Ordinary Differential Equations in Vision and Language

Tong Xiao XIAOTONG@MAIL.NEU.EDU.CN
Northeastern University, China

Junhao Ruan RANGEHOW@OUTLOOK.COM
Northeastern University, China

Bei Li LIBEI_NEU@OUTLOOK.COM
Northeastern University, China

Zhengtao Yu ZTYU@HOTMAIL.COM
Kunming University of Science and Technology, China

Min Zhang ZHANGMIN2021@HIT.EDU.CN
Harbin Institute of Technology (Shenzhen), China

Jingbo Zhu ZHUJINGBO@MAIL.NEU.EDU.CN
Northeastern University, China

Abstract

Ordinary differential equations (ODEs) provide a mathematical foundation for understanding the continuous evolution of complex systems. While deep learning has traditionally relied on discrete layer-wise computations, many state-of-the-art models in vision and language can be fundamentally interpreted as discretized approximations of continuous dynamical processes. In this paper, we present a unified theoretical framework that aligns diverse neural architectures and generative processes through the formalism of ODEs. We demonstrate that this continuous-time perspective offers a principled approach to model design across three progressive dimensions. First, we show how classic residual networks and Transformers can be formulated as specific discretization schemes of underlying ODEs, motivating the design of advanced architectures. Second, we analyze neural ODEs and flow-based models, and discuss how fully continuous-time formulations enable more flexible density estimation and parameter efficiency. Third, we treat diffusion models as a connection between discrete and continuous frameworks. We show that modern generative models in vision and language can be interpreted from this unified perspective of differential equations. More broadly, ODEs can be used as a general paradigm for posing and analyzing deep learning problems in vision and language. This perspective can, in turn, drive new architecture and algorithm designs by leveraging the powerful mathematical toolkit of continuous dynamical systems.

Contents

1 Mathematical Preliminaries **4**
1.1 Notation of ODEs . . . 4
1.2 Solving ODEs . . . 8
1.3 Two Examples . . . 12
1.3.1 Training with Gradient Descent . . . 12
1.3.2 Continuous-time Markov Chains . . . 13

2 The ODE Perspective on Transformers **15**
2.1 The Connection Between Transformers and ODEs . . . 15
2.2 ODE-inspired Architecture Desgin . . . 17
2.2.1 Architectures Interpreted from Physical Systems . . . 17
2.2.2 Architectures based on Higher-order Methods . . . 19
2.2.3 Architectures based on Implicit Methods . . . 20
2.3 Remarks on Stability and Stiffness . . . 22

3 Neural ODEs and Flows **24**
3.1 Neural ODEs . . . 25
3.2 The Adjoint Sensitivity Method . . . 27
3.2.1 Adjoint Dynamics . . . 28
3.2.2 The Augmented ODE and Memory Efficiency . . . 30
3.3 Neural Dynamics and Flows . . . 31
3.3.1 Vector Fields . . . 31
3.3.2 Flows and Diffeomorphisms . . . 32
3.3.3 Continuous Normalizing Flows . . . 34
3.3.4 Variants of Flows . . . 37

4 Diffusion Models in Vision **38**
4.1 Problem Statement . . . 39
4.1.1 Generative Modeling as Probability Transport . . . 39
4.1.2 The Duality of Forward and Backward Processes . . . 41
4.1.3 Training and Inference . . . 43
4.2 The Stochastic Perspective: SDEs . . . 44
4.2.1 Forward and Reverse SDEs . . . 44
4.2.2 Learning the Score Function . . . 46
4.2.3 The VE and VP SDEs . . . 48
4.2.4 Numerical Solvers . . . 50
4.3 The Deterministic Perspective: Probability Flow ODEs . . . 51
4.3.1 From SDEs to ODEs . . . 51
4.3.2 Flow Matching . . . 52
4.4 Towards More Efficient Generation . . . 55
4.4.1 High-Order and Specialized Solvers . . . 56

4.4.2 Trajectory Rectification . . . 57
4.4.3 Consistency Models . . . 58

5 Diffusion Models in Language **59**
5.1 Token Sequence Generation . . . 60
5.1.1 Autoregressive Generation vs. Non-autoregressive Generation . . . 60
5.1.2 Diffusion Modeling as Non-autoregressive Generation . . . 61
5.2 Diffusion in Continuous Space: Embedding-Space Diffusion . . . 63
5.2.1 General Framework . . . 63
5.2.2 Length Prediction . . . 65
5.2.3 Rounding . . . 67
5.3 Diffusion in Token Space: Discrete Diffusion . . . 68
5.3.1 The CTMC Perspective . . . 68
5.3.2 Masked Diffusion Models . . . 73
5.4 Remarks on Discrete Diffusion . . . 77
5.4.1 Rethinking Non-absorbing Replacement Diffusion . . . 77
5.4.2 Masked Diffusion Modeling vs Masked Language Modeling . . . 78
5.4.3 Simplex and Logit Dynamics as Continuous Relaxations . . . 80
5.4.4 Relation to Iterative Refinement . . . 81
5.4.5 Diffusion Transformers . . . 82
5.4.6 Flow-based Views . . . 83

6 Conclusions and Future Directions **85**

A Deriving the Heavy Ball ODE **87**

B Deriving CNFs from the Continuity Equation **87**

C Deriving Probability Flow ODEs from SDEs **89**

1. Mathematical Preliminaries

The concept of **ordinary differential equations** (ODEs) comes from a more general concept — **differential equations**. This field studies the problem of how variables change and has a long and successful history [Edwards, 1994; Dunham, 2005]. Today, differential equations have become the essential language of modern science and engineering, and their applications are wide-ranging. In this section, we introduce the basic concepts of differential equations. In particular, we focus on ODEs and demonstrate a few applications of these mathematical tools in deep learning.

1.1 Notation of ODEs

We begin by introducing a simple problem of a moving car, which we will use as an example throughout this section to motivate several key concepts. Suppose we observe a car moving along a straight path. Let $t \in \mathbb{R}$ represent time, which serves as our independent variable. We denote the position of the car at any given time t as $x(t)$. Here $x(t)$ is the dependent variable, whose value depends on the independent variable t. In some cases, we may omit the argument to simplify the notation (i.e., use x instead of $x(t)$).

We know that the velocity of the car is the rate of change of its position with respect to time. In calculus, this quantity is denoted as the derivative $\frac{dx(t)}{dt}$. Here $dx(t)$ and dt are called **differentials**, which represent infinitesimal changes in the dependent and independent variables, respectively. The derivative $\frac{dx(t)}{dt}$ describes the instantaneous rate of change of the position $x(t)$ with respect to time t. So this concept is defined at a point in time, not over a finite interval. Mathematically, $\frac{dx(t)}{dt}$ can be written as the limit of the difference quotient

$$\frac{dx(t)}{dt} = \lim_{\Delta t \to 0} \frac{x(t + \Delta t) - x(t)}{\Delta t}. \tag{1}$$

One can find more details on the formal definition of derivatives in calculus textbooks [Apostol, 1991; Spivak, 2006].

If the velocity of the car is governed by a rule depending on its current position and the time (for example, due to varying road friction or wind resistance), we can express this relationship using a function $f(\cdot)$

$$\frac{dx(t)}{dt} = f(x(t), t). \tag{2}$$

This equation is formally known as an ODE. The term *ordinary* implies that the dependent variable $x(t)$ is a function of a single independent variable t. This stands in contrast to **partial differential equations** (PDEs), where the unknown function depends on multiple independent variables[1].

1. An example of partial differential equations is the **heat equation**, which describes how the temperature u changes over time t and spatial location s. It is typically written as

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial s^2}, \tag{3}$$

where α is a positive coefficient. In such cases, the equation involves partial derivatives, denoted by the symbol ∂. $\frac{\partial^2 u}{\partial s^2}$ is the second spatial derivative of u in the s direction.

Alternatively, Eq. (2) can also be written in differential form

$$dx(t) = f(x(t), t)dt. \tag{4}$$

This notation suggests that the infinitesimal change in position is the product of the velocity and the time increment. We will show in subsequent sections that such a form is particularly useful for developing numerical integration methods and modeling stochastic processes.

Clearly, the complexity of the ODE given by Eqs. (2) and (4) is determined by that of the function $f(\cdot)$. The simplest case is that $f(\cdot)$ is a constant. For example, if the car travels at a constant velocity v, then the function takes the form $f(x(t), t) = v$. In this scenario, Eq. (2) reduces to

$$\frac{dx(t)}{dt} = v. \tag{5}$$

It is easy to figure out that this ODE corresponds to the familiar linear equation of motion

$$x(t) \quad = \quad vt + x(0), \tag{6}$$

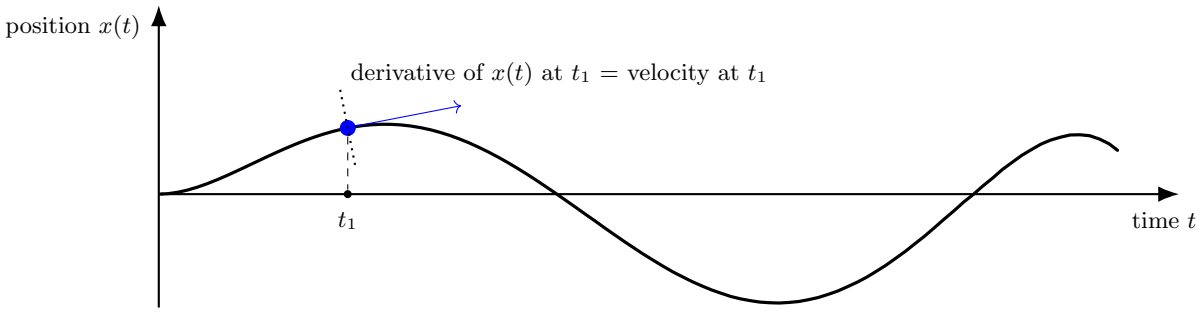
where $x(0)$ represents the position of the car at time $t = 0$. This simple example demonstrates a fundamental property of differential equations: the general solution typically involves an arbitrary constant (here, $x(0)$). Geometrically, this means the ODE describes a family of parallel curves in the (x, t) plane, each corresponding to a different starting point.

The situation becomes more interesting when $f(\cdot)$ depends explicitly on the position $x(t)$. For instance, suppose the velocity decreases as the car moves away from the origin due to some control rule. If the function $f(\cdot)$ depends linearly on $x(t)$ (e.g., $f(x(t), t) = -kx(t)$ for some constant k), the equation is classified as a **linear ODE**. Such equations are of particular interest because they often admit closed-form analytical solutions [Hartman, 2002].

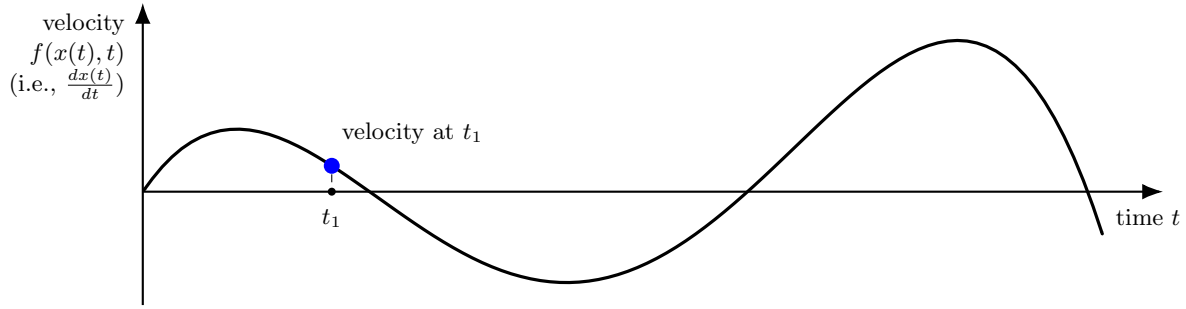
However, in many real-world applications, $f(\cdot)$ is nonlinear. If our car were moving in an environment where the governing relationship involved terms like $x(t)^2$ or $\sin(x(t))$, we would be dealing with a **nonlinear ODE**. Unlike their linear counterparts, nonlinear equations rarely possess simple analytical solutions. Consequently, exact analytical derivation is often impossible, and we must resort to numerical approximation techniques. As most problems discussed in the subsequent chapters involve nonlinear ODEs, exploring approximate solutions will be the focus of the following discussion.

Note that, although $f(\cdot)$ takes two arguments, $x(t)$ and t, the term $x(t)$ itself is a function of t. Thus, along any specific trajectory, the velocity can be seen as a composite function of t. In this case, we can plot $f(\cdot)$ as a function of t, where at any point the value corresponds to the derivative $\frac{dx(t)}{dt}$. See Figure 1 for a simple example.

In practice, however, the specific dependence of $f(\cdot)$ on t is often unknown because the trajectory $x(t)$ itself is unknown. Furthermore, additional complexity arises from the explicit time-dependence of $f(\cdot)$. Unlike autonomous functions that possess a static structure, here the functional form itself evolves over time (to emphasize this, we can adopt the notation $f_t(\cdot)$ to represent $f(\cdot, t)$). This means that even if the car returns to the exact same position at a later instant, it



(a) Position $x(t)$ against time t



(b) Velocity $f(x(t), t)$ (i.e., $\frac{dx(t)}{dt}$) against time t

Figure 1: Plots of the dependent variable ($x(t)$) and its derivative ($\frac{dx(t)}{dt}$) of an ODE. While the ODE requires $x(t)$ as an input to $f(\cdot)$, $x(t)$ itself is driven by time. Therefore, if the trajectory is fixed, the velocity $f(\cdot)$ implicitly becomes a function of t alone. So we can plot the velocity $f(\cdot)$ directly against t.

may have a different velocity. Since the "rules of the road" are not constant, the behavior of the car cannot be predicted solely by its current state $x(t)$, and one must also account for the specific moment in time.

More fundamentally, such an ODE defines a **dynamical system** [Hirsch et al., 2013]. In this framework, the variable $x(t)$ represents the **state** of the system, which describes its configuration at any specific moment. The function $f(\cdot)$ characterizes the **dynamics** of the system (often referred to as the **dynamic rule**), which indicates how the state changes at any given instant. From this perspective, the distinction regarding time-dependence becomes a classification of the rules themselves: if the dynamic rule is static and depends solely on the state (i.e., $\frac{dx(t)}{dt} = f(x(t))$), the system is **autonomous**. If the rule itself changes over time (i.e., $\frac{dx(t)}{dt} = f(x(t), t)$), the system is **non-autonomous**. Non-autonomous systems are powerful tools for describing complex processes, though they are inherently more complicated than autonomous systems. For example, in Section 2, we will show that a stack of Transformer layers can be modeled as a non-autonomous system.

Notation	Description
t	The independent variable, typically representing time.
$x(t)$	The dependent variable, representing the state of the system at time t.
$dx(t)/dt$	The derivative of the state, representing the instantaneous rate of change (e.g., velocity).
$f(x(t), t)$	The function (or dynamic rule) governing the system's evolution. It describes the dynamics of the system.
$\frac{dx(t)}{dt} = f(x(t), t)$	A non-autonomous ODE, where the dynamics explicitly depend on time.
$\frac{dx(t)}{dt} = f(x(t))$	An autonomous ODE, where the dynamics depend solely on the state (time-invariant rules).
$dx(t) = f(x(t), t)dt$	The differential form of the ODE, often used in numerical integration and stochastic calculus.
$\frac{d\mathbf{x}(t)}{dt} = f(\mathbf{x}(t), t)$	A vector-valued ODE, describing a system with a multi-dimensional state vector $\mathbf{x}(t) \in \mathbb{R}^d$ (where $f(\cdot)$ defines a vector field).

Table 1: Basic concepts in ODEs.

So far, our discussion has focused on scalar variables and functions, that is, the state $x(t)$ is a scalar variable and the function $f(\cdot)$ is a scalar function. One natural extension is to define ODEs on vectors. Let $\mathbf{x}(t) \in \mathbb{R}^d$ denote a vector state containing d distinct components (e.g., the neuron activations in a neural network layer). Correspondingly, the function $f(\cdot)$ is defined as a vector-valued mapping: $\mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$. Then, the ODE becomes

$$\frac{d\mathbf{x}(t)}{dt} = f(\mathbf{x}(t), t). \tag{7}$$

Here, the derivative $\frac{d\mathbf{x}(t)}{dt}$ is applied component-wise to the vector $\mathbf{x}(t)$. This shift to higher dimensions introduces a richer geometric interpretation. While the scalar function in our previous examples simply represents the slope of a curve, the vector function $f(\cdot)$ defines a **vector field** over the state space. At every point $\mathbf{x}(t)$, $f(\mathbf{x}(t), t)$ assigns a vector that specifies both the magnitude and direction of the system's instantaneous velocity.

Although we extend the state to a d-dimensional space, the fundamental concepts and properties of ODEs remain valid. In this section, we will continue to use scalar notation for simplicity. In subsequent sections, however, we will primarily employ vector-valued ODEs to model more complex problems. We summarize the key concepts introduced thus far in Table 1.

1.2 Solving ODEs

We have shown that ODEs describe the derivative of the state $x(t)$. One might ask: why do we define the system using derivatives instead of describing the state $x(t)$ directly? Indeed, if the full trajectory of $x(t)$ were known, the ODE would be redundant. However, in many applications, we do not have analytical expressions for $x(t)$. Instead, we are only provided with the dynamics $f(\cdot)$ and the initial condition $x(0)$. This is analogous to driving a car: while it is easy to know our current speed at any moment, it is much harder to determine the current position relative to a starting point. In the language of ODEs, we are given the "speedometer" $f(\cdot)$ and the starting point $x(0)$, and our goal is to reconstruct the "map" of the journey $x(t)$.

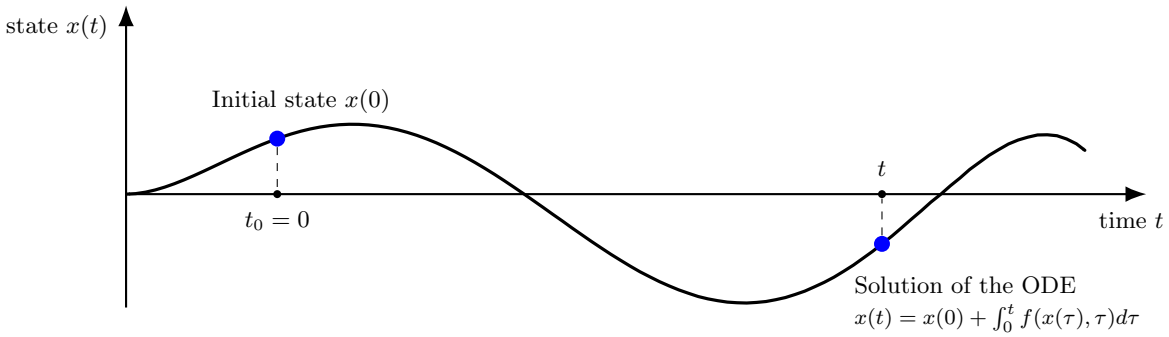
In this case, we need to solve ODEs. This is often referred to as the **initial value problem** (IVP) where an ODE and a specific constraint (e.g., the initial state $x(0)$) are given, and we recover the state $x(t)$ at any time t from its derivative. By the fundamental theorem of calculus, the state at time t is the sum of the initial state and the accumulation of all instantaneous changes over the interval $[0, t]$. Thus we can write the solution to the ODE by integration

$$\begin{aligned} x(t) &= x(0) + \int_0^t \frac{dx(\tau)}{d\tau} d\tau \\ &= x(0) + \int_0^t f(x(\tau), \tau) d\tau. \end{aligned} \tag{8}$$

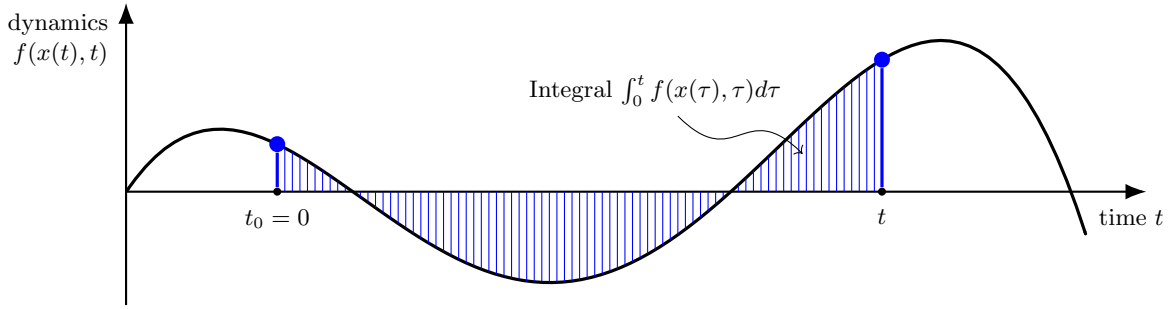
As shown in Figure 2, this formula has a very intuitive geometric interpretation. The integral term $\int_0^t f(x(\tau), \tau) d\tau$ represents the signed area under the curve of the function $f(x(\tau), \tau)$, from $\tau = 0$ to $\tau = t$. This area quantifies the total accumulated change in the state $x(t)$ over the interval $[0, t]$. Therefore, the equation simply states that the state at time t is found by adding this total change (the area) to the initial state $x(0)$.

However, this integral formulation presents a computational challenge of ODE solving: while the equation appears simple, it cannot be directly computed because the integrand $f(x(\tau), \tau)$ depends on the unknown state trajectory $x(\tau)$ itself. Moreover, even if the trajectory were implicitly defined, evaluating the integral analytically is often difficult due to the complexity and nonlinearity of the dynamics $f(\cdot)$ (e.g., when modeling with deep neural networks). Instead, we must resort to numerical methods to approximate the integration. The core idea behind these methods is **discretization**: instead of tracking the continuous evolution of $x(t)$ at every infinitesimal moment, we approximate the trajectory at a sequence of discrete time steps $\{t_0, t_1, \ldots, t_N\}$. Since evaluating the function $f(\cdot)$ at a finite number of points is computationally inexpensive, discretization provides a very efficient way to obtain approximate solutions.

Discretization can be interpreted from the perspective of Riemann integration. We can approximate the continuous integral in Eq. (8) by partitioning the integration interval into small sub-intervals and summing the areas of rectangles defined by the derivative function. Figure 3 shows an illustration. To formalize this, let us define a temporal grid on the interval $[0, t]$ consisting of $N + 1$ points $t_0 < t_1 < \cdots < t_N$, where $t_0 = 0$ is the start time and $t_N = t$ is the end time.



(a) State $x(t)$ against time t



(b) Dynamics $f(x(t), t)$ against time t

Figure 2: Illustration of solving the ODE $\frac{dx(t)}{dt} = f(x(t), t)$ by integration. The blue shaded area represents the integral $\int_0^t f(x(\tau), \tau) d\tau$. The solution to the ODE at time t is given by the sum of the initial state $x(0)$ and the integral: $x(t) = x(0) + \int_0^t f(x(\tau), \tau) d\tau$.

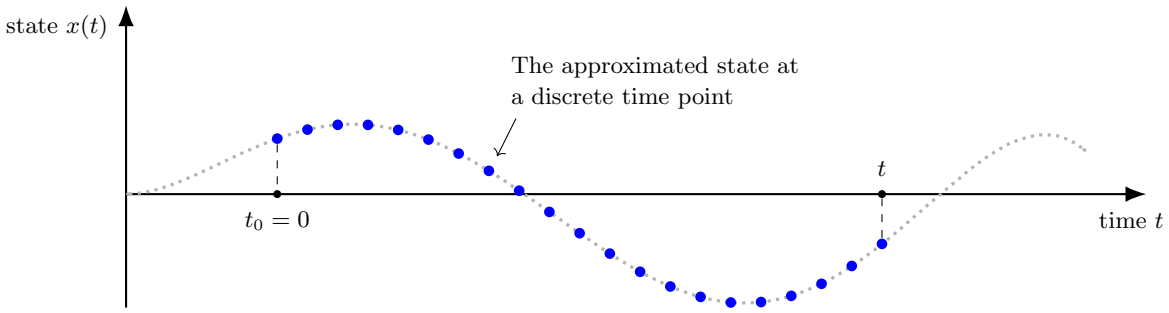
For simplicity, we often assume the time intervals are constant, i.e.,

$$\Delta t \quad = \quad t_{k+1} - t_k, \tag{9}$$

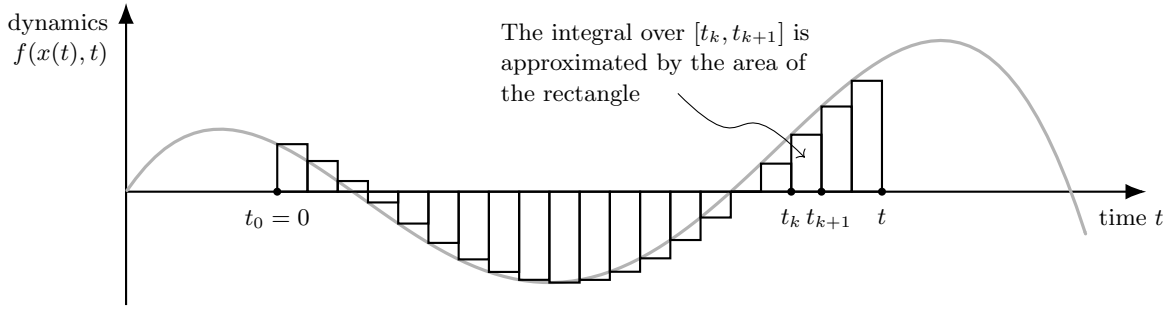
where Δt is referred to as the **step size**.

Our objective is to compute a sequence of values $\{\bar{x}_0, \bar{x}_1, \ldots, \bar{x}_N\}$, where each $\bar{x}_k$ serves as an approximation to the true state $x(t_k)$. The transition from the current state $\bar{x}_k$ to the next state $\bar{x}_{k+1}$ requires approximating the integral of the dynamics over the small interval $[t_k, t_{k+1}]$. Mathematically, we can express the discretized model using the following recurrent form

$$\bar{x}_{k+1} = \bar{x}_k + \text{Approx}\left(\int_{t_k}^{t_{k+1}} f(x(\tau), \tau) d\tau \right), \tag{10}$$



(a) State $x(t)$ against time t



(b) Dynamics $f(x(t), t)$ against time t

Figure 3: Illustration of discretizing an ODE. The function $f(\cdot)$ is evaluated at a number of discrete time points. At each time point, an approximation of the state is obtained, denoted by blue circles. The integration of $f(\cdot)$, which determines the state evolution, is approximated over each sub-interval by the area of a rectangle, given by $\Delta t \cdot \Psi(\bar{x}_k, t_k, \Delta t, f)$, where Δt is the width and $\Psi(\bar{x}_k, t_k, \Delta t, f)$ is the height.

where the function Approx$(\cdot)$ approximates the local integral. In discretized ODE models, this function is generally defined using the local geometry of $f(\cdot)$, like this

$$\text{Approx}\left(\int_{t_k}^{t_{k+1}} f(x(\tau), \tau) d\tau\right) \quad = \quad \Delta t \cdot \Psi(\bar{x}_k, t_k, \Delta t, f). \tag{11}$$

Then, Eq. (10) can be rewritten as

$$\bar{x}_{k+1} = \bar{x}_k + \Delta t \cdot \Psi(\bar{x}_k, t_k, \Delta t, f). \tag{12}$$

Here, $\Psi(\cdot)$ is called the **increment function** or **step function**. It estimates the integral over the interval using the values of $f(\cdot)$ computed at the available discrete points. Different numerical methods correspond to different choices of $\Psi(\cdot)$. For example, choosing $\Psi(\cdot)$ based on the slope at the left endpoint (t_k) corresponds to the classic **Euler method** (analogous to a left Riemann

sum). In this case, the increment function is simply the dynamics function itself: $\Psi(\bar{x}_k, t_k, \Delta t, f) = f(\bar{x}_k, t_k)$. Substituting this back into Eq. (12) yields the update rule of the Euler method

$$\bar{x}_{k+1} = \bar{x}_k + \Delta t \cdot f(\bar{x}_k, t_k). \tag{13}$$

As only one function evaluation is required per step, the Euler method is computationally efficient. However, it often suffers from poor accuracy, particularly when the step size Δt is large or when the underlying dynamics $f(\cdot)$ vary rapidly. To address these limitations, numerical solvers generally follow two main strategies: higher-order single-step methods and multi-step methods.

- **Higher-order Single-step Methods**. In numerical analysis, the order of a method refers to the rate at which the approximation error decreases as the step size Δt approaches zero. A method is of order p if the local error scales with $O((\Delta t)^{p+1})$. While the classic Euler method is first-order, higher-order methods improve accuracy by evaluating the dynamics $f(\cdot)$ at multiple intermediate points within the interval $[t_k, t_{k+1}]$ to better estimate the average slope. For instance, the fourth-order **Runge-Kutta method** (RK4) computes a weighted average of four different slopes (one at the start, two at the midpoint, and one at the end). It can achieve high precision without requiring an extremely small step size.

- **Multi-step Methods**. Unlike single-step methods (like Euler and RK4) which discard previous information and compute the next state $\bar{x}_{k+1}$ relying solely on the current state $\bar{x}_k$, multi-step methods exploit the history of the trajectory. By fitting a polynomial to the past derivative values, methods such as the Adams-Bashforth family can predict the integral over the next interval. The primary advantage of multi-step methods is computational efficiency: they can achieve high-order accuracy by reusing previously computed evaluations of $f(\cdot)$, rather than performing multiple new expensive function evaluations per step. However, they are not self-starting and require a single-step method to initialize the first few steps of the trajectory.

There are also other ways to improve ODE solvers. For instance, in **adaptive step-size methods**, the assumption of a constant step size Δt is relaxed. Instead, the solver dynamically adjusts Δt at every step. Here we do not go into details of these ODE solvers, and will use them as standard tools in the following discussion. Interested readers can refer to books on numerical analysis for further details [Stoer et al., 1980; Burden et al., 2015].

Before proceeding, it is necessary to clarify the notation used in this work, as conventions vary across the literature. Typically, the function notation $x(t)$ denotes the continuous-time state variable. In contrast, the subscript notation x_t is frequently used in machine learning and stochastic calculus to represent the state at a specific time snapshot t. In the context of numerical solvers, x_k (or x_{t_k}) often refers to the computed value at the k-th integration step corresponding to time t_k.

In our previous discussion, we distinguished the numerical approximation $\bar{x}_k$ from the true analytical solution $x(t_k)$. However, to avoid cluttered notation, and to make our mathematical representation more consistent with existing literature, we will use x_k (without a bar) to represent

the estimated value at step k in numerical methods. More generally, we can use x_k to denote the state at some discrete time step. Although this introduces a slight abuse of notation, the meaning will be clear from the context. For instance, the Euler method can be written as

$$x_{k+1} = x_k + \Delta t \cdot f(x_k, t_k) \tag{14}$$

Likewise, occasionally we may use $x(t)$ instead of $\bar{x}(t)$ to denote the estimated value of the state at time t, where the distinction from the true solution is not critical.

To summarize, given an ODE governed by the dynamics f and an initial state $x(0)$, we can view the ODE solver as a function $\phi(\cdot)$. For any $t \geq 0$, this function returns an approximation of the state, denoted as

$$x(t) \quad = \quad \phi(f, x(0), t), \tag{15}$$

while satisfying the initial condition $x(0) = \phi(f, x(0), 0)$.

1.3 Two Examples

We now illustrate the application of ODEs through two examples. The first example concerns modeling parameter updates in training, and the second example focuses on **continuous-time Markov chains**.

1.3.1 Training with Gradient Descent

The most popular method for training neural networks is gradient descent. The idea is that we first define a loss function $L(\theta)$ that quantifies the discrepancy between the model's predictions and the ground truth, where $\theta \in \mathbb{R}^d$ represents the vector of model parameters. To minimize this loss, we iteratively adjust the parameters in the direction of the steepest descent, which is given by the negative gradient $-\frac{dL(\theta)}{d\theta}$. At each iteration k, the update rule is given by

$$\theta_{k+1} = \theta_k - \eta \cdot \frac{dL(\theta_k)}{d\theta_k}, \tag{16}$$

where $\eta > 0$ is the learning rate.

Recall from Eq. (14) that the Euler method for solving ODEs is given by $x_{k+1} = x_k + \Delta t \cdot f(x_k, t_k)$. By comparing the two equations, we observe a direct correspondence between them:

$$\begin{aligned} \text{State } x_k &\quad \Leftrightarrow \quad \text{Parameters } \theta_k \\ \text{Step Size } \Delta t &\quad \Leftrightarrow \quad \text{Learning Rate } \eta \\ \text{Dynamics } f(x_k, t_k) &\quad \Leftrightarrow \quad \text{Negative Gradient } -dL(\theta_k)/d\theta_k \end{aligned}$$

Considering the similarity between these equations, we can view gradient descent as the explicit Euler discretization of a continuous-time ODE. If we take the limit as the learning rate $\eta \to 0$

and correspondingly the number of steps $k \to \infty$, the sequence of discrete parameter updates converges to the solution of the following ODE

$$\frac{d\theta_t}{dt} = -\frac{dL(\theta_t)}{d\theta_t}. \tag{17}$$

This ODE is formally known as **gradient flow**. Imagine a massless particle moving on a complex surface (e.g., the loss landscape). Since the particle has no mass, its instantaneous velocity is determined entirely by the slope of the terrain. The steeper the slope, the faster it moves, guided strictly by the direction of the negative gradient. This continuous descent process is exactly what Eq. (17) describes.

By connecting gradient descent with ODEs, we can use the well-established theory of ODEs to better understand the behavior of neural network optimization. For instance, the stability of the ODE solution provides insights into why training might diverge if the learning rate is too large (a phenomenon known as stiffness in numerical analysis). Furthermore, this perspective can extend naturally to more advanced optimizers. Consider stochastic gradient descent with momentum [Qian, 1999], which is designed to accelerate training by accumulating past gradients. The update rule is typically formulated as a system of two coupled equations:

$$v_{k+1} = \mu v_k - \eta \cdot \frac{dL(\theta_k)}{d\theta_k} \tag{18}$$

$$\theta_{k+1} = \theta_k + v_{k+1}, \tag{19}$$

where v_k represents the momentum variable at step k, and $\mu \in [0, 1)$ is the momentum coefficient that controls how much of the past accumulation is retained.

If we analyze the continuous-time limit of these discrete updates (by letting the step size $\eta \to 0$), we no longer obtain a first-order ODE. Instead, the system converges to a second-order ODE [Su et al., 2016]

$$\frac{d^2\theta_t}{dt^2} + \lambda \frac{d\theta_t}{dt} = -\frac{dL(\theta_t)}{d\theta_t}, \tag{20}$$

where λ is a coefficient related to the damping friction. This equation is physically analogous to Newton's second law of motion and is formally known as the **heavy ball ODE**. The interested reader can refer to Appendix A for a more detailed derivation of the heavy ball ODE.

1.3.2 Continuous-time Markov Chains

A continuous-time Markov chain (CTMC) describes a stochastic process that moves among a finite number of discrete states, where the transitions occur continuously in time [Norris, 1998]. While CTMCs are often introduced via jump processes, they can be modeled as a deterministic linear dynamical system governing the evolution of probability distributions. This perspective connects probability theory to ODEs.

Consider a system with N discrete states, denoted by the set $\mathcal{S} = \{1, 2, \ldots, N\}$. Instead of tracking a single random trajectory $x(t)$, we focus on the evolution of the probability distribution

over all states. Let $\mathbf{p}(t) = \begin{bmatrix} p_1(t) & p_2(t) & \cdots & p_N(t) \end{bmatrix}^\top \in \mathbb{R}^N$ be a column vector where the i-th component represents the probability of being in state i at time t:

$$p_i(t) \quad = \quad \Pr(x(t) = i), \tag{21}$$

subject to

$$\sum_{i=1}^{N} p_i(t) \quad = \quad 1. \tag{22}$$

To describe the evolution of the system, we need to specify how this probability distribution changes with respect to time. Intuitively, we can view this as a flow of probability mass between states. The dynamics of this flow are defined by a transition rate matrix $\mathbf{Q} \in \mathbb{R}^{N \times N}$, often called the infinitesimal generator. The elements of $\mathbf{Q}$ quantify the rate of probability mass moving between states:

- Off-diagonal elements (q_{ij} for $i \neq j$): represent the instantaneous rate of transition from state i to state j. These must be non-negative ($q_{ij} \geq 0$).
- Diagonal elements (q_{ii}): defined as $q_{ii} = -\sum_{j \neq i} q_{ij}$. This ensures that each row sums to zero, reflecting the conservation of total probability (mass flowing out must equal the total rate of departure).

Given the infinitesimal generator, the evolution of the probability vector $\mathbf{p}(t)$ can be defined by a system of linear ODEs known as the **Kolmogorov forward equation** (or the **master equation** in physics):

$$\frac{d\mathbf{p}(t)}{dt} \quad = \quad \mathbf{Q}^\top \mathbf{p}(t). \tag{23}$$

This ODE represents a global balance of probability currents. The matrix product $\mathbf{Q}^\top \mathbf{p}(t)$ aggregates the probability flow for each state. Specifically, for a state j, the rate of change $\frac{dp_j}{dt}$ is determined by the difference between the total probability mass entering j from all other states (inflow) and the mass departing from j (outflow). Thus, Eq. (23) simply states that the rate of change of the probability distribution depends linearly on the current distribution and the transition rates.

If $\mathbf{Q}$ is constant, the CTMC is time-homogeneous. In this case, Eq. (23) is a standard linear ODE system, and its analytical solution is given by the matrix exponential

$$\mathbf{p}(t) = e^{\mathbf{Q}^\top t} \mathbf{p}(0), \tag{24}$$

where $\mathbf{p}(0)$ is the initial distribution, and $e^{\mathbf{Q}^\top t}$ is the propagator that maps the initial distribution to the distribution at time t. In Section 5, we show that CTMCs can serve as the foundation for discrete diffusion models.

2. The ODE Perspective on Transformers

In the previous section, we saw that while ODEs were originally designed to model continuous-time processes, they are capable of dealing with discrete-time problems via discretization. In fact, depending on the discretization scheme, we can apply ODEs to different problems. In this section, we discuss one important case where ODEs are fully discretized to inspire neural network design. Specifically, we present interpretations of and improvements to residual network-based models, such as Transformers, from the discretized ODE perspective. We show that ODEs can provide insights into designing such discrete structures.

