Generative Adversarial Nets

25.11.11. 202192022 여명구

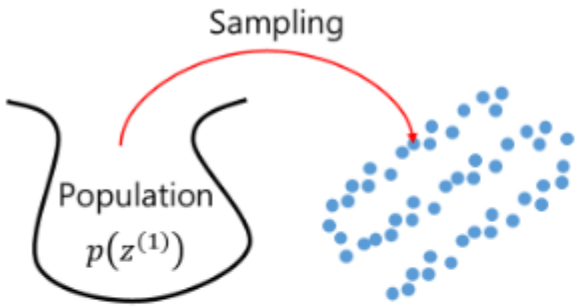
목차

- 생성모델이란?
- GAN의 등장 배경
- GAN의 아이디어
- GAN의 이론적 목표
- 학습 알고리즘
- 실제 구현(PyTorch, keras)
- (QR코드)

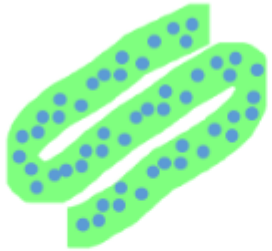
- 참고
  - Goodfellow et al., 2014. Generative Adversarial Nets.
  - 딥러닝 수업자료. 15_01 GAN 기초
  - wikipedia

생성모델

- 샘플링된(확보된) **유한**한 데이터를 이용하여 확률분포를 **추정** 후 추가 샘플링하여 데이터를 확보하기 위함.

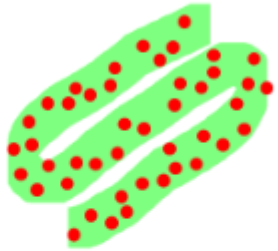


Input $z_i^{(1)}, i = 1, \dots, n$

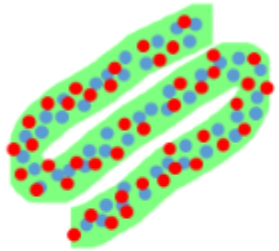


Sampling된 Data $z_i^{(1)}$의 분포
$p(z^{(1)})$

Data $z_i^{(1)}$를 이용하여 추정한 분포
$p_{model}(z^{(1)}; \hat{\theta})$



추정한 분포 $p_{model}(z^{(1)}; \hat{\theta})$로 부터 Sampling 된 Data
$z_{(g)i}^{(1)}, i = 1, \dots, N$



확보된 총 Data 수:
$n + N$

GAN의 배경

- GAN이 제안되기 전, 이전 생성모델들은 다루기 힘든 확률적 계산때문에 크게 발전하지 못함.
- **정규화 상수 Z**: RBM, DBM과 같은 당시 에너지 기반 모델은 데이터의 확률을 물리학의 에너지 개념을 가져와 정의함.
  - 이때 이 확률의 공리를 적용하려면 전체 확률의 합이 1이 되게하는 정규화 상수 Z가 필요함.
  - 이 Z 계산이 고차원에서 매우 다루기 힘든(intractable) 계산임.

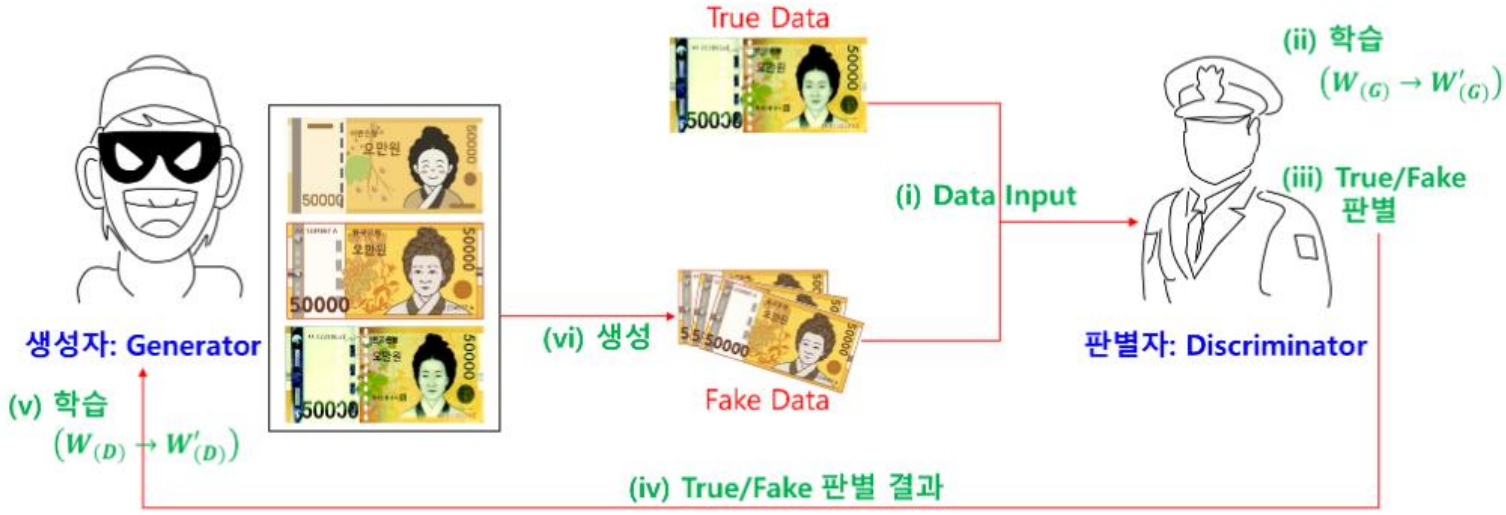
$$p_\theta(\mathbf{x}) = \frac{\exp(f_\theta(\mathbf{x}))}{Z(\theta)} = \frac{\exp(f_\theta(\mathbf{x}))}{\int \exp(f_\theta(\mathbf{x}))\mathrm{d}\mathbf{x}}$$

