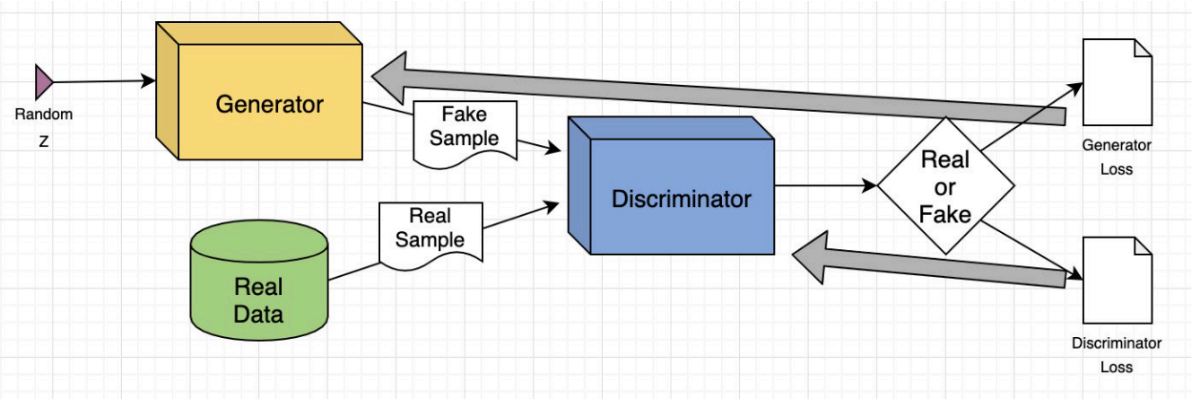
- Generative Model
  - GAN: Generative Adversarial Networks https://arxiv.org/pdf/1406.2661
    - Novelty
    - Architecture:
    - Intuition:
    - Proof:
  - Findings of :
    - Limitation of V(D*, G) as the loss function:
  - VAE: Auto-Encoding Variational Bayes https://arxiv.org/pdf/1312.6114
    - Novelty:
    - Architecture:
    - Intuition:
    - Proof:
  - Understanding disentangling in β-VAE https://arxiv.org/pdf/1804.03599
    - Novelty:
    - Architecture:
    - Experiment of disentangling on Dsprites dataset:
    - Code Implementation:
  - Variational Inference with Normalizing Flows https://arxiv.org/pdf/1505.05770
    - Novelty:
    - Prerequisite:
      - Deep Latent Gaussian Models:
      - Change of Variables for Bijective Transformations:
    - Architecture:
    - Code Implementation:
  - MADE: Masked Autoencoder for Distribution Estimation https://arxiv.org/pdf/1502.03509
    - Novelty:
    - Architecture:
    - Limitation:

Generative Model

GAN: Generative Adversarial Networks https://arxiv.org/pdf/1406.2661



Novelty

- Adversarial Training Paradigm: Before GAN, the generative models often relied on the maximum likelihood estimation. GAN introduced a discrimitive network as the adversary to the generative network, which drives the generator to generate the samples that are close to the real samples.
- Implicit density estimation: The generative model is not trained with the explicit density estimation. Instead, it is trained with purely the log-likelihood loss.

Architecture:

Components:

- $G(z)$: The generator network. It takes the noise z as the input and generates the sample x.
- $p_z(z)$: The prior distribution of the noise.
- $p_d(x)$: The real data distribution.
- $D(x)$: The discriminator network. It takes the sample x as the input and outputs the probability of the sample being real.

Objective:

$$\min_G \max_D V(D,G) = E_{x\sim p_d}[\log D(x)] + E_{z\sim p_z}[\log(1 - D(G(z)))]$$

Intuition:

- The discriminator is trying to maximize the expected value of $logD(x)$ for x sampled from the real data distribution $p_d(x)$. And it is trying to minimize the expected value of $log(D(G(z)))$ for z sampled from the prior distribution (noise) $p_z(z)$ and image

generated by the generator function $G(z)$, which is then re-written as maximizing the expected value of $log(1 - D(G(z)))$ in the objective function.

- The generator is trying to maximize the expected value of $log(D(G(z)))$ for the fake samples generated by the generator $G(z)$. In other words, it is trying to generate the samples that can fool the discriminator, and it is written as minimizing expected value of $log(1 - D(G(z)))$ in the objective function.

Proof:

Theorem 1: For any generator G, the optimal discriminator D is

$$D_G^*(x) = \frac{p_d(x)}{p_d(x) + p_g(x)}$$

Proof:

$$\begin{aligned} V(D, G) &= E_{x \sim p_d}[\log D(x)] + E_{z \sim p_z}[\log(1 - D(G(z)))] \\ &= \int_x p_d(x) \log D(x) dx + \int_z p_z(z) \log(1 - D(G(z))) dz \\ &= \int_x p_d(x) \log D(x) dx + \int_x p_g(x) \log(1 - D(x)) dx \\ &= \int_x (p_d(x) \log D(x) + p_g(x) \log(1 - D(x))) dx \end{aligned}$$

Consider the function $f(y) = a \log y + b \log(1 - y)$, the function achieves the maximum when $y = \frac{a}{a+b}$ (by taking the derivative and set it to 0).

Therefore, the optimal discriminator is $D^*(x) = \frac{p_d(x)}{p_d(x)+p_g(x)}$

Theorem 2: Give an optimal discriminator D^*, the optimal generator G^*, which aims to minimize the objective function, is obtained when $p_g = p_d$

Proof:

$$\begin{aligned} V(D^*, G) &= E_{x \sim p_d}[\log D^*(x)] + E_{x \sim p_g}[\log(1 - D^*(x))] \\ &= E_{x \sim p_d}[\log(\frac{p_d(x)}{p_d(x) + p_g(x)})] + E_{x \sim p_g}[\log(\frac{p_g(x)}{p_d(x) + p_g(x)})] \\ &= E_{x \sim p_d}[\log(\frac{p_d(x)}{p_d(x) + p_g(x)})] + E_{x \sim p_g}[\log(\frac{p_g(x)}{p_d(x) + p_g(x)})] \\ &= \int_x (p_d(x) \log(\frac{p_d(x)}{p_d(x) + p_g(x)}) + p_g(x) \log(\frac{p_g(x)}{p_d(x) + p_g(x)})) dx \end{aligned}$$

Let a_i and b_i to denote the probability of x_i being sampled from $p_d(x)$ and $p_g(x)$ respectively.

