Aprendizado Profundo

Normalizing Flows

Lucas Silveira Kupssinskü

Machine Learning Theory and Applications Laboratory
PUCRS

agosto de 2026

Visão Geral

1. Modelos Geradores – Revisão
2. Visão Geral de Normalizing Flows
3. Mudança de Variável
4. Loss
5. O que são Flows
6. Coupling Flows
7. Discreto vs. Contínuo
8. De NFs a *Flow Matching*
9. Panorama Geral

Visão Geral

Modelos Geradores

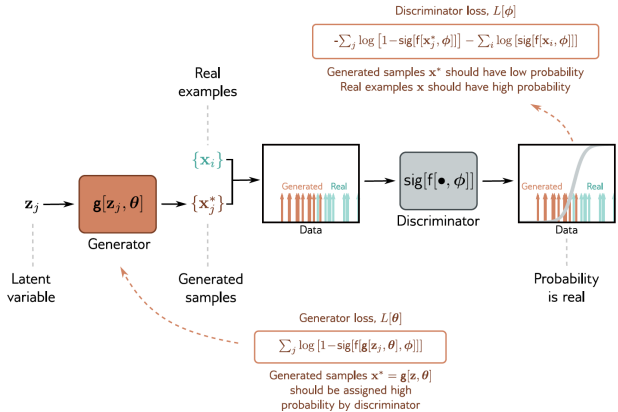
- O objetivo de um **modelo gerador** é aprender uma **distribuição de probabilidade** sobre uma variável aleatória $\mathbf{X}$ usando um conjunto de dados observados $\{x^{(i)}\}_{i=1}^{N}$.
- A densidade $p_X(x)$ é parametrizada por $\boldsymbol{\theta}$.
- Dois usos fundamentais de um modelo gerador:
  - **Amostrar**: obter novos exemplos a partir do modelo.
  - **Avaliar**: calcular $p_X(x)$ para uma instância dada.

Observação

Diferentes famílias de modelos geradores são fortes em um ou outro desses aspectos; poucos fazem os dois bem.

Modelos Geradores – Voltando aos GANs

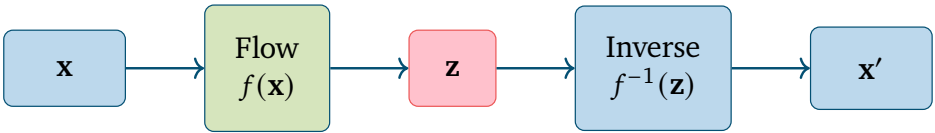
- GANs:
  - **Amostrar** p_X: conseguimos (passagem pelo gerador).
  - **Avaliar** p_X: inviável, não há densidade explícita.
- Isso motiva a busca por modelos que também permitam avaliar p_X.



Visão Geral

Normalizing Flows[1]

- *Normalizing Flows* são modelos geradores probabilísticos que permitem **tanto amostrar quanto avaliar** $p_X(x)$.
- Consistem em **transformar** uma distribuição tratável base (e.g. $z \sim \mathcal{N}(0, 1)$) em uma outra distribuição usando **funções inversíveis**.



[1] Ivan Kobyzev, Simon J. D. Prince e Marcus A. Brubaker. "Normalizing Flows: An Introduction and Review of Current Methods". Em: IEEE TPAMI 43.11 (2020), pp. 3964–3979.

Normalizing Flows – O marco de 2018: Glow[2]

Faces sintetizadas pelo *Glow* (2018), um *Normalizing Flow* com convoluções invertíveis 1×1.

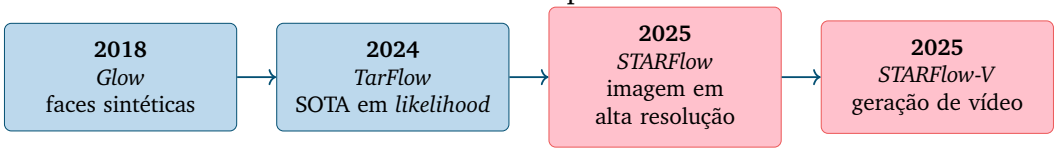
Por muito tempo, a referência visual

O *Glow* foi durante anos o melhor resultado de imagem de um *flow*, firmando a percepção de que NFs ficavam atrás de GANs e modelos difusores. Como veremos a seguir, isso mudou.

[2] Durk P. Kingma e Prafulla Dhariwal. "Glow: Generative Flow with Invertible 1x1 Convolutions". Em: NeurIPS. vol. 31. 2018.

O renascimento dos *Normalizing Flows* (2024–2025)

Trabalhos recentes mostram NFs competindo com modelos difusores:



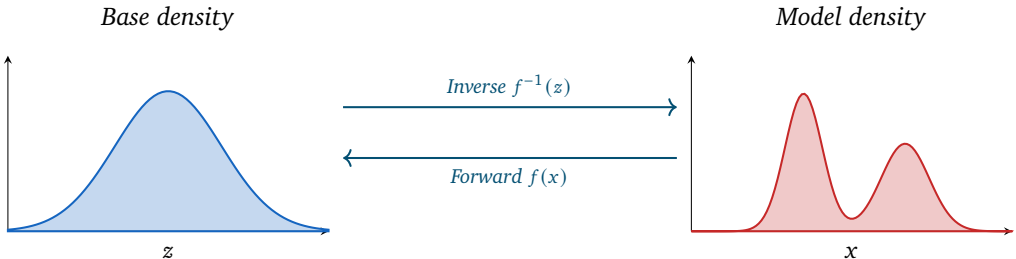
- *TarFlow*[3]: *flow* autorregressivo baseado em *Transformer*. Bate o estado da arte em verossimilhança de imagens e gera amostras comparáveis às de difusão.
- *STARFlow*[4]: escala a ideia para síntese em alta resolução no espaço latente.

[3] Shuangfei Zhai et al. "Normalizing Flows are Capable Generative Models". Em: ICML. 2025.

[4] Jiatao Gu et al. "STARFlow: Scaling Latent Normalizing Flows for High-resolution Image Synthesis". Em: arXiv preprint arXiv:2506.06276 (2025).