GAN의 배경

- **MCMC 의존성**: 다루기 힘든 확률적 계산을 피하기 위해 MCMC와 같은 복잡한 근사 기법으로 우회하려 했지만 MCMC는 샘플링 과정이 매우 느리고 분포의 다양한 mode를 제대로 탐색하지 못하는 mixing문제도 존재.
- **ReLU 활용의 어려움**: RBM이나 GSN같은 모델들은 샘플링 과정에서 피드백 루프를 사용하는데 이때 ReLU를 사용하면 활성화 값이 발산하기 쉬웠음.

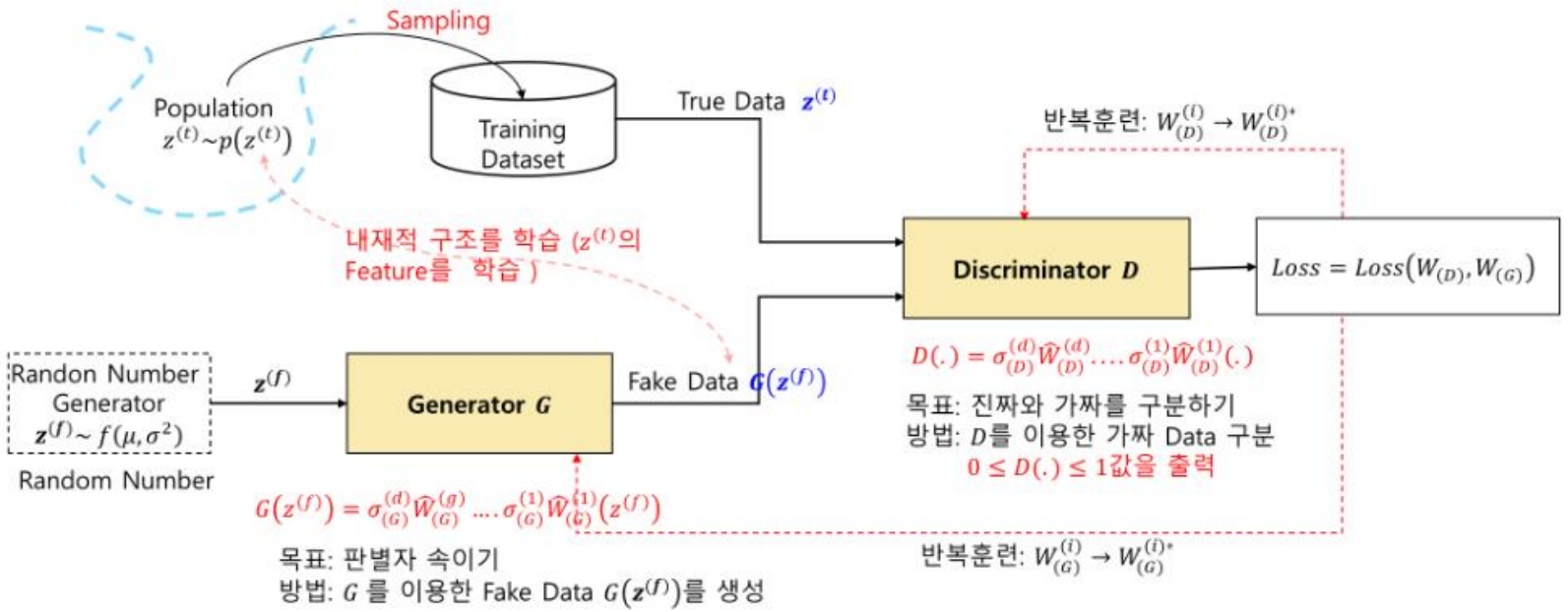
GAN의 아이디어

- 두 개의 신경망이 서로 경쟁하는 **적대적 과정**으로 재정의.
- **생성자(G, Generator)**: 무작위 노이즈를 입력받아 실제 데이터와 유사한 가짜 데이터를 생성하는 것이 목표.
  - G는 확률분포를 직접 다루지 않고 D의 출력값에 따라 학습함.
  - 기존의 변분추론과 다른 접근 방식
- **판별자(D, Discriminator)**: 입력데이터가 true인지 G가 만든 fake인지 구별하는 것이 목표.(입력 데이터가 진짜일 확률(0~1)을 출력



GAN의 아이디어

- D의 목표: true에는 1, fake(G(z))에는 0을 출력하도록 학습
- G의 목표: 자신이 만든 fake를 D가 진짜라고 착각하도록 학습
- 이 경쟁은 G가 만든 fake와 true와 구별 불가능하게 될 때까지 계속되며, 이 과정에서 G와 D는 **함께 발전**함.



