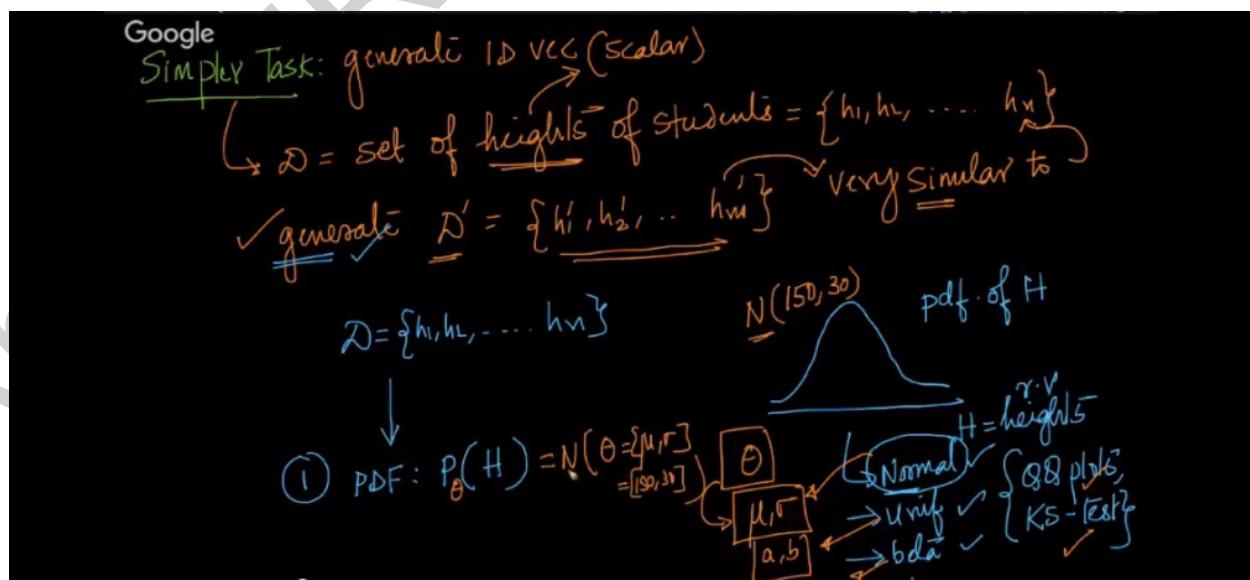
Timestamp : 21:50

In the above image, we can see a sample image from the MNIST dataset. Basically each image is of 28\*28 pixels. We can flatten them into a 1-dimensional vector. So, the dimension of the vector would be 784 since there are total of 784 pixel values in a 28\*28 image. In a dataset, we have a collection of images. This means, we have a collection of 784 dimensional vectors which are represented as  $x_i$  where  $i$  ranges from 1 to  $n$  where  $n$  is the total number of images in the dataset.

Our task is to sample some more vectors which are similar to the vectors in the dataset. We represent those vectors as  $x'_i$ . This way, we are basically generating images which are similar to the images in the MNIST dataset.

Let's consider a simpler task.

Let  $D$  be the set of heights of students in a classroom. So,  $D$  is represented as  $\{h_1, h_2, h_3, \dots, h_n\}$ . Each  $h_i$  belongs to the real space i.e., a real number. Now, our task is to generate  $h'_i$  which are similar to the  $h_i$ 's in the dataset  $D$ . So, we need to generate a new dataset  $D'$  which is represented as  $\{h'_1, h'_2, \dots, h'_n\}$  such that it's similar to  $D$ . Why is this a simpler task? In the previous case, we need to generate a 784 dimensional vector which is quite large. Here, we only need to generate a real number which is a scalar. So, the problem is relatively simple.



Timestamp : 29:09

Since we have the set of values in our dataset D, we can plot an empirical pdf. It can be done using the function histogram() available in the numpy library. But ,the problem is we don't know the theoretical distribution from which these values come. To find it, we can use the QQ plot or any other statistical techniques like KS-test,etc. Let's assume, we found that these points are sampled from a normal distribution with mean and standard deviation specified. The parameters i.e, the mean and standard deviation can be estimated using maximum likelihood estimation method. So, now we know the exact distribution which the values come from.

### **So, how to sample or generate new points or height values which are similar to the points in the D ?**

It's quite simple. We can simply generate random numbers from a normal distribution with mean and standard deviation specified. We can use the function random.normal() available in the numpy library. So, we achieved our task.

This means D and D' are similar. In this context, similarity basically means that the set of generated values i.e, D' are also sampled from the same distribution as the D does.

To make it simple, there is one distribution which is represented as P. We sample some points and put them in a set named D. Now, using the values in the set D, we are basically generating values which are part of P. That is, we are indirectly learning about the distribution P.

Since both D and D' are sampled from P, they are similar.

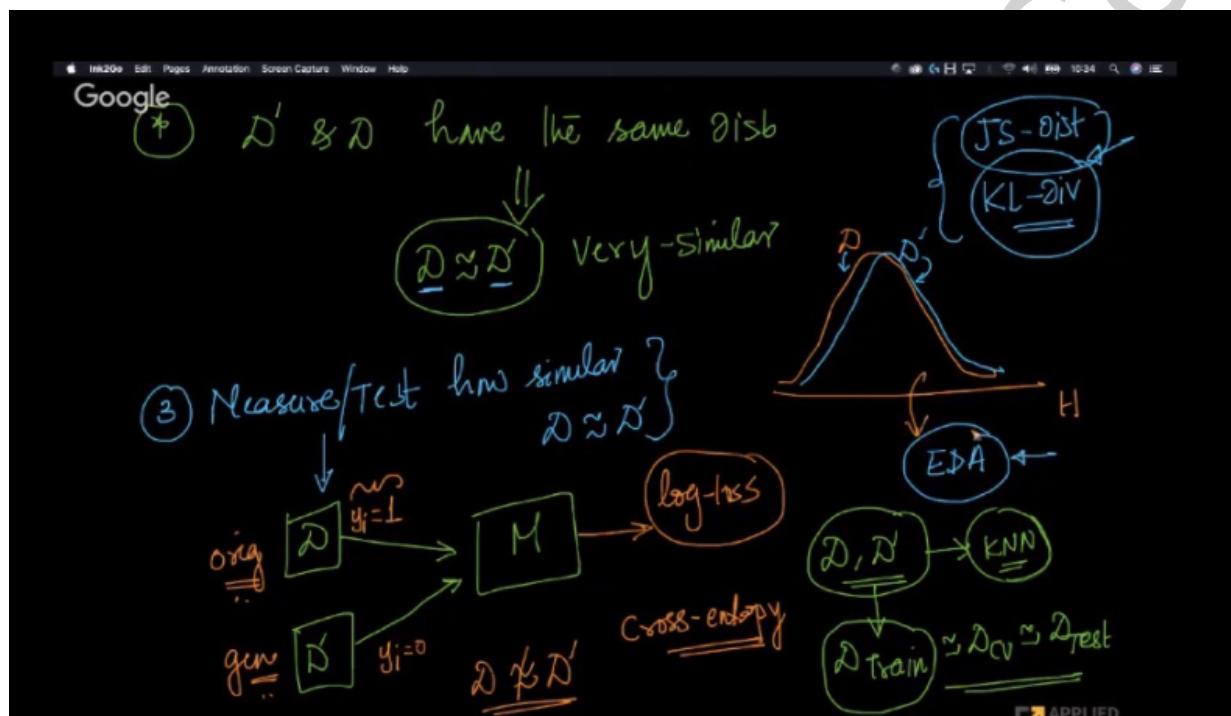
There are many measures available to check whether the two distributions are similar or not. For example, we have KL- divergence.

One way to check is as follows.

Create a dataset A with  $\{(x_i, y_i)\}$  where  $x_i$  belongs to  $D$  and  $y_i$  is 1 for all  $i$ .

Create a dataset B with  $\{(x_i, y_i)\}$  where  $x_i$  belongs to  $D'$  and  $y_i$  is 0 for all  $i$ .

Now, create a model M. The model M should return 1 if  $x_i$  is in A or 0.



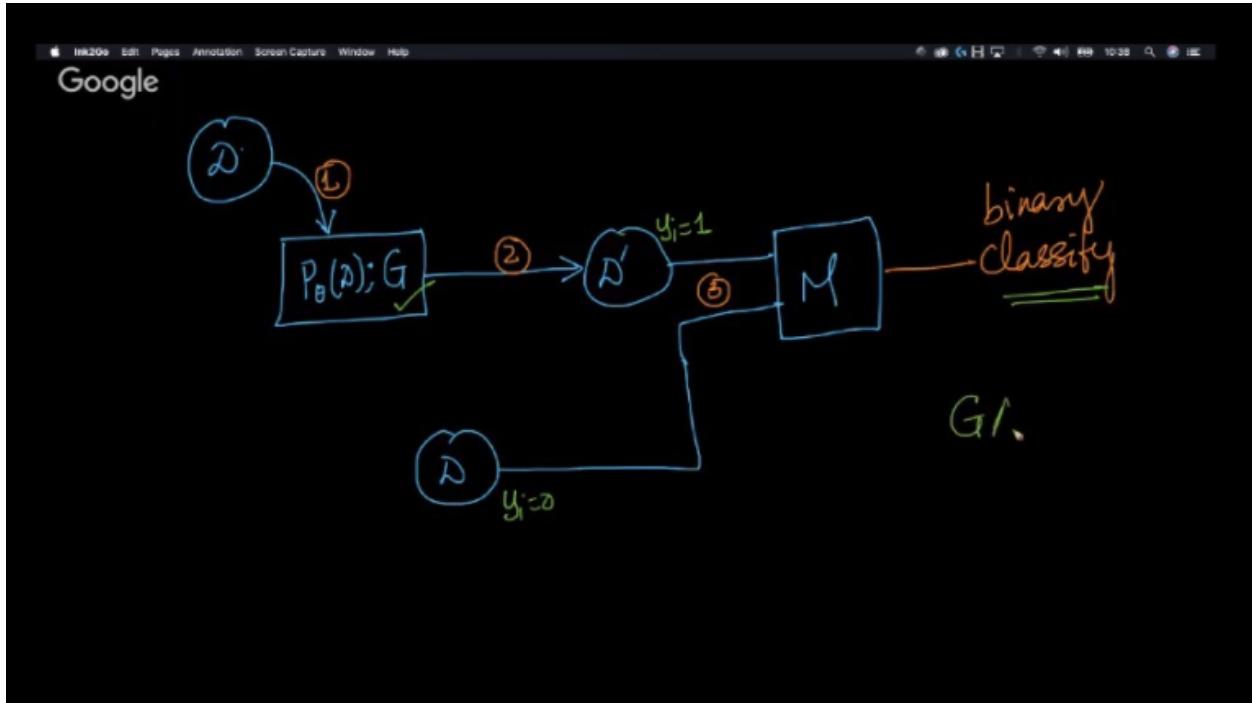
Timestamp : 36:43

For this problem, we use log-loss . This is a simple binary classification problem.

So, if the model gets a good accuracy, then it can separate  $D$  and  $D'$ . This implies that  $D$  and  $D'$  are not similar. Similarly, they are similar if the accuracy is low.

The model M could be any neural network or any valid algorithm.

## Big picture of GAN.



Timestamp : 41:02

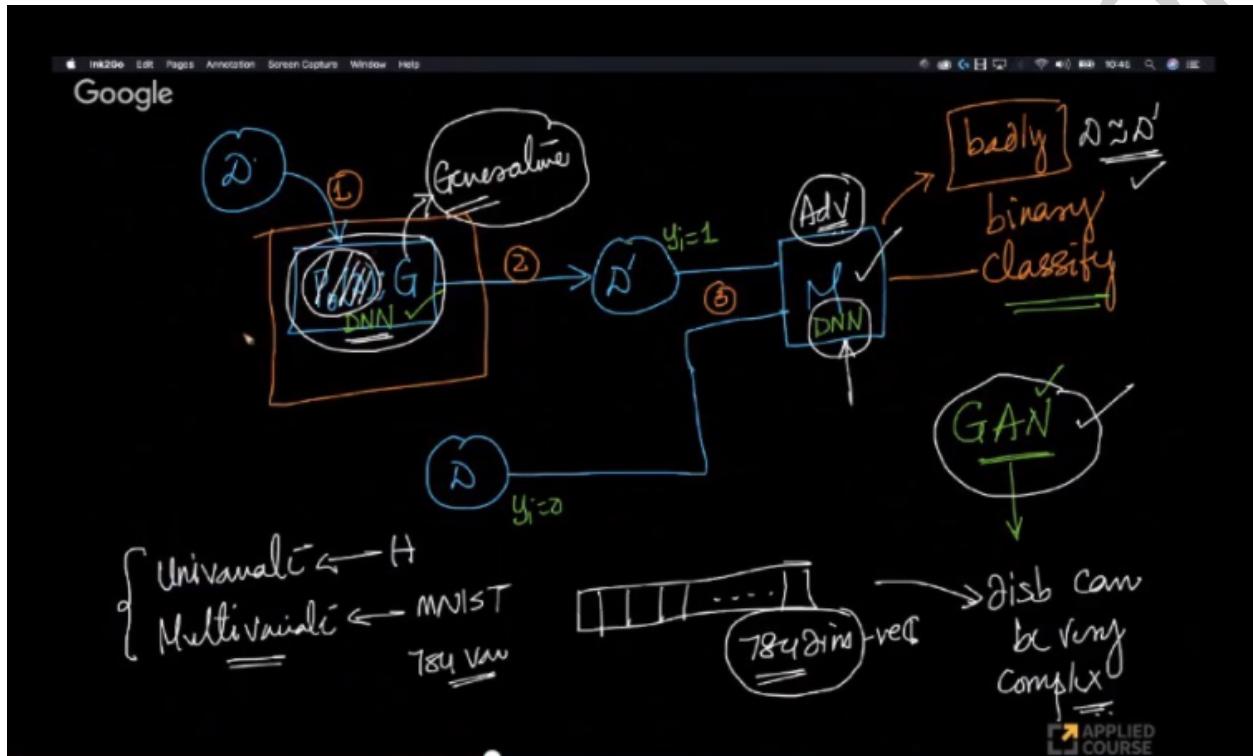
- 1.) We first generate some samples and put them in  $D'$  or  $D'$  is the set of generated samples. How do we generate them ? We already discussed the procedure. We first find the probability distribution from where the  $D$  comes from or find the parameters to it. Once we know, we sample from that distribution.
- 2.) Then, we send both  $D$  and  $D'$  to the model and try to classify them.
- 3.) If the accuracy is high, then  $D$  and  $D'$  are not similar. If the accuracy is low, then  $D$  and  $D'$  are similar.

In GAN, rather than finding a probability distribution using some statistical techniques or any plotting techniques, we use a deep neural network. Also, we use the same for the final model too. The first part of the network is called the generator network and the second part is the discriminator network. It's self-evident.

We previously discussed the solution for a real valued feature where the dimension is 1. But, in the start we were trying to solve it for the MNIST

dataset where the dimension of the feature is 784. So, finding the distribution for high dimensional data is quite difficult. Even visualizing the 784 dimensional space is hard right ? We are confined to 3 dimensions.

KI divergence and other metrics doesn't work well for high dimensional data. So, we need to rely on other methods.



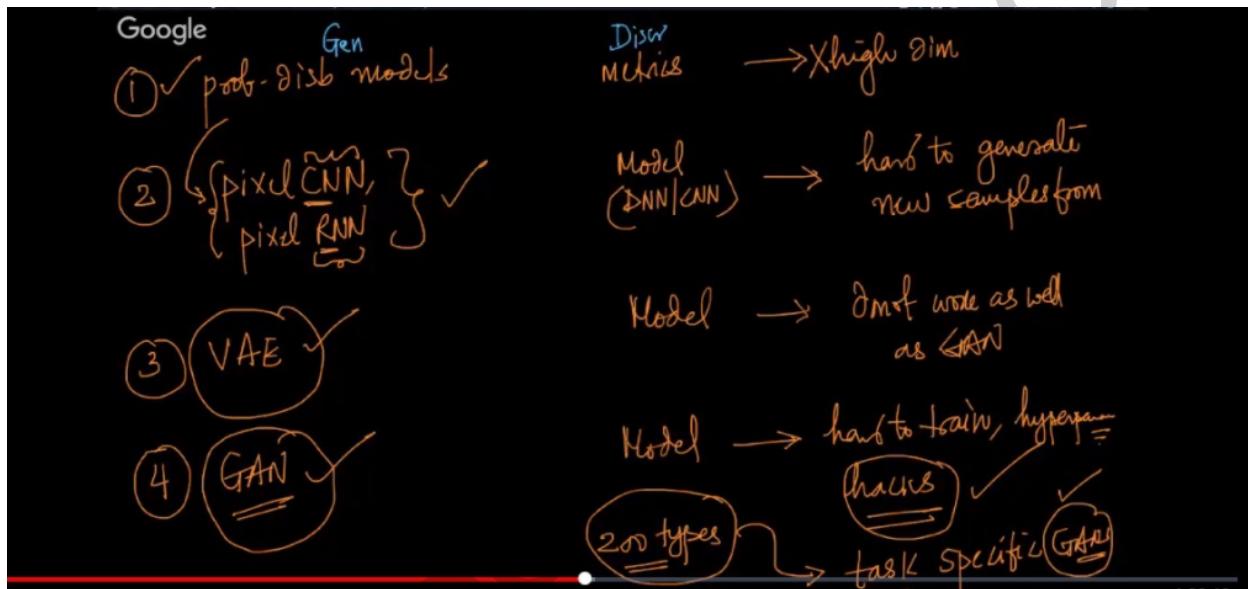
We can see that we replaced both the parts by a deep neural network. So, if the final model M which is a dnn gets a bad accuracy, then D and D' are similar. This means, we can simply take the first model which is a dnn and generate samples. This is because D and D' are similar.

Actually, GAN became popular after 2017. But, they are extremely hard to train. We will discuss some of them.

Finding the parameters for the probability distribution for high dimensional data is very difficult. So, maximum likelihood estimation or any statistical based techniques doesn't work well.

Another method is using pixel cnn/pixel rnn. But, using this method it's hard to generate new samples.

We can also use variational autoencoders.



