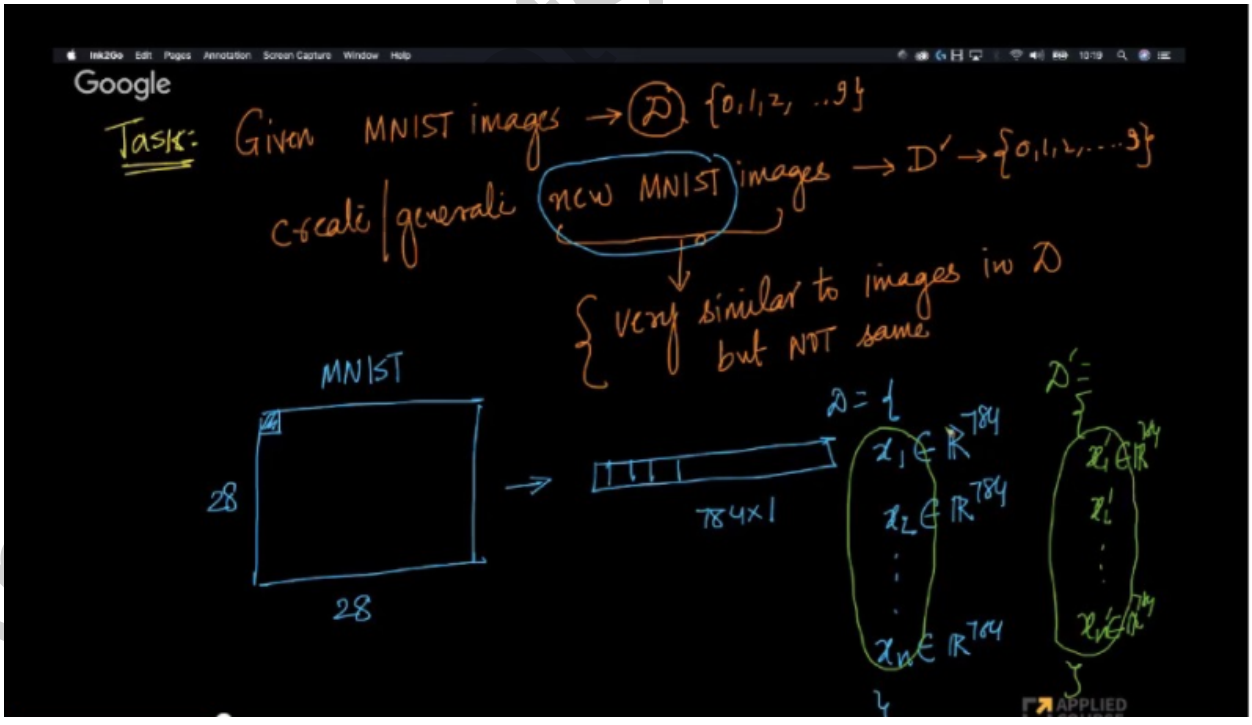
63.1 Generative Adversarial Networks (GAN)

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.



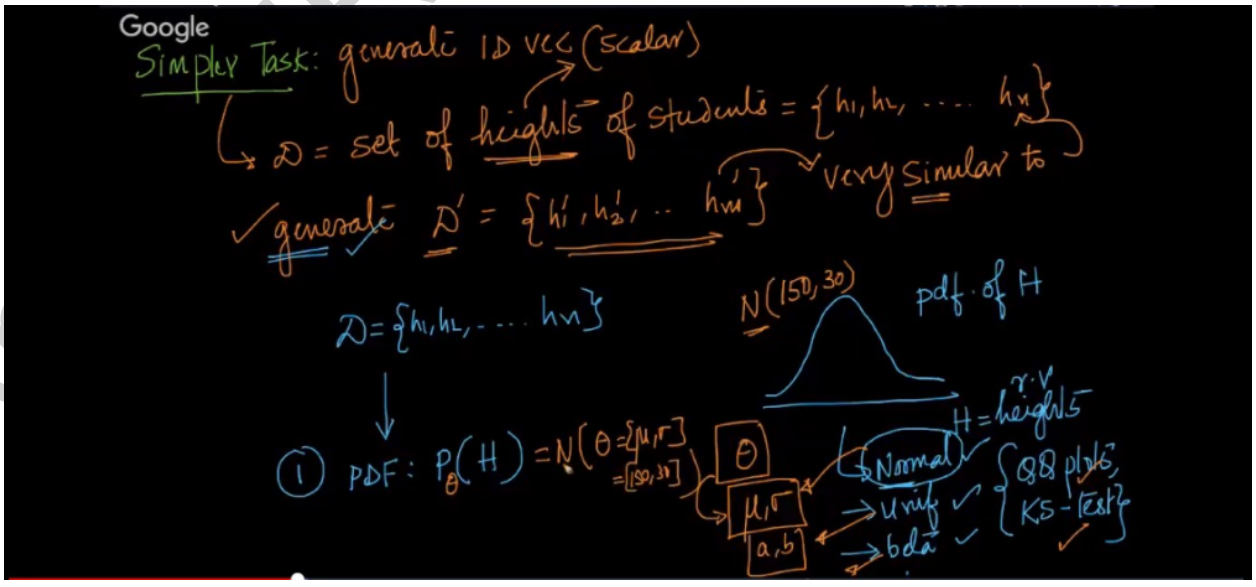
Timestamp : 21:50

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 xi 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 xi' . 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 {h1,h2,h3,...hn} . Each hi belongs to the real space i.e, a real number. Now, our task is to generate hi' which are similar to the hi's in the dataset D. So, we need to generate a new dataset D which is represented as {h1',h2',...hn'} 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.

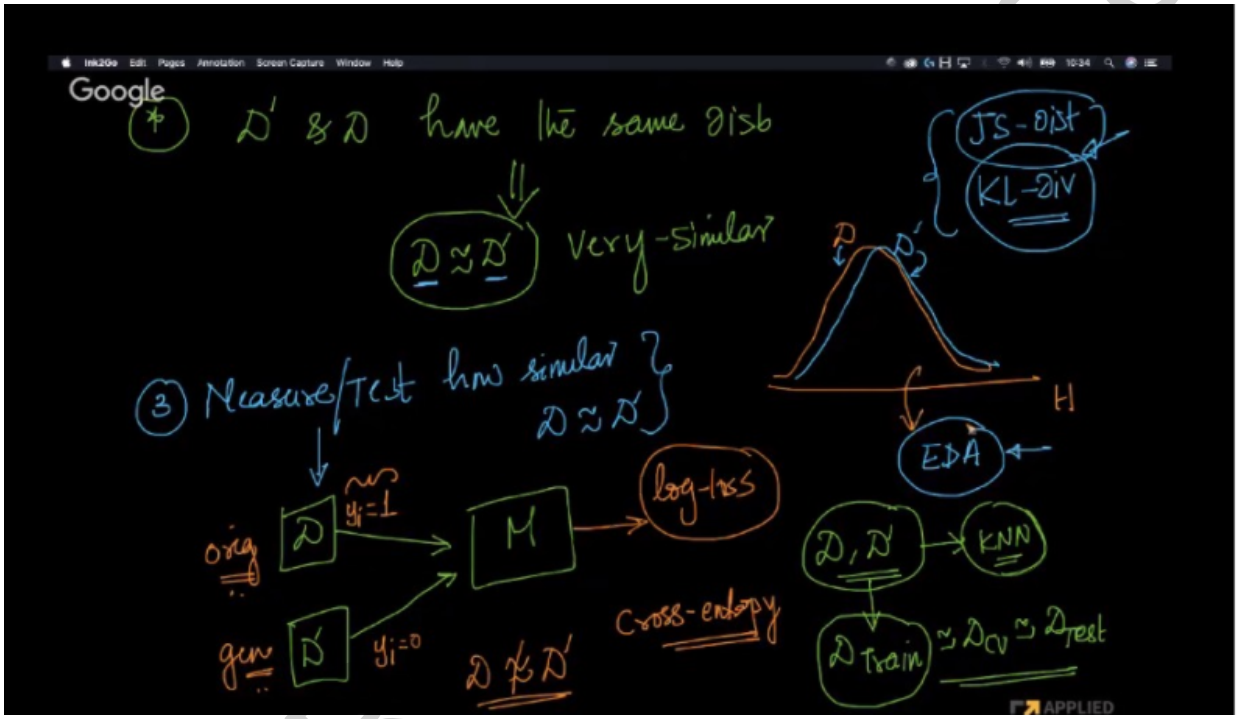
There are many measures available to check whether the two distributions are similar or not. For example, we have Kl- divergence.

One way to check is as follows.

Create a dataset A with {(xi,yi)} where xi belongs to D and yi is 1 for all i.

Create a dataset B with {(xi,yi)} where xi belongs to D' and yi is 0 for all i.

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



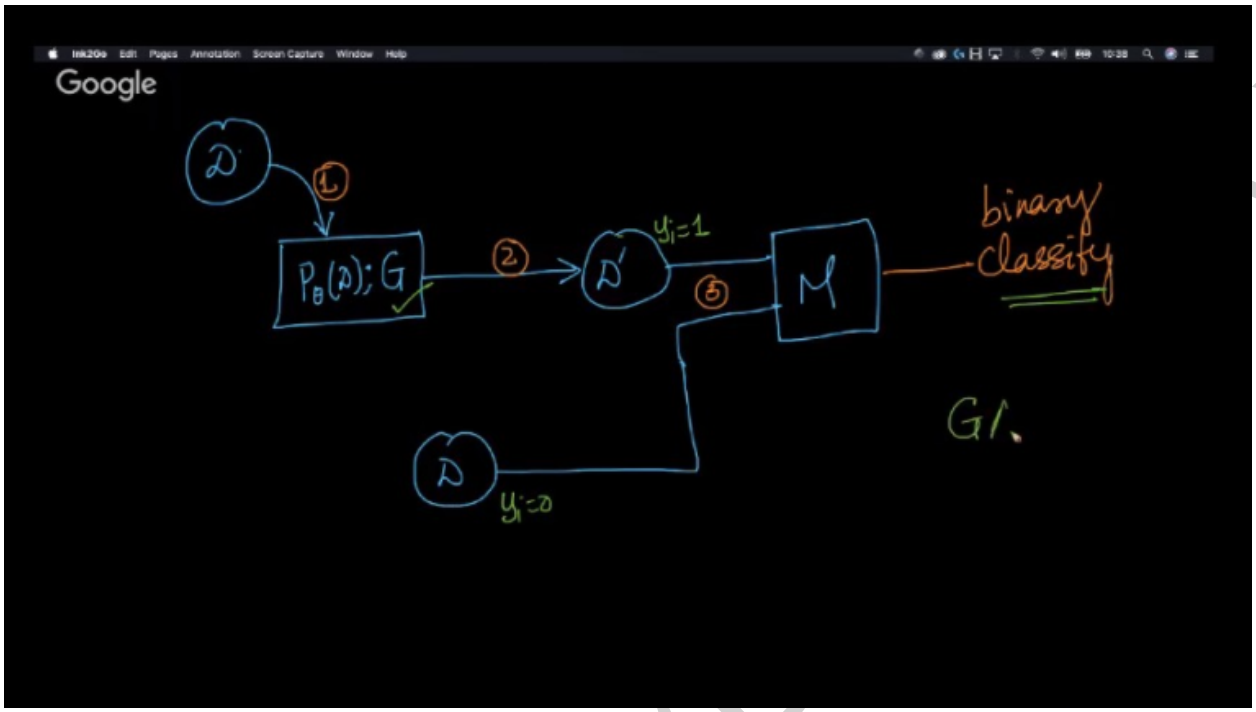
Timestamp : 36:43

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

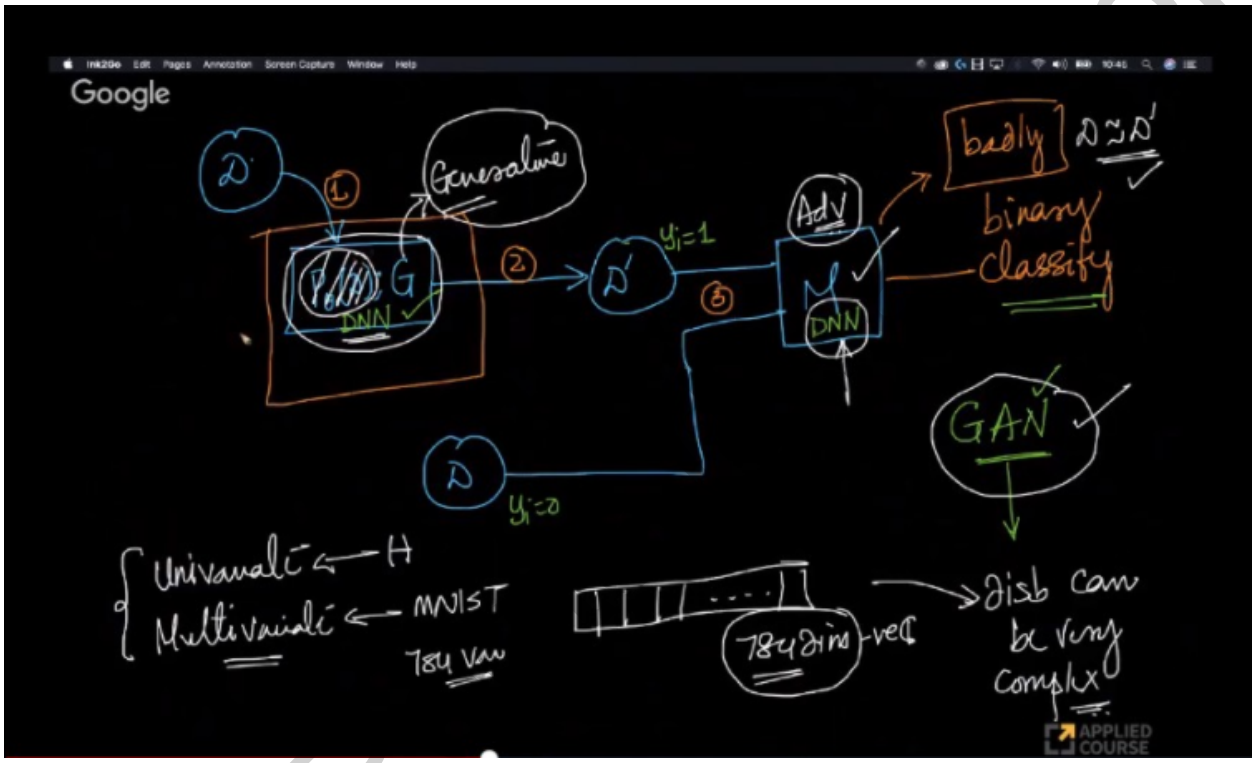
1.) We first generate some samples and put them in D' or D' is the set of generated samples. How do we generate them ? We already discussed the procedure. We first find the probability distribution from where the D comes from or find the parameters to it. Once we know, we sample from that distribution.
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.

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



Timestamp : 48:14

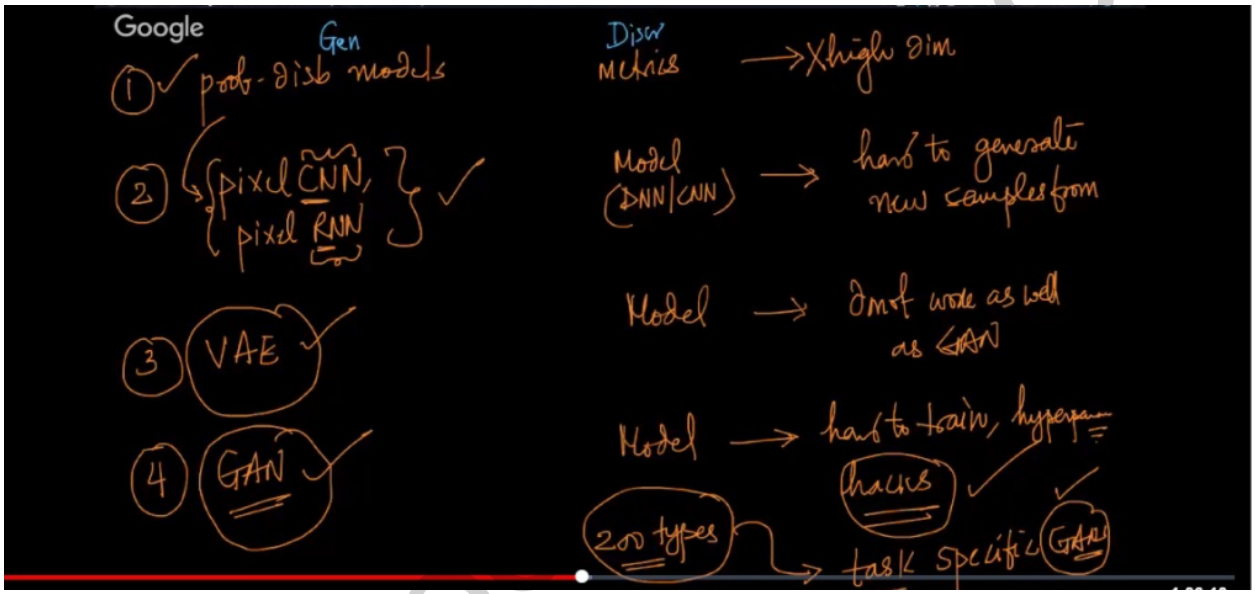
We can see that we replaced both the parts by a deep neural network. So, if the final model M which is a dnn gets a bad accuracy, then D and D' are similar. This means, we can simply take the first model which is a dnn and generate samples. This is because D and D' are similar.

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

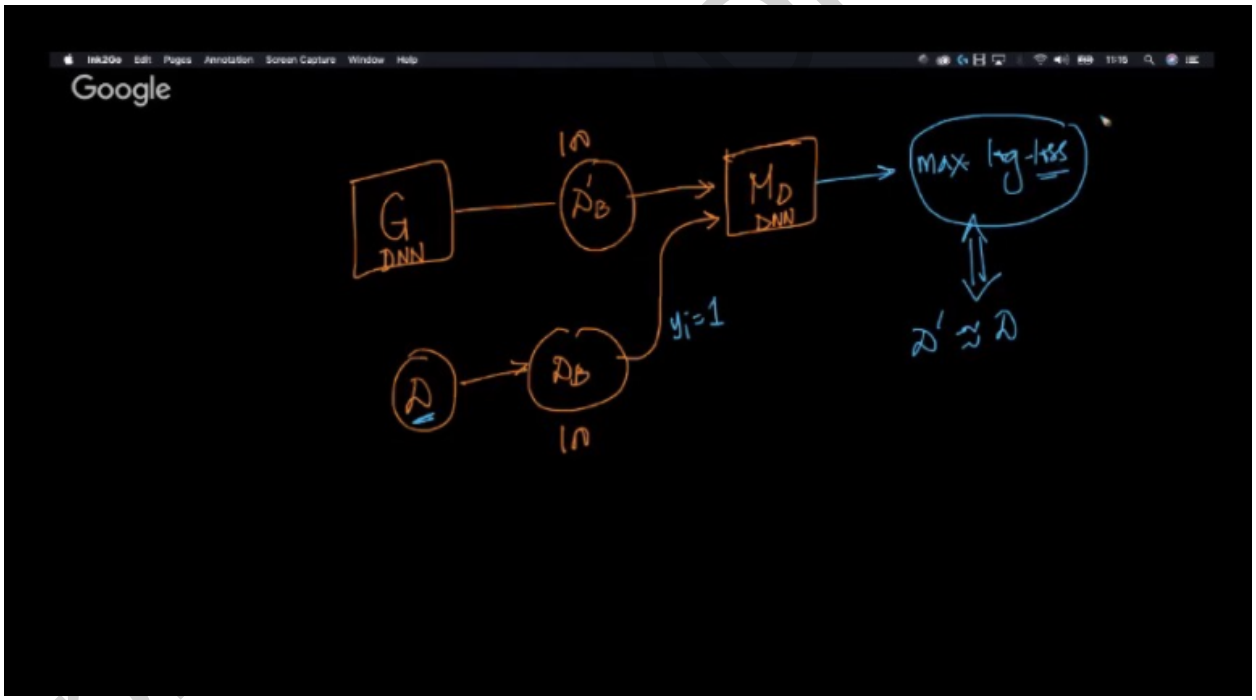
Some applications of GAN.