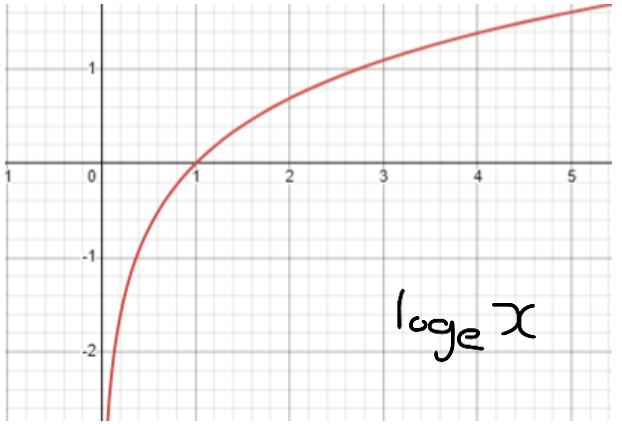
Notations	설명
$z^{(t)}$	• Discriminator Network D에 Input되는 True Input vector
$z^{(f)}$	• Random vector로 Generator Network G에 Input되어 Fake Data $G(z^{(f)})$를 생성하는 Randon Seed
G	• GAN의 Generator Network $G(z^{(f)}) = \sigma_{(G)}^{(g)} \widehat{W}_{(G)}^{(g)} \cdots \sigma_{(G)}^{(1)} \widehat{W}_{(G)}^{(1)}(z^{(f)})$
$G(z^{(f)})$	• Generator Network G을 이용하여 생성된 Fake Data • Discriminator Network D에 Input됨
D	• GAN의 Discriminator Network $D(x) = \sigma_{(D)}^{(d)} \widehat{W}_{(D)}^{(d)} \cdots \sigma_{(D)}^{(1)} \widehat{W}_{(D)}^{(1)}(x)$ Perfect Discriminator: $D(x) = \begin{cases} 1 & if\ x = z^{(t)} \\ 0 & if\ x = G(z^{(f)}) \end{cases}$

GAN의 이론적 목표

- 이 경쟁 과정을 minimax 2인용 게임이라는 수학적 틀로 정의.
- 이 게임의 유일한 균형점(**내시 균형**)은 생성자가 실제 데이터 분포를 완벽히 복제한 상태임.
- D는 이 가치함수를 최대화, G는 최소화하려함.
- 기댓값을 적분형태로 바꿔 D(x)에 대한 편미분을 수행하여 가치 함수를 특정 값으로 만드는 D를 찾을 수 있음.

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

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



GAN의 이론적 목표(내시균형)

- **내시 균형**(Nash equilibrium)은 게임이론에서 경쟁자 대응에 따라 최선의 선택을 하면 서로가 자신의 선택을 바꾸지 않는 균형상태를 말한다.
- Value function V(G,D)를 D로 편미분 유도
  - y=a/a+b일때 V가 전역 최댓값을 가짐.
  - a=b라면 y(D(x))=0.5

$p_{data}(x)\log D(x) + p_g(x)\log(1-D(x))$

$p_{data}(x) = a,\ p_g(x) = b,\ D(x) = y,\ (a, b > 0)$

$f(y) = a\log y + b\log(1-y)$

$\frac{\partial f(y)}{\partial y} = \frac{a}{y} - \frac{b}{1-y} = 0$ 인 지점에서 최적해를 가짐.

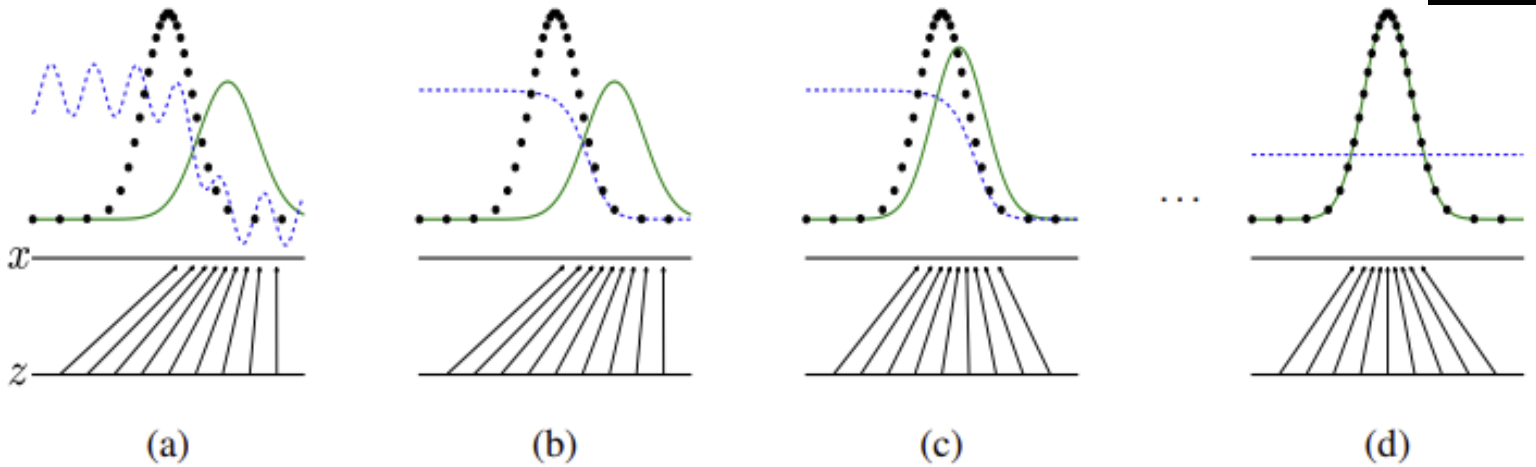
$\frac{a}{y} = \frac{b}{1-y} \Rightarrow a(1-y) = by \Rightarrow a - ay = by$

