An Introduction to Flow Matching and Diffusion Models

Peter Holderrieth and Ezra Erives

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

1 Introduction

> Creating noise from data is easy; creating data from noise is generative modeling.
>
> Song et al. [29]

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** [31] and **flow matching** [15, 17, 1, 16]. 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 became 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, our goal is to provide a self-contained introduction to the necessary mathematical toolbox regarding differential equations to enable you to systematically understand these models. 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. Sections 1-6 represent the core "canonical" ingredients for diffusion models, while sections 7 and 8 are advanced topics and optional.

- **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.
- **Section 6, Large-Scale Image and Video Generators:** 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 ?? (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 **??**.

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_{data}. Mathematically, one can think of p_{data} as a **probability density**, i.e. a function $p_{\text{data}} : \mathbb{R}^d \to \mathbb{R}_{\geq 0}$ 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_{data}. With this, we can mathematically express the task of generation as sampling from the (unknown) distribution p_{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_{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_{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_{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_{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 class, 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_{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_{data} after training.

2 Flow and Diffusion Models

In the previous section, we formalized generative modeling as sampling from a data distribution p_{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

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

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

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

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:

$$\frac{\mathrm{d}}{\mathrm{d}t} X_t = u_t(X_t) \qquad \blacktriangleright \text{ ODE} \tag{1a}$$

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

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

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

$$\frac{\mathrm{d}}{\mathrm{d}t} \psi_t(x_0) = u_t(\psi_t(x_0)) \qquad \blacktriangleright \text{ flow ODE} \tag{2b}$$

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

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

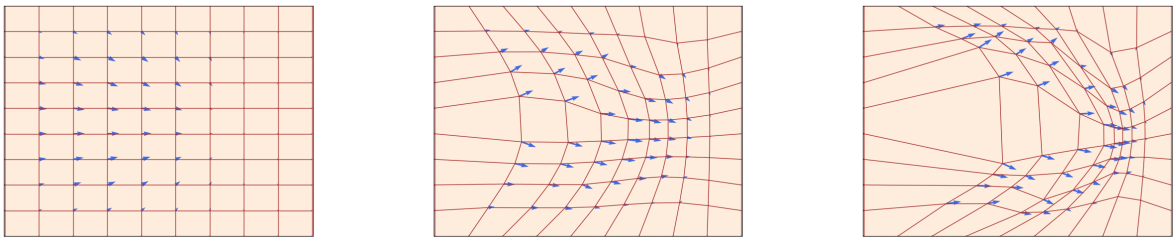

Figure 1: A flow $\psi_t : \mathbb{R}^d \to \mathbb{R}^d$ (red square grid) is defined by a velocity field $u_t : \mathbb{R}^d \to \mathbb{R}^d$ (visualized with blue arrows) that prescribes its instantaneous movements at all locations (here, $d = 2$). We show three different times t. As one can see, a flow is a diffeomorphism that "warps" space. Figure from [16].

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

Theorem 3 (Flow existence and uniqueness)
If $u : \mathbb{R}^d \times [0, 1] \to \mathbb{R}^d$ is continuously differentiable with a bounded derivative, then the ODE in (2) has a unique solution given by a flow ψ_t. In this case, ψ_t is a **diffeomorphism** for all t, i.e. ψ_t is continuously differentiable with a continuously differentiable inverse ψ_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 [22, 3].

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

$$\frac{\mathrm{d}}{\mathrm{d}t}\psi_t(x_0) \overset{(3)}{=} \frac{\mathrm{d}}{\mathrm{d}t}\left(\exp(-\theta t)\, x_0\right) \overset{(i)}{=} -\theta \exp(-\theta t)\, x_0 \overset{(3)}{=} -\theta\psi_t(x_0) = u_t(\psi_t(x_0)),$$

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 ψ_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 [11]. 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

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

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

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

Intuitively, the 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^θ. For now, we simply mean that u_t^θ is a parameterized function $u_t^\theta : \mathbb{R}^d \times [0,1] \to \mathbb{R}^d$ with parameters θ. 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_{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 simple make the initial condition X_0 random. Specifically, we choose an **initial distribution** p_{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

$$X_0 \sim p_{\text{init}} \qquad \blacktriangleright \text{ random initialization}$$
$$\frac{\mathrm{d}}{\mathrm{d}t} X_t = u_t^\theta(X_t) \qquad \blacktriangleright \text{ ODE}$$

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

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

where ψ_t^θ describes the flow induced by u_t^θ. Note however: although it is called *flow model*, **the neural network parameterizes the vector field, not the flow** (at least for now). 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^θ, 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: $\quad X_{t+h} = X_t + hu_t^\theta(X_t)$
6: $\quad$ 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

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

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 $0 \leq t_0 < t_1 < \cdots < t_n = 1$, the increments $W_{t_1} - W_{t_0}, \ldots, W_{t_n} - W_{t_{n-1}}$ are independent random variables.

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

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

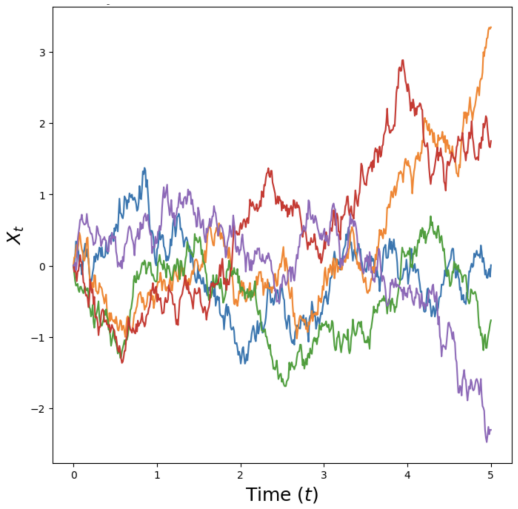

Figure 2: Sample trajectories of a Brownian motion W_t in dimension $d = 1$ simulated using Equation (5).

In Figure 8, 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

[1] Nobert Wiener was a famous mathematician who taught at MIT. You can still see his portraits hanging at the MIT math department.

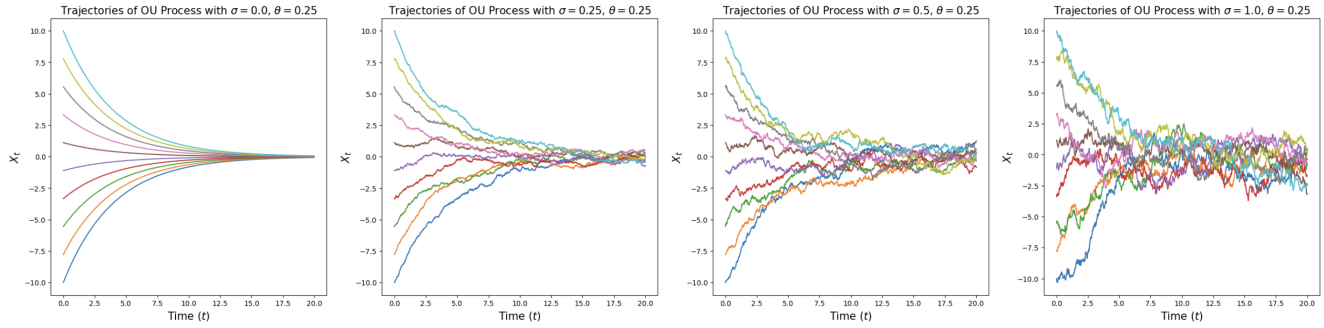

Figure 3: Illustration of Ornstein-Uhlenbeck processes (Equation (8)) in dimension $d = 1$ for $\theta = 0.25$ and various choices of σ (increasing from left to right). For $\sigma = 0$, we recover a flow (smooth, deterministic trajectories) that converges to the origin as $t \to \infty$. For $\sigma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

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 $(X_t)_{0 \le t \le 1}$ of an ODE as follows:

$$\begin{aligned} & \frac{\mathrm{d}}{\mathrm{d}t} X_t = u_t(X_t) && \blacktriangleright \text{ expression via derivatives} \\ \overset{(i)}{\Leftrightarrow} \quad & \frac{1}{h}\left(X_{t+h} - X_t\right) = u_t(X_t) + R_t(h) \\ \Leftrightarrow \quad & X_{t+h} = X_t + h u_t(X_t) + h R_t(h) && \blacktriangleright \text{ expression via infinitesimal updates} \end{aligned}$$

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

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

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

$$\mathrm{d}X_t = u_t(X_t)\mathrm{d}t + \sigma_t \mathrm{d}W_t \qquad \blacktriangleright \text{ SDE} \tag{7a}$$

