AI Applications Lecture 13
Image Generation AI
Reverse Diffusion Process

SUZUKI, Atsushi
Jing WANG

October 31, 2025

Contents

1 Introduction

1.1 Recap of Previous Lecture

In the previous lecture, we learned that a **natural image decoder**, such as those realized by **Variational Autoencoders (VAE)** [6, 8], can construct high-resolution **natural images** from low-resolution **latent images**. Specifically, we reconfirmed that the VAE decoder $\mathrm{Dec}_\gamma : \mathcal{Z} \to \mathcal{I}$ 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 $\mathcal{Z}$.

1.2 Learning Outcomes

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.

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 (set-builder notation, extension).
  - $\{x \in \mathcal{A} \mid P(x)\}$: 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 $\mathbb{Z}_{>0}$.

- **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:

$$\boldsymbol{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}. \tag{1}$$

  - Standard inner product:

$$\langle \boldsymbol{u}, \boldsymbol{v} \rangle \coloneqq \sum_{i=1}^{d_{\mathrm{emb}}} u_i v_i. \tag{2}$$

- **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

$$\boldsymbol{A} = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix}. \tag{3}$$

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

$$\boldsymbol{A}^\top = \begin{bmatrix} a_{1,1} & \cdots & a_{m,1} \\ \vdots & \ddots & \vdots \\ a_{1,n} & \cdots & a_{m,n} \end{bmatrix}. \tag{4}$$

  - The transpose of a vector is

$$\boldsymbol{v}^{\top} = \begin{bmatrix} v_1 & \cdots & v_n \end{bmatrix}. \tag{5}$$

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

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

The goal of the latent generator LatentGen_{β} 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 Dec_{γ} 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.

4 Motivation for Introducing the Reverse Diffusion Process

4.1 Problem Formulation

Given a text condition $\left(\boldsymbol{c}^{[i]}\right)_{i=1}^{m}$, let $\mathrm{P}_{(\boldsymbol{c}^{[i]})_{i=1}^{m}}$ 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

$$\boldsymbol{x} \sim \mathrm{P}_{(\boldsymbol{c}^{[i]})_{i=1}^{m}}. \tag{6}$$

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

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.

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.

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.

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.

4.2 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 λ be the base distribution on it (in practice, the standard multivariate normal distribution). We find a **measurable function**

$$f_{(\cdot)} : \left(\prod_{i=1}^{m} \mathbb{R}^{d^{(i)}}\right) \times \mathcal{S} \to \mathcal{I} \tag{7}$$

and define the **pushforward** as

$$f\left(\left(\boldsymbol{c}^{[i]}\right)_{i=1}^{m}, \cdot\right) \sharp\lambda(A) \coloneqq \lambda\left(\left\{\boldsymbol{\epsilon} \in \mathcal{S} \;\middle|\; f\left(\left(\boldsymbol{c}^{[i]}\right)_{i=1}^{m}, \boldsymbol{z}\right) \in A\right\}\right), \quad A \subset \mathcal{I}. \tag{8}$$

The target condition is

$$f\left(\left(\boldsymbol{c}^{[i]}\right)_{i=1}^{m}, \cdot\right) \sharp\lambda \approx \mathrm{P}_{\left(\boldsymbol{c}^{[i]}\right)_{i=1}^{m}}. \tag{9}$$

If this is achieved, we can realize (6) simply by generating $\boldsymbol{z} \sim \lambda$ with pseudo-random numbers and setting

$$\boldsymbol{x} = f\left(\left(\boldsymbol{c}^{[i]}\right)_{i=1}^{m}, \boldsymbol{z}\right). \tag{10}$$

4.3 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, 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 $z = f^{-1}(x)$. However, if we consider a rotation map Rot centered at the origin, $(f \circ \mathrm{Rot})\sharp\lambda_{\mathrm{normal}}$ is equal to $f\sharp\lambda_{\mathrm{normal}}$, so it is also ideal. In this case, $\boldsymbol{x}$ corresponds to $z' = \mathrm{Rot}^{-1} \circ f^{-1}(\boldsymbol{x})$. We cannot say that z is the correct input, nor that z' is the correct input. Therefore, it is not a simple "input-output correspondence regression problem".