$\Rightarrow a = ay + by \Rightarrow y = \frac{a}{a+b}$.

$f''(y)$를 유도하여 볼록성을 확인하면 $a, b > 0$ 이므로

$f''(y)$는 항상 음수를 가지므로 $f'(y) = 0$ 인 지점인 $y = \frac{a}{a+b}$ 에서

global maximum을 가짐.



GAN의 이론적 목표(내시균형)

- G의 관점에서 V를 C(G)로 정의했을 때 이론적으로 G가 true 분포(p_data(x))를 완벽히 복제하여 p_g(x) = p_data(x)를 달성했다면 C(G)가 전역 최적해(minimum) –log4에 수렴함.

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

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

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

Algorithm

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 $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

- 보통 k=1로 설정
- 먼저 판별자가 진짜와 가짜를 잘 구별하게 학습(maximize V)
- 생성자는 고정된 판별자를 속이는 방향으로 업데이트(minimize V)

실제 구현(PyTorch)

```python
class MLPGenerator(nn.Module):
    def __init__(self, z_dim=Z_DIM, out_dim=IMG_DIM, hidden_dim=HIDDEN_DIM):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(z_dim, hidden_dim), nn.ReLU(inplace=True),
            nn.Linear(hidden_dim, hidden_dim), nn.ReLU(inplace=True),
            # G의 출력 범위를 mnist정규화와 맞추기 위해 Tanh 사용
            nn.Linear(hidden_dim, out_dim), nn.Tanh()
        )
    def forward(self, z):
        return self.model(z)
```

```python
Z_DIM = 100
HIDDEN_DIM = 256
IMG_SIZE = 28
IMG_CH = 1
IMG_DIM = IMG_SIZE * IMG_SIZE * IMG_CH
LR = 2e-4

BATCH_SIZE = 512
EPOCHS = 100
```

```python
class MLPDiscriminator(nn.Module):
    def __init__(self, in_dim=IMG_DIM, hidden_dim=HIDDEN_DIM):
        super().__init__()
        self.model = nn.Sequential(
            # 논문에서는 maxout을 사용.
            nn.Linear(in_dim, hidden_dim), nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(hidden_dim, hidden_dim), nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(hidden_dim, 1)  # logits 출력 -> 이후 BCEWithLogitsLoss 사용
        )
    def forward(self, x):
        return self.model(x)
```

실제 구현(PyTorch)

```
real_labels = torch.ones(b, 1, device=device) # true
fake_labels = torch.zeros(b, 1, device=device) # fak

# 1) Discriminator
opt_D.zero_grad(set_to_none=True)

d_real_logits = discriminator(real_flat)
loss_d_real = bce(d_real_logits, real_labels) # max log(D(x))

z = torch.randn(b, Z_DIM, device=device) # noise sampling
# .detach()로 D를 학습시킬 때 G의 그래디언트가 계산되지 않도록 구현.
fake_flat = generator(z).detach()
d_fake_logits = discriminator(fake_flat)
loss_d_fake = bce(d_fake_logits, fake_labels) # max log(1 - D(G(z)))

loss_d = loss_d_real + loss_d_fake
loss_d.backward() # gradient 계산
opt_D.step() # 파라미터 업데이트

# 2) Generator (non-saturating: target=real)
opt_G.zero_grad(set_to_none=True)

z = torch.randn(b, Z_DIM, device=device) # 새로운 noise sampling
fake_flat = generator(z)
d_fake_logits_g = discriminator(fake_flat)
# G 학습 시 fake에 대한 label로 1을 사용(min log(1 - D(G(z))) 대신 max log(D(G(z))) )
loss_g = bce(d_fake_logits_g, real_labels) # max log(D(G(z)))

loss_g.backward() # gradient 계산
opt_G.step() # 파라미터 업데이트
```

