# AI Applications Lecture 13

Image Generation AI
Reverse Diffusion Process

SUZUKI, Atsushi
Jing WANG

# Outline

# Introduction

In the previous lecture, we learned that a **natural image decoder**, such as those realized by **Variational Autoencoders (VAE)** [6, 7], can construct high-resolution **natural images** from low-resolution **latent images**. Specifically, we reconfirmed that the VAE decoder

$$\text{Dec}_\gamma : \mathcal{Z} \to \mathcal{I}$$

can output high-resolution images in the pixel space $\mathcal{I} \subset \mathbb{R}^{H \times W \times C}$ given inputs from the latent space $\mathcal{Z}$.

# 1. Introduction

Upon completion of this lecture, students should be able to explain and execute the following:

- Explain **what the inputs and outputs** of the **reverse diffusion process** are in the text-to-image pipeline.
- Explain the **conditions that the neural network** constituting the reverse diffusion process must **satisfy**.
- Explain how the **output of the text encoder** is handled in the reverse diffusion process and how it **controls generation**.
- Explain the differences between **denoising schedulers** and be able to appropriately select and **execute** the reverse diffusion process according to the situation.

# Preparation: Mathematical Notations

# 2. Preparation: Mathematical Notations

- **Definition:**
  - $(\mathrm{LHS}) \coloneqq (\mathrm{RHS})$: Indicates that the left-hand side is defined by the right-hand side. For example, $a \coloneqq b$ indicates that $a$ is defined as $b$.
- **Set:**
  - Sets are often denoted by uppercase calligraphic letters. Example: $\mathcal{A}$.
  - $x \in \mathcal{A}$: Indicates that the element $x$ belongs to the set $\mathcal{A}$.
  - $\{\}$: The empty set.
  - $\{a, b, c\}$: The set consisting of elements $a, b, c$.

# Preparation: Mathematical Notations (2)

- $\{x \in \mathcal{A} \mid P(x)\}$: The set of elements in $\mathcal{A}$ for which the proposition $P(x)$ is true (set-builder notation, intension).
- $|\mathcal{A}|$: The number of elements in the set $\mathcal{A}$ (used only for finite sets in this lecture).
- $\mathbb{R}$: The set of all real numbers. $\mathbb{R}_{>0}$, $\mathbb{R}_{\geq 0}$, etc., are defined similarly.
- $\mathbb{Z}$: The set of all integers. $\mathbb{Z}_{>0}$, $\mathbb{Z}_{\geq 0}$, etc., are defined similarly.
- $[1, k]_{\mathbb{Z}}$: For $k \in \mathbb{Z}_{>0} \cup \{+\infty\}$, if $k < +\infty$ then $\{1, \ldots, k\}$, if $k = +\infty$ then $\mathbb{Z}_{>0}$.

# Preparation: Mathematical Notations (3)

- **Function:**
  - $f : \mathcal{X} \to \mathcal{Y}$ denotes that $f$ is a map.
  - $y = f(x)$ denotes the output $y \in \mathcal{Y}$ for the input $x \in \mathcal{X}$.
- **Vector:**
  - Vectors are denoted by bold italic lowercase letters. Example: $\boldsymbol{v}$. $\boldsymbol{v} \in \mathbb{R}^n$.
  - The $i$-th component is written as $v_i$:

$$\boldsymbol{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}.$$

  - Standard inner product:

$$\langle \boldsymbol{u}, \boldsymbol{v} \rangle \coloneqq \sum_{i=1}^{d_{\mathrm{emb}}} u_i v_i.$$

# Preparation: Mathematical Notations (4)

- **Sequence:**
  - $\boldsymbol{a} : [1, n]_{\mathbb{Z}} \to \mathcal{A}$ is called a sequence of length $n$. If $n < +\infty$, $\boldsymbol{a} = (a_1, \ldots, a_n)$; if $n = +\infty$, $\boldsymbol{a} = (a_1, a_2, \ldots)$.
  - The length is written as $|\boldsymbol{a}|$.
- **Matrix:**
  - Matrices are denoted by bold italic uppercase letters. Example: $\boldsymbol{A} \in \mathbb{R}^{m,n}$.
  - The elements are $a_{i,j}$, and

$$\boldsymbol{A} = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix}.$$

# Preparation: Mathematical Notations (5)

- Transpose $\boldsymbol{A}^\top \in \mathbb{R}^{n,m}$:

$$\boldsymbol{A}^\top = \begin{bmatrix} a_{1,1} & \cdots & a_{m,1} \\ \vdots & \ddots & \vdots \\ a_{1,n} & \cdots & a_{m,n} \end{bmatrix}.$$

- The transpose of a vector is

$$\boldsymbol{v}^\top = \begin{bmatrix} v_1 & \cdots & v_n \end{bmatrix}.$$

- **Tensor:**
  - A tensor as a multi-dimensional array is denoted by an underlined bold italic uppercase letter $\underline{\boldsymbol{A}}$.

# Revisiting the Goal of the Low-Resolution Latent Image Generator

# 3. Revisiting the Goal of the Low-Resolution Latent Image Generator

The goal of the latent generator $\text{LatentGen}_{\boldsymbol{\beta}}$ is as follows:

- To receive the text encoder output $\left(\boldsymbol{c}^{[i]}\right)_{i=1}^{m}$ and generate a **low-"resolution" latent image** $\boldsymbol{x} \in \mathcal{X}$ to be passed to the **natural image decoder**. The decoder $\text{Dec}_{\boldsymbol{\gamma}}$ can then convert this $\boldsymbol{x}$ into a high-resolution natural image.
- Even for the same $\left(\boldsymbol{c}^{[i]}\right)_{i=1}^{m}$ obtained from the same prompt, to be able to output a **diverse** set of candidate images by changing the **random seed** (pseudo-random number). Since text-to-image tasks tend to lack sufficient input information, **diversity** in the output is essential.

