

# An Introduction to Flow Matching and Diffusion Models

Peter Holderrieth and Ezra Erives

Website: <https://diffusion.csail.mit.edu/>

---

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Overview	3
1.2	Course Structure	4
1.3	Generative Modeling As Sampling	4
<b>2</b>	<b>Flow and Diffusion Models</b>	<b>7</b>
2.1	Flow Models	7
2.2	Diffusion Models	10
<b>3</b>	<b>Flow Matching</b>	<b>14</b>
3.1	Conditional and Marginal Probability Path	14
3.2	Conditional and Marginal Vector Fields	16
3.3	Learning the Marginal Vector Field	19
<b>4</b>	<b>Score Functions and Score Matching</b>	<b>25</b>
4.1	Conditional and Marginal Score Functions	25
4.2	Sampling with SDEs	27
4.3	Score Matching	31
<b>5</b>	<b>Guidance: How To Condition on a Prompt</b>	<b>34</b>
5.1	Vanilla Guidance	34
5.2	Classifier-Free Guidance	35
<b>6</b>	<b>Building Large-Scale Image or Video Generators</b>	<b>41</b>
6.1	Neural Network Architectures	41
6.2	Working in Latent Space: (Variational) Autoencoders	46
6.3	Case Study: Stable Diffusion 3 and Meta Movie Gen	52
<b>7</b>	<b>Discrete Diffusion Models: Building Language Models with Diffusion</b>	<b>54</b>
7.1	Continuous-Time Markov chain (CTMC) models	54
7.2	Training CTMC models	59
<b>8</b>	<b>References</b>	<b>66</b>
<b>A</b>	<b>A Reminder on Probability Theory</b>	<b>70</b>
A.1	Random vectors	70
A.2	Conditional densities and expectations	70

---

<b>B</b>	<b>A Proof of the Fokker-Planck equation</b>	<b>72</b>
<b>C</b>	<b>Existence and Uniqueness of Continuous-time Markov chains</b>	<b>74</b>
<b>D</b>	<b>Additional Perspectives on VAEs</b>	<b>77</b>
<b>E</b>	<b>A Guide to the Diffusion Model Literature</b>	<b>81</b>

---

---

# 1 Introduction

Creating noise from data is easy; creating data from noise is generative modeling.

---

Song et al. [43]

## 1.1 Overview

In recent years, we all have witnessed a tremendous revolution in artificial intelligence (AI). Image generators like *Nano Banana* or *Stable Diffusion 3* can generate photorealistic and artistic images across a diverse range of styles, video models like Meta’s *VEO-3* can generate highly realistic movie clips, and large language models like *ChatGPT* can generate seemingly human-level responses to text prompts. At the heart of this revolution lies a new ability of AI systems: the ability to **generate** objects. While previous generations of AI systems were mainly used for **prediction**, these new AI system are creative: they dream or come up with new objects based on user-specified input. Such **generative AI** systems are at the core of this recent AI revolution.

The goal of this class is to teach you two of the most widely used generative AI algorithms: **denoising diffusion models** [45] and **flow matching** [25, 27, 1, 26]. These models are the backbone of the best image, audio, and video generation models (e.g., *Nano Banana*, *FLUX*, or *VEO-3*), and have most recently become the state-of-the-art in scientific applications such as protein structures (e.g., *AlphaFold3* is a diffusion model). Without a doubt, understanding these models is truly an extremely useful skill to have.

All of these generative models generate objects by iteratively converting **noise** into **data**. This evolution from noise to data is facilitated by the simulation of **ordinary or stochastic differential equations (ODEs/SDEs)**. Flow matching and denoising diffusion models are a family of techniques that allow us to construct, train, and simulate, such ODEs/SDEs at large scale with deep neural networks. While these models are rather simple to implement, the technical nature of SDEs can make these models difficult to understand. In this course, we provide a self-contained introduction to the necessary mathematical toolbox regarding differential equations to enable you to systematically understand these models. We then explain step-by-step the modern stack of state-of-the-art image and video generators. Beyond being widely applicable, we believe that the theory behind flow and diffusion models is elegant in its own right. Therefore, most importantly, we hope that this course will be a lot of fun to you.

### Remark 1 (Additional Resources)

While these lecture notes are self-contained, there are two additional resources that we encourage you to use:

1. **Lecture recordings:** These guide you through each section in a lecture format.
2. **Labs:** These guide you in implementing your own diffusion model from scratch. We highly recommend that you “get your hands dirty” and code.

You can find these on our course website: <https://diffusion.csail.mit.edu/>.

## 1.2 Course Structure

We give a brief overview over of this document.

- **Section 1, Generative Modeling as Sampling:** We formalize what it means to “generate” an image, video, protein, etc. We will translate the problem of e.g., “how to generate an image of a dog?” into the more precise problem of sampling from a probability distribution.
- **Section 2, Flow and Diffusion Models:** We explain the machinery of generation. As you can guess by the name of this class, this machinery consists of simulating ordinary and stochastic differential equations. We provide an introduction to differential equations and explain how to use them to construct generative models.
- **Section 3, Flow Matching:** Next, we explain and derive *flow matching*, a simple and scalable algorithm lying at the core of all afore-mentioned large-scale generative models such as Stable Diffusion, Nano Banana, or SORA.
- **Section 4, Score Matching:** We study *score functions* and how they can be learnt via *score matching*. Not only is this the training algorithm for diffusion models, but it unlocks SDE sampling and guidance.
- **Section 5, Guidance:** We learn how to condition our samples on a prompt (e.g. “an image of a cat”) and how we can enforce adherence to such a prompt via classifier-free guidance.
- **Section 6, Latent Spaces, Neural Network architectures:** We discuss how one builds large-scale image and video generators such as *Nano Banana*. This includes common neural network architectures and how to build things in latent space. We also survey state-of-the-art models.
- **Section 7 (Optional), Discrete Diffusion Models:** We learn how to translate the principles of diffusion models from Euclidean space to discrete data such as language. This enables the construction of large language models using the principles of diffusion models.

**Required background.** Due to the technical nature of this subject, we recommend some base level of mathematical maturity, and in particular some familiarity with probability theory. For this reason, we included a brief reminder section on probability theory in [Section A](#). Don’t worry if some of the concepts there are unfamiliar to you.

## 1.3 Generative Modeling As Sampling

Let’s begin by thinking about various data types, or **data modalities**, that we might encounter, and how we will go about representing them numerically:

1. **Image:** Consider images with  $H \times W$  pixels where  $H$  describes the height and  $W$  the width of the image, each with three color channels (RGB). For every pixel and every color channel, we are given an intensity value in  $\mathbb{R}$ . Therefore, an image can be represented by an element  $z \in \mathbb{R}^{H \times W \times 3}$ .
2. **Video:** A video is simply a series of images in time. If we have  $T$  time points or **frames**, a video would therefore be represented by an element  $z \in \mathbb{R}^{T \times H \times W \times 3}$ .

3. **Molecular structure:** A naive way would be to represent the structure of a molecule by a matrix  $z = (z^1, \dots, z^N) \in \mathbb{R}^{3 \times N}$  where  $N$  is the number of atoms in the molecule and each  $z^i \in \mathbb{R}^3$  describes the location of that atom. Of course, there are other, more sophisticated ways of representing such a molecule.

In all of the above examples, the object that we want to generate can be mathematically represented as a vector (potentially after flattening). Therefore, throughout this document, we will have:

**Key Idea 1** (Objects as Vectors)

We identify the objects being generated as vectors  $z \in \mathbb{R}^d$ .

A notable exception to the above is text data, which is typically modeled as a discrete object by language models (such as *ChatGPT*). While continuous data  $z \in \mathbb{R}^d$  is our main focus, we also study text generation in [Section 7](#).

**Generation as Sampling.** Let us define what it means to “generate” something. For example, let’s say we want to generate an image of a dog. Naturally, there are *many* possible images of dogs that we would be happy with. In particular, there is no one single “best” image of a dog. Rather, there is a spectrum of images that fit better or worse. In machine learning, it is common to realize this diversity of possible images as a *probability distribution* over the *space of images*. We call such a distribution a **data distribution** and denote it as  $p_{\text{data}}$ . Mathematically, one can think of  $p_{\text{data}}$  as a **probability density**, i.e. a function  $p_{\text{data}} : \mathbb{R}^d \rightarrow \mathbb{R}_{\geq 0}$  that assigns each possible object  $z \in \mathbb{R}^d$  a *likelihood*  $p_{\text{data}}(z) \geq 0$ . In the example of dog images, this distribution would therefore give higher likelihood  $p_{\text{data}}(z)$  to images  $z$  that look more like a dog. Therefore, how “good” an image/video/molecule fits - a rather subjective statement - is replaced by how “likely” it is under the data distribution  $p_{\text{data}}$ . With this, we can mathematically express the task of generation as sampling from the (unknown) distribution  $p_{\text{data}}$ :

**Key Idea 2** (Generation as Sampling)

Generating an object  $z$  is modeled as sampling from the data distribution  $z \sim p_{\text{data}}$ .

A **generative model** is a machine learning model that allows us to generate samples from  $p_{\text{data}}$ . In machine learning, we require data to train models. In generative modeling, we usually assume access to a finite number of examples sampled independently from  $p_{\text{data}}$ , which together serve as a proxy for the true distribution.

**Key Idea 3** (Dataset)

A dataset consists of a finite number of samples  $z_1, \dots, z_N \sim p_{\text{data}}$ .

For images, we might construct a dataset by compiling publicly available images from the internet. For videos, we might similarly look to use YouTube. For protein structures, sources like the RCSB Protein Data Bank (PDB) provide hundreds of thousands of experimentally resolved structures. As the size of our dataset grows very large, it becomes an increasingly better representation of the underlying distribution  $p_{\text{data}}$ .

**Guided/Conditional Generation.** In many cases, we want to generate an object **conditioned** on some data  $y$ . For example, we might want to generate an image conditioned on  $y =$ “a dog running down a hill covered with snow with mountains in the background”. We can rephrase this as sampling from a **conditional distribution**:

**Key Idea 4** (Guided Generation)

Guided generation involves sampling from  $z \sim p_{\text{data}}(\cdot|y)$ , where  $y$  is a conditioning variable.

We call  $p_{\text{data}}(\cdot|y)$  the **guided data distribution**. The guided generative modeling task typically involves learning to condition on an arbitrary, rather than fixed, choice of  $y$ . Using our previous example, we might alternatively want to condition on a different text prompt, such as  $y$  = “a photorealistic image of a cat blowing out birthday candles”. We therefore seek a single model which may be conditioned on any such choice of  $y$ . It turns out that techniques for unconditional generation are readily generalized to the conditional case. Therefore, for the first 3 sections, we will focus almost exclusively on the unconditional case (keeping in mind that conditional generation is what we’re building towards).

**Generative Models.** Abstractly speaking, a generative model is an algorithm that returns samples from  $z \sim p_{\text{data}}$  (or at least approximately). If  $p_{\text{data}}$  is the distribution of images of dogs, this algorithm would return random images of dogs. In this course, we will focus on the specific construction of generative models using flow or diffusion models as these represent the current state-of-the-art. However, it is important to keep in mind that many other generative models were developed (and maybe even more that will be discovered in the future).

**Summary 2** (Generation as Sampling)

We summarize the findings of this section:

1. In this work, we mainly consider the task of generating objects that are represented as vectors  $z \in \mathbb{R}^d$  such as images, videos, and molecular structures.
2. Generation is the task of generating samples from a probability distribution  $p_{\text{data}}$  having access to a dataset of samples  $z_1, \dots, z_N \sim p_{\text{data}}$  during training.
3. Guided generation assumes that we condition the distribution on a label  $y$  and we want to sample from  $p_{\text{data}}(\cdot|y)$  having access to data set of pairs  $(z_1, y) \dots, (z_N, y)$  during training.
4. Our goal is to construct a generative model, i.e. a model that returns samples from  $p_{\text{data}}$  after training.

---

## 2 Flow and Diffusion Models

In the previous section, we formalized generative modeling as sampling from a data distribution  $p_{\text{data}}$ . Further, we formalized our goal: To construct a generative model, i.e. an algorithm that returns samples  $z \sim p_{\text{data}}$ . In this section, we describe how a generative model can be built as the simulation of a suitably constructed differential equation. For example, flow matching and diffusion models involve simulating **ordinary differential equations** (ODEs) and **stochastic differential equations** (SDEs), respectively. The goal of this section is therefore to define and construct these generative models as they will be used throughout the remainder of the notes. Specifically, we first define ODEs and SDEs, and discuss their simulation. Second, we describe how to parameterize an ODE/SDE using a deep neural network. This leads to the definition of a flow and diffusion model and the fundamental algorithms to sample from such models. In later sections, we then explore how to train these models.

### 2.1 Flow Models

We start by defining **ordinary differential equations (ODEs)**. A solution to an ODE is defined by a **trajectory**, i.e. a function of the form

$$X : [0, 1] \rightarrow \mathbb{R}^d, \quad t \mapsto X_t,$$

that maps from time  $t$  to some location in space  $\mathbb{R}^d$ . Every ODE is defined by a **vector field**  $u$ , i.e. a function of the form

$$u : \mathbb{R}^d \times [0, 1] \rightarrow \mathbb{R}^d, \quad (x, t) \mapsto u_t(x),$$

i.e. for every time  $t$  and location  $x$  we get a vector  $u_t(x) \in \mathbb{R}^d$  specifying a velocity in space (see [Figure 1](#)). An ODE imposes a condition on a trajectory: we want a trajectory  $X$  that “follows along the lines” of the vector field  $u_t$ , starting at the point  $x_0$ . We may formalize such a trajectory as being the solution to the equation:

$$\frac{d}{dt} X_t = u_t(X_t) \quad \blacktriangleright \text{ODE} \tag{1a}$$

$$X_0 = x_0 \quad \blacktriangleright \text{initial conditions} \tag{1b}$$

[Equation \(1a\)](#) requires that the derivative of  $X_t$  is specified by the direction given by  $u_t$ . [Equation \(1b\)](#) requires that we start at  $x_0$  at time  $t = 0$ . We may now ask: if we start at  $X_0 = x_0$  at  $t = 0$ , where are we at time  $t$  (what is  $X_t$ )? This question is answered by a function called the **flow**, which is a solution to the ODE

$$\psi : \mathbb{R}^d \times [0, 1] \rightarrow \mathbb{R}^d, \quad (x_0, t) \mapsto \psi_t(x_0) \tag{2a}$$

$$\frac{d}{dt} \psi_t(x_0) = u_t(\psi_t(x_0)) \quad \blacktriangleright \text{flow ODE} \tag{2b}$$

$$\psi_0(x_0) = x_0 \quad \blacktriangleright \text{flow initial conditions} \tag{2c}$$

For a given initial condition  $X_0 = x_0$ , a trajectory of the ODE is recovered via  $X_t = \psi_t(X_0)$ . Therefore, vector fields, ODEs, and flows are, intuitively, three descriptions of the same object: **vector fields define ODEs whose solutions are flows**. As with every equation, we should ask ourselves about an ODE: Does a solution exist and if

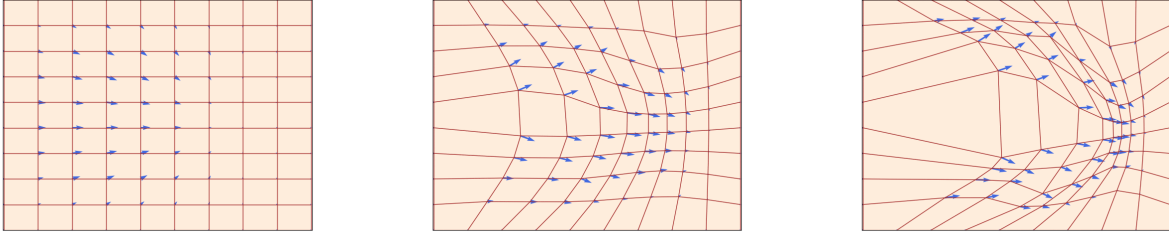


Figure 1: A flow  $\psi_t : \mathbb{R}^d \rightarrow \mathbb{R}^d$  (red square grid) is defined by a velocity field  $u_t : \mathbb{R}^d \rightarrow \mathbb{R}^d$  (visualized with blue arrows) that prescribes its instantaneous movements at all locations (here,  $d = 2$ ). We show three different times  $t$ . As one can see, a flow is a diffeomorphism that "warps" space. Figure from [26].

so, is it unique? A fundamental result in mathematics is "yes!" to both, as long as we impose weak assumptions on  $u_t$ :

**Theorem 3** (Flow existence and uniqueness)

If  $u : \mathbb{R}^d \times [0, 1] \rightarrow \mathbb{R}^d$  is continuously differentiable with a bounded derivative, then the ODE in (2) has a unique solution given by a flow  $\psi_t$ . In this case,  $\psi_t$  is a **diffeomorphism** for all  $t$ , i.e.  $\psi_t$  is continuously differentiable with a continuously differentiable inverse  $\psi_t^{-1}$ .

Note that the assumptions required for the existence and uniqueness of a flow are almost always fulfilled in machine learning, as we use neural networks to parameterize  $u_t(x)$  and they always have bounded derivatives. Therefore, **Theorem 3** should not be a concern for you but rather good news: **flows exist and are unique solutions to ODEs in our cases of interest**. A proof can be found in [32, 9].

**Example 4** (Linear Vector Fields)

Let us consider a simple example of a vector field  $u_t(x)$  that is a simple linear function in  $x$ , i.e.  $u_t(x) = -\theta x$  for  $\theta > 0$ . Then the function

$$\psi_t(x_0) = \exp(-\theta t) x_0 \tag{3}$$

defines a flow  $\psi$  solving the ODE in Equation (2). You can check this yourself by checking that  $\psi_0(x_0) = x_0$  and computing

$$\frac{d}{dt} \psi_t(x_0) \stackrel{(3)}{=} \frac{d}{dt} (\exp(-\theta t) x_0) \stackrel{(i)}{=} -\theta \exp(-\theta t) x_0 \stackrel{(3)}{=} -\theta \psi_t(x_0) = u_t(\psi_t(x_0)),$$

where in (i) we used the chain rule. In Figure 3, we visualize a flow of this form converging to 0 exponentially.

**Simulating an ODE.** In general, it is not possible to compute the flow  $\psi_t$  explicitly if  $u_t$  is not as simple as in the previous example. In these cases, one uses **numerical methods** to simulate ODEs. Fortunately, this is a classical and well researched topic in numerical analysis, and a myriad of powerful methods exist [21]. One of the simplest

and most intuitive methods is the **Euler method**. In the Euler method, we initialize with  $X_0 = x_0$  and update via

$$X_{t+h} = X_t + hu_t(X_t) \quad (t = 0, h, 2h, 3h, \dots, 1 - h) \quad (4)$$

where  $h = n^{-1} > 0$  is the **step size** and  $n \in \mathbb{N}$  is the number of simulation steps. For this class, the Euler method will be good enough. To give you a taste of a more complex method, let us consider **Heun's method** defined via the update rule

$$\begin{aligned} X'_{t+h} &= X_t + hu_t(X_t) && \blacktriangleright \text{initial guess of new state (same as Euler step)} \\ X_{t+h} &= X_t + \frac{h}{2}(u_t(X_t) + u_{t+h}(X'_{t+h})) && \blacktriangleright \text{update with average } u \text{ at current and guessed state} \end{aligned}$$

Intuitively, Heun's method is as follows: it takes a first guess  $X'_{t+h}$  of what the next step could be but corrects the direction initially taken via an updated guess.

**Flow models.** We can now construct a generative model via an ODE by making the vector field a **neural network vector field**  $u_t^\theta$ . For now, we simply mean that  $u_t^\theta$  is a parameterized function  $u_t^\theta : \mathbb{R}^d \times [0, 1] \rightarrow \mathbb{R}^d$  with parameters  $\theta$ . Later, we will discuss particular choices of neural network architectures. Remember that our goal was to generate samples  $z \sim p_{\text{data}}$  from a distribution  $p_{\text{data}}$ . In particular, these samples must be *random*. Note though that an ODE itself is not random but fully deterministic. To inject some randomness, we simply make the initial condition  $X_0$  random. Specifically, we choose an **initial distribution**  $p_{\text{init}}$ . In most cases, we set  $p_{\text{init}} = \mathcal{N}(0, I_d)$  to be a simple standard Gaussian. Most importantly, whatever distribution you choose, it must be one that we can easily sample from at inference-time. A **flow model** is then described by the ODE

$$\begin{aligned} X_0 &\sim p_{\text{init}} && \blacktriangleright \text{random initialization} \\ \frac{d}{dt}X_t &= u_t^\theta(X_t) && \blacktriangleright \text{ODE} \end{aligned}$$

Our goal is to make the endpoint  $X_1$  of the trajectory have distribution  $p_{\text{data}}$ , i.e.

$$X_1 \sim p_{\text{data}} \quad \Leftrightarrow \quad \psi_1^\theta(X_0) \sim p_{\text{data}}$$

where  $\psi_t^\theta$  describes the flow induced by  $u_t^\theta$ . Note however: although it is called *flow model*, **the neural network parameterizes the vector field, not the flow**. In order to compute the flow, we need to simulate the ODE. In [Algorithm 1](#), we summarize the procedure how to sample from a flow model.

---

**Algorithm 1** Sampling from a Flow Model with Euler method

---

**Require:** Neural network vector field  $u_t^\theta$ , number of steps  $n$

- 1: Set  $t = 0$
  - 2: Set step size  $h = \frac{1}{n}$
  - 3: Draw a sample  $X_0 \sim p_{\text{init}}$
  - 4: **for**  $i = 1, \dots, n$  **do**
  - 5:    $X_{t+h} = X_t + hu_t^\theta(X_t)$
  - 6:   Update  $t \leftarrow t + h$
  - 7: **end for**
  - 8: **return**  $X_1$
-

## 2.2 Diffusion Models

Stochastic differential equations (SDEs) extend the deterministic trajectories from ODEs with **stochastic** trajectories. A stochastic trajectory is commonly called a **stochastic process**  $(X_t)_{0 \leq t \leq 1}$  and is given by

$$X_t \text{ is a random variable for every } 0 \leq t \leq 1$$

$$X : [0, 1] \rightarrow \mathbb{R}^d, \quad t \mapsto X_t \text{ is a random trajectory for every draw of } X$$

In particular, when we simulate the same stochastic process twice, we might get different outcomes because the dynamics are designed to be random.

**Brownian Motion.** SDEs are constructed via a **Brownian motion** - a fundamental stochastic process that came out of the study physical diffusion processes. You can think of a Brownian motion as a continuous random walk. Let us define it: A **Brownian motion**  $W = (W_t)_{0 \leq t \leq 1}$  is a stochastic process such that  $W_0 = 0$ , the trajectories  $t \mapsto W_t$  are continuous, and the following two conditions hold:

1. **Normal increments:**  $W_t - W_s \sim \mathcal{N}(0, (t-s)I_d)$  for all  $0 \leq s < t$ , i.e. increments have a Gaussian distribution with variance increasing linearly in time ( $I_d$  is the identity matrix).
2. **Independent increments:** For any  $0 \leq t_0 < t_1 < \dots < t_n = 1$ , the increments  $W_{t_1} - W_{t_0}, \dots, W_{t_n} - W_{t_{n-1}}$  are independent random variables.

Brownian motion is also called a **Wiener process**, which is why we denote it with a "W".<sup>1</sup> We can easily simulate a Brownian motion approximately with step size  $h > 0$  by setting  $W_0 = 0$  and updating

$$W_{t+h} = W_t + \sqrt{h}\epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, I_d) \quad (t = 0, h, 2h, \dots, 1-h) \quad (5)$$

In [Figure 2](#), we plot a few example trajectories of a Brownian motion. Brownian motion is as central to the study of stochastic processes as the Gaussian distribution is to the study of probability distributions. From finance to statistical physics to epidemiology, the study of Brownian motion has far reaching applications beyond machine learning. In finance, for example, Brownian motion is used to model the price of complex financial instruments. Also just as a mathematical construction, Brownian motion is fascinating: For example, while the paths of a Brownian motion are continuous (so that you could draw it without ever lifting a pen), they are infinitely long (so that you would never stop drawing).

**From ODEs to SDEs.** The idea of an SDE is to extend the deterministic dynamics of an ODE by adding stochastic dynamics driven by a Brownian motion. Because everything is stochastic, we may no longer take the derivative as

<sup>1</sup>Norbert Wiener was a famous mathematician who taught at MIT. You can still see his portraits hanging at the MIT math department.

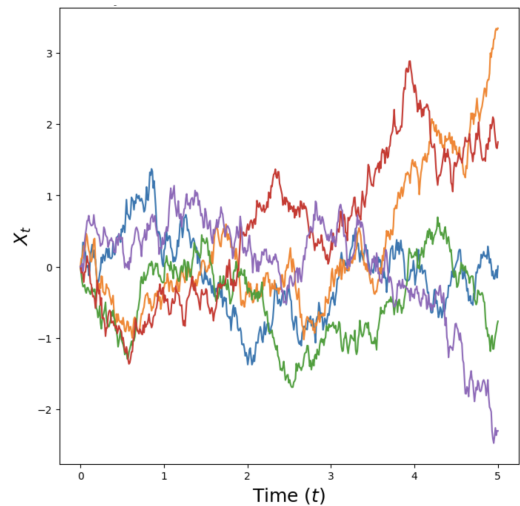


Figure 2: Sample trajectories of a Brownian motion  $W_t$  in dimension  $d = 1$  simulated using [Equation \(5\)](#).

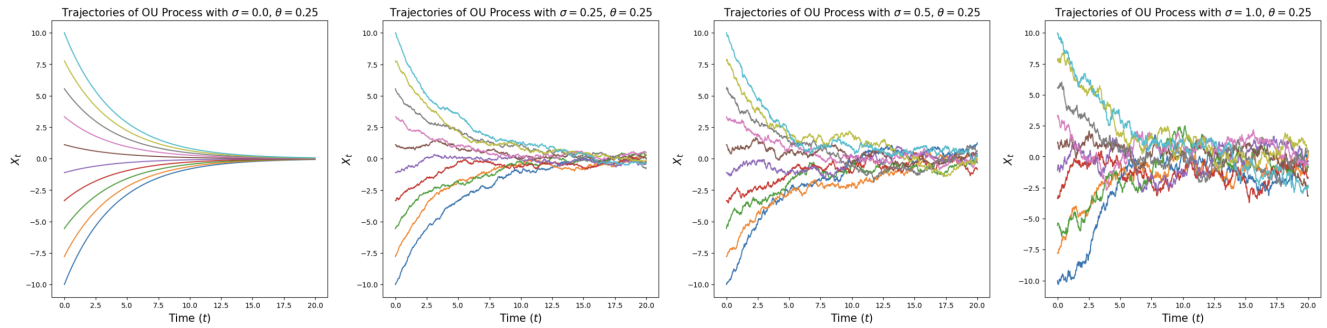


Figure 3: Illustration of Ornstein-Uhlenbeck processes (Equation (8)) in dimension  $d = 1$  for  $\theta = 0.25$  and various choices of  $\sigma$  (increasing from left to right). For  $\sigma = 0$ , we recover a flow (smooth, deterministic trajectories) that converges to the origin as  $t \rightarrow \infty$ . For  $\sigma > 0$  we have random paths which converge towards the Gaussian  $\mathcal{N}(0, \frac{\sigma^2}{2\theta})$  as  $t \rightarrow \infty$ .

in Equation (1a). Hence, we need to find an **equivalent formulation of ODEs that does not use derivatives**. For this, let us therefore rewrite trajectories  $(X_t)_{0 \leq t \leq 1}$  of an ODE as follows:

$$\begin{aligned} \frac{d}{dt} X_t &= u_t(X_t) && \blacktriangleright \text{expression via derivatives} \\ \stackrel{(i)}{\Leftrightarrow} \frac{1}{h} (X_{t+h} - X_t) &= u_t(X_t) + R_t(h) \\ \Leftrightarrow X_{t+h} &= X_t + hu_t(X_t) + hR_t(h) && \blacktriangleright \text{expression via infinitesimal updates} \end{aligned}$$

where  $R_t(h)$  describes a negligible function for small  $h$ , i.e. such that  $\lim_{h \rightarrow 0} R_t(h) = 0$ , and in (i) we simply use the definition of derivatives. The derivation above simply restates what we already know: A trajectory  $(X_t)_{0 \leq t \leq 1}$  of an ODE takes, at every timestep, a small step in the direction  $u_t(X_t)$ . We may now amend the last equation to make it stochastic: A trajectory  $(X_t)_{0 \leq t \leq 1}$  of an SDE takes, at every timestep, a small step in the direction  $u_t(X_t)$  *plus* some contribution from a Brownian motion:

$$X_{t+h} = X_t + \underbrace{hu_t(X_t)}_{\text{deterministic}} + \sigma_t \underbrace{(W_{t+h} - W_t)}_{\text{stochastic}} + \underbrace{hR_t(h)}_{\text{error term}} \tag{6}$$

where  $\sigma_t \geq 0$  describes the **diffusion coefficient** and  $R_t(h)$  describes a stochastic error term such that the standard deviation  $\mathbb{E}[|R_t(h)|^2]^{1/2} \rightarrow 0$  goes to zero for  $h \rightarrow 0$ . The above describes a **stochastic differential equation (SDE)**. It is common to denote it in the following symbolic notation:

$$dX_t = u_t(X_t)dt + \sigma_t dW_t \tag{7a} \blacktriangleright \text{SDE}$$

$$X_0 = x_0 \tag{7b} \blacktriangleright \text{initial condition}$$

However, always keep in mind that the " $dX_t$ "-notation above is a purely informal notation of Equation (6). Unfortunately, SDEs do not have a flow map  $\phi_t$  anymore. This is because the value  $X_t$  is not fully determined by  $X_0 \sim p_{\text{init}}$  anymore as the evolution itself is stochastic. Still, in the same way as for ODEs, we have:

**Theorem 5** (SDE Solution Existence and Uniqueness)

If  $u : \mathbb{R}^d \times [0, 1] \rightarrow \mathbb{R}^d$  is continuously differentiable with a bounded derivative and  $\sigma_t$  is continuous, then the SDE in (7) has a solution given by the unique stochastic process  $(X_t)_{0 \leq t \leq 1}$  satisfying Equation (6).

If this was a stochastic calculus class, we would spend several lectures proving this theorem and constructing SDEs with full mathematical rigor, i.e. constructing a Brownian motion from first principles and constructing the process  $X_t$  via **stochastic integration**. As we focus on machine learning in this class, we refer to [29] for a more technical treatment. Finally, note that every ODE is also an SDE - simply with a vanishing diffusion coefficient  $\sigma_t = 0$ . Therefore, for the remainder of this class, **when we speak about SDEs, we consider ODEs as a special case.**

**Example 6** (Ornstein-Uhlenbeck Process)

Let us consider a constant diffusion coefficient  $\sigma_t = \sigma \geq 0$  and a constant linear drift  $u_t(x) = -\theta x$  for  $\theta > 0$ , yielding the SDE

$$dX_t = -\theta X_t dt + \sigma dW_t. \quad (8)$$

A solution  $(X_t)_{0 \leq t \leq 1}$  to the above SDE is known as an **Ornstein-Uhlenbeck (OU) process**. We visualize it in Figure 3. The vector field  $-\theta x$  pushes the process back to its center 0 (since the drift always points in the direction opposite to the current position), while the diffusion coefficient  $\sigma$  always adds more noise. This process converges towards a Gaussian distribution  $\mathcal{N}(0, \sigma^2/(2\theta))$  if we simulate it for  $t \rightarrow \infty$ . Note that for  $\sigma = 0$ , we have a flow with linear vector field that we have studied in Equation (3).

**Simulating an SDE.** If you struggle with the abstract definition of an SDE so far, then don't worry about it. A more intuitive way of thinking about SDEs is given by answering the question: How might we simulate an SDE? The simplest such scheme is known as the **Euler-Maruyama method**, and is essentially to SDEs what the Euler method is to ODEs. Using the Euler-Maruyama method, we initialize  $X_0 = x_0$  and update iteratively via

$$X_{t+h} = X_t + hu_t(X_t) + \sqrt{h}\sigma_t\epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, I_d) \quad (9)$$

where  $h = n^{-1} > 0$  is a step size hyperparameter for  $n \in \mathbb{N}$ . In other words, to simulate using the Euler-Maruyama method, we take a small step in the direction of  $u_t(X_t)$  as well as add a little bit of Gaussian noise scaled by  $\sqrt{h}\sigma_t$ . When simulating SDEs in this class (such as in the accompanying labs), we will usually stick to the Euler-Maruyama method.

**Diffusion Models.** We can now construct a generative model via an SDE in the same way as we did for ODEs. Remember that our goal was to convert a simple distribution  $p_{\text{init}}$  into a complex distribution  $p_{\text{data}}$ . Like for ODEs, the simulation of an SDE randomly initialized with  $X_0 \sim p_{\text{init}}$  is a natural choice for this transformation. To parameterize this SDE, we can simply parameterize its central ingredient - the vector field  $u_t$  - via a neural

**Algorithm 2** Sampling from a Diffusion Model (Euler-Maruyama method)**Require:** Neural network  $u_t^\theta$ , number of steps  $n$ , diffusion coefficient  $\sigma_t$ 

- 1: Set  $t = 0$
- 2: Set step size  $h = \frac{1}{n}$
- 3: Draw a sample  $X_0 \sim p_{\text{init}}$
- 4: **for**  $i = 1, \dots, n$  **do**
- 5:   Draw a sample  $\epsilon \sim \mathcal{N}(0, I_d)$
- 6:    $X_{t+h} = X_t + hu_t^\theta(X_t) + \sigma_t\sqrt{h}\epsilon$
- 7:   Update  $t \leftarrow t + h$
- 8: **end for**
- 9: **return**  $X_1$

network  $u_t^\theta$ . A **diffusion model** is thus given by

$$\begin{aligned}
 X_0 &\sim p_{\text{init}} && \blacktriangleright \text{random initialization} \\
 dX_t &= u_t^\theta(X_t)dt + \sigma_t dW_t && \blacktriangleright \text{SDE}
 \end{aligned}$$

In [Algorithm 2](#), we describe the procedure by which to sample from a diffusion model with the Euler-Maruyama method. We summarize the results of this section as follows.

**Summary 7** (SDE generative model)

Throughout this document, a **diffusion model** consists of a neural network  $u_t^\theta$  with parameters  $\theta$  that parameterize a vector field and a fixed diffusion coefficient  $\sigma_t$ :

$$\begin{aligned}
 \textbf{Neural network: } &u^\theta : \mathbb{R}^d \times [0, 1] \rightarrow \mathbb{R}^d, (x, t) \mapsto u_t^\theta(x) \text{ with parameters } \theta \\
 \textbf{Fixed: } &\sigma_t : [0, 1] \rightarrow [0, \infty), t \mapsto \sigma_t
 \end{aligned}$$

To obtain samples from our SDE model (i.e. generate objects), the procedure is as follows:

- |                        |  |   |
|------------------------|--|---|
| <b>Initialization:</b> | $X_0 \sim p_{\text{init}}$                 | $\blacktriangleright$ Initialize with simple distribution, e.g. a Gaussian      |
| <b>Simulation:</b>     | $dX_t = u_t^\theta(X_t)dt + \sigma_t dW_t$ | $\blacktriangleright$ Simulate SDE from 0 to 1                                  |
| <b>Goal:</b>           | $X_1 \sim p_{\text{data}}$                 | $\blacktriangleright$ Goal is to make $X_1$ have distribution $p_{\text{data}}$ |

A diffusion model with  $\sigma_t = 0$  is a **flow model**.

### 3 Flow Matching

In the previous section, we constructed flow and diffusion models as generative models parameterized by a neural network vector field  $u_t^\theta$ . However, we have not yet discussed how to *train* them, i.e. how to optimize the parameters  $\theta$  such that generative model returns something sensible, e.g. a nice-looking image or exciting video. Next, we discuss **flow matching** [25, 1, 27], a algorithm to train  $u_t^\theta$  that is simple, scalable, and represents the current state-of-the-art.