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

However, always keep in mind that the "$\mathrm{d}X_t$"-notation above is a purely informal notation of Equation (6). Unfortunately, SDEs do not have a flow map ϕ_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 $u : \mathbb{R}^d \times [0,1] \to \mathbb{R}^d$ is continuously differentiable with a bounded derivative and σ_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 [19] 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

$$\mathrm{d}X_t = -\theta X_t \mathrm{d}t + \sigma dW_t. \tag{8}$$

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 (as I always go the inverse direction of where I am), while the diffusion coefficient σ always adds more noise. This process converges towards a Gaussian distribution $\mathcal{N}(0, \sigma^2/(2\theta))$ if we simulate it for $t \to \infty$. 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

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

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_{init} into a complex distribution p_{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 - a neural network

Algorithm 2 Sampling from a Diffusion Model (Euler-Maruyama method)

Require: Neural network u_t^θ, number of steps n, diffusion coefficient σ_t
Set $t = 0$
Set step size $h = \frac{1}{n}$
Draw a sample $X_0 \sim p_{\text{init}}$
for $i = 1, \dots, n$ **do**
  Draw a sample $\epsilon \sim \mathcal{N}(0, I_d)$
  $X_{t+h} = X_t + h u_t^\theta(X_t) + \sigma_t \sqrt{h} \epsilon$
  Update $t \leftarrow t + h$
end for
return X_1

u_t^θ. A **diffusion model** is thus given by

$$X_0 \sim p_{\text{init}} \quad \blacktriangleright \text{ random initialization}$$
$$\mathrm{d}X_t = u_t^\theta(X_t)\mathrm{d}t + \sigma_t \mathrm{d}W_t \quad \blacktriangleright \text{ SDE}$$

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^θ with parameters θ that parameterize a vector field and a fixed diffusion coefficient σ_t:

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

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

Initialization: $X_0 \sim p_{\text{init}}$ ▶ Initialize with simple distribution, e.g. a Gaussian
Simulation: $\mathrm{d}X_t = u_t^\theta(X_t)\mathrm{d}t + \sigma_t \mathrm{d}W_t$ ▶ Simulate SDE from 0 to 1
Goal: $X_1 \sim p_{\text{data}}$ ▶ Goal is to make X_1 have distribution p_{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^θ. However, we have not yet discussed how to train them. i.e. how to optimize the parameters θ such that generative model returns something sensible, e.g. a nice-looking image or exciting video. Next, we discuss **flow matching** [15, 1, 17], a algorithm to train u_t^θ 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^θ and obtain samples from the generative model by simulating the ODE

$$X_0 \sim p_{\text{init}}, \quad \mathrm{d}X_t = u_t^\theta(X_t)\mathrm{d}t \qquad \text{(Flow model)} \tag{10}$$

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_{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 θ such that simulating the flow model in Equation (10) results in samples from the data distribution $X_1 \sim p_{\text{data}}$?**

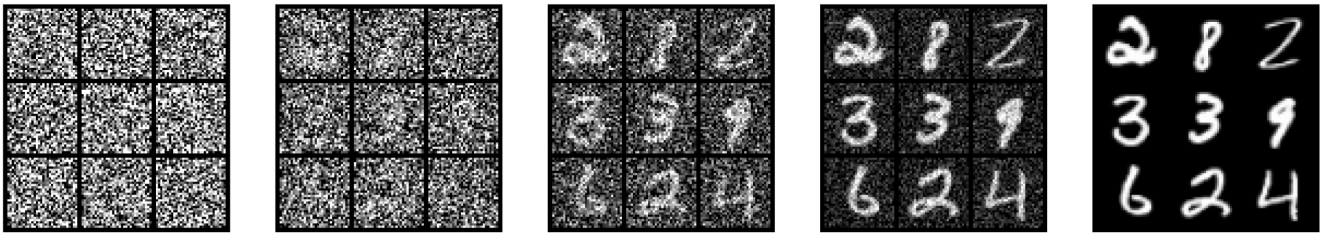

Figure 4: Gradual interpolation from noise to data via a Gaussian conditional probability path for a collection of images. Note that each image is a data point of dimension $d = 32 \times 32$, so we are plotting individual samples from the probability path, while in Figure 5 we plot the distribution 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_{init} and data p_{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 δ_z the **Dirac delta** "distribution". This is the simplest distribution that one can imagine: sampling from δ_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:

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

In other words, a conditional probability path gradually converts the initial distribution p_{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.

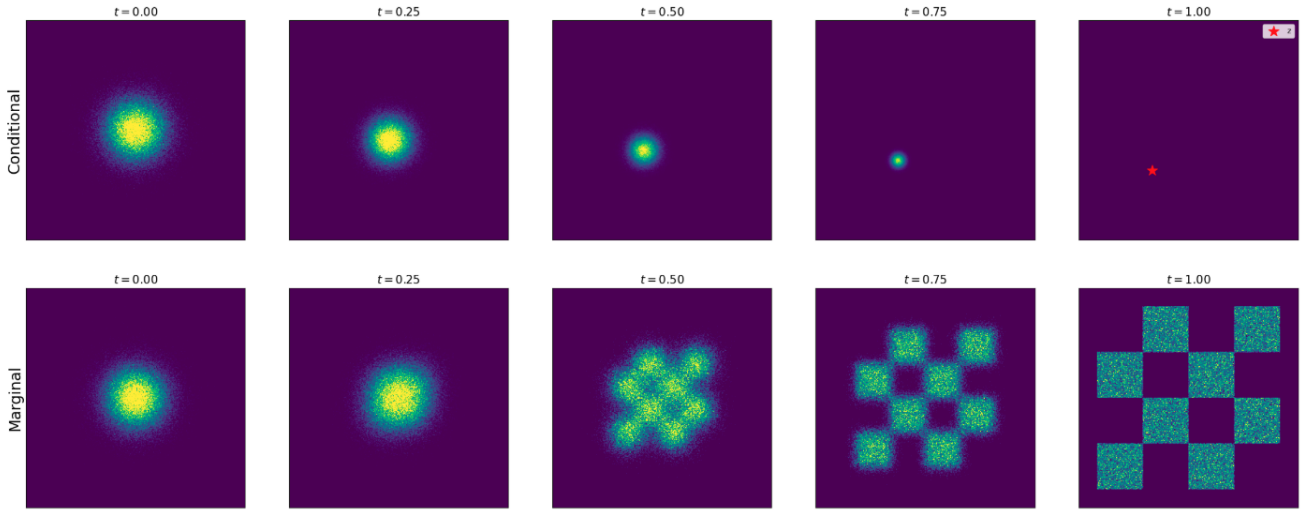

Figure 5: Illustration of a conditional (top) and marginal (bottom) probability path. Here, we plot a Gaussian probability path with $\alpha_t = t, \beta_t = 1 - t$. The conditional probability path interpolates a Gaussian $p_{\text{init}} = \mathcal{N}(0, I_d)$ and $p_{\text{data}} = \delta_z$ for single data point z. The marginal probability path interpolates a Gaussian and a data distribution p_{data} (Here, p_{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 $p_t(\cdot|z)$:

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

$$p_t(x) = \int p_t(x|z) p_{\text{data}}(z) \mathrm{d}z \qquad \blacktriangleright \text{ density of marginal path} \tag{13}$$

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_{init} and p_{data}:

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

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 denoising diffusion models**. Let α_t, β_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

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

which, by the conditions we imposed on α_t and β_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 δ_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:

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

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 $(p_t)_{0 \le t \le 1}$ specified what distributions $X_t \sim p_t$ the points X_t along a trajectory *should* have. At this point, this is just 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

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

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_{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

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

follows the marginal probability path, i.e.

