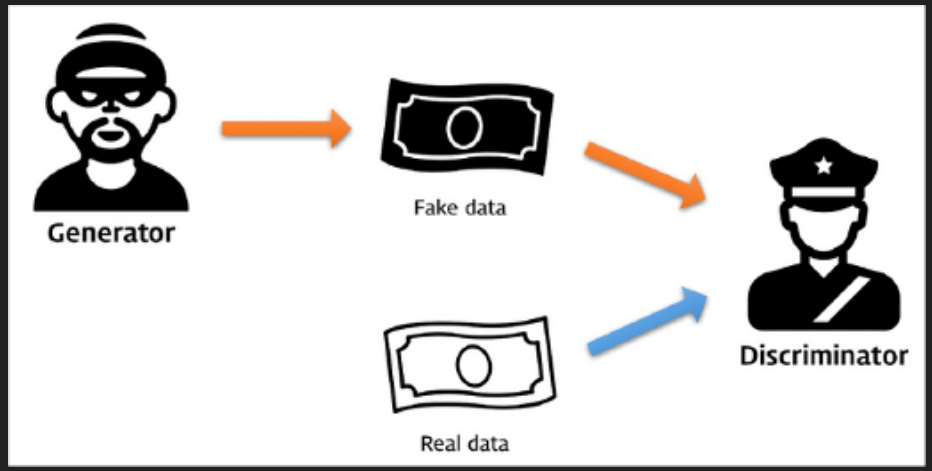
Generative Adversarial Networks

김연수

Overview

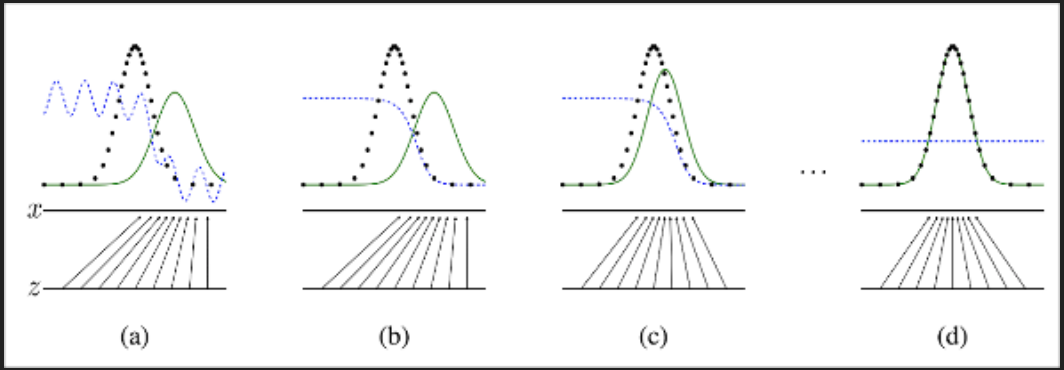


Minimax two-player game

Image를 만들어 내는 Generator(G)가 이 만들어진 모델을 평가하는 Discriminator(D)를 최대한 속일 수 있도록,
확률 분포의 차이를 줄이는 것이 목적

- 즉, G는 D를 최대한 속이려고 노력하고, D는 G가 만든 이미지를 최대한 감별하려고 노력함.
- 이 경쟁 속에서 두 모델은 모두 발전하게 되고, 결과적으로는 G가 만든 이미지를 구별할 수 없는 상태에 도달하게 됨.

Overview



위의 목표를 이루기 위해서는, *(ref. output -> [0,1] : 0==false, 1==true)*

- D : 진짜 이미지를 진짜 이미지라고 인식(분류)하도록 학습
- G : random한 코드를 받아서 img를 생성한 후, 그 이미지가 D를 속여야 함.
  - 즉, D(G(z)) = 1(진짜라 인식)이 나오도록 학습.
    - 학습할수록 진짜같은 가짜 img가 만들어지는 것

Introduction

Deep generative models have had less of an impact, due to the difficulty of approximating many intractable probabilistic computations that arise in maximum likelihood estimation and related strategies, and due to difficulty of leveraging the benefits of piecewise linear units in the generative context. We propose a new generative model estimation procedure that sidesteps these difficulties. 1

In the proposed adversarial nets framework, the generative model is pitted against an adversary: a discriminative model that learns to determine whether a sample is from the model distribution or the data distribution. The generative model can be thought of as analogous to a team of counterfeiters, trying to produce fake currency and use it without detection, while the discriminative model is analogous to the police, trying to detect the counterfeit currency. Competition in this game drives both teams to improve their methods until the counterfeits are indistiguishable from the genuine articles.