Normalizing Flows – Transformação 1D

Gaussiana base → distribuição bimodal via *flow*:



- *Forward* f: leva os dados $p(x)$ para a base $p(z)$ (avaliação via mudança de variável).
- *Inverse* f^{-1}: faz o caminho oposto, de $p(z)$ para $p(x)$ (amostragem).

Visão Geral

Mudança de Variável – Caso discreto

Mudança de variável no caso discreto é **trivial**. Considere o lançamento de um dado:

$$p_X(x = 1) = \frac{1}{6}$$

- Aplicando uma função inversível, por exemplo $f(x) = x^2$, o espaço amostral do experimento muda:

$$\{1, 2, 3, 4, 5, 6\} \longrightarrow \{1, 4, 9, 16, 25, 36\}$$

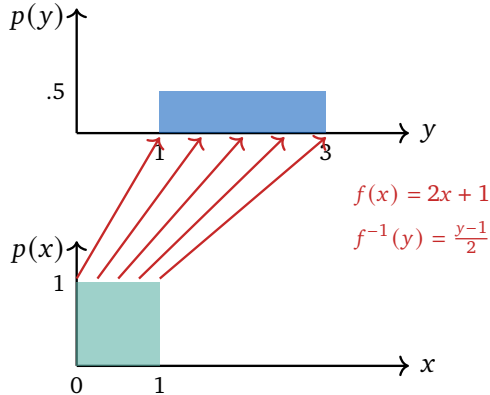
- Para calcularmos $p_Z(z = 9)$, basta perceber que $z = 9 \equiv x = 3$: **as probabilidades são iguais**.

Caso discreto

No discreto, cada evento mantém sua massa de probabilidade: só mudam os rótulos.

Mudança de Variável – Caso contínuo

- No contínuo, a noção fundamental passa a ser **intervalo de probabilidade**, não evento pontual.
- Ao aplicar uma função **inversível e derivável**, a densidade sofre uma **deformação**.
- A massa em um volume pode ser achatada ou alongada ao passar pela transformação.
- A altura da densidade é **inversamente proporcional** ao alargamento: área total sempre = 1.



Uniforme $p(x)$ transformada em $p(y)$ mais larga e menos alta por $y = 2x + 1$.

Mudança de Variável – Equação base

A conservação de probabilidade exige que infinitesimais de probabilidade se preservem:

$$p(x)\,dx = p(z)\,dz$$

- Isso é a lei fundamental da mudança de variável em densidades.
- Intuitivamente: a área sob a curva em um intervalo transformado deve ser igual à área no intervalo original.

Mudança de Variável – Fórmula explícita

Reorganizando, expressamos $p(x)$ em função de $p(z)$:

$$p(x)\,dx = p(z)\,dz \quad \Longrightarrow \quad p(x) = p(z)\left|\frac{dz}{dx}\right|$$

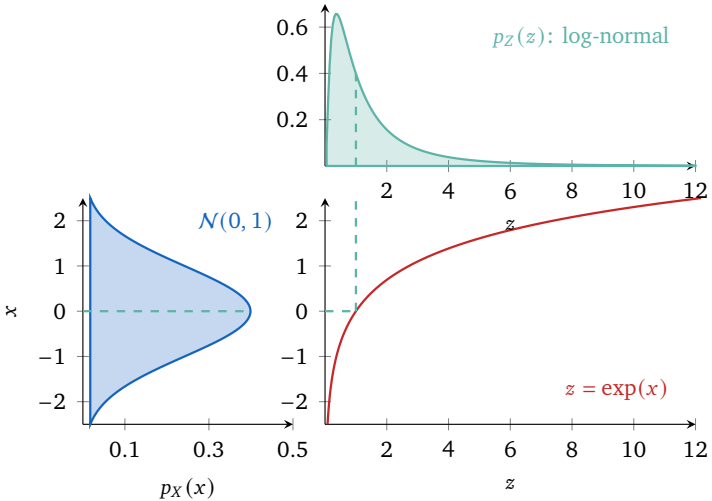
Por que o módulo?

Garante que:

- o resultado seja **positivo** (densidades são não negativas);
- não precisamos nos preocupar com direções de integração/derivação.

Mudança de Variável – Exemplo: log-normal

Setup: $X \sim \mathcal{N}(0,1)$ e $Z = \exp(X)$. Qual é a densidade $p_Z(z)$?



Mudança de Variável – Solução log-normal

Passos

1. Densidade de X: $p_X(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right)$.
2. Como $z = \exp(x)$, a inversa é $x = \ln(z)$.
3. $P(Z \le z) = P(X \le \ln z)$, logo $\mathrm{CDF}_Z(z) = \mathrm{CDF}_X(\ln z)$.
4. Derivando: $p_Z(z) = p_X(\ln z)\, \dfrac{\partial(\ln z)}{\partial z}$.
5. Substituindo $p_X(\ln z)$ e $\dfrac{\partial(\ln z)}{\partial z} = \dfrac{1}{z}$:

$$p_Z(z) = \underbrace{\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(\ln z)^2}{2}\right)}_{p_X(\ln z)} \cdot \underbrace{\frac{1}{z}}_{\partial(\ln z)/\partial z} = \frac{1}{z\sqrt{2\pi}} \exp\left(-\frac{(\ln z)^2}{2}\right).$$

6. Essa é a densidade **log-normal**.

Mudança de Variável – Caso multidimensional

Em mais de uma dimensão, a derivada $\left|\frac{dz}{dx}\right|$ vira o **módulo do determinante da matriz Jacobiana**:

$$p_X(\mathbf{x}) = p_Z(\mathbf{z}) \left|\det \frac{\partial \mathbf{z}}{\partial \mathbf{x}}\right| = p_Z(\mathbf{z}) \left|\det \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}\right|, \qquad \mathbf{z} = f(\mathbf{x}).$$

- O caso 1D já visto é um caso particular (determinante de uma matriz 1×1).
- Em logaritmo (forma que aparece na loss a seguir):

$$\log p_X(\mathbf{x}) = \log p_Z(f(\mathbf{x})) + \log \left|\det \frac{\partial f(\mathbf{x}, \boldsymbol{\theta})}{\partial \mathbf{x}}\right|.$$

- Treinar = empurrar os dados por f até $\mathbf{z}$ e acumular o **log-determinante**.

