# Consistency Models

Yang Song, et al. ICML2023
OpenAI, Stanford Univ.

Suk-Hwan Lee

Artificial Intelligence

Creating the Future

Dong-A University

Division of Computer Engineering &
Artificial Intelligence

# References

Yang Song, Prafulla Dhariwal, Mark Chen, Ilya Sutskever, "**Consistency Models**," arXiv Preprint, arXiv:2303.01469, (ICML2023)

https://github.com/openai/consistency_models

[blog]
https://junia3.github.io/blog/consistency
https://thecho7.tistory.com/entry/

EDM : Tero Karras, Miika Aittala, Timo Aila, Samuli Laine, "**Elucidating the Design Space of Diffusion-Based Generative Models**," NeurIPS 2022. https://github.com/NVlabs/edm

https://www.youtube.com/watch?v=T0Qxzf0eaio

Yang Song, Jascha Sohl-Dickstein, Diederik P. Kingma, Abhishek Kumar, Stefano Ermon, Ben Poole, "**Score-Based Generative Modeling through Stochastic Differential Equations**," ICLR 2021.

https://github.com/yang-song/score_sde

[blog]
https://kimjy99.github.io/논문리뷰/sbgm/
https://blog.si-analytics.ai/49

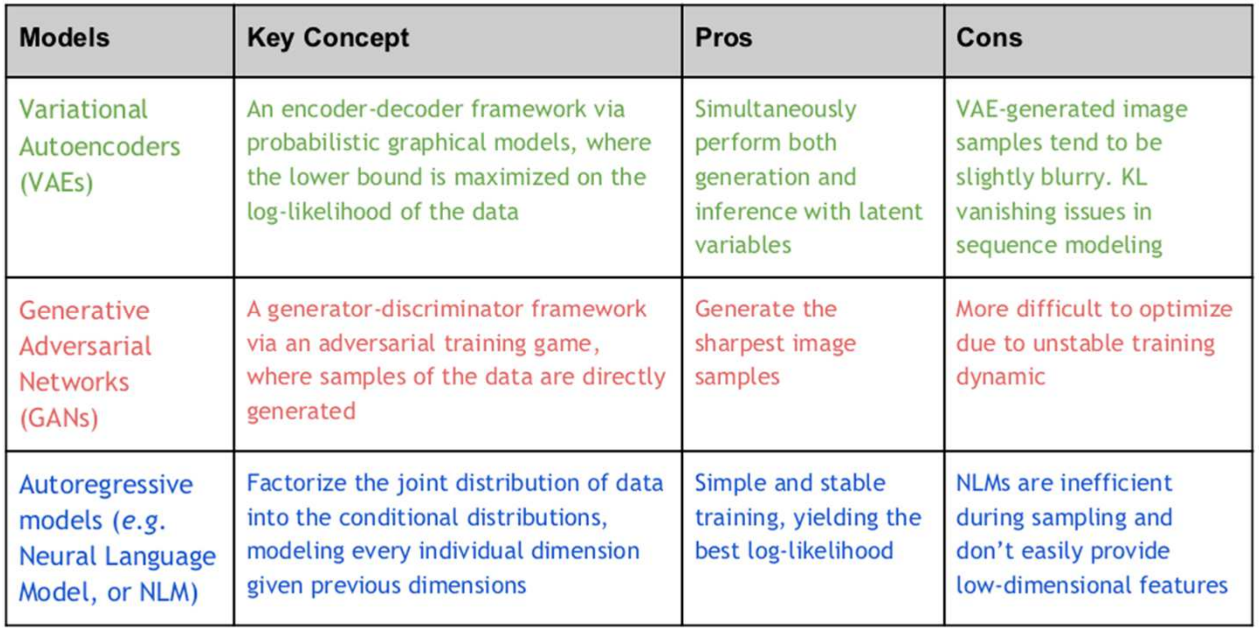
## Consistency Models

### ❖ Diffusion Model : Fast Sampling 불가능한 이유

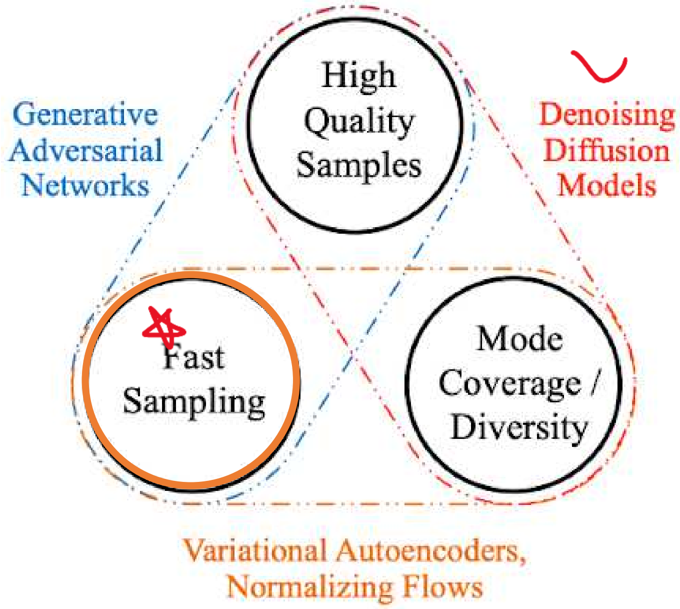
Noise으로부터 원본 방향으로 Denoising 하는 과정 (Foward Process) $q(\mathbf{x}_t|\mathbf{x}_{t-1})$에서, Noise $\beta$가 매우 작다는 가정 하에 Normal Distribution을 사용할 수 있음. 따라서 기존 Diffusion 모델은 수백~수천번의 Iteration으로 학습함

https://thecho7.tistory.com/entry/

### ❖ VAE & GAN & Diffusion



| Models | Key Concept | Pros | Cons |
|---|---|---|---|
| Variational Autoencoders (VAEs) | An encoder-decoder framework via probabilistic graphical models, where the lower bound is maximized on the log-likelihood of the data | Simultaneously perform both generation and inference with latent variables | VAE-generated image samples tend to be slightly blurry. KL vanishing issues in sequence modeling |
| Generative Adversarial Networks (GANs) | A generator-discriminator framework via an adversarial training game, where samples of the data are directly generated | Generate the sharpest image samples | More difficult to optimize due to unstable training dynamic |
| Autoregressive models (*e.g.* Neural Language Model, or NLM) | Factorize the joint distribution of data into the conditional distributions, modeling every individual dimension given previous dimensions | Simple and stable training, yielding the best log-likelihood | NLMs are inefficient during sampling and don't easily provide low-dimensional features |



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

Figure 1. Generative learning trilemma

https://developer.nvidia.com/blog/improving-diffusion-models-as-an-alternative-to-gans-part-1/

- Generative models (openai.com) : https://openai.com/research/generative-models
- A deep generative model trifecta: Three advances that work towards harnessing large-scale power - Microsoft Research https://www.microsoft.com/en-us/research/blog/a-deep-generative-model-trifecta-three-advances-that-work-towards-harnessing-large-scale-power/

# Consistency Models

## Abstract

- **Diffusion models** depend on an **iterative sampling process** that **causes slow generation**.
- **Propose consistency models**, a new family of models that **generate high quality samples** by **directly mapping noise to data**.
  - ✓ **Support** ***fast one-step generation*** by design, while still allowing **multistep sampling** to trade compute for **sample quality**.
  - ✓ **Support** ***zero-shot data editing***, such as *image inpainting*, *colorization*, and *super-resolution*, **without requiring explicit training** on these tasks.
  - ✓ Can be **trained** either by **distilling pre-trained diffusion models**, or as **standalone generative models altogether**.

- Outperform existing distillation techniques for diffusion models in **one- and few-step sampling**, achieving **SOTA FID of 3.55** on CIFAR-10 and **6.20** on ImageNet 64x64 for ***one-step generation***.
- When trained in *isolation*, **consistency models become a new family of generative models that can outperform existing one-step, non-adversarial generative models** on standard benchmarks such as CIFAR-10, ImageNet 64x64 and LSUN 256x256.

❖ **Objective**

- Create generative models that facilitate **efficient**, **single-step generation** without sacrificing important advantages of **iterative refinement**.
- Ability to **trade-off compute for sample quality** when necessary, as well as the capability to perform **zeroshot data editing** tasks.

# Consistency Models

## 1. Introduction

❖ **Fast Sampling ?**

- 기존의 Diffusion이 장인 정신으로 한땀한땀(시간축 $t$에 따라서) Noise를 제거해가는 방식 대신에, 하이패스를 달아서 한번에 Noise로부터 샘플을 생성하는 것이다. - 즉 기존의 GAN이 가능했던 빠른 샘플링인 $G(z)$를 Dffiusion에 대해서도 가능하게 하고 싶다는 것이다. 이러한 생각에서 나온 방법들 중 일부는 다음과 같다.
- ***DDIM*** : Markovian process와 동일한 marginal likelihood를 가지는 Non-Markovian forward process를 정의하고, 이를 통해 샘플링 시퀀스의 time step을 간소화
- ***Diffusion model distillation*** : 샘플링 성능이 좋은 Diffusion 모델 ex) DDPM 을 사용하여 단일 step으로 좋은 샘플링이 가능하게끔 *probability flow ODE*를 학습

✓ Jiaming Song, Chenlin Meng, Stefano Ermon, "**Denoising Diffusion Implicit Models**," ICLR2021.

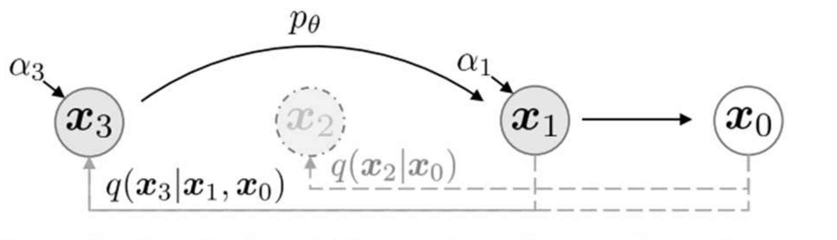


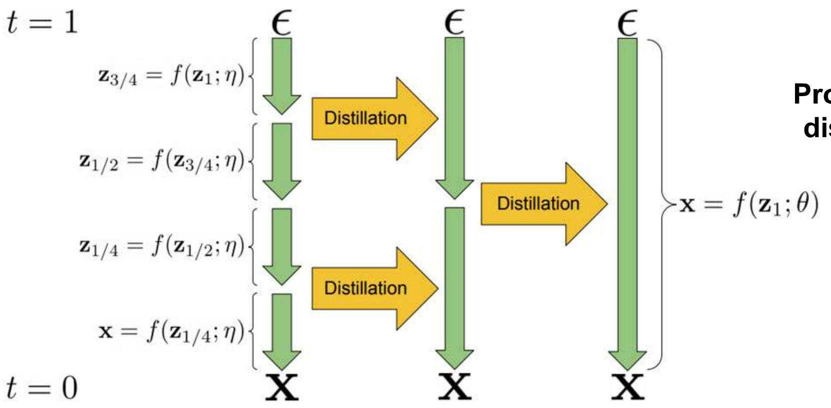
Figure 2: Graphical model for accelerated generation, where $\tau = [1, 3]$.

**DDIM**

✓ https://junia3.github.io/blog/consistency

- 여전히 DDIM을 포함하여 probability flow ODE의 경우에도 샘플링의 속도를 빠르게 하면 할수록 발생하는 샘플 퀄리티의 하락을 무시할 수 없다.
- 샘플링 단계를 최소화하면서 샘플링 성능의 저하를 막는 것이 주요 포인트인데, 이게 기존 방식으로 해결하기에는 벅차다. 그나마 distillation 방법이 probability flow ODE에 대해 좋은 Diffusion 모델의 성능을 transfer하기 좋은 방법이긴 하지만, 결국 Diffusion 모델의 생성에 의존해야한다는 점 때문에 학습 속도가 현저히 느려지게 된다는 bottleneck에서 벗어날 수 없다.

✓ Salimans, T. and Ho, J. **Progressive distillation for fast sampling of diffusion models**. ICLR 2022.



$t = 1$
$\epsilon$
$\epsilon$
$\epsilon$
$\mathbf{z}_{3/4} = f(\mathbf{z}_1; \eta)$
$\mathbf{z}_{1/2} = f(\mathbf{z}_{3/4}; \eta)$
$\mathbf{z}_{1/4} = f(\mathbf{z}_{1/2}; \eta)$
$\mathbf{x} = f(\mathbf{z}_{1/4}; \eta)$
Distillation
Distillation
Distillation
$\mathbf{x} = f(\mathbf{z}_1; \theta)$
$t = 0$
$\mathbf{X}$
$\mathbf{X}$
$\mathbf{X}$

**Progressive distillation**

2장 설명

# Consistency Models

## 1. Introduction

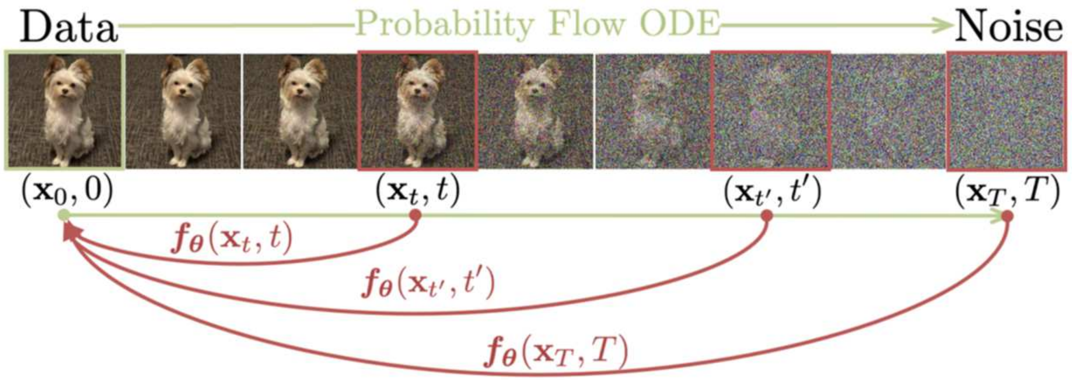


Figure 1: Given a **Probability Flow (PF) ODE** that **smoothly converts data to noise**, we learn to map any point (e.g., $\mathbf{x}_t$, $\mathbf{x}_{t'}$, and $\mathbf{x}_T$) on the *ODE trajectory* to its origin (e.g., $\mathbf{x}_0$) for generative modeling. Models of these mappings are called **consistency models**, as **their outputs are trained to be consistent for points on the same trajectory**.

- Build on top of the **probability flow (PF) ordinary differential equation (ODE)** in **continuous-time diffusion models** (Song et al., 2021), whose **trajectories smoothly transition the data distribution into a tractable noise distribution**.

- Propose **to learn a model that maps any point at any time step to the trajectory's starting point**.

- A notable property of our model is ***self-consistency***: ***points on the same trajectory map to the same initial point***. Refer to such models as **consistency models**.

- Consistency models allow us to **generate data samples** (**initial points of ODE trajectories**, e.g., $\mathbf{x}_0$ in Fig. 1) by **converting random noise vectors** (endpoints of ODE trajectories, e.g., $\mathbf{x}_T$ in Fig. 1) **with only one network evaluation**.

- Importantly, by **chaining the outputs of consistency models at multiple time steps**, we can **improve sample quality** and **perform zero-shot data editing** at the cost of more compute, similar to what iterative refinement enables for diffusion models.

# Consistency Models

## 1. Introduction

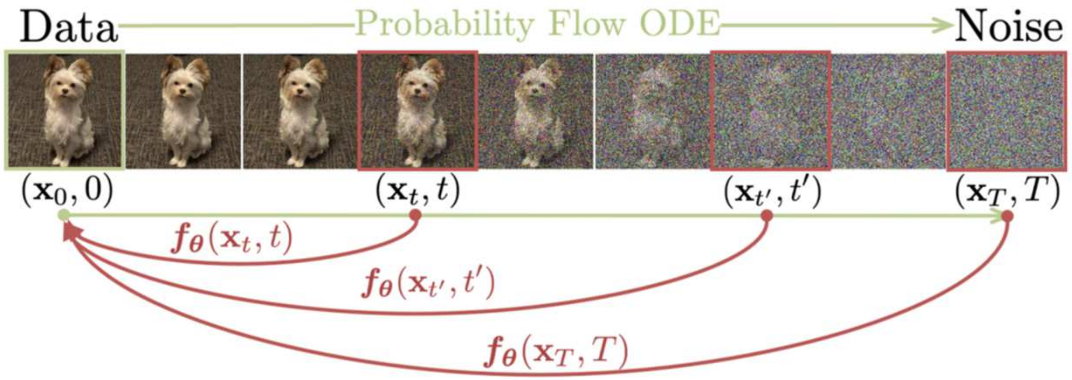


Figure 1: Given a **Probability Flow (PF) ODE** that **smoothly converts data to noise**, we learn to map any point (e.g., $\mathbf{x}_t$, $\mathbf{x}_{t'}$, and $\mathbf{x}_T$) on the *ODE trajectory* to its origin (e.g., $\mathbf{x}_0$) for generative modeling. Models of these mappings are called **consistency models**, as **their outputs are trained to be consistent for points on the same trajectory**.

- **To train a consistency model**, we offer **two methods based on enforcing the self-consistency property**.

1) First method : Relies on **using numerical ODE solvers and a pre-trained diffusion model** to **generate pairs of adjacent points on a PF ODE trajectory**.
   - By **minimizing the difference between model outputs for these pairs**, we can effectively **distill a diffusion model into a consistency model**, which allows generating **high-quality samples** with **one network evaluation**.

2) Second method : Eliminates the need for a pre-trained diffusion model altogether, allowing us to **train a consistency model in isolation**.
   - Situates **consistency models** as an **independent family of generative models**.

- Crucially, **neither approach requires adversarial training**, and both training methods permit **flexible neural network architectures** for consistency models.

# Consistency Models

## 2. Diffusion Models

- Consistency models are heavily inspired by the theory of (continuous-time) diffusion models.

✓ 모든 diffusion process는 **marginal likelihood** $p(x_{0:T})$를 동일하게 가지는 **ODE**를 찾을 수 있다.

✓ 원래의 Diffusion SDE는 식(1)

- Diffusion models start by diffusing $p_{\text{data}}(\mathbf{x})$ with a **stochastic differential equation (SDE)**

$$\mathrm{d}\mathbf{x}_t = \boldsymbol{\mu}(\mathbf{x}_t, t)\,\mathrm{d}t + \boxed{g(t)\,\mathrm{d}\mathbf{w}_t,} \qquad t \in [0, T] \qquad (1)$$

  - ✓ $p_{data}(\mathbf{x})$ : Data distribution
  - ✓ $\boldsymbol{\mu}(\cdot,\cdot)$ and $g(\cdot)$ : ***Drift*** and ***diffusion*** coefficients
  - ✓ $\{\mathbf{w}_t\}_{t\in[0,T]}$ : Standard Brownian motion

- $p_t(\mathbf{x})$ : Distribution of $\mathbf{x}_t$ and as a result $p_0(\mathbf{x}) \equiv p_{\text{data}}(\mathbf{x})$.

✓ Song, Y., Sohl-Dickstein, J., Kingma, D. P., Kumar, A., Ermon, S., and Poole, B. "**Score-based generative modeling through stochastic differential equations**," ICLR, 2021.

✓ Song, Y., Shen, L., Xing, L., and Ermon, S. "**Solving inverse problems in medical imaging with score-based generative models**," ICLR, 2022.

✓ Karras, T., Aittala, M., Aila, T., and Laine, S. "**Elucidating the design space of diffusion-based generative models**," Proc. NeurIPS, 2022

✓ SDE (1)와 동일한 **marginal likelihood**를 가지는 **ODE**는 식(2)와 같다.

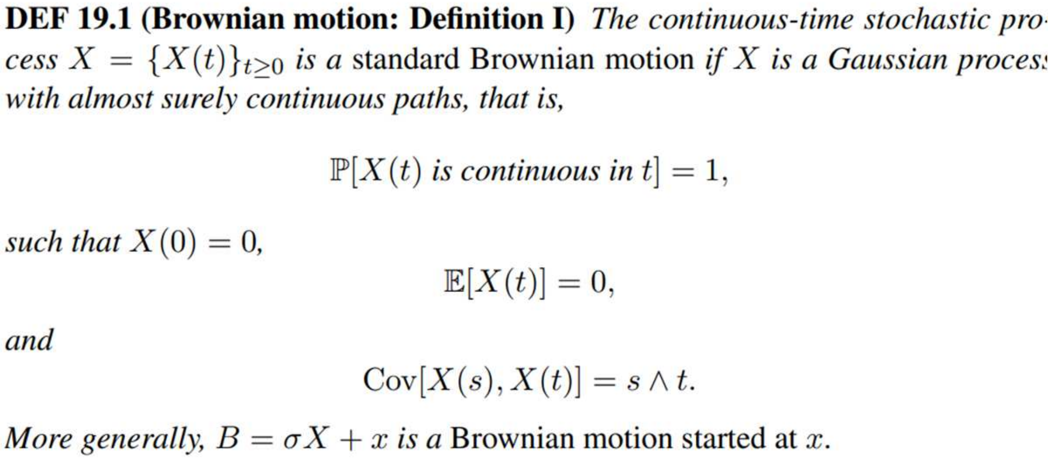
- Remarkable property of SDE; Ordinary differential equation (ODE), dubbed ***the Probability Flow (PF) ODE*** by Song et al. (2021), whose solution trajectories sampled at $t$ are distributed according to $p_t(\mathbf{x})$ :

$$\mathrm{d}\mathbf{x}_t = \left[\boldsymbol{\mu}(\mathbf{x}_t, t)\boxed{-\frac{1}{2}g(t)^2\underline{\nabla \log p_t(\mathbf{x}_t)}}\right]\mathrm{d}t. \qquad (2)$$

  - ✓ $\nabla \log p_t(\mathbf{x})$ : *Score functions* of $p_t(\mathbf{x})$
  - ✓ Diffusion models are known as *score-based generative models*.

# Consistency Models

- **Standard Brownian Motion**



**DEF 19.1 (Brownian motion: Definition I)** *The continuous-time stochastic process $X = \{X(t)\}_{t\geq 0}$ is a* standard Brownian motion *if $X$ is a Gaussian proces with almost surely continuous paths, that is,*

$$\mathbb{P}[X(t) \text{ is continuous in } t] = 1,$$

*such that $X(0) = 0$,*

$$\mathbb{E}[X(t)] = 0,$$

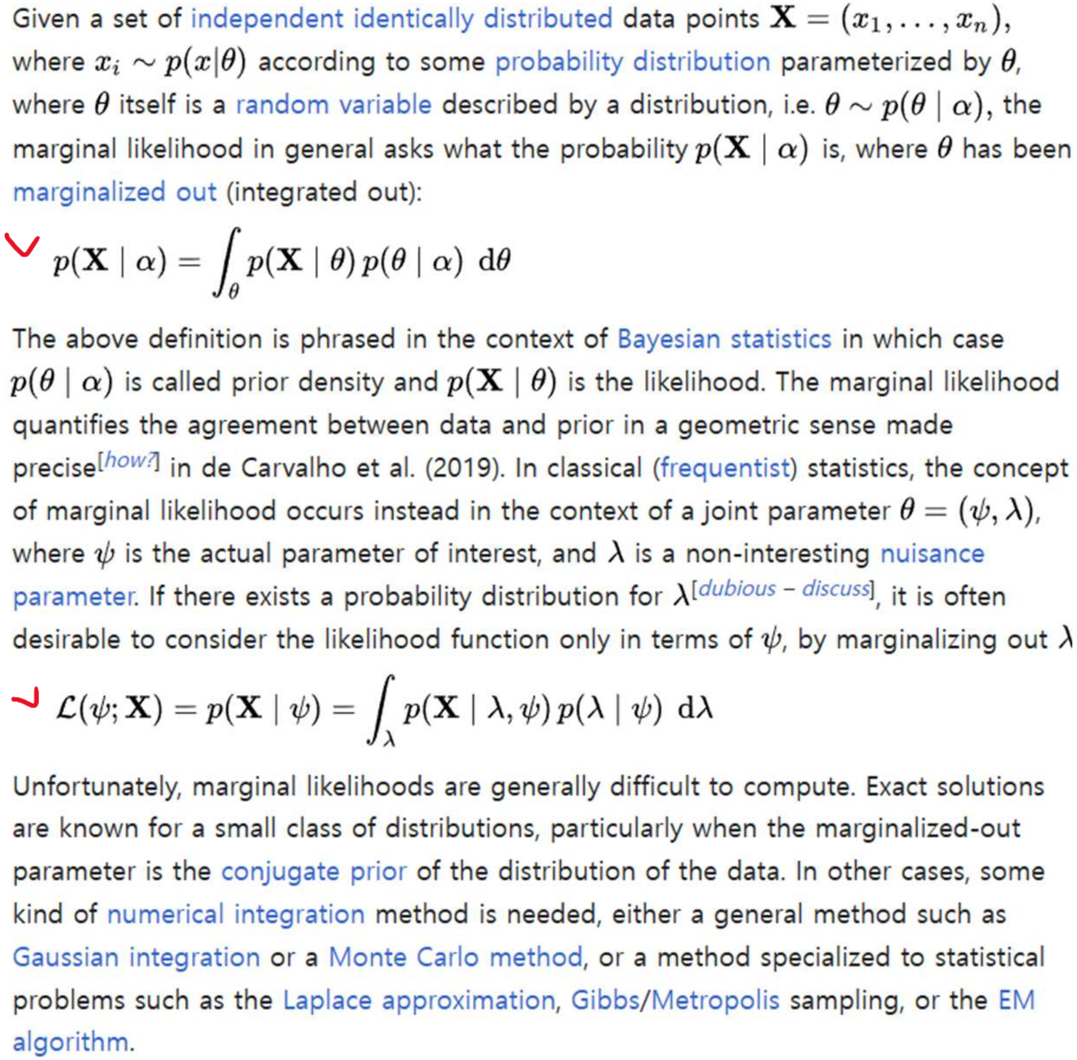
*and*

$$\mathrm{Cov}[X(s), X(t)] = s \wedge t.$$

*More generally, $B = \sigma X + x$ is a* Brownian motion started at $x$.

✓ https://people.math.wisc.edu/~roch/teaching_files/275b.1.12w/lect18-web.pdf

- **Marginal likelihood** ✓ https://en.wikipedia.org/wiki/Marginal_likelihood



Given a set of independent identically distributed data points $\mathbf{X} = (x_1, \ldots, x_n)$, where $x_i \sim p(x|\theta)$ according to some probability distribution parameterized by $\theta$, where $\theta$ itself is a random variable described by a distribution, i.e. $\theta \sim p(\theta \mid \alpha)$, the marginal likelihood in general asks what the probability $p(\mathbf{X} \mid \alpha)$ is, where $\theta$ has been marginalized out (integrated out):

$$p(\mathbf{X} \mid \alpha) = \int_\theta p(\mathbf{X} \mid \theta)\, p(\theta \mid \alpha)\ \mathrm{d}\theta$$

The above definition is phrased in the context of Bayesian statistics in which case $p(\theta \mid \alpha)$ is called prior density and $p(\mathbf{X} \mid \theta)$ is the likelihood. The marginal likelihood quantifies the agreement between data and prior in a geometric sense made precise[how?] in de Carvalho et al. (2019). In classical (frequentist) statistics, the concept of marginal likelihood occurs instead in the context of a joint parameter $\theta = (\psi, \lambda)$, where $\psi$ is the actual parameter of interest, and $\lambda$ is a non-interesting nuisance parameter. If there exists a probability distribution for $\lambda$[dubious – discuss], it is often desirable to consider the likelihood function only in terms of $\psi$, by marginalizing out $\lambda$

$$\mathcal{L}(\psi; \mathbf{X}) = p(\mathbf{X} \mid \psi) = \int_\lambda p(\mathbf{X} \mid \lambda, \psi)\, p(\lambda \mid \psi)\ \mathrm{d}\lambda$$

Unfortunately, marginal likelihoods are generally difficult to compute. Exact solutions are known for a small class of distributions, particularly when the marginalized-out parameter is the conjugate prior of the distribution of the data. In other cases, some kind of numerical integration method is needed, either a general method such as Gaussian integration or a Monte Carlo method, or a method specialized to statistical problems such as the Laplace approximation, Gibbs/Metropolis sampling, or the EM algorithm.

# Consistency Models

## 2. Diffusion Models

- Diffusion models start by diffusing $p_{\text{data}}(x)$ with a **stochastic differential equation (SDE)**

$$\mathrm{d}\mathbf{x}_t = \boldsymbol{\mu}(\mathbf{x}_t, t)\,\mathrm{d}t + \boxed{g(t)\,\mathrm{d}\mathbf{w}_t} \qquad t \in [0, T] \qquad (1)$$

- Typically, the **SDE** in Eq. (1) is designed such that $p_T(\mathbf{x})$ is close to a tractable Gaussian distribution $\pi(\mathbf{x})$.
- We hereafter adopt the configurations in Karras et al. (2022), who set $\boldsymbol{\mu}(\mathbf{x}, t) = 0$ and $g(t) = \sqrt{2t}$.
- In this case, we have $p_t(\mathbf{x}) = p_{\text{data}}(\mathbf{x}) \otimes \mathcal{N}(\mathbf{0}, t^2\boldsymbol{I})$, where $\otimes$ denotes the convolution operation, and $\pi(\mathbf{x}) = \mathcal{N}(\mathbf{0}, T^2\boldsymbol{I})$.

- For sampling, we first train a *score model* $\boldsymbol{s}_\phi(\mathbf{x}, t) \approx \nabla \log p_t(\mathbf{x}_t)$ via *score matching*, then plug it into Eq. (2) to obtain an *empirical estimate* of the PF ODE,

- ***Probability Flow (PF) ODE*** by Song et al. (2021), whose solution trajectories sampled at $t$ are distributed according to $p_t(\mathbf{x})$ :

$$\mathrm{d}\mathbf{x}_t = \left[\boldsymbol{\mu}(\mathbf{x}_t, t) - \frac{1}{2} g(t)^2 \underline{\nabla \log p_t(\mathbf{x}_t)}\right] \mathrm{d}t. \qquad (2)$$

$\boldsymbol{\mu}(\mathbf{x}, t) = 0$, $g(t) = \sqrt{2t}$ in Karras et al.

$\boldsymbol{s}_\phi(\mathbf{x}, t) \approx \nabla \log p_t(\mathbf{x}_t)$

- ***Empirical PF ODE***

$$\frac{\mathrm{d}\mathbf{x}_t}{\mathrm{d}t} = -t\boldsymbol{s}_\phi(\mathbf{x}_t, t). \qquad (3)$$

- Next, we sample $\hat{\mathbf{x}}_T \sim \pi = \mathcal{N}(\mathbf{0}, T^2\boldsymbol{I})$ to initialize the *empirical PF ODE* and solve it backwards in time with any *numerical ODE solver* to obtain the **solution trajectory** $\{\hat{\mathbf{x}}_t\}_{t\in[0,T]}$.
- The resulting $\hat{\mathbf{x}}_0$ can be viewed as an approximate sample from the data distribution $p_{\text{data}}(\mathbf{x})$.
- To avoid numerical instability, one typically stops the solver at $t = \epsilon$, where $\epsilon$ is a fixed small positive number, and instead accepts $\hat{\mathbf{x}}_\epsilon$ as the approximate sample.

11페이지 설명

# Consistency Models

## 2. Diffusion Models

✓ https://junia3.github.io/blog/consistency

***Stochastic differential equation (SDE)***

$$\mathrm{d}\mathbf{x}_t = \boldsymbol{\mu}(\mathbf{x}_t, t)\,\mathrm{d}t + \boxed{g(t)\,\mathrm{d}\mathbf{w}_t} \qquad t \in [0, T] \tag{1}$$

***Probability Flow (PF) ODE***

$$\mathrm{d}\mathbf{x}_t = \left[\boldsymbol{\mu}(\mathbf{x}_t, t) - \frac{1}{2}g(t)^2 \nabla \log p_t(\mathbf{x}_t)\right]\mathrm{d}t. \tag{2}$$

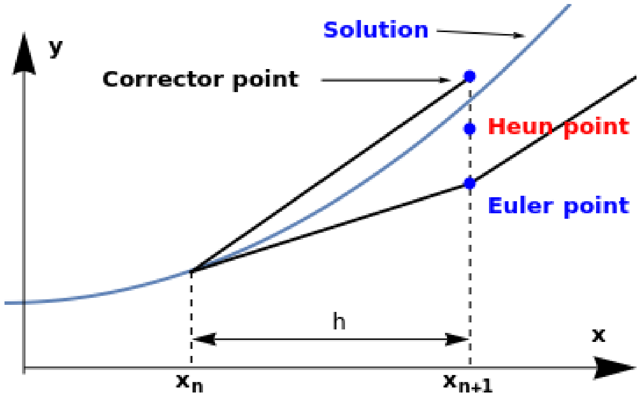
$\boldsymbol{\mu}(\mathbf{x}, t) = 0,\ g(t) = \sqrt{2t}$ in Karras et al.

