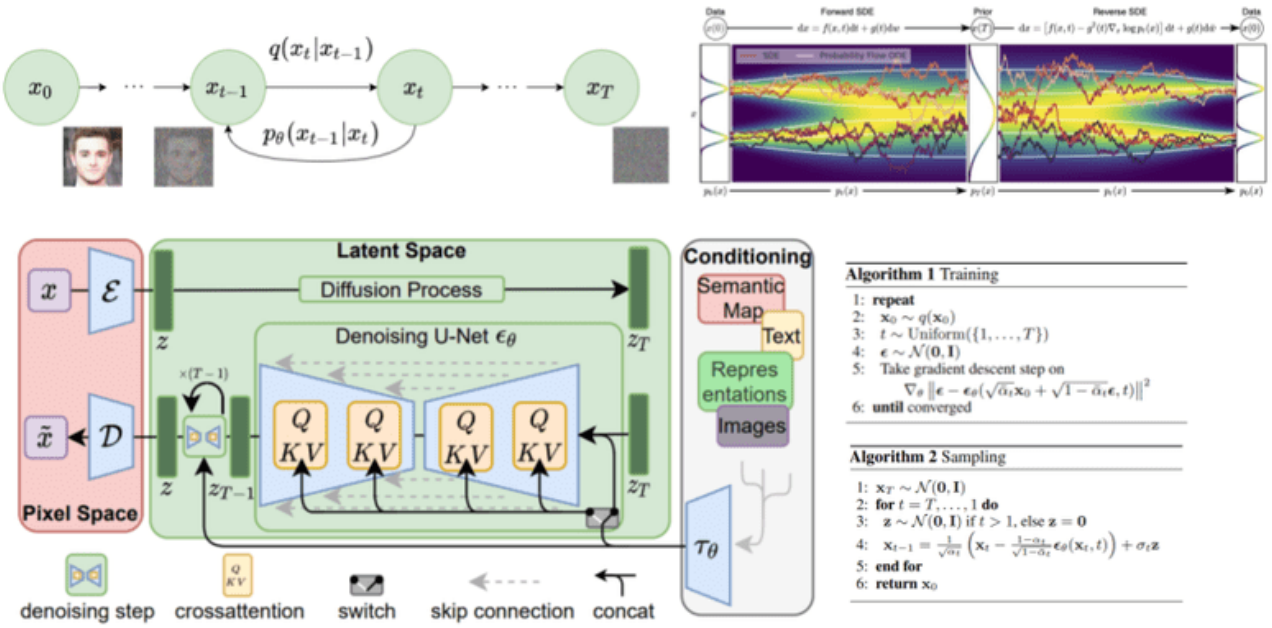
Diffusion Models



Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$
6: **until** converged

Algorithm 2 Sampling

1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = T, \dots, 1$ **do**
3: $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4: $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$

First, we need to define What is a generative model?

A generative model learns a probability distribution of the data set such that we can then sample from the distribution to create new instances of data. For example, if we have many pictures of cats and we train a generative model on it, we then sample from this distribution to create new images of cats.

Now what are Diffusion Models are generative models which have been gaining significant popularity in the past several years, and for good reason. A handful of seminal papers released in the 2020s *alone* have shown the world what Diffusion models are capable of, such as beating [GANs] on image synthesis. Most recently, practitioners will have seen Diffusion Models used in DALL-E 2, OpenAI's image generation model released last month.

Diffusion Models—Introduction

Diffusion Models are **generative** models, meaning that they are used to generate data similar to the data on which they are trained. Fundamentally, Diffusion Models work by **destroying training data** through the successive addition of Gaussian noise, and then **learning to recover** the data by *reversing* this noising process. After training, we can use the Diffusion Model to generate data by simply **passing randomly sampled noise through the learned denoising process.**

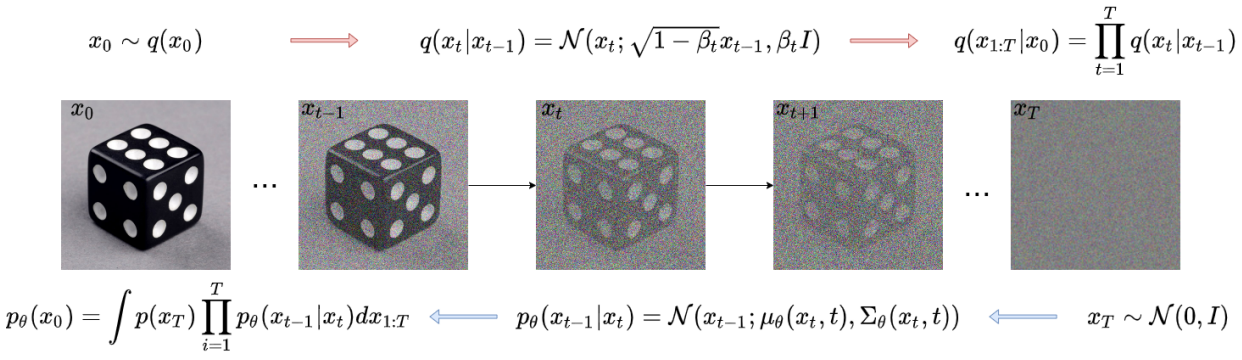
Diffusion models are inspired by **non-equilibrium thermodynamics**. They define a **Markov chain** of diffusion steps to slowly add random noise to data and then learn to reverse the diffusion process to construct desired data samples from the noise. Unlike **VAE or flow models**, diffusion models are learned with a fixed

procedure and the latent variable has high dimensionality (same as the original data).

Now get deeper into the diffusion models:

diffusion models consists of two processes as shown in the image below:

- Forward process (with red lines).
- Reverse process (with blue lines).



> As mentioned above, a Diffusion Model consists of a **forward process** (or **diffusion process**), in which a datum (generally an image) is progressively noised, and a **reverse process** (or **reverse diffusion process**), in which noise is transformed back into a sample from the target distribution.

In a bit more detail for images, the set-up consists of 2 processes:

- a fixed (or predefined) forward diffusion process q of our choosing, that gradually adds Gaussian noise to an image, until you end up with pure noise
- a learned reverse denoising diffusion process p_θ, where a neural network is trained to gradually denoise an image starting from pure noise, until you end up with an actual image.

1. Forward Process (Fixed):

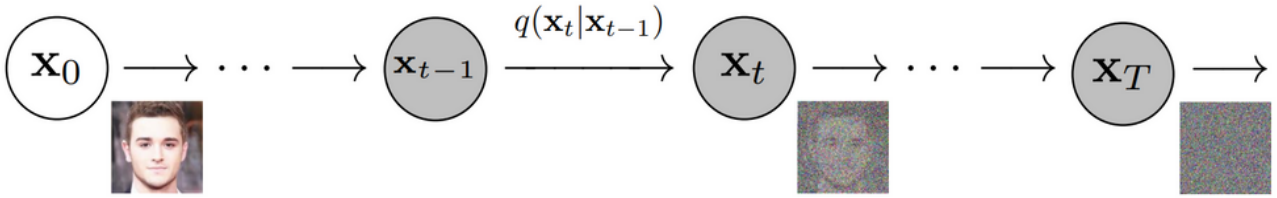
The sampling chain transitions in the forward process can be set to conditional Gaussians when the noise level is sufficiently low. Combining this fact with the Markov assumption leads to a simple parameterization of the forward process:

Given a data point sampled from a real data distribution $\mathbf{x}_0 \sim q(\mathbf{x})$, let us define a *forward diffusion process* in which we add small amount of Gaussian noise to

the sample in T steps, producing a sequence of noisy samples $\mathbf{x}_1, \ldots, \mathbf{x}_T$. The step sizes are controlled by a variance schedule $\{\beta_t \in (0,1)\}_{t=1}^T$.

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \quad q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1})$$

Where $\beta_1, \ldots, \beta_T$ is a variance schedule (either learned or fixed) which, if well-behaved, **ensures that x_T is nearly an isotropic Gaussian for sufficiently large T**.
The data sample $\mathbf{x}_0$ gradually loses its distinguishable features as the step t becomes larger. Eventually when $T \to \infty$, $\mathbf{x}_T$ is equivalent to an isotropic Gaussian distribution.

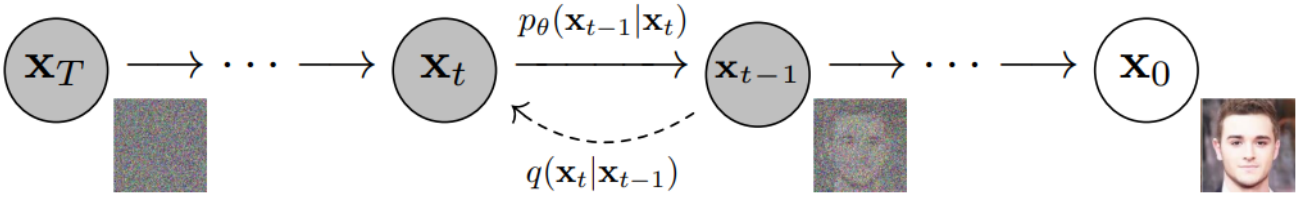


2. Reverse Process (Learned)

As mentioned previously, the "magic" of diffusion models comes in the **reverse process** During training, the model learns to reverse this diffusion process in order to generate new data. Starting with the pure Gaussian noise $p(\mathbf{x}_T) := \mathcal{N}(\mathbf{x}_T, \mathbf{0}, \mathbf{I})$, the model learns the joint distribution $p_\theta(\mathbf{x}_{0:T})$ as

Ultimately, the image is asymptotically transformed to pure Gaussian noise. The **goal** of training a diffusion model is to learn the **reverse** process - i.e. training $p_\theta(x_{t-1}|x_t)$. By traversing backwards along this chain, we can generate new data.

$$p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T) \prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) \quad p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t))$$



where the time-dependent parameters of the Gaussian transitions are learned. Note in particular that the Markov formulation asserts that a given reverse

diffusion transition distribution depends only on the previous timestep (or following timestep, depending on how you look at it).

> ✎ **Both the forward and reverse process indexed by t happen for some number of finite time steps T (the DDPM authors use $T = 1000$). You start with $t = 0$ where you sample a real image x_0 from your data distribution (let's say an image of a cat from ImageNet), and the forward process samples some noise from a Gaussian distribution at each time step t, which is added to the image of the previous time step. Given a sufficiently large T and a well behaved schedule for adding noise at each time step, you end up with what is called an isotropic Gaussian distribution at $t = T$ via a gradual process.**

Denoising Diffusion Probabilistic Model

Paper background:

2 Background

Diffusion models [53] are latent variable models of the form $p_\theta(\mathbf{x}_0) := \int p_\theta(\mathbf{x}_{0:T})\, d\mathbf{x}_{1:T}$, where $\mathbf{x}_1, \ldots, \mathbf{x}_T$ are latents of the same dimensionality as the data $\mathbf{x}_0 \sim q(\mathbf{x}_0)$. The joint distribution $p_\theta(\mathbf{x}_{0:T})$ is called the *reverse process*, and it is defined as a Markov chain with learned Gaussian transitions starting at $p(\mathbf{x}_T) = \mathcal{N}(\mathbf{x}_T; \mathbf{0}, \mathbf{I})$:

$$p_\theta(\mathbf{x}_{0:T}) := p(\mathbf{x}_T) \prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t), \qquad p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) := \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t)) \tag{1}$$

What distinguishes diffusion models from other types of latent variable models is that the approximate posterior $q(\mathbf{x}_{1:T}|\mathbf{x}_0)$, called the *forward process* or *diffusion process*, is fixed to a Markov chain that gradually adds Gaussian noise to the data according to a variance schedule $\beta_1, \ldots, \beta_T$:

$$q(\mathbf{x}_{1:T}|\mathbf{x}_0) := \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1}), \qquad q(\mathbf{x}_t|\mathbf{x}_{t-1}) := \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t \mathbf{I}) \tag{2}$$

Training is performed by optimizing the usual variational bound on negative log likelihood:

$$\mathbb{E}\left[-\log p_\theta(\mathbf{x}_0)\right] \leq \mathbb{E}_q\left[-\log \frac{p_\theta(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\right] = \mathbb{E}_q\left[-\log p(\mathbf{x}_T) - \sum_{t \geq 1} \log \frac{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_t|\mathbf{x}_{t-1})}\right] =: L \tag{3}$$

The forward process variances β_t can be learned by reparameterization [33] or held constant as hyperparameters, and expressiveness of the reverse process is ensured in part by the choice of Gaussian conditionals in $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$, because both processes have the same functional form when β_t are small [53]. A notable property of the forward process is that it admits sampling $\mathbf{x}_t$ at an arbitrary timestep t in closed form: using the notation $\alpha_t := 1 - \beta_t$ and $\bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$, we have

$$q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I}) \tag{4}$$

> ✎ **the mathematical equations with the red color are the forward process, and with the yellow color is the revers process, and the**

equation in the middle which assigned in number (3) is the learning process for the reverse.

> This paper presents progress in diffusion probabilistic models [53]. A diffusion probabilistic model (which we will call a “diffusion model” for brevity) is a parameterized Markov chain trained using variational inference to produce samples matching the data after finite time. Transitions of this chain are learned to reverse a diffusion process, which is a Markov chain that gradually adds noise to the data in the opposite direction of sampling until signal is destroyed. When the diffusion consists of small amounts of Gaussian noise, it is sufficient to set the sampling chain transitions to conditional Gaussians too, allowing for a particularly simple neural network parameterization. Diffusion models are straightforward to define and efficient to train, but to the best of our knowledge, there has been no demonstration that they are capable of generating high quality samples. We show that diffusion models actually are capable of generating high quality samples, sometimes better than the published results on other types of generative models (Section 4). In addition, we show that a certain parameterization of diffusion models reveals an equivalence with denoising score matching over multiple noise levels during training and with annealed Langevin dynamics during sampling (Section 3.2) [55, 61]. We obtained our best sample quality results using this parameterization (Section 4.2), so we consider this equivalence to be one of our primary contributions. Despite their sample quality, our models do not have competitive log likelihoods compared to other likelihood-based models (our models do, however, have log likelihoods better than the large estimates annealed importance sampling has been reported to produce for energy based models and score matching [11, 55]). We find that the majority of our models’ lossless codelengths are consumed to describe imperceptible image details (Section 4.3). We present a more refined analysis of this phenomenon in the language of lossy compression, and we show that the sampling procedure of diffusion models is a type of progressive decoding that resembles autoregressive decoding along a bit ordering that vastly generalizes what is normally possible with autoregressive models.

