Before We Start

- This talk is relatively math heavy, however
  - It won't affect your ability to understand high level mechanism
  - It won't stop you from experimenting

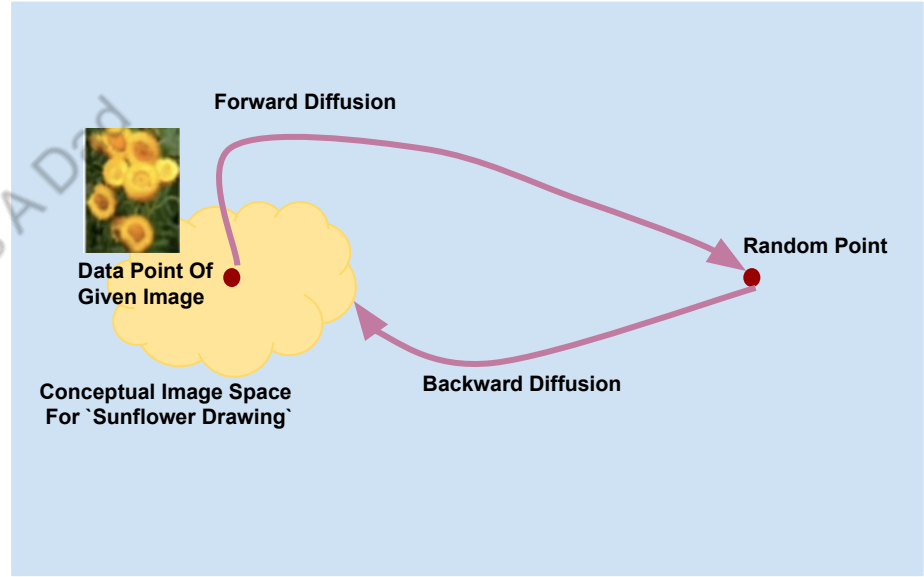
- I will cover topics including:
  - DDPM (Denoising Diffusion Probabilistic Models)
  - DDIM (Denoising Diffusion Implicit Models)
  - U-Net
  - LDM (Latent Diffusion Model or Stable Diffusion)

- I will **not** cover some topics in details, if those are covered in previous videos (AutoEncoder Deep Dive, Generative Image Overview etc)
  - CNN basics (convolution, transposed convolution etc)
  - ELBO basics
  - KL Divergence basics
  - Reparameterization basics

Just try to enjoy and have fun :D

Diffusion High Level

- Say we have 256X256 pixels to represent an image, each pixel has 3 color channel (RGB). We now have a space of 256X256X3 to represent images (blue box)
- Each element needs to be specific values to form a meaningful image, say `sunflower drawing`. Say all the possible combinations of `sunflower drawing` forms a space/distribution (yellow cloud).
- We will be able to generate images of `sunflower drawing`, if we can train a model with the capability to **traverse back to the yellow cloud from any random point in the blue box**.
- We will be able to generate more images, if we extend beyond one yellow cloud representing `sunflower drawing`, to more space/distribution representing more images.



Conceptual Image Space For 256X256X3 Pixels

Inspired By Diffusion In Physics

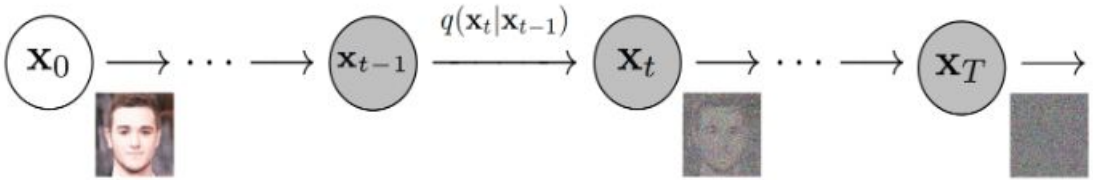
- In physics, diffusion describes the net movement of anything (e.g., atoms, molecules, energy) from a region of higher concentration to a region of lower concentration. This movement is driven by a concentration gradient and the random motion of individual particles (often referred to as Brownian motion).

- **Key Inspirations from Physics:**
  - **Non-equilibrium Thermodynamics:** The process of gradually adding noise and then reversing it is analogous to a system moving away from and then back towards an ordered state.
  - **Stochastic Processes:** Diffusion is inherently a stochastic (random) process. Diffusion models leverage the mathematics of stochastic processes to define how image is transformed into noise and how it's generated back from noise.

Diffusion

- A drop of ink spreading out in a glass of water. Initially, the ink is concentrated, but over time, the ink molecules randomly move and collide, gradually mixing with the water until the color is uniform.
- Image from Wiki, By BruceBlaus

Forward Diffusion Process



- Image is transformed into pure noise (standard normal distribution from larger image space) in ***T*** steps
  - This is done via a Markov chain with T steps, where the image at timestep t maps to its subsequent state at timestep t+1. Each step depends only on the previous one.

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

  - Gaussian noise added at each step in the Markov chain is not constant or arbitrary. Instead, the noise is derived from the structure of the original image and the rate at which it's added steadily increases with each consecutive step, all determined by a **scheduler**.
  - $\{\beta_t \in (0,1)\}_{t=1}^{T}$ is the diffusion rate precalculated by the scheduler, ***I*** is an identity matrix.

- ***q(x_t\|x_{t-1})*** is standard normal distribution, but how do we add noise at each step, since it's random? Can we just randomly pick everytime?
  - Yes we can. But it means whenever we need a sample at X_t , we need to iterate through timestep {0, …, X_{t-1}} in the Markov chain.
  - Or we can use the ***reparameterization trick*** (covered in AutoEncoder deep dive), along with the ***addition property of Gaussian distribution***

Forward Diffusion Process

$$\alpha_t = 1 - \beta_t \qquad \bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$$

$$\mathbf{x}_t = \sqrt{\alpha_t}\mathbf{x}_{t-1} + \sqrt{1-\alpha_t}\boldsymbol{\epsilon}_{t-1} \qquad \text{;where } \boldsymbol{\epsilon}_{t-1}, \boldsymbol{\epsilon}_{t-2}, \cdots \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

$$= \sqrt{\alpha_t\alpha_{t-1}}\mathbf{x}_{t-2} + \sqrt{1-\alpha_t\alpha_{t-1}}\bar{\boldsymbol{\epsilon}}_{t-2} \qquad \text{;where } \bar{\boldsymbol{\epsilon}}_{t-2} \text{ merges two Gaussians (*).}$$

$$= \dots$$

$$= \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}$$

Reparameterization trick

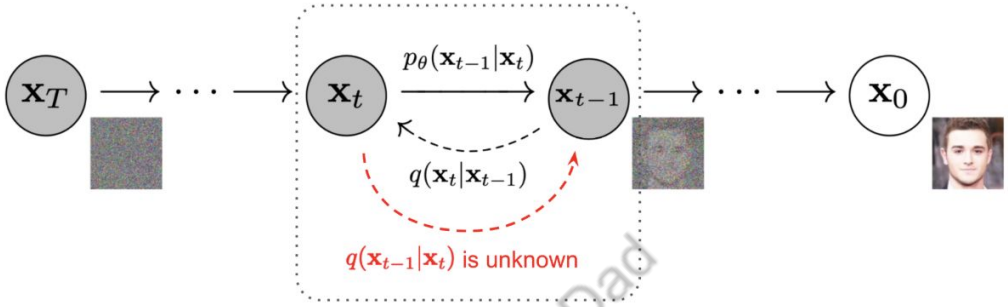
Addition property of Gaussian

- Based on above, we have $q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I})$
  - Now whenever we need a sample at $\boldsymbol{X_t}$, we can directly go from the original image $\boldsymbol{X_0}$ since all scheduling parameters are precalculated

- Scheduler Variances
  - The original DDPM defines a linear schedule ranging $\boldsymbol{\beta}$ from 0.0001 at timestep 0 to 0.02 at timestep T.
  - There are other variances like cosine scheduler

$$\beta_t = \text{clip}(1 - \frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}}, 0.999) \qquad \bar{\alpha}_t = \frac{f(t)}{f(0)} \qquad \text{where } f(t) = \cos\left(\frac{t/T + s}{1+s} \cdot \frac{\pi}{2}\right)^2$$

