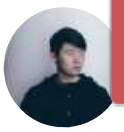
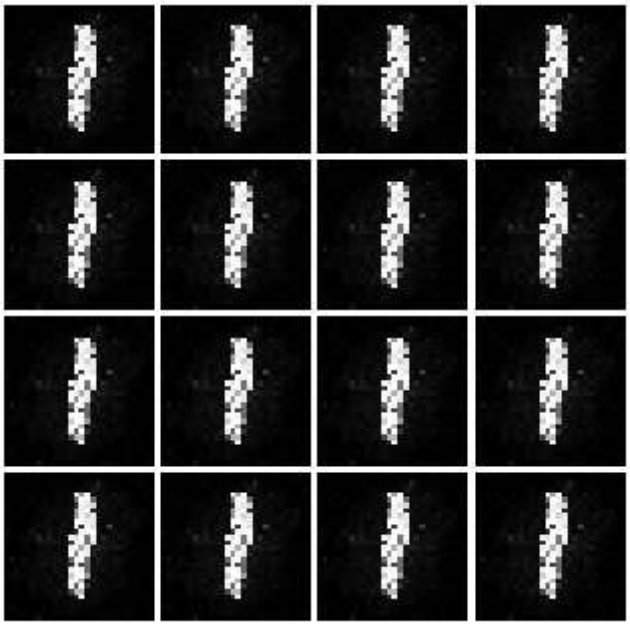


Feb 1 · 6 min read

Only Numpy: Implementing GAN (General Adversarial Networks) and Adam Optimizer using Numpy with Interactive Code. (Run GAN Online)

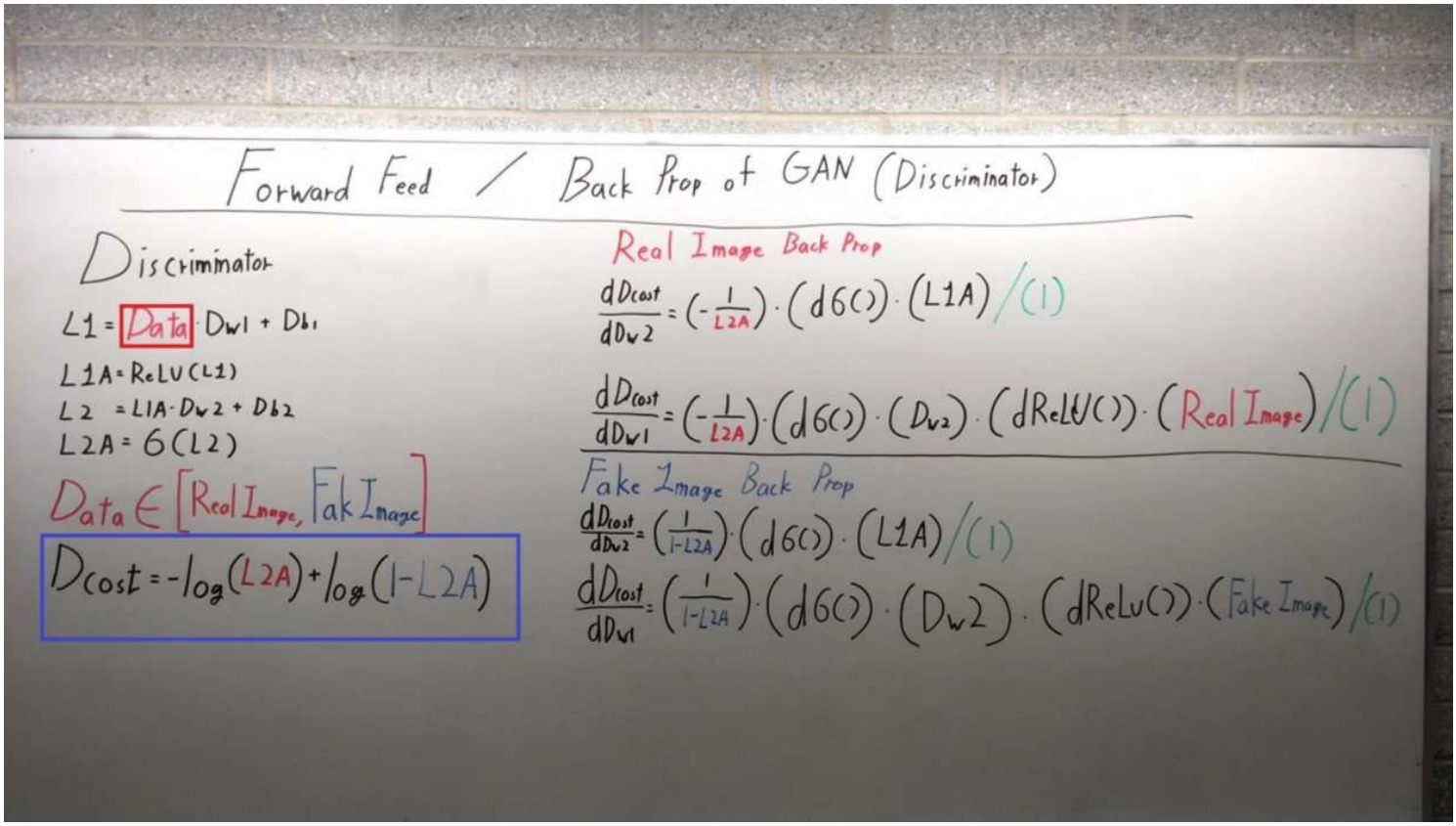
So today I was inspired by this blog post, "Generative Adversarial Nets in TensorFlow" and I wanted to implement GAN myself using Numpy. Here is the original GAN paper by @goodfellow_ian .Below is a gif of all generated images from Simple GAN.