Timestamp : 58:30

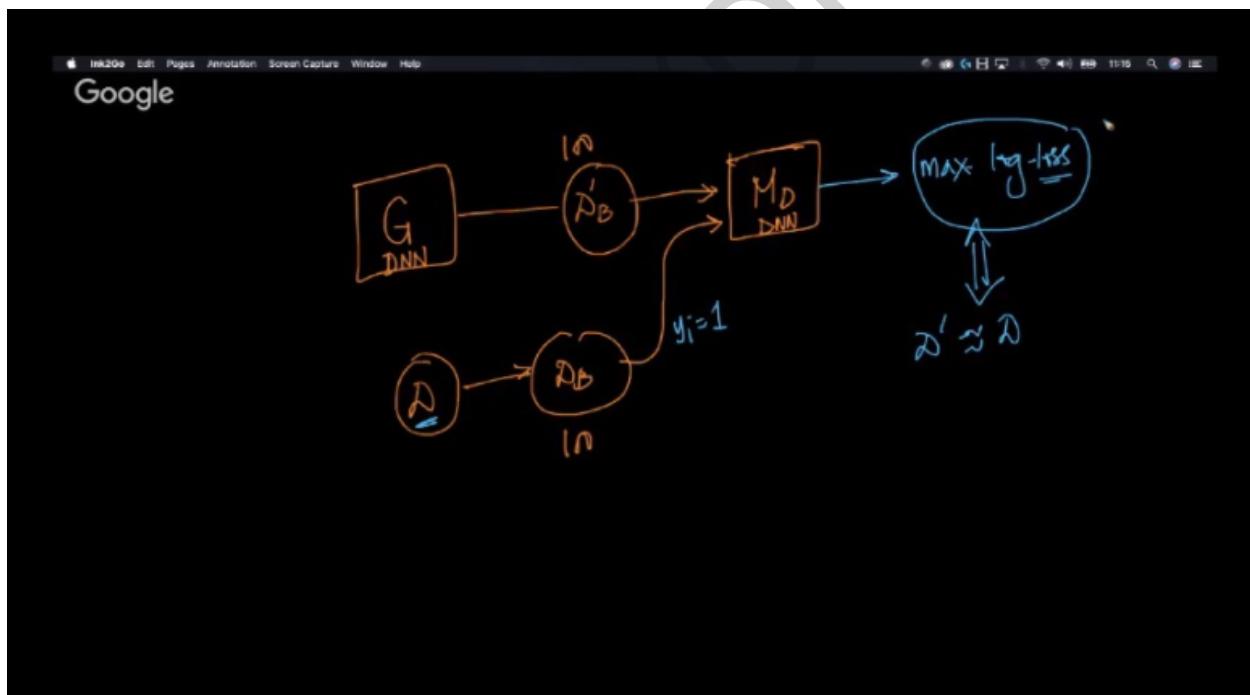
## Some applications of GAN.

- 1.) Generating anime characters. It can also colorize it. For both generator and discriminator networks, we use a CNN.
- 2.) Pose guided person image generation. In this, given an image of a person and a description of the keypoints, we need to generate a new image of that same person according to the keypoints. Simply, we need to generate an image of that same person in various poses.
- 3.) Cycle GAN. In this, given an image A and an image B, we need to reconstruct the image A in terms of B. For example, in social media

while sharing the pictures, we used to add the filters. Here, the filter serves as the image B and our own image is A.

- 4.) PixelGAN : Creating clothing images and styles from an image.
- 5.) SRGAN : Creating super resolution images.
- 6.) Progressive GAN : Progressively using the GAN to generate high quality images. Basically, the entire process is split into multiple phases. In each phase, we generate a small image. At the end of the phase, by some specified procedure we put it altogether and create the final image.

Even though there are many applications to it and seems different for each case, the underlying mechanism of GAN remains to be the same.



Timestamp : 01:18:27

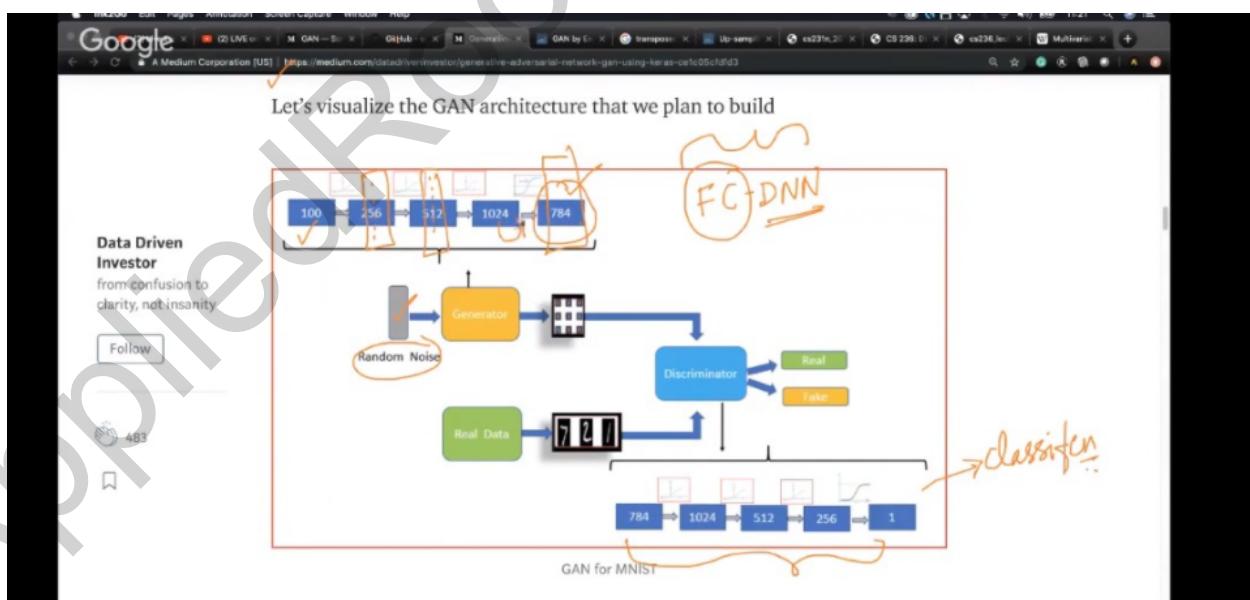
- 1.) First, we generate some sample data using the generator network. Let's say we do it in batches. Let's also assume that each batch contains 100 points.

- 2.) So, to the discriminator model, we need to provide 100 points which are from the D which is the original dataset we are working with. For this, we give a label 1 since it's the original sample points.
- 3.) For the generated points, we give it a label 0.
- 4.) This point would be a little confusing. Here, we need to maximize the log loss. This is because we want the generated images to be closer to the actual ones.

The process of GAN is similar to game theory.

## Training process.

- 1.) At first, the generator network needs to generate some images or a vector right ? We know that a generator network is a neural network. For any neural network, we want the input to be present. Otherwise, we can't produce an output. So, for this generator network we also need some inputs. In this case, we give a random vector of size 100 or any valid size drawn from the normal distribution.
- 2.) What does the generator network really do ?



Timestamp : 01:23:50

It takes in a 100 dimensional random vector drawn from a normal distribution. With series of layers in the generator network which is a neural network, it transforms the input to the final 784 dimensional output which is a image if we reshape it to 28\*28 (MNIST) . Here, the generator network is a CNN , since we are dealing with images. The reason why we again downsample from 1024 to 784 will be discussed later.

First, we initialize the generator network which is a neural network with some random weights. This means, the output produced by the generator network at the very first epoch is some random vector or nothing meaningful.

- 1.) First, fix the final model M which is a discriminator and then generate some random 100 vectors which are 784 dimensional. Then, train the generator network.
- 2.) Now, fix the generator and train the discriminator network. Here we need to maximize the log loss. So, for this we use gradient ascent rather than gradient descent algorithm.
- 3.) Actually, people have tried updating both the networks simultaneously rather than the former approach discussed. But, that doesn't work well. This is called alternate minimization/maximization.

**Let's look at the code.**

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

import keras
from keras.layers import Dense, Dropout, Input
from keras.models import Model, Sequential
from keras.datasets import mnist
from tensorflow import keras
from keras.layers.advanced_activations import LeakyReLU
from keras.optimizers import adam
```

Let's import the required libraries

Data Driven Investor  
from confusion to clarity, not insanity

Follow 483

Loading the data from mnist dataset. we create a function load\_data() function

Timestamp : 01:34:34

As usual, we are importing a set of libraries which are needed. We are using LeakyReLU rather than ReLU.

```
def load_data():
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    x_train = (x_train.astype(np.float32) - 127.5)/127.5

    # convert shape of x_train from (60000, 28, 28) to (60000, 784)
    # 784 columns per row
    x_train = x_train.reshape(60000, 784)
    return (x_train, y_train, x_test, y_test)

(X_train, y_train, X_test, y_test)=load_data()
print(X_train.shape)
```

Applied Course

(60000, 784)

X\_train.shape

APPLIED COURSE

Timestamp : 01:34:57

We are loading the MNIST dataset. It's readily available in the keras library.

Then, we write the code for defining the adam optimizer.

The screenshot shows a browser window displaying a Medium article. The code in the article is:

```
def create_generator():
    generator=Sequential()
    generator.add(Dense(units=256, input_dim=100))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(units=512))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(units=1024))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(units=784, activation='tanh'))

    generator.compile(loss='binary_crossentropy',
                      optimizer=adam_optimizer())
    return generator
g=create_generator()
g.summary()
```

Below the code, there is a table showing the model summary:

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 256)	25856

Timestamp : 01:35:43

Now, we are going to define the generator network. As we already know, a generator is a neural network. So, we create a sequence of Dense layers with LeakyReLU as the activation function. The input\_dim is 100 for the first Dense layer. This means, our input is a 100 dimensional random vector drawn from the normal distribution. We use the binary cross entropy loss as we are dealing with a binary classification problem.

## Why do we use tanh activation in the final Dense layer ?

Rather than normalizing the images in the range [0,1] , we normalize it to the range [-1,1] . So, we made the tanh as the activation function in the last layer since it's range also matches with it. The reason solely depends on the empirical observations.

Then, we can print the summary of our model.

```
g=create_generator()
g.summary()
```

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 256)	25856
leaky_re_lu_1 (LeakyReLU)	(None, 256)	0
dense_2 (Dense)	(None, 512)	131584
leaky_re_lu_2 (LeakyReLU)	(None, 512)	0
dense_3 (Dense)	(None, 1024)	525312
leaky_re_lu_3 (LeakyReLU)	(None, 1024)	0
dense_4 (Dense)	(None, 784)	803600

Total params: 1,486,352  
Trainable params: 1,486,352  
Non-trainable params: 0

Timestamp : 01:38:39

We have around 1,486,352 parameters.

Then, we need to define the discriminator network.

```
def create_discriminator():
    discriminator=Sequential()
    discriminator.add(Dense(units=1024,input_dim=784))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))

    discriminator.add(Dense(units=512))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))

    discriminator.add(Dense(units=256))
    discriminator.add(LeakyReLU(0.2))

    discriminator.add(Dense(units=1, activation='sigmoid'))

    discriminator.compile(loss='binary_crossentropy',
        optimizer=adam_optimizer())
    return discriminator
d =create_discriminator()
d.summary()
```

Layer (type)	Output Shape	Param #
Dense (Dense)	(None, 1024)	1025
LeakyReLU (Activation)	(None, 1024)	0
Dropout (Dropout)	(None, 1024)	0
Dense (Dense)	(None, 512)	513
LeakyReLU (Activation)	(None, 512)	0
Dropout (Dropout)	(None, 512)	0
Dense (Dense)	(None, 256)	257
LeakyReLU (Activation)	(None, 256)	0
Dense (Dense)	(None, 1)	1
Sigmoid (Activation)	(None, 1)	0

Timestamp : 01:39:54

Here, the input size is 784 since the discriminator takes the original image as the input. Then, as usual we define a sequence of Dense layers with LeakyReLU as the activation function. We use binary cross entropy as the loss function. In the final Dense layer, we use sigmoid as the activation function, since we are outputting the probability values and not generating any images like the generator does.

Now, we have to create the GAN from the generator and discriminator network.

A screenshot of a web browser displaying a Medium article. The code shown is for creating a Generative Adversarial Network (GAN) using Keras. It defines a function `create\_gan` that takes a discriminator and generator model as inputs. The discriminator is set to non-trainable. An input vector of shape (100,) is passed through the generator to produce a sample. This sample is then passed through the discriminator to get a binary cross-entropy loss. The generator and discriminator are compiled with Adam optimizer. Finally, the GAN model is created by combining the generator and discriminator, and its summary is printed.

```
def create_gan(discriminator, generator):
    discriminator.trainable=False
    gan_input = Input(shape=(100,))
    x = generator(gan_input)
    gan_output=discriminator(x)
    gan= Model(inputs=gan_input, outputs=gan_output)
    gan.compile(loss='binary_crossentropy', optimizer='adam')
    return gan

gan = create_gan(d,g)
gan.summary()
```

The summary table shows the following details:

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 100)	0
sequential_1 (Sequential)	(None, 784)	1486352
sequential_2 (Sequential)	(None, 1)	1460225
Total params:	2,946,577	
Trainable params:	1,486,352	

Timestamp : 01:40:48

- 1.) First, we are freezing the discriminator model by setting it's trainable attribute to False.
- 2.) Then, we are defining the input to the gan as a 100 dimensional input vector.
- 3.) Then, we are passing the input to the generator and getting the corresponding output.
- 4.) Also, we pass the input to the discriminator network.
- 5.) Then, we are compiling the model with loss as binary cross entropy and adam as the optimizer.

Then, we can print the summary of our model.

A screenshot of a web browser displaying a Python script and its summary table. The script defines a GAN model and creates a summary. The summary table shows the following data:

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 100)	0
sequential_1 (Sequential)	(None, 784)	1486352
sequential_2 (Sequential)	(None, 1)	1460225
Total params:	2,946,577	
Trainable params:	1,486,352	
Non-trainable params:	1,460,225	

The last two rows of the table, which contain the total and trainable parameters, are circled in red.

```
gan_output= discriminator(x)
gan= Model(inputs=gan_input, outputs=gan_output)
gan.compile(loss='binary_crossentropy', optimizer='adam')
return gan

gan = create_gan(d,g)
gan.summary()

Data Driven
Investor
from confusion to
clarity, not insanity

Follow

483

GAN Summary
```

Before we start training the model, we will write a function

Timestamp : 01:42:28

Since we have freezed the discriminator network, we also have some non-trainable parameters.

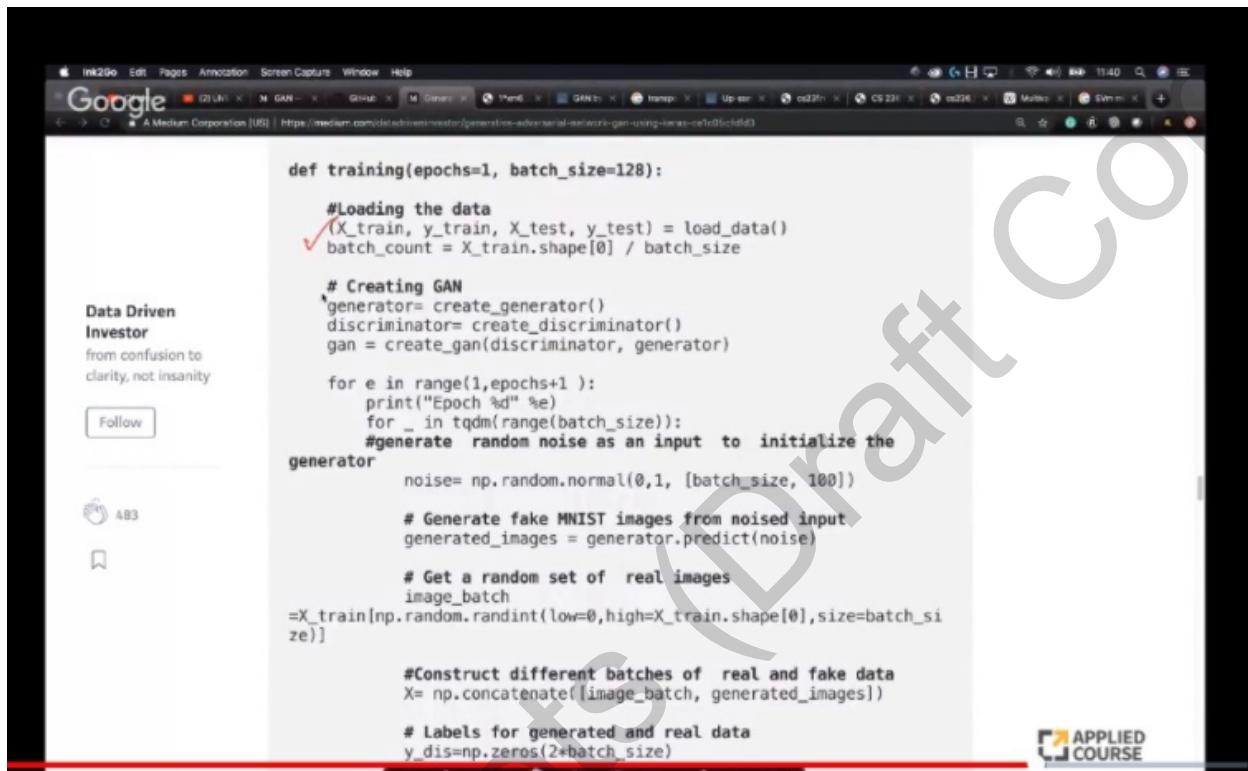
Then, we are defining a utility function which plots the images which are generated by the network.

A screenshot of a web browser displaying a Python script for generating and plotting images. The script defines a function `plot_generated_images` that takes an epoch, generator, and examples as input. It generates noise, uses the generator to predict images, reshapes them, and then plots them in a grid. The plot is saved as a PNG file.

```
def plot_generated_images(epoch, generator, examples=100, dim=(10,10), figsize=(10,10)):
    noise= np.random.normal(loc=0, scale=1, size=[examples, 100])
    generated_images = generator.predict(noise)
    generated_images = generated_images.reshape(100,28,28)
    plt.figure(figsize=figsize)
    for i in range(generated_images.shape[0]):
        plt.subplot(dim[0], dim[1], i+1)
        plt.imshow(generated_images[i], interpolation='nearest')
        plt.axis('off')
    plt.tight_layout()
    plt.savefig('gan_generated_image %d.png' %epoch)
```

Timestamp : 01:42:55

Next, we need to train both the generator and discriminator. Here, we can't simply use model.fit() and train it. In GAN, we are using alternate minimization/maximization. So, the procedure changes.