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

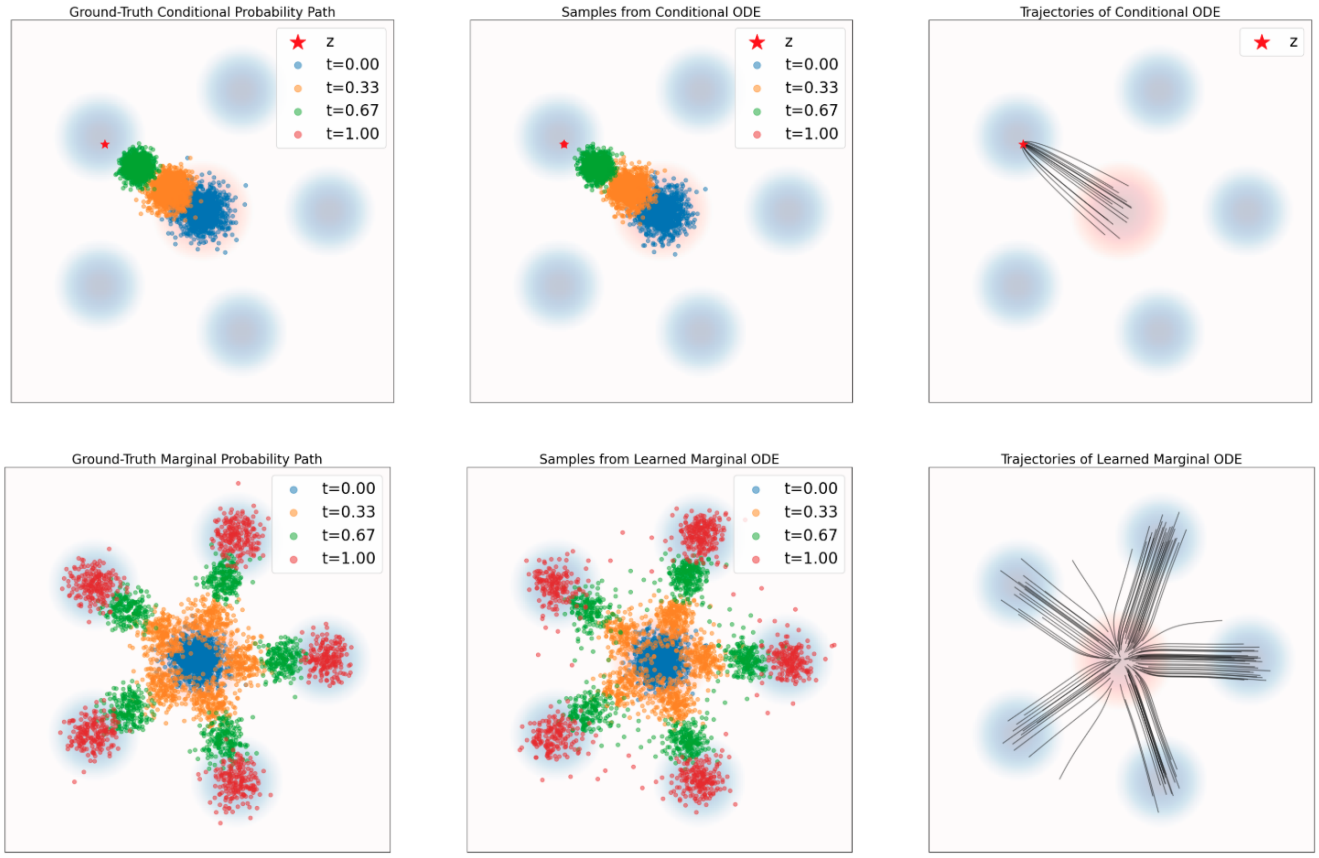

Figure 6: Illustration of Theorem 9. Simulating a probability path with ODEs. Data distribution p_{data} in blue background. Gaussian p_{init} in red background. Top row: Conditional probability path. Left: Ground truth samples from conditional path $p_t(\cdot|z)$. Middle: ODE samples over time. Right: 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.

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

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 α_t, β_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 α_t and β_t. Here, we want to show that the **conditional Gaussian vector field** given by

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

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

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

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

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

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

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

where in (i) we used the definition of $\psi_t^{\text{target}}(x|z)$ (Equation (21)), in (ii) we reparameterized $x \to (x - \alpha_t z)/\beta_t$, 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.[a] □

[a] 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 a 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 div as

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

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^{target} with $X_0 \sim p_{\text{init}}$. Then $X_t \sim p_t$ for all $0 \leq t \leq 1$ if and only if

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

where $\partial_t p_t(x) = \frac{\mathrm{d}}{\mathrm{d}t} p_t(x)$ 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^{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^{target}, as defined as in Equation (18), satisfies the continuity equation. We can do this by direct calculation:

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

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^{target}. By Theorem 11, this is enough to imply Equation (19), and we are done. □

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^θ such that it equals the marginal vector field u_t^{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

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

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

where $p_t(x) = \int p_t(x|z)p_{\text{data}}(z)\text{d}z$ 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^{target} by Theorem 9, we cannot compute it efficiently. 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**

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

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 θ. 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 $\mathcal{L}_{\text{FM}}(\theta)$ with in the same fashion. In particular, **for the minimizer θ^* 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).

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

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

where (i) holds by definition, in (ii) we used the formula $\|a-b\|^2 = \|a\|^2 - 2a^Tb + \|b\|^2$, 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:

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

where in (i) we expressed the expected value as an integral, in (ii) we use **??**, 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:

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

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

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^θ has been trained, we may simulate the flow model

$$\mathrm{d}X_t = u_t^\theta(X_t)\,\mathrm{d}t, \qquad X_0 \sim p_{\text{init}} \tag{27}$$

via e.g., Algorithm 1 to obtain samples $X_1 \sim p_{\text{data}}$. This whole pipeline is called **flow matching** in the literature [15, 17, 1, 16]. Let us now instantiate the conditional flow matching loss for the choice of Gaussian probability paths:

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

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

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

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

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

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

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

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 ϵ 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 it in a simple example and in Algorithm 3 we summarize the training procedure.

Let us summarize the results of this section.

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^θ

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

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

6: Compute loss

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

7: Update $\theta \leftarrow \text{grad\_update}(\mathcal{L}(\theta))$.

8: **end for**

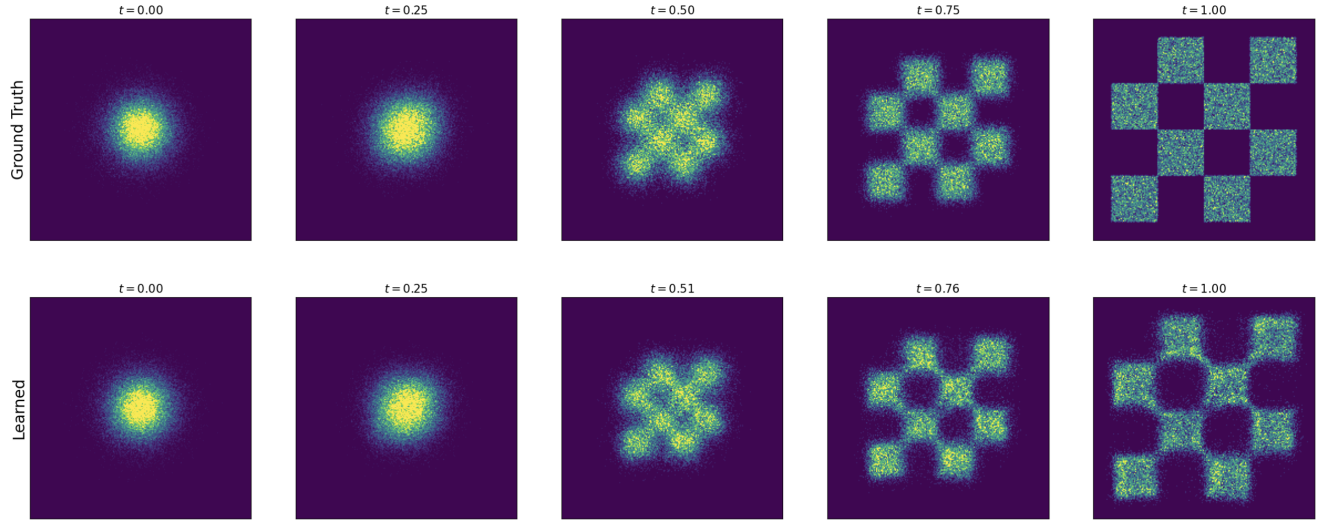

Figure 7: Illustration of Theorem 12 with a Gaussian CondOT probability path: simulating an ODE from a trained flow matching model. The data distribution is the chess board pattern (top right). Top row: Histogram from ground truth marginal probability path $p_t(x)$. Bottom row: Histogram of samples from 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.

Summary 14 (Flow Matching)

Flow matching training consists of learning the marginal vector field u_t^{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^{target} satisfies the continuity equation. Then the **marginal vector field** defined by

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

follows the marginal probability path, i.e.,

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

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

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

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

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

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

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

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 [31, 30] 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.

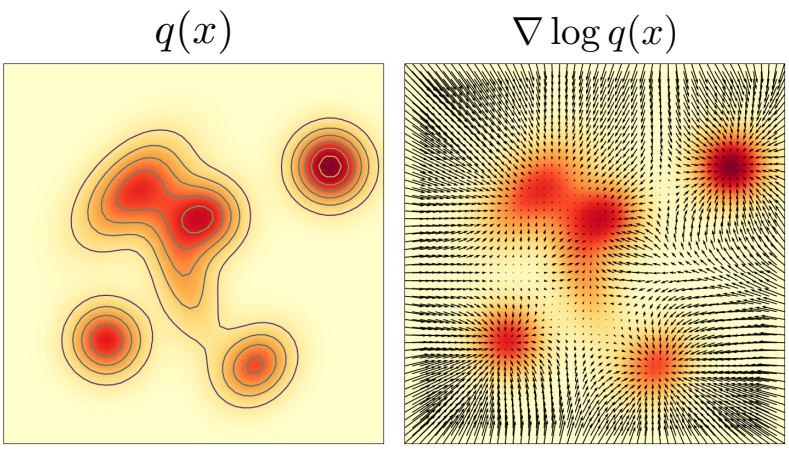

Figure 8: Illustration of score function $\nabla \log q(x)$ plotted as black rows (right) of a general probability distribution $q(x)$ (left).

