

# Mini-Projeto 1: Neural ODEs e Continuous Normalizing Flows

**Prazo:** 30/11/2025

**Trabalho:** Individual ou duplas

---

## Objetivos de Aprendizagem

Ao completar este projeto, os alunos e alunas devem ser capazes de:

1. **Implementar** Neural ODEs usando `torchdiffeq`
  2. **Compreender** continuous normalizing flows e change of variables
  3. **Aplicar** trace estimation techniques (Hutchinson)
  4. **Analisar** trade-offs computacionais de CNFs vs discrete flows
  5. **Preparar-se** conceitualmente para Flow Matching (Módulo 2)
- 

## Parte 1: Setup e Ambiente

### 1.1 Stack Tecnológico

Referências de código:

- **torchdiffeq**: <https://github.com/rtqichen/torchdiffeq>
- **FFJORD repo** (referência): <https://github.com/rtqichen/ffjord>
  - Apenas para consulta, **não usar diretamente**
  - Entender a implementação, mas escrever seu próprio código

### 1.2 Estrutura do Projeto

```
1 mini-project-1/
2 |— README.md
3 |— requirements.txt
4 |— src/
5 |   |— models/
6 |   |   |— vector_field.py      # VectorField architectures
7 |   |   |— neural_ode.py       # Neural ODE básico
8 |   |   |— cnf.py              # CNF com trace exato
9 |   |   |— ffjord.py           # FFJORD com Hutchinson
10 |   |— utils/
11 |   |   |— datasets.py         # Data loading
12 |   |   |— training.py        # Training loops
13 |   |   |— trace.py           # Trace computation utilities
```

```

14 | | | visualization.py      # Plotting
15 | | | experiments/
16 | | | | exp1_ode_solvers.py
17 | | | | exp2_regularization.py
18 | | | | exp3_architectures.py
19 | | notebooks/
20 | | | 01_neural_ode_2d.ipynb
21 | | | 02_cnf_trace_comparison.ipynb
22 | | | 03_ffjord_mnist.ipynb
23 | | results/
24 | | | figures/
25 | | | checkpoints/

```

## Parte 2: Implementação Progressiva

### Milestone 1: Neural ODE Básico (Semana 1)

**Objetivo:** Implementar Neural ODE para aprender transformações em dados 2D.

#### 2.1.1 Vector Field

```

1  import torch
2  import torch.nn as nn
3
4  class VectorField(nn.Module):
5      """
6      Parametriza  $dx/dt = f(x, t)$  usando rede neural.
7
8      Args:
9          features: dimensão dos dados
10         hidden_dims: lista com dimensões das camadas ocultas
11         time_embed_dim: dimensão do embedding temporal
12     """
13     def __init__(self, features, hidden_dims=[64, 64], time_embed_dim=16):
14         super().__init__()
15         self.features = features
16         self.time_embed_dim = time_embed_dim
17
18         # TODO: Implementar
19         # Dicas:
20         # 1. Time embedding: usar sinusoidal encoding
21         # 2. Network: MLP simples ou com skip connections
22         # 3. Inicialização: última camada com pesos pequenos ( $\sigma=0.01$ )
23
24     def time_embedding(self, t):
25         """
26         Sinusoidal time embedding.
27
28         Args:
29             t: (batch,) ou escalar
30         Returns:
31             embedded: (batch, time_embed_dim)
32         """

```

```

33         # TODO: implementar
34         # t_emb[2i] = sin(t / 10000^(2i/d))
35         # t_emb[2i+1] = cos(t / 10000^(2i/d))
36         pass
37
38     def forward(self, t, x):
39         """
40         Calcula f(x, t).
41
42         Args:
43             t: tempo (escalar ou (batch,))
44             x: estado (batch, features)
45         Returns:
46             dx_dt: (batch, features)
47         """
48         # TODO: implementar
49         # 1. Expandir t para batch se necessário
50         # 2. Time embedding
51         # 3. Concatenar [x, t_emb]
52         # 4. Passar pela rede
53         pass