✎ **Note that the forward process is fixed we just add noise to the image by using the formula, but the reverse process is the main formula for the diffusion model, where the diffusion model actually learn, but how we can make the model learn by just using the reverse process. A Diffusion Model is trained by finding the reverse Markov transitions**

that maximize the likelihood of the training data. In practice, training equivalently consists of minimizing the variational upper bound on the negative log likelihood.

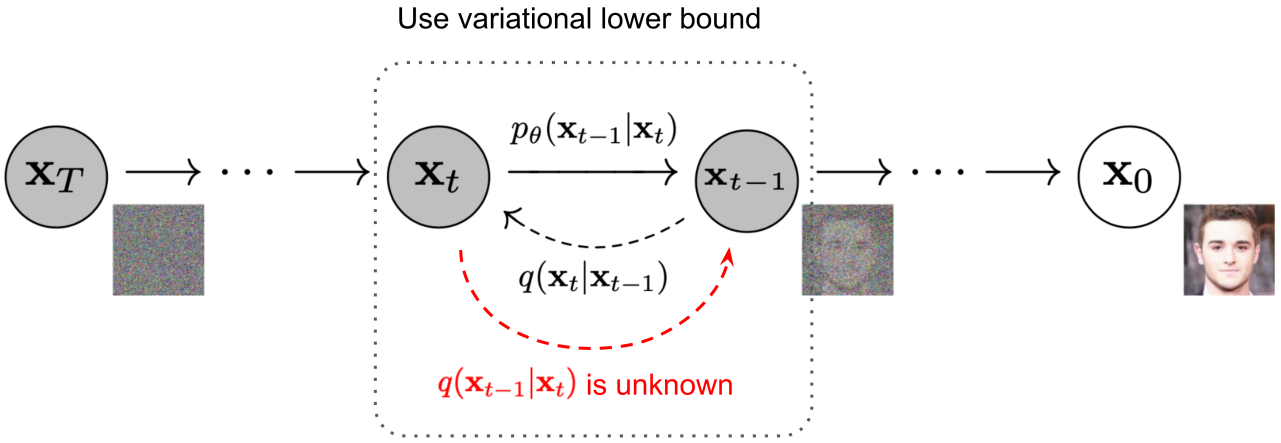
Paper experiments:

4 Experiments

We set $T = 1000$ for all experiments so that the number of neural network evaluations needed during sampling matches previous work [53, 55]. We set the forward process variances to constants increasing linearly from $\beta_1 = 10^{-4}$ to $\beta_T = 0.02$. These constants were chosen to be small relative to data scaled to $[-1, 1]$, ensuring that reverse and forward processes have approximately the same functional form while keeping the signal-to-noise ratio at $\mathbf{x}_T$ as small as possible ($L_T = D_{\mathrm{KL}}(q(\mathbf{x}_T|\mathbf{x}_0) \parallel \mathcal{N}(\mathbf{0}, \mathbf{I})) \approx 10^{-5}$ bits per dimension in our experiments).

To represent the reverse process, we use a U-Net backbone similar to an unmasked PixelCNN++ [52, 48] with group normalization throughout [66]. Parameters are shared across time, which is specified to the network using the Transformer sinusoidal position embedding [60]. We use self-attention at the 16×16 feature map resolution [63, 60]. Details are in Appendix B.

Mathematical explanation:



$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \quad q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1})$$

A nice property o

f the above process is that we can sample $\mathbf{x}_t$ at any arbitrary time step t in a closed form using reparameterization trick. Let $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$:

$$
\begin{aligned}
\mathbf{x}_t &= \sqrt{\alpha_t}\mathbf{x}_{t-1} + \sqrt{1-\alpha_t}\boldsymbol{\epsilon}_{t-1} && \text{;where } \boldsymbol{\epsilon}_{t-1}, \boldsymbol{\epsilon}_{t-2}, \cdots \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \\
&= \sqrt{\alpha_t\alpha_{t-1}}\mathbf{x}_{t-2} + \sqrt{1-\alpha_t\alpha_{t-1}}\bar{\boldsymbol{\epsilon}}_{t-2} && \text{;where } \bar{\boldsymbol{\epsilon}}_{t-2} \text{ merges two Gaussians (*).} \\
&= \dots \\
&= \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon} \\
q(\mathbf{x}_t|\mathbf{x}_0) &= \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I})
\end{aligned}
$$

as we mentioned before, the reverse process is where the model has to learn to do so how to do, by finding the reverse Markov transitions that maximize the likelihood of the training data. In practice, training equivalently consists of minimizing the variational upper bound on the negative log likelihood.

First you should know what is the Evidence Lower Bound (ELBO), Variational Lower Bound (VLB), what is the Variational Autoencoder (VAE) model, how it works, what is the Kull-back Divergence (D_{kl}), cause the variational autoencoder is the heart of the diffusion models.

Basic math concepts to know:

Expectation of a random variable $E_x[f(x)] = \int xf(x)dx$

Chain rule of probability $P(x, y) = P(x|y)P(y)$

Bayes' Theorem $P(x \mid y) = \frac{P(y|x)P(x)}{P(y)}$

Kullback-Leibler Divergence

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

Properties:

- Not symmetric.
- Always ≥ 0
- It is equal to 0 if and only if $P = Q$

$$\log p_\theta(\boldsymbol{x}) = \log p_\theta\ (\boldsymbol{x})$$

$$= \log p_\theta\ (\boldsymbol{x}) \int q_\varphi(\boldsymbol{z}|\boldsymbol{x})d\boldsymbol{z} \qquad \text{Multiply by 1}$$

$$= \int \log p_\theta\ (\boldsymbol{x})\, q_\varphi(\boldsymbol{z}|\boldsymbol{x})d\boldsymbol{z} \qquad \text{Bring inside the integral}$$

$$= E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}[\log p_\theta\ (\boldsymbol{x})] \qquad \text{Definition of expectation}$$

$$= E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_\theta(\boldsymbol{x},\boldsymbol{z})}{p_\theta(\boldsymbol{z}|\boldsymbol{x})}\right] \qquad \text{Apply the equation } p_\theta(\boldsymbol{x}) = \frac{p_\theta(\boldsymbol{x},\boldsymbol{z})}{p_\theta(\boldsymbol{z}|\boldsymbol{x})}$$

$$= E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_\theta(\boldsymbol{x},\boldsymbol{z})q_\varphi(\boldsymbol{z}|\boldsymbol{x})}{p_\theta(\boldsymbol{z}|\boldsymbol{x})q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\right] \qquad \text{Multiply by 1}$$

$$= E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_\theta(\boldsymbol{x},\boldsymbol{z})}{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\right] + E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}{p_\theta(\boldsymbol{z}|\boldsymbol{x})}\right] \qquad \text{Split the expectation}$$

$$= E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_\theta(\boldsymbol{x},\boldsymbol{z})}{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\right] + \underbrace{D_{KL}\left(q_\varphi(\boldsymbol{z}|\boldsymbol{x})\|p_\theta(\boldsymbol{z}|\boldsymbol{x})\right)}_{\geq 0} \qquad \text{Definition of KL divergence}$$

$$\log p_\theta(\boldsymbol{x}) = \underbrace{E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_\theta(\boldsymbol{x},\boldsymbol{z})}{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\right]}_{\textbf{ELBO}} + \underbrace{D_{KL}\left(q_\varphi(\boldsymbol{z}|\boldsymbol{x})\|p_\theta(\boldsymbol{z}|\boldsymbol{x})\right)}_{\geq 0}$$

ELBO = Evidence Lower Bound

Total Compensation = Base Salary + Bonus, where Bonus ≥ 0

$$\log p_\theta(\boldsymbol{x}) \geq E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_\theta(\boldsymbol{x},\boldsymbol{z})}{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\right] = E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_\theta(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})}{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\right] \qquad \text{Chain rule of probability}$$

$$= E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})] + E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p(\boldsymbol{z})}{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\right] \qquad \text{Split the expectation}$$

$$= E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})] - D_{KL}\left(q_\varphi(\boldsymbol{z}|\boldsymbol{x})\|p_\theta(\boldsymbol{z})\right) \qquad \text{Definition of KL divergence}$$

Profit = Revenue - Costs

Now after we understand the ELBO and the D_{KL} derivations, the following derivations for the reverse process by finding the reverse Markov transitions that maximize the likelihood of the training data. In practice,

training equivalently consists of minimizing the variational upper bound on the negative log likelihood.

> **Note that Lvlb is technically an *upper* bound (the negative of the ELBO) which we are trying to minimize, but we refer to it as Lvlb for consistency with the literature.**

thus we can use the variational lower bound to optimize the negative log-likelihood:

$$
\begin{aligned}
-\log p_\theta(\mathbf{x}_0) &\leq -\log p_\theta(\mathbf{x}_0) + D_{\mathrm{KL}}(q(\mathbf{x}_{1:T}|\mathbf{x}_0)\|p_\theta(\mathbf{x}_{1:T}|\mathbf{x}_0)) \\
&= -\log p_\theta(\mathbf{x}_0) + \mathbb{E}_{\mathbf{x}_{1:T}\sim q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\Big[\log \frac{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}{p_\theta(\mathbf{x}_{0:T})/p_\theta(\mathbf{x}_0)}\Big] \\
&= -\log p_\theta(\mathbf{x}_0) + \mathbb{E}_q\Big[\log \frac{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}{p_\theta(\mathbf{x}_{0:T})} + \log p_\theta(\mathbf{x}_0)\Big] \\
&= \mathbb{E}_q\Big[\log \frac{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}{p_\theta(\mathbf{x}_{0:T})}\Big] \\
\text{Let } L_{\mathrm{VLB}} &= \mathbb{E}_{q(\mathbf{x}_{0:T})}\Big[\log \frac{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}{p_\theta(\mathbf{x}_{0:T})}\Big] \geq -\mathbb{E}_{q(\mathbf{x}_0)} \log p_\theta(\mathbf{x}_0)
\end{aligned}
$$

It is also straightforward to get the same result using Jensen's inequality. Say we want to minimize the cross entropy as the learning objective, the following is **optional:

$$
\begin{aligned}
L_{\mathrm{CE}} &= -\mathbb{E}_{q(\mathbf{x}_0)} \log p_\theta(\mathbf{x}_0) \\
&= -\mathbb{E}_{q(\mathbf{x}_0)} \log \Big(\int p_\theta(\mathbf{x}_{0:T}) d\mathbf{x}_{1:T}\Big) \\
&= -\mathbb{E}_{q(\mathbf{x}_0)} \log \Big(\int q(\mathbf{x}_{1:T}|\mathbf{x}_0) \frac{p_\theta(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} d\mathbf{x}_{1:T}\Big) \\
&= -\mathbb{E}_{q(\mathbf{x}_0)} \log \Big(\mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \frac{p_\theta(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\Big) \\
&\leq -\mathbb{E}_{q(\mathbf{x}_{0:T})} \log \frac{p_\theta(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \\
&= \mathbb{E}_{q(\mathbf{x}_{0:T})}\Big[\log \frac{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}{p_\theta(\mathbf{x}_{0:T})}\Big] = L_{\mathrm{VLB}}
\end{aligned}
$$

now here is the Variational lower bound derivation for the revers process:

$$
\begin{aligned}
L_{\text{VLB}} &= \mathbb{E}_{q(\mathbf{x}_{0:T})}\Big[\log \frac{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}{p_\theta(\mathbf{x}_{0:T})}\Big] \\
&= \mathbb{E}_q\Big[\log \frac{\prod_{t=1}^T q(\mathbf{x}_t|\mathbf{x}_{t-1})}{p_\theta(\mathbf{x}_T)\prod_{t=1}^T p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}\Big] \\
&= \mathbb{E}_q\Big[-\log p_\theta(\mathbf{x}_T) + \sum_{t=1}^T \log \frac{q(\mathbf{x}_t|\mathbf{x}_{t-1})}{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}\Big] \\
&= \mathbb{E}_q\Big[-\log p_\theta(\mathbf{x}_T) + \sum_{t=2}^T \log \frac{q(\mathbf{x}_t|\mathbf{x}_{t-1})}{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)} + \log \frac{q(\mathbf{x}_1|\mathbf{x}_0)}{p_\theta(\mathbf{x}_0|\mathbf{x}_1)}\Big] \\
&= \mathbb{E}_q\Big[-\log p_\theta(\mathbf{x}_T) + \sum_{t=2}^T \log \Big(\frac{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)}{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)} \cdot \frac{q(\mathbf{x}_t|\mathbf{x}_0)}{q(\mathbf{x}_{t-1}|\mathbf{x}_0)}\Big) + \log \frac{q(\mathbf{x}_1|\mathbf{x}_0)}{p_\theta(\mathbf{x}_0|\mathbf{x}_1)}\Big] \\
&= \mathbb{E}_q\Big[-\log p_\theta(\mathbf{x}_T) + \sum_{t=2}^T \log \frac{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)}{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)} + \sum_{t=2}^T \log \frac{q(\mathbf{x}_t|\mathbf{x}_0)}{q(\mathbf{x}_{t-1}|\mathbf{x}_0)} + \log \frac{q(\mathbf{x}_1|\mathbf{x}_0)}{p_\theta(\mathbf{x}_0|\mathbf{x}_1)}\Big] \\
&= \mathbb{E}_q\Big[-\log p_\theta(\mathbf{x}_T) + \sum_{t=2}^T \log \frac{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)}{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)} + \log \frac{q(\mathbf{x}_T|\mathbf{x}_0)}{q(\mathbf{x}_1|\mathbf{x}_0)} + \log \frac{q(\mathbf{x}_1|\mathbf{x}_0)}{p_\theta(\mathbf{x}_0|\mathbf{x}_1)}\Big] \\
&= \mathbb{E}_q\Big[\log \frac{q(\mathbf{x}_T|\mathbf{x}_0)}{p_\theta(\mathbf{x}_T)} + \sum_{t=2}^T \log \frac{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)}{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)} - \log p_\theta(\mathbf{x}_0|\mathbf{x}_1)\Big] \\
&= \mathbb{E}_q[\underbrace{D_{\text{KL}}(q(\mathbf{x}_T|\mathbf{x}_0) \parallel p_\theta(\mathbf{x}_T))}_{L_T} + \sum_{t=2}^T \underbrace{D_{\text{KL}}(q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) \parallel p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t))}_{L_{t-1}} \underbrace{- \log p_\theta(\mathbf{x}_0|\mathbf{x}_1}_{L_0}
\end{aligned}
$$

