Join at

slido.com


#3098404

# Grooming Diffusion

## Basic diffusion


NAVER AI Lab

김준호

https://github.com/taki0112


NAVER AI LAB

https://github.com/taki0112/diffusion-pytorch

# Recall

NAVER AI LAB

Youtube

- The recipe of GANs
  - 2014 ~ 2020 GANs 모델 연구 요약 (기초 ~ 심화)
- The diffusion theory
  - diffusion을 이해하기 위한 이론 (기초)
- The applications of diffusion
  - Text-to-image 모델 소개 (심화)

**완전히 똑같진 않으나, 복습하기에 좋을것입니다. : )**

# Junho Kim

* Research Scientist @ NAVER AI Lab

* Advisor @ BiX

* Google scholar citations: 1700++

* Github Stars: 16000++

01
## ML Research

- Multimodal Learning
- Machine Learning Algorithm
- Trustworthy AI

02
## Backbone Research

- Backbone Architecture
- Representation Learning
- Machine Learning Optimization

03
## Language Research

- Large Language Models
- Language Representations
- AI Ethics & Safety

04
## Generation Research

- Diffusion & NeRFs
- Multi-modal Generation
- Real-Time 3D Rendering

05
## HCI Research

- AI-Infused Interactive Applications
- Understanding People around AI

07
## Healthcare AI

- LLM-based Healthcare AI
- Medical Data AI

01
## ML Research

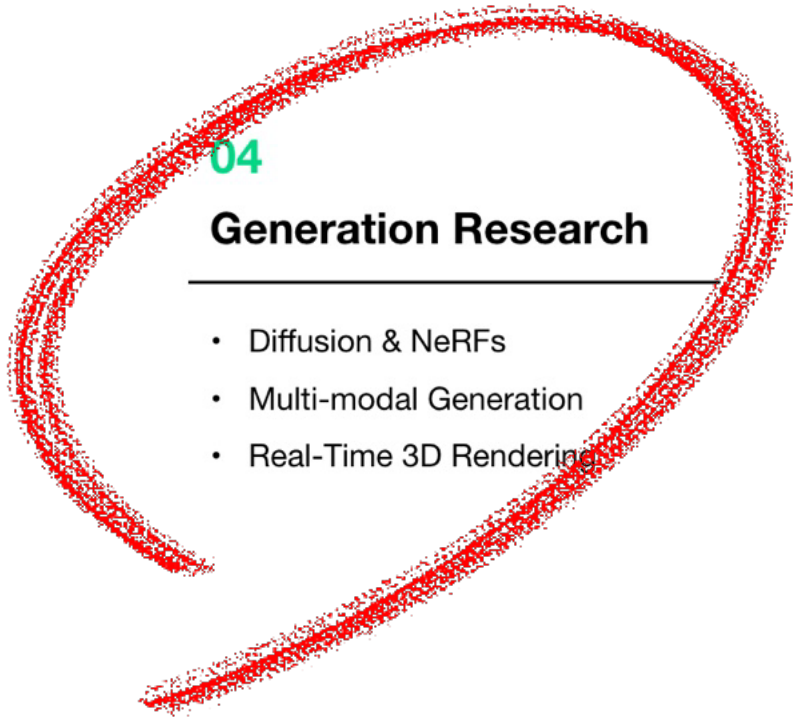
- Multimodal Learning
- Machine Learning Algorithm
- Trustworthy AI

02
## Backbone Research

- Backbone Architecture
- Representation Learning
- Machine Learning Optimization

03
## Language Research

- Large Language Models
- Language Representations
- AI Ethics & Safety



04
## Generation Research

- Diffusion & NeRFs
- Multi-modal Generation
- Real-Time 3D Rendering

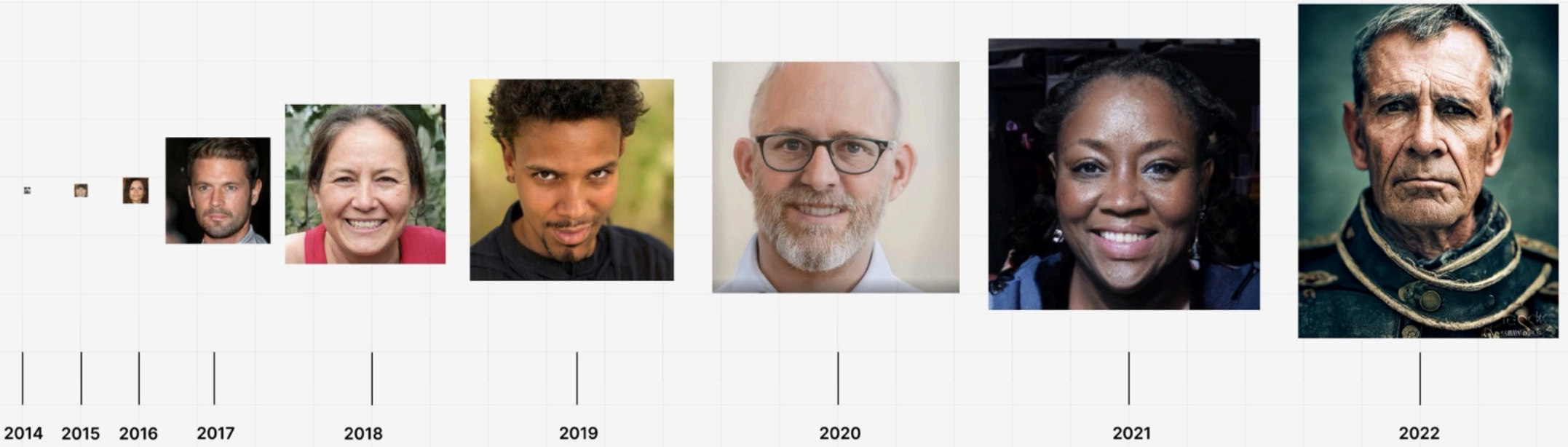
05
## HCI Research

- AI-Infused Interactive Applications
- Understanding People around AI

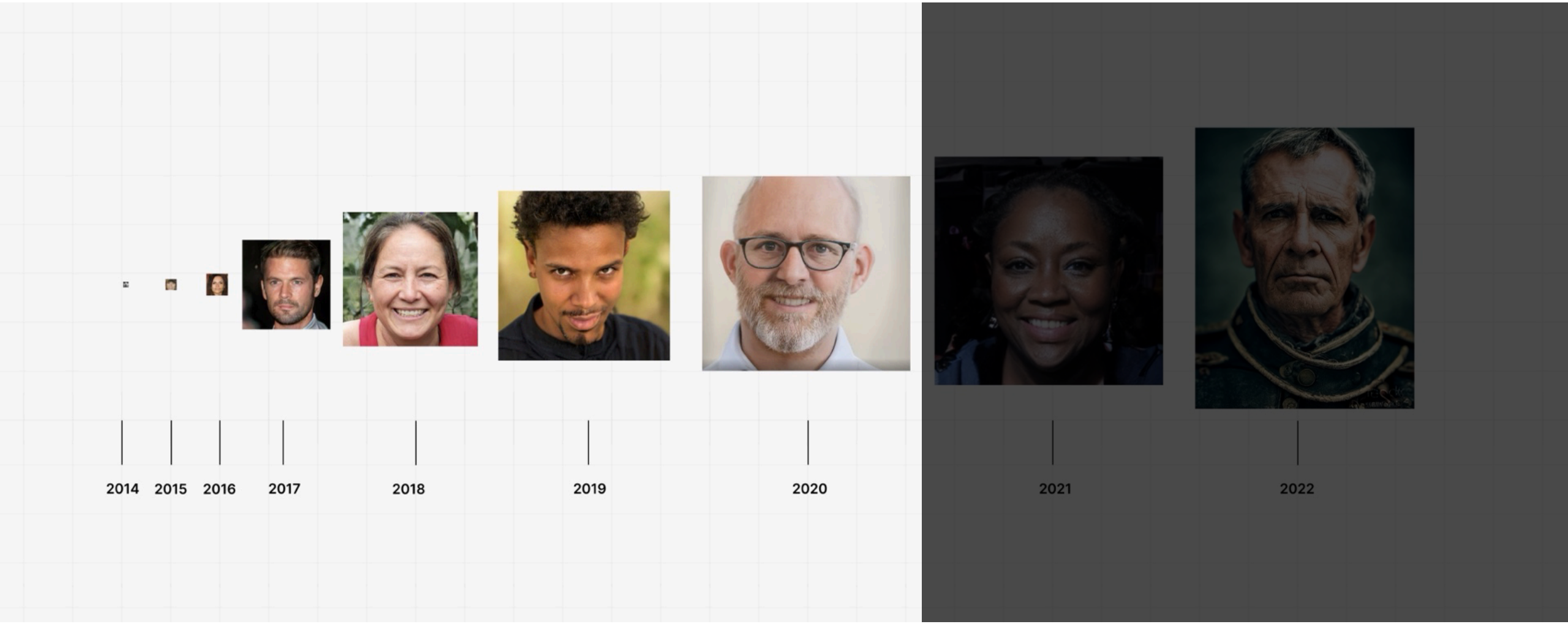
07
## Healthcare AI

- LLM-based Healthcare AI
- Medical Data AI

NAVER AI LAB



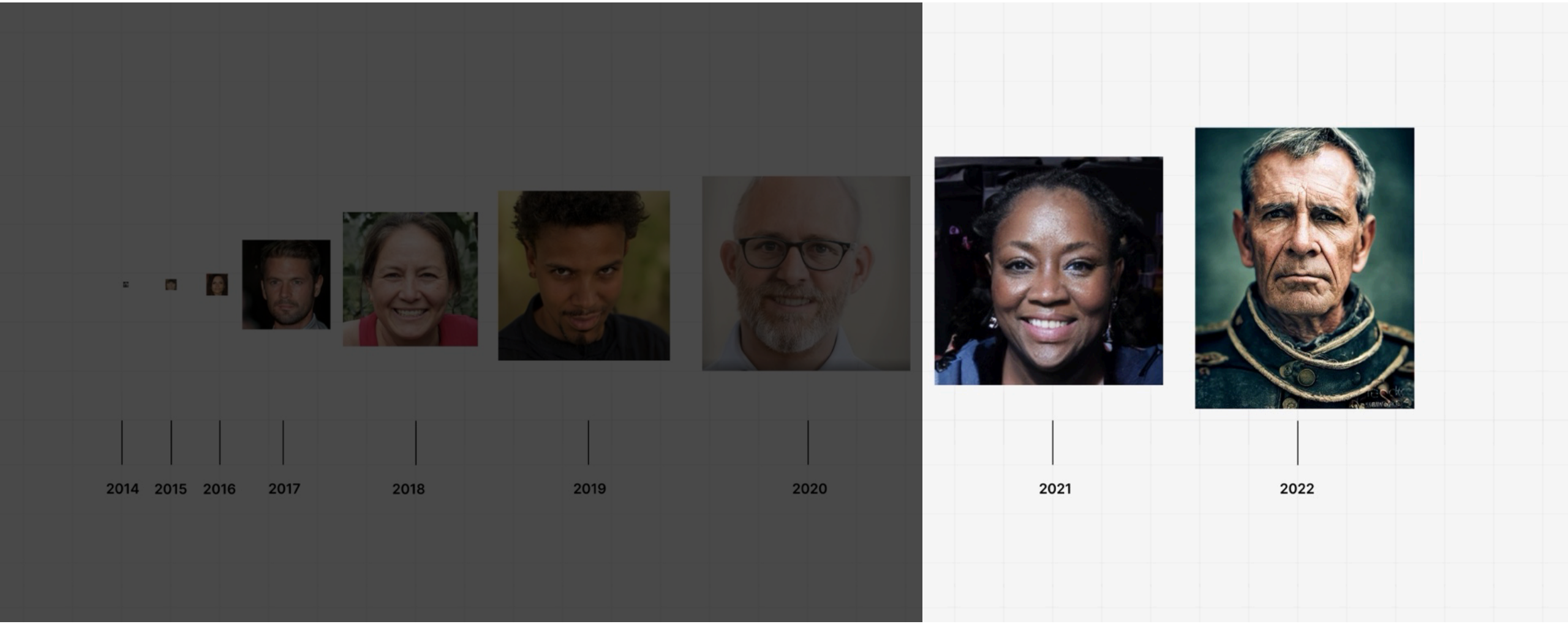
NAVER AI LAB




## 과거를 이해하는 시간

16개 논문

NAVER AI LAB




## 현재를 이해하고, 미래를 준비하는 시간

Grooming the diffusion

# Summary of GANs



**# 2014 ~ 2016**

- Unconditional generation
  - GANs
  - Diverse loss
- Conditional generation
  - ACGAN
  - Multi-task discriminator
  - Projection discriminator

NAVER AI LAB

## # 2014 ~ 2016

- Unconditional generation
  - GANs
  - Diverse loss
- Conditional generation
  - ACGAN
  - Multi-task discriminator
  - Projection discriminator

## # 2017 ~ 2018

- Progressive GAN
  - Progressive training
- BigGAN
  - Conditional batch normalization
  - Large scale
  - Truncation trick
- StyleGAN
  - Disentangle the latent space with mapping layer
  - Style Mixing (determine the coarse, middle, fine style)
    - A module = Global aspects
    - B module = Local aspects
  - Truncation trick

NAVER AI LAB

## # 2014 ~ 2016

- Unconditional generation
  - GANs
  - Diverse loss
- Conditional generation
  - ACGAN
  - Multi-task discriminator
  - Projection discriminator

## # 2017 ~ 2018

- Progressive GAN
  - Progressive training
- BigGAN
  - Conditional batch normalization
  - Large scale
  - Truncation trick
- StyleGAN
  - Disentangle the latent space with mapping layer
  - Style Mixing (determine the coarse, middle, fine style)
    - A module = Global aspects
    - B module = Local aspects
  - Truncation trick

## # 2019 ~ 2020

- StyleGAN2
  - StyleGAN + Weight modulation + Lazy regularization.
- DiffAugment
  - Prevent the overfitting in a discriminator.
  - Apply the differentiable augmentation to generator & discriminator.
- ADA
  - Prevent the overfitting in a discriminator.
  - Apply the adaptively augmentation to generator & discriminator.

NAVER AI LAB

## # 2014 ~ 2016

- Unconditional generation
  - GANs
  - Diverse loss
- Conditional generation
  - ACGAN
  - Multi-task discriminator
  - Projection discriminator

## # 2017 ~ 2018

- Progressive GAN
  - Progressive training
- BigGAN
  - Conditional batch normalization
  - Large scale
  - Truncation trick
- StyleGAN
  - Disentangle the latent space with mapping layer
  - Style Mixing (determine the coarse, middle, fine style)
    - A module = Global aspects
    - B module = Local aspects
  - Truncation trick

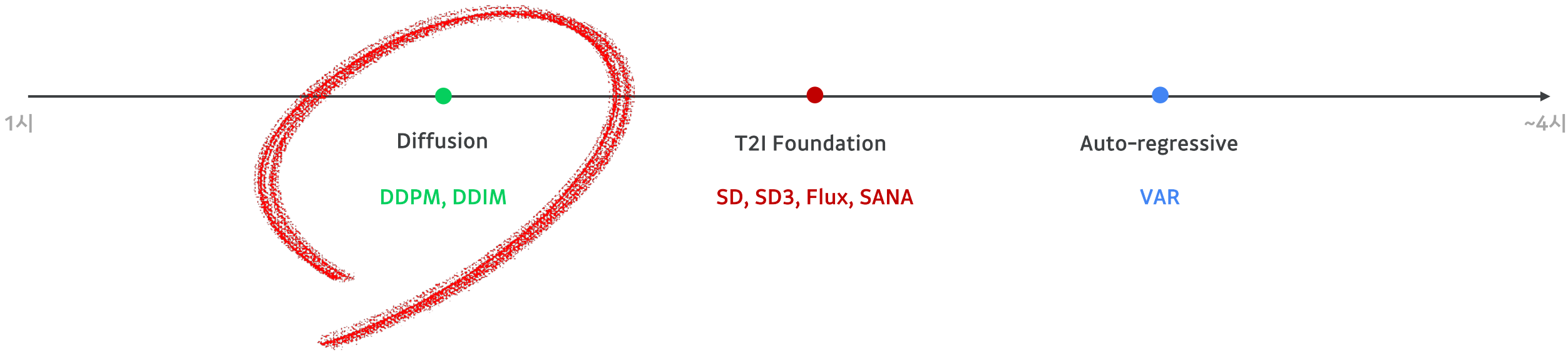
## # 2019 ~ 2020

- StyleGAN2
  - StyleGAN + Weight modulation + Lazy regularization.
- DiffAugment
  - Prevent the overfitting in a discriminator.
  - Apply the differentiable augmentation to generator & discriminator.
- ADA
  - Prevent the overfitting in a discriminator.
  - Apply the adaptively augmentation to generator & discriminator.

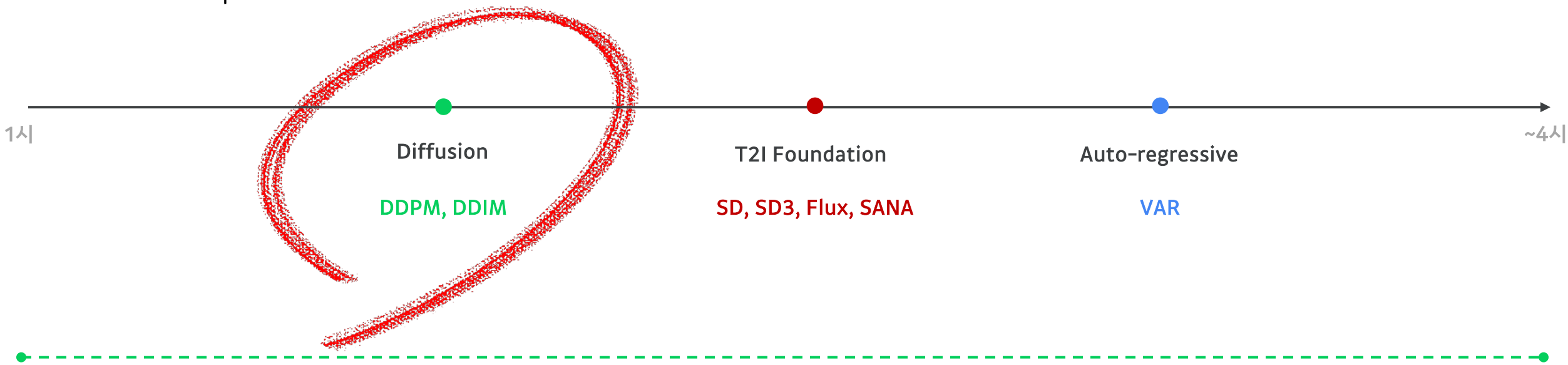
## # 2014 ~ 2020: Techniques

- Consistency regularization
  - CR-GAN: augmented real images for discriminator.
  - bCR-GAN: augmented real & fake images for discriminator.
  - zCR-GAN: augmented latent codes for generator & discriminator.
  - ICR-GAN: bCR + zCR
- FSMR: Feature Statistics Mixing Regularization
  - Reduce style-bias in discriminator.
- GGDR: Generator Guided Discriminator Regularization
  - Dense supervision for discriminator.

NAVER AI LAB

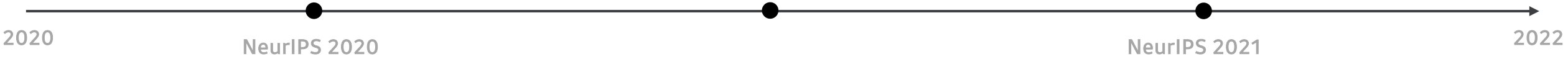

1시
~4시
Diffusion
DDPM, DDIM
T2I Foundation
SD, SD3, Flux, SANA
Auto-regressive
VAR

NAVER AI LAB




Denoising diffusion implicit models (DDIM)

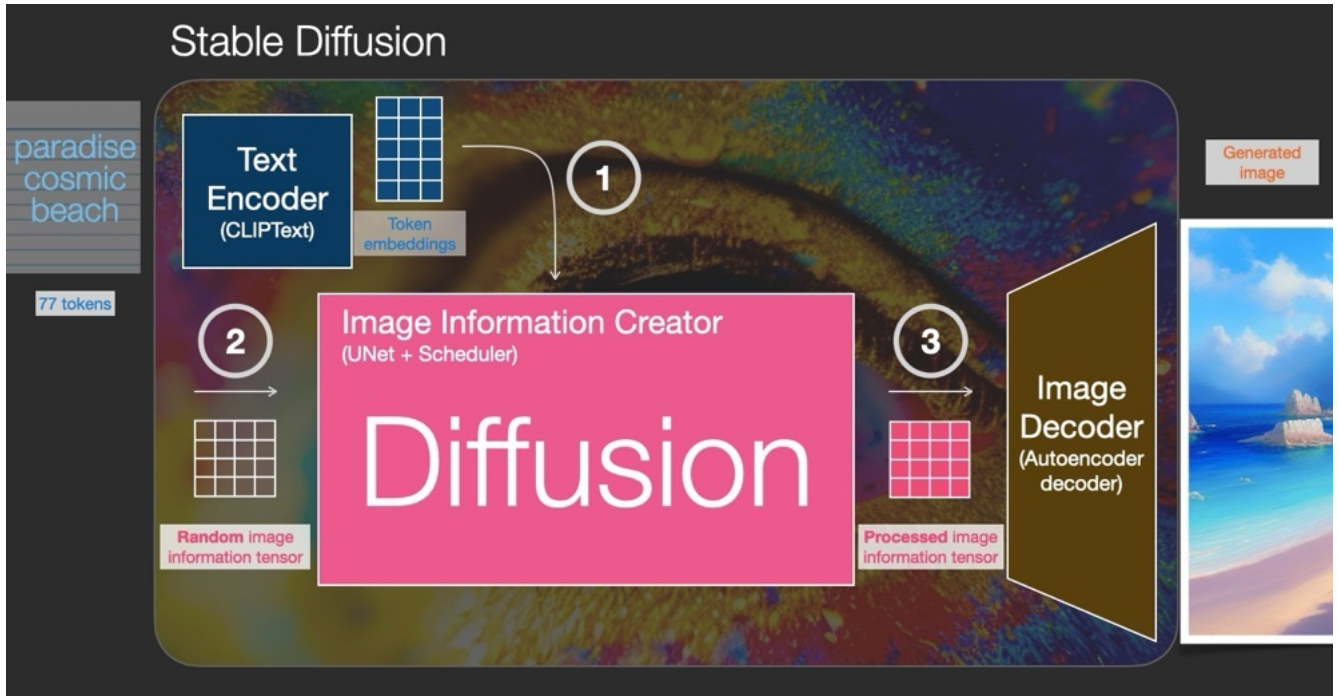
ICLR 2021




Denoising diffusion probabilistic models (DDPM)

Diffusion Models Beat GANs on Image Synthesis

NAVER AI LAB




*"An astronaut riding a horse in a photorealistic style"*

↓

DALLE 2

↓

*"A brain riding a rocketship heading towards the moon"*

↓

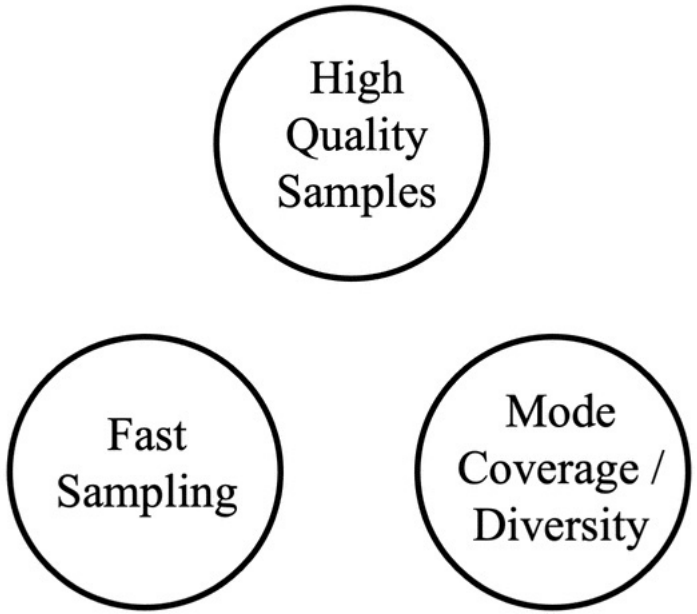
Imagen

↓

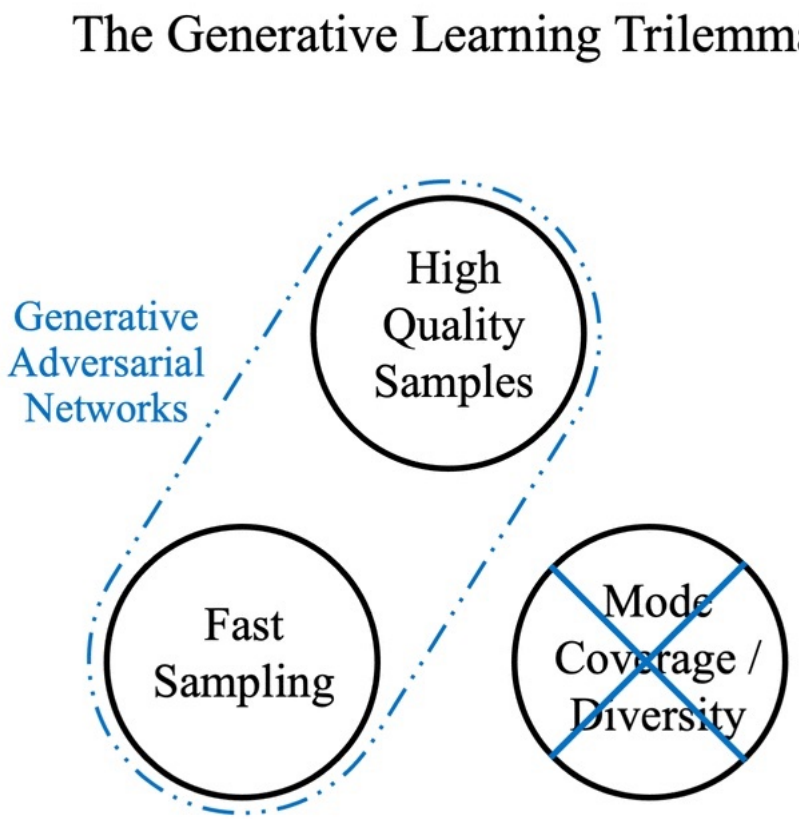
jalammar.github.io/illustrated-stable-diffusion/

NAVER AI LAB

The Generative Learning Trilemma



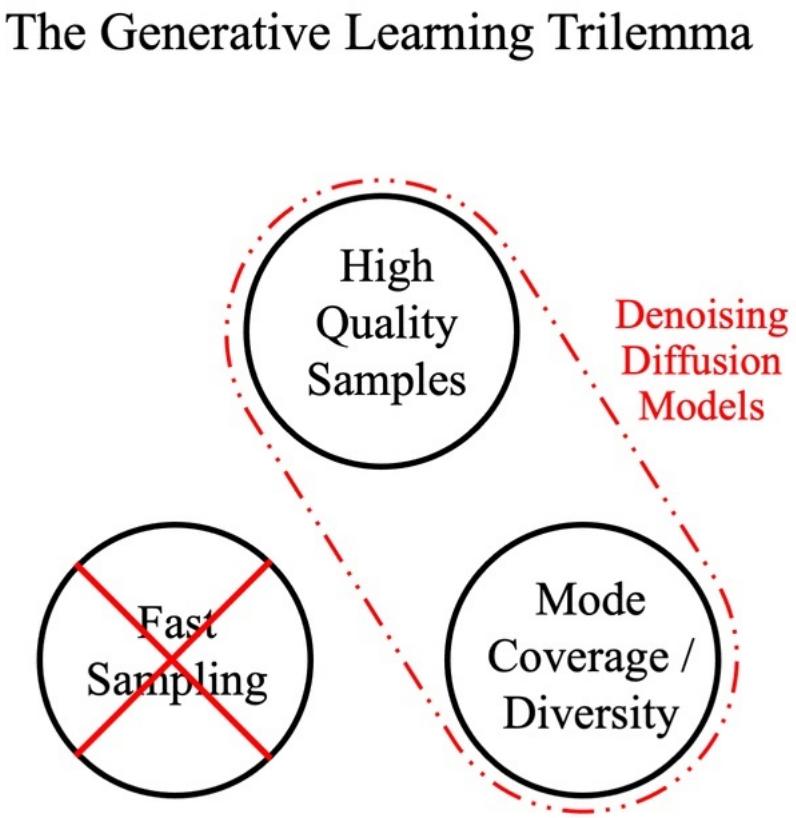

Tackling the Generative Learning Trilemma with Denoising Diffusion GANs. ICLR 2022

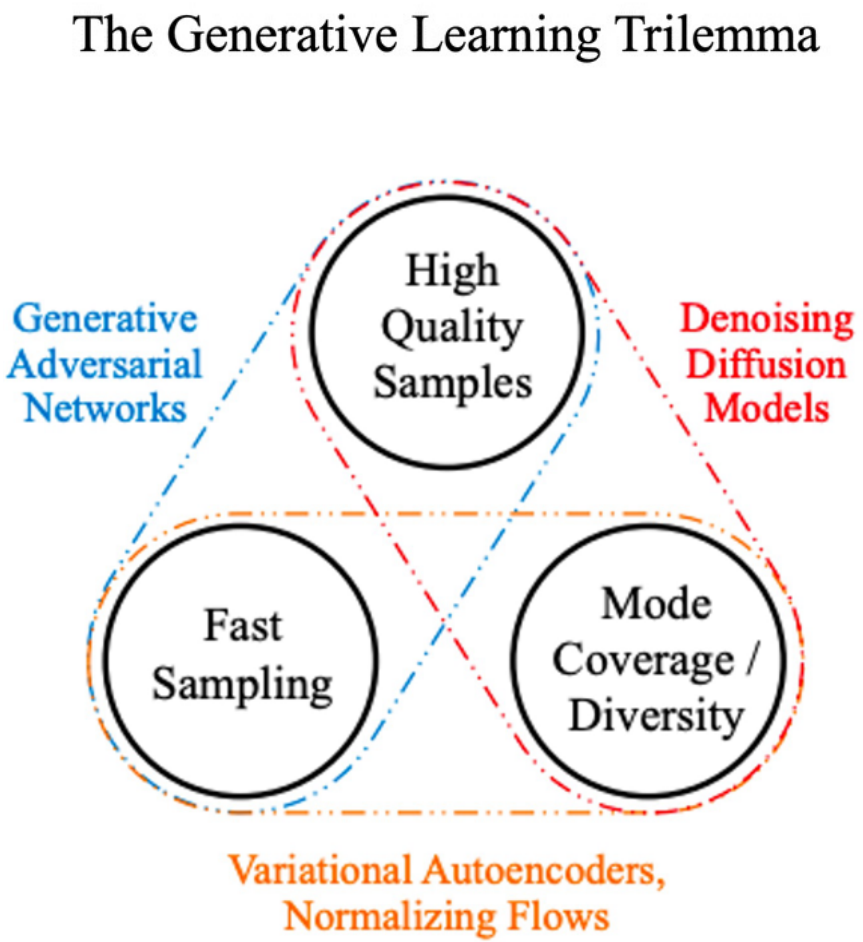


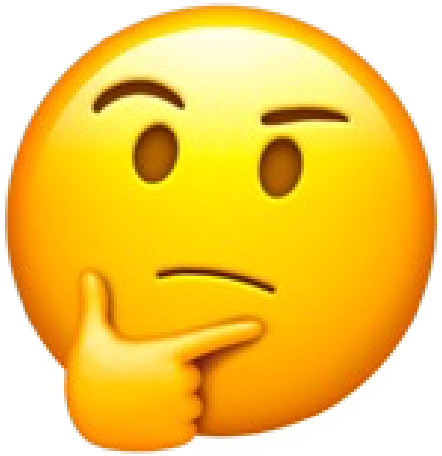

Tackling the Generative Learning Trilemma with Denoising Diffusion GANs. ICLR 2022

NAVER AI LAB

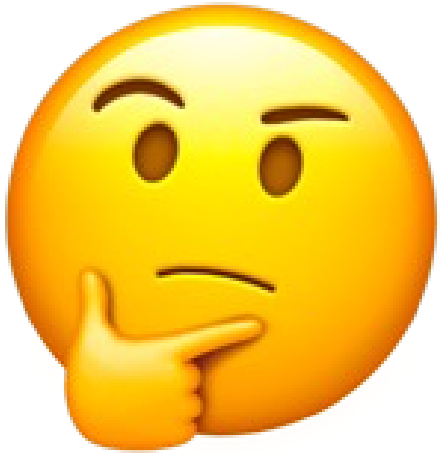

Tackling the Generative Learning Trilemma with Denoising Diffusion GANs. ICLR 2022


The Generative Learning Trilemma
High Quality Samples
Denoising Diffusion Models
Fast Sampling
Mode Coverage / Diversity


The Generative Learning Trilemma
High Quality Samples
Generative Adversarial Networks
Denoising Diffusion Models
Fast Sampling
Mode Coverage / Diversity
Variational Autoencoders, Normalizing Flows



# Diffusion process



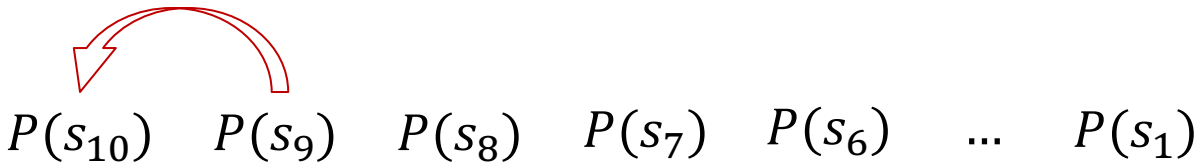
NAVER AI LAB

## ❑ Markov Chain

- Markov 성질을 갖는 이산 확률과정

  ✓ Markov 성질 : "특정 상태의 확률(t+1)은 오직 현재(t)의 상태에 의존한다"

  ✓ 이산 확률과정 : 이산적인 시간(0초, 1초, 2초, ..) 속에서의 확률적 현상



$$P(s_{10}) \quad P(s_9) \quad P(s_8) \quad P(s_7) \quad P(s_6) \quad \dots \quad P(s_1)$$

$$P[s_{t+1}|s_t] = \mathrm{P}[s_{t+1}|s_1, .., s_t]$$

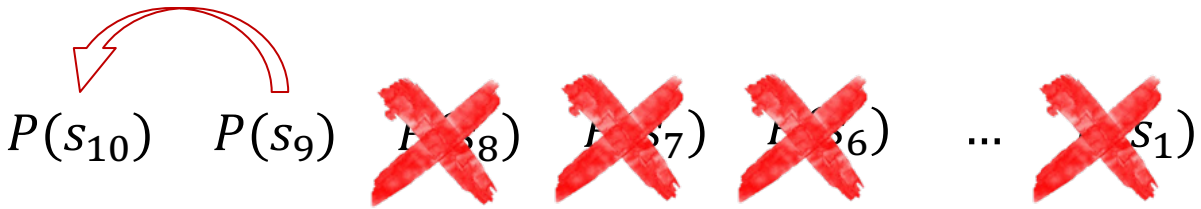
NAVER AI LAB

## ❑ Markov Chain

- Markov 성질을 갖는 이산 확률과정

  ✓ Markov 성질 : "특정 상태의 확률(t+1)은 오직 현재(t)의 상태에 의존한다"

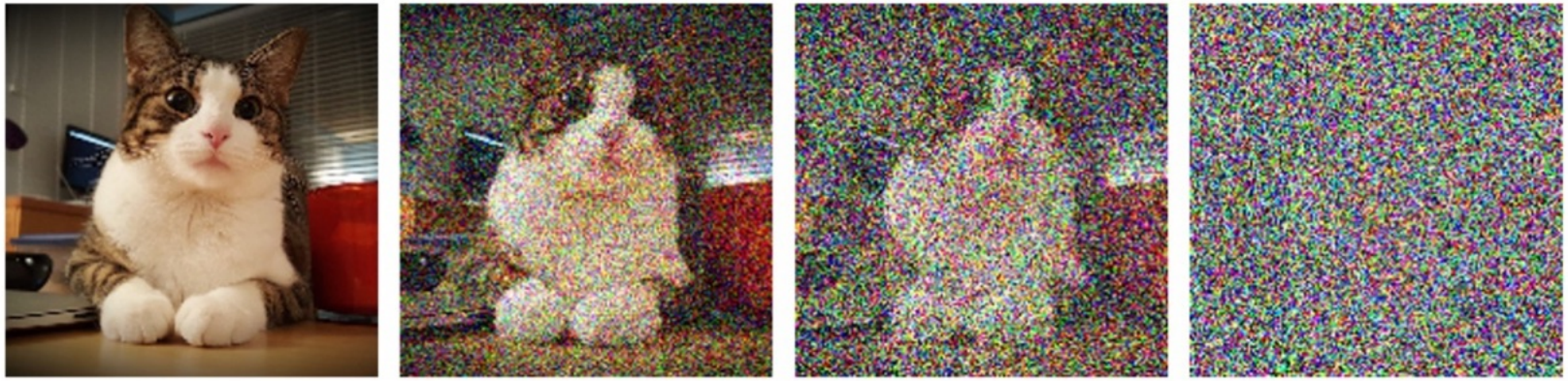
  ✓ 이산 확률과정 : 이산적인 시간(0초, 1초, 2초, ..) 속에서의 확률적 현상




$$P[s_{t+1}|s_t] = \mathrm{P}[s_{t+1}|s_1, .., s_t]$$

NAVER AI LAB

+ noise



$X_0$

$X_T$

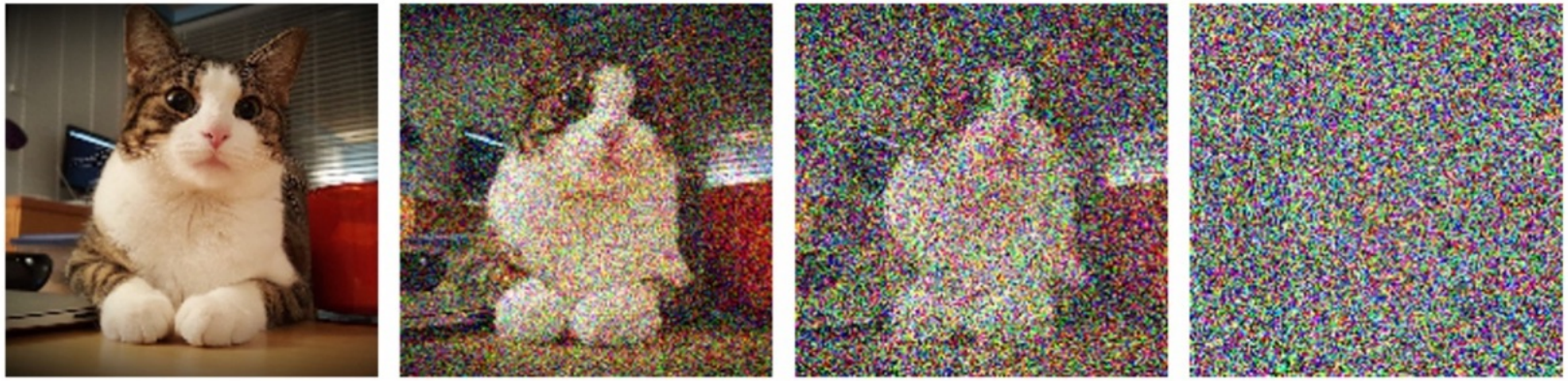
- noise

NAVER AI LAB

Diffusion process

+ noise

$X_0$



$X_T$

– noise

Denoising process

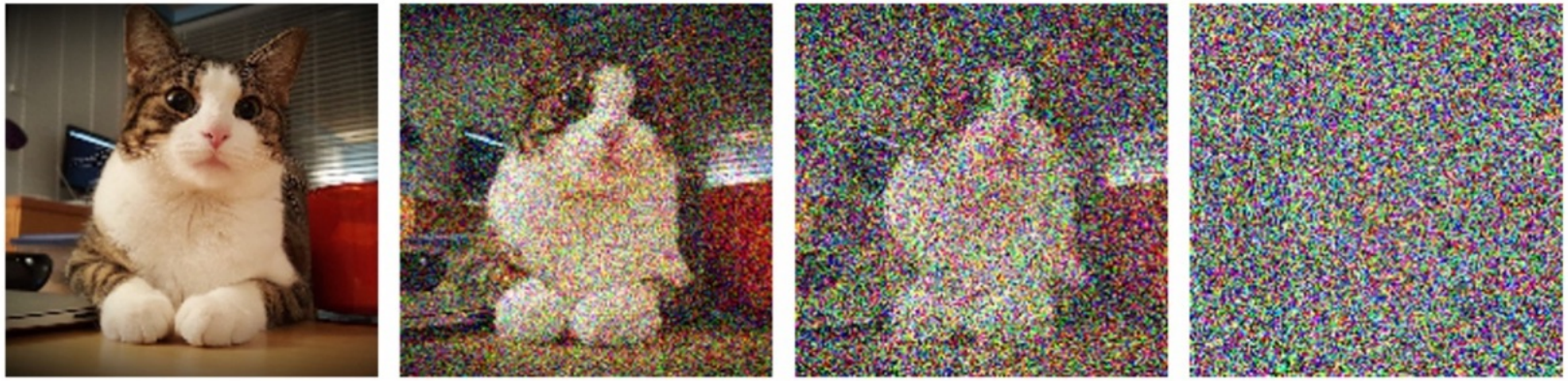
NAVER AI LAB

Forward process

Diffusion process

+ noise

$X_0$



$X_T$

- noise

Denoising process

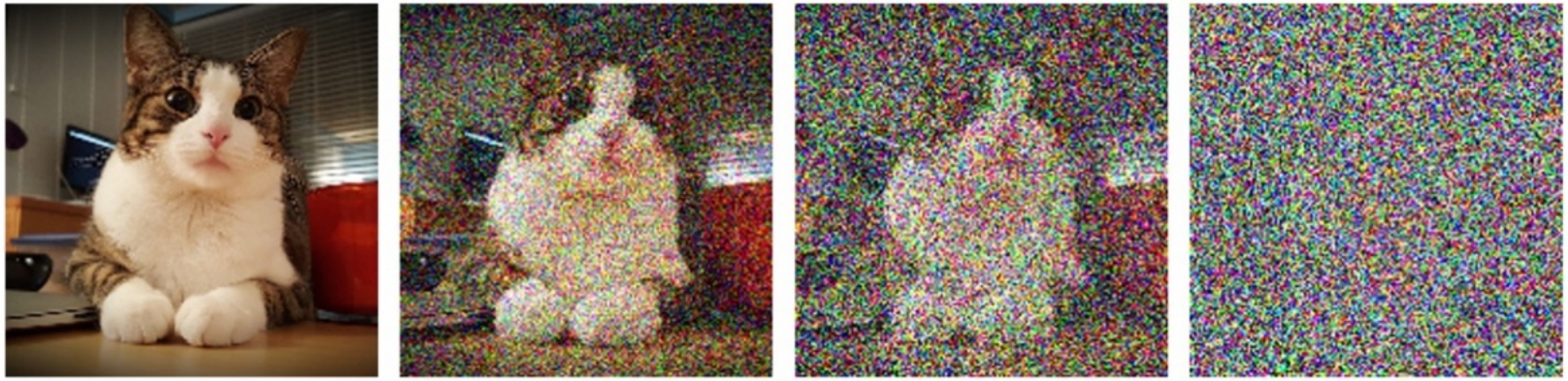
Reverse process

NAVER AI LAB

Forward process

Diffusion process

$q(X_t \mid X_{t-1})$ + noise



$X_0$ $X_T$

– noise $q(X_{t-1} \mid X_t)$

Denoising process

Reverse process

NAVER AI LAB

Forward process

이건 학습없이 가능함

Diffusion process

$q(X_t \mid X_{t-1})$

+ noise

$X_0$

$X_T$

- noise

$q(X_{t-1} \mid X_t)$

학습을 통해
가능하게 만들자

Denoising process

Reverse process

NAVER AI LAB

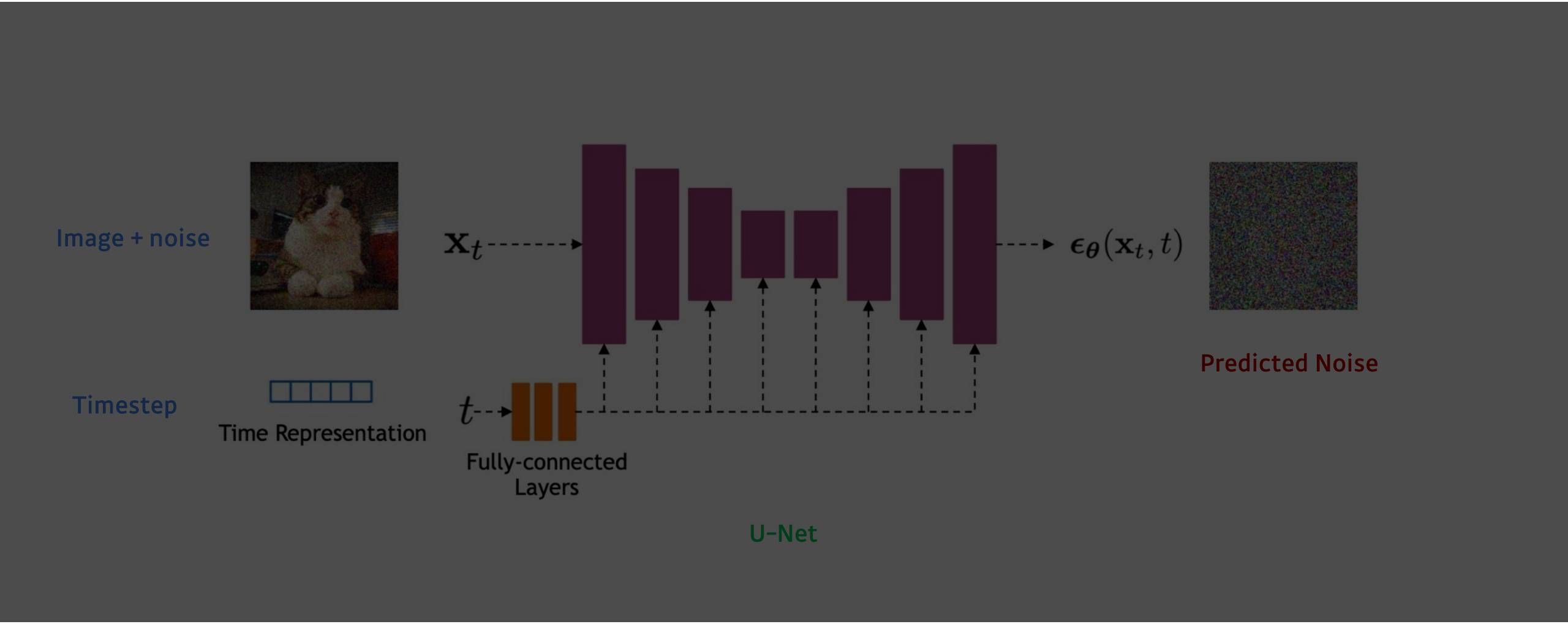


Image + noise
$\mathbf{x}_t$
$\epsilon_\theta(\mathbf{x}_t, t)$
Predicted Noise
Timestep
Time Representation
$t$
Fully-connected Layers
U-Net

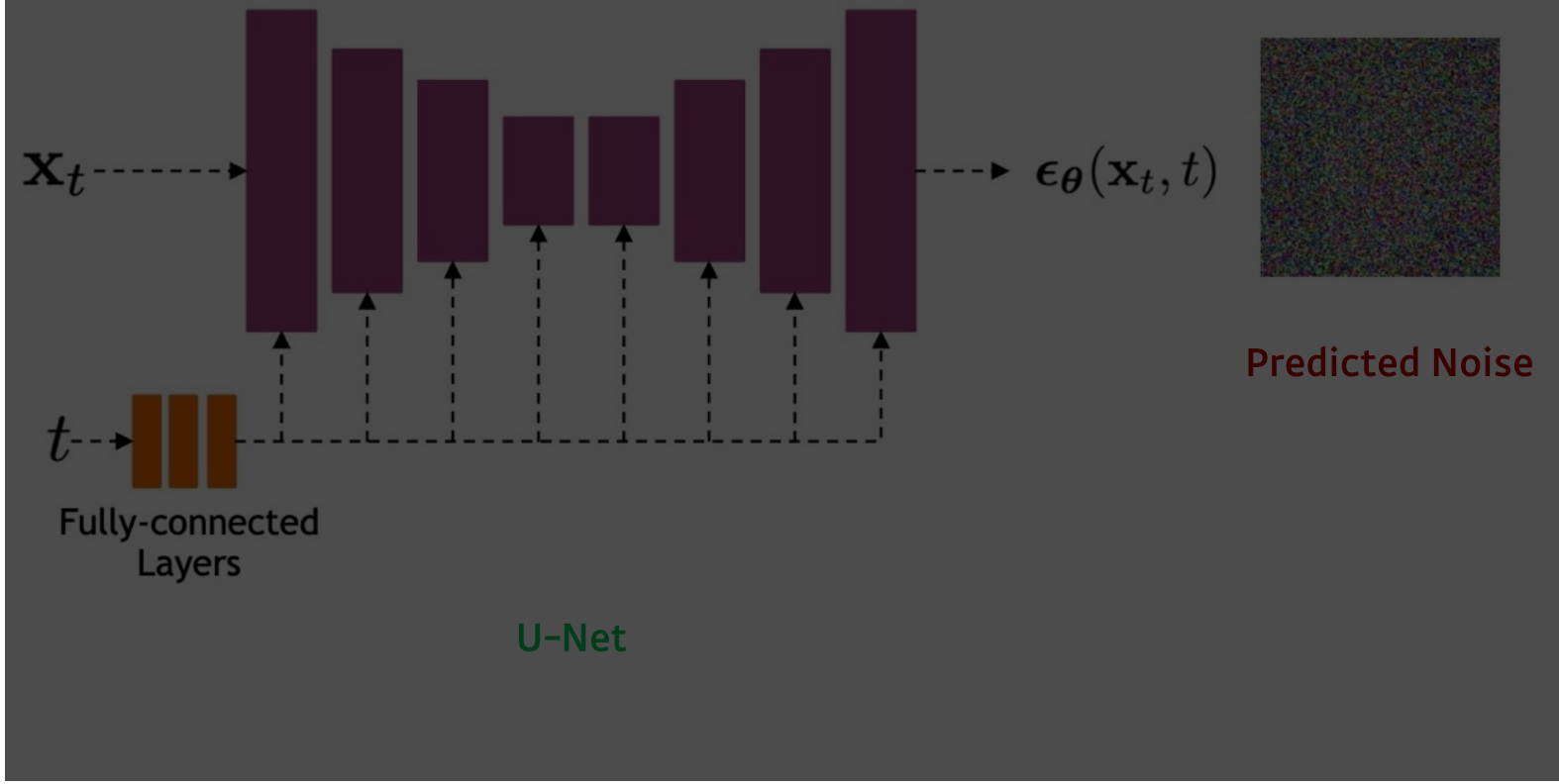
NAVER AI LAB

Forward

Image + noise

Timestep

Time Representation




Input

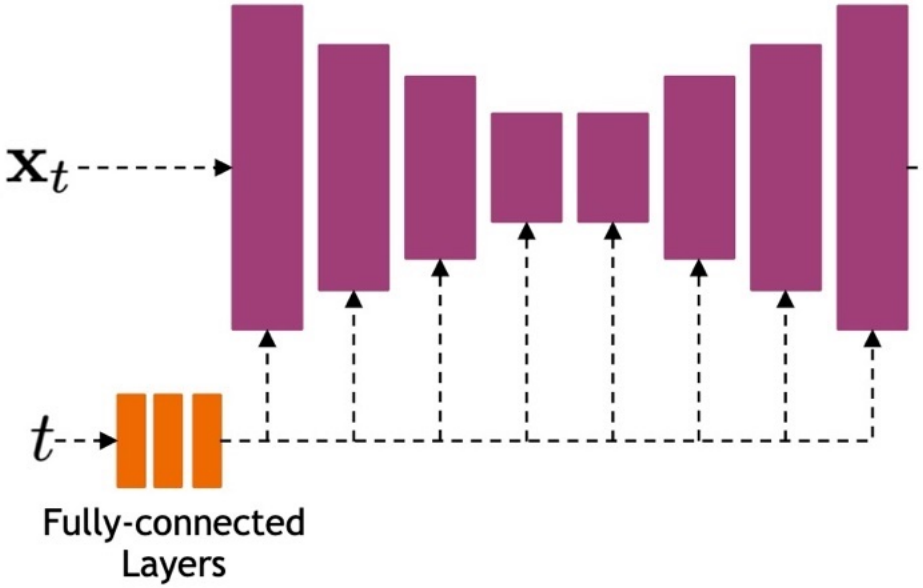
NAVER AI LAB

Forward

Image + noise

Timestep

Time Representation




Predicted Noise

Input

Network

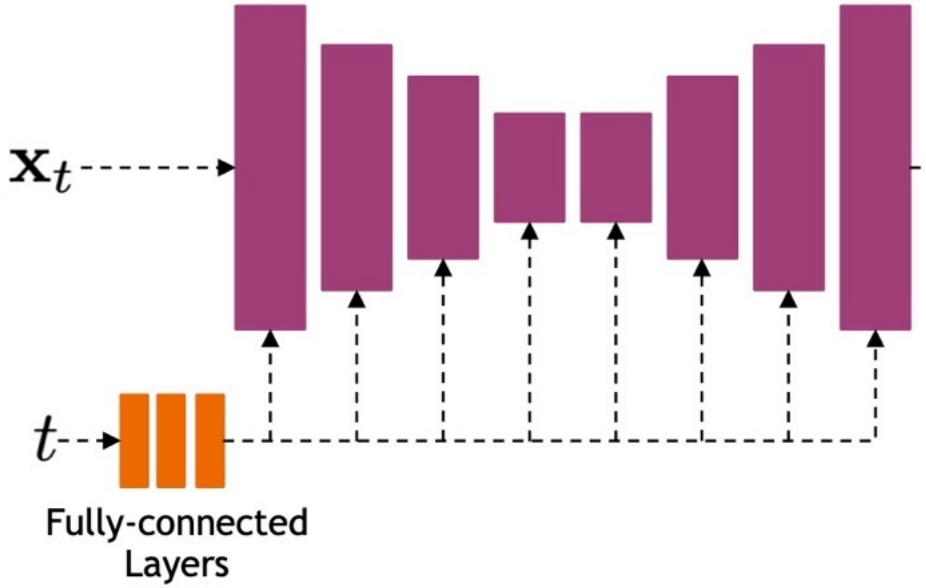
NAVER AI LAB

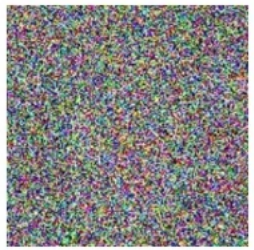
Forward

Image + noise

Timestep

Time Representation



Predicted Noise

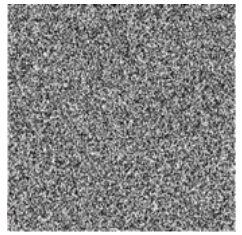
U-Net

Input

Network

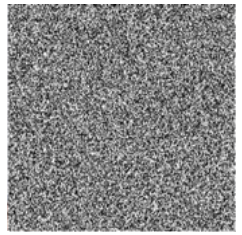
Output

NAVER AI LAB



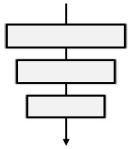
$x_T$

NAVER AI LAB

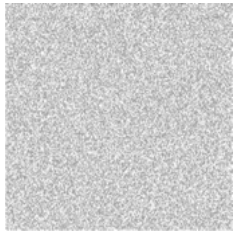


$x_T$

학습한 unet

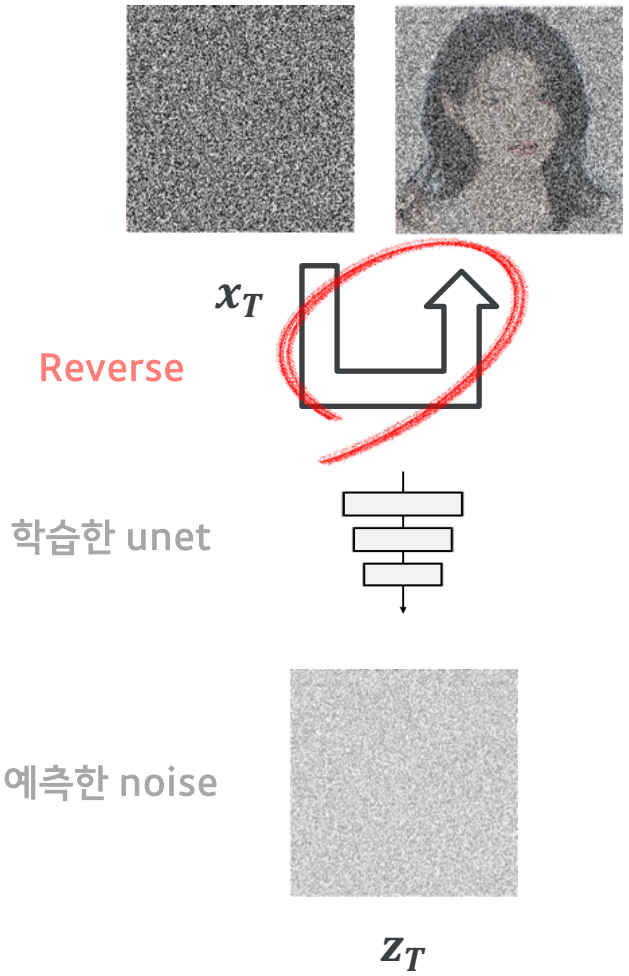


예측한 noise

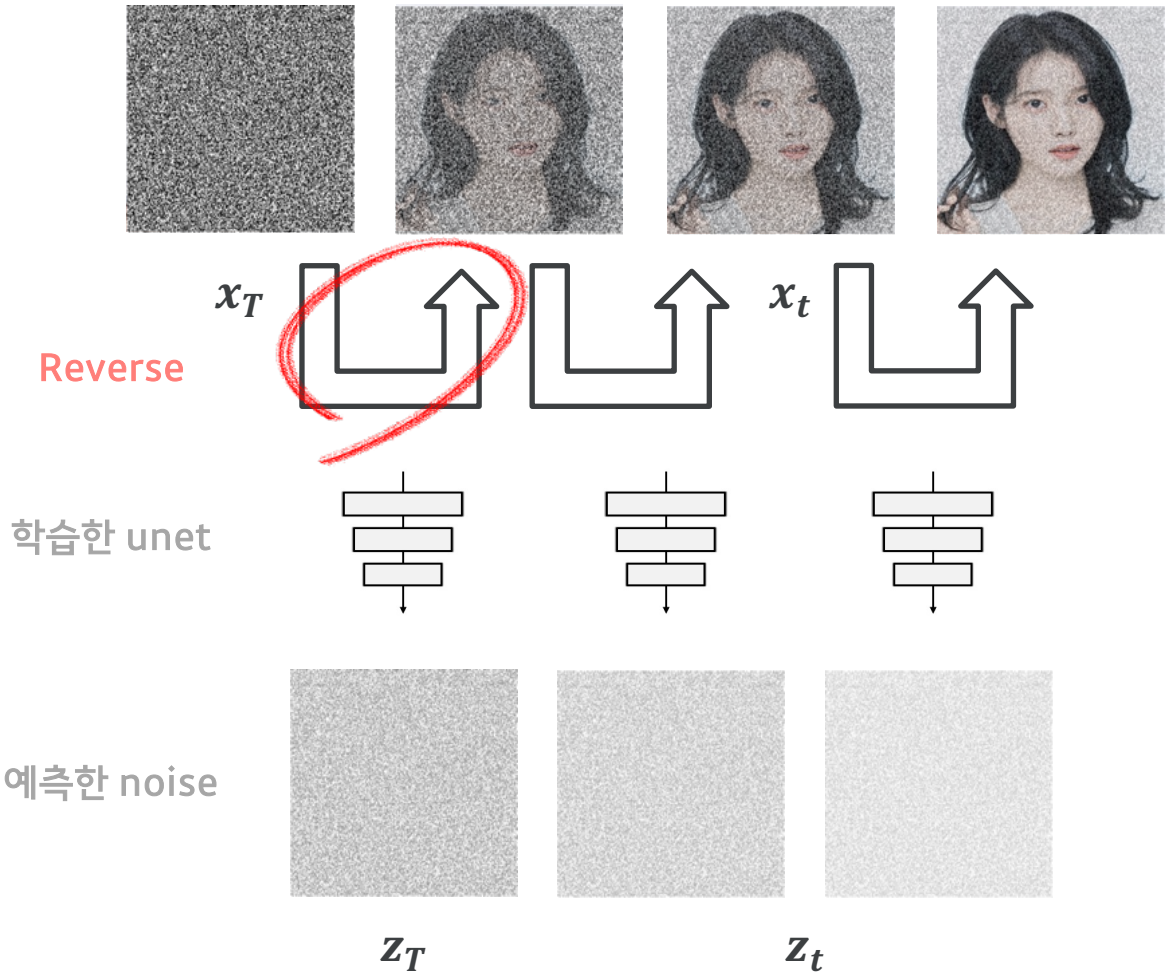


$z_T$

NAVER AI LAB

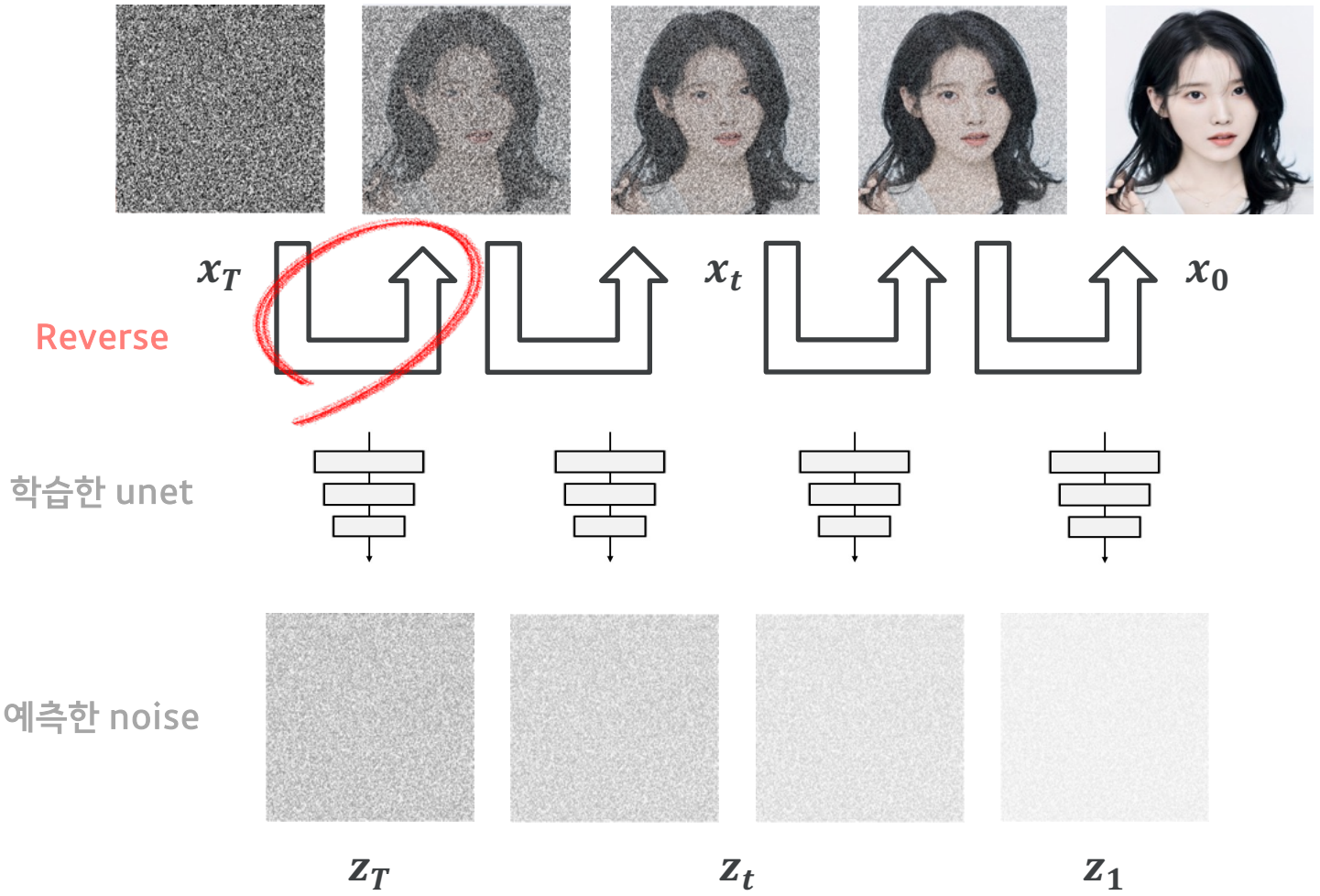


NAVER AI LAB



NAVER AI LAB

NAVER AI LAB




reverse process 종류

DDPM

DDIM

NAVER AI LAB

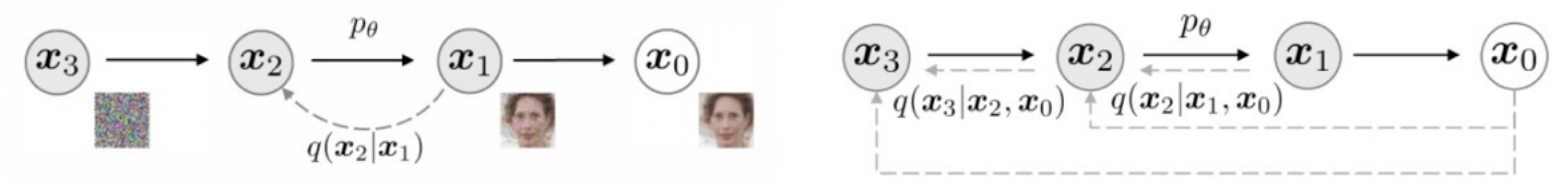



Figure 1: Graphical models for diffusion (left) and non-Markovian (right) inference models.

Denoising Diffusion Implicit Models

## DDPM

- $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon,\ \epsilon \sim \mathcal{N}(0, \mathbf{I})$ `(Forward)`
  - $\beta_1 = 10^{-4}, \beta_T = 0.02$
  - $\alpha_t := 1 - \beta_t,\ \bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$
- $\epsilon - \epsilon_\theta(\mathbf{x}_t) = \epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon)$ `(Loss)`
  - $\epsilon_\theta$ = prediction network
- $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\mathbf{x}_t, t)\right) + \sqrt{\tilde{\beta}_t}\epsilon,\ \epsilon \sim \mathcal{N}(0, \mathbf{I})$ `(Reverse)`
  - $\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$
    - $\tilde{\beta}_t = \beta_t$로 해도 성능차이 없음

## DDIM

- `In paper,` DDIM $\alpha$ = DDPM $\bar{\alpha}$
- $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$ `(Forward)`
- $\epsilon - \epsilon_\theta(\mathbf{x}_t) = \epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon)$ `(Loss)`
  - $\epsilon_\theta$ = prediction network
- $\mathbf{x}_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\underbrace{\left(\frac{\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\epsilon_\theta(\mathbf{x}_t)}{\sqrt{\bar{\alpha}_t}}\right)}_{\text{predicted } \mathbf{x}_0 = f_\theta(\mathbf{x}_t)} + \underbrace{\sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\cdot\epsilon_\theta(\mathbf{x}_t)}_{\text{direction pointing to } \mathbf{x}_t} + \underbrace{\sigma_t\epsilon}_{\text{noise}}$

  `(Reverse)`
  - deterministic when $\sigma_t = 0$ → consistency (DDIM)
  - stochastic when $\sigma_t = 1$ → inconsistency (DDPM)

NAVER AI LAB

- Prob & Stats

- Distribution

- Advanced

NAVER AI LAB

- Prob & Stats
  - $X = N(\mu, \Sigma)$ mean, variance
    - $\varepsilon \sim N(0, I)$
  - $E(aX + b) = aE(x) + b$
  - $V(aX + b) = a^2V(X)$
  - $\sigma(aX + b) = |a|\sigma(X)$ standard deviation = var^2
  - $P(B \mid A) = P(A \mid B)\frac{P(B)}{P(A)}$
- Distribution
- Advanced

NAVER AI LAB

- Prob & Stats
  - $X = N(\mu, \Sigma)$ mean, variance
    - $\varepsilon \sim N(0, I)$
  - $E(aX + b) = aE(x) + b$
  - $V(aX + b) = a^2 V(X)$
  - $\sigma(aX + b) = |a|\sigma(X)$ standard deviation = var^2
  - $P(B \mid A) = P(A \mid B)\frac{P(B)}{P(A)}$

- Distribution
  - $q(x)$: $real\ distribution$
  - $p_\theta(x)$: $network\ distribution$

- Advanced

- Prob & Stats
  - $X = N(\mu, \Sigma)$ mean, variance
    - $\varepsilon \sim N(0, I)$
  - $E(aX + b) = aE(x) + b$
  - $V(aX + b) = a^2V(X)$
  - $\sigma(aX + b) = |a|\sigma(X)$ standard deviation = var^2
  - $P(B \mid A) = P(A \mid B)\frac{P(B)}{P(A)}$

- Distribution
  - $q(x)$: $real\ distribution$
  - $p_\theta(x)$: $network\ distribution$

- Advanced
  - $N(x;\ \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} = f(x)$ probability density function (pdf)
  - $T_f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n = f(a) + f'(a)(x-a) + \frac{1}{2}f''(a)(x-a)^2 + \ \ldots$
    - $\approx f(a) + f'(a)(x-a)$

- Prob & Stats
  - $X = N(\mu, \Sigma)$ mean, variance
    - $\varepsilon \sim N(0, I)$
  - $E(aX + b) = aE(x) + b$
  - $V(aX + b) = a^2V(X)$
  - $\sigma(aX + b) = |a|\sigma(X)$ standard deviation = var^2
  - $P(B \mid A) = P(A \mid B)\frac{P(B)}{P(A)}$

- Distribution
  - $q(x)$: $real\ distribution$
  - $p_\theta(x)$: $network\ distribution$

- Advanced
  - $N(x;\ \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} = f(x)$ probability density function (pdf)
  - $T_f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n = f(a) + f'(a)(x-a) + \frac{1}{2}f''(a)(x-a)^2 + \ \ldots$
    - $\approx f(a) + f'(a)(x-a)$

# Goal

Read a paper

NAVER AI LAB

### 3.1 DDPM Sampling with Manifold Constraint

In DDPMs (Ho et al., 2020), starting from a clean image $\boldsymbol{x}_0 \sim q(\boldsymbol{x}_0)$, a forward diffusion process $q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$ is described as a Markov chain that gradually adds Gaussian noise at every time steps $t$:

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

where $\{\beta\}_{t=0}^{T}$ is a variance schedule. By denoting $\alpha_t := 1 - \beta_t$ and $\bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$, the forward diffused sample at $t$, i.e. $\boldsymbol{x}_t$, can be sampled in one step as:

$$\boldsymbol{x}_t = \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, \quad \text{where} \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}). \tag{2}$$

As the reverse of the forward step $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ is intractable, DDPM learns to maximize the variational lowerbound through a parameterized Gaussian transitions $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ with the parameter $\theta$. Accordingly, the reverse process is approximated as Markov chain with learned mean and fixed variance, starting from $p(\boldsymbol{x}_T) = \mathcal{N}(\boldsymbol{x}_T; \boldsymbol{0}, \boldsymbol{I})$:

$$p_\theta(\boldsymbol{x}_{0:T}) := p_\theta(\boldsymbol{x}_T) \prod_{t=1}^{T} p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t), \quad \text{where} \quad p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) := \mathcal{N}(\boldsymbol{x}_{t-1}; \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t), \sigma_t^2 \boldsymbol{I}). \tag{3}$$

where

$$\boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) := \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\right), \tag{4}$$

Here, $\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)$ is the diffusion model trained by optimizing the objective:

$$\min_\theta L(\theta), \quad \text{where} \quad L(\theta) := \mathbb{E}_{t, \boldsymbol{x}_0, \boldsymbol{\epsilon}}\left[\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)\|^2\right]. \tag{5}$$

After the optimization, by plugging learned score function into the generative (or reverse) diffusion process, one can simply sample from $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ by

$$\boldsymbol{x}_{t-1} = \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) + \sigma_t \boldsymbol{\epsilon} = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\right) + \sigma_t \boldsymbol{\epsilon} \tag{6}$$

In image translation using *conditional* diffusion models (Saharia et al., 2022a; Sasaki et al., 2021), the diffusion model $\boldsymbol{\epsilon}_\theta$ in (5) and (6) should be replaced with $\boldsymbol{\epsilon}_\theta(\boldsymbol{y}, \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)$ where $\boldsymbol{y}$ denotes the matched target image. Accordingly, the sample generation is tightly controlled by the matched target in a supervised manner, so that the image content change rarely happen. Unfortunately, the requirement of the *matched* targets for the training makes this approach impractical.

To address this, Dhariwal & Nichol (2021) proposed classifier-guided image translation using the unconditional diffusion model training as in (5) and a pre-trained classifier $p_\phi(\boldsymbol{y}|\boldsymbol{x}_t)$. Specifically, $\boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t)$ in (4) and (6) are supplemented with the gradient of the classifier, i.e. $\hat{\boldsymbol{\mu}}_\theta(\boldsymbol{x}_t, t) := \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) + \sigma_t \nabla_{\boldsymbol{x}_t} \log p_\phi(\boldsymbol{y}|\boldsymbol{x}_t)$. However, most of the classifiers, which should be separately trained, are not usually sufficient to control the content of the samples from the reverse diffusion process.

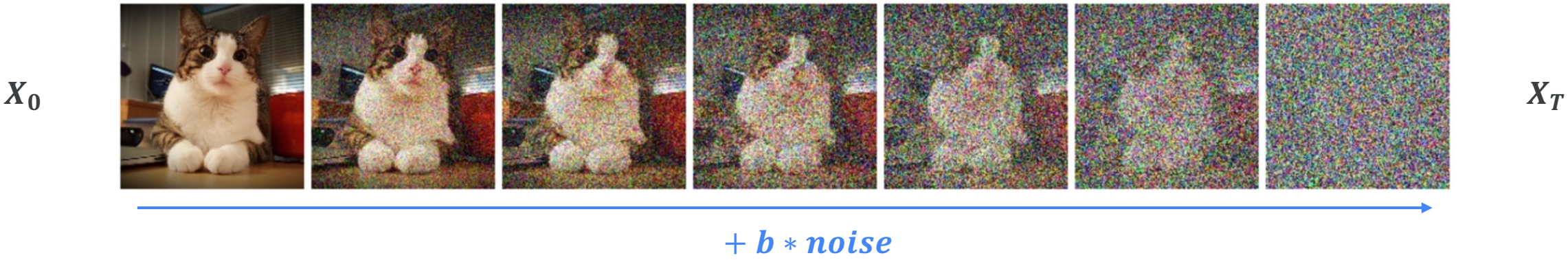
Diffusion-based Image Translation using Disentangled Style and Content Representation, arXiv

# Forward process

# Diffusion | Forward process

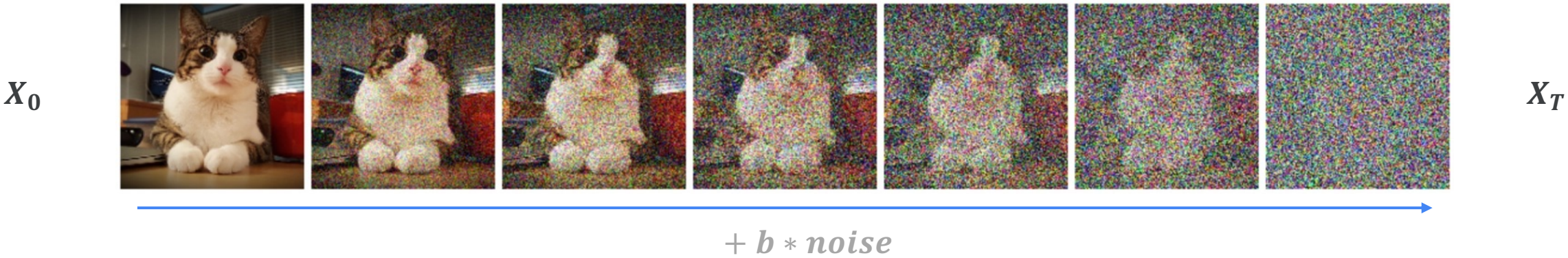
NAVER AI LAB

- Overview (mean, variance)
  - $q(X_t \mid X_{t-1}) = N(X_t;\ \mu_{X_{t-1}},\ \Sigma_{X_{t-1}})$
    - Conditional gaussian distribution



NAVER AI LAB

- Overview mean variance
  - $q(X_t \mid X_{t-1}) = N(X_t;\ \mu_{X_{t-1}},\ \Sigma_{X_{t-1}})$
    - Conditional gaussian distribution




$$x_t = a * x_{t-1} + b * noise$$

# Diffusion | Forward process

NAVER AI LAB

- Overview
  - $q(X_t \mid X_{t-1}) = N(X_t;\ \mu_{X_{t-1}},\ \Sigma_{X_{t-1}}) = N(X_t; \sqrt{1-\beta_t}X_{t-1}, \beta_t * I)$ (mean, variance)
    - Conditional gaussian distribution
    - $0 < \beta_1 < \beta_2 < \ ... < \beta_T < 1$
    - 0.0001 ~ 0.02

Note

$Var(aX) = a^2 Var(X)$


$$+ b * noise$$

$$x_t = a * x_{t-1} + b * noise$$

$$\boldsymbol{x_t = \sqrt{1-\beta_t} * x_{t-1} + \sqrt{\beta_t} * noise}$$

# Diffusion | Forward process

NAVER AI LAB

- Overview
  - $q(X_t \mid X_{t-1}) = N(X_t;\ \mu_{X_{t-1}},\ \Sigma_{X_{t-1}}) = N(X_t; \sqrt{1-\beta_t}X_{t-1}, \beta_t * I)$ (mean, variance)
    - Conditional gaussian distribution
    - $0 < \beta_1 < \beta_2 < \ ... < \beta_T < 1$
    - 0.0001 ~ 0.02
  - $x_t = \sqrt{1-\beta_t}x_{t-1} + \sqrt{\beta_t}\varepsilon$ (Reparameterization trick) x = mean + std * noise
    - $\varepsilon \sim N(0, I)$

Note

$Var(aX) = a^2 Var(X)$

$X_0$ $X_T$

$+ \beta * noise$

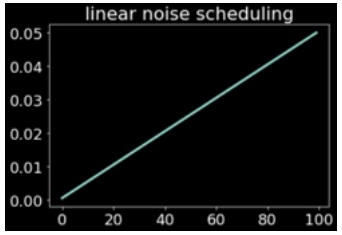
$x_t = a * x_{t-1} + b * noise$

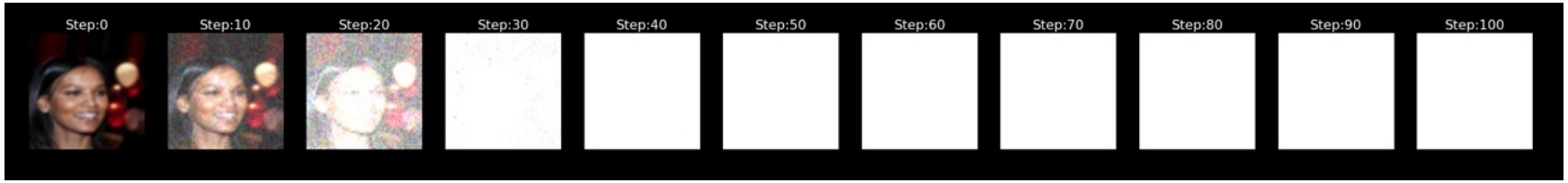
$$x_t = \sqrt{1-\beta_t} * x_{t-1} + \sqrt{\beta_t} * noise$$

NAVER AI LAB

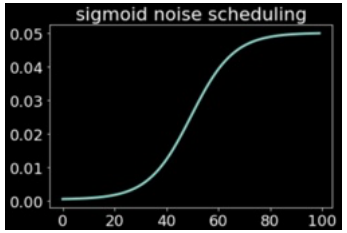
- Overview
  - Linear, Quad, Sigmoid, Cosine, ...

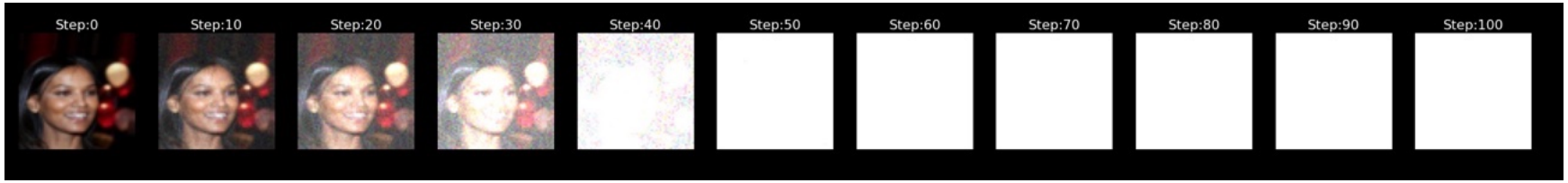
✓ Linear scheduling



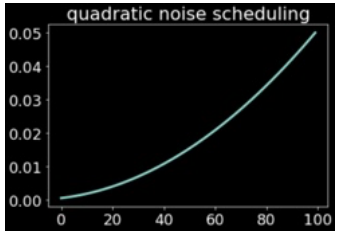



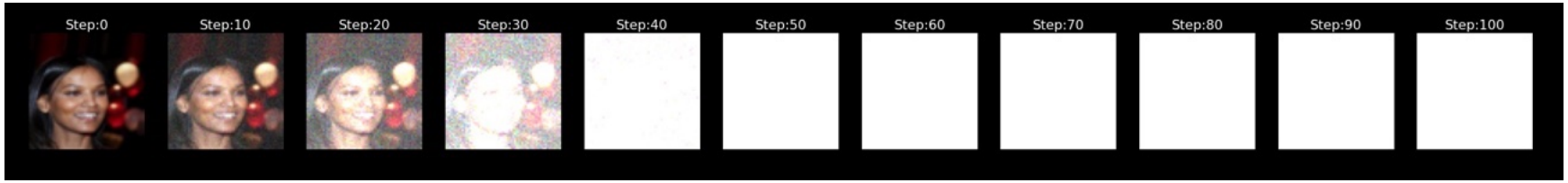

✓ Sigmoid scheduling




✓ Quadratic scheduling




www.youtube.com/watch?v=_JQSMhqXw-4

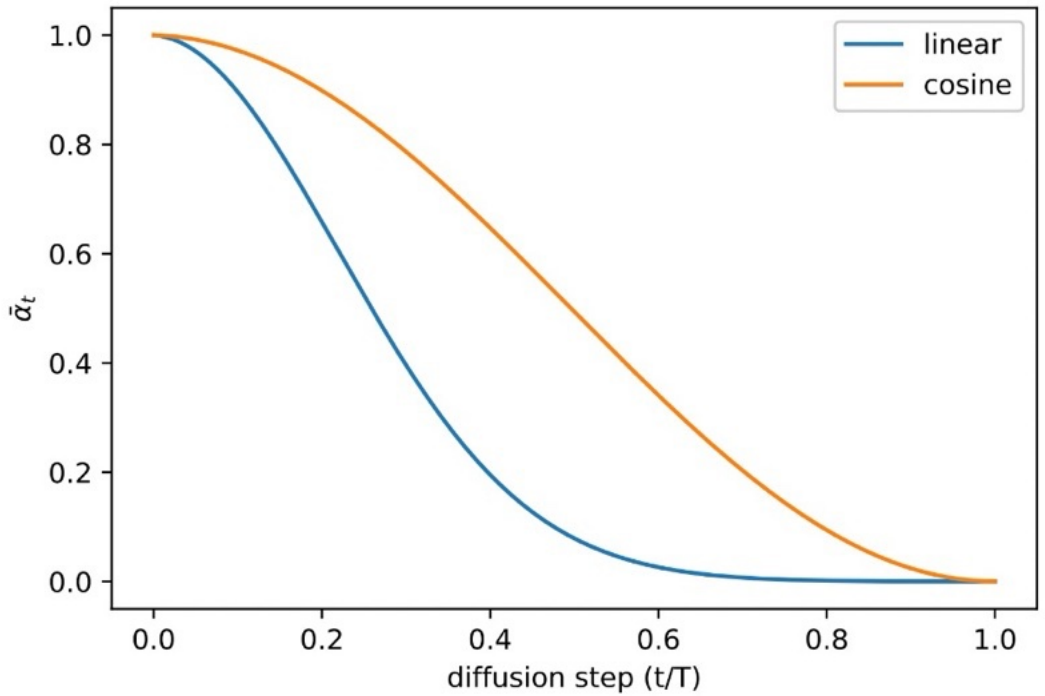
NAVER AI LAB

- Overview
  - Linear, Quad, Sigmoid, Cosine, ...

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

where the small offset $s$ is to prevent $\beta_t$ from being too small when close to $t = 0$.