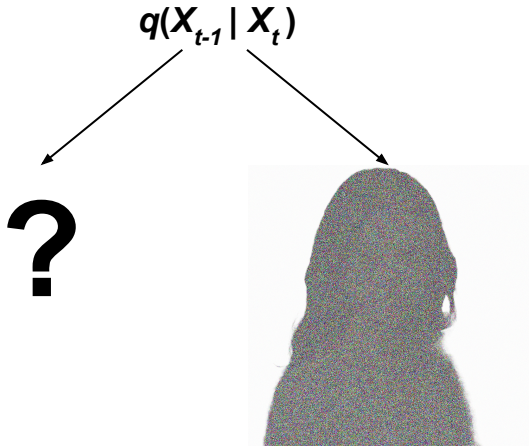
Reverse Diffusion Process

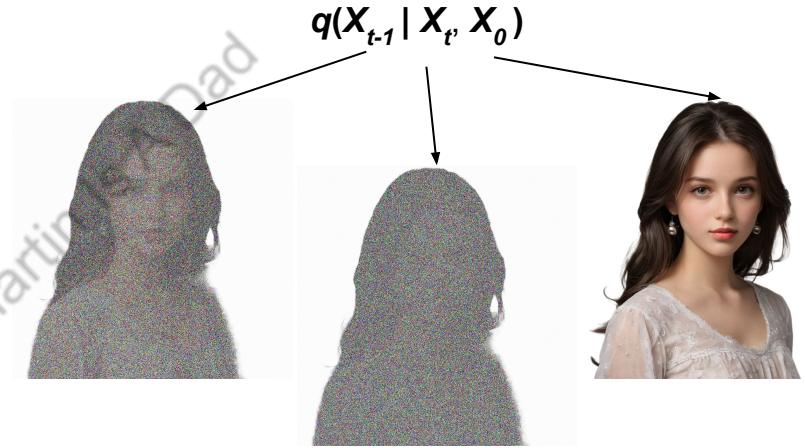


- This is where the machine learning happens, the model is trained to reverse the forward process
- Directly transforming random noise into a new image is extremely difficult and complex, but transforming a noisy image into a slightly less noisy image is relatively easy. Thus we start pure noise $X_T \sim N(0, I)$ and gradually denoise it iteratively to produce a realistic image sample X_0
- **Challenge:** the true reverse transition $q(X_{t-1} \mid X_t)$ is intractable to compute since
  - From Bayes' theorem $q(X_{t-1} \mid X_t) = q(X_t \mid X_{t-1}) * q(X_{t-1}) / q(X_t)$
  - Both $q(X_{t-1})$ and $q(X_t)$ requires knowledge of the entire distribution of image $q(X_0)$
    - $q(X_t) = \int q(X_t \mid X_0) q(X_0) dX_0$
- **Solution**: Approximate the true reverse transition with a learned model $p_\theta(X_{t-1} \mid X_t)$, typically also a Gaussian: $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t))$

Reverse Diffusion Process

- Notice that although $q(X_{t-1} \mid X_t)$ is intractable, the posterior conditioned on X_0 , $q(X_{t-1} \mid X_t, X_0)$ is tractable, intuition is the following:





- $q(X_{t-1} \mid X_t, X_0)$ is also Gaussian: $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_{t-1}; \tilde{\boldsymbol{\mu}}(\mathbf{x}_t, \mathbf{x}_0), \tilde{\beta}_t \mathbf{I})$

Reverse Diffusion Process

$$
\begin{aligned}
q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) &= q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0)\frac{q(\mathbf{x}_{t-1}|\mathbf{x}_0)}{q(\mathbf{x}_t|\mathbf{x}_0)} \\
&\propto \exp\Big(-\frac{1}{2}\big(\frac{(\mathbf{x}_t - \sqrt{\alpha_t}\mathbf{x}_{t-1})^2}{\beta_t} + \frac{(\mathbf{x}_{t-1} - \sqrt{\bar{\alpha}_{t-1}}\mathbf{x}_0)^2}{1-\bar{\alpha}_{t-1}} - \frac{(\mathbf{x}_t - \sqrt{\bar{\alpha}_t}\mathbf{x}_0)^2}{1-\bar{\alpha}_t}\big)\Big) \\
&= \exp\Big(-\frac{1}{2}\big(\frac{\mathbf{x}_t^2 - 2\sqrt{\alpha_t}\mathbf{x}_t\mathbf{x}_{t-1} + \alpha_t\mathbf{x}_{t-1}^2}{\beta_t} + \frac{\mathbf{x}_{t-1}^2 - 2\sqrt{\bar{\alpha}_{t-1}}\mathbf{x}_0\mathbf{x}_{t-1} + \bar{\alpha}_{t-1}\mathbf{x}_0^2}{1-\bar{\alpha}_{t-1}} - \frac{(\mathbf{x}_t - \sqrt{\bar{\alpha}_t}\mathbf{x}_0)^2}{1-\bar{\alpha}_t}\big)\Big) \\
&= \exp\Big(-\frac{1}{2}\big((\frac{\alpha_t}{\beta_t} + \frac{1}{1-\bar{\alpha}_{t-1}})\mathbf{x}_{t-1}^2 - (\frac{2\sqrt{\alpha_t}}{\beta_t}\mathbf{x}_t + \frac{2\sqrt{\bar{\alpha}_{t-1}}}{1-\bar{\alpha}_{t-1}}\mathbf{x}_0)\mathbf{x}_{t-1} + C(\mathbf{x}_t, \mathbf{x}_0)\big)\Big)
\end{aligned}
$$

$$
\begin{aligned}
\tilde{\beta}_t &= 1/(\frac{\alpha_t}{\beta_t} + \frac{1}{1-\bar{\alpha}_{t-1}}) = 1/(\frac{\alpha_t - \bar{\alpha}_t + \beta_t}{\beta_t(1-\bar{\alpha}_{t-1})}) = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\cdot\beta_t \\
\tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0) &= (\frac{\sqrt{\alpha_t}}{\beta_t}\mathbf{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}}{1-\bar{\alpha}_{t-1}}\mathbf{x}_0)/(\frac{\alpha_t}{\beta_t} + \frac{1}{1-\bar{\alpha}_{t-1}}) \\
&= (\frac{\sqrt{\alpha_t}}{\beta_t}\mathbf{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}}{1-\bar{\alpha}_{t-1}}\mathbf{x}_0)\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\cdot\beta_t \\
&= \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\mathbf{x}_0
\end{aligned}
$$

$$
q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_{t-1}; \tilde{\boldsymbol{\mu}}(\mathbf{x}_t, \mathbf{x}_0), \tilde{\beta}_t\mathbf{I})
$$

Reverse Diffusion Process

- Given $\mathbf{x}_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}(\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_t)$ from the forward diffusion process, we can have

$$\begin{aligned}\tilde{\boldsymbol{\mu}}_t &= \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\frac{1}{\sqrt{\bar{\alpha}_t}}(\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_t) \\ &= \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_t\right)\end{aligned}$$

- Now we have several options on model's learning objective:
  - Model to learn to predict the entire $\tilde{\boldsymbol{\mu}}_t$
  - (Picked) Model to learn to predict the noise at each timestep $\boldsymbol{\epsilon}_t$

- For reasons I'll explain later, the model's learning objective is to predict the noise at each timestep. Then we have the model estimated expectation:

$$\boldsymbol{\mu}_\theta(\mathbf{x}_t, t) = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right)$$

- Then we have

What about this?

$$\mathbf{x}_{t-1} = \mathcal{N}(\mathbf{x}_{t-1}; \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boxed{\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)}\right), \boxed{\boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t)})$$

Learned by Model

Reverse Diffusion Process

- The DDPM paper found that fixing the variance $\Sigma_\theta(x_t, t)$ to be $\beta_t \mathbf{I}$ or $\tilde{\beta}_t \mathbf{I}$ works well in practice. Using $\tilde{\beta}_t \mathbf{I}$ is theoretically motivated as it corresponds to true posterior variance $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_{t-1}; \tilde{\boldsymbol{\mu}}(\mathbf{x}_t, \mathbf{x}_0), \tilde{\beta}_t \mathbf{I})$

- Given the fixed variance that simplifies model's learning, now we have

$$\mathbf{x}_{t-1} = \mathcal{N}(\mathbf{x}_{t-1}; \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right), \tilde{\beta}_t \mathbf{I} \qquad)$$

- Using the parameterized mean $\mu_\theta(x_t, t)$ and a fixed variance (e.g., $\Sigma_\theta(x_t, t) = \sigma_t^2 \mathbf{I}$, where $\sigma_t^2 = \beta_t$ or $\tilde{\beta}_t$), a single step of the reverse process involves:

  1. Predict the noise $\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)$ using the trained neural network.
  2. Calculate the mean: $\boldsymbol{\mu}_\theta(\mathbf{x}_t, t) = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right)$
  3. Sample $\boldsymbol{X}_{t-1}$ $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t \mathbf{z}$,where $z \sim \mathcal{N}(0, \mathbf{I})$

Loss Function

- The training objective of diffusion-based generative models is equivalent to maximizing the likelihood (or log likelihood) of the sample generated (at the end of the reverse process) belonging to the original data distribution.

- To train our neural network, we define the loss function as the objective function's negative. So we can minimize the loss function to maximize the log likelihood.