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.) PixelDTGAN : 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

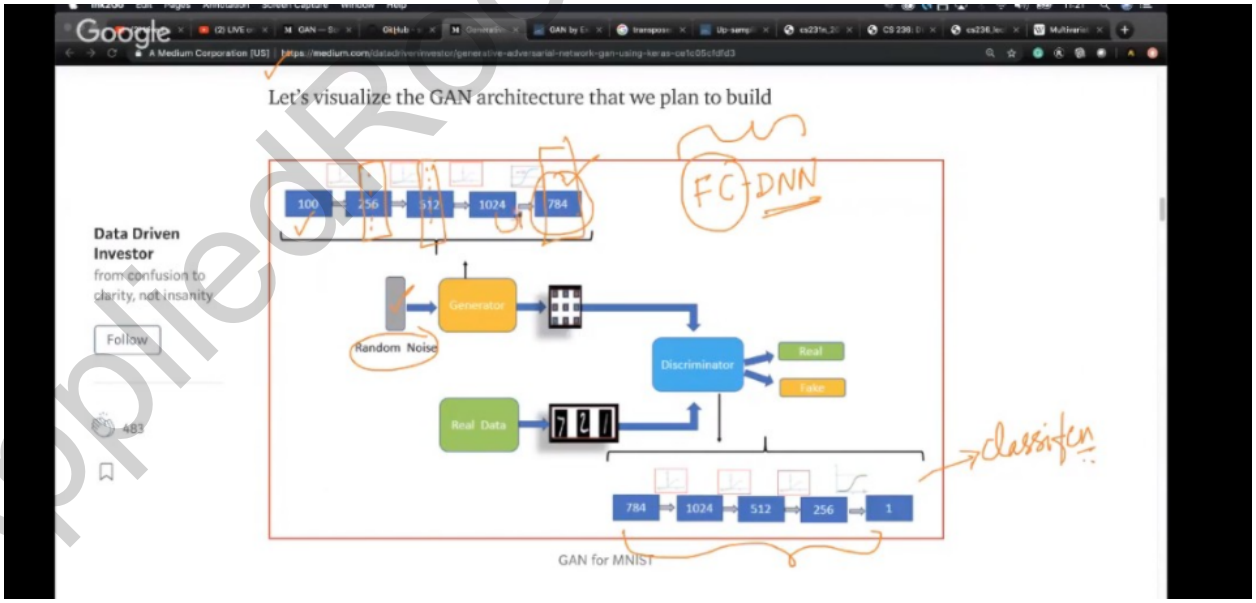
1.) First, we generate some sample data using the generator network. Let's say we do it in batches. Let's also assume that each batch contains 100 points.

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 ?



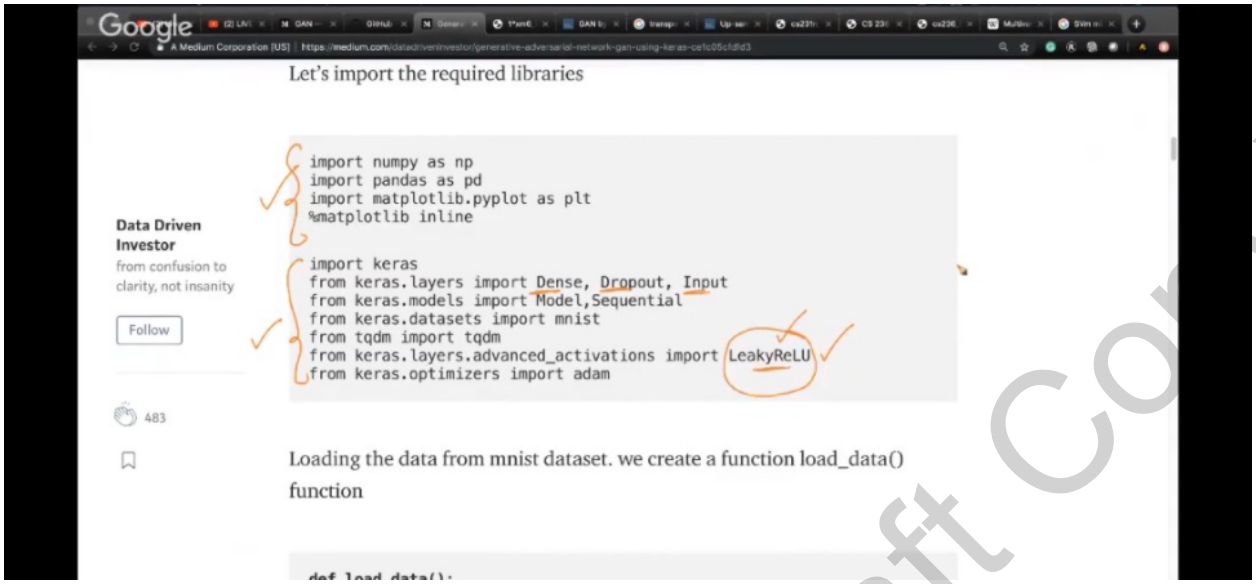
Timestamp : 01:23:50

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.

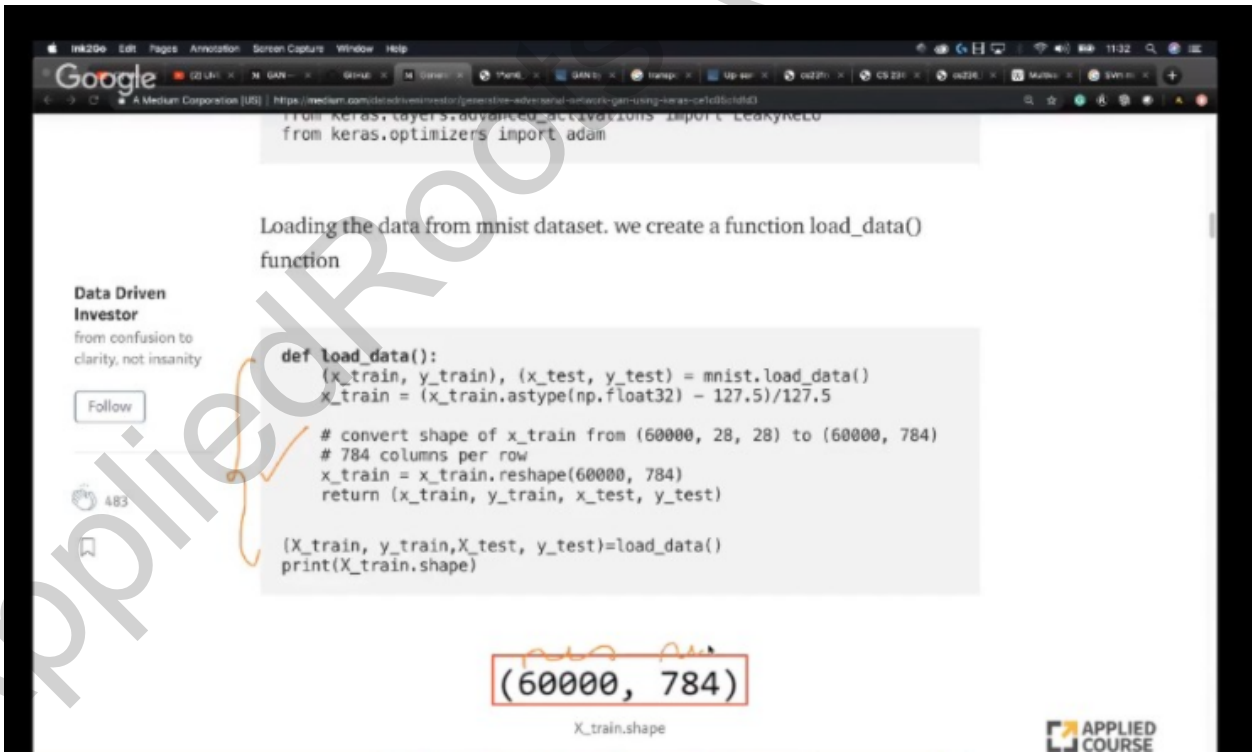
1.) First, fix the final model M which is a discriminator and then generate some random 100 vectors which are 784 dimensional. Then, train the generator network.
2.) Now, fix the generator and train the discriminator network. Here we need to maximize the log loss. So, for this we use gradient ascent 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.



Timestamp : 01:34:34

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



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.

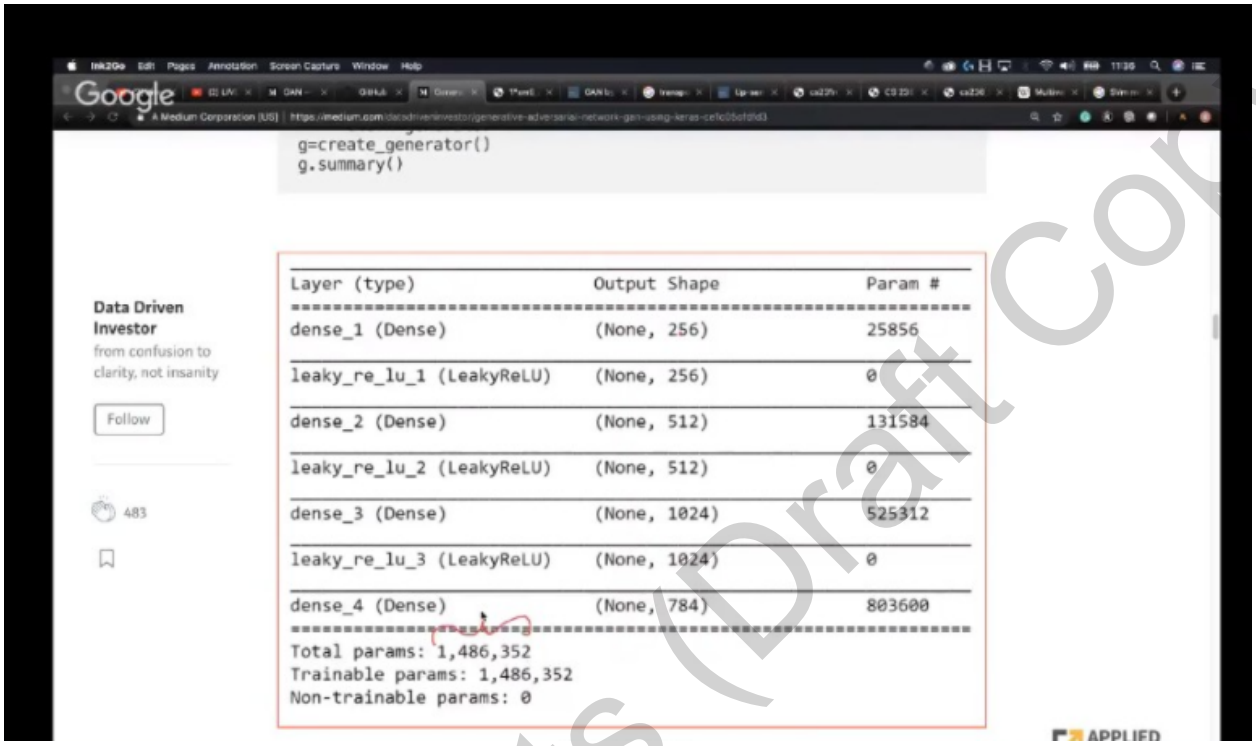
Timestamp : 01:35:43

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.



Timestamp : 01:38:39

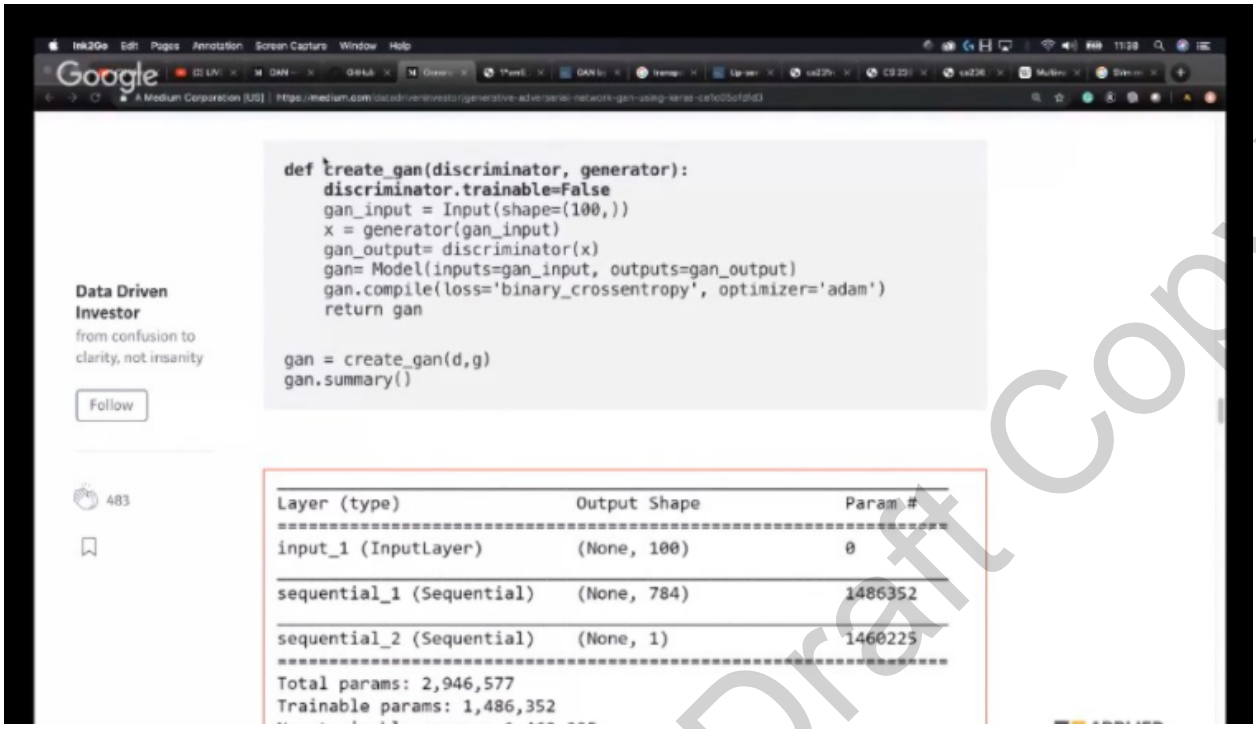
We have around 1,486,352 parameters.

Then, we need to define the discriminator network.

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.

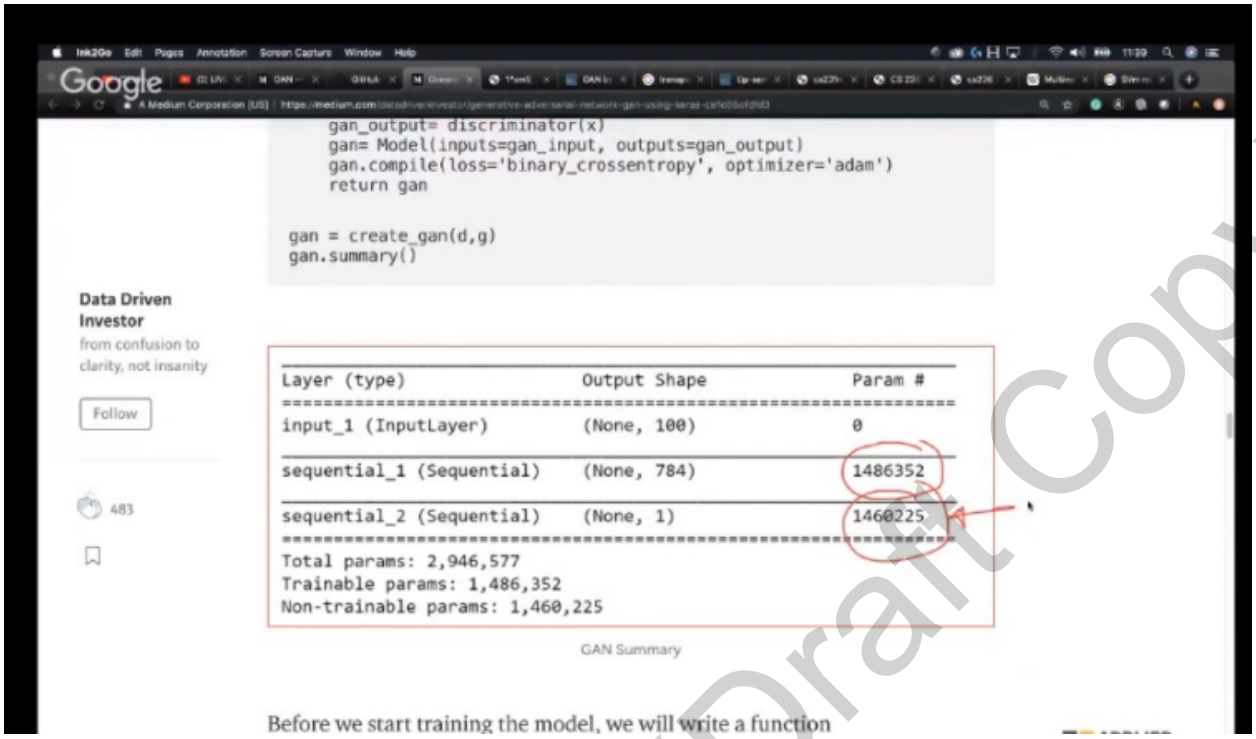
Now, we have to create the GAN from the generator and discriminator network.



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.

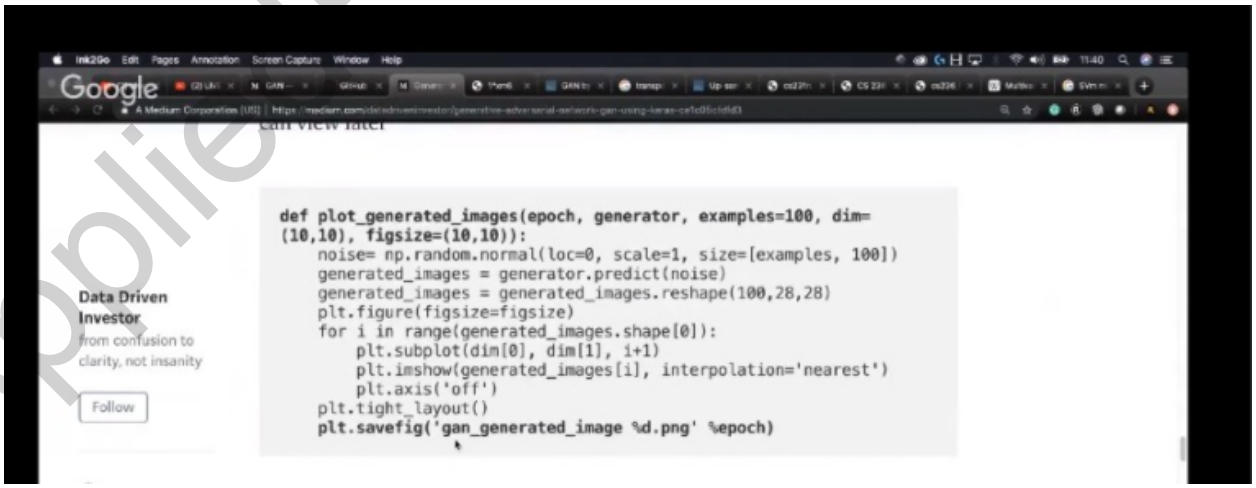
Then, we can print the summary of our model.