Therefore, if we want to formulate it as learning an input-output relationship, we need to determine the "input" $z^{(i)}$ 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. Autoencoders, including the VAE explained last time, are also based on this idea.

4.4 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 (9) 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).

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

$$p_X(\boldsymbol{x}^{(n)} \mid \boldsymbol{\theta}) = p_Z\big(f_{\boldsymbol{\theta}}^{-1}(\boldsymbol{x}^{(n)})\big) \left|\det \frac{\partial f_{\boldsymbol{\theta}}^{-1}(\boldsymbol{x}^{(n)})}{\partial \boldsymbol{x}^{\top}}\right|, \tag{11}$$

so the log-likelihood of all data is

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{n=1}^{N} \log p_X(\boldsymbol{x}^{(n)} \mid \boldsymbol{\theta}) = \sum_{n=1}^{N} \left\{ \log p_Z\left(f_{\boldsymbol{\theta}}^{-1}(\boldsymbol{x}^{(n)})\right) + \log \left| \det \frac{\partial f_{\boldsymbol{\theta}}^{-1}(\boldsymbol{x}^{(n)})}{\partial \boldsymbol{x}^{\top}} \right| \right\}. \tag{12}$$

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

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \sum_{n=1}^{N} \left[\nabla_{\boldsymbol{\theta}} \log p_Z\left(f_{\boldsymbol{\theta}}^{-1}(\boldsymbol{x}^{(n)})\right) + \nabla_{\boldsymbol{\theta}} \log \left| \det \frac{\partial f_{\boldsymbol{\theta}}^{-1}(\boldsymbol{x}^{(n)})}{\partial \boldsymbol{x}^{\top}} \right| \right], \tag{13}$$

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.

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.

4.5 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.

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.

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 τ_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 γ_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 5.1 (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 (21), (38), (54) described later. The **noise predictor** (or **data predictor**) f_θ in the reverse diffusion process is

$$f_{\boldsymbol{\theta}} : \mathbb{R}^{d_{\text{latent}}} \times \mathbb{R}^{d_{\text{context}}} \times \mathcal{T} \to \mathbb{R}^{d_{\text{latent}}} \tag{14}$$

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 5.2 (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

$$\widehat{\boldsymbol{\epsilon}}_{\text{cond}} \coloneqq f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, \boldsymbol{c}_+, t), \tag{15}$$

$$\widehat{\boldsymbol{\epsilon}}_{\text{uncond}} \coloneqq f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, \boldsymbol{c}_-, t), \tag{16}$$

$$\widehat{\boldsymbol{\epsilon}}_{\text{CFG}} \coloneqq \widehat{\boldsymbol{\epsilon}}_{\text{uncond}} + s\big(\widehat{\boldsymbol{\epsilon}}_{\text{cond}} - \widehat{\boldsymbol{\epsilon}}_{\text{uncond}}\big), \tag{17}$$

where $s \in \mathbb{R}_{\geq 1}$ is the **guidance scale**. This definition allows CFG to be applied without breaking the function type (14) of f_θ.

Remark 5.1. In practice, implementations often batch $\boldsymbol{x}_t$ and $\boldsymbol{c}_\pm$ together to compute (15)–(16) simultaneously in a single forward pass, but the mathematical type is always (14). In this document, we will henceforth describe it in the form of applying twice and then composing, as in (15)–(17). The timestep t called at this time is the discrete timestep given by the scheduler.

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

Hereafter, focusing only on the inference stage, we will strictly provide the step update equations of representative schedulers in a one-to-one correspondence with the relevant code in the implementation (Hugging Face Diffusers). The original papers are also cited [2, 4, 7].