We can rewrite the integral as the summation of the probabilities in the discrete case:

$$\sum_{i=1}^{n} [a_i \log(\frac{a_i}{a_i + b_i}) + b_i \log(\frac{b_i}{a_i + b_i})]$$

And we know that the summation of probability distribution function is 1. Then we get the following constraint:

$$\sum_{i=1}^{n} a_i = 1 \quad \text{and} \quad \sum_{i=1}^{n} b_i = 1$$

Then, we can formulate the Lagrangian multiplier function:

$$\begin{aligned} L(a, b, \lambda, \mu) &= \sum_{i=1}^{n} [a_i \log(\frac{a_i}{a_i + b_i}) + b_i \log(\frac{b_i}{a_i + b_i})] - \lambda([\sum_{i=1}^{n} a_i] - 1) - \mu([\sum_{i=1}^{n} b_i] - 1) \\ &= \sum_{i=1}^{n} [a_i \log(\frac{a_i}{a_i + b_i}) - \lambda a_i + b_i \log(\frac{b_i}{a_i + b_i}) - \mu b_i] + \lambda + \mu \end{aligned}$$

Let $F_i = a_i \log(\frac{a_i}{a_i+b_i}) - \lambda a_i + b_i \log(\frac{b_i}{a_i+b_i}) - \mu b_i$.

We want to have $\frac{\partial F_i}{\partial a_i} = 0 \quad \text{and} \quad \frac{\partial F_i}{\partial b_i} = 0$ for every i, to make sure F is at the minimum.

By taking the partial derivative of F_i with respect to a_i and b_i, we get:

$$\frac{\partial F_i}{\partial a_i} = \log \frac{a_i}{a_i + b_i} - \lambda = 0 \implies \lambda = \log \frac{a_i}{a_i + b_i}$$

$$\frac{\partial F_i}{\partial b_i} = \log \frac{b_i}{a_i + b_i} - \mu = 0 \implies \mu = \log \frac{b_i}{a_i + b_i}$$

We then have $\frac{a_i}{a_i+b_i} = e^{\lambda} \quad \text{and} \quad \frac{b_i}{a_i+b_i} = e^{\mu}$,

and $\frac{a_i}{a_i+b_i} + \frac{b_i}{a_i+b_i} = 1 = e^{\lambda} + e^{\mu}$.

If we let $e^{\lambda} = e^{\mu} = \frac{1}{2}$, then we have $a_i = b_i$.

Thus, we have $\log \frac{a_i}{a_i+b_i} = \log \frac{b_i}{a_i+b_i} \implies a_i = b_i$.

Therefore, if we have $a_i = b_i$ for all x_i, then the objective function $V(D^*, G^*)$ is minimized with value of $-2log2$

Findings of $V(D^*, G)$:

$V(D^*, G)$ can also be written as

- $-2log2 + KL(p_d(x)||\frac{p_d(x)+p_g(x)}{2}) + KL(p_g(x)||\frac{p_d(x)+p_g(x)}{2})$
- $-2log2 + 2JS(p_d(x)||p_g(x))$

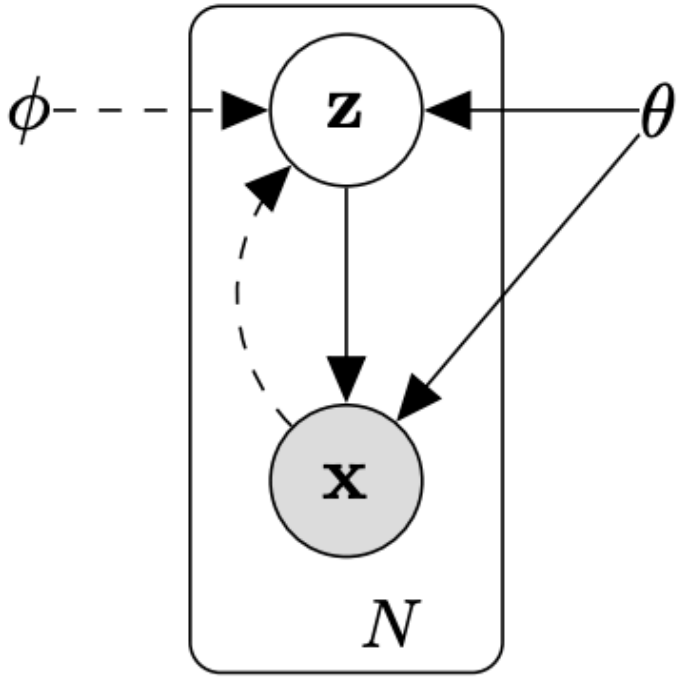
This is because:

$$
\begin{aligned}
V(D^*, G) &= E_{x\sim p_d}[\log(\frac{p_d(x)}{p_d(x)+p_g(x)})] + E_{x\sim p_g}[\log(\frac{p_g(x)}{p_d(x)+p_g(x)})] \\
&= E_{x\sim p_d}[\log(\frac{p_d(x)}{2 * (p_d(x)+p_g(x))}) - \log 2] + E_{x\sim p_g}[\log(\frac{p_g(x)}{2 * (p_d(x)+p_g(x))}) - \log 2] \\
&= -2log2 + E_{x\sim p_d}[\log(\frac{p_d(x)}{p_d(x)+p_g(x)})] + E_{x\sim p_g}[\log(\frac{p_g(x)}{p_d(x)+p_g(x)})] \\
&= -2log2 + KL(p_d(x)||\frac{p_d(x)+p_g(x)}{2}) + KL(p_g(x)||\frac{p_d(x)+p_g(x)}{2}) \\
&= -2log2 + 2JS(p_d(x)||p_g(x))
\end{aligned}
$$

Limitation of V(D*, G) as the loss function:

Max value of JS divergence is always log2 when two distributions do not overlap. However, the differences between their centers might vary. For the two non-overlapping distributions with close centers, they are supposed to be similar, but the JS divergence is still log2. And this brings difficulty in the training of the generator.

VAE: Auto-Encoding Variational Bayes https://arxiv.org/pdf/1312.6114



Novelty:

- Implicit density estimation: The generative model is not trained with the explicit density estimation. Instead, it is trained with purely the log-likelihood loss.
- Latent Variable: The VAE maps the input x to latent variable z sampled from the trained latent space distribution, and then to the input x again. This gives the overall architecture a generative nature.
- Reparameterization Trick: The VAE uses the reparameterization trick (Normal Distribution) to sample the latent variables from the posterior distribution, which maintains the differentiable property of the model.
- Evidence Lower Bound (ELBO): The VAE uses the Evidence Lower Bound (ELBO) as the loss function. By minimizing the ELBO, we can minimize the lower bound of the log-likelihood of the input x.
- The VAE architure is generative because

Architecture:

Components:

- $q_\phi(z|x)$: The trained encoder network. It takes the input x and outputs the parameters of the posterior distribution of the latent variables z.
- $p_\theta(z)$: The "true" prior distribution of the latent variables we want the trained encoder network match. In the paper, it is the standard normal distribution. And it means that we want the distrubution of $q_\phi(z|x)$ to get close to the standard normal distribution.
- $p_\theta(x|z)$: The trained decoder network. It takes the latent variables z and outputs the parameters of the generative distribution of the input x.

