Generative modelling: normalizing flows

Mathurin Massias

SenHub AI Dakar 2025

11/04/2025

https://mathurinm.github.io

Teacher presentation (Mathurin)

- Tenured Researcher at INRIA (French national institute for AI) since 2021
- PhD in Optimization for ML from Institut Polytechnique de Paris (Télécom)
- Work in ML, Optimization, Generative models
- Teaching:
  - Part time teacher at Ecole Polytechnique and Ecole Normale Supérieure since 2019
  - Executive education for BCG & Ecole Polytechnique
- Open source in Python: maintainer of `celer`, `skglm`, `benchopt`

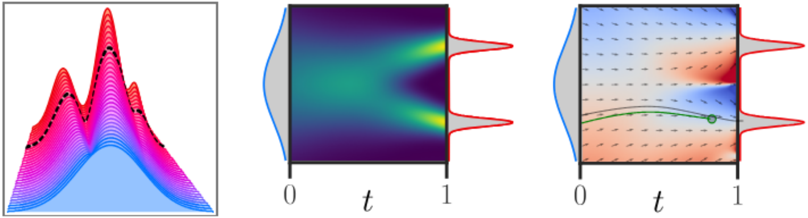
scikit learn skglm Benchopt

`https://mathurinm.github.io/`

Blog post

https://dl.heeere.com/cfm/

"*A Visual Dive into Conditional Flow Matching*", A. Gagneux, S. Martin, R. Emonet, Q. Bertrand, M. Massias
International Conference on Learning Representations (ICLR) 2025 Blog post



Outline

Generative modelling: the big picture

Normalizing flows

Continuous normalizing flows

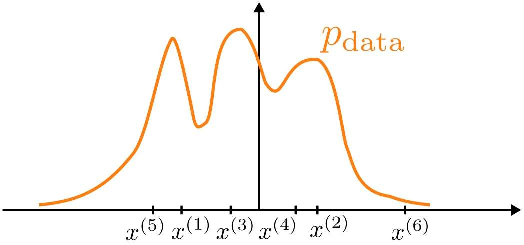
Generative modelling

Given $x^{(1)}, \ldots, x^{(n)}$ sampled from p_{data}, learn to sample from p_{data}

Example:

- $x^{(1)}, \ldots, x^{(n)}$ = real images
- p_{data} = distribution of real images

Main challenges of generative modelling?

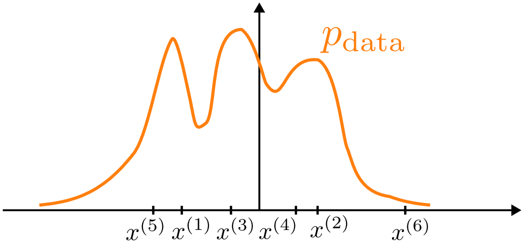


Generative modelling

Given $x^{(1)}, \dots, x^{(n)}$ sampled from p_{data}, learn to sample from p_{data}

Example:

- $x^{(1)}, \dots, x^{(n)}$ = real images
- p_{data} = distribution of real images

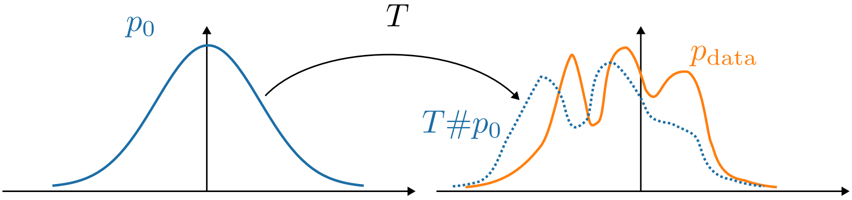


Main challenges of generative modelling?

- enforce fast sampling
- generate high quality samples
- properly cover the diversity of p_{data}

Modern way to do generative modelling

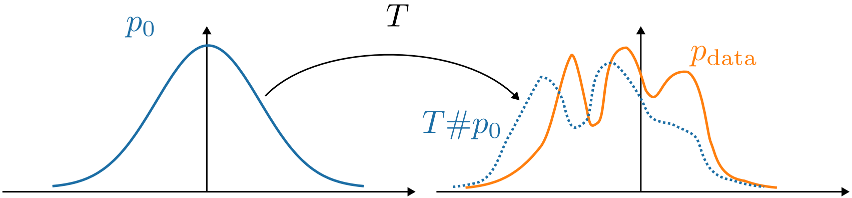
Map **simple** *base distribution* (e.g. Gaussian), p_0, to p_{data} through a map T



Vocabulary: the distribution of $T(x)$ when $x \sim p_0$ is the *pushforward*, $T\#p_0$

Modern way to do generative modelling

Map **simple** *base distribution* (e.g. Gaussian), p_0, to p_{data} through a map T



Vocabulary: the distribution of $T(x)$ when $x \sim p_0$ is the *pushforward*, $T\#p_0$

Why should the base distribution be simple?