2.1 The Connection Between Transformers and ODEs

Here we consider Transformers as sequence models that take a sequence of tokens as input and produce contextualized representations as output [Vaswani et al., 2017]. For example, most recent **large language models** (LLMs) are based on the Transformer decoder architecture [Brown et al., 2020; Bai et al., 2023; Liu et al., 2024]. This architecture consists of a stack of identical layers, called Transformer layers. Within each Transformer layer, there are two sub-layers: a **multi-head self-attention** (MHSA) sub-layer and a position-wise **feed-forward network** (FFN) sub-layer. A crucial design choice in Transformers is the use of residual connections [He et al., 2016], which introduce a shortcut path adding the input directly to the output of the sub-layers.

In this work, we consider the **pre-norm** architecture, which applies layer normalization to the inputs of the sub-layers rather than after the residual sum [Wang et al., 2019; Baevski and Auli, 2019]. This architecture is widely used in the latest LLMs as it facilitates the training of very deep networks. Formally, let us view the network as a sequence of residual blocks (i.e., sub-layers). Let $\mathbf{x}_l$ denote the input of the l-th residual block. The update rule for this block can be expressed as

$$\mathbf{x}_{l+1} \quad = \quad \mathbf{x}_l + F(\mathrm{LN}(\mathbf{x}_l), \theta_l), \tag{25}$$

where $\mathbf{x}_l \in \mathbb{R}^d$ denotes the input, $\mathbf{x}_{l+1} \in \mathbb{R}^d$ denotes the output, $\mathrm{LN}(\cdot)$ denotes the layer normalization function, $F(\cdot)$ denotes the transformation function of the sub-layer (either self-attention or FFN), and θ_l denotes the parameters of the function $F(\cdot)$.

Here we absorb $\mathrm{LN}(\cdot)$ into $F(\cdot)$, thereby making $F(\cdot)$ a composite function that performs layer normalization followed by the sub-layer transformation. Then, we can rewrite Eq. (25) as

$$\mathbf{x}_{l+1} \quad = \quad \mathbf{x}_l + F(\mathbf{x}_l, \theta_l). \tag{26}$$

This formulation closely resembles the Euler method as discussed in the previous section. To connect Transformer residual blocks to ODEs, let us consider an ODE that describes how a state $\mathbf{x}(t)$ changes over time t

$$\frac{d\mathbf{x}(t)}{dt} \quad = \quad f(\mathbf{x}(t), \theta(t), t) \tag{27}$$

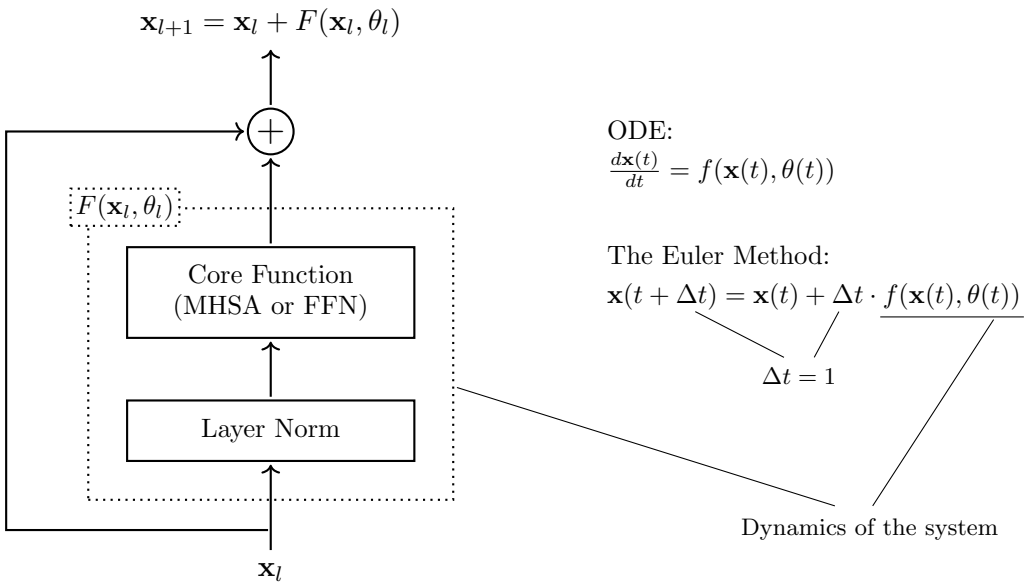


Figure 4: Illustration of a Transformer sub-layer and the corresponding ODE. The sub-layer can be described as $\mathbf{x}_{l+1} = \mathbf{x}_l + F(\mathrm{LN}(\mathbf{x}_l), \theta_l)$, where the addition of $\mathbf{x}_l$ represents the residual connection and $F(\mathrm{LN}(\mathbf{x}_l), \theta_l)$ represents the transformation defined by the sub-layer. This corresponds to the Euler discretization of the ODE $\frac{d\mathbf{x}(t)}{dt} = f(\mathbf{x}(t), \theta(t))$. More specifically, for the update rule $\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t \cdot f(\mathbf{x}(t), \theta(t))$, we treat $f(\mathbf{x}(t), \theta(t))$ as the transformation function of the sub-layer, and set $\Delta t = 1$.

This is a **non-autonomous** ODE, as the dynamics depend explicitly on time through the time-varying parameters $\theta(t)$. We will use $f(\mathbf{x}(t), \theta(t))$ as a shorthand for $f(\mathbf{x}(t), \theta(t), t)$ to keep the notation uncluttered.

Given a discrete time step size Δt, the state at the next time step can be approximated by the Euler discretization

$$\mathbf{x}(t + \Delta t) \quad = \quad \mathbf{x}(t) + \Delta t \cdot f(\mathbf{x}(t), \theta(t)). \tag{28}$$

By comparing Eq. (26) and Eq. (28), we can establish a direct correspondence between the residual connections in Transformers and the Euler discretization of the ODE. Specifically, if we set the time step size to $\Delta t = 1$, the update rule of the Transformer sub-layer becomes mathematically equivalent to a single step of the Euler method.

In this analogy, the layer index l corresponds to the discrete time step t, and the hidden state $\mathbf{x}_l$ corresponds to the state $\mathbf{x}(t)$. As a result, the transformation function $F(\mathbf{x}_l, \theta_l)$ acts as the derivative function $f(\mathbf{x}(t), \theta(t))$ that governs the dynamics in the ODE. It is important to note that while $\theta(t)$ in the continuous ODE formulation represents a parameter trajectory defined continuously over time t, the distinct parameters θ_l in the Transformer can be viewed as the values of this continuous trajectory discretized at integer time steps. Figure 4 illustrates the correspondence between a Transformer sub-layer and the ODE.

Thus, the forward pass of a Transformer with L layers (i.e., $2L$ sub-layers) can be interpreted as numerically integrating an ODE from an initial time $t = 0$ to a final time $t = 2L$ using the Euler method with a unit step size. This perspective suggests that training a Transformer is essentially learning the trajectory of a dynamical system. In this model, the depth of the network is not merely a structural hyperparameter, but can be viewed as the duration of a dynamic process governing the transformation of input representations. Such an insight also motivates the development of neural models based on continuous-time ODEs, such as neural ODEs, which will be discussed in Section 3.

2.2 ODE-inspired Architecture Desgin

Although standard pre-norm Transformers can be interpreted as the Euler discretization of ODEs, they have limitations from the perspective of numerical analysis. For example, the Euler discretization is a first-order method and has a local truncation error of $O((\Delta t)^2)$. In Transformers, the error can accumulate significantly as the network grows deeper. As a result, the trajectory computed by the Transformer may deviate from the ideal dynamics of the underlying ODE. Also, the standard Euler method is an explicit method, which is typically considered less numerically stable than implicit ODE solvers. Furthermore, Transformers update representations by executing self-attention and feed-forward layers sequentially, which introduces splitting errors in modeling the coupled dynamics. Therefore, several studies have proposed replacing the simple Euler discretization with more sophisticated schemes to design better neural architectures.

2.2.1 Architectures Interpreted from Physical Systems

One approach is to consider alternative frameworks used in other fields where ODEs have well-established foundations. For example, Lu et al. [2020] interpret Transformers as solvers for the convection-diffusion equation in multi-particle dynamic systems. To be more specific, let us consider a Transformer layer consisting of an MHSA sub-layer and an FFN sub-layer, given by

$$\mathbf{x}_{l+1} = \mathbf{x}_l + F_{\text{att}}(\mathbf{x}_l, \omega_l) \tag{29}$$

$$\mathbf{x}_{l+2} = \mathbf{x}_{l+1} + F_{\text{ffn}}(\mathbf{x}_{l+1}, \pi_{l+1}) \tag{30}$$

where $F_{\text{att}}(\cdot)$ and $F_{\text{ffn}}(\cdot)$ denote the functions of the MHSA and FFN sub-layers respectively, and ω_l and π_{l+1} are the corresponding parameters. By substituting Eq. (29) into Eq. (30) and applying a layer index k (letting $\mathbf{h}_k = \mathbf{x}_{2k}$), we obtain the unified expression for the k-th Transformer layer:

$$\mathbf{h}_{k+1} = \mathbf{h}_k + F_{\text{att}}(\mathbf{h}_k, \omega_k) + F_{\text{ffn}}\big(\mathbf{h}_k + F_{\text{att}}(\mathbf{h}_k, \omega_k), \pi_k\big) \tag{31}$$

From the perspective of physical systems, this formulation is analogous to the convection–diffusion equation, which models a diffusion process and a convection process simultaneously. Let us define the continuous operators $\text{Att}(\mathbf{h}) = F_{\text{att}}(\mathbf{h}, \omega_k)$ and $\text{FFN}(\mathbf{h}) = F_{\text{ffn}}(\mathbf{h}, \pi_k)$. We can view Eq. (31)

as a numerical discretization of the following ODE:

$$\frac{d\mathbf{h}(t)}{dt} = \mathrm{Att}(\mathbf{h}(t)) + \mathrm{FFN}(\mathbf{h}(t)) \tag{32}$$

This ODE differs from those presented earlier in this paper because it contains two distinct terms on the right-hand side. In physics and mathematics, this type of ODE is common, e.g., a physical system evolves under two different forces simultaneously. Solving this combined equation directly is often difficult or computationally expensive. In this case, we often resort to numerical techniques called **operator splitting schemes**, which allow us to approximate the solution by solving the dynamics of each term separately and alternately. From this viewpoint, the sequential update rule of the standard Transformer (Eq. (31)) inherently corresponds to the **Lie-Trotter splitting scheme**, implemented via Euler discretization. In this method, we first advance the state using the diffusion dynamics

$$\frac{d\mathbf{h}(t)}{dt} = \mathrm{Att}(\mathbf{h}(t)) \tag{33}$$

Applying a single Euler step to the initial state $\mathbf{h}_k$ yields the intermediate state $\mathbf{h}'_k = \mathbf{h}_k + \mathrm{Att}(\mathbf{h}_k)$. This is then used as the initial condition for the convection dynamics

$$\frac{d\mathbf{h}(t)}{dt} = \mathrm{FFN}(\mathbf{h}(t)) \tag{34}$$

Similarly, applying a second Euler step yields $\mathbf{h}_{k+1} = \mathbf{h}'_k + \mathrm{FFN}(\mathbf{h}'_k)$. Substituting $\mathbf{h}'_k$ back recovers the exact form of the standard Transformer.

However, the Lie-Trotter splitting is asymmetric (i.e., the order matters), and is only a first-order approximation. To achieve higher accuracy and stability, one can employ the **Strang splitting scheme** (also known as symmetric splitting), which provides a second-order approximation [Strang, 1968]. To be more specific, the Strang splitting scheme uses a symmetric arrangement of the operators. Instead of simply applying $\mathrm{Att}(\cdot)$ followed by $\mathrm{FFN}(\cdot)$, it performs a half-step of one operator, a full step of the other, and finally another half-step of the first operator. Mathematically, if we choose to split the FFN operator, the update rule for a Transformer layer corresponds to the following sequence of compositions

$$\mathbf{h}_{k+1/3} = \mathbf{h}_k + \frac{1}{2}F_{\mathrm{ffn}}(\mathbf{h}_k, \pi_k^{(1)}) \tag{35}$$

$$\mathbf{h}_{k+2/3} = \mathbf{h}_{k+1/3} + F_{\mathrm{att}}(\mathbf{h}_{k+1/3}, \omega_k) \tag{36}$$

$$\mathbf{h}_{k+1} = \mathbf{h}_{k+2/3} + \frac{1}{2}F_{\mathrm{ffn}}(\mathbf{h}_{k+2/3}, \pi_k^{(2)}) \tag{37}$$

This implies a sandwiched structure where one MHSA sub-layer is placed between two FFN sub-layers. The superscripts (1) and (2) indicate that while mathematical Strang splitting typically reuses the same operator, in a neural network context, we can parameterize the two FFN sub-layers with distinct parameters $\pi_k^{(1)}$ and $\pi_k^{(2)}$ to increase expressivity. Theoretically, however, this

structure guarantees an improvement in the local truncation error to $O((\Delta t)^3)$ only when the parameters are shared (i.e., $\pi_k^{(1)} = \pi_k^{(2)}$).

The connection between operator splitting schemes and Transformer architectures provides a powerful tool for architecture design: rather than relying solely on trial-and-error, we can derive novel and potentially superior network structures by applying advanced numerical discretization and splitting techniques to the underlying continuous dynamics. For instance, other higher-order splitting methods or adaptive splitting schemes could theoretically motivate further variations of Transformers.

2.2.2 Architectures based on Higher-order Methods

The Euler method is a first-order method. A natural improvement is to consider higher-order numerical solvers, which use more sophisticated update rules to achieve higher precision.

A widely-used family of high-order solvers is the Runge-Kutta (RK) series. While the Euler method estimates the solution using only the derivative at the current state, RK methods improve approximation by aggregating derivatives calculated at multiple intermediate points. For instance, in Transformers, the standard residual block can be replaced with a block modeled by RK methods, as proposed by Li et al. [2022a]. A general form of the explicit n-th order RK method (where $n \leq 4$) with a step size of 1 is given by[2]

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \sum_{i=1}^{n} \gamma_i F_i \tag{38}$$

$$F_1 = F(\mathbf{x}_k, \theta_k) \tag{39}$$

$$F_i = F\left(\mathbf{x}_k + \sum_{j=1}^{i-1} \beta_{ij} F_j, \theta_k\right) \tag{40}$$

where γ_i, β_{ij} are the coefficients determined by the specific RK scheme (e.g., the classical RK4). In this architecture, each F_i acts as an intermediate estimation of the slope, allowing the network to probe the function landscape at multiple points before making the final update. Theoretically, this RK method achieves a local truncation error of $O((\Delta t)^{n+1})$. Note that the parameters θ_k are shared within the block. Although we need to evaluate the function $F(\cdot)$ n times, it reuses the same parameters θ_k for each evaluation. This means that a high-order RK method increases the computational cost to achieve higher precision, but it is parameter efficient.

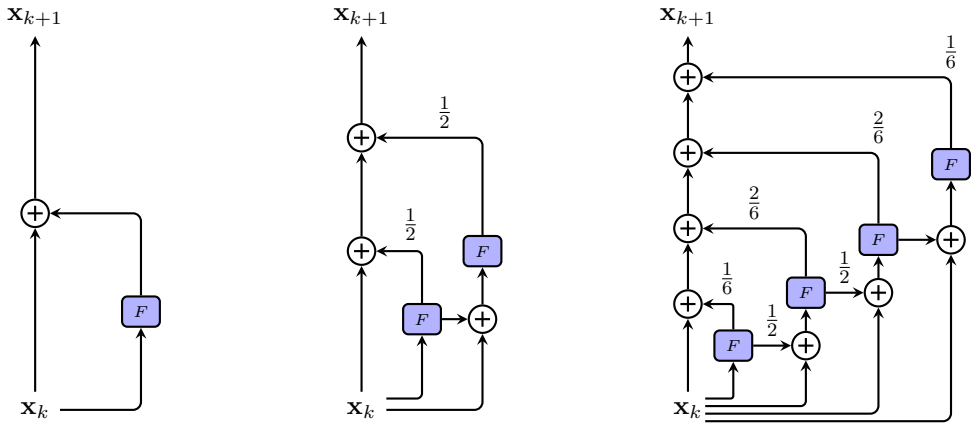
As an example, an RK2-based Transformer sub-layer can be formulated as

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \frac{1}{2}(F_1 + F_2) \tag{41}$$

$$F_1 = F(\mathbf{x}_k, \theta_k) \tag{42}$$

$$F_2 = F(\mathbf{x}_k + F_1, \theta_k) \tag{43}$$

2. Strictly speaking, the summation limit n in the equation represents the number of *stages* (function evaluations), while the term *order* refers to the convergence rate of the truncation error. For explicit RK methods, the number of stages equals the order only when the order is ≤ 4. Achieving an order $p > 4$ requires strictly more than p stages (e.g., a 5th-order method requires at least 6 stages). In this context, we assume $n \leq 4$ for simplicity.



(a) Pre-norm (b) RK2 (c) RK4

Figure 5: Comparison of Transformer sub-layer architectures inspired by numerical ODE solvers [Li et al., 2022a]. Subfigure (a): The baseline pre-norm architecture represents a single-step update based on the Euler method. Subfigures (b) and (c): The RK2 and RK4 architectures use multiple function evaluations (stages) within a single block. The output is computed as a weighted aggregation of these intermediate states, as indicated by the coefficients on the edges.

In this configuration, F_1 serves as a predictor, that provides an initial guess of the slope, while F_2 acts as a "refiner", that adjusts the update based on the estimated future state. Figure 5 shows illustrations of RK methods with different orders.

Note that these RK-based architectures introduce a new dimension for scaling neural networks. While current LLM scaling primarily focuses on increasing sequence length [Guo et al., 2025] or inter-layer depth [Yang et al., 2024], RK schemes enable intra-layer depth scaling. This approach provides a parameter-efficient mechanism to decouple computational budget from both parameter count and sequence length.

2.2.3 Architectures based on Implicit Methods

Both the Euler method and the standard Runge-Kutta methods described above belong to the category of **explicit methods**, where the next state $\mathbf{x}_{k+1}$ is computed directly from the current state $\mathbf{x}_k$. However, explicit methods can sometimes lack numerical stability. To further improve the model's robustness, one approach is to adopt **implicit methods**, where the next state is defined by an equation involving the unknown future state itself.

Designing architectures based on implicit methods is challenging because solving for $\mathbf{x}_{k+1}$ requires complex root-finding operations. Therefore, more advanced methods are generally used. One example is the PCformer [Li et al., 2024], which adopts a predictor-corrector paradigm with a high-order explicit predictor (like RK) and a multistep implicit corrector.

In the prediction phase, instead of a simple Euler update, PCformer uses a high-order explicit solver (such as the 4th-order RK method) to obtain a high-quality initial estimate, denoted as $\hat{\mathbf{x}}_{k+1}$. For instance, we can use Eqs. (41-43) to get a 2nd-order RK estimate. As an improvement,

PCformer uses an **exponential moving average** (EMA) for coefficient learning (i.e., determining γ_i in Eq. (38)). In this method, it is hypothesized that higher-order intermediate approximations (e.g., F_4 in RK4) are more accurate than lower-order ones (e.g., F_1) and thus should contribute more to the output. Instead of using the fixed coefficients prescribed by standard numerical methods (e.g., $1/6, 2/6, 2/6, 1/6$ for RK4), PCformer assigns weights using an EMA decay factor b. For an n-order predictor, the output is given by

$$\hat{\mathbf{x}}_{k+1} = \mathbf{x}_k + \sum_{i=1}^{n} b(1-b)^{n-i} F_i, \tag{44}$$

where b is a learnable parameter (initialized to 0.5).

In the correction phase, the model employs an implicit linear multistep method (e.g., Adams-Moulton). Since implicit methods require the value of the function at the next time step $F(\mathbf{x}_{k+1})$, which is unknown, the predictor's output $\hat{\mathbf{x}}_{k+1}$ is used as a substitute. The final update rule involves a weighted combination of current evaluations and historical states from previous layers

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha F(\hat{\mathbf{x}}_{k+1}, \theta_k) + \sum_{j=0}^{m} \beta_j F(\mathbf{x}_{k-j}, \theta_{k-j}), \tag{45}$$

where α and β_j are learnable coefficients, and m is the number of previous layers the model remembers to help correct the current state. The equation is a learnable Adams-Moulton corrector, where the classical fixed coefficients are replaced by learnable weights α and β_j to better adapt to the data distribution. This predictor-corrector architecture reduces truncation errors and enhances generation quality. However, it introduces more computational steps, thus potentially hindering inference efficiency compared to standard Transformers.

To improve inference efficiency, we can adopt the iterative **implicit Euler Transformer** (IIET) architecture [Liu et al., 2025]. Unlike explicit methods that use only the current state to compute the next state, the implicit Euler method is defined as $\mathbf{x}_{k+1} = \mathbf{x}_k + F(\mathbf{x}_{k+1})$. However, solving this equation directly is difficult because $\mathbf{x}_{k+1}$ appears on both sides. IIET addresses this by using fixed-point iteration as a solver within each layer.

Specifically, we start with an initial estimation $\mathbf{x}_{k+1}^{(0)}$ (typically obtained via an explicit Euler step). This estimate is then progressively refined through r iterations:

$$\mathbf{x}_{k+1}^{(0)} = \mathbf{x}_k + F(\mathbf{x}_k, \theta_k) \tag{46}$$

$$\mathbf{x}_{k+1}^{(i)} = \mathbf{x}_k + F(\mathbf{x}_{k+1}^{(i-1)}, \theta_k), \quad \text{for } i = 1, \ldots, r. \tag{47}$$

The final output of the layer is $\mathbf{x}_{k+1} = \mathbf{x}_{k+1}^{(r)}$. This iterative refinement can approach the true solution of the implicit equation more closely than a single-step method. To reduce the inference cost further, one can consider dynamically pruning the number of iterations r for different layers based on their contribution to the final output.

2.3 Remarks on Stability and Stiffness

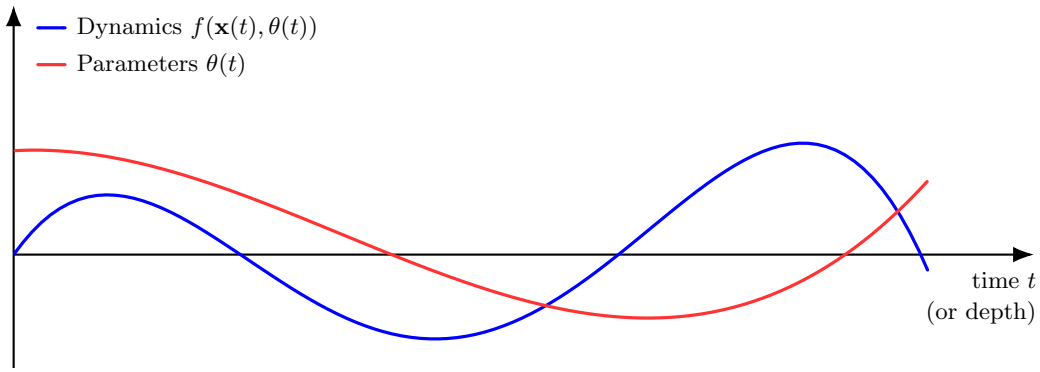
While ODEs offer theoretical interpretations in model design, the stability of the underlying dynamic system remains a challenge when applying these concepts to deep neural networks. From the perspective of ODEs, the stability of a Transformer is closely related to the smoothness of the information flow and the stiffness of the ODE.

An issue in modeling Transformers as ODEs is that the learned dynamics can be stiff. Here **stiffness** refers to the phenomenon where the continuous dynamics of the hidden states evolve at vastly different rates, e.g., some components change rapidly while others vary slowly. As a result, a stiff ODE is an ODE for which standard explicit numerical methods are severely unstable, thus requiring very small step sizes to solve the equation stably. In the context of neural networks, this implies that standard explicit solvers (or fixed-depth residual connections) may fail to propagate signals effectively, as they are prone to numerical instability when attempting to capture these rapid transient changes without a sufficiently fine-grained discretization. As noted in classical numerical analysis literature [Hairer and Wanner, 1996], explicit methods like the standard Euler or Runge-Kutta methods have a limited region of stability. When applied to stiff problems, they generally require a small step size to avoid numerical divergence. By contrast, implicit methods (e.g., the methods presented in 2.2.3) are often preferred for stiff problems because they possess a much larger stability region.

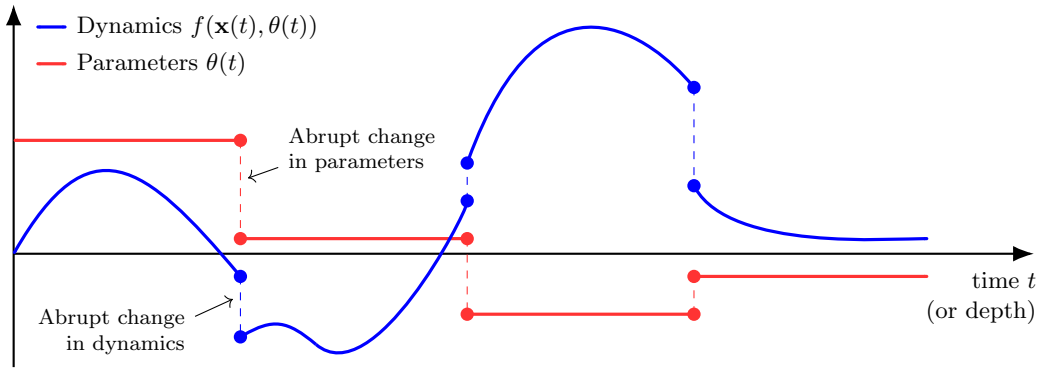
For Transformers, this issue is more severe due to the discrete nature of the layers. In these models, the parameters θ are independent and can differ drastically between layers. This contradicts the assumption of smooth dynamics required for stable ODE solutions. Recall the ODE formulation $\frac{d\mathbf{x}(t)}{dt} = F(\mathbf{x}(t), \theta(t))$, where $\theta(t)$ represents parameters that vary continuously with time t. However, in standard Transformers, parameters are fixed for each layer, thereby discretizing $\theta(t)$ into a step function (i.e., $\theta_1, \theta_2, \dots$). These abrupt changes in parameters cause the system dynamics to shift sharply from one layer to the next. For instance, in LLMs, it is often observed that the first few layers possess markedly different parameter values, resulting in intermediate representations that differ significantly across layers. Figure 6 shows an illustration of this problem.

To mitigate the numerical instability caused by stiffness and abrupt parameter shifts, it is natural to constrain the variability of the function F. One common way to achieve this is through the Lipschitz continuity condition. Specifically, to ensure a well-conditioned ODE that can be solved stably, there must exist a constant K such that $\|F(\mathbf{x}_1) - F(\mathbf{x}_2)\| \leq K\|\mathbf{x}_1 - \mathbf{x}_2\|$. In neural networks, the Lipschitz constant is closely related to the spectral norm of the weight matrices in each layer [Neyshabur, 2017; Miyato et al., 2018]. The spectral norm is defined as the largest singular value of the matrix. Gouk et al. [2021] demonstrate that constraining the spectral norm can effectively bound the Lipschitz constant. This constraint not only smoothens the dynamics but also ensures that the amplification of signals through the depth of the network remains controlled. This helps prevent numerical divergence.

More broadly, we can achieve higher stability using explicit regularization techniques. Two common strategies can be considered:



(a) Smooth evolution of system dynamics and parameters over time



(b) Sharp transitions in system dynamics caused by abrupt changes in parameters

Figure 6: Illustration of the shift in system dynamics caused by abrupt parameter changes. The blue line represents the evolution of system dynamics $f(\mathbf{x}(t), \theta(t))$, and the red line represents the evolution of parameters $\theta(t)$. Subfigure (a) shows an ideal scenario where both parameters and dynamics evolve smoothly and continuously over time. Subfigure (b) shows a realistic scenario where parameters in discrete layers behave as a step function. The abrupt discontinuities in parameters at specific time steps lead to sharp transitions in the system dynamics.

- **Parameter Regularization**. To enforce a smoother evolution of the dynamics with respect to depth in Transformers, one can introduce a penalty to the loss function. The following equation shows an example of the penalty term [Haber and Ruthotto, 2017]

$$\mathcal{L}_{\text{param}} = \lambda_{\text{param}} \sum_{k=1}^{L-1} ||\theta_{k+1} - \theta_k||_2^2, \tag{48}$$

where λ_{param} is the weight of the penalty, and L is the depth of the Transformer layer stack[3]. This term constrains adjacent layers to have similar parameters. In the limiting case (e.g., as $\lambda_{\text{param}} \to \infty$), this constraint forces $\theta_{k+1} \approx \theta_k$, which results in parameter-sharing architectures [Dehghani et al., 2019; Lan et al., 2020]. In such models, θ becomes constant across layers, and the non-autonomous ODE is transformed into an autonomous system (time-invariant). This effectively treats the depth dimension as the time step of a recurrent neural network. In general, autonomous systems are more stable and parameter-efficient.

- **State Regularization**. This method directly controls the change of output across layers. For instance, we can penalize large divergence between outputs of adjacent layers, like this

$$\mathcal{L}_{\text{state}} = \lambda_{\text{state}} \sum_{k=1}^{L-1} ||\mathbf{x}_{k+1} - \mathbf{x}_k||_2^2, \tag{49}$$

where λ_{state} is the weight of this regularization term. In this sense, normalization methods like layer normalization can be seen as an effective way to regularize the hidden states in Transformers. Another approach involves taking the outputs of previous layers as input to produce a combined output. Such layer combination techniques have been extensively used in training deep neural networks [Huang et al., 2017; Wang et al., 2018]. From a numerical analysis perspective, these architectures can be interpreted as multi-step numerical integrators (e.g., linear multistep methods), where the approximation of the next state relies on a history of previous states rather than solely on the immediately preceding one. This sequential dependency effectively smooths the trajectory of the hidden states $\mathbf{x}(t)$. By aggregating information from multiple previous steps, the propagation of signals can be relatively robust even when the discrete transformations vary rapidly.

3. Neural ODEs and Flows

In the previous section, we interpreted discrete residual networks and Transformers as discretized approximations of ODEs. One significant limitation of such models is that the discretizations use fixed step sizes (i.e., $\Delta t = 1$), which prevents the models from adapting their computational effort to the complexity of the input instance. Such coarse discretizations are often ill-suited for problems requiring fine-grained modeling, where the underlying dynamics may evolve rapidly and require smaller time steps for accurate approximation. Furthermore, since the number of steps is determined in the model design phase, the network depth becomes a static hyperparameter rather than an adaptive property.

In this section, we begin by looking at **neural ODEs** [Chen et al., 2018], which model the evolution of system dynamics by combining neural networks and continuous-time ODEs. An important property of neural ODEs is that they treat the network depth as a continuous time variable. Instead of specifying a fixed sequence of discrete layers, the derivative of the hidden

3. Here, a Transformer layer comprises two sub-layers. θ_k denotes the collective parameters of the entire layer.

state is defined by a continuous-time ODE, and the final output is obtained by solving this ODE using a black-box ODE solver. Moreover, we describe the training of neural ODEs, in particular the adjoint sensitivity method which provides a memory-efficient way to optimize such models. Finally, we discuss the concept of **continuous normalizing flows**, which is a natural extension of neural ODEs for describing the continuous evolution of probability densities. This framework lays the foundation for further discussions on generative modeling (see Section 4).

3.1 Neural ODEs

We have seen that in residual networks, the hidden state is updated according to $\mathbf{x}_{l+1} = \mathbf{x}_l + \Delta t \cdot F(\mathbf{x}_l, \theta_l)$. This equation corresponds to the Euler discretization of a continuous-time ODE as $\Delta t \to 0$

$$\frac{d\mathbf{x}(t)}{dt} = f(\mathbf{x}(t), \theta(t)). \tag{50}$$

Here, we use two function notations F and f to distinguish between the discrete residual mapping and the instantaneous rate of change of the state, respectively.

A key distinction between the two models lies in the parameterization. In standard residual networks, each layer l has its own distinct set of parameters θ_l. In the continuous limit expressed in Eq. (50), the parameters $\theta(t)$ can vary continuously with time. However, in practice, standard neural ODEs typically use a shared set of parameters θ across the entire integration interval. To allow the dynamics to vary with depth (time), the current time t is explicitly appended as an input to the network. Thus, the dynamics can be formulated using a non-autonomous ODE

$$\frac{d\mathbf{x}(t)}{dt} = f(\mathbf{x}(t), \theta, t). \tag{51}$$

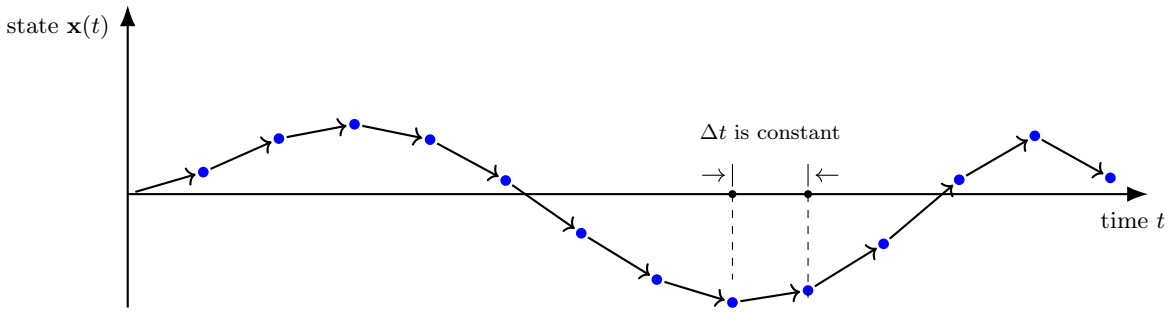
The introduction of the continuous time variable t as a replacement for the layer index l allows the model to evaluate the state at any time point rather than at fixed time steps.

Neural ODEs are called *neural* because the dynamical function $f(\cdot)$ is a neural network. In other words, we use a neural network to estimate the dynamics of the system. Given a neural ODE, there are two fundamental processes, as in supervised learning problems:

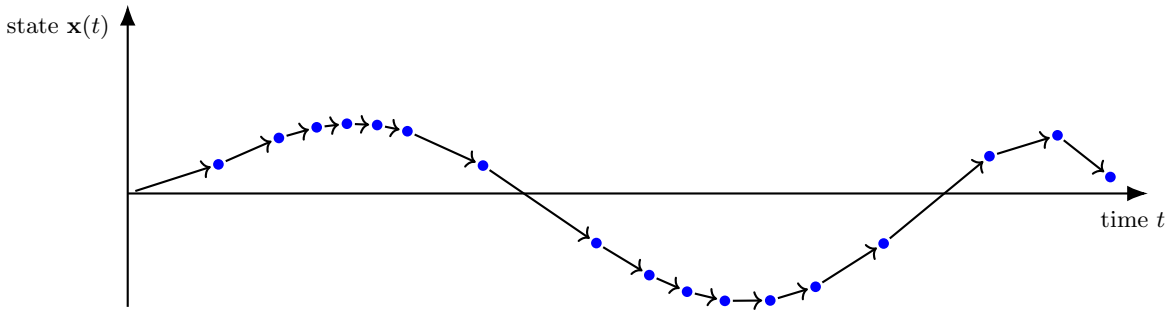
- **Inference** (Forward Pass). The inference process aims to compute the state at any time given the parameters.
- **Training** (Backward Pass). The training process aims to optimize the parameters θ to minimize a loss function.

For inference, the input to the model is the initial state $\mathbf{x}(t_0)$, and the output is the state $\mathbf{x}(t_1)$ at some final time t_1. That is, the forward pass is essentially the task of solving this IVP. The output state $\mathbf{x}(t_1)$ is the solution to the ODE, and is computed by integrating the dynamics

$$\mathbf{x}(t_1) = \mathbf{x}(t_0) + \int_{t_0}^{t_1} f(\mathbf{x}(\tau), \tau; \theta) d\tau. \tag{52}$$



(a) A system whose state evolves with a fixed time step (like residual networks)



(b) A system whose state evolves continuously or with a dynamic time step (like neural ODEs)

Figure 7: Illustrations of ODE solving with fixed and dynamic step sizes. Subfigure (a) shows a trajectory generated with a constant step size, similar to the discrete steps in a residual network (which can be seen as an Euler discretization of an ODE). Subfigure (b) shows a trajectory obtained with an dynamic step-size solver, which can be adopted in neural ODEs that model continuous state evolution.

Since the integral of a complex neural network f cannot be evaluated analytically, the output is obtained using a numerical ODE solver (e.g., Euler, Runge-Kutta, or more advanced methods). This allows the evaluation

$$\mathbf{x}(t_1) = \text{ODESolve}(\mathbf{x}(t_0), f, t_0, t_1, \theta). \tag{53}$$

Here, ODESolve$(\cdot)$ represents a general ODE solver, which is typically implemented using off-the-shelf numerical integration tools. This formulation decouples the definition of the dynamics from the computation of the trajectory. So the solver can adapt the step size based on the complexity of the trajectory, which is a significant feature not present in fixed-depth residual networks. Figure 7 shows illustration of ODE solving with fixed and dynamic step sizes.

The training of neural ODEs can be seen as a standard task of training deep neural networks, where the parameters θ are updated iteratively by computing the gradient of the loss function with

respect to these parameters. A straightforward approach is to perform standard backpropagation through the operations of the ODE solver. However, this often suffers from a high memory cost because it requires storing the intermediate states of the entire trajectory to apply the chain rule. The memory bottleneck may restrict the use of fine-grained solvers or long integration times. To address this, neural ODEs are typically trained using the **adjoint sensitivity method**, which computes gradients with a constant memory cost with respect to the number of integration steps. We will discuss the details of training neural ODEs further in Section 3.2.

Simply put, neural ODEs use a neural network $f(\cdot)$ to approximate the dynamics of a system $\frac{d\mathbf{x}(t)}{dt}$. In the training stage, we optimize this network to model these dynamics. Then, in the testing stage, we can recover the state of the system at any given moment by simply solving the neural ODE.