In this section, we restrict ourselves to flow models, i.e. we have a neural network  $u_t^\theta$  and obtain samples from the generative model by simulating the ODE

$$X_0 \sim p_{\text{init}}, \quad dX_t = u_t^\theta(X_t)dt \quad (\text{Flow model}) \quad (10)$$

and using the endpoints  $X_1$  fro  $t = 1$  as samples. As we discussed, our goal is that  $X_1$  is distributed according to the data distribution  $p_{\text{data}}$ , i.e.  $X_1 \sim p_{\text{data}}$ . Therefore, the question “how to train” the neural network is really the following question: **How do we optimize  $\theta$  such that simulating the flow model in Equation (10) results in samples from the data distribution  $X_1 \sim p_{\text{data}}$ ?**

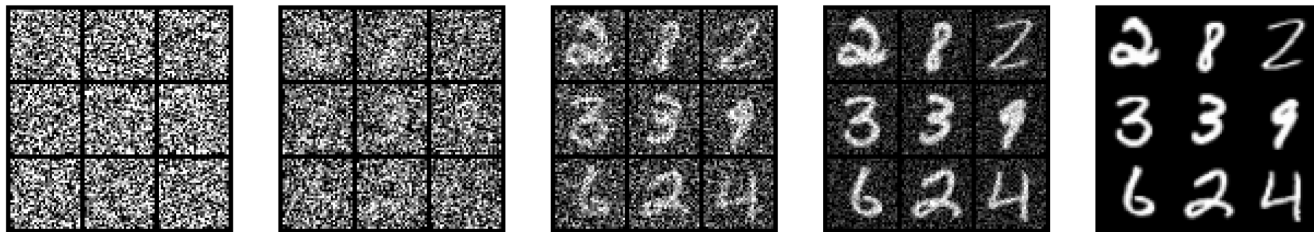


Figure 4: Gradual interpolation from noise to data via a Gaussian conditional probability path for a collection of images. Note that each image is a data point of dimension  $d = 32 \times 32$ , so we are plotting individual samples from the probability path, while in Figure 5 we plot the distribution as a 2d histogram.

#### 3.1 Conditional and Marginal Probability Path

The first step of flow matching is to specify a **probability path**. Intuitively, a probability path specifies a gradual interpolation between noise  $p_{\text{init}}$  and data  $p_{\text{data}}$  (see Figure 4). But why would we want that? Remember that our desired ODE trajectory fulfills  $X_0 \sim p_{\text{init}}$  for  $t = 0$  and  $X_1 \sim p_{\text{data}}$  for  $t = 1$ . But what about times  $0 < t < 1$  in between start and end? It turns out that we have some freedom to choose what should happen in between and this is what is mathematically formalized in a probability path.

In the following, for a data point  $z \in \mathbb{R}^d$ , we denote with  $\delta_z$  the **Dirac delta** “distribution”. This is the simplest distribution that one can imagine: sampling from  $\delta_z$  always returns  $z$  (i.e. it is deterministic). A **conditional (interpolating) probability path** is a set of distribution  $p_t(x|z)$  over  $\mathbb{R}^d$  such that:

$$p_0(\cdot|z) = p_{\text{init}}, \quad p_1(\cdot|z) = \delta_z \quad \text{for all } z \in \mathbb{R}^d. \quad (11)$$

In other words, a conditional probability path gradually converts the initial distribution  $p_{\text{init}}$  into a *single* data point (see e.g. Figure 4). You can think of a probability path as a trajectory in the space of distributions.

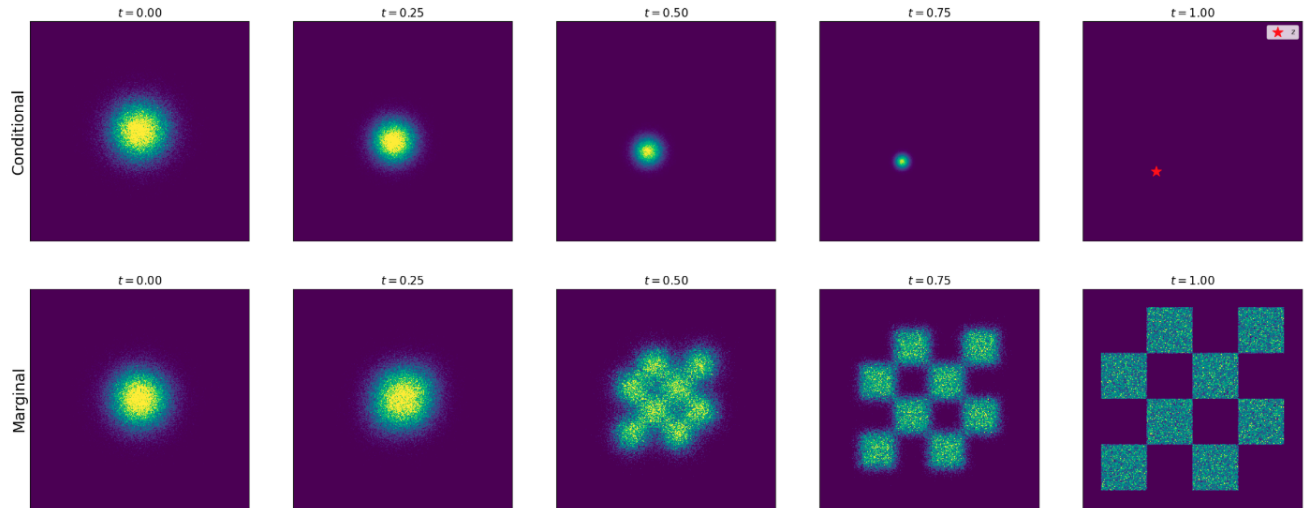


Figure 5: Illustration of a conditional (top) and marginal (bottom) probability path. Here, we plot a Gaussian probability path with  $\alpha_t = t, \beta_t = 1 - t$ . The conditional probability path interpolates a Gaussian  $p_{\text{init}} = \mathcal{N}(0, I_d)$  and  $p_{\text{data}} = \delta_z$  for single data point  $z$ . The marginal probability path interpolates a Gaussian and a data distribution  $p_{\text{data}}$  (Here,  $p_{\text{data}}$  is a toy distribution in dimension  $d = 2$  represented by a chess board pattern.)

Every conditional probability path  $p_t(x|z)$  induces a **marginal probability path**  $p_t(x)$  defined as the distribution that we obtain by first sampling a data point  $z \sim p_{\text{data}}$  from the data distribution and then sampling from  $p_t(\cdot|z)$ :

$$z \sim p_{\text{data}}, \quad x \sim p_t(\cdot|z) \quad \Rightarrow \quad x \sim p_t \quad \blacktriangleright \quad \text{sampling from marginal path} \quad (12)$$

$$p_t(x) = \int p_t(x|z)p_{\text{data}}(z)dz \quad \blacktriangleright \quad \text{density of marginal path} \quad (13)$$

Note that we know how to sample from  $p_t$  but we don't know the density values  $p_t(x)$  as the integral is intractable (i.e. we can actually compute Equation (12) but not Equation (13)). Check for yourself that because of the conditions on  $p_t(\cdot|z)$  in Equation (11), the marginal probability path  $p_t$  interpolates between  $p_{\text{init}}$  and  $p_{\text{data}}$ :

$$p_0 = p_{\text{init}} \quad \text{and} \quad p_1 = p_{\text{data}}. \quad \blacktriangleright \quad \text{noise-data interpolation} \quad (14)$$

The - by far - most important example of a probability path is the Gaussian probability path - hence, we strongly recommend reading the next example thoroughly.

#### Example 8 (Gaussian Conditional Probability Path)

One particularly popular probability path is the **Gaussian probability path**. This is the **probability path used by most state-of-the-art models**. Let  $\alpha_t, \beta_t$  be **noise schedulers**: two continuously differentiable, monotonic functions with  $\alpha_0 = \beta_1 = 0$  and  $\alpha_1 = \beta_0 = 1$ . We then define the conditional probability path

$$p_t(\cdot|z) = \mathcal{N}(\alpha_t z, \beta_t^2 I_d) \quad \blacktriangleright \quad \text{Gaussian conditional path} \quad (15)$$

which, by the conditions we imposed on  $\alpha_t$  and  $\beta_t$ , fulfills

$$p_0(\cdot|z) = \mathcal{N}(\alpha_0 z, \beta_0^2 I_d) = \mathcal{N}(0, I_d), \quad \text{and} \quad p_1(\cdot|z) = \mathcal{N}(\alpha_1 z, \beta_1^2 I_d) = \delta_z,$$

where we have used the fact that a normal distribution with zero variance and mean  $z$  is just  $\delta_z$ . Therefore, this choice of  $p_t(x|z)$  fulfills Equation (11) for  $p_{\text{init}} = \mathcal{N}(0, I_d)$  and is therefore a valid conditional interpolating path. In Figure 4, we illustrate its application to an image. We can express sampling from the marginal path  $p_t$  as:

$$z \sim p_{\text{data}}, \quad \epsilon \sim p_{\text{init}} = \mathcal{N}(0, I_d) \Rightarrow x = \alpha_t z + \beta_t \epsilon \sim p_t \quad \blacktriangleright \text{ sampling from marginal Gaussian path} \quad (16)$$

Intuitively, the above procedure adds more noise for lower  $t$  until time  $t = 0$ , at which point there is only noise. In Figure 5, we plot an example of such an interpolating path.

### 3.2 Conditional and Marginal Vector Fields

A probability path  $(p_t)_{0 \leq t \leq 1}$  specifies what distributions  $X_t \sim p_t$  the points  $X_t$  along a trajectory *should* have. At this point, this is just what we “wish” to be the case. But how can we find a vector field such that the trajectories  $X_t$  follow the probability path? Flow matching explicitly constructs such a vector field - the “marginal vector field” - which we explain in this section.

For every data point  $z \in \mathbb{R}^d$ , let  $u_t^{\text{target}}(\cdot|z)$  denote a **conditional vector field**. This can be any vector field such that corresponding ODE yields the conditional probability path  $p_t(\cdot|z)$ , i.e. such that it holds

$$X_0 \sim p_{\text{init}}, \quad \frac{d}{dt} X_t = u_t^{\text{target}}(X_t|z) \Rightarrow X_t \sim p_t(\cdot|z) \quad (0 \leq t \leq 1). \quad (17)$$

We can often find a conditional vector field  $u_t^{\text{target}}(\cdot|z)$  analytically by hand (i.e. by just doing some algebra ourselves). We illustrate this by deriving a conditional vector field  $u_t(x|z)$  for our running example of a Gaussian probability path in Example 10.

At first sight, a conditional vector field seems useless because all endpoints of the ODE  $X_1$  will collapse to  $X_1 = z$ , i.e. we are just re-generating known data points  $z$ . However, the conditional vector field serves as a building block for a vector field that generates actual samples from  $p_{\text{data}}$ :

#### Theorem 9 (Marginalization trick)

Let  $u_t^{\text{target}}(x|z)$  be a conditional vector field (Equation (17)). Then the **marginal vector field**  $u_t^{\text{target}}(x)$  defined as

$$u_t^{\text{target}}(x) = \int u_t^{\text{target}}(x|z) \frac{p_t(x|z) p_{\text{data}}(z)}{p_t(x)} dz, \quad (18)$$

follows the marginal probability path, i.e.

$$X_0 \sim p_{\text{init}}, \quad \frac{d}{dt} X_t = u_t^{\text{target}}(X_t) \Rightarrow X_t \sim p_t \quad (0 \leq t \leq 1). \quad (19)$$

**In particular,  $X_1 \sim p_{\text{data}}$  for this ODE, so that we might say " $u_t^{\text{target}}$  converts noise  $p_{\text{init}}$  into data  $p_{\text{data}}$ ".**

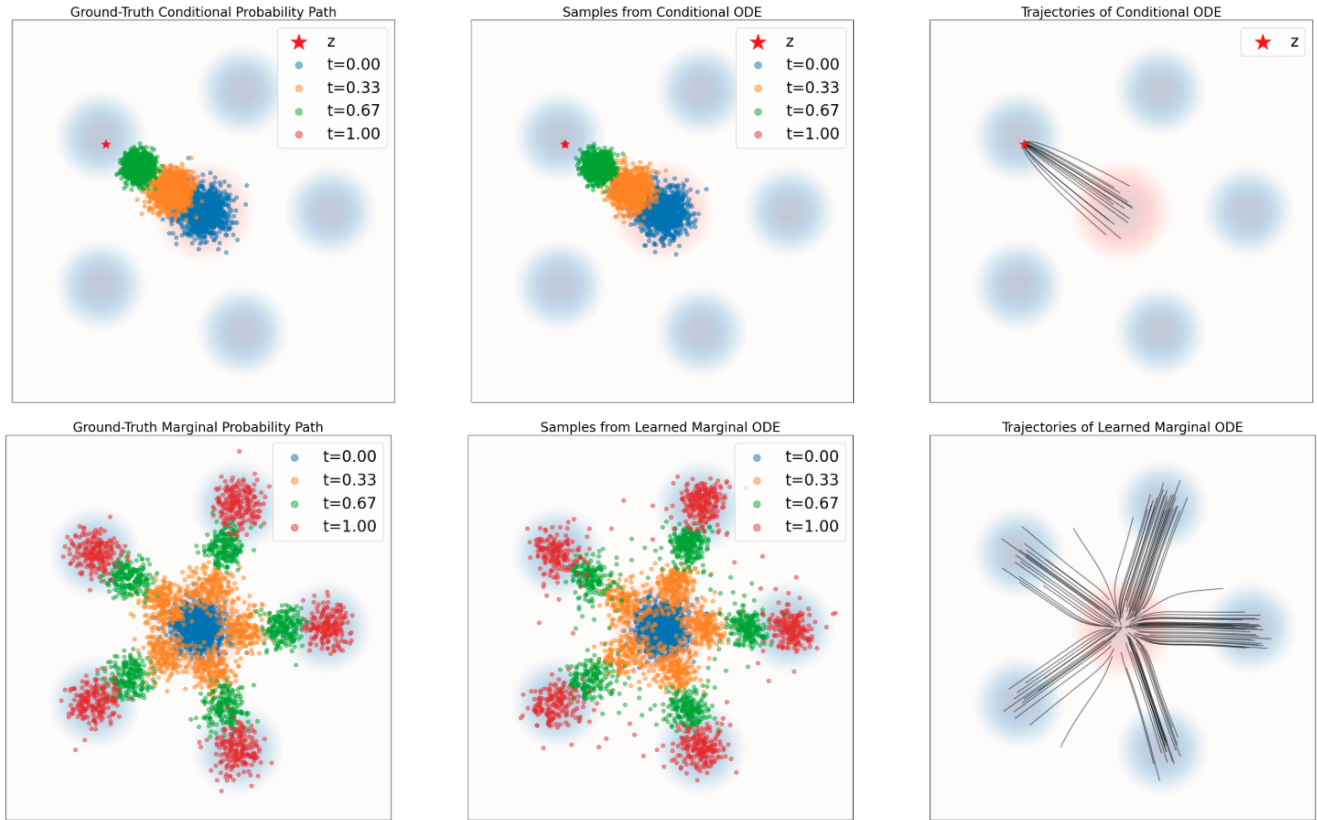


Figure 6: Illustration of [Theorem 9](#). Simulating a probability path with ODEs. Data distribution  $p_{\text{data}}$  in blue background. Gaussian  $p_{\text{init}}$  in red background. Top row: Conditional probability path. Left: Ground truth samples from conditional path  $p_t(\cdot|z)$ . Middle: ODE samples over time. Right: Trajectories by simulating ODE with  $u_t^{\text{target}}(x|z)$  in [Equation \(20\)](#). Bottom row: Simulating a marginal probability path. Left: Ground truth samples from  $p_t$ . Middle: ODE samples over time. Right: Trajectories by simulating ODE with marginal vector field  $u_t^{\text{flow}}(x)$ . As one can see, the conditional vector field follows the conditional probability path and the marginal vector field follows the marginal probability path.

**Example 10** (Target ODE for Gaussian probability paths)

As before, let  $p_t(\cdot|z) = \mathcal{N}(\alpha_t z, \beta_t^2 I_d)$  for noise schedulers  $\alpha_t, \beta_t$  (see [Equation \(15\)](#)). Let  $\dot{\alpha}_t = \partial_t \alpha_t$  and  $\dot{\beta}_t = \partial_t \beta_t$  denote respective time derivatives of  $\alpha_t$  and  $\beta_t$ . Here, we want to show that the **conditional Gaussian vector field** given by

$$u_t^{\text{target}}(x|z) = \left( \dot{\alpha}_t - \frac{\dot{\beta}_t}{\beta_t} \alpha_t \right) z + \frac{\dot{\beta}_t}{\beta_t} x \quad (20)$$

is a valid conditional vector field model in the sense of [Theorem 9](#): its ODE trajectories  $X_t$  satisfy  $X_t \sim p_t(\cdot|z) = \mathcal{N}(\alpha_t z, \beta_t^2 I_d)$  if  $X_0 \sim \mathcal{N}(0, I_d)$ . In [Figure 6](#), we confirm this visually by comparing samples from the conditional probability path (ground truth) to samples from simulated ODE trajectories of this flow. As you can see, the distribution match. We will now prove this.

*Proof.* Let us construct a conditional flow model  $\psi_t^{\text{target}}(x|z)$  first by defining

$$\psi_t^{\text{target}}(x|z) = \alpha_t z + \beta_t x. \quad (21)$$

If  $X_t$  is the ODE trajectory of  $\psi_t^{\text{target}}(\cdot|z)$  with  $X_0 \sim p_{\text{init}} = \mathcal{N}(0, I_d)$ , then by definition

$$X_t = \psi_t^{\text{target}}(X_0|z) = \alpha_t z + \beta_t X_0 \sim \mathcal{N}(\alpha_t z, \beta_t^2 I_d) = p_t(\cdot|z).$$

We conclude that the trajectories are distributed like the conditional probability path (i.e., Equation (17) is fulfilled). It remains to extract the vector field  $u_t^{\text{target}}(x|z)$  from  $\psi_t^{\text{target}}(x|z)$ . By the definition of a flow (Equation (2b)), it holds

$$\begin{aligned} \frac{d}{dt} \psi_t^{\text{target}}(x|z) &= u_t^{\text{target}}(\psi_t^{\text{target}}(x|z)|z) \quad \text{for all } x, z \in \mathbb{R}^d \\ \stackrel{(i)}{\Leftrightarrow} \quad \dot{\alpha}_t z + \dot{\beta}_t x &= u_t^{\text{target}}(\alpha_t z + \beta_t x|z) \quad \text{for all } x, z \in \mathbb{R}^d \\ \stackrel{(ii)}{\Leftrightarrow} \quad \dot{\alpha}_t z + \dot{\beta}_t \left( \frac{x - \alpha_t z}{\beta_t} \right) &= u_t^{\text{target}}(x|z) \quad \text{for all } x, z \in \mathbb{R}^d \\ \stackrel{(iii)}{\Leftrightarrow} \quad \left( \dot{\alpha}_t - \frac{\dot{\beta}_t}{\beta_t} \alpha_t \right) z + \frac{\dot{\beta}_t}{\beta_t} x &= u_t^{\text{target}}(x|z) \quad \text{for all } x, z \in \mathbb{R}^d \end{aligned}$$

where in (i) we used the definition of  $\psi_t^{\text{target}}(x|z)$  (Equation (21)), in (ii) we reparameterized  $x \rightarrow (x - \alpha_t z)/\beta_t$ , and in (iii) we just did some algebra. Note that the last equation is the conditional Gaussian vector field as we defined in Equation (20). This proves the statement.<sup>a</sup>  $\square$

<sup>a</sup>One can also double check this by plugging it into the continuity equation introduced later in this section.

See Figure 6 for an illustration of Theorem 9. Let's gain some intuition for the marginal vector field. **Bayes' rule** from statistics says that the following term describes a posterior distribution

$$\frac{p_t(x|z)p_{\text{data}}(z)}{p_t(x)} = \text{"posterior over data points } z \text{ given noisy data } x\text{"}$$

where  $p_{\text{data}}(z)$  is the prior distribution. The marginal vector field then is simply an average: for every *possible* data point  $z$  it takes the velocity  $u_t(x|z)$  - i.e. the direction that would bring us to  $z$  - and then weighs this velocity by how much we believe that  $x$  comes from  $z$ . Averaging over all data points, we obtain the marginal vector field.

The remainder of this section will make this intuition rigorous and prove Theorem 9. As the main mathematical tool, we will use the **continuity equation**, a fundamental equation in mathematics and physics. Define the **divergence** operator  $\text{div}$  as

$$\text{div}(v_t)(x) = \sum_{i=1}^d \frac{\partial}{\partial x_i} v_t^i(x) \quad (22)$$

where  $v_t^i$  is the  $i$ -th coordinate of  $v_t$ .

**Theorem 11** (Continuity Equation)

Let us consider an flow model with vector field  $u_t^{\text{target}}$  with  $X_0 \sim p_{\text{init}} = p_0$ . Then  $X_t \sim p_t$  for all  $0 \leq t \leq 1$  if and only if

$$\partial_t p_t(x) = -\text{div}(p_t u_t^{\text{target}})(x) \quad \text{for all } x \in \mathbb{R}^d, 0 \leq t \leq 1, \quad (23)$$

where  $\partial_t p_t(x) = \frac{d}{dt} p_t(x)$  denotes the time-derivative of  $p_t(x)$ . Equation 23 is known as the **continuity equation**.

For the mathematically-inclined reader, we present a self-contained proof of the Continuity Equation in Section B. Before we move on, let us try and understand intuitively the continuity equation. The left-hand side  $\partial_t p_t(x)$  describes how much the probability  $p_t(x)$  at  $x$  changes over time. Intuitively, the change should correspond to the net inflow of probability mass. For a flow model, a particle  $X_t$  follows along the vector field  $u_t^{\text{target}}$ . As you might recall from physics, the divergence measures a sort of net outflow from the vector field. Therefore, the negative divergence measures the net inflow. Scaling this by the total probability mass currently residing at  $x$ , we get that the net  $-\text{div}(p_t u_t)$  measures the total inflow of probability mass. Since probability mass is conserved (always integrates to 1), the left-hand and right-hand side of the equation should be the same! We now proceed with a proof of the marginalization trick from Theorem 9.

*Proof of Theorem 9.* By Theorem 11, we have to show that the marginal vector field  $u_t^{\text{target}}$ , as defined as in Equation (18), satisfies the continuity equation. We can do this by direct calculation:

$$\begin{aligned} \partial_t p_t(x) &\stackrel{(i)}{=} \partial_t \int p_t(x|z) p_{\text{data}}(z) dz = \int \partial_t p_t(x|z) p_{\text{data}}(z) dz \\ &\stackrel{(ii)}{=} \int -\text{div}(p_t(\cdot|z) u_t^{\text{target}}(\cdot|z))(x) p_{\text{data}}(z) dz \\ &\stackrel{(iii)}{=} -\text{div} \left( \int p_t(x|z) u_t^{\text{target}}(x|z) p_{\text{data}}(z) dz \right) \\ &\stackrel{(iv)}{=} -\text{div} \left( p_t(x) \int u_t^{\text{target}}(x|z) \frac{p_t(x|z) p_{\text{data}}(z)}{p_t(x)} dz \right) (x) \\ &\stackrel{(v)}{=} -\text{div} (p_t u_t^{\text{target}}) (x), \end{aligned}$$

where in (i) we used the definition of  $p_t(x)$  in Equation (12), in (ii) we used the continuity equation for the conditional probability path  $p_t(\cdot|z)$ , in (iii) we swapped the integral and divergence operator using Equation (22), in (iv) we multiplied and divided by  $p_t(x)$ , and in (v) we used Equation (18). The beginning and end of the above chain of equations show that the continuity equation is fulfilled for  $u_t^{\text{target}}$ . By Theorem 11, this is enough to imply Equation (19), and we are done.  $\square$

### 3.3 Learning the Marginal Vector Field

Now, we are ready to describe the training algorithm. The goal of flow matching is to train the neural network  $u_t^\theta$  such that it equals the marginal vector field  $u_t^{\text{target}}$ . If this holds, we know that the endpoints  $X_1 \sim p_{\text{data}}$  have the desired distribution by Theorem 9. In the following, we denote by  $\text{Unif} = \text{Unif}_{[0,1]}$  the uniform distribution on the interval  $[0, 1]$ , and by  $\mathbb{E}$  the expected value of a random variable. An intuitive way of obtaining  $u_t^\theta \approx u_t^{\text{target}}$  is to

use a mean-squared error, i.e. to use the **flow matching loss** defined as

$$\mathcal{L}_{\text{FM}}(\theta) = \mathbb{E}_{t \sim \text{Unif}, x \sim p_t} [\|u_t^\theta(x) - u_t^{\text{target}}(x)\|^2] \quad (24)$$

$$\stackrel{(i)}{=} \mathbb{E}_{t \sim \text{Unif}, z \sim p_{\text{data}}, x \sim p_t(\cdot|z)} [\|u_t^\theta(x) - u_t^{\text{target}}(x)\|^2], \quad (25)$$

where  $p_t(x) = \int p_t(x|z)p_{\text{data}}(z)dz$  is the marginal probability path and in (i) we used the sampling procedure given by Equation (12). Intuitively, this loss says: First, draw a random time  $t \in [0, 1]$ . Second, draw a random point  $z$  from our data set, sample from  $p_t(\cdot|z)$  (e.g., by adding some noise), and compute  $u_t^\theta(x)$ . Finally, compute the mean-squared error between the output of our neural network and the marginal vector field  $u_t^{\text{target}}(x)$ . Unfortunately, we are *not* done here. While we do know the formula for  $u_t^{\text{target}}$  by Theorem 9, we cannot compute it efficiently as the integral is intractable. Instead, we will exploit the fact that the **conditional** velocity field  $u_t^{\text{target}}(x|z)$  is tractable. To do so, let us define the **conditional flow matching loss**

$$\mathcal{L}_{\text{CFM}}(\theta) = \mathbb{E}_{t \sim \text{Unif}, z \sim p_{\text{data}}, x \sim p_t(\cdot|z)} [\|u_t^\theta(x) - u_t^{\text{target}}(x|z)\|^2]. \quad (26)$$

Note the difference to Equation (24): we use the conditional vector field  $u_t^{\text{target}}(x|z)$  instead of the marginal vector  $u_t^{\text{target}}(x)$ . As we have an analytical formula for  $u_t^{\text{target}}(x|z)$ , we can minimize the above loss easily. But wait, what sense does it make to regress against the conditional vector field if it's the marginal vector field we care about? As it turns out, by *explicitly* regressing against the tractable, conditional vector field, we are *implicitly* regressing against the intractable, marginal vector field. The next result makes this intuition precise.

### Theorem 12

The marginal flow matching loss equals the conditional flow matching loss up to a constant. That is,

$$\mathcal{L}_{\text{FM}}(\theta) = \mathcal{L}_{\text{CFM}}(\theta) + C,$$

where  $C$  is independent of  $\theta$ . Therefore, their gradients coincide:

$$\nabla_\theta \mathcal{L}_{\text{FM}}(\theta) = \nabla_\theta \mathcal{L}_{\text{CFM}}(\theta).$$

Hence, minimizing  $\mathcal{L}_{\text{CFM}}(\theta)$  with e.g., stochastic gradient descent (SGD) is equivalent to minimizing  $\mathcal{L}_{\text{FM}}(\theta)$  in the same fashion. In particular, **for the minimizer  $\theta^*$  of  $\mathcal{L}_{\text{CFM}}(\theta)$ , it will hold that  $u_t^{\theta^*} = u_t^{\text{target}}$ , i.e. the neural network will equal the marginal vector field** (assuming an infinitely expressive parameterization).

*Direct Proof.* The proof works by expanding the mean-squared error into three components and removing constants:

$$\begin{aligned} \mathcal{L}_{\text{FM}}(\theta) &\stackrel{(i)}{=} \mathbb{E}_{t \sim \text{Unif}, x \sim p_t} [\|u_t^\theta(x) - u_t^{\text{target}}(x)\|^2] \\ &\stackrel{(ii)}{=} \mathbb{E}_{t \sim \text{Unif}, x \sim p_t} [\|u_t^\theta(x)\|^2 - 2u_t^\theta(x)^T u_t^{\text{target}}(x) + \|u_t^{\text{target}}(x)\|^2] \\ &\stackrel{(iii)}{=} \mathbb{E}_{t \sim \text{Unif}, x \sim p_t} [\|u_t^\theta(x)\|^2] - 2\mathbb{E}_{t \sim \text{Unif}, x \sim p_t} [u_t^\theta(x)^T u_t^{\text{target}}(x)] + \underbrace{\mathbb{E}_{t \sim \text{Unif}[0,1], x \sim p_t} [\|u_t^{\text{target}}(x)\|^2]}_{=: C_1} \\ &\stackrel{(iv)}{=} \mathbb{E}_{t \sim \text{Unif}, z \sim p_{\text{data}}, x \sim p_t(\cdot|z)} [\|u_t^\theta(x)\|^2] - 2\mathbb{E}_{t \sim \text{Unif}, x \sim p_t} [u_t^\theta(x)^T u_t^{\text{target}}(x)] + C_1 \end{aligned}$$

where (i) holds by definition, in (ii) we used the formula  $\|a - b\|^2 = \|a\|^2 - 2a^T b + \|b\|^2$ , in (iii) we define a constant  $C_1$  and in (iv) we used the sampling procedure of  $p_t$  given by Equation (12). Let us reexpress the second summand:

$$\begin{aligned}
 \mathbb{E}_{t \sim \text{Unif}, x \sim p_t} [u_t^\theta(x)^T u_t^{\text{target}}(x)] &\stackrel{(i)}{=} \int_0^1 \int p_t(x) u_t^\theta(x)^T u_t^{\text{target}}(x) dx dt \\
 &\stackrel{(ii)}{=} \int_0^1 \int p_t(x) u_t^\theta(x)^T \left[ \int u_t^{\text{target}}(x|z) \frac{p_t(x|z) p_{\text{data}}(z)}{p_t(x)} dz \right] dx dt \\
 &\stackrel{(iii)}{=} \int_0^1 \int \int u_t^\theta(x)^T u_t^{\text{target}}(x|z) p_t(x|z) p_{\text{data}}(z) dz dx dt \\
 &\stackrel{(iv)}{=} \mathbb{E}_{t \sim \text{Unif}, z \sim p_{\text{data}}, x \sim p_t(\cdot|z)} [u_t^\theta(x)^T u_t^{\text{target}}(x|z)]
 \end{aligned}$$

where in (i) we expressed the expected value as an integral, in (ii) we use Equation (18), in (iii) we use the fact that integrals are linear, in (iv) we express the integral as an expected value. Note that this was really the crucial step of the proof: The beginning of the equality used the marginal vector field  $u_t^{\text{target}}(x)$ , while the end uses the conditional vector field  $u_t^{\text{target}}(x|z)$ . We plug is into the equation for  $\mathcal{L}_{\text{FM}}$  to get:

$$\begin{aligned}
 \mathcal{L}_{\text{FM}}(\theta) &\stackrel{(i)}{=} \mathbb{E}_{t \sim \text{Unif}, z \sim p_{\text{data}}, x \sim p_t(\cdot|z)} [\|u_t^\theta(x)\|^2] - 2\mathbb{E}_{t \sim \text{Unif}, z \sim p_{\text{data}}, x \sim p_t(\cdot|z)} [u_t^\theta(x)^T u_t^{\text{target}}(x|z)] + C_1 \\
 &\stackrel{(ii)}{=} \mathbb{E}_{t \sim \text{Unif}, z \sim p_{\text{data}}, x \sim p_t(\cdot|z)} [\|u_t^\theta(x)\|^2 - 2u_t^\theta(x)^T u_t^{\text{target}}(x|z) + \|u_t^{\text{target}}(x|z)\|^2 - \|u_t^{\text{target}}(x|z)\|^2] + C_1 \\
 &\stackrel{(iii)}{=} \mathbb{E}_{t \sim \text{Unif}, z \sim p_{\text{data}}, x \sim p_t(\cdot|z)} [\|u_t^\theta(x) - u_t^{\text{target}}(x|z)\|^2] + \underbrace{\mathbb{E}_{t \sim \text{Unif}, z \sim p_{\text{data}}, x \sim p_t(\cdot|z)} [-\|u_t^{\text{target}}(x|z)\|^2]}_{C_2} + C_1 \\
 &\stackrel{(iv)}{=} \mathcal{L}_{\text{CFM}}(\theta) + \underbrace{C_2 + C_1}_{=: C}
 \end{aligned}$$

where in (i) we plugged in the derived equation, in (ii) we added and subtracted the same value, in (iii) we used the formula  $\|a - b\|^2 = \|a\|^2 - 2a^T b + \|b\|^2$  again, and in (iv) we defined a constant in  $\theta$ . This finishes the proof.  $\square$

Therefore, **flow matching training consists of minimizing the conditional flow matching loss**. The training procedure is summarized in Algorithm 3 and visualized in Figure 7. Note that there are several striking features about this algorithm: First, we never actually simulate any ODE during training. People call this feature of the algorithm **simulation-free**. This makes training extremely cheap as you don't have to roll out trajectories of the ODE during training (which takes a lot of steps). Second, the training is a simple regression objective - we are just regressing against  $u_t^{\text{target}}(x|z)$ . So it is not too different from supervised learning after all. Finally, the algorithm is extremely simple - it is hard to think of a much simpler training objective. All of this makes flow matching an extremely appealing method for large-scale machine learning models. Once  $u_t^\theta$  has been trained, we may simulate the flow model

$$dX_t = u_t^\theta(X_t) dt, \quad X_0 \sim p_{\text{init}} \quad (27)$$

via e.g., Algorithm 1 to obtain samples  $X_1 \sim p_{\text{data}}$ . This whole pipeline is called **flow matching** in the literature [25, 27, 1, 26]. Let us now instantiate the conditional flow matching loss for Gaussian probability paths:

---

**Algorithm 3** Flow Matching Training Procedure (for Gaussian CondOT path  $p_t(x|z) = \mathcal{N}(tz, (1-t)^2)$ )

---

**Require:** A dataset of samples  $z \sim p_{\text{data}}$ , neural network  $u_t^\theta$ 

- 1: **for** each mini-batch of data **do**
- 2:   Sample a data example  $z$  from the dataset.
- 3:   Sample a random time  $t \sim \text{Unif}_{[0,1]}$ .
- 4:   Sample noise  $\epsilon \sim \mathcal{N}(0, I_d)$
- 5:   Set

$$x = tz + (1-t)\epsilon \quad (\text{General case: } x \sim p_t(\cdot | z))$$

- 6:   Compute loss

$$\mathcal{L}(\theta) = \|u_t^\theta(x) - (z - \epsilon)\|^2 \quad (\text{General case: } = \|u_t^\theta(x) - u_t^{\text{target}}(x|z)\|^2)$$

- 7:   Update  $\theta \leftarrow \text{grad\_update}(\mathcal{L}(\theta))$ .
  - 8: **end for**
- 

**Example 13** (Flow Matching for Gaussian Conditional Probability Paths)

Let us return to the example of Gaussian probability paths  $p_t(\cdot|z) = \mathcal{N}(\alpha_t z; \beta_t^2 I_d)$ , where we may sample from the conditional path via

$$\epsilon \sim \mathcal{N}(0, I_d) \quad \Rightarrow \quad x_t = \alpha_t z + \beta_t \epsilon \sim \mathcal{N}(\alpha_t z, \beta_t^2 I_d) = p_t(\cdot|z). \quad (28)$$

As we derived in Equation (20), the conditional vector field  $u_t^{\text{target}}(x|z)$  is given by

$$u_t^{\text{target}}(x|z) = \left( \dot{\alpha}_t - \frac{\dot{\beta}_t}{\beta_t} \alpha_t \right) z + \frac{\dot{\beta}_t}{\beta_t} x, \quad (29)$$

where  $\dot{\alpha}_t = \partial_t \alpha_t$  and  $\dot{\beta}_t = \partial_t \beta_t$  are the respective time derivatives. Plugging in this formula, the conditional flow matching loss reads

$$\mathcal{L}_{\text{CFM}}(\theta) = \mathbb{E}_{t \sim \text{Unif}, z \sim p_{\text{data}}, x \sim \mathcal{N}(\alpha_t z, \beta_t^2 I_d)} [\|u_t^\theta(x) - \left( \dot{\alpha}_t - \frac{\dot{\beta}_t}{\beta_t} \alpha_t \right) z - \frac{\dot{\beta}_t}{\beta_t} x\|^2] \quad (30)$$

$$\stackrel{(i)}{=} \mathbb{E}_{t \sim \text{Unif}, z \sim p_{\text{data}}, \epsilon \sim \mathcal{N}(0, I_d)} [\|u_t^\theta(\alpha_t z + \beta_t \epsilon) - (\dot{\alpha}_t z + \dot{\beta}_t \epsilon)\|^2] \quad (31)$$

where in (i) we plugged in Equation (28) and replaced  $x$  by  $\alpha_t z + \beta_t \epsilon$ . Note the simplicity of  $\mathcal{L}_{\text{CFM}}$ : We sample a data point  $z$ , sample some noise  $\epsilon$  and then we take a mean squared error. Let us make this even more concrete for the special case of  $\alpha_t = t$ , and  $\beta_t = 1 - t$ . The corresponding probability  $p_t(x|z) = \mathcal{N}(tz, (1-t)^2)$  is sometimes referred to as the (Gaussian) **CondOT probability path**. Then we have  $\dot{\alpha}_t = 1$ ,  $\dot{\beta}_t = -1$ , so that

$$\mathcal{L}_{\text{cfm}}(\theta) = \mathbb{E}_{t \sim \text{Unif}, z \sim p_{\text{data}}, \epsilon \sim \mathcal{N}(0, I_d)} [\|u_t^\theta(tz + (1-t)\epsilon) - (z - \epsilon)\|^2]$$

Many famous state-of-the-art models have been trained using this simple yet effective procedure, e.g. *Stable Diffusion 3*, Meta's *Movie Gen Video*, and probably many more proprietary models. In Figure 7, we visualize

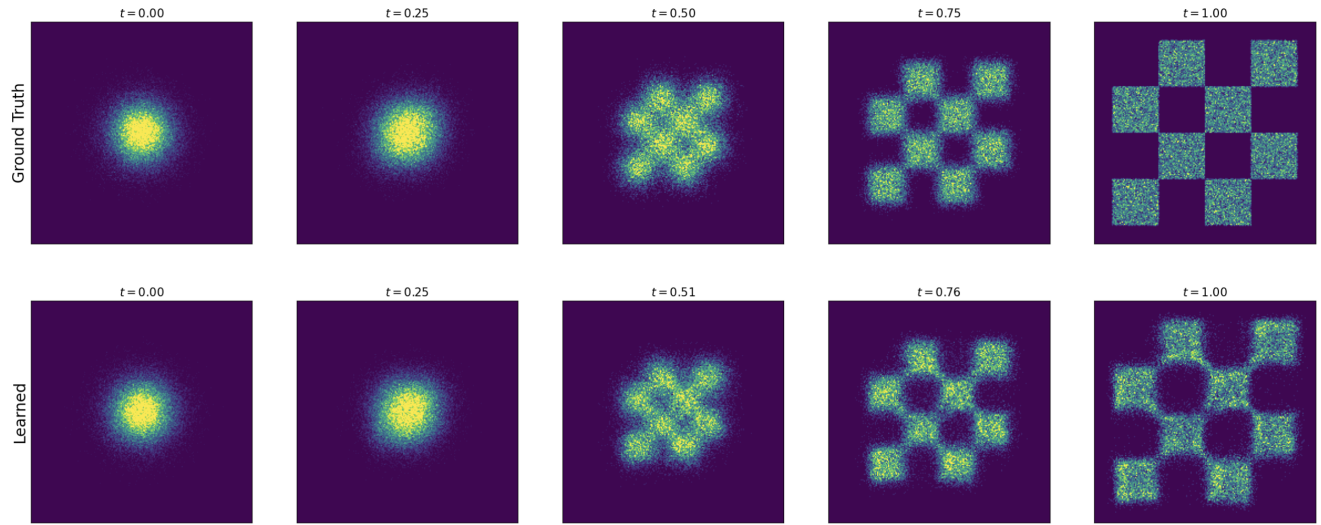


Figure 7: Illustration of [Theorem 12](#) with a Gaussian CondOT probability path: simulating an ODE from a trained flow matching model. The data distribution is the chess board pattern (top right). Top row: Histogram from ground truth marginal probability path  $p_t(x)$ . Bottom row: Histogram of samples from flow matching model. As one can see, the top row and bottom row match after training (up to training error). The model was trained using [Algorithm 3](#).

it in a simple example and in [Algorithm 3](#) we summarize the training procedure.