# Motivation for Introducing the Reverse Diffusion Process

# 4. Motivation for Introducing the Reverse Diffusion Process

Given a text condition $\left(\boldsymbol{c}^{[i]}\right)_{i=1}^{m}$, let $\mathrm{P}_{\left(\boldsymbol{c}^{[i]}\right)_{i=1}^{m}}$ denote the distribution that covers the corresponding **set of images** with high probability. The goal is to obtain an algorithm that can pseudo-**sample**/**simulate** $\boldsymbol{x}$ such that

$$\boldsymbol{x} \sim \mathrm{P}_{\left(\boldsymbol{c}^{[i]}\right)_{i=1}^{m}}. \tag{1}$$

What is important is **data generation** itself, not explicitly obtaining the **formula of the distribution** (probability mass function/probability density function).

# Clarification of the Goal

Here, it should be noted that the goal, although often confused, is not to obtain a specific representation of $\mathrm{P}_{(\boldsymbol{c}^{[i]})_{i=1}^{m}}$ by its probability density function or probability mass function. Even if we obtain the representation of the probability mass function or probability density function, it does not mean we can immediately sample data according to that distribution. We need to re-recognize that our goal is data generation, not the representation of the distribution. Also, representing the probability mass function or probability density function is difficult.

# Why Representation is Difficult (1)

For example, if we try to represent the support as a discrete object using a probability mass function, there are $256^{C \times W \times H}$ possible values, making it practically impossible to specify them.

# Why Representation is Difficult (2)

If we try to represent the support as a continuous object using a probability density function, it is impossible naively due to infinite degrees of freedom. If we try to represent it directly with a parametric function, practically the only useful parametric probability model on a multidimensional vector space is the multivariate normal distribution.

# Why Representation is Difficult (3)

It is also difficult to construct a probability mass function or probability density function using a neural network. This is also often confused; neural networks are suitable for representing complex functions, but that is when there are no constraints on the function. Representing a function that satisfies the condition of being a probability, i.e., summing to 1, is not easy, regardless of whether it's a neural network. In natural language processing, softmax was used to construct a probability mass function, but this was possible because the number of elements in the support was not that large. In the case of a probability density function, construction by softmax is itself impossible.

## 4.1 Framework of Pushforward Measure

For the reasons stated in the previous section, instead of directly constructing the probability mass function or probability density function, we take the method of transforming random numbers using a function, thereby implicitly sampling from a **pushforward measure**. Let $\mathcal{S}$ be the support, which has the same dimension as the output, and let $\lambda$ be the base distribution on it (in practice, the standard multivariate normal distribution). We find a **measurable function**

$$f_{(\cdot)} : \left( \prod_{i=1}^{m} \mathbb{R}^{d^{(i)}} \right) \times \mathcal{S} \to \mathcal{I} \tag{2}$$

and define the **pushforward** as

$$f\left( \left( \boldsymbol{c}^{[i]} \right)_{i=1}^{m}, \cdot \right) \sharp \lambda(A) \coloneqq \lambda\left( \left\{ \boldsymbol{\epsilon} \in \mathcal{S} \;\middle|\; f\left( \left( \boldsymbol{c}^{[i]} \right)_{i=1}^{m}, \boldsymbol{z} \right) \in A \right\} \right), \quad A \subset \mathcal{I}. \tag{3}$$

# Target Condition via Pushforward

The target condition is

$$f\left(\left(\boldsymbol{c}^{[i]}\right)_{i=1}^{m}, \cdot\right) \sharp \lambda \approx \mathrm{P}_{\left(\boldsymbol{c}^{[i]}\right)_{i=1}^{m}}. \tag{4}$$

If this is achieved, we can realize (1) simply by generating $\boldsymbol{z} \sim \lambda$ with pseudo-random numbers and setting

$$\boldsymbol{x} = f\left(\left(\boldsymbol{c}^{[i]}\right)_{i=1}^{m}, \boldsymbol{z}\right). \tag{5}$$

## 4.2 Distribution Approximation is Not Input-Output Correspondence Learning

What is given is the **distribution on the output side**, and the **correct answer** for the input $z$ is not uniquely determined. For example, with Gaussian input, composing a rotation centered at the origin in the input space leaves the output distribution invariant. To put it more formally,

# Distribution Approximation is Not Input-Output Learning (2)

Suppose the pushforward distribution $f\sharp\lambda_{\mathrm{normal}}$ of the standard Gaussian distribution $\lambda_{\mathrm{normal}}$ by some bijective map $f : \mathcal{S} \to \mathcal{S}$ is ideal. In this case, an output $\boldsymbol{x} \in \mathcal{S}$ corresponds to $\boldsymbol{z} = f^{-1}(\boldsymbol{x})$. However, if we consider a rotation map $\mathrm{Rot}$ centered at the origin, then

$$(f \circ \mathrm{Rot})\sharp\lambda_{\mathrm{normal}} = f\sharp\lambda_{\mathrm{normal}},$$

so it is also ideal. In this case, $\boldsymbol{x}$ corresponds to

$$\boldsymbol{z}' = \mathrm{Rot}^{-1}\big(f^{-1}(\boldsymbol{x})\big).$$

We cannot say that $\boldsymbol{z}$ is the correct input, nor that $\boldsymbol{z}'$ is the correct input. Therefore, it is not a simple “input-output correspondence regression problem”.

## Consequence for Learning

Therefore, if we want to formulate it as learning an input-output relationship, we need to determine the "input" $\boldsymbol{z}^{(i)}$ for each output $\boldsymbol{x}^{(i)}$, which is not uniquely determined in principle, and then learn the input-output relationship.

## Consequence for Learning

Therefore, if we want to formulate it as learning an input-output relationship, we need to determine the “input” $\boldsymbol{z}^{(i)}$ for each output $\boldsymbol{x}^{(i)}$, which is not uniquely determined in principle, and then learn the input-output relationship.

Although it might seem mathematically simpler due to the freedom in determining the input, if the determination of the “input” is poor, the input-output relationship becomes complicated, and a slight change in the input (inputting values not used during training is essential for achieving diverse outputs, so this case must always be considered) can lead to a large change in the output, resulting in an unstable distribution unsuitable for practical use. Therefore, it becomes a difficult problem of not just learning the input-output relationship, but also determining the “input” appropriately in some sense. In fact, many distribution generation problems in neural networks determine the “input” in some sense (e.g., autoencoders).