```
def training(epochs=1, batch_size=128):
    #Loading the data
    ✓(X_train, y_train, X_test, y_test) = load_data()
    batch_count = X_train.shape[0] / batch_size

    # Creating GAN
    generator= create_generator()
    discriminator= create_discriminator()
    gan = create_gan(discriminator, generator)

    for e in range(1,epochs+1):
        print("Epoch %d" %e)
        for _ in tqdm(range(batch_size)):
            #generate random noise as an input to initialize the
            generator
            noise= np.random.normal(0,1, [batch_size, 100])

            # Generate fake MNIST images from noised input
            generated_images = generator.predict(noise)

            # Get a random set of real images
            image_batch
            =X_train[np.random.randint(low=0,high=X_train.shape[0],size=batch_si
            ze)]

            #Construct different batches of real and fake data
            X= np.concatenate([image_batch, generated_images])

            # Labels for generated and real data
            y_dis=np.zeros(2*batch_size)
```

Timestamp : 01:43:31

- 1.) First ,we load the data.
- 2.) Then , we create both the generator and discriminator network by calling the functions create\_generator() and create\_discriminator().
- 3.) Then, creating the gan by calling the function create\_gan()
- 4.) First, we will generate a random noise vector which is of size 100.
- 5.) Next, we pass the noise vector to the generator network and get the generated image.
- 6.) Now, create the set of images for the discriminator network.
- 7.) Next, we need to create the real and fake data i.e, we need to label the two sets of images.

The screenshot shows a Google search result for 'Data Driven Investor' on Medium. The page title is 'GANs in Python using Keras - Data Driven Investor'. The main content is a Python script for training a Generative Adversarial Network (GAN) using Keras. The code includes comments explaining the steps: pre-training the discriminator, tricking the generator's input, training the GAN by alternating between discriminator and generator training, and plotting generated images every 20 epochs.

```
# Labels for generated and real data
y_dis=np.zeros(2*batch_size)
y_dis[:batch_size]=0.9

#Pre train discriminator on fake and real data before
starting the gan.
discriminator.trainable=True
discriminator.train_on_batch(X, y_dis)

#Tricking the noised input of the Generator as real data
noise= np.random.normal(0,1, [batch_size, 100])
y_gen = np.ones(batch_size)

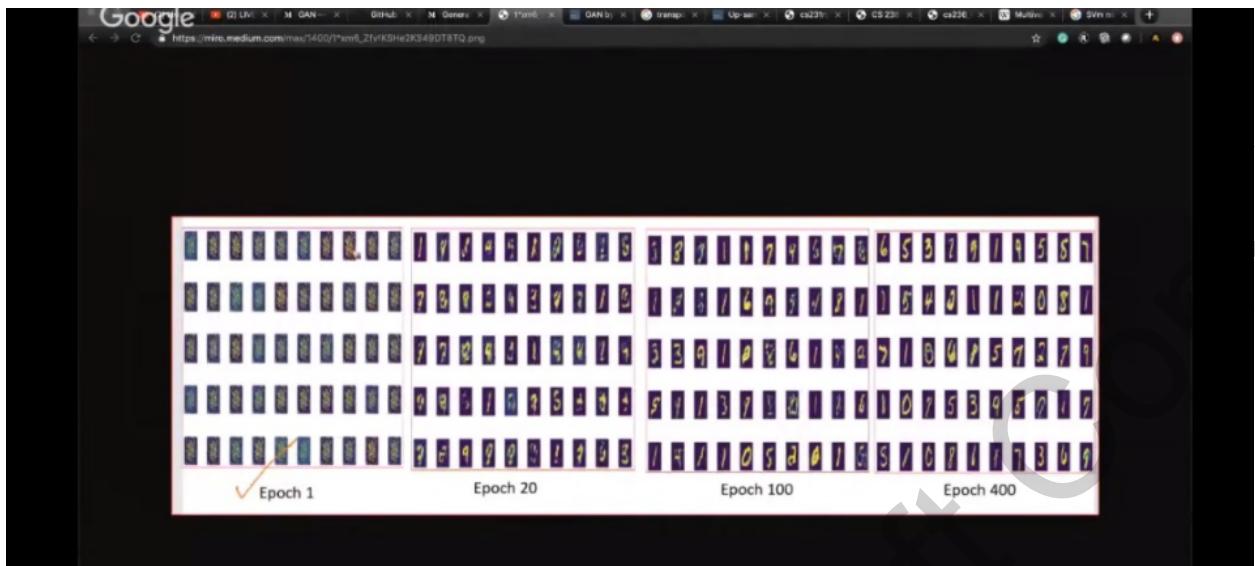
# During the training of gan,
# the weights of discriminator should be fixed.
#We can enforce that by setting the trainable flag
discriminator.trainable=False

#training the GAN by alternating the training of the
Discriminator
#and training the chained GAN model with Discriminator's
weights freezed.
gan.train_on_batch(noise, y_gen)

if e == 1 or e % 20 == 0:
    plot_generated_images(e, generator)
training(400,128)
```

Timestamp : 01:47:03

- 8.) We set the attribute trainable belonging to the discriminator as True.  
This means, we are training the discriminator alone.
- 9.) Now, set the attribute trainable belonging to the discriminator as False. Now, we are training only the generator network.
- 10.) This is called alternate optimization.

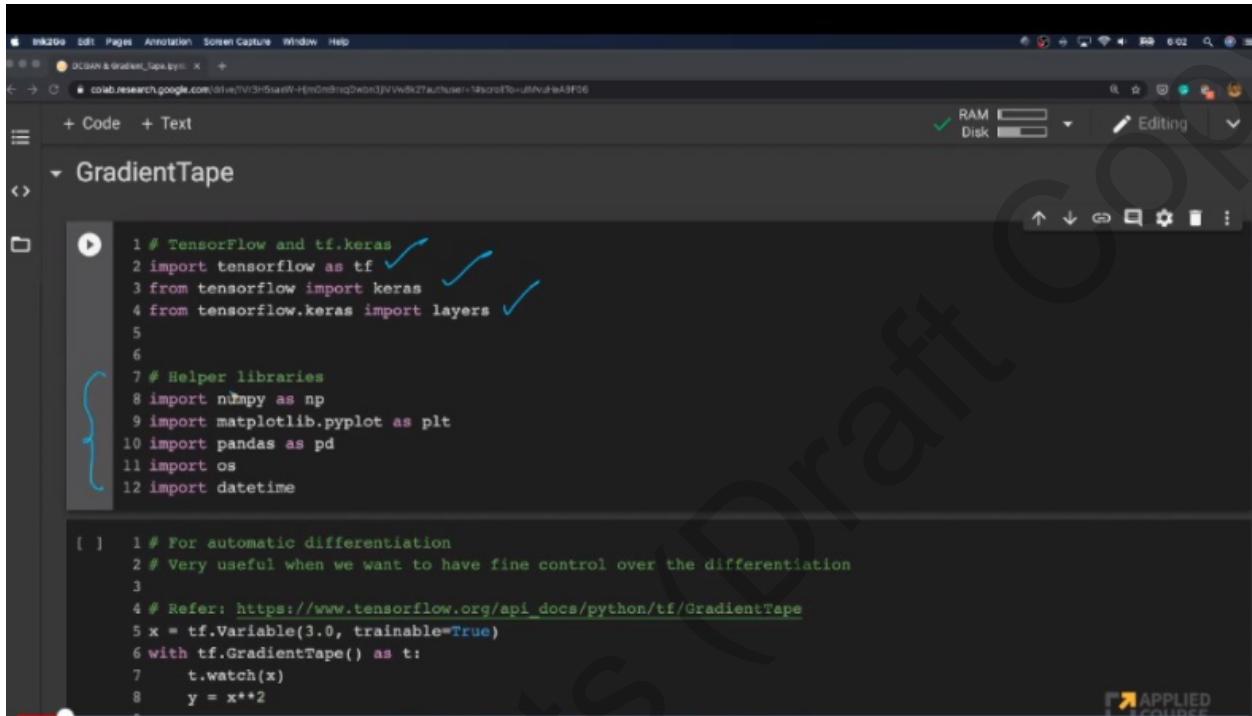


Timestamp : 01:50:18

The above image describes the generated images by the generator network as the epoch increases.

As we can see, as the epoch increases the generated images tend to look like the original ones.

## 63.2 Code Walkthrough: DC-GANs and GradientTape in TensorFlow 2



```
# TensorFlow and tf.keras
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import os
import datetime

# For automatic differentiation
# Very useful when we want to have fine control over the differentiation
# Refer: https://www.tensorflow.org/api_docs/python/tF/GradientTape
x = tf.Variable(3.0, trainable=True)
with tf.GradientTape() as t:
    t.watch(x)
    y = x**2
```

Timestamp: 05:37

As usual, we are importing a set of libraries which are already familiar to us.

**Gradient Tape** : It gives us a fine control over the gradients. In keras, while defining any layer, we only define the forward function. The operations needed to be done in the backward stage i.e, backpropagation are taken care of by the keras. It's called automatic differentiation.

```
1 # For automatic differentiation
2 # Very useful when we want to have fine control over the differentiation
3
4 # Refer: https://www.tensorflow.org/api\_docs/python/tf/GradientTape
5 x = tf.Variable(3.0, trainable=True)
6 with tf.GradientTape() as t:
7     t.watch(x)
8     y = x**2
9 print(t.gradient(y, x).numpy())
10
```

Timestamp : 19:48

- 1.) First, we are defining a tensorflow variable named x by calling `tf.Variable()`. We pass the value the variable needs to hold and we set the parameter `trainable` as `True`. This means, gradients can be calculated with respect to this variable. If it's set to `False`, then the gradients cannot be computed.
- 2.) Then, we are in the context of `tf.GradientTape` by using the `with` keyword.
- 3.) Then, we call the `watch` function on the variable x. This function will simply track the variable x for further gradient computations.
- 4.) Then, we define another variable y as `x**2`. This is the forward computation we were talking about.
- 5.) Then, we can call the function `gradient(y,x)` and get the numerical gradient of the variable y with respect to x.

This seems like a simple thing to do. But, with this we can perform complex computations too.

The screenshot shows a Jupyter Notebook interface with two code cells. The top cell contains Python code using TensorFlow's GradientTape to compute the second derivative of  $y = x^2$ . The bottom cell shows the resulting output: a tensor of shape () with value 6.0 and a tensor of shape () with value 2.0.

```
4 # Refer: https://www.tensorflow.org/api\_docs/python/tf/GradientTape
5 x = tf.Variable(3.0, trainable=True)
6 with tf.GradientTape() as t:
7     t.watch(x)
8     y = x**2
9
10 print(t.gradient(y, x).numpy())
11
12
13 x = tf.constant(3.0)
14 with tf.GradientTape() as g:
15     g.watch(x)
16     with tf.GradientTape() as gg:
17         gg.watch(x)
18         y = x * x
19         dy_dx = gg.gradient(y, x)      # Will compute to 6.0
20         d2y_dx2 = g.gradient(dy_dx, x) # Will compute to 2.0
21
22
23 print(dy_dx)
24 print(d2y_dx2)

tf.Tensor(6.0, shape=(), dtype=float32)
tf.Tensor(2.0, shape=(), dtype=float32)
```

Timestamp : 21:21

In the previous example, we did first order differentiation i.e, we computed the first order derivative of  $y$  with respect to  $x$ . Let's say we want to perform the second order differentiation i.e, computing the second order derivative of  $y$  with respect to  $x$ . How to do it?

- 1.) First, we will define  $x$  as a tensorflow constant with 3.0
- 2.) Then, we are in the context of Gradient Tape with  $g$  as the context variable.
- 3.) Then, we call the watch function on the variable  $x$ . Even Though,  $x$  is defined to be a constant, when we call `watch()` on this variable, we can compute the gradients with respect to this variable. Now, we open another context with  $gg$  as the context variable.
- 4.) Again, we call the `watch()` function on the variable  $x$  with  $gg$  as the context variable. Then, we define  $y$  as  $x^2$ .
- 5.) Now, we compute the  $dy/dx$  which is nothing but  $2*x$  and store it in the  $dy_dx$  variable. This is done by using the `gradient()` function on the  $x$  variable with  $gg$  as the context variable.

6.) Then, we compute the second order derivative by calling the gradient() function on dy\_dx and x . Note: dy\_dx holds 2\*x. The second order derivative is nothing but the constant 2.

Let's see another example.

```
 8 d2y_dx2 = g.gradient(dy_dx, x) # Will compute to 2.0
 9
10 print(dy_dx)
11 print(d2y_dx2)

[ ] 1 x = tf.constant(3.0)
2 with tf.GradientTape() as g:
3     g.watch(x)
4     y = x * x
5     z = y * y
6 dz_dx = g.gradient(z, x) # 108.0 (4*x^3 at x = 3)
7 dy_dx = g.gradient(y, x) # 6.0
8
9

-----  
RuntimeError                                     Traceback (most recent call last)  
<ipython-input-5-907f055cfbc1> in <module>()
```

$y = x^2$   
 $z = y^2$ .

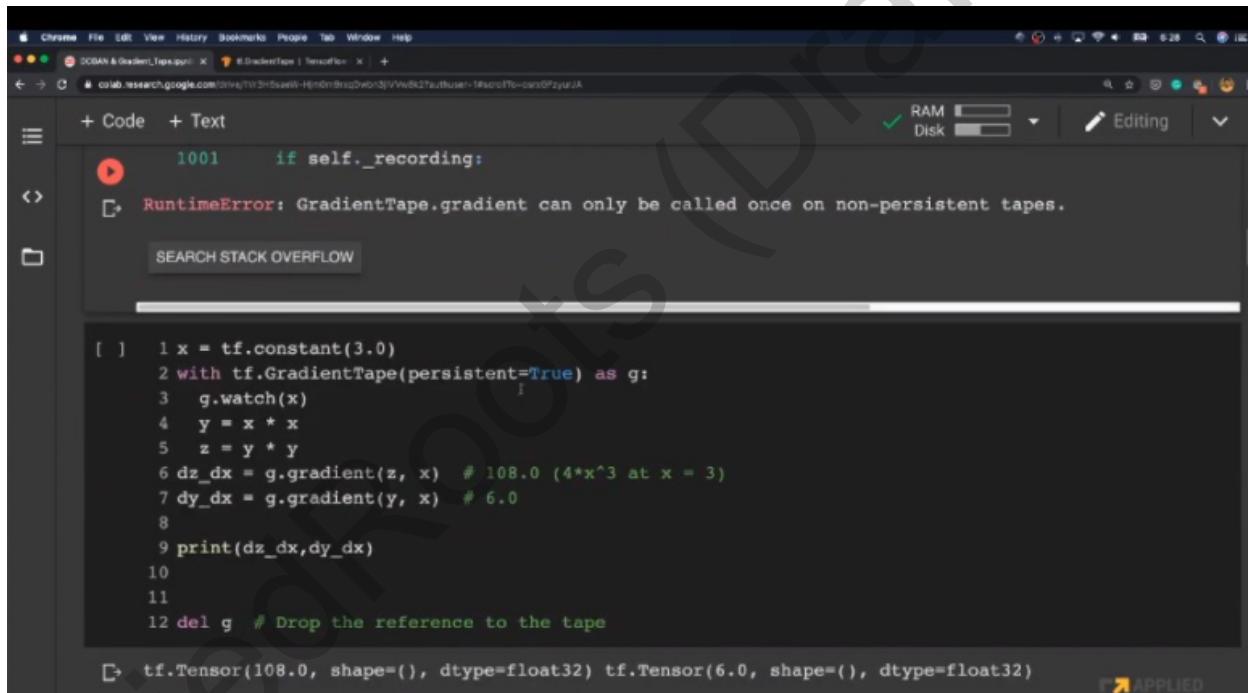
Timestamp : 27:32

Here, we are getting a runtime error. We will understand the reason behind it.

- 1.) First, we defined a tensorflow constant as x. Then, we stored the value 3 in it.
- 2.) Then, we are in the GradientTape context with the context variable as g.
- 3.) Now, we are calling the watch function on the variable .
- 4.) Now, we are defining y as x\*x and z as y\*y.
- 5.) Now, we are calculating dz\_dx which is nothing but the derivative of z with respect to x. This is done by calling the gradient function on z and x.

6.) Now, we are calculating dy\_dx which is nothing but the derivative of y with respect to x. This is done by calling the gradient function on y and x. The error occurs while calculating dy\_dx.

**Reason :** While calculating dz\_dx, we didn't face any error. But, while calculating the dy\_dx, we got a runtime error. The statement seems logically correct. But, the error occurred due to how we define the GradientTape. By default, persistent is set to False in the GradientTape function. This means, we can only calculate the gradient once with respect to the x variable. If we want to compute it again or multiple times , then we need to set the persistent parameter as True.



The screenshot shows a Jupyter Notebook cell with the following code:

```
1001     if self._recording:
1002         raise RuntimeError("GradientTape.gradient can only be called once on non-persistent tapes.")

[1]: x = tf.constant(3.0)
      with tf.GradientTape(persistent=True) as g:
      1      g.watch(x)
      2      y = x * x
      3      z = y * y
      4      dz_dx = g.gradient(z, x) # 108.0 (4*x^3 at x = 3)
      5      dy_dx = g.gradient(y, x) # 6.0
      6
      7      print(dz_dx,dy_dx)
      8
      9
      10
      11
      12 del g # Drop the reference to the tape
```

The output of the cell is:

```
[1]: tf.Tensor(108.0, shape=(), dtype=float32) tf.Tensor(6.0, shape=(), dtype=float32)
```

Timestamp : 31:05

We can see from the above snapshot that there is no error raised.

del g : It simply deletes the reference g.

**Can we compute the gradient with respect to multiple variables ?**

```
11
[ ] 12 del g # Drop the reference to the tape
[ ] tf.Tensor(108.0, shape=(), dtype=float32) tf.Tensor(6.0, shape=(), dtype=float32)
[ ] 1 # gradient w.r.t multiple variables
2
3 x = tf.constant(3.0)
4 with tf.GradientTape(persistent=True) as g:
5     g.watch(x)
6     y = x * x
7     z = y * y
8     dz_dx_dy = g.gradient(z, [x,y])
9
10 print(x)
11 print(y)
12 print(dz_dx_dy)
13
[ ] tf.Tensor(3.0, shape=(), dtype=float32)
tf.Tensor(9.0, shape=(), dtype=float32)
[<tf.Tensor: shape=(), dtype=float32, numpy=108.0>, <tf.Tensor: shape=(), dtype=float32, numpy=18.0>]
```

Timestamp : 32:50

We only need to focus on the 8th line. Every other line is already explained in detail.

Rather than giving a single variable, we can give a list of variables. The function will calculate the gradient with respect to each of those variables in the list and returns the values in a list.

