Image Generation using Generative Adversarial Networks (GANs)

Paper Link - https://arxiv.org/abs/1406.2661

Group Number - 39

Shivam Gupta	2014A7PS066P
Rahul Banerji	2014A7PS082P
Mayank Agarwal	2014A7PS111P
Karan Deep Batra	2014A7PS160P

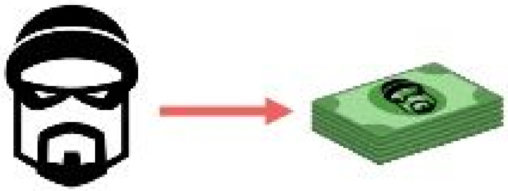
Team Contribution

- GAN Discriminator Model - Shivam Gupta
- GAN Generator Model - Karan Deep Batra
- GAN training - Mayank Agarwal, Karan Deep Batra
- Graph Analysis for GAN - Mayank Agarwal
- CGAN Discriminator Model - Rahul Banerji
- CGAN Generator Model - Mayank Agarwal
- CGAN training - Shivam Gupta, Rahul Banerji

What are GANs?

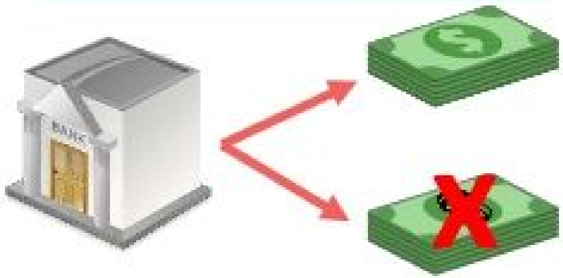
First, an intuition

generator



Goal: produce counterfeit money that is as similar as real money.

discriminator



Goal: distinguish between real and counterfeit money.

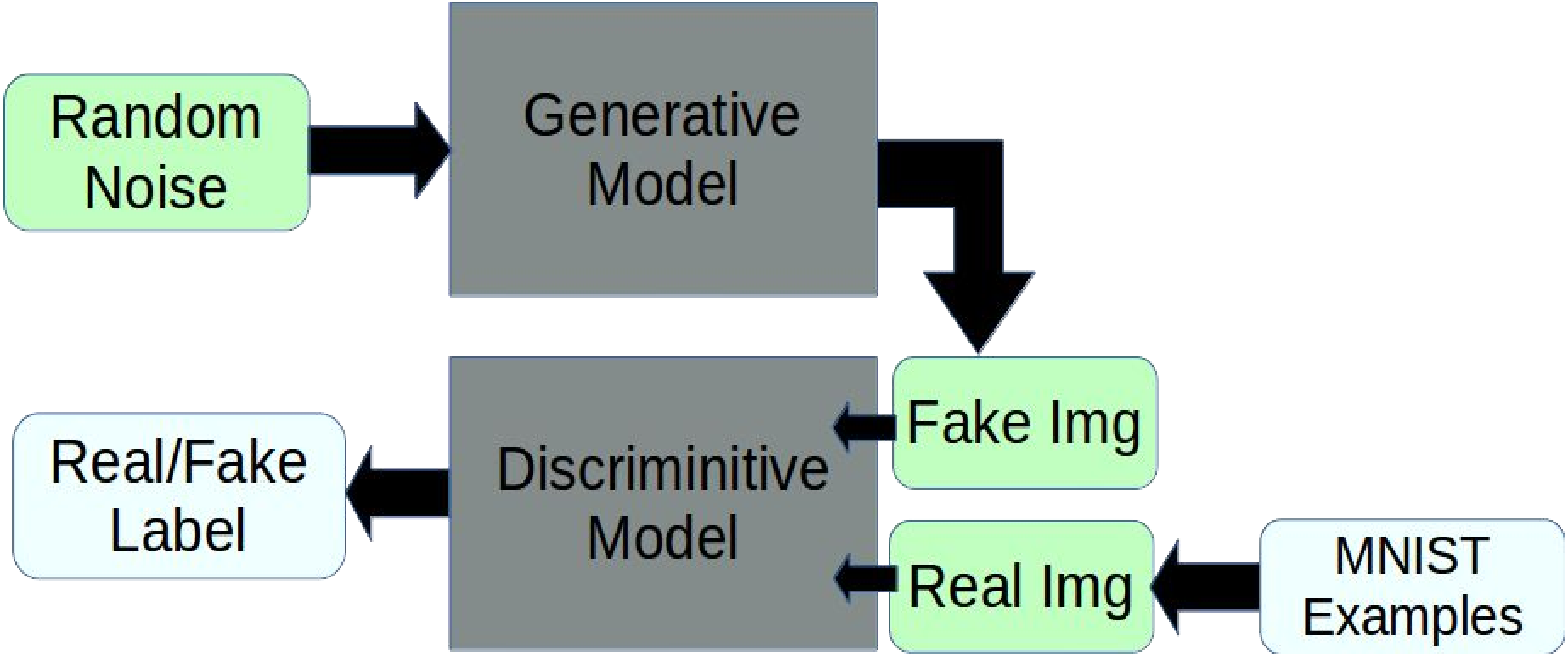
Generative Adversarial Networks

Problem: Want to sample from complex, high-dimensional training distribution. No direct way to do this!

Solution: Sample from a simple distribution, e.g. random noise. Learn transformation to training distribution.

We can use Neural Networks to represent this complex transformation

Framework for creating Generative models



Training GANs: Two-player game

Generator Network: Try to fool the discriminator by generating real-looking images

Discriminator Network: Try to distinguish the real and fake images.