6.1 DDPM (Variance-Preserving Discrete Process)

Definition 6.1 (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 $\boldsymbol{\beta} = (\beta_0, \dots, \beta_{T-1}) \in (0,1)^T$ and derived from it

$$\alpha_t \coloneqq (1-\beta_t), \quad \bar{\alpha}_t \coloneqq \prod_{s=0}^{t} (1-\beta_s), \quad t = 0, 1, \dots, T-1. \tag{18}$$

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

$$f_{\boldsymbol{\theta}} : \mathbb{R}^{d_{\text{latent}}} \times \mathbb{R}^{d_{\text{context}}} \times \{0, 1, \dots, T-1\} \to \mathbb{R}^{d_{\text{latent}}}. \tag{19}$$

- **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 $N \le T$.
  - Guidance scale $s \in \mathbb{R}_{\ge 1}$.

At this time, the reverse diffusion algorithm by DDPM is as follows.
(0) Initialization. Set the step ratio as

$$r \coloneqq \left\lfloor \frac{T}{N} \right\rfloor \in \mathbb{Z}_{>0} \tag{20}$$

and define the discrete timestep sequence as

$$t_k \coloneqq r\,(N-1-k), \quad k = 0, 1, \dots, N-1. \tag{21}$$

The initial sample is generated by

$$\boldsymbol{x}_{t_0} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}_{d_{\text{latent}}}). \tag{22}$$

(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:

$$\widehat{\boldsymbol{\epsilon}}_{\text{cond}} \coloneqq f_{\boldsymbol{\theta}}(\boldsymbol{x}_{t_k}, \boldsymbol{c}_+, t_k), \tag{23}$$
$$\widehat{\boldsymbol{\epsilon}}_{\text{uncond}} \coloneqq f_{\boldsymbol{\theta}}(\boldsymbol{x}_{t_k}, \boldsymbol{c}_-, t_k), \tag{24}$$
$$\widehat{\boldsymbol{\epsilon}}_{t_k} \coloneqq \widehat{\boldsymbol{\epsilon}}_{\text{uncond}} + s\big(\widehat{\boldsymbol{\epsilon}}_{\text{cond}} - \widehat{\boldsymbol{\epsilon}}_{\text{uncond}}\big). \tag{25}$$

- Set the variance as

$$\sigma_{t_k}^2 \coloneqq \frac{1 - \bar{\alpha}_{t_{k+1}}}{1 - \bar{\alpha}_{t_k}}\, \beta_k \tag{26}$$

  and using Gaussian noise

$$\boldsymbol{z}_{t_k} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}_{d_{\text{latent}}}) \tag{27}$$

  update as

$$\boldsymbol{x}_{t_{k+1}} \coloneqq \frac{1}{\sqrt{\alpha_{t_k}}} \boldsymbol{x}_{t_k} - \frac{\beta_{t_k}}{\sqrt{\bar{\alpha}_{t_k}}\sqrt{1 - \bar{\alpha}_{t_k}}} \widehat{\boldsymbol{\epsilon}}_{t_k} + \sigma_{t_k} \boldsymbol{z}_{t_k}. \tag{28}$$

Remark 6.1 (β sequence in Stable Diffusion 1.5)**.** In the public config file for Stable Diffusion 1.5[a], the β sequence is defined by the **scaled linear** method as

$$\beta_t = \Big(\operatorname{linspace}\big(\sqrt{0.00085}, \sqrt{0.012}, 1000\big)_t\Big)^2, \quad t = 0, 1, \ldots, 999. \tag{29}$$

Specifically

$$\beta_0 = 0.00085, \tag{30}$$
$$\beta_1 \approx 0.0008546986, \tag{31}$$
$$\beta_2 \approx 0.0008594103, \tag{32}$$
$$\beta_{10} \approx 0.0008975693, \tag{33}$$
$$\beta_{100} \approx 0.0013839726, \tag{34}$$
$$\beta_{999} = 0.012, \tag{35}$$

and $\bar{\alpha}_t$ in (18) is uniquely determined from this sequence.
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 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.