# 4.3 Difficulty of Naive Likelihood Maximization and Flow-based Models

Now, since we want to select the optimal function on a computer, we use a parametric function $f_{(\cdot)}$. If we try to achieve (4) using this parametric function and optimize the log-likelihood directly with a data sequence $\{\boldsymbol{x}^{(n)}\}_{n=1}^{N}$, we need the **inverse map** $f_{\boldsymbol{\theta}}^{-1}$ and the **Jacobian determinant** (change of variables rule for invertible transformations).

# Flow-based Likelihood (2)

Here, suppose the observed data $\boldsymbol{x}^{(n)} \in \mathbb{R}^d$ are independent and identically distributed, and are mapped to the base distribution $p_Z$ by $z = f_{\boldsymbol{\theta}}^{-1}(x)$. Then the likelihood of each sample is

$$p_X(\boldsymbol{x}^{(n)} \mid \boldsymbol{\theta}) = p_Z\big(f_{\boldsymbol{\theta}}^{-1}(\boldsymbol{x}^{(n)})\big) \left| \det \frac{\partial f_{\boldsymbol{\theta}}^{-1}(\boldsymbol{x}^{(n)})}{\partial \boldsymbol{x}^\top} \right|, \tag{6}$$

so the log-likelihood of all data is

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{n=1}^{N} \log p_X(\boldsymbol{x}^{(n)} \mid \boldsymbol{\theta}) = \sum_{n=1}^{N} \left\{ \log p_Z\big(f_{\boldsymbol{\theta}}^{-1}(\boldsymbol{x}^{(n)})\big) + \log \left| \det \frac{\partial f_{\boldsymbol{\theta}}^{-1}(\boldsymbol{x}^{(n)})}{\partial \boldsymbol{x}^\top} \right| \right\}. \tag{7}$$

# Gradient of Flow-based Likelihood (3)

In this case, the gradient (first derivative with respect to the parameters) is given by

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \sum_{n=1}^{N} \left[ \nabla_{\boldsymbol{\theta}} \log p_Z\left(f_{\boldsymbol{\theta}}^{-1}(\boldsymbol{x}^{(n)})\right) + \nabla_{\boldsymbol{\theta}} \log \left| \det \frac{\partial f_{\boldsymbol{\theta}}^{-1}(\boldsymbol{x}^{(n)})}{\partial \boldsymbol{x}^{\top}} \right| \right], \tag{8}$$

Here, the first term is the derivative of the log-density of the base distribution $p_Z$ through $f_{\boldsymbol{\theta}}^{-1}$, and the second term originates from the Jacobian term of the variable transformation. Therefore, calculating this gradient generally requires the explicit calculation of $f_{\boldsymbol{\theta}}^{-1}$ and its Jacobian determinant.

# Flow-based Models and Limitations (4)

However, calculating $f_{\boldsymbol{\theta}}^{-1}$ for a general neural network is difficult. To avoid this, there are **flow-based** models (RealNVP [1], Glow [5]) that impose constraints of **invertibility and efficiently computable Jacobian**, but this tends to sacrifice **freedom in architecture design**. One of the advantages of neural networks is the flexibility in design, being able to freely design the graph structure according to the application field, so losing that is a pain in practice.

# 4.4 Positioning of the Reverse Diffusion Process

From the above, distribution approximation requires **non-trivial methods**. One of them is the **diffusion model**. Here, we will not go into the details of the **training stage**, but formulate the **reverse diffusion process** as the **inference stage (data generation stage)**. It is called “reverse” because during training, the meaning of the **input** is established through **forward diffusion (diffusion process)**, and during generation, the **reverse** of that is traced [2, 4]. The **denoising scheduler** provides the specific **implementation procedure** for the reverse diffusion process.

# Reverse Diffusion Process: Neural Network Inputs, Outputs, and Conditions

# 5. Reverse Diffusion Process: Neural Network Inputs, Outputs, and Conditions

The reverse diffusion process is an algorithm that achieves the target distribution by repeatedly applying a function that is close to, but slightly different from, the identity function (meaning the input and output do not change much). Each time the function is applied, the input moves slightly. If we view this as a process where the distribution is deformed by the composite function, the initial input Gaussian distribution is slightly deformed by the pushforward each time the function is applied. It is expected that this will eventually approach the target distribution.

# Concrete Construction with a Neural Network

To explain the construction using a neural network more concretely, it is as follows. There are $k = 0, 1, ..., N-1$ and a corresponding monotonically decreasing sequence called timesteps $\tau_0 > \tau_1 > ... > \tau_{N-1}$, and

- At each step, the neural network receives the output $\boldsymbol{x}^{(k)}$ from the previous step, the input $(\boldsymbol{c}^{[i]})_{i=1}^{m}$ from the text encoder, and $\tau_k$, and determines the negative direction $\widehat{\boldsymbol{\epsilon}}^{(k)}$ in which $\boldsymbol{x}^{(k)}$ moves.
- Roughly, the point is moved as $\boldsymbol{x}^{(k+1)} \leftarrow \boldsymbol{x}^{(k)} - \gamma_t \widehat{\boldsymbol{\epsilon}}^{(k)}$. How $\gamma_t$ is determined and the detailed behavior vary depending on the algorithm called the **denoising scheduler**. In some cases, Gaussian noise is further added.

# Definition: Network Type for Reverse Diffusion

**Definition (Input/Output Types of the Network Used in the Reverse Diffusion Process)**

Let $d_{\text{latent}} \in \mathbb{Z}_{>0}$ be the desired latent dimension, $d_{\text{context}} \in \mathbb{Z}_{>0}$ be the text condition dimension, and $\mathcal{T}$ be the set of timesteps. Here $\mathcal{T}$ is the set of **discrete timesteps** generated by the scheduler before inference, given by (16), (33), (**??**) described later. The **noise predictor** (or **data predictor**) $f_{\boldsymbol{\theta}}$ in the reverse diffusion process is

$$f_{\boldsymbol{\theta}} : \mathbb{R}^{d_{\text{latent}}} \times \mathbb{R}^{d_{\text{context}}} \times \mathcal{T} \to \mathbb{R}^{d_{\text{latent}}} \tag{9}$$

and the input is (i) the current sample $\boldsymbol{x}_t \in \mathbb{R}^{d_{\text{latent}}}$, (ii) the text encoder output (including both positive and negative) $\boldsymbol{c} \in \mathbb{R}^{d_{\text{context}}}$, and (iii) the timestep index $t \in \mathcal{T}$. The output is a tensor of the **same dimension** (e.g., **predicted noise** $\widehat{\boldsymbol{\epsilon}}$).