Minimax Objective Function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Discriminator wants to maximize objective such that D(x) is close to 1 and D(G(z)) is close to 0

Generator wants to minimize objective such that D(G(z)) is close to 1 (i.e. Discriminator is fooled into thinking generated G(z) is real)

Training GANs: Two-player game

Alternate between:

Gradient ascent on discriminator

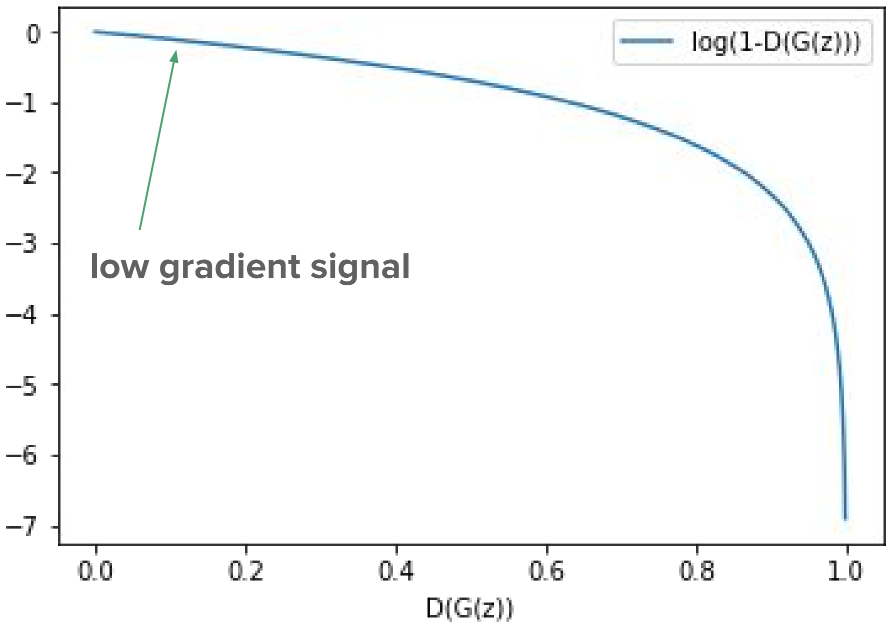
$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

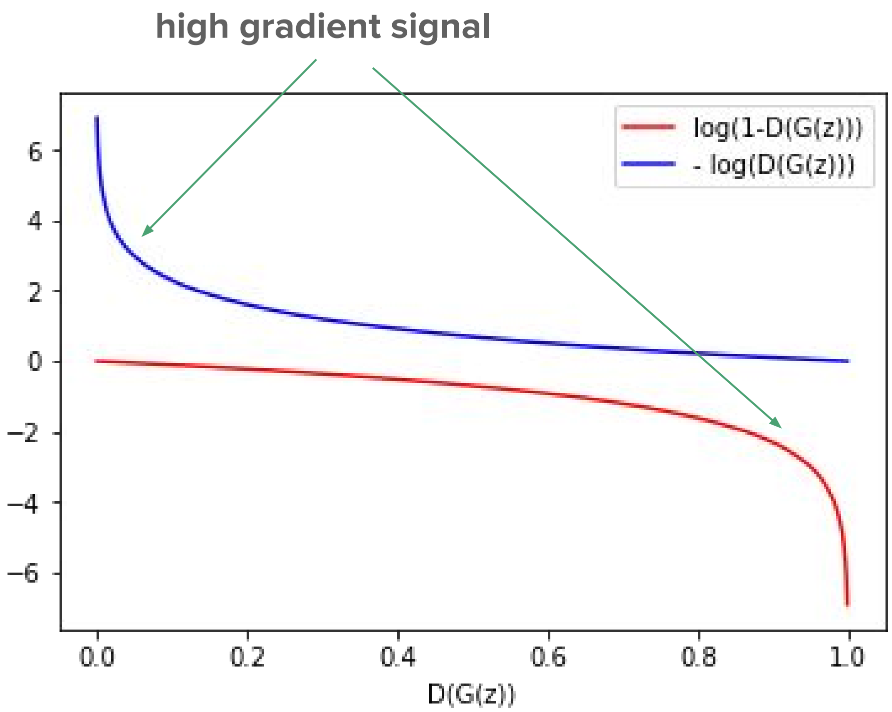
Gradient descent on generator

$$\min_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

In practice, this generator optimization does not work well!