3.2 The Adjoint Sensitivity Method

The adjoint sensitivity method (or the adjoint method) is a classical technique from the field of optimal control used to compute sensitivities for systems governed by differential equations. In deep learning, it can be interpreted as continuous-time backpropagation. The core idea is to introduce a set of auxiliary variables, known as the **adjoint states**, which track how the gradient of the loss evolves backwards through time. It computes the gradients of the model parameters by solving a second differential equation, rather than explicitly applying the chain rule to the discrete operations of the forward pass.

This distinction should be emphasized when comparing it to standard backpropagation. In a naive implementation (often referred to as discretize-then-optimize), one would treat the numerical ODE solver as a computation graph with a finite number of layers and backpropagate directly through the internal operations of the solver. However, since an ODE solver may take thousands of small steps to achieve high precision, standard backpropagation requires storing the intermediate activations for every single step. This leads to prohibitive memory usage that scales linearly with the number of integration steps ($O(T)$). In other words, the discretization of ODE solving with a large number of steps makes the training memory-intensive, or sometimes even infeasible.

Formally, let us define the adjoint state $\mathbf{a}(t)$ as the gradient of the loss function $\mathcal{L}$ with respect to the hidden state $\mathbf{x}(t)$ at any given time t

$$\mathbf{a}(t) \quad = \quad \frac{\partial \mathcal{L}}{\partial \mathbf{x}(t)}. \tag{54}$$

Intuitively, $\mathbf{a}(t)$ represents the sensitivity of the final loss to a small perturbation in the state at time t. Since the loss is computed based on the final state $\mathbf{x}(t_1)$, the boundary condition for the adjoint state is known

$$\mathbf{a}(t_1) \quad = \quad \frac{\partial \mathcal{L}}{\partial \mathbf{x}(t_1)}. \tag{55}$$

Note that while $\mathbf{x}(t)$ is solved forward from t_0 to t_1, the adjoint state $\mathbf{a}(t)$ must be solved backwards from t_1 to t_0, using $\mathbf{a}(t_1)$ as the initial value for the backward integration. Figure 8 shows a high-level illustration of the method.

3.2.1 Adjoint Dynamics

Instead of backpropagating through steps of the solver, we compute $\mathbf{a}(t)$ by solving a new ODE. To understand the dynamics of $\mathbf{a}(t)$, consider the discretization of the forward pass using a simple Euler method with a step size Δt. The state update from t to $t + \Delta t$ is

$$\mathbf{x}(t + \Delta t) \quad = \quad \mathbf{x}(t) + \Delta t \cdot f(\mathbf{x}(t), \theta, t). \tag{56}$$

Applying the chain rule, the gradient of the loss with respect to the hidden state $\mathbf{x}(t)$ can be expressed as

$$\begin{aligned} \mathbf{a}(t) &= \frac{\partial \mathcal{L}}{\partial \mathbf{x}(t)} \\ &= \left(\frac{\partial \mathbf{x}(t + \Delta t)}{\partial \mathbf{x}(t)}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{x}(t + \Delta t)} \\ &= \left(\frac{\partial \mathbf{x}(t + \Delta t)}{\partial \mathbf{x}(t)}\right)^\top \mathbf{a}(t + \Delta t), \end{aligned} \tag{57}$$

where $\frac{\partial \mathbf{x}(t+\Delta t)}{\partial \mathbf{x}(t)} \in \mathbb{R}^{d \times d}$ is the Jacobian matrix and $\mathbf{a}(t), \mathbf{a}(t + \Delta t) \in \mathbb{R}^d$ are column vectors.

Substituting Eq. (56) into Eq. (57) yields

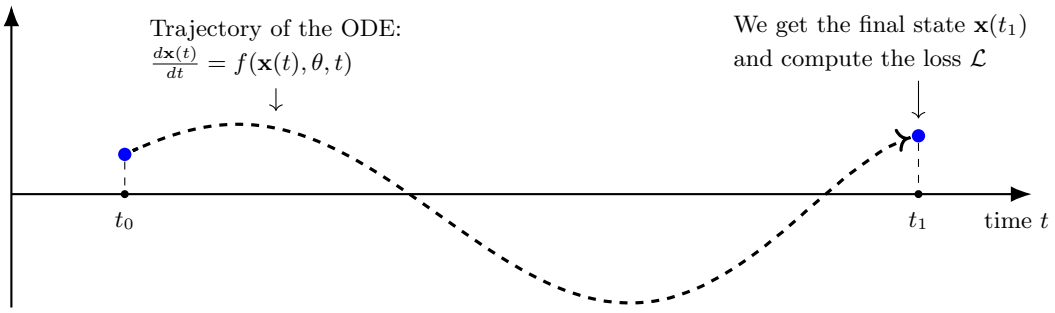
$$\begin{aligned} \mathbf{a}(t) &= \left(\frac{\partial \big(\mathbf{x}(t) + \Delta t \cdot f(\mathbf{x}(t), \theta, t)\big)}{\partial \mathbf{x}(t)}\right)^\top \mathbf{a}(t + \Delta t) \\ &= \left(\frac{\partial \mathbf{x}(t) + \partial \Delta t \cdot f(\mathbf{x}(t), \theta, t)}{\partial \mathbf{x}(t)}\right)^\top \mathbf{a}(t + \Delta t) \\ &= \left(\mathbf{I} + \Delta t \cdot \frac{\partial f(\mathbf{x}(t), \theta, t)}{\partial \mathbf{x}(t)}\right)^\top \mathbf{a}(t + \Delta t). \end{aligned} \tag{58}$$

Rearranging this equation yields the difference quotient for the adjoint state

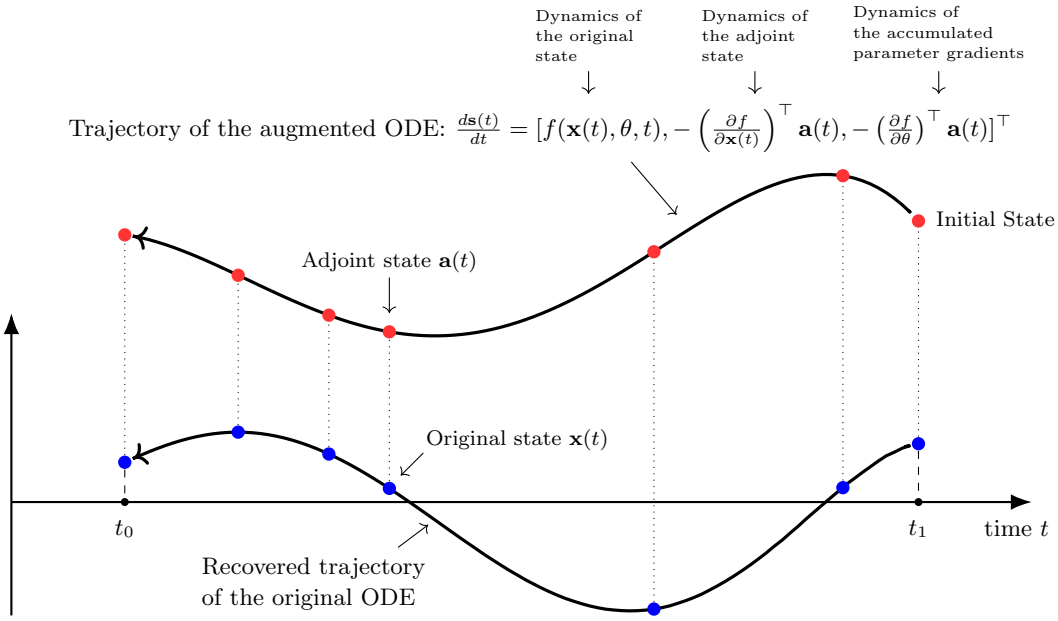
$$\frac{\mathbf{a}(t + \Delta t) - \mathbf{a}(t)}{\Delta t} \quad = \quad -\left(\frac{\partial f(\mathbf{x}(t), \theta, t)}{\partial \mathbf{x}(t)}\right)^\top \mathbf{a}(t + \Delta t). \tag{59}$$

By taking the limit as $\Delta t \to 0$, we obtain the continuous-time dynamics of the adjoint state. This leads to the differential equation describing the instantaneous rate of change of $\mathbf{a}(t)$

$$\frac{d\mathbf{a}(t)}{dt} \quad = \quad -\left(\frac{\partial f(\mathbf{x}(t), \theta, t)}{\partial \mathbf{x}(t)}\right)^\top \mathbf{a}(t). \tag{60}$$



(a) Forward pass (integrating the dynamics of the ODE from t_0 to t_1)



(b) Backward pass (integrating the dynamics of the augmented ODE from t_1 to t_0)

Figure 8: Illustration of the adjoint sensitivity method in neural ODEs [Chen et al., 2018]. There are two steps. In the forward pass (subfigure (a)), the state $\mathbf{x}(t)$ evolves from t_0 to t_1 by integrating the dynamics defined by the neural network $f(\mathbf{x}(t), \theta, t)$. The final state $\mathbf{x}(t_1)$ is used to compute the scalar loss $\mathcal{L}$. In the backward pass (subfigure (b)), instead of backpropagating through the solver's discrete steps, an augmented ODE system is solved backwards from t_1 to t_0. This augmented system simultaneously reconstructs the original state trajectory $\mathbf{x}(t)$, computes the adjoint state $\mathbf{a}(t)$ (the gradient of the loss with respect to the state), and accumulates the gradients with respect to the parameters θ. This approach allows for gradient computation with $O(1)$ memory cost as it avoids storing intermediate states during integration.

Here, $\frac{\partial f}{\partial \mathbf{x}}$ is the Jacobian of the dynamics function. This equation tells us that the adjoint state is defined by a linear ODE that depends on the Jacobian of the original function $f(\cdot)$.

Similarly, we can derive the gradient for the parameters θ. At each time step t, the parameters θ make a small contribution to the change in state, which in turn affects the loss. The instantaneous gradient contribution at time t is the sensitivity of the loss to the state ($\mathbf{a}(t)$) multiplied by the sensitivity of the state change to the parameters ($\frac{\partial f}{\partial \theta}$). The total gradient is the accumulation (integral) of these instantaneous contributions over the entire trajectory

$$\frac{\partial \mathcal{L}}{\partial \theta} = -\int_{t_1}^{t_0} \left(\frac{\partial f(\mathbf{x}(t), \theta, t)}{\partial \theta}\right)^\top \mathbf{a}(t) dt. \tag{61}$$

Note that the integral is performed backwards from t_1 to t_0, consistent with the backward pass of backpropagation.

3.2.2 The Augmented ODE and Memory Efficiency

Eqs. (60) and (61) give exact gradients but depend on the state trajectory $\mathbf{x}(t)$ at all times t. In standard backpropagation, this requires storing all intermediate states $\mathbf{x}(t)$ from the forward pass, resulting in $O(T)$ memory cost.

The key advantage of the adjoint sensitivity method is to avoid this storage by reconstructing the state trajectory backwards in time. Since the ODE describing the state evolution is deterministic and typically invertible, we can recover $\mathbf{x}(t)$ from the final state $\mathbf{x}(t_1)$ by running the dynamics backwards[4].

To compute the gradients efficiently, we construct an **augmented state** that contains the original state, the adjoint state, and the accumulated parameter gradients

$$\mathbf{s}(t) = \begin{bmatrix} \mathbf{x}(t) \\ \mathbf{a}(t) \\ \frac{\partial \mathcal{L}}{\partial \theta}(t) \end{bmatrix}, \tag{62}$$

where $\frac{\partial \mathcal{L}}{\partial \theta}(t)$ is the backwards integral from t_1 to t, i.e., $\frac{\partial \mathcal{L}}{\partial \theta}(t) = -\int_{t_1}^{t} \left(\frac{\partial f(\mathbf{x}(\tau), \theta, \tau)}{\partial \theta}\right)^\top \mathbf{a}(\tau) d\tau$. The dynamics of this combined system are solved jointly backwards from t_1 to t_0:

$$\frac{d\mathbf{s}(t)}{dt} = \begin{bmatrix} f(\mathbf{x}(t), \theta, t) \\ -\left(\frac{\partial f}{\partial \mathbf{x}(t)}\right)^\top \mathbf{a}(t) \\ -\left(\frac{\partial f}{\partial \theta}\right)^\top \mathbf{a}(t) \end{bmatrix}, \tag{63}$$

4. An ODE is not inherently invertible in the simple function sense, but the concept applies to neural ODEs and flows (i.e., solutions over time) in machine learning. Informally, invertibility means reversing the transformation to find prior states or map between spaces.

with the initial condition

$$\mathbf{s}(t_1) \quad = \quad \begin{bmatrix} \mathbf{x}(t_1) \\ \frac{\partial \mathcal{L}}{\partial \mathbf{x}(t_1)} \\ \mathbf{0} \end{bmatrix}. \tag{64}$$

The first component of the augmented dynamics is simply the original ODE. This means that as the solver integrates backwards, it reconstructs the trajectory $\mathbf{x}(t)$ on-the-fly. At any specific time t, the value of $\mathbf{x}(t)$ required to evaluate the Jacobians (in the second and third components) is readily available within the current augmented state $\mathbf{s}(t)$. As a result, we do not need to store the intermediate states of the forward pass, and the memory complexity is reduced from $O(T)$ to $O(1)$. In essence, the adjoint method trades the computational overhead of re-solving the state trajectory for memory savings. Note that this augmented ODE can be solved using off-the-shelf numerical solvers. One can also balance computational efficiency and numerical precision by simply adjusting the tolerance parameters based on specific requirements.

To wrap up, the training procedure of a neural ODE can be outlined by the following two steps:

1. **Forward Pass:** Solve the ODE from t_0 to t_1 to get $\mathbf{x}(t_1)$ and compute the loss $\mathcal{L}$.

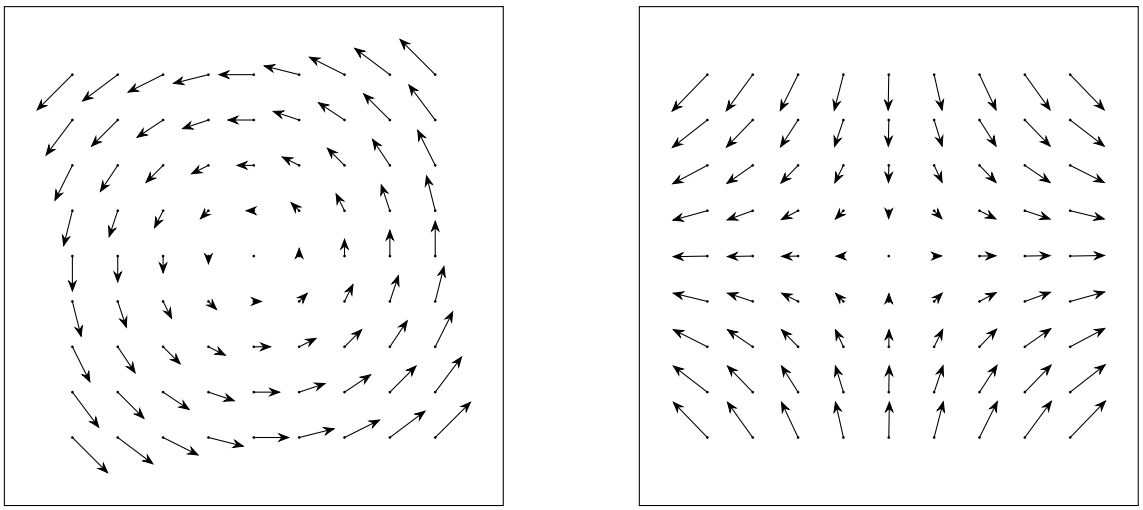
2. **Backward Pass:** Construct the initial value for the augmented state at t_1: $\mathbf{s}(t_1) = [\mathbf{x}(t_1), \frac{\partial \mathcal{L}}{\partial \mathbf{x}(t_1)}, \mathbf{0}]^\top$. Use the ODE solver to integrate the augmented dynamics $\frac{d\mathbf{s}(t)}{dt} = [f(\mathbf{x}(t), \theta, t), -\left(\frac{\partial f}{\partial \mathbf{x}(t)}\right)^\top \mathbf{a}(t), -\left(\frac{\partial f}{\partial \theta}\right)^\top \mathbf{a}(t)]^\top$ backwards from t_1 to t_0.

3.3 Neural Dynamics and Flows

Neural ODEs provide a powerful tool for describing system dynamics. A key insight is that we can use a neural network $f(\cdot)$ to model a vector field. This defines a flow which represents as a continuous, invertible transformation that evolves the state space over time. In this subsection, we formalize these concepts to move from individual state trajectories to the evolution of entire probability distributions. This framework will serve as the foundation for generative models presented in the following sections.

3.3.1 Vector Fields

A vector field is a mathematical concept used to describe a space where every point has a specific magnitude and direction associated with it. To understand it intuitively, imagine the surface of a flowing river. At any given location and any specific moment, the water moves with a particular speed and direction. If we were to draw an arrow at every point in the river representing this local velocity, the resulting collection of arrows is a vector field. As another example that is more close to our previous discussion, imagine the state space is filled with a fluid. At every point in this space, the fluid flows with a specific velocity and direction. This assignment of a velocity vector to every point in space and time is known as a vector field.



(a) Vector field $f(\mathbf{x}) = \begin{bmatrix} -x_2 \\ x_1 \end{bmatrix}$

(b) Vector field $f(\mathbf{x}) = \begin{bmatrix} \sin(x_1) + \sin(x_2) \\ \sin(x_1) - \sin(x_2) \end{bmatrix}$

Figure 9: Two examples of vector fields in the 2D plane. Each arrow represents the instantaneous velocity vector at that location. Subfigure (a) represents a linear system showing circular streamlines around the origin. Subfigure (b) represents a nonlinear system defined by sine functions, creating a periodic arrangement of fixed points and flow patterns.

Formally, a vector field on $\mathbb{R}^d$ is a map

$$f : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d. \tag{65}$$

It assigns a velocity vector to each point $\mathbf{x}$ at time t. In the context of neural ODEs, we parameterize this field using a neural network $f(\mathbf{x}(t), \theta, t)$. Given a current state $\mathbf{x}(t)$ and time t, the network outputs the time derivative by using the ODE $\frac{d\mathbf{x}(t)}{dt} = f(\mathbf{x}(t), \theta, t)$, indicating "how fast and in what direction the data is moving right now". Note that the vector field here is time-dependent as the velocity at a point changes over time.

Figure 9 shows two example vector fields in a plane. The first is a vector field defined by $f(\mathbf{x}) = [-x_2, x_1]^\top$. A particle placed in this field will move perpendicular to its position vector, tracing out a circle. The second is a vector field defined by $f(\mathbf{x}) = [\sin(x_1) + \sin(x_2), \sin(x_1) - \sin(x_2)]^\top$. This creates a more complex flow pattern. In real-world applications, by using a deep neural network to parameterize $f(\cdot)$, we can model highly complex, non-linear, and time-varying dynamics.

3.3.2 Flows and Diffeomorphisms

While a vector field describes the instantaneous velocity at any given point, a flow describes the long-term trajectory of points moving through that space. If we return to our river analogy, the vector field tells us how fast the water is moving at every specific coordinate right now. The flow,

Symbol	Name	Definition	Analogy	Interpretation
f	Vector Field	The derivative $\frac{d\mathbf{x}(t)}{dt}$	The instantaneous velocity of the water at a specific location and time.	The neural network itself. It defines how data evolves.
$\mathbf{x}(t)$	State/Trajectory	The value in $\mathbb{R}^d$ at time t.	The coordinates of a leaf at a particular moment t.	The hidden state at a certain time.
Φ	Flow Map	$\Phi : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$	A general function to find the end point given any start and any time.	The general solution of the ODE, i.e., the cumulative effect of the network over time.
Φ_t	Time-t Map	$\Phi_t(\cdot) = \Phi(\cdot, t)$ $(\mathbb{R}^d \to \mathbb{R}^d)$	A snapshot of the whole river: how the entire surface has shifted after t seconds.	The total transformation (diffeomorphism) from input to output.

Table 2: Concepts related to flows.

on the other hand, tells us where a leaf, dropped into the river at a certain position, will end up after a certain amount of time.

Mathematically, a flow map Φ is a function that tracks the evolution of all possible initial states over time. Formally, we define the flow as a mapping $\Phi : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$, which takes an initial point $\mathbf{x}(0)$ and a time parameter t to produce the state at time t. This is the solution to the initial value problem

$$\frac{\partial \Phi(\mathbf{x}(0), t)}{\partial t} \quad = \quad f(\Phi(\mathbf{x}(0), t), \theta, t), \tag{66}$$

subject to the initial condition

$$\Phi(\mathbf{x}(0), 0) \quad = \quad \mathbf{x}(0). \tag{67}$$

To analyze how the entire space evolves, we often fix the time t and consider the time-t map, denoted as $\Phi_t : \mathbb{R}^d \to \mathbb{R}^d$, where $\Phi_t(\cdot) = \Phi(\cdot, t)$. While the vector field $f(\cdot)$ represents the velocity, the map Φ_t represents the resulting transformation of the entire space. As t increases, Φ_t pushes all points in $\mathbb{R}^d$ along the trajectories defined by the vector field $f(\cdot)$. It is important to note that the state $\mathbf{x}(t)$ is the evaluation of the flow at a specific initial point, i.e., $\mathbf{x}(t) = \Phi_t(\mathbf{x}(0))$. To further clarify the distinctions and relationships between these symbols, we summarize them in Table 2.

Importantly, for a flow to be well-defined and useful in generative modeling, it must be a **homeomorphism** and, more strictly, a **diffeomorphism**. A homeomorphism is a bijective (one-to-one and onto) map $g : \mathcal{X} \to \mathcal{Y}$ such that both $g(\cdot)$ and its inverse $g^{-1}(\cdot)$ are continuous.

In intuitive terms, a homeomorphism is a rubber-sheet deformation. Imagine the state space is a sheet of flexible rubber. You can stretch it, shrink it, or twist it as much as you like, but you are not allowed to tear it (which would break continuity) or glue different parts together (which would break bijectivity). If one space can be transformed into another via a homeomorphism, the two spaces are considered topologically equivalent. From the perspective of neural ODEs, the flow $\Phi_t(\cdot)$ acts as a homeomorphism. This ensures that the global structure of the data is preserved: points that are close together in the initial distribution will remain relatively close in the transformed distribution, and no holes are created in the space.

In neural ODEs and flow-based models, we often need a stronger condition than just a homeomorphism. We need the transformation to be smooth so that we can perform calculus (e.g., compute gradients and use the change-of-variables formula). This leads us to the concept of a diffeomorphism.

A map $\Phi_t : \mathbb{R}^d \to \mathbb{R}^d$ is a diffeomorphism if:

- $\Phi_t(\cdot)$ is a homeomorphism (it is bijective, and both $\Phi_t(\cdot)$ and $\Phi_t^{-1}(\cdot)$ exist and are continuous).
- Both $\Phi_t(\cdot)$ and $\Phi_t^{-1}(\cdot)$ are continuously differentiable (C^1 or higher).

The diffeomorphic property of the flow is the foundation of many flow-based models. In a neural ODE, the vector field $f(\cdot)$ is typically parameterized by a neural network with smooth activation functions (e.g., Tanh), making $f(\cdot)$ continuously differentiable (C^1). While the Lipschitz continuity is sufficient to guarantee the uniqueness of trajectories (ensuring Φ_t is a homeomorphism) according to the Picard–Lindelöf theorem, the C^1 smoothness of the network further ensures that the flow $\Phi_t(\cdot)$ is a diffeomorphism[5].

This property has advantages in many applications, especially in generative models. First, because $\Phi_t(\cdot)$ is a diffeomorphism, the transformation is fully reversible. If we know the state of the system at time t_1, we can integrate the same ODE backward in time to recover the exact initial state at t_0. This is the foundation of exact density estimation and memory-efficient training. Second, a flow-based transformation cannot change the fundamental topology of the input space. For example, a flow cannot transform a single connected component into two separate islands, nor can it tie a knot in a string that was not already knotted. This path-preserving nature ensures that the neural network learns a continuous deformation of the probability mass, rather than a chaotic reordering of the points. Further details on flows can be found in related papers, such as [Lipman et al., 2024].

3.3.3 Continuous Normalizing Flows

While a standard flow tracks the movement of individual points, continuous normalizing flows (CNFs) extend this framework to the evolution of entire probability distributions.

Imagine that, instead of a single leaf, we release a cloud of colored dye into the river. As the water flows, the dye cloud deforms, stretches, and compresses to follow the complex currents.

5. This property is also called smooth dependence on initial conditions, which is a natural extension of the Picard–Lindelöf theorem to dynamical systems and manifolds [Coddington, 1955].

While the shape of the cloud changes, the total mass of the dye remains constant. CNFs define an invertible mapping between a simple distribution (like a Gaussian) and the complex data distribution. In this context, the term *normalizing* refers to the process of simplifying (or normalizing) complex data back into the base distribution.

To formalize the evolution of probability distributions, we must define how the flow $\Phi_t(\cdot)$, which works on points in the state space, induces a transformation on probability measures. Let $p_t(\mathbf{x})$ denote the probability density function of the state at time t. The evolution of this density is governed by the conservation of probability mass: as the space expands or contracts under the flow, the local density must adjust inversely to preserve the total mass.

Consider an arbitrary region $\mathcal{S}_0$ in the initial space at time $t = 0$. Under the flow $\Phi_t(\cdot)$, this region evolves into a new region $\mathcal{S}_t = \Phi_t(\mathcal{S}_0)$ at time t. Since probability mass is neither created nor destroyed, the total probability contained within the region must remain invariant

$$\int_{\mathcal{S}_t} p_t(\mathbf{x}) d\mathbf{x} \quad = \quad \int_{\mathcal{S}_0} p_0(\mathbf{z}) d\mathbf{z}, \tag{68}$$

where $\mathbf{x}$ and $\mathbf{z}$ represent a sample in the spaces at time t and 0, respectively. To relate these integrals, we apply the multivariate change of variables theorem to the left-hand side. We substitute the integration variable $\mathbf{x}$ with its pre-image $\mathbf{z}$ using the mapping $\mathbf{x} = \Phi_t(\mathbf{z})$. This substitution introduces the absolute value of the Jacobian determinant to account for the change in volume element

$$\int_{\mathcal{S}_0} p_t(\Phi_t(\mathbf{z})) \left| \det \frac{\partial \Phi_t(\mathbf{z})}{\partial \mathbf{z}} \right| d\mathbf{z} \quad = \quad \int_{\mathcal{S}_0} p_0(\mathbf{z}) d\mathbf{z}. \tag{69}$$

Geometrically, the Jacobian determinant term $\left| \det \frac{\partial \Phi_t(\mathbf{z})}{\partial \mathbf{z}} \right|$ can be seen as a volume expansion factor[6]. It quantifies the ratio of the infinitesimal volume in the target space to the volume in the source space. The presence of this term ensures that the integration measure is correctly scaled to reflect the distortion of the coordinate system caused by the flow.

Since Eq. (69) holds for any arbitrary region $\mathcal{S}_0$, the integrands must be equal pointwise. By setting $\mathbf{z} = \mathbf{x}(0)$ and $\Phi_t(\mathbf{z}) = \mathbf{x}(t)$, we obtain the explicit relationship between the densities

$$p_t(\mathbf{x}(t)) \left| \det \frac{\partial \mathbf{x}(t)}{\partial \mathbf{x}(0)} \right| \quad = \quad p_0(\mathbf{x}(0)). \tag{70}$$

Rearranging this term yields the standard formula for the density transformation under a diffeomorphism

$$p_t(\mathbf{x}(t)) \quad = \quad p_0(\mathbf{x}(0)) \left| \det \frac{\partial \mathbf{x}(t)}{\partial \mathbf{x}(0)} \right|^{-1}. \tag{71}$$

6. For instance, if the determinant is 2, an infinitesimal volume at $\mathbf{z}$ is stretched to become twice as large at $\mathbf{x}$.

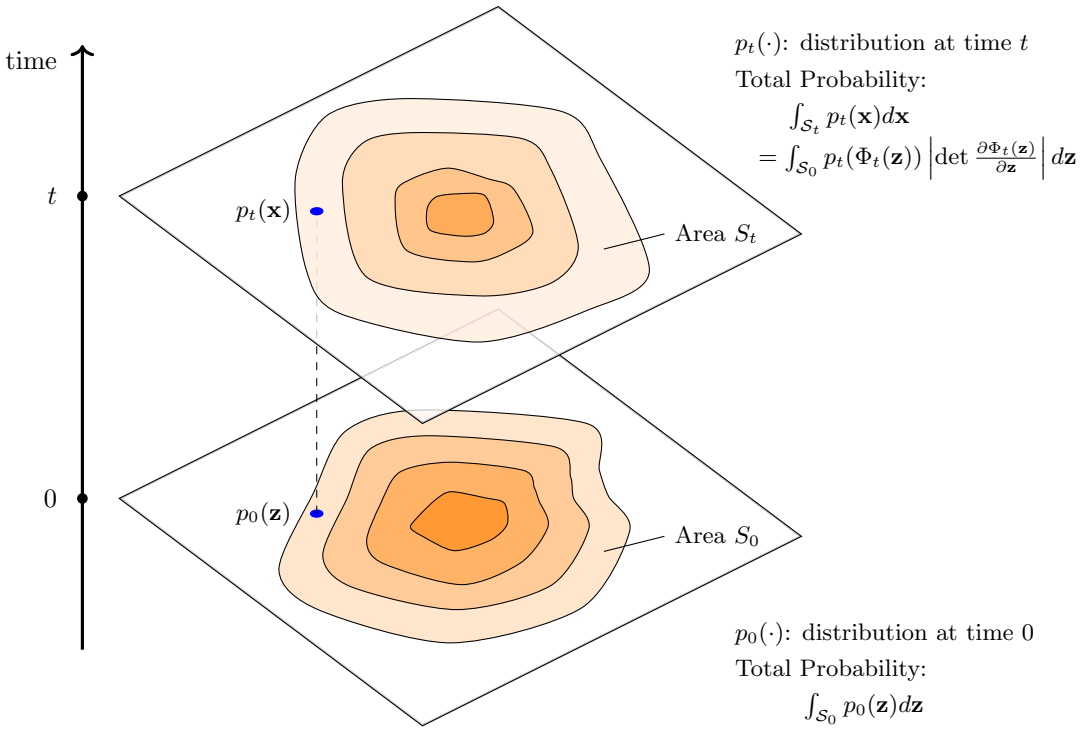


Figure 10: Illustration of probability density transformation in a CNF. The diagram visualizes the continuous evolution of a probability distribution over time. A point $\mathbf{z}$ in the base distribution is mapped to $\mathbf{x} = \Phi_t(\mathbf{z})$ at time t. The transformation of the density function $p_t(\mathbf{x})$ is governed by the conservation of probability: although the geometry of the distribution (represented by the contour maps) stretches and compresses, the integral of the density over the corresponding areas remains constant.

Here, $\frac{\partial \mathbf{x}(t)}{\partial \mathbf{x}(0)}$ is the Jacobian matrix of the flow map, which represents the accumulation of local deformations from time 0 to t. See Figure 10 for an illustration of probability density transformation in a CNF.

Taking the logarithm of Eq. (71) gives the change in log-density

$$\log p_t(\mathbf{x}(t)) \quad = \quad \log p_0(\mathbf{x}(0)) - \log \left|\det \frac{\partial \Phi_t(\mathbf{x}(0))}{\partial \mathbf{x}(0)}\right|. \tag{72}$$

This is also called the push-forward density formula under the push-forward measure [Tao, 2011].

In practice, computing the determinant of the Jacobian for the entire flow is computationally expensive. We can simplify this by considering the infinitesimal change. Following Chen et al. [2018]'s work, by differentiating the log-density with respect to time, we arrive at the **instantaneous change of variables theorem**

$$\frac{d \log p_t(\mathbf{x}(t))}{dt} \quad = \quad -\mathrm{Tr}\left(\frac{\partial f(\mathbf{x}(t), \theta, t)}{\partial \mathbf{x}(t)}\right). \tag{73}$$

Here, $\mathrm{Tr}(\frac{\partial f}{\partial \mathbf{x}})$ is the trace of the Jacobian of the vector field, which is equivalent to the divergence, denoted as $\nabla \cdot f$. This quantity describes the instantaneous rate of expansion or contraction of an infinitesimal volume. If the divergence is positive, the volume expands, and the log-density decreases at a rate equal to the divergence. Conversely, if the divergence is negative, the volume contracts and the density increases.

This continuous formulation offers a computational advantage over discrete normalizing flows. In discrete flows, ensuring the tractability of the Jacobian determinant often requires restricting the neural network architecture (e.g., to triangular matrices) [Hoogeboom et al., 2019; Tran et al., 2019]. In contrast, CNFs require only the trace of the Jacobian. This allows $f(\cdot)$ to be parameterized by arbitrary deep neural networks, as the trace can be efficiently approximated using stochastic estimators (such as the Hutchinson trace estimator), compared to the $O(d^3)$ cost of a full determinant.

At inference time, we can integrate the joint dynamics of the state and the log-density. For example, in generative modeling, given a target data point $\mathbf{x}_{\text{data}}$ at time t_1, we can compute its log-likelihood by integrating backward to from t_1 to 0

$$\begin{aligned} \log p_{t_1}(\mathbf{x}_{\text{data}}) &= \log p_0(\mathbf{x}(0)) + \int_{t_1}^{0} \mathrm{Tr}\left(\frac{\partial f(\mathbf{x}(t), \theta, t)}{\partial \mathbf{x}(t)}\right) dt \\ &= \log p_0(\mathbf{x}(0)) - \int_{0}^{t_1} \mathrm{Tr}\left(\frac{\partial f(\mathbf{x}(t), \theta, t)}{\partial \mathbf{x}(t)}\right) dt. \end{aligned} \tag{74}$$

This provides a framework where the neural network learns a vector field that smoothly deforms a simple prior distribution into the complex data distribution. This continuous-time perspective on density evolution is at the core of advanced generative models like diffusion models. In the next section, we will explore how these models are applied to transform noise back into high-fidelity data.

3.3.4 Variants of Flows

While CNFs define the transformation via the integration of a vector field, the earlier and broader family of flow-based models focuses on constructing discrete sequences of invertible mappings. The primary challenge in these models is designing the neural networks such that the Jacobian determinant is computationally efficient (ideally linear in dimensionality, $O(d)$) while retaining high expressivity. Based on how they structure the transformation and the Jacobian matrix, these models can be categorized into several variants:

- **Coupling Flows**. Pioneered by NICE [Dinh et al., 2014] and RealNVP [Dinh et al., 2017], these models partition the input dimensions into two subsets. One subset remains unchanged, while the other undergoes an affine transformation parameterized by the first subset. This construction enforces a triangular Jacobian matrix, making the determinant calculation trivial (the product of diagonal terms) and computationally efficient.

- **Autoregressive Flows**. These models treat the input dimensions as a sequence, where the transformation of the i-th dimension depends only on the previous dimensions $\{1, \ldots, i-1\}$.

This structure also yields a triangular Jacobian. Notable examples include masked autoregressive flow (MAF) [Papamakarios et al., 2017], which is efficient for density estimation, and inverse autoregressive flow (IAF) [Kingma et al., 2016], which is efficient for sampling.

- **Invertible Linear Transformations**. To ensure all dimensions can interact with each other, models like Glow [Kingma and Dhariwal, 2018] introduce learnable 1×1 convolutions. To maintain efficiency, the weight matrix is often parameterized via LU decomposition, reducing the complexity of the determinant calculation from cubic $O(d^3)$ to linear $O(d)$.
- **Residual Flows**. Inspired by residual networks, these flows define the mapping as $\mathbf{y} = \mathbf{x} + g(\mathbf{x})$. To guarantee invertibility, the Lipschitz constant of $g(\cdot)$ must be strictly bounded (typically < 1) [Behrmann et al., 2019]. Unlike coupling layers, residual flows do not have a triangular Jacobian. Instead, they often rely on stochastic estimators (such as the Russian Roulette estimator) to approximate the infinite series expansion of the log-determinant [Chen et al., 2019].

It should be noted that flow is a very general concept and has been extensively discussed in many fields, particularly in the study of dynamical systems and differential geometry. Mathematically, a flow can be formalized as a **one-parameter group of diffeomorphisms**. This definition implies that the mapping $\Phi_t(\cdot)$ satisfies the group law

$$\Phi_0(\mathbf{x}) = \mathbf{x}, \tag{75}$$

$$\Phi_{t+s}(\mathbf{x}) = \Phi_t(\Phi_s(\mathbf{x})). \tag{76}$$

This algebraic structure provides the theoretical guarantee for invertibility: the inverse of a transformation over time t is simply the transformation over time $-t$, i.e., $\Phi_t^{-1}(\cdot) = \Phi_{-t}(\cdot)$.

Furthermore, from the perspective of fluid mechanics and statistical physics, the flow of a probability density is governed by the **continuity equation** [Batchelor, 2000], a PDE that describes the conservation of mass

$$\frac{\partial p_t(\mathbf{x})}{\partial t} + \nabla \cdot (p_t(\mathbf{x}) f(\mathbf{x}, \theta, t)) = 0. \tag{77}$$

This PDE states that the rate of change of density at a point is exactly balanced by the divergence of the probability flux. The instantaneous change of variables formula used in CNFs (Eq. (73)) is, in fact, the solution to this continuity equation along the trajectories of the particles. Thus, flow-based models can be viewed not merely as neural networks with specific architectural constraints, but as computational implementations of classical transport phenomena. See Appendix B for a more detailed derivation of the instantaneous change of variables formula from the continuity equation.

4. Diffusion Models in Vision

In computer vision, diffusion models refer to a class of generative models that define a parameterized transformation from a simple prior to the data distribution. An important application is

image synthesis, where samples are generated by iteratively denoising Gaussian noise. Although diffusion models have roots in non-equilibrium thermodynamics and variational inference, they can be elegantly interpreted using differential equations. As discussed in previous sections, residual networks represent a simple Euler discretization of ODEs, whereas neural ODEs model the continuous-time dynamics directly. Diffusion models link these concepts. Conceptually, they define a continuous-time evolution (similar to neural ODEs/SDEs), yet in practice, they are often optimized and deployed using discrete time steps, akin to deep residual networks.

In this section, we begin with a problem formulation of generative modeling by considering a dual process of forward and backward transformations. Then we introduce two perspectives for modeling the generation problem: the stochastic perspective and the deterministic perspective. The former is based on **stochastic differential equations** (SDEs), which frame generation as stochastic processes of adding and removing noise. The latter is based on **probability flows**, which treat generation as a deterministic process. Finally, we discuss methods for improving the efficiency of diffusion models.