In this article, we explore the special case when the generative model generates samples by passing random noise through a multilayer perceptron, and the discriminative model is also a multilayer perceptron.

No approximate inference or Markov chains are necessary.

Adversarial nets

- Generator
  - generator's distribution p_g over data x 를 학습하기 위해서,
  - input noise variables p_z(z) 를 먼저 정의할 필요가 있고
  - data space로의 mapping을 G(z; θ_g)로 표현할 수 있다.
  - G는 미분 가능한 함수로, θ_g를 파라미터로 갖는 multilayer perceptron이다.
- Discriminator
  - D(x; θd) that outputs a single scalar (확률값이니까)
  - D(x) represents the probability that x came from the data rather than p_g.
- D to maximize the probability of assigning the correct label to both training examples and samples from G.
  We simultaneously train G to minimize log(1 − D(G(z)))

Adversarial nets

In other words, D and G play the following two-player minimax game with value function V (G, D)

D(x)=1일 때 최대

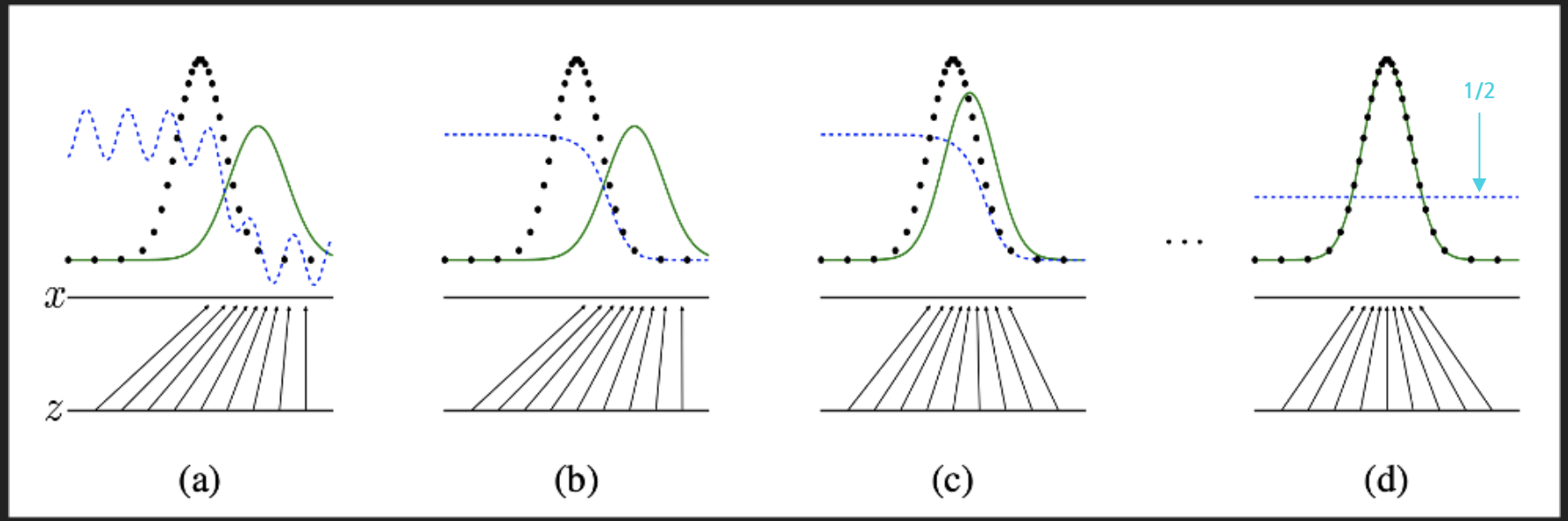
D(G(z))=0일 때 최대

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

D(G(z))=1일 때 최소

- D 입장(이상적인 Discriminator의 경우) 에서는 위 수식이 0인게 Maximize
  - D가 보는 sample x가 실제로 data (distribution)이라면 D(x)=1
    - 첫번째 term에서 log 값이 사라짐 => 0
  - G가 만든 data라면 D(G(z))=0
    - 두 번째 term도 0이 됨
  - 위 두 상황이 합쳐졌을 때 V의 최대값이 얻어지는 것
  - 즉 D의 입장에서는 위의 수식이 0이 되어야 Maximize하는 것으로 볼 수 있다.

Adversarial nets



Theoretical Results

1. This minimax game has a global optimum for p_g = p_data
2. 이 논문에서 제시하는 알고리즘이 global optimum을 갖는다.

Theoretical Results

1. This minimax game has a global optimum for p_g = p_data

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

- Proposition 1. For G fixed, *the optimal discriminator D* is

$$D^*_G(\boldsymbol{x}) = \frac{p_{data}(\boldsymbol{x})}{p_{data}(\boldsymbol{x}) + p_g(\boldsymbol{x})}$$

y → a log(y) + b log(1 − y) achieves its maximum in [0, 1] at a/(a+b)

  - Proof