Before reading along, please note that I won't be covering too much of math. Rather the implementation of the code and results, I will cover the math maybe later. And I am using Adam Optimizer, however, I won't go into explaining the implementation of Adam at this post.

• • •

Forward Feed / Partial Back Propagation of Discriminator in GAN



Again, I won't go into too much details, but please note the Red Boxed Region called Data. For the Discriminator Network in GAN, that Data either can be Real Image or Fake Image Generated by the Generator Network. Our images are (1,784) vector of MNIST data set.

One more thing to note is **Red (L2A) and Blue (L2A)**. Red (L2A) is the final output of our Discriminator Network with Real Image as input. And Blue (L2A) is the final output of our Discriminator Network with Fake Image as input.

```
126
127     random_int = np.random.randint(len(images) - 5)
128     current_image = np.expand_dims(images[random_int],axis=0)
129
130     # Func: Generate The first Fake Data
131     Z = np.random.uniform(-1., 1., size=[1, G_input])
132     Gl1 = Z.dot(G_W1) + G_b1
133     Gl1A = arctan(Gl1)
134     Gl2 = Gl1A.dot(G_W2) + G_b2
135     Gl2A = ReLu(Gl2)
136     Gl3 = Gl2A.dot(G_W3) + G_b3
137     Gl3A = arctan(Gl3)
138
139     Gl4 = Gl3A.dot(G_W4) + G_b4
140     Gl4A = ReLu(Gl4)
141     Gl5 = Gl4A.dot(G_W5) + G_b5
142     Gl5A = tanh(Gl5)
143     Gl6 = Gl5A.dot(G_W6) + G_b6
144     Gl6A = ReLu(Gl6)
145     Gl7 = Gl6A.dot(G_W7) + G_b7
146
147     current_fake_data = log(Gl7)
148
149     # Func: Forward Feed for Real data
150     Dl1_r = current_image.dot(D_W1) + D_b1
151     Dl1_rA = ReLu(Dl1_r)
152     Dl2_r = Dl1_rA.dot(D_W2) + D_b2
153     Dl2_rA = log(Dl2_r)
154
155     # Func: Forward Feed for Fake Data
156     Dl1_f = current_fake_data.dot(D_W1) + D_b1
157     Dl1_fA = ReLu(Dl1_f)
158     Dl2_f = Dl1_fA.dot(D_W2) + D_b2
159     Dl2_fA = log(Dl2_f)
160
161     # Func: Cost D
162     D_cost = -np.log(Dl2_rA) + np.log(1.0- Dl2_fA)
163
```

The way we implement this is by getting the real image, and fake data before putting them both into the network.

Line 128—Getting the Real Image Data
Line 147—Getting the Fake Image Data (Generated By Generator Network)

Line 162—Cost Function of our Discriminator Network.

Also, please take note of the Blue Box Region, that is our cost function. Lets compare the cost function from the original paper, shown below.

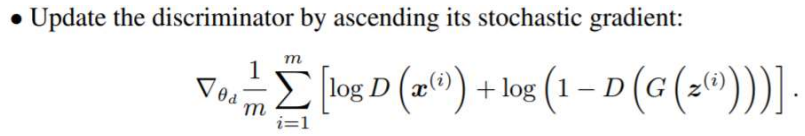


Image from original Paper

The difference is the fact that we are putting a (-) negative sign in front of the first term log(L2A).

```
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))
```

Above, we use negative sign for the loss functions because they need to be maximized, whereas TensorFlow's optimizer can only do minimization.