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 $\nabla p_t(x|z)$ via

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

Hence, **the relation between the conditional and marginal score is analogous to the relation between the conditional and marginal vector field**. This import result suggests that it might be possible to develop a flow-matching-like loss for the score function, an idea we'll revisit momentarily. Note that we can prove Equation (38) via

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

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 (87)) to get

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

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

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

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

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:

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

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:

$$\begin{aligned} u^{\text{target}}(x) = \int u_t^{\text{target}}(x|z) \frac{p_t(x|z) p_{\text{data}}(z)}{p_t(x)} \mathrm{d}z &= \int \left[a_t \nabla \log p_t(x|z) + b_t x\right] \frac{p_t(x|z) p_{\text{data}}(z)}{p_t(x)} \mathrm{d}z \\ &\overset{(i)}{=} a_t \nabla \log p_t(x) + b_t x \end{aligned}$$

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

Proposition 1 is striking because it says that once we've learned u_t^{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 of the equation depend only on x, the constants $\alpha_t, \dot{\alpha}_t, \beta_t, \dot{\beta}_t$, and the posterior mean $\mathbb{E}_{z|x}[z]$ (see Equation (38)). Since both x and the constants are easily available, it follows that any third quantity from which $\mathbb{E}_{z|x}[z]$ may be recovered 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

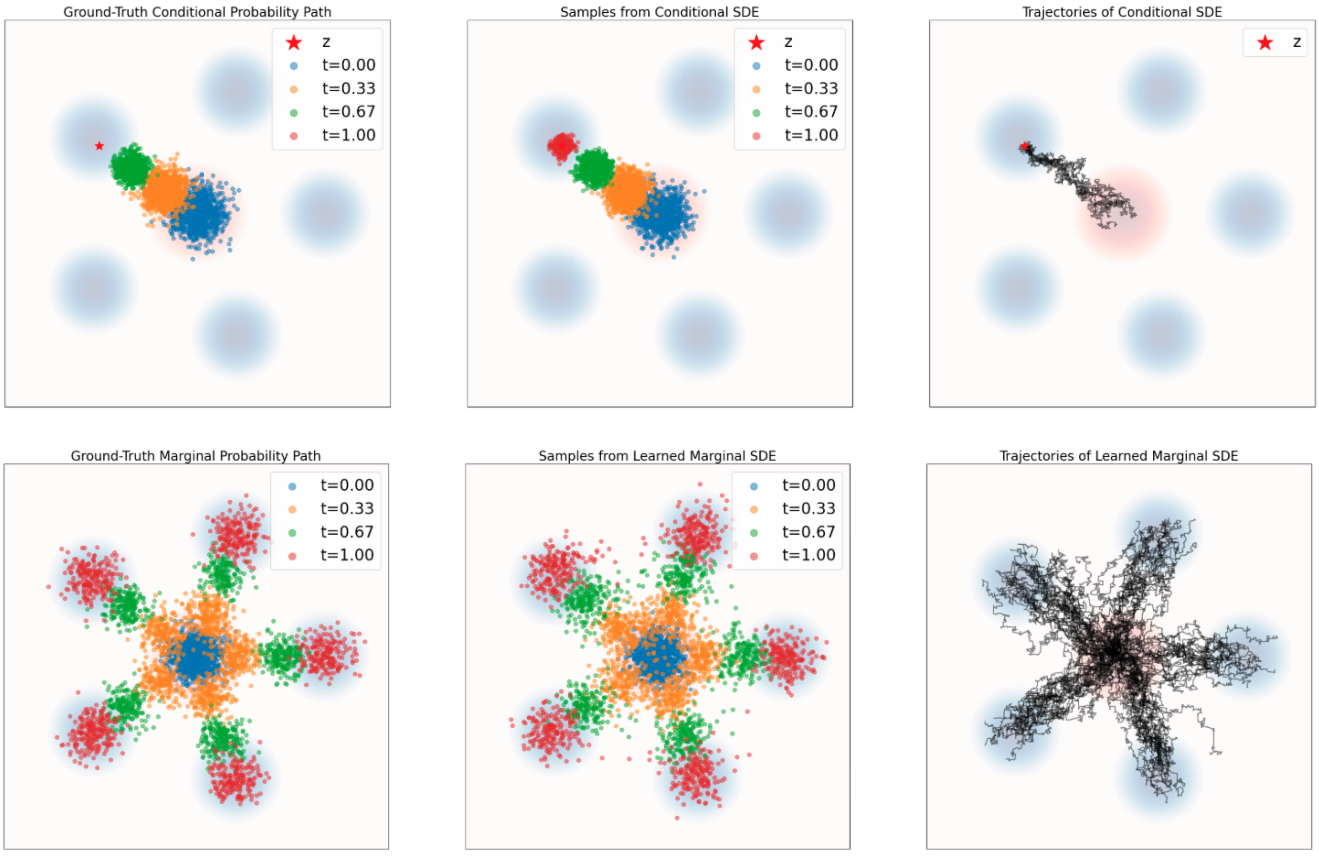

Figure 9: Illustration of Theorem 17. Simulating a probability path with SDEs. This repeats the plots from Figure 6 with SDE sampling using Equation (46). Data distribution p_{data} in blue background. Gaussian p_{init} in red background. Top row: Conditional path. Bottom row: Marginal probability path. As one can see, the SDE transports samples from p_{init} into samples from δ_z (for the conditional path) and to p_{data} (for the marginal path).

itself, often referred to as the *denoiser*. Formally, we define the **conditional and marginal denoiser** as

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

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.[a] People often call such models **denoising diffusion models** as learning D_t and learning u_t^{target} are theoretically equivalent.

[a] 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^{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:

$$X_0 \sim p_{\text{init}}, \qquad \mathrm{d}X_t = u_t^{\text{target}}(X_t)\mathrm{d}t + \frac{\sigma_t^2}{2}\nabla \log p_t(X_t)\mathrm{d}t + \sigma_t \mathrm{d}W_t \tag{44}$$

$$= \left[u_t^{\text{target}}(X_t) + \frac{\sigma_t^2}{2}\nabla \log p_t(X_t)\right]\mathrm{d}t + \sigma_t \mathrm{d}W_t$$

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

In particular, $X_1 \sim p_{\text{data}}$ for this SDE. 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 σ_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 [13, 1, 18].[2]

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:

$$X_0 \sim p_{\text{init}}, \quad \mathrm{d}X_t = \left[\left(a_t + \frac{\sigma_t^2}{2}\right)\nabla \log p_t(X_t) + b_t X_t\right]\mathrm{d}t + \sigma_t \mathrm{d}W_t \tag{46}$$

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

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 Δ via

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

[2] Again, we stress that the existence of a "best σ_t" is an artifact of imperfectly trained models and finite compute budgets rather than a theoretical statement about the dynamics in their continuous limit.

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

Theorem 19 (Fokker-Planck Equation)
Let p_t be a probability path and let us consider the SDE

$$X_0 \sim p_{\text{init}}, \quad \mathrm{d}X_t = u_t(X_t)\mathrm{d}t + \sigma_t \mathrm{d}W_t.$$

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

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

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 Δp_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 that the SDE defined in Equation (46) satisfies the Fokker-Planck equation for p_t. We can do this by direction calculation:

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

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 (46) 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. □

Remark 20 (Langevin Dynamics, Optional)
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

$$\mathrm{d}X_t = \frac{\sigma_t^2}{2}\nabla \log p(X_t)\mathrm{d}t + \sigma_t dW_t, \tag{50}$$

which is commonly known as **Langevin dynamics**. The fact that p_t is constant implies that $\partial_t p_t(x) = 0$. It