$$\begin{aligned} V(G,D) &= \int_{\boldsymbol{x}} p_{\text{data}}(\boldsymbol{x}) \log(D(\boldsymbol{x}))dx + \int_z p_{\boldsymbol{z}}(\boldsymbol{z}) \log(1 - D(g(\boldsymbol{z})))dz \\ &= \int_{\boldsymbol{x}} p_{\text{data}}(\boldsymbol{x}) \log(D(\boldsymbol{x})) + p_g(\boldsymbol{x}) \log(1 - D(\boldsymbol{x}))dx \end{aligned}$$

Theoretical Results

1. This minimax game has a global optimum for p_g = p_data

    - Proof

$$
\begin{aligned}
C(G) &= \max_D V(G, D) \\
&= \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}} [\log D_G^*(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}} [\log(1 - D_G^*(G(\boldsymbol{z})))] \\
&= \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}} [\log D_G^*(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{x} \sim p_g} [\log(1 - D_G^*(\boldsymbol{x}))] \\
&= \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}} \left[\log \frac{p_{\text{data}}(\boldsymbol{x})}{P_{\text{data}}(\boldsymbol{x}) + p_g(\boldsymbol{x})} \right] + \mathbb{E}_{\boldsymbol{x} \sim p_g} \left[\log \frac{p_g(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_g(\boldsymbol{x})} \right]
\end{aligned}
\tag{4}
$$

Theoretical Results

1. This minimax game has a global optimum for p_g = p_data

- Theorem 1.
  The global minimum of the virtual training criterion C(G) is achieved
  if and only if p_g = p_data. At that point, C(G) achieves the value − log 4.

  ○ Proof. (다음 슬라이드)

$$= \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}} [\log D_G^*(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{x} \sim p_g} [\log(1 - D_G^*(\boldsymbol{x}))]$$

$$= \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}} \left[\log \frac{p_{\text{data}}(\boldsymbol{x})}{P_{\text{data}}(\boldsymbol{x}) + p_g(\boldsymbol{x})} \right] + \mathbb{E}_{\boldsymbol{x} \sim p_g} \left[\log \frac{p_g(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_g(\boldsymbol{x})} \right]$$

