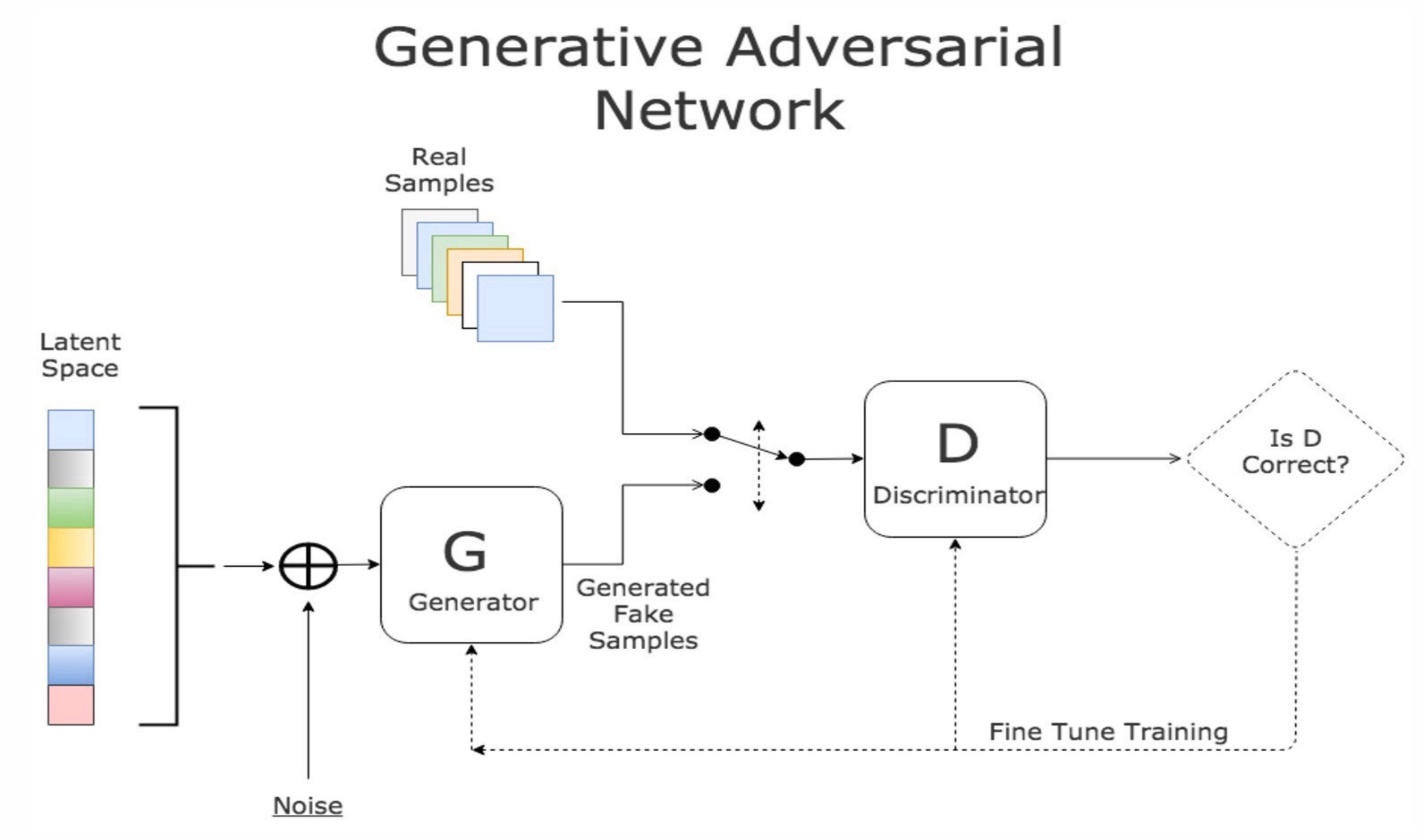


|  |
| --- |
| Deep Convolutional Generative Adversarial Network |
|  |
| April 2021  IIT Dharwad  Authored by: ISHIKA SHARMA E-Mail: ishikasharma.aug2001@gmail.com |



