[GAN] 1D Gaussian Distribution Generation

요약

연세대학교 빅데이터 학회 YBIGTA의 GAN팀에서 준비한 두번째 시간에는 GAN을 활용하여 1D Gaussian Distribution을 generate하는 튜토리얼을 살펴보았다.

목차

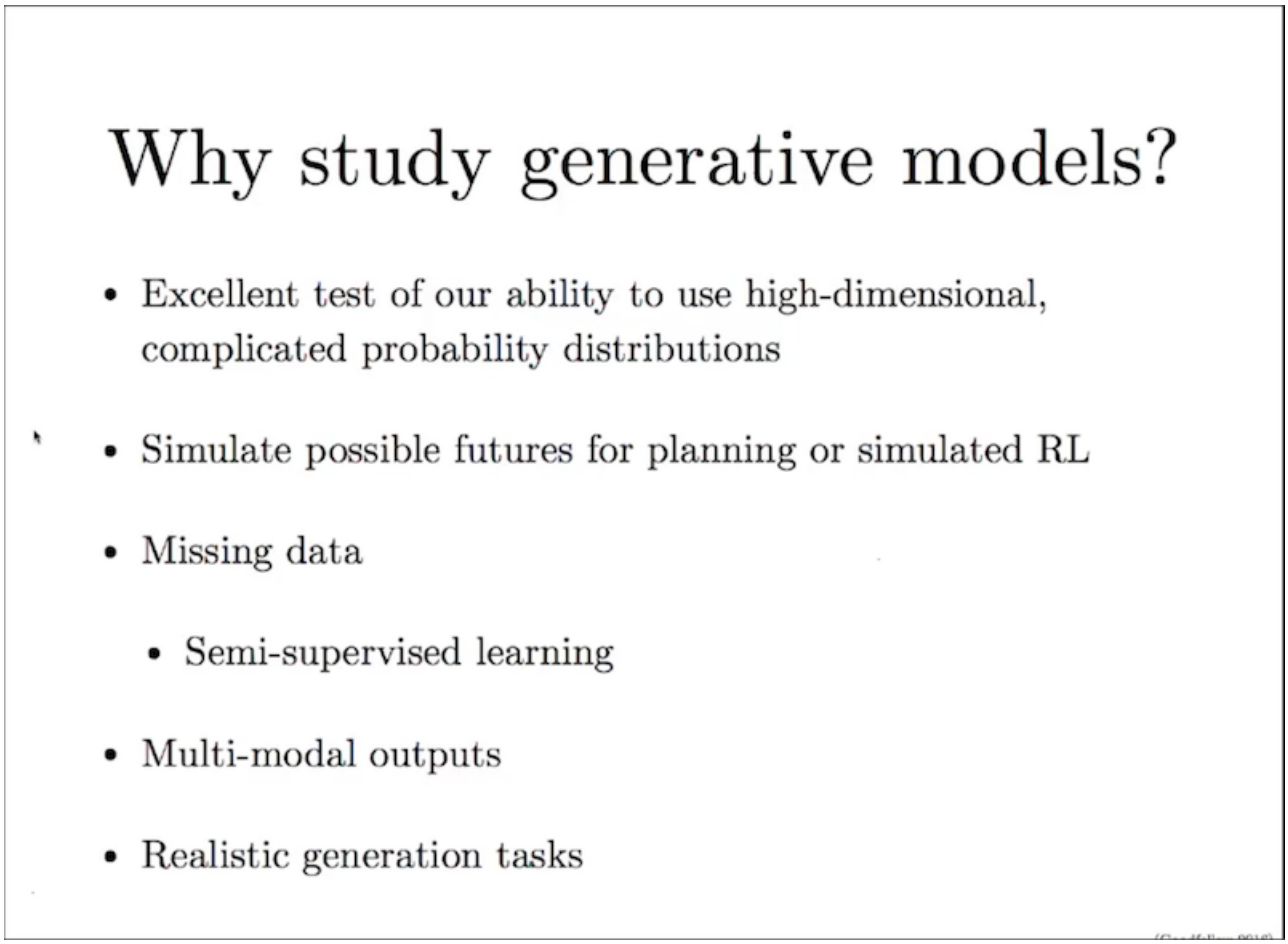
GAN?

GAN에 대해 간단히 복습하자면, GAN(Generative Adversarial Nets)이란 서로 대립하는 두 개의 신경망을 동시에 학습시키면서 원본의 sample과 유사한 sample을 만들어내는 방식을 말한다.

가장 중요한 두 개의 역할이 바로 **Generator** 와 **Discriminator** 인데, 지폐 위조 범죄에 비추어 예시를 들자면, Generator는 실제 지폐와 똑같이 생긴 지폐를 만들려고 노력하는 **위조지폐범** 이고 Discriminator는 이를 제대로 구별하려는 **경찰** 이다.