$\boldsymbol{s}_\phi(\mathbf{x}, t) \approx \nabla \log p_t(\mathbf{x}_t)$

***Empirical PF ODE***

$$\frac{\mathrm{d}\mathbf{x}_t}{\mathrm{d}t} = -t\boldsymbol{s}_\phi(\mathbf{x}_t, t). \tag{3}$$

- **ODE**로 변형했을 때 **SDE**에 대해 가지는 장점은 stochastic한 diffusion coefficient ($d\mathbf{w}_t$)를 가지지 않아서, Probability flow ODE를 기준으로 starting point ($\mathbf{x}_0$)를 잡는다면 미분 방정식의 solution이 그리는 trajectory를 따라가는 $\mathbf{x}_T$ 까지의 모든 점 $\mathbf{x}_t$에 대해 하나의 선으로 이을 수 있게 된다 (그림 참고).

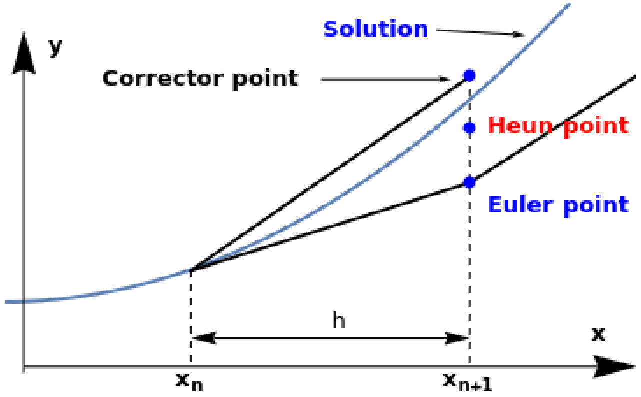


- SDE는 drift term($\boldsymbol{\mu}(\cdot,\cdot)$)이 방향만 정해줄 뿐, 실질적으로 뻗어나가는 구조는 랜덤한 요소가 좌우하므로 starting point와 ending point만 알 뿐, 그 내부에서 각각의 $\mathbf{x}_t$가 서로 교차하고 얽히는 과정을 알 수 없으므로, 1대1 mapping이 불가능하다는 단점이 있다.

- ODE의 경우에는 trajectory를 그리는 요소에 시간축이라는 단일 변수가 관여할 수 있게 된다

- 만약 특정 시점의 데이터인 $\mathbf{x}_t$에 대해 score를 예측할 수 있는 모델인 $\boldsymbol{s}_\phi(\mathbf{x}, t) \approx \nabla \log p_t(\mathbf{x})$가 있다면, 위의 방정식은 perturbation kernel $p_t(\mathbf{x}) = p_{\text{data}}(\mathbf{x}) \otimes \mathcal{N}(\mathbf{0}, t^2\boldsymbol{I})$에 대해 식(3)과 같은 form으로 나타낼 수 있다.

## 2. Diffusion Models



- 만약 $\mathbf{x}_T \sim \pi = \mathcal{N}(\mathbf{0}, T^2 \boldsymbol{I})$를 정의하고 이에 따른 probability flow ODE (식(3))를 풀어낸다면, $\mathbf{x}_0$과 $\mathbf{x}_T$를 잇는 하나의 trajectory를 구할 수 있게 된다.
- 미분 방정식을 푸는 방식은 Euler나 Heun solver와 같은 numerical 방법을 통해 함수의 형상을 예측하는 형태가 된다

$$\frac{d\mathbf{x}_t}{dt} = -t\boldsymbol{s}_\phi(\mathbf{x}, t) \text{ where } \boldsymbol{s}_\phi(\mathbf{x}, t) \approx \nabla \log p_t(\mathbf{x}_t)$$

- 실제로 Analytic하게 풀어낼 수 없는(정해진 solution이 없는) 미분 방정식을 마주했을때, 아주 작은 변수의 변화에 대한 함숫값 변화를 예측하는 과정을 의미한다.
- 그러나 그림을 보면 알 수 있듯이 실제로 numerical하게 풀어낸 미분 방정식의 해는 실제 solution과 오차가 클 수 밖에 없으며, 이는 시간 축이 길어지면 길어질수록, 샘플링 간격이 늘어나면 늘어날수록 variance가 높아진다.

$$\hat{\mathbf{x}}_T, t \in (0, T)$$

- 논문에서는 numerical instability를 보완할 목적으로 $t \in \epsilon(0.002)$ 의 위치에서의 solution을 실제 데이터 샘플인 $\mathbf{x}_0$에 근사한 값으로 간주했으며, time step의 총 수는 $T = 80$ 을 사용함

# Consistency Models

## 2. Diffusion Models

- Diffusion models are bottlenecked by their **slow sampling speed**.
- Clearly, **using ODE solvers for sampling** requires **many evaluations of the score model** $\boldsymbol{s}_{\phi}(\mathbf{x}, t)$.
- Existing methods for fast sampling include *faster numerical ODE solvers* and *distillation techniques*.
  - ✓ ***ODE solvers*** : Still need more than 10 evaluation steps to generate competitive samples - 꽤 좋은 퀄리티의 데이터를 생성하기 위해서는 **단일 step으로는 불가능**
  - ✓ ***Distillation*** *methods* : Rely on collecting a large dataset of samples from the diffusion model prior to distillation. - 보통 DDPM과 같은 Diffusion의 prior에 의존하게 되는데, 결국 **DDPM에서 각 time step에 대한 Noise 데이터를 샘플링** 해야하기 때문에 연산량이 부담
- To our best knowledge, the only distillation approach that does not suffer from this drawback is ***progressive distillation*** (PD, Salimans & Ho (2022))

✓ Salimans, T. and Ho, J. **Progressive distillation for fast sampling of diffusion models**. In International Conference on Learning Representations, 2022.

➢ **Progressive distillation** : 점진적으로 distillation을 수행하는 time step 수를 줄임

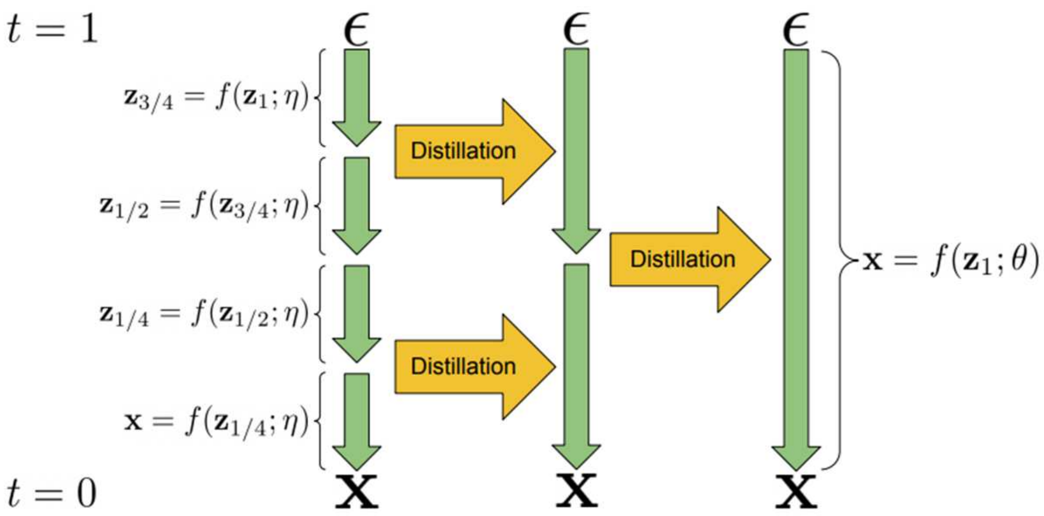

$t = 1$
$\epsilon$
$\epsilon$
$\epsilon$
$\mathbf{z}_{3/4} = f(\mathbf{z}_1; \eta)$
$\mathbf{z}_{1/2} = f(\mathbf{z}_{3/4}; \eta)$
$\mathbf{z}_{1/4} = f(\mathbf{z}_{1/2}; \eta)$
$\mathbf{x} = f(\mathbf{z}_{1/4}; \eta)$
Distillation
Distillation
Distillation
$\mathbf{x} = f(\mathbf{z}_1; \theta)$
$t = 0$
$\mathbf{x}$
$\mathbf{x}$
$\mathbf{x}$

Figure 1: A visualization of two iterations of our proposed *progressive distillation* algorithm. A sampler $f(\mathbf{z}; \eta)$, mapping random noise $\epsilon$ to samples $\mathbf{x}$ in 4 deterministic steps, is distilled into a new sampler $f(\mathbf{z}; \theta)$ taking only a single step. The original sampler is derived by approximately integrating the *probability flow ODE* for a learned diffusion model, and distillation can thus be understood as learning to integrate in fewer steps, or *amortizing* this integration into the new sampler.

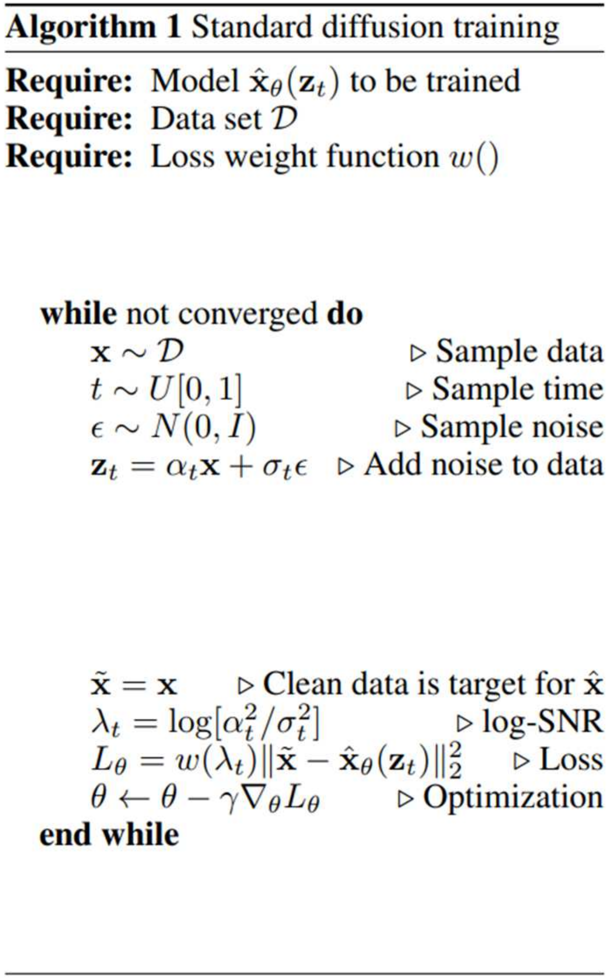
## Consistency Models

### 2. Diffusion Models

✓ Salimans, T. and Ho, J. **Progressive distillation for fast sampling of diffusion models**. In International Conference on Learning Representations, 2022.

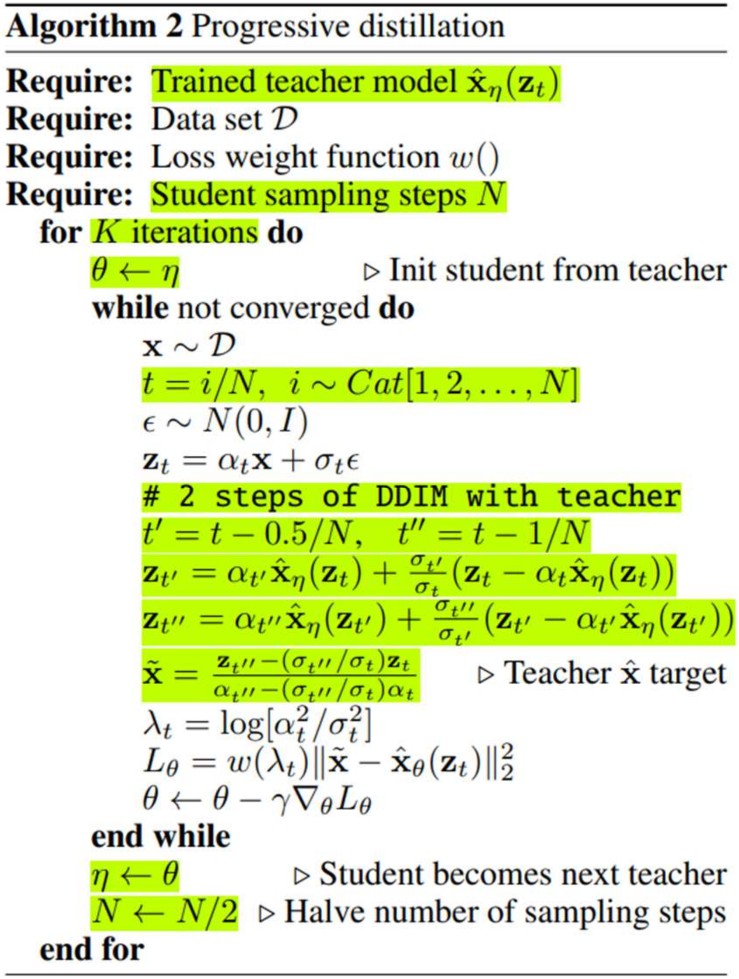
➢ **Progressive distillation**

- 처음부터 단일 **trajectory**을 모두 학습하려면 **score** 예측에 필요한 샘플이 그만큼 늘어나게 된다.
- 하지만 만약 여러 **trajectory**에 대해 부분적으로 학습된 **ODE score estimator**가 서로 연결되게끔 **distillation** 하면서 그 수를 줄여나가면, 굳이 처음부터 엄청난 수의 샘플을 사용하지 않고도 충분히 좋은 성능을 보일 수 있다는 것이 그 방법이다.
- 본 논문에서는 위와 같이 **progressive distillation**을 사용하지는 않지만 **consistency distillation**을 사용하여 **ODE solver**에 대한 예측과 **prior**에 대한 예측을 일치시키는 작업을 진행하게 된다.



**Algorithm 1** Standard diffusion training

```
Require: Model $\hat{\mathbf{x}}_\theta(\mathbf{z}_t)$ to be trained
Require: Data set $\mathcal{D}$
Require: Loss weight function $w()$

while not converged do
    $\mathbf{x} \sim \mathcal{D}$                          ▷ Sample data
    $t \sim U[0,1]$                        ▷ Sample time
    $\epsilon \sim N(0, I)$                       ▷ Sample noise
    $\mathbf{z}_t = \alpha_t \mathbf{x} + \sigma_t \epsilon$    ▷ Add noise to data

    $\tilde{\mathbf{x}} = \mathbf{x}$      ▷ Clean data is target for $\hat{\mathbf{x}}$
    $\lambda_t = \log[\alpha_t^2 / \sigma_t^2]$             ▷ log-SNR
    $L_\theta = w(\lambda_t) \|\tilde{\mathbf{x}} - \hat{\mathbf{x}}_\theta(\mathbf{z}_t)\|_2^2$     ▷ Loss
    $\theta \leftarrow \theta - \gamma \nabla_\theta L_\theta$          ▷ Optimization
end while
```



**Algorithm 2** Progressive distillation

```
Require: Trained teacher model $\hat{\mathbf{x}}_\eta(\mathbf{z}_t)$
Require: Data set $\mathcal{D}$
Require: Loss weight function $w()$
Require: Student sampling steps $N$
  for $K$ iterations do
      $\theta \leftarrow \eta$                 ▷ Init student from teacher
      while not converged do
          $\mathbf{x} \sim \mathcal{D}$
          $t = i/N, \ \ i \sim Cat[1, 2, \ldots, N]$
          $\epsilon \sim N(0, I)$
          $\mathbf{z}_t = \alpha_t \mathbf{x} + \sigma_t \epsilon$
          # 2 steps of DDIM with teacher
          $t' = t - 0.5/N, \quad t'' = t - 1/N$
          $\mathbf{z}_{t'} = \alpha_{t'} \hat{\mathbf{x}}_\eta(\mathbf{z}_t) + \frac{\sigma_{t'}}{\sigma_t}(\mathbf{z}_t - \alpha_t \hat{\mathbf{x}}_\eta(\mathbf{z}_t))$
          $\mathbf{z}_{t''} = \alpha_{t''} \hat{\mathbf{x}}_\eta(\mathbf{z}_{t'}) + \frac{\sigma_{t''}}{\sigma_{t'}}(\mathbf{z}_{t'} - \alpha_{t'} \hat{\mathbf{x}}_\eta(\mathbf{z}_{t'}))$
          $\tilde{\mathbf{x}} = \frac{\mathbf{z}_{t''} - (\sigma_{t''}/\sigma_t)\mathbf{z}_t}{\alpha_{t''} - (\sigma_{t''}/\sigma_t)\alpha_t}$     ▷ Teacher $\hat{\mathbf{x}}$ target
          $\lambda_t = \log[\alpha_t^2 / \sigma_t^2]$
          $L_\theta = w(\lambda_t) \|\tilde{\mathbf{x}} - \hat{\mathbf{x}}_\theta(\mathbf{z}_t)\|_2^2$
          $\theta \leftarrow \theta - \gamma \nabla_\theta L_\theta$
      end while
      $\eta \leftarrow \theta$            ▷ Student becomes next teacher
      $N \leftarrow N/2$      ▷ Halve number of sampling steps
  end for
```

## 3. Consistency Models

- Consistency model, A new type generative models
  - ✓ Support **single-step generation** at the core design
  - ✓ While still allowing **iterative generation** for **zero-shot data editing** and **trade-offs between sample quality and compute**.

- Trained in either the distillation mode or the isolation mode.
  - ✓ ***Disitllation mode*** : Distill the knowledge of pre-trained diffusion models into a *single-step sampler*, significantly improving other distillation approaches in sample quality while allowing zero-shot image editing applications.
  - ***Isolation mode*** : With no dependence on pre-trained diffusion models. Makes them an independent new class of generative models.

- Consistency model은 diffusion process의 SDE를 기반으로 하는 **probability flow ODE**를 수학적 접근 프레임으로 삼는데, 이때 ODE를 풀어가는 방식에 만약 굳이 사전 학습된 DDPM에 의한 distillation이 불필요하게 된다면 이는 곧 scratch 부터 학습될 수 있는 새로운 생성 모델의 기본이 되는 것이다.

# Consistency Models

## 3. Consistency Models

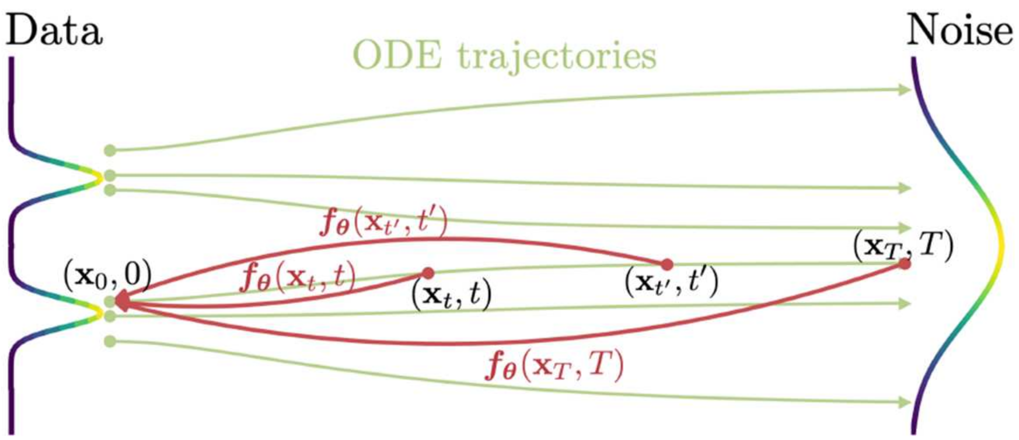

Data
ODE trajectories
Noise
$\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}_{t'}, t')$
$(\mathbf{x}_T, T)$
$(\mathbf{x}_0, 0)$
$\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}_t, t)$
$(\mathbf{x}_t, t)$
$(\mathbf{x}_{t'}, t')$
$\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}_T, T)$

Figure 2: **Consistency models** are trained to map points on any trajectory of the **PF ODE** to the trajectory's origin.

ODE Trajectories(초록색)이 시간축 상에서 ϵ부터 T까지 뻗어있는 PF ODE의 솔루션 궤도이며, 모든 시간축 상의 점들을 태초마을로 귀환시켜버리는 것이 논문에서 학습시키고자 하는 consistency model의 주된 목적이다.

즉, consistency model은 ODE를 통해 궤도를 예측하면서 남은 발자국의 출발점을 똑같은 곳인 $\mathbf{x}_\epsilon$으로 보내는 과정이 된다

### 3.1) Definition

- Given a solution trajectory $\{\mathbf{x}_t\}_{t\in[\epsilon,T]}$ of the PF ODE in Eq. (2),

$$\mathrm{d}\mathbf{x}_t = \left[\boldsymbol{\mu}(\mathbf{x}_t, t) - \frac{1}{2}\sigma(t)^2 \nabla \log p_t(\mathbf{x}_t)\right] \mathrm{d}t.$$

- Define the *consistency function* as

$$f : \{\mathbf{x}_t, t\} \mapsto \boxed{\mathbf{x}_\epsilon.}$$

  ✓ Property; ***Self-consistency***

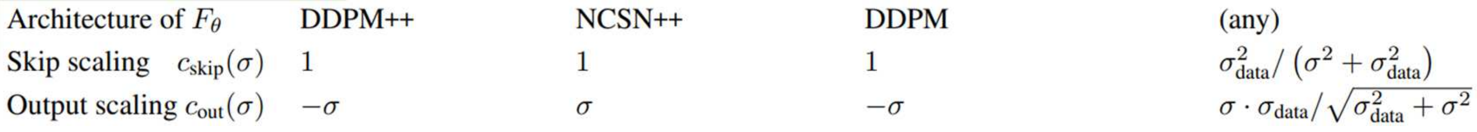
  its outputs are consistent for arbitrary pairs of $\{\mathbf{x}_t, t\}$ that belong to the same PF ODE trajectory, i.e., $f(\mathbf{x}_t, t)$=$f(\mathbf{x}_{t'}, t')$ for all $t, t' \in [\epsilon, T]$

  ✓ With fixed time argument, $f(\cdot, t)$ is always an invertible function.

- As illustrated in Fig. 2, the goal of a consistency model, symbolized as $\boldsymbol{f}_{\boldsymbol{\theta}}$, is to estimate this consistency function $\boldsymbol{f}$ from data by learning to enforce the *self-consistency* property.

# Consistency Models

## 3. Consistency Models



| Architecture of $F_\theta$ | DDPM++ | NCSN++ | DDPM | (any) |
|---|---|---|---|---|
| Skip scaling $c_{\text{skip}}(\sigma)$ | 1 | 1 | 1 | $\sigma_{\text{data}}^2 / \left(\sigma^2 + \sigma_{\text{data}}^2\right)$ |
| Output scaling $c_{\text{out}}(\sigma)$ | $-\sigma$ | $\sigma$ | $-\sigma$ | $\sigma \cdot \sigma_{\text{data}} / \sqrt{\sigma_{\text{data}}^2 + \sigma^2}$ |

### 3.2) Parameterization

- *Boundary condition* : For consistency function $f(\cdot,\cdot)$; $f(\cdot,\epsilon)$ is an identity function.

$$f(\mathbf{x}_\epsilon, \epsilon) = \mathbf{x}_\epsilon$$

- Discuss *two ways* to implement the boundary condition for consistency models
  - ✓ $F_\theta(\mathbf{x}, t)$ : a free-form deep neural network, Output has the same dimensionality as $\mathbf{x}$.
  - ✓ First way : Simply parameterize the consistency model as

$$\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}, t) = \begin{cases} \mathbf{x} & t = \epsilon \\ F_{\boldsymbol{\theta}}(\mathbf{x}, t) & t \in (\epsilon, T] \end{cases}. \quad (4)$$

  - ✓ Second way : Parameterize the consistency model using *skip connections*

$$\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}, t) = c_{\text{skip}}(t)\mathbf{x} + c_{\text{out}}(t) F_{\boldsymbol{\theta}}(\mathbf{x}, t), \quad (5)$$

$c_{\text{skip}}(t)$ and $c_{\text{out}}(t)$ are differentiable functions such that $c_{\text{skip}}(\epsilon) = 1$, and $c_{\text{out}}(\epsilon) = 0$

- The consistency model is differentiable at $t = \epsilon$ if $F_\theta(\mathbf{x}, t)$ and scaling coefficients are differentiable, which is critical for training continuous-time consistency models

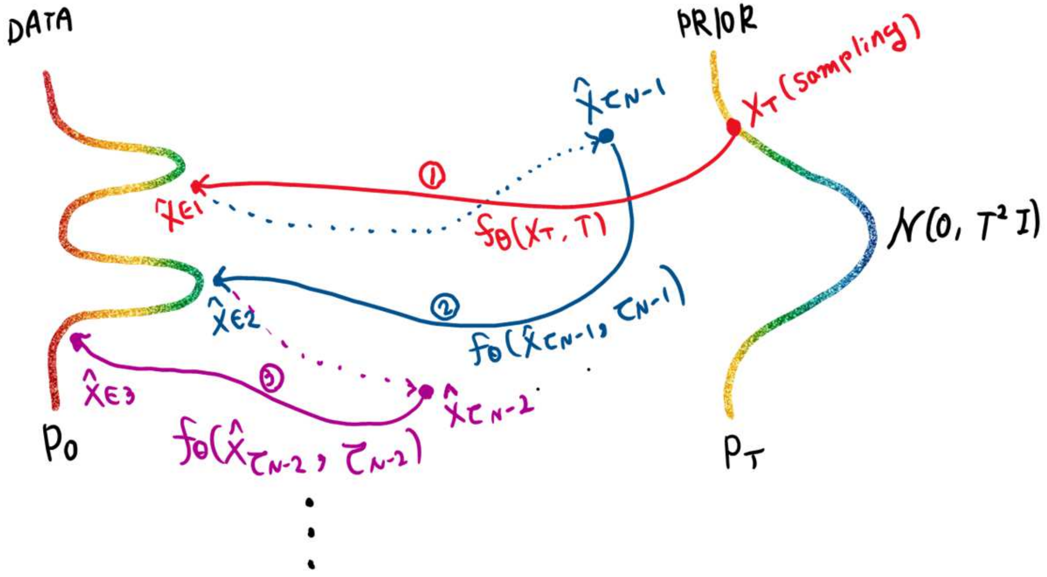
- Bears strong *resemblance* to many successful diffusion models, making it easier to borrow powerful diffusion model architecture for constructing consistency models

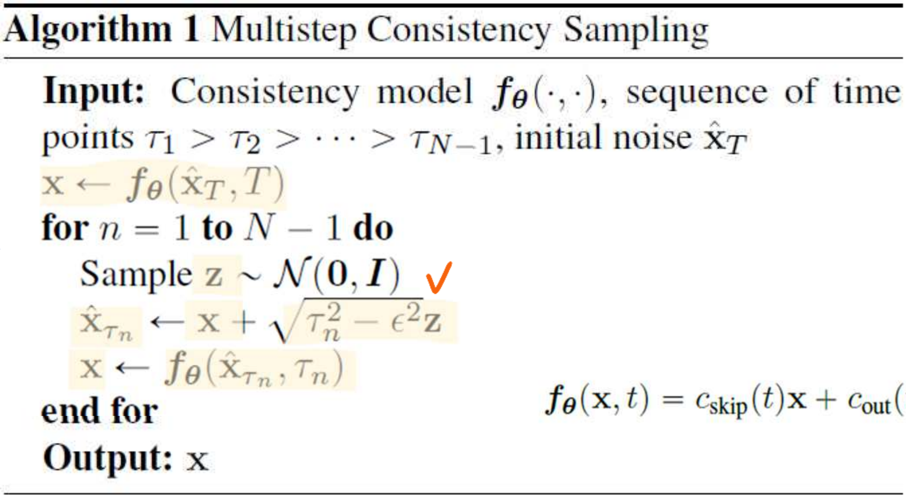
## 3. Consistency Models

### 3.3) Sampling

- Single Step Sampling
  - With a well-trained consistency model $f_\theta(\cdot,\cdot)$, we can generate samples by
    - ✓ Sampling from the initial distribution $\hat{\mathrm{x}}_T \sim N(0, T^2\mathbf{I})$
    - ✓ Evaluating the consistency model for $\hat{\mathrm{x}}_\epsilon \sim f_\theta(\hat{\mathrm{x}}_T, T)$
- One can also evaluate the consistency model *multiple times* by alternating **denoising** and **noise injection** steps for improved sample quality.
- Multistep Sampling
  - Provide the flexibility to trade computing for sample quality
  - Zero-shot data editing
  - 태초 마을로 데려갔다가 다시 Noise를 더했다가 하는 과정을 반복

- We find **time points** in Algorithm 1 with a greedy algorithm, where the time points are pinpointed one at a time using **ternary search to optimize the FID of samples** obtained from Algorithm 1





**Algorithm 1** Multistep Consistency Sampling

```
Input: Consistency model $f_\theta(\cdot,\cdot)$, sequence of time points $\tau_1 > \tau_2 > \cdots > \tau_{N-1}$, initial noise $\hat{\mathbf{x}}_T$
$\mathbf{x} \leftarrow f_\theta(\hat{\mathbf{x}}_T, T)$
for $n = 1$ to $N - 1$ do
    Sample $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ ✓
    $\hat{\mathbf{x}}_{\tau_n} \leftarrow \mathbf{x} + \sqrt{\tau_n^2 - \epsilon^2}\mathbf{z}$
    $\mathbf{x} \leftarrow f_\theta(\hat{\mathbf{x}}_{\tau_n}, \tau_n)$
end for
Output: $\mathbf{x}$
```

$$f_\theta(\mathbf{x}, t) = c_{\text{skip}}(t)\mathbf{x} + c_{\text{out}}(t)F_\theta(\mathbf{x}, t),$$

## 3. Consistency Models

✓ https://junia3.github.io/blog/consistency

### 3.4) Zero-Shot Data Editing

- Enable **various data editing** and **manipulation applications** in **zero shot**; Do not require explicit training.

- Define a **one-to-one mapping** from a **Gaussian noise vector** to a **data sample**. Similar to latent variable models like GANs, VAEs, and normalizing flows, consistency models can easily **interpolate between samples by traversing the latent space** (Fig. 11)

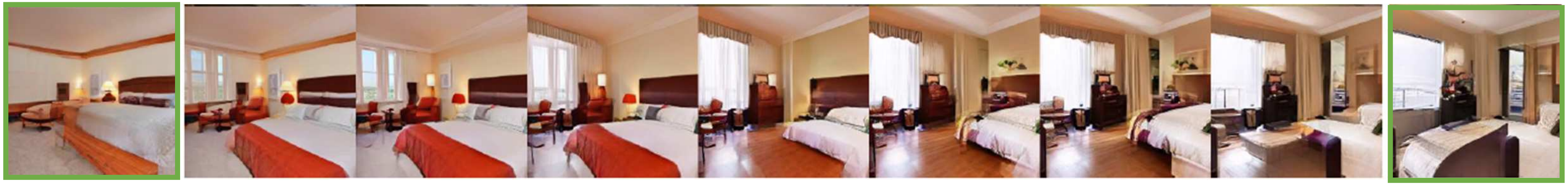
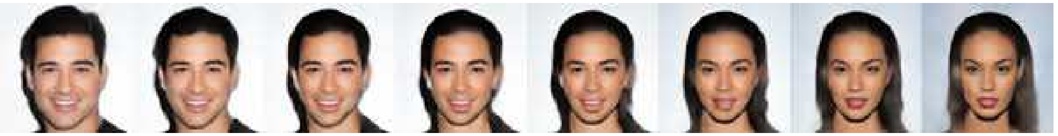


Fig. 11. Interpolating between leftmost and rightmost images with spherical linear interpolation. All samples are generated by a consistency model trained on LSUN Bedroom 256x256.

## 3. Consistency Models

### 3.4) Zero-Shot Data Editing

- Consistency model의 특징(prior를 기준으로 data와 대응되는 궤도 상의 어떤 점에서 출발하더라도 원래의 $\mathbf{x}_0$로 수렴하는 성질)을 사용하게 된다면 image editing이나 manipulation을 zero shot으로 수행할 수 있다.
- 가장 간단하게 생각해볼 수 있는 것은 GAN, VAE와 같은 latent variable model에서 할 수 있는 interpolation이다.