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

$$L = -log(p_\theta(x_0))$$

- This is intractable because we need to integrate over a very high dimensional image space over T timesteps. **Instead we can do ELBO (Evidence Lower Bound, or VLB, Variational Lower Bound). This is similar with Variational AutoEncoder.**

Loss Function

KL Divergence is also gone through in AutoEncoder Deep Dive

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

Loss Function

$$
\begin{aligned}
L_{\text{VLB}} &= \mathbb{E}_{q(\mathbf{x}_{0:T})}\left[\log \frac{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}{p_\theta(\mathbf{x}_{0:T})}\right] \\
&= \mathbb{E}_q\left[\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)}\right] \\
&= \mathbb{E}_q\left[-\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)}\right] \\
&= \mathbb{E}_q\left[-\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)}\right] \\
&= \mathbb{E}_q\left[-\log p_\theta(\mathbf{x}_T) + \sum_{t=2}^{T}\log \left(\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)}\right) + \log\frac{q(\mathbf{x}_1|\mathbf{x}_0)}{p_\theta(\mathbf{x}_0|\mathbf{x}_1)}\right] \\
&= \mathbb{E}_q\left[-\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)}\right] \\
&= \mathbb{E}_q\left[-\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)}\right] \\
&= \mathbb{E}_q\left[\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)\right] \\
&= \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}
$$

Loss Function

- As shown above, L_{vlb} is broken into 3 parts:
  - L_0: this value is minor and the original authors got better result without this
  - L_T: this is the KL divergence between the distribution of the **final step in the forward process** $q(X_T | X_0)$ and the **first step in the reverse process** $p_\theta(X_T)$. Both of that requires no model for learning, we can ignore it from loss function since it's constant

- L_{VLB} (simplified) = $E_q[\sum_{t=2}^{T} \underbrace{D_{\text{KL}}(q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) \parallel p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t))}_{L_{t-1}}]$

- Given $$L_t = \mathbb{E}_{\mathbf{x}_0,\boldsymbol{\epsilon}}\left[\frac{1}{2\|\boldsymbol{\Sigma}_\theta(\mathbf{x}_t,t)\|_2^2}\|\tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t,\mathbf{x}_0) - \boldsymbol{\mu}_\theta(\mathbf{x}_t,t)\|^2\right]$$
$$= \mathbb{E}_{\mathbf{x}_0,\boldsymbol{\epsilon}}\left[\frac{1}{2\|\boldsymbol{\Sigma}_\theta\|_2^2}\|\frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_t\right) - \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t,t)\right)\|^2\right]$$
$$= \mathbb{E}_{\mathbf{x}_0,\boldsymbol{\epsilon}}\left[\frac{(1-\alpha_t)^2}{2\alpha_t(1-\bar{\alpha}_t)\|\boldsymbol{\Sigma}_\theta\|_2^2}\|\boldsymbol{\epsilon}_t - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t,t)\|^2\right]$$
Constant

- We have the simplified loss function as $L_t^{\text{simple}} = \mathbb{E}_{t\sim[1,T],\mathbf{x}_0,\boldsymbol{\epsilon}_t}\left[\|\boldsymbol{\epsilon}_t - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t,t)\|^2\right]$
  - $\boldsymbol{\epsilon}_t$ is the noise generated in forward diffusion process
  - $\boldsymbol{\epsilon}_\theta(x_t, t)$ is the noise predicted in reverse diffusion process
  - This is just a Mean Squared Error (MSE) between $\boldsymbol{\epsilon}_t$ and $\boldsymbol{\epsilon}_\theta(x_t, t)$

Summary

- Training
  - Forward diffusion $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}$
  - Loss function $L_t^{\text{simple}} = \mathbb{E}_{t\sim[1,T],\mathbf{x}_0,\boldsymbol{\epsilon}_t}\left[\|\boldsymbol{\epsilon}_t - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\|^2\right]$
- Sampling $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t\mathbf{z}$

Why Is Scheduler Needed

Can’t we just randomly add noise and denoise an image?

- Necessary for efficient training
  - Forward process Markov chain $q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I})$ $q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1})$ depends on β_t being a fixed known constant for timestep ***t***
  - This is crucial so we can directly sample a noisy image at any timestep ***t*** based on $q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I})$
- Necessary for reverse process with the simplified loss function
  - The tractable conditional reverse step $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_{t-1}; \tilde{\boldsymbol{\mu}}(\mathbf{x}_t, \mathbf{x}_0), \tilde{\beta}_t\mathbf{I})$ depends on β_t being a fixed known constant for timestep ***t***
  - Thus the simplified loss function $L_t^{\text{simple}} = \mathbb{E}_{t\sim[1,T],\mathbf{x}_0,\boldsymbol{\epsilon}_t}\left[\|\boldsymbol{\epsilon}_t - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\|^2\right]$ also depends on scheduler
- A carefully designed scheduler can affect model performance (linear, cosine etc).

Why Predict Noise Instead Of Whole Image

The goal is to learn the reverse process $p_\theta(X_{t-1} \mid X_t)$, so it's matching the true but intractable $q(X_{t-1} \mid X_t)$. Why can't we predict the mean of the reverse step $\mu_\theta(\mathbf{x}_t, t)$ instead of noise $\epsilon_\theta(\mathbf{x}_t, t)$?

- Simplified loss function
  - By designing the model to predict noise instead of the mean, the loss function can be simplified to a MSE form $L_t^{\text{simple}} = \mathbb{E}_{t\sim[1,T],\mathbf{x}_0,\boldsymbol{\epsilon}_t}\left[\|\boldsymbol{\epsilon}_t - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\|^2\right]$
  - This loss is much easier to implement and optimize, and is mathematically equivalent

- Better scaling
  - The noise ϵ added at each step is sampled from a standard normal distribution. This means the target model is trying to predict always has a consistent scale and statistical properties, regardless of the time step t or the specific image X_0.
  - If we choose to predict the image itself, If t is large, X_t is almost pure noise. Asking the model to directly regress from near-pure noise to a highly structured, clean image with potentially large variations in pixel values might be much more difficult. The scale and complexity of the image X_0 can vary greatly, while the noise target ϵ is more uniform.

- Empirical success: probably due to above reasons, predicting noise ϵ just works :)

Diffusion Training Process

For each batch of the training process:

- Sample a random timestep ***t*** for each image sample within the batch
- Add noise from Gaussian distribution, by using the closed form formula $q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I})$ instead of iteratively from timestep 0 to ***t***) according to the timestep ***t***
- Convert timestep ***t*** into time embeddings, preparing to feed into U-Net
- Feed the time embedding, along with the noisy image, to U-Net to predict the noise
- Compare the predicted noise to the actual nose to calculate loss function
- Update the parameters of Diffusion model via backpropagation and gradient descent

