Image Generation using Normalizing Flows and Flow Matching

Nilanjan Ray

Math primers needed for some upcoming generative models

- Maximum likelihood estimation (MLE)
- Ordinary differential equations (ODE)
- Transformation of random variables

MLE

🔢 1️⃣ Maximum Likelihood Estimation (MLE)

We start with **a parametric model** $p(x;\theta)$
and a dataset $\{x_i\}_{i=1}^N$ assumed **i.i.d.** from that model.

The **likelihood** is:

$$L(\theta) = \prod_{i=1}^{N} p(x_i;\theta)$$

and we usually work with the **log-likelihood**:

$$\ell(\theta) = \sum_{i=1}^{N} \log p(x_i;\theta)$$

The MLE is:

$$\hat{\theta} = \arg\max_{\theta} \ell(\theta)$$

Example: 1D Gaussian

$$p(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

The log-likelihood:

$$\ell(\mu,\sigma) = -N\log\sigma - \frac{1}{2\sigma^2}\sum_i (x_i-\mu)^2 + \text{const.}$$

Set derivatives to zero:

$$\frac{\partial \ell}{\partial \mu} = 0 \;\Rightarrow\; \hat{\mu} = \frac{1}{N}\sum_i x_i$$

$$\frac{\partial \ell}{\partial \sigma} = 0 \;\Rightarrow\; \hat{\sigma}^2 = \frac{1}{N}\sum_i (x_i-\hat{\mu})^2$$

✅ This is how MLE "learns" parameters by maximizing data likelihood — the same principle flows and VAEs use, but in much higher dimensions.

ODEs

🧩 2️⃣ Ordinary Differential Equations (ODEs)

A simple 1D ODE defines how a variable $y(t)$ evolves with time:

$$\frac{dy}{dt} = f(y, t)$$

Given an initial condition $y(0) = y_0$, the **solution** is the curve that satisfies this equation.

Example

$$\frac{dy}{dt} = -2y, \quad y(0) = 3$$

Analytic solution:

$$y(t) = 3e^{-2t}$$

At $t = 0.5$, $y = 3e^{-1} \approx 1.10$

Euler's Method (numerical)

We can approximate the solution iteratively:

$$y_{t+\Delta t} = y_t + \Delta t\, f(y_t, t)$$

Example:

$$y_{t+1} = y_t - 2\Delta t\, y_t$$

If $\Delta t = 0.1$:

$$y_1 = 3 - 0.6 = 2.4,\ y_2 = 2.4 - 0.48 = 1.92,\ \text{etc.}$$

✅ Euler's method is the foundation of how **Neural ODEs** integrate continuous dynamics of hidden states.

Transformation of R.V.s

🔁 3️⃣ Random Variable Transformation (1D Case)

Suppose $Z \sim p_Z(z)$, and we define
$X = g(Z)$, where g is **invertible** and **differentiable**.

Then the probability densities satisfy:

$$p_X(x) = p_Z(g^{-1}(x)) \left| \frac{dg^{-1}(x)}{dx} \right|$$

Example

Let $Z \sim \mathcal{N}(0, 1)$, and define $X = g(Z) = 2Z + 3$.

Then:

$$g^{-1}(x) = \frac{x-3}{2}, \quad \frac{dg^{-1}(x)}{dx} = \frac{1}{2}$$

$$p_X(x) = \frac{1}{2} p_Z\left(\frac{x-3}{2}\right)$$

✅ So the mean becomes 3, the std becomes 2:
the Gaussian transforms exactly as expected.

🌐 4️⃣ Random Variable Transformation (n-D Case)

For a multivariate invertible transform $\mathbf{x} = g(\mathbf{z})$:

$$p_X(\mathbf{x}) = p_Z(g^{-1}(\mathbf{x})) \left| \det \frac{\partial g^{-1}(\mathbf{x})}{\partial \mathbf{x}} \right|$$

Equivalently (by inverting the Jacobian):

$$\log p_X(\mathbf{x}) = \log p_Z(\mathbf{z}) - \log \left| \det \frac{\partial g(\mathbf{z})}{\partial \mathbf{z}} \right|$$

This is the **change-of-variables formula** — the cornerstone of **normalizing flows**.

✅ Interpretation

- $p_Z(\mathbf{z})$: simple base distribution (e.g., Gaussian)
- $g(\mathbf{z})$: invertible neural network transforming latent to data space
- The Jacobian determinant adjusts the density for stretching/compression of space.

→ If we can compute $\det J$, we can do **exact MLE** on high-dimensional data.