Let us summarize the results of this section.

#### Summary 14 (Flow Matching)

**Flow matching training consists of learning the marginal vector field  $u_t^{\text{target}}$ .** To construct it, we choose a **conditional probability path**  $p_t(x|z)$  that fulfils  $p_0(\cdot|z) = p_{\text{init}}$ ,  $p_1(\cdot|z) = \delta_z$ . Next, we find a **conditional vector field**  $u_t^{\text{target}}(x|z)$  such that its corresponding flow  $\psi_t^{\text{target}}(x|z)$  fulfills

$$X_0 \sim p_{\text{init}} \quad \Rightarrow \quad X_t = \psi_t^{\text{target}}(X_0|z) \sim p_t(\cdot|z),$$

or, equivalently, that  $u_t^{\text{target}}$  satisfies the continuity equation. Then the **marginal vector field** defined by

$$u_t^{\text{target}}(x) = \int u_t^{\text{target}}(x|z) \frac{p_t(x|z)p_{\text{data}}(z)}{p_t(x)} dz, \quad (32)$$

follows the marginal probability path, i.e.,

$$X_0 \sim p_{\text{init}}, \quad dX_t = u_t^{\text{target}}(X_t)dt \Rightarrow X_t \sim p_t \quad (0 \leq t \leq 1). \quad (33)$$

In particular,  $X_1 \sim p_{\text{data}}$  for this ODE, so that  $u_t^{\text{target}}$  "converts noise into data", as desired. To learn it, we minimize the **conditional flow matching loss**

$$\mathcal{L}_{\text{CFM}}(\theta) = \mathbb{E}_{t \sim \text{Unif}, z \sim p_{\text{data}}, x \sim p_t(\cdot|z)} [\|u_t^\theta(x) - u_t^{\text{target}}(x|z)\|^2]. \quad (34)$$

The most widely used example is the **Gaussian probability path**. For this case, the formulas become:

$$p_t(x|z) = \mathcal{N}(x; \alpha_t z, \beta_t^2 I_d) \quad (35)$$

$$u_t^{\text{flow}}(x|z) = \left( \dot{\alpha}_t - \frac{\dot{\beta}_t}{\beta_t} \alpha_t \right) z + \frac{\dot{\beta}_t}{\beta_t} x \quad (36)$$

$$\mathcal{L}_{\text{CFM}}(\theta) = \mathbb{E}_{t \sim \text{Unif}, z \sim p_{\text{data}}, \epsilon \sim \mathcal{N}(0, I_d)} [\|u_t^\theta(\alpha_t z + \beta_t \epsilon) - (\dot{\alpha}_t z + \dot{\beta}_t \epsilon)\|^2] \quad (37)$$

for **noise schedulers**  $\alpha_t, \beta_t \in \mathbb{R}$ , i.e. continuously differentiable, monotonic functions that we choose such that  $\alpha_0 = \beta_1 = 0$   $\alpha_1 = \beta_0 = 1$  (e.g.  $\alpha_t = t, \beta_t = 1 - t$ ).

## 4 Score Functions and Score Matching

In the last section, we showed how to train a flow model with flow matching. In this section, we discuss diffusion models and demonstrate how to train them using **score matching**.

### 4.1 Conditional and Marginal Score Functions

So far, the central object of interest for our investigation was a vector field  $u_t(x)$ . Diffusion models [45, 44] take a different perspective focused on **score functions**. Therefore, in this section, we will rephrase what we have learned here in the language of score functions - providing a novel perspective. Let  $q(x)$  be an arbitrary probability distribution. Then the **score function** of  $q$  is defined as  $\nabla \log q(x)$ , i.e. as the gradient of the log-likelihood of  $q$  with respect to  $x$ . The score has an intuitive meaning:  $\nabla \log q(x)$  is the direction of steepest ascent with respect to log-likelihood. This is illustrated in Figure 8.

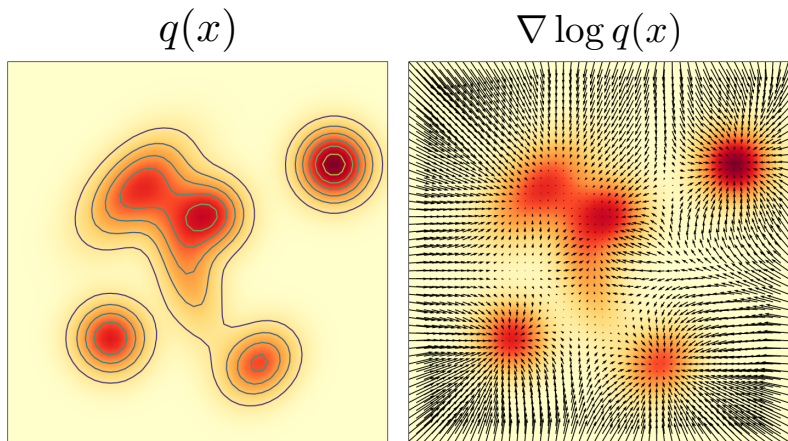


Figure 8: Illustration of score function  $\nabla \log q(x)$  plotted as black rows (right) of a general probability distribution  $q(x)$  (left).

Let us return to the setting of conditional probability paths  $p_t(x|z)$  and marginal probability paths  $p_t(x)$  as in Section 3. Then we can equivalently define the **conditional score function** as  $\nabla \log p_t(x|z)$  and the **marginal score function** as  $\nabla \log p_t(x)$ . Similar to Equation (18), the marginal score can be expressed via the conditional score function  $\nabla \log p_t(x|z)$  via

$$\nabla \log p_t(x) = \int \nabla \log p_t(x|z) \frac{p_t(x|z)p_{\text{data}}(z)}{p_t(x)} dz. \quad (38)$$

Hence, **the relation between the conditional and marginal score is analogous to the relation between the conditional and marginal vector field**. Note that we can prove Equation (38) via

$$\nabla \log p_t(x) = \frac{\nabla p_t(x)}{p_t(x)} = \frac{\nabla \int p_t(x|z)p_{\text{data}}(z) dz}{p_t(x)} = \frac{\int \nabla p_t(x|z)p_{\text{data}}(z) dz}{p_t(x)} = \int \nabla \log p_t(x|z) \frac{p_t(x|z)p_{\text{data}}(z)}{p_t(x)} dz, \quad (39)$$

where we have used the rule  $\partial_y \log y = 1/y$  combined with the chain rule twice.

#### Example 15 (Score Function for Gaussian Probability Paths.)

For the Gaussian path  $p_t(x|z) = \mathcal{N}(x; \alpha_t z, \beta_t^2 I_d)$ , we can use the form of the Gaussian probability density (see Equation (97)) to get

$$\nabla \log p_t(x|z) = \nabla \log \mathcal{N}(x; \alpha_t z, \beta_t^2 I_d) = -\frac{x - \alpha_t z}{\beta_t^2}. \quad (40)$$

Note that the score function for a Gaussian probability path is a linear function of  $x$  and  $z$ . The same is true for the conditional vector field  $u_t(x|z)$  (see Equation (20)). It is thus possible to convert between the two, as the next proposition illustrates.

**Proposition 1** (Conversion Formula for Gaussian Probability Paths)

For the Gaussian probability path  $p_t(x|z) = \mathcal{N}(\alpha_t z, \beta_t^2 I_d)$ , the conditional (resp. marginal) vector field and the conditional (resp. marginal) score are related by the following identities

$$u_t^{\text{target}}(x|z) = a_t \nabla \log p_t(x|z) + b_t x, \quad a_t = \left( \beta_t^2 \frac{\dot{\alpha}_t}{\alpha_t} - \dot{\beta}_t \beta_t \right), \quad b_t = \frac{\dot{\alpha}_t}{\alpha_t} \quad (41)$$

$$u_t^{\text{target}}(x) = a_t \nabla \log p_t(x) + b_t x. \quad (42)$$

In particular, we note that the conditional (resp. marginal) vector field can be recovered from the conditional (resp. marginal) score, and vice versa.

*Proof.* For the conditional vector field and conditional score, we can derive:

$$u_t^{\text{target}}(x|z) = \left( \dot{\alpha}_t - \frac{\dot{\beta}_t}{\beta_t} \alpha_t \right) z + \frac{\dot{\beta}_t}{\beta_t} x \stackrel{(i)}{=} \left( \beta_t^2 \frac{\dot{\alpha}_t}{\alpha_t} - \dot{\beta}_t \beta_t \right) \left( \frac{\alpha_t z - x}{\beta_t^2} \right) + \frac{\dot{\alpha}_t}{\alpha_t} x = \left( \beta_t^2 \frac{\dot{\alpha}_t}{\alpha_t} - \dot{\beta}_t \beta_t \right) \nabla \log p_t(x|z) + \frac{\dot{\alpha}_t}{\alpha_t} x$$

where in (i) we just did some algebra. By taking integrals, the same identity holds for the marginal flow vector field and the marginal score function:

$$\begin{aligned} u_t^{\text{target}}(x) &= \int u_t^{\text{target}}(x|z) \frac{p_t(x|z) p_{\text{data}}(z)}{p_t(x)} dz = \int [a_t \nabla \log p_t(x|z) + b_t x] \frac{p_t(x|z) p_{\text{data}}(z)}{p_t(x)} dz \\ &\stackrel{(i)}{=} a_t \nabla \log p_t(x) + b_t x \end{aligned}$$

where in (i) we used Equation (38) and the fact that posterior density integrates to 1.  $\square$

Proposition 1 is striking because it says that once we've learned  $u_t^{\text{target}}$  we've also learned the score function  $\nabla \log p_t(x)$ , and vice versa. Therefore, many diffusion models learn the score function  $\nabla \log p_t(x)$  instead via a neural network. We will discuss this in Section 4.3.

**Remark 16** (Reparameterization of the Score)

The reparameterization formula for Gaussian probability paths in Equation (41) is possible because both sides (conditional vector field and conditional score) are *linear* functions of  $x$  and  $z$ . Once we marginalize (marginal vector field and marginal score), both sides are just a linear reparameterization of the posterior mean  $\mathbb{E}_{z|x}[z]$ . It follows that any quantity that allows to recover  $\mathbb{E}_{z|x}[z]$  can in turn be used to recover the unconditional vector field and score. Further, doing so might even be preferable from a numerical/training stability standpoint. One common choice is the posterior mean itself, often referred to as the *denoiser*. Formally, we define the **conditional and marginal denoiser** as

$$D_t(x|z) = z, \quad D_t(x) = \int z \frac{p_t(x|z) p_{\text{data}}(z)}{p_t(x)} dz \stackrel{(i)}{=} \frac{1}{\dot{\alpha}_t \beta_t - \alpha_t \dot{\beta}_t} (\beta_t u_t^{\text{target}}(x_t) - \dot{\beta}_t x_t). \quad (43)$$

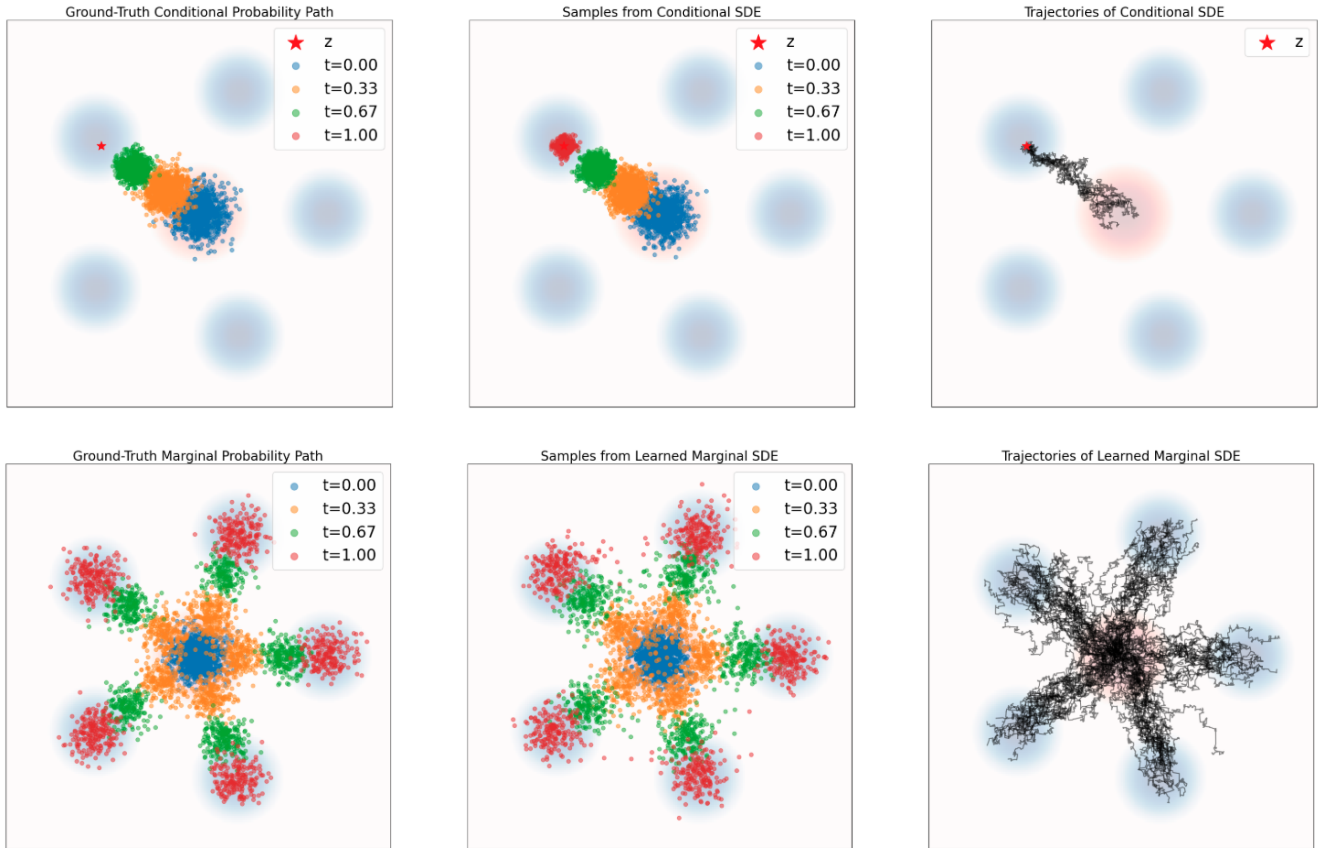


Figure 9: Illustration of [Theorem 17](#). Simulating a probability path with SDEs. This repeats the plots from [Figure 6](#) with SDE sampling using [Equation \(44\)](#). Data distribution  $p_{\text{data}}$  in blue background. Gaussian  $p_{\text{init}}$  in red background. Top row: Conditional path. Bottom row: Marginal probability path. As one can see, the SDE transports samples from  $p_{\text{init}}$  into samples from  $\delta_z$  (for the conditional path) and to  $p_{\text{data}}$  (for the marginal path).

Here, (i) follows from an equivalent derivation as in [Proposition 1](#). The denoiser has a very intuitive interpretation: it is the expected value of clean data  $z$  given noisy data  $x$ .<sup>a</sup> People often call such models **denoising diffusion models** as learning  $D_t$  and learning  $u_t^{\text{target}}$  are theoretically equivalent.

<sup>a</sup>Food for thought: will the denoiser always output a “clean” data point? Why or why not, and what might this depend on?

## 4.2 Sampling with SDEs

So far, we have demonstrated how one can construct a trajectory  $X_t$  of an ODE that follows a desired probability path  $p_t$  via a marginal vector field  $u_t^{\text{target}}$ . But this approach is constrained to flow models. What about diffusion models? Using score functions, let us now extend this result to SDEs.

### Theorem 17 (SDE Extension Trick)

Define the conditional and marginal vector fields  $u_t^{\text{target}}(x|z)$  and  $u_t^{\text{target}}(x)$  as before. Then, for any diffusion

coefficient  $\sigma_t \geq 0$ , we may construct an SDE by adding **stochastic dynamics** to the dynamics of the **original ODE** as follows:

$$X_0 \sim p_{\text{init}}, \quad dX_t = u_t^{\text{target}}(X_t)dt + \frac{\sigma_t^2}{2} \nabla \log p_t(X_t)dt + \sigma_t dW_t \quad (44)$$

$$= \left[ u_t^{\text{target}}(X_t) + \frac{\sigma_t^2}{2} \nabla \log p_t(X_t) \right] dt + \sigma_t dW_t$$

$$\Rightarrow X_t \sim p_t \quad (0 \leq t \leq 1). \quad (45)$$

In particular,  $X_1 \sim p_{\text{data}}$  for this SDE. We note that the **stochastic dynamics** are closely related to the **Langevin dynamics**, and can be thought of as injecting noise while preserving the marginal distribution  $p_t$ . We discuss Langevin dynamics briefly in [Remark 20](#).

We illustrate the dynamics described in [Theorem 17](#) in [Figure 9](#). As one can see, the trajectories are now zig-zagged, illustrating the stochastic nature of the SDE’s evolution. As [Theorem 17](#) establishes however, the marginals  $p_t$  stay the same. Note that the above result is striking in that we can choose *any* diffusion coefficient  $\sigma_t \geq 0$  even after having trained the networks. In theory, [Theorem 17](#) holds for any choice of  $\sigma_t$ . However, *in practice*, we suffer from both *training error* (the neural network does not perfectly approximate the marginal vector field and score) and *simulation error* (e.g. for  $\sigma_t \gg 0$ , we would need to take prohibitively small step sizes in [Algorithm 2](#)). In practice, for a fixed trained model, there is then an optimal  $\sigma_t \geq 0$  which can be empirically determined [[23](#), [1](#), [28](#)].<sup>2</sup>

For Gaussian probability paths, we get the score function for free by having learned the marginal vector field.

**Example 18** (Gaussian SDE Extension Trick)

By [Proposition 1](#), for Gaussian probability paths, we can express the SDE from [Theorem 17](#) purely using score functions:

$$X_0 \sim p_{\text{init}}, \quad dX_t = \left[ \left( a_t + \frac{\sigma_t^2}{2} \right) \nabla \log p_t(X_t) + b_t X_t \right] dt + \sigma_t dW_t \quad (46)$$

$$\Rightarrow X_t \sim p_t \quad (0 \leq t \leq 1) \quad (47)$$

where  $a_t, b_t$  are defined as in [Proposition 1](#).

In the remainder of this section, we will prove [Theorem 17](#) via the **Fokker-Planck equation**, which extends the continuity equation from ODEs to SDEs. To do so, let us first define the **Laplacian** operator  $\Delta$  via

$$\Delta w_t(x) = \sum_{i=1}^d \frac{\partial^2}{\partial x_i^2} w_t(x) = \text{div}(\nabla w_t)(x), \quad (48)$$

for scalar field  $w_t : \mathbb{R}^d \rightarrow \mathbb{R}$ .

<sup>2</sup>Again, we stress that the existence of a “best  $\sigma_t$ ” is an artifact of imperfectly trained models and finite compute budgets rather than a theoretical statement about the dynamics in their continuous limit.

**Theorem 19** (Fokker-Planck Equation)

Let  $p_t$  be a probability path and let us consider the SDE

$$X_0 \sim p_{\text{init}}, \quad dX_t = u_t(X_t)dt + \sigma_t dW_t.$$

Then  $X_t$  has distribution  $p_t$  for all  $0 \leq t \leq 1$  if and only if the **Fokker-Planck equation** holds:

$$\partial_t p_t(x) = -\text{div}(p_t u_t)(x) + \frac{\sigma_t^2}{2} \Delta p_t(x) \quad \text{for all } x \in \mathbb{R}^d, 0 \leq t \leq 1, \quad (49)$$

A self-contained proof of the Fokker-Planck equation can be found in [Section B](#). Note that [Theorem 11](#) is recovered from the Fokker-Planck equation when  $\sigma_t = 0$ . The additional Laplacian term  $\Delta p_t$  might be hard to rationalize at first. Those familiar with physics will note that the same term also appears in the heat equation (which is in fact a special case of the Fokker-Planck equation). Heat diffuses through a medium. We also add a diffusion process (not a physical but a mathematical one) and hence we add this additional Laplacian term. Let us now use the Fokker-Planck equation to help us prove [Theorem 17](#).

*Proof of [Theorem 17](#).* By [Theorem 19](#), we need to show that the SDE defined in [Equation \(44\)](#) satisfies the Fokker-Planck equation for  $p_t$ . We can do this by direction calculation:

$$\begin{aligned} \partial_t p_t(x) &\stackrel{(i)}{=} -\text{div}(p_t u_t^{\text{target}})(x) \\ &\stackrel{(ii)}{=} -\text{div}(p_t u_t^{\text{target}})(x) - \frac{\sigma_t^2}{2} \Delta p_t(x) + \frac{\sigma_t^2}{2} \Delta p_t(x) \\ &\stackrel{(iii)}{=} -\text{div}(p_t u_t^{\text{target}})(x) - \text{div}\left(\frac{\sigma_t^2}{2} \nabla p_t\right)(x) + \frac{\sigma_t^2}{2} \Delta p_t(x) \\ &\stackrel{(iv)}{=} -\text{div}(p_t u_t^{\text{target}})(x) - \text{div}\left(p_t \left[\frac{\sigma_t^2}{2} \nabla \log p_t\right]\right)(x) + \frac{\sigma_t^2}{2} \Delta p_t(x) \\ &\stackrel{(v)}{=} -\text{div}\left(p_t \left[u_t^{\text{target}} + \frac{\sigma_t^2}{2} \nabla \log p_t\right]\right)(x) + \frac{\sigma_t^2}{2} \Delta p_t(x), \end{aligned}$$

where in (i) we used [Theorem 11](#), in (ii) we added and subtracted the same term, in (iii) we used the definition of the Laplacian ([Equation \(48\)](#)), in (iv) we used that  $\nabla \log p_t = \frac{\nabla p_t}{p_t}$ , and in (v) we used the linearity of the divergence operator. The above derivation shows that the SDE defined in [Equation \(44\)](#) satisfies the Fokker-Planck equation for  $p_t$ . By [Theorem 19](#), this implies  $X_t \sim p_t$  for  $0 \leq t \leq 1$ , as desired.  $\square$

**Remark 20** (Optional: Langevin Dynamics)

The above construction has a famous special case when the probability path is constant, i.e.  $p_t = p$  for a fixed distribution  $p$ . In this case, we set  $u_t^{\text{target}} = 0$  and obtain the SDE

$$dX_t = \frac{\sigma_t^2}{2} \nabla \log p(X_t)dt + \sigma_t dW_t, \quad (50)$$

which is commonly known as **Langevin dynamics**. The fact that  $p_t$  is constant implies that  $\partial_t p_t(x) = 0$ . It follows immediately from [Theorem 17](#) that these dynamics satisfy the Fokker-Planck equation for the static path

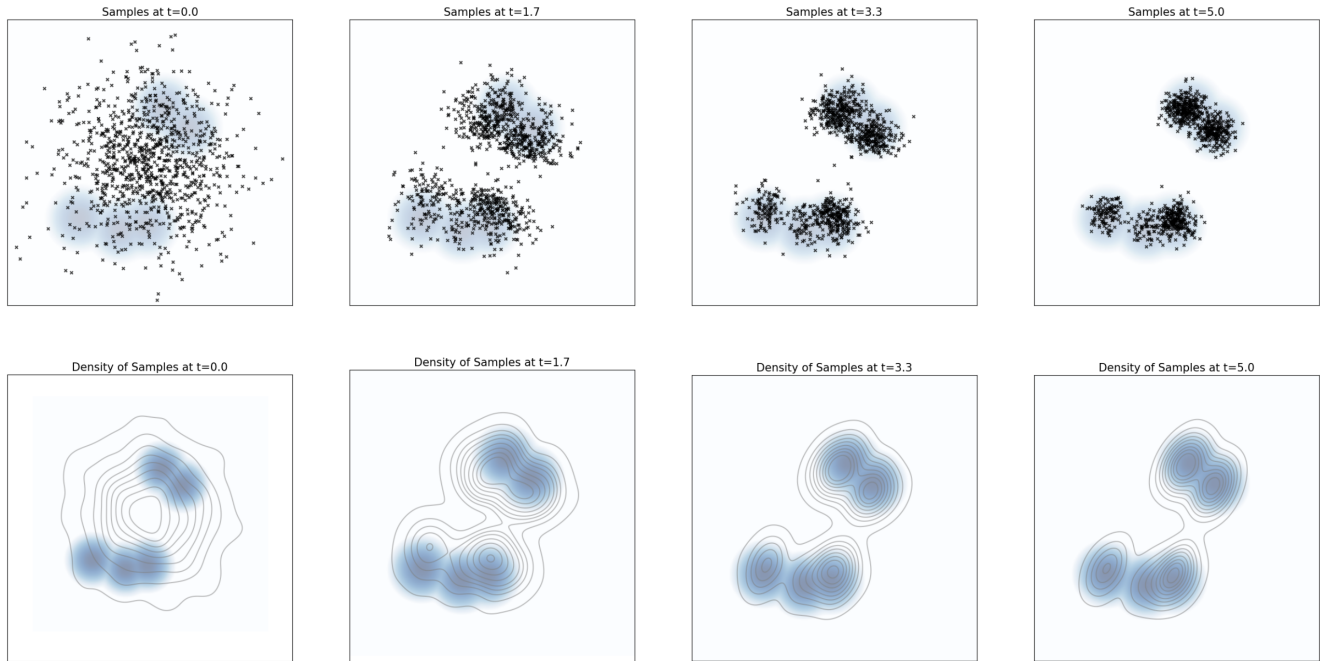


Figure 10: Top row: Particles evolving under the Langevin dynamics given by Equation (50), with  $p(x)$  taken to be a Gaussian mixture with 5 modes. Bottom row: A kernel density estimate of the same samples shown in the top row. As one can see, the distribution of samples converges to the equilibrium distribution  $p$  (blue background colour).

$p_t = p$  in Theorem 17. Therefore, we may conclude that  $p$  is a stationary distribution of Langevin dynamics:

$$X_0 \sim p \Rightarrow X_t \sim p \quad (t \geq 0).$$

As with many Markov processes, these dynamics converge to the stationary distribution  $p$  under rather general conditions. That is, if we instead we take  $X_0 \sim p' \neq p$ , so that  $X_t \sim p'_t$ , then under mild conditions  $p_t \rightarrow p$ . This fact makes Langevin dynamics extremely useful, and it accordingly serves as the basis for e.g., **molecular dynamics** simulations, and many other Markov chain Monte Carlo (MCMC) methods across Bayesian statistics and the natural sciences. In particular, the Ornstein-Uhlenbeck processes are recovered as the special case of the Langevin dynamics when  $p$  is a Gaussian, and serve as the basis for initial formulations of diffusion models.

**Remark 21** (Optional: GLASS Flows, Stochastic evolution with ODEs)

The remarkable property of SDE sampling (compared to ODEs) is that the evolution becomes stochastic, i.e. the initial point  $X_0$  does not fully determine  $X_t$  for  $t > 0$ . Perhaps surprisingly, it is also possible to get the same stochastic transitions purely via ODEs via a simple sampling trick called *GLASS Flows* [20]. This allows to exploit the stochastic nature of SDEs (e.g. via search algorithms) while keeping the efficiency of ODEs.

### 4.3 Score Matching

It remains to show how we can learn the marginal score function  $\nabla \log p_t(x)$ . Of course, for Gaussian probability paths, we can simply transform  $u_t^{\text{target}}(x)$  by [Proposition 1](#). However, what about in general? It turns out that we can also learn marginal score functions directly. To approximate the marginal score  $\nabla \log p_t$ , we use a neural network that we call **score network**  $s_t^\theta : \mathbb{R}^d \times [0, 1] \rightarrow \mathbb{R}^d$ . In the same way as before, we can design a **score matching** loss and a **denoising score matching** loss:

$$\begin{aligned} \mathcal{L}_{\text{SM}}(\theta) &= \mathbb{E}_{t \sim \text{Unif}, z \sim p_{\text{data}}, x \sim p_t(\cdot|z)} \left[ \left\| s_t^\theta(x) - \nabla \log p_t(x) \right\|^2 \right] && \blacktriangleright \text{score matching loss} \\ \mathcal{L}_{\text{CSM}}(\theta) &= \mathbb{E}_{t \sim \text{Unif}, z \sim p_{\text{data}}, x \sim p_t(\cdot|z)} \left[ \left\| s_t^\theta(x) - \nabla \log p_t(x|z) \right\|^2 \right] && \blacktriangleright \text{conditional score matching loss} \end{aligned}$$

where again the difference is using the marginal score  $\nabla \log p_t(x)$  vs. using the conditional score  $\nabla \log p_t(x|z)$ . As before, we ideally would want to minimize the score matching loss but can't because we don't know  $\nabla \log p_t(x)$ . But similarly as before, the denoising score matching loss is a tractable alternative:

#### Theorem 22

The score matching loss equals the denoising score matching loss up to a constant:

$$\mathcal{L}_{\text{SM}}(\theta) = \mathcal{L}_{\text{CSM}}(\theta) + C,$$

where  $C$  is independent of parameters  $\theta$ . Therefore, their gradients coincide:

$$\nabla_\theta \mathcal{L}_{\text{SM}}(\theta) = \nabla_\theta \mathcal{L}_{\text{CSM}}(\theta).$$

In particular, for the minimizer  $\theta^*$ , it will hold that  $s_t^{\theta^*} = \nabla \log p_t$ .

*Proof.* Note that the formula for  $\nabla \log p_t$  ([Equation \(38\)](#)) looks the same as the formula for  $u_t^{\text{target}}$  ([Equation \(18\)](#)). Therefore, the proof is identical to the proof of [Theorem 12](#) replacing  $u_t^{\text{target}}$  with  $\nabla \log p_t$ .  $\square$

#### Example 23 (Denoising Diffusion Models: Score Matching for Gaussian Probability Paths)

Let us instantiate the denoising score matching loss for the case of  $p_t(x|z) = \mathcal{N}(\alpha_t z, \beta_t^2 I_d)$ . As we derived in [Equation \(40\)](#), the conditional score  $\nabla \log p_t(x|z)$  has the formula

$$\nabla \log p_t(x|z) = -\frac{x - \alpha_t z}{\beta_t^2}. \tag{51}$$

Plugging in this formula, the conditional score matching loss becomes:

$$\begin{aligned}\mathcal{L}_{\text{CSM}}(\theta) &= \mathbb{E}_{t \sim \text{Unif}, z \sim p_{\text{data}}, x \sim p_t(\cdot|z)} \left[ \left\| s_t^\theta(x) + \frac{x - \alpha_t z}{\beta_t^2} \right\|^2 \right] \\ &\stackrel{(i)}{=} \mathbb{E}_{t \sim \text{Unif}, z \sim p_{\text{data}}, \epsilon \sim \mathcal{N}(0, I_d)} \left[ \left\| s_t^\theta(\alpha_t z + \beta_t \epsilon) + \frac{\epsilon}{\beta_t} \right\|^2 \right] \\ &= \mathbb{E}_{t \sim \text{Unif}, z \sim p_{\text{data}}, \epsilon \sim \mathcal{N}(0, I_d)} \left[ \frac{1}{\beta_t^2} \left\| \beta_t s_t^\theta(\alpha_t z + \beta_t \epsilon) + \epsilon \right\|^2 \right]\end{aligned}$$

where in (i) we plugged in Equation (28) and replaced  $x$  by  $\alpha_t z + \beta_t \epsilon$ . Note that the network  $s_t^\theta$  essentially learns to predict the noise that was used to corrupt a data sample  $z$ . This explains why the above training loss is called **denoising score matching**. It was soon realized that the above loss is numerically unstable for  $\beta_t \approx 0$  close to zero (i.e. denoising score matching only works if you add a sufficient amount of noise). In some of the first works on denoising diffusion models (see **Denoising Diffusion Probabilistic Models**, [17]) it was therefore proposed to drop the constant  $\frac{1}{\beta_t^2}$  in the loss and reparameterize  $s_t^\theta$  into a **noise predictor** network  $\epsilon_t^\theta : \mathbb{R}^d \times [0, 1] \rightarrow \mathbb{R}^d$  via:

$$-\beta_t s_t^\theta(x) = \epsilon_t^\theta(x) \quad \Rightarrow \quad \mathcal{L}_{\text{DDPM}}(\theta) = \mathbb{E}_{t \sim \text{Unif}, z \sim p_{\text{data}}, \epsilon \sim \mathcal{N}(0, I_d)} \left[ \left\| \epsilon_t^\theta(\alpha_t z + \beta_t \epsilon) - \epsilon \right\|^2 \right]$$

As before, the network  $\epsilon_t^\theta$  essentially learns to predict the noise that was used to corrupt a data sample  $z$ . In Algorithm 4, we summarize the training procedure.