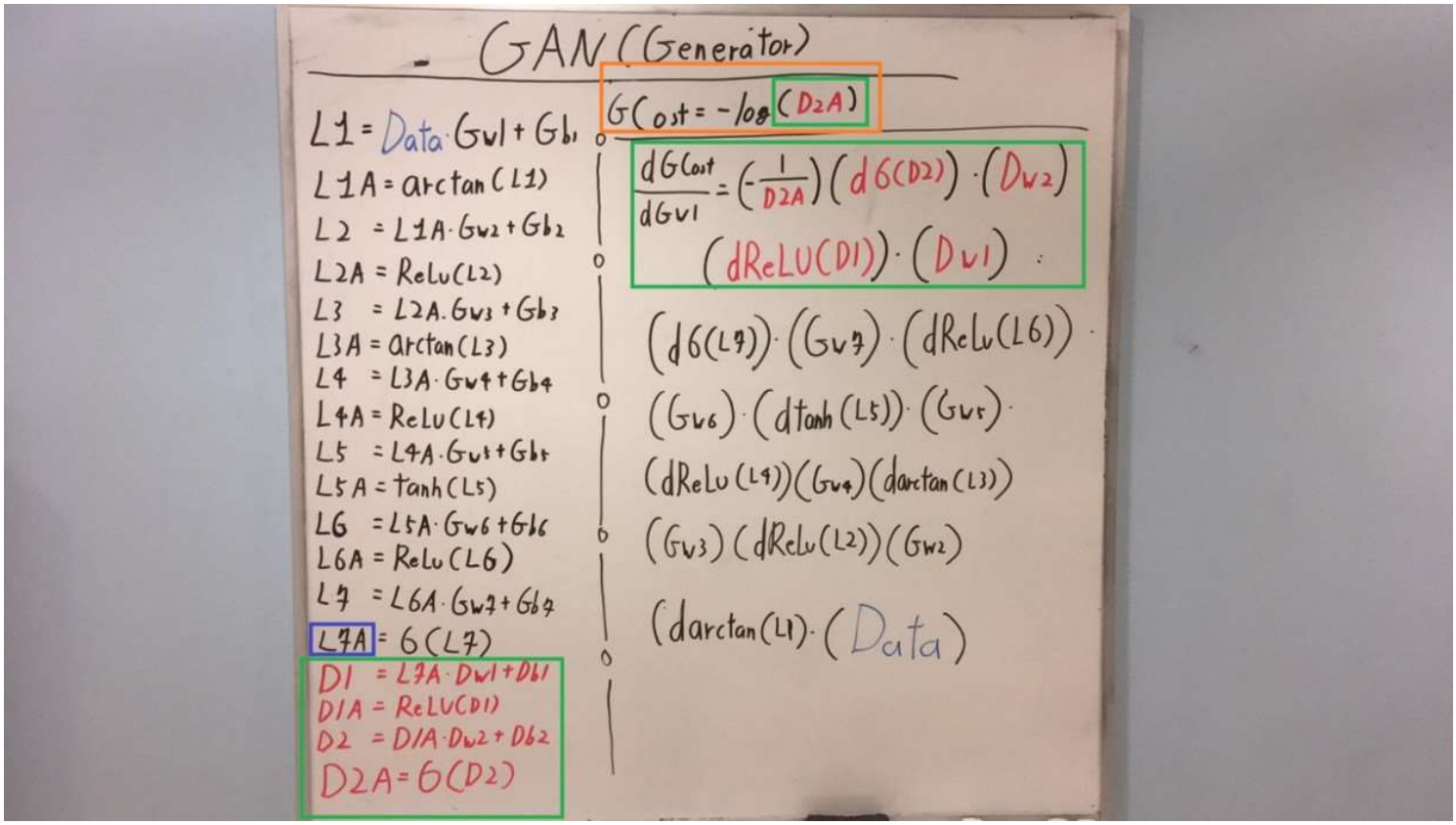
Image from Agustinus Kristiadi

As seen above, in TensorFlow implementation we flip the signs if we want to Maximize some value, since TF auto differentiation only can Minimize.

I thought about this and I decided to implement in a similar way. Cuz I wanted to Maximize the chance of our discriminator guessing right for real image while Minimize the chance of our discriminator guessing wrong for fake images, and I wanted the sum of those values to balance out. However, I am not 100 % sure of this part as of yet, and will revisit this matter soon.

