A PROJECT REPORT

Generative Adversarial Networks (GANs)

Art of Research and

Publication

EXC1002

Under the guidance of

Prof. Akella

Sivaramakrishna

Department of Chemistry

School of Advanced Sciences

VIT Vellore

Submitted by:

Bevis Mathew – 18BCE2163

Meet Bhatnagar – 18BCE0683

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

DECEMBER, 2020

ABSTRACT

Generative Adversarial Networks (GANs) are one of the most interesting ideas in computer science today. Two models are trained simultaneously by an adversarial process. A generator learns to create images that look real, while a discriminator learns to tell real images apart from fakes. A generative adversarial network (GAN) is a class of machine learning systems invented by Ian Goodfellow and his colleagues in 2014. Two neural networks contest with each other in a game (in the sense of game theory, often but not always in the form of a zero-sum game). Given a training set, this technique learns to generate new data with the same statistics as the training set. For example, a GAN trained on photographs can generate new photographs that look at least superficially authentic to human observers, having many realistic characteristics. Though originally proposed as a form of generative model for unsupervised learning, GANs have also proven useful for semi-supervised learning, fully supervised learning, and reinforcement learning. The network itself generated these images and they look very convincing. If you start looking
closely you can tell that certain things are off. But this is really incredible work from the latest results of Nvidia. And these networks basically compete against each other. So the classic analogy for GANs is the forger versus the detective.
So you have a forger trying to forge paintings and make them real enough that the detective won't notice and you can think of the forger as a generator and the detective as the discriminator. And basically, what happens is the generator makes fake data to pass discriminator So the discriminator also sees real data that is real images. So, for example real images of paintings or real images of faces. And it predicts if the data received is real or fake and the generator is being trained to try to fool the discriminator it wants the output data to look as close as possible to real data and the discriminator is trained to figure out which data is real and which data is fake and what ends up happening is that the generator learns to make data that's indistinguishable from your old data to this cremator.

Table of Contents

1. AIM AND OBJECTIVE

Aim:

The aim of the project is to create a model to detect the difference between the fake and real images. A generator learns to create images that look real, while a discriminator learns to tell real images apart from fakes.

Objective:

During training, the generator progressively becomes better at creating images that look real, while the discriminator becomes better at telling them apart. The process reaches equilibrium when the discriminator can no longer distinguish real images from fakes.

This project demonstrates the process on the MNIST dataset. The output shows a series of images produced by the generator as it was trained for 50 epochs. The images begin as random noise, and increasingly resemble hand written digits over time.

One neural network, called the generator, generates new data instances, while the other, the discriminator, evaluates them for authenticity; i.e. the discriminator decides whether each instance of data that it reviews belongs to the actual training dataset or not.

Let’s say we’re trying to do something more banal than mimic the Mona Lisa. We’re going to generate hand-written numerals like those found in the MNIST dataset, which is taken from the real world. The goal of the discriminator, when shown an instance from the true MNIST dataset, is to recognize those that are authentic.

Meanwhile, the generator is creating new, synthetic images that it passes to the discriminator. It does so in the hopes that they, too, will be deemed authentic, even though they are fake. The goal of the generator is to generate passable hand-written digits: to lie without being caught. The goal of the discriminator is to identify images coming from the generator as fake.

Here are the steps a GAN takes:

• The generator takes in random numbers and returns an image.

• This generated image is fed into the discriminator alongside a stream of images taken from the actual, ground-truth dataset.

• The discriminator takes in both real and fake images and returns probabilities, a number between 0 and 1, with 1 representing a prediction of authenticity and 0 representing fake.

So we have a double feedback loop:

• The discriminator is in a feedback loop with the ground truth of the images, which we know.

• The generator is in a feedback loop with the discriminator.