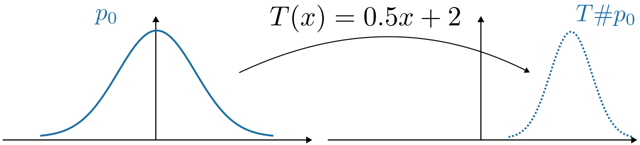
Illustrative example

- In 1D: $x \in \mathbb{R}$
- suppose we only know how to sample from a **standard** Gaussian, $\mathcal{N}(0, 1)$
- we want to generate samples from $\mathcal{N}(a, b^2)$ (Gaussian with mean a, standard deviation b)
- how do we achieve this?

Illustrative example

- In 1D: $x \in \mathbb{R}$
- suppose we only know how to sample from a **standard** Gaussian, $\mathcal{N}(0,1)$
- we want to generate samples from $\mathcal{N}(a,b^2)$ (Gaussian with mean a, standard deviation b)
- how do we achieve this?

$\hookrightarrow$ we sample x from $\mathcal{N}(0,1)$, use $T(x) = a + bx$. Then $T(x) \sim \mathcal{N}(a,b^2)$



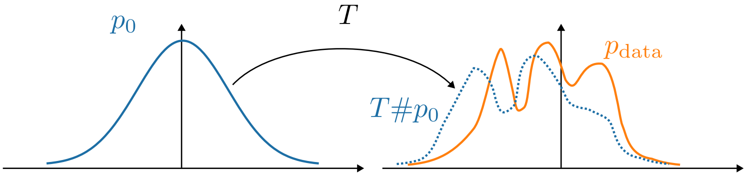
With a more complicated T, we can create more complex distributions $T\#p_0$!

How to find a good T?

Remember our approach:

- sample x from simple distribution (e.g. Gaussian noise)
- the generated image is $T(x)$

Want: $T\#p_0$ close to p_{data}



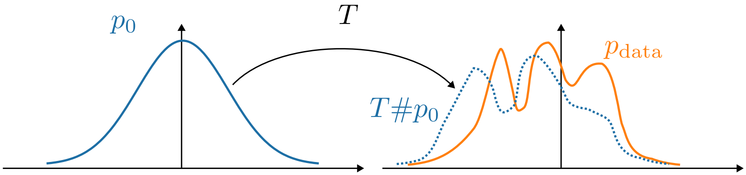
what's the difference with the example in previous slide?

How to find a good T?

Remember our approach:

- sample x from simple distribution (e.g. Gaussian noise)
- the generated image is $T(x)$

Want: $T\#p_0$ close to p_{data}



what's the difference with the example in previous slide?

Big question: "close" in which sense? How could I achieve this?

Outline

Generative modelling: the big picture

Normalizing flows

Continuous normalizing flows

Maximum likelihood detour

- Suppose I flip a coin ten times, and get: HHTHHTTTHTT (5 head, 5 tail)
- Then I ask you to choose between 2 models of the coin:
  - model 1: the coin lands on H with probability 0.1 (T w. proba 0.9)
  - model 2: the coin lands on H with probability 0.5 (T w. proba 0.5)

Which one do you choose? Why?

Maximum likelihood detour

- Suppose I flip a coin ten times, and get: HHTHHTTTHTT (5 head, 5 tail)
- Then I ask you to choose between 2 models of the coin:
    - model 1: the coin lands on H with probability 0.1 (T w. proba 0.9)
    - model 2: the coin lands on H with probability 0.5 (T w. proba 0.5)

Which one do you choose? Why?

- Under model 1, probability of observing said sequence is $0.1^5\, 0.9^5 \approx 6.10^{-6}$
- Under model 2, probability of observing said sequence is $0.5^5\, 0.5^5 \approx 1.10^{-4}$

“The best model is the one that explains the observed data the best”

Maximum likelihood detour

Is there a model under which the observed sequence is even more probable?
= amongst all models, which is the best?