Let's label each component in the variational lower bound loss separately:

$$
\begin{aligned}
L_{\text{VLB}} &= L_T + L_{T-1} + \cdots + L_0 \\
\text{where } L_T &= D_{\text{KL}}(q(\mathbf{x}_T|\mathbf{x}_0) \parallel p_\theta(\mathbf{x}_T)) \\
L_t &= D_{\text{KL}}(q(\mathbf{x}_t|\mathbf{x}_{t+1}, \mathbf{x}_0) \parallel p_\theta(\mathbf{x}_t|\mathbf{x}_{t+1})) \text{ for } 1 \leq t \leq T-1 \\
L_0 &= -\log p_\theta(\mathbf{x}_0|\mathbf{x}_1)
\end{aligned}
$$

> ***for better understanding how the derivations happen, see the [Handwritten Derivations]***

The following is the training, and the sampling algorithms:

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$$
6: **until** converged

Algorithm 2 Sampling

1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = T, \dots, 1$ **do**
3: $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4: $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$

Training algorithm:

In other words:

- we take a random sample x_0 from the real unknown and possibly complex data distribution $q(x_0)$
- we sample a noise level t uniformally between 1 and T (i.e., a random time step)
- we sample some noise from a Gaussian distribution and corrupt the input by this noise at level t (using the nice property defined above)
- the neural network is trained to predict this noise based on the corrupted image x_t (i.e. noise applied on x_0 based on known schedule β_t)

> **In reality, all of this is done on batches of data, as one uses stochastic gradient descent to optimize neural networks, to understand why we use SGD over BGD, read this section Stochastic Gradient Descent vs Batch Gradient Descent.**

1: repeat
2: $x_0 \sim q(x_0)$ We take a sample from our dataset.
3: $t \sim \text{Uniform}(1, \dots, T)$ We generate a random number t, between 1 and T.
4: $\epsilon \sim \mathcal{N}(0, I)$ We sample some noise.
5:

$$\nabla_\theta \left\| \boldsymbol{\epsilon}_t - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_t, t) \right\|^2$$

We add noise to our image, and we train the model to learn to predict the amount of noise present in it.

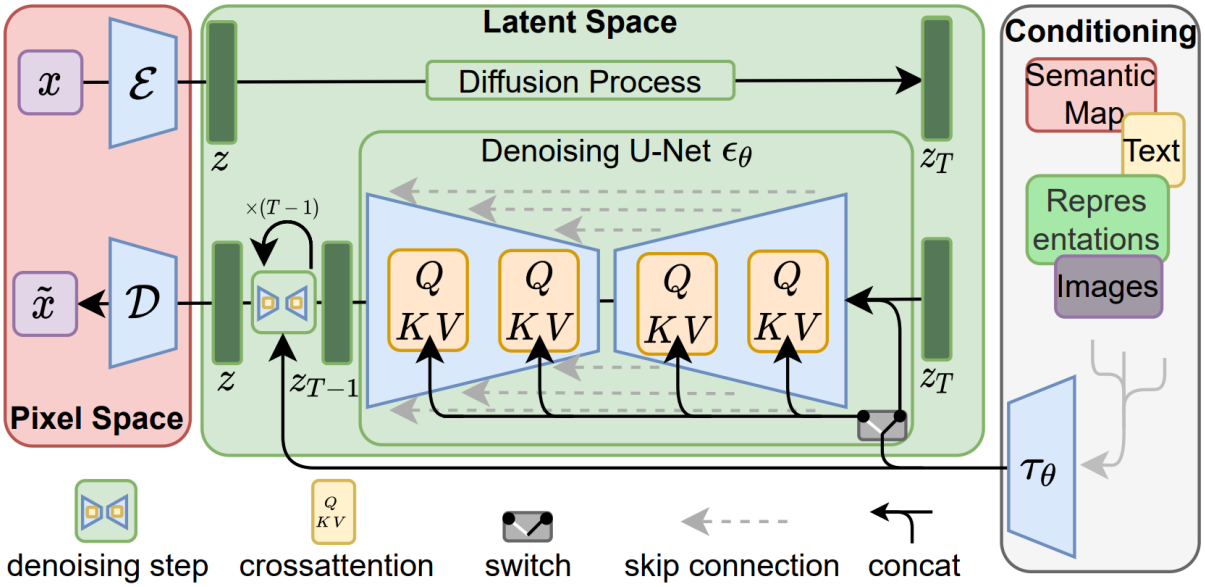
Sampling algorithm:

Algorithm 2 Sampling

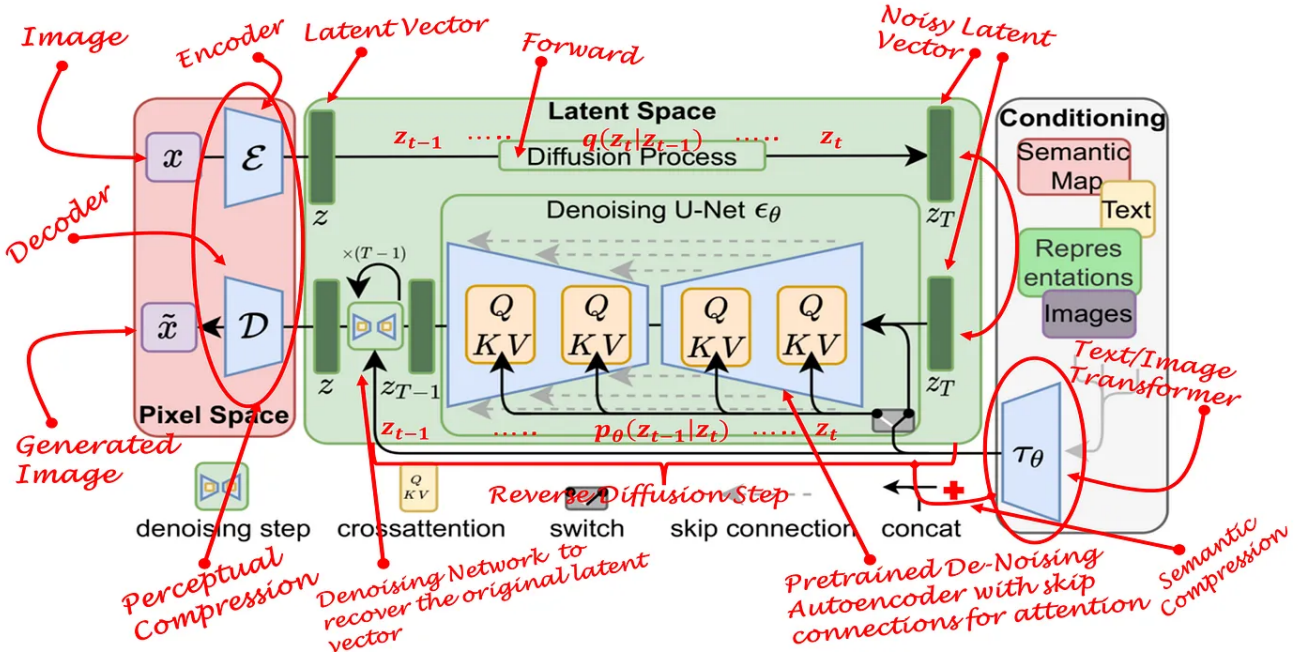
1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ We sample some noise
2: **for** $t = T, \ldots, 1$ **do**
3: $\quad \mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4: $\quad \mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$

We keep denoising the image progressively for T steps.

Model Architecture:



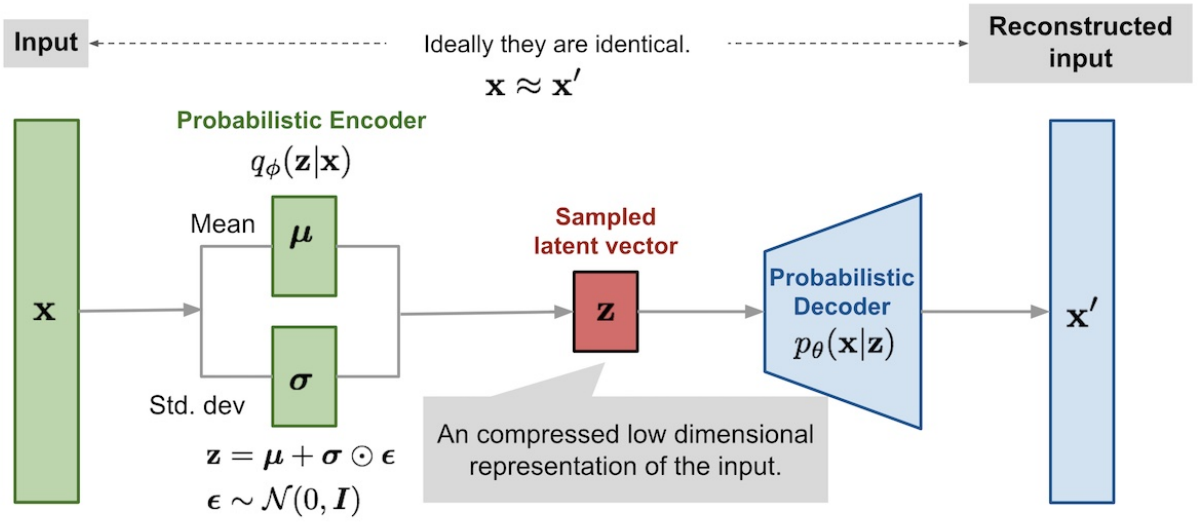
Latent diffusion can reduce the memory and compute complexity by applying the diffusion process over a lower dimensional *latent* space, instead of using the actual pixel space. This is the key difference between standard diffusion and latent diffusion models: **in latent diffusion the model is trained to generate latent (compressed) representations of the images.**

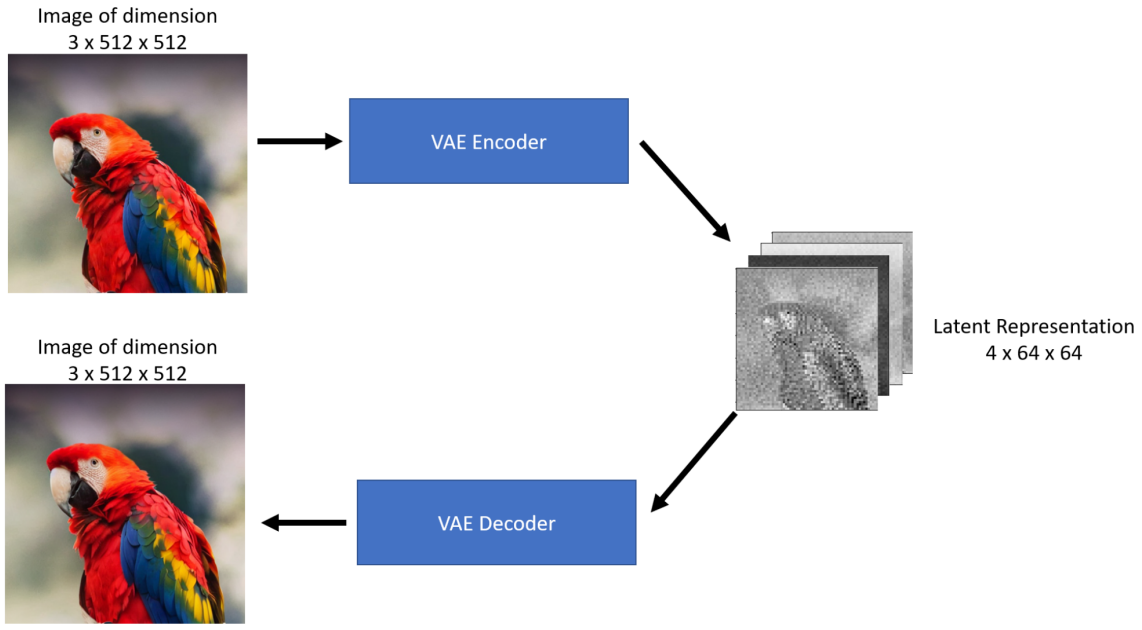


The Pipeline of the model: What's their role in the Stable diffusion pipeline

There are three main components in latent diffusion.