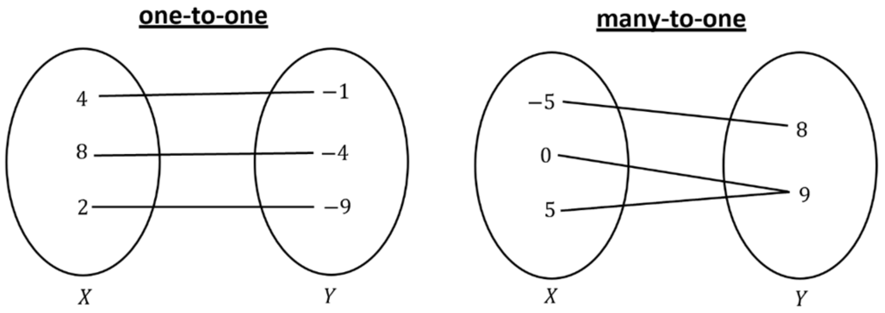


- Latent와 생성되는 sample이 parameter로 구성된 **implicit decoder**의 출력이 되는 GAN나 VAE의 경우에는 샘플 $\mathbf{x}_0$를 만들어내는 latent $\mathbf{z}_0$, 그리고 샘플 $\mathbf{x}_1$을 만들어내는 latent $\mathbf{z}_1$ 사이의 보간을 통해 중간 이미지 (Image ($\mathbf{x}_0$, $\mathbf{x}_1$))를 생성할 수 있고, 이는 곧 특징자 벡터를 자유롭게 사용하여 생성되는 이미지를 바꿀 수 있다는 장점이 된다.

$$F_\theta(\alpha \mathbf{z}_0 + (1-\alpha)\mathbf{z}_1) = \mathrm{Image}(\mathbf{x}_0, \mathbf{x}_1)$$

✓ https://junia3.github.io/blog/consistency

- 확률 미분 방정식에서의 diffusion process를 그대로 사용하는 **DDPM baseline**의 경우에 prior sample인 $\mathbf{x}_T$ 와 이에 대해 생성한 샘플 $F_{\theta_{1:T}}(\mathbf{x}_T) = \mathbf{x}_0$이 1대1 대응이 아니라는 점을 생각해보자. 하나의 latent sample $\mathbf{x}_T$가 포함된 Modality에서 이에 대응될 수 있는 dataset Modality 샘플 $\mathbf{x}_0^1, \cdots, \mathbf{x}_0^T$은 Markov process를 전제로 샘플링하기 때문에 latent interpolation이 image에서 유의미한 interpolation으로 이어지지 않는다는 문제가 있다.
- 그런데 이를 **consistency model**과 같이 **Probability flow ODE**의 **solution**에 대해 풀게 된다면 $\mathbf{x}_T$는 더이상 **data modallity**에 대해 **one to many mapping**이 아니게 된다.
- 따라서 GAN이 가지는 장점 중 **하나인 latent manipulation**을 **통한 이미지 manipulation**이 **용이**하다는 특징을 가져갈 수 있다.



## 3. Consistency Models

### 3.4) Zero-Shot Data Editing

- As consistency models are trained to recover $\mathbf{x}_\epsilon$ from any noisy input $\mathbf{x}_t\ t \in [\epsilon, T]$, they can perform **denoising for various noise levels** (Fig. 12)

- 추가적으로 sample의 modality와 더불어 condition이 들어가는 경우에도 zero-shot으로 사용할 수 있다는 장점이 발생한다. 예컨데 좋은 성능의 image inpainting, colorization 그리고 super-resolution 등을 수행할 수 있는 Diffusion 기반의 모델은 모두 해당 task에 대한 목적을 가지고 explicit 하게 학습이 전제되어야한다. 하지만 앞서 말했던 것과 같이 consistency model은 어떠한 수준의 noise에서도 $\mathbf{x}_\epsilon$을 복구할 수 있게끔 학습되기 때문에 여러 noise level에 대한 denoising이 가능하다.

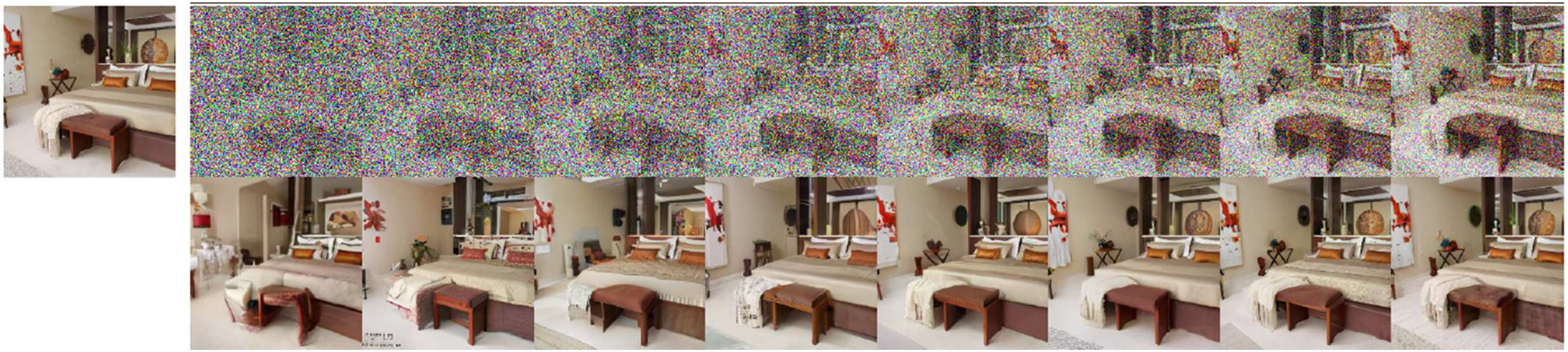


Fig. 12. Single-step denoising with a consistency model. The leftmost images are ground truth. For every two rows, the top row shows noisy images with different noise levels, while the bottom row gives denoised images.

## 3. Consistency Models

### 3.4) Zero-Shot Data Editing

- Multistep generation procedure in Algorithm 1
  - Uuseful for **solving certain inverse problems in zero shot** by using an **iterative replacement procedure** similar to that of diffusion models.
  - Enables many applications in the context of **image editing**, including **inpainting** (Fig. 10), **colorization** (Fig. 8), **super-resolution** (Fig. 6b) and **stroke-guided image editing** (Fig. 13) as in SDEdit (Meng et al., 2021).

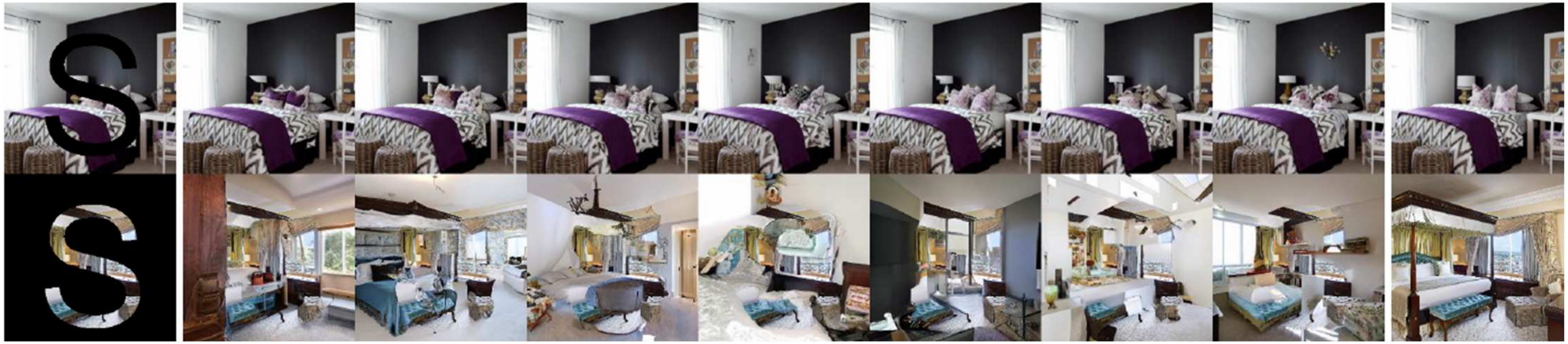
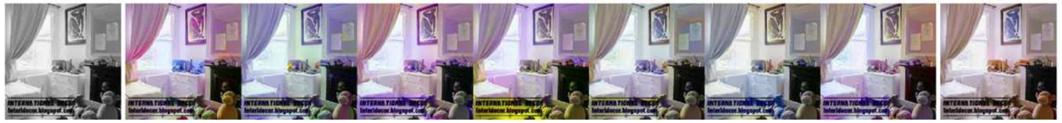


Fig. 10. Masked images (left), imputed images by a consistency model (middle), and ground truth (right).

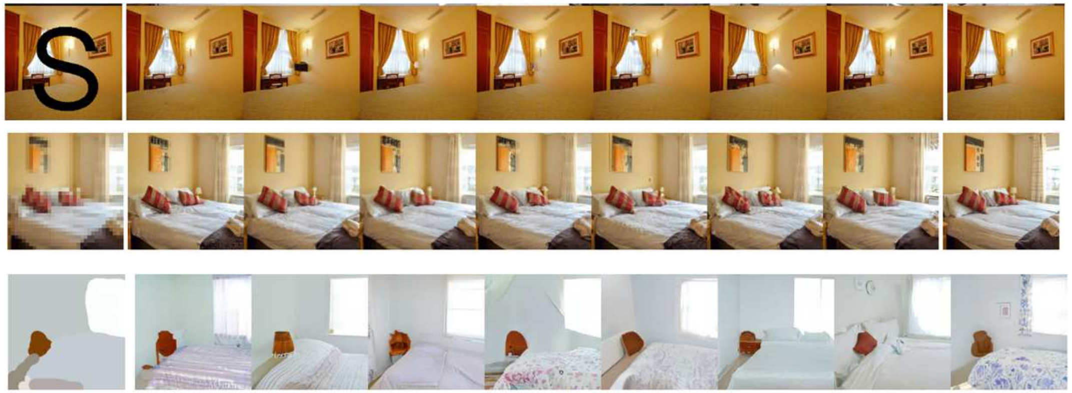
## 3. Consistency Models

### 3.4) Zero-Shot Data Editing

- 어떠한 input이 들어가더라도 **multiple step generation**을 수행하게 되면 임의의 input에 대해 그 시작점을 찾을 수 있게 되는 것이다 (condition이 들어갈 때는 단순히 prior sampling 부분만 skip하면 될 것 같음).
- 만약 input이 grey image라면 이를 consistency model에 대해 multistep(Noise를 더하고 $\mathbf{x}_0$를 예측하고를 수차례 반복)을 적용할 수 있다.



- 이렇듯 딱히 condition에 대해 따로 학습할 필요가 없다는 부분은 아래와 같이 inpainting, super-resolution 그리고 SDEdit(painting to image)와 같은 task에 자연스럽게 사용될 수 있다는 장점을 부여해준다.

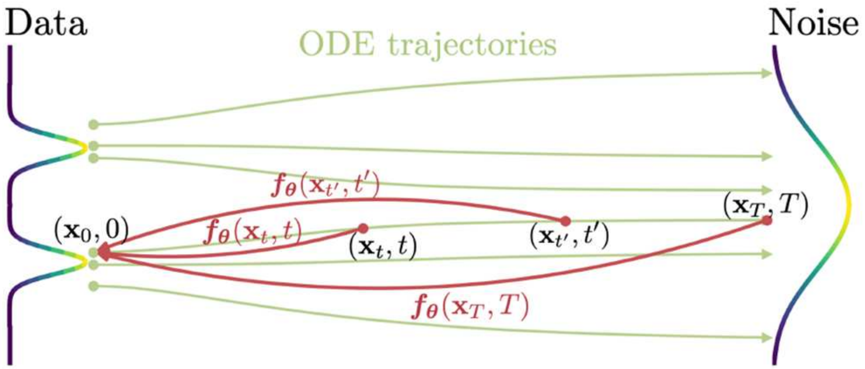


## 4. Training Consistency Models vis Distillation

- Method for training consistency models based on **distilling a pre-trained score model $s_\phi(\mathbf{x}, t)$.**

***Empirical PF ODE***

$$\frac{\mathrm{d}\mathbf{x}_t}{\mathrm{d}t} = -t s_\phi(\mathbf{x}_t, t). \qquad (3)$$



- Consider discretizing the time horizon $[\epsilon, T]$; Ts into $N-1$ sub-intervals, with boundaries $t_1 = \epsilon < t_2 < \cdots < t_N = T$
- In practice, follow Karras et al. (2022), boundaries

$$t_i = (\epsilon^{\frac{1}{\rho}} + \frac{i-1}{N-1}(T^{\frac{1}{\rho}} - \epsilon^{\frac{1}{\rho}}))^\rho,\ \rho=7$$

**Consistency Distillation (CD)**

***PF ODE***

$$\mathrm{d}\mathbf{x}_t = \left[\boldsymbol{\mu}(\mathbf{x}_t, t) - \frac{1}{2}\sigma(t)^2 \nabla \log p_t(\mathbf{x}_t)\right] \mathrm{d}t. \qquad (2)$$

PF ODE에서 실제 데이터 분포에 대한 socre를 구할 수 없으므로, 학습된 네트워크의 score prediction을 대입하면 식(3)과 같이 empirical PF-ODE를 문제로 가져올 수 있다.

- When $N$ **is sufficiently large**, obtain an **accurate estimate of $\mathbf{x}_{t_n}$ from $\mathbf{x}_{t_{n+1}}$ by running one discretization step of a numerical ODE solver.**
- Estimate $\hat{\mathbf{x}}^\phi_{t_n}$ (Solver가 예측한 특정 시점에서의 함수값)

$$\hat{\mathbf{x}}^\phi_{t_n} := \mathbf{x}_{t_{n+1}} + (t_n - t_{n+1})\Phi(\mathbf{x}_{t_{n+1}}, t_{n+1}; \phi), \qquad (6)$$

$\boldsymbol{\Phi}(\cdots; \boldsymbol{\phi})$ : Update function of a **one-step ODE solver** applied to the empirical PF ODE. ($\phi$ 가 ODE Solving에 관여하는 이유가 score estimator가 emprirical PF ODE를 풀고자 하며, 사전 학습된 score estimator를 사용할 것임을 알려줌)

Using the Euler solver, $\Phi(\mathbf{x}, t; \phi) = -s_\phi(\mathbf{x}, t)$

$$\hat{\mathbf{x}}^\phi_{t_n} = \mathbf{x}_{t_{n+1}} - (t_n - t_{n+1}) t_{n+1} s_\phi(\mathbf{x}_{t_{n+1}}, t_{n+1})$$

# Consistency Models

## 4. Training Consistency Models vis Distillation

- Due to the connection between the PF ODE in Eq. (2) and the SDE in Eq. (1), one can sample along the distribution of ODE trajectories by first sampling $\mathbf{x} \sim p_{\text{data}}$, then adding Gaussian noise to $\mathbf{x}$.

- Specifically, given a data point $\mathbf{x}$, we can generate a pair of adjacent data points $(\hat{\mathbf{x}}^{\phi}_{t_n}, \mathbf{x}_{t_{n+1}})$ on the PF ODE trajectory efficiently by sampling $\mathbf{x}$ from the dataset, followed by sampling $\mathbf{x}_{t_{n+1}}$ from *the transition density of the SDE* $\mathcal{N}(\mathbf{x}; t_{n+1}^2 \boldsymbol{I})$ and then computing $\hat{\mathbf{x}}^{\phi}_{t_n}$ using one discretization step of the numerical ODE solver according to Eq. (6).

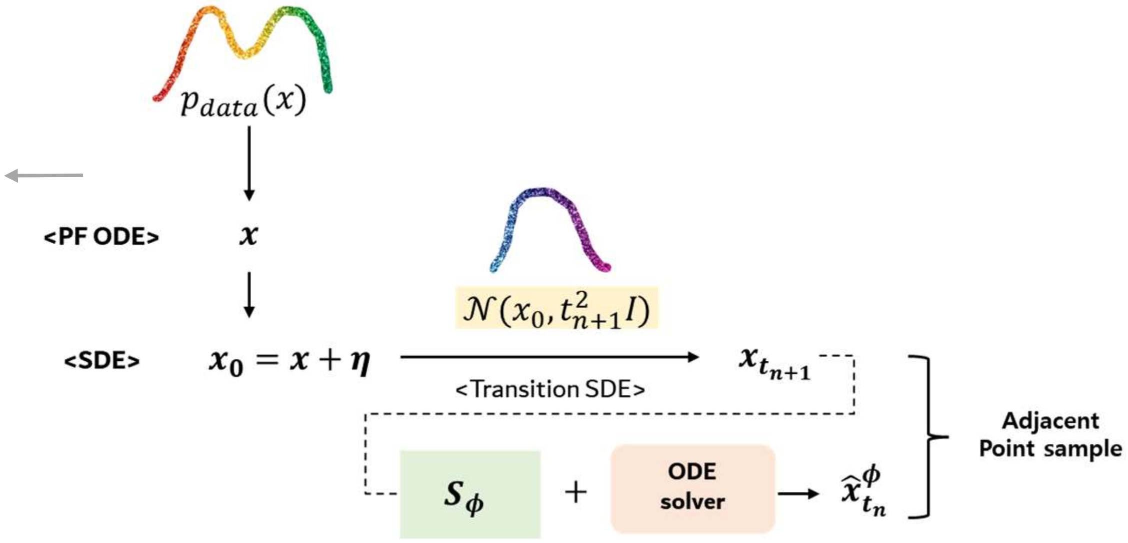
$$\hat{\mathbf{x}}^{\phi}_{t_n} = \mathbf{x}_{t_{n+1}} - (t_n - t_{n+1}) t_{n+1} s_{\phi}(\mathbf{x}_{t_{n+1}}, t_{n+1}) \qquad (6)$$

- Afterwards, we train the consistency model by **minimizing its output differences on the pair** $(\hat{\mathbf{x}}^{\phi}_{t_n}, \mathbf{x}_{t_{n+1}})$. This motivates our following ***consistency distillation loss*** for training consistency models.

### Consistency Distillation (CD)

사실 **SDE를 PF-ODE로** 바꾸면서 생기는 오차는 실제 **score estimate function** 과의 오차와 부합하게 된다. 이 부분에 대한 connection을 해주기 위해서 강제로 1 to many mapping을 만들어줄 수 있다.;

$\mathbf{x} \sim p_{\text{data}}$, $\mathbf{x} = \mathbf{x} + \eta$ *(Gaussian Noise)*



- 이렇게 샘플링한 adjacent point들에 대해 consistency network를 학습함

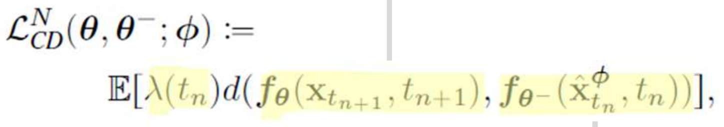
# Consistency Models

## 4. Training Consistency Models vis Distillation

## Consistency Distillation (CD)

➢ **Definition 1**

- The ***consistency distillation loss*** is defined as



$$\mathcal{L}_{CD}^{N}(\boldsymbol{\theta},\boldsymbol{\theta}^{-};\boldsymbol{\phi}) := \mathbb{E}[\lambda(t_n)d(\boldsymbol{f_\theta}(\mathbf{x}_{t_{n+1}},t_{n+1}),\boldsymbol{f_{\theta^-}}(\hat{\mathbf{x}}_{t_n}^{\phi},t_n))],$$

The expectation is taken with respect to $\mathbf{x}\sim p_{\text{data}}$, $n\sim\mathcal{U}[1,N-1]$, and $\mathbf{x}_{t_{n+1}}\sim\mathcal{N}(\mathbf{x};t_{n+1}^2\boldsymbol{I})$.

✓ $\mathcal{U}[1,N-1]$ : Uniform distribution over $\{1,2,\cdots,N-1\}$

✓ $\lambda(\cdot)\in\mathbb{R}^+$ : A positive weighting function, ($\lambda(t_n)=1$ performs well across all tasks and datasets); 시간에 따른 kenrel 분포 변화때문에 loss에 weight를 주기 위한 term

✓ $\theta^-$ : A running average of the past values of $\theta$ during the course of optimization

✓ $d(\cdot,\cdot)$ : A metric function that satisfies $\forall\mathbf{x},\mathbf{y}$: $d(\mathbf{x},\mathbf{y})\geqslant 0$ and $d(\mathbf{x},\mathbf{y})=0$ if and only if $\mathbf{x}=\mathbf{y}$.

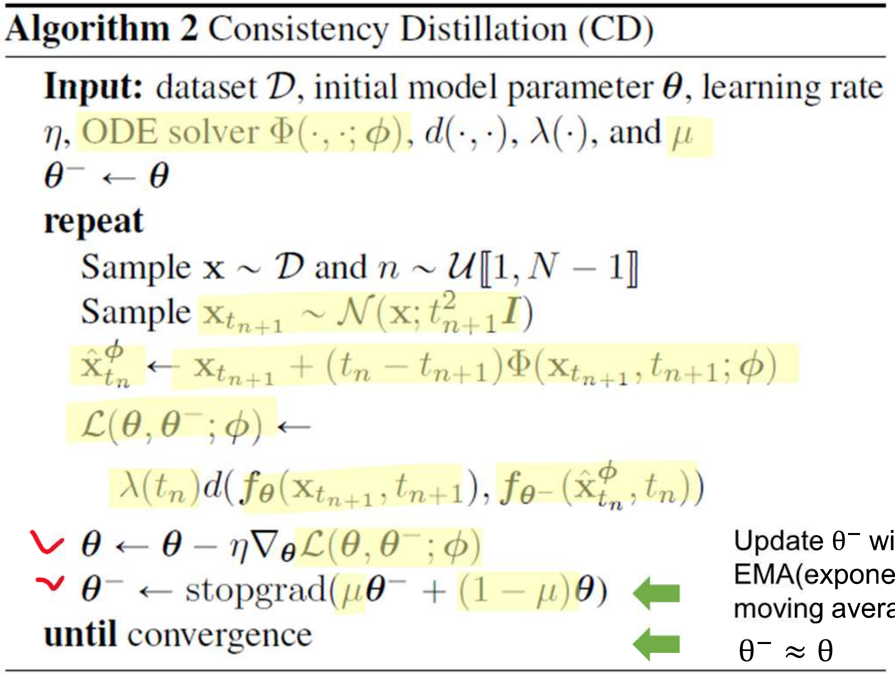
- 학습 방법은 간단하게 **두 인접한 sample point**(하나는 **forward SDE**에 따라 샘플링, 하나는 이렇게 샘플링된 것을 **score estimator**와 **numerical ODE solver**를 통해 궤도 예측)를 각각 네트워크에 통과한 결과가 서로 같도록 한다.

- 학습의 주체가 되는 $\boldsymbol{\theta}$가 **student parameter**로 loss에 대한 gradient descent를 받게 되고, $\boldsymbol{\theta}^-$는 **teacher parameter**로 student parameter를 EMA 방식으로 가져간다. 흔히 알고있는 distillation 방법이랑 동일하다.

- 거리 메트릭은 이것저것 다 가능한데 이미지 생성에 주로 사용되는 MSE, L1 그리고 LPIPS를 해당 논문에서는 모두 실험했으며, weight term인 $\lambda(\cdot)$ 는 심플하게 1로 고정해서 사용하는 것이 모든 task 및 dataset에 대해 괜찮은 성능을 보였다고 밝힌다

# Consistency Models

## 4. Training Consistency Models vis Distillation



```
Algorithm 2 Consistency Distillation (CD)
  Input: dataset $\mathcal{D}$, initial model parameter $\boldsymbol{\theta}$, learning rate
  $\eta$, ODE solver $\Phi(\cdot,\cdot;\boldsymbol{\phi})$, $d(\cdot,\cdot)$, $\lambda(\cdot)$, and $\mu$
  $\boldsymbol{\theta}^- \leftarrow \boldsymbol{\theta}$
  repeat
    Sample $\mathbf{x} \sim \mathcal{D}$ and $n \sim \mathcal{U}[\![1, N-1]\!]$
    Sample $\mathbf{x}_{t_{n+1}} \sim \mathcal{N}(\mathbf{x}; t_{n+1}^2 \boldsymbol{I})$
    $\hat{\mathbf{x}}_{t_n}^{\boldsymbol{\phi}} \leftarrow \mathbf{x}_{t_{n+1}} + (t_n - t_{n+1})\Phi(\mathbf{x}_{t_{n+1}}, t_{n+1}; \boldsymbol{\phi})$
    $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\theta}^-; \boldsymbol{\phi}) \leftarrow$
        $\lambda(t_n) d(\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}_{t_{n+1}}, t_{n+1}), \boldsymbol{f}_{\boldsymbol{\theta}^-}(\hat{\mathbf{x}}_{t_n}^{\boldsymbol{\phi}}, t_n))$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\theta}^-; \boldsymbol{\phi})$
    $\boldsymbol{\theta}^- \leftarrow \text{stopgrad}(\mu \boldsymbol{\theta}^- + (1-\mu)\boldsymbol{\theta})$
  until convergence
```

Update $\theta^-$ with EMA(exponential moving average)

$\theta^- \approx \theta$

### Consistency Distillation (CD)

- $f_\theta$ : 학습의 주체가 되는 $f_\theta$를 student network가 아닌 online(학습되는) network
- $f_{\theta^-}$ : EMA로 파라미터를 받는 $f_{\theta^-}$를 teacher network가 아닌 target(목적이 되는) network

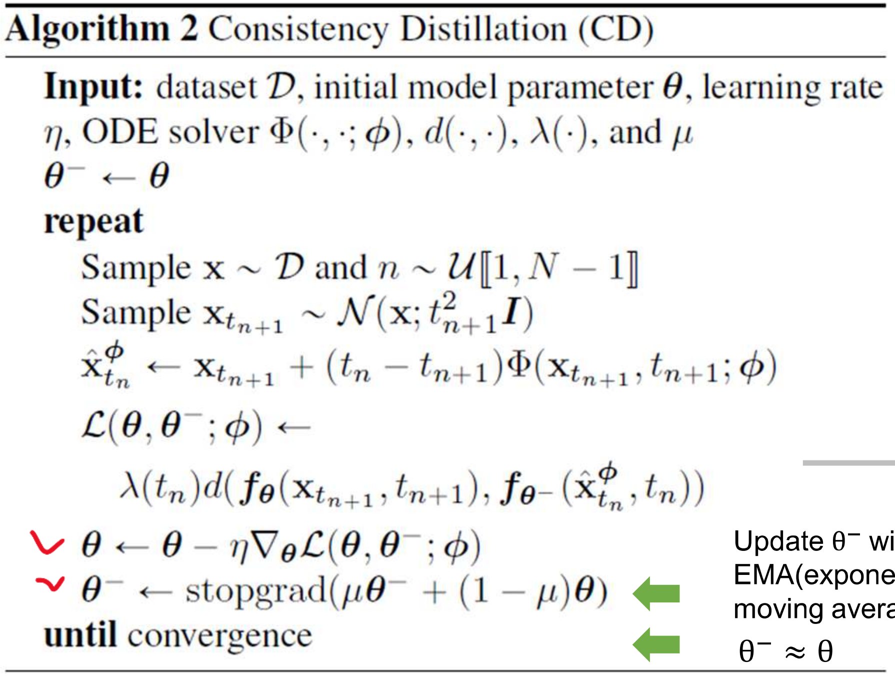
- ***EMA update*** and "***stopgrad***" operator can greatly stabilize the training process and improve the final performance of the consistency model
- Since $\theta^-$ is a running average of the history of $\theta$, we have $\theta^- = \theta$ when the optimization of Algorithm 2 converges

➢ The target and online consistency models will eventually match each other

- Consistency distillation loss는 무한히 증가하는 time step sample $N$에 대해 학습될 때 target과 online parameter를 같게 만들 수 있으며, 이는 곧 distillation의 주체가 되는 consistency network가 완벽하게 모든 정보를 이어받았다고 이해할 수 있다.

# Consistency Models

## 4. Training Consistency Models vis Distillation

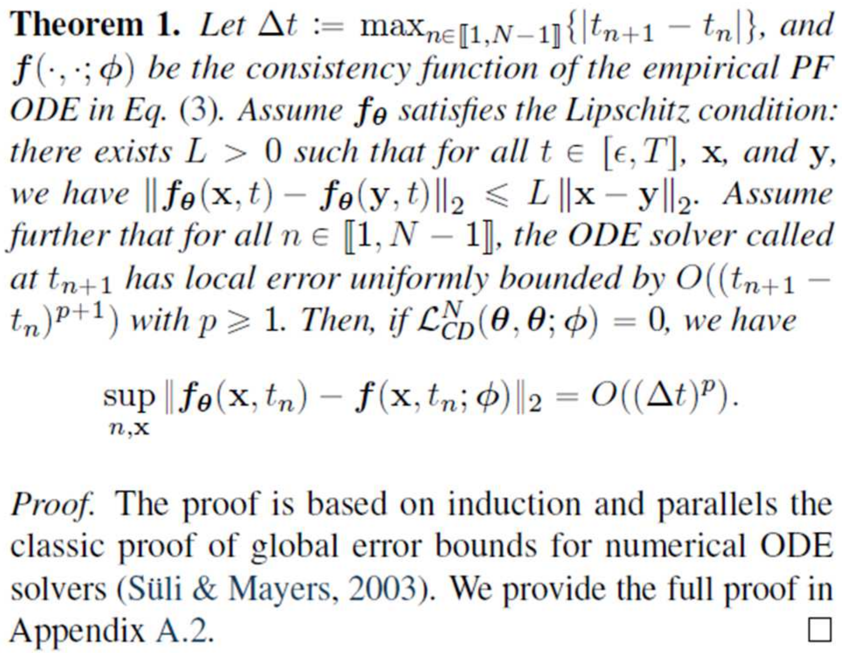


**Algorithm 2** Consistency Distillation (CD)

```
Input: dataset $\mathcal{D}$, initial model parameter $\boldsymbol{\theta}$, learning rate $\eta$, ODE solver $\Phi(\cdot, \cdot; \boldsymbol{\phi})$, $d(\cdot, \cdot)$, $\lambda(\cdot)$, and $\mu$
$\boldsymbol{\theta}^- \leftarrow \boldsymbol{\theta}$
repeat
    Sample $\mathbf{x} \sim \mathcal{D}$ and $n \sim \mathcal{U}[\![1, N-1]\!]$
    Sample $\mathbf{x}_{t_{n+1}} \sim \mathcal{N}(\mathbf{x}; t_{n+1}^2 \boldsymbol{I})$
    $\hat{\mathbf{x}}_{t_n}^{\boldsymbol{\phi}} \leftarrow \mathbf{x}_{t_{n+1}} + (t_n - t_{n+1}) \Phi(\mathbf{x}_{t_{n+1}}, t_{n+1}; \boldsymbol{\phi})$
    $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\theta}^-; \boldsymbol{\phi}) \leftarrow$
        $\lambda(t_n) d(\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}_{t_{n+1}}, t_{n+1}), \boldsymbol{f}_{\boldsymbol{\theta}^-}(\hat{\mathbf{x}}_{t_n}^{\boldsymbol{\phi}}, t_n))$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\theta}^-; \boldsymbol{\phi})$
    $\boldsymbol{\theta}^- \leftarrow \operatorname{stopgrad}(\mu \boldsymbol{\theta}^- + (1 - \mu) \boldsymbol{\theta})$
until convergence
```

Update $\theta^-$ with EMA(exponential moving average)

$\theta^- \approx \theta$

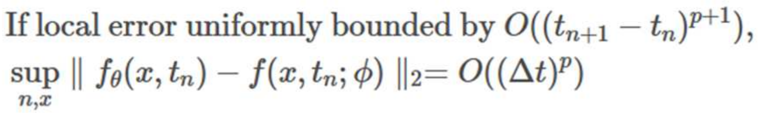


**Theorem 1.** *Let $\Delta t := \max_{n \in [\![1, N-1]\!]} \{|t_{n+1} - t_n|\}$, and $\boldsymbol{f}(\cdot, \cdot; \boldsymbol{\phi})$ be the consistency function of the empirical PF ODE in Eq. (3). Assume $\boldsymbol{f}_{\boldsymbol{\theta}}$ satisfies the Lipschitz condition: there exists $L > 0$ such that for all $t \in [\epsilon, T]$, $\mathbf{x}$, and $\mathbf{y}$, we have $\|\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}, t) - \boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{y}, t)\|_2 \leqslant L \|\mathbf{x} - \mathbf{y}\|_2$. Assume further that for all $n \in [\![1, N-1]\!]$, the ODE solver called at $t_{n+1}$ has local error uniformly bounded by $O((t_{n+1} - t_n)^{p+1})$ with $p \geqslant 1$. Then, if $\mathcal{L}_{\mathrm{CD}}^N(\boldsymbol{\theta}, \boldsymbol{\theta}; \boldsymbol{\phi}) = 0$, we have*

$$\sup_{n, \mathbf{x}} \|\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}, t_n) - \boldsymbol{f}(\mathbf{x}, t_n; \boldsymbol{\phi})\|_2 = O((\Delta t)^p).$$

*Proof.* The proof is based on induction and parallels the classic proof of global error bounds for numerical ODE solvers (Süli & Mayers, 2003). We provide the full proof in Appendix A.2. □

- **Numerical ODE**가 가지는 **bounded condition**(numerical하게 푼 solution이 실제 solution과 가지는 오차가 특정 범위 내에 존재한다는 가정)과 **consistency network** $f$가 가지는 **Lipshitz condition**을 만족한다는 조건 상에서 **loss function**의 **supremum 또한 수렴**한다는 증명을 할 수 있다. 이는 곧 **empirical PF ODE(Consistency model)**, 보다 엄밀히 말하자면 **consistency network가 distillation되는 실제 SDE 궤도에 따라 Numerical ODE와 함께 수렴이 가능**하다는 증거가 된다.



If local error uniformly bounded by $O((t_{n+1} - t_n)^{p+1})$,

$$\sup_{n,x} \| f_\theta(x, t_n) - f(x, t_n; \phi) \|_2 = O((\Delta t)^p)$$

# Consistency Models

## 5. Training Consistency Models in Isolation

- *Training in Distillation* : Consistency model를 **score network의 정보**와 **ODE solver**를 사용하여 어떤 식으로 consistency loss를 수렴시킬 수 있는지에 대해 증명하는 과정이었다.
- *Training in Isolation* : Consistency model이 기존 diffusion 방식에서 벗어난 **PF ODE 자체로의 가능성**을 보여주며, **새로운 생성 모델의 시작이라는 기준이 된 학습법**에 대해 언급하도록 한다.

