Index

GANs

VAEs

DDPM

Score matching

- Score matching part 1
- Score matching part 2
- Score matching part 3
- Difference between SMLD and DDPM
- Conditional score matching

SDEs

- Precursor: Stochastic Differential Equations (SDEs)
- Stochastic Differential Equations (SDEs)
- Some example problems
- Anderson's result
- DDPMs as SDE
- SMLD as SDE
- Main takeaway
- DDIM
- DDIM inversion

Sequence2sequence models

- Transformers
- State space models & Mamba

Self supervised learning

- Noise contrastive estimation
- Info-NCE
- Masked reconstruction
- JEPA - Joint Embedding Prediction and Autoencoding

Misc

- Distillation

Whats the difference between generative model and discriminative model ?

Generative Model

Let the training dataset be $data = \{(x_i)\}_{i=1}^{N}$ where $x_i \in \mathbb{R}^d$ called data points. These are iid samples from the true data distribution $P(x)$.

Aim of generative model is to extimate $P(x)$ using the training data and sample new data points from it.

GANs, GMMs etc are a example of generative model. In GANs we use training data to train a neural network to learn the distribution of the data. Once we have learnt this distribution we can sample new data points from it.

Discriminative Model

In this case the data is in a different format $data = \{(x_i, y_i)\}_{i=1}^{N}$ where $x_i \in \mathbb{R}^d$ are the input features and $y_i \in \mathbb{R}^k$ are the corresponding output labels or target variables.

Aim of discriminative model is to estimate the conditional probability $P(y|x)$ using the training data. In simple words, we are given an input x and we want to predict the corresponding output y.

If y can take only discrete values, then the model is called a classification model. If y can take continuous values, then the model is called a regression model.

Difference between generative and discriminative models

- In generative models, we sample new data points from the learned distribution to get a new data point that might look like one of the data points in the training dataset.

- In discriminative models, for a given input x, we sample y from the learned conditional distribution $P(y|x)$.

Then what is conditional generative model ?

In most practical cases, we want the generative model to generate new data point x_i based on some conditioned input y_i.

In this case the the training dataset is $data = \{(x_i, y_i)\}_{i=1}^{N}$ and we want to estimate $P(x|y)$ and sample from it.

KL Divergence - Kullback-Leibler Divergence

KL divergence is a measure of how one probability distribution differes from another probability distribution.

Mathematically, for two probability distributions $P(x)$ and $Q(x)$, the KL divergence from Q to P is defined as:

$D_{KL}(p||q) = \Sigma p(x) * log\left(\frac{p(x)}{q(x)}\right)$ for discrete distributions

$D_{KL}(p||q) = \int p(x) * log\left(\frac{p(x)}{q(x)}\right) dx$ for continuous distributions

where the sum/integral is over all possible events x. And $p(x)$ and $q(x)$ are the probability density functions of distributions $P(x)$ and $Q(x)$ respectively.

Intuition

KL divergence is a measure of how one probability distribution diverges from another. It is a measure of the information lost when Q is used to approximate P.

Properties

- KL divergence is not symmetric: $D_{KL}(P||Q) \neq D_{KL}(Q||P)$
- KL divergence is always non-negative: $D_{KL}(P||Q) \geq 0$
- KL divergence is 0 if and only if P and Q are the same distribution

Usefulness in Machine Learning - Minimizing KL Divergence is Equivalent to Maxmimizing Likelihood

For a typical ML problem, all we have are samples from the true distribution $P(x)$ ie $data = \{(x_i)\}_{i=1}^{N}$ where $x_i \in \mathbb{R}^d$ are iid samples from the true distribution $P(x)$.

We do not know the true distribution $P(x)$ explicitly.

We try our best to estimate the true distribution $P(x)$ by $Q(x;\theta)$ where θ are the parameters of the model.

We need to know how well our model $Q(x;\theta)$ is performing. We can do this by calculating the KL divergence between the true distribution $P(x)$ and the estimated distribution $Q(x;\theta)$.

$$D_{KL}(P||Q) = \int p(x) * log\left(\frac{p(x)}{q(x;\theta)}\right) dx$$

$$D_{KL}(P||Q) = E_{x\sim p(x)}[log\left(\frac{p(x)}{q(x;\theta)}\right)]$$

$$D_{KL}(P||Q) = E_{x\sim p(x)}[log\left(p(x)\right)] - E_{x\sim p(x)}[log\left(q(x;\theta)\right)]$$

We are trying to find the parameters θ^* that minimize the KL divergence between $p(x)$ and $q(x;\theta)$.

$$\theta^* = \underset{\theta}{argmin}\ D_{KL}(p||q(x;\theta))$$

$$\theta^* = \underset{\theta}{argmin}\ E_{x\sim p(x)}[log\left(p(x)\right)] - E_{x\sim p(x)}[log\left(q(x;\theta)\right)]$$

because $E_{x\sim p(x)}[log\left(p(x)\right)]$ does not depend on θ, we can ignore it.

$$\theta^* = \underset{\theta}{argmin}\ -E_{x\sim p(x)}[log\left(q(x;\theta)\right)]$$

$$\theta^* = \underset{\theta}{argmax}\ E_{x\sim p(x)}[log\left(q(x;\theta)\right)]$$

$E_{x\sim p(x)}[log\left(q(x;\theta)\right)]$ is called the **Expected Log Likelihood**,

By the law of large numbers, we can approximate the expected log likelihood by the average log likelihood of the data:

$$E_{x\sim p(x)}[log(q(x;\theta))] \approx \frac{1}{N}\sum_{i=1}^{N} log(q(x_i;\theta))$$

Therefore, our optimization problem becomes:

$$\theta^* = \underset{\theta}{argmax}\ \frac{1}{N}\sum_{i=1}^{N} log(q(x_i;\theta))$$

This is equivalent to maximizing the log likelihood of the data under the model $q(x;\theta)$.

$$\theta^* = \underset{\theta}{argmax}\ \frac{1}{N}\sum_{i=1}^{N} log(q(x_i;\theta))$$

hence θ is also called the **maximum log likelihood estimate**.

F-Divergence

F-divergence is a generalized measure of difference between two probability distributions. For probability distributions P and Q over a event space X, the F-divergence is defined as:

$$D_f(P||Q) = \int_X q(x) f(\frac{p(x)}{q(x)}) dx$$

Where:

- **Generator function** $f : R^+ \to R$ is a lower semi-continuous convex function with $f(1) = 0$. The F in F-divergence comes from generator "f"unction.
- $p(x)$ and $q(x)$ are the probability density functions of P and Q respectively

Some examples of F-divergences include:

- Kullback-Leibler divergence:
  - $D_{KL}(P||Q) = \int_X p(x) log(\frac{p(x)}{q(x)}) dx$
  - when $f(x) = xlog(x)$
  - Note that KL divergence is just a special case of F-divergence
- Reverse Kullback-Leibler divergence:
  - $D_{KL}(Q||P) = \int_X q(x) log(\frac{q(x)}{p(x)}) dx$
  - when $f(x) = -log(x)$
- Jensen-Shannon divergence:
  - $D_{JS}(P||Q) = \frac{1}{2} \int_X p(x) \log\left(\frac{2p(x)}{p(x)+q(x)}\right) + q(x) \log\left(\frac{2q(x)}{p(x)+q(x)}\right) dx$
  - when $f(x) = -(x+1)\log\left(\frac{1+x}{2}\right) + x\log(x)$

Properties of F-divergences

1. F-divergence is always non-negative, i.e., $D_f(P||Q) \geq 0$

Proof:
By Jensen's inequality, since f is convex and $p(x)$ and $q(x)$ are probability densities:

$$\int_X q(x) f(\frac{p(x)}{q(x)}) dx \geq f(\int_X q(x)(\frac{p(x)}{q(x)}) dx)$$

$$\int_X q(x) f(\frac{p(x)}{q(x)}) dx \geq f(\int_X p(x) dx)$$

$$\int_X q(x) f(\frac{p(x)}{q(x)}) dx \geq f(1) = 0$$

$$D_f(P||Q) \geq 0$$

Hence, F-divergence is always non-negative.

2. F-divergence is 0 if and only if P = Q

Proof:
We'll prove both directions:

- If $P = Q$, then $D_f(P||Q) = 0$, and
- If $D_f(P||Q) = 0$, then $P = Q$.

First, if $P = Q$, then $D_f(P||Q) = 0$:

When $P = Q$ then $p(x) = q(x)$ for all x. Hence:

$$D_f(P||Q) = \int_X q(x) f(\frac{p(x)}{q(x)}) dx$$

$$D_f(P||Q) = \int_X q(x) f(\frac{p(x)}{p(x)}) dx$$

$$D_f(P||Q) = \int_X q(x) f(1) dx$$

$$D_f(P||Q) = f(1) \int_X q(x) dx$$

$$D_f(P||Q) = f(1) * 1 = 0$$

Since $f(1) = 0$ by definition of F-divergence.

Second, if $D_f(P||Q) = 0$, then $P = Q$:

Assume $D_f(P||Q) = 0$. This means:

$$\int_X q(x) f(\frac{p(x)}{q(x)}) dx = 0$$

Since $q(x) \geq 0$ for all x, and f is convex, the integral can only be zero if:

$$f(\frac{p(x)}{q(x)}) = 0 \text{ for all } x \in X$$

Given that $f(1) = 0$ and f is convex, the only way $f(\frac{p(x)}{q(x)}) = 0$ is if:

$$\frac{p(x)}{q(x)} = 1 \text{ for all } x \in X$$

This means that:

$$p(x) = q(x) \text{ for all } x \in X$$

Thus

$$P = Q$$

Therefore, $D_f(P||Q) = 0$ if and only if $P = Q$.

3. F-divergence is not symmetric, i.e., $D_f(P||Q) \neq D_f(Q||P)$ in general

Proof:
Consider the following counterexample:

Let $X = \{0, 1\}$, and let P and Q be the following probability distributions:

$P(0) = 0.3, P(1) = 0.7$
$Q(0) = 0.7, Q(1) = 0.3$

Let $f(x) = x \log(x)$ (KL divergence).

Then:

$D_f(P||Q) = \int_X p(x) f(\frac{p(x)}{q(x)}) dx$
$= 0.3 f(\frac{0.3}{0.7}) + 0.7 f(\frac{0.7}{0.3})$
$= 0.3(\frac{0.3}{0.7} \log(\frac{0.3}{0.7})) + 0.7(\frac{0.7}{0.3} \log(\frac{0.7}{0.3}))$
$\approx -0.1209 + 0.9650 = 0.8441$

$D_f(Q||P) = \int_X q(x) f(\frac{q(x)}{p(x)}) dx$
$= 0.7 f(\frac{0.7}{0.3}) + 0.3 f(\frac{0.3}{0.7})$
$= 0.7(\frac{0.7}{0.3} \log(\frac{0.7}{0.3})) + 0.3(\frac{0.3}{0.7} \log(\frac{0.3}{0.7}))$
$\approx 1.3786 - 0.0518 = 1.3268$

In this case, $D_f(P||Q) \approx 0.8441 \neq D_f(Q||P) \approx 1.3268$.

Therefore, F-divergence is not symmetric in general.

Convex conjugate (also known as the Fenchel conjugate or the Legendre transform)

Definition

Let $f : R^n \to R$ be a convex function. The convex conjugate of f, denoted as f^*, is defined as:

$$f^*(y) = \sup_{x \in \mathrm{dom} f} (y^T x - f(x))$$

Where:

- $y \in R^n$ is the variable
- $\mathrm{dom} f$ is the domain of f
- $\sup$ denotes the supremum (least upper bound)

Properties

1. Convexity: $f^{**}(x)$ is also convex

Proof: Let $y_1, y_2 \in \mathbb{R}^n$ and $\lambda \in [0, 1]$. We need to show that:

$$f^*(\lambda y_1 + (1-\lambda) y_2) \leq \lambda f^*(y_1) + (1-\lambda) f^*(y_2)$$

1. The definition states: $f^*(y) = \sup_{x \in \mathrm{dom} f}(y^T x - f(x))$
2. Here, our y is $\lambda y_1 + (1-\lambda) y_2$
3. Substituting this into the definition gives us:

$$f^*(\lambda y_1 + (1-\lambda) y_2) = \sup_x \{(\lambda y_1 + (1-\lambda) y_2)^T x - f(x)\}$$

Next step will distribute the transpose operation and separate the terms:

$$f^*(\lambda y_1 + (1-\lambda) y_2) = \sup_x \{\lambda y_1^T x + (1-\lambda) y_2^T x - f(x)\}$$

$$f^*(\lambda y_1 + (1-\lambda)y_2) = \sup_x \{\lambda(y_1^T x - f(x)) + (1-\lambda)(y_2^T x - f(x))\}$$

The supremum of a sum is less than or equal to the sum of the suprema:

$$f^*(\lambda y_1 + (1-\lambda)y_2) \leq \sup_x \lambda\{y_1^T x - f(x)\} + \sup_x (1-\lambda)\{y_2^T x - f(x)\}$$

$$f^*(\lambda y_1 + (1-\lambda)y_2) \leq \lambda \sup_x \{y_1^T x - f(x)\} + (1-\lambda) \sup_x \{y_2^T x - f(x)\}$$

$$f^*(\lambda y_1 + (1-\lambda)y_2) \leq \lambda f^*(y_1) + (1-\lambda) f^*(y_2)$$

Thus, f^* is convex.

2. Conjugate of conjugate: $f^{**}(y) = f(y)$

Proof:

We know that:

$$f^*(y) = \sup_{x \in \mathrm{dom} f} (y^T x - f(x))$$

Therefore:

$$f^{**}(y) = \sup_x \{y^T x - f^*(x)\}$$

$$f^{**}(y) = \sup_x \{y^T x - \sup_{z \in \mathrm{dom} f} (x^T z - f(z))\}$$

$$f^{**}(y) = \sup_x \inf_{z \in \mathrm{dom} f} \{y^T x - (x^T z - f(z))\}$$

$$f^{**}(y) = \sup_x \inf_{z \in \mathrm{dom} f} \{y^T x - x^T z + f(z)\}$$

$$f^{**}(y) = \sup_x \inf_{z \in \mathrm{dom} f} \{x^T (y - z) + f(z)\}$$

Now, we can apply the minimax theorem, which allows us to swap the order of sup and inf:

$$f^{**}(y) = \inf_{z \in \mathrm{dom} f} \sup_x \{x^T (y - z) + f(z)\}$$

The inner supremum is unbounded unless $y = z$, in which case it equals $f(z)$.
Therefore:

$$f^{**}(y) = \inf_{z \in \mathrm{dom} f} \begin{cases} f(z) & \text{if } y = z \\ \infty & \text{otherwise} \end{cases}$$

This simplifies to $f^{**}(y) = f(y)$ because:

1. When $y \neq z$, the infimum over z where $f^{**}(y) = \infty$ cannot be the actual infimum, since we have at least one finite value when $y = z$

2. When $y = z$, we get $f^{**}(y) = f(z) = f(y)$

3. Therefore, the infimum must occur at $y = z$, giving us $f^{**}(y) = f(y)$

Thus, we have shown that $f^{**}(y) = f(y)$, completing the proof.

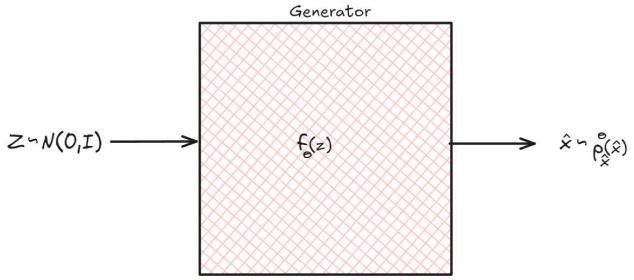
Generative Adversarial Networks (GANs)

Recall that generative models try to learn the data distribution P_{data}.

GANs try to learn P_{data} by approximating it by $P_{generator}$ by minimizing F-divergence $D_f(P_{data}||P_{generator})$.

Derivation of GANs

Let θ be the parameters of the generator such that it maps a sample $z \sim N(0, I)$ to a sample $x \sim p_{generator}$.



Then, the optimal parameters for the generator can be expressed as:

$$\theta^* = \underset{\theta}{\operatorname{argmin}} D_f(P_{data}||P_{generator})$$

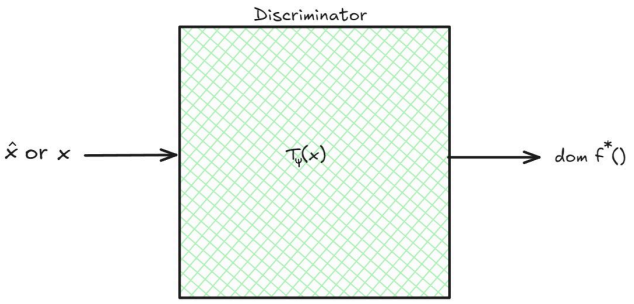
$$\theta^* = \underset{\theta}{\operatorname{argmin}} \int_x p_{generator}(x) f\left(\frac{p_{data}(x)}{p_{generator}(x)}\right) dx$$

We know that $f^*(y) = \sup_{x \in \operatorname{dom} f} (y^T x - f(x))$. Therefore, we can rewrite our optimization problem as:

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \int_x p_{generator}(x) \sup_{t \in \operatorname{dom} f^*} \left[t \frac{p_{data}(x)}{p_{generator}(x)} - f^*(t) \right] dx$$

where $f\left(\frac{p_{data}(x)}{p_{generator}(x)}\right) = \sup_{t \in \operatorname{dom} f^*} \left[t \frac{p_{data}(x)}{p_{generator}(x)} - f^*(t) \right]$ as f is the convex conjugate of f^*.

Let's model $t \in \operatorname{dom} f^*$ by a neural network with parameters ϕ. We'll denote this neural network as $T_\phi(x)$.



$$\theta^* = \underset{\theta}{\operatorname{argmin}} \int_x p_{generator}(x) \sup_{\phi} \left[T_\phi(x) \frac{p_{data}(x)}{p_{generator}(x)} - f^*(T_\phi(x)) \right] dx$$

Now see that:

$$\underset{\theta}{\operatorname{argmin}} \int_x p_{generator}(x) \sup_{\phi} \left[T_\phi(x) \frac{p_{data}(x)}{p_{generator}(x)} - f^*(T_\phi(x)) \right] dx \leq \underset{\theta}{\operatorname{argmin}} \sup_{\phi} \int_x p_{generator}(x) \left[T_\phi(x) \frac{p_{data}(x)}{p_{generator}(x)} - f^*(T_\phi(x)) \right] dx$$

We will replace $\underset{\theta}{\operatorname{argmin}} \int_x p_{generator}(x) \sup_{\phi} \left[T_\phi(x) \frac{p_{data}(x)}{p_{generator}(x)} - f^*(T_\phi(x))\right] dx$ with $\underset{\theta}{\operatorname{argminsup}}_{\phi} \int_x p_{generator}(x) \left[T_\phi(x) \frac{p_{data}(x)}{p_{generator}(x)} - f^*(T_\phi(x))\right] dx$ in the expression of θ^*. Hence the θ^* we are optimising is minimum lower bound of original θ^* which is one of the draw backs of GANs. For not changing notations we will use the same notation θ^* for the new θ^* we are optimising.

Hence, by swapping the order of supremum and integral we get:

$$\theta^* = \underset{\theta}{\operatorname{argminsup}}_{\phi} \int_x p_{generator}(x) \left[T_\phi(x) \frac{p_{data}(x)}{p_{generator}(x)} - f^*(T_\phi(x))\right] dx \qquad (eqn-1)$$

$$\theta^* = \underset{\theta}{\operatorname{argminsup}}_{\phi} \int_x \left[T_\phi(x) p_{data}(x) - p_{generator}(x) f^*(T_\phi(x))\right] dx$$

$$\theta^* = \underset{\theta}{\operatorname{argminsup}}_{\phi} \left[\int_x T_\phi(x) p_{data}(x) dx - \int_x p_{generator}(x) f^*(T_\phi(x)) dx\right]$$

The first integral is an expectation over the data distribution, and the second is an expectation over the generator distribution:

$$\theta^* = \underset{\theta}{\operatorname{argminsup}}_{\phi} \left[\mathbb{E}_{x \sim p_{data}}[T_\phi(x)] - \mathbb{E}_{x \sim p_{generator}}[f^*(T_\phi(x))]\right]$$

This simplified form is the core of the GAN objective function. The generator (parameterized by θ) tries to minimize this expression, while the discriminator also known as critic (parameterized by ϕ) tries to maximize it.

Why is it called adversarial network?

The term “adversarial” in GANs refers to the adversarial relationship between the generator and the discriminator. In the context of GANs, the generator and the discriminator play a two-player min-max game. The generator tries to minimize the objective function while the discriminator tries to maximize it. This adversarial relationship is what gives rise to the term “adversarial network.”

Naive GANs

A Naive GAN (Generative Adversarial Network) refers to the basic or “vanilla” version of GANs, introduced by Ian Goodfellow in 2014.

Derivation

Recall the genereral expression for the GANs that we got by minimizing F-divergence:

$$\theta^* = \arg\min_{\theta} \sup_{\phi} \left[\mathbb{E}_{x \sim p_{data}}[T_\phi(x)] - \mathbb{E}_{x \sim p_{generator}}[f^*(T_\phi(x))] \right]$$

where:

- θ: Parameters of the generator network
- ϕ: Parameters of the discriminator (critic) network
- θ^*: Optimal parameters for the generator
- $\arg\min_\theta$: Argument that minimizes the expression with respect to θ
- $\sup_\phi$: Supremum (least upper bound) with respect to ϕ
- $\mathbb{E}_{x \sim p_{data}}$: Expectation over the real data distribution
- $\mathbb{E}_{x \sim p_{generator}}$: Expectation over the generator’s distribution
- $T_\phi(x)$: The discriminator function, parameterized by ϕ
- f^*: The convex conjugate of the function f used in the F-divergence
- p_{data}: The true data distribution
- $p_{generator}$: The distribution of the generated data

Write $T_\phi(x)$ as composite function $T_\phi(x) = \sigma(V_\phi(x))$, where σ is the sigmoid function Substitute this in the expression for GANs:

$$\theta^* = \arg\min_{\theta} \sup_{\phi} \left[\mathbb{E}_{x \sim p_{data}}[\sigma(V_\phi(x))] - \mathbb{E}_{x \sim p_{generator}}[f^*(\sigma(V_\phi(x)))] \right]$$

For the naive GAN, we use the Jensen-Shannon divergence, which corresponds to:

$$f(t) = t \log t - (t+1) \log(t+1)$$

The convex conjugate of this function is:

$$f^*(t) = -\log(1 - e^t)$$

Substituting this into our expression:

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sup_{\phi} \left[\mathbb{E}_{x \sim p_{data}}[\sigma(V_\phi(x))] - \mathbb{E}_{x \sim p_{generator}}[-\log(1 - e^{\sigma(V_\phi(x))})] \right]$$

Substitute $\sigma(V_\phi(x)) = \frac{1}{1+e^{-V_\phi(x)}}$, we get:

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sup_{\phi} \left[\mathbb{E}_{x \sim p_{data}}[\frac{1}{1+e^{-V_\phi(x)}}] - \mathbb{E}_{x \sim p_{generator}}[-\log(1 - \frac{1}{1+e^{-V_\phi(x)}})] \right]$$

Simplify the expression using:

$$1 - \frac{1}{1+e^{-V_\phi(x)}} = \frac{e^{-V_\phi(x)}}{1+e^{-V_\phi(x)}}$$

This gives us:

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sup_{\phi} \left[\mathbb{E}_{x \sim p_{data}}[\frac{1}{1+e^{-V_\phi(x)}}] - \mathbb{E}_{x \sim p_{generator}}[-\log(\frac{e^{-V_\phi(x)}}{1+e^{-V_\phi(x)}})] \right]$$

$$\theta^* = \mathbb{E}_{x \sim p_{data}}[\frac{1}{1+e^{-V_\phi(x)}}] - \mathbb{E}_{x \sim p_{generator}}[-(-V_\phi(x) - \log(1+e^{-V_\phi(x)}))]$$

For the second expectation term:

$$-(-V_\phi(x) - \log(1+e^{-V_\phi(x)}))$$

$$= V_\phi(x) + \log(1+e^{-V_\phi(x)})$$

$$= \log(e^{V_\phi(x)}) + \log(1+e^{-V_\phi(x)})$$

$$= \log(e^{V_\phi(x)}(1+e^{-V_\phi(x)}))$$

$$= \log(e^{V_\phi(x)}+1)$$

$$= \log(1+e^{V_\phi(x)})$$

$$\theta^* = \mathbb{E}_{x \sim p_{data}}[\frac{1}{1 + e^{-V_\phi(x)}}] - \mathbb{E}_{x \sim p_{generator}}[\log(1 + e^{-V_\phi(x)})]$$

Using the properties of logarithms and the fact that $\frac{e^{-V_\phi(x)}}{1+e^{-V_\phi(x)}} = 1 - \frac{1}{1+e^{-V_\phi(x)}}$, we can rewrite:

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \underset{\phi}{\operatorname{sup}} \left[\mathbb{E}_{x \sim p_{data}}[-\log(1 + e^{-V_\phi(x)})] - \mathbb{E}_{x \sim p_{generator}}[\log(1 + e^{V_\phi(x)})]\right]$$

This is equivalent to:

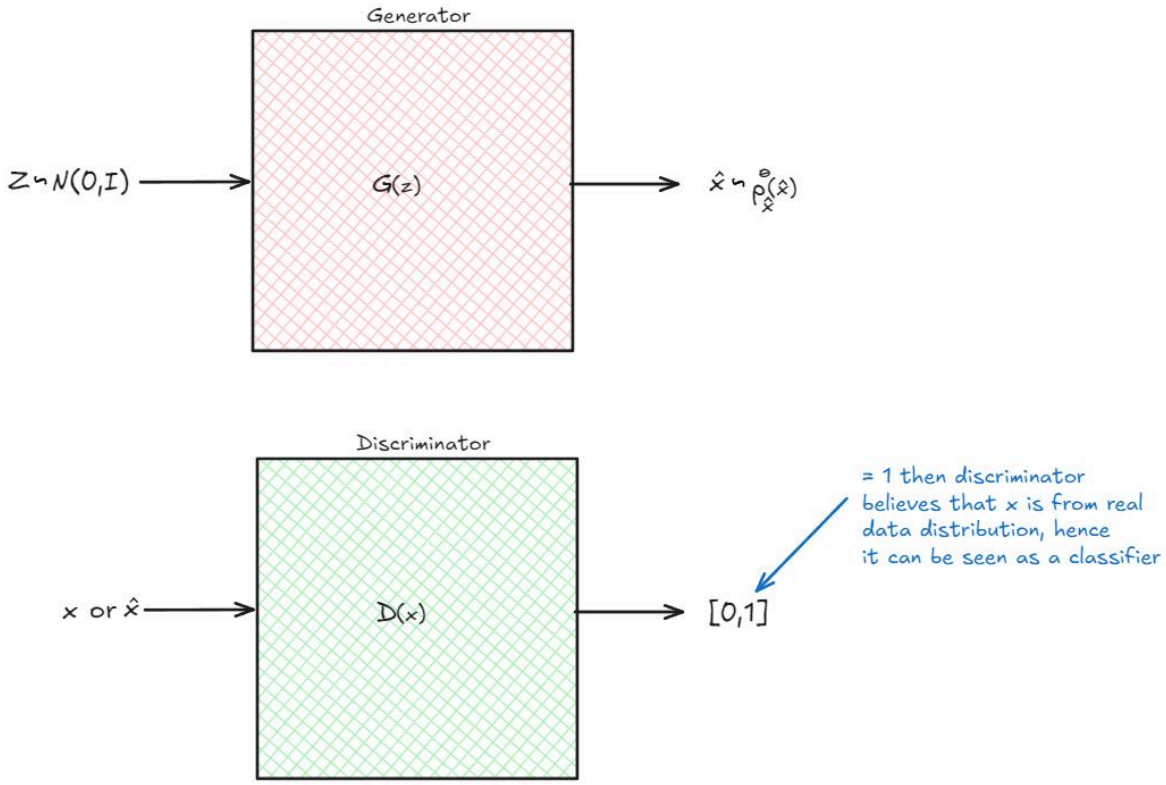
$$\theta^* = \underset{\theta}{\operatorname{argmin}} \underset{\phi}{\operatorname{sup}} \left[\mathbb{E}_{x \sim p_{data}}[\log(\frac{1}{1 + e^{-V_\phi(x)}})] + \mathbb{E}_{x \sim p_{generator}}[\log(\frac{e^{-V_\phi(x)}}{1 + e^{-V_\phi(x)}})]\right]$$

Let's define the discriminator function $D_\phi(x) = \frac{1}{1+e^{-V_\phi(x)}}$, which maps inputs to probabilities in [0,1]. Then our objective function takes the elegant form:

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \underset{\phi}{\operatorname{sup}} \left[\mathbb{E}_{x \sim p_{data}}[\log(D_\phi(x))] + \mathbb{E}_{x \sim p_{generator}}[\log(1 - D_\phi(x))]\right]$$

This is the standard form of the GAN objective function as presented in the original paper by Goodfellow et al.

