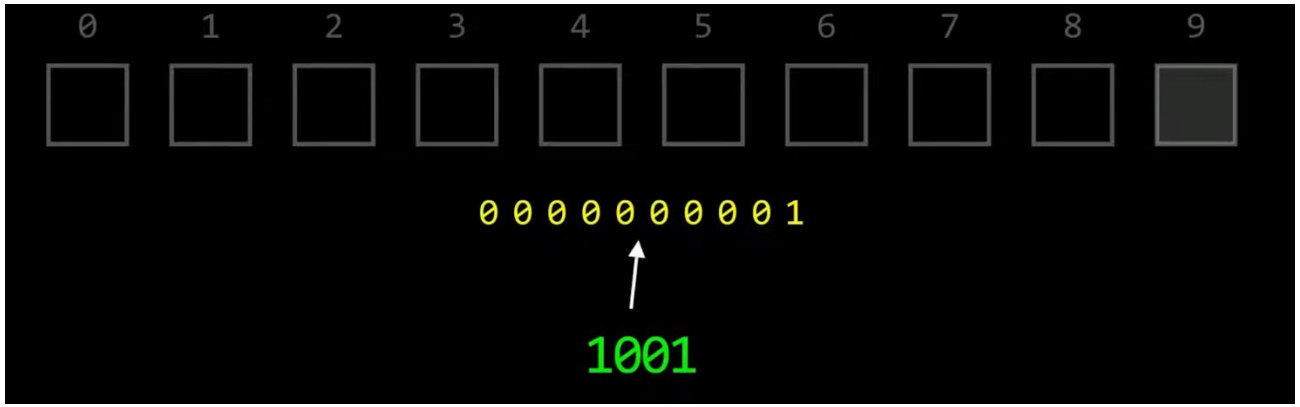
AAE

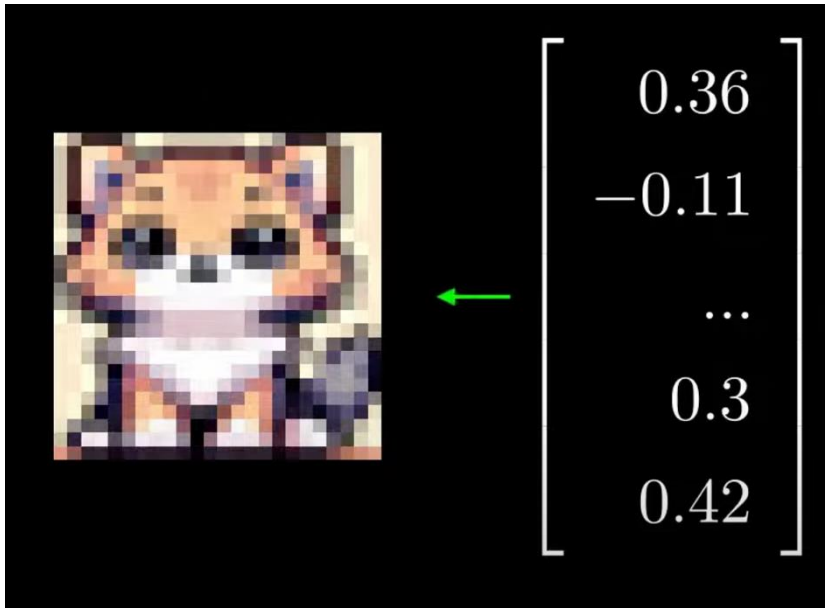
Adversarial Auto Encoder

Encoding

숫자 인코딩



이미지 인코딩

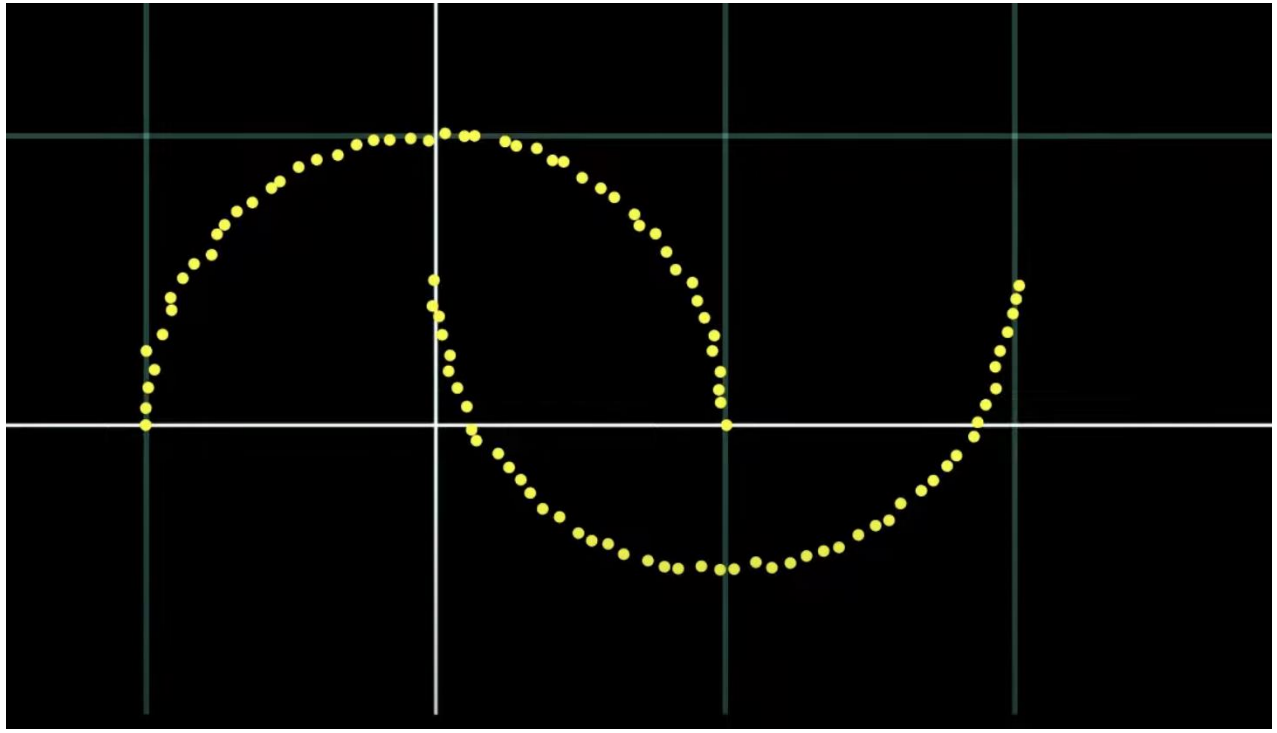


인코딩은 특정 가정을 통해서 정보를 더 적은 차원으로 나타내는 것을 의미한다.

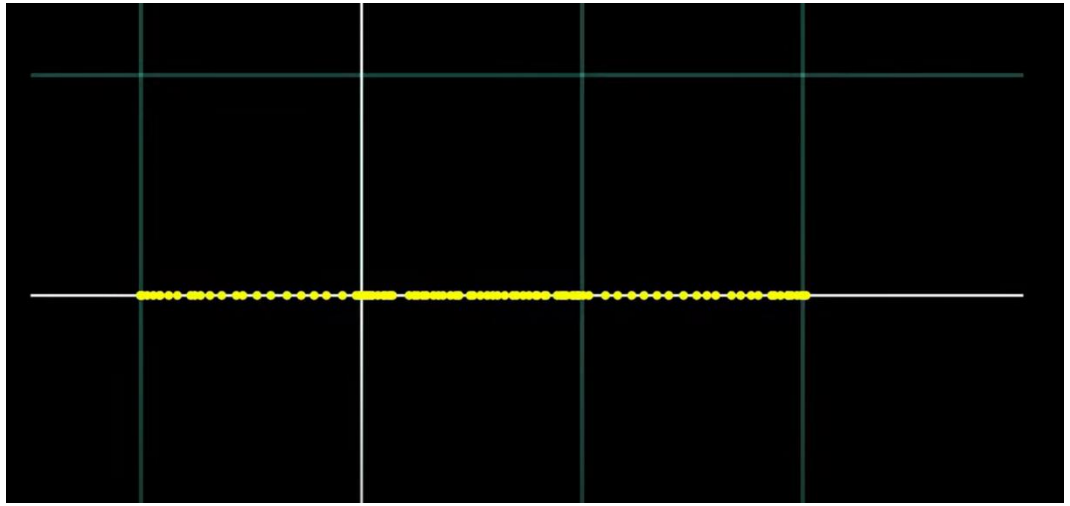
그리고 이를 복원하는 디코딩 또한 가능해야 합니다.