GOOGLE COLAB

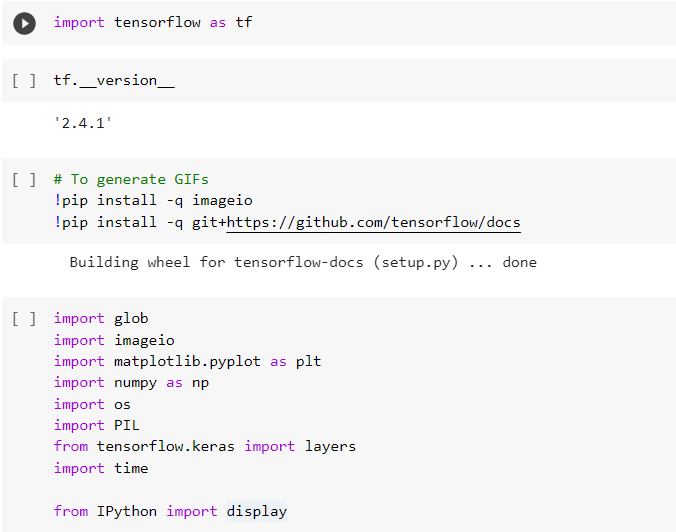
|  |
| --- |
| “DCGAN”**Deep Convolutional Generative Adversarial Network** We will use Deep Convolutional Generative Adversarial  Network (DCGAN  We will use Deep Convolutional Generative Adversarial  Network (DCGAN  We will use the potential of deep learning to generate real-like images and create a DCGAN Model MNIST Dataset. Our model will be able to generate images of handwritten digits using TensorFlow.  Before we begin, let’s have a quick introduction to GANs: What are GANs? The concept of generative adversarial networks (GANs) was introduced by [Ian Goodfellow](http://www.iangoodfellow.com/). Goodfellow uses the metaphor of an art critic and an artist to describe the two models—discriminators and generators—that make up GANs.  An art critic (the discriminator) looks at an image and tries to determine if its real or a forgery.  An artist (the generator) who wants to fool the art critic tries to make a forged image that looks as realistic as possible.  These two models “battle” each other; the discriminator uses the output of the generator as training data, and the generator gets feedback from the discriminator.  Each model becomes stronger in the process.  In this way, GANs are able to generate new complex data, based on some amount of known input data, in this case, images.  For example, Generating an image of cat:  A diagram of a generator and discriminator |
| A second diagram of a generator and discriminator  GANs have proven to be really successful in modeling and generating high dimensional data, which is why they’ve become so popular. But they’ve their own pros and cons. Some are listed below:   * They currently generate the sharpest images * They are easy to train (since no statistical inference is required), and only back-propagation is needed to obtain gradients * GANs are difficult to optimize due to unstable training dynamics. * No statistical inference can be done with them   Now we begin with our Model: Architecture of our DCGAN Here, we are not trying to mimic simple numerical data—we are trying to mimic an image, which should even be able to fool a human. The generator takes a randomly generated noise vector as input data and then uses a technique called deconvolution to transform the data into an image.  The discriminator is a classical convolutional neural network, which classifies real and fake images.  visualization of a GAN Generative Adversarial Network |



Global concept of a GAN

*****CODE:*****

1. **Importing the required Libraries:**

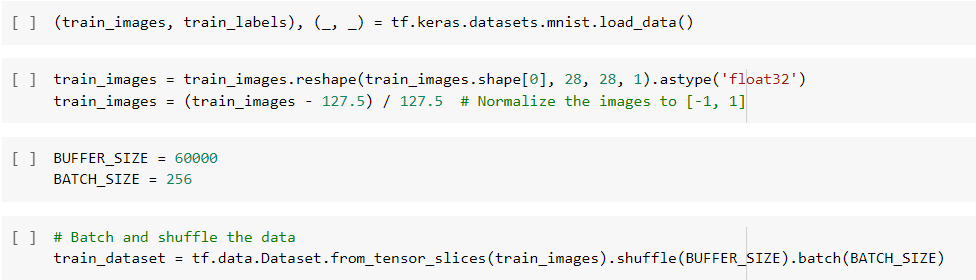


1. **Loading the dataset:**

#### ****MNIST Dataset:****

It is a dataset containing hand-written digits from 0 to 9.

Our generator will generate handwritten digits resembling the MNIST data.