Linear가 너무 빨리 정보 지워버림




Improved Denoising Diffusion Probabilistic Models, ICML 2021

# Diffusion | Forward process $(x_t, x_0)$

NAVER AI LAB

- Overview
  - $q(x_t \mid x_{t-1}) = N\left(x_t; \sqrt{1-\beta_t} x_{t-1}, \beta_t * I\right)$

# Diffusion | Forward process $(x_t, x_0)$

NAVER AI LAB

- Overview
  - $q(x_t \mid x_{t-1}) = N\left(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t * I\right)$
  - $q(x_t \mid x_0) = N\left(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t) * I\right)$ x0로부터 어떤 xt든 한번에 만들자
    - $\alpha_t = 1 - \beta_t$
    - $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$

# Diffusion | Forward process $(x_t, x_0)$

NAVER AI LAB

- Overview
  - $q(x_t \mid x_{t-1}) = N\left(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t * I\right)$
  - $q(x_t \mid x_0) = N\left(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t) * I\right)$ x0로부터 어떤 xt든 한번에 만들자
    - $\alpha_t = 1 - \beta_t$
    - $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$
  - $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon$

Note

$0.0001 < \beta_t < 0.02$

- linear, cosine ...

Note

$$
\begin{aligned}
x_t &= \sqrt{\alpha_t}x_{t-1} + \sqrt{1-\alpha_t}z_{t-1} \\
&= \sqrt{\alpha_t\alpha_{t-1}}x_{t-2} + \sqrt{\alpha_t}\sqrt{1-\alpha_{t-1}}z_{t-2} + \sqrt{1-\alpha_t}z_{t-1} \\
&= \sqrt{\alpha_t\alpha_{t-1}}x_{t-2} + \sqrt{1-\alpha_t\alpha_{t-1}}\bar{z}_{t-2} \dots (*) \\
&= \cdots \\
&= \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}z
\end{aligned}
$$

(*) $X \sim \mathcal{N}(\mu_X, \sigma_X^2)$와 $Y \sim \mathcal{N}(\mu_Y, \sigma_Y^2)$에서 $Z = X + Y$는 $Z \sim \mathcal{N}(\mu_X + \mu_Y, \sigma_X^2 + \sigma_Y^2)$입니다.

lilianweng.github.io/posts/2021-07-11-diffusion-models

NAVER AI LAB

Forward process

이건 학습없이 가능함

Diffusion process

$q(X_t \mid X_{t-1})$

+ noise

$X_0$

$X_T$

- noise

$q(X_{t-1} \mid X_t)$

학습을 통해
가능하게 만들자

Denoising process

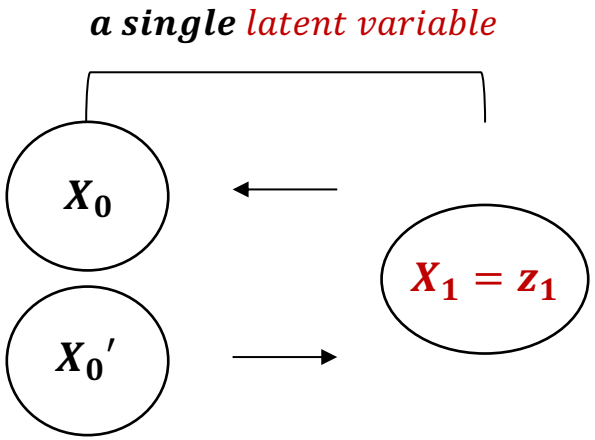
Reverse process

# Objective for Reverse process

# Diffusion | Objective for Reverse process

NAVER AI LAB

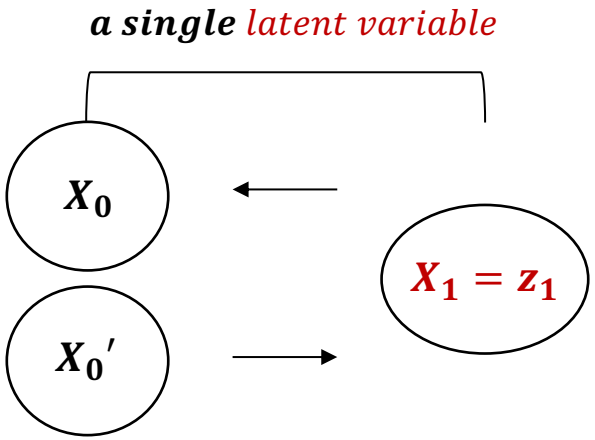
- ✓ VAE



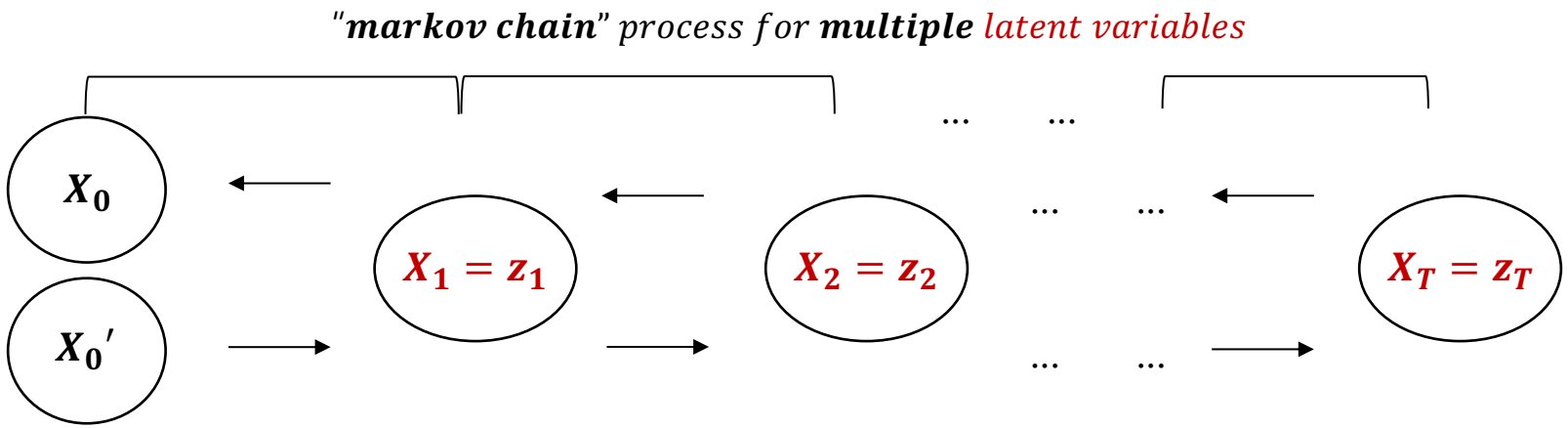
# Diffusion | Objective for Reverse process

NAVER AI LAB

✓ VAE



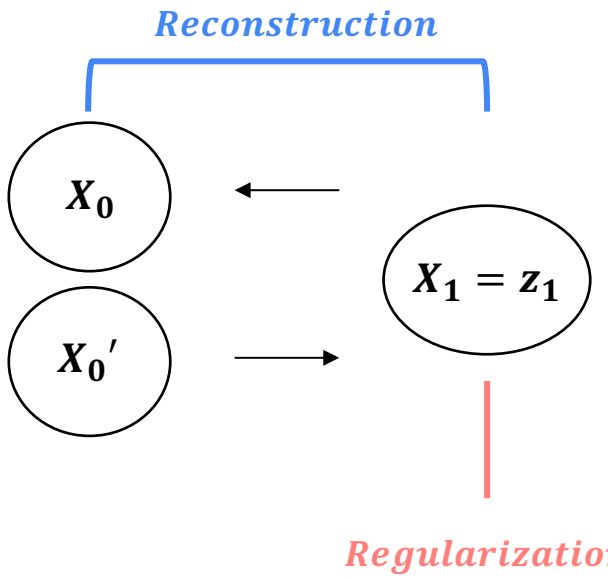

✓ Diffusion




www.youtube.com/watch?v=_JQSMhqXw-4

NAVER AI LAB

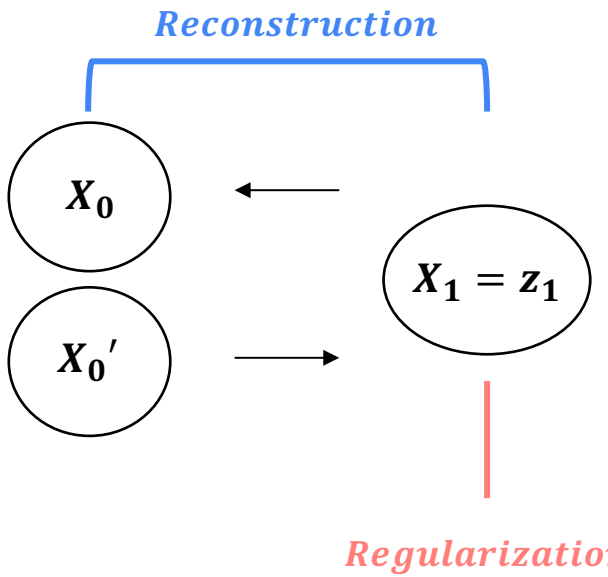
✓ VAE




*Regularizer on Encoder* *Reconstruction on Decoder*

$$Loss_{VAE} = D_{KL}\big(q(z|x) \parallel p_\theta(z)\big) - E_{z\sim q(z|x)} \log P_\theta(x|z)$$

✓ VAE




## Maximize $P_\theta(x)$

Variational autoencoder

$$\mathbb{E}_{x_T \sim q(x_T|x_0)}[-\log p_\theta(x_0)]$$

$$= \int_{-\infty}^{\infty} (-\log p_\theta(x_0)) \cdot q(x_T|x_0) dx_T \quad \because \text{definition of expectation}$$

$$= \int_{-\infty}^{\infty} \left(-\log \frac{p_\theta(x_0, x_T)}{p_\theta(x_T|x_0)}\right) \cdot q(x_T|x_0) dx_T \quad \because \text{bayes rule}, p_\theta(x_T|x_0) = \frac{p_\theta(x_T, x_0)}{p_\theta(x_0)}$$

$$= \int_{-\infty}^{\infty} (-\log \frac{p_\theta(x_0, x_T)}{p_\theta(x_T|x_0)} \cdot \frac{q(x_T|x_0)}{q(x_T|x_0)}) \cdot q(x_T|x_0) dx_T$$

$$\leq \int_{-\infty}^{\infty} \left(-\log \frac{p_\theta(x_0, x_T)}{q(x_T|x_0)}\right) \cdot q(x_T|x_0) dx_T \quad \because KL\ divergence > 0, \text{"ELBO"}$$

$$= \int_{-\infty}^{\infty} \left(-\log \frac{p_\theta(x_0|x_T) \cdot p_\theta(x_T)}{q(x_T|x_0)}\right) \cdot q(x_T|x_0) dx_T \quad \because bayes\ rule, p_\theta(x_0, x_T) = p_\theta(x_0|x_T) p_\theta(x_T)$$

$$= \int_{-\infty}^{\infty} (-\log p_\theta(x_0|x_T)) \cdot q(x_T|x_0) dx_T + \int_{-\infty}^{\infty} (-\log \frac{p_\theta(x_T)}{q(x_T|x_0)}) \cdot q(x_T|x_0) dx_T \quad \because separate\ \log$$

$$= \mathbb{E}_{x_T \sim q(x_T|x_0)}[-\log p_\theta(x_0|x_T)] + \mathbb{E}_{x_T \sim q(x_T|x_0)}[-\log \frac{p_\theta(x_T)}{q(x_T|x_0)}] \quad \because definition\ of\ expectation$$

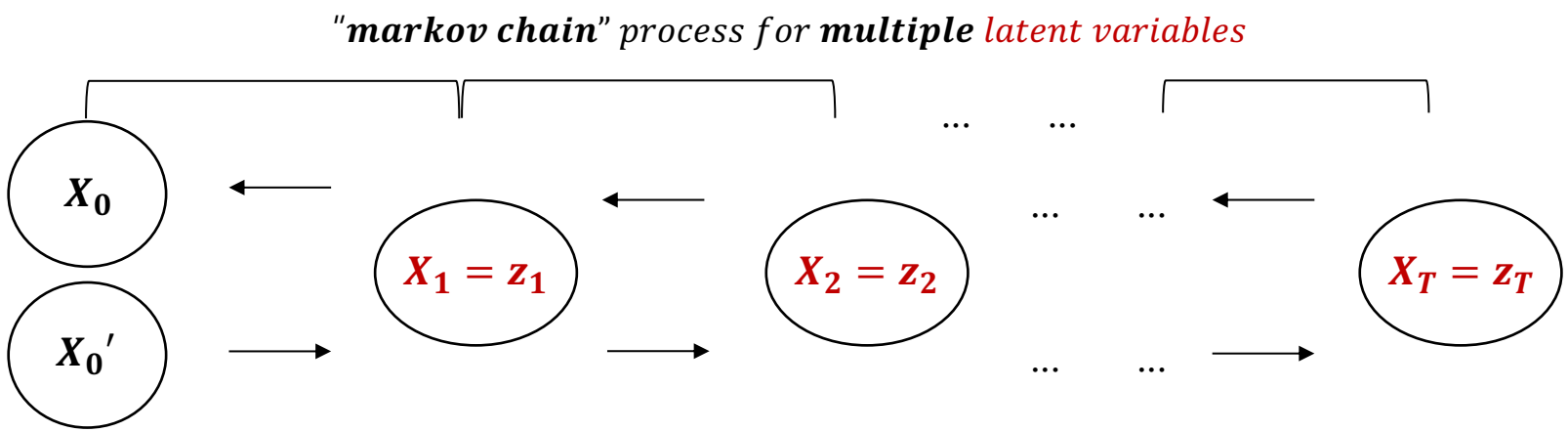
위 수식을 통해 loss가 나오긴함 ...

*Regularizer on Encoder* *Reconstruction on Decoder*

$$Loss_{VAE} = D_{KL}\big(q(z|x) \parallel p_\theta(z)\big) - E_{z \sim q(z|x)} \log P_\theta(x|z)$$

NAVER AI LAB

- ✓ **Diffusion**



NAVER AI LAB

## Maximize $P_\theta(x)$

- ✓ **Diffusion**

$$\mathbb{E}_{x_T\sim q(x_T|x_0)}[-\log p_\theta(x_0)]$$

① $= \mathbb{E}_{x_T\sim q(x_T|x_0)}\left[-\log \frac{p_\theta(x_0,x_1,x_2,\dots,x_T)}{p_\theta(x_1,x_2,x_3,\dots,x_T|x_0)}\right]$ ∵ bayes rule, $p_\theta(x_T|x_0)=\frac{p_\theta(x_T,x_0)}{p_\theta(x_0)}$

② $= \mathbb{E}_{x_T\sim q(x_T|x_0)}\left[-\log \frac{p_\theta(x_0,x_1,x_2,\dots,x_T)}{p_\theta(x_1,x_2,x_3,\dots,x_T|x_0)}\cdot\frac{q(x_{1:T}|x_0)}{q(x_{1:T}|x_0)}\right]$

③ $\leq \mathbb{E}_{x_T\sim q(x_T|x_0)}\left[-\log \frac{p_\theta(x_0,x_1,x_2,\dots,x_T)}{q(x_{1:T}|x_0)}\right]$ ∵ *KL divergence* $> 0$, "**ELBO**"

④ $= \mathbb{E}_{x_T\sim q(x_T|x_0)}\left[-\log \frac{p_\theta(x_{0:T})}{q(x_{1:T}|x_0)}\right]$ ∵ *Notation*

⑤ $= \mathbb{E}_{x_T\sim q(x_T|x_0)}\left[-\log \frac{p_\theta(\boldsymbol{x_T})\prod_{t=1}^{T}p_\theta(\boldsymbol{x_{t-1}}|\boldsymbol{x_t})}{\prod_{t=1}^{T}q(\boldsymbol{x_t}|\boldsymbol{x_{t-1}})}\right]$ ∵ *Below Markov chain property*

⑥ $= \mathbb{E}_{x_{1:T}\sim q(x_{1:T}|x_0)}\left[-\log p_\theta(\boldsymbol{x_T})-\sum_{t=1}^{T}\log\frac{p_\theta(x_{t-1}|x_t)}{q(x_t|x_{t-1})}\right]$ ∵ *separating to summation in logarithm*

$$\mathbb{E}_{x_{1:T}\sim q(x_{1:T}|x_0)}[-\log p_\theta(x_0)]$$

⑦ $\leq \mathbb{E}_{x_{1:T}\sim q(x_{1:T}|x_0)}\left[-\log p_\theta(\boldsymbol{x_T})-\sum_{t=1}^{T}\log\frac{p_\theta(x_{t-1}|x_t)}{q(x_t|x_{t-1})}\right]$

⑧ $= \mathbb{E}_{x_{1:T}\sim q(x_{1:T}|x_0)}\left[-\log p_\theta(\boldsymbol{x_T})-\sum_{t=2}^{T}\log\frac{p_\theta(x_{t-1}|x_t)}{q(x_t|x_{t-1})}-\log\frac{p_\theta(x_0|x_1)}{q(x_1|x_0)}\right]$

⑨ $= \mathbb{E}_{x_{1:T}\sim q(x_{1:T}|x_0)}\left[-\log p_\theta(\boldsymbol{x_T})-\sum_{t=2}^{T}\log\frac{p_\theta(x_{t-1}|x_t)}{q(x_{t-1}|x_t,x_0)}\cdot\frac{q(x_{t-1}|x_0)}{q(x_t|x_0)}-\log\frac{p_\theta(x_0|x_1)}{q(x_1|x_0)}\right]$ ∵*

⑩ $= \mathbb{E}_{x_{1:T}\sim q(x_{1:T}|x_0)}\left[-\log p_\theta(\boldsymbol{x_T})-\sum_{t=2}^{T}\log\frac{p_\theta(x_{t-1}|x_t)}{q(x_{t-1}|x_t,x_0)}-\sum_{t=2}^{T}\log\frac{q(x_{t-1}|x_0)}{q(x_t|x_0)}-\log\frac{p_\theta(x_0|x_1)}{q(x_1|x_0)}\right]$

⑪ $= \mathbb{E}_{x_{1:T}\sim q(x_{1:T}|x_0)}\left[-\log p_\theta(\boldsymbol{x_T})-\sum_{t=2}^{T}\log\frac{p_\theta(x_{t-1}|x_t)}{q(x_{t-1}|x_t,x_0)}-\log\frac{q(x_1|x_0)}{q(x_T|x_0)}-\log\frac{p_\theta(x_0|x_1)}{q(x_1|x_0)}\right]$

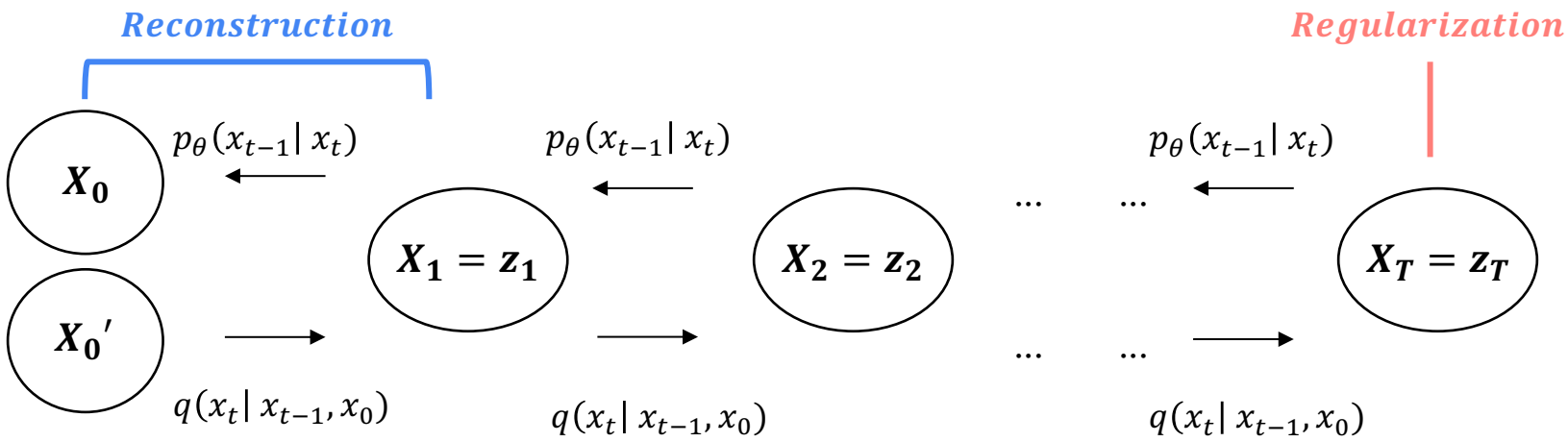
⑫ $= \mathbb{E}_{x_{1:T}\sim q(x_{1:T}|x_0)}\left[-\log\frac{p_\theta(\boldsymbol{x_T})}{q(x_T|x_0)}-\sum_{t=2}^{T}\log\frac{p_\theta(x_{t-1}|x_t)}{q(x_{t-1}|x_t,x_0)}-\log p_\theta(x_0|x_1)\right]$

$$p_\theta(\boldsymbol{x_{0:T}}) \coloneqq p_\theta(\boldsymbol{x_T})\prod_{t=1}^{T}p_\theta(\boldsymbol{x_{t-1}}|\boldsymbol{x_t}) \qquad q(\boldsymbol{x_{1:T}}|\boldsymbol{x_0}) \coloneqq \prod_{t=1}^{T}q(\boldsymbol{x_t}|\boldsymbol{x_{t-1}})$$

$$\begin{aligned}
* \; q(x_t|x_{t-1}) &= q(x_t|x_{t-1},x_0) && \because \text{Markov chain property}\\
&= \frac{q(x_t,x_{t-1},x_0)}{q(x_{t-1},x_0)} && \because \text{bayes rule}\\
&= \frac{q(x_{t-1},x_t,x_0)}{q(x_{t-1},x_0)}\cdot\frac{q(x_t,x_0)}{q(x_t,x_0)}\\
&= q(x_{t-1}|x_t,x_0)\cdot\frac{q(x_t,x_0)}{q(x_{t-1},x_0)}
\end{aligned}$$

NAVER AI LAB

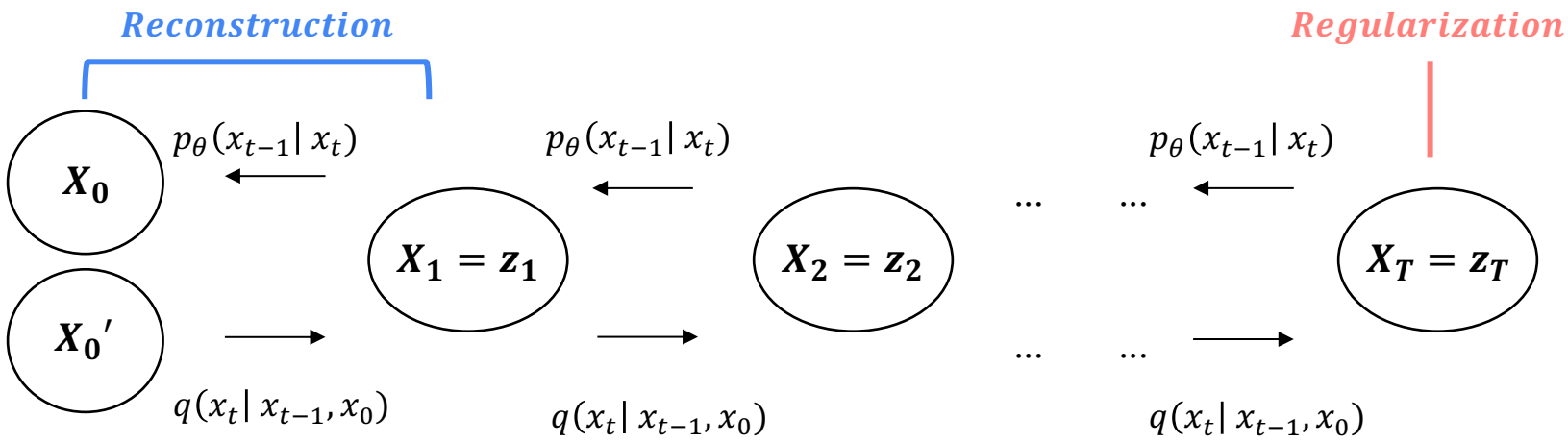
✓ **Diffusion**




$$Loss_{VAE} = \underbrace{D_{KL}\left(q(z|x) \parallel p_\theta(z)\right)}_{\text{Regularization}} - \underbrace{E_{z\sim q(z|x)} \log P_\theta(x|z)}_{\text{Reconstruction}}$$

NAVER AI LAB

✓ **Diffusion**

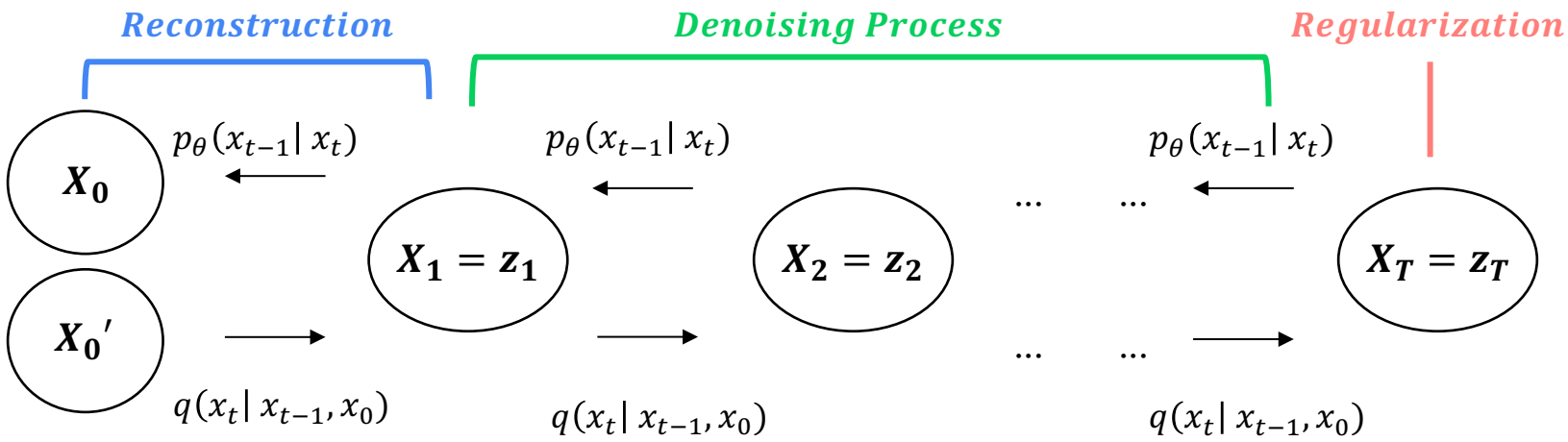



$$Loss_{VAE} = \underbrace{D_{KL}\big(q(z|x) \parallel p_\theta(z)\big)}_{\text{Regularization}} - \underbrace{E_{z\sim q(z|x)} \log P_\theta(x|z)}_{\text{Reconstruction}}$$

$$Loss_{Diffusion} = \underbrace{D_{KL}\big(q(x_T|x_0) \parallel p_\theta(x_T)\big)}_{\text{Regularization}} + \sum_{t=2} D_{KL}\big(q(x_{t-1}|x_t, x_0) \parallel P_\theta(x_{t-1}|x_t)\big) - \underbrace{E_q \log P_\theta(x_0|x_1)}_{\text{Reconstruction}}$$

www.youtube.com/watch?v=_JQSMhqXw-4

✓ **Diffusion**

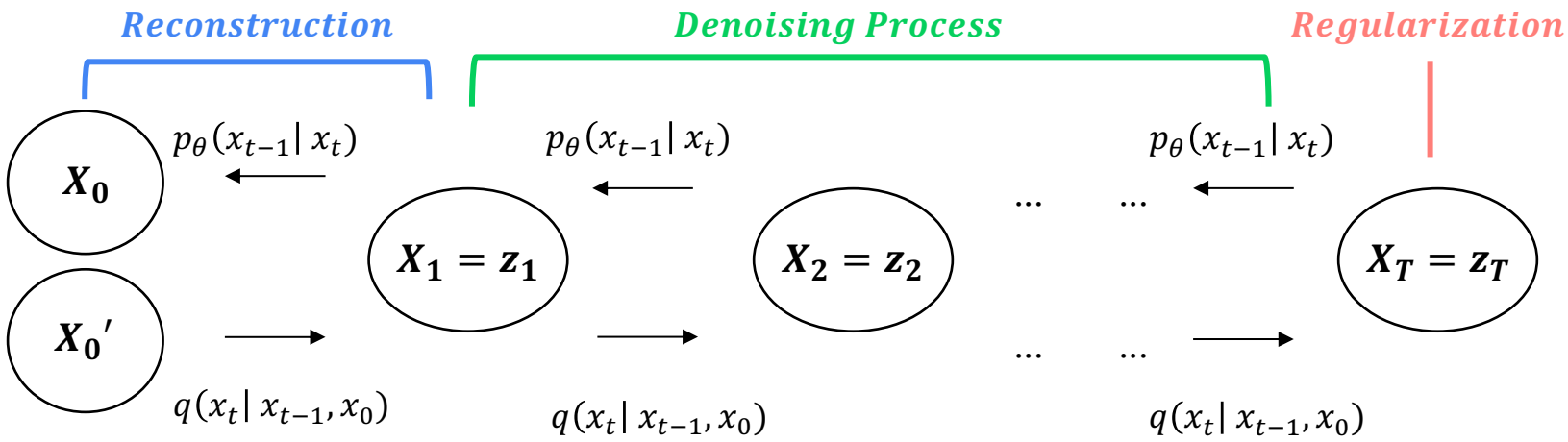



$$Loss_{VAE} = \underbrace{D_{KL}\big(q(z|x) \parallel p_\theta(z)\big)}_{\text{Regularization}} - \underbrace{E_{z\sim q(z|x)} \log P_\theta(x|z)}_{\text{Reconstruction}}$$

$$Loss_{Diffusion} = \underbrace{D_{KL}\big(q(x_T|x_0) \parallel p_\theta(x_T)\big)}_{\text{Regularization}} + \underbrace{\sum_{t=2} D_{KL}\big(q(x_{t-1}|x_t, x_0) \parallel P_\theta(x_{t-1}|x_t)\big)}_{\text{Denoising Process}} - \underbrace{E_q \log P_\theta(x_0|x_1)}_{\text{Reconstruction}}$$

NAVER AI LAB

✓ Diffusion



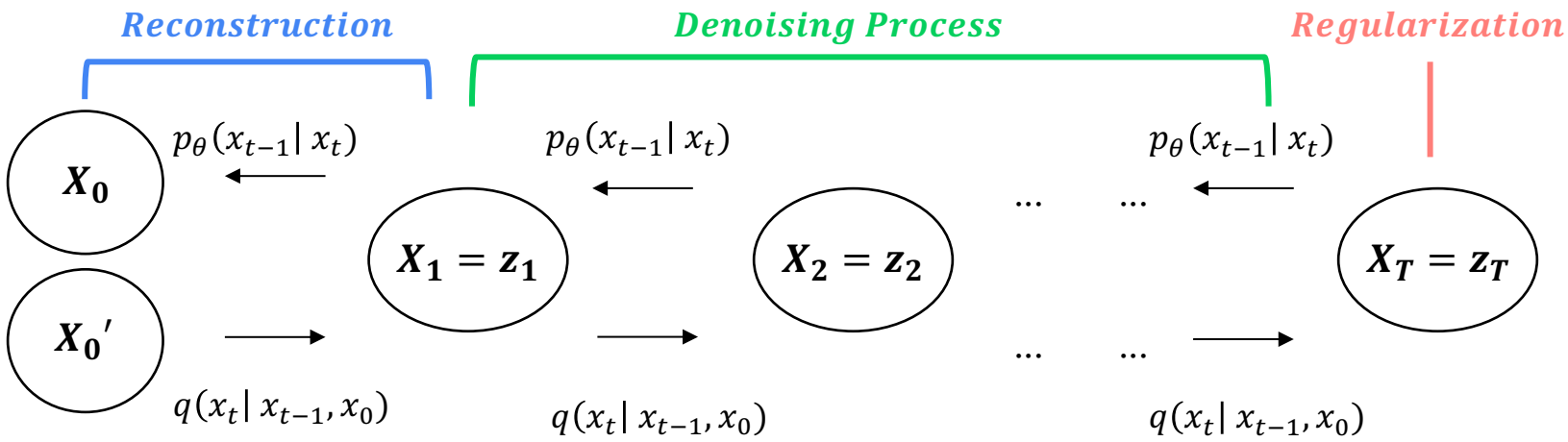

얘넨 왜 무시할까 ?

$$Loss_{VAE} = \underbrace{D_{KL}(q(z|x) \parallel p_\theta(z))}_{\text{Regularization}} - \underbrace{E_{z\sim q(z|x)} \log P_\theta(x|z)}_{\text{Reconstruction}}$$

$$Loss_{Diffusion} = \underbrace{D_{KL}(q(x_T|x_0) \parallel p_\theta(x_T))}_{\text{Regularization}} + \underbrace{\sum_{t=2} D_{KL}(q(x_{t-1}|x_t, x_0) \parallel P_\theta(x_{t-1}|x_t))}_{\text{Denoising Process}} - \underbrace{E_q \log P_\theta(x_0|x_1)}_{\text{Reconstruction}}$$

NAVER AI LAB

✓ Diffusion




ddpm, ddim 잘 알아두자

$$Loss_{VAE} = \underbrace{D_{KL}(q(z|x) \parallel p_\theta(z))}_{\text{Regularization}} - \underbrace{E_{z\sim q(z|x)} \log P_\theta(x|z)}_{\text{Reconstruction}}$$

$$Loss_{Diffusion} = \underbrace{D_{KL}(q(x_T|x_0) \parallel p_\theta(x_T))}_{\text{Regularization}} + \underbrace{\sum_{t=2} D_{KL}(q(x_{t-1}|x_t, x_0) \parallel P_\theta(x_{t-1}|x_t))}_{\text{Denoising Process}} - \underbrace{E_q \log P_\theta(x_0|x_1)}_{\text{Reconstruction}}$$

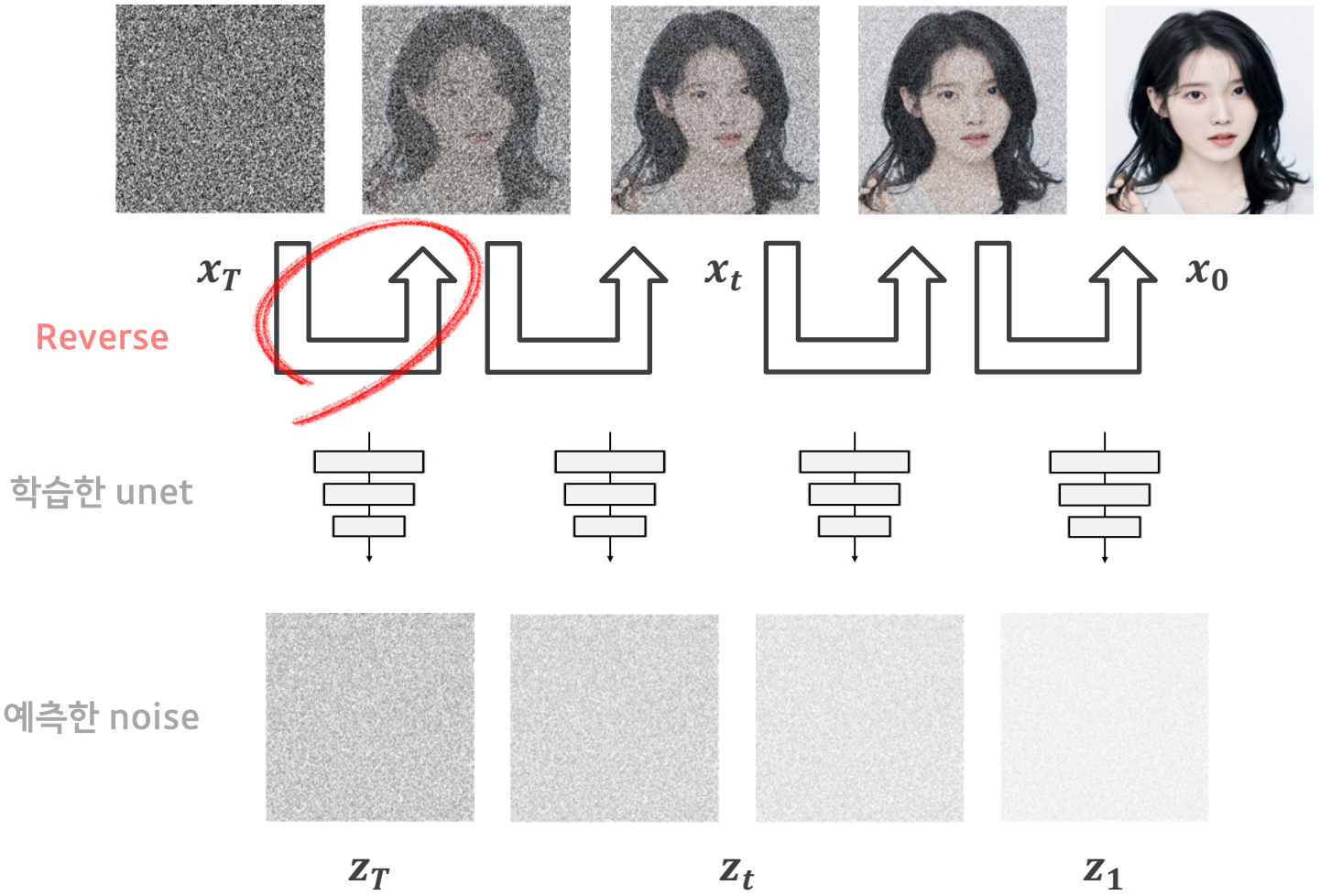
어차피 둘다 가우시안임

ddpm, ddim

x1이나 x0나..

www.youtube.com/watch?v=_JQSMhqXw-4

NAVER AI LAB




reverse process 종류

DDPM

DDIM

# DDPM

Denoising diffusion probabilistic models
NeurIPS 2020

NAVER AI LAB

- Overview
  - $q(X_{t-1} \mid X_t) \approx p_\theta(X_{t-1} \mid X_t)$
    - $N\left(X_{t-1};\ \mu_\theta(X_t, t),\ \Sigma_\theta(X_t, t)\right)$

실제와 network 분포를 맞추자

mean, variance

# DDPM | Reverse process

NAVER AI LAB

- Overview
  - $q(X_{t-1} \mid X_t) \approx p_\theta(X_{t-1} \mid X_t)$
    - $N\left(X_{t-1};\ \mu_\theta(X_t, t),\ \Sigma_\theta(X_t, t)\right)$
  - $q(X_{t-1} \mid X_t) \rightarrow q(X_{t-1} \mid X_t, X_0)$
    - $N\left(X_{t-1};\ \tilde{\mu}(X_t, X_0), \tilde{\Sigma}(X_t, X_0)\right)$

실제와 network 분포를 맞추자

mean, variance

xt들 분포는 알 수 없으니,
tractable하게 바꾸려 x0를 조건부로 추가

# DDPM | Reverse process

NAVER AI LAB

- Overview
  - $q(X_{t-1} \mid X_t) \approx p_\theta(X_{t-1} \mid X_t)$
    - $N\big(X_{t-1};\ \mu_\theta(X_t, t),\ \Sigma_\theta(X_t, t)\big)$
  - $q(X_{t-1} \mid X_t) \rightarrow q(X_{t-1} \mid X_t, X_0)$
    - $N\left(X_{t-1};\ \tilde{\mu}(X_t, X_0), \tilde{\Sigma}(X_t, X_0)\right)$

실제와 network 분포를 맞추자

mean, variance

xt들 분포는 알 수 없으니,
tractable하게 바꾸려 x0를 조건부로 추가

Note

$$P(B \mid A) = P(A \mid B)\frac{P(B)}{P(A)}$$

Note

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

$$\begin{aligned} q(x_{t-1}|x_t, x_0) &= \frac{q(x_t|x_{t-1}, x_0)q(x_{t-1}, x_0)}{q(x_t, x_0)} \\ &= \frac{q(x_t|x_{t-1}, x_0)q(x_{t-1}|x_0)q(x_0)}{q(x_t|x_0)q(x_0)} \\ &= q(x_t|x_{t-1}, x_0) \times \frac{q(x_{t-1}|x_0)}{q(x_t|x_0)} \end{aligned}$$

# DDPM | Reverse process

NAVER AI LAB

- Overview
  - $q(X_{t-1} \mid X_t) \approx p_\theta(X_{t-1} \mid X_t)$
    - $N\big(X_{t-1};\ \mu_\theta(X_t, t),\ \Sigma_\theta(X_t, t)\big)$
  - $q(X_{t-1} \mid X_t) \ \rightarrow q(X_{t-1} \mid X_t, X_0)$
    - $N\left(X_{t-1};\ \tilde{\mu}(X_t, X_0), \tilde{\Sigma}(X_t, X_0)\right)$

실제와 network 분포를 맞추자

mean, variance

xt들 분포는 알 수 없으니,
tractable하게 바꾸려 x0를 조건부로 추가

Note

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

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

$$q(x_t|x_{t-1}) = \frac{1}{\sqrt{2\pi\beta_t}} \exp\left(-\frac{(x_t - \sqrt{1-\beta_t}x_{t-1})^2}{2\beta_t}\right)$$

$$q(x_t|x_0) = \frac{1}{\sqrt{2\pi(1-\bar{\alpha}_t)}} \exp\left(-\frac{(x_t - \sqrt{\bar{\alpha}_t}x_0)^2}{2(1-\bar{\alpha}_t)}\right)$$

$$q(x_{t-1}|x_0) = \frac{1}{\sqrt{2\pi(1-\bar{\alpha}_{t-1})}} \exp\left(-\frac{(x_{t-1} - \sqrt{\bar{\alpha}_{t-1}}x_0)^2}{2(1-\bar{\alpha}_{t-1})}\right)$$

Note

$$\boldsymbol{P(B \mid A) = P(A \mid B) \frac{P(B)}{P(A)}}$$

$$\boldsymbol{N\big(x;\ \mu, \sigma^2\big) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}}$$

- $q(x_t \mid x_{t-1}) = N\big(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t * I\big)$
- $q(x_t \mid x_0) = N\big(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t) * I\big)$
  - $\alpha_t = 1 - \beta_t$
  - $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$

lilianweng.github.io/posts/2021-07-11-diffusion-models

# DDPM | Reverse process

NAVER AI LAB

- Overview
  - $q(X_{t-1} \mid X_t) \approx p_\theta(X_{t-1} \mid X_t)$
    - $N\big(X_{t-1};\ \mu_\theta(X_t, t),\ \Sigma_\theta(X_t, t)\big)$
  - $q(X_{t-1} \mid X_t) \ \rightarrow q(X_{t-1} \mid X_t, X_0)$
    - $N\left(X_{t-1};\ \tilde{\mu}(X_t, X_0), \tilde{\Sigma}(X_t, X_0)\right)$

실제와 network 분포를 맞추자

mean, variance

xt들 분포는 알 수 없으니,
tractable하게 바꾸려 x0를 조건부로 추가

Note

$$P(B \mid A) = P(A \mid B)\frac{P(B)}{P(A)}$$

$$N\big(x;\ \mu, \sigma^2\big) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

- $q(x_t \mid x_{t-1}) = N\big(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t * I\big)$
- $q(x_t \mid x_0) = N\big(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t) * I\big)$
  - $\alpha_t = 1 - \beta_t$
  - $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$

Note

$$\therefore q(x_{t-1}|x_t, x_0) = \frac{1}{\sqrt{2\pi\beta_t(\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t})}}\exp\left(-\frac{(x_t - \sqrt{1-\beta_t}x_{t-1})^2}{2\beta_t} - \frac{(x_{t-1} - \sqrt{\bar{\alpha}_{t-1}}x_0)^2}{2(1-\bar{\alpha}_{t-1})} + \frac{(x_t - \sqrt{\bar{\alpha}_t}x_0)^2}{2(1-\bar{\alpha}_t)}\right)$$

$$= \frac{1}{\sqrt{2\pi\beta_t(\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t})}}\exp\left(-\left(\left[\frac{1}{2(1-\bar{\alpha}_{t-1})} + \frac{1-\beta_t}{2\beta_t}\right]x_{t-1}^2 - \left[\frac{2\sqrt{1-\beta_t}}{2\beta_t}x_t + \frac{2\sqrt{\bar{\alpha}_{t-1}}}{2(1-\bar{\alpha}_{t-1})}x_0\right]x_{t-1} + C\right)\right)$$

$$= \frac{1}{\sqrt{2\pi\beta_t(\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t})}}\exp\left(-\frac{1}{2\beta_t(\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t})}\left[x_{t-1}^2 - \left(\frac{2\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\alpha_t}x_t + \frac{2\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}x_0\right)x_{t-1} + C\right]\right)$$

$$\approx \frac{1}{\sqrt{2\pi\beta_t(\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t})}}\exp\left(-\frac{1}{2\beta_t(\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t})}\left[x_{t-1} - \left(\frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_{t-1}}x_t\right)\right]^2\right)$$

대입해서 풀면 ...

lilianweng.github.io/posts/2021-07-11-diffusion-models

NAVER AI LAB

- Overview
  - $q(X_{t-1} \mid X_t) \approx p_\theta(X_{t-1} \mid X_t)$
    - $N\big(X_{t-1};\ \mu_\theta(X_t, t),\ \Sigma_\theta(X_t, t)\big)$
  - $q(X_{t-1} \mid X_t) \ \rightarrow q(X_{t-1} \mid X_t, X_0)$
    - $N\left(X_{t-1};\ \tilde{\mu}(X_t, X_0), \tilde{\Sigma}(X_t, X_0)\right)$
    - $N\big(X_{t-1}; \boldsymbol{\mu}, \boldsymbol{\sigma^2}\big)$

실제와 network 분포를 맞추자

mean, variance

xt들 분포는 알 수 없으니,
tractable하게 바꾸려 x0를 조건부로 추가

Note

$$\boldsymbol{P(B \mid A) = P(A \mid B)\frac{P(B)}{P(A)}}$$

$$\boldsymbol{N\big(x;\ \mu, \sigma^2\big) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}}$$

- $q(x_t \mid x_{t-1}) = N\big(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t * I\big)$
- $q(x_t \mid x_0) = N\big(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t) * I\big)$
  - $\alpha_t = 1 - \beta_t$
  - $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$

Note

$$\therefore q(x_{t-1}|x_t, x_0) = \frac{1}{\sqrt{2\pi\beta_t(\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t})}} \exp\left(-\frac{(x_t - \sqrt{1-\beta_t}x_{t-1})^2}{2\beta_t} - \frac{(x_{t-1} - \sqrt{\bar{\alpha}_{t-1}}x_0)^2}{2(1-\bar{\alpha}_{t-1})} + \frac{(x_t - \sqrt{\bar{\alpha}_t}x_0)^2}{2(1-\bar{\alpha}_t)}\right)$$

$$= \frac{1}{\sqrt{2\pi\beta_t(\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t})}} \exp\left(-\left(\left[\frac{1}{2(1-\bar{\alpha}_{t-1})} + \frac{1-\beta_t}{2\beta_t}\right]x_{t-1}^2 - \left[\frac{2\sqrt{1-\beta_t}}{2\beta_t}x_t + \frac{2\sqrt{\bar{\alpha}_{t-1}}}{2(1-\bar{\alpha}_{t-1})}x_0\right]x_{t-1} + C\right)\right)$$

$$= \frac{1}{\sqrt{2\pi\beta_t(\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t})}} \exp\left(-\frac{1}{2\beta_t(\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t})}\left[x_{t-1}^2 - \left(\frac{2\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\alpha_t}x_t + \frac{2\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}x_0\right)x_{t-1} + C\right]\right)$$

$$\approx \frac{1}{\sqrt{2\pi\beta_t(\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t})}} \exp\left(-\frac{1}{2\beta_t(\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t})}\left[x_{t-1} - \left(\frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_{t-1}}x_t\right)\right]^2\right)$$

$\boldsymbol{\sigma^2}$ $\qquad$ $\boldsymbol{\mu}$

대입해서 풀면 ...

lilianweng.github.io/posts/2021-07-11-diffusion-models

NAVER AI LAB

- Overview
  - $q(X_{t-1} \mid X_t) \rightarrow q(X_{t-1} \mid X_t, X_0)$
    - $N\left(X_{t-1};\ \tilde{\mu}(X_t, X_0), \tilde{\Sigma}(X_t, X_0)\right)$
    - $N(X_{t-1}; \boldsymbol{\mu}, \boldsymbol{\sigma^2}) = N\left(X_{t-1}; \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t} x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} x_t\ , \beta_t \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\right)$

lilianweng.github.io/posts/2021-07-11-diffusion-models

# DDPM | Training

NAVER AI LAB

- Overview
  - $q(X_{t-1} \mid X_t) \rightarrow q(X_{t-1} \mid X_t, X_0)$
    - $N\left(X_{t-1};\ \tilde{\mu}(X_t, X_0), \tilde{\Sigma}(X_t, X_0)\right)$
    - $N(X_{t-1}; \boldsymbol{\mu}, \boldsymbol{\sigma^2}) = N\left(X_{t-1}; \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t} x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} x_t\ , \beta_t \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\right)$
  - $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon$ Forward process

**Note**

- $q(x_t \mid x_{t-1}) = N(x_t; \sqrt{1-\beta_t} x_{t-1}, \beta_t * I)$
- $q(x_t \mid x_0) = N(x_t; \sqrt{\bar{\alpha}_t} x_0, (1-\bar{\alpha}_t) * I)$
  - $\alpha_t = 1 - \beta_t$
  - $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$

lilianweng.github.io/posts/2021-07-11-diffusion-models

# DDPM | Training

NAVER AI LAB

- Overview
  - $q(X_{t-1} \mid X_t) \rightarrow q(X_{t-1} \mid X_t, X_0)$
    - $N\left(X_{t-1};\ \tilde{\mu}(X_t, X_0), \tilde{\Sigma}(X_t, X_0)\right)$
    - $N(X_{t-1}; \boldsymbol{\mu}, \boldsymbol{\sigma^2}) = N\left(X_{t-1}; \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t} x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} x_t\ , \beta_t \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\right)$
  - $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon$ Forward process

**Note**

- $q(x_t \mid x_{t-1}) = N(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t * I)$
- $q(x_t \mid x_0) = N(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t) * I)$
  - $\alpha_t = 1 - \beta_t$
  - $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$

**Note** x0를 xt로 바꿔서 정리합시다.

$$\tilde{\mu}_t = \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\frac{1}{\sqrt{\bar{\alpha}_t}}(x_t - \sqrt{1-\bar{\alpha}_t}\epsilon) + \frac{\sqrt{\bar{\alpha}_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}x_t$$

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

$$= \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon\right)$$

lilianweng.github.io/posts/2021-07-11-diffusion-models

NAVER AI LAB

- Overview
  - $q(X_{t-1} \mid X_t) \rightarrow q(X_{t-1} \mid X_t, X_0)$
    - $N\left(X_{t-1};\ \tilde{\mu}(X_t, X_0), \tilde{\Sigma}(X_t, X_0)\right)$
    - $N(X_{t-1}; \boldsymbol{\mu}, \boldsymbol{\sigma^2}) = N\left(X_{t-1}; \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t} x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} x_t\ , \beta_t \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\right)$
  - $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon$ Forward process

**Note**

- $q(x_t \mid x_{t-1}) = N(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t * I)$
- $q(x_t \mid x_0) = N(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t) * I)$
  - $\alpha_t = 1 - \beta_t$
  - $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$

**Note** x0를 xt로 바꿔서 정리합시다.

$$\tilde{\mu}_t = \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\frac{1}{\sqrt{\bar{\alpha}_t}}(x_t - \sqrt{1-\bar{\alpha}_t}\epsilon) + \frac{\sqrt{\bar{\alpha}_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}x_t$$

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

$$= \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon\right)$$

$$\mu_\theta = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta\right)$$

lilianweng.github.io/posts/2021-07-11-diffusion-models

# DDPM | Training

NAVER AI LAB

- Overview
  - $q(X_{t-1} \mid X_t) \rightarrow q(X_{t-1} \mid X_t, X_0)$
    - $N\left(X_{t-1};\ \tilde{\mu}(X_t, X_0), \tilde{\Sigma}(X_t, X_0)\right)$
    - $N\left(X_{t-1}; \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t} x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} x_t\ , \beta_t \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\right)$
    - $N\left(X_{t-1}; \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon\right), \tilde{\beta}_t\right)$

**Note**

- $q(x_t \mid x_{t-1}) = N\left(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t * I\right)$
- $q(x_t \mid x_0) = N\left(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t) * I\right)$
  - $\alpha_t = 1 - \beta_t$
  - $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$

$$\tilde{\beta}_t := \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$$

lilianweng.github.io/posts/2021-07-11-diffusion-models

# DDPM | Training

NAVER AI LAB

- Overview
  - $q(X_{t-1} \mid X_t) \rightarrow q(X_{t-1} \mid X_t, X_0)$
    - $N\left(X_{t-1};\ \tilde{\mu}(X_t, X_0), \tilde{\Sigma}(X_t, X_0)\right)$
    - $N\left(X_{t-1}; \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t} x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} x_t\ , \beta_t \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\right)$
    - $N\left(X_{t-1}; \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon\right), \tilde{\beta}_t\right)$
    - $N\left(X_{t-1}; \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta(x_t)\right), \tilde{\beta}_t\right)$ xt에 대한 noise를 예측해야 뒤로가지

**Note**

- $q(x_t \mid x_{t-1}) = N(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t * I)$
- $q(x_t \mid x_0) = N(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t) * I)$
  - $\alpha_t = 1 - \beta_t$
  - $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$

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

$$\tilde{\beta}_t := \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$$

lilianweng.github.io/posts/2021-07-11-diffusion-models

# DDPM | Training

NAVER AI LAB

- Overview
  - $q(X_{t-1} \mid X_t) \rightarrow q(X_{t-1} \mid X_t, X_0)$
    - $N\left(X_{t-1};\ \tilde{\mu}(X_t, X_0), \tilde{\Sigma}(X_t, X_0)\right)$
    - $N\left(X_{t-1}; \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t} x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} x_t\ , \beta_t \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\right)$
    - $N\left(X_{t-1}; \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon\right), \tilde{\beta}_t\right)$
    - $N\left(X_{t-1}; \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta(x_t)\right), \tilde{\beta}_t\right)$ xt에 대한 noise를 예측해야 뒤로가지

**Note**

- $q(x_t \mid x_{t-1}) = N(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t * I)$
- $q(x_t \mid x_0) = N(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t) * I)$
  - $\alpha_t = 1 - \beta_t$
  - $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$

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

$$\tilde{\beta}_t := \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$$

**Note**

$$\boldsymbol{Loss = \varepsilon - \varepsilon_\theta(x_t)}$$ 따라서, xt를 만든 noise와 loss

lilianweng.github.io/posts/2021-07-11-diffusion-models

NAVER AI LAB

- Overview
  - $q(X_{t-1} \mid X_t) \rightarrow q(X_{t-1} \mid X_t, X_0)$
    - $N\left(X_{t-1};\ \tilde{\mu}(X_t, X_0), \tilde{\Sigma}(X_t, X_0)\right)$
    - $N\left(X_{t-1}; \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t} x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} x_t\ , \beta_t \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\right)$
    - $N\left(X_{t-1}; \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon\right), \tilde{\beta}_t\right)$
    - $N\left(X_{t-1}; \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta(x_t)\right), \tilde{\beta}_t\right)$ xt에 대한 noise를 예측해야 뒤로가지

**Note**

- $q(x_t \mid x_{t-1}) = N(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t * I)$
- $q(x_t \mid x_0) = N(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t) * I)$
  - $\alpha_t = 1 - \beta_t$
  - $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$

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

$$\tilde{\beta}_t := \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$$

**Note**

$\boldsymbol{Loss = \varepsilon - \varepsilon_\theta(x_t)}$ 따라서, xt를 만든 noise와 loss

$$Loss_{Diffusion} = \underbrace{D_{KL}\left(q(x_T|x_0) \parallel p_\theta(x_T)\right)}_{Regularization} + \underbrace{\sum_{t=2} D_{KL}\left(q(x_{t-1}|x_t, x_0) \parallel P_\theta(x_{t-1}|x_t)\right)}_{Denoising\ Process} - \underbrace{E_q \log P_\theta(x_0|x_1)}_{Reconstruction}$$

lilianweng.github.io/posts/2021-07-11-diffusion-models

# DDPM | Training

NAVER AI LAB

- Overview
  - $q(X_{t-1} \mid X_t) \rightarrow q(X_{t-1} \mid X_t, X_0)$
    - $N\left(X_{t-1};\ \tilde{\mu}(X_t, X_0), \tilde{\Sigma}(X_t, X_0)\right)$
    - $N\left(X_{t-1}; \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t} x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} x_t\ , \beta_t \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\right)$
    - $N\left(X_{t-1}; \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon\right), \tilde{\beta}_t\right)$
    - $N\left(X_{t-1}; \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta(x_t)\right), \tilde{\beta}_t\right)$ xt에 대한 noise를 예측해야 뒤로가지

**Note**

- $q(x_t \mid x_{t-1}) = N(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t * I)$
- $q(x_t \mid x_0) = N(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t) * I)$
  - $\alpha_t = 1 - \beta_t$
  - $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$

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

$$\tilde{\beta}_t := \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$$

**Note**

$$\boldsymbol{Loss} = \boldsymbol{\varepsilon} - \boldsymbol{\varepsilon_\theta}(\boldsymbol{x_t})$$ 따라서, xt를 만든 noise와 loss

$$\mathbb{E}_{\mathbf{x}_0,\boldsymbol{\epsilon}}\left[\frac{\beta_t^2}{2\sigma_t^2\alpha_t(1-\bar{\alpha}_t)}\left\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta\left(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t\right)\right\|^2\right]$$

lilianweng.github.io/posts/2021-07-11-diffusion-models

# DDPM | Training

NAVER AI LAB

- Overview
  - $q(X_{t-1} \mid X_t) \rightarrow q(X_{t-1} \mid X_t, X_0)$
    - $N\left(X_{t-1};\ \tilde{\mu}(X_t, X_0), \tilde{\Sigma}(X_t, X_0)\right)$
    - $N\left(X_{t-1}; \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t} x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} x_t\ , \beta_t \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\right)$
    - $N\left(X_{t-1}; \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon\right), \tilde{\beta}_t\right)$
    - $N\left(X_{t-1}; \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta(x_t)\right), \tilde{\beta}_t\right)$ xt에 대한 noise를 예측해야 뒤로가지

**Note**

- $q(x_t \mid x_{t-1}) = N(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t * I)$
- $q(x_t \mid x_0) = N(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t) * I)$
  - $\alpha_t = 1 - \beta_t$
  - $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$

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

$$\tilde{\beta}_t := \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$$

**Note**

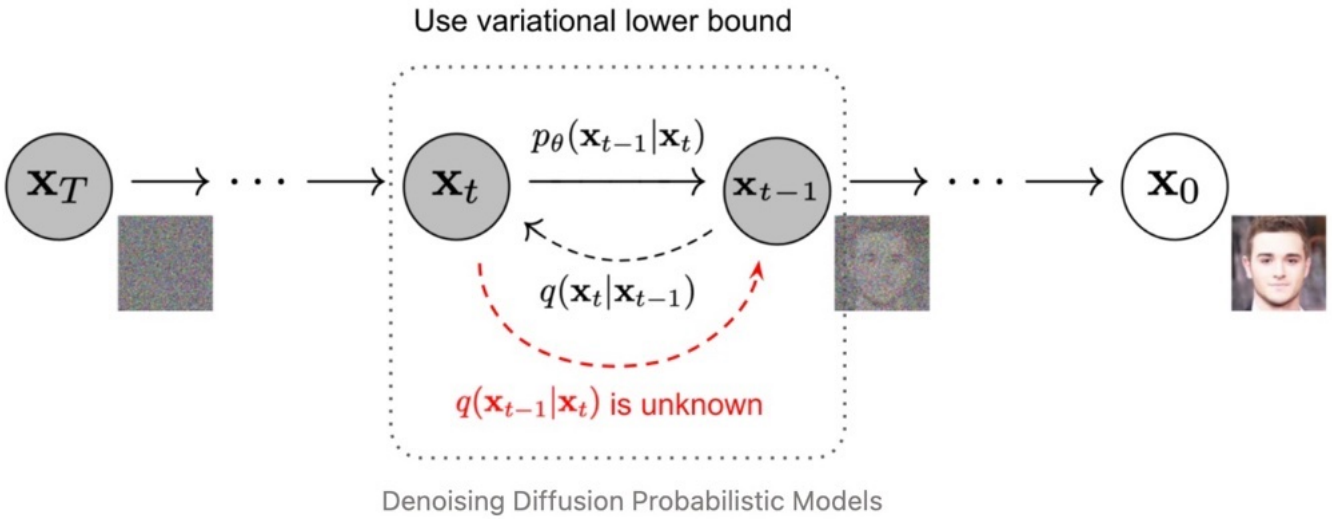
$$\boldsymbol{Loss} = \boldsymbol{\varepsilon} - \boldsymbol{\varepsilon_\theta}(\boldsymbol{x_t})$$ 따라서, xt를 만든 noise와 loss

$$\mathbb{E}_{\mathbf{x}_0,\boldsymbol{\epsilon}}\left[\frac{\beta_t^2}{2\sigma_t^2\alpha_t(1-\bar{\alpha}_t)}\left\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta\left(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t\right)\right\|^2\right]$$

$$L_{\text{simple}}(\theta) := \mathbb{E}_{t,\mathbf{x}_0,\boldsymbol{\epsilon}}\left[\left\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta\left(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t\right)\right\|^2\right]$$

t가 커질수록, 값이 작아져서.

lilianweng.github.io/posts/2021-07-11-diffusion-models



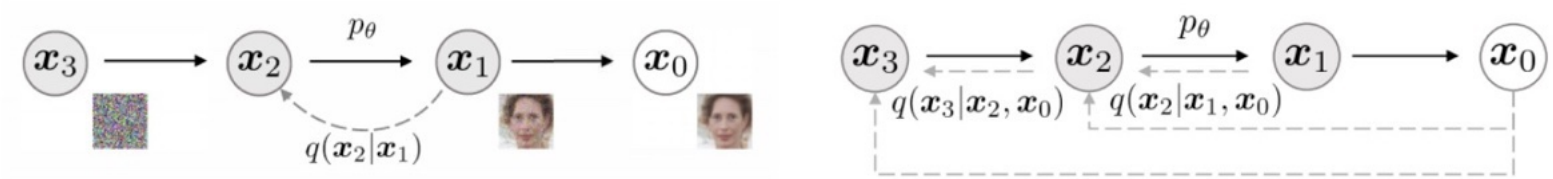

- $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon,\ \epsilon \sim \mathcal{N}(0, \mathbf{I})$ **(Forward)**
  - $\beta_1 = 10^{-4}, \beta_T = 0.02$
  - $\alpha_t := 1 - \beta_t,\ \bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$
- $\epsilon - \epsilon_\theta(\mathbf{x}_t) = \epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon)$ **(Loss)**
  - $\epsilon_\theta$ = prediction network
- $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\mathbf{x}_t, t)\right) + \sqrt{\tilde{\beta}_t}\epsilon,\ \epsilon \sim \mathcal{N}(0, \mathbf{I})$ **(Reverse)**
  - $\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$
    - $\tilde{\beta}_t = \beta_t$로 해도 성능차이 없음

# DDIM

Denoising diffusion implicit models
ICLR 2021

- Overview
  - DDPM
    - $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon$ (Forward)
    - $x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta(x_t)\right) + \sqrt{\tilde{\beta}_t}\varepsilon$ (Reverse)

속도 개느림




ddpm

Figure 1: Graphical models for diffusion (left) and non-Markovian (right) inference models.

Denoising Diffusion Implicit Models

lilianweng.github.io/posts/2021-07-11-diffusion-models

- Overview
  - DDPM
    - $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon$ (Forward)
    - $x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta(x_t)\right) + \sqrt{\tilde{\beta}_t}\varepsilon$ (Reverse)

속도 개느림

  - DDIM
    - $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon$ (Forward)
    - $x_{t-1} = \sqrt{\bar{\alpha}_{t-1}}f_\theta(x_t) + \sqrt{1-\bar{\alpha}_{t-1}-\tilde{\beta}_t}\varepsilon_\theta(x_t) + \sqrt{\tilde{\beta}_t}\epsilon$ (Reverse)

mean std

xt로부터 x0를 예측해서,
markov -> non-markov

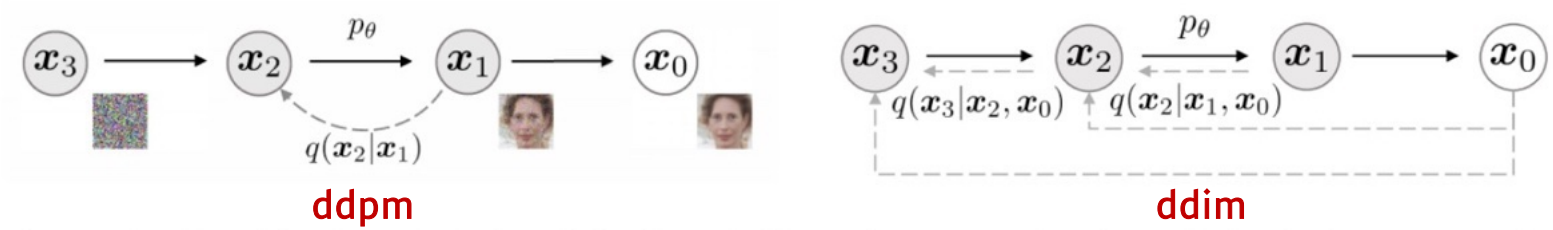



Figure 1: Graphical models for diffusion (left) and non-Markovian (right) inference models.

Denoising Diffusion Implicit Models

lilianweng.github.io/posts/2021-07-11-diffusion-models

# DDIM | Reverse process

NAVER AI LAB

- Overview
  - DDIM
    - $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\varepsilon$ (Forward)
    - $x_{t-1} = \sqrt{\bar{\alpha}_{t-1}} f_\theta(x_t) + \sqrt{1 - \bar{\alpha}_{t-1} - \tilde{\beta}_t}\varepsilon_\theta(x_t) + \sqrt{\tilde{\beta}_t}\epsilon$ (Reverse)

**Note**

- DDPM
  - $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\varepsilon$ (Forward)
  - $x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta(x_t)\right) + \sqrt{\tilde{\beta}_t}\varepsilon$ (Reverse)

$$\tilde{\beta}_t = \sigma_t^2 = \frac{1 - \bar{\alpha}_{t-1}}{1 - \bar{\alpha}_t} \cdot \beta_t$$

lilianweng.github.io/posts/2021-07-11-diffusion-models

# DDIM | Reverse process

NAVER AI LAB

- Overview
  - DDIM
    - $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon$ (Forward)
    - $x_{t-1} = \sqrt{\bar{\alpha}_{t-1}} f_\theta(x_t) + \sqrt{1-\bar{\alpha}_{t-1}-\tilde{\beta}_t}\varepsilon_\theta(x_t) + \sqrt{\tilde{\beta}_t}\epsilon$ (Reverse)
      - $= \sqrt{\bar{\alpha}_{t-1}}\hat{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\varepsilon_\theta(x_t) + \sigma_t\epsilon$
      - $N(x_{t-1}; \sqrt{\bar{\alpha}_{t-1}}\hat{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\varepsilon_\theta(x_t), \sigma_t^2)$

**Note**

- DDPM
  - $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon$ (Forward)
  - $x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta(x_t)\right) + \sqrt{\tilde{\beta}_t}\varepsilon$ (Reverse)

$$\tilde{\beta}_t = \sigma_t^2 = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\cdot\beta_t$$

$$\sigma_t^2 = \eta\cdot\tilde{\beta}_t$$

lilianweng.github.io/posts/2021-07-11-diffusion-models

# DDIM | Reverse process

NAVER AI LAB

- Overview
  - DDIM
    - $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon$ (Forward)
    - $x_{t-1} = \sqrt{\bar{\alpha}_{t-1}}f_\theta(x_t) + \sqrt{1-\bar{\alpha}_{t-1}-\tilde{\beta}_t}\varepsilon_\theta(x_t) + \sqrt{\tilde{\beta}_t}\epsilon$ (Reverse)
      - $= \sqrt{\bar{\alpha}_{t-1}}\hat{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\varepsilon_\theta(x_t) + \sigma_t\epsilon$
      - $N(x_{t-1}; \sqrt{\bar{\alpha}_{t-1}}\hat{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\varepsilon_\theta(x_t), \sigma_t^2)$

**Note**

- DDPM
  - $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon$ (Forward)
  - $x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta(x_t)\right) + \sqrt{\tilde{\beta}_t}\varepsilon$ (Reverse)

$$\tilde{\beta}_t = \sigma_t^2 = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\cdot\beta_t$$

$$\sigma_t^2 = \eta\cdot\tilde{\beta}_t$$

**Note**

- DDPM
  - $N\left(X_{t-1}; \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon\right), \tilde{\beta}_t\right)$
  - $N\left(X_{t-1}; \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_{t-1}}x_t, \tilde{\beta}_t\right)$

$x_0$를 $x_t$로 정리해서 대입해서 풀었었음

lilianweng.github.io/posts/2021-07-11-diffusion-models

# DDIM | Reverse process

NAVER AI LAB

- Overview
  - DDIM
    - $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon$ (Forward)
    - $x_{t-1} = \sqrt{\bar{\alpha}_{t-1}}f_\theta(x_t) + \sqrt{1-\bar{\alpha}_{t-1}-\tilde{\beta}_t}\varepsilon_\theta(x_t) + \sqrt{\tilde{\beta}_t}\epsilon$ (Reverse)
      - $= \sqrt{\bar{\alpha}_{t-1}}\hat{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\varepsilon_\theta(x_t) + \sigma_t\epsilon$
      - $N(x_{t-1}; \sqrt{\bar{\alpha}_{t-1}}\hat{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\varepsilon_\theta(x_t), \sigma_t^2)$

**Note**

- DDPM
  - $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon$ (Forward)
  - $x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta(x_t)\right) + \sqrt{\tilde{\beta}_t}\varepsilon$ (Reverse)

$$\tilde{\beta}_t = \sigma_t^2 = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\cdot\beta_t$$

$$\sigma_t^2 = \eta\cdot\tilde{\beta}_t$$

**Note**

- DDPM
  - $N\left(X_{t-1}; \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon\right), \tilde{\beta}_t\right)$
  - $N\left(X_{t-1}; \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_{t-1}}x_t, \tilde{\beta}_t\right)$

$x_0$를 $x_t$로 정리해서 대입해서 풀었었음

$\varepsilon_{t-1}$를 $\varepsilon_t$로 표현하는데,
* variance 맞춰주고
* $\varepsilon_{t-1}$와 $\varepsilon_t$ 차이를 $\varepsilon$으로 매꾸자

$$\begin{aligned}
\mathbf{x}_{t-1} &= \sqrt{\bar{\alpha}_{t-1}}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}}\boldsymbol{\epsilon}_{t-1} \\
&= \sqrt{\bar{\alpha}_{t-1}}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\boldsymbol{\epsilon}_t + \sigma_t\boldsymbol{\epsilon} \\
&= \sqrt{\bar{\alpha}_{t-1}}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\frac{\mathbf{x}_t - \sqrt{\bar{\alpha}_t}\mathbf{x}_0}{\sqrt{1-\bar{\alpha}_t}} + \sigma_t\boldsymbol{\epsilon} \\
q_\sigma(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) &= \mathcal{N}(\mathbf{x}_{t-1}; \sqrt{\bar{\alpha}_{t-1}}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\frac{\mathbf{x}_t - \sqrt{\bar{\alpha}_t}\mathbf{x}_0}{\sqrt{1-\bar{\alpha}_t}}, \sigma_t^2\mathbf{I})
\end{aligned}$$

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

(*) Recall that when we merge two Gaussians with different variance, $\mathcal{N}(\mathbf{0}, \sigma_1^2\mathbf{I})$ and $\mathcal{N}(\mathbf{0}, \sigma_2^2\mathbf{I})$, the new distribution is $\mathcal{N}(\mathbf{0}, (\sigma_1^2 + \sigma_2^2)\mathbf{I})$. Here the merged standard deviation is $\sqrt{(1-\alpha_t) + \alpha_t(1-\alpha_{t-1})} = \sqrt{1-\alpha_t\alpha_{t-1}}$.

lilianweng.github.io/posts/2021-07-11-diffusion-models

# DDIM | Reverse process

NAVER AI LAB

- Overview
  - DDIM
    - $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon$ (Forward)
    - $x_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\hat{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\varepsilon_\theta(x_t) + \sigma_t\epsilon$ (Reverse)

- In paper, DDIM $\alpha$ = DDPM $\bar{\alpha}$
- $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$ (Forward)
- $\epsilon - \epsilon_\theta(\mathbf{x}_t) = \epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon)$ (Loss)
  - $\epsilon_\theta$ = prediction network
- $$\mathbf{x}_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\underbrace{\left(\frac{\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\epsilon_\theta(\mathbf{x}_t)}{\sqrt{\bar{\alpha}_t}}\right)}_{\text{predicted } \mathbf{x}_0 = f_\theta(\mathbf{x}_t)} + \underbrace{\sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\cdot\epsilon_\theta(\mathbf{x}_t)}_{\text{direction pointing to } \mathbf{x}_t} + \underbrace{\sigma_t\epsilon}_{\text{noise}}$$
  (Reverse)
  - deterministic when $\sigma_t = 0$ → consistency (DDIM)
  - stochastic when $\sigma_t = 1$ → inconsistency (DDPM)

**Note**

- DDPM
  - $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon$ (Forward)
  - $x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta(x_t)\right) + \sqrt{\tilde{\beta}_t}\varepsilon$ (Reverse)

$$\tilde{\beta}_t = \sigma_t^2 = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\cdot\beta_t$$

$$\sigma_t^2 = \eta\cdot\tilde{\beta}_t$$

lilianweng.github.io/posts/2021-07-11-diffusion-models

NAVER AI LAB

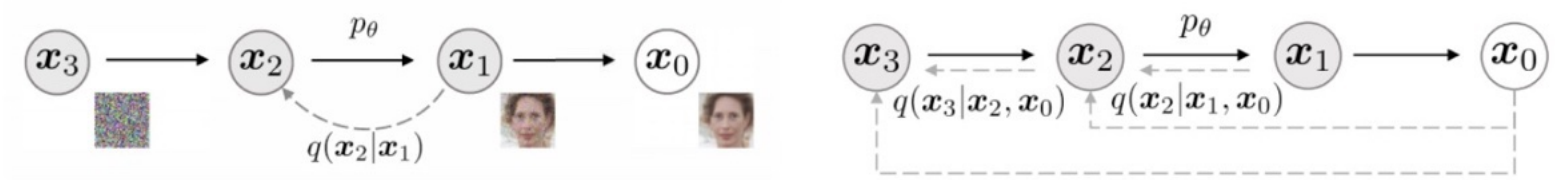



Figure 1: Graphical models for diffusion (left) and non-Markovian (right) inference models.

Denoising Diffusion Implicit Models

## DDPM

- $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon,\ \epsilon \sim \mathcal{N}(0, \mathbf{I})$ `(Forward)`
  - $\beta_1 = 10^{-4}, \beta_T = 0.02$
  - $\alpha_t := 1 - \beta_t,\ \bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$
- $\epsilon - \epsilon_\theta(\mathbf{x}_t) = \epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon)$ `(Loss)`
  - $\epsilon_\theta$ = prediction network
- $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\mathbf{x}_t, t)\right) + \sqrt{\tilde{\beta}_t}\epsilon,\ \epsilon \sim \mathcal{N}(0, \mathbf{I})$ `(Reverse)`
  - $\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$
    - $\tilde{\beta}_t = \beta_t$로 해도 성능차이 없음

## DDIM

- `In paper,` DDIM $\alpha$ = DDPM $\bar{\alpha}$
- $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$ `(Forward)`
- $\epsilon - \epsilon_\theta(\mathbf{x}_t) = \epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon)$ `(Loss)`
  - $\epsilon_\theta$ = prediction network
- $\mathbf{x}_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\underbrace{\left(\frac{\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\epsilon_\theta(\mathbf{x}_t)}{\sqrt{\bar{\alpha}_t}}\right)}_{\text{predicted } \mathbf{x}_0 = f_\theta(\mathbf{x}_t)} + \underbrace{\sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\cdot\epsilon_\theta(\mathbf{x}_t)}_{\text{direction pointing to } \mathbf{x}_t} + \underbrace{\sigma_t\epsilon}_{\text{noise}}$

  `(Reverse)`
  - deterministic when $\sigma_t = 0$ → consistency (DDIM)
  - stochastic when $\sigma_t = 1$ → inconsistency (DDPM)

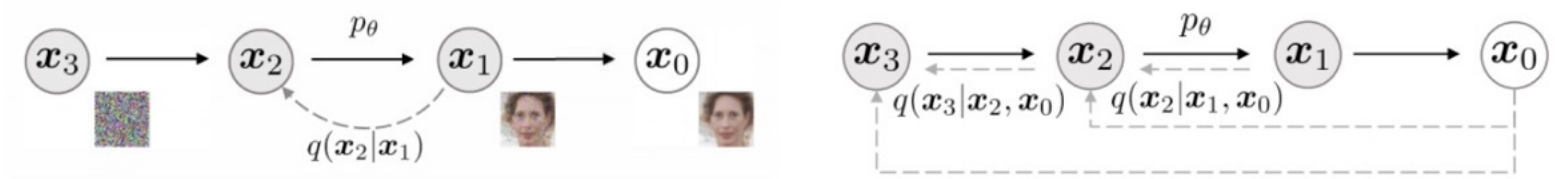



Figure 1: Graphical models for diffusion (left) and non-Markovian (right) inference models.

Denoising Diffusion Implicit Models

최근 Trend

$$x_{t-1} = \hat{\mu}_t(x_t) + \sigma_t \epsilon_t$$

$$x_{t-1} = \sqrt{\bar{\alpha}_{t-1}} P(f_t(x_t)) + D(f_t(x_t)) + \sigma_t \epsilon_t$$

$$* \; P(f_t(x_t)) = \frac{x_t - \sqrt{1-\bar{\alpha}_t} f_t(x_t)}{\sqrt{\bar{\alpha}_t}}$$

$$* \; D(f_t(x_t)) = \sqrt{1 - \bar{\alpha}_{t-1} - \sigma_t^2} f_t(x_t)$$

## DDPM

- $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon,\ \epsilon \sim \mathcal{N}(0, \mathbf{I})$ (Forward)
  - $\beta_1 = 10^{-4}, \beta_T = 0.02$
  - $\alpha_t := 1 - \beta_t,\ \bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$
- $\epsilon - \epsilon_\theta(\mathbf{x}_t) = \epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon)$ (Loss)
  - $\epsilon_\theta$ = prediction network
- $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\mathbf{x}_t, t)\right) + \sqrt{\tilde{\beta}_t}\epsilon,\ \epsilon \sim \mathcal{N}(0, \mathbf{I})$ (Reverse)
  - $\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$
    - $\tilde{\beta}_t = \beta_t$로 해도 성능차이 없음

## DDIM

- In paper, DDIM $\alpha$ = DDPM $\bar{\alpha}$
- $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$ (Forward)
- $\epsilon - \epsilon_\theta(\mathbf{x}_t) = \epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon)$ (Loss)
  - $\epsilon_\theta$ = prediction network
- $\mathbf{x}_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\underbrace{\left(\frac{\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\epsilon_\theta(\mathbf{x}_t)}{\sqrt{\bar{\alpha}_t}}\right)}_{\text{predicted } \mathbf{x}_0 = f_\theta(\mathbf{x}_t)} + \underbrace{\sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\cdot\epsilon_\theta(\mathbf{x}_t)}_{\text{direction pointing to } \mathbf{x}_t} + \underbrace{\sigma_t\epsilon}_{\text{noise}}$

  (Reverse)
  - deterministic when $\sigma_t = 0$ → consistency (DDIM)
  - stochastic when $\sigma_t = 1$ → inconsistency (DDPM)

# Diffusion Models Beat GANs on Image Synthesis

NeurIPS 2021

# Guided diffusion | Overview

NAVER AI LAB

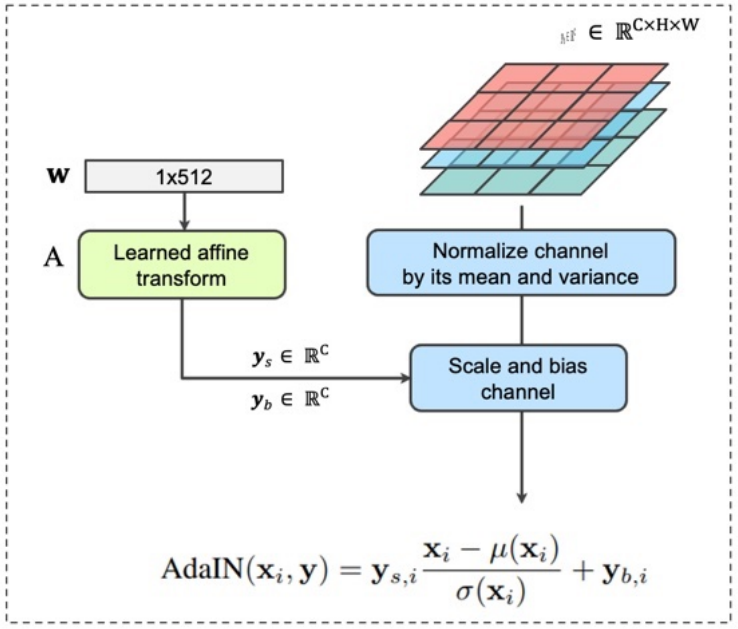
- Overview
  - Architecture improvements
    - Multi head attention
    - Multi resolution attention
    - Adaptive Group Normalization (AdaGN)
  - Truncation trick (fidelity & diversity)
    - Classifier guidance

Which do you like better, coffee or tea? - 문장 타입에 집중하는 어텐션

Which do you like better, coffee or tea? - 명사에 집중하는 어텐션

Which do you like better, coffee or tea? - 관계에 집중하는 어텐션

Which do you like better, coffee or tea? - 강조에 집중하는 어텐션




torch.mean(dim), torch.std(dim)

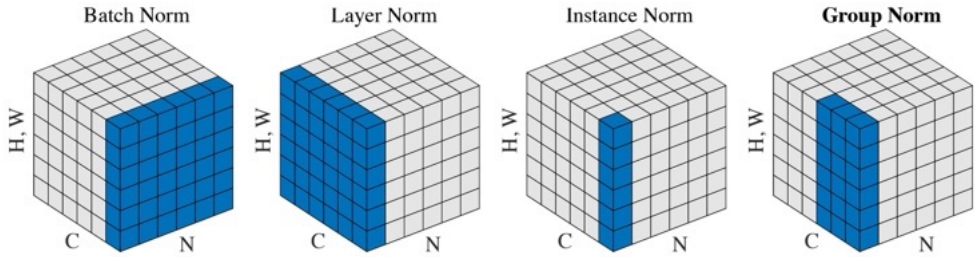



Figure 2. **Normalization methods**. Each subplot shows a feature map tensor, with $N$ as the batch axis, $C$ as the channel axis, and $(H, W)$ as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

[0] [1, 2, 3] [2, 3]

https://www.blossominkyung.com/deeplearning/transformer-mha

# Guided diffusion | Classifier guidance

NAVER AI LAB

- Overview
  - Truncation trick
    - Classifier guidance
      - $x_{t-1} \leftarrow N(\mu + s\Sigma\nabla \log p_\phi(y|x_t), \Sigma)$

Note

BigGAN

- Training = Gaussian normal distribution
- Inference = Truncated gaussian normal distribution

$\psi = 2$ $\psi = 1$ $\psi = 0.5$ $\psi = 0.04$

StyleGAN

- $w' = \bar{w} + \psi(w - \bar{w})$
  - $\bar{w} = \mathbf{mean}(w_1, \ldots, w_{4096})$
  - $\psi = 0 \rightarrow \mathbf{mean}\ of\ latent\ codes$
  - $\psi = 1 \rightarrow latent\ code$

$\psi = 1$ $\psi = 0.7$ $\psi = 0.5$ $\psi = 0$ $\psi = -0.5$ $\psi = -1$

# Guided diffusion | Classifier guidance

NAVER AI LAB

- Overview
  - Truncation trick
    - Classifier guidance
      - $x_{t-1} \leftarrow N\left(\mu + s\Sigma\nabla \log p_\phi(y|x_t), \Sigma\right)$

y는 class label

Note

Note

$$p_{\theta,\phi}(x_t|x_{t+1}, y) = Z p_\theta(x_t|x_{t+1}) p_\phi(y|x_t)$$

Z는 상수

$\theta$는 xt에 낀 noise 예측하는 diffusion network

$\phi$는 xt의 class label을 예측하는 classifier network

# Guided diffusion | Classifier guidance

NAVER AI LAB

- Overview
  - Truncation trick
    - Classifier guidance
      - $x_{t-1} \leftarrow N(\mu + s\Sigma\nabla \log p_\phi(y|x_t), \Sigma)$

y는 class label

Note

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Note

$$p_{\theta,\phi}(x_t|x_{t+1}, y) = Z p_\theta(x_t|x_{t+1}) p_\phi(y|x_t)$$

Z는 상수

$\theta$는 xt에 낀 noise 예측하는 diffusion network

$\phi$는 xt의 class label을 예측하는 classifier network

pdf 표현

$$p_\theta(x_t|x_{t+1}) = \mathcal{N}(\mu, \Sigma)$$
$$\log p_\theta(x_t|x_{t+1}) = -\frac{1}{2}(x_t - \mu)^T \Sigma^{-1}(x_t - \mu) + C$$

# Guided diffusion | Classifier guidance

NAVER AI LAB

- Overview
  - Truncation trick
    - Classifier guidance
      - $x_{t-1} \leftarrow N(\mu + s\Sigma\nabla \log p_\phi(y|x_t), \Sigma)$ y는 class label

Note

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

$$T_f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n = f(a) + f'(a)(x-a)$$

Note

$$p_{\theta,\phi}(x_t|x_{t+1}, y) = Z p_\theta(x_t|x_{t+1}) p_\phi(y|x_t)$$

Z는 상수
$\theta$는 xt에 낀 noise 예측하는 diffusion network
$\phi$는 xt의 class label을 예측하는 classifier network

pdf 표현

$$p_\theta(x_t|x_{t+1}) = \mathcal{N}(\mu, \Sigma)$$
$$\log p_\theta(x_t|x_{t+1}) = -\frac{1}{2}(x_t - \mu)^T \Sigma^{-1}(x_t - \mu) + C$$

테일러 급수

$$\log p_\phi(y|x_t) \approx \log p_\phi(y|x_t)|_{x_t=\mu} + (x_t - \mu)\nabla_{x_t} \log p_\phi(y|x_t)|_{x_t=\mu}$$
$$= (x_t - \mu)g + C_1$$

# Guided diffusion | Classifier guidance

NAVER AI LAB

- Overview
  - Truncation trick
    - Classifier guidance
      - $x_{t-1} \leftarrow N\left(\mu + s\Sigma\nabla \log p_\phi(y|x_t), \Sigma\right)$

y는 class label

Note

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

$$T_f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n = f(a) + f'(a)(x-a)$$

Note

$$p_{\theta,\phi}(x_t|x_{t+1}, y) = Z p_\theta(x_t|x_{t+1}) p_\phi(y|x_t)$$

Z는 상수

$\theta$는 xt에 낀 noise 예측하는 diffusion network

$\phi$는 xt의 class label을 예측하는 classifier network

$$p_\theta(x_t|x_{t+1}) = \mathcal{N}(\mu, \Sigma)$$
$$\log p_\theta(x_t|x_{t+1}) = -\frac{1}{2}(x_t - \mu)^T \Sigma^{-1}(x_t - \mu) + C$$

$$\log p_\phi(y|x_t) \approx \log p_\phi(y|x_t)|_{x_t=\mu} + (x_t - \mu)\nabla_{x_t} \log p_\phi(y|x_t)|_{x_t=\mu}$$
$$= (x_t - \mu)g + C_1$$

$$\log(p_\theta(x_t|x_{t+1}) p_\phi(y|x_t)) \approx -\frac{1}{2}(x_t - \mu)^T \Sigma^{-1}(x_t - \mu) + (x_t - \mu)g + C_2$$
$$= -\frac{1}{2}(x_t - \mu - \Sigma g)^T \Sigma^{-1}(x_t - \mu - \Sigma g) + \frac{1}{2} g^T \Sigma g + C_2$$
$$= -\frac{1}{2}(x_t - \mu - \Sigma g)^T \Sigma^{-1}(x_t - \mu - \Sigma g) + C_3$$
$$= \log p(z) + C_4, z \sim \mathcal{N}(\mu + \Sigma g, \Sigma)$$

# Guided diffusion | Classifier guidance

NAVER AI LAB

- Overview
  - Truncation trick
    - Classifier guidance
      - $x_{t-1} \leftarrow N\left(\mu + s\Sigma\nabla \log p_\phi(y|x_t), \Sigma\right)$

Note

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

$$T_f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n = f(a) + f'(a)(x-a)$$

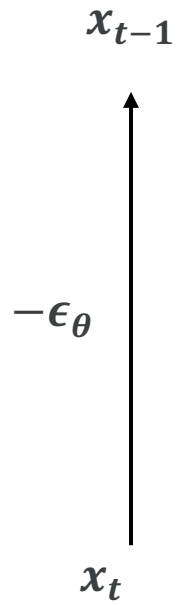
Note

**Algorithm 1** Classifier guided diffusion sampling, given a diffusion model $(\mu_\theta(x_t), \Sigma_\theta(x_t))$, classifier $p_\phi(y|x_t)$, and gradient scale $s$.