```
1 # gradient w.r.t multiple variables
2
3 x = tf.constant(3.0)
4 y = tf.constant(4.0)
5 with tf.GradientTape(persistent=True) as g:
6     g.watch(x)
7     g.watch(y)
8     z = y * y + x * x
9     dz_dx_dy = g.gradient(z, [x,y])
10
11 print(x)
12 print(y)
13 print(dz_dx_dy)
14
```

z =  $y^2 + x^2$

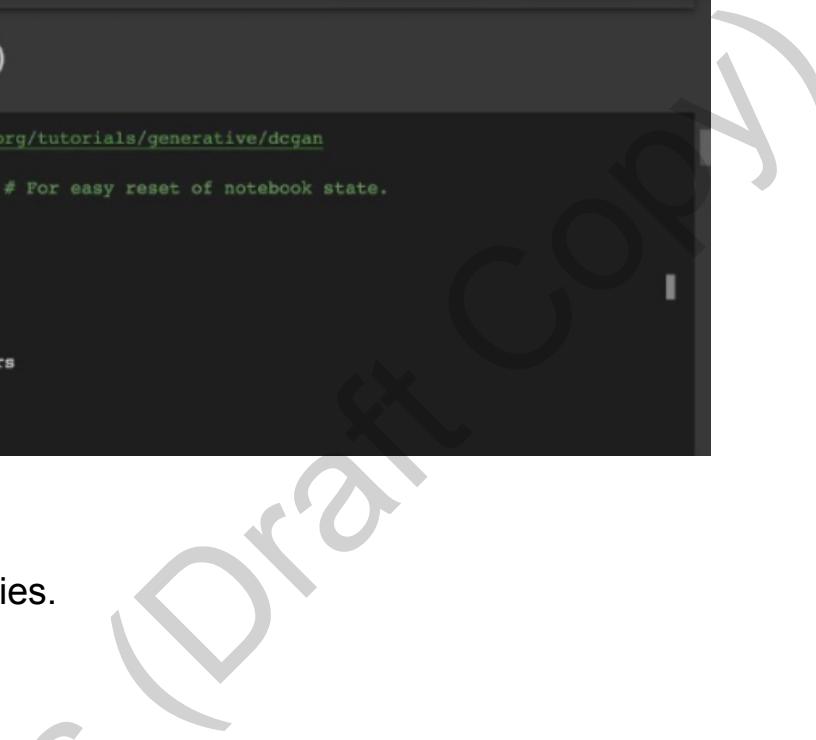
$\frac{\partial z}{\partial x}$

Timestamp : 37:54

Now, we can even get the derivative for the multivariable function. Here, both x and y are variables and it's being watched. Everything else remains the same.

Next, we will discuss Deep Convolutional Generative Adversarial Networks.

## Deep Convolutional Generative Adversarial Networks

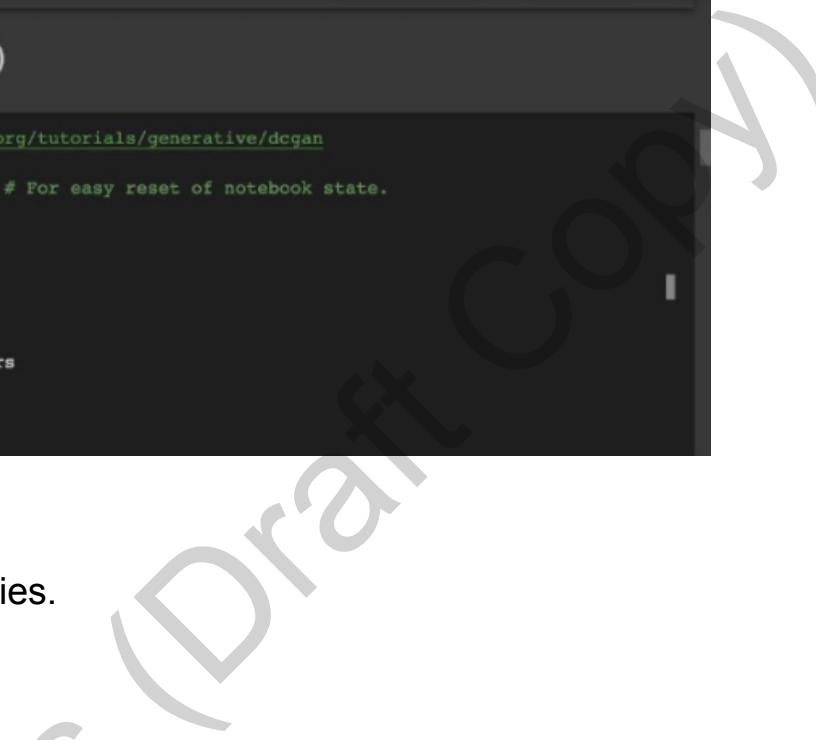


```
[ ] 1 # Source: https://www.tensorflow.org/tutorials/generative/dcgan
2
3 tf.keras.backend.clear_session() # For easy reset of notebook state.
4
5 import glob
6 import matplotlib.pyplot as plt
7 import numpy as np
8 import os
9 import PIL
10 from tensorflow.keras import layers
11 import time
12
13 from IPython import display
```

Timestamp : 39:11

As usual we import a set of libraries.

Then, we need to load the data.



```
[ ] 10 from tensorflow.keras import layers
[ ] 11 import time
[ ] 12
[ ] 13 from IPython import display

[ ] [1] # load all of MNIST dataset
[ ] 2 (train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()
[ ] 3 print(train_images.shape)

[ ] Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
[ ] 11493376/11490434 [=====] - 0s 0us/step
[ ] (60000, 28, 28)

[ ] 1 # Standardize image data
[ ] 2 train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).astype('float32')
[ ] 3 print(train_images.shape)
[ ]
[ ] 5 # [-1, 1] normalization
[ ] 6 train_images = (train_images - 127.5) / 127.5 # Normalize the images to [-1, 1]
```

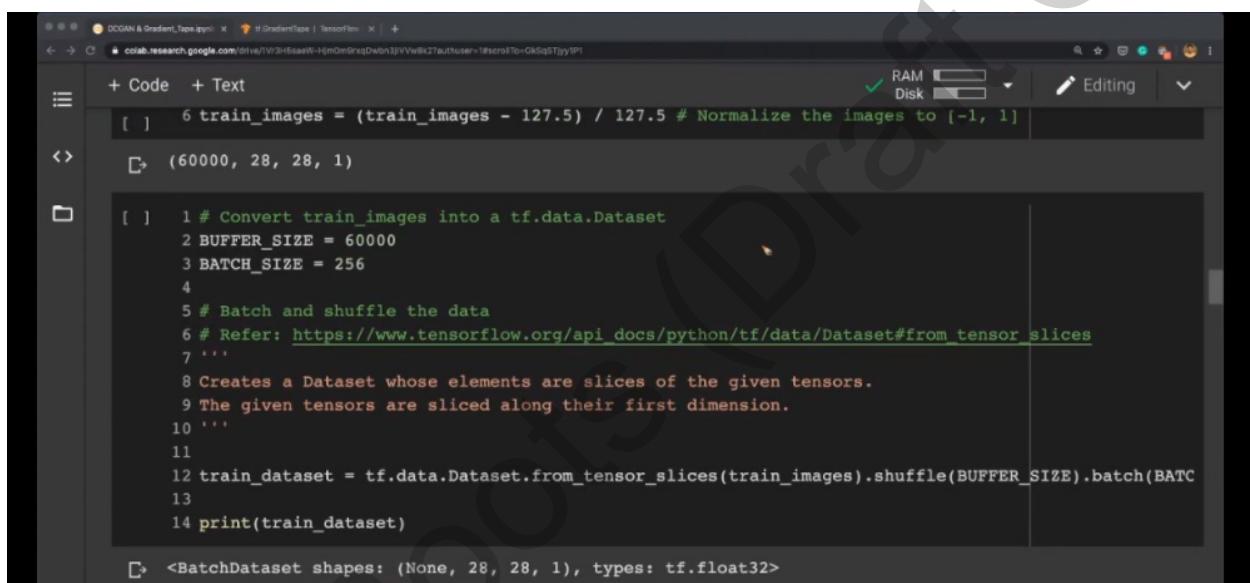
Timestamp : 39:17

We can load the data using the function `load_data()` in the `mnist` class.

We are going to normalize all the images in the range  $[-1, 1]$ . Originally, the values were in the range  $[0, 255]$ .

Also, we are reshaping the image to  $(60000, 28, 28, 1)$ . We are explicitly mentioning it as a gray scale image.

Now, we are going to define the dataset.



```
6 train_images = (train_images - 127.5) / 127.5 # Normalize the images to [-1, 1]
7
8 train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
9
10 print(train_dataset)

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

Timestamp : 43:32

Using the function `from_tensor_slices` from the `Dataset` class, we can transform those numpy arrays to a tensorflow dataset object. Now, we are going to shuffle the images with `buffer_size` as the parameter. Since in this case `buffer_size` equals the dataset size, it shuffles the entire dataset. If we set the `buffer_size` to be less than the dataset size, then it randomly picks `buffer_size` points and shuffles it. Then, we are specifying the `batch_size` by calling the `batch` function.

Next, we will define the generator network.

```
[ ] 1 def make_generator_model():
2     model = tf.keras.Sequential()
3     model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
4     model.add(layers.BatchNormalization())
5     model.add(layers.LeakyReLU())
6
7     model.add(layers.Reshape((7, 7, 256)))
8
9     # assert is used to debug
10    assert model.output_shape == (None, 7, 7, 256) # Note: None is the batch size
11
12    # Discussed in the GAN session
13    # Refer: https://medium.com/@vaibhavshukla182/why-do-we-need-conv2d-transpose-2534cd2a4d98
14    # Conv2DTranspose <=> DeConvolution
15    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
16    assert model.output_shape == (None, 7, 7, 128)
17    model.add(layers.BatchNormalization())
18    model.add(layers.LeakyReLU())
19
20    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
21    assert model.output_shape == (None, 14, 14, 64)
```

Timestamp : 51:40

For the first Dense layer, the input shape is 100. This is because the dimension of the noise vector which is drawn from the normal distribution is 100.

As usual, we add batch normalization and LeakyReLU as the activation function.

Then ,we reshape it to a (7,7,256) shape tensor.

The batch size specified for the generator network and the discriminator network could vary. This is because they are trained separately.

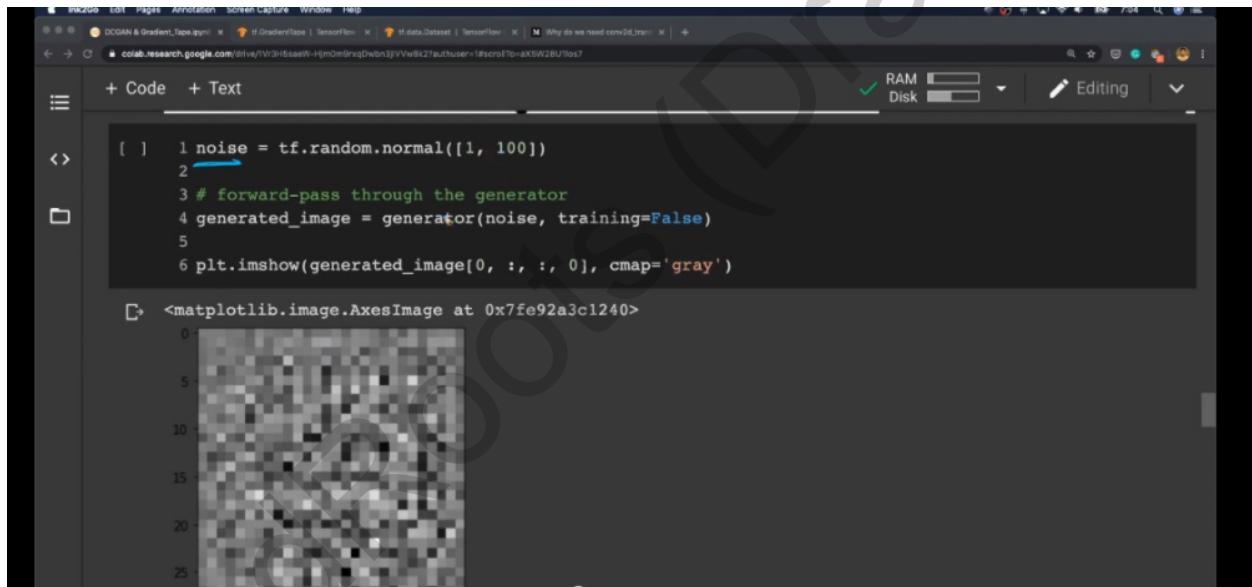
Then, we can check whether the shape of the model output is correct or not by defining an assert statement.\

Then, we are adding Conv2DTranspose layers which do the deconvolution or transposed convolution.

We are adding a sequence of such layers. In the context of generator networks, we need deconvolution or Conv2DTranspose as the layer. The reason is quite simple. If we want to generate something, then we start with fundamental things. Then, we build upon those. So, here basically , the fundamental brick is the noise vector. By building up, we mean putting it all together and generating. The deconvolution operation does it.

At the end, the output is None,28,28,1 which is the shape of the image. Here, None specifies the batch size.

Let's understand what actually the generator network generates.



```
[ ] 1 noise = tf.random.normal([1, 100])
2
3 # forward-pass through the generator
4 generated_image = generator(noise, training=False)
5
6 plt.imshow(generated_image[0, :, :, 0], cmap='gray')
```

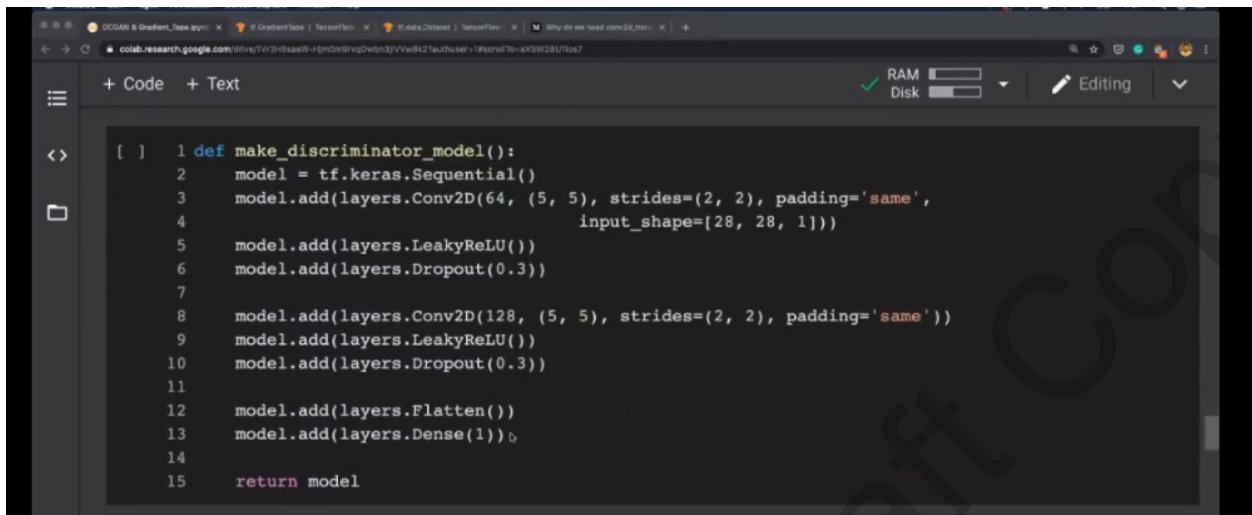
<matplotlib.image.AxesImage at 0x7fe92a3c1240>

Timestamp : 01:06:49

Now, the generator network is not trained. We specify it by setting the parameter training as False. So, it gives some random predictions.

We can see the noise image in the above snapshot. The input to the generator network is a random vector which is drawn from the normal distribution as mentioned earlier.

Next, we will define the discriminator network.



```
[ ] 1 def make_discriminator_model():
2     model = tf.keras.Sequential()
3     model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
4                           input_shape=[28, 28, 1]))
5     model.add(layers.LeakyReLU())
6     model.add(layers.Dropout(0.3))
7
8     model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
9     model.add(layers.LeakyReLU())
10    model.add(layers.Dropout(0.3))
11
12    model.add(layers.Flatten())
13    model.add(layers.Dense(1))
14
15    return model
```

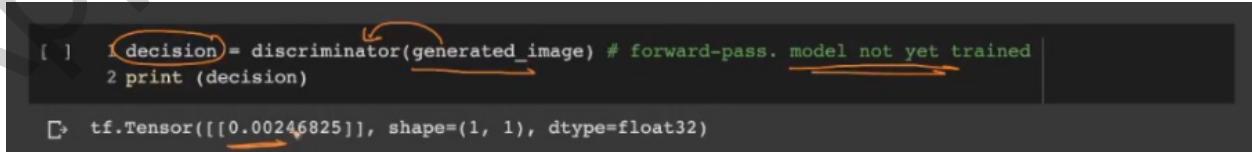
Timestamp : 01:10:04

Here, the input is the actual image of shape (None,28,28,1) and the output we want is the probability of the input image being original or the probability of the input image being drawn from the original data distribution. This means, if the input to it is an image generated by the generator network , then the output must be zero.

It's a sequence of Conv2D layers with dropout and LeakyReLU as the activation function.

In the final layer, we defined a Dense layer with output being a single neuron.

Now, let's pass the image generated by the generator network to the discriminator network.