Example: 2D Transformation and the Jacobian Determinant

Suppose we start with a 2D random variable

$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \sim p_Z(\mathbf{z})$$

and define an **invertible transformation** $\mathbf{x} = g(\mathbf{z})$:

$$\begin{cases} x_1 = 2z_1 + z_2 \\ x_2 = z_1 + 3z_2 \end{cases}$$

This is a **linear transformation**, so we can write:

$$\mathbf{x} = A\mathbf{z}, \quad A = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$$

2D Example...

1️⃣ Compute the Jacobian

The **Jacobian matrix** $J_g(\mathbf{z}) = \frac{\partial \mathbf{x}}{\partial \mathbf{z}}$ is:

$$J_g(\mathbf{z}) = \begin{bmatrix} \frac{\partial x_1}{\partial z_1} & \frac{\partial x_1}{\partial z_2} \\ \frac{\partial x_2}{\partial z_1} & \frac{\partial x_2}{\partial z_2} \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$$

(same as A since this transform is linear)

2️⃣ Determinant of the Jacobian

$$\det(J_g) = (2)(3) - (1)(1) = 6 - 1 = 5$$

This tells us that:

- The transformation **expands area** by a factor of 5
- i.e., one unit square in z-space maps to an area 5× larger in x-space

3️⃣ Log Determinant

$$\log |\det(J_g)| = \log 5 \approx 1.609$$

The log-determinant term will appear in the log-likelihood when transforming densities.

4️⃣ Inverse Transformation

We can compute g^{-1} since A is invertible:

$$A^{-1} = \frac{1}{5} \begin{bmatrix} 3 & -1 \\ -1 & 2 \end{bmatrix}$$

So:

$$\begin{cases} z_1 = \frac{1}{5}(3x_1 - x_2) \\ z_2 = \frac{1}{5}(-x_1 + 2x_2) \end{cases}$$

2D Example...

5️⃣ Density Transformation

If $\mathbf{z} \sim p_Z(\mathbf{z})$, then the new density $p_X(\mathbf{x})$ is:

$$p_X(\mathbf{x}) = p_Z(g^{-1}(\mathbf{x})) \, |\det J_{g^{-1}}(\mathbf{x})|$$

But since $J_{g^{-1}} = (J_g)^{-1}$,

$$|\det J_{g^{-1}}| = \frac{1}{|\det J_g|} = \frac{1}{5}$$

So:

$$p_X(\mathbf{x}) = \frac{1}{5} p_Z(g^{-1}(\mathbf{x}))$$

and

$$\log p_X(\mathbf{x}) = \log p_Z(g^{-1}(\mathbf{x})) - \log 5$$

✅ The second term, $-\log |\det J_g|$, is the **volume correction** term that normalizing flows use in log-likelihood computation.

6️⃣ Geometric Intuition

- Determinant $= 5 \rightarrow$ the transformation *stretches* 2D space 5×
- If p_Z is a Gaussian centered at (0,0), the transformed p_X is an **elliptical Gaussian** stretched along the eigenvectors of A.

Optional (Nonlinear Extension)

If we had:

$$x_1 = 2z_1 + \sin(z_2), \quad x_2 = z_2^2$$

then the Jacobian would depend on z:

$$J_g = \begin{bmatrix} 2 & \cos(z_2) \\ 0 & 2z_2 \end{bmatrix}$$

$$\det(J_g) = 4z_2 \quad \text{and} \quad \log|\det(J_g)| = \log|4z_2|$$

which illustrates how **flows handle local stretching/compression** at each point.

Normalizing Flows

Goal:

Learn a flexible target distribution $p_x(x)$ by transforming a simple base distribution $p_z(z)$ (e.g., $\mathcal{N}(0, I)$) through a sequence of *invertible* mappings.

Key Idea:

We generate samples via

$$x = g(z), \quad z \sim p_z(z).$$

If each transformation is **invertible** and has a **tractable Jacobian**, then:

- We can compute **log-likelihoods** of x
- We can **sample** efficiently (just draw z)
- We can **train** using maximum likelihood

Normalizing Flow =

A chain of bijective maps:

$$x = g(z) = g_K \circ g_{K-1} \circ \cdots \circ g_1(z).$$

Each bijection gradually warps the simple base density p_z into the more complex target density p_x.

Change of Variables & Training Objective

Exact log-likelihood:

Since the mapping is invertible, we use the change-of-variables formula:

$$\log p_x(x) = \log p_z(z) - \sum_{k=1}^{K} \log \left| \det J_{g_k}(z_{k-1}) \right| ,$$

where:

- $z_0 = z$,
- $z_K = x$,
- J_{g_k} is the Jacobian of layer g_k.