## Consistency Training (CT)

- Distillation 방식의 경우 사전 학습된 diffusion process model이 필요하고, 이를 통해 score estimation $s_\phi(\mathrm{x}, t)$ 를 미분 방정식의 한 요소로 사용할 수 밖에 없었다.
- 만약 Consistency model을 단독으로 학습시키고자 한다면 해당 의존성을 없애버려야한다(아래의 식에서 $\nabla_x \log p_t(\mathrm{x})$ 를 구해야함).

***PF ODE***

$$dx_t = \left(\mu(x_t, t) - \frac{1}{2}\sigma(t)^2 \nabla_x \log p_t(x_t)\right) dt$$

➢ PD ODE에 Score estimator 없이 구하는 방법

- 구하고 싶은 **score**를 실제 data의 **marginal distribution**에 대해 역으로 projection하면 적분식이 나온다.

$$\nabla \log p_t(x_t) = \nabla_{x_t} \log \int p_{\text{data}}(x) p(x_t|x) dx$$

- log에 대한 미분은 closed form으로 정리된다.

$$\nabla \log p_t(x_t) = \frac{\int p_{\text{data}}(x) \nabla_{x_t} p(x_t|x) dx}{\int p_{\text{data}}(x) p(x_t|x) dx}$$

- 확률 분포 $p(x_t|x)$에 대한 미분은 log likelihood $log(p(x_t|x))$ 에 대한 미분으로 치환할 수 있고, 분모/분자 정리하면 다음과 같다.

$$\nabla \log p_t(x_t) = \frac{\int p_{\text{data}}(x) p(x_t|x) \nabla_{x_t} \log p(x_t|x) dx}{\int p_{\text{data}}(x) p(x_t|x) dx}$$

$$= \frac{\int p_{\text{data}}(x) p(x_t|x) \nabla_{x_t} \log p(x_t|x) dx}{p_t(x_t)}$$

- $x_t$는 $x$와 관련없으므로, 상수 취급

$$= \int \frac{p_{\text{data}}(x) p(x_t|x)}{p_t(x_t)} \nabla_{x_t} \log p(x_t|x) dx$$

# Consistency Models

## 5. Training Consistency Models in Isolation

Consistency Training (CT)

- Bayes' rule에 따라 조건부의 위치가 바뀌게 되고,

$$\begin{aligned}\nabla \log p_t(x_t) &= \int \frac{p_{\text{data}}(x)p(x_t|x)}{p_t(x_t)} \nabla_{x_t} \log p(x_t|x)dx \\ &= \int p(x|x_t)\nabla_{x_t} \log p(x_t|x)dx\end{aligned}$$

- 이는 $x_t$를 조건으로 하는 확률 분포에 따른 $x$에 대해 평균을 구하는 것과 같다.

$$\begin{aligned}\nabla \log p_t(x_t) &= \mathbb{E}(\nabla_{x_t} \log p(x_t|x)|x_t) \\ &= -\mathbb{E}\left(\frac{x_t - x}{t^2}|x_t\right)\end{aligned}$$

- 조건부 확률은 diffusion process에서 가우시안 커널로 정의가 됨

- 이처럼 근사시킬 수 있다. 물론 가지고 있는 샘플 내에서 평균을 구하는 과정이 되기 때문에 numerical error는 존재할 수 밖에 없다. 아무튼 이렇게 구한 score를 사용하게 되면 score estimation을 해주는 pre-trained network 없이 샘플링이 가능하고, 이 샘플들을 통해 consistency network 학습이 가능하다.

# Consistency Models

## 5. Training Consistency Models in Isolation

- Trained without relying on any pre-trained diffusion models.
- This differs from diffusion distillation techniques, **making consistency models a new independent family of generative models**

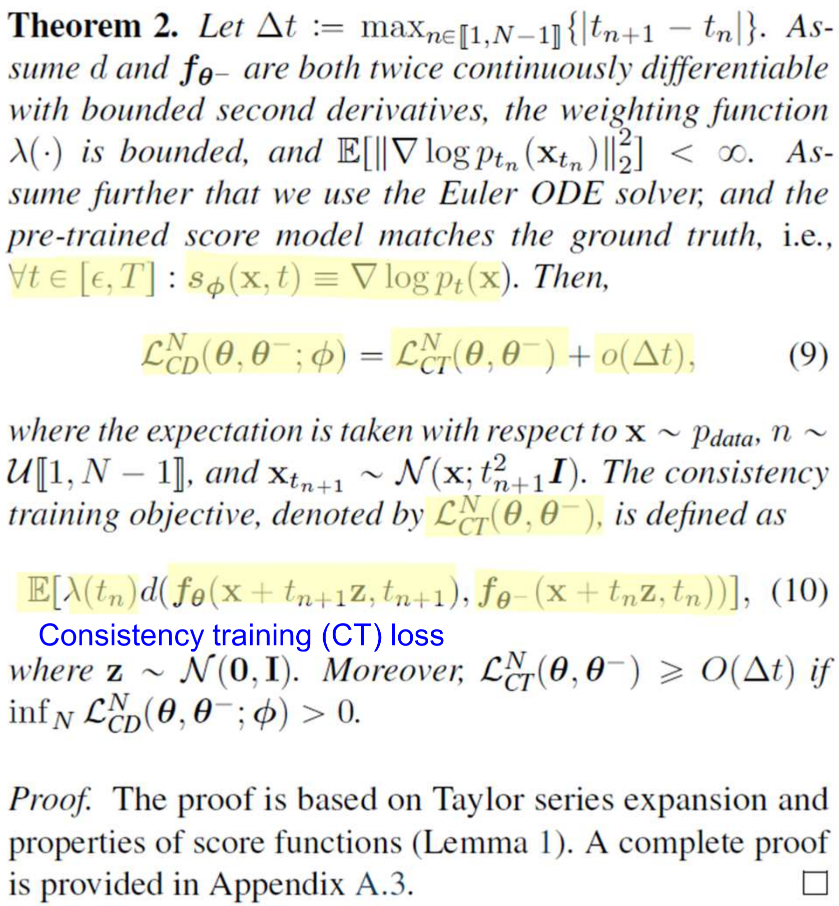
- Consistency distillation ; Use a pre-trained score model $s_\phi(\mathrm{x}, t)$ to approximate the ground truth score function $\nabla \log p_t(\mathrm{x})$
- To get rid of this dependency, we need to seek other ways to estimate the score function

- There exists an **unbiased estimator** of $\nabla \log p_t(\mathbf{x}_t)$ due to the following identity

$$\nabla \log p_t(\mathbf{x}_t) = -\mathbb{E}\left[\frac{\mathbf{x}_t - \mathbf{x}}{t^2} \,\middle|\, \mathbf{x}_t\right]$$

$\mathbf{x} \sim p_{\text{data}}$ and $\mathbf{x}_t \sim \mathcal{N}(\mathbf{x}; t^2 \boldsymbol{I})$

- Given $\mathbf{x}$ and $\mathbf{x}_t$, we can form a *Monte Carlo estimate* of $\nabla \log p_t(\mathbf{x}_t)$ with $-(\mathbf{x}_t - \mathbf{x})/t^2$.



**Theorem 2.** *Let $\Delta t := \max_{n \in [\![1, N-1]\!]}\{|t_{n+1} - t_n|\}$. Assume $d$ and $\boldsymbol{f}_{\boldsymbol{\theta}^-}$ are both twice continuously differentiable with bounded second derivatives, the weighting function $\lambda(\cdot)$ is bounded, and $\mathbb{E}[\|\nabla \log p_{t_n}(\mathbf{x}_{t_n})\|_2^2] < \infty$. Assume further that we use the Euler ODE solver, and the pre-trained score model matches the ground truth, i.e., $\forall t \in [\epsilon, T] : s_{\boldsymbol{\phi}}(\mathbf{x}, t) \equiv \nabla \log p_t(\mathbf{x})$. Then,*

$$\mathcal{L}_{CD}^{N}(\boldsymbol{\theta}, \boldsymbol{\theta}^-; \boldsymbol{\phi}) = \mathcal{L}_{CT}^{N}(\boldsymbol{\theta}, \boldsymbol{\theta}^-) + o(\Delta t), \qquad (9)$$

*where the expectation is taken with respect to $\mathbf{x} \sim p_{\text{data}}$, $n \sim \mathcal{U}[\![1, N-1]\!]$, and $\mathbf{x}_{t_{n+1}} \sim \mathcal{N}(\mathbf{x}; t_{n+1}^2 \boldsymbol{I})$. The consistency training objective, denoted by $\mathcal{L}_{CT}^{N}(\boldsymbol{\theta}, \boldsymbol{\theta}^-)$, is defined as*

$$\mathbb{E}[\lambda(t_n) d(\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x} + t_{n+1}\mathbf{z}, t_{n+1}), \boldsymbol{f}_{\boldsymbol{\theta}^-}(\mathbf{x} + t_n\mathbf{z}, t_n))], \qquad (10)$$

Consistency training (CT) loss

*where $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. Moreover, $\mathcal{L}_{CT}^{N}(\boldsymbol{\theta}, \boldsymbol{\theta}^-) \geqslant O(\Delta t)$ if $\inf_N \mathcal{L}_{CD}^{N}(\boldsymbol{\theta}, \boldsymbol{\theta}^-; \boldsymbol{\phi}) > 0$.*

*Proof.* The proof is based on Taylor series expansion and properties of score functions (Lemma 1). A complete proof is provided in Appendix A.3. □

- This estimate actually suffices to replace the pre-trained diffusion model in consistency distillation, when using the Euler method (or any higher order method) as the ODE solver in the limit of $N \to \infty$

# Consistency Models

## 5. Training Consistency Models in Isolation

**Consistency training (CT) loss**

$$\mathcal{L}_{CT}^{N}(\boldsymbol{\theta},\boldsymbol{\theta}^{-}) := \mathbb{E}[\lambda(t_n)d(\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}+t_{n+1}\mathbf{z},t_{n+1}),\boldsymbol{f}_{\boldsymbol{\theta}^{-}}(\mathbf{x}+t_n\mathbf{z},t_n))] \quad (10)$$

1) $N$ is small (i.e., $\Delta t$ is large) :

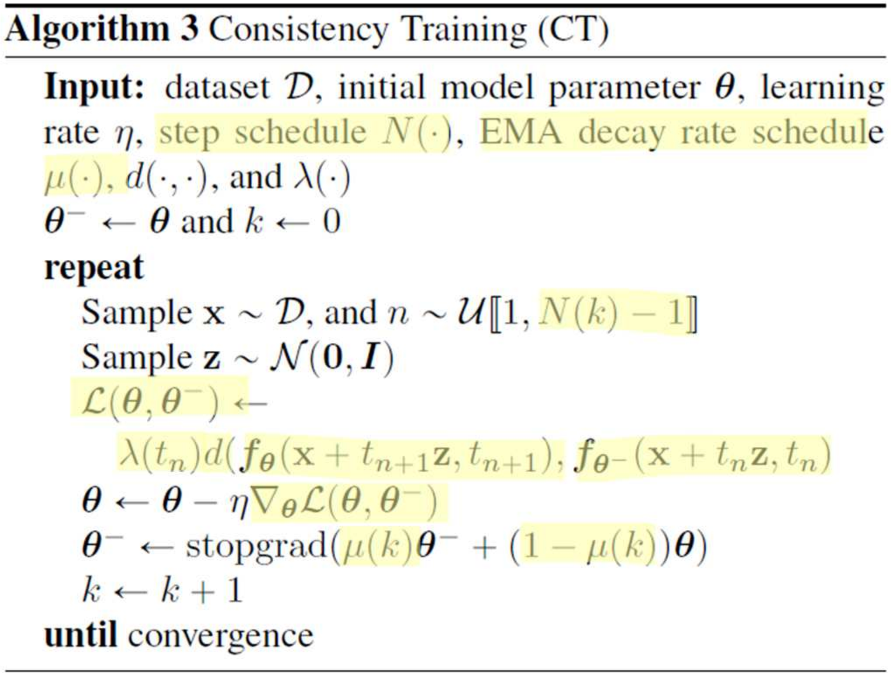
- CT loss has less "variance" but more "**bias**" with respect to the underlying consistency distillation loss
- Facilitates faster convergence at the beginning of training

2) $N$ is large (i.e., $\Delta t$ is small) :

- CT loss has more "variance" but less "bias"
- Desirable when closer to the end of training.

➢ For best performance, find that $\mu$ should change along with $N$, according to a schedule function $\mu(\cdot)$.

**Consistency Training (CT)**



**Algorithm 3** Consistency Training (CT)

```
Input: dataset $\mathcal{D}$, initial model parameter $\boldsymbol{\theta}$, learning rate $\eta$, step schedule $N(\cdot)$, EMA decay rate schedule $\mu(\cdot)$, $d(\cdot,\cdot)$, and $\lambda(\cdot)$
$\boldsymbol{\theta}^{-} \leftarrow \boldsymbol{\theta}$ and $k \leftarrow 0$
repeat
    Sample $\mathbf{x} \sim \mathcal{D}$, and $n \sim \mathcal{U}[\![1, N(k)-1]\!]$
    Sample $\mathbf{z} \sim \mathcal{N}(\mathbf{0},\boldsymbol{I})$
    $\mathcal{L}(\boldsymbol{\theta},\boldsymbol{\theta}^{-}) \leftarrow$
        $\lambda(t_n)d(\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}+t_{n+1}\mathbf{z},t_{n+1}),\boldsymbol{f}_{\boldsymbol{\theta}^{-}}(\mathbf{x}+t_n\mathbf{z},t_n)$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta},\boldsymbol{\theta}^{-})$
    $\boldsymbol{\theta}^{-} \leftarrow \mathrm{stopgrad}(\mu(k)\boldsymbol{\theta}^{-} + (1-\mu(k))\boldsymbol{\theta})$
    $k \leftarrow k+1$
until convergence
```

# Consistency Models

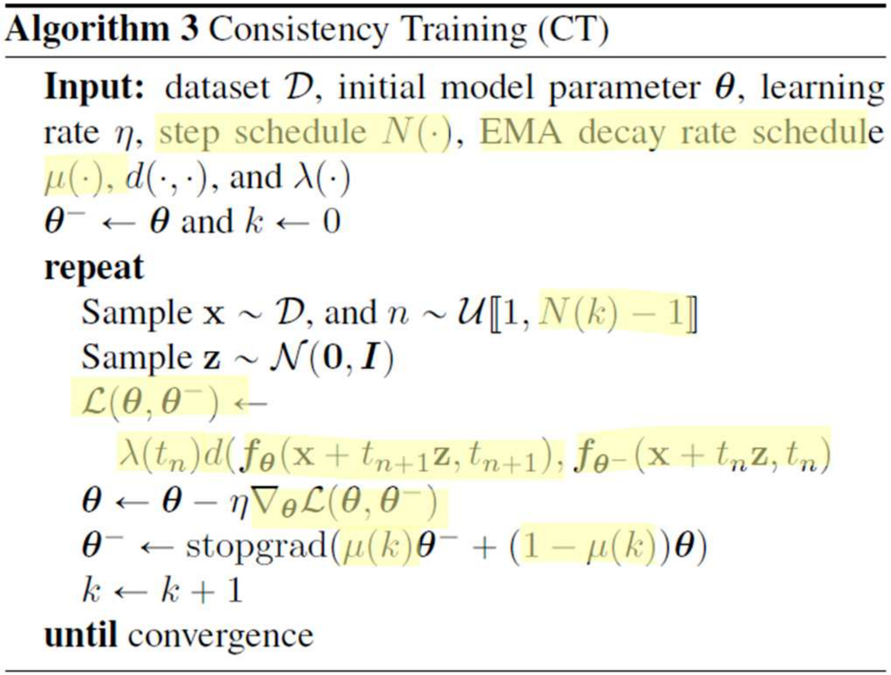
## 5. Training Consistency Models in Isolation

### Consistency Training (CT)

- 길게 증명과정이 있었지만 process는 간단하게도 인접 샘플들을 모두 사전 정의한 diffusion SDE에 따라 생성, 이를 사용하여 consistency model을 학습시키게 된다.
- 해당 process를 따르는 consistency network 학습 loss는 다변수 standard gaussian 변수 $z \sim \mathcal{N}(0, I)$에 대해 다음과 같이 변한다.

$$\mathcal{L}_{CT}^{N}(\theta, \theta^{-}) := \mathbb{E}(\lambda(t_n) d(f_\theta(x + t_{n+1} z, t_{n+1}),\ f_{\theta^-}(x + t_n z, t_n)))$$

- 해당 loss를 수렴시키는 과정이 distillation loss를 수렴시키는 것과 결과적으로 동일함을 증명할 수 있다 ☞ Appendix 2



```
Algorithm 3 Consistency Training (CT)
    Input: dataset $\mathcal{D}$, initial model parameter $\boldsymbol{\theta}$, learning
    rate $\eta$, step schedule $N(\cdot)$, EMA decay rate schedule
    $\mu(\cdot)$, $d(\cdot, \cdot)$, and $\lambda(\cdot)$
    $\boldsymbol{\theta}^{-} \leftarrow \boldsymbol{\theta}$ and $k \leftarrow 0$
    repeat
        Sample $\mathbf{x} \sim \mathcal{D}$, and $n \sim \mathcal{U}[\![1, N(k) - 1]\!]$
        Sample $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$
        $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\theta}^{-}) \leftarrow$
            $\lambda(t_n) d(\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x} + t_{n+1}\mathbf{z}, t_{n+1}), \boldsymbol{f}_{\boldsymbol{\theta}^{-}}(\mathbf{x} + t_n \mathbf{z}, t_n)$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\theta}^{-})$
        $\boldsymbol{\theta}^{-} \leftarrow \text{stopgrad}(\mu(k)\boldsymbol{\theta}^{-} + (1 - \mu(k))\boldsymbol{\theta})$
        $k \leftarrow k + 1$
    until convergence
```

## 6. Experimental Results

### 6.1 Training Consistency Models

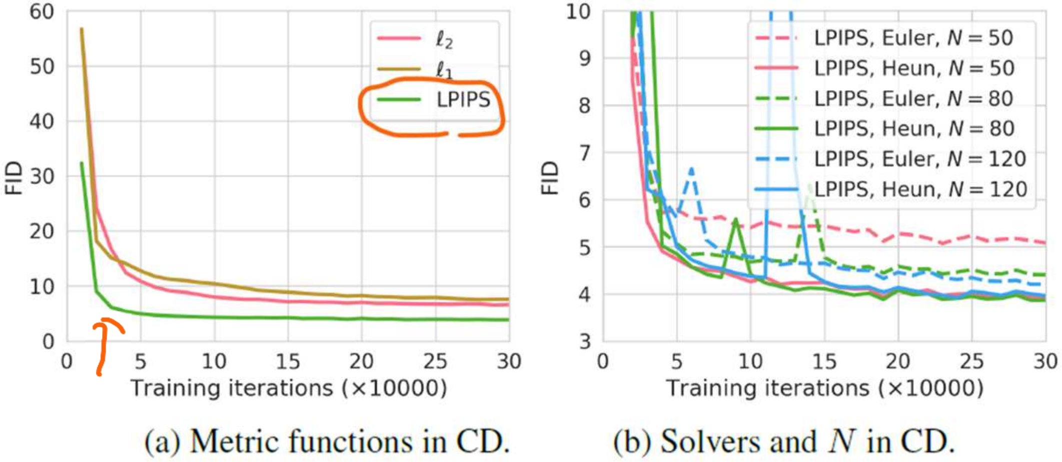
- The effect of various hyperparameters on the performance of consistency models trained by consistency distillation (CD) and consistency training (CT).

1) Focus on effect of the metric function $d(\cdot,\cdot)$, the ODE solver, and the number of discretization steps *N* in CD,

2) Then, Investigate effect of the schedule functions $N(\cdot)$ and $\mu(\cdot)$ in CT.

- Fig.3 (a) : LPIPS loss가 가장 효과적인 distance metric임을 알 수 있음
- Fig.3 (b) : LPIPS를 고정 metric으로 활용하여 단계단계 실험을 진행한다. (b)에서는 solver에 대한 학습 효과를 보는데, 1차 근사만 고려하는 Euler보다는 2차 근사를 고려하는 Heun이 좀 더 좋은 성능을 보임.
- Fig.3 (c) : 앞서 bias를 줄이기 위해 테스트한 time step sample 수 *N*에 대한 실험, 당연하게도 *N*이 커질수록 성능이 좋아진다. *N*이 어느 정도 증가하면 그 이후로는 성능 수렴이 발생하는 것도 함께 확인할 수 있다. (아무래도 numerical ODE에 따른 성능 향상의 bottleneck이지 않을까 생각해봄.)

Metric Function : $l_2$ distance, $l_1$ distance and the Learned Perceptual Image Patch Similarity (**LPIPS**, Zhang et al. (2018))



(a) Metric functions in CD. (b) Solvers and $N$ in CD.

Fig. 3. Various factors that affect consistency distillation (CD) and consistency training (CT) on CIFAR-10. The best configuration for CD is LPIPS, Heun ODE solver, and *N*=18. Our adaptive schedule functions for *N* and $\mu$ make CT converge significantly faster than fixing them to be constants during the course of optimization.

## 6. Experimental Results

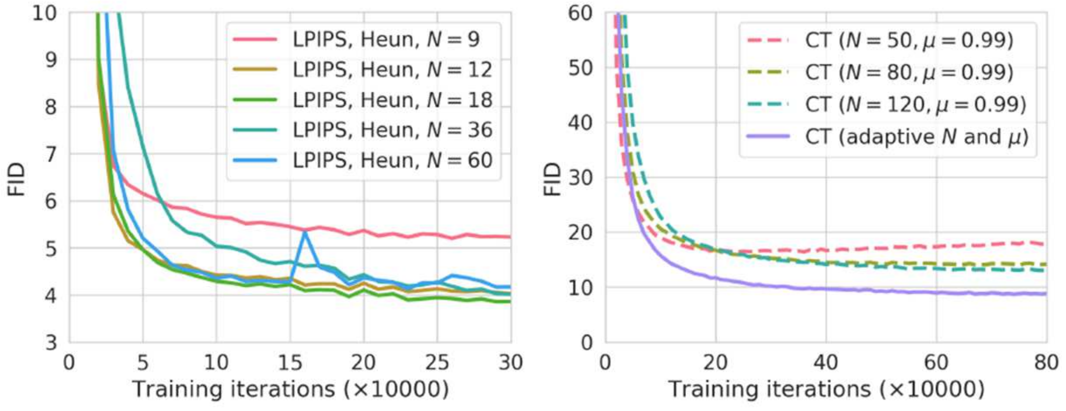
### 6.1 Training Consistency Models

- Fig.3 (c) : 앞서 bias를 줄이기 위해 테스트한 time step sample 수 *N*에 대한 실험, 당연하게도 *N*이 커질수록 성능이 좋아진다. *N*이 어느 정도 증가하면 그 이후로는 성능 수렴이 발생하는 것도 함께 확인할 수 있다. (아무래도 numerical ODE에 따른 성능 향상의 bottleneck이지 않을까 생각해봄.)

- Fig.4(d) : CT를 사용한 학습 과정 (일단 FID가 현저히 떨어지는건 어쩔 수 없는 한계점)

  ✓ CT의 경우에는 CD와는 다르게 특정 numerical ODE solver에 성능이 좌우되지 않기 때문에(학습에 사용되는 샘플링은 사전에 정의된 커널로 함) solver를 사용할 필요가 없다. CT의 경우에는 distillation이 사용되지 않기 때문에 *N*에 대한 효과가 두드러짐. → 예컨데 N이 너무 작으면 빠른 수렴은 가능했지만 샘플링 성능이 좋지 못하고, N을 키우면 수렴은 좀 느려지지만 샘플링 성능은 향상된다. 이 두 가지 장점을 같이 사용하기 위해 ***N*을 조금씩 증가시키면서 학습시키는 방법(보라색)**을 고안하였고, **EMA factor μ** 또한 이에 맞춰 점차 증가시키는 방법을 사용하였다.

  ✓ 그래프를 보면 빠른 성능 수렴 + 높은 샘플링 퀄리티(FID)를 보이는 것을 확인할 수 있다.

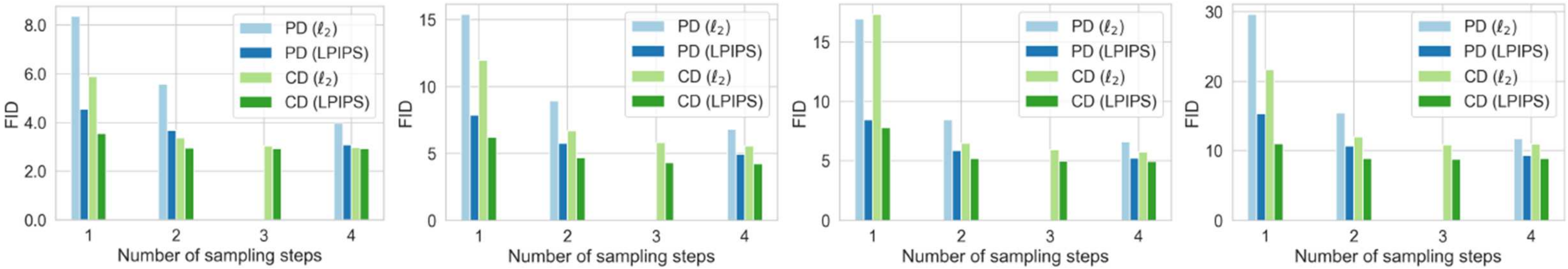


(c) $N$ with Heun solver in CD. (d) Adaptive $N$ and $\mu$ in CT.

Fig. 3. Various factors that affect consistency distillation (CD) and consistency training (CT) on CIFAR-10. The best configuration for CD is LPIPS, Heun ODE solver, and *N*=18. Our adaptive schedule functions for *N* and $\mu$ make CT converge significantly faster than fixing them to be constants during the course of optimization.

## 6. Experimental Results

### 6.1 Training Consistency Models



(a) CIFA-10 (b) ImageNet 64x64 (c) Bedroom 256x256 (d) Cat 256x256

Fig. 4. **Multistep image generation** with consistency distillation (CD). CD outperforms progressive distillation (PD) across all datasets and sampling steps. The only exception is single-step generation on Bedroom 256x256.

## 6. Experimental Results

### 6.2. Few-Step Image Generation

#### Distillation

- All methods **distill** from an EDM(**Karras** et al., 2022) model that we pre-trained in-house.
- Note that across all sampling iterations, using the LPIPS metric uniformly improves PD(progressive distillation) compared to the squared ℓ2 distance in the original paper of Salimans & Ho (2022).
- Both **PD(progressive distillation)** and **CD(consistency distillation) improve** as we take **more sampling steps**.
- We find that CD uniformly outperforms PD across all datasets, sampling steps, and metric functions considered, except for single-step generation on Bedroom 256x256, where CD with ℓ2 slightly underperforms PD with ℓ2.
- Table 1, CD even outperforms distillation approaches that require synthetic dataset construction, such as Knowledge Distillation (Luhman & Luhman, 2021) and DFNO (Zheng et al., 2022).

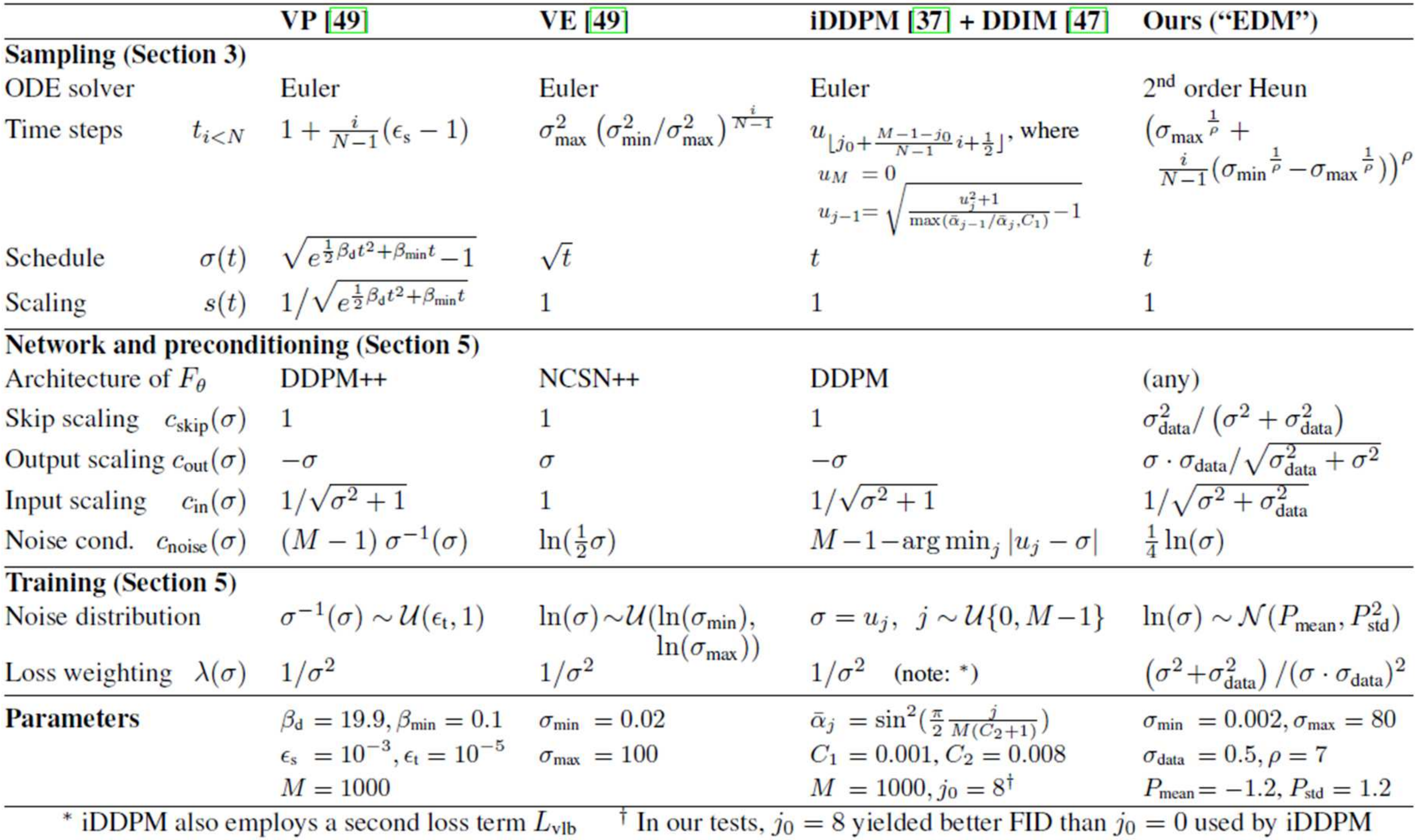
❖ EDM : Tero Karras, Miika Aittala, Timo Aila, Samuli Laine, "Elucidating the Design Space of Diffusion-Based Generative Models," NeurIPS 2022. https://github.com/NVlabs/edm

#### Direct Generation

- Compare the sample quality of consistency training (CT) with other generative models using one-step and two-step generation.
- CT outperforms all single-step, non-adversarial generative models.
- CT obtains comparable quality to PD for single-step generation without relying on distillation

