Generative Adversarial Networks

Members：朱偉綸、黃秉茂、許齊崴

Trace Code

R: The original, genuine data set

I: The random noise that goes into the generator as a source of entropy

G: The generator which tries to copy/mimic the original data set

D: The discriminator which tries to tell apart G's output from R

The actual 'training' loop where we teach G to trick D and D to beware G.

R

In our case, we’ll start with the simplest possible R (a bell curve). This function takes a mean and a standard deviation and returns a function which provides the right shape of sample data from a Gaussian with those parameters.

In our sample code, we’ll use a mean of 4.0 and a standard deviation of 1.25.

```
#真實資料分布 Target Data(使用高斯分布)
# Gaussian
def get_distribution_sampler(mu, sigma):
    return lambda n: torch.Tensor(np.random.normal(mu, sigma, (1, n)))
```

I

The input into the generator is also random, but to make our job a little bit harder, let's use a uniform distribution rather than a normal one.

This means that our model G can't simply shift/scale the input to copy R, but has to reshape the data in a non-linear way.

```
#Generator輸入資料
# Uniform-dist data into generator, _NOT_ Gaussian
def get_generator_input_sampler():
    return lambda m, n: torch.rand(m, n)
```

G

The generator is a standard feedforward graph — two hidden layers, three linear maps. We’re using a hyperbolic tangent activation function.

G is going to get the uniformly distributed data samples from I and somehow mimic the normally distributed samples from R (without ever seeing R).

```python
#Generator Model
class Generator(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, f):
        super(Generator, self).__init__()
        self.map1 = nn.Linear(input_size, hidden_size) #輸入層
        self.map2 = nn.Linear(hidden_size, hidden_size) #隱藏層
        self.map3 = nn.Linear(hidden_size, output_size) #輸出層
        self.f = f

    #Feedforward Network
    def forward(self, x):
        x = self.map1(x) #輸入層
        x = self.f(x)
        x = self.map2(x) #隱藏層
        x = self.f(x)
        x = self.map3(x) #輸出層
        return x
```

D

The discriminator code is very similar to G's generator code; a feedforward graph with two hidden layers and three linear maps.

The activation function here is a sigmoid.

It's going to get samples from either R or G and will output a single scalar between 0 and 1, interpreted as 'fake' vs. 'real'.

In other words, this is about as milquetoast as a neural net can get.

```python
#Discriminator Model
class Discriminator(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, f):
        super(Discriminator, self).__init__()
        self.map1 = nn.Linear(input_size, hidden_size) #輸入層
        self.map2 = nn.Linear(hidden_size, hidden_size) #隱藏層
        self.map3 = nn.Linear(hidden_size, output_size) #輸出層
        self.f = f

    #Feedforward Network
    def forward(self, x):
        x = self.f(self.map1(x))
        x = self.f(self.map2(x))
        x = self.f(self.map3(x))
        return x
```

Finally, the training loop alternates between two modes: first training D on real data vs. fake data, with accurate labels, and then training G to fool D, with inaccurate labels . It’s a fight between real and fake, people.

For each epoch:

(1) We push both types of data through D and apply a differentiable criterion to D's guesses vs. the actual labels. That pushing is the 'forward' step; we then call 'backward()' explicitly in order to calculate gradients, which are then used to update D's parameters in the d_optimizer step() call. G is used but isn't trained here.

(2)we do the same thing for G — note that we also run G's output through D (we're essentially giving the forger a detective to practice on) but we do not optimize or change D at this step. We don't want the detective D to learn the wrong labels. Hence, we only call g_optimizer.step().

Training D(discriminator)

```
for d_index in range(d_steps):

    #先訓練Discriminator在Real & Fake Data
    D.zero_grad()

    #  1A: Train Discriminator on real
    d_real_data = Variable(d_sampler(d_input_size))
    d_real_decision = D(preprocess(d_real_data))
    d_real_error = criterion(d_real_decision, Variable(torch.ones([1,1])))  # ones = true
    d_real_error.backward() # compute/store gradients, but don't change params

    #  1B: Train Discriminator on fake
    d_gen_input = Variable(gi_sampler(minibatch_size, g_input_size))
    d_fake_data = G(d_gen_input).detach()  # detach to avoid training G on these labels
    d_fake_decision = D(preprocess(d_fake_data.t()))
    d_fake_error = criterion(d_fake_decision, Variable(torch.zeros([1,1])))  # zeros = fake
    d_fake_error.backward()
    d_optimizer.step()     # Only optimizes D's parameters; changes based on stored gradients from backward()

    dre, dfe = extract(d_real_error)[0], extract(d_fake_error)[0]
```

Training G(generator)

```
for g_index in range(g_steps):
    #根據Discriminator的回饋來訓練Generator
    G.zero_grad()

    gen_input = Variable(gi_sampler(minibatch_size, g_input_size))
    g_fake_data = G(gen_input)
    dg_fake_decision = D(preprocess(g_fake_data.t()))
    g_error = criterion(dg_fake_decision, Variable(torch.ones([1,1])))  # Train G to pretend it's genuine

    g_error.backward()
    g_optimizer.step()  # Only optimizes G's parameters
    ge = extract(g_error)[0]
```