---

**Algorithm 4** Score Matching Training Procedure for Gaussian probability path

---

**Require:** A dataset of samples  $z \sim p_{\text{data}}$ , score network  $s_t^\theta$  or noise predictor  $\epsilon_t^\theta$

- 1: **for** each mini-batch of data **do**
- 2:   Sample a data example  $z$  from the dataset.
- 3:   Sample a random time  $t \sim \text{Unif}_{[0,1]}$ .
- 4:   Sample noise  $\epsilon \sim \mathcal{N}(0, I_d)$
- 5:   Set  $x_t = \alpha_t z + \beta_t \epsilon$  (General case:  $x_t \sim p_t(\cdot|z)$ )
- 6:   Compute loss

$$\mathcal{L}(\theta) = \left\| s_t^\theta(x_t) + \frac{\epsilon}{\beta_t} \right\|^2 \quad \text{(General case: } = \left\| s_t^\theta(x_t) - \nabla \log p_t(x_t|z) \right\|^2)$$

$$\text{Alternatively: } \mathcal{L}(\theta) = \left\| \epsilon_t^\theta(x_t) - \epsilon \right\|^2$$

- 7:   Update the model parameters  $\theta$  via gradient descent on  $\mathcal{L}(\theta)$ .
  - 8: **end for**
- 

Let us summarize the results of this section:

**Summary 24** (Score Functions, Score Matching, and Stochastic Sampling)

Let  $p_t(x|z), p_t(x)$  be the conditional and marginal probability path. The **conditional score function** is given by  $\nabla \log p_t(x|z)$  and the **marginal score function** is given by  $\nabla \log p_t(x)$ . For every diffusion coefficient  $\sigma_t \geq 0$ ,

the trajectories of the following SDE follow the probability path:

$$X_0 \sim p_{\text{init}}, \quad dX_t = \left[ u_t^{\text{target}}(X_t) + \frac{\sigma_t^2}{2} \nabla \log p_t(X_t) \right] dt + \sigma_t dW_t \quad (52)$$

$$\Rightarrow X_t \sim p_t \quad (0 \leq t \leq 1), \quad (53)$$

where  $u_t^{\text{target}}(x)$  be the marginal vector field as before (see Equation (18)).

**Score Matching.** To learn the marginal score function  $\nabla \log p_t(x)$ , we can use a **score network**  $s_t^\theta$  and train it via **denoising score matching**

$$\mathcal{L}_{\text{CSM}}(\theta) = \mathbb{E}_{z \sim p_{\text{data}}, t \sim \text{Unif}, x \sim p_t(\cdot|z)} [\|s_t^\theta(x) - \nabla \log p_t(x|z)\|^2] \quad (\text{denoising score matching loss}) \quad (54)$$

**Gaussian Probability Paths.** For the - most important - case of a Gaussian probability path  $p_t(x|z) = \mathcal{N}(x; \alpha_t z, \beta_t^2 I_d)$ , there is no need to train  $s_t^\theta$  and  $u_t^\theta$  separately as we can convert them via the formula:

$$u_t^\theta(x) = a_t s_t^\theta(x) + b_t x, \quad a_t = \left( \beta_t^2 \frac{\dot{\alpha}_t}{\alpha_t} - \dot{\beta}_t \beta_t \right), \quad b_t = \frac{\dot{\alpha}_t}{\alpha_t}$$

After training, we can simulate the following SDE

$$X_0 \sim p_{\text{init}}, \quad dX_t = \left[ \left( 1 + \frac{\sigma_t^2}{2a_t} \right) u_t^\theta(X_t) - \frac{\sigma_t^2 b_t}{2a_t} X_t \right] dt + \sigma_t dW_t \quad (55)$$

$$= \left[ \left( a_t + \frac{\sigma_t^2}{2} \right) s_t^\theta(X_t) + b_t X_t \right] dt + \sigma_t dW_t \quad (56)$$

for any diffusion coefficient  $\sigma_t \geq 0$  to obtain approximate samples  $X_1 \sim p_{\text{data}}$ . One can empirically find the optimal  $\sigma_t \geq 0$ .

---

## 5 Guidance: How To Condition on a Prompt

So far, the generative models we considered were **unguided**, e.g. an image model would simply generate *some* image. Mathematically speaking, this meant that our model returned samples from an *unconditional* data distribution  $p_{\text{data}}(z)$ . However, in most cases, our goal is not to merely generate an arbitrary object, but to generate an object **conditioned on some additional information**. In other words, we want to **guide** the model to generate objects of a certain kind. For example, one might imagine a generative model for images which takes in a text prompt  $y$ , and then generates an image  $x$  that fits to the text prompt  $y$ . As discussed in [Section 1](#), this means that we want to sample from  $p_{\text{data}}(z|y)$ , that is, the **guided data distribution conditioned on  $y$** . We are going to discuss this in this section.

### Remark 25 (Terminology)

To avoid a notation and terminology clash with the use of the word “conditional” to refer to conditioning on  $z \sim p_{\text{data}}$  (conditional probability path/vector field), we will make use of the term **guided** to refer specifically to conditioning on  $y$  such as a text prompt.

### 5.1 Vanilla Guidance

First, we discuss the “standard” way of how one would go about building a guided generative model. The short answer is as follows: We simply provide the input prompt  $y$  to the network during training and inference and do everything in the same way as before. We formalize this in the following. We think of a conditioning variable or prompt  $y$  to live in a space  $\mathcal{Y}$ . When  $y$  corresponds to a text-prompt, for example,  $\mathcal{Y}$  is the space of all texts. When  $y$  corresponds to some discrete class label,  $\mathcal{Y}$  would be discrete. We pose no constraints on  $\mathcal{Y}$ .

We define a **guided diffusion model** to consist of a **guided vector field**  $u_t^\theta(\cdot|y)$ , parameterized by some neural network, and a time-dependent diffusion coefficient  $\sigma_t$ , together given by

$$\text{Neural network: } u^\theta : \mathbb{R}^d \times \mathcal{Y} \times [0, 1] \rightarrow \mathbb{R}^d, (x, y, t) \mapsto u_t^\theta(x|y)$$

$$\text{Fixed: } \sigma_t : [0, 1] \rightarrow [0, \infty), t \mapsto \sigma_t$$

Notice the difference from [summary 7](#): we are additionally guiding  $u_t^\theta$  with the input  $y \in \mathcal{Y}$ . For any such  $y \in \mathcal{Y}$ , samples may then be generated from such a model as follows:

- |   |   |
|---|---|
| <b>Initialization:</b> $X_0 \sim p_{\text{init}}$               | ► Initialize with simple distribution (such as a Gaussian)              |
| <b>Simulation:</b> $dX_t = u_t^\theta(X_t y)dt + \sigma_t dW_t$ | ► Simulate SDE from $t = 0$ to $t = 1$ .                                |
| <b>Goal:</b> $X_1 \sim p_{\text{data}}(\cdot y)$                | ► Goal is for $X_1$ to be distributed like $p_{\text{data}}(\cdot y)$ . |

When  $\sigma_t = 0$ , we say that such a model is a **guided flow model**. In the following, we restrict ourselves to flow matching and flow models to make things more concise but everything applies similarly to the general case.

Next, we discuss: How would we train a guided flow model  $u_t^\theta(x|y)$ ? A simple trick might to fix our choice of  $y$ , and to take our data distribution as  $p_{\text{data}}(x|y)$ . Then we have recovered the unguided generative problem as



Figure 11: Image generation with prompt/class  $y = \text{“corgi dog”}$ . Left: samples generated with vanilla guidance - the images do not fit well to the prompt. Right: samples generated with classifier guidance and  $w = 4$ . As shown, classifier-free guidance improves the adherence to the prompt. Figure taken from [18].

before, and we can accordingly construct a generative model using the conditional flow matching objective, viz.,

$$\mathbb{E}_{z \sim p_{\text{data}}(\cdot|y), x \sim p_t(\cdot|z)} \|u_t^\theta(x|y) - u_t^{\text{target}}(x|z)\|^2. \quad (57)$$

Note that the label  $y$  does not affect the conditional probability path  $p_t(\cdot|z)$  or the conditional vector field  $u_t^{\text{target}}(x|z)$  (although in principle, we could make it dependent). Expanding the expectation over all such choices of  $y$ , we thus obtain a **guided conditional flow matching objective**

$$\mathcal{L}_{\text{CFM}}^{\text{guided}}(\theta) = \mathbb{E}_{(z,y) \sim p_{\text{data}}(z,y), t \sim \text{Unif}[0,1], x \sim p_t(\cdot|z)} \|u_t^\theta(x|y) - u_t^{\text{target}}(x|z)\|^2. \quad (58)$$

One of the main differences between the guided objective in Equation (58) and the unguided objective from Equation (26) is that here we are sampling  $(z, y) \sim p_{\text{data}}$  rather than just  $z \sim p_{\text{data}}$ . The reason is that our data distribution is now, in principle, a joint distribution over e.g., both images  $z$  and text prompts  $y$ . In practice, this means that a PyTorch implementation of Equation (58) would involve a dataloader which returned batches of **both  $z$  and  $y$** .

## 5.2 Classifier-Free Guidance

In theory, vanilla guidance should lead to a faithful generation procedure of  $p_{\text{data}}(\cdot|y)$ . However, it was soon empirically realized that images samples with this procedure did not fit well enough to the desired label  $y$  (see Figure 11). This can have a diversity of reasons: the model might underfit (i.e. we do not actually learn the true marginal vector field) or our data might be imperfect (e.g. text-image pairs from the world wide web have a lot of errors). Therefore, to truly generate samples that fit better to a prompt, we have to find a way to artificially reinforce the prompt variable  $y$ . The main technique for doing so is called **classifier-free guidance** that is widely used in the context of state-of-the-art diffusion models, and which we discuss next.

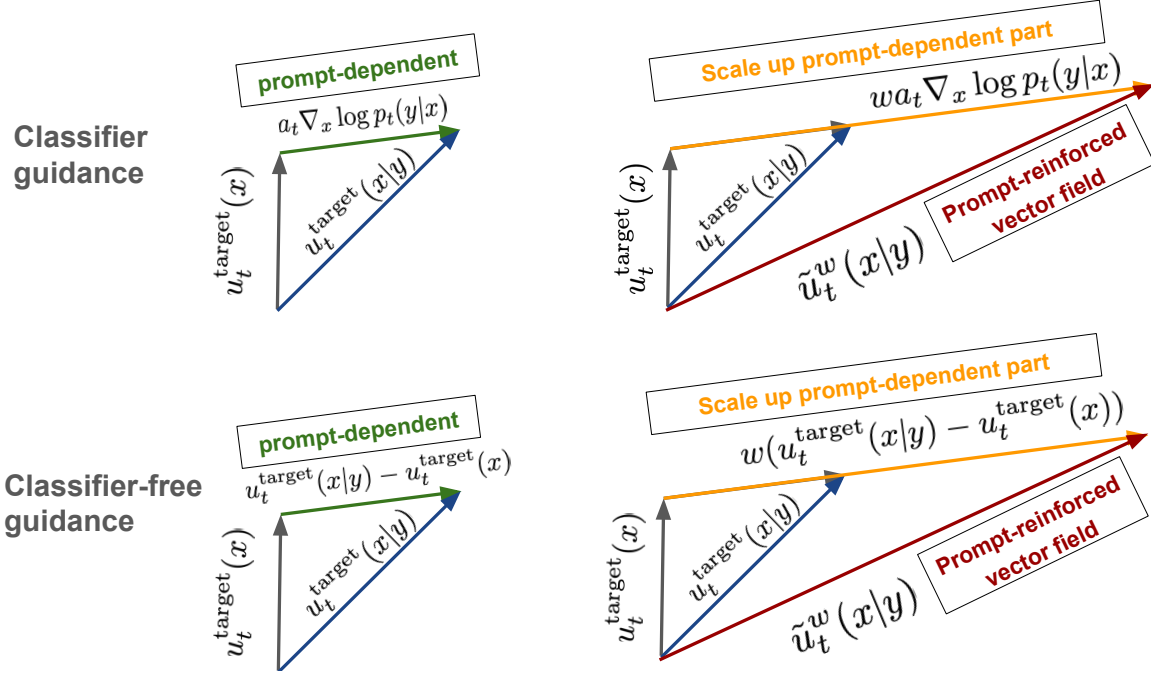


Figure 12: Illustration of classifier and classifier-free guidance. Classifier guidance decomposes the guided vector field  $u_t^{\text{target}}(x|y)$  and the gradient of a classifier  $\log p_t(y|x)$  and scales up the classifier with guidance scale  $w > 1$ . Classifier-free guidance scales up the difference between both vector fields, thereby achieving the same effect but without having to train a separate classifier model.

**Classifier Guidance.** For simplicity, we will focus here on the case of Gaussian probability paths. Recall from Equation (15) that a Gaussian conditional probability path is given by  $p_t(\cdot|z) = \mathcal{N}(\alpha_t z, \beta_t^2 I_d)$  where the noise schedulers  $\alpha_t$  and  $\beta_t$  are continuously differentiable, monotonic, and satisfy  $\alpha_0 = \beta_1 = 0$  and  $\alpha_1 = \beta_0 = 1$ . Further, recall that we can use Proposition 1 to rewrite the guided vector field  $u_t^{\text{target}}(x|y)$  in the following form using the guided score function  $\nabla \log p_t(x|y)$

$$u_t^{\text{target}}(x|y) = a_t \nabla \log p_t(x|y) + b_t x, \quad (59)$$

Next, realize that  $p_t(x|y)$  is a conditional density. Hence, we can use Bayes' rule to rewrite the guided score as

$$p_t(x|y) = \frac{p_t(x)p_t(y|x)}{p_t(y)} \quad (60)$$

$$\nabla \log p_t(x|y) = \nabla \log \left( \frac{p_t(x)p_t(y|x)}{p_t(y)} \right) = \nabla \log p_t(x) + \nabla \log p_t(y|x), \quad (61)$$

where we used that the gradient  $\nabla$  is taken with respect to the variable  $x$ , so that  $\nabla \log p_t(y) = 0$ . We may thus rewrite

$$u_t^{\text{target}}(x|y) = b_t x + a_t (\nabla \log p_t(x) + \nabla \log p_t(y|x)) = u_t^{\text{target}}(x) + a_t \nabla \log p_t(y|x).$$

Notice the shape of the above equation: The guided vector field  $u_t^{\text{target}}(x|y)$  is a sum of the unguided vector field  $u_t^{\text{target}}(x)$  plus a gradient of the likelihood  $p_t(y|x)$  of the guidance variable  $y$ . As people observed that their image

$x$  did not fit their prompt  $y$  well enough, it was a natural idea to scale up the contribution of the  $\nabla \log p_t(y|x)$  term, yielding

$$\tilde{u}_t(x|y) = u_t^{\text{target}}(x) + wa_t \nabla \log p_t(y|x), \quad (\text{classifier guidance}) \quad (62)$$

where  $w > 1$  is known as the **guidance scale**. How can we learn the term  $\log p_t(y|x)$ ? Note that this can be considered as a sort of classifier of noised data (i.e. it gives the log-likelihoods of  $y$  given  $x$ ). So we can simply learn it via supervised learning. This leads to **classifier guidance** [11, 43] (see Figure 12 for an illustration). Classifier guidance was largely superseded by classifier-free guidance, which is why we will not discuss it further here. However, it forms the basis for the classifier-free guidance, as we will see next. Finally, note that this is a heuristic: for  $w \neq 1$ , it holds that  $\tilde{u}_t(x|y) \neq u_t^{\text{target}}(x|y)$ , i.e. therefore not the ‘‘true’’ guided vector field.

**Classifier-Free Guidance.** While classifier guidance is possible in principle, it comes with difficulties: The first thing is that we need to train a classifier alongside a flow/diffusion model - so we have 2 networks instead of 1. Further, if the  $y$  is high-dimensional, e.g. a text prompt and not just a class, then  $p_t(y|x)$  might be very hard to learn and the gradient  $\nabla \log p_t(y|x)$  hard to obtain. For this reason, **classifier-free guidance** [18] was introduced. Classifier-free guidance results in the theoretically equivalent effect as classifier guidance but without having to train a separate classifier.

To do so, we may again apply the equality

$$\nabla \log p_t(x|y) = \nabla \log p_t(x) + \nabla \log p_t(y|x)$$

to obtain

$$\begin{aligned} \tilde{u}_t(x|y) &= u_t^{\text{target}}(x) + wa_t \nabla \log p_t(y|x) \\ &= u_t^{\text{target}}(x) + wa_t (\nabla \log p_t(x|y) - \nabla \log p_t(x)) \\ &= u_t^{\text{target}}(x) - (wb_t x + wa_t \nabla \log p_t(x)) + (wb_t x + wa_t \nabla \log p_t(x|y)) \\ &= (1 - w)u_t^{\text{target}}(x) + wu_t^{\text{target}}(x|y). \end{aligned}$$

We may therefore express the scaled guided vector field  $\tilde{u}_t(x|y)$  as the linear combination of the unguided vector field  $u_t^{\text{target}}(x)$  with the guided vector field  $u_t^{\text{target}}(x|y)$ . The idea might then to to train both an unguided  $u_t^{\text{target}}(x)$  (using e.g., Equation (26)) as well as a guided  $u_t^{\text{target}}(x|y)$  (using e.g., Equation (58)), and then combine them at inference time to obtain  $\tilde{u}_t(x|y)$ . ‘‘But wait!’’, you might ask, ‘‘wouldn’t we need to train two models then !?’’. It turns out that we can train both in one model: we may augment our label set with a new, additional  $\emptyset$  label that denotes **the absence of conditioning**. We can then treat  $u_t^{\text{target}}(x) = u_t^{\text{target}}(x|\emptyset)$ . With that, we do not need to train a separate model to reinforce the effect of a hypothetical classifier. This approach of training a conditional and unconditional model in one (and subsequently reinforcing the conditioning) is known as **classifier-free guidance** (CFG) [18] (see Figure 12 for an illustration).

**Remark 26** (Derivation for general probability paths)

Note that the construction

$$\tilde{u}_t(x|y) = (1 - w)u_t^{\text{target}}(x) + wu_t^{\text{target}}(x|y),$$

is equally valid for any choice probability path, not just a Gaussian one. When  $w = 1$ , it is straightforward to verify that  $\tilde{u}_t(x|y) = u_t^{\text{target}}(x|y)$ . Our derivation using Gaussian paths was simply to illustrate the intuition behind the construction, and in particular of amplifying the contribution of a hypothetical “classifier”  $\nabla \log p_t(y|x)$ .

**Training and Classifier-Free Guidance.** We must now amend the guided conditional flow matching objective from Equation (58) to account for the possibility of  $y = \emptyset$ . The challenge is that when sampling  $(z, y) \sim p_{\text{data}}$ , we will never obtain  $y = \emptyset$ . It follows that we must introduce the possibility of  $y = \emptyset$  artificially. To do so, we will define some hyperparameter  $\eta$  to be the probability that we discard the original label  $y$ , and replace it with  $\emptyset$ . We thus arrive at our **CFG conditional flow matching training objective**

$$\mathcal{L}_{\text{CFM}}^{\text{CFG}}(\theta) = \mathbb{E}_{\square} \|u_t^\theta(x|y) - u_t^{\text{target}}(x|z)\|^2 \quad (63)$$

$$\square = (z, y) \sim p_{\text{data}}(z, y), t \sim \text{Unif}[0, 1], x \sim p_t(\cdot|z), \text{replace } y = \emptyset \text{ with prob. } \eta \quad (64)$$

---

**Algorithm 5** Classifier-free guidance training for Gaussian probability path  $p_t(x|z) = \mathcal{N}(x; \alpha_t z, \beta_t^2 I_d)$

---

**Require:** Paired dataset  $(z, y) \sim p_{\text{data}}$ , neural network  $u_t^\theta$

- 1: **for** each mini-batch of data **do**
- 2:   Sample a data example  $(z, y)$  from the dataset.
- 3:   Sample a random time  $t \sim \text{Unif}[0, 1]$ .
- 4:   Sample noise  $\epsilon \sim \mathcal{N}(0, I_d)$
- 5:   Set  $x = \alpha_t z + \beta_t \epsilon$
- 6:   With probability  $p$  drop label:  $y \leftarrow \emptyset$
- 7:   Compute loss

$$\mathcal{L}(\theta) = \|u_t^\theta(x|y) - (\alpha_t z + \beta_t \epsilon)\|^2$$

- 8:   Update the model parameters  $\theta$  via gradient descent on  $\mathcal{L}(\theta)$ .
  - 9: **end for**
- 

We summarize our findings below.

**Summary 27** (Classifier-Free Guidance for Flow Models)

Given the unguided marginal vector field  $u_t^{\text{target}}(x|\emptyset)$ , the guided marginal vector field  $u_t^{\text{target}}(x|y)$ , and a **guidance scale**  $w > 1$ , we define the **classifier-free guided vector field**  $\tilde{u}_t(x|y)$  by

$$\tilde{u}_t(x|y) = (1 - w)u_t^{\text{target}}(x|\emptyset) + wu_t^{\text{target}}(x|y). \quad (65)$$

By approximating  $u_t^{\text{target}}(x|\emptyset)$  and  $u_t^{\text{target}}(x|y)$  using the same neural network, we may leverage the following

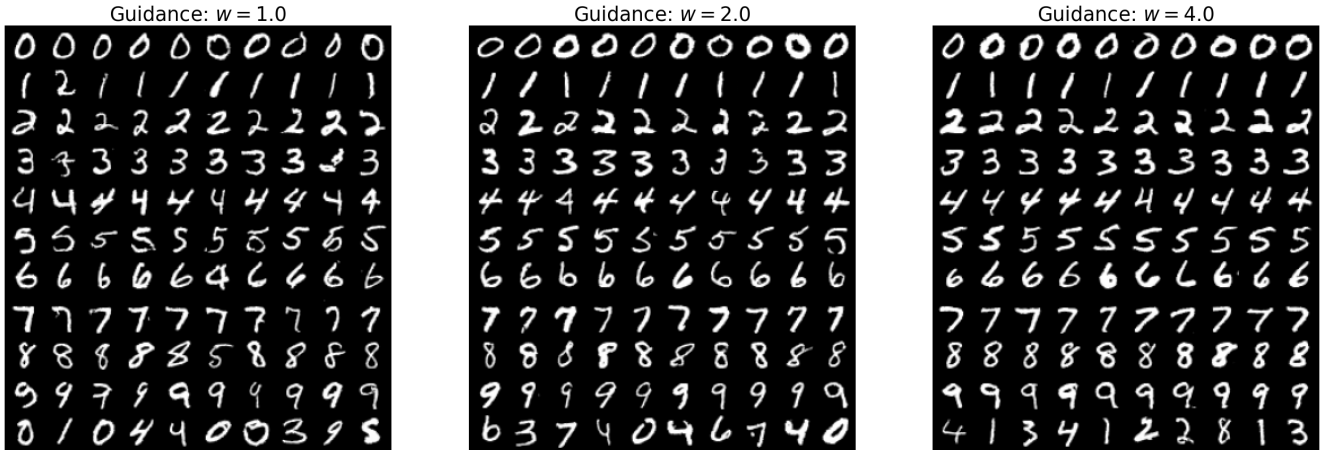


Figure 13: The effect of classifier-free guidance applied at various guidance scales for the MNIST dataset of handwritten digits. Left: Guidance scale set to  $w = 1.0$ . Middle: Guidance scale set to  $w = 2.0$ . Right: Guidance scale set to  $w = 4.0$ . You will generate a similar image yourself in the lab three!

**classifier-free guidance CFM** (CFG-CFM) objective, given by

$$\mathcal{L}_{\text{CFM}}^{\text{CFG}}(\theta) = \mathbb{E}_{\square} \|u_t^\theta(x|y) - u_t^{\text{target}}(x|z)\|^2 \quad (66)$$

$$\square = (z, y) \sim p_{\text{data}}(z, y), t \sim \text{Unif}[0, 1], x \sim p_t(\cdot|z), \text{replace } y = \emptyset \text{ with prob. } \eta \quad (67)$$

In plain English,  $\mathcal{L}_{\text{CFM}}^{\text{CFG}}$  might be approximated by

- |  |   |
|--|---|
| $(z, y) \sim p_{\text{data}}(z, y)$  | ► Sample $(z, y)$ from data distribution.                       |
| $t \sim \text{Unif}[0, 1]$   | ► Sample $t$ uniformly on $[0, 1]$ .                            |
| $x \sim p_t(x z)$  | ► Sample $x$ from the conditional probability path $p_t(x z)$ . |
| with prob. $\eta$ , $y \leftarrow \emptyset$   | ► Replace $y$ with $\emptyset$ with probability $\eta$ .        |
| $\widehat{\mathcal{L}}_{\text{CFM}}^{\text{CFG}}(\theta) = \ u_t^\theta(x y) - u_t^{\text{target}}(x z)\ ^2$ | ► Regress model against conditional vector field.               |

At inference time, for a fixed choice of  $y$ , we may sample via

- |   |   |
|---|---|
| <b>Initialization:</b> $X_0 \sim p_{\text{init}}(x)$    | ► Initialize with simple distribution (such as a Gaussian)  |
| <b>Simulation:</b> $dX_t = \tilde{u}_t^\theta(X_t y)dt$ | ► Simulate ODE from $t = 0$ to $t = 1$ .                    |
| <b>Samples:</b> $X_1$                                   | ► Goal is for $X_1$ to adhere to the guiding variable $y$ . |

Note that the distribution of  $X_1$  is not necessarily aligned with  $X_1 \sim p_{\text{data}}(\cdot|y)$  anymore if we use a weight  $w > 1$ . However, empirically, this shows better alignment with conditioning. Classifier-free guidance is therefore a **heuristic** that is predominantly justified by its excellent empirical results. In fact, almost any image or video that you see that is AI-generated relied heavily on classifier-free guidance  $w \geq 4$ . In Figure 11, we illustrate class-based classifier-free guidance on 128x128 ImageNet, as in [18]. Similarly, in Figure 13, we visualize the affect of various guidance scales  $w$  when applying classifier-free guidance to sampling from the MNIST dataset of handwritten digits.

**Remark 28** (Guidance for Diffusion Models)

It is straight-forward to extend the discussion from flow models to diffusion models. One simply replaces  $u_t^\theta(x|y)$  by  $\tilde{u}_t^\theta(x|y)$  and samples using SDEs as discussed in [Section 4](#).

---

## 6 Building Large-Scale Image or Video Generators

In the previous sections, we learned how to train a flow matching or diffusion model to sample from a distribution  $p_{\text{data}}(x|y)$ . This recipe is general and can be applied to a variety of different data types and applications. In this section, we examine in depth the particular cases of large-scale image and video generation, and including well-known models such as *FLUX 2.0*, *Stable Diffusion 3*, *Nano Banana* and *VEO-3 or Meta Movie Gen Video*. Finally, we'll apply what we've learned so far in the lab to build our own version of such models from scratch! This section is broadly arranged as follows:

1. **Neural network architectures:** We first discuss how raw conditioning input, including the time  $t$ , and guidance variable  $y_{\text{raw}}$  (i.e., a discrete class label or raw text), is converted, or **embedded** into a vector-valued form digestible by the model  $u_t^\theta(x|y)$  itself. Then we discuss popular architectural choices for  $u_t^\theta(x|y)$ , including the **U-Net** and **diffusion transformer**.
2. **Latent Space:** We discuss **variational autoencoders**, which allow for generative modeling in a lower dimensional **latent space**, thereby enabling ultra high-resolution image generation.
3. **Case Studies:** Finally, we will examine in depth the two state-of-the-art image and video models mentioned above - *Stable Diffusion* and *Meta MovieGen* - to give you a taste of how things are done at scale.

### 6.1 Neural Network Architectures

Let us first turn our attention toward the design of scalable neural network architectures for flow and diffusion models targeting image-like modalities (e.g., images and videos). Specifically, we'll explore how the task of the (guided) vector field  $u_t^\theta(x|y)$  with parameters  $\theta$  is implemented in practice. Note that the neural network must have 3 inputs: a vector  $x \in \mathbb{R}^d$ , a conditioning variable  $y \in \mathcal{Y}$ , and a time value  $t \in [0, 1]$ , as well as one output, a vector  $u_t^\theta(x|y) \in \mathbb{R}^d$ . For low-dimensional distributions (e.g. the toy distributions we have seen in previous sections), it is sufficient to parameterize  $u_t^\theta(x|y)$  as a multi-layer perceptron (MLP), otherwise known as a fully connected neural network. That is, in this simple setting, a forward pass through  $u_t^\theta(x|y)$  would involve concatenating our input  $x$ ,  $y$ , and  $t$ , and passing them through an MLP. However, for complex, high-dimensional distributions, such as those over images, videos, and proteins, an MLP will likely not suffice, and it is common to use special, application-specific architectures. For the remainder of this subsection, we will consider the case of **images** (and by extension, videos). First, we'll consider how the raw conditioning information - the time  $t$  and the conditioning variable  $y$  - are **embedded** into a vector-valued form digestible by the actual model. Second, we'll consider two common architectural choices for such a model: the **U-Net** [38, 17, 22, 11], and the **diffusion transformer** (DiT) [12, 30, 28].

#### 6.1.1 Embedding the Conditioning Variables

**Embedding Time.** For simple toy models, concatenating the raw value of  $t$  to the input is sufficient to train a reasonably performant network. In practice, the scalar time is often embedded in a higher dimensional space using **Fourier features**, allowing the model to more faithfully capture high-frequency time dependence [46]. Explicitly,

the featurization is given by

$$\text{TimeEmb}(t) = \sqrt{\frac{2}{d}} \left[ \cos(2\pi w_1 t) \quad \cdots \quad \cos(2\pi w_{d/2} t) \quad \sin(2\pi w_1 t) \quad \cdots \quad \sin(2\pi w_{d/2} t) \right]^T, \quad (68)$$

where the frequencies  $w_i$  are set in the following way

$$w_i = w_{\min} \left( \frac{w_{\max}}{w_{\min}} \right)^{\frac{i-1}{d/2-1}}, \quad i = 1, \dots, d/2. \quad (69)$$

This choice of TimeEmb is a standard choice but this exact form is not strictly necessary. Rather, the above is simply a convenient way of obtaining a normed embedding of dimension  $d$ , i.e.  $\|\text{TimeEmb}(t)\| = 1$  (because  $\sin^2 + \cos^2 = 1$ ).

**Embedding Class Labels.** When  $y_{\text{raw}} \in \mathcal{Y} \triangleq \{0, \dots, N\}$  is just a class label, then it is often easiest to simply learn a separate embedding vector for each of the  $N + 1$  possible values of  $y_{\text{raw}}$ , and set  $y$  to this embedding vector. One would consider the parameters of these embeddings to be included in the parameters of  $u_t^\theta(x|y)$ , and would therefore learn these during training.

**Embedding Textual Input** When  $y_{\text{raw}}$  is a text-prompt, the situation is more complex, and approaches largely rely on frozen, pre-trained models. Such models are trained to embed a discrete text input into a continuous vector that captures the relevant information. One such model is known as **CLIP** (Contrastive Language-Image Pre-training). CLIP is trained to learn a shared embedding space for both images and text-prompts, using a training loss designed to encourage image embeddings to be close to their corresponding prompts, while being farther from the embeddings of other images and prompts [34]. We might therefore take  $y = \text{CLIP}(y_{\text{raw}}) \in \mathbb{R}^{d_{\text{CLIP}}}$  to be the embedding produced by a frozen, pre-trained CLIP model. In certain cases, it may be undesirable to compress the entire sequence into a single representation. In this case, one might additionally consider embedding the prompt using a pre-trained transformer so as to obtain a sequence of embeddings. It is also common to combine multiple such pretrained embeddings when conditioning so as to simultaneously reap the benefits of each model [14, 33]. For our purposes, one can simply assume that after applying such a model the prompt embedding has shape

$$\text{PromptEmbed}(y_{\text{raw}}) \in \mathbb{R}^{S \times k}$$

### 6.1.2 Diffusion Transformers

Before we dive into the specifics of these architectures, let us recall from the introduction that an image is simply a vector  $x \in \mathbb{R}^{C_{\text{image}} \times H \times W}$ . Here  $C_{\text{image}}$  denotes the number of **channels** (an RGB image typically would have  $C_{\text{input}} = 3$  color channels), and  $H$  and  $W$  respectively denote the **height** and **width** of the image in pixels. One particularly prominent architectural class are so-called **diffusion transformers** (DiTs), and their variants, which use the **attention** mechanism to construct the network [49, 30, 28]. There are different flavors of diffusion transformers. We explain here a generic design, and note though that specific instantiations of DiTs might differ depending on model and application. For the remainder of this section, we will use  $d$  to denote the hidden dimension,  $L$  to denote the number of transformer layers, and  $h$  to denote the number of heads per layer. Diffusion transformers are based on **vision transformers** (ViTs), whose main idea is essentially to divide up an image into patches, embed

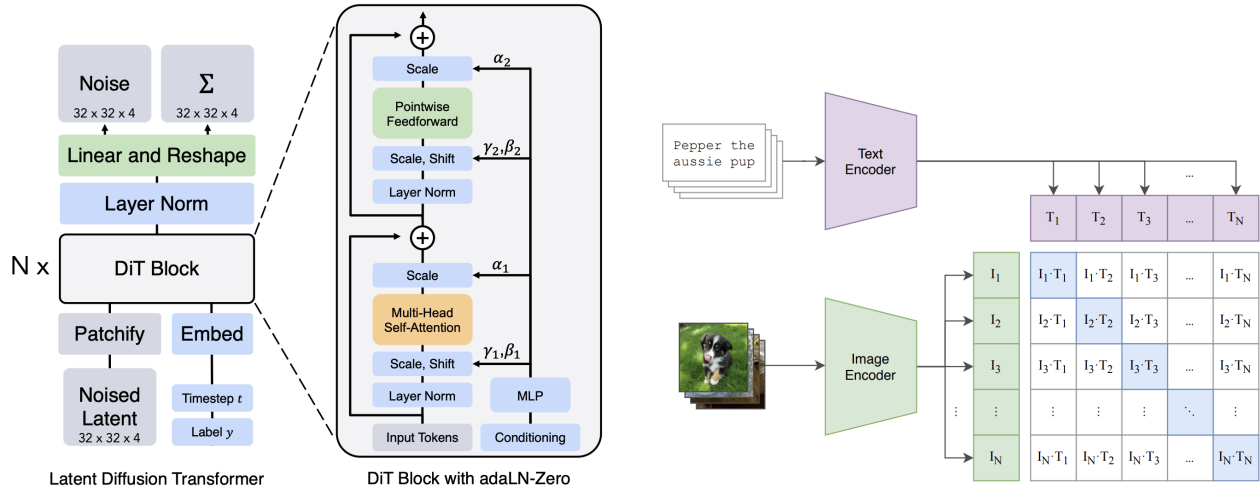


Figure 14: Left: An overview of the diffusion transformer architecture, taken from [30]. Right: A schematic of the contrastive CLIP loss, in which a shared image-text embedding space is learned, taken from [34].

the patches to obtain a sequence of tokens, and process the resulting tokens via standard attention [13]. A final depatchification operation is applied at the end to recover an image of the correct shape. The initial patchification operation is simply a restructuring of the image tensor  $x \in \mathbb{R}^{C \times H \times W}$ :

$$\text{Patchify}(x) \in \mathbb{R}^{N \times C'}$$

where  $C' = CP^2$ ,  $N = (H/P) \cdot (W/P)$  for  $P$  the patch size. Next, we apply a linear transformation to the output giving us the final patch embedding

$$\text{PatchEmb}(x) = \text{Patchify}(x)W \in \mathbb{R}^{N \times d}$$

where  $W \in \mathbb{R}^{C' \times d}$  is a learnable weight matrix. The inputs to the diffusion transformer are then the time embedding, the prompt embedding, and the patchified image tensor given by (see Section 6.1.1):

$$\begin{aligned} \tilde{t} &= \text{TimeEmb}(t) \in \mathbb{R}^d \\ \tilde{y} &= \text{PromptEmb}(y) \in \mathbb{R}^{S \times d} \\ \tilde{x}_0 &= \text{PatchEmb}(x) \in \mathbb{R}^{N \times d} \end{aligned}$$

Note that all elements have now the desired hidden dimension of the transformer. The diffusion transformer then iteratively updates  $\tilde{z}_i$  via for  $i = 0, \dots, L - 1$  via transformer layers in a **DiTBlock** (see Remark 29 for details):

$$\tilde{x}_{i+1} = \text{DiTBlock}(\tilde{x}_i, \tilde{t}, \tilde{y}) \in \mathbb{R}^{N \times d} \quad (i = 0, \dots, L - 1). \quad (70)$$

where  $N$  is the number of layers. Finally, a final operation applies a depatchification operation which maps the DiT output back to the desired output shape:

$$u = \text{Depatchify}(\tilde{x}_N \tilde{W}) \in \mathbb{R}^{C \times H \times W},$$

where  $\tilde{W} \in \mathbb{R}^{d \times C'}$ . The final tensor  $u$  then serves as the output of the model and the predicted velocity  $u_t^\theta(x|y)$ .