Interpretation



In the naive GAN framework, we can interpret the roles of the discriminator and generator as follows:

1. Discriminator (D):

- The discriminator aims to maximize $\mathbb{E}_{x \sim p_{data}}[\log(D_\phi(x))] + \mathbb{E}_{x \sim p_{generator}}[\log(1 - D_\phi(x))]$.
- Because discriminator is of the form $D_\phi(x) = \frac{1}{1+e^{-V_\phi(x)}}$, it outputs a probability between 0 and 1, where:
- $D_\phi(x) \approx 1$ indicates the discriminator believes x is from the real data distribution
- $D_\phi(x) \approx 0$ indicates the discriminator believes x is from the generator (fake data)
- This pushes the discriminator to correctly classify real and fake samples.

2. Generator (G):

- The generator aims to minimize $\mathbb{E}_{x \sim p_{data}}[\log(D_\phi(x))] + \mathbb{E}_{x \sim p_{generator}}[\log(1 - D_\phi(x))]$. The first term is the expected log-likelihood of the discriminator

classifying a real sample as real, and the second term is the expected log-likelihood of the discriminator classifying a generated sample as fake. The first term is a constant with respect to the generator, so the generator aims to minimize the second term. Hence,generator tries minimise $\mathbb{E}_{x\sim p_{generator}}[\log(1-D_\phi(x))]$,

- If the generator produces samples that the discriminator classifies as real, $D_\phi(x) \approx 1$. Then $\log(1-D_\phi(x)) \approx 0$.

Note that this kind of neat interpretation of the roles of the discriminator and generator is not possible for other choice of f and f^*.

Training

The training process for GANs involves alternating between training the discriminator and the generator. Here's a step-by-step explanation of the training process:

1. Initialize the Generator (G) and Discriminator (D) networks.

2. For each training iteration:

a. Train the Discriminator:

- Generate a batch of fake samples using the generator: $x_{fake} = G(z)$, where z is random noise.

- Sample a batch of real data: x_{real}.

- Calculate the discriminator's loss:

$$L_D = -[\mathbb{E}_{x\sim p_{data}}[\log(D(x))] + \mathbb{E}_{x\sim p_g}[\log(1-D(G(z)))]]$$

- Update the discriminator's parameters using gradient descent to minimize L_D.

b. Train the Generator:

- Generate a new batch of fake samples: $x_{fake} = G(z)$

- Calculate the generator's loss:

$$L_G = -\mathbb{E}_{z\sim p_g}[\log(1-D(G(z)))]$$

- In practice, to avoid vanishing gradients early in training, the generator's loss is often implemented as:

$$L_G = -\mathbb{E}_{z \sim p_g}[\log(D(G(z)))]$$

- Update the generator's parameters using gradient descent to minimize L_G.

The training process continues until the generator produces samples that are indistinguishable from real data, or until a predetermined number of iterations is reached.

Wasserstein metric

Definition

We can use Wasserstein metric in place of F-divergence to define a distance between two distributions P and Q both defined over the same space X.

The Wasserstein distance $W_p(P, Q)$ of order p is defined as:

$$W_p(P, Q) = \left(\inf_{\gamma \in \Gamma(P,Q)} \mathbb{E}_{(x,y)\sim\gamma}[d(x, y)^p] \right)^{1/p}$$

where:

- $\Gamma(P, Q)$ is the set of all possible joint distributions of (x, y) where $x, y \in X$ with marginals $x \sim P$ and $y \sim Q$ (also called couplings)
- $\gamma(x, y)$ is a particular joint distribution chosen out of all possible joint distributions in $\Gamma(P, Q)$ that minimizes $\mathbb{E}_{(x,y)\sim\gamma}[d(x, y)^p]$
- $d(x, y)$ is the metric (distance) between points x and y in the space X
- $p \geq 1$ is the order of the Wasserstein distance. Common choices are $p = 1$ (first-order Wasserstein distance) and $p = 2$ (second-order Wasserstein distance).

Properties of Wasserstein metric

1. When $p = 1$, the Wasserstein distance is equivalent to the Earth Mover's Distance (EMD), because it can be interpreted as the minimum amount of work required to transform one distribution into another.
2. When $p = 2$, the Wasserstein distance is equivalent to the squared Euclidean distance.
3. The Wasserstein distance is always finite, unlike the F-divergence, which can be infinite.
4. The Wasserstein distance is a true metric, meaning it satisfies the triangle inequality.
5. When KL divergence is 0 then Wasserstein distance is 0, but not vice versa.

Proof for 5th property - KL divergence is 0 then Wasserstein distance is 0

To prove that when the KL divergence is 0, the Wasserstein distance is also 0, we start with the definitions of both metrics.

KL Divergence: The Kullback-Leibler (KL) divergence between two distributions P and Q is defined as:

$$D_{KL}(P \parallel Q) = \int_X p(x) \log \left(\frac{p(x)}{q(x)} \right) dx$$

where $p(x)$ and $q(x)$ are the probability density functions of P and Q respectively.

Wasserstein Distance: The Wasserstein distance of order 1 between two distributions P and Q is defined as:

$$W_1(P, Q) = \inf_{\gamma \in \Gamma(P, Q)} \mathbb{E}_{(x,y)\sim\gamma}[d(x, y)]$$

where $\Gamma(P, Q)$ is the set of all possible joint distributions with marginals P and Q.

Now, if $D_{KL}(P \parallel Q) = 0$, it implies that P and Q are identical almost everywhere, i.e., $p(x) = q(x)$ for almost all $x \in X$. This is because the KL divergence is zero if and only if the two distributions are the same.

Since P and Q are identical, the optimal coupling γ in the definition of the Wasserstein distance will be such that $x = y$ almost surely. Therefore, the expected distance $d(x, y)$ will be zero.

Hence, $W_1(P, Q) = 0$.

This completes the proof that when the KL divergence is 0, the Wasserstein distance is also 0. To prove that the vice versa is not true, i.e., when the Wasserstein distance is 0, the KL divergence is not necessarily 0, we can consider the following example:

Consider two distributions P and Q defined over the real line $\mathbb{R}$.

Let P be a Dirac delta distribution centered at 0:

$$P(x) = \delta(x)$$

Let Q be a uniform distribution over the interval $[-\epsilon, \epsilon]$ for some small $\epsilon > 0$:

$$Q(x) = \begin{cases} \frac{1}{2\epsilon} & \text{if } x \in [-\epsilon, \epsilon] \\ 0 & \text{otherwise} \end{cases}$$

Now, let's compute the Wasserstein distance $W_1(P, Q)$ and the KL divergence $D_{KL}(P \parallel Q)$.

Wasserstein Distance: The Wasserstein distance of order 1 between P and Q is given by:

$$W_1(P, Q) = \inf_{\gamma \in \Gamma(P, Q)} \mathbb{E}_{(x,y)\sim\gamma}[|x - y|]$$

Since P is a Dirac delta distribution centered at 0, the optimal coupling γ will pair the point mass at 0 in P with the uniform distribution over $[-\epsilon, \epsilon]$ in Q. The expected distance is:

$$W_1(P, Q) = \int_{-\epsilon}^{\epsilon} |0 - x| \cdot \frac{1}{2\epsilon} \, dx = \int_{-\epsilon}^{\epsilon} \frac{|x|}{2\epsilon} \, dx$$

Splitting the integral at 0, we get:

$$W_1(P, Q) = \int_0^{\epsilon} \frac{x}{2\epsilon} \, dx + \int_{-\epsilon}^{0} \frac{-x}{2\epsilon} \, dx = \frac{1}{2\epsilon} \left(\int_0^{\epsilon} x \, dx + \int_0^{\epsilon} x \, dx \right) = \frac{1}{2\epsilon} \left(\frac{\epsilon^2}{2} + \frac{\epsilon^2}{2} \right) = \frac{\epsilon}{2}$$

As $\epsilon \to 0$, $W_1(P, Q) \to 0$.

KL Divergence: The KL divergence between P and Q is given by:

$$D_{KL}(P \parallel Q) = \int_X p(x) \log \left(\frac{p(x)}{q(x)} \right) dx$$

cannot be directly applied in this case because the Dirac delta measure is singular with respect to the Lebesgue measure (the measure used for the uniform distribution Q).

When two probability measures are singular (i.e., they are concentrated on disjoint sets), the KL divergence between them is infinite by definition. In our case:

- P (Dirac delta) is concentrated at a single point {0}
- Q (uniform distribution) is absolutely continuous with respect to Lebesgue measure
- Therefore, P and Q are singular measures

Thus, $D_{KL}(P \parallel Q) = \infty$.

Therefore, even though the Wasserstein distance $W_1(P, Q)$ can be made arbitrarily small by choosing a small ϵ, the KL divergence $D_{KL}(P \parallel Q)$ is infinite.

This example demonstrates that the vice versa is not true: when the Wasserstein distance is 0, the KL divergence is not necessarily 0.

Key Takeaway: The Wasserstein distance captures the geometric difference between distributions, while KL divergence requires absolute continuity between the measures. When

measures are singular, as in this case with a Dirac delta and a continuous distribution, the KL divergence is infinite regardless of how geometrically close the distributions might be.

Wasserstein GANs (WGANs)

Perfect Discriminator theorem

The Perfect Discriminator theorem states that if two distributions p_x and q_x have support on two disjoint subsets M and P respectively, there always exists a discriminator $D^* : x \to [0, 1]$ that has accuracy 1:

$$\forall x \in M \cup P, D^*(x) = \begin{cases} 1 & \text{if } x \in M \\ 0 & \text{if } x \in P \end{cases}$$

This discriminator achieves perfect classification, as it correctly identifies the origin of every sample with 100% accuracy.

The Perfect Discriminator theorem has significant implications for GANs:

1. It can lead to training instability and convergence issues, as the generator receives no useful gradient information when the discriminator achieves perfect accuracy.
2. It may result in mode collapse, where the generator produces a limited set of samples rather than capturing the full diversity of the real data distribution.
3. This theorem motivates the use of regularization techniques and alternative GAN formulations to ensure meaningful learning can occur.

Wasserstein GANs (WGANs)

Recall in the original formulation of GANs, we have:

$$\theta^* = \arg\min_\theta D_f(P_{data} \parallel P_{generator})$$

In Wasserstein GANs, we replace the F-divergence with the Wasserstein distance of order 1:

$$\theta^* = \arg\min_\theta W_1(P_{data} \parallel P_{generator})$$

$$\theta^* = \arg\min_\theta \inf_{\gamma \in \Gamma(P_{data}, P_{generator})} \mathbb{E}_{(x,y)\sim\gamma}[d(x, y)]$$

use Kantorovich-Rubinstein duality to get (*derivation skipped*):

$$\theta^* = \arg\min_{\theta} \sup_{f \in \text{Lip}_1} \mathbb{E}_{x \sim P_{data}}[f(x)] - \mathbb{E}_{x \sim P_{generator}}[f(x)] \qquad (eqn-1)$$

where f is a 1-Lipschitz function.

What is k - Lipschitz?

K lipschitz is defined as:

$$|f(x) - f(y)| \leq K|x - y|$$

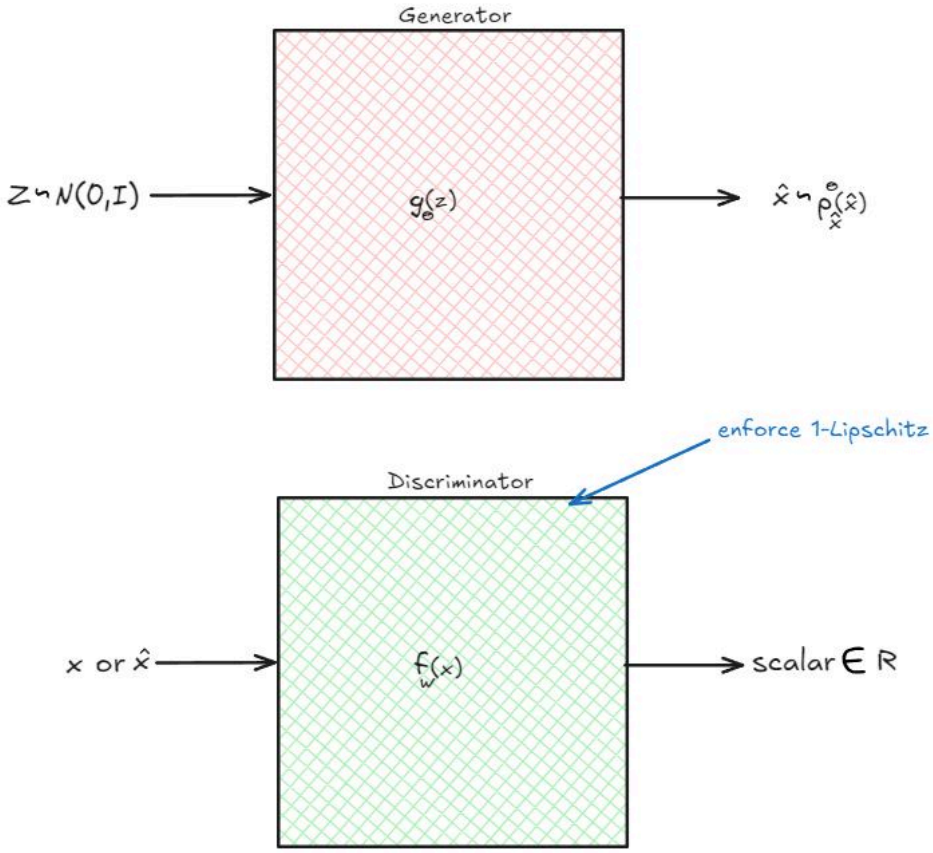
Hence 1-Lipschitz is defined as:

$$|f(x) - f(y)| \leq 1|x - y|$$

How to enforce 1-Lipschitz constraint?

We use neural networks with parameter ω to model f, hence we can rewrite (1) as:

$$\theta^* = \arg\min_{\theta} \max_{\omega} \mathbb{E}_{x \sim P_{data}}[f(x)] - \mathbb{E}_{x \sim P_{generator}}[f(x)]$$



We enforce the 1-Lipschitz constraint on f ie discriminator network by:

- Gradient clipping: $||\nabla f|| \leq 1$
- Weight normalization: $||\omega|| = 1$

Frechet Inception Distance (FID)

Why generative models cannot be evaluated like discriminative models

Usually, we train a deep learning model on training data and then evaluate it on test data. However, for generative models, we cannot directly compare the generated samples to the test data in the same way we evaluate discriminative models. This is because:

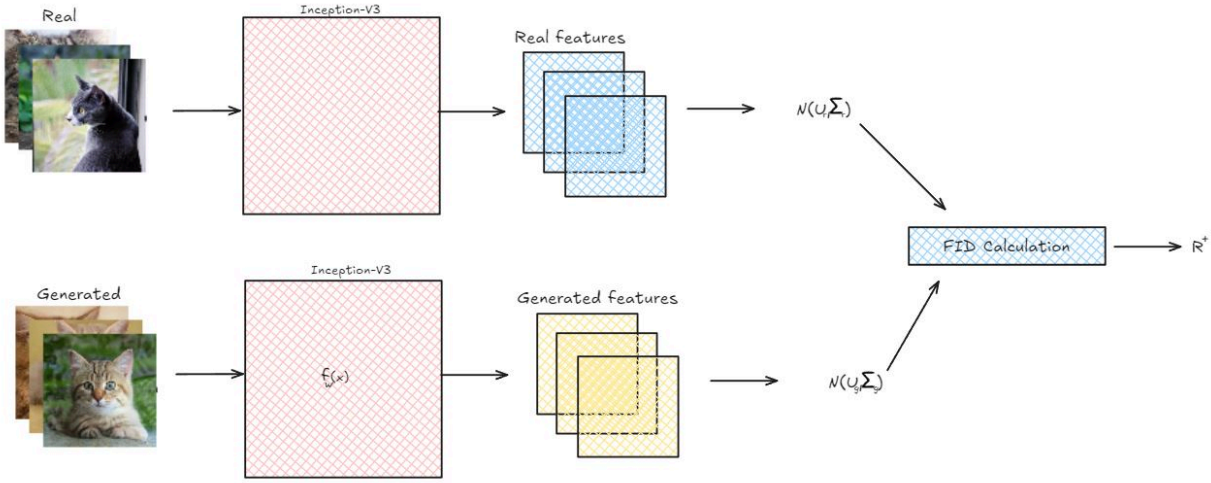
1. Generative models produce new, unique samples rather than predictions on existing data.
2. There's no one-to-one correspondence between generated samples and real data points.
3. We need to assess the quality, diversity, and realism of the generated samples, which requires different metrics.
4. Traditional evaluation metrics like accuracy or mean squared error are not applicable to generative tasks.

Therefore, we need specialized methods to evaluate generative models, such as the Frechet Inception Distance (FID), which measures the similarity between the distribution of generated samples and the distribution of real data.

What is FID?

Frechet Inception Distance (FID) is a metric used to assess the quality of images generated by generative models, such as GANs (Generative Adversarial Networks). It measures how similar the generated images are to real images in terms of their statistical properties.

How to calculate FID:



1. Feature Extraction: FID uses a pre-trained Inception v3 network to extract features from both real and generated images. This network captures high-level representations of the images.

2. Distribution Comparison: It assumes that the feature vectors for both real and generated images follow a multidimensional Gaussian distribution.

3. Statistical Moments: FID calculates the mean and covariance of the feature distributions for both real and generated images.

4. Distance Calculation: The Frechet distance between these two Gaussian distributions is then computed. This distance is defined as:

$$FID = ||\mu_r - \mu_g||^2 + Tr(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^(1/2))$$

   Where:

   - μ_r and μ_g are the mean feature vectors for real and generated images
   - Σ_r and Σ_g are the covariance matrices for real and generated images
   - Tr denotes the trace of a matrix (sum of diagonal elements)

5. Interpretation: A lower FID score indicates that the generated images are more similar to the real images. A score of 0 would mean the two distributions are identical.

6. Advantages:

   - Considers both quality and diversity of generated images
   - Correlates well with human judgment of image quality

- More robust than other metrics like Inception Score

7. Limitations:

    - Depends on the choice of the feature extractor (Inception v3)
    - May not capture all aspects of image quality or diversity
    - Computationally expensive for large datasets

FID has become a standard evaluation metric in the field of generative models, particularly for image generation tasks, due to its ability to provide a meaningful measure of the similarity between generated and real image distributions.

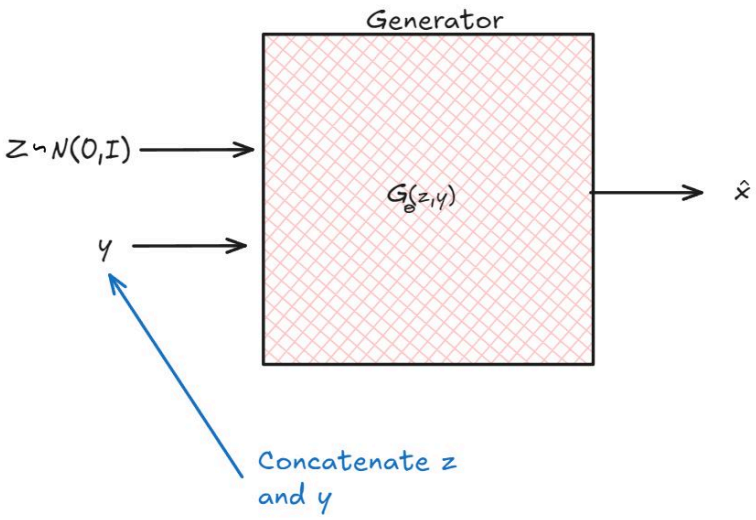
Conditional GANs(cGANs)

A Conditional Generative Adversarial Network (cGAN) is a type of Generative Adversarial Network (GAN) where the generation process is conditioned on some additional information, such as class labels or data from other modalities.

Architecture modification

Generator

Concatenate the conditional information Y with the noise input Z and feed it to the generator.

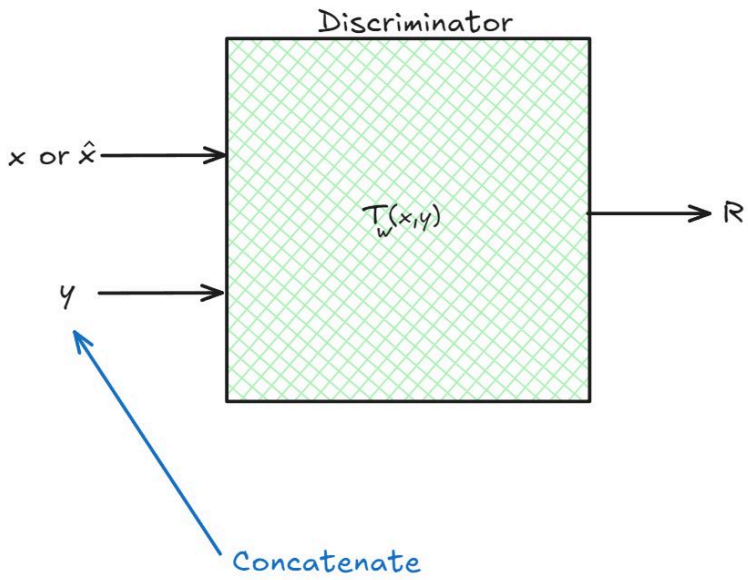


Here Y can be:

- One hot encoded class label
- Text embedding etc

Discriminator

The discriminator in a cGAN is modified similarly to the generator. Concatenate the conditional information Y with the input image X and feed this combined input (X, Y) to the discriminator.



This allows the discriminator to assess both the realism of the image and its correspondence to the given condition.

Objective of cGAN

The objective function of cGAN is an extension of the GAN objective, where both the generator and discriminator are conditioned on y:

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

Where:

- G is the generator
- D is the discriminator
- x is the real data
- z is the random noise input
- y is the conditional information
- $p_{data}(x|y)$ is the conditional probability density function of the real data given y
- $p_z(z)$ is the probability density function of the noise distribution

This objective function encourages the generator to produce samples that not only look realistic but also correspond to the given condition y. The discriminator, in turn, learns to distinguish between real and fake samples while taking the condition into account.

Optimization process

The optimization process for cGANs involves alternating between training the discriminator and the generator:

Discriminator Optimization

The discriminator aims to maximize the probability of correctly classifying real and generated samples, conditioned on y:

$$L_D = -\mathbb{E}_{x \sim p_{data}(x|y)}[\log D(x|y)] - \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z|y)|y))]$$

Generator Optimization

The generator tries to minimize the discriminator's ability to differentiate between real and fake samples by minimizing:

$$L_G = -\mathbb{E}_{z \sim p_z(z)}[\log D(G(z|y)|y)]$$

During training, these two steps are alternated, with the discriminator and generator parameters updated using gradient descent (or a variant) to minimize their respective loss functions.

Domain adversarial networks (DANs) also known as Unsupervised domain adaptation (UDA)

Introduction

Domain adversarial networks (DANs) are a type of neural network architecture designed to address the problem of domain shift, where the distribution of data differs between the training and testing phases. DANs use adversarial training to learn domain-invariant features that are robust across different domains.

Aim

The aim is to learn a classifier that performs well on both the source and target domains, even though the target domain may have different characteristics than the source domain. For example, we train a model to classify medical images at company A and then we want to use this model to classify medical images at some other hospital.

Problem setting

Given:

- A source domain D_s = $\{(x_i^s, y_i^s)\}$, $i = 1, ..., n$ with data distribution $p_s(x, y)$
- A target domain D_t = $\{(x_i^t)\}$, $i = 1, ..., j$ with data distribution $p_t(x)$

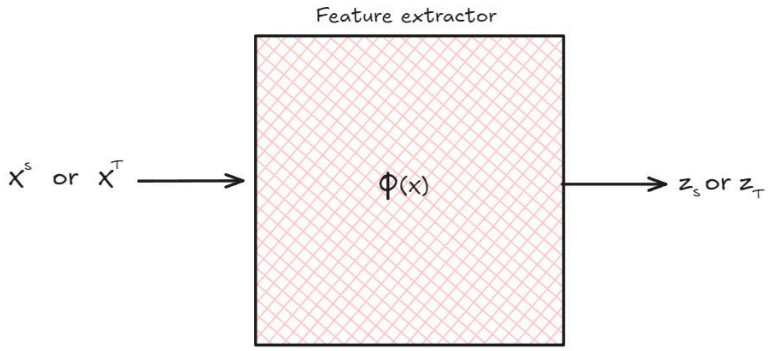
Note that we don't have access to the target labels y_i^t for the target domain.

Goal: learn a classifier $h(x)$ that performs well on the target domain, even though the target domain may have different characteristics than the source domain.

Approach

Feature extractor - Domain invariant feature learning

First a feature extractor is learned that is able to extract features that are domain invariant.



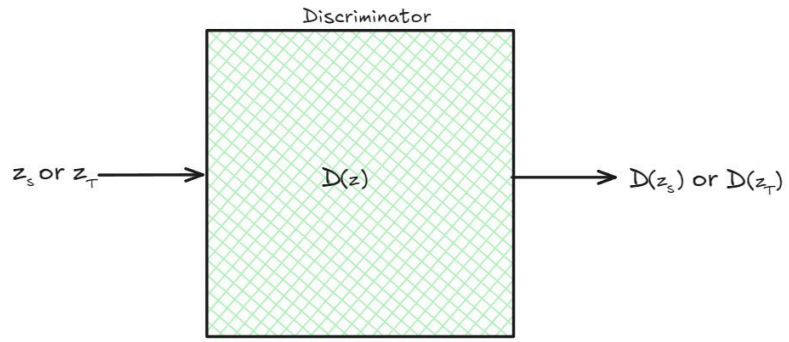
Here $Z_s = \phi(x^s)$ and $Z_t = \phi(x^t)$ are the features extracted from the source and target domains respectively.