Key Point

Hyper parameter

```
# Model parameters
g_input_size = 1      # Random noise dimension coming into generator, per output vector
g_hidden_size = 5     # Generator complexity
g_output_size = 1     # Size of generated output vector
d_input_size = 500   # Minibatch size - cardinality of distributions
d_hidden_size = 10   # Discriminator complexity
d_output_size = 1    # Single dimension for 'real' vs. 'fake' classification
minibatch_size = d_input_size

d_learning_rate = 1e-3
g_learning_rate = 1e-3
sgd_momentum = 0.9

num_epochs = 5000
print_interval = 100
d_steps = 20
g_steps = 20

dfe, dre, ge = 0, 0, 0
d_real_data, d_fake_data, g_fake_data = None, None, None

#Discriminator使用Sigmoid Function
discriminator_activation_function = torch.sigmoid

#Gnerator使用Tangent Hyperbolic Function
generator_activation_function = torch.tanh
```

D(discriminator)'s Implementation Detail

Generate real data.

Training D with real data.

Calculate real loss and then calculate gradient with it.

Generate fake data from data which generate from G(generator).

Training D with fake data.

Calculate fake loss and then calculate gradient with it.

Update D’s parameter according to the gradient generated by real loss and fake loss.

D(discriminator)'s Vision

Hope the real loss is as small as possible

=> represents that it can distinguish which data is real more precisely

Hope the fake loss is as small as possible

=> represents that it can distinguish which data is fake more precisely

The smaller the loss is, the stricter the G (generator) will be

G(generator)'s Implementation Detail

Generate data.

Score D with data.

Calculate loss and then calculate gradient with it.

Update G’s parameter according to the gradient generated by loss.

G(generator)'s Vision

Smaller loss is better => it means that the generated data is fitter for our expectation.

The smaller the loss is, the more able to fool D (discriminator) to achieve a high score.

Other Implementation Key Point

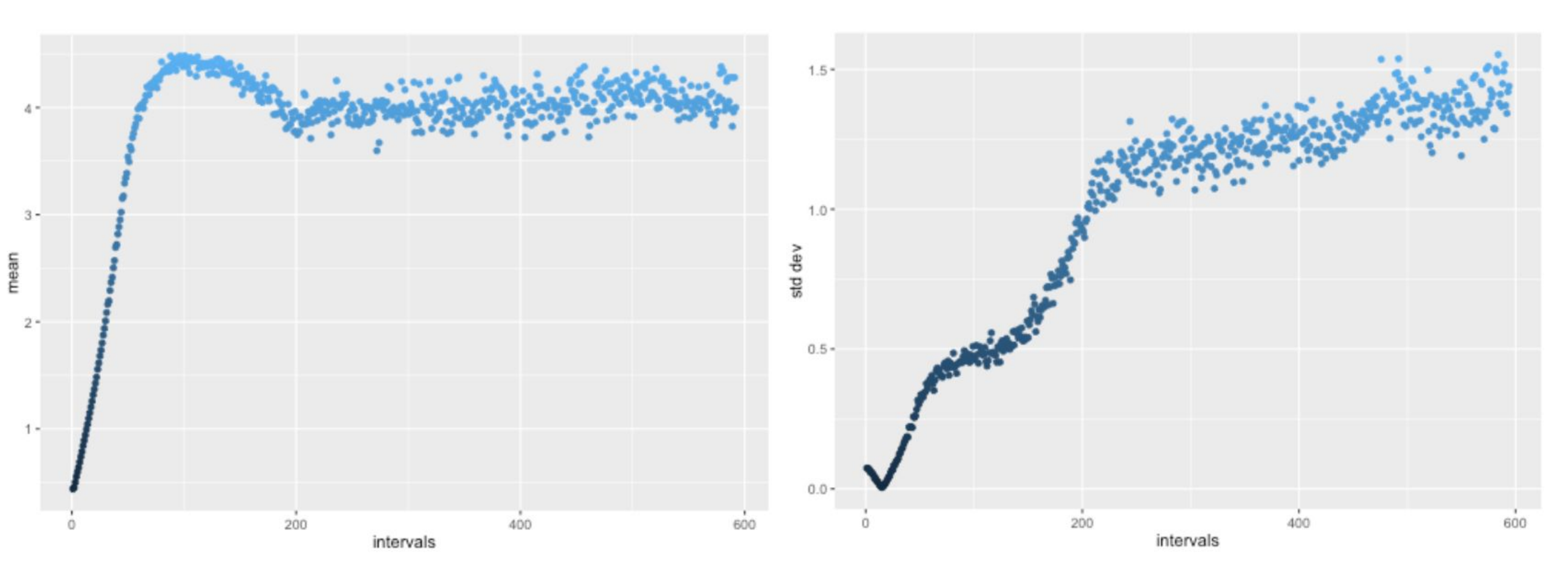
Do not update G when updating D, vice versa.

Remember using zero_grad(), or the calculation of gradient will fail.