Objective:

$$\max_{\phi,\theta} \mathrm{E}_{z\sim p_\phi(z|x)}[\log p_\theta(x|z)] - \mathrm{KL}(q_\phi(z|x)||p_\theta(z))$$

- The loss function is derived from the Evidence Lower Bound (ELBO) of the log-likelihood of the input x.
- The first term is the reconstruction loss. It is the difference between the actual input x and the generated output $\hat{x}$ by the decoder network.
- The second term is the KL divergence between the posterior distribution $q_\phi(z|x)$ of the latent variables z and the prior distribution $p_\theta(z)$. By minimizing the KL divergence, we can make the posterior distribution $q_\phi(z|x)$ of the latent variables z close to the prior distribution $p_\theta(z)$, which is the standard normal distribution in the context of this paper.

Intuition:

- We train the encoder network $q_\phi(z|x)$ to map the input x to the latent variable z. We are also trying to make the distribution of the trained latent space to be close to the prior distribution $p_\theta(z)$, which is the standard normal distribution in the context of this paper.
- We train the decoder network $p_\theta(x|z)$ to map the latent variable z to reproduce input $\hat{x}$

Why the encoder network makes the overall architecture generative?

Suppose the training dataset contains only images of circles. Inputting a image of circle, we should expect the value of latent variable z to be close to the center of the latent space distribution, and it will lead to generate a image close to a circle. If we input an image of a triangle, we should expect VAE to generate "something" that has circle-like features instead of a strict triangle (may even be very different from a triangle). Because the encoder network $q_\phi(z|x)$ maps the input x to the latent variable z that is sampled from the trained latent space distribution. If the input image is different from the training dataset, we can expect the value of z to be distant from the the center of the latent space distribution, but it should not be too far away from the center of the latent space distribution (like completely outside the latent space and its nearby), otherwise, it means that the encoder network overfits. Giving such condition,

the decoder network $p_\theta(x|z)$ should generate a image more or less with the features of a circle.

Proof:

Evidence Lower Bound (ELBO):

For any datapoint x, their likelihood function can be written as the marginal likelihood:

$$\begin{aligned} p(x) &= \int_z p(x,z)dz \\ &= \int_z q(z|x)\frac{p(x,z)}{q(z|x)}dz \\ &= \mathrm{E}_{z\sim q(z|x)}[\frac{p(x,z)}{q(z|x)}] \end{aligned}$$

By Jensen's inequality $\mathrm{E}[f(x)] \geq f(\mathrm{E}[x])$, we have:

$$\begin{aligned} \log p(x) &= \log \mathrm{E}_{z\sim q(z|x)}[\frac{p(x,z)}{q(z|x)}] \\ &\geq \mathrm{E}_{z\sim q(z|x)}[\log \frac{p(x,z)}{q(z|x)}] \\ &= \mathrm{E}_{z\sim q(z|x)}[\log p(x|z)] - \mathrm{E}_{z\sim q(z|x)}[\log \frac{q(z|x)}{p(z)}] \\ &= \mathrm{E}_{z\sim q(z|x)}[\log p(x|z)] - \mathrm{KL}(q(z|x)||p(z)) \end{aligned}$$

Therefore, the loss function can be written as:

$$\max_{\phi,\theta} \mathrm{E}_{z\sim q_\phi(z|x)}[\log p_\theta(x|z)] - \mathrm{KL}(q_\phi(z|x)||p_\theta(z))$$

Reparameterization Trick:

The reparameterization trick is used to sample the latent variables from the posterior distribution $q_\phi(z|x)$ in a differentiable way.

We let $q_\phi(z|x)$ output the parameters of the distribution function (Normal Distribution) for the latent variables z, which is the mean μ and the standard deviation σ of the normal distribution.

We can then rewrite the latent variables z as a deterministic function of the random noise ϵ and the parameters ϕ and x:

$$z = g(\mu, \phi, \epsilon) = \mu + \epsilon\sigma, \quad \text{where} \quad \epsilon \sim \mathrm{N}(0, I) \text{ is a random noise.}$$

Proof:

We want $q_\phi(z|x)$ to be a normal distribution, where μ and σ are computed based on the input x.Then we get $q_\phi(z|x) = p(z|\mu, \sigma)$.

Suppose we try to estimate the value based on z (for example, the final loss function can be seen as a function based on z), let's call it $f(z)$.

$$\mathrm{E}_{z\sim p(z|\mu,\sigma)}[f(z)] = \int_z p(z|\mu, \sigma)f(z)dz$$

Let $z = g(\epsilon) = \mu + \epsilon\sigma$, then we have:

$$\begin{aligned} p(g(\epsilon)|\mu, \sigma)g'(\epsilon)\, d\epsilon &= p(\mu + \epsilon\sigma|\mu, \sigma)\sigma\, d\epsilon \\ &= \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(\mu+\epsilon\sigma-\mu)^2}{2\sigma^2}} \sigma\, d\epsilon \\ &= \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(\epsilon\sigma)^2}{2\sigma^2}} \sigma\, d\epsilon \\ &= \frac{1}{\sqrt{2\pi}} e^{-\frac{(\epsilon)^2}{2}}\, d\epsilon \\ &= p(\epsilon|0, 1)\, d\epsilon \end{aligned}$$

Therefore, by changing the variable of z to $g(\epsilon)$, we get:

$$\begin{aligned} \mathrm{E}_{z\sim p(z|\mu,\sigma)}[f(z)] &= \int_z p(z|\mu, \sigma)f(z)dz \\ &= \int_\epsilon p(\epsilon|\mu, \sigma)f(\mu + \epsilon\sigma)g'(\epsilon)d\epsilon \\ &= \int_\epsilon p(\epsilon|0, 1)f(\mu + \epsilon\sigma)\, d\epsilon \\ &= \mathrm{E}_{\epsilon\sim p(\epsilon|0,1)}[f(\mu + \epsilon\sigma)] \end{aligned}$$