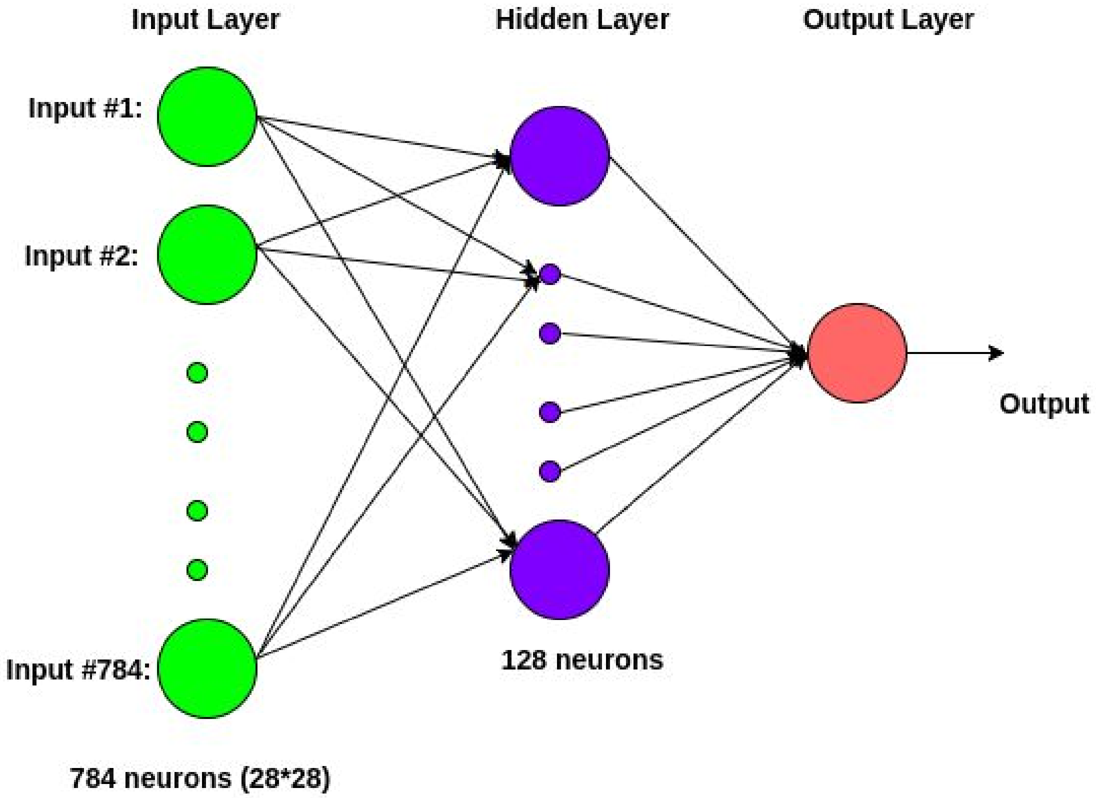
Generator optimization



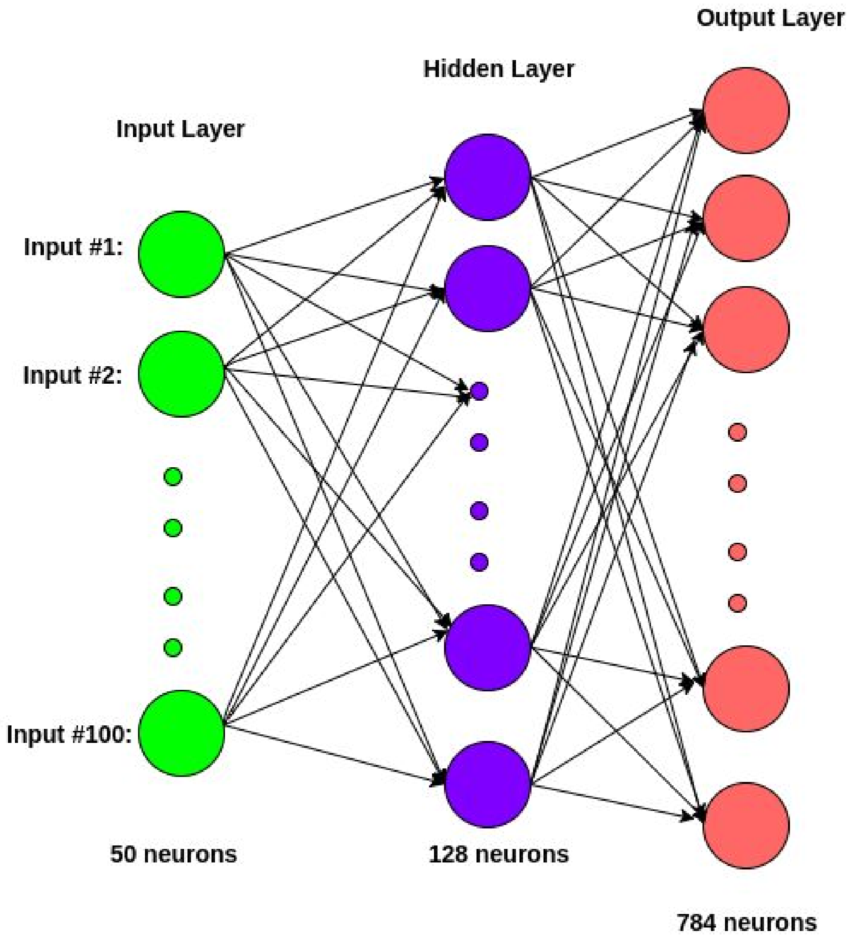


Instead of **minimizing log(1-D(G(z)))**, we can **maximize log(D(G(z)))** or equivalently **minimize -log(D(G(z))),** since this function has high gradient signal where the discriminator network is able to easily able to recognize fake images.