- suppose you observe n results of a coin toss, $y_1, \ldots, y_n \in \{0, 1\}$
- Bernoulli model $\mathbb{P}(y = 1) = p \in [0, 1]$
- is it true that $\mathbb{P}(y = y_i) = p^{y_i}(1 - p)^{1-y_i} \in [0, 1]$?
- for a given p, what is the probability of observing the full observation set $(y_1, \ldots, y_n)$?

Maximum likelihood detour

Is there a model under which the observed sequence is even more probable?
= amongst all models, which is the best?

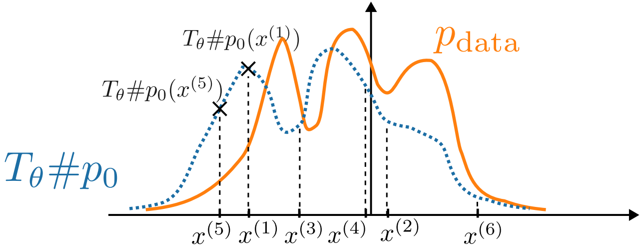
- suppose you observe n results of a coin toss, $y_1, \dots, y_n \in \{0, 1\}$
- Bernoulli model $\mathbb{P}(y = 1) = p \in [0, 1]$
- is it true that $\mathbb{P}(y = y_i) = p^{y_i}(1-p)^{1-y_i} \in [0, 1]$?
- for a given p, what is the probability of observing the full observation set $(y_1, \dots, y_n)$?
- likelihood of the observations (probability to observe $(y_1, \dots, y_n)$): $\prod_1^n p^{y_i}(1-p)^{1-y_i}$
- maximize the likelihood = minimize the negative log likelihood = $-\sum_1^n y_i \log p - \sum_1^n (1 - y_i) \log(1 - p)$
- solution in p?

Back to generative: how to find a good T

- choose T as parametric map: T_θ (examples of T_θ?)
- find best θ by **maximizing the log-likelihood** of available samples:

$$\theta^* = \operatorname*{argmax}_\theta \sum_{i=1}^n \log \left(\underbrace{(T_\theta \# p_0)}_{:=p_1}(x^{(i)}) \right)$$

(links with empirically minimizing the Kullback-Leibler divergence $\mathrm{KL}(p_{\mathrm{data}}, T_\theta \# p_0)$)



How to find a good T: compute the likelihood

$$\theta^* = \operatorname*{argmax}_{\theta} \sum_{i=1}^{n} \log \left(\underbrace{(T_\theta \# p_0)}_{:= p_1} (x^{(i)}) \right)$$

- we have this objective to maximize in θ, but can we actually compute it?

How to find a good T: compute the likelihood

$$\theta^* = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{n} \log \left(\underbrace{(T_\theta \# p_0)}_{:=p_1}(x^{(i)}) \right)$$

- we have this objective to maximize in θ, but can we actually compute it?
- we can rely on the *change of variable formula*:

$$\log p_1(x) = \log p_0(T_\theta^{-1}(x)) + \log |\det J_{T_\theta^{-1}}(x)|$$

How to find a good T: compute the likelihood

$$\theta^* = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{n} \log \left(\underbrace{(T_\theta \# p_0)}_{:= p_1}(x^{(i)}) \right)$$

- we have this objective to maximize in θ, but can we actually compute it?
- we can rely on the *change of variable formula*:

$$\log p_1(x) = \log p_0(T_\theta^{-1}(x)) + \log |\det J_{T_\theta^{-1}}(x)|$$

$J_{T_\theta^{-1}}$ is the *Jacobian* (=matrix of partial derivatives – in 1d: $J_f(x) = f'(x)$)

Exercise: $p_0 = \mathcal{N}(0,1)$, $T_\theta(x) = ax + b$, compute T_θ^{-1}, its derivative, and then p_1

The change of variable formula

$$\boxed{\log p_1(x) = \log p_0(T_\theta^{-1}(x)) + \log |\det J_{T_\theta^{-1}}(x)|}$$

= a mathematical formula to compute the probability of a generated image $T_\theta(x)$

What do I need to use it "practically"?

