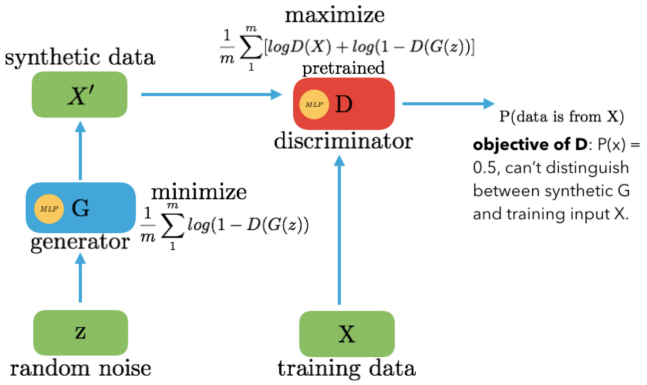
GENERATIVE ADVERSARIAL NETWORKS (GAN)

Edit

This post will cover the basics of the generative adversarial network. Our implementation will be reproducing a simple distribution but I provide some information on extending to complex and more useful cases. The code I have provided is an extension from Eric Jang's **post**.

I have already covered the basic math behind GAN's in this post **here**.

ARCHITECTURE:

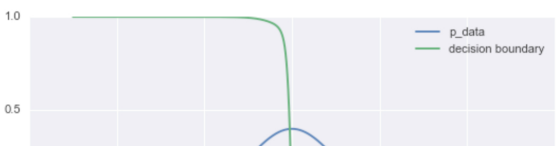


A simplified interpretation would be a minimax situation where we want the discriminator D to correct distinguish between X (training data) and G(z) (result from random noise) . However, we want our generator G to train to produce results for D, so that D is not able to distinguish between z and X. **D()** always gives us the probability that a given sample is from training data X. So for our generator, we want to minimize log (1-D(G(z)) [a high D(G(z)) means that D thinks G(z) is X, which makes 1-D(G(z)) very low, so we want to minimize this to make it even lower]. As for the discriminator we want to maximize D(X) and (1-D(G(z)). We want both sides to "win" so the optimal state for D will be **P(x) = 0.5** (it is not able to distinguish between x and G(z)).

CODE BREAKDOWN:

We will start by viewing the preliminary plots of our true data distribution as well as the decision boundary for our discriminator D without any training.

```
def plot_data_and_D(sess, model, FLAGS):

    # True data distribution with untrained D
    f, ax = plt.subplots(1)

    # p_data
    X = np.linspace(int(FLAGS.mu-3.0*FLAGS.sigma),
                    int(FLAGS.mu+3.0*FLAGS.sigma),
                    FLAGS.num_points)
    y = norm.pdf(X, loc=FLAGS.mu, scale=FLAGS.sigma)
    ax.plot(X, y, label='p_data')

    # Untrained p_discriminator
    untrained_D = np.zeros((FLAGS.num_points,1))
    for i in range(FLAGS.num_points/FLAGS.batch_size):
        batch_X = np.reshape(
            X[FLAGS.batch_size*i:FLAGS.batch_size*(i+1)],
            (FLAGS.batch_size,1))
        untrained_D[FLAGS.batch_size*i:FLAGS.batch_size*(i
            sess.run(model.D,
                feed_dict={model.pretrained_inputs: batch_
    ax.plot(X, untrained_D, label='untrained_D')

    plt.legend()
    plt.show()
```



- SEARCH -

Search …

- FOLLOW ME ON TWITTER -

Tweets by @GokuMohandas

Goku Mohandas
@GokuMohandas

Your model may be performing really well by incorrectly focusing on confounding features (extraneous influencers in the data that aren't accounted for). Check out @johnrzech's post where x-ray stickers unintentionally influenced the classifications: medium.com/@jrzech/what-a…

Embed View on Twitter

- RECENT POSTS -

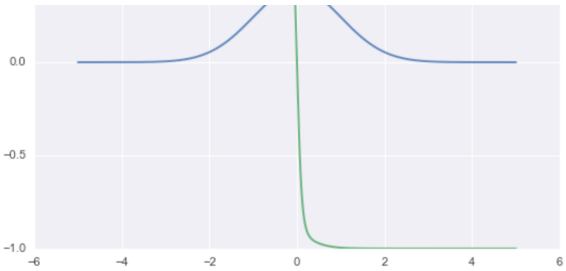
Update on Embeddings (Spring 2017)