Input: class label $y$, gradient scale $s$
$x_T \leftarrow$ sample from $\mathcal{N}(0, \mathbf{I})$
**for all** $t$ from $T$ to 1 **do**
  $\mu, \Sigma \leftarrow \mu_\theta(x_t), \Sigma_\theta(x_t)$
  $x_{t-1} \leftarrow$ sample from $\mathcal{N}(\mu + s\Sigma\nabla_{x_t} \log p_\phi(y|x_t), \Sigma)$
**end for**
**return** $x_0$

**Algorithm 2** Classifier guided DDIM sampling, given a diffusion model $\epsilon_\theta(x_t)$, classifier $p_\phi(y|x_t)$, and gradient scale $s$.

Input: class label $y$, gradient scale $s$
$x_T \leftarrow$ sample from $\mathcal{N}(0, \mathbf{I})$
**for all** $t$ from $T$ to 1 **do**
  $\hat{\epsilon} \leftarrow \epsilon_\theta(x_t) - \sqrt{1-\bar{\alpha}_t}\nabla_{x_t} \log p_\phi(y|x_t)$
  $x_{t-1} \leftarrow \sqrt{\bar{\alpha}_{t-1}}\left(\frac{x_t - \sqrt{1-\bar{\alpha}_t}\hat{\epsilon}}{\sqrt{\bar{\alpha}_t}}\right) + \sqrt{1-\bar{\alpha}_{t-1}}\hat{\epsilon}$
**end for**
**return** $x_0$



# Guided diffusion | Classifier guidance

NAVER AI LAB

- Overview
  - Truncation trick
    - Classifier guidance
      - $x_{t-1} \leftarrow N\left(\mu + s\Sigma\nabla \log p_\phi(y|x_t), \Sigma\right)$

Note

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

$$T_f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n = f(a) + f'(a)(x-a)$$

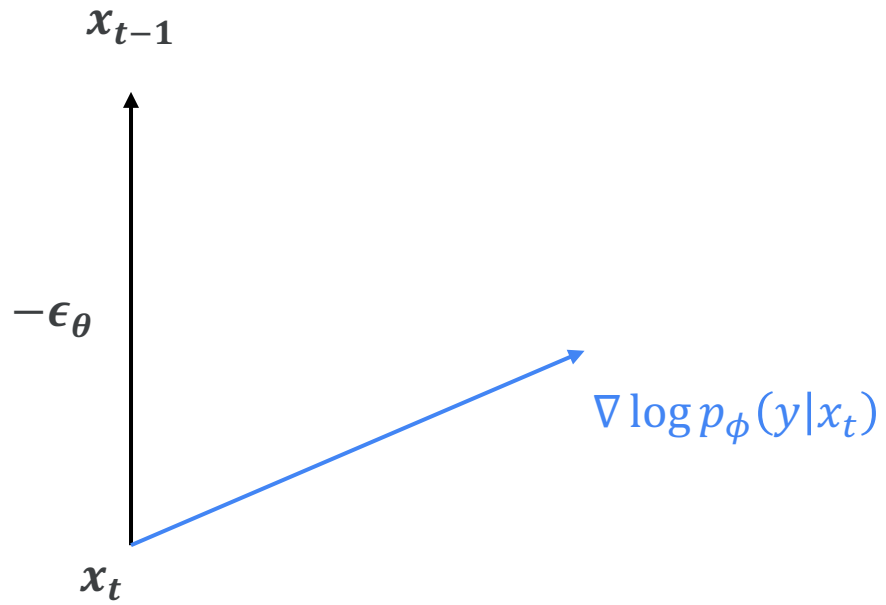
Note

**Algorithm 1** Classifier guided diffusion sampling, given a diffusion model $(\mu_\theta(x_t), \Sigma_\theta(x_t))$, classifier $p_\phi(y|x_t)$, and gradient scale $s$.

Input: class label $y$, gradient scale $s$
$x_T \leftarrow$ sample from $\mathcal{N}(0, \mathbf{I})$
**for all** $t$ from $T$ to 1 **do**
  $\mu, \Sigma \leftarrow \mu_\theta(x_t), \Sigma_\theta(x_t)$
  $x_{t-1} \leftarrow$ sample from $\mathcal{N}(\mu + s\Sigma\,\nabla_{x_t} \log p_\phi(y|x_t), \Sigma)$
**end for**
**return** $x_0$

**Algorithm 2** Classifier guided DDIM sampling, given a diffusion model $\epsilon_\theta(x_t)$, classifier $p_\phi(y|x_t)$, and gradient scale $s$.

Input: class label $y$, gradient scale $s$
$x_T \leftarrow$ sample from $\mathcal{N}(0, \mathbf{I})$
**for all** $t$ from $T$ to 1 **do**
  $\hat{\epsilon} \leftarrow \epsilon_\theta(x_t) - \sqrt{1-\bar{\alpha}_t}\,\nabla_{x_t} \log p_\phi(y|x_t)$
  $x_{t-1} \leftarrow \sqrt{\bar{\alpha}_{t-1}}\left(\frac{x_t - \sqrt{1-\bar{\alpha}_t}\hat{\epsilon}}{\sqrt{\bar{\alpha}_t}}\right) + \sqrt{1-\bar{\alpha}_{t-1}}\hat{\epsilon}$
**end for**
**return** $x_0$



# Guided diffusion | Classifier guidance

NAVER AI LAB

- Overview
  - Truncation trick
    - Classifier guidance
      - $x_{t-1} \leftarrow N(\mu + s\Sigma\nabla \log p_\phi(y|x_t), \Sigma)$

Note

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

$$T_f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n = f(a) + f'(a)(x-a)$$

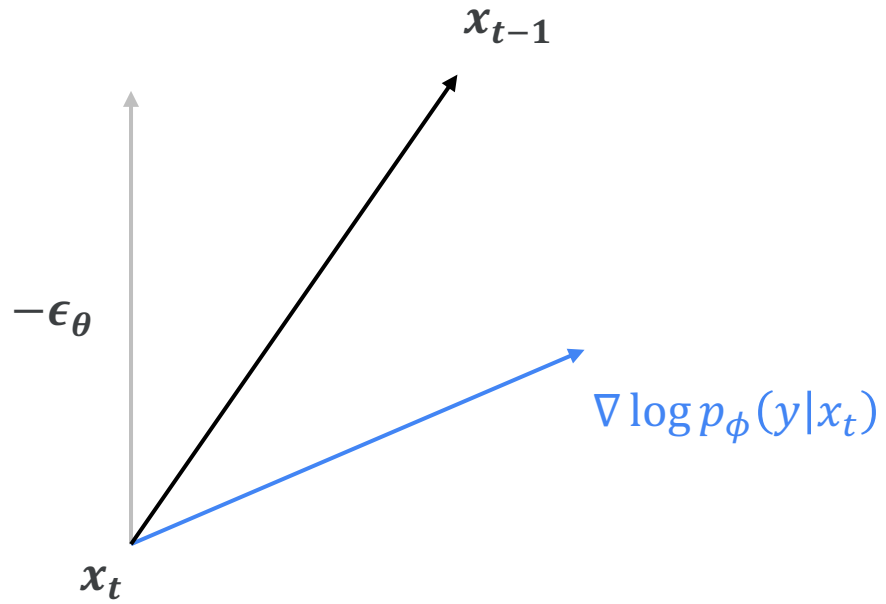
Note

**Algorithm 1** Classifier guided diffusion sampling, given a diffusion model $(\mu_\theta(x_t), \Sigma_\theta(x_t))$, classifier $p_\phi(y|x_t)$, and gradient scale $s$.

Input: class label $y$, gradient scale $s$
$x_T \leftarrow$ sample from $\mathcal{N}(0, \mathbf{I})$
**for all** $t$ from $T$ to 1 **do**
  $\mu, \Sigma \leftarrow \mu_\theta(x_t), \Sigma_\theta(x_t)$
  $x_{t-1} \leftarrow$ sample from $\mathcal{N}(\mu + s\Sigma\nabla_{x_t} \log p_\phi(y|x_t), \Sigma)$
**end for**
**return** $x_0$

**Algorithm 2** Classifier guided DDIM sampling, given a diffusion model $\epsilon_\theta(x_t)$, classifier $p_\phi(y|x_t)$, and gradient scale $s$.

Input: class label $y$, gradient scale $s$
$x_T \leftarrow$ sample from $\mathcal{N}(0, \mathbf{I})$
**for all** $t$ from $T$ to 1 **do**
  $\hat{\epsilon} \leftarrow \epsilon_\theta(x_t) - \sqrt{1-\bar{\alpha}_t}\nabla_{x_t} \log p_\phi(y|x_t)$
  $x_{t-1} \leftarrow \sqrt{\bar{\alpha}_{t-1}}\left(\frac{x_t - \sqrt{1-\bar{\alpha}_t}\hat{\epsilon}}{\sqrt{\bar{\alpha}_t}}\right) + \sqrt{1-\bar{\alpha}_{t-1}}\hat{\epsilon}$
**end for**
**return** $x_0$



# Guided diffusion | Classifier guidance

NAVER AI LAB

- Overview
  - Truncation trick
    - Classifier guidance
      - $x_{t-1} \leftarrow N\left(\mu + s\Sigma\nabla \log p_\phi(y|x_t), \Sigma\right)$

Note

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

$$T_f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n = f(a) + f'(a)(x-a)$$

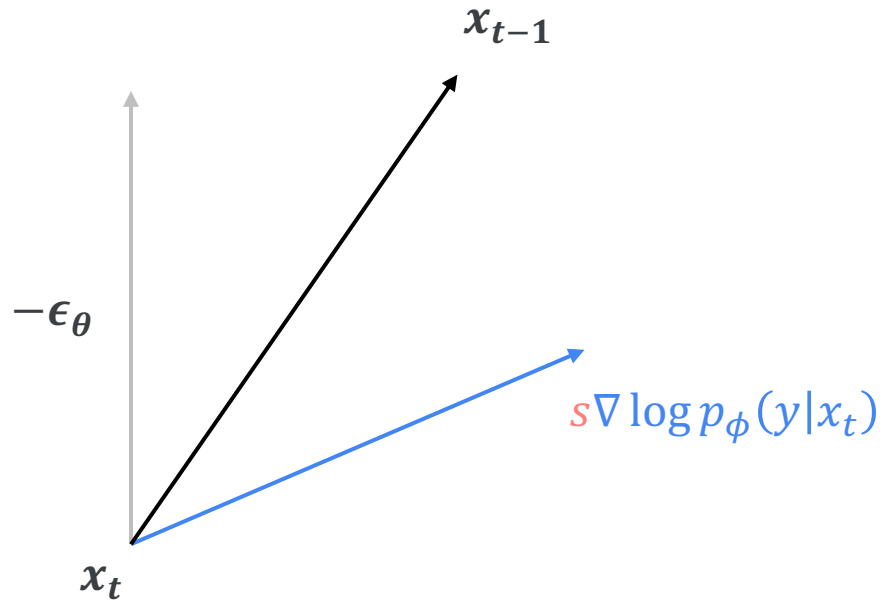
Note

**Algorithm 1** Classifier guided diffusion sampling, given a diffusion model $(\mu_\theta(x_t), \Sigma_\theta(x_t))$, classifier $p_\phi(y|x_t)$, and gradient scale $s$.

Input: class label $y$, gradient scale $s$
$x_T \leftarrow$ sample from $\mathcal{N}(0, \mathbf{I})$
**for all** $t$ from $T$ to 1 **do**
  $\mu, \Sigma \leftarrow \mu_\theta(x_t), \Sigma_\theta(x_t)$
  $x_{t-1} \leftarrow$ sample from $\mathcal{N}(\mu + s\Sigma\,\nabla_{x_t} \log p_\phi(y|x_t), \Sigma)$
**end for**
**return** $x_0$

**Algorithm 2** Classifier guided DDIM sampling, given a diffusion model $\epsilon_\theta(x_t)$, classifier $p_\phi(y|x_t)$, and gradient scale $s$.

Input: class label $y$, gradient scale $s$
$x_T \leftarrow$ sample from $\mathcal{N}(0, \mathbf{I})$
**for all** $t$ from $T$ to 1 **do**
  $\hat{\epsilon} \leftarrow \epsilon_\theta(x_t) - \sqrt{1-\bar{\alpha}_t}\,\nabla_{x_t} \log p_\phi(y|x_t)$
  $x_{t-1} \leftarrow \sqrt{\bar{\alpha}_{t-1}}\left(\frac{x_t - \sqrt{1-\bar{\alpha}_t}\hat{\epsilon}}{\sqrt{\bar{\alpha}_t}}\right) + \sqrt{1-\bar{\alpha}_{t-1}}\hat{\epsilon}$
**end for**
**return** $x_0$



# Guided diffusion | Classifier guidance

NAVER AI LAB

- Overview
  - Truncation trick
    - Classifier guidance
      - $x_{t-1} \leftarrow N(\mu + s\Sigma\nabla \log p_\phi(y|x_t), \Sigma)$

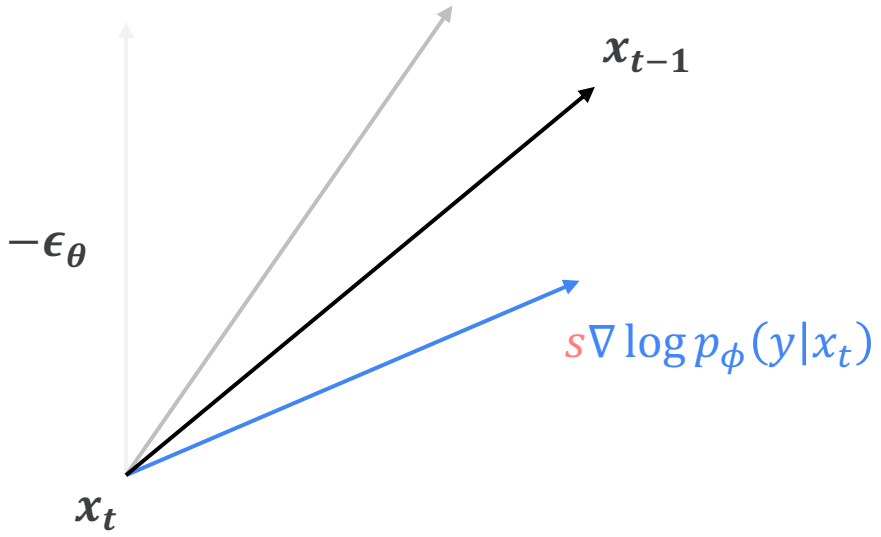
Note

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

$$T_f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n = f(a) + f'(a)(x-a)$$

Note

Figure 3: Samples from an unconditional diffusion model with classifier guidance to condition on the class "Pembroke Welsh corgi". Using classifier scale 1.0 (left; FID: 33.0) does not produce convincing samples in this class, whereas classifier scale 10.0 (right; FID: 12.0) produces much more class-consistent images.



# Guided diffusion | Classifier guidance

NAVER AI LAB

- Overview
  - Truncation trick
    - Classifier guidance
      - $x_{t-1} \leftarrow N(\mu + s\Sigma\nabla \log p_\phi(y|x_t), \Sigma)$

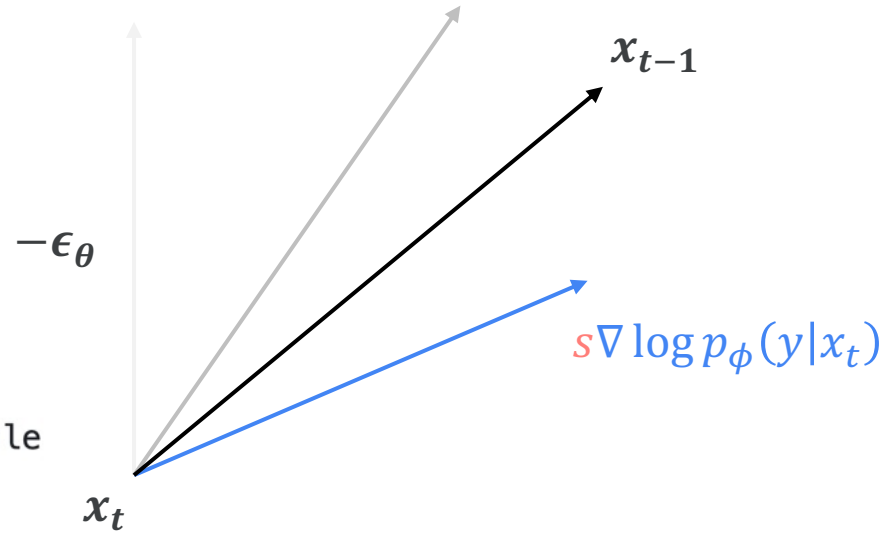
Note

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

$$T_f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n = f(a) + f'(a)(x-a)$$

Note

```
def cond_fn(x, t, y=None):
    assert y is not None
    with th.enable_grad():
        x_in = x.detach().requires_grad_(True)
        logits = classifier(x_in, t)
        log_probs = F.log_softmax(logits, dim=-1)
        selected = log_probs[range(len(logits)), y.view(-1)]
        return th.autograd.grad(selected.sum(), x_in)[0] * args.classifier_scale
```



### 3.1 DDPM Sampling with Manifold Constraint

In DDPMs (Ho et al., 2020), starting from a clean image $\boldsymbol{x}_0 \sim q(\boldsymbol{x}_0)$, a forward diffusion process $q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$ is described as a Markov chain that gradually adds Gaussian noise at every time steps $t$:

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

where $\{\beta\}_{t=0}^{T}$ is a variance schedule. By denoting $\alpha_t := 1 - \beta_t$ and $\bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$, the forward diffused sample at $t$, i.e. $\boldsymbol{x}_t$, can be sampled in one step as:

$$\boldsymbol{x}_t = \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, \quad \text{where} \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}). \tag{2}$$

As the reverse of the forward step $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ is intractable, DDPM learns to maximize the variational lowerbound through a parameterized Gaussian transitions $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ with the parameter $\theta$. Accordingly, the reverse process is approximated as Markov chain with learned mean and fixed variance, starting from $p(\boldsymbol{x}_T) = \mathcal{N}(\boldsymbol{x}_T; \boldsymbol{0}, \boldsymbol{I})$:

$$p_\theta(\boldsymbol{x}_{0:T}) := p_\theta(\boldsymbol{x}_T) \prod_{t=1}^{T} p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t), \quad \text{where} \quad p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) := \mathcal{N}(\boldsymbol{x}_{t-1}; \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t), \sigma_t^2 \boldsymbol{I}). \tag{3}$$

where

$$\boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) := \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\right), \tag{4}$$

Here, $\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)$ is the diffusion model trained by optimizing the objective:

$$\min_\theta L(\theta), \quad \text{where} \quad L(\theta) := \mathbb{E}_{t, \boldsymbol{x}_0, \boldsymbol{\epsilon}}\left[\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)\|^2\right]. \tag{5}$$

After the optimization, by plugging learned score function into the generative (or reverse) diffusion process, one can simply sample from $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ by

$$\boldsymbol{x}_{t-1} = \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) + \sigma_t \boldsymbol{\epsilon} = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\right) + \sigma_t \boldsymbol{\epsilon} \tag{6}$$

In image translation using *conditional* diffusion models (Saharia et al., 2022a; Sasaki et al., 2021), the diffusion model $\boldsymbol{\epsilon}_\theta$ in (5) and (6) should be replaced with $\boldsymbol{\epsilon}_\theta(\boldsymbol{y}, \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)$ where $\boldsymbol{y}$ denotes the matched target image. Accordingly, the sample generation is tightly controlled by the matched target in a supervised manner, so that the image content change rarely happen. Unfortunately, the requirement of the *matched* targets for the training makes this approach impractical.

To address this, Dhariwal & Nichol (2021) proposed classifier-guided image translation using the unconditional diffusion model training as in (5) and a pre-trained classifier $p_\phi(\boldsymbol{y}|\boldsymbol{x}_t)$. Specifically, $\boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t)$ in (4) and (6) are supplemented with the gradient of the classifier, i.e. $\hat{\mu}_\theta(\boldsymbol{x}_t, t) := \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) + \sigma_t \nabla_{\boldsymbol{x}_t} \log p_\phi(\boldsymbol{y}|\boldsymbol{x}_t)$. However, most of the classifiers, which should be separately trained, are not usually sufficient to control the content of the samples from the reverse diffusion process.

### 3.1 DDPM Sampling with Manifold Constraint

In DDPMs (Ho et al., 2020), starting from a clean image $\boldsymbol{x}_0 \sim q(\boldsymbol{x}_0)$, a forward diffusion process $q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$ is described as a Markov chain that gradually adds Gaussian noise at every time steps $t$:

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

where $\{\beta\}_{t=0}^{T}$ is a variance schedule. By denoting $\alpha_t := 1 - \beta_t$ and $\bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$, the forward diffused sample at $t$, i.e. $\boldsymbol{x}_t$, can be sampled in one step as:

$$\boldsymbol{x}_t = \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, \quad \text{where} \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}). \tag{2}$$

As the reverse of the forward step $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ is intractable, DDPM learns to maximize the variational lowerbound through a parameterized Gaussian transitions $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ with the parameter $\theta$. Accordingly, the reverse process is approximated as Markov chain with learned mean and fixed variance, starting from $p(\boldsymbol{x}_T) = \mathcal{N}(\boldsymbol{x}_T; \boldsymbol{0}, \boldsymbol{I})$:

$$p_\theta(\boldsymbol{x}_{0:T}) := p_\theta(\boldsymbol{x}_T) \prod_{t=1}^{T} p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t), \quad \text{where} \quad p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) := \mathcal{N}(\boldsymbol{x}_{t-1}; \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t), \sigma_t^2 \boldsymbol{I}). \tag{3}$$

where

$$\boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) := \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\right), \tag{4}$$

Here, $\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)$ is the diffusion model trained by optimizing the objective:

$$\min_\theta L(\theta), \quad \text{where} \quad L(\theta) := \mathbb{E}_{t, \boldsymbol{x}_0, \boldsymbol{\epsilon}}\left[\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)\|^2\right]. \tag{5}$$

After the optimization, by plugging learned score function into the generative (or reverse) diffusion process, one can simply sample from $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ by

$$\boldsymbol{x}_{t-1} = \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) + \sigma_t \boldsymbol{\epsilon} = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\right) + \sigma_t \boldsymbol{\epsilon} \tag{6}$$

In image translation using *conditional* diffusion models (Saharia et al., 2022a; Sasaki et al., 2021), the diffusion model $\boldsymbol{\epsilon}_\theta$ in (5) and (6) should be replaced with $\boldsymbol{\epsilon}_\theta(\boldsymbol{y}, \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)$ where $\boldsymbol{y}$ denotes the matched target image. Accordingly, the sample generation is tightly controlled by the matched target in a supervised manner, so that the image content change rarely happen. Unfortunately, the requirement of the *matched* targets for the training makes this approach impractical.

To address this, Dhariwal & Nichol (2021) proposed classifier-guided image translation using the unconditional diffusion model training as in (5) and a pre-trained classifier $p_\phi(\boldsymbol{y}|\boldsymbol{x}_t)$. Specifically, $\boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t)$ in (4) and (6) are supplemented with the gradient of the classifier, i.e. $\hat{\mu}_\theta(\boldsymbol{x}_t, t) := \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) + \sigma_t \nabla_{\boldsymbol{x}_t} \log p_\phi(\boldsymbol{y}|\boldsymbol{x}_t)$. However, most of the classifiers, which should be separately trained, are not usually sufficient to control the content of the samples from the reverse diffusion process.

forward 함수네
(noise를 더하는)

### 3.1 DDPM Sampling with Manifold Constraint

In DDPMs (Ho et al., 2020), starting from a clean image $\boldsymbol{x}_0 \sim q(\boldsymbol{x}_0)$, a forward diffusion process $q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$ is described as a Markov chain that gradually adds Gaussian noise at every time steps $t$:

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

where $\{\beta\}_{t=0}^{T}$ is a variance schedule. By denoting $\alpha_t := 1 - \beta_t$ and $\bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$, the forward diffused sample at $t$, i.e. $\boldsymbol{x}_t$, can be sampled in one step as:

$$\boldsymbol{x}_t = \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, \quad \text{where} \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}). \tag{2}$$

As the reverse of the forward step $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ is intractable, DDPM learns to maximize the variational lowerbound through a parameterized Gaussian transitions $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ with the parameter $\theta$. Accordingly, the reverse process is approximated as Markov chain with learned mean and fixed variance, starting from $p(\boldsymbol{x}_T) = \mathcal{N}(\boldsymbol{x}_T; \boldsymbol{0}, \boldsymbol{I})$:

$$p_\theta(\boldsymbol{x}_{0:T}) := p_\theta(\boldsymbol{x}_T) \prod_{t=1}^{T} p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t), \quad \text{where} \quad p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) := \mathcal{N}(\boldsymbol{x}_{t-1}; \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t), \sigma_t^2 \boldsymbol{I}). \tag{3}$$

where

$$\boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) := \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\right), \tag{4}$$

Here, $\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)$ is the diffusion model trained by optimizing the objective:

$$\min_\theta L(\theta), \quad \text{where} \quad L(\theta) := \mathbb{E}_{t, \boldsymbol{x}_0, \boldsymbol{\epsilon}}\left[\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)\|^2\right]. \tag{5}$$

After the optimization, by plugging learned score function into the generative (or reverse) diffusion process, one can simply sample from $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ by

$$\boldsymbol{x}_{t-1} = \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) + \sigma_t \boldsymbol{\epsilon} = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\right) + \sigma_t \boldsymbol{\epsilon} \tag{6}$$

In image translation using *conditional* diffusion models (Saharia et al., 2022a; Sasaki et al., 2021), the diffusion model $\boldsymbol{\epsilon}_\theta$ in (5) and (6) should be replaced with $\boldsymbol{\epsilon}_\theta(\boldsymbol{y}, \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)$ where $\boldsymbol{y}$ denotes the matched target image. Accordingly, the sample generation is tightly controlled by the matched target in a supervised manner, so that the image content change rarely happen. Unfortunately, the requirement of the *matched* targets for the training makes this approach impractical.

To address this, Dhariwal & Nichol (2021) proposed classifier-guided image translation using the unconditional diffusion model training as in (5) and a pre-trained classifier $p_\phi(\boldsymbol{y}|\boldsymbol{x}_t)$. Specifically, $\boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t)$ in (4) and (6) are supplemented with the gradient of the classifier, i.e. $\hat{\mu}_\theta(\boldsymbol{x}_t, t) := \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) + \sigma_t \nabla_{\boldsymbol{x}_t} \log p_\phi(\boldsymbol{y}|\boldsymbol{x}_t)$. However, most of the classifiers, which should be separately trained, are not usually sufficient to control the content of the samples from the reverse diffusion process.

Loss를 구하는 과정이네

### 3.1 DDPM Sampling with Manifold Constraint

In DDPMs (Ho et al., 2020), starting from a clean image $\boldsymbol{x}_0 \sim q(\boldsymbol{x}_0)$, a forward diffusion process $q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$ is described as a Markov chain that gradually adds Gaussian noise at every time steps $t$:

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

where $\{\beta\}_{t=0}^{T}$ is a variance schedule. By denoting $\alpha_t := 1 - \beta_t$ and $\bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$, the forward diffused sample at $t$, i.e. $\boldsymbol{x}_t$, can be sampled in one step as:

$$\boldsymbol{x}_t = \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, \quad \text{where} \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}). \tag{2}$$

As the reverse of the forward step $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ is intractable, DDPM learns to maximize the variational lowerbound through a parameterized Gaussian transitions $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ with the parameter $\theta$. Accordingly, the reverse process is approximated as Markov chain with learned mean and fixed variance, starting from $p(\boldsymbol{x}_T) = \mathcal{N}(\boldsymbol{x}_T; \boldsymbol{0}, \boldsymbol{I})$:

$$p_\theta(\boldsymbol{x}_{0:T}) := p_\theta(\boldsymbol{x}_T) \prod_{t=1}^{T} p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t), \quad \text{where} \quad p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) := \mathcal{N}(\boldsymbol{x}_{t-1}; \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t), \sigma_t^2 \boldsymbol{I}). \tag{3}$$

where

$$\boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) := \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\right), \tag{4}$$

Here, $\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)$ is the diffusion model trained by optimizing the objective:

$$\min_\theta L(\theta), \quad \text{where} \quad L(\theta) := \mathbb{E}_{t, \boldsymbol{x}_0, \boldsymbol{\epsilon}}\left[\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)\|^2\right]. \tag{5}$$

After the optimization, by plugging learned score function into the generative (or reverse) diffusion process, one can simply sample from $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ by

$$\boldsymbol{x}_{t-1} = \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) + \sigma_t \boldsymbol{\epsilon} = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\right) + \sigma_t \boldsymbol{\epsilon} \tag{6}$$

In image translation using *conditional* diffusion models (Saharia et al., 2022a; Sasaki et al., 2021), the diffusion model $\boldsymbol{\epsilon}_\theta$ in (5) and (6) should be replaced with $\boldsymbol{\epsilon}_\theta(\boldsymbol{y}, \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)$ where $\boldsymbol{y}$ denotes the matched target image. Accordingly, the sample generation is tightly controlled by the matched target in a supervised manner, so that the image content change rarely happen. Unfortunately, the requirement of the *matched* targets for the training makes this approach impractical.

To address this, Dhariwal & Nichol (2021) proposed classifier-guided image translation using the unconditional diffusion model training as in (5) and a pre-trained classifier $p_\phi(\boldsymbol{y}|\boldsymbol{x}_t)$. Specifically, $\boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t)$ in (4) and (6) are supplemented with the gradient of the classifier, i.e. $\hat{\mu}_\theta(\boldsymbol{x}_t, t) := \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) + \sigma_t \nabla_{\boldsymbol{x}_t} \log p_\phi(\boldsymbol{y}|\boldsymbol{x}_t)$. However, most of the classifiers, which should be separately trained, are not usually sufficient to control the content of the samples from the reverse diffusion process.

reverse 함수네
(noise를 빼는)

NAVER AI LAB

### 3.1 DDPM Sampling with Manifold Constraint

In DDPMs (Ho et al., 2020), starting from a clean image $\boldsymbol{x}_0 \sim q(\boldsymbol{x}_0)$, a forward diffusion process $q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$ is described as a Markov chain that gradually adds Gaussian noise at every time steps $t$:

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

where $\{\beta\}_{t=0}^{T}$ is a variance schedule. By denoting $\alpha_t := 1 - \beta_t$ and $\bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$, the forward diffused sample at $t$, i.e. $\boldsymbol{x}_t$, can be sampled in one step as:

$$\boldsymbol{x}_t = \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, \quad \text{where} \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}). \tag{2}$$

As the reverse of the forward step $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ is intractable, DDPM learns to maximize the variational lowerbound through a parameterized Gaussian transitions $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ with the parameter $\theta$. Accordingly, the reverse process is approximated as Markov chain with learned mean and fixed variance, starting from $p(\boldsymbol{x}_T) = \mathcal{N}(\boldsymbol{x}_T; \boldsymbol{0}, \boldsymbol{I})$:

$$p_\theta(\boldsymbol{x}_{0:T}) := p_\theta(\boldsymbol{x}_T) \prod_{t=1}^{T} p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t), \quad \text{where} \quad p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) := \mathcal{N}(\boldsymbol{x}_{t-1}; \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t), \sigma_t^2 \boldsymbol{I}). \tag{3}$$

where

$$\boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) := \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\right), \tag{4}$$

Here, $\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)$ is the diffusion model trained by optimizing the objective:

$$\min_\theta L(\theta), \quad \text{where} \quad L(\theta) := \mathbb{E}_{t, \boldsymbol{x}_0, \boldsymbol{\epsilon}}\left[\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)\|^2\right]. \tag{5}$$

After the optimization, by plugging learned score function into the generative (or reverse) diffusion process, one can simply sample from $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ by

$$\boldsymbol{x}_{t-1} = \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) + \sigma_t \boldsymbol{\epsilon} = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\right) + \sigma_t \boldsymbol{\epsilon} \tag{6}$$

In image translation using *conditional* diffusion models (Saharia et al., 2022a; Sasaki et al., 2021), the diffusion model $\boldsymbol{\epsilon}_\theta$ in (5) and (6) should be replaced with $\boldsymbol{\epsilon}_\theta(\boldsymbol{y}, \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)$ where $\boldsymbol{y}$ denotes the matched target image. Accordingly, the sample generation is tightly controlled by the matched target in a supervised manner, so that the image content change rarely happen. Unfortunately, the requirement of the *matched* targets for the training makes this approach impractical.

To address this, Dhariwal & Nichol (2021) proposed classifier-guided image translation using the unconditional diffusion model training as in (5) and a pre-trained classifier $p_\phi(\boldsymbol{y}|\boldsymbol{x}_t)$. Specifically, $\boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t)$ in (4) and (6) are supplemented with the gradient of the classifier, i.e. $\hat{\mu}_\theta(\boldsymbol{x}_t, t) := \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) + \sigma_t \nabla_{\boldsymbol{x}_t} \log p_\phi(\boldsymbol{y}|\boldsymbol{x}_t)$. However, most of the classifiers, which should be separately trained, are not usually sufficient to control the content of the samples from the reverse diffusion process.

classifier guidance끼

NAVER AI LAB

### 3.1 DDPM Sampling with Manifold Constraint

In DDPMs (Ho et al., 2020), starting from a clean image $\boldsymbol{x}_0 \sim q(\boldsymbol{x}_0)$, a forward diffusion process $q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$ is described as a Markov chain that gradually adds Gaussian noise at every time steps $t$:

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

where $\{\beta\}_{t=0}^{T}$ is a variance schedule. By denoting $\alpha_t := 1 - \beta_t$ and $\bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$, the forward diffused sample at $t$, i.e. $\boldsymbol{x}_t$, can be sampled in one step as:

$$\boldsymbol{x}_t = \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, \quad \text{where} \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}). \tag{2}$$

As the reverse of the forward step $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ is intractable, DDPM learns to maximize the variational lowerbound through a parameterized Gaussian transitions $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ with the parameter $\theta$. Accordingly, the reverse process is approximated as Markov chain with learned mean and fixed variance, starting from $p(\boldsymbol{x}_T) = \mathcal{N}(\boldsymbol{x}_T; \boldsymbol{0}, \boldsymbol{I})$:

$$p_\theta(\boldsymbol{x}_{0:T}) := p_\theta(\boldsymbol{x}_T) \prod_{t=1}^{T} p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t), \quad \text{where} \quad p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) := \mathcal{N}(\boldsymbol{x}_{t-1}; \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t), \sigma_t^2 \boldsymbol{I}). \tag{3}$$

where

$$\boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) := \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\right), \tag{4}$$

Here, $\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)$ is the diffusion model trained by optimizing the objective:

$$\min_\theta L(\theta), \quad \text{where} \quad L(\theta) := \mathbb{E}_{t, \boldsymbol{x}_0, \boldsymbol{\epsilon}}\left[\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)\|^2\right]. \tag{5}$$

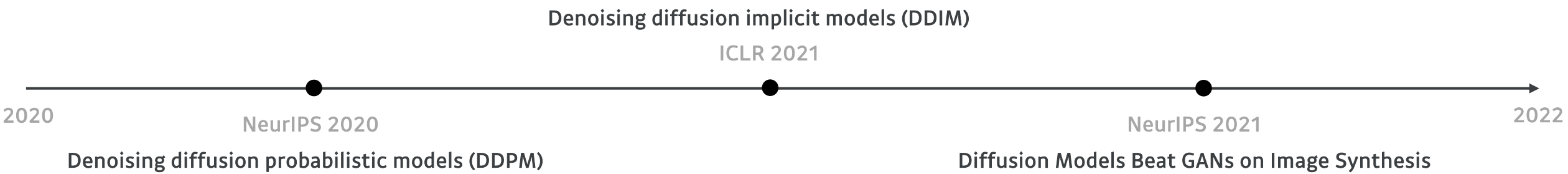
After the optimization, by plugging learned score function into the generative (or reverse) diffusion process, one can simply sample from $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ by

$$\boldsymbol{x}_{t-1} = \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) + \sigma_t \boldsymbol{\epsilon} = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\right) + \sigma_t \boldsymbol{\epsilon} \tag{6}$$

In image translation using *conditional* diffusion models (Saharia et al., 2022a; Sasaki et al., 2021), the diffusion model $\boldsymbol{\epsilon}_\theta$ in (5) and (6) should be replaced with $\boldsymbol{\epsilon}_\theta(\boldsymbol{y}, \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)$ where $\boldsymbol{y}$ denotes the matched target image. Accordingly, the sample generation is tightly controlled by the matched target in a supervised manner, so that the image content change rarely happen. Unfortunately, the requirement of the *matched* targets for the training makes this approach impractical.

To address this, Dhariwal & Nichol (2021) proposed classifier-guided image translation using the unconditional diffusion model training as in (5) and a pre-trained classifier $p_\phi(\boldsymbol{y}|\boldsymbol{x}_t)$. Specifically, $\boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t)$ in (4) and (6) are supplemented with the gradient of the classifier, i.e. $\hat{\mu}_\theta(\boldsymbol{x}_t, t) := \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) + \sigma_t \nabla_{\boldsymbol{x}_t} \log p_\phi(\boldsymbol{y}|\boldsymbol{x}_t)$. However, most of the classifiers, which should be separately trained, are not usually sufficient to control the content of the samples from the reverse diffusion process.

# Summary


Denoising diffusion implicit models (DDIM)
ICLR 2021
2020
NeurIPS 2020
NeurIPS 2021
2022
Denoising diffusion probabilistic models (DDPM)
Diffusion Models Beat GANs on Image Synthesis

# Summary

NAVER AI LAB

2020

NeurIPS 2020

**Denoising diffusion probabilistic models (DDPM)**

- $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon,\ \epsilon \sim \mathcal{N}(0, \mathbf{I})$ (Forward)
  - $\beta_1 = 10^{-4}, \beta_T = 0.02$
  - $\alpha_t := 1 - \beta_t,\ \bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$
- $\epsilon - \epsilon_\theta(\mathbf{x}_t) = \epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon)$ (Loss)
  - $\epsilon_\theta$ = prediction network
- $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\mathbf{x}_t, t)\right) + \sqrt{\tilde{\beta}_t}\epsilon,\ \epsilon \sim \mathcal{N}(0, \mathbf{I})$ (Reverse)
  - $\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$
    - $\tilde{\beta}_t = \beta_t$로 해도 성능차이 없음

**Denoising diffusion implicit models (DDIM)**

ICLR 2021

NeurIPS 2021

**Diffusion Models Beat GANs on Image Synthesis**

2022

# Summary

NAVER AI LAB

- DDIM $\alpha$ = DDPM $\bar{\alpha}$
- $\mathbf{x}_t = \sqrt{\alpha_t}\mathbf{x}_0 + \sqrt{1-\alpha_t}\epsilon$ (Forward)
- $\epsilon - \epsilon_\theta(\mathbf{x}_t) = \epsilon - \epsilon_\theta(\sqrt{\alpha_t}\mathbf{x}_0 + \sqrt{1-\alpha_t}\epsilon)$ (Loss)
  - $\epsilon_\theta$ = prediction network
- $\mathbf{x}_{t-1} = \sqrt{\alpha_{t-1}}\underbrace{\left(\frac{\mathbf{x}_t - \sqrt{1-\alpha_t}\epsilon_\theta(\mathbf{x}_t)}{\sqrt{\alpha_t}}\right)}_{\text{predicted } \mathbf{x}_0 = f_\theta(\mathbf{x}_t)} + \underbrace{\sqrt{1-\alpha_{t-1}-\sigma_t^2}\cdot\epsilon_\theta(\mathbf{x}_t)}_{\text{direction pointing to } \mathbf{x}_t} + \underbrace{\sigma_t\epsilon}_{\text{noise}}$ (Reverse)
  - deterministic when $\sigma_t = 0$ → consistency

Denoising diffusion implicit models (DDIM)

ICLR 2021

2020

NeurIPS 2020

Denoising diffusion probabilistic models (DDPM)

- $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon,\ \epsilon \sim \mathcal{N}(0, \mathbf{I})$ (Forward)
  - $\beta_1 = 10^{-4}, \beta_T = 0.02$
  - $\alpha_t := 1 - \beta_t,\ \bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$
- $\epsilon - \epsilon_\theta(\mathbf{x}_t) = \epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon)$ (Loss)
  - $\epsilon_\theta$ = prediction network
- $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\mathbf{x}_t, t)\right) + \sqrt{\tilde{\beta}_t}\epsilon,\ \epsilon \sim \mathcal{N}(0, \mathbf{I})$ (Reverse)
  - $\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$
    - $\tilde{\beta}_t = \beta_t$로 해도 성능차이 없음

NeurIPS 2021

Diffusion Models Beat GANs on Image Synthesis

2022

# Summary

NAVER AI LAB

- DDIM $\alpha$ = DDPM $\bar{\alpha}$
- $\mathbf{x}_t = \sqrt{\alpha_t}\mathbf{x}_0 + \sqrt{1-\alpha_t}\epsilon$ (Forward)
- $\epsilon - \epsilon_\theta(\mathbf{x}_t) = \epsilon - \epsilon_\theta(\sqrt{\alpha_t}\mathbf{x}_0 + \sqrt{1-\alpha_t}\epsilon)$ (Loss)
  - $\epsilon_\theta$ = prediction network
- $\mathbf{x}_{t-1} = \sqrt{\alpha_{t-1}}\underbrace{\left(\frac{\mathbf{x}_t - \sqrt{1-\alpha_t}\epsilon_\theta(\mathbf{x}_t)}{\sqrt{\alpha_t}}\right)}_{\text{predicted } \mathbf{x}_0 = f_\theta(\mathbf{x}_t)} + \underbrace{\sqrt{1-\alpha_{t-1}-\sigma_t^2}\cdot\epsilon_\theta(\mathbf{x}_t)}_{\text{direction pointing to } \mathbf{x}_t} + \underbrace{\sigma_t\epsilon}_{\text{noise}}$ (Reverse)
  - deterministic when $\sigma_t = 0$ → consistency

Denoising diffusion implicit models (DDIM)

ICLR 2021

2020

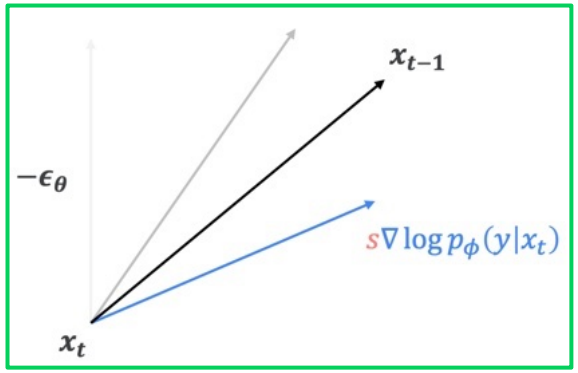
NeurIPS 2020

Denoising diffusion probabilistic models (DDPM)

- $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon,\ \epsilon \sim \mathcal{N}(0, \mathbf{I})$ (Forward)
  - $\beta_1 = 10^{-4}, \beta_T = 0.02$
  - $\alpha_t := 1 - \beta_t,\ \bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$
- $\epsilon - \epsilon_\theta(\mathbf{x}_t) = \epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon)$ (Loss)
  - $\epsilon_\theta$ = prediction network
- $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\mathbf{x}_t, t)\right) + \sqrt{\tilde{\beta}_t}\epsilon,\ \epsilon \sim \mathcal{N}(0, \mathbf{I})$ (Reverse)
  - $\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$
    - $\tilde{\beta}_t = \beta_t$로 해도 성능차이 없음

NeurIPS 2021

Diffusion Models Beat GANs on Image Synthesis




2022

# Grooming Diffusion

## Advanced diffusion

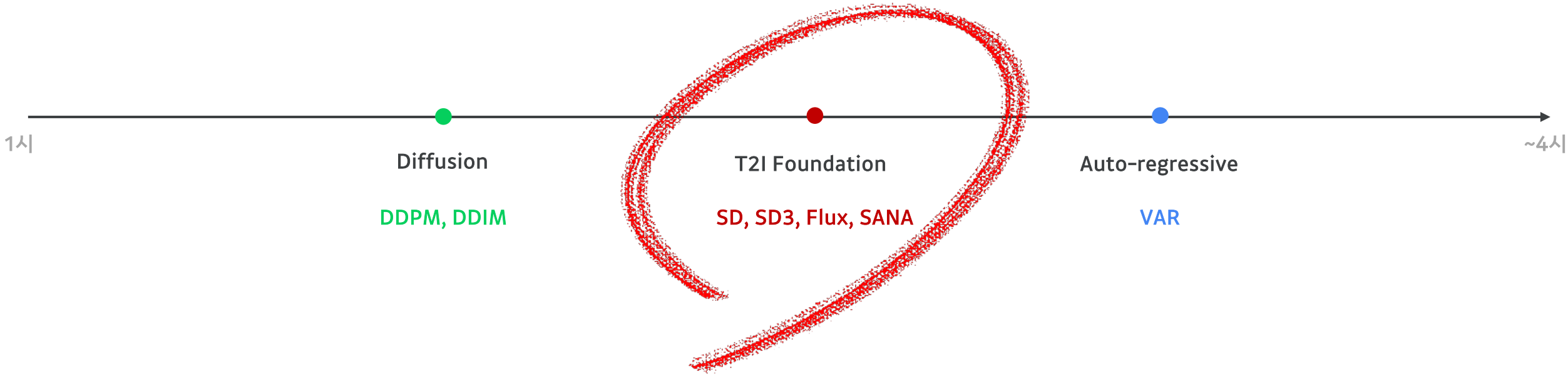
NAVER AI Lab

김준호

https://github.com/taki0112

NAVER AI LAB

NAVER AI LAB


1시
~4시
Diffusion
DDPM, DDIM
T2I Foundation
SD, SD3, Flux, SANA
Auto-regressive
VAR

NAVER AI LAB

# 🤗 Diffusers


🤗 Diffusers provides pretrained diffusion models across multiple modalities, such as vision and audio, and serves as a modular toolbox for inference and training of diffusion models.

More precisely, 🤗 Diffusers offers:

- State-of-the-art diffusion pipelines that can be run in inference with just a couple of lines of code (see src/diffusers/pipelines). Check this overview to see all supported pipelines and their corresponding official papers.
- Various noise schedulers that can be used interchangeably for the preferred speed vs. quality trade-off in inference (see src/diffusers/schedulers).
- Multiple types of models, such as UNet, can be used as building blocks in an end-to-end diffusion system (see src/diffusers/models).
- Training examples to show how to train the most popular diffusion model tasks (see examples, *e.g.* unconditional-image-generation).

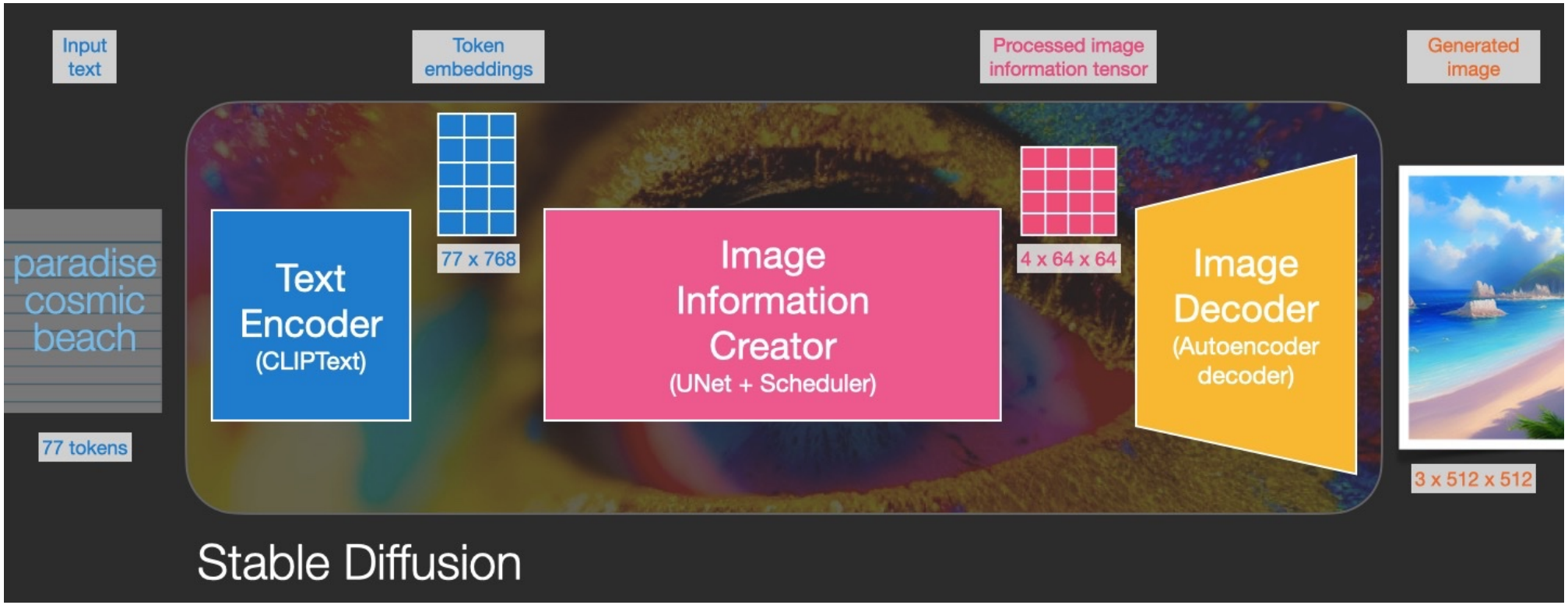
github.com/huggingface/diffusers
github.com/modelscope/DiffSynth-Studio

NAVER AI LAB

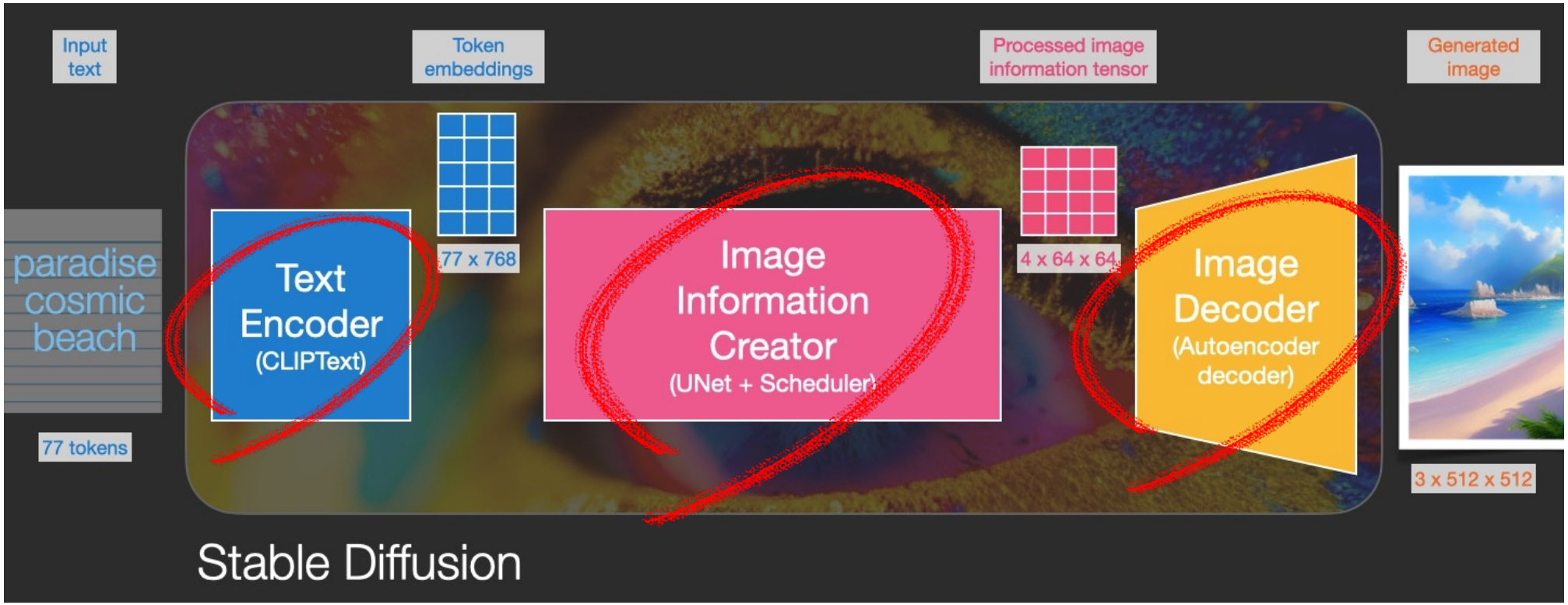

jalammar.github.io/illustrated-stable-diffusion/

# Stable Diffusion | Components

NAVER AI LAB



# Stable Diffusion | Components

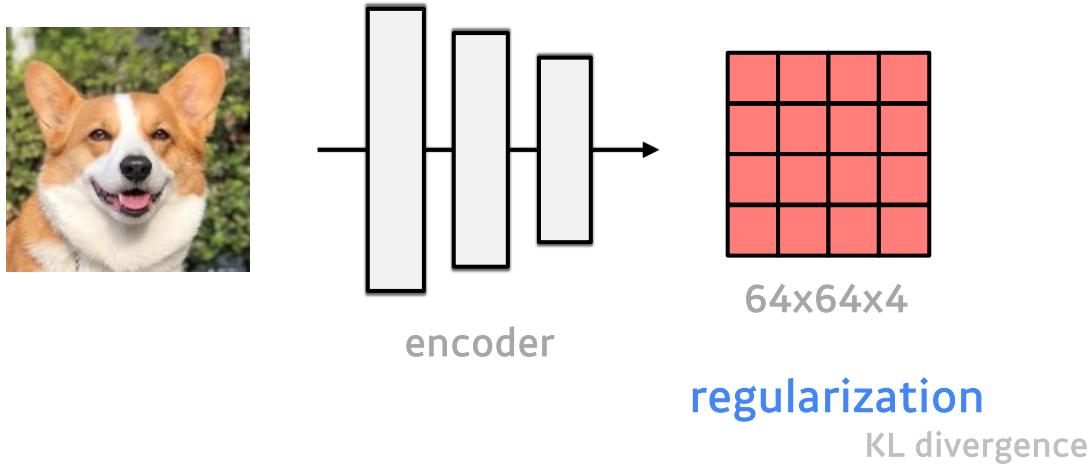


# Stable Diffusion

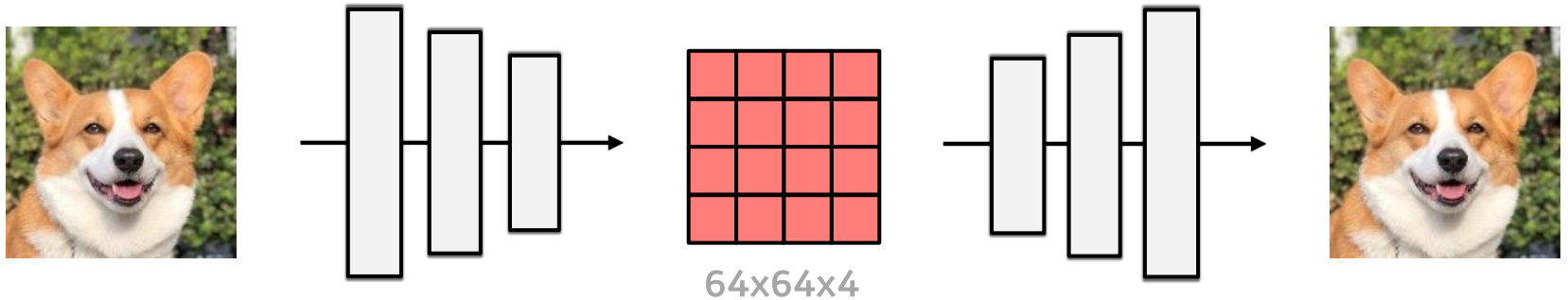
# Stable Diffusion | Autoencoder

NAVER AI LAB

# Stable Diffusion | Autoencoder

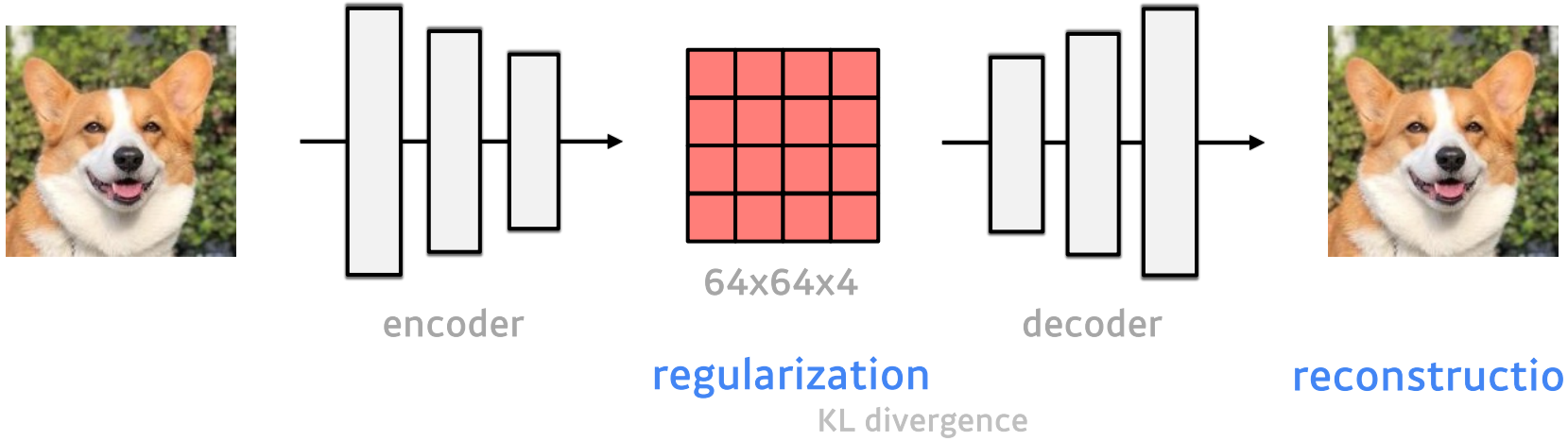


# Stable Diffusion | Autoencoder



# Stable Diffusion | Autoencoder

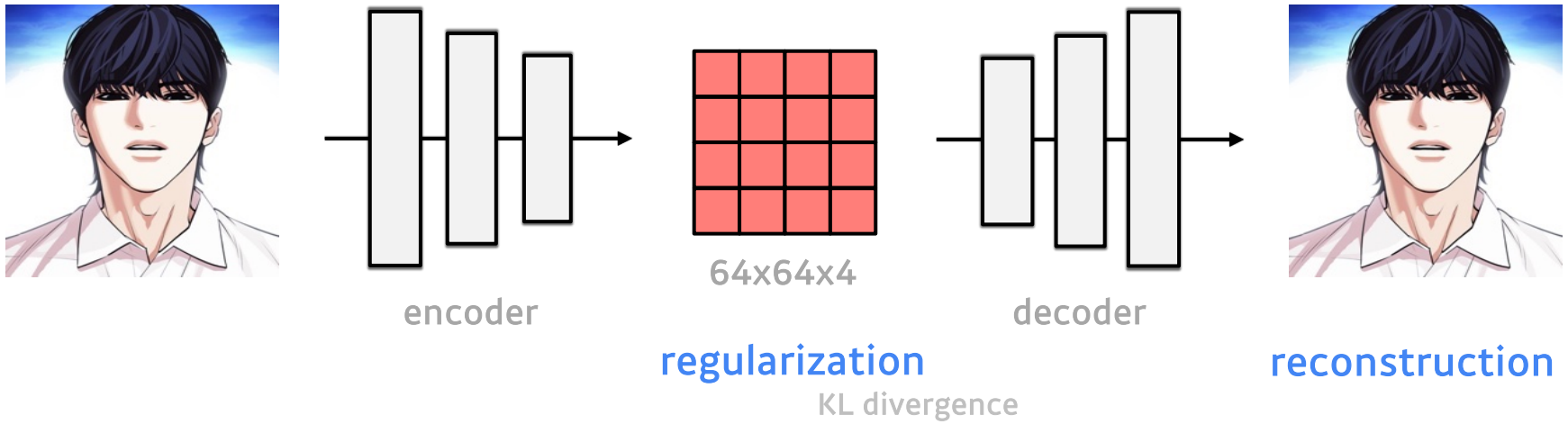
NAVER AI LAB





High-Resolution Image Synthesis with Latent Diffusion Models, CVPR 2022
github.com/rom1504/img2dataset

# Stable Diffusion | Autoencoder





High-Resolution Image Synthesis with Latent Diffusion Models, CVPR 2022
github.com/rom1504/img2dataset

# Stable Diffusion | CLIP

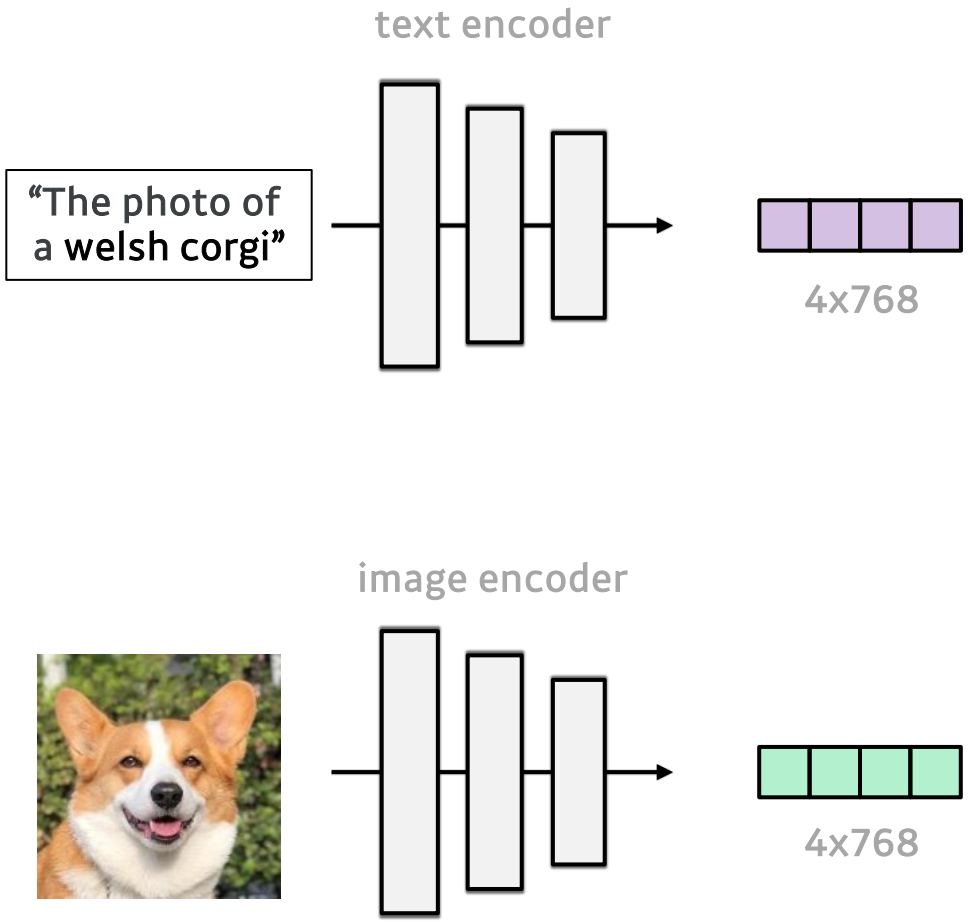
NAVER AI LAB

"The photo of a welsh corgi"

Learning Transferable Visual Models From Natural Language Supervision, ICML 2021

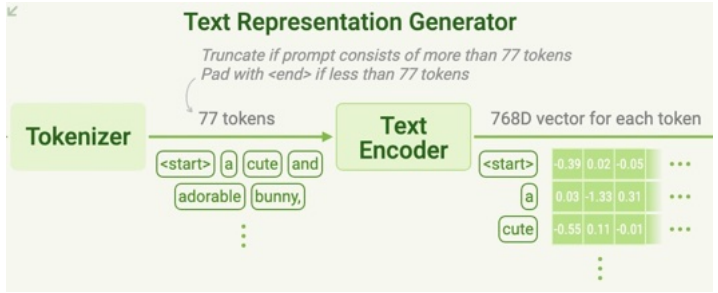
# Stable Diffusion | CLIP

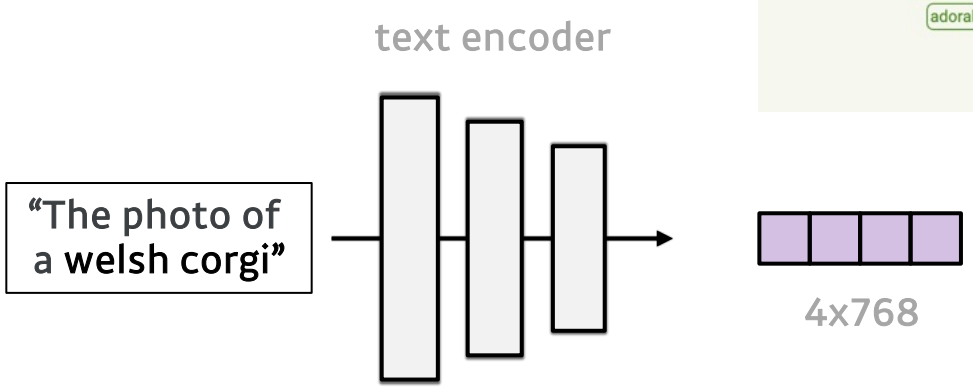
NAVER AI LAB

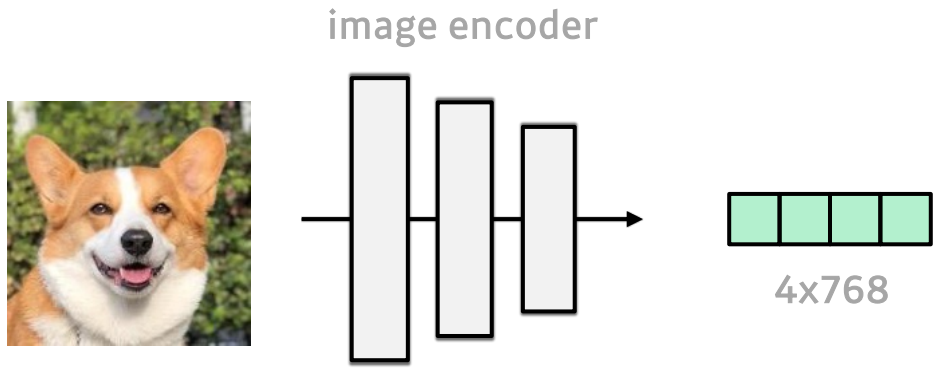



Learning Transferable Visual Models From Natural Language Supervision, ICML 2021

# Stable Diffusion | CLIP

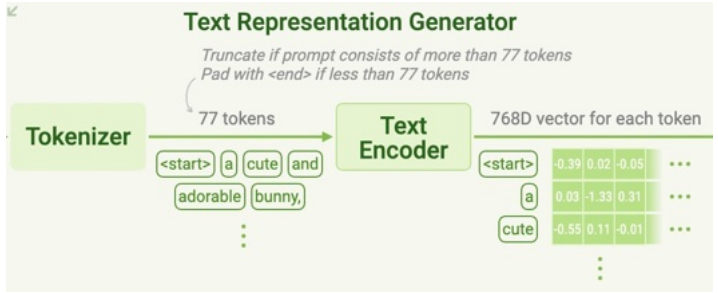


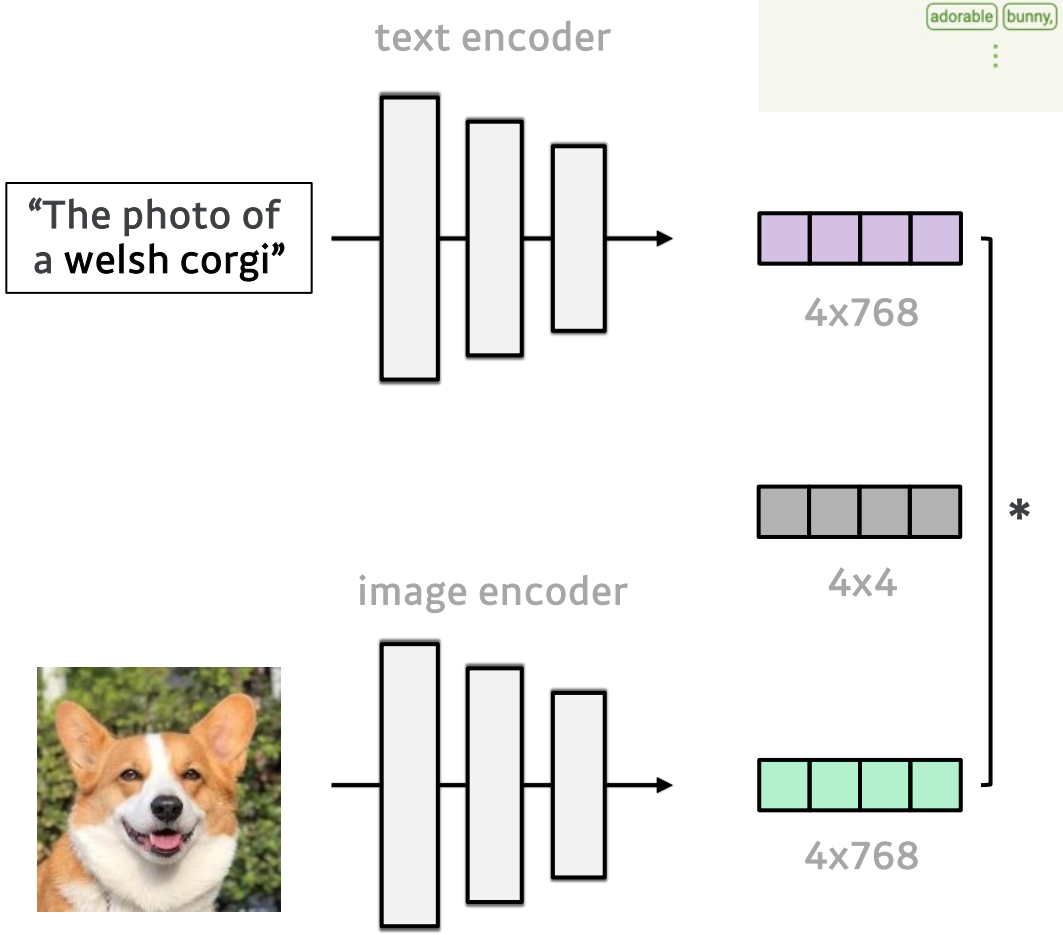



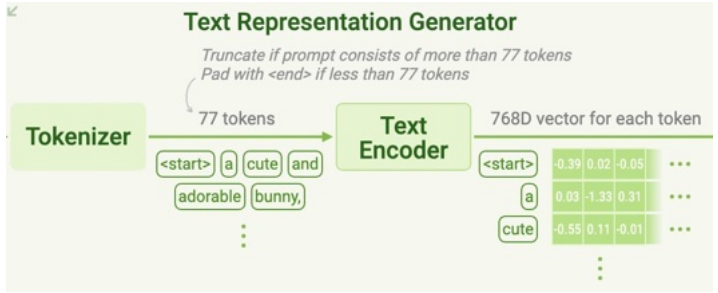

Learning Transferable Visual Models From Natural Language Supervision, ICML 2021

# Stable Diffusion | CLIP



# Stable Diffusion | CLIP




text encoder

"The photo of a welsh corgi"

4x768

text embed

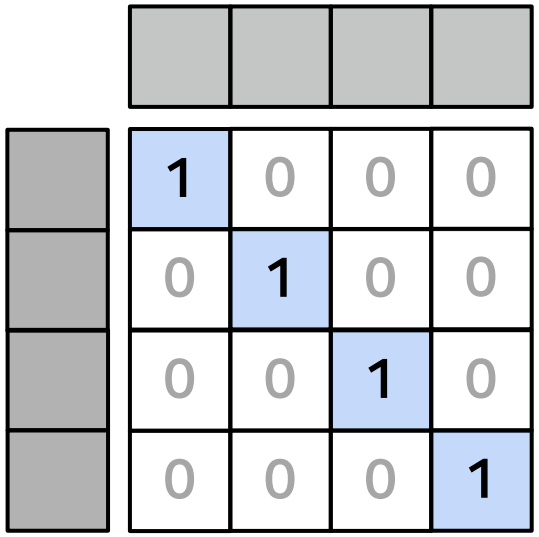



image encoder

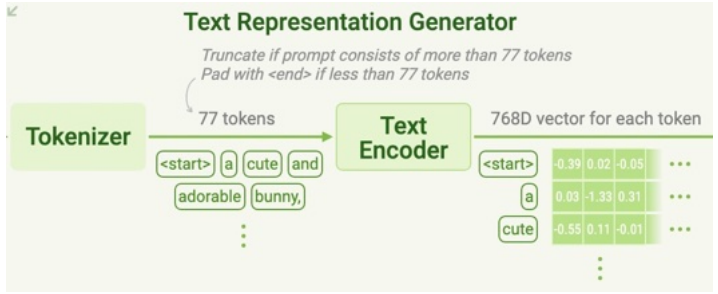
4x4

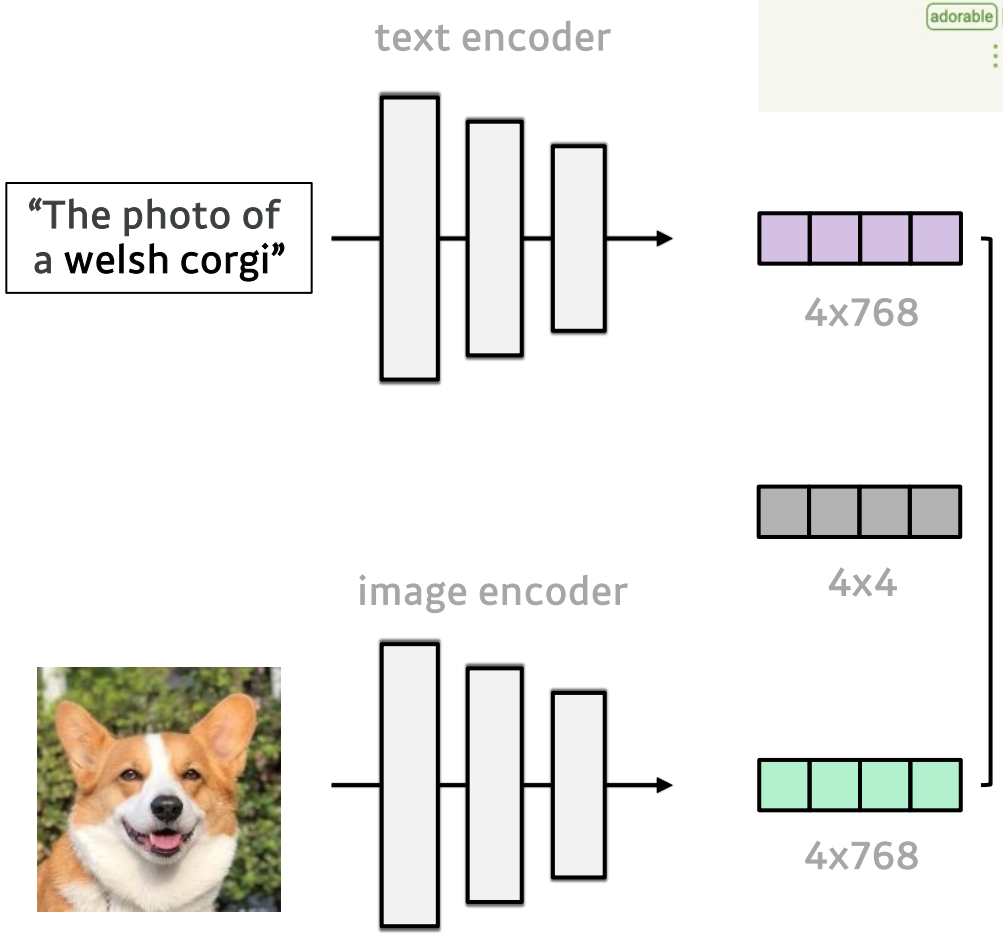
4x768

image embed

# Stable Diffusion | CLIP

NAVER AI LAB





LAION-5B, 84.8TB

2B-en

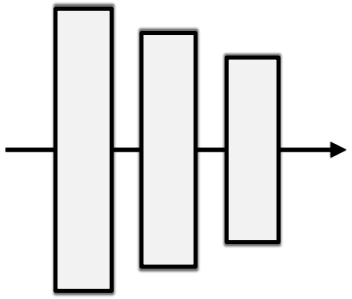
```python
import torch
from PIL import Image
import open_clip

model, _, preprocess = open_clip.create_model_and_transforms('ViT-B-32-quickgelu', pretrained='laio
tokenizer = open_clip.get_tokenizer('ViT-B-32-quickgelu')

image = preprocess(Image.open("CLIP.png")).unsqueeze(0)
text = tokenizer(["a diagram", "a dog", "a cat"])

with torch.no_grad(), torch.cuda.amp.autocast():
    image_features = model.encode_image(image)
    text_features = model.encode_text(text)
    image_features /= image_features.norm(dim=-1, keepdim=True)
    text_features /= text_features.norm(dim=-1, keepdim=True)

    text_probs = (100.0 * image_features @ text_features.T).softmax(dim=-1)

print("Label probs:", text_probs)  # prints: [[1., 0., 0.]]
```

github.com/mlfoundations/open_clip

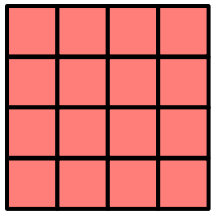
Learning Transferable Visual Models From Natural Language Supervision, ICML 2021