. . .

Forward Feed / Partial Back Propagation of Generator in GAN



The Back Propagation process for generator network in GAN is bit complicated.

Blue Box—Generated Fake Data from the Generator Network
Green Box (Left Corner)—Discriminator Takes the Generated (Blue Box) Input and perform Forward Feed process

Orange Box—Cost Function for Generator Network (Again we want to Maximize the chance of producing a Realistic Data)

Green Box (Right Corner)—Back Propagation Process for Generator Network, but we have to pass Gradient all the way Discriminator Network.

Below is the screen shot of implemented code.

```
# Func: Gradient
grad_G_w7_part_1 = ((-1/Dl2_A) * d_log(Dl2).dot(D_W2.T) * (d_ReLu(Dl1))).dot(D_W1.T)
grad_G_w7_part_2 = d_log(Gl7)
grad_G_w7_part_3 = Gl6A
grad_G_w7 = grad_G_w7_part_3.T.dot(grad_G_w7_part_1 * grad_G_w7_part_1)
grad_G_b7 = grad_G_w7_part_1 * grad_G_w7_part_2

grad_G_w6_part_1 = (grad_G_w7_part_1 * grad_G_w7_part_2).dot(G_W7.T)
grad_G_w6_part_2 = d_ReLu(Gl6)
grad_G_w6_part_3 = Gl5A
grad_G_w6 = grad_G_w6_part_3.T.dot(grad_G_w6_part_1 * grad_G_w6_part_2)
grad_G_b6 = (grad_G_w6_part_1 * grad_G_w6_part_2)

grad_G_w5_part_1 = (grad_G_w6_part_1 * grad_G_w6_part_2).dot(G_W6.T)
grad_G_w5_part_2 = d_tanh(Gl5)
grad_G_w5_part_3 = Gl4A
grad_G_w5 = grad_G_w5_part_3.T.dot(grad_G_w5_part_1 * grad_G_w5_part_2)
grad_G_b5 = (grad_G_w5_part_1 * grad_G_w5_part_2)

grad_G_w4_part_1 = (grad_G_w5_part_1 * grad_G_w5_part_2).dot(G_W5.T)
grad_G_w4_part_2 = d_ReLu(Gl4)
grad_G_w4_part_3 = Gl3A
grad_G_w4 = grad_G_w4_part_3.T.dot(grad_G_w4_part_1 * grad_G_w4_part_2)
grad_G_b4 = (grad_G_w4_part_1 * grad_G_w4_part_2)

grad_G_w3_part_1 = (grad_G_w4_part_1 * grad_G_w4_part_2).dot(G_W4.T)
grad_G_w3_part_2 = d_arctan(Gl3)
grad_G_w3_part_3 = Gl2A
grad_G_w3 = grad_G_w3_part_3.T.dot(grad_G_w3_part_1 * grad_G_w3_part_2)
grad_G_b3 = (grad_G_w3_part_1 * grad_G_w3_part_2)

grad_G_w2_part_1 = (grad_G_w3_part_1 * grad_G_w3_part_2).dot(G_W3.T)
grad_G_w2_part_2 = d_ReLu(Gl2)
grad_G_w2_part_3 = Gl1A
grad_G_w2 = grad_G_w2_part_3.T.dot(grad_G_w2_part_1 * grad_G_w2_part_2)
grad_G_b2 = (grad_G_w2_part_1 * grad_G_w2_part_2)

grad_G_w1_part_1 = (grad_G_w2_part_1 * grad_G_w2_part_2).dot(G_W2.T)
grad_G_w1_part_2 = d_arctan(Gl1)
grad_G_w1_part_3 = Z
grad_G_w1 = grad_G_w1_part_3.T.dot(grad_G_w1_part_1 * grad_G_w1_part_2)
grad_G_b1 = grad_G_w1_part_1 * grad_G_w1_part_2
```