Encoding

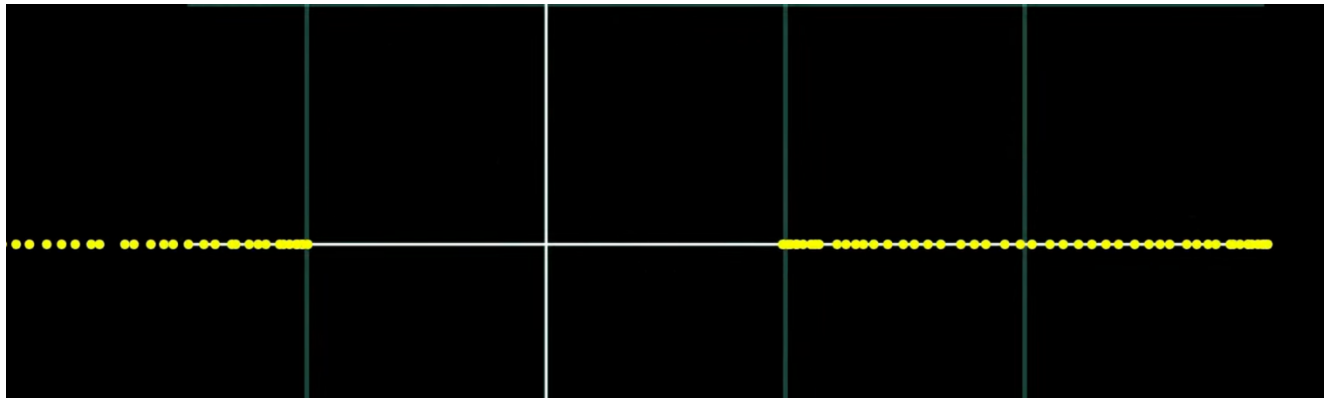
```
<0.29, -0.19>  <0.03, 1.01>  <0.01, 0.38>  <0.70, 0.69>  <1.22, -0.48>
<1.95, 0.26>  <0.96, 0.26>  <-0.83, 0.56>  <-0.27, 0.97>  <-1.00, 0.26>
<0.84, -0.50>  <1.99, 0.39>  <1.80, -0.11>  <0.98, 0.08>  <1.72, -0.19>
<0.37, -0.29>  <0.98, -0.50>  <0.69, 0.73>  <1.34, -0.43>  <1.52, -0.35>
<0.43, -0.32>  <2.02, 0.48>  <0.44, 0.91>  <0.60, -0.41>  <-0.94, 0.31>
<0.49, -0.37>  <0.12, -0.02>  <-0.63, 0.77>  <0.51, 0.86>  <0.62, 0.79>
<0.73, -0.47>  <1.90, 0.07>  <1.93, 0.20>  <-0.21, 0.99>  <0.92, -0.49>
<0.11, 0.06>  <-0.69, 0.74>  <0.65, -0.45>  <1.93, 0.13>  <0.24, 0.98>
<-0.01, 0.41>  <1.46, -0.38>  <-0.77, 0.59>  <0.40, 0.92>  <1.87, 0.01>
<0.57, 0.82>  <-0.91, 0.40>  <1.77, -0.14>  <-0.57, 0.82>  <0.35, 0.96>
<-0.16, 0.99>  <-0.41, 0.92>  <1.00, 0.00>  <1.68, -0.23>  <0.96, 0.31>
<1.03, -0.50>  <0.80, 0.60>  <1.61, -0.27>  <-0.54, 0.84>  <1.28, -0.45>
<0.14, -0.05>  <1.39, -0.42>  <-0.87, 0.51>  <0.93, 0.37>  <-0.34, 0.93>
<0.07, 0.13>  <-1.00, 0.00>  <0.98, 0.12>  <0.90, 0.43>  <2.00, 0.44>
<0.13, 1.00>  <0.28, 0.97>  <1.11, -0.47>  <0.89, 0.49>  <0.25, -0.15>
<0.83, 0.54>  <-0.92, 0.44>  <-0.99, 0.13>  <-0.09, 0.99>  <-1.00, 0.06>
<0.33, -0.23>  <0.54, -0.40>  <1.16, -0.49>  <1.57, -0.33>  <0.02, 0.31>
<1.86, -0.04>  <0.05, 0.24>  <-0.47, 0.89>  <0.04, 0.19>  <-0.02, 0.98>
<0.98, 0.18>  <-0.76, 0.66>  <-0.73, 0.69>  <0.10, 1.00>  <-0.01, 0.50>
<0.22, -0.10>  <0.76, 0.66>  <-0.97, 0.19>  <1.97, 0.32>  <0.79, -0.49>
```



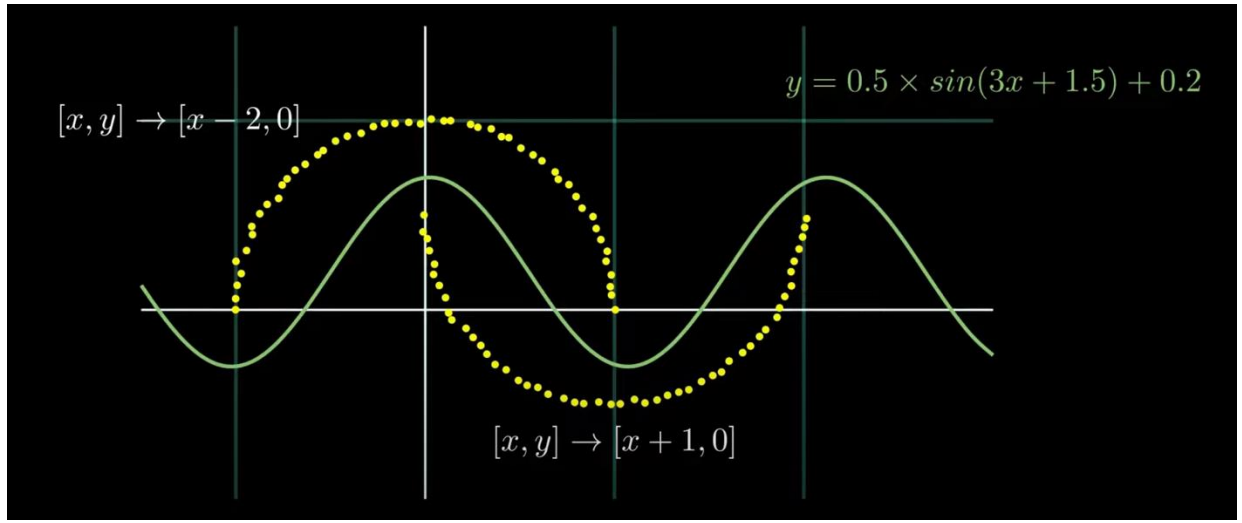
단순히 x 값만 사용하는 경우



곡선 위, 아래를 참고하여 분리

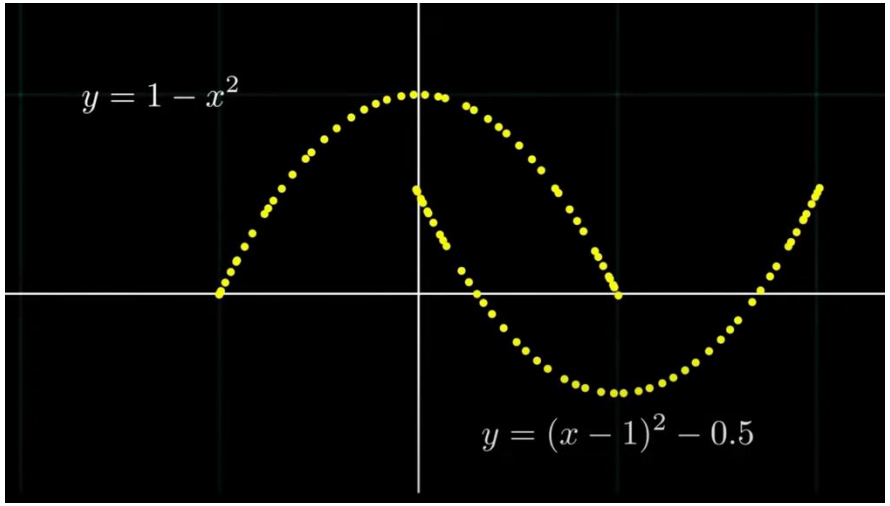


이렇게 함수의 값만 사용해서 위의 복잡한 숫자 정보의 규칙(가정)을 찾음



Encoding

다시 복원시 원래 함수가 뭔지 모르니깐 일단은 대충 2차 함수 형식으로 되돌린다



즉, 지금까지의 과정을 살펴보면 복잡한 숫자 데이터들을 하나의 수식으로 표현을 했고, 이를 통해서 1차원으로 데이터들을 Encoding 하였습니다.
그리고 Encoding된 데이터들을 사용해서 특정 2차 함수를 사용해서 원본으로 되돌리는 Decoding 작업을 수행하였습니다. 하지만 이러한 과정들은 우리가 데이터를 직접 보고 인코딩 식과 디코딩 식을 직접 모델링하여 대입하는 수동 인코딩 방식입니다.

우리가 사용한 가정

Encoding : 특정 sin 함수의 위,아래로 데이터가 나뉜다.
Decoding : 데이터가 2차함수를 따를 것이다.