# Stable Diffusion | Training

NAVER AI LAB



encoder



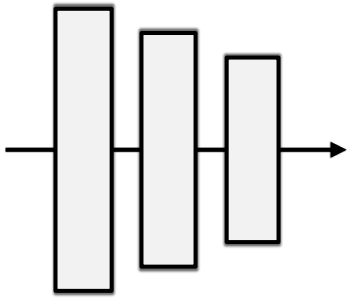
64x64x4

forward process

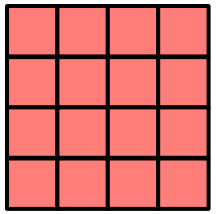
step 1

# Stable Diffusion | Training

NAVER AI LAB



encoder

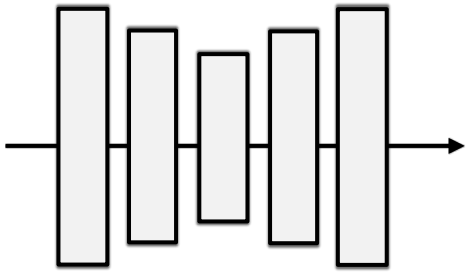


64x64x4

forward process

step 1

time embed



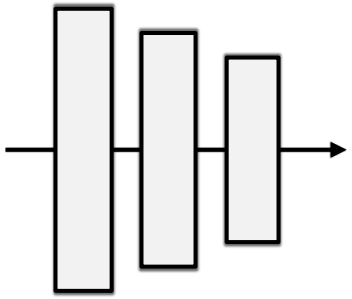
Unet

pred noise

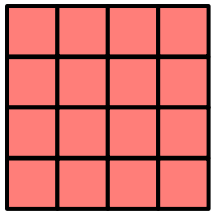
text embed
77x768

# Stable Diffusion | Training

NAVER AI LAB



encoder

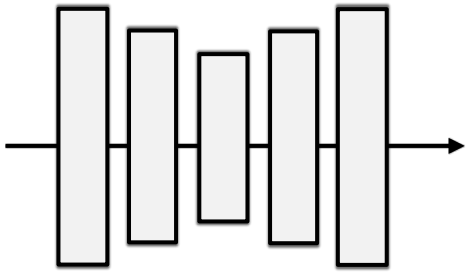


64x64x4

forward process

step 1

time embed



Unet

L2 loss

pred noise

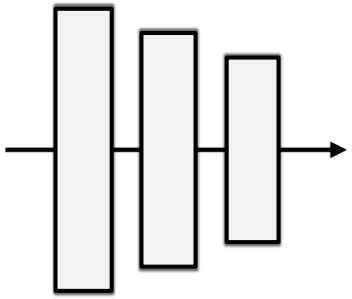
-

GT noise

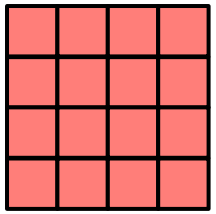
text embed
77x768

# Stable Diffusion | Training

NAVER AI LAB



encoder

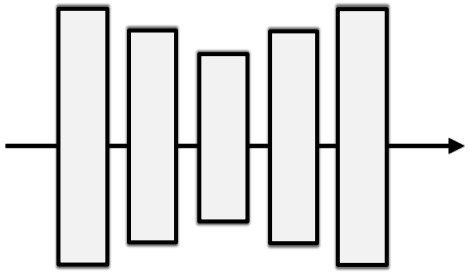


64x64x4

forward process

step 1

time embed



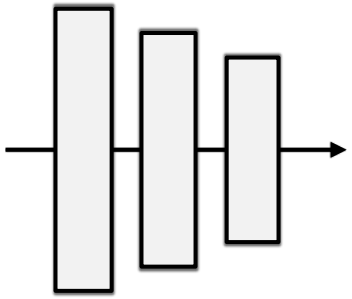
Unet

pred noise

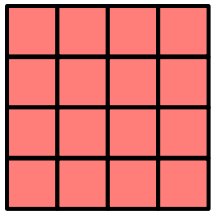
text embed
77x768

# Stable Diffusion | Training

NAVER AI LAB



encoder

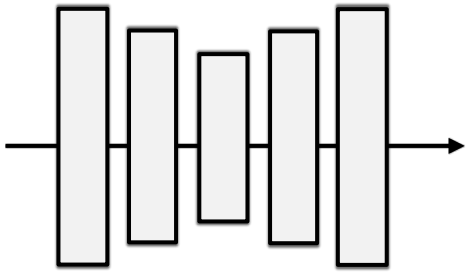


64x64x4

forward process

step 1

time embed



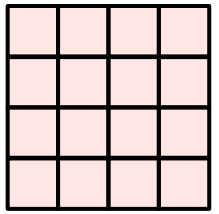
Unet

input latent

-

pred noise

=

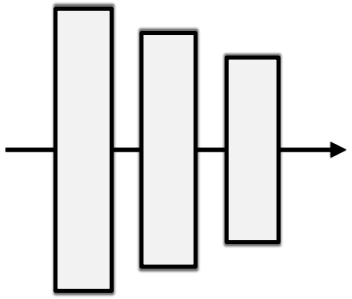


64x64x4

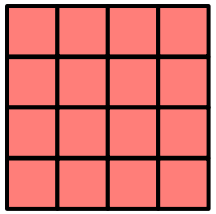
text embed
77x768

# Stable Diffusion | Training

NAVER AI LAB



encoder

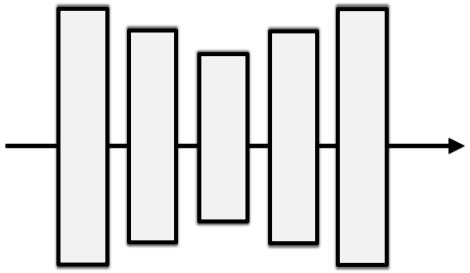


64x64x4

forward process

step 4

time embed



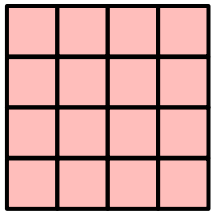
Unet

input latent

\-

pred noise

=

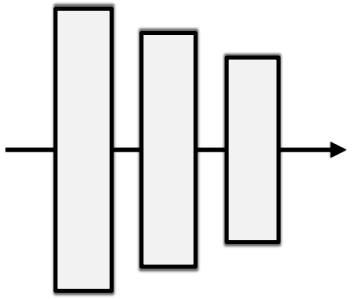


64x64x4

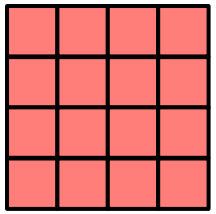
text embed
77x768

# Stable Diffusion | Training

NAVER AI LAB



encoder

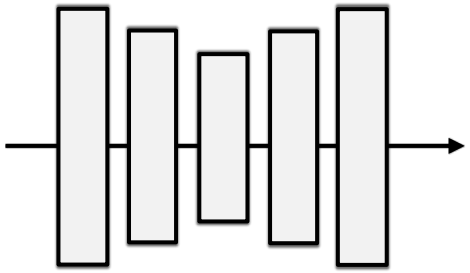


64x64x4

forward process

step 10

time embed



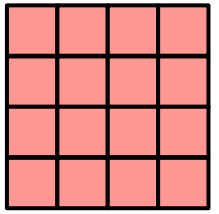
Unet

input latent

-

pred noise

=

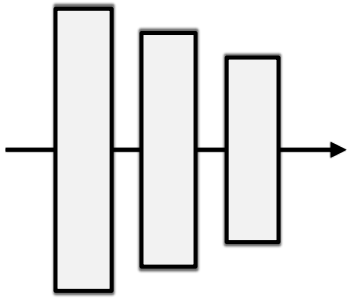


64x64x4

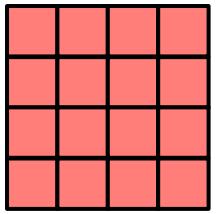
text embed
77x768

# Stable Diffusion | Training

NAVER AI LAB



encoder

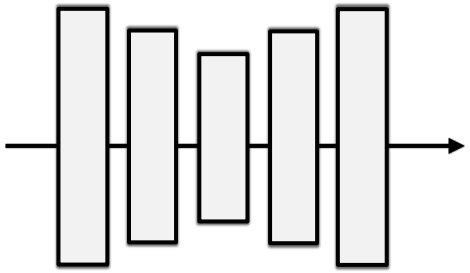


64x64x4

forward process

step 50

time embed



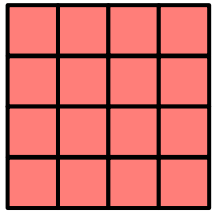
Unet

input latent

-

pred noise

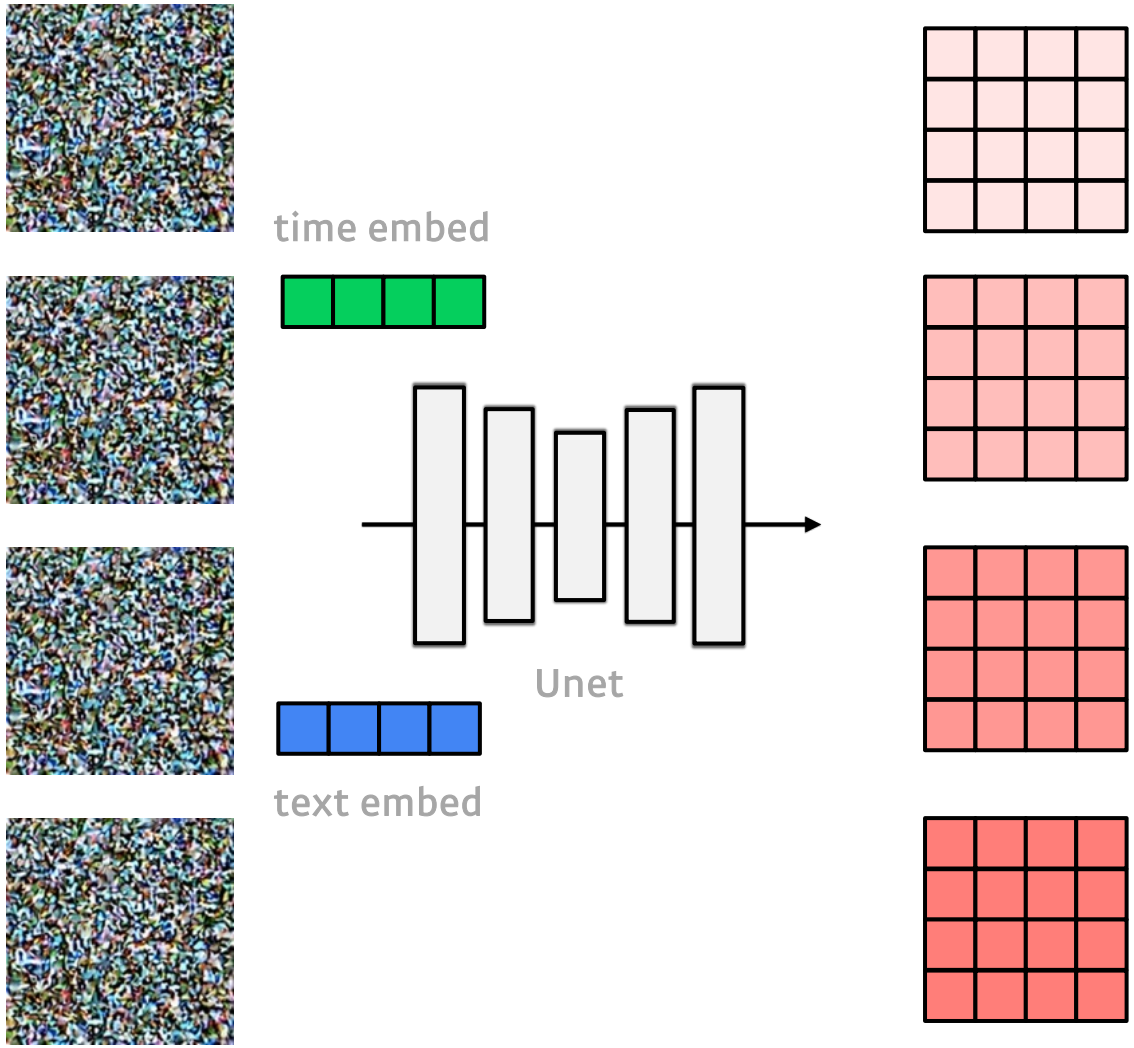
=



64x64x4

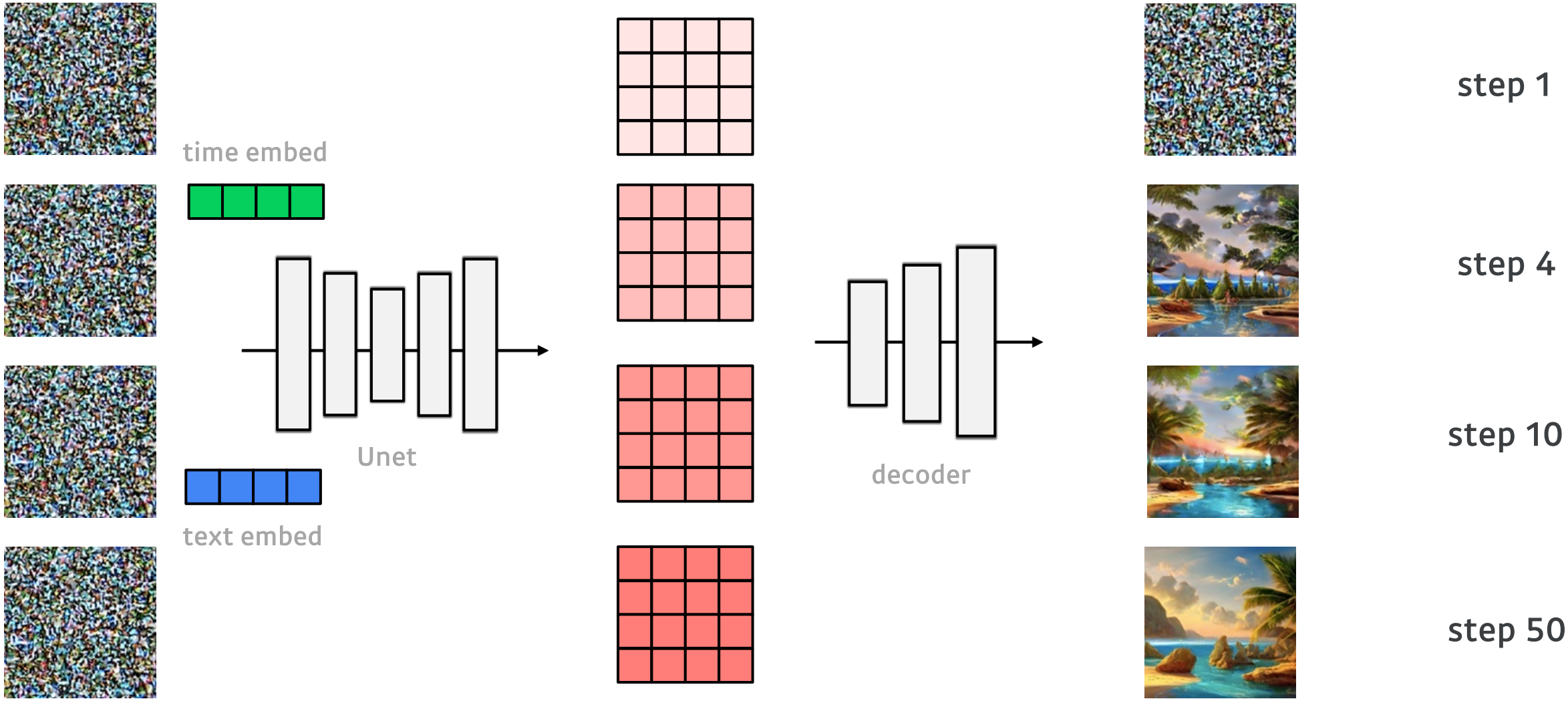
text embed
77x768

# Stable Diffusion | Training results



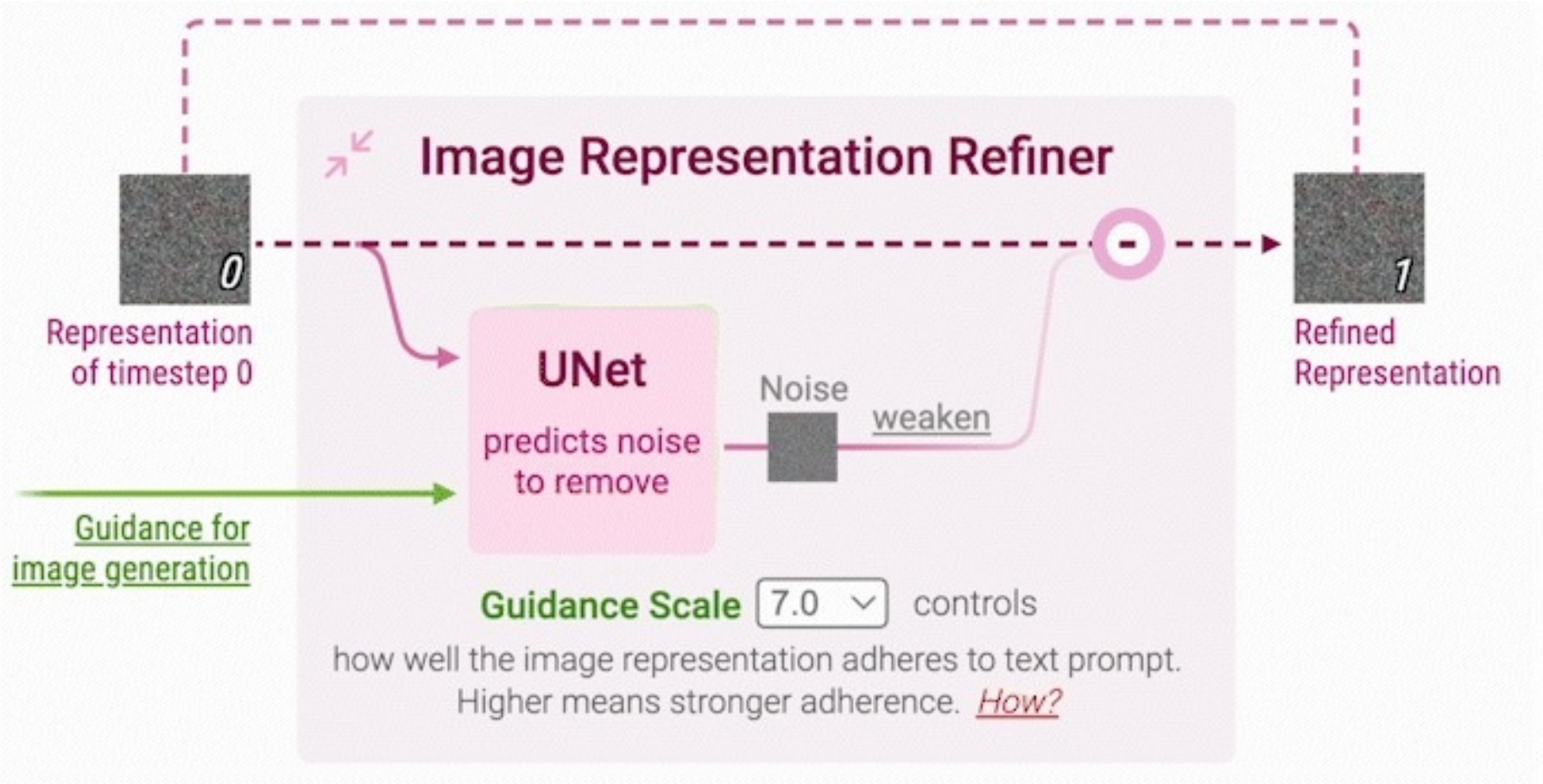
# Stable Diffusion | Training results

NAVER AI LAB



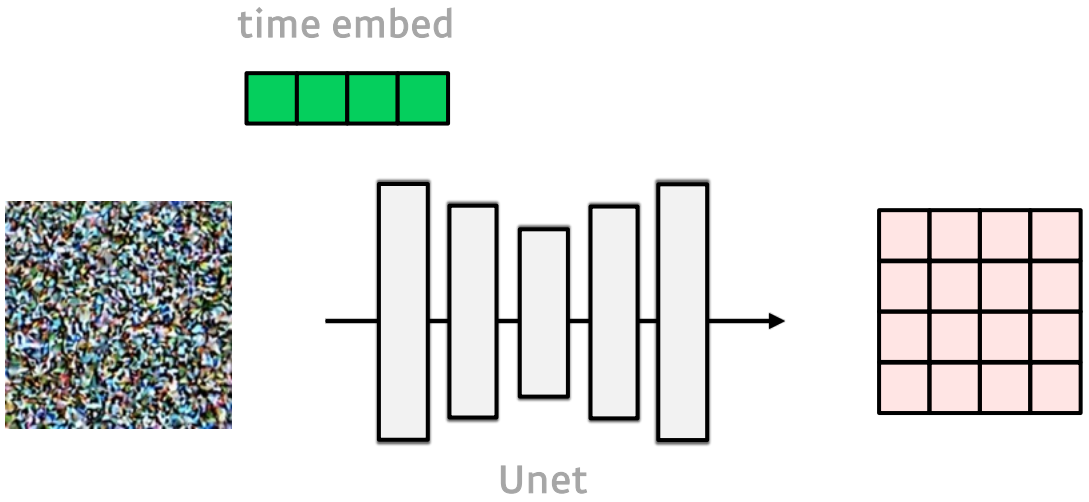
# Stable Diffusion | Training results

NAVER AI LAB




https://poloclub.github.io/diffusion-explainer/

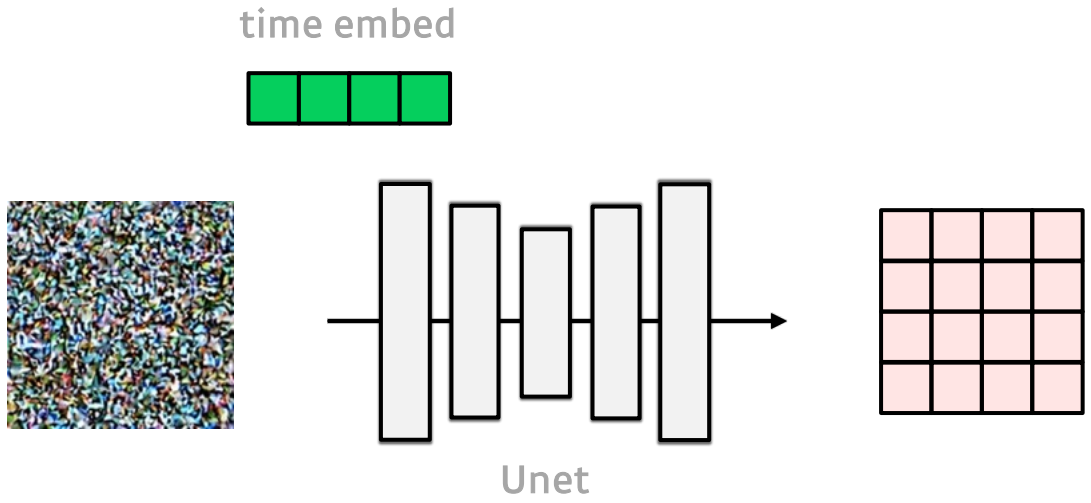
# Stable Diffusion | Conditioning



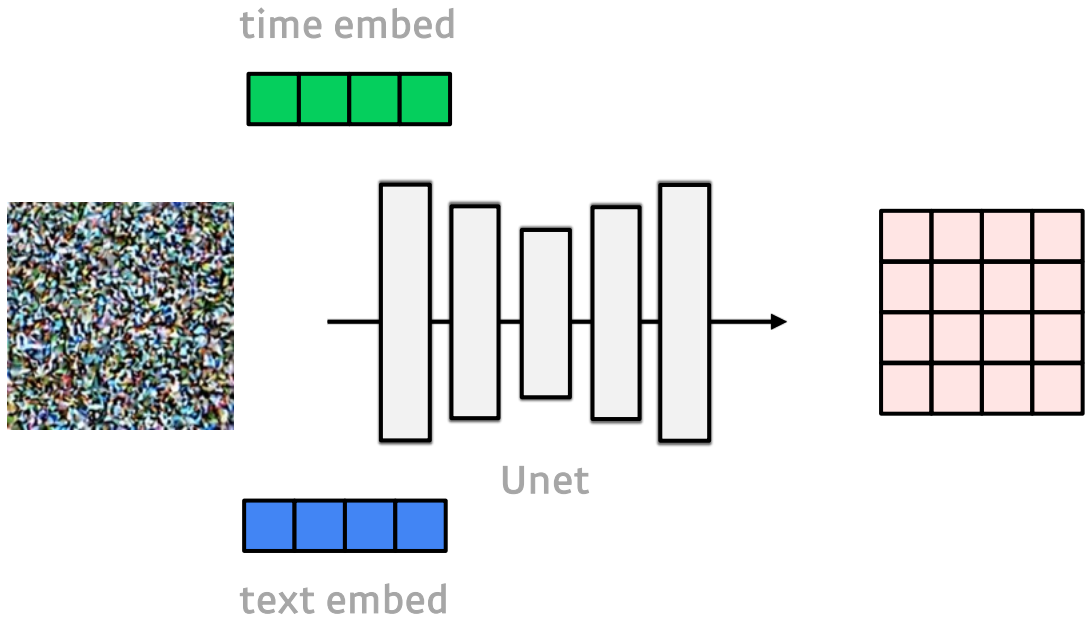

residual connection

# Stable Diffusion | Conditioning

NAVER AI LAB




residual connection




cross attention

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

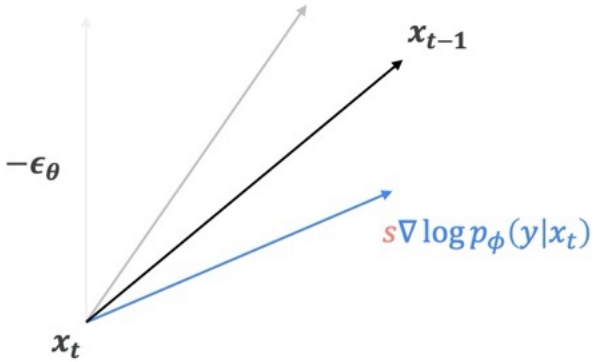
$$Q = (N, d_k), K = (N, d_k), V = (N, d_v)$$

# Stable Diffusion | Classifier free guidance

NAVER AI LAB

$$x_{t-1} \leftarrow N\left(\mu + s * \Sigma \nabla \log p_\phi(y|x_t), \Sigma\right)$$

Figure 3: Samples from an unconditional diffusion model with classifier guidance to condition on the class "Pembroke Welsh corgi". Using classifier scale 1.0 (left; FID: 33.0) does not produce convincing samples in this class, whereas classifier scale 10.0 (right; FID: 12.0) produces much more class-consistent images.

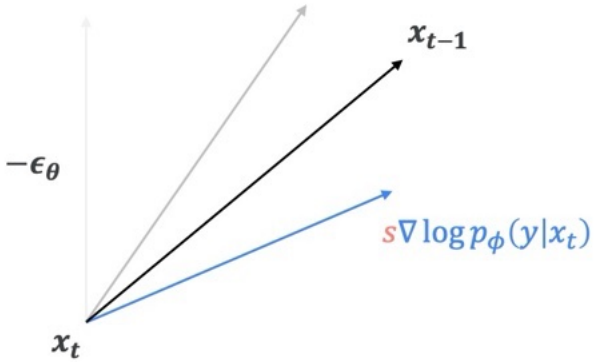


# Stable Diffusion | Classifier free guidance

NAVER AI LAB

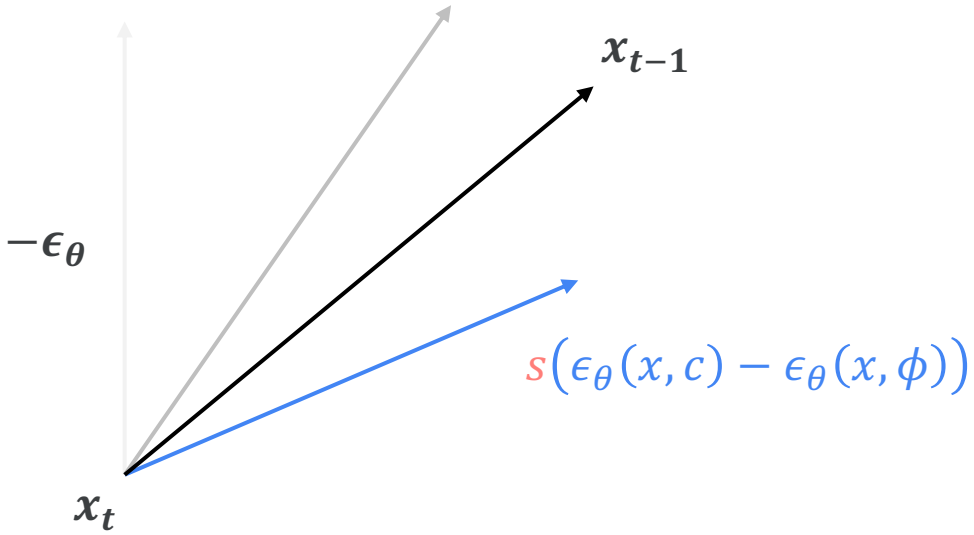
$$x_{t-1} \leftarrow N\left(\mu + s * \Sigma \nabla \log p_\phi(y|x_t), \Sigma\right)$$

Figure 3: Samples from an unconditional diffusion model with classifier guidance to condition on the class "Pembroke Welsh corgi". Using classifier scale 1.0 (left; FID: 33.0) does not produce convincing samples in this class, whereas classifier scale 10.0 (right; FID: 12.0) produces much more class-consistent images.




$$\tilde{\epsilon}_\theta(x, c) = \epsilon_\theta(x, \phi) + s * \left(\epsilon_\theta(x, c) - \epsilon_\theta(x, \phi)\right)$$

text준거랑, 안준거랑 차이를 키우자 !
= 그 방향으로 더 가자



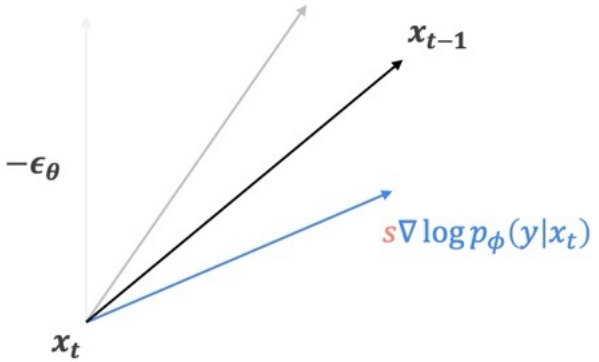

Classifier-Free Diffusion Guidance, NeurIPS 2021

# Stable Diffusion | Classifier free guidance

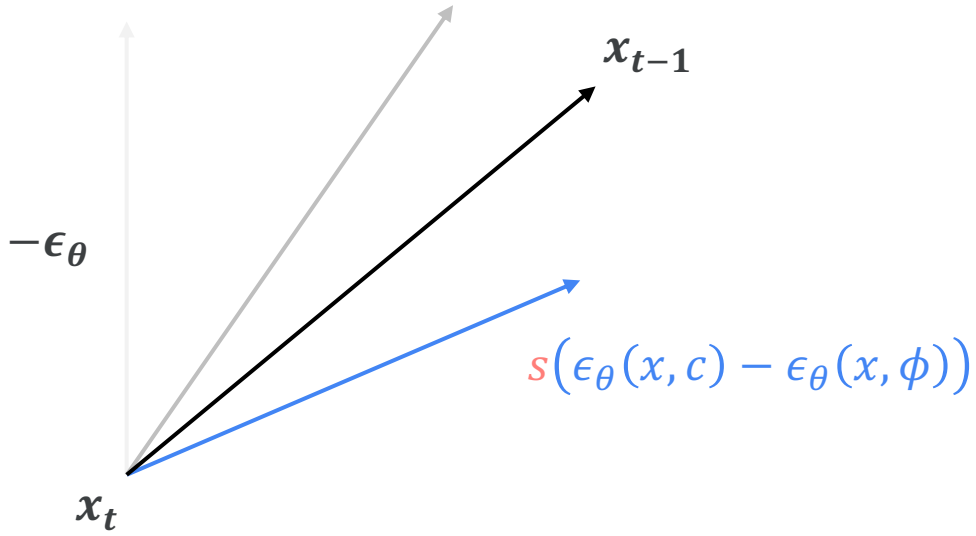
NAVER AI LAB

$$x_{t-1} \leftarrow N\left(\mu + s * \Sigma \nabla \log p_\phi(y|x_t), \Sigma\right)$$

Figure 3: Samples from an unconditional diffusion model with classifier guidance to condition on the class "Pembroke Welsh corgi". Using classifier scale 1.0 (left; FID: 33.0) does not produce convincing samples in this class, whereas classifier scale 10.0 (right; FID: 12.0) produces much more class-consistent images.




$$\tilde{\epsilon}_\theta(x, c) = \epsilon_\theta(x, \phi) + s * \left(\epsilon_\theta(x, c) - \epsilon_\theta(x, \phi)\right)$$



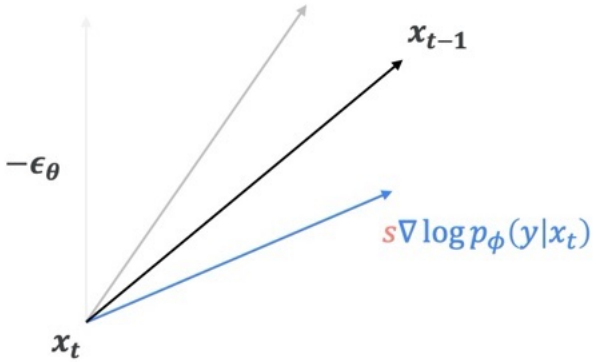

Classifier-Free Diffusion Guidance, NeurIPS 2021

# Stable Diffusion | Classifier free guidance

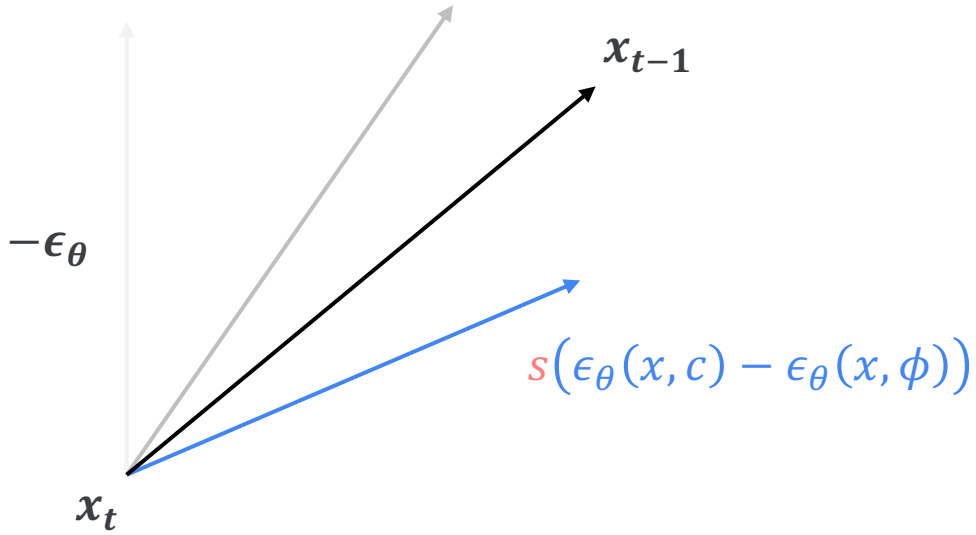
NAVER AI LAB

$$x_{t-1} \leftarrow N\left(\mu + s * \Sigma \nabla \log p_\phi(y|x_t), \Sigma\right)$$

Figure 3: Samples from an unconditional diffusion model with classifier guidance to condition on the class "Pembroke Welsh corgi". Using classifier scale 1.0 (left; FID: 33.0) does not produce convincing samples in this class, whereas classifier scale 10.0 (right; FID: 12.0) produces much more class-consistent images.




$$\tilde{\epsilon}_\theta(x, c) = \epsilon_\theta(x, \phi) + s * \left(\epsilon_\theta(x, c) - \epsilon_\theta(x, \phi)\right)$$



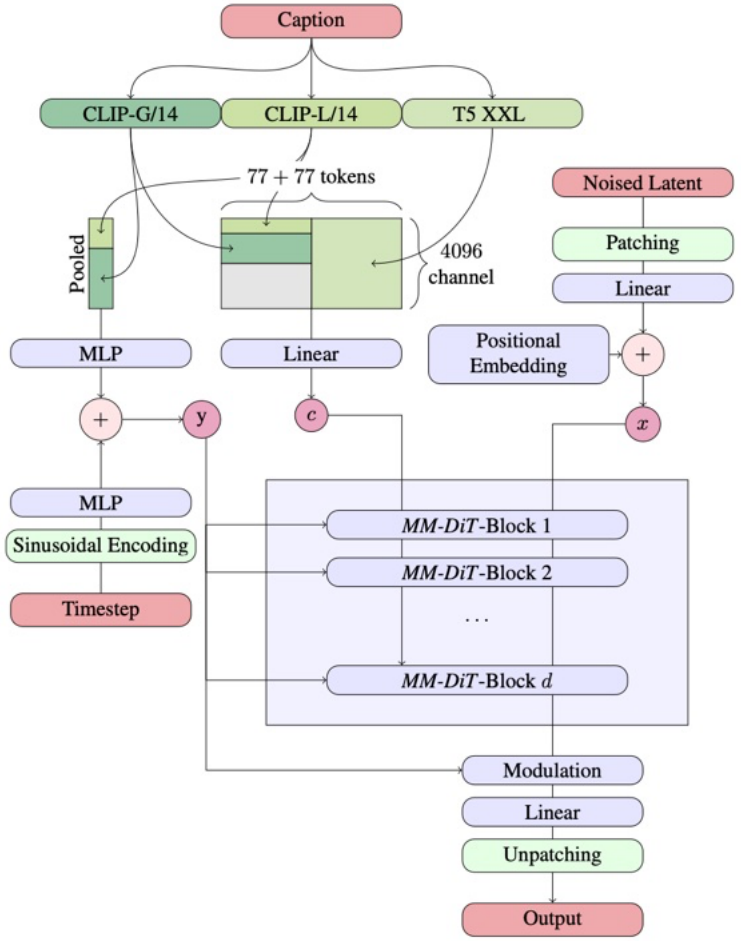

```
# perform guidance
if do_classifier_free_guidance:
    noise_pred_uncond, noise_pred_text = noise_pred.chunk(2)
    noise_pred = noise_pred_uncond + guidance_scale * (noise_pred_text - noise_pred_uncond)
```

Classifier-Free Diffusion Guidance, NeurIPS 2021

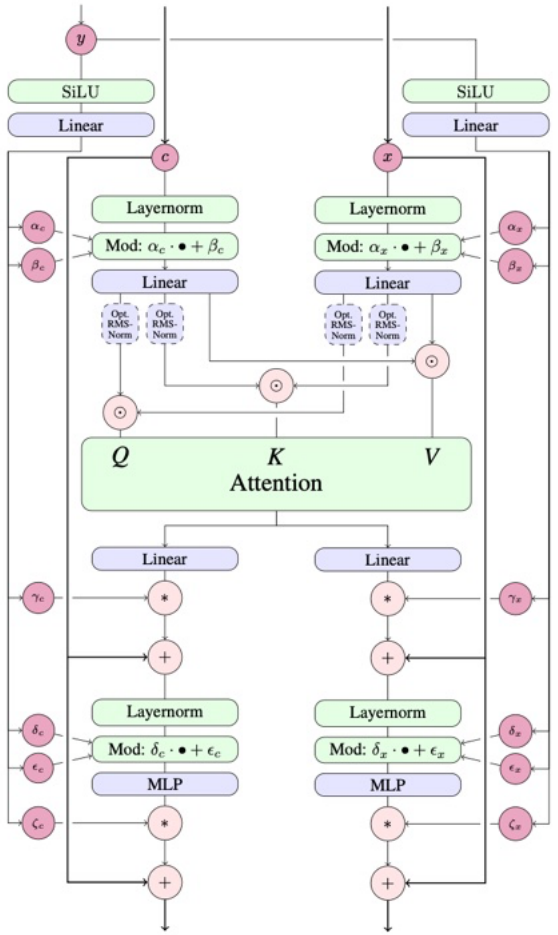
# Stable Diffusion 3 | Summary

NAVER AI LAB

- Proposed
  - Preprocessing
    - Data filtering
    - Precomputed embedding
    - Synthetic caption
  - Architecture
    - MM-DiT
    - Flexible text encoders
  - Training
    - Rectified flow
    - Logit-normal sampling (time t)
    - Conditional flow matching loss
  - Finetuning
    - QK-normalization
    - Timestep shifting
    - Improved position encoding
- Conclusion
  - 8B model
  - Validation loss = evaluation metrics




(a) Overview of all components.




(b) One *MM-DiT* block

# Stable Diffusion 3 | Data preprocessing

NAVER AI LAB

## E. Data Preprocessing for Large-Scale Text-to-Image Training

### E.1. Precomputing Image and Text Embeddings

Our model uses the output of multiple pretrained, frozen networks as inputs (autoencoder latents and text encoder representations). Since these outputs are constant during training, we precompute them once for the entire dataset. This comes with two main advantages: (i) The encoders do not need to be available on the GPU during training, lowering the required memory. (ii) The forward encoding pass is skipped during training, saving time and total needed compute after the first epoch, see Tab. 7.

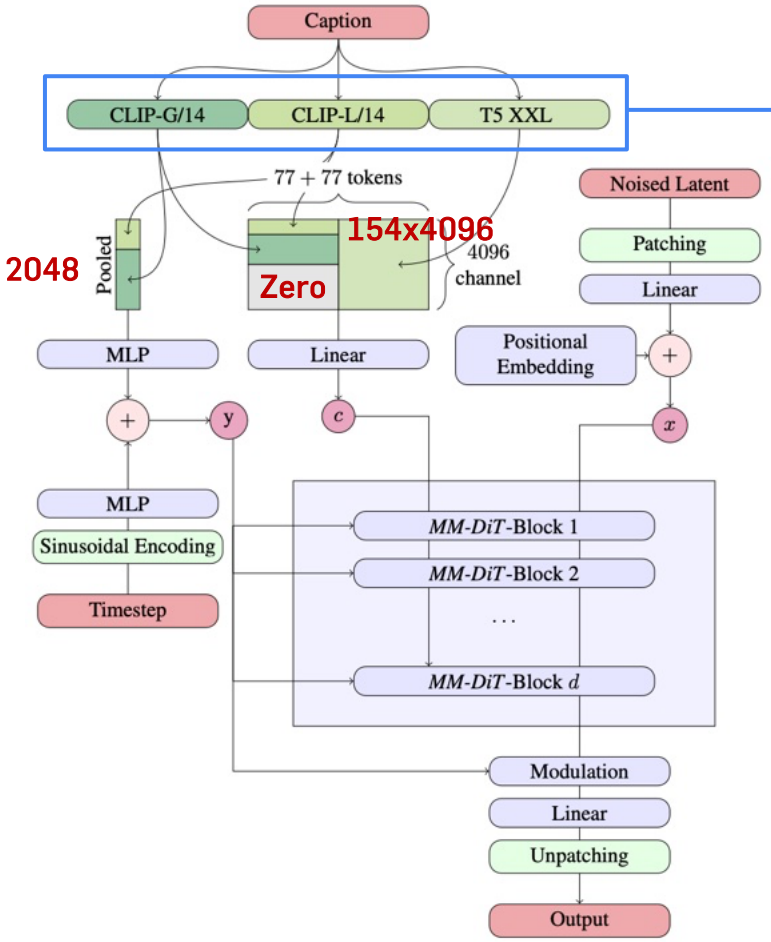
This approach has two disadvantages: First, random augmentation for each sample every epoch is not possible and we use square-center cropping during precomputation of image latents. For finetuning our model at higher resolutions, we specify a number of aspect ratio buckets, and resize and crop to the closest bucket first and then precompute in that aspect ratio. Second, the dense output of the text encoders is particularly large, creating additional storage cost and longer loading times during training (*c.f.* Tab. 7). We save the embeddings of the language models in half precision, as we do not observe a deterioration in performance in practice.

### E.2. Preventing Image Memorization

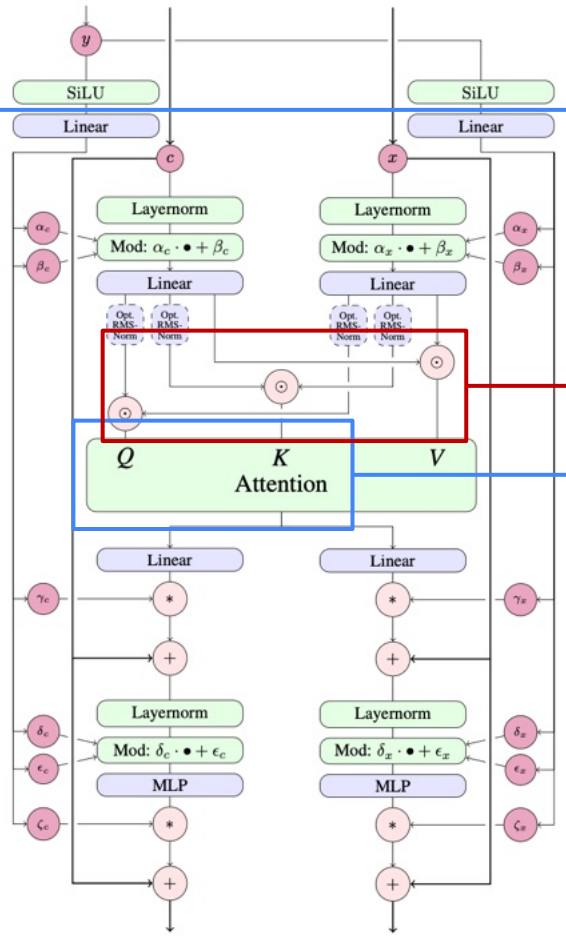
In the context of generative image models memorization of training samples can lead to a number of issues (Somepalli et al., 2023a; Carlini et al., 2023; Somepalli et al., 2023b). To avoid verbatim copies of images by our trained models, we carefully scan our training dataset for duplicated examples and remove them.

- Data filtering
  - SSCD 알고리즘을 통해, 이미지 중복 제거
    - 비슷한 이미지 찾기 Sota
- Precomputed embeddings
  - augmentation 활용을 많이 못함
  - 많은 storage 필요함
  - 그래도, 학습 빨라지는 이점
- Synthetic caption
  - CogVLM을 통해 real:fake caption 1:1 사용

# Stable Diffusion 3 | Architecture




(a) Overview of all components.




(b) One *MM-DiT* block

- **Multi text encoder**
  - **T5:** detailed caption
    - 없어도 이미지 퀄리티는 영향 x
  - **CLIP**: image-text matching
- **QK-Normalization**
  - **RMS Norm**

- **Step1. Image** $x \in \mathbb{R}^{h \times w \times c}$를 encoding해 **patch embedding**을 만들어줍니다.
  - 길이 $\frac{1}{2} \cdot h \cdot \frac{1}{2} \cdot w$의 2x2 사이즈의 patch들을 만들어내고
  - 각각의 patching를 embedding화 합니다.
- **Step2. Text embedding**의 sequence $c_{ctxt}$와 위 **patch embedding**의 차원을 맞춰줍니다.
- **Step3.** 두 sequence를 **concat**해 modulated attention과 MLP를 적용합니다.
  ** 각각의 modality에 대해서는 구분된 weight를 활용합니다.

-> 이러면 text의 이해력이 높아짐.

**8B**

# Stable Diffusion 3 | Techniques

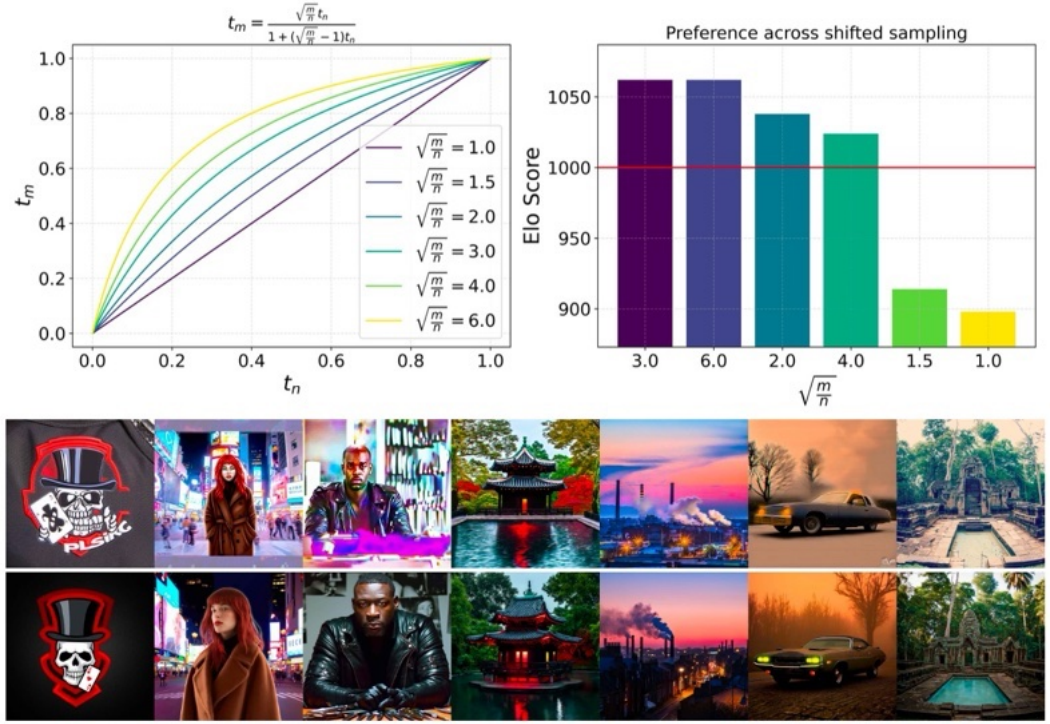



*Figure 6.* **Timestep shifting at higher resolutions.** *Top right:* Human quality preference rating when applying the shifting based on Equation (23). *Bottom row:* A $512^2$ model trained and sampled with $\sqrt{m/n} = 1.0$ (*top*) and $\sqrt{m/n} = 3.0$ (*bottom*). See Section 5.3.2.

**Time-shifting**

$$t_m = \frac{\sqrt{\frac{m}{n}}\, t_n}{1 + (\sqrt{\frac{m}{n}} - 1) t_n}$$

- **Problem**
  - 256x256 학습한뒤 -> high resolution으로 finetuning할때
    - mixed precision하면 학습이 불안정
    - full precision하면 학습이 잘 안됨 (?)
- **Solution**
  - QK RMS norm
  - Positional Encodings for Varying Aspect Ratios
    - 최근 RoPE 써도 되지 않을까.. 싶은..
  - Resolution-dependent shifting of timestep schedules
    - high-res는 low-res보다 더 많은 noise가 필요하므로, 그에 대응되도록 timestep schedule 변경

**QK Norm effects**

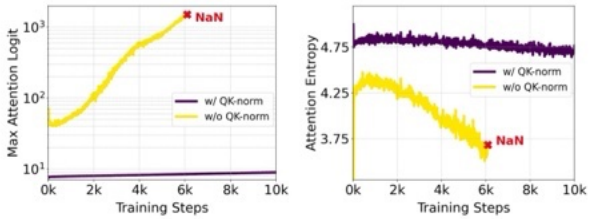



*Figure 5.* **Effects of QK-normalization.** Normalizing the Q- and K-embeddings before calculating the attention matrix prevents the attention-logit growth instability (*left*), which causes the attention entropy to collapse (*right*) and has been previously reported in the discriminative ViT literature (Dehghani et al., 2023; Wortsman et al., 2023). In contrast with these previous works, we observe this instability in the last transformer blocks of our networks. Maximum attention logits and attention entropies are shown averaged over the last 5 blocks of a 2B (d=24) model.

**PE for varying aspect ratios**

$$\left(\left(p - \frac{h_{\max} - s}{2}\right) \cdot \frac{256}{S}\right)_{p=0}^{h_{\max}-1}$$

NAVER AI LAB

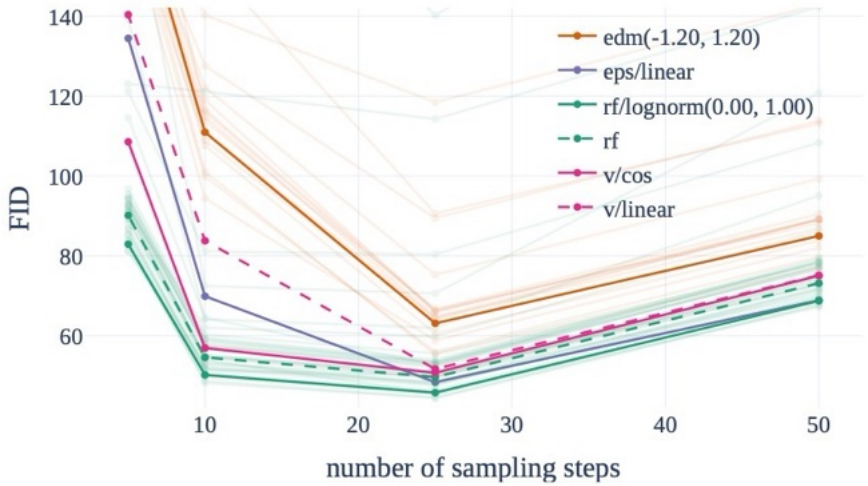



*Figure 3.* **Rectified flows are sample efficient.** Rectified Flows perform better then other formulations when sampling fewer steps. For 25 and more steps, only `rf/lognorm(0.00, 1.00)` remains competitive to `eps/linear`.

| Metric | 4 chn | 8 chn | 16 chn |
|---|---|---|---|
| FID (↓) | 2.41 | 1.56 | **1.06** |
| Perceptual Similarity (↓) | 0.85 | 0.68 | **0.45** |
| SSIM (↑) | 0.75 | 0.79 | **0.86** |
| PSNR (↑) | 25.12 | 26.40 | **28.62** |

*Table 3.* **Improved Autoencoders.** Reconstruction performance metrics for different channel configurations. The downsampling factor for all models is $f = 8$.

| | Original Captions | 50/50 Mix |
|---|---|---|
| | success rate [%] | success rate [%] |
| Color Attribution | 11.75 | 24.75 |
| Colors | 71.54 | 68.09 |
| Position | 6.50 | 18.00 |
| Counting | 33.44 | 41.56 |
| Single Object | 95.00 | 93.75 |
| Two Objects | 41.41 | 52.53 |
| Overall score | 43.27 | 49.78 |

*Table 4.* **Improved Captions.** Using a 50/50 mixing ratio of synthetic (via CogVLM (Wang et al., 2023)) and original captions improves text-to-image performance. Assessed via the GenEval (Ghosh et al., 2023) benchmark.

- **Improved Autoencoders**
  - 채널 늘리면 좋음 (16)
- **Improved captions**
  - dataset caption들은 detail이 빠져있음.
  - CogVLM같은거 통해서, fake caption을 만들고 함께 쓰면, detail 부족을 채울 수 있음.

# Stable Diffusion 3 | Ablation study

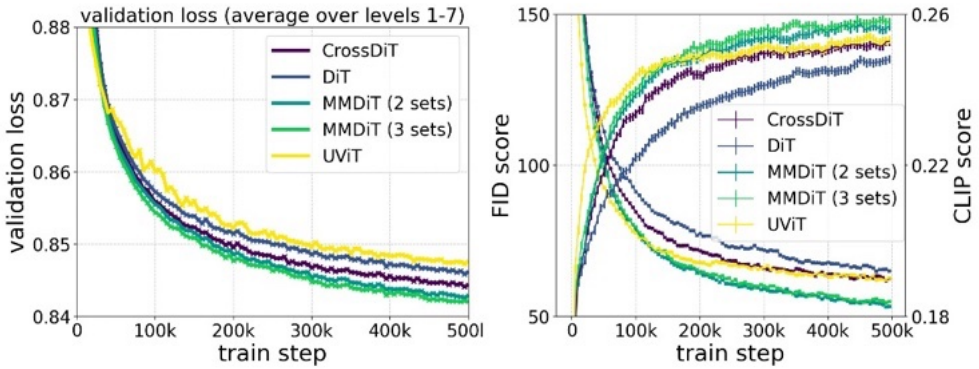



*Figure 4.* **Training dynamics of model architectures.** Comparative analysis of *DiT*, *CrossDiT*, *UViT*, and *MM-DiT* on CC12M, focusing on validation loss, CLIP score, and FID. Our proposed *MM-DiT* performs favorably across all metrics.

| 모델 | 특징 | 성능 |
|---|---|---|
| **DiT** | 텍스트와 이미지 토큰을 단순 시퀀스로 결합. | 제한된 텍스트-이미지 매칭 성능, 낮은 표현력. |
| **CrossDiT** | 텍스트 토큰에 교차 주의 메커니즘 추가. | DiT보다 높은 텍스트-이미지 매칭 성능, 학습 속도는 다소 느림. |
| **UViT** | UNet과 트랜스포머를 혼합한 아키텍처. | 초기 학습 속도 빠르지만, 최종 성능은 CrossDiT에 미치지 못함. |
| **MM-DiT** | 텍스트와 이미지에 별도의 가중치를 사용하며, 양방향 정보 교환. | 모든 평가 지표에서 가장 높은 성능(CLIP 점수, FID)과 빠른 수렴. |

# Stable Diffusion 3 | Results

NAVER AI LAB

Detailed pen and ink drawing of a happy pig butcher selling meat in its shop.

a massive alien space ship that is shaped like a pretzel.

A kangaroo holding a beer, wearing ski goggles and passionately singing silly songs.

An entire universe inside a bottle sitting on the shelf at walmart on sale.

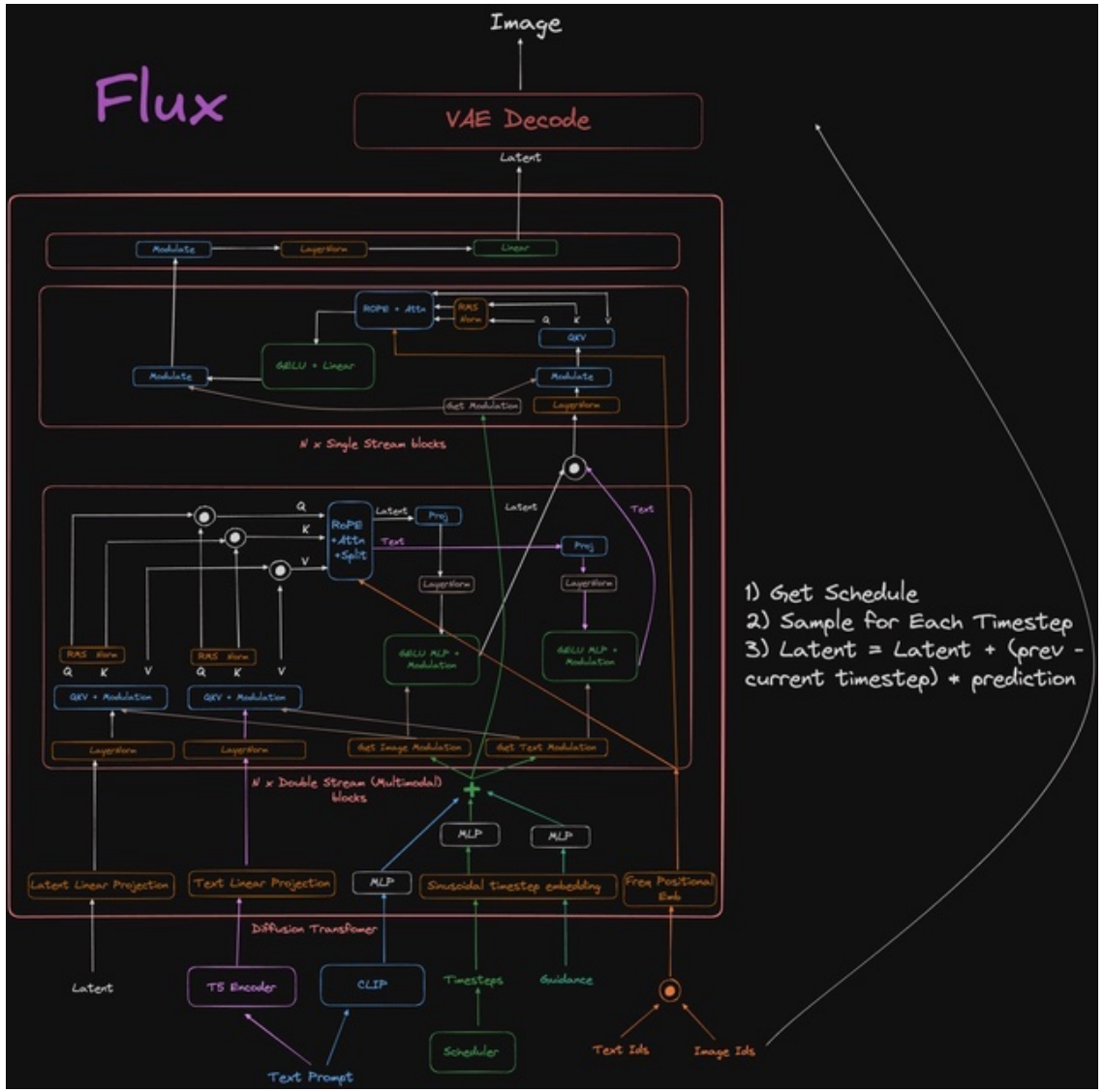
A cheesburger surfing the vibe wave at night

A swamp ogre with a pearl earring by Johannes Vermeer

A car made out of vegetables.

heat death of the universe, line art

# Flux | Summary




| GPU Device | VRAM (GB) | Stable Diffusion 3.5 Medium (2.5B) | SDXL (6.5B including refiner) | Playground v2.5 (3.5B) | AuraFlow v0.2 (8.7B) | Stable Diffusion 3.5 Large / Large Turbo (8.1B) | FLUX.1 [dev] (12B) | FLUX.1 [schnell] (12B) |
|---|---|---|---|---|---|---|---|---|
| NVIDIA GeForce RTX 4060 | 8 | ! | ! | ! | ! | ! | ! | ! |
| NVIDIA GeForce RTX 3080 | 10 | ✓ | ! | ! | ! | ! | ! | ! |
| NVIDIA GeForce RTX 3060<br>NVIDIA GeForce RTX 4070<br>AMD Radeon RX 7700 XT | 12 | ✓ | ! | ! | ! | ! | ! | ! |
| NVIDIA GeForce RTX 4060 Ti<br>NVIDIA GeForce RTX 4070 Ti<br>NVIDIA GeForce RTX 4080<br>AMD Radeon RX 7800 XT<br>AMD Radeon RX 7600 XT | 16 | ✓ | ✓ | ✓ | ! | ! | ! | ! |
| AMD Radeon RX 7900 XT | 20 | ✓ | ✓ | ✓ | ✓ | ! | ! | ! |
| NVIDIA GeForce RTX 3090<br>NVIDIA GeForce RTX 4090<br>AMD Radeon 7900XTX | 24 | ✓ | ✓ | ✓ | ✓ | ✓ | ! | ! |
| NVIDIA H100<br>AMD Instinct MI250X<br>AMD Instinct MI300A<br>AMD Instinct MI300X | 32 (or greater) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

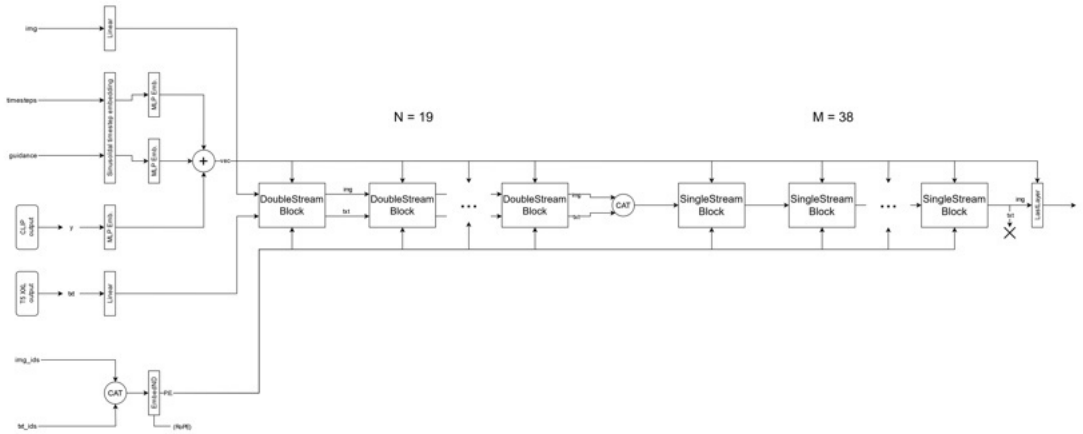
Hardware compatibility and VRAM requirements for open-image base models.

✓ *indicates the model runs on this device without any performance tradeoffs.*

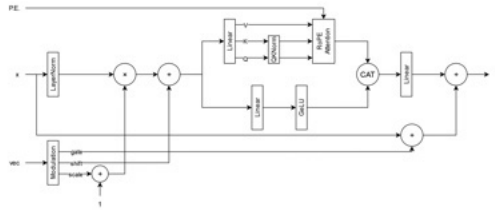
! *indicates the model requires performance-compromising optimizations, such as quantization or sequential offloading, to run on this device.*

https://www.reddit.com/r/LocalLLaMA/comments/1ekr7ji/fluxs_architecture_diagram_dont_think_theres_a/

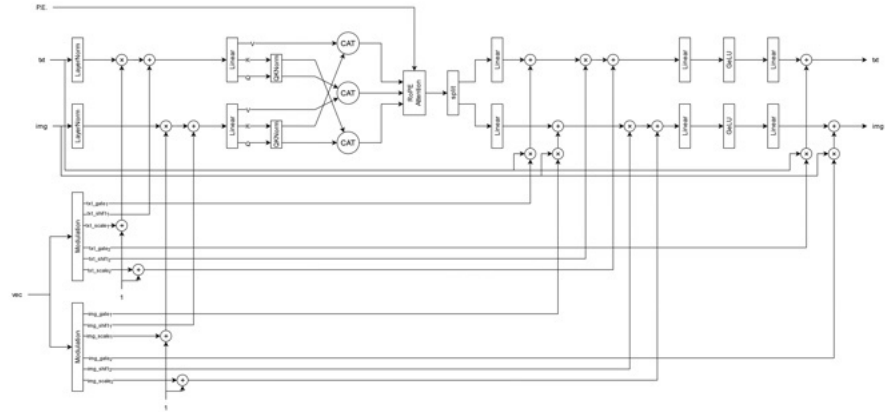
NAVER AI LAB




Flux.1 global architecture

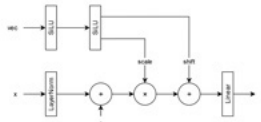


SingleStream block

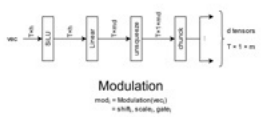


DoubleStream block

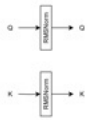
MLP Emb.



LastLayer



Modulation



QKNorm

References

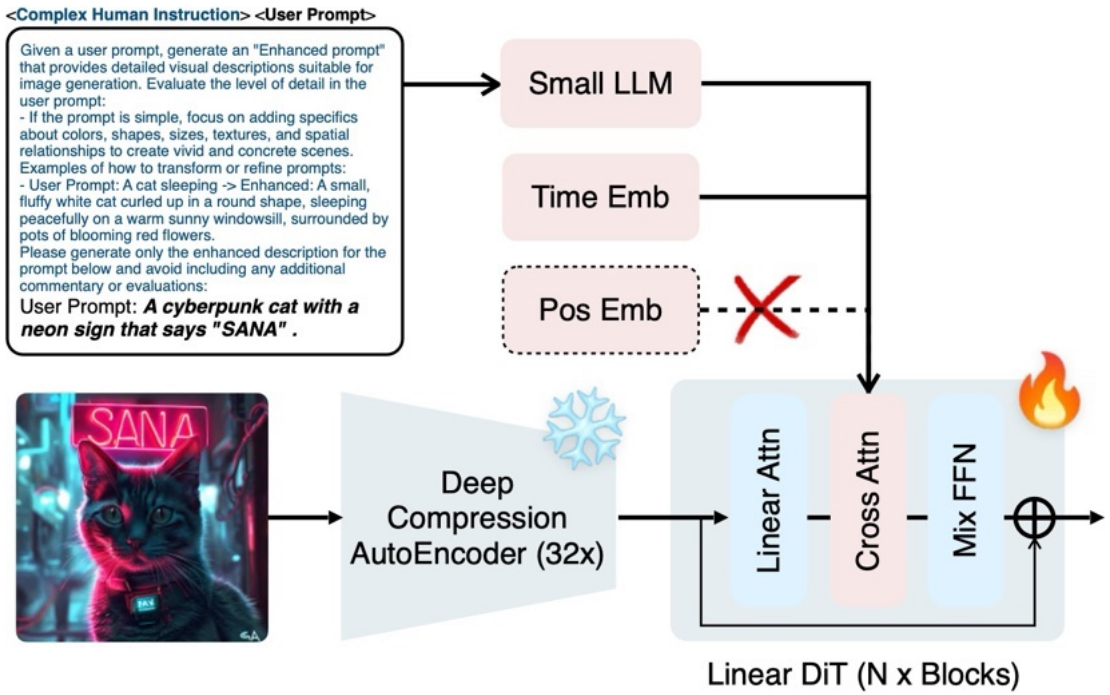
- source code: github.com/black-forest-labs/flux
- first architecture diagram: @nrehiew on X
- tensor shapes: PyTorch documentation
- DiT block for SingleStream: Dehghani et al., 2023

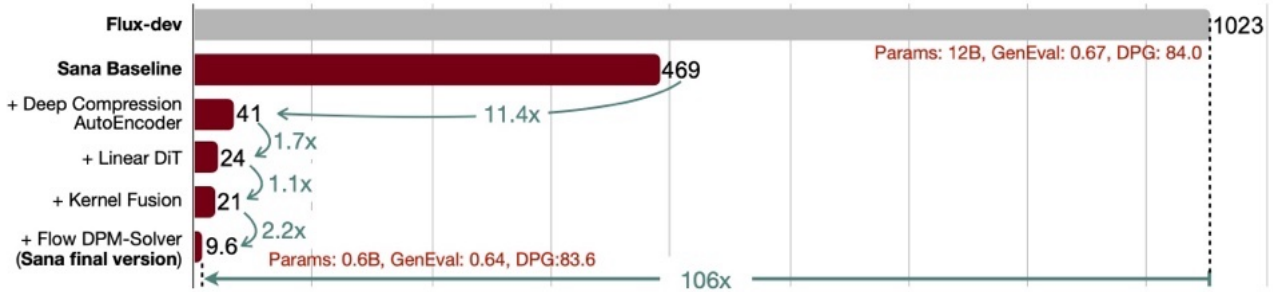
https://www.reddit.com/r/StableDiffusion/comments/1fds59s/a_detailled_flux1_architecture_diagram/

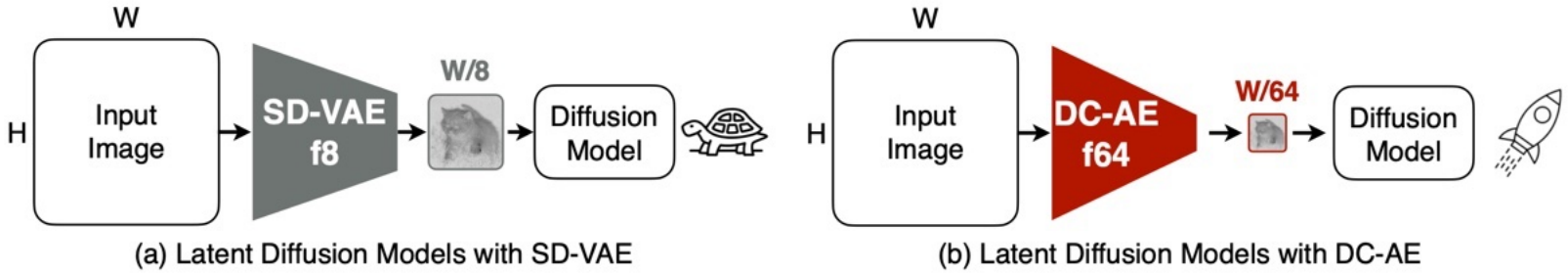
# SANA | Summary

NAVER AI LAB

- **Motivation**
  - high-quality & high-resolution & high-efficiency
- **Proposed**
  - Deep compression AE (x32)
    - 기존 x8
  - Efficient DiT
    - Vanilla attention -> Linear attention
      - $O(N^2)$ -> $O(N)$
    - No position embedding (NoPE)
  - Text encoder
    - T5 -> decoder-only LLM (Gemma2-2B)
  - Efficient training & sampling strategy
    - Flow-DPM-Solver (14-20 steps)
- **Conclusion**
  - 100x faster (4K), 40x faster (1K) vs FLUX
  - 1024x1024, 0.37s, 4090 GPU (Sana-0.6B)
- **Limitation**
  - 학습데이터는 뭐 썼는지 안나와있음. 학습시간도 없음.






(a) Latent Diffusion Models with SD-VAE (b) Latent Diffusion Models with DC-AE

Figure 1: DC-AE accelerates diffusion models by increasing autoencoder's spatial compression ratio.

## Summary

- **Residual Autoencoding**
  - 높은 압축률을 가능하게함 (x128)

- **Decoupled High-Resolution Adaptation (3 phase training)**
  - 고해상도 이미지에 generalization을 가능하게함 (2048x2048)

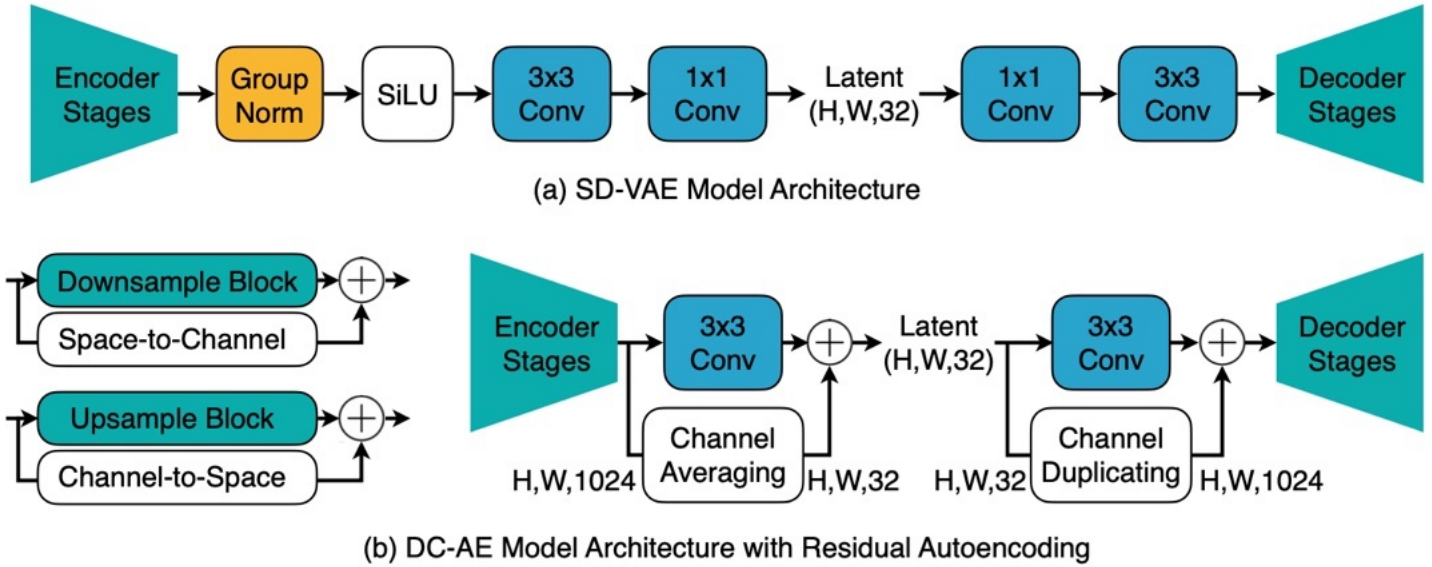



Figure 4: **Illustration of Residual Autoencoding.** It adds non-parametric shortcuts to let the neural network modules learn residuals based on the space-to-channel operation.

**Summary**

- residual -> not identity mapping
- space-to-channel mapping
  - pixel-shuffle, pixel-unshuffle

NAVER AI LAB

$$H \times W \times C \xrightarrow{\text{space-to-channel}} \frac{H}{2} \times \frac{W}{2} \times 4C \xrightarrow{\text{split into two groups}} [\frac{H}{2} \times \frac{W}{2} \times 2C, \frac{H}{2} \times \frac{W}{2} \times 2C] \xrightarrow{\text{average}} \frac{H}{2} \times \frac{W}{2} \times 2C.$$

channel averaging

```python
class PixelUnshuffleChannelAveragingDownSampleLayer(nn.Module):
    def __init__(
        self,
        in_channels: int,
        out_channels: int,
        factor: int,
    ):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.factor = factor
        assert in_channels * factor**2 % out_channels == 0
        self.group_size = in_channels * factor**2 // out_channels

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = F.pixel_unshuffle(x, self.factor)
        B, C, H, W = x.shape
        x = x.view(B, self.out_channels, self.group_size, H, W)
        x = x.mean(dim=2)
        return x
```

$$\frac{H}{2} \times \frac{W}{2} \times 2C \xrightarrow{\text{channel-to-space}} H \times W \times \frac{C}{2} \xrightarrow{\text{duplicate}} [H \times W \times \frac{C}{2}, H \times W \times \frac{C}{2}] \xrightarrow{\text{concat}} H \times W \times C.$$

channel duplicating

```python
class ChannelDuplicatingPixelUnshuffleUpSampleLayer(nn.Module):
    def __init__(
        self,
        in_channels: int,
        out_channels: int,
        factor: int,
    ):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.factor = factor
        assert out_channels * factor**2 % in_channels == 0
        self.repeats = out_channels * factor**2 // in_channels

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = x.repeat_interleave(self.repeats, dim=1)
        x = F.pixel_shuffle(x, self.factor)
        return x
```

https://github.com/mit-han-lab/efficientvit/blob/master/efficientvit/models/nn/ops.py

factor = 2

NAVER AI LAB

$$H \times W \times C \xrightarrow{\text{space-to-channel}} \frac{H}{2} \times \frac{W}{2} \times 4C$$

$$\xrightarrow{\text{split into two groups}} [\frac{H}{2} \times \frac{W}{2} \times 2C, \frac{H}{2} \times \frac{W}{2} \times 2C] \xrightarrow{\text{average}} \frac{H}{2} \times \frac{W}{2} \times 2C.$$

channel averaging

```python
class PixelUnshuffleChannelAveragingDownSampleLayer(nn.Module):
    def __init__(
        self,
        in_channels: int,
        out_channels: int,
        factor: int,
    ):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.factor = factor
        assert in_channels * factor**2 % out_channels == 0
        self.group_size = in_channels * factor**2 // out_channels

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = F.pixel_unshuffle(x, self.factor)
        B, C, H, W = x.shape
        x = x.view(B, self.out_channels, self.group_size, H, W)
        x = x.mean(dim=2)
        return x
```

$$\frac{H}{2} \times \frac{W}{2} \times 2C \xrightarrow{\text{channel-to-space}} H \times W \times \frac{C}{2}$$

$$\xrightarrow{\text{duplicate}} [H \times W \times \frac{C}{2}, H \times W \times \frac{C}{2}] \xrightarrow{\text{concat}} H \times W \times C.$$

channel duplicating

```python
class ChannelDuplicatingPixelUnshuffleUpSampleLayer(nn.Module):
    def __init__(
        self,
        in_channels: int,
        out_channels: int,
        factor: int,
    ):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.factor = factor
        assert out_channels * factor**2 % in_channels == 0
        self.repeats = out_channels * factor**2 // in_channels

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = x.repeat_interleave(self.repeats, dim=1)
        x = F.pixel_shuffle(x, self.factor)
        return x
```

https://github.com/mit-han-lab/efficientvit/blob/master/efficientvit/models/nn/ops.py

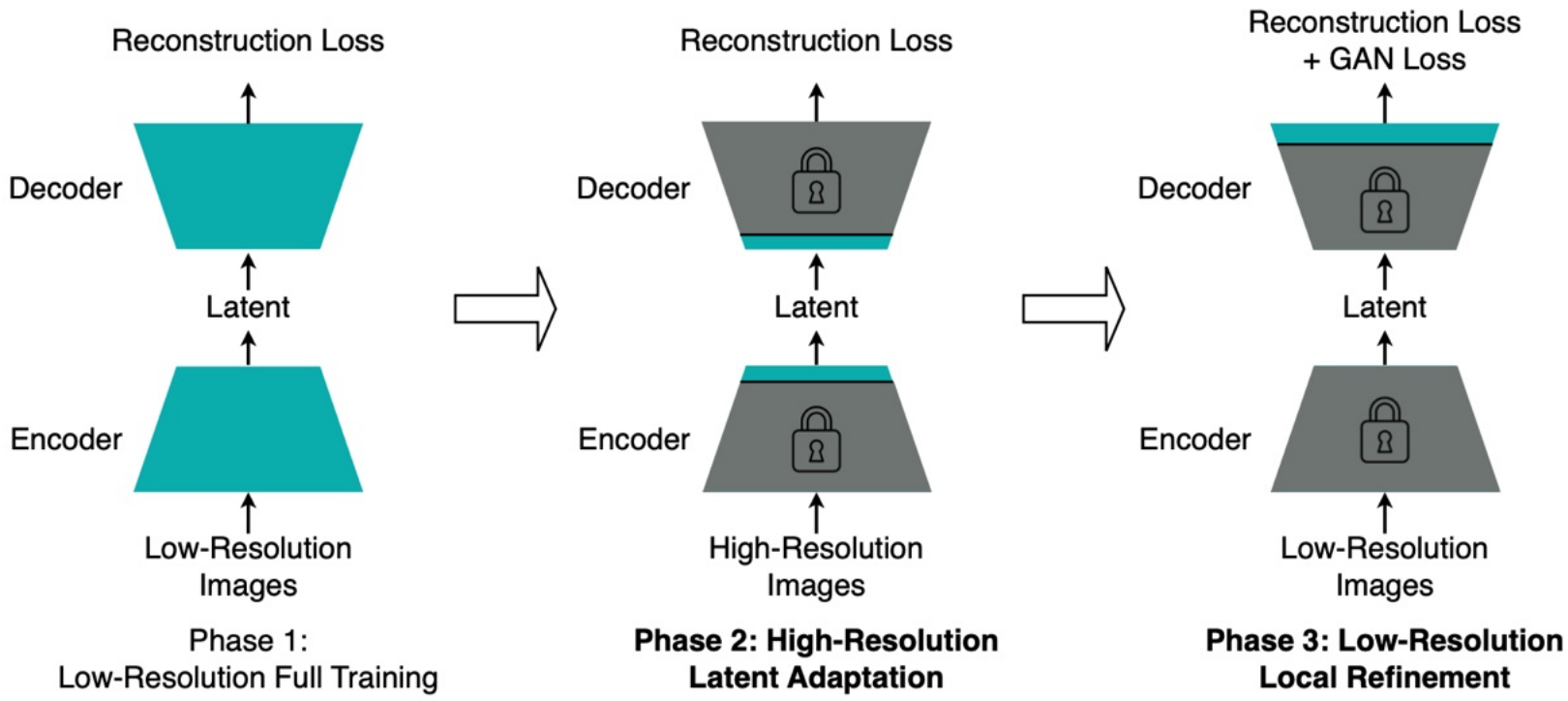



Figure 6: **Illustration of Decoupled High-Resolution Adaptation.**

**Key-points**

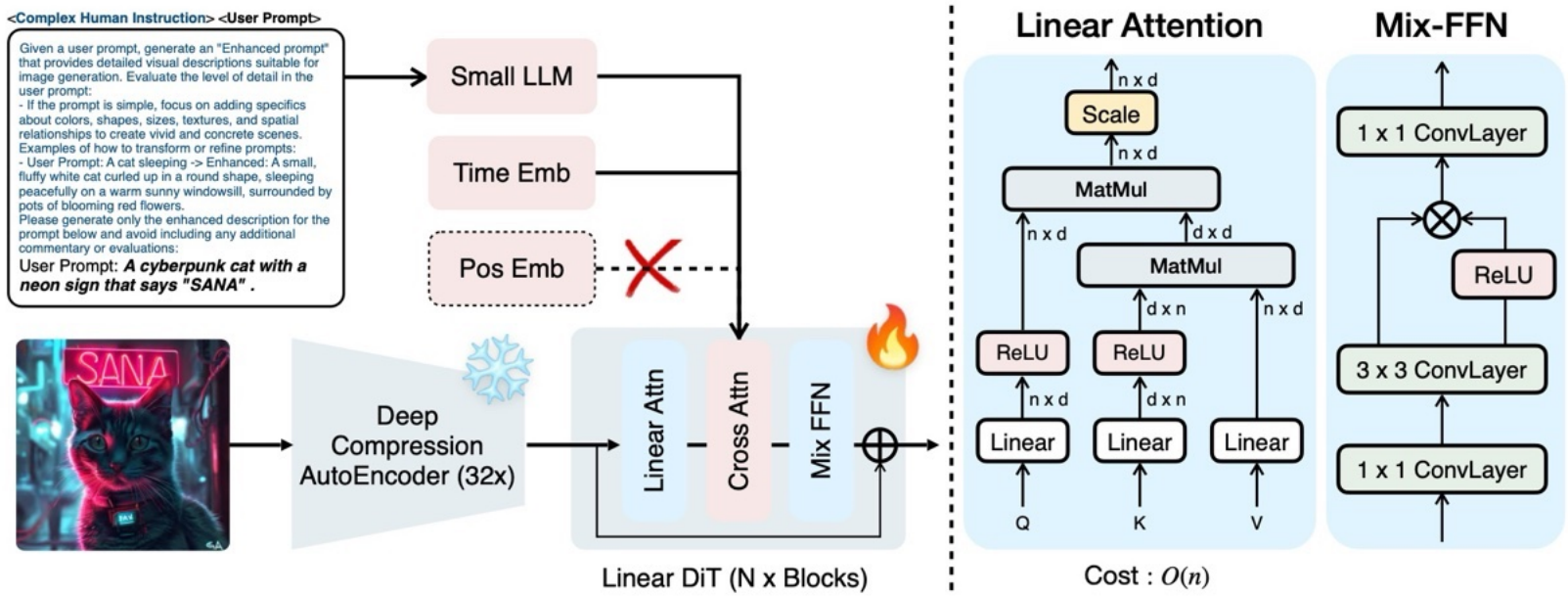
- GAN은 local detail을 향상시키고, local artifact를 제거한다.

**Summary**

- phase 1
  - full-training
  - 256x256
  - recon loss
  - Initialization
- phase 2
  - middle layer training
    - encoder's output
    - decoder's input
  - 1024x1024
  - recon loss
  - Generalization
- phase 3
  - decoder's output training
  - 256x256
  - recon + gan
  - Refinement

NAVER AI LAB

NAVER AI LAB

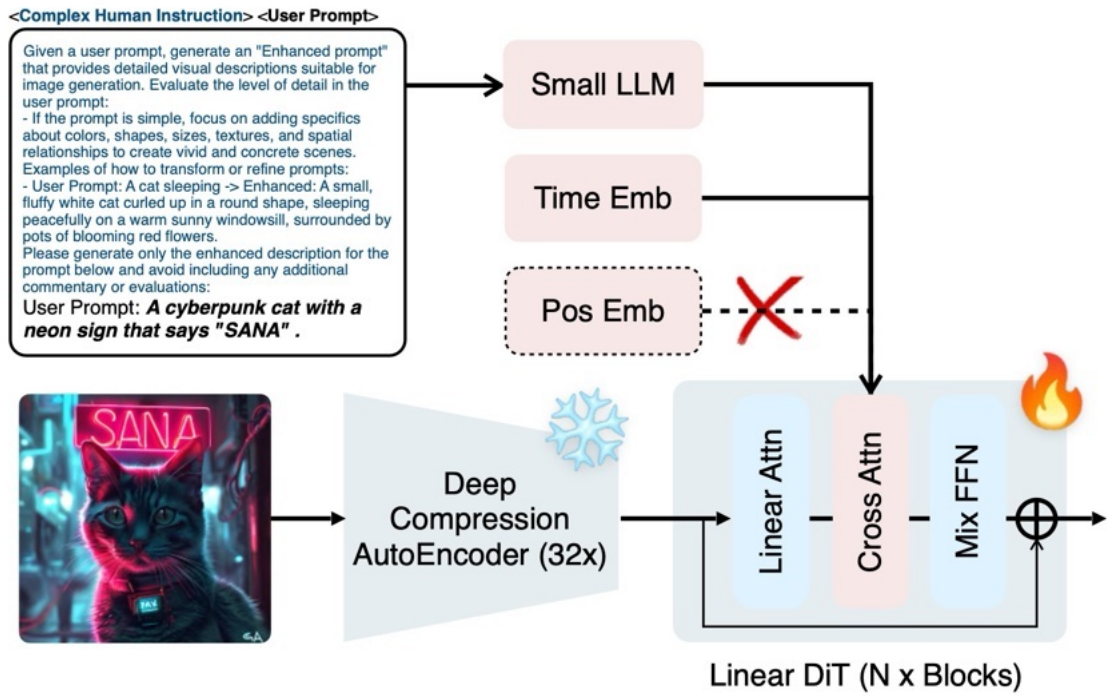



(a). Architecture overview of our Sana. (b). Linear DiT Module.

Figure 5: **Overview of Sana**: Fig. (a) describes the high-level training pipeline, containing our 32× deep compression Autoencoder, Linear DiT, and complex human instruction. Note that Positional embedding is not required in our framework. Fig. (b) describes the detailed design of the Linear Attention and Mix-FFN in Linear DiT.