Training:

Maximize the likelihood of data:

$$\max_{\theta} \ \mathbb{E}_{x \sim \text{data}} [\log p_x(x)].$$

Sampling:

Just sample $z \sim p_z(z)$ and push through flow:

$$x = g(z).$$

An example

Base distribution (in 2D):

$$z = (z_1, z_2)^\top \sim p_z(z) = \mathcal{N}(0, I_2).$$

We build $x = g(z)$ in **two steps:**

1. **Shear (coupling-like) transform**

$$h = g_1(z): \quad \begin{cases} h_1 = z_1 \\ h_2 = z_2 + a\, z_1 \end{cases}$$

Jacobian:

$$J_{g_1}(z) = \begin{pmatrix} 1 & 0 \\ a & 1 \end{pmatrix}, \quad \det J_{g_1} = 1.$$

2. **Scale + shift (per-dimension affine)**

$$x = g_2(h): \quad \begin{cases} x_1 = \sigma_1 h_1 + b_1 \\ x_2 = \sigma_2 h_2 + b_2 \end{cases}$$

Jacobian:

$$J_{g_2}(h) = \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix}, \quad \det J_{g_2} = \sigma_1 \sigma_2.$$

Full flow:

$$x = g(z) = g_2(g_1(z)), \quad J_g(z) = J_{g_2}(h)\, J_{g_1}(z), \quad \det J_g = \sigma_1 \sigma_2.$$

(*Note*: the shear doesn't change volume; the scaling does.)

Log probability

We want the **log-density at a data point x**:

1. **Invert the flow (go back to z)**

From $x = g_2(h)$:

$$h_1 = \frac{x_1 - b_1}{\sigma_1}, \quad h_2 = \frac{x_2 - b_2}{\sigma_2}.$$

From $h = g_1(z)$:

$$z_1 = h_1, \quad z_2 = h_2 - a\, h_1.$$

So:

$$z_1 = \frac{x_1 - b_1}{\sigma_1}, \quad z_2 = \frac{x_2 - b_2}{\sigma_2} - a\,\frac{x_1 - b_1}{\sigma_1}.$$

2. **Compute base log-density $\log p_z(z)$**

For $z \sim \mathcal{N}(0, I_2)$:

$$\log p_z(z) = -\frac{1}{2}\left(z_1^2 + z_2^2\right) - \log(2\pi).$$

3. **Account for volume change (Jacobian)**

With forward map $x = g(z)$:

$$\log p_x(x) = \log p_z(z) - \log\left|\det J_g(z)\right|.$$

Here $\det J_g = \sigma_1\sigma_2$, so

$$\boxed{\log p_x(x) = \log p_z(z) - \log\left|\sigma_1\sigma_2\right|}$$

And we already expressed z_1, z_2 in terms of x, so this gives a **closed-form log-probability** for any x.

Normalizing Flows with RealNVP

Idea

Split the input into two parts and only transform one part conditioned on the other.

Let

$$u = (u_1,\ u_2)$$

Define the coupling transformation:

$$v_1 = u_1$$

$$v_2 = u_2 \odot \exp(s(u_1)) + t(u_1)$$

where:

- $s(\cdot)$ = scale network
- $t(\cdot)$ = shift network
- Only u_2 is transformed, conditioned on u_1

This simple design guarantees **invertibility** and **efficient computation**.

Log-Jacobian Computation is Efficient

The Jacobian of the mapping $u \mapsto v$ is:

$$J = \begin{pmatrix} I & 0 \\ \frac{\partial v_2}{\partial u_1} & \operatorname{diag}(\exp(s(u_1))) \end{pmatrix}$$

This matrix is **lower-triangular**, so:

$$\det J = \prod_i \exp(s_i(u_1)) = \exp\Big(\sum_i s_i(u_1)\Big).$$

Log-determinant is trivial:

$$\log|\det J| = \sum_i s_i(u_1).$$

No large determinant.

No matrix inversion.

Just a **sum** → extremely fast for high-dimensional u.

Inverting the function is easy!

Because the transformation is simple and structured, inversion is closed-form:

$$u_1 = v_1,$$

$$u_2 = \big(v_2 - t(v_1)\big) \odot \exp\big(-s(v_1)\big).$$

Benefits for Normalizing Flows

- Exact likelihood
- Exact inverse
- Fast forward and backward
- Efficient training in high dimensions
- Modular: stack many coupling layers to form a deep flow

This is why coupling layers are the backbone of **RealNVP**, **Glow**, and many modern normalizing-flow models.

Limitations of NF

1. Invertibility Constraints Limit Expressiveness

Because each layer must be **bijective**, flow layers cannot freely use:

- arbitrary neural nets
- dimension changes (no downsampling / upsampling)
- non-invertible operations (ReLU, max-pool, etc.)

This restricts the model class compared to VAEs, GANs, or Diffusion Models.