Solution to the KL divergence:

$$
\begin{aligned}
\log \frac{q_\phi(z|x)}{p_\theta(z)} &= \log \frac{N(z;\mu,\sigma)}{N(z|0,1)} \\
&= \log \frac{\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(z-\mu)^2}{2\sigma^2}}}{\frac{1}{\sqrt{2\pi}} e^{-\frac{z^2}{2}}} \\
&= -\log\sigma + \log e^{-\frac{(z-\mu)^2}{2\sigma^2} + \frac{z^2}{2}} \\
&= -\log\sigma - \frac{(z-\mu)^2}{2\sigma^2} + \frac{z^2}{2} \\
&= -\log\sigma - \frac{1}{2\sigma^2}(z-\mu)^2 + \frac{1}{2}z^2
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{KL}(q_\phi(z|x)||p_\theta(z)) &= \mathrm{E}_{z\sim q_\phi(z|x)}[\log \frac{q_\phi(z|x)}{p_\theta(z)}] \\
&= \mathrm{E}_{z\sim q_\phi(z|x)}[-\log\sigma - \frac{1}{2\sigma^2}(z-\mu)^2 + \frac{1}{2}z^2] \\
&= -\log\sigma - \frac{1}{2\sigma^2}\mathrm{E}_{z\sim q_\phi(z|x)}[(z-\mu)^2] + \frac{1}{2}\mathrm{E}_{z\sim q_\phi(z|x)}[z^2] \\
&= -\log\sigma - \frac{1}{2\sigma^2}\sigma^2 + \frac{1}{2}(\mu^2+\sigma^2) \\
&= -\log\sigma - \frac{1}{2} + \frac{1}{2}\mu^2 + \frac{1}{2}\sigma^2 \\
&= \frac{1}{2}\mu^2 + \frac{1}{2}\sigma^2 - \frac{1}{2}\log\sigma^2 - \frac{1}{2} \\
&= \frac{1}{2}[(\mu^2+\sigma^2) - \log\sigma^2 - 1]
\end{aligned}
$$

Note that

- $\mathrm{E}_{z\sim q_\phi(z|x)}[z^2] = \mathrm{E}_{z\sim q_\phi(z|x)}[(\mu+\epsilon\sigma)^2] = \mu^2+\sigma^2$ because of the second moment equation of the normal distribution.
- $\mathrm{E}_{z\sim q_\phi(z|x)}[(z-\mu)^2] = \sigma^2$ because of the variance equation of the normal distribution.

Understanding disentangling in β-VAE https://arxiv.org/pdf/1804.03599

Novelty:

- Disentangling: The authors suggested with large β, the latent space is disentangled into different independent factors of variation.

- Weighted factor of KL Divergence: The authors used the lagrangian multiplier to regularize the KL divergence between the posterior distribution and the prior distribution of latent variables.

Architecture:

Objective:

$$\max_{\phi,\theta} \mathrm{E}_{z\sim q_\phi(z|x)}[\log p_\theta(x|z)] - \beta|\mathrm{KL}(q_\phi(z|x)||p_\theta(z)) - C|$$

Intuition:

We can think the original VAE objective function as maximization problem of reconstruction log likelihood with constraint of KL divergence between the posterior distribution and the prior distribution of latent variables:

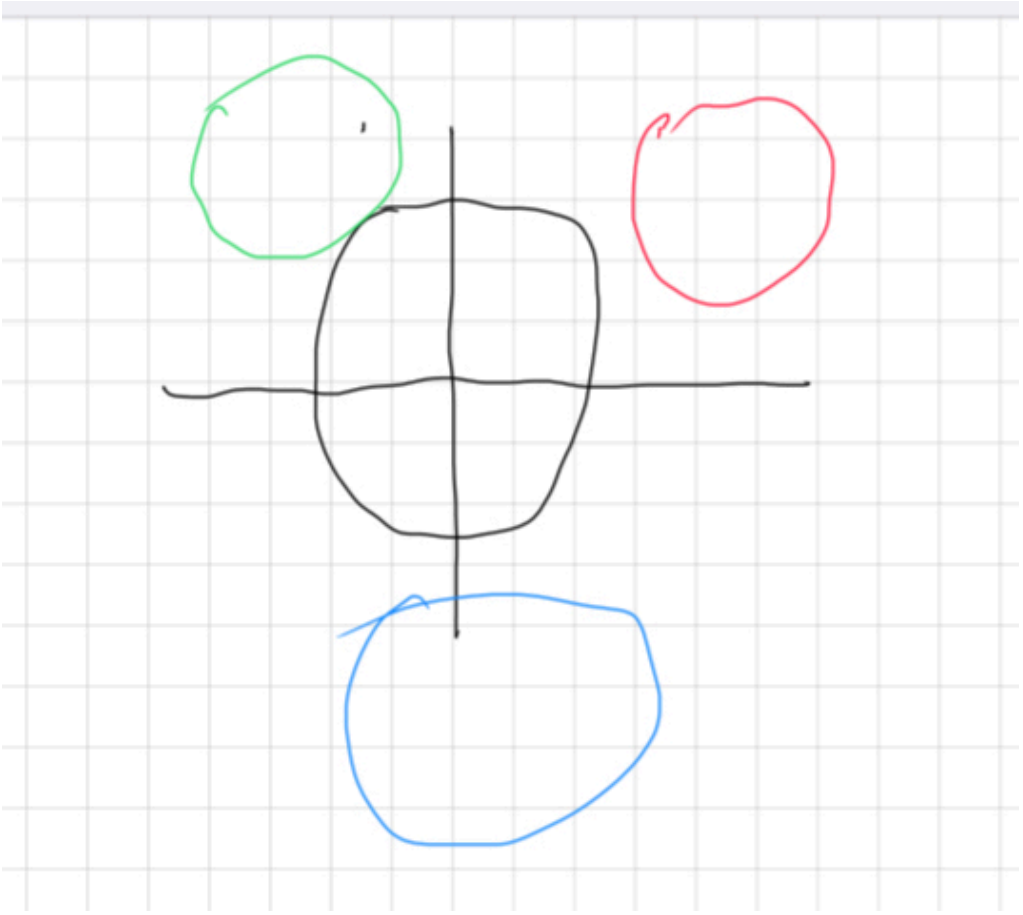
$$\max_{\phi,\theta} \mathrm{E}_{z\sim q_\phi(z|x)}[\log p_\theta(x|z)], \quad \text{with constraint: } \mathrm{KL}(q_\phi(z|x)||p_\theta(z)) - C = 0$$

By applying the lagrangian multiplier, we can rewrite the objective function as:

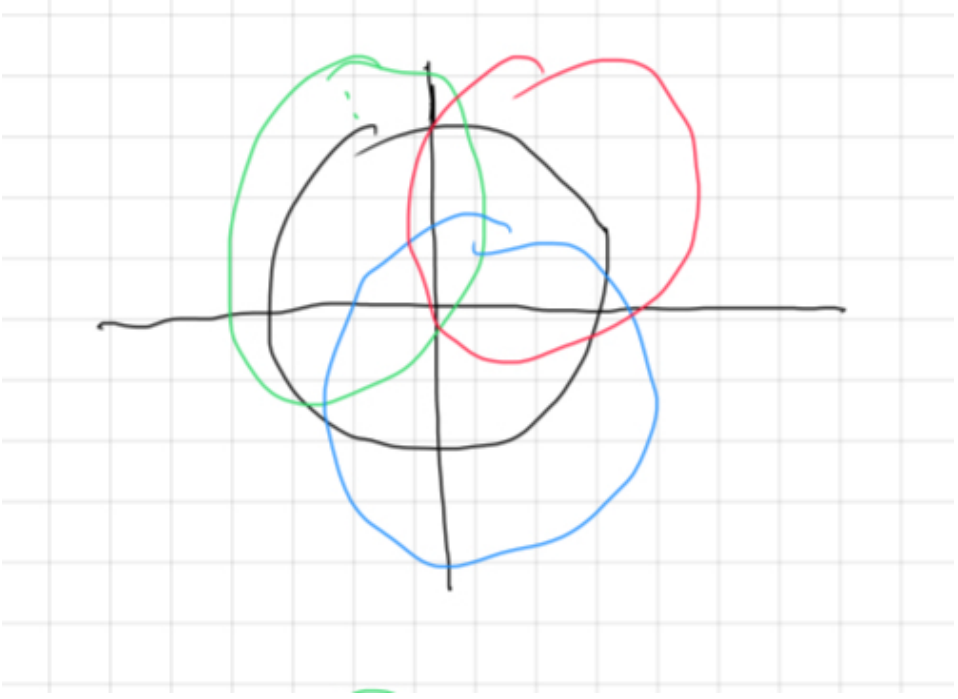
$$\max_{\phi,\theta} \mathrm{E}_{z\sim q_\phi(z|x)}[\log p_\theta(x|z)] - \beta(\mathrm{KL}(q_\phi(z|x)||p_\theta(z)) - C)$$

By modifying $(q_\phi(z|x)||p_\theta(z)) - C)$, to be $|(q_\phi(z|x)||p_\theta(z)) - C|$, we are making the constraint constant term C differentiable. Because it will make the gradient of $\mathrm{KL}(q_\phi(z|x)||p_\theta(z))$ dependent on the sign of $\mathrm{KL}(q_\phi(z|x)||p_\theta(z)) - C$.

Comparison between different β and C:

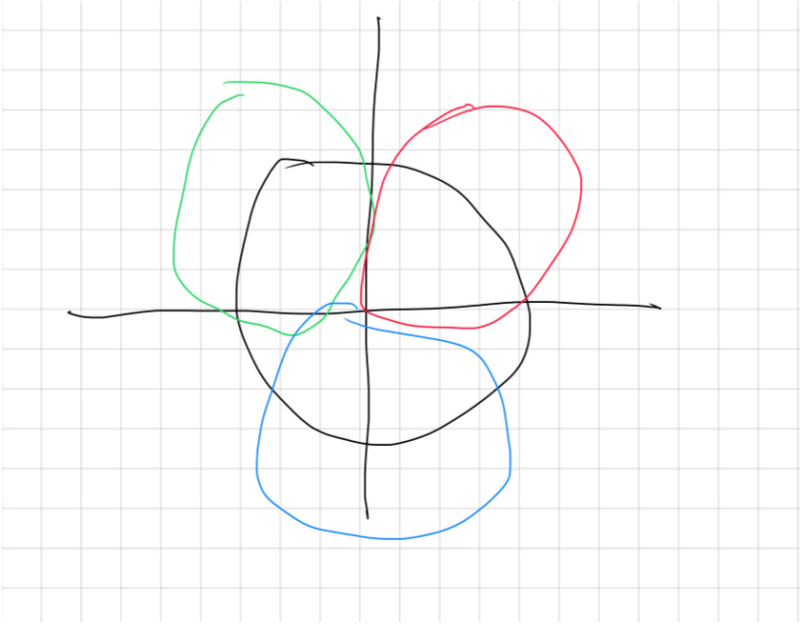


- When β is small, it is more likely to have a larger $\mathrm{KL}(q_\phi(z|x)||p_\theta(z))$, which means that the posterior distribution $q_\phi(z|x)$ is more likely to be distant to the prior distribution $p_\theta(z)$. We can get a decent reconstruction network without having it learning the disentangling property of the latent space.



- When β is large, it is more likely to have a smaller $\mathrm{KL}(q_\phi(z|x)||p_\theta(z))$, which means that the posterior distribution $q_\phi(z|x)$ is more likely to be close to the prior distribution $p_\theta(z)$. We can get a better disentangling property of the latent space. However, it is at the cost

of the reconstruction quality because the distribution between the latent space of different classes is more likely to be tightly overlapped.



- By introducing a constant C, we can control the trade-off between the reconstruction quality and the disentangling property of the latent space. Because it keeps the distribution of the latent space of different classes apart from $N(0, 1)$, which can also be seen as an act to keep the latent space of different classes apart from each other. It can help to maintain a good quality of reconstruction while having a disentangling property of the latent space.

Analysis of β and C from the perspective of Lagrangian multiplier:

Consider the objective function L with respect to β:

$$\frac{dL}{d\beta} = C$$

That means if we increase β by a small amount $\Delta\beta$, we will have L increased by $\Delta\beta * C$.

Therefore, by introducing a constant C, we make log likelihood of reconstruction higher.

Experiment of disentangling on Dsprites dataset:

Original VAE (Beta=0, C=0):

Beta-VAE (Beta=100, C=37):

Based on the results, we can see that the latent traversal of beta-vae is more smooth and has better representation. Alought the authors did not provide detailed theoretical analysis on why increasing β can lead to better disentangling, based on the empirical results, we can conclude that it is more likely to have a better disentangling property of the latent space when β is larger.