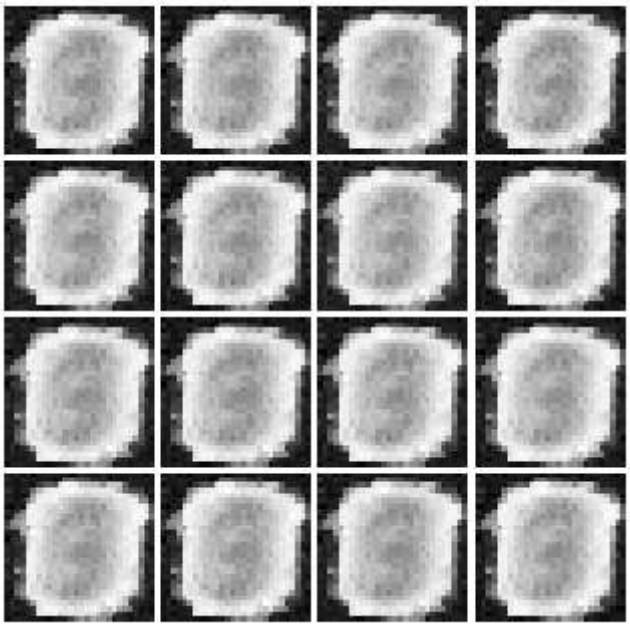
Standard Back propagation, nothing too special.

. . .

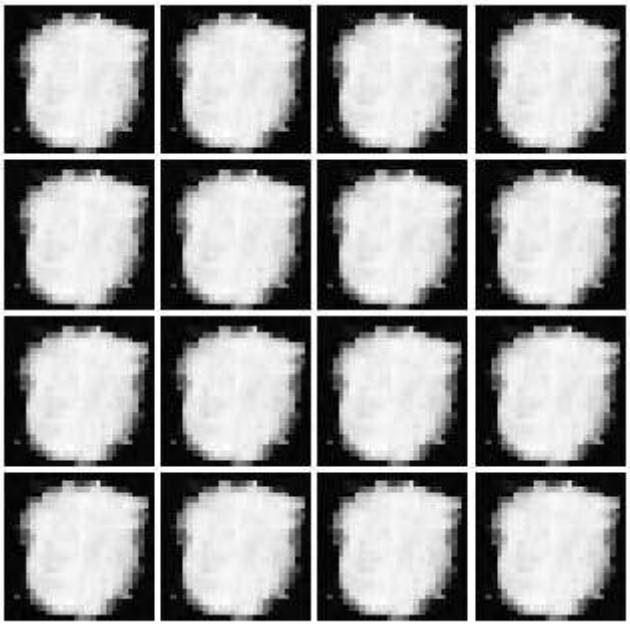
Training Results : Failed Attempts

I quickly realized that training GAN is extremely hard, even with Adam Optimizer, the network just didn't seem to converge well. So, I will first present you all of the failed attempts and it's network architectures.