[a] https://github.com/runwayml/stable-diffusion/blob/main/configs/stable-diffusion/v1-inference.yaml

Remark 6.2. (6.1) 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.

6.2 Euler a (Euler Ancestral)

Definition 6.2 (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

$$\bar{\alpha}_t \in (0,1], \quad t = 0, 1, \ldots, T-1. \tag{36}$$

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

$$f_{\boldsymbol{\theta}} : \mathbb{R}^{d_{\text{latent}}} \times \mathbb{R}^{d_{\text{context}}} \times [0, T-1] \to \mathbb{R}^{d_{\text{latent}}}. \tag{37}$$

- **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 $s \in \mathbb{R}_{\geq 1}$.

At this time, Euler a operates by the following procedure.
(0) Timestep sequence generation. For $k = 0, 1, \ldots, N-1$, define

$$\tau_k \coloneqq \frac{T-1}{N-1}(N-1-k). \tag{38}$$

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

$$\sigma_t^{\text{train}} \coloneqq \sqrt{\frac{1-\bar{\alpha}_t}{\bar{\alpha}_t}}, \quad t = 0, 1, \ldots, T-1 \tag{39}$$

and by linear interpolation

$$\mathrm{LinInterp}(\{\sigma_t^{\mathrm{train}}\}_{t=0}^{T-1};\ \tau) \coloneqq \sigma_{\lfloor\tau\rfloor}^{\mathrm{train}} + (\tau - \lfloor\tau\rfloor)\big(\sigma_{\lceil\tau\rceil}^{\mathrm{train}} - \sigma_{\lfloor\tau\rfloor}^{\mathrm{train}}\big) \tag{40}$$

let

$$\sigma_k \coloneqq \mathrm{LinInterp}(\{\sigma_t^{\mathrm{train}}\};\ \tau_k), \quad k = 0, 1, \ldots, N-1. \tag{41}$$

Furthermore, add

$$\sigma_N \coloneqq 0. \tag{42}$$

(2) Initialization. Let

$$\boldsymbol{x}^{(0)} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}_{d_{\mathrm{latent}}}). \tag{43}$$

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

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

$$\widetilde{\boldsymbol{x}}^{(k)} \coloneqq \frac{\boldsymbol{x}^{(k)}}{\sqrt{1+\sigma_k^2}}. \tag{44}$$

- Composite the noise prediction by CFG:

$$\widehat{\boldsymbol{\epsilon}}_{\mathrm{cond}}^{(k)} \coloneqq f_{\boldsymbol{\theta}}(\widetilde{\boldsymbol{x}}^{(k)}, \boldsymbol{c}_+, \tau_k), \tag{45}$$

$$\widehat{\boldsymbol{\epsilon}}_{\mathrm{uncond}}^{(k)} \coloneqq f_{\boldsymbol{\theta}}(\widetilde{\boldsymbol{x}}^{(k)}, \boldsymbol{c}_-, \tau_k), \tag{46}$$

$$\widehat{\boldsymbol{\epsilon}}^{(k)} \coloneqq \widehat{\boldsymbol{\epsilon}}_{\mathrm{uncond}}^{(k)} + s\big(\widehat{\boldsymbol{\epsilon}}_{\mathrm{cond}}^{(k)} - \widehat{\boldsymbol{\epsilon}}_{\mathrm{uncond}}^{(k)}\big). \tag{47}$$

- Let the Euler coefficients be

$$\sigma_k^{\uparrow} \coloneqq \sqrt{\frac{\sigma_{k+1}^2(\sigma_k^2 - \sigma_{k+1}^2)}{\sigma_k^2}}, \tag{48}$$

$$\sigma_k^{\downarrow} \coloneqq \sqrt{\sigma_{k+1}^2 - (\sigma_k^{\uparrow})^2}. \tag{49}$$

  Note that $\left(\sigma_k^{\uparrow}\right)^2 + \left(\sigma_k^{\downarrow}\right)^2 = \sigma_{k+1}^2$.