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

```
def diffuse_image(x, t):
    sqrt_alpha_hat = torch.sqrt(alpha_hat[t])
    sqrt_one_minus_alpha_hat = torch.sqrt(1 - alpha_hat[t])
    Ɛ = torch.randn_like(x)
    return sqrt_alpha_hat * x + sqrt_one_minus_alpha_hat *   Ɛ, Ɛ

def train():
    model = UNet().to(device)
    optimizer = optim.AdamW(model.parameters(), lr=args.lr)
    mse = nn.MSELoss()

    for epoch in range(epochs):
        for images in dataloader:
            t = sample_timesteps(0, MAX_STEPS)
            x_t, noise = diffuse_image(images, t)
            predicted_noise = model(x_t, t)
            loss = mse(noise, predicted_noise)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

Diffusion Training Process

Input to Diffusion Model

Calculate Loss Function &
Update Model Parameters

Original Image From Training Dataset

...

Time Embedding

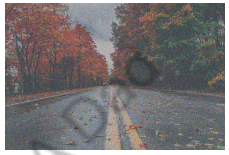
t=121

t=11

...

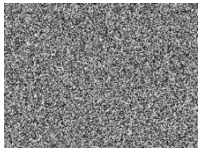
t=65

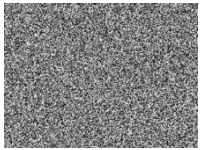
Noisy Image



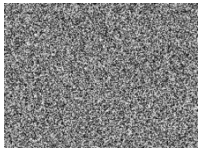
...

Predicted Noise

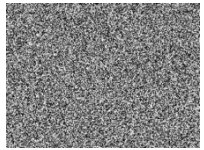


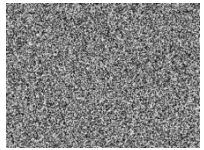


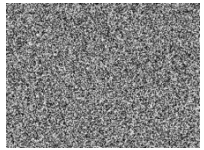
...



Actual Noise







Diffusion Sampling Process

For Sampling new image with the model:

- Sample random noise from Gaussian distribution, and define how many timesteps to generate the new image, say 1000
- For each timestep, the Diffusion model predicts the whole noise presented in current image
- We only remove a fraction of the whole noise, as defined by the scheduler
  - To get x_{t-1} from x_t , the formula is $x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t)\right) + \sqrt{\tilde{\beta}_t}z$
  - Thus the removed fraction is $\frac{\beta_t}{\sqrt{\alpha_t(1-\bar{\alpha}_t)}}$
- Repeat for T (in this example 1000) times to generate the new image

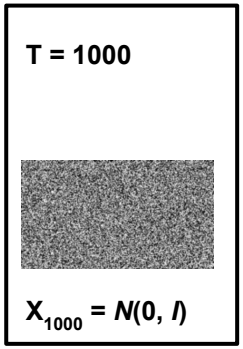
Algorithm 2 Sampling

1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = T, \ldots, 1$ **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** $\mathbf{x}_0$

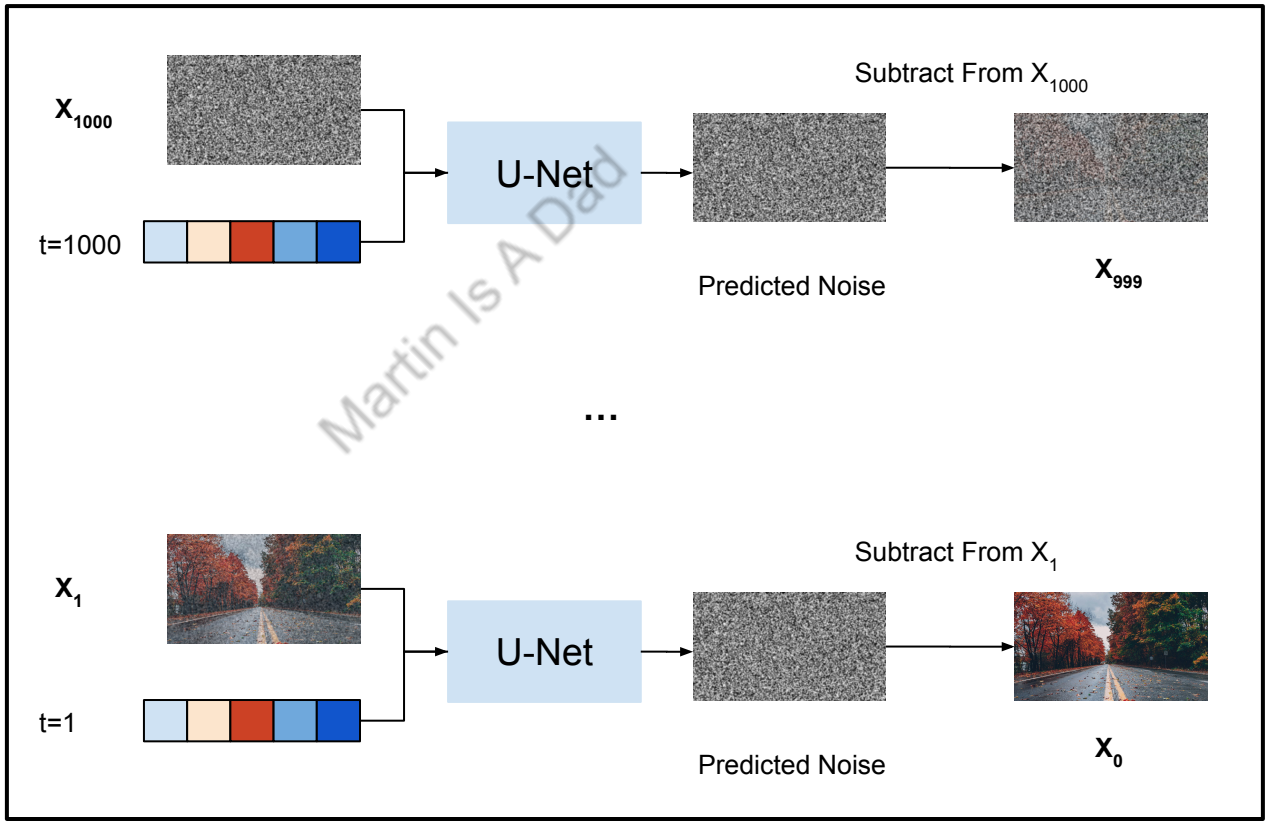
```
def sample(model):
    with torch.no_grad():
        x = torch.randn((3, 256, 256))
        for i in range(noise_steps):
            predicted_noise = model(x, t)
            if i > 1:
                noise = torch.randn_like(x)
            else:
                noise = torch.zeros_like(x)
            x = 1 / torch.sqrt(alpha) * (x - ((1 - alpha) / (torch.sqrt(1 -
alpha_hat))) * predicted_noise) + torch.sqrt(beta) * noise
    return x
```

Diffusion Sampling Process

Sample Random Gaussian Noise



Iteratively Remove Noise T times



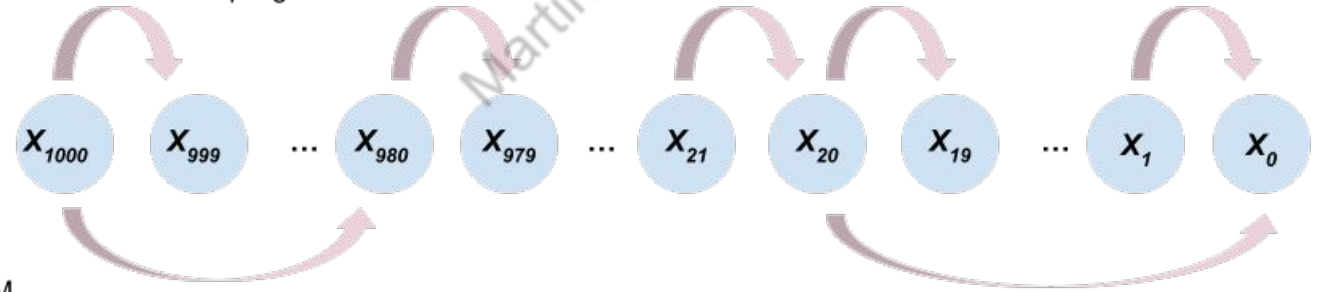
DDPM *vs* DDIM

- The original version of Diffusion is called DDPM (**Denoising Diffusion Probabilistic Models**). The sampling is slow since:
  - Each timestep is a Stochastic Markovian process (current state only depend on previous state)
  - Need to iteratively goes through many timesteps to generate a new image.

- DDIM (**Denoising Diffusion Implicit Model**) aims to solve this problem
  - The training process and objective is identical with DDPM, only difference is sampling process
  - DDIM accelerates the sampling process since it allows much smaller sampling steps

DDPM

- T = 1000 for sampling



DDIM

- Selected S = 50 timesteps for sampling
- Uniform partition strategy (most common): $S = \{\tau_{50}, \tau_{49}, \dots, \tau_1\} = \{X_{1000}, X_{980}, \dots, X_{20}, X_0\}$
- Non-uniform partition strategy:
  - Spend more steps in later stage of sampling
  - Use model to learn the optimal partition

DDIM Details

- **Observation**: $q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I})$ is still a tractable Gaussian, regardless of whether the forward process is Markovian or not, as long as the forward process can be expressed as adding noise scaled by $\sqrt{1-\bar{\alpha}_t}$. This distribution is what the DDPM training objective (specifically, the simplified noise prediction loss) primarily relies on.

- DDIM considers a more general non-Markovian forward process. The goal is still to define a reverse process that can generate $\boldsymbol{X_0}$ from $\boldsymbol{X_T}$. Unlike DDPM which derives this based on the fixed forward Markov chain, DDIM directly proposes a implicit function $\boldsymbol{X_{t-1}} = f(\boldsymbol{x_t}, \boldsymbol{t})$.

- **Math**:
  - From $q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I})$ we can get $\mathbf{x}_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}(\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_t)$
  - DDPM's trained model predicts the noise $\boldsymbol{\epsilon}_\theta(x_t, t)$. Using this we can estimate $\boldsymbol{X_0}$:

$$\hat{x}_0 = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\epsilon_\theta(x_t, t)}{\sqrt{\bar{\alpha}_t}}$$

  - DDPM proposed a new general form based on the above estimation and a new variance parameter $\boldsymbol{\sigma_t}$:

$$x_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\hat{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\epsilon_t + \sigma_t\mathbf{z}, \text{ where } \mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$$

DDIM Details

- **Math**:
  - Substitute the expression for $\hat{x}_0$

$$\boldsymbol{x}_{t-1} = \sqrt{\alpha_{t-1}} \underbrace{\left(\frac{\boldsymbol{x}_t - \sqrt{1-\alpha_t}\epsilon_\theta^{(t)}(\boldsymbol{x}_t)}{\sqrt{\alpha_t}} \right)}_{\text{“ predicted } \boldsymbol{x}_0\text{”}} + \underbrace{\sqrt{1-\alpha_{t-1}-\sigma_t^2} \cdot \epsilon_\theta^{(t)}(\boldsymbol{x}_t)}_{\text{“direction pointing to } \boldsymbol{x}_t\text{”}} + \underbrace{\sigma_t \epsilon_t}_{\text{random noise}}$$

- **Advantages**:
  - **Flexible sampling**: The parameter $\boldsymbol{\sigma_t}$ introduces flexibility in the reverse process. When $\boldsymbol{\sigma_t}$=0, the reverse process becomes deterministic. This means that for a given initial noise $\boldsymbol{X_T}$, the generated sample $\boldsymbol{X_0}$ will always be the same.
  - **Accelerated sampling**: Since the dependency on Markov chain is removed, DDIM allows for sampling with much less steps $\boldsymbol{S} << \boldsymbol{T}$ by selecting a subsequence of timesteps $\tau=\{\tau_1,\ldots,\tau_S\}$. The reverse process then only needs to be computed for these selected timesteps.

- Relationship with DDPM
  - DDPM is a subset of DDIM. By setting $\sigma_t^2 = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$ DDIM reverse process becomes equivalent to the DDPM reverse process

DDIM Summary

Feature	**DDPM**	**DDIM**
Forward Process Type	Markovian	Non-Markovian
Forward Step Dependency	$q(x_t \mid x_{t-1})$	$q(x_{t-1} \mid x_t, x_0)$
Reverse Process Type	Stochastic Approximation to $q(x_{t-1} \mid x_t)$	Implicit Function $X_{t-1} = f(x_t, t)$
Sampling Stochasticity	Yes, Gaussian noise added	No when $\sigma_t = 0$
Sampling Speed (Steps)	Slow (Typically T≈1000)	Fast (Typically S≪T, e.g., 20-100)
Training Objective	L_{simple}	L_{simple} (Identical with DDPM)

U-Net Architecture

- Diffusion uses U-Net like architecture to predict noise of the image (output size = input size)
- Original U-Net gets its name from the U shape in the model diagram, U-Net shares architecture similarity with Autoencoders, including downsampling, upsampling, bottlenecks etc

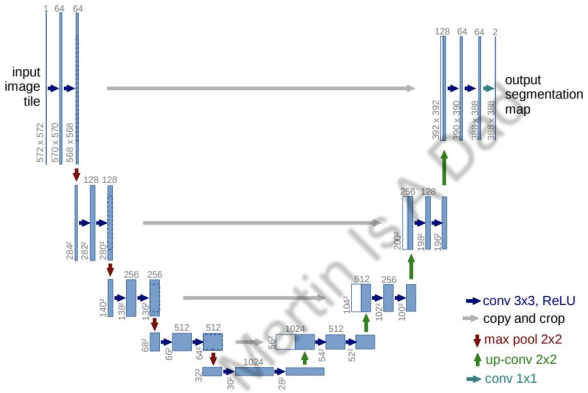
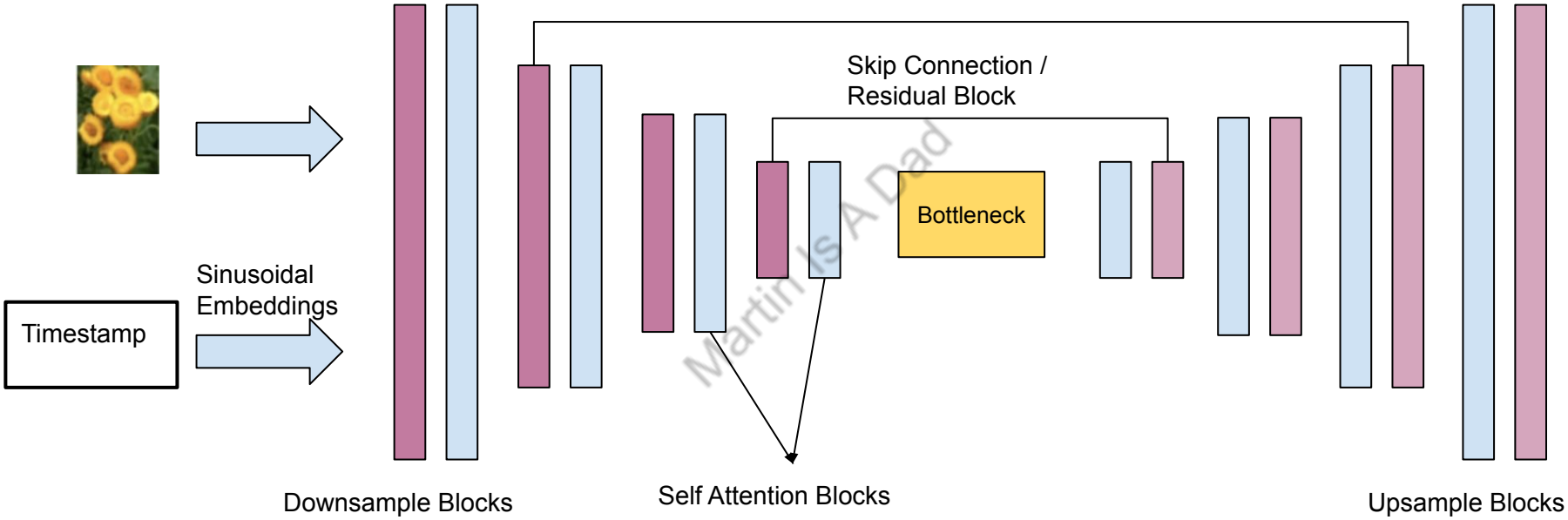


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

- Diffusion model's implementation contains modifications to original U-Net, including residual blocks, multi-head attention and time-step embeddings etc.
  - Residual/skip connection: helps preserve spatial details when down scaling features
  - Timestep embeddings: make diffusion model aware of the different timestep and level of noises
  - Multihead Attention: better capture relationship of original images' feature to predict noise

Diffusion Model Architecture



Diffusion Model Architecture

U-Net Main Function (full version in https://nn.labml.ai/diffusion/ddpm/unet.html):

- `x` has shape `[batch_size, in_channels, height, width]`
- `t` has shape `[batch_size]`