```

**Exercício obrigatório:** Implemente a arquitetura **SimpleMLP** básica.

**Exercícios opcionais (bônus):** Implemente variantes avançadas:

1. **ResNetVF**: Com skip connections (inspirado em ResNet, He et al. 2015)
2. **TimeConditionedVF**: Usando FiLM layers para condicionamento temporal (Perez et al., *FiLM: Visual Reasoning with a General Conditioning Layer* (2018))

**Nota:** ResNet e FiLM não aparecem nos papers originais de CNF/FFJORD, mas são refinamentos arquiteturais modernos que podem melhorar performance. São opcionais para exploração adicional.

## 2.1.2 Neural ODE

```

1  from torchdiffeq import odeint
2
3  class NeuralODE(nn.Module):
4      """
5      Neural ODE: integra dx/dt = f(x,t) de t=0 até t=1.
6      """
7      def __init__(self, vector_field, solver='dopri5', rtol=1e-3, atol=1e-4):
8          super().__init__()
9          self.vf = vector_field
10         self.solver = solver
11         self.rtol = rtol
12         self.atol = atol
13
14     def forward(self, x0, t_span=None):
15         """
16         Integra ODE de t=0 até t=1.
17
18         Args:
19             x0: estado inicial (batch, features)
20             t_span: tempos para avaliar (default: [0, 1])

```

```

21     Returns:
22         x_t: trajetória (len(t_span), batch, features)
23     """
24     if t_span is None:
25         t_span = torch.tensor([0., 1.]).to(x0)
26
27     # TODO: usar odeint do torchdiffeq
28     # Dica: odeint(self.vf, x0, t_span, method=self.solver, ...)
29     pass

```

### Tarefas Milestone 1:

1. **Implementar** VectorField e NeuralODE
2. **Treinar** em dataset 2D sintético:

```

1     from sklearn.datasets import make_moons
2     X, _ = make_moons(n_samples=5000, noise=0.05)

```

### 3. Visualizar:

- Trajetórias:  $x(t)$  para  $t \in [0, 1]$
- Vector field: quiver plot de  $f(x, t)$  em grid 2D
- Transformação:  $z \sim N(0, I) \rightarrow x = \varphi(z, 1)$

### 4. Analisar:

- Quantas NFEs (number of function evaluations)?
- Como NFEs varia com `rtol` e `atol`?
- Comparar solvers: `euler`, `rk4`, `dopri5`

**Entregável:** Notebook `01_neural_ode_2d.ipynb` com visualizações e análise.

## Milestone 2: CNF com Trace Exato (Semana 1-2)

**Objetivo:** Implementar CNF completo, incluindo cálculo de log-likelihood via change of variables.

### 2.2.1 Change of Variables Formula

**Teoria** (incluir no relatório):

Para uma transformação invertível  $z = f(x)$ :

$$\log p(x) = \log p(z) + \log \left| \det \frac{\partial f}{\partial x} \right|$$

Para CNF com  $\varphi_t(x)$  integrando  $dx/dt = f(x, t)$ :

$$\log p(x) = \log p(z) + \int_0^1 \text{tr} \left( \frac{\partial f}{\partial x} \right) dt$$

onde  $z = \varphi_0(x)$  (integração reversa).

### 2.2.2 Divergence Computation (Trace Exato)

```

1  def divergence_exact(f, x):
2      """
3      Calcula trace( $\partial f / \partial x$ ) exatamente usando autograd.
4      ATENÇÃO: Custo  $O(d^2)$  – só viável para dimensão baixa!
5
6      Args:
7          f: função  $R^d \rightarrow R^d$ 
8          x: input (batch, d)
9      Returns:
10         trace: (batch,)
11     """
12     batch_size, dim = x.shape
13
14     # TODO: Implementar
15     # Estratégia:
16     # 1. Para cada dimensão i:
17     #     - Compute  $\partial f_i / \partial x_i$  usando torch.autograd.grad
18     # 2. Somar todas as derivadas diagonais
19
20     # Pseudo-código:
21     # trace = 0
22     # for i in range(dim):
23     #     # Compute  $\partial f[i] / \partial x[i]$ 
24     #     df_i = autograd.grad(f[:, i].sum(), x, create_graph=True)[0]
25     #     trace += df_i[:, i]
26     # return trace
27
28     pass