기존의 하나의 cost function을 가지고 이를 최적화 했던 다른 신경망 방식과 다르게, 두 개의 신경망을 동시에 학습시키면서 Generator는 Discriminator의 구분 확률을 최대한 줄이고, 이와 동시에 Discriminator는 real sample(실제 지폐)와 fake sample(위조 지폐)의 구분 정확도를 높이는 목적을 두고 있다.

NIPS2016에서 저자인 Ian Goodfellow가 발표한 내용에 따르면, GAN은 다음과 같은 상황에서 매우 뛰어난 성능을 보여준다.



가장 핵심은, GAN은 기존의 머신러닝 기법과 같이 원데이터의 분포에 대해 직접적으로 알아내고 분석한다기보다는, 원데이터와 최대한 비슷한 sample을 만들어내는 데에만 목표를 둔다는 점이다. 따라서 원데이터가 매우 복잡하고 고차원이거나, 결측치가 많거나, 원데이터가 무한히 많을 때 좋은 성능을 보여준다.

더 자세한 내용은 저번 포스트(Link)를 참고하기 바란다.

1D Gaussian Distribution

GAN 첫번째 튜토리얼로는 정규분포를 만든 다음에 이를 GAN을 활용해서 최대한 비슷하게 generate해보았다. 작업 환경은 다음과 같다.

- Python : 3.6.1
- Pytorch : 0.1.12
- OS : MAC OSX

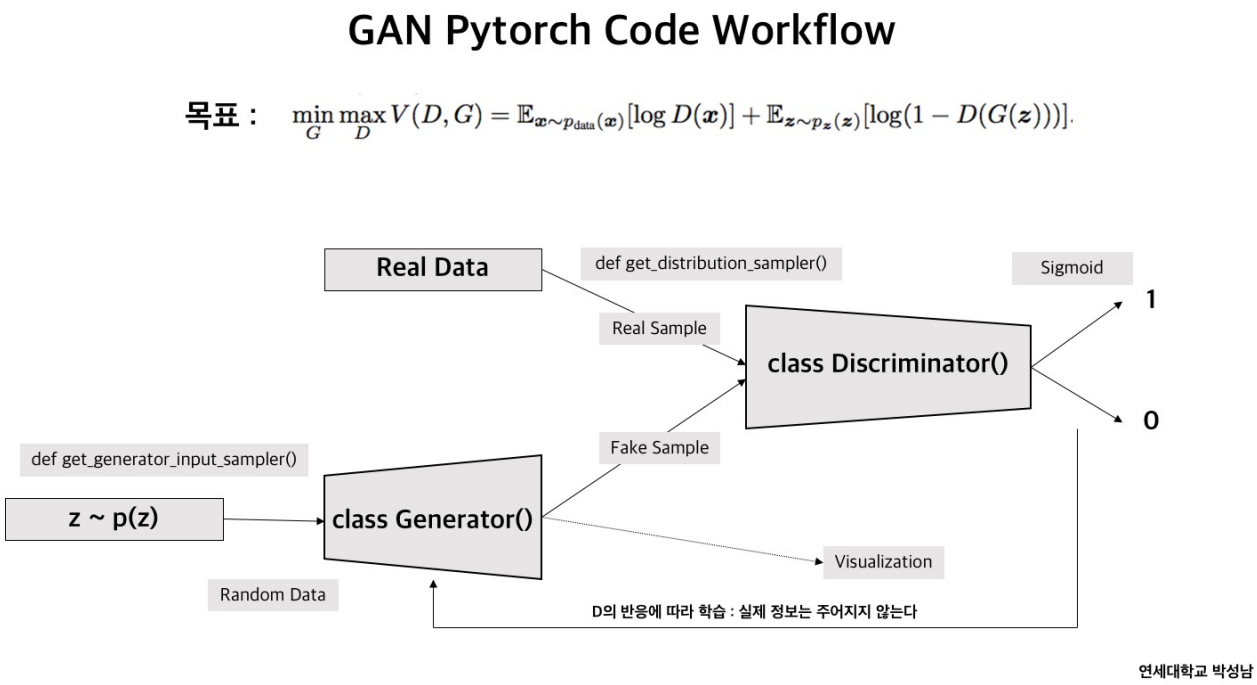
Why Pytorch?

이 튜토리얼과 관련하여 TensorFlow, Keras, Pytorch로 구현한 모든 github 예제를 분석해보았는데, 처음엔 TensorFlow 코드를 보고 이를 Pytorch로 바꾸어볼려고 했지만, 둘 다 사용법이 미숙하니 시간상으

로 도저히 안되겠다는 것을 느꼈다. 그래서 (*팀장님이 추천해주시기도 했고*) 코드가 비교적 간단한 pytorch로 보기로 했다. 이거 결정하는 데만 시간이 엄청 걸렸다..ㅠ

Workflow

코드의 workflow는 다음과 같다.