Diffusion models have been extensively discussed in many papers [Croitoru et al., 2023; Yang et al., 2023] and blogs [Song, 2021; Dieleman, 2023]. This section does not aim to cover every detail of these models. Instead, we focus on discussing them from the differential equation perspective. Occasionally, we extend our discussion a bit to ensure completeness.

4.1 Problem Statement

Prior to introducing diffusion models, we provide a formal definition of the generative modeling problem.

4.1.1 Generative Modeling as Probability Transport

In the standard setting of generative modeling, the goal is to learn the underlying probability distribution of a dataset, denoted as $p_{\text{data}}(\cdot)$, and to generate new samples from it. For instance, if we knew the true distribution of natural images, we could generate a novel image $\mathbf{x}$ simply by sampling from it. However, since the true distribution is unknown, the typical approach is to approximate it by learning from a set of observed images.

To achieve this, we usually rely on the concept of **probability transport**. Since directly modeling and sampling from the complex data distribution $p_{\text{data}}(\mathbf{x})$ is difficult, we instead introduce a latent variable $\mathbf{z}$ sampled from a tractable prior distribution $p_{\text{prior}}(\mathbf{z})$, such as a standard Gaussian $\mathcal{N}(\mathbf{0}, \mathbf{I})$. The generation task is then reduced to finding a deterministic or stochastic mapping $\Phi : \mathbb{R}^d \to \mathbb{R}^d$ that transforms the latent variable $\mathbf{z}$ into a data sample $\mathbf{x} \approx \Phi(\mathbf{z})$.

From the perspective of differential equations, this mapping is not instantaneous but is induced by a continuous-time dynamic process. In this process, $\mathbf{x}$ and $\mathbf{z}$ are viewed as the states of a system at the start and end of a trajectory. While neural ODEs (discussed in Section 3) generally model dynamics over an arbitrary interval $[t_0, t_1]$, diffusion models specifically formulate this transport as a continuous-time evolution over a fixed interval $t \in [0, T]$. So we follow the convention that the process starts from the data distribution at $t = 0$ and evolves to the prior distribution at $t = T$, or vice versa.

Formally, we can view this dynamic evolution as a flow. Let $\Phi_{0\to t} : \mathbb{R}^d \to \mathbb{R}^d$ denote the flow map that transports a sample from its state at time 0 to its state at time t. While this map operates on individual samples, it induces a corresponding transformation on the probability distribution itself. This relationship is mathematically described by the **pushforward** operation. Specifically, if we start with the data distribution $p_0(\mathbf{x})$, the distribution $p_t(\mathbf{x})$ at any intermediate time t is the pushforward of p_0 by the flow map, denoted as

$$p_t = (\Phi_{0\to t})_{\#} p_0. \tag{78}$$

This equality implies that probability mass is conserved during the transport: if a random variable $\mathbf{x}$ follows p_0, then the transformed variable $\Phi_{0\to t}(\mathbf{x})$ follows p_t. In general, the flow is assumed to be a diffeomorphism, and thus this operation is fully invertible. Then, we can define the generation process as the reverse pushforward $p_0 = (\Phi_{T\to 0})_{\#} p_T$, which transports the simple prior p_T back to the complex data distribution p_0.

To understand the pushforward operation and the evolution of probability distributions, consider the analogy of dropping colored dye into flowing water. In this example, the probability distribution corresponds to the concentration of the dye, and the data samples correspond to individual dye molecules. The flow map $\Phi_{0\to t}$ can be thought of as the current of the water, which determines the precise trajectory of every molecule over time. As these molecules follow the current, the overall shape of the dye cloud changes. The pushforward operation is the mathematical tool that helps us determine the density of the dye at any specific location and time, based on the movements of the molecules. See Figure 11 for an illustration.

Note that, although the local density $p_t(\mathbf{x})$ changes as the fluid expands or contracts, the total amount of dye remains invariant. This means that the integral of the probability density over the entire space is always equal to 1. Thus, by defining the flow of the "particles" (samples), we implicitly and uniquely define the evolution of the "cloud" (distribution).

Now, we have our task of generative modeling: we learn a reverse flow $\Phi_{T\to 0}$, as well as the corresponding pushforward operation $(\Phi_{T\to 0})_{\#} p_T$, such that we can pushforward samples from the prior noise to the data distribution. The final output is a sample drawn from this data distribution. As $\Phi_{T\to 0}$ is a flow, we can model it using ODEs or related tools in differential equations. In the rest of this section, we will see several methods for modeling probability transport, which can be well interpreted using differential equations.

Although a flow can be seen as a single, continuous-time evolution of system states, in practice, we typically discretize it into a number of simpler steps to facilitate modeling and learning. Instead of attempting to learn a monolithic mapping that directly transports samples between the noise and data distributions, we decompose the process into a sequence of incremental transitions. Specifically, let N be the number of discretization steps and $0 = t_0 < t_1 < \cdots < t_N = T$ be the time points. We represent the trajectory from $\mathbf{z}$ to $\mathbf{x}$ as $\mathbf{z} = \mathbf{x}(t_N) \to \mathbf{x}(t_{N-1}) \to \cdots \to \mathbf{x}(t_1) \to \mathbf{x}(t_0) = \mathbf{x}$. In image generation, this means that we start with a purely noisy image, and remove the noise step by step to generate a realistic image. At any specific step k, the transformation required to map the distribution p_{t_k} to the subsequent distribution $p_{t_{k-1}}$ is

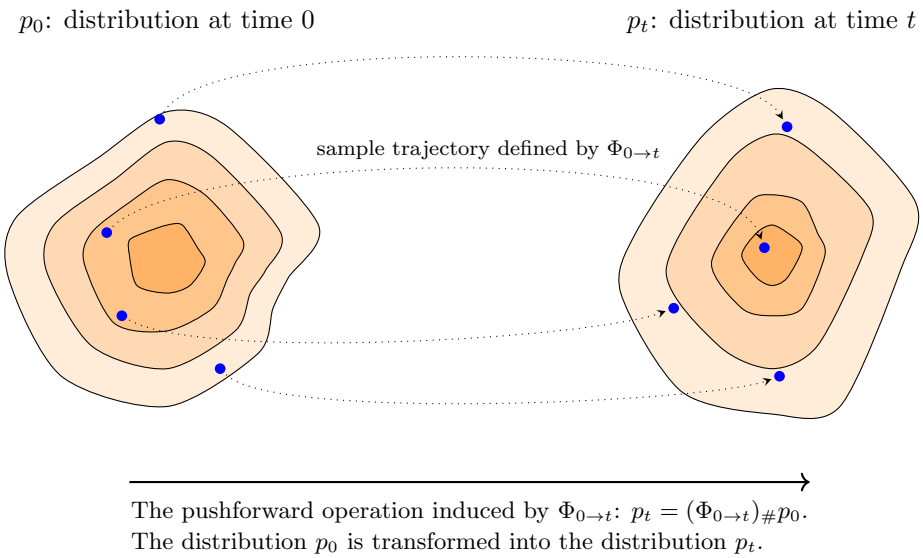


Figure 11: Illustration of the pushforward operation. The distributions at time 0 and time t are visualized as the evolution of dye density in a fluid. The map $\Phi_{0\to t}$ describes the trajectories of individual dye molecules over time. As these molecules move to new positions, the local density varies, yet the total mass remains conserved. The pushforward operation formally defines this transformation, mapping the initial density distribution p_0 to the resulting distribution p_t based on the collective trajectories defined by Φ.

simple. By decomposing the global transport problem into these local transitions, the model is only required to learn the direction of the flow (i.e., the vector field) at each step.

4.1.2 The Duality of Forward and Backward Processes

As discussed above, central to the design of diffusion models is the division of the task into two processes: a process that destroys data and a process that creates it. We refer to these as the **forward process** and the **backward process**, respectively.

The forward process defines the transformation from the data distribution to the prior distribution as time evolves from $t = 0$ to T. This process can be formulated either stochastically or deterministically. This leads to two different perspectives in designing diffusion models, as will be presented later in this section. As a simple example, here we describe the forward process using deterministic differential equations via the perspective of ODEs. In this view, we model the sample trajectories via an ODE

$$\frac{d\mathbf{x}(t)}{dt} = f(\mathbf{x}(t), t), \tag{79}$$

where $f(\mathbf{x}(t), t)$ is a vector field which is typically pre-defined (e.g., Gaussian noise injection or linear interpolation). Given the flow defined by this ODE, we can transform the distribution of the samples from time 0 to any time t using the pushforward operation (see Eq. (78)).

The backward process is the reverse of the forward process. If we take the ODE view, it corresponds to the time-reversal of the dynamics, evolving from $t = T$ back to 0. Given the forward ODE above, the generative trajectories are defined by the reverse-time ODE

$$\frac{d\mathbf{x}(t)}{dt} = -f(\mathbf{x}(t), t), \tag{80}$$

which is solved backwards from T to 0. Note that the ideal backward vector field is simply the negation of the forward field, which we can approximate using a neural network.

This bi-directional framework naturally aligns with the architecture of autoencoders [Kingma and Welling, 2019; Bank et al., 2023], as illustrated in Figure 12. The forward process corresponds to the encoder, which maps the observed data $\mathbf{x}$ (i.e., $\mathbf{x}(0)$) to the latent variable $\mathbf{z}$ (i.e., $\mathbf{x}(T)$) through a sequence of intermediate states $\{\mathbf{x}(t)\}$. The backward process corresponds to the decoder, which reconstructs the data $\mathbf{x}$ from the latent variable $\mathbf{z}$ through the reverse sequence.

However, there are some differences between diffusion models and traditional autoencoders:

- First, regarding dimensionality, standard autoencoders typically perform dimensionality reduction: the dimension of the latent variable $\mathbf{z}$ is smaller than that of the input $\mathbf{x}$. In contrast, diffusion models generally maintain the same dimensionality throughout the process. The input data $\mathbf{x}$, the latent variable $\mathbf{z}$, and all intermediate states share the same dimension $\mathbb{R}^d$. This aligns with the task objective where we transform samples rather than changing their dimensionality.

- Second, regarding the functional relationship between the forward and backward paths, the general autoencoder framework does not require the encoder and decoder to share the same model architecture or parameters. They can be parameterized by distinct neural networks that are optimized jointly but operate as separate functions. In contrast, in the context of probability flow ODEs for diffusion models, the forward and backward processes are intrinsically coupled. As shown in Eqs. (79) and (80), the backward process is governed by the negation of the forward vector field. This implies a mathematical symmetry: defining the dynamics of the forward evolution implicitly defines the dynamics of the backward generation process.

Despite these differences, the theoretical foundations of autoencoders provide valuable insights for training diffusion models [Luo, 2022; Kingma and Gao, 2023]. For example, the **evidence lower bound** (ELBO), which is a commonly used loss function in **variational autoencoders** (VAEs), is closely related to the objective used in training diffusion models[7]. In this section, we will not discuss the details of autoencoders, but will occasionally use them as a tool for interpreting diffusion models.

7. By maximizing the ELBO [Kingma and Welling, 2013], we minimize the divergence (typically the Kullback-Leibler divergence) between the forward and backward transition distributions at every time step. This formulation links the physical intuition of "reversing the flow" with the statistical objective of "maximizing data likelihood". As a result, the problem of generative modeling is reduced to a regression task: the model learns to estimate the vector field that generated the current state, thereby learning the reversal of the process.

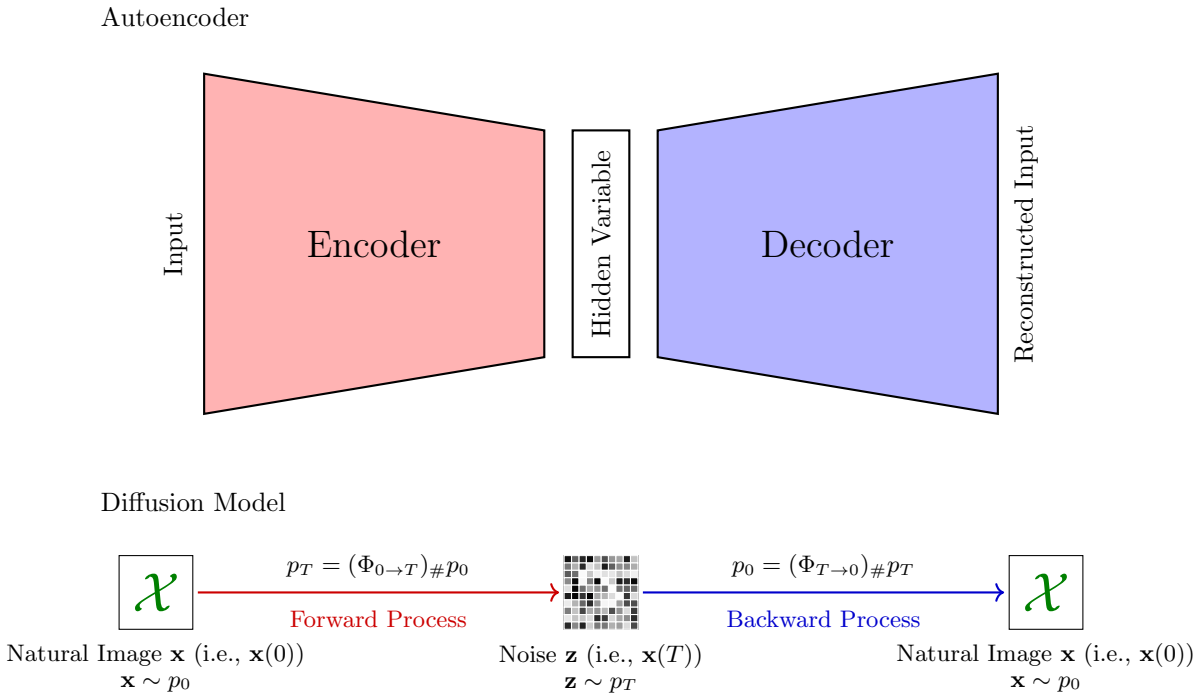


Figure 12: Comparison between autoencoders and diffusion models. The top panel depicts a standard autoencoder architecture, which maps inputs to a lower-dimensional hidden variable space via an encoder and reconstructs them via a decoder. The bottom panel depicts a diffusion model. In contrast to autoencoders, which rely on a bottleneck architecture to compress data, diffusion models define an invertible mapping between the data distribution and a noise distribution. The forward process pushes the data distribution p_0 to a simple noise prior p_T, while the backward process maps the distribution back to p_0. Here, $\mathbf{x}$ represents a natural image drawn from p_0, and $\mathbf{z}$ represents noise drawn from p_T. $(\Phi_{0\to T})_\#$ and $(\Phi_{T\to 0})_\#$ denote the pushforward operations for the forward and backward processes, respectively.

4.1.3 Training and Inference

While the forward and backward processes are theoretically symmetric, they play distinct roles. In the inference phase, our goal is to generate new samples. Thus, we only execute the backward process. We initialize the system with a latent variable $\mathbf{z}$ (i.e., $\mathbf{x}(T)$) sampled from the prior distribution, and apply the learned reverse dynamics to iteratively map the noise back to a data sample $\mathbf{x}$ (i.e., $\mathbf{x}(0)$).

A natural question arises: if the actual application only involves the backward process, why is it necessary to explicitly define the forward process? To answer this, consider the analogy of traversing a maze: if we have walked from the entrance to the exit, finding the way back is difficult without observing valid paths. The forward process serves as the guide that constructs these training paths. By transforming a data sample $\mathbf{x}$ into noise $\mathbf{z}$ through a sequence of intermediate states, the forward process provides the necessary supervision signals. It tells the model, for

any given time t, what the correct velocity (or direction) should be to return towards the data distribution.

Therefore, the training of diffusion models typically involves two steps. First, we execute the forward process to generate the trajectories from data to noise, thereby creating the ground truth paths. Then, we optimize the parameters of the backward process to accurately match this ground truth flow.

4.2 The Stochastic Perspective: SDEs

While we have seen that ODEs provide a deterministic view of generative modeling, it is important to recognize that the underlying mechanism of diffusion is inherently random. From this perspective, the forward process is modeled not as a deterministic flow, but as a stochastic process that gradually corrupts the input sample with noise. Indeed, the canonical formulation of these models relies on stochastic differential equations (SDEs) to describe the dynamics of data corruption and reconstruction [Oksendal, 2013]. In what follows, we introduce SDE-based diffusion models.

4.2.1 Forward and Reverse SDEs

An SDE is a differential equation in which one or more terms are stochastic processes. Typically, it describes the evolution of a system that is subject to both deterministic forces (predictable trends) and random noise (unpredictable fluctuations).

To understand this, let us first recall the standard ODE describing the evolution of a state vector $\mathbf{x}(t)$ over time t: $\frac{d\mathbf{x}(t)}{dt} = f(\mathbf{x}(t), t)$. Here, $f(\cdot)$ is a deterministic vector field that describes the velocity of the state. Given an initial condition $\mathbf{x}(0)$, the future trajectory is entirely determined.

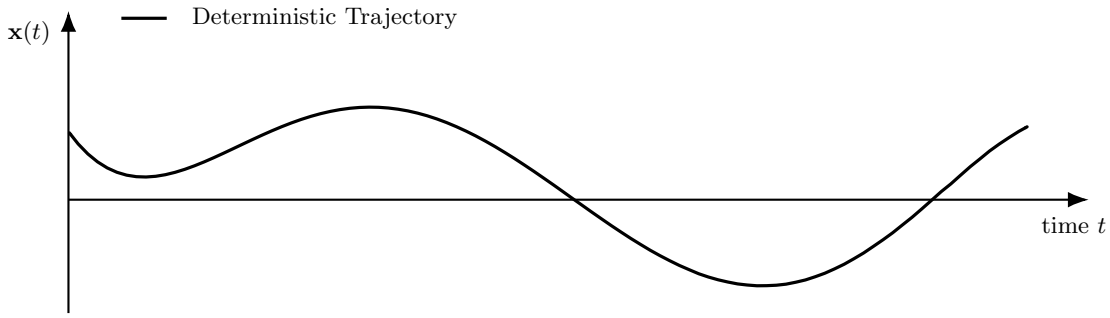
However, in diffusion models, we introduce random noise to the data continuously. To formulate this, we must add a stochastic term to the equation. Since standard Brownian motion is nowhere differentiable, we cannot simply write $\frac{d\mathbf{x}(t)}{dt} = f + \text{noise}$. Instead, we write the equation in differential form involving a random component

$$d\mathbf{x}(t) \quad = \quad \underbrace{f(\mathbf{x}(t), t)dt}_{\text{deterministic}} + \underbrace{g(t)d\mathbf{w}(t)}_{\text{stochastic}}. \tag{81}$$

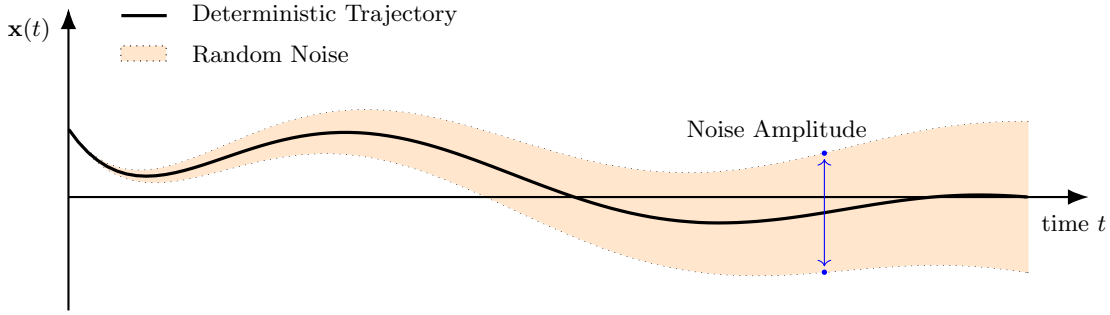
This is a generic Itô SDE[8]. The components of this equation are defined as follows

- The drift coefficient $f(\mathbf{x}(t), t) \in \mathbb{R}^d$ represents the deterministic part of the evolution, similar to the vector field in an ODE. It determines the general trend of the evolution.
- The diffusion coefficient $g(t)$ is typically a scalar function that controls the magnitude of the stochastic noise injected into the system at time t.

8. Stochastic calculus requires a specific interpretation of the integral $\int g(t)d\mathbf{w}(t)$. The **Itô interpretation** evaluates the integrand at the beginning of each time interval (i.e., the left endpoint) during discretization [Karatzas and Shreve, 2014]. This choice is standard in diffusion models because it ensures that the noise increment $d\mathbf{w}$ is independent of the current state, thereby preserving the martingale property and simplifying the computation of expectations.



(a) ODE (deterministic trajectory)



(b) SDE (with random noise)

Figure 13: Deterministic vs. stochastic trajectories. For comparison, subfigure (a) shows a deterministic trajectory governed by an ODE. Subfigure (b) shows the trajectory of the SDE defined by $d\mathbf{x}(t) = f(\mathbf{x}(t), t)dt + g(t)d\mathbf{w}(t)$. The term $f(\mathbf{x}(t), t)dt$ represents the deterministic drift (analogous to the ODE), and $g(t)d\mathbf{w}(t)$ introduces Gaussian noise. In this example, the drift coefficient $f(\mathbf{x}(t), t)$ gradually decays to zero, whereas the diffusion coefficient $g(t)$ increases over time. At the terminal time T, we have $f(\mathbf{x}(T), T) = 0$ and $g(T) = 1$, and the state converges to a standard Gaussian distribution. This behavior is analogous to the forward process of a diffusion model: as time progresses, more noise is injected while the deterministic signal diminishes.

- The Wiener process (also called Brownian motion) $\mathbf{w}(t) \in \mathbb{R}^d$ represents the source of randomness. The increment $d\mathbf{w}(t)$ can be thought of as an infinitesimal Gaussian noise vector with mean $\mathbf{0}$ and covariance $dt \cdot \mathbf{I}$.

Eq. (81) is called the forward SDE. By using this SDE, we define a forward process that gradually adds noise to a data sample $\mathbf{x}(0) \sim p_{\text{data}}$ as time t flows from 0 to T. During this process, the distribution $p_t(\mathbf{x}(t))$ diffuses, and the distribution of $\mathbf{x}(T)$ will eventually converge to a known prior distribution (usually $\mathcal{N}(\mathbf{0}, \mathbf{I})$), as illustrated in Figure 13.

The remarkable property of SDEs, derived by Anderson [1982], is that for any forward SDE, there exists a corresponding reverse-time SDE. If we simulate the process backwards in time from $t = T$ to 0, we can generate samples from the data distribution. The reverse SDE is given by

$$d\mathbf{x}(t) \quad = \quad \left[f(\mathbf{x}(t), t) - g(t)^2 \nabla_{\mathbf{x}(t)} \log p_t(\mathbf{x}(t))\right] dt + g(t) d\bar{\mathbf{w}}(t), \tag{82}$$

where dt represents an infinitesimal negative time increment, $d\bar{\mathbf{w}}(t)$ is a standard Wiener process in the reverse time, and $\nabla_{\mathbf{x}(t)} \log p_t(\mathbf{x}(t))$ is the gradient of the log-probability density with respect to the data, also called the **score function**. Although this equation appears a bit complicated, its application is straightforward. Given a forward SDE (Eq. (81)), if we can access the score function, we can use the reverse SDE (Eq. (82)) to transport samples from the prior noise p_T back to the data distribution p_0.

There are two fundamental issues when applying the reverse SDE to generative modeling.

- **How to obtain the score function in the reverse SDE?** The backward process depends on the gradient of the log-density $\nabla_{\mathbf{x}(t)} \log p_t(\mathbf{x}(t))$. However, the intermediate marginal distribution $p_t(\mathbf{x}(t))$ is obtained by marginalizing over all possible data samples and diffusion paths, making it analytically intractable. We only have access to samples from the data distribution p_0, not an explicit expression for p_t.
- **How to define the forward SDE?** We need to specify the drift coefficient $f(\mathbf{x}(t), t)$ and the diffusion coefficient $g(t)$. These functions determine the trajectory of the diffusion process. They must be carefully designed so that the complex data distribution p_0 is effectively transformed into a simple prior distribution p_T at the end of the process.

In what follows, we address these issues by introducing the **score matching** technique and discussing SDE formulations commonly used in the literature.

4.2.2 Learning the Score Function

As mentioned above, since the marginal distribution $p_t(\mathbf{x}(t))$ is generally intractable, we cannot compute the gradient $\nabla_{\mathbf{x}(t)} \log p_t(\mathbf{x}(t))$ directly. Instead, we resort to score matching [Hyvärinen and Dayan, 2005], in particular the **denoising score matching** technique [Vincent, 2011].

The core idea is that while the score of the marginal distribution is unknown, the score of the conditional transition distribution $p(\mathbf{x}(t)|\mathbf{x}(0))$ is often easy to compute. Specifically, consider a transition kernel where Gaussian noise $\boldsymbol{\epsilon}$ is added to a data point $\mathbf{x}(0)$ such that $\mathbf{x}(t) \sim \mathcal{N}(\mathbf{x}(0), \sigma(t)^2 \mathbf{I})$, that is,

$$\mathbf{x}(t) \quad = \quad \mathbf{x}(0) + \boldsymbol{\epsilon}. \tag{83}$$

The score of this conditional distribution is given by

$$\begin{aligned} \nabla_{\mathbf{x}(t)} \log p(\mathbf{x}(t)|\mathbf{x}(0)) \quad &= \quad -\frac{\mathbf{x}(t) - \mathbf{x}(0)}{\sigma(t)^2} \\ &= \quad -\frac{\boldsymbol{\epsilon}}{\sigma(t)^2}. \end{aligned} \tag{84}$$

Vincent [2011] proved that training a neural network $s_\theta(\mathbf{x}(t), t)$ to approximate this conditional score (which is proportional to the added noise $\boldsymbol{\epsilon}$) is equivalent to learning the marginal score $\nabla_{\mathbf{x}(t)} \log p_t(\mathbf{x}(t))$. Thus, the objective of generative modeling can be effectively reduced to a denoising regression problem.

Formally, this intuition leads to the following objective function. We seek to minimize the expected squared Euclidean distance between the estimated score and the true conditional score:

$$\mathcal{L}(\theta) \quad = \quad \mathbb{E}_{t,\mathbf{x}(0),\boldsymbol{\epsilon}} \left[\lambda(t) \left\| s_\theta(\mathbf{x}(t), t) - \left(-\frac{\boldsymbol{\epsilon}}{\sigma(t)^2} \right) \right\|^2 \right], \tag{85}$$

where $\lambda(t)$ is a positive weighting function. A common choice is $\lambda(t) = \sigma(t)^2$, which balances the magnitude of the score across different noise levels.

By substituting this weighting and rearranging the terms, the objective simplifies to

$$\mathcal{L}(\theta) \quad = \quad \mathbb{E}_{t,\mathbf{x}(0),\boldsymbol{\epsilon}} \left[\frac{1}{\sigma(t)^2} \left\| \boldsymbol{\epsilon} + \sigma(t)^2 s_\theta(\mathbf{x}(0) + \boldsymbol{\epsilon}, t) \right\|^2 \right]. \tag{86}$$

This formulation highlights that optimizing the score model is mathematically equivalent to training a network to predict the noise $\boldsymbol{\epsilon}$ added to the sample. In practice, the weighting factor $1/\sigma(t)^2$ is often discarded to prioritize perceptual quality, which leads to a simple noise prediction objective.

It is worth noting that the above approach shares some theoretical connections with **energy-based models** (EBMs) [LeCun et al., 2006]. In statistical physics, a probability distribution is often expressed via an energy function $E_\theta(\mathbf{x})$ as $p_\theta(\mathbf{x}) = \frac{\exp(-E_\theta(\mathbf{x}))}{Z_\theta}$, where Z_θ is the normalization constant or partition function.

An advantage of modeling the score function, rather than the likelihood directly, is that it can bypass the computation of the intractable normalization constant Z_θ. For a model to be normalized, the integral of the density over the entire space must equal 1. For example, language models use a Softmax function to normalize logits, which requires a summation over the vocabulary. However, for high-dimensional continuous data like images, calculating $Z_\theta = \int \exp(-E_\theta(\mathbf{x}))d\mathbf{x}$ is computationally intractable. Score-based models avoid this issue because the gradient of the log-partition function with respect to the data is zero, i.e., $\nabla_{\mathbf{x}} \log Z_\theta = 0$. Thus, we have

$$\begin{aligned} \nabla_{\mathbf{x}} \log p_\theta(\mathbf{x}) \quad &= \quad \nabla_{\mathbf{x}} \left(-E_\theta(\mathbf{x}) - \log Z_\theta \right) \\ &= \quad -\nabla_{\mathbf{x}} E_\theta(\mathbf{x}). \end{aligned} \tag{87}$$

This property allows us to train unnormalized models without ever evaluating the partition function.

Once the score network $s_\theta(\mathbf{x}, t)$ is trained, we can use it to generate new samples. Specifically, we substitute our learned approximation $s_\theta(\mathbf{x}(t), t) = \nabla_{\mathbf{x}(t)} \log p_t(\mathbf{x}(t))$ into Eq. (82), and obtain

$$d\mathbf{x}(t) = \left[f(\mathbf{x}(t), t) - g(t)^2 s_\theta(\mathbf{x}(t), t) \right] dt + g(t) d\bar{\mathbf{w}}(t). \tag{88}$$

At test time, we initialize $\mathbf{x}(T)$ from the prior distribution (e.g., $\mathcal{N}(\mathbf{0}, \mathbf{I})$) and integrate this equation backwards from $t = T$ to $t = 0$ using numerical solvers such as the Euler-Maruyama method [Kloeden and Pearson, 1977]. By gradually removing the noise, this procedure transforms the initial noisy sample into a sample from the target data distribution.

4.2.3 The VE and VP SDEs

We now turn to the problem of defining the forward SDE, that is, we define the drift coefficient $f(\mathbf{x}(t), t)$ and the diffusion coefficient $g(t)$ in Eq. (81). There are many ways to construct a forward SDE. Here we consider two commonly used methods. They correspond to the continuous-time limits of two discrete-time models: **noise conditional score network** (NCSN) [Song and Ermon, 2019] and **denoising diffusion probabilistic models** (DDPM) [Ho et al., 2020b]. Song et al. [2021] unified these methods under the SDE framework, and termed them **variance exploding** (VE) and **variance preserving** (VP) SDEs.

In the NCSN framework, the forward process perturbs the data with a sequence of increasing Gaussian noise levels $\{\sigma_1, \sigma_2, \ldots, \sigma_N\}$, where $\sigma_1 < \sigma_2 < \cdots < \sigma_N$. Consider a discrete process where the state at step i, denoted by $\mathbf{x}_i$, evolves from $\mathbf{x}_{i-1}$ by adding the necessary amount of noise to increase the total variance from σ_{i-1}^2 to σ_i^2. Then, the update rule is given by

$$\mathbf{x}_i = \mathbf{x}_{i-1} + \sqrt{\sigma_i^2 - \sigma_{i-1}^2}\boldsymbol{\epsilon}_i, \tag{89}$$

where $\boldsymbol{\epsilon}_i \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ is the incremental noise at step i. This model ensures that if we start with data $\mathbf{x}_0$, the distribution at step i has a variance of σ_i^2[9].

Now, let us move to the continuous limit. We define a continuous variance function $\sigma(t)$ with respect to time t. As the number of steps approaches infinity ($N \to \infty$), the discrete time interval approaches zero ($\Delta t \to 0$). The difference in variance between adjacent steps becomes a differential

$$\sigma_i^2 - \sigma_{i-1}^2 \approx \frac{d\sigma^2(t)}{dt}\Delta t. \tag{90}$$

Recall that in stochastic calculus, the increment of a Wiener process $d\mathbf{w}$ scales with $\sqrt{\Delta t}$ (i.e., $\sqrt{\Delta t}\boldsymbol{\epsilon}_i \approx d\mathbf{w}(t)$). We can rewrite the update rule by substituting the variance difference

$$\mathbf{x}_i - \mathbf{x}_{i-1} \approx \sqrt{\frac{d\sigma^2(t)}{dt}\Delta t}\,\boldsymbol{\epsilon}_i. \tag{91}$$

Taking the limit $\Delta t \to 0$, we obtain the continuous-time SDE

$$d\mathbf{x}(t) = \sqrt{\frac{d\sigma^2(t)}{dt}}d\mathbf{w}(t). \tag{92}$$

In this SDE, the drift coefficient is defined as $f(\mathbf{x}(t), t) = \mathbf{0}$, and the diffusion coefficient is defined as $g(t) = \sqrt{\frac{d\sigma^2(t)}{dt}}$. Since the variance of the distribution increases monotonically as $t \to T$,

9. More precisely, since the initial state $\mathbf{x}_0$ is given, we should say that the conditional distribution given $\mathbf{x}_0$ has a variance of σ_i^2

this SDE maps the data distribution to a noise distribution with diverging variance. Hence, it is referred to as the variance exploding SDE.

Alternatively, the DDPM framework adopts a different noise injection strategy. Instead of simply adding noise with increasing variance, it rescales the data at each step to keep the variance bounded. In the discrete formulation, the forward process is defined by a variance schedule $0 < \beta_1 < \beta_2 < \cdots < \beta_N < 1$. The transition from $\mathbf{x}_{i-1}$ to $\mathbf{x}_i$ is given by

$$\mathbf{x}_i = \sqrt{1-\beta_i}\mathbf{x}_{i-1} + \sqrt{\beta_i}\boldsymbol{\epsilon}_i, \tag{93}$$

where $\boldsymbol{\epsilon}_i \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. Unlike the NCSN formulation, the coefficient $\sqrt{1-\beta_i}$ shrinks the mean of the previous state. This design ensures that if the input data follows a standard normal distribution, the marginal distribution remains a standard normal distribution at every step (i.e., the variance is preserved).

Then, the increment $\mathbf{x}_i - \mathbf{x}_{i-1}$ can be expressed as

$$\mathbf{x}_i - \mathbf{x}_{i-1} \quad = \quad (\sqrt{1-\beta_i} - 1)\mathbf{x}_{i-1} + \sqrt{\beta_i}\boldsymbol{\epsilon}_i. \tag{94}$$

To obtain the continuous-time SDE, we again let $N \to \infty$ and introduce a continuous function $\beta(t)$ such that $\beta_i \approx \beta(t)\Delta t$. Using the Taylor expansion approximation $\sqrt{1-x} \approx 1 - \frac{1}{2}x$ for small x, we have $\sqrt{1-\beta(t)\Delta t} - 1 \approx -\frac{1}{2}\beta(t)\Delta t$. Substituting this back into Eq. (94) and applying the scaling rule $\sqrt{\beta(t)\Delta t}\boldsymbol{\epsilon}_i \approx \sqrt{\beta(t)}d\mathbf{w}(t)$, we obtain the limit

$$d\mathbf{x}(t) \quad = \quad -\frac{1}{2}\beta(t)\mathbf{x}(t)dt + \sqrt{\beta(t)}d\mathbf{w}(t). \tag{95}$$

In this SDE, the drift coefficient is $f(\mathbf{x}(t), t) = -\frac{1}{2}\beta(t)\mathbf{x}(t)$, and the diffusion coefficient is $g(t) = \sqrt{\beta(t)}$. The drift term serves as a restoring force that pulls the state towards the origin (zero mean), and so counteracts the variance injected by the diffusion term. As a result, if the initial distribution has unit variance, the variance remains fixed at unity throughout the process. For this reason, Eq. (95) is called the variance preserving SDE.

Here, for brevity, we do not explicitly define the functional forms of the parameters $\{\sigma_i\}$ (or $\sigma(t)$) and $\{\beta_i\}$ (or $\beta(t)$). It is worth noting that in DDPMs, auxiliary variables (such as $\alpha_i = 1-\beta_i$ and $\bar{\alpha}_i = \prod_{s=1}^{i} \alpha_s$) are often introduced to simplify the derivation and implementation. The careful design of the scheduling parameters can make the resulting model theoretically sound. We refer interested readers to the original papers for detailed discussions on the selection of these scheduling parameters [Song and Ermon, 2019; Ho et al., 2020b].

While we focus on the continuous-time SDE formulation in this subsection to establish a more unified theoretical foundation, it is important to note that in practice, diffusion models are generally implemented via discretization steps (see Table 3). The discrete variables $\mathbf{x}_i$ introduced in the NCSN and DDPM methods essentially serve as numerical approximations of the continuous process. In actual implementation, we simulate the evolution of the SDE by iterating through these discrete time steps $\{\mathbf{x}_i\}$. This approach reflects a common paradigm in applying differential equations to deep learning: we use continuous differential equations for their theoretical com-

Entry	Variance Exploding (NCSN)	Variance Preserving (DDPM)
Drift coefficient $f(\mathbf{x}(t), t)$	$\mathbf{0}$	$-\frac{1}{2}\beta(t)\mathbf{x}(t)$
Diffusion coefficient $g(t)$	$\sqrt{\frac{d\sigma^2(t)}{dt}}$	$\sqrt{\beta(t)}$
Continuous-time SDE	$d\mathbf{x}(t) = \sqrt{\frac{d\sigma^2(t)}{dt}}d\mathbf{w}(t)$	$d\mathbf{x}(t) = -\frac{1}{2}\beta(t)\mathbf{x}(t)dt + \sqrt{\beta(t)}d\mathbf{w}(t)$
Iterative diffusion (discrete)	$\mathbf{x}_i = \mathbf{x}_{i-1} + \sqrt{\sigma_i^2 - \sigma_{i-1}^2}\boldsymbol{\epsilon}_i$	$\mathbf{x}_i = \sqrt{1-\beta_i}\mathbf{x}_{i-1} + \sqrt{\beta_i}\boldsymbol{\epsilon}_i$
Single-step diffusion (discrete)	$\mathbf{x}_i = \mathbf{x}_0 + \sigma_i\boldsymbol{\epsilon}$	$\mathbf{x}_i = \sqrt{\bar{\alpha}_i}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_i}\boldsymbol{\epsilon}$

Table 3: Continuous-time and discrete-time forms of the VP and VE SDEs [Song et al., 2021]. $\mathbf{x}(t)$, $\mathbf{w}(t)$, $\sigma(t)$, and $\beta(t)$ are continuous functions of time t. $\mathbf{x}_i$ is the state at discrete time step i, and β_i, σ_i, and $\bar{\alpha}_i$ are corresponding parameters. Here β_i and σ_i are specified in advance, and $\bar{\alpha}_i$ is defined as $\prod_{s=1}^{i}(1-\beta_s)$, which determines the relative weight of the signal $\mathbf{x}_0$ in the single-step diffusion process. Both $\boldsymbol{\epsilon}_i$ and $\boldsymbol{\epsilon}$ are standard Gaussian noise. The former represents the incremental noise at step i, and the latter represents the total noise perturbing $\mathbf{x}_0$ directly to $\mathbf{x}_i$.

pleteness and expressive power, while relying on discrete numerical approximations for efficient training and inference.