** 수동으로 하면 복잡하고, 자원이 만이 들어가고 정확하지 않음 & 가정이 틀린 경우 제대로 디코딩이 되지 않는다

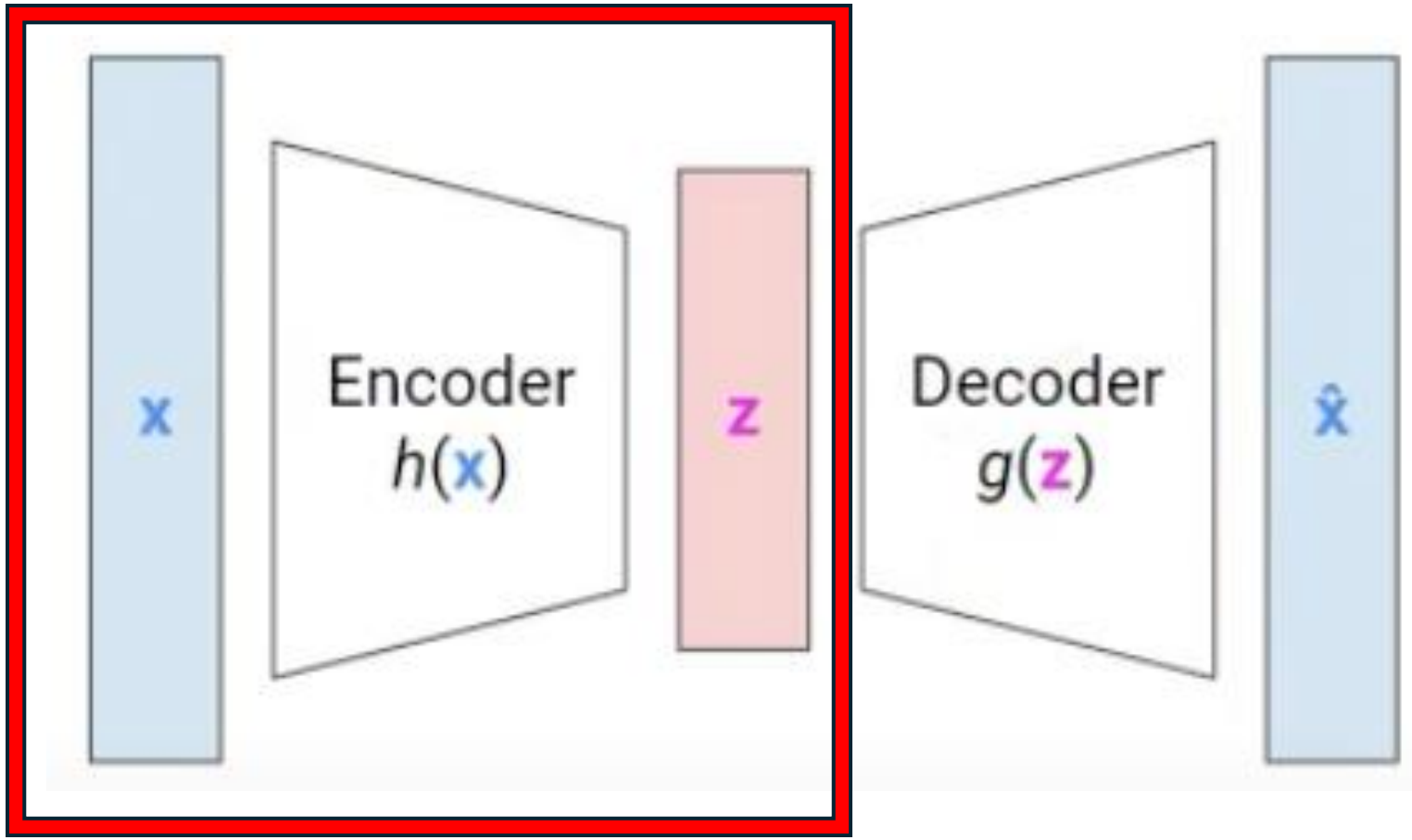
Auto Encoder

지금까지 수동을 했던 인코딩을 자동으로 해준다.