1. An autoencoder (VAE).
   An autoencoder contains two parts -
   1. `Encoder` takes an image as input and converts it into a low dimensional latent representation
   2. `Decoder` takes the latent representation and converts it back into an image





As we can see above, the Encoder acts like a compressor that squishes the image into lower dimensions and the decoder recreates the original image back from the compressed version.

```
p = FastDownload().download('https://lafeber.com/pet-birds/wp-content/uploads/2018/06/Scarlet-Macaw-2.jpg')
img = load_image(p)
print(f"Dimension of this image: {np.array(img).shape}") img
```

```
Dimension of this image: (512, 512, 3)
```

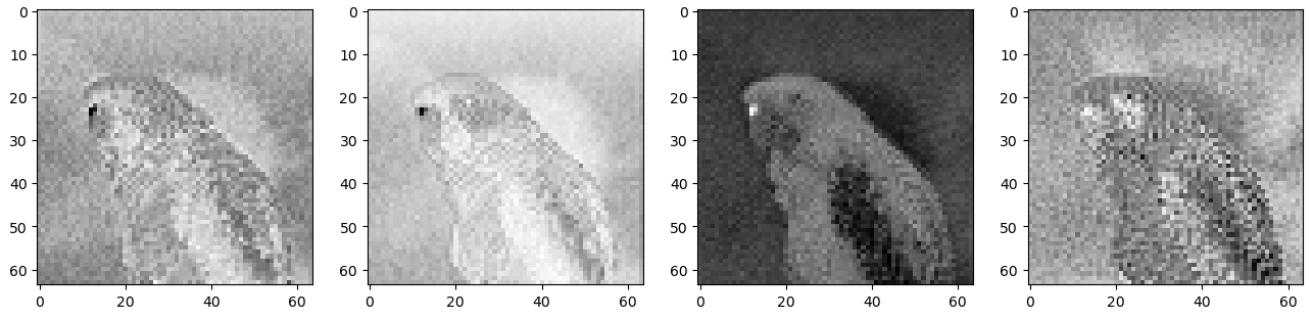
```
![[Pasted image 20240322160606.png]]
Now let's compress this image by using the VAE encoder, we will be using the `pil_to_latents` helper function.
```

```
latent_img = pil_to_latents(img)
print(f"Dimension of this latent representation: {latent_img.shape}")
```

```
Dimension of this latent representation: torch.Size([1, 4, 64, 64])
```

As we can see how the VAE compressed a 3 x 512 x 512 dimension image into a 4 x 64 x 64 image. That's a compression ratio of 48x! Let's visualize these four channels of latent representations.

```
fig, axs = plt.subplots(1, 4, figsize=(16, 4)) []for c in range(4):
[])axs[c].imshow(latent_img[0][c].detach().cpu(), cmap='Greys')
```



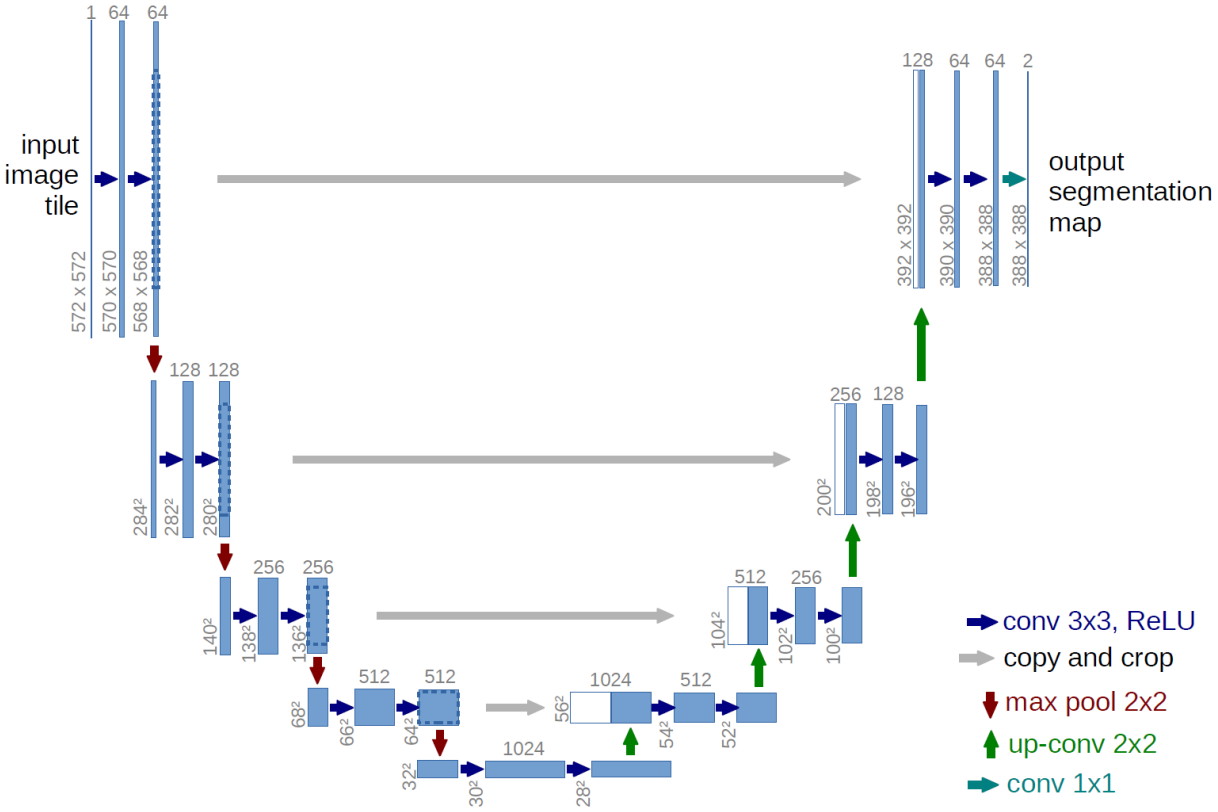
This latent representation in theory should capture a lot of information about the original image. Let's use the decoder on this representation to see what we get back. For this, we will use the `latents_to_pil` helper function.

```
decoded_img = latents_to_pil(latent_img)
decoded_img[0]
```

```
![[Pasted image 20240322161321.png]]
As we can see from the figure above VAE decoder was able to recover the
original image from a 48x compressed latent representation. That's
impressive!
```

If you look closely at the decoded image, it's not the same as the original image, notice the difference around the eyes. That's why VAE encoder/decoder is not a lossless compression.

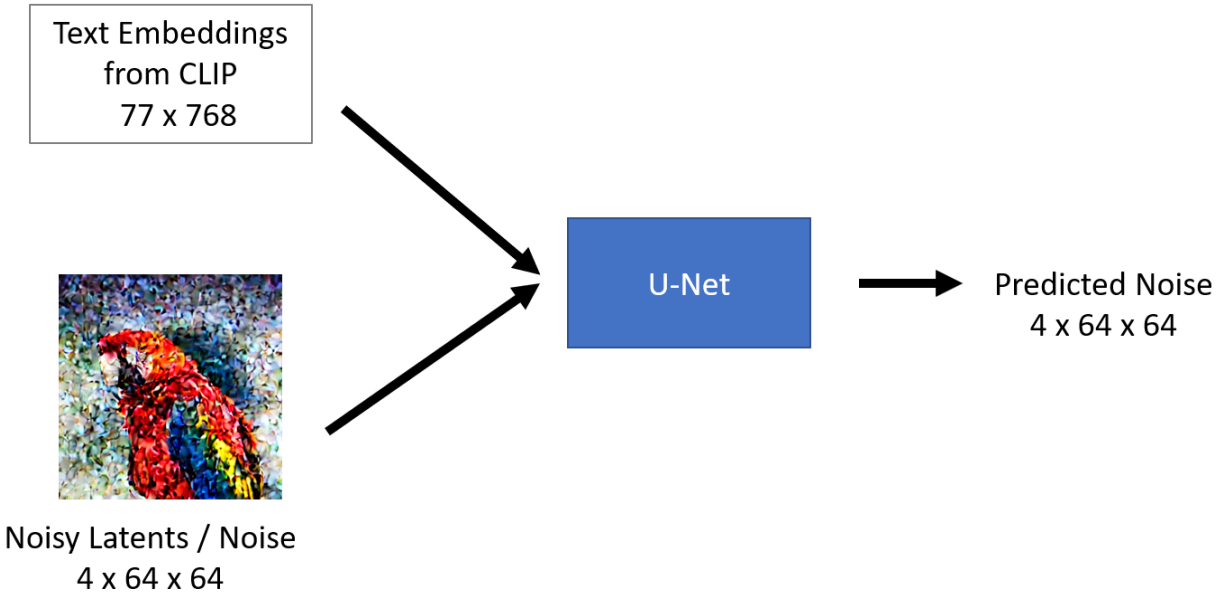
2. A U-Net.



The U-Net model takes two inputs -

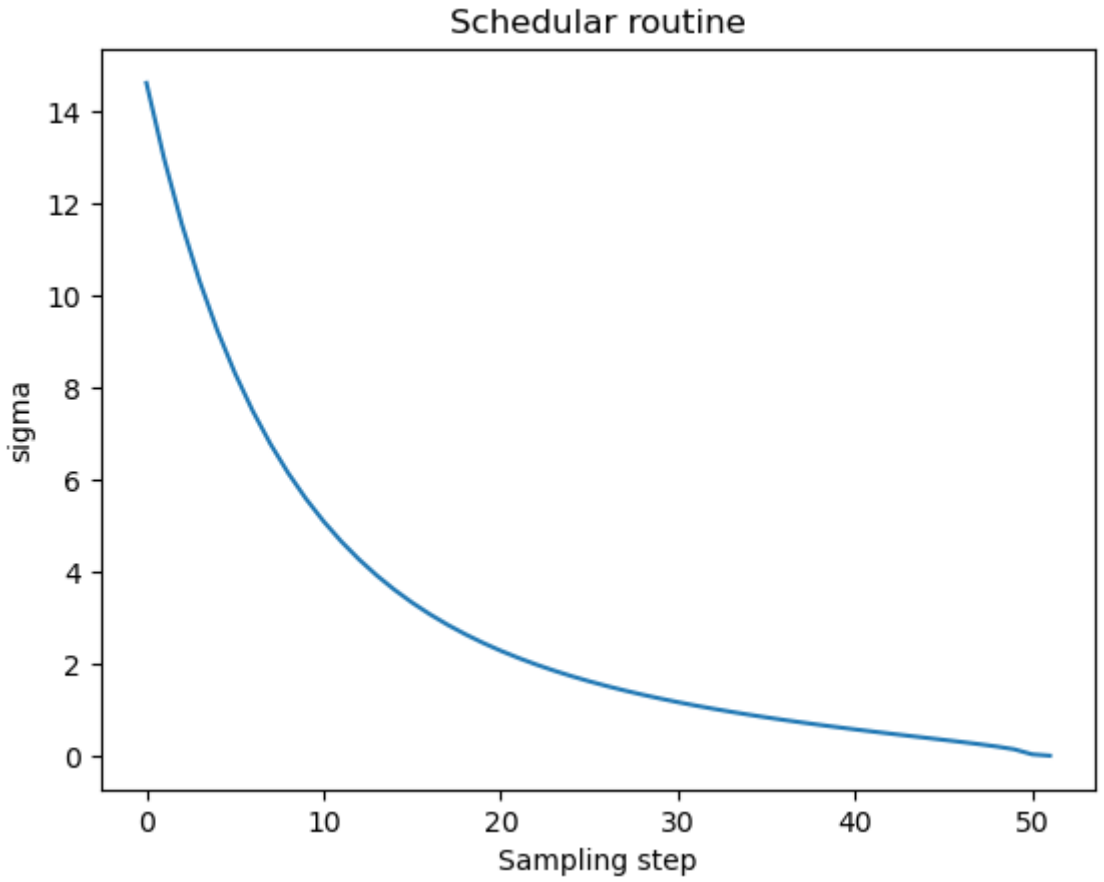
1. `Noisy latent` or `Noise` - Noisy latents are latents produced by a VAE encoder (in case an initial image is provided) with added noise or it can take pure noise input in case we want to create a random new image based solely on a textual description
2. `Text embeddings` - CLIP-based embedding generated by input textual prompts

The output of the U-Net model is the predicted noise residual which the input noisy latent contains. In other words, it predicts the noise which is subtracted from the noisy latents to return the original de-noised latents.

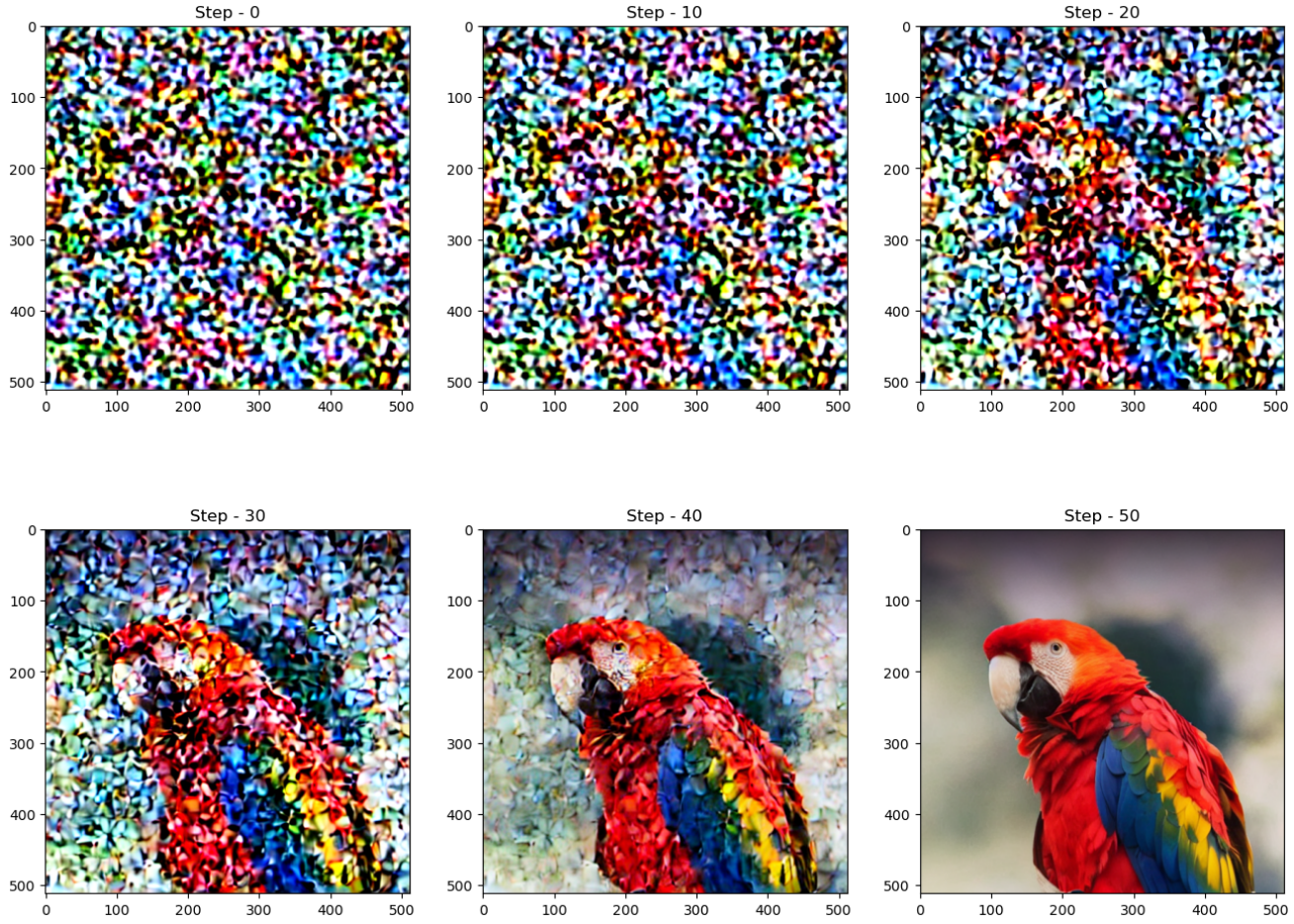


```
from diffusers import UNet2DConditionModel, LMSDiscreteScheduler [][]
## Initializing a scheduler
scheduler = LMSDiscreteScheduler(beta_start=0.00085, beta_end=0.012,
beta_schedule="scaled_linear", num_train_timesteps=1000)
## Setting number of sampling steps
scheduler.set_timesteps(51)

unet = UNet2DConditionModel.from_pretrained("CompVis/stable-diffusion-
v1-4", subfolder="unet", torch_dtype=torch.float16).to("cuda")
```