4.2.4 Numerical Solvers

To generate samples from a trained diffusion model, we need to simulate the reverse-time SDE (Eq. (82)) starting from a sample $\mathbf{x}(T) \sim p_T$ and evolving backwards to $t = 0$. Since analytical solutions are rarely available for complex data distributions, we resort to numerical solvers to approximate the continuous trajectory.

The simplest and most common approach is the Euler-Maruyama method, which generalizes the Euler method to SDEs [Kloeden and Platen, 1992]. We first discretize the time interval $[0, T]$ into N steps: $t_N = T > t_{N-1} > \cdots > t_0 = 0$. For a small time step $\Delta t = t_{i+1} - t_i$, the reverse SDE can be approximated by

$$\mathbf{x}_{t_i} \approx \mathbf{x}_{t_{i+1}} - \left[f(\mathbf{x}_{t_{i+1}}, t_{i+1}) - g(t_{i+1})^2 s_\theta(\mathbf{x}_{t_{i+1}}, t_{i+1})\right]\Delta t + g(t_{i+1})\sqrt{\Delta t}\,\boldsymbol{\epsilon}_{i+1}, \tag{96}$$

Note that because we are integrating backwards in time, the computation step in the numerical solver corresponds to a negative time increment in the physical time t. This formulation recovers the sampling steps used in the original NCSN and DDPM papers, and provides a theoretical justification for their heuristic sampling algorithms[10].

10. For instance, applying the Euler-Maruyama discretization to the reverse VP SDE (from t_{i+1} to t_i) yields $\mathbf{x}_{t_i} \approx (1 + \frac{1}{2}\beta_{i+1})\mathbf{x}_{t_{i+1}} + \beta_{i+1}s_\theta(\mathbf{x}_{t_{i+1}}, t_{i+1}) + \sqrt{\beta_{i+1}}\boldsymbol{\epsilon}_{i+1}$. By substituting the score parameterization $s_\theta(\mathbf{x}, t) \approx -\boldsymbol{\epsilon}_\theta(\mathbf{x}, t)/\sqrt{1-\bar{\alpha}_t}$ (where indices correspond to t_{i+1}) and recognizing that $1 + \frac{1}{2}\beta_{i+1}$ corresponds to the first-order Taylor expansion of the scaling term $(1-\beta_{i+1})^{-1/2}$ used in DDPM, the derived update rule becomes identical to the standard DDPM sampling algorithm.

An advantage of the SDE framework is the ability to combine numerical integration with **Markov Chain Monte Carlo** (MCMC) methods [Robert et al., 1999]. For instance, by using the predictor-corrector method, the numerical discretization can be decoupled from the score-based correction. In the predictor step, a numerical solver (such as Euler-Maruyama or a higher-order solver) estimates the state at the next time step $\mathbf{x}_{t_i}$ based on the current state $\mathbf{x}_{t_{i+1}}$. This step evolves the process backward in time. After the predictor step, an MCMC method (such as Langevin dynamics) is applied to $\mathbf{x}_{t_i}$ for several iterations. Since the score function $s_\theta(\mathbf{x}_{t_i}, t_i)$ defines the distribution p_{t_i}, running Langevin dynamics moves the predicted sample towards the high-density regions of the marginal distribution p_{t_i} without changing the time t_i.

4.3 The Deterministic Perspective: Probability Flow ODEs

An alternative perspective on generative modeling frames the generation task as a probability flow defined by ODEs. Unlike the stochastic perspective, this approach provides a deterministic formulation of probability transport, which enables a bidirectional mapping between the data distribution and the prior distribution through continuous-time ODE trajectories. In this subsection, we first demonstrate the theoretical connection between diffusion SDEs and these deterministic flows. We then discuss methods for learning these deterministic flows directly, such as **flow matching**.

4.3.1 From SDEs to ODEs

An interesting theoretical result in score-based generative modeling is that the stochastic process described by the SDE has an equivalent deterministic representation. Song et al. [2021] demonstrated that for any diffusion SDE, there exists an associated deterministic ODE whose trajectories induce the exact same marginal probability densities $\{p_t(\mathbf{x}(t))\}_{t=0}^T$ as the SDE.

This ODE is referred to as the probability flow ODE, given by

$$\frac{d\mathbf{x}(t)}{dt} = f(\mathbf{x}(t), t) - \frac{1}{2} g(t)^2 \nabla_{\mathbf{x}(t)} \log p_t(\mathbf{x}(t)). \tag{97}$$

The derivation relies on the Fokker-Planck equation (also known as the Kolmogorov forward equation), which describes the time evolution of the probability density function $p_t(\mathbf{x})$ under the dynamics of an SDE. It can be shown that the continuity equation induced by the ODE in Eq. (97) is identical to the Fokker-Planck equation of the SDE in Eq. (81) (see Appendix C for a more detailed derivation). As a result, if we sample $\mathbf{x}(T) \sim p_T$ and solve this ODE backwards in time (from T to 0), the resulting sample $\mathbf{x}(0)$ will be distributed according to the data distribution p_0, identical to the result of the reverse SDE.

This equivalence provides an important link between the stochastic and deterministic perspectives. Note that Eq. (97) depends on the score function $\nabla_{\mathbf{x}(t)} \log p_t(\mathbf{x}(t))$. This means that once we have trained a score model $s_\theta(\mathbf{x}(t), t)$, we can immediately use it to construct the probability flow ODE

$$\frac{d\mathbf{x}(t)}{dt} \approx f(\mathbf{x}(t), t) - \frac{1}{2} g(t)^2 s_\theta(\mathbf{x}(t), t). \tag{98}$$

In practice, using the ODE approach for sampling offers several advantages over the SDE approach. First, given a fixed initial noise vector $\mathbf{x}(T)$, the generation process is deterministic. Thus, we have a unique flow map, and can invert a real image into its corresponding latent noise representation by integrating the ODE forward in time. Second, ODE trajectories are generally smoother and easier to solve than SDEs. We can employ advanced black-box ODE solvers (such as Runge-Kutta methods) with adaptive step sizes to generate high-quality samples in fewer steps compared to the Euler-Maruyama method required for SDEs. Third, using the instantaneous change of variables formula, one can compute the exact log-likelihood of data under the probability flow ODE, which facilitates quantitative evaluation of model performance.

4.3.2 Flow Matching

While the probability flow ODE derived from SDEs provides a deterministic process for generation, it is essentially a by-product of the stochastic diffusion process. The vector field $f(\mathbf{x}(t), t) - \frac{1}{2}g(t)^2\nabla_{\mathbf{x}(t)} \log p_t(\mathbf{x}(t))$ in Eq. (97) is determined by the specific choice of the forward diffusion SDE (e.g., variance exploding or variance preserving). This raises a question: can we learn a vector field for the probability flow ODE directly, without relying on the intermediate diffusion formulation? Moreover, can we design this vector field to have desirable properties, such as straighter trajectories, which would be easier to integrate numerically?

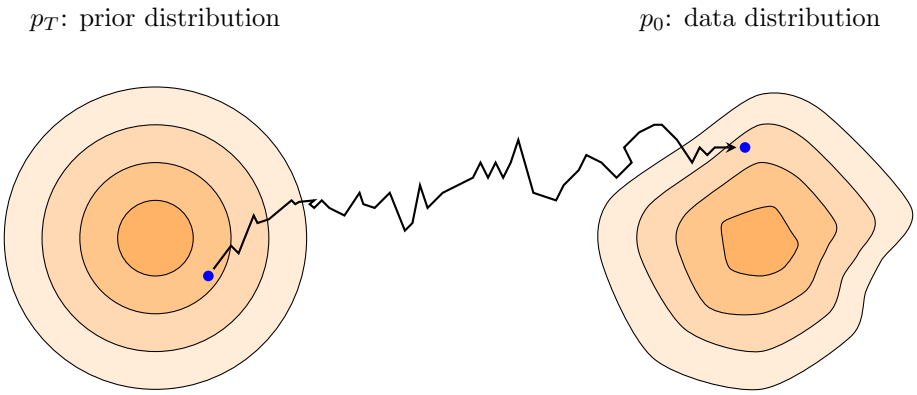
This motivates the framework of flow matching [Lipman et al., 2023; Liu et al., 2023; Albergo and Vanden-Eijnden, 2023]. Flow matching is a simulation-free approach to training continuous normalizing flows that allows for flexible definitions of the probability path.

The core idea of flow matching is to directly regress a neural network vector field $v(\mathbf{x}(t), \theta, t)$ (or $v_\theta(\mathbf{x}(t), t)$) against a target vector field $u_t(\mathbf{x}(t))$ that generates a desired probability path $p_t(\mathbf{x}(t))$. Here, we compare the flow matching-based approach with the score-based approach, as follows:

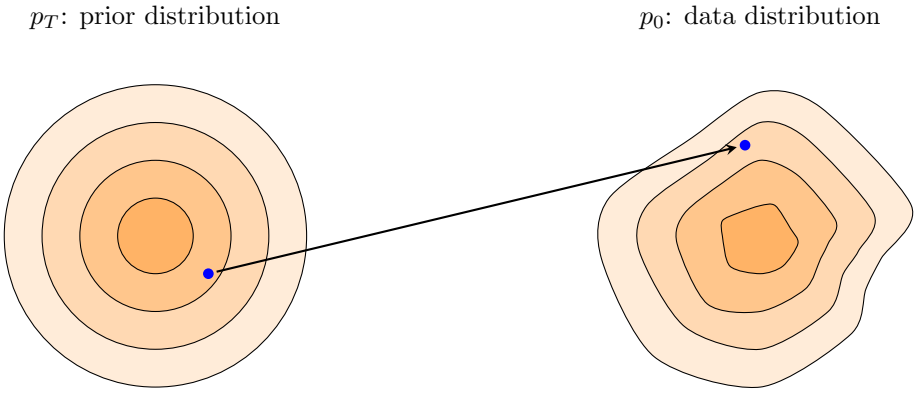
- **Score-based Approach**. The forward process is a fixed Gaussian diffusion. As shown in the previous subsections, the corresponding probability flow ODE has a vector field composed of a linear drift and a score term. The resulting trajectories are often fluctuating because the diffusion process destroys information according to a rigid schedule (e.g., exponentially decaying signal). As a result, solving the reverse ODE requires many steps or complex solvers to trace these fluctuating paths accurately.

- **Flow Matching Approach**. Instead of having the dynamics determined by an SDE, flow matching allows us to design the path. We explicitly define how a sample from the noise distribution maps to a sample from the data distribution. For instance, the simplest transport map between two points is a straight line. By enforcing straight trajectories, the vector field becomes constant in time along the path. During inference, such an ODE is easier to solve.

Figure 14 illustrates this difference. While diffusion models find a curved path between noise and data, flow matching aims to find a more efficient path that can reach the target data more directly.



(a) Trajectory of the score-based approach



(b) Trajectory of the flow matching approach (straight path transport)

Figure 14: Comparison of generative trajectories from the prior noise (p_T) to the data distribution (p_0). In subfigure (a), the score-based approach (via SDE) generates a stochastic trajectory. The process involves random noise injection, resulting in a jagged path that typically requires many discretization steps. In subfigure (b), the flow matching approach constructs a vector field that transports samples along a straight path. Here we show the ideal case where generation is performed in a single step. Although more generation steps are generally required in practice, this linearity makes the generation significantly more efficient compared to SDE paths.

To align with the notation used in this paper, we adapt the standard flow matching formulation (typically defined from $t = 0$ as noise to $t = 1$ as data, as in [Liu et al., 2023]) to our time setting where $t = 0$ corresponds to the data distribution p_0 and $t = T$ corresponds to the noise prior p_T.

In the SDE framework, the probability path $p_t(\mathbf{x}(t))$ is implicitly defined by the diffusion process. We do not know the closed-form expression of an intermediate sample $\mathbf{x}(t)$ without simulating the SDE or computing complex marginals. Flow matching flips this logic: we construct the path explicitly. The key insight of flow matching is to work with conditional probability paths. Instead of modeling the intractable marginal vector field $u_t(\mathbf{x}(t))$ directly, we condition the path on a specific data sample $\mathbf{x}(0) \sim p_0$ and a specific noise sample $\mathbf{x}(T) \sim p_T$. We define a conditional flow $\psi_t(\mathbf{x}(t)|\mathbf{x}(0), \mathbf{x}(T))$ that creates a bridge between them. A simple and effective choice is the linear interpolation

$$\begin{aligned} \mathbf{x}(t) &= \psi_t(\mathbf{x}(t)|\mathbf{x}(0), \mathbf{x}(T)) \\ &= \left(1 - \frac{t}{T}\right)\mathbf{x}(0) + \frac{t}{T}\mathbf{x}(T), \end{aligned} \tag{99}$$

This equation defines a forward process purely through geometry rather than stochastic diffusion steps. We can interpret this as drawing a straight line connecting a data point and a noise point in the high-dimensional space. The velocity of this path is simply the time derivative of $\mathbf{x}(t)$, given by

$$\begin{aligned} u_t(\mathbf{x}(t)|\mathbf{x}(0), \mathbf{x}(T)) &= \frac{d\mathbf{x}(t)}{dt} \\ &= \frac{1}{T}(\mathbf{x}(T) - \mathbf{x}(0)). \end{aligned} \tag{100}$$

Note that this target vector field is constant with respect to time for a specific pair $(\mathbf{x}(0), \mathbf{x}(T))$, which implies a straight trajectory. This path design is often referred to as the **optimal transport** (OT) conditional vector field, as it represents the most efficient (shortest) path to transport mass from one distribution to another under squared Euclidean cost.

From a formal perspective, this modeling approach offers a significant advantage: it eliminates the dependency on a stochastic diffusion process to define the forward dynamics. Instead, the forward trajectory from $t = 0$ to T is obtained directly via the analytic interpolation in Eq. (99). This allows us to efficiently generate intermediate states $\mathbf{x}(t)$ at any arbitrary time t without the computational overhead of iterative simulation. These interpolated samples, along with their corresponding target velocities, serve directly as the ground truth for training the "backward" model. Specifically, the neural network vector field $v_\theta(\mathbf{x}(t), t)$ is optimized to regress against this target velocity u_t, thereby learning to reverse the flow for generation.

Given the above formulation, we can learn the global vector field $v_\theta(\mathbf{x}, t)$ by regressing it against the conditional vector fields. Lipman et al. [2023] proved that minimizing the difference between the model output and the conditional vector field is equivalent to minimizing the difference against the true marginal vector field.

The training objective, known as **conditional flow matching** (CFM), is defined as

$$\mathcal{L}_{\text{CFM}}(\theta) = \mathbb{E}_{t,\mathbf{x}(0),\mathbf{x}(T)}\left[\|v_\theta(\mathbf{x}(t), t) - u_t(\mathbf{x}(t)|\mathbf{x}(0), \mathbf{x}(T))\|^2\right]. \tag{101}$$

In practice, the training procedure is quite simple:

Training a Flow Matching Model:

1. Sample a data point $\mathbf{x}(0) \sim p_{\text{data}}$ and a noise point $\mathbf{x}(T) \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$.
2. Sample a time $t \sim \mathcal{U}[0, T]$.
3. Compute the intermediate state $\mathbf{x}(t)$ via linear interpolation (Eq. (99)).
4. Compute the target velocity $u_t = (\mathbf{x}(T) - \mathbf{x}(0))/T$.
5. Update the network v_θ to minimize $\|v_\theta(\mathbf{x}(t), t) - u_t\|^2$.

Intuitively, during training, the model sees many crossing straight lines connecting data and noise. By considering all these conditional paths, the network learns a coherent global vector field that pushes noise towards data regions (and vice versa) in the most direct way possible. This leads us to an important theoretical insight regarding the nature of the learned vector field. Since the objective function employs the mean squared error, the optimized model v_θ does not memorize individual trajectories. Instead, it converges to the conditional expectation of the target vector fields given the current state. Mathematically, the optimal vector field $v^*(\mathbf{x}, t)$, referred to as the **marginal vector field**, satisfies

$$v^*(\mathbf{x}, t) = \mathbb{E}_{p(\mathbf{x}(0), \mathbf{x}(T)|\mathbf{x}(t))} \left[u_t(\mathbf{x}(t)|\mathbf{x}(0), \mathbf{x}(T)) \right]. \tag{102}$$

It implies that at any specific location $\mathbf{x}(t)$ and time t, the learned vector is the **statistical average** of the instantaneous velocities of all possible straight-line trajectories passing through this point, as illustrated in Figure 15. As a result, the learned global vector field generates the marginal probability path $p_t(\mathbf{x})$, which transports the entire noise distribution to the data distribution.

Once the model $v_\theta(\mathbf{x}, t)$ is trained to approximate the vector field pointing from $t = 0$ (data) to $t = T$ (noise), generation is performed by solving the ODE backwards in time. We start with pure noise $\mathbf{x}(T) \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and integrate to $t = 0$. Since we integrate backwards, the numerical update uses the negative direction of the field: $\mathbf{x}(t - \Delta t) \approx \mathbf{x}(t) - v_\theta(\mathbf{x}(t), t)\Delta t$.

The primary advantage of this approach over standard diffusion models is the trajectory shape. Flow matching with the optimal transport path encourages the learned global vector field to be as straight as possible. The resulting straight trajectories are much easier for ODE solvers to track, and we can adopt larger step sizes Δt. As a result, flow matching models can often generate high-quality samples in fewer steps compared to SDE-based diffusion models.

4.4 Towards More Efficient Generation

A bottleneck in deploying diffusion models for real-world applications is their high computational cost during inference. Efficient diffusion modeling is itself a broad topic [Shen et al., 2025]. Here we briefly outline recent methods for efficient generation, including high-order numerical solvers, trajectory rectification, and consistency models.

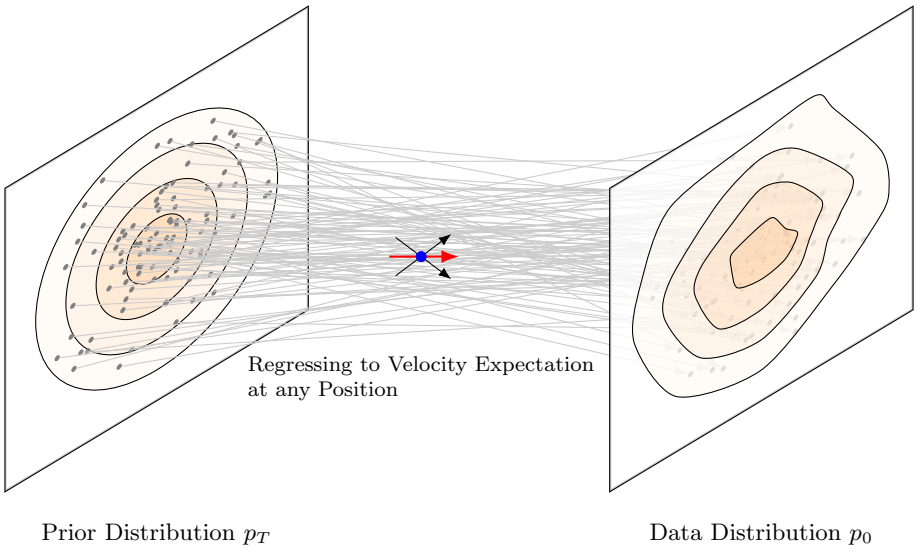


Figure 15: Illustration of the marginal vector field learning in flow matching. The gray lines represent linear interpolation trajectories connecting noise samples $\mathbf{x}(T)$ to data samples $\mathbf{x}(0)$. During training, at a specific location $\mathbf{x}(t)$ (blue dot), multiple trajectories may pass through the same point. Since the training objective minimizes the mean squared error, the learned vector field $v_\theta(\mathbf{x}, t)$ does not memorize individual paths but instead converges to the conditional expectation of the target velocities (red arrow).

4.4.1 High-Order and Specialized Solvers

One of the most straightforward approaches to acceleration is to use high-order numerical solvers. However, standard solvers like the first-order Euler method (e.g., DDIM sampling in the diffusion context) require many steps to minimize discretization errors when trajectories are curved. To address this, high-order solvers have been used to exploit the specific mathematical structure of diffusion ODEs.

For example, DPM-Solver and DPM-Solver++ make use of the semi-linear structure of diffusion ODEs to analytically compute the linear part of the solution, thus simplifying the non-linear component into an exponentially weighted integral [Lu et al., 2022]. The core principle lies in the variation of constants formula, which separates the ODE into a linear drift term and a non-linear score-based term. By analytically solving the linear part, the solver only needs to approximate the non-linear score function using $(k-1)$-th order polynomials. DPM-Solver++ further improves stability by using a data-prediction parameterization and dynamic thresholding to prevent the solution from drifting outside the valid data range [Lu et al., 2025]. Most recently, Zheng et al. [2023] introduced DPM-Solver-v3, featuring a predictor-corrector framework that refines the integral approximation using an inner predictor step. By combining this with coefficients optimized via empirical model statistics, it achieves superior performance in few-step regimes (e.g., < 10 steps) without additional function evaluations.

Beyond general numerical methods, several distinct approaches have been developed from geometric and mathematical perspectives. For instance, the approximate mean-direction solver (AMED-Solver) exploits the observation that sampling trajectories often reside in low-dimensional subspaces [Zhou et al., 2024]. By learning to predict the mean direction of the flow, it achieves convergence in fewer steps. Similarly, based on exponential integrators, the diffusion exponential integrator sampler (DEIS) reduces discretization error by fitting polynomials in a transformed log-rho space [Zhang and Chen, 2022]. This transformation linearizes the curvature of the sampling trajectory more effectively than standard time-steps. Alternatively, the unified predictor-corrector method (UniPC) introduces a free corrector step that reuses the noise prediction already evaluated during the predictor step [Zhao et al., 2023]. By reusing model outputs, this method increases the order of accuracy without requiring additional neural function evaluations. Pseudo numerical methods for diffusion models (PNDM) treat the denoising process as solving a differential equation on a manifold, which represents the high-density region of the data. They decompose the numerical step into a gradient part and a nonlinear transfer part. The transfer part ensures that even if the trajectory is curved, the resulting state x_t remains constrained to the data manifold [Liu et al., 2022]. In addition, the elucidating diffusion model (EDM) uses a 2nd-order Heun solver [Karras et al., 2022]. By scaling network inputs, outputs, and losses to unit variance, it stabilizes training dynamics and improves score estimation.

4.4.2 Trajectory Rectification

As presented previously, flow matching aims to learn a vector field $v_\theta(\mathbf{x}(t), t)$ that induces a probability flow capable of transforming a simple prior distribution p_T into a complex data distribution p_0. The core objective is to minimize the difference between the learned velocity field and the ground-truth conditional velocities that transport samples along a chosen path (Eq. (101)). However, a problem arises from the coupling between the data distribution p_0 and the noise distribution p_T.

In the initial training phase, frequently referred to as 1-rectified flow, the data points $\mathbf{x}(0)$ and noise points $\mathbf{x}(T)$ are paired randomly from their respective distributions. This random pairing leads to a phenomenon where trajectories in the high-dimensional latent space frequently intersect. When multiple straight trajectories cross at the same point $\mathbf{x}(t)$ in spacetime, the learned marginal vector field $v_\theta(\mathbf{x}(t), t)$ (i.e., the expectation of these various intersecting velocities) becomes curved. Consequently, the actual paths followed during the integration of the ODE are not straight, which leads to more discretization errors when adopting large step sizes in numerical solvers [Zhang et al., 2025b; Yang et al., 2025a]. In the rectified flow framework, one can improve the coupling between the noise and data distributions by reducing the interference between conditional paths. First, a 1-rectified flow v_θ^1 is trained using random pairings $(\mathbf{x}(0), \mathbf{x}(T))$. Then, the trained model v_θ^1 is used to generate a new synthetic dataset. Specifically, for a set of noise samples $\{\mathbf{x}_i(T)\}$, the ODE $\frac{d\mathbf{x}(t)}{dt} = -v_\theta^1(\mathbf{x}, t)$ is solved to obtain the corresponding generated data samples $\{\hat{\mathbf{x}}_i(0)\}$. This creates a set of deterministic, rewired pairs $(\hat{\mathbf{x}}_i(0), \mathbf{x}_i(T))$. Finally, a new model v_θ^2 (called 2-rectified flow) is trained using these aligned pairs. Because $\hat{\mathbf{x}}_i(0)$ was generated from $\mathbf{x}_i(T)$ via the dynamics of the first model, these pairs are highly aligned in the latent space. Thus the resulting vector field is significantly more linear.

The search for straight trajectories is rooted in the principles of optimal transport and the minimization of the Wasserstein distance. An important result in this domain is that for a rectified flow from a Gaussian to a mixture of two Gaussians, a few rectifications are sufficient to achieve straight flows. This insight underpins the empirical success of models like InstaFlow and Stable Diffusion, which make use of the straightening properties of rectified flow to reduce the number of function evaluations while maintaining sample quality [Esser et al., 2024]. The success of these commercialized models also demonstrates that diffusion models with rectified flows can be well scaled up for high-quality image generation.

The straightness of the trajectory is also linked to the Lipschitz constant of the learned vector field. If the 1-rectified flow is L-Lipschitz, nearby noise samples are prevented from mapping to arbitrarily distant data points. However, in practice, L can be large, leading to significant bending of the transport paths in the early stages of generation. This bending degrades the accuracy of one-step Euler sampling, which may require further rectification or more advanced sampling schedules.

The practical implementation of trajectory rectification also involves architectural improvements. Researchers have replaced simple U-Net architectures with sophisticated multimodal diffusion Transformers (MMDiT). For example, Stable Diffusion 3 introduced the MMDiT architecture [Esser et al., 2024], while Flux.1 further scaled the rectified flow transformer architecture to 12 billion parameters [Greenberg, 2025]. This shows that diffusion models can be scaled to very large sizes while maintaining high sampling efficiency.

To further enhance efficiency and stability, it is also possible to apply the divide-and-conquer strategy to rectified flows. For example, the piecewise rectified flow (PeRFlow) partitions the generative trajectory into several discrete time windows [Yan et al., 2024]. Instead of straightening the global flow, it straightens segments within each interval individually, and this localization reduces the computational cost of simulating training trajectories and minimizes numerical errors. Furthermore, researchers have addressed the stability of conditional generation through advanced guidance techniques. For example, Rectified-CFG++ employs an adaptive predictor-corrector scheme [Saini et al., 2025]. Each generation step first executes a conditional update to anchor the sample near the learned transport path, then applies a manifold-aware correction based on the difference between conditional and unconditional velocities. This ensures that sampling trajectories remain within a stable tubular neighborhood of the data manifold.

4.4.3 Consistency Models

Consistency modeling represents a shift away from iterative integration toward a direct mapping paradigm. Instead of traversing a path, a consistency model is designed to learn a function that maps any point at any time step on a probability flow ODE trajectory directly to its origin [Song et al., 2023]. This self-consistency property ensures that all points residing on the same path result in the same output image, and so we can generate high-quality results using one-step/few-step generation.

To achieve this, consistency models are generally trained via two mechanisms: consistency training and consistency distillation. While consistency training allows training from scratch, consistency distillation has proven more practical for large-scale applications by distilling the

knowledge of a pre-trained standard diffusion model (the teacher) into a consistency model (the student). The distillation loss minimizes the discrepancy between the prediction of the student at the current time step t and its prediction at the next step. This compresses the multi-step trajectory of the teacher into a single jump. To further improve generation quality, especially for one-step sampling, adversarial loss functions have been introduced to ensure the predicted samples remain indistinguishable from real data distributions [Sauer et al., 2024].

An extension of this paradigm is the latent consistency model (LCM), which applies consistency distillation to the latent space of high-resolution image synthesis models like Stable Diffusion [Luo et al., 2023a]. By distilling the probability flow of the latent ODE, LCMs drastically reduce inference latency while retaining the high-resolution capabilities of the parent model. Furthermore, to address the computational overhead of retraining large backbones, LCM-LoRA introduces a parameter-efficient fine-tuning strategy. This approach trains a lightweight low-rank adaptation (LoRA) module that acts as a universal accelerator. Thus, various fine-tuned diffusion models can be used to acquire consistency properties without altering their pre-trained parameters [Luo et al., 2023b].

While early LCMs achieved significant speedups, they often suffered from degraded quality as the number of sampling steps increased. To address this, trajectory consistency distillation (TCD) reformulates the consistency function in a semi-linear framework inspired by exponential integrators to allow mapping to arbitrary intermediate steps [Zheng et al., 2024]. Moreover, phased consistency models (PCM) partition the trajectory into multiple sub-trajectories to enable deterministic multi-step sampling without stochasticity error accumulation [Wang et al., 2024]. Recent continuous-time models like simplified consistency models (sCM) also introduce the TrigFlow formulation to stabilize training [Lu and Song, 2025].

Note that, although consistency models are optimized for single-step generation, they also support multistep consistency sampling. By taking 2 to 4 steps and re-encoding the intermediate outputs, the model can iteratively correct discretization errors. This offers a flexible trade-off between inference speed and sample fidelity.

5. Diffusion Models in Language

We have seen that image generation can be treated as the evolution of a state along a differential equation trajectory. The resulting diffusion models provide a powerful tool to describe the bidirectional transformation between prior and data distributions. In this section, we extend this approach to the generation of token sequences, which is a fundamental problem in modern **natural language processing** (NLP).

A key challenge is that tokens are discrete variables that carry linguistic meaning. Specifically, unlike the continuous space of images, there is no meaningful intermediate state between two discrete tokens. Moreover, the categorical and sequential nature of language means that even small changes in token identity or order can drastically alter linguistic meaning. Therefore, we need to adapt diffusion models to token sequence modeling by modifying existing methods and algorithms. In this section, we begin by introducing autoregressive and non-autoregressive generation paradigms. We then consider two approaches to diffusion modeling in language: embedding-space

diffusion and discrete diffusion. The former is based on continuous-time diffusion models and operates in the embedding space of tokens. The latter directly defines generation in the discrete token space and generally leverages token masking and prediction mechanisms. Furthermore, we discuss the application of diffusion Transformer architectures and flow matching in token sequence generation, as well as the issues of inference and generation control.

Diffusion modeling is a wide-ranging topic. Here, we try to discuss it from the perspective of differential equations, and so connect key concepts to those presented in previous sections. We also present methods that are not originally motivated by differential equations but are included to have a comprehensive discussion.

5.1 Token Sequence Generation

We first present the concepts of **autoregressive (AR) generation** and **non-autoregressive (NAR) generation**, and then show that diffusion modeling can be treated as an NAR generation problem.

5.1.1 Autoregressive Generation vs. Non-autoregressive Generation

Token sequence generation (or **sequence generation** for short) is a fundamental task in NLP, and lays the foundation for **large language models** (LLMs) [Radford et al., 2018]. The dominant approach is AR generation[11]. In NLP, a typical implementation is left-to-right generation, which predicts one token at a time given all its preceding tokens. Formally, let $\mathbf{x} = \{x_1, ..., x_n\}$ be a token sequence. The probability of this sequence is defined using the chain rule of probability

$$\begin{aligned} \Pr(\mathbf{x}) &= \Pr(x_1, ..., x_n) \\ &= \prod_{i=1}^{n} \Pr(x_i | x_{<i}). \end{aligned} \tag{103}$$

Here, x_i represents the token being generated at the current step, and $x_{<i}$ represents all the tokens generated previously. $\Pr(x_i|x_{<i})$ is the conditional probability of the next token, which reflects the nature of left-to-right generation. There are many ways to implement $\Pr(x_i|x_{<i})$. A popular method is to use neural networks to model this probability. For example, LSTMs and Transformers are both widely used architectures in neural language modeling.

In many text generation tasks, the generation is conditioned on user-provided information. We denote this contextual information as $\mathbf{c}$, which can be broadly viewed as a sequence of variables $\mathbf{c} = \{c_1, ..., c_m\}$. Then, the generation task is to model the probability of predicting the following token sequence $\mathbf{x}$ given the context $\mathbf{c}$

$$\Pr(\mathbf{x}|\mathbf{c}) = \prod_{i=1}^{n} \Pr(x_i | x_{<i}, \mathbf{c}). \tag{104}$$

11. In statistics, *regression* refers to estimating the relationship between a dependent variable (Y) and independent variables (X). The prefix *auto* means *self* in Greek. Therefore, *autoregression* means performing a regression analysis where the independent variables are the lagged values of the variable itself. Instead of predicting Y based on X, one predicts Y_{today} based on $Y_{\text{yesterday}}$.

Here $\mathbf{c}$ can be either a discrete token sequence (e.g., a prompt) or a real-valued vector sequence (e.g., representations generated by a neural network). In general, $\mathbf{c}$ can be defined according to the task of interest. For example, we can treat it as a source sentence in translation, or a prompt in QA, or a real-valued representation of history in an external memory.

The primary advantage of AR generation lies in its simplicity. This leads to several desirable properties. For example, it aligns well with how humans perceive speech and writing, and is intuitive for tasks like dialogue and storytelling. Moreover, AR generation can naturally handle sequences of varying lengths. The model continues generating until it outputs a special End-of-Sequence (EOS) token, rather than being forced to fill a fixed-size template. However, AR generation has disadvantages that have long been concerns in the NLP community. A major disadvantage is its slow generation speed. Because step i depends on step $i-1$, generation cannot be parallelized. To generate a sentence with 100 tokens, the model must run the forward pass 100 times sequentially. This makes real-time applications computationally expensive. Another disadvantage lies in its unidirectional context scope. When predicting token x_i, the model can only see to the left ($x_{<i}$). It cannot look ahead to the right to see how the sentence should end.

NAR generation is an alternative to AR generation that aims to break the sequential dependency chain [Gu et al., 2018]. Instead of generating tokens one by one, NAR models generate all tokens in the sequence in parallel. Mathematically, this approach relies on the assumption of conditional independence among target tokens given the input. The probability of the sequence $\mathbf{x}$ can then be factorized as

$$\Pr(\mathbf{x}|\mathbf{c}) \quad = \quad \prod_{i=1}^{n} \Pr(x_i|\mathbf{c}) \tag{105}$$

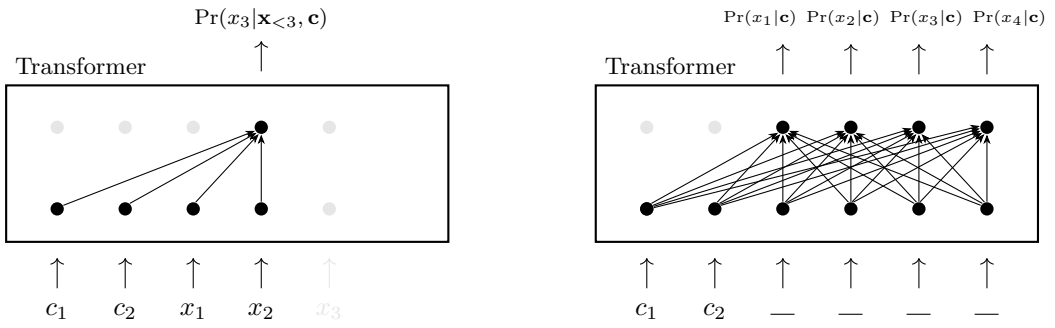
In this formulation, the prediction of token x_i does not depend on other target tokens x_j ($j \neq i$). Thus, NAR generation is efficient for inference. By decoupling the dependencies between tokens, the model can predict the entire sequence in a single forward pass, which reduces the number of inference steps from $O(n)$ to $O(1)$. This parallelism allows modern GPUs to be utilized more efficiently, and thus makes NAR models attractive for real-time applications where low latency is critical. In addition, NAR models can make full use of information from both the past and the future positions during the generation process. Figure 16 illustrates the differences between AR and NAR generation.

It should be noted, however, that the conditional independence assumption also has limitations. For example, in natural language, there are often multiple valid ways to complete a sentence. Without knowing what tokens have been selected at other positions, an NAR model might mix different valid patterns, leading to the so-called multimodality problem[12].

5.1.2 Diffusion Modeling as Non-autoregressive Generation

Despite the above limitations, the parallel nature of NAR generation aligns perfectly with diffusion models, which typically operate on the entire data dimensionality simultaneously. Therefore,

12. This problem may result in repetition, omission, or grammatical incoherence, e.g., choosing "New" at position i and "London" at position $i+1$ when the valid entities are "New York" or "London".



(a) Autoregressive Generation (Single Step)

(b) Non-autoregressive Generation (Single Step)

Figure 16: Autoregressive (AR) vs. non-autoregressive (NAR) generation. $\mathbf{c} = \{c_1, c_2\}$ denotes the user-provided context, and $\mathbf{x} = \{x_1, x_2, x_3, x_4\}$ denotes the target sequence. We aim to model the probability of $\mathbf{x}$ given $\mathbf{c}$. (a) In AR generation, tokens are predicted sequentially, and the prediction of x_i depends on the history $\mathbf{x}_{<i}$ and $\mathbf{c}$. (b) In NAR generation, all tokens in the sequence are predicted in parallel. The dashes in (b) represent a template of 4 mask tokens (given a target length).

most of the diffusion models in NLP are based on the NAR generation paradigm. To understand this connection, it is helpful to generalize the concept of NAR generation beyond a single-step prediction. Let us reimagine the generation process as a dynamical system evolving over time t. We define $\mathbf{s}(t)$ as the intermediate state of the system at time t. This abstract state $\mathbf{s}(t)$ serves as a unified representation: it can be either a sequence of discrete variables (e.g., corrupted tokens) or a sequence of continuous vectors (e.g., token embeddings). During generation, the system evolves backwards in time from $t = T$ to $t = 0$. The states of this system are defined as follows

- The starting point $\mathbf{s}(T)$. This represents a sample drawn from a prior distribution (e.g., Gaussian noise vectors or completely random tokens). It contains no semantic information.
- The intermediate states $\mathbf{s}(t)$ (for $0 < t < T$). These states represent the evolving latent variables. They can be viewed as noisy approximations of the target data.
- The end point $\mathbf{s}(0)$. The state $\mathbf{s}(0)$ represents the fully denoised signal. Finally, a deterministic or stochastic mapping (e.g., rounding or an argmax operation) projects $\mathbf{s}(0)$ back to the valid token sequence $\mathbf{x} = \{x_1, \ldots, x_n\}$.