Proof. For $p_g = p_{\text{data}}$, $D_G^*(\boldsymbol{x}) = \frac{1}{2}$, (consider Eq. 2). Hence, by inspecting Eq. 4 at $D_G^*(\boldsymbol{x}) = \frac{1}{2}$, we find $C(G) = \log \frac{1}{2} + \log \frac{1}{2} = -\log 4$. To see that this is the best possible value of $C(G)$, reached only for $p_g = p_{\text{data}}$, observe that

$$\mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}} [-\log 2] + \mathbb{E}_{\boldsymbol{x} \sim p_g} [-\log 2] = -\log 4$$

and that by subtracting this expression from $C(G) = V(D_G^*, G)$, we obtain:

$$C(G) = -\log(4) + KL\left(p_{\text{data}} \,\middle\|\, \frac{p_{\text{data}} + p_g}{2}\right) + KL\left(p_g \,\middle\|\, \frac{p_{\text{data}} + p_g}{2}\right) \tag{5}$$

where KL is the Kullback–Leibler divergence. We recognize in the previous expression the Jensen–Shannon divergence between the model's distribution and the data generating process:

$$C(G) = -\log(4) + 2 \cdot JSD\left(p_{\text{data}} \,\|\, p_g\right) \tag{6}$$

$$D_{\mathrm{KL}}(P \| Q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} dx$$

$$\mathrm{JSD}(P \parallel Q) = \frac{1}{2} D_{KL}(P \parallel M) + \frac{1}{2} D_{KL}(Q \parallel M)$$

$$
\begin{aligned}
C(G) &= C(G) + log(4) - log(4) && (9)\\
&= -log(4) + KL\left(p_{data} || \frac{p_{data} + p_g}{2}\right) + KL\left(p_g || \frac{p_{data} + p_g}{2}\right) && (10)\\
&= -log(4) + 2 \cdot JSD(p_{data} || p_g) && (11)
\end{aligned}
$$

$$D_{\mathrm{KL}}(P\|Q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} dx$$

$$\mathrm{JSD}(P \parallel Q) = \frac{1}{2} D_{KL}(P \parallel M) + \frac{1}{2} D_{KL}(Q \parallel M)$$

Theoretical Results

2. 이 논문에서 제시하는 알고리즘이 global optimum을 갖는다.

- Proposition 2. If *G* and *D* have enough capacity, and at each step of Algorithm 1, the discriminator is allowed to reach its optimum given *G*, and p_g is updated so as to improve the criterion

$$\mathbb{E}_{\boldsymbol{x}\sim p_{data}}[\log D_G^*(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{x}\sim p_g}[\log(1 - D_G^*(\boldsymbol{x}))]$$