```
383  def forward(self, x: torch.Tensor, t: torch.Tensor):
```

Get time-step embeddings

```
390      t = self.time_emb(t)
```

Get image projection

```
393      x = self.image_proj(x)
```

`h` will store outputs at each resolution for skip connection

```
396      h = [x]
```

First half of U-Net

```
398      for m in self.down:
399          x = m(x, t)
400          h.append(x)
```

Middle (bottom)

```
403      x = self.middle(x, t)
```

Second half of U-Net

```
406      for m in self.up:
407          if isinstance(m, Upsample):
408              x = m(x, t)
409          else:
```

Get the skip connection from first half of U-Net and concatenate

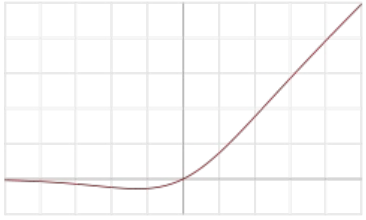
```
411              s = h.pop()
412              x = torch.cat((x, s), dim=1)
```

```
414              x = m(x, t)
```

Final normalization and convolution

```
417      return self.final(self.act(self.norm(x)))
```

Diffusion Model Architecture



Swish Function

- Activation function: `self.act = Swish()`
  - Swish(x) = x * sigmoid(x)
- Normalizing function: `self.norm = nn.GroupNorm(8, n_channels)`
  - GroupNorm divides the channels of a feature map into groups and normalizes each group independently, reducing the dependence on batch size and improving performance across different batch sizes.
  - Diffusion models, particularly those using U-Nets, can process images of varying sizes or resolutions. So independent of batch size is important.
  - (Why not LayerNorm?) LayerNorm is also independent of batch size, could be used in Diffusion models. GroupNorm is usually preferred since it's cheaper than LayerNorm.
- Time Embedding
  - Sinusoidal position embeddings similar with Transformer
  - Embedding goes through MLP for complex, non-linear pattern learning

Create sinusoidal position embeddings same as those from the transformer

$$PE^{(1)}_{t,i} = sin\left(\frac{t}{10000^{\frac{i}{d-1}}}\right)$$

$$PE^{(2)}_{t,i} = cos\left(\frac{t}{10000^{\frac{i}{d-1}}}\right)$$

where d is `half_dim`

```
72        half_dim = self.n_channels // 8
73        emb = math.log(10_000) / (half_dim - 1)
74        emb = torch.exp(torch.arange(half_dim, device=t.device) * -emb)
75        emb = t[:, None] * emb[None, :]
76        emb = torch.cat((emb.sin(), emb.cos()), dim=1)
```

Transform with the MLP

```
79        emb = self.act(self.lin1(emb))
80        emb = self.lin2(emb)
```

```
83        return emb
```

Diffusion Model Architecture

Down Block: Combines ResidualBlock and AttentionBlock. Used in the first half of U-Net.