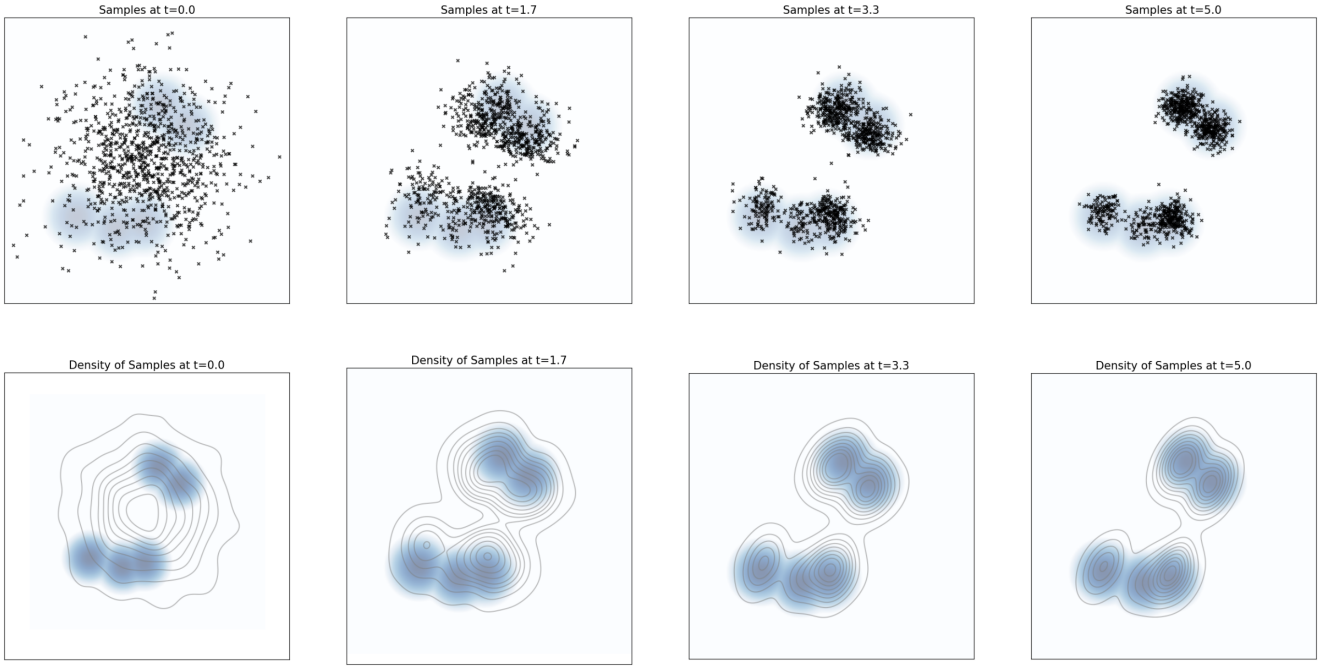

Figure 10: Top row: Particles evolving under the Langevin dynamics given by Equation (50), with $p(x)$ taken to be a Gaussian mixture with 5 modes. Bottom row: A kernel density estimate of the same samples shown in the top row. As one can see, the distribution of samples converges to the equilibrium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

follows immediately from Theorem 17 that these dynamics satisfy the Fokker-Planck equation for the static path $p_t = p$ in Theorem 17. Therefore, we may conclude that p is a stationary distribution of Langevin dynamics, so that

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

As with many Markov processes, these dynamics converge to the stationary distribution p under rather general conditions (see Section 4.2). That is, if we instead we take $X_0 \sim p' \neq p$, so that $X_t \sim p'_t$, then under mild conditions $p_t \to p$. 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 serves, and serve as the basis for initial formulations of diffusion models. The Langevin dynamics also have elegant connections to the theories of gradients flows and optimal transport, both of which are beyond the scope of these notes.

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** $s_t^\theta : \mathbb{R}^d \times [0,1] \to \mathbb{R}^d$. In the same way as before, we can design a **score matching** loss and a **denoising score matching** loss:

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

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

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 21

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 θ. Therefore, their gradients coincide:

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

In particular, for the minimizer θ^*, 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^{target} (Equation (18)). Therefore, the proof is identical to the proof of Theorem 12 replacing u_t^{target} with $\nabla \log p_t$. □

Example 22 (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:

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

where in (i) we plugged in Equation (28) and replaced x by $\alpha_t z + \beta_t \epsilon$. Note that the network s_t^θ 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 Probabilitic Models**, [9]) it was therefore proprosed to drop the constant $\frac{1}{\beta_t^2}$ in the loss and reparameterize s_t^θ into a **noise predictor** network $\epsilon_t^\theta : \mathbb{R}^d \times [0,1] \to \mathbb{R}^d$ via:

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

As before, the network ϵ_t^θ 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^θ or noise predictor ϵ_t^θ

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

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

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

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

Let us summarize the results of this section:

Summary 23 (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:

$$X_0 \sim p_{\text{init}}, \quad \mathrm{d}X_t = \left[u_t^{\text{target}}(X_t) + \frac{\sigma_t^2}{2}\nabla \log p_t(X_t)\right]\mathrm{d}t + \sigma_t \mathrm{d}W_t \tag{52}$$

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

where is $u_t^{\text{target}}(x)$ 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^θ and train it via **denoising score matching**

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

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^θ and u_t^θ separately as we can convert them via the formula:

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

After training, we can simulate the following SDE

$$X_0 \sim p_{\text{init}}, \quad \mathrm{d}X_t = \left[\left(1 + \frac{\sigma_t^2}{2b_t}\right) u_t^\theta(X_t) - \frac{\sigma_t^2 a_t}{2b_t} X_t\right] \mathrm{d}t + \sigma_t \mathrm{d}W_t \tag{55}$$

$$= \left[\left(a_t + \frac{\sigma_t^2}{2}\right) s_t^\theta(X_t) + b_t X_t\right] \mathrm{d}t + \sigma_t \mathrm{d}W_t \tag{56}$$

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 24 (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 σ_t, together given by

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

Notice the difference from summary 7: we are additionally guiding u_t^θ 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:

Initialization:	$X_0 \sim p_{\text{init}}$	▶ Initialize with simple distribution (such as a Gaussian)
Simulation:	$\mathrm{d}X_t = u_t^\theta(X_t\|y)\mathrm{d}t + \sigma_t \mathrm{d}W_t$	▶ Simulate SDE from $t = 0$ to $t = 1$.
Goal:	$X_1 \sim p_{\text{data}}(\cdot\|y)$	▶ 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 ="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 [10].

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

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

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**

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

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 Classifer-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.

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 α_t and β_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 $\nabla \log p_t(x|y)$

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

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

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

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

where we used that the gradient ∇ 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

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

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** [4, 29]. 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** [10] that 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 $\varnothing$ label that denotes **the absence of conditioning**. We can then treat $u_t^{\text{target}}(x) = u_t^{\text{target}}(x|\varnothing)$. 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) [10].

Remark 25 (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 $y = \varnothing$. The challenge is that when sampling $(z, y) \sim p_{\text{data}}$, we will never obtain $y = \varnothing$. It follows that we must introduce the possibility of $y = \varnothing$ artificially. To do so, we will define some hyperparameter η to be the probability that we discard the original label y, and replace it with $\varnothing$. We thus arrive at our **CFG conditional flow matching training objective**

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

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

We summarize our findings below.

Summary 26 (Classifier-Free Guidance for Flow Models)
Given the unguided marginal vector field $u_t^{\text{target}}(x|\varnothing)$, 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

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

By approximating $u_t^{\text{target}}(x|\varnothing)$ and $u_t^{\text{target}}(x|y)$ using the same neural network, we may leverage the following **classifier-free guidance CFM** (CFG-CFM) objective, given by

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

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

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. η, $y \leftarrow \varnothing$	▶ Replace y with $\varnothing$ with probability η.
$\widehat{\mathcal{L}_{\text{CFM}}^{\text{CFG}}}(\theta) = \|u_t^\theta(x\|y) - u_t^{\text{target}}(x\|z)\|^2$	▶ Regress model against conditional vector field.

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

Initialization:	$X_0 \sim p_{\text{init}}(x)$	▶ Initialize with simple distribution (such as a Gaussian)
Simulation:	$\mathrm{d}X_t = \tilde{u}_t^\theta(X_t\|y)\mathrm{d}t$	▶ Simulate ODE from $t=0$ to $t=1$.
Samples:	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 [10]. Similarly, in Figure 12, we visualize the affect of various guidance scales w when applying classifier-free guidance to sampling from the MNIST dataset of handwritten digits.

Remark 27 (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.

Figure 12: 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!

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^θ

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: $y \leftarrow \varnothing$
7: Compute loss

$$\mathcal{L}(\theta) = \|u_t^\theta(x|y) - (\dot{\alpha}_t \epsilon + \dot{\beta}_t z)\|^2$$

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

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_{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 θ 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 architectural choices for such a model: the **U-Net** [27, 9, 12, 4], and the **diffusion transformer** (DiT) [5, 20, 18].

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 [32]. Explicitly, the featurization is given by

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

where the frequencies w_i are set in the following way

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

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 $y_{\text{raw}} \in \mathcal{Y} \triangleq \{0, \ldots, N\}$ 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_{raw}, and set y to this embedding vector.