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

학습 초기 : 강한 D, 약한 G

$D(G(z)) \simeq 0$, $G(z) = \hat{x}$, $D(x) = \frac{1}{1+e^{-x}}$

G의 그래디언트

$$\frac{\partial L}{\partial D(z)} = \frac{\partial \log(1-D(z))}{\partial(1-D(z))} \cdot \frac{\partial(1-D(z))}{\partial D(z)} \cdot \frac{\partial D(z)}{\partial z}$$

$$= \frac{1}{1-D(z)} \cdot (-1) \cdot D(z)(1-D(z))$$

$$= -D(z)$$

학습 초기 $D(z) \simeq 0$, G에게 전달되는 그래디언트가 0으로 소실

$\min \log(1-D(z)) \rightarrow \max \log D(z)$

$$\frac{\partial L}{\partial D(z)} = \frac{\partial \log D(z)}{\partial D(z)} \cdot \frac{\partial D(z)}{\partial z} = \frac{1}{D(z)} \cdot D(z)(1-D(z))$$

$= 1-D(z)$, $D(z) \simeq 0$, 그래디언트 크기 $= 1$

Pytorch, keras 구현 비교(1)

- **데이터 형상(shape)**
- Pytorch - 학습 루프에서 (N, C, H, W) 형태로 입력받음.

```
fixed_z = torch.randn(64, Z_DIM, 1, 1, device=device)
```

- Keras – 모델 정의할 때 (N, H, W, C) 형태로 입력받음.

```
model.add(layers.Input(shape=(z_dim,)))
model.add(layers.Reshape((1, 1, z_dim)))
```

Pytorch, keras 구현 비교(2)

- **패딩(padding) 정의 방식**
- pytorch – CNN레이어를 정의할 때 패딩 수치를 직접 계산해서 입력

```
nn.ConvTranspose2d(z_dim, ngf * 4, kernel_size=7, stride=1, padding=0, bias=False),
```

- Keras – ‘same’으로 프레임워크가 알아서 입/출력 크기를 자동 계산해줌.

```
model.add(layers.Conv2DTranspose(128, 4, 2, 'same', use_bias=False))
```

Pytorch, keras 구현 비교(3)

- **미분 및 그래디언트 관리**
- Pytorch – backward 호출 즉시 역전파 수행

```
# D step (판별자 훈련)
opt_D.zero_grad(set_to_none=True)

# 1. 진짜 이미지로 학습 (target=1)
d_real = D(real)
loss_d_real = criterion(d_real, real_labels)

# 2. 가짜 이미지로 학습 (target=0)
z = torch.randn(b, Z_DIM, 1, 1, device=device)
fake = G(z).detach() # G 그래디언트 차단
d_fake = D(fake)
loss_d_fake = criterion(d_fake, fake_labels)

loss_d = loss_d_real + loss_d_fake
loss_d.backward()
opt_D.step()
```

- Keras – 미분에 필요한 연산을 기록(tf.GradientTape)한 뒤 gradien로 일괄 계산 후 반환(호출할 때마다 새로운 미분값 반환(초기화 필요 X)

```
# D Step
with tf.GradientTape() as tape:
    generated_images = self.generator(random_latent_vectors, training=True)
    real_logits = self.discriminator(real_images, training=True)
    fake_logits = self.discriminator(generated_images, training=True)

    real_labels = tf.ones((batch_size, 1))
    fake_labels = tf.zeros((batch_size, 1))

    d_loss_real = self.loss_fn(real_labels, real_logits)
    d_loss_fake = self.loss_fn(fake_labels, fake_logits)
    d_loss = d_loss_real + d_loss_fake

    if isinstance(self.d_optimizer, mixed_precision.LossScaleOptimizer):
        scaled_d_loss = self.d_optimizer.get_scaled_loss(d_loss)
    else:
        scaled_d_loss = d_loss

grads = tape.gradient(scaled_d_loss, self.discriminator.trainable_variables)
if isinstance(self.d_optimizer, mixed_precision.LossScaleOptimizer):
    grads = self.d_optimizer.get_unscaled_gradients(grads)
self.d_optimizer.apply_gradients(zip(grads, self.discriminator.trainable_variables))
```

Pytorch, keras 구현 비교(4)

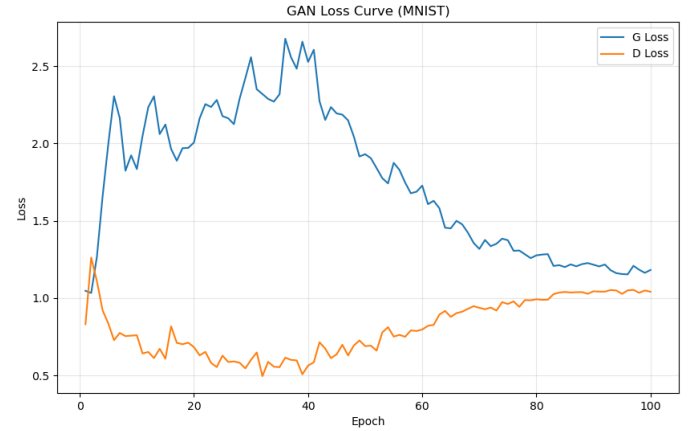
- **그래디언트 차단**
- Pytorch - 그래디언트가 끊어진(detached) 텐서를 D에 입력