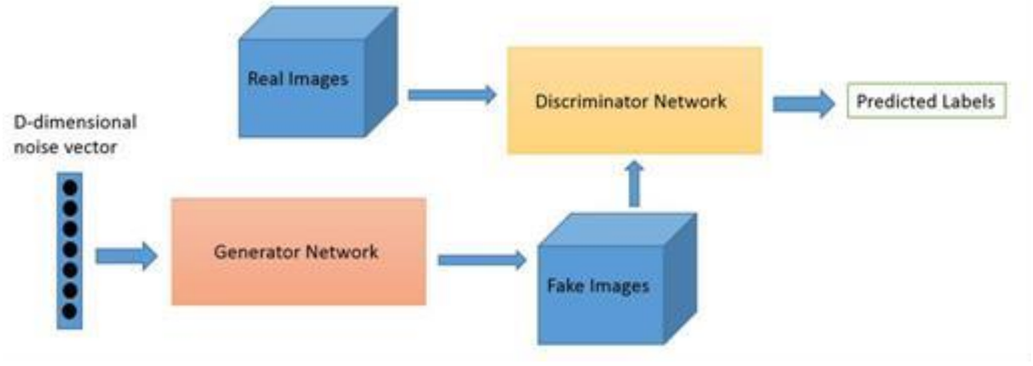


Figure 1: Structural Flowchart

We can think of a GAN as the opposition of a counterfeiter and a cop in a game of cat and mouse, where the counterfeiter is learning to pass false notes, and the cop is learning to detect them. Both are dynamic; i.e. the cop is in training, too (to extend the analogy, maybe the central bank is flagging bills that slipped through), and each side comes to learn the other's methods in a constant escalation.

For MNIST, the discriminator network is a standard convolutional network that can categorize the images fed to it, a binomial classifier labelling images as real or fake. The generator is an inverse convolutional network, in a sense: While a standard convolutional classifier takes an image and down samples it to produce a probability, the generator takes a vector of random noise and up samples it to an image. The first throws away data through down sampling techniques like maxpooling, and the second generates new data.

Both nets are trying to optimize a different and opposing objective function, or loss function, in a zero-zum game. This is essentially an actor-critic model. As the discriminator changes its behaviour, so does the generator, and vice versa. Their losses push against each other.

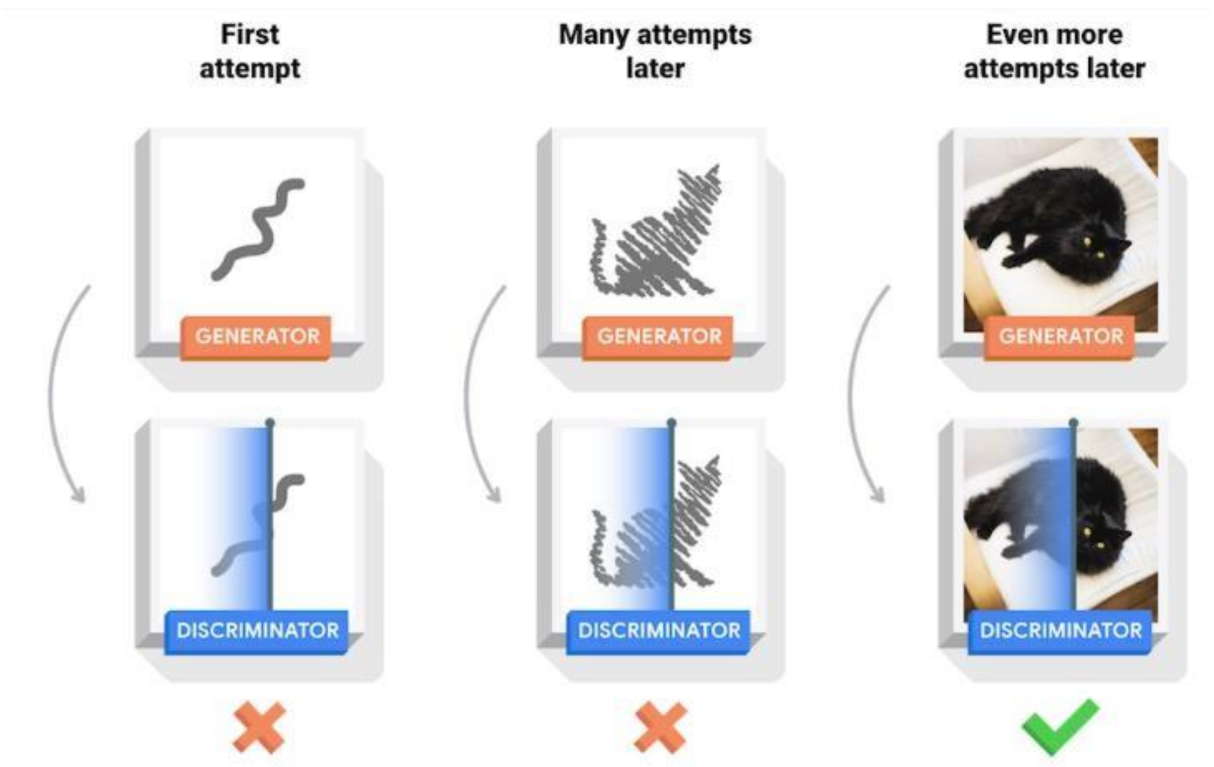


Figure 2: Training Process Representation

When you train the discriminator, hold the generator values constant; and when you train the generator, hold the discriminator constant. Each should train against a static adversary. For example, this gives the generator a better read on the gradient it must learn by.