---

2. Designing Easy Jacobians Limits Flexibility

Flows require:

$$\log \left| \det J_g \right| \text{ cheap to compute}$$

This forces layers to have **triangular** or **structured** Jacobians, reducing expressiveness and leading to architectural compromises (e.g., coupling layers transform only half the variables per layer).

Limitations of NF...

3. High-Dimensional Data Requires Many Layers

On images (e.g., 32×32×3 = 3072-D):

- A few coupling layers barely mix information
- Must stack **tens or hundreds** of invertible blocks to get competitive performance

This increases memory and compute.

---

4. Limited Capacity for Multi-Modal or Discontinuous Densities

Flows learn **continuous, smooth** densities due to invertibility and smoothness of g.

This makes it difficult to model:

- sharply separated modes
- discrete or categorical structure
- heavy tails unless carefully engineered

Limitations of NF...

5. Exact Likelihood Does Not Guarantee Generative Quality

Flows optimize likelihood:

$$\max \log p_x(x)$$

But high likelihood does **not** always correlate with perceptual quality (e.g., Glow samples look blurry despite high NLL).

---

6. Large Memory Footprint

Invertible architectures often require:

- storing intermediate activations for backprop
- storing large invertible 1×1 convolutions
- multi-scale squeezes that increase channel count

This is heavier than VAEs or diffusion models.

---

7. Training Instability for Certain Flow Types

Flows with:

- large log-determinants
- aggressive scaling
- poorly parameterized invertible convs

  can cause exploding/vanishing gradients.

From Discrete Flows to Continuous Normalizing Flows

Discrete Flows Recap

We built flows by composing *finite* invertible maps:

$$x = g_K \circ \cdots \circ g_1(z), \quad z \sim p_z(z).$$

Each map g_k has:

- an easy inverse
- a tractable log-determinant
- a structured Jacobian

But discrete flows face limitations:

- rigid architecture
- triangular Jacobians
- shallow transformations per layer
- many layers needed for complex data

Idea: Make the Flow Continuous

Instead of applying many discrete steps, define a **continuous trajectory** that transports z to x:

$$\frac{dz(t)}{dt} = f(z(t), t; \theta), \quad z(0) \sim p_z(z),$$

and the final data point is:

$$x = z(1).$$

This transforms the generative mapping into solving an **ODE** — a continuous-depth model.

CNF

Continuous Flow = Infinite-Depth Limit

Think of replacing:

$$g_K \circ \cdots \circ g_1$$

with:

$$z(1) = z(0) + \int_0^1 f(z(t), t)\, dt.$$

The flow becomes a *smooth warping* of the base density, controlled by the vector field f.

Key Benefit

Continuous flows remove many discrete invertibility constraints:

- No need for triangular Jacobians
- No coupling-layer structure
- Arbitrary neural nets for f

While still enabling **exact log densities** via the instantaneous change of variables formula.

Continuous Flow Dynamics & Instantaneous Change of Variables

Continuous Normalizing Flow (CNF)

Transform base distribution $p_z(z)$ via an ODE:

$$\frac{dz(t)}{dt} = f(z(t), t; \theta), \quad z(0) \sim p_z(z).$$

At the end of the flow:

$$x = z(1), \quad x \sim p_x(x).$$

Instantaneous Change of Variables

In continuous-time flows, the log-density evolves according to:

$$\frac{d}{dt} \log p(z(t)) = -\mathrm{Tr}\left(\frac{\partial f(z(t), t)}{\partial z(t)} \right).$$

This replaces the discrete-flow log-determinant with a **trace of the Jacobian** of the vector field.

Interpretation

- The ODE warps the distribution smoothly
- The trace term accounts for **infinitesimal volume change**
- No need for triangular or structured Jacobians
- Any neural network can parameterize f

Combined ODE for State and Log-Probability

To compute both the transformed variable and its probability, we solve a **single augmented ODE**:

$$\boxed{\frac{d}{dt}\begin{bmatrix} z(t) \\ \log p(z(t)) \end{bmatrix} = \begin{bmatrix} f(z(t), t) \\ -\mathrm{Tr}\left(\frac{\partial f}{\partial z}\right) \end{bmatrix}}$$

Integrate from $t = 0$ to $t = 1$:

Final Log Density

$$\log p_x(x) = \log p_z(z(0)) - \int_0^1 \mathrm{Tr}\left(\frac{\partial f}{\partial z}\right) dt.$$

Why This Matters

- **Exact likelihoods** (not approximate like VAEs)
- **Flexible architectures** (no invertibility constraints on layers)
- **Smooth, interpretable transformations**
- Works well for continuous manifolds and scientific ML

2D CNF: Simple Linear Scaling ODE

We start from a simple base distribution in 2D:

$$z(0) = (z_1(0), z_2(0))^\top \sim p_z(z) = \mathcal{N}(0, I_2).$$

Define a **linear ODE** (independent scaling along each axis):

$$\frac{d}{dt}\begin{pmatrix} z_1(t) \\ z_2(t) \end{pmatrix} = \begin{pmatrix} a_1 & 0 \\ 0 & a_2 \end{pmatrix}\begin{pmatrix} z_1(t) \\ z_2(t) \end{pmatrix} = A\, z(t),$$

with constants $a_1, a_2 \in \mathbb{R}$.