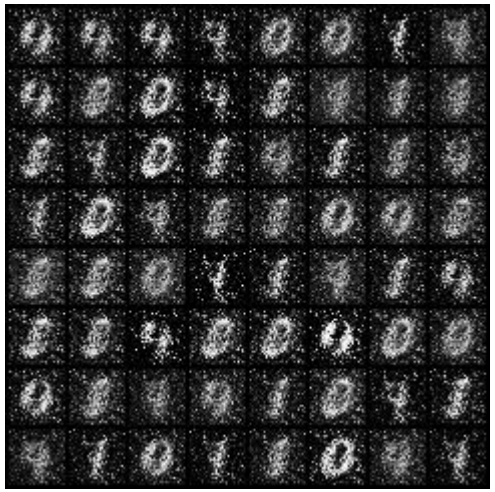
```
# 2. 가짜 이미지로 학습 (target=0)
z = torch.randn(b, Z_DIM, 1, 1, device=device)
fake = G(z).detach() # G 그래디언트 차단
d_fake = D(fake)
loss_d_fake = criterion(d_fake, fake_labels)
```

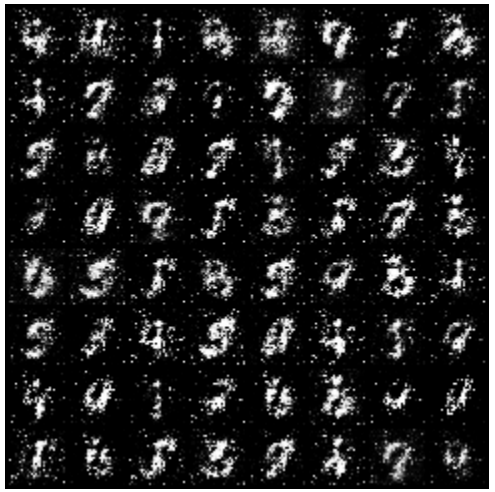
- Keras – 미분할 변수 목록에 D의 변수만 전달함으로써 G를 자연스럽게 배제

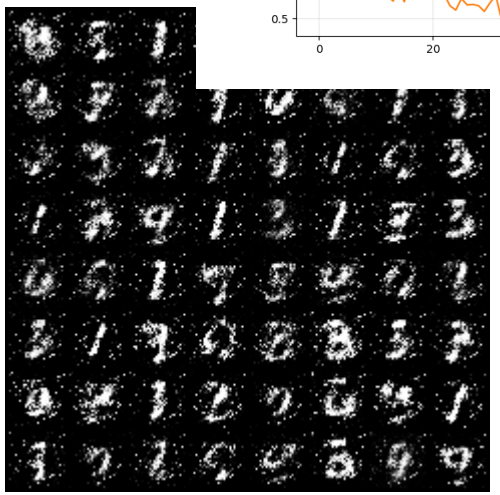
```
grads = tape.gradient(scaled_d_loss, self.discriminator.trainable_variables)
```

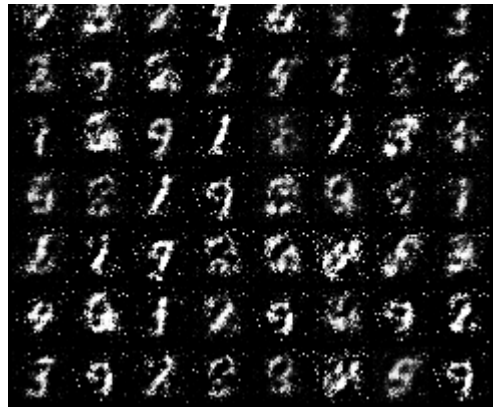
실제 구현(PyTorch, MLP)