As you may have noticed from code above, we not only imported `unet` but also a `scheduler`. The purpose of a `schedular` is to determine how much noise to add to the latent at a given step in the diffusion process. Let's visualize the schedular function -



The diffusion process follows this sampling schedule where we start with high noise and gradually denoise the image. Let's visualize this process -



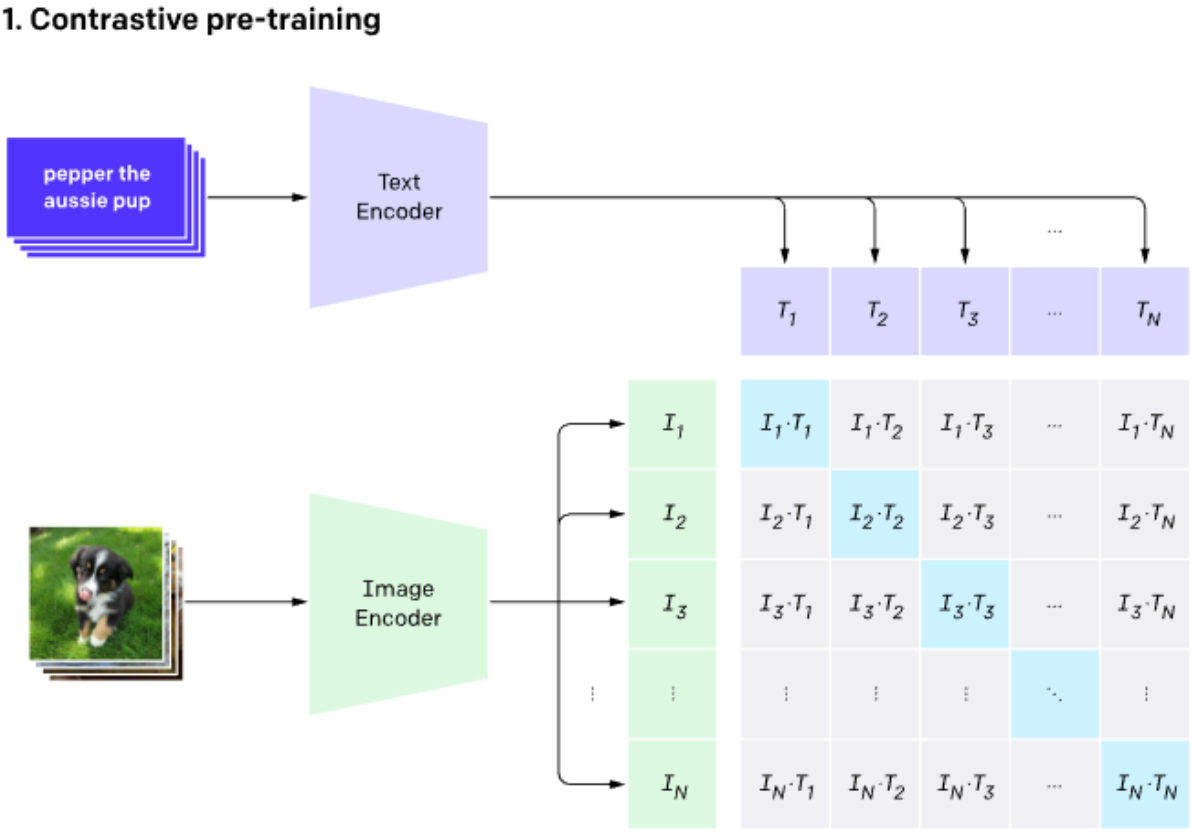
Let’s see how a U-Net removes the noise from the image. Let’s start by adding some noise to the image.

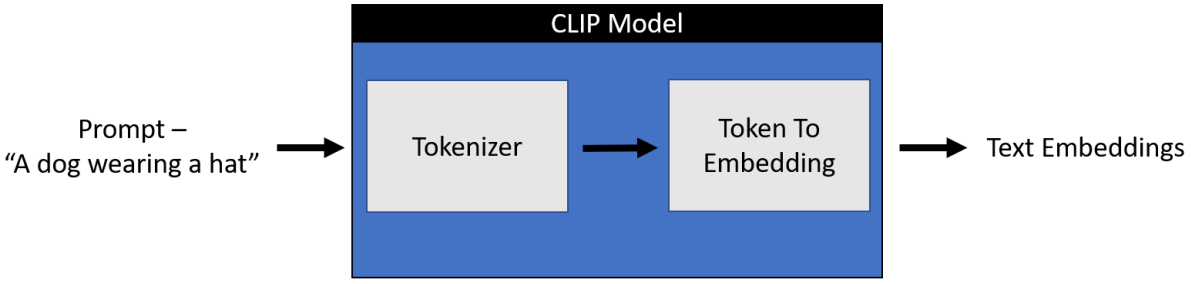
```
prompt = [""]

text_input = tokenizer(prompt, padding="max_length",
max_length=tokenizer.model_max_length, truncation=True,
return_tensors="pt")

with torch.no_grad():
text_embeddings = text_encoder(text_input.input_ids.to("cuda"))[0]

latent_model_input
=torch.cat([encoded_and_noised.to("cuda").float()]).half() with
torch.no_grad():
noise_pred = unet(latent_model_input, 40,
encoder_hidden_states=text_embeddings)["sample"]

latents_to_pil(encoded_and_noised- noise_pred)[0]
```

```
![[Pasted image 20240322163243.png]]
As we can see above the U-Net output is clearer than the original noisy input passed.
```

2. A text-encoder, *e.g.* CLIP's Text Encoder.

   Any machine learning model doesn't understand text data. For any model to understand text data, we need to convert this text into numbers that hold the meaning of the text, referred to as `embeddings`. The process of converting a text to a number can be broken down into two parts -
   1. ***Tokenizer*** - Breaking down each word into sub-words and then using a lookup table to convert them into a number
   2. ***Token-To-Embedding Encoder*** - Converting those numerical sub-words into a representation that contains the representation of that text





Stable diffusion only uses a CLIP trained encoder for the conversion of text to embeddings. This becomes one of the inputs to the U-net. On a high level, CLIP uses an image encoder and text encoder to create embeddings that are similar in latent space. This similarity is more precisely defined as a Contrastive objective.

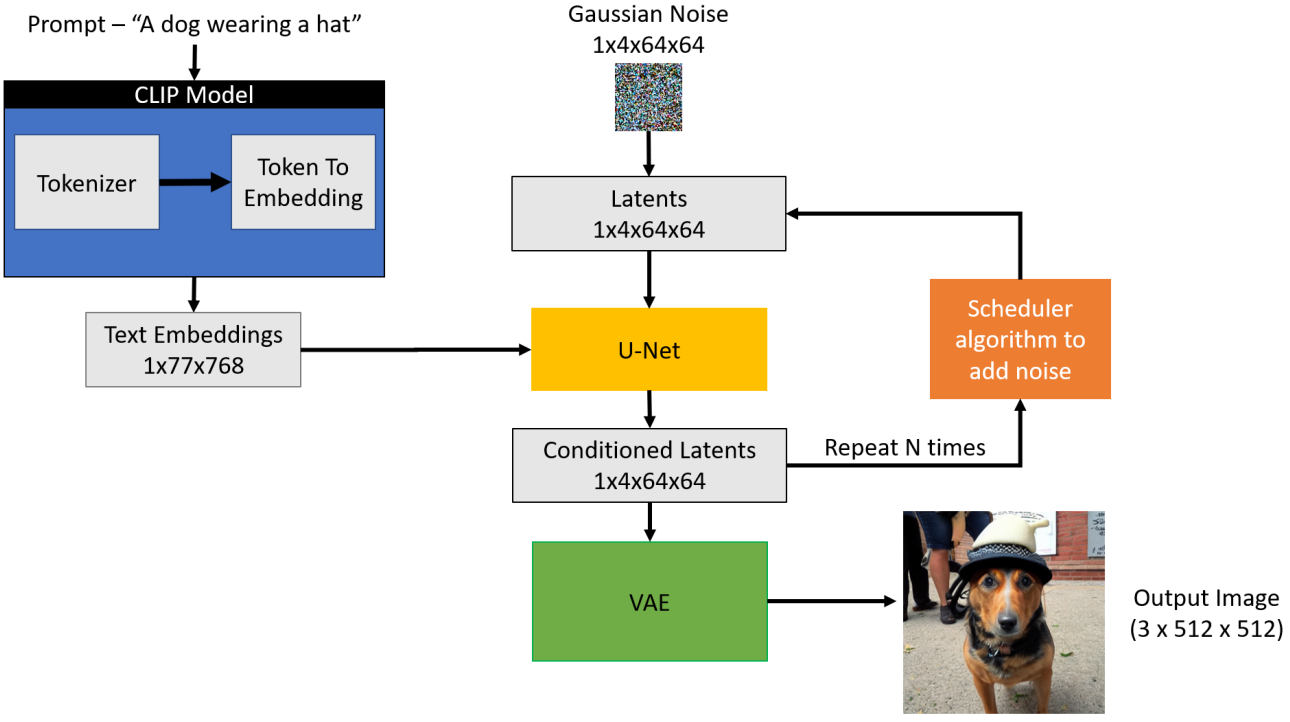
```
tokenizer = CLIPTokenizer.from_pretrained("openai/clip-vit-large-
patch14", torch_dtype=torch.float16)

text_encoder = CLIPTextModel.from_pretrained("openai/clip-vit-large-
patch14", torch_dtype=torch.float16).to("cuda")

prompt = ["a dog wearing hat"] []tok =tokenizer(prompt,
padding="max_length", max_length=tokenizer.model_max_length,
truncation=True, return_tensors="pt") []print(tok.input_ids.shape) []tok
```

```
torch.Size([1, 77])
{'input_ids': tensor([[49406,   320,  1929,  3309,  3801, 49407, 49407,
49407, 49407, 49407,
         49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407,
49407,
         49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407,
49407,
         49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407,
49407,
         49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407,
49407,
         49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407,
49407,
         49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407, 49407,
49407,
         49407, 49407, 49407, 49407, 49407, 49407, 49407]]),
'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0,
          0, 0, 0, 0, 0]])}
```

What's their role in the Stable diffusion pipeline

Latent diffusion uses the U-Net to gradually subtract noise in the latent space over several steps to reach the desired output. With each step, the amount of noise added to the latents is reduced till we reach the final de-noised output. U-Nets were first introduced by this paper for Biomedical image segmentation. The U-Net has an encoder and a decoder which are comprised of ResNet blocks. The stable diffusion U-Net also has cross-attention layers to provide them with the ability to condition the output based on the text description provided. The Cross-attention layers are added to both the encoder and the decoder part of the U-Net usually between ResNet blocks. You can learn more about this U-Net architecture here.



The stable diffusion model takes the textual input and a seed. The textual input is then passed through the CLIP model to generate textual embedding of size 77x768 and the seed is used to generate Gaussian noise of size 4x64x64 which becomes the first latent image representation.

Next, the U-Net iteratively denoises the random latent image representations while conditioning on the text embeddings. The output of the U-Net is predicted noise residual, which is then used to compute conditioned latents via a scheduler algorithm. This process of denoising and text conditioning is repeated N times (We will use 50) to retrieve a better latent image representation. Once this process is

complete, the latent image representation (4x64x64) is decoded by the VAE decoder to retrieve the final output image (3x512x512).

High-Resolution Image Synthesis with Latent Diffusion Models

1. Paper Introduction:

> Diffusion Models [82] are probabilistic models designed to learn a data distribution p(x) by gradually denoising a normally distributed variable, which corresponds to learning the reverse process of a fixed Markov Chain of length T. For image synthesis, the most successful models [15,30,72] rely on a reweighted variant of the variational lower bound on p(x), which mirrors denoising score-matching [85]. These models can be interpreted as an equally weighted sequence of denoising autoencoders θ(xt, t); t = 1 . . . T, which are trained to predict a denoised variant of their input xt, where xt is a noisy version of the input x. The corresponding objective can be simplified to (Sec. B)
>
> $$L_{DM} = \mathbb{E}_{x,\epsilon\sim\mathcal{N}(0,1),t}\left[\|\epsilon - \epsilon_\theta(x_t, t)\|_2^2\right], \qquad (1)$$
>
> with t uniformly sampled from {1, . . . , T}. Generative Modeling of Latent Representations With our trained perceptual compression models consisting of E and D, we now have access to an efficient, low-dimensional latent space in which high-frequency, imperceptible details are abstracted away. Compared to the high-dimensional pixel space, this space is more suitable for likelihood-based generative models, as they can now (i) focus on the important, semantic bits of the data and (ii) train in a lower dimensional, computationally much more efficient space. Unlike previous work that relied on autoregressive, attention-based transformer models in a highly compressed, discrete latent space [23,66,103], we can take advantage of image-specific inductive biases that our model offers. This d to work with the two-dimensional structure of our learned latent space $z = E(x)$, we can use relatively mild compression rates and achieve very good reconstructions. This is in contrast to previous works [23, 66], which relied on an arbitrary 1D ordering of the learned space z to model its distribution autoregressively and thereby ignored much of the

inherent structure of z. Hence, our compression model preserves details of x better (see Tab. 8). The full objective and training details can be found in the supplement. 3.2. Latent Diffusion Models Diffusion Models [82] are probabilistic models designed to learn a data distribution p(x) by gradually denoising a normally distributed variable, which corresponds to learning the reverse process of a fixed Markov Chain of length T. For image synthesis, the most successful models [15,30,72] rely on a reweighted variant of the variational lower bound on p(x), which mirrors denoising score-matching [85]. These models can be interpreted as an equally weighted sequence of denoising autoencoders $\theta(xt, t)$; $t = 1 \dots T$, which are trained to predict a denoised variant of their input x_t, where x_t is a noisy version of the input x. The corresponding objective can be simplified to (Sec. B)
$LDM = Ex, \sim N(0, 1), thk - \theta(xt, t)k2^2i$, (1) with t uniformly sampled from {1, . . . , T}. Generative Modeling of Latent Representations With our trained perceptual compression models consisting of E and D, we now have access to an efficient, low-dimensional latent space in which high-frequency, imperceptible details are abstracted away. Compared to the high-dimensional pixel space, this space is more suitable for likelihood-based generative models, as they can now (i) focus on the important, semantic bits of the data and (ii) train in a lower dimensional, computationally much more efficient space. Unlike previous work that relied on autoregressive, attention-based transformer models in a highly compressed, discrete latent space [23,66,103], we can take advantage of image-specific inductive biases that our model offers. This Semantic Map cross-attention Latent Space Conditioning Text Diffusion Process denoising step switch skip connection Repres entations Pixel Space Images Denoising U-Net concat Figure 3. We condition LDMs either via concatenation or by a more general cross-attention mechanism. See Sec. 3.3 includes the ability to build the underlying U-Net primarily from 2D convolutional layers, and further focusing the objective on the perceptually most relevant bits using the reweighted bound, which now reads
$LLDM := EE(x), \sim N(0, 1), thk - \theta(zt, t)k2^2i$. (2) The neural backbone $\theta(\circ, t)$ of our model is realized as a time-conditional U-Net [71]. Since the forward process is fixed, z_t can be efficiently obtained from E during training, and samples from $p(z)$ can be decoded to image space with a single pass through D.