1. **Creating the Model:**

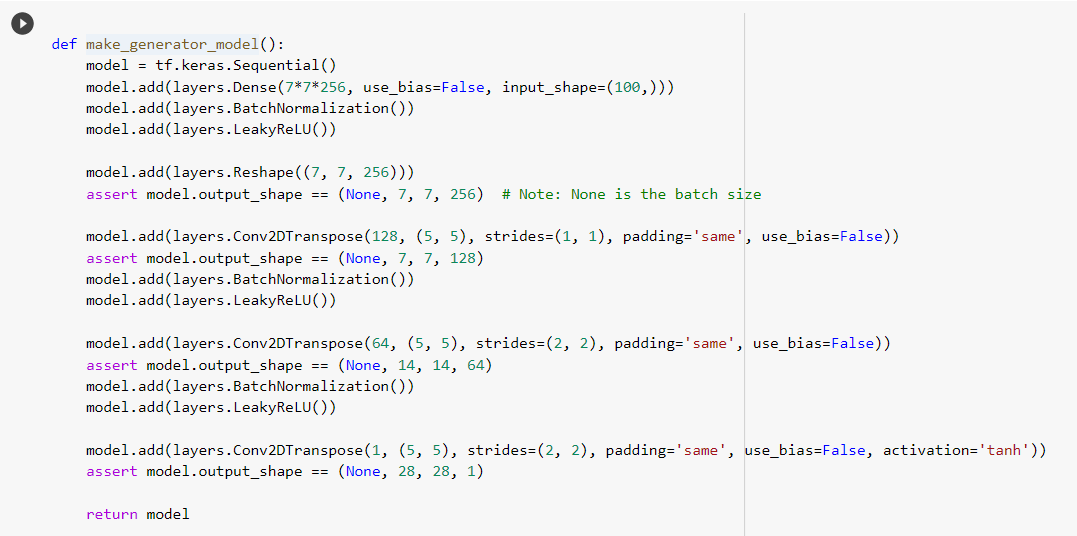
### ****GENERATOR****

It is a generative network, which generates the images, from the input of a vector of noise. We will use the following layers in our model:

* *Dense:* The noise layer of our generator.
* *Conv2DTranspose:* Upscaling and convolving the image at the same time.
* *LeakyReLU:* We use it to avoid gradient vanish.
* *BatchNormalization:* Normalizing the result of a convolution to get better results.

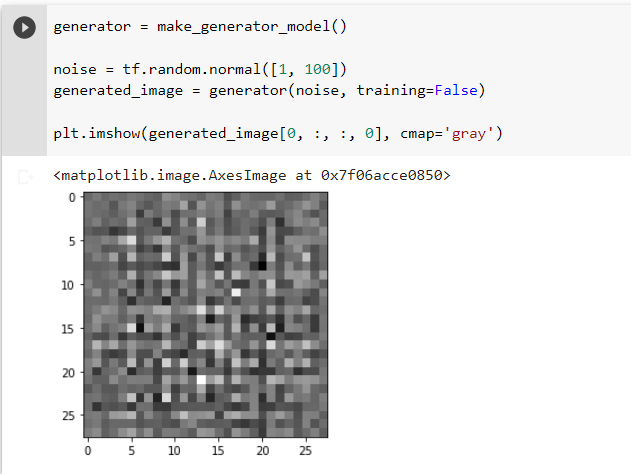
**Basic Idea**: From the random noise, we will convolve the data until we generate an image.

The generator goes the other way: It is the artist who is trying to fool the discriminator. This network consists of 3 convolutional layers. Each convolutional layer performs a convolution and then performs batch normalization and a leaky ReLu as well. Then, we return the tanh activation function.



**Checking our generator works fine:**

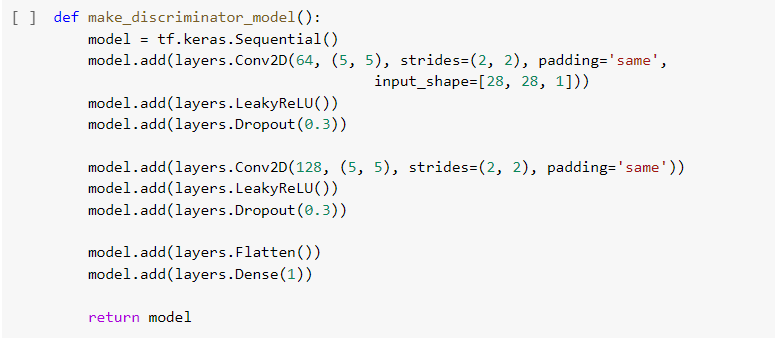
Our network is not trained yet so, it should generate only noise.