1. *Generator, 2 Layers: 200, 560 Hidden Neurons, Input Vector Size 100*

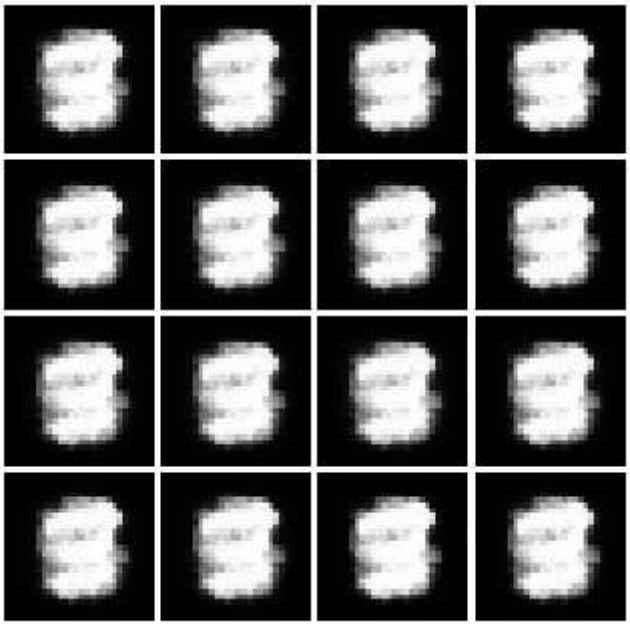


2. Generator, tanh() Activation, 2 Layers: 245, 960 Hidden Neurons, IVS 100

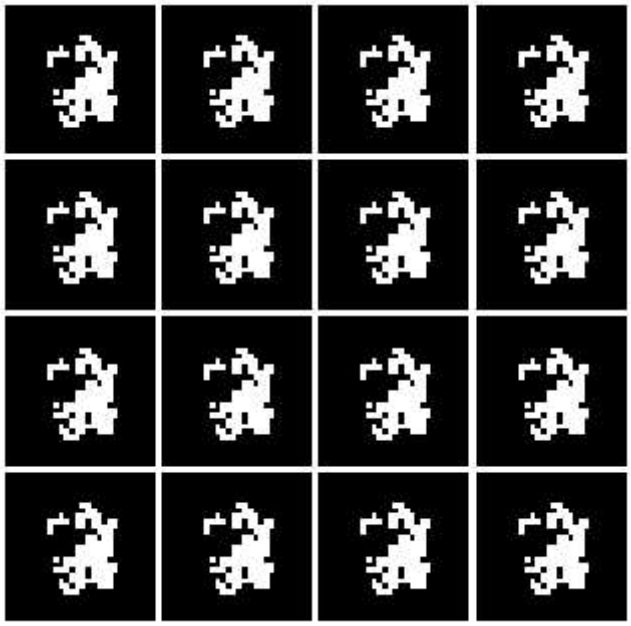


3. Generator, 3 Layers: 326, 356,412 Hidden Neurons, Input Vector Size 326

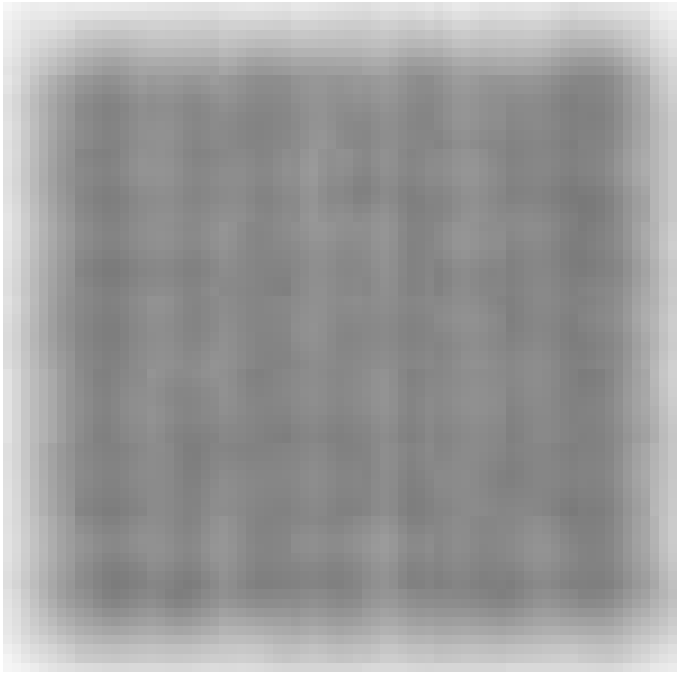
4. Generator, 2 Layers: 420, 640 Hidden Neurons, Input Vector Size 350



5. Generator, 2 Layers: 660, 780 Hidden Neurons, Input Vector Size 600



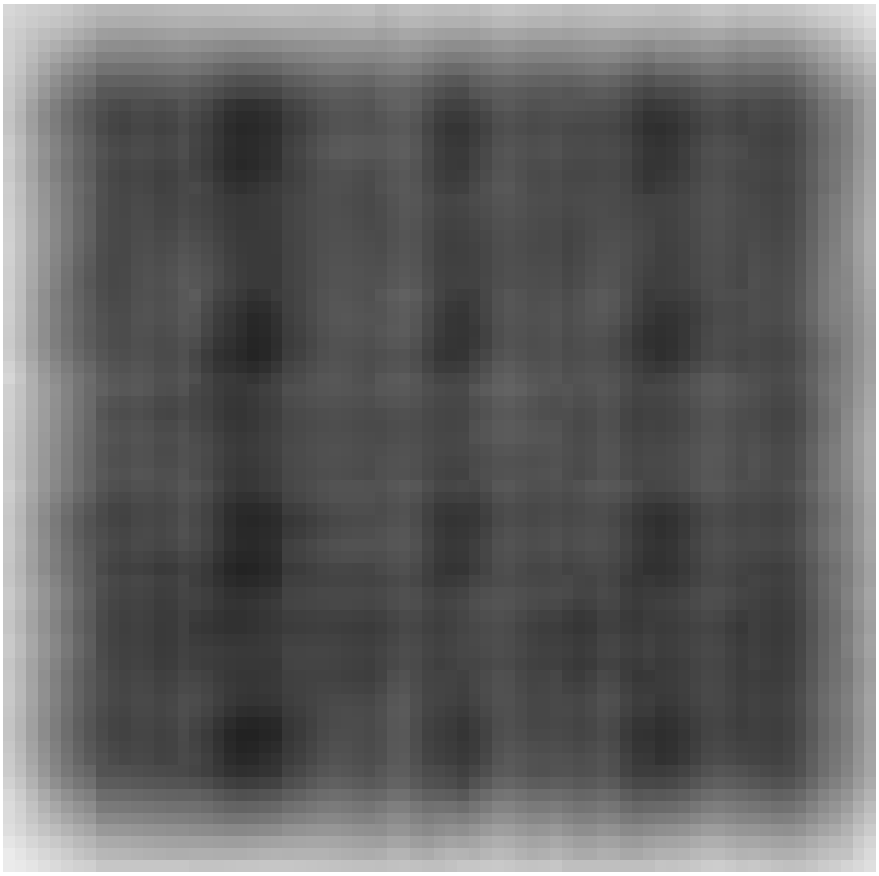
6. Generator, 2 Layers: 320, 480 Hidden Neurons, Input Vector Size 200