3.3. Conditioning Mechanisms

Similar to other types of generative models [56, 83], diffusion models are in principle capable of modeling conditional distributions of the form $p(z|y)$. This can be implemented with a conditional denoising autoencoder $\epsilon_\theta(z_t, t, y)$ and paves the way to controlling the synthesis process through inputs y such as text [68], semantic maps [33, 61] or other image-to-image translation tasks [34].

In the context of image synthesis, however, combining the generative power of DMs with other types of conditionings beyond class-labels [15] or blurred variants of the input image [72] is so far an under-explored area of research.

We turn DMs into more flexible conditional image generators by augmenting their underlying UNet backbone with the cross-attention mechanism [97], which is effective for learning attention-based models of various input modalities [35, 36]. To pre-process y from various modalities (such as language prompts) we introduce a domain specific encoder τ_θ that projects y to an intermediate representation $\tau_\theta(y) \in \mathbb{R}^{M \times d_\tau}$, which is then mapped to the intermediate layers of the UNet via a cross-attention layer implementing $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) \cdot V$, with

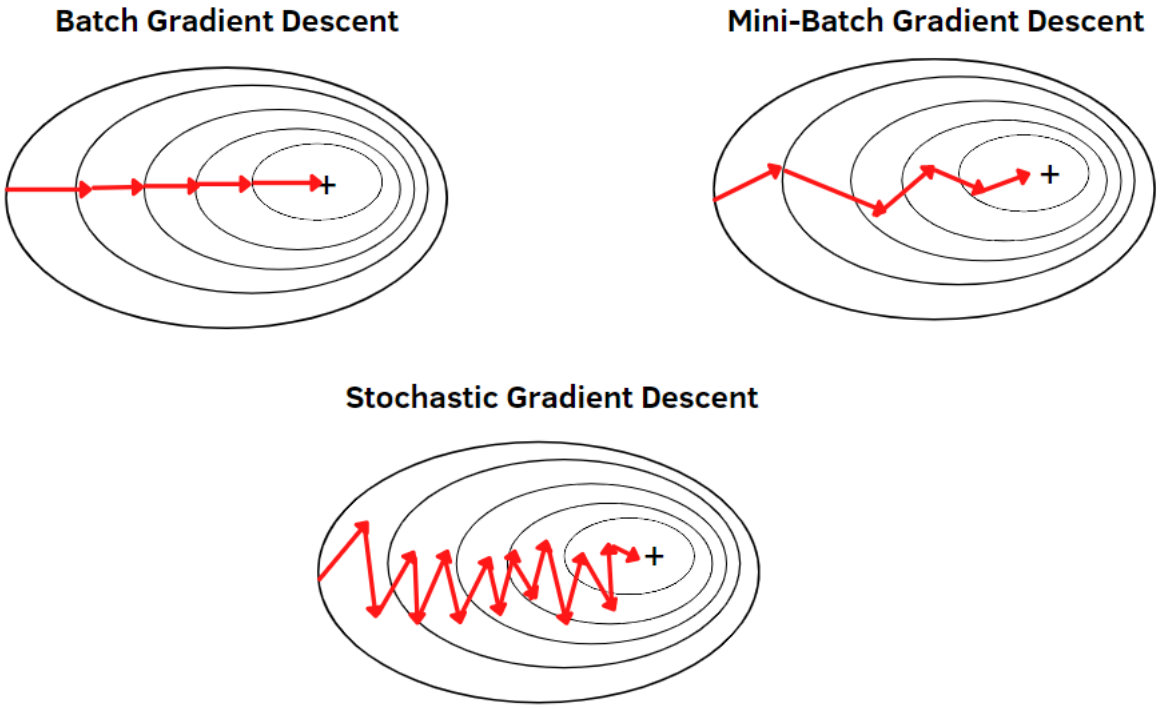
$$Q = W_Q^{(i)} \cdot \varphi_i(z_t), \; K = W_K^{(i)} \cdot \tau_\theta(y), \; V = W_V^{(i)} \cdot \tau_\theta(y).$$

2. Paper Experiments:

4. Experiments

LDMs provide means to flexible and computationally tractable diffusion based image synthesis of various image modalities, which we empirically show in the following. Firstly, however, we analyze the gains of our models compared to pixel-based diffusion models in both training and inference. Interestingly, we find that *LDMs* trained in *VQ*-regularized latent spaces sometimes achieve better sample quality, even though the reconstruction capabilities of *VQ*-regularized first stage models slightly fall behind those of their continuous counterparts, *cf*. Tab. 8. A visual comparison between the effects of first stage regularization schemes on *LDM* training and their generalization abilities to resolutions $> 256^2$ can be found in Appendix D.1. In E.2 we list details on architecture, implementation, training and evaluation for all results presented in this section.

Stochastic Gradient Descent vs Batch Gradient Descent:

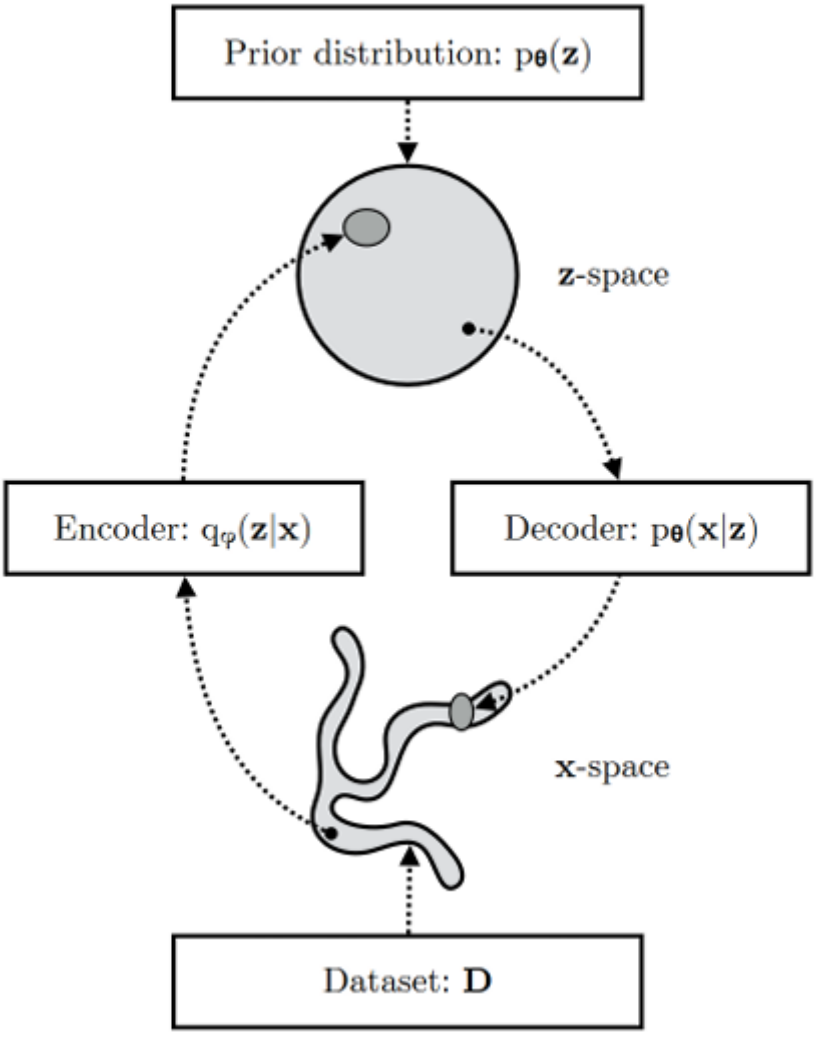


1. **Batch Gradient Descent (BGD)**:
    - In BGD, the model parameters are updated using the gradients computed over the entire dataset.
    - This means that for each iteration, the gradients are calculated by considering the entire dataset, leading to potentially slower updates.
    - BGD ensures a more precise estimation of the gradient, as it considers the complete dataset.
    - However, in the context of large datasets, BGD can be computationally expensive and memory-intensive, as it requires storing and processing the entire dataset at once.
2. **Stochastic Gradient Descent (SGD)**:
    - In SGD, the model parameters are updated using the gradient computed from a single randomly chosen data point or a small subset of data points (mini-batch).
    - This results in faster updates since only a small portion of the dataset is considered for each iteration.
    - SGD introduces more noise in the parameter updates due to its reliance on individual or small subsets of data points.
    - Despite the noise, SGD can escape local minima more easily and can converge faster, especially in noisy or high-dimensional datasets.
    - Additionally, SGD is less memory-intensive as it only requires processing a single data point or a small subset at a time.

In the context of diffusion models, SGD is often preferred over BGD for several reasons:

- **Efficiency**: Diffusion models often deal with large datasets or high-dimensional data. SGD's efficiency in terms of memory usage and computational speed makes it more practical for these scenarios compared to BGD.
- **Robustness to Noise**: Diffusion models often involve noisy data or complex interactions. SGD's stochastic nature helps it navigate through noise and converge to a reasonable solution.
- **Scalability**: As datasets grow larger, the computational and memory requirements of BGD become prohibitive. SGD's ability to handle large datasets in a more scalable manner makes it a preferred choice.

Variational Autoencoder (VAE):



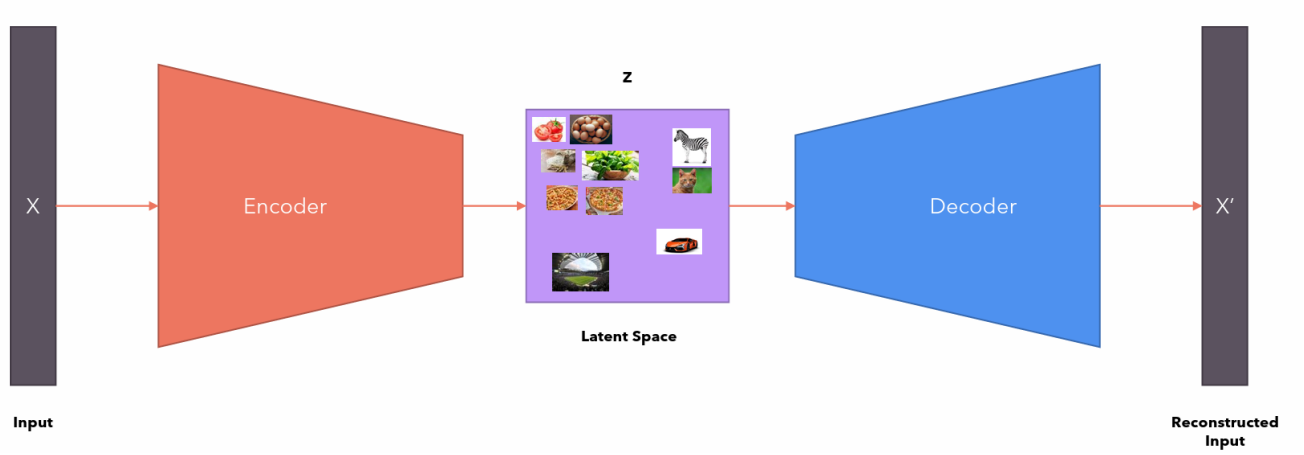
The Variational Autoencoder (VAE) is a type of generative model used in machine learning and artificial intelligence. It's a probabilistic model that aims to learn a low-dimensional representation of input data in an unsupervised manner. VAEs are particularly popular for tasks like generating new data samples that resemble the training data.