Discriminator Neural Network



Generator Neural Network



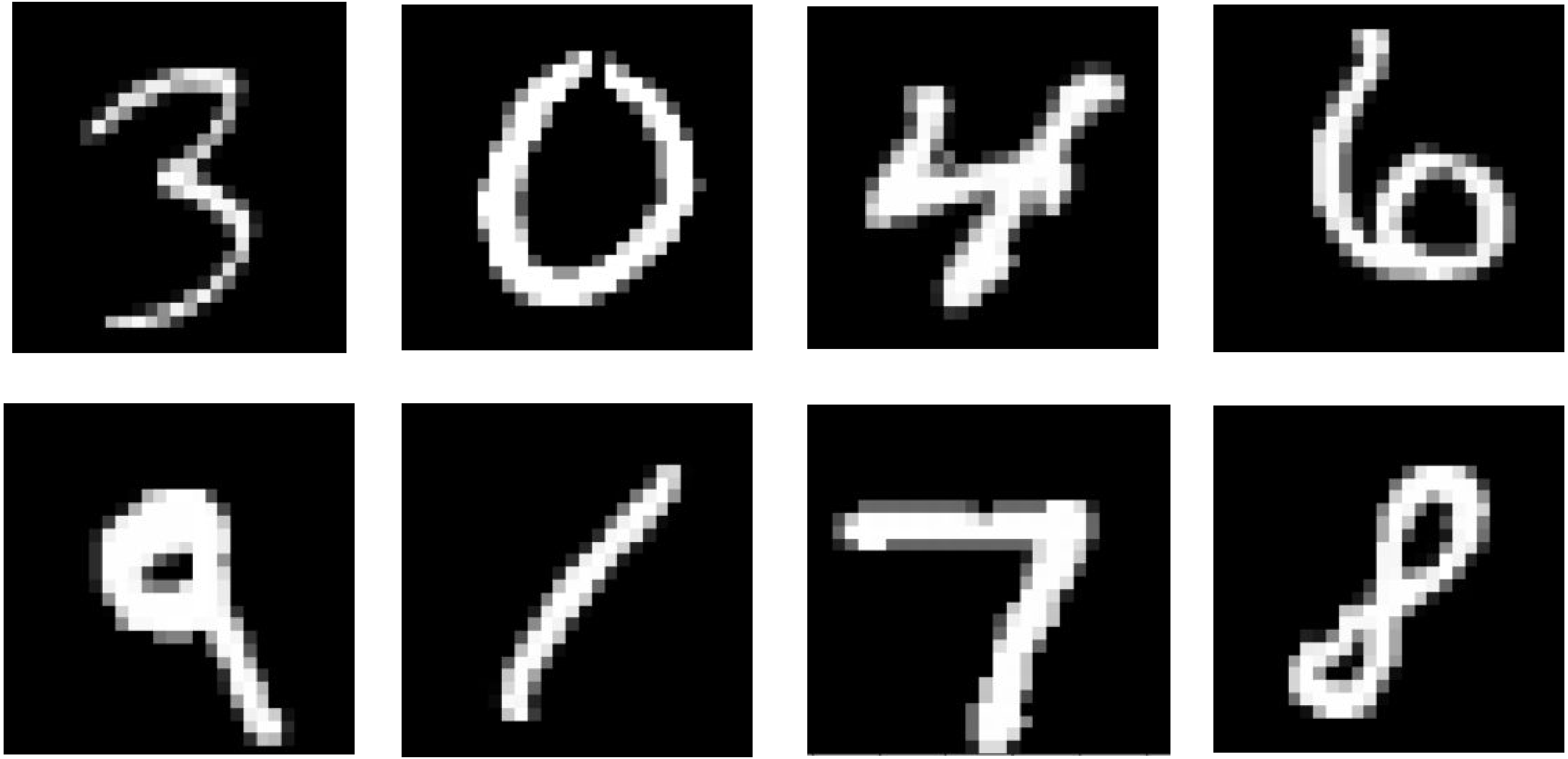
Data Set

The MNIST dataset was used as the primary dataset for training the model. The MNIST database consists of handwritten digits and has a training set of 60,000 examples, and a test set of 10,000 examples. Our objective being generating numbers from the dataset given. MNIST consists of Grey Level images centered in a 28X28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field.

Framework Used

The code was written using the tensor flow library and its python api. The hardware used for the training are a Core i5 Quad core processor coupled with a CUDA capable GTX 1060 along with 8gb of memory. The runtime on this setup for our dataset was roughly 10 hours.

Training Images from MNIST Dataset



GAN Algorithm

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

end for

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Important Functions

1. **Discriminator Network Initialization**

```
# input to the discriminator
# None is used to automatically adjust according to bumber of input images in a batch
X = tf.placeholder(tf.float32, shape=[None, input_dim])

# Weights are intitialized between the first hidden layer and the input layer
# Shape --> input_dim x neuron_dim
Disc_W1 = tf.Variable(weight_initialize([input_dim, neuron_dim]))
# Biases are intialized to zeros
Disc_b1 = tf.Variable(tf.zeros(shape=[neuron_dim]))

# Weights are intialized between hidden layer and output node
# Shape --> neuron_dim x 1
Disc_W2 = tf.Variable(weight_initialize([neuron_dim, 1]))
# Biases are intialized to zeros
Disc_b2 = tf.Variable(tf.zeros(shape=[1]))

# Discriminator variable
var_D = [Disc_W1, Disc_W2, Disc_b1, Disc_b2]
```

2. Generator Network Initialization

```
# input noise to the generator
z = tf.placeholder(tf.float32, shape=[None, noise_dim])

# Weights are intialized between noise layer and first hidden layer
# Shape --> noise_dim x neuron_dim
Genr_W1 = tf.Variable(weight_initialize([noise_dim, neuron_dim]))
# Biases are intialized to zeros
Genr_b1 = tf.Variable(tf.zeros(shape=[neuron_dim]))

# Weights are intialized between hidden layer and output layer (generated image)
# Shape --> neuron_dim x input_dim
Genr_W2 = tf.Variable(weight_initialize([neuron_dim, input_dim]))
# Biases are intialized to zeros
Genr_b2 = tf.Variable(tf.zeros(shape=[input_dim]))

# Generator variable
var_G = [Genr_W1, Genr_W2, Genr_b1, Genr_b2]
```

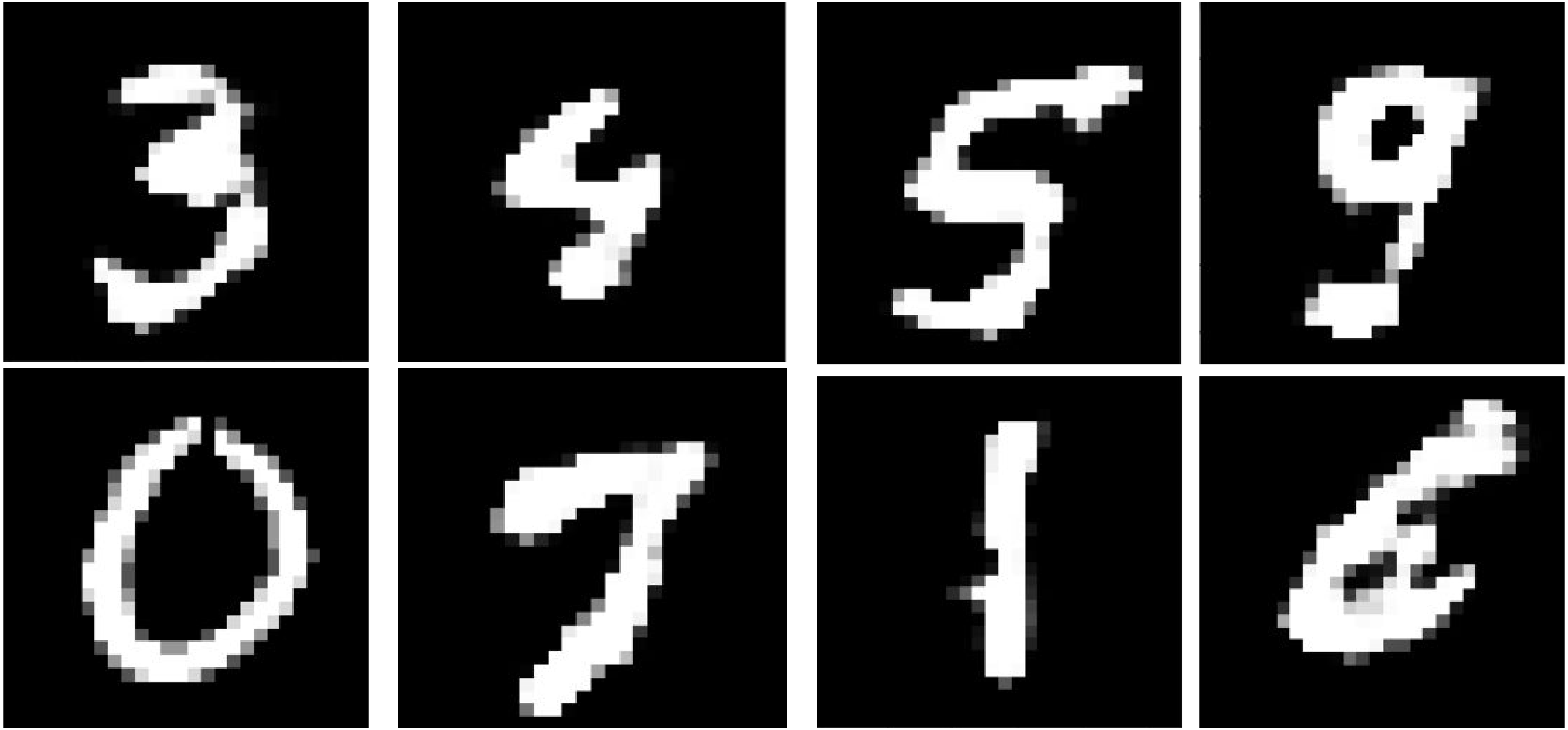
3. Discriminator & Generator Functions

```
def generator(z):
    # [None, noise_dim] x [noise_dim, neuron_dim] = None, neuron_dim
    Genr_h = tf.nn.relu(tf.matmul(z, Genr_W1) + Genr_b1)
    # [None, neuron_dim] x [neuron_dim, input_dim] = None, input_dim
    Genr_log_prob = tf.matmul(Genr_h, Genr_W2) + Genr_b2
    Genr_prob = tf.nn.sigmoid(Genr_log_prob)
    return Genr_prob

def discriminator(x):
    # [None, input_dim] x [input_dim, neuron_dim] = None, neuron_dim
    Disc_h = tf.nn.relu(tf.matmul(x, Disc_W1) + Disc_b1)
    # [None, neuron_dim] x [neuron_dim, 1] = None, 1
    Disc_log_prob = tf.matmul(Disc_h, Disc_W2) + Disc_b2
    return Disc_log_prob
```