## Consistency Models

### EDM (Elucidating the Design Space of Diffusion-Based Generative Models)



| | | VP [49] | VE [49] | iDDPM [37] + DDIM [47] | Ours ("EDM") |
|---|---|---|---|---|---|
| **Sampling (Section 3)** | | | | | |
| ODE solver | | Euler | Euler | Euler | $2^{\text{nd}}$ order Heun |
| Time steps | $t_{i<N}$ | $1+\frac{i}{N-1}(\epsilon_s-1)$ | $\sigma_{\max}^2\left(\sigma_{\min}^2/\sigma_{\max}^2\right)^{\frac{i}{N-1}}$ | $u_{\lfloor j_0+\frac{M-1-j_0}{N-1}i+\frac{1}{2}\rfloor}$, where<br>$u_M=0$<br>$u_{j-1}=\sqrt{\frac{u_j^2+1}{\max(\bar{\alpha}_{j-1}/\bar{\alpha}_j,C_1)}-1}$ | $\left({\sigma_{\max}}^{\frac{1}{\rho}}+\frac{i}{N-1}({\sigma_{\min}}^{\frac{1}{\rho}}-{\sigma_{\max}}^{\frac{1}{\rho}})\right)^{\rho}$ |
| Schedule | $\sigma(t)$ | $\sqrt{e^{\frac{1}{2}\beta_d t^2+\beta_{\min}t}-1}$ | $\sqrt{t}$ | $t$ | $t$ |
| Scaling | $s(t)$ | $1/\sqrt{e^{\frac{1}{2}\beta_d t^2+\beta_{\min}t}}$ | 1 | 1 | 1 |
| **Network and preconditioning (Section 5)** | | | | | |
| Architecture of $F_\theta$ | | DDPM++ | NCSN++ | DDPM | (any) |
| Skip scaling | $c_{\text{skip}}(\sigma)$ | 1 | 1 | 1 | $\sigma_{\text{data}}^2/\left(\sigma^2+\sigma_{\text{data}}^2\right)$ |
| Output scaling | $c_{\text{out}}(\sigma)$ | $-\sigma$ | $\sigma$ | $-\sigma$ | $\sigma\cdot\sigma_{\text{data}}/\sqrt{\sigma_{\text{data}}^2+\sigma^2}$ |
| Input scaling | $c_{\text{in}}(\sigma)$ | $1/\sqrt{\sigma^2+1}$ | 1 | $1/\sqrt{\sigma^2+1}$ | $1/\sqrt{\sigma^2+\sigma_{\text{data}}^2}$ |
| Noise cond. | $c_{\text{noise}}(\sigma)$ | $(M-1)\,\sigma^{-1}(\sigma)$ | $\ln(\frac{1}{2}\sigma)$ | $M-1-\arg\min_j\lvert u_j-\sigma\rvert$ | $\frac{1}{4}\ln(\sigma)$ |
| **Training (Section 5)** | | | | | |
| Noise distribution | | $\sigma^{-1}(\sigma)\sim\mathcal{U}(\epsilon_t,1)$ | $\ln(\sigma)\sim\mathcal{U}(\ln(\sigma_{\min}),\ln(\sigma_{\max}))$ | $\sigma=u_j,\ \ j\sim\mathcal{U}\{0,M-1\}$ | $\ln(\sigma)\sim\mathcal{N}(P_{\text{mean}},P_{\text{std}}^2)$ |
| Loss weighting | $\lambda(\sigma)$ | $1/\sigma^2$ | $1/\sigma^2$ | $1/\sigma^2$ (note: *) | $\left(\sigma^2+\sigma_{\text{data}}^2\right)/(\sigma\cdot\sigma_{\text{data}})^2$ |
| **Parameters** | | $\beta_d=19.9,\beta_{\min}=0.1$<br>$\epsilon_s=10^{-3},\epsilon_t=10^{-5}$<br>$M=1000$ | $\sigma_{\min}=0.02$<br>$\sigma_{\max}=100$ | $\bar{\alpha}_j=\sin^2(\frac{\pi}{2}\frac{j}{M(C_2+1)})$<br>$C_1=0.001,C_2=0.008$<br>$M=1000,j_0=8^{\dagger}$ | $\sigma_{\min}=0.002,\sigma_{\max}=80$<br>$\sigma_{\text{data}}=0.5,\rho=7$<br>$P_{\text{mean}}=-1.2,P_{\text{std}}=1.2$ |

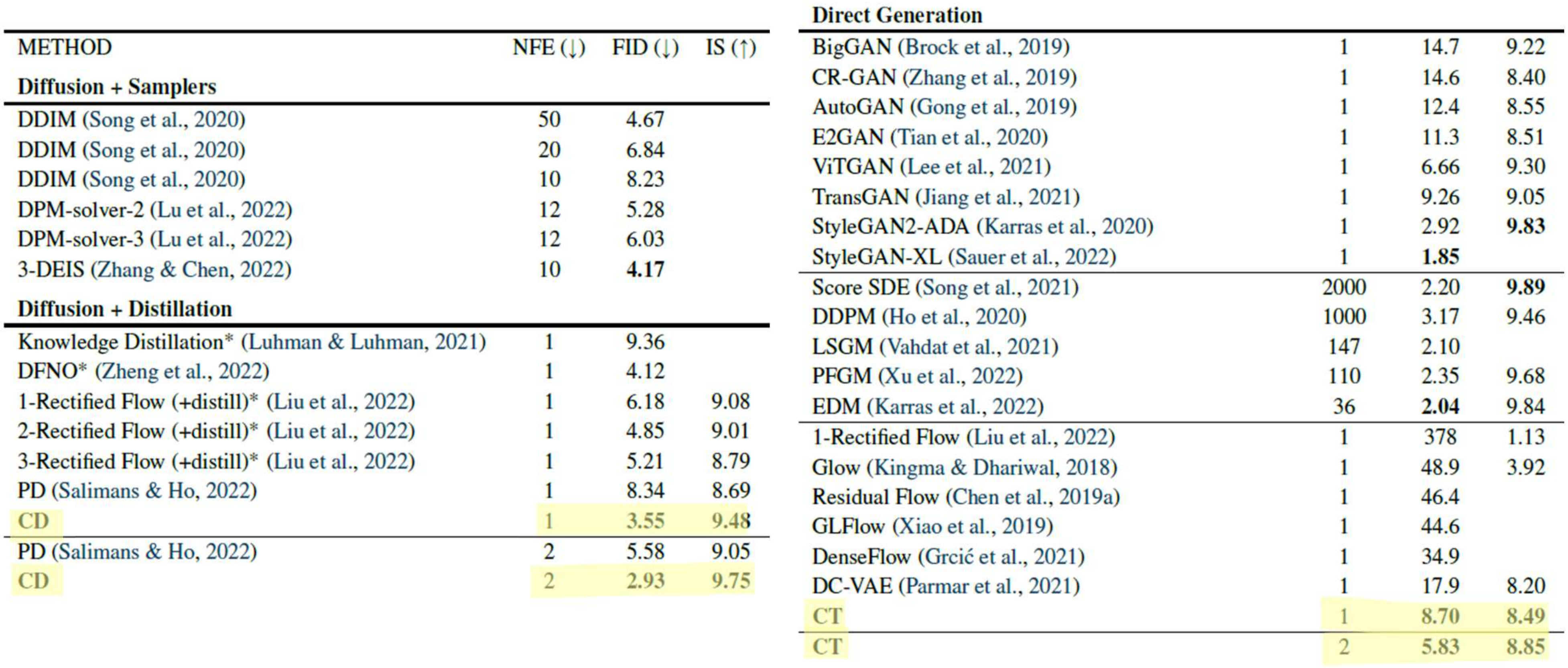
* iDDPM also employs a second loss term $L_{\text{vlb}}$ † In our tests, $j_0=8$ yielded better FID than $j_0=0$ used by iDDPM

## Consistency Models

## 6. Experimental Results

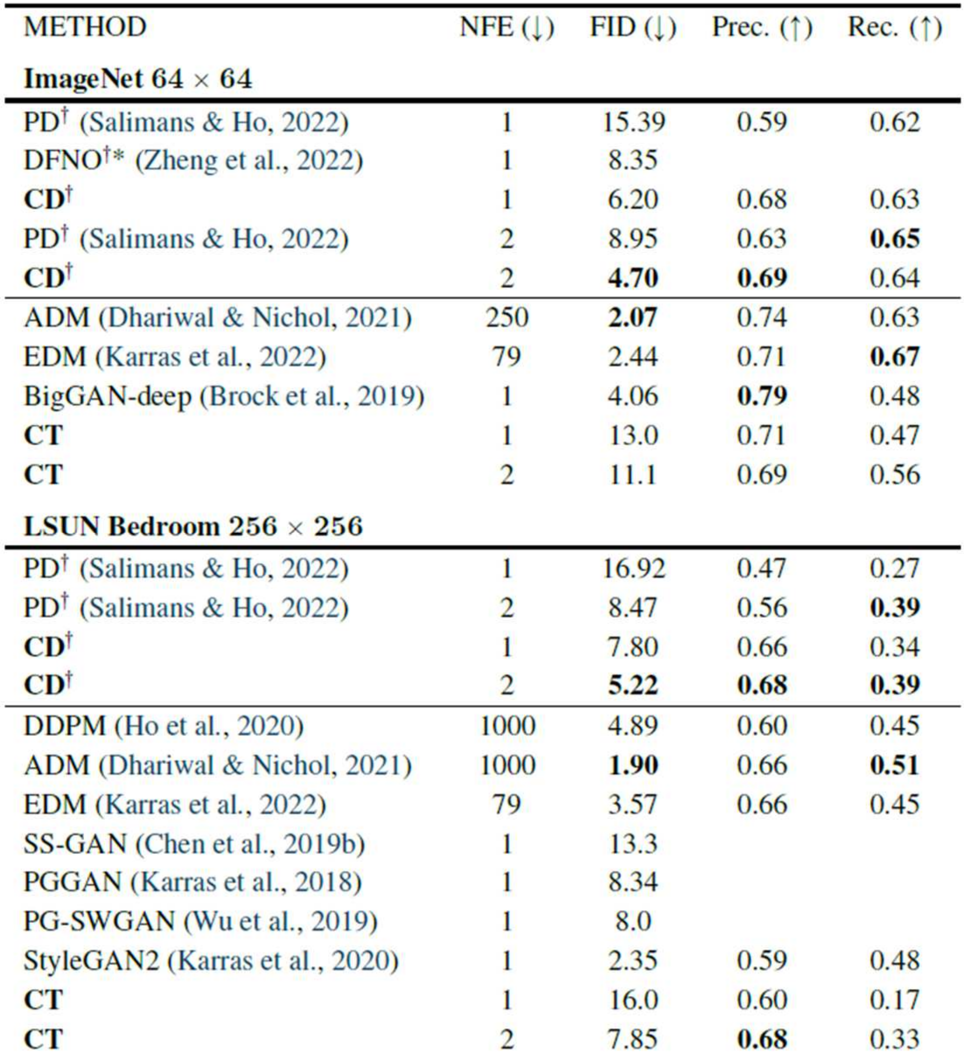
### 6.2. Few-Step Image Generation

Table 1: Sample quality on CIFAR-10. *Methods that require synthetic data construction for distillation

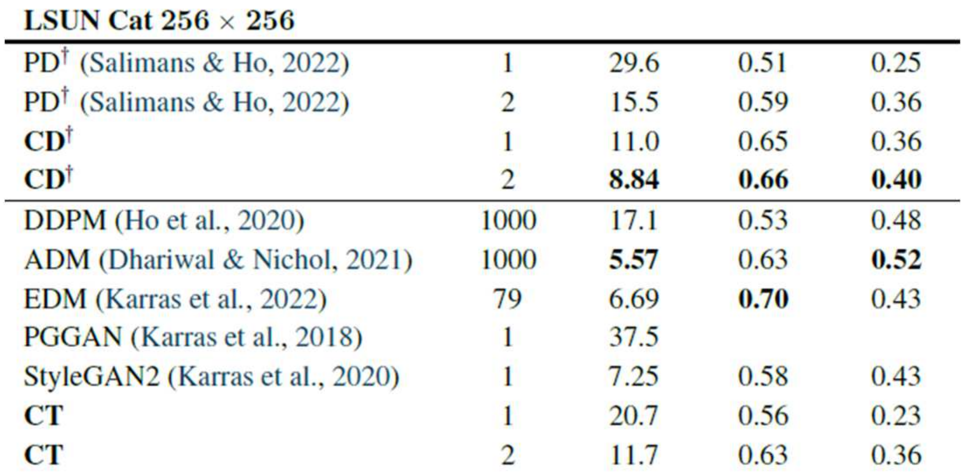


| METHOD | NFE (↓) | FID (↓) | IS (↑) |
|---|---|---|---|
| **Diffusion + Samplers** | | | |
| DDIM (Song et al., 2020) | 50 | 4.67 | |
| DDIM (Song et al., 2020) | 20 | 6.84 | |
| DDIM (Song et al., 2020) | 10 | 8.23 | |
| DPM-solver-2 (Lu et al., 2022) | 12 | 5.28 | |
| DPM-solver-3 (Lu et al., 2022) | 12 | 6.03 | |
| 3-DEIS (Zhang & Chen, 2022) | 10 | **4.17** | |
| **Diffusion + Distillation** | | | |
| Knowledge Distillation* (Luhman & Luhman, 2021) | 1 | 9.36 | |
| DFNO* (Zheng et al., 2022) | 1 | 4.12 | |
| 1-Rectified Flow (+distill)* (Liu et al., 2022) | 1 | 6.18 | 9.08 |
| 2-Rectified Flow (+distill)* (Liu et al., 2022) | 1 | 4.85 | 9.01 |
| 3-Rectified Flow (+distill)* (Liu et al., 2022) | 1 | 5.21 | 8.79 |
| PD (Salimans & Ho, 2022) | 1 | 8.34 | 8.69 |
| CD | 1 | 3.55 | 9.48 |
| PD (Salimans & Ho, 2022) | 2 | 5.58 | 9.05 |
| CD | 2 | 2.93 | 9.75 |
| **Direct Generation** | | | |
| BigGAN (Brock et al., 2019) | 1 | 14.7 | 9.22 |
| CR-GAN (Zhang et al., 2019) | 1 | 14.6 | 8.40 |
| AutoGAN (Gong et al., 2019) | 1 | 12.4 | 8.55 |
| E2GAN (Tian et al., 2020) | 1 | 11.3 | 8.51 |
| ViTGAN (Lee et al., 2021) | 1 | 6.66 | 9.30 |
| TransGAN (Jiang et al., 2021) | 1 | 9.26 | 9.05 |
| StyleGAN2-ADA (Karras et al., 2020) | 1 | 2.92 | **9.83** |
| StyleGAN-XL (Sauer et al., 2022) | 1 | **1.85** | |
| Score SDE (Song et al., 2021) | 2000 | 2.20 | **9.89** |
| DDPM (Ho et al., 2020) | 1000 | 3.17 | 9.46 |
| LSGM (Vahdat et al., 2021) | 147 | 2.10 | |
| PFGM (Xu et al., 2022) | 110 | 2.35 | 9.68 |
| EDM (Karras et al., 2022) | 36 | **2.04** | 9.84 |
| 1-Rectified Flow (Liu et al., 2022) | 1 | 378 | 1.13 |
| Glow (Kingma & Dhariwal, 2018) | 1 | 48.9 | 3.92 |
| Residual Flow (Chen et al., 2019a) | 1 | 46.4 | |
| GLFlow (Xiao et al., 2019) | 1 | 44.6 | |
| DenseFlow (Grcić et al., 2021) | 1 | 34.9 | |
| DC-VAE (Parmar et al., 2021) | 1 | 17.9 | 8.20 |
| CT | 1 | 8.70 | 8.49 |
| CT | 2 | 5.83 | 8.85 |

# Consistency Models



| METHOD | NFE (↓) | FID (↓) | Prec. (↑) | Rec. (↑) |
|---|---|---|---|---|
| **ImageNet 64 × 64** | | | | |
| PD† (Salimans & Ho, 2022) | 1 | 15.39 | 0.59 | 0.62 |
| DFNO†* (Zheng et al., 2022) | 1 | 8.35 | | |
| **CD†** | 1 | 6.20 | 0.68 | 0.63 |
| PD† (Salimans & Ho, 2022) | 2 | 8.95 | 0.63 | **0.65** |
| **CD†** | 2 | **4.70** | **0.69** | 0.64 |
| ADM (Dhariwal & Nichol, 2021) | 250 | **2.07** | 0.74 | 0.63 |
| EDM (Karras et al., 2022) | 79 | 2.44 | 0.71 | **0.67** |
| BigGAN-deep (Brock et al., 2019) | 1 | 4.06 | **0.79** | 0.48 |
| **CT** | 1 | 13.0 | 0.71 | 0.47 |
| **CT** | 2 | 11.1 | 0.69 | 0.56 |
| **LSUN Bedroom 256 × 256** | | | | |
| PD† (Salimans & Ho, 2022) | 1 | 16.92 | 0.47 | 0.27 |
| PD† (Salimans & Ho, 2022) | 2 | 8.47 | 0.56 | **0.39** |
| **CD†** | 1 | 7.80 | 0.66 | 0.34 |
| **CD†** | 2 | **5.22** | **0.68** | **0.39** |
| DDPM (Ho et al., 2020) | 1000 | 4.89 | 0.60 | 0.45 |
| ADM (Dhariwal & Nichol, 2021) | 1000 | **1.90** | 0.66 | **0.51** |
| EDM (Karras et al., 2022) | 79 | 3.57 | 0.66 | 0.45 |
| SS-GAN (Chen et al., 2019b) | 1 | 13.3 | | |
| PGGAN (Karras et al., 2018) | 1 | 8.34 | | |
| PG-SWGAN (Wu et al., 2019) | 1 | 8.0 | | |
| StyleGAN2 (Karras et al., 2020) | 1 | 2.35 | 0.59 | 0.48 |
| **CT** | 1 | 16.0 | 0.60 | 0.17 |
| **CT** | 2 | 7.85 | **0.68** | 0.33 |



| **LSUN Cat 256 × 256** | | | | |
|---|---|---|---|---|
| PD† (Salimans & Ho, 2022) | 1 | 29.6 | 0.51 | 0.25 |
| PD† (Salimans & Ho, 2022) | 2 | 15.5 | 0.59 | 0.36 |
| **CD†** | 1 | 11.0 | 0.65 | 0.36 |
| **CD†** | 2 | **8.84** | **0.66** | **0.40** |
| DDPM (Ho et al., 2020) | 1000 | 17.1 | 0.53 | 0.48 |
| ADM (Dhariwal & Nichol, 2021) | 1000 | **5.57** | 0.63 | **0.52** |
| EDM (Karras et al., 2022) | 79 | 6.69 | **0.70** | 0.43 |
| PGGAN (Karras et al., 2018) | 1 | 37.5 | | |
| StyleGAN2 (Karras et al., 2020) | 1 | 7.25 | 0.58 | 0.43 |
| **CT** | 1 | 20.7 | 0.56 | 0.23 |
| **CT** | 2 | 11.7 | 0.63 | 0.36 |

Table 2: Sample quality on ImageNet 64x64, and LSUN Bedroom & Cat 256x256. †Distillation techniques

# Consistency Models

## Direct Generation

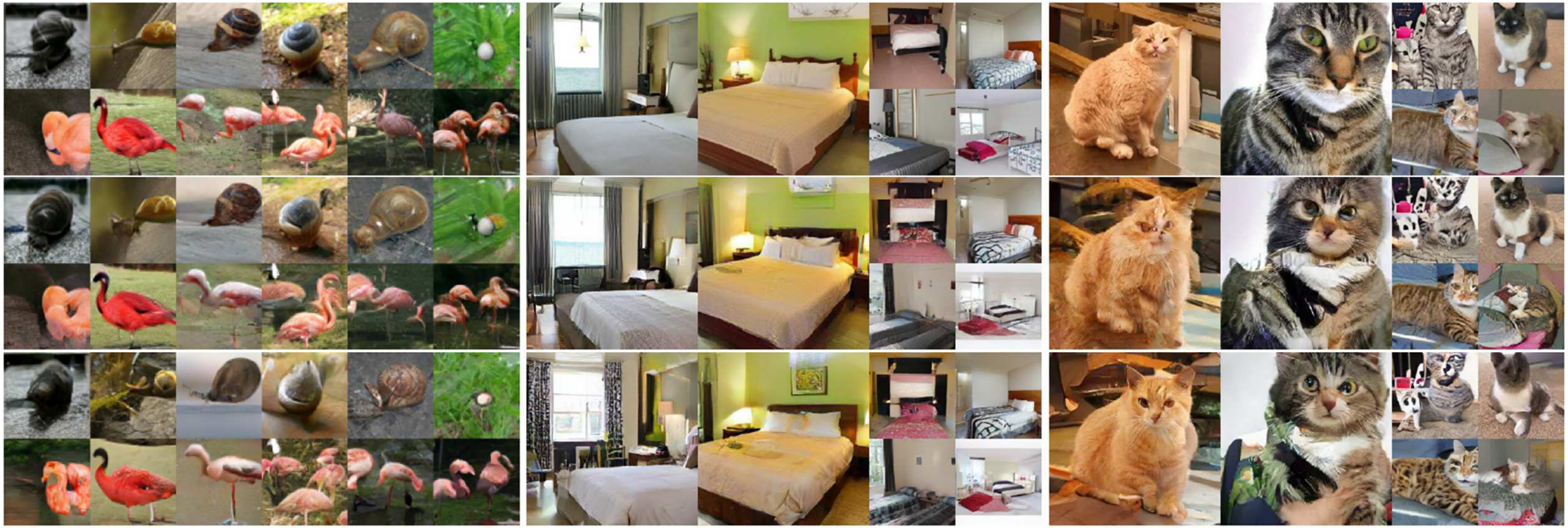
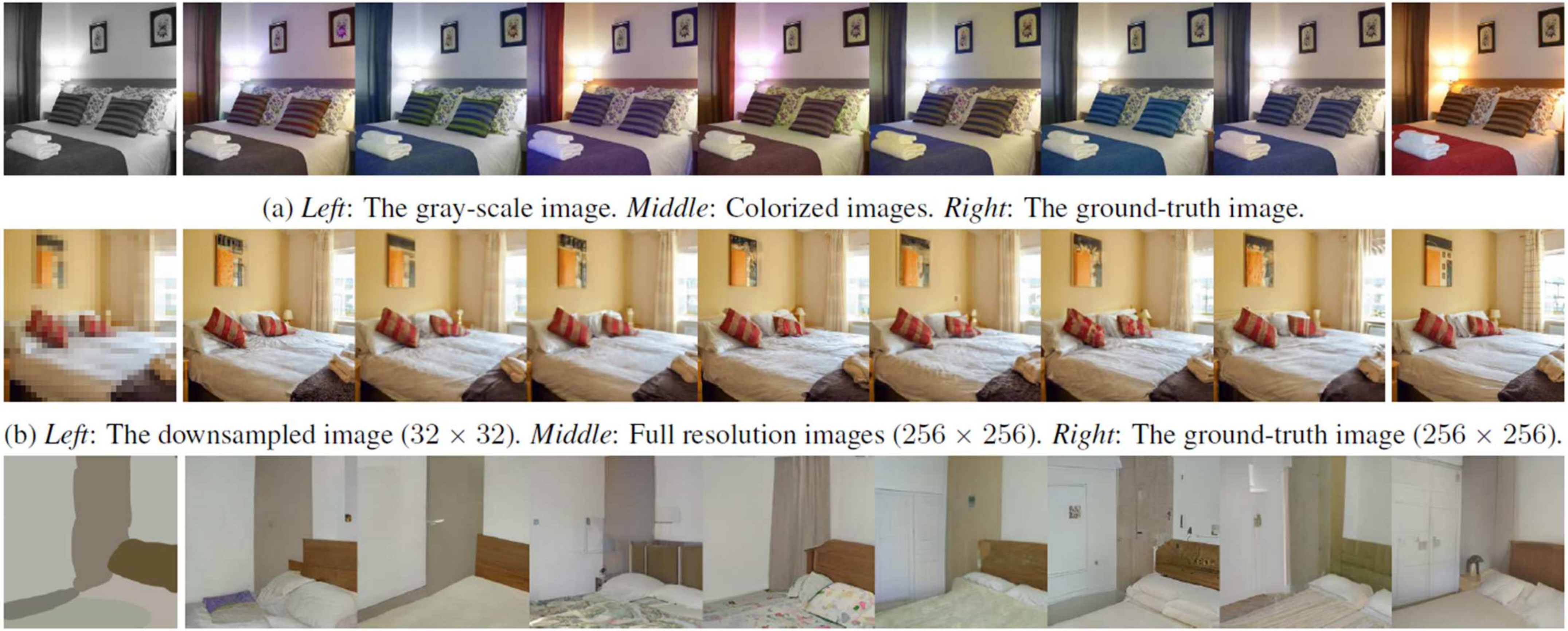


Figure 5: Samples generated by EDM (*top*), CT + single-step generation (*middle*), and CT + 2-step generation (*Bottom*). All corresponding images are generated from the same initial noise.

# Consistency Models

## 6.3 Zero-Shot Image Editing



(a) *Left*: The gray-scale image. *Middle*: Colorized images. *Right*: The ground-truth image.

(b) *Left*: The downsampled image (32 × 32). *Middle*: Full resolution images (256 × 256). *Right*: The ground-truth image (256 × 256).

(c) *Left*: A stroke input provided by users. *Right*: Stroke-guided image generation.

Figure 6: Zero-shot image editing with a consistency model trained by consistency distillation on LSUN Bedroom 256 × 256.

## Appendix C. Additional Experimental Details

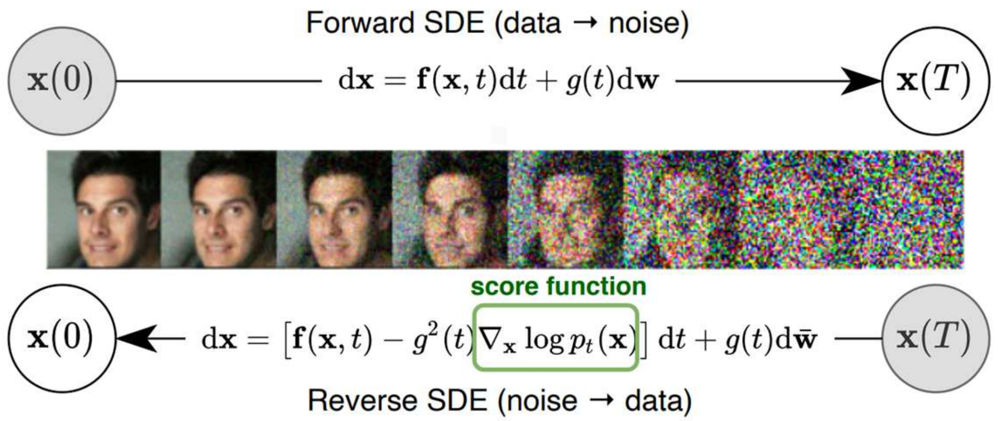
### Model Architecture

- Follow Song et al. (2021), Dhariwal & Nichol (2021) for model architectures.
- Use the **NCSN++ architecture** in **Song et al**. (2021) for all CIFAR-10 experiments, and take the corresponding network architectures from **Dhariwal & Nichol** (2021) when performing experiments on ImageNet 64x64, LSUN Bedroom 256x256 and LSUN Cat 256x256.

**Song, Y**., et al., **Score-based generative modeling through stochastic differential equations**. ICLR 2021.

**Dhariwal, P**. and Nichol, A. **Diffusion models beat gans on image synthesis**. NeurIPS 2021.

Score-based Generative Modeling through Stochastic Differential Equations



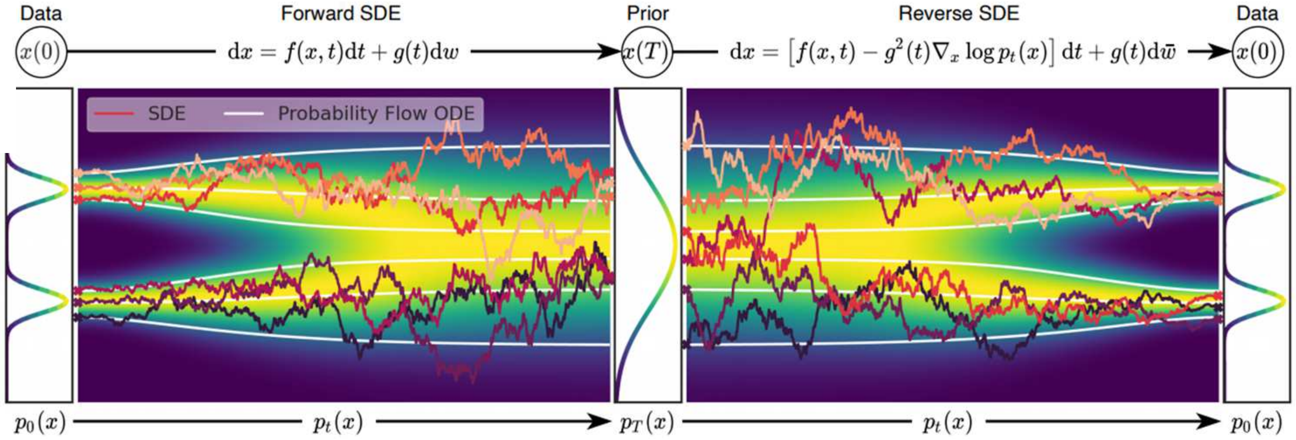
Forward SDE (data → noise)

$$\mathbf{x}(0) \longrightarrow \mathrm{d}\mathbf{x} = \mathbf{f}(\mathbf{x}, t)\mathrm{d}t + g(t)\mathrm{d}\mathbf{w} \longrightarrow \mathbf{x}(T)$$

score function

$$\mathbf{x}(0) \longleftarrow \mathrm{d}\mathbf{x} = \left[\mathbf{f}(\mathbf{x}, t) - g^2(t)\boxed{\nabla_\mathbf{x} \log p_t(\mathbf{x})}\right] \mathrm{d}t + g(t)\mathrm{d}\bar{\mathbf{w}} \longleftarrow \mathbf{x}(T)$$

Reverse SDE (noise → data)

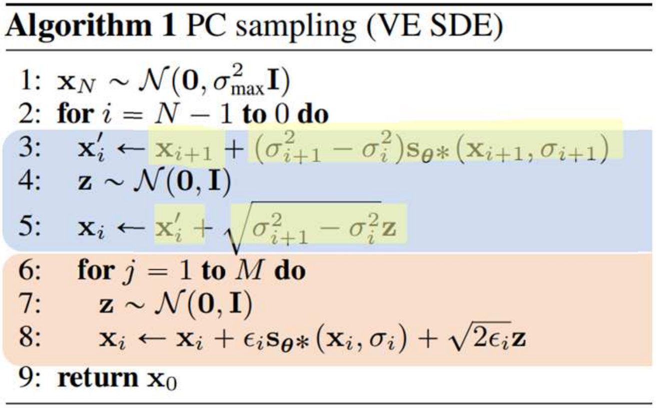


https://blog.si-analytics.ai/49

# Consistency Models; Appendices

## Appendix C. Additional Experimental Details

**Song, Y.**, et al., **Score-based generative modeling through stochastic differential equations**. ICLR 2021.



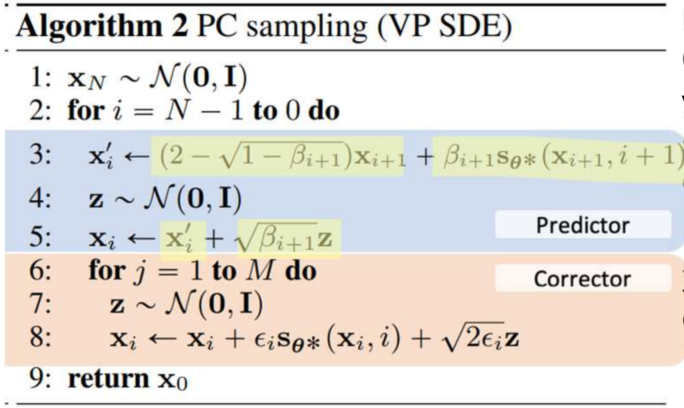
**Algorithm 1** PC sampling (VE SDE)

```
$\mathbf{x}_N \sim \mathcal{N}(\mathbf{0}, \sigma_{\max}^2 \mathbf{I})$
for $i = N - 1$ to $0$ do
    $\mathbf{x}'_i \leftarrow \mathbf{x}_{i+1} + (\sigma_{i+1}^2 - \sigma_i^2)\mathbf{s}_{\theta^*}(\mathbf{x}_{i+1}, \sigma_{i+1})$
    $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
    $\mathbf{x}_i \leftarrow \mathbf{x}'_i + \sqrt{\sigma_{i+1}^2 - \sigma_i^2}\mathbf{z}$
    for $j = 1$ to $M$ do
        $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
        $\mathbf{x}_i \leftarrow \mathbf{x}_i + \epsilon_i \mathbf{s}_{\theta^*}(\mathbf{x}_i, \sigma_i) + \sqrt{2\epsilon_i}\mathbf{z}$
return $\mathbf{x}_0$
```

**Predictor-Corrector (PC) sampling**

Variance Exploding stochastic differential equation (VE SDE)

※ NCSN++ cont. (VE) model



**Algorithm 2** PC sampling (VP SDE)

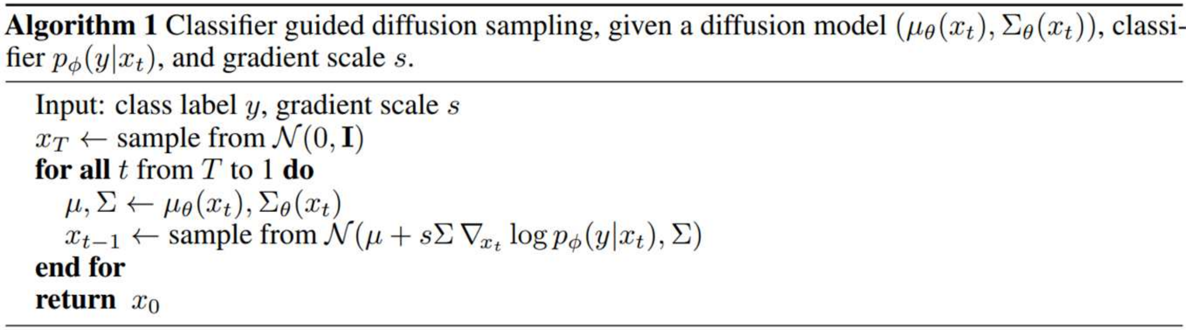
```
$\mathbf{x}_N \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
for $i = N - 1$ to $0$ do
    $\mathbf{x}'_i \leftarrow (2 - \sqrt{1 - \beta_{i+1}})\mathbf{x}_{i+1} + \beta_{i+1}\mathbf{s}_{\theta^*}(\mathbf{x}_{i+1}, i + 1)$
    $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
    $\mathbf{x}_i \leftarrow \mathbf{x}'_i + \sqrt{\beta_{i+1}}\mathbf{z}$
    for $j = 1$ to $M$ do
        $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
        $\mathbf{x}_i \leftarrow \mathbf{x}_i + \epsilon_i \mathbf{s}_{\theta^*}(\mathbf{x}_i, i) + \sqrt{2\epsilon_i}\mathbf{z}$
return $\mathbf{x}_0$
```

