Generative Adversarial Nets in TensorFlow

Generative Adversarial Nets, or GAN in short, is a quite popular neural net. It was first introduced in a NIPS 2014 paper by Ian Goodfellow, et al (http://papers.nips.cc /paper/5423-generative-adversarial-nets.pdf). This paper literally sparked a lot of interest in adversarial training of neural net, proved by the number of citation of the paper. Suddenly, many flavors of GAN came up: DCGAN, Sequence-GAN, LSTM-GAN, etc. In NIPS 2016, there will even be a whole workshop (https://sites.google.com/site/nips2016adversarial/) dedicated for adversarial training!

Note, the code is available in https://github.com/wiseodd/generative-models (https://github.com/wiseodd/generative-models).

First, let's review the main points about the paper. After that, as always, we will try to implement GAN using TensorFlow, with MNIST data.

Generative Adversarial Nets

Let's consider the rosy relationship between a money conterfeiting criminal and a cop. What's the objective of the criminal and what's the objective of the cop in term of counterfeited money? Let's enumerate:

- To be a successful money counterfeiter, the criminal wants to fool the cop, so

that the cop can't tell the difference between counterfeited money and real money
- To be a paragon of justice, the cop wants to detect counterfeited money as good as possible

There, we see we have a clash of interest. This kind of situation could be modeled as a minimax game in Game Theory. And this process is called Adversarial Process.

Generative Adversarial Nets (GAN), is a special case of Adversarial Process where the components (the cop and the criminal) are neural net. The first net generates data, and the second net tries to tell the difference between the real data and the fake data generated by the first net. The second net will output a scalar `[0, 1]` which represents a probability of real data.

In GAN, the first net is called Generator Net $G(Z)$ and the second net called Discriminator Net $D(X)$.

$$\min_G \max_D V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

At the equilibrium point, which is the optimal point in minimax game, the first net will models the real data, and the second net will output probability of 0.5 as the output of the first net = real data.

"BTW why do we interested in training GAN?" might come in mind. It's because probability distribution of data P_{data} might be a very complicated distribution and very hard and intractable to infer. So, having a generative machine that could generate samples from P_{data} without having to deal with nasty probability distribution is very nice. If we have this, then we could use it for another process that require sample from P_{data} as we could get samples relatively cheaply using the trained Generative Net.