So the vector field is

$$f(z(t), t) = A\, z(t).$$

Solve the ODE

This linear system has a closed-form solution:

$$z_1(t) = e^{a_1 t} z_1(0), \qquad z_2(t) = e^{a_2 t} z_2(0).$$

At the final time $t = 1$, define

$$x = z(1) = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} e^{a_1} z_1(0) \\ e^{a_2} z_2(0) \end{pmatrix}.$$

So this CNF is just **continuous-time anisotropic scaling** of a 2D Gaussian.

2D CNF: Analytical Log-Probability

Step 1: Log-density evolution (trace term)

For the vector field $f(z,t) = Az$,

$$\frac{\partial f}{\partial z} = A = \begin{pmatrix} a_1 & 0 \\ 0 & a_2 \end{pmatrix}, \quad \mathrm{Tr}\left(\frac{\partial f}{\partial z}\right) = a_1 + a_2.$$

By the CNF formula,

$$\frac{d}{dt}\log p(z(t)) = -\mathrm{Tr}\left(\frac{\partial f}{\partial z}\right) = -(a_1 + a_2).$$

Integrate from $t = 0$ to $t = 1$:

$$\log p(z(1)) = \log p(z(0)) - (a_1 + a_2).$$

Step 2: Express $z(0)$ in terms of x

From the solution:

$$x_1 = e^{a_1} z_1(0) \;\Rightarrow\; z_1(0) = e^{-a_1} x_1,$$

$$x_2 = e^{a_2} z_2(0) \;\Rightarrow\; z_2(0) = e^{-a_2} x_2.$$

Step 3: Plug into the base Gaussian

Base density:

$$\log p_z(z(0)) = -\frac{1}{2}\left(z_1(0)^2 + z_2(0)^2\right) - \log(2\pi).$$

Substitute:

$$\log p_z(z(0)) = -\frac{1}{2}\left(e^{-2a_1} x_1^2 + e^{-2a_2} x_2^2\right) - \log(2\pi).$$

Step 4: Final log-density of x

$$\boxed{\log p_x(x) = -\frac{1}{2}\left(e^{-2a_1} x_1^2 + e^{-2a_2} x_2^2\right) - \log(2\pi) - (a_1 + a_2)}$$

This matches what you'd get from a discrete linear flow with Jacobian determinant $\det = e^{a_1} e^{a_2} = e^{a_1 + a_2}$, but here derived using the **CNF ODE + trace formula**.

Why Do We Need Numerical ODE Solvers?

In our toy 2D example, we could solve the ODE **analytically**:

$$\frac{dz(t)}{dt} = f(z(t), t), \quad \frac{d}{dt}\log p(z(t)) = -\mathrm{Tr}\left(\frac{\partial f}{\partial z}\right).$$

But in real CNFs:

- $f(z(t), t)$ is a **neural network** (highly nonlinear).
- The trace term $\mathrm{Tr}\left(\frac{\partial f}{\partial z}\right)$ depends on **autograd/Jacobians**.
- There is **no closed-form solution** for $z(t)$ or $\log p(z(t))$.

So we treat the combined system

$$\frac{d}{dt}\begin{bmatrix} z(t) \\ \log p(z(t)) \end{bmatrix} = \begin{bmatrix} f(z(t), t) \\ -\mathrm{Tr}\left(\frac{\partial f}{\partial z}\right) \end{bmatrix}$$

as a **black-box ODE** and use a **numerical ODE solver**.

Solving the CNF ODE in Practice

We numerically integrate from $t = 0 \to 1$:

$$\begin{bmatrix} z(1) \\ \log p(z(1)) \end{bmatrix} = \texttt{ODESolve}\left(\begin{bmatrix} z(0) \\ \log p_z(z(0)) \end{bmatrix}, \; f, \; t \in [0, 1] \right).$$

Typical solvers (conceptually):

- **Euler / Heun / RK4** — fixed-step, good for simple flows
- **Adaptive solvers** — automatically adjust step size
  - trade off **accuracy vs speed**
  - cost measured in **# of function evaluations** of f

Key points

- The **architecture** is just the vector field $f(z, t)$.
- The **flow** is determined by the ODE solver.
- During training:
  - Forward pass integrates the ODE to get $x = z(1)$ and $\log p_x(x)$.
  - Backprop differentiates *through* the ODE solver (autograd or adjoint).