In this process, we need to incorporate the condition $\mathbf{c}$ into each state $\mathbf{s}(t)$ to form a joint state $(\mathbf{s}(t), \mathbf{c})$. Hence the evolution can be described as

$$(\mathbf{s}(T), \mathbf{c}) \to \cdots \to (\mathbf{s}(t), \mathbf{c}) \to \cdots \to (\mathbf{s}(0), \mathbf{c}) \to \mathbf{x}. \tag{106}$$

The final map $(\mathbf{s}(0), \mathbf{c}) \to \mathbf{x}$ is generally performed by using a separate system (e.g., mapping embeddings to tokens). The steps $(\mathbf{s}(T), \mathbf{c}) \to \cdots \to (\mathbf{s}(t), \mathbf{c}) \to \cdots \to (\mathbf{s}(0), \mathbf{c})$ can be viewed as

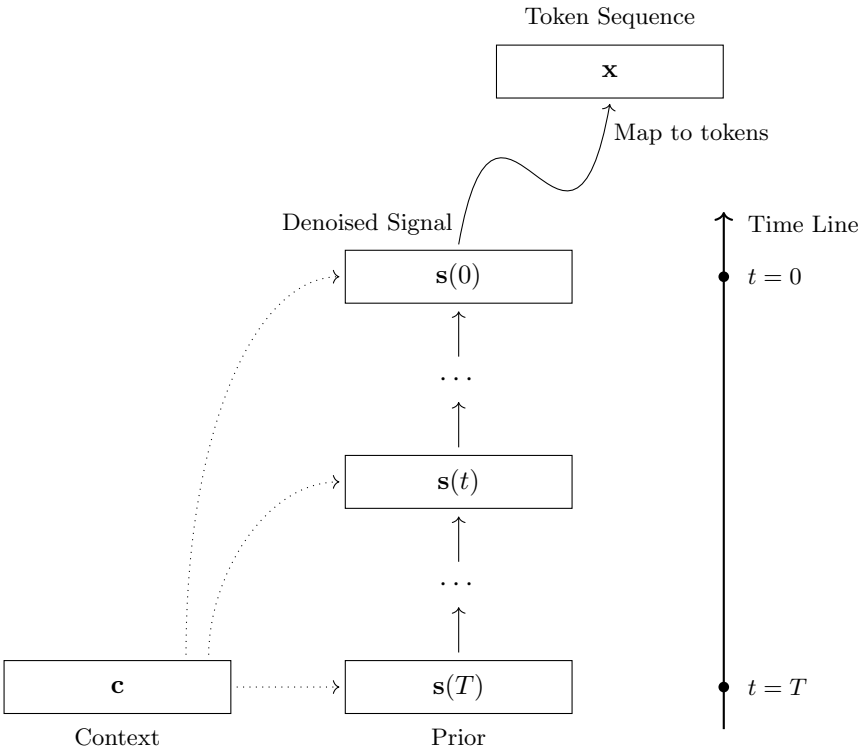


Figure 17: Schematic illustration of NAR generation based on diffusion models. The generation process is modeled as the evolution of a state $\mathbf{s}(t)$ over time t. Here $\mathbf{s}(T)$ denotes the starting point (prior), and $\mathbf{s}(0)$ denotes the end point (fully denoised signal). The contextual information $\mathbf{c}$ is injected into the state during generation. The final output is a token sequence that is mapped from $\mathbf{s}(0)$.

the evolution of a dynamical system, as in standard diffusion models. This process is illustrated in Figure 17.

In the remainder of this section, we use diffusion models as tools to implement NAR generation. We categorize the approaches based on how they define the state space of the dynamical system $\mathbf{s}(t)$. We first introduce embedding-space diffusion, which maps tokens to continuous vectors to apply continuous Gaussian diffusion. We then discuss discrete diffusion, which defines the corruption and denoising processes directly on the discrete tokens.

5.2 Diffusion in Continuous Space: Embedding-Space Diffusion

In this subsection, we introduce the embedding-space diffusion framework, which adapts continuous diffusion models to token sequence generation.

5.2.1 General Framework

One intuitive approach to applying diffusion models to text is to map discrete tokens into a continuous embedding space $\mathbb{R}^d$, so that the diffusion and generation processes can operate in

a continuous domain. Several prominent works are based on this embedding-space diffusion framework [Li et al., 2022b; Strudel et al., 2022; Gong et al., 2023]. In these methods, the first step is to transform each token of a sequence into a d-dimensional real-valued vector (called token embedding). Formally, let V be a vocabulary of size $|V|$. Each token $x \in V$ is typically represented as a one-hot vector and projected into a dense vector. Then, a sequence of tokens $\mathbf{x} = \{x_1, \ldots, x_n\}$ can be mapped to a sequence of continuous vectors via an embedding function $\text{Embed}(\cdot)$, as follows

$$\begin{aligned} \text{Embed}(\mathbf{x}) &= [\mathbf{e}_1, \ldots, \mathbf{e}_n], \\ &= \mathbf{e} \end{aligned} \tag{107}$$

where $\mathbf{e}_i \in \mathbb{R}^d$ denotes the embedding at position i, and $\mathbf{e} \in \mathbb{R}^{d \times n}$ denotes the sequence of embeddings (stacked as a matrix). Likewise, we can transform the context token sequence $\mathbf{c}$ into an embedding sequence $\mathbf{e}_c \in \mathbb{R}^{d \times n_c}$.

A simple way to define $\text{Embed}(\cdot)$ is via a static embedding matrix $\mathbf{E} \in \mathbb{R}^{d \times |V|}$ [Mikolov et al., 2013a;b]. While simple, this approach treats tokens as independent units, and thus ignores their contextual dependencies in the sequence. Alternatively, one can map the entire token sequence into a sequence of contextualized representations. For instance, pre-trained LLMs can be used to encode both the tokens and their surrounding context [Devlin et al., 2019; Brown et al., 2020]. Note that this transformation is not limited to a one-to-one correspondence between tokens and vectors, nor is it constrained to a fixed sequence length n. Instead, we can employ higher-level abstractions to map the token sequence into a latent continuous sequence of a different length n' (where n' need not equal n). For instance, by using VAEs, discrete sequences can be compressed into denser, semantically rich latent representations. In this latent diffusion paradigm [Rombach et al., 2022], the diffusion process operates on these compressed latent variables rather than on raw token embeddings.

Once the tokens are represented as embeddings, the generation problem can be formulated using a diffusion model in a standard manner. Formally, let $\mathbf{e}_c$ be the context embedding sequence. We define $\bar{\mathbf{s}}(t)$ as the augmented state combining $\mathbf{e}_c$ and $\mathbf{s}(t)$

$$\bar{\mathbf{s}}(t) = [\mathbf{e}_c, \mathbf{s}(t)]. \tag{108}$$

We treat the concatenation of $\mathbf{e}_c$ and $\mathbf{e}$ as the initial state $\bar{\mathbf{s}}(0) = [\mathbf{e}_c, \mathbf{e}] \in \mathbb{R}^{d \times (n_c + n)}$ of a continuous diffusion process. Then we define a forward process that gradually adds Gaussian noise to the target embeddings over time $t \in [0, T]]$, while keeping the context embeddings fixed. It transforms the meaningful token embeddings into standard Gaussian noise $\mathcal{N}(\mathbf{0}, \mathbf{I})$.

As described previously, there are several models to choose from. For example, we can describe the diffusion process by an SDE

$$d\bar{\mathbf{s}}(t) = f(\bar{\mathbf{s}}(t), t)dt + g(t)d\mathbf{w}, \tag{109}$$

where $f(\bar{\mathbf{s}}(t), t)$ is the drift coefficient, $g(t)$ is the diffusion coefficient, and $\mathbf{w}$ is a standard Wiener process. The generation process corresponds to the reverse-time SDE, which reconstructs $\bar{\mathbf{s}}(0)$

from a state $\bar{\mathbf{s}}(T)$ where the target sequence part is Gaussian noise:

$$d\bar{\mathbf{s}}(t) = [f(\bar{\mathbf{s}}(t), t) - g(t)^2 \nabla_{\bar{\mathbf{s}}(t)} \log p_t(\bar{\mathbf{s}}(t))]dt + g(t)d\bar{\mathbf{w}}, \tag{110}$$

where $\bar{\mathbf{w}}$ is the reverse-time Wiener process, and $\nabla_{\bar{\mathbf{s}}(t)} \log p_t(\bar{\mathbf{s}}(t))$ is the score function. Alternatively, one can use flow matching to learn a vector field that pushes forward the distribution from Gaussian noise to the data distribution. A detailed discussion of these methods is omitted here. Readers can refer to Section 4 for more details on various diffusion models.

Note that since the context $\mathbf{c}$ and its embedding $\mathbf{e}_c$ remain constant throughout the process, we can simply use $\mathbf{s}(t)$ instead of $\bar{\mathbf{s}}(t)$ to describe the diffusion and generation processes. Hence Eqs. (109) and (110) can be equivalently rewritten as

$$d\mathbf{s}(t) = f(\mathbf{s}(t), t)dt + g(t)d\mathbf{w}, \tag{111}$$

$$d\mathbf{s}(t) = [f(\mathbf{s}(t), t) - g(t)^2 \nabla_{\mathbf{s}(t)} \log p_t(\mathbf{s}(t))]dt + g(t)d\bar{\mathbf{w}}. \tag{112}$$

A key difference from computer vision models is that in NLP we typically employ Transformers as the backbone architecture for learning score functions or vector fields. Given a state $\bar{\mathbf{s}}(t) = [\mathbf{e}_c, \mathbf{s}(t)] \in \mathbb{R}^{d\times(n_c+n)}$, the Transformer outputs a real-valued matrix of the same shape, estimating the score or vector field at each position, as illustrated in Figure 18. Unlike image generation, where the spatial dimensions are typically fixed, natural language sequences are inherently variable in length. Thus we need to determine the target sequence length n during inference. Furthermore, although the diffusion and generation processes operate in the continuous space to obtain $\bar{\mathbf{s}}(0)$, the goal is to produce discrete tokens. As a result, an additional module is required to decode the generated continuous embeddings back into the discrete tokens. We briefly discuss these two challenges below.

5.2.2 Length Prediction

Length prediction is a recurring problem in NLP with a long history of research. A straightforward approach is to explicitly predict the target sequence length based on the context. For instance, the target length can be derived simply by multiplying the source length by a specific ratio. This concept dates back to the early era of **statistical machine translation** (SMT), where fertility models were employed to determine the number of target words corresponding to a single source word [Brown et al., 1993]. Similar mechanisms were adopted during the development of NAR generation models [Xiao et al., 2023]. In early NAR generation models (typically encoder-decoder architectures), a fertility predictor was integrated into the encoder. This predictor estimates the number of times each encoder hidden state should be copied to the decoder, thereby planning the target sequence length prior to decoding. In such methods, length prediction is framed as modeling the probability $\Pr(n|\mathbf{c})$. The predictor is typically a neural network, such as a **multi-layer perceptron** (MLP), trained to maximize the likelihood of the ground-truth length given the context. During inference, the target length is determined by selecting the length n^* with the highest probability.

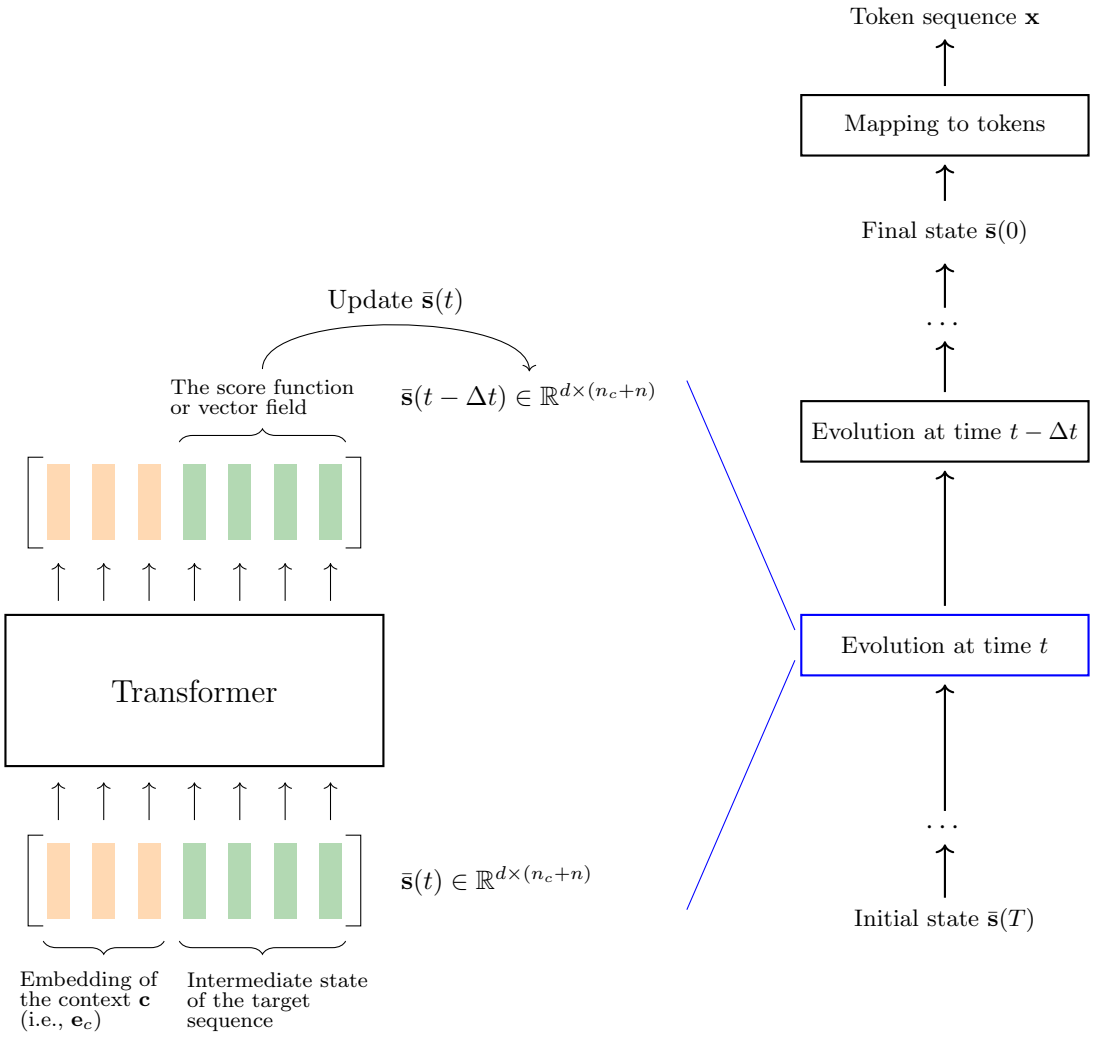


Figure 18: Schematic illustration of the Transformer-based generation process. The generation (i.e., reverse diffusion) process begins at $\bar{\mathbf{s}}(T)$, initialized as standard Gaussian noise for the target sequence. At each time step t, the state $\bar{\mathbf{s}}(t)$ represents the concatenation of the context embeddings and the noisy target embeddings. A Transformer model estimates the score function (or vector field) based on $\bar{\mathbf{s}}(t)$, which is then used to update the state to $\bar{\mathbf{s}}(t-\Delta t)$. This iterative process continues until the clean state $\bar{\mathbf{s}}(0)$ is reached, which is finally decoded into discrete tokens. Note that the context embedding $\mathbf{e}_c$ remain fixed during updates.

Explicit length prediction is simple but often leads to repetition and hallucination, as the model attempts to fill the unused capacity of the pre-allocated window with meaningless or redundant tokens. Instead, most current diffusion language models adopt implicit length determination. This approach is consistent with the behavior of AR generation by allowing the model to signal its own termination. In practice, the generation process starts with a sufficiently long window (e.g.,

the maximum sequence length). Once this process is complete and the continuous embeddings have been decoded into discrete tokens, the system identifies the first occurrence of a termination marker (such as the EOS token). This position is treated as the logical end of the sentence, and any subsequent tokens are discarded.

Recent studies have also addressed the computational inefficiency of pre-allocating maximum-length windows by introducing dynamic length adaptation mechanisms [Li et al., 2025; Yang et al., 2025b]. Rather than relying solely on post-hoc truncation via EOS tokens, these approaches integrate length modulation directly into the diffusion and generation processes. Through discrete insertion and deletion operations or flexible noise schedules, they can adjust the sequence length on-the-fly.

5.2.3 Rounding

As described above, the generation process terminates at $t = 0$, yielding a sequence of continuous vectors $\mathbf{s}(0)$. Let us denote the generated embedding sequence within $\bar{\mathbf{s}}(0)$ as $\hat{\mathbf{e}} = [\hat{\mathbf{e}}_1, \ldots, \hat{\mathbf{e}}_n]$, where each $\hat{\mathbf{e}}_i \in \mathbb{R}^d$. Since our goal is to generate text, we must map these continuous vectors back to discrete tokens in the vocabulary V. This operation is commonly referred to as **rounding** or decoding.

The most straightforward rounding method is **nearest neighbor search**. Suppose we represent tokens as embeddings using a pre-trained embedding matrix $\mathbf{E} \in \mathbb{R}^{d \times |V|}$, where each column corresponds to the embedding of a token $v \in V$. Then, we can select the token whose embedding is closest to the generated vector $\hat{\mathbf{e}}_i$ under a specific distance metric. Since diffusion models are typically trained using an MSE loss, the Euclidean distance is a natural choice

$$x_i \quad = \quad \arg\min_{v \in V} \|\hat{\mathbf{e}}_i - \mathbf{e}_v\|_2^2, \tag{113}$$

where $\mathbf{e}_v$ denotes the column of $\mathbf{E}$ corresponding to token v. Alternatively, one can employ a trainable projection layer followed by a softmax function, similar to the output layer in standard AR language models. In this case, the probability of token x_i is given by the Softmax function.

In contrast, for models based on latent or contextualized embeddings (e.g., those using VAEs or LLM representations), simple nearest neighbor search is inapplicable because the generated latent variables do not directly correspond to static token embeddings. In these approaches, a learnable decoder is required to map the continuous latent sequence back to the discrete token space. The decoding process usually involves a neural network that predicts the probability distribution over the vocabulary for each latent vector, as in AR generation models.

Rounding is not trivial due to the mismatch between the continuous diffusion latent space and the discrete nature of token embeddings. The standard diffusion training objective (such as MSE) encourages the model to predict the expected value of the data. In the context of language, if the distribution is multimodal (e.g., the next token could plausibly be "cat" or "dog"), the model might predict the average of their embeddings: $(\mathbf{e}_{\text{cat}} + \mathbf{e}_{\text{dog}})/2$. This averaged vector lies in the interstitial space between valid embeddings. Simply rounding this vector to the nearest neighbor may result in a semantically broad token (e.g., "animal"), or an entirely unrelated token if the

embedding space is not perfectly smooth. This phenomenon is often observed as the generated vectors failing to commit to a specific discrete mode.

To mitigate this issue, researchers have introduced auxiliary objectives and architectural modifications. Li et al. [2022b] proposed a trainable rounding parameter and added an auxiliary regularization term to the training objective. This term explicitly encourages the model to generate vectors that lie close to the valid embeddings in $\mathbf{E}$, thereby reducing the rounding error during inference. Strudel et al. [2022] introduced the concept of self-conditioning, where the estimate of the denoised token is projected back to the embedding space and used as input for the next step. This effectively clamps the intermediate states closer to the valid data manifold, and enables the final state $\mathbf{s}(0)$ to be reliably mapped to discrete tokens.

5.3 Diffusion in Token Space: Discrete Diffusion

While embedding-space diffusion fits well into the framework of standard diffusion models, it introduces the non-trivial challenge of rounding continuous vectors back to discrete tokens. To bypass this issue and align the generation process more naturally with the discrete nature of language, a growing body of research explores discrete diffusion. In this paradigm, the corruption process operates directly on the discrete vocabulary space V, thereby removing the need for embedding projection and rounding.

5.3.1 The CTMC Perspective

To model diffusion and generation processes directly on the discrete vocabulary V, we move away from the Gaussian noise and SDEs that are commonly used in continuous spaces. In a discrete state space, the process evolves in continuous time but changes state only via instantaneous jumps: it stays constant for a random holding time and then jumps to a new state. The mathematical framework that naturally describes such probabilistic jumps is the continuous-time Markov chain (CTMC). Just as the SDE framework views diffusion as the limit of a random walk as the step size approaches zero, the CTMC framework views discrete diffusion as the limit of a discrete-time Markov chain as the number of time steps approaches infinity.

Since the CTMC here operates on discrete tokens, we use $\mathbf{x}(t)$ instead of $\mathbf{s}(t)$ to denote a state representing a token sequence. Let $\{\mathbf{x}(t)\}_{t\geq 0}$ be a stochastic process where $\mathbf{x}(t)$ is a sequence of n tokens at time t. Let $\mathbf{x}$ and $\mathbf{y}$ denote specific states (i.e., token sequences) in the discrete space V^n. The evolution of the probability distribution $p_t(\mathbf{x}) = \Pr(\mathbf{x}(t) = \mathbf{x})$ is governed by the Kolmogorov forward equation

$$\frac{dp_t(\mathbf{x})}{dt} = \sum_{\mathbf{y}\in V^n} p_t(\mathbf{y})[\mathbf{Q}^{\text{seq}}(t)]_{\mathbf{yx}}, \tag{114}$$

where $\mathbf{Q}^{\text{seq}}(t)$ is the (time-dependent) generator matrix for the entire sequence. The term $[\mathbf{Q}^{\text{seq}}(t)]_{\mathbf{yx}}$ denotes the instantaneous rate of transitioning from state $\mathbf{y}$ to state $\mathbf{x}$, which is an entry of the large matrix $\mathbf{Q}^{\text{seq}}(t)$.

However, the state space size $|V|^n$ is exponentially large, making the full matrix $\mathbf{Q}^{\text{seq}}(t)$ computationally intractable. To address this, we generally assume independence between token

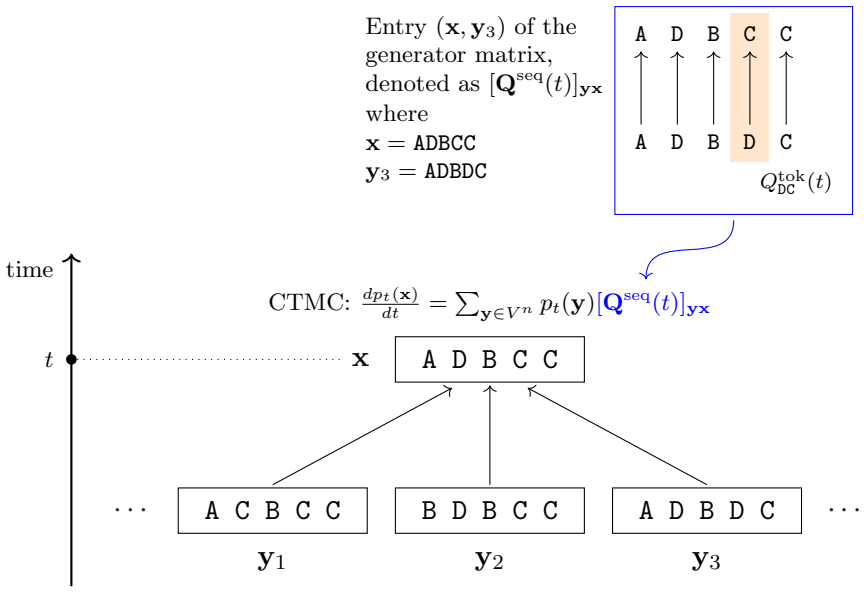


Figure 19: Illustration of the CTMC in token sequence generation. The dynamics of the CTMC at time t is $\frac{dp_t(\mathbf{x})}{dt} = \sum_{\mathbf{y}\in V^n} p_t(\mathbf{y})[\mathbf{Q}^{\text{seq}}(t)]_{\mathbf{yx}}$. It states that the rate of change of the probability of observing a target sequence $\mathbf{x}$ (e.g., ADBCC) is the sum of the probabilities from all possible source sequences (e.g., $\mathbf{y}_1$, $\mathbf{y}_2$, $\mathbf{y}_3$), weighted by the sequence-level transition rate $[\mathbf{Q}^{\text{seq}}(t)]_{\mathbf{yx}}$. The global transition rate is determined by the underlying token-level dynamics. For example, a transition from sequence $\mathbf{y}_3$ to $\mathbf{x}$ involves individual token changes, such as the rate $Q^{\text{tok}}_{\text{DC}}(t)$ for the fourth token changing from D to C. Note that while the CTMC models dynamics in continuous time, the state transitions themselves are discrete jumps rather than continuous changes.

positions during the forward process. Under this assumption, the transitions occur independently at each position. Thus, we can model the process using a much smaller single-token generator matrix $\mathbf{Q}^{\text{tok}}(t) \in \mathbb{R}^{|V|\times|V|}$ that operates on individual tokens. As a result, the complex high-dimensional dynamics of $\mathbf{Q}^{\text{seq}}(t)$ can be factorized into n parallel processes, each governed by the same single-token generator $\mathbf{Q}^{\text{tok}}(t)$. Figure 19 shows an illustration of Eq. (114).

Note that, while CTMCs provide a continuous-time view via the forward equation in Eq. (114), practical discrete diffusion models in NLP typically operate on a discretized time grid $t \in \{0, 1, \ldots, T\}$. In this case, the process is implemented through transition kernels (or matrices) between adjacent time steps, and CTMC expressions can be understood as the continuous-time limit of these discretized dynamics.

Using CTMCs, we describe the forward and backward processes as follows:

- **The Forward Process**. In CTMCs, instead of adding Gaussian noise, we corrupt the text via stochastic token substitutions. This is defined by a predetermined generator matrix $\mathbf{Q}^{\text{tok}}(t)$. Commonly used corruption schemes include:

  - **Uniform Diffusion:** The token has a uniform rate of jumping to any other token in the vocabulary. This is analogous to the Gaussian noise in continuous space which drives the data distribution toward a standard normal distribution. Here, the distribution converges to a uniform distribution over V.
  - **Masking Diffusion (Absorbing State):** Tokens transition to a special `[MASK]` token with a certain rate but never transition back. This corresponds to an absorbing state in the Markov chain.

- **The Reverse Process**. A standard result in stochastic processes states that if the forward process is a CTMC with generator $\mathbf{Q}^{\text{tok}}(t)$, the reverse process running backward in time from T to 0 is also a CTMC [Anderson, 1982]. Its generative rate matrix $\tilde{\mathbf{Q}}^{\text{tok}}(t)$ is given by

$$\tilde{Q}^{\text{tok}}_{vu}(t) \quad = \quad Q^{\text{tok}}_{uv}(t)\frac{p_t(u)}{p_t(v)}, \tag{115}$$

  where $u, v \in V$ represent specific token values in the vocabulary. Here, $\tilde{Q}^{\text{tok}}_{vu}(t)$ denotes the rate of transitioning from value v to value u in the reverse process, and $p_t(v)$ is the marginal probability of a token taking the value v at time t.

The above equation establishes the fundamental connection between CTMCs and discrete diffusion models. It is the discrete analog of the reverse-time SDE formula, which relates the reverse drift to the score function $\nabla \log p_t(\mathbf{x})$. In the discrete case, the probability ratio $p_t(u)/p_t(v)$ acts as the "discrete score", which guides the generation process toward high-probability tokens.

Eq. (115) is a theoretical characterization of the time-reversed CTMC, where the reverse rates depend on the true marginals $p_t(\cdot)$. In discrete diffusion models for text, we typically do not estimate $p_t(\cdot)$ explicitly. Instead, the reverse dynamics are constructed through a Bayes factorization of the reverse posterior $\Pr(\mathbf{x}(t-\Delta t) \mid \mathbf{x}(t), \mathbf{x}(0), \mathbf{c})$. Thus, Eq. (115) mainly serves to provide intuition for why probability ratios or posterior reweighting guide the reverse process toward high-probability tokens. In practice, we parameterize the reverse dynamics with a neural network (typically a Transformer) that predicts a denoising distribution such as $p_\theta(\mathbf{x}(0)|\mathbf{x}(t), \mathbf{c})$ [13].

A representative example in CTMC-based diffusion modeling is the **structured denoising diffusion probabilistic model** (D3PM) proposed by Austin et al. [2021]. While earlier attempts at discrete diffusion often relied on simple uniform transition probabilities, D3PM generalizes the framework to support arbitrary noise processes through structured transition matrices. Importantly, Austin et al. [2021] demonstrated that the choice of the corruption process significantly impacts sample quality. They introduced several structured transition schemes beyond standard uniform diffusion. In terms of model parameterization, D3PM is different from the noise-prediction paradigm. Since noise is not additive in discrete space, D3PM is trained to predict the clean data distribution $p_\theta(\mathbf{x}(0)|\mathbf{x}(t), \mathbf{c})$ directly from the noisy input.

13. In most settings, the forward corruption process is chosen independent of $\mathbf{c}$. We keep $\mathbf{c}$ in the notation to match the conditional generation setting and to allow the more general case where the noise process may depend on $\mathbf{c}$.

Note that since the ground truth $\mathbf{x}(0)$ is available during training, we can directly compute the true reverse posterior $q(\mathbf{x}(t-\Delta t)|\mathbf{x}(t),\mathbf{x}(0),\mathbf{c})$. Consequently, this true posterior serves as the supervision signal for the model's transition probabilities.

Formally, this reverse posterior corresponds to the distribution over the previous time step. Following common notation in diffusion models, we denote the forward conditional distribution as q. Then, the reverse posterior can be derived using Bayes' rule:

$$q(\mathbf{x}(t-\Delta t)|\mathbf{x}(t),\mathbf{x}(0),\mathbf{c}) \quad = \quad \frac{q(\mathbf{x}(t)|\mathbf{x}(t-\Delta t),\mathbf{c})\, q(\mathbf{x}(t-\Delta t)|\mathbf{x}(0),\mathbf{c})}{q(\mathbf{x}(t)|\mathbf{x}(0),\mathbf{c})}. \tag{116}$$

Here, the probability distributions over the sequence factorize into independent per-token transitions, e.g.,

$$q(\mathbf{x}(t)|\mathbf{x}(0),\mathbf{c}) \quad = \quad \prod_{i=1}^{n} q(x_i(t)|x_i(0),\mathbf{c}), \tag{117}$$

where each probability $q(x_i(t)|x_i(0),\mathbf{c})$ is determined by the single-token generator $\mathbf{Q}^{\text{tok}}(t)$[14]. In practice, for structured $\mathbf{Q}^{\text{tok}}(t)$ matrices like uniform and masking diffusion, the forward transition probability $q(x(t)|x(0),\mathbf{c})$ has a simple analytical closed-form solution. For uniform diffusion, we have

$$q(x(t)|x(0),\mathbf{c}) \quad = \quad \alpha_t \cdot \mathbb{I}(x(t)=x(0)) + (1-\alpha_t)\cdot\frac{1}{|V|}, \tag{118}$$

where α_t is a time-dependent scalar schedule parameter that controls the signal retention rate. Similarly, for the absorbing state (masking) diffusion, the probability mass is directed towards a special `[MASK]` token. The corresponding closed-form transition probability is given by

$$q(x(t)|x(0),\mathbf{c}) \quad = \quad \begin{cases} \alpha_t, & \text{if } x(t)=x(0), \\ 1-\alpha_t, & \text{if } x(t)=\texttt{[MASK]}, \\ 0, & \text{otherwise.} \end{cases} \tag{119}$$

These analytical solutions allow us to bypass the computationally expensive matrix operations in the forward process. With these tractable forward probabilities, the backward process is realized by substituting the neural network's prediction of the original token sequence, $p_\theta(\mathbf{x}(0)|\mathbf{x}(t),\mathbf{c})$, into the Bayesian factorization derived above. This yields a reverse transition distribution $p_\theta(\mathbf{x}(t-1)|\mathbf{x}(t),\mathbf{c})$ from which we can sample. Figure 20 illustrates the forward and backward processes in the D3PM with uniform diffusion, assuming $\Delta t = 1$.

To train the model, we wish to maximize the log-likelihood of the observed data $\log p_\theta(\mathbf{x}(0)|\mathbf{c})$. However, this requires marginalizing over all possible trajectories $\mathbf{x}(1{:}T)$, which is intractable. By

14. To be precise, $q(x_i(t)|x_i(0),\mathbf{c})$ corresponds to the entry $(x_i(0),x_i(t))$ of the single-token transition probability matrix $\mathbf{P}^{\text{tok}}(0,t) = \exp\left(\int_0^t \mathbf{Q}^{\text{tok}}(\tau)\,d\tau\right)$ (assuming commutativity over time). Physically, $\mathbf{P}^{\text{tok}}(0,t)$ represents the total probability mass transferred between tokens over the interval $[0,t]$. Due to the Markov property, this matrix can also be viewed as the product of transition matrices over consecutive discrete time steps: if we discretize the interval into $0 = t_0 < t_1 < \cdots < t_k = t$, then $\mathbf{P}^{\text{tok}}(0,t) = \mathbf{P}^{\text{tok}}(t_{k-1},t_k) \times \cdots \times \mathbf{P}^{\text{tok}}(t_0,t_1)$.

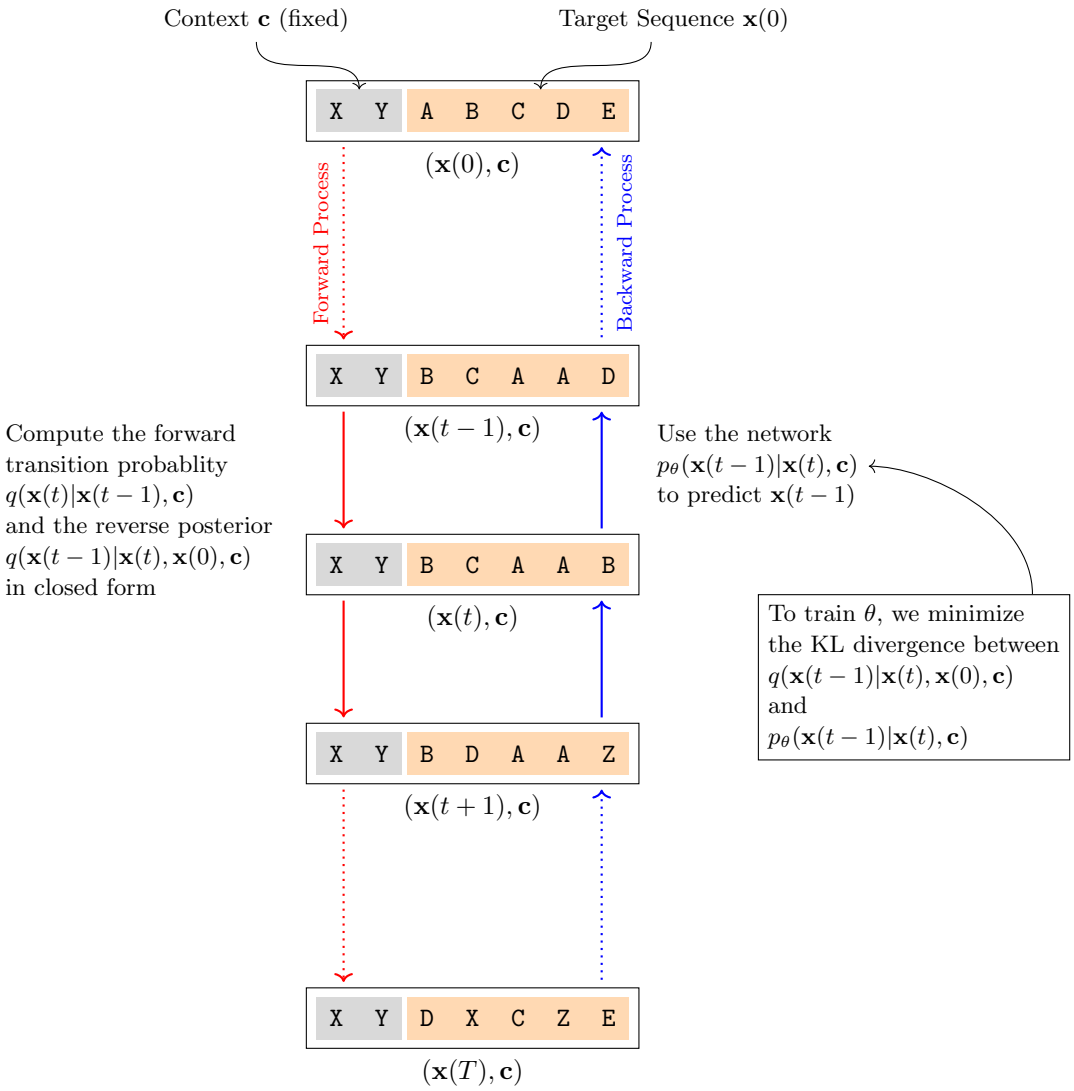


Figure 20: Illustration of the forward and backward processes in the D3PM with uniform diffusion. During the forward process, a state jumps to a new state at each time step of $\Delta t = 1$, with the initial state $(\mathbf{x}(0), \mathbf{c})$ and the final state $(\mathbf{x}(T), \mathbf{c})$. In a forward step, we compute the forward transition probability $q(\mathbf{x}(t)|\mathbf{x}(t-1), \mathbf{c})$, which can then be used to compute the reverse posterior $q(\mathbf{x}(t-1)|\mathbf{x}(t), \mathbf{x}(0), \mathbf{c})$ for training the backward model. During the backward process, the model evolves in reverse time. At each backward step, we use the network $p_\theta(\mathbf{x}(t-1)|\mathbf{x}(t), \mathbf{c})$ to predict the token sequence $\mathbf{x}(t-1)$, and then move on to $t-1$. The network is trained to minimize the KL divergence between $q(\mathbf{x}(t-1)|\mathbf{x}(t), \mathbf{x}(0), \mathbf{c})$ and $p_\theta(\mathbf{x}(t-1)|\mathbf{x}(t), \mathbf{c})$ in expectation over q.

introducing the forward process $q(\mathbf{x}(1{:}T)|\mathbf{x}(0), \mathbf{c})$ as an auxiliary distribution, we can rewrite the

log marginal probability as an expectation

$$\begin{aligned}\log p_\theta(\mathbf{x}(0)|\mathbf{c}) &= \log \sum_{\mathbf{x}(1:T)} q(\mathbf{x}(1{:}T)|\mathbf{x}(0),\mathbf{c}) \frac{p_\theta(\mathbf{x}(0{:}T)|\mathbf{c})}{q(\mathbf{x}(1{:}T)|\mathbf{x}(0),\mathbf{c})} \\ &= \log \mathbb{E}_{q(\mathbf{x}(1:T)|\mathbf{x}(0),\mathbf{c})}\left[\frac{p_\theta(\mathbf{x}(0{:}T)|\mathbf{c})}{q(\mathbf{x}(1{:}T)|\mathbf{x}(0),\mathbf{c})}\right]. \end{aligned} \tag{120}$$

Applying Jensen's inequality, we move the logarithm inside the expectation to derive a tractable **evidence lower bound** (ELBO):

$$\log p_\theta(\mathbf{x}(0)|\mathbf{c}) \geq \mathbb{E}_{q(\mathbf{x}(1:T)|\mathbf{x}(0),\mathbf{c})}\left[\log \frac{p_\theta(\mathbf{x}(0{:}T)|\mathbf{c})}{q(\mathbf{x}(1{:}T)|\mathbf{x}(0),\mathbf{c})}\right]. \tag{121}$$

By expanding both the joint distribution $p_\theta(\mathbf{x}(0{:}T)|\mathbf{c})$ and the forward trajectory $q(\mathbf{x}(1{:}T)|\mathbf{x}(0),\mathbf{c})$ into products of their respective conditional distributions, and then rearranging terms, the negative ELBO can be decomposed into a sum of per-step KL divergence terms (see [Austin et al., 2021] for the full derivation in the discrete setting). The resulting training objective, which the model minimizes, is given by

$$\begin{aligned}\mathcal{L}_{\text{ELBO}} = \mathbb{E}_{q(\mathbf{x}(1:T)|\mathbf{x}(0),\mathbf{c})}\Bigg[&- \log p(\mathbf{x}(T)) + \\ &\sum_{t=1}^{T} \text{KL}\Big(q(\mathbf{x}(t-1)|\mathbf{x}(t),\mathbf{x}(0),\mathbf{c}) \,\|\, p_\theta(\mathbf{x}(t-1)|\mathbf{x}(t),\mathbf{c})\Big)\Bigg], \end{aligned} \tag{122}$$

In this objective, the term $q(\mathbf{x}(t-1)|\mathbf{x}(t),\mathbf{x}(0),\mathbf{c})$ represents the true posterior of the reverse transition. A key advantage here is that, for structured forward processes (like uniform noise or masking), this posterior distribution can be efficiently computed in closed form using Bayes' rule. Regarding the first term, since the prior $p(\mathbf{x}(T))$ is typically a fixed distribution (e.g., a uniform distribution or a deterministic masked state) with no learnable parameters, $-\log p(\mathbf{x}(T))$ is constant with respect to θ and can be omitted. As a result, the core learning signal drives the model p_θ to align its reverse transition probabilities with the ideal reverse path dictated by the forward dynamics.