Here's how a VAE typically works:

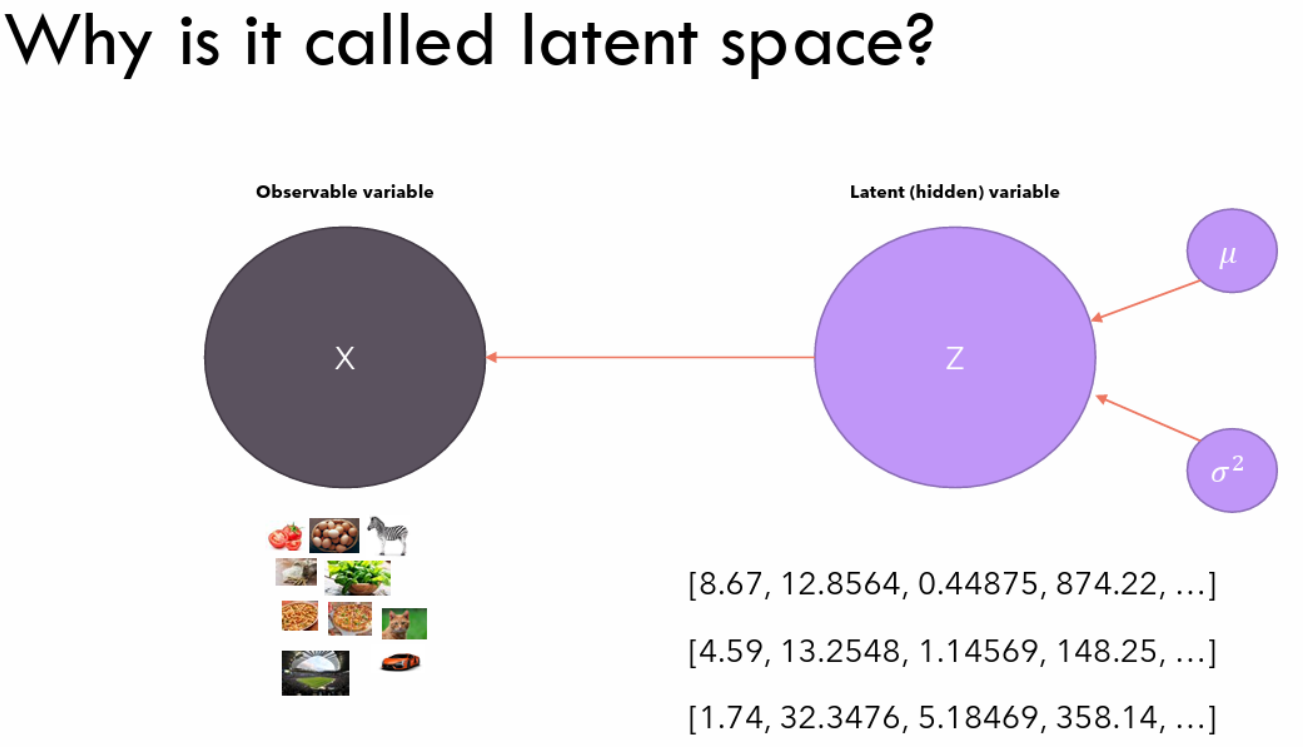
1. **Encoder Network (Recognition Model):** The encoder network takes input data (such as images, text, etc.) and maps it to a probability distribution in a latent space (usually Gaussian distribution). This network learns to encode the input data into a latent representation. The encoder network can be a convolutional neural network (CNN) for image data or a recurrent neural network (RNN) for sequential data like text.
2. **Latent Space:** The latent space is a lower-dimensional space where each point represents a potential encoding of the input data. This space is often constrained to have certain properties, such as a Gaussian distribution, to facilitate sampling.
3. **Sampling:** Once the encoder network has produced the parameters of the probability distribution in the latent space (mean and variance), a point is sampled from this distribution. This sampled point represents the latent representation of the input data.
4. **Decoder Network (Generative Model):** The decoder network takes the sampled latent point and reconstructs the input data from it. It learns to decode the latent representation back into the original data space. Like the encoder, the decoder can be a CNN, RNN, or another architecture suitable for the data type.
5. **Reconstruction Loss:** The reconstruction loss measures the difference between the input data and the data reconstructed by the decoder. This loss is typically the cross-entropy loss for binary data (e.g., images) or the mean squared error for continuous data.
6. **Regularization Loss (KL Divergence):** In addition to the reconstruction loss, VAEs include a regularization term called the KL divergence. This term encourages the latent space to follow a specific distribution, often a unit Gaussian. It helps ensure that the latent space is continuous and well-structured, facilitating meaningful interpolation and generation of new data samples.
7. **Training:** During training, the VAE optimizes a combination of the reconstruction loss and the KL divergence regularization term. The goal is to

minimize the reconstruction loss while keeping the latent space close to the desired distribution.

8. **Generation:** Once trained, the decoder network can be used to generate new data samples by sampling points from the latent space and decoding them. By exploring different regions of the latent space, the VAE can generate diverse and realistic-looking data samples.

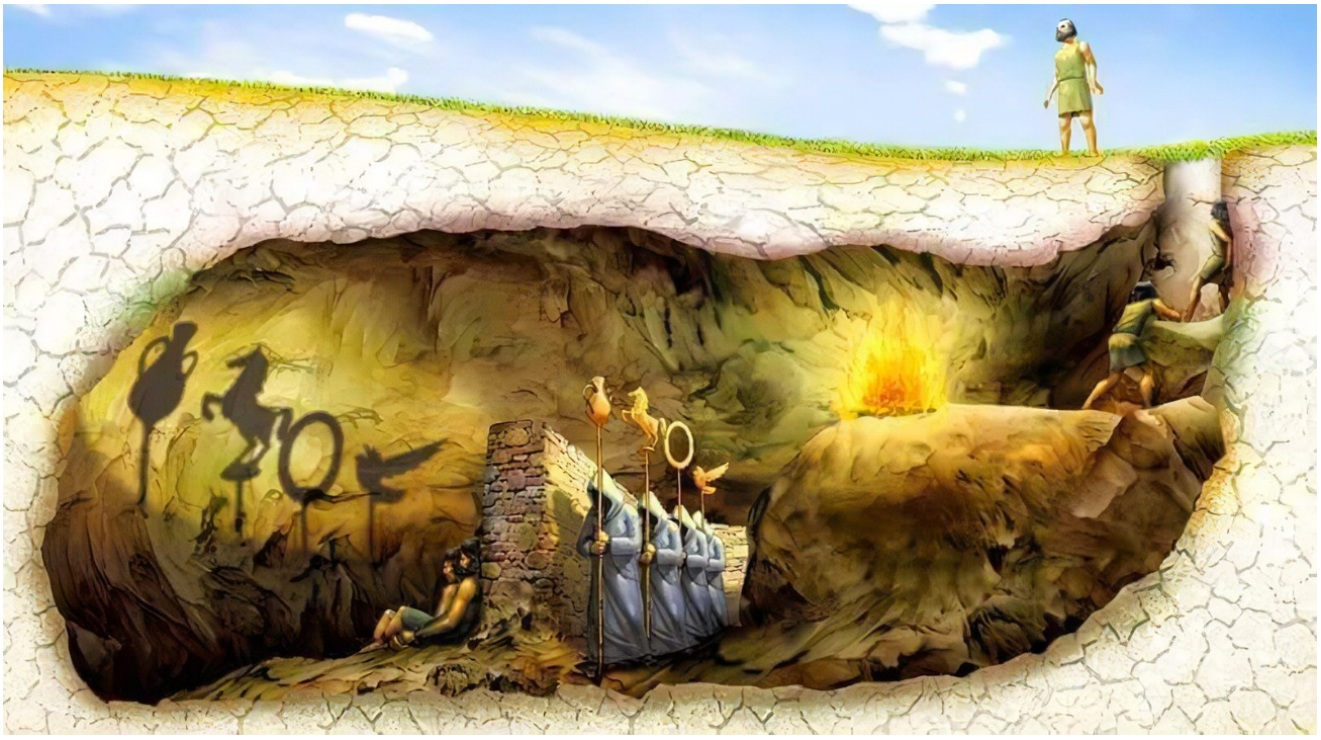


Just like when you use Python to generate a random number between 1 and 100, you're sampling from a uniform (pseudo)random distribution between 1 and 100. In the same way, we can sample from the latent space in order to generate a random vector, give it to the decoder and generate new data.



To understand the latent space:

*By using Plato’s allegory of the cave



In this allegory, we see that the prisoners see the shadows of the statues, and they believe that what they see are the actual objects (the observable data). However, at the same time, the actual objects are behind them (the latent, hidden data).

> ***I am not going to get deep in the VAE, but i am going to explain the most important things we need to know for the diffusion models: ELBO, Divergence Kull-back, VLB, and the derivations we need to know.***

ELBO:

In the context of Variational Autoencoders (VAEs), the Evidence Lower Bound (ELBO) is a crucial concept used in the training and evaluation of these models. VAEs are generative models that learn a probabilistic representation of input data, typically images or sequences, in a latent space. They consist of two main components: an encoder network and a decoder network.

The ELBO serves as a lower bound on the log likelihood of the data and plays a key role in the optimization process during training. It is derived from the

variational inference framework, where the goal is to approximate the true posterior distribution over latent variables given the observed data.

Here's how the ELBO is formulated in the context of VAEs:

Let $p(x, z)$ denote the joint distribution of data x and latent variables z, and let $q(z \mid x)$ be the variational distribution, which approximates the true posterior p(z|x). The ELBO is defined as:

$$ELBO = Eq(z \mid x)[logp(x \mid z)] - KL[q(z \mid x) \mid\mid p(z)]$$

where:

- $Eq(z \mid x)[logp(x \mid z)]$ is the reconstruction term, representing the expected log-likelihood of the data under the decoder distribution.
- $KL[q(z \mid x) \mid\mid p(z)]$ is the Kullback-Leibler (KL) divergence between the variational distribution and the prior distribution over the latent space. This term encourages the variational distribution to stay close to the prior distribution, acting as a regularization term.

The ELBO can also be interpreted as the negative of the evidence gap, i.e., the difference between the marginal likelihood of the data and the KL divergence between the variational distribution and the prior.

During training, the VAE aims to maximize the ELBO with respect to the model parameters (encoder and decoder networks). Maximizing the ELBO encourages the model to learn a good representation of the data in the latent space while keeping the variational distribution close to the prior.

Puzzle analogy for the evidence lower bound(ELBO):

Alright, imagine you have a big puzzle to solve, but you don't have the complete picture. You have some pieces, but not all of them. Now, you want to figure out what the complete picture might look like. This is kind of what Variational Autoencoders (VAEs) do but with data instead of puzzles.

Let's break it down:

1. **Puzzle Pieces (Data):** In our case, these are the pictures, like photos of animals or houses, anything you want to learn about.
2. **Complete Picture (Latent Space):** This is like the big picture of the puzzle. We can't see it directly, but we want to figure out what it might look like. In

VAEs, we call this the "latent space." It's a space where each point represents a different possible complete picture.

3. **Decoder (Putting Pieces Together):** Just like you might put puzzle pieces together to see the complete picture, the decoder in a VAE tries to take a point from the latent space and turn it into a picture.
4. **Encoder (Finding Pieces):** Now, if we have a picture, we might want to figure out what puzzle pieces we used to make it. The encoder in a VAE tries to do this. It takes a picture and tries to find the point in the latent space that might have been used to make it.
5. **Evidence Lower Bound (ELBO):** Now, the ELBO is like a helpful friend who tells us how good our guesses are. It helps us make sure our guesses are close to reality. In our puzzle analogy, the ELBO is like a measure that checks how well our guessed complete picture matches the real complete picture and how well the pieces we think were used to make the picture match the actual pieces.

So, when we're training a VAE, we want to adjust our guesses (the decoder and encoder) to make the ELBO as high as possible. This means our guessed complete picture looks more like the real complete picture, and the pieces we think we used to make it match the actual pieces better.

In simpler terms, the ELBO helps us make sure that our VAE learns to create good pictures and figure out which pieces were used to make them, even if we can't see the complete picture directly.

Kull-back Divergence:

The Kullback-Leibler (KL) divergence is a measure of how two probability distributions are different from each other. Specifically, in the context of VAEs, it's used to quantify the difference between two important distributions:

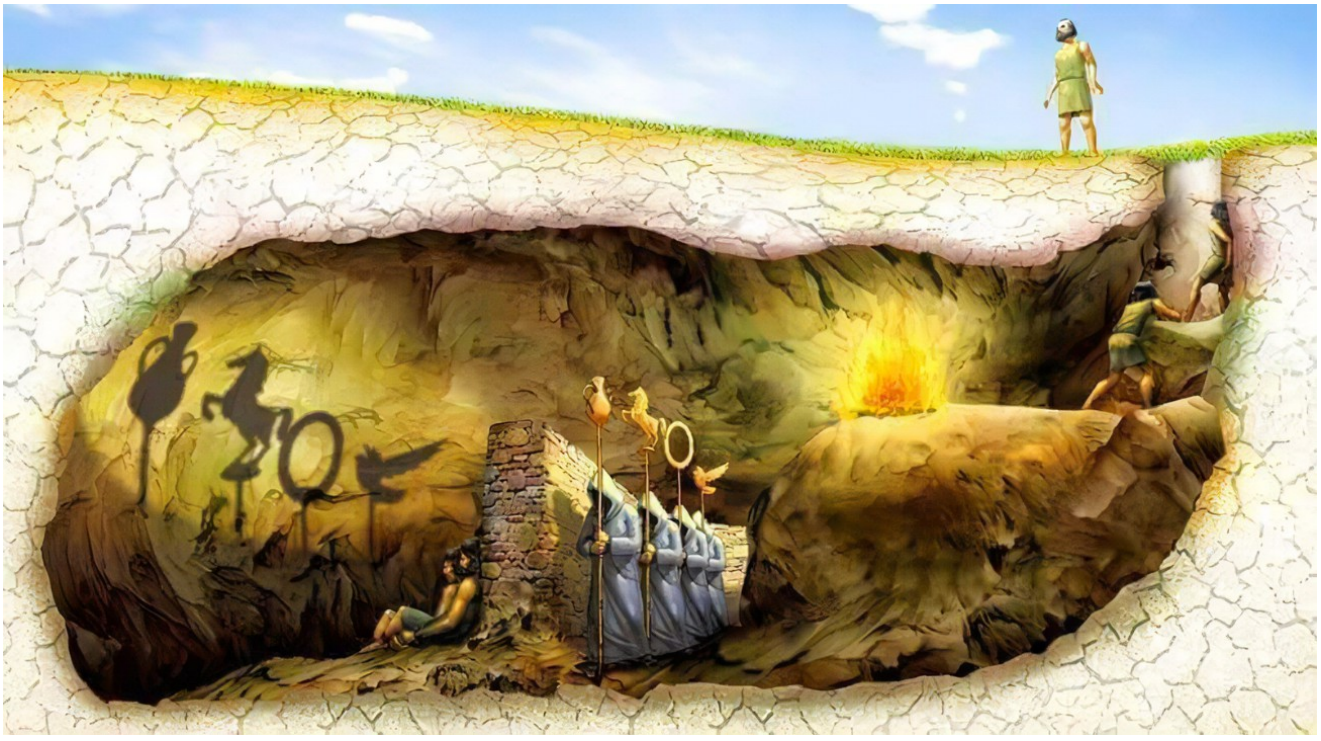
1. **Variational Distribution (q(z|x)):** This is the distribution of latent variables (z) given the input data (x). In simpler terms, it tells us how likely different values of the latent variables are, given a particular input data point.
2. **Prior Distribution (p(z)):** This is the distribution of latent variables we assume before seeing any specific data. It represents our initial beliefs or expectations about the latent space.

Now, the KL divergence between these two distributions $(KL[q(z|x)||p(z)])$ in VAEs serves an important purpose:

- It measures how much information is lost when we use the variational distribution $(q(z|x))$ to approximate the true distribution $(p(z))$.
- If the KL divergence is low, it means the variational distribution is very similar to the prior distribution, which is good because it means our model is learning meaningful representations of the data.
- If the KL divergence is high, it means the variational distribution is quite different from the prior distribution, which suggests that our model might not be capturing the underlying structure of the data very well.