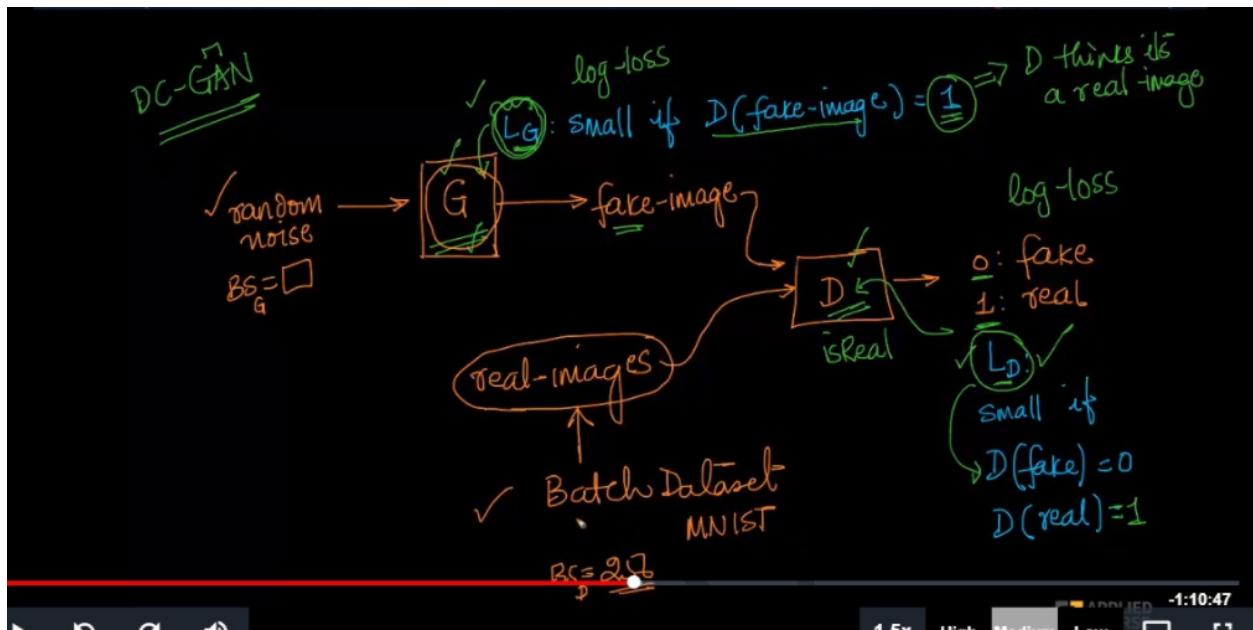


```
[ ] 1 decision = discriminator(generated_image) # forward-pass. model not yet trained
2 print (decision)

[ ] tf.Tensor([0.00246825], shape=(1, 1), dtype=float32)
```

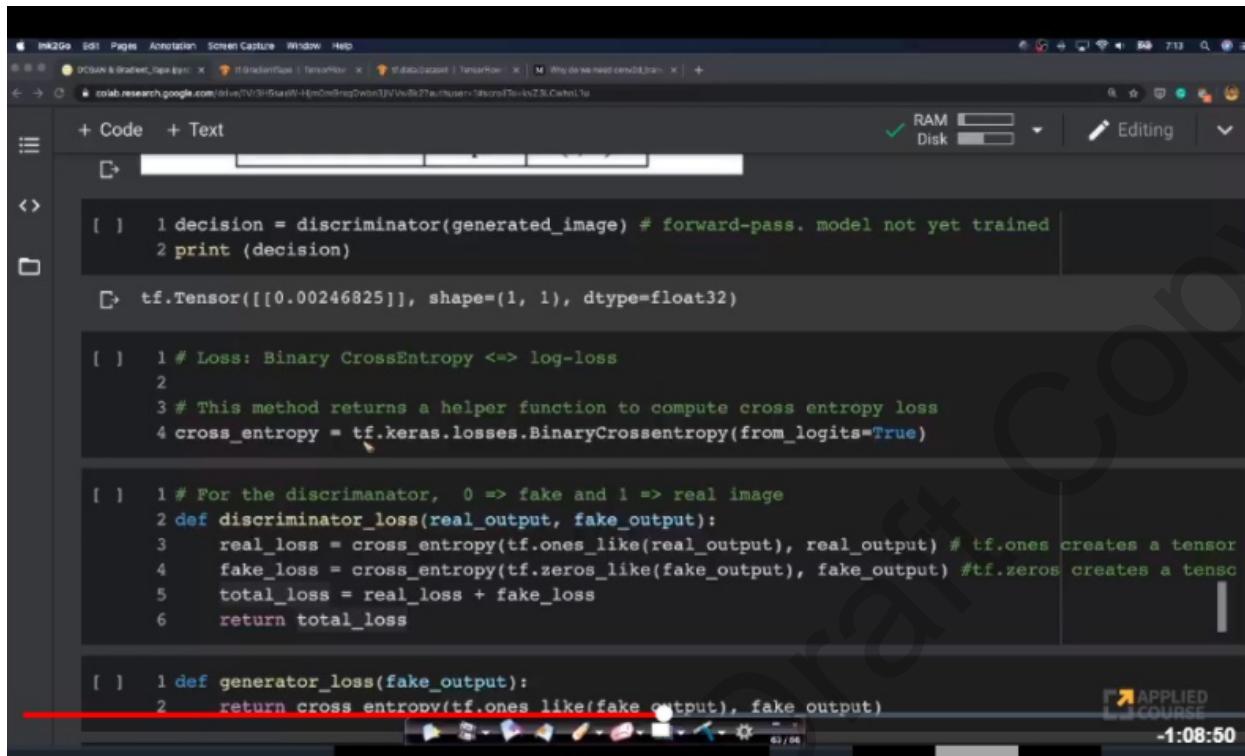
Timestamp : 01:13:24

In the above snapshot, we can see that the discriminator has produced the output.



Timestamp : 01:31:51

In the above snapshot, we can see the entire architecture of DCGAN.



```
[ ] 1 decision = discriminator(generated_image) # forward-pass. model not yet trained
[ ] 2 print (decision)
[ ] tf.Tensor([0.00246825]), shape=(1, 1), dtype=float32

[ ] 1 # Loss: Binary CrossEntropy <=> log-loss
[ ] 2
[ ] 3 # This method returns a helper function to compute cross entropy loss
[ ] 4 cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

[ ] 1 # For the discriminator, 0 => fake and 1 => real image
[ ] 2 def discriminator_loss(real_output, fake_output):
[ ] 3     real_loss = cross_entropy(tf.ones_like(real_output), real_output) # tf.ones creates a tensor
[ ] 4     fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output) #tf.zeros creates a tensor
[ ] 5     total_loss = real_loss + fake_loss
[ ] 6     return total_loss

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

Timestamp : 01:15:48

Next, we will define the loss as binary cross entropy. Here, we are passing True to from\_logits parameter. This is because, in the discriminator network, we didn't use the sigmoid activation function.

We will define the function called discriminator\_loss.

It takes two parameters i.e, real\_output and fake\_output. Here, real\_output is the list of values that the discriminator network took while it took the real images as the input. The fake\_output is the list of values that the discriminator network took while it took the fake images i.e, the images generated by the generator network as the input.

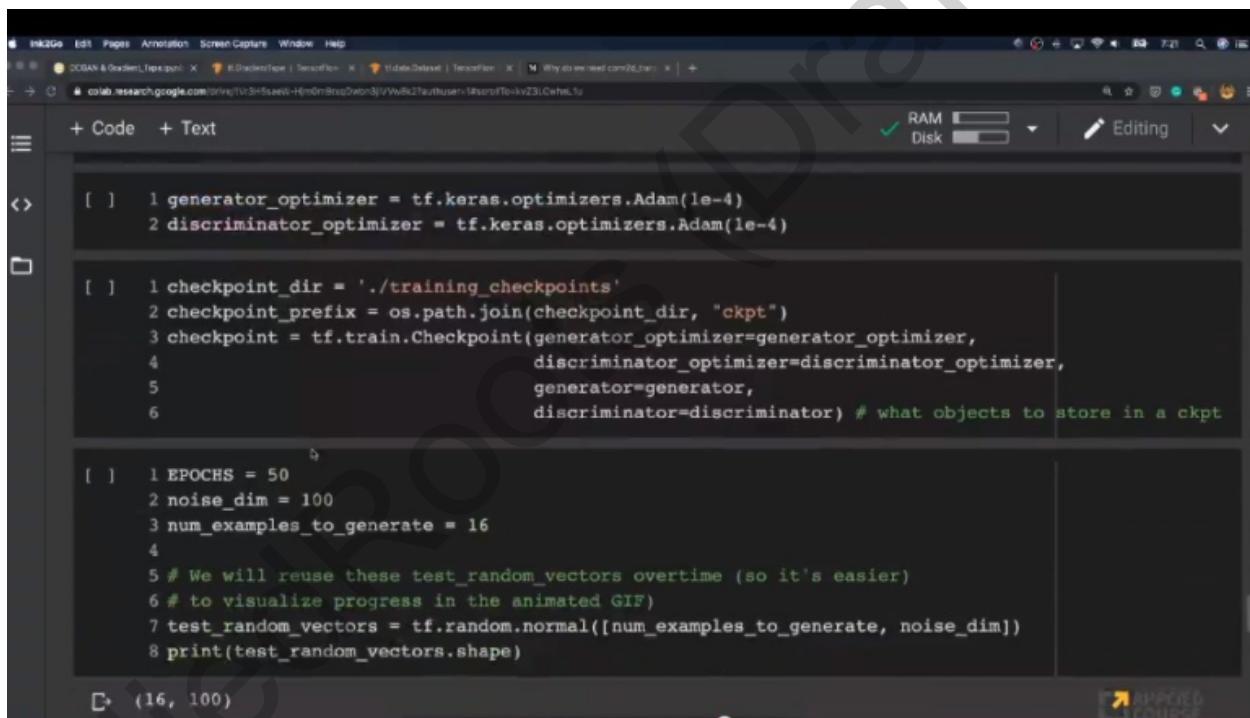
## The procedure for computing the loss is easy to understand.

- 1.) First, we compare the real\_output and the vector of ones. This is because we want the real\_output to be a vector of ones. Comparing means, computing the binary cross entropy loss between them.

- 2.) Next, we compare the fake\_output and the vector of zeros. This is because we want the fake\_output to be a vector of zeros.
- 3.) Then, we add the total loss and return it.

Now, we will define the function generator\_loss which takes the fake\_output as the parameter.

The job of the generator is to fool the discriminator. So, we should incorporate this while computing the loss. Here, we are comparing the fake\_output with ones. This is because, ideally, we want it to be a vector of ones.



```
[ ] 1 generator_optimizer = tf.keras.optimizers.Adam(1e-4)
2 discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

[ ] 1 checkpoint_dir = './training_checkpoints'
2 checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
3 checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
4                                 discriminator_optimizer=discriminator_optimizer,
5                                 generator=generator,
6                                 discriminator=discriminator) # what objects to store in a ckpt

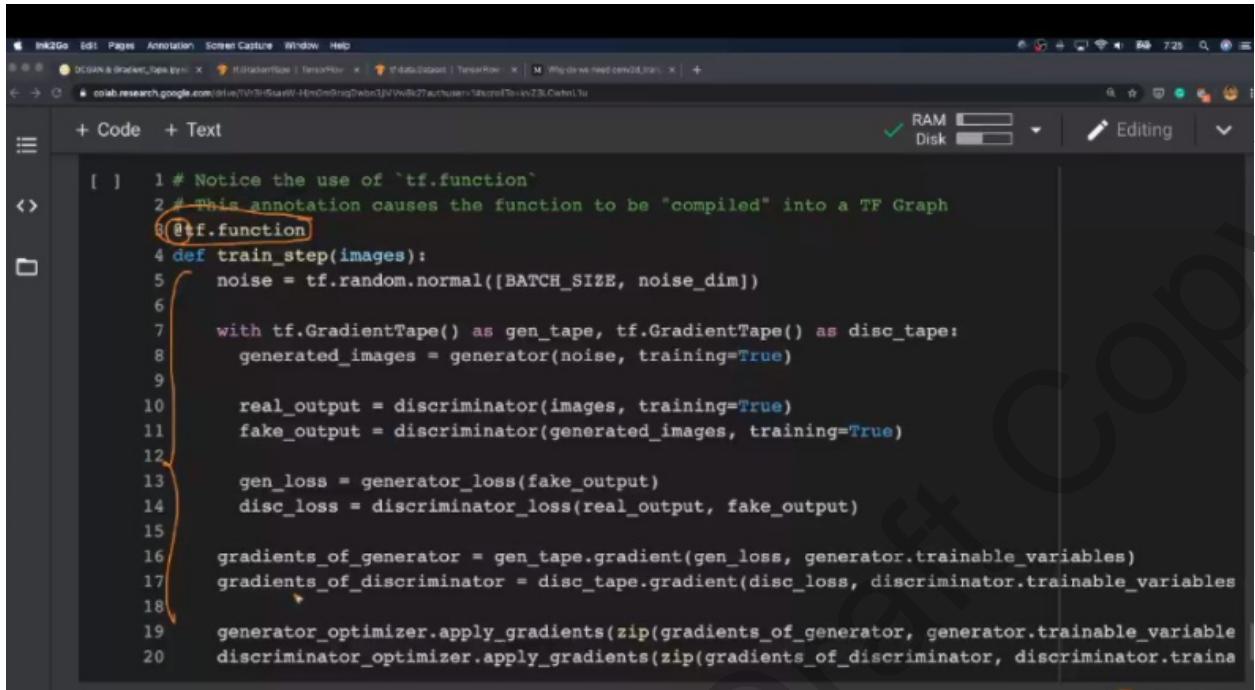
[ ] 1 EPOCHS = 50
2 noise_dim = 100
3 num_examples_to_generate = 16
4
5 # We will reuse these test_random_vectors overtime (so it's easier)
6 # to visualize progress in the animated GIF)
7 test_random_vectors = tf.random.normal([num_examples_to_generate, noise_dim])
8 print(test_random_vectors.shape)
```

Timestamp : 01:23:41

Then, we defined the optimizer and created the checkpoint.

At each epoch, we want to generate 16 examples. It is given as the input to the generator network.

The test\_random\_vectors hold those samples.



```
[ ] 1 # Notice the use of `tf.function`
2 # This annotation causes the function to be "compiled" into a TF Graph
3 @tf.function
4 def train_step(images):
5     noise = tf.random.normal([BATCH_SIZE, noise_dim])
6
7     with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
8         generated_images = generator(noise, training=True)
9
10        real_output = discriminator(images, training=True)
11        fake_output = discriminator(generated_images, training=True)
12
13        gen_loss = generator_loss(fake_output)
14        disc_loss = discriminator_loss(real_output, fake_output)
15
16        gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
17        gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)
18
19        generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
20        discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))
```

Timestamp : 01:28:15

**@tf.function** : This causes the function to be compiled into a tensorflow graph. This means, we can do backpropagation here since it's a computational graph.

Next, we define the train\_step function which takes in images which are real as the parameter.

Then, we are creating the noise tensor with shape (256,100).

We are creating two gradient tapes namely gen\_tape and disc\_tape.

Then, we pass the noise vector to the generator network.

real\_output contains the output after passing the real images to the discriminator network.

fake\_output contains the output after passing the generated images to the discriminator network.

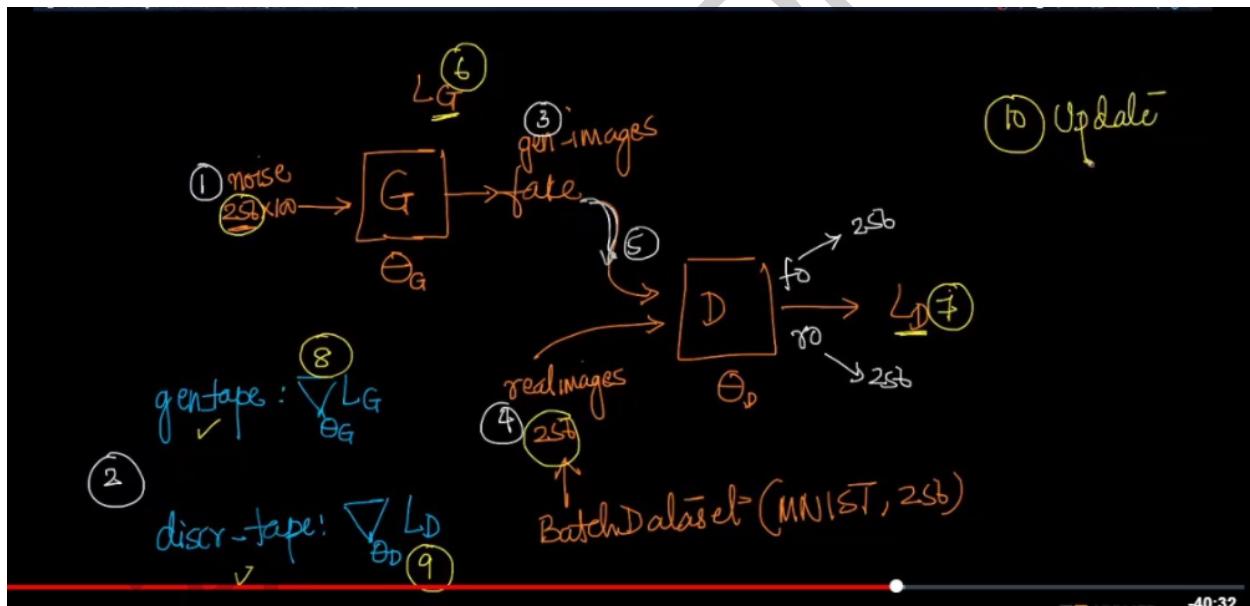
Then, we calculate both the loss based on the generator network and the discriminator network.

We can now calculate the gradients of the loss function with respect to the trainable parameters in both the discriminator and the generator network.

Then, using the function `apply_gradients` in the optimizer object we can update the trainable parameters.

Using the zip function, we can make a correspondence between the trainable parameters and its gradient.

## Overview :



Timestamp : 01:44:06

```
[ ] 1 def train(dataset, epochs):
2     for epoch in range(epochs):
3         start = time.time()
4
5         for image_batch in dataset:
6             train_step(image_batch)
7
8         # Produce images for the GIF as we go
9         display.clear_output(wait=True)
10        generate_and_save_images(generator,
11                                  epoch + 1,
12                                  test_random_vectors)
13
14        # Save the model every 15 epochs
15        if (epoch + 1) % 15 == 0:
16            checkpoint.save(file_prefix = checkpoint_prefix)
17
18        print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))
19
```

Timestamp : 01:44:30

This function takes in two input parameters which are the dataset and the epochs.

For each image in the dataset, we are calling the `train_step` function on it. This does the entire process till the update procedure.

The `generate_and_save_images` function passes the noise vector to the generator network and saves the output image to the local disk.

For every 15 epochs, we are storing the checkpoints.



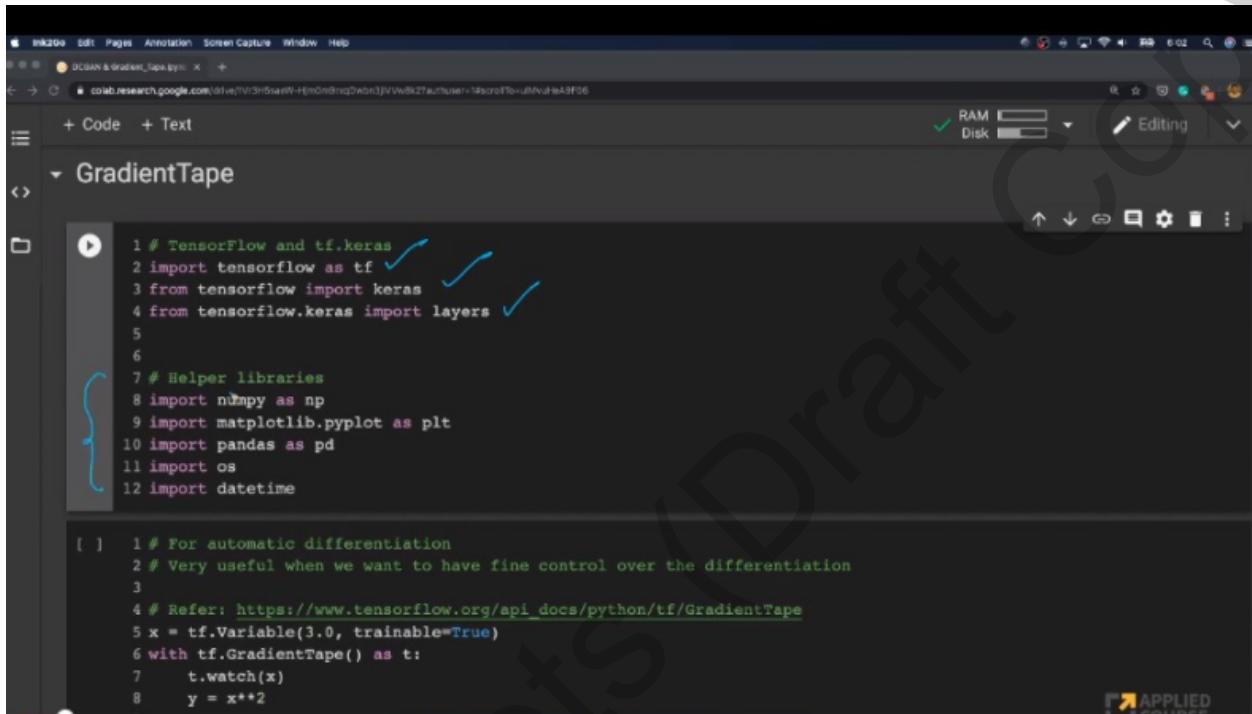
```
+ Code + Text
[ ] 24          test_random_vectors)

