Using GANs to generate MNIST digits

Let us see if we can implement what we have thus far discussed. We will download the popular MNIST digit dataset, and use TensorFlow to create some adversarial nets. Using the MNIST data, we will train the discriminator (D) and then run the alternation game, where the generator G will seek to improve its digit 'imposters' until D can (and hopefully the reader) no longer differentiate whether it is a real or a fake.

First, we create two networks. This is a fairly open task, allowing the experimenter to build nets as complex or as simple as they desire. In order to get a clear picture of the process, we will build two simple 2-layer nets for D and G.

Of course, we'll start by importing some dependencies.

In [ ]:
```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import os
```

In [ ]:
```
# the Discriminator
X = tf.placeholder(tf.float32, shape=[None, 784])

D_W1 = tf.Variable(xavier_init([784, 128]))
D_b1 = tf.Variable(tf.zeros(shape=[128]))

D_W2 = tf.Variable(xavier_init([128, 1]))
D_b2 = tf.Variable(tf.zeros(shape=[1]))

theta_D = [D_W1, D_W2, D_b1, D_b2]

# the Generator
Z = tf.placeholder(tf.float32, shape=[None, 100])

G_W1 = tf.Variable(xavier_init([100, 128]))
G_b1 = tf.Variable(tf.zeros(shape=[128]))

G_W2 = tf.Variable(xavier_init([128, 784]))
G_b2 = tf.Variable(tf.zeros(shape=[784]))

theta_G = [G_W1, G_W2, G_b1, G_b2]
```

The generator will take as input a 100-dimensional vector and transform it into an output of 768 dimensions. That is, a vector representing a standard MNIST image of size 28x28.

The discriminator will take as input a MNIST image and return its prediction in the form of a scalar representing the probability it is a true image from the original dataset.

In [ ]:
```
def generator(z):
    G_h1 = tf.nn.relu(tf.matmul(z, G_W1) + G_b1)
    G_log_prob = tf.matmul(G_h1, G_W2) + G_b2
    G_prob = tf.nn.sigmoid(G_log_prob)

    return G_prob


def discriminator(x):
    D_h1 = tf.nn.relu(tf.matmul(x, D_W1) + D_b1)
    D_logit = tf.matmul(D_h1, D_W2) + D_b2
    D_prob = tf.nn.sigmoid(D_logit)

    return D_prob, D_logit
```

Now we are ready to define the loss function. Following Goodfellow's formulation in his 2014 paper presented at NIPS, we have:

In [ ]:
```
G_sample = generator(Z)
D_real, D_logit_real = discriminator(X)
D_fake, D_logit_fake = discriminator(G_sample)

D_loss = -tf.reduce_mean(tf.log(D_real) + tf.log(1. - D_fake))
G_loss = -tf.reduce_mean(tf.log(D_fake))
```

Since TensorFlow's optimizer can only hanlde minimizations, we take the opposite of the loss and attempt to minimize it.

Now we can let the adversaries tango, if you will. We run the training process for 100,000 iterations locally (given computational constraints, namely 4GB RAM).

In [ ]:
```
for it in range(100000):
    if it % 1000 == 0:
        samples = sess.run(G_sample, feed_dict={Z: sample_Z(16, Z_dim)})

        fig = plot(samples)
        plt.savefig('out2/{}.png'.format(str(i).zfill(3)),
bbox_inches='tight')
        i += 1
        plt.close(fig)

    X_mb, _ = mnist.train.next_batch(mb_size)

    _, D_loss_curr = sess.run([D_solver, D_loss], feed_dict={X: X_mb, Z:
 sample_Z(mb_size, Z_dim)})
    _, G_loss_curr = sess.run([G_solver, G_loss], feed_dict={Z:
sample_Z(mb_size, Z_dim)})
```

By sampling the generator every 1000 iterations, we can get a glimpse of the training process. Amazingly, in only 6000 iterations, the generated images go from random blurr to semi-destinguishable digits (Don't worry, we will see the result of 100k iterations soon...).

In [ ]:
```
def sample_Z(m, n):
    return np.random.uniform(-1., 1., size=[m, n])
```

In [2]:
```
from IPython.display import Image
Image(url='digits_loss1.gif')
```

Out[2]:

We can also try defining the loss functions slightly differently and see if the results are still in line with what we want. Considering D(x) given some input x, D wishes to minimize D(x) when x is fake, and maximize it when x is real. In other words, the discriminator seeks to minimize the expression D(G(x)). On the other hand, the generator wants to maximize the probability it is accepted as a 'real' sample: max D(G(x)).