4. Loss Functions

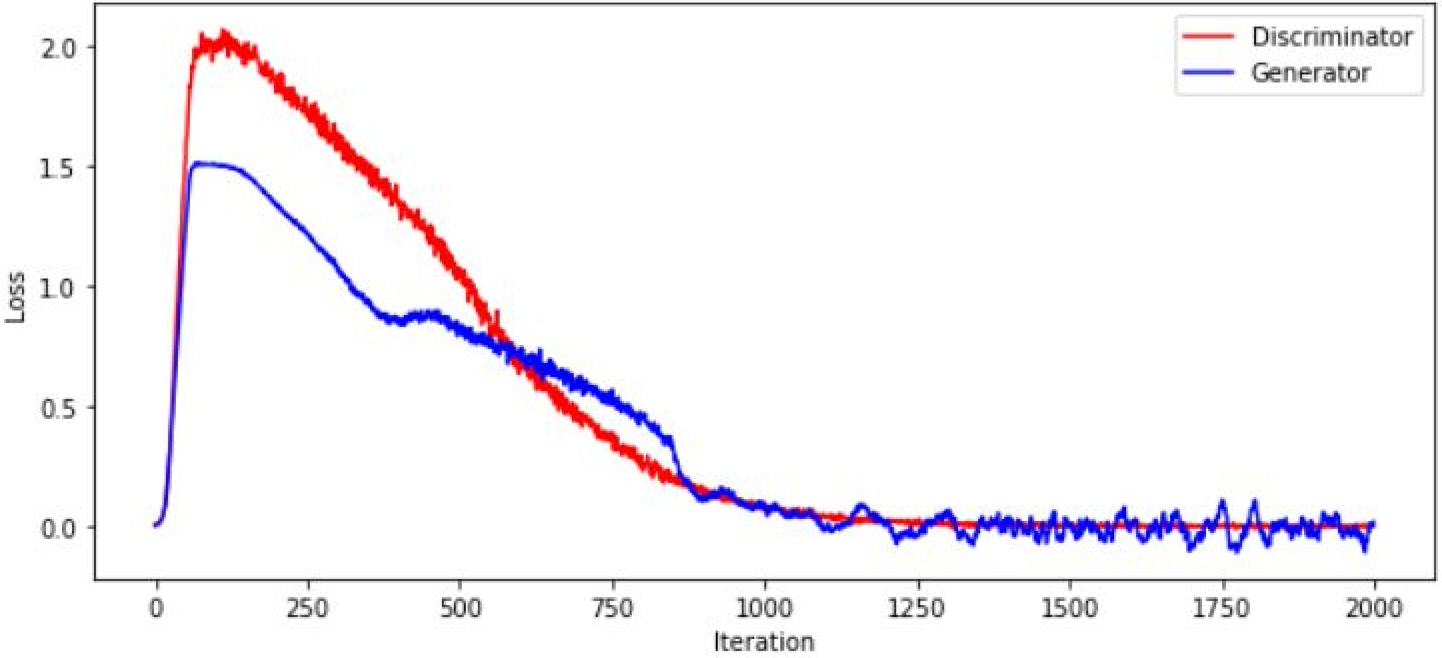
```
G_sample = generator(z)
D_real = discriminator(X)
D_fake = discriminator(G_sample)

D_loss = tf.reduce_mean(D_real) - tf.reduce_mean(D_fake) # Maximize this
G_loss = -tf.reduce_mean(D_fake) # Minimize this

D_solver = (tf.train.RMSPropOptimizer(learning_rate=0.0001).minimize(-D_loss, var_list=var_D))
G_solver = (tf.train.RMSPropOptimizer(learning_rate=0.0001).minimize(G_loss, var_list=var_G))
```

Output from Generator (10^6 Iterations)