Code Implementation:

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt


# --- Define a simple convolutional VAE architecture ---
class VAE(nn.Module):
    def __init__(self, latent_dim=10):
        super(VAE, self).__init__()
        self.latent_dim = latent_dim
        # Encoder: input (1, 64, 64) -> feature vector
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=4, stride=2, padding=1),  # -> (32, 32,
32)
            nn.ReLU(),
            nn.Conv2d(32, 32, kernel_size=4, stride=2, padding=1), # -> (32, 16,
16)
            nn.ReLU(),
            nn.Conv2d(32, 32, kernel_size=4, stride=2, padding=1),# -> (32, 8, 8)
            nn.ReLU(),
            nn.Conv2d(32, 32, kernel_size=4, stride=2, padding=1),# -> (32, 4, 4)
            nn.ReLU(),
            nn.Flatten()  # 32*4*4 = 512
        )
        # Linear layers to output the mean and log-variance of z
        self.fc_mu = nn.Linear(32 * 4 * 4, latent_dim)
        self.fc_logvar = nn.Linear(32 * 4 * 4, latent_dim)
        # Decoder: project latent vector back to image space
        self.decoder_input = nn.Linear(latent_dim, 32 * 4 * 4)
        self.decoder = nn.Sequential(
            nn.Unflatten(1, (32, 4, 4)),
            nn.ConvTranspose2d(32, 32, kernel_size=4, stride=2, padding=1),  # ->
(32, 8, 8)
            nn.ReLU(),
            nn.ConvTranspose2d(32, 32, kernel_size=4, stride=2, padding=1),   # -
> (32, 16, 16)
            nn.ReLU(),
            nn.ConvTranspose2d(32, 32, kernel_size=4, stride=2, padding=1),    #
-> (32, 32, 32)
            nn.ReLU(),
            nn.ConvTranspose2d(32, 1, kernel_size=4, stride=2, padding=1),     #
```

```
-> (1, 64, 64)
            nn.Sigmoid()
        )

    def encode(self, x):
        h = self.encoder(x)
        mu = self.fc_mu(h)
        logvar = self.fc_logvar(h)
        return mu, logvar

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def decode(self, z):
        h = self.decoder_input(z)
        return self.decoder(h)

    def forward(self, x):
        mu, logvar = self.encode(x)
        z = self.reparameterize(mu, logvar)
        x_recon = self.decode(z)
        return x_recon, mu, logvar