- **Summary**
  - Linear attention
    - 계산량 감소, O(N)
  - Mix-FFN (depth-wise conv)
    - local information 보완
  - NoPE
    - 없어도 무방. (실험 없음)
  - Triton kernel fusion
    - Accelerate
  - LLM
    - not Text encoder(e.g. T5)

NAVER AI LAB




N x d N x d

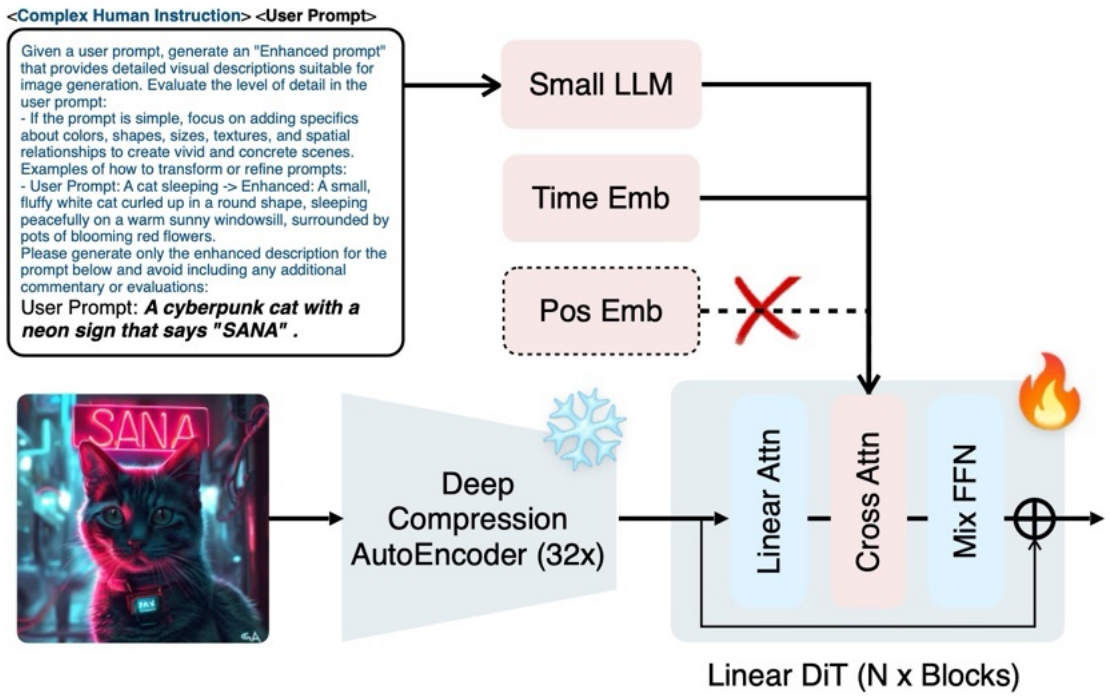
$$O_i = \sum_{j=1}^{N} \frac{Sim(Q_i, K_j)}{\sum_{j=1}^{N} Sim(Q_i, K_j)} V_j,$$

N x d

$$Sim(Q, K) = \exp\left(\frac{QK^T}{\sqrt{d}}\right)$$

$O(N^2)$

NAVER AI LAB




N x d N x d

$$O_i = \sum_{j=1}^{N} \frac{Sim(Q_i, K_j)}{\sum_{j=1}^{N} Sim(Q_i, K_j)} V_j, \quad \text{N x d}$$

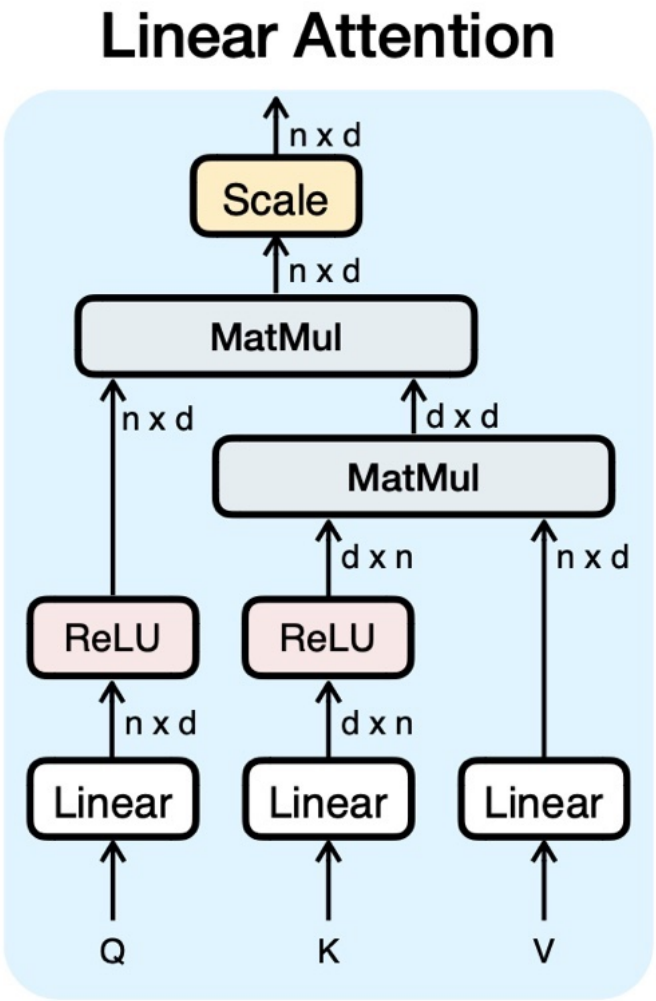
$$Sim(Q, K) = \exp\left(\frac{QK^T}{\sqrt{d}}\right)$$

$O(N^2)$

Similarity function을 재정의하자. (양수 되는걸로 !)

$$Sim(Q, K) = \mathrm{ReLU}(Q)\mathrm{ReLU}(K)^T$$

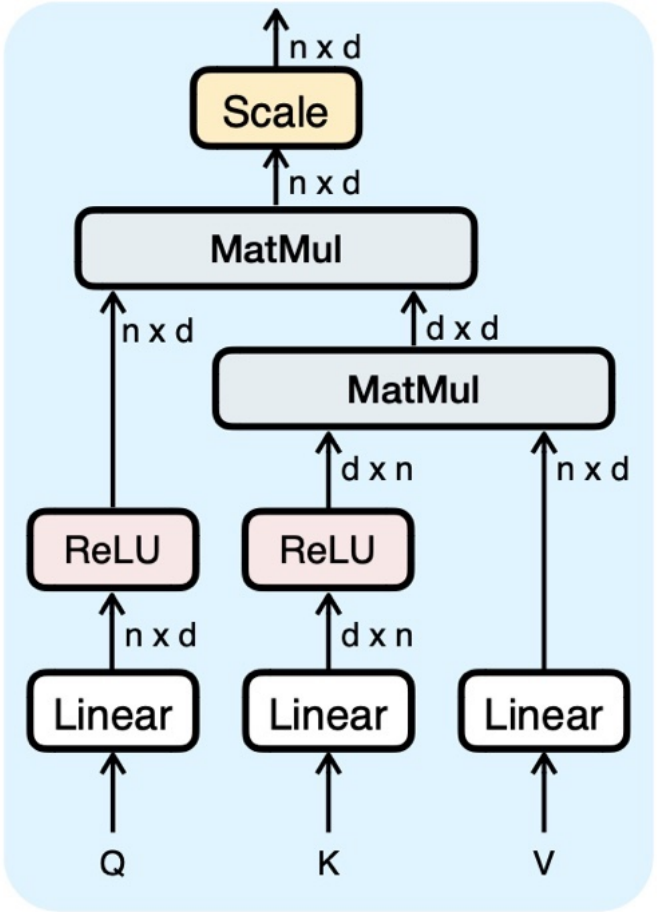
NAVER AI LAB




Cost : $O(n)$

NAVER AI LAB

## Linear Attention




Cost : $O(n)$

N x d  d x N

$$O_i = \sum_{j=1}^{N} \frac{\mathrm{ReLU}(Q_i)\mathrm{ReLU}(K_j)^T}{\sum_{j=1}^{N} \mathrm{ReLU}(Q_i)\mathrm{ReLU}(K_j)^T} V_j \quad \text{N x d}$$

$$= \frac{\sum_{j=1}^{N} (\mathrm{ReLU}(Q_i)\mathrm{ReLU}(K_j)^T) V_j}{\mathrm{ReLU}(Q_i) \sum_{j=1}^{N} \mathrm{ReLU}(K_j)^T}.$$

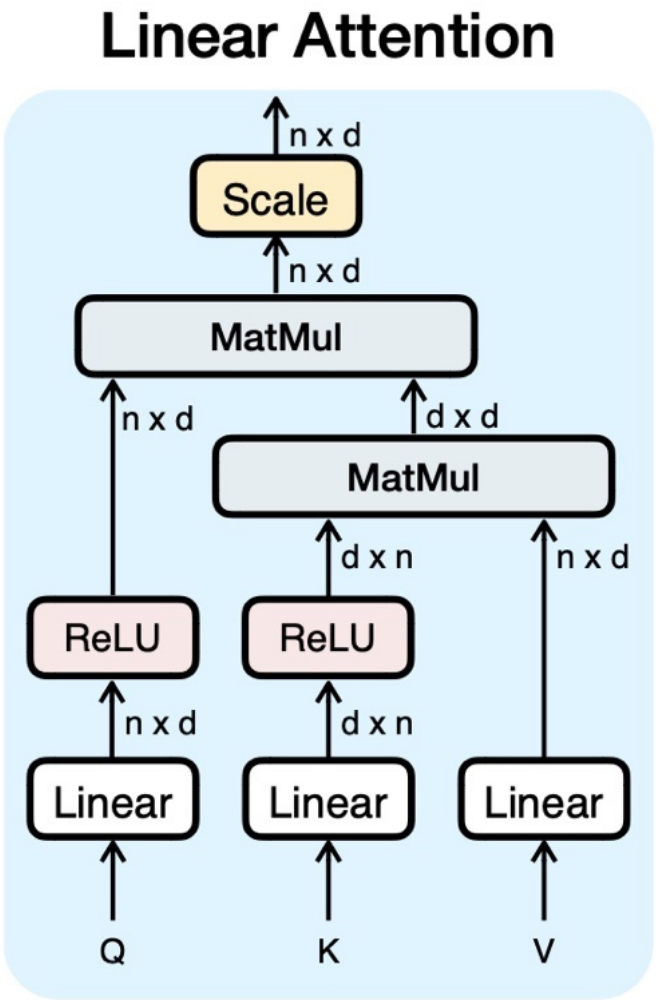
$$O_i = \frac{\sum_{j=1}^{N} [\mathrm{ReLU}(Q_i)\mathrm{ReLU}(K_j)^T] V_j}{\mathrm{ReLU}(Q_i) \sum_{j=1}^{N} \mathrm{ReLU}(K_j)^T}$$

$$= \frac{\sum_{j=1}^{N} \mathrm{ReLU}(Q_i)[(\mathrm{ReLU}(K_j)^T V_j)]}{\mathrm{ReLU}(Q_i) \sum_{j=1}^{N} \mathrm{ReLU}(K_j)^T}$$

$$= \frac{\mathrm{ReLU}(Q_i)(\sum_{j=1}^{N} \mathrm{ReLU}(K_j)^T V_j)}{\mathrm{ReLU}(Q_i)(\sum_{j=1}^{N} \mathrm{ReLU}(K_j)^T)}.$$

행렬의 곱셈 결합법칙

NAVER AI LAB




Cost : $O(n)$

```python
# lightweight linear attention
q = self.kernel_func(q)
k = self.kernel_func(k)

# linear matmul
trans_k = k.transpose(-1, -2)

v = F.pad(v, (0, 0, 0, 1), mode="constant", value=1)
vk = torch.matmul(v, trans_k)
out = torch.matmul(vk, q)
if out.dtype == torch.bfloat16:
    out = out.float()
out = out[:, :, :-1] / (out[:, :, -1:] + self.eps)

out = torch.reshape(out, (B, -1, H, W))
return out
```


Figure 3: **Softmax Attention vs. ReLU Linear Attention.** Unlike softmax attention, ReLU linear attention cannot produce sharp attention distributions due to a lack of the non-linear similarity function. Thus, its local information extraction ability is weaker than the softmax attention.

local information 추출 능력이 떨어짐.

NAVER AI LAB

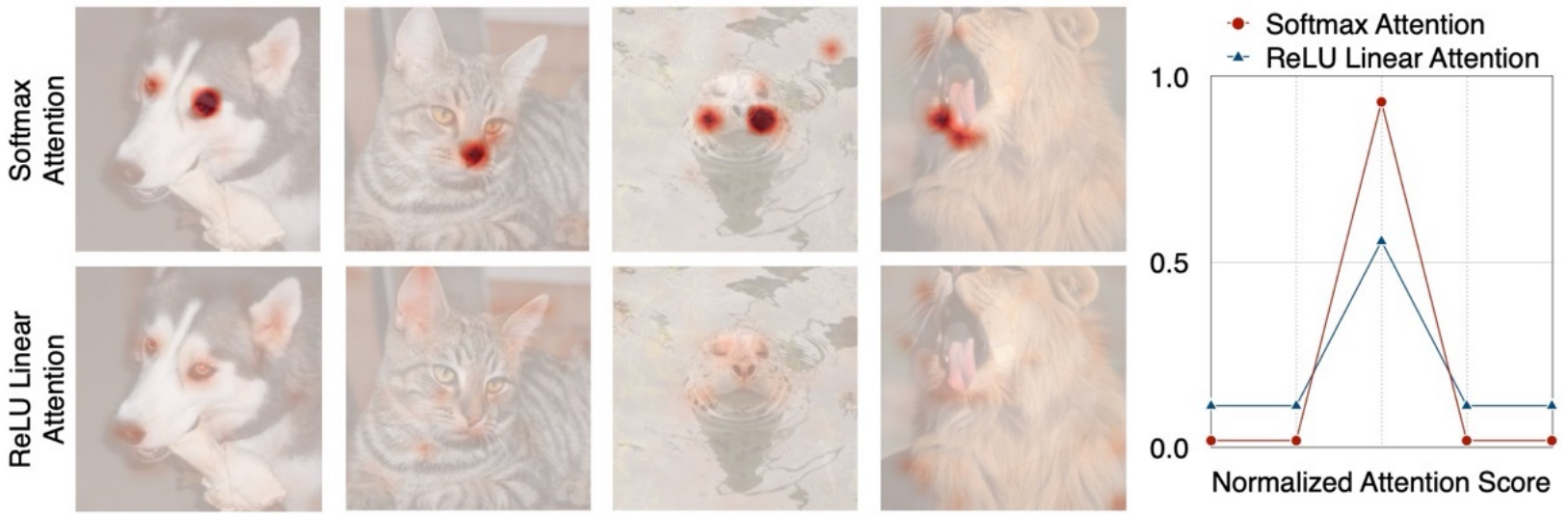
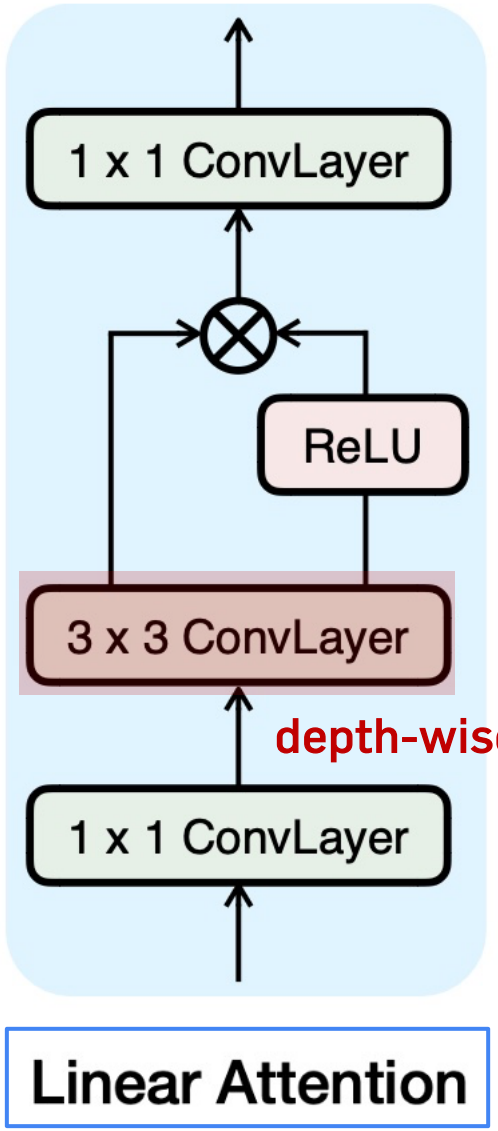



Figure 3: **Softmax Attention vs. ReLU Linear Attention.** Unlike softmax attention, ReLU linear attention cannot produce sharp attention distributions due to a lack of the non-linear similarity function. Thus, its local information extraction ability is weaker than the softmax attention.

local information 추출 능력이 떨어짐.

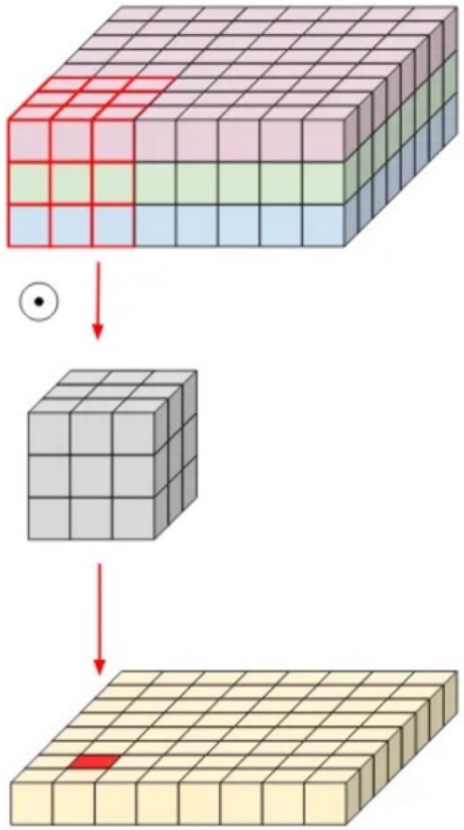
## Mix-FFN




**Linear Attention**

NAVER AI LAB

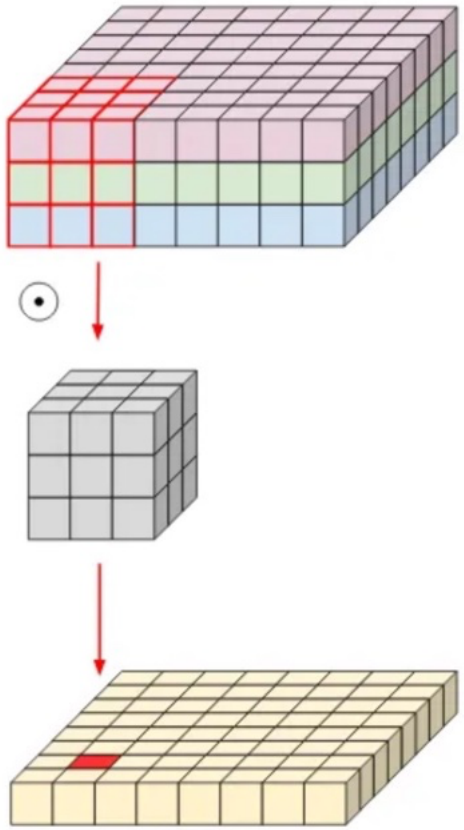
## Standard Conv



채널 간 상호작용

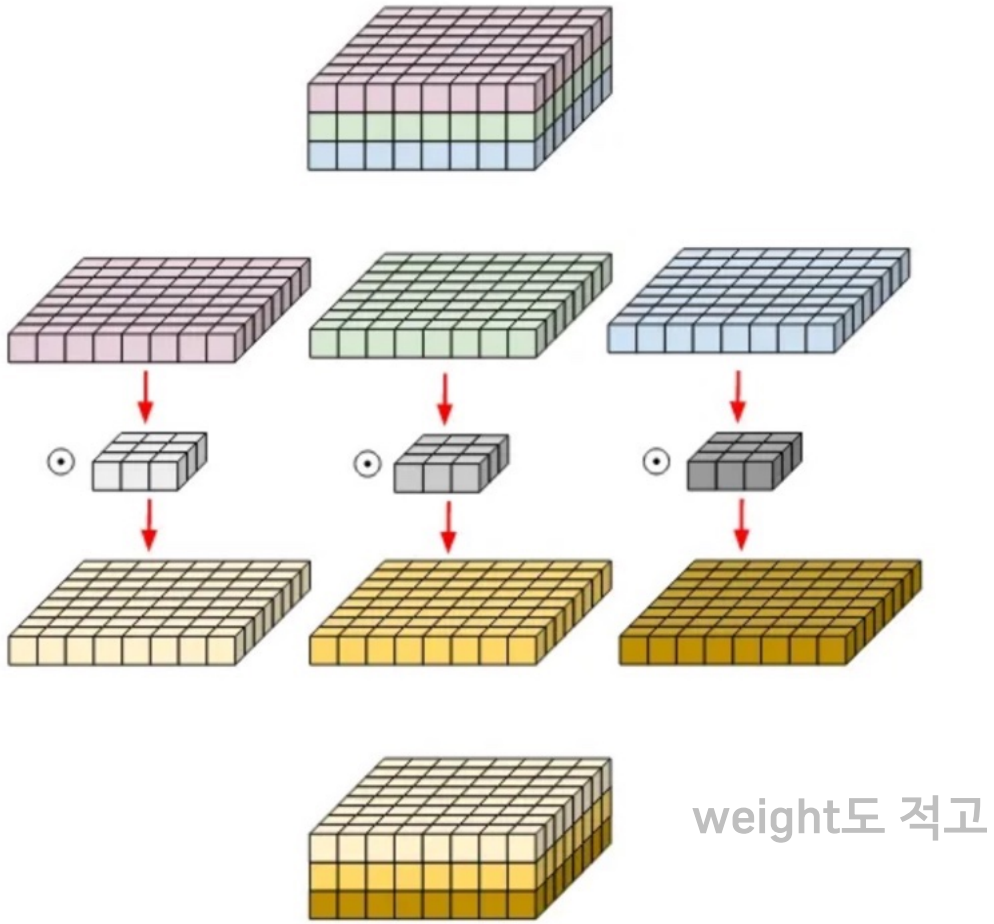
NAVER AI LAB

## Standard Conv



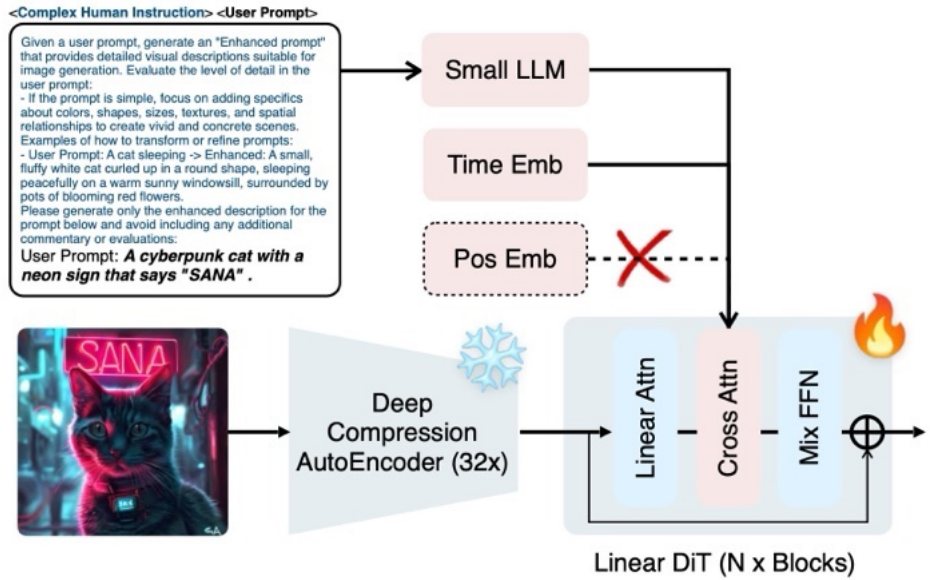
채널 간 상호작용

## Depth-wise Conv




각 채널에 독립적 필터 -> 채널별 로컬 정보 추출 가능

NAVER AI LAB


## Summary

- Motivation
  - LLM이 더 좋으니까
- How
  - last layer of features of the Gemme-2
- Useful tricks
  - RMS Norm after the text encoder.
    - normalizes the variance to 1.0
  - small learnable scale factor multiply to text emb
    - 0.01
  - Complex Human Instruction (CHI)
    - simple caption -> detailed caption

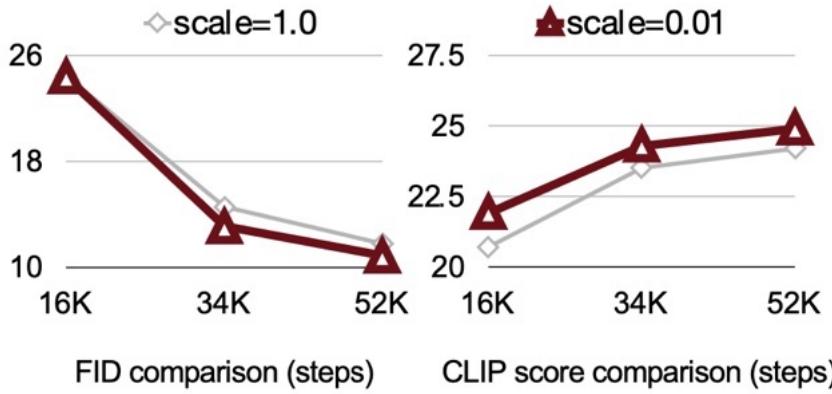



Figure 6: Ablation study of whether using text embedding normalization and small scale factor.

NAVER AI LAB


Figure 13: Visualization of zero-shot language transfer ability. Our Sana only has English prompts during training but can understand Chinese/Emoji during inference. This benefits from the generalization brought by the powerful pre-training of Gemma-2.

Table 13: Comparison of various T5 models and Gemma models based on speed and parameters. The sequence length (Seq Len) is the number of text tokens.

| Text Encoder | Batch Size | Seq Len | Latency (s) | Params (M) |
|---|---|---|---|---|
| T5-XXL | 32 | 300 | 1.6 | 4762 |
| T5-XL | | | 0.5 | 1224 |
| T5-large | | | 0.2 | 341 |
| T5-base | | | 0.1 | 110 |
| T5-small | | | 0.0 | 35 |
| Gemma-2b | | | 0.2 | 2506 |
| Gemma-2-2b | | | 0.3 | 2614 |

Table 9: **Comparison of different Text-Encoders.** All models are tested with an A100 GPU with FP16 precision. Gemma-2B models achieve better performance than T5-large at a similar speed and comparable performance to the larger, much slower T5-XXL.

| Text-Encoder | #Params (M) | Latency (s) | FID ↓ | CLIP ↑ |
|---|---|---|---|---|
| T5-XXL | 4762 | 1.61 | 6.1 | 27.1 |
| T5-Large | 341 | 0.17 | 6.1 | 26.2 |
| Gemma2-2B | 2614 | 0.28 | 6.0 | 26.9 |
| Gemma-2B-IT | 2506 | 0.21 | 5.9 | 26.8 |
| Gemma2-2B-IT | 2614 | 0.28 | 6.1 | 26.9 |

성능차이보단,
zero-shot ability가 더 쌔보임.

NAVER AI LAB

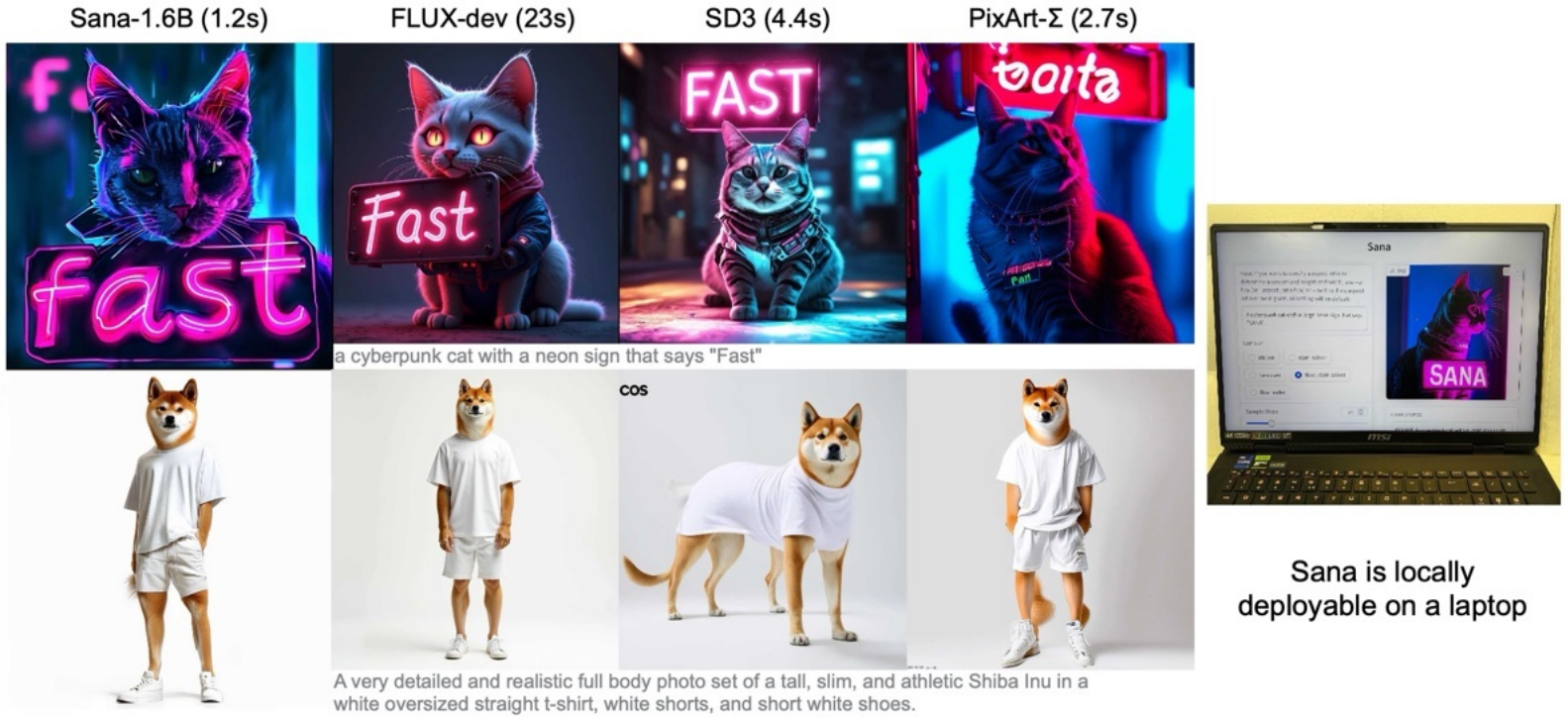



Figure 9: **Left**: Visualization comparison of Sana-1.6B vs FLUX-dev, SD3 and PixArt-Σ. The speed is tested on an A100 GPU with FP16 precision. **Right**: Quantize Sana-1.6B is deployable on a GPU laptop generating an 1K×1K image within 1 seconds.

**Summary**

- Pixart
  - text 랜더링 안좋음.
- SD3
  - text 반영 안좋음.
- Sana & FLUX
  - 비슷함.

Table 7: **Comprehensive comparison of our method with SOTA approaches in efficiency and performance.** The speed is tested on one A100 GPU with FP16 Precision. Throughput: Measured with batch=16. Latency: Measured with batch=1 and sampling step=20. We highlight the **best**, second best, and *third best* entries.

| Methods | Throughput (samples/s) | Latency (s) | Params (B) | Speedup | FID ↓ | CLIP ↑ | GenEval ↑ | DPG ↑ |
|---|---|---|---|---|---|---|---|---|
| **512 ×512 resolution** | | | | | | | | |
| PixArt-$\alpha$ (Chen et al., 2024b) | 1.5 | 1.2 | 0.6 | 1.0× | 6.14 | 27.55 | 0.48 | 71.6 |
| PixArt-$\Sigma$ (Chen et al., 2024a) | 1.5 | 1.2 | 0.6 | 1.0× | *6.34* | *27.62* | *0.52* | *79.5* |
| **Sana-0.6B** | 6.7 | 0.8 | 0.6 | 5.0× | 5.67 | 27.92 | 0.64 | 84.3 |
| **Sana-1.6B** | 3.8 | 0.6 | 1.6 | 2.5× | **5.16** | **28.19** | **0.66** | **85.5** |
| **1024 ×1024 resolution** | | | | | | | | |
| LUMINA-Next (Zhuo et al., 2024) | 0.12 | 9.1 | 2.0 | 2.8× | 7.58 | 26.84 | 0.46 | 74.6 |
| SDXL (Podell et al., 2023) | 0.15 | 6.5 | 2.6 | 3.5× | 6.63 | 29.03 | 0.55 | 74.7 |
| PlayGroundv2.5 (Li et al., 2024a) | 0.21 | 5.3 | 2.6 | 4.9× | *6.09* | **29.13** | 0.56 | 75.5 |
| Hunyuan-DiT (Li et al., 2024c) | 0.05 | 18.2 | 1.5 | 1.2× | 6.54 | 28.19 | 0.63 | 78.9 |
| PixArt-$\Sigma$ (Chen et al., 2024a) | 0.4 | 2.7 | 0.6 | 9.3× | 6.15 | 28.26 | 0.54 | 80.5 |
| DALLE3 (OpenAI, 2023) | - | - | - | - | - | - | 0.67 | 83.5 |
| SD3-medium (Esser et al., 2024) | 0.28 | 4.4 | 2.0 | 6.5× | 11.92 | 27.83 | 0.62 | 84.1 |
| FLUX-dev (Labs, 2024) | 0.04 | 23.0 | 12.0 | 1.0× | 10.15 | 27.47 | 0.67 | *84.0* |
| FLUX-schnell (Labs, 2024) | 0.5 | 2.1 | 12.0 | 11.6× | 7.94 | 28.14 | **0.71** | **84.8** |
| **Sana-0.6B** | 1.7 | 0.9 | 0.6 | 39.5× | 5.81 | 28.36 | 0.64 | 83.6 |
| **Sana-1.6B** | 1.0 | 1.2 | 1.6 | 23.3× | **5.76** | *28.67* | *0.66* | **84.8** |

Table 10: **Comparison of SOTA methods on GenEval with details.** The table includes different metrics such as overall performance, single object, two objects, counting, colors, position, and color attribution.

| Model | Params (B) | Overall ↑ | Objects | | Counting | Colors | Position | Color Attribution |
|---|---|---|---|---|---|---|---|---|
| | | | Single | Two | | | | |
| **512 ×512 resolution** | | | | | | | | |
| PixArt-$\alpha$ | 0.6 | **0.48** | 0.98 | 0.50 | 0.44 | 0.80 | 0.08 | 0.07 |
| PixArt-$\Sigma$ | 0.6 | **0.52** | 0.98 | 0.59 | 0.50 | 0.80 | 0.10 | 0.15 |
| **Sana-0.6B** (Ours) | 0.6 | **0.64** | 0.99 | 0.71 | 0.63 | 0.91 | 0.16 | 0.42 |
| **Sana-1.6B** (Ours) | 0.6 | **0.66** | 0.99 | 0.79 | 0.63 | 0.88 | 0.18 | 0.47 |
| **1024 ×1024 resolution** | | | | | | | | |
| LUMINA-Next (Zhuo et al., 2024) | 2.0 | **0.46** | 0.92 | 0.46 | 0.48 | 0.70 | 0.09 | 0.13 |
| SDXL (Podell et al., 2023) | 2.6 | **0.55** | 0.98 | 0.74 | 0.39 | 0.85 | 0.15 | 0.23 |
| PlayGroundv2.5 (Li et al., 2024a) | 2.6 | **0.56** | 0.98 | 0.77 | 0.52 | 0.84 | 0.11 | 0.17 |
| Hunyuan-DiT (Li et al., 2024c) | 1.5 | **0.63** | 0.97 | 0.77 | 0.71 | 0.88 | 0.13 | 0.30 |
| DALLE3 (OpenAI, 2023) | - | **0.67** | 0.96 | 0.87 | 0.47 | 0.83 | 0.43 | 0.45 |
| SD3-medium (Esser et al., 2024) | 2.0 | **0.62** | 0.98 | 0.74 | 0.63 | 0.67 | 0.34 | 0.36 |
| FLUX-dev (Labs, 2024) | 12.0 | **0.67** | 0.99 | 0.81 | 0.79 | 0.74 | 0.20 | 0.47 |
| FLUX-schnell (Labs, 2024) | 12.0 | **0.71** | 0.99 | 0.92 | 0.73 | 0.78 | 0.28 | 0.54 |
| **Sana-0.6B** (Ours) | 0.6 | **0.64** | 0.99 | 0.76 | 0.64 | 0.88 | 0.18 | 0.39 |
| **Sana-1.6B** (Ours) | 1.6 | **0.66** | 0.99 | 0.77 | 0.62 | 0.88 | 0.21 | 0.47 |

Table 11: **Comparison of SOTA methods on DPG-Bench and ImageReward with details.** The table includes different metrics such as overall performance, entity, attribute, relation, and other categories.

| Model | Params (B) | Overall ↑ | Global | Entity | Attribute | Relation | Other | ImageReward ↑ |
|---|---|---|---|---|---|---|---|---|
| **512 ×512 resolution** | | | | | | | | |
| PixArt-$\alpha$ (Chen et al., 2024b) | 0.6 | **71.6** | 81.7 | 80.1 | 80.4 | 81.7 | 76.5 | 0.92 |
| PixArt-$\Sigma$ (Chen et al., 2024a) | 0.6 | **79.5** | 87.5 | 87.1 | 86.5 | 84.0 | 86.1 | 0.97 |
| **Sana-0.6B** (Ours) | 0.6 | **84.3** | 82.6 | 90.0 | 88.6 | 90.1 | 91.9 | 0.93 |
| **Sana-1.6B** (Ours) | 0.6 | **85.5** | 90.3 | 91.2 | 89.0 | 88.9 | 92.0 | 1.04 |
| **1024 ×1024 resolution** | | | | | | | | |
| LUMINA-Next (Zhuo et al., 2024) | 2.0 | **74.6** | 82.8 | 88.7 | 86.4 | 80.5 | 81.8 | - |
| SDXL (Podell et al., 2023) | 2.6 | **74.7** | 83.3 | 82.4 | 80.9 | 86.8 | 80.4 | 0.69 |
| PlayGroundv2.5 (Li et al., 2024a) | 2.6 | **75.5** | 83.1 | 82.6 | 81.2 | 84.1 | 83.5 | 1.09 |
| Hunyuan-DiT (Li et al., 2024c) | 1.5 | **78.9** | 84.6 | 80.6 | 88.0 | 74.4 | 86.4 | 0.92 |
| PixArt-$\Sigma$ (Chen et al., 2024a) | 0.6 | **80.5** | 86.9 | 82.9 | 88.9 | 86.6 | 87.7 | 0.87 |
| DALLE3 (OpenAI, 2023) | - | **83.5** | 91.0 | 89.6 | 88.4 | 90.6 | 89.8 | - |
| SD3-medium (Esser et al., 2024) | 2.0 | **84.1** | 87.9 | 91.0 | 88.8 | 80.7 | 88.7 | 0.86 |
| FLUX-dev (Labs, 2024) | 12.0 | **84.0** | 82.1 | 89.5 | 88.7 | 91.1 | 89.4 | - |
| FLUX-schnell (Labs, 2024) | 12.0 | **84.8** | 91.2 | 91.3 | 89.7 | 86.5 | 87.0 | 0.91 |
| **Sana**-0.6B (Ours) | 0.6 | **83.6** | 83.0 | 89.5 | 89.3 | 90.1 | 90.2 | 0.97 |
| **Sana**-1.6B (Ours) | 1.6 | **84.8** | 86.0 | 91.5 | 88.9 | 91.9 | 90.7 | 0.99 |

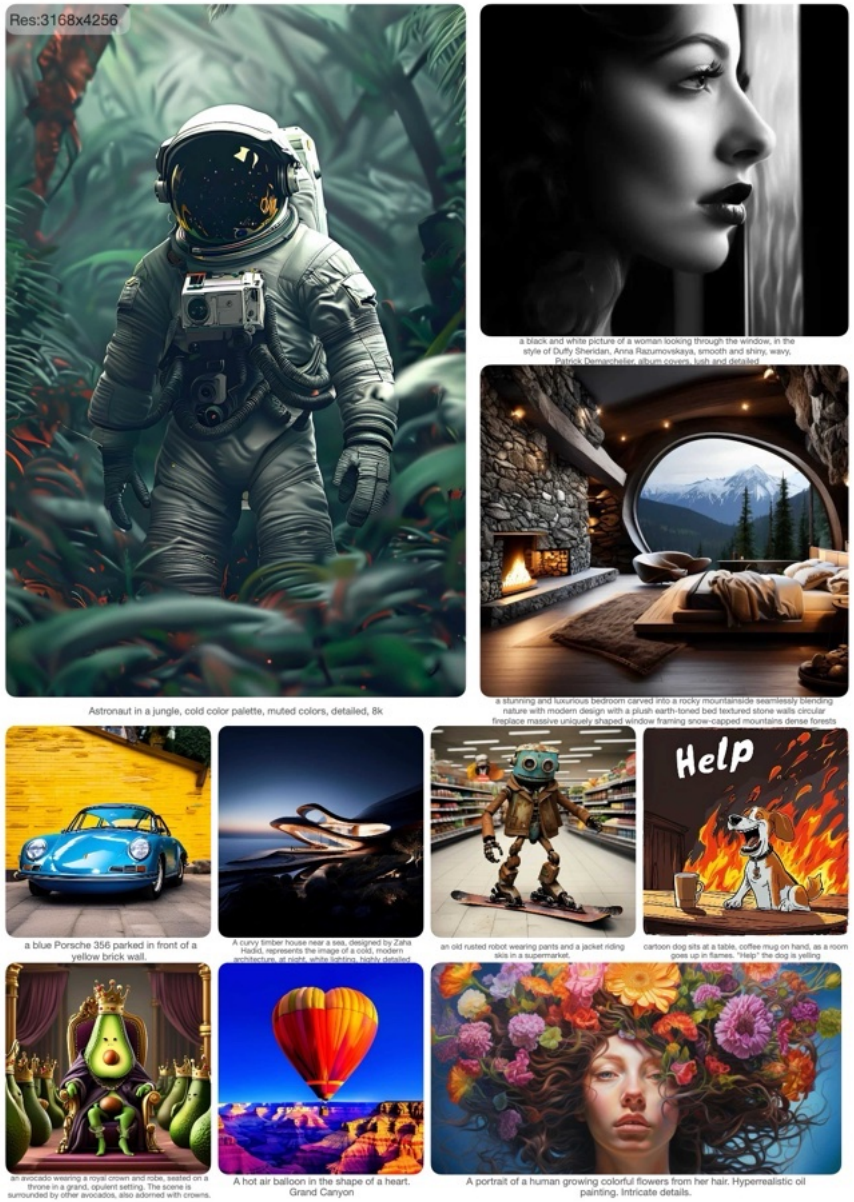
NAVER AI LAB

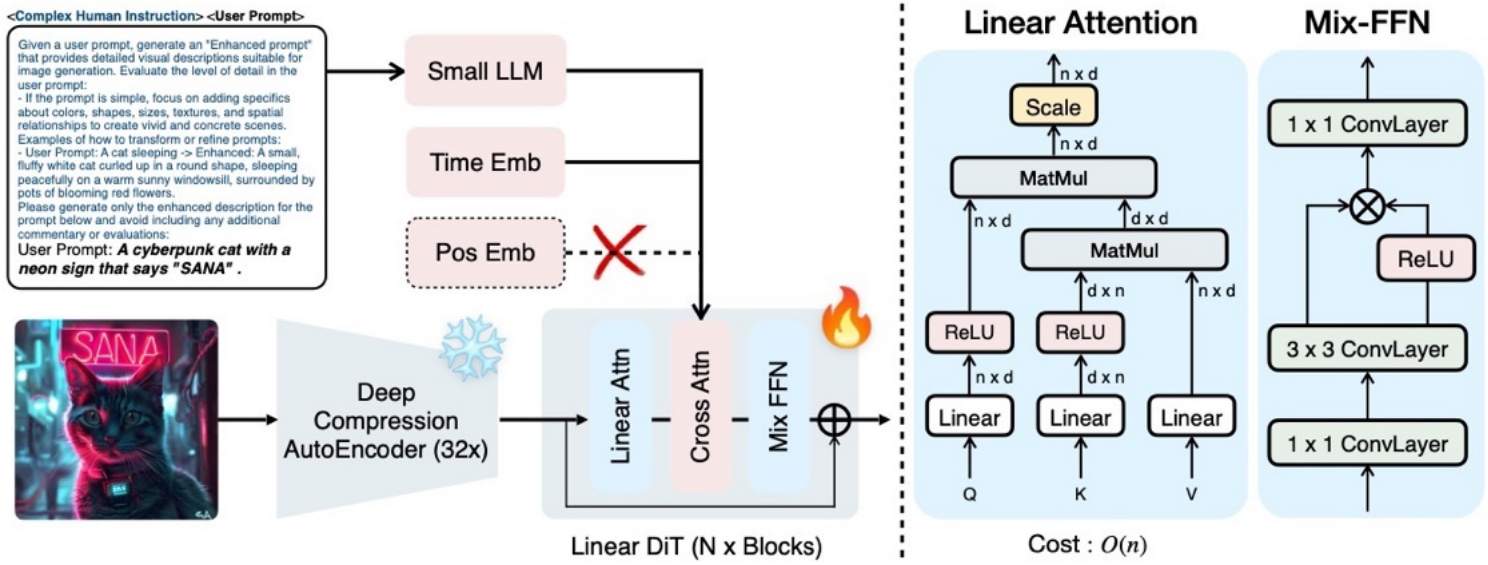
Table 14: Comparison of throughput and latency under different resolutions. All models tested on an A100 GPU with FP16 precision.

| Methods | Speedup | Throughput(/s) | Latency(ms) | Methods | Speedup | Throughput(/s) | Latency(ms) |
|---|---|---|---|---|---|---|---|
| | 512×512 Resolution | | | | 1024×1024 Resolution | | |
| SD3 | 7.6x | 1.14 | 1.4 | SD3 | 7.0x | 0.28 | 4.4 |
| FLUX-schnell | 10.5x | 1.58 | 0.7 | FLUX-schnell | 12.5x | 0.50 | 2.1 |
| FLUX-dev | 1.0x | 0.15 | 7.9 | FLUX-dev | 1.0x | 0.04 | 23 |
| PixArt-Σ | 10.3x | 1.54 | 1.2 | PixArt-Σ | 10.0x | 0.40 | 2.7 |
| HunyuanDiT | 1.3x | 0.20 | 5.1 | HunyuanDiT | 1.2x | 0.05 | 18 |
| **Sana-0.6B** | 44.5x | 6.67 | 0.8 | **Sana-0.6B** | 43.0x | 1.72 | 0.9 |
| **Sana-1.6B** | 25.6x | 3.84 | 0.6 | **Sana-1.6B** | 25.2x | 1.01 | 1.2 |
| | 2048×2048 Resolution | | | | 4096×4096 Resolution | | |
| SD3 | 5.0x | 0.04 | 22 | SD3 | 4.0x | 0.004 | 230 |
| FLUX-schnell | 11.2x | 0.09 | 10.5 | FLUX-schnell | 13.0x | 0.013 | 76 |
| FLUX-dev | 1.0x | 0.008 | 117 | FLUX-dev | 1.0x | 0.001 | 1023 |
| PixArt-Σ | 7.5x | 0.06 | 18.1 | PixArt-Σ | 5.0x | 0.005 | 186 |
| HunyuanDiT | 1.2x | 0.01 | 96 | HunyuanDiT | 1.0x | 0.001 | 861 |
| **Sana-0.6B** | 53.8x | 0.43 | 2.5 | **Sana-0.6B** | 104.0x | 0.104 | 9.6 |
| **Sana-1.6B** | 31.2x | 0.25 | 4.1 | **Sana-1.6B** | 66.0x | 0.066 | 5.9 |

# SANA | Results

NAVER AI LAB




(a). Architecture overview of our Sana.

(b). Linear DiT Module.

# Visual AutoRegressive Modeling (VAR)

## Scalable Image Generation via Next-Scale Prediction with code


NAVER AI Lab, Generation research

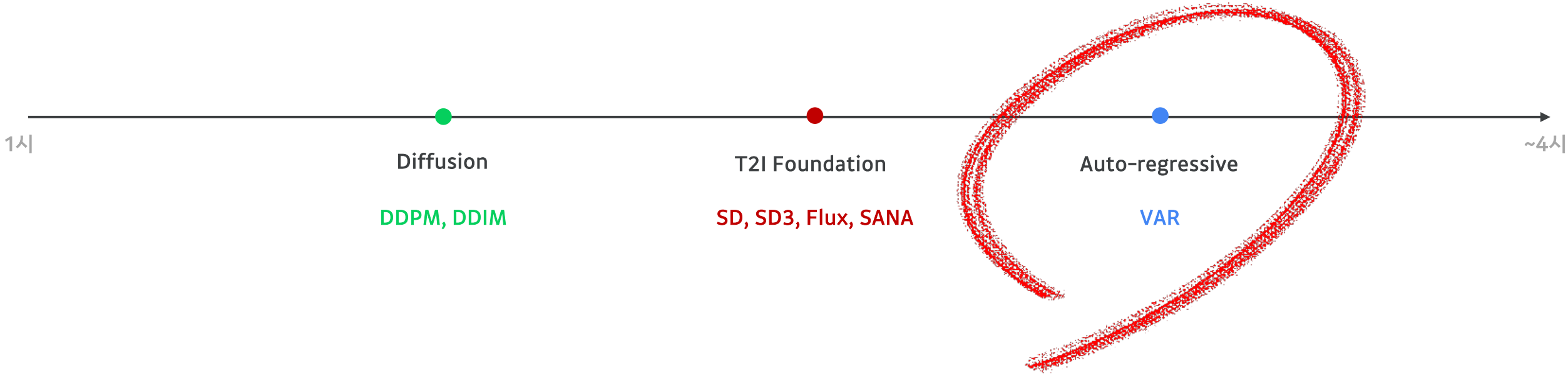
김준호

https://github.com/taki0112

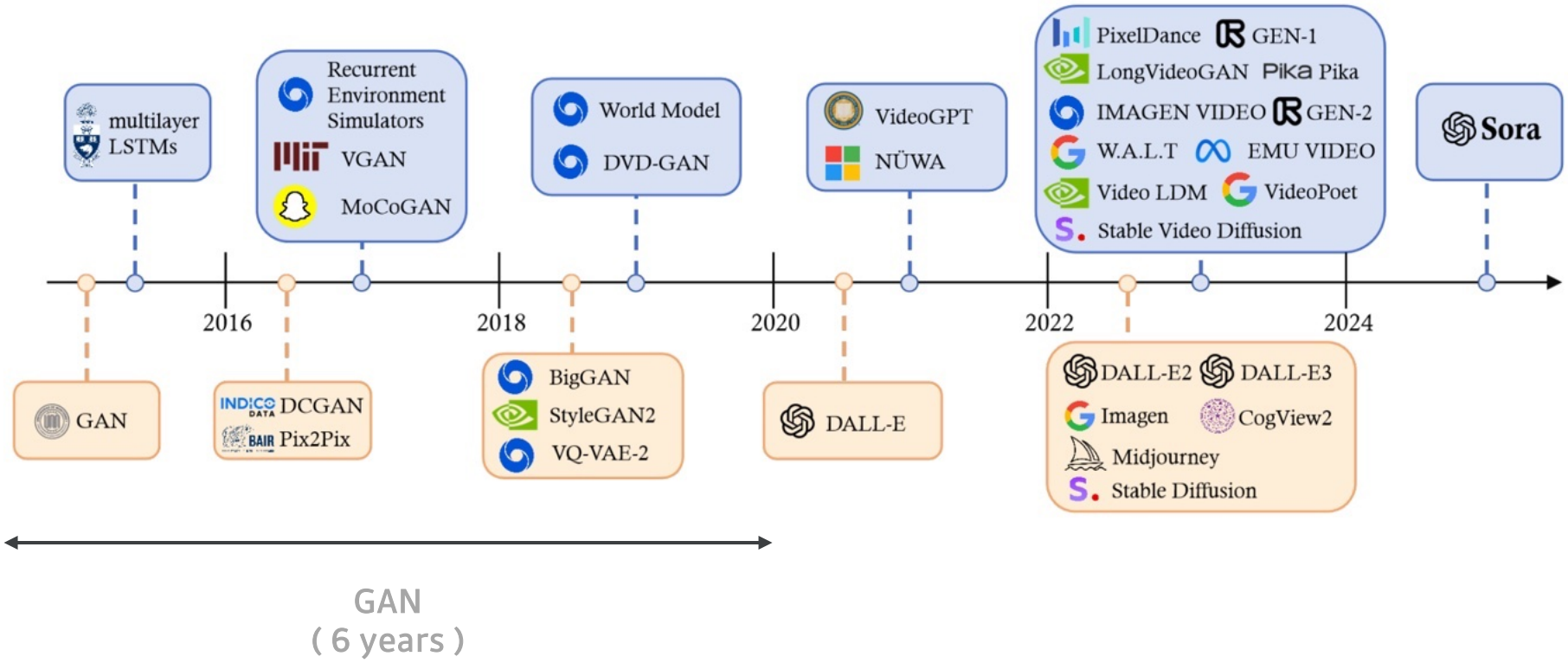

NAVER AI LAB

https://github.com/taki0112/pseudo_var

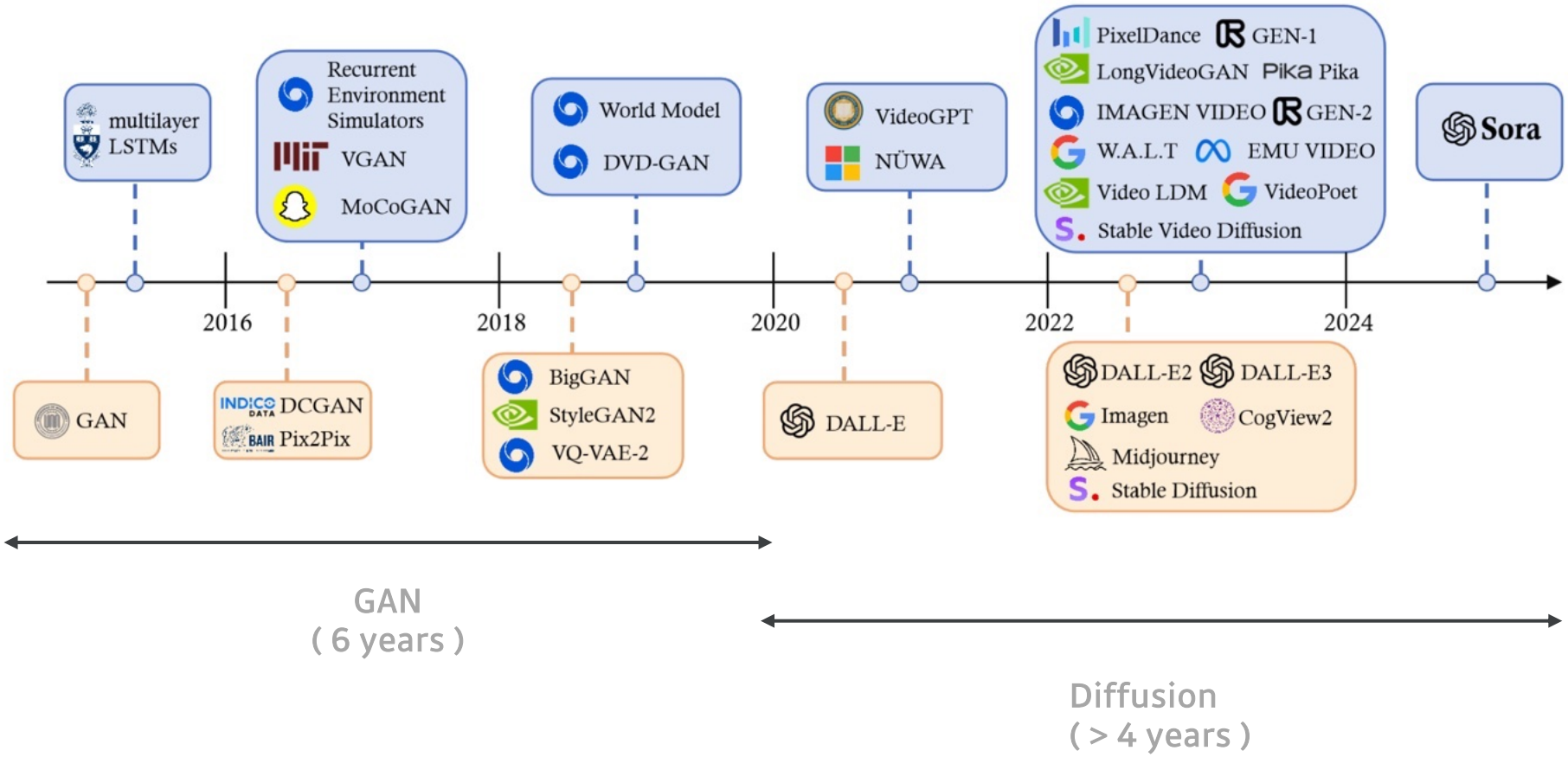
NAVER AI LAB


1시
Diffusion
DDPM, DDIM
T2I Foundation
SD, SD3, Flux, SANA
Auto-regressive
VAR
~4시

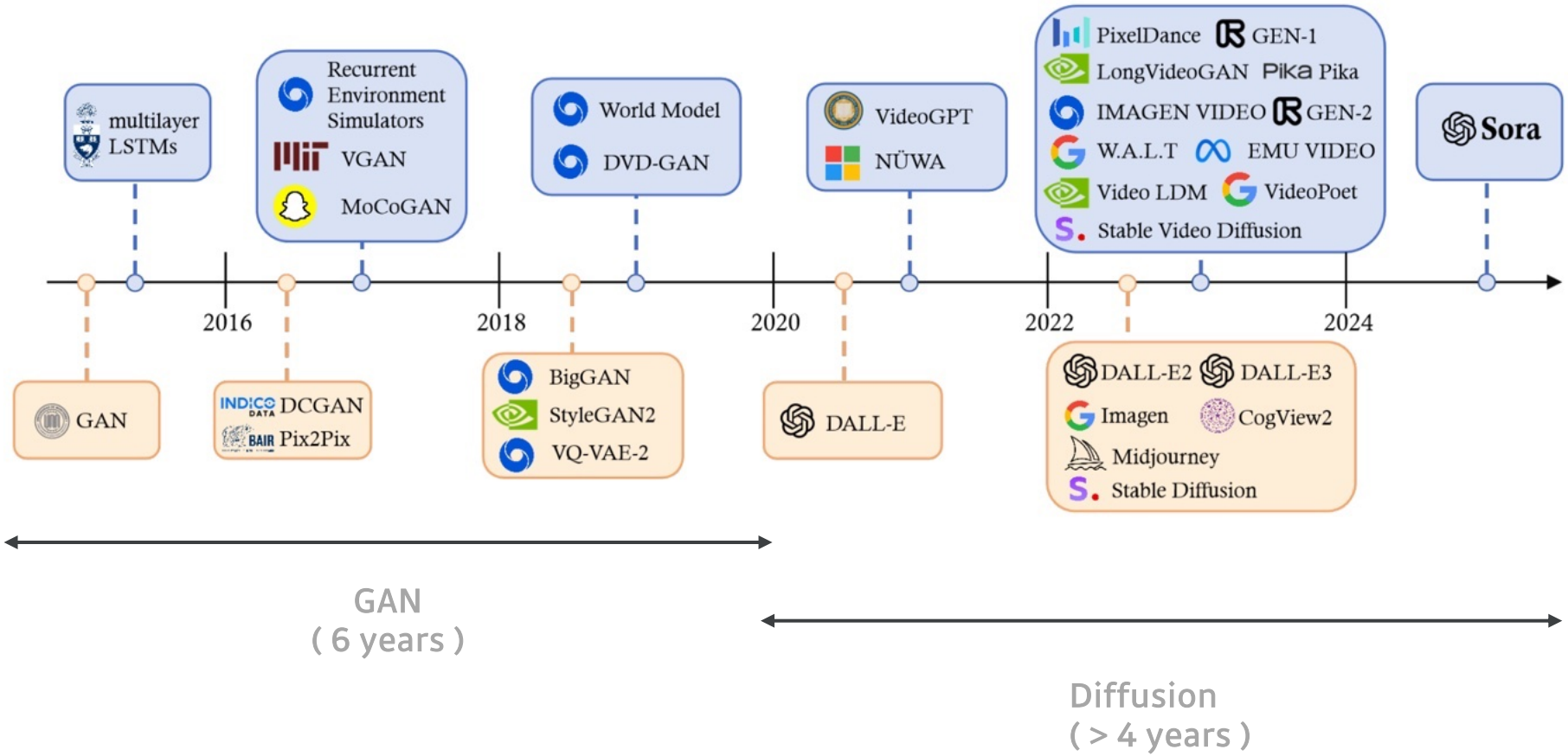
NAVER AI LAB


multilayer LSTMs
Recurrent Environment Simulators
VGAN
MoCoGAN
World Model
DVD-GAN
VideoGPT
NÜWA
PixelDance
GEN-1
LongVideoGAN
Pika
IMAGEN VIDEO
GEN-2
W.A.L.T
EMU VIDEO
Video LDM
VideoPoet
Stable Video Diffusion
Sora
2016
2018
2020
2022
2024
GAN
DCGAN
Pix2Pix
BigGAN
StyleGAN2
VQ-VAE-2
DALL-E
DALL-E2
DALL-E3
Imagen
CogView2
Midjourney
Stable Diffusion
GAN
( 6 years )

NAVER AI LAB


multilayer LSTMs
Recurrent Environment Simulators
VGAN
MoCoGAN
World Model
DVD-GAN
VideoGPT
NÜWA
PixelDance
GEN-1
LongVideoGAN
Pika Pika
IMAGEN VIDEO
GEN-2
W.A.L.T
EMU VIDEO
Video LDM
VideoPoet
Stable Video Diffusion
Sora
2016
2018
2020
2022
2024
GAN
DCGAN
Pix2Pix
BigGAN
StyleGAN2
VQ-VAE-2
DALL-E
DALL-E2
DALL-E3
Imagen
CogView2
Midjourney
Stable Diffusion
GAN
( 6 years )
Diffusion
( > 4 years )

NAVER AI LAB



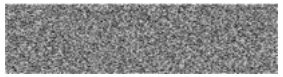

# VAR

2024.04

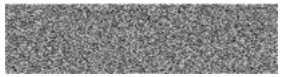
Autoregressive 방식

Visual Autoregressive Modeling: Scalable Image Generation via Next-Scale Prediction

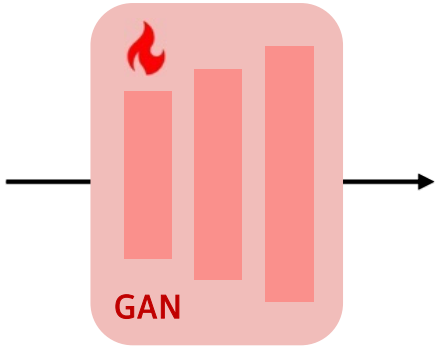
NAVER AI LAB



[bs, 512]

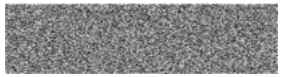


[bs, 512]

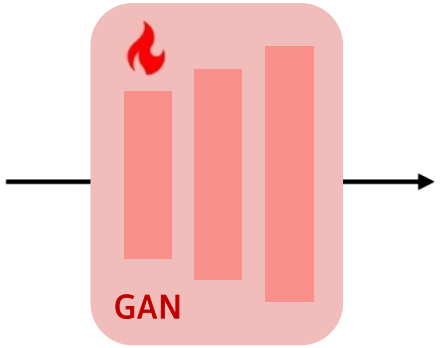



[bs, c, h, w]

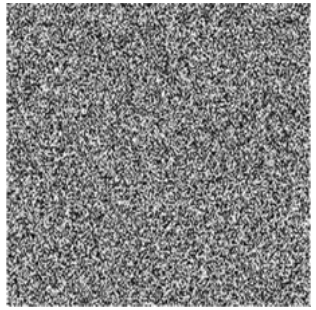
NAVER AI LAB



[bs, 512]

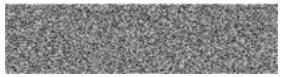



[bs, c, h, w]

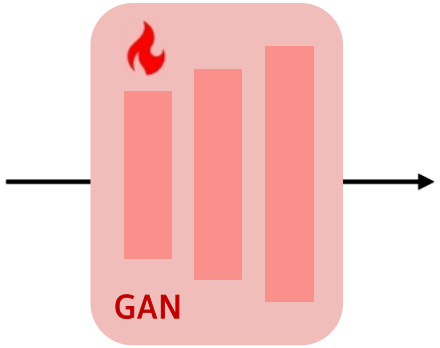


[bs, c, h, w]

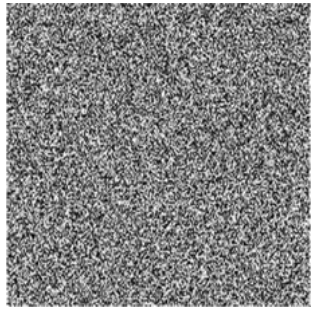
NAVER AI LAB



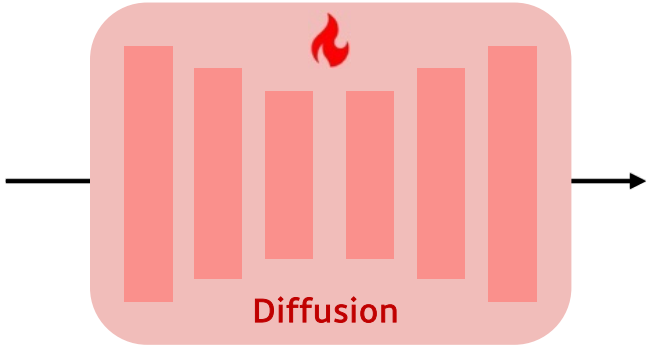
[bs, 512]




[bs, c, h, w]

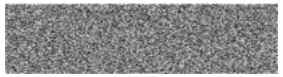


[bs, c, h, w]

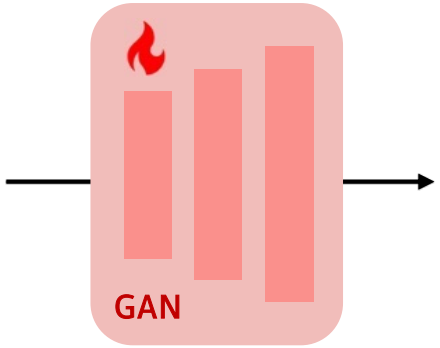



[bs, c, h, w]

NAVER AI LAB



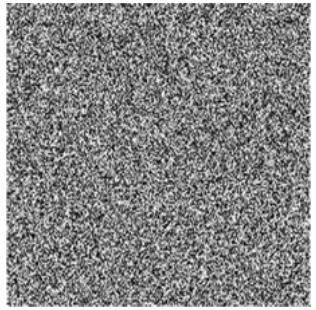
[bs, 512]



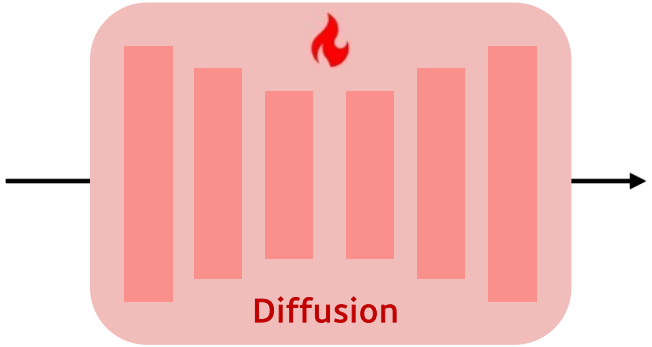

[bs, c, h, w]

속도 빠름
퀄이 별로

과거의 영광

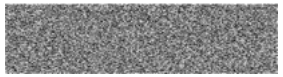


[bs, c, h, w]

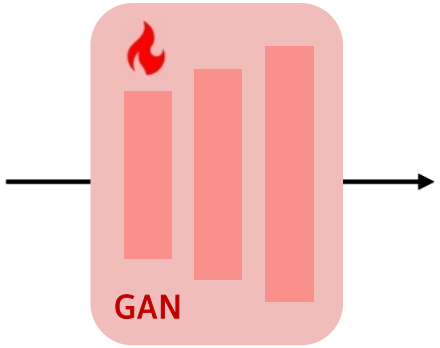



[bs, c, h, w]

NAVER AI LAB



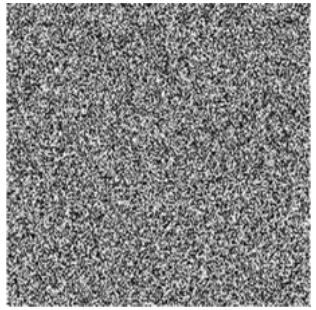
[bs, 512]



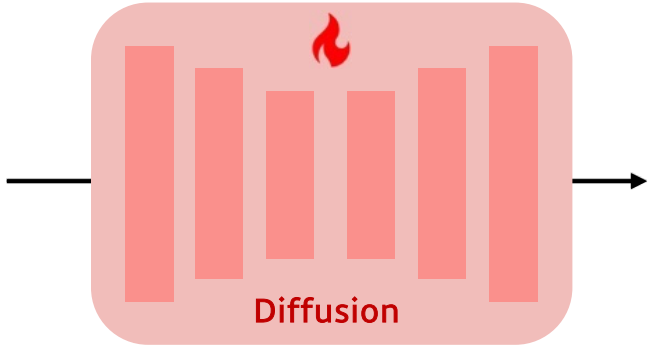

[bs, c, h, w]

속도 빠름
퀄이 별로

과거의 영광



[bs, c, h, w]




[bs, c, h, w]

속도 느림
퀄이 좋음

현재의 대세

NAVER AI LAB

## # 2014 ~ 2016

- Unconditional generation
  - GANs
  - Diverse loss
- Conditional generation
  - ACGAN
  - Multi-task discriminator
  - Projection discriminator

## # 2017 ~ 2018

- Progressive GAN
  - Progressive training
- BigGAN
  - Conditional batch normalization
  - Large scale
  - Truncation trick
- StyleGAN
  - Disentangle the latent space with mapping layer
  - Style Mixing (determine the coarse, middle, fine style)
    - A module = Global aspects
    - B module = Local aspects
  - Truncation trick

## # 2019 ~ 2020

- StyleGAN2
  - StyleGAN + Weight modulation + Lazy regularization.
- DiffAugment
  - Prevent the overfitting in a discriminator.
  - Apply the differentiable augmentation to generator & discriminator.
- ADA
  - Prevent the overfitting in a discriminator.
  - Apply the adaptively augmentation to generator & discriminator.

## # 2014 ~ 2020: Techniques

- Consistency regularization
  - CR-GAN: augmented real images for discriminator.
  - bCR-GAN: augmented real & fake images for discriminator.
  - zCR-GAN: augmented latent codes for generator & discriminator.
  - ICR-GAN: bCR + zCR
- FSMR: Feature Statistics Mixing Regularization
  - Reduce style-bias in discriminator.
- GGDR: Generator Guided Discriminator Regularization
  - Dense supervision for discriminator.

NAVER AI LAB

- DDIM $\alpha$ = DDPM $\bar{\alpha}$
- $\mathbf{x}_t = \sqrt{\alpha_t}\mathbf{x}_0 + \sqrt{1-\alpha_t}\epsilon$ (Forward)
- $\epsilon - \epsilon_\theta(\mathbf{x}_t) = \epsilon - \epsilon_\theta(\sqrt{\alpha_t}\mathbf{x}_0 + \sqrt{1-\alpha_t}\epsilon)$ (Loss)
  - $\epsilon_\theta$ = prediction network
- $\mathbf{x}_{t-1} = \sqrt{\alpha_{t-1}}\underbrace{\left(\frac{\mathbf{x}_t - \sqrt{1-\alpha_t}\epsilon_\theta(\mathbf{x}_t)}{\sqrt{\alpha_t}}\right)}_{\text{predicted } \mathbf{x}_0 = f_\theta(\mathbf{x}_t)} + \underbrace{\sqrt{1-\alpha_{t-1}-\sigma_t^2}\cdot\epsilon_\theta(\mathbf{x}_t)}_{\text{direction pointing to } \mathbf{x}_t} + \underbrace{\sigma_t\epsilon}_{\text{noise}}$ (Reverse)
  - deterministic when $\sigma_t = 0$ → consistency

**Denoising diffusion implicit models (DDIM)**

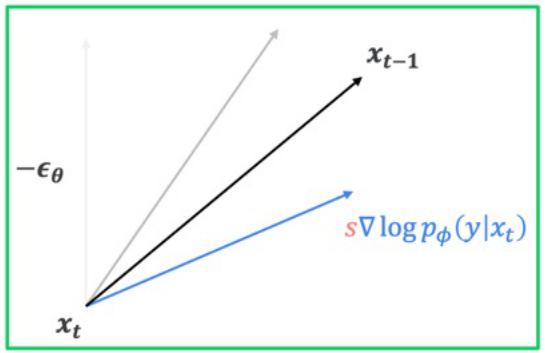
ICLR 2021

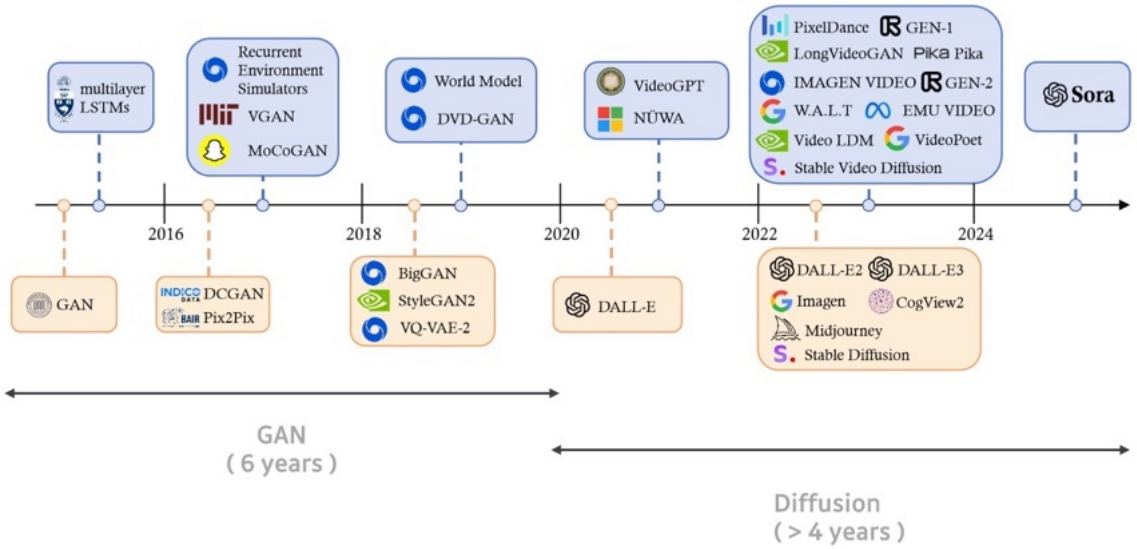
2020 — NeurIPS 2020 — NeurIPS 2021 — 2022

**Denoising diffusion probabilistic models (DDPM)**

- $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon,\ \epsilon \sim \mathcal{N}(0, \mathbf{I})$ (Forward)
  - $\beta_1 = 10^{-4}, \beta_T = 0.02$
  - $\alpha_t := 1 - \beta_t,\ \bar{\alpha}_t := \prod_{s=1}^{t}\alpha_s$
- $\epsilon - \epsilon_\theta(\mathbf{x}_t) = \epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon)$ (Loss)
  - $\epsilon_\theta$ = prediction network
- $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\mathbf{x}_t, t)\right) + \sqrt{\tilde{\beta}_t}\epsilon,\ \epsilon \sim \mathcal{N}(0, \mathbf{I})$ (Reverse)
  - $\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$
    - $\tilde{\beta}_t = \beta_t$로 해도 성능차이 없음

**Diffusion Models Beat GANs on Image Synthesis**






## Youtube

- The recipe of GANs
  - 2014 ~ 2020 GANs 모델 연구 요약 (기초 ~ 심화)
- The diffusion theory
  - diffusion을 이해하기 위한 이론 (기초)
- The applications of diffusion
  - Text-to-image 모델 소개 (심화)

https://github.com/taki0112/pseudo_var

https://github.com/taki0112/diffusion-pytorch

NAVER AI LAB

Figure 1: **Generated samples from Visual AutoRegressive (VAR) transformers trained on ImageNet**. We show 512×512 samples (top), 256×256 samples (middle), and zero-shot image editing results (bottom).

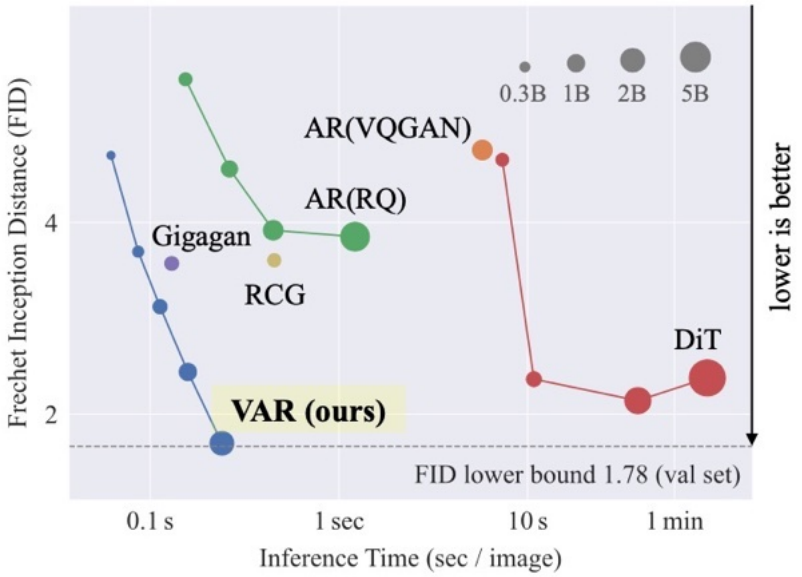



Figure 3: **Scaling behavior** of different models on ImageNet 256×256 conditional generation benchmark. The FID of the validation set serves as a reference lower bound (1.78). VAR with 2B parameters reaches FID 1.80, surpassing L-DiT with 3B or 7B parameters.