Domain discriminator

Let P_{Z_s} and P_{Z_t} denote their distributions of Z_s and Z_t respectively. Our goal is to:

$$\phi^* = \arg\min_{\phi} D(P_{Z_s} \| P_{Z_t})$$

We can use adversarial training to learn this feature extractor. We introduce a domain discriminator $D(z)$ that attempts to distinguish between the source and target features.



Training of feature extractor and domain discriminator

The domain adaptation loss is given by:

$$\mathcal{L}_{DA} = \mathbb{E}_{x^s \sim p_s(x)}[\log(D(\phi(x^s)))] + \mathbb{E}_{x^t \sim p_t(x)}[\log(1 - D(\phi(x^t)))]$$

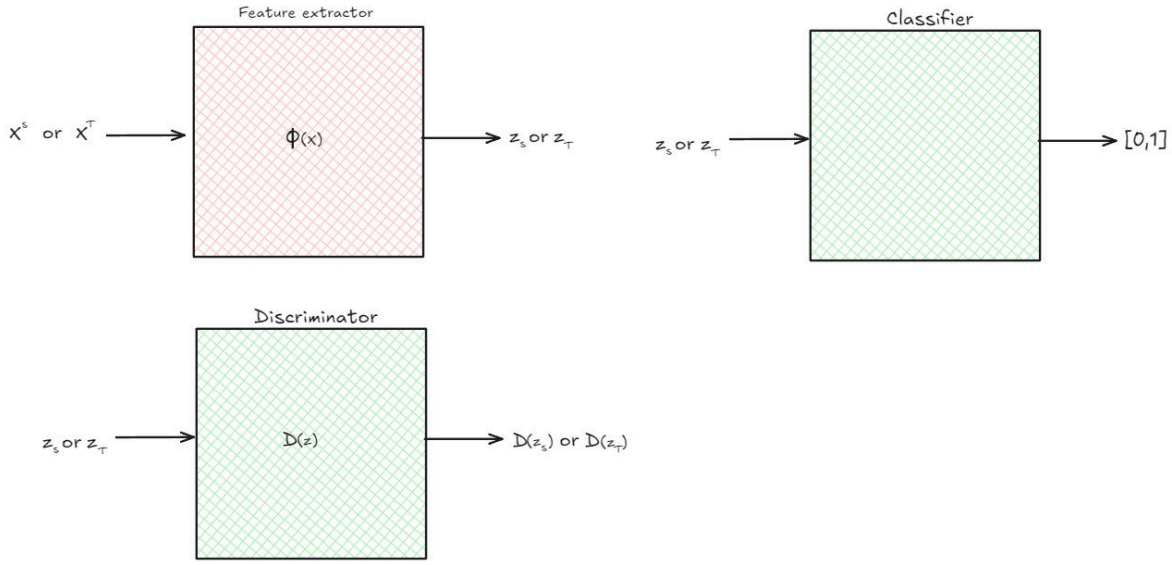
The feature extractor is trained to minimize this loss, while the domain discriminator is trained to maximize it.

Classifier

We train a classifier using pairs of features and labels from the source domain, i.e, D_s = { (Z_i^s, y_i^s)}, i = 1, ..., n, where $Z_i^s = \phi(x_i^s)$ is the extracted feature corresponding to each source data point. This trained classifier is then used to predict the labels of the target domain data.

In actual practice:

- The classifier is trained jointly with the feature extractor and domain discriminator in an end-to-end manner



- The total loss function combines:
    1. The classification loss on source domain data
    2. The domain adaptation loss between source and target domains
- The feature extractor is trained to:
    1. Minimize the classification loss (to learn discriminative features)
    2. Minimize the domain adaptation loss (to learn domain-invariant features)
- The domain discriminator is trained to maximize the domain adaptation loss

Takeaway

- Whenever you encounter a problem where you have a source domain and a target domain, you should first try to learn a domain invariant feature extractor.
- Adversarial training is a powerful technique that should come to your mind whenever you encounter a problem where you want to minimize the distance between two distributions.

Variational encoding(VAEs)

Problem setting

- Given a set of data $D = \{x_i\}_{i=1}^N$ iid samples from some unknown distribution p_{data}.
- We want to model p_{data} as p_θ using a neural network.
- We model p_θ as a latent variable model

Modelling

Log likelihood is given by:

$$\ell(\theta) = \log p_\theta(x)$$

We assume that each data point x_i is associated with a latent variable z_i.
Hence, we will introduce the latent variable z and marginalize over it:

$$\ell(\theta) = \log \sum_z p_\theta(x, z)$$

Let $q(z|x)$ be a conditional distribution over z given x. We multiply and divide by $q(z|x)$ to get:

$$\ell(\theta) = \log \sum_z q(z|x) \frac{p_\theta(x, z)}{q(z|x)}$$

This can be written as:

$$\ell(\theta) = \log \mathbb{E}_{q(z|x)} \left[\frac{p_\theta(x, z)}{q(z|x)} \right]$$

By Jensen's inequality, we have:

$$\log \mathbb{E}_{q(z|x)} \left[\frac{p_\theta(x, z)}{q(z|x)} \right] \geq \mathbb{E}_{q(z|x)} \left[\log \frac{p_\theta(x, z)}{q(z|x)} \right]$$

Hence, we have:

$$\ell(\theta) \geq \mathbb{E}_{q(z|x)} \left[\log \frac{p_\theta(x, z)}{q(z|x)} \right] = F_\theta(q)$$

Where $F_\theta(q)$ is the evidence lower bound (ELBO).

$$F_\theta(q) = \mathbb{E}_{q(z|x)} \left[\log \frac{p_\theta(x|z) p_\theta(z)}{q(z|x)} \right]$$

$$F_\theta(q) = \mathbb{E}_{q(z|x)} \left[\log p_\theta(x|z) \right] + \mathbb{E}_{q(z|x)} \left[\log \frac{p_\theta(z)}{q(z|x)} \right]$$

$$F_\theta(q) = \mathbb{E}_{q(z|x)}\left[\log p_\theta(x|z)\right] - \mathbb{E}_{q(z|x)}\left[\log \frac{q(z|x)}{p_\theta(z)}\right]$$

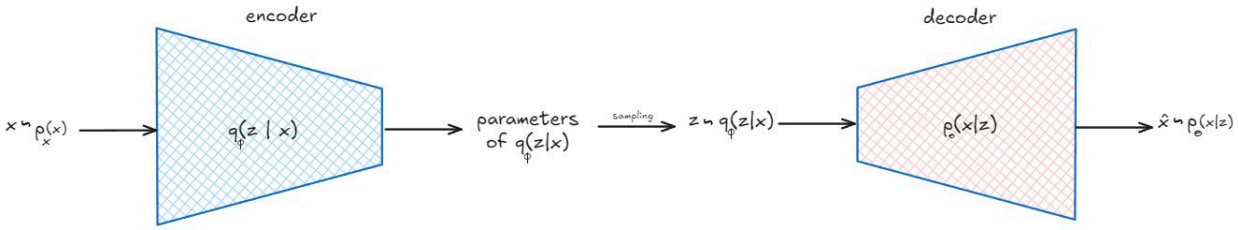
$$F_\theta(q) = \mathbb{E}_{q(z|x)}\left[\log p_\theta(x|z)\right] - D_{KL}\left(q(z|x) \mid p_\theta(z)\right)$$

Here the first term is the conditional log likelihood of the data under the model. We want to maximise this term.

The second term is the KL divergence between the posterior and the prior. We want to minimise this term.

Variational Autoencoder

The Variational Autoencoder (VAE) architecture can be visualized as follows:



In this diagram, we can see the key components of a VAE:

- **Encoder:** We model $q(z|x)$ as a neural network with parameters ϕ. The network takes in an observation x and outputs the parameters of a Gaussian distribution ie mean $\mu_\phi(x)$ and covariance $\Sigma_\phi(x)$.
- **Decoder:** We model $p_\theta(x|z)$ as a neural network with parameters θ. The network takes in a sampled latent variable z from the distribution with parameters $\mu_\phi(x)$ and $\Sigma_\phi(x)$ and outputs a data sample $\hat{x}$. Post training, we use the decoder to generate new data samples ie works as generator.

Motivation for reparameterisation trick

Evidence is intractable, so we are optimizing a lower bound on it.

We need to be able to compute the gradient of the ELBO to be able to do gradient descent.

ELBO's first term's gradient is intractable because the sampling process is non-differentiable hence we use reparameterisation trick.

Reparameterisation trick

Recall that we have to minimize ELBO:

$$F_\theta(q) = \mathbb{E}_{q(z|x)}\left[\log p_\theta(x|z)\right] - D_{KL}\left(q(z|x) \mid p_\theta(z)\right)$$

Focus on first term

$$\mathbb{E}_{q(z|x)}\left[\log p_\theta(x|z)\right]$$

We introduce a function $g_\phi(\epsilon)$ that transforms a noise variable ϵ into z

$$z = g_\phi(\epsilon)$$

Where:

- $\epsilon \sim p(\epsilon)$ (typically a standard normal distribution)
- $g_\phi(\epsilon)$ is our reparameterization function typically $z = g_\phi(\epsilon) = \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon$

This allows us to rewrite the expectation in terms of ϵ

$$\mathbb{E}_{q(z|x)}\left[\log p_\theta(x|z)\right] = \mathbb{E}_{p(\epsilon)}\left[\log p_\theta(x|g_\phi(\epsilon))\right]$$

The gradient of the first term(required for backpropagation) can then be estimated using Monte Carlo sampling:

$$\nabla_\phi \mathbb{E}_{q(z|x)}[\log p_\theta(x|z)] \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_\phi [\log p_\theta(x|g_\phi(\epsilon_i))] \quad \epsilon_i \sim p(\epsilon)$$

Reparameterisation trick in practice

In practice, the reparameterization trick is implemented as follows:

1. We pass a particulat data point x_i through encoder network to get $\mu_\phi(x_i)$ and $\Sigma_\phi(x_i)$ which are the parameters of the distribution $q(z|x)$
2. We sample m number of ϵ_j from a standard normal distribution $N(0, 1)$ for $j = 1, \ldots, m$
3. We compute $z_j^i = \mu_\phi(x_i) + \sigma_\phi(x_i) \odot \epsilon_j$ for $j = 1, \ldots, m$, where m is the number of latent variables we want to sample, and $\sigma_\phi(x_i) = \sqrt{\Sigma_\phi(x_i)}$ is the standard deviation. This gives us m different z_j^i values, each representing a point in the latent space.
4. We then pass each z_j^i through the decoder network to get m different data samples $\hat{x}_j^i$ where $j = 1, \ldots, m$.
5. We want to compute $\mathbb{E}_{q(z|x)}\left[\log p_\theta(x|z)\right] = \mathbb{E}_{p(\epsilon)}\left[\log p_\theta(x|g_\phi(\epsilon))\right]$.
    1. If we assume $p_\theta(x|z) = p_\theta(x|g_\phi(\epsilon)) \sim N(x; x_i, I)$, which is a model assumption. This allows us to calculate the log-likelihood $\log p_\theta(x|z)$ using the generated samples $\hat{x}_j^i$ and the original input x_i as follows(derivation skipped):

$$\mathbb{E}_{q(x|z)}\left[\log p_\theta(x|z)\right] = \mathbb{E}_{p(\epsilon)}\left[\log p_\theta(x|g_\phi(\epsilon))\right] \approx \frac{1}{m}\sum_{j=1}^{m} \log p_\theta(x|g_\phi(\epsilon_j)) \propto \frac{1}{m}\sum_{j=1}^{m} \|x_i - \hat{x}_j^i\|_2^2$$

1. Alternatively, if we assume $p_\theta(x|z) = p_\theta(x|g_\phi(\epsilon))$ follows a Bernoulli distribution, which is often used for binary data, we can calculate the log-likelihood as follows:

$$\mathbb{E}_{q(z|x)}\left[\log p_\theta(x|z)\right] = \mathbb{E}_{p(\epsilon)}\left[\log p_\theta(x|g_\phi(\epsilon))\right] \approx \frac{1}{m}\sum_{j=1}^{m}\sum_{t=1}^{T} x_i^t \log(\hat{x}_j^{i,t}) + (1 - x_i^t)\log(1 - \hat{x}_j^{i,t})$$

Where:

- x_i^t is the t-th dimension of the i-th input data point
- $\hat{x}_j^{i,t}$ is the t-th dimension of the j-th reconstructed sample for the i-th input
- T is the dimensionality of the input data

This formulation is particularly useful for tasks like image generation where pixel values can be treated as binary (black or white) or probabilities of being active.

6. Propagate the gradient of the log-likelihood with respect to the model parameters θ to update the decoder network parameters.

7. Backpropagate through the encoder network to update its parameters ϕ.

Second term of ELBO

Recall that:

$$F_\theta(q) = \mathbb{E}_{q(z|x)}\left[\log p_\theta(x|z)\right] - D_{KL}\left(q(z|x) \mid p_\theta(z)\right)$$

the second term is:

$$D_{KL}\left(q(z|x) \mid p_\theta(z)\right)$$

We want to minimise this term.

We assume the latent prior $p_\theta(z) \sim N(0, I)$, where I is the identity matrix.

The approximate posterior $q(z|x)$ is modeled as $N(z; \mu_\phi(x), \Sigma_\phi(x))$.

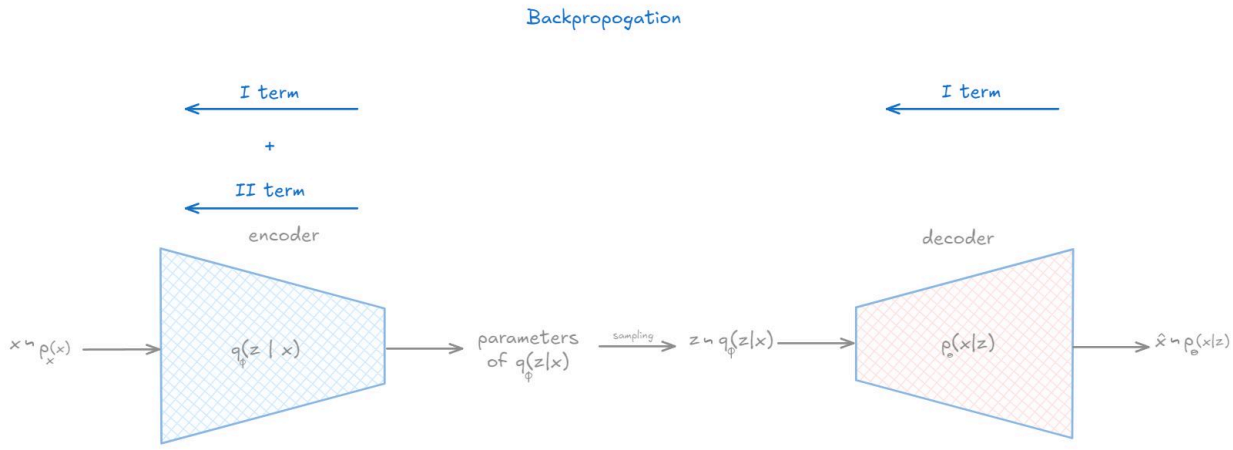
Given these assumptions, we can derive the KL divergence in closed form as:

$$D_{KL}(N(z; \mu_\phi(x), \Sigma_\phi(x)) \| N(0, I)) = \frac{1}{2}\sum_{j=1}^{J}\left(\mu_{\phi,j}^2(x) + \Sigma_{\phi,j}(x) - \log \Sigma_{\phi,j}(x) - 1\right)$$

Where:

- J is the dimensionality of the latent space
- $\mu_{\phi,j}(x)$ is the j-th element of the mean vector
- $\Sigma_{\phi,j}(x)$ is the j-th diagonal element of the covariance matrix

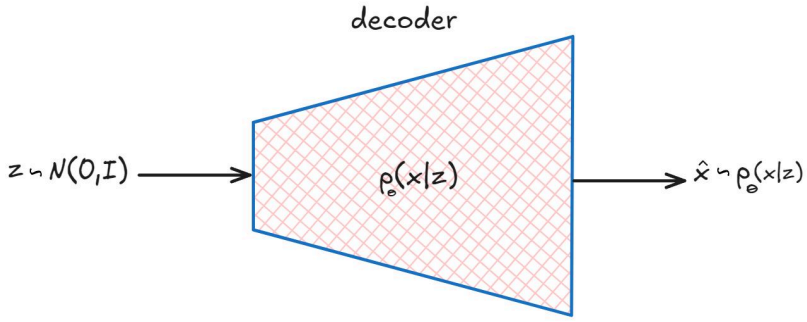
Complete back propagation



Here's a step-by-step breakdown of the complete backpropagation process for a Variational Autoencoder (VAE):

1. Complete picture of passing x_i through encoder
2. Get ϵ_i
3. Get multiple z ... z
4. Pass all of them through decoder get $\hat{x}^1$ ... $\hat{x}^m$
5. Train decoder using only first term
6. While training decoder, also train using second term

Inference

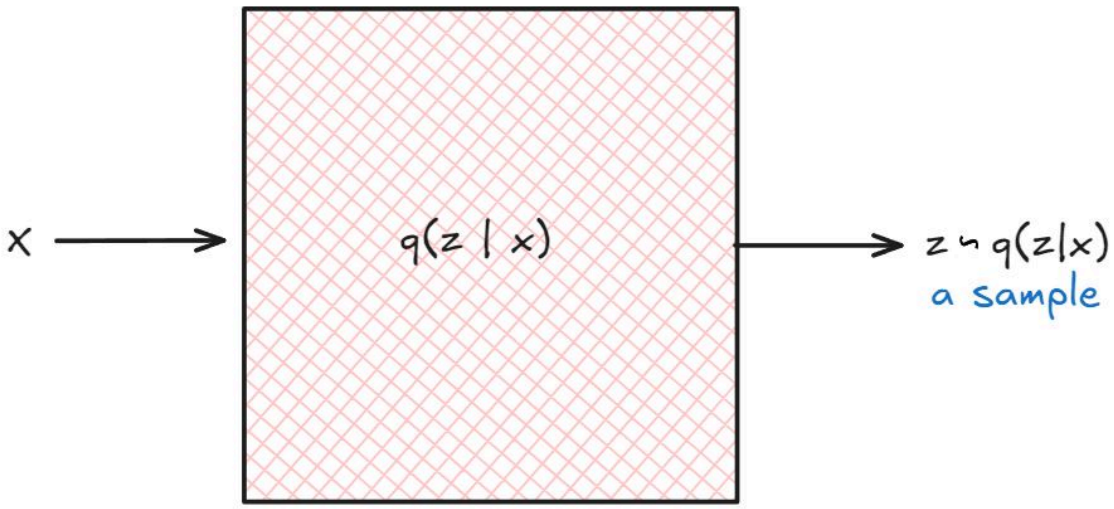


1. Sample from $N(0, I)$ to get z. This works because we trained the decoder wrt to the second term $D_{KL}\left(q(z|x) \mid p_\theta(z)\right)$
2. Pass z through decoder to get $\hat{x}$

Two types of neural networks can be used to model a distribution

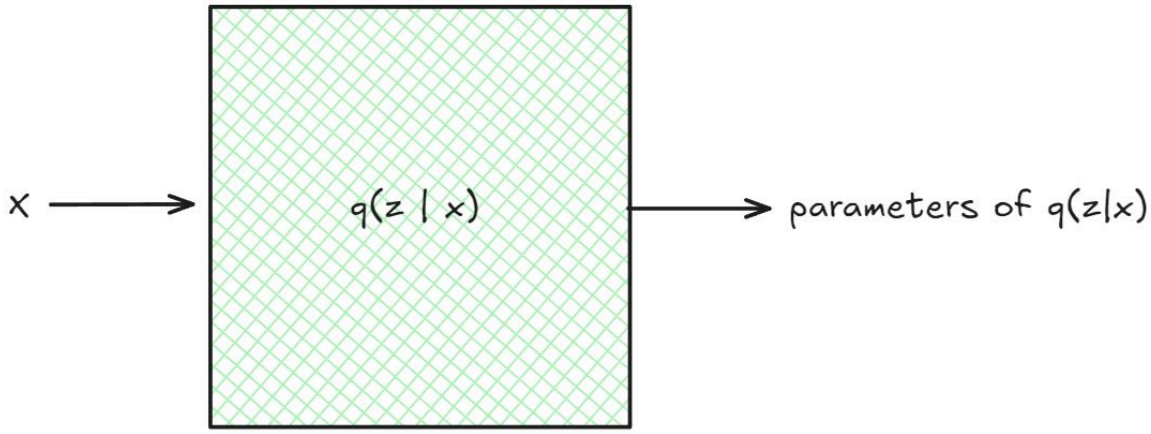
Deterministic way

In this way the output of neural network is a sample from the distribution.



Probabilistic way

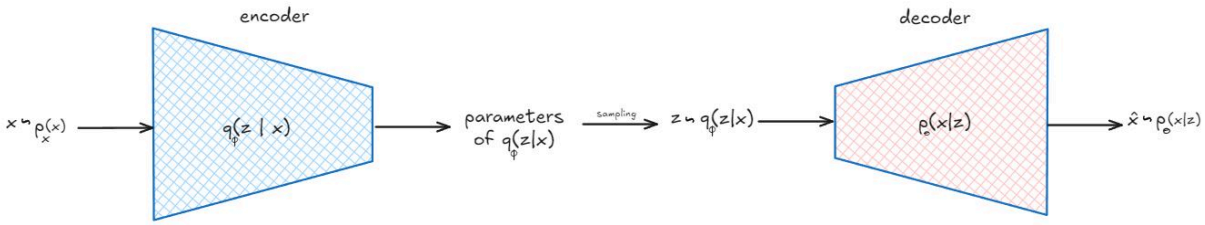
In this way the output of neural network are parameters of the distribution.



Aggregated posterior mismatch

Why did we not have such a problem in EM algorithm?

The Variational Autoencoder (VAE) architecture can be visualized as follows:



Before expanding the KL divergence term, recall that:

latent variable $q_\phi(z|x)$ is the approximate posterior (i.e., how likely the latent variable z is given the input x).
$p_\theta(z|x)$ is the true posterior (i.e., the actual distribution of z given x under the model).
The KL divergence measures how much $q_\phi(z|x)$ deviates from $p_\theta(z|x)$:

We can expand this KL divergence as:

$$D_{KL}(q_\phi(z|x)||p_\theta(z|x)) = \int q_\phi(z|x) \log \frac{q_\phi(z|x)}{p_\theta(z|x)} dz$$

Using Bayes' rule, $p_\theta(z|x) = \frac{p_\theta(x,z)}{p_\theta(x)}$:

$$D_{KL}(q_\phi(z|x)||p_\theta(z|x)) = \int q_\phi(z|x) \log \frac{q_\phi(z|x)}{p_\theta(x,z)/p_\theta(x)} dz$$

Simplifying the fraction:

$$D_{KL}(q_\phi(z|x)||p_\theta(z|x)) = \int q_\phi(z|x) \log \frac{q_\phi(z|x)}{p_\theta(x,z)} p_\theta(x) dz$$

Using the properties of logarithms:

$$D_{KL}(q_\phi(z|x)||p_\theta(z|x)) = \int q_\phi(z|x) [\log q_\phi(z|x) - \log p_\theta(x,z) + \log p_\theta(x)] dz$$

Since $\log p_\theta(x)$ is constant with respect to z:

$$D_{KL}(q_\phi(z|x)||p_\theta(z|x)) = \log p_\theta(x) + \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{q_\phi(z|x)}{p_\theta(x,z)}\right]$$

Using the joint probability decomposition $p_\theta(x,z) = p_\theta(x)p_\theta(z)$:

$$D_{KL}(q_\phi(z|x)||p_\theta(z|x)) = \log p_\theta(x) + \int q_\phi(z|x) \log \frac{q_\phi(z|x)}{p_\theta(x)p_\theta(z)} dz$$

Rearranging the fraction inside the expectation:

$$D_{KL}(q_\phi(z|x)||p_\theta(z|x)) = \log p_\theta(x) - \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x,z)}{q_\phi(z|x)}\right]$$

By definition of the evidence lower bound (ELBO):

$$D_{KL}(q_\phi(z|x)||p_\theta(z|x)) = \log p_\theta(x) - F_\theta(q_\phi)$$

Where $F_\theta(q_\phi)$ is the evidence lower bound (ELBO) we derived earlier.

$$F_\theta(q_\phi) = \log p_\theta(x) - D_{KL}(q_\phi(z|x)||p_\theta(z|x))$$

Recall that in the EM algorithm, we made $q_\phi(z|x) = p_\theta(z|x)$ in the E-step. And found θ by maximizing $F_\theta(q_\phi)$.

However here we cannot get $q_\phi(z|x) = p_\theta(z|x)$. Hence we cannot set the KL divergence to zero, so we choose a family of distributions $q_\phi(z|x)$ and optimize ϕ to minimize the KL divergence between $q_\phi(z|x)$ and $p_\theta(z|x)$.

What is the problem of Aggregated posterior mismatch in VAE?

Consider the distribution called aggregated posterior defined as:

$$q_\phi(z) = \int q_\phi(z|x)p(x)dx$$

For every x, we get a conditional distribution. To get the aggregated posterior, we marginalize over all x.

The aggregated posterior $q_\phi(z)$ represents the overall distribution of the latent variable z across all data points x. Ideally, this should match the model's prior distribution $p_\theta(z)$, which we assume before seeing any data.

However, when there is a mismatch between $q_\phi(z|x)$ and $p_\theta(z|x)$ for some data points, it leads to a discrepancy:

$$q_\phi(z) = \int q_\phi(z|x)p(x)dx \neq \int p_\theta(z|x)p(x)dx = p_\theta(z)$$

This discrepancy between $q_\phi(z)$ and $p_\theta(z)$ is called the aggregated posterior mismatch. It indicates that the learned latent representations are not aligned with the true prior, which can result in poorer generalization and generation quality in VAEs.

VAE as regularized autoencoder

Understanding VAE as a Regularized Autoencoder

Recall that the Variational Autoencoder (VAE) objective is given by:

$$F_\theta(q) = \mathbb{E}_{q_\phi(z|x)} \left[\log p_\theta(x|z)\right] - D_{KL} \left(q_\phi(z|x) \| p(z)\right)$$

We can rewrite this as:

$$F_\theta(q) = \mathbb{E}_{q_\phi(z|x)} \left[\|x - \hat{x}\|^2\right] - D_{KL} \left(q_\phi(z|x) \| p(z)\right)$$

Where:

- $\mathbb{E}_{q_\phi(z|x)} \left[\|x - \hat{x}\|^2\right]$ is the reconstruction error
- $D_{KL} \left(q_\phi(z|x) \| p(z)\right)$ is the KL divergence, which acts as a regularizer

This formulation shows that the VAE objective can be interpreted as a regularized autoencoder, where the regularization term encourages the latent distribution to be close to the prior $p(z)$.