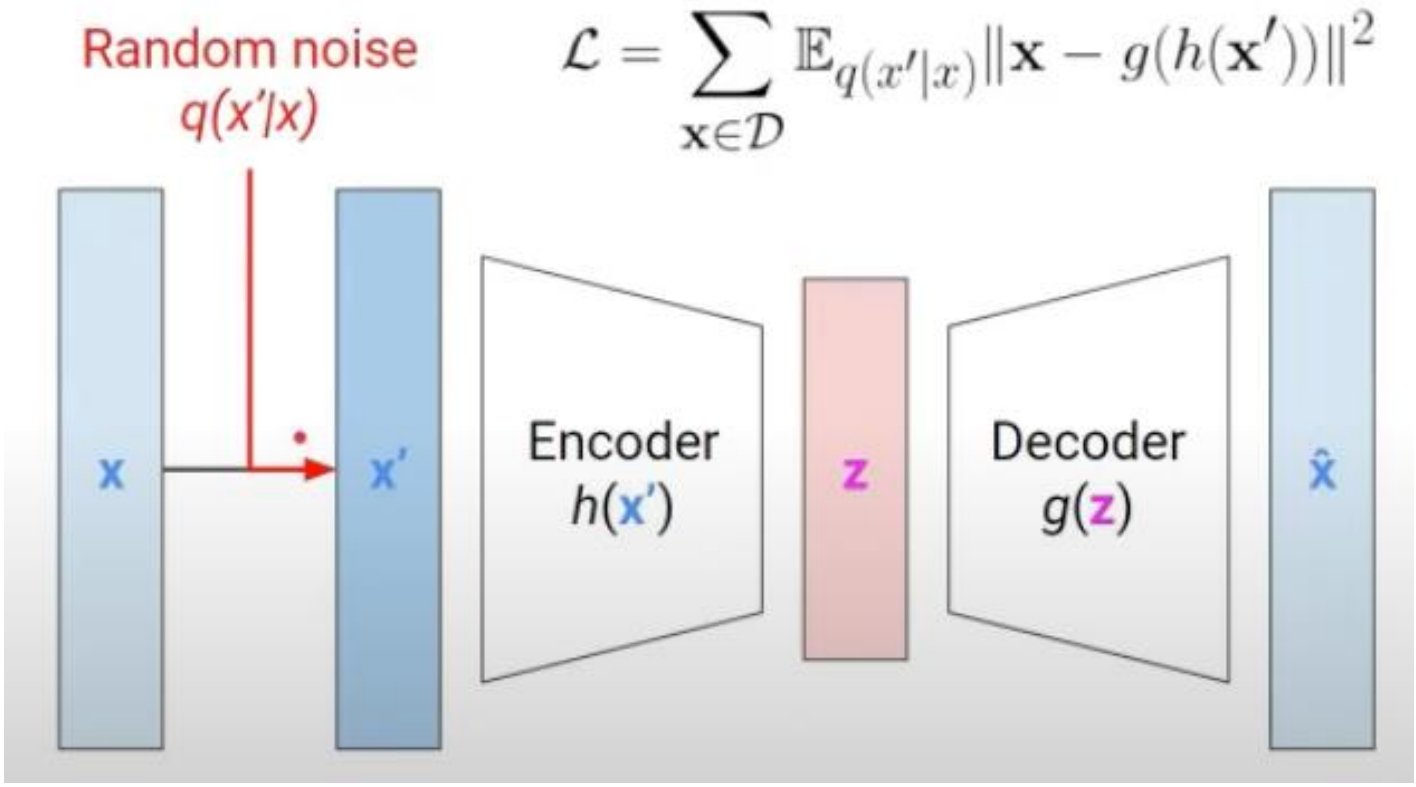
Auto Encoder

Auto Encoder의 목표는 Representation Learning

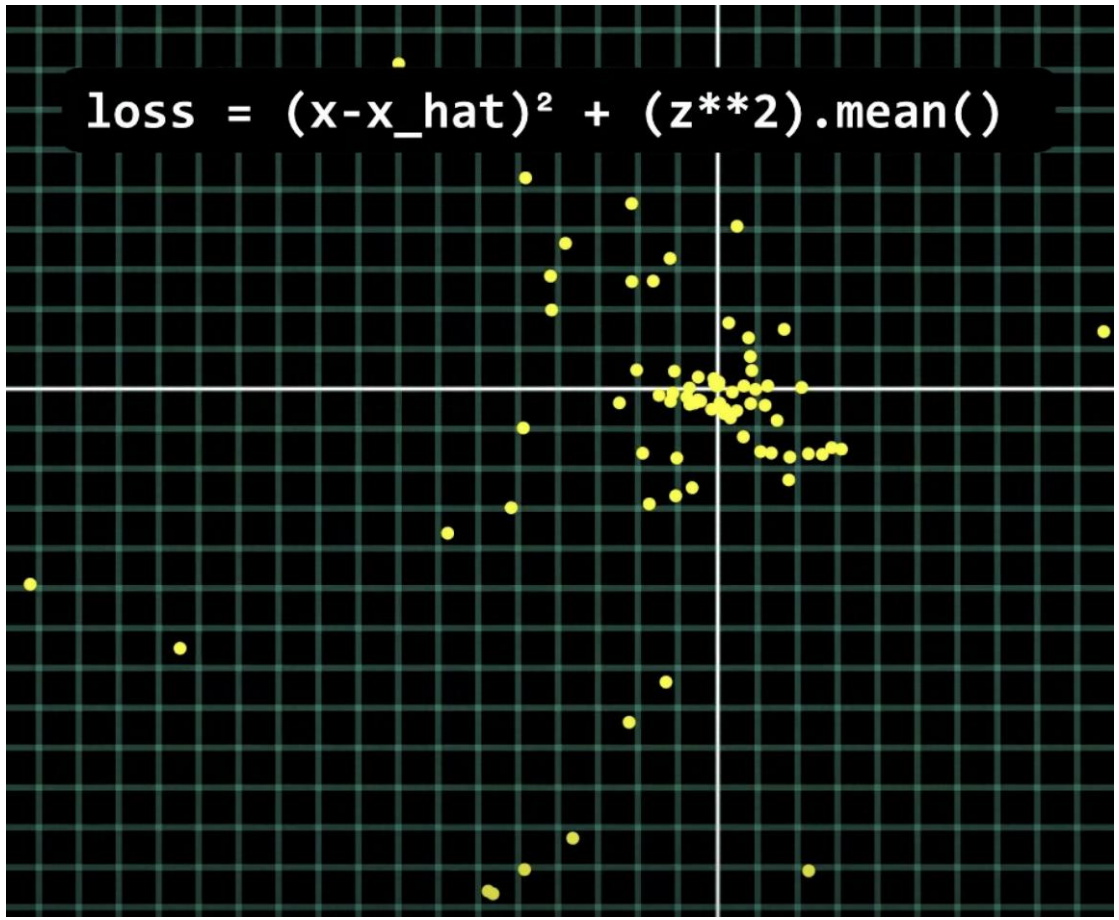
$$Loss = \sum ||(x - g(h(x))||^2$$

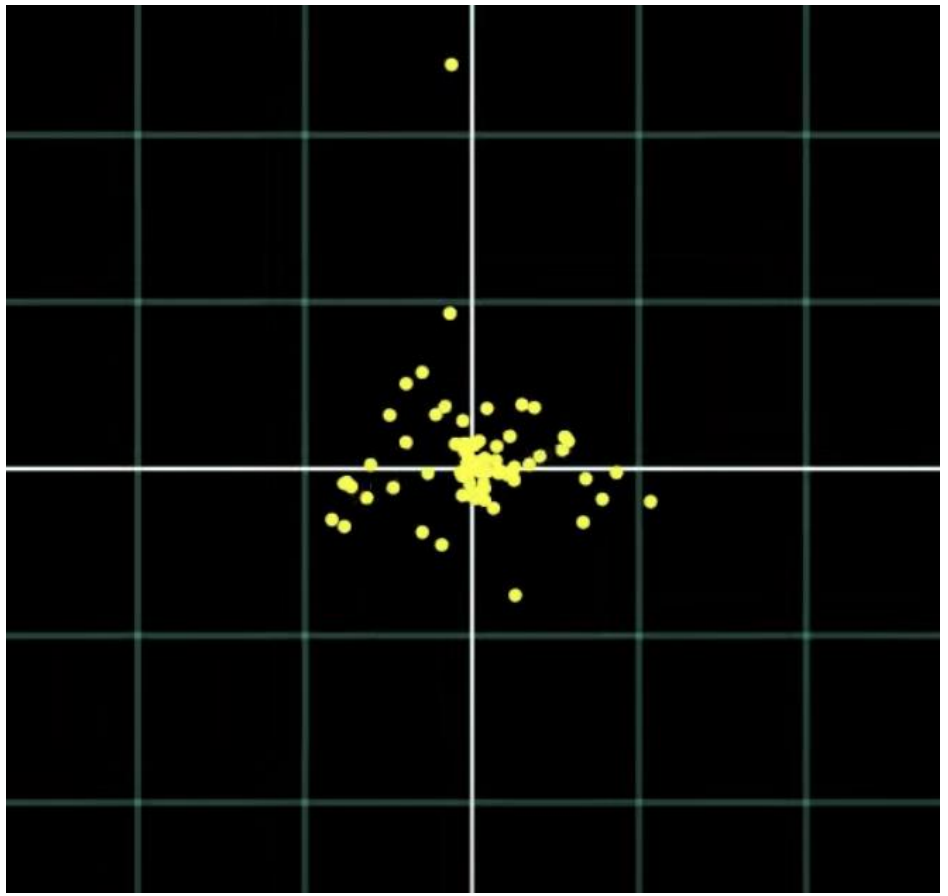