```python
class DownBlock(Module):

    def __init__(self, in_channels: int, out_channels: int, time_channels: int, has_attn: bool):
        super().__init__()
        self.res = ResidualBlock(in_channels, out_channels, time_channels)
        if has_attn:
            self.attn = AttentionBlock(out_channels)
        else:
            self.attn = nn.Identity()

    def forward(self, x: torch.Tensor, t: torch.Tensor):
        x = self.res(x, t)
        x = self.attn(x)
        return x
```

```python
class Downsample(nn.Module):

    def __init__(self, n_channels):
        super().__init__()
        self.conv = nn.Conv2d(n_channels, n_channels, (3, 3), (2, 2), (1, 1))

    def forward(self, x: torch.Tensor, t: torch.Tensor):
```

Downsample Block: Scale down the feature map by 0.5X with convolution

First half of U-Net:

- Goes through Down Blocks (residual/skip connection and multihead attention)
- Then goes through Downsample Blocks

```python
down = []

out_channels = in_channels = n_channels

for i in range(n_resolutions):

    out_channels = in_channels * ch_mults[i]

    for _ in range(n_blocks):
        down.append(DownBlock(in_channels, out_channels, n_channels * 4, is_attn[i]))
        in_channels = out_channels

    if i < n_resolutions - 1:
        down.append(Downsample(in_channels))
```

Diffusion Model Architecture

Middle Block (Bottleneck):

- Combines a ResidualBlock , AttentionBlock , followed by another ResidualBlock.
- This block is applied at the lowest resolution of the U-Net, since it's the bottleneck

```
class MiddleBlock(Module):

    def __init__(self, n_channels: int, time_channels: int):
        super().__init__()
        self.res1 = ResidualBlock(n_channels, n_channels, time_channels)
        self.attn = AttentionBlock(n_channels)
        self.res2 = ResidualBlock(n_channels, n_channels, time_channels)

    def forward(self, x: torch.Tensor, t: torch.Tensor):
        x = self.res1(x, t)
        x = self.attn(x)
        x = self.res2(x, t)
        return x
```

Diffusion Model Architecture

Up Block: Combines ResidualBlock and AttentionBlock. Used in the second half of U-Net.

```
class UpBlock(Module):

    def __init__(self, in_channels: int, out_channels: int, time_channels: int, has_attn: bool):
        super().__init__()

        self.res = ResidualBlock(in_channels + out_channels, out_channels, time_channels)
        if has_attn:
            self.attn = AttentionBlock(out_channels)
        else:
            self.attn = nn.Identity()

    def forward(self, x: torch.Tensor, t: torch.Tensor):
        x = self.res(x, t)
        x = self.attn(x)
        return x
```

```
class Upsample(nn.Module):

    def __init__(self, n_channels):
        super().__init__()
        self.conv = nn.ConvTranspose2d(n_channels, n_channels, (4, 4), (2, 2), (1, 1))

    def forward(self, x: torch.Tensor, t: torch.Tensor):
```

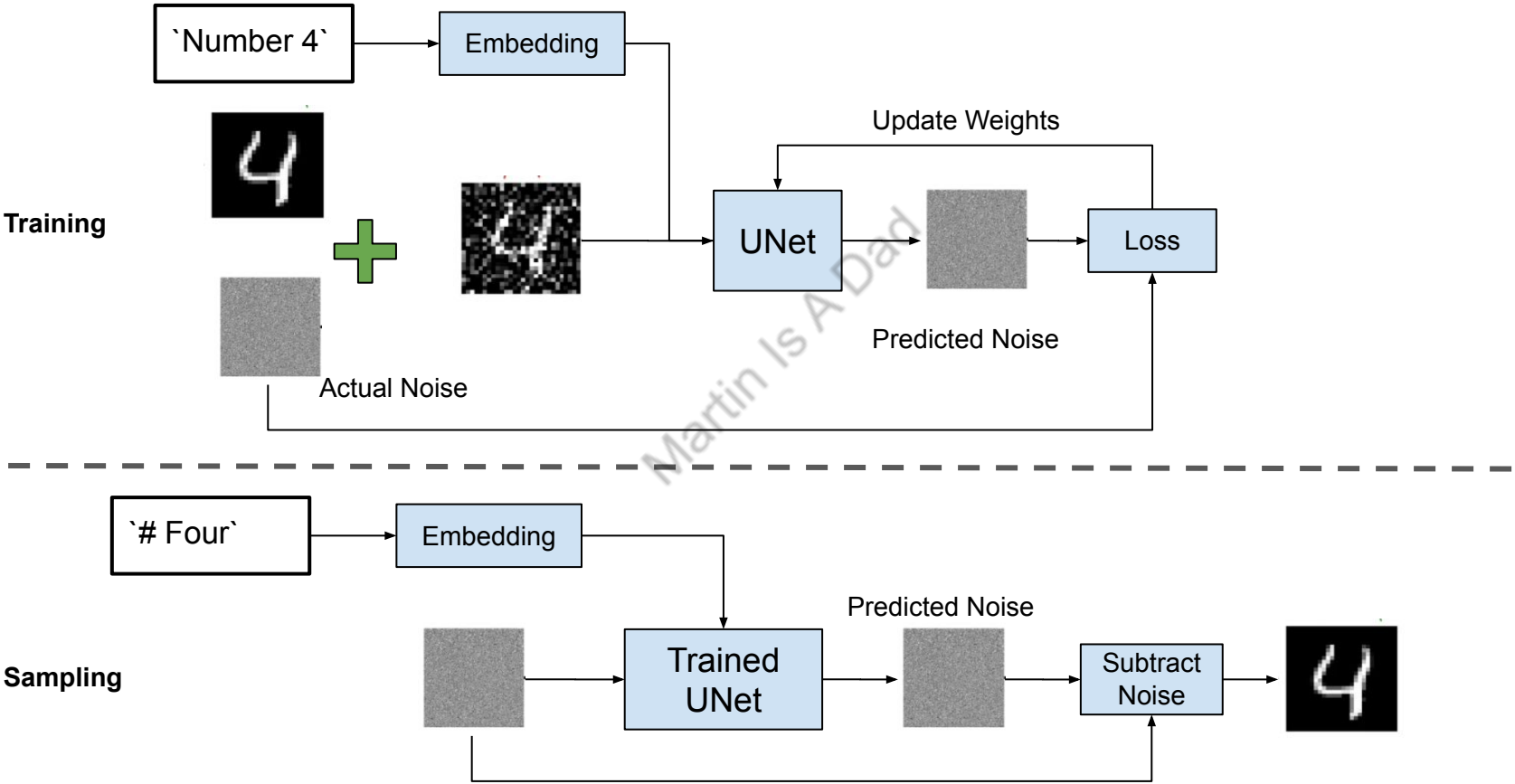
Upsample Block: Scale up the feature map by 2X with transposed convolution

Second half of U-Net:

- Goes through Up Blocks (residual/skip connection and multihead attention)
- Then goes through Upsample Blocks

```
up = □

in_channels = out_channels

for i in reversed(range(n_resolutions)):

    out_channels = in_channels
    for _ in range(n_blocks):
        up.append(UpBlock(in_channels, out_channels, n_channels * 4, is_attn[i]))

    out_channels = in_channels // ch_mults[i]
    up.append(UpBlock(in_channels, out_channels, n_channels * 4, is_attn[i]))
    in_channels = out_channels