Visão Geral

Loss – *Maximum Likelihood*

Para treinar, seguimos o *framework* de maximizar a **verossimilhança** dos dados:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \left[\prod_{i=1}^{N} P\left(x^{(i)} \mid \boldsymbol{\theta}\right) \right]$$

- O produto vem da suposição de independência das amostras.
- Usualmente trabalhamos com a log-verossimilhança (soma, não produto) e a minimizamos com sinal trocado.

Loss – NLL passo a passo (1/2)

Passo 1: trocamos produto por soma negativa (NLL):

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \left[\prod_{i} P(x^{(i)} \mid \boldsymbol{\theta}) \right]$$

Produtos de muitas probabilidades pequenas sofrem *underflow* numérico. Aplicando o logaritmo, o produto vira soma e a otimização fica estável.

Loss – NLL passo a passo (1/2)

Passo 1: trocamos produto por soma negativa (NLL):

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \left[\prod_i P(x^{(i)} \mid \boldsymbol{\theta}) \right]$$

$$= \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \left[\sum_{i=1}^{N} -\log P\left(x^{(i)} \mid \boldsymbol{\theta}\right) \right]$$

Mudança de variável do *flow*

$$p_X(x) = p_Z(z) \left| \det \frac{\partial f(x, \boldsymbol{\theta})}{\partial x} \right|, \quad z = f(x, \boldsymbol{\theta})$$

Vamos substituir $P(x^{(i)} \mid \boldsymbol{\theta})$ na expressão acima.

Loss – NLL passo a passo (2/2)

Passo 2: substituir p_X pela mudança de variável:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \left[\sum_{i=1}^{N} -\log\left(P(z^{(i)}) \left| \det \frac{\partial f(x^{(i)}, \boldsymbol{\theta})}{\partial x^{(i)}} \right| \right) \right]$$

Loss – NLL passo a passo (2/2)

Passo 2: substituir p_X pela mudança de variável:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \left[\sum_{i=1}^{N} -\log\left(P(z^{(i)}) \left| \det \frac{\partial f(x^{(i)}, \boldsymbol{\theta})}{\partial x^{(i)}} \right| \right) \right]$$

$$= \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \left[\sum_{i=1}^{N} \left[-\log \left| \det \frac{\partial f(x^{(i)}, \boldsymbol{\theta})}{\partial x^{(i)}} \right| - \log P(z^{(i)}) \right] \right]$$

Dois termos, dois papéis

- $-\log \left| \det \frac{\partial f}{\partial x} \right|$: depende de $\boldsymbol{\theta}$ via a rede. Mede a deformação local de volume induzida pelo *flow*.
- $-\log P(z)$: a distribuição base é fixa (gaussiana), com densidade em forma fechada. $\boldsymbol{\theta}$ entra apenas através de $z = f(x, \boldsymbol{\theta})$.

Visão Geral

Flows

Um *flow* é uma função $f(\mathbf{x})$ com quatro propriedades essenciais:

- **Paramétrica**: queremos definir parâmetros aprendíveis.
- **Inversível**: queremos calcular o *flow* nos dois sentidos (amostragem e avaliação).
- **Diferenciável**: queremos otimizar via *backprop*.
- **Jacobiana tratável**: conseguimos computar a inversa e o determinante da Jacobiana de **maneira eficiente**.

Desafio de engenharia

Satisfazer os quatro requisitos simultaneamente é o que define um *flows*.

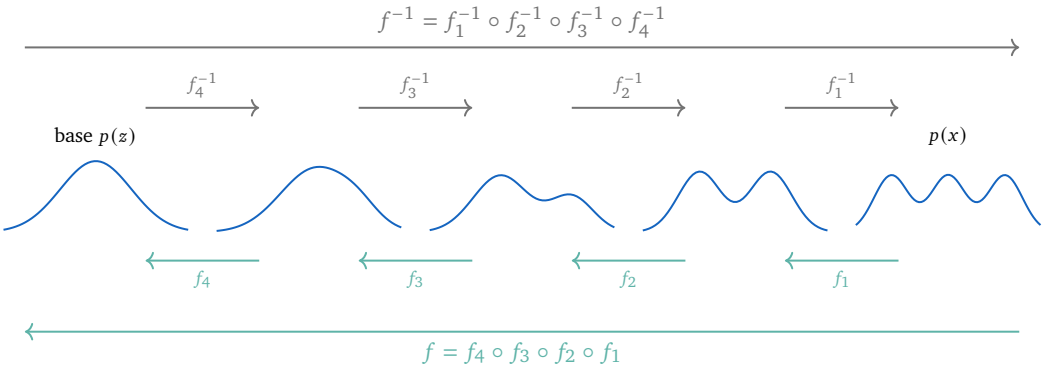
Flows – Composição

A classe de funções diferenciáveis e inversíveis é **fechada em composição**:

$$f = f_k \circ f_{k-1} \circ \cdots \circ f_2 \circ f_1$$

- Isso nos possibilita **projetar camadas** que computam um *flow* e empilhá-las em redes profundas.
- A expressividade vem da composição: cada camada pode ser simples desde que a composição seja rica.

Flows – Caminho direto e inverso



- Normalização ($f = f_4 \circ f_3 \circ f_2 \circ f_1$): leva os dados $p(x)$ à base $p(z)$ (verde).
- Geração ($f^{-1} = f_1^{-1} \circ f_2^{-1} \circ f_3^{-1} \circ f_4^{-1}$): o caminho inverso, da base $p(z)$ aos dados $p(x)$ (cinza).

Flows – Jacobiana da composição

Pela regra da cadeia, o determinante da Jacobiana de uma composição é o **produto dos determinantes**:

$$\left|\frac{\partial f[z, \boldsymbol{\theta}]}{\partial z}\right| = \left|\frac{\partial f_k[f_{k-1}, \boldsymbol{\theta}_k]}{\partial f_{k-1}}\right| \cdot \left|\frac{\partial f_{k-1}[f_{k-2}, \boldsymbol{\theta}_{k-1}]}{\partial f_{k-2}}\right| \cdots \left|\frac{\partial f_1[z, \boldsymbol{\theta}_1]}{\partial z}\right|$$

- Por isso, basta que cada camada tenha Jacobiana fácil de calcular para que a rede inteira também tenha.
- Esse é o princípio que guia o design de arquiteturas de *flows*.