5.3.2 Masked Diffusion Models

Masked diffusion models (MDMs) represent a specialized implementation of discrete diffusion where the transition rate matrix $\mathbf{Q}^{\text{tok}}(t)$ is designed to move probability mass toward a special absorbing state, denoted as the `[MASK]` token [Sahoo et al., 2024; Nie et al., 2025]. This approach provides a unique practical advantage: once a token transitions to the masked state in the forward process, it remains there, and thus creates a unidirectional path toward a fully corrupted state.

The forward process progressively replaces tokens in the original sequence with `[MASK]`. It starts from a fully observed sequence and gradually destroys information until the entire sequence becomes a mask-only sequence. In practice, we discretize time and implement the process as

a discrete-time Markov chain on discrete time steps $\{0, 1, \ldots, T\}$. Let $\mathbf{x}(t)$ be the corrupted sequence at time $t \in \{0, 1, \ldots, T\}$ with $\mathbf{x}(0)$ being clean data and $\mathbf{x}(T) = \texttt{[MASK]}^n$, and $x_i(t)$ be the i-th token of $\mathbf{x}(t)$. Under the standard independence assumption across positions in the forward process, masking diffusion can be written as a per-token absorbing transition

$$q(\mathbf{x}(t)|\mathbf{x}(0), \mathbf{c}) \quad = \quad \prod_{i=1}^{n} q(x_i(t)|x_i(0), \mathbf{c}), \tag{123}$$

where the single-token forward kernel is

$$q(x_i(t)|x_i(0), \mathbf{c}) \quad = \quad \alpha_t \cdot \mathbb{I}(x_i(t) = x_i(0)) + (1 - \alpha_t) \cdot \mathbb{I}(x_i(t) = \texttt{[MASK]}). \tag{124}$$

Here $\alpha_t \in [0, 1]$ is a monotonically decreasing schedule that controls how much signal is retained at time t. Equivalently, we may view $\mathbf{x}(t)$ as produced by sampling a binary mask indicator vector $\mathbf{m}_t \in \{0, 1\}^n$ with

$$m_{t,i} \sim \text{Bernoulli}(1 - \alpha_t), \tag{125}$$

$$x_i(t) = \begin{cases} \texttt{[MASK]}, & m_{t,i} = 1, \\ x_i(0), & m_{t,i} = 0. \end{cases} \tag{126}$$

Thus the diffusion trajectory can be interpreted as repeatedly increasing the set of masked positions until eventually all positions are masked, as illustrated in Figure 21.

The backward process runs from $t = T$ to $t = 0$. Given the context $\mathbf{c}$ and an initial fully masked sequence $\mathbf{x}(T) = \texttt{[MASK]}^n$, the MDM iteratively replaces [MASK] tokens with real tokens until obtaining a fully instantiated sequence in V^n. Formally, we define a parameterized reverse Markov chain

$$p_\theta(\mathbf{x}(t-1)|\mathbf{x}(t), \mathbf{c}), \quad t = T, T-1, \ldots, 1, \tag{127}$$

Here $p_\theta(\mathbf{x}(t-1)|\mathbf{x}(t), \mathbf{c})$ is the notation of the reverse conditional distribution $\Pr_\theta(\mathbf{x}(t-1)|\mathbf{x}(t), \mathbf{c})$, where the subscript θ emphasizes that the reverse-time dynamics are parameterized by θ.

A simple parameterization is to let the network predict the denoising distribution over the clean tokens,

$$p_\theta(\mathbf{x}(0)|\mathbf{x}(t), \mathbf{c}) = \prod_{i=1}^{n} p_\theta(x_i(0)|\mathbf{x}(t), \mathbf{c}). \tag{128}$$

Then, we construct the reverse transition by only updating masked positions, while keeping unmasked tokens unchanged. For each position i, we have

$$p_\theta(x_i(t-1)|\mathbf{x}(t), \mathbf{c}) = \begin{cases} \mathbb{I}(x_i(t-1) = x_i(t)), & \text{if } x_i(t) \neq \texttt{[MASK]}, \\ \pi_{\theta,t,i}(\cdot|\mathbf{x}(t), \mathbf{c}), & \text{if } x_i(t) = \texttt{[MASK]}, \end{cases} \tag{129}$$

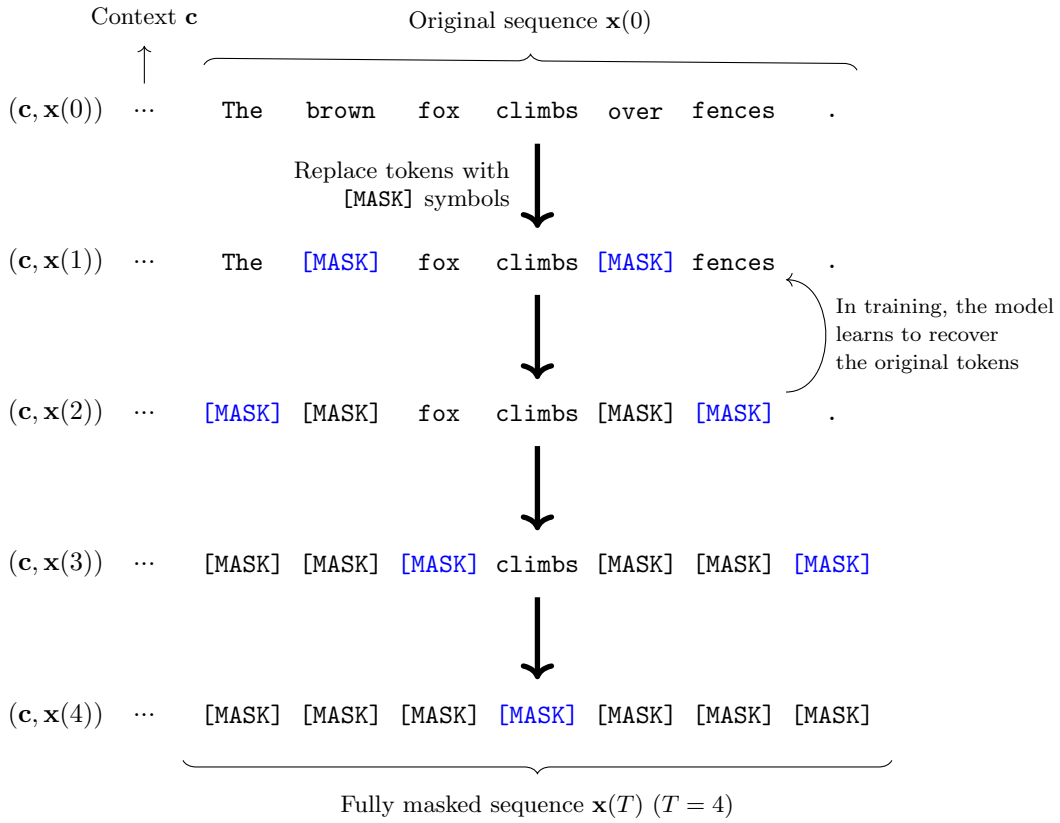


Figure 21: Illustration of the forward process in MDMs, where an original token sequence is iteratively transformed into a fully masked sequence. During this process, tokens are replaced with [MASK] symbols until the entire sequence is masked. Note that the context $\mathbf{c}$ remains unmasked throughout. To train an MDM, the model learns to recover the masked tokens by conditioning on both the fixed context $\mathbf{c}$ and the partially corrupted sequence.

where $\pi_{\theta,t,i}(\cdot|\mathbf{x}(t),\mathbf{c})$ is a categorical distribution on V produced by the model. A common choice is

$$\pi_{\theta,t,i}(v|\mathbf{x}(t),\mathbf{c}) = p_\theta(x_i(0) = v|\mathbf{x}(t),\mathbf{c}), \quad v \in V, \tag{130}$$

i.e., directly sample the missing token from the predicted clean-token distribution.

Given a dataset of examples $(\mathbf{x}(0),\mathbf{c}) \sim p_{\text{data}}$, MDMs learn the reverse model p_θ by maximizing the conditional likelihood of the clean sequence, or equivalently minimizing its negative log-likelihood. Similar to other diffusion models, we can train MDMs using the ELBO (see also D3PM in Section 5.3.1).

Although the above training objective appears a bit tedious, in practice, a simpler and widely used approach is to train the model to recover the original tokens at masked positions. We sample a time step $t \sim U\{1,\dots,T\}$ and then sample a corrupted sequence $\mathbf{x}(t) \sim q(\mathbf{x}(t)|\mathbf{x}(0),\mathbf{c})$. We

define the masked index set to be

$$\mathcal{M}(t) \quad = \quad \{i \in \{1, \ldots, n\} : x_i(t) = \texttt{[MASK]}\}. \tag{131}$$

The loss is then defined as the conditional negative log-likelihood on masked positions, as follows

$$\begin{aligned} & \mathcal{L}_{\text{denoise}}(\theta) \\ = \quad & \mathbb{E}_{(\mathbf{x}(0),\mathbf{c})\sim p_{\text{data}}} \, \mathbb{E}_{t\sim U[T]} \, \mathbb{E}_{\mathbf{x}(t)\sim q(\mathbf{x}(t)|\mathbf{x}(0),\mathbf{c})} \left[- \sum_{i\in\mathcal{M}(t)} \log p_\theta(x_i(0)|\mathbf{x}(t),\mathbf{c}) \right]. \end{aligned} \tag{132}$$

Optionally, one can reweight time steps to emphasize certain noise levels:

$$\mathcal{L}_{\text{denoise}}(\theta) \quad = \quad \mathbb{E}\left[\gamma_t \cdot \Big(- \sum_{i\in\mathcal{M}(t)} \log p_\theta(x_i(0)|\mathbf{x}(t),\mathbf{c})\Big)\right], \tag{133}$$

where $\gamma_t \geq 0$ is a predefined schedule. Intuitively, as in curriculum learning, small t corresponds to light corruption (i.e., easier denoising), while large t corresponds to heavy corruption (i.e., harder denoising). The weighting γ_t can be tuned to balance these regimes.

After training, we use the learned reverse kernel $p_\theta(\mathbf{x}(t-1)|\mathbf{x}(t),\mathbf{c})$ to generate text by running the Markov chain backward from a fully masked state. The inference starts from

$$\mathbf{x}(T) = \texttt{[MASK]}^n, \tag{134}$$

and iteratively samples

$$\mathbf{x}(t-1) \sim p_\theta(\mathbf{x}(t-1)|\mathbf{x}(t),\mathbf{c}), \quad t = T, T-1, \ldots, 1. \tag{135}$$

Under the per-token factorization in Eq. (129), the reverse step only updates masked positions. For each i, we have

$$x_i(t-1) = \begin{cases} x_i(t), & x_i(t) \neq \texttt{[MASK]}, \\ \text{Cat}(\pi_{\theta,t,i}(\cdot|\mathbf{x}(t),\mathbf{c})), & x_i(t) = \texttt{[MASK]}, \end{cases} \tag{136}$$

where $\text{Cat}(\cdot)$ denotes sampling a discrete token from the categorical distribution specified by its argument, i.e., $x_i(t-1) \sim \pi_{\theta,t,i}(\cdot|\mathbf{x}(t),\mathbf{c})$, when $x_i(t) = \texttt{[MASK]}$. Thus, the model repeatedly fills in missing tokens conditioned on the already-filled context, until all positions become unmasked.

The above approach does not specify how many tokens should be unmasked per step. In practice, MDMs typically adopt an explicit unmasking schedule. Let k_t be the number of tokens to be newly unmasked when going from t to $t-1$, and let $\mathcal{U}(t) \subseteq \mathcal{M}(t)$ be the selected subset of masked positions to update, with $|\mathcal{U}(t)| = k_t$. Then a common implementation is

$$x_i(t-1) = \begin{cases} x_i(t), & i \notin \mathcal{U}(t), \\ \text{Cat}(\pi_{\theta,t,i}(\cdot|\mathbf{x}(t),\mathbf{c})), & i \in \mathcal{U}(t). \end{cases} \tag{137}$$

The schedule $\{k_t\}_{t=1}^{T}$ trades off speed and accuracy: larger k_t leads to fewer iterations but requires the model to fill many tokens under high uncertainty, whereas smaller k_t yields more refinement steps but increases compute.

A simple choice is a deterministic schedule that reveals a fixed fraction each step (e.g., linear or cosine in t), such that the expected number of masked tokens decreases from n to 0. Another common strategy is to pick $\mathcal{U}(t)$ based on model confidence. Given $\pi_{\theta,t,i}$, we can define a confidence score as

$$s_{t,i} = \max_{v \in V} \ \pi_{\theta,t,i}(v|\mathbf{x}(t), \mathbf{c}) \,, \tag{138}$$

Then we can unmask positions with the highest confidence first. This easy-first heuristic tends to stabilize generation by anchoring the sequence with reliable tokens before filling more ambiguous ones.

5.4 Remarks on Discrete Diffusion

In this subsection, we discuss several modeling considerations for discrete diffusion in token space.

5.4.1 Rethinking Non-absorbing Replacement Diffusion

The absorbing `[MASK]` construction has an appealing information monotonicity: the forward process only removes information, and the reverse process only fills it back in. This design aligns well with Transformer denoising and has recently demonstrated strong scalability in large-scale pretraining of diffusion language models [Sahoo et al., 2024; Nie et al., 2025]. Nevertheless, it is valuable to revisit non-absorbing replacement diffusion methods (such as uniform diffusion), where tokens may be repeatedly substituted throughout the trajectory.

First, uniform replacement is a simple and well-behaved baseline. The uniform substitution kernel used in D3PM is the canonical example. At each step, a token is either kept or replaced by a uniformly sampled vocabulary element. This strategy yields closed-form transition probabilities, thereby keeping training tractable. Moreover, because the corruption is reversible in principle, the reverse process is not constrained by an irreversible absorbing sink. Conceptually, uniform replacement is the discrete analog of adding isotropic Gaussian noise. It drives the distribution toward a simple prior while preserving analytical control of the forward marginals.

Second, the reverse updates are dense in non-absorbing replacement diffusion. In masking diffusion, unmasked tokens are typically preserved and only masked positions are updated. In non-absorbing replacement diffusion, every position remains stochastic, so a natural reverse kernel can update all tokens at each step. This resembles the structure of continuous-time diffusion models, where all dimensions are refined throughout the trajectory. Rather than treating visible coordinates as fixed, the dynamics can revise any component to improve global consistency.

Third, non-absorbing replacement can be difficult in language. If tokens can be corrupted multiple times, the noisy sequence $\mathbf{x}(t)$ may contain locally fluent but globally inconsistent fragments, and the reverse model must both infer missing semantics and correct erroneous tokens. This is in contrast to masking diffusion, where generated tokens are typically clean. From the

CTMC viewpoint, non-absorbing replacement corresponds to a generator $\mathbf{Q}^{\text{tok}}(t)$ with nonzero off-diagonal rates between all token pairs, so that probability mass continuously circulates on V instead of flowing into a single sink. As t increases, the signal-to-noise ratio can degrade faster than in absorbing diffusion, because even observed tokens are unreliable.

Recent work has explored methods of combining absorbing masking with additional replacement noise to induce a corrective behavior. In this way, the model can detect and fix corrupted tokens rather than only fill masked positions [Zhang et al., 2025a; Rütte et al., 2025]. This hybrid view suggests that non-absorbing noise is not merely a competitor to masking diffusion, but can be a complementary mechanism for robustness and self-correction.

A simple idea is to consider structured replacement strategies. In NLP, a natural method is to bias replacement using token-dependent or structure-aware transition rates, for example,

$$Q^{\text{tok}}_{uv}(t) \quad = \quad w_{uv}(t), \tag{139}$$

where $w_{uv}(t)$ models confusability which could be frequency-aware, subword-related, or embedding-similarity-based. The motivation is to make early noise semantically local (easy to denoise) and late noise globally mixing (approaching a simple prior). This is similar to variance-preserving vs. variance-exploding design choices in continuous diffusion, but now expressed as a schedule over discrete transition structure.

Another interesting direction is to interpolate between masking-style absorbing diffusion and uniform replacement diffusion. For example, one may consider a combination at the generator level:

$$\mathbf{Q}^{\text{tok}}(t) = \lambda_t \cdot \mathbf{Q}^{\text{tok}}_{\text{mask}}(t) + (1 - \lambda_t) \cdot \mathbf{Q}^{\text{tok}}_{\text{unif}}(t), \tag{140}$$

where the model behaves like masking diffusion at certain noise levels while retaining the ability to distribute probability mass among unmasked tokens at others. Such a combination has recently been explored as a way to obtain the best of both regimes (stability from absorbing corruption and flexibility from replacement corruption).

5.4.2 Masked Diffusion Modeling vs Masked Language Modeling

In the general framework of corruption-and-denoise, the training objective of masked diffusion models is very close to that of **masked language modeling** (MLM). For example, in BERT-like models, one samples a mask pattern and trains a bidirectional Transformer to predict the original tokens at masked positions from partially observed context [Devlin et al., 2019]. If we interpret Eq. (126) as sampling a mask indicator $\mathbf{m}_t$ with mask rate $(1 - \alpha_t)$, then the denoising loss in Eq. (132) is essentially an MLM cross-entropy, except that the corruption level is explicitly indexed by time through α_t and the network is conditioned on the diffusion step t. The main conceptual difference is therefore not the supervised signal but the deployment of the denoiser. BERT's MLM is primarily a pretraining objective for representation learning, whereas masked diffusion models interpret the same denoising primitive as a reverse-time transition kernel and apply it iteratively from a fully masked state to generate a natural token sequence. In this sense,

modern masked diffusion models often look like BERT in their architectures, but they turn MLM from a pretraining task into a procedure for sequence generation.

The connection between masked diffusion models and MLM suggests an interesting direction: since the forward process in masked diffusion is a particular choice of corruption, many noising strategies developed for NLP pretraining can be reinterpreted as alternative forward kernels for discrete diffusion. That is, beyond independent token masking, one can vary what gets corrupted (tokens vs. spans), how it is corrupted (masking vs. replacement vs. permutation), and how the corruption level evolves with time (hand-designed vs. learned schedules). Below are some common noising strategies that can be considered in designing masked diffusion models.

- **Token replacement with non-uniform distributions.** Instead of replacing tokens uniformly, one can sample replacements from frequency-adjusted distributions or from a learned proposal. An example is ELECTRA, where a small generator proposes plausible replacements and a discriminator is trained to detect them [Clark et al., 2019]. From the diffusion perspective, such harder non-uniform replacements can be viewed as injecting more semantically confusable noise than random tokens, which may encourage stronger conditional denoisers and reduce the gap between training corruptions and inference-time uncertainty.

- **Span corruption.** The T5 series models replace contiguous spans with a single sentinel token and learn to reconstruct the missing content [Raffel et al., 2020]. For diffusion-style generation, span corruption is natural: the reverse process can be interpreted as progressively refining coarse missing regions into detailed text. This matches the intuition of iterative denoising over structured slots rather than independent token blanks. However, a challenge with this approach is that replacing multiple tokens with a single sentinel alters the sequence length, which violates the fixed-dimension assumption of standard diffusion models. To address this, we can adopt in-place span masking similar to SpanBERT [Joshi et al., 2020], where contiguous tokens are masked but the sequence length remains constant. Alternatively, one can extend the forward kernel to explicitly model insertion and deletion operations to handle variable-length trajectories [Gu et al., 2019; Reid et al., 2022].

- **Token shuffling and permutation noise.** BART introduces denoising pretraining with corruptions such as token deletion and sentence permutation. This method requires the model to learn not only content restoration but also reordering [Lewis et al., 2020]. Recasting this as a diffusion forward kernel suggests a broader class of discrete dynamics that perturb both identity and position information. Such a method can lead to reverse processes that perform joint "edit + reorder" refinement rather than pure infilling.

- **Adversarial noising.** Another direction is to generate more challenging corruptions using model-guided perturbations or auxiliary networks. Adversarial training methods for NLP (e.g., embedding-level perturbations) show that stronger, structured noise can improve robustness [Zhu et al., 2020]. While many adversarial approaches are defined in continuous embedding space, the core idea is applicable to diffusion modeling. We can choose corruptions that maximally stress the denoiser at each noise level, which in principle can make the learned reverse dynamics more stable under iterative sampling.

- **Learned noising schedules.** Rather than fixing the schedule α_t by hand, one can attempt to learn how much and what type of noise to apply at each step, for example, by optimizing a combination of objectives or selecting corruption regimes that best serve downstream use [Tay et al., 2023]. In diffusion modeling, this motivates learning time-inhomogeneous corruption policies that shape the trajectory for better sample quality or fewer generation steps.

5.4.3 Simplex and Logit Dynamics as Continuous Relaxations

We have seen that the CTMC formulation frames discrete diffusion as distributional dynamics. Although the sample path $\mathbf{x}(t) \in V^n$ jumps among categorical states, the marginal $p_t(\cdot)$ evolves smoothly over t through the Kolmogorov equation in Eq. (114). This observation motivates a useful relaxation for language. Instead of treating the intermediate state as a hard token sequence, we represent each position by a probability column vector on the simplex,

$$\mathbf{p}_i(t) \in \Delta^{|V|-1}, \tag{141}$$

$$\mathbf{p}_i(t)[v] = \Pr(x_i(t) = v \mid \mathbf{c}). \tag{142}$$

We then aggregate these vectors into a matrix $\mathbf{P}(t) = [\mathbf{p}_1(t), \ldots, \mathbf{p}_n(t)] \in \mathbb{R}^{|V|\times n}$, where each column lies on the simplex. This converts the discrete trajectory into a continuous curve on a product of simplices. Then, we can apply ODE/SDE tools while still respecting the categorical nature of tokens.

For a token-level CTMC with generator $\mathbf{Q}^{\text{tok}}(t) \in \mathbb{R}^{|V|\times|V|}$, the marginal distribution of a single position satisfies the Kolmogorov Forward equation

$$\frac{d\mathbf{p}_i(t)}{dt} = \left[\mathbf{Q}^{\text{tok}}(t)\right]^\top \mathbf{p}_i(t), \tag{143}$$

which is an ODE on the simplex. Note that the rows of $\mathbf{Q}^{\text{tok}}(t)$ sum to zero, so the total probability is conserved. Under the standard independence assumption across positions in the noising process, each $\mathbf{p}_i(t)$ evolves according to the same ODE. This gives an ODE viewpoint in language: discrete diffusion can be seen as choosing a linear forward vector field on the simplex, induced by $\mathbf{Q}^{\text{tok}}(t)$, rather than a Gaussian SDE in $\mathbb{R}^d$.

Working directly with probabilities can be numerically inconvenient near the boundary of the simplex (e.g., one-hot corners). A common alternative is to parameterize $\mathbf{p}_i(t)$ by logits $\mathbf{z}_i(t) \in \mathbb{R}^{|V|}$

$$\mathbf{p}_i(t) = \text{Softmax}(\mathbf{z}_i(t)). \tag{144}$$

We can then define a continuous-time dynamics in logit space,

$$\frac{d\mathbf{z}_i(t)}{dt} = \mathbf{v}_\theta\big(\mathbf{z}_i(t), t, \mathbf{c}\big), \tag{145}$$

where $\mathbf{v}_\theta$ is a Transformer-defined vector field (shared across positions). Intuitively, $\mathbf{z}_i(t)$ behaves like a continuously refined "soft token", and the terminal decoding $\mathbf{z}_i(0) \mapsto x_i$ can be implemented by arg max from $\mathrm{Softmax}(\mathbf{z}_i(0))$. Relaxations of categorical sampling such as the Gumbel-Softmax distribution can also be used when differentiable sampling is desired [Jang et al., 2017; Maddison et al., 2017].

This logit view clarifies why continuous relaxations often feel similar to embedding-space diffusion, yet remain closer to discrete modeling: the state stays vocabulary-aligned (dimension $|V|$ per position) rather than living in an arbitrary embedding space $\mathbb{R}^d$. It also offers a direct connection to the CTMC marginal ODE in Eq. (143). A chosen $\mathbf{Q}^{\mathrm{tok}}(t)$ implies a specific evolution of $\mathbf{p}_i(t)$, which can be re-expressed as an evolution in $\mathbf{z}_i(t)$ via the Softmax map.

As presented in Section 4, in continuous diffusion, the reverse-time dynamics can be expressed either as an SDE involving the score $\nabla \log p_t(\mathbf{s})$ or as a probability-flow ODE. In discrete diffusion, the exact reverse CTMC rates depend on probability ratios $p_t(u)/p_t(v)$ (Eq. (115)), which play the role of a discrete score. When we lift the state to $\mathbf{p}(t)$ or $\mathbf{z}(t)$, we can similarly interpret the reverse process as learning a vector field that flows from a simple prior (e.g., all-masked or uniform) to a distribution concentrated near one-hot corners representing real text. Recent discrete score-based formulations show that one can define tractable training objectives that avoid explicitly computing $p_t(\cdot)$ while still producing principled reverse-time dynamics [Hoogeboom et al., 2021; Lou et al., 2024].

To summarize, we outline the practical benefits of the simplex and logit relaxations, as follows

- **More efficient ODE solvers and fewer steps.** Once the reverse dynamics are expressed as an ODE like Eq. (145), we can use adaptive ODE solvers or coarse discretizations to reduce the number of sampling steps, echoing the acceleration strategies used in continuous diffusion and flow models (Section 4.4).
- **Compatibility with flow matching.** On the simplex, flow matching naturally corresponds to learning $\mathbf{v}_\theta$ that transports soft token distributions toward sharp, data-like categorical distributions, without requiring an explicit likelihood-based ELBO at training time.
- **A spectrum between discrete and continuous.** Masked diffusion (hard `[MASK]` states) and embedding diffusion (continuous embeddings) can be seen as two endpoints. Simplex/logit dynamics occupy a middle ground. The state is continuous but remains directly interpretable as token probabilities, and the final projection to text is well-defined.

5.4.4 Relation to Iterative Refinement

The reverse process of discrete diffusion is closely related to a line of research on **iterative refinement** for non-autoregressive generation [Lee et al., 2018]. In iterative refinement, the model does not obtain a final sequence in one pass. Instead, it repeatedly generates a full sequence in parallel and then revises parts of it over multiple refinement rounds. An example is the mask-predict method for conditional masked language models, which starts from an all-`[MASK]` template and alternates between parallel prediction of tokens and re-masking a subset of uncertain positions, until convergence or a fixed number of iterations [Ghazvininejad et al., 2019].

This method can be expressed in a form that is similar to Eq. (137). Let $\mathbf{x}^{(r)}$ denote the sequence after r refinement rounds and let $\pi_{\theta,i}^{(r)}(v|\mathbf{x}^{(r)}, \mathbf{c}) = p_\theta(x_i^{(r+1)} = v|\mathbf{x}^{(r)}, \mathbf{c})$ be the predicted distribution of round r at position i. A refinement step can be expressed as

$$x_i^{(r+1)} = \begin{cases} x_i^{(r)}, & i \notin \mathcal{U}^{(r)}, \\ \mathrm{Cat}\left(\pi_{\theta,i}^{(r)}(\cdot|\mathbf{x}^{(r)}, \mathbf{c})\right), & i \in \mathcal{U}^{(r)}, \end{cases} \tag{146}$$

where $\mathcal{U}^{(r)}$ is the set of positions to update at round r. Typically, $\mathcal{U}^{(r)}$ is defined by low confidence, while in simple diffusion implementations it is often defined by a time schedule. In other words, masked diffusion sampling can be viewed as a structured refinement procedure in which the masking pattern evolves monotonically with time t (i.e., iteration index in iterative refinement), whereas classic refinement methods allow non-monotone decisions (e.g., re-masking already filled tokens) based on the uncertainty of the model. This difference matters: monotone unmasking is stable but may freeze early mistakes. By contrast, non-monotone refinement can correct errors but risks oscillation if the selection rule is not well defined.

Recent diffusion language models have begun to explicitly incorporate the non-monotone correction behavior that is standard in iterative refinement. One approach is to re-mask low-confidence tokens during inference, and combine diffusion sampling with mask-predict-like revisiting [Koh et al., 2025; Zhang et al., 2025a]. Another approach is to modify the forward corruption so that the reverse model must learn not only to fill blanks but also to fix visible but wrong tokens. This can be achieved by mixing absorbing masking noise with replacement noise (e.g., uniform replacement), which yields a reverse process that naturally supports correction and editing [Rütte et al., 2025]. These approaches overlap those presented in Section 5.4.1, and echo our discussion of combining different diffusion strategies.

We can also see iterative refinement from the perspective of diffusion modeling. The simplex and logit relaxations in Eqs. (143)–(145) provide a differential-equation view on iterative refinement. If we interpret each refinement step as a small update of a soft token state, then iterative refinement becomes an explicit numerical discretization of a continuous-time transport toward sharp, near-one-hot distributions. This connects to recent discrete flow matching formulations, which aim to learn continuous-time flows that can be sampled in few refinement steps while maintaining quality [Gat et al., 2024; Monsefi et al., 2025]. In this sense, diffusion modeling, flow-based modeling, and classical iterative refinement can be viewed as variants on the same theme. They differ mainly in how the intermediate state is represented (hard tokens vs. soft distributions), how the update set $\mathcal{U}$ is selected (schedule vs. confidence), and whether the dynamics are framed as a Markov process or a general learned refinement operator.

5.4.5 Diffusion Transformers

A practical reason that discrete diffusion has become competitive in NLP is that the reverse dynamics can be parameterized by Transformer. This architecture is the most scalable backbone that dominates language modeling. The term *diffusion transformer* (DiT) is most widely used in vision, where replacing the U-Net with a Transformer backbone yields strong scaling behavior

[Peebles and Xie, 2023]. Presenting the details of Transformers is beyond the scope of this paper. We refer interested readers to related papers [Vaswani et al., 2017; Xiao and Zhu, 2023].

In NLP, Transformers provide a powerful mechanism for encoding a token sequence into a sequence of real-valued representations of equal length. Depending on how these representations are defined and interpreted, such models can be employed to learn arbitrary mappings from input token sequences. In diffusion modeling, Transformers must be conditioned on the diffusion time t (or a discrete index). This is typically done by injecting a learned time embedding into every layer, e.g., via additive bias, or cross-attention to a time token. The network output can be interpreted in several ways, depending on the chosen parameterization. In continuous-space diffusion, common choices include predicting the score $\nabla_{\mathbf{s}} \log p_t(\mathbf{s})$, the additive noise $\boldsymbol{\epsilon}$, the denoised data $\mathbf{s}(0)$, or a velocity variable that linearly combines the above [Ho et al., 2020a; Salimans and Ho, 2022]. These parameterizations induce different discretizations and stability properties, but all can be essentially viewed as learning a time-indexed vector field that transports probability mass from a simple base distribution to the data distribution.

5.4.6 Flow-based Views

Given the modeling flexibility offered by Transformers, we need not commit to an explicit likelihood-based ELBO (see Eq. (122)) to train diffusion models. A natural alternative is flow matching. In the continuous setting, flow matching learns a time-dependent vector field whose ODE transports a simple base distribution to the data distribution. For language, the key difficulty is that the state is defined on the discrete product space V^n, so the flow must be defined either as a dynamics of probability mass over tokens (a discrete-state view), or as an ODE on a continuous relaxation such as the probability simplex or logit space (a continuous-state view). Recent **discrete flow matching** (DFM) methods provide both perspectives and make them practical at the scale of modern language modeling [Gat et al., 2024; Campbell et al., 2024; Shaul et al., 2025].

A useful starting point is the simplex representation already introduced in Eq. (143). Instead of treating $\mathbf{x}(t)$ as a discrete token sequence, we view the intermediate state as per-position categorical probabilities

$$\mathbf{P}(t) \quad = \quad [\mathbf{p}_1(t), \ldots, \mathbf{p}_n(t)]^\top \in (\Delta^{|V|-1})^n, \tag{147}$$

where

$$\sum_{v \in V} \mathbf{p}_i(t)[v] \quad = \quad 1. \tag{148}$$

In this representation, a flow is an ODE on the product-of-simplices,

$$\frac{d\mathbf{p}_i(t)}{dt} \quad = \quad \mathbf{v}_\theta(\mathbf{p}_i(t), t, \mathbf{c}), \tag{149}$$

$$\sum_{v \in V} \mathbf{v}_\theta(\mathbf{p}_i(t), t, \mathbf{c})[v] \quad = \quad 0, \tag{150}$$

where the tangent constraint enforces probability conservation. Sampling is then an ODE integration problem which is similar to that in vision: given $\mathbf{p}(T)$ from a base distribution, integrate Eq. (150) backward to obtain $\mathbf{p}(0)$ and decode tokens by arg max or categorical sampling.

What remains is to specify the training target for $\mathbf{v}_\theta$. Discrete flow matching does this by defining a family of probability paths that interpolate between a source distribution and the data distribution, and then matching the model vector field to the instantaneous probability velocity induced by these paths. More specifically, for each training example $\mathbf{x}(0)$, one defines a conditional path $q_t(\cdot|\mathbf{x}(0), \mathbf{c})$ on V^n or on per-token marginals, and draws a noisy sample $\mathbf{x}(t) \sim q_t(\cdot|\mathbf{x}(0), \mathbf{c})$. The flow-matching regression target is the conditional velocity along that path,

$$\mathbf{v}^\star(\mathbf{x}(t), t, \mathbf{x}(0), \mathbf{c}) \quad = \quad \frac{d\mathbb{E}\left[\mathbf{1}_{\mathbf{x}(t)}|\mathbf{x}(0), \mathbf{c}\right]}{dt}, \tag{151}$$

where $\mathbf{1}_{\mathbf{x}(t)}$ is the one-hot representation for $\mathbf{x}(t)$. The resulting objective is similar to that of continuous conditional flow matching, except that the state and velocity are constrained by the simplex geometry and the categorical nature of the endpoint.

The noising kernels can be very similar to those used in the diffusion language models discussed previously. For each position, we can define a time-dependent mixture between a clean point mass and a base distribution $\bar{p}_i(\cdot)$,

$$q_t(x_i|x_i(0), \mathbf{c}) \quad = \quad \alpha_t \cdot \mathbb{I}(x_i = x_i(0)) + (1 - \alpha_t) \cdot \bar{p}_i(x_i), \tag{152}$$

with a schedule α_t decreasing from 1 to 0. This induces a smooth probability path in the simplex even though sample paths are discrete.

Differentiating Eq. (152) with respect to time yields the analytic form of the target velocity

$$\mathbf{v}_i^\star(x_i(t), t, x_i(0), \mathbf{c}) \quad = \quad \dot{\alpha}_t \cdot \left[\mathbf{1}_{x_i(0)} - \bar{p}_i\right], \tag{153}$$

where $\dot{\alpha}_t = d\alpha_t/dt$ denotes the time derivative of the schedule. This simple linear form implies that the conditional flow directs probability mass straight from the base distribution $\bar{p}_i$ towards the target token $x_i(0)$ at a rate determined by $\dot{\alpha}_t$.