먼저 Generator의 입장에서 말하자면, 입력값은 0과 1사이의 임의의 값이고, 이를 노이즈를 줘서 fake sample을 만들어 Discriminator에게 보낸다. 후에, Discriminator의 반응에 따라 신경망을 최적화하는데, 즉 어떤 sample에 대해서는 1(real)이라 반응했다 / 어떤 sample은 0(fake)이라 반응했다 의 여부에 따라 Generator가 최적화되는 것이다. 인상깊은 점은 **Generator는 sample이 실제로 real sample인지 fake sample인지 모른다.** 다만 Discriminator가 어떻게 평가했는지의 여부만 보고 최적화하는 것이다.

다음으로 Discriminator의 입장에서 말하지면, 입력값은 real data와 fake data이고 이들이 실제로 real인지 fake인지의 여부이다. 일단 본인이 알아서 평가한 후에, 실제 real/fake 여부에 따라 본인을 최적화한다. 이 때 본인이 평가한 정보를 Generator에게 보낸다.

Algorithm

필요한 라이브러리 불러오기

```
import numpy as np
import torch

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable

import cufflinks as cf
import plotly
plotly.tools.set_credentials_file(username='your_username', api_key='your_api
```

파라미터 선정

```
# 모집단의 평균, 표준편차 지정 : 실제 지폐의 모양
<p class="mume-header " id="모집단의-평균-표준편차-지정-실제-지폐의-모양"></p>

data_mean = 4
data_stddev = 1.25

# 모델의 파라미터 지정
<p class="mume-header " id="모델의-파라미터-지정"></p>

g_input_size = 1     # Random noise dimension coming into generator, per outp
g_hidden_size = 50   # G의 레이어 크기
g_output_size = 1    # G의 결과값의 크기 : (1,N) - 위조 지폐의 크기
d_input_size = 100   # Minibatch size - cardinality of distributions
d_hidden_size = 50   # D의 레이어 크기
d_output_size = 1    # 진짜인지 가짜인지의 여부(확률값) : (1,1)
minibatch_size = d_input_size

d_learning_rate = 2e-4  # 2e-4
g_learning_rate = 2e-4
optim_betas = (0.9, 0.999)
num_epochs = 2400
print_interval = 10
d_steps = 1  # 'k' steps in the original GAN paper. Can put the discriminator
g_steps = 1
```

데이터 입력 관련 함수들

```
# preprocess : data를 입력받고 preprocessing
<p class="mume-header " id="preprocess-data를-입력받고-preprocessing"></p>

(name, preprocess, d_input_func) = ("Data and variances", lambda data: decorc

print("Using data [%s]" % (name))

# ##### DATA: Target data and generator input data
<p class="mume-header " id="data-target-data-and-generator-input-data"></p>


# 한번에 바로 평균이 mu이고 표준편차가 sigma인 정규분포에서 샘플 (1,n)개 뽑기
<p class="mume-header " id="한번에-바로-평균이-mu이고-표준편차가-sigma인-정규분포에서-샘플

def get_distribution_sampler(mu, sigma):
    return lambda n: torch.Tensor(np.random.normal(mu, sigma, (1, n)))  # Gau

# 0부터 1사이의 숫자중 임의로 샘플 (m,n)개 뽑기
<p class="mume-header " id="0부터-1사이의-숫자중-임의로-샘플-mn개-뽑기"></p>

def get_generator_input_sampler():
    return lambda m, n: torch.rand(m, n)  # Uniform-dist data into generator,
```

Generator & Discriminator

```
class Generator(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(Generator, self).__init__()
        self.map1 = nn.Linear(input_size, hidden_size)
        self.map2 = nn.Linear(hidden_size, hidden_size)
        self.map3 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = F.elu(self.map1(x))
        x = F.sigmoid(self.map2(x))
        return self.map3(x)

class Discriminator(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(Discriminator, self).__init__()
        self.map1 = nn.Linear(input_size, hidden_size)
        self.map2 = nn.Linear(hidden_size, hidden_size)
        self.map3 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = F.elu(self.map1(x))
        x = F.elu(self.map2(x))
        return F.sigmoid(self.map3(x))
```