In essence, minimizing the KL divergence encourages the variational distribution to stay close to the prior distribution, which helps in learning a good representation of the data in the latent space.

Plato's analogy for the Kull-back divergence:



Plato's allegory of the cave can provide a helpful analogy to understand the roles of the variational distribution and the prior distribution in a Variational Autoencoder (VAE).

1. **Variational Distribution (q(z|x)):** In the allegory of the cave, imagine the prisoners who are chained inside the cave and can only see the shadows on the wall. These shadows represent the observed data (x) in our analogy. Now, the variational distribution (q(z|x)) corresponds to the prisoners' interpretations or guesses about what objects in the real world might be

casting those shadows. It's their attempt to understand the true nature of the objects based solely on the shadows they see.

2. **Prior Distribution (p(z)):** In Plato's allegory, the prisoners have a certain worldview or belief system shaped by their experiences inside the cave. This worldview represents our prior beliefs about the latent space. It's like the prisoners' assumptions about what the real world outside the cave might be like, even though they haven't directly experienced it.

Now, let's connect this to the VAE:

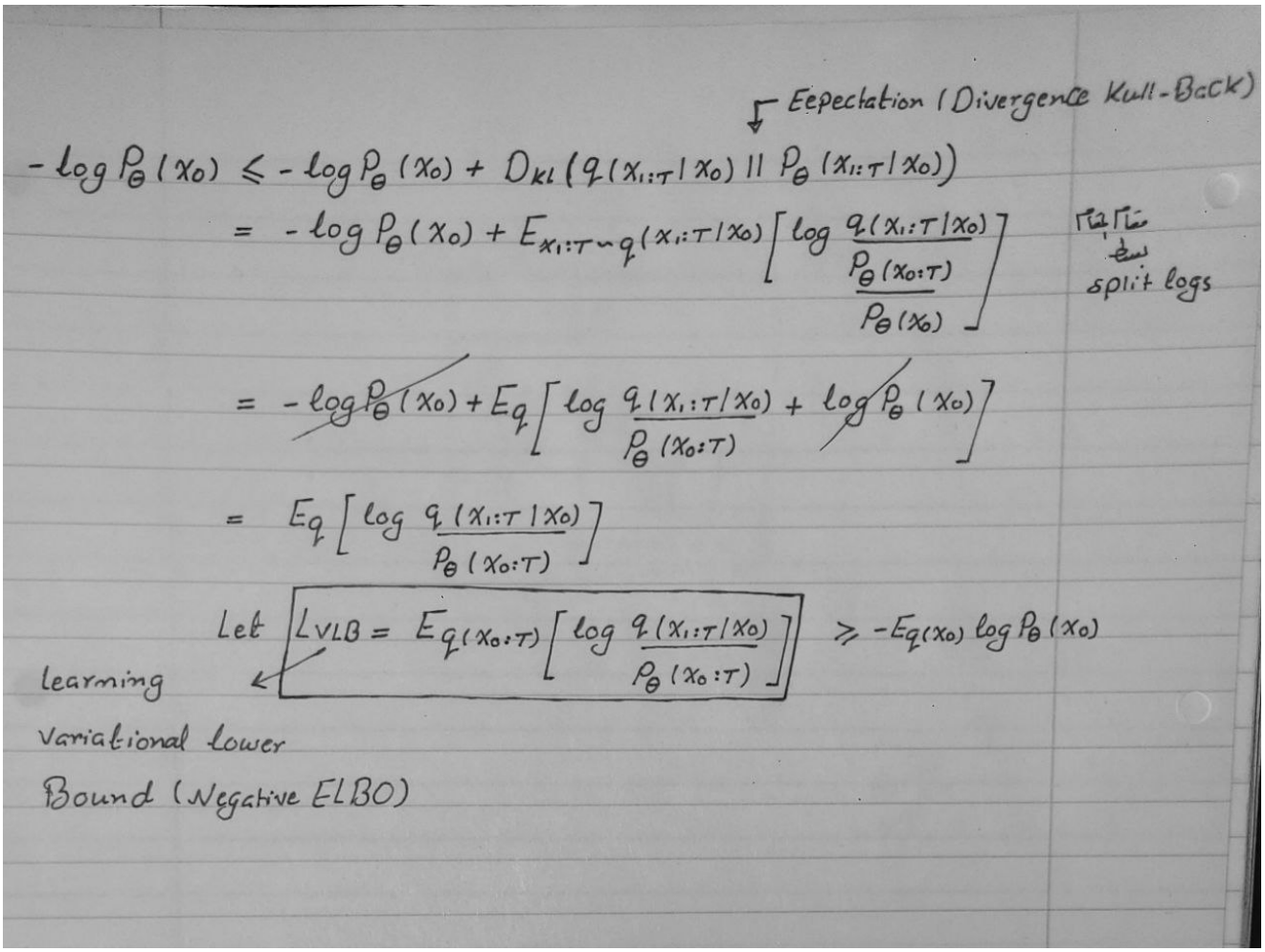
- The shadows on the cave wall represent the observed data, analogous to the input data (x) in the VAE.
- The prisoners' interpretations of these shadows represent the variational distribution (q(z|x)), which is the model's attempt to understand the latent structure of the data based on the observed inputs.
- The prisoners' worldview or belief system corresponds to the prior distribution (p(z)), which represents our initial assumptions about the latent space before observing any data.

In the context of the allegory, the VAE's goal is akin to the prisoners' desire to understand the true nature of the objects casting the shadows. Minimizing the KL divergence between the variational and prior distributions helps the model converge to a representation of the latent space that aligns with our prior beliefs while accurately explaining the observed data, much like how the prisoners' interpretations of the shadows should correspond with their worldview about the real world.

> 🔥 **Now we understand the terms what are they, we are going to do some derivations Mathematical explanation.**

Handwritten Derivations:

- **Derivate the Variational autoencoder (the Negative ELBO):**



- **Derivate the $L_T + L_t - 1 + L_0$:**

$$L_{VLB} = E_{q(x_{0:T})}\left[\log \frac{q(x_{1:T}|x_0)}{p_\theta(x_{0:T})}\right]$$

$q(x_{1:T}|x_0)$ → add noise

$p_\theta(x_{0:T})$ → Remove noise

$$= E_q\left[\log \frac{\prod_{t=1}^{T} q(x_t|x_{t-1})}{p_\theta(x_T)\prod_{t=1}^{T} p_\theta(x_{t-1}|x_t)}\right]$$

$$= E_q\left[-\log p_\theta(x_T) + \sum_{t=1}^{T}\log \frac{q(x_t|x_{t-1})}{p_\theta(x_{t-1}|x_t)}\right]$$

$$= E_q\left[-\log p_\theta(x_T) + \sum_{t=2}^{T}\log \frac{q(x_t|x_{t-1})}{p_\theta(x_{t-1}|x_t)} + \log \frac{q(x_1|x_0)}{p_\theta(x_0|x_1)}\right]$$ apply Bayes rule

$$= E_q\left[-\log p_\theta(x_T) + \sum_{t=2}^{T}\log \left(\frac{q(x_{t-1}|x_t,x_0)}{p_\theta(x_{t-1}|x_t)}\cdot\frac{q(x_t|x_0)}{q(x_{t-1}|x_0)}\right) + \log \frac{q(x_1|x_0)}{p_\theta(x_0|x_1)}\right]$$

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

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

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

$$= E_q\left[\log \frac{q(x_T|x_0)}{p_\theta(x_T)} + \sum_{t=2}^{T}\log \frac{q(x_{t-1}|x_t,x_0)}{p_\theta(x_{t-1}|x_t)} - \log p_\theta(x_0|x_1)\right]$$ D_{KL} حولنا لـ

$$= E_q\left[D_{KL}(q(x_T|x_0)\,\|\,p_\theta(x_T)) + \sum_{t=2}^{T} D_{KL}(q(x_{t-1}|x_t,x_0)\,\|\,p_\theta(x_{t-1}|x_t)) - \log p_\theta(x_0|x_1)\right]$$

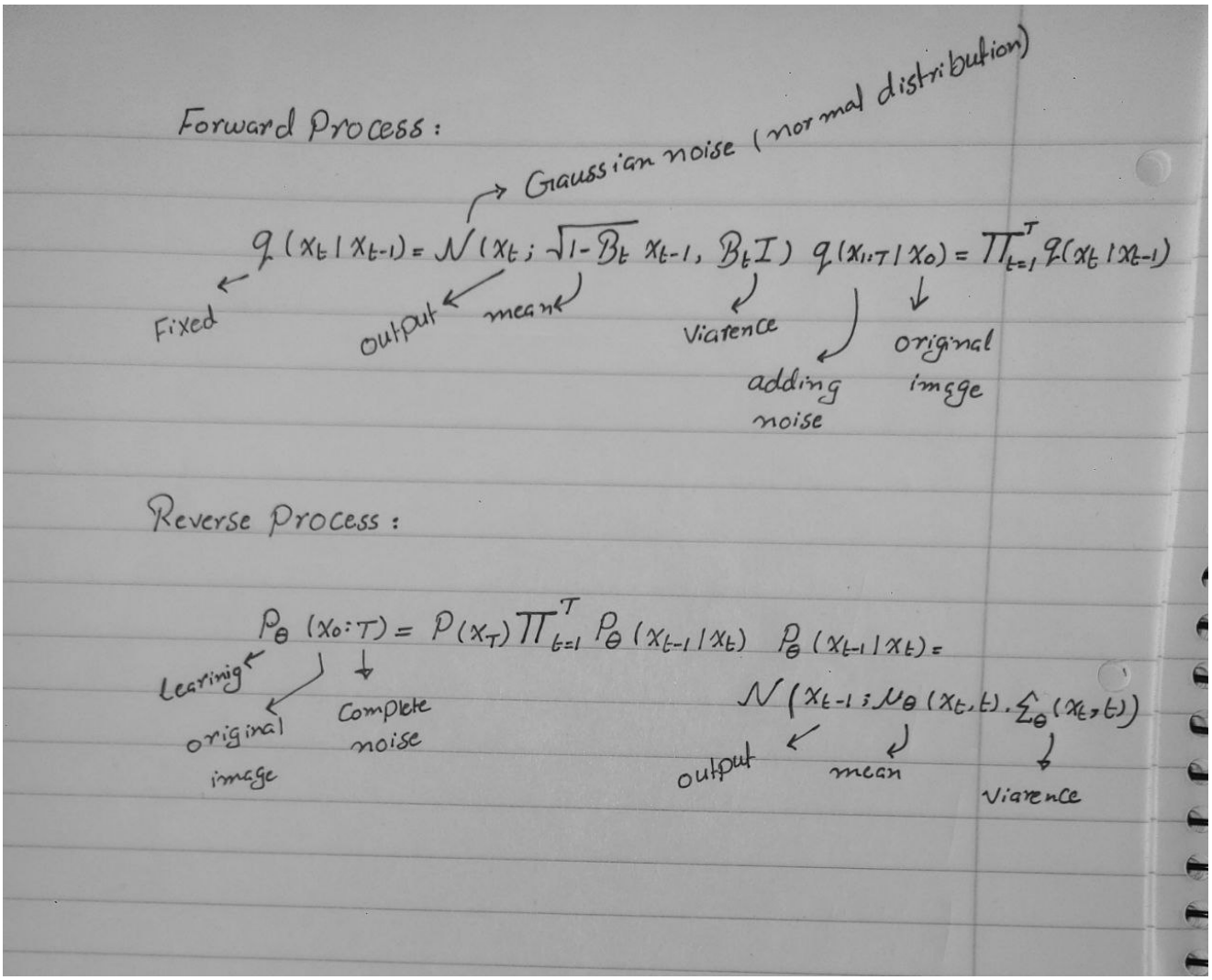
$$= E_q(L_T + L_{t-1} \dots L_0) = E_q\left(L_T + \sum_{t=2}^{T} L_{t-1} + L_0\right)$$

L_T ← learning with complete noise

$L_{t-1}, L_{t-2}, \dots$

$$\Rightarrow \boxed{\begin{aligned} &L_{VLB} = L_T + L_{T-1} + \dots + L_0 \\ &\text{where } L_T = D_{KL}(q(x_T|x_0)\,\|\,p_\theta(x_T)) \\ &L_{T-1} = \sum_{t=2}^{T} D_{KL}(q(x_{t-1}|x_t,x_0)\,\|\,p_\theta(x_{t-1}|x_t)) \\ &L_0 = -\log p_\theta(x_0|x_1) \end{aligned}}$$

- **Forward and Reverse Process:**



---

let's take a deeper dive at the hyper-parameters of the stable diffusion.

1. `prompt` - this is the textual prompt we pass through to generate an image. Similar to the `pipe(prompt)` function we saw in part 1
2. `g` or `guidance scale` - It's a value that determines how close the image should be to the textual prompt. This is related to a technique called Classifier free guidance which improves the quality of the images generated. The higher the value of the guidance scale, more close it will be to the textual prompt
3. `seed` - This sets the seed from which the initial Gaussian noisy latents are generated
4. `steps` - Number of de-noising steps taken for generating the final latents.
5. `dim` - dimension of the image, for simplicity we are currently generating square images, so only one value is needed
6. `save_int` - This is optional, a boolean flag, if we want to save intermediate latent images, helps in visualization.

*Resources:

Research Papers:

[2006.11239] Denoising Diffusion Probabilistic Models (arxiv.org)
[2112.10752] High-Resolution Image Synthesis with Latent Diffusion Models (arxiv.org)

Websites:

lilianweng_diffusion-models
assemblyai
param Hanji
HuggingFace-DMs
HuggingFace_2-DMs
HuggingFace-github_blog

Youtube:

Outlier
Umar Jamil
George Hotz_part1
George Hotz_part3
ComputerPhile
Machine Learning at Berkeley

Citations:

```
@misc{ho2020denoising,
    title   = {Denoising Diffusion Probabilistic Models},
    author  = {Jonathan Ho and Ajay Jain and Pieter Abbeel},
    year    = {2020},
    eprint  = {2006.11239},
    archivePrefix = {arXiv},
    primaryClass = {cs.LG}
}
```

```bibtex
@misc{ho2020denoising,
    title   = {Denoising Diffusion Probabilistic Models},
    author  = {Jonathan Ho and Ajay Jain and Pieter Abbeel},
    year    = {2020},
    eprint  = {2006.11239},
    archivePrefix = {arXiv},
    primaryClass = {cs.LG}
}
```