# Definition: Sequential Application of CFG

**Definition (Sequential Application of Classifier-Free Guidance)**

Given $\boldsymbol{x}_t \in \mathbb{R}^{d_{\text{latent}}}$, positive embedding $\boldsymbol{c}_+ \in \mathbb{R}^{d_{\text{context}}}$, negative (unconditional) embedding $\boldsymbol{c}_- \in \mathbb{R}^{d_{\text{context}}}$, and timestep $t \in \mathcal{T}$. The composition of model outputs in CFG is defined as

$$\widehat{\boldsymbol{\epsilon}}_{\text{cond}} \coloneqq f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, \boldsymbol{c}_+, t), \tag{10}$$

$$\widehat{\boldsymbol{\epsilon}}_{\text{uncond}} \coloneqq f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, \boldsymbol{c}_-, t), \tag{11}$$

$$\widehat{\boldsymbol{\epsilon}}_{\text{CFG}} \coloneqq \widehat{\boldsymbol{\epsilon}}_{\text{uncond}} + s\big(\widehat{\boldsymbol{\epsilon}}_{\text{cond}} - \widehat{\boldsymbol{\epsilon}}_{\text{uncond}}\big), \tag{12}$$

where $s \in \mathbb{R}_{\geq 1}$ is the **guidance scale**. This definition allows CFG to be applied without breaking the function type (9) of $f_{\boldsymbol{\theta}}$.

# Remark on CFG Implementation

**Remark**

In practice, implementations often batch $\boldsymbol{x}_t$ and $\boldsymbol{c}_\pm$ together to compute (**??**)–(**??**) simultaneously in a single forward pass, but the mathematical type is always (9). In this document, we will henceforth describe it in the form of applying twice and then composing, as in (**??**)–(**??**). The timestep $t$ called at this time is the discrete timestep given by the scheduler.

# Denoising Schedulers (DDPM, Euler a, DPM 2+ Karras)

**Definition (Reverse Diffusion Algorithm by DDPM)**

The following data are given.

- **Quantities determined during training**:
  - Number of training timesteps $T \in \mathbb{Z}_{>0}$.
  - Beta sequence $\boldsymbol{\beta} = (\beta_0, \ldots, \beta_{T-1}) \in (0,1)^T$ and derived from it

$$\alpha_t \coloneqq (1 - \beta_t), \quad \bar{\alpha}_t \coloneqq \prod_{s=0}^{t} (1 - \beta_s), \quad t = 0, 1, \ldots, T-1. \tag{13}$$

    For convenience, let $\bar{\alpha}_{-1} \coloneqq 1$.
  - Noise prediction network trained with epsilon prediction type

$$f_{\boldsymbol{\theta}} : \mathbb{R}^{d_{\text{latent}}} \times \mathbb{R}^{d_{\text{context}}} \times \{0, 1, \ldots, T-1\} \to \mathbb{R}^{d_{\text{latent}}}. \tag{14}$$

# DDPM: Given Data (2)

- **Quantities determined by other networks**:
  - Positive embedding $\boldsymbol{c}_+ \in \mathbb{R}^{d_{\text{context}}}$ obtained from the text encoder.
  - Negative (unconditional) embedding $\boldsymbol{c}_- \in \mathbb{R}^{d_{\text{context}}}$ obtained from the text encoder.
- **Quantities specified by the user at inference time**:
  - Number of inference timesteps $N \in \mathbb{Z}_{>0}$ satisfying $N \leq T$.
  - Guidance scale $s \in \mathbb{R}_{\geq 1}$.

# DDPM: Initialization

At this time, the reverse diffusion algorithm by DDPM is as follows.

**(0) Initialization.** Set the step ratio as

$$r \coloneqq \left\lfloor \frac{T}{N} \right\rfloor \in \mathbb{Z}_{>0} \tag{15}$$

and define the discrete timestep sequence as

$$t_k \coloneqq r\,(N-1-k), \quad k = 0, 1, \ldots, N-1. \tag{16}$$

The initial sample is generated by

$$\boldsymbol{x}_{t_0} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}_{d_{\text{latent}}}). \tag{17}$$

# DDPM: Main Loop (1)

**(1) Main loop.** For $k = 0, 1, \ldots, N-2$, perform the following.

- Let the current time be $t_k$ and the next time be $t_{k+1}$. In this case, $t_{k+1} < t_k$.
- Let the current sample be $\boldsymbol{x}_{t_k} \in \mathbb{R}^{d_{\text{latent}}}$.
- Composite the noise prediction by CFG:

$$\widehat{\boldsymbol{\epsilon}}_{\text{cond}} \coloneqq f_{\boldsymbol{\theta}}(\boldsymbol{x}_{t_k}, \boldsymbol{c}_+, t_k), \tag{18}$$

$$\widehat{\boldsymbol{\epsilon}}_{\text{uncond}} \coloneqq f_{\boldsymbol{\theta}}(\boldsymbol{x}_{t_k}, \boldsymbol{c}_-, t_k), \tag{19}$$

$$\widehat{\boldsymbol{\epsilon}}_{t_k} \coloneqq \widehat{\boldsymbol{\epsilon}}_{\text{uncond}} + s\big(\widehat{\boldsymbol{\epsilon}}_{\text{cond}} - \widehat{\boldsymbol{\epsilon}}_{\text{uncond}}\big). \tag{20}$$