**Remark 29** (DiT Block)

For completeness, we present a brief mathematical description of a single DiT layer. While we attempt to include enough detail to allow for a general understanding of the DiT model family, we remind the reader that these choose to emphasize key algorithmic choices rather than architectural details. Now, let  $x \in \mathbb{R}^{N \times d}$  denote the current sequence of patch tokens (here  $x = \tilde{x}_i$ ), and let  $y \in \mathbb{R}^{S \times d}$  denote the embedded guiding variable (here  $y = \tilde{y}$ ). Then, a typical DiT block updates  $x$  using (i) self-attention on patches, (ii) cross-attention to the prompt, and (iii) time conditioning via adaptive normalization (AdaLN).

**Scaled Dot Product Attention.** Given queries  $Q \in \mathbb{R}^{N \times d_h}$ , keys  $K \in \mathbb{R}^{M \times d_h}$ , and values  $V \in \mathbb{R}^{M \times d_h}$ ,

$$\text{Attn}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_h}}\right) V \in \mathbb{R}^{N \times d_h},$$

where the softmax is applied row-wise.

**Multi-Head Attention.** Let  $h$  denote the number of heads and  $d_h = \frac{d}{h}$  the per-head dimension. For each head  $h \in \{1, \dots, n_{\text{heads}}\}$ , learn projection matrices  $W_Q^{(h)}, W_K^{(h)}, W_V^{(h)} \in \mathbb{R}^{k \times d_h}$ . Define

$$\text{head}_h(x, z) = \text{Attn}(xW_Q^{(h)}, zW_K^{(h)}, zW_V^{(h)}),$$

where the *source sequence*  $z$  is either

$$z = x \quad (\text{self-attention on patches}), \quad z = y \quad (\text{cross-attention to the prompt}).$$

Concatenate heads and apply an output projection  $W_O \in \mathbb{R}^{d \times d}$ :

$$\text{MultiHeadattention}(x, z) = \text{Concat}(\text{head}_1(x, z), \dots, \text{head}_h(x, z)) W_O \in \mathbb{R}^{N \times d}.$$

**Time Conditioning via Adaptive Normalization.** Let  $\tilde{t} \in \mathbb{R}^d$  be the timestep embedding. A standard choice in DiTs is to use  $\tilde{t}$  to produce per-channel scale/shift parameters that modulate normalized activations [31]. Concretely, let  $g : \mathbb{R}^d \rightarrow \mathbb{R}^{2d}$  be an MLP and set

$$(\gamma, \beta) = g(\tilde{t}),$$

where  $\gamma, \beta \in \mathbb{R}^d$  (or, depending on the implementation, separate  $(\gamma, \beta)$  pairs for different sub-layers such as attention and MLP). Given a token matrix  $x \in \mathbb{R}^{N \times d}$  and a normalization operator  $\text{Norm}(\cdot)$  (e.g. LayerNorm),

define the modulated normalization

$$\text{AdaNorm}_{\tilde{t}}(x) = (1 + \gamma) \odot \text{Norm}(H) + \beta,$$

where  $\odot$  denotes elementwise multiplication with broadcasting over the token dimension.

**Putting It Together.** The combined operation, and thus the DitBlock, is given by.

$$\begin{aligned} x &\leftarrow x + g_{\text{self}}(\tilde{t}) \odot \text{MultiHeadattention}(\text{AdaNorm}_{\tilde{t}}(x), \text{AdaNorm}_{\tilde{t}}(x)) \\ x &\leftarrow x + g_{\text{cross}}(\tilde{t}) \text{MultiHeadattention}(\text{AdaNorm}_{\tilde{t}}(x), y) \\ x &\leftarrow x + g_{\text{MLP}}(\tilde{t}) \text{MLP}(\text{AdaNorm}_{\tilde{t}}(x)), \end{aligned}$$

where the MLP is a position-wise feed-forward network, and the  $g_{\dots}$  are learnable gating parameters. The output  $x \in \mathbb{R}^{N \times d}$  becomes the next-layer patch-token sequence (in our notation,  $\tilde{x}_{i+1}$ ). Finally, we note that class-conditioned DiT's, such as the one implemented in the lab, are typically simpler and eschew the cross attention layer in favor of a time and class-based AdaNorm conditioning.

### 6.1.3 U-Net

The **U-Net** architecture [38] is an alternative architecture to the DiT architecture and is a specific type of convolutional neural network. Originally designed for image segmentation, its crucial feature is that both its input and its output have the shape of images (possibly with a different number of channels). This makes it ideal for parameterizing a vector field  $x \mapsto u_t^\theta(x|y)$ , as for fixed  $y, t$  its input has the shape of an image and its output does, too. Accordingly, U-Nets have seen widespread use across much of the early literature on diffusion models [17, 22, 11]. A U-Net consists of a series of **encoders**  $\mathcal{E}_i$ , and a corresponding sequence of **decoders**  $\mathcal{D}_i$ , along with a latent processing block in between, which we shall refer to as a **midcoder**.<sup>3</sup> For sake of example, let us walk through the path taken by an image  $x_t \in \mathbb{R}^{3 \times 256 \times 256}$  (we have taken  $(C_{\text{input}}, H, W) = (3, 256, 256)$ ) as it is processed by the U-Net:

$$\begin{aligned} x_t^{\text{input}} &\in \mathbb{R}^{3 \times 256 \times 256} && \blacktriangleright \text{Input to the U-Net.} \\ x_t^{\text{latent}} &= \mathcal{E}(x_t^{\text{input}}) \in \mathbb{R}^{512 \times 32 \times 32} && \blacktriangleright \text{Pass through encoders to obtain latent.} \\ x_t^{\text{latent}} &= \mathcal{M}(x_t^{\text{latent}}) \in \mathbb{R}^{512 \times 32 \times 32} && \blacktriangleright \text{Pass latent through midcoder.} \\ x_t^{\text{output}} &= \mathcal{D}(x_t^{\text{latent}}) \in \mathbb{R}^{3 \times 256 \times 256} && \blacktriangleright \text{Pass through decoders to obtain output.} \end{aligned}$$

Notice that as the input passes through the encoders, the number of channels in its representation increases, while the height and width of the images are decreased. Both the encoder and the decoder usually consist of a series of convolutional layers (with activation functions, pooling operations, etc. in between). Not shown above are two points: First, the input  $x_t^{\text{input}} \in \mathbb{R}^{3 \times 256 \times 256}$  is often fed into an initial pre-encoding block to increase the number of channels before being fed into the first encoder block. Second, the encoders and decoders are often connected by **residual connections**. The complete picture is shown in Figure 15. At a high level, most U-Nets involve some

<sup>3</sup>Midcoder is a completely non-standard term used here to refer to the bottom-most part of the U-Net stack, and in analogy with the encoder and decoder.

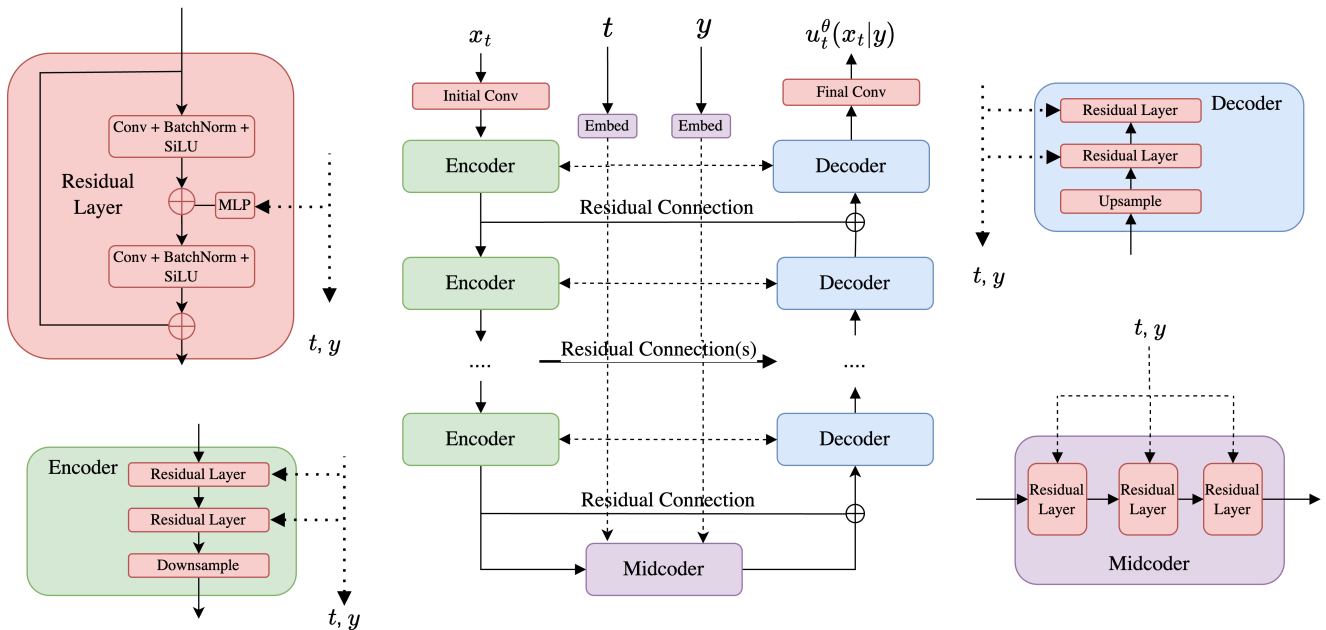


Figure 15: A simplified U-Net architecture (an architecture like this was used in lab 03 of the 2025 version of this course).

variant of what is described above. However, certain of the design choices described above may well differ from various implementations in practice. In particular, we opt above for a purely-convolutional architecture whereas it is common to include attention layers as well throughout the encoders and decoders. The U-Net derives its name from the “U”-like shape formed by its encoders and decoders (see Figure 15).

## 6.2 Working in Latent Space: (Variational) Autoencoders

Thus far, we have operated in the data space  $\mathbb{R}^d$ . However, the cost of modeling directly within such a space quickly becomes prohibitively expensive as one scales to increasingly higher resolution images. For example, a  $1024 \times 1024$  image with three RGB color channels corresponds to a total dimension of  $d = H \cdot W \cdot 3 \approx 3 * 10^6!$  Note that the dimension increases further for videos as everything scales with the number of frames  $T$ . As you can imagine, training over such a space quickly becomes infeasible. Unlike image classification, whose low-dimensional outputs allow for narrowing convolutional stacks, our flow-based modeling approach requires that our output  $u_t^\theta(x) \in \mathbb{R}^d$  be just as large as our input. The important question thus becomes: **How can we model high-dimensional images within a reasonable memory and computation budget?**

### 6.2.1 Standard Autoencoders

A natural answer to this question lies in **compression**: perhaps the actual space of images, for example, lies near a much lower-dimensional manifold of the high dimensional image space. More concretely, we might consider an **encoder**  $\mu_\phi : \mathbb{R}^d \rightarrow \mathbb{R}^k$ , together with some **decoder**  $\mu_\theta : \mathbb{R}^k \rightarrow \mathbb{R}^d$ , which together map raw images  $x \in \mathbb{R}^d$  to and from latents  $z \in \mathbb{R}^k$ , respectively. The dimension  $k$  is typically chosen to be much smaller than  $d$ . For images, in

which, for example,  $d = 3 \times 1024 \times 1024$ , it is not uncommon to downsample to obtain e.g.,  $k = 3 \times \frac{1024}{16} \times \frac{1024}{16}$ . Together,  $\mu_\phi$  and  $\mu_\theta$  are referred to as an **autoencoder**. Ideally,  $\mu_\phi$  and  $\mu_\theta$  are chosen so as to achieve high reconstruction quality, or in other words, so that  $\mu_\theta(\mu_\phi(x))$  resembles  $x$  on average. Accordingly, autoencoders are usually trained with the **reconstruction loss**

$$\mathcal{L}_{\text{Recon}}(\phi, \theta) = \mathbb{E}_{x \sim p_{\text{data}}} [\|\mu_\theta(\mu_\phi(x)) - x\|^2].$$

which measures the squared error between the original data point  $x$  and the reconstructed one  $\mu_\theta(\mu_\phi(x))$ .

**Amenability to Generative Modeling.** Unfortunately, the reconstruction loss above is not enough to train a “good” autoencoder. Recall that our eventual goal is to train a generative model in the latent space, and targeting the latent distribution  $p_{\text{latent}}(z)$  given by  $z = \mu_\phi(x), x \sim p_{\text{data}}$ . A generative model for  $p_{\text{data}}(x)$  is then realized by passing the output of our latent generative model through the decoder  $\mu_\theta$ . A subtle issue arises with autoencoders as we have currently formulated them in that we have little to no control over  $p_{\text{latent}}(z)$ , and thus essentially no guarantee that  $p_{\text{latent}}(z)$  is even well-behaved enough to be amenable to training such a generative model (i.e., nice, simple, Gaussian-like). While transforming our data in latent space might have compressed it, we might have transformed the data distribution  $p_{\text{data}}$  into a very hard-to-learn distribution  $p_{\text{latent}}$ . Therefore, the question is: how can we make sure that the latent distribution  $p_{\text{latent}}$  is still well-behaved and easy-to-learn? To allow for more explicit regularization of the latent distribution, we will now recast the concept of autoencoder in a more general probabilistic framework leading to the concept of a variational autoencoder.

### 6.2.2 Variational Autoencoders

A **variational autoencoder** (VAE) is obtained from our (deterministic) standard autoencoder formulation by relaxing the constraint that the encoder and decoder are deterministic functions. In particular, let us consider an encoder  $q_\phi(z|x)$  with parameters  $\phi$ , and a decoder  $p_\theta(x|z)$  with parameters  $\theta$ . The most common choice is to take

$$q_\phi(z|x) = \mathcal{N}(z; \mu_\phi(x), \text{diag}(\sigma_\phi^2(x))), \quad p_\theta(x|z) = \mathcal{N}(x; \mu_\theta(z), \sigma_\theta^2(z)I_d) \quad (71)$$

where  $\mu_\phi(x) \in \mathbb{R}^k$ ,  $\sigma_\phi^2(x) \in \mathbb{R}_{\geq 0}^k$ ,  $\mu_\theta(z) \in \mathbb{R}^d$ , and  $\sigma_\theta^2(z) \in \mathbb{R}_{\geq 0}$  are parameterized as neural networks and  $\text{diag}$  denotes the diagonal matrix. To encode or decode a variable, we sample

$$\begin{aligned} z &\sim q_\phi(\cdot|x) && \text{(encode)} \\ x &\sim p_\theta(\cdot|z) && \text{(decode)} \end{aligned}$$

Finally, we note that when  $\sigma_\phi(x) = 0$  and  $\sigma_\theta(z) = 0$  always, we recover a standard autoencoder. Let us examine what a reconstruction loss looks like. A natural objective is the following:

$$\mathcal{L}_{\text{VAE-Recon}}(\phi, \theta) = -\mathbb{E}_{x \sim p_{\text{data}}(x), z \sim q_\phi(\cdot|x)} [\log p_\theta(x|z)] \quad (72)$$

Note the two changes: Instead of a deterministic encoding, we now sample  $z \sim q_\phi(z|x)$ . Further, we now take the negative log-likelihood of  $x$  under decoding, i.e. the loss effectively asks: how likely would our original data point  $x$  be if we encoded and decoded it - and we take all possible decodings/encodings into account as things have become

random now. For the Gaussian case, this reconstruction loss becomes:

$$\mathcal{L}_{\text{VAE-Recon}}(\phi, \theta) = \mathbb{E}_{x \sim p_{\text{data}}(x), z \sim q_{\phi}(z|x)} \left[ \frac{1}{2\sigma_{\theta}^2(z)} \|x - \mu_{\theta}(z)\|^2 + \frac{d}{2} \log \sigma_{\theta}^2(z) \right] + \text{const} \quad (73)$$

where we used the density of the normal distribution (see Equation (97)). Hence, the VAE reconstruction loss is not that different from the standard AE reconstruction loss, we simply have to take into account all possible encodings  $z \sim q_{\phi}(\cdot|x)$ . The second term depending on the decoder variance controls the tradeoff between reconstruction accuracy and predictive uncertainty. Many implementations, including that in the lab, fix  $\sigma_{\phi}(x)$  and  $\sigma_{\theta}(z)$  to learned scalar constants (that is, independent of  $x$  and  $z$ , respectively), thereby avoiding pathological behavior and numerical stability when learning variances. Therefore, the VAE reconstruction loss in this case then becomes basically the standard autoencoder reconstruction loss up to stochasticity in the encoding and constants:

$$\mathcal{L}_{\text{VAE-Recon}}(\phi, \theta) = \mathbb{E}_{x \sim p_{\text{data}}(x), z \sim q_{\phi}(z|x)} \left[ \frac{1}{2\sigma_{\theta}^2} \|x - \mu_{\theta}(z)\|^2 \right] + \text{const} \quad (74)$$

Let us now revisit our goal: We want to create an encoding of our data distribution  $p_{\text{data}}(x)$  such that after mapping it into a latent space, the distribution becomes “nice” or easy-to-learn. Toward this end, let us now introduce a **prior distribution**  $p_{\text{prior}}(z)$  over latents  $z$ . For our purposes, we will take  $p_{\text{prior}} = \mathcal{N}(0, I_k)$  to be an isotropic Gaussian. This choice of prior distribution  $p_{\text{prior}}$  effectively represents the “ideal” case for what the latent distribution should look like. A normal distribution would be very easy to learn, and would therefore satisfy our goal of obtaining a “trainable” latent distribution. The big idea is thus to regularize our encoder so as to ensure that the encoded data distribution is as close as possible to the  $p_{\text{prior}}$ , which we accomplish via the auxiliary loss

$$\mathcal{L}_{\text{VAE-Prior}}(\phi) = \mathbb{E}_{x \sim p_{\text{data}}(x)} [D_{\text{KL}}(q_{\phi}(\cdot|x) \parallel p_{\text{prior}})], \quad (75)$$

and where  $D_{\text{KL}}$  is the **Kullback-Leibler (KL) divergence**. The KL-divergence is a fundamental way of measuring how different two probability distributions are. Explaining it in detail would go beyond the scope of this work but we give a brief background in Remark 30 as a reminder for the reader. The loss  $\mathcal{L}_{\text{VAE-Prior}}$  defined here now is very intuitive: We want that the encoding distributions looks like a Gaussian distribution for any data point  $x$ . If we do this for all  $x$ , it is natural to expect that then our latent distribution will look a Gaussian as well.

**Remark 30** (Background on KL-divergence)

For two probability densities  $q, p$ , the **Kullback-Leibler divergence** (KL-divergence) is defined as

$$D_{\text{KL}}(q(x) \parallel p(x)) = \int q(x) \log \frac{q(x)}{p(x)} = \mathbb{E}_{X \sim q} \left[ \log \frac{q(X)}{p(X)} \right].$$

The KL divergence is a standard measure of dissimilarity between distributions. In particular, the KL divergence satisfies the following useful properties:

$$D_{\text{KL}}(q(x) \parallel p(x)) \geq 0, \quad (76)$$

$$D_{\text{KL}}(q(x) \parallel p(x)) = 0 \iff q = p. \quad (77)$$

i.e. it is always non-negative and it is zero if and only the two probability distributions coincide.

To define the loss function for a variational autoencoder, we can now combine both the reconstruction and the prior loss with a parameter weight  $\beta \geq 0$  to **VAE training objective** given by

$$\mathcal{L}_{\text{VAE}}(\phi, \theta) = \mathcal{L}_{\text{VAE-Recon}}(\phi, \theta) + \beta \mathcal{L}_{\text{VAE-Prior}}(\phi) \quad (78)$$

$$= -\mathbb{E}_{x \sim p_{\text{data}}(x), z \sim q_{\phi}(z|x)} [\log p_{\theta}(x | z)] + \beta \mathbb{E}_{x \sim p_{\text{data}}(x)} [D_{\text{KL}}(q_{\phi}(\cdot|x) || p_{\text{prior}})] \quad (79)$$

where the first summand enforces that latent variables can be efficiently decoded back to data and the second summand enforces that our latent distribution is close to being a Gaussian. The parameter  $\beta$  controls the strength of each. To make this loss more specific, let us derive the KL divergence for the Gaussian case:

**Example 31** (KL Divergence Between Isotropic Gaussians)

Let  $q(x) = \mathcal{N}(x; \mu_q, \text{diag}(\sigma_q^2))$  and  $p(x) = \mathcal{N}(x; \mu_p, \text{diag}(\sigma_p^2))$  be Gaussians with diagonal covariance matrices, with  $\sigma_q, \sigma_p \in \mathbb{R}_{\geq 0}^d$ , and where  $x \in \mathbb{R}^d$ . Then

$$D_{\text{KL}}(q || p) = \frac{1}{2} \left( \mathcal{K} \left( \frac{\sigma_q^2}{\sigma_p^2} \right) + \frac{\|\mu_q - \mu_p\|^2}{\sigma_p^2} \right), \quad \text{where } \mathcal{K}(\alpha) = \sum_{i=1}^d \alpha_i - \log \alpha_i - 1. \quad (80)$$

The expression above is intuitive: If the mean and variances coincide, that then  $D_{\text{KL}}(q || p) = 0$ . Further, it increases with the squared error  $\|\mu_q - \mu_p\|^2$  between the mean vectors. Finally, the function  $\mathcal{K}(\alpha)$  has a unique minimum at  $\alpha = 1$  so that  $D_{\text{KL}}(q || p)$  is minimized when  $\sigma_q = \sigma_p$ .

*Proof.* We do the proof for  $d = 1$  (proof is analogous for  $d > 1$  by summing up each dimension). Given the density of the normal distribution, we know that (see Equation (97)):

$$\log q(x) = -\frac{1}{2} \log(2\pi\sigma_q^2) - \frac{1}{2\sigma_q^2} \|x - \mu_q\|^2, \quad \log p(x) = -\frac{1}{2} \log(2\pi\sigma_p^2) - \frac{1}{2\sigma_p^2} \|x - \mu_p\|^2$$

Then

$$D_{\text{KL}}(q||p) = \mathbb{E}_{x \sim q} [\log q(x) - \log p(x)] = \frac{1}{2} \log \frac{\sigma_p^2}{\sigma_q^2} + \frac{1}{2\sigma_p^2} \mathbb{E}_q [\|x - \mu_p\|^2] - \frac{1}{2\sigma_q^2} \mathbb{E}_q [\|x - \mu_q\|^2]. \quad (81)$$

For  $x \sim \mathcal{N}(\mu_q, \sigma_q^2 I)$  we have

$$\mathbb{E}_q [\|x - \mu_q\|^2] = \text{tr}(\sigma_q^2 I) = \sigma_q^2.$$

Combining this with the fact that  $x - \mu_p = (x - \mu_q) + (\mu_q - \mu_p)$ , and  $\mathbb{E}_q[x - \mu_q] = 0$ , we obtain

$$\mathbb{E}_q [\|x - \mu_p\|^2] = \mathbb{E}_q [\|x - \mu_q\|^2] + \|\mu_q - \mu_p\|^2 = \sigma_q^2 + \|\mu_q - \mu_p\|^2.$$

Plugging these into Equation (81) yields (80). □

Let us now assume a Gaussian shape of the encoder. Then we obtain:

$$\mathcal{L}_{\text{VAE-Prior}}(\phi) = \mathbb{E}_{x \sim p_{\text{data}}(x)} [D_{\text{KL}}(q_{\phi}(\cdot|x) || \mathcal{N}(0, I_k))] = \mathbb{E} \left[ \frac{1}{2} \mathcal{K}(\sigma_{\phi}^2(x)) + \frac{1}{2} \|\mu_{\phi}(x)\|^2 \right] \quad (82)$$

This loss is intuitive: The mean  $\mu_{\phi}(x)$  is penalized for being different from zero and the variance penalized for

being different from 1. As a total loss for the VAE, we obtain

$$\begin{aligned}
\mathcal{L}_{\text{VAE}}(\phi, \theta) &= \mathcal{L}_{\text{VAE-Recon}}(\phi, \theta) + \beta \mathcal{L}_{\text{VAE-Prior}}(\phi) \\
&= \mathbb{E}_{x \sim p_{\text{data}}(x), z \sim q_{\phi}(z|x)} \left[ \underbrace{\frac{1}{2\sigma_{\theta}^2(z)} \|x - \mu_{\theta}(z)\|^2}_{\text{recon. error}} + \underbrace{\frac{d}{2} \log \sigma_{\theta}^2(z)}_{\text{decoder confidence}} + \underbrace{\frac{\beta}{2} \mathcal{K}(\sigma_{\phi}^2(x))}_{\text{make latent variance=1}} + \underbrace{\frac{\beta}{2} \|\mu_{\phi}(x)\|^2}_{\text{make latent mean=0}} \right] \quad (83)
\end{aligned}$$

The four terms of the above loss function are very intuitive: The first term is simply a reconstruction error. The second error describes the decoder’s uncertainty: smaller variance makes the decoder more “confident” but also penalizes reconstruction errors more strongly. Further, we want to make the latent variance 1 and the mean to be 0 - to enforce that the distribution in latent is close to being Gaussian.

**Training a VAE.** It remains to discuss how we would minimize the VAE loss  $\mathcal{L}_{\text{VAE}}(\phi, \theta)$ . The problem with the loss is that so far, the distribution we take the expected value over ( $q_{\phi}(z|x)$ ) still depends on the parameter  $\phi$ . However, we can apply the so-called **reparameterization trick** to rewrite it. Specifically, for

$$q_{\phi}(z|x) = \mathcal{N}(z; \mu_{\phi}(x), \sigma_{\phi}^2(x)I_k)$$

we can obtain samples via

$$\epsilon \sim \mathcal{N}(0, I_k), \quad z = \mu_{\phi}(x) + \sigma_{\phi}(x)\epsilon \quad \Rightarrow \quad z \sim q_{\phi}(\cdot|x)$$

Note that in this equation, the only source of noise/stochasticity is from  $\epsilon$  whose distribution is independent of  $\phi$ . Therefore, we can rewrite the loss as:

$$\mathcal{L}_{\text{VAE}}(\phi, \theta) = \mathbb{E}_{x \sim p_{\text{data}}(x), \epsilon \sim \mathcal{N}(0, I_k)} \left[ \frac{1}{2\sigma_{\theta}^2(z)} \|x - \mu_{\theta}(\mu_{\phi}(x) + \sigma_{\phi}(x)\epsilon)\|^2 + \frac{d}{2} \log \sigma_{\theta}^2(z) + \frac{\beta}{2} \mathcal{K}(\sigma_{\phi}^2(x)) + \frac{\beta}{2} \|\mu_{\phi}(x)\|^2 \right]$$

After reparameterization, the randomness comes only from  $\epsilon \sim \mathcal{N}(0, I_k)$ , whose distribution does not depend on  $\phi$ . Therefore, we can minimize this loss with the standard tools of deep learning. To simplify things even further, we can set  $\sigma_{\theta}^2(z) = \sigma^2$  constant everywhere again and obtain:

$$\mathcal{L}_{\text{VAE}}(\phi, \theta) = \mathbb{E}_{x \sim p_{\text{data}}(x), \epsilon \sim \mathcal{N}(0, I_k)} \left[ \frac{1}{2\sigma^2} \|x - \mu_{\theta}(\mu_{\phi}(x) + \sigma_{\phi}(x)\epsilon)\|^2 + \frac{\beta}{2} \mathcal{K}(\sigma_{\phi}^2(x)) + \frac{\beta}{2} \|\mu_{\phi}(x)\|^2 \right]$$

In [Algorithm 6](#), we summarize the training procedure of the VAE.

**Practical remarks.** The construction we developed here show the principles of autoencoder design. Of course, in practice, people might add more loss terms or other constraints. Therefore, we finally add a practical remarks about autoencoders:

1. **Choosing  $\beta$  (and KL warm-up).** Large  $\beta$  enforces latents closer to the prior but can hurt reconstructions and may trigger *posterior collapse* (the encoder ignores  $x$  and outputs  $q_{\phi}(z|x) \approx \mathcal{N}(0, I_k)$ ). A common stabilization is *KL warm-up*: start with  $\beta = 0$  and gradually increase it to a target value over the first epochs. However, in all modern autoencoders, the  $\beta$  value is very small, i.e.  $\beta \ll 1$ .

---

**Algorithm 6**  $\beta$ -VAE Training Procedure (Gaussian decoder with fixed variance  $p_\theta(x|z) = \mathcal{N}(x; \mu_\theta(z), \tilde{\sigma}^2 I_d)$ )

---

**Require:** Dataset of samples  $x \sim p_{\text{data}}$ , encoder networks  $(\mu_\phi(x), \log \sigma_\phi^2(x))$ , decoder network  $\mu_\theta(z)$ , latent dim  $k$ , constants  $\beta \geq 0, \sigma^2 > 0$

- 1: **for** each mini-batch  $\{x_i\}_{i=1}^B$  **do**
- 2:   Encode each  $x_i$ :  $\mu_i \leftarrow \mu_\phi(x_i), \log \sigma_i^2 \leftarrow \log \sigma_\phi^2(x_i)$
- 3:   Sample noise  $\epsilon_i \sim \mathcal{N}(0, I_k)$
- 4:   Reparameterize:  $z_i \leftarrow \mu_i + \sigma_i \odot \epsilon_i$  (where  $\sigma_i = \exp(\frac{1}{2} \log \sigma_i^2)$ )
- 5:   Decode mean:  $\hat{x}_i \leftarrow \mu_\theta(z_i)$
- 6:   Reconstruction loss:

$$\mathcal{L}_{\text{recon}} \leftarrow \frac{1}{B} \sum_{i=1}^B \frac{1}{2\tilde{\sigma}^2} \|x_i - \hat{x}_i\|^2$$

- 7:   KL loss to the prior  $p_{\text{prior}}(z) = \mathcal{N}(0, I_k)$ :

$$\mathcal{L}_{\text{KL}} \leftarrow \frac{1}{B} \sum_{i=1}^B \frac{1}{2} \sum_{j=1}^k (\mu_{i,j}^2 + \sigma_{i,j}^2 - \log \sigma_{i,j}^2 - 1)$$

- 8:   Total loss:  $\mathcal{L} \leftarrow \mathcal{L}_{\text{recon}} + \beta \mathcal{L}_{\text{KL}}$
  - 9:   Update  $(\phi, \theta) \leftarrow \text{grad\_update}(\mathcal{L})$
  - 10: **end for**
- 

2. **Decoder variance.** Learning a Gaussian decoder variance  $\sigma_\theta^2$  can be numerically delicate and may lead to degenerate solutions unless regularized. For stability, many implementations fix  $p_\theta(x|z) = \mathcal{N}(x; \mu_\theta(z), \sigma^2 I_d)$  with constant  $\sigma^2$ , which makes the reconstruction term proportional to mean-squared error (up to constants).
3. **Reconstruction losses beyond pixel MSE.** For images, a pixelwise Gaussian likelihood (mean squared error) often yields overly smooth reconstructions. In practice, people add *perceptual losses* (feature-space losses using a pretrained network) to improve sharpness and semantic fidelity.
4. **Adversarial and hybrid objectives.** To further improve visual realism, one can combine the VAE objective with an *adversarial loss* (VAE-GAN style), using a discriminator on decoded samples. This typically sharpens outputs but introduces additional optimization instability and extra hyperparameters.

**Remark 32** (Working in Latent Space)

To train a latent generative model, we simply follow the existing training recipe, but work directly in the **latent space**. At training time, we draw samples from  $q_\phi(z|x)$  with  $x \sim p_{\text{data}}$ , and at inference time, we sample  $z$  from the latent diffusion or flow model, and then decode using  $x = \mu_{\text{mean}}(z)$  (note that we take the mean rather than a random sample to avoid noise-induced artifacts). Intuitively, a well-trained autoencoder can be thought of as filtering out high-frequency or otherwise semantically meaningless details, allowing the generative model to “focus” on important, perceptually relevant features [36]. At the time of the writing of this document, nearly all state-of-the-art approaches to image and video generation follow the so-called **latent diffusion** paradigm involving training a flow or diffusion model within the latent space of an autoencoder [36, 48]. However, it is important to note: one also needs to train the autoencoder before training the diffusion models. Crucially,

performance now depends also on how good the autoencoder compresses images into latent space and recovers aesthetically pleasing images.

We provide additional discussion on VAEs in [Section D](#).

### 6.3 Case Study: Stable Diffusion 3 and Meta Movie Gen

We conclude this section by briefly examining two large-scale generative models: *Stable Diffusion 3* for image generation and Meta’s *Movie Gen Video* for video generation [14, 33]. As you will see, these models use the techniques we have described in this work along with additional architectural enhancements to both scale and accommodate richly structured conditioning modalities, such as text-based input.

#### 6.3.1 Stable Diffusion 3

Stable Diffusion is a series of state-of-the-art image generation models. These models were among the first to use large-scale latent diffusion models for image generation. If you have not done so, we highly recommend testing it for yourself online (<https://stability.ai/news/stable-diffusion-3>).

Stable Diffusion 3 uses the same conditional flow matching objective that we study in this work (see [Algorithm 4](#)).<sup>4</sup> As outlined in their paper, they extensively tested various flow and diffusion alternatives and found flow matching to perform best. For training, it uses classifier-free guidance training (with dropping class labels) as outlined above. Further, Stable Diffusion 3 follows the approach outlined in [Section 6.1](#) by training within the latent space of a pre-trained autoencoder. Training a good autoencoder was a big contribution of the first stable diffusion papers.