By the same token, pretraining the discriminator against MNIST before you start training the generator will establish a clearer gradient.

Each side of the GAN can overpower the other. If the discriminator is too good, it will return values so close to 0 or 1 that the generator will struggle to read the gradient. If the generator is too good, it will persistently exploit weaknesses in the discriminator that lead to false negatives. This may be mitigated by the nets' respective learning rates. The two neural networks must have a similar "skill level."
GANs take a long time to train. On a single GPU a GAN might take hours, and on a single CPU more than a day. While difficult to tune and therefore to use, GANs have stimulated a lot of interesting research and writing.

2. HARDWARE AND SOFTWARE SPECIFICATION

Hardware Requirements:

a. Intel i3-7100U CPU 7th Gen Processor
b. Clock speed @2.40Ghz
c. Ram 8.00GB
d. Nvidia GeForce GTX 1050

Software Requirements:

a. Microsoft Windows 10 Education 64-bit or Ubuntu 18.04 64bit
b. Python 3.8
c. Anaconda or Spyder or Visual Code or Pycharm or Google collab.

3. ARCHITECTURE

The architecture is simple that we train two models that is generator and discriminator in which the generator takes in random numbers and returns an image. This generated image is fed into the discriminator alongside a stream of images taken from the actual, ground-truth dataset. The discriminator takes in both real and fake images and returns probabilities, a number between 0 and 1, with 1 representing a prediction of authenticity and 0 representing fake. So, you have a double feedback loop the discriminator is in a feedback loop with the ground truth of the images, which we know. The generator is in a feedback loop with the discriminator. A simple demonstration is shown in *Figure 3.*

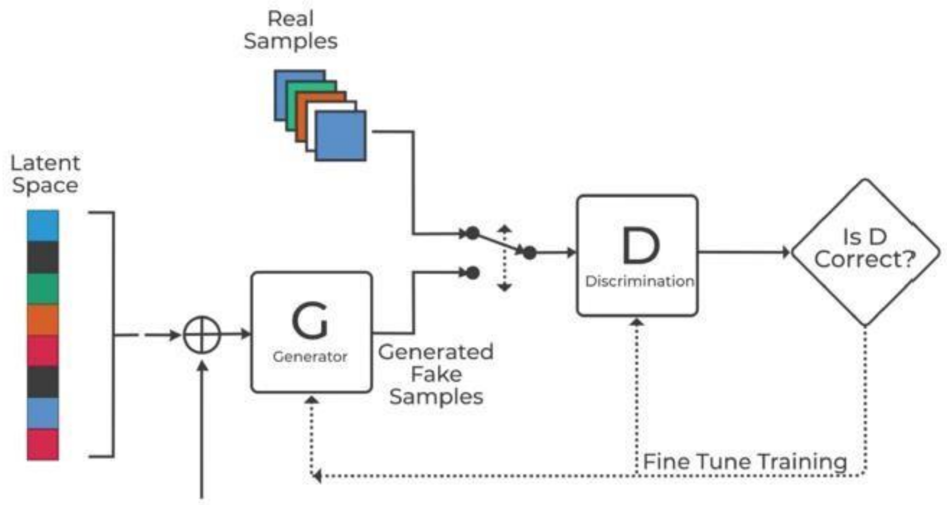


Figure 3: Architecture Diagram

3.1 Modules

Both the generator module and discriminator module are defined using the Keras Sequential API.

Generator Module:

Creates dataset to check.
The generator uses tf.keras.layers.Conv2DTranspose (upsampling) layers to produce an image from a seed (random noise). Start with a Dense layer that takes this seed as input, then upsample several times until you reach the desired image size of 28x28x1.

Discriminator Module:

The discriminator is a CNN-based image classifier.
Evaluates them for authenticity; i.e. the discriminator decides whether each instance of data that it reviews belongs to the actual training dataset or not.

Discriminator Loss Module:

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.