Linear Flow

Uma transformação **linear** pode ser um *Linear Flow* se a matriz admitir inversa:

$$f(\mathbf{x}) = \boldsymbol{\theta}\mathbf{x} + \mathbf{b} \qquad f^{-1}(\mathbf{z}) = \boldsymbol{\theta}^{-1}(\mathbf{z} - \mathbf{b})$$

Problemas

- **Pouco expressiva**: composição de transformações lineares continua linear.
- Determinante e inversa de matriz densa: custo $O(d^3)$.

Linear Flow – Custo por estrutura

Impondo estrutura na matriz $\boldsymbol{\theta}$, reduzimos drasticamente o custo de inverter e de calcular o determinante. Diagonal é barata mas muito restritiva; LU factorized e 1×1 convolution (Glow) são os melhores compromissos na prática.

Estrutura	Inversa	Determinante
Full	$\mathcal{O}(d^3)$	$\mathcal{O}(d^3)$
Diagonal	$\mathcal{O}(d)$	$\mathcal{O}(d)$
Triangular	$\mathcal{O}(d^2)$	$\mathcal{O}(d)$
Block Diagonal	$\mathcal{O}(c^3 d)$	$\mathcal{O}(c^3 d)$
LU Factorized	$\mathcal{O}(d^2)$	$\mathcal{O}(d)$
Spatial Convolution	$\mathcal{O}(d \log d)$	$\mathcal{O}(d)$
1×1 Convolution	$\mathcal{O}(c^3 + c^2 d)$	$\mathcal{O}(c^3)$

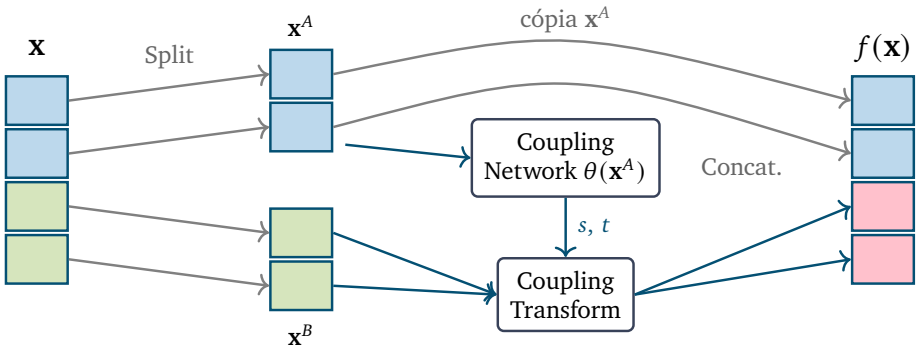
LU *factorized* e *spatial*/1×1 convolution: Kingma & Dhariwal (2018); Hoogeboom et al. (2019). d: dimensão; c: canais.

Visão Geral

Coupling Flows[5]

Dividir a entrada em duas metades e transformar apenas uma delas, condicionando na outra:

$$f(\mathbf{x}) = \begin{bmatrix} \mathbf{x}^A \\ \hat{f}\left(\mathbf{x}^B \mid \theta(\mathbf{x}^A)\right) \end{bmatrix}$$

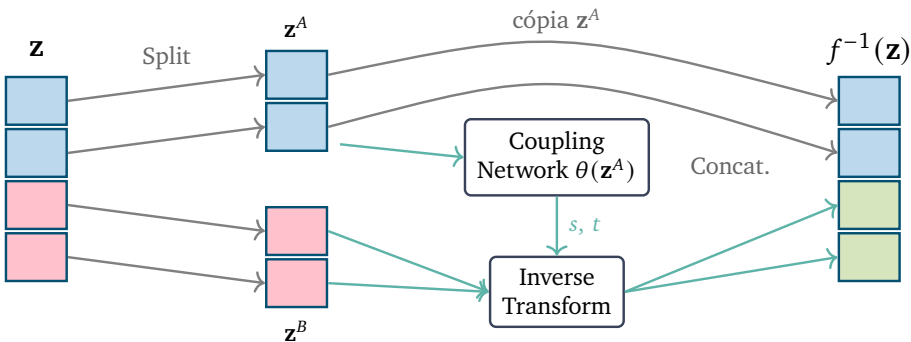


[5] Laurent Dinh, Jascha Sohl-Dickstein e Samy Bengio. "Density Estimation using Real NVP". Em: ICLR. 2017.

Coupling Flows – Inversa

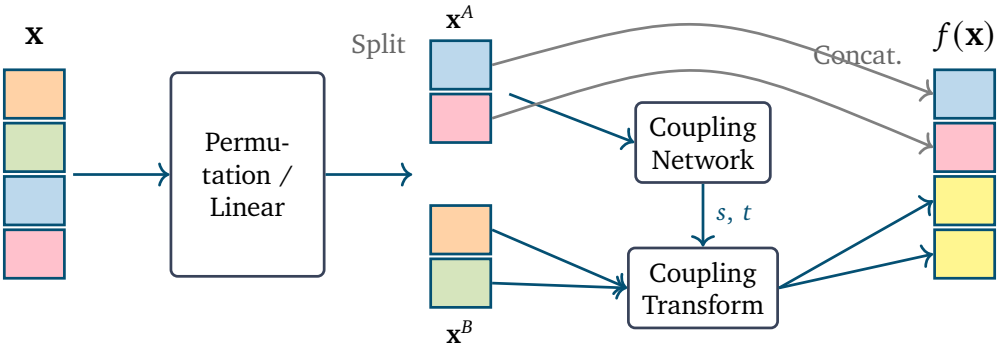
Como $\mathbf{x}^A$ é copiado direto, temos $\mathbf{z}^A = \mathbf{x}^A$. Para inverter basta aplicar $\hat{f}^{-1}$ em $\mathbf{z}^B$ usando a mesma *coupling network* que produziu os parâmetros:

$$f^{-1}(\mathbf{z}) = \begin{bmatrix} \mathbf{z}^A \\ \hat{f}^{-1}\left(\mathbf{z}^B \mid \theta(\mathbf{z}^A)\right) \end{bmatrix}$$



Coupling Flows – Combinando com permutações

Um único *coupling layer* só transforma metade das dimensões. **Alternando permutações** (ou *linear flows*) entre as camadas, todas as dimensões são afetadas:



Cada camada: permutação → split → coupling transform → concat.