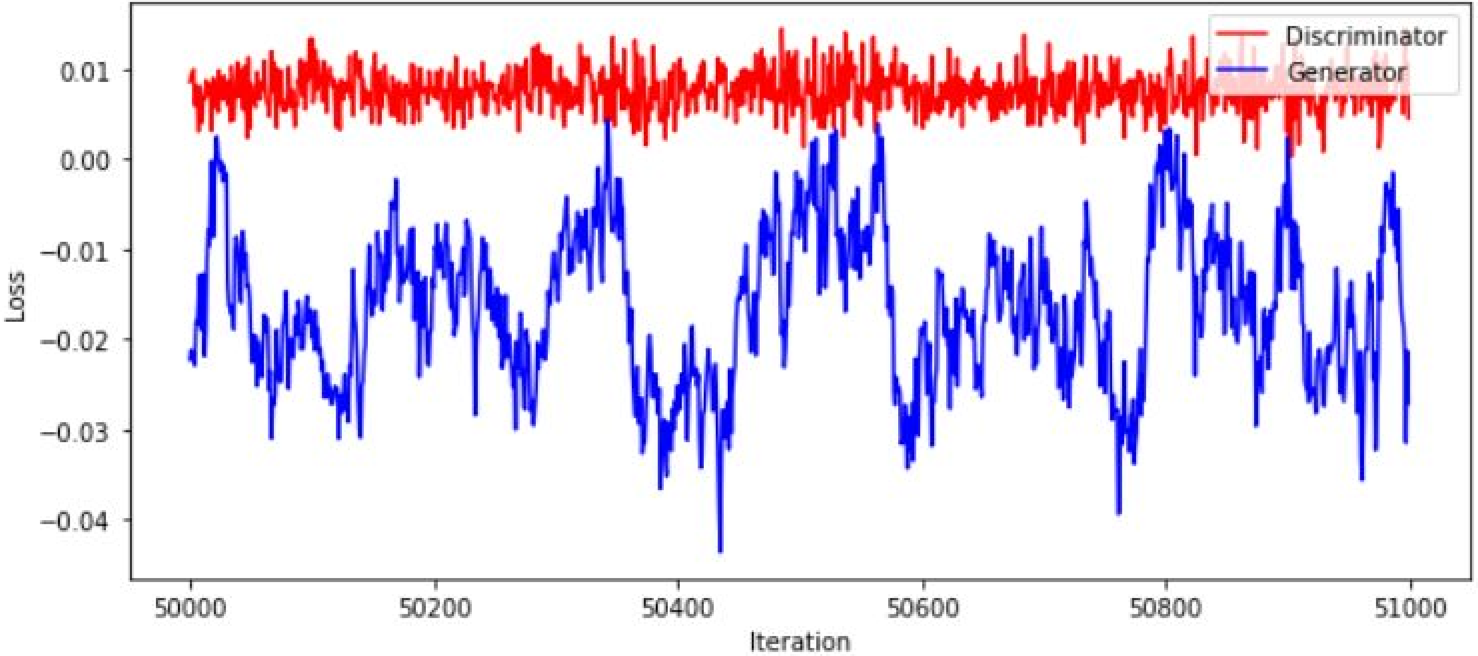


Results

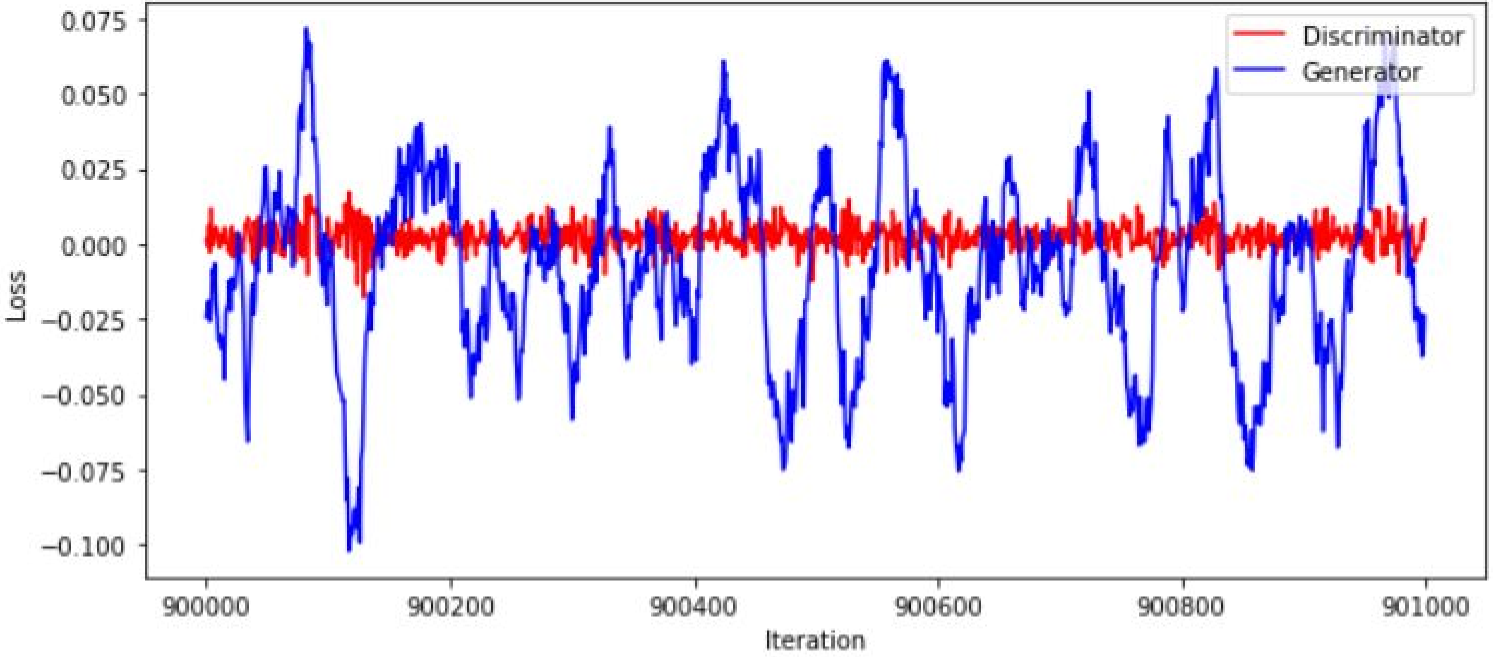
Comparing Loss functions of Generator and Discriminator during Training