Generator Loss Module:

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, we will compare the discriminators decisions on the generated images to an array of 1s.

Save Checkpoints:

How to save and restore models, which can be helpful in case a long running training task is interrupted.

Training Loop Module:

The training loop begins with generator receiving a random seed as input. That seed is used to produce an image. The discriminator is then used to classify 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.

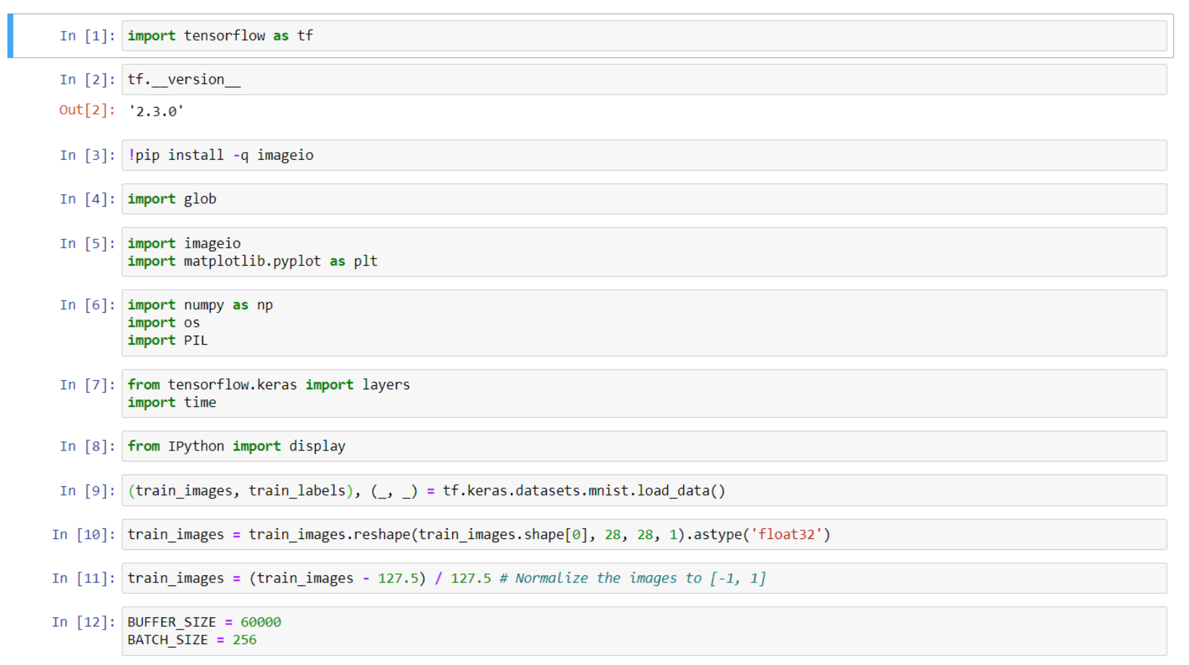
Train Module:

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.

4. EXPERIMENTAL ANALYSIS

4.1 Implementation codes and results



```
In [1]: import tensorflow as tf

In [2]: tf.__version__
Out[2]: '2.3.0'

In [3]: !pip install -q imageio

In [4]: import glob

In [5]: import imageio
        import matplotlib.pyplot as plt

In [6]: import numpy as np
        import os
        import PIL

In [7]: from tensorflow.keras import layers
        import time

In [8]: from IPython import display

In [9]: (train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()

In [10]: train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).astype('float32')

In [11]: train_images = (train_images - 127.5) / 127.5 # Normalize the images to [-1, 1]

In [12]: BUFFER_SIZE = 60000
         BATCH_SIZE = 256
```

```
In [13]: train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

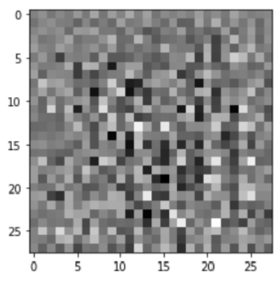
```
In [14]: def make_generator_model():
             model = tf.keras.Sequential()
             model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
             model.add(layers.BatchNormalization())
             model.add(layers.LeakyReLU())

             model.add(layers.Reshape((7, 7, 256)))
             assert model.output_shape == (None, 7, 7, 256) # Note: None is the batch size

             model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
             assert model.output_shape == (None, 7, 7, 128)
             model.add(layers.BatchNormalization())
             model.add(layers.LeakyReLU())