# DDPM: Main Loop (2)

- Set the variance as

$$\sigma_{t_k}^2 \coloneqq \frac{1-\bar{\alpha}_{t_{k+1}}}{1-\bar{\alpha}_{t_k}}\,\beta_k \tag{21}$$

and using Gaussian noise

$$\boldsymbol{z}_{t_k} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}_{d_{\text{latent}}}) \tag{22}$$

update as

$$\boldsymbol{x}_{t_{k+1}} \coloneqq \frac{1}{\sqrt{\alpha_{t_k}}}\boldsymbol{x}_{t_k} - \frac{\beta_{t_k}}{\sqrt{\bar{\alpha}_{t_k}}\sqrt{1-\bar{\alpha}_{t_k}}}\widehat{\boldsymbol{\epsilon}}_{t_k} + \sigma_{t_k}\boldsymbol{z}_{t_k}. \tag{23}$$

# Remark: Beta Sequence in SD 1.5

**Remark (Beta sequence in Stable Diffusion 1.5)**

In the public config file for Stable Diffusion 1.5[1], the $\beta$ sequence is defined by the **scaled linear** method as

$$\beta_t = \Big(\mathrm{linspace}\big(\sqrt{0.00085}, \sqrt{0.012}, 1000\big)_t\Big)^2, \quad t = 0, 1, \ldots, 999. \tag{24}$$

Specifically

$$\beta_0 = 0.00085, \tag{25}$$
$$\beta_1 \approx 0.0008546986, \tag{26}$$
$$\beta_2 \approx 0.0008594103, \tag{27}$$
$$\beta_{10} \approx 0.0008975693, \tag{28}$$
$$\beta_{100} \approx 0.0013839726, \tag{29}$$
$$\beta_{999} = 0.012, \tag{30}$$

# Remark: Structure of DDPM Update

**Remark**

Very roughly speaking, in the update step, the coefficient of $\boldsymbol{x}_{t_k}$ is relatively close to 1, and the values of $-\widehat{\boldsymbol{\epsilon}}_{t_k}$ and $\boldsymbol{z}_{t_k}$ are small compared to 1. That is, roughly speaking, the DDPM update has a structure of obtaining the step direction with a neural network, scaling it, subtracting it, and adding small Gaussian noise close to standard deviation $\sqrt{\beta}_t$ to it.

# Remark: Correspondence to Diffusers

**Remark**

Definition 3 corresponds one-to-one with the default settings (epsilon prediction, fixed small variance) of `DDPMScheduler.step`, and applying CFG twice does not require changes to the scheduler's API.

# Euler a: Given Data

**Definition (Reverse Diffusion Algorithm by Euler a)**

The following data are given.

- **Quantities determined during training**:
  - Number of training timesteps $T \in \mathbb{Z}_{>0}$.
  - Cumulative product sequence on training timesteps

$$\bar{\alpha}_t \in (0, 1], \quad t = 0, 1, \ldots, T-1. \tag{31}$$

  - U-Net (noise predictor) trained with epsilon prediction

$$f_{\boldsymbol{\theta}} : \mathbb{R}^{d_{\text{latent}}} \times \mathbb{R}^{d_{\text{context}}} \times [0, T-1] \to \mathbb{R}^{d_{\text{latent}}}. \tag{32}$$

# Euler a: User Quantities

- **Quantities determined by other networks**:
  - Positive embedding $\boldsymbol{c}_+ \in \mathbb{R}^{d_{\text{context}}}$.
  - Negative embedding $\boldsymbol{c}_- \in \mathbb{R}^{d_{\text{context}}}$.
- **Quantities specified by the user at inference time**:
  - Number of inference timesteps $N \in \mathbb{Z}_{>0}$.
  - Guidance scale $s \in \mathbb{R}_{\geq 1}$.

# Euler a: Timestep Sequence

**(0) Timestep sequence generation.** For $k = 0, 1, \ldots, N-1$, define

$$\tau_k \coloneqq \frac{T-1}{N-1}(N-1-k). \tag{33}$$

# Euler a: Sigma Sequence

**(1) Sigma sequence generation.** Corresponding to the training timesteps, set

$$\sigma_t^{\text{train}} \coloneqq \sqrt{\frac{1-\bar{\alpha}_t}{\bar{\alpha}_t}}, \quad t = 0, 1, \ldots, T-1 \tag{34}$$

and by linear interpolation

$$\text{LinInterp}(\{\sigma_t^{\text{train}}\}_{t=0}^{T-1};\ \tau) \coloneqq \sigma_{\lfloor\tau\rfloor}^{\text{train}} + (\tau - \lfloor\tau\rfloor)\big(\sigma_{\lceil\tau\rceil}^{\text{train}} - \sigma_{\lfloor\tau\rfloor}^{\text{train}}\big) \tag{35}$$

let

$$\sigma_k \coloneqq \text{LinInterp}(\{\sigma_t^{\text{train}}\};\ \tau_k), \quad k = 0, 1, \ldots, N-1. \tag{36}$$

Furthermore, add

$$\sigma_N \coloneqq 0. \tag{37}$$

# Euler a: Initialization and Main Loop (1)

**(2) Initialization.** Let

$$\boldsymbol{x}^{(0)} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}_{d_{\text{latent}}}). \tag{38}$$

**(3) Main loop.** For $k = 0, 1, \ldots, N-1$, perform the following.

# Euler a: Main Loop (2)

- Let the current sample be $\boldsymbol{x}^{(k)}$, current time $\tau_k$, current sigma $\sigma_k$, and next sigma $\sigma_{k+1}$.
- Before inputting to the U-Net, normalize as

$$\widetilde{\boldsymbol{x}}^{(k)} := \frac{\boldsymbol{x}^{(k)}}{\sqrt{1+\sigma_k^2}}. \tag{39}$$

- Composite the noise prediction by CFG:

$$\widehat{\boldsymbol{\epsilon}}_{\text{cond}}^{(k)} := f_{\boldsymbol{\theta}}(\widetilde{\boldsymbol{x}}^{(k)}, \boldsymbol{c}_+, \tau_k), \tag{40}$$

$$\widehat{\boldsymbol{\epsilon}}_{\text{uncond}}^{(k)} := f_{\boldsymbol{\theta}}(\widetilde{\boldsymbol{x}}^{(k)}, \boldsymbol{c}_-, \tau_k), \tag{41}$$

$$\widehat{\boldsymbol{\epsilon}}^{(k)} := \widehat{\boldsymbol{\epsilon}}_{\text{uncond}}^{(k)} + s\big(\widehat{\boldsymbol{\epsilon}}_{\text{cond}}^{(k)} - \widehat{\boldsymbol{\epsilon}}_{\text{uncond}}^{(k)}\big). \tag{42}$$