Connection to Regularization and MAP Estimation

In regularized empirical risk minimization (ERM), we have:

$$R_{reg}(\theta) = R(\theta) + \lambda \Omega(\theta)$$

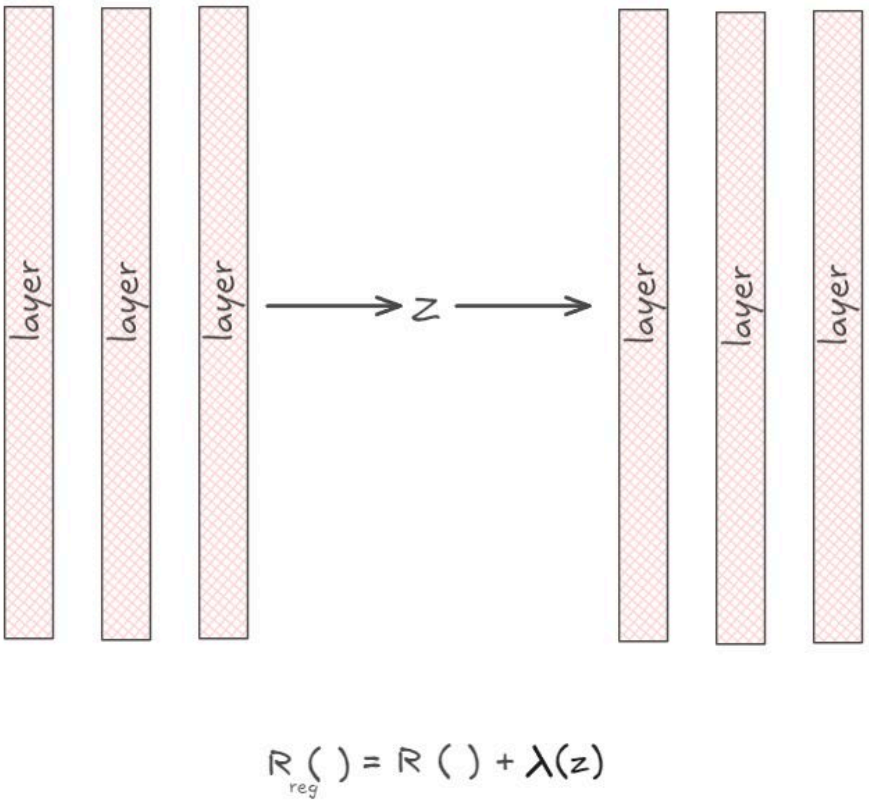
Where:

- $R(\theta)$ is the empirical risk
- $\Omega(\theta)$ is the regularization term
- λ is the regularization strength

Regularized ERM can be seen as equivalent to Maximum A Posteriori (MAP) estimation:

$$R_{reg}(\theta) \sim MAP$$

However, the prior does not need to be on the parameters θ; it can also be on the outputs of an intermediate layer. In the context of VAEs, the KL divergence term acts as a prior on the latent variable z.



Thus, the VAE objective:

$$F_\theta(q) = \mathbb{E}_{q_\phi(z|x)}\left[\|x - \hat{x}\|^2\right] - D_{KL}\left(q_\phi(z|x)\|p(z)\right)$$

can be seen as a form of regularized ERM with a prior on the output of the latent space.

Beta VAEs

Beta VAEs are an extension of Variational Autoencoders (VAEs) that introduce a hyperparameter β to control the trade-off between reconstruction accuracy and latent space regularization. The objective function is given by:

$$F_\theta(q) = \mathbb{E}_{q_\phi(z|x)}\left[\|x - \hat{x}\|^2\right] + \beta D_{KL}\left(q_\phi(z|x)\|p(z)\right)$$

Where:

- The first term $\mathbb{E}_{q_\phi(z|x)}\left[\|x - \hat{x}\|^2\right]$ is the reconstruction loss, measuring how well the model reconstructs input x from latent z

- The second term $D_{KL}\left(q_\phi(z|x)||p(z)\right)$ is the KL divergence regularizer ensuring the learned latent distribution matches the prior
- The hyperparameter β controls the regularization strength:
    - When $\beta = 1$, this reduces to the standard VAE
    - When $\beta > 1$, stronger regularization encourages more disentangled latent representations
    - When $\beta < 1$, the model prioritizes reconstruction accuracy over prior matching

Info VAEs: Information Maximizing Variational Autoencoders

InfoVAE addresses the aggregated posterior mismatch problem in standard VAEs by modifying the objective function to explicitly match the aggregated posterior with the prior distribution.

Key Ideas

1. If the aggregated posterior $q_\phi(z)$ becomes a Dirac delta, all inputs x map to the same latent code z
2. In such a scenario, the latent code z would contain no information about the input x
3. The standard VAE objective does not directly encourage the matching between $q_\phi(z)$ and the prior $p(z)$

InfoVAE Objective Function

The standard VAE objective is given by:

$$F_\theta(q) = \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z)\right] - D_{KL}\left(q_\phi(z|x)\|p(z)\right)$$

This can also be written as(derivation skipped):

$$L_{ELBO} = -D_{KL}(q_\phi(z)\|p(z)) - \mathbb{E}_{q_\phi(z)}[D_{KL}(q_\phi(x|z)\|p_\theta(x|z))]$$

Here $p_\theta(x|z)$ is inverted encoder distribution. The first term means that $q_\phi(z)$ should match $p(z)$. The second term means that $q_\phi(x|z)$ should match $p_\theta(x|z)$.

Incorporating Mutual Information

To further enhance information retention, the InfoVAE objective incorporates a mutual information term:

$$L_{ELBO} = -\lambda D_{KL}(q_\phi(z)\|p(z)) - \mathbb{E}_{q_\phi(z)}[D_{KL}(q_\phi(x|z)\|p_\theta(x|z))] + \alpha I_q(x;z)$$

Terms Breakdown:

- $I_q(x;z)$: Mutual information between input x and latent code z, encouraging z to retain meaningful information about x. The mutual information $I_q(x;z)$ is defined as:

$$I_q(x;z) = \sum_x \sum_z p_\theta(x,z) \log \frac{p_\theta(x,z)}{p_\theta(x)p_\theta(z)}$$

- α: Coefficient controlling the importance of the mutual information term

InfoVAE addresses the aggregated posterior mismatch problem by:

1. Explicitly minimizing the KL divergence between the aggregated posterior $q_\phi(z)$ and the prior $p_\theta(z)$
2. Preserving mutual information between the input x and the latent code z

Training InfoVAE

Because optimizing the mutual information term is difficult, we optimize the following objective instead(derivation skipped however it is in paper):

$$L_{ELBO} = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - (1-\alpha)D_{KL}(q_\phi(z|x)||p(z)) - (\alpha+\lambda-1)D_{KL}(q_\phi(z)||p(z))$$

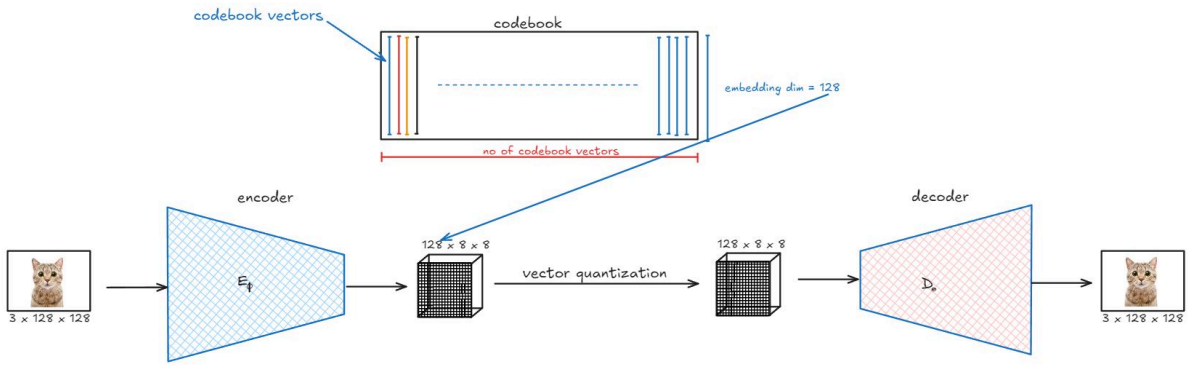
The first two terms are optimized similar to VAE while the third term, which is minimization of the divergence between the aggregated posterior and the latent prior, is carried out using techniques such as adversarial minimization.

Vector Quantized Variational Autoencoders (VQ-VAEs)

Vector Quantized Variational Autoencoders (VQ-VAEs) were introduced to address several limitations of regular VAEs:

1. The “posterior collapse” problem where the latent code is ignored
2. The continuous latent space can make it difficult to model discrete structures
3. The aggregated posterior mismatch between $q_\phi(z)$ and the prior $p(z)$

VQ-VAEs solve these issues by using discrete latent variables through vector quantization. The architecture consists of:



1. An encoder network E_ϕ that maps input $x \in \mathbb{R}^N$ to continuous latent vectors $z_e = E_\phi(x) \in \mathbb{R}^D$

2. A codebook $\mathcal{C} = \{e_k\}_{k=1}^K$ containing K learnable embedding vectors $e_k \in \mathbb{R}^D$, where typically K is large (e.g., 512 or 1024)

3. A vector quantization operation that maps z_e to the nearest codebook vector: $z_q = e_k$ where $k = \arg\min_j \|z_e - e_j\|_2$

4. A decoder network D_θ that reconstructs the input from the quantized vectors: $\hat{x} = D_\theta(z_q)$

The training objective consists of three terms:

$$L = \underbrace{\|x - D_\theta(z_q)\|_2^2}_{\text{reconstruction loss}} + \underbrace{\|\text{sg}[z_e] - e_k\|_2^2}_{\text{codebook loss}} + \underbrace{\beta\|z_e - \text{sg}[e_k]\|_2^2}_{\text{commitment loss}}$$

where:

- $\text{sg}[\cdot]$ is the stop-gradient operator
- The codebook loss updates the embeddings to match the encoder output
- The commitment loss ensures the encoder commits to codebook vectors
- β is a hyperparameter controlling the commitment (typically 0.25)

Unlike regular VAEs that use a continuous latent space with sampling and KL divergence regularization, VQ-VAEs use a discrete latent space through the codebook. This allows them to better model discrete structures while avoiding posterior collapse and distribution matching issues.

Denoising Diffusion Probabilistic Models (DDPM) - part 1

DDPM as special case of variational autoencoders (VAE)

Impose following structure on the VAE:

- Multiple latent variables, where each latent variable is associated with a timestep.
- Dim of all the latent variables is the same as the dim of the data.
- Assume a non learnable encoder unlike VAE where the encoder was learnable.

Notations

- $X_1, ..., X_T$: Denote latent variables at different timesteps
- $p(X_0)$: Denote the model data distribution
- $X_{1:T} = (X_1, X_2, ..., X_T)$: Denote the sequence of all latent variables
- $q(X_{1:T}|X_0)$: Denote the latent posterior

Latent posterior

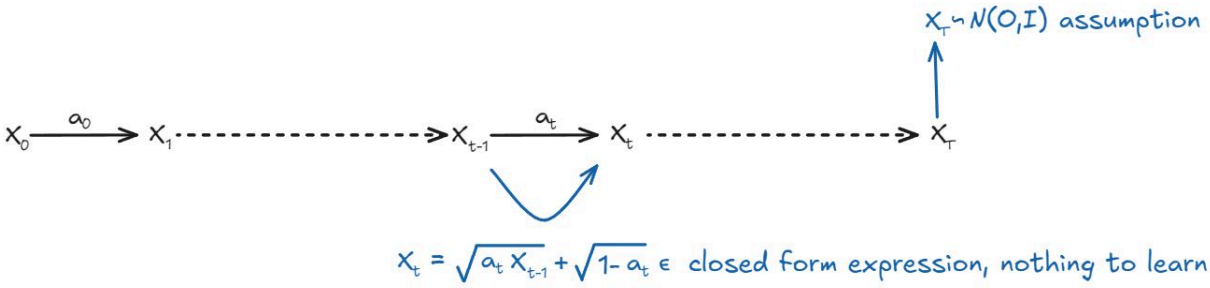
Because $q(X_{1:T}|X_0)$ is not learnable, we need to define it - we make it first order markovian chain with gaussian transitions:

$$X_0 \to X_1 \to X_2 \to ... \to X_T$$

This is also called encoding or forward process or adding noise process. Here:

$$X_t = \sqrt{\alpha_t} X_{t-1} + \sqrt{1 - \alpha_t}\epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

where $\alpha_t \in (0, 1)$ called variance schedule are hyperparameters that control the variance of the latent variables at different timesteps. Generally, α_t is chosen to be a constant or a slowly decaying function of t to ensure that the variance of the latent variables at different timesteps is not too high or too low.



Because we are interested in modeling latents, we use the chain rule and Markov property of the forward process:

$$q(X_{1:T}|X_0) = \prod_{t=1}^{T} q(X_t|X_{t-1})$$

Because of reparameterization of $X_t = \sqrt{\alpha_t}X_{t-1} + \sqrt{1-\alpha_t}\epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$

$$q(X_t|X_{t-1}) = \mathcal{N}(X_t; \sqrt{\alpha_t}X_{t-1}, (1-\alpha_t)I)$$

Model

We define model $p_\theta(X_0)$ as:

$$p_\theta(X_0) = \int p_\theta(X_{0:T})\, dX_{1:T}$$

However, this is not tractable to compute directly. Instead, we can use the Evidence Lower Bound (ELBO) to optimize our model.

Recall the Evidence Lower Bound (ELBO) in general is given by:

$$F_\theta(q) \geq \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z)\right] - D_{KL}\left(q_\phi(z|x)||p(z)\right) = ELBO$$

substitute $q_\phi(z|x)$ with $q(X_{1:T}|X_0)$ and $p(z)$ with $p(X_{1:T})$:

$$F_\theta(q) \geq \mathbb{E}_{q(X_{1:T}|X_0)}\left[\log p_\theta(X_0|X_{1:T})\right] - D_{KL}\left(q(X_{1:T}|X_0)||p(X_{1:T})\right) = ELBO$$

KL divergence is given by:

$$D_{KL}\left(q(X_{1:T}|X_0)||p(X_{1:T})\right) = \mathbb{E}_{q(X_{1:T}|X_0)}\left[\log \frac{q(X_{1:T}|X_0)}{p(X_{1:T})}\right]$$

Hence,

$$ELBO = \mathbb{E}_{q(X_{1:T}|X_0)}\left[\log p_\theta(X_0|X_{1:T})\right] - \mathbb{E}_{q(X_{1:T}|X_0)}\left[\log \frac{q(X_{1:T}|X_0)}{p(X_{1:T})}\right]$$

$$= \mathbb{E}_{q(X_{1:T}|X_0)}\left[\log p_\theta(X_0|X_{1:T}) + \log \frac{p_\theta(X_{1:T})}{q(X_{1:T}|X_0)}\right]$$

$$= \mathbb{E}_{q(X_{1:T}|X_0)}\left[\log \frac{p_\theta(X_0|X_{1:T})p_\theta(X_{1:T})}{q(X_{1:T}|X_0)}\right]$$

$$= \mathbb{E}_{q(X_{1:T}|X_0)}\left[\log \frac{p_\theta(X_{0:T})}{q(X_{1:T}|X_0)}\right]$$

The markov chain in reverse is also known as reverse/decoding/denoising/sampling process:

$$X_T \to X_{T-1} \to X_{T-2} \to ... \to X_0$$

- It is also a first order markov chain with gaussian transitions
- Hence, we can express the modelled joint distribution of the reverse process by applying the chain rule of the reverse process:

$$p_\theta(X_{0:T}) = p(X_T)\prod_{t=1}^{T} p_\theta(X_{t-1}|X_t)$$

Where:

- $p(X_T)$ is the prior distribution of the latent variable at the final timestep
- $p_\theta(X_{t-1}|X_t)$ represents the transition probability from X_{t-1} to X_t in the reverse process, parameterized by θ which we aim to learn

Substitute $p_\theta(X_{0:T})$ in ELBO:

$$ELBO = \mathbb{E}_{q(X_{1:T}|X_0)}\left[\log \frac{p(X_T)\prod_{t=1}^{T} p_\theta(X_{t-1}|X_t)}{q(X_{1:T}|X_0)}\right]$$

Using the Markov property of the forward process, we can rewrite $q(X_{1:T}|X_0)$ as:

$$q(X_{1:T}|X_0) = \prod_{t=1}^{T} q(X_t|X_{t-1})$$

Substituting this into the ELBO equation:

$$ELBO = \mathbb{E}_{q(X_{1:T}|X_0)}\left[\log \frac{p(X_T)\prod_{t=1}^{T} p_\theta(X_{t-1}|X_t)}{\prod_{t=1}^{T} q(X_t|X_{t-1})}\right]$$

$$ELBO = \mathbb{E}_{q(X_{1:T}|X_0)}\left[\log\left(\frac{p(X_T)\prod_{t=1}^{T} p_\theta(X_{t-1}|X_t)}{q(X_1|X_0)\prod_{t=2}^{T} q(X_t|X_{t-1})}\right)\right]$$

Breaking down the ELBO equation into components:

1. First, we separate the log terms:

$$ELBO = \mathbb{E}_{q(X_{1:T}|X_0)}\left[\log p(X_T) + \log\left(\frac{p_\theta(X_0|X_1)}{q(X_1|X_0)}\right) + \sum_{t=2}^{T}\log\left(\frac{p_\theta(X_{t-1}|X_t)}{q(X_t|X_{t-1})}\right)\right]$$

Because of the Markov property $q(X_t|X_{t-1}) = q(X_t|X_{t-1}, X_0)$, we can write the last term as:

$$ELBO = \mathbb{E}_{q(X_{1:T}|X_0)}\left[\log p(X_T) + \log\left(\frac{p_\theta(X_0|X_1)}{q(X_1|X_0)}\right) + \sum_{t=2}^{T}\log\left(\frac{p_\theta(X_{t-1}|X_t)}{q(X_t|X_{t-1}, X_0)}\right)\right]$$

Now apply Bayes' rule(for 3 random variables) to the last term ie:

$$q(X_{t-1}|X_t, X_0) = \frac{q(X_t|X_{t-1}, X_0)q(X_{t-1}|X_0)}{q(X_{t-1}|X_0)}$$

Substitute this in the last term of the ELBO equation and simplify($logq(x_1|x_0)$ cancels out) to get final form of the ELBO equation:

$$ELBO = \mathbb{E}_{q(X_{1:T}|X_0)}\left[\log p_\theta(X_0|X_1) + \log\frac{p(X_T)}{q(X_T|X_0)} + \sum_{t=2}^{T}\log\left(\frac{p_\theta(X_{t-1}|X_t)}{q(X_{t-1}|X_t, X_0)}\right)\right]$$

This gives us our final ELBO equation with three distinct terms:

1. A reconstruction term: $\log p_\theta(X_0|X_1)$
2. A prior matching term: $\log \frac{p(X_T)}{q(X_T|X_0)}$
3. A transition matching term: $\sum_{t=2}^{T}\log \frac{p_\theta(X_{t-1}|X_t)}{q(X_{t-1}|X_t, X_0)}$

Denoising Diffusion Probabilistic Models (DDPM) - part 2

We were using latent variables X_t to model the data as:

$$\log p_\theta(x_0) = \log \int p_\theta(x_{0:T}) dx_{1:T}$$

This formulation expresses the log-likelihood of the observed data x_0 as an integral over all possible latent variable sequences $x_{1:T}$. The model's goal is to maximize this log-likelihood, which involves learning the parameters θ that define the generative process from x_T back to x_0.

Also:

- $p_\theta(x_T) = \mathcal{N}(0, I)$ - The prior distribution of the final latent variable is a standard normal distribution
- $p_\theta(x_{0:T}) = p_\theta(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t)$ - The joint distribution of all latent variables is factorized using the chain rule
- $q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{\alpha_t}x_{t-1}, (1-\alpha_t)I)$ - The forward process transition probability is a Gaussian distribution
- $q(x_{1:T}|x_0) = \prod_{t=1}^{T} q(x_t|x_{t-1})$ - The posterior of all latent variables given the data is factorized using the chain rule

Because the log likelihood is intractable, we constructed the ELBO of the model:

$$\mathcal{L} = \mathbb{E}_{q(x_{0:T}|x_0)} \left[\log \frac{p_\theta(x_{0:T})}{q(x_{1:T}|x_0)} \right]$$

We did algebraic manipulations to decompose ELBO into three terms: reconstruction, prior matching, and transition matching terms.

$$\mathcal{L} = \mathbb{E}_{q(x_{1:T}|x_0)} \left[\log p_\theta(x_0|x_1) + \log \frac{p_\theta(x_T)}{q(x_T|x_0)} + \sum_{t=2}^{T} \log \frac{p_\theta(x_{t-1}|x_t)}{q(x_{t-1}|x_t, x_0)} \right]$$

Reconstruction term

The reconstruction term is given by:

$$T_1 = \mathbb{E}_{q(x_{1:T}|x_0)} \log p_\theta(x_0|x_1)$$

Since this term inside the expectation only depends on x_1 when predicting x_0, we can simplify it to $\mathbb{E}_{q(x_1|x_0)}$

$$T_1 = \mathbb{E}_{q(x_1|x_0)} \log p_\theta(x_0|x_1)$$

This step demonstrates that the reconstruction term only depends on x_0 and x_1, regardless of the other latent variables.

Recall that the forward process is defined as:

$$x_1 = \sqrt{\alpha_1} x_0 + \sqrt{1 - \alpha_1}\epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

To reconstruct x_0 from x_1 (the first latent sample), we assume a Gaussian distribution:

$$p_\theta(x_0|x_1) = \mathcal{N}(x_0; \mu_\theta(x_1, 1), \sigma_1^2 I)$$

Where:

- $\mu_\theta(x_1, 1)$ is the mean predicted by the model
- σ_1^2 is a fixed variance (usually set to $\beta_1 = 1 - \alpha_1$)

With this Gaussian assumption, the reconstruction term becomes:

$$T_1 \approx -\frac{1}{2\sigma_1^2}\mathbb{E}_{q(x_1|x_0)}\|\mu_\theta(x_1, 1) - x_0\|^2 + C$$

Where C is a constant independent of θ

Prior matching term

The prior matching term is given by:

$$T_2 = \mathbb{E}_{q(x_{1:T}|x_0)}\left[\log \frac{p_\theta(x_T)}{q(x_T|x_0)}\right]$$

Because the term inside the expectation $\mathbb{E}_{q(x_{1:T}|x_0)}$ is independent of $x_{2:T}$, we can simplify it to $\mathbb{E}_{q(x_T|x_0)}$

$$T_2 = \mathbb{E}_{q(x_T|x_0)} \left[\log \frac{p_\theta(x_T)}{q(x_T|x_0)} \right]$$

This term can be simplified as:

$$T_2 = \mathbb{E}_{q(x_T|x_0)} \left[\log p_\theta(x_T) - \log q(x_T|x_0) \right]$$

$$T_2 = -D_{KL}(q(x_T|x_0) || p_\theta(x_T))$$

Note that T_2 can be ignored during optimization because:

1. It doesn't depend on the model parameters θ
2. $p_\theta(x_T) = \mathcal{N}(0, I)$ is fixed

This prior matching term is similar to the KL divergence term in Variational Autoencoders (VAEs), $D_{KL}(q(z|x)||p(z))$, where it encourages the approximate posterior to match the prior distribution.

Transition matching/ Consistency/ Denoising matching term

The transition matching term is given by:

$$T_3 = \sum_{t=2}^{T} \mathbb{E}_{q(x_{1:T}|x_0)} \left[\log \frac{p_\theta(x_{t-1}|x_t)}{q(x_{t-1}|x_t, x_0)} \right]$$

Since the term inside the expectation $\mathbb{E}_{q(x_{1:T}|x_0)}$ is independent of $x_{2:T}$, we can simplify it to $\mathbb{E}_{q(x_t, x_{t-1}|x_0)}$

$$T_3 = \sum_{t=2}^{T} \mathbb{E}_{q(x_t, x_{t-1}|x_0)} \left[\log \frac{p_\theta(x_{t-1}|x_t)}{q(x_{t-1}|x_t, x_0)} \right]$$

It can be shown that:

$$T_3 = -\sum_{t=2}^{T} \mathbb{E}_{q(x_t|x_0)} [D_{KL}(q(x_{t-1}|x_t, x_0) || p_\theta(x_{t-1}|x_t))]$$

Consider the first term in the KL divergence. By applying Bayes' law, we can rewrite it as:

$$q(x_{t-1}|x_t, x_0) = q(x_t|x_{t-1}, x_0)\frac{q(x_{t-1}|x_0)}{q(x_t|x_0)}$$

$$q(x_{t-1}|x_t, x_0) = \frac{q(x_t|x_{t-1})q(x_{t-1}|x_0)}{q(x_t|x_0)}$$

We know that:

- $q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{\alpha_t}x_{t-1}, (1-\alpha_t)I)$
- $q(x_t|x_0) = \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)I)$ where $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$ - Skipped derivation but it is in class notes
- $q(x_{t-1}|x_0) = \mathcal{N}(x_{t-1}; \sqrt{\bar{\alpha}_{t-1}}x_0, (1-\bar{\alpha}_{t-1})I)$

Substituting these into the equation for $q(x_{t-1}|x_t, x_0)$, we get(Derivation skipped but it is eqn 84 in https://arxiv.org/pdf/2208.11970):

$$q(x_{t-1}|x_t, x_0) = \frac{\mathcal{N}(x_t; \sqrt{\alpha_t}x_{t-1}, (1-\alpha_t)I)\mathcal{N}(x_{t-1}; \sqrt{\bar{\alpha}_{t-1}}x_0, (1-\bar{\alpha}_{t-1})I)}{\mathcal{N}(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)I)}$$

This can be simplified to:

$$q(x_{t-1}|x_t, x_0) = \mathcal{N}(x_{t-1}; \mu_q(x_t, x_0), \Sigma_q(t))$$

Where:

$$\mu_q(x_t, x_0) = \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})x_t + \sqrt{\bar{\alpha}_{t-1}}(1-\alpha_t)x_0}{1-\bar{\alpha}_t}$$

$$\Sigma_q(t) = \frac{(1-\alpha_t)(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}I$$

Now, let's consider the second term in the KL divergence, $p_\theta(x_{t-1}|x_t)$. In DDPM, this is modeled as a Gaussian distribution:

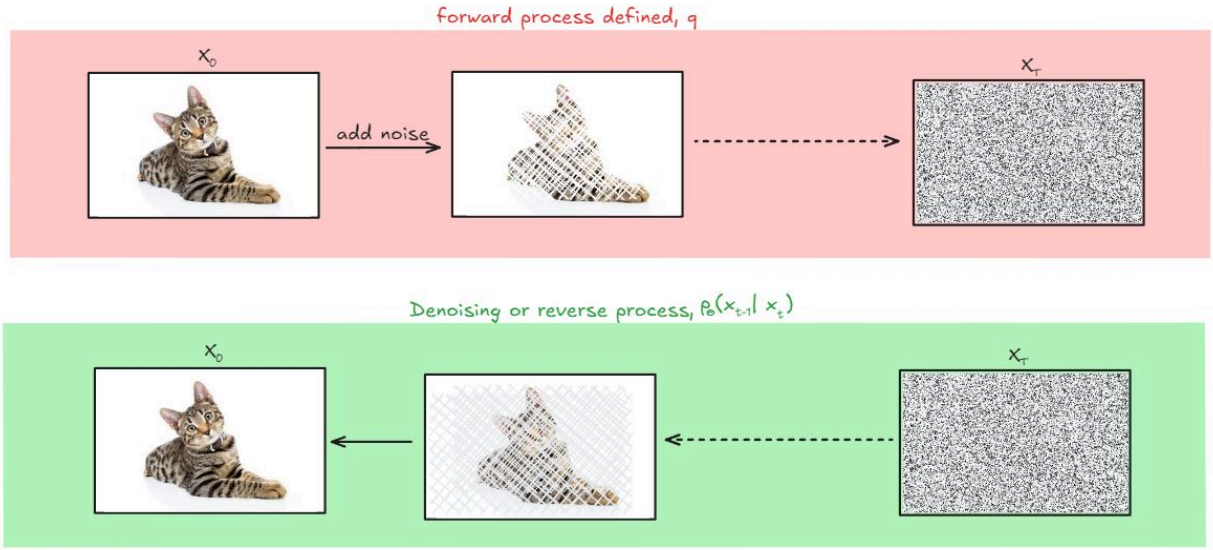
$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t), \Sigma_\theta(x_t))$$

The KL divergence between two Gaussian distributions has a closed-form expression. Therefore, we can compute T_3 analytically:

$$T_3 = -\sum_{t=2}^{T} \mathbb{E}_{q(x_t|x_0)}\left[\frac{1}{2}\left(\log\frac{|\Sigma_\theta|}{|\Sigma_q|} - d + tr(\Sigma_\theta^{-1}\Sigma_q) + (\mu_q - \mu_\theta)^T\Sigma_\theta^{-1}(\mu_q - \mu_\theta)\right)\right]$$

Here, d is the dimensionality of the data, and tr denotes the trace of a matrix.