- Let the Gaussian noise be

$$\boldsymbol{z}^{(k)} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}_{d_{\mathrm{latent}}}) \tag{50}$$

  and define the next sample as

$$\boldsymbol{x}^{(k+1)} \coloneqq \boldsymbol{x}^{(k)} - \big(\sigma_k - \sigma_k^{\downarrow}\big)\widehat{\boldsymbol{\epsilon}}^{(k)} + \sigma_k^{\uparrow}\boldsymbol{z}^{(k)}. \tag{51}$$

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

Remark 6.3. 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 $\sqrt{\beta}_t$ to it.

6.3 DPM 2+ (Karras Sigma Sequence)

Definition 6.3 (Reverse Diffusion Algorithm by DPM 2+ (Karras))**.** The following data are given.

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

$$\bar{\alpha}_t \in (0, 1], \quad t = 0, 1, \ldots, T-1. \tag{52}$$

  - Noise predictor trained with epsilon prediction

$$f_{\boldsymbol{\theta}} : \mathbb{R}^{d_{\text{latent}}} \times \mathbb{R}^{d_{\text{context}}} \times [0, T-1] \to \mathbb{R}^{d_{\text{latent}}}. \tag{53}$$

- **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 $s \in \mathbb{R}_{\geq 1}$.

At this time, the algorithm is as follows.
(0) Base timestep sequence generation. For $k = 0, 1, \ldots, N-1$, define

$$\tau_k \coloneqq \frac{T-1}{N-1}(N-1-k). \tag{54}$$

(1) Base sigma sequence generation. On the training timesteps, define

$$\sigma_t^{\text{train}} \coloneqq \sqrt{\frac{1-\bar{\alpha}_t}{\bar{\alpha}_t}}, \quad t = 0, 1, \ldots, T-1 \tag{55}$$

and by linear interpolation

$$\text{LinInterp}(\{\sigma_t^{\text{train}}\};\ \tau) \coloneqq \sigma_{\lfloor\tau\rfloor}^{\text{train}} + (\tau - \lfloor\tau\rfloor)\big(\sigma_{\lceil\tau\rceil}^{\text{train}} - \sigma_{\lfloor\tau\rfloor}^{\text{train}}\big) \tag{56}$$

let

$$\sigma_k \coloneqq \mathrm{LinInterp}(\{\sigma_t^{\mathrm{train}}\};\ \tau_k), \quad k = 0, 1, \ldots, N-1 \tag{57}$$

and furthermore, add

$$\sigma_N \coloneqq 0. \tag{58}$$

(2) Log-sigma sequence and midpoint sigma sequence. For $k = 0, 1, \ldots, N$, define

$$\lambda_k \coloneqq \log \sigma_k \tag{59}$$

and give the midpoint sigma sequence by

$$\sigma_0^{\mathrm{mid}} \coloneqq \sigma_0, \tag{60}$$

$$\sigma_k^{\mathrm{mid}} \coloneqq \exp\left(\frac{\log \sigma_k + \log \sigma_{k-1}}{2}\right), \quad k = 1, 2, \ldots, N. \tag{61}$$

(3) Construction of the long sigma sequence. Define a sequence of length $2N + 2$, $(\tilde{\sigma}_m)_{m=0}^{2N+1}$, by

$$\tilde{\sigma}_0 \coloneqq \sigma_0, \quad \tilde{\sigma}_{2k-1} \coloneqq \sigma_k, \quad \tilde{\sigma}_{2k} \coloneqq \sigma_k, \quad \tilde{\sigma}_{2N+1} \coloneqq \sigma_N, \quad k = 1, 2, \ldots, N. \tag{62}$$