Coupling Flows – Jacobiana por blocos

Por que essa arquitetura? Porque a Jacobiana é **triangular por blocos**:

Lembrete: como ler a Jacobiana

A linha i reúne as derivadas parciais da i-ésima componente da saída, e a coluna j reúne as derivadas em relação à j-ésima componente da entrada. Em símbolos, $[Df]_{ij} = \partial f_i / \partial x_j$.

$$Df(\mathbf{x}) = \begin{matrix} \\ \mathbf{x}^A \\ \hat{f} \end{matrix} \begin{matrix} \begin{matrix} \mathbf{x}^A & \mathbf{x}^B \end{matrix} \\ \begin{pmatrix} \frac{\partial \mathbf{x}^A}{\partial \mathbf{x}^A} & \frac{\partial \mathbf{x}^A}{\partial \mathbf{x}^B} \\ \frac{\partial \hat{f}}{\partial \mathbf{x}^A} & \frac{\partial \hat{f}}{\partial \mathbf{x}^B} \end{pmatrix} \end{matrix} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \frac{\partial \hat{f}}{\partial \mathbf{x}^A} & \frac{\partial \hat{f}}{\partial \mathbf{x}^B} \end{bmatrix}$$

- $\partial \mathbf{x}^A / \partial \mathbf{x}^A = \mathbf{I}$: bloco superior-esquerdo (identidade).
- $\partial \mathbf{x}^A / \partial \mathbf{x}^B = \mathbf{0}$: bloco superior-direito (pois $\mathbf{x}^A$ é só copiado, não depende de $\mathbf{x}^B$).

Coupling Flows – Exemplo concreto em 4D

Seja $\mathbf{x} = (x_1, x_2, x_3, x_4)$ com $\mathbf{x}^A = (x_1, x_2)$ e $\hat{f}(\mathbf{x}^B) = (g_1(x_1, x_2) \cdot x_3,\ g_2(x_1, x_2) \cdot x_4)$. Rotulamos as colunas pela entrada x_j e as linhas pela saída $f_i(\mathbf{x})$:

$$Df = \begin{array}{c} \\ f_1 \\ f_2 \\ f_3 \\ f_4 \end{array} \begin{array}{c} \begin{array}{cccc} x_1 & x_2 & x_3 & x_4 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \frac{\partial g_1}{\partial x_1} x_3 & \frac{\partial g_1}{\partial x_2} x_3 & g_1 & 0 \\ \frac{\partial g_2}{\partial x_1} x_4 & \frac{\partial g_2}{\partial x_2} x_4 & 0 & g_2 \end{pmatrix} \end{array}$$

Determinante via triangularidade

$$\det Df = 1 \cdot 1 \cdot g_1 \cdot g_2 = g_1 \cdot g_2$$

Basta o **produto dos elementos diagonais do bloco inferior-direito**. Os termos da linha inferior à esquerda **não importam** para o determinante.

Coupling Flows – Determinante geral

Generalizando, o determinante da Jacobiana é:

$$|\det Df| = \left|\det \frac{\partial \hat{f}(\mathbf{x}^B | \theta(\mathbf{x}^A))}{\partial \mathbf{x}^B}\right|$$

- Só depende do bloco inferior-direito.
- Se $\hat{f}$ for aplicada **elemento a elemento**, esse bloco é **diagonal** e o determinante é trivial.
- Consequência: todos os modelos modernos de *flow* para imagens (*RealNVP*, *Glow*, *Flow++*) usam alguma variante dessa ideia.

Affine Coupling Flows[6] – *Forward*

A escolha mais comum de $\hat{f}$ é uma **transformação afim** elemento a elemento:

Forward pass

$$\mathbf{y}^B = \mathbf{x}^B \odot \exp\big(\mathbf{s}(\mathbf{x}^A)\big) + \mathbf{t}(\mathbf{x}^A), \qquad \mathbf{y}^A = \mathbf{x}^A.$$

- A coupling network $\theta(\mathbf{x}^A)$ produz **dois vetores**: $\mathbf{s}$ (log-escala) e $\mathbf{t}$ (translação).
- $\exp(\cdot)$ garante escala positiva e permite expressar alongamento ($s > 0$) e compressão ($s < 0$).
- Jacobiana **diagonal**: $\det = \prod_i \exp(s_i)$, logo $\log|\det| = \sum_i s_i$ (barato).

[6] Laurent Dinh, Jascha Sohl-Dickstein e Samy Bengio. "Density Estimation using Real NVP". Em: ICLR. 2017.

Affine Coupling Flows – Inverse

A inversa é **quase de graça**: note que a coupling network usa apenas $\mathbf{x}^A$, que é copiado para $\mathbf{y}^A$. Então podemos **reaplicá-la** na direção inversa:

Inverse pass

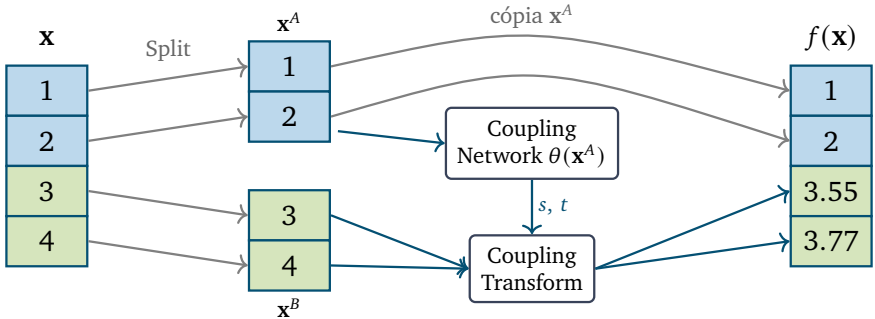
1. $\mathbf{x}^A = \mathbf{y}^A$ (cópia direta).
2. Calcule $\mathbf{s} = \mathbf{s}(\mathbf{x}^A)$, $\mathbf{t} = \mathbf{t}(\mathbf{x}^A)$ usando a mesma rede.
3. $\mathbf{x}^B = (\mathbf{y}^B - \mathbf{t}) \odot \exp(-\mathbf{s})$.

Crucial

A *coupling network* pode ser **qualquer** rede neural (não precisa ser inversível!) – porque ela nunca é invertida, só reavaliada no lado correto.