```

**Importante:** Explique no relatório por que isso é  $O(d^2)$ :

- Para cada dimensão  $i$ , precisamos de 1 backward pass
- Total:  $d$  backward passes
- Cada backward é  $O(d)$  → Total  $O(d^2)$

### 2.2.3 CNF Implementation

```

1  from torchdiffeq import odeint_adjoint
2
3  class CNF(nn.Module):
4      """
5      Continuous Normalizing Flow com trace exato.
6      """
7      def __init__(self, vector_field, base_dist=None):
8          super().__init__()
9          self.vf = vector_field
10
11         if base_dist is None:
12             # Prior:  $N(0, I)$ 
13             features = vector_field.features
14             self.base_dist = torch.distributions.MultivariateNormal(
15                 torch.zeros(features),
16                 torch.eye(features)
17             )
18         else:
19             self.base_dist = base_dist
20

```

```

21     def _augmented_dynamics(self, t, state):
22         """
23         Augmented ODE: integra [x, log_det] simultaneamente.
24
25         dx/dt = f(x, t)
26         d(log_det)/dt = trace(∂f/∂x)
27
28         Args:
29             t: tempo escalar
30             state: (batch, features + 1) # [x, log_det]
31         Returns:
32             d_state: (batch, features + 1)
33         """
34         batch_size = state.shape[0]
35         x = state[:, :-1] # (batch, features)
36
37         # Habilitar gradientes para x
38         x = x.requires_grad_(True)
39
40         # Compute vector field
41         dx_dt = self.vf(t, x) # (batch, features)
42
43         # Compute trace do Jacobiano
44         trace = divergence_exact(lambda x: self.vf(t, x), x) # (batch,)
45
46         # d(log_det)/dt = -trace (note o sinal!)
47         dlogdet_dt = -trace.unsqueeze(-1) # (batch, 1)
48
49         return torch.cat([dx_dt, dlogdet_dt], dim=-1)
50
51     def forward(self, x):
52         """
53         Forward:  $x \rightarrow z$  (usado para sampling)
54         Integra de t=0 para t=1.
55         """
56         batch_size = x.shape[0]
57
58         # Estado inicial: [x, 0]
59         log_det_init = torch.zeros(batch_size, 1).to(x)
60         state_0 = torch.cat([x, log_det_init], dim=-1)
61
62         # Integrar
63         t_span = torch.tensor([0., 1.]).to(x)
64         state_1 = odeint_adjoint(
65             self._augmented_dynamics,
66             state_0,
67             t_span,
68             method='dopri5',
69             rtol=1e-3,
70             atol=1e-4
71         )[-1] # Pegar apenas t=1
72
73         z = state_1[:, :-1]
74         delta_log_det = state_1[:, -1]
75
76         return z, delta_log_det
77
78     def log_prob(self, x):
79         """

```

```

80     Calcula log p(x) usando change of variables.
81     """
82     # Forward pass:  $x \rightarrow z$ 
83     z, delta_log_det = self.forward(x)
84
85     # log p(z) do prior
86     log_pz = self.base_dist.log_prob(z)
87
88     # log p(x) = log p(z) + log |det J|
89     log_px = log_pz + delta_log_det
90
91     return log_px
92
93     def sample(self, n_samples):
94         """
95         Sampling:  $z \sim p(z) \rightarrow x = \phi^{-1}(z)$ 
96         Integra de t=1 para t=0 (reverso).
97         """
98         # Sample do prior
99         z = self.base_dist.sample((n_samples,))
100
101         # Integrar de t=1 para t=0
102         # Apenas precisamos de x, não de log_det durante sampling
103         t_span = torch.tensor([1., 0.]).to(z)
104
105         # Usar ODE sem augmentation (mais rápido)
106         x = odeint(
107             self.vf,
108             z,
109             t_span,
110             method='dopri5',
111             rtol=1e-3,
112             atol=1e-4
113         )[-1]
114
115         return x