To train the model, we need a conditional velocity that depends only on what is available at time t. As in standard flow matching, the training objective is defined as

$$\mathcal{L}_{\text{DFM}}(\theta) \quad = \quad \mathbb{E}_{t,\mathbf{x}(0),\mathbf{x}(t)}\left[\left\|\mathbf{v}_\theta(\mathbf{x}(t), t, \mathbf{c}) - \mathbf{v}^\star(\mathbf{x}(t), t, \mathbf{x}(0), \mathbf{c})\right\|^2\right]. \tag{154}$$

The simplex-ODE view above describes a continuous curve of marginals, but it does not specify how discrete samples should evolve during generation. A complementary view is to represent discrete flows through a CTMC parameterized by a time-dependent generator matrix $\mathbf{Q}^{\text{tok}}(t) \in \mathbb{R}^{|V|\times|V|}$. The connection between the marginal velocity $\mathbf{v}(t)$ and the generator is given by the Kolmogorov forward equation

$$\frac{d\mathbf{p}_i(t)}{dt} \quad = \quad \left[\mathbf{Q}^{\text{tok}}(t)\right]^\top \mathbf{p}_i(t), \tag{155}$$

where $\mathbf{p}_i(t)$ is a vector representing the probability distribution over tokens. By expanding this matrix multiplication for a specific target token z, we can interpret the velocity as a flux balance equation

$$\mathbf{v}_i(t)[z] \quad = \quad \sum_{y \neq z} \underbrace{\mathbf{p}_i(t)[y] Q^{\text{tok}}_{yz}(t)}_{\text{inflow from } y \to z} - \sum_{y \neq z} \underbrace{\mathbf{p}_i(t)[z] Q^{\text{tok}}_{zy}(t)}_{\text{outflow from } z \to y}. \tag{156}$$

Here, $Q^{\text{tok}}_{yz}(t)$ denotes the instantaneous rate of jumping from token y to z. Note that the diagonal elements satisfy $Q^{\text{tok}}_{yy}(t) = -\sum_{z \neq y} Q^{\text{tok}}_{yz}(t)$ to ensure probability conservation.

While the marginal velocity $\mathbf{v}^\star$ derived in Eq. (153) fixes the net change in probability, it does not uniquely determine the transition rates $\mathbf{Q}^{\text{tok}}$. Discrete flow matching methods resolve this ambiguity by defining a target generator $\mathbf{Q}^{\star,\text{tok}}$ that implements the transport with minimal noise. A common choice in recent works is to define rates that only allow transitions directed towards the target sample $x_i(0)$ [Campbell et al., 2024]. Specifically, the conditional target rates are defined as

$$Q^{\star,\text{tok}}_{yz}(t \mid x_i(0)) \quad = \quad \frac{\dot{\alpha}_t}{1 - \alpha_t} \cdot \mathbb{I}(z = x_i(0)), \quad \text{for } y \neq x_i(0). \tag{157}$$

This generator drives any non-target token y to jump directly to the target $x_i(0)$, while keeping the correct marginal evolution described by $\mathbf{v}^\star$.

As a result, the model is parameterized to predict this generator matrix, and the training objective becomes a rate matching loss rather than the vector regression in Eq. (154). During generation, one can simulate the process using algorithms like Gillespie's method [Gillespie, 1977] or Tau-Leaping [Gillespie, 2001], and perform discrete jumps $x(t) \to x(t + \Delta t)$ that statistically adhere to the learned probability flow.

6. Conclusions and Future Directions

In this paper, we have introduced the fundamental concepts of ODEs and explored their application in modeling both discrete processes, such as neural network architectures, and continuous processes, such as flows. We further extended the discussion to the broader framework of differential equations, including SDEs, focusing on how these mathematical tools define generation processes in vision and language diffusion modeling. Along the way, we addressed practical challenges such as the stiffness of dynamical systems and the complexities of discrete diffusion in token spaces.

While the intersection of differential equations and deep learning has already produced remarkable results, several promising directions remain for future exploration:

- One direction is to consider the issues of scaling and numerical stability. As models scale to billions of parameters, ensuring the numerical stability of the underlying ODEs becomes increasingly difficult. Future research could investigate more robust adaptive solvers and regularization techniques to handle stiff dynamics in large-scale architectures without excessive computational overhead.

- It is also valuable to consider efficient Inference methods. Reducing the number of sampling steps remains a major priority for real-world deployment. Techniques like trajectory rectification, consistency distillation, and specialized high-order solvers offer paths to achieving high-fidelity generation in just one or a few steps.
- Applying continuous modeling to discrete data like text is not straightforward. Exploring native discrete flow matching and more sophisticated continuous relaxations may lead to more effective non-autoregressive language models that rival the quality of their autoregressive counterparts.
- The perspective of dynamical systems provides a unique lens for model interpretability. Analyzing fixed points, attractors, and the geometry of learned trajectories could provide deeper insights into how neural networks work on real-world problems.
- It is natural to extend the dynamical-systems viewpoint to a broader range of learning tasks, not only to the problems we discussed in this paper. For example, viewing optimization itself as a dynamical system may help connect training stability and generalization with classical results in stability and perturbation theory, and may provide a unified way to understand how algorithmic choices (e.g., step size schedules, momentum, or implicit updates) interact with model architecture and data.

Acknowledgements

This work was supported in part by the National Science Foundation of China (Nos. 62276056 and U24A20334), the Yunnan Fundamental Research Projects (No.202401BC070021), the Yunnan Science and Technology Major Project (No. 202502AD080014), the Fundamental Research Funds for the Central Universities (Nos. N25BSS054 and N25BSS094), and the Program of Introducing Talents of Discipline to Universities, Plan 111 (No.B16009). The authors thank Junxiang Zhang and Hengyu Li for their valuable comments, which helped improve the manuscript.

Appendix A. Deriving the Heavy Ball ODE

Recall that stochastic gradient descent with momentum defines the following update rule

$$v_{k+1} = \mu v_k - \eta \cdot \frac{dL(\theta_k)}{d\theta_k} \tag{158}$$

$$\theta_{k+1} = \theta_k + v_{k+1}, \tag{159}$$

where $\mu \in [0, 1)$ is the momentum coefficient and $\eta > 0$ is the learning rate.

To derive the continuous-time limit, we first eliminate the momentum (or velocity) variable v_k. From Eq. (159), we have $v_{k+1} = \theta_{k+1} - \theta_k$, which implies $v_k = \theta_k - \theta_{k-1}$. Substituting these expressions into Eq. (158) yields a single recurrence relation for the parameters

$$\theta_{k+1} - \theta_k = \mu(\theta_k - \theta_{k-1}) - \eta \cdot \frac{dL(\theta_k)}{d\theta_k}. \tag{160}$$

Rearranging the terms to identify finite difference approximations, we obtain

$$\underbrace{(\theta_{k+1} - 2\theta_k + \theta_{k-1})}_{\text{Acceleration Term}} + \underbrace{(1-\mu)(\theta_k - \theta_{k-1})}_{\text{Velocity Term}} = -\eta \cdot \frac{dL(\theta_k)}{d\theta_k}. \tag{161}$$

To obtain a meaningful limit, we introduce a temporal step size Δt such that $t = k \cdot \Delta t$. Following the standard scaling analysis [Polyak, 1964; Su et al., 2016], we assume the learning rate scales quadratically with the step size ($\eta = (\Delta t)^2$) and the momentum coefficient scales as $\mu = 1 - \lambda(\Delta t)^2$, where $\lambda > 0$ acts as a damping factor. Substituting these scalings into the equation and dividing by $(\Delta t)^2$, we get

$$\frac{\theta_{k+1} - 2\theta_k + \theta_{k-1}}{(\Delta t)^2} + \lambda \cdot \frac{\theta_k - \theta_{k-1}}{\Delta t} = -\frac{dL(\theta_k)}{d\theta_k}. \tag{162}$$

Taking the limit as $\Delta t \to 0$, the finite difference quotients converge to the second and first derivatives with respect to time, respectively. Finally, the sequence of updates converges to the heavy ball ODE

$$\frac{d^2\theta_t}{dt^2} + \lambda \cdot \frac{d\theta_t}{dt} = -\frac{dL(\theta_t)}{d\theta_t}. \tag{163}$$

Appendix B. Deriving CNFs from the Continuity Equation

Here we provide a derivation showing that the instantaneous change of variables formula used in CNFs is a direct consequence of the continuity equation governing probability conservation.

Let $p_t(\mathbf{x})$ denote the probability density function at time t, and let $f(\mathbf{x}, \theta, t)$ denote the continuous vector field generated by the neural network. Note that, for notational brevity, here we use $\mathbf{x}$ to denote the time-dependent trajectory $\mathbf{x}(t)$. The conservation of probability mass is described

by the continuity equation (Eulerian view)

$$\frac{\partial p_t(\mathbf{x})}{\partial t} + \nabla \cdot (p_t(\mathbf{x}) f(\mathbf{x}, \theta, t)) \quad = \quad 0, \tag{164}$$

where $\nabla\cdot$ denotes the divergence operator with respect to $\mathbf{x}$.

Using the vector calculus identity $\nabla \cdot (\phi \mathbf{A}) = (\nabla \phi) \cdot \mathbf{A} + \phi(\nabla \cdot \mathbf{A})$, we expand the divergence term, as follows

$$\frac{\partial p_t(\mathbf{x})}{\partial t} + \nabla p_t(\mathbf{x}) \cdot f(\mathbf{x}, \theta, t) + p_t(\mathbf{x}) \nabla \cdot f(\mathbf{x}, \theta, t) \quad = \quad 0. \tag{165}$$

While Eq. (165) describes the density evolution at a fixed point in space, CNFs operate by tracking the density along the trajectory of a moving particle (Lagrangian view). The trajectory is defined by the ODE $\frac{d\mathbf{x}}{dt} = f(\mathbf{x}(t), \theta, t)$.

To bridge these two perspectives, we compute the total time derivative (also known as the material derivative) of the probability density along this trajectory. First, applying the multivariate chain rule to $p_t(\mathbf{x})$, we obtain the general form

$$\frac{dp_t(\mathbf{x})}{dt} \quad = \quad \frac{\partial p_t(\mathbf{x})}{\partial t} + \nabla p_t(\mathbf{x}) \cdot \frac{d\mathbf{x}}{dt}. \tag{166}$$

Here, from the perspective of fluid mechanics, the partial derivative $\frac{\partial p_t}{\partial t}$ can be seen as the local rate of change at a fixed spatial location, while the total derivative $\frac{dp_t}{dt}$ can be seen as the total change experienced by the particle as it moves along its trajectory.

Next, by substituting the specific ODE of the flow $\frac{d\mathbf{x}}{dt} = f(\mathbf{x}, \theta, t)$ into Eq. (166), we obtain the material derivative in terms of the vector field f

$$\frac{dp_t(\mathbf{x})}{dt} \quad = \quad \frac{\partial p_t(\mathbf{x})}{\partial t} + \nabla p_t(\mathbf{x}) \cdot f(\mathbf{x}, \theta, t). \tag{167}$$

Now, by substituting the expression for the material derivative from Eq. (167) into the expanded continuity Eq. (165), we can identify and replace the first two terms

$$\underbrace{\frac{\partial p_t}{\partial t} + \nabla p_t \cdot f}_{\frac{dp_t}{dt}} + p_t(\nabla \cdot f) \quad = \quad 0, \tag{168}$$

where arguments for p_t and f are omitted for clarity. This simplifies the relation to

$$\frac{dp_t(\mathbf{x})}{dt} = -p_t(\mathbf{x}) \nabla \cdot f(\mathbf{x}, \theta, t). \tag{169}$$

To derive the standard CNF formulation, we consider the log-density $\log p_t(\mathbf{x})$. Using the derivative rule $\frac{d}{dt} \log u(t) = \frac{1}{u(t)} \frac{du(t)}{dt}$, we divide both sides by $p_t(\mathbf{x})$, and obtain

$$\frac{d \log p_t(\mathbf{x})}{dt} \quad = \quad -\nabla \cdot f(\mathbf{x}, \theta, t). \tag{170}$$

Finally, recognizing that the divergence of the vector field is the trace of its Jacobian matrix, i.e., $\nabla \cdot f = \mathrm{Tr}\left(\frac{\partial f}{\partial \mathbf{x}}\right)$, we arrive at the instantaneous change of variables formula (i.e., Eq. (73))

$$\frac{d \log p_t(\mathbf{x})}{dt} = -\mathrm{Tr}\left(\frac{\partial f(\mathbf{x}, \theta, t)}{\partial \mathbf{x}}\right). \tag{171}$$

Appendix C. Deriving Probability Flow ODEs from SDEs

Here, we provide the detailed derivation of the probability flow ODE introduced in Section 4.3. We demonstrate that for a given diffusion SDE, there exists a deterministic ODE whose trajectories induce exactly the same marginal probability densities $\{p_t\}_{t=0}^{T}$.

Consider the forward SDE defined as

$$d\mathbf{x}(t) = f(\mathbf{x}(t), t)dt + g(t)d\mathbf{w}(t), \tag{172}$$

where $f(\mathbf{x}(t), t)$ is the drift coefficient, $g(t)$ is the diffusion coefficient, and $\mathbf{w}(t)$ is the standard Wiener process. In the following we omit the argument t in $\mathbf{x}(t)$ to keep the notation uncluttered.

The time evolution of the probability density function $p_t(\mathbf{x})$ of the state variable $\mathbf{x}$ is governed by the Fokker-Planck equation (also known as the Kolmogorov forward equation):

$$\frac{\partial p_t(\mathbf{x})}{\partial t} = -\nabla \cdot [f(\mathbf{x}, t)p_t(\mathbf{x})] + \frac{1}{2}g(t)^2 \Delta p_t(\mathbf{x}), \tag{173}$$

where $\nabla\cdot$ denotes the divergence operator and $\Delta = \nabla \cdot \nabla$ is the Laplace operator. The first term on the right-hand side represents the advection of probability mass due to the drift, while the second term represents the diffusion caused by the stochastic noise.

To derive the equivalent ODE, we rewrite Eq. (173) in the form of a continuity equation, which describes the density evolution of a deterministic system (also see Appendix B). A continuity equation involves only first-order derivatives and takes the form

$$\frac{\partial p_t(\mathbf{x})}{\partial t} = -\nabla \cdot [\tilde{f}(\mathbf{x}, t)p_t(\mathbf{x})], \tag{174}$$

where $\tilde{f}(\mathbf{x}, t)$ corresponds to the vector field of the deterministic ODE.

The key step is to express the diffusion term $\Delta p_t(\mathbf{x})$ (which involves second-order derivatives) as the divergence of a first-order term. We use the identity $\nabla p_t(\mathbf{x}) = p_t(\mathbf{x})\nabla \log p_t(\mathbf{x})$ to rewrite the Laplacian as

$$\begin{aligned} \Delta p_t(\mathbf{x}) &= \nabla \cdot (\nabla p_t(\mathbf{x})) \\ &= \nabla \cdot (p_t(\mathbf{x})\nabla \log p_t(\mathbf{x})). \end{aligned} \tag{175}$$

Substituting this identity back into the Fokker-Planck Eq. (173), we can group the terms under a single divergence operator, as follows

$$\begin{aligned}\frac{\partial p_t(\mathbf{x})}{\partial t} &= -\nabla\cdot[f(\mathbf{x},t)p_t(\mathbf{x})] + \frac{1}{2}g(t)^2\nabla\cdot[p_t(\mathbf{x})\nabla\log p_t(\mathbf{x})] \\ &= -\nabla\cdot\left[f(\mathbf{x},t)p_t(\mathbf{x}) - \frac{1}{2}g(t)^2 p_t(\mathbf{x})\nabla\log p_t(\mathbf{x})\right] \\ &= -\nabla\cdot\left[\left(f(\mathbf{x},t) - \frac{1}{2}g(t)^2\nabla\log p_t(\mathbf{x})\right)p_t(\mathbf{x})\right]. \end{aligned} \tag{176}$$

Comparing this result with the continuity equation in Eq. (174), we find that the equations match if we define the effective deterministic vector field $\tilde{f}(\mathbf{x},t)$ as:

$$\tilde{f}(\mathbf{x},t) = f(\mathbf{x},t) - \frac{1}{2}g(t)^2\nabla\log p_t(\mathbf{x}). \tag{177}$$

Therefore, the deterministic ODE yields identical same marginal distributions $p_t(\mathbf{x})$ as the stochastic process defined by the original SDE. This ODE is given by

$$\frac{d\mathbf{x}}{dt} = f(\mathbf{x},t) - \frac{1}{2}g(t)^2\nabla\log p_t(\mathbf{x}). \tag{178}$$

References

Michael Samuel Albergo and Eric Vanden-Eijnden. Building normalizing flows with stochastic interpolants. In The Eleventh International Conference on Learning Representations, 2023.

Brian D.O. Anderson. Reverse-time diffusion equation models. Stochastic Processes and their Applications, 12(3):313–326, 1982.

Tom M Apostol. Calculus, Volume 1. John Wiley & Sons, 1991.

Jacob Austin, Daniel D Johnson, Jonathan Ho, Daniel Tarlow, and Rianne Van Den Berg. Structured denoising diffusion models in discrete state-spaces. Advances in neural information processing systems, 34:17981–17993, 2021.

Alexei Baevski and Michael Auli. Adaptive input representations for neural language modeling. In International Conference on Learning Representations, 2019.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. Qwen technical report. arXiv preprint arXiv:2309.16609, 2023.

Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders. Machine learning for data science handbook: data mining and knowledge discovery handbook, pages 353–374, 2023.

George Keith Batchelor. An introduction to fluid dynamics. Cambridge university press, 2000.

Jens Behrmann, Will Grathwohl, Ricky TQ Chen, David Duvenaud, and Jörn-Henrik Jacobsen. Invertible residual networks. In International conference on machine learning, pages 573–582. PMLR, 2019.

Peter F. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: Parameter estimation. Computational Linguistics, 19(2):263–311, 1993.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. Advances in neural information processing systems, 33:1877–1901, 2020.

Richard L. Burden, J. Douglas Faires, and Annette M. Burden. Numerical Analysis. Cengage Learning, 2015.

Andrew Campbell, Jason Yim, Regina Barzilay, Tom Rainforth, and Tommi Jaakkola. Generative flows on discrete state-spaces: Enabling multimodal flows with applications to protein co-design. In International Conference on Machine Learning, pages 5453–5512. PMLR, 2024.

Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. Advances in neural information processing systems, 31, 2018.

Ricky TQ Chen, Jens Behrmann, David K Duvenaud, and Jörn-Henrik Jacobsen. Residual flows for invertible generative modeling. Advances in neural information processing systems, 32, 2019.

Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. In Proceedings of International Conference on Learning Representations, 2019.

EA Coddington. Theory of ordinary differential equations. McGraw-Hill Book Company, 1955.

Florinel-Alin Croitoru, Vlad Hondru, Radu Tudor Ionescu, and Mubarak Shah. Diffusion models in vision: A survey. IEEE transactions on pattern analysis and machine intelligence, 45(9): 10850–10869, 2023.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. In International Conference on Learning Representations, 2019.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pages 4171–4186, 2019.

Sander Dieleman. Perspectives on diffusion, 2023. URL https://sander.ai/2023/07/20/perspectives.html.

Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear independent components estimation. arXiv preprint arXiv:1410.8516, 2014.

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real nvp. In International Conference on Learning Representations, 2017.

William Dunham. The calculus gallery: Masterpieces from Newton to Lebesgue. Princeton University Press, 2005.

CH Jr Edwards. The historical development of the calculus. Springer Science & Business Media, 1994.

Patrick Esser, Sumith Kulal, Andreas Blattmann, Rahim Entezari, Jonas Müller, Harry Saini, Yam Levi, Dominik Lorenz, Axel Sauer, Frederic Boesel, Dustin Podell, Tim Dockhorn, Zion English, Kyle Lacey, Alex Goodwin, Yannik Marek, and Robin Rombach. Scaling rectified flow transformers for high-resolution image synthesis. In Forty-first international conference on machine learning, 2024.

Itai Gat, Tal Remez, Neta Shaul, Felix Kreuk, Ricky TQ Chen, Gabriel Synnaeve, Yossi Adi, and Yaron Lipman. Discrete flow matching. Advances in Neural Information Processing Systems, 37:133345–133385, 2024.

Marjan Ghazvininejad, Omer Levy, Yinhan Liu, and Luke Zettlemoyer. Mask-predict: Parallel decoding of conditional masked language models. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), pages 6112–6121, 2019.

Daniel T Gillespie. Exact stochastic simulation of coupled chemical reactions. The journal of physical chemistry, 81(25):2340–2361, 1977.

Daniel T Gillespie. Approximate accelerated stochastic simulation of chemically reacting systems. The Journal of chemical physics, 115(4):1716–1733, 2001.

Shansan Gong, Mukai Li, Jiangtao Feng, Zhiyong Wu, and Lingpeng Kong. Diffuseq: Sequence to sequence text generation with diffusion models. In The Eleventh International Conference on Learning Representations, 2023.

Henry Gouk, Eibe Frank, Bernhard Pfahringer, and Michael J Cree. Regularisation of neural networks by enforcing lipschitz continuity. Machine Learning, 110(2):393–416, 2021.

Or Greenberg. Demystifying flux architecture. arXiv preprint arXiv:2507.09595, 2025.

J Gu, J Bradbury, C Xiong, VOK Li, and R Socher. Non-autoregressive neural machine translation. In International Conference on Learning Representations (ICLR), 2018.

Jiatao Gu, Changhan Wang, and Junbo Zhao. Levenshtein transformer. Advances in neural information processing systems, 32, 2019.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, Hao Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun

Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, Tao Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. Nat., 645(8081):633–638, 2025. doi: 10.1038/S41586-025-09422-Z. URL https://doi.org/10.1038/s41586-025-09422-z.

Eldad Haber and Lars Ruthotto. Stable architectures for deep neural networks. Inverse problems, 34(1), 2017.

Ernst Hairer and Gerhard Wanner. Solving Ordinary Differential Equations II. Cambridge university press, 1996.

Philip Hartman. Ordinary differential equations. SIAM, 2002.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.

Morris W Hirsch, Stephen Smale, and Robert L Devaney. Differential equations, dynamical systems, and an introduction to chaos. Academic press, 2013.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. Advances in neural information processing systems, 33:6840–6851, 2020a.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. Advances in neural information processing systems, 33:6840–6851, 2020b.

Emiel Hoogeboom, Jorn Peters, Rianne Van Den Berg, and Max Welling. Integer discrete flows and lossless compression. Advances in Neural Information Processing Systems, 32, 2019.

Emiel Hoogeboom, Didrik Nielsen, Priyank Jaini, Patrick Forré, and Max Welling. Argmax flows and multinomial diffusion: Learning categorical distributions. Advances in neural information processing systems, 34:12454–12465, 2021.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 4700–4708, 2017.

Aapo Hyvärinen and Peter Dayan. Estimation of non-normalized statistical models by score matching. Journal of Machine Learning Research, 6(4), 2005.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. In International Conference on Learning Representations, 2017.

Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S Weld, Luke Zettlemoyer, and Omer Levy. Spanbert: Improving pre-training by representing and predicting spans. Transactions of the association for computational linguistics, 8:64–77, 2020.

Ioannis Karatzas and Steven Shreve. Brownian motion and stochastic calculus. springer, 2014.

Tero Karras, Miika Aittala, Timo Aila, and Samuli Laine. Elucidating the design space of diffusion-based generative models. Advances in neural information processing systems, 35:26565–26577, 2022.

Diederik P Kingma and Ruiqi Gao. Understanding diffusion objectives as the ELBO with simple data augmentation. In Thirty-seventh Conference on Neural Information Processing Systems, 2023.

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

Diederik P Kingma and Max Welling. An introduction to variational autoencoders. Foundations and Trends® in Machine Learning, 12(4):307–392, 2019.

Durk P Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. Advances in neural information processing systems, 31, 2018.

Durk P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improved variational inference with inverse autoregressive flow. Advances in neural information processing systems, 29, 2016.

Peter E Kloeden and RA Pearson. The numerical solution of stochastic differential equations. The ANZIAM Journal, 20(1):8–12, 1977.

Peter E. Kloeden and Eckhard Platen. Numerical Solution of Stochastic Differential Equations. Springer Berlin, 1992.

Hyukhun Koh, Minha Jhang, Dohyung Kim, Sangmook Lee, and Kyomin Jung. Conditional [mask] discrete diffusion language model. In Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 8910–8934, 2025.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In International Conference on Learning Representations, 2020.

Yann LeCun, Sumit Chopra, Raia Hadsell, M Ranzato, Fujie Huang, et al. A tutorial on energy-based learning. Predicting structured data, 1, 2006.

Jason Lee, Elman Mansimov, and Kyunghyun Cho. Deterministic non-autoregressive neural sequence modeling by iterative refinement. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, pages 1173–1182, 2018.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 7871–7880, 2020.

Bei Li, Quan Du, Tao Zhou, Yi Jing, Shuhan Zhou, Xin Zeng, Tong Xiao, Jingbo Zhu, Xuebo Liu, and Min Zhang. Ode transformer: An ordinary differential equation-inspired model for sequence generation. In Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 8335–8351, 2022a.

Bei Li, Tong Zheng, Rui Wang, Jiahao Liu, Junliang Guo, Xu Tan, Tong Xiao, JingBo Zhu, Jingang Wang, and Xunliang Cai. Predictor-corrector enhanced transformers with exponential moving average coefficient learning. Advances in Neural Information Processing Systems, 37: 20358–20382, 2024.

Jinsong Li, Xiaoyi Dong, Yuhang Zang, Yuhang Cao, Jiaqi Wang, and Dahua Lin. Beyond fixed: Training-free variable-length denoising for diffusion large language models, 2025.

Xiang Li, John Thickstun, Ishaan Gulrajani, Percy S Liang, and Tatsunori B Hashimoto. Diffusion-lm improves controllable text generation. Advances in neural information processing systems, 35:4328–4343, 2022b.

Yaron Lipman, Ricky T. Q. Chen, Heli Ben-Hamu, Maximilian Nickel, and Matthew Le. Flow matching for generative modeling. In The Eleventh International Conference on Learning Representations, 2023.

Yaron Lipman, Marton Havasi, Peter Holderrieth, Neta Shaul, Matt Le, Brian Karrer, Ricky TQ Chen, David Lopez-Paz, Heli Ben-Hamu, and Itai Gat. Flow matching guide and code. arXiv preprint arXiv:2412.06264, 2024.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding,

Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report. arXiv preprint arXiv:2412.19437, 2024.

Luping Liu, Yi Ren, Zhijie Lin, and Zhou Zhao. Pseudo numerical methods for diffusion models on manifolds. In International Conference on Learning Representations, 2022.

Xingchao Liu, Chengyue Gong, and qiang liu. Flow straight and fast: Learning to generate and transfer data with rectified flow. In The Eleventh International Conference on Learning Representations, 2023.

Xinyu Liu, Bei Li, Jiahao Liu, Junhao Ruan, Kechen Jiao, Hongyin Tang, Jingang Wang, Tong Xiao, and Jingbo Zhu. Iiet: Efficient numerical transformer via implicit iterative euler method. In Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 8955–8969, 2025.

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion modeling by estimating the ratios of the data distribution. In Proceedings of the 41st International Conference on Machine Learning, pages 32819–32848, 2024.

Cheng Lu and Yang Song. Simplifying, stabilizing and scaling continuous-time consistency models. In The Thirteenth International Conference on Learning Representations, 2025.

Cheng Lu, Yuhao Zhou, Fan Bao, Jianfei Chen, Chongxuan Li, and Jun Zhu. Dpm-solver: A fast ode solver for diffusion probabilistic model sampling in around 10 steps. Advances in neural information processing systems, 35:5775–5787, 2022.

Cheng Lu, Yuhao Zhou, Fan Bao, Jianfei Chen, Chongxuan Li, and Jun Zhu. Dpm-solver++: Fast solver for guided sampling of diffusion probabilistic models. Machine Intelligence Research, pages 1–22, 2025.

Yiping Lu, Zhuohan Li, Di He, Zhiqing Sun, Bin Dong, Tao Qin, Liwei Wang, and Tie-yan Liu. Understanding and improving transformer from a multi-particle dynamic system point of view. In ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations, 2020.

Calvin Luo. Understanding diffusion models: A unified perspective. arXiv, 2022.

Simian Luo, Yiqin Tan, Longbo Huang, Jian Li, and Hang Zhao. Latent consistency models: Synthesizing high-resolution images with few-step inference. arXiv preprint arXiv:2310.04378, 2023a.

Simian Luo, Yiqin Tan, Suraj Patil, Daniel Gu, Patrick Von Platen, Apolinário Passos, Longbo Huang, Jian Li, and Hang Zhao. Lcm-lora: A universal stable-diffusion acceleration module. arXiv preprint arXiv:2311.05556, 2023b.

Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. In International Conference on Learning Representations, 2017.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Proceedings of the International Conference on Learning Representations (ICLR 2013), 2013a.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2, pages 3111–3119, 2013b.

Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral normalization for generative adversarial networks. In International Conference on Learning Representations, 2018.

Amin Karimi Monsefi, Nikhil Bhendawade, Manuel Rafael Ciosici, Dominic Culver, Yizhe Zhang, and Irina Belousova. Fs-dfm: Fast and accurate long text generation with few-step diffusion language models. arXiv preprint arXiv:2509.20624, 2025.

Behnam Neyshabur. Implicit regularization in deep learning. arXiv preprint arXiv:1709.01953, 2017.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models. arXiv preprint arXiv:2502.09992, 2025.

James R Norris. Markov chains. Cambridge university press, 1998.

Bernt Oksendal. Stochastic differential equations: an introduction with applications. Springer Science & Business Media, 2013.

George Papamakarios, Theo Pavlakou, and Iain Murray. Masked autoregressive flow for density estimation. Advances in neural information processing systems, 30, 2017.

William Peebles and Saining Xie. Scalable diffusion models with transformers. In Proceedings of the IEEE/CVF international conference on computer vision, pages 4195–4205, 2023.

Boris T Polyak. Some methods of speeding up the convergence of iteration methods. Ussr computational mathematics and mathematical physics, 4(5):1–17, 1964.

Ning Qian. On the momentum term in gradient descent learning algorithms. Neural networks, 12(1):145–151, 1999.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. OpenAI, 2018.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of Machine Learning Research, 21(140):1–67, 2020.

Machel Reid, Vincent J Hellendoorn, and Graham Neubig. Diffuser: Discrete diffusion via edit-based reconstruction. arXiv preprint arXiv:2210.16886, 2022.

Christian P Robert, George Casella, and George Casella. Monte Carlo statistical methods, volume 2. Springer, 1999.

Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, pages 10684–10695, 2022.

Dimitri von Rütte, Janis Fluri, Yuhui Ding, Antonio Orvieto, Bernhard Schölkopf, and Thomas Hofmann. Generalized interpolating discrete diffusion. In Forty-second International Conference on Machine Learning, 2025.

Subham Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models. Advances in Neural Information Processing Systems, 37:130136–130184, 2024.

Shreshth Saini, Shashank Gupta, and Alan C Bovik. Rectified-cfg++ for flow based models. arXiv preprint arXiv:2510.07631, 2025.

Tim Salimans and Jonathan Ho. Progressive distillation for fast sampling of diffusion models. In International Conference on Learning Representations, 2022.

Axel Sauer, Dominik Lorenz, Andreas Blattmann, and Robin Rombach. Adversarial diffusion distillation. In European Conference on Computer Vision, pages 87–103. Springer, 2024.

Neta Shaul, Itai Gat, Marton Havasi, Daniel Severo, Anuroop Sriram, Peter Holderrieth, Brian Karrer, Yaron Lipman, and Ricky T. Q. Chen. Flow matching with general discrete paths: A kinetic-optimal perspective. In The Thirteenth International Conference on Learning Representations, 2025.

Hui Shen, Jingxuan Zhang, Boning Xiong, Rui Hu, Shoufa Chen, Zhongwei Wan, Xin Wang, Yu Zhang, Zixuan Gong, Guangyin Bao, Chaofan Tao, Yongfeng Huang, Ye Yuan, and Mi Zhang. Efficient diffusion models: A survey. arXiv preprint arXiv:2502.06805, 2025.

Yang Song. Generative modeling by estimating gradients of the data distribution, 2021. URL https://yang-song.net/blog/2021/score/.

Yang Song and Stefano Ermon. Generative modeling by estimating gradients of the data distribution. Advances in neural information processing systems, 32, 2019.

Yang Song, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. In International Conference on Learning Representations, 2021.

Yang Song, Prafulla Dhariwal, Mark Chen, and Ilya Sutskever. Consistency models. In International Conference on Machine Learning, pages 32211–32252. PMLR, 2023.

Michael Spivak. Calculus. Cambridge University Press, 2006.

Josef Stoer, Roland Bulirsch, R Bartels, Walter Gautschi, and Christoph Witzgall. Introduction to numerical analysis. Springer, 1980.

Gilbert Strang. On the construction and comparison of difference schemes. SIAM journal on numerical analysis, 5(3):506–517, 1968.

Robin Strudel, Corentin Tallec, Florent Altché, Yilun Du, Yaroslav Ganin, Arthur Mensch, Will Grathwohl, Nikolay Savinov, Sander Dieleman, Laurent Sifre, and Rémi Leblond. Self-conditioned embedding diffusion for text generation. arXiv preprint arXiv:2211.04236, 2022.

Weijie Su, Stephen Boyd, and Emmanuel J Candes. A differential equation for modeling nesterov's accelerated gradient method: Theory and insights. Journal of Machine Learning Research, 17 (153):1–43, 2016.

Terence Tao. An introduction to measure theory, volume 126. American Mathematical Society, 2011.

Yi Tay, Mostafa Dehghani, Vinh Q. Tran, Xavier Garcia, Jason Wei, Xuezhi Wang, Hyung Won Chung, Dara Bahri, Tal Schuster, Steven Zheng, Denny Zhou, Neil Houlsby, and Donald Metzler. UL2: Unifying language learning paradigms. In The Eleventh International Conference on Learning Representations, 2023.

Dustin Tran, Keyon Vafa, Kumar Agrawal, Laurent Dinh, and Ben Poole. Discrete flows: Invertible generative models of discrete data. Advances in Neural Information Processing Systems, 32, 2019.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. Advances in neural information processing systems, 30, 2017.

Pascal Vincent. A connection between score matching and denoising autoencoders. Neural computation, 23(7):1661–1674, 2011.

Fu-Yun Wang, Zhaoyang Huang, Alexander Bergman, Dazhong Shen, Peng Gao, Michael Lingelbach, Keqiang Sun, Weikang Bian, Guanglu Song, Yu Liu, et al. Phased consistency models. Advances in neural information processing systems, 37:83951–84009, 2024.

Qiang Wang, Fuxue Li, Tong Xiao, Yanyang Li, Yinqiao Li, and Jingbo Zhu. Multi-layer representation fusion for neural machine translation. In Proceedings of the 27th international conference on computational linguistics, pages 3015–3026, 2018.

Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F Wong, and Lidia S Chao. Learning deep transformer models for machine translation. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, pages 1810–1822, 2019.

Tong Xiao and Jingbo Zhu. Introduction to transformers: an nlp perspective. arXiv preprint arXiv:2311.17633, 2023.

Yisheng Xiao, Lijun Wu, Junliang Guo, Juntao Li, Min Zhang, Tao Qin, and Tie-yan Liu. A survey on non-autoregressive generation for neural machine translation and beyond. IEEE Transactions on Pattern Analysis and Machine Intelligence, 45(10):11407–11427, 2023.

Hanshu Yan, Xingchao Liu, Jiachun Pan, Jun Hao Liew, qiang liu, and Jiashi Feng. PeRFlow: Piecewise rectified flow as universal plug-and-play accelerator. In The Thirty-eighth Annual Conference on Neural Information Processing Systems, 2024.

Ling Yang, Zhilong Zhang, Yang Song, Shenda Hong, Runsheng Xu, Yue Zhao, Wentao Zhang, Bin Cui, and Ming-Hsuan Yang. Diffusion models: A comprehensive survey of methods and applications. ACM computing surveys, 56(4):1–39, 2023.

Liu Yang, Kangwook Lee, Robert D Nowak, and Dimitris Papailiopoulos. Looped transformers are better at learning learning algorithms. In The Twelfth International Conference on Learning Representations, 2024. URL https://openreview.net/forum?id=HHbRxoDTxE.

Xiaofeng Yang, Chen Cheng, Xulei Yang, Fayao Liu, and Guosheng Lin. Text-to-image rectified flow as plug-and-play priors. In The Thirteenth International Conference on Learning Representations, 2025a.

Yicun Yang, Cong Wang, Shaobo Wang, Zichen Wen, Biqing Qi, Hanlin Xu, and Linfeng Zhang. Diffusion llm with native variable generation lengths: Let [eos] lead the way, 2025b.

Qinsheng Zhang and Yongxin Chen. Fast sampling of diffusion models with exponential integrator. In NeurIPS 2022 Workshop on Score-Based Methods, 2022.

Shuibai Zhang, Fred Zhangzhi Peng, Yiheng Zhang, Jin Pan, and Grigorios G Chrysos. Corrective diffusion language models. arXiv preprint arXiv:2512.15596, 2025a.

Xinxi Zhang, Shiwei Tan, Quang Nguyen, Quan Dao, Ligong Han, Xiaoxiao He, Tunyu Zhang, Alen Mrdovic, and Dimitris Metaxas. Flow straighter and faster: Efficient one-step generative modeling via meanflow on rectified trajectories. arXiv preprint arXiv:2511.23342, 2025b.

Wenliang Zhao, Lujia Bai, Yongming Rao, Jie Zhou, and Jiwen Lu. Unipc: A unified predictor-corrector framework for fast sampling of diffusion models. Advances in Neural Information Processing Systems, 36:49842–49869, 2023.

Jianbin Zheng, Minghui Hu, Zhongyi Fan, Chaoyue Wang, Changxing Ding, Dacheng Tao, and Tat-Jen Cham. Trajectory consistency distillation. CoRR, 2024.

Kaiwen Zheng, Cheng Lu, Jianfei Chen, and Jun Zhu. DPM-solver-v3: Improved diffusion ODE solver with empirical model statistics. In Thirty-seventh Conference on Neural Information Processing Systems, 2023. URL https://openreview.net/forum?id=9fWKExmKa0.

Zhenyu Zhou, Defang Chen, Can Wang, and Chun Chen. Fast ode-based sampling for diffusion models in around 5 steps. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 7777–7786, 2024.

Chen Zhu, Yu Cheng, Zhe Gan, Siqi Sun, Tom Goldstein, and Jingjing Liu. Freelb: Enhanced adversarial training for natural language understanding. In International Conference on Learning Representations, 2020.