기타 함수들

```
# data의 정보를 리스트화하여 반환
<p class="mume-header " id="data의-정보를-리스트화하여-반환"></p>

def extract(v):
    return v.data.storage().tolist()

# data의 평균과 표준편차 반환
<p class="mume-header " id="data의-평균과-표준편차-반환"></p>

def stats(d):
    return [np.mean(d), np.std(d)]

# data를 입력받으면, data에 정규화한 data를 가로로 붙여서 반환
<p class="mume-header " id="data를-입력받으면-data에-정규화한-data를-가로로-붙여서-반환"

def decorate_with_diffs(data, exponent):
    # torch.mean(data,1) : (m,n)크기의 샘플을 각 행별로 평균을 구해서 (m,1)크기로 반환
    mean = torch.mean(data.data, 1)

    # torch.ones(m,n) : (m,n)크기의 1로만 이루어진 벡터
    # mean.tolist()[0][0] : data의 1행의 평균
    # mean_broadcast : data와 같은 크기의 벡터, 모든 성분은 data의 1행의 평균으로 구성됨
    mean_broadcast = torch.mul(torch.ones(data.size()), mean.tolist()[0][0])

    # diffs : dat에서 mean_broadcast를 뺀 후 이를 exponet제곱한다
    diffs = torch.pow(data - Variable(mean_broadcast), exponent)

    # torch.cat : concat의 의미, 0이면 세로로 붙이고, 1이면 가로로 붙임
    # return : data와 diff를 이어서 반환
    return torch.cat([data, diffs], 1)

# d_sampler : data의 평균과 표준편차와 같은 평균과 표준편차를 가지는 정규분포에서 sample뽑는 함
<p class="mume-header " id="d_sampler-data의-평균과-표준편차와-같은-평균과-표준편차를-가

d_sampler = get_distribution_sampler(data_mean, data_stddev)

# 임의의 0에서 1사이의 N개의 sample 뽑는 함수 : gi_sampler(N)
<p class="mume-header " id="임의의-0에서-1사이의-n개의-sample-뽑는-함수-gi_samplern">

gi_sampler = get_generator_input_sampler()
```

G,D,optimizer 정의

```
# G와 D 정의
<p class="mume-header " id="g와-d-정의"></p>

G = Generator(input_size=g_input_size, hidden_size=g_hidden_size, output_size
D = Discriminator(input_size=d_input_func(d_input_size), hidden_size=d_hidder

criterion = nn.BCELoss()  # Binary cross entropy: http://pytorch.org/docs/nn.

# Optimizer 정의
<p class="mume-header " id="optimizer-정의"></p>

d_optimizer = optim.Adam(D.parameters(), lr=d_learning_rate, betas=optim_betc
g_optimizer = optim.Adam(G.parameters(), lr=g_learning_rate, betas=optim_betc
```

학습 시작

```
    for epoch in range(num_epochs):
        for d_index in range(d_steps):
            # 1. Train D on real+fake
            D.zero_grad()

            #  1A: 실제 데이터로 D 학습
            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)))  #
            d_real_error.backward() # compute/store gradients, but don't change p

            #  1B: 가짜 데이터로 D 학습
            d_gen_input = Variable(gi_sampler(minibatch_size, g_input_size))
            d_fake_data = G(d_gen_input).detach()  # detach to avoid training G o
            d_fake_decision = D(preprocess(d_fake_data.t()))
            d_fake_error = criterion(d_fake_decision, Variable(torch.zeros(1)))
            d_fake_error.backward()
            d_optimizer.step()     # Only optimizes D's parameters; changes basec

        for g_index in range(g_steps):

            # 2. D의 반응에 따라 G학습

            # G 초기화
            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)))  # we

            g_error.backward()
            g_optimizer.step()  # Only optimizes G's parameters

        if (epoch+1) % (print_interval*10) == 0:
            print("%s: D: %.4f/%.4f G: %.4f (Real: %s, Fake: %s) \n" % (epoch+1,
                                                                        extract(d_real_er
                                                                        extract(d_fake_er
                                                                        extract(g_error)[
                                                                        stats(extract(d_r
                                                                        stats(extract(d_f
        if (epoch+1) % print_interval == 0:
            [mu_real, sigma_real] = stats(extract(d_real_data))
            [mu_fake, sigma_fake] = stats(extract(d_fake_data))

            x1 = np.linspace(mu_real-9*sigma_real,mu+9*sigma_real, 100)
            x2 = np.linspace(mu_fake-9*sigma_fake,mu+9*sigma_fake, 100)
            plt.plot(x1,mlab.normpdf(x1, mu_real, sigma_real))
            plt.plot(x2,mlab.normpdf(x2, mu_fake, sigma_fake))
            plt.title('Generate 1D Gaussian Distribution using GAN: %7d epoch'%(e
            plt.xlabel('Data values')
```

```
        plt.ylabel('Probability density')
        plt.savefig('img/Generate 1D Gaussian Distribution using GAN: %7d epc
        plt.show()
```

Visualization

Reference

[GAN keras code]
https://hussamhamdan.wordpress.com/2017/04/29/generative-adversarial-networks-keras-code/

[GAN tf code]
https://github.com/hwalsuklee/tensorflow-GAN-1d-gaussian-ex

[GAN pytorch code]
https://medium.com/@devnag/generative-adversarial-networks-gans-in-50-lines-of-code-pytorch-e81b79659e3f