Timestamp : 01:42:28

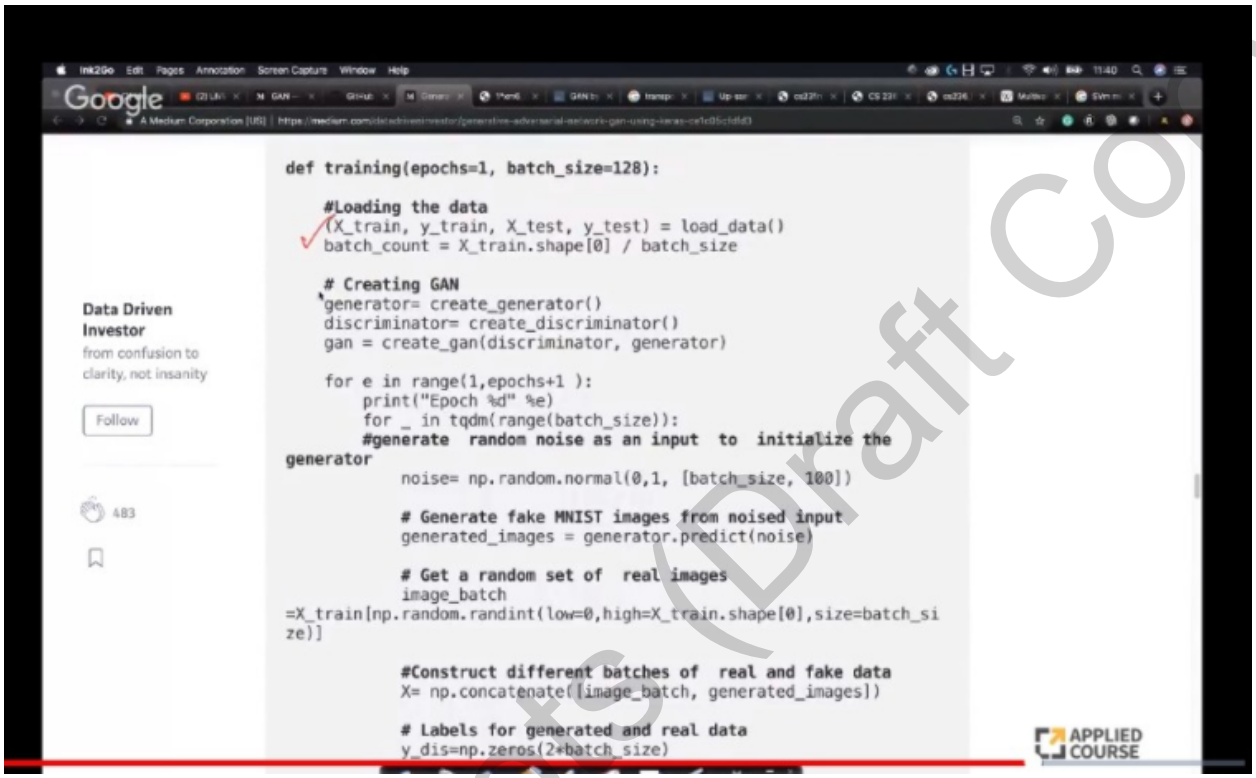
Since we have freezed the discriminator network, we also have some non-trainable parameters.

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



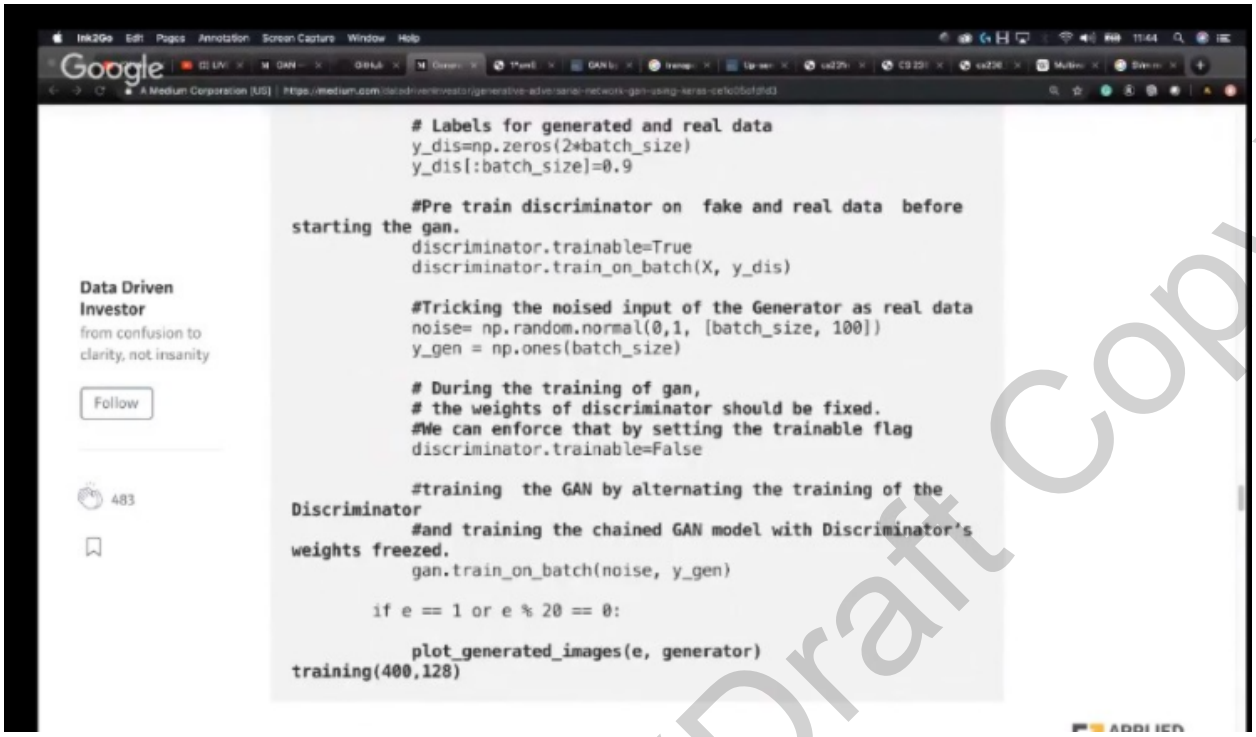
Timestamp : 01:42:55

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.



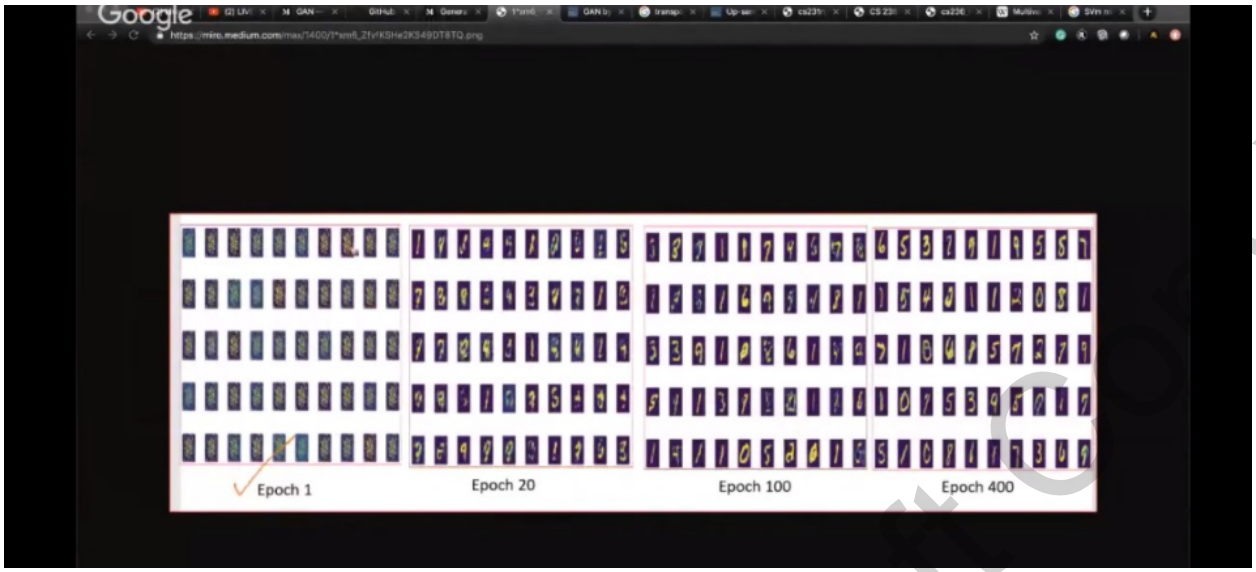
Timestamp : 01:43:31

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.



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.

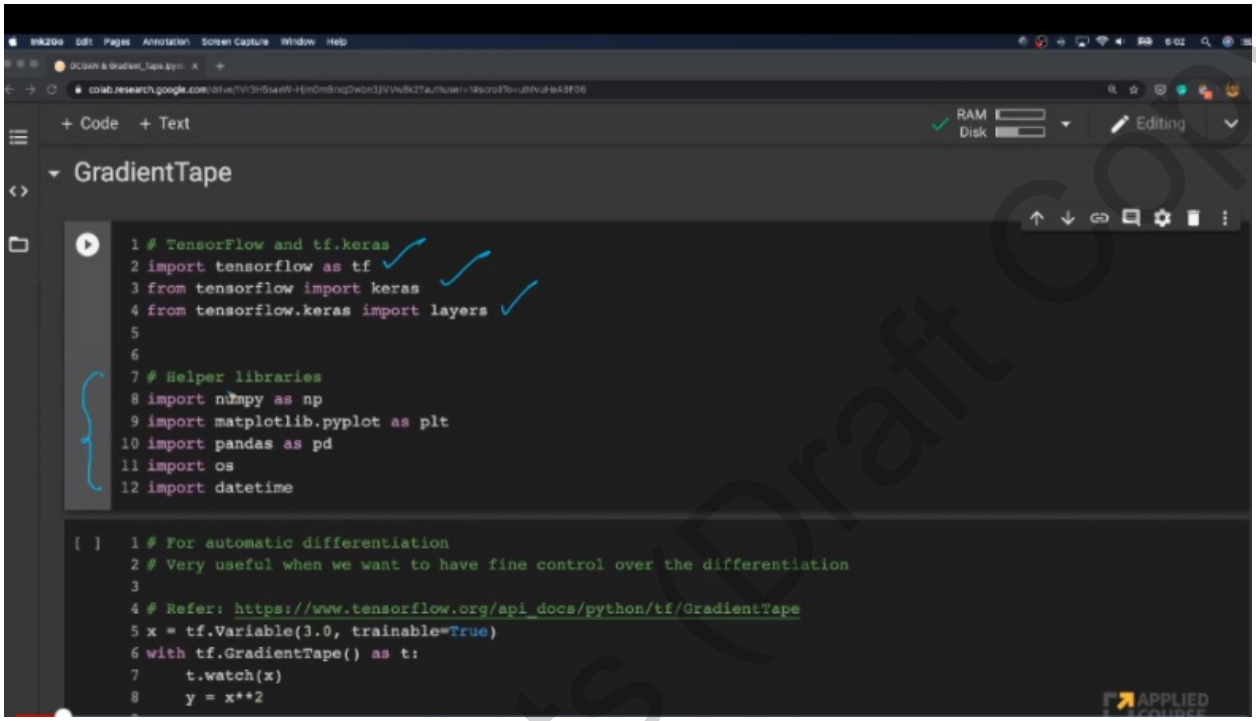


Timestamp : 01:50:18

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.

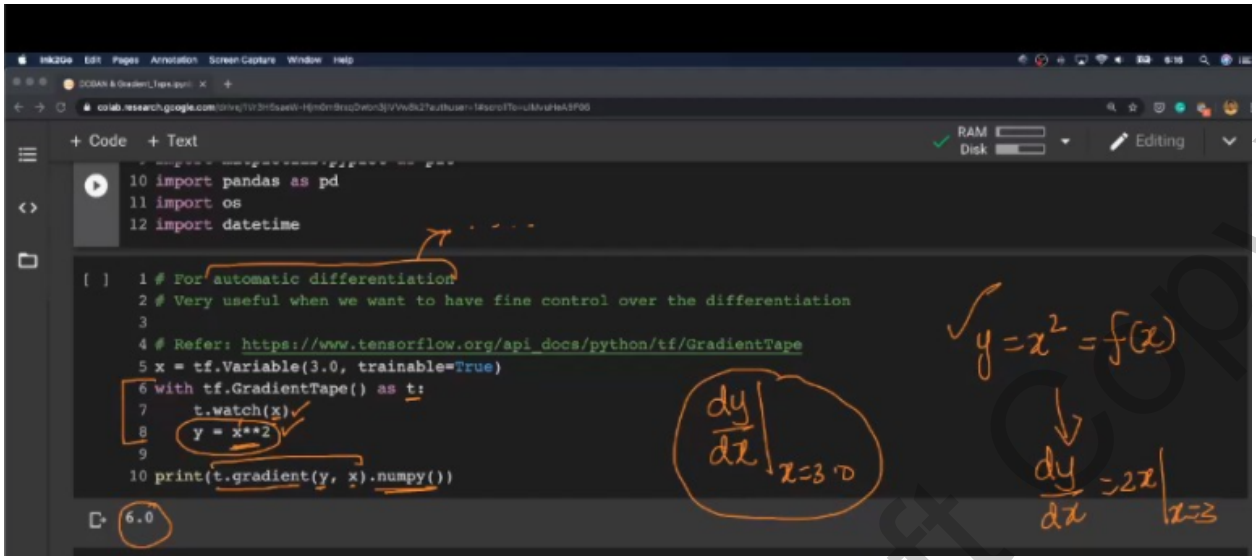
63.2 Code Walkthrough: DC-GANs and GradientTape in TensorFlow 2



Timestamp: 05:37

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

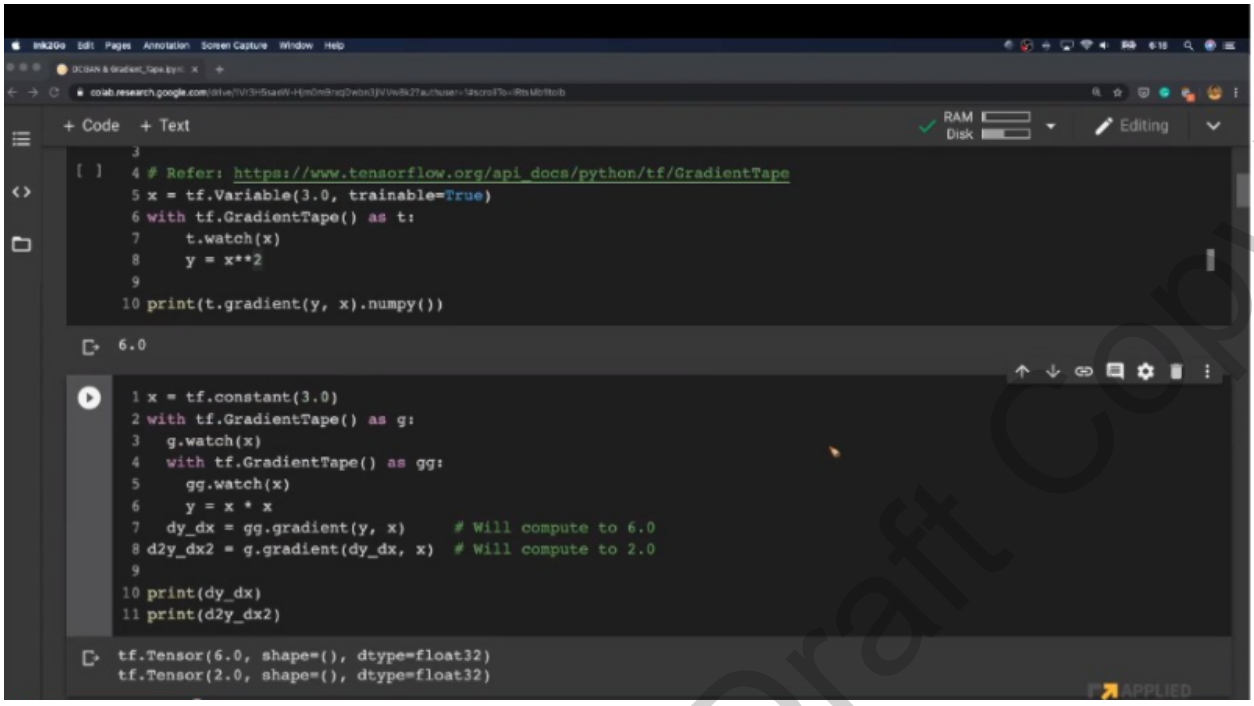
Gradient Tape : It gives us a fine control over the gradients. In keras, while defining any layer, we only define the forward function. The operations needed to be done in the backward stage i.e, backpropagation are taken care of by the keras. It's called automatic differentiation.



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.



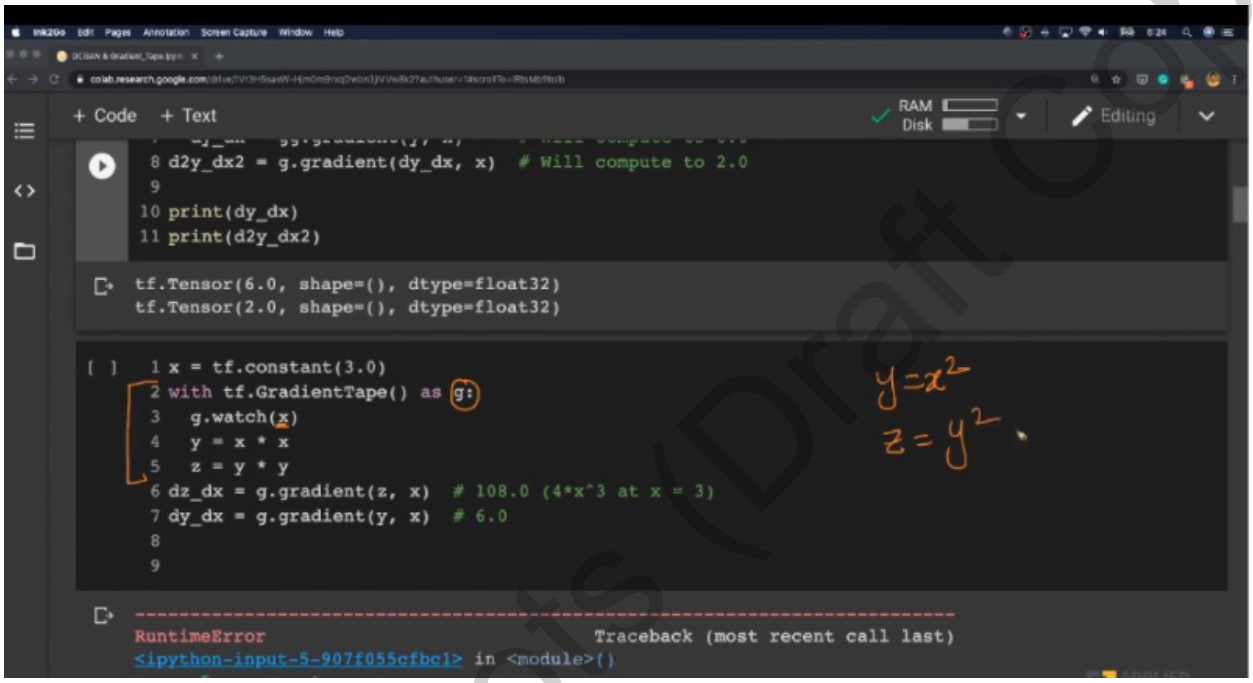
Timestamp : 21:21

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**x.
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.