This is why CNFs are often called **Neural ODE flows**:

The neural network defines f, and the ODE solver defines the depth.

Backpropagation Through an ODE Solver (Euler Example)

Forward ODE Step (Euler)

For intuition, use the simplest solver:

$$z_{k+1} = z_k + \Delta t\, f(z_k, t_k).$$

A full ODE solve from $t = 0 \to 1$ is just a **long chain** of such steps:

$$z_0 \;\to\; z_1 \;\to\; z_2 \;\to\; \cdots \;\to\; z_N.$$

Each step depends on the previous one → just like an **unrolled RNN**.

Backprop Through All Euler Steps

To compute gradients w.r.t. parameters θ:

$$\frac{\partial L}{\partial \theta} = \sum_{k=0}^{N-1} \frac{\partial L}{\partial z_{k+1}} \frac{\partial z_{k+1}}{\partial \theta}.$$

Since

$$z_{k+1} = z_k + \Delta t\, f(z_k, t_k; \theta),$$

we get:

$$\frac{\partial z_{k+1}}{\partial \theta} = \Delta t\, \frac{\partial f(z_k, t_k; \theta)}{\partial \theta}.$$

Key observation:

To compute
$\frac{\partial L}{\partial z_k}$,
we need all future steps:

$$\frac{\partial L}{\partial z_k} = \frac{\partial L}{\partial z_{k+1}} \frac{\partial z_{k+1}}{\partial z_k}.$$

Thus the computational graph includes **every Euler step**.

→ **We must store all intermediate states** $z_0, \ldots, z_N$.

→ Memory cost grows with **# of ODE steps**.

Problem

- Modern flows use **adaptive solvers**
- They may take **hundreds or thousands** of function evaluations
- Backprop would require storing **all intermediate states**

This is often **too much memory**.

The Adjoint Method: Backprop by Solving Another ODE

To avoid storing every intermediate state, CNFs use the **adjoint method**.

Core idea

Define a new ODE for the sensitivity:

$$a(t) = \frac{\partial L}{\partial z(t)},$$

which evolves according to:

$$\frac{da(t)}{dt} = -a(t)^\top \frac{\partial f(z(t), t)}{\partial z(t)}.$$

This ODE is solved **backward in time**:

$$t = 1 \;\rightarrow\; 0.$$

Practical consequence

- **Forward pass:**
  Solve the original ODE from $t = 0 \rightarrow 1$ to obtain $z(1)$.
- **Backward pass:**
  Instead of backpropagating through every saved step, solve the *adjoint ODE* from $t = 1 \rightarrow 0$.
- The adjoint ODE yields:
  - gradients w.r.t. intermediate states
  - gradients w.r.t. parameters θ
- Only the final state needs to be stored → **dramatic memory savings**.

Key takeaway

Naively backpropagating through every ODE step is memory-heavy.
The adjoint method avoids this by treating gradient computation as **another ODE**, solved backward in time, eliminating the need to store the entire forward trajectory.

Estimating the Trace Term Efficiently (Hutchinson Estimator)

Hutchinson's Trick

Use a random vector v with the property:

$$\mathbb{E}[v^\top v] = I.$$

Examples:

- v with Rademacher entries (±1 with probability ½)
- $v \sim \mathcal{N}(0, I)$

Then:

$$\mathrm{Tr}(A) = \mathbb{E}_v \left[v^\top A v \right].$$

Apply this to the Jacobian:

$$\mathrm{Tr}\left(\frac{\partial f}{\partial z}\right) \approx v^\top \left(\frac{\partial f}{\partial z} v\right).$$

In a CNF, the log-density evolves as:

$$\frac{d}{dt} \log p(z(t)) = -\mathrm{Tr}\left(\frac{\partial f(z(t), t)}{\partial z(t)}\right).$$

Problem

The Jacobian of the vector field
$\frac{\partial f}{\partial z}$
is typically **D × D**, where D is the data dimension.

- For images, D can be **thousands**
- Computing the full Jacobian is **O(D²)**
- Computing its trace directly is often too expensive

We need a way to estimate the trace **without forming the Jacobian**.

Key benefit

We never compute the full Jacobian.

We only need the **Jacobian–vector product**:

$$\frac{\partial f}{\partial z} v,$$

which can be obtained by **automatic differentiation** in O(D) time.

Final CNF Update Using Hutchinson

$$\frac{d}{dt} \log p(z(t)) \approx -v^\top \left(\frac{\partial f}{\partial z} v\right).$$

This gives an efficient, unbiased estimate of the trace with **one or a few random vectors**.