             model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
             assert model.output_shape == (None, 14, 14, 64)
             model.add(layers.BatchNormalization())
             model.add(layers.LeakyReLU())

             model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation='tanh'))
             assert model.output_shape == (None, 28, 28, 1)

             return model
```

```
In [15]: generator = make_generator_model()
```

```
In [16]: noise = tf.random.normal([1, 100])
```

```
In [17]: generated_image = generator(noise, training=False)
```

```
In [18]: plt.imshow(generated_image[0, :, :, 0], cmap='gray')
Out[18]: <matplotlib.image.AxesImage at 0x23b42701eb0>
```



```
In [19]: def make_discriminator_model():
             model = tf.keras.Sequential()
             model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                                              input_shape=[28, 28, 1]))
             model.add(layers.LeakyReLU())
             model.add(layers.Dropout(0.3))

             model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
             model.add(layers.LeakyReLU())
             model.add(layers.Dropout(0.3))

             model.add(layers.Flatten())
             model.add(layers.Dense(1))

             return model
```

```
In [20]: discriminator = make_discriminator_model()
```

```
In [21]: decision = discriminator(generated_image)
```

```
In [22]: print (decision)
         tf.Tensor([[0.001626]], shape=(1, 1), dtype=float32)
```

```
In [23]: cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

```
In [24]: def discriminator_loss(real_output, fake_output):
             real_loss = cross_entropy(tf.ones_like(real_output), real_output)
             fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
             total_loss = real_loss + fake_loss
             return total_loss
```

```
In [25]: def generator_loss(fake_output):
             return cross_entropy(tf.ones_like(fake_output), fake_output)
```

```
In [26]: generator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

```
In [27]: discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

```
In [28]: checkpoint_dir = './training_checkpoints'
         checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
         checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                          discriminator_optimizer=discriminator_optimizer,
                                          generator=generator,
                                          discriminator=discriminator)
```

```
In [29]: checkpoint

Out[29]: <tensorflow.python.training.tracking.util.Checkpoint at 0x23b4794c6a0>
```

```
In [30]: EPOCHS = 50
         noise_dim = 100
         num_examples_to_generate = 16
```

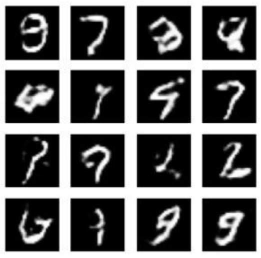
```
In [31]: seed = tf.random.normal([num_examples_to_generate, noise_dim])
```

```
In [32]: @tf.function
         def train_step(images):
             noise = tf.random.normal([BATCH_SIZE, noise_dim])

             with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
               generated_images = generator(noise, training=True)

               real_output = discriminator(images, training=True)
               fake_output = discriminator(generated_images, training=True)

               gen_loss = generator_loss(fake_output)
               disc_loss = discriminator_loss(real_output, fake_output)

             gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
             gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)

             generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
             discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))
```

```
In [33]: def train(dataset, epochs):
           for epoch in range(epochs):
             start = time.time()

             for image_batch in dataset:
               train_step(image_batch)

             # Produce images for the GIF as we go
             display.clear_output(wait=True)
             generate_and_save_images(generator,
                                      epoch + 1,
                                      seed)

             # Save the model every 15 epochs
             if (epoch + 1) % 15 == 0:
               checkpoint.save(file_prefix = checkpoint_prefix)

             print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

           # Generate after the final epoch
           display.clear_output(wait=True)
           generate_and_save_images(generator,
                                    epochs,
                                    seed)
```

```
In [34]: def generate_and_save_images(model, epoch, test_input):
           # Notice `training` is set to False.
           # This is so all layers run in inference mode (batchnorm).
           predictions = model(test_input, training=False)

           fig = plt.figure(figsize=(4,4))

           for i in range(predictions.shape[0]):
               plt.subplot(4, 4, i+1)
               plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
               plt.axis('off')

           plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
           plt.show()
```

```
In [35]: train(train_dataset, EPOCHS)
```

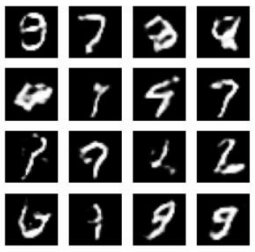


```
In [36]: checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))

Out[36]: <tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x23b59d60220>
```

```
In [37]: # Display a single image using the epoch number
         def display_image(epoch_no):
           return PIL.Image.open('image_at_epoch_{:04d}.png'.format(epoch_no))
```

```
In [38]: display_image(EPOCHS)
```