속도 빠르고, 성능 좋고, 파라미터 비례

conditional image generation (not text)

NAVER AI LAB

Reference

* https://ljvmiranda921.github.io/notebook/2021/08/08/clip-vqgan/
* https://www.assemblyai.com/blog/what-is-residual-vector-quantization/

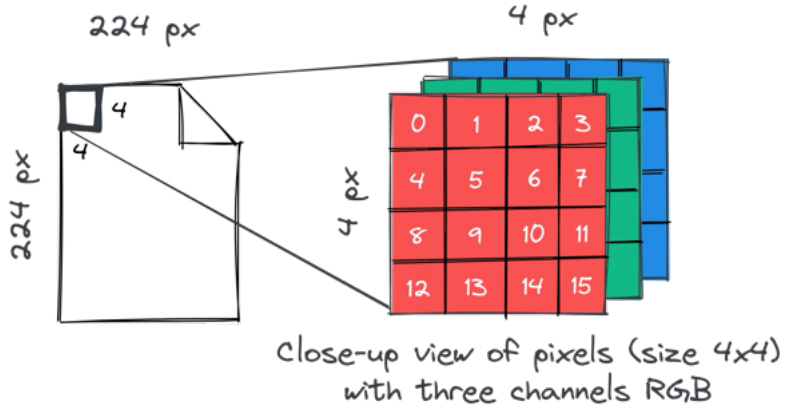
* Preliminaries (Residual Vector Quantization)
* VQVAE Training
* VAR Training
* VAR Inference

https://github.com/taki0112/pseudo_var

# Motivation | 기존 AR 문제점

NAVER AI LAB

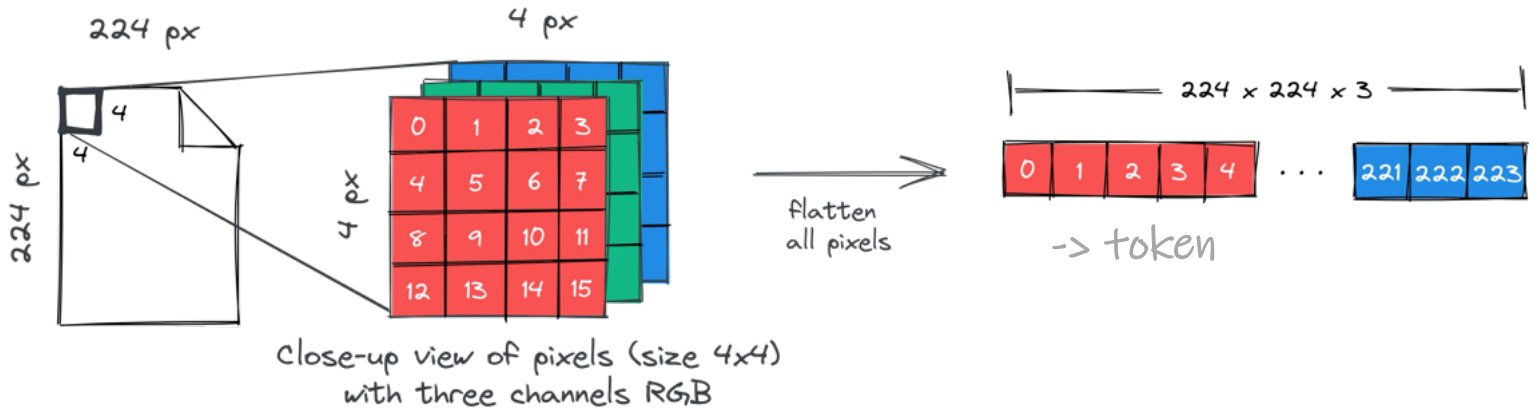
## * Flatten all pixels



# Motivation | 기존 AR 문제점

NAVER AI LAB

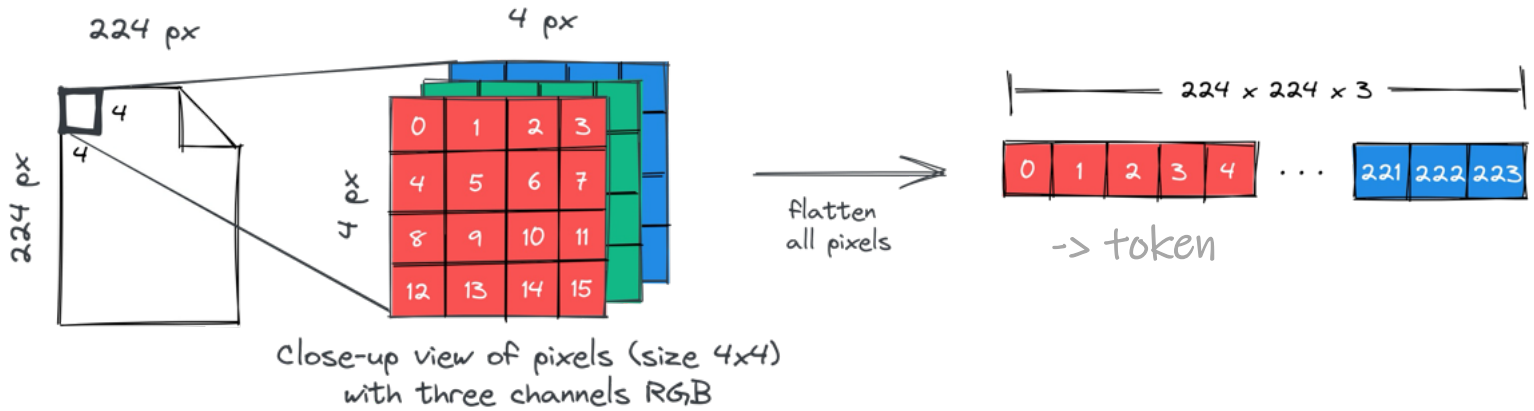
## * Flatten all pixels



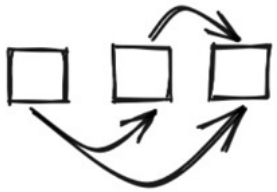
# Motivation | 기존 AR 문제점

NAVER AI LAB

## * Flatten all pixels




## * Naive

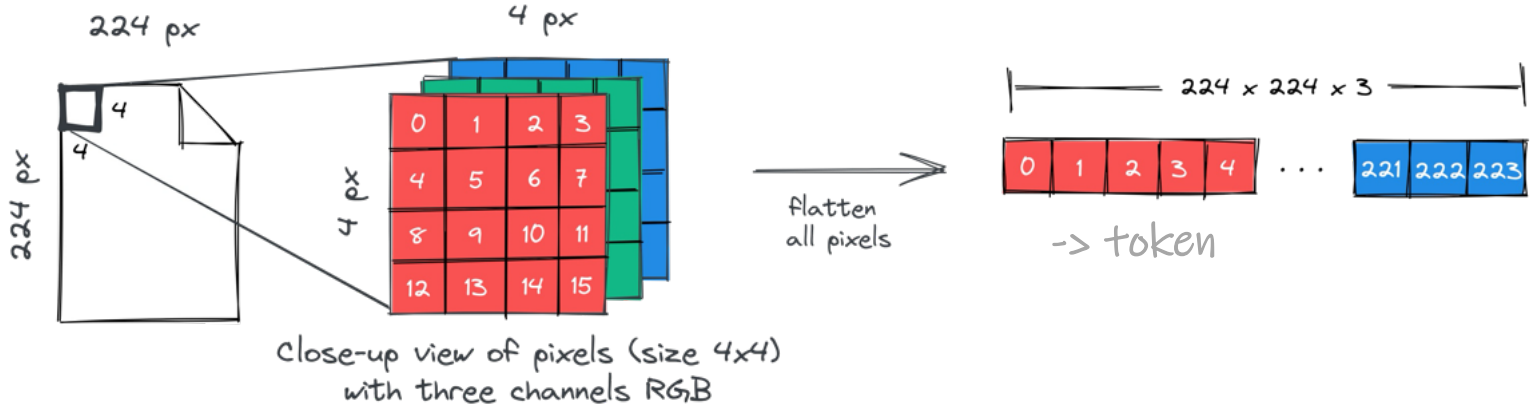


Long-range
Interactions

# Motivation | 기존 AR 문제점

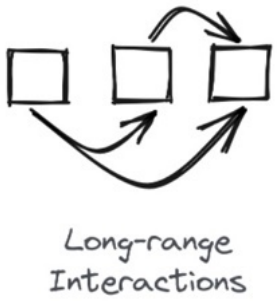
NAVER AI LAB

## * Flatten all pixels




* Spatial locality 무시
  * 1D sequence로 바꿔서
* No generalization
  * Zero-shot task 안됨

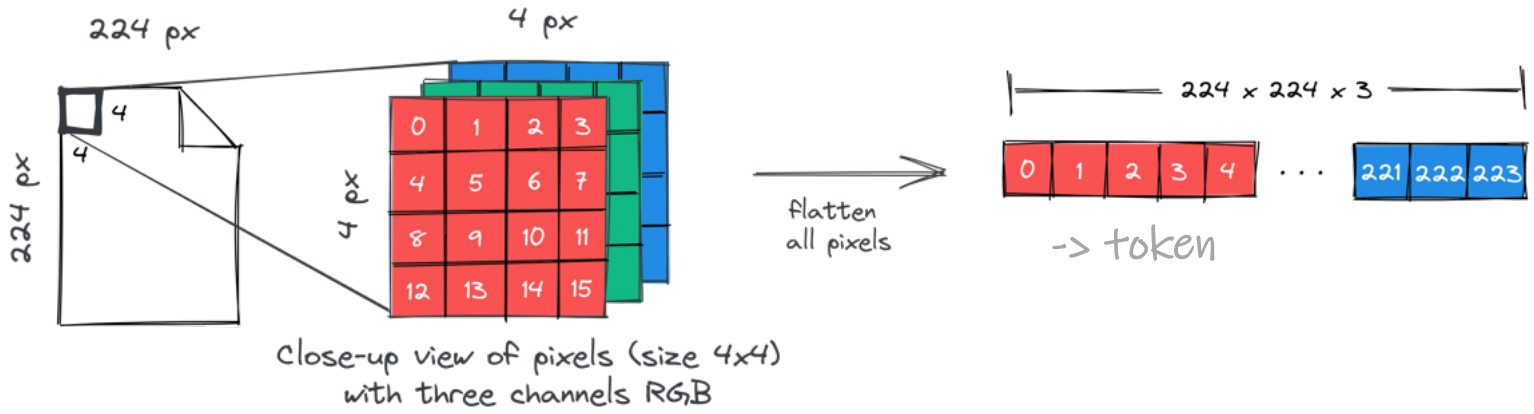
## * Naive



# Motivation | 기존 AR 문제점

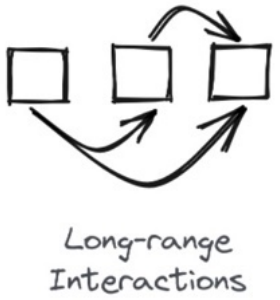
NAVER AI LAB

## * Flatten all pixels

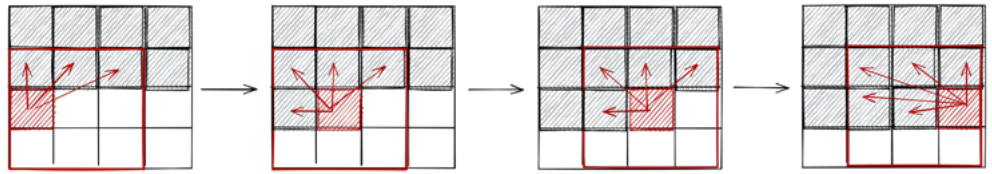



* Spatial locality 무시
  * 1D sequence로 바꿔서
* No generalization
  * Zero-shot task 안됨

## * Naive



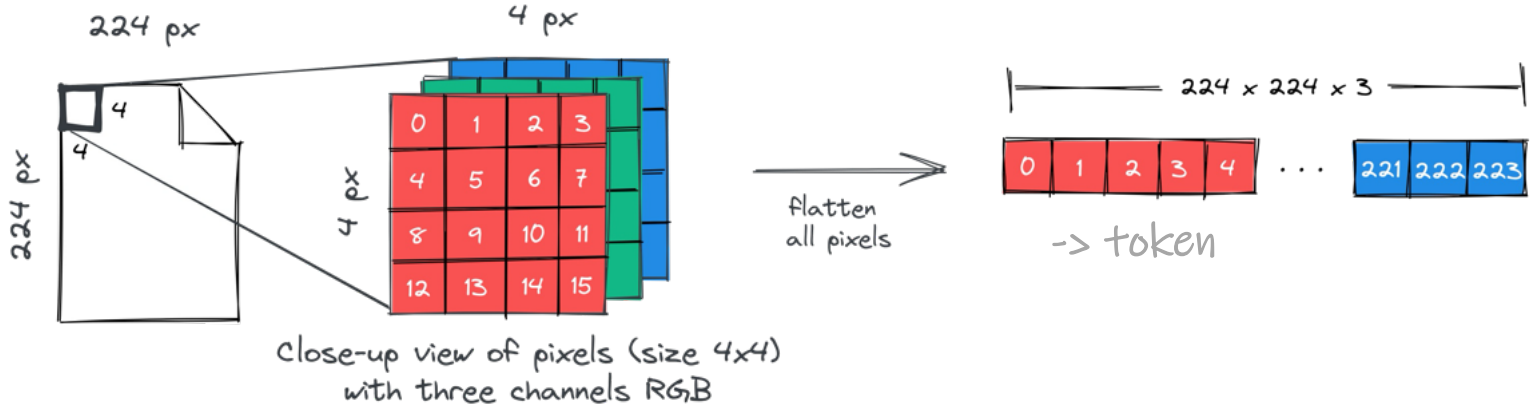

## * Sliding window



# Motivation | 기존 AR 문제점

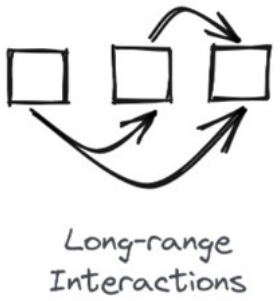
NAVER AI LAB

## * Flatten all pixels

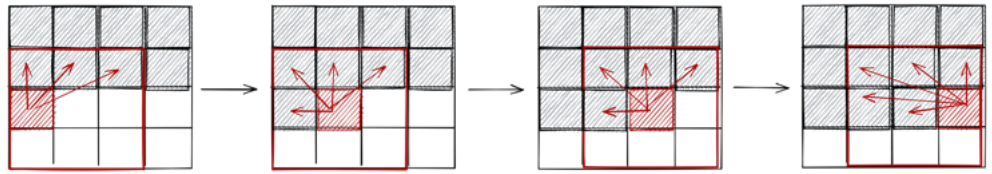



* Spatial locality 무시
  * 1D sequence로 바꿔서
* No generalization
  * Zero-shot task 안됨

## * Naive

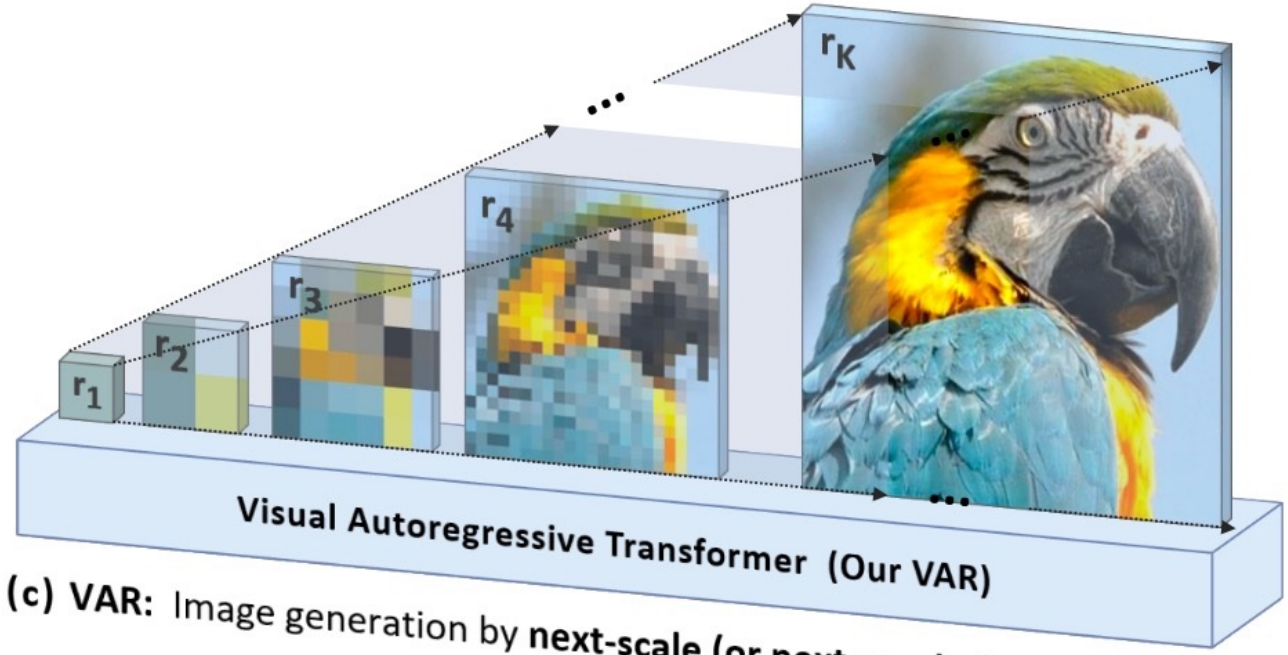



## * Sliding window



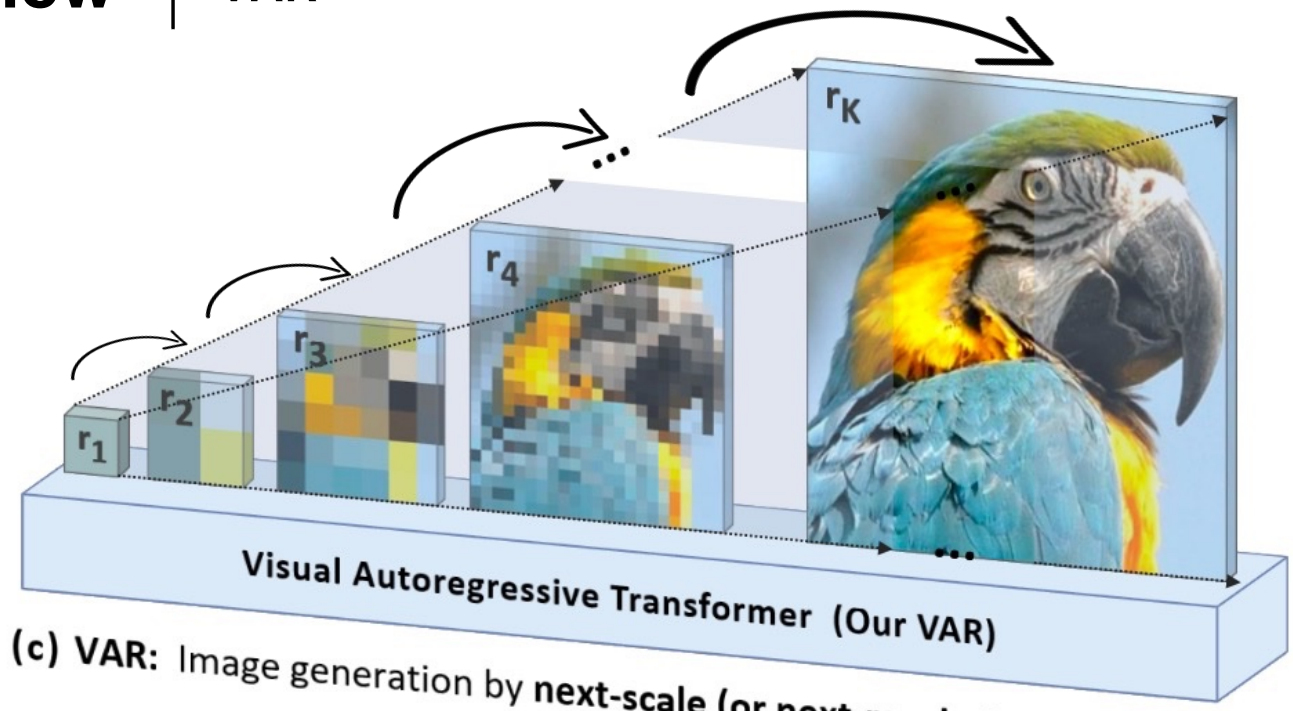
* No efficiency
  * 각 픽셀 = 토큰
* No scalability
  * 낮은 해상도에서만

# Overview | VAR




**(c) VAR:** Image generation by **next-scale (or next-resolution)** prediction

# Overview | VAR



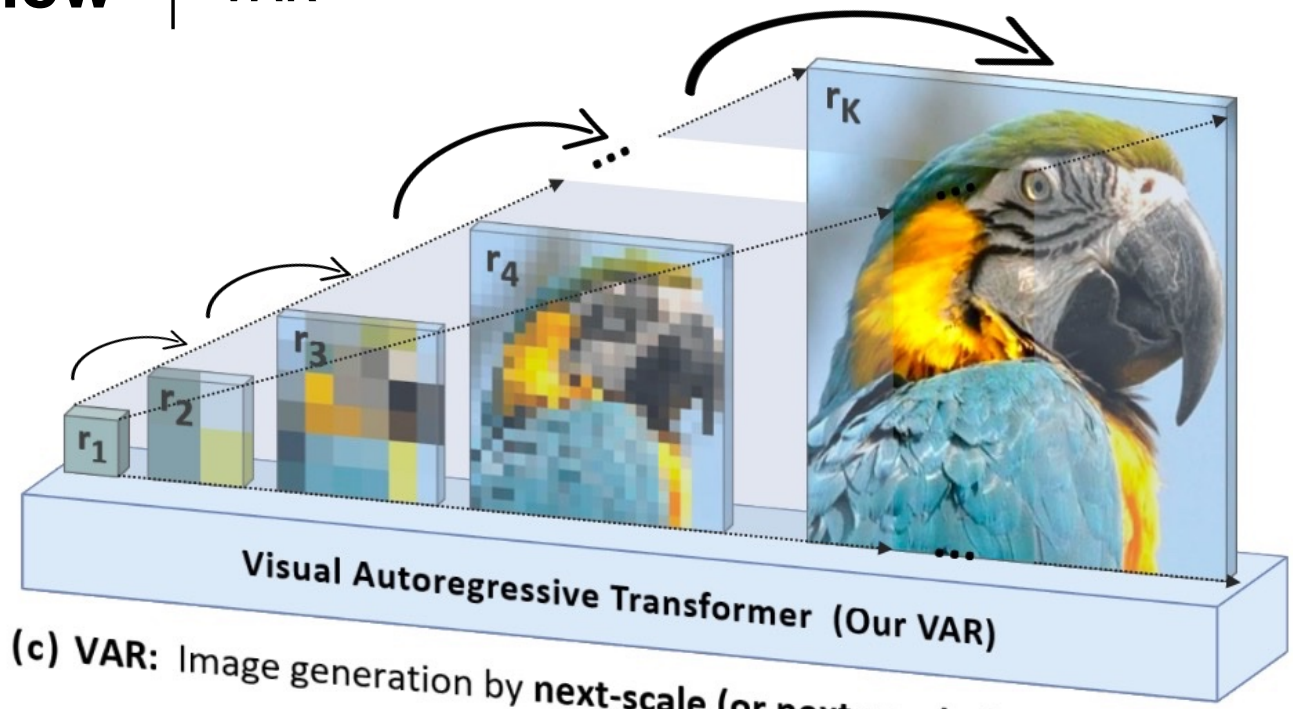

(c) **VAR:** Image generation by **next-scale (or next-resolution)** prediction

토큰 = 이미지

* Multi-scale approach
* Efficiency
* Scalability
* Generalization

# Overview | VAR

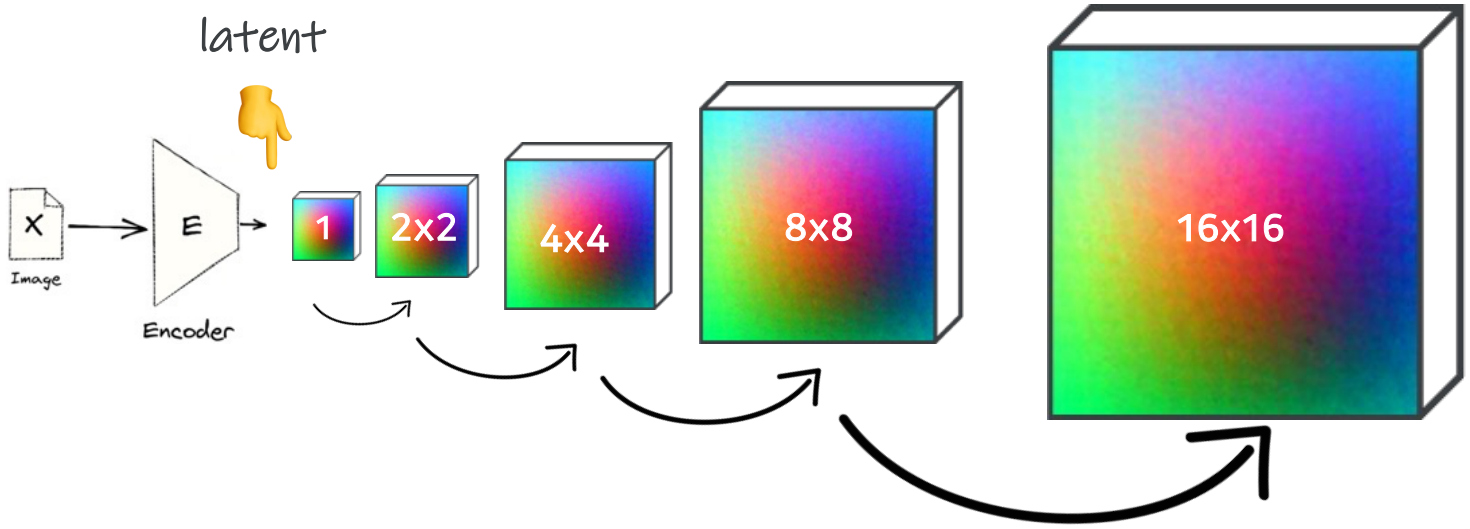
NAVER AI LAB




(c) **VAR:** Image generation by **next-scale (or next-resolution)** prediction

토큰 = 이미지

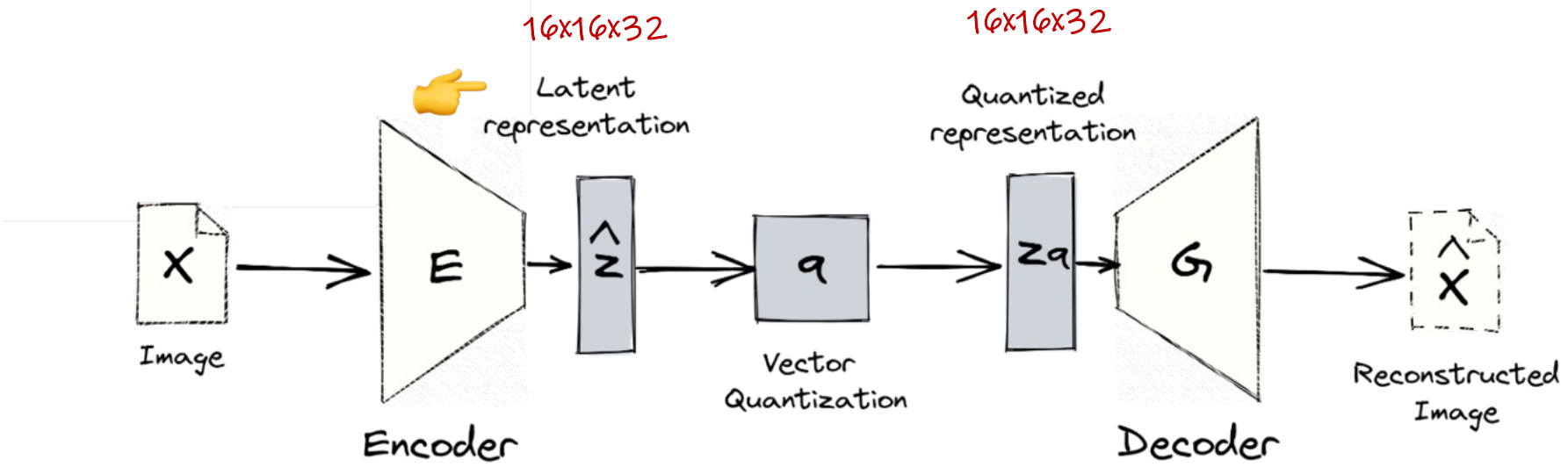
* Multi-scale approach
* Efficiency
* Scalability
* Generalization



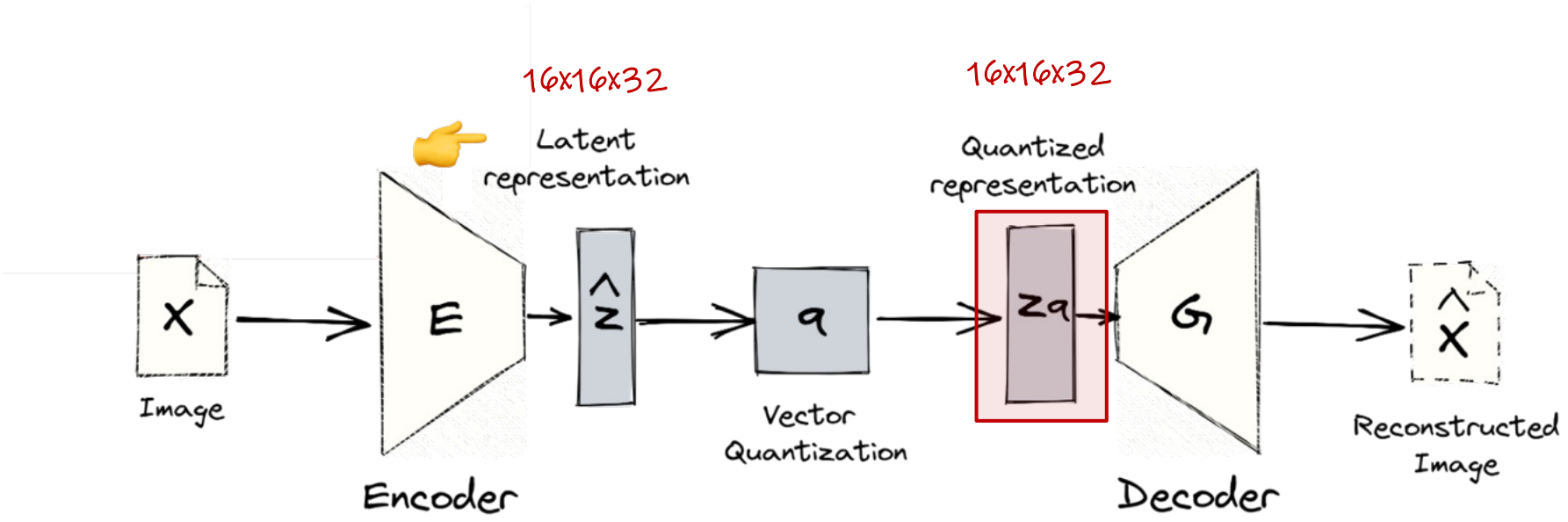

◆ ◆ ◆

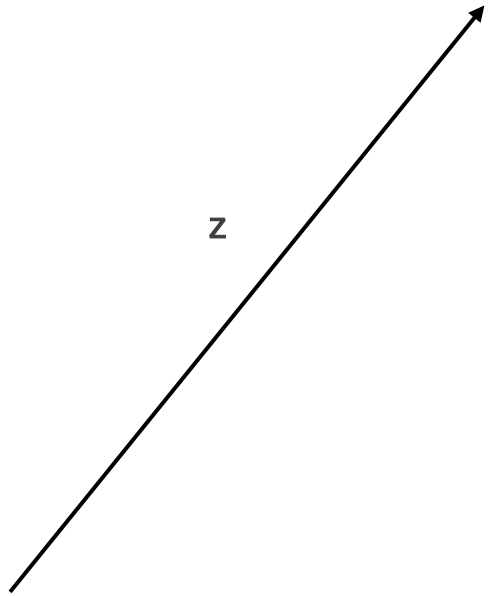
이미지상은 아니고,
latent상에서 함

# Method | Preliminaries: VQVAE

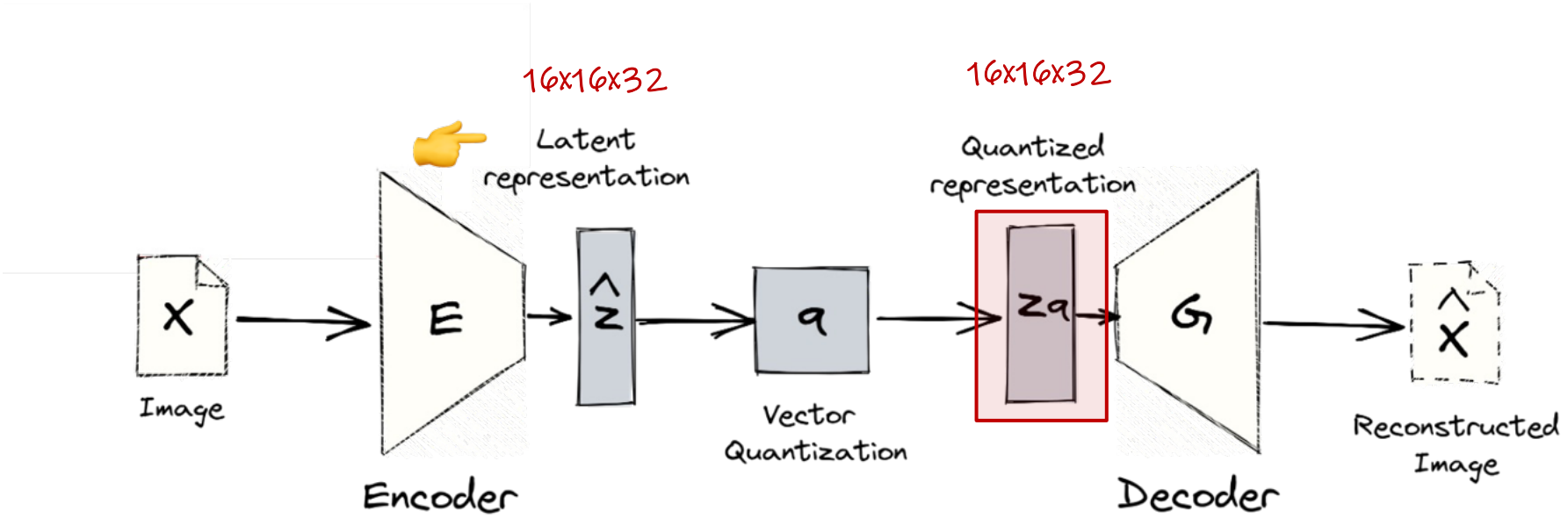


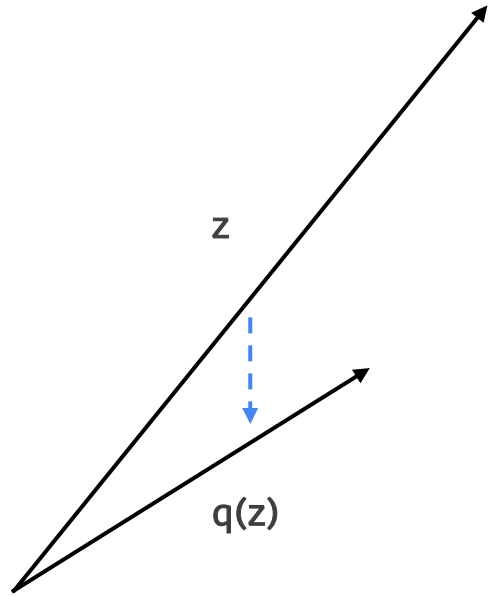
# Method | Preliminaries: VQVAE



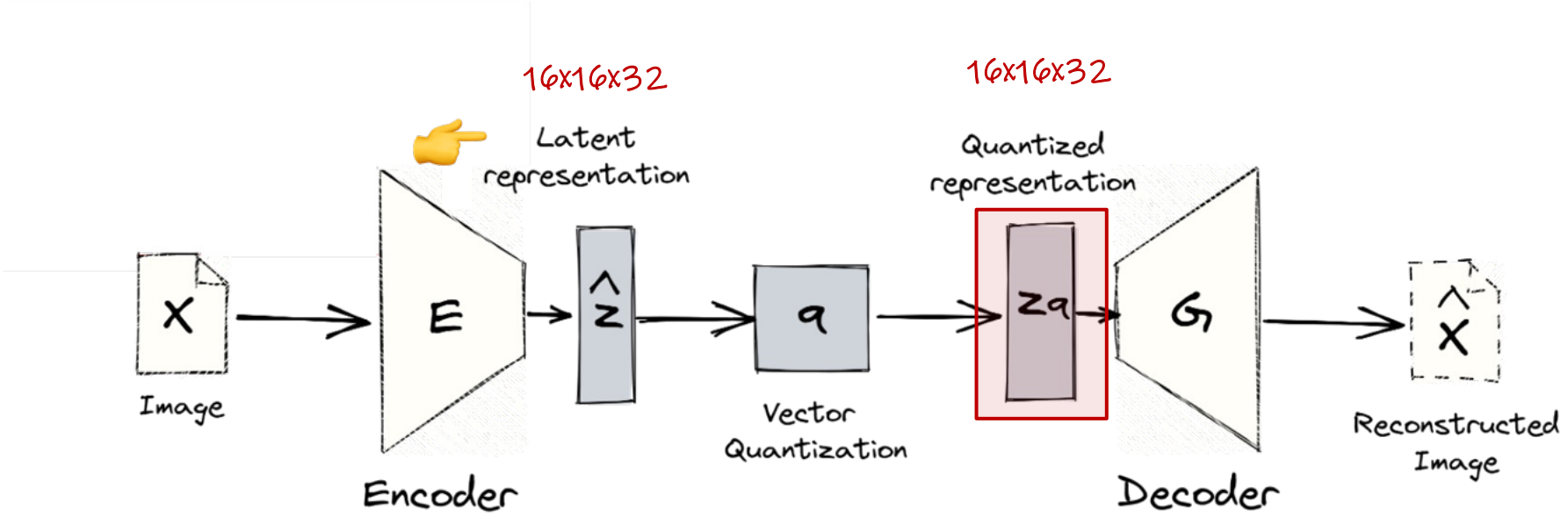



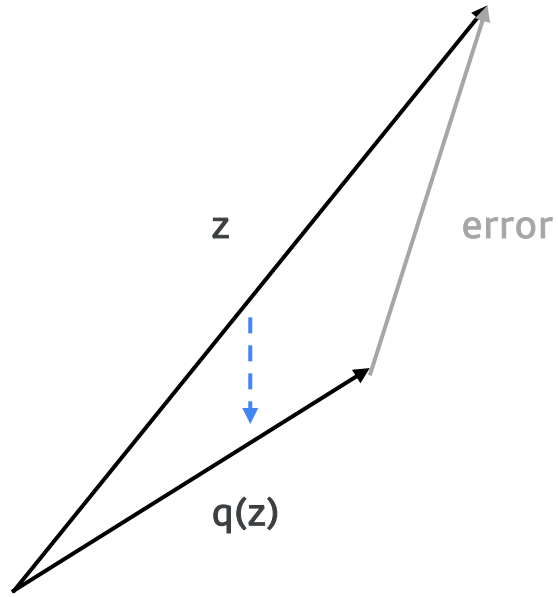
# Method | Preliminaries: VQVAE



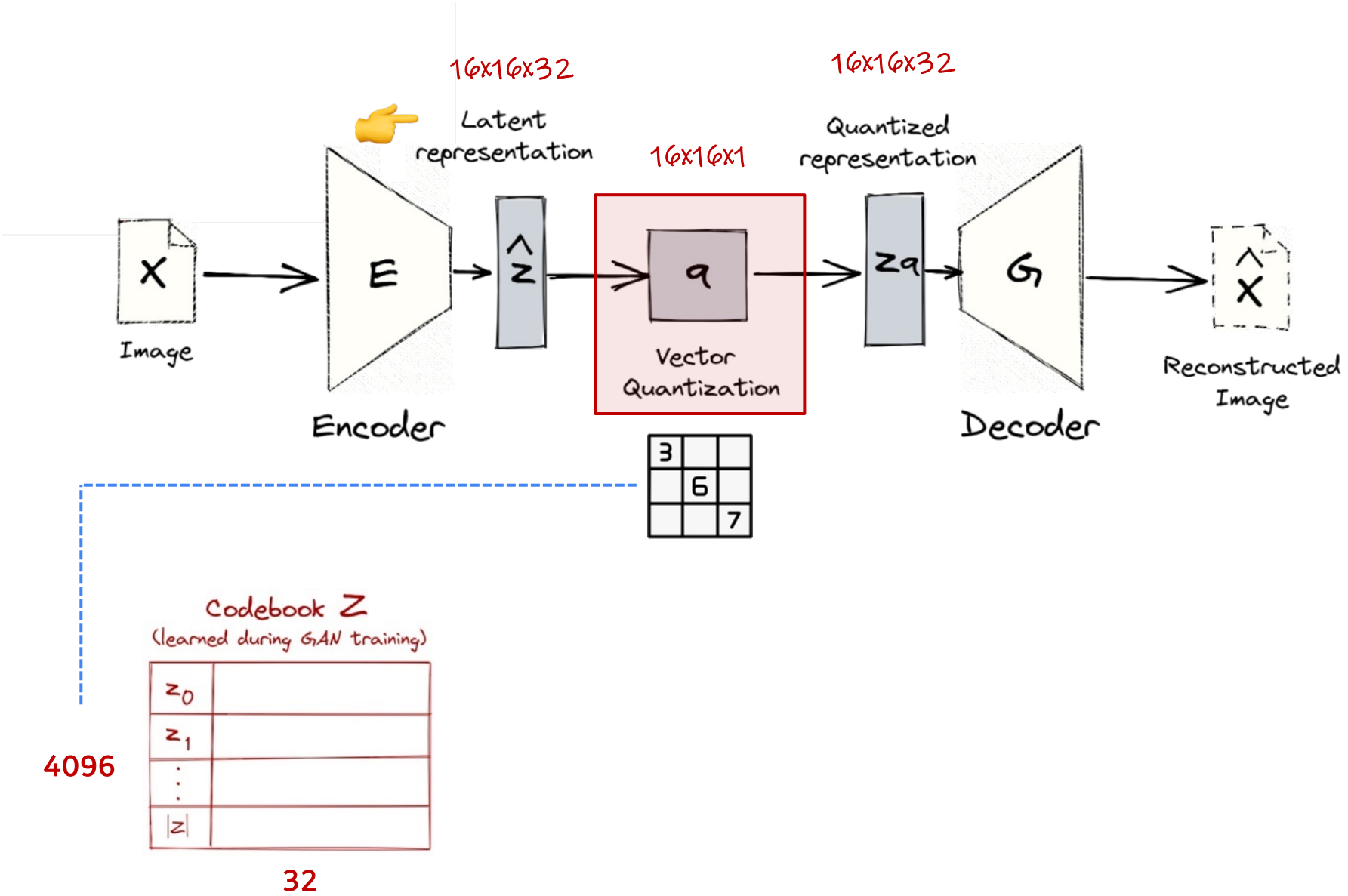



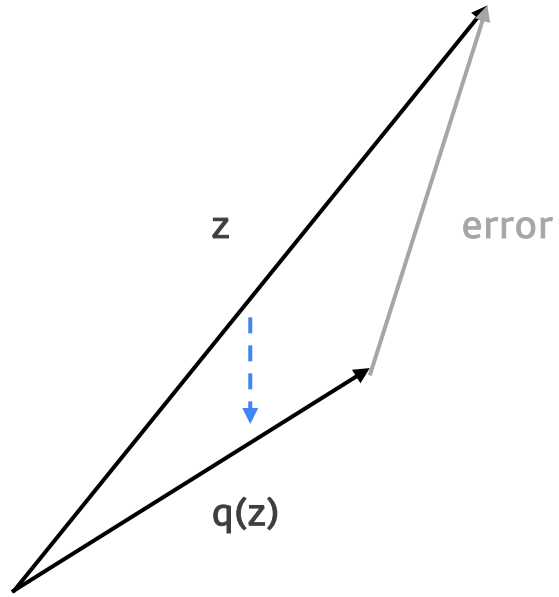
# Method | Preliminaries: VQVAE



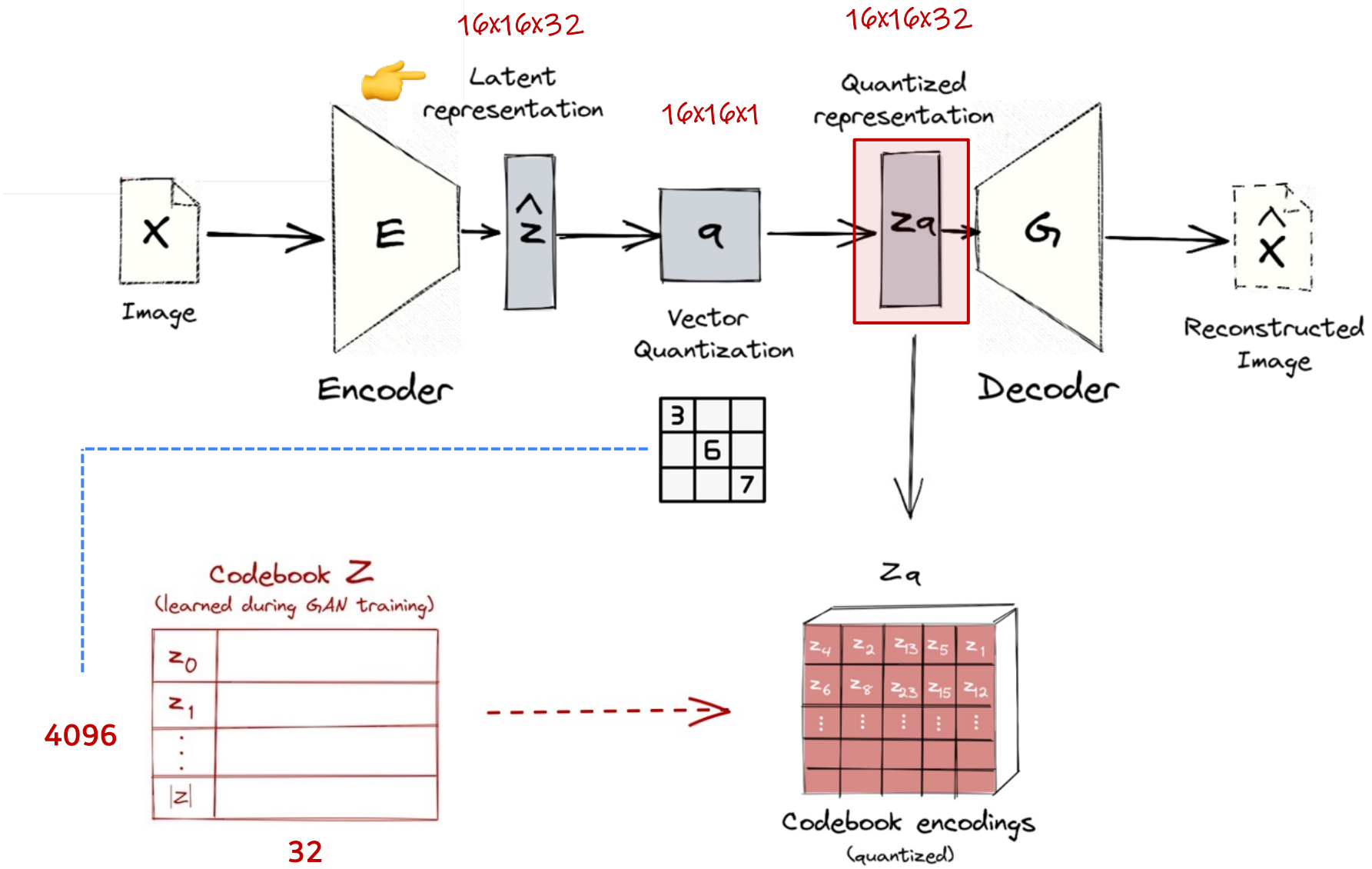



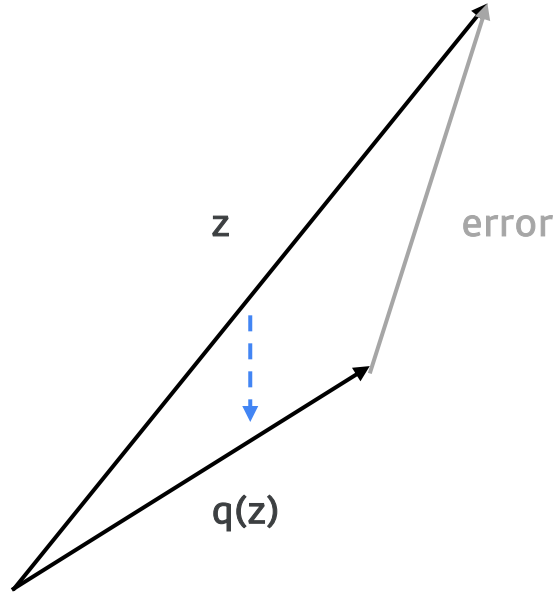
# Method | Preliminaries: VQVAE



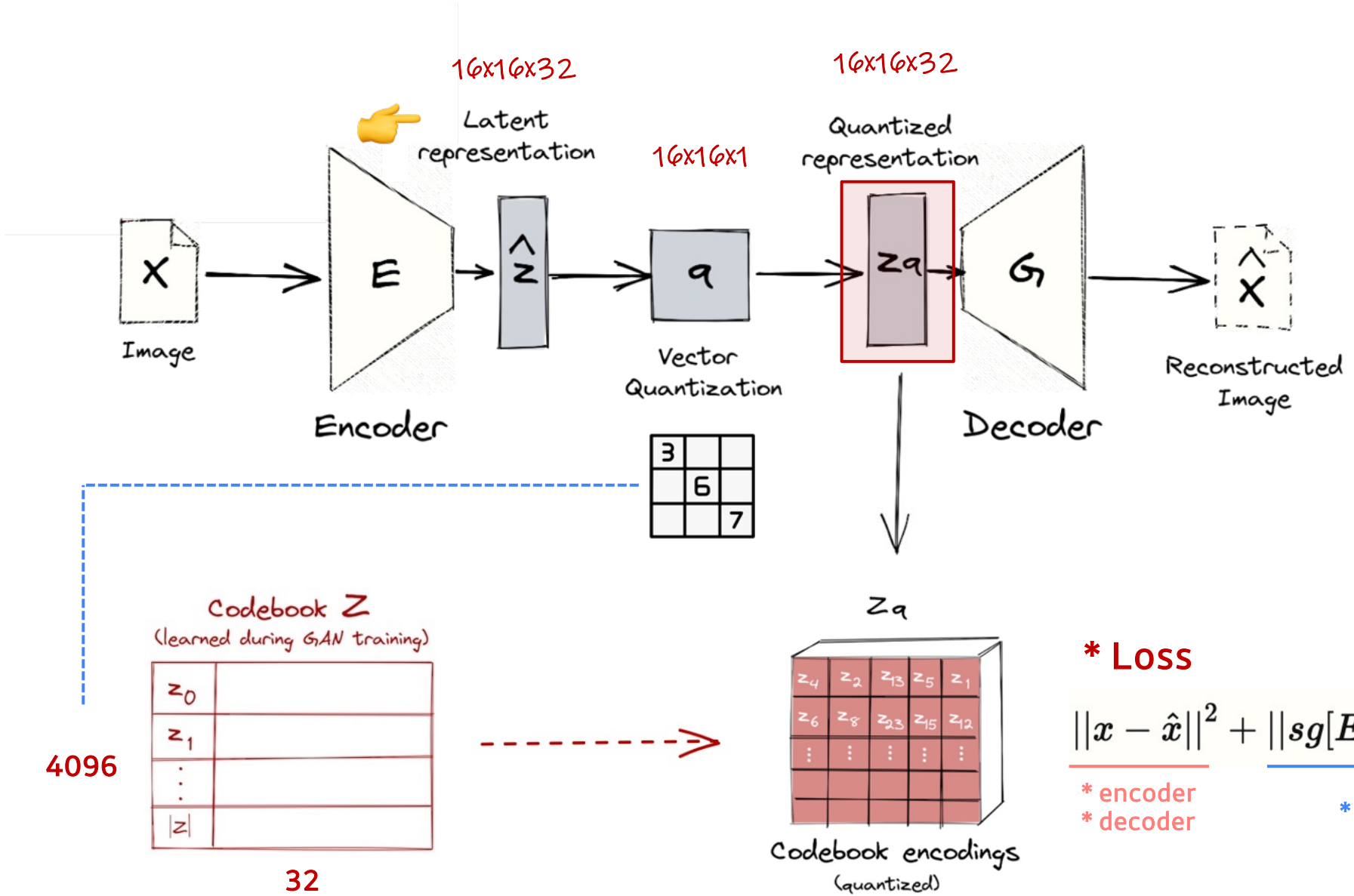



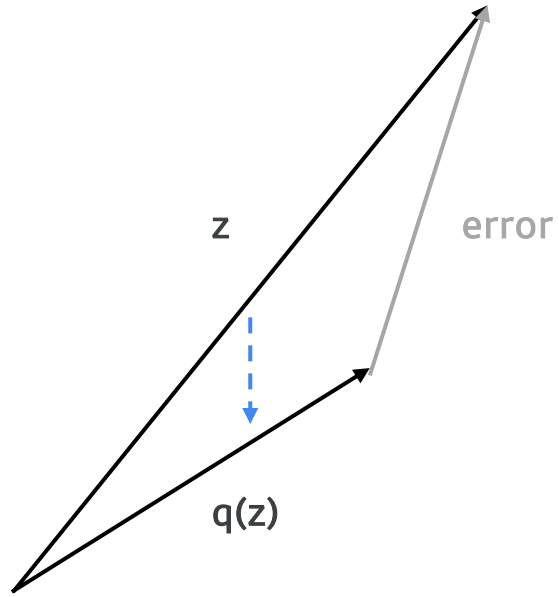
# Method | Preliminaries: VQVAE



NAVER AI LAB




## * Loss

$$||x - \hat{x}||^2 + ||sg[E(x)] - z_q||_2^2 + ||sg[z_q] - E(x)||_2^2$$

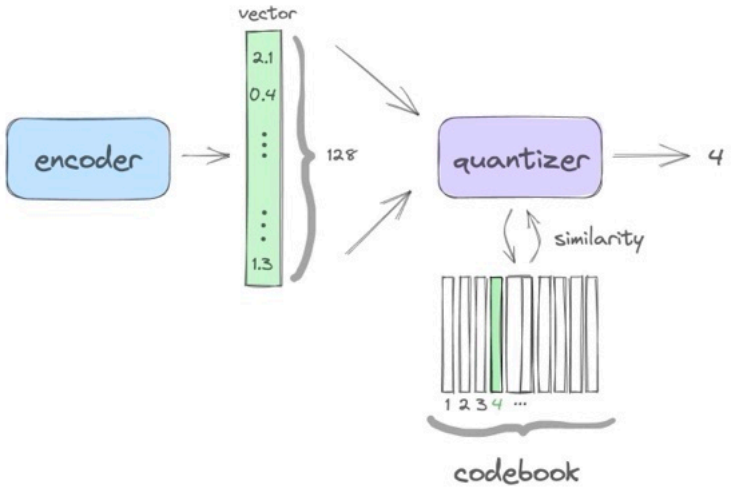
* encoder
* decoder

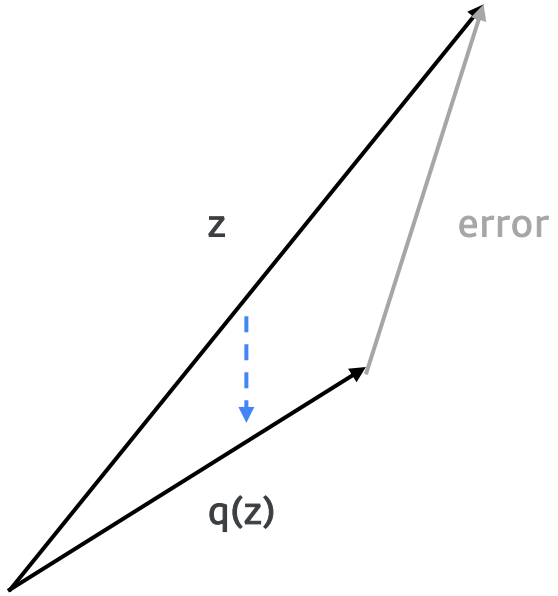
* codebook

* encoder
( reg )

NAVER AI LAB

기존

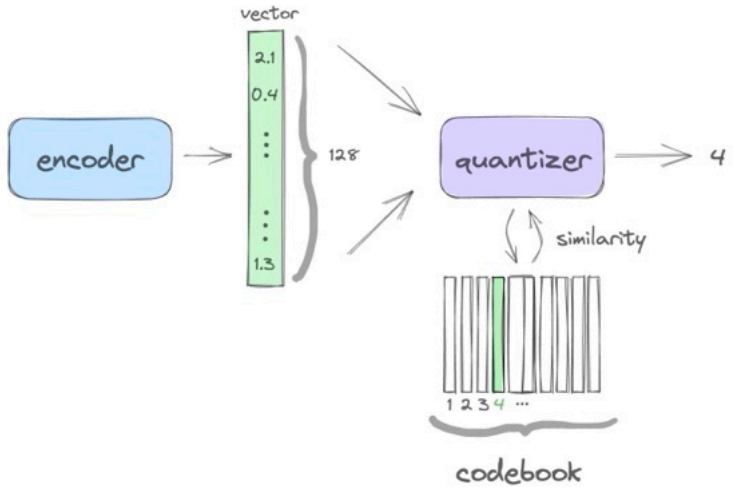


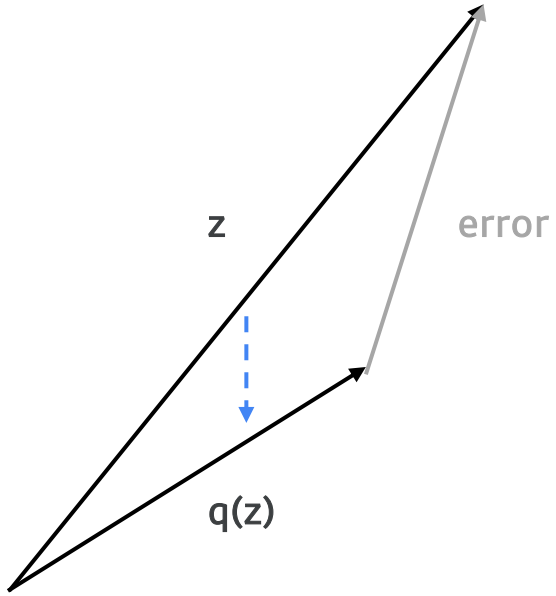



# Method | Preliminaries: Residual Quantization

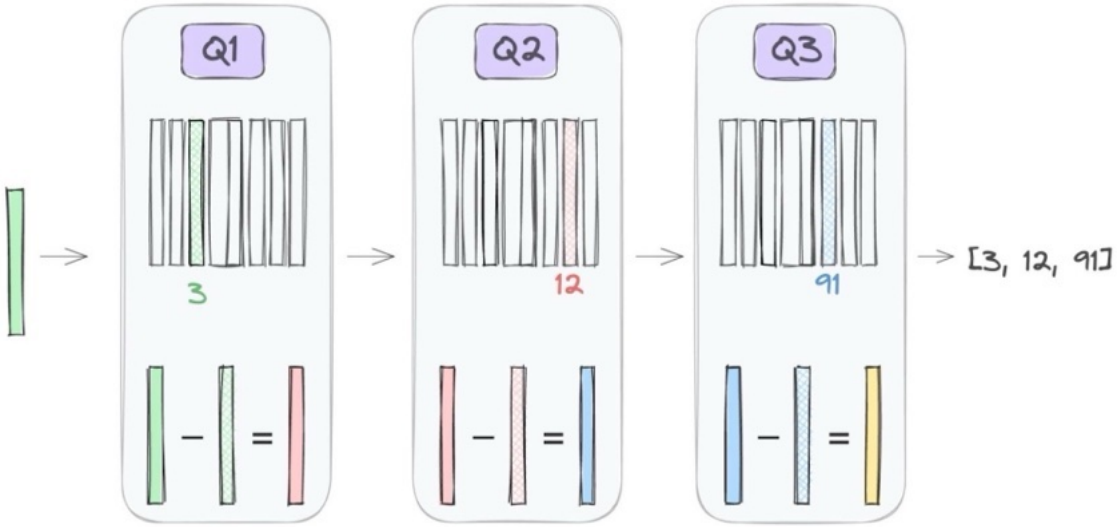
NAVER AI LAB

기존



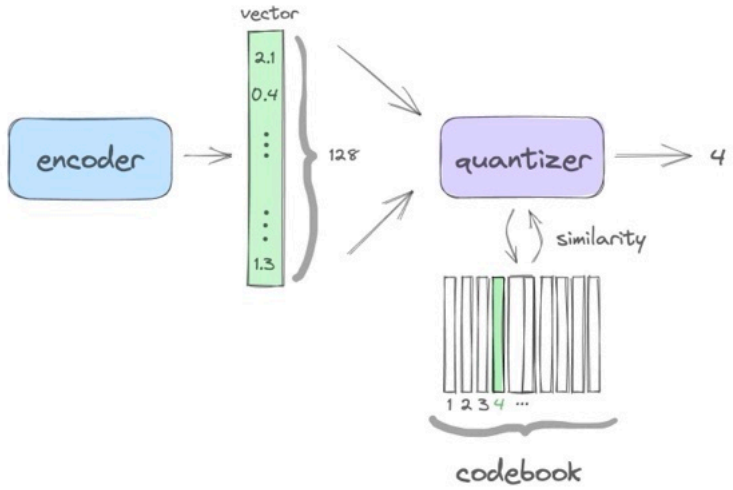



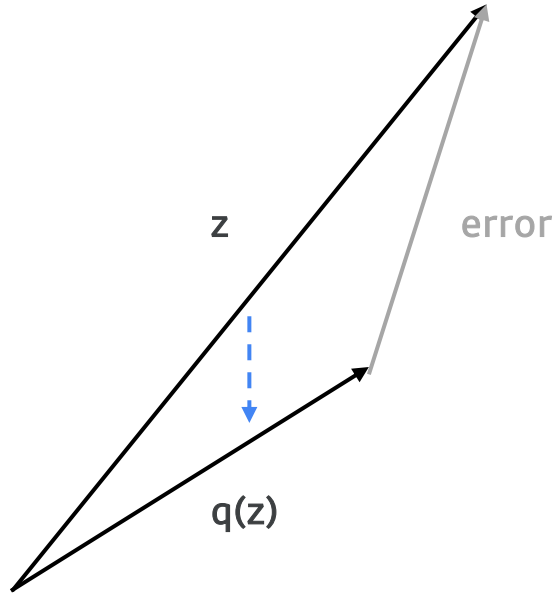

Residual



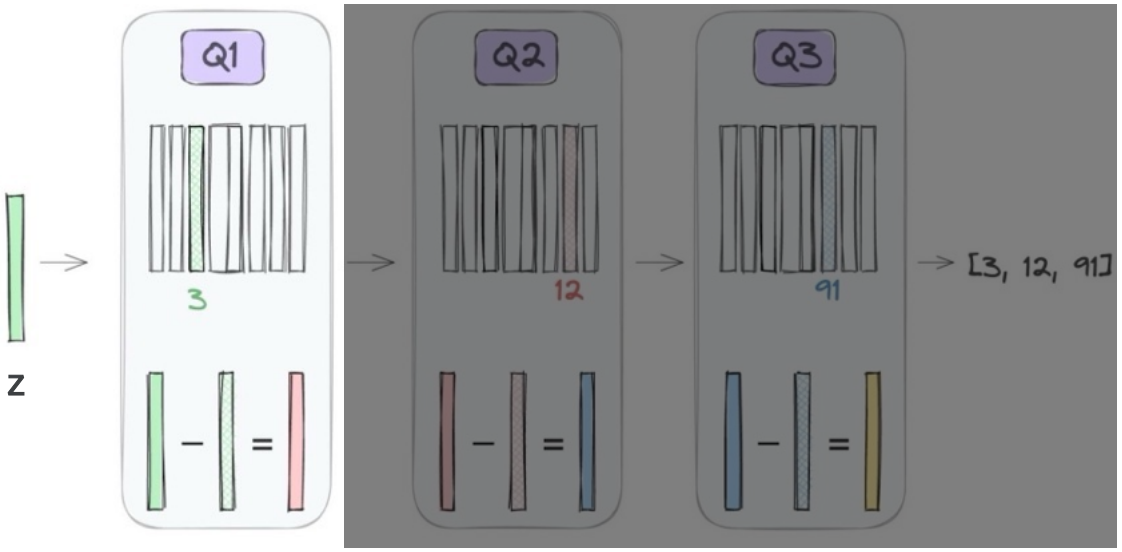
NAVER AI LAB

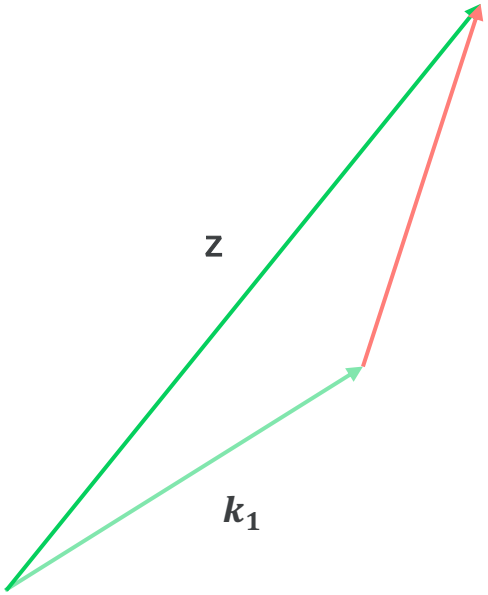
기존



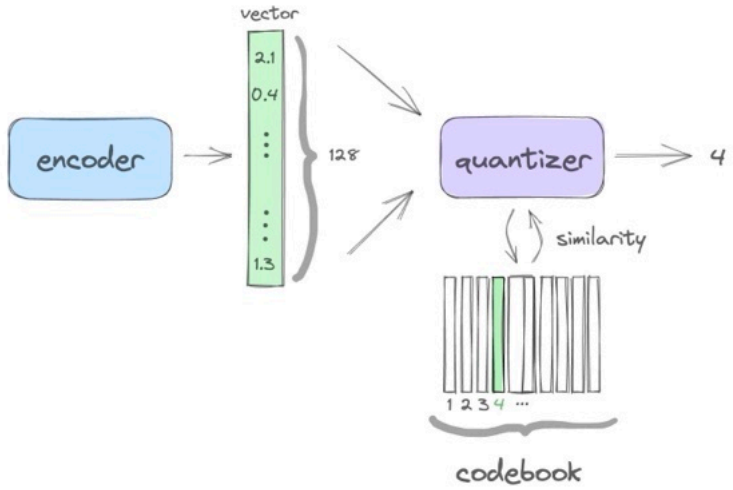

Residual

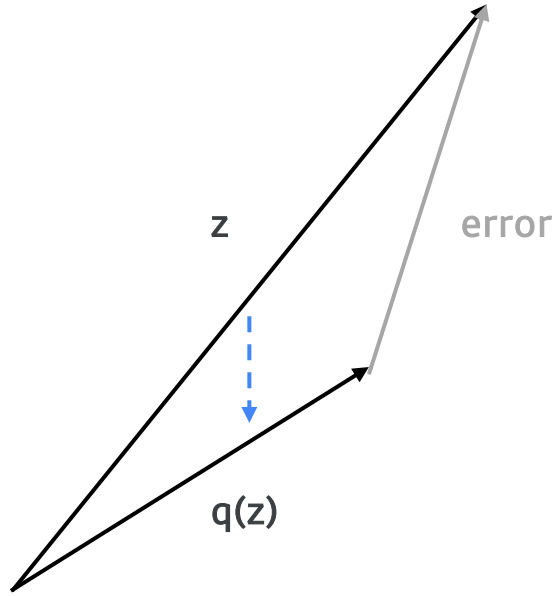


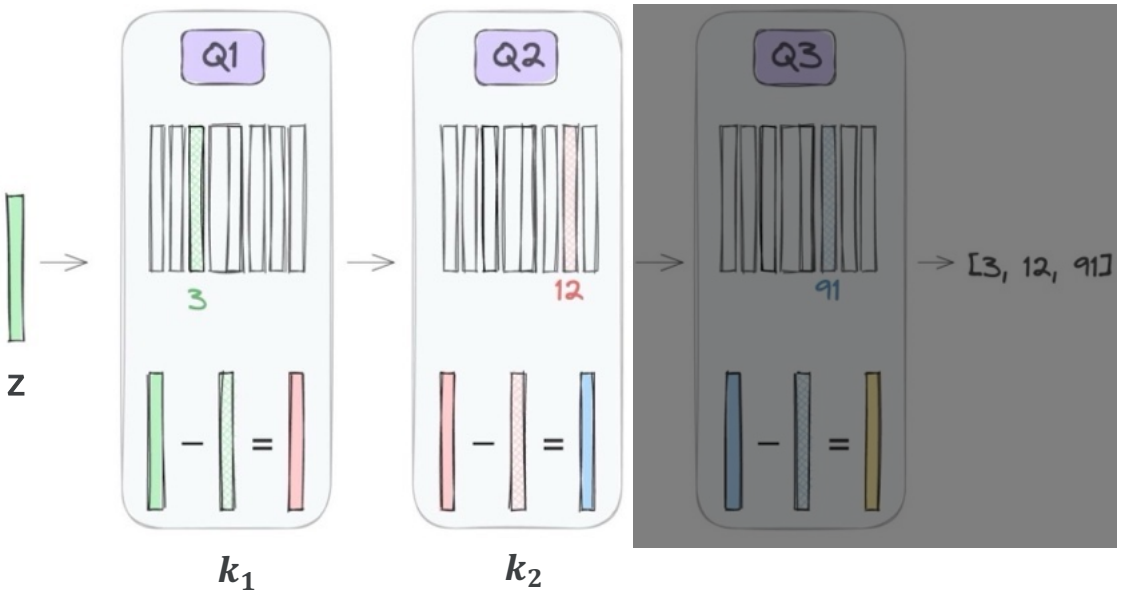
NAVER AI LAB

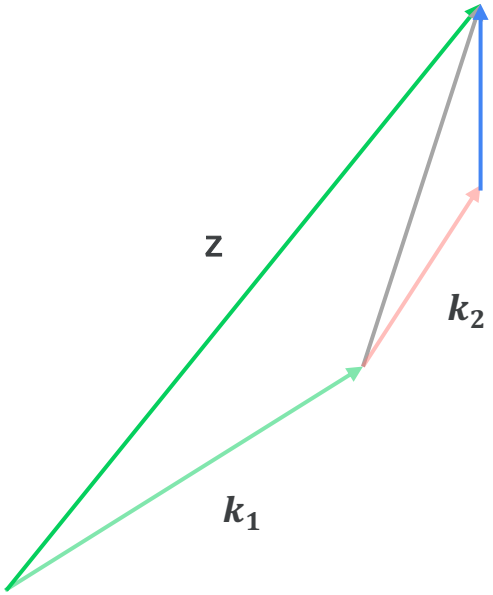
기존



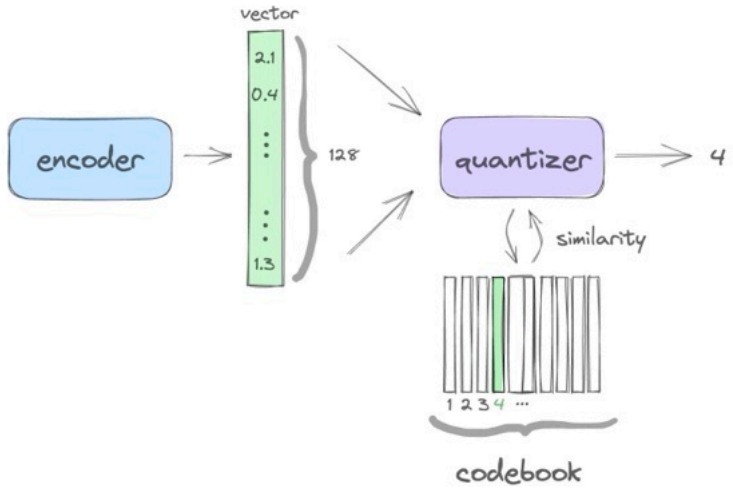

Residual

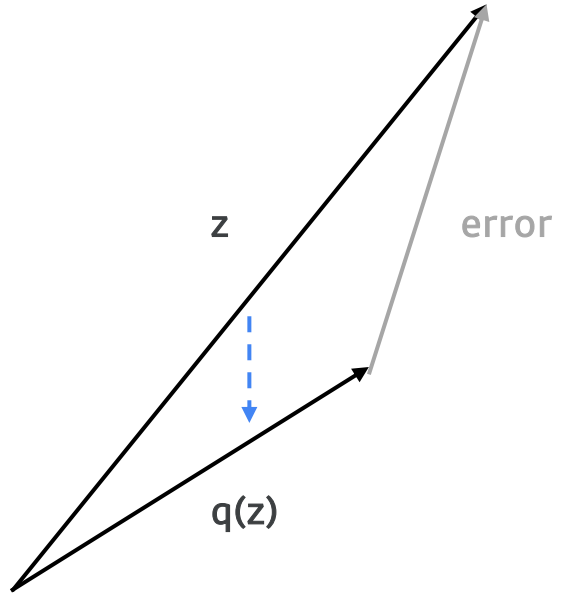


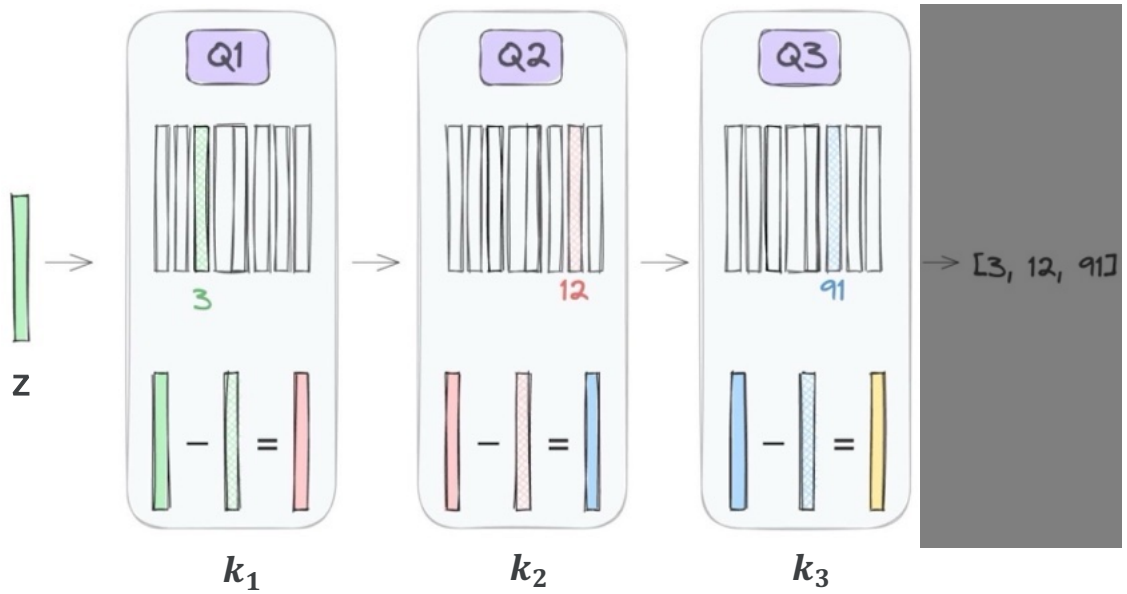
NAVER AI LAB

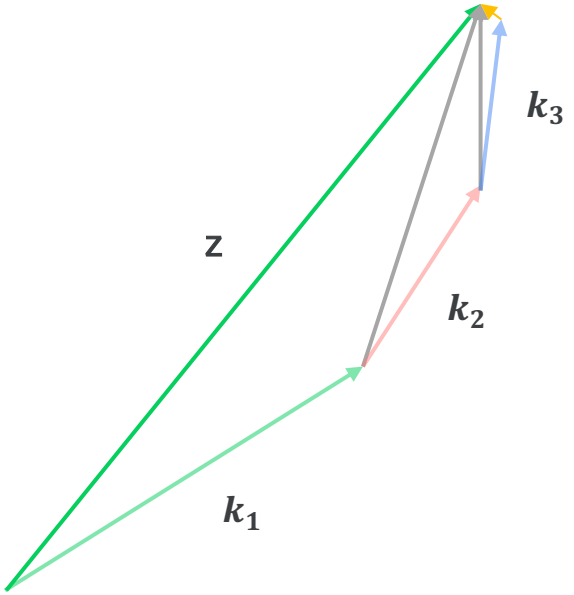
기존



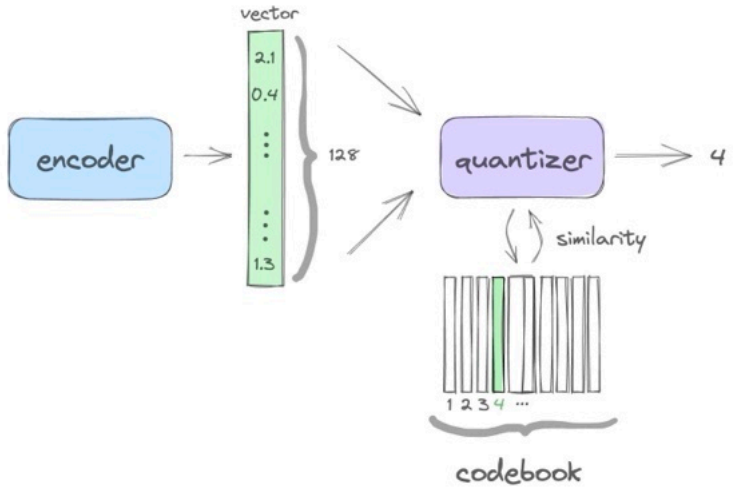

Residual

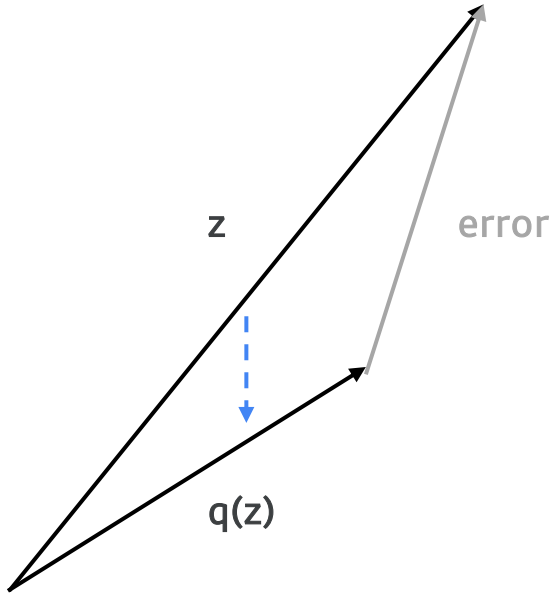


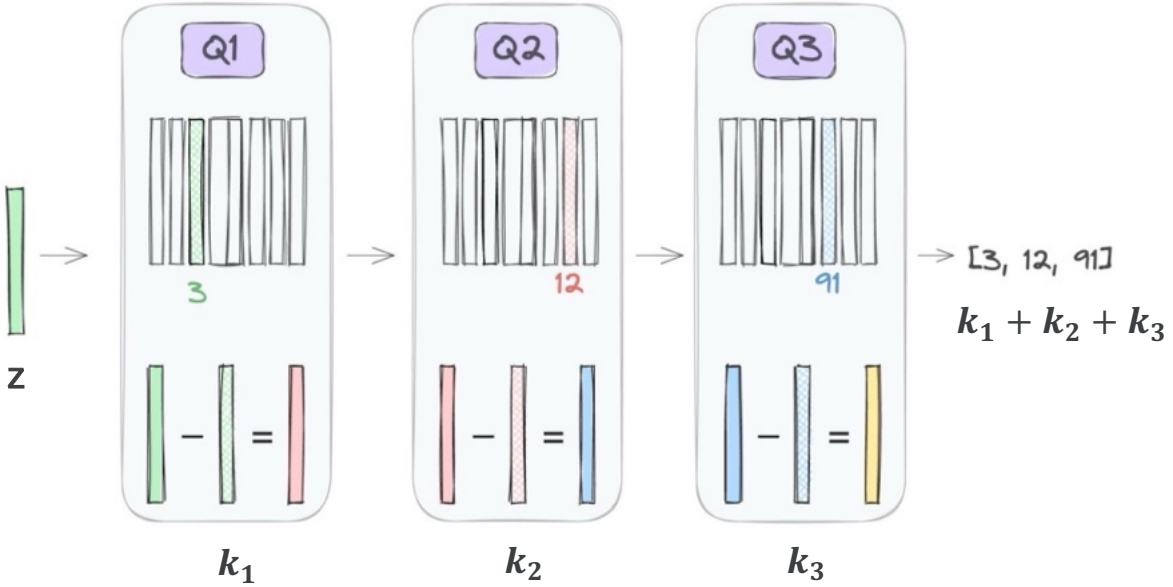
NAVER AI LAB

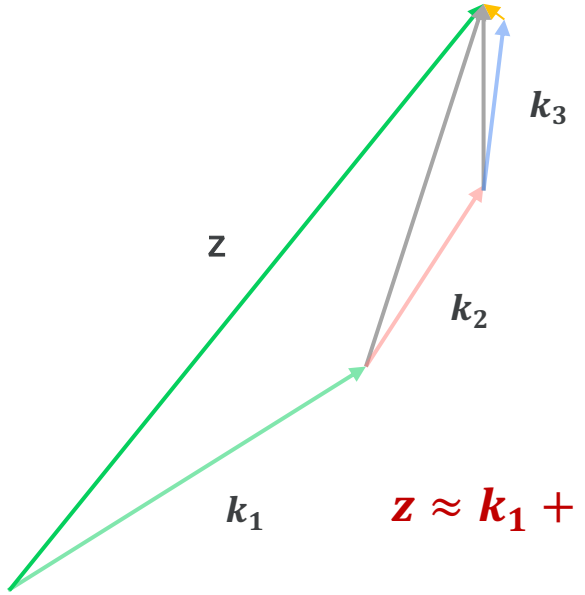
기존




Residual

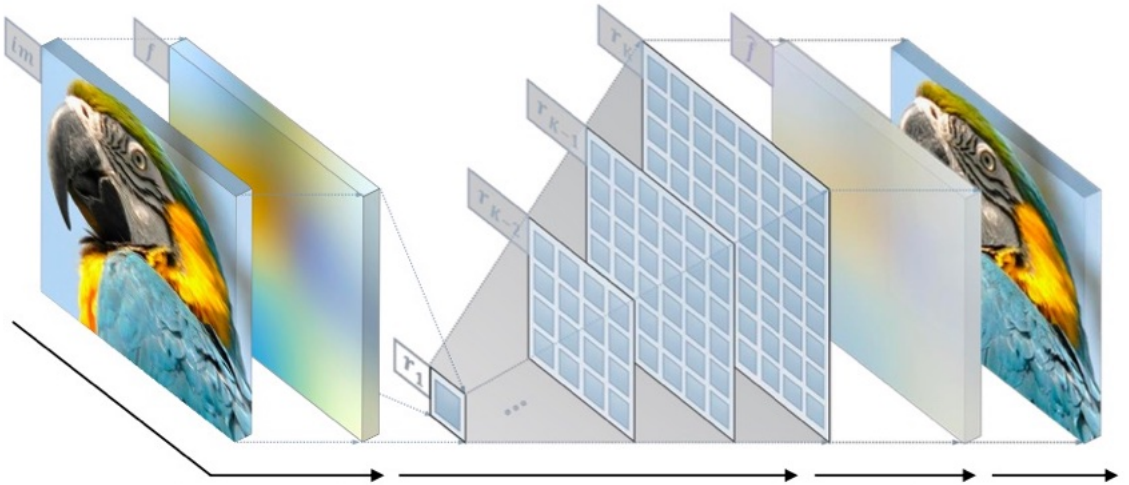


NAVER AI LAB

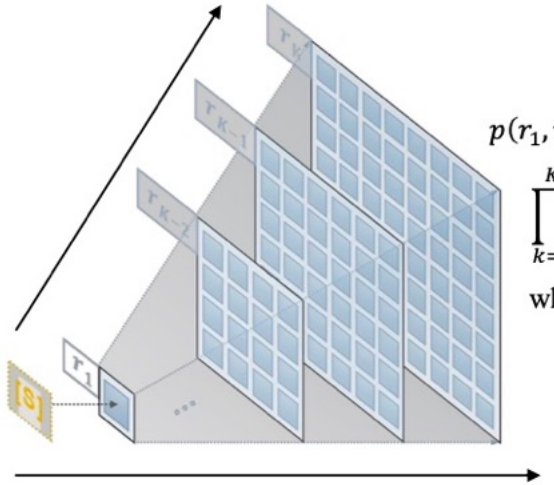
**Stage 1: Training multi-scale VQVAE on images**

( to provide the ground truth for Stage 2's training )




**Stage 2: Training VAR transformer on tokens**

([S] means a start token w/ or w/o condition information)



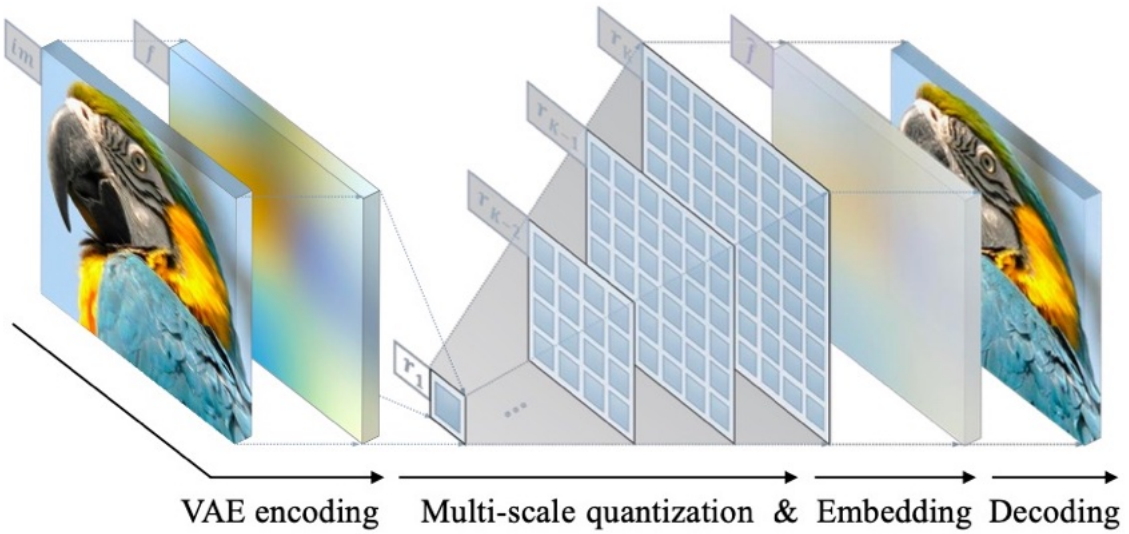

$$p(r_1, r_2, \dots r_K) = \prod_{k=1}^{K} p(r_k \mid r_1, \dots r_{k-1}),$$

where $p(r_0) = \delta([S])$.

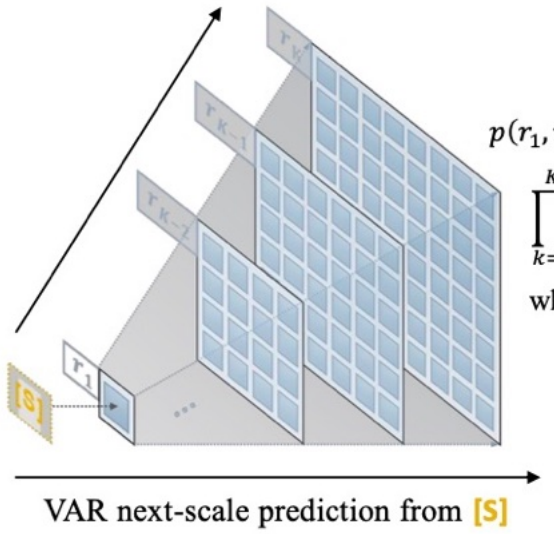
# Method | VAR Overview

NAVER AI LAB

**Stage 1: Training multi-scale VQVAE on images**
( to provide the ground truth for Stage 2's training )

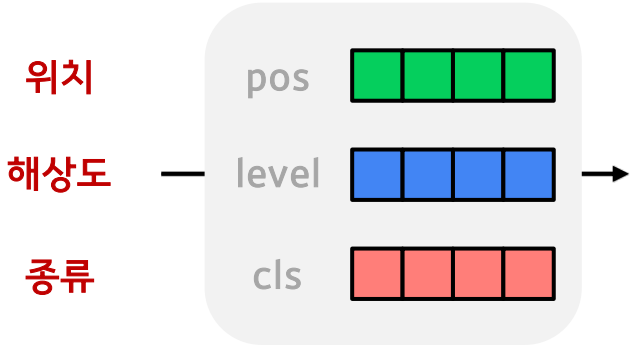



**Stage 2: Training VAR transformer on tokens**
([S] means a start token w/ or w/o condition information)




$$p(r_1, r_2, \dots r_K) = \prod_{k=1}^{K} p(r_k \mid r_1, \dots r_{k-1}),$$

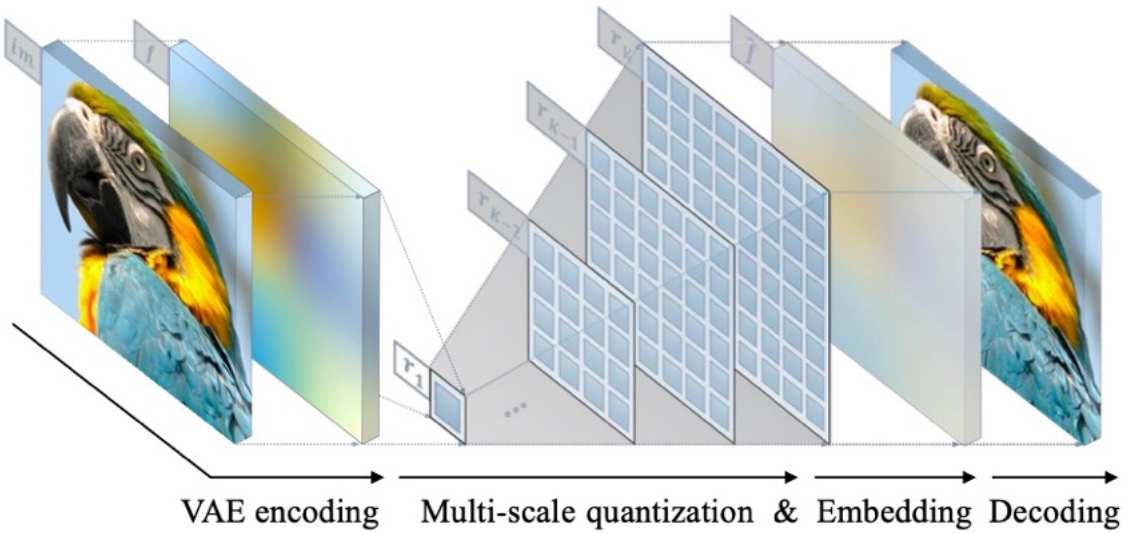
where $p(r_0) = \delta([S])$.