[ ] 1 def generate_and_save_images(model, epoch, test_input):
2   # Notice `training` is set to False.
3   # This is so all layers run in inference mode (batchnorm).
4   predictions = model(test_input, training=False)
5
6   fig = plt.figure(figsize=(4,4))
7
8   for i in range(predictions.shape[0]):
9     plt.subplot(4, 4, i+1)
10    plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
11    plt.axis('off')
12
13 plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
14 plt.show()
```

Timestamp : 01:47:30

It simply plots the predictions of the generator network as an image. It also saves it at the end. We set training as False. While computing the predictions, we don't want the model to train.

## 63.2 Code Walkthrough: DC-GANs and GradientTape in TensorFlow 2



```
1 # TensorFlow and tf.keras
2 import tensorflow as tf
3 from tensorflow import keras
4 from tensorflow.keras import layers
5
6
7 # Helper libraries
8 import numpy as np
9 import matplotlib.pyplot as plt
10 import pandas as pd
11 import os
12 import datetime
13
14 # For automatic differentiation
15 # Very useful when we want to have fine control over the differentiation
16
17 # Refer: https://www.tensorflow.org/api\_docs/python/tf/GradientTape
18 x = tf.Variable(3.0, trainable=True)
19 with tf.GradientTape() as t:
20     t.watch(x)
21     y = x**2
```

Timestamp: 05:37

As usual, we are importing a set of libraries which are already familiar to us.

**Gradient Tape :** It gives us a fine control over the gradients. In keras, while defining any layer, we only define the forward function. The operations needed to be done in the backward stage i.e, backpropagation are taken care of by the keras. It's called automatic differentiation.

```
1 # For automatic differentiation
2 # Very useful when we want to have fine control over the differentiation
3
4 # Refer: https://www.tensorflow.org/api\_docs/python/tf/GradientTape
5 x = tf.Variable(3.0, trainable=True)
6 with tf.GradientTape() as t:
7     t.watch(x)
8     y = x**2
9 print(t.gradient(y, x).numpy())
10
```

$y = x^2 = f(x)$

$\frac{dy}{dx} \Big|_{x=3} = 2x \Big|_{x=3}$

Timestamp : 19:48

- 1.) First, we are defining a tensorflow variable named x by calling `tf.Variable()`. We pass the value the variable needs to hold and we set the parameter `trainable` as `True`. This means, gradients can be calculated with respect to this variable. If it's set to `False`, then the gradients cannot be computed.
- 2.) Then, we are in the context of `tf.GradientTape` by using the `with` keyword.
- 3.) Then, we call the `watch` function on the variable x. This function will simply track the variable x for further gradient computations.
- 4.) Then, we define another variable y as `x**2`. This is the forward computation we were talking about.
- 5.) Then, we can call the function `gradient(y,x)` and get the numerical gradient of the variable y with respect to x.

This seems like a simple thing to do. But, with this we can perform complex computations too.

The screenshot shows a Jupyter Notebook interface with two code cells. The top cell contains Python code using TensorFlow's GradientTape to compute the second derivative of  $y = x^2$ . The bottom cell shows the resulting output: a tensor of value 6.0 and a tensor of value 2.0.

```
4 # Refer: https://www.tensorflow.org/api\_docs/python/tf/GradientTape
5 x = tf.Variable(3.0, trainable=True)
6 with tf.GradientTape() as t:
7     t.watch(x)
8     y = x**2
9
10 print(t.gradient(y, x).numpy())
11
12
13 x = tf.constant(3.0)
14 with tf.GradientTape() as g:
15     g.watch(x)
16     with tf.GradientTape() as gg:
17         gg.watch(x)
18         y = x * x
19         dy_dx = gg.gradient(y, x)      # Will compute to 6.0
20         d2y_dx2 = g.gradient(dy_dx, x) # Will compute to 2.0
21
22
23 print(dy_dx)
24 print(d2y_dx2)

tf.Tensor(6.0, shape=(), dtype=float32)
tf.Tensor(2.0, shape=(), dtype=float32)
```

Timestamp : 21:21

In the previous example, we did first order differentiation i.e, we computed the first order derivative of  $y$  with respect to  $x$ . Let's say we want to perform the second order differentiation i.e, computing the second order derivative of  $y$  with respect to  $x$ . How to do it?

- 1.) First, we will define  $x$  as a tensorflow constant with 3.0
- 2.) Then, we are in the context of Gradient Tape with  $g$  as the context variable.
- 3.) Then, we call the watch function on the variable  $x$ . Even Though,  $x$  is defined to be a constant, when we call `watch()` on this variable, we can compute the gradients with respect to this variable. Now, we open another context with  $gg$  as the context variable.
- 4.) Again, we call the `watch()` function on the variable  $x$  with  $gg$  as the context variable. Then, we define  $y$  as  $x^2$ .
- 5.) Now, we compute the  $dy/dx$  which is nothing but  $2*x$  and store it in the  $dy_dx$  variable. This is done by using the `gradient()` function on the  $x$  variable with  $gg$  as the context variable.

6.) Then, we compute the second order derivative by calling the gradient() function on dy\_dx and x . Note: dy\_dx holds 2\*x. The second order derivative is nothing but the constant 2.

Let's see another example.

The screenshot shows a Jupyter Notebook interface with a code cell containing Python and TensorFlow code. The code defines a tensor `x`, creates a GradientTape context `g`, calculates intermediate tensors `y` and `z`, and then attempts to calculate the gradients `dz_dx` and `dy_dx`. A handwritten note on the right side of the screen shows the equations  $y = x^2$  and  $z = y^2$ .

```
8 d2y_dx2 = g.gradient(dy_dx, x) # Will compute to 2.0
9
10 print(dy_dx)
11 print(d2y_dx2)

[ ] 1 x = tf.constant(3.0)
2 with tf.GradientTape() as g:
3     g.watch(x)
4     y = x * x
5     z = y * y
6 dz_dx = g.gradient(z, x) # 108.0 (4*x^3 at x = 3)
7 dy_dx = g.gradient(y, x) # 6.0
8
9

-----  
RuntimeError Traceback (most recent call last)  
<ipython-input-5-907f055cfbc1> in <module>()
```

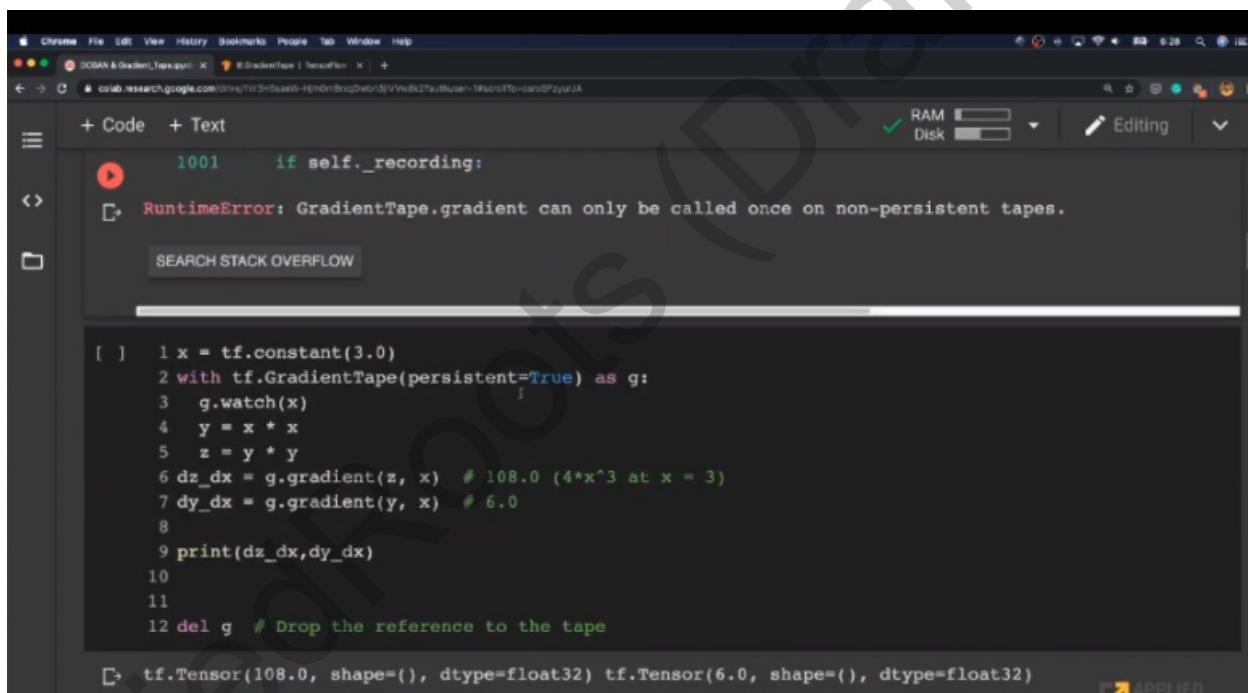
Timestamp : 27:32

Here, we are getting a runtime error. We will understand the reason behind it.

- 1.) First, we defined a tensorflow constant as x. Then, we stored the value 3 in it.
- 2.) Then, we are in the GradientTape context with the context variable as g.
- 3.) Now, we are calling the watch function on the variable .
- 4.) Now, we are defining y as x\*x and z as y\*y.
- 5.) Now, we are calculating dz\_dx which is nothing but the derivative of z with respect to x. This is done by calling the gradient function on z and x.

6.) Now, we are calculating `dy_dx` which is nothing but the derivative of `y` with respect to `x`. This is done by calling the `gradient` function on `y` and `x`. The error occurs while calculating `dy_dx`.

**Reason :** While calculating `dz_dx`, we didn't face any error. But, while calculating the `dy_dx`, we got a runtime error. The statement seems logically correct. But, the error occurred due to how we define the `GradientTape`. By default, `persistent` is set to `False` in the `GradientTape` function. This means, we can only calculate the gradient once with respect to the `x` variable. If we want to compute it again or multiple times , then we need to set the `persistent` parameter as `True`.



The screenshot shows a Jupyter Notebook cell with the following code:

```
1001     if self._recording:
1002         raise RuntimeError("GradientTape.gradient can only be called once on non-persistent tapes.")

[1]: x = tf.constant(3.0)
      with tf.GradientTape(persistent=True) as g:
      1      g.watch(x)
      2      y = x * x
      3      z = y * y
      4      dz_dx = g.gradient(z, x) # 108.0 (4*x^3 at x = 3)
      5      dy_dx = g.gradient(y, x) # 6.0
      6
      7      print(dz_dx,dy_dx)
      8
      9
      10
      11
      12 del g # Drop the reference to the tape
```

The output of the cell is:

```
[1]: tf.Tensor(108.0, shape=(), dtype=float32) tf.Tensor(6.0, shape=(), dtype=float32)
```

Timestamp : 31:05

We can see from the above snapshot that there is no error raised.

`del g` : It simply deletes the reference `g`.

## Can we compute the gradient with respect to multiple variables ?

```
 11
[ ] 12 del g # Drop the reference to the tape
[ ] tf.Tensor(108.0, shape=(), dtype=float32) tf.Tensor(6.0, shape=(), dtype=float32)
[ ] 1 # gradient w.r.t multiple variables
2
3 x = tf.constant(3.0)
4 with tf.GradientTape(persistent=True) as g:
5     g.watch(x)
6     y = x * x
7     z = y * y
8 dz_dx_dy = g.gradient(z, [x,y])
9
10 print(x)
11 print(y)
12 print(dz_dx_dy)
13
[ ] tf.Tensor(3.0, shape=(), dtype=float32)
tf.Tensor(9.0, shape=(), dtype=float32)
[<tf.Tensor: shape=(), dtype=float32, numpy=108.0>, <tf.Tensor: shape=(), dtype=float32, numpy=18.0>
```

Timestamp : 32:50

We only need to focus on the 8th line. Every other line is already explained in detail.

Rather than giving a single variable, we can give a list of variables. The function will calculate the gradient with respect to each of those variables in the list and returns the values in a list.

```
1 # gradient w.r.t multiple variables
2
3 x = tf.constant(3.0)
4 y = tf.constant(4.0)
5 with tf.GradientTape(persistent=True) as g:
6     g.watch(x)
7     g.watch(y)
8     z = y * y + x * x
9     dz_dx_dy = g.gradient(z, [x,y])
10
11 print(x)
12 print(y)
13 print(dz_dx_dy)
14
```

$$z = y^2 + x^2$$

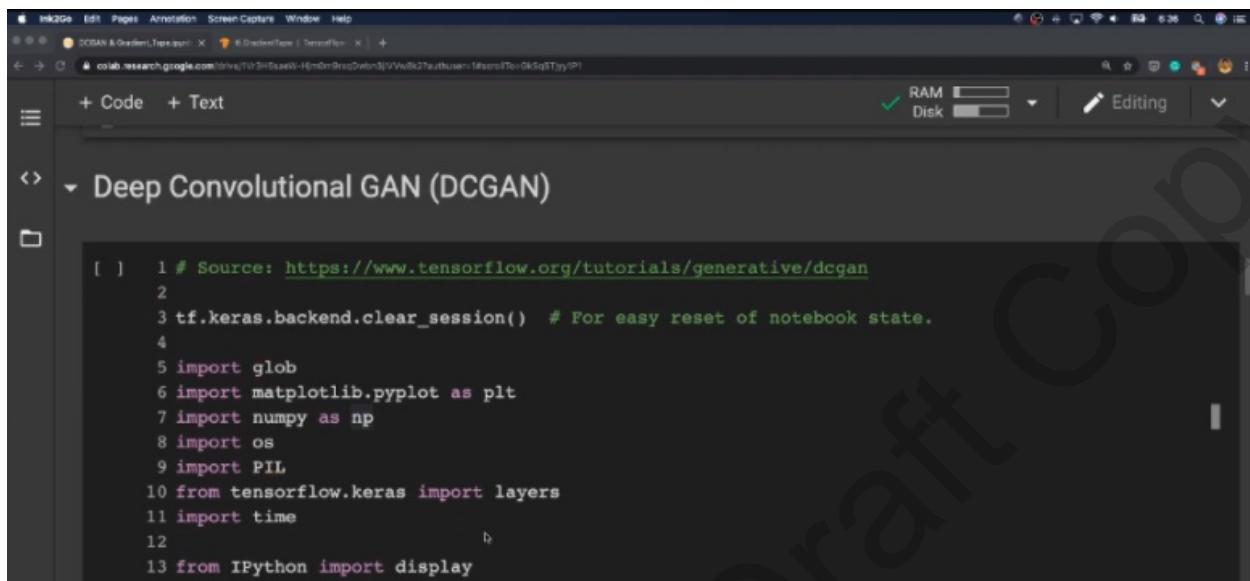
$$\frac{\partial z}{\partial x}$$

Timestamp : 37:54

Now, we can even get the derivative for the multivariable function. Here, both x and y are variables and it's being watched. Everything else remains the same.

Next, we will discuss Deep Convolutional Generative Adversarial Networks.

# Deep Convolutional Generative Adversarial Networks

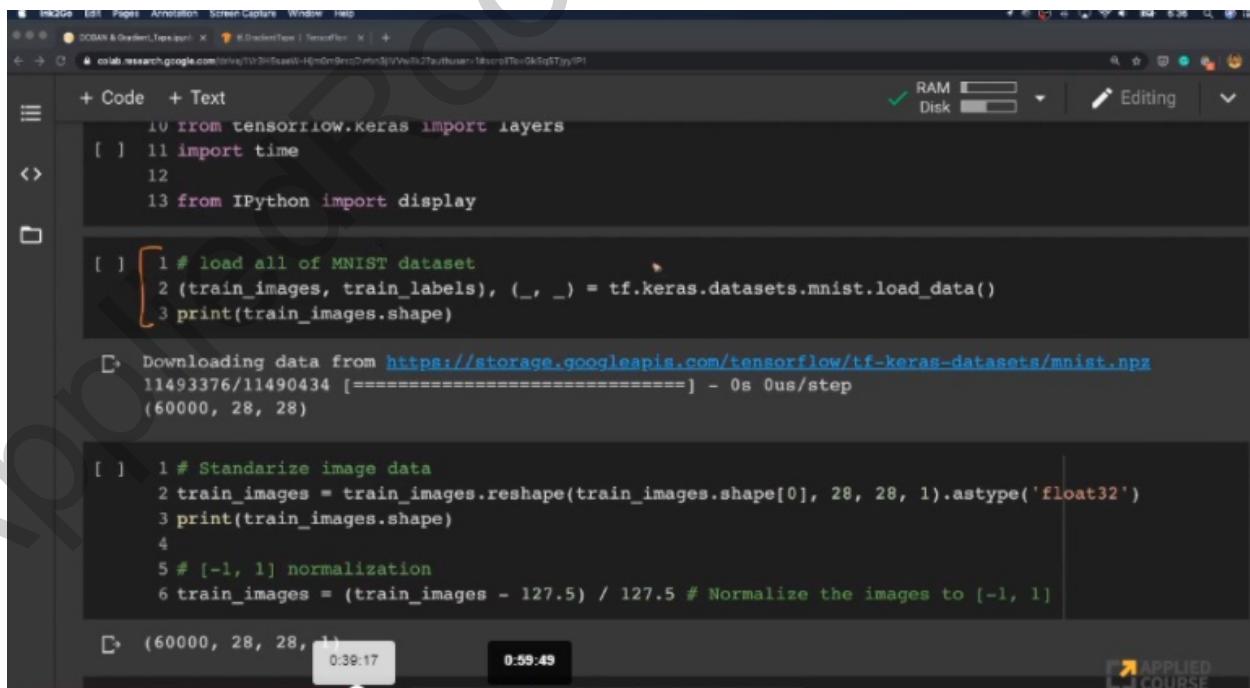


```
Ink2Go Edit Pages Annotation Screen Capture Window Help
DCGAN & GAN.ipynb X DCGAN.ipynb | Tensorflow X |
colab.research.google.com/drive/1V2HGeewW-HjmOn9rcqDvln3jVVwB27euthuser1dscofTeeOkSqlTgyIP
+ Code + Text RAM Disk Editing
Deep Convolutional GAN (DCGAN)
[ ] 1 # Source: https://www.tensorflow.org/tutorials/generative/dcgan
2
3 tf.keras.backend.clear_session() # For easy reset of notebook state.
4
5 import glob
6 import matplotlib.pyplot as plt
7 import numpy as np
8 import os
9 import PIL
10 from tensorflow.keras import layers
11 import time
12
13 from IPython import display
```

Timestamp : 39:11

As usual we import a set of libraries.

Then, we need to load the data.