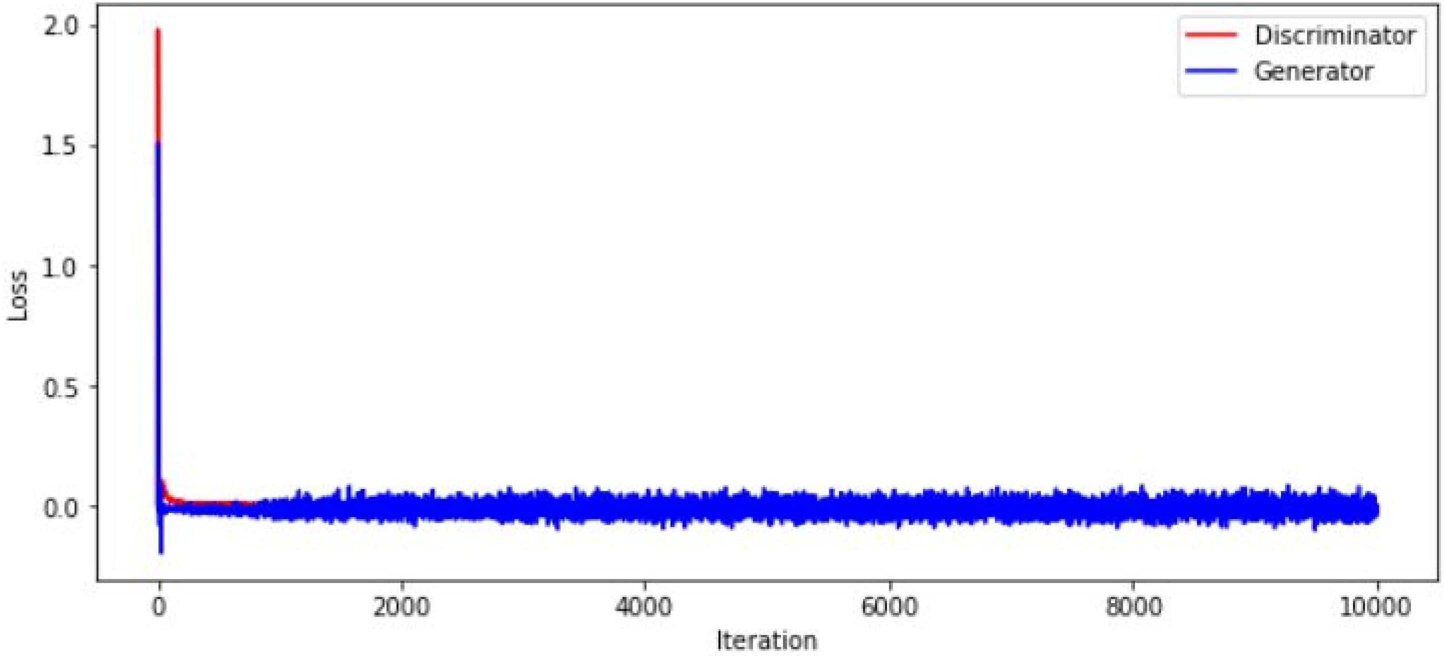


Before Convergence



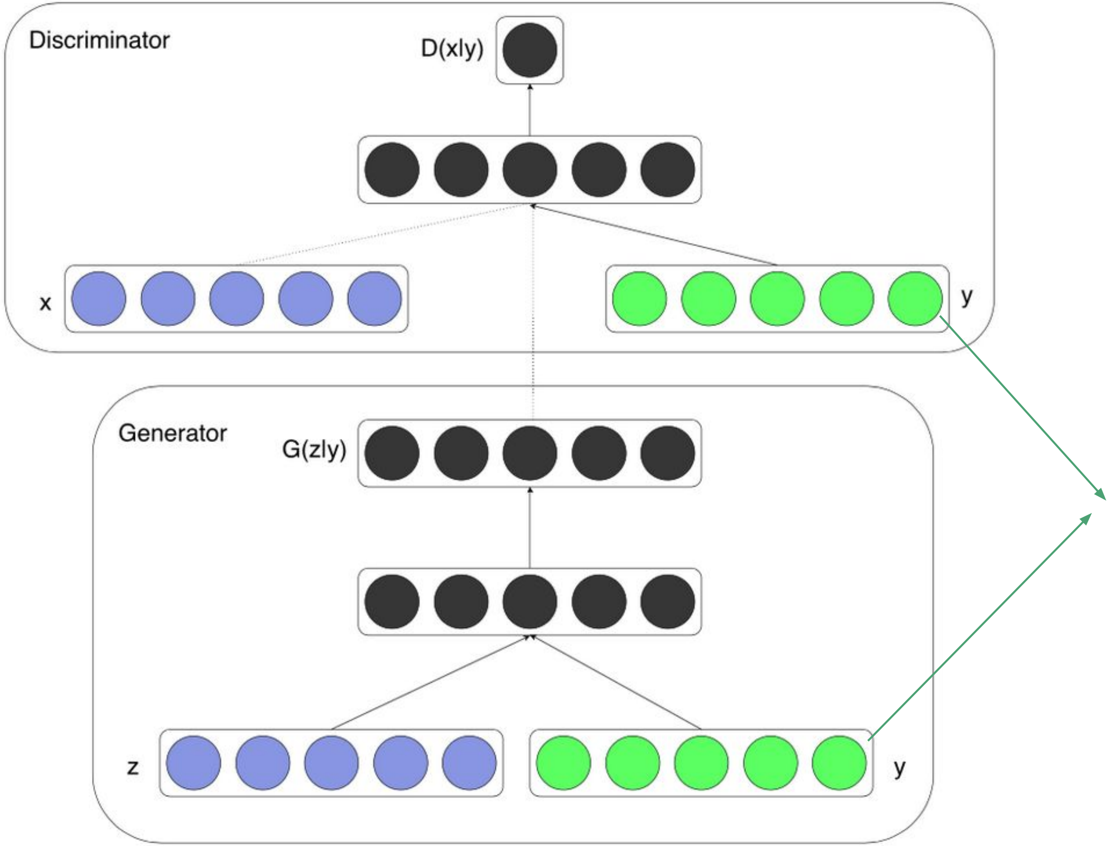
After Convergence





Generator and Discriminator compete against each to finally settle on a value where discriminator loss is maximised and generator loss is minimised