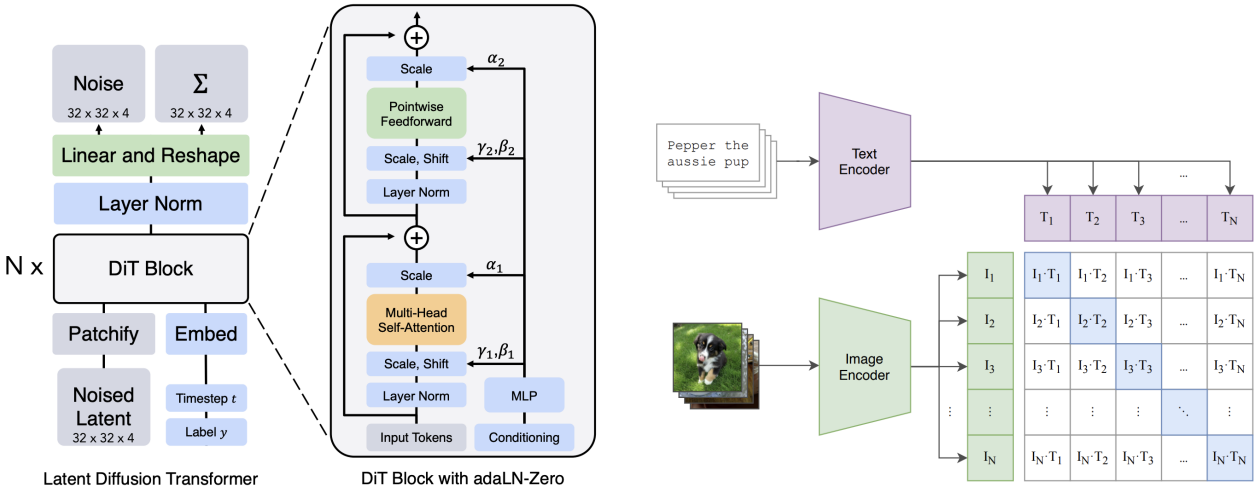

Figure 13: Left: An overview of the diffusion transformer architecture, taken from [20]. Right: A schematic of the contrastive CLIP loss, in which a shared image-text embedding space is learned, taken from [24].

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_{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 [24]. 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 [7, 23]. 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_{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 [35, 20, 18]. 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 the patches to obtain a sequence of tokens, and process the resulting tokens via standard attention [6]. 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 $i = 0, \cdots, L-1$ via transformer layers in a **DitBlock** (see Remark 28 for details):

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

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 28 (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, and refer the reader elsewhere for guidance on current best practices and conventions. 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 $V \in \mathbb{R}^{M \times d_h}$,

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

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 $h \in \{1, \ldots, n_{\text{heads}}\}$, learn projection matrices $W_Q^{(h)}, W_K^{(h)}, W_V^{(h)} \in \mathbb{R}^{k \times d_h}$. Define

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

where the *source sequence* z is either

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

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

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

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 [21]. Concretely, let $g : \mathbb{R}^d \to \mathbb{R}^m$ be an MLP and set

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

where $\gamma, \beta \in \mathbb{R}^d$ (or, depending on the implementation, separate (γ, β) 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 $\mathrm{Norm}(\cdot)$ (e.g. LayerNorm), define the modulated normalization

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

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

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

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

where the MLP is a position-wise feed-forward network, and the $g_{\ldots}$ 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 [27] 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 [9, 12, 4]. 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**.[3] For sake of example, let us walk through the path taken by an image $x_t \in \mathbb{R}^{3\times256\times256}$ (we have taken $(C_{\text{input}}, H, W) = (3, 256, 256)$) as it is processed by the U-Net:

$$
\begin{aligned}
x_t^{\text{input}} &\in \mathbb{R}^{3\times256\times256} && \blacktriangleright \text{ Input to the U-Net.} \\
x_t^{\text{latent}} &= \mathcal{E}(x_t^{\text{input}}) \in \mathbb{R}^{512\times32\times32} && \blacktriangleright \text{ Pass through encoders to obtain latent.} \\
x_t^{\text{latent}} &= \mathcal{M}(x_t^{\text{latent}}) \in \mathbb{R}^{512\times32\times32} && \blacktriangleright \text{ Pass latent through midcoder.} \\
x_t^{\text{output}} &= \mathcal{D}(x_t^{\text{latent}}) \in \mathbb{R}^{3\times256\times256} && \blacktriangleright \text{ Pass through decoders to obtain output.}
\end{aligned}
$$

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 $x_t^{\text{input}} \in \mathbb{R}^{3\times256\times256}$ 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 14. At a high level, most U-Nets involve some 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 14).

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 1024×1024 image with three RGB color channels corresponds to a total dimension of $d = H \cdot W \cdot 3 \approx 3 * 10^6$! 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, and not unlike the U-Net

[3] 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.

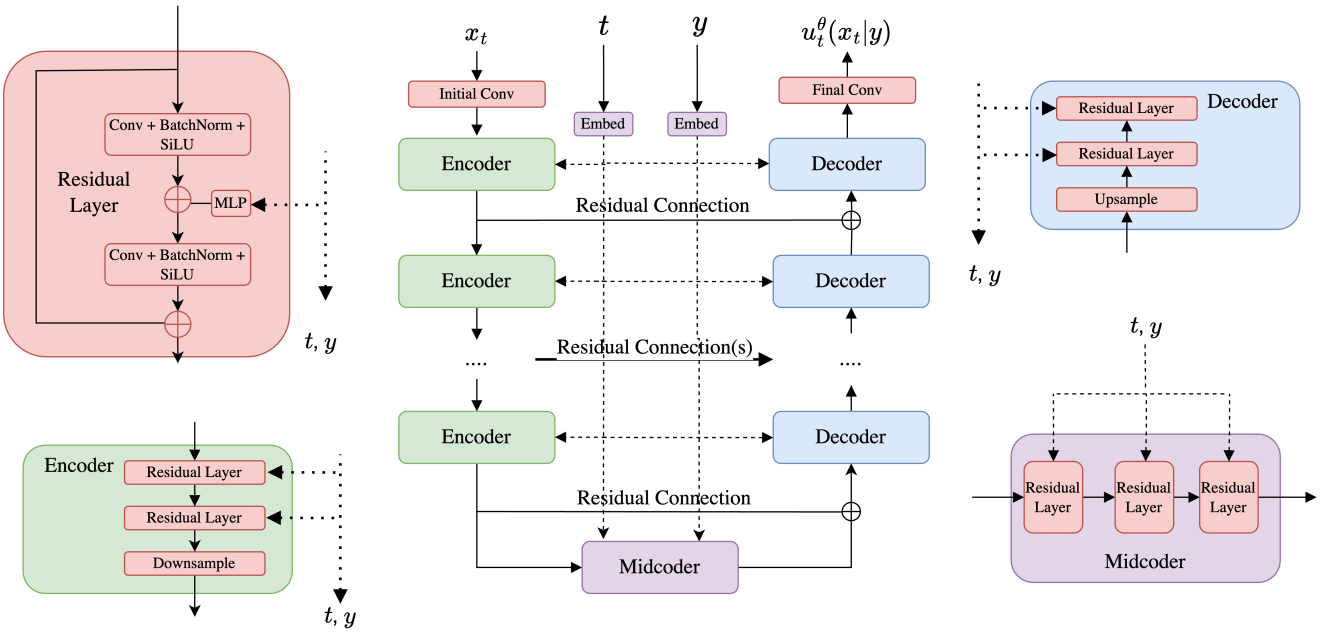

Figure 14: A simplified U-Net architecture used in last year's lab three.

described in the previous section and whose notation we borrow, we might consider an **encoder** $\mu_\phi : \mathbb{R}^d \to \mathbb{R}^k$, together with some **decoder** $\mu_\theta : \mathbb{R}^k \to \mathbb{R}^d$, 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, μ_ϕ and μ_θ are referred to as an **autoencoder**. Ideally, μ_ϕ and μ_θ 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**

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

Amenability to Generative Modeling. 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 μ_θ. 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). To allow for more explicit regularization of the latent distribution, we will now recast the concept of autoencoder in a more general probabilistic framework.

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 ϕ, and a decoder $p_\theta(x|z)$ with parameters θ. The most common choice is to take

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

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

$$z \sim q_\phi(\cdot|x) \qquad \text{(encode)}$$
$$x \sim p_\theta(\cdot|z) \qquad \text{(decode)}$$

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

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

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:

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

where we used the density of the normal distribution (see Equation (87)) 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 $z \sim q_\phi(\cdot|x)$. Further, we want the variance $\sigma_\theta^2(z)$ to be minimal. Many implementations fix $\sigma_\theta^2 = \sigma^2$ to a constant, which avoids 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:

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

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. For this purposes, let us now introduce a **prior distribution** $p_{\text{prior}}(z)$ over latents z. For our purposes, $p_{\text{prior}} = \mathcal{N}(0, I_k)$ almost always. The prior distribution p_{prior} effectively gives the "ideal" case of how the latent distribution should look like. A normal distribution would be very easy to learn and this would certainly satisfy our goals to build a "trainable" latent space. A natural way to enforce this is such that encoding distribution is close to the prior distribution

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

where D_{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 29 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 29 (Background on KL-divergence)
For two probability densities q, p, the **Kullback-Leibler divergence** (KL-divergence) is defined as

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

The KL divergence is a standard measure of dissimilarity between distributions, and will be our choice to replace $\mathcal{D}$ in **??**. In particular, the KL divergence satisfies the following useful properties:

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

$$D_{\mathrm{KL}}(q(x) \parallel p(x)) = 0 \quad \Leftrightarrow \quad q = p. \tag{77}$$

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

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

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

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 β controls the strength of each. To make this loss more specific, let us derive the KL divergence for the Gaussian case:

Example 30 (KL Divergence Between Isotropic Gaussians)
Let $q(x) = \mathcal{N}(x; \mu_q, \sigma_q^2 I_d)$ and $p(x) = \mathcal{N}(x; \mu_p, \sigma_p^2 I_d)$ be isotropic Gaussians, with $\sigma_q, \sigma_p \in \mathbb{R}^d_{\geq 0}$, and where $x \in \mathbb{R}^d$. Then

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

The expression above is intuitive: If the mean and variances coincide, that then $D_{\mathrm{KL}}(q \parallel p) = 0$. 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 $D_{\mathrm{KL}}(q \parallel p)$ 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 (87)):

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

Then

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

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

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

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

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

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

Let us now assume a Gaussian shape of the encoder

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

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

$$\mathcal{L}_{\text{VAE}}(\phi,\theta) \tag{83}$$

$$= \mathcal{L}_{\text{VAE-Recon}}(\phi,\theta) + \beta\mathcal{L}_{\text{VAE-Prior}}(\phi) \tag{84}$$

$$= \mathbb{E}_{x\sim p_{\text{data}}(x), z\sim q_\phi(z|x)}\Bigg[\underbrace{\frac{1}{2\sigma_\theta^2(x)}\|x-\mu_\theta(z)\|^2}_{\text{recon. error}} + \underbrace{\frac{1}{2}\log\sigma_\theta^2(z)}_{\text{decoder confidence}} + \underbrace{\frac{\beta}{2}\mathcal{K}\left(\sigma_\phi^2(x)\right)}_{\text{make latent variance=1}} + \underbrace{\frac{\beta}{2}\|\mu_\phi(x)\|^2}_{\text{make latent mean=0}}\Bigg] \tag{85}$$

The 4 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 ϕ. 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 ϵ whose distribution is independent of ϕ. Therefore, we can rewrite the loss as:

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

After reparameterization, the randomness comes only from $\epsilon\sim\mathcal{N}(0,I_d)$, whose distribution does not depend on ϕ. 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:

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

In Algorithm 6, we summarize the training procedure of the VAE.

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

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:

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

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

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

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**

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 β (and KL warm-up).** Large β 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 β value is very small, i.e. $\beta<<1$.

2. **Decoder variance.** Learning a Gaussian decoder variance σ_θ^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 σ^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 31 (Working in Latent Space)
To train a diffusion model, we simply work in **latent space** now. In this case, one first encodes the training dataset in the **latent space** via an VAE (usually we simply take the mean $\mu_\theta(x)$ of each data point in latent space), and then training the flow or diffusion model in the latent space. Sampling is performed by first sampling in the latent space using the trained flow or diffusion model, and then decoding of the output via the decoder. Intuitively, a well-trained autoencoder can be thought of as filtering out semantically meaningless details, allowing the generative model to "focus" on important, perceptually relevant features [26]. By now, nearly all state-of-the-art approaches to image and video generation involve training a flow or diffusion model in the latent space of an autoencoder - so called **latent diffusion models** [26, 34]. 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.

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 [7, 23]. 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).[4] 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

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

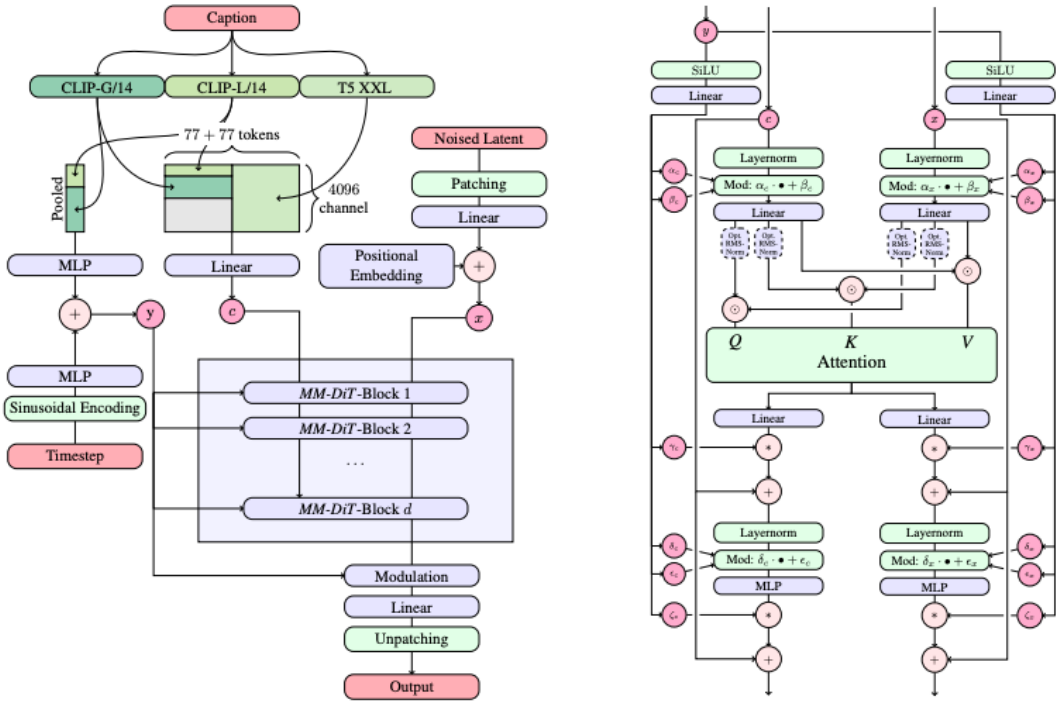

Figure 15: The architecture of the multi-modal diffusion transformer (MM-DiT) proposed in [7]. Figure also taken from [7].

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 [25], and similar to approaches taken in [2, 28]). 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 15. 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 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 $\frac{T'}{T} = \frac{H'}{H} = \frac{W'}{W} = 8$ [23]. To accomodate long videos, a temporal tiling procedure is proposed by which the video is chopped up into pieces, each piece is encoder separately, and the latents are stictched together [23]. 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 [33], ByT5 embeddings, for attending to character-level details (for e.g., prompts explicitly requesting specific text to be present) [36], and MetaCLIP embeddings, trained in a shared text-image embedding space [14, 23]. 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 [23].

7 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] 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.

[3] Earl A Coddington, Norman Levinson, and T Teichmann. *Theory of ordinary differential equations*. 1956.

[4] 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.

[5] Alexey Dosovitskiy. "An image is worth 16x16 words: Transformers for image recognition at scale". In: *arXiv preprint arXiv:2010.11929* (2020).

[6] 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.

[7] 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.

[8] Lawrence C Evans. *Partial differential equations*. Vol. 19. American Mathematical Society, 2022.

[9] Jonathan Ho, Ajay Jain, and Pieter Abbeel. "Denoising diffusion probabilistic models". In: *Advances in neural information processing systems* 33 (2020), pp. 6840–6851.

[10] Jonathan Ho and Tim Salimans. *Classifier-Free Diffusion Guidance*. 2022. arXiv: 2207.12598 [cs.LG]. URL: https://arxiv.org/abs/2207.12598.

[11] Arieh Iserles. *A first course in the numerical analysis of differential equations*. Cambridge university press, 2009.

[12] Alexia Jolicoeur-Martineau et al. "Adversarial score matching and improved sampling for image generation". In: *arXiv preprint arXiv:2009.05475* (2020).

[13] 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.

[14] 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.

[15] Yaron Lipman et al. "Flow matching for generative modeling". In: *arXiv preprint arXiv:2210.02747* (2022).

[16] Yaron Lipman et al. "Flow Matching Guide and Code". In: *arXiv preprint arXiv:2412.06264* (2024).

[17] 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).

[18] Nanye Ma et al. "Sit: Exploring flow and diffusion-based generative models with scalable interpolant transformers". In: *arXiv preprint arXiv:2401.08740* (2024).

[19] Xuerong Mao. *Stochastic differential equations and applications*. Elsevier, 2007.

[20] William Peebles and Saining Xie. *Scalable Diffusion Models with Transformers*. 2023. arXiv: 2212.09748 [cs.CV]. URL: https://arxiv.org/abs/2212.09748.

[21] 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.

[22] Lawrence Perko. *Differential equations and dynamical systems*. Vol. 7. Springer Science & Business Media, 2013.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] Yang Song et al. "Score-Based Generative Modeling through Stochastic Differential Equations". In: *International Conference on Learning Representations (ICLR)*. 2021.

[31] Yang Song et al. "Score-based generative modeling through stochastic differential equations". In: *arXiv preprint arXiv:2011.13456* (2020).

[32] 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.

[33] Yi Tay et al. *UL2: Unifying Language Learning Paradigms*. 2023. arXiv: 2205.05131 [cs.CL]. URL: https://arxiv.org/abs/2205.05131.

[34] 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.

[35] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[36] 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.

A A Reminder on Probability Theory

We present a brief overview of basic concepts from probability theory. This section was partially taken from [16].