Limitations of Continuous Normalizing Flows (CNFs)

1. Expensive ODE Solves

Sampling and likelihood evaluation require solving an ODE:

- Accuracy ↔ speed trade-off
- Adaptive solvers may take **hundreds or thousands** of function evaluatio
- Training can be significantly slower than discrete flows, VAEs, or diffusio models

---

2. Adjoint Method Is Not Free

The adjoint method reduces memory but:

- Adds extra ODE solves during backprop
- Can suffer from **numerical instability**
- Gradients can drift if the backward dynamics diverge from the forward trajectory

---

3. Limited Expressiveness Under Lipschitz Constraints

To ensure stable ODE integration:

- The vector field $f(z, t)$ often needs to be **Lipschitz-controlled**
- Overly smooth vector fields may limit expressivity
- Sharp transitions, discontinuities, or multimodal densities are harder to represent

---

4. Difficulty Modeling Very Complex Distributions

Even though CNFs are more flexible than discrete flows:

- They still require solving **invertible, continuous** transformations
- Hard to represent highly separated modes
- Struggle with distributions needing non-smooth transport maps

This can result in underfitting on high-dimensional structured data (e.g., images).

---

5. Solver Behavior Depends on Data and Parameters

The ODE solver's path depends on:

- data point
- network parameters
- step size adaptation

Small parameter changes may cause large shifts in solver trajectories, leading to:

- noisy gradients
- training irregularities
- solver-induced variance

---

6. High Deployment Cost

At inference time:

- Each sample requires an ODE integration
- Latency is high compared to forward-pass-only models (GANs, VAEs, flows with fixed layers)

Derivation of ODE (just for completeness, an aside)

Consider the ODE flow

$$\frac{dz(t)}{dt} = f(z(t), t), \quad z(t) \in \mathbb{R}^d.$$

Let $p(z(t))$ be the density of $z(t)$. We look at how p changes over a *small* time step Δt.

---

1. Small time step: local map

For small Δt:

$$z' = z(t + \Delta t) = z(t) + \Delta t\, f(z(t), t).$$

This defines a local change of variables
$z \mapsto z' = g(z) = z + \Delta t\, f(z, t)$.

---

2. Jacobian of the local map

The Jacobian is

$$\frac{\partial z'}{\partial z} = I + \Delta t \frac{\partial f(z,t)}{\partial z}.$$

Using $\det(I + \epsilon A) \approx 1 + \epsilon \operatorname{Tr}(A)$ for small ϵ:

$$\det\left(\frac{\partial z'}{\partial z}\right) \approx 1 + \Delta t \operatorname{Tr}\left(\frac{\partial f(z,t)}{\partial z}\right).$$

So

$$\log\det\left(\frac{\partial z'}{\partial z}\right) \approx \Delta t \operatorname{Tr}\left(\frac{\partial f(z,t)}{\partial z}\right).$$

3. Change of variables for densities

Densities satisfy

$$p_{t+\Delta t}(z') = p_t(z) \left|\det\left(\frac{\partial z}{\partial z'}\right)\right|.$$

But

$$\det\left(\frac{\partial z}{\partial z'}\right) = \left(\det\left(\frac{\partial z'}{\partial z}\right)\right)^{-1} \approx 1 - \Delta t \operatorname{Tr}\left(\frac{\partial f}{\partial z}\right).$$

Therefore,

$$\log p_{t+\Delta t}(z') = \log p_t(z) - \log\det\left(\frac{\partial z'}{\partial z}\right) \approx \log p_t(z) - \Delta t \operatorname{Tr}\left(\frac{\partial f}{\partial z}\right).$$

---

4. Take the limit $\Delta t \to 0$

Rewriting:

$$\frac{\log p_{t+\Delta t}(z') - \log p_t(z)}{\Delta t} \approx -\operatorname{Tr}\left(\frac{\partial f}{\partial z}\right).$$

In the limit $\Delta t \to 0$:

$$\boxed{\frac{d}{dt}\log p(z(t)) = -\operatorname{Tr}\left(\frac{\partial f(z(t),t)}{\partial z(t)}\right)}$$

Flow matching

Setup: Transport From p_0 to p_1

We want to learn a flow that transforms a **source distribution** $p_0(x_0)$ into a **target distribution** $p_1(x_1)$.

Key Construction (The Magic of Flow Matching)

We sample:

$$x_0 \sim p_0, \qquad x_1 \sim p_1,$$

completely independently.

This defines the joint:

$$p(x_0, x_1) = p_0(x_0)\, p_1(x_1).$$

Important insight:

There is *no coupling* between x_0 and x_1.

We simply pair **any noise sample** x_0 with **any data sample** x_1.

> As if we are connecting "any noise pattern" with "any image."
> Flow Matching does *not* require aligned pairs.

This independence is what makes flow matching extremely simple and scalable.