Similarly, define a long midpoint sigma sequence of length $2N + 2$, $(\tilde{\sigma}_m^{\mathrm{mid}})_{m=0}^{2N+1}$, by

$$\tilde{\sigma}_0^{\mathrm{mid}} \coloneqq \sigma_0^{\mathrm{mid}}, \quad \tilde{\sigma}_{2k-1}^{\mathrm{mid}} \coloneqq \sigma_k^{\mathrm{mid}}, \quad \tilde{\sigma}_{2k}^{\mathrm{mid}} \coloneqq \sigma_k^{\mathrm{mid}}, \quad \tilde{\sigma}_{2N+1}^{\mathrm{mid}} \coloneqq \sigma_N^{\mathrm{mid}}, \quad k = 1, 2, \ldots, N. \tag{63}$$

(4) Conversion to pseudo-timesteps ($\sigma \mapsto t$). Let the log-sigmas on the training timesteps be

$$L_t \coloneqq \log \sigma_t^{\mathrm{train}}, \quad t = 0, 1, \ldots, T-1 \tag{64}$$

and for $\sigma > 0$, define

$$j(\sigma) \coloneqq \max\{\, i \in \{0, 1, \ldots, T-2\} \mid L_i \le \log \sigma \,\}, \tag{65}$$

$$j^+(\sigma) \coloneqq j(\sigma) + 1, \tag{66}$$

$$w(\sigma) \coloneqq \frac{L_{j(\sigma)} - \log \sigma}{L_{j(\sigma)} - L_{j^+(\sigma)}} \in [0, 1], \tag{67}$$

$$\mathrm{SigmaToT}(\sigma) \coloneqq (1 - w(\sigma))\, j(\sigma) + w(\sigma)\, j^+(\sigma). \tag{68}$$

(5) Construction of the long timestep sequence. For the base timestep sequence $(\tau_k)_{k=0}^{N-1}$, define a sequence of length $2N - 1$, $(\theta_m)_{m=0}^{2N-2}$, by

$$\theta_0 \coloneqq \tau_0, \quad \theta_{2k-1} \coloneqq \mathrm{SigmaToT}(\sigma_k^{\mathrm{mid}}), \quad \theta_{2k} \coloneqq \tau_k, \quad k = 1, 2, \ldots, N-1. \tag{69}$$

(6) Initialization. Let

$$\boldsymbol{x}^{(0)} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}_{d_{\text{latent}}}). \tag{70}$$

(7) Main loop (Heun-type two-stage method). For $m = 0, 1, \ldots, 2N-2$, perform the following.

- Let the current pseudo-timestep be θ_m, current sigma $\tilde{\sigma}_m$, midpoint sigma $\tilde{\sigma}_{m+1}^{\text{mid}}$, and next sigma $\tilde{\sigma}_{m+1}$.
- Let the current sample be $\boldsymbol{x}^{(m)}$ and compute the following:

$$\widehat{\boldsymbol{\epsilon}}_{\text{cond}}^{(m)} := f_{\boldsymbol{\theta}}(\boldsymbol{x}^{(m)}, \boldsymbol{c}_+, \theta_m), \tag{71}$$

$$\widehat{\boldsymbol{\epsilon}}_{\text{uncond}}^{(m)} := f_{\boldsymbol{\theta}}(\boldsymbol{x}^{(m)}, \boldsymbol{c}_-, \theta_m), \tag{72}$$

$$\widehat{\boldsymbol{\epsilon}}^{(m)} := \widehat{\boldsymbol{\epsilon}}_{\text{uncond}}^{(m)} + s\big(\widehat{\boldsymbol{\epsilon}}_{\text{cond}}^{(m)} - \widehat{\boldsymbol{\epsilon}}_{\text{uncond}}^{(m)}\big). \tag{73}$$

- When m is even (1st stage), set the input sigma $\sigma_{\text{in}}^{(m)} := \tilde{\sigma}_m$, and find the original image prediction by

$$\widehat{\boldsymbol{x}}_0^{(m)} := \boldsymbol{x}^{(m)} - \sigma_{\text{in}}^{(m)} \widehat{\boldsymbol{\epsilon}}^{(m)}. \tag{74}$$