# Euler a: Main Loop (3)

- Let the Euler coefficients be

$$\sigma_k^{\uparrow} \coloneqq \sqrt{\frac{\sigma_{k+1}^2(\sigma_k^2 - \sigma_{k+1}^2)}{\sigma_k^2}}, \tag{43}$$

$$\sigma_k^{\downarrow} \coloneqq \sqrt{\sigma_{k+1}^2 - (\sigma_k^{\uparrow})^2}. \tag{44}$$

  Note that $\left(\sigma_k^{\uparrow}\right)^2 + \left(\sigma_k^{\downarrow}\right)^2 = \sigma_{k+1}^2$.
- Let the Gaussian noise be

$$\boldsymbol{z}^{(k)} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}_{d_{\text{latent}}}) \tag{45}$$

  and define the next sample as

$$\boldsymbol{x}^{(k+1)} \coloneqq \boldsymbol{x}^{(k)} - \left(\sigma_k - \sigma_k^{\downarrow}\right)\widehat{\boldsymbol{\epsilon}}^{(k)} + \sigma_k^{\uparrow}\boldsymbol{z}^{(k)}. \tag{46}$$

# Euler a: Output and Remark

**(4) Output.** The final sample $\boldsymbol{x}^{(N)}$ corresponds to $\sigma_N = 0$ and is output.

**Remark**

Although Euler a is a different algorithm from DDPM, it shares with DDPM the structure of obtaining the step direction with a neural network, scaling it, subtracting it, and adding small Gaussian noise close to standard deviation square root beta t to it.

# DPM++ M Karras: Overview

One of the most frequently used samplers in practical implementations such as Stable Diffusion models is **DPM++ M Karras**. This corresponds to the case where the `DPMSolverMultistepScheduler` class in the HuggingFace `diffusers` library is initialized with

- `algorithm_type="dpmsolver++"`,
- `solver_order=2`,
- `solver_type="midpoint"`,
- `use_karras_sigmas=True`,
- `final_sigmas_type="zero"`,
- `lower_order_final=True`

[2]

[2] See the relevant part of `https://github.com/huggingface/diffusers/blob/main/src/diffusers/schedulers/scheduling_dpmsolver_multistep.py`.

# DPM++ M Karras: Definition (Data 1/2)

## Definition (Reverse Diffusion Algorithm by DPM++ M Karras)

We are given the following data.

- **Quantities determined during training**:
  - Number of training timesteps $T \in \mathbb{Z}_{>0}$.
  - Cumulative product sequence over training timesteps

$$\bar{\alpha}_t \in (0, 1], \quad t = 0, 1, \ldots, T-1. \tag{47}$$

  - Training sigma sequence corresponding to training timesteps

$$\sigma_t^{\text{train}} \coloneqq \sqrt{\frac{1 - \bar{\alpha}_t}{\bar{\alpha}_t}}, \quad t = 0, 1, \ldots, T-1. \tag{48}$$

  - U-Net trained with noise (epsilon) prediction

$$f_{\boldsymbol{\theta}} : \mathbb{R}^{d_{\text{latent}}} \times \mathbb{R}^{d_{\text{context}}} \times [0, T-1] \to \mathbb{R}^{d_{\text{latent}}}. \tag{49}$$

# DPM++ M Karras: Definition (Data 2/2)

**Definition (Reverse Diffusion Algorithm by DPM++ M Karras (cont.))**

- **Quantities determined by other networks**:
  - Positive embedding $\boldsymbol{c}_+ \in \mathbb{R}^{d_{\text{context}}}$.
  - Negative embedding $\boldsymbol{c}_- \in \mathbb{R}^{d_{\text{context}}}$.
- **Quantities specified by the user at inference time**:
  - Number of inference timesteps $N \in \mathbb{Z}_{>0}$.
  - Guidance scale $s \in \mathbb{R}_{\geq 1}$.

# Step (0): Karras Sigma Sequence

**Definition (Reverse Diffusion Algorithm by DPM++ M Karras (cont.))**

**(0) Generation of Karras sigma sequence.** Let the Karras-type parameter be

$$\rho \coloneqq 7.0 \tag{50}$$

Let the minimum and maximum of the training sigmas be

$$\sigma_{\min} \coloneqq \min_{0 \le t \le T-1} \sigma_t^{\text{train}}, \tag{51}$$

$$\sigma_{\max} \coloneqq \max_{0 \le t \le T-1} \sigma_t^{\text{train}} \tag{52}$$

Then, for $k = 0, 1, \ldots, N-1$, define

$$r_k \coloneqq \frac{k}{N-1}, \tag{53}$$

$$\sigma_k \coloneqq \left(\sigma_{\max}^{1/\rho} + r_k(\sigma_{\min}^{1/\rho} - \sigma_{\max}^{1/\rho})\right)^{\rho} \tag{54}$$

# Step (1): Map $\sigma \to$ Training Timestep

**Definition (Reverse Diffusion Algorithm by DPM++ M Karras (cont.))**

**(1) Mapping from sigmas to training timesteps.** Let the logarithm of the training sigmas be

$$\ell_t \coloneqq \log \sigma_t^{\text{train}}, \quad t = 0, 1, \ldots, T-1 \tag{56}$$

For each $k = 0, 1, \ldots, N-1$, define

$$\tau_k \coloneqq \arg \min_{0 \le t \le T-1} \left| \log \sigma_k - \ell_t \right| \tag{57}$$

# Step (2): Define $\alpha_k$

**Definition (Reverse Diffusion Algorithm by DPM++ M Karras (cont.))**

**(2) Definition of $\alpha_k$.** For each $k = 0, 1, \ldots, N$, let

$$\alpha_k \coloneqq \frac{1}{\sqrt{1+\sigma_k^2}} \tag{58}$$

At this time,

$$\frac{1}{\sqrt{1+\sigma_k^2}} = \alpha_k, \qquad \sqrt{1+\sigma_k^2} = \frac{1}{\alpha_k} \tag{59}$$

holds.