The change of variable formula

$$\boxed{\log p_1(x) = \log p_0(T_\theta^{-1}(x)) + \log|\det J_{T_\theta^{-1}}(x)|}$$

= a mathematical formula to compute the probability of a generated image $T_\theta(x)$

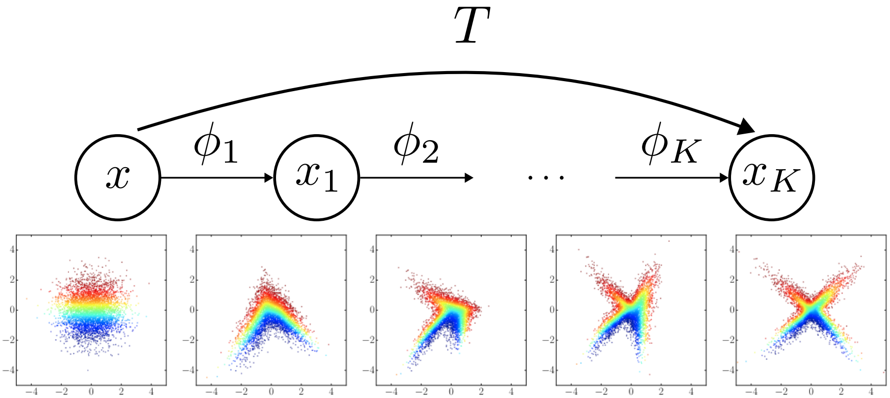
What do I need to use it "practically"?

- T_θ must be invertible
- T_θ^{-1} should be easy to compute in order to evaluate the first right-hand side term
- T_θ^{-1} must be differentiable
- the (log) determinant of the Jacobian of T_θ^{-1} must not be too costly to compute

Normalizing Flows = neural architectures satisfying these requirements

Normalizing flows

- Key observation: If T and T' satisfy the requirements, so does $T \circ T'$
- Build T as composition of simple blocks ϕ_k satisfying the invertibility + Jacobian constraints



Examples of normalizing flows

- planar flow: $\phi_k(x) = x + \sigma(b_k^\top x + c_k)a_k$ (parameters to learn $a_k \in \mathbb{R}^d, b_k \in \mathbb{R}^d, c_k \in \mathbb{R}$)

$$J_{\phi_k}(x) = \mathrm{Id} + \sigma'(b_k^\top x + c_k)a_k b_k^\top$$

id + rank one, all good for the determinant ($\det(\mathrm{Id} + uv^\top) = 1 + v^\top u$)

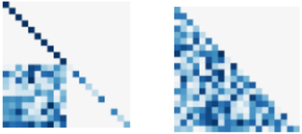
Examples of normalizing flows

- planar flow: $\phi_k(x) = x + \sigma(b_k^\top x + c_k) a_k$ (parameters to learn $a_k \in \mathbb{R}^d, b_k \in \mathbb{R}^d, c_k \in \mathbb{R}$)

$$J_{\phi_k}(x) = \mathrm{Id} + \sigma'(b_k^\top x + c_k) a_k b_k^\top$$

  id + rank one, all good for the determinant ($\det(\mathrm{Id} + uv^\top) = 1 + v^\top u$)

- real NVP (triangular Jacobian, details in blog post)



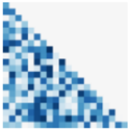
Examples of normalizing flows

- planar flow: $\phi_k(x) = x + \sigma(b_k^\top x + c_k) a_k$ (parameters to learn $a_k \in \mathbb{R}^d, b_k \in \mathbb{R}^d, c_k \in \mathbb{R}$)

$$J_{\phi_k}(x) = \mathrm{Id} + \sigma'(b_k^\top x + c_k) a_k b_k^\top$$

  id + rank one, all good for the determinant ($\det(\mathrm{Id} + uv^\top) = 1 + v^\top u$)

- real NVP (triangular Jacobian, details in blog post)



but **too many constraints** on the architecture, restricts the expressivity

Outline

Generative modelling: the big picture

Normalizing flows

Continuous normalizing flows

From discrete to continuous time: ResNets

Residual Network reminder: from layer ℓ equation

$$x_{\ell+1} = \sigma(W x_\ell + b_\ell)$$

... to

$$x_{\ell+1} = x_\ell + \sigma(W x_\ell + b_\ell)$$

Why does this help?

From discrete to continuous time: ResNets

Residual Network reminder: from layer ℓ equation

$$x_{\ell+1} = \sigma(W x_\ell + b_\ell)$$

... to

$$x_{\ell+1} = x_\ell + \sigma(W x_\ell + b_\ell)$$

Why does this help?