To enhance text conditioning, Stable Diffusion 3 makes use of both 3 different types of text embeddings (including CLIP embeddings as well as the sequential outputs produced by a pretrained instance of the encoder of Google’s T5-XXL [35], and similar to approaches taken in [3, 39]). Whereas CLIP embeddings provide a coarse, overarching embedding of the input text, the T5 embeddings provide a more granular level of context, allowing for the possibility of the model attending to particular elements of the conditioning text. To accommodate these sequential context embeddings, the authors then propose to extend the diffusion transformer to attend not just to patches of the image, but to the text embeddings as well, thereby extending the conditioning capacity from the class-based scheme originally proposed for DiT to sequential context embeddings. This proposed modified DiT is referred to as a **multi-modal DiT** (MM-DiT), and is depicted in [Figure 16](#). Their final, largest model has **8 billion parameters**. For sampling, they use 50 steps (i.e. they have to evaluate the network 50 times) using a Euler simulation scheme and a classifier-free guidance weight between 2.0-5.0.

#### 6.3.2 Meta Movie Gen Video

Next, we discuss Meta’s video generator, *Movie Gen Video* (<https://ai.meta.com/research/movie-gen/>). As the data are not images but *videos*, the data  $x$  lie in the space  $\mathbb{R}^{T \times C \times H \times W}$  where  $T$  represents the new **temporal** dimension (i.e. the number of frames). As we shall see, many of the design choices made in this video setting can

<sup>4</sup>In their work, they use a different convention to condition on the noise. But this is only notation and the algorithm is the same.

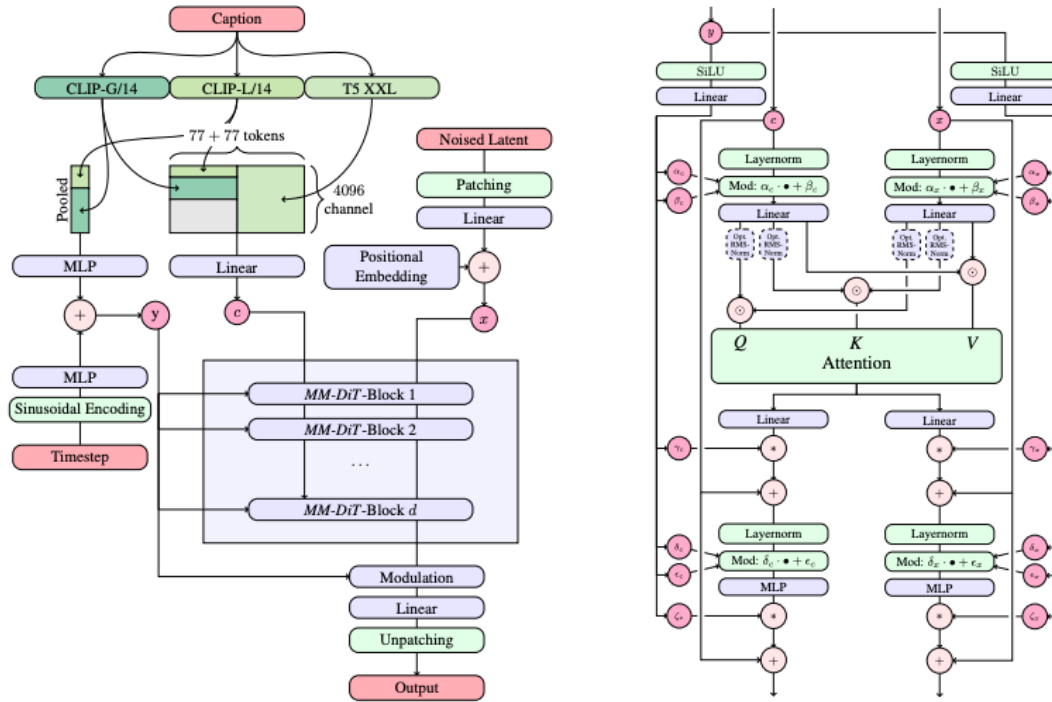


Figure 16: The architecture of the multi-modal diffusion transformer (MM-DiT) proposed in [14]. Figure also taken from [14].

be seen as adapting existing techniques (e.g., autoencoders, diffusion transformers, etc.) from the image setting to handle this extra temporal dimension.

Movie Gen Video utilizes the conditional flow matching objective with the same straight line schedulers  $\alpha_t = t, \sigma_t = 1 - t$ . Like Stable Diffusion 3, Movie Gen Video also operates in the latent space of frozen, pretrained autoencoder. Note that the autoencoder to reduce memory consumption is even more important for videos than for images - which is why most video generators right now are pretty limited in the length of the video they generate. Specifically, the authors propose to handle the added time dimension by introducing a **temporal autoencoder** (TAE) which maps a raw video  $x'_t \in \mathbb{R}^{T' \times 3 \times H \times W}$  to a latent  $x_t \in \mathbb{R}^{T \times C \times H \times W}$ , with  $\frac{T'}{T} = \frac{H'}{H} = \frac{W'}{W} = 8$  [33]. To accommodate long videos, a temporal tiling procedure is proposed by which the video is chopped up into pieces, each piece is encoded separately, and the latents are stitched together [33]. The model itself - that is,  $u_t^\theta(x_t)$  - is given by a DiT-like backbone in which  $x_t$  is patchified along the time and space dimensions. The image patches are then passed through a transformer employing both self-attention among the image patches, and cross-attention with language model embeddings, similar to the MM-DiT employed by Stable Diffusion 3. For text conditioning, Movie Gen Video employs three types of text embeddings: UL2 embeddings, for granular, text-based reasoning [47], ByT5 embeddings, for attending to character-level details (for e.g., prompts explicitly requesting specific text to be present) [50], and MetaCLIP embeddings, trained in a shared text-image embedding space [24, 33]. Their final, largest model has **30 billion parameters**. For a significantly more detailed and expansive treatment, we encourage the reader to check out the Movie Gen technical report itself [33].

## 7 Discrete Diffusion Models: Building Language Models with Diffusion

In previous sections, we explored flow and diffusion models as generative models over *Euclidean space*  $\mathbb{R}^d$  that allow us to generate data points represented by *vectors*  $z \in \mathbb{R}^d$ . However, not all data is naturally modeled as a point in Euclidean space  $\mathbb{R}^d$ . Many data types, such as text or DNA, are more naturally viewed as elements of a *discrete* state space  $S$ . Most importantly, language consists of a sequence of discrete tokens that we want to model. How could we apply flow and diffusion models to such data types? It turns out that the principles that we have learned in previous sections extend to such data types as well. The resulting models are called **discrete diffusion models** in the machine learning literature [5, 16]. However, it is important to keep in mind that there is no mathematical diffusion process (SDEs don't exist in discrete state spaces). Instead of having ODEs/SDEs, we use **continuous-time Markov chains (CTMCs)**. In the following, we will explain CTMC models (see Section 7.1) and how to learn them (see Section 7.2) allowing us to build large language models (LLMs) using the principles of flow and diffusion models.

### 7.1 Continuous-Time Markov chain (CTMC) models

In this section, we explain continuous-time Markov chains (CTMCs). You can think of CTMCs as a discrete analogue of SDEs that we can use to build neural network models that generate discrete states. Further, we will introduce CTMC models, i.e. neural network models that allow to generate discrete sequences such as text using CTMCs.

Let us begin by characterizing our **state space**  $S$ . Let  $\mathcal{V} = \{v_1, \dots, v_V\}$  be our vocabulary. The state space is given by  $S = \mathcal{V}^d$  where  $d \in \mathbb{N}$  is sequence length and  $V \in \mathbb{N}$  is the vocabulary size. For language,  $\{v_1, \dots, v_V\}$  could enumerate our alphabet or a set of discrete tokens and  $S$  would represent the set of sequences (or sentences) of length  $d$ . For DNA,  $\{v_1, \dots, v_V\}$  could be all 4 DNA bases and  $S$  all DNA sequences of length  $d$ .

Next, let  $X_t$  be a stochastic process on  $S$ , i.e. a random trajectory  $X : [0, 1] \rightarrow S, t \mapsto X_t$  in  $S$ . We require  $X_t$  to be a **Markov process**, i.e. a process that has no memory. Specifically, this means that the following condition holds

$$\underbrace{p(X_{t+h}|X_t, X_{t_1}, \dots, X_{t_k})}_{\text{prob. of future given present and past}} = \underbrace{p(X_{t+h}|X_t)}_{\text{prob. of future given present}} \quad (\text{for all } 0 < h, 0 \leq t_1 < t_2 < \dots < t_k < t)$$

In other words, the probabilities of future events only depend on the present - the past has no relevance for the future anymore. Note that ODE/SDEs - while not on discrete state spaces - are also Markov processes. Here,  $X_t$  is on a discrete space and therefore is called a Markov *chain*, specifically a **Continuous-time Markov chain (CTMC)**. The quantity  $p_{t+h|t}(X_{t+h}|X_t)$  are the **transition probabilities** and they fully determine the CTMC together with

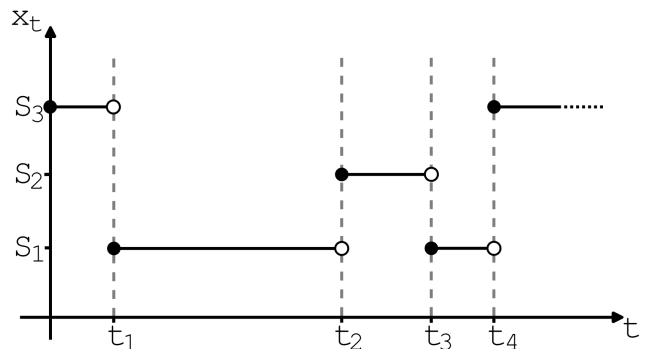


Figure 17: Illustration of a CTMC trajectory with state space  $S = \{S_1, S_2, S_3\}$  (sequence length  $d = 1$ ). Figure adapted from [5].

the initial distribution  $X_0 \sim p_0$  of the Markov chain. Therefore, when we say CTMC, you can also just think of transition probabilities  $p_{t+h|t}(X_{t+h}|X_t)$ .

Next, let us derive the analogue of a vector field in the discrete setting. As we are in a discrete setting, we can only jump (or switch) between states - we cannot go into a direction anymore like we did when specifying ODEs. Therefore, we define a **rate matrix**  $Q_t(y|x)$  that effectively summarizes the rate of jumping (or switching) from state  $x \in S$  to state  $y \in S$ . Formally, a rate matrix  $Q_t$  is given by a bounded function (continuous in time)

$$Q : S \times S \times [0, 1] \rightarrow \mathbb{R}, \quad (x, y, t) \mapsto Q_t(y|x) \quad (84)$$

where  $Q_t(y|x)$  describes the rate of switching from  $x$  to  $y$  such that

$$(1) \text{ Outgoing rates are positives: } Q_t(y|x) \geq 0 \quad \text{whenever } x \neq y \quad (85)$$

$$(2) \text{ Rate staying equals negative outgoing rate: } Q_t(x|x) = - \sum_{y \neq x} Q_t(y|x) \quad \text{for all } x \quad (86)$$

The two conditions are intuitive: The first condition says that the rate of switching from  $x$  to a different state  $y \neq x$  can only be non-negative (not switching just corresponds to 0 - so it does not make sense to have a rate that is smaller than 0). The second condition says that the rate  $Q_t(x|x)$  of staying at  $x$  should cancel out with the rate of leaving  $x$  - it is essentially a consistency condition saying that you have to either stay at  $x$  or leave (there is no third option). Note that these conditions imply in particular that  $Q_t(x|x) \leq 0$ . Hence,  $Q_t(y|x)$  is a matrix whose diagonal entries are all non-positive while all off-diagonal entries are non-negative.

We can now define the analogue of a differential equation, i.e. a condition on a CTMC to “follow” the rate matrix. The idea is basically that the distribution or evolution of  $X$  should follow the rate matrix  $Q_t$ . In other words, we require that the transition probabilities fulfill

$$\frac{d}{dh} p_{t+h|t}(X_{t+h} = y | X_t = x) |_{h=0} = Q_t(y|x) \quad \text{for all } x, y \in S, 0 \leq t \quad (87)$$

The left-hand side is the infinitesimal rate of change of the probability of switching from  $x$  to  $y$ . We impose the condition that these probabilities should change as specified by the rate matrix. Let’s briefly check that it is reasonable to request these conditions, i.e. we simply set  $Q_t(y|x)$  as in [Equation \(87\)](#), would it be a valid rate matrix? For  $h = 0$ , the probability of switching from  $x$  to  $y \neq x$  is zero (as no time has passed), i.e.  $p_{t|t}(y|x) = 0$  for all  $y \neq x$ . Therefore, we know that the derivative must be non-negative and  $Q_t(y|x) \geq 0$  whenever  $y \neq x$ . This checks that the first condition in [Equation \(85\)](#) holds. Further, we know that

$$\begin{aligned} \sum_{y \neq x} Q_t(y|x) &= \sum_{y \neq x} \frac{d}{dh} p(X_{t+h} = y | X_t = x) |_{h=0} = \frac{d}{dh} \sum_{y \neq x} p(X_{t+h} = y | X_t = x) |_{h=0} = \frac{d}{dh} (1 - p(X_{t+h} = x | X_t = x)) \\ &= - Q_t(x|x) \end{aligned}$$

where we used that probabilities sum to 1. This shows [Equation \(86\)](#). This checks that every CTMC has at least one rate matrix satisfying [Equation \(87\)](#). But what if we go backwards - what if we specify  $Q_t$ , is there a corresponding CTMC and if so, is it unique? This is indeed the case.

**Theorem 33** (CTMC existence and uniqueness)

For any rate matrix  $Q_t$  (bounded and continuous in time  $t$ ), there is a unique Markov chain  $X_t$  (i.e. a unique set of transition probabilities  $p_{t+h|t}(y|x)$ ) such that Equation (87) holds.

For the interested reader, we provide a self-contained proof in Section C. The key takeaway from this theorem is that for the purposes of machine learning, we can construct a rate matrix  $Q_t$  (e.g. via a neural network) and assume that there is a unique Markov chain that corresponds to  $Q_t$ .

**Example 34** (Two-state CTMC with equal jump rates)

Let  $S = \{a, b\}$  and consider a time-homogeneous CTMC  $(X_t)_{t \geq 0}$  that switches between both states at a constant rate  $\lambda > 0$ :

$$Q = \begin{array}{c|cc} & a & b \\ \hline a & -\lambda & \lambda \\ \hline b & \lambda & -\lambda \end{array}.$$

Then the transition probabilities over a time increment  $h \geq 0$  are also constant in time  $t$  and given by

$$\begin{pmatrix} p(X_{t+h} = a|X_t = a) & p(X_{t+h} = a|X_t = b) \\ p(X_{t+h} = b|X_t = a) & p(X_{t+h} = b|X_t = b) \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1 + e^{-2\lambda h} & 1 - e^{-2\lambda h} \\ 1 - e^{-2\lambda h} & 1 + e^{-2\lambda h} \end{pmatrix}.$$

One can check by hand that Equation (87) holds, i.e. these transition probabilities indeed are the correct ones for that rate matrix. In fact, these rates are very intuitive: The chain keeps flipping with an instantaneous rate  $\lambda$ . The exponential term  $e^{-2\lambda h}$  captures how the memory of the initial state decays. As infinite time passes, i.e. for  $h \rightarrow \infty$ , it holds that

$$P(h) \rightarrow \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix},$$

so the chain forgets where it started and is in  $a$  or  $b$  with probability  $1/2$ . This convergence is faster the higher the rate  $\lambda > 0$  of switching.

**Simulation of CTMC.** Next, let us think about how one would go about simulating a trajectory of a CTMC. Let  $h > 0$  be a step size and  $p_{\text{init}}$  be an initial distribution over  $S$ , e.g.  $p_{\text{init}} = \text{Unif}_S$  is the uniform distribution over  $S$ . Then we can simulate it iteratively by setting  $X_0 \sim p_{\text{init}}$  and setting

$$X_{t+h} \sim p_{t+h|t}(\cdot|X_t)$$

Now, this would work if we knew  $p_{t+h|t}(\cdot|X_t)$ . However, for all but the simplest CTMCs, we typically do not know the transition kernel in closed form and only have access to the rate matrix  $Q_t$ . Still, by Equation (87):

$$p_{t+h|t}(X_{t+h} = y|X_t = x) = p_{t|t}(X_t = y|X_t = x) + hQ_t(y|x) + R_t(h) = 1_{y=x} + hQ_t(y|x) + R_t(h)$$

where  $R_t(h)$  is an error term that we can neglect for small  $h$ . Therefore, for small  $h$ , we can set

$$p_{t+h|t}(X_{t+h} = y|X_t = x) \approx 1_{y=x} + hQ_t(y|x) =: \tilde{p}_{t+h|t}(y|x)$$

One can check that  $\tilde{p}_{t+h|t}(y|x)$  is indeed a valid probability distribution for small  $h$  by the conditions we imposed on the rate matrix. Therefore, we can approximately sample the next point via

$$X_{t+h} \sim \tilde{p}_{t+h|t}(\cdot|x) = (1_{y=x} + hQ_t(y|x))_{y \in S} \quad (88)$$

As the above is just a discrete distribution, we can sample from it easily via standard methods. This is a simple way to simulate a CTMC.

**CTMC model.** Next, let us define how we can parameterize a CTMC in a neural network. A **CTMC model** (or **discrete diffusion model**) is given by an initial distribution  $p_{\text{init}}$  over  $S$  and a neural network  $Q_t^\theta$  with parameters  $\theta$  such that for every input  $x \in S$  the model returns a single column of the rate matrix

$$x \mapsto \{Q_t^\theta(y|x)\}_{y \in S}$$

We want the model to return an entire column because we require it for simulation of the CTMC (Equation (88)), i.e. sampling the next state.

One complication with the above model is that the space  $S$  can be *very* large. In particular,  $|S| = V^d$  where  $V$  is our vocabulary size and  $d$  is the sequence length. This exponential growth makes it basically impossible to store an entire column of the rate matrix in memory -  $\{Q_t^\theta(y|x)\}_{y \in S}$  could never be represented in a computer. Therefore, we have to constrain the model. Specifically, almost all CTMC models are *factorized* (see Figure 18), which is effectively a sparsity constraint. Specifically, a **factorized CTMC model** is given by a CTMC model  $Q_t^\theta$  such that for all  $y = (y_1, \dots, y_d), x = (x_1, \dots, x_d) \in S = \mathcal{V}^d$  it holds

$$Q_t^\theta(y|x) = 0 \quad \text{whenever } y_i \neq x_i \text{ for more than one position } i$$

We call all  $y$  that differ from  $x$  in at most one token the **neighbors**  $N(x)$  of  $x$ . We can write such a factorized CTMC model as

$$x \mapsto \{Q_t^\theta(y|x)\}_{y \in N(x)} = \begin{pmatrix} Q_t^\theta(v_1, 1|x) & \cdots & Q_t^\theta(v_V, 1|x) \\ \cdots & & \cdots \\ Q_t^\theta(v_1, d|x) & \cdots & Q_t^\theta(v_V, d|x) \end{pmatrix}$$

where  $Q_t(y|x) = Q_t^\theta(v_i, j|x)$  now gives the rate of going from  $x = (x_1, \dots, x_d)$  to the neighbor of  $x$  that we obtain swapping out the  $j$ -th element with  $v_i$ , i.e.  $y = (x_1, \dots, x_{j-1}, v_i, x_{j+1}, \dots, x_d)$ . Each row corresponds to a rate matrix per position  $i = 1, \dots, d$ , i.e. we require

$$Q_t^\theta(v, i|x) \geq 0 \text{ if } v \neq x_i, \quad Q_t(x_i, i|x) = - \sum_{v \neq x_i} Q_t^\theta(v, i|x)$$

We can enforce these conditions on the output of a neural network easily, e.g. one can use a transformer model on sequence length  $d$  with output dimension  $V$ . Note also that the factorized rate matrix makes the output shape  $d \times V$  - this size increases linearly in the dimension (as opposed to exponentially).

**Simulating a CTMC model.** To sample from a CTMC model, we sample  $X_0 \sim p_{\text{init}}$  and perform an iteration where we sample the next state according to Equation (88). We present an algorithm in Algorithm 7. As shown

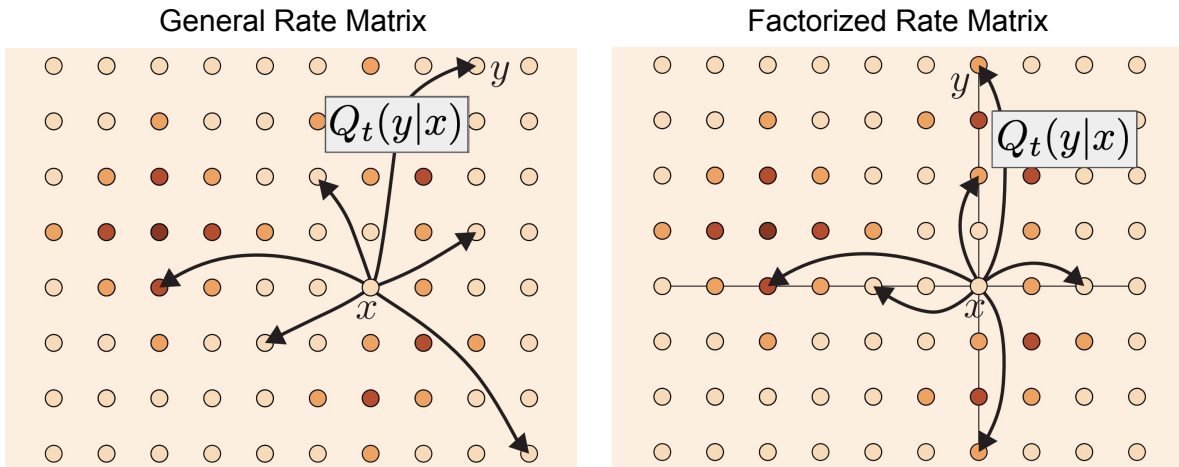


Figure 18: Illustration of a factorized CTMC model. Factorized CTMCs have only non-zero rates ( $Q_t(y|x) \neq 0$ ) if the start and end point differ by only one dimension (here,  $d = 2$ ). Figure taken from [26].

there, for factorized CTMC models, one can use a parallel per-token Euler approximation, where each token is updated independently during a small step  $h > 0$ . This approximation agrees with the full CTMC Euler step up to first order in  $h$ , but allows for a  $O(h^2)$  probability of simultaneous updates to multiple tokens.

---

**Algorithm 7** Sampling from a Factorized CTMC Model

---

**Require:** Rate network  $Q_t^\theta$  (factorized), initial distribution  $p_{\text{init}}$ , number of steps  $n$

- 1: Set  $t \leftarrow 0$ , step size  $h \leftarrow \frac{1}{n}$
- 2: Draw a sample  $X_0 \sim p_{\text{init}}$ , where  $X_0 = (X_0^{(1)}, \dots, X_0^{(d)}) \in \mathcal{V}^d$
- 3: **for**  $i = 1, \dots, n$  **do**
- 4:   Compute factorized jump rates  $\{q_j(v)\}_{j=1..d, v \in \mathcal{V}} \leftarrow Q_t^\theta(\cdot | X_t)$
- 5:   **for**  $j = 1, \dots, d$  (**in parallel**) **do**
- 6:      $x \leftarrow X_t^{(j)}$  {current token at position  $j$ }
- 7:     Define the per-position Euler transition probabilities  $\tilde{p}_{j,t}(\cdot | X_t^{(j)} = x)$  by

$$\tilde{p}_{j,t}(v | x) = \begin{cases} h q_j(v), & v \neq x, \\ 1 - h \sum_{v' \in \mathcal{V} \setminus \{x\}} q_j(v'), & v = x. \end{cases}$$

- 8:     Sample  $X_{t+h}^{(j)} \sim \text{CATEGORICAL}(\{\tilde{p}_{j,t}(v | x)\}_{v \in \mathcal{V}})$
  - 9:   **end for**
  - 10:   Set  $t \leftarrow t + h$
  - 11: **end for**
  - 12: **return**  $X_1$
-

## 7.2 Training CTMC models

We next discuss how to learn CTMC models. The principles are the same as for flow matching: (1) We construct a probability path interpolating between noise and data. (2) We derive a conditional rate matrix and marginal rate matrix. (3) We learn the marginal rate matrix in a simulation-free manner. We will explain this recipe now step-by-step.

In this section, the **data distribution**  $p_{\text{data}}$  is a distribution over  $S$  characterized by a probability mass function. Namely,  $p_{\text{data}} : S \rightarrow \mathbb{R}_{\geq 0}, z \mapsto p_{\text{data}}(z)$  with  $\sum_{z \in S} p_{\text{data}}(z) = 1$ . We do not know  $p_{\text{data}}$  but we access to samples  $z \sim p_{\text{data}}$  during training in form of a data set. For example, all texts on the world wide web. Our goal is to learn to generate samples  $z \sim p_{\text{data}}$ . Our goal is to train the CTMC model  $Q_t^\theta$  such that

$$X_0 \sim p_{\text{init}}, \quad X_t \text{ CTMC of } Q_t^\theta \quad \Rightarrow \quad X_1 \sim p_{\text{data}}$$

So as you might realize, this is no different from the Euclidean case  $\mathbb{R}^d$  (see [Sections 2 and 3](#)), just that we use a CTMC model instead of a flow/diffusion model.

### 7.2.1 Conditional and Marginal Probability Path

We define  $\delta_z(x)$  to be function such that  $\delta_z(x) = 0$  if  $x \neq z$  and  $\delta_z(x) = 1$  if  $x = z$ . A (discrete) **conditional probability path** is given by set of distributions  $p_t(x|z)$  for  $x, z \in S$  and  $0 \leq t \leq 1$  such that

$$p_0(\cdot|z) = p_{\text{init}}, \quad p_1(\cdot|z) = \delta_z$$

So similar to the Euclidean case, a discrete conditional probability path interpolates between a distribution that is independent of  $z$  to a distribution that has all mass on  $z$ . A (discrete) **marginal probability path** is then given by

$$p_t(x) = \sum_{z \in S} p_t(x|z) p_{\text{data}}(z)$$

One can easily check that the marginal probability path interpolates “noise” and data:

$$p_0 = p_{\text{init}}, \quad p_1 = p_{\text{data}} \tag{89}$$

#### Example 35 (Factorized mixture path (independent noising per token))

Let  $S = \mathcal{V}^d$  and let  $p_{\text{init}}(x) = \prod_{j=1}^d p_{\text{init}}^{(j)}(x_j)$  be a factorized initial distribution. Fix a **scheduler**  $0 \leq \kappa_t \leq 1$  such that  $\kappa_0 = 0, \kappa_1 = 1$  with  $\frac{d}{dt} \kappa_t \geq 0$ . Define the conditional path by

$$p_t(x|z) = \prod_{j=1}^d \left[ (1 - \kappa_t) p_{\text{init}}^{(j)}(x_j) + \kappa_t \delta_{z_j}(x_j) \right].$$

Equivalently, one can sample  $x \sim p_t(\cdot | z)$  by drawing i.i.d. masks  $m_j = 0, 1$  and noise  $\xi_j \sim p_{\text{init}}^{(j)}$ , then setting

$$\begin{aligned}
 m_j &\sim \text{Bernoulli}(\kappa_t), \quad \xi_j \sim p_{\text{init}}^{(j)} \\
 x_j &= m_j z_j + (1 - m_j) \xi_j, \quad j = 1, \dots, d \\
 x &= (x_1, \dots, x_d)
 \end{aligned}$$

We call the above the **factorized mixture path**. The above procedure effectively “destroys” the  $j$ -th token independently for each position in the sequence with a probability  $1 - \kappa_t$ , i.e. for  $t = 0$   $1 - \kappa_t = 1$  and all information is destroyed and for  $t = 1$  it holds that  $1 - \kappa_t = 0$  and no information is destroyed. Note that this is similar to the Gaussian probability path [Example 8](#) in the sense that information is destroyed progressively with a speed determined by a scheduler  $\kappa_t$ . However, it is also different from the Gaussian probability path as the factorized mixture path does *not* move/transports probability mass (there is no direction as we are in discrete space) - it simply fades in one distribution and fades out another.

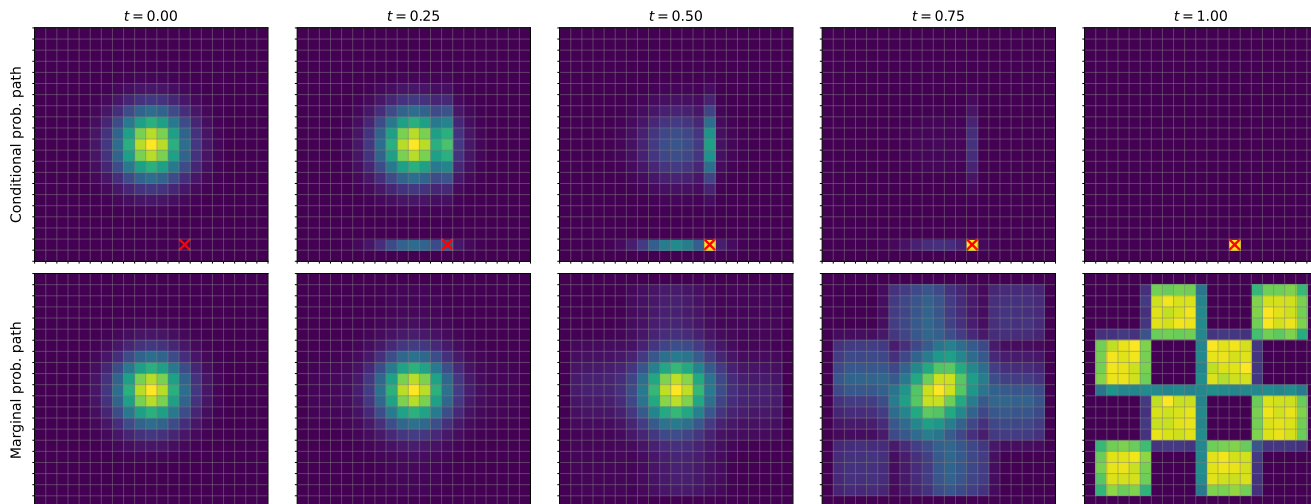


Figure 19: Illustration of a discrete probability path for  $d = 2$ . Top row: Conditional probability path interpolating between initial distribution and Dirac distribution. Bottom row: Interpolation between initial distribution and data distribution (here, chess board pattern). Note the similarity and differences to [Figure 5](#): Here, the probability path is “teleported” (we downweigh the initial distribution and upweigh the terminal distribution).

### 7.2.2 Conditional and Marginal Rate Matrix

As a next step, we will now construct the training target of discrete flow matching. First, we construct a conditional rate matrix - the analogue to the conditional vector field for flow matching. Let  $Q_t^z(y|x)$  be a rate matrix for every data point  $z \in S$ . Then we call it a **conditional rate matrix** if

$$X_0 \sim p_{\text{init}}, \quad X_t \text{ CTMC of } Q_t^z \quad \Rightarrow \quad X_t \sim p_t(\cdot | z)$$

In other words, the conditional rate matrix is such that its CTMC “follows” the conditional probability path. The conditional rate matrix serves as a building block to construct the marginal rate matrix that follows the marginal probability path:

**Theorem 36** (Discrete marginalization trick)

The **marginal rate matrix** defined by

$$Q_t(y|x) = \sum_{z \in S} Q_t^z(y|x) \frac{p_t(x|z)p_{\text{data}}(z)}{p_t(x)} = \sum_{z \in S} Q_t^z(y|x)p_{1|t}(z|x) \quad \text{where } p_{1|t}(z|x) := \frac{p_t(x|z)p_{\text{data}}(z)}{p_t(x)} \quad (90)$$

is a valid rate matrix and fulfills the following condition:

$$X_0 \sim p_{\text{init}}, \quad X_t \text{ CTMC of } Q_t \quad \Rightarrow \quad X_t \sim p_t$$

In particular,  $X_1 \sim p_{\text{data}}$  by [Equation \(89\)](#), i.e. the CTMC of the marginal rate matrix converts noise to data.

To prove this statement, we need a fundamental equation for CTMCs, the so-called *Kolmogorov Forward equation*:

**Proposition 2** (Kolmogorov Forward Equation)

Let  $p_t$  be a set of distributions on  $S$  for every  $0 \leq t \leq 1$ . Further, let  $X_t$  be a CTMC with matrix  $Q_t$  and initial distribution  $p_0$ . Then  $X_t \sim p_t$  for all  $0 \leq t \leq 1$  if and only if the **Kolmogorov Forward Equation (KFE)** holds:

$$\frac{d}{dt} p_t(x) = \sum_{y \in S} Q_t(x|y)p_t(y)$$

*Proof of KFE.* To show that the KFE is necessary, assume that  $p_t(x)$  are the true marginals of the CTMC, i.e.  $X_t \sim p_t$  for every  $0 \leq t \leq 1$ . Then we can compute:

$$\begin{aligned} \frac{d}{dt} p_t(x) &\stackrel{(i)}{=} \frac{d}{dh} \Big|_{h=0} p_{t+h}(x) \\ &\stackrel{(ii)}{=} \frac{d}{dh} \Big|_{h=0} \sum_y p_{t+h|t}(x|y)p_t(y) \\ &\stackrel{(iii)}{=} \sum_y \frac{d}{dh} \Big|_{h=0} p_{t+h|t}(x|y)p_t(y) \\ &\stackrel{(iv)}{=} \sum_y Q_t(x|y)p_t(y) \end{aligned}$$

where in (i) we simply use a time offset, in (ii) we use the definition of the transition probabilities, in (iii) we swap sum and derivative, and in (iv) we use the definition of the rate matrix (see [Equation \(87\)](#)).

Next, to show that the KFE is sufficient, we can rewrite the KFE in matrix form:

$$\frac{d}{dt} p_t = Q_t p_t$$

where in this equation we consider  $p_t = (p_t(x))_{x \in S}$  as a vector and  $Q_t = (Q_t(y|x))_{x,y \in S}$  as a matrix. Note that the above is a linear ODE over vector space  $\mathbb{R}^S$ . Its initial condition is fixed by  $p_0$  as stated in the theorem. Therefore,

if any other set of marginals  $q_t$  fulfills this equation, we know that by the uniqueness of ODEs (see [Theorem 3](#)) that we can conclude that  $q_t = p_t$ . This shows that the KFE is also sufficient.  $\square$

*Proof of Theorem 36.* Using the KFE, it remains to show that marginal rate matrix defined as in the theorem (see [Equation \(90\)](#)) fulfills the KFE:

$$\begin{aligned}
 \frac{d}{dt}p_t(x) &\stackrel{(i)}{=} \frac{d}{dt} \sum_{z \in S} p_t(x|z)p_{\text{data}}(z) \\
 &\stackrel{(ii)}{=} \sum_{z \in S} \frac{d}{dt} p_t(x|z)p_{\text{data}}(z) \\
 &\stackrel{(iii)}{=} \sum_{z \in S} \left[ \sum_{y \in S} Q_t^z(x|y)p_t(y|z) \right] p_{\text{data}}(z) \\
 &\stackrel{(iv)}{=} \sum_{y \in S} p_t(y) \left[ \sum_{z \in S} Q_t^z(x|y) \frac{p_t(y|z)p_{\text{data}}(z)}{p_t(y)} \right] \\
 &\stackrel{(v)}{=} \sum_{y \in S} p_t(y)Q_t(x|y)
 \end{aligned}$$

where (i) follows by the definition of the marginal probability path, in (ii) we swap the sum and the derivative, in (iii) we use the KFE on the conditional rate matrix, in (iv) we multiply and divide by  $p_t(y)$ , and in (v) we use the definition of the marginal rate matrix  $Q_t(y|x)$ . This shows that the KFE is fulfilled. The statement follows by [Proposition 2](#).  $\square$

Let us now derive a concrete example of a conditional rate matrix for the factorized mixture path.