# Step (3): Initialization

**Definition (Reverse Diffusion Algorithm by DPM++ M Karras (cont.))**

**(3) Initialization.** Let

$$\boldsymbol{x}^{(0)} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}_{d_{\text{latent}}}) \tag{60}$$

Also, set the history vector to

$$\boldsymbol{m}^{(-1)} := \mathbf{0} \tag{61}$$

**Definition (Reverse Diffusion Algorithm by DPM++ M Karras (cont.))**

**(4) First step.** When $k = 0$, the normalized input to the U-Net is

$$\widetilde{\boldsymbol{x}}^{(0)} \coloneqq \alpha_0 \, \boldsymbol{x}^{(0)} \tag{62}$$

Apply CFG to this to obtain the noise prediction

$$\widehat{\boldsymbol{\epsilon}}^{(0)}_{\text{cond}} \coloneqq f_{\boldsymbol{\theta}}(\widetilde{\boldsymbol{x}}^{(0)}, \boldsymbol{c}_+, \tau_0), \tag{63}$$

$$\widehat{\boldsymbol{\epsilon}}^{(0)}_{\text{uncond}} \coloneqq f_{\boldsymbol{\theta}}(\widetilde{\boldsymbol{x}}^{(0)}, \boldsymbol{c}_-, \tau_0), \tag{64}$$

$$\widehat{\boldsymbol{\epsilon}}^{(0)} \coloneqq \widehat{\boldsymbol{\epsilon}}^{(0)}_{\text{uncond}} + s\big(\widehat{\boldsymbol{\epsilon}}^{(0)}_{\text{cond}} - \widehat{\boldsymbol{\epsilon}}^{(0)}_{\text{uncond}}\big) \tag{65}$$

# Step (4): First Step — Data Prediction and Update

**Definition (Reverse Diffusion Algorithm by DPM++ M Karras (cont.))**

Define the noise prediction converted to data prediction as

$$\boldsymbol{m}^{(0)} \coloneqq \frac{1}{\alpha_0}\,\boldsymbol{x}^{(0)} - \sigma_0\,\widehat{\boldsymbol{\epsilon}}^{(0)} \tag{66}$$

Here, if we let

$$\Delta_0 \coloneqq \frac{\sigma_1\alpha_0 - \alpha_1\sigma_0}{\sigma_0} \tag{67}$$

the update for the first step is given by

$$\boldsymbol{x}^{(1)} \coloneqq \frac{\sigma_1}{\sigma_0}\,\boldsymbol{x}^{(0)} - \Delta_0\,\boldsymbol{m}^{(0)} \tag{68}$$

**Definition (Reverse Diffusion Algorithm by DPM++ M Karras (cont.))**

**(5) Intermediate steps.** For $k = 1, 2, \ldots, N-2$, perform the following. Let the normalized input be

$$\widetilde{\boldsymbol{x}}^{(k)} \coloneqq \alpha_k \, \boldsymbol{x}^{(k)} \tag{69}$$

and obtain

$$\widehat{\boldsymbol{\epsilon}}^{(k)}_{\text{cond}} \coloneqq f_{\boldsymbol{\theta}}(\widetilde{\boldsymbol{x}}^{(k)}, \boldsymbol{c}_+, \tau_k), \tag{70}$$

$$\widehat{\boldsymbol{\epsilon}}^{(k)}_{\text{uncond}} \coloneqq f_{\boldsymbol{\theta}}(\widetilde{\boldsymbol{x}}^{(k)}, \boldsymbol{c}_-, \tau_k), \tag{71}$$

$$\widehat{\boldsymbol{\epsilon}}^{(k)} \coloneqq \widehat{\boldsymbol{\epsilon}}^{(k)}_{\text{uncond}} + s\big(\widehat{\boldsymbol{\epsilon}}^{(k)}_{\text{cond}} - \widehat{\boldsymbol{\epsilon}}^{(k)}_{\text{uncond}}\big) \tag{72}$$

# Step (5): Intermediate Steps — Data Prediction Terms

**Definition (Reverse Diffusion Algorithm by DPM++ M Karras (cont.))**

Convert this back to data prediction and let it be

$$\boldsymbol{m}^{(k)} \coloneqq \frac{1}{\alpha_k}\,\boldsymbol{x}^{(k)} - \sigma_k\,\widehat{\boldsymbol{\epsilon}}^{(k)} \tag{73}$$

Also, define

$$\Delta_k \coloneqq \frac{\sigma_{k+1}\alpha_k - \alpha_{k+1}\sigma_k}{\sigma_k} \tag{74}$$

Furthermore, define the quantity corresponding to the log difference ratio between the previous step and the current step by

$$r_k \coloneqq \frac{\log\left(\frac{\alpha_k\sigma_{k-1}}{\sigma_k\alpha_{k-1}}\right)}{\log\left(\frac{\alpha_{k+1}\sigma_k}{\sigma_{k+1}\alpha_k}\right)} \tag{75}$$

# Step (5): Intermediate Steps — 2-Stage Midpoint Update

**Definition (Reverse Diffusion Algorithm by DPM++ M Karras (cont.))**

At this time, the 2-stage midpoint update is

$$\boldsymbol{x}^{(k+1)} := \frac{\sigma_{k+1}}{\sigma_k}\,\boldsymbol{x}^{(k)} - \Delta_k\,\boldsymbol{m}^{(k)} - \frac{\Delta_k}{2r_k}\left(\boldsymbol{m}^{(k)} - \boldsymbol{m}^{(k-1)}\right) \tag{76}$$