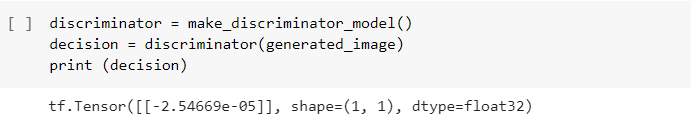
### ****DISCRIMINATOR****

Our discriminator network is a normal convolutional neural network. It will take an image as input and as output will return a binary value.

For every layer of the network, we are going to perform a convolution, then we use LeakyRelu to avoid gradient vanish, followed by a dropout layers to avoid overfitting. We complete the structure of discriminator with Flatten and Dense layers.



Our model, yet untrained so, the discriminator should classify the generated images as real or fake. The model will be trained to output positive values for real images, and negative values for fake images.

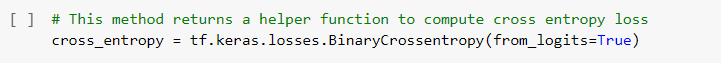


1. Define the loss and optimizers:

We need to define three loss functions:

* The loss of the generator,
* The loss of the discriminator when using real images, and
* The loss of the discriminator when using fake images.

The sum of the fake image and real image loss is the overall discriminator loss.



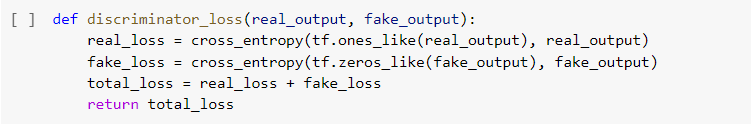
### ****Discriminator LOSS****

This method quantifies how well the discriminator is able to distinguish real images from fakes. It compares the discriminator's predictions on real images to an array of 1s, and the discriminator's predictions on fake (generated) images to an array of 0s

First, we define our loss for the real images. For that, we pass the output of the discriminator when dealing with real images and compare it with the labels, which are all 1, which means true.

Then, we define the loss for our fake images. This time we pass in the output of the discriminator when dealing with fake images and compare it to our labels, which are all 0, which means they are fake.

Lastly, our total discriminator loss is sum of the above two losses.



### ****Generator LOSS****

The generator's loss quantifies how well it was able to trick the discriminator. Intuitively, if the generator is performing well, the discriminator will classify the fake images as real (or 1). Here, compare the discriminators decisions on the generated images to an array of 1s.

For the generator loss, we do the same like we did for fake loss, but instead of comparing the output with all 0s, we compare it with 1s, since we want to fool the discriminator.

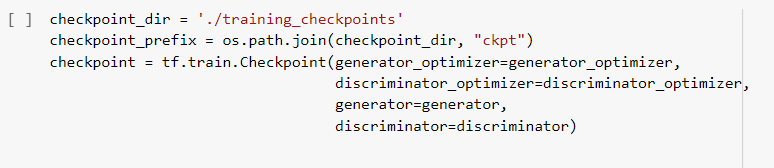


We optimize the generator and discriminator individually, since we wnat to train both the networks separately.



1. **Saving the Checkpoints**

We need to save checkpoints to reload our model back, if we accidently get disconnected.