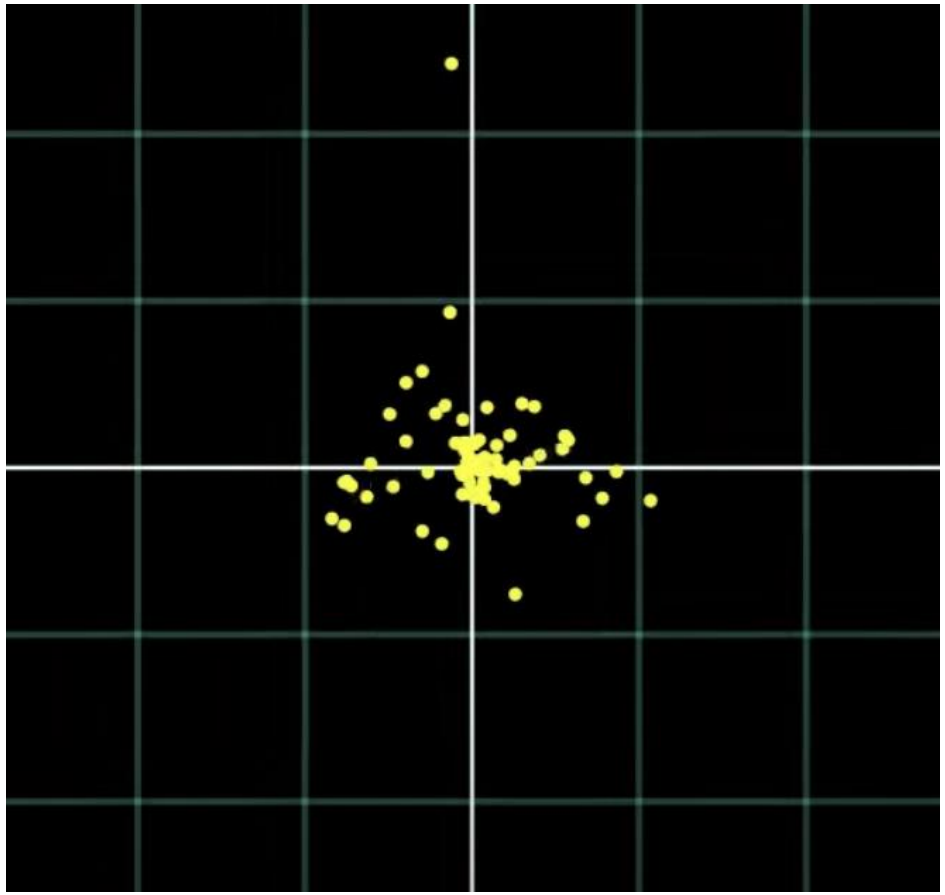
Denoising Auto Encoder

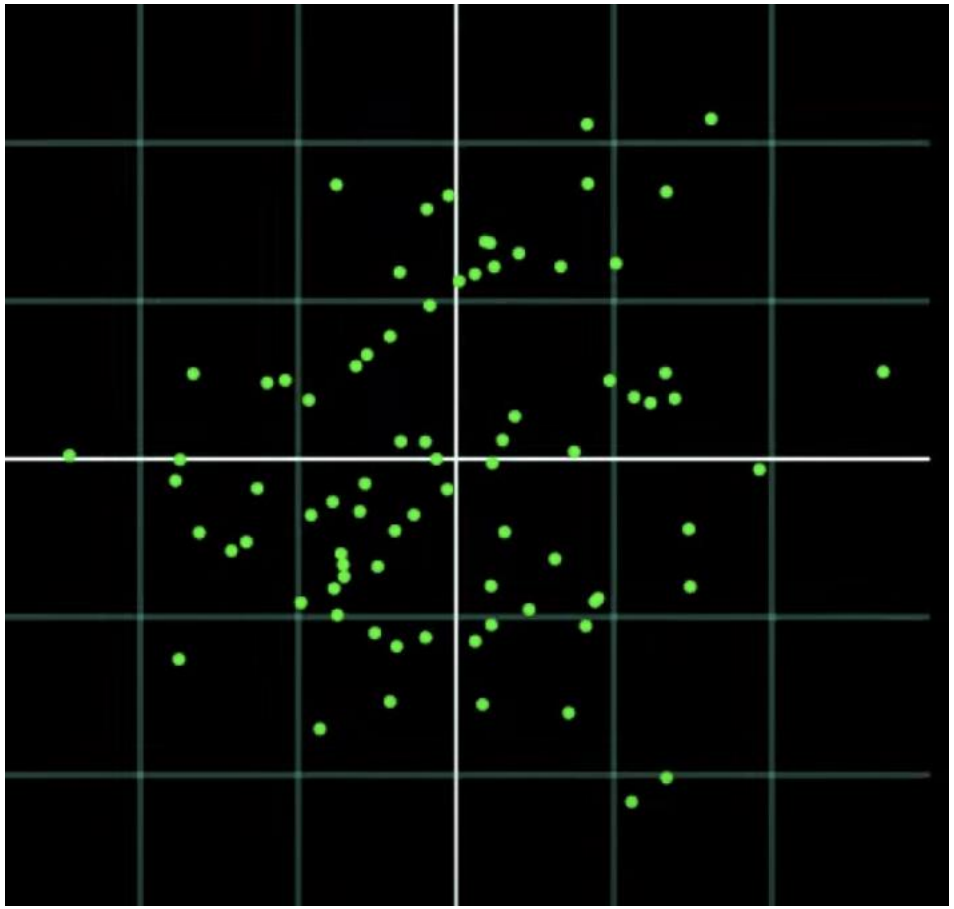


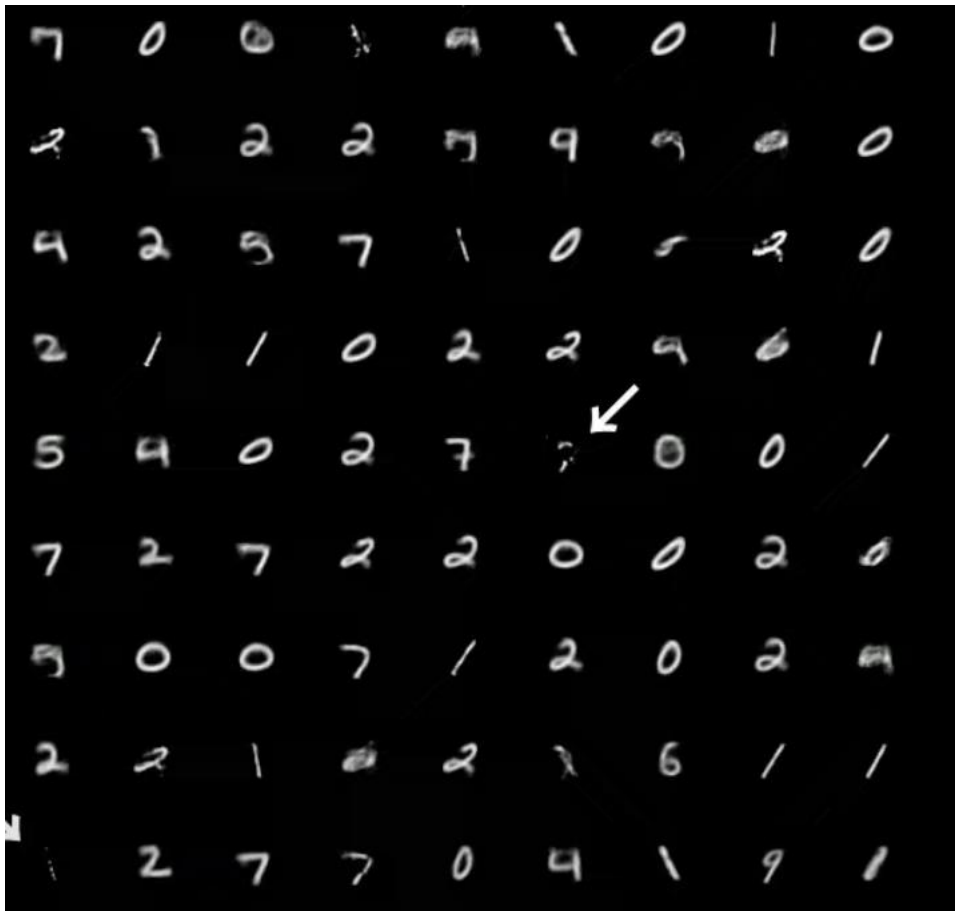
Variational Auto Encoder

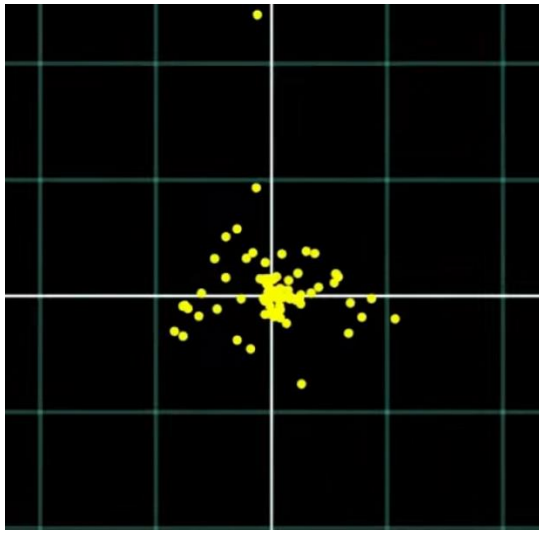

loss = (x-x_hat)² + (z**2).mean()