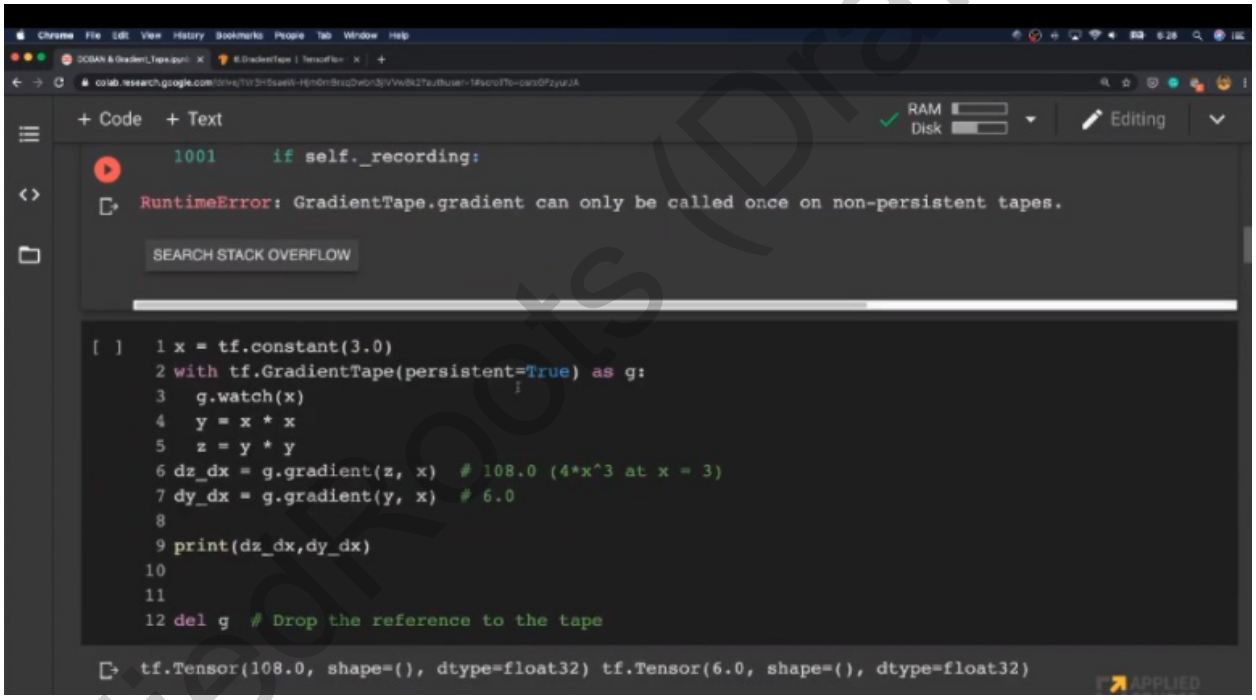
Timestamp : 27:32

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.

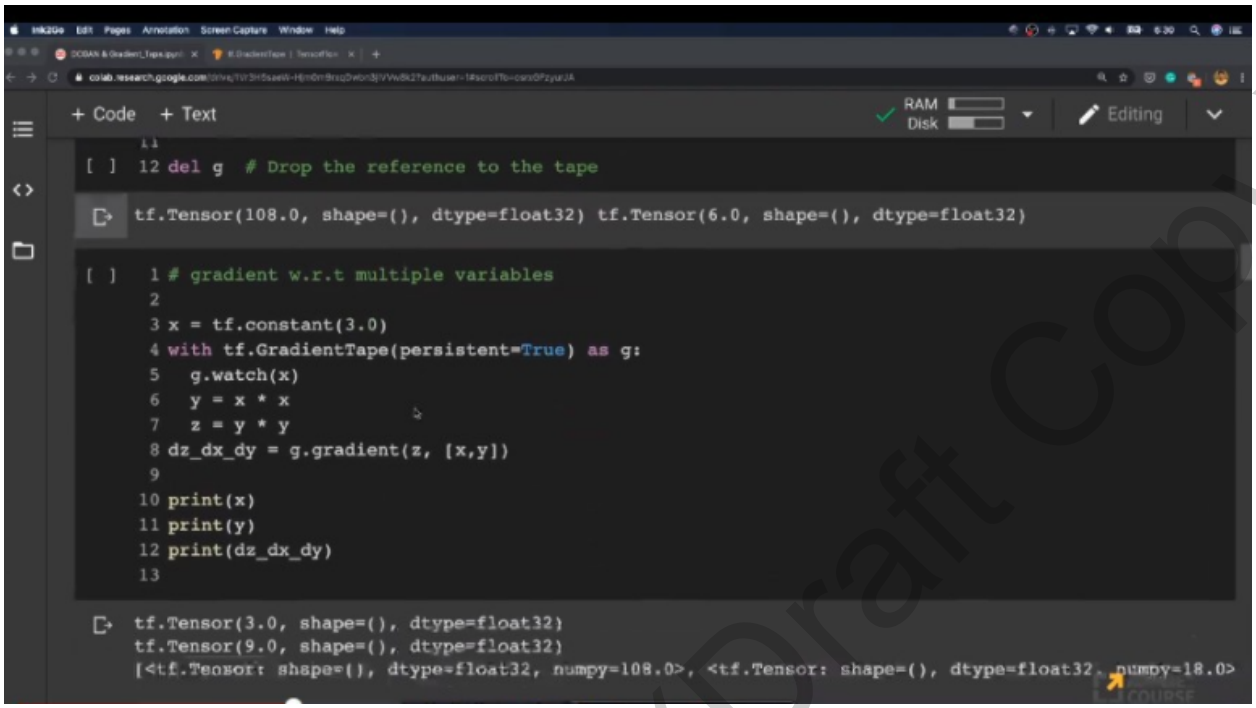


Timestamp : 31:05

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

del g : It simply deletes the reference g.

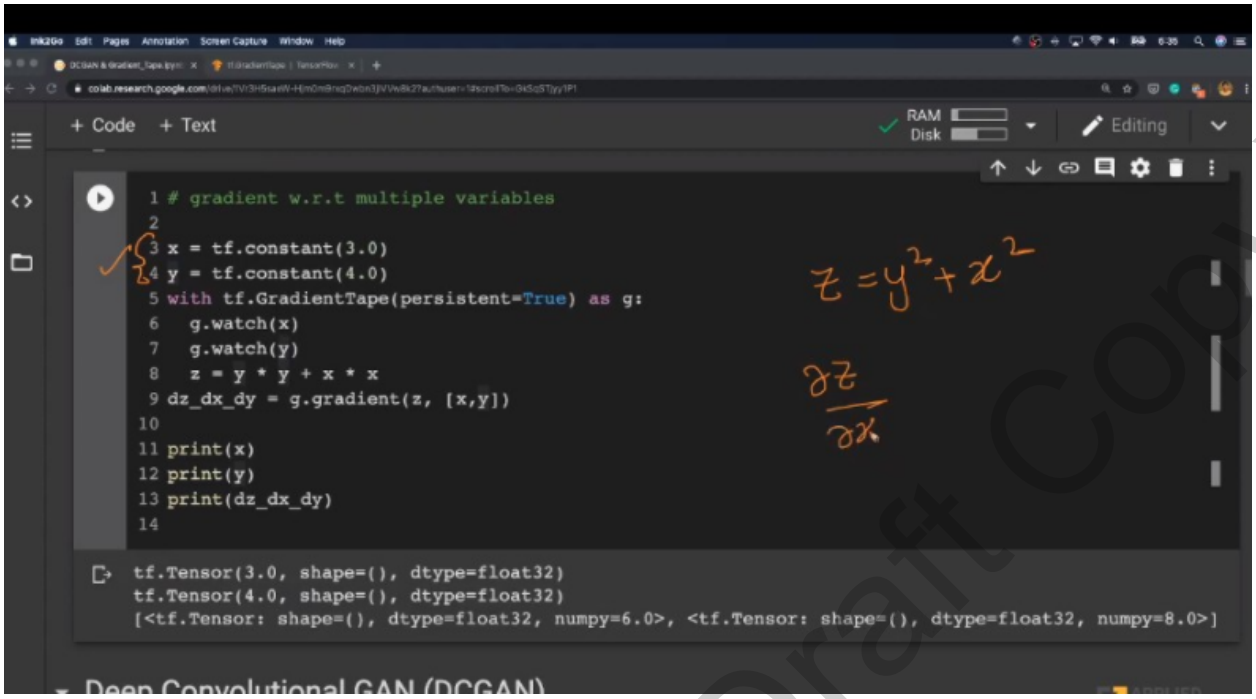
Can we compute the gradient with respect to multiple variables ?



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.

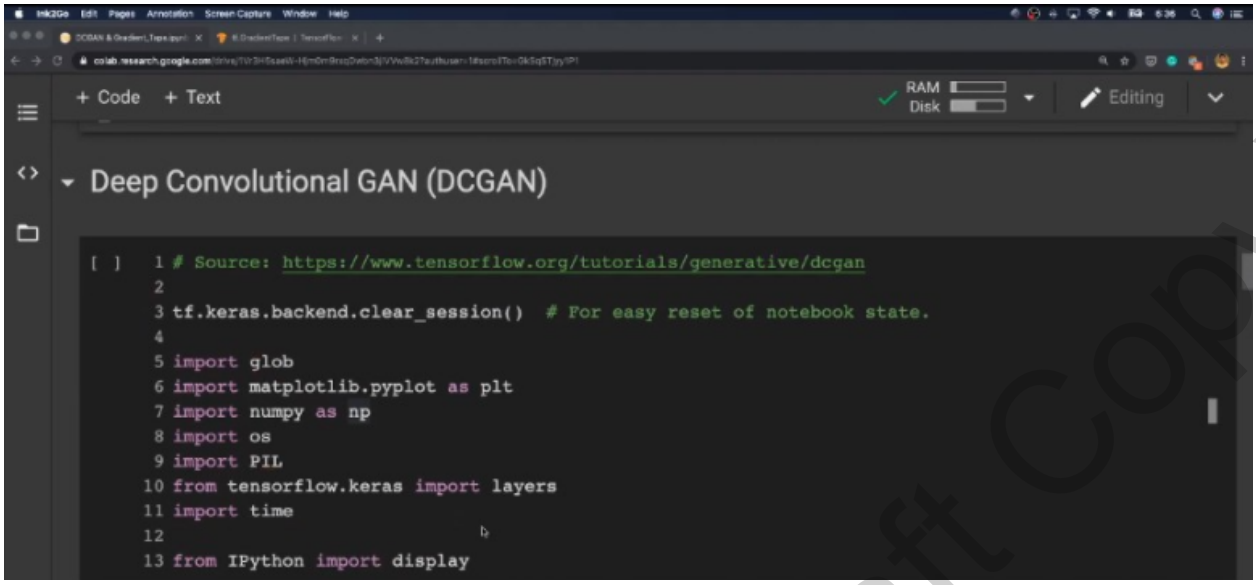


Timestamp : 37:54

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.

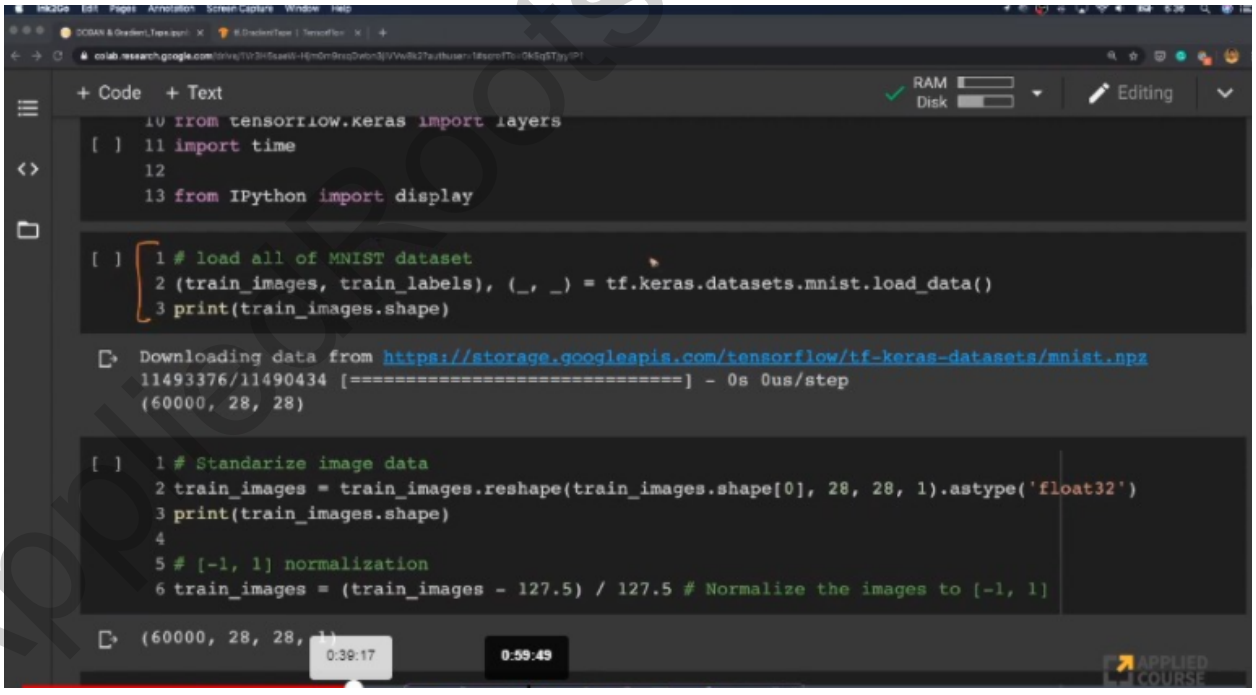
Deep Convolutional Generative Adversarial Networks



Timestamp : 39:11

As usual we import a set of libraries.

Then, we need to load the data.



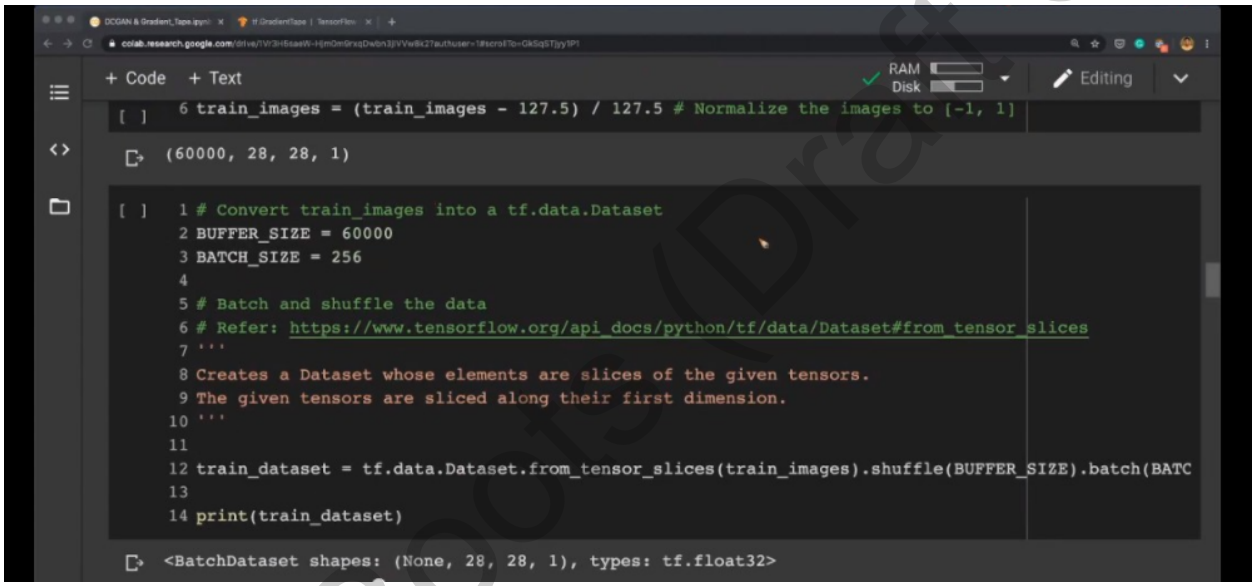
Timestamp : 39:17

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.

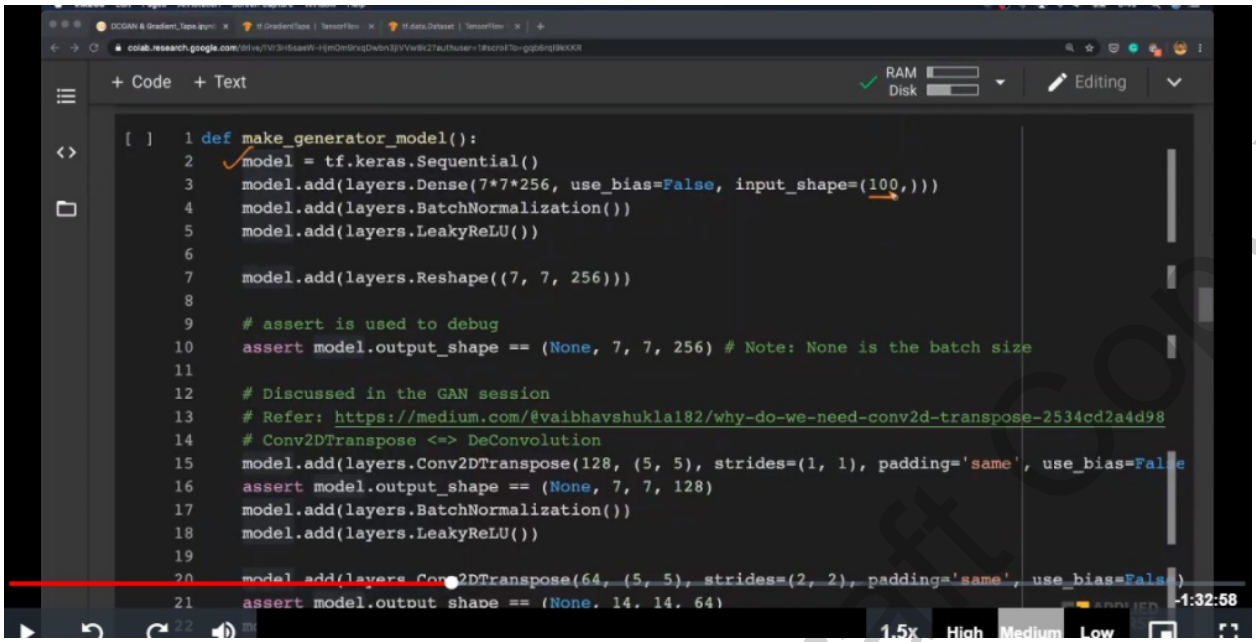
Now, we are going to define the dataset.