This term encourages the learned reverse process $p_\theta(x_{t-1}|x_t)$ to match the true posterior $q(x_{t-1}|x_t, x_0)$, which is why it's called the transition matching or consistency term.



Denoising Diffusion Probabilistic Models (DDPM) - part 3

Recall that:

$$T_3 = -\sum_{t=2}^{T} \mathbb{E}_{q(x_t|x_0)}[D_{KL}(q(x_{t-1}|x_t, x_0)||p_\theta(x_{t-1}|x_t))]$$

Here:

- $q(x_{t-1}|x_t, x_0)$ is the noising distribution after applying the forward process for $t-1$ steps
- $p_\theta(x_{t-1}|x_t)$ is the model's predicted denoising distribution after applying the reverse process

We also know that:

$$q(x_{t-1}|x_t, x_0) = \mathcal{N}(x_{t-1}; \mu_q(x_t, x_0), \Sigma_q(t))$$

where $\mu_q(x_t, x_0)$ and $\Sigma_q(t)$ are the mean and covariance of the posterior distribution and:

$$\mu_q(x_t, x_0) = \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})x_t + \sqrt{\bar{\alpha}_{t-1}}(1-\alpha_t)x_0}{1-\bar{\alpha}_t}$$

The covariance $\Sigma_q(t)$ is given by:

$$\Sigma_q(t) = \frac{(1-\alpha_t)(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} I = \sigma_q^2(t) I$$

where $\sigma_q^2(t)$ is a scalar that depends on t. This is a model assumption that simplifies the calculations by assuming $\Sigma_q(t)$ is a diagonal matrix (proportional to the identity matrix I) for each timestep t, with its elements determined by $\sigma_q^2(t)$.

Also, we have:

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \sigma_\theta^2 I)$$

where $\mu_\theta(x_t, t)$ is the mean of the model's prediction and σ_θ^2 is a scalar variance, assuming the covariance is proportional to the identity matrix I.

Note: In both the forward process (q) and the reverse process (p), the covariance matrices are assumed to be diagonal and proportional to the identity matrix. This simplification is indeed a key assumption in the DDPM model.

Hence:

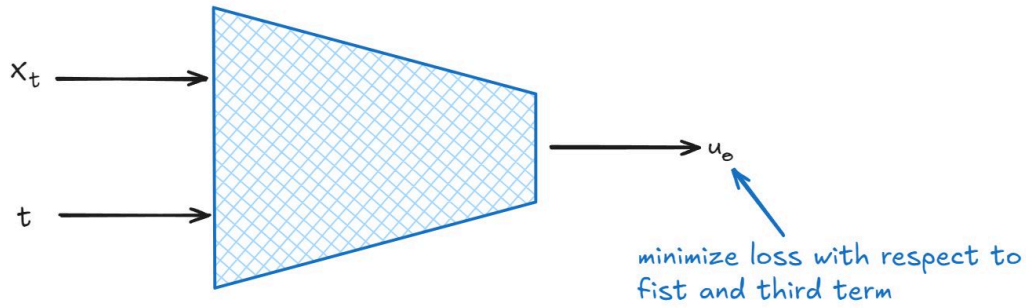
$$T_3 = -\sum_{t=2}^{T} \mathbb{E}_{q(x_t|x_0)}[D_{KL}(q(x_{t-1}|x_t, x_0)||p_\theta(x_{t-1}|x_t))]$$

$$T_3 = -\sum_{t=2}^{T} \mathbb{E}_{q(x_t|x_0)}[D_{KL}(\mathcal{N}(\mu_q(x_t, x_0), \sigma_q^2(t)I) \| \mathcal{N}(\mu_\theta(x_t, t), \sigma_\theta^2 I))]$$

$$T_3 = -\sum_{t=2}^{T} \mathbb{E}_{q(x_t|x_0)}[\frac{1}{2\sigma_\theta^2} ||\mu_q(x_t, x_0) - \mu_\theta(x_t, t)||^2]$$

Notice that:

- $\mu_q(x_t, x_0)$ is the mean of the posterior distribution, which is a function of x_t and x_0. This is a known value.
- $\mu_\theta(x_t, t)$ is the mean of the model's prediction, which is a function of x_t and t. This is a learnable parameter.



This could tatally work however the loss is formulated by reparametrizing.

Reparameterization

Recall that the forward process is defined as:

$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

From this, we can rearrange to express x_0 in terms of x_t:

$$x_0 = \frac{x_t - \sqrt{1 - \bar{\alpha}_t}\epsilon}{\sqrt{\bar{\alpha}_t}}$$

Substituting this expression for x_0 into the equation for $\mu_q(x_t, x_0)$, we get:

$$\mu_q(x_t, x_0) = \sqrt{\bar{\alpha}_t}(1 - \alpha_{t-1})x_t + \sqrt{\bar{\alpha}_{t-1}}(1 - \alpha_t)\left(\frac{x_t - \sqrt{1 - \bar{\alpha}_t}\epsilon}{\sqrt{\bar{\alpha}_t}}\right)$$

Simplifying, we have:

$$\mu_q(x_t, x_0) = \frac{1}{\sqrt{\bar{\alpha}_t}} x_t - \frac{(1 - \alpha_t)}{\sqrt{1 - \bar{\alpha}_t}\sqrt{\bar{\alpha}_t}}\epsilon$$

Note that this is a known value.

Also we can write $\mu_\theta(x_t, t)$ as:

$$\mu_\theta(x_t, t) = \frac{1}{\sqrt{\bar{\alpha}_t}} x_t - \frac{(1-\alpha_t)}{\sqrt{1-\bar{\alpha}_t}\sqrt{\bar{\alpha}_t}} \epsilon_\theta(x_t, t)$$

where $\epsilon_\theta(x_t, t)$ is the model's prediction for the noise at time step t. This can be done because we can always reparametrize one Gaussian distribution to another Gaussian distribution.

Instead of learning $\mu_\theta(x_t, t)$(learn to predict the noise), we learn $\epsilon_\theta(x_t, t)$ and then use the above equation to get $\mu_\theta(x_t, t)$.

Now substitute $\mu_q(x_t, x_0)$ and $\mu_\theta(x_t, t)$ into T_3 we get:

$$T_3 = -\sum_{t=2}^{T} \mathbb{E}_{q(x_t|x_0)} \left[\frac{1}{2\sigma_\theta^2} \left\| \left(\frac{1}{\sqrt{\bar{\alpha}_t}} x_t - \frac{(1-\alpha_t)}{\sqrt{1-\bar{\alpha}_t}\sqrt{\bar{\alpha}_t}} \epsilon \right) - \left(\frac{1}{\sqrt{\bar{\alpha}_t}} x_t - \frac{(1-\alpha_t)}{\sqrt{1-\bar{\alpha}_t}\sqrt{\bar{\alpha}_t}} \epsilon_\theta(x_t, t) \right) \right\|_2^2 \right]$$

Simplifying the expression inside the norm, we get:

$$T_3 = -\sum_{t=2}^{T} \mathbb{E}_{q(x_t|x_0)} \left[\frac{1}{2\sigma_\theta^2} \left\| \frac{(1-\alpha_t)}{\sqrt{1-\bar{\alpha}_t}\sqrt{\bar{\alpha}_t}} (\epsilon - \epsilon_\theta(x_t, t)) \right\|_2^2 \right]$$

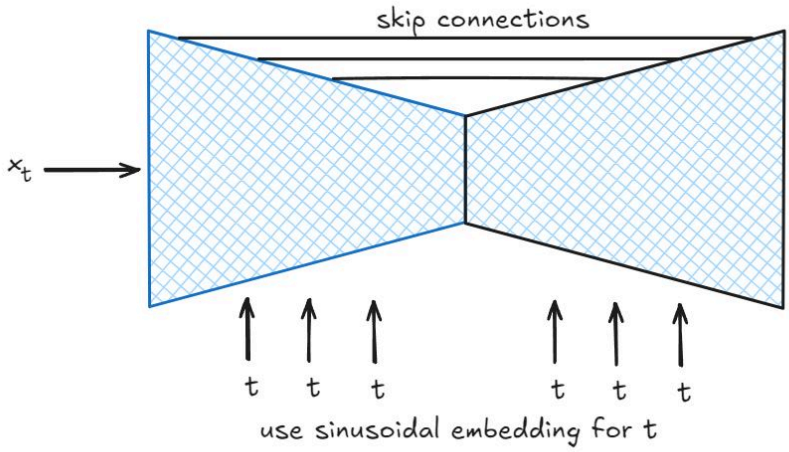
This can be further simplified to:

$$T_3 = -\sum_{t=2}^{T} \mathbb{E}_{q(x_t|x_0)} \left[\frac{(1-\alpha_t)^2}{2\sigma_\theta^2 (1-\bar{\alpha}_t)\bar{\alpha}_t} \left\| \epsilon - \epsilon_\theta(x_t, t) \right\|_2^2 \right]$$

Finally, we can express the proportionality:

$$T_3 \propto \sum_{t=2}^{T} \mathbb{E}_{q(x_t|x_0)} \left\| \epsilon - \epsilon_\theta(x_t, t) \right\|_2^2$$

Architecture



We do not give a scalar t to the network because the network will simply ignore it, hence we give time using sinusoidal embedding. The time embedding is concatenated with the residual path as:

Training

The training process for DDPM involves the following steps:

1. **Sample a random data point**: Select a data point x_0 from the training dataset.
2. **Sample a random time step**: Choose a random time step t from a uniform distribution over the range $[1, T]$.
3. **Sample noise**: Generate a noise sample $\epsilon \sim \mathcal{N}(0, I)$.
4. **Generate noisy data**: Create the noisy data x_t using the forward process equation:

$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon$$

5. **Predict the noise**: Use the model to predict the noise $\epsilon_\theta(x_t, t)$ at time step t.
6. **Compute the loss**: Calculate the loss as the mean squared error between the true noise ϵ and the predicted noise $\epsilon_\theta(x_t, t)$:

$$L_t = \|\epsilon - \epsilon_\theta(x_t, t)\|_2^2$$

7. **Backpropagation**: Perform backpropagation to compute the gradients of the loss with respect to the model parameters.
8. **Update the model parameters**: Use an optimizer (e.g., Adam) to update the model parameters based on the computed gradients.
9. **Repeat**: Iterate through the training dataset and repeat the above steps for a fixed number of epochs or until convergence.

By following these steps, the model learns to predict the noise at each time step, which can then be used to generate new samples by reversing the diffusion process.

Inference

The inference process in DDPM involves reversing the diffusion process to generate new samples. The steps are as follows:

1. **Sample from the prior**: Start by sampling $x_T \sim \mathcal{N}(0, I)$, which is the prior distribution.

2. **Reverse the diffusion process**: Sequentially sample x_{t-1} from x_t for $t = T, T-1, \ldots, 1$ using the learned reverse process $p_\theta(x_{t-1}|x_t)$.

   The reverse process is defined as:
   x_T to x_0 through intermediate steps $x_{T-1}, \ldots, x_1$.

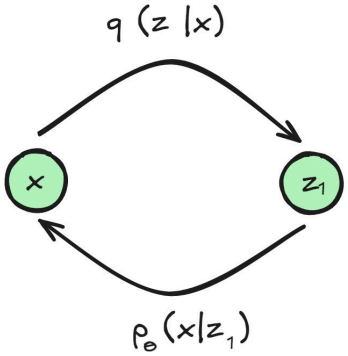
   $$x_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \epsilon_\theta(x_t, t) \right) + \sigma_t z$$

   where $z \sim \mathcal{N}(0, I)$ if $t > 1$, and $z = 0$ if $t = 1$ and $\sigma_t^2 = \beta_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t} \cdot (1-\alpha_t)$.

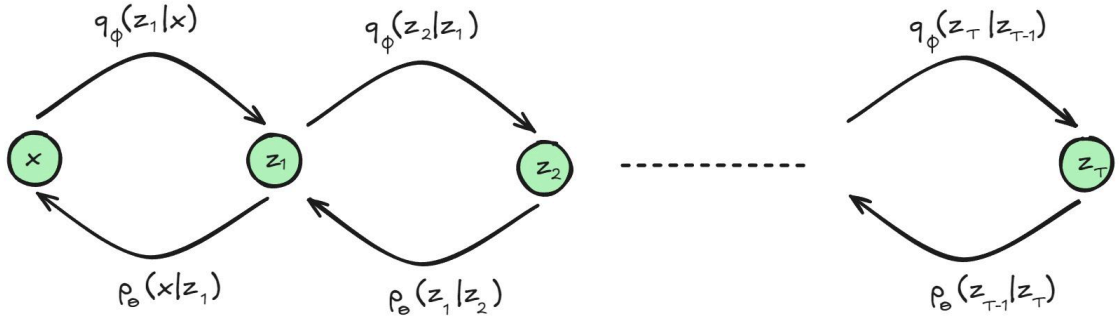
3. **Obtain the final sample**: The final sample x_0 is obtained after completing the reverse process.

Summarizing DDPM

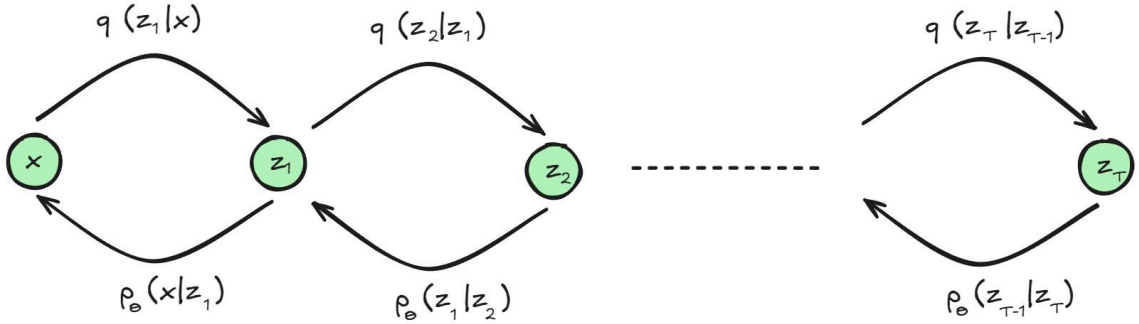
- VAE:



- Hierarchical VAE:



- DDPM(Hierarchical VAE with deterministic forward process):



Notice that ϕ has been removed from $q_\phi()$.

Also see that:

- The ELBO for a Variational Autoencoder (VAE) is given by:

$$\text{ELBO (VAE)} = \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p(x|z)}{q_\phi(z|x)}\right]$$

- For Denoising Diffusion Models (DDPM), the ELBO is expressed as:

$$\text{ELBO (DM)} = \mathbb{E}_{q(x_{1:T}|x_0)} \left[\log \frac{p(x_{0:T})}{q(x_{1:T}|x_0)} \right]$$

Score matching part 1

Sampling as optimization

In probabilistic modeling, sampling can be viewed as an optimization problem where we aim to find the most probable sample from a probability distribution. This can be formulated mathematically as:

$$x^* = \arg\max_x p(x)$$

Taking the log of the probability (which preserves the optimum due to monotonicity of log):

$$x^* = \arg\max_x \log p(x)$$

And converting to a minimization problem:

$$x^* = \arg\min_x -\log p(x)$$

This optimization problem can be solved using gradient descent, where we iteratively update our sample:

$$x_{t+1} = x_t + \alpha \cdot \nabla \log p(x_t)$$

where α is the learning rate that controls the size of each update step.

However, simple gradient descent can get stuck in local optima. A more robust approach is to use the Langevin dynamics equation:

$$x_{t+1} = x_t + \alpha \cdot \nabla_x \log p(x_t) + \sqrt{2\alpha}\epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

The Langevin equation has three key components:

- The current state x_t
- A gradient term $\nabla_x \log p(x_t)$ that guides the sample toward higher probability regions
- A stochastic noise term $\sqrt{2\alpha}\epsilon$ where $\epsilon \sim \mathcal{N}(0, I)$ that helps explore the full distribution and escape local optima

This stochastic differential equation defines a Markov chain whose stationary distribution converges to our target distribution $p(x)$, allowing us to generate samples that accurately represent the underlying probability distribution.

Score function

The score function is defined as the gradient of the log probability density with respect to the input data:

$$s(x) = \nabla_x \log p(x)$$

This function plays a crucial role in the Langevin dynamics equation that we use for sampling:

$$x_{t+1} = x_t + \alpha \cdot \nabla_x \log p(x_t) + \sqrt{2\alpha}\epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

Intuitively, the score function acts as a vector field that points in the direction of increasing probability density. At each point x, $s(x)$ provides the direction and magnitude of the “force” that pushes samples towards regions of higher probability in the data distribution.

Since we typically don’t have direct access to the true data distribution $p(x)$, we need to estimate the score function. This is done by training a neural network $s_\theta(x)$ to approximate $s(x)$. The neural network takes data points as input and outputs vectors that estimate the gradient of the log probability density at those points.

To train this score estimator network, we need an objective function that measures how well our approximation matches the true score function. This leads us to the concept of score matching, where we develop loss functions that allow us to train the network without requiring explicit knowledge of p(x).

Naive score matching or explicit score matching

*Explicit means something that is clearly and directly stated, leaving no room for confusion or doubt.

$$J_{ESM}(\theta) = \mathbb{E}_{p(x)}\left[\|\hat{s}(x;\theta) - s(x)\|^2\right]$$

where,

- $s(x)$ is the true score function.
- $\hat{s}(x)$ is the estimated score function.

This loss function is not tractable because we don't know the true score function.

Implicit score matching

*Implicit means something that is implied or suggested but not directly expressed.

*We will not prove the following derivation, but it is there in class notes.

The implicit score matching loss function can be written as:

$$J_{ISM}(\theta) = \mathbb{E}_{p(x)} \left[\frac{1}{2} \|\hat{s}(x;\theta)\|^2 + \text{tr}(\nabla_x \hat{s}(x;\theta)) \right] + C$$

As per the theorm this is equivalent to:

$$J_{ISM}(\theta) = J_{ESM}(\theta) + C$$

Where:

- $\hat{s}(x;\theta)$ is the estimated score function
- $\text{tr}(\nabla_x \hat{s}(x;\theta))$ is the trace of the Jacobian of the estimated score function
- C is a constant term independent of θ

Also following assumptions were made for proving the theorm:

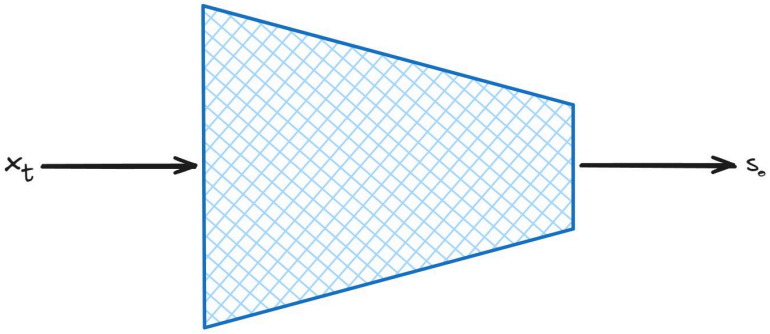
1. $p(x)$ is differentiable
2. $\mathbb{E}_{p(x)}[\|\nabla_x \log p(x)\|^2] < \infty$ for any θ &
   $\mathbb{E}_{p(x)}[\|\hat{s}(x;\theta)\|^2] < \infty$ for any θ
3. $p(x)\hat{s}(x;\theta) \to 0$ for any θ as $\|x\| \to \infty$

This formulation allows us to optimize the score function without explicitly knowing the true score function $\nabla_x \log p(x)$. The trace term $\text{tr}(\nabla_x \hat{s}(x;\theta))$ comes from the divergence of the score function, which is equal to $\nabla_x \cdot (\nabla_x \log p(x)) = \nabla_x^2 \log p(x)$.

Score matching part 2

Recap

- $J_{ESM}(\theta) = \mathbb{E}_{p(x)}\left[\frac{1}{2}\|\hat{s}(x;\theta) - \nabla_x \log p(x)\|^2\right]$
- $J_{ISM}(\theta) = \mathbb{E}_{p(x)}\left[\frac{1}{2}\|\hat{s}(x;\theta)\|^2 + \text{tr}(\nabla_x \hat{s}(x;\theta))\right] + C$
- Theorem: $J_{ISM}(\theta) = J_{ESM}(\theta) + C$
- $\theta^* = \arg\min_\theta J_{ISM}(\theta)$
- $x_{t+1} = x_t + \alpha \cdot \hat{s}(x_t;\theta^*) + \sqrt{2\alpha}\epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$
- $\hat{s}(x;\theta)$ modelled using a neural network



There is a problem with calculating the trace term

The implicit score matching loss function, as we discussed earlier, is given by:

$$J_{ISM}(\theta) = \mathbb{E}_{p(x)}\left[\frac{1}{2}\|\hat{s}(x;\theta)\|^2 + \text{tr}(\nabla_x \hat{s}(x;\theta))\right] + C$$

However, there's a significant challenge in calculating this loss function, particularly with the trace term:

$$\text{tr}(\nabla_x \hat{s}(x;\theta))$$

The problem arises because:

1. $\nabla_x \hat{s}(x;\theta)$ is the Hessian of $\log p(x)$ with respect to the input x.
2. For high-dimensional data (which is common in many applications), this Hessian becomes extremely large.

3. Computing the trace of this large Hessian matrix is computationally expensive and often intractable.

This computational challenge motivates the need for alternative approaches to score matching, which we'll explore in the following sections.

Projected score matching

Projected score matching is a technique that involves projecting the score function onto a lower-dimensional subspace.

The projected score matching loss function can be expressed as:

$$J_{PSM}(\theta) = \frac{1}{2}\mathbb{E}_v\mathbb{E}_{p(x)}\left[\|v^\top \hat{s}_\theta(x) - v^\top s(x)\|^2\right]$$

Where:

- $\hat{s}_\theta(x)$ is the estimated score function
- $s(x)$ is the true score function
- v is a random projection vector

This formulation allows us to estimate the score function without explicitly computing the trace of the Hessian, which is computationally expensive in high-dimensional spaces. Instead, we project the score function onto random vectors v, effectively reducing the dimensionality of the problem.

However this cannot be computed(similar to the problem with calculating J_{ISM}) because we do not know $s(x)$

Hence we need to find an alternative way to compute the loss function which leads us to sliced score matching

Sliced Score Matching

The SSM loss function can be expressed as:

$$J_{SSM}(\theta) = \mathbb{E}_v\mathbb{E}_{p(x)}\left[\frac{1}{2}(v^\top \hat{s}_\theta(x))^2 + v^\top(\nabla_x \hat{s}_\theta(x))v\right]$$

Where:

- $\hat{s}_\theta(x)$ is the estimated score function
- v is a random unit vector sampled from a uniform distribution on the unit sphere

Key aspects of Sliced Score Matching:

1. Random Projections: Like projected score matching, SSM uses random projections to reduce dimensionality. However, SSM specifically uses unit vectors sampled from a uniform distribution on the unit sphere.

2. Efficient Computation: The second term $v^\top(\nabla_x \hat{s}_\theta(x))v$ can be computed efficiently using automatic differentiation, avoiding the need to explicitly calculate the full Hessian matrix.

Theorm

The relationship between Projected Score Matching (PSM) and Sliced Score Matching (SSM) can be expressed through the following theorem:

$$J_{PSM}(\theta) = J_{SSM}(\theta) + C$$

Where:

- $J_{PSM}(\theta)$ is the loss function for Projected Score Matching
- $J_{SSM}(\theta)$ is the loss function for Sliced Score Matching
- C is a constant term independent of θ

Following assumptions were made to prove the theorem:

- A1. $p(x)$ and $s(x)$ are differentiable
- A2. $\mathbb{E}_{p(x)}[\|\nabla_x \log p(x)\|^2] < \infty$ and $\mathbb{E}_{p(x)}[\|\hat{s}(x;\theta)\|^2] < \infty$ for any θ
- A3. $\lim_{\|x\|\to\infty} p(x)\hat{s}(x;\theta) = 0$ for any θ
- A4. $\mathbb{E}_{p_v}[(\|v\|^2)] < \infty$ and $\mathbb{E}_{p_v}[(v^T v)] > 0$ (positive definite) - *new assumption*

This theorem demonstrates that under certain conditions, optimizing the SSM objective is equivalent to optimizing the PSM objective, up to a constant difference. This relationship provides a theoretical foundation for the use of Sliced Score Matching as an efficient alternative to Projected Score Matching.

How to compute J_{SSM}

J_{SSM} can be computed using Monte Carlo estimates. Given a datapoint x_i take m projections and approximate the SSM loss function as follows:

$$J_{SSM}(\theta) \approx \frac{1}{NM} \sum_{i=1}^{N} \sum_{j=1}^{M} \left[\frac{1}{2} (v_j^\top \hat{s}_\theta(x_i))^2 + v_j^\top (\nabla_x \hat{s}_\theta(x_i)) v_j \right]$$

Where:

- N is the number of data samples
- M is the number of random projection vectors
- x_i are the data samples
- v_j are random unit vectors sampled from a uniform distribution on the unit sphere
- $\hat{s}_\theta(x_i)$ is the estimated score function for sample x_i

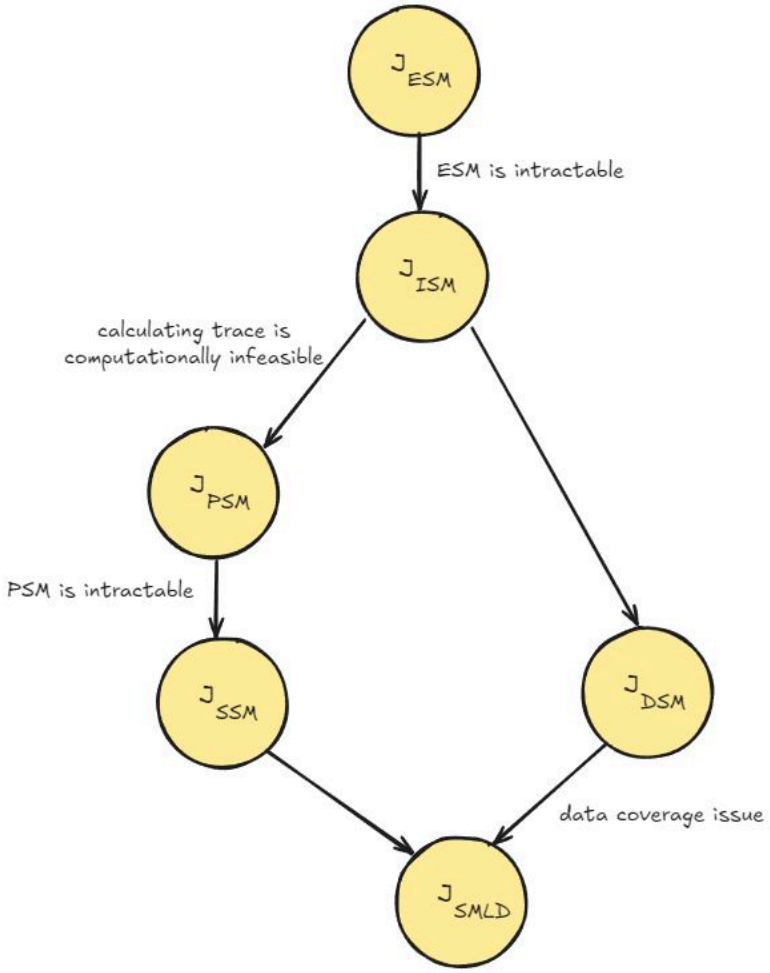
This Monte Carlo approximation allows us to practically compute the SSM loss using a finite number of data samples and random projections, making it feasible for optimization in machine learning models.