# Loss function with adjustable beta (β = 1 for VAE; β > 1 for β-VAE)
def loss_function(x, x_recon, mu, logvar, beta=1.0, C=0.0):
    # Binary cross entropy for reconstruction
    recon_loss = nn.functional.binary_cross_entropy(x_recon, x, reduction='sum')
/ x.size(0)
    # KL divergence loss
    kl_loss = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp()) / x.size(0)
    kl_loss = kl_loss - C
    return recon_loss + beta * torch.abs(kl_loss), recon_loss, torch.abs(kl_loss)
```

Variational Inference with Normalizing Flows https://arxiv.org/pdf/1505.05770

Novelty:

- Normalizing Flows: The authors proposed a new method to sample from the posterior distribution of the latent variables by using the normalizing flows to transform the simple distribution (like Gaussian Distribution) into a complex distribution.

Prerequisite:

Deep Latent Gaussian Models:

$$p(x, z_1, z_2, ..., z_L) = p(x|f_0(z_1)) \prod_{l=1}^{L} p(z_l|f_l(z_{l+1}))$$

Change of Variables for Bijective Transformations:

Let $z' = f(z)$, then we have:

$$q(z') = q(z) \left| \frac{\partial f^{-1}}{\partial z'} \right| = q(z) \left| \frac{\partial f}{\partial z} \right|^{-1}$$

Proof:

$$\int_{z'} q(z')dz' = \int_{z} q(z) \left| \frac{\partial f^{-1}}{\partial z'} \right| dz' = \int_{z} q(z) \left| \frac{\partial f}{\partial z} \right|^{-1} dz = 1$$

Let $f : A \rightarrow f(A)$ be a bijective function, then we have for any $z \in A$, $z' = f(z)$.

Because f is a bijective function, then the inverse function f^{-1} exists. And the probability for $z' \in f(A) = U$ is the same as the probability for $z \in A$.

$$P(z' \in U) = P(z \in A)$$

$$\int_{z' \in U} q(z')dz' = \int_{z \in A} q(z)dz$$

We know that $z = f^{-1}(z')$, and $(f^{-1})^{-1} = f$. By the change of variables, we have:

$$\begin{aligned} \int_{z \in A} q(z)dz &= \int_{z' \in U} q(f^{-1}(z')) \left| \det(\frac{\partial f^{-1}}{\partial z'}) \right| dz' \\ &= \int_{z' \in U} q(z) \left| \det(\frac{\partial f}{\partial z}) \right|^{-1} dz' \quad \text{because } \det(J_{f^{-1}}(p)) = \det(J_f^{-1}(f^{-1}(p))) \end{aligned}$$

Therefore, we have:

$$\int_{z' \in U} q(z')dz' = \int_{z' \in U} q(z) \left| \det(\frac{\partial f}{\partial z}) \right|^{-1} dz'$$

Then we have:

$$q(z') = q(z) \left| \det(\frac{\partial f}{\partial z}) \right|^{-1}$$

Architecture:

Let $q_K(z)$ obtained by successively applying K bijective transformations $f_1, f_2, ..., f_K$ on z_0, we have:

$$q_K(z_K) = q_0(z_0) \prod_{k=1}^{K} \left| \det(\frac{\partial f_k}{\partial z_{k-1}}) \right|^{-1}$$

Apply the ln, then we get:

$$\ln q_K(z_K) = \ln q_0(z_0) - \sum_{k=1}^{K} \ln \left| \det(\frac{\partial f_k}{\partial z_{k-1}}) \right|$$

Objective:

Similar to ELBO in VAE, we have:

$$\begin{aligned} F(x) &= \mathrm{E}_{z\sim q_\phi(z|x)}[\ln q_\phi(z|x) - \ln p(x,z)] \\ &= \mathrm{E}_{z\sim q_0(z_0)}[\ln q_K(z_K) - \ln p(x,z_K)] \\ &= \mathrm{E}_{z_0\sim q_0(z_0)}[\ln q_0(z_0)] - \mathrm{E}_{z_0\sim q_0(z_0)}[\sum_{k=1}^{K} \ln \left| \det(\frac{\partial f_k}{\partial z_{k-1}}) \right|] - \mathrm{E}_{z_0\sim q_0(z_0)}[\ln p(x,z_K)] \end{aligned}$$

- $\mathrm{E}_{z_0\sim q_0(z_0)}[\ln q_0(z_0)]$: The prior distribution of the latent variable z_0 encoded by the input x .
- $\mathrm{E}_{z_0\sim q_0(z_0)}[\sum_{k=1}^{K} \ln \left| \det(\frac{\partial f_k}{\partial z_{k-1}}) \right|]$: The log determinant of the Jacobian matrix of the transformation f_k.
- $\mathrm{E}_{z_0\sim q_0(z_0)}[\ln p(x,z_K)]$: The log likelihood of the generated image x obtained by the latent variable z_K.

Steps of the algorithm:

1. The encoder outputs μ and σ based on the input x.

2. Sample $z_0 = \mu + \sigma \odot \epsilon$ (using the reparameterization trick)

3. Apply each flow f_k sequentially to obtain z_K and accumulate the log-determinants:

$$z_k = f_k(z_{k-1})$$

$$\ln q_K(z_K) = \ln q_0(z_0) - \sum_{k=1}^{K} \ln \left| \det(\frac{\partial f_k}{\partial z_{k-1}}) \right|$$

4. Use z_K to decode the image x and compute the log likelihood:

$$\ln p(x, z_K)$$

5. Compute the loss function:

$$F(x) = \mathrm{E}_{z_0 \sim q_0(z_0)}[\ln q_0(z_0)] - \mathrm{E}_{z_0 \sim q_0(z_0)}[\sum_{k=1}^{K} \ln \left|\det(\frac{\partial f_k}{\partial z_{k-1}})\right|] - \mathrm{E}_{z_0 \sim q_0(z_0)}[\ln p(x, z_K)]$$

Planar Flow:

Let $u \in \mathrm{R}^d, w \in \mathrm{R}^d, b \in \mathrm{R}$ be the parameters of the planar flow, and h be the $tanh$ function, then we have:

$$z' = f(z) = z + uh(w^T z + b)$$

Let $\psi(z)$ be the Jacobian matrix of $h(w^T z + b)$ (think about the derivative of $tanh$), then we have:

$$\left|\det(\frac{\partial f}{\partial z})\right| = \left|1 + u^T \psi(z)\right|$$

Then this turns $\sum_{k=1}^{K} \ln \left|\det(\frac{\partial f_k}{\partial z_{k-1}})\right|$ into:

$$\sum_{k=1}^{K} \ln \left|\det(\frac{\partial f_k}{\partial z_{k-1}})\right| = \sum_{k=1}^{K} \ln \left|1 + u_k^T \psi(z_{k-1})\right|$$

Code Implementation:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

# Define a single planar flow layer
class PlanarFlow(nn.Module):
    def __init__(self, z_dim):
        super(PlanarFlow, self).__init__()
        self.u = nn.Parameter(torch.randn(1, z_dim))
        self.w = nn.Parameter(torch.randn(1, z_dim))
        self.b = nn.Parameter(torch.randn(1))

    def forward(self, z):
        # z: [batch, z_dim]
        linear = torch.matmul(z, self.w.t()) + self.b  # [batch, 1]
        activation = torch.tanh(linear)                # [batch, 1]
        z_new = z + self.u * activation                # [batch, z_dim]

        # Compute the derivative of tanh(linear): 1 - tanh^2(linear)
        psi = (1 - torch.tanh(linear) ** 2) * self.w     # [batch, z_dim]
        # Determinant: |1 + u^T psi|
        u_psi = torch.matmul(psi, self.u.t())            # [batch, 1]
        log_det_jacobian = torch.log(torch.abs(1 + u_psi) + 1e-8).squeeze(1)  #
```

```
[batch]
        return z_new, log_det_jacobian

# Define the VAE with flows
class VAEWithFlows(nn.Module):
    def __init__(self, input_dim, hidden_dim, z_dim, num_flows):
        super(VAEWithFlows, self).__init__()
        # Encoder network
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc_mu = nn.Linear(hidden_dim, z_dim)
        self.fc_logvar = nn.Linear(hidden_dim, z_dim)
        # Decoder network
        self.fc3 = nn.Linear(z_dim, hidden_dim)
        self.fc4 = nn.Linear(hidden_dim, input_dim)
        # Flow layers
        self.flows = nn.ModuleList([PlanarFlow(z_dim) for _ in range(num_flows)])

    def encode(self, x):
        h = F.relu(self.fc1(x))
        mu = self.fc_mu(h)
        logvar = self.fc_logvar(h)
        return mu, logvar

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std  # z0

    def decode(self, z):
        h = F.relu(self.fc3(z))
        return torch.sigmoid(self.fc4(h))  # For binary data

    def compute_log_normal(self, z, mu, logvar):
        # Compute log probability of z under N(mu, exp(logvar))
        return -0.5 * (logvar + math.log(2 * math.pi) + ((z - mu) ** 2) /
torch.exp(logvar)).sum(dim=1)

    def forward(self, x):
        # 1. Encode x into parameters of q0(z|x)
        mu, logvar = self.encode(x)
        z0 = self.reparameterize(mu, logvar)
        log_qz0 = self.compute_log_normal(z0, mu, logvar)

        # 2. Apply the sequence of flows to obtain z_K
        sum_log_det = 0.
        z = z0
        for flow in self.flows:
            z, log_det = flow(z)
            sum_log_det += log_det
        log_q_zK = log_qz0 - sum_log_det

        # 3. Compute prior log-probability for z_K (assume standard Normal)
        log_pz = self.compute_log_normal(z, torch.zeros_like(z),
torch.zeros_like(z))

        # 4. Decode z_K to reconstruct x
        x_recon = self.decode(z)
        # Reconstruction likelihood (binary cross entropy for binarized data)
        recon_loss = F.binary_cross_entropy(x_recon, x, reduction='sum')
```