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.



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.

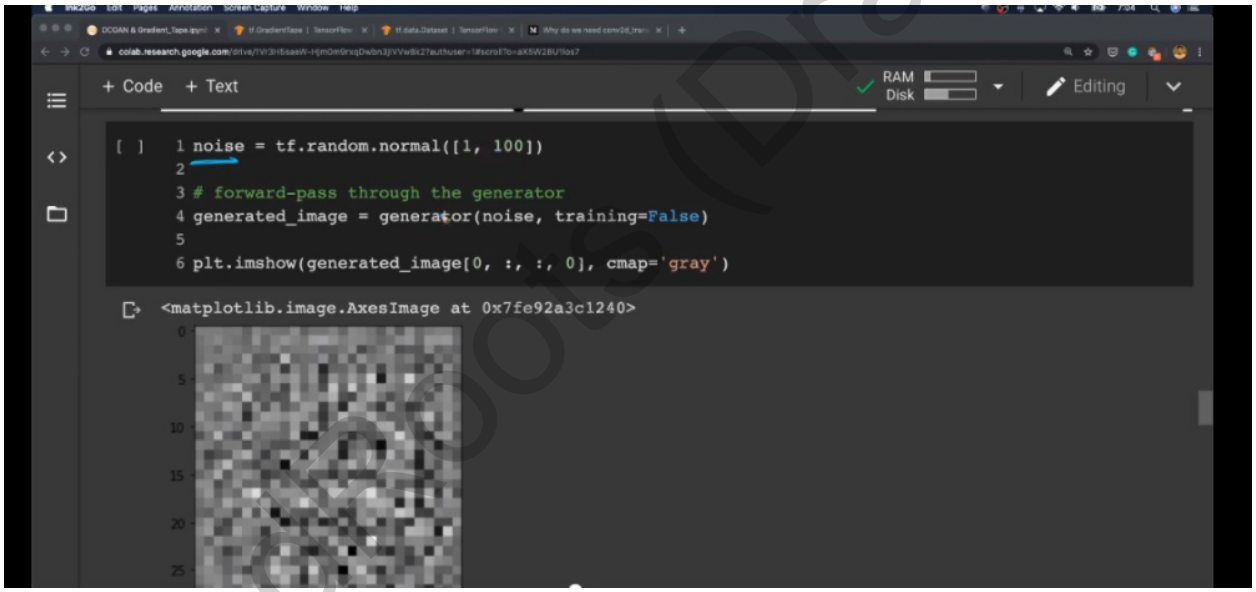
Then, we can check whether the shape of the model output is correct or not by defining an assert statement.\

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.

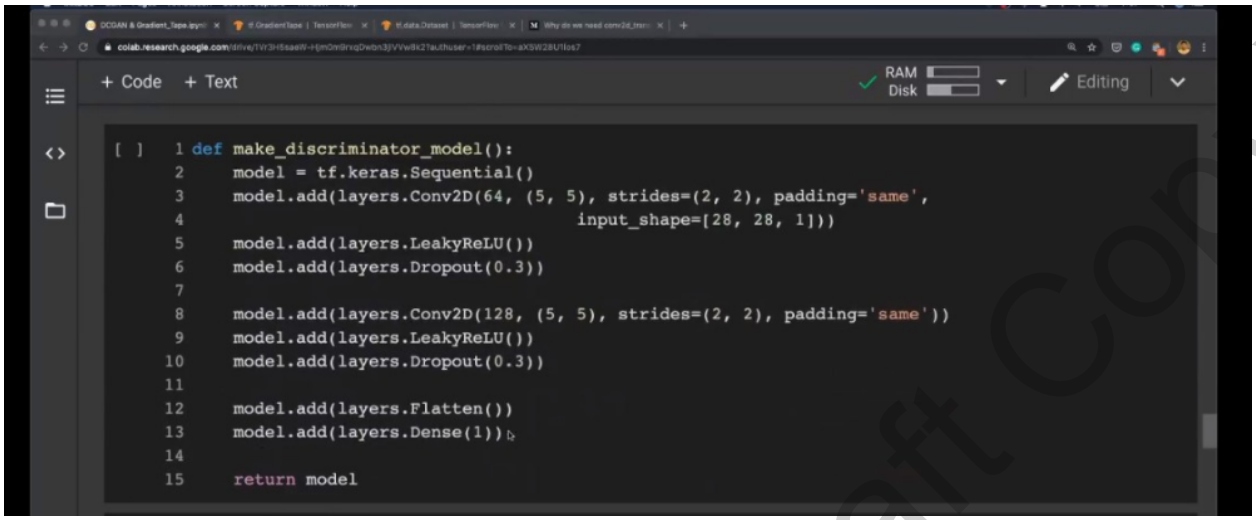


Timestamp : 01:06:49

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.



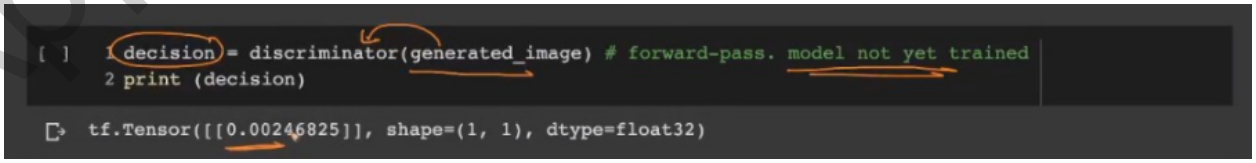
Timestamp : 01:10:04

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.

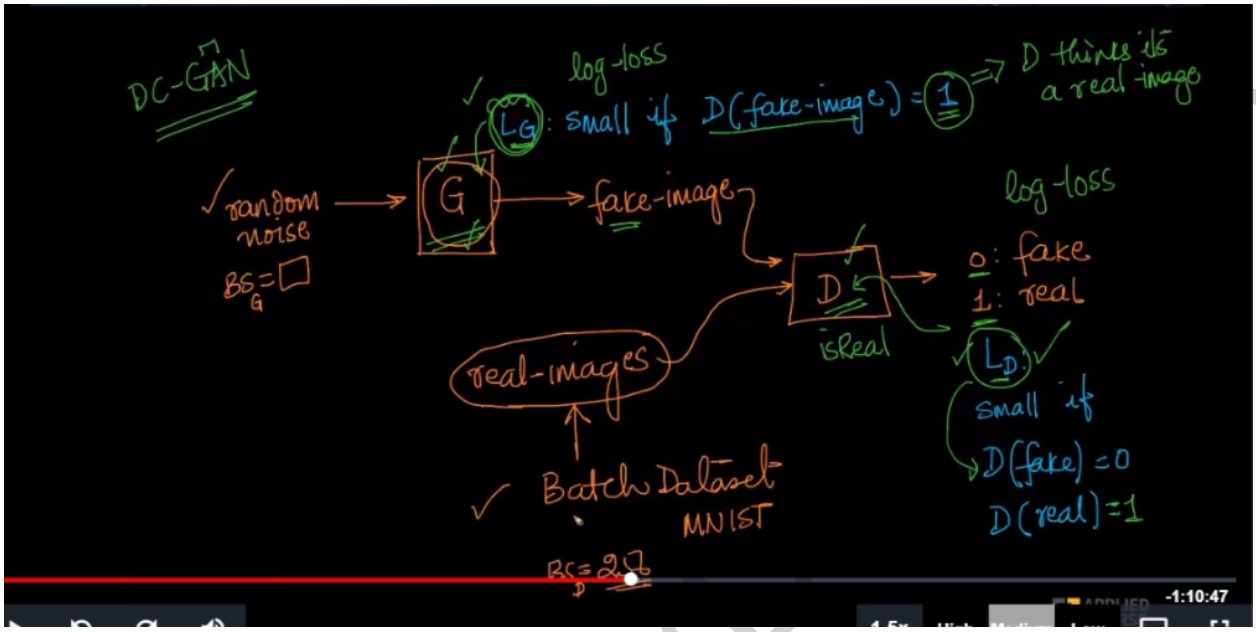
In the final layer, we defined a Dense layer with output being a single neuron.

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



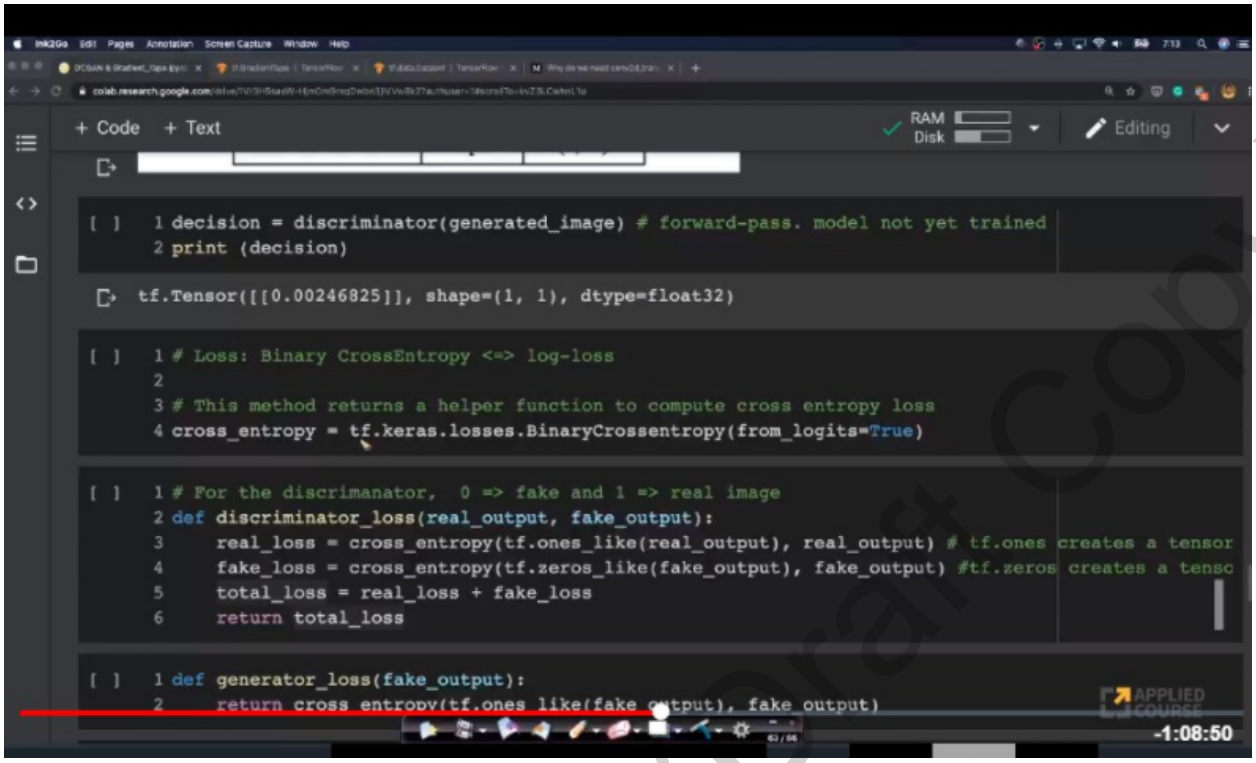
Timestamp : 01:13:24

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.



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.

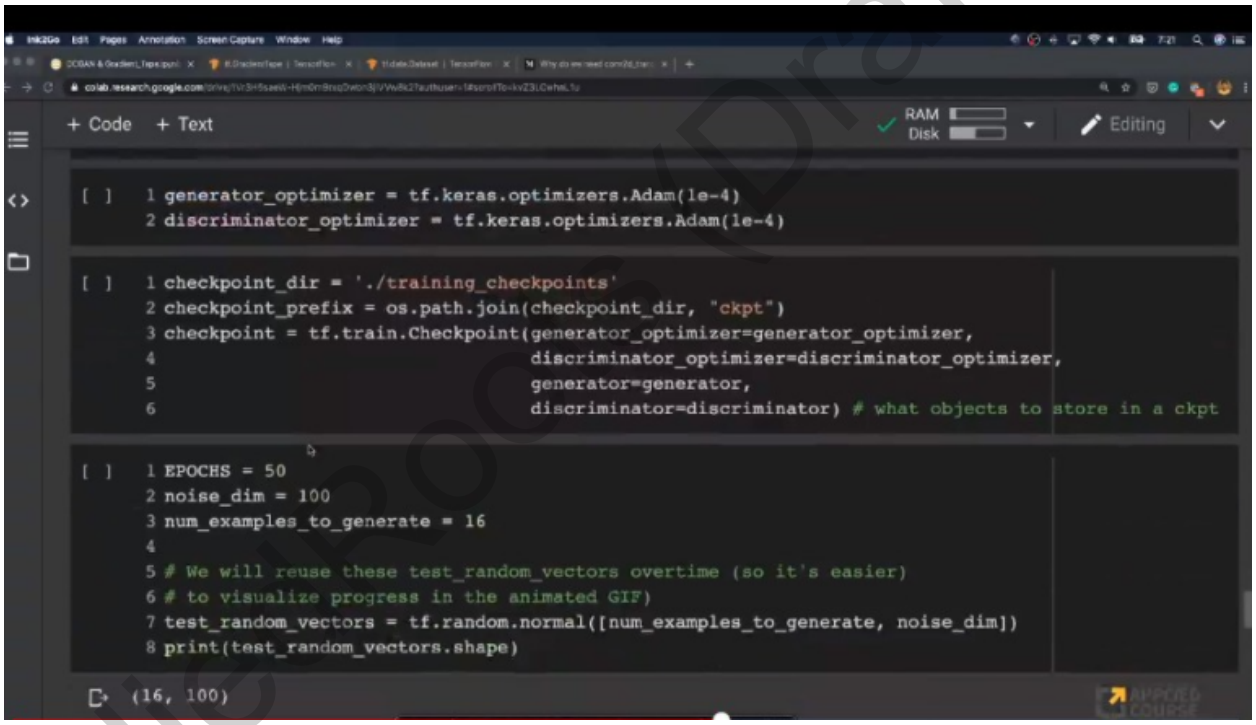
The procedure for computing the loss is easy to understand.

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.

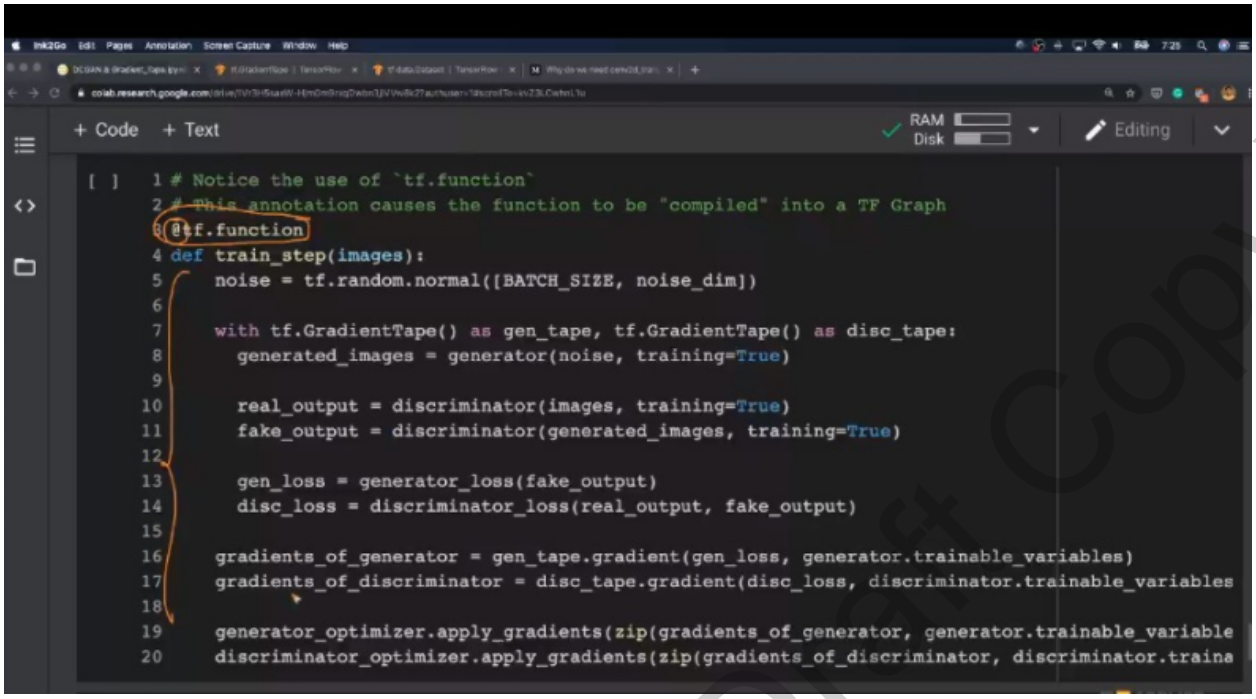


Timestamp : 01:23:41

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.



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.

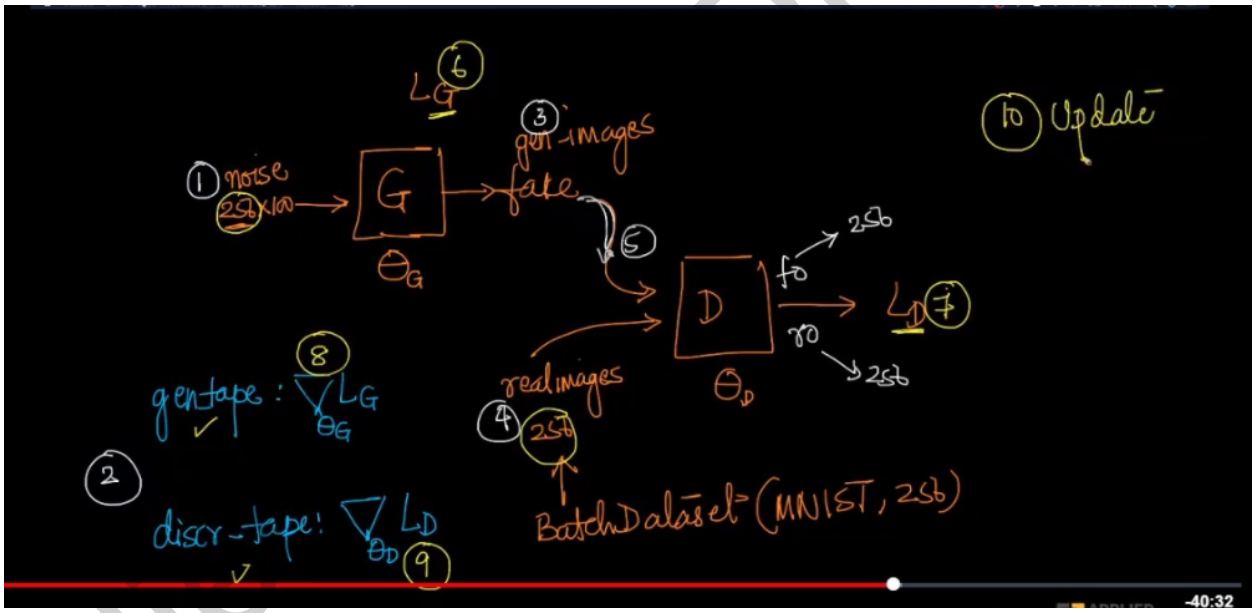
Then, we calculate both the loss based on the generator network and the discriminator network.