```
Ink2Go Edit Pages Annotation Screen Capture Window Help
DCGAN & GAN.ipynb X DCGAN.ipynb | Tensorflow X |
colab.research.google.com/drive/1V2HGeewW-HjmOn9rcqDvln3jVVwB27euthuser1dscofTeeOkSqlTgyIP
+ Code + Text RAM Disk Editing
[ ] 10 from tensorflow.keras import layers
[ ] 11 import time
[ ] 12
[ ] 13 from IPython import display
[ ] [ 1 # load all of MNIST dataset
  2 (train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()
  3 print(train_images.shape)
  4
  5 # [-1, 1] normalization
  6 train_images = (train_images - 127.5) / 127.5 # Normalize the images to [-1, 1]
  7
  8 # Create the training data
  9 train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(60000).batch(32)
  10
  11 # Create the test data
  12 test_dataset = tf.data.Dataset.from_tensor_slices(train_images[60000:]).batch(32)
  13
  14 # Create the generator
  15 def make_generator_model():
  16     model = tf.keras.Sequential()
  17     model.add(layers.Dense(7 * 7 * 32, use_bias=False, input_shape=(100,)))
  18     model.add(layers.BatchNormalization())
  19     model.add(layers.ReLU())
  20
  21     model.add(layers.Reshape((7, 7, 32)))
  22     model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
  23     model.add(layers.BatchNormalization())
  24     model.add(layers.ReLU())
  25
  26     model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation='tanh'))
  27     return model
  28
  29 # Create the discriminator
  30 def make_discriminator_model():
  31     model = tf.keras.Sequential()
  32     model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same', input_shape=[28, 28, 1]))
  33     model.add(layers.BatchNormalization())
  34     model.add(layers.LeakyReLU())
  35
  36     model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
  37     model.add(layers.BatchNormalization())
  38     model.add(layers.LeakyReLU())
  39
  40     model.add(layers.Flatten())
  41     model.add(layers.Dense(1))
  42
  43     return model
  44
  45 # Create the generator and discriminator
  46 generator = make_generator_model()
  47 discriminator = make_discriminator_model()
  48
  49 # Create the loss functions
  50 cross_entropy = tf.keras.losses.BinaryCrossentropy()
  51
  52 # Create the optimizers
  53 generator_optimizer = tf.keras.optimizers.Adam(1e-4)
  54 discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
  55
  56 # Create the training loop
  57 @tf.function
  58 def train_step(images):
  59     # Generate fake images
  60     noise = tf.random.normal([32, 100])
  61     fake_images = generator(noise, training=True)
  62
  63     # Train the discriminator
  64     with tf.GradientTape() as tape:
  65         real_loss = discriminator(images, training=True)
  66         fake_loss = discriminator(fake_images, training=True)
  67         loss = tf.reduce_mean(real_loss) + tf.reduce_mean(fake_loss)
  68
  69     # Compute gradients
  70     gradients_of_discriminator = tape.gradient(loss, discriminator.trainable_variables)
  71     # Update weights
  72     discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))
  73
  74     # Train the generator
  75     with tf.GradientTape() as tape:
  76         fake_loss = discriminator(fake_images, training=True)
  77         loss = -tf.reduce_mean(fake_loss)
  78
  79     # Compute gradients
  80     gradients_of_generator = tape.gradient(loss, generator.trainable_variables)
  81     # Update weights
  82     generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
  83
  84     # Print training progress
  85     if step % 100 == 0:
  86         print(f"Step {step}, Generator Loss: {loss}, Discriminator Loss: {real_loss + fake_loss}")
  87
  88     # Save generated images
  89     if step % 1000 == 0:
  90         save_images(generator, step)
```

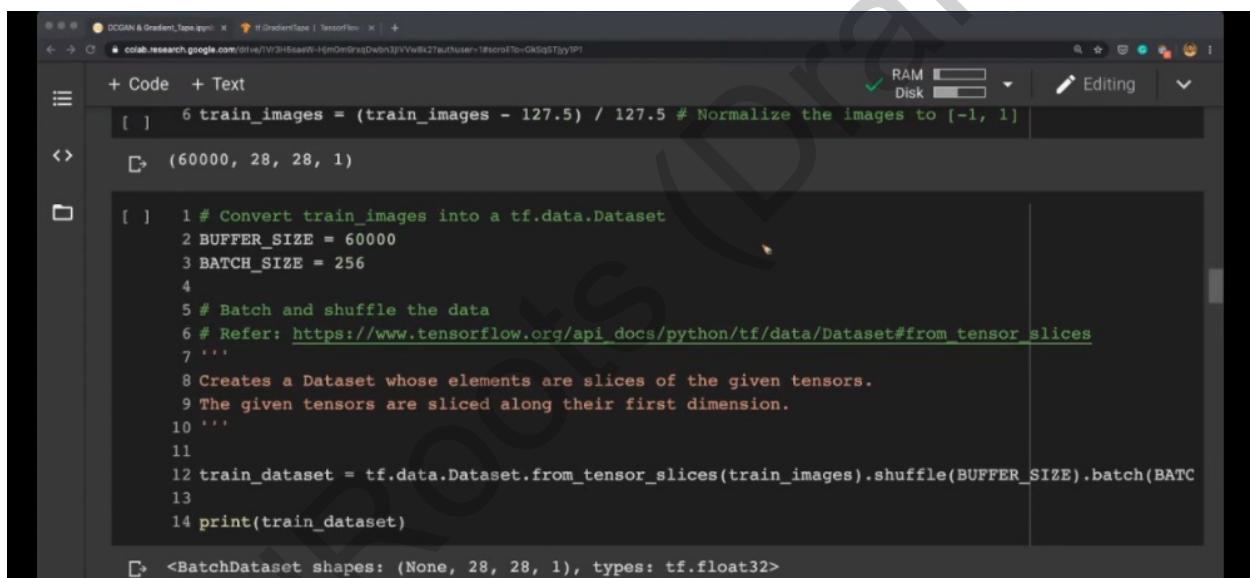
Timestamp : 39:17

We can load the data using the function `load_data()` in the `mnist` class.

We are going to normalize all the images in the range  $[-1, 1]$ . Originally , the values were in the range  $[0, 255]$ .

Also, we are reshaping the image to  $(60000, 28, 28, 1)$ . We are explicitly mentioning it as a gray scale image.

Now, we are going to define the dataset.



```
+ Code + Text
[ ] 6 train_images = (train_images - 127.5) / 127.5 # Normalize the images to [-1, 1]
[ ] ↴ (60000, 28, 28, 1)

[ ] 1 # Convert train_images into a tf.data.Dataset
[ ] 2 BUFFER_SIZE = 60000
[ ] 3 BATCH_SIZE = 256
[ ] 4
[ ] 5 # Batch and shuffle the data
[ ] 6 # Refer: https://www.tensorflow.org/api\_docs/python/tf/data/Dataset#from\_tensor\_slices
[ ] 7 '''
[ ] 8 Creates a Dataset whose elements are slices of the given tensors.
[ ] 9 The given tensors are sliced along their first dimension.
[ ] 10 '''
[ ] 11
[ ] 12 train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
[ ] 13
[ ] 14 print(train_dataset)
[ ] ↴ <BatchDataset shapes: (None, 28, 28, 1), types: tf.float32>
```

Timestamp : 43:32

Using the function `from_tensor_slices` from the `Dataset` class, we can transform those numpy arrays to a tensorflow dataset object. Now, we are going to shuffle the images with `buffer_size` as the parameter. Since in this case `buffer_size` equals the dataset size, it shuffles the entire dataset. If we set the `buffer_size` to be less than the dataset size, then it randomly picks `buffer_size` points and shuffles it. Then, we are specifying the `batch_size` by calling the `batch` function.

Next, we will define the generator network.

```
[ ] 1 def make_generator_model():
2     model = tf.keras.Sequential()
3     model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
4     model.add(layers.BatchNormalization())
5     model.add(layers.LeakyReLU())
6
7     model.add(layers.Reshape((7, 7, 256)))
8
9     # assert is used to debug
10    assert model.output_shape == (None, 7, 7, 256) # Note: None is the batch size
11
12    # Discussed in the GAN session
13    # Refer: https://medium.com/@vaibhavshukla182/why-do-we-need-conv2d-transpose-2534cd2a4d98
14    # Conv2DTranspose <=> DeConvolution
15    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
16    assert model.output_shape == (None, 7, 7, 128)
17    model.add(layers.BatchNormalization())
18    model.add(layers.LeakyReLU())
19
20    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
21    assert model.output_shape == (None, 14, 14, 64)
```

Timestamp : 51:40

For the first Dense layer, the input shape is 100. This is because the dimension of the noise vector which is drawn from the normal distribution is 100.

As usual, we add batch normalization and LeakyReLU as the activation function.

Then ,we reshape it to a (7,7,256) shape tensor.

The batch size specified for the generator network and the discriminator network could vary. This is because they are trained separately.

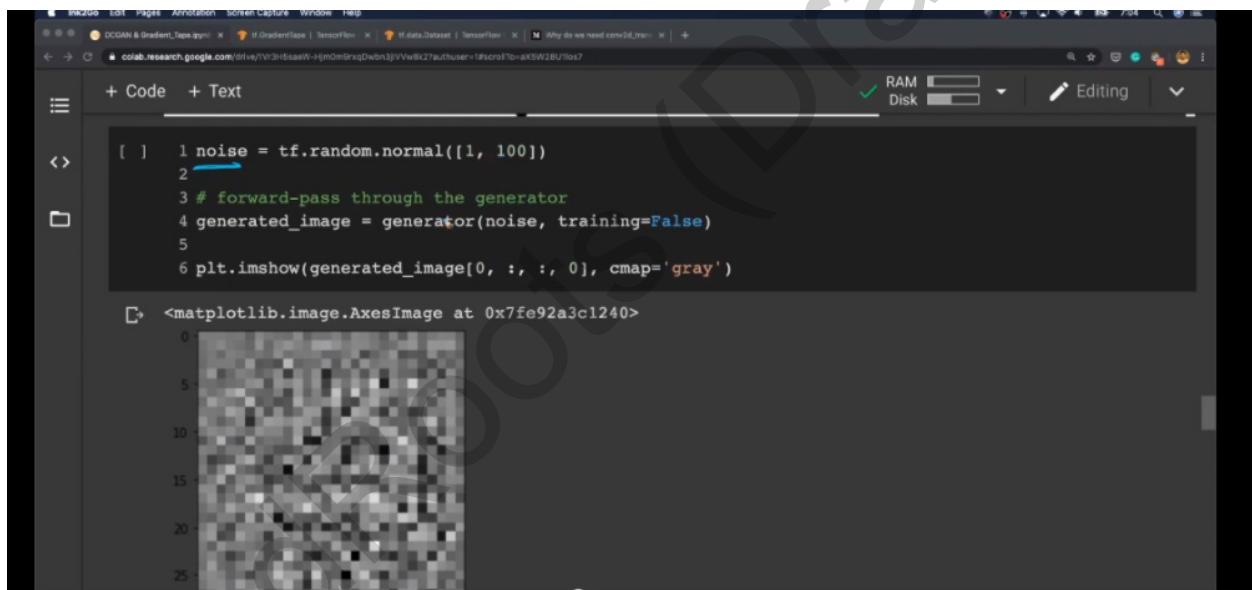
Then, we can check whether the shape of the model output is correct or not by defining an assert statement.\

Then, we are adding Conv2DTranspose layers which do the deconvolution or transposed convolution.

We are adding a sequence of such layers. In the context of generator networks, we need deconvolution or Conv2DTranspose as the layer. The reason is quite simple. If we want to generate something, then we start with fundamental things. Then, we build upon those. So, here basically , the fundamental brick is the noise vector. By building up, we mean putting it all together and generating. The deconvolution operation does it.

At the end, the output is None,28,28,1 which is the shape of the image. Here, None specifies the batch size.

Let's understand what actually the generator network generates.



```
[ ] 1 noise = tf.random.normal([1, 100])
2
3 # forward-pass through the generator
4 generated_image = generator(noise, training=False)
5
6 plt.imshow(generated_image[0, :, :, 0], cmap='gray')
```

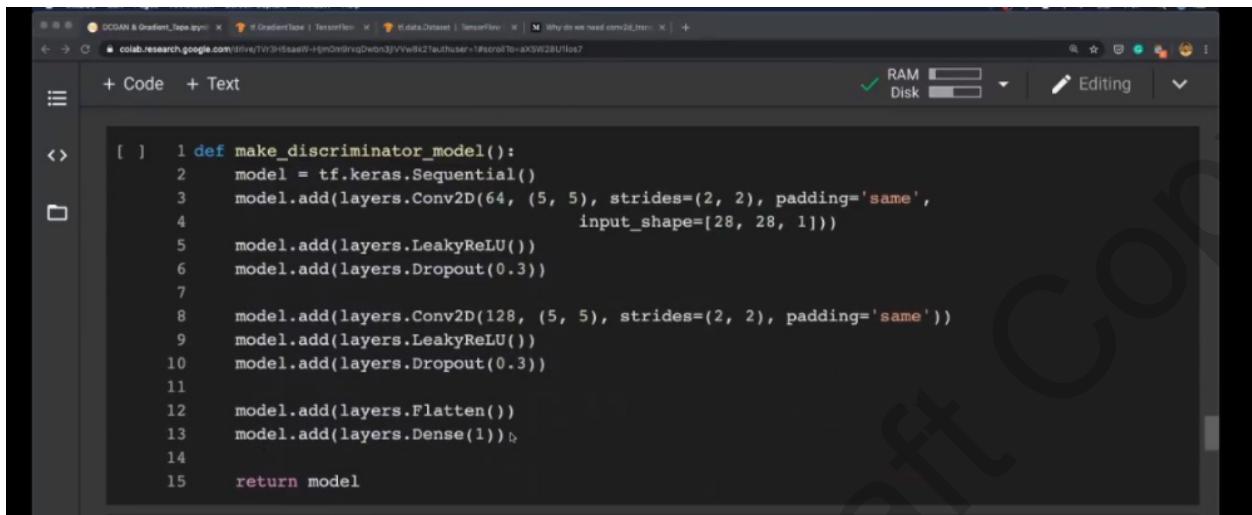
<matplotlib.image.AxesImage at 0x7fe92a3c1240>

Timestamp : 01:06:49

Now, the generator network is not trained. We specify it by setting the parameter training as False. So, it gives some random predictions.

We can see the noise image in the above snapshot. The input to the generator network is a random vector which is drawn from the normal distribution as mentioned earlier.

Next, we will define the discriminator network.