Exploring Sparsity in Recurrent Neural Networks

Question Answering from Unstructured Text by Retrieval and Comprehension

Overcoming Catastrophic Forgetting in Neural Networks

Opening the Black Box of Deep Neural Networks via Information

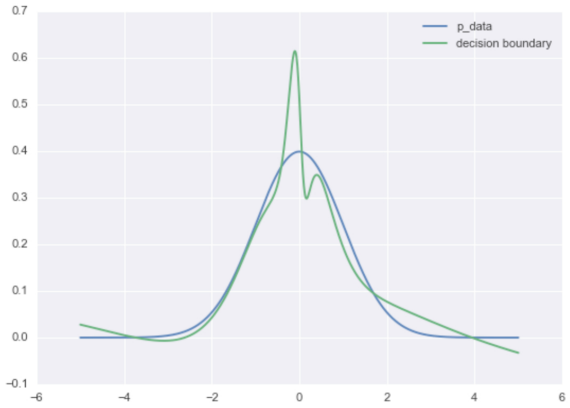


For any point X, the role of the discriminator is to determine probability that it is from the training data. The blue line is our true distribution, for any X we use pdf to determine probability that the particular X belongs to our distribution. The discriminator in this case was a simple MLP. We feed in the same X and it returns a probability for which it believes X is from the training data (true distribution). Right now these two plots are very different because we have not done any training yet.

Now let's pretrain our D so that it knows what the true distribution looks like. We want to do this before feeding in any inputs from our generator so D can easily (at least initially) reject the samples from the generator with great confidence which will allow G to train faster (larger error to propagate).

```
# Let's train the discriminator D
losses = np.zeros(FLAGS.num_points)
for i in xrange(FLAGS.num_points):
    batch_X = (np.random.uniform(int(FLAGS.mu-3.0*FLAGS.si
                                 int(FLAGS.mu+3.0*FLAGS.sig
                                 FLAGS.batch_size))
    batch_y = norm.pdf(batch_X, loc=FLAGS.mu, scale=FLAGS.
    batch_X = np.reshape(batch_X, (-1,1))
    batch_y = np.reshape(batch_y, (-1,1))
    D, theta_D, losses[i], _ = pretrain_D.step_pretrain_D(
                                                          ba
                                                          ba
```

Here is the trained D's decision boundary:



Now we are ready to starting generating synthetic data in order to feed into the discriminator. The objective will be to change the D's decision boundary from above to a flat line at y=0.5 (P(x)=0.5). Which means that it cannot differentiate between the training data and the synthetic data or any given X, which means our generator has been trained well.

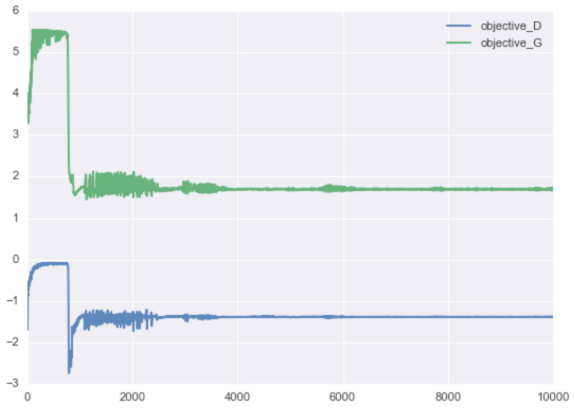
Before we take a look at the GAN code, I want to go over the objective of G in a bit more detail. We have some random noise z which needs to be mapped to a synthetic X' by G. But what does this mean? At any X_n, we want D to predict the p_data probability (height) for that point X_n. Now with G, we want G to map most of the z's to the X's in the middle of the distribution because that is where the probability is high (in our normal dist). So G is learning to map most of z's to the ceoncentrated high probability X's and only a few of the z's will point to the edges of p_data. This will help G learn how p_data behaves and D will slowly not be able to tell where the X's are coming from G is starting to always produce those high probability X locations, as p_data has been doing from the get-go.

This learning (mapping) of z to high probability X is easier when we scale the outputs from G to the range of X (so G does not have to learn it) and this idea of manifold alignment (learning the underlying features of a distribution) is made easier when we also sort our X and z. **Note**: You will see below, z is not completely random, but we do stratified sampling where we take values from a sorted range and add a little bit of random noise (not enough to disorient the order).

```
# Let's train the GAN
k=1
objective_Ds = np.zeros(FLAGS.num_epochs)
objective_Gs = np.zeros(FLAGS.num_epochs)
for i in xrange(FLAGS.num_epochs):