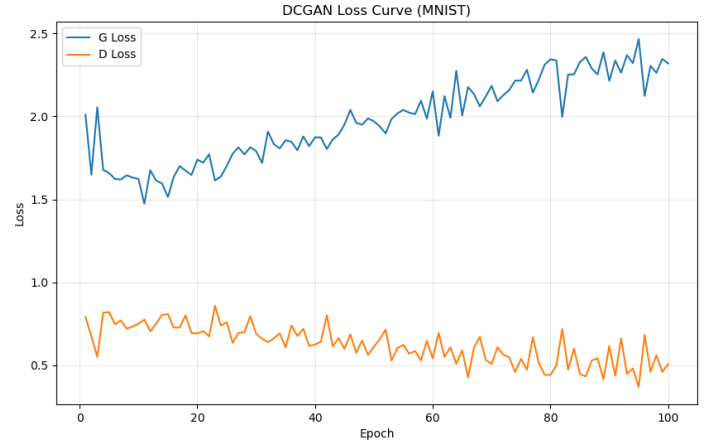


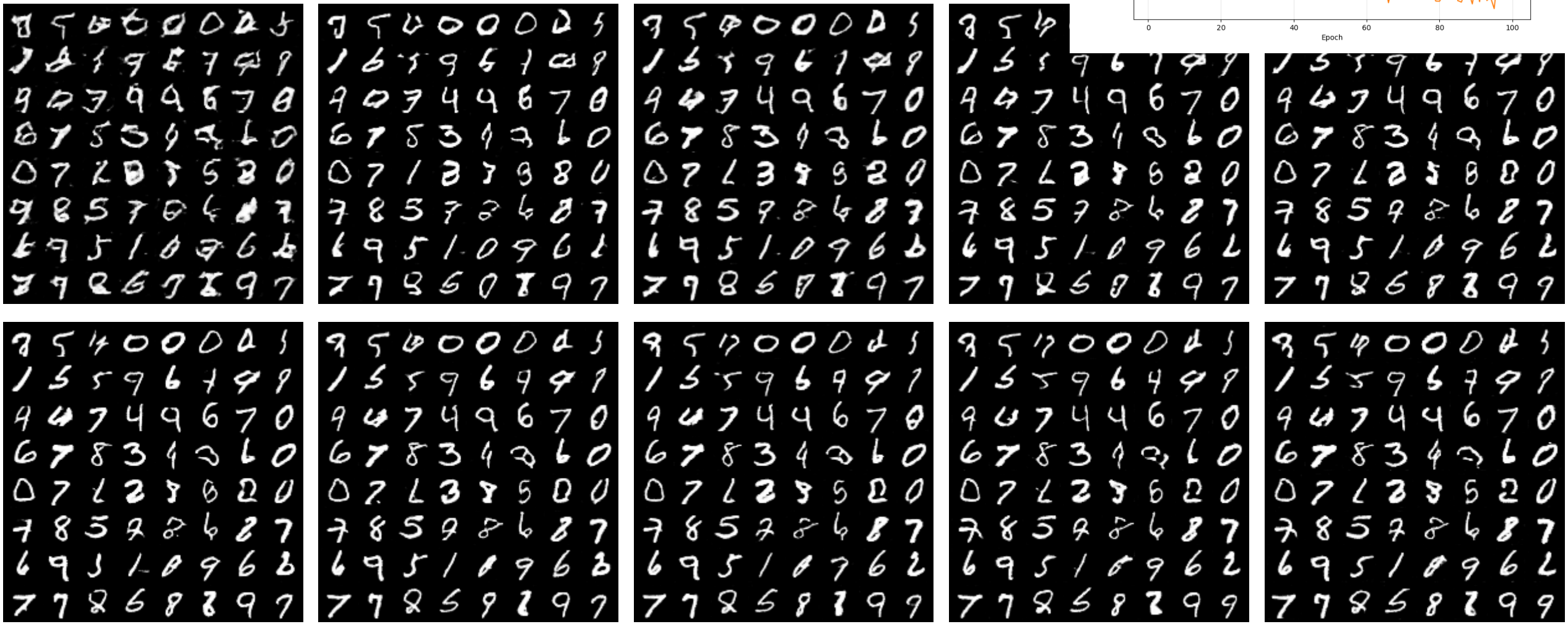




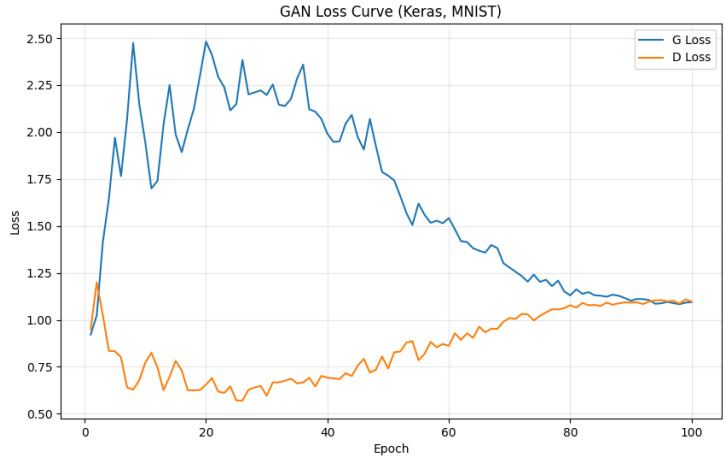


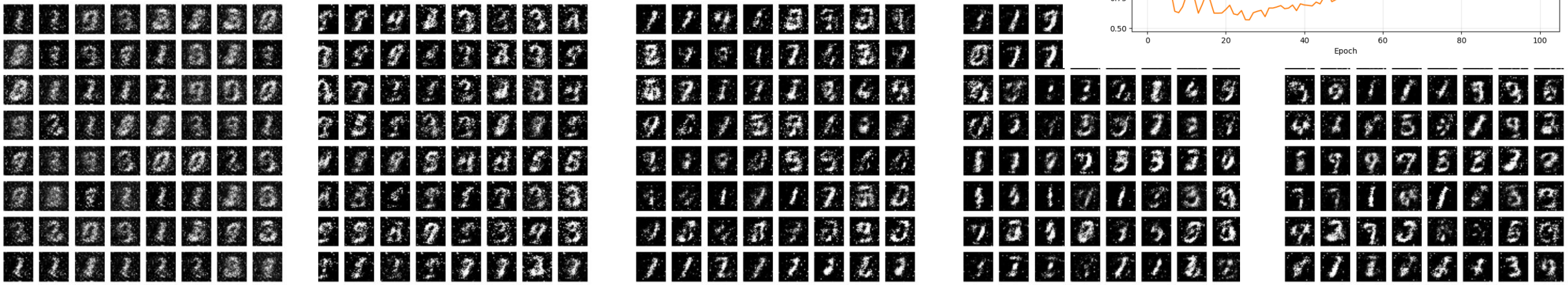
실제 구현(PyTorch, DCGAN)