```

## Tarefas Milestone 2:

1. **Implementar** `divergence_exact` e `CNF`
2. **Treinar** em:
  - 2D sintético (moons, circles)
  - MNIST reduzido: usar apenas 100 pixels mais importantes (PCA)
3. **Comparar** com Real NVP (baseline de NF, pode ser acessado via a biblioteca Zuko):
  - Log-likelihood no test set
  - Tempo de treinamento (segundos/epoch)
  - Tempo de sampling (1000 samples)
  - Qualidade visual de samples
4. **Analisar**:
  - Por que trace exato não escala para MNIST completo (784 dim)?
  - Como NFEs compara entre CNF e Real NVP?

**Entregável:** Notebook `02_cnf_trace_comparison.ipynb`.

## Milestone 3: FFJORD com Hutchinson Estimator (Semana 2)

**Objetivo:** Escalar CNF para alta dimensão usando trace estimation.

### 2.3.1 Hutchinson's Trace Estimator

**Teoria** (incluir no relatório):

Para matriz  $A \in \mathbb{R}^{d \times d}$ :

$$\text{tr}(A) = \mathbb{E}_{\epsilon}[\epsilon^T A \epsilon]$$

onde  $\epsilon \sim p$  com  $\mathbb{E}[\epsilon] = 0$  e  $\mathbb{E}[\epsilon \epsilon^T] = I$ .

Distribuições comuns:

- **Gaussian:**  $\epsilon \sim N(0, I)$
- **Rademacher:**  $\epsilon_i \sim \text{Uniform}(\{-1, +1\})$

**Implementação:**

```
1  def hutchinson_trace_estimator(f, x, n_samples=1, noise='gaussian'):
2      """
3      Estima trace(∂f/∂x) usando Hutchinson's trick.
4      Custo: O(d) por sample – escalável!
5
6      Args:
7          f: função R^d -> R^d
8          x: input (batch, d)
9          n_samples: número de samples para estimativa
10         noise: 'gaussian' ou 'rademacher'
11
12     Returns:
13         trace_estimate: (batch,)
14
15     """
16     batch_size, dim = x.shape
17
18     trace_estimates = []
19
20     for _ in range(n_samples):
21         # Sample ε
22         if noise == 'gaussian':
23             epsilon = torch.randn_like(x) # N(0, I)
24         elif noise == 'rademacher':
25             epsilon = torch.randint(0, 2, x.shape).float() * 2 - 1 # {-1, +1}
26         else:
27             raise ValueError(f"Unknown noise type: {noise}")
28
29         # Compute f(x)
30         with torch.enable_grad():
31             x_req = x.requires_grad_(True)
32             f_x = f(x_req)
33
34         # Compute ε^T (∂f/∂x) via vector-Jacobian product
35         # Equivalente a ε^T A onde A = ∂f/∂x
36         # torch.autograd.grad computa exatamente isso!
37         vjp = torch.autograd.grad(
```



```

36         f_x,
37         x_req,
38         grad_outputs=epsilon,
39         create_graph=True,
40         retain_graph=True
41     )[0]
42
43     # trace  $\approx \epsilon^T v$  onde  $v = (\partial f / \partial x)^T \epsilon$ 
44     trace_est = (epsilon * vjp).sum(dim=-1) # (batch,)
45
46     trace_estimates.append(trace_est)
47
48     # Média sobre samples
49     return torch.stack(trace_estimates).mean(dim=0)

```

**Análise de Variância** (incluir no relatório):

Variância do estimador:  $\text{Var}[\text{trace\_est}] = \text{Var}[\epsilon^T A \epsilon]$