So as seen above, all of them seems to learn something, but not really LOL. However, I was able to use one neat trick to generate an image that kinda look like numbers.

• • •

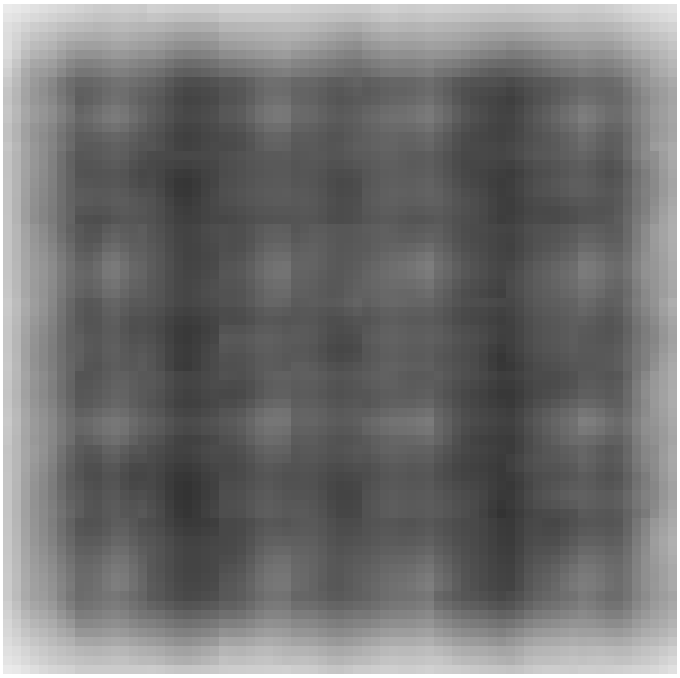
Extreme Step Wise Gradient Decay



Above is a gif, I know the difference is subtle but trust me I ain't Rick Rollying you. The trick is extremely simple and easy to implement. We first set the learning rate high rate for the first training, and after the first training we set the decay the learning rate by factor of 0.01. And for unknown reason (I want to investigate further more into this, this seems to work.)

But with a huge cost, I think we are converging to a 'place' where the network is only able to generate only certain kind of data. Meaning, from the uniform distribution of numbers between -1 and 1. The Generator will only generate image that ONLY looks like a 3 or a 2 etc…. But the key point here is that the network is not able to generate different set of numbers. This is evidence by the fact that, well, all of the numbers represented in the image look like 3.