    if i > 0:
        up.append(Upsample(in_channels))
```

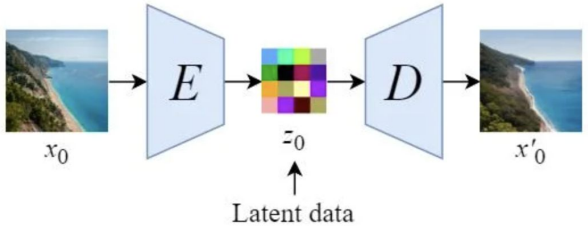
Guided Diffusion Models

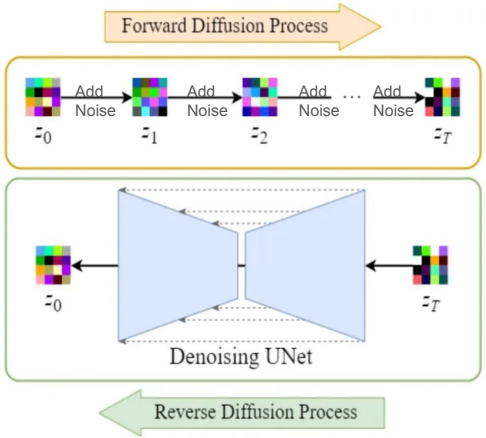


Latent Diffusion Model (LDM or Stable Diffusion)

- Motivation
  - Training and sampling of diffusion model can be expensive, computation and memory wise, since it requires multiple denoising passes
  - Can we introduce a lower dimension latent space and perform diffusion there?
  - **Stable Diffusion** performs the diffusion process in the latent space

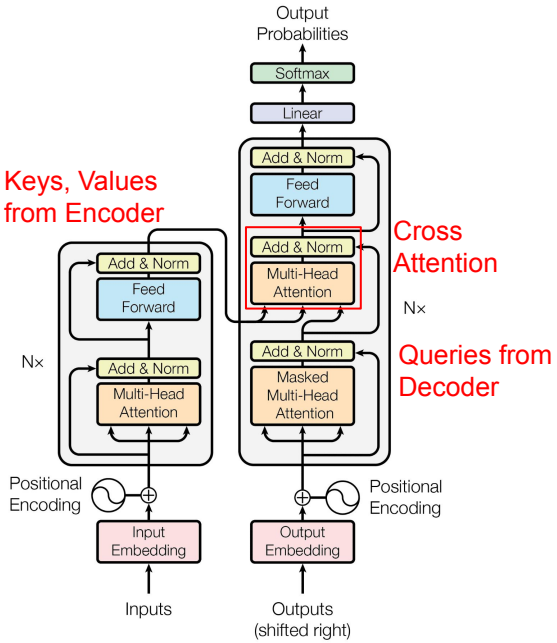
- TLDR
  - Trained Encoder for encoding a full-size image to a lower dimension representation (latent space).
  - Forward diffusion process and the reverse diffusion process within the latent space.
  - Trained Decoder can decode the image from its latent representation back to the pixel-space



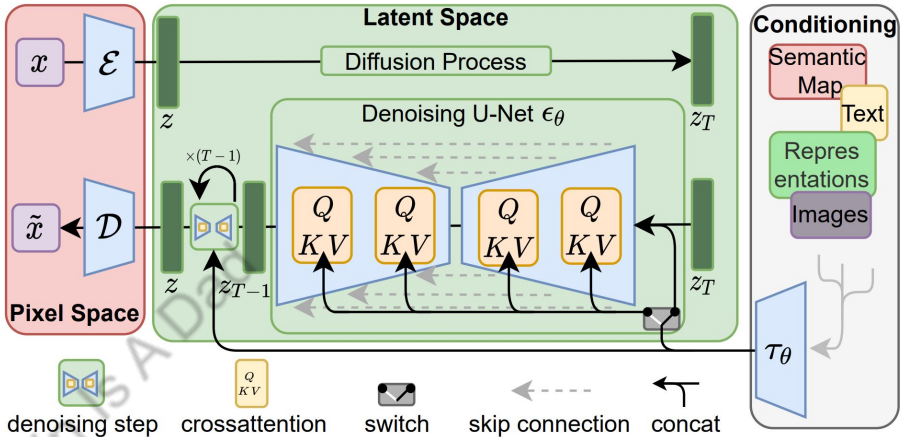


Images from codoraven.com/blog/ai/stable-diffusion-clearly-explained

Text Conditioning With Cross Attention

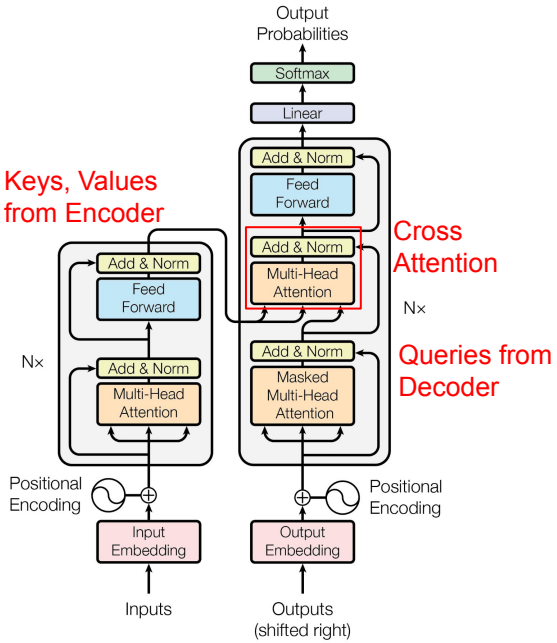


Intuition: For a given decoder query, what encoder information is the most important to pay attention to?



- Stable Diffusion model can generate images from text prompts. The inner diffusion model is turned into a conditional image generator by augmenting its denoising U-Net with the cross-attention mechanism.
- Text inputs are first converted into embeddings using a language model τ_θ (e.g. BERT, CLIP), and then they are mapped into the U-Net via the Multi-head Attention layer.
  - K, V are from text prompts, Q is from the latent representation of image

Text Conditioning With Cross Attention



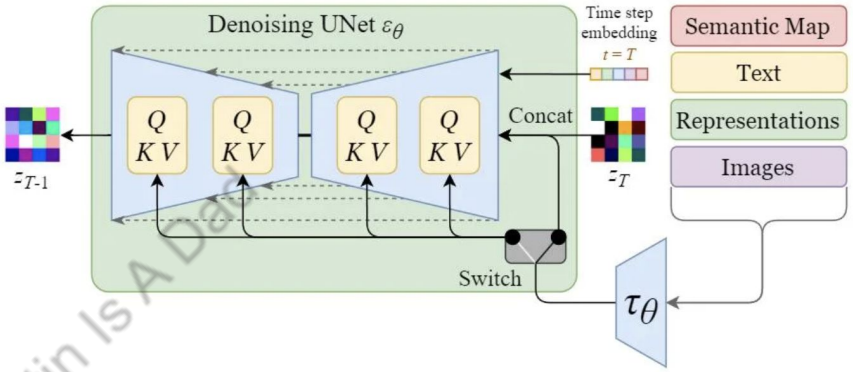


Image from codoraven.com/blog/ai/stable-diffusion-clearly-explained

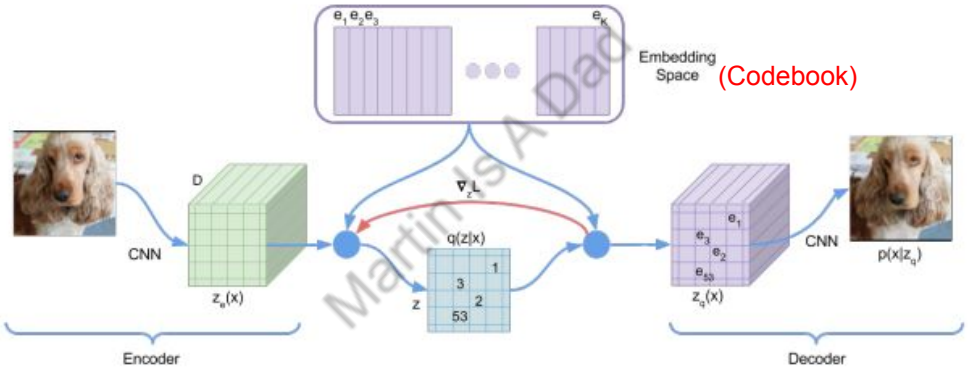
- Other conditioning inputs such as spatially aligned data (semantic maps, images etc), the conditioning is done using concatenation.

Intuition: For a given decoder query, what encoder information is the most important to pay attention to?

LDM Training (Phase 1)

Two Phased Training:

- **Phase 1**: Training Autoencoder reproducing lower dimensional representations
  - Using VQ-VAE (Vector Quantization VAE)
  - Latent representation has a structure of a downsampled image

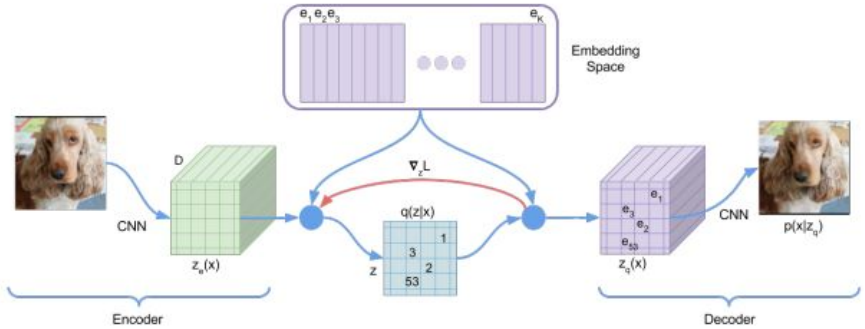


- VQ-VAE vs VAE
  - VAEs use a continuous latent space, while VQ-VAEs use a discrete latent space represented by a learned ***codebook***.
  - Code is a learned vocabulary or dictionary of embedding vectors. It is a finite set of *K* vectors, where each vector, often referred to as a "codevector" or "embedding," has a specific dimensionality *D*. These K vectors represent the possible discrete states that the VQ-VAE's encoder can map its continuous output to

LDM Training (Phase 1)

How VQ-VAE works:

- **Encoding:** The encoder processes the input data (e.g., an image or audio) and produces a continuous latent representation.