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

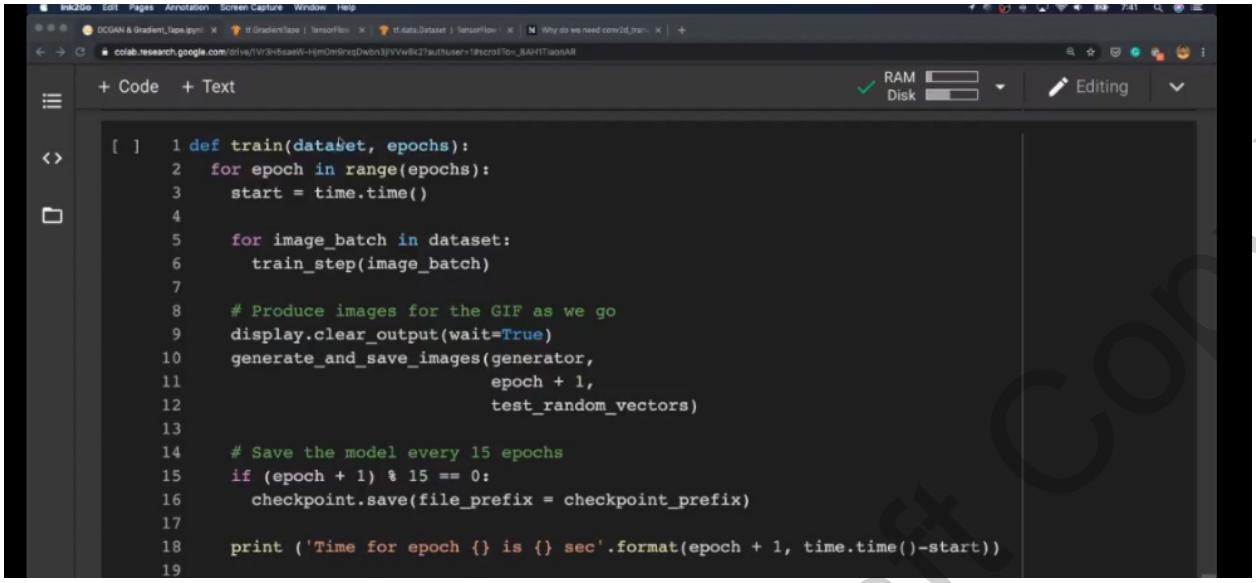
Then,using the function apply_gradients in the optimizer object we can update the trainable parameters.

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

Overview :



Timestamp : 01:44:06



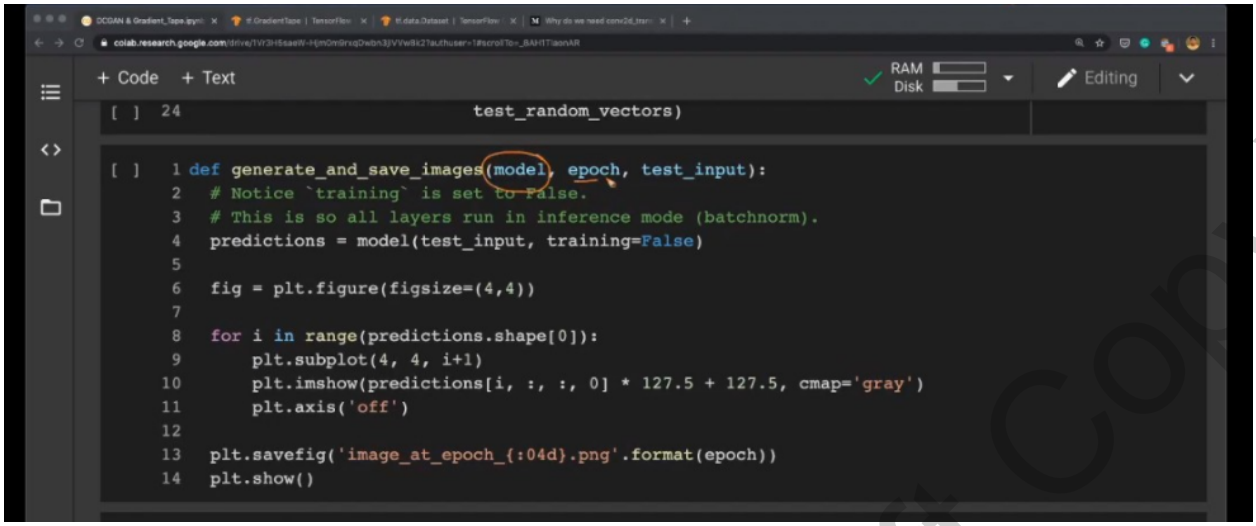
Timestamp : 01:44:30

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.

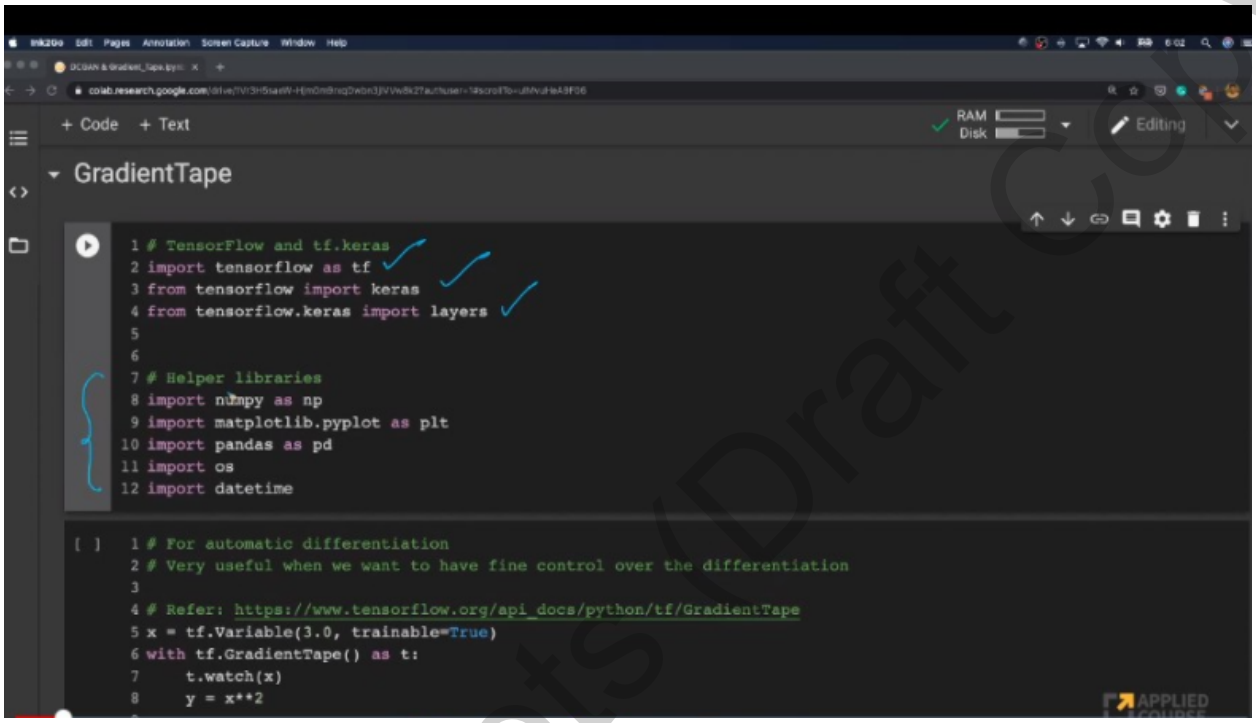
For every 15 epochs, we are storing the checkpoints.



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.

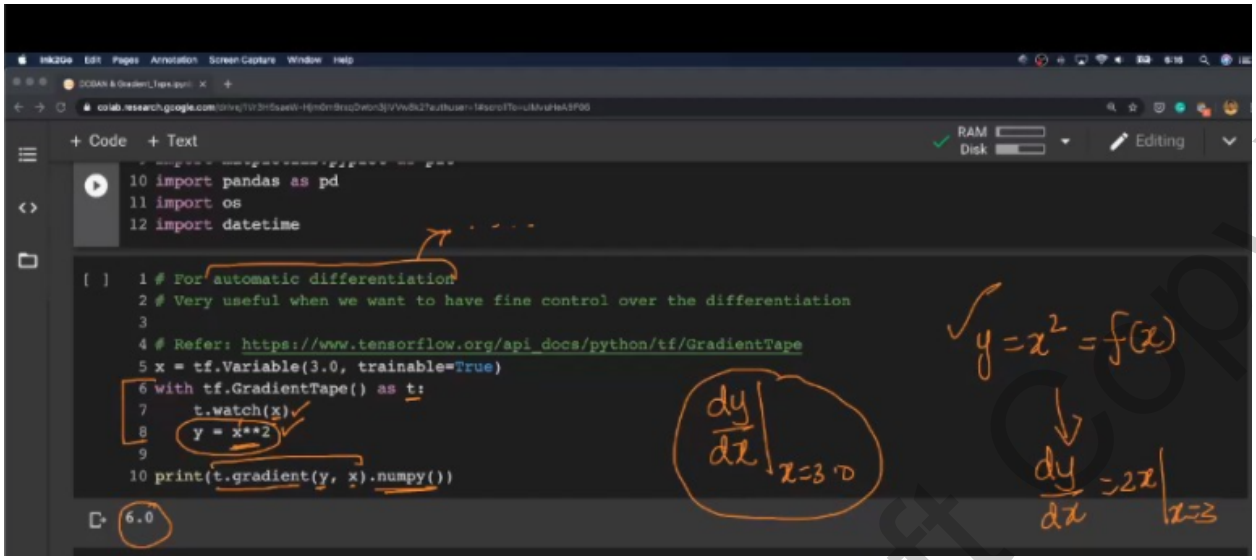
63.2 Code Walkthrough: DC-GANs and GradientTape in TensorFlow 2



Timestamp: 05:37

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

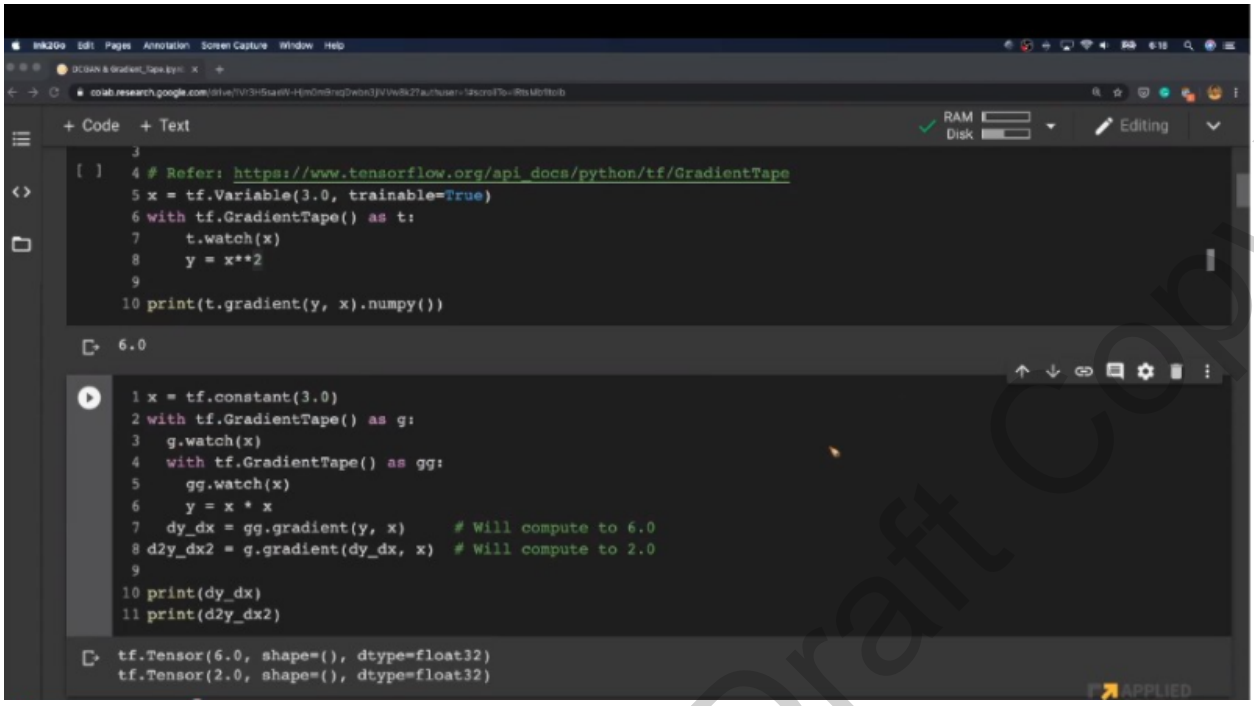
Gradient Tape : It gives us a fine control over the gradients. In keras, while defining any layer, we only define the forward function. The operations needed to be done in the backward stage i.e, backpropagation are taken care of by the keras. It's called automatic differentiation.



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.



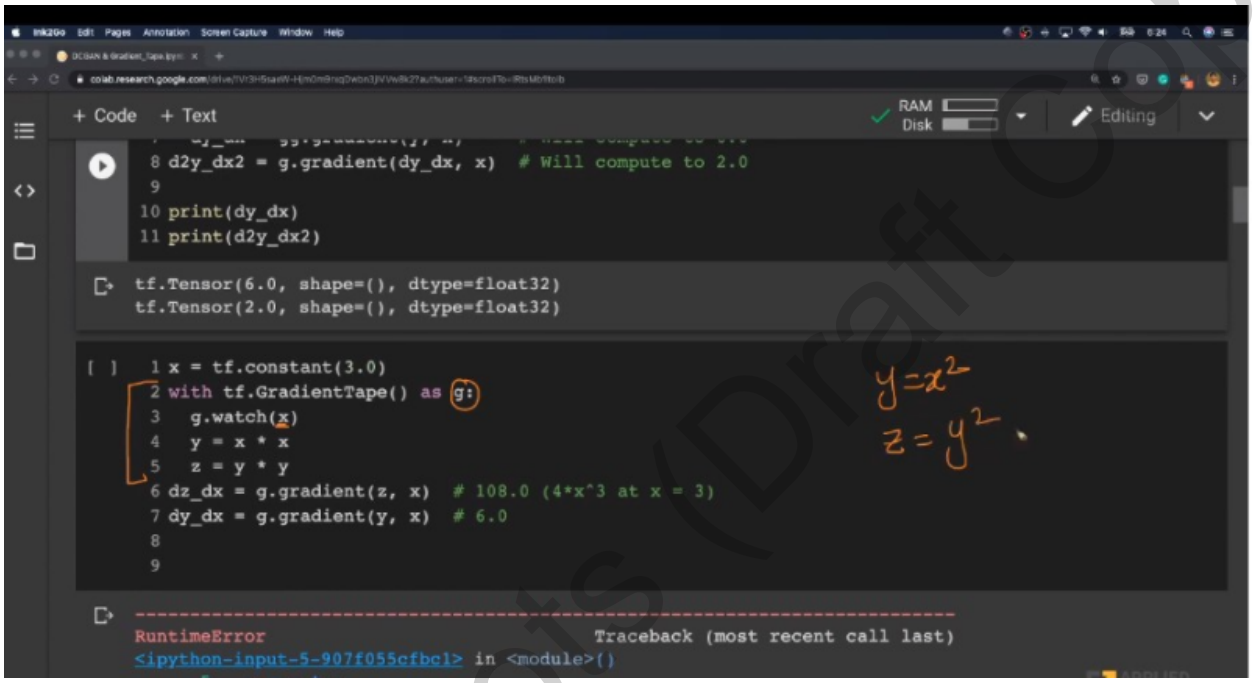
Timestamp : 21:21

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**x.
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.



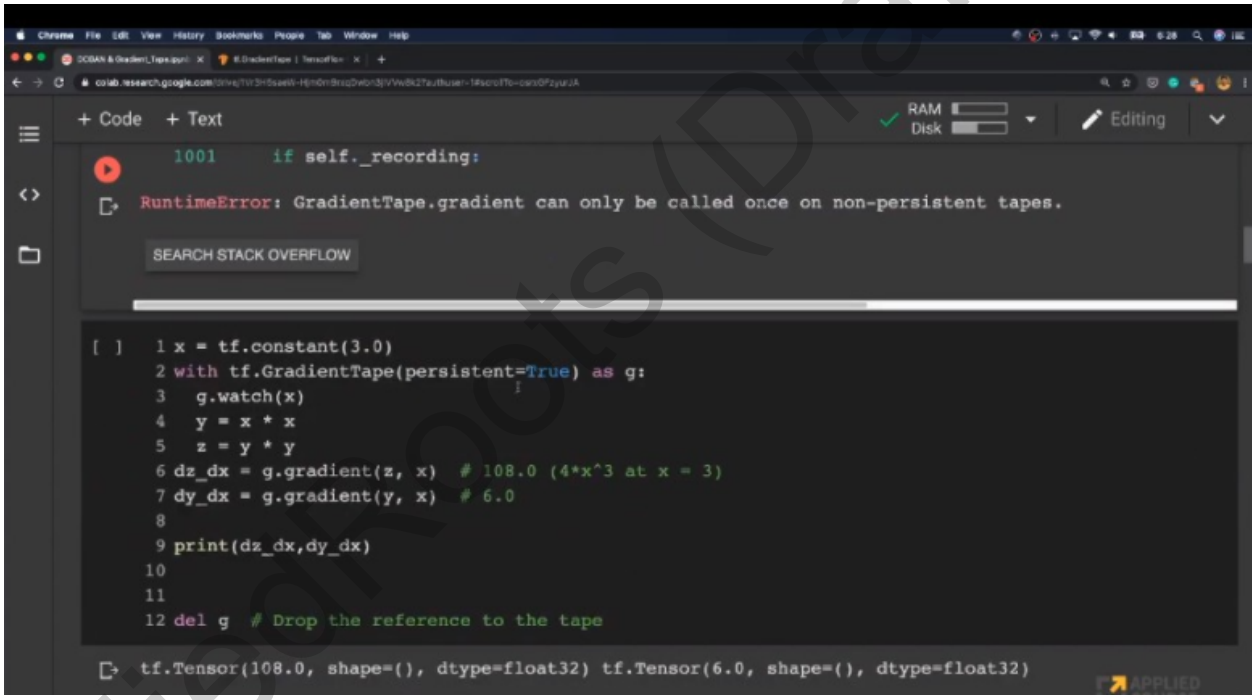
Timestamp : 27:32

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.