```
[ ] 1 def make_discriminator_model():
2     model = tf.keras.Sequential()
3     model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
4                           input_shape=[28, 28, 1]))
5     model.add(layers.LeakyReLU())
6     model.add(layers.Dropout(0.3))
7
8     model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
9     model.add(layers.LeakyReLU())
10    model.add(layers.Dropout(0.3))
11
12    model.add(layers.Flatten())
13    model.add(layers.Dense(1))
14
15    return model
```

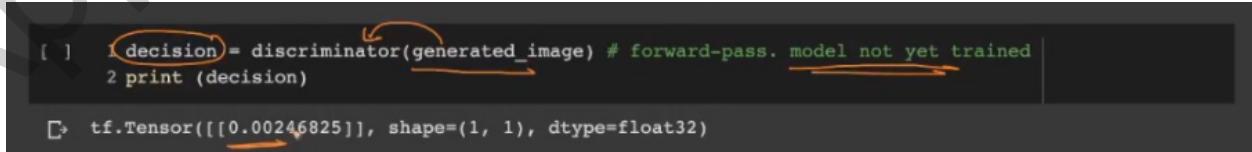
Timestamp : 01:10:04

Here, the input is the actual image of shape (None,28,28,1) and the output we want is the probability of the input image being original or the probability of the input image being drawn from the original data distribution. This means, if the input to it is an image generated by the generator network , then the output must be zero.

It's a sequence of Conv2D layers with dropout and LeakyReLU as the activation function.

In the final layer, we defined a Dense layer with output being a single neuron.

Now, let's pass the image generated by the generator network to the discriminator network.

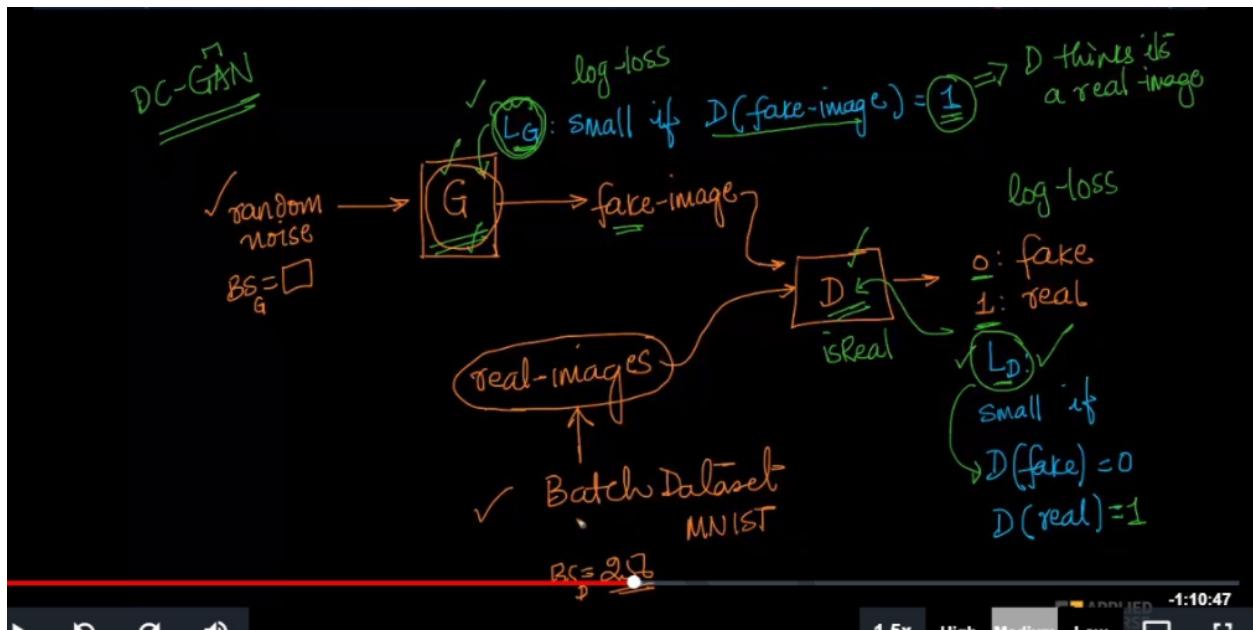


```
[ ] 1 decision = discriminator(generated_image) # forward-pass. model not yet trained
2 print (decision)

[ ] tf.Tensor([0.00246825], shape=(1, 1), dtype=float32)
```

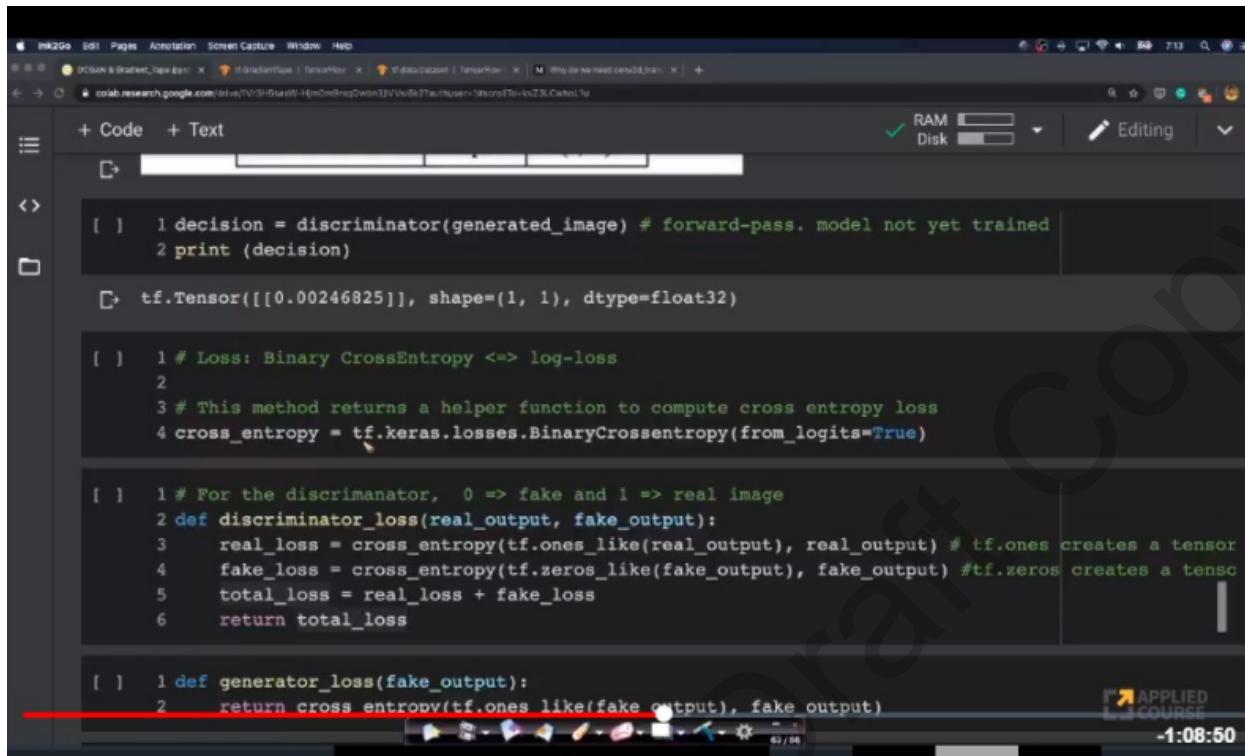
Timestamp : 01:13:24

In the above snapshot, we can see that the discriminator has produced the output.



Timestamp : 01:31:51

In the above snapshot, we can see the entire architecture of DCGAN.



```
[ ] 1 decision = discriminator(generated_image) # forward-pass. model not yet trained
[ ] 2 print (decision)
[ ] tf.Tensor([0.00246825]), shape=(1, 1), dtype=float32

[ ] 1 # Loss: Binary CrossEntropy <=> log-loss
[ ] 2
[ ] 3 # This method returns a helper function to compute cross entropy loss
[ ] 4 cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

[ ] 1 # For the discriminator, 0 => fake and 1 => real image
[ ] 2 def discriminator_loss(real_output, fake_output):
[ ] 3     real_loss = cross_entropy(tf.ones_like(real_output), real_output) # tf.ones creates a tensor
[ ] 4     fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output) #tf.zeros creates a tensor
[ ] 5     total_loss = real_loss + fake_loss
[ ] 6     return total_loss

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

Timestamp : 01:15:48

Next, we will define the loss as binary cross entropy. Here, we are passing True to from\_logits parameter. This is because, in the discriminator network, we didn't use the sigmoid activation function.

We will define the function called discriminator\_loss.

It takes two parameters i.e, real\_output and fake\_output. Here, real\_output is the list of values that the discriminator network took while it took the real images as the input. The fake\_output is the list of values that the discriminator network took while it took the fake images i.e, the images generated by the generator network as the input.

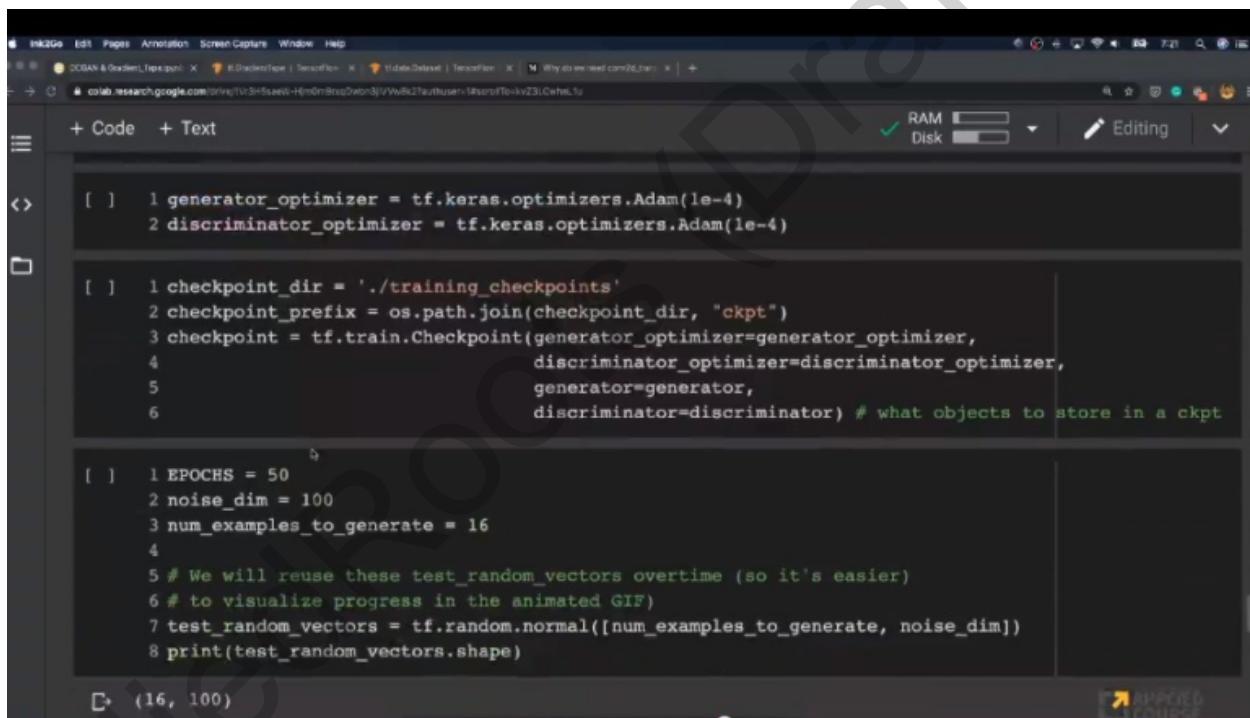
## The procedure for computing the loss is easy to understand.

- 1.) First, we compare the real\_output and the vector of ones. This is because we want the real\_output to be a vector of ones. Comparing means, computing the binary cross entropy loss between them.

- 2.) Next, we compare the fake\_output and the vector of zeros. This is because we want the fake\_output to be a vector of zeros.
- 3.) Then, we add the total loss and return it.

Now, we will define the function generator\_loss which takes the fake\_output as the parameter.

The job of the generator is to fool the discriminator. So, we should incorporate this while computing the loss. Here, we are comparing the fake\_output with ones. This is because, ideally, we want it to be a vector of ones.



```
[ ] 1 generator_optimizer = tf.keras.optimizers.Adam(1e-4)
2 discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

[ ] 1 checkpoint_dir = './training_checkpoints'
2 checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
3 checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
4                                 discriminator_optimizer=discriminator_optimizer,
5                                 generator=generator,
6                                 discriminator=discriminator) # what objects to store in a ckpt

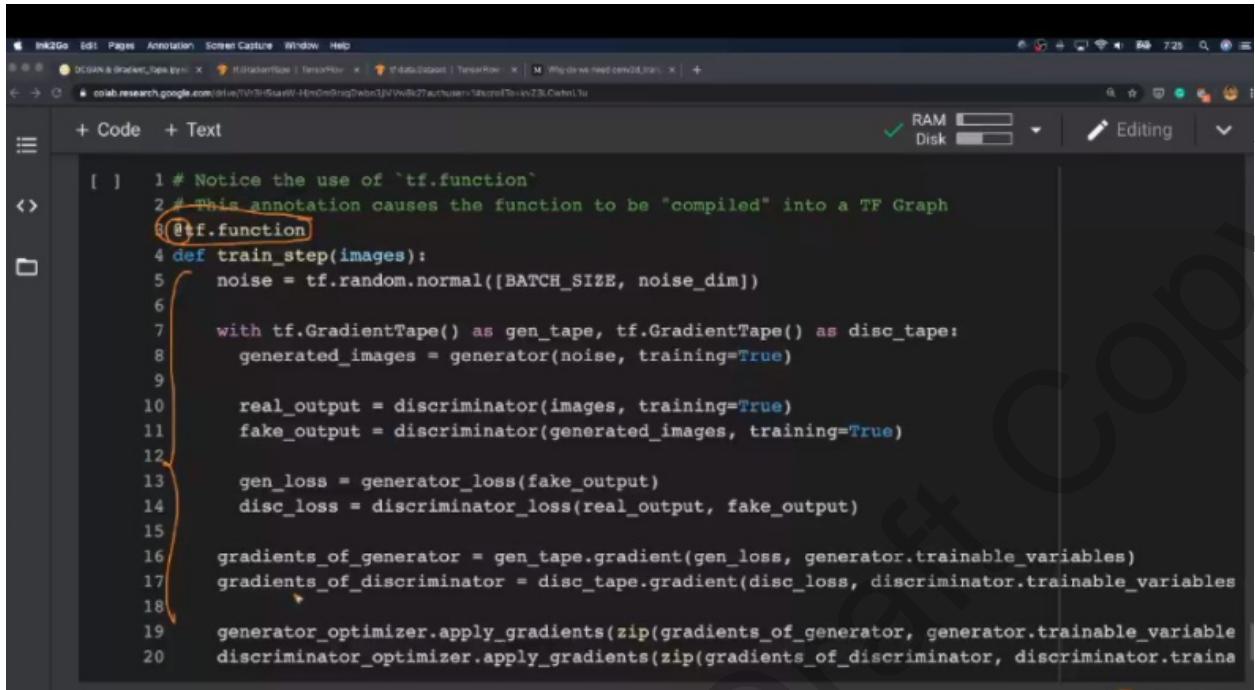
[ ] 1 EPOCHS = 50
2 noise_dim = 100
3 num_examples_to_generate = 16
4
5 # We will reuse these test_random_vectors overtime (so it's easier)
6 # to visualize progress in the animated GIF)
7 test_random_vectors = tf.random.normal([num_examples_to_generate, noise_dim])
8 print(test_random_vectors.shape)
```

Timestamp : 01:23:41

Then, we defined the optimizer and created the checkpoint.

At each epoch, we want to generate 16 examples. It is given as the input to the generator network.

The test\_random\_vectors hold those samples.



```
[ ] 1 # Notice the use of `tf.function`
2 # This annotation causes the function to be "compiled" into a TF Graph
3 @tf.function
4 def train_step(images):
5     noise = tf.random.normal([BATCH_SIZE, noise_dim])
6
7     with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
8         generated_images = generator(noise, training=True)
9
10        real_output = discriminator(images, training=True)
11        fake_output = discriminator(generated_images, training=True)
12
13        gen_loss = generator_loss(fake_output)
14        disc_loss = discriminator_loss(real_output, fake_output)
15
16        gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
17        gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)
18
19        generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
20        discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))
```

Timestamp : 01:28:15

**@tf.function** : This causes the function to be compiled into a tensorflow graph. This means, we can do backpropagation here since it's a computational graph.

Next, we define the train\_step function which takes in images which are real as the parameter.

Then, we are creating the noise tensor with shape (256,100).

We are creating two gradient tapes namely gen\_tape and disc\_tape.

Then, we pass the noise vector to the generator network.

real\_output contains the output after passing the real images to the discriminator network.

fake\_output contains the output after passing the generated images to the discriminator network.

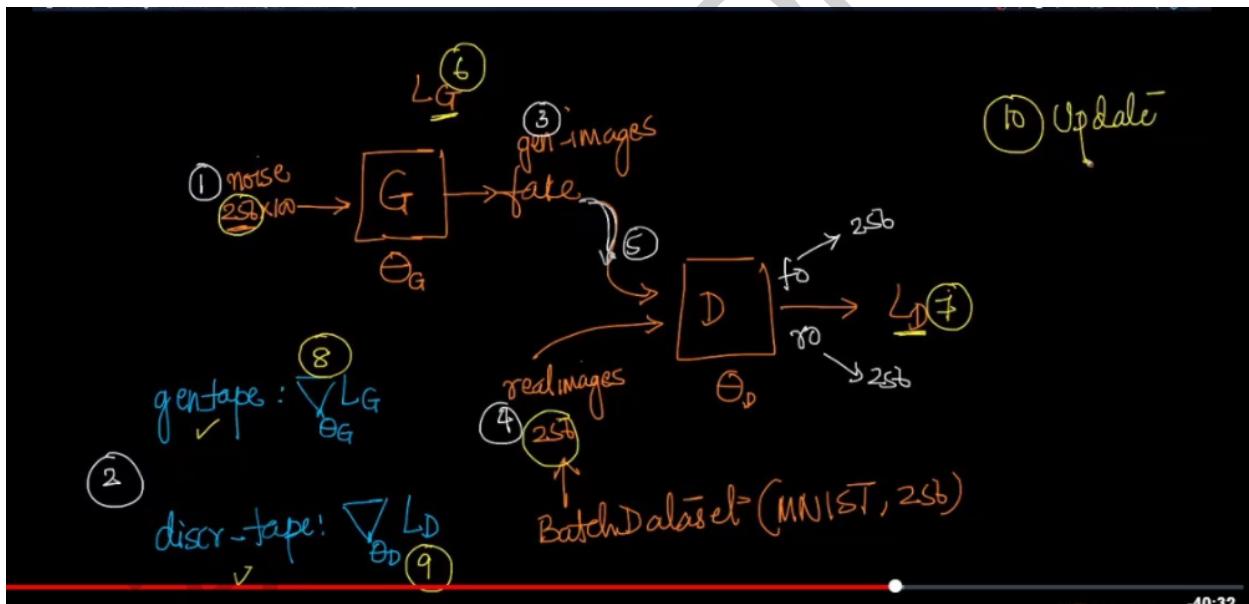
Then, we calculate both the loss based on the generator network and the discriminator network.

We can now calculate the gradients of the loss function with respect to the trainable parameters in both the discriminator and the generator network.

Then, using the function `apply_gradients` in the optimizer object we can update the trainable parameters.

Using the `zip` function, we can make a correspondence between the trainable parameters and its gradient.

## Overview :



Timestamp : 01:44:06

```
[ ] 1 def train(dataset, epochs):
2     for epoch in range(epochs):
3         start = time.time()
4
5         for image_batch in dataset:
6             train_step(image_batch)
7
8         # Produce images for the GIF as we go
9         display.clear_output(wait=True)
10        generate_and_save_images(generator,
11                                  epoch + 1,
12                                  test_random_vectors)
13
14        # Save the model every 15 epochs
15        if (epoch + 1) % 15 == 0:
16            checkpoint.save(file_prefix = checkpoint_prefix)
17
18        print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))
19
```

Timestamp : 01:44:30

This function takes in two input parameters which are the dataset and the epochs.

For each image in the dataset, we are calling the train\_step function on it. This does the entire process till the update procedure.

The generate\_and\_save\_images function passes the noise vector to the generator network and saves the output image to the local disk.

For every 15 epochs, we are storing the checkpoints.



```
+ Code + Text
[ ] 24          test_random_vectors)

[ ] 1 def generate_and_save_images(model, epoch, test_input):
2   # Notice `training` is set to False.
3   # This is so all layers run in inference mode (batchnorm).
4   predictions = model(test_input, training=False)
5
6   fig = plt.figure(figsize=(4,4))
7
8   for i in range(predictions.shape[0]):
9     plt.subplot(4, 4, i+1)
10    plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
11    plt.axis('off')
12
13 plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
14 plt.show()
```

Timestamp : 01:47:30

It simply plots the predictions of the generator network as an image. It also saves it at the end. We set training as False. While computing the predictions, we don't want the model to train.

AppliedRoots (Draft Copy)

AppliedRoots (Draft Copy)