- **Quantization:** Instead of directly passing this continuous representation to the decoder, the VQ layer steps in. For each vector in the continuous latent representation, it finds the closest matching vector in the codebook based on a distance metric (commonly Euclidean distance or edit distance).
- **Discrete Representation:** The index of the closest codebook vector is then used as the discrete latent code for that part of the input. Effectively, the continuous vector is "quantized" or snapped to the nearest available codebook entry.
- **Decoding:** The decoder receives the selected codebook vector (the discrete representation) and uses it to reconstruct the original input data



LDM Training (Phase 1)

Why Use VQ-VAE instead of VAE?

- **More Meaningful Latent Codes**
  - VQ-VAE quantizes the encoder's output to the nearest vector in a learned codebook. This forces the model to pick a specific discrete representation from a finite set. Each codebook vector can potentially represent a recurring pattern or concept in the data. This can lead to a more interpretable and structured latent space.
  - VAE maps to a continuous distribution, it can sometimes lead to less distinct or fuzzier latent representations

- **Avoidance of "Posterior Collapse"**
  - VAEs when paired with powerful decoders can suffer from "posterior collapse." This is where the latent variables become uninformative, and the decoder learns to ignore them, relying mostly on its own parameters. The KL divergence term in the VAE objective, which pushes the learned posterior towards the prior, can exacerbate this.
  - In VQ-VQE, the model is forced to use the information encoded in the selected codebook entries to reconstruct the input. This ensures the latent space remains informative.

- **Computational Efficiency for the Diffusion Model**
  - VQ-VAE has very high compression ratio, the sequence of discrete latent codes is much smaller than the original data
  - The discrete tokens produced by a VQ-VAE are well-suited for powerful sequential models like Transformers or diffusion models decoder

LDM Training (Phase 2)

Phase 2: Use trained encoder and decoder to perform diffusion:

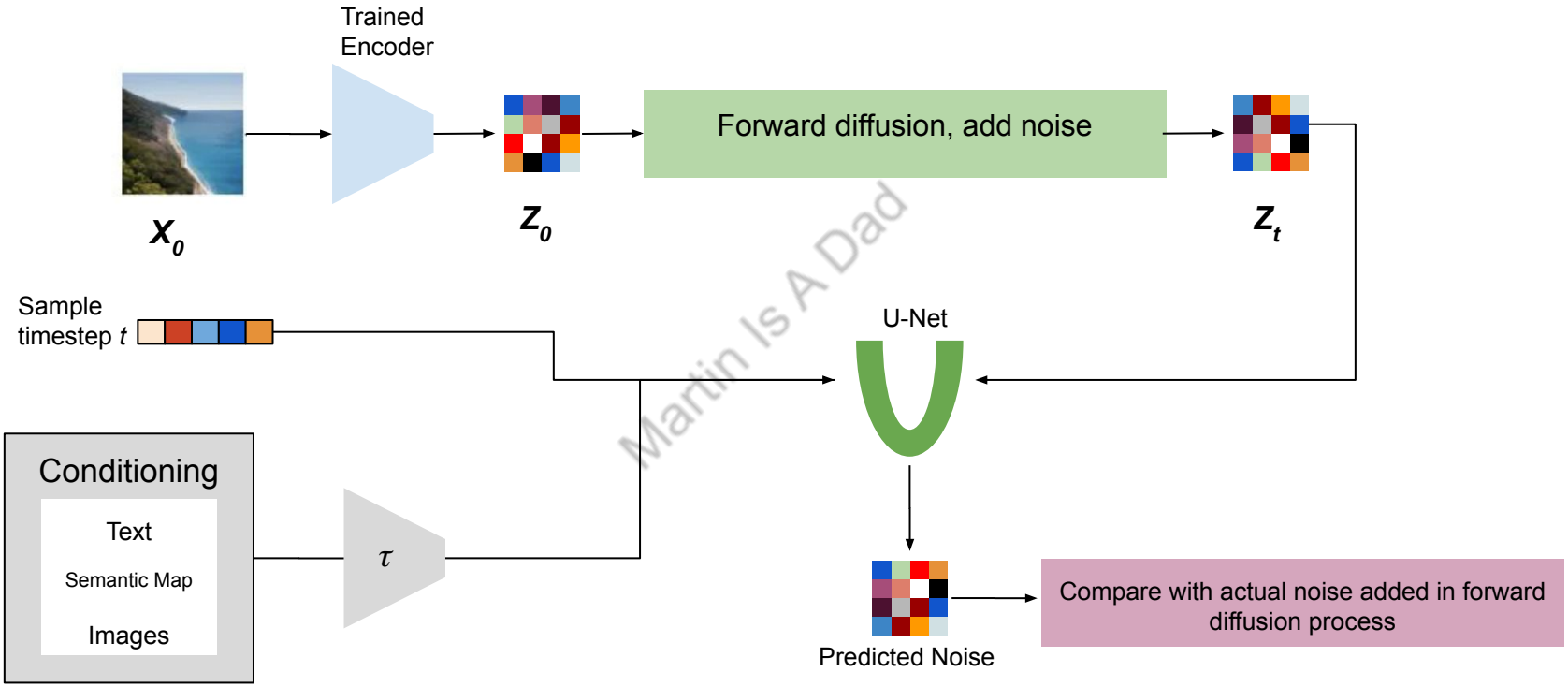
- Very similar to DDPM, with few differences (input is in latent space, conditioning cross attention etc)
- Conditioning through cross-attention layer operating on conditioning input's embeddings from encoder τ_θ feeding into keys and values:

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

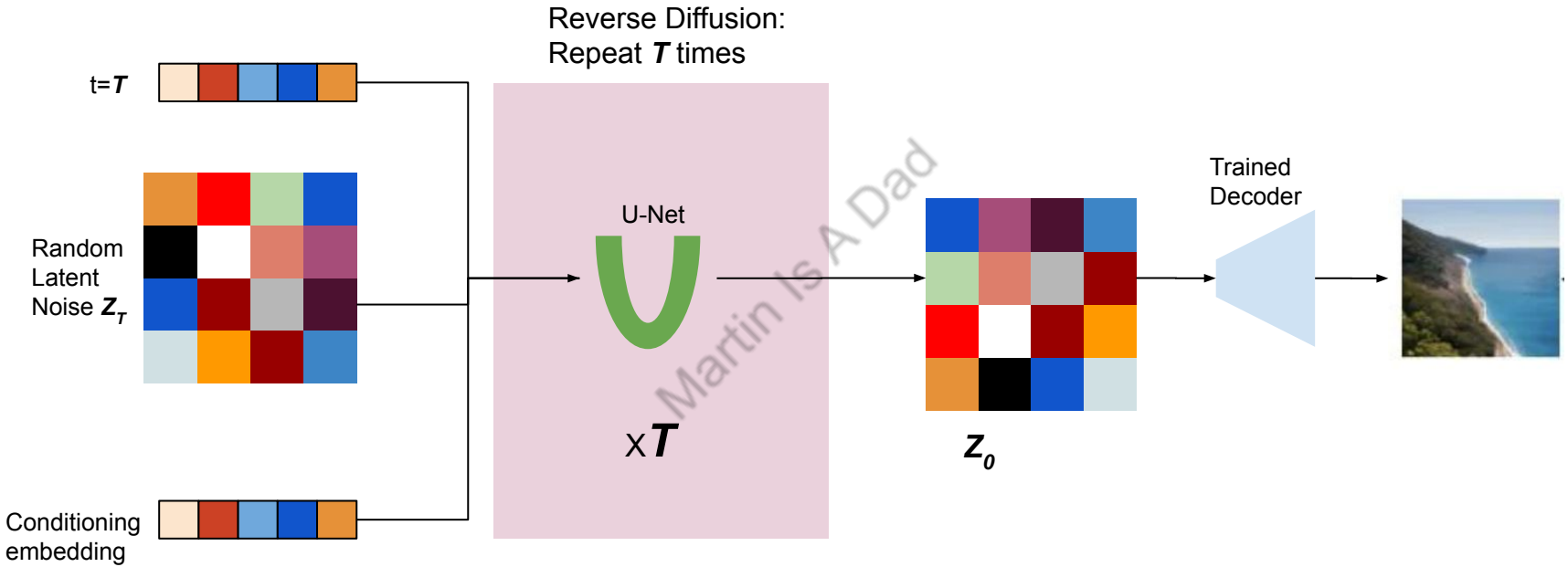
- Loss function with conditioning:

$$L_{LDM} := \mathbb{E}_{\mathcal{E}(x),y,\epsilon\sim\mathcal{N}(0,1),t}\left[\|\epsilon - \epsilon_\theta(z_t, t, \tau_\theta(y))\|_2^2\right]$$

LDM Training (Phase 2)



LDM Sampling



Since the size of the latent space is a lot smaller than the original image size, the denoising process is much faster.