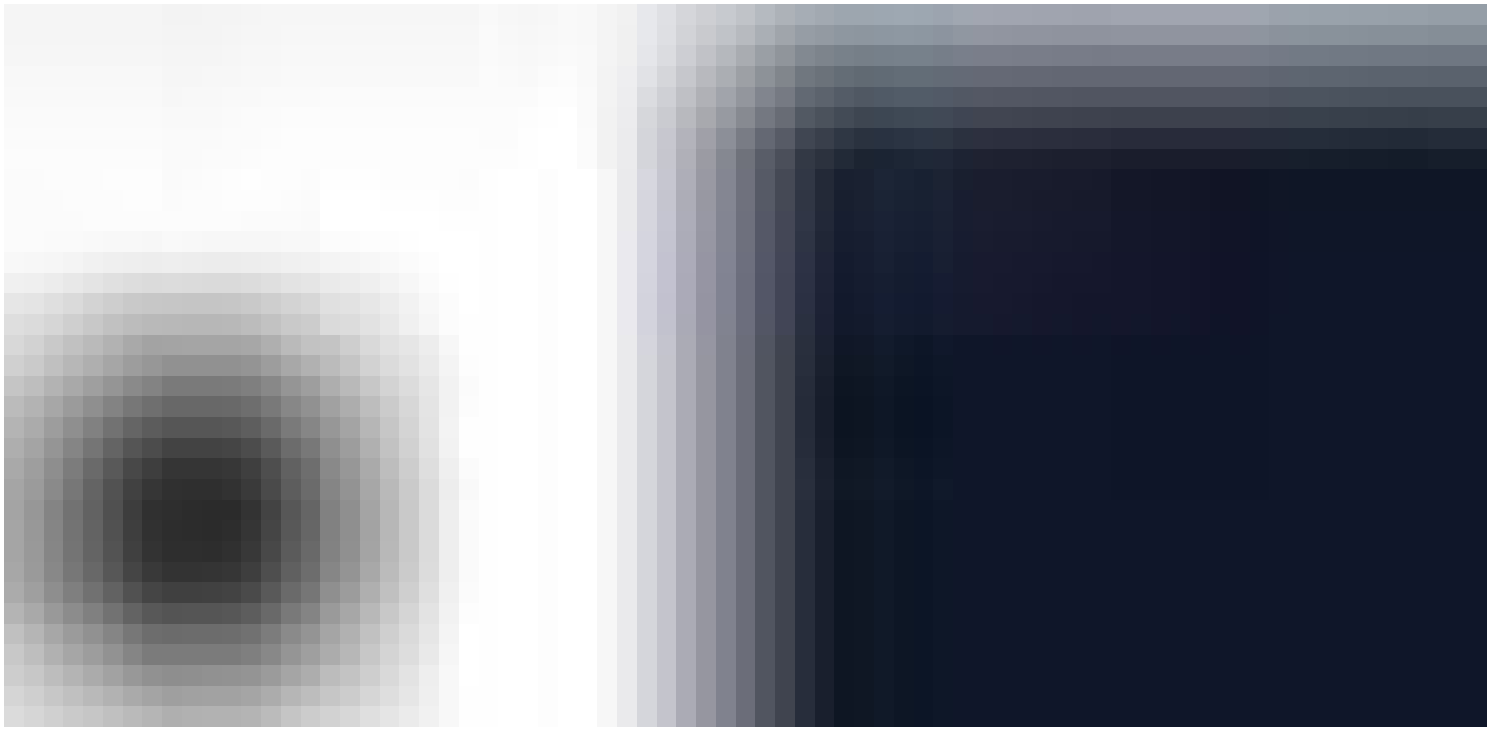
However, it is some what reasonable image that looks like a number. So lets see some more results.



As seen above, as time goes on, the numbers become sharper. A good example is the generated image of 3 or 9.

. . .

Interactive Code



Update: I moved to Google Colab for Interactive codes! So you would need a google account to view the codes, also you can't run read only scripts in Google Colab so make a copy on your play ground. Finally, I will never ask for permission to access your files on Google Drive, just FYI. Happy Coding!

Please click here to access the interactive code, online.

When running the code, make sure you are on the 'main.py' tap, as seen above in the Green Box. The program will ask you a random number for seeding, as seen in the Blue Box. After it will generate one image, to view that image please click on the click me tab above, Red Box.

. . .

Final Words

Training GAN to even partially work is a huge chunk of work, I want to investigate on more effective way of training of GAN's. One last thing, shout out to @replit, these guys are amazing!

If any errors are found, please email me at jae.duk.seo@gmail.com.

Meanwhile follow me on my twitter here, and visit my website, or my Youtube channel for more content. I also did comparison of Decoupled Neural Network here if you are interested.

. . .

References

1. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., … & Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems* (pp. 2672–2680).
2. Free Online Animated GIF Maker—Make GIF Images Easily. (n.d.). Retrieved January 31, 2018, from http://gifmaker.me/
3. Generative Adversarial Nets in TensorFlow. (n.d.). Retrieved January 31, 2018, from https://wiseodd.github.io/techblog/2016/09/17/gan-tensorflow/
4. J. (n.d.). Jrios6/Adam-vs-SGD-Numpy. Retrieved January 31, 2018, from https://github.com/jrios6/Adam-vs-SGD-Numpy/blob/master/Adam%20vs%20SGD%20-%20On%20Kaggles%20Titanic%20Dataset.ipynb
5. Ruder, S. (2018, January 19). An overview of gradient descent optimization algorithms. Retrieved January 31, 2018, from http://ruder.io/optimizing-gradient-descent/index.html#adam
6. E. (1970, January 01). Eric Jang. Retrieved January 31, 2018, from https://blog.evjang.com/2016/06/generative-adversarial-nets-in.html