Com  $n$  samples:  $\text{Var\_final} \approx \text{Var} / n$

**Tarefas:** Experimento para medir variância empírica vs teoria.

## 2.3.2 FFJORD Implementation

```

1  class FFJORD(nn.Module):
2      """
3      Free-Form Jacobian of Reversible Dynamics.
4      CNF escalável usando Hutchinson estimator.
5      """
6      def __init__(self, vector_field, base_dist=None,
7                  n_trace_samples=1, noise='rademacher'):
9          super().__init__()
10         self.vf = vector_field
11         self.n_trace_samples = n_trace_samples
12         self.noise = noise
13
14         if base_dist is None:
15             features = vector_field.features
16             self.base_dist = torch.distributions.MultivariateNormal(
17                 torch.zeros(features),
18                 torch.eye(features)
19             )
20         else:
21             self.base_dist = base_dist
22
23     def _augmented_dynamics(self, t, state):
24         """
25         Augmented dynamics com trace estimation.
26         """
27         batch_size = state.shape[0]
28         x = state[:, :-1]
29
30         x = x.requires_grad_(True)
31
32         # Vector field
33         dx_dt = self.vf(t, x)

```

```

34         # Trace estimation
35         trace = hutchinson_trace_estimator(
36             lambda x: self.vf(t, x),
37             x,
38             n_samples=self.n_trace_samples,
39             noise=self.noise
40         )
41
42         dlogdet_dt = -trace.unsqueeze(-1)
43
44         return torch.cat([dx_dt, dlogdet_dt], dim=-1)
45
46     # forward, log_prob, sample: similares ao CNF
47     # (copiar implementação, mudando apenas _augmented_dynamics)

```

### Tarefas Milestone 3:

1. **Implementar** Hutchinson estimator e FFJORD
2. **Treinar** em MNIST completo (784 dim):

```

1  # Préprocessamento importante!
2  def preprocess_mnist(x):
3      # Dequantization:  $x \in \{0, \dots, 255\} \rightarrow x \in (0, 256)$ 
4      x = x + torch.rand_like(x)
5      x = x / 256.0 #  $\rightarrow (0, 1)$ 
6
7      # Logit transform para evitar boundary issues
8      alpha = 0.05
9      x = alpha + (1 - 2*alpha) * x
10     x = torch.logit(x)
11     return x

```

3. **Experimentos:**

#### E1: Variance do Trace Estimator

```

1  n_samples_list = [1, 2, 5, 10]
2  noise_types = ['gaussian', 'rademacher']
3
4  # Para cada combinação:
5  # - Treinar FFJORD
6  # - Medir variance empírica de trace durante treinamento
7  # - Plotar learning curves

```

#### E2: Gaussian vs Rademacher

- Comparar convergência
- Comparar tempo computacional
- Conclusão: Rademacher geralmente é melhor (mais rápido, variance similar)

4. **Comparar** com CNF (2D) e Real NVP:

Método	Dataset	Dim	Log-lik	NFE (train)	Time/epoch	Trace Cost
Real NVP	MNIST	784	-1200	N/A	30s	N/A
CNF (exact)	2D	2	?	?	?s	$O(d^2)$

Método	Dataset	Dim	Log-lik	NFE (train)	Time/epoch	Trace Cost
FFJORD	MNIST	784	?	?	?s	O(d)

Entregável: Notebook `03_ffjord_mnist.ipynb`.

## Parte 3: Análise Exploratória (Semana 2-3)

### 3.1 Experimento 1: Impacto de ODE Solvers

**Objetivo:** Comparar diferentes solvers em termos de accuracy vs speed.

```

1 solvers_config = [
2     {'method': 'euler', 'rtol': None, 'atol': None}, # Fixed step
3     {'method': 'rk4', 'rtol': None, 'atol': None},
4     {'method': 'dopri5', 'rtol': 1e-3, 'atol': 1e-4},
5     {'method': 'dopri5', 'rtol': 1e-4, 'atol': 1e-5},
6     {'method': 'dopri5', 'rtol': 1e-5, 'atol': 1e-6},
7 ]
8
9 # Para cada config:
10 # - Treinar modelo (ou usar checkpoint)
11 # - Medir NFE médio
12 # - Medir tempo de forward/backward
13 # - Medir log-likelihood final