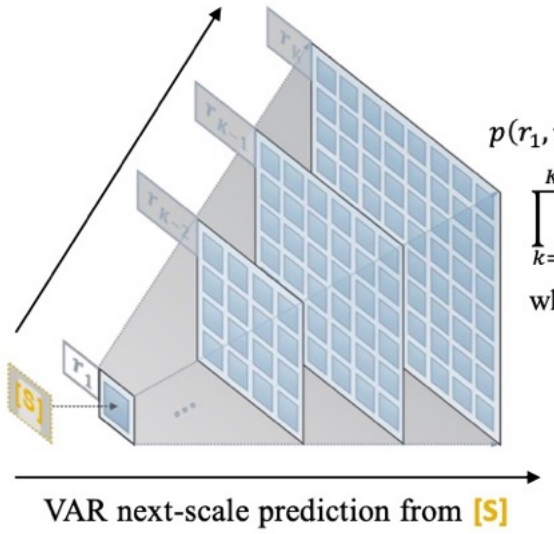
# Method | VAR Overview

NAVER AI LAB

**Stage 1: Training multi-scale VQVAE on images**
( to provide the ground truth for Stage 2's training )



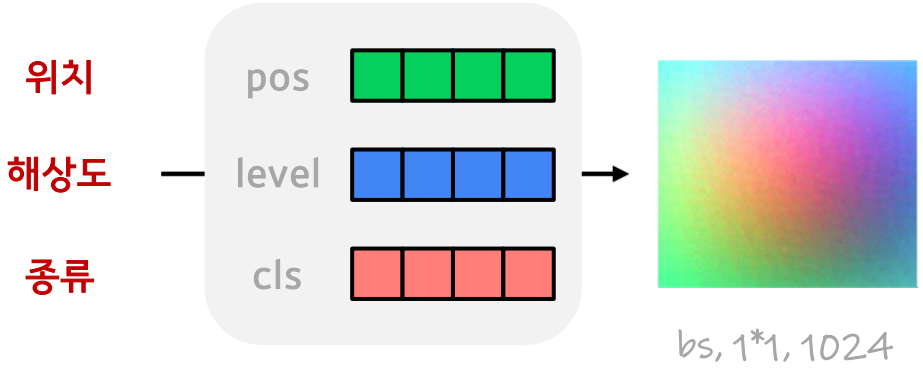

**Stage 2: Training VAR transformer on tokens**
([S] means a start token w/ or w/o condition information)



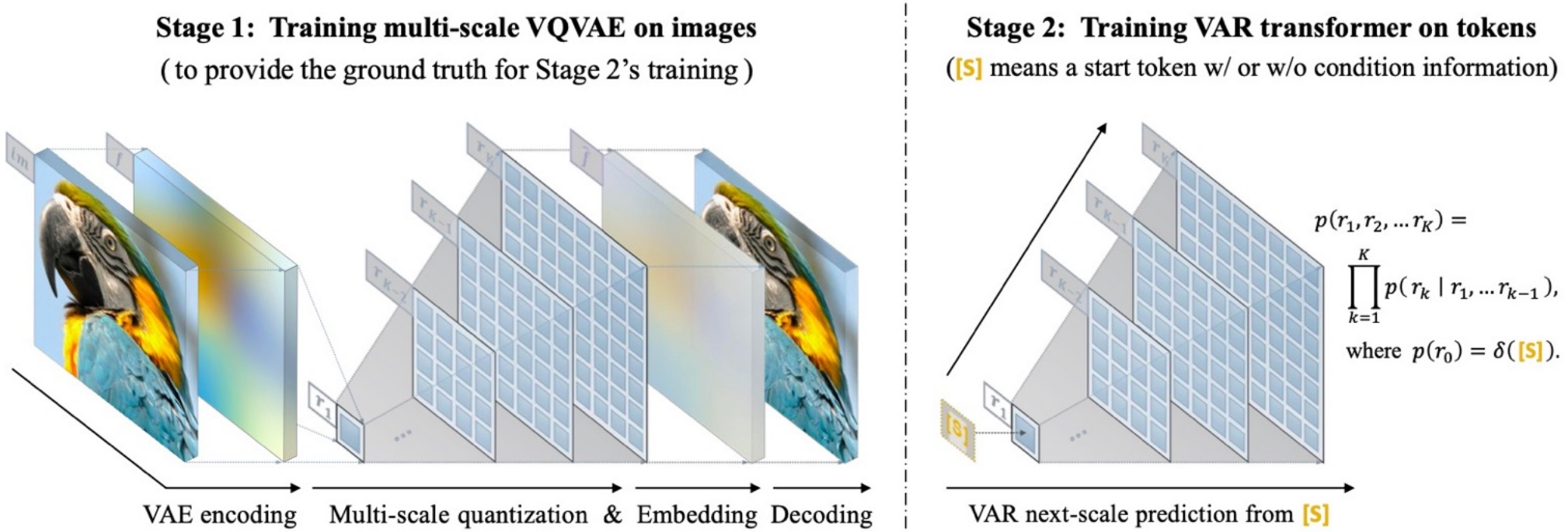

$$p(r_1, r_2, \dots r_K) = \prod_{k=1}^{K} p(r_k \mid r_1, \dots r_{k-1}),$$

where $p(r_0) = \delta([S])$.

patch: [1,2,3, ... , 16]

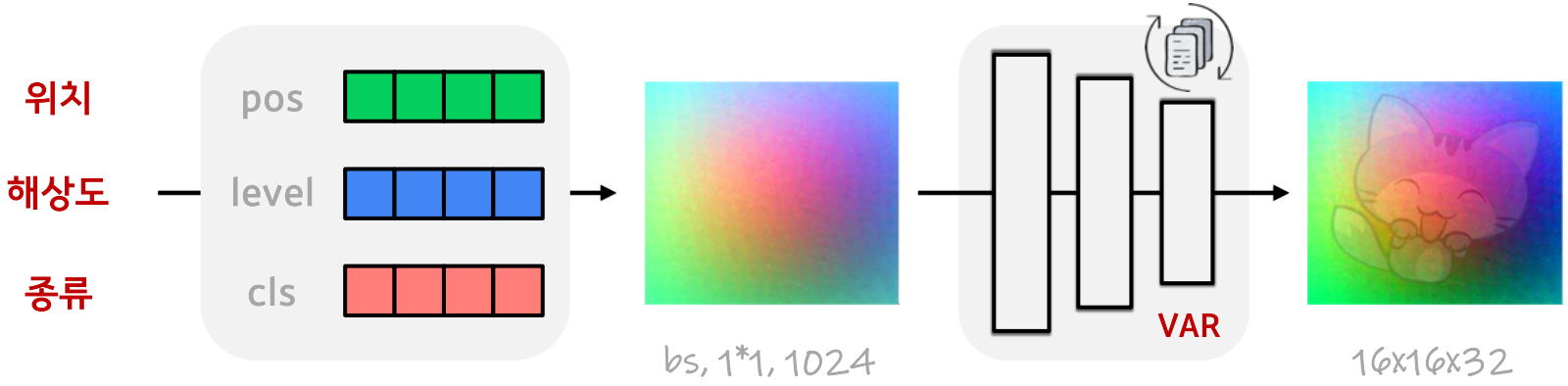


# Method | VAR Overview

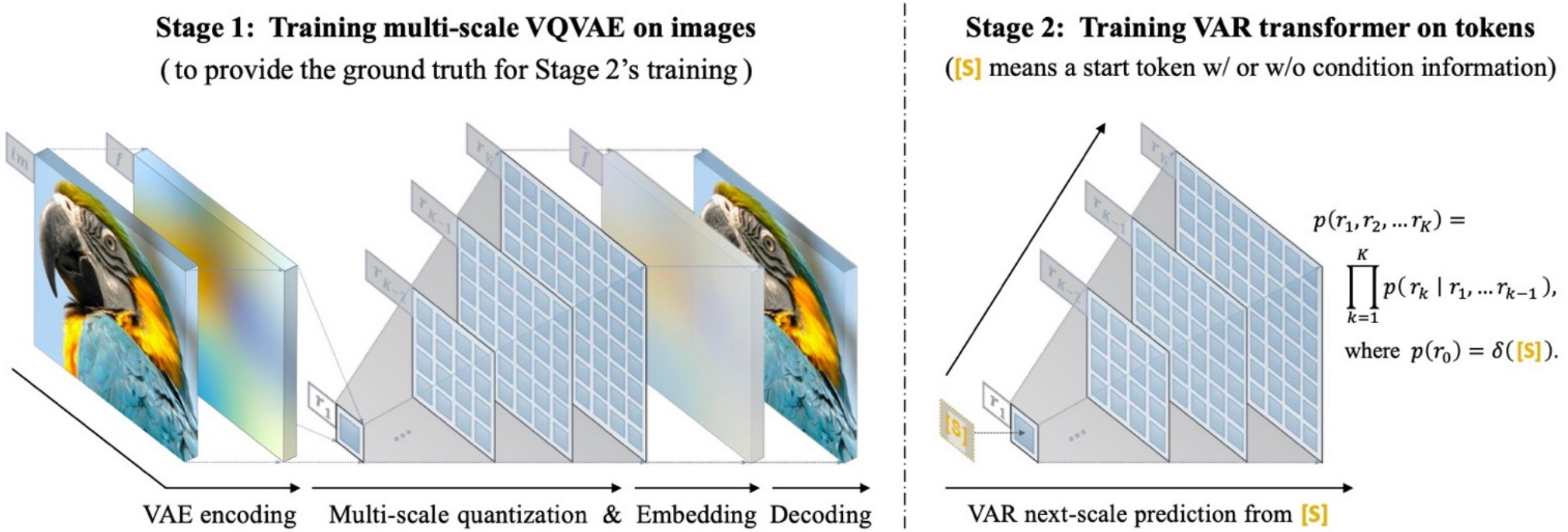



patch: [1,2,3, ... , 16]

1x1 -> 2x2 -> ... -> 16x16

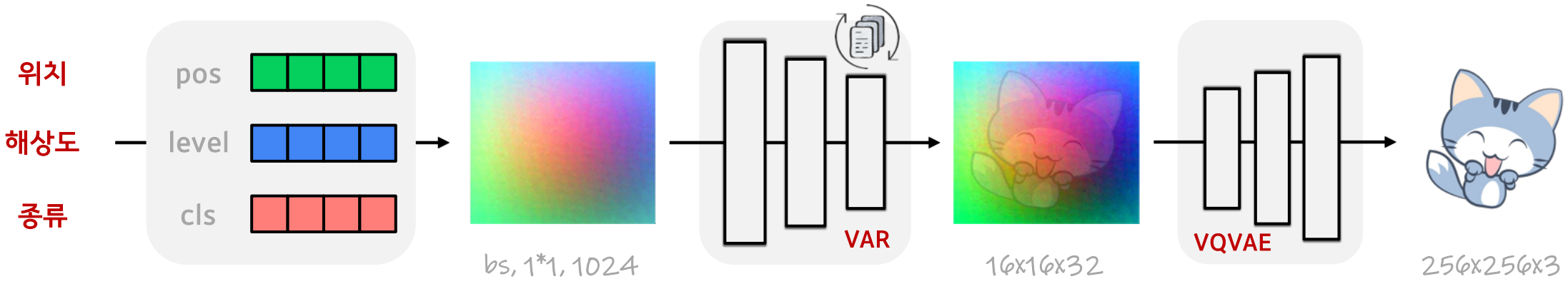


# Method | VAR Overview




patch: [1,2,3, ... , 16]

1x1 -> 2x2 -> ... -> 16x16



# Method | VQVAE Training

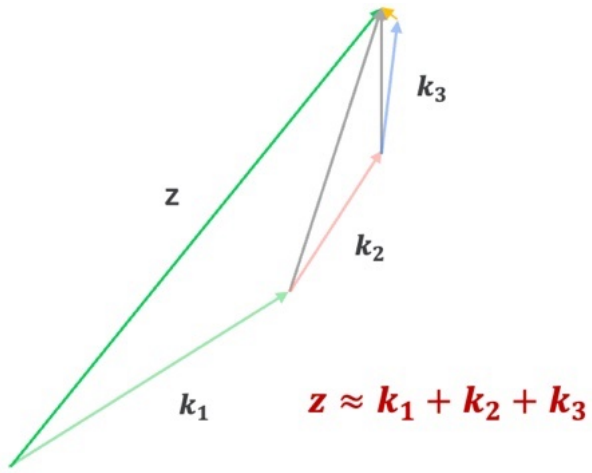
NAVER AI LAB

**Algorithm 1:** Multi-scale VQVAE Encoding

**Inputs:** raw image $im$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$f = \mathcal{E}(im)$, $R = []$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \mathcal{Q}(\text{interpolate}(f, h_k, w_k))$;
  $R = \text{queue\_push}(R, r_k)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $f = f - \phi_k(z_k)$;
**Return:** multi-scale tokens $R$;

**Algorithm 2:** Multi-scale VQVAE Reconstruction

**Inputs:** multi-scale token maps $R$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$\hat{f} = 0$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \text{queue\_pop}(R)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $\hat{f} = \hat{f} + \phi_k(z_k)$;
$\hat{im} = \mathcal{D}(\hat{f})$;
**Return:** reconstructed image $\hat{im}$;



# Method | VQVAE Training

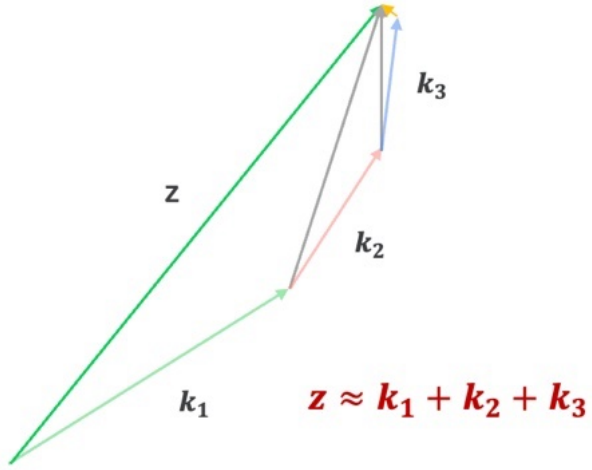
NAVER AI LAB

**Algorithm 1:** Multi-scale VQVAE Encoding

**Inputs:** raw image $im$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$f = \mathcal{E}(im)$, $R = []$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \mathcal{Q}(\text{interpolate}(f, h_k, w_k))$;
  $R = \text{queue\_push}(R, r_k)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $f = f - \phi_k(z_k)$;
**Return:** multi-scale tokens $R$;

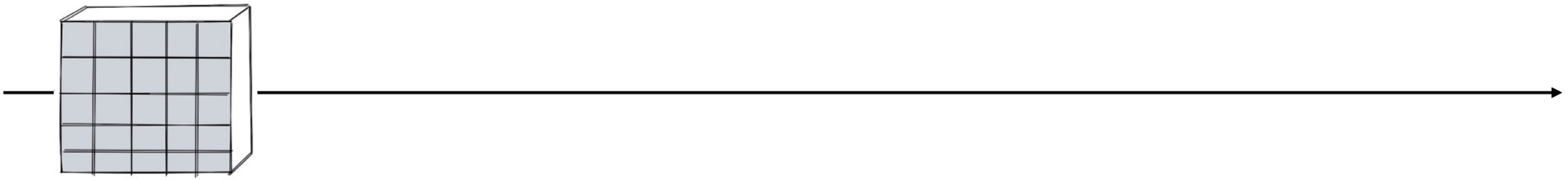
**Algorithm 2:** Multi-scale VQVAE Reconstruction

**Inputs:** multi-scale token maps $R$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$\hat{f} = 0$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \text{queue\_pop}(R)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $\hat{f} = \hat{f} + \phi_k(z_k)$;
$\hat{im} = \mathcal{D}(\hat{f})$;
**Return:** reconstructed image $\hat{im}$;




patch: [1,2,3, ... , 16]

for k in patch_K:



$f$

16x16x32

# Method | VQVAE Training

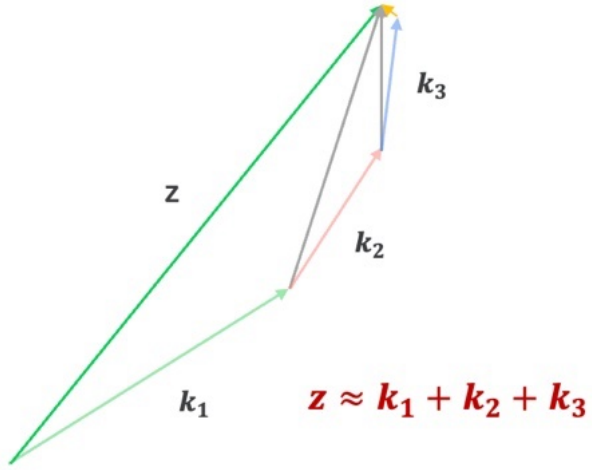
NAVER AI LAB

**Algorithm 1:** Multi-scale VQVAE Encoding

**Inputs:** raw image $im$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$f = \mathcal{E}(im)$, $R = []$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \mathcal{Q}(\text{interpolate}(f, h_k, w_k))$;
  $R = \text{queue\_push}(R, r_k)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $f = f - \phi_k(z_k)$;
**Return:** multi-scale tokens $R$;

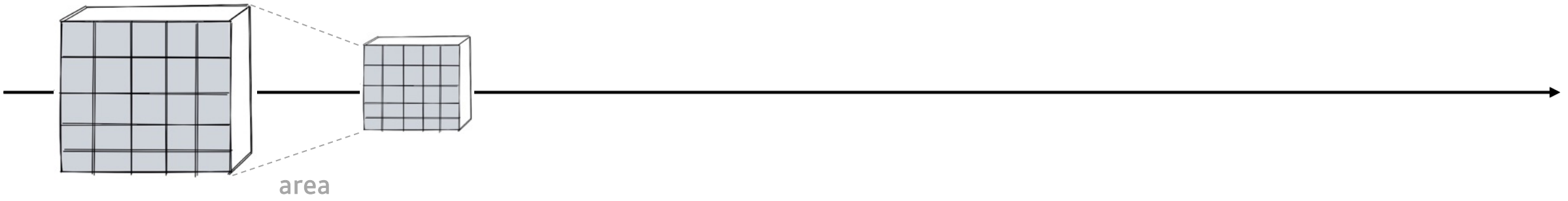
**Algorithm 2:** Multi-scale VQVAE Reconstruction

**Inputs:** multi-scale token maps $R$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$\hat{f} = 0$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \text{queue\_pop}(R)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $\hat{f} = \hat{f} + \phi_k(z_k)$;
$\hat{im} = \mathcal{D}(\hat{f})$;
**Return:** reconstructed image $\hat{im}$;




patch: [1,2,3, ... , 16]

**for k in patch_K:**



# Method | VQVAE Training

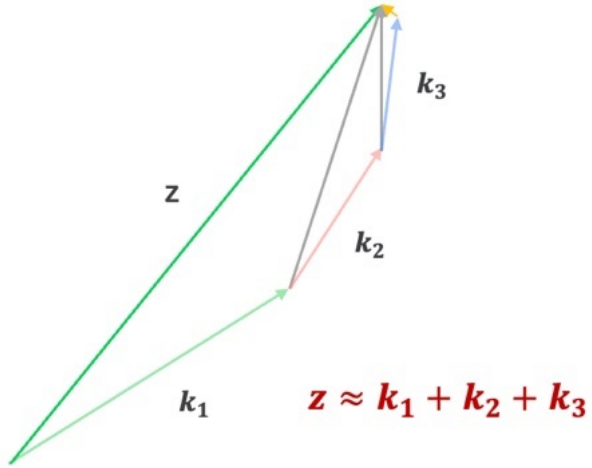
NAVER AI LAB

**Algorithm 1:** Multi-scale VQVAE Encoding

**Inputs:** raw image $im$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$f = \mathcal{E}(im)$, $R = []$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \mathcal{Q}(\text{interpolate}(f, h_k, w_k))$;
  $R = \text{queue\_push}(R, r_k)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $f = f - \phi_k(z_k)$;
**Return:** multi-scale tokens $R$;

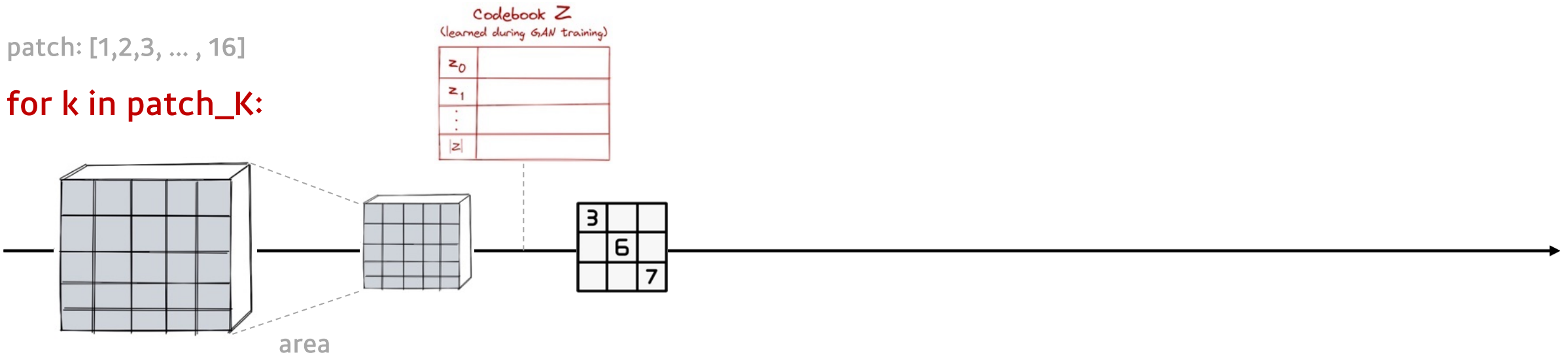
**Algorithm 2:** Multi-scale VQVAE Reconstruction

**Inputs:** multi-scale token maps $R$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$\hat{f} = 0$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \text{queue\_pop}(R)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $\hat{f} = \hat{f} + \phi_k(z_k)$;
$\hat{im} = \mathcal{D}(\hat{f})$;
**Return:** reconstructed image $\hat{im}$;




patch: [1,2,3, … , 16]

**for k in patch_K:**



# Method | VQVAE Training

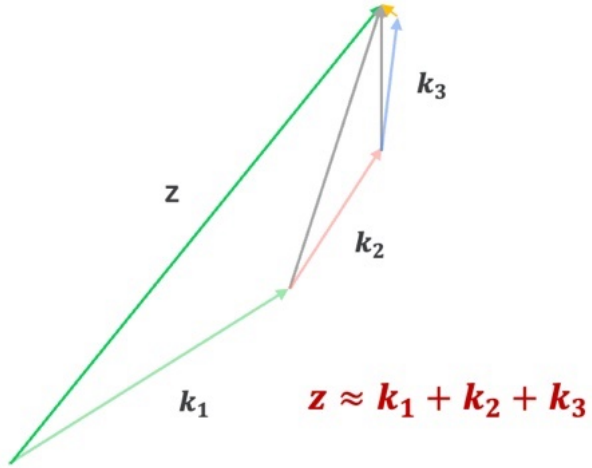
NAVER AI LAB

**Algorithm 1:** Multi-scale VQVAE Encoding

**Inputs:** raw image $im$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$f = \mathcal{E}(im)$, $R = []$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \mathcal{Q}(\text{interpolate}(f, h_k, w_k))$;
  $R = \text{queue\_push}(R, r_k)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $f = f - \phi_k(z_k)$;
**Return:** multi-scale tokens $R$;

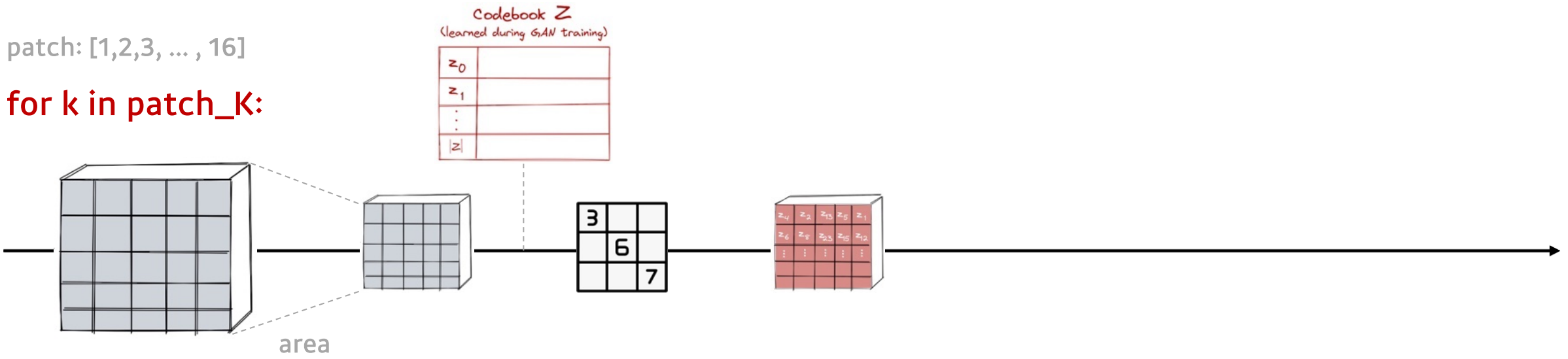
**Algorithm 2:** Multi-scale VQVAE Reconstruction

**Inputs:** multi-scale token maps $R$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$\hat{f} = 0$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \text{queue\_pop}(R)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $\hat{f} = \hat{f} + \phi_k(z_k)$;
$\hat{im} = \mathcal{D}(\hat{f})$;
**Return:** reconstructed image $\hat{im}$;




patch: [1,2,3, ... , 16]

**for k in patch_K:**




| $f$ | $f_k$ | $r_k$ | $z_k$ |
|---|---|---|---|
| 16x16x32 | pxpx32 | pxpx1 | pxpx32 |

# Method | VQVAE Training

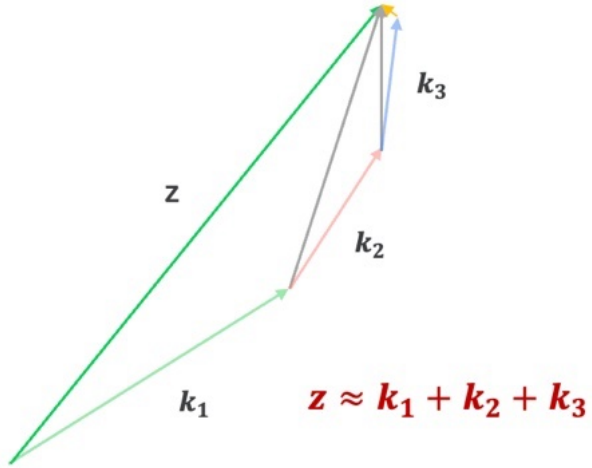
NAVER AI LAB

**Algorithm 1:** Multi-scale VQVAE Encoding

**Inputs:** raw image $im$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$f = \mathcal{E}(im)$, $R = []$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \mathcal{Q}(\text{interpolate}(f, h_k, w_k))$;
  $R = \text{queue\_push}(R, r_k)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $f = f - \phi_k(z_k)$;
**Return:** multi-scale tokens $R$;

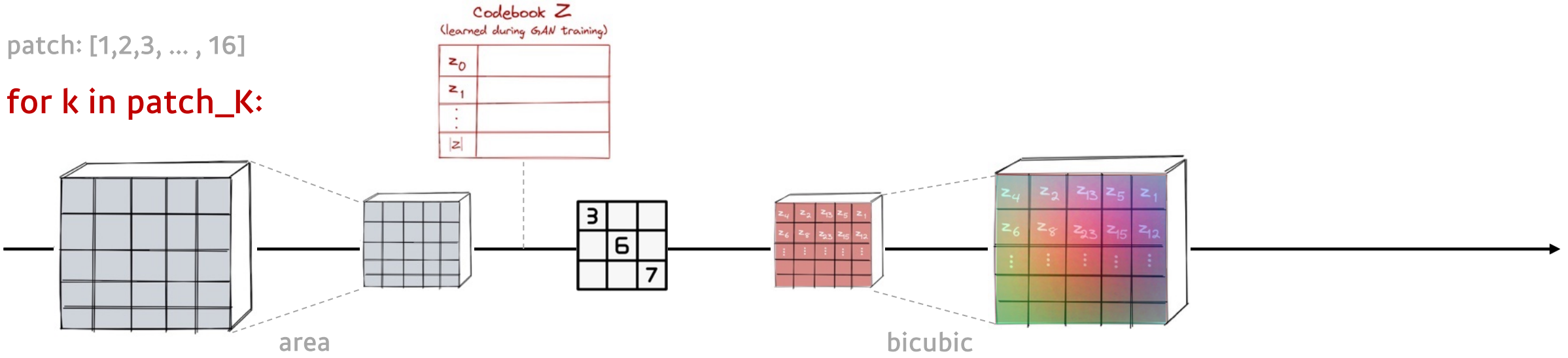
**Algorithm 2:** Multi-scale VQVAE Reconstruction

**Inputs:** multi-scale token maps $R$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$\hat{f} = 0$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \text{queue\_pop}(R)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $\hat{f} = \hat{f} + \phi_k(z_k)$;
$\hat{im} = \mathcal{D}(\hat{f})$;
**Return:** reconstructed image $\hat{im}$;




patch: [1,2,3, … , 16]

for k in patch_K:




| $f$ | $f_k$ | $r_k$ | $z_k$ | $\phi_k(z_k)$ |
|---|---|---|---|---|
| 16x16x32 | pxpx32 | pxpx1 | pxpx32 | 16x16x32 |

# Method | VQVAE Training

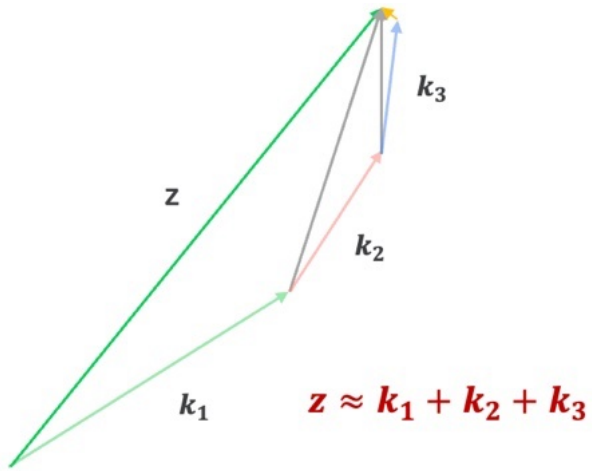
NAVER AI LAB

**Algorithm 1:** Multi-scale VQVAE Encoding

**Inputs:** raw image $im$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$f = \mathcal{E}(im)$, $R = []$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \mathcal{Q}(\text{interpolate}(f, h_k, w_k))$;
  $R = \text{queue\_push}(R, r_k)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $f = f - \phi_k(z_k)$;
**Return:** multi-scale tokens $R$;

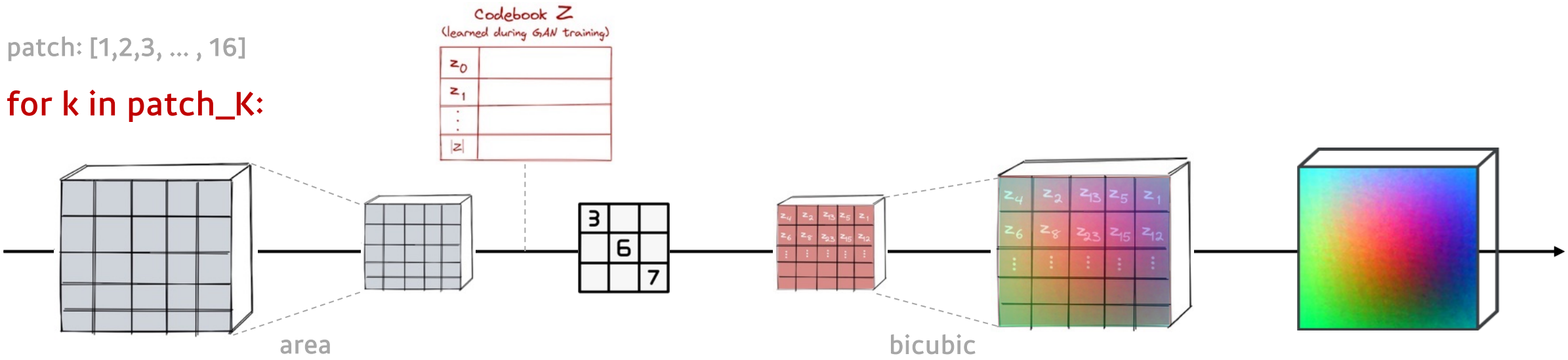
**Algorithm 2:** Multi-scale VQVAE Reconstruction

**Inputs:** multi-scale token maps $R$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$\hat{f} = 0$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \text{queue\_pop}(R)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $\hat{f} = \hat{f} + \phi_k(z_k)$;
$\hat{im} = \mathcal{D}(\hat{f})$;
**Return:** reconstructed image $\hat{im}$;




patch: [1,2,3, ... , 16]

**for k in patch_K:**




| $f$ | $f_k$ | $r_k$ | $z_k$ | $\phi_k(z_k)$ | $f - \phi_k(z_k)$ |
|---|---|---|---|---|---|
| 16x16x32 | pxpx32 | pxpx1 | pxpx32 | 16x16x32 | 16x16x32 |

# Method | VQVAE Training

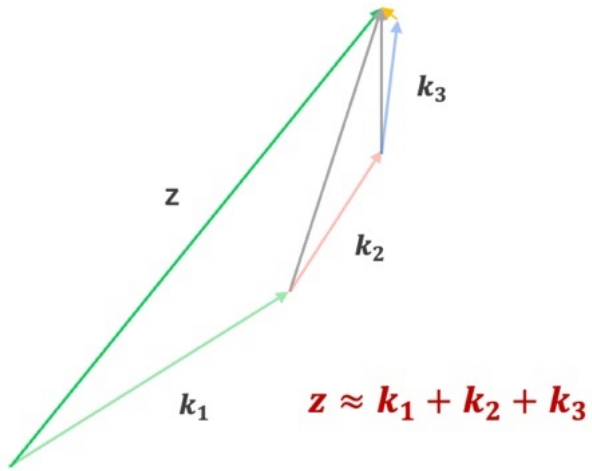
NAVER AI LAB

**Algorithm 1:** Multi-scale VQVAE Encoding

**Inputs:** raw image $im$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$f = \mathcal{E}(im)$, $R = []$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \mathcal{Q}(\text{interpolate}(f, h_k, w_k))$;
  $R = \text{queue\_push}(R, r_k)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $f = f - \phi_k(z_k)$;
**Return:** multi-scale tokens $R$;

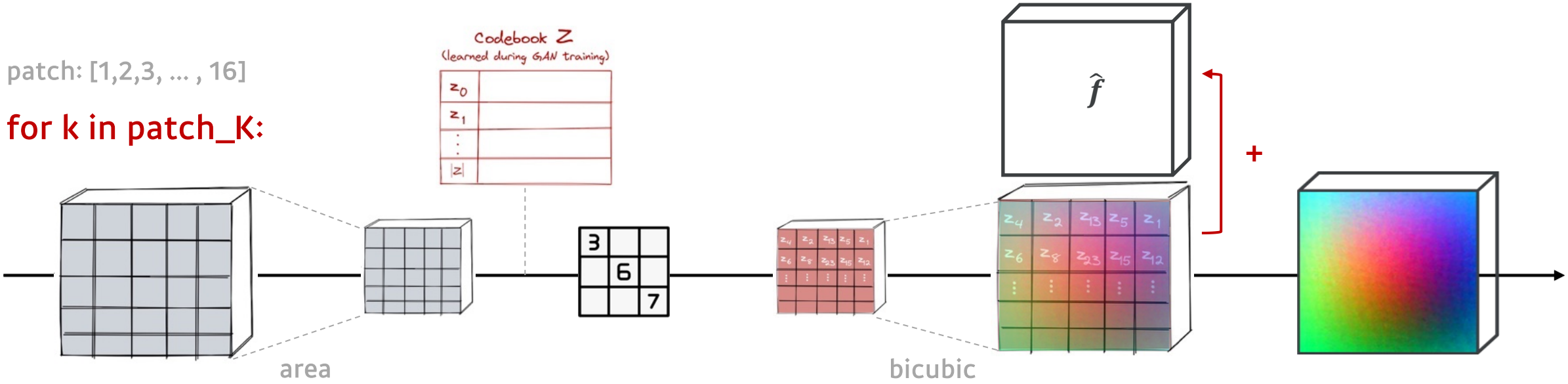
**Algorithm 2:** Multi-scale VQVAE Reconstruction

**Inputs:** multi-scale token maps $R$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$\hat{f} = 0$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \text{queue\_pop}(R)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $\hat{f} = \hat{f} + \phi_k(z_k)$;
$\hat{im} = \mathcal{D}(\hat{f})$;
**Return:** reconstructed image $\hat{im}$;




patch: [1,2,3, ... , 16]

**for k in patch_K:**




| $f$ | $f_k$ | $r_k$ | $z_k$ | $\phi_k(z_k)$ | $f - \phi_k(z_k)$ |
|---|---|---|---|---|---|
| 16x16x32 | pxpx32 | pxpx1 | pxpx32 | 16x16x32 | 16x16x32 |

# Method | VQVAE Training

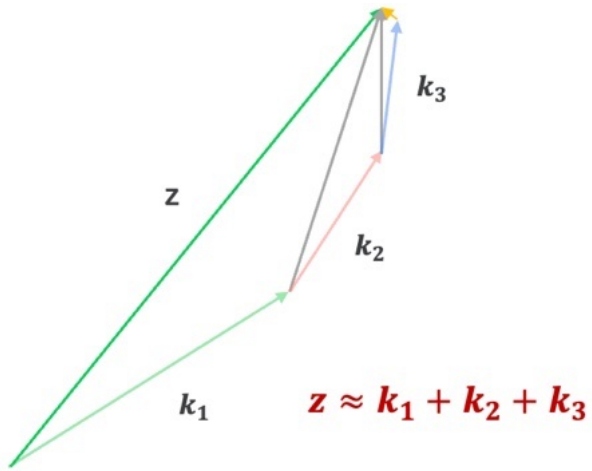
NAVER AI LAB

**Algorithm 1:** Multi-scale VQVAE Encoding

**Inputs:** raw image $im$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$f = \mathcal{E}(im)$, $R = []$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \mathcal{Q}(\text{interpolate}(f, h_k, w_k))$;
  $R = \text{queue\_push}(R, r_k)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $f = f - \phi_k(z_k)$;
**Return:** multi-scale tokens $R$;

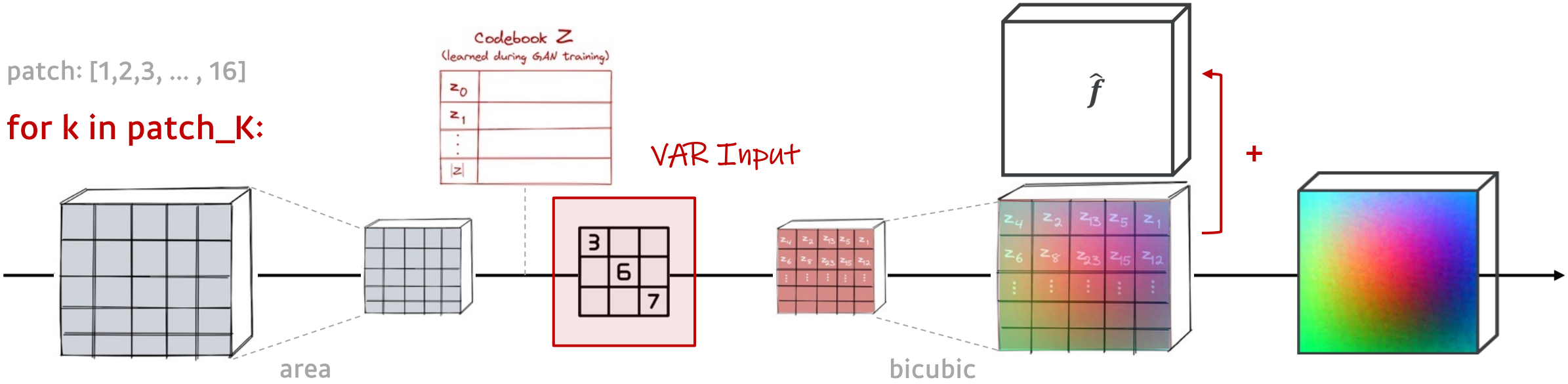
**Algorithm 2:** Multi-scale VQVAE Reconstruction

**Inputs:** multi-scale token maps $R$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$\hat{f} = 0$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \text{queue\_pop}(R)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $\hat{f} = \hat{f} + \phi_k(z_k)$;
$\hat{im} = \mathcal{D}(\hat{f})$;
**Return:** reconstructed image $\hat{im}$;




patch: [1,2,3, ... , 16]

**for k in patch_K:**




| $f$ | $f_k$ | $r_k$ | $z_k$ | $\phi_k(z_k)$ | $f - \phi_k(z_k)$ |
|---|---|---|---|---|---|
| 16x16x32 | pxpx32 | pxpx1 | pxpx32 | 16x16x32 | 16x16x32 |

# Method | VQVAE Training

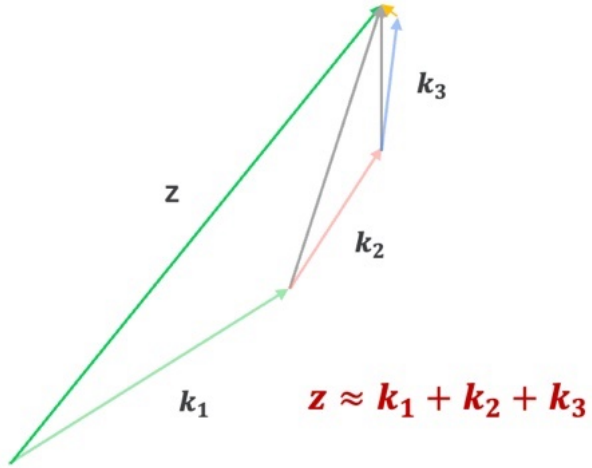
NAVER AI LAB

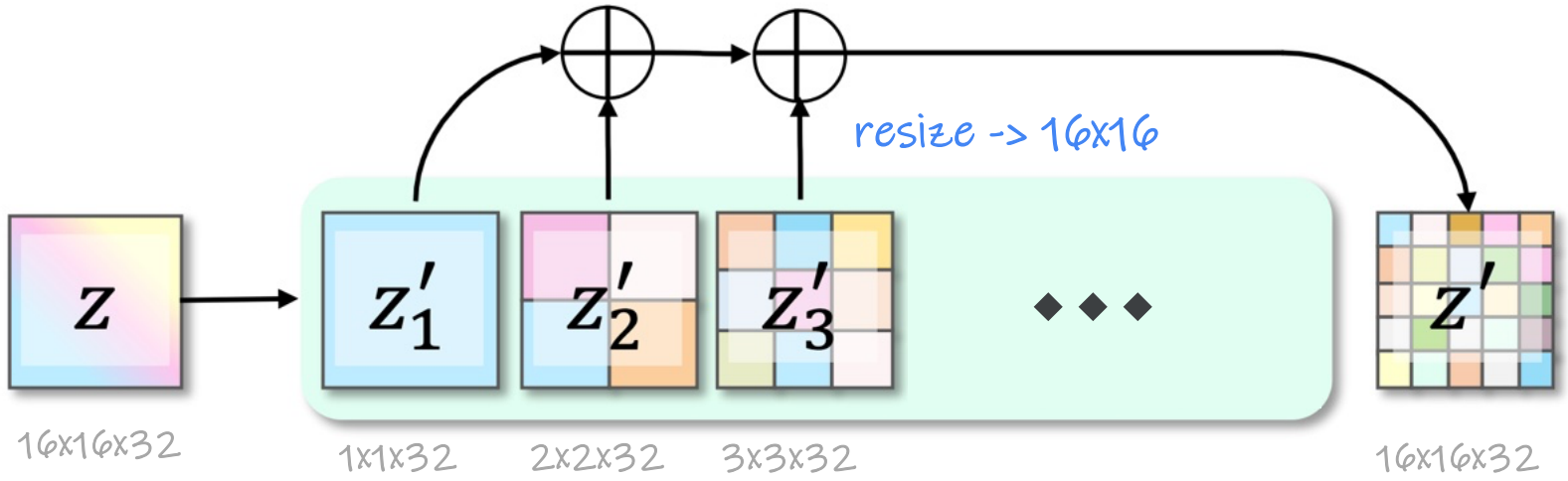
**Algorithm 1:** Multi-scale VQVAE Encoding

**Inputs:** raw image $im$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$f = \mathcal{E}(im)$, $R = []$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \mathcal{Q}(\text{interpolate}(f, h_k, w_k))$;
  $R = \text{queue\_push}(R, r_k)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $f = f - \phi_k(z_k)$;
**Return:** multi-scale tokens $R$;

**Algorithm 2:** Multi-scale VQVAE Reconstruction

**Inputs:** multi-scale token maps $R$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$\hat{f} = 0$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \text{queue\_pop}(R)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $\hat{f} = \hat{f} + \phi_k(z_k)$;
$\hat{im} = \mathcal{D}(\hat{f})$;
**Return:** reconstructed image $\hat{im}$;




$$z'_{k+1} = z - \sum_{i=1}^{k} z'_i$$

# Method | VQVAE Training

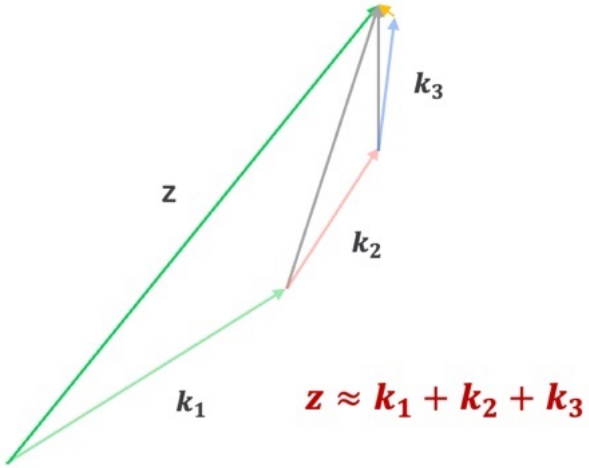
NAVER AI LAB

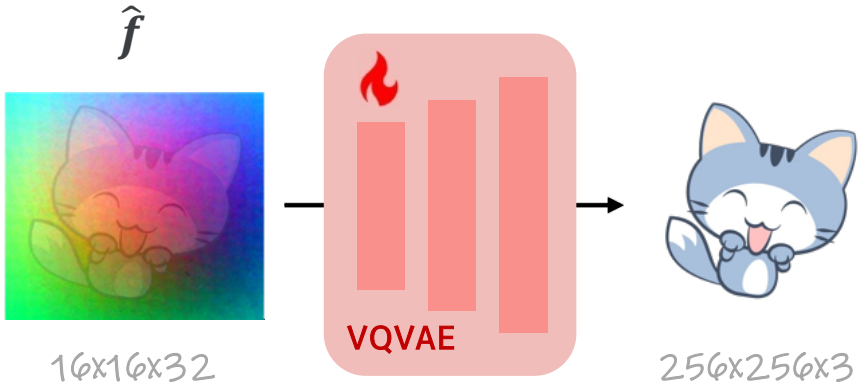
**Algorithm 1:** Multi-scale VQVAE Encoding

**Inputs:** raw image $im$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$f = \mathcal{E}(im)$, $R = []$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \mathcal{Q}(\text{interpolate}(f, h_k, w_k))$;
  $R = \text{queue\_push}(R, r_k)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $f = f - \phi_k(z_k)$;
**Return:** multi-scale tokens $R$;

**Algorithm 2:** Multi-scale VQVAE Reconstruction

**Inputs:** multi-scale token maps $R$;
**Hyperparameters:** steps $K$, resolutions $(h_k, w_k)_{k=1}^{K}$;
$\hat{f} = 0$;
**for** $k = 1, \cdots, K$ **do**
  $r_k = \text{queue\_pop}(R)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $\hat{f} = \hat{f} + \phi_k(z_k)$;
$\hat{im} = \mathcal{D}(\hat{f})$;
**Return:** reconstructed image $\hat{im}$;




## * Loss

$$||x - \hat{x}||^2 + ||sg[E(x)] - z_{\mathbf{q}}||_2^2 + ||sg[z_{\mathbf{q}}] - E(x)||_2^2$$

* encoder
* decoder

* codebook

* encoder
( reg )

# Code | VQVAE Training

NAVER AI LAB

```python
def vqvae_training(img, patch_nums):
    # hyper-parameter
    codebook_num = 4096
    codebook_dim = 32
    codebook = nn.Embedding(codebook_num, codebook_dim)

    # encode
    latent = vqvae_encode(img) # [batch_size, 32, 16, 16]
    f_hat = get_fhat(f=latent, codebook=codebook, patch_nums=patch_nums)

    # recon
    recon_img = vqvae_decode(f_hat)

    # loss
    perceptual_loss = 0
    discriminative_loss = 0
    loss = mse_loss(latent, f_hat) + mse_loss(img, recon_img) + perceptual_loss + discriminative_loss

    return loss
```

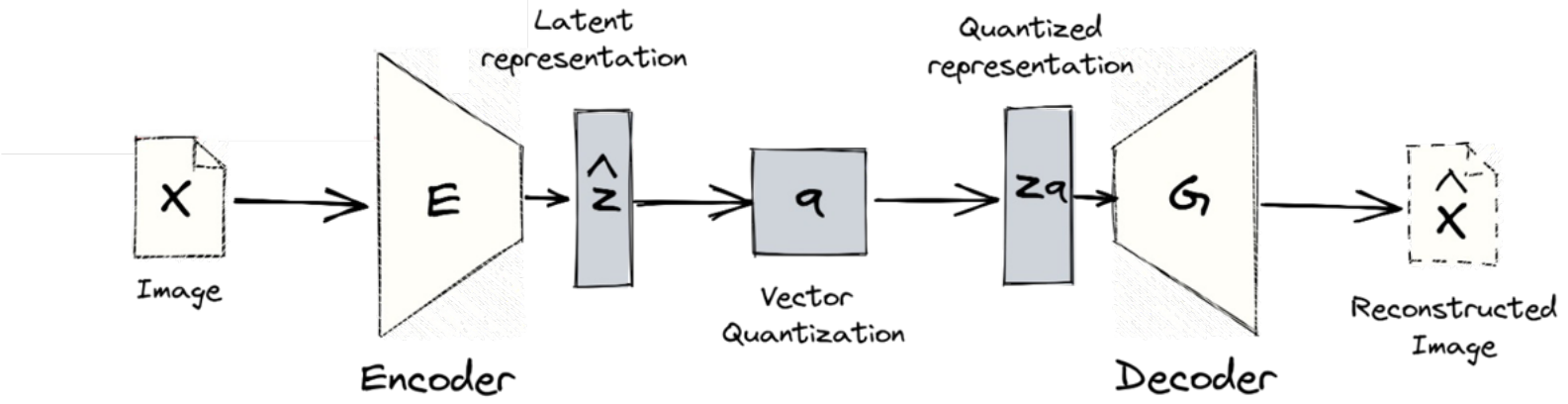
```python
def get_fhat(f, codebook, patch_nums):

    batch_size = f.shape[0]
    codebook_dim = codebook.weight.shape[-1]
    origin_size = patch_nums[-1]

    # init
    f_hat = 0

    for pk in patch_nums:
        fk = resize(f, pk, mode='area').permute(0, 2, 3, 1).view(-1, codebook_dim) # [batch_size * pk * pk, codebook_dim]

        rk = get_idx_with_codebook(fk, codebook).view([batch_size, pk, pk]) # R.append(rk)

        zk = codebook(idx).permute(0, 3, 1, 2) # [bs, codebook_dim, pk, pk]
        zk = resize(zk, origin_size, mode='bicubic')
        zk = phi_conv(zk) # resize error 보완

        f_hat += zk # for recon
        f -= zk # for next token

    return f_hat
```

# Code | VQVAE Training

NAVER AI LAB

```python
def vqvae_training(img, patch_nums):
    # hyper-parameter
    codebook_num = 4096
    codebook_dim = 32
    codebook = nn.Embedding(codebook_num, codebook_dim)

    # encode
    latent = vqvae_encode(img) # [batch_size, 32, 16, 16]
    f_hat = get_fhat(f=latent, codebook=codebook, patch_nums=patch_nums)

    # recon
    recon_img = vqvae_decode(f_hat)

    # loss
    perceptual_loss = 0
    discriminative_loss = 0
    loss = mse_loss(latent, f_hat) + mse_loss(img, recon_img) + perceptual_loss + discriminative_loss

    return loss
```

```python
def get_fhat(f, codebook, patch_nums):

    batch_size = f.shape[0]
    codebook_dim = codebook.weight.shape[-1]
    origin_size = patch_nums[-1]

    # init
    f_hat = 0

    for pk in patch_nums:
        fk = resize(f, pk, mode='area').permute(0, 2, 3, 1).view(-1, codebook_dim) # [batch_size * pk * pk, codebook_dim]

        rk = get_idx_with_codebook(fk, codebook).view([batch_size, pk, pk]) # R.append(rk)

        zk = codebook(idx).permute(0, 3, 1, 2) # [bs, codebook_dim, pk, pk]
        zk = resize(zk, origin_size, mode='bicubic')
        zk = phi_conv(zk) # resize error 보완

        f_hat += zk # for recon
        f -= zk # for next token

    return f_hat
```




*** Loss**

$$\underbrace{||x - \hat{x}||^2}_{\text{* encoder, * decoder}} + \underbrace{||sg[E(x)] - z_{\mathbf{q}}||_2^2}_{\text{* codebook}} + \underbrace{||sg[z_{\mathbf{q}}] - E(x)||_2^2}_{\text{* encoder (reg)}}$$

# Code | VQVAE Training

NAVER AI LAB

```python
def vqvae_training(img, patch_nums):
    # hyper-parameter
    codebook_num = 4096
    codebook_dim = 32
    codebook = nn.Embedding(codebook_num, codebook_dim)

    # encode
    latent = vqvae_encode(img) # [batch_size, 32, 16, 16]
    f_hat = get_fhat(f=latent, codebook=codebook, patch_nums=patch_nums)

    # recon
    recon_img = vqvae_decode(f_hat)

    # loss
    perceptual_loss = 0
    discriminative_loss = 0
    loss = mse_loss(latent, f_hat) + mse_loss(img, recon_img) + perceptual_loss + discriminative_loss

    return loss
```

```python
def get_fhat(f, codebook, patch_nums):

    batch_size = f.shape[0]
    codebook_dim = codebook.weight.shape[-1]
    origin_size = patch_nums[-1]

    # init
    f_hat = 0

    for pk in patch_nums:
        fk = resize(f, pk, mode='area').permute(0, 2, 3, 1).view(-1, codebook_dim) # [batch_size * pk * pk, codebook_dim]

        rk = get_idx_with_codebook(fk, codebook).view([batch_size, pk, pk]) # R.append(rk)

        zk = codebook(idx).permute(0, 3, 1, 2) # [bs, codebook_dim, pk, pk]
        zk = resize(zk, origin_size, mode='bicubic')
        zk = phi_conv(zk) # resize error 보완

        f_hat += zk # for recon
        f -= zk # for next token

    return f_hat
```

# Code | VQVAE Training

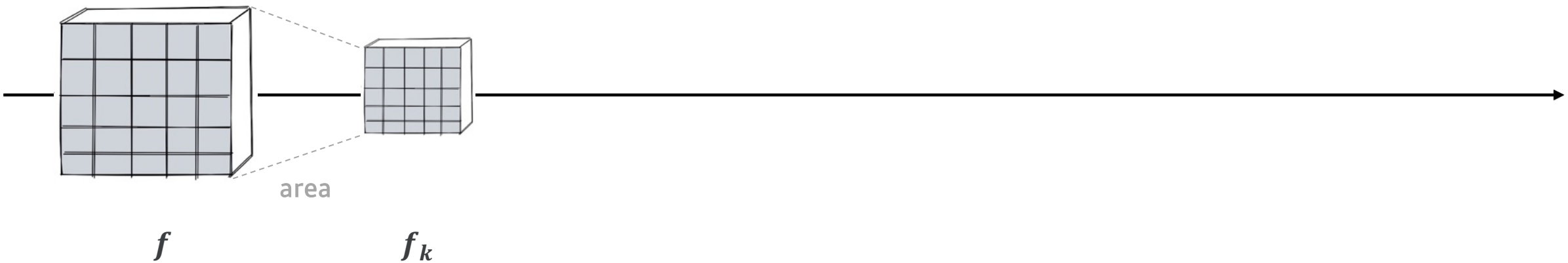
NAVER AI LAB

```python
def vqvae_training(img, patch_nums):
    # hyper-parameter
    codebook_num = 4096
    codebook_dim = 32
    codebook = nn.Embedding(codebook_num, codebook_dim)

    # encode
    latent = vqvae_encode(img) # [batch_size, 32, 16, 16]
    f_hat = get_fhat(f=latent, codebook=codebook, patch_nums=patch_nums)

    # recon
    recon_img = vqvae_decode(f_hat)

    # loss
    perceptual_loss = 0
    discriminative_loss = 0
    loss = mse_loss(latent, f_hat) + mse_loss(img, recon_img) + perceptual_loss + discriminative_loss

    return loss
```

```python
def get_fhat(f, codebook, patch_nums):

    batch_size = f.shape[0]
    codebook_dim = codebook.weight.shape[-1]
    origin_size = patch_nums[-1]

    # init
    f_hat = 0

    for pk in patch_nums:
        fk = resize(f, pk, mode='area').permute(0, 2, 3, 1).view(-1, codebook_dim) # [batch_size * pk * pk, codebook_dim]

        rk = get_idx_with_codebook(fk, codebook).view([batch_size, pk, pk]) # R.append(rk)

        zk = codebook(idx).permute(0, 3, 1, 2) # [bs, codebook_dim, pk, pk]
        zk = resize(zk, origin_size, mode='bicubic')
        zk = phi_conv(zk) # resize error 보완

        f_hat += zk # for recon
        f -= zk # for next token

    return f_hat
```

patch: [1,2,3, … , 16]

**for k in patch_K:**



# Code | VQVAE Training

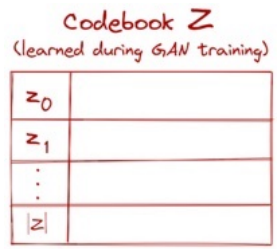
NAVER AI LAB

```python
def vqvae_training(img, patch_nums):
    # hyper-parameter
    codebook_num = 4096
    codebook_dim = 32
    codebook = nn.Embedding(codebook_num, codebook_dim)

    # encode
    latent = vqvae_encode(img) # [batch_size, 32, 16, 16]
    f_hat = get_fhat(f=latent, codebook=codebook, patch_nums=patch_nums)

    # recon
    recon_img = vqvae_decode(f_hat)

    # loss
    perceptual_loss = 0
    discriminative_loss = 0
    loss = mse_loss(latent, f_hat) + mse_loss(img, recon_img) + perceptual_loss + discriminative_loss

    return loss
```

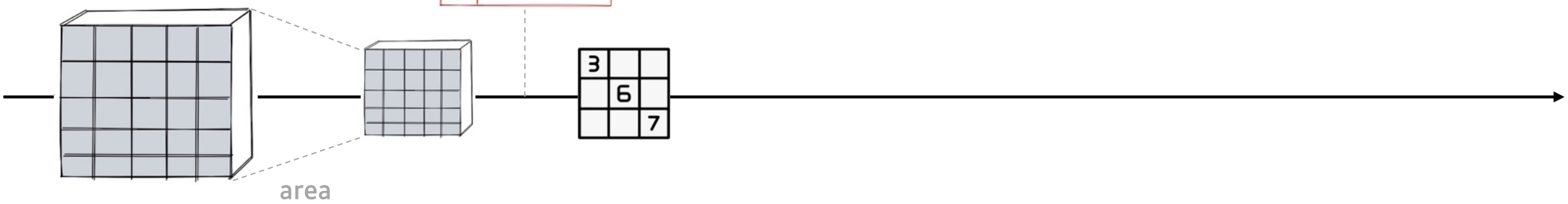
```python
def get_fhat(f, codebook, patch_nums):

    batch_size = f.shape[0]
    codebook_dim = codebook.weight.shape[-1]
    origin_size = patch_nums[-1]

    # init
    f_hat = 0

    for pk in patch_nums:
        fk = resize(f, pk, mode='area').permute(0, 2, 3, 1).view(-1, codebook_dim) # [batch_size * pk * pk, codebook_dim]

        rk = get_idx_with_codebook(fk, codebook).view([batch_size, pk, pk]) # R.append(rk)

        zk = codebook(idx).permute(0, 3, 1, 2) # [bs, codebook_dim, pk, pk]
        zk = resize(zk, origin_size, mode='bicubic')
        zk = phi_conv(zk) # resize error 보완

        f_hat += zk # for recon
        f -= zk # for next token

    return f_hat
```

patch: [1,2,3, … , 16]

for k in patch_K:




```python
def get_idx_with_codebook(rk, codebook):
    distances = F.pairwise_distance(rk.unsqueeze(1), codebook.weight.data.unsqueeze(0))  # [N, 1, C], [1, V, C] -> [N, V]
    idx = torch.argmin(distances, dim=1)
    return idx
```




$f$ $f_k$ $r_k$

# Code | VQVAE Training

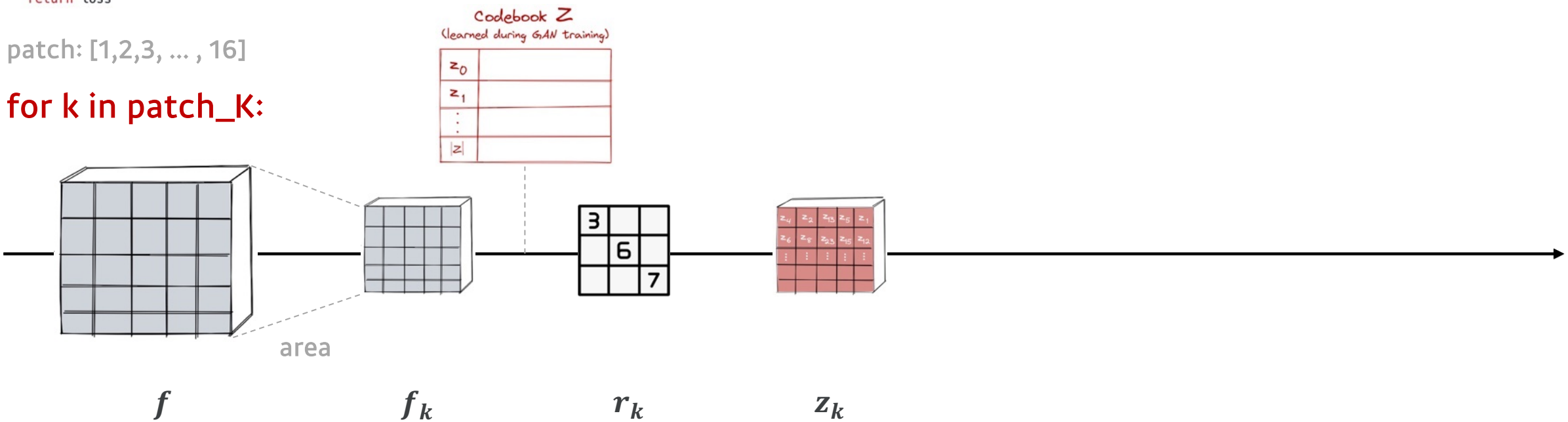
NAVER AI LAB

```python
def vqvae_training(img, patch_nums):
    # hyper-parameter
    codebook_num = 4096
    codebook_dim = 32
    codebook = nn.Embedding(codebook_num, codebook_dim)

    # encode
    latent = vqvae_encode(img) # [batch_size, 32, 16, 16]
    f_hat = get_fhat(f=latent, codebook=codebook, patch_nums=patch_nums)

    # recon
    recon_img = vqvae_decode(f_hat)

    # loss
    perceptual_loss = 0
    discriminative_loss = 0
    loss = mse_loss(latent, f_hat) + mse_loss(img, recon_img) + perceptual_loss + discriminative_loss

    return loss
```

```python
def get_fhat(f, codebook, patch_nums):

    batch_size = f.shape[0]
    codebook_dim = codebook.weight.shape[-1]
    origin_size = patch_nums[-1]

    # init
    f_hat = 0

    for pk in patch_nums:
        fk = resize(f, pk, mode='area').permute(0, 2, 3, 1).view(-1, codebook_dim) # [batch_size * pk * pk, codebook_dim]

        rk = get_idx_with_codebook(fk, codebook).view([batch_size, pk, pk]) # R.append(rk)

        zk = codebook(idx).permute(0, 3, 1, 2) # [bs, codebook_dim, pk, pk]
        zk = resize(zk, origin_size, mode='bicubic')
        zk = phi_conv(zk) # resize error 보완

        f_hat += zk # for recon
        f -= zk # for next token

    return f_hat
```

patch: [1,2,3, … , 16]

**for k in patch_K:**



# Code | VQVAE Training

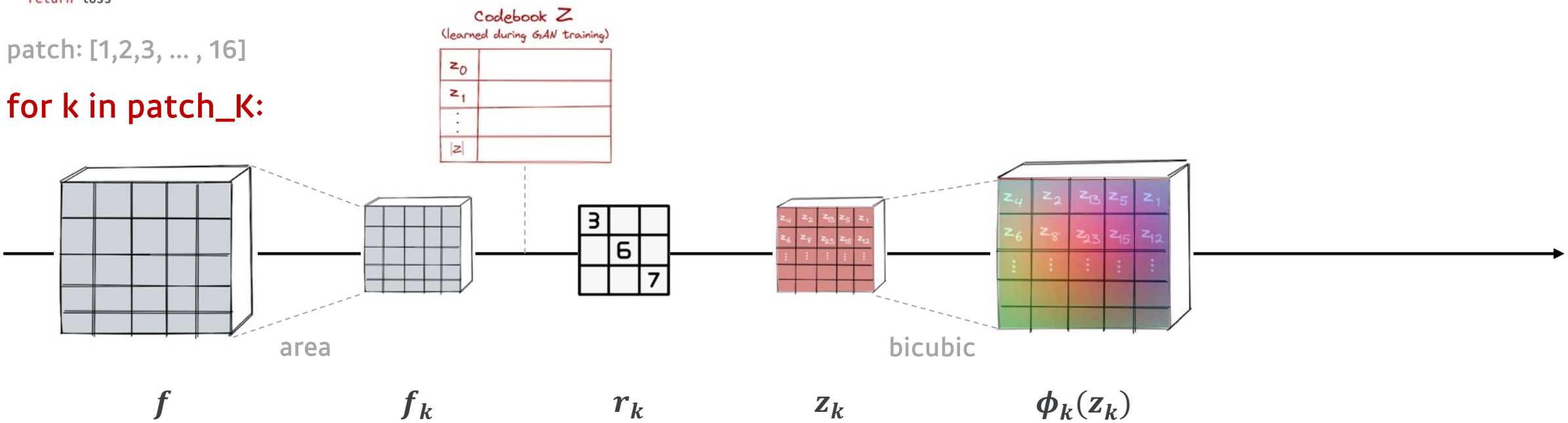
NAVER AI LAB

```python
def vqvae_training(img, patch_nums):
    # hyper-parameter
    codebook_num = 4096
    codebook_dim = 32
    codebook = nn.Embedding(codebook_num, codebook_dim)

    # encode
    latent = vqvae_encode(img) # [batch_size, 32, 16, 16]
    f_hat = get_fhat(f=latent, codebook=codebook, patch_nums=patch_nums)

    # recon
    recon_img = vqvae_decode(f_hat)

    # loss
    perceptual_loss = 0
    discriminative_loss = 0
    loss = mse_loss(latent, f_hat) + mse_loss(img, recon_img) + perceptual_loss + discriminative_loss

    return loss
```

```python
def get_fhat(f, codebook, patch_nums):

    batch_size = f.shape[0]
    codebook_dim = codebook.weight.shape[-1]
    origin_size = patch_nums[-1]

    # init
    f_hat = 0

    for pk in patch_nums:
        fk = resize(f, pk, mode='area').permute(0, 2, 3, 1).view(-1, codebook_dim) # [batch_size * pk * pk, codebook_dim]

        rk = get_idx_with_codebook(fk, codebook).view([batch_size, pk, pk]) # R.append(rk)

        zk = codebook(idx).permute(0, 3, 1, 2) # [bs, codebook_dim, pk, pk]
        zk = resize(zk, origin_size, mode='bicubic')
        zk = phi_conv(zk) # resize error 보완

        f_hat += zk # for recon
        f -= zk # for next token

    return f_hat
```

patch: [1,2,3, … , 16]

**for k in patch_K:**



# Code | VQVAE Training

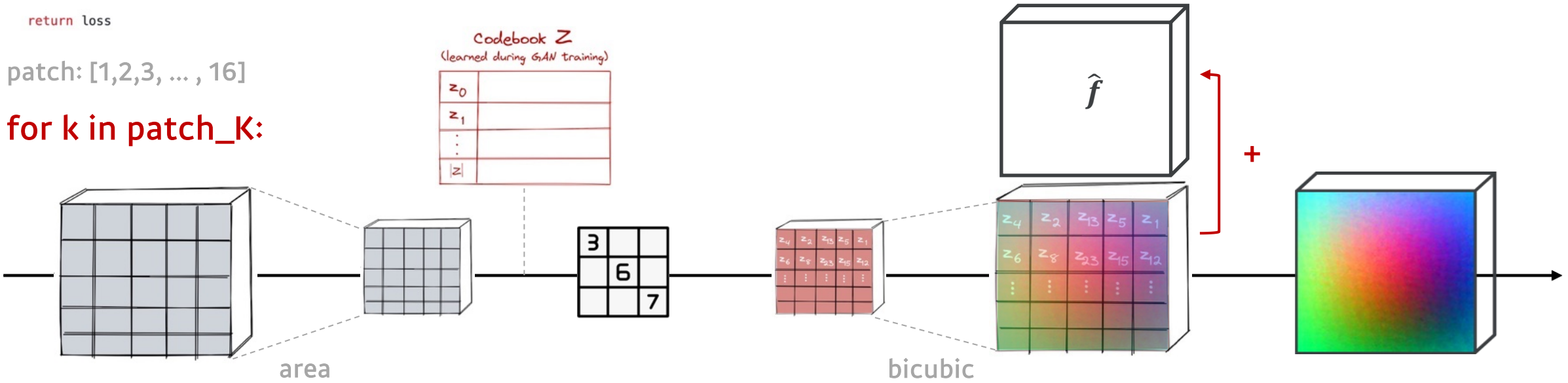
NAVER AI LAB

```python
def vqvae_training(img, patch_nums):
    # hyper-parameter
    codebook_num = 4096
    codebook_dim = 32
    codebook = nn.Embedding(codebook_num, codebook_dim)

    # encode
    latent = vqvae_encode(img) # [batch_size, 32, 16, 16]
    f_hat = get_fhat(f=latent, codebook=codebook, patch_nums=patch_nums)

    # recon
    recon_img = vqvae_decode(f_hat)

    # loss
    perceptual_loss = 0
    discriminative_loss = 0
    loss = mse_loss(latent, f_hat) + mse_loss(img, recon_img) + perceptual_loss + discriminative_loss

    return loss
```

```python
def get_fhat(f, codebook, patch_nums):

    batch_size = f.shape[0]
    codebook_dim = codebook.weight.shape[-1]
    origin_size = patch_nums[-1]

    # init
    f_hat = 0

    for pk in patch_nums:
        fk = resize(f, pk, mode='area').permute(0, 2, 3, 1).view(-1, codebook_dim) # [batch_size * pk * pk, codebook_dim]

        rk = get_idx_with_codebook(fk, codebook).view([batch_size, pk, pk]) # R.append(rk)

        zk = codebook(idx).permute(0, 3, 1, 2) # [bs, codebook_dim, pk, pk]
        zk = resize(zk, origin_size, mode='bicubic')
        zk = phi_conv(zk) # resize error 보완

        f_hat += zk # for recon
        f -= zk # for next token

    return f_hat
```

patch: [1,2,3, … , 16]

**for k in patch_K:**

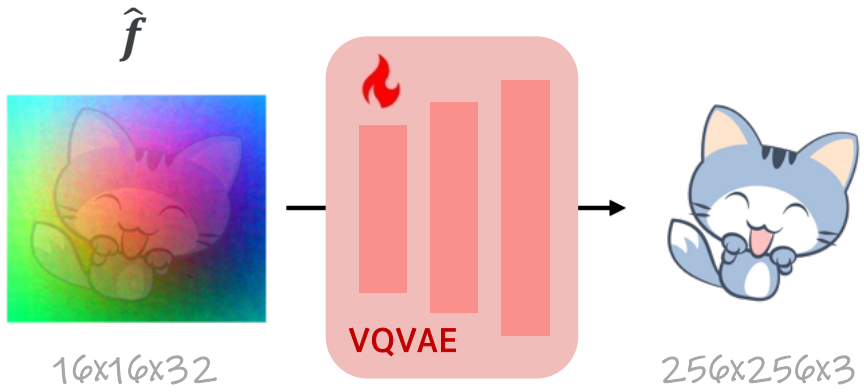


# Code | VQVAE Training

NAVER AI LAB

```python
def vqvae_training(img, patch_nums):
    # hyper-parameter
    codebook_num = 4096
    codebook_dim = 32
    codebook = nn.Embedding(codebook_num, codebook_dim)

    # encode
    latent = vqvae_encode(img) # [batch_size, 32, 16, 16]
    f_hat = get_fhat(f=latent, codebook=codebook, patch_nums=patch_nums)

    # recon
    recon_img = vqvae_decode(f_hat)

    # loss
    perceptual_loss = 0
    discriminative_loss = 0
    loss = mse_loss(latent, f_hat) + mse_loss(img, recon_img) + perceptual_loss + discriminative_loss

    return loss
```

```python
def get_fhat(f, codebook, patch_nums):

    batch_size = f.shape[0]
    codebook_dim = codebook.weight.shape[-1]
    origin_size = patch_nums[-1]

    # init
    f_hat = 0

    for pk in patch_nums:
        fk = resize(f, pk, mode='area').permute(0, 2, 3, 1).view(-1, codebook_dim) # [batch_size * pk * pk, codebook_dim]

        rk = get_idx_with_codebook(fk, codebook).view([batch_size, pk, pk]) # R.append(rk)

        zk = codebook(idx).permute(0, 3, 1, 2) # [bs, codebook_dim, pk, pk]
        zk = resize(zk, origin_size, mode='bicubic')
        zk = phi_conv(zk) # resize error 보완

        f_hat += zk # for recon
        f -= zk # for next token

    return f_hat
```



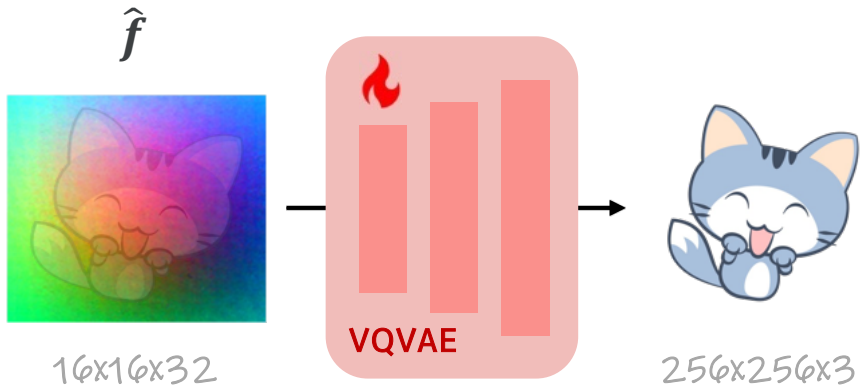

The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale

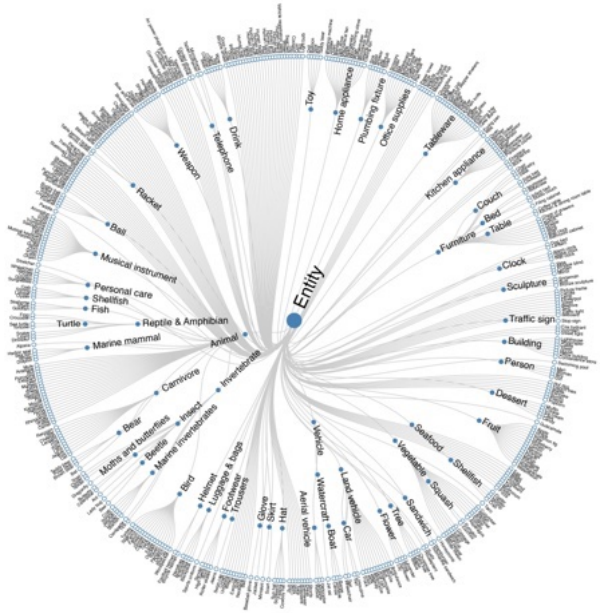
# Code | VQVAE Training

NAVER AI LAB

```python
def vqvae_training(img, patch_nums):
    # hyper-parameter
    codebook_num = 4096
    codebook_dim = 32
    codebook = nn.Embedding(codebook_num, codebook_dim)

    # encode
    latent = vqvae_encode(img) # [batch_size, 32, 16, 16]
    f_hat = get_fhat(f=latent, codebook=codebook, patch_nums=patch_nums)

    # recon
    recon_img = vqvae_decode(f_hat)

    # loss
    perceptual_loss = 0
    discriminative_loss = 0
    loss = mse_loss(latent, f_hat) + mse_loss(img, recon_img) + perceptual_loss + discriminative_loss

    return loss
```

```python
def get_fhat(f, codebook, patch_nums):

    batch_size = f.shape[0]
    codebook_dim = codebook.weight.shape[-1]
    origin_size = patch_nums[-1]

    # init
    f_hat = 0

    for pk in patch_nums:
        fk = resize(f, pk, mode='area').permute(0, 2, 3, 1).view(-1, codebook_dim) # [batch_size * pk * pk, codebook_dim]

        rk = get_idx_with_codebook(fk, codebook).view([batch_size, pk, pk]) # R.append(rk)

        zk = codebook(idx).permute(0, 3, 1, 2) # [bs, codebook_dim, pk, pk]
        zk = resize(zk, origin_size, mode='bicubic')
        zk = phi_conv(zk) # resize error 보완

        f_hat += zk # for recon
        f -= zk # for next token

    return f_hat
```

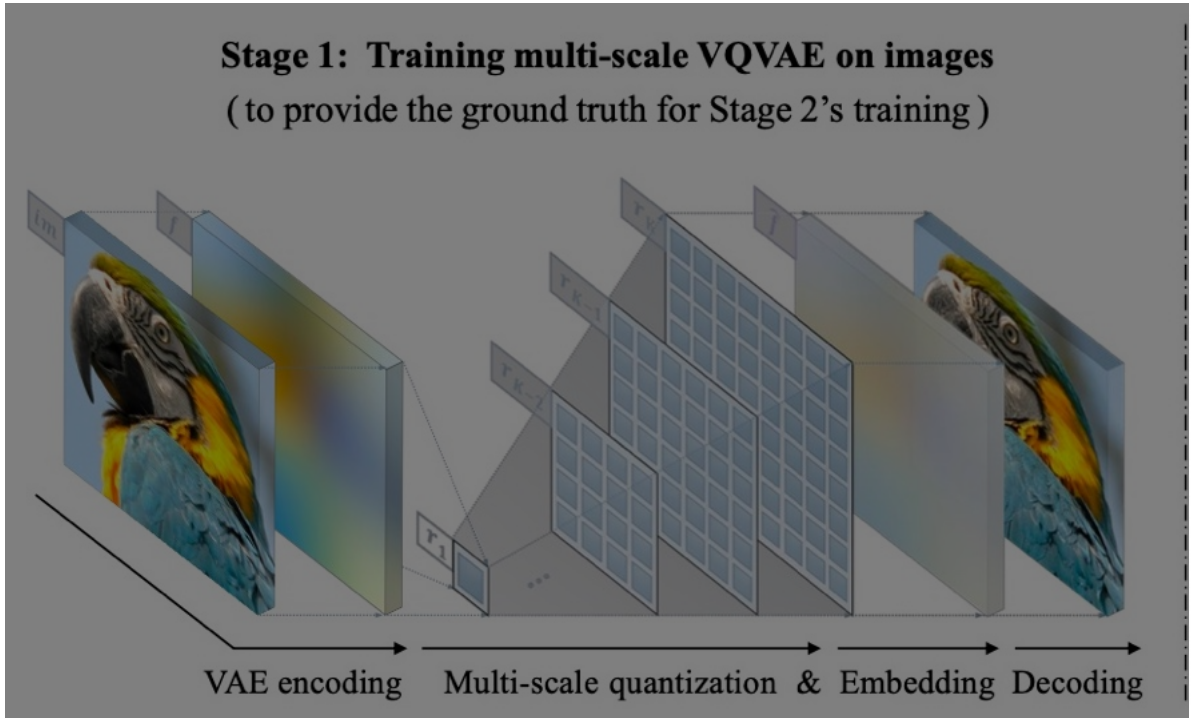



**9M numbers**
**6k class**

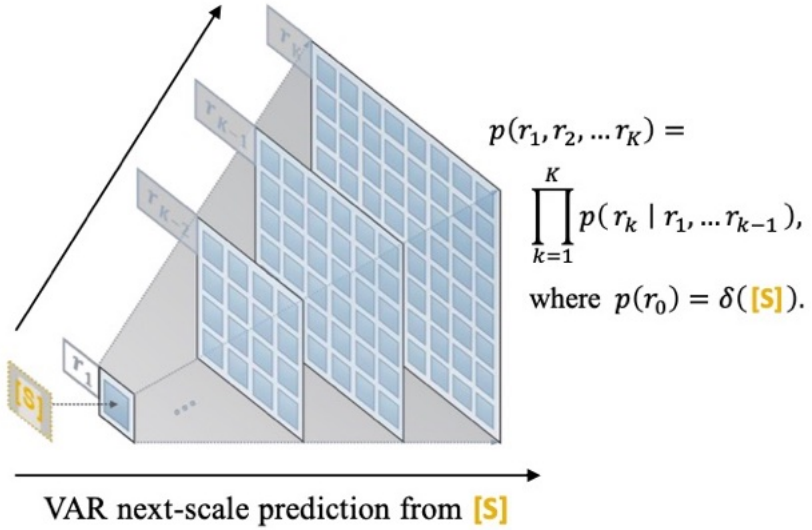
The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale

# Code | VAR Training




**Stage 2: Training VAR transformer on tokens**

([S] means a start token w/ or w/o condition information)



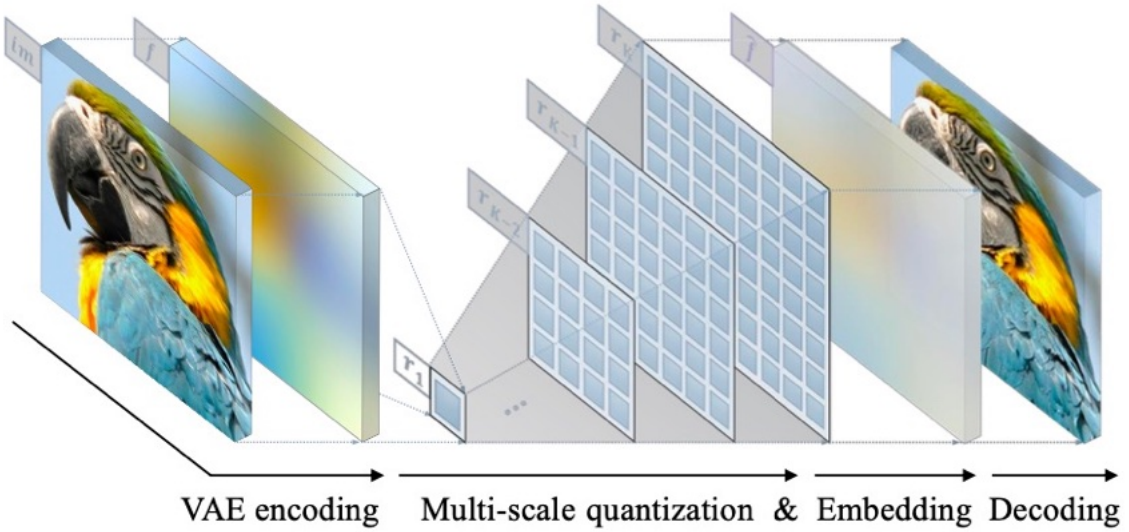

$$p(r_1, r_2, \dots r_K) = \prod_{k=1}^{K} p(r_k \mid r_1, \dots r_{k-1}),$$

where $p(r_0) = \delta([S])$.