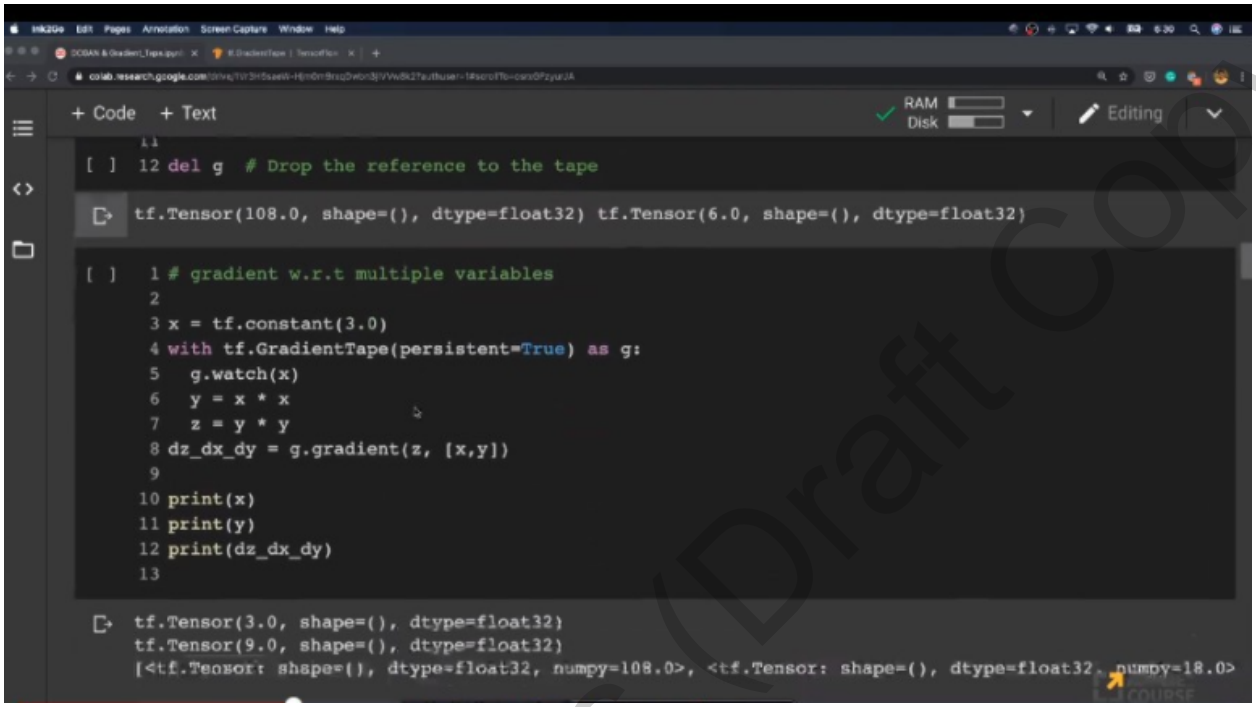


Timestamp : 31:05

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

del g : It simply deletes the reference g.

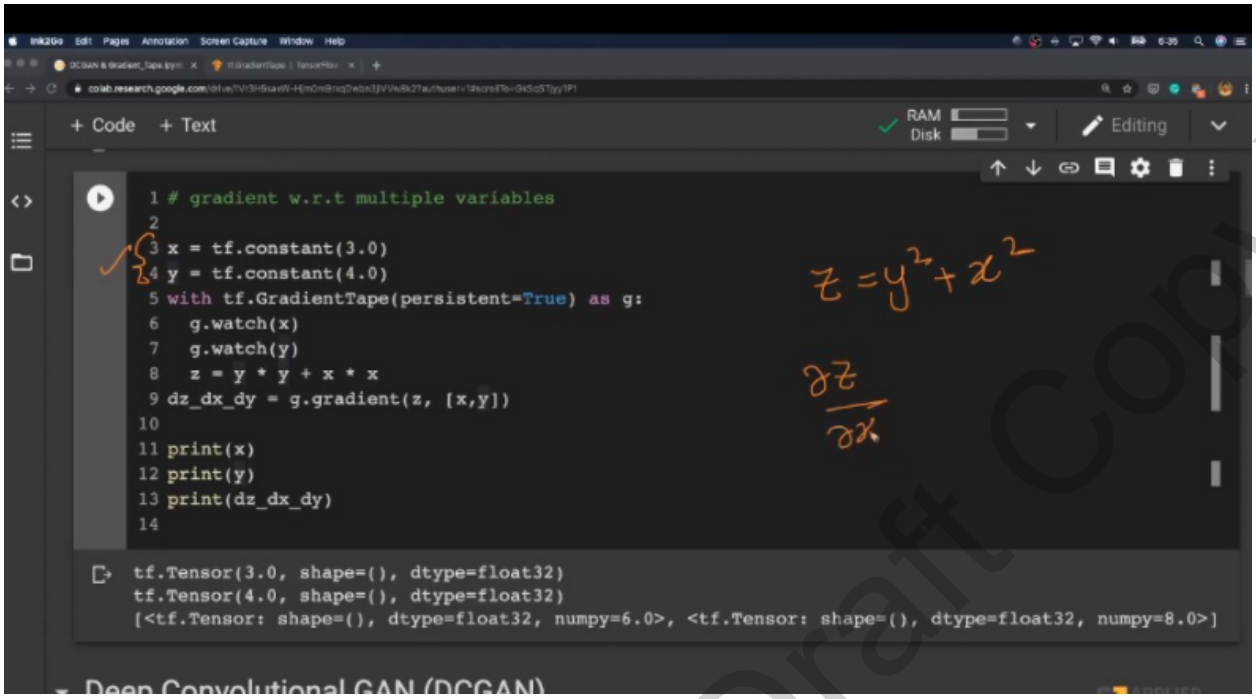
Can we compute the gradient with respect to multiple variables ?



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.

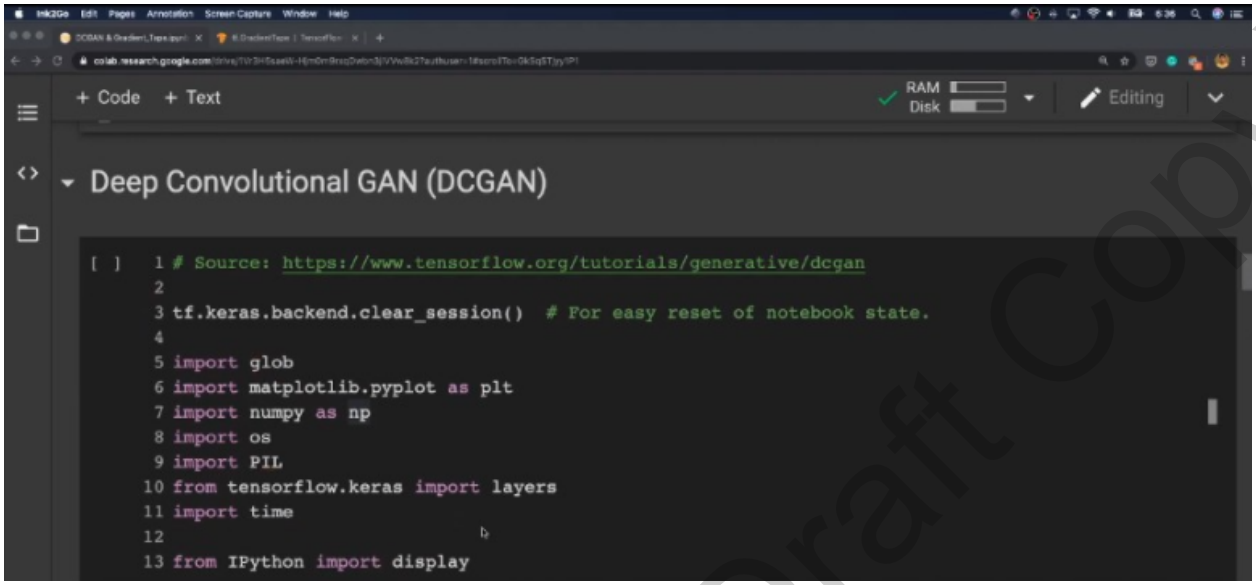


Timestamp : 37:54

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.

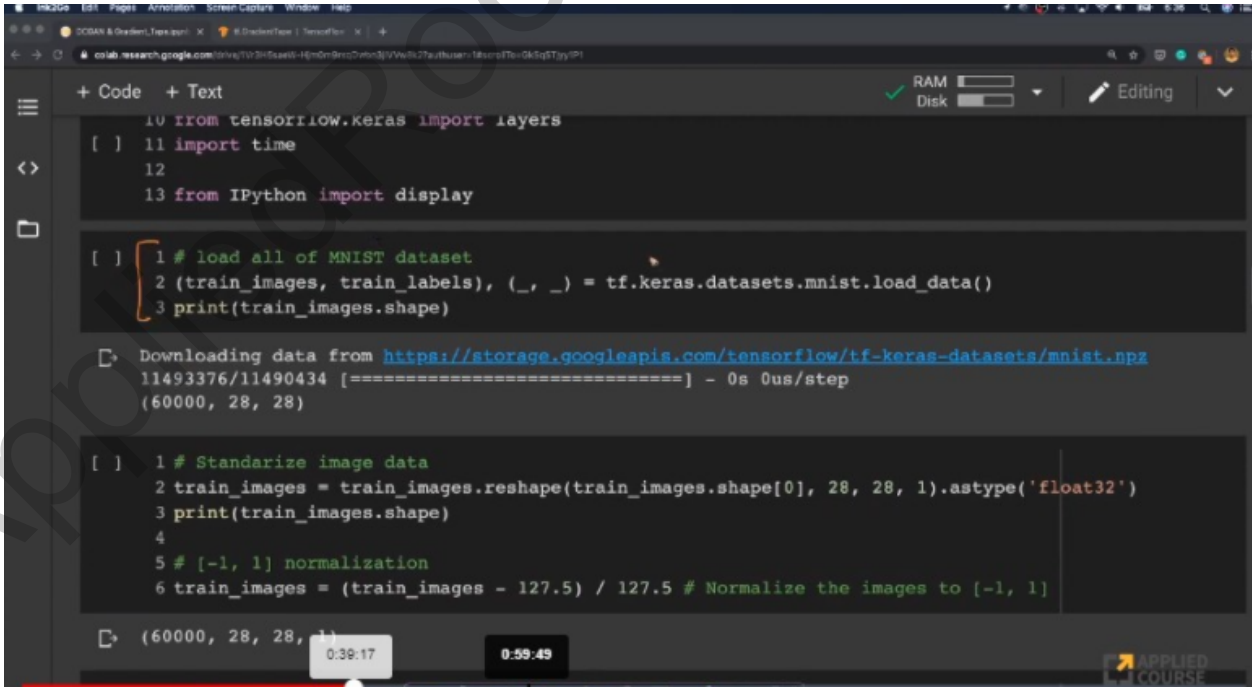
Deep Convolutional Generative Adversarial Networks



Timestamp : 39:11

As usual we import a set of libraries.

Then, we need to load the data.



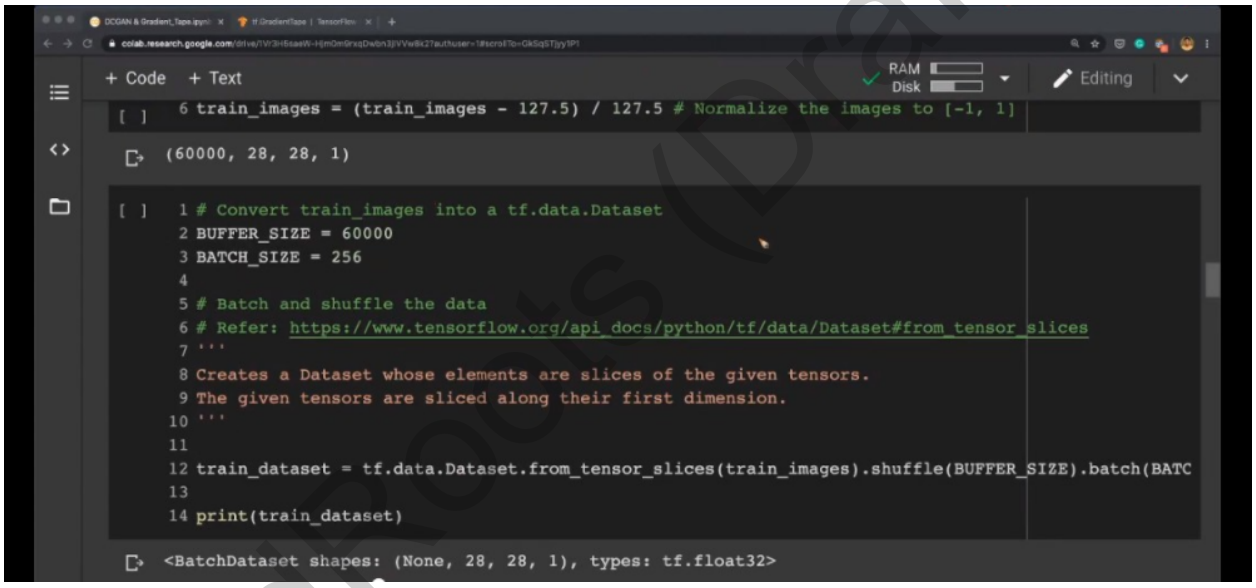
Timestamp : 39:17

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.

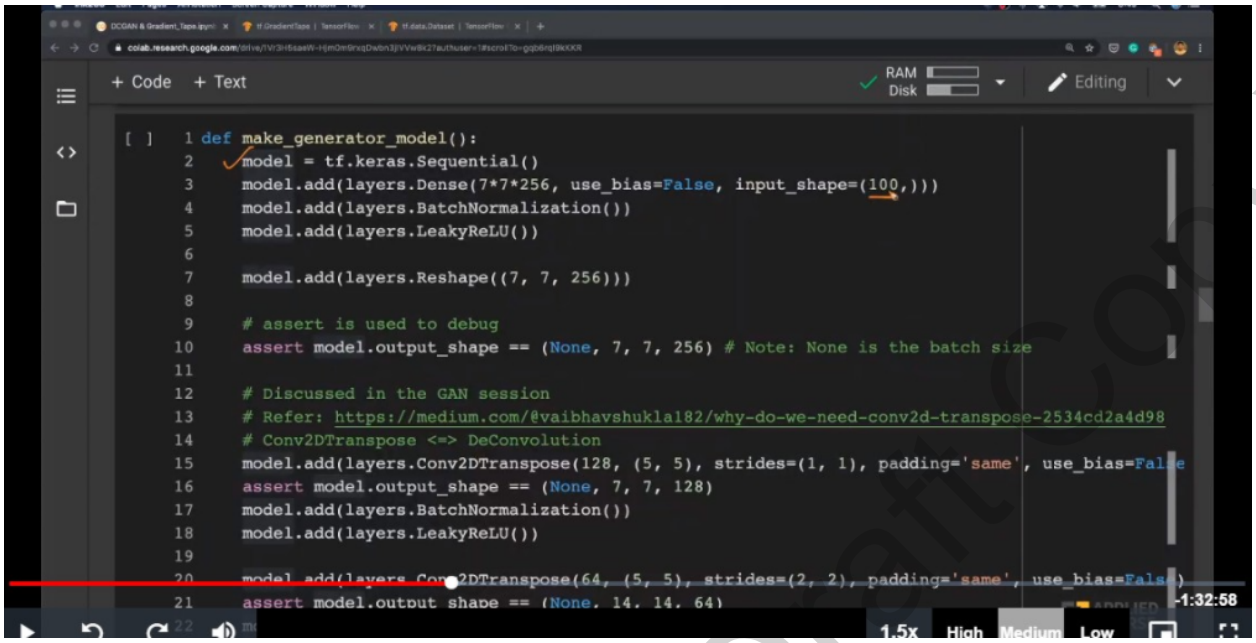
Now, we are going to define the dataset.



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.



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.

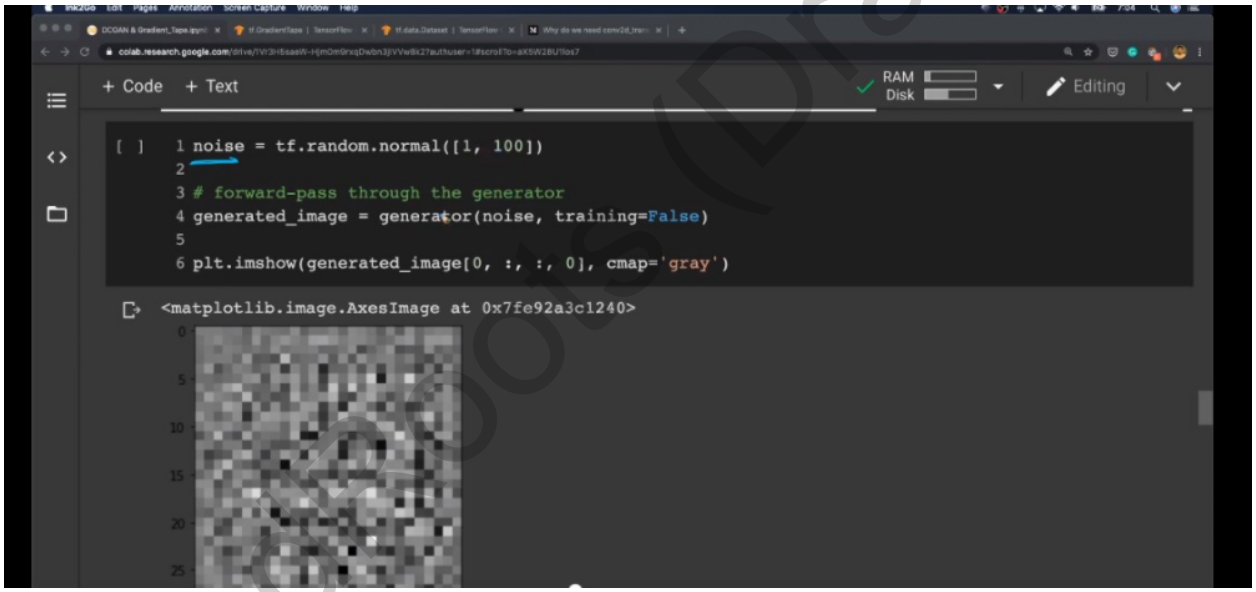
Then, we can check whether the shape of the model output is correct or not by defining an assert statement.\

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.

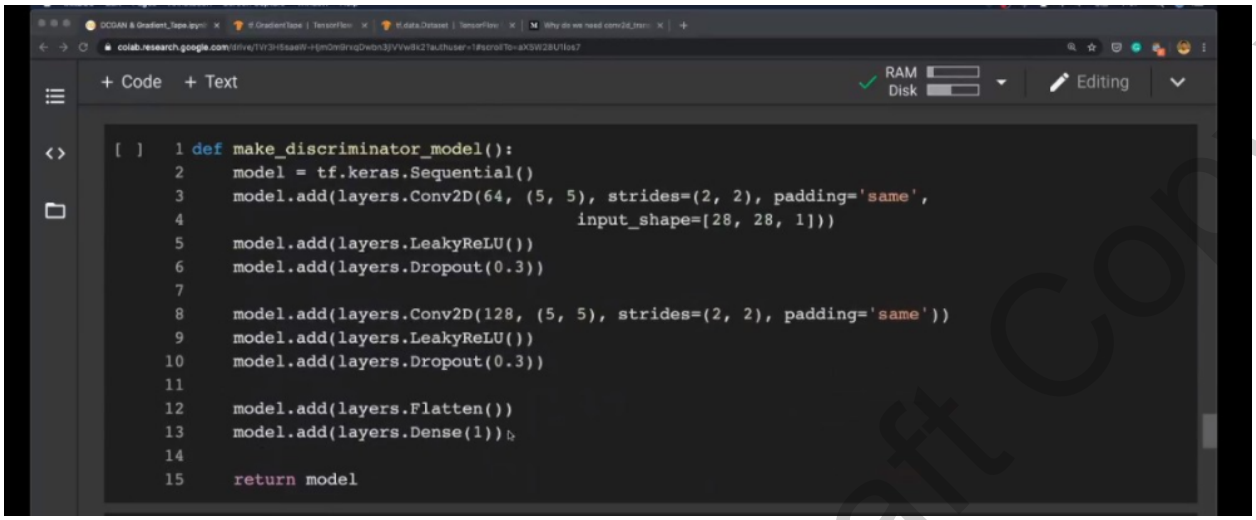


Timestamp : 01:06:49

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.