Score matching part 3

Recap

- Langevin equation: $x_{t+1} = x_t + \alpha \cdot \nabla_x \log p(x_t) + \sqrt{2\alpha}\epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$
- score function: $\nabla_x \log p(x) = s(x)$
- $J_{ESM}(\theta) = \mathbb{E}_{p(x)} \left[\frac{1}{2} \| \hat{s}(x;\theta) - \nabla_x \log p(x) \|^2 \right]$
- $J_{ISM}(\theta) = \mathbb{E}_{p(x)} \left[\frac{1}{2} \| \hat{s}(x;\theta) \|^2 + \mathrm{tr}(\nabla_x \hat{s}(x;\theta)) \right] + C$
- Theorem: $J_{ISM}(\theta) = J_{ESM}(\theta) + C$
- $J_{PSM}(\theta) = \frac{1}{2} \mathbb{E}_v \mathbb{E}_{p(x)} \left[\| v^\top \hat{s}_\theta(x) - v^\top s(x) \|^2 \right]$
- $J_{SSM}(\theta) = \mathbb{E}_v \mathbb{E}_{p(x)} \left[\frac{1}{2} (v^\top \hat{s}_\theta(x))^2 + v^\top (\nabla_x \hat{s}_\theta(x)) v \right]$
- Theorem: $J_{PSM}(\theta) = J_{SSM}(\theta) + C$

Big picture plan



Denoising score matching

Denoising score matching (DSM) is an alternative approach to score matching. It introduces an auxiliary variable and works with conditional scores, which offers several advantages.

Let's start with the DSM objective:

$$J_{DSM}(\theta) = \frac{1}{2}\mathbb{E}_{p(x,x')}\left[\|\hat{s}_\theta(x) - \nabla_x \log p(x|x')\|^2\right]$$

Where:

- x is a sample from the true data distribution $p(x)$
- x' is auxiliary variable obtained by adding Gaussian noise to x: $x' = x + \epsilon$ where $\epsilon \sim \mathcal{N}(0, \sigma^2)$
- $\hat{s}_\theta(x)$ is the estimated score function that we want to learn
- $\nabla_x \log p(x|x')$ is the conditional score, which has an analytical form for Gaussian noise

The DSM objective allows us to learn the score function $\nabla_x \log p(x)$ by:

1. Generating training pairs (x, x') by adding noise to data samples
2. Leveraging the known form of $p(x|x')$ for Gaussian noise
3. Avoiding direct computation of the intractable normalization constant

Key aspects of Denoising Score Matching:

1. Auxiliary Variable: DSM introduces x', which is a noisy version of the original data point x. This allows us to work with conditional distributions.

2. Conditional Score: Instead of estimating the score of the data distribution directly, DSM estimates the conditional score $\nabla_x \log p(x|x')$.

3. Gaussian Noise: Typically, Gaussian noise is added to create x'. This has two significant benefits:

    - We can add Gaussian noise to our data easily.
    - The conditional distribution $p(x|x')$ becomes Gaussian, which has a known analytical form for its score.

4. Theorem: There's an important relationship between DSM and the original score matching objective:

$$J_{ESM}(\theta) = J_{DSM}(\theta) + C$$

Where C is a constant independent of θ. This theorem shows that optimizing the DSM objective is equivalent to optimizing the original score matching objective.

The benefits of using conditional scores and Gaussian noise make DSM a powerful and practical approach to score matching, addressing many of the computational challenges faced by earlier methods.

Continue DSM

Let's continue exploring Denoising Score Matching (DSM) in more detail. The original DSM objective was:

$$J_{DSM}(\theta) = \frac{1}{2}\mathbb{E}_{p(x,x')}\left[\|\hat{s}_\theta(x) - \nabla_x \log p(x|x')\|^2\right]$$

To align with the lecture notation make we'll make the following notation changes:

- Original datapoint x becomes $\tilde{x}$
- Noisy version x' becomes x

This gives us:

$$J_{DSM}(\theta) = \frac{1}{2}\mathbb{E}_{p(\tilde{x},x)}\left[\|\hat{s}_\theta(\tilde{x}) - \nabla_{\tilde{x}} \log p(\tilde{x}|x)\|^2\right]$$

We can further expand this by explicitly defining the perturbation distribution:

$$J_{DSM}(\theta) = \frac{1}{2}\mathbb{E}_{p(x)}\mathbb{E}_{q_\sigma(\tilde{x}|x)}\left[\|s_\theta(\tilde{x}) - \nabla_{\tilde{x}} \log q_\sigma(\tilde{x}|x)\|^2\right]$$

Key components:

1. Perturbation Distribution ($q_\sigma(\tilde{x}|x)$):
    - Defined as $q_\sigma(\tilde{x}|x) = \mathcal{N}(\tilde{x}|x, \sigma^2 I)$
    - Generated via $\tilde{x} = x + \sigma\epsilon$ where $\epsilon \sim \mathcal{N}(0, I)$
    - Adds controlled Gaussian noise to create noisy versions of data points

2. Score of Perturbation Kernel:
   The term $\nabla_{\tilde{x}} \log q_\sigma(\tilde{x}|x)$ can be derived analytically:

$$\begin{aligned}\nabla_{\tilde{x}} \log q_\sigma(\tilde{x}|x) &= -\nabla_{\tilde{x}} \frac{(\tilde{x}-x)^2}{2\sigma^2} \\ &= -\frac{\tilde{x}-x}{\sigma^2}\end{aligned}$$

Substituting this back into our objective:

$$J_{DSM}(\theta) = \frac{1}{2}\mathbb{E}_{p(x)}\mathbb{E}_{q_\sigma(\tilde{x}|x)}\left[\|s_\theta(\tilde{x}) + \frac{\tilde{x}-x}{\sigma^2}\|^2\right]$$

Reparameterization for Efficient Training

We can make this objective more computationally tractable through reparameterization. Instead of sampling $\tilde{x}$ directly, we express it in terms of x and ϵ:

$$\tilde{x} = x + \sigma\epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

This transforms our objective into:

$$J_{DSM}(\theta) = \frac{1}{2}\mathbb{E}_{p(x)}\mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)}\left[\|s_\theta(x+\sigma\epsilon) + \frac{\epsilon}{\sigma}\|^2\right]$$

Further simplification yields:

$$J_{DSM}(\theta) = \frac{1}{2\sigma^2}\mathbb{E}_{p(x)}\mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)}\left[\|\sigma s_\theta(x+\sigma\epsilon) + \epsilon\|^2\right]$$

In the lecture, this was presented in a slightly different but equivalent form:

$$\mathbb{E}_{x,\epsilon}\left(\frac{1}{2\sigma^2}\|\epsilon_\theta(\tilde{x}) - \epsilon\|^2\right)$$

Critical Properties of this Formulation:

1. Scale-Aware Training: The $\frac{1}{2\sigma^2}$ factor naturally weights the loss more heavily at smaller noise levels. This ensures accurate denoising across different scales.

2. Direct Noise Prediction: Instead of predicting scores, the model learns to predict the noise directly.

Data Coverage Problem

A fundamental challenge in score matching is ensuring accurate score estimation across the entire data distribution $p(x)$, particularly in regions of low density. This is known as the data coverage problem.

Noise Conditional Score Networks (NCSNs) address this challenge through a multi-scale approach:

1. Noise Scale Hierarchy:

$$\{\sigma_i\}_{i=1}^{L}, \quad \text{where } \sigma_1 < \sigma_2 < ... < \sigma_L$$

   - L typically ranges from 10 to several hundred scales
   - $\sigma_1 = \sigma_{min}$ (minimal noise)
   - $\sigma_L = \sigma_{max}$ (maximal noise)

2. Distribution Bridging:
   Two key conditions are enforced:

   - $p_{\sigma_{min}}(x) \approx p(x)$ (preserves original distribution)
   - $p_{\sigma_{max}}(x) \approx \mathcal{N}(0, I)$ (approaches Gaussian)

3. Noise Application:
   For each scale σ_i:

$$p_{\sigma_i}(\tilde{x}|x) = \mathcal{N}(\tilde{x}|x, \sigma_i^2 I)$$

4. Score Estimation:
   The model learns scale-conditional scores:

$$s_\theta(x, \sigma_i) \approx \nabla_x \log p_{\sigma_i}(x)$$

5. Scale-Specific Loss:

$$\mathcal{L}(\theta, \sigma_i) = \frac{1}{2}\mathbb{E}_{p(x)}\mathbb{E}_{q_{\sigma_i}(\tilde{x}|x)}\left[\left\| s_\theta(\tilde{x}, \sigma_i) + \frac{\tilde{x} - x}{\sigma_i^2} \right\|^2\right]$$

6. Combined Training Objective:

$$\mathcal{L}(\theta; \{\sigma_i\}_{i=1}^{L}) = \frac{1}{L}\sum_{i=1}^{L} \lambda(\sigma_i)\mathcal{L}(\theta, \sigma_i)$$

where $\lambda(\sigma_i)$ weights different scales' contributions

Benefits of this Multi-Scale Approach:

1. Comprehensive Coverage:
    - High-density regions: Accurate modeling with small σ_i
    - Low-density regions: Stable estimation with large σ_i
2. Smooth Interpolation:
    - Gradual transition between noise levels
    - Continuous coverage of the data manifold
3. Training Stability:
    - Different scales provide complementary learning signals
    - Reduced sensitivity to individual scale choices

This approach enables NCSNs to effectively model complex data distributions while maintaining stability and accuracy across all regions of the data space.

Difference between SMLD and DDPM

Note that: NCSN: Noise Conditional Score Network is same as SMLD: Score matching Langevin Dynamics

Inference

The inference processes also differ between SMLD and DDPM:

1. SMLD (Noise Conditional Score Networks):
    - We start with a sequence of noise scales arranged from largest to smallest:

$$\{\sigma_i\}_{i=1}^{L}, \quad \sigma_1 > \sigma_2 > ... > \sigma_L \quad (L \approx 10)$$

    - Start with random Gaussian noise:

$$x_0 \sim \mathcal{N}(0, \sigma_1^2 I)$$

    - For each noise scale σ_i from $i = 1$ to L:
        - Perform K Langevin steps ($K \approx 100$) at fixed σ_i:

$$x_{k+1} = x_k + \alpha s_\theta(x_k, \sigma_i) + \sqrt{2\alpha}\epsilon_k, \quad \epsilon_k \sim \mathcal{N}(0, I)$$

          for $k = 0, 1, ..., K-1$
        - Use final sample x_K as starting point for next scale σ_{i+1}
    - This gradually denoises the image by:
        1. Getting a good sample at high noise levels (large σ)
        2. Using that sample as initialization for next lower noise level
        3. Repeating until reaching smallest noise scale σ_L
2. DDPM (Denoising Diffusion Probabilistic Models):
    - Uses a deterministic reverse process
    - Starting from random Gaussian noise, applies:

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}} \epsilon_\theta(x_t, t) \right) + \sigma_t z$$

    - Where:
        - $\alpha_t = 1 - \beta_t$
        - $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$
        - $\sigma_t^2 = \beta_t$ (simplified version)
        - $z \sim \mathcal{N}(0, I)$
    - One step per noise level
    - Follows the reverse Markov chain exactly

Key differences:

- SMLD requires multiple Langevin steps per noise level, while DDPM uses single deterministic steps
- SMLD's sampling is more computationally intensive due to multiple steps per level
- DDPM's reverse process is more structured and follows the exact reverse of the forward process
- SMLD can be less stable due to Langevin dynamics, while DDPM's deterministic steps are more stable

Forward process

The forward processes of SMLD and DDPM differ in how they add noise to the data:

1. SMLD (Noise Conditional Score Networks):
    - Uses a predefined sequence of noise scales: $\{\sigma_i\}_{i=1}^{L}$
    - Forward process:

$$x_{\sigma_{i+1}} = x_{\sigma_i} + \sigma_{i+1}\epsilon_{i+1}, \quad \epsilon_i, \epsilon_{i+1} \sim \mathcal{N}(0, I)$$

$$x_{\sigma_i} = x + \sigma_i \epsilon_i, \quad \epsilon_i \sim \mathcal{N}(0, I)$$

$$x_{\sigma_{i+1}} = x_{\sigma_i} + k\epsilon \quad (\text{derivation skip})$$

    - The value of k can be derived by considering the variance:

$$\text{Var}(x_{\sigma_{i+1}}) = \text{Var}(x_{\sigma_i} + k\epsilon)$$

$$\sigma_{i+1}^2 = \sigma_i^2 + k^2$$

- Solving for k:

$$k = \sqrt{\sigma_{i+1}^2 - \sigma_i^2}$$

- Therefore the complete forward process step is:

$$x_{\sigma_{i+1}} = x_{\sigma_i} + \sqrt{\sigma_{i+1}^2 - \sigma_i^2}\epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

- This ensures the variance increases smoothly between noise scales

2. DDPM (Denoising Diffusion Probabilistic Models):

- Uses a Markov chain of diffusion steps: $\{x_t\}_{t=0}^T$
- Forward process:

$$x_{t+1} = \sqrt{\alpha_t}x_t + \sqrt{1 - \alpha_t}\epsilon$$

where $\epsilon \sim \mathcal{N}(0, I)$

Conditional score matching

When we had unlabeled data, we estimated the score function:

$$s(x) = \nabla_x \log p(x)$$

Now, we have labeled data ie $D = \{(x_i, y_i)\}_{i=1}^N$ we instead estimate the conditional score function:

$$s(x|y) = \nabla_x \log p(x|y)$$

Where:

- $s(x)$ is the unconditional score function
- $s(x|y)$ is the conditional score function
- $p(x)$ is the data distribution
- $p(x|y)$ is the conditional data distribution given some condition y

This can be done in 2 ways:

- Classifier guided score matching
- Classifier free guidance

Classifier guided score matching

- Bayes rule

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)}$$

- Take log on both sides

$$\log p(x|y) = \log p(y|x) + \log p(x) - \log p(y)$$

- Take gradient on both sides

$$\nabla_x \log p(x|y) = \nabla_x \log p(y|x) + \nabla_x \log p(x)$$

- The gradient of the log likelihood is the score function, so we get:

$$s(x|y) = s(y|x) + s(x)$$

Note that $\nabla_x \log p(y|x)$ is not a score because the gradient w.r.t x and not y. However we write it like this for convenience.

- Practically, we can use a classifier to estimate $s(y|x)$
  - Train a classifier $p_\phi(y|x)$ using cross entropy loss
  - The classifier is trained independently from the score model(after training the score model)
  - The gradient $\nabla_x \log p_\phi(y|x)$ gives us $s_\phi(y|x)$
  - The langevin dynamics equation becomes:

$$x_{t+1} = x_t + \alpha(s_\phi(y|x) + s_\theta(x)) + \sqrt{2\alpha}\epsilon$$

    where:
    - $s_\phi(y|x)$ is the classifier guidance term
    - $s_\theta(x)$ is the score model
    - α is the step size
    - $\epsilon \sim \mathcal{N}(0, I)$ is random noise
  - This results in a stable diffusion process in latent space

Classifier free guidance

Train a conditional score model $p(x|y)$ with conditioning dropout:

- During training, for each batch:
  - With probability p (e.g. $p = 0.1$):
    - Replace the conditioning y with a null token (e.g. zero vector or special embedding)
    - Train the model to predict $s_\theta(x|\text{null})$
  - With probability $(1 - p)$:
    - Keep the original conditioning y
    - Train the model to predict $s_\theta(x|y)$
- This training strategy results in a single model that can:
  - Act as a conditional score model $s_\theta(x|y)$ when given real conditioning
  - Act as an unconditional score model $s_\theta(x)$ when given the null token
  - The unconditional model emerges from training with dropped conditions
- During sampling:

- Can interpolate between conditional and unconditional by:

$$s_\theta(x|y)_{CFG} = (1+w)s_\theta(x|y) - ws_\theta(x|\text{null})$$

  where w controls the classifier-free guidance strength

Precursor: Stochastic Differential Equations (SDEs)

Motivation

We explored different sampling methods for generative models:

1. Langevin dynamics (LE):

$$x_{t+1} = x_t + \alpha \nabla_x \log p(x_t) + \sqrt{2\alpha}\epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

2. DDPM (Denoising Diffusion Probabilistic Models):

$$x_{t+1} = \sqrt{1-\beta_t}x_t + \sqrt{\beta_t}\epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

3. SMLD (Score Matching with Langevin Dynamics):

$$x_{t+1} = x_t + \sqrt{\sigma_{t+1}^2 - \sigma_t^2}\epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

A natural question arises: What is the unifying mathematical framework that connects all these sampling methods?

The answer lies in Stochastic Differential Equations (SDEs). We will show that all these methods are actually discrete approximations of continuous-time SDEs. Understanding this connection provides several benefits:

1. A unified theoretical framework for analyzing different sampling methods
2. Better understanding of the relationship between different approaches
3. Potential for developing new sampling methods by working with SDEs directly

Discrete sampling steps as approximations of continuous processes

Let's start by understanding how discrete sampling steps can be viewed as approximations of continuous processes with help of examples.

Example: From Discrete to Continuous Time

To understand how discrete sampling steps can be viewed as approximations of continuous processes, let's consider the following example where the discrete process is defined as:

$$x_i = (1 - \frac{\beta}{2})x_{i-1}$$

Also, we have:

- $\Delta t = \frac{1}{N}$: The time step size, where N is the total number of steps
- $i = \frac{t}{\Delta t}$: The current step index, where t is the continuous time variable
- β_t : The noise schedule at time t
- x_i : The state at step i
- x_{i-1} : The state at previous step
- $\epsilon \sim \mathcal{N}(0, I)$: Random noise sampled from standard normal distribution
- I : Identity matrix

Let's convert this discrete process to continuous time:

1. First, we can write x_i in terms of continuous time t:

$$x_i = x(\frac{i}{N}) = x(t)$$

2. Similarly for the next step:

$$x_{i-1} = x(\frac{i-1}{N}) = x(t - \Delta t)$$

3. The discrete equation can be rewritten as:

$$x(t + \Delta t) = (1 - \frac{\beta \Delta t}{2})x(t)$$

   Note: We multiply β by Δt because β represents a rate of change per unit time. When we discretize time into small steps Δt, we need to scale β accordingly to get
   the correct amount of change for that time step. Without this scaling, the discrete steps would
   not properly approximate the continuous process as $\Delta t \to 0$.

4. This is equivalent to:

$$x(t + \Delta t) - x(t) = -\frac{\beta \Delta t}{2} x(t)$$

5. Dividing both sides by Δt:

$$\frac{x(t + \Delta t) - x(t)}{\Delta t} = -\frac{\beta}{2} x(t)$$

6. Taking the limit as $\Delta t \to 0$:

$$\lim_{\Delta t \to 0} \frac{x(t + \Delta t) - x(t)}{\Delta t} = -\frac{\beta}{2} x(t)$$

7. The left side is the definition of the derivative, so we get:

$$\frac{dx}{dt} = -\frac{\beta}{2} x(t)$$

8. This ODE has the solution:

$$x(t) = e^{-\frac{\beta}{2} t}$$

   This represents exponential decay of the state over time.

This is our continuous-time ordinary differential equation (ODE).

Another example

Let's look at another example of converting a discrete equation to continuous form:

1. Starting with the discrete equation:

$$x_i = x_{i-1} - \beta_{i-1} \nabla f(x_{i-1})$$

   where β is the gradient descent step size

2. Make it continuous by substituting:

$$x_i = x(\frac{i}{N}), \quad \Delta t = \frac{1}{N}, \quad \beta_{i-1} = \beta(t) \Delta t$$

3. This gives us:

$$x(t + \Delta t) = x(t) - \beta(t) \Delta t \nabla f(x(t))$$

4. Rearranging to get the differential form:

$$\frac{x(t+\Delta t) - x(t)}{\Delta t} = -\beta(t)\nabla f(x(t))$$

5. Taking the limit as $\Delta t \to 0$:

$$\frac{dx(t)}{dt} = -\beta(t)\nabla f(x(t))$$

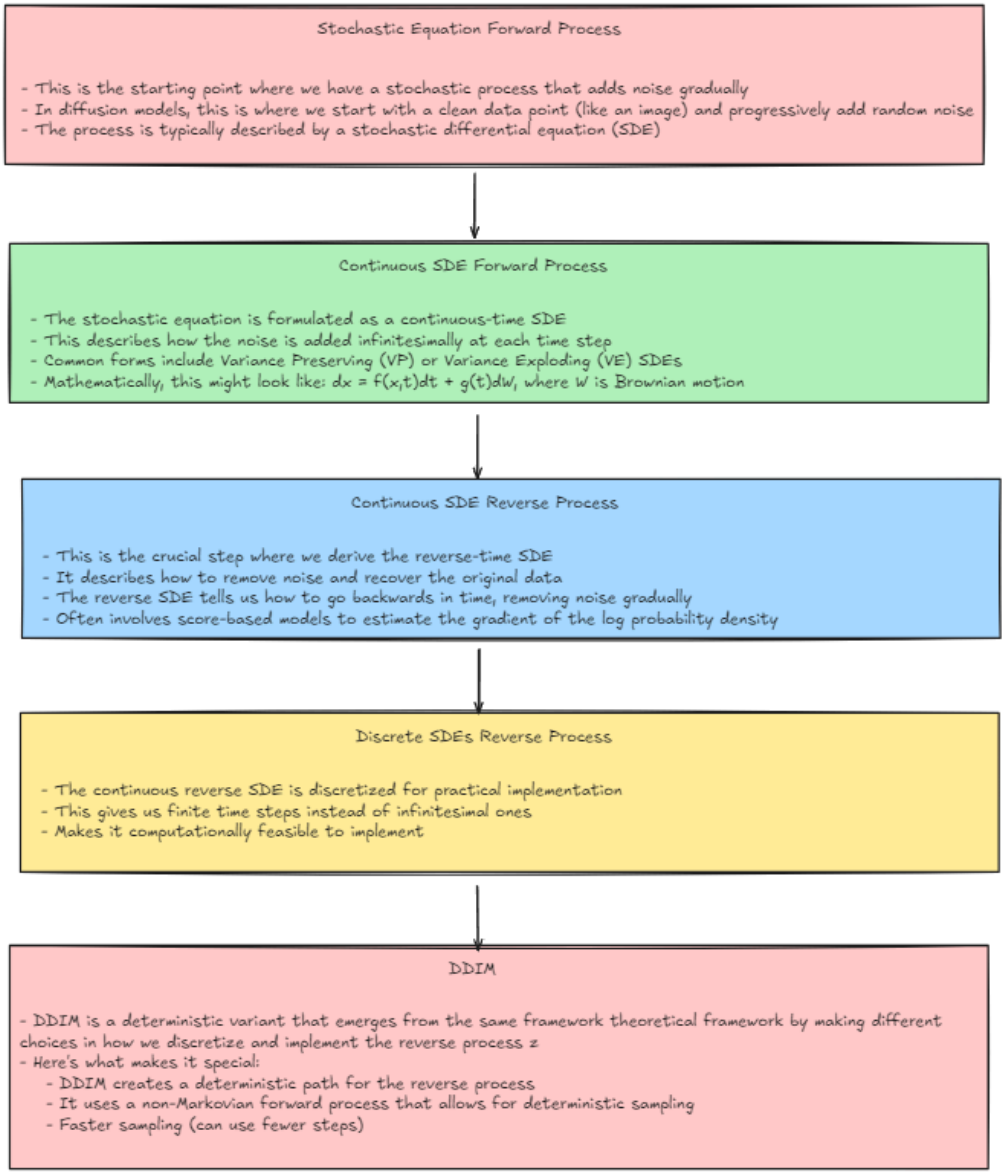
6. Finally, we can write this in the more compact differential form:

$$dx = -\beta(t)\nabla f(x(t))dt$$

This demonstrates how a discrete gradient descent step can be viewed as a discretization of a continuous ordinary differential equation (ODE).

Stochastic Differential Equations (SDEs)

Big picture of what we are going to do:



Big picture of SDEs

This topic comes under stochastic calculus, which deals with functions whose derivatives yield random vectors rather than deterministic points.

In traditional calculus, when we take the derivative of a function at a point, we get a fixed value ie a scalar that represents the instantaneous rate of change. However, in stochastic calculus, the derivative at any point is a random variable.

Formal definition of SDEs

Now let's understand the formal definition of SDEs:

1. An Ordinary Differential Equation (ODE) can be written as:

$$\frac{dx(t)}{dt} = f(x,t)$$

   or in differential form:

$$dx = f(x,t)dt$$

2. A Stochastic Differential Equation (SDE) adds a noise term:

$$\frac{dx(t)}{dt} = f(x,t) + g(x,t)dW(t)$$

   or in differential form:

$$dx = f(x,t)dt + g(x,t)dB_t$$

   where:

   - $f(x,t)$ is called the drift term (deterministic component that describes the average behavior)
   - $g(x,t)$ is called the diffusion term (controls the magnitude of random fluctuations)
   - $W(t)$ is a Wiener process (Brownian noise) - a continuous-time stochastic process
   - B_t is a Brownian motion, dB_t is the increment of Brownian motion where

$$dB_t \sim N(0, dt)$$

$$dB_t = \sqrt{dt}\epsilon \quad \text{where} \quad \epsilon \sim N(0,1)$$

Key points about SDEs:

- An ODE is a special case where $g(x, t) = 0$
- The SDE is a stochastic process

We can define important properties of an SDE:

1. Mean/drift:

$$m(t) = \mathbb{E}[x(t)] \leftarrow f(x, t)$$

2. Variance:

$$v(t) = \mathrm{Var}(x(t)) \leftarrow g(x, t)$$

This provides one way to analyze and understand SDEs.