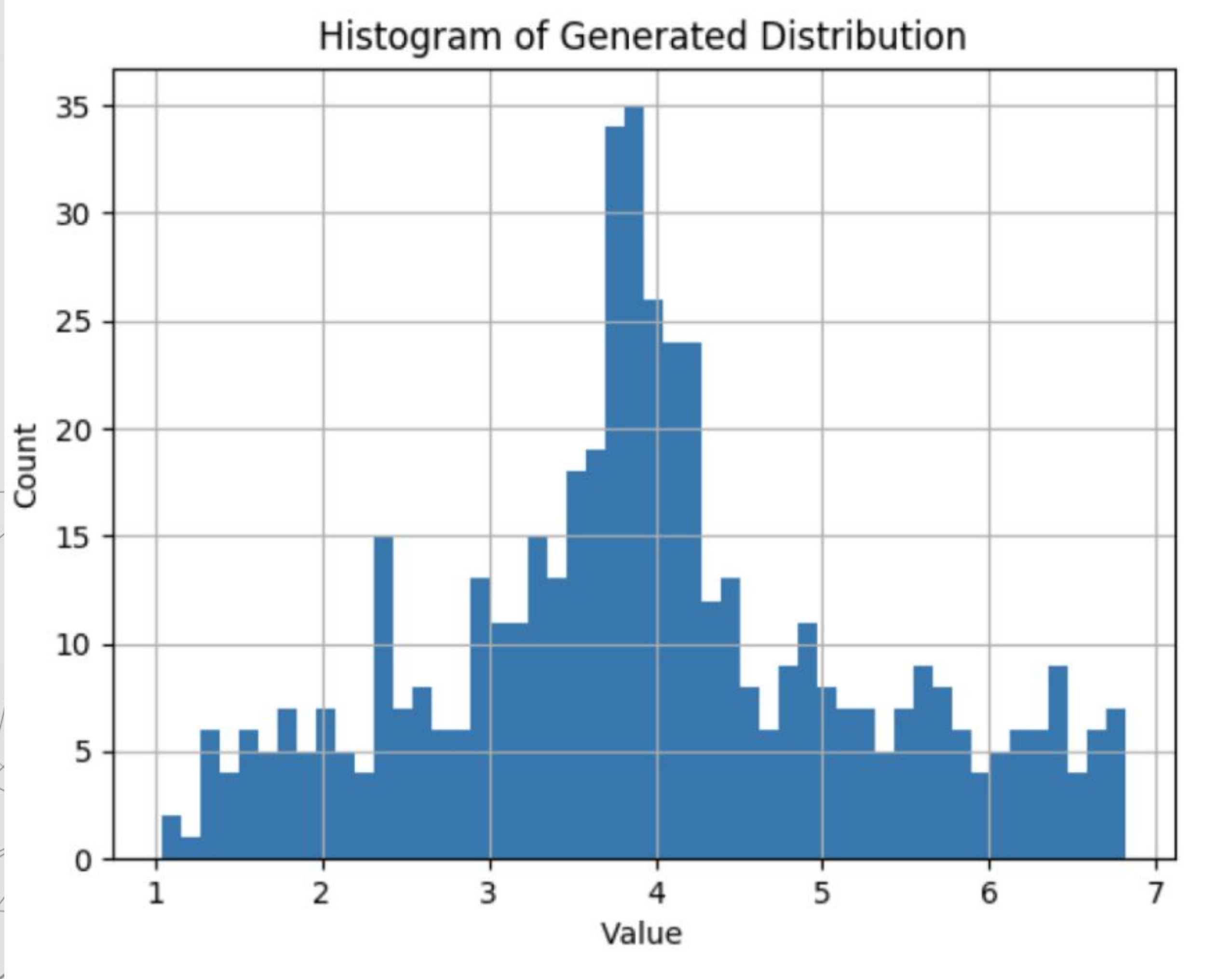
Since back propagation can deal with chain rule, we can only calculate gradient of data’s statistic.

Result

Over 5,000 training rounds, training D for 20 times and then G for 20 times in each round, the mean of G's output overshoots 4 but then comes back in a fairly stable, correct range (left picture above in page). Likewise, the standard deviation initially drops in the wrong direction but then rises up to the desired 1.25 range (right picture in above page), matching R.



mean
4
3
2
1
0
200
400
600
intervals
std dev
1.5
1.0
0.5
0.0
0
200
400
600
intervals



Histogram of Generated Distribution
Count
Value
0
5
10
15
20
25
30
35
1
2
3
4
5
6
7

Conclusion

The power of GAN (Generative Adversarial Networks) is that it only needs to know the real data, and it can learn what he wants through the interaction between G and D, and we may be able to understand how and what the model would learn through the scoring mechanism of D. Moreover, perhaps its censorship standard is something that humans have never thought of, and therefore it may be more reasonable than human judgments in some aspects.

Resource

https://github.com/devnag/pytorch-generative-adversarial-networks?fbclid=IwAR355HFMPWE4i12-L51rmKEuLhe_GEEjsssJp87Y-DB7ILEuMv1II-FKwg8

Thanks for listening