  Let the derivative be

$$\boldsymbol{g}^{(m)} := \frac{\boldsymbol{x}^{(m)} - \widehat{\boldsymbol{x}}_0^{(m)}}{\sigma_{\text{in}}^{(m)}} \tag{75}$$

  and the step size be

$$\Delta\sigma^{(m)} := \tilde{\sigma}_{m+1}^{\text{mid}} - \tilde{\sigma}_m \tag{76}$$

  and update as

$$\boldsymbol{x}^{(m+1)} := \boldsymbol{x}^{(m)} + \Delta\sigma^{(m)}\, \boldsymbol{g}^{(m)} \tag{77}$$

  and temporarily store $\boldsymbol{x}^{(m)}$ and $\boldsymbol{g}^{(m)}$.

- When m is odd (2nd stage), set the input sigma $\sigma_{\text{in}}^{(m)} := \tilde{\sigma}_m^{\text{mid}}$, and from

$$\widehat{\boldsymbol{x}}_0^{(m)} := \boldsymbol{x}^{(m)} - \sigma_{\text{in}}^{(m)} \widehat{\boldsymbol{\epsilon}}^{(m)} \tag{78}$$

  obtain

$$\boldsymbol{g}^{(m)} := \frac{\boldsymbol{x}^{(m)} - \widehat{\boldsymbol{x}}_0^{(m)}}{\sigma_{\text{in}}^{(m)}}. \tag{79}$$

  Using $\boldsymbol{x}^{(m-1)}$ and $\boldsymbol{g}^{(m-1)}$ stored in the 1st stage, update as

$$\boldsymbol{x}^{(m+1)} := \boldsymbol{x}^{(m-1)} + \frac{\tilde{\sigma}_{m+1} - \tilde{\sigma}_{m-1}}{2}\big(\boldsymbol{g}^{(m-1)} + \boldsymbol{g}^{(m)}\big). \tag{80}$$

(8) Output. Output $\boldsymbol{x}^{(2N-1)}$ at $m = 2N-2$ as the final latent.

Remark 6.4. Definition (6.3) is very complicated, but there are two major differences from DDPM and Euler a.

- It uses the neural network twice per step. Therefore, even if the same number of inference steps N is specified, the inference time tends to be longer.
- It does not use pseudo-random numbers in steps other than initialization.

7 Advantages of Each Scheduler (Practical Perspective)

Remark 7.1 (DDPM)**.** Theoretically clear and consistent with the training formulation, but slow if the number of inference steps is large. In practice, faster schedulers like DDIM, PNDM, Euler, DPM-based are often selected[a].

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

Remark 7.2 (Euler / Euler a)**.** High quality in few steps (20–30 steps target), easy to implement/tune, and highly robust[a].

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

Remark 7.3 (DPM 2+ (Karras))**.** A second-order multi-step method, easy to obtain a high quality/speed trade-off. Implementations combining with the Karras sigma sequence for high quality in few steps are widely used [7].

8 Summary and Next Lecture Preview

8.1 Summary Corresponding to Learning Outcomes

- Input and Output: The reverse diffusion process takes the current sample, text embedding (pos/neg), and time (DDPM's integer time, Euler a's float time, or KDPM2's pseudo-time) as input, and outputs the next sample.
- Network Conditions: The output dimension matches the input sample dimension ((14)), and it can accept conditional input and time.
- Text Handling/Control: The quality/diversity trade-off can be adjusted by CFG (17) [3].
- Scheduler Selection: DDPM is fundamental, Euler/Euler a are for fast general-purpose use, and DPM 2+ (Karras) has strengths in high quality in few steps.

8.2 Next Lecture Preview

Next time, we will move away from inference for a moment and outline the diffusion process used for training the reverse diffusion process.

References

[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.

[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.

[7] Chunqi Lu, Junxian He, Chenyang Xu, and Yang Song. Dpm-solver++: Fast solver for guided sampling of diffusion probabilistic models. arXiv preprint arXiv:2211.01095, 2022.

[8] 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.