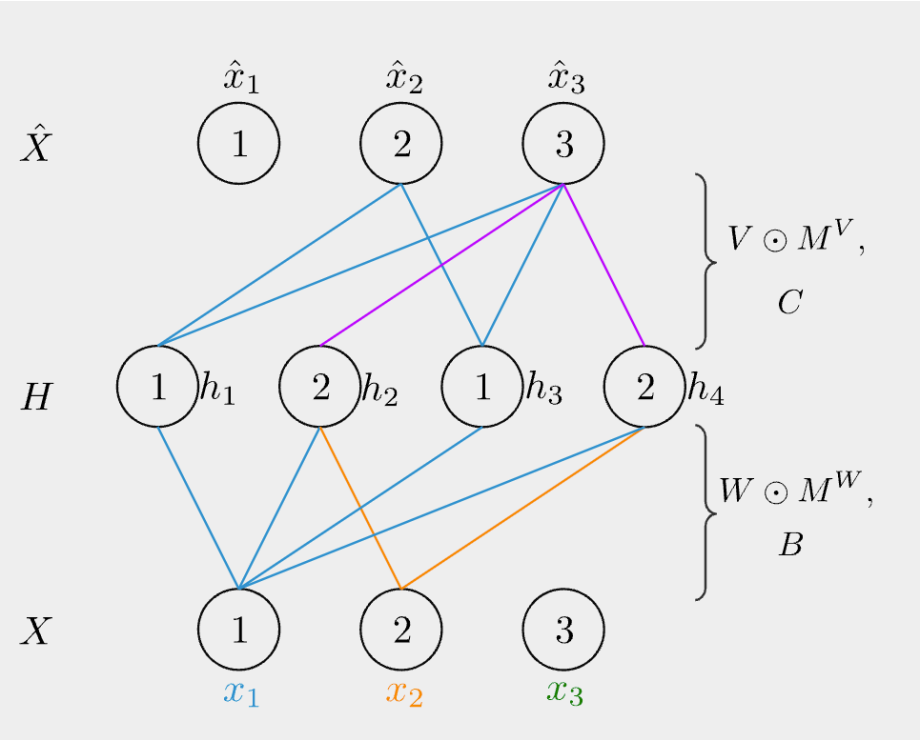
```
        # ELBO: log p(x|z_K) + log p(z_K) - log q(z_K|x)
        elbo = -recon_loss + log_pz - log_q_zK
        return elbo, x_recon, mu, logvar, z

# Example usage:
if __name__ == '__main__':
    # Example hyperparameters
    input_dim = 784  # e.g., flattened MNIST images (28x28)
    hidden_dim = 400
    z_dim = 40
    num_flows = 10

    model = VAEWithFlows(input_dim, hidden_dim, z_dim, num_flows)

    # Dummy batch of data
    x = torch.randn(64, input_dim)  # batch of 64 images
    elbo, x_recon, mu, logvar, z = model(x)

    # We maximize the ELBO (or equivalently, minimize -ELBO)
    loss = -elbo.mean()
    loss.backward()
    print("Loss:", loss.item())
```

MADE: Masked Autoencoder for Distribution Estimation https://arxiv.org/pdf/1502.03509

Novelty:

- Mask Matrix: Applied a mask matrix to the weight matrix of the network to simulate autoregressive depenency of output to the input.

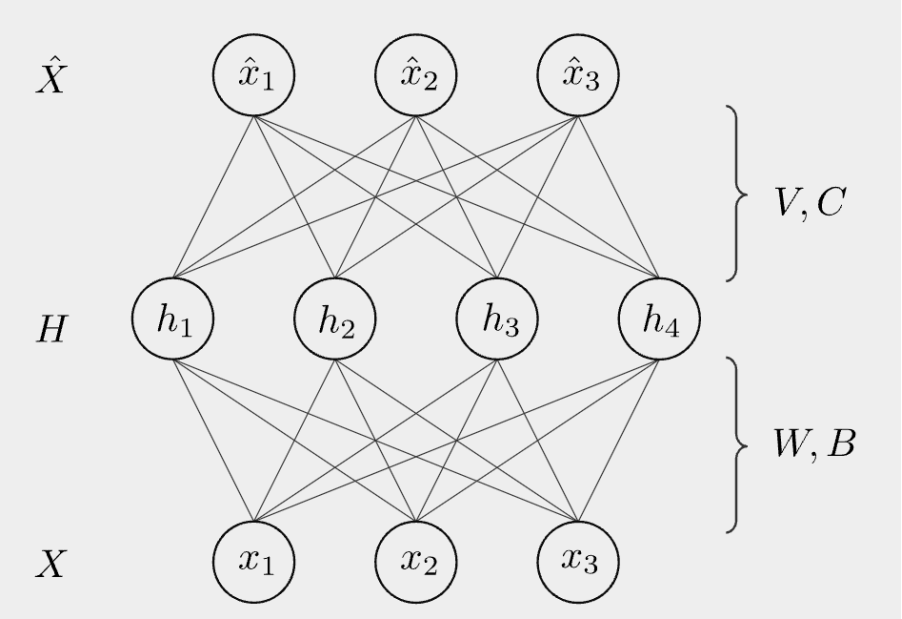
Architecture:



Components:

- W the weight matrix between input X and the hidden layer H
- V the weight matrix between hidden layer H and the output layer $\hat{X}$
- M^W the mask matrix for W
- M^V the mask matrix for V
- $m(k)^l$ the value of the kth neuron in the l th layer, such that for the input layer and output layer $m(k) = k$, while for the hidden layer $m(k)$ is randomly sampled from $\{1, 2, ..., D - 1\}$

Intuition:



For a fully connected network, given a 1-D vector $x = (x_1, x_2, x_3)$ of image, we have $\hat{x}_i$ generated based on all the input pixels x_1, x_2, x_3. Written in probability distribution, we have:

$$p(x_1) = p(x_1|x_1, x_2, x_3)$$
$$p(x_2) = p(x_2|x_1, x_2, x_3)$$
$$p(x_3) = p(x_3|x_1, x_2, x_3)$$

However, the goal of probability distribution we want is:

$$p(x_1, x_2, x_3)$$

which then can be factorized as:

$$p(x_1, x_2, x_3) = p(x_1)p(x_2|x_1)p(x_3|x_1, x_2)$$
$$= \prod_{i=1}^{3} p(x_i|x_{<i})$$

And this is essentially the autoregressive model that pixel x_i is generated based on the previous pixels $x_1, x_2, ..., x_{i-1}$.

To establish the autoregressive dependency, for example, the output $\hat{x}_2$. We want it to be generated based on the input of x_1. What we can do is to first pick the nodes at the hidden layer that we want to use to generate $\hat{x}_2$, and then cut their connections with the input x_2 and x_3. By doing this, we can ensure that the output $\hat{x}_2$ is only generated based on the input x_1.

The authors of the paper proposed a brilliant methodlogy to perform connection cutting.

1. For the input layer, we assign each input x_i a value i
2. For the hidden layer, we assign each neuron a value $m(k)$ between 1 and $D - 1$ (where D is the dimension of the input).
3. Construct the mask matrix M^W that would cut the connection between x_i and h_j if $m(x_i) > m(h_j)$.
4. Construct the mask matrix M^V that would cut the connection between h_j and $\hat{x}_i$ if $m(h_j) >= m(\hat{x}_i)$.

By such construction, we can ensure that the output $\hat{x}_i$ is never dependent on the input $x_i, ...x_D$. In other words, the output $\hat{x}_i$ is only dependent on the input $x_1, x_2, ..., x_{i-1}$, and we can write the probability distribution as $p(x_i|x_1, x_2, ..., x_{i-1})$ which is exactly the autoregressive dependency we want.

Add-Ons:

- Order-agnostic training: The authors proposed to train the model by randomly shuffing the m_k values for the input and output layers.

Limitation:

- The model is not scalable to high-dimensional data.
- For a image input, the pixel is likely to be dependent on the neighboring pixels instead of the previous pixels. Therefore, the model is not able to capture the spatial information.