Out[38]:



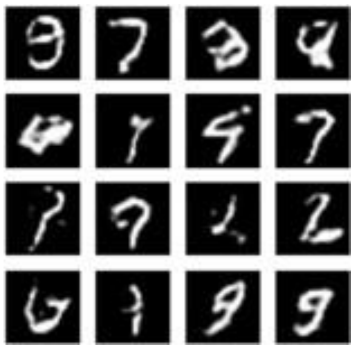
```
In [41]: anim_file = 'dcgan.gif'

         with imageio.get_writer(anim_file, mode='I') as writer:
           filenames = glob.glob('image*.png')
           filenames = sorted(filenames)
           for filename in filenames:
             image = imageio.imread(filename)
             writer.append_data(image)
           image = imageio.imread(filename)
           writer.append_data(image)
```

```
In [45]: conda install git

Collecting package metadata (current_repodata.json): ...working... done
Solving environment: ...working... done

## Package Plan ##

  environment location: C:\Users\MeetDarkPow\anaconda3

  added / updated specs:
    - git


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    conda-4.9.0                |           py38_0         2.9 MB
    git-2.23.0                 |       h6bb4b03_0        10.5 MB
    ------------------------------------------------------------
                                           Total:        13.4 MB

The following NEW packages will be INSTALLED:

  git                pkgs/main/win-64::git-2.23.0-h6bb4b03_0

The following packages will be UPDATED:

  conda                                       4.8.3-py38_0 --> 4.9.0-py38_0



Downloading and Extracting Packages

Note: you may need to restart the kernel to use updated packages.

git-2.23.0           | 10.5 MB   |            |   0% 
git-2.23.0           | 10.5 MB   |            |   0% 
git-2.23.0           | 10.5 MB   | 2          |   2% 
git-2.23.0           | 10.5 MB   | #          |  11% 
git-2.23.0           | 10.5 MB   | ###1       |  32% 
git-2.23.0           | 10.5 MB   | ###9       |  39% 
git-2.23.0           | 10.5 MB   | #####      |  51% 
git-2.23.0           | 10.5 MB   | ######6    |  66% 
git-2.23.0           | 10.5 MB   | ########## | 100% 

conda-4.9.0          | 2.9 MB    |            |   0% 
conda-4.9.0          | 2.9 MB    | ##3        |  24% 
conda-4.9.0          | 2.9 MB    | ########## | 100% 
Preparing transaction: ...working... done
Verifying transaction: ...working... done
Executing transaction: ...working... done
```

```
In [46]: !pip install -q git+https://github.com/tensorflow/docs

In [47]: import tensorflow_docs.vis.embed as embed

In [48]: embed.embed_file(anim_file)
Out[48]:
```

```
In [ ]:
```

4.2. RESULT ANALYSIS



During training, the generator progressively becomes better at creating images that look real, while the discriminator becomes better at telling them apart. The process reaches equilibrium when the discriminator can no longer distinguish real images from fakes.

This notebook demonstrates this process on the MNIST dataset. The following animation shows a series of images produced by the generator as it was trained for 50 epochs. The images begin as random noise, and increasingly resemble hand written digits over time.

5. CONCLUSION

Generative Adversarial Networks (GANs) are one of the most interesting ideas in computer science today. Two models are trained simultaneously by an adversarial process. A *generator* ("the artist") learns to create images that look real, while a *discriminator* ("the art critic") learns to tell real images apart from fakes.

GANs’ potential is huge, because they can learn to mimic any distribution of data. That is, GANs can be taught to create worlds eerily similar to our own in any domain: images, music, speech, prose. They are robot artists in a sense, and their output is impressive – poignant even.

6. REFERENCES

[1] Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. CoRR, abs/1511.06434.

[2] Ian J. Goodfellow∗, Jean Pouget-Abadie†, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair‡, Aaron Courville, Yoshua Bengio Universite de Montr ´ eal ´ Montreal, QC H3C 3J7

[3] Self-Attention Generative Adversarial Networks Han Zhang Ian Goodfellow Dimitris Metaxas Augustus Odena (Submitted on 21 May 2018 (v1), last revised 14 Jun 2019 (this version, v2))