**Predictor-Corrector (PC) sampling**

Variance Preserving stochastic differential equation (VP SDE)

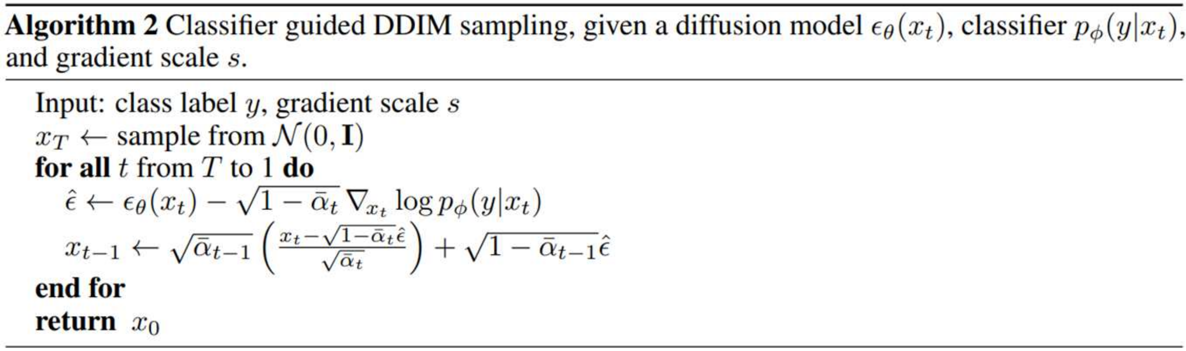
※ DDPM++ cont. (VP) model

**Dhariwal, P.** and Nichol, A. **Diffusion models beat gans on image synthesis**. NeurIPS 2021.



**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 $x_0$
```

# Consistency Models; Appendices

## Appendix C. Additional Experimental Details

### Parameterization for Consistency Models

- Use the same architectures for consistency models as those used for **EDMs**. Slightly **modify the skip connections in EDM** to ensure the boundary condition holds for consistency models.
- Parameterize a consistency model

$$\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}, t) = c_{\text{skip}}(t)\mathbf{x} + c_{\text{out}}(t) F_{\boldsymbol{\theta}}(\mathbf{x}, t).$$

✓ EDM

$$c_{\text{skip}}(t) = \frac{\sigma_{\text{data}}^2}{t^2 + \sigma_{\text{data}}^2}, \quad c_{\text{out}}(t) = \frac{\sigma_{\text{data}} t}{\sqrt{\sigma_{\text{data}}^2 + t^2}}, \quad \text{where } \sigma_{\text{data}} = 0.5.$$

it does not satisfy the boundary condition when the smallest time instant $\epsilon \neq 0$.

✓ Modify them

$$c_{\text{skip}}(t) = \frac{\sigma_{\text{data}}^2}{(t-\epsilon)^2 + \sigma_{\text{data}}^2}, \quad c_{\text{out}}(t) = \frac{\sigma_{\text{data}}(t-\epsilon)}{\sqrt{\sigma_{\text{data}}^2 + t^2}},$$

which clearly satisfies $c_{\text{skip}}(\epsilon) = 1$ and $c_{\text{out}}(\epsilon) = 0$.

### Schedule Functions for Consistency Training

- Consistency generation requires specifying schedule functions *N*() and $\mu$() for best performance.

$$N(k) = \left\lceil \sqrt{\frac{k}{K}((s_1+1)^2 - s_0^2) + s_0^2} - 1 \right\rceil + 1$$

$$\mu(k) = \exp\left(\frac{s_0 \log \mu_0}{N(k)}\right),$$

- *K* : the total number of training iterations
- $s_0$ : the initial discretization steps
- $s_1 > s_0$ : the target discretization steps at the end of training
- $\mu_0$ : the EMA decay rate at the beginning of model training.

# Consistency Models; Appendices

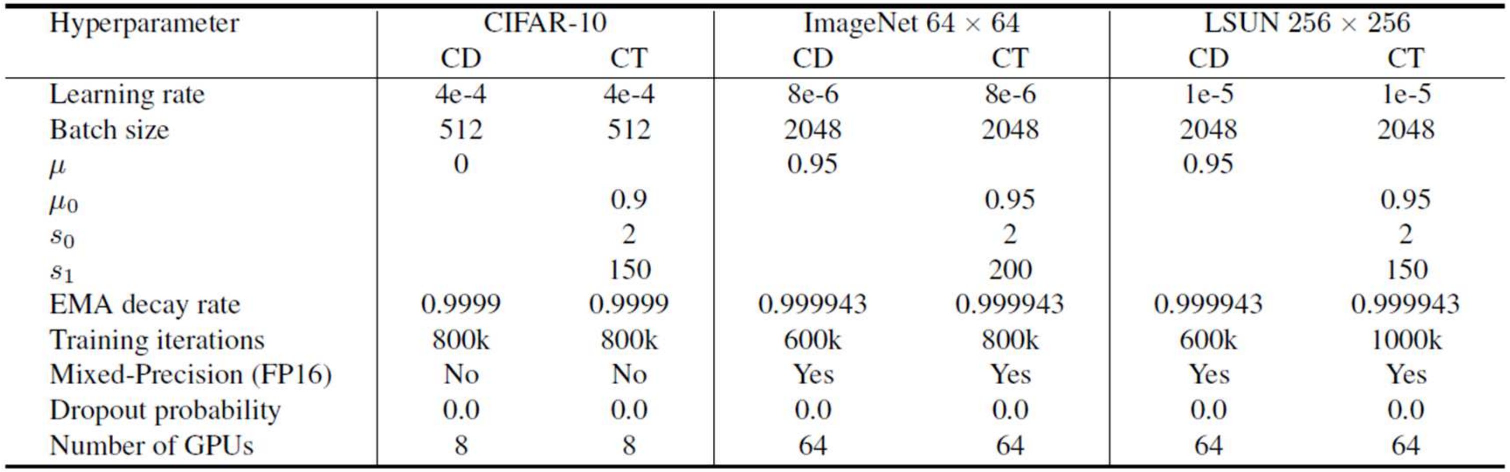
## Appendix C. Additional Experimental Details

### Training Details

- In both **consistency distillation** and **progressive distillation**,
  - **Distill EDMs** (Karras et al., 2022).
  - Trained these EDMs for 600k and 300k iterations and reduced the batch size from 4096 to 2048 according to the specifications given in Karras et al. (2022).
  - In all **distillation** experiments, Initialized the consistency model with pre-trained EDM weights.

- For **consistency training**,
  - Initialized the model randomly, just as did for training the EDMs.
  - Trained all consistency models with the **Rectified Adam** optimizer (Liu et al., 2019), with **no learning rate decay** or warm-up, and **no weight decay**.
  - Applied **EMA** to the **weights of the online consistency models** in both consistency distillation and consistency training, as well as to **the weights of the training online consistency models** according to Karras et al. (2022).

Table 3: Hyperparameters used for training CD and CT models



| Hyperparameter | CIFAR-10 | | ImageNet 64 × 64 | | LSUN 256 × 256 | |
|---|---|---|---|---|---|---|
| | CD | CT | CD | CT | CD | CT |
| Learning rate | 4e-4 | 4e-4 | 8e-6 | 8e-6 | 1e-5 | 1e-5 |
| Batch size | 512 | 512 | 2048 | 2048 | 2048 | 2048 |
| $\mu$ | 0 | | 0.95 | | 0.95 | |
| $\mu_0$ | | 0.9 | | 0.95 | | 0.95 |
| $s_0$ | | 2 | | 2 | | 2 |
| $s_1$ | | 150 | | 200 | | 150 |
| EMA decay rate | 0.9999 | 0.9999 | 0.999943 | 0.999943 | 0.999943 | 0.999943 |
| Training iterations | 800k | 800k | 600k | 800k | 600k | 1000k |
| Mixed-Precision (FP16) | No | No | Yes | Yes | Yes | Yes |
| Dropout probability | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| Number of GPUs | 8 | 8 | 64 | 64 | 64 | 64 |

## Consistency Models; Official Code

https://github.com/openai/consistency_models

- Provide examples of **EDM training**, **consistency distillation**, **consistency training**, **single-step generation**, and **multistep generation** in scripts/launch.sh

EDM : Karras, T., Aittala, M., Aila, T., and Laine, S. Elucidating the design space of diffusion-based generative models. In Proc. NeurIPS, 2022

# Consistency Models; Official Code

## scripts/launch.sh

```
################################################################
# Training EDM models on class-conditional ImageNet-64, and LSUN 256
################################################################

mpiexec -n 8 python edm_train.py --attention_resolutions 32,16,8 --class_cond
True --use_scale_shift_norm True --dropout 0.1 --ema_rate
0.999,0.9999,0.9999432189950708 --global_batch_size 4096 --image_size 64 --lr
0.0001 --num_channels 192 --num_head_channels 64 --num_res_blocks 3
--resblock_updown True --schedule_sampler lognormal --use_fp16 True
--weight_decay 0.0 --weight_schedule karras --data_dir /path/to/imagenet

python -m orc.diffusion.scripts.train_imagenet_edm --attention_resolutions
32,16,8 --class_cond False --dropout 0.1 --ema_rate
0.999,0.9999,0.9999432189950708 --global_batch_size 256 --image_size 256 --lr
0.0001 --num_channels 256 --num_head_channels 64 --num_res_blocks 2
--resblock_updown True --schedule_sampler lognormal --use_fp16 True
--use_scale_shift_norm False --weight_decay 0.0 --weight_schedule karras
--data_dir /path/to/lsun_bedroom

################################################################
# Sampling from EDM models on class-conditional ImageNet-64
################################################################

mpiexec -n 8 python image_sample.py --training_mode edm --batch_size 64
--sigma_max 80 --sigma_min 0.002 --s_churn 0 --steps 40 --sampler heun
--model_path edm_imagenet64_ema.pt --attention_resolutions 32,16,8
--class_cond True --dropout 0.1 --image_size 64 --num_channels 192
--num_head_channels 64 --num_res_blocks 3 --num_samples 50000
--resblock_updown True --use_fp16 True --use_scale_shift_norm True
--weight_schedule karras
```

```
################################################################
# Consistency distillation on class-conditional ImageNet-64
################################################################

## L_CD^N (l2) on ImageNet-64
mpiexec -n 8 python cm_train.py --training_mode consistency_distillation
--target_ema_mode fixed --start_ema 0.95 --scale_mode fixed --start_scales 40
--total_training_steps 600000 --loss_norm l2 --lr_anneal_steps 0
--teacher_model_path /path/to/edm_imagenet64_ema.pt
--attention_resolutions 32,16,8 --class_cond True --use_scale_shift_norm True
--dropout 0.0 --teacher_dropout 0.1 --ema_rate 0.999,0.9999,0.9999432189950708
--global_batch_size 2048 --image_size 64 --lr 0.000008 --num_channels 192
--num_head_channels 64 --num_res_blocks 3 --resblock_updown True
--schedule_sampler uniform --use_fp16 True --weight_decay 0.0 --weight_schedule
uniform --data_dir /path/to/data

################################################################
# Consistency training on class-conditional ImageNet-64
################################################################

## L_CT^N on ImageNet-64
mpiexec -n 8 python cm_train.py --training_mode consistency_training
--target_ema_mode adaptive --start_ema 0.95 --scale_mode progressive
--start_scales 2 --end_scales 200 --total_training_steps 800000 --loss_norm lpips
--lr_anneal_steps 0 --teacher_model_path /path/to/edm_imagenet64_ema.pt
--attention_resolutions 32,16,8 --class_cond True --use_scale_shift_norm True
--dropout 0.0 --teacher_dropout 0.1 --ema_rate 0.999,0.9999,0.9999432189950708
--global_batch_size 2048 --image_size 64 --lr 0.0001 --num_channels 192
--num_head_channels 64 --num_res_blocks 3 --resblock_updown True
--schedule_sampler uniform --use_fp16 True --weight_decay 0.0 --weight_schedule
uniform --data_dir /path/to/imagenet64
```

# Consistency Models; Official Code

**scripts/launch.sh**

```
######################################################################
# Sampling from consistency models on class-conditional ImageNet-64
######################################################################

## ImageNet-64
mpiexec -n 8 python image_sample.py --batch_size 256 --training_mode
consistency_distillation --sampler onestep --model_path /path/to/checkpoint
--attention_resolutions 32,16,8 --class_cond True --use_scale_shift_norm True
--dropout 0.0 --image_size 64 --num_channels 192 --num_head_channels 64
--num_res_blocks 3 --num_samples 500 --resblock_updown True --use_fp16 True
--weight_schedule uniform
```

```
######################################################################
Tenary search for multi-step sampling on class-conditional ImageNet-64
######################################################################

## CD on ImageNet-64
mpiexec -n 8 python ternary_search.py --begin 0 --end 39 --steps 40 --generator
determ --ref_batch /root/consistency/ref_batches/imagenet64.npz --batch_size 256
--model_path /root/consistency/cd_imagenet64_lpips.pt
--attention_resolutions 32,16,8 --class_cond True --use_scale_shift_norm True
--dropout 0.0 --image_size 64 --num_channels 192 --num_head_channels 64
--num_res_blocks 3 --num_samples 50000 --resblock_updown True --use_fp16 True
--weight_schedule uniform

## CT on ImageNet-64
mpiexec -n 8 python ternary_search.py --begin 0 --end 200 --steps 201 --generator
determ --ref_batch /root/consistency/ref_batches/imagenet64.npz --batch_size 256
--model_path /root/consistency/ct_imagenet64.pt
--attention_resolutions 32,16,8 --class_cond True --use_scale_shift_norm True
--dropout 0.0 --image_size 64 --num_channels 192 --num_head_channels 64
--num_res_blocks 3 --num_samples 50000 --resblock_updown True --use_fp16 True
--weight_schedule uniform
```

# Consistency Models; Official Code

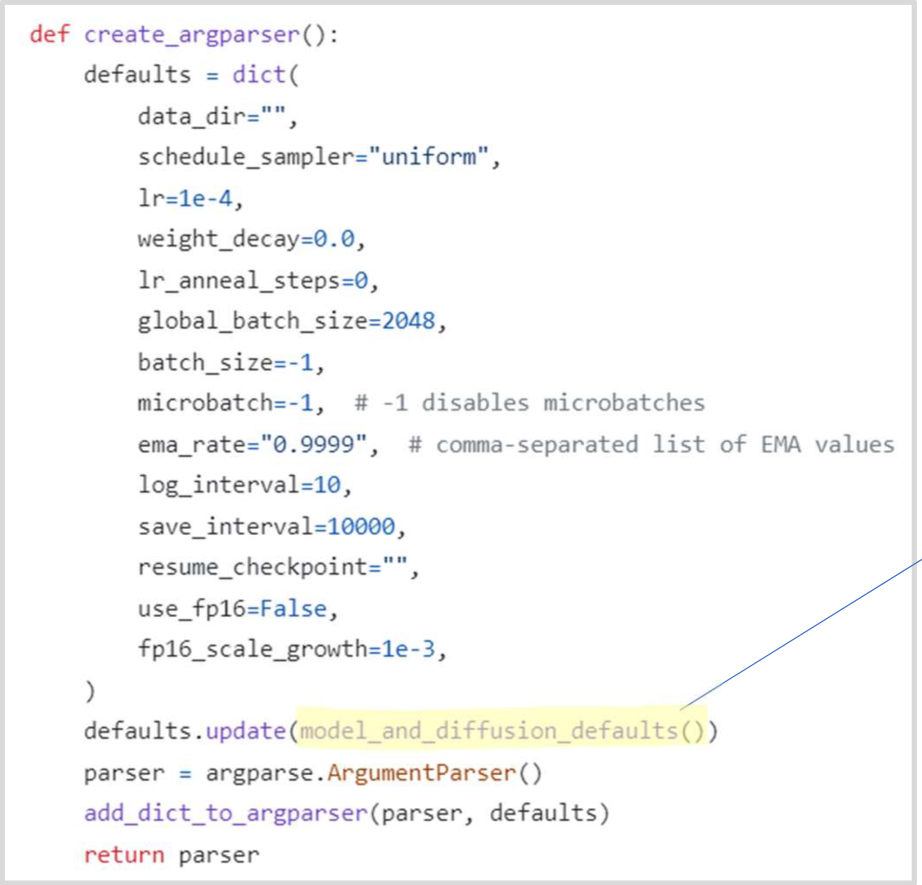
**scripts/launch.sh**

```
######################################################################
# Multistep sampling on class-conditional ImageNet-64
######################################################################

## Two-step sampling for CD (LPIPS) on ImageNet-64
mpiexec -n 8 python image_sample.py --batch_size 256 --training_mode
consistency_distillation --sampler multistep --ts 0,22,39 --steps 40 --model_path
/path/to/cd_imagenet64_lpips.pt --attention_resolutions 32,16,8 --class_cond True
--use_scale_shift_norm True --dropout 0.0 --image_size 64 --num_channels 192
--num_head_channels 64 --num_res_blocks 3 --num_samples 500
--resblock_updown True --use_fp16 True --weight_schedule uniform

## Two-step sampling for CD (L2) on ImageNet-64
mpiexec -n 8 python image_sample.py --batch_size 256 --training_mode
consistency_distillation --sampler multistep --ts 0,22,39 --steps 40 --model_path
/path/to/cd_imagenet64_l2.pt --attention_resolutions 32,16,8 --class_cond True
--use_scale_shift_norm True --dropout 0.0 --image_size 64 --num_channels 192
--num_head_channels 64 --num_res_blocks 3 --num_samples 500
--resblock_updown True --use_fp16 True --weight_schedule uniform

## Two-step sampling for CT on ImageNet-64
mpiexec -n 8 python image_sample.py --batch_size 256 --training_mode
consistency_training --sampler multistep --ts 0,106,200 --steps 201 --model_path
/path/to/ct_imagenet64.pt --attention_resolutions 32,16,8 --class_cond True
--use_scale_shift_norm True --dropout 0.0 --image_size 64 --num_channels 192
--num_head_channels 64 --num_res_blocks 3 --num_samples 500
--resblock_updown True --use_fp16 True --weight_schedule uniform
```

## EDM (Karras et al. (2022))

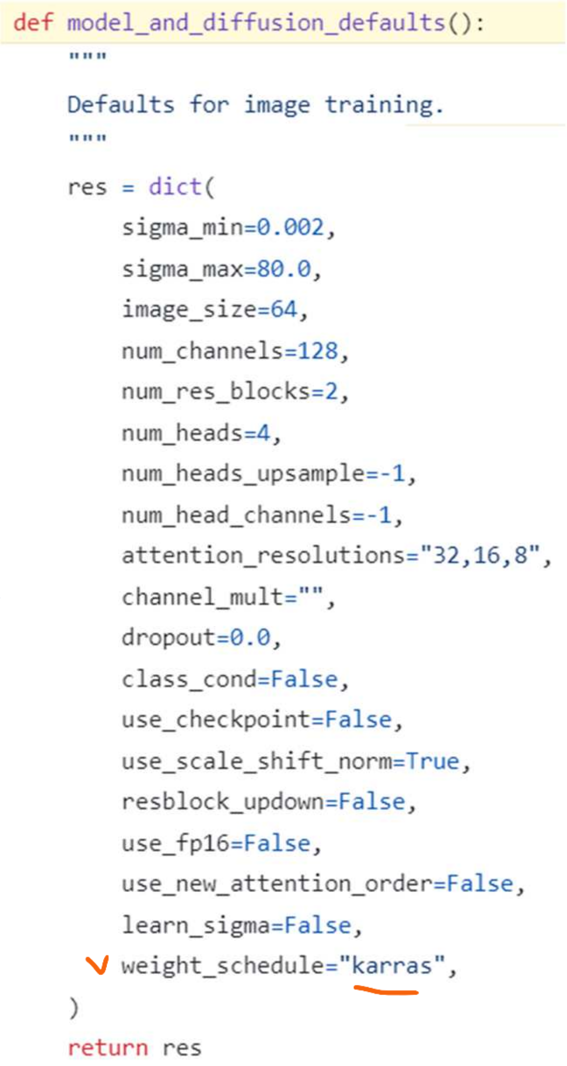
**Karras, T., Aittala, M., Aila, T., and Laine, S. Elucidating the design space of diffusion-based generative models. In Proc. NeurIPS, 2022**

### scripts/edm_train.py



```
def create_argparser():
    defaults = dict(
        data_dir="",
        schedule_sampler="uniform",
        lr=1e-4,
        weight_decay=0.0,
        lr_anneal_steps=0,
        global_batch_size=2048,
        batch_size=-1,
        microbatch=-1,  # -1 disables microbatches
        ema_rate="0.9999",  # comma-separated list of EMA values
        log_interval=10,
        save_interval=10000,
        resume_checkpoint="",
        use_fp16=False,
        fp16_scale_growth=1e-3,
    )
    defaults.update(model_and_diffusion_defaults())
    parser = argparse.ArgumentParser()
    add_dict_to_argparser(parser, defaults)
    return parser
```

cm/dist_util.py : model_and_diffusion_defaults()

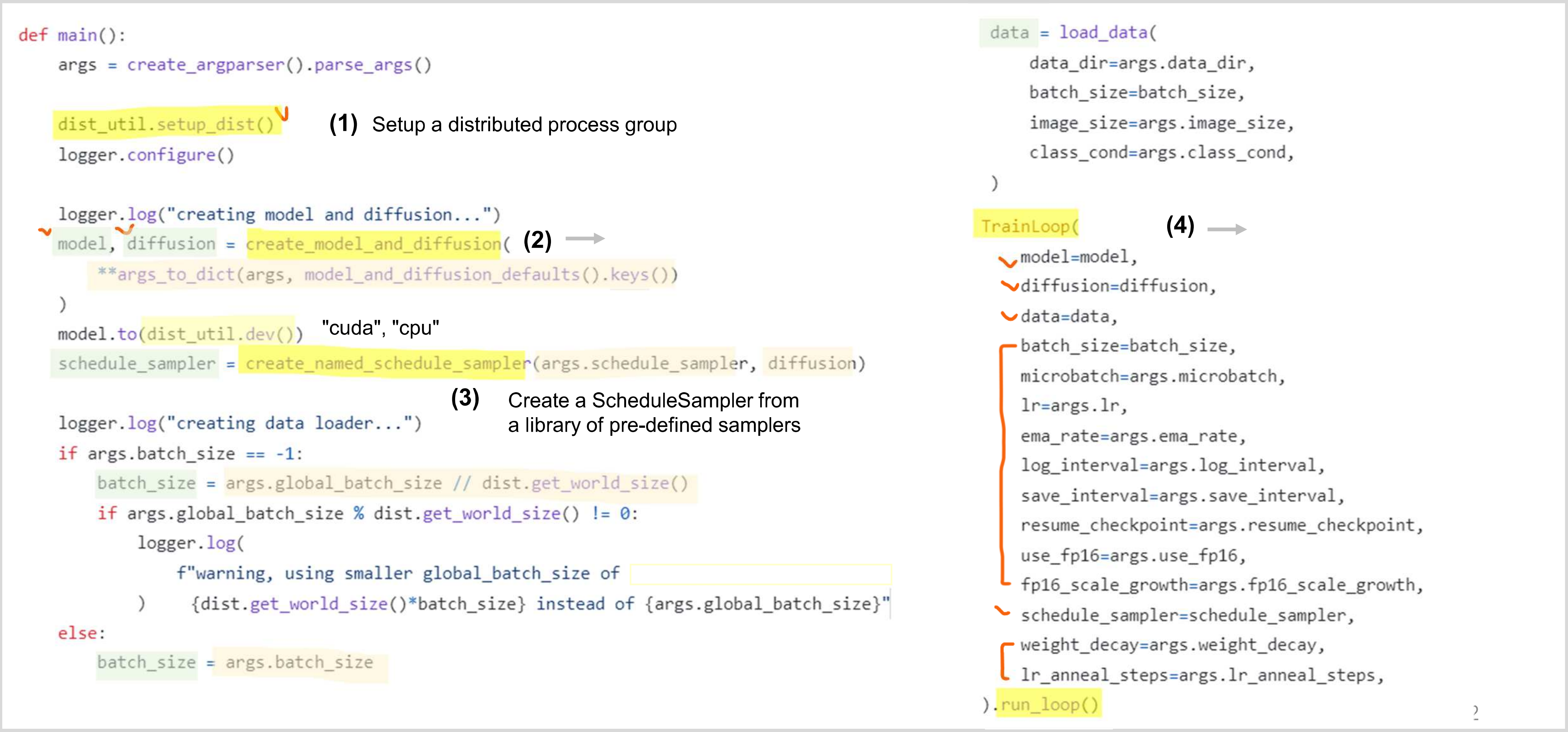


```
def model_and_diffusion_defaults():
    """
    Defaults for image training.
    """
    res = dict(
        sigma_min=0.002,
        sigma_max=80.0,
        image_size=64,
        num_channels=128,
        num_res_blocks=2,
        num_heads=4,
        num_heads_upsample=-1,
        num_head_channels=-1,
        attention_resolutions="32,16,8",
        channel_mult="",
        dropout=0.0,
        class_cond=False,
        use_checkpoint=False,
        use_scale_shift_norm=True,
        resblock_updown=False,
        use_fp16=False,
        use_new_attention_order=False,
        learn_sigma=False,
        weight_schedule="karras",
    )
    return res
```

```
mpiexec -n 8 python edm_train.py
--attention_resolutions 32,16,8
--class_cond True
--use_scale_shift_norm True
--dropout 0.1
--ema_rate
0.999,0.9999,0.9999432189950708
--global_batch_size 4096
--image_size 64
--lr 0.0001
--num_channels 192
--num_head_channels 64
--num_res_blocks 3
--resblock_updown True
--schedule_sampler lognormal
--use_fp16 True
--weight_decay 0.0
--weight_schedule karras
--data_dir /path/to/imagenet
```

## EDM (Karras et al. (2022))

**scripts/edm_train.py**



```
def main():
    args = create_argparser().parse_args()

    dist_util.setup_dist()
    logger.configure()

    logger.log("creating model and diffusion...")
    model, diffusion = create_model_and_diffusion(
        **args_to_dict(args, model_and_diffusion_defaults().keys())
    )
    model.to(dist_util.dev())
    schedule_sampler = create_named_schedule_sampler(args.schedule_sampler, diffusion)

    logger.log("creating data loader...")
    if args.batch_size == -1:
        batch_size = args.global_batch_size // dist.get_world_size()
        if args.global_batch_size % dist.get_world_size() != 0:
            logger.log(
                f"warning, using smaller global_batch_size of
            )    {dist.get_world_size()*batch_size} instead of {args.global_batch_size}"
    else:
        batch_size = args.batch_size
```

**(1)** Setup a distributed process group

**(2)** →

"cuda", "cpu"

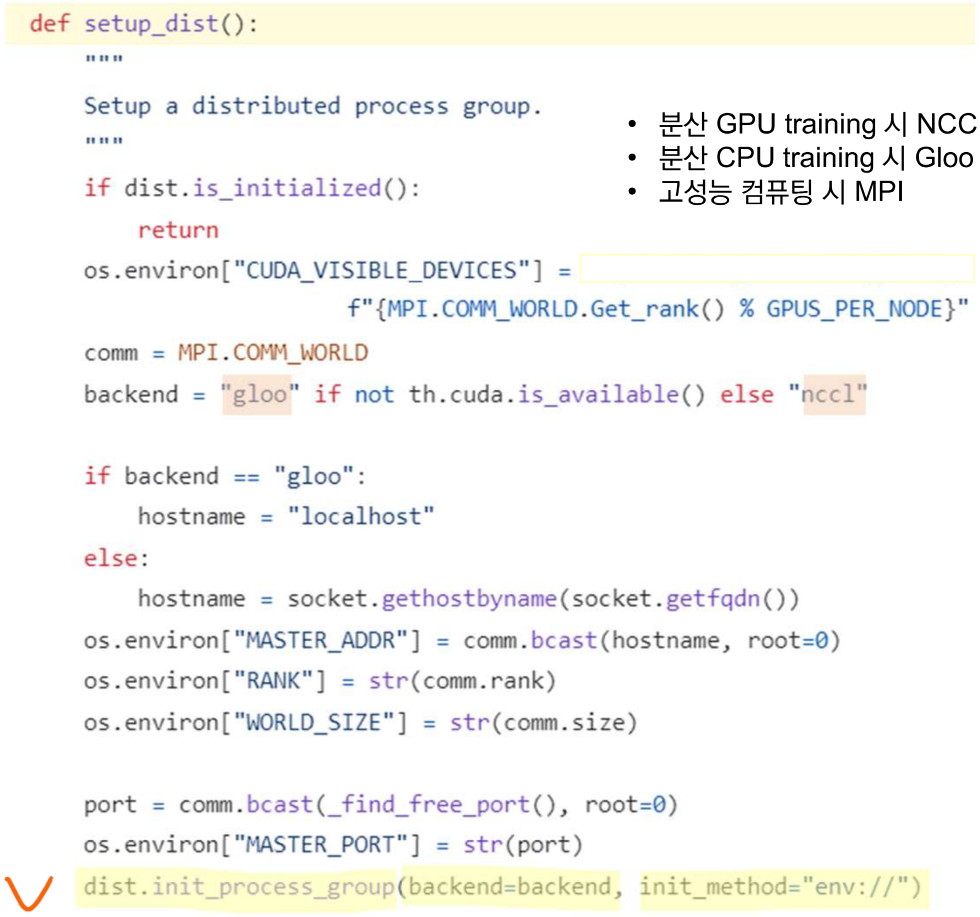
**(3)** Create a ScheduleSampler from a library of pre-defined samplers

```
data = load_data(
    data_dir=args.data_dir,
    batch_size=batch_size,
    image_size=args.image_size,
    class_cond=args.class_cond,
)

TrainLoop(
    model=model,
    diffusion=diffusion,
    data=data,
    batch_size=batch_size,
    microbatch=args.microbatch,
    lr=args.lr,
    ema_rate=args.ema_rate,
    log_interval=args.log_interval,
    save_interval=args.save_interval,
    resume_checkpoint=args.resume_checkpoint,
    use_fp16=args.use_fp16,
    fp16_scale_growth=args.fp16_scale_growth,
    schedule_sampler=schedule_sampler,
    weight_decay=args.weight_decay,
    lr_anneal_steps=args.lr_anneal_steps,
).run_loop()
```

**(4)** →

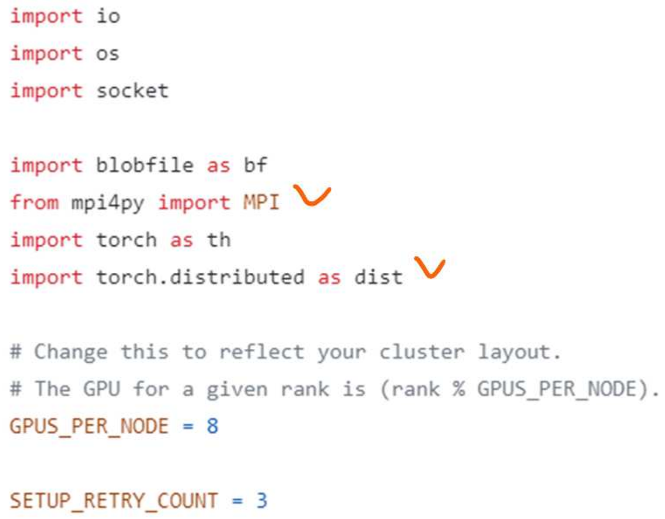
# EDM (Karras et al. (2022))

## (1) cm/dist_util.py : *setpup_dist()*



```
def setup_dist():
    """
    Setup a distributed process group.
    """
    if dist.is_initialized():
        return
    os.environ["CUDA_VISIBLE_DEVICES"] =
                        f"{MPI.COMM_WORLD.Get_rank() % GPUS_PER_NODE}"
    comm = MPI.COMM_WORLD
    backend = "gloo" if not th.cuda.is_available() else "nccl"

    if backend == "gloo":
        hostname = "localhost"
    else:
        hostname = socket.gethostbyname(socket.getfqdn())
    os.environ["MASTER_ADDR"] = comm.bcast(hostname, root=0)
    os.environ["RANK"] = str(comm.rank)
    os.environ["WORLD_SIZE"] = str(comm.size)

    port = comm.bcast(_find_free_port(), root=0)
    os.environ["MASTER_PORT"] = str(port)
    dist.init_process_group(backend=backend, init_method="env://")
```

- 분산 GPU training 시 NCCL
- 분산 CPU training 시 Gloo
- 고성능 컴퓨팅 시 MPI

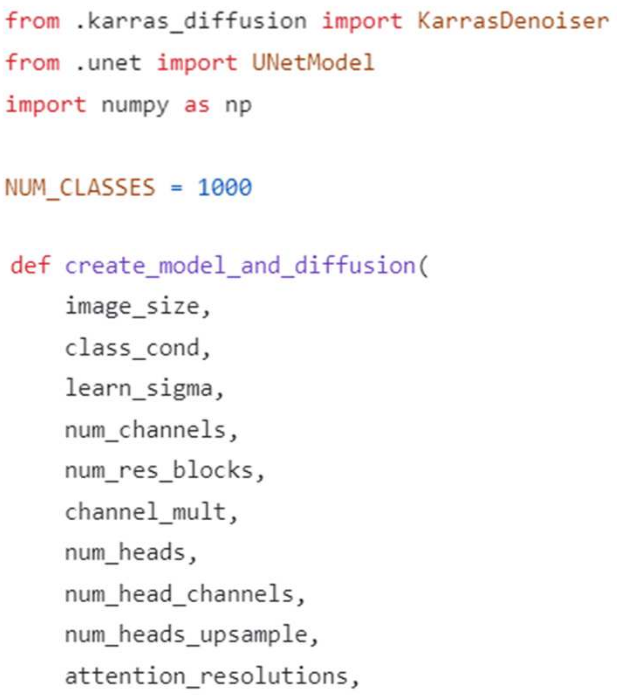