Continuous time limit: neural ODEs

$$\begin{aligned} x_{\ell+1} &= x_\ell + \delta \sigma(W x_\ell + b_\ell) \\ \frac{x_{\ell+1} - x_\ell}{\delta} &= \sigma(W x_\ell + b_\ell) \\ &= f(x_\ell) \end{aligned}$$

Is the last equation reminiscent of something?

From discrete to continuous time

Back to planar flow, idea similar to ResNets:

$$
\begin{aligned}
x_k &= \phi_k(x_{k-1}) \\
&= x_{k-1} + \sigma(b_k^\top x_{k-1} + c)a_k \\
&= x_{k-1} + \frac{1}{K} \underbrace{u_{k-1}(x_{k-1})}_{\text{we define } u_k \text{ like this}}
\end{aligned}
$$

This is an Euler discretization scheme for the ODE

$$
\begin{cases}
x(0) = x_0 \\
\partial_t x(t) = u(x(t), t) \quad \forall t \in [0, 1]
\end{cases}
$$

which is called an *initial value problem* (IVP)

First win: the mapping defined by the ODE, $T(x_0) := x(1)$ is inherently invertible (why?)

Continuous normalizing flows (CNF)

- work in the continuous-time domain: $t \in [0, 1]$
- model the continuous solution $(x(t))_{t\in[0,1]}$ instead of a finite number of discretized steps $x_1, \dots, x_K$
- learn the **velocity field** u as $u_\theta : \mathbb{R}^d \times [0, 1] \to \mathbb{R}^d$
- sample by solving the ODE with $x_0 \sim p_0$

The map T is no longer explicit, it is defined by solving an ODE

Mathematical toolbox: the IVP trifecta

$$\boxed{\begin{aligned} &x(0) = x_0 \\ &\partial_t x(t) = u(x(t), t) \quad \forall t \in [0,1] \end{aligned}}$$

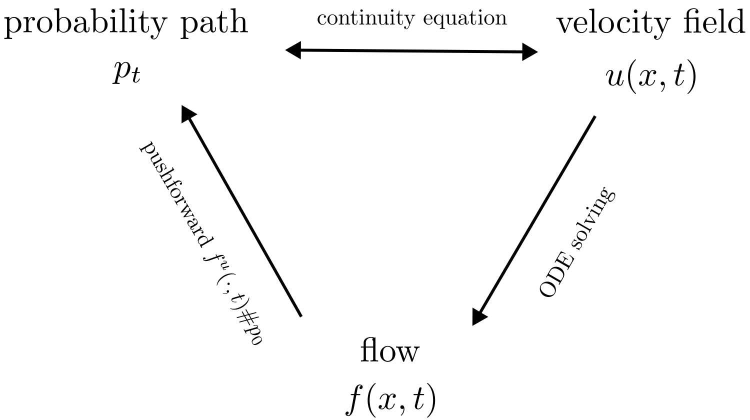
3 objects associated to the initial value problem:

- the **velocity field** $u : \mathbb{R}^d \times [0,1] \to \mathbb{R}^d$
- the **flow** $f^u : \mathbb{R}^d \times [0,1] \to \mathbb{R}^d$: $f^u(x,t)$ = solution at time t to the initial value problem with initial condition $x(0) = x$
- the **probability path** $(p_t)_{t\in[0,1]}$ = the distributions of $f^u(x,t)$ when $x \sim p_0$ ($p_t = f^u(\cdot, t)\#p_0$)

Link: continuity equation

$$\partial_t p_t + \mathrm{div}(u_t p_t) = 0$$

The IVP trifecta



How to learn the velocity u_θ?

- Continuity equation $\implies$ *instantaneous* change of variable formula

$$\frac{\mathrm{d}}{\mathrm{d}t} \log p_t(x(t)) = -\mathrm{tr}\, J_{u_\theta(\cdot,t)}(x(t)) = -\operatorname{div} u_\theta(\cdot,t)(x(t)) \quad \forall t \in [0,1]$$

- allows computing $\log p_1(x^{(i)})$: solving ODE
- nice: avoid computing the full Jacobian with the Hutchinson trace trick (`https://mathurinm.github.io/blog/hutchinson/`)
- constraints on u much less stringent than in discrete NFs: only need unique ODE solution (OK if u Lipschitz in x and cts in t)

Issues of CNFs

- during training, we need to solve ODEs (why?)
- we then need to backpropagate inside an ODE solver $\hookrightarrow$ no black box
- this is terribly unstable

$\hookrightarrow$ this will be solved by Flow Matching: a different way to train CNFs!