**Example 37** (Conditional rate matrix for factorized mixture path)

Set  $\frac{d}{dt}\kappa_t = \dot{\kappa}_t$ . The factorized mixture path has a factorized conditional rate matrix given by

$$\begin{aligned}
 Q_t^z(y|x) &= (Q_t^z(v_i, j|x_j))_{v_i, j} \\
 Q_t^z(v_i, j|x_j) &= \frac{\dot{\kappa}_t}{1 - \kappa_t} (\delta_{z_j}(v_i) - \delta_{x_j}(v_i)) \\
 &= \frac{\dot{\kappa}_t}{1 - \kappa_t} \begin{cases} 0 & \text{if } x_j = z_j \\ 1 & \text{if } v_i = z_j, x_j \neq z_j \\ 0 & \text{if } v_i \neq z_j, x_j \neq z_j \\ -1 & \text{if } v_i = x_j, x_j \neq z_j \end{cases}
 \end{aligned}$$

Note that this is a very simple rate matrix: It only allows for jumps to  $z^j$  - i.e. if any token  $j$  is updated, it must jump to the token value of the terminal data point  $z = (z_1, \dots, z_d)$  - and it only jumps to  $z^j$  if we are not yet there.

*Proof.* We note that the factorized mixture path completely factorizes into independent components and so does the suggested conditional rate matrix. Therefore, we can without loss of generality assume that  $d = 1$ . So

we just do the calculation per dimension. Then, we can derive:

$$\begin{aligned}
\frac{d}{dt}p_t(x|z) &\stackrel{(i)}{=} \frac{d}{dt} [(1 - \kappa_t)p_{\text{init}}(x) + \kappa_t\delta_z(x)] \\
&\stackrel{(ii)}{=} \dot{\kappa}_t\delta_z(x) - \dot{\kappa}_tp_{\text{init}}(x) \\
&\stackrel{(iii)}{=} \frac{\dot{\kappa}_t}{1 - \kappa_t}(\delta_z(x) - [(1 - \kappa_t)p_{\text{init}}(x) + \kappa_t\delta_z(x)]) \\
&\stackrel{(iv)}{=} \frac{\dot{\kappa}_t}{1 - \kappa_t}(\delta_z(x) - p_t(x|z)) \\
&\stackrel{(v)}{=} \frac{\dot{\kappa}_t}{1 - \kappa_t}\delta_z(x)(1 - p_t(x|z)) + \frac{\dot{\kappa}_t}{1 - \kappa_t}(\delta_z(x) - 1)p_t(x|z) \\
&\stackrel{(vi)}{=} \sum_{y \neq x} \frac{\dot{\kappa}_t}{1 - \kappa_t}\delta_z(x)p_t(y|z) + \frac{\dot{\kappa}_t}{1 - \kappa_t}(\delta_z(x) - 1)p_t(x|z) \\
&\stackrel{(vii)}{=} \sum_{y \neq x} Q_t^z(x|y)p_t(y|z) + Q_t^z(x|x)p_t(x|z) \\
&\stackrel{(viii)}{=} \sum_{y \in S} Q_t^z(x|y)p_t(y|z)
\end{aligned}$$

where (i) uses the definition of the factorized mixture path for  $d = 1$ , (ii) is obtained by taking derivatives and setting  $\frac{d}{dt}\kappa_t = \dot{\kappa}_t$ , (iii) follows by simple algebra, (iv) by the definition of the factorized mixture path, (v) by simple algebra, (vi) follows by the definition the fact that  $\sum_{y \in S} p_t(y|z) = 1$ , (vii) by the definition of the rate matrix, and (viii) by simple algebra. The above shows that the KFE is fulfilled and therefore the statement follows.  $\square$

### 7.2.3 Learning the Marginal Rate Matrix

In this section, we derive the fundamental algorithm for training CTMC models. By [Theorem 36](#), training a CTMC model  $Q_t^g(y|x)$  can be achieved by learning the marginal rate matrix.

In this section, we now restrict ourselves to the factorized mixture path (see [Example 35](#)) as this is the path most discrete diffusion/flow matching models use so far. In this case, the marginal rate matrix has a very intuitive shape:

#### **Theorem 38** (Marginalization trick for factorized mixture path)

The marginal rate matrix of the factorized mixture path is factorized and has the form

$$Q_t(v_i, j|x) = \frac{\dot{\kappa}_t}{1 - \kappa_t}(p_{1|t}(z_j = v_i|x) - \delta_{x_j}(v_i))$$

where  $p_{1|t}(z_j = v_i|x)$  is the conditional probability of the  $j$ -th position ( $j$ -th token in the sequence) being equal to  $v_i$  given the full noisy sequence  $x$ .

*Proof.* The marginal rate matrix is given by

$$Q_t(y|x) = \sum_{z \in S} Q_t^z(y|x)p_{1|t}(z|x) \tag{91}$$

Now, whenever  $y$  and  $x$  are not neighbors (differ by more than one token),  $Q_t^z(y|x) = 0$  for every  $z$ . Therefore, also  $Q_t(y|x) = 0$  in this case. This shows that marginal rate matrix factorizes as well. It then holds that

$$Q_t(v_i, j|x) = \sum_{z \in S} Q_t^z(v_i, j|x) p_{1|t}(z|x) \quad (92)$$

$$\stackrel{(i)}{=} \sum_{z \in S} \frac{\dot{\kappa}_t}{1 - \kappa_t} (\delta_{z_j}(v_i) - \delta_{x_j}(v_i)) p_{1|t}(z|x) \quad (93)$$

$$\stackrel{(ii)}{=} \frac{\dot{\kappa}_t}{1 - \kappa_t} \left( \sum_{z \in S} \delta_{z_j}(v_i) p_{1|t}(z|x) - \delta_{x_j}(v_i) \right) \quad (94)$$

$$\stackrel{(iii)}{=} \frac{\dot{\kappa}_t}{1 - \kappa_t} (p_{1|t}(z_j = v_i|x) - \delta_{x_j}(v_i)) \quad (95)$$

where (i) follows by the formula for the conditional rate matrix (see [Example 37](#)), (ii) follows by the fact that  $\sum_{z \in S} p_{1|t}(z|x) = 1$ , and (iii) follows by marginalization. This finishes the proof.  $\square$

The previous theorem is remarkable: The marginal rate matrix is effectively a reparameterization of the probabilities  $p_{1|t}(z_j = v_i|x)$ . This is effectively nothing else than learning a classifier for each token position  $j = 1, \dots, d$ . In other words, we can simply define a **denoising probabilities network** as

$$p_{1|t}^\theta : \underbrace{x}_{\text{network input}} \mapsto \underbrace{(p_{1|t}^\theta(z_j = v_i|x))_{j=1, \dots, d, v_i \in \mathcal{V}}}_{\text{network output}}$$

Note that the network output has shape  $d \times V$ . One can obtain probabilities per token position via simple softmax layer. The network itself can be a standard sequence-to-sequence network, e.g. a transformer works (see [Section 6.1.2](#)).

As this is simply a classifier per position  $j$ , we can train such a network via the cross-entropy loss per  $j = 1, \dots, d$ . This leads to the **Discrete Flow Matching loss** given by

$$\mathcal{L}_{\text{DFM}}(\theta) = \mathbb{E}_{z \sim p_{\text{data}}, t \sim \text{Unif}_{[0,1]}, x \sim p_t(\cdot|z)} \left[ \sum_{j=1}^d -\log p_{1|t}^\theta(z_j|x) \right]$$

This is remarkable: To train a generative model, all we need to do is to train a classifier model per position  $j$ . In the same way as continuous flow matching reduced to simple regression (see [Section 3](#)), discrete flow matching and discrete diffusion models reduce to simple classification training. In [Algorithm 8](#), we summarize the training algorithm. Post-training, we can sample via [Algorithm 7](#).

#### Example 39 (Masked Diffusion Language Model)

A specific case of the above method is **masked diffusion language models (MDLMs)**. The idea of MDLMs is that we can extend the vocabulary of tokens  $\mathcal{V} = \{v_1, \dots, v_V\}$  with a new token [mask] that indicates that this token is missing (or was masked). Specifically, we set  $\mathcal{V} = \{v_1, \dots, v_V, [\text{mask}]\}$  and the initial point is simply [mask]<sup>d</sup>, i.e. the sequence that is all-masked. Formally, this means setting  $p_{\text{init}} = \delta_{[\text{mask}]^d}$  in the above framework. The sampling procedure is illustrated in [Figure 20](#).

**Algorithm 8** Training factorized CTMC Model (Discrete Diffusion)

- Require:** Dataset of sequences  $z \sim p_{\text{data}}$  with  $z = (z_1, \dots, z_d) \in \mathcal{V}^d$ ;  
 initial (noise) token marginals  $p_{\text{init}}^{(j)}$  on  $\mathcal{V}$ ; schedule  $\kappa_t \in [0, 1]$ ;  
 posterior network  $f_\theta$  returning per-position logits over  $\mathcal{V}$ ; optimizer OPT
- 1: **for** each training iteration **do**
  - 2:   Sample a data point  $z \sim p_{\text{data}}$
  - 3:   Sample time  $t \sim \text{Unif}[0, 1]$  and compute  $\kappa \leftarrow \kappa_t$
  - 4:   Sample a noisy state  $x \sim p_t(\cdot | z)$  (factorized mixture path):
  - 5:   **for**  $j = 1, \dots, d$  (**in parallel**) **do**
  - 6:     Sample mask  $m_j \sim \text{Bernoulli}(\kappa)$
  - 7:     Sample noise token  $\xi_j \sim p_{\text{init}}^{(j)}$
  - 8:     Set  $x_j \leftarrow m_j z_j + (1 - m_j) \xi_j$
  - 9:   **end for**
  - 10:    $x \leftarrow (x_1, \dots, x_d)$
  - 11:   Predict terminal-token posteriors via logits from the network:

$$\ell_j(\cdot) \leftarrow f_\theta(x, t)_j \quad \Rightarrow \quad p_{1|t}^\theta(v | x)_j = \text{Softmax}(\ell_j)(v)$$

- 12:   Discrete Flow Matching loss (token-wise NLL of  $z$ ):

$$\mathcal{L}_{\text{DFM}}(\theta) \leftarrow \sum_{j=1}^d \left[ -\log p_{1|t}^\theta(z_j | x)_j \right]$$

- 13:   Update parameters:  $\theta \leftarrow \text{OPT.STEP}(\nabla_\theta \mathcal{L}_{\text{DFM}}(\theta))$
- 14: **end for**

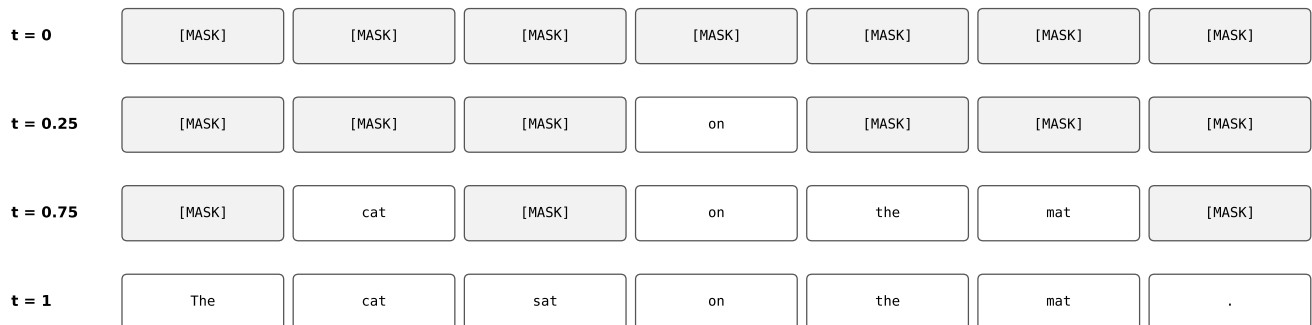


Figure 20: Illustration of the trajectory of a Masked Diffusion Language Model.

This completes now a full pipeline of training and sampling CTMC models that allows us to generate discrete sequences such as text. Current state-of-the-art discrete diffusion models [4] use the recipe described in this work, with neural networks (usually transformers) trained on web-scale data.

---

**Remark 40** (Generator Matching)

You may wonder why the principles of flow/diffusion models could be translated so seamlessly to discrete state spaces. As it turns out, the principles of flow matching are not unique to flows or even CTMCs. Rather, these are general learning principles for constructing generative models with *Markov processes*. This idea leads to the **Generator Matching** framework [19], a framework that extends and unifies both discrete and continuous flow and diffusion models into one. A generator is a generalization of a vector field  $u_t$  and a rate matrix  $Q_t$ . Markov processes and generators can be built for any data modality and state spaces. For example, you can build models for smooth manifolds [8, 10] (e.g. geometric data), mixed state spaces (e.g. joint text and image generation) [6], and other Markov processes such as jump processes [19, 7].

## 8 References

- [1] Michael S Albergo, Nicholas M Boffi, and Eric Vanden-Eijnden. “Stochastic interpolants: A unifying framework for flows and diffusions”. In: *arXiv preprint arXiv:2303.08797* (2023).
- [2] Brian DO Anderson. “Reverse-time diffusion equation models”. In: *Stochastic Processes and their Applications* 12.3 (1982), pp. 313–326.
- [3] Yogesh Balaji et al. *eDiff-I: Text-to-Image Diffusion Models with an Ensemble of Expert Denoisers*. 2023. arXiv: 2211.01324 [cs.CV]. URL: <https://arxiv.org/abs/2211.01324>.
- [4] Tiwei Bie et al. “Llada2. 0: Scaling up diffusion language models to 100b”. In: *arXiv preprint arXiv:2512.15745* (2025).
- [5] Andrew Campbell et al. “A continuous time framework for discrete denoising models”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 28266–28279.
- [6] Andrew Campbell et al. “Generative flows on discrete state-spaces: Enabling multimodal flows with applications to protein co-design”. In: *arXiv preprint arXiv:2402.04997* (2024).
- [7] Andrew Campbell et al. “Trans-dimensional generative modeling via jump diffusion models”. In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 42217–42257.
- [8] Ricky TQ Chen and Yaron Lipman. “Flow matching on general geometries”. In: *arXiv preprint arXiv:2302.03660* (2023).
- [9] Earl A Coddington, Norman Levinson, and T Teichmann. *Theory of ordinary differential equations*. 1956.
- [10] Valentin De Bortoli et al. “Riemannian score-based generative modelling”. In: *Advances in neural information processing systems* 35 (2022), pp. 2406–2422.
- [11] Prafulla Dhariwal and Alex Nichol. *Diffusion Models Beat GANs on Image Synthesis*. 2021. arXiv: 2105.05233 [cs.LG]. URL: <https://arxiv.org/abs/2105.05233>.
- [12] Alexey Dosovitskiy. “An image is worth 16x16 words: Transformers for image recognition at scale”. In: *arXiv preprint arXiv:2010.11929* (2020).
- [13] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2021. arXiv: 2010.11929 [cs.CV]. URL: <https://arxiv.org/abs/2010.11929>.

- 
- [14] Patrick Esser et al. *Scaling Rectified Flow Transformers for High-Resolution Image Synthesis*. 2024. arXiv: 2403.03206 [cs.CV]. URL: <https://arxiv.org/abs/2403.03206>.
- [15] Lawrence C Evans. *Partial differential equations*. Vol. 19. American Mathematical Society, 2022.
- [16] Itai Gat et al. “Discrete flow matching”. In: *Advances in Neural Information Processing Systems 37* (2024), pp. 133345–133385.
- [17] Jonathan Ho, Ajay Jain, and Pieter Abbeel. “Denoising diffusion probabilistic models”. In: *Advances in neural information processing systems 33* (2020), pp. 6840–6851.
- [18] Jonathan Ho and Tim Salimans. *Classifier-Free Diffusion Guidance*. 2022. arXiv: 2207.12598 [cs.LG]. URL: <https://arxiv.org/abs/2207.12598>.
- [19] Peter Holderrieth et al. “Generator matching: Generative modeling with arbitrary markov processes”. In: *arXiv preprint arXiv:2410.20587* (2024).
- [20] Peter Holderrieth et al. “GLASS Flows: Transition Sampling for Alignment of Flow and Diffusion Models”. In: *arXiv preprint arXiv:2509.25170* (2025).
- [21] Arieh Iserles. *A first course in the numerical analysis of differential equations*. Cambridge university press, 2009.
- [22] Alexia Jolicœur-Martineau et al. “Adversarial score matching and improved sampling for image generation”. In: *arXiv preprint arXiv:2009.05475* (2020).
- [23] Tero Karras et al. “Elucidating the design space of diffusion-based generative models”. In: *Advances in Neural Information Processing Systems 35* (2022), pp. 26565–26577.
- [24] Samuel Lavoie et al. *Modeling Caption Diversity in Contrastive Vision-Language Pretraining*. 2024. arXiv: 2405.00740 [cs.CV]. URL: <https://arxiv.org/abs/2405.00740>.
- [25] Yaron Lipman et al. “Flow matching for generative modeling”. In: *arXiv preprint arXiv:2210.02747* (2022).
- [26] Yaron Lipman et al. “Flow Matching Guide and Code”. In: *arXiv preprint arXiv:2412.06264* (2024).
- [27] Xingchao Liu, Chengyue Gong, and Qiang Liu. “Flow straight and fast: Learning to generate and transfer data with rectified flow”. In: *arXiv preprint arXiv:2209.03003* (2022).
- [28] Nanye Ma et al. “Sit: Exploring flow and diffusion-based generative models with scalable interpolant transformers”. In: *arXiv preprint arXiv:2401.08740* (2024).
- [29] Xuerong Mao. *Stochastic differential equations and applications*. Elsevier, 2007.
- [30] William Peebles and Saining Xie. *Scalable Diffusion Models with Transformers*. 2023. arXiv: 2212.09748 [cs.CV]. URL: <https://arxiv.org/abs/2212.09748>.
- [31] Ethan Perez et al. “Film: Visual reasoning with a general conditioning layer”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. 1. 2018.
- [32] Lawrence Perko. *Differential equations and dynamical systems*. Vol. 7. Springer Science & Business Media, 2013.
- [33] Adam Polyak et al. *Movie Gen: A Cast of Media Foundation Models*. 2024. arXiv: 2410.13720 [cs.CV]. URL: <https://arxiv.org/abs/2410.13720>.

- 
- [34] Alec Radford et al. *Learning Transferable Visual Models From Natural Language Supervision*. 2021. arXiv: 2103.00020 [cs.CV]. URL: <https://arxiv.org/abs/2103.00020>.
- [35] Colin Raffel et al. *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. 2023. arXiv: 1910.10683 [cs.LG]. URL: <https://arxiv.org/abs/1910.10683>.
- [36] Robin Rombach et al. *High-Resolution Image Synthesis with Latent Diffusion Models*. 2022. arXiv: 2112.10752 [cs.CV]. URL: <https://arxiv.org/abs/2112.10752>.
- [37] Robin Rombach et al. “High-resolution image synthesis with latent diffusion models”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2022, pp. 10684–10695.
- [38] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. “U-net: Convolutional networks for biomedical image segmentation”. In: *Medical image computing and computer-assisted intervention—MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*. Springer. 2015, pp. 234–241.
- [39] Chitwan Saharia et al. *Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding*. 2022. arXiv: 2205.11487 [cs.CV]. URL: <https://arxiv.org/abs/2205.11487>.
- [40] Simo Särkkä and Arno Solin. *Applied stochastic differential equations*. Vol. 10. Cambridge University Press, 2019.
- [41] Jascha Sohl-Dickstein et al. “Deep unsupervised learning using nonequilibrium thermodynamics”. In: *International conference on machine learning*. PMLR. 2015, pp. 2256–2265.
- [42] Yang Song and Stefano Ermon. “Generative modeling by estimating gradients of the data distribution”. In: *Advances in neural information processing systems* 32 (2019).
- [43] Yang Song et al. *Score-Based Generative Modeling through Stochastic Differential Equations*. 2021. arXiv: 2011.13456 [cs.LG]. URL: <https://arxiv.org/abs/2011.13456>.
- [44] Yang Song et al. “Score-Based Generative Modeling through Stochastic Differential Equations”. In: *International Conference on Learning Representations (ICLR)*. 2021.
- [45] Yang Song et al. “Score-based generative modeling through stochastic differential equations”. In: *arXiv preprint arXiv:2011.13456* (2020).
- [46] Matthew Tancik et al. *Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains*. 2020. arXiv: 2006.10739 [cs.CV]. URL: <https://arxiv.org/abs/2006.10739>.
- [47] Yi Tay et al. *UL2: Unifying Language Learning Paradigms*. 2023. arXiv: 2205.05131 [cs.CL]. URL: <https://arxiv.org/abs/2205.05131>.
- [48] Arash Vahdat, Karsten Kreis, and Jan Kautz. “Score-based generative modeling in latent space”. In: *Advances in neural information processing systems* 34 (2021), pp. 11287–11302.
- [49] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: <https://arxiv.org/abs/1706.03762>.
- [50] Linting Xue et al. *ByT5: Towards a token-free future with pre-trained byte-to-byte models*. 2022. arXiv: 2105.13626 [cs.CL]. URL: <https://arxiv.org/abs/2105.13626>.

- 
- [51] Jingfeng Yao, Bin Yang, and Xinggang Wang. “Reconstruction vs. generation: Taming optimization dilemma in latent diffusion models”. In: *Proceedings of the Computer Vision and Pattern Recognition Conference*. 2025, pp. 15703–15712.

## A A Reminder on Probability Theory

We present a brief overview of basic concepts from probability theory. This section was partially taken from [26].

### A.1 Random vectors

Consider data in the  $d$ -dimensional Euclidean space  $x = (x^1, \dots, x^d) \in \mathbb{R}^d$  with the standard Euclidean inner product  $\langle x, y \rangle = \sum_{i=1}^d x^i y^i$  and norm  $\|x\| = \sqrt{\langle x, x \rangle}$ . We will consider random variables (RVs)  $X \in \mathbb{R}^d$  with continuous probability density function (PDF), defined as a *continuous* function  $p_X : \mathbb{R}^d \rightarrow \mathbb{R}_{\geq 0}$  providing event  $A$  with probability

$$\mathbb{P}(X \in A) = \int_A p_X(x) dx, \quad (96)$$

where  $\int p_X(x) dx = 1$ . By convention, we omit the integration interval when integrating over the whole space ( $\int \equiv \int_{\mathbb{R}^d}$ ). To keep notation concise, we will refer to the PDF  $p_{X_t}$  of RV  $X_t$  as simply  $p_t$ . We will use the notation  $X \sim p$  or  $X \sim p(X)$  to indicate that  $X$  is distributed according to  $p$ . One common PDF in generative modeling is the  $d$ -dimensional isotropic Gaussian:

$$\mathcal{N}(x; \mu, \sigma^2 I) = (2\pi\sigma^2)^{-\frac{d}{2}} \exp\left(-\frac{\|x - \mu\|_2^2}{2\sigma^2}\right), \quad (97)$$

where  $\mu \in \mathbb{R}^d$  and  $\sigma \in \mathbb{R}_{>0}$  stand for the mean and the standard deviation of the distribution, respectively.

The expectation of a RV is the constant vector closest to  $X$  in the least-squares sense:

$$\mathbb{E}[X] = \arg \min_{z \in \mathbb{R}^d} \int \|x - z\|^2 p_X(x) dx = \int x p_X(x) dx. \quad (98)$$

One useful tool to compute the expectation of *functions of RVs* is the *law of the unconscious statistician*:

$$\mathbb{E}[f(X)] = \int f(x) p_X(x) dx. \quad (99)$$

When necessary, we will indicate the random variables under expectation as  $\mathbb{E}_X f(X)$ .

### A.2 Conditional densities and expectations

Given two random variables  $X, Y \in \mathbb{R}^d$ , their joint PDF  $p_{X,Y}(x, y)$  has marginals

$$\int p_{X,Y}(x, y) dy = p_X(x) \text{ and } \int p_{X,Y}(x, y) dx = p_Y(y). \quad (100)$$

See Figure 21 for an illustration of the joint PDF of two RVs in  $\mathbb{R}$  ( $d = 1$ ). The *conditional* PDF  $p_{X|Y}$  describes the PDF of the random variable  $X$  when conditioned on an event  $Y = y$  with density  $p_Y(y) > 0$ :

$$p_{X|Y}(x|y) := \frac{p_{X,Y}(x, y)}{p_Y(y)}, \quad (101)$$

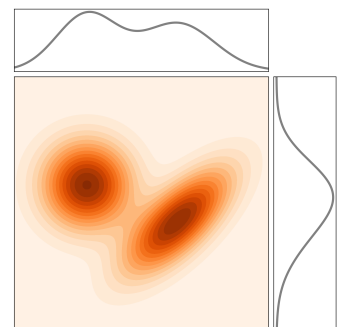


Figure 21: Joint PDF  $p_{X,Y}$  (in shades) and its marginals  $p_X$  and  $p_Y$  (in black lines). Figure from [26]

and similarly for the conditional PDF  $p_{Y|X}$ . Bayes' rule expresses the conditional PDF  $p_{Y|X}$  with  $p_{X|Y}$  by

$$p_{Y|X}(y|x) = \frac{p_{X|Y}(x|y)p_Y(y)}{p_X(x)}, \quad (102)$$

for  $p_X(x) > 0$ .

The *conditional expectation*  $\mathbb{E}[X|Y]$  is the best approximating *function*  $g_*(Y)$  to  $X$  in the least-squares sense:

$$\begin{aligned} g_* &:= \arg \min_{g: \mathbb{R}^d \rightarrow \mathbb{R}^d} \mathbb{E} \left[ \|X - g(Y)\|^2 \right] = \arg \min_{g: \mathbb{R}^d \rightarrow \mathbb{R}^d} \int \|x - g(y)\|^2 p_{X,Y}(x,y) dx dy \\ &= \arg \min_{g: \mathbb{R}^d \rightarrow \mathbb{R}^d} \int \left[ \int \|x - g(y)\|^2 p_{X|Y}(x|y) dx \right] p_Y(y) dy. \end{aligned} \quad (103)$$

For  $y \in \mathbb{R}^d$  such that  $p_Y(y) > 0$  the conditional expectation function is therefore

$$\mathbb{E}[X|Y = y] := g_*(y) = \int x p_{X|Y}(x|y) dx, \quad (104)$$

where the second equality follows from taking the minimizer of the inner brackets in [Equation \(103\)](#) for  $Y = y$ , similarly to [Equation \(98\)](#). Composing  $g_*$  with the random variable  $Y$ , we get

$$\mathbb{E}[X|Y] := g_*(Y), \quad (105)$$

which is a random variable in  $\mathbb{R}^d$ . Rather confusingly, both  $\mathbb{E}[X|Y = y]$  and  $\mathbb{E}[X|Y]$  are often called *conditional expectation*, but these are different objects. In particular,  $\mathbb{E}[X|Y = y]$  is a function  $\mathbb{R}^d \rightarrow \mathbb{R}^d$ , while  $\mathbb{E}[X|Y]$  is a random variable assuming values in  $\mathbb{R}^d$ . To disambiguate these two terms, our discussions will employ the notations introduced here.

The *tower property* is an useful property that helps simplify derivations involving conditional expectations of two RVs  $X$  and  $Y$ :

$$\mathbb{E}[\mathbb{E}[X|Y]] = \mathbb{E}[X] \quad (106)$$

Because  $\mathbb{E}[X|Y]$  is a RV, itself a function of the RV  $Y$ , the outer expectation computes the expectation of  $\mathbb{E}[X|Y]$ . The tower property can be verified by using some of the definitions above:

$$\begin{aligned} \mathbb{E}[\mathbb{E}[X|Y]] &= \int \left( \int x p_{X|Y}(x|y) dx \right) p_Y(y) dy \\ &\stackrel{(101)}{=} \int \int x p_{X,Y}(x,y) dx dy \\ &\stackrel{(100)}{=} \int x p_X(x) dx = \mathbb{E}[X]. \end{aligned}$$

Finally, consider a helpful property involving two RVs  $f(X, Y)$  and  $Y$ , where  $X$  and  $Y$  are two arbitrary RVs. Then, by using the Law of the Unconscious Statistician with [\(104\)](#), we obtain the identity

$$\mathbb{E}[f(X, Y)|Y = y] = \int f(x, y) p_{X|Y}(x|y) dx. \quad (107)$$

## B A Proof of the Fokker-Planck equation

In this section, we give here a self-contained proof of the Fokker-Planck equation which includes the continuity equation as a special case ([Theorem 11](#)). We stress that **this section is not necessary to understand the remainder of this document** and is mathematically more advanced. If you desire to understand where the Fokker-Planck equation comes from, then this section is for you.

### Theorem 41 (Fokker-Planck Equation)

Let  $p_t$  be a probability path with  $p_0 = p_{\text{init}}$  and let us consider the SDE

$$X_0 \sim p_{\text{init}}, \quad dX_t = u_t(X_t)dt + \sigma_t dW_t.$$

Then  $X_t$  has distribution  $p_t$  for all  $0 \leq t \leq 1$  if and only if the [Fokker-Planck equation](#) holds:

$$\partial_t p_t(x) = -\text{div}(p_t u_t)(x) + \frac{\sigma_t^2}{2} \Delta p_t(x) \quad \text{for all } x \in \mathbb{R}^d, 0 \leq t \leq 1, \quad (108)$$

We start by showing that the Fokker-Planck is a necessary condition, i.e. if  $X_t \sim p_t$ , then the Fokker-Planck equation is fulfilled. The trick for the proof is to use [test functions](#)  $f$ , i.e. functions  $f : \mathbb{R}^d \rightarrow \mathbb{R}$  that are infinitely differentiable ("smooth") and are only non-zero within a bounded domain (compact support). We use the fact that for arbitrary integrable functions  $g_1, g_2 : \mathbb{R}^d \rightarrow \mathbb{R}$  it holds that

$$g_1(x) = g_2(x) \text{ for all } x \in \mathbb{R}^d \Leftrightarrow \int f(x)g_1(x)dx = \int f(x)g_2(x)dx \text{ for all test functions } f \quad (109)$$

In other words, we can express the pointwise equality as equality of taking integrals. The useful thing about test functions is that they are smooth, i.e. we can take gradients and higher-order derivatives. In particular, we can use [integration by parts](#) for arbitrary test functions  $f_1, f_2$ :

$$\int f_1(x) \frac{\partial}{\partial x_i} f_2(x) dx = - \int f_2(x) \frac{\partial}{\partial x_i} f_1(x) dx \quad (110)$$

under the condition that  $f_1, f_2$  and their product  $f_1 \cdot f_2$  is integrable. By using this together with the definition of the divergence and Laplacian (see [Equation \(22\)](#)), we get the identities:

$$\int \nabla f_1^T(x) f_2(x) dx = - \int f_1(x) \text{div}(f_2)(x) dx \quad (f_1 : \mathbb{R}^d \rightarrow \mathbb{R}, f_2 : \mathbb{R}^d \rightarrow \mathbb{R}^d) \quad (111)$$

$$\int f_1(x) \Delta f_2(x) dx = \int f_2(x) \Delta f_1(x) dx \quad (f_1 : \mathbb{R}^d \rightarrow \mathbb{R}, f_2 : \mathbb{R}^d \rightarrow \mathbb{R}) \quad (112)$$

Now let's proceed to the proof. We use the stochastic update of SDE trajectories as in [Equation \(6\)](#):

$$X_{t+h} = X_t + h u_t(X_t) + \sigma_t (W_{t+h} - W_t) + h R_t(h) \quad (113)$$

$$\approx X_t + h u_t(X_t) + \sigma_t (W_{t+h} - W_t) \quad (114)$$

where for now we simply ignore the error term  $R_t(h)$  for readability as we will take  $h \rightarrow 0$  anyway. We can then

make the following calculation:

$$\begin{aligned}
f(X_{t+h}) - f(X_t) &\stackrel{(114)}{=} f(X_t + hu_t(X_t) + \sigma_t(W_{t+h} - W_t)) - f(X_t) \\
&\stackrel{(i)}{=} \nabla f(X_t)^T (hu_t(X_t) + \sigma_t(W_{t+h} - W_t)) \\
&\quad + \frac{1}{2} (hu_t(X_t) + \sigma_t(W_{t+h} - W_t))^T \nabla^2 f(X_t) (hu_t(X_t) + \sigma_t(W_{t+h} - W_t)) \\
&\stackrel{(ii)}{=} h \nabla f(X_t)^T u_t(X_t) + \sigma_t \nabla f(X_t)^T (W_{t+h} - W_t) \\
&\quad + \frac{1}{2} h^2 u_t(X_t)^T \nabla^2 f(X_t) u_t(X_t) + h \sigma_t u_t(X_t)^T \nabla^2 f(X_t) (W_{t+h} - W_t) + \\
&\quad + \frac{1}{2} \sigma_t^2 (W_{t+h} - W_t)^T \nabla^2 f(X_t) (W_{t+h} - W_t)
\end{aligned}$$

where in (i) we used a 2nd Taylor approximation of  $f$  around  $X_t$  and in (ii) we used the fact that the Hessian  $\nabla^2 f$  is a symmetric matrix. Note that  $\mathbb{E}[W_{t+h} - W_t | X_t] = 0$  and  $W_{t+h} - W_t | X_t \sim \mathcal{N}(0, hI_d)$ . Therefore

$$\begin{aligned}
&\mathbb{E}[f(X_{t+h}) - f(X_t) | X_t] \\
&= h \nabla f(X_t)^T u_t(X_t) + \frac{1}{2} h^2 u_t(X_t)^T \nabla^2 f(X_t) u_t(X_t) + \frac{h}{2} \sigma_t^2 \mathbb{E}_{\epsilon_t \sim \mathcal{N}(0, I_d)} [\epsilon_t^T \nabla^2 f(X_t) \epsilon_t] \\
&\stackrel{(i)}{=} h \nabla f(X_t)^T u_t(X_t) + \frac{1}{2} h^2 u_t(X_t)^T \nabla^2 f(X_t) u_t(X_t) + \frac{h}{2} \sigma_t^2 \text{trace}(\nabla^2 f(X_t)) \\
&\stackrel{(ii)}{=} h \nabla f(X_t)^T u_t(X_t) + \frac{1}{2} h^2 u_t(X_t)^T \nabla^2 f(X_t) u_t(X_t) + \frac{h}{2} \sigma_t^2 \Delta f(X_t)
\end{aligned}$$

where in (i) we used the fact that  $\mathbb{E}_{\epsilon_t \sim \mathcal{N}(0, I_d)} [\epsilon_t^T A \epsilon_t] = \text{trace}(A)$  and in (ii) we used the definition of the Laplacian and the Hessian matrix. With this, we get that

$$\begin{aligned}
&\partial_t \mathbb{E}[f(X_t)] \\
&= \lim_{h \rightarrow 0} \frac{1}{h} \mathbb{E}[f(X_{t+h}) - f(X_t)] \\
&= \lim_{h \rightarrow 0} \frac{1}{h} \mathbb{E}[\mathbb{E}[f(X_{t+h}) - f(X_t) | X_t]] \\
&= \mathbb{E}[\lim_{h \rightarrow 0} \frac{1}{h} \left( h \nabla f(X_t)^T u_t(X_t) + \frac{1}{2} h^2 u_t(X_t)^T \nabla^2 f(X_t) u_t(X_t) + \frac{h}{2} \sigma_t^2 \Delta f(X_t) \right)] \\
&= \mathbb{E}[\nabla f(X_t)^T u_t(X_t) + \frac{1}{2} \sigma_t^2 \Delta f(X_t)] \\
&\stackrel{(i)}{=} \int \nabla f(x)^T u_t(x) p_t(x) dx + \int \frac{1}{2} \sigma_t^2 \Delta f(x) p_t(x) dx \\
&\stackrel{(ii)}{=} - \int f(x) \text{div}(u_t p_t)(x) dx + \int \frac{1}{2} \sigma_t^2 f(x) \Delta p_t(x) dx \\
&= \int f(x) \left( -\text{div}(u_t p_t)(x) + \frac{1}{2} \sigma_t^2 \Delta p_t(x) \right) dx
\end{aligned}$$

where in (i) we used the assumption that  $p_t$  as the distribution of  $X_t$  and in (ii) we used [Equation \(111\)](#) and [Equation \(112\)](#). Note that to use this, we require integrability of the product  $p_t(x)u_t(x)$ , i.e. such that

$$\int p_t(x) \|u_t(x)\| dx < \infty$$

Note that this condition almost always holds in machine learning (bounded data and functions because of numerical precision limits). Therefore, it holds that

$$\partial_t \mathbb{E}[f(X_t)] = \int f(x) \left( -\operatorname{div}(p_t u_t)(x) + \frac{\sigma_t^2}{2} \Delta p_t(x) \right) dx \quad (\text{for all } f \text{ and } 0 \leq t \leq 1) \quad (115)$$

$$\stackrel{(i)}{\Leftrightarrow} \partial_t \int f(x) p_t(x) dx = \int f(x) \left( -\operatorname{div}(p_t u_t)(x) + \frac{\sigma_t^2}{2} \Delta p_t(x) \right) dx \quad (\text{for all } f \text{ and } 0 \leq t \leq 1) \quad (116)$$

$$\stackrel{(ii)}{\Leftrightarrow} \int f(x) \partial_t p_t(x) dx = \int f(x) \left( -\operatorname{div}(p_t u_t)(x) + \frac{\sigma_t^2}{2} \Delta p_t(x) \right) dx \quad (\text{for all } f \text{ and } 0 \leq t \leq 1) \quad (117)$$

$$\stackrel{(iii)}{\Leftrightarrow} \partial_t p_t(x) = -\operatorname{div}(p_t u_t)(x) + \frac{\sigma_t^2}{2} \Delta p_t(x) \quad (\text{for all } x \in \mathbb{R}^d, 0 \leq t \leq 1) \quad (118)$$

where in (i) we used the assumption that  $X_t \sim p_t$ , in (ii) we swapped the derivative with the integral and (iii) we used [Equation \(109\)](#). This completes the proof that the Fokker-Planck equation is a necessary condition.