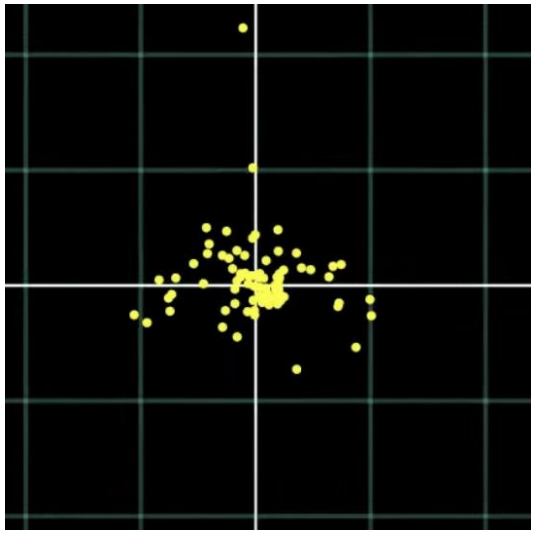




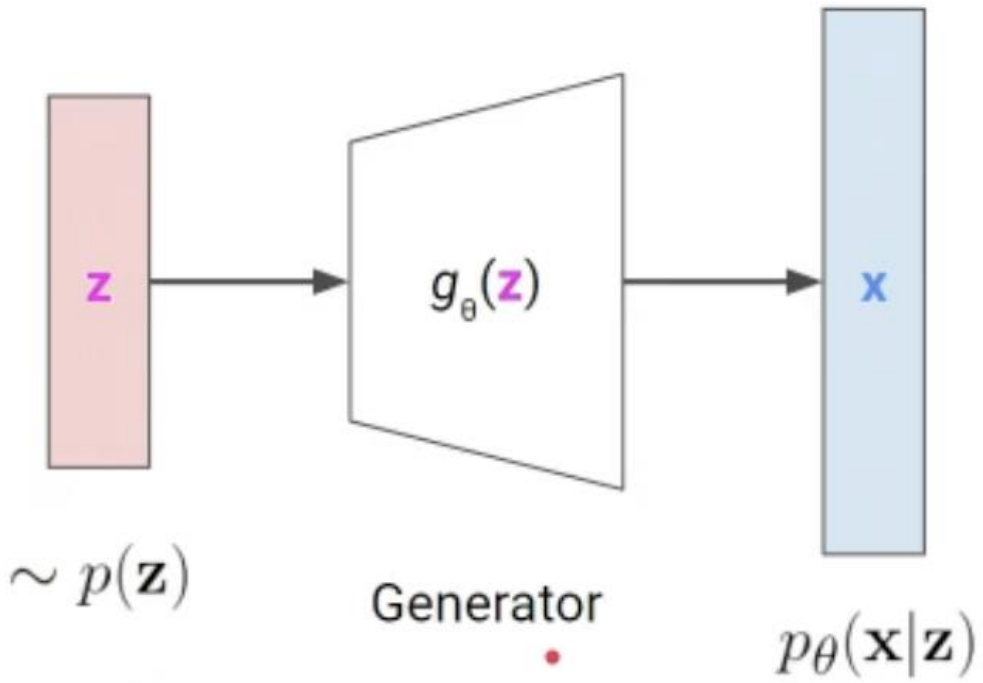


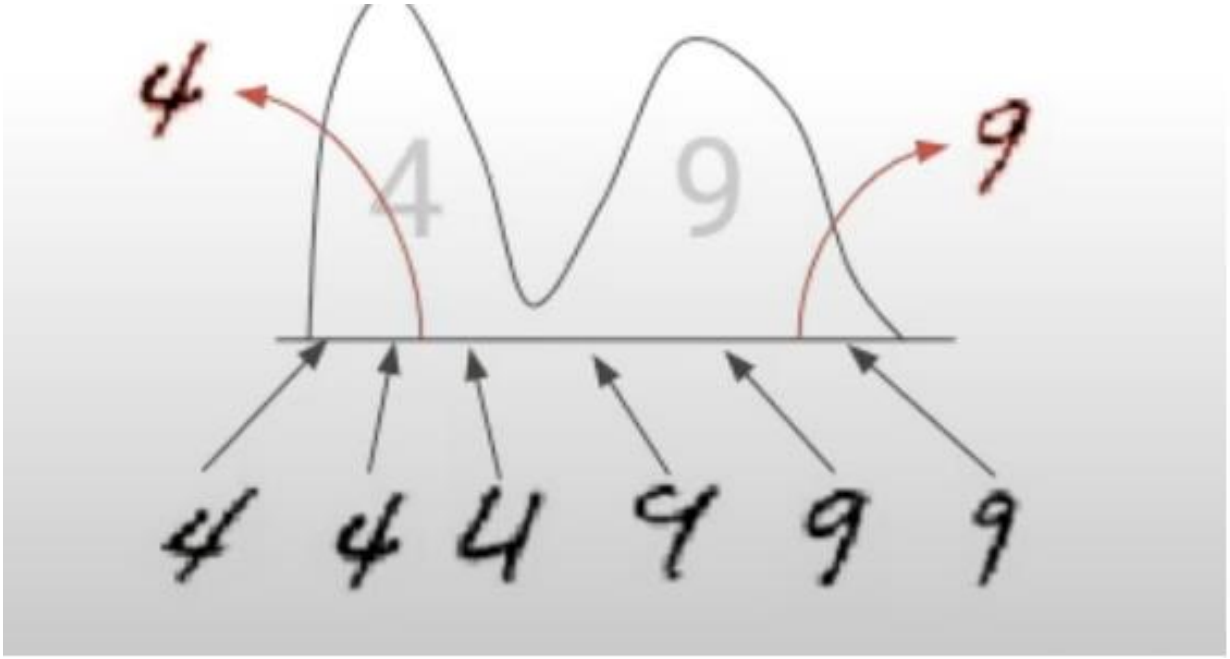




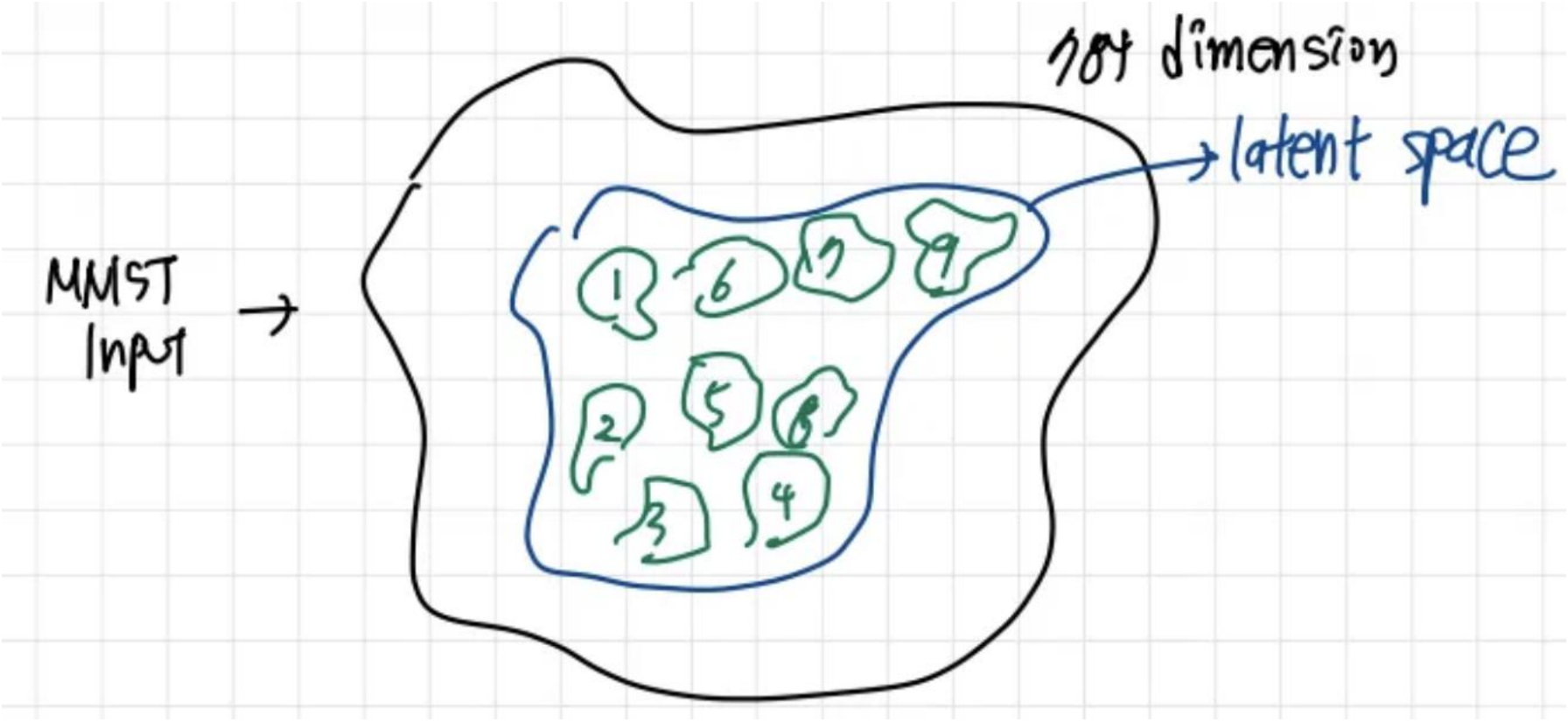


VAE (Variational Auto Encoder)





Manifold assumption



D차원의 모든 차원이 데이터와 관련이 있지 않다. 데이터는 마치 cluster처럼 특정 차원에만 존재한다.
이러한 가정을 통해서 우리는 데이터를 나타내는 차원을 줄여도 충분히 데이터를 표현 가능하다.

VAE (Variational Auto Encoder)

$$p(\mathbf{x}) = \int p(\mathbf{x}|g_\theta(\mathbf{z}))p(\mathbf{z})dz$$

$$p(\mathbf{x}) \approx \sum_i p(\mathbf{x}_i|g_\theta(\mathbf{z}_i))p(\mathbf{z}_i) \approx \mathbb{E}_{p(\mathbf{z})}\left[p(\mathbf{x}|g_\theta(z))\right]$$

$$L(\theta) = \prod_{i=1}^{N} p(\mathbf{x}_i|g_\theta(\mathbf{z}_i)) \qquad \ell(\theta) = \log \prod_{i=1}^{N} p(\mathbf{x}_i|g_\theta(\mathbf{z}_i)) = \sum_{i=1}^{N} \log p(\mathbf{x}_i|g_\theta(\mathbf{z}_i))$$

VAE (Variational Auto Encoder)

가정 : p(x|z) 확률 분포가 정규분포를 따른다

$$p(\mathbf{x} \mid g_\theta(\mathbf{z})) \sim N(\mathbf{x} \mid g_\theta(\mathbf{z}), \sigma^2 \mathbf{I})$$

$$\ell(\theta) = \sum_{i=1}^{N} \log p(\mathbf{x}_i | g_\theta(\mathbf{z}_i)) = \sum_{i=1}^{N} \log \frac{1}{Z} \exp\left\{ -\frac{\|\mathbf{x}_i - g_\theta(\mathbf{z}_i)\|^2}{\sigma^2} \right\}$$

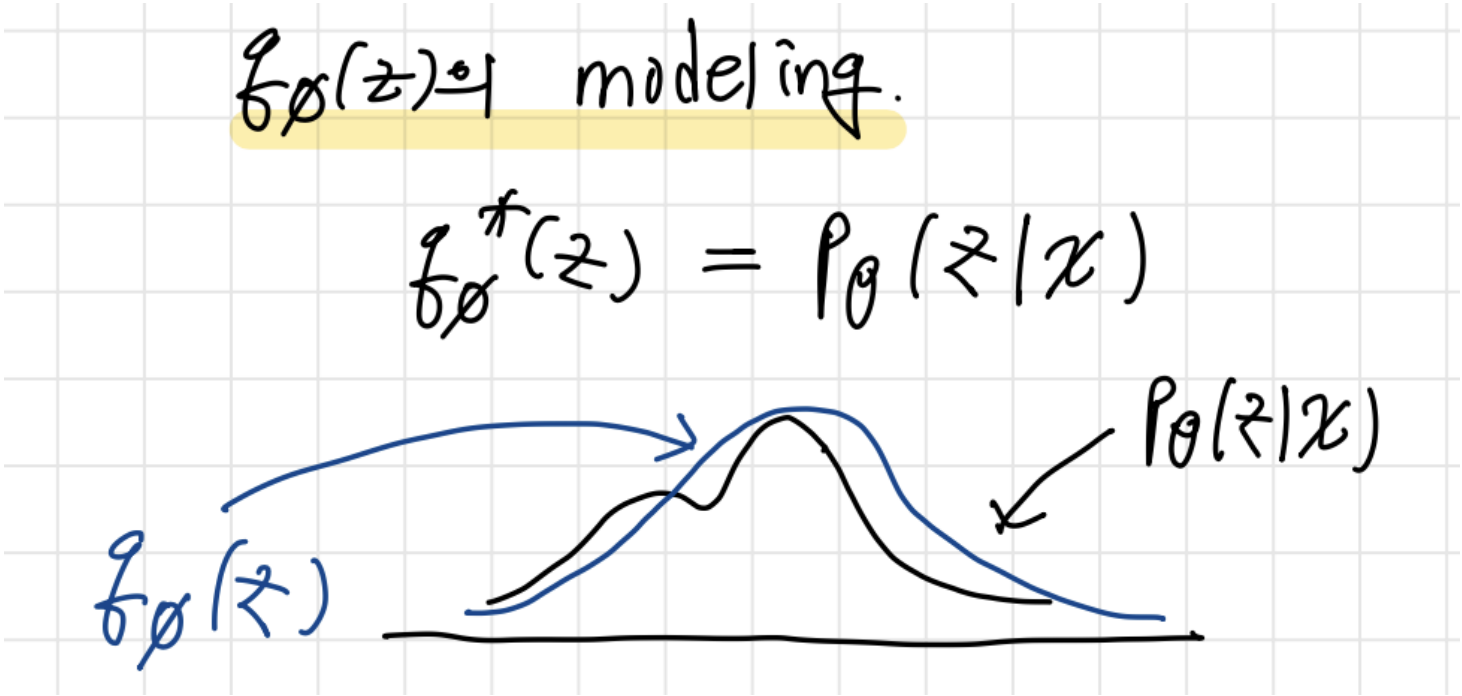
$$= \sum_{i=1}^{N} -\frac{\|\mathbf{x}_i - g_\theta(\mathbf{z}_i)\|^2}{\sigma^2} + \text{const}$$

즉, 결국 squared Loss를 minimize하는 방식으로 확률 분포를 학습해야 한다는 결과가 나오게 된다.

이미지 도메인에서 squared loss를 선호하지 않는 이유

VAE (Variational Auto Encoder)

p(z)가 표준 정규분포를 따른다는 가정을 한다.
그러면 최적의 p(z)를 구하기 위해서는 직접 데이터를 보고 결정하는 것이 최적의 p(z)이이기에 p(z|x)를 근사해야한다. 하지만 p(z|x)자체는 직접 구하기 어렵기 때문에 q라는 parameteric 분포를 설정하여 q라는 분포가 p(z|x)를 최대한 근사하도록 설정합니다. 그래서 이 둘을 KL Divergence를 통해서 최소화하게 됩니다.