Brownian Motion definition

A standard Brownian Motion (BM) is a random process $X = \{X_t : t \in [0, \infty)\}$ with state space $\mathbb{R}$ ie $X_t \in \mathbb{R}$ that satisfies the following properties:

1. $X_0 = 0$ with probability 1
    - *The process always starts at zero*
2. Has stationary increments:
    - For any $t \in [0, \infty)$, the distribution of $X_t - X_s$ only depends on the time difference $(t - s)$
    - *The behavior of changes doesn't depend on when we start observing*
3. Has independent increments:
    - For any times $t_1 < t_2 < ... < t_n \in [0, \infty)$
    - The increments $X_{t_2} - X_{t_1}, X_{t_3} - X_{t_2}, ..., X_{t_n} - X_{t_{n-1}}$ are independent
    - For any non-overlapping time intervals
    - *What happens in one time period doesn't affect what happens in another*
4. For any $t \in [0, \infty)$:
   $X_t \sim \mathcal{N}(0, t)$ (Normal distribution with mean 0 and variance t)

    - *The position at any time follows a normal distribution with variance growing linearly with time*

5. With probability 1, $t \mapsto X_t$ is continuous on $[0, \infty)$

    - *The path is continuous - no sudden jumps*

Out of the above properties, we will use only the following:

- $X_t - X_s \sim \mathcal{N}(0, t-s)$
    - *The change over any interval follows a normal distribution*
- $X_t \sim \mathcal{N}(0, t)$
    - *The position at any time follows a normal distribution*
- $X_{t+s} - X_s \perp X_s$ (independence of increments)
    - *Future changes are independent of current position*

Lets talk about B_t - What it is not

Let's consider how Brownian motion behaves when we look at small time intervals:

Consider $B_{t+h} - B_t$ where:

- $h > 0$: Looking at a small time step
- $B_{t+h} - B_t \sim B_h$ (from stationarity)
- $h < 0$: $B_{t+h} - B_t \sim -B_{-h}$ (from symmetry)

When we combine both:
$B_{t+h} - B_t \sim B_{|h|}$

Therefore:
$\frac{B_{t+h}-B_t}{h} \sim \mathcal{N}(0, |h|)$

$$\frac{B_{t+h} - B_t}{h} \sim \frac{1}{h}\mathcal{N}(0, |h|) = \mathcal{N}(0, \frac{1}{h})$$

Therefore, the variance is:

$$\mathrm{Var}(\frac{B_{t+h} - B_t}{h}) = \mathrm{Var}(\frac{1}{h}B_{|h|}) = \frac{1}{h^2}\mathrm{Var}(B_{|h|}) = \frac{1}{h^2}|h| = \frac{1}{h}$$

Taking the limit as h approaches 0:

$$\lim_{h \to 0} \text{Var}\left(\frac{B_{t+h} - B_t}{h}\right) = \lim_{h \to 0} \frac{1}{h} = \infty$$

Theorem: With probability 1, Brownian motion B_t is nowhere differentiable on $[0, \infty)$.

This means that **Brownian motion is not differentiable** this is because:

1. The variance of the rate of change becomes infinite as we look at smaller time intervals
2. This means the rate of change becomes arbitrarily large and fluctuates wildly
3. No well-defined derivative can exist under these conditions

Hence, $\frac{B_{t+h} - B_t}{h}$ does not converge in distribution (weakest form of convergence).

Lets talk about B_t - What it is

While we cannot directly talk about the derivative of B_t (since Brownian motion is not differentiable in the classical sense), we can still analyze it in terms of finite differences and perturbations.

Let's define a perturbation W_t^ϵ as:

$$W_t^\epsilon = \frac{B_{t+\epsilon} - B_t}{\epsilon}$$

where ϵ represents a small time increment.

Now, consider the limit as $\epsilon \to 0$:

$$W_t = \lim_{\epsilon \to 0} W_t^\epsilon = \lim_{\epsilon \to 0} \frac{B_{t+\epsilon} - B_t}{\epsilon}$$

However, since B_t is nowhere differentiable, W_t does not exist in the usual sense. Instead, this expression can be interpreted in a distributional sense, often written informally as:

$$W_t \triangleq \frac{dB_t}{dt}$$

Understanding the "White Noise" Process
Note: This is an abuse of notation since B_t is not differentiable. Instead, W_t represents

what we call white noise, which can be thought of as the “derivative” of Brownian motion in a generalized sense.

White noise, W_t, is a random process with constant power spectral density and is formally characterized by:

$$W_t \sim \frac{1}{\sqrt{dt}} \cdot \mathcal{N}(0,1)$$

This scaling factor $\frac{1}{\sqrt{dt}}$ ensures that the variance of W_t scales inversely with the time increment dt, making its variance effectively infinite as $dt \to 0$ — a key property of white noise.

Increment Representation of Brownian Motion
Using this notion of white noise, we can more rigorously express the increment of Brownian motion B_t over a small interval dt as:

$$dB_t = \sqrt{dt} \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0,1)$$

This expression indicates that the increment dB_t over a tiny interval dt behaves like a normal random variable with mean 0 and variance dt.

This understanding forms the basis for Itô calculus, where these infinitesimal increments dB_t are used to define stochastic integrals and develop a calculus for non-differentiable processes.

Some example problems

Example 1 - convert continuous SDE to discrete soln without drift term

The continuous stochastic differential equation (SDE) is given by:

$$dx = \epsilon\sqrt{dt} \quad //\text{cont eqn}$$

where $\epsilon \sim \mathcal{N}(0, 1)$.

To convert this continuous SDE to a discrete solution, we consider the increments:

$$X_{t_{i+1}} - X_{t_i} = \sqrt{t_{i+1} - t_i} \cdot \epsilon$$

Assuming t were not discrete, we can express it as:

$$X_{t_{i+1}} = X_{t_i} + \sqrt{t_{i+1} - t_i} \cdot \epsilon$$

Finally, for a discrete solution, we have:

$$x_{i+1} = x_i + \epsilon, \quad \epsilon \sim \mathcal{N}(0, 1) \quad //\text{discrete soln}$$

Example 2 - convert continuous SDE to discrete soln with drift term

Let's look at another example of converting a discrete equation to a continuous form, this time with a drift term:

1. Starting with the continuous SDE:

$$dx = -\frac{\alpha}{2}xdt + \beta dB_t$$

   where α is the drift term and β is the diffusion term.

2. To discretize this equation, we:

- Replace dt with finite time step $\Delta t = t_{i+1} - t_i$
- Replace dB_t with $\epsilon_i \sim \mathcal{N}(0, I)$ scaled by $\sqrt{\Delta t}$

3. This gives us:

$$x_{t_{i+1}} - x_{t_i} = -\frac{\alpha}{2} x_{t_i} (t_{i+1} - t_i) + \beta \sqrt{t_{i+1} - t_i} \epsilon_i$$

4. Rearranging terms:

$$x_{t_{i+1}} = x_{t_i} \left(1 - \frac{\alpha}{2}(t_{i+1} - t_i)\right) + \beta \sqrt{t_{i+1} - t_i} \epsilon_i$$

5. For unit time steps where $t_{i+1} - t_i = 1$, we get:

$$x_{i+1} = x_i \left(1 - \frac{\alpha}{2}\right) + \beta \epsilon_i \quad \text{where} \quad \epsilon_i \sim \mathcal{N}(0, I)$$

This example shows how a continuous SDE with both drift and diffusion terms can be discretized into an iterative update rule.

Anderson's result

Anderson's result states that for a forward SDE of the form:

$$dx = f(x,t)dt + g(t)dB_t$$

The corresponding reverse SDE takes the form:

$$dx = \left(f(x,t) - g^2(t)\nabla_x(\log p_t(x))\right)dt + g(t)dB_t$$

where:

- $f(x,t)$ is the drift term of the forward process
- $g(t)$ is the diffusion coefficient
- $\nabla_x(\log p_t(x))$ is the score function
- The reverse process has the same diffusion term $g(t)dB_t$ as the forward process

This result shows that to reverse a diffusion process, we need to:

1. Keep the same diffusion term
2. Modify the drift term by subtracting the score matching term scaled by $g^2(t)$

DDPMs as SDEs

Step 1 of big picture (Stochastic forward process to continuous SDE)

Now let's see how DDPMs can be viewed as discretizations of SDEs.

The forward process in DDPMs is given by:

$$x_{i+1} = \sqrt{1-\beta_{i+1}}x_i + \sqrt{\beta_{i+1}}\epsilon_i \quad \text{where} \quad \epsilon_i \sim \mathcal{N}(0, I)$$

This discrete process corresponds to a continuous-time SDE of the form(*derivation done in class*):

$$dx = -\frac{\beta(t)}{2}xdt + \sqrt{\beta(t)}dB_t$$

where:

- $\beta(t)$ is a continuous-time version of the discrete noise schedule β_i, with $\beta_i \in (0, 1)$ and $\beta_T \approx 1$ because we want x_T to be pure noise
- $\beta(t)$ must be bounded for the SDE to be well-defined
- The drift term $-\frac{\beta(t)}{2}x$ controls the gradual destruction of the data
- The diffusion term $\sqrt{\beta(t)}dB_t$ adds noise to the process
- This is also called a variance-preserving SDE since it maintains constant variance throughout the diffusion process

Step 2 of big picture (Continuous SDE to continuous reverse SDE)

Using Anderson's result, we can derive the reverse SDE for our DDPM forward process.

Given:

- Forward drift $f(x, t) = -\frac{\beta(t)}{2}x$
- Forward diffusion $g(t) = \sqrt{\beta(t)}$

Substituting into Anderson's reverse SDE formula we get:

$$dx = \left(-\frac{\beta(t)}{2} x - \beta(t) \nabla_x (\log p_t(x)) \right) dt + \sqrt{\beta(t)} dB_t$$

Step 3 of big picture (Continuous reverse SDE to discrete reverse process)

Derivation done in class - the final result is correct but the intermediate steps not checked

1. Starting with the continuous reverse SDE:

$$dx = \left(-\frac{\beta(t)}{2} x - \beta(t) \nabla_x (\log p_t(x)) \right) dt + \sqrt{\beta(t)} dB_t$$

2. For discretization, we make the following substitutions:
   - $dx \approx x_t - x_{t+\Delta t}$ (finite difference)
   - $dt \to \Delta t$
   - $dB_t = \sqrt{\Delta t}\epsilon$ where $\epsilon \sim \mathcal{N}(0, I)$

3. Substituting these into the SDE:

$$x_t - x_{t+\Delta t} = \left(-\frac{\beta(t)}{2} x_t - \beta(t) \nabla_x (\log p_t(x)) \right) \Delta t + \sqrt{\beta(t) \Delta t} \epsilon$$

4. Rearranging to solve for $x_{t+\Delta t}$:

$$x_{t+\Delta t} = x_t - \left(-\frac{\beta(t)}{2} x_t - \beta(t) \nabla_x (\log p_t(x)) \right) \Delta t - \sqrt{\beta(t) \Delta t} \epsilon$$

5. Simplifying:

$$x_{t+\Delta t} = x_t + \frac{\beta(t)}{2} x_t \Delta t + \beta(t) \nabla_x (\log p_t(x)) \Delta t + \sqrt{\beta(t) \Delta t} \epsilon$$

6. Let $\Delta t = 1$ and $t \to i$ for discrete time steps:

$$x_{i+1} = x_i + \frac{\beta_i}{2} x_i + \beta_i \nabla_x (\log p_i(x)) + \sqrt{\beta_i} \epsilon$$

7. Rearranging terms:

$$x_{i+1} = \left(1 + \frac{\beta_i}{2}\right) x_i + \beta_i \nabla_x (\log p_i(x)) + \sqrt{\beta_i}\epsilon$$

8. Let $\epsilon_\theta(x_i, i) = -\nabla_x(\log p_i(x))$ be our neural network prediction:

$$x_{i+1} = \left(1 + \frac{\beta_i}{2}\right) x_i - \beta_i \epsilon_\theta(x_i, i) + \sqrt{\beta_i}\epsilon$$

9. Finally, to match the DDPM reverse process shown in the image:

$$x_{i+1} = \frac{1}{\sqrt{1-\beta_i}} \left(x_i + \beta_i \nabla_x (\log p_i(x))\right) + \sqrt{\beta_i} z_i$$

   where $z_i \sim \mathcal{N}(0, I)$

10. In practice, this is often rewritten using the neural network ϵ_θ which predicts the noise:

$$x_{i-1} = \frac{1}{\sqrt{1-\beta_i}} \left(x_i - \frac{\beta_i}{\sqrt{1-\beta_i}} \epsilon_\theta(x_i, i)\right) + \sigma_i z_i$$

This looks like the DDPM reverse process!

SMLD as SDE

Step 1 of big picture (Stochastic forward process to continuous SDE)

We know SMLD (Score Matching with Langevin Dynamics) forward process is:

$$x_i = x_{i-1} + \sqrt{\sigma_{t+1}^2 - \sigma_t^2} z_{i-1}, \quad z_{i-1} \sim \mathcal{N}(0, I)$$

Let's convert the discrete SMLD process to a continuous SDE:

1. First, let's write the discrete process in terms of continuous time:
    - Let $\Delta t = \frac{1}{N}$ be the time step
    - Map discrete index i to continuous time t: $i = \frac{t}{N}$
    - Write $\sigma_i^2 = \sigma(t)^2$ for continuous noise schedule
2. The discrete process becomes:

    $$x(t + \Delta t) = x(t) + \sqrt{\sigma(t + \Delta t)^2 - \sigma(t)^2} z_t$$

    where $z_t \sim \mathcal{N}(0, I)$
3. Rearranging to get the change in x:

    $$x(t + \Delta t) - x(t) = \sqrt{\sigma(t + \Delta t)^2 - \sigma(t)^2} z_t$$
4. Note that for small Δt:

    $$\sigma(t + \Delta t)^2 - \sigma(t)^2 \approx \frac{d(\sigma(t)^2)}{dt} \Delta t$$
5. Also, $z_t \sqrt{\Delta t} = dB_t$ for Brownian motion increments
6. Substituting these we get:

    $$dx = \sqrt{\frac{d(\sigma(t)^2)}{dt} \Delta t} \frac{dB_t}{\sqrt{\Delta t}}$$
7. Simplifying:

$$dx = \sqrt{\frac{d(\sigma(t)^2)}{dt}} dB_t$$

Step 2 of big picture (Continuous SDE to continuous reverse SDE)

Now we can write the reverse SDE:

1. Using Anderson's result, with:
   - Forward drift $f(x, t) = 0$ (no drift term)
   - Forward diffusion $g(t) = \sqrt{\frac{d(\sigma(t)^2)}{dt}}$
2. The reverse SDE is:

$$dx = \left(-g(t)^2 \nabla_x (\log p_t(x))\right) dt + g(t) dB_t$$

3. Substituting $g(t)$:

$$dx = \left(-\frac{d(\sigma(t)^2)}{dt} \nabla_x (\log p_t(x))\right) dt + \sqrt{\frac{d(\sigma(t)^2)}{dt}} dB_t$$

Step 3 of big picture (Continuous reverse SDE to discrete reverse process)

1. To discretize this SDE:
   - Let $\alpha(t) = \frac{d(\sigma(t)^2)}{dt}$
   - Then $\alpha(t)\Delta t = \Delta(\sigma(t)^2)$
2. The discretized equation becomes:

$$x(t + \Delta t) - x(t) = -\alpha(t) \nabla_x (\log p_t(x)) \Delta t - \sqrt{\alpha(t)\Delta t} z(t)$$

3. Rearranging:

$$x(t + \Delta t) = x(t) + \alpha(t) \nabla_x (\log p_t(x)) \Delta t + \sqrt{\alpha(t)\Delta t} z(t)$$

4. Converting back to discrete indices:

$$x_{i-1} = x_i + (\sigma_i^2 - \sigma_{i-1}^2)\nabla_x \log p_i(x) + \sqrt{\sigma_i^2 - \sigma_{i-1}^2} z_i$$

where $z_i \sim \mathcal{N}(0, I)$

This looks similar to the Langevin dynamics equation we saw earlier, with the gradient term $\nabla_x \log p_i(x)$ guiding the sampling process.
The key difference is that here the noise schedule σ_i^2 controls both the step size and noise magnitude, while in Langevin dynamics these were separate parameters.

Main takeaway

The main takeaway from this exploration is that whenever you see a random process converting one distribution to another, SDEs should come to mind as a powerful mathematical framework. The three sampling methods we studied (Langevin dynamics, DDPM, SMLD) are all discrete approximations of continuous SDEs, differing mainly in their drift and diffusion terms. Understanding them through the SDE lens helps unify these approaches and provides a foundation for developing new sampling techniques.

DDIM (Denoising Diffusion Implicit Models)

Motivation

During inference, we often had to sequentially traverse through 1000 time steps because of the Markov property assumption, which can be computationally intensive.

Is it possible to give up the Markov property assumption so that we can train the same way but sample faster - not have to traverse through 1000 time steps?

Recall the DDPM

Forward process:

$$q(x_t|x_0) = \int q(x_{t:1}|x_0)dx_{1:(t-1)}$$

$$q(x_t|x_0) = \mathcal{N}(x_t; \sqrt{\alpha_t}x_0, (1-\alpha_t)\mathbb{I})$$

$$\Rightarrow x_t = \sqrt{\alpha_t}x_0 + \sqrt{1-\alpha_t}\epsilon, \quad \epsilon \sim \mathcal{N}(.)$$

Note that here α_t is $\bar{\alpha}$ that we studied in DDPM.

Reverse process:

$$p_\theta(x_{0:T}) = p_\theta(x_T)\prod_{t=1}^{T} p_\theta(x_{t-1}|x_t)$$

The training loss function L_T is given by:

$$L_T = \sum_{t=1}^{T} \mathbb{E}_{x_0,\epsilon}\left[||\epsilon - \epsilon_\theta(\sqrt{\alpha_t}x_0 + \sqrt{1-\alpha_t}\epsilon, t)||^2\right]$$

where:

- ϵ_θ is the neural network that predicts the noise
- ϵ is the random noise added during training
- x_0 is the original data point
- α_t controls the noise schedule
- $\sqrt{\alpha_t}x_0 + \sqrt{1-\alpha_t}\epsilon = x_t$ is the forward process

This loss depends on the marginal distribution:

$$q(x_t|x_0) = \mathcal{N}(x_t; \sqrt{\alpha_t}x_0, (1-\alpha_t)\mathbb{I})$$

But importantly, it does not depend on the specific choice of intermediate distributions:

$$q(x_{1:T}|x_0) = \prod_{t=1}^{T} q(x_t|x_{t-1})$$

We can potentially modify the intermediate distributions while keeping the same marginal distribution to enable faster sampling

This observation opens up possibilities for accelerated sampling methods that don't require going through all timesteps sequentially.

Formulation of DDIM

Let's formulate DDIM by deriving an alternative forward process that:

1. Maintains the same marginal distribution $q(x_t|x_0)$ as DDPM
2. Has different intermediate distributions $q(x_{1:T}|x_0)$

The key idea is to find a new forward process such that:

1. The marginal distribution matches DDPM:

$$q_\sigma(x_t|x_0) = q(x_t|x_0) = \mathcal{N}(x_t; \sqrt{\alpha_t}x_0, (1-\alpha_t)\mathbb{I})$$

2. But the intermediate distributions $\prod_{t=1}^{T} q_\sigma(x_t|x_{t-1})$ can be different from DDPM

This is possible because the same marginal distribution can arise from different joint distributions, as long as they integrate to give the same $q(x_t|x_0)$.

Deriving DDIM

We can write the joint distribution $q_\sigma(x_t, x_{t-1}|x_0)$ using Bayes rule as:

$$q_\sigma(x_t|x_{t-1}, x_0) = \frac{q_\sigma(x_{t-1}|x_t, x_0)q_\sigma(x_t|x_0)}{q_\sigma(x_{t-1}|x_0)}$$

We assume $q_\sigma(x_{t-1}|x_t, x_0)$ follows a Gaussian distribution:

$$q_\sigma(x_{t-1}|x_t, x_0) = \mathcal{N}(x_{t-1}; \tilde{\mu}_t(x_t, x_0), \sigma^2\mathbb{I})$$

where $\tilde{\mu}_t = \sqrt{\alpha_{t-1}}x_0 + \sqrt{1 - \alpha_{t-1}^2 - \sigma^2}\left(\frac{x_t - \sqrt{\alpha_t}x_0}{\sqrt{1-\alpha_t}}\right)$

derivation not done in class

This formulation:

1. Maintains the same marginal distribution as DDPM
2. Introduces a new parameter σ that controls the stochasticity
3. When $\sigma^2 = 1 - \alpha_{t-1}/\alpha_t$, recovers the original DDPM
4. When $\sigma = 0$, gives a deterministic process (DDIM)

How do we do inference with DDIM?

For inference with DDIM, we follow these steps:

1. First, we note that x_0 can be expressed in terms of x_t and ϵ as:

$$x_0 = \frac{x_t - \sqrt{1 - \alpha_t}\epsilon}{\sqrt{\alpha_t}}$$

2. We can predict x_0 using a neural network $f_\theta^{(t)}(x_t) = \frac{x_t - \sqrt{1-\alpha_t}\epsilon_\theta(x_t, t)}{\sqrt{\alpha_t}}$

3. Then, we can sample x_{t-1} using:

$$p_\theta(x_{t-1}|x_t) = \begin{cases} \mathcal{N}(f_\theta^{(t)}(x_t), \sigma_t^2\mathbb{I}) & \text{if } t = 1 \\ q_\sigma(x_{t-1}|x_t, f_\theta^{(t)}(x_t)) & \text{otherwise} \end{cases}$$

The iterations drop from 1000 to 20-30 steps, and the samples are of comparable quality to DDPM.

DDIM inversion

X_T to X_0

For $\sigma = 0$, the DDIM process becomes deterministic:

$$x_{t-1} = \sqrt{\alpha_{t-1}} \left(\frac{x_t - \sqrt{1-\alpha_t}\epsilon_\theta^{(t)}(x_t)}{\sqrt{\alpha_t}} \right) + \sqrt{1-\alpha_{t-1}}\epsilon_\theta^{(t)}(x_t)$$

where:

- $\epsilon_\theta^{(t)}(x_t)$ is the predicted noise from our trained model
- α_t is the cumulative product of $(1 - \beta_t)$

This deterministic nature means that:

1. For a given x_T, we can find a unique x_0 by following the forward process
2. Given this x_T, we can recover the exact same x_0 through the reverse process

This property makes DDIM particularly useful for:

- Finding latent representations of real images
- Performing image editing in the latent space
- Interpolating between images in a semantically meaningful way

The key insight is that unlike DDPM which has stochastic transitions, DDIM with $\sigma = 0$ gives us a one-to-one mapping between x_0 and x_t, making the inversion process well-defined and deterministic.

X_0 to X_T

For the forward process from x_0 to x_T, we can write:

$$\frac{x_t}{\sqrt{\alpha_t}} = \frac{x_{t-1}}{\sqrt{\alpha_{t-1}}} + \epsilon_\theta^{(t)}(x_t) \left(\sqrt{\frac{1-\alpha_t}{\alpha_t}} - \sqrt{\frac{1-\alpha_{t-1}}{\alpha_{t-1}}} \right)$$

To convert this to continuous form, we can write:

$$\frac{dx_t}{\sqrt{\alpha_t}} = \frac{x_{t-1}dt}{\sqrt{\alpha_{t-1}}} + \epsilon_\theta^{(t)}(x_t)\left(\sqrt{\frac{1-\alpha_t}{\alpha_t}} - \sqrt{\frac{1-\alpha_{t-1}}{\alpha_{t-1}}}\right)dt$$

Let's define:

- $\bar{x}(t) = \frac{x_t}{\sqrt{\alpha_t}}$
- $\sigma(t) = \sqrt{\frac{1-\alpha_t}{\alpha_t}}$

Then we get (derivation not done in class):

$$d\bar{x}(t) = \epsilon_\theta^{(t)}\left(\frac{\bar{x}(t)}{\sqrt{1+\sigma^2(t)}}\right)d\sigma(t)$$

This continuous equation can be discretized back as:

$$\bar{x}(t) = \bar{x}(t-\Delta t) + \Delta\bar{x}(t-\Delta t)$$

Which gives us our original forward process equation:

$$\frac{x_t}{\sqrt{\alpha_t}} = \frac{x_{t-1}}{\sqrt{\alpha_{t-1}}} + \epsilon_\theta^{(t)}(x_{t-1})\left(\sqrt{\frac{1-\alpha_t}{\alpha_t}} - \sqrt{\frac{1-\alpha_{t-1}}{\alpha_{t-1}}}\right)$$

For DDIM inversion, we can rearrange this to solve for the previous timestep:

$$\frac{x_{t-1}}{\sqrt{\alpha_{t-1}}} = \frac{x_t}{\sqrt{\alpha_t}} - \epsilon_\theta^{(t)}(x_t)\left(\sqrt{\frac{1-\alpha_t}{\alpha_t}} - \sqrt{\frac{1-\alpha_{t-1}}{\alpha_{t-1}}}\right)$$

This is a deterministic ODE that we can solve using numerical methods to find a unique x_T for a given x_0.

Sequence to sequence models (Transformers)

Let the data be

$$D = \{(x_i, y_i)\}_{i=1}^{N} \text{ iid } \sim p(x, y).$$

- here $x_i = \{x_i^1, x_i^2, \ldots, x_i^k\}$, where $x_j^i \in \mathbb{R}^d$ represents a sequence of k vectors of dimension d.They are called tokens in usual NLP models.
- here $y_i = \{y_i^1, y_i^2, \ldots, y_i^m\}$, where $y_j^i \in \mathbb{R}^{d'}$ represents a sequence of m vectors of dimension d'. Note that $d' \neq d$. It represents a softmax distribution over a vocabulary of size d' in NLP models.

The models that map x_i to y_i are called seq2seq models.

We will stufy transformers as regularizers just as we studied CNNs as regularizers.

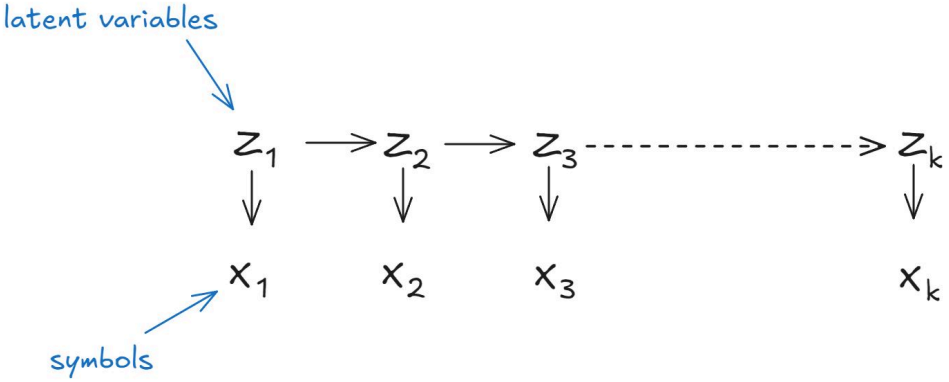
Historical models - Hidden Markov Models(HMMs)

In 1980s, Hidden Markov Models were used to model sequences before RNNs were popular.

We assume a sequence of latent variables $Z_1 \to Z_2 \to \ldots \to Z_k$.

This follows a Markovian assumption on the latent variables, meaning that the future state depends only on the current state.

At every transition, the latent variable emits a symbol x_i.



We model the joint distribution of the sequence as:

$$p(x) = p(x|z)p(z)$$

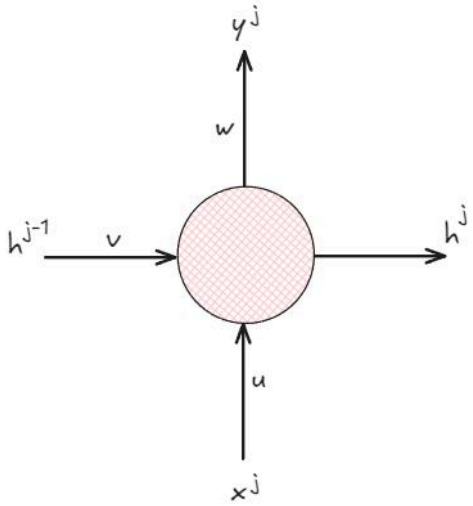
We model $p(x|z)$ as a Gaussian mixture model and use EM algorithm to estimate the parameters.
We model $p(z)$ as a Markov chain and try to estimate the transition probabilities.

The intution behind this model is:

1. The model was usually used for speech modeling.
2. Humans also think something in their brain which is not observable just like the latent variables and emit sounds which are observable just like the symbols.