A Simple Path Between Two Independent Points

We take the independently sampled pair:

$$x_0 \sim p_0, \qquad x_1 \sim p_1,$$

even though they have *no relationship at all*.

Linear Interpolation Path

Define:

$$x_t = (1-t)x_0 + tx_1, \qquad t \in [0,1].$$

This gives a straight-line path connecting:

- a **random noise point** x_0
- to a **random data point** x_1

regardless of how unrelated they are.

Densities Induced by the Path

Let $p_t(x_t)$ be the distribution of x_t when (x_0, x_1) are drawn independently from p_0 and p_1.

Then:

- As $t \to 0$:

$$x_t \to x_0 \implies p_t(x) \to p_0(x)$$

- As $t \to 1$:

$$x_t \to x_1 \implies p_t(x) \to p_1(x)$$

Thus, from purely **independent** endpoint samples, the linear path defines a **continuous bridge** between source and target distributions:

$$p_0 \longrightarrow p_t \longrightarrow p_1.$$

No coupling.

No alignment.

Just independent sampling and interpolation.

The ODE Dynamics We Want

We want a flow described by an ODE:

$$\frac{dx_t}{dt} = f(x_t, t), \qquad t \in [0, 1],$$

with initial condition

$$x_0 \sim p_0.$$

Desired Behavior

As the ODE evolves:

$$x_t \sim p_t,$$

where p_t is the density induced by the interpolation

$$x_t = (1-t)x_0 + tx_1, \quad (x_0, x_1) \sim p_0 p_1.$$

Thus:

- At $t = 0$, $x_t \sim p_0$
- For intermediate t, $x_t \sim p_t$
- At $t = 1$, $x_t \sim p_1$

Goal

> Learn a vector field $f(x, t)$ such that solving the ODE drives samples from the source density p_0 through all intermediate densities p_t, ending at the target p_1.

No coupling between (x_0, x_1).

No alignment.

Just matching the marginal flow $p_0 \to p_t \to p_1$.

Learning the Velocity Field $f(x,t)$

We use the interpolation path:

$$x_t = (1-t)x_0 + tx_1, \qquad (x_0, x_1) \sim p_0 p_1,$$

where x_0 and x_1 are fully **independent**.

1. True Velocity Along the Path

Differentiate the interpolation:

$$\frac{dx_t}{dt} = x_1 - x_0.$$

So for any path sample (x_0, x_1, t):

- We know the position on the path:

$$x_t = (1-t)x_0 + tx_1$$

- We know the exact instantaneous velocity:

$$v_t = x_1 - x_0$$

These are directly observable from sampling.

2. The Ideal Vector Field

What we want to learn is a vector field $f(x,t)$ such that the ODE

$$\frac{dx_t}{dt} = f(x_t, t)$$

reproduces the evolution of the marginals
$p_0 \to p_t \to p_1$.

The required flow field is:

$$\boxed{f^\star(x,t) = \mathbb{E}[x_1 - x_0 \mid x_t = x]}$$

Interpretation:

> At any point x and time t, the correct velocity is the **average direction of all straight-line paths that pass through** x.

This is the core computational idea:
Learning $f(x,t)$ reduces to **learning a conditional expectation**.

Using the Learned Flow to Generate Samples

After training, we have a neural vector field $f_\theta(x, t)$.

Sampling from p_1 is simple:

1. Draw a noise sample:

$$x_0 \sim p_0$$

2. Solve the ODE forward:

$$\frac{dx_t}{dt} = f_\theta(x_t, t), \qquad t : 0 \to 1.$$

3. The endpoint is a sample from the learned target:

$$x_1 \approx p_1.$$

This gives a **deterministic transport map** from noise to data.

No randomness during sampling.

No need for reverse processes.

Just one ODE solve.

Pros and cons of various generative models

Method	Training Objective	Sampling Speed	Likelihood	Sample Quality	Key Limitations
VAE	ELBO (variational)	**Fast** (1 forward pass)	Approximate	Medium	Blurry outputs, posterior gap
GAN	Adversarial loss	**Very Fast**	No	**High**	Mode collapse, instability
Normalizing Flow (NF)	Exact MLE via change-of-vars	**Fast** (invertible map)	**Exact**	Medium	Invertibility/triangular Jacobian limits expressiveness
Continuous NF (CNF)	Exact MLE via ODE + trace	**Medium–Slow** (solve ODE)	**Exact**	Medium–High	ODE cost, adjoint complexity, numerical instability
Diffusion Models	Score matching (denoising)	**Slow** (many denoise steps)	Implicit	**Very High**	Expensive sampling, long reverse chain
Flow Matching (FM)	**Supervised learning** of velocities	**Medium–Fast** (short ODE solve)	Implicit	High	Depends on path choice; ODE still needed