VAE (Variational Auto Encoder)

* $q_\phi(z)$는 Gaussian distribution 가정.

$$\arg\min \; KL\left(q_\phi(z) \mid p_\theta(z|x)\right)$$

$$\arg\min \; q_\phi(z) \cdot \log \frac{q_\phi(z)}{p_\theta(z|x)}$$

$$\arg\min \; E_q\left[\log \frac{q_\phi(z)}{p_\theta(z|x)}\right]$$

$$\arg\min \; E_q[\log q_\phi(z)] - E_q[\log p_\theta(z|x)]$$

$$\arg\min \; E_q[\log q_\phi(z)] - E_q\left[\log \frac{p_\theta(x|z)p_\theta(z)}{p_\theta(x)}\right] \quad \text{(Bayes)}$$

$$\arg\min \; E_q\left[\log \frac{q_\phi(z)}{p_\theta(z)}\right] - E_q[\log p(x|z)] + E_q[\log p_\theta(x)]$$

$$KL\left(q_\phi(z) \mid p_\theta(z|x)\right) = E_q\left[\log \frac{q_\phi(z)}{p_\theta(z)}\right] - E_q[\log p(x|z)] + \cancel{E_q[\log p_\theta(x)]}$$

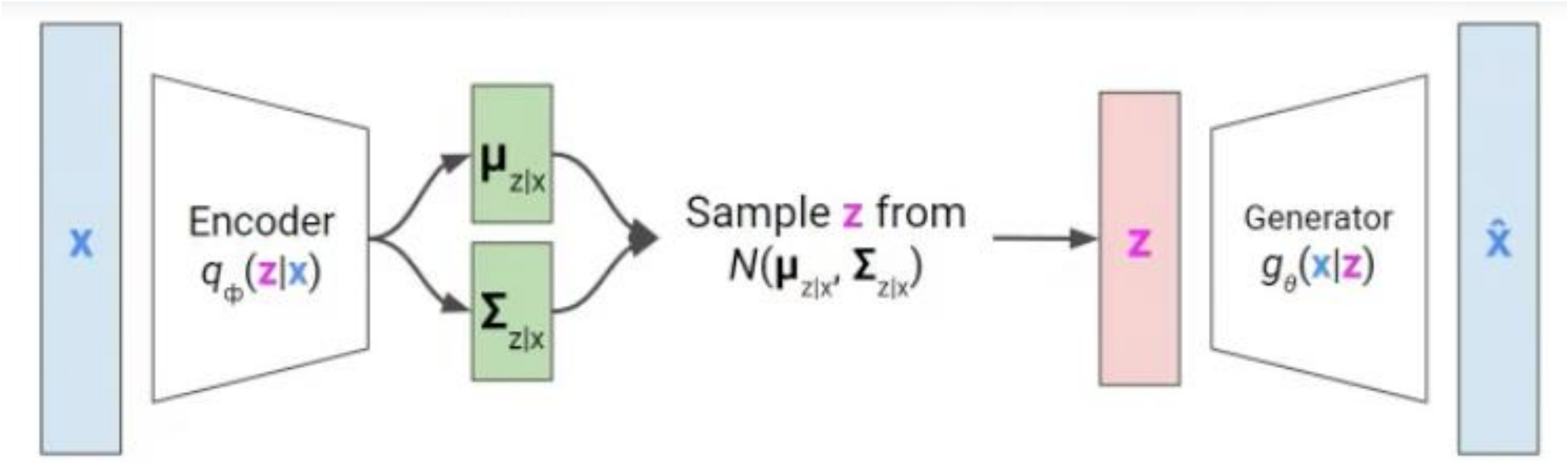
$\log p_\theta(x)$

$$\therefore \log p_\theta(x) = \underbrace{KL\left(q_\phi(z) \mid p_\theta(z|x)\right)}_{\geq 0} - E_q\left[\log \frac{q_\phi(z)}{p_\theta(x)}\right] + E_q[\log p(x|z)]$$

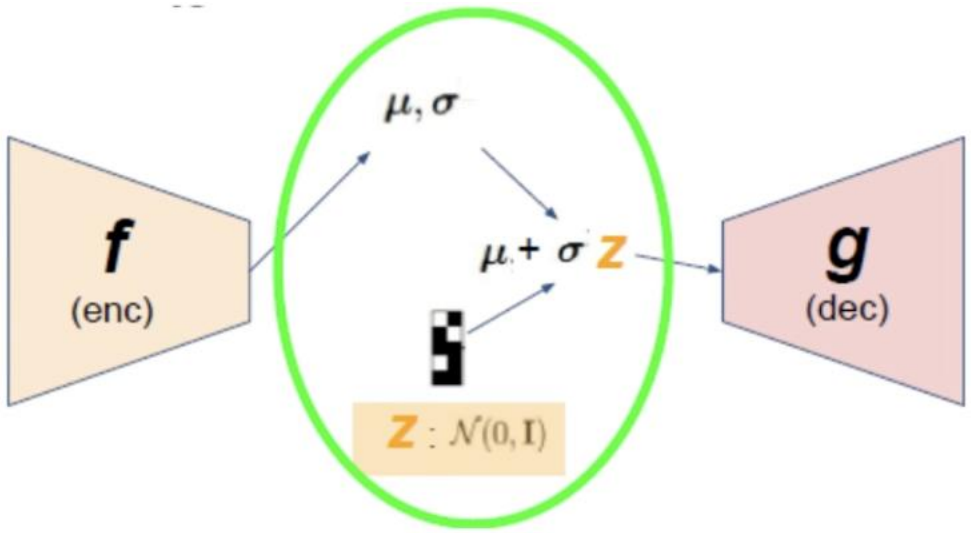
$$\geq E_q[\log p(x|z)] - KL\left(q_\phi(z) \mid p_\theta(z|x)\right)$$

Reconstruction　　Regularization

VAE (Variational Auto Encoder)