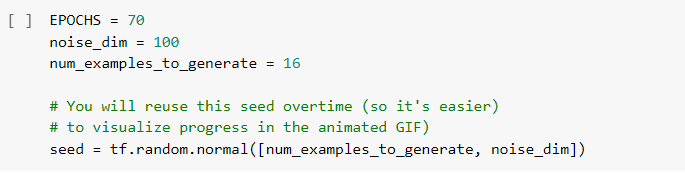
1. Training the GAN model:

Training is the hardest part and since a GAN contains two separately trained networks, its training algorithm must address two complications:

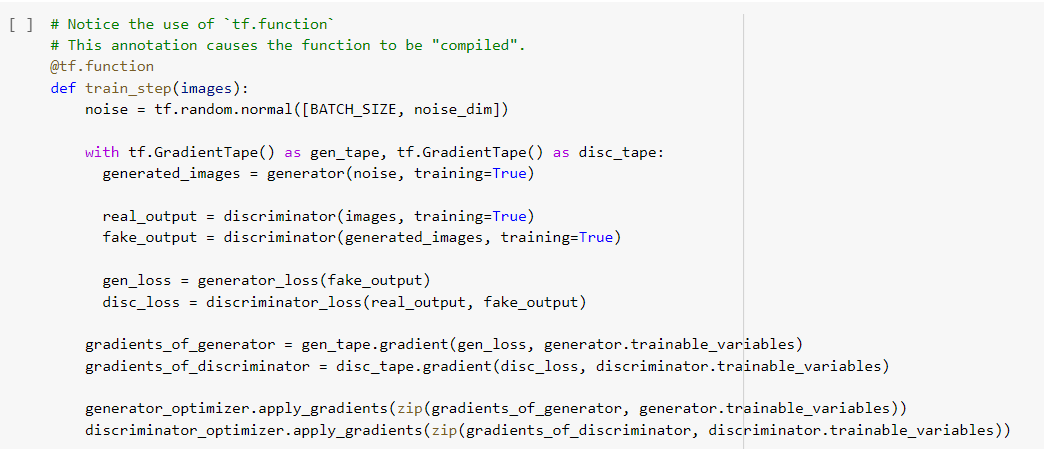
* GANs must juggle two different kinds of training (generator and discriminator).
* GAN convergence is hard to identify.

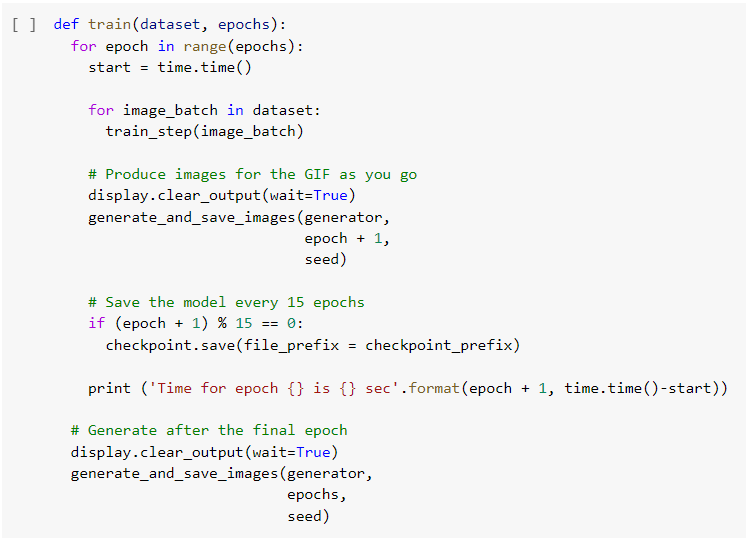
As the generator improves with training, the discriminator performance gets worse because the discriminator can’t easily tell the difference between real and fake. If the generator succeeds perfectly, then the discriminator has a 50% accuracy. In effect, the discriminator flips a coin to make its prediction.

This progression poses a problem for convergence of the GAN as a whole: the discriminator feedback gets less meaningful over time. If the GAN continues training past the point when the discriminator is giving completely random feedback, then the generator starts to train on junk feedback, and its quality may collapse.

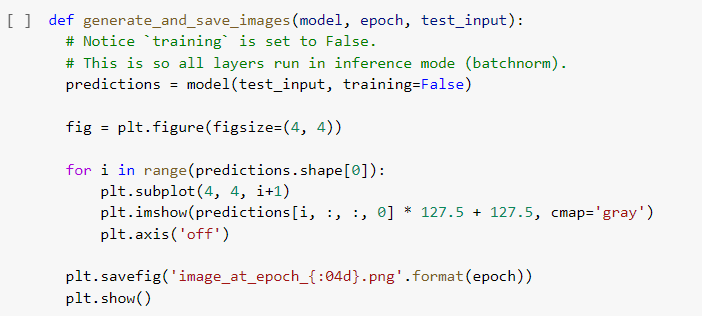


The training loop begins with generator receiving random noise as input, which is then used to produce an image. The discriminator then classifies real images (drawn from the training set) and fakes images (produced by the generator). The loss is calculated for each of these models, and the gradients are used to update the generator and discriminator.