## Step (6): Final Step

**Definition (Reverse Diffusion Algorithm by DPM++ M Karras (cont.))**

**(6) Final step.** Similarly, when $k = N - 1$, let

$$\widetilde{\boldsymbol{x}}^{(N-1)} \coloneqq \alpha_{N-1}\, \boldsymbol{x}^{(N-1)} \tag{77}$$

obtain the noise prediction

$$\widehat{\boldsymbol{\epsilon}}_{\mathrm{cond}}^{(N-1)} \coloneqq f_{\boldsymbol{\theta}}(\widetilde{\boldsymbol{x}}^{(N-1)}, \boldsymbol{c}_{+}, \tau_{N-1}), \tag{78}$$

$$\widehat{\boldsymbol{\epsilon}}_{\mathrm{uncond}}^{(N-1)} \coloneqq f_{\boldsymbol{\theta}}(\widetilde{\boldsymbol{x}}^{(N-1)}, \boldsymbol{c}_{-}, \tau_{N-1}), \tag{79}$$

$$\widehat{\boldsymbol{\epsilon}}^{(N-1)} \coloneqq \widehat{\boldsymbol{\epsilon}}_{\mathrm{uncond}}^{(N-1)} + s\big(\widehat{\boldsymbol{\epsilon}}_{\mathrm{cond}}^{(N-1)} - \widehat{\boldsymbol{\epsilon}}_{\mathrm{uncond}}^{(N-1)}\big) \tag{80}$$

and let

$$\boldsymbol{m}^{(N-1)} \coloneqq \frac{1}{\alpha_{N-1}}\, \boldsymbol{x}^{(N-1)} - \sigma_{N-1}\, \widehat{\boldsymbol{\epsilon}}^{(N-1)} \tag{81}$$

Here, since $\sigma_N = 0$ and $\alpha_N = 1$,

# Step (7): Output

**Definition (Reverse Diffusion Algorithm by DPM++ M Karras (cont.))**

**(7) Output.** Output the final sample $\boldsymbol{x}^{(N)}$.

# Remarks on DPM++ M Karras

**Remark**

The DPM++ M Karras algorithm is complex, but its characteristic points are:

- It updates deterministically in the intermediate steps without using pseudo-random numbers.
- It requires not only the output of the immediately preceding step but also the output before that.

# Factored Midpoint Update and Intuition

## Remark

Equation (76) can be written as

$$\boldsymbol{x}^{(k+1)} = \frac{\sigma_{k+1}}{\sigma_k}\,\boldsymbol{x}^{(k)} - \Delta_k \left( \boldsymbol{m}^{(k)} + \frac{1}{2r_k}\big(\boldsymbol{m}^{(k)} - \boldsymbol{m}^{(k-1)}\big) \right) \tag{84}$$

Here,

$$\Delta_k = \frac{\sigma_{k+1}\alpha_k - \alpha_{k+1}\sigma_k}{\sigma_k} \tag{85}$$

represents the "discrepancy between the decrease in $\alpha$ and the decrease in $\sigma$" between timesteps $k$ and $k+1$, and it can be interpreted that second-order accuracy is obtained by multiplying this discrepancy by the current data prediction and its first-order difference. Also, the fact that the final equation (83) simply becomes $\boldsymbol{m}^{(N-1)}$ is due to the design where the noise completely vanishes at $\sigma_N = 0$.

# Advantages of Each Scheduler (Practical Perspective)

# Practical Remarks: DDPM

**Remark (DDPM)**

Theoretically clear and consistent with the training formulation, but slow if the number of inference steps is large. In practice, faster schedulers such as DDIM, PNDM, Euler, and DPM-based schedulers are often chosen[3].

[3] Diffusers guide:
`https://huggingface.co/docs/diffusers/en/using-diffusers/write_own_pipeline`.

# Practical Remarks: Euler / Euler a

**Remark (Euler / Euler a)**

High quality in few steps (guideline of 20–30 steps), easy to implement and tune, and highly robust[4].

[4] Diffusers Euler-based API:
`https://huggingface.co/docs/diffusers/en/api/schedulers/euler`,
`https://huggingface.co/docs/diffusers/en/api/schedulers/euler_ancestral`.

# Practical Remarks: DPM++ M Karras

**Remark (DPM++ M Karras)**

It is a second-order multistep method, and by combining the data-prediction-type DPMSolver++ with the Karras sigma sequence, it is easy to achieve a high quality/speed trade-off even with few steps [**?**, **?**].

# Summary and Next Lecture Preview

## Summary Corresponding to Learning Outcomes

- Input and Output: The reverse diffusion process takes the current sample, text embeddings (positive/negative), and timestep (integer timestep for DDPM, float timestep for Euler a, or pseudo-timestep for DPM++ M Karras) as input, and outputs the next sample.
- Network Conditions: The output dimension must match the input sample dimension ((**??**)), and it must be able to accept conditional inputs and timesteps.
- Handling/Control of Text: The quality/diversity trade-off can be adjusted using CFG (**??**) [3].
- Scheduler Selection: DDPM is fundamental, Euler/Euler a are for high-speed general use, and DPM++ M Karras has strengths in high quality with few steps.

# Next Lecture Preview

Next time, we will move away from inference for a moment and give an overview of the diffusion process, which is used for training the reverse diffusion process.

# References i


[1] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio.
**Density estimation using real nvp.**
In International Conference on Learning Representations (ICLR), 2017.

[2] Jonathan Ho, Ajay Jain, and Pieter Abbeel.
**Denoising diffusion probabilistic models.**
Advances in Neural Information Processing Systems, 33:6840–6851, 2020.

[3] Jonathan Ho and Tim Salimans.
**Classifier-free diffusion guidance.**
arXiv preprint arXiv:2207.12598, 2022.

# References ii


[4] Tero Karras, Miika Aittala, Samuli Laine, Erik Härkönen, Janne Hellsten, Jaakko Lehtinen, and Timo Aila.
**Elucidating the design space of diffusion-based generative models.**
Advances in Neural Information Processing Systems, 35:26565–26577, 2022.

[5] Diederik P. Kingma and Prafulla Dhariwal.
**Glow: Generative flow with invertible 1x1 convolutions.**
In Advances in Neural Information Processing Systems (NeurIPS), volume 31, 2018.

[6] Diederik P. Kingma and Max Welling.
**Auto-encoding variational bayes.**
arXiv preprint arXiv:1312.6114, 2013.

# References iii

[7] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer.
**High-resolution image synthesis with latent diffusion models.**
Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pages 10684–10695, 2022.