GAN Implementation

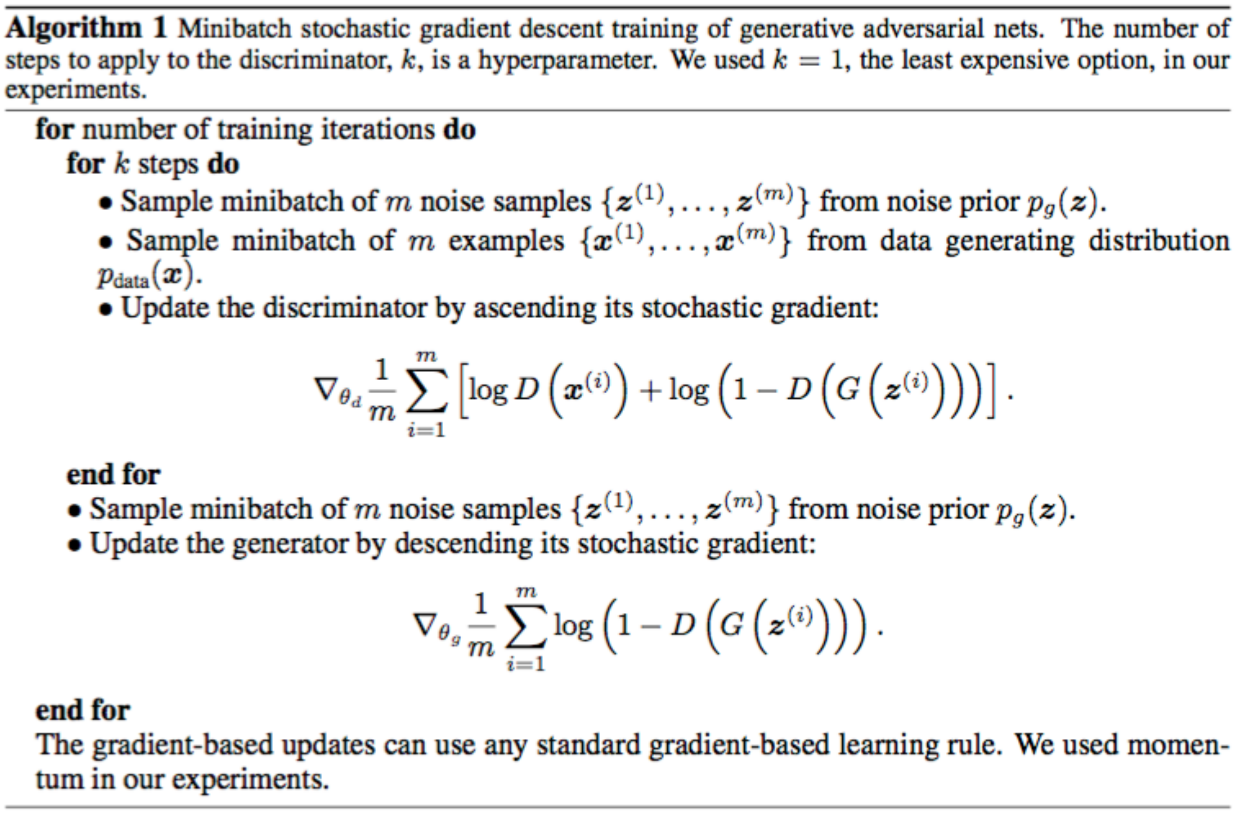
By the definition of GAN, we need two nets. This could be anything, be it a sophisticated net like convnet or just a two layer neural net. Let's be simple first and use a two layer nets for both of them. We'll use TensorFlow for this purpose.

```
# Discriminator Net
X = tf.placeholder(tf.float32, shape=[None, 784], name='X')

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

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

theta_D = [D_W1, D_W2, D_b1, D_b2]

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

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

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

theta_G = [G_W1, G_W2, G_b1, G_b2]


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
```

Above, `generator(z)` takes 100-dimensional vector and returns 786-dimensional vector, which is MNIST image (28x28). `z` here is the prior for the $G(Z)$. In a way it learns a mapping between the prior space to P_{data}.

The `discriminator(x)` takes MNIST image(s) and return a scalar which represents a probability of real MNIST image.

Now, let's declare the Adversarial Process for training this GAN. Here's the training algorithm from the paper:



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 $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
        • Sample minibatch of $m$ examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
        • Update the discriminator by ascending its stochastic gradient:
            $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[ \log D\left(x^{(i)}\right) + \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right) \right].$
    end for
    • Sample minibatch of $m$ noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
    • Update the generator by descending its stochastic gradient:
        $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right).$
end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.
```

```
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))
```

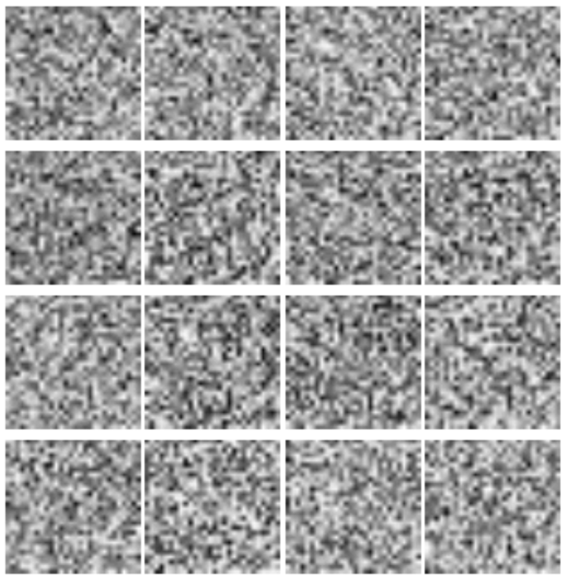
Above, we use negative sign for the loss functions because they need to be maximized, whereas TensorFlow's optimizer can only do minimization.

Also, as per the paper's suggestion, it's better to maximize `tf.reduce_mean(tf.log(D_fake))` instead of minimizing `tf.reduce_mean(1 - tf.log(D_fake))` in the algorithm above.

Then we train the networks one by one with those Adversarial Training, represented by those loss functions above.