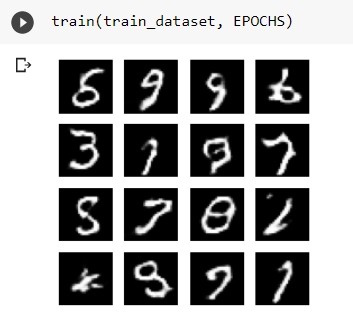


1. **Generate and Save Images:**

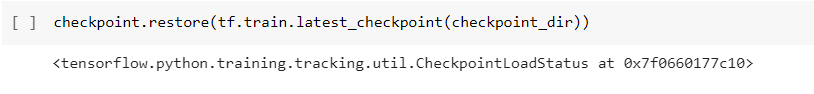


We call the train() method defined above to train the generator and discriminator simultaneously. Note, training GANs can be tricky. It's important that the generator and discriminator do not overpower each other (e.g., that they train at a similar rate).

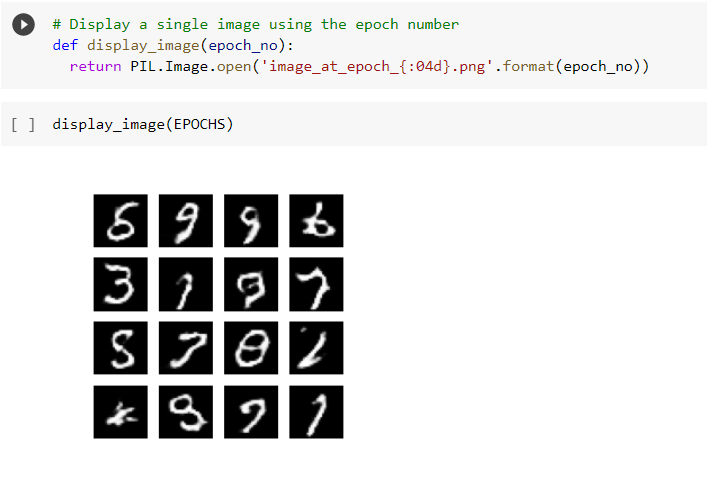
At the beginning of the training, the generated images look like random noise. As training progresses, the generated digits will look increasingly real. After about 50 epochs, they resemble MNIST digits. This may take about one minute / epoch with the default settings on Colab.

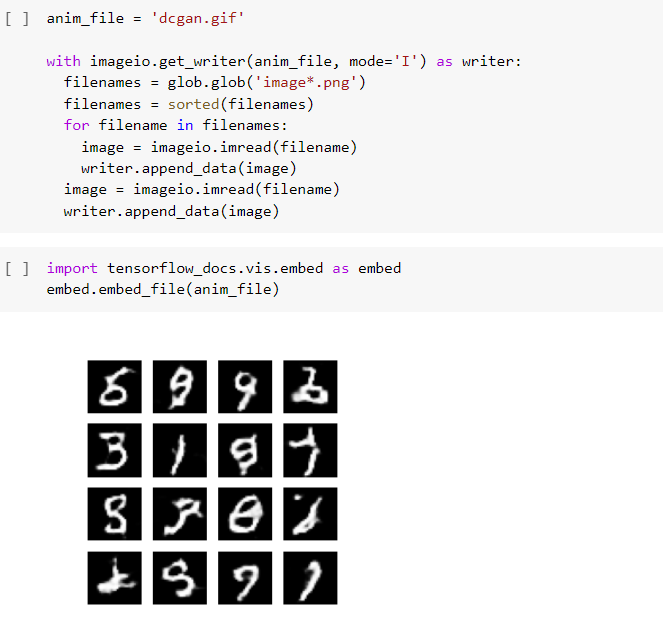


**Restore the latest checkpoint**



1. **Create GIF:**

****

****

**REFERENCES:**

* Code is taken from [https://www.tensorflow.org/tutorials/generative/dcgan]