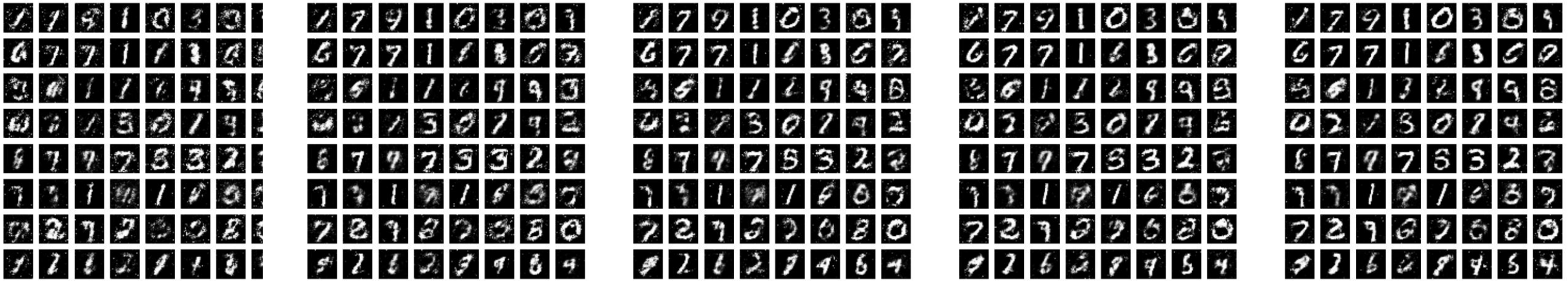




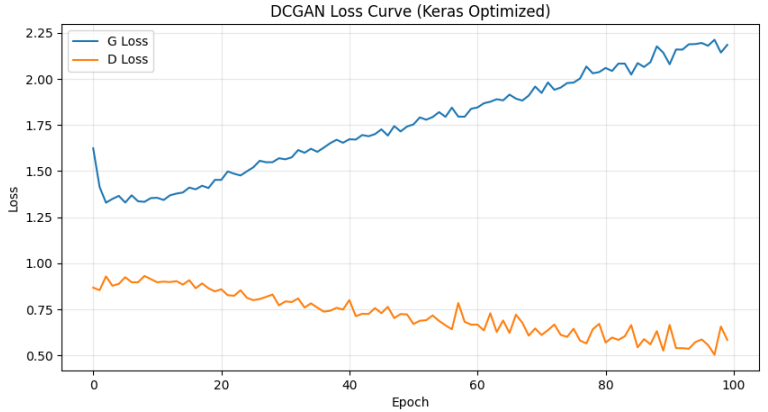
실제 구현(keras, MLP)

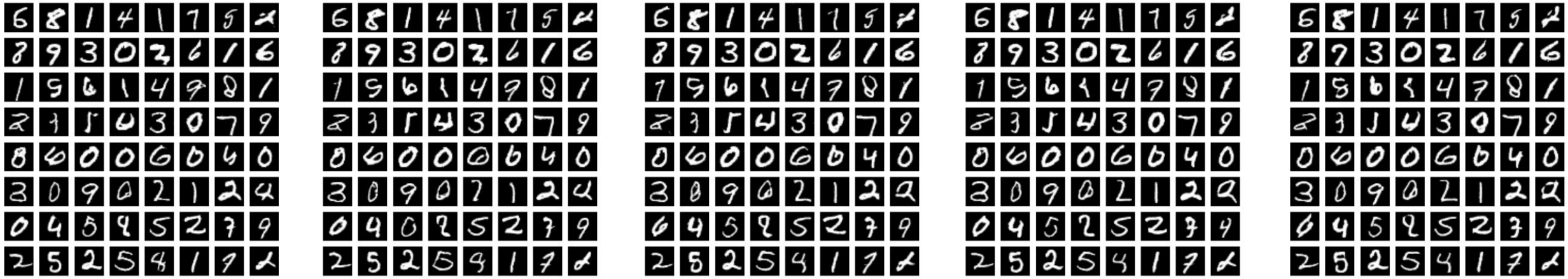






실제 구현(keras, DCGAN)





결론

- **MLP 기반 GAN**:
  - 28x28 2D이미지를 1D벡터로 펼치며 이웃 픽셀간 관계, 선, 모서리 등 공간적/구조적 정보가 파괴됨. D의 손실이 증가하고 G의 손실이 감소하는 것은 G가 손쉽게 D를 속임.
- **DCGAN**(deep convolution):
  - 여러 2D 국소적 특징을 보고 층이 깊어짐에 따라 이들을 조합해 전역적 특징을 학습함.
- GAN의 학습은 판별자가 충분히 강해야 생성자한테 의미있는 신호(gradient)를 전달하여 더 나은 샘플을 생성하도록 유도할 수 있음.