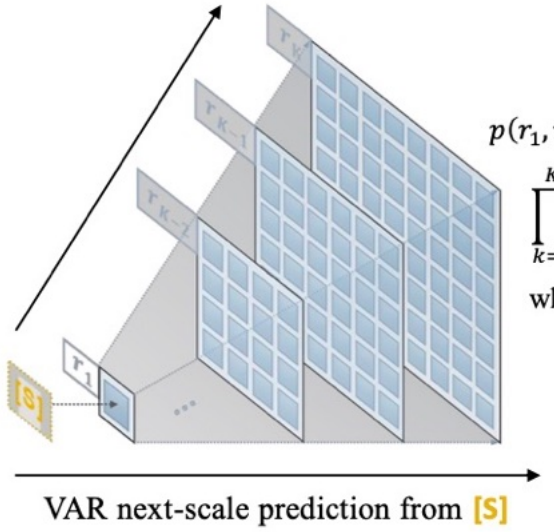
# Code | VAR Training

NAVER AI LAB

**Stage 1: Training multi-scale VQVAE on images**
( to provide the ground truth for Stage 2's training )

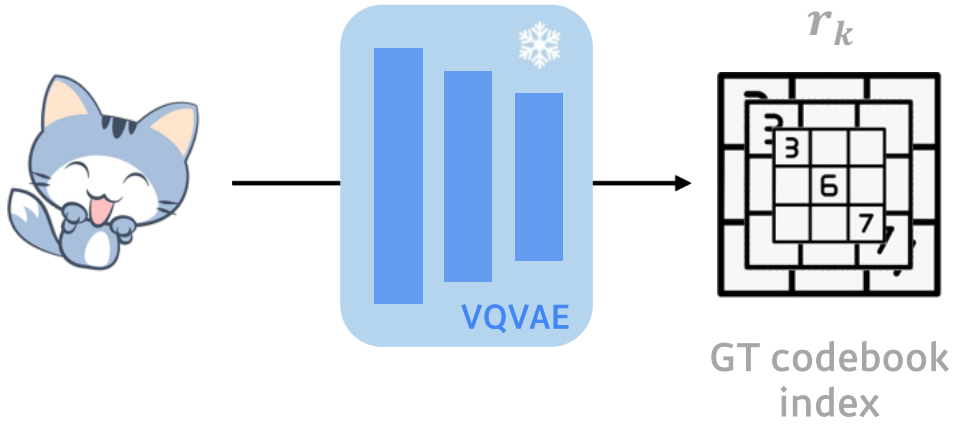



**Stage 2: Training VAR transformer on tokens**
([S] means a start token w/ or w/o condition information)




$$p(r_1, r_2, \dots r_K) = \prod_{k=1}^{K} p(r_k \mid r_1, \dots r_{k-1}),$$

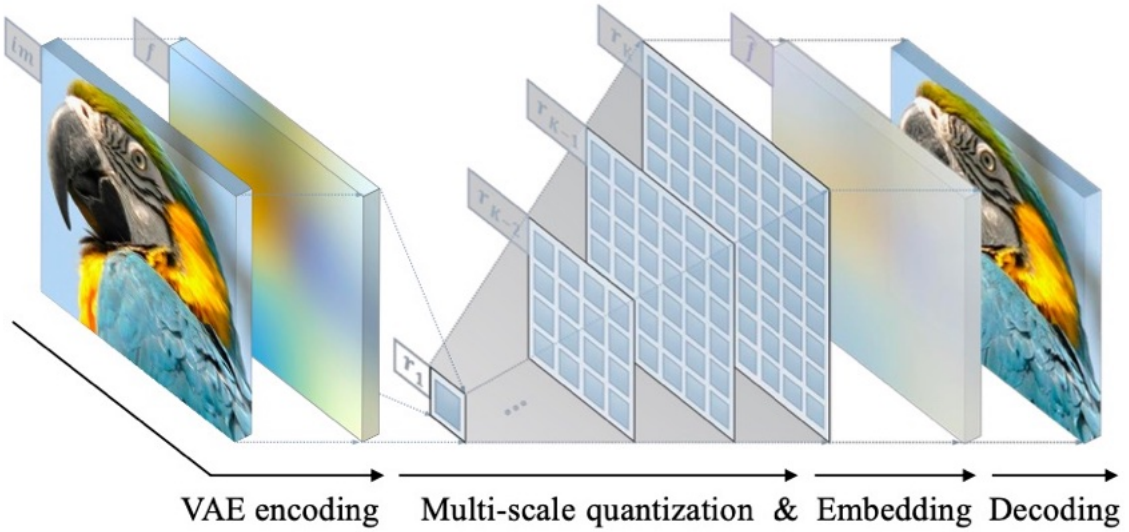
where $p(r_0) = \delta([S])$.



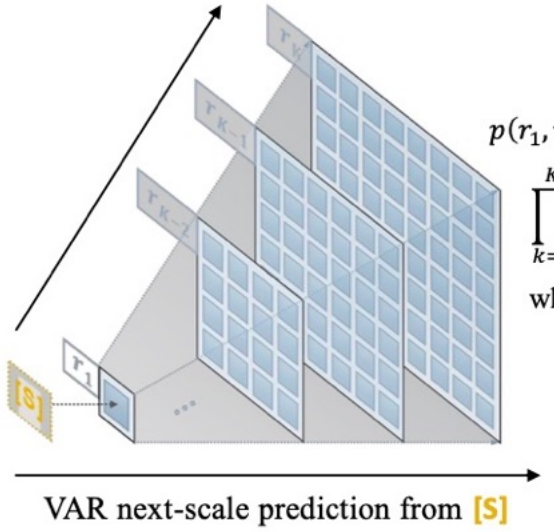
# Code | VAR Training

NAVER AI LAB

**Stage 1: Training multi-scale VQVAE on images**
( to provide the ground truth for Stage 2's training )

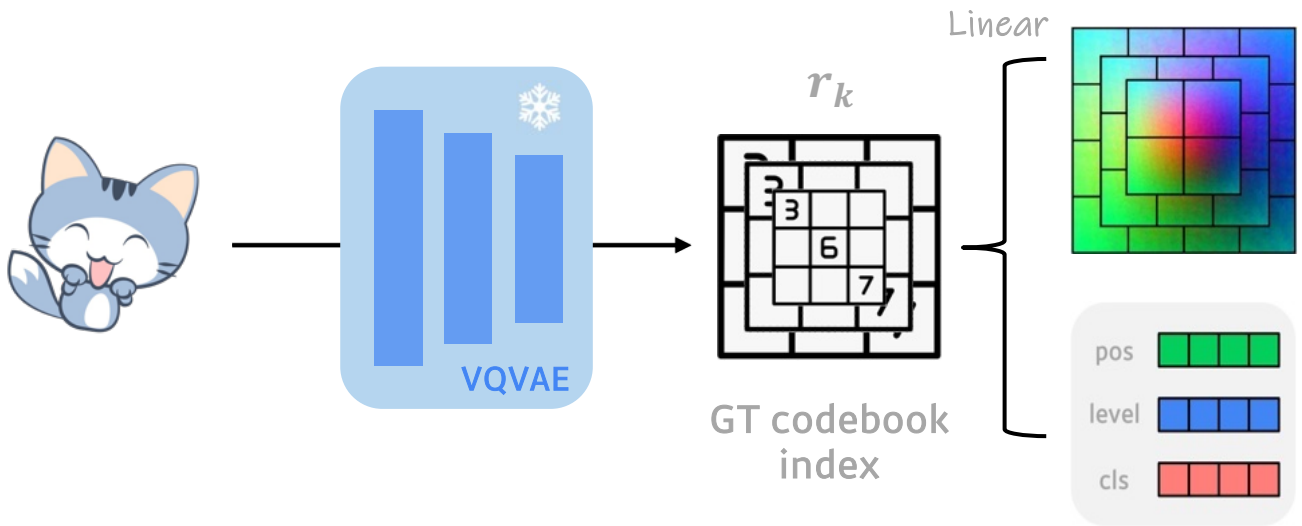



**Stage 2: Training VAR transformer on tokens**
([S] means a start token w/ or w/o condition information)




$$p(r_1, r_2, \dots r_K) = \prod_{k=1}^{K} p(r_k \mid r_1, \dots r_{k-1}),$$

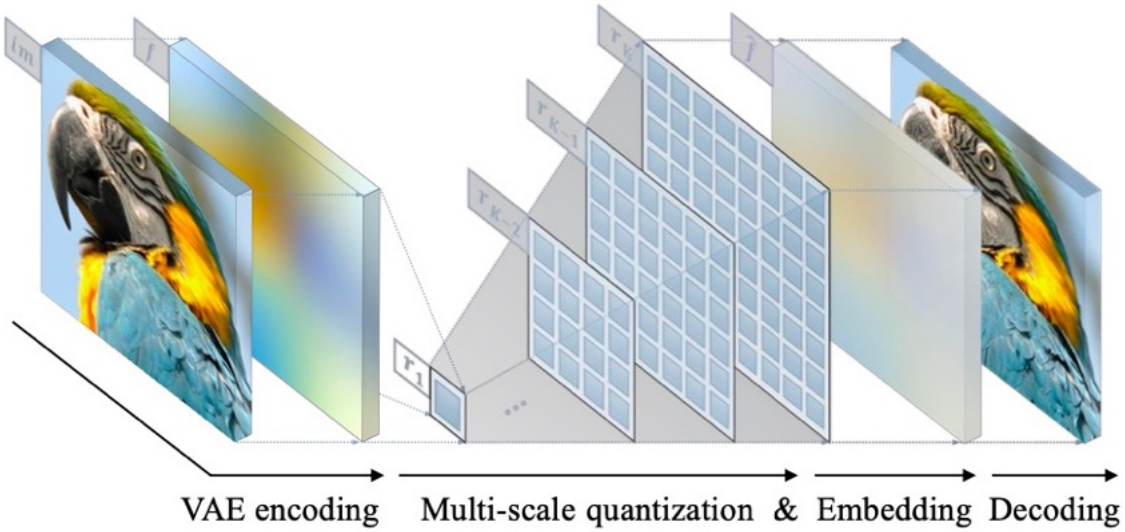
where $p(r_0) = \delta([S])$.



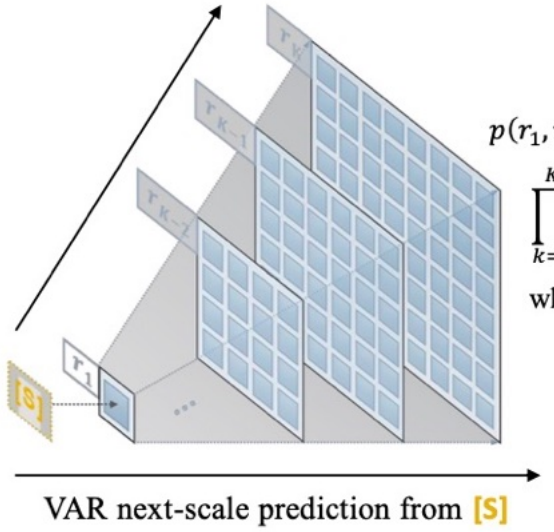
# Code | VAR Training

NAVER AI LAB

**Stage 1: Training multi-scale VQVAE on images**
( to provide the ground truth for Stage 2's training )

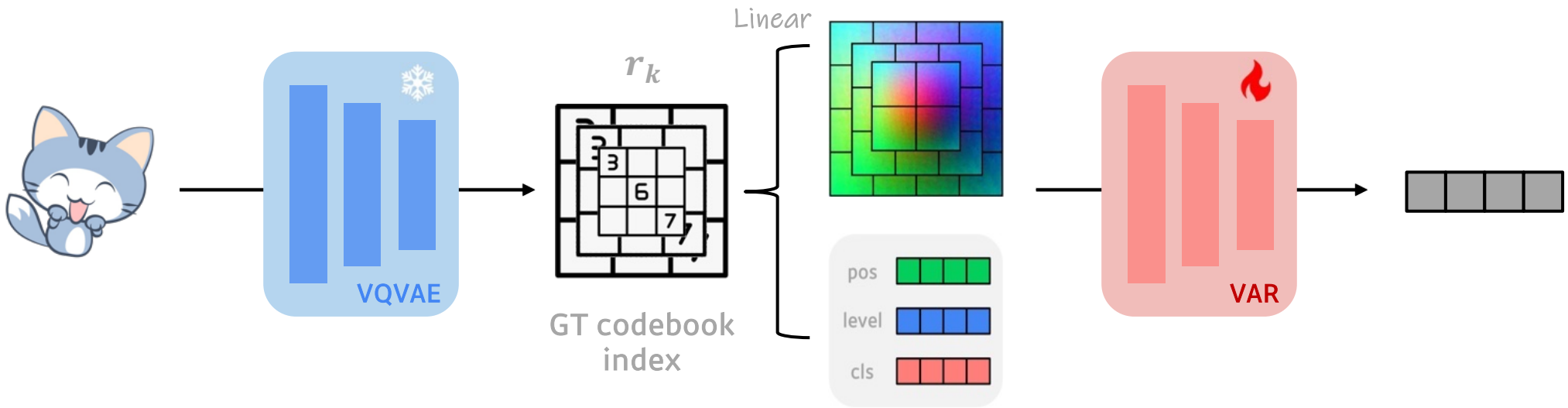



**Stage 2: Training VAR transformer on tokens**
([S] means a start token w/ or w/o condition information)




$$p(r_1, r_2, \dots r_K) = \prod_{k=1}^{K} p(r_k \mid r_1, \dots r_{k-1}),$$

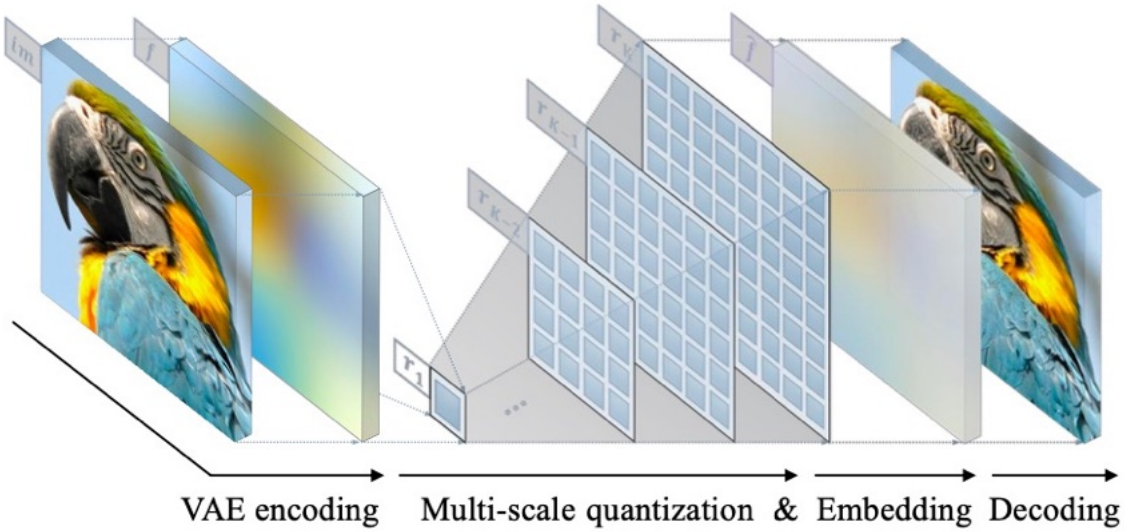
where $p(r_0) = \delta([S])$.



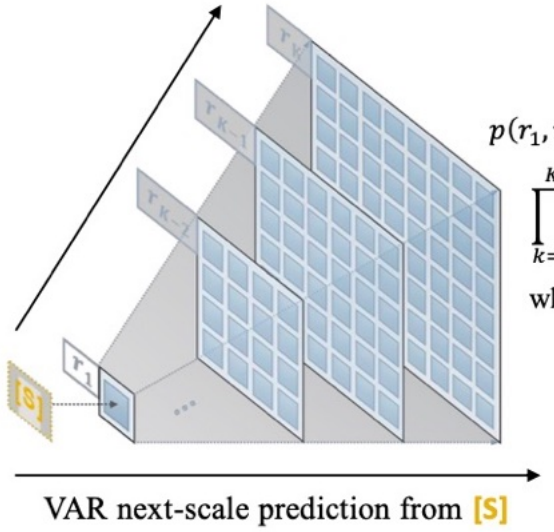
# Code | VAR Training

NAVER AI LAB

**Stage 1: Training multi-scale VQVAE on images**
(to provide the ground truth for Stage 2's training)

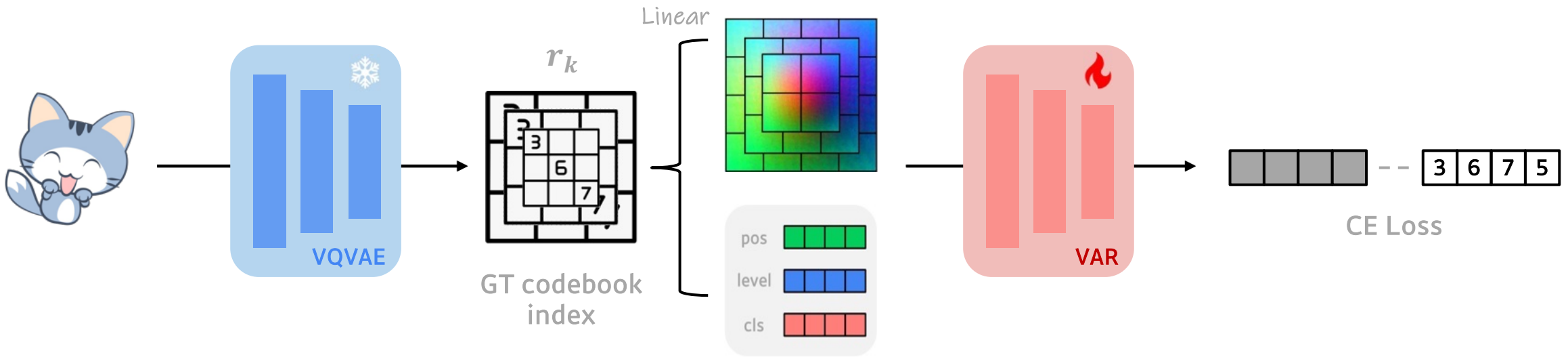



**Stage 2: Training VAR transformer on tokens**
([S] means a start token w/ or w/o condition information)




$$p(r_1, r_2, \dots r_K) = \prod_{k=1}^{K} p(r_k \mid r_1, \dots r_{k-1}),$$

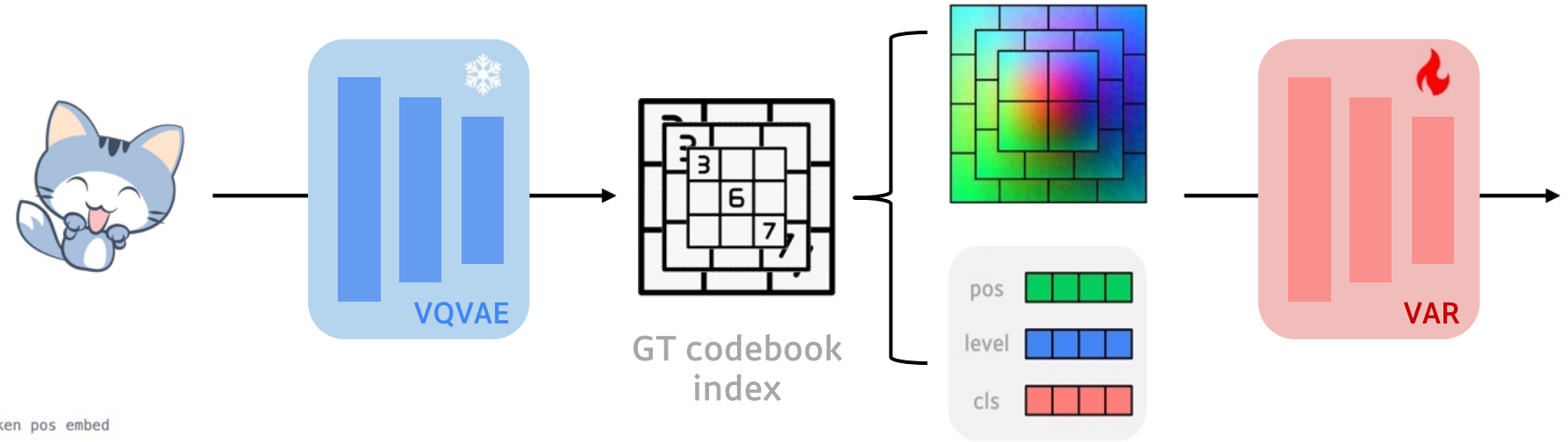
where $p(r_0) = \delta([S])$.



# Code | VAR Training

NAVER AI LAB

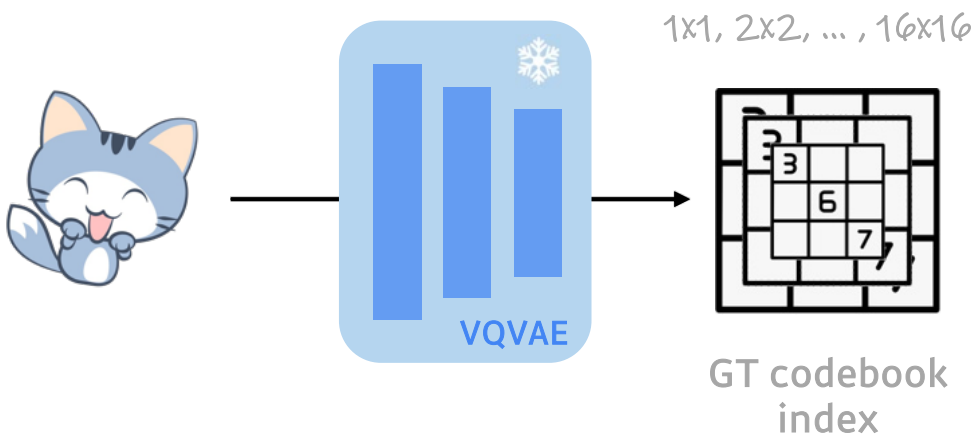
```python
def var_training(img, patch_nums, class_label):
    depth = 16
    class_num = 1000 # ImageNet
    embed_dim = 1024 # depth * 64
    L = sum(pn * pn for pn in patch_nums)

    """ GT Index w/ pretrained vqvae """
    gt_idx, codebook = get_idx_GT(img, patch_nums)
    gt_idx = torch.cat(gt_idx, dim=1) # [batch_size, L]

    """ define embedding """
    # codebook
    class_codebook = nn.Embedding(class_num+1, embed_dim)
    level_codebook = nn.Embedding(len(patch_nums), embed_dim)

    # position
    pos_start_embed = torch.randn([1, 1, embed_dim]) # start token pos embed
    pos_emb = torch.randn([1, L, embed_dim])

    # level
    level_seq = [torch.full((pn * pn,), i) for i, pn in enumerate(patch_nums)]
    level_seq = torch.cat(level_seq).unsqueeze(0)

    # embedding
    level_emb = level_codebook(level_seq) + pos_emb
    class_emb = class_codebook(class_label) + pos_start_embed

    """ Input """
    token_maps = get_token_maps(R, codebook, patch_nums) # [batch_size, L-1, codebook_dim]
    token_maps_emb = token_linear(token_maps, embed_dim)
    token_maps_emb = torch.cat([class_emb, token_maps_emb], dim=1) + level_emb

    """ Insert condition """
    for i in range(depth):
        token_maps_emb = token_block(token_maps_emb, class_emb) # SA + FFN

    """ Output: loss """
    # logit
    logit = get_logit(token_maps_emb, class_emb) # [bs, L, 4096]

    # loss
    loss = cross_entropy(logit, gt_idx) # [batch_size, L]
    loss = loss.sum(dim=1).mean()

    return loss
```



# Code | VAR Training

NAVER AI LAB

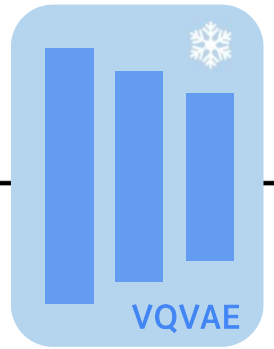
```python
def var_training(img, patch_nums, class_label):
    depth = 16
    class_num = 1000 # ImageNet
    embed_dim = 1024 # depth * 64
    L = sum(pn * pn for pn in patch_nums)

    """ GT Index w/ pretrained vqvae """
    gt_idx, codebook = get_idx_GT(img, patch_nums)
    gt_idx = torch.cat(gt_idx, dim=1) # [batch_size, L]

    """ define embedding """
    # codebook
    class_codebook = nn.Embedding(class_num+1, embed_dim)
    level_codebook = nn.Embedding(len(patch_nums), embed_dim)

    # position
    pos_start_embed = torch.randn([1, 1, embed_dim]) # start token pos embed
    pos_emb = torch.randn([1, L, embed_dim])

    # level
    level_seq = [torch.full((pn * pn,), i) for i, pn in enumerate(patch_nums)]
    level_seq = torch.cat(level_seq).unsqueeze(0)

    # embedding
    level_emb = level_codebook(level_seq) + pos_emb
    class_emb = class_codebook(class_label) + pos_start_embed

    """ Input """
    token_maps = get_token_maps(R, codebook, patch_nums) # [batch_size, L-1, codebook_dim]
    token_maps_emb = token_linear(token_maps, embed_dim)
    token_maps_emb = torch.cat([class_emb, token_maps_emb], dim=1) + level_emb

    """ Insert condition """
    for i in range(depth):
        token_maps_emb = token_block(token_maps_emb, class_emb) # SA + FFN

    """ Output: loss """
    # logit
    logit = get_logit(token_maps_emb, class_emb) # [bs, L, 4096]

    # loss
    loss = cross_entropy(logit, gt_idx) # [batch_size, L]
    loss = loss.sum(dim=1).mean()

    return loss
```

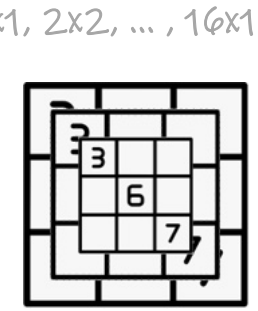


# Code | VAR Training

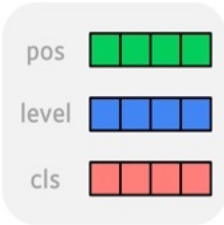
NAVER AI LAB

```python
def var_training(img, patch_nums, class_label):
    depth = 16
    class_num = 1000 # ImageNet
    embed_dim = 1024 # depth * 64
    L = sum(pn * pn for pn in patch_nums)

    """ GT Index w/ pretrained vqvae """
    gt_idx, codebook = get_idx_GT(img, patch_nums)
    gt_idx = torch.cat(gt_idx, dim=1) # [batch_size, L]

    """ define embedding """
    # codebook
    class_codebook = nn.Embedding(class_num+1, embed_dim)
    level_codebook = nn.Embedding(len(patch_nums), embed_dim)

    # position
    pos_start_embed = torch.randn([1, 1, embed_dim]) # start token pos embed
    pos_emb = torch.randn([1, L, embed_dim])

    # level
    level_seq = [torch.full((pn * pn,), i) for i, pn in enumerate(patch_nums)]
    level_seq = torch.cat(level_seq).unsqueeze(0)

    # embedding
    level_emb = level_codebook(level_seq) + pos_emb
    class_emb = class_codebook(class_label) + pos_start_embed

    """ Input """
    token_maps = get_token_maps(R, codebook, patch_nums) # [batch_size, L-1, codebook_dim]
    token_maps_emb = token_linear(token_maps, embed_dim)
    token_maps_emb = torch.cat([class_emb, token_maps_emb], dim=1) + level_emb

    """ Insert condition """
    for i in range(depth):
        token_maps_emb = token_block(token_maps_emb, class_emb) # SA + FFN

    """ Output: loss """
    # logit
    logit = get_logit(token_maps_emb, class_emb) # [bs, L, 4096]

    # loss
    loss = cross_entropy(logit, gt_idx) # [batch_size, L]
    loss = loss.sum(dim=1).mean()

    return loss
```




GT codebook index

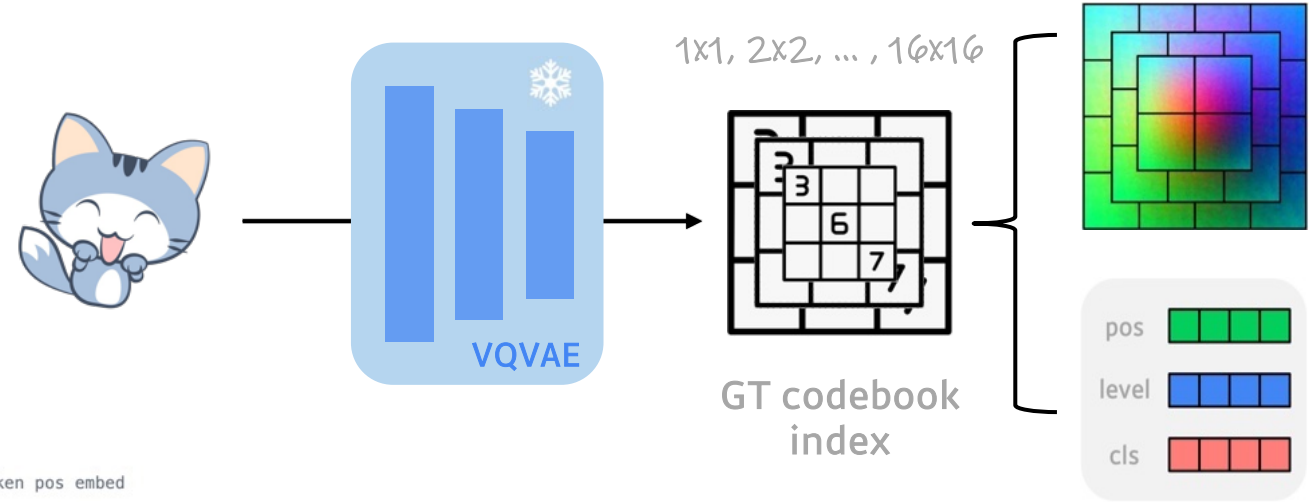



```python
def _init_embed(init_embed, embed_dim):
    init_std = math.sqrt(1 / embed_dim / 3)

    # 클래스 임베딩 or level embedding
    embed = nn.Embedding(init_embed, embed_dim)
    nn.init.trunc_normal_(embed.weight.data, mean=0, std=init_std)

    return embed

def _init_position_embed(patch_nums, embed_dim):
    init_std = math.sqrt(1 / embed_dim / 3)

    # 첫 위치 임베딩 (for class emb)
    first_patch_size = patch_nums[0] ** 2
    pos_start_embed = torch.empty(1, first_patch_size, embed_dim) # [1, 1, 1024]
    nn.init.trunc_normal_(pos_start_embed, mean=0, std=init_std)
    pos_start_embed = nn.Parameter(pos_start_embed)

    # 전체 위치 임베딩
    pos_embed = []
    for pn in patch_nums:
        patch_size = pn ** 2
        pe = torch.empty(1, patch_size, embed_dim)
        nn.init.trunc_normal_(pe, mean=0, std=init_std)
        pos_embed.append(pe)
    pos_embed = nn.Parameter(torch.cat(pos_embed, dim=1)) # [1, total_patches, embed_dim]

    return pos_start_embed, pos_embed
```

# Code | VAR Training

NAVER AI LAB

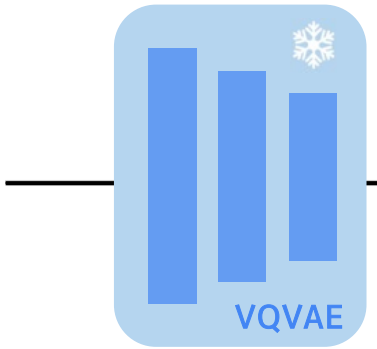
```python
def var_training(img, patch_nums, class_label):
    depth = 16
    class_num = 1000 # ImageNet
    embed_dim = 1024 # depth * 64
    L = sum(pn * pn for pn in patch_nums)

    """ GT Index w/ pretrained vqvae """
    gt_idx, codebook = get_idx_GT(img, patch_nums)
    gt_idx = torch.cat(gt_idx, dim=1) # [batch_size, L]

    """ define embedding """
    # codebook
    class_codebook = nn.Embedding(class_num+1, embed_dim)
    level_codebook = nn.Embedding(len(patch_nums), embed_dim)

    # position
    pos_start_embed = torch.randn([1, 1, embed_dim]) # start token pos embed
    pos_emb = torch.randn([1, L, embed_dim])

    # level
    level_seq = [torch.full((pn * pn,), i) for i, pn in enumerate(patch_nums)]
    level_seq = torch.cat(level_seq).unsqueeze(0)

    # embedding
    level_emb = level_codebook(level_seq) + pos_emb
    class_emb = class_codebook(class_label) + pos_start_embed

    """ Input """
    token_maps = get_token_maps(R, codebook, patch_nums) # [batch_size, L-1, codebook_dim]
    token_maps_emb = token_linear(token_maps, embed_dim)
    token_maps_emb = torch.cat([class_emb, token_maps_emb], dim=1) + level_emb

    """ Insert condition """
    for i in range(depth):
        token_maps_emb = token_block(token_maps_emb, class_emb) # SA + FFN

    """ Output: loss """
    # logit
    logit = get_logit(token_maps_emb, class_emb) # [bs, L, 4096]

    # loss
    loss = cross_entropy(logit, gt_idx) # [batch_size, L]
    loss = loss.sum(dim=1).mean()

    return loss
```

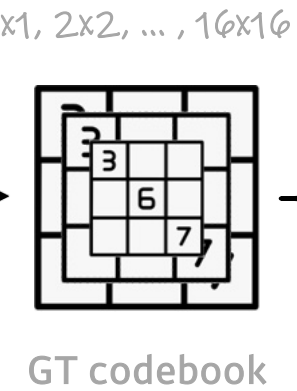


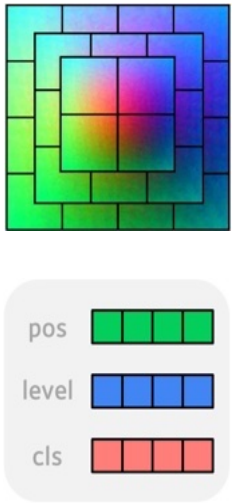
# Code | VAR Training

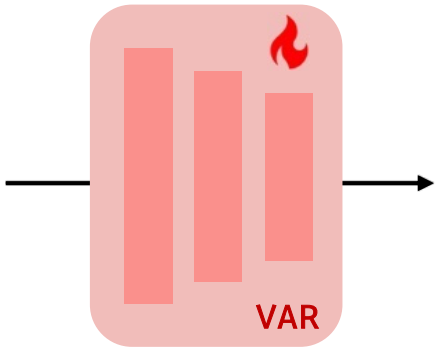
NAVER AI LAB

```python
def var_training(img, patch_nums, class_label):
    depth = 16
    class_num = 1000 # ImageNet
    embed_dim = 1024 # depth * 64
    L = sum(pn * pn for pn in patch_nums)

    """ GT Index w/ pretrained vqvae """
    gt_idx, codebook = get_idx_GT(img, patch_nums)
    gt_idx = torch.cat(gt_idx, dim=1) # [batch_size, L]

    """ define embedding """
    # codebook
    class_codebook = nn.Embedding(class_num+1, embed_dim)
    level_codebook = nn.Embedding(len(patch_nums), embed_dim)

    # position
    pos_start_embed = torch.randn([1, 1, embed_dim]) # start token pos embed
    pos_emb = torch.randn([1, L, embed_dim])

    # level
    level_seq = [torch.full((pn * pn,), i) for i, pn in enumerate(patch_nums)]
    level_seq = torch.cat(level_seq).unsqueeze(0)

    # embedding
    level_emb = level_codebook(level_seq) + pos_emb
    class_emb = class_codebook(class_label) + pos_start_embed

    """ Input """
    token_maps = get_token_maps(R, codebook, patch_nums) # [batch_size, L-1, codebook_dim]
    token_maps_emb = token_linear(token_maps, embed_dim)
    token_maps_emb = torch.cat([class_emb, token_maps_emb], dim=1) + level_emb

    """ Insert condition """
    for i in range(depth):
        token_maps_emb = token_block(token_maps_emb, class_emb) # SA + FFN

    """ Output: loss """
    # logit
    logit = get_logit(token_maps_emb, class_emb) # [bs, L, 4096]

    # loss
    loss = cross_entropy(logit, gt_idx) # [batch_size, L]
    loss = loss.sum(dim=1).mean()

    return loss
```

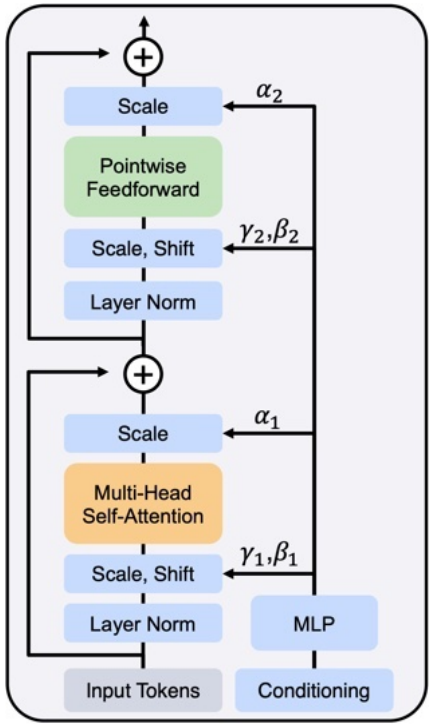


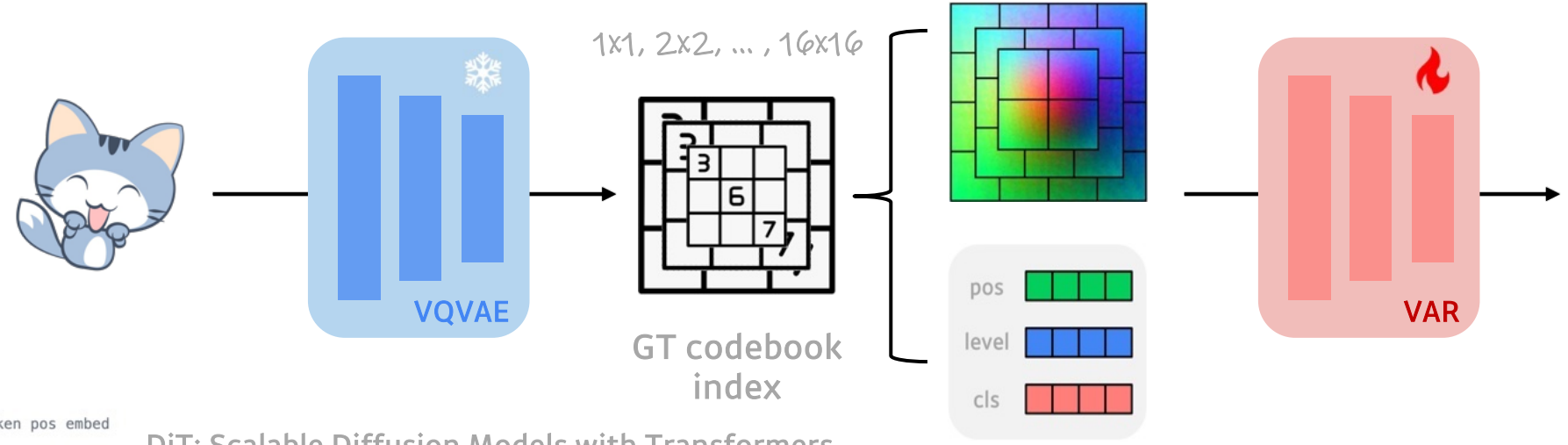

DiT: Scalable Diffusion Models with Transformers



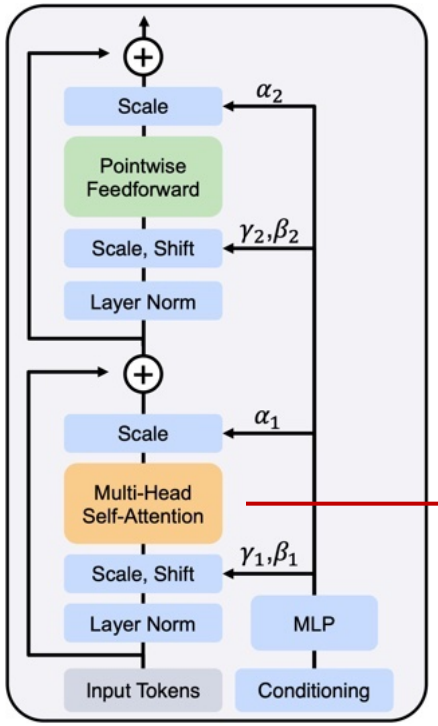
# Code | VAR Training

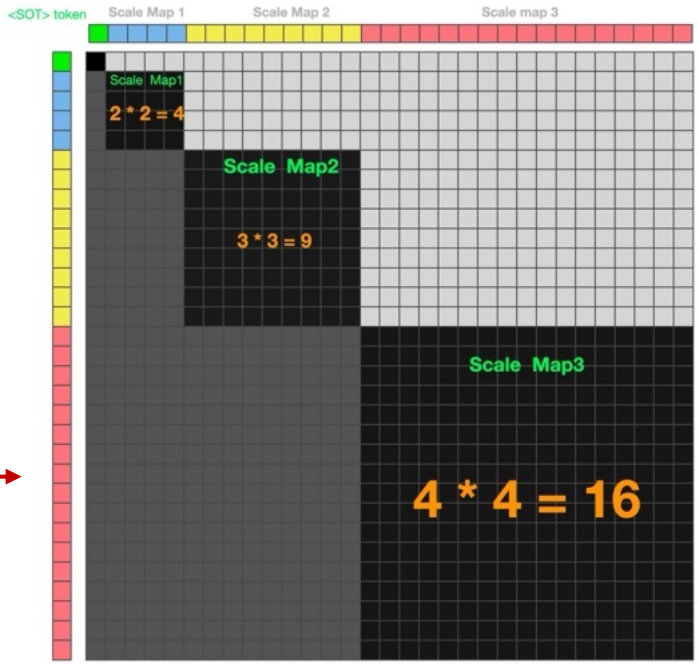
NAVER AI LAB

```python
def var_training(img, patch_nums, class_label):
    depth = 16
    class_num = 1000 # ImageNet
    embed_dim = 1024 # depth * 64
    L = sum(pn * pn for pn in patch_nums)

    """ GT Index w/ pretrained vqvae """
    gt_idx, codebook = get_idx_GT(img, patch_nums)
    gt_idx = torch.cat(gt_idx, dim=1) # [batch_size, L]

    """ define embedding """
    # codebook
    class_codebook = nn.Embedding(class_num+1, embed_dim)
    level_codebook = nn.Embedding(len(patch_nums), embed_dim)

    # position
    pos_start_embed = torch.randn([1, 1, embed_dim]) # start token pos embed
    pos_emb = torch.randn([1, L, embed_dim])

    # level
    level_seq = [torch.full((pn * pn,), i) for i, pn in enumerate(patch_nums)]
    level_seq = torch.cat(level_seq).unsqueeze(0)

    # embedding
    level_emb = level_codebook(level_seq) + pos_emb
    class_emb = class_codebook(class_label) + pos_start_embed

    """ Input """
    token_maps = get_token_maps(R, codebook, patch_nums) # [batch_size, L-1, codebook_dim]
    token_maps_emb = token_linear(token_maps, embed_dim)
    token_maps_emb = torch.cat([class_emb, token_maps_emb], dim=1) + level_emb

    """ Insert condition """
    for i in range(depth):
        token_maps_emb = token_block(token_maps_emb, class_emb) # SA + FFN

    """ Output: loss """
    # logit
    logit = get_logit(token_maps_emb, class_emb) # [bs, L, 4096]

    # loss
    loss = cross_entropy(logit, gt_idx) # [batch_size, L]
    loss = loss.sum(dim=1).mean()

    return loss
```




DiT: Scalable Diffusion Models with Transformers



# Code | VAR Inference

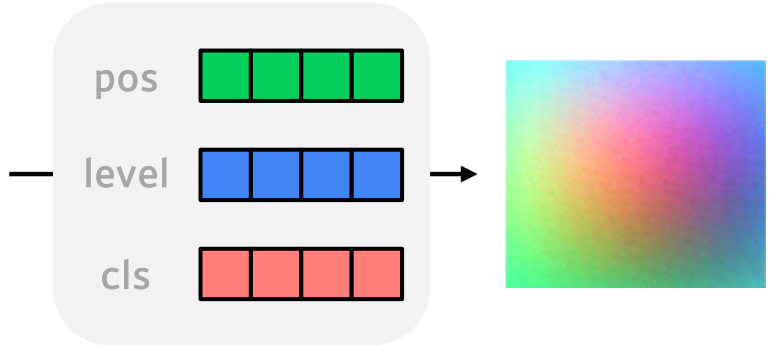
NAVER AI LAB

```python
def var_inference(class_label, patch_nums):
    batch_size = class_label.shape[0]
    L = sum(pn * pn for pn in patch_nums)
    embed_dim = 1024
    depth = 16

    # pretrained-vqvae
    codebook_num = 4096
    codebook_dim = 32
    vqvae_codebook = nn.Embedding(codebook_num, codebook_dim)

    # pre embedding
    class_emb = torch.randn([batch_size * 2, 1, embed_dim]) # class_embedding(class_label) + pos_start
    pos_emb = torch.randn([batch_size, L, embed_dim])
    level_emb = torch.randn([batch_size, L, embed_dim])

    """ Input """
    token_emb = class_emb + pos_emb[:, :1] + level_emb[:, :1] # [bs * 2, 1, embed_dim] -> start_token
    latent = torch.zeros([batch_size, 32, 16, 16])

    """ Insert condition """
    last_stage_index = len(patch_nums) - 1
    current_position = 0
    for stage_index, pn in enumerate(patch_nums):
        stage_ratio = stage_index / last_stage_index
        current_position += pn * pn

        for i in range(depth):
            token_emb = token_block(token_emb, class_emb)

        logit = get_logit(token_emb, class_emb)

        # classfier-free guidance
        cfg_scale = 5
        t = cfg_scale * stage_ratio
        logit = (1 + t) * logit[:batch_size] - t * logit[batch_size:]

        # sampling
        sampled_logit = sampling_top_k_p(logit, top_k=900, top_p=0.96, num_samples=1)[:, :, 0]

        # get token embedding
        token_emb = vqvae_codebook(sampled_logit)
        token_emb = token_emb.transpose(1, 2).view(batch_size, codebook_dim, pn, pn)

        # get next token embedding
        latent, next_token_emb = get_next_autoregressive_input(stage_index, patch_nums, latent, token_emb)

        if stage_index != last_stage_index:
            next_token_emb = next_token_emb.view(batch_size, codebook_dim, -1).transpose(1, 2)

            next_position = current_position + patch_nums[stage_index + 1] ** 2

            # [bs, next_patch*next_patch, embed_dim]
            next_token_map = token_linear(next_token_emb, embed_dim) + level_emb[:, current_position: next_position]

            token_emb = next_token_map.repeat(2, 1, 1)

    """ Output: Generated image """
    img = vqvae_decode(latent)

    return img
```

# Code | VAR Inference

NAVER AI LAB

```python
def var_inference(class_label, patch_nums):
    batch_size = class_label.shape[0]
    L = sum(pn * pn for pn in patch_nums)
    embed_dim = 1024
    depth = 16

    # pretrained-vqvae
    codebook_num = 4096
    codebook_dim = 32
    vqvae_codebook = nn.Embedding(codebook_num, codebook_dim)

    # pre embedding
    class_emb = torch.randn([batch_size * 2, 1, embed_dim]) # class_embedding(class_label) + pos_start
    pos_emb = torch.randn([batch_size, L, embed_dim])
    level_emb = torch.randn([batch_size, L, embed_dim])

    """ Input """
    token_emb = class_emb + pos_emb[:, :1] + level_emb[:, :1] # [bs * 2, 1, embed_dim] -> start_token
    latent = torch.zeros([batch_size, 32, 16, 16])

    """ Insert condition """
    last_stage_index = len(patch_nums) - 1
    current_position = 0
    for stage_index, pn in enumerate(patch_nums):
        stage_ratio = stage_index / last_stage_index
        current_position += pn * pn

        for i in range(depth):
            token_emb = token_block(token_emb, class_emb)

        logit = get_logit(token_emb, class_emb)

        # classfier-free guidance
        cfg_scale = 5
        t = cfg_scale * stage_ratio
        logit = (1 + t) * logit[:batch_size] - t * logit[batch_size:]

        # sampling
        sampled_logit = sampling_top_k_p(logit, top_k=900, top_p=0.96, num_samples=1)[:, :, 0]

        # get token embedding
        token_emb = vqvae_codebook(sampled_logit)
        token_emb = token_emb.transpose(1, 2).view(batch_size, codebook_dim, pn, pn)

        # get next token embedding
        latent, next_token_emb = get_next_autoregressive_input(stage_index, patch_nums, latent, token_emb)

        if stage_index != last_stage_index:
            next_token_emb = next_token_emb.view(batch_size, codebook_dim, -1).transpose(1, 2)

            next_position = current_position + patch_nums[stage_index + 1] ** 2

            # [bs, next_patch*next_patch, embed_dim]
            next_token_map = token_linear(next_token_emb, embed_dim) + level_emb[:, current_position: next_position]

            token_emb = next_token_map.repeat(2, 1, 1)

    """ Output: Generated image """
    img = vqvae_decode(latent)

    return img
```

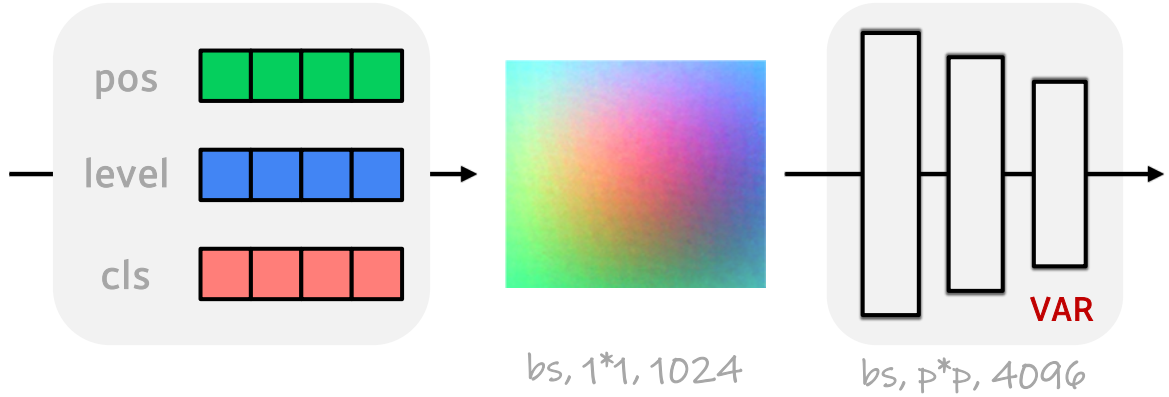



bs, 1*1, 1024

# Code | VAR Inference

NAVER AI LAB

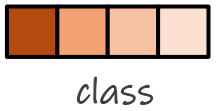
```python
def var_inference(class_label, patch_nums):
    batch_size = class_label.shape[0]
    L = sum(pn * pn for pn in patch_nums)
    embed_dim = 1024
    depth = 16

    # pretrained-vqvae
    codebook_num = 4096
    codebook_dim = 32
    vqvae_codebook = nn.Embedding(codebook_num, codebook_dim)

    # pre embedding
    class_emb = torch.randn([batch_size * 2, 1, embed_dim]) # class_embedding(class_label) + pos_start
    pos_emb = torch.randn([batch_size, L, embed_dim])
    level_emb = torch.randn([batch_size, L, embed_dim])

    """ Input """
    token_emb = class_emb + pos_emb[:, :1] + level_emb[:, :1] # [bs * 2, 1, embed_dim] -> start_token
    latent = torch.zeros([batch_size, 32, 16, 16])

    """ Insert condition """
    last_stage_index = len(patch_nums) - 1
    current_position = 0
    for stage_index, pn in enumerate(patch_nums):
        stage_ratio = stage_index / last_stage_index
        current_position += pn * pn

        for i in range(depth):
            token_emb = token_block(token_emb, class_emb)

        logit = get_logit(token_emb, class_emb)

        # classfier-free guidance
        cfg_scale = 5
        t = cfg_scale * stage_ratio
        logit = (1 + t) * logit[:batch_size] - t * logit[batch_size:]

        # sampling
        sampled_logit = sampling_top_k_p(logit, top_k=900, top_p=0.96, num_samples=1)[:, :, 0]

        # get token embedding
        token_emb = vqvae_codebook(sampled_logit)
        token_emb = token_emb.transpose(1, 2).view(batch_size, codebook_dim, pn, pn)

        # get next token embedding
        latent, next_token_emb = get_next_autoregressive_input(stage_index, patch_nums, latent, token_emb)

        if stage_index != last_stage_index:
            next_token_emb = next_token_emb.view(batch_size, codebook_dim, -1).transpose(1, 2)

            next_position = current_position + patch_nums[stage_index + 1] ** 2

            # [bs, next_patch*next_patch, embed_dim]
            next_token_map = token_linear(next_token_emb, embed_dim) + level_emb[:, current_position: next_position]

            token_emb = next_token_map.repeat(2, 1, 1)

    """ Output: Generated image """
    img = vqvae_decode(latent)

    return img
```

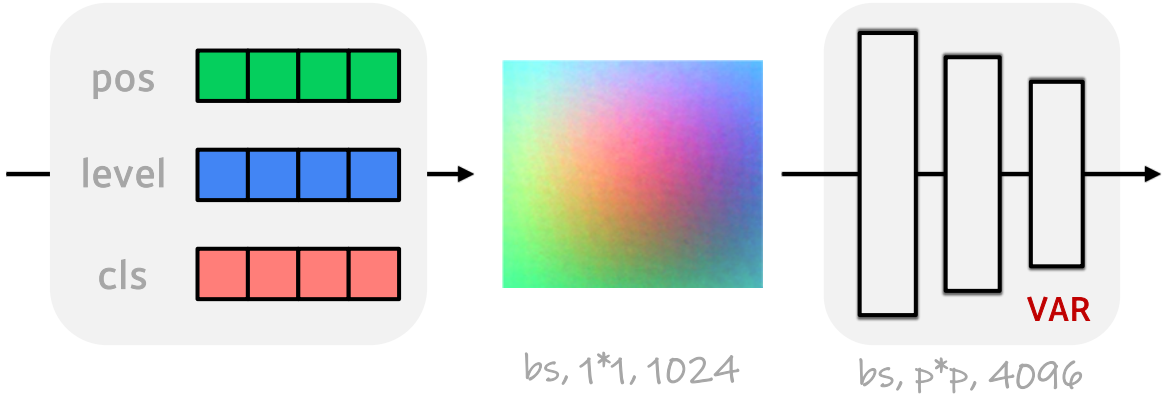


# Code | VAR Inference

NAVER AI LAB

```python
def var_inference(class_label, patch_nums):
    batch_size = class_label.shape[0]
    L = sum(pn * pn for pn in patch_nums)
    embed_dim = 1024
    depth = 16

    # pretrained-vqvae
    codebook_num = 4096
    codebook_dim = 32
    vqvae_codebook = nn.Embedding(codebook_num, codebook_dim)

    # pre embedding
    class_emb = torch.randn([batch_size * 2, 1, embed_dim]) # class_embedding(class_label) + pos_start
    pos_emb = torch.randn([batch_size, L, embed_dim])
    level_emb = torch.randn([batch_size, L, embed_dim])

    """ Input """
    token_emb = class_emb + pos_emb[:, :1] + level_emb[:, :1] # [bs * 2, 1, embed_dim] -> start_token
    latent = torch.zeros([batch_size, 32, 16, 16])

    """ Insert condition """
    last_stage_index = len(patch_nums) - 1
    current_position = 0
    for stage_index, pn in enumerate(patch_nums):
        stage_ratio = stage_index / last_stage_index
        current_position += pn * pn

        for i in range(depth):
            token_emb = token_block(token_emb, class_emb)

        logit = get_logit(token_emb, class_emb)

    # classfier-free guidance
    cfg_scale = 5
    t = cfg_scale * stage_ratio
    logit = (1 + t) * logit[:batch_size] - t * logit[batch_size:]

    # sampling
    sampled_logit = sampling_top_k_p(logit, top_k=900, top_p=0.96, num_samples=1)[:, :, 0]

    # get token embedding
    token_emb = vqvae_codebook(sampled_logit)
    token_emb = token_emb.transpose(1, 2).view(batch_size, codebook_dim, pn, pn)

    # get next token embedding
    latent, next_token_emb = get_next_autoregressive_input(stage_index, patch_nums, latent, token_emb)

    if stage_index != last_stage_index:
        next_token_emb = next_token_emb.view(batch_size, codebook_dim, -1).transpose(1, 2)

        next_position = current_position + patch_nums[stage_index + 1] ** 2

        # [bs, next_patch*next_patch, embed_dim]
        next_token_map = token_linear(next_token_emb, embed_dim) + level_emb[:, current_position: next_position]

        token_emb = next_token_map.repeat(2, 1, 1)

""" Output: Generated image """
img = vqvae_decode(latent)

return img
```

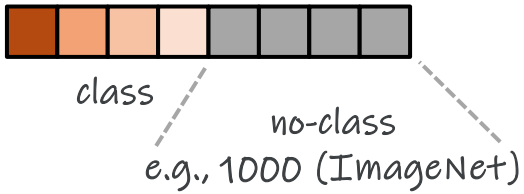


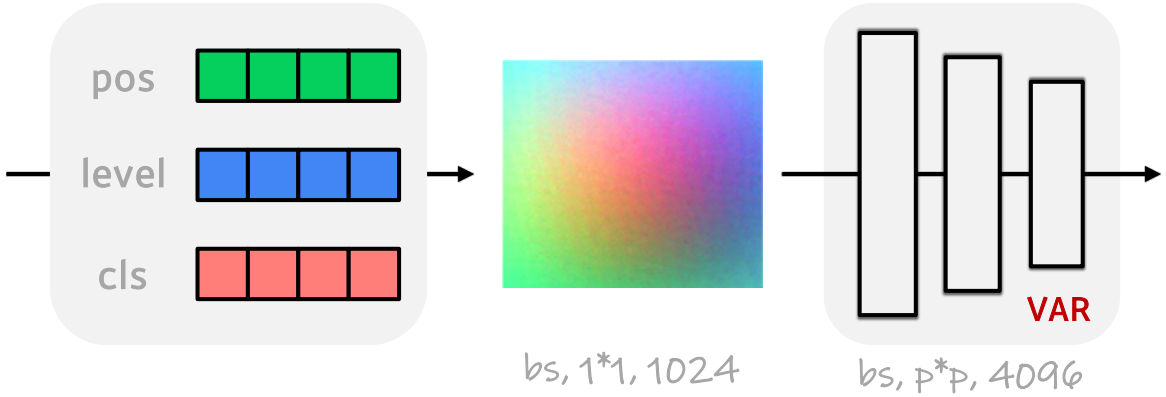



# Code | VAR Inference

NAVER AI LAB

```python
def var_inference(class_label, patch_nums):
    batch_size = class_label.shape[0]
    L = sum(pn * pn for pn in patch_nums)
    embed_dim = 1024
    depth = 16

    # pretrained-vqvae
    codebook_num = 4096
    codebook_dim = 32
    vqvae_codebook = nn.Embedding(codebook_num, codebook_dim)

    # pre embedding
    class_emb = torch.randn([batch_size * 2, 1, embed_dim]) # class_embedding(class_label) + pos_start
    pos_emb = torch.randn([batch_size, L, embed_dim])
    level_emb = torch.randn([batch_size, L, embed_dim])

    """ Input """
    token_emb = class_emb + pos_emb[:, :1] + level_emb[:, :1] # [bs * 2, 1, embed_dim] -> start_token
    latent = torch.zeros([batch_size, 32, 16, 16])

    """ Insert condition """
    last_stage_index = len(patch_nums) - 1
    current_position = 0
    for stage_index, pn in enumerate(patch_nums):
        stage_ratio = stage_index / last_stage_index
        current_position += pn * pn

        for i in range(depth):
            token_emb = token_block(token_emb, class_emb)

        logit = get_logit(token_emb, class_emb)

    # classfier-free guidance
    cfg_scale = 5
    t = cfg_scale * stage_ratio
    logit = (1 + t) * logit[:batch_size] - t * logit[batch_size:]

    # sampling
    sampled_logit = sampling_top_k_p(logit, top_k=900, top_p=0.96, num_samples=1)[:, :, 0]

    # get token embedding
    token_emb = vqvae_codebook(sampled_logit)
    token_emb = token_emb.transpose(1, 2).view(batch_size, codebook_dim, pn, pn)

    # get next token embedding
    latent, next_token_emb = get_next_autoregressive_input(stage_index, patch_nums, latent, token_emb)

    if stage_index != last_stage_index:
        next_token_emb = next_token_emb.view(batch_size, codebook_dim, -1).transpose(1, 2)

        next_position = current_position + patch_nums[stage_index + 1] ** 2

        # [bs, next_patch*next_patch, embed_dim]
        next_token_map = token_linear(next_token_emb, embed_dim) + level_emb[:, current_position: next_position]

        token_emb = next_token_map.repeat(2, 1, 1)

    """ Output: Generated image """
    img = vqvae_decode(latent)

    return img
```



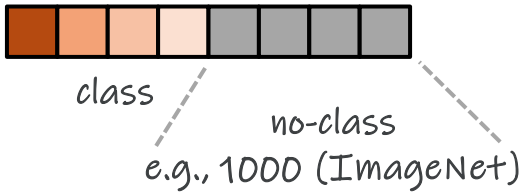



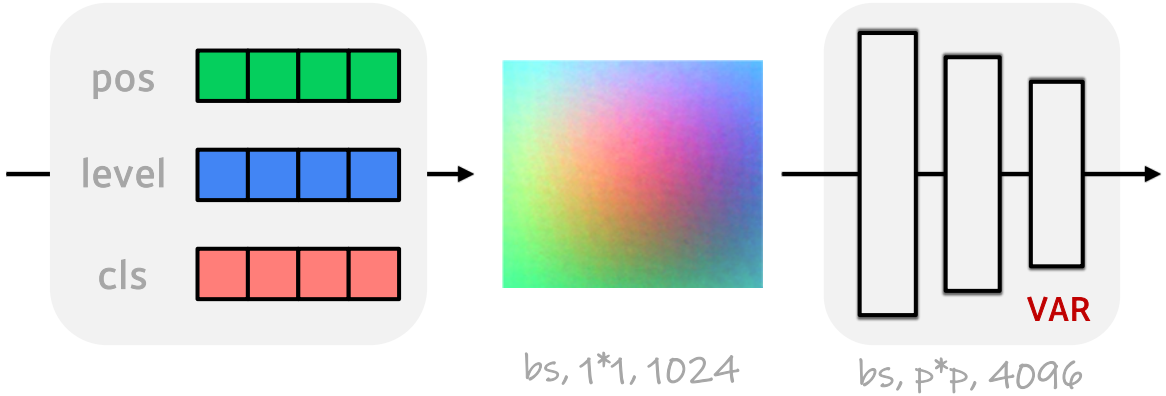
# Code | VAR Inference

NAVER AI LAB

```
def var_inference(class_label, patch_nums):
    batch_size = class_label.shape[0]
    L = sum(pn * pn for pn in patch_nums)
    embed_dim = 1024
    depth = 16

    # pretrained-vqvae
    codebook_num = 4096
    codebook_dim = 32
    vqvae_codebook = nn.Embedding(codebook_num, codebook_dim)

    # pre embedding
    class_emb = torch.randn([batch_size * 2, 1, embed_dim]) # class_embedding(class_label) + pos_start
    pos_emb = torch.randn([batch_size, L, embed_dim])
    level_emb = torch.randn([batch_size, L, embed_dim])

    """ Input """
    token_emb = class_emb + pos_emb[:, :1] + level_emb[:, :1] # [bs * 2, 1, embed_dim] -> start_token
    latent = torch.zeros([batch_size, 32, 16, 16])

    """ Insert condition """
    last_stage_index = len(patch_nums) - 1
    current_position = 0
    for stage_index, pn in enumerate(patch_nums):
        stage_ratio = stage_index / last_stage_index
        current_position += pn * pn

        for i in range(depth):
            token_emb = token_block(token_emb, class_emb)

        logit = get_logit(token_emb, class_emb)

        # classfier-free guidance
        cfg_scale = 5
        t = cfg_scale * stage_ratio
        logit = (1 + t) * logit[:batch_size] - t * logit[batch_size:]

        # sampling
        sampled_logit = sampling_top_k_p(logit, top_k=900, top_p=0.96, num_samples=1)[:, :, 0]

        # get token embedding
        token_emb = vqvae_codebook(sampled_logit)
        token_emb = token_emb.transpose(1, 2).view(batch_size, codebook_dim, pn, pn)

        # get next token embedding
        latent, next_token_emb = get_next_autoregressive_input(stage_index, patch_nums, latent, token_emb)

        if stage_index != last_stage_index:
            next_token_emb = next_token_emb.view(batch_size, codebook_dim, -1).transpose(1, 2)

            next_position = current_position + patch_nums[stage_index + 1] ** 2

            # [bs, next_patch*next_patch, embed_dim]
            next_token_map = token_linear(next_token_emb, embed_dim) + level_emb[:, current_position: next_position]

            token_emb = next_token_map.repeat(2, 1, 1)

    """ Output: Generated image """
    img = vqvae_decode(latent)

    return img
```

for diversity



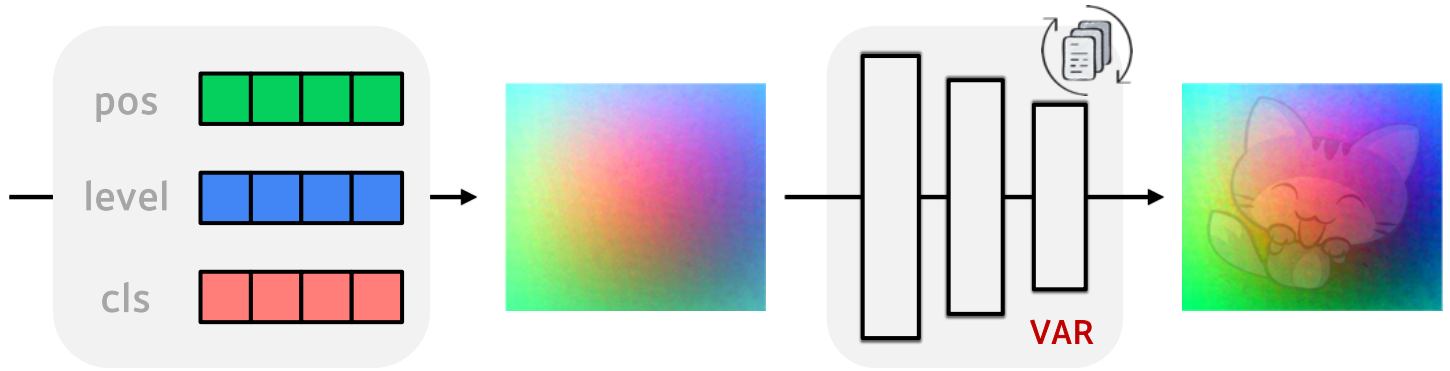
# Code | VAR Inference

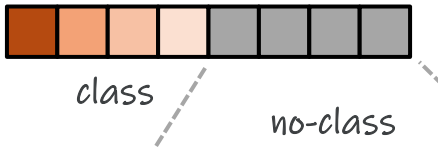
NAVER AI LAB

```python
def var_inference(class_label, patch_nums):
    batch_size = class_label.shape[0]
    L = sum(pn * pn for pn in patch_nums)
    embed_dim = 1024
    depth = 16

    # pretrained-vqvae
    codebook_num = 4096
    codebook_dim = 32
    vqvae_codebook = nn.Embedding(codebook_num, codebook_dim)

    # pre embedding
    class_emb = torch.randn([batch_size * 2, 1, embed_dim]) # class_embedding(class_label) + pos_start
    pos_emb = torch.randn([batch_size, L, embed_dim])
    level_emb = torch.randn([batch_size, L, embed_dim])

    """ Input """
    token_emb = class_emb + pos_emb[:, :1] + level_emb[:, :1] # [bs * 2, 1, embed_dim] -> start_token
    latent = torch.zeros([batch_size, 32, 16, 16])

    """ Insert condition """
    last_stage_index = len(patch_nums) - 1
    current_position = 0
    for stage_index, pn in enumerate(patch_nums):
        stage_ratio = stage_index / last_stage_index
        current_position += pn * pn

        for i in range(depth):
            token_emb = token_block(token_emb, class_emb)

        logit = get_logit(token_emb, class_emb)
```

for diversity

```python
    # classfier-free guidance
    cfg_scale = 5
    t = cfg_scale * stage_ratio
    logit = (1 + t) * logit[:batch_size] - t * logit[batch_size:]

    # sampling
    sampled_logit = sampling_top_k_p(logit, top_k=900, top_p=0.96, num_samples=1)[:, :, 0]

    # get token embedding
    token_emb = vqvae_codebook(sampled_logit)
    token_emb = token_emb.transpose(1, 2).view(batch_size, codebook_dim, pn, pn)

    # get next token embedding
    latent, next_token_emb = get_next_autoregressive_input(stage_index, patch_nums, latent, token_emb)

    if stage_index != last_stage_index:
        next_token_emb = next_token_emb.view(batch_size, codebook_dim, -1).transpose(1, 2)

        next_position = current_position + patch_nums[stage_index + 1] ** 2

        # [bs, next_patch*next_patch, embed_dim]
        next_token_map = token_linear(next_token_emb, embed_dim) + level_emb[:, current_position: next_position]

        token_emb = next_token_map.repeat(2, 1, 1)

"""" Output: Generated image """"
img = vqvae_decode(latent)

return img
```



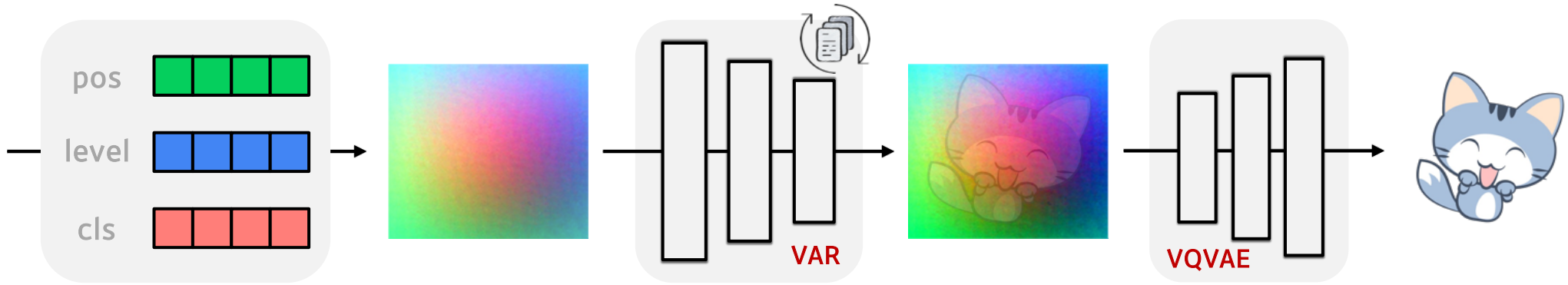

```python
def get_next_autoregressive_input(stage_index, patch_nums, latent, token) :
    # latent = f_hat
    # token = zk

    origin_size = patch_nums[-1]

    if stage_index != len(patch_nums)-1:
        token = resize(token, size=origin_size, mode='bicubic')

        h = phi_conv(token)   # conv after upsample
        latent = latent + h

        next_token = resize(latent, size=patch_nums[stage_index + 1], mode='area') # 이건 확대인데 area네...

        return latent, next_token
    else:
        h = phi_conv(token)
        latent = latent + h

        return latent, latent
```

1x1 -> 2x2 -> 3x3 -> ... -> 16x16

# Code | VAR Inference

NAVER AI LAB

```python
def var_inference(class_label, patch_nums):
    batch_size = class_label.shape[0]
    L = sum(pn * pn for pn in patch_nums)
    embed_dim = 1024
    depth = 16

    # pretrained-vqvae
    codebook_num = 4096
    codebook_dim = 32
    vqvae_codebook = nn.Embedding(codebook_num, codebook_dim)

    # pre embedding
    class_emb = torch.randn([batch_size * 2, 1, embed_dim]) # class_embedding(class_label) + pos_start
    pos_emb = torch.randn([batch_size, L, embed_dim])
    level_emb = torch.randn([batch_size, L, embed_dim])

    """ Input """
    token_emb = class_emb + pos_emb[:, :1] + level_emb[:, :1] # [bs * 2, 1, embed_dim] -> start_token
    latent = torch.zeros([batch_size, 32, 16, 16])

    """ Insert condition """
    last_stage_index = len(patch_nums) - 1
    current_position = 0
    for stage_index, pn in enumerate(patch_nums):
        stage_ratio = stage_index / last_stage_index
        current_position += pn * pn

        for i in range(depth):
            token_emb = token_block(token_emb, class_emb)

        logit = get_logit(token_emb, class_emb)

        # classfier-free guidance
        cfg_scale = 5
        t = cfg_scale * stage_ratio
        logit = (1 + t) * logit[:batch_size] - t * logit[batch_size:]

        # sampling
        sampled_logit = sampling_top_k_p(logit, top_k=900, top_p=0.96, num_samples=1)[:, :, 0]

        # get token embedding
        token_emb = vqvae_codebook(sampled_logit)
        token_emb = token_emb.transpose(1, 2).view(batch_size, codebook_dim, pn, pn)

        # get next token embedding
        latent, next_token_emb = get_next_autoregressive_input(stage_index, patch_nums, latent, token_emb)

        if stage_index != last_stage_index:
            next_token_emb = next_token_emb.view(batch_size, codebook_dim, -1).transpose(1, 2)

            next_position = current_position + patch_nums[stage_index + 1] ** 2

            # [bs, next_patch*next_patch, embed_dim]
            next_token_map = token_linear(next_token_emb, embed_dim) + level_emb[:, current_position: next_position]

            token_emb = next_token_map.repeat(2, 1, 1)

    """ Output: Generated image """
    img = vqvae_decode(latent)

    return img
```



# Results | VAR

NAVER AI LAB

Table 1: **Generative model family comparison on class-conditional ImageNet 256×256**. "↓" or "↑" indicate lower or higher values are better. Metrics include Fréchet inception distance (FID), inception score (IS), precision (Pre) and recall (rec). "#Step": the number of model runs needed to generate an image. Wall-clock inference time relative to VAR is reported. Models with the suffix "-re" used rejection sampling. †: taken from MaskGIT [11].

| Type | Model | FID↓ | IS↑ | Pre↑ | Rec↑ | #Para | #Step | Time |
|---|---|---|---|---|---|---|---|---|
| GAN | BigGAN [7] | 6.95 | 224.5 | **0.89** | 0.38 | 112M | 1 | – |
| GAN | GigaGAN [29] | 3.45 | 225.5 | 0.84 | **0.61** | 569M | 1 | – |
| GAN | StyleGan-XL [57] | 2.30 | 265.1 | 0.78 | 0.53 | 166M | 1 | 0.3 [57] |
| Diff. | ADM [16] | 10.94 | 101.0 | 0.69 | 0.63 | 554M | 250 | 168 [57] |
| Diff. | CDM [25] | 4.88 | 158.7 | – | – | – | 8100 | – |
| Diff. | LDM-4-G [53] | 3.60 | 247.7 | – | – | 400M | 250 | – |
| Diff. | DiT-L/2 [46] | 5.02 | 167.2 | 0.75 | 0.57 | 458M | 250 | 31 |
| Diff. | DiT-XL/2 [46] | 2.27 | 278.2 | 0.83 | 0.57 | 675M | 250 | 45 |
| Diff. | L-DiT-3B [2] | 2.10 | 304.4 | 0.82 | 0.60 | 3.0B | 250 | >45 |
| Diff. | L-DiT-7B [2] | 2.28 | 316.2 | 0.83 | 0.58 | 7.0B | 250 | >45 |
| Mask. | MaskGIT [11] | 6.18 | 182.1 | 0.80 | 0.51 | 227M | 8 | 0.5 [11] |
| Mask. | MaskGIT-re [11] | 4.02 | 355.6 | – | – | 227M | 8 | 0.5 [11] |
| Mask. | RCG (cond.) [38] | 3.49 | 215.5 | – | – | 502M | 20 | 1.9 [38] |
| AR | VQVAE-2† [52] | 31.11 | ∼45 | 0.36 | 0.57 | 13.5B | 5120 | – |
| AR | VQGAN† [19] | 18.65 | 80.4 | 0.78 | 0.26 | 227M | 256 | 19 [11] |
| AR | VQGAN [19] | 15.78 | 74.3 | – | – | 1.4B | 256 | 24 |
| AR | VQGAN-re [19] | 5.20 | 280.3 | – | – | 1.4B | 256 | 24 |
| AR | ViTVQ [71] | 4.17 | 175.1 | – | – | 1.7B | 1024 | >24 |
| AR | ViTVQ-re [71] | 3.04 | 227.4 | – | – | 1.7B | 1024 | >24 |
| AR | RQTran. [37] | 7.55 | 134.0 | – | – | 3.8B | 68 | 21 |
| AR | RQTran.-re [37] | 3.80 | 323.7 | – | – | 3.8B | 68 | 21 |
| VAR | VAR-*d*16 | 3.60 | 257.5 | 0.85 | 0.48 | 310M | 10 | 0.4 |
| VAR | VAR-*d*20 | 2.95 | 306.1 | 0.84 | 0.53 | 600M | 10 | 0.5 |
| VAR | VAR-*d*24 | 2.33 | 320.1 | 0.82 | 0.57 | 1.0B | 10 | 0.6 |
| VAR | VAR-*d*30 | 1.97 | 334.7 | 0.81 | **0.61** | 2.0B | 10 | 1 |
| VAR | VAR-*d*30-re | **1.80** | **356.4** | 0.83 | 0.57 | 2.0B | 10 | 1 |
| | (validation data) | 1.78 | 236.9 | 0.75 | 0.67 | | | |

Table 2: **ImageNet 512×512 conditional generation.** †: quoted from MaskGIT [11]. "-s": a single shared AdaLN layer is used due to resource limitation.

| Type | Model | FID↓ | IS↑ | Time |
|---|---|---|---|---|
| GAN | BigGAN [7] | 8.43 | 177.9 | – |
| Diff. | ADM [16] | 23.24 | 101.0 | – |
| Diff. | DiT-XL/2 [46] | 3.04 | 240.8 | 81 |
| Mask. | MaskGIT [11] | 7.32 | 156.0 | 0.5 |
| AR | VQGAN† [19] | 26.52 | 66.8 | 25 |
| VAR | VAR-*d*36-s | **2.63** | **303.2** | 1 |

Table 3: **Ablation study of VAR.** The first two rows compare GPT-2-style transformers trained with AR or VAR algorithm. Subsequent lines show the influence of VAR enhancements. "AdaLN": adaptive layernorm. "CFG": classifier-free guidance. "Cost": inference cost relative to the baseline. "Δ": reduction in FID.

| | Description | Para. | Model | AdaLN | Top-$k$ | CFG | Cost | FID↓ | Δ |
|---|---|---|---|---|---|---|---|---|---|
| 1 | AR [11] | 227M | AR | ✗ | ✗ | ✗ | 1 | 18.65 | 0.00 |
| 2 | AR to VAR | 207M | VAR-*d*16 | ✗ | ✗ | ✗ | 0.013 | 5.22 | −13.43 |
| 3 | +AdaLN | 310M | VAR-*d*16 | ✓ | ✗ | ✗ | 0.016 | 4.95 | −13.70 |
| 4 | +Top-$k$ | 310M | VAR-*d*16 | ✓ | 600 | ✗ | 0.016 | 4.64 | −14.01 |
| 5 | +CFG | 310M | VAR-*d*16 | ✓ | 600 | 2.0 | 0.022 | 3.60 | −15.05 |
| 6 | +Scale up | 2.0B | VAR-*d*30 | ✓ | 600 | 2.0 | 0.052 | 1.80 | −16.85 |

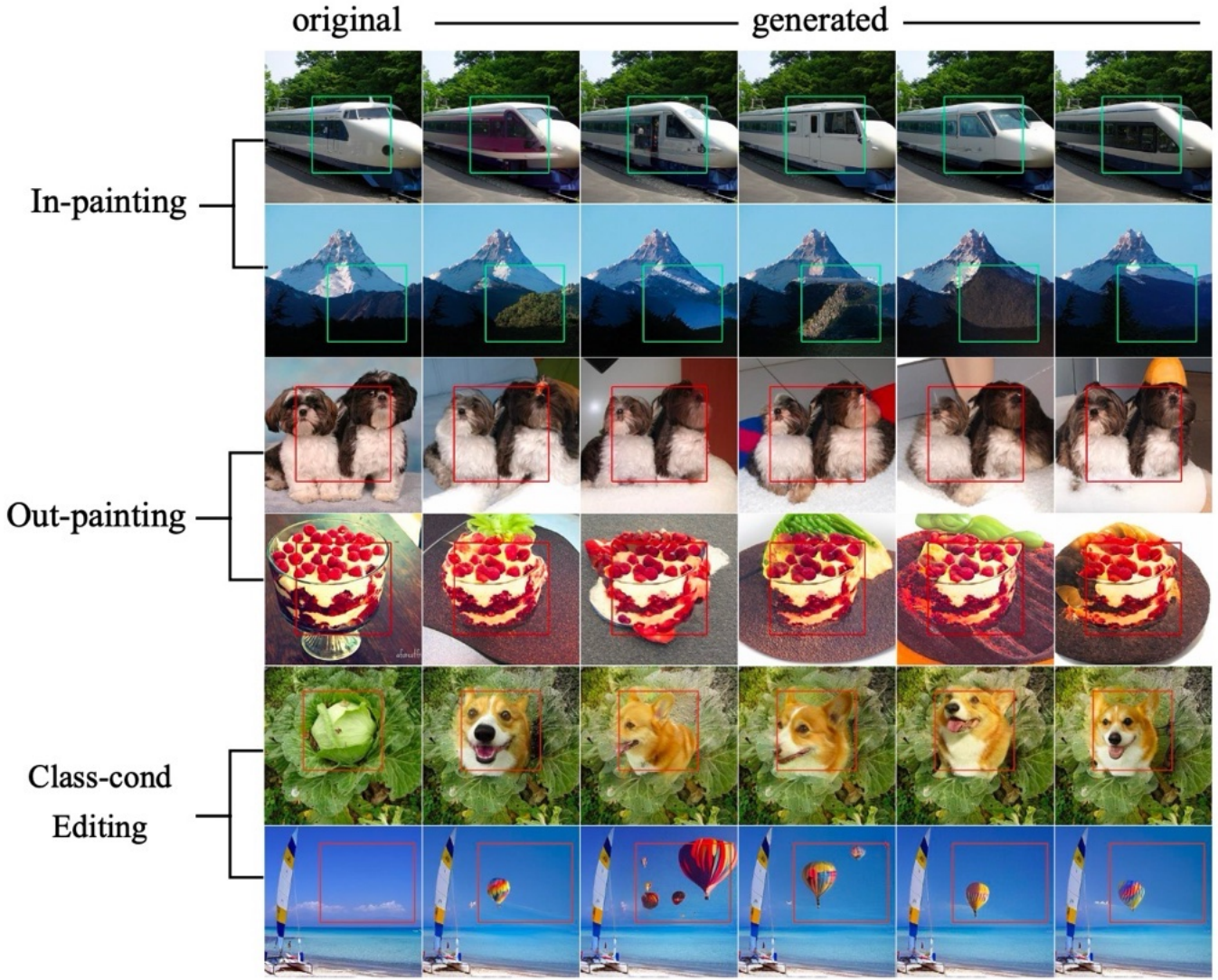



Figure 8: **Zero-shot evaluation in downstream tasks** containing in-painting, out-painting, and class-conditional editing. The results show that VAR can generalize to novel downstream tasks without special design and finetuning. Zoom in for a better view.

# Results | VAR-Related works

NAVER AI LAB

## Infinity∞: Scaling Bitwise AutoRegressive Modeling for High-Resolution Image Synthesis

Jian Han*, Jinlai Liu*, Yi Jiang*, Bin Yan
Yuqi Zhang, Zehuan Yuan†, Bingyue Peng, Xiaobing Liu
ByteDance
{hanjian.thu123,liujinlai.licio,jiangyi.enjoy,bin.yan}@bytedance.com,
{zhangyuqi.hi,yuanzehuan,bingyue.peng,will.liu}@bytedance.com,

Codes and models: https://github.com/FoundationVision/Infinity

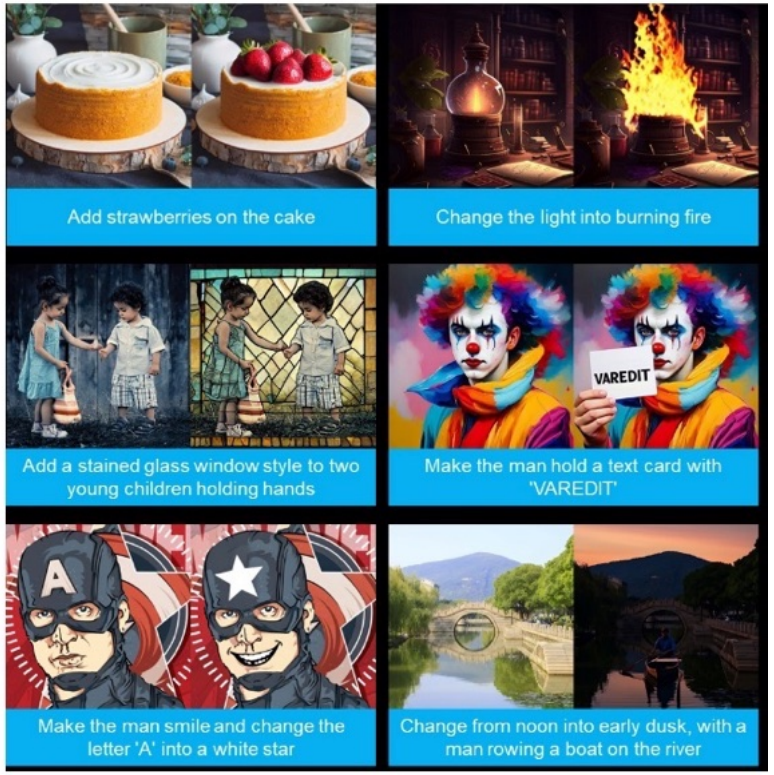

Figure 1: **High-resolution image synthesis results from Infinity**, showcasing its capabilities in precise prompt following, spatial reasoning, text rendering, and aesthetics across different styles and aspect ratios.

## HART: Efficient Visual Generation with Hybrid Autoregressive Transformer

Haotian Tang[1*] Yecheng Wu[1,3*] Shang Yang[1] Enze Xie[2] Junsong Chen[2]
Junyu Chen[1,3] Zhuoyang Zhang[1] Han Cai[2] Yao Lu[2] Song Han[1,2]
MIT[1] NVIDIA[2] Tsinghua University[3]
https://hanlab.mit.edu/projects/hart


Figure 1: **HART** is an early autoregressive model that can directly generate **1024×1024** images with quality comparable to diffusion models, while offering significantly improved efficiency. It achieves **4.5-7.7**× higher throughput, **3.1-5.9**× lower latency (measured on A100), and **6.9-13.4**× lower MACs compared to state-of-the-art diffusion models. Check out our online demo and video.

## VAREdit



# Audience Q&A


https://naver-career.gitbook.io/en/positions/ai-ml/generation-research

Thank you for your attention !