```
# Only update D(X)'s parameters, so var_list = theta_D
D_solver = tf.train.AdamOptimizer().minimize(D_loss, var_list=theta_D)
# Only update G(X)'s parameters, so var_list = theta_G
G_solver = tf.train.AdamOptimizer().minimize(G_loss, var_list=theta_G)

def sample_Z(m, n):
    '''Uniform prior for G(Z)'''
    return np.random.uniform(-1., 1., size=[m, n])

for it in range(1000000):
    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,
    _, G_loss_curr = sess.run([G_solver, G_loss], feed_dict={Z: sample_Z(mb_size, Z_dim)})
```

And we’re done! We can see the training process by sampling $G(Z)$ every now and then:



We start with random noise and as the training goes on, $G(Z)$ starts going more and more toward P_{data}. It's proven by the more and more similar samples generated by $G(Z)$ compared to MNIST data.

Alternative Loss Formulation

We could formulate the loss function `D_loss` and `G_loss` using different notion.

Let's follow our intuition. This is inspired by the post about image completion in Brandon Amos' blog (http://bamos.github.io/2016/08/09/deep-completion/).

If we think about it, the `discriminator(X)` wants to make all of the outputs to be `1`, as per definition, we want to maximize the probability of real data. The `discriminator(G_sample)` wants to make all of the outputs to be `0`, as again by definition, $D(G(Z))$ wants to minimize the probability of fake data.

What about `generator(Z)`? It wants to maximize the probability of fake data! It's the opposite objective of $D(G(Z))$!

Hence, we could formulate the loss as follow.

```
# Alternative losses:
# -------------------
D_loss_real = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(D_logit_real, tf.ones_
D_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(D_logit_fake, tf.zeros
D_loss = D_loss_real + D_loss_fake
G_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(D_logit_fake, tf.ones_like(
```

We're using the Logistic Loss, following the notion above. Changing the loss functions won't affect the GAN we're training as this is just a different way to think and formulate the problem.

Conclusion

In this post, we looked at Generative Adversarial Network (GAN), which was published by Ian Goodfellow, et al. at NIPS 2014. We looked at the formulation of Adversarial Process and the intuition behind it.

Next, we implemented the GAN with two layer neural net for both the Generator and Discriminator Net. We then follow the algorithm presented in Goodfellow, et al, 2014 to train the GAN.

Lastly, we thought about the different way to think about GAN loss functions. In the alternative loss functions, we think intuitively about the two networks and used Logistic Loss to model the alternative loss functions.

For the full code, head to https://github.com/wiseodd/generative-models (https://github.com/wiseodd/generative-models)!

References

- Goodfellow, Ian, et al. "Generative adversarial nets." Advances in Neural Information Processing Systems. 2014. (http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf)
- Image Completion with Deep Learning in TensorFlow (http://bamos.github.io/2016/08/09/deep-completion/)

← PREVIOUS POST (/TRAVEL/2016/09/10/KEBAB/)

NEXT POST → (/TECHBLOG/2016/10/13/RESIDUAL-NET/)

21 Comments **Agustinus Kristiadi's Blog** **Login**

Recommend 2 **Share** **Sort by Best**

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS ?

Name

Sangeet Kumar Mishra • 22 days ago

Thanks a lot man ! Really awesome article ! The dynamic plots are really great to visualise the progress

• Reply • Share ›

sean • 3 months ago

Hi! Thanks for the tutorial. I ran the code, and as the training progresses, the outputs of the GAN become all 1s. Do you know why this happens?

• Reply • Share ›

sean sean • 3 months ago

Some reading on the topic suggests this is an example of mode collapse. How can I prevent this?

• Reply • Share ›

wiseodd Mod sean • 3 months ago

Hi sean,
It's indeed mode collapse.
How to prevent it, it's still an open question. Most of the papers about GAN is motivated by this. So, you can try to use newer GAN variants and see if it's help.

• Reply • Share ›

● (/feed.xml) ● (https://twitter.com/wiseodd)
● (https://www.facebook.com/agustinus.kristiadi7)
● (https://github.com/wiseodd)