```
import io
import os
import socket

import blobfile as bf
from mpi4py import MPI
import torch as th
import torch.distributed as dist

# Change this to reflect your cluster layout.
# The GPU for a given rank is (rank % GPUS_PER_NODE).
GPUS_PER_NODE = 8

SETUP_RETRY_COUNT = 3
```

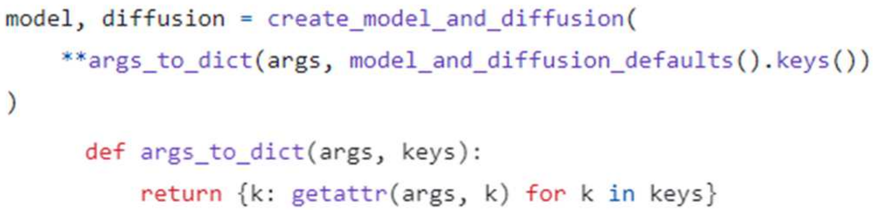
- [MASTER_PORT] 0-순위의 프로세스를 호스트할 기기의 비어있는 포트 번호(free port)
- [MASTER_ADDR] 0-순위의 프로세스를 호스트할 기기의 IP 주소
- [WORLD_SIZE] : Number of processes participating in the job (작업에 사용되는 프로세스들의 개수) 즉, 분산 처리에서 사용할 총 gpu 개수
- [RANK] : Data Distributed Parallel에서 가동되는 process ID
  - ✓ Global Rank: 전체 node에 가동되는 process id
  - ✓ Local Rank: 각 node별 process id
- [Local Rank] : 노드 내 프로세스의 로컬 순위
  - ✓ Local Rank를 0으로 한다면 0번째 GPU를 우선순위로 작업이 진행되는 듯.

# EDM (Karras et al. (2022))

## (2) cm/script_util.py : *create_model_and_diffusion*



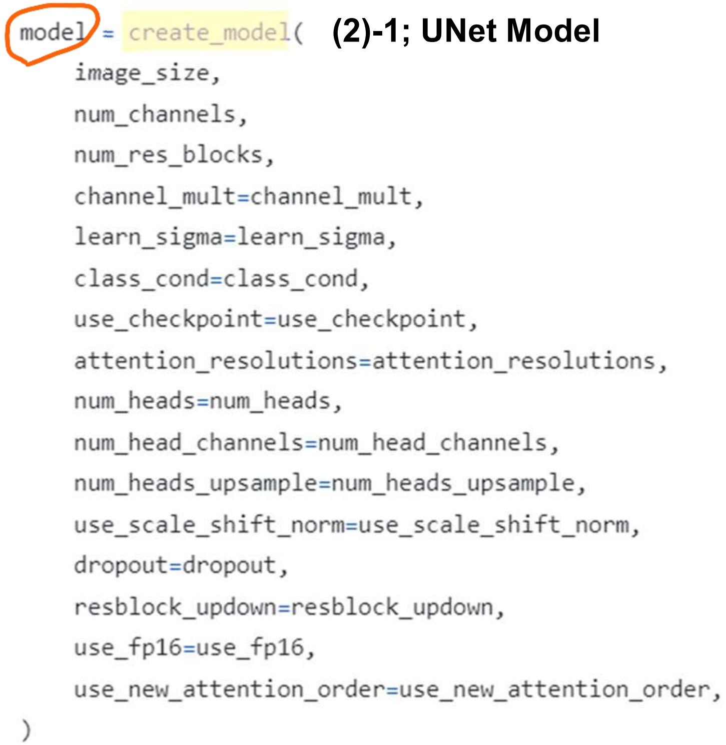
```python
from .karras_diffusion import KarrasDenoiser
from .unet import UNetModel
import numpy as np

NUM_CLASSES = 1000

def create_model_and_diffusion(
    image_size,
    class_cond,
    learn_sigma,
    num_channels,
    num_res_blocks,
    channel_mult,
    num_heads,
    num_head_channels,
    num_heads_upsample,
    attention_resolutions,
```



```python
model, diffusion = create_model_and_diffusion(
    **args_to_dict(args, model_and_diffusion_defaults().keys())
)
```

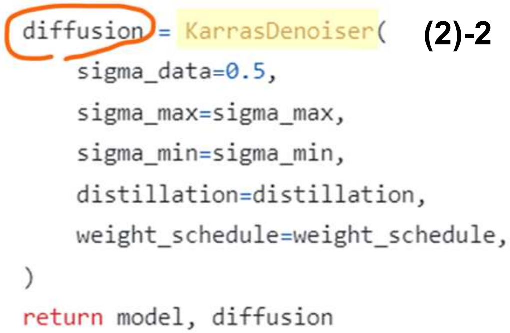
```python
    def args_to_dict(args, keys):
        return {k: getattr(args, k) for k in keys}
```

(2)-1; UNet Model



```python
model = create_model(
    image_size,
    num_channels,
    num_res_blocks,
    channel_mult=channel_mult,
    learn_sigma=learn_sigma,
    class_cond=class_cond,
    use_checkpoint=use_checkpoint,
    attention_resolutions=attention_resolutions,
    num_heads=num_heads,
    num_head_channels=num_head_channels,
    num_heads_upsample=num_heads_upsample,
    use_scale_shift_norm=use_scale_shift_norm,
    dropout=dropout,
    resblock_updown=resblock_updown,
    use_fp16=use_fp16,
    use_new_attention_order=use_new_attention_order,
)
```

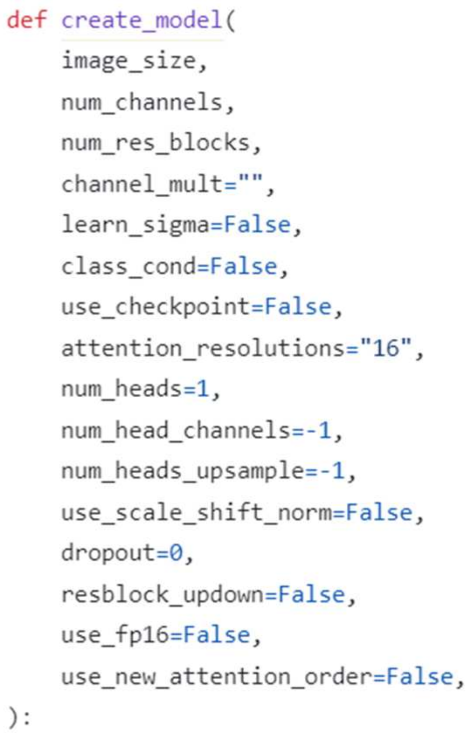
(2)-2



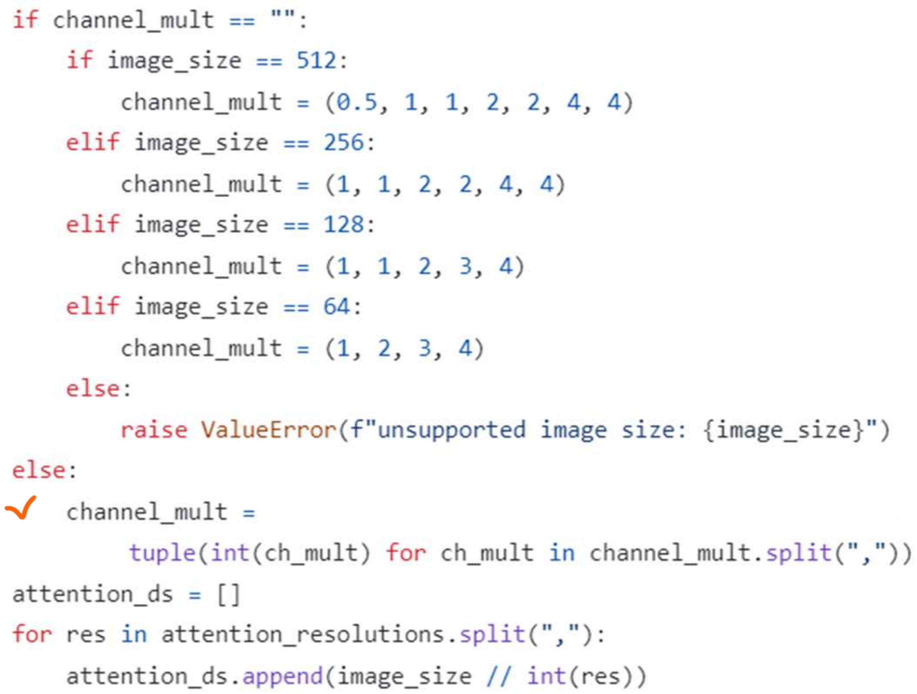
```python
diffusion = KarrasDenoiser(
    sigma_data=0.5,
    sigma_max=sigma_max,
    sigma_min=sigma_min,
    distillation=distillation,
    weight_schedule=weight_schedule,
)
return model, diffusion
```

# EDM (Karras et al. (2022))

## (2)-1 cm/script_util.py : *create_model*



```python
def create_model(
    image_size,
    num_channels,
    num_res_blocks,
    channel_mult="",
    learn_sigma=False,
    class_cond=False,
    use_checkpoint=False,
    attention_resolutions="16",
    num_heads=1,
    num_head_channels=-1,
    num_heads_upsample=-1,
    use_scale_shift_norm=False,
    dropout=0,
    resblock_updown=False,
    use_fp16=False,
    use_new_attention_order=False,
):
```

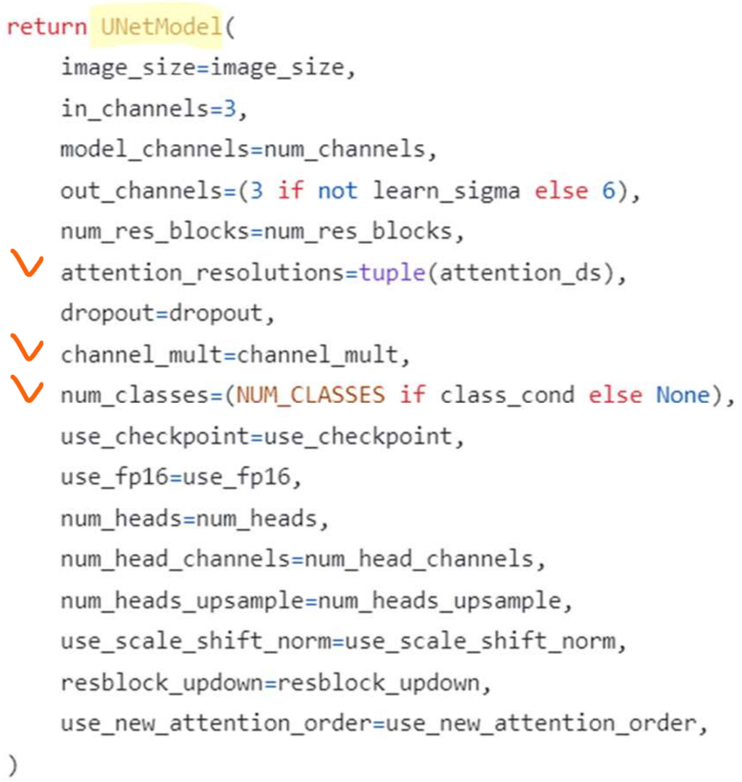


```python
if channel_mult == "":
    if image_size == 512:
        channel_mult = (0.5, 1, 1, 2, 2, 4, 4)
    elif image_size == 256:
        channel_mult = (1, 1, 2, 2, 4, 4)
    elif image_size == 128:
        channel_mult = (1, 1, 2, 3, 4)
    elif image_size == 64:
        channel_mult = (1, 2, 3, 4)
    else:
        raise ValueError(f"unsupported image size: {image_size}")
else:
    channel_mult =
        tuple(int(ch_mult) for ch_mult in channel_mult.split(","))
attention_ds = []
for res in attention_resolutions.split(","):
    attention_ds.append(image_size // int(res))
```

image_size == 64
channel_mult = (1,2,3,4)
attention_resolutions="32,16,8"

attention_ds = [2, 4, 8]

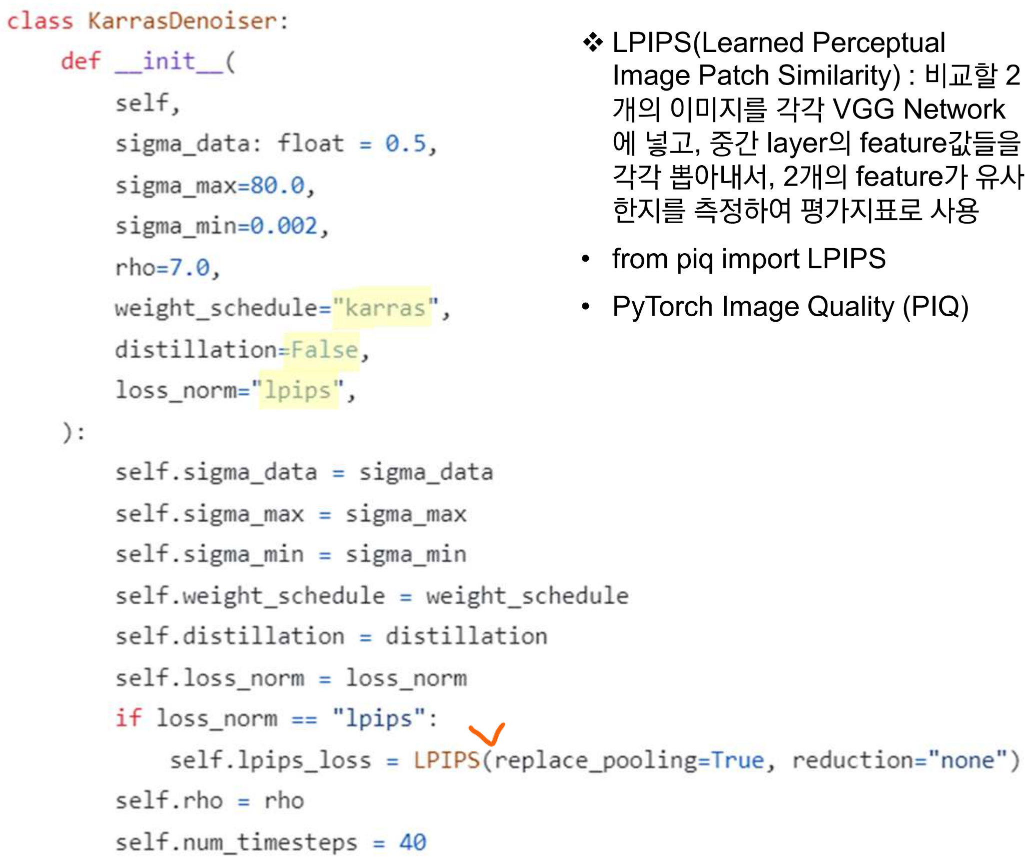
The full UNet model with attention and timestep embedding.



```python
return UNetModel(
    image_size=image_size,
    in_channels=3,
    model_channels=num_channels,
    out_channels=(3 if not learn_sigma else 6),
    num_res_blocks=num_res_blocks,
    attention_resolutions=tuple(attention_ds),
    dropout=dropout,
    channel_mult=channel_mult,
    num_classes=(NUM_CLASSES if class_cond else None),
    use_checkpoint=use_checkpoint,
    use_fp16=use_fp16,
    num_heads=num_heads,
    num_head_channels=num_head_channels,
    num_heads_upsample=num_heads_upsample,
    use_scale_shift_norm=use_scale_shift_norm,
    resblock_updown=resblock_updown,
    use_new_attention_order=use_new_attention_order,
)
```

# EDM (Karras et al. (2022))

## (2)-2 cm/karras_diffusion.py : *class karrasDenoiser*

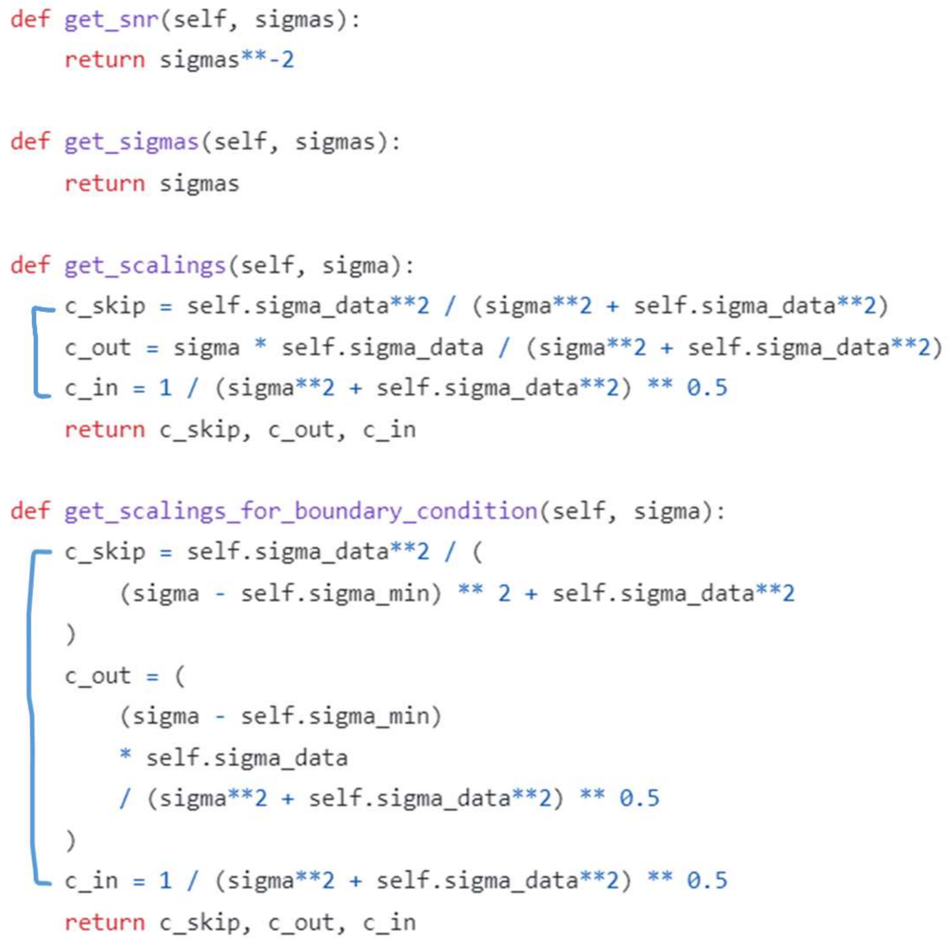


```python
class KarrasDenoiser:
    def __init__(
        self,
        sigma_data: float = 0.5,
        sigma_max=80.0,
        sigma_min=0.002,
        rho=7.0,
        weight_schedule="karras",
        distillation=False,
        loss_norm="lpips",
    ):
        self.sigma_data = sigma_data
        self.sigma_max = sigma_max
        self.sigma_min = sigma_min
        self.weight_schedule = weight_schedule
        self.distillation = distillation
        self.loss_norm = loss_norm
        if loss_norm == "lpips":
            self.lpips_loss = LPIPS(replace_pooling=True, reduction="none")
        self.rho = rho
        self.num_timesteps = 40
```

❖ LPIPS(Learned Perceptual Image Patch Similarity) : 비교할 2개의 이미지를 각각 VGG Network에 넣고, 중간 layer의 feature값들을 각각 뽑아내서, 2개의 feature가 유사한지를 측정하여 평가지표로 사용

- from piq import LPIPS
- PyTorch Image Quality (PIQ)

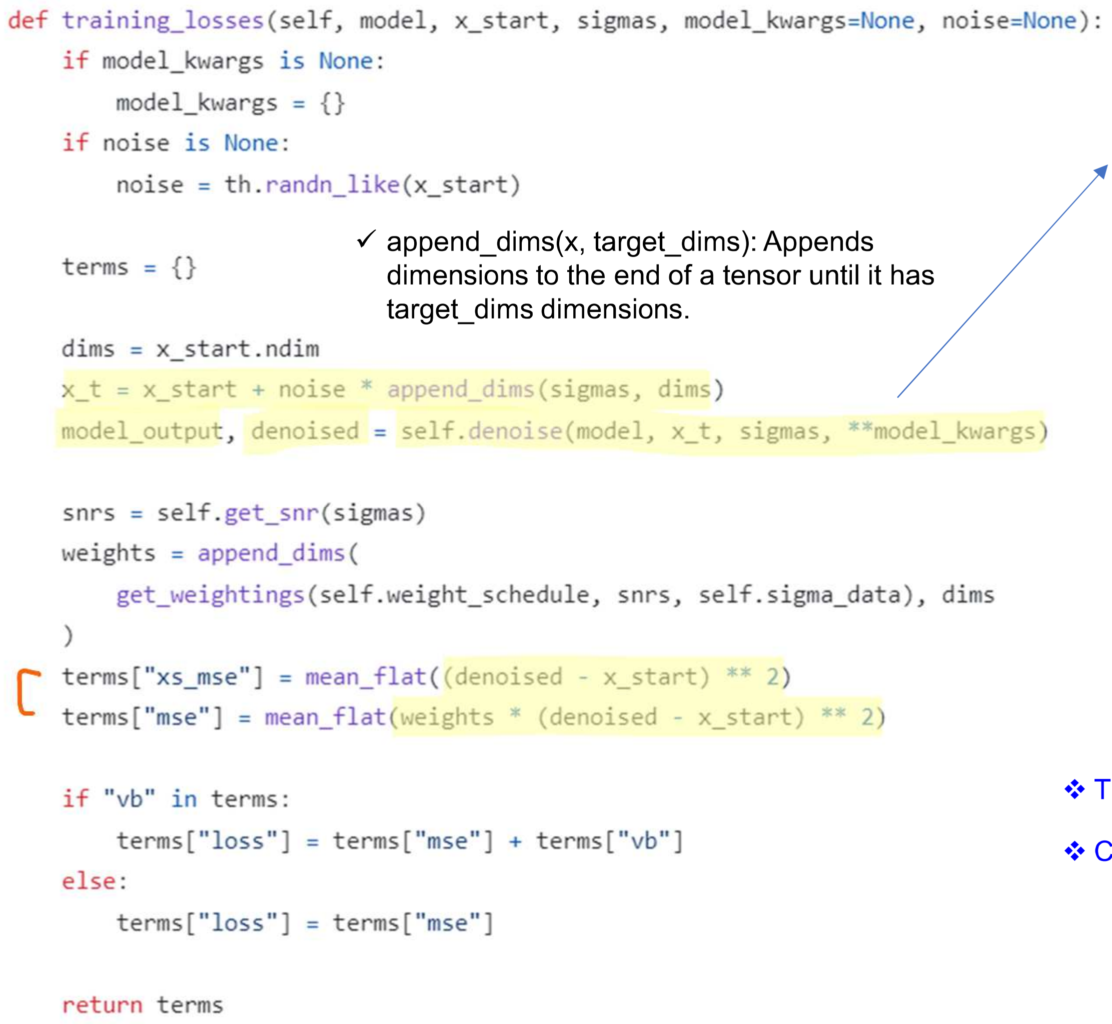
Based on https://github.com/crowsonkb/k-diffusion
Elucidating the Design Space of Diffusion-Based Generative Models (Karras et al., 2022)



```python
def get_snr(self, sigmas):
    return sigmas**-2

def get_sigmas(self, sigmas):
    return sigmas

def get_scalings(self, sigma):
    c_skip = self.sigma_data**2 / (sigma**2 + self.sigma_data**2)
    c_out = sigma * self.sigma_data / (sigma**2 + self.sigma_data**2)
    c_in = 1 / (sigma**2 + self.sigma_data**2) ** 0.5
    return c_skip, c_out, c_in

def get_scalings_for_boundary_condition(self, sigma):
    c_skip = self.sigma_data**2 / (
        (sigma - self.sigma_min) ** 2 + self.sigma_data**2
    )
    c_out = (
        (sigma - self.sigma_min)
        * self.sigma_data
        / (sigma**2 + self.sigma_data**2) ** 0.5
    )
    c_in = 1 / (sigma**2 + self.sigma_data**2) ** 0.5
    return c_skip, c_out, c_in
```

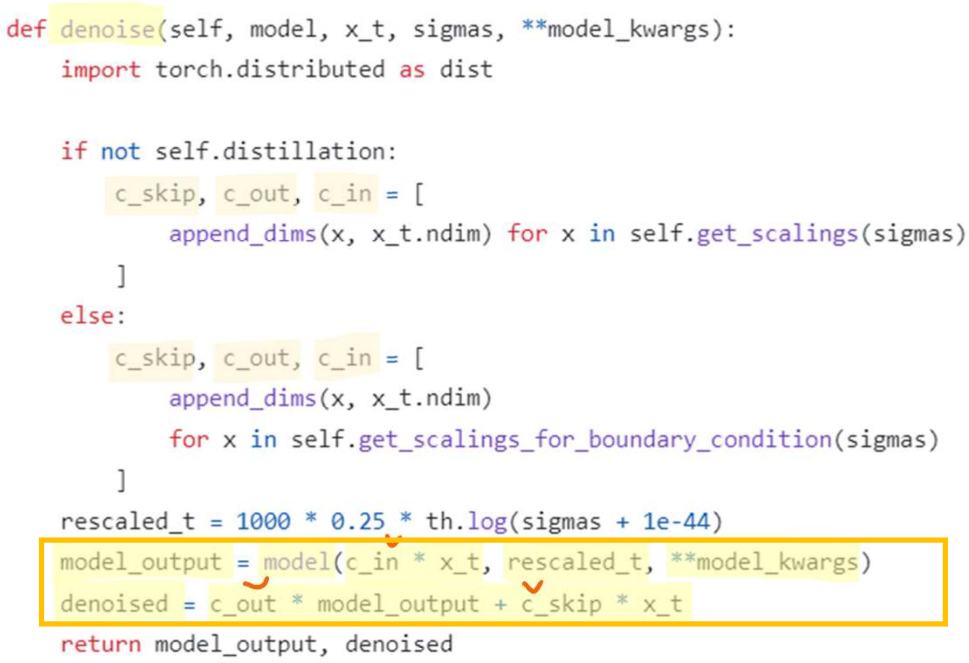
## EDM (Karras et al. (2022))

(2)-2 cm/karras_diffusion.py : *class karrasDenoiser*



```
def training_losses(self, model, x_start, sigmas, model_kwargs=None, noise=None):
    if model_kwargs is None:
        model_kwargs = {}
    if noise is None:
        noise = th.randn_like(x_start)

    terms = {}

    dims = x_start.ndim
    x_t = x_start + noise * append_dims(sigmas, dims)
    model_output, denoised = self.denoise(model, x_t, sigmas, **model_kwargs)

    snrs = self.get_snr(sigmas)
    weights = append_dims(
        get_weightings(self.weight_schedule, snrs, self.sigma_data), dims
    )
    terms["xs_mse"] = mean_flat((denoised - x_start) ** 2)
    terms["mse"] = mean_flat(weights * (denoised - x_start) ** 2)

    if "vb" in terms:
        terms["loss"] = terms["mse"] + terms["vb"]
    else:
        terms["loss"] = terms["mse"]

    return terms
```

✓ append_dims(x, target_dims): Appends dimensions to the end of a tensor until it has target_dims dimensions.

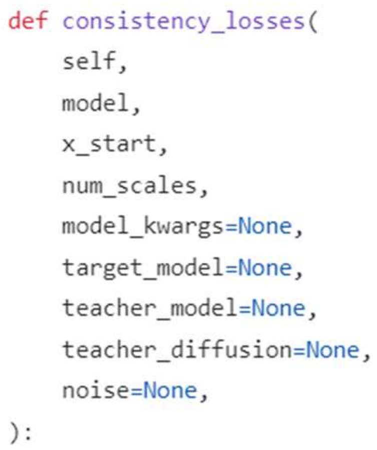


```
def denoise(self, model, x_t, sigmas, **model_kwargs):
    import torch.distributed as dist

    if not self.distillation:
        c_skip, c_out, c_in = [
            append_dims(x, x_t.ndim) for x in self.get_scalings(sigmas)
        ]
    else:
        c_skip, c_out, c_in = [
            append_dims(x, x_t.ndim)
            for x in self.get_scalings_for_boundary_condition(sigmas)
        ]
    rescaled_t = 1000 * 0.25 * th.log(sigmas + 1e-44)
    model_output = model(c_in * x_t, rescaled_t, **model_kwargs)
    denoised = c_out * model_output + c_skip * x_t
    return model_output, denoised
```

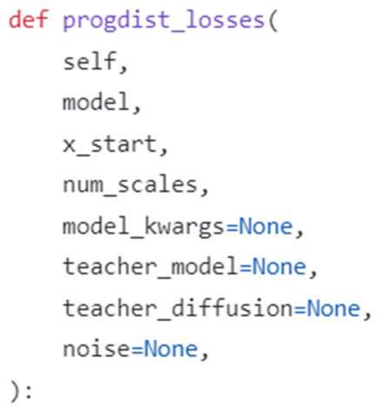
- ❖ TrainLoop Class : forward_backward-> training_losses
- ❖ CMTrainLoop Class : forward_backward -> consistency_losses, progdist_losses

# EDM (Karras et al. (2022))

## (2)-2 cm/karras_diffusion.py : *class karrasDenoiser*



```
def consistency_losses(
    self,
    model,
    x_start,
    num_scales,
    model_kwargs=None,
    target_model=None,
    teacher_model=None,
    teacher_diffusion=None,
    noise=None,
):
```



```
def progdist_losses(
    self,
    model,
    x_start,
    num_scales,
    model_kwargs=None,
    teacher_model=None,
    teacher_diffusion=None,
    noise=None,
):
```

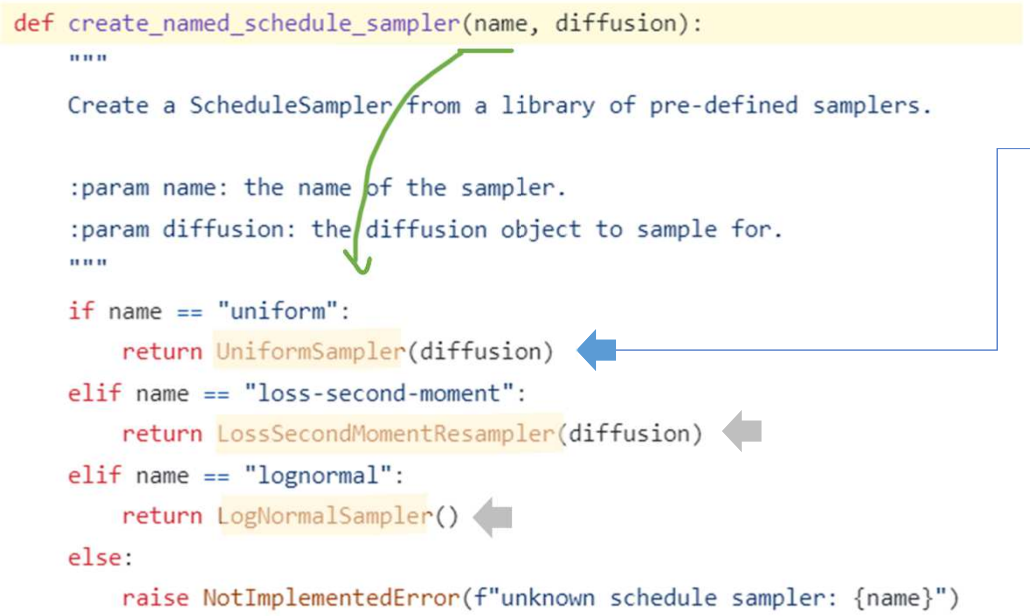
❖ CMTrainLoop Class에 사용

- ➢ forward_backward -> consistency_losses, progdist_losses

# EDM (Karras et al. (2022))

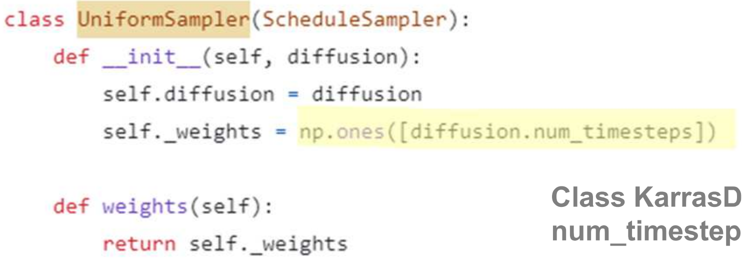
## (3) cm/resample.py

schedule_sampler="uniform"



```
def create_named_schedule_sampler(name, diffusion):
    """
    Create a ScheduleSampler from a library of pre-defined samplers.