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 $p_X : \mathbb{R}^d \to \mathbb{R}_{\geq 0}$ providing event A with probability

$$\mathbb{P}(X \in A) = \int_A p_X(x) \mathrm{d}x, \tag{86}$$

where $\int p_X(x)\mathrm{d}x = 1$. 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:

$$\mathcal{N}(x; \mu, \sigma^2 I) = (2\pi\sigma^2)^{-\frac{d}{2}} \exp\left(-\frac{\|x - \mu\|_2^2}{2\sigma^2}\right), \tag{87}$$

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:

$$\mathbb{E}[X] = \arg\min_{z \in \mathbb{R}^d} \int \|x - z\|^2 p_X(x)\mathrm{d}x = \int x p_X(x)\mathrm{d}x. \tag{88}$$

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

$$\mathbb{E}[f(X)] = \int f(x) p_X(x)\mathrm{d}x. \tag{89}$$

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

$$\int p_{X,Y}(x, y)\mathrm{d}y = p_X(x) \text{ and } \int p_{X,Y}(x, y)\mathrm{d}x = p_Y(y). \tag{90}$$

See Figure 16 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 $p_Y(y) > 0$:

$$p_{X|Y}(x|y) \coloneqq \frac{p_{X,Y}(x, y)}{p_Y(y)}, \tag{91}$$

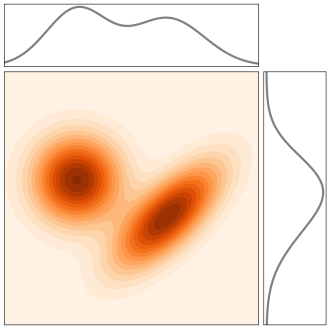

Figure 16: Joint PDF $p_{X,Y}$ (in shades) and its marginals p_X and p_Y (in black lines). Figure from [16]

.

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

$$p_{Y|X}(y|x) = \frac{p_{X|Y}(x|y)p_Y(y)}{p_X(x)}, \tag{92}$$

for $p_X(x) > 0$.

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

$$\begin{aligned} g_\star &\coloneqq \operatorname*{arg\,min}_{g:\mathbb{R}^d\to\mathbb{R}^d} \mathbb{E}\left[\|X - g(Y)\|^2\right] = \operatorname*{arg\,min}_{g:\mathbb{R}^d\to\mathbb{R}^d} \int \|x - g(y)\|^2 p_{X,Y}(x,y)\mathrm{d}x\mathrm{d}y \\ &= \operatorname*{arg\,min}_{g:\mathbb{R}^d\to\mathbb{R}^d} \int \left[\int \|x - g(y)\|^2 p_{X|Y}(x|y)\mathrm{d}x\right] p_Y(y)\mathrm{d}y. \end{aligned} \tag{93}$$

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

$$\mathbb{E}[X|Y = y] \coloneqq g_\star(y) = \int x p_{X|Y}(x|y)\mathrm{d}x, \tag{94}$$

where the second equality follows from taking the minimizer of the inner brackets in Equation (93) for $Y = y$, similarly to Equation (88). Composing $g_\star$ with the random variable Y, we get

$$\mathbb{E}[X|Y] \coloneqq g_\star(Y), \tag{95}$$

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 $\mathbb{R}^d \to \mathbb{R}^d$, 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:

$$\mathbb{E}[\mathbb{E}[X|Y]] = \mathbb{E}[X] \tag{96}$$

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:

$$\begin{aligned} \mathbb{E}[\mathbb{E}[X|Y]] &= \int \left(\int x p_{X|Y}(x|y)\mathrm{d}x\right) p_Y(y)\mathrm{d}y \\ &\overset{(91)}{=} \int\int x p_{X,Y}(x,y)\mathrm{d}x\mathrm{d}y \\ &\overset{(90)}{=} \int x p_X(x)\mathrm{d}x = \mathbb{E}[X]. \end{aligned}$$

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 (94), we obtain the identity

$$\mathbb{E}[f(X,Y)|Y = y] = \int f(x,y) p_{X|Y}(x|y)\mathrm{d}x. \tag{97}$$

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 32 (Fokker-Planck Equation)
Let p_t be a probability path with $p_0 = p_{\text{init}}$ and let us consider the SDE

$$X_0 \sim p_{\text{init}}, \quad \mathrm{d}X_t = u_t(X_t)\mathrm{d}t + \sigma_t \mathrm{d}W_t.$$

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

$$\partial_t p_t(x) = -\mathrm{div}(p_t u_t)(x) + \frac{\sigma_t^2}{2}\Delta p_t(x) \quad \text{for all } x \in \mathbb{R}^d, 0 \leq t \leq 1, \tag{98}$$

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 $f : \mathbb{R}^d \to \mathbb{R}$ 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 $g_1, g_2 : \mathbb{R}^d \to \mathbb{R}$ it holds that

$$g_1(x) = g_2(x) \text{ for all } x \in \mathbb{R}^d \quad \Leftrightarrow \quad \int f(x) g_1(x) \mathrm{d}x = \int f(x) g_2(x) \mathrm{d}x \text{ for all test functions } f \tag{99}$$

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:

$$\int f_1(x) \frac{\partial}{\partial x_i} f_2(x) \mathrm{d}x = -\int f_2(x) \frac{\partial}{\partial x_i} f_1(x) \mathrm{d}x \tag{100}$$

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:

$$\int \nabla f_1^T(x) f_2(x) \mathrm{d}x = -\int f_1(x) \mathrm{div}(f_2)(x) \mathrm{d}x \quad (f_1 : \mathbb{R}^d \to \mathbb{R}, f_2 : \mathbb{R}^d \to \mathbb{R}^d) \tag{101}$$

$$\int f_1(x) \Delta f_2(x) \mathrm{d}x = \int f_2(x) \Delta f_1(x) \mathrm{d}x \quad (f_1 : \mathbb{R}^d \to \mathbb{R}, f_2 : \mathbb{R}^d \to \mathbb{R}) \tag{102}$$

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

$$X_{t+h} = X_t + h u_t(X_t) + \sigma_t(W_{t+h} - W_t) + h R_t(h) \tag{103}$$

$$\approx X_t + h u_t(X_t) + \sigma_t(W_{t+h} - W_t) \tag{104}$$

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

make the following calculation:

$$\begin{aligned}
f(X_{t+h}) - f(X_t) &\overset{(104)}{=} f(X_t + hu_t(X_t) + \sigma_t(W_{t+h} - W_t)) - f(X_t) \\
&\overset{(i)}{=} \nabla f(X_t)^T \left(hu_t(X_t) + \sigma_t(W_{t+h} - W_t)\right) \\
&\quad + \frac{1}{2}\left(hu_t(X_t) + \sigma_t(W_{t+h} - W_t)\right)^T \nabla^2 f(X_t) \left(hu_t(X_t) + \sigma_t(W_{t+h} - W_t)\right) \\
&\overset{(ii)}{=} h\nabla f(X_t)^T u_t(X_t) + \sigma_t \nabla f(X_t)^T (W_{t+h} - W_t) \\
&\quad + \frac{1}{2} h^2 u_t(X_t)^T \nabla^2 f(X_t) u_t(X_t) + h\sigma_t u_t(X_t)^T \nabla^2 f(X_t)(W_{t+h} - W_t) + \\
&\quad + \frac{1}{2}\sigma_t^2 (W_{t+h} - W_t)^T \nabla^2 f(X_t)(W_{t+h} - W_t)
\end{aligned}$$

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

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

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

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

where in (i) we used the assumption that p_t as the distribution of X_t and in (ii) we used Equation (101) and Equation (102). Note that to use this, we require integrability of the product $p_t(x)u_t(x)$, i.e. such that

$$\int p_t(x) \|u_t(x)\| \mathrm{d}x < \infty$$

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

$$\partial_t \mathbb{E}[f(X_t)] = \int f(x) \left(-\operatorname{div}(p_t u_t)(x) + \frac{\sigma_t^2}{2} \Delta p_t(x) \right) \mathrm{d}x \quad \text{(for all } f \text{ and } 0 \leq t \leq 1) \tag{105}$$

$$\overset{(i)}{\Leftrightarrow} \quad \partial_t \int f(x) p_t(x) \mathrm{d}x = \int f(x) \left(-\operatorname{div}(p_t u_t)(x) + \frac{\sigma_t^2}{2} \Delta p_t(x) \right) \mathrm{d}x \quad \text{(for all } f \text{ and } 0 \leq t \leq 1) \tag{106}$$

$$\overset{(ii)}{\Leftrightarrow} \quad \int f(x) \partial_t p_t(x) \mathrm{d}x = \int f(x) \left(-\operatorname{div}(p_t u_t)(x) + \frac{\sigma_t^2}{2} \Delta p_t(x) \right) \mathrm{d}x \quad \text{(for all } f \text{ and } 0 \leq t \leq 1) \tag{107}$$

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

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 (99) . 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. [8, Chapter 7]). Now, if Equation (98) 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.