Exemplo Numérico: *Forward* (1/2)

Vetor de features 4D $\mathbf{x} = (1, 2, 3, 4)$. O *split* separa $\mathbf{x}^A = (1, 2)$, que é **copiado**, de $\mathbf{x}^B = (3, 4)$, que é **transformado**. A coupling network devolve $s = (0.3, -0.2)$ e $t = (-0.5, 0.5)$.

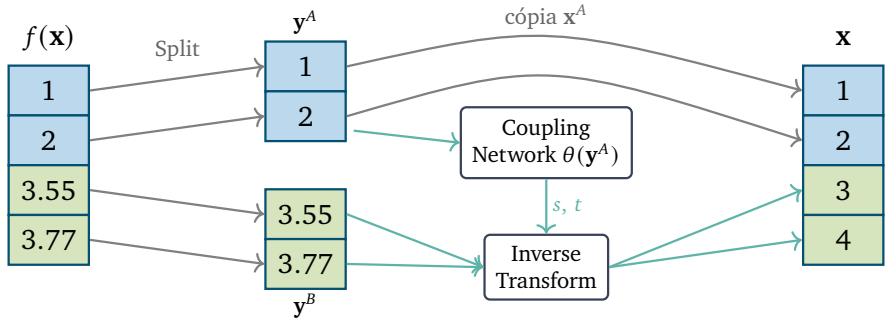


Affine transform, elemento a elemento (cada $y_i^B = x_i^B e^{s_i} + t_i$):

$$y_1^B = 3\,e^{0.3} - 0.5 \approx 3.55, \qquad y_2^B = 4\,e^{-0.2} + 0.5 \approx 3.77, \qquad \log|\det J| = s_1 + s_2 = 0.1.$$

Exemplo Numérico: *Inverse* (2/2)

A inversa parte de $f(\mathbf{x}) = (1, 2, 3.55, 3.77)$. Como $\mathbf{x}^A = \mathbf{y}^A$ já está disponível, a coupling network é reavaliada (nunca invertida) para recuperar s, t e desfazer a transformação em $\mathbf{x}^B$.



Desfazendo a transform, elemento a elemento (cada $x_i^B = (y_i^B - t_i)\, e^{-s_i}$):

$$x_1^B = (3.55 + 0.5)\, e^{-0.3} \approx 3.00, \qquad x_2^B = (3.77 - 0.5)\, e^{0.2} \approx 4.00. \quad \checkmark$$

Affine Coupling Flows – PyTorch

```python
class AffineCoupling(nn.Module):
    def __init__(self, d, hidden=128):
        super().__init__()
        self.half = d // 2
        self.net = nn.Sequential(
            nn.Linear(self.half, hidden), nn.ReLU(),
            nn.Linear(hidden, 2 * self.half),
        )

    def forward(self, x):
        x_a, x_b = x[:, :self.half], x[:, self.half:]
        s, t = self.net(x_a).chunk(2, dim=-1)
        s = torch.tanh(s)
        y_b = x_b * torch.exp(s) + t
        log_det = s.sum(dim=-1)
        return torch.cat([x_a, y_b], dim=-1), log_det

    # inverse: swap + / - and exp(s) / exp(-s)
```

Ideias-chave: split em A/B; rede prediz (s, t) a partir de A; tanh em s estabiliza; `log_det` acumulado é a soma dos s.

Limitações dos *Normalizing Flows*

A exigência de inversibilidade impõe restrições aos NFs:

- **Dimensão preservada**: a bijeção obriga $\dim(\mathbf{z}) = \dim(\mathbf{x})$, logo não há gargalo de compressão como no VAE. A consequência é um custo alto de memória e de compute em imagens, o que motiva arquiteturas *multi-scale* (operação *squeeze*) em RealNVP e Glow.
- **Tensão expressividade $\times$ tratabilidade**: cada camada precisa manter a Jacobiana barata, então a expressividade vem de **empilhar muitas camadas** simples, não de camadas individualmente poderosas.

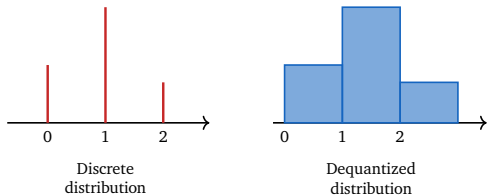
Para onde isso aponta

Custo e escala são exatamente o que os *flows* contínuos (*flow matching*) e as arquiteturas de 2024–2025 atacam, como veremos a seguir.

Visão Geral

Discreto vs. Contínuo – *Dequantization*

- Como **imagens são quantizadas** (pixels inteiros 0–255), podemos ter problemas de singularidades ao ajustar uma densidade contínua.
- Solução: adicionar **ruído**, usualmente uniforme em $[0, 1)$, à distribuição modelada.
- Isso transforma picos discretos em uma densidade contínua passível de modelagem.



Visão Geral

Flows contínuos[8]

E se, em vez de empilhar um número finito de camadas, deixássemos a profundidade tender ao contínuo? O *flow* passa a ser indexado por um **tempo** t e definido por uma **equação diferencial**:

$$\frac{d\mathbf{z}(t)}{dt} = f(\mathbf{z}(t), t, \boldsymbol{\theta})$$

- Amostrar e avaliar viram **integrar a ODE** (*Neural ODE*, *FFJORD*[7]).
- A mudança de variável também vira contínua: o log-determinante se transforma em um **traço** integrado no tempo:

$$\frac{\partial \log p(\mathbf{z}(t))}{\partial t} = -\operatorname{tr}\left(\frac{\partial f}{\partial \mathbf{z}(t)}\right)$$

- Vantagem: não precisamos mais projetar cada camada para ter Jacobiana tratável. A desvantagem é o custo de integrar a ODE durante o treino.

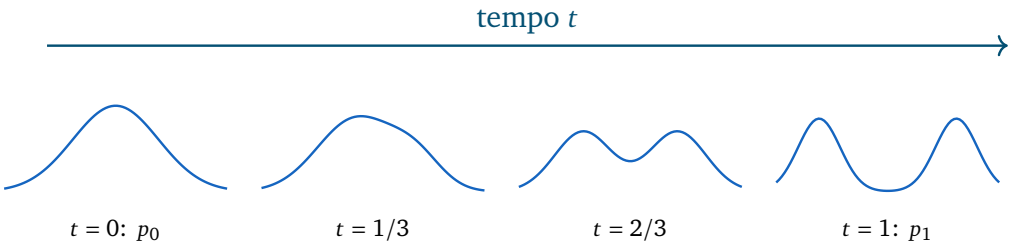
[7] Will Grathwohl et al. "FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models". Em: ICLR. 2019.