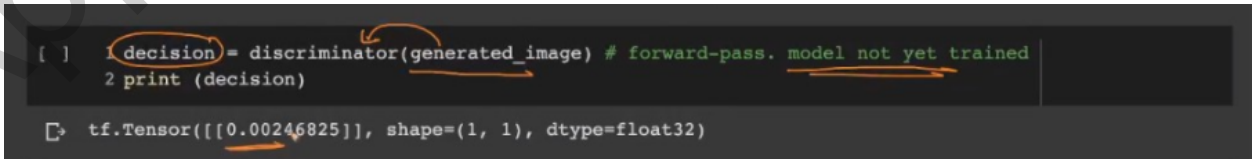
Timestamp : 01:10:04

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.

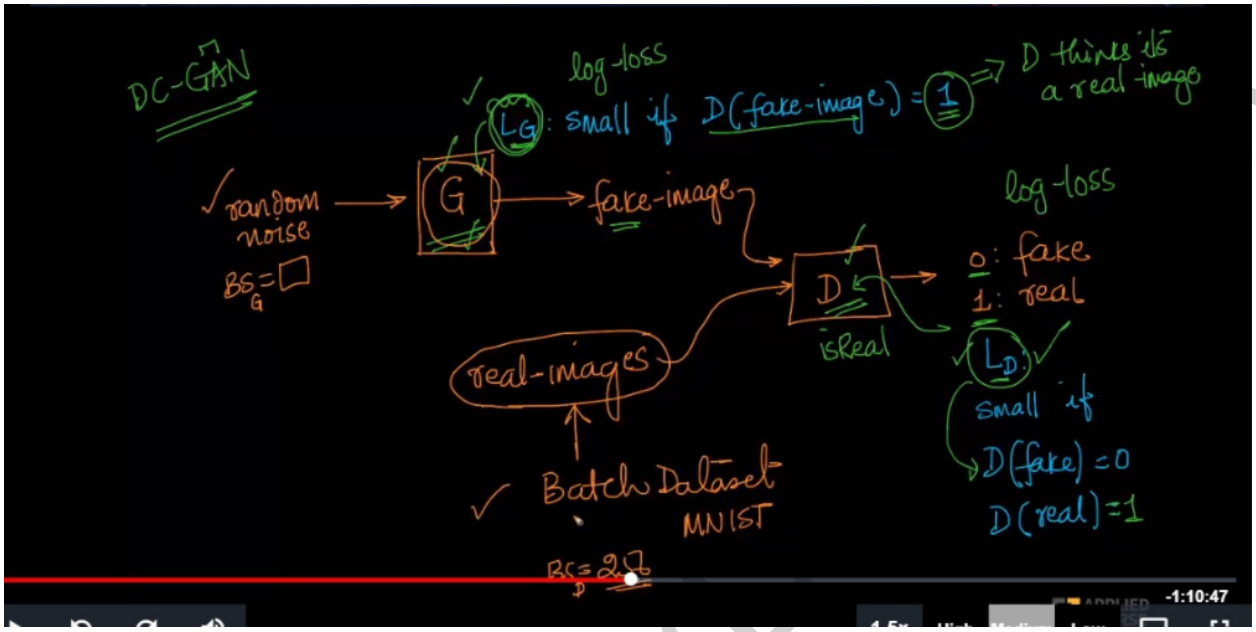
In the final layer, we defined a Dense layer with output being a single neuron.

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



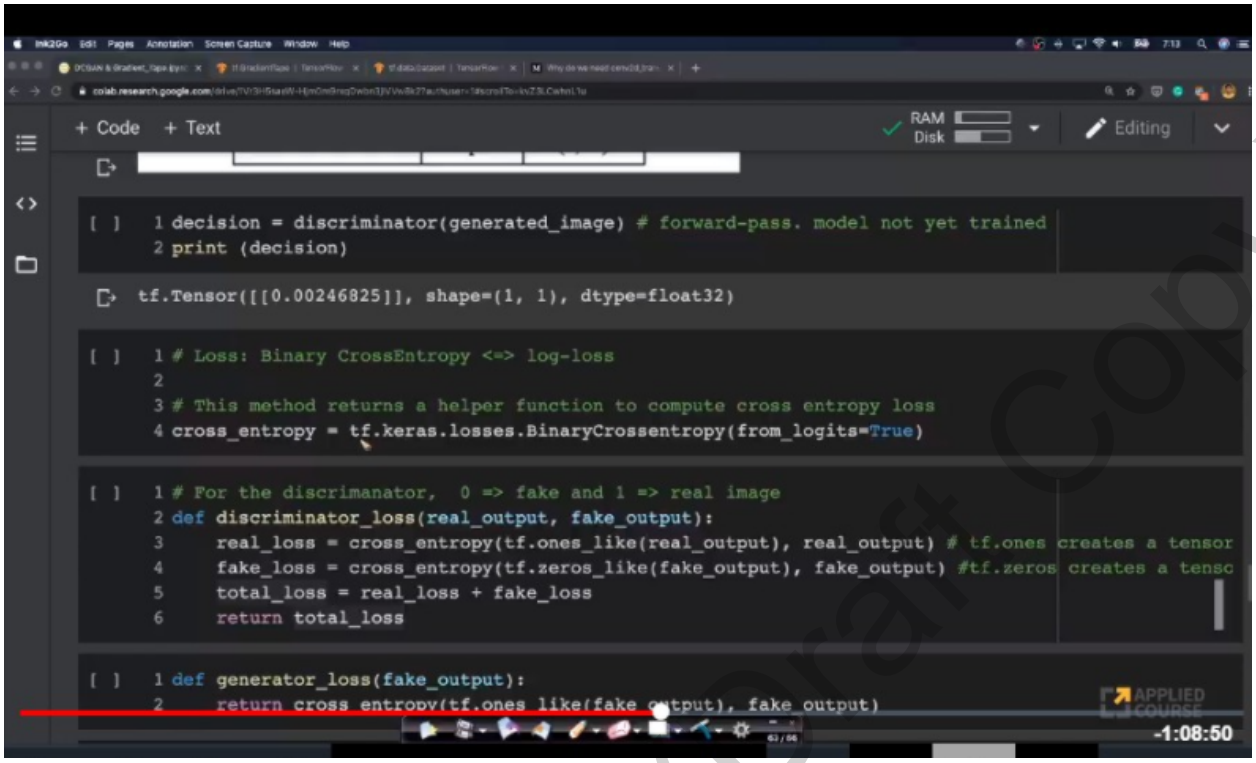
Timestamp : 01:13:24

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.



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.

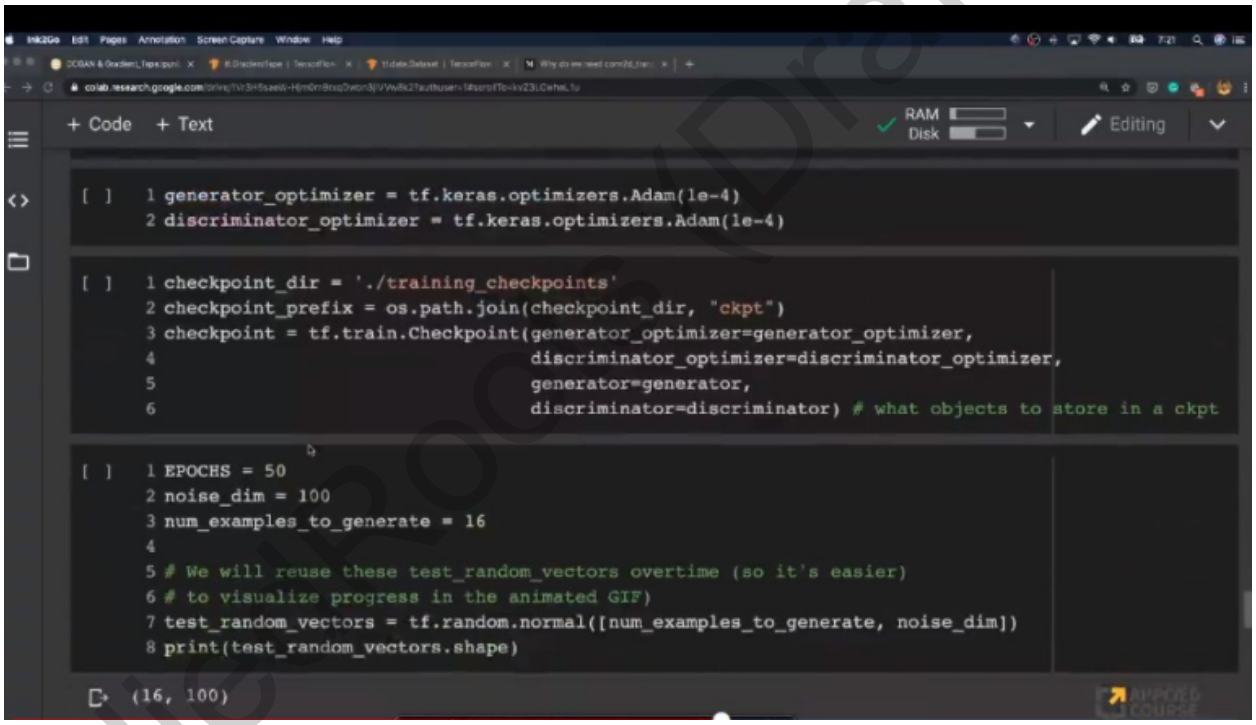
The procedure for computing the loss is easy to understand.

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.

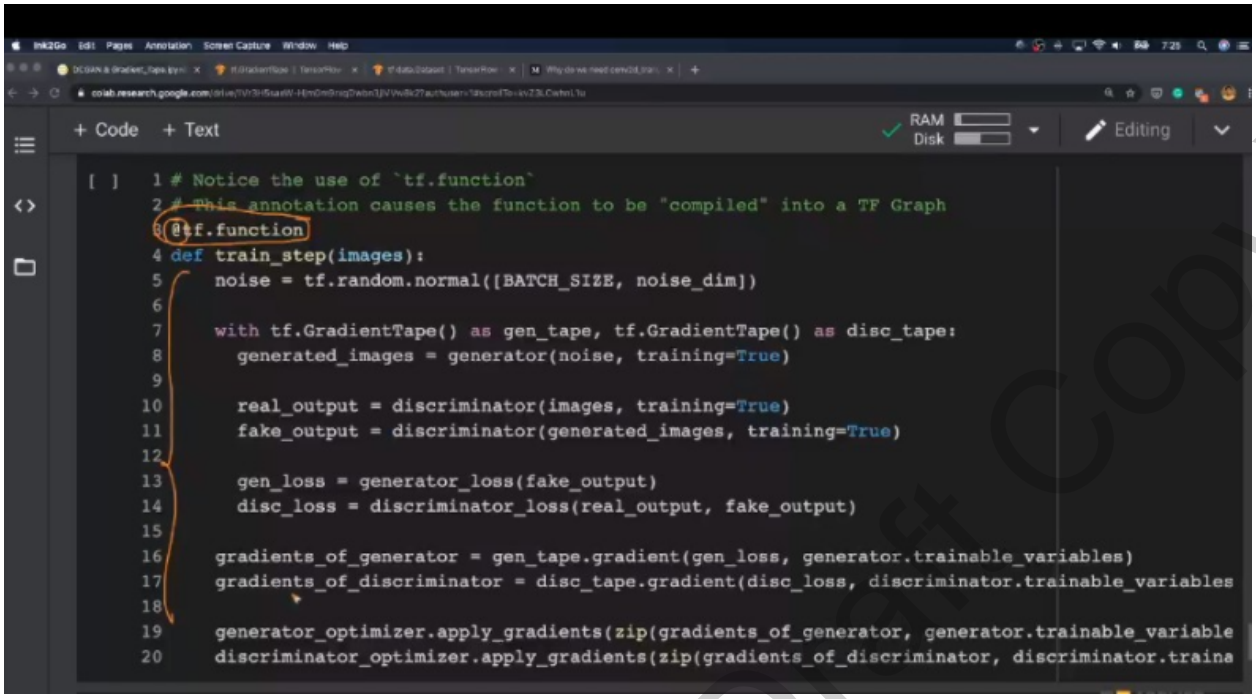


Timestamp : 01:23:41

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.



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.

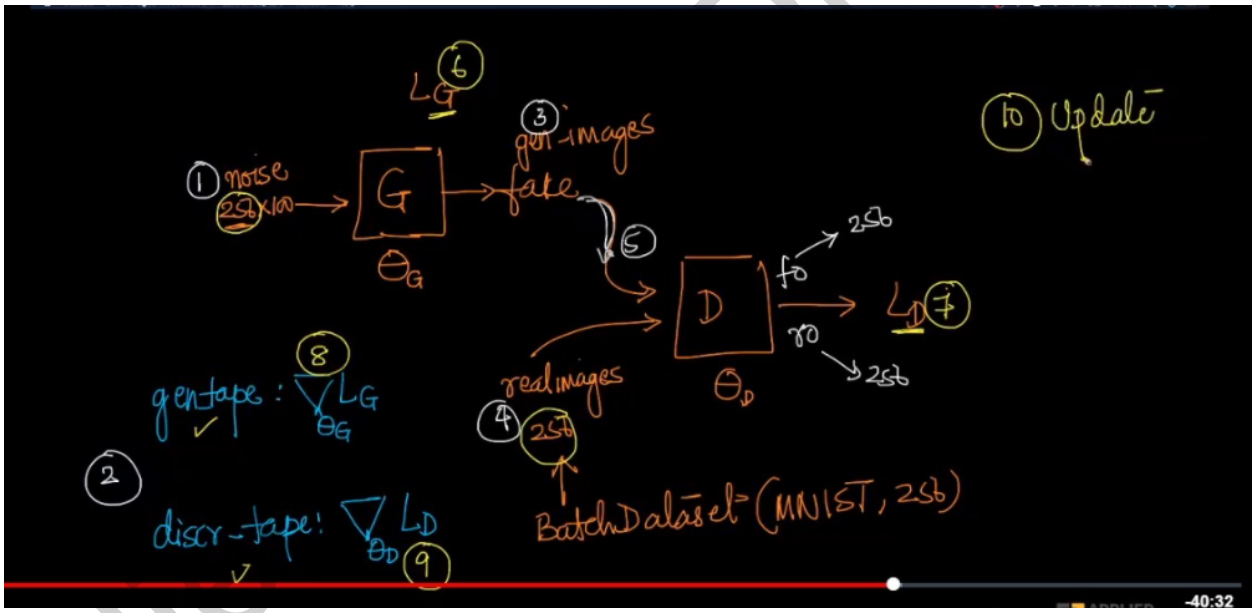
Then, we calculate both the loss based on the generator network and the discriminator network.

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

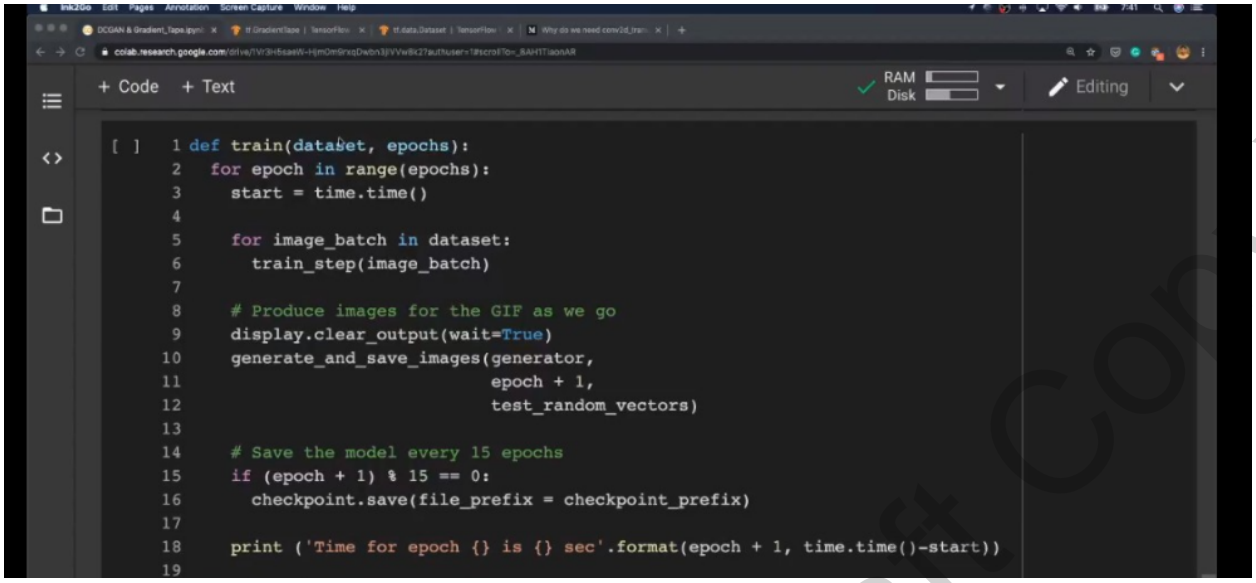
Then,using the function apply_gradients in the optimizer object we can update the trainable parameters.

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

Overview :



Timestamp : 01:44:06



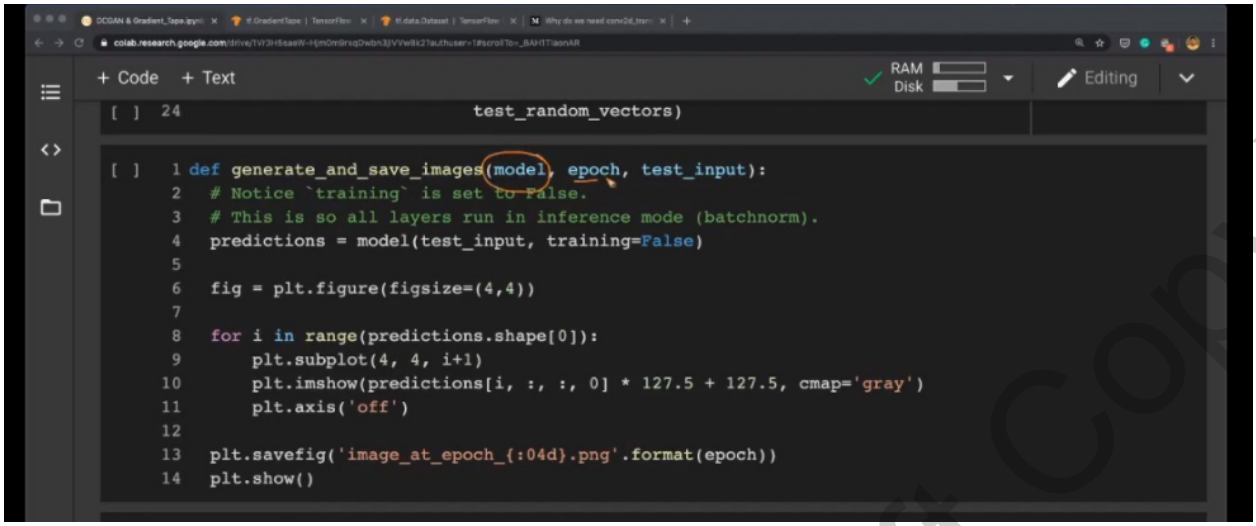
Timestamp : 01:44:30

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.



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.