```

**Análise esperada:**

- Trade-off: tolerância menor → mais NFEs → melhor accuracy
- `dopri5` é adaptive, `euler` / `rk4` são fixed-step
- Encontrar "sweet spot" para seu problema

### 3.2 Experimento 2: Regularizações

**Objetivo:** Estabilizar treinamento e melhorar generalização.

**Regularizações do paper FFJORD:**

```

1 def compute_regularizations(vf, x, t):
2     """
3     Computa termos de regularização.
4     """
5     x = x.requires_grad_(True)
6     v = vf(t, x) # (batch, d)
7
8     # R1: Kinetic Energy
9     # Penaliza velocidades altas: E[||v||²]
10    kinetic_energy = (v ** 2).sum(dim=-1).mean()
11
12    # R2: Jacobian Frobenius Norm
13    # Penaliza Jacobian complexo: E[||∂v/∂x||_F²]
14    jac_frob = 0.0
15    for i in range(v.shape[1]):

```

```

16         grad_v_i = torch.autograd.grad(
17             v[:, i].sum(), x,
18             create_graph=True, retain_graph=True
19         )[0]
20         jac_frob += (grad_v_i ** 2).sum()
21     jac_frob = jac_frob / v.shape[0] # Average over batch
22
23     return {
24         'kinetic_energy': kinetic_energy,
25         'jacobian_frobenius': jac_frob
26     }
27
28 # Loss total
29 loss = -log_prob.mean() +  $\lambda_{KE}$  * KE +  $\lambda_{JF}$  * JF

```

### Experimentos:

- Sem regularização (baseline)
- $\lambda_{KE} = 0.01, \lambda_{JF} = 0$
- $\lambda_{KE} = 0, \lambda_{JF} = 0.01$
- $\lambda_{KE} = 0.01, \lambda_{JF} = 0.01$

### Análise:

- Impacto na convergência
- Impacto na qualidade de samples
- Trade-off regularização vs log-likelihood

## 3.3 Experimento 3: Vector Field Architectures

**Objetivo:** Comparar diferentes architectures para o vector field.

### Arquiteturas Obrigatórias

Implementar 3 variantes obrigatórias:

```

1 # 1. SimpleMLP (OBRIGATÓRIA - baseline do paper FFJORD)
2 class SimpleMLP(VectorField):
3     """
4     MLP simples como no paper FFJORD original.
5     Architecture: Linear → ReLU → ... → Linear
6     """
7     def __init__(self, features, hidden_dims=[64, 64], time_embed_dim=16):
8         super().__init__()
9         self.features = features
10        self.time_embed_dim = time_embed_dim
11
12        # TODO: implementar
13        # Input: features + time_embed_dim
14        # Hidden: len(hidden_dims) camadas com ReLU
15        # Output: features
16        # IMPORTANTE: última camada com inicialização pequena (std=0.01)
17
18    def forward(self, t, x):
19        # TODO:

```

```

20         # 1. Time embedding
21         # 2. Concatenar [x, t_emb]
22         # 3. Passar pela rede
23         pass

```

**Variantes de profundidade/width** (escolher 2 para comparar):

- **Shallow:** `hidden_dims=[64, 64]` (2 camadas)
- **Medium:** `hidden_dims=[64, 64, 64, 64]` (4 camadas)
- **Deep:** `hidden_dims=[128, 128, 128, 128, 128, 128]` (6 camadas)
- **Wide:** `hidden_dims=[256, 256]` (2 camadas largas)

**Variantes de time embedding** (escolher 1 para comparar):

- **Sinusoidal:** Múltiplas frequências `sin(w_i * t), cos(w_i * t)`
- **Learnable:** Time embedding via pequena rede aprendível
- **Simple:** Apenas concatenar `t` como escalar (sem embedding)