[8] Ricky T. Q. Chen et al. "Neural Ordinary Differential Equations". Em: NeurIPS. 2018.

Flow Matching – Caminho de probabilidade[9]

Integrar a ODE a cada passo de treino é caro. A ideia que destravou os *flows* contínuos foi treinar **sem simular** a ODE (*simulation-free*). Para isso, partimos de um **caminho de probabilidade** $p_t(\mathbf{x})$, $t \in [0, 1]$, ligando a base aos dados:

$$p_0 = \mathcal{N}(\mathbf{0}, \mathbf{I}) \ \text{(ruído)}, \qquad p_1 = p_{\text{dados}}.$$



- O modelo aprende a **transportar** amostras ao longo desse caminho, de p_0 até p_1.
- Resta definir como se faz esse transporte.

[9] Yaron Lipman et al. "Flow Matching for Generative Modeling". Em: ICLR. 2023.

Flow Matching – Campo vetorial alvo

O transporte é descrito por um **campo vetorial** $u_t(\mathbf{x})$ que gera o caminho p_t via uma ODE:

$$\frac{d\mathbf{x}_t}{dt} = u_t(\mathbf{x}_t), \qquad \mathbf{x}_0 \sim p_0.$$

- Pense em u_t como o vento que carrega cada partícula $\mathbf{x}_t$. Integrando de $t = 0$ a $t = 1$, a partícula sai de p_0 e chega em p_1.
- Se aprendermos uma rede $v_{\boldsymbol{\theta}}(\mathbf{x}, t) \approx u_t(\mathbf{x})$, basta integrar $\dot{\mathbf{x}}_t = v_{\boldsymbol{\theta}}(\mathbf{x}_t, t)$ para amostrar.

Objetivo natural: *Flow Matching*

$$\mathcal{L}_{\mathrm{FM}}(\boldsymbol{\theta}) = \mathbb{E}_{t,\, \mathbf{x}_t \sim p_t} \left\| v_{\boldsymbol{\theta}}(\mathbf{x}_t, t) - u_t(\mathbf{x}_t) \right\|^2 .$$

Problema

Não conhecemos u_t em forma fechada para um caminho marginal arbitrário, então essa *loss* **não é usável** diretamente.

Conditional Flow Matching

Truque: fixe uma amostra $\mathbf{x}_1$ dos dados e defina:

- um **caminho condicional** $p_t(\mathbf{x} \mid \mathbf{x}_1)$ (por exemplo, uma gaussiana centrada em $\mathbf{x}_1$ que encolhe com t);
- um **campo condicional** $u_t(\mathbf{x} \mid \mathbf{x}_1)$ que o gera.

Os dois têm **forma fechada** e são fáceis de amostrar. Os marginais se recuperam integrando sobre $\mathbf{x}_1 \sim p_{\text{dados}}$. A *loss* condicional substitui o alvo desconhecido pelo condicional:

$$\mathcal{L}_{\text{CFM}}(\boldsymbol{\theta}) = \mathbb{E}_{t,\,\mathbf{x}_1,\,\mathbf{x}_t \sim p_t(\cdot \mid \mathbf{x}_1)} \left\| v_{\boldsymbol{\theta}}(\mathbf{x}_t, t) - u_t(\mathbf{x}_t \mid \mathbf{x}_1) \right\|^2 .$$

Teorema (Lipman et al., 2023)

$\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\text{CFM}} = \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\text{FM}}$: os dois objetivos têm o **mesmo gradiente**. Treinamos com o alvo condicional e, no ótimo, $v_{\boldsymbol{\theta}}$ aproxima o campo marginal u_t.

Rectified Flow – Caminhos retos[10]

Escolha mais simples possível para o caminho condicional: interpolação **linear** entre ruído $\mathbf{x}_0 \sim p_0$ e dado $\mathbf{x}_1 \sim p_{\text{dados}}$:

$$\mathbf{x}_t = (1-t)\,\mathbf{x}_0 + t\,\mathbf{x}_1.$$

- Derivando no tempo, $\dfrac{d\mathbf{x}_t}{dt} = \mathbf{x}_1 - \mathbf{x}_0$, **constante** na trajetória.
- O alvo condicional é o próprio vetor de deslocamento:

$$u_t(\mathbf{x}_t \mid \mathbf{x}_0, \mathbf{x}_1) = \mathbf{x}_1 - \mathbf{x}_0.$$

- A *loss* colapsa em uma **regressão simples**:

$$\mathcal{L}_{\text{RF}}(\boldsymbol{\theta}) = \mathbb{E}_{t,\,\mathbf{x}_0,\,\mathbf{x}_1} \left\| v_{\boldsymbol{\theta}}(\mathbf{x}_t, t) - (\mathbf{x}_1 - \mathbf{x}_0) \right\|^2.$$

Por que retas importam

Trajetórias retas são o transporte de menor comprimento, então a ODE de amostragem precisa de **poucos passos** (Euler com K pequeno já gera boas amostras).

[10] Xingchao Liu, Chengyue Gong e Qiang Liu. "Flow Straight and Fast: Learning to Generate and Transfer Data with Rectified Flow". Em: ICLR. 2023.

Rectified Flow – Treino e amostragem

Treino (por *minibatch*)

1. amostre $\mathbf{x}_0 \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, $\mathbf{x}_1 \sim p_{\text{dados}}$, $t \sim \mathcal{U}[0,1]$;
2. compute $\mathbf{x}_t = (1-t)\,\mathbf{x}_0 + t\,\mathbf{x}_1$;
3. prediga $\hat{\mathbf{v}} = v_{\boldsymbol{\theta}}(\mathbf{x}_t, t)$;
4. minimize $\|\hat{\mathbf{v}} - (\mathbf{x}_1 - \mathbf{x}_0)\|^2$.