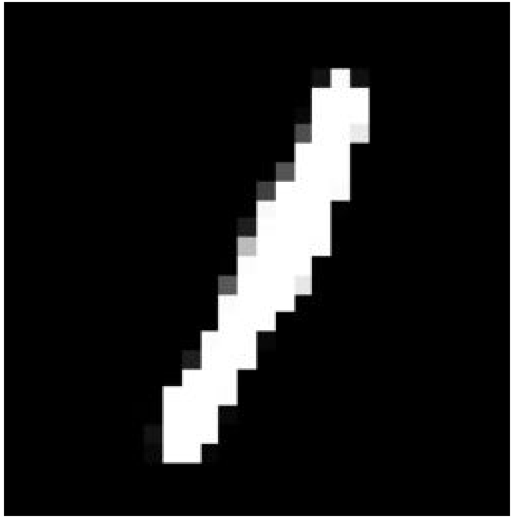
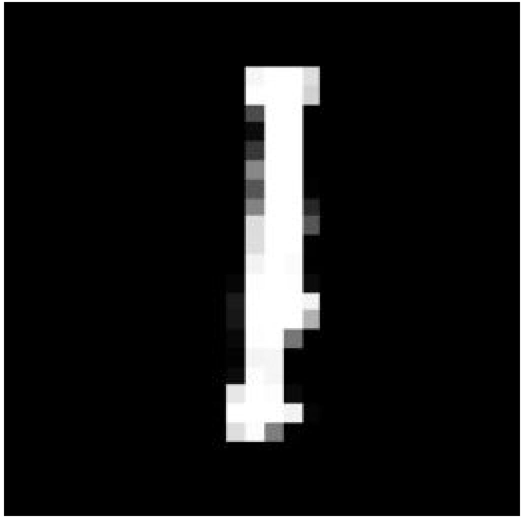
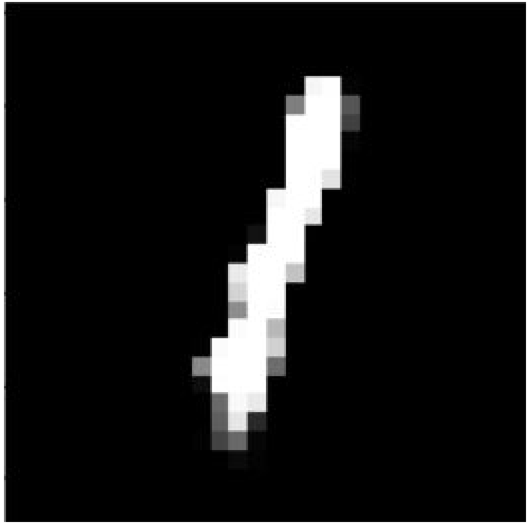
Class Specific GANs



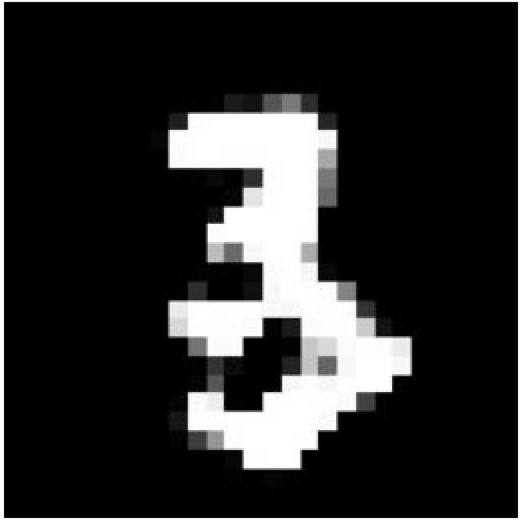
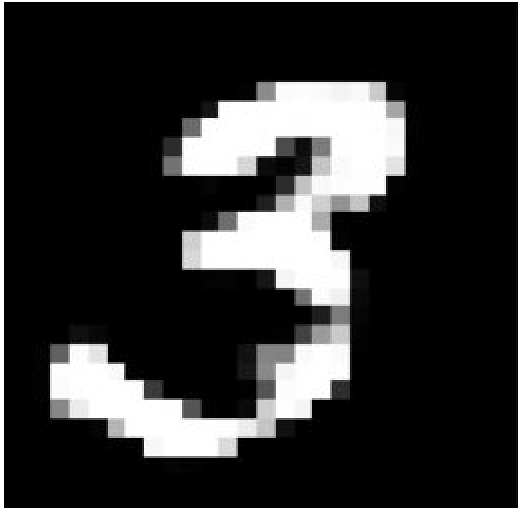
One hot representation of class labels fed parallely into both Discriminator and Generator for producing class specific images.

Class Specific GANs Output

Input 1

Input 3

Input 7

Future Work

1. Predicting the noise pattern given the generated image i.e. $P(z|x)$
   It has been proved that the input noise is not random but follows certain patterns i.e. For generating images belonging to a particular class, the noise vectors must follow a particular pattern but that relation has not been found yet.

2. Efficiency improvements
   Training could be accelerated greatly by devising better methods for coordinating G and D or determining better distributions to sample z from during training.

3. Predicting distribution function for input Dataset
   One of the disadvantages of Adversarial training is that it fails to predict the exact distribution functions of the input data. If we a have distribution function, then any point on the function would represent a new generated image which would be very efficient.

References

1. https://www.oreilly.com/learning/generative-adversarial-networks-for-beginners
2. http://kvfrans.com/generative-adversial-networks-explained/
3. https://arxiv.org/abs/1411.1784
4. https://arxiv.org/abs/1506.05751
5. https://wiseodd.github.io/techblog/2016/09/17/gan-tensorflow/
6. https://youtu.be/0VPQHbMvGzg
7. https://www.youtube.com/watch?v=QPkb5VcgXAM

Thank You

- From Group 39