    :param name: the name of the sampler.
    :param diffusion: the diffusion object to sample for.
    """
    if name == "uniform":
        return UniformSampler(diffusion)
    elif name == "loss-second-moment":
        return LossSecondMomentResampler(diffusion)
    elif name == "lognormal":
        return LogNormalSampler()
    else:
        raise NotImplementedError(f"unknown schedule sampler: {name}")
```

다음 페이지

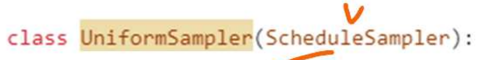


```
class UniformSampler(ScheduleSampler):
    def __init__(self, diffusion):
        self.diffusion = diffusion
        self._weights = np.ones([diffusion.num_timesteps])

    def weights(self):
        return self._weights
```

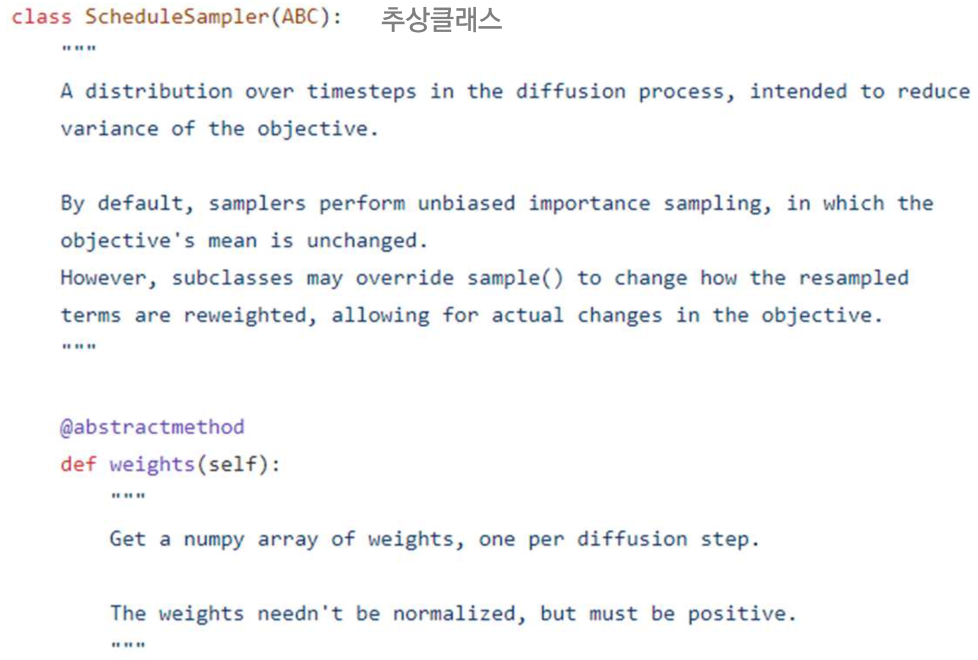
**Class KarrasDenoiser**
**num_timesteps = 40**

# EDM (Karras et al. (2022))

## (3) cm/resample.py

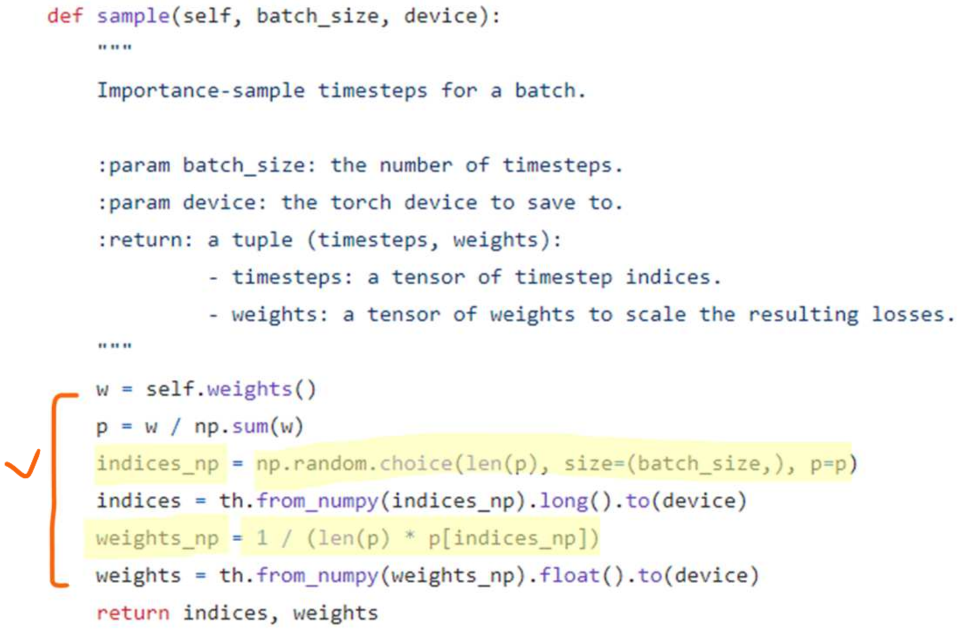


```
class UniformSampler(ScheduleSampler):
```



```
class ScheduleSampler(ABC):
    """
    A distribution over timesteps in the diffusion process, intended to reduce
    variance of the objective.

    By default, samplers perform unbiased importance sampling, in which the
    objective's mean is unchanged.
    However, subclasses may override sample() to change how the resampled
    terms are reweighted, allowing for actual changes in the objective.
    """

    @abstractmethod
    def weights(self):
        """
        Get a numpy array of weights, one per diffusion step.

        The weights needn't be normalized, but must be positive.
        """
```

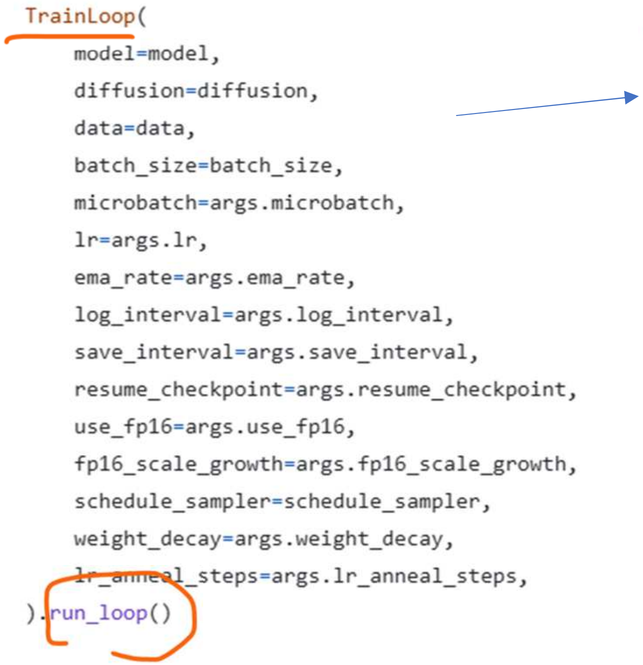
추상클래스



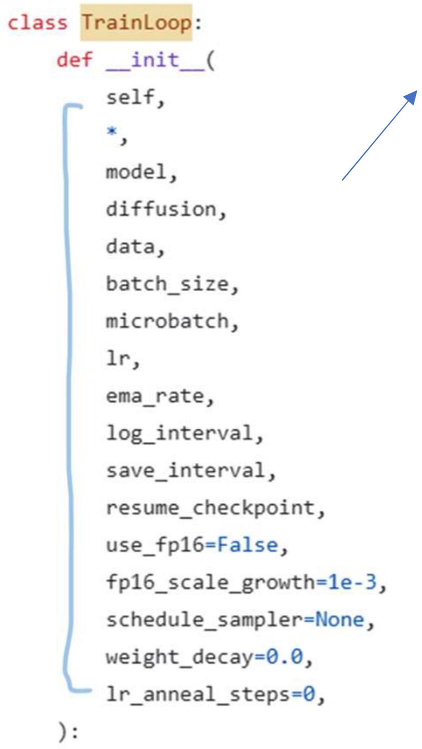
```
def sample(self, batch_size, device):
    """
    Importance-sample timesteps for a batch.

    :param batch_size: the number of timesteps.
    :param device: the torch device to save to.
    :return: a tuple (timesteps, weights):
             - timesteps: a tensor of timestep indices.
             - weights: a tensor of weights to scale the resulting losses.
    """
    w = self.weights()
    p = w / np.sum(w)
    indices_np = np.random.choice(len(p), size=(batch_size,), p=p)
    indices = th.from_numpy(indices_np).long().to(device)
    weights_np = 1 / (len(p) * p[indices_np])
    weights = th.from_numpy(weights_np).float().to(device)
    return indices, weights
```

## EDM (Karras et al. (2022))

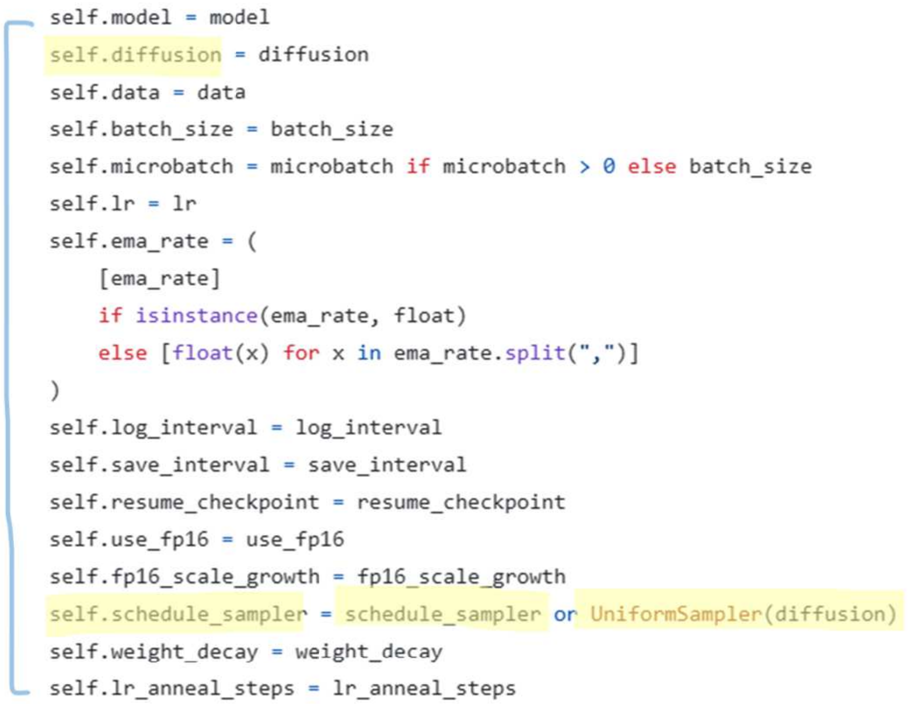
### (4) cm/train_util.py



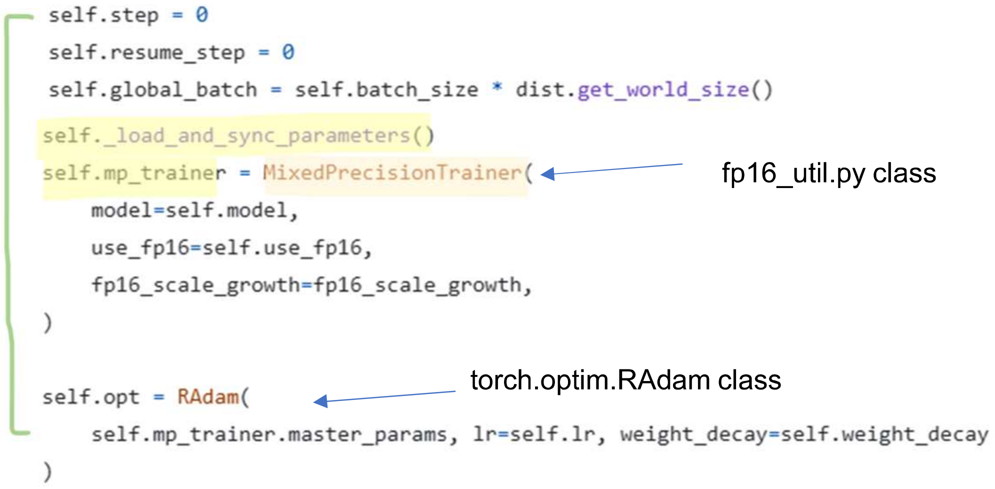
```
TrainLoop(
    model=model,
    diffusion=diffusion,
    data=data,
    batch_size=batch_size,
    microbatch=args.microbatch,
    lr=args.lr,
    ema_rate=args.ema_rate,
    log_interval=args.log_interval,
    save_interval=args.save_interval,
    resume_checkpoint=args.resume_checkpoint,
    use_fp16=args.use_fp16,
    fp16_scale_growth=args.fp16_scale_growth,
    schedule_sampler=schedule_sampler,
    weight_decay=args.weight_decay,
    lr_anneal_steps=args.lr_anneal_steps,
).run_loop()
```



```
class TrainLoop:
    def __init__(
        self,
        *,
        model,
        diffusion,
        data,
        batch_size,
        microbatch,
        lr,
        ema_rate,
        log_interval,
        save_interval,
        resume_checkpoint,
        use_fp16=False,
        fp16_scale_growth=1e-3,
        schedule_sampler=None,
        weight_decay=0.0,
        lr_anneal_steps=0,
    ):
```



```
self.model = model
self.diffusion = diffusion
self.data = data
self.batch_size = batch_size
self.microbatch = microbatch if microbatch > 0 else batch_size
self.lr = lr
self.ema_rate = (
    [ema_rate]
    if isinstance(ema_rate, float)
    else [float(x) for x in ema_rate.split(",")]
)
self.log_interval = log_interval
self.save_interval = save_interval
self.resume_checkpoint = resume_checkpoint
self.use_fp16 = use_fp16
self.fp16_scale_growth = fp16_scale_growth
self.schedule_sampler = schedule_sampler or UniformSampler(diffusion)
self.weight_decay = weight_decay
self.lr_anneal_steps = lr_anneal_steps
```



```
self.step = 0
self.resume_step = 0
self.global_batch = self.batch_size * dist.get_world_size()

self._load_and_sync_parameters()
self.mp_trainer = MixedPrecisionTrainer(
    model=self.model,
    use_fp16=self.use_fp16,
    fp16_scale_growth=fp16_scale_growth,
)

self.opt = RAdam(
    self.mp_trainer.master_params, lr=self.lr, weight_decay=self.weight_decay
)
```

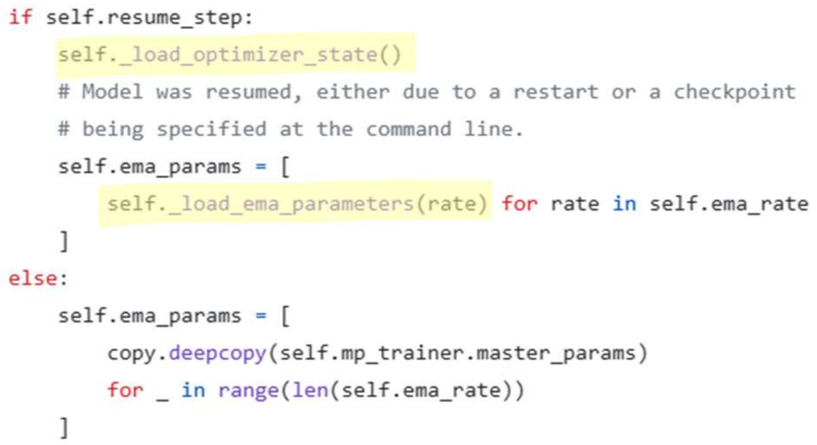
fp16_util.py class

torch.optim.RAdam class

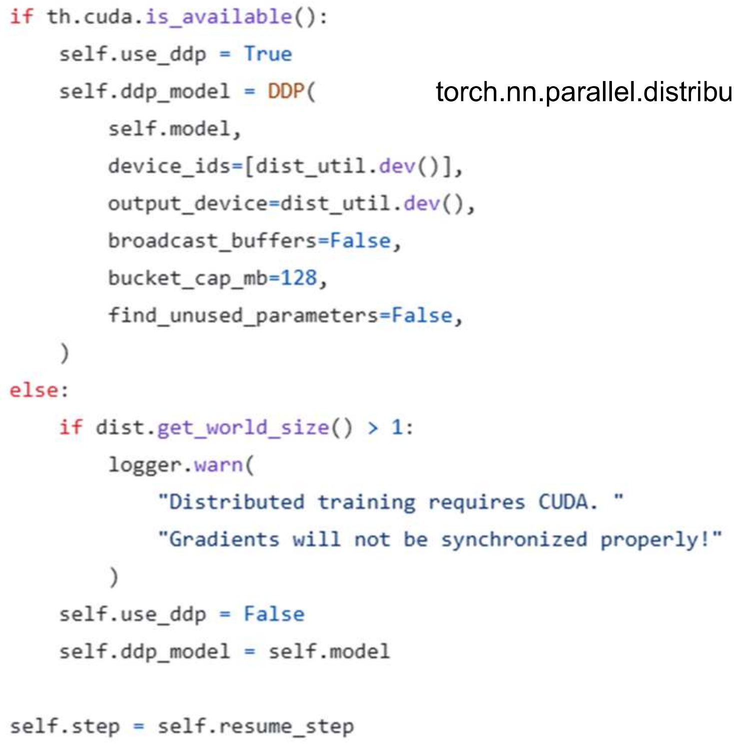
RAdam (Rectified Adam) : adaptive learning rate term의 분산을 rectify하여 학습의 안정성, 분산을 consistent하게 하는 rectification term을 구하고 이를 곱함

# EDM (Karras et al. (2022))

## (4) cm/train_util.py



```
if self.resume_step:
    self._load_optimizer_state()
    # Model was resumed, either due to a restart or a checkpoint
    # being specified at the command line.
    self.ema_params = [
        self._load_ema_parameters(rate) for rate in self.ema_rate
    ]
else:
    self.ema_params = [
        copy.deepcopy(self.mp_trainer.master_params)
        for _ in range(len(self.ema_rate))
    ]
```

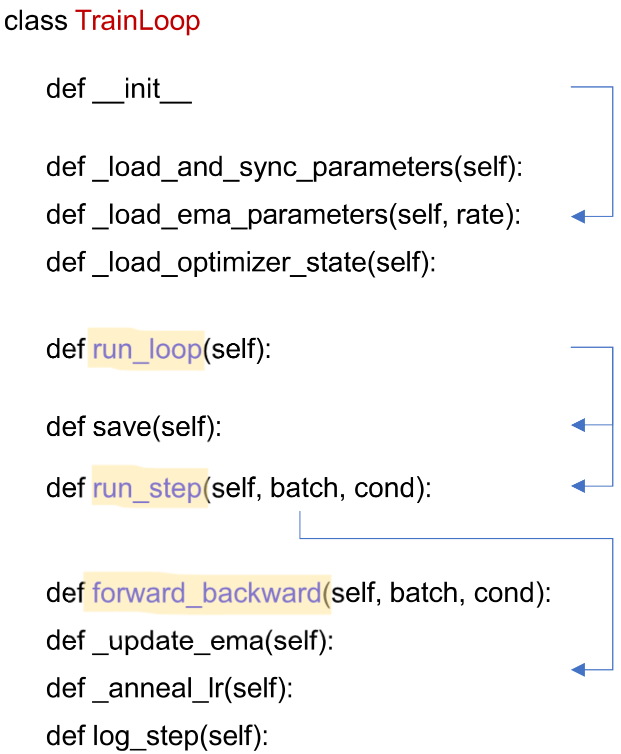


```
if th.cuda.is_available():
    self.use_ddp = True
    self.ddp_model = DDP(
        self.model,
        device_ids=[dist_util.dev()],
        output_device=dist_util.dev(),
        broadcast_buffers=False,
        bucket_cap_mb=128,
        find_unused_parameters=False,
    )
else:
    if dist.get_world_size() > 1:
        logger.warn(
            "Distributed training requires CUDA. "
            "Gradients will not be synchronized properly!"
        )
    self.use_ddp = False
    self.ddp_model = self.model

self.step = self.resume_step
```

torch.nn.parallel.distributed.DistributedDataParallel

# EDM (Karras et al. (2022))

## (4) cm/train_util.py



class TrainLoop

def __init__

def _load_and_sync_parameters(self):

def _load_ema_parameters(self, rate):

def _load_optimizer_state(self):

def run_loop(self):

def save(self):

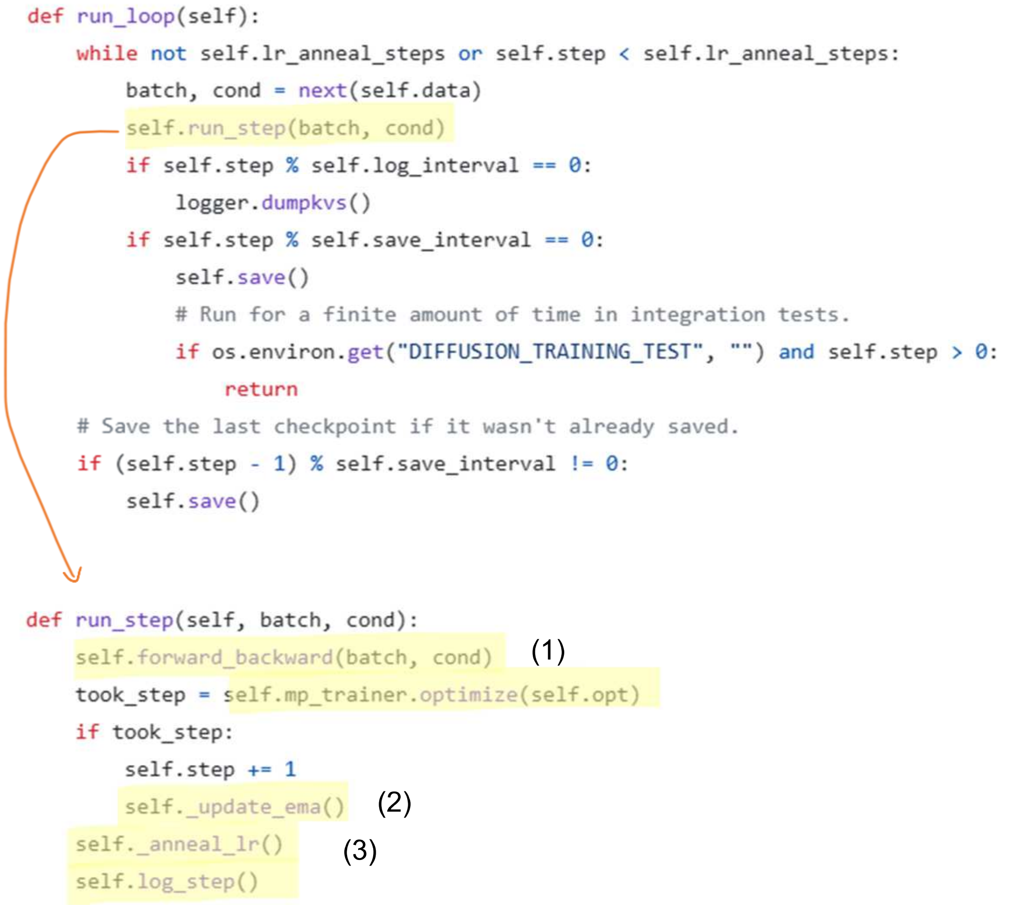
def run_step(self, batch, cond):

def forward_backward(self, batch, cond):

def _update_ema(self):

def _anneal_lr(self):

def log_step(self):



```
def run_loop(self):
    while not self.lr_anneal_steps or self.step < self.lr_anneal_steps:
        batch, cond = next(self.data)
        self.run_step(batch, cond)
        if self.step % self.log_interval == 0:
            logger.dumpkvs()
        if self.step % self.save_interval == 0:
            self.save()
            # Run for a finite amount of time in integration tests.
            if os.environ.get("DIFFUSION_TRAINING_TEST", "") and self.step > 0:
                return
    # Save the last checkpoint if it wasn't already saved.
    if (self.step - 1) % self.save_interval != 0:
        self.save()
```

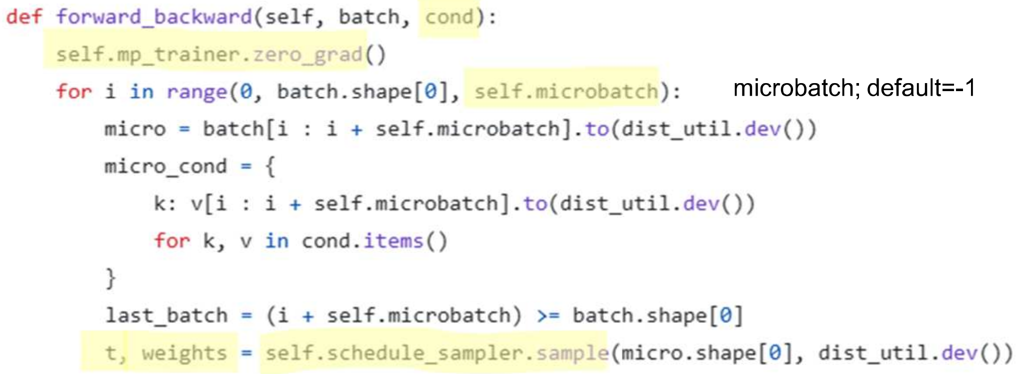
```
def run_step(self, batch, cond):
    self.forward_backward(batch, cond)
    took_step = self.mp_trainer.optimize(self.opt)
    if took_step:
        self.step += 1
        self._update_ema()
    self._anneal_lr()
    self.log_step()
```

# EDM (Karras et al. (2022))

## (4) cm/train_util.py

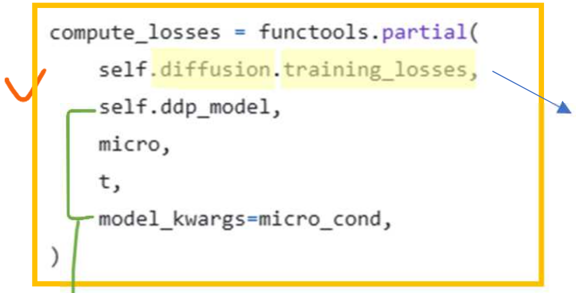
self.mp_trainer → class MixedPrecisionTrainer
self.schedule_sampler
self.diffusion

(1) class TrainLoop : method runloop → runstep → forward_backward

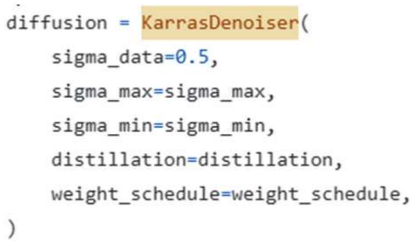


```
def forward_backward(self, batch, cond):
    self.mp_trainer.zero_grad()
    for i in range(0, batch.shape[0], self.microbatch):
        micro = batch[i : i + self.microbatch].to(dist_util.dev())
        micro_cond = {
            k: v[i : i + self.microbatch].to(dist_util.dev())
            for k, v in cond.items()
        }
        last_batch = (i + self.microbatch) >= batch.shape[0]
        t, weights = self.schedule_sampler.sample(micro.shape[0], dist_util.dev())
```

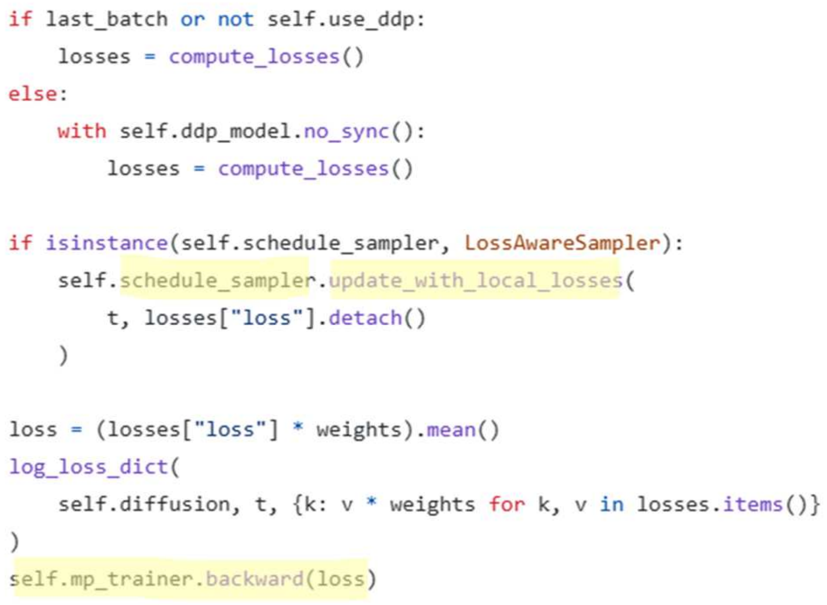
microbatch; default=-1



```
        compute_losses = functools.partial(
            self.diffusion.training_losses,
            self.ddp_model,
            micro,
            t,
            model_kwargs=micro_cond,
        )
```

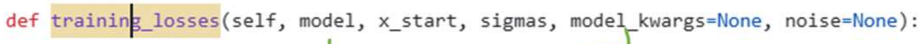


```
diffusion = KarrasDenoiser(
    sigma_data=0.5,
    sigma_max=sigma_max,
    sigma_min=sigma_min,
    distillation=distillation,
    weight_schedule=weight_schedule,
)
```



```
if last_batch or not self.use_ddp:
    losses = compute_losses()
else:
    with self.ddp_model.no_sync():
        losses = compute_losses()

if isinstance(self.schedule_sampler, LossAwareSampler):
    self.schedule_sampler.update_with_local_losses(
        t, losses["loss"].detach()
    )

loss = (losses["loss"] * weights).mean()
log_loss_dict(
    self.diffusion, t, {k: v * weights for k, v in losses.items()}
)
self.mp_trainer.backward(loss)
```

class KarrasDenoiser

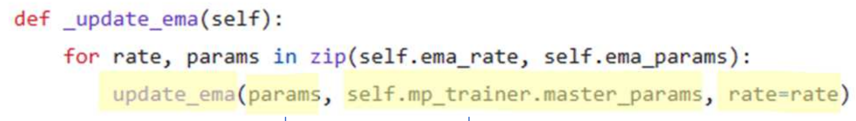


```
def training_losses(self, model, x_start, sigmas, model_kwargs=None, noise=None):
```

# EDM (Karras et al. (2022))

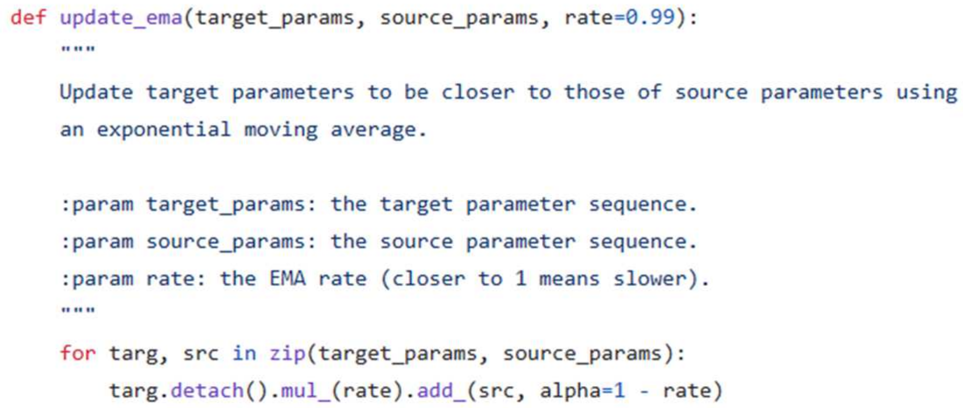
## (4) cm/train_util.py

(2) class TrainLoop : method runloop → runstep → _update_ema()



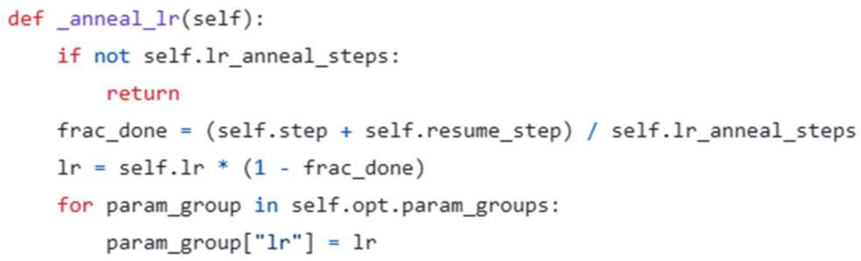
```
def _update_ema(self):
    for rate, params in zip(self.ema_rate, self.ema_params):
        update_ema(params, self.mp_trainer.master_params, rate=rate)
```

..nn.py



```
def update_ema(target_params, source_params, rate=0.99):
    """
    Update target parameters to be closer to those of source parameters using
    an exponential moving average.

    :param target_params: the target parameter sequence.
    :param source_params: the source parameter sequence.
    :param rate: the EMA rate (closer to 1 means slower).
    """
    for targ, src in zip(target_params, source_params):
        targ.detach().mul_(rate).add_(src, alpha=1 - rate)
```

(3) class TrainLoop : method runloop → runstep → _anneal_lr()



```
def _anneal_lr(self):
    if not self.lr_anneal_steps:
        return
    frac_done = (self.step + self.resume_step) / self.lr_anneal_steps
    lr = self.lr * (1 - frac_done)
    for param_group in self.opt.param_groups:
        param_group["lr"] = lr
```