```

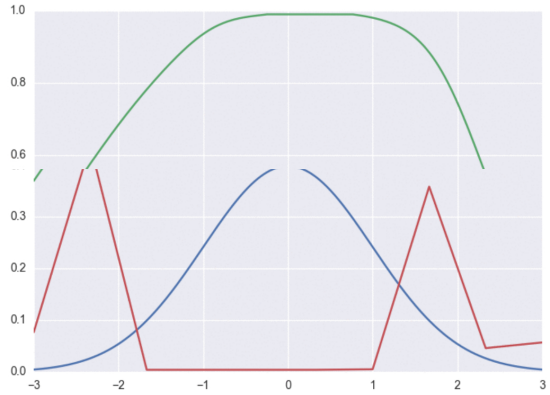
```
    if i%1000 == 0:
        print i/float(FLAGS.num_epochs)

    # k updates to discriminator
    for j in xrange(k):
        batch_X = np.random.normal(FLAGS.mu,
                                   FLAGS.sigma,
                                   FLAGS.batch_size)
        batch_X.sort()
        batch_X = np.reshape(batch_X, (FLAGS.batch_size, 1
        batch_z = np.linspace(int(FLAGS.mu-3.0*FLAGS.sigma
                              int(FLAGS.mu+3.0*FLAGS.sigma
                            FLAGS.batch_size) + \
                            np.random.random(FLAGS.batch_s
        batch_z = np.reshape(batch_z, (FLAGS.batch_size, 1

        ''' Both batch_X and batch_Z are sorted for manifo
        Manifold alignment
        (https://sites.google.com/site/changwangnk/home/ma
        G learn the underlying structure of p_data. By sor
        making this process easier, since now adjacent poi
        will directly map to adjacent points in X (and eve
        will match since we scaled outputs from G.)
        '''

        objective_Ds[i], _ = GAN.step_D(sess, batch_z, bat

    # 1 update to G
    batch_z = np.linspace(int(FLAGS.mu-3.0*FLAGS.sigma),
                          int(FLAGS.mu+3.0*FLAGS.sigma),
                            FLAGS.batch_size) + \
                            np.random.random(FLAGS.batch_s
    batch_z = np.reshape(batch_z, (FLAGS.batch_size, 1))
    objective_Gs[i], _ = GAN.step_G(sess, batch_z)
```

As we train, here are the changes to the objectives:



Note that for every K updates to D, we do 1 update to G. This is to prevent overfitting and here k=1 because the distribution is not complex. But as we will see in subsequent GAN implementations, we will need to increase k (~5) in order to learn the true distribution.

RESULTS:



Visually, we can see that our p_g is similar to p_data (discrepancy is height is minimal). And we have reached our objective where the decision boundary is 0.5 across the range of the distributions. Therefore, D is not able to discriminate which distribution a given input is from. **NOTE:** GANs are known to be VERY tricky to train. Take a look at the momentum optimizer here, try changing the optimizer or few of the parameters and you will see how fragile the system is. This is an active area os research right now, check out this **paper** for more information.

NUANCES:

Here are two parts that may seem tricky/different from the previous tensorflow code I have

posted. First is using scope within the same scope:

```
with tf.variable_scope("D") as scope:
    # Feed X into D
    self.X = tf.placeholder(tf.float32,
        shape=[FLAGS.batch_size, 1], name="X")
    D_X = mlp(self.X)
    # Scale the prediction
    self.D_X = tf.maximum(tf.minimum(D_X, 0.99), 0.01)

    scope.reuse_variables()

    # Feed X' into D
    D_X_prime = mlp(self.G)
    # Scale the prediction
    self.D_X_prime = tf.maximum(tf.minimum(D_X_prime, 0.99
```

We need to say **scope.reuse_variables()** because we use mlp twice under the same scope. This will create a conflict since D/__ already exists, but we can reuse() the same weights because they both belong to the discriminator.

Second, is taking the weights from one scope and sharing it to another without the scopes having the same name. We needed to explicitly do it here for the optimizer to properly decay.

```
vars_ = tf.trainable_variables()
self.theta_G = [v for v in vars_ if v.name.startswith('G/')
self.theta_D = [v for v in vars_ if v.name.startswith('D/')
...
# Initialize weights for D from pretrain_D
for i,v in enumerate(GAN.theta_D):
    sess.run(v.assign(theta_D[i]))
```

theta_D comes from the pretrained_D and GAN.theta_D is the mlp weights in our GAN for the discriminator

EXTENSIONS:

There are many useful/cool applications that have come out of GANs and many we have yet to see. But one my favorite is detailed in my post **here**, which covers using deep convolutional GANs for unsupervised learning.

GITHUB REPO:

Repo (Updating all repos, will be back up soon!)

Posted in: *Generative Adversarial Networks*
Tagged: *Tutorials*

← *Fully Character-Level Neural Machine Translation without Explicit Segmentation*

Image Recognition →

3 THOUGHTS ON "GENERATIVE ADVERSARIAL NETWORKS (GAN)"

Pingback: Improved Techniques for Training GANs – The Neural Perspective Edit

BOB March 12, 2017 at 3:09 pm EDIT REPLY →

Really great post! Do you have the whole code posted somewhere? (I mean instead of small code snippets disjoint from each other).

★ Like

Pingback: Generative Adversarial Nets 原理-大数据算法 Edit

LEAVE A REPLY

Enter your comment here...