Finally, we explain why it is also a sufficient condition. The Fokker-Planck equation is a partial differential equation (PDE). More specifically, it is a so-called *parabolic partial differential equation*. Similar to [Theorem 3](#), such differential equations have a unique solution given fixed initial conditions (see e.g. [15, Chapter 7]). Now, if [Equation \(108\)](#) holds for  $p_t$ , we just shown above that it must also hold for true distribution  $q_t$  of  $X_t$  (i.e.  $X_t \sim q_t$ ) - in other words, both  $p_t$  and  $q_t$  are solutions to the parabolic PDE. Further, we know that the initial conditions are the same, i.e.  $p_0 = q_0 = p_{\text{init}}$  by construction of an interpolating probability path. Hence, by uniqueness of the solution of the differential equation, we know that  $p_t = q_t$  for all  $0 \leq t \leq 1$  - which means  $X_t \sim q_t = p_t$  and which is what we wanted to show.

## C Existence and Uniqueness of Continuous-time Markov chains

We prove [Theorem 33](#) in this section.

*Proof. Uniqueness:* We need to show that there can be only one transition kernel  $p_{t'|t}(X_{t'} = y | X_t = x)$  that satisfies [Equation \(87\)](#). As a first step, we realize that [Equation \(87\)](#) implies that

$$\frac{d}{dt'} p_{t'|t}(X_{t'} = y | X_t = x) \quad (119)$$

$$= \frac{d}{dh} p_{t'+h|t}(X_{t'+h} = y | X_t = x) \Big|_{h=0} \quad (120)$$

$$= \frac{d}{dh} \left[ \sum_{z \in S} p_{t'+h|t'}(X_{t'+h} = y | X_{t'} = z) p_{t'|t}(X_{t'} = z | X_t = x) \right] \Big|_{h=0} \quad (121)$$

$$= \sum_{z \in S} Q_{t'}(y|z) p_{t'|t}(X_{t'} = z | X_t = x) \quad (122)$$

For fixed  $x, t$ , one can consider  $t' \mapsto p_{t'|t}(X_{t'} = y | X_t = x)$  as a vector-valued function and the above is a linear ODE of that function (the Kolmogorov forward equation, in fact, see [Proposition 2](#)) with a known initial condition, i.e.  $p_{t|t}(X_t = y | X_t = x) = \delta_y(x)$ . As we know, every linear ODE has a unique solution (see [Theorem 3](#)), therefore  $p_{t'|t}(X_{t'} = y | X_t = x)$  must also be unique.

**Existence:** Conversely, any linear ODE has a solution, i.e. we know that for every  $x, t$  there is a  $p_{t'|t}(X_{t'} = y|X_t = x)$  such that

$$p_{t|t}(X_t = y|X_t = x) = \delta_y(x) \quad (123)$$

$$\frac{d}{dt'} p_{t'|t}(X_{t'} = y|X_t = x) = \sum_{z \in S} Q_{t'}(y|z) p_{t'|t}(X_{t'} = z|X_t = x) \quad (124)$$

For  $t' = t$ , this implies Equation (87) in particular. It remains to show that  $p_{t'|t}(X_{t'} = y|X_t = x)$  is a valid transition kernel in this case, i.e. the following 3 properties must hold:

$$\sum_{y \in S} p_{t'|t}(X_{t'} = y|X_t = x) = 1 \quad (125)$$

$$p_{t'|t}(X_{t'} = y|X_t = x) \geq 0 \quad (126)$$

$$\sum_{z \in S} p_{t_2|t_1}(X_{t_2} = y|X_{t_1} = z) p_{t_1|t_0}(X_{t_1} = z|X_{t_0} = x) = p_{t_2|t_0}(y|x) \quad (127)$$

To the first property, one can observe that it holds for  $t' = t$  by Equation (123) and that

$$\frac{d}{dt'} \sum_{y \in S} p_{t'|t}(X_{t'} = y|X_t = x) \quad (128)$$

$$= \sum_{y \in S} \frac{d}{dt'} p_{t'|t}(X_{t'} = y|X_t = x) \quad (129)$$

$$= \sum_{z \in S} \left[ \sum_{y \in S} Q_{t'}(y|z) \right] p_{t'|t}(X_{t'} = z|X_t = x) \quad (130)$$

$$= 0 \quad (131)$$

where we used the fact that the columns of rate matrices sum to 0. To show the second property, note that it holds at time  $t' = t$ . Further, whenever  $p_{t'|t}(X_{t'} = y|X_t = x) = 0$ , it must hold that

$$\begin{aligned} \frac{d}{dt'} p_{t'|t}(X_{t'} = y|X_t = x) &= \sum_{z \neq y} \underbrace{Q_{t'}(y|z)}_{\geq 0} p_{t'|t}(X_{t'} = z|X_t = x) \\ &\geq 0 \end{aligned}$$

Therefore, whenever  $p_{t'|t}(X_{t'} = y|X_t = x) = 0$ , it can only increase. Therefore,  $p_{t'|t}(X_{t'} = y|X_t = x)$  will never be negative.

To show the third property, define  $q_{t_2|t_0}(y|x)$  to be

$$q_{t_2|t_0}(y|x) = \sum_{z \in S} p_{t_2|t_1}(X_{t_2} = y|X_{t_1} = z) p_{t_1|t_0}(X_{t_1} = z|X_{t_0} = x)$$

---

Then we know that

$$q_{t_2=t_1|t_0}(y|x) = \sum_{z \in S} \delta_y(z) p_{t_1|t_0}(X_{t_1} = z | X_{t_0} = x) = p_{t_1|t_0}(X_{t_1} = y | X_{t_0} = x)$$

and

$$\begin{aligned} \frac{d}{dt_2} q_{t_2|t_0}(y|x) &= \sum_{z \in S} \frac{d}{dt_2} p_{t_2|t_1}(X_{t_2} = y | X_{t_1} = z) p_{t_1|t_0}(X_{t_1} = z | X_{t_0} = x) \\ &= \sum_{z \in S} \sum_{\tilde{z} \in S} Q_{t_2}(y|\tilde{z}) p_{t_2|t_1}(X_{t_2} = \tilde{z} | X_{t_1} = z) p_{t_1|t_0}(X_{t_1} = z | X_{t_0} = x) \\ &= \sum_{\tilde{z} \in S} Q_{t_2}(y|\tilde{z}) \left[ \sum_{z \in S} p_{t_2|t_1}(X_{t_2} = \tilde{z} | X_{t_1} = z) p_{t_1|t_0}(X_{t_1} = z | X_{t_0} = x) \right] \\ &= \sum_{\tilde{z} \in S} Q_{t_2}(y|\tilde{z}) q_{t_2|t_0}(\tilde{z}|x) \end{aligned}$$

This shows that  $p_{t_2|t_0}(z|x)$  and  $q_{t_2|t_0}(z|x)$  fulfill the same ODE. Hence, it must hold

$$\sum_{z \in S} p_{t_2|t_1}(X_{t_2} = y | X_{t_1} = z) p_{t_1|t_0}(X_{t_1} = z | X_{t_0} = x) = q_{t_2|t_0}(y|x) = p_{t_2|t_0}(y|x)$$

This shows the third property. So  $p_{t'|t}(y|x)$  is indeed the transition kernel satisfying [Equation \(87\)](#). This finishes the proof.  $\square$

## D Additional Perspectives on VAEs

In this section, we expand on the treatment of VAEs presented in the main text and provide a variational derivation of the total VAE loss from Equation (83). As a first step, notice that both the encoder and decoder give rise to a joint distribution over both  $x$  and the latent  $z$ , viz.,

$$\begin{aligned} q_\phi(x, z) &= p_{\text{data}}(x)q_\phi(\cdot|x) && \text{(encoder joint)} \\ p_\theta(x, z) &= p_\theta(x|z)p_{\text{prior}}(z) && \text{(decoder joint)} \end{aligned}$$

We might therefore conceptualize training the VAE as learning  $\phi$  and  $\theta$  so that the encoder and decoder joint distributions are reasonably similar. We can do this via the KL-divergence of the joint latent and data distribution:

$$\begin{aligned} D_{\text{KL}}(q_\phi(x, z) \parallel p_\theta(x, z)) &= D_{\text{KL}}(p_{\text{data}}(x)q_\phi(z|x) \parallel p_\theta(x|z)p_{\text{prior}}(z)) \\ &= \mathbb{E}_{\blacksquare} \left[ \log \left( \frac{p_{\text{data}}(x)q_\phi(z|x)}{p_\theta(x|z)p_{\text{prior}}(z)} \right) \right] \\ &= \mathbb{E}_{\blacksquare} [\log p_{\text{data}}(x)] + \mathbb{E}_{\blacksquare} \left[ \log \left( \frac{q_\phi(z|x)}{p_{\text{prior}}(z)} \right) \right] - \mathbb{E}_{\blacksquare} [\log p_\theta(x|z)] \end{aligned} \tag{132}$$

$\blacksquare = x \sim p_{\text{data}}(x) \ z \sim q_\phi(z|x).$

Let us now examine each of the three remaining terms in turn. First, we find that

$$\mathbb{E}_{\blacksquare} [\log p_{\text{data}}(x)] = \mathbb{E}_{x \sim p_{\text{data}}(x)} [\log p_{\text{data}}(x)] = C, \tag{133}$$

for some constant  $C$  independent of  $\phi$  and  $\theta$ . Next, we find that

$$\mathbb{E}_{\blacksquare} \left[ \log \left( \frac{q_\phi(z|x)}{p_{\text{prior}}(z)} \right) \right] = \mathbb{E}_{x \sim p_{\text{data}}(x)} [D_{\text{KL}}(q_\phi(z|x) \parallel p_{\text{prior}}(z))] \tag{134}$$

encourages  $q_\phi(z|x)$  to resemble the prior  $p_{\text{prior}}(z)$ . Finally, we find that

$$-\mathbb{E}_{x \sim p_{\text{data}}(x) \ z \sim q_\phi(z|x)} [\log p_\theta(x|z)] \tag{135}$$

corresponds the average negative log-likelihood, and thus serves as to minimize the reconstruction loss. Ignoring the constant term, we combine the **prior penalty** and **reconstruction** terms to obtain that the VAE loss is actually simply the KL-divergence in joint data and latent space:

$$\mathcal{L}_{\text{VAE}}(\phi, \theta) = \underbrace{\mathbb{E}_{x \sim p_{\text{data}}(x)} [D_{\text{KL}}(q_\phi(z|x) \parallel p_{\text{prior}}(z))]}_{\text{prior enforcement loss}} - \underbrace{\mathbb{E}_{x \sim p_{\text{data}}(x) \ z \sim q_\phi(z|x)} [\log p_\theta(x|z)]}_{\text{reconstruction loss}} \tag{136}$$

$$= D_{\text{KL}}(q_\phi(x, z) \parallel p_\theta(x, z)) + \text{const} \tag{137}$$

Therefore, we can interpret the VAE as a KL-divergence in the joint space of latents and images.

**VAEs as generative models.** We now explain how one could interpret VAEs as generative models. We could generate a sample by setting  $z \sim p_{\text{prior}} = \mathcal{N}(0, I_k)$  and sampling  $x \sim p_\theta(\cdot|z)$  from the decoder. The resulting

distribution that we would get is given by:

$$p_\theta(x) = \int_z p_\theta(x|z)p_{\text{prior}}(z)dz$$

We now want to demonstrate that the VAE learns to approximately sample from  $p_\theta$ . To show this, we need the following result:

**Proposition 3 (Chain rule)**

Let  $q(x, z), p(x, z)$  be distributions over two variables  $x \in \mathbb{R}^{l_1}, z \in \mathbb{R}^{l_2}$ . Then, it holds that:

$$D_{\text{KL}}(q(z, x) \parallel p(z, x)) = D_{\text{KL}}(q(x) \parallel p(x)) + \mathbb{E}_{x \sim q} [D_{\text{KL}}(q(z|x) \parallel p(z|x))].$$

In particular, as the second summand is non-negative due to [Equation \(76\)](#), we obtain the data-processing inequality

$$D_{\text{KL}}(q(x) \parallel p(x)) \leq D_{\text{KL}}(q(z, x) \parallel p(z, x)). \tag{138}$$

*Proof.*

$$\begin{aligned} D_{\text{KL}}(q(z, x) \parallel p(z, x)) &= \mathbb{E}_q \left[ \log \frac{q(z, x)}{p(z, x)} \right] \\ &= \mathbb{E}_{(x, z) \sim q} \left[ \log \frac{q(z|x) q(x)}{p(z|x) p(x)} \right] \\ &= \mathbb{E}_{(x, z) \sim q} \left[ \log \frac{q(z|x)}{p(z|x)} \right] + \mathbb{E}_{x \sim q} \left[ \log \frac{q(x)}{p(x)} \right] \\ &= D_{\text{KL}}(q(x) \parallel p(x)) + \mathbb{E}_{x \sim q} [D_{\text{KL}}(q(z|x) \parallel p(z|x))] \end{aligned}$$

where we have repeatedly applied the definition of KL divergence. □

By [Proposition 3](#), we can now show that

$$\mathcal{L}_{\text{VAE}}(\phi, \theta) = D_{\text{KL}}(q_\phi(x, z) \parallel p_\theta(x, z)) + \text{const} \geq D_{\text{KL}}(p_{\text{data}}(x) \parallel p_\theta(x)) + \text{const} \tag{139}$$

where we used the fact the  $x$ -marginal of  $q_\phi(x, z)$  is  $p_{\text{data}}$ . In other words, the VAE loss minimizes an upper bound on the KL-divergence between the data distribution  $p_{\text{data}}$  and the distribution generated by the VAE. Hence, we can look at VAEs as generative models in their own right. In the same way, we can show that

$$\mathcal{L}_{\text{VAE}}(\phi, \theta) = D_{\text{KL}}(q_\phi(x, z) \parallel p_\theta(x, z)) + \text{const} \geq D_{\text{KL}}(q_\phi(z) \parallel p_{\text{prior}}(z)) + \text{const} \tag{140}$$

In other words, the VAE objective minimize an upper bound to the KL-divergence between latent distribution and the prior.

**Why not stop at VAEs?** Per the discussion above, VAEs can be realized as generative models in their own right, with the encoder simply existing to facilitate the training of a complementary decoder which transforms

a Gaussian into the desired data distribution. Samples could then be obtained by sampling  $z \sim p_{\text{prior}}$  and then  $x \sim p_{\theta}(x|z)$ . Why then, we do insist on training a separate generative model within the learned latent space? The answer has to do with the so-called **amortization gap** between the left and right hand sides of both [Equation \(139\)](#) and [Equation \(140\)](#), corresponding precisely to the gap in the information processing inequality. This gap is zero if and only if  $q_{\phi}(z|x) = p_{\theta}(z|x)$ , in which case the encoder represents the true posterior. Thus, while e.g.,  $D_{\text{KL}}(q_{\phi}(x, z) \parallel p_{\theta}(x, z))$  is minimized implies  $D_{\text{KL}}(q_{\phi}(z) \parallel p_{\text{prior}}(z))$  is minimized (see [Equation \(140\)](#)), a decrease in the former does not necessarily imply an equal decrease in the latter. Consequently, at the end of training, it is simultaneously true that both  $D_{\text{KL}}(q_{\phi}(x, z) \parallel p_{\theta}(x, z))$  and the amortization gap

$$D_{\text{KL}}(q_{\phi}(x, z) \parallel p_{\theta}(x, z)) - D_{\text{KL}}(q_{\phi}(z) \parallel p_{\text{prior}}(z)) \quad (141)$$

are not completely minimized, so that  $q_{\phi}(z) \neq p_{\text{prior}}(z)$ . Finally, observe that during training, the decoder learns to reconstruct from  $q_{\phi}(z)$  rather than  $p_{\text{prior}}(z)$ , so that switching to reconstructing from  $p_{\text{prior}}(z)$  during inference would amount to going out of distribution from training. In practice however, this mismatch is a feature rather than a bug. Practice has shown flow and diffusion models to be more capable models in general than the convolutional stacks used to implement the VAE decoder, so that it makes sense to farm off some of the generative complexity to the latent generative model. We return to this line of discussion later on in the discussion. Additionally, and beyond the scope of these notes, variational formulations of diffusion and flow models realize these modeling families as VAEs in their own right.

**The evidence lower bound.** Properly rearranged, the terms within [Equation \(132\)](#) can present various complementary perspectives. One is the so-called **evidence lower bound**, which we extract as follows. Observe that for fixed  $x$

$$\begin{aligned} \mathbb{E}_{z \sim q_{\phi}(z|x)} \left[ \log \left( \frac{q_{\phi}(z|x)}{p_{\theta}(x|z)p_{\text{prior}}(z)} \right) \right] &= \mathbb{E}_{z \sim q_{\phi}(z|x)} \left[ \log \left( \frac{q_{\phi}(z|x)}{p_{\theta}(z|x)} \right) \right] - \log p_{\theta}(x) \\ &= D_{\text{KL}}(q_{\phi}(z|x) \parallel p_{\theta}(z|x)) - \log p_{\theta}(x) \end{aligned} \quad (142)$$

where the first equality is obtained from

$$p_{\theta}(z|x) = \frac{p_{\theta}(x|z)p_{\text{prior}}(z)}{p_{\theta}(x)}.$$

We may thus rearrange [Equation \(142\)](#) to obtain

$$\mathbb{E}_{z \sim q_{\phi}(z|x)} \left[ \log \left( \frac{p_{\theta}(x|z)p_{\text{prior}}(z)}{q_{\phi}(z|x)} \right) \right] + D_{\text{KL}}(q_{\phi}(z|x) \parallel p_{\theta}(z|x)) = \log p_{\theta}(x), \quad (143)$$

from which it follows that

$$\underbrace{\mathbb{E}_{z \sim q_{\phi}(z|x)} \left[ \log \left( \frac{p_{\theta}(x|z)p_{\text{prior}}(z)}{q_{\phi}(z|x)} \right) \right]}_{\triangleq \text{ELBO}(x; \phi, \theta)} \leq \underbrace{\log p_{\theta}(x)}_{\text{evidence}}. \quad (144)$$

The left-hand side is therefore commonly referred to as the evidence lower bound, or ELBO. We may now rewrite  $\mathcal{L}_{\text{VAE}}$  from Equation (136) in terms of the ELBO via

$$\begin{aligned}
\mathcal{L}_{\text{VAE}} &= D_{\text{KL}}(q_\phi(x, z) \parallel p_\theta(x, z)) + \text{const} \\
&= \mathbb{E}_{x \sim p_{\text{data}}} \mathbb{E}_{z \sim q_\phi(z|x)} \left[ \log \left( \frac{p_{\text{data}}(x) q_\phi(z|x)}{p_\theta(x|z) p_{\text{prior}}(z)} \right) \right] + \text{const} \\
&= \mathbb{E}_{x \sim p_{\text{data}}} [\log p_{\text{data}}(x) - \text{ELBO}(x; \phi, \theta)] + \text{const} \\
&= -\mathbb{E}_{x \sim p_{\text{data}}} [\text{ELBO}(x; \phi, \theta)] - \underbrace{H(p_{\text{data}})}_{\text{const}} + \text{const} \\
&= -\mathbb{E}_{x \sim p_{\text{data}}} [\text{ELBO}(x; \phi, \theta)] + \text{const}
\end{aligned} \tag{145}$$

so that the original VAE objective can be seen as simply trying to maximize the expected ELBO. Finally, let's consider what occurs in the limit that we train our VAE perfectly.

**Remark 42** (What Happens When  $q_\phi(x, z) \approx p_\theta(x, z)$ ?)

First, note that the sampling distribution used to train our latent generative model is given by the marginal

$$q_\phi(z) = \int_x q_\phi(z|x) p_{\text{data}}(x) dx.$$

If  $q_\phi(x, z) = p_\theta(x, z)$ , then in particular

$$q_\phi(z) = p_\theta(z) = p_{\text{prior}}(z).$$

Thus,  $q_\phi(x, z) \approx p_\theta(x, z)$  **implies regularization of the latent sampling distribution**. Second,  $q_\phi(x, z) \approx p_\theta(x, z)$  implies that the variational approximation  $p_\theta(x|z) \approx q_\phi(x|z)$  is good, and in turn **implies low reconstruction error**.

**Remark 43** (What's Variational About VAEs?)

Why can't we simply take  $q_\phi(\cdot|x) = p_\theta(\cdot|x)$ , thereby guaranteeing  $q_\phi(x, z) = p_\theta(x, z) = 0$ ? The reason is that while we know the **likelihood**  $p_\theta(x|z)$ , the **posterior**

$$p_\theta(z|x) = \frac{p_\theta(x|z) p_{\text{prior}}(z)}{p_\theta(x)}$$

is generally intractable, as we lack access to the likelihood  $p_\theta(x)$ . The presence of *variational* in VAE is thus due to the fact that  $q_\phi(\cdot|x)$  serves as a substitute, or **variational approximation**, of the intractable posterior  $p_\theta(\cdot|x)$ .

**Reconstruction vs Generation.** Given an encoder  $q_\phi(z|x)$ , decoder  $p_\theta(x|z)$ , and latent generative model  $r_\psi$  trained to sample from  $q_\phi(z)$ , we may consider the following two generative models

$$r_{\psi,\theta}^{\text{recon}}(x_{\text{out}}) = \int_{z, x_{\text{in}}} p_\theta(x_{\text{out}} | z) q_\phi(z | x_{\text{data}}) p_{\text{data}}(x_{\text{data}}) dz dx_{\text{in}} \quad (\text{reconstruction sampler})$$

$$r_{\psi,\phi}^{\text{gen}}(x_{\text{out}}) = \int_{z_{\text{gen}}} p_\theta(x_{\text{out}} | z_{\text{gen}}) r_\psi(z_{\text{gen}}) dz_{\text{gen}} \quad (\text{generative sampler})$$

In other words, the reconstruction sampler starts at  $x_{\text{data}} \in p_{\text{data}}$  encodes to  $z$ , and decodes to  $x_{\text{out}}$ , while the generative sampler starts from  $z_{\text{gen}} \in r_\psi$  from the generative model, and then passes through the decoder. By computing the Fréchet inception distance of the two respective samplers’ distributions to  $p_{\text{data}}$ , we obtain the **reconstruction-FID** (rFID) and **generative-FID** (gFID). One might also consider measuring the quality of the reconstruction sampler via the average **distortion** (root mean square error of reconstruction), although such a metric would not make sense for the generative sampler. As it turns out, there is a natural tension between the quality of the reconstruction sampler, and the quality of the generative sampler. Low rFID (a high quality reconstruction sampler) generally indicates low information loss in the latent, so that the latent distribution  $q_\phi(z)$  largely resembles  $p_{\text{data}}$ , and so that the task of learning the latent generative model is likely more difficult, raising gFID. Conversely, high rFID generally indicates high information loss, and an easier latent distribution  $q_\phi(z)$  to learn, thereby lowering gFID. This phenomena is visualized in [Figure 22](#).

**The Division of Labor.** The reconstruction-generative sampler tradeoff forces us to consider how information loss should be divided up between the autoencoder and the latent generative model  $r_\psi$ . Intuitively,  $r_\psi$ , via some learned vector field  $u_t^\psi(z_t)$ , transports a standard Gaussian to  $q_\phi(z) \approx p_{\text{prior}}$ , after which the decoder  $p_\theta(x|z)$  transports  $q_\phi(z)$  to  $p_{\text{data}}$ . Let us now (imprecisely) define the **rate** as the degree to which the latent distribution  $q_\phi(z)$  matches the  $p_{\text{prior}}(z)$ , and by extension, the degree to which the task of generation is farmed off to the latent generative model.<sup>5</sup> This division of labor can be visualized by plotting the Pareto frontier between rate and distortion, as shown in [Figure 22](#). In particular, when the rate is high, the distortion is low, and vice versa, offering a second perspective on the preceding discussion of reconstruction versus generation sampler quality. We culminate our discussion in the following insight.

**Intuition 44** (The Division of Labor)

The key insight from [Figure 22](#) is that an optimal division of labor exists at the “knee” of the Pareto frontier, at which point we obtain low rate (high compression!) without high distortion. In other words, such a point corresponds to a level of compression which simultaneously reduces the difficulty of training the underlying generative model while preserving reasonable reconstruction quality.

## E A Guide to the Diffusion Model Literature

There is a whole family of models around diffusion models and flow matching in the literature. When you read these papers, you will likely find a different (but equivalent) way of presenting the material from this class. This

<sup>5</sup>We defer a more technical discussion of the rate to the next subsection.

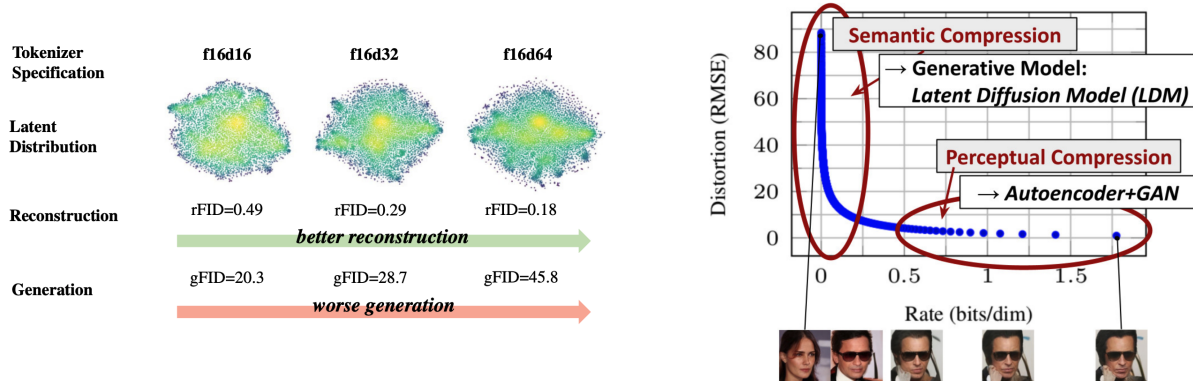


Figure 22: Right: The tradeoff between between gFID and rFID, figure taken from [51]. Here,  $f$  denotes the downsampling factor, and  $d$  denotes the latent channel dimension. Right: Distortion (reconstruction quality) vs rate, taken from [17, 37]. We remark that this particular curve was generated using a DDPM (itself a type of VAE). While certain technical subtleties in the distortion and rate computations may differ from the imprecise definition presented in this text, the overall intuition remains the same.

makes it sometimes a little confusing to read these papers. For this reason, we want to give a brief overview over various frameworks and their differences and put them also in their historical context. **This is not necessary to understand the remainder of this document** but rather intended to be a support for you in case you read the literature.

**Discrete time vs. continuous time.** The first denoising diffusion model papers [41, 42, 17] did not use SDEs but constructed Markov chains in **discrete time**, i.e. with time steps  $t = 0, 1, 2, 3, \dots$ . To this date, you will find a lot of works in the literature working with this discrete-time formulation. While this construction is appealing due to its simplicity, the disadvantage of the time-discrete approach is that it forces you to choose a time discretization before training. Further, the loss function needs to be approximated via an **evidence lower bound (ELBO)** - which is, as the name suggests, only a lower bound to the loss we actually want to minimize. Later, Song et al. [45] showed that these constructions were essentially an approximation of a time-continuous SDEs. Further, the ELBO loss becomes tight (i.e. it is not a lower bound anymore) in the continuous time case (e.g. note that [Theorem 12](#) and [Theorem 22](#) are equalities and not lower bounds - this would be different in the discrete time case). This made the SDE construction popular because it was considered mathematically "cleaner" and that one could control the simulation error via ODE/SDE samplers post training. It is important to note however that both models employ the same loss and are *not* fundamentally different.

**"Forward process" vs probability paths.** The first wave of denoising diffusion models [41, 42, 17, 45] did not use the term **probability path** but constructed a noising procedure of a data point  $z \in \mathbb{R}^d$  via a so-called **forward process**. This is an SDE of the form

$$\bar{X}_0 = z, \quad d\bar{X}_t = u_t^{\text{forw}}(\bar{X}_t)dt + \sigma_t^{\text{forw}}d\bar{W}_t \quad (146)$$

The idea is that after drawing a data point  $z \sim p_{\text{data}}$  one simulates the forward process and thereby corrupts or "noises" the data. The forward process is designed such that for  $t \rightarrow \infty$  its distribution converges to a Gaussian  $\mathcal{N}(0, I_d)$ . In other words, for  $T \gg 0$  it holds that  $\bar{X}_T \sim \mathcal{N}(0, I_d)$  approximately. Note that this essentially corresponds to a probability path: the conditional distribution of  $\bar{X}_t$  given  $\bar{X}_0 = z$  is a conditional probability path  $\bar{p}_t(\cdot|z)$  and the distribution of  $\bar{X}_t$  marginalized over  $z \sim p_{\text{data}}$  corresponds to a marginal probability path  $\bar{p}_t$ .<sup>6</sup> However, note that with this construction, we need to know the distribution of  $X_t|X_0 = z$  in closed form in order to train our models to avoid simulating the SDE. This essentially restricts the vector field  $u_t^{\text{forw}}$  to ones such that we know the distribution  $\bar{X}_t|\bar{X}_0 = z$  in closed form. Therefore, throughout the diffusion model literature, vector fields in forward processes are always of the affine form, i.e.  $u_t^{\text{forw}}(x) = a_t x$  for some continuous function  $a_t$ . For this choice, we can use known formulas of the conditional distribution [40, 44, 23]:

$$\bar{X}_t|\bar{X}_0 = z \sim \mathcal{N}(\alpha_t z, \beta_t^2 I), \quad \alpha_t = \exp\left(\int_0^t a_r dr\right), \quad \beta_t^2 = \alpha_t^2 \int_0^t \frac{(\sigma_r^{\text{forw}})^2}{\alpha_r^2} dr$$

Note that these are simply Gaussian probability paths. Therefore, one can say that **a forward process is a specific way of constructing a (Gaussian) probability path**. The term probability path was introduced by flow matching [25] to both simplify the construction and make it more general at the same time: First, the "forward process" of diffusion models is never actually simulated (only samples from  $\bar{p}_t(\cdot|z)$  are drawn during training). Second, a forward process only converges for  $t \rightarrow \infty$  (i.e. we will never arrive at  $p_{\text{init}}$  in finite time). Therefore, we choose to use probability paths in this document.

**Time-Reversals vs Solving the Fokker-Planck equation.** The original description of diffusion models did not construct the training target  $u_t^{\text{target}}$  or  $\nabla \log p_t$  via the Fokker-Planck equation (or Continuity equation) but via a **time-reversal** of the forward process [2]. A time-reversal  $(X_t)_{0 \leq t \leq T}$  is an SDE with the same distribution over trajectories inverted in time, i.e.

$$\mathbb{P}[\bar{X}_{t_1} \in A_1, \dots, \bar{X}_{t_n} \in A_n] = \mathbb{P}[X_{T-t_1} \in A_1, \dots, X_{T-t_n} \in A_n] \quad (147)$$

$$\text{for all } 0 \leq t_1, \dots, t_n \leq T, \text{ and } A_1, \dots, A_n \subset S \quad (148)$$

As shown in Anderson [2], one can obtain a time-reversal satisfying the above condition by the SDE:

$$dX_t = [-u_t(X_t) + \sigma_t^2 \nabla \log p_t(X_t)] dt + \sigma_t dW_t, \quad u_t(x) = u_{T-t}^{\text{forw}}(x), \sigma_t = \bar{\sigma}_{T-t}$$

As  $u_t(X_t) = a_t X_t$ , the above corresponds to a specific instance of training target we derived in [Proposition 1](#) (this is not immediately trivial as different time conventions are used. See e.g. [26] for a derivation). However, for the purposes of generative modeling, we often only use the final point  $X_1$  of the Markov process (e.g., as a generated image) and discard earlier time points. Therefore, whether a Markov process is a "true" time-reversal or follows along a probability path does not matter for many applications. Therefore, using a time-reversal is not necessary and often leads to suboptimal results, e.g. the probability flow ODE is often better [23, 28]. All ways of sampling from a diffusion models that are different from the time-reversal rely again on using the Fokker-Planck equation. We hope that this illustrates why nowadays many people construct the training targets directly via the

<sup>6</sup>Note however that they use an **inverted time convention**:  $\bar{p}_0(\cdot|z) = p_{\text{data}}$  here.

---

Fokker-Planck equation - as pioneered by [25, 27, 1] and done in this class.

**Flow Matching [25] and Stochastic Interpolants [1].** The framework that we present is most closely related to the frameworks of flow matching and **stochastic interpolants (SIs)**. As we learnt, flow matching restricts itself to flows. In fact, one of the key innovations of flow matching was to show that one does not need a construction via a forward process and SDEs but flow models alone can be trained in a scalable manner. Due to this restriction, you should keep in mind that sampling from a flow matching model will be deterministic (only the initial  $X_0 \sim p_{\text{init}}$  will be random). Stochastic interpolants included both the pure flow and the SDE extension via "Langevin dynamics" that we use here (see [Theorem 17](#)). Stochastic interpolants get their name from a **interpolant function**  $I(t, x, z)$  intended to interpolate between two distributions. In the terminology we use here, this corresponds to a different yet (mainly) equivalent way of constructing a conditional and marginal probability path. The advantage of flow matching and stochastic interpolants over diffusion models is both their simplicity and their generality: their training framework is very simple but at the same time they allow you to go from an arbitrary distribution  $p_{\text{init}}$  to an arbitrary distribution  $p_{\text{data}}$  - while denoising diffusion models only work for Gaussian initial distributions and Gaussian probability path. This opens up new possibilities for generative modeling that we will touch upon briefly later in this class.

#### **Summary 45** (Alternative Diffusion Formulations)

Alternative formulations for diffusion models that are popular in the literature often involve some combination of the following elements:

1. **Discrete-time:** Approximations of SDEs via discrete-time Markov chains are often used.
2. **Inverted time convention:** It is popular to use an inverted time convention where  $t = 0$  corresponds to  $p_{\text{data}}$  (as opposed to here where  $t = 0$  corresponds to  $p_{\text{init}}$ ).
3. **Forward process:** Forward processes (or noising processes) are ways of constructing (Gaussian) probability paths.
4. **Training target via time-reversal:** A training target can also be constructed via the time-reversal of SDEs. This is a specific instance of the construction presented here (with an inverted time convention).