Reparameterization Trick

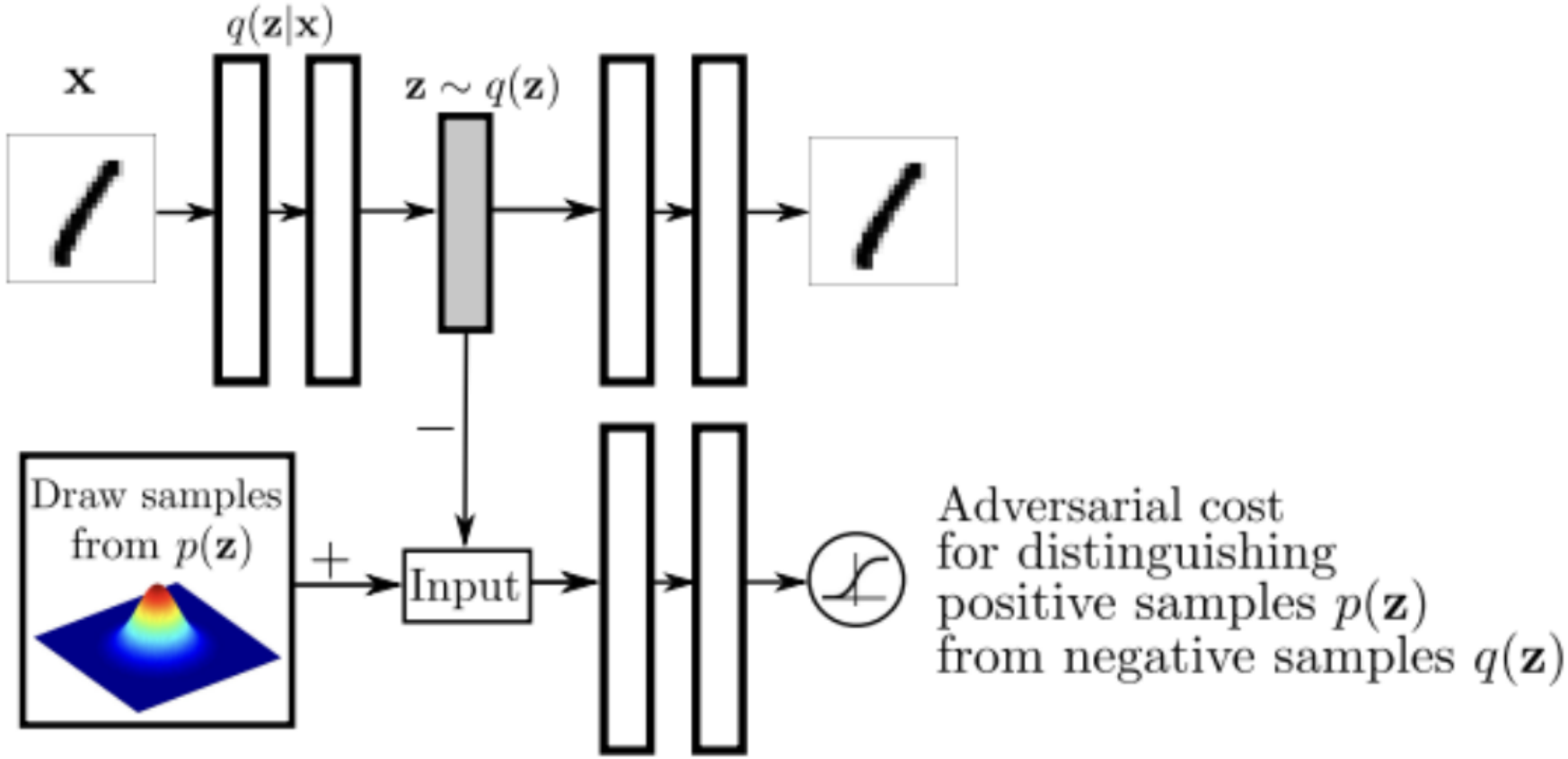


$$z^{i,l} = \mu_i + \sigma_i \odot \epsilon$$
$$\epsilon \sim N(0, I)$$

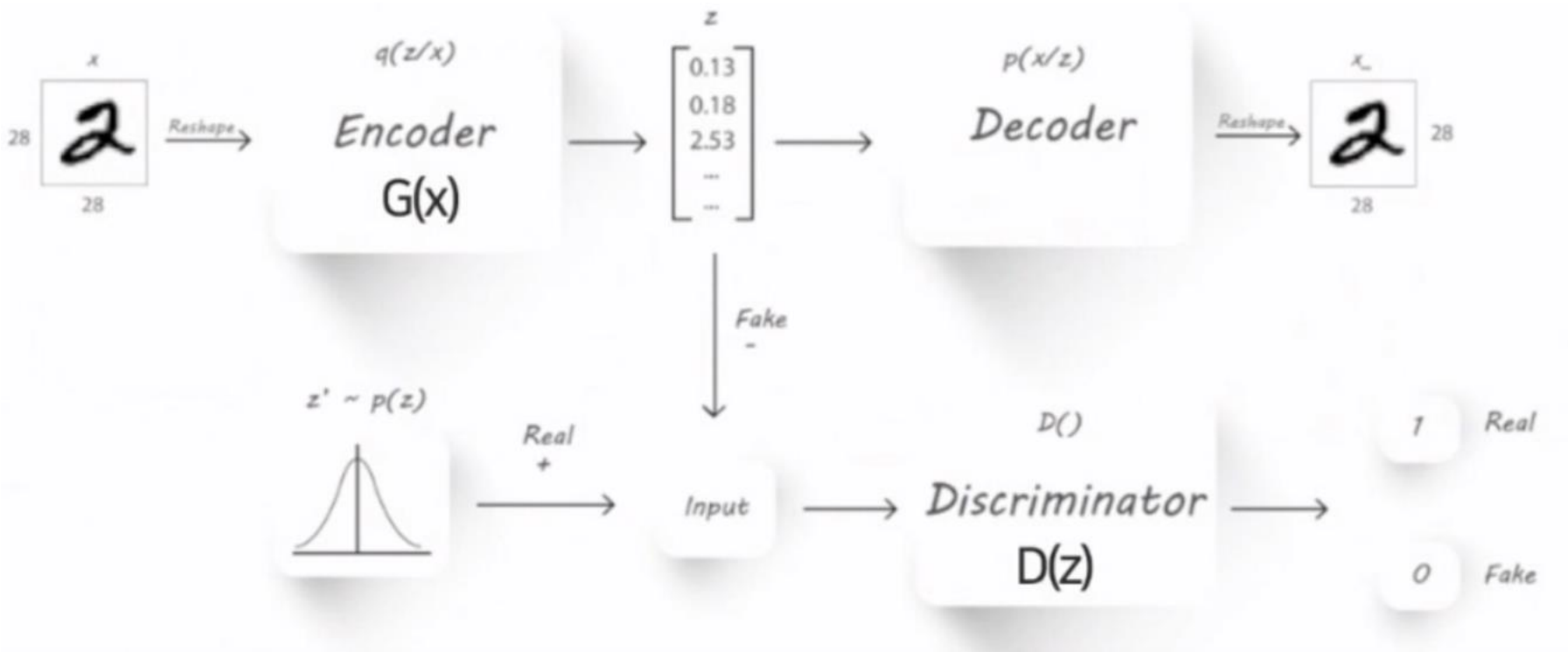
단순히 평균과 분산을 따른다고 하면 Backpropagation을 할 수 없으니, 표쥰 정규분포를 따르는 Noise를 샘플링하여 특정 z값을 추출하고, 이렇게 샘플링된 z를 deocding하는식으로 작동하게 됩니다. 이를 통해서 backpropagation도 가능하고, 근처의 noise 데이터들에 대해서도 안정적인 생성이 가능하게 됩니다.

AAE 구조

Auto Encoder + GAN



AAE



$$\min_G \max_D E_{\mathbf{x}\sim p_{data}}[logD(\mathbf{x})] + E_{\mathbf{z}\sim p(\mathbf{z})}[log(1 - D(G(\mathbf{z})))]$$

VAE vs AAE

VAE(Variational AE)

학습 방향 :
사용자가 임의로 형태를 정한 q(z) 의 분포를
실제 데이터 분포인 p(z)에 맞춰 나감

ex)- 실제 데이터 분포인 p(z)의 수식을 구할 수 없어서,
정규 분포라고 가정한 q(z)을 최대한 p(z)와 비슷한 형태로 조율해감

방법 :
Reconstruction + KL- Divergence 규제항 활용

$$E_{\mathbf{x}\sim p_d(\mathbf{x})}[-logp(\mathbf{x})] < E_{\mathbf{x}}[-log(p(\mathbf{x}|\mathbf{z}))] + E_{\mathbf{x}}[KL(q(\mathbf{z}|\mathbf{x})||p(\mathbf{z}))]$$
$$= E_{\mathbf{x}}[-log(p(\mathbf{x}|\mathbf{z}))] - E_{\mathbf{x}}[H(q(\mathbf{z}|\mathbf{x}))] + E_{q(\mathbf{z})}[-logp(\mathbf{z})]$$
$$= E_{\mathbf{x}}[-log(p(\mathbf{x}|\mathbf{z}))] - E_{\mathbf{x}}[\Sigma_i log\sigma_i(\mathbf{x})] + E_{q(\mathbf{z})}[-logp(\mathbf{z})] + Const$$

AAE(Adversarial AE)

학습 방향 :
사용자가 임의로 정한 p(z)의 분포에 맞게
Aggregated Posterior q(z)을 변형시킴

ex)- 실제 데이터 분포 q(z)가 어떤 형상을 가지든,
내가 원하는 분포 형상을 가진 p(z)로 만들어 주는 네트워크 만들기

방법 :
Reconstruction + Min-max 규제항

$$\min_G \max_D E_{\mathbf{x}\sim p_{data}}[logD(\mathbf{x})] + E_{\mathbf{z}\sim p(\mathbf{z})}[log(1 - D(G(\mathbf{z})))]$$

AAE

```python
class Encoder(nn.Module):
    def __init__(self, latent_dim=10):
        super().__init__()
        self.fc = nn.Sequential(
            nn.Linear(784, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(inplace=True),

            nn.Linear(512, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(inplace=True),

            nn.Linear(256, latent_dim)
        )

    def forward(self, x):
        x = x.view(x.size(0), -1)
        z = self.fc(x)
        return z
```

```python
class Decoder(nn.Module):
    def __init__(self, latent_dim=10):
        super().__init__()
        self.fc = nn.Sequential(
            nn.Linear(latent_dim, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(inplace=True),

            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(inplace=True),

            nn.Linear(512, 784),
            nn.Sigmoid()
        )

    def forward(self, z):
        x_hat = self.fc(z)
        x_hat = x_hat.view(z.size(0), 1, 28, 28)
        return x_hat
```

```python
class Discriminator(nn.Module):
    def __init__(self, latent_dim=10):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(latent_dim, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(inplace=True),

            nn.Linear(128, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(inplace=True),

            nn.Linear(128, 1),
            nn.Sigmoid()
        )

    def forward(self, z):
        validity = self.net(z)
        return validity
```

AAE

AutoEncoder Loss

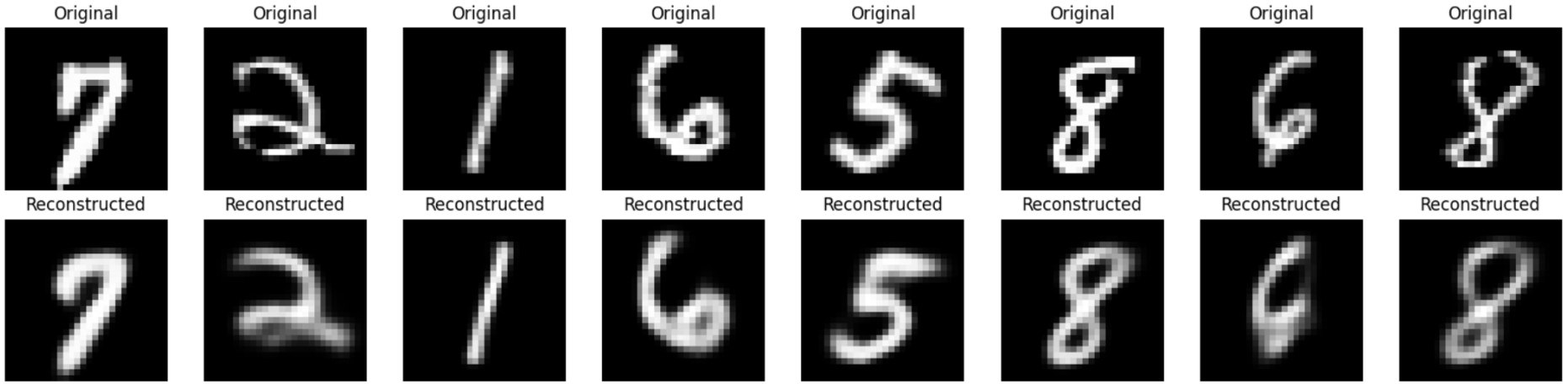
```
z_fake = encoder(imgs)
x_recon = decoder(z_fake)

# 재구성 손실
recon_loss = F.binary_cross_entropy(x_recon, imgs, reduction='sum') / imgs.size(0)

# Adversarial loss
pred_fake = discriminator(z_fake)
valid = torch.ones_like(pred_fake)
adv_loss = bce_loss(pred_fake, valid)

ae_loss = recon_loss + adv_loss
```

Discriminator Loss

```
real_z = torch.randn(imgs.size(0), latent_dim).to(device)
z_fake = encoder(imgs).detach()

pred_real = discriminator(real_z)
pred_fake = discriminator(z_fake)

real_labels = torch.ones_like(pred_real)
fake_labels = torch.zeros_like(pred_fake)

disc_loss_real = bce_loss(pred_real, real_labels)
disc_loss_fake = bce_loss(pred_fake, fake_labels)
disc_loss = disc_loss_real + disc_loss_fake
```

Result

Reconstruction



Generation (정규분포에서 sampling 진행)