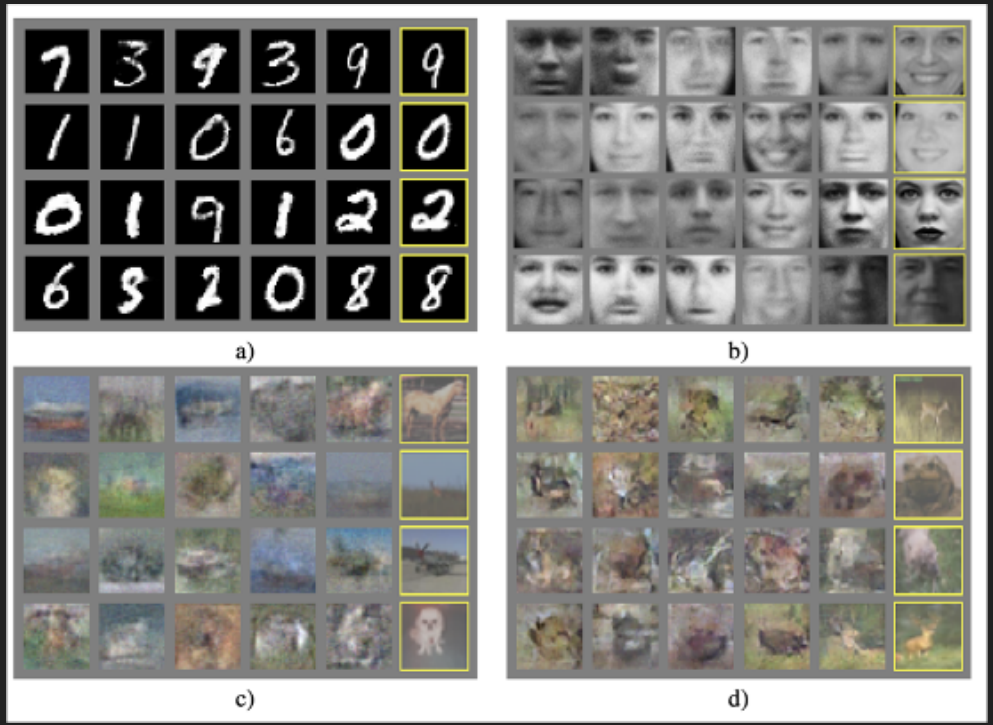
  then p_g converges to p_data

  - Proof. (?)
  - In practice, MLP => reasonable model

Experiments

1. MNIST
2. Toronto Face Database (TFD)
3. CIFAR-10

- *Samples are fair random draws, not cherry-picked.*



Advantages and Disadvantages

- Advantages
  - Markov chains are never needed
  - primarily computational
  - ...

- Disadvantages
  - no explicit representation of p_g(x)
  - D must be synchronized well with G during training

Code

```
# Discriminator
D = nn.Sequential(
    nn.Linear(image_size, hidden_size),
    nn.LeakyReLU(0.2),
    nn.Linear(hidden_size, hidden_size),
    nn.LeakyReLU(0.2),
    nn.Linear(hidden_size, 1),
    nn.Sigmoid())

# Generator
G = nn.Sequential(
    nn.Linear(latent_size, hidden_size),
    nn.ReLU(),
    nn.Linear(hidden_size, hidden_size),
    nn.ReLU(),
    nn.Linear(hidden_size, image_size),
    nn.Tanh())

# Device setting
D = D.to(device)
G = G.to(device)

# Binary cross entropy loss and optimizer
criterion = nn.BCELoss()
d_optimizer = torch.optim.Adam(D.parameters(), lr=0.0002)
g_optimizer = torch.optim.Adam(G.parameters(), lr=0.0002)
```

```
        # ================================================================== #
        #                        Train the generator                         #
        # ================================================================== #

        # Compute loss with fake images
        z = torch.randn(batch_size, latent_size).to(device)
        fake_images = G(z)
        outputs = D(fake_images)

        # We train G to maximize log(D(G(z)) instead of minimizing log(1-D(G(z)))
        # For the reason, see the last paragraph of section 3. https://arxiv.org/pdf/1406.2661.pdf
        g_loss = criterion(outputs, real_labels)

        # Backprop and optimize
        reset_grad()
        g_loss.backward()
        g_optimizer.step()

# Start training
total_step = len(data_loader)
for epoch in range(num_epochs):
    for i, (images, _) in enumerate(data_loader):
        images = images.reshape(batch_size, -1).to(device)

        # Create the labels which are later used as input for the BCE loss
        real_labels = torch.ones(batch_size, 1).to(device)
        fake_labels = torch.zeros(batch_size, 1).to(device)

        # ================================================================== #
        #                      Train the discriminator                       #
        # ================================================================== #

        # Compute BCE_Loss using real images where BCE_Loss(x, y): - y * log(D(x)) - (1-y) * log(1 - D(x))
        # Second term of the loss is always zero since real_labels == 1
        outputs = D(images)
        d_loss_real = criterion(outputs, real_labels)
        real_score = outputs

        # Compute BCELoss using fake images
        # First term of the loss is always zero since fake_labels == 0
        z = torch.randn(batch_size, latent_size).to(device)
        fake_images = G(z)
        outputs = D(fake_images)
        d_loss_fake = criterion(outputs, fake_labels)
        fake_score = outputs

        # Backprop and optimize
        d_loss = d_loss_real + d_loss_fake
        reset_grad()
        d_loss.backward()
        d_optimizer.step()
```

References

- https://www.youtube.com/watch?v=L3hz57whyNw&ab_channel=SungKim
- https://www.youtube.com/watch?v=odpjk7_tGY0&ab_channel=naverd2
- http://jaejunyoo.blogspot.com/2017/01/generative-adversarial-nets-1.html
- https://velog.io/@tobigs-gm1/basicofgan#loss
- https://hyunw.kim/blog/2017/10/27/KL_divergence.html
- https://github.com/yunjey/pytorch-tutorial/blob/master/tutorials/03-advanced/generative_adversarial_network/main.py