Recurrent Neural Networks

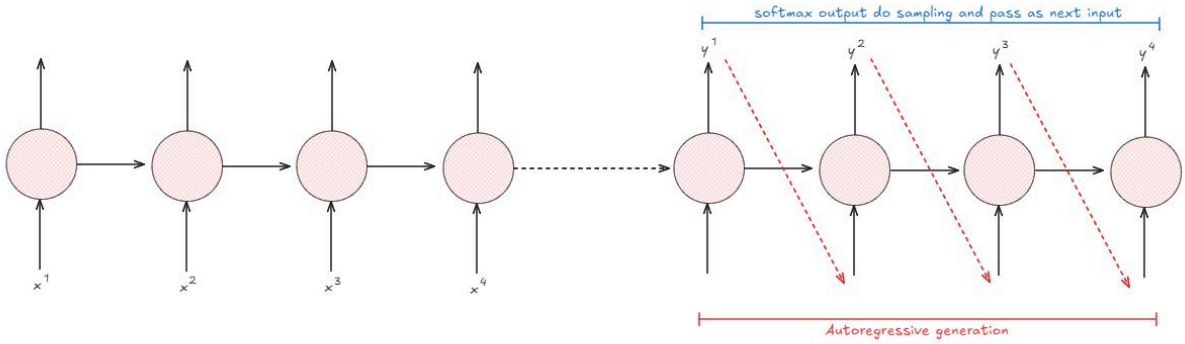
The problem with HMMs was that the length of the input sequence was always fixed.



here:

- $h^j = \sigma(vh^{j-1} + \alpha x^j + b)$ is the hidden state at time j
- $y^j = \sigma(wh^j + b')$ is the output at time j

RNN were able to handle variable length inputs by sharing parameters across time and were able to be used as language models.



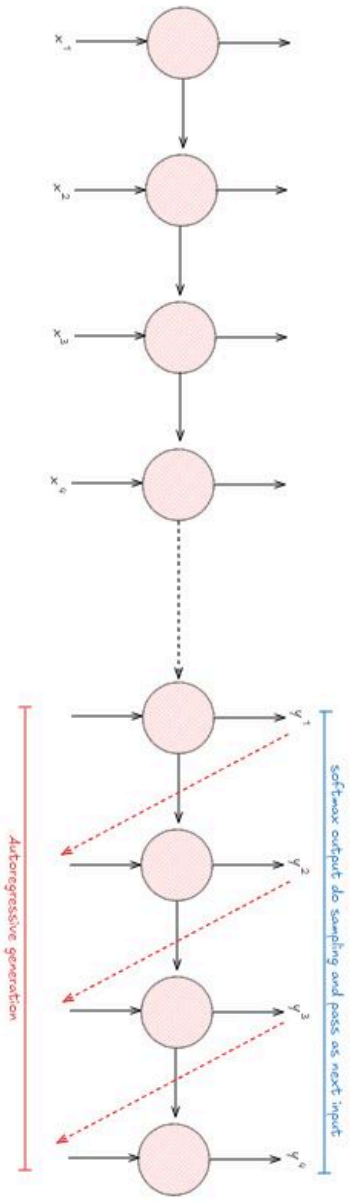
We train such models using cross entropy loss by doing ERM with gradient descent. The main problem with RNNs was that we had to pass each token one by one and could not parallelize the computation.

At the same time Unet architectures with skip connections were becoming popular in CV. We can use the same idea in NLP.

Transformers

Attention head

Transformers can be seen as RNNs with skip connections flipped 90 degrees.



Lets consider we have just 1 data point consisting of k tokens $x = \{x^1, x^2, \ldots, x^k\}$
We define 3 learnable matrices W_q, W_k, W_v defined as:

- $q^j = W_q x^j$ is called the query vector
- $k^j = W_k x^j$ is called the key vector
- $v^j = W_v x^j$ is called the value vector

There is a latent/hidden vector z^j for each token x^j calculated as:

$$z^j = \sum_{t=1}^{m} \alpha_j^t v^t$$

where α_j^t is a measure of how much attention the token x^j pays to the token x^t and is calculated by just taking the dot product of the query vector:

$$\alpha_j^t = (q^j)^T k^t$$

However, we scale down the dot product by $\sqrt{d}$ to prevent it from exploding where d is the dimension of the key and query vectors.

Also, we take a softmax of the attention scores to ensure that they sum to 1. hence usually you will find the following in the literature:

$$\alpha_j^t = \text{softmax}\left(\frac{(q^j)^T k^t}{\sqrt{d}}\right)$$

A good way to think of attention is to think of it as projecting every token to a new space that is a function of all the tokens.

We can concatenate all the tokens into matrices $Q = [q^1, q^2, \ldots, q^k]$, $K = [k^1, k^2, \ldots, k^k]$, and $V = [v^1, v^2, \ldots, v^k]$ to compute attention for all tokens in parallel using matrix multiplication:

$$Z = V \cdot \text{softmax}\left(\frac{Q^T K}{\sqrt{d}}\right)$$

Multi-head attention

Just like we have multiple kernels in CNNs, we can have multiple attention heads.
To get multihead attention, say h heads, we concatenate the outputs $Z_1, Z_2, \ldots, Z_h$ from each attention head and project it using a learnable matrix W_o to get the final output:

$$Z = W_o[Z_1; Z_2; \ldots; Z_h]$$

State space models

Motivation for state space models

- RNNs can handle variable length inputs however they are not parallelizable.
- CNNs are parallelizable however they cannot handle variable length inputs.

Can we get the best of both worlds?

Linear State Space Models

A linear state space model consists of two equations:

$$h'(t) = Ah(t) + Bx(t)$$

$$y(t) = Ch(t) + Dx(t)$$

where:

- $h(t)$ is the hidden state at time t
- $x(t)$ is the input at time t
- $y(t)$ is the output at time t
- $h'(t)$ is the derivative of the hidden state with respect to time
- A, B, C, D are learnable parameter matrices that define the dynamics

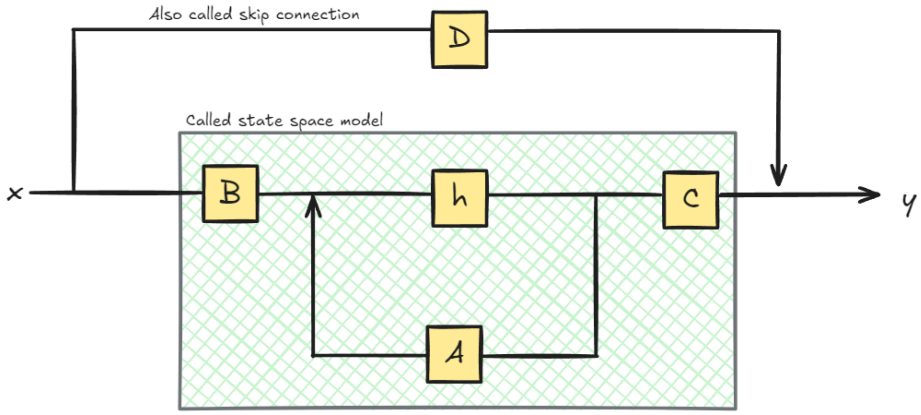
The first equation describes how the hidden state evolves over time based on:

1. The current hidden state $Ah(t)$ term
2. The current input $Bx(t)$ term

The second equation describes how the output is generated from:

1. The current hidden state $Ch(t)$ term

2. The current input $Dx(t)$ term



Euler's method

One common way to solve these equations numerically is using Euler's method. Given a small time step Δt, we can approximate the derivative as:

$$\frac{dh(t)}{dt} \approx \frac{h(t+\Delta t) - h(t)}{\Delta t}$$

This approximation comes from the definition of a derivative as the limit of a difference quotient. Rearranging this gives us the forward Euler update rule:

$$h(t+\Delta t) \approx h(t) + \Delta t \cdot h'(t)$$

Substituting in the state equation $h'(t) = Ah(t) + Bx(t)$:

$$h(t+\Delta t) \approx h(t) + \Delta t \cdot (Ah(t) + Bx(t))$$

$$= h(t) + \Delta t \cdot Ah(t) + \Delta t \cdot Bx(t)$$

$$= (I + \Delta t A)h(t) + \Delta t B x(t)$$

Where I is the identity matrix. This gives us a discrete-time update rule that we can implement efficiently. The accuracy of this approximation depends on the time step Δt - smaller steps yield better accuracy but require more computational steps. This tradeoff between accuracy and computational efficiency is a key consideration when implementing these models.

Now, we can rewrite this using discrete time notation where t represents time steps:
Let $h(t + \Delta t) = h_t$ and $h(t) = h_{t-1}$
Let $\bar{A} = (I + \Delta t A)$ and $\bar{B} = \Delta t B$

This gives us:

$$h_t = \bar{A} h_{t-1} + \bar{B} x_t$$

The second equation doesn't involve derivatives, so it simply becomes:

$$y_t = C h_t + D x_t$$

where:

- $t \in \{t_1, t_2, ..., t_k\}$ represents discrete time steps
- $\bar{A} = (I + \Delta t A)$ is the discretized state transition matrix
- $\bar{B} = \Delta t B$ is the discretized input matrix
- h_t is the hidden state vector at discrete time step t
- x_t is the input vector at time step t
- y_t is the output vector at time step t

Unrolling the equations

Let's assume initial state $h_{-1} = 0$. Then:

$$h_0 = \bar{B} x_0$$

$$y_0 = C h_0 = C \bar{B} x_0$$

Now lets assume $D = 0$. Then:

$$h_1 = \bar{A} h_0 + \bar{B} x_1 = \bar{A} \bar{B} x_0 + \bar{B} x_1$$

$$y_1 = C h_1 = C(\bar{A} \bar{B} x_0 + \bar{B} x_1)$$

Similarly:

$$h_2 = \bar{A} h_1 + \bar{B} x_2 = \bar{A}^2 \bar{B} x_0 + \bar{A} \bar{B} x_1 + \bar{B} x_2$$

$$y_2 = Ch_2 = C(\bar{A}^2\bar{B}x_0 + \bar{A}\bar{B}x_1 + \bar{B}x_2)$$

And in general, for any time step k:

$$h_k = \bar{A}^k\bar{B}x_0 + \bar{A}^{k-1}\bar{B}x_1 + ... + \bar{A}\bar{B}x_{k-1} + \bar{B}x_k$$

$$y_k = Ch_k = \sum_{i=0}^{k} C\bar{A}^{k-i}\bar{B}x_i$$

where we define $\bar{A}^0 = I$ (identity matrix).

The matrix part of y_k is called the convolution kernel of the state space model:

$$K = [C\bar{B}, C\bar{A}\bar{B}, C\bar{A}^2\bar{B}, ...]$$

A major benefit of representing the SSM as a convolution is that it can be trained in parallel like Convolutional Neural Networks (CNNs). However, due to the fixed kernel size, their inference is not as fast and unbounded as RNNs.

Initialization of A matrix

Matrix A is initialized using HiPPO for High-order Polynomial Projection Operators. The HiPPO matrix A is defined as follows:

For an N x N matrix A, the entries $A[n, k]$ are:

- For entries below the diagonal ($n > k$):

$$A[n, k] = ((2n + 1)^{1/2} \cdot (2k + 1)^{1/2})$$

- For entries on the diagonal ($n = k$):

$$A[n, k] = n + 1$$

- For entries above the diagonal ($n < k$):

$$A[n, k] = 0$$

Building matrix A using HiPPO was shown to be much better than initializing it as a random matrix.
The idea behind the HiPPO Matrix is that it produces a hidden state that memorizes its history.
Mathematically, it does so by tracking the coefficients of a Legendre polynomial which allows it to approximate all of the previous history.

A problem

If $\bar{A}$ is not diagonalizable, then we cannot compute $\bar{A}^k$ efficiently. Hence it is written as:

$$\bar{A} = \Lambda + P\beta^H$$

where:

- Λ is a diagonal matrix
- P is a low rank matrix
- β^H is the conjugate transpose of some vector β

To compute powers efficiently, we can:

1. Transform the matrix to the inverse Fourier domain
2. Compute the powers in the inverse Fourier domain
3. Transform the result back to the time domain

This allows computation in linear time rather than having to explicitly calculate matrix powers.

Linear Time Invariance

Recall that:

$$y_k = Ch_k = \sum_{i=0}^{k} C\bar{A}^{k-i}\bar{B}x_i$$

These representations share an important property, namely that of Linear Time Invariance (LTI). LTI states that the SSMs parameters, A, B, and C, are fixed for all

timesteps. This means that matrices A, B, and C are the same for every token the SSM generates.

In other words, regardless of what sequence you give the SSM, the values of A, B, and C remain the same. We have a static representation that is not content-aware.

Mamba

Mamba is a state space model that addresses the limitations of Linear Time Invariance (LTI) through several key innovations:

1. Recurrent State Space Model:
   The continuous SSM is discretized into a recurrent form:

$$h_{k+1} = \bar{A}h_k + \bar{B}x_k$$

$$y_k = \bar{C}h_k$$

   where:

- $\bar{A} = (I + \Delta_k A)$
- $\bar{B} = \Delta_k B$
- $\bar{C} = C$
- Δ_k is a learned step size that varies with input
- h_k is the hidden state at step k

2. HiPPO Matrix Initialization:
   Matrix A is initialized using HiPPO to effectively capture long-range dependencies through polynomial approximations. This provides a strong foundation for modeling historical information in the sequence.

3. Selective Scan:
   A selective mechanism D_k is introduced that learns to selectively retain or discard information:

$$h_{k+1} = D_k \odot (\bar{A}h_k + \bar{B}x_k)$$

where D_k is computed from the input sequence and $\odot$ represents element-wise multiplication.

4. Hardware-Efficient Implementation:
   The model uses a hardware-aware algorithm that:

- Reorders operations to maximize parallel computation
- Fuses multiple operations into single kernels
- Optimizes memory access patterns
- Achieves state-of-the-art inference speed on modern hardware

The matrices B, C, step size Δ_k, and selective mechanism D_k are all learned from the input sequence, making the model content-aware while maintaining computational efficiency. This Mamba block can be used in place of a self-attention layer in a Transformer model.

Noise Contrastive Estimation

Idea

The core idea behind Noise Contrastive Estimation (NCE) is:

1. Given samples from a distribution, if you know how to distinguish them from noise samples, then you implicitly know the underlying distribution.

2. If you know how to tell what is noise from what is real data, you must have learned something about the true data distribution.

This provides an alternative way to learn probability distributions by transforming the density estimation problem into a binary classification problem between real and noise samples.

Formulation

- Let the dataset be denoted as

$$D = \{x_1, x_2, \ldots, x_t\}$$

  where each element x_i represents a sample from the true data distribution.

- Let a noise sample be denoted as

$$D_N = \{y_1, y_2, \ldots, y_t\}$$

- Our objective is to estimate the true data distribution $p_D(x)$ given the dataset D and the noise samples D_N which we represent as $p_\theta(x)$ the model's estimate of the data distribution.

- We cast it as a binary classification problem between real and noise samples.

$$U = \{(u_1, c_1), (u_2, c_2), \ldots, (u_{2t}, c_{2t})\}$$

  where

  - $u_i \in \{x_1, x_2, \ldots, x_t, y_1, y_2, \ldots, y_t\}$ and

- $c_i \in \{0, 1\}$ is a binary label indicating whether u_i is a real data sample ($c_i = 1$) or a noise sample ($c_i = 0$).

- Now the posterior is given by:
  - $p(u_i|c_i = 1) = p_D(u_i)$ ie the data distribution
  - $p(u_i|c_i = 0) = p_N(u_i)$ ie the noise distribution that is a known distribution

- Assume that the priors are equal, ie $p(c_i = 1) = p(c_i = 0) = 0.5$ ie the data and noise samples are equally likely because we there are as many data samples as noise samples. Then:
  - Using Bayes' rule, we can write:

$$p(c = 1|u) = \frac{p(u|c = 1)p(c = 1)}{p(u|c = 0)p(c = 0) + p(u|c = 1)p(c = 1)}$$

  - Substituting the distributions and priors:

$$p(c = 1|u) = \frac{p_D(u)}{p_N(u) + p_D(u)}$$

  - This gives us a way to estimate the probability that a sample u comes from the real data distribution versus the noise distribution.

- We define
  - $G(u; \theta) = \log p_\theta(u) - \log p_N(u)$ which is just the difference between the estimated log probabilities of the data distribution and the noise distribution.
  - $p(c = 1|u) = h_\theta(u) = \sigma(G(u; \theta)) = \frac{1}{1+\exp(-G(u;\theta))}$ where $\sigma(s) = \frac{1}{1+\exp(-s)}$ is the sigmoid function representing the probability that a sample u comes from the real data distribution versus the noise distribution.

- The likelihood function for the binary classification problem is:

$$L(\theta) = \sum_{t=1}^{2T} c_t \log p(c_t = 1|u_t; \theta) + (1 - c_t) \log p(c_t = 0|u_t; \theta)$$

  Which can be rewritten as:

$$L(\theta) = \sum_{t=1}^{T} \log h_\theta(x_t) + \sum_{t=T}^{2T} \log(1 - h_\theta(y_t))$$

  where:

- $h_\theta(u) = p(c = 1|u)$ is the classifier's prediction
- The first sum is over real data samples
- The second sum is over noise samples
- θ are the parameters we want to learn
- $L(\theta)$ is also known as NCE estimator

- From law of large numbers, we can write the expected value of the likelihood:

$$J(\theta) = \frac{1}{2}\mathbb{E}_{p_{DN}}[\log h_\theta(x) + \log(1 - h_\theta(y))]$$

where:

- The expectation is taken over the joint distribution of data and noise samples
- x follows the data distribution p_D
- y follows the noise distribution p_N

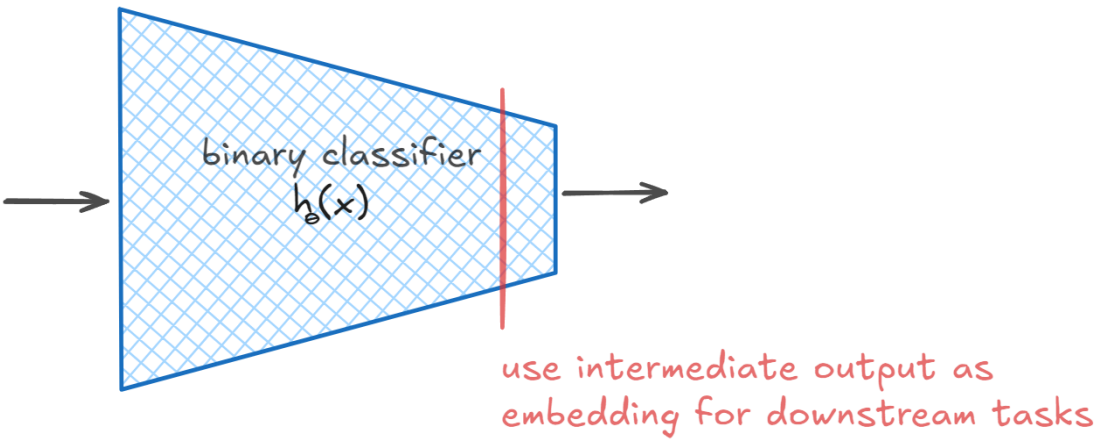
- After some algebraic manipulation, we can write:

$$J(\theta) = \frac{1}{2}\mathbb{E}[\log r(log(p_\theta(x)) - log(p_N(x))) + \log(1 - r(log(p_\theta(y)) - log(p_N(y))))]$$

where:

- $r(\cdot)$ is the sigmoid function
- The first two terms correspond to real data samples
- The last two terms correspond to noise samples

- Theorem(without proof): $J(\theta)$ attains when $p_\theta(x) = p_D(x)$ if $p_N(x)$ is chosen such that $p_N(x)$ is nonzero for all x in the support of $p_D(x)$.



Info-NCE

Idea

Instead of using 1 negative sample, why not use more?

Formulation

Let us formalize the Info-NCE framework:

1. Given:
    - A data point $x \in \mathcal{X}$ from input space $\mathcal{X}$
    - A context $c \in \mathcal{C}$ from context space $\mathcal{C}$
    - A positive sampling distribution $p_{pos}(x, c)$
    - A negative sampling distribution $p_{neg}(x, c)$
2. We sample:
    - One positive sample $x^+ \sim p_{pos}(x, c)$
    - $N - 1$ negative samples $\{x_i^-\}_{i=1}^{N-1} \sim p_{neg}(x, c)$
3. The InfoNCE loss is defined as:

$$\mathcal{L}_{\text{InfoNCE}} = -\mathbb{E}\left[\log \frac{\exp(f_\theta(x)^\top f_\theta(x^+))}{\exp(f_\theta(x)^\top f_\theta(x^+)) + \sum_{i=1}^{N-1} \exp(f_\theta(x)^\top f_\theta(x_i^-))}\right]$$

   where:
    - $f_\theta : \mathcal{X} \to \mathbb{R}^d$ is a neural network encoder that maps inputs to d-dimensional embeddings
    - θ represents the learnable parameters of the encoder
    - x^+ denotes the positive sample
    - x_i^- denotes the i-th negative sample
    - The expectation is taken over the sampling distributions

4. This formulation can be interpreted as a softmax-based classifier that tries to identify the positive sample among N-1 negative samples.

5. We can use these embeddings to train other models downstream.

Masked reconstruction

Idea

Contrastive learning is one way to learn representations from unlabeled data. Masked reconstruction is another way.

Formulation

Let us formalize the masked reconstruction framework:

1. Given:
    - Input space $\mathcal{X}$
    - A data point $x \in \mathcal{X}$
    - A masking operator $\hat{x}$ that corrupts the input
    - An encoder-decoder model $T_\theta(x)$ parameterized by θ
2. The process works as follows:
    - Apply masking operator: $\hat{x} = \text{mask}(x)$
    - Pass masked input through model: $\tilde{x} = T_\theta(\hat{x})$
    - Reconstruct original input: $\hat{x} \to \tilde{x} \approx x$
3. The objective is to minimize reconstruction loss:

$$\mathcal{L}(\theta) = \mathbb{E}_{x \sim p_{\text{data}}} \left[d(x, T_\theta(\text{mask}(x))) \right]$$

    where:
    - $d(\cdot, \cdot)$ is a distance metric (e.g. MSE, cross-entropy)
    - The expectation is taken over the data distribution
    - θ represents the learnable parameters
4. Common masking strategies include:
    - Random token masking (like in BERT)
    - Span masking (consecutive tokens)

- Structured masking (task-specific patterns)

5. The learned representations can be used for:

    - Pre-training for downstream tasks
    - Feature extraction
    - Transfer learning

This approach forces the model to learn meaningful representations by reconstructing missing or corrupted parts of the input using available context.

JEPA - Joint Embedding Prediction and Autoencoding

Idea

Joint Embedding Predictive Architecture (JEPA) combines predictive learning with representation learning in a unified framework.

Formulation

Let us formalize the JEPA framework:

1. Given:
    - Input space $\mathcal{X}$
    - Context patch $x_c \in \mathcal{X}$
    - Target patch $x_t \in \mathcal{X}$
    - Context encoder $f_\theta : \mathcal{X} \to \mathcal{Z}$
    - Target encoder $g_\phi : \mathcal{X} \to \mathcal{Z}$
    - Predictor network $h_\psi : \mathcal{Z} \to \mathcal{Z}$
2. The process works as follows:
    - Encode context: $z_c = f_\theta(x_c)$
    - Encode target: $z_t = g_\phi(x_t)$
    - Predict target embedding: $\hat{z}_t = h_\psi(z_c)$
3. The objective is to minimize prediction error in embedding space:

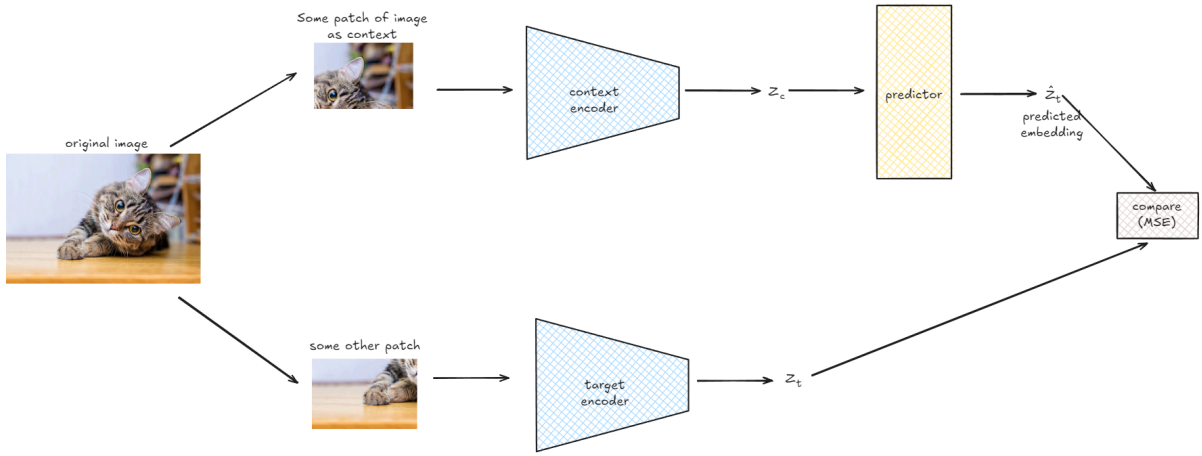
$$\mathcal{L}(\theta, \phi, \psi) = \mathbb{E}_{(x_c, x_t) \sim p_{\text{data}}} \left[\| z_t - \hat{z}_t \|^2 \right]$$

    where:
    - The expectation is over pairs of context and target patches
    - $\| \cdot \|$ denotes Euclidean distance
    - Parameters θ, ϕ, ψ are learned jointly

4. Key properties:

    - Learns representations by predicting embeddings rather than raw inputs
    - Uses vision transformer architecture as underlying backbone
    - Can leverage any encoder architecture for inference
    - Enables both predictive and autoencoding objectives

This approach combines benefits of predictive learning and representation learning while avoiding pixel-level reconstruction.



Training and Downstream Usage

1. During training, three components are trained jointly:

    - Context encoder f_θ that maps context patches to embeddings
    - Target encoder g_ϕ that maps target patches to embeddings
    - Predictor network h_ψ that predicts target embeddings from context embeddings

2. The training process optimizes:

    - Context encoder parameters θ
    - Target encoder parameters ϕ
    - Predictor network parameters ψ

    Through minimizing the prediction error in embedding space.

3. For downstream tasks:

    - The target encoder g_ϕ is typically used as the main feature extractor

- This is because the target encoder learns to create embeddings that:
  - Capture meaningful semantic information about inputs
  - Are predictable from context, implying they encode important features
  - Don't rely on predicting low-level details like pixels
- The context encoder can also be used but generally performs slightly worse since it's optimized for prediction rather than representation

4. Common downstream applications include:

- Image classification: Use target encoder embeddings as input features
- Object detection: Use target encoder as backbone network
- Semantic segmentation: Use target encoder for dense feature extraction
- Other vision tasks that benefit from pre-trained representations

Distillation

- Teacher model is a large model that was trained on a large dataset as usual.
- Student model is a smaller model that is trained to mimic the behavior of the teacher model.
- The student model has 2 losses:

$$\mathcal{L} = \mathcal{L}_{\text{task}} + \mathcal{L}_{\text{distill}}$$

  - $\mathcal{L}_{\text{task}}$ is the task-specific loss.
  - $\mathcal{L}_{\text{distill}}$ is the distillation loss - This loss is used to make the student model's output close to the teacher model's softened outputs.