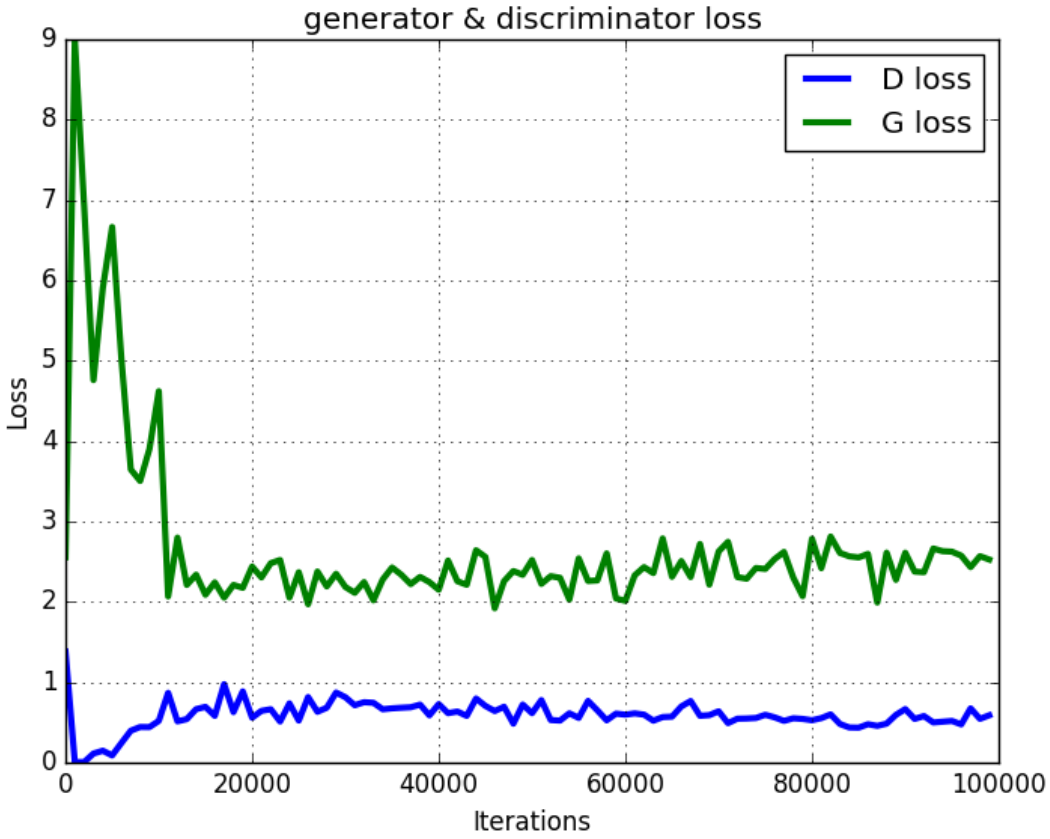
Using this alternative definition, we have the following loss functions:

In [ ]:
```
D_loss_real = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(log
its=D_logit_real, labels=tf.ones_like(D_logit_real)))
D_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(log
its=D_logit_fake, labels=tf.zeros_like(D_logit_fake)))
D_loss = D_loss_real + D_loss_fake
G_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=D
_logit_fake, labels=tf.ones_like(D_logit_fake)))
```

Training the GAN for 100,000 iterations again, we observe loss as a function of iterations in the below graph.

In [3]:
```
from IPython.display import Image
Image(filename='loss2_plot.png')
```

Out[3]:

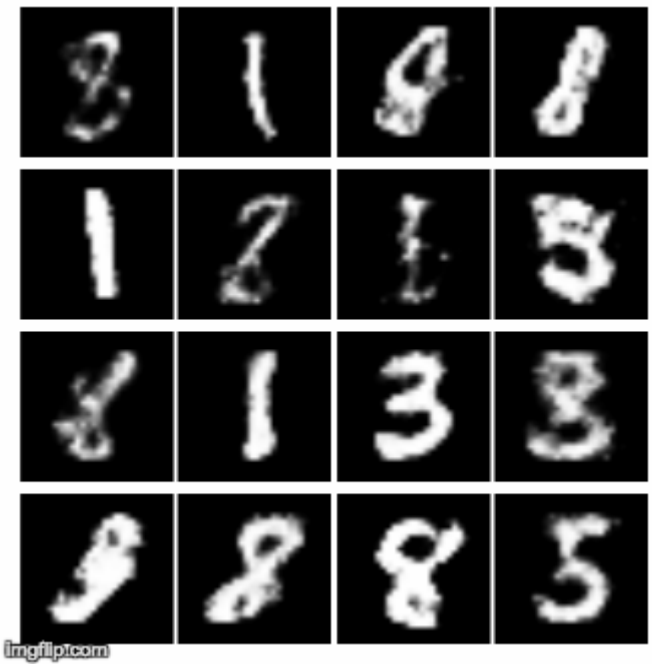


From this graph we can see the generator loss drops dramatically in the first 15,000 iterations and then stabilizes with some fluctuation until the end of training. Similarly, we see discriminator loss drop sharply in the first few iterations, but then return to a homeostasis between 0 and 1.

Sampling the generator every 1000 iterations, we see well defined digits born out of what started as a nebulous zygote. In this animation, we see a sequence of 100 frames, one for every 1000 iterations.

In [4]:
```
from IPython.display import Image
Image(url='digits_loss2.gif')
```

Out[4]:



The results are pretty staggering, considering the generator's first images and how rapidly they became viable MNIST candidates.

Concluding Remarks

In this blog post, we brought generative adversarial networks to the fore and discussed their defining properties, as defined by their creator - Ian Goodfellow. Looking at the dynamic between the generative and discriminative nets, we saw what makes the adversarial training unique and so effective.

With these fundementals, we were able to see how GANs are producing such realistic results in the field of image generation and how they might be improved to create convincing video frame prediction in the futre.

Lastly, we implemented a GAN using TensorFlow and the used MNIST digit dataset to produce digit images very comparable to the originals.