Amostragem (Euler, K passos)

1. $\mathbf{x} \leftarrow \mathbf{x}_0 \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$;
2. para $k = 0, 1, \ldots, K-1$:

$$\mathbf{x} \leftarrow \mathbf{x} + \tfrac{1}{K}\, v_{\boldsymbol{\theta}}\Big(\mathbf{x},\ \tfrac{k}{K}\Big);$$

3. retorne $\mathbf{x}$ como amostra de p_1.

- Difusão clássica costuma usar $K \approx 1000$. Com *rectified flow*, $K \in [4, 25]$ já produz amostras competitivas.
- Nenhuma simulação de ODE durante o treino, **simulation-free**.

A ideia de *flow* no estado da arte: SD3 e Flux

- *Rectified flow* é o **objetivo de treino** por trás de modelos como **Stable Diffusion 3**[11] e **Flux**.
- A mesma intuição dos NFs, transformar uma distribuição base simples em dados por uma aplicação aprendida, está no **núcleo** dos geradores de imagem mais capazes de hoje.
- A fronteira entre *normalizing flows* e modelos difusores ficou **difusa**: ambos podem ser vistos como transporte de probabilidade ao longo de uma trajetória.

[11] Patrick Esser et al. "Scaling Rectified Flow Transformers for High-Resolution Image Synthesis". Em: ICML. 2024.

Visão Geral

Normalizing Flows – Aplicações

NFs brilham onde a **densidade tratável** importa e, desde 2024, voltaram a ser competitivos também em **geração**:

Onde a densidade tratável importa

- **Detecção de anomalias**: dados anômalos têm $\log p_X(x)$ baixo (indústria, segurança de redes).
- **Inferência variacional**: substituem $q(z \mid x)$ do VAE por uma distribuição expressiva.
- **Física e química**: *Boltzmann Generators* amostram conformações moleculares em equilíbrio.

Geração de imagem e vídeo

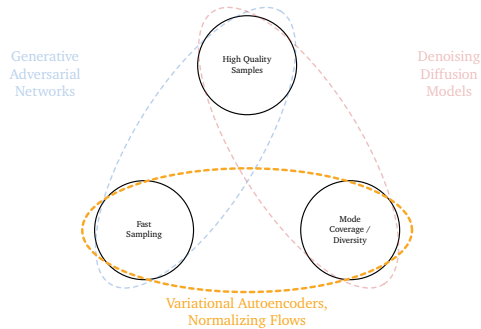
TarFlow, *STARFlow* e *STARFlow-V* (2024–2025) aproximam os NFs da qualidade dos modelos difusores.

Flow matching em modelos SOTA

A formulação contínua (*rectified flow*) treina geradores de ponta como **Stable Diffusion 3** e **Flux**.

Onde NFs ficam no trilema?

- **Amostragem rápida**: ✓ (1 forward).
- **Cobertura / densidade**: ✓ (treino por MLE cobre toda a distribuição).
- **Qualidade visual**: historicamente ×, mas NFs recentes (*TarFlow*, *STARFlow*) melhoraram nesse quesito.



TL;DR

- *Normalizing Flows* são modelos geradores capazes de **amostragem** e **avaliação da probabilidade**.
- São construídos com base em uma série de **transformações invertíveis**, com Jacobianas tratáveis.
- Em 2024–2025, NFs autônomos (*TarFlow*, *STARFlow*) passaram a igualar modelos difusores em vários *benchmarks* de imagem.
- Sua versão contínua, o *flow matching / rectified flow*, está no núcleo de geradores SOTA como SD3 e Flux.

Quando usar NF

Quando precisar de **densidade tratável**: detecção de anomalias, compressão, inferência variacional, física computacional.

Resumo

- *Normalizing Flows*: transformações **inversíveis e diferenciáveis** de uma distribuição base em uma modelada.
- Treinados por **máxima verossimilhança**, usando a fórmula de mudança de variável com o log-determinante da Jacobiana.
- *Linear flows* são limitados; estruturas como LU, 1×1 convolução reduzem custo.
- *Coupling flows* (RealNVP, Glow) dão expressividade com Jacobiana triangular.
- *Dequantization* é necessária para dados discretos (imagens).
- Desde 2024, NFs voltaram a ser competitivos em geração (*TarFlow*, *STARFlow*), e sua versão contínua (*flow matching*) treina modelos SOTA como SD3 e Flux.

Leituras Recomendadas

- Capítulo 16: Simon J. D. Prince. Understanding Deep Learning. MIT Press, 2023
- Ivan Kobyzev, Simon J. D. Prince e Marcus A. Brubaker. “Normalizing Flows: An Introduction and Review of Current Methods”. Em: IEEE TPAMI 43.11 (2020), pp. 3964–3979

Referências I

[1] Ricky T. Q. Chen et al. “Neural Ordinary Differential Equations”. Em: NeurIPS. 2018.

[2] Laurent Dinh, Jascha Sohl-Dickstein e Samy Bengio. “Density Estimation using Real NVP”. Em: ICLR. 2017.

[3] Patrick Esser et al. “Scaling Rectified Flow Transformers for High-Resolution Image Synthesis”. Em: ICML. 2024.

[4] Will Grathwohl et al. “FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models”. Em: ICLR. 2019.

[5] Jiatao Gu et al. “STARFlow: Scaling Latent Normalizing Flows for High-resolution Image Synthesis”. Em: arXiv preprint arXiv:2506.06276 (2025).

[6] Durk P. Kingma e Prafulla Dhariwal. “Glow: Generative Flow with Invertible 1x1 Convolutions”. Em: NeurIPS. Vol. 31. 2018.

[7] Ivan Kobyzev, Simon J. D. Prince e Marcus A. Brubaker. “Normalizing Flows: An Introduction and Review of Current Methods”. Em: IEEE TPAMI 43.11 (2020), pp. 3964–3979.

[8] Yaron Lipman et al. “Flow Matching for Generative Modeling”. Em: ICLR. 2023.

[9] Xingchao Liu, Chengyue Gong e Qiang Liu. “Flow Straight and Fast: Learning to Generate and Transfer Data with Rectified Flow”. Em: ICLR. 2023.

[10] Simon J. D. Prince. Understanding Deep Learning. MIT Press, 2023.

[11] Shuangfei Zhai et al. “Normalizing Flows are Capable Generative Models”. Em: ICML. 2025.