## Arquiteturas Opcionais (Bônus: +5 pontos)

Se tiver tempo e quiser explorar refinamentos:

```

1  # 2. ResNetVF (OPCIONAL - não está no FFJORD original)
2  class ResNetVectorField(VectorField):
3      """
4      Vector field com skip connections (inspirado em ResNet).
5      Pode ajudar com gradient flow e estabilidade.
6      """
7      def __init__(self, features, n_blocks=3, hidden_dim=64, time_embed_dim=16):
8          super().__init__()
9          self.features = features
10
11         # TODO: implementar
12         # Architecture:
13         # x_0 = [x, t_emb]
14         # Para cada block i:
15         #   h_i = ResidualBlock(x_{i-1})
16         #   x_i = x_{i-1} + h_i # skip connection
17
18         def forward(self, t, x):
19             # TODO: implementar
20             pass
21
22 # 3. TimeConditionedVF (OPCIONAL - usa FiLM layers)
23 class TimeConditionedVF(VectorField):
24     """
25     FiLM: Feature-wise Linear Modulation (Perez et al., 2018).
26     Ao invés de concatenar tempo, usa-o para modular features.
27     """
28     def __init__(self, features, hidden_dims=[64, 64], time_embed_dim=32):
29         super().__init__()
30         self.features = features
31
32         # TODO: implementar
33         # FiLM:  $h = \gamma(t) \odot W(x) + \beta(t)$ 

```

```

34         # Onde  $\gamma$  e  $\beta$  são pequenas redes que dependem de t
35
36     def forward(self, t, x):
37         # TODO: implementar
38         # Para cada camada:
39         #     h = layer(x)
40         #      $\gamma, \beta$  = film_generator(t)
41         #     h =  $\gamma * h + \beta$ 
42     pass

```

### Nota sobre arquiteturas opcionais:

- **ResNet skip connections**: Não aparecem no FFJORD original (Grathwohl et al., 2018), mas são práticas comuns em deep learning que podem melhorar treinamento
- **FiLM layers**: Técnica de condicionamento de Perez et al., *FiLM: Visual Reasoning with a General Conditioning Layer* (2018), aplicada a CNFs como refinamento moderno
- Ambas são **explorações além do paper**, não requisitos do projeto

## Experimentos

### Obrigatórios:

1. **Profundidade**: Comparar Shallow vs Medium vs Deep
  - Log-likelihood no test set
  - Tempo de treinamento
  - NFEs
  - Facilidade de convergência
2. **Width**: Comparar hidden\_dim = 64 vs 128 vs 256
  - Trade-off: parâmetros vs performance
3. **Time embedding**: Comparar estratégias de encoding
  - Sinusoidal vs Learnable vs Simple
  - Impacto na qualidade do vector field aprendido

### Opcional (bônus):

4. **ResNet**: Comparar SimpleMLP vs ResNet/VF
  - ResNet melhora gradient flow?
  - Reduz NFE necessário?
5. **FiLM**: Comparar concatenação vs FiLM modulation
  - FiLM melhora condicionamento temporal?
  - Custo computacional adicional vale a pena?

### Análise esperada:

- Qual architecture é mais eficiente (parâmetros vs performance)?
- Profundidade vs width: qual impacta mais?
- Time embedding faz diferença significativa?
- (Se fez bônus) Refinamentos modernos (ResNet/FiLM) justificam complexidade?

---

## Parte 4: Conexão com Flow Matching

### 4.1 Questões Reflexivas (incluir no relatório)

#### Q1: Simulation-free Training

Atualmente, para treinar CNF/FFJORD você precisa:

1. Integrar ODE (odeint) → caro (muitas NFEs)
2. Backward através da integração (adjoint) → ainda mais caro

**Pergunta:** É possível treinar o vector field **sem integrar ODEs**?

**Dica:** E se você soubesse o "target" para  $f(x, t)$  diretamente?

---

#### Q2: Straight Trajectories

Observe suas visualizações 2D: as trajetórias  $x(t)$  são **curvas**.

CNF aprende essas trajetórias de forma "natural" (minimizando alguma energia).

**Pergunta:** E se **forçássemos** trajetórias **retas**?

Benefícios:

- Menos NFEs necessárias?
- Integração mais estável?

Como impor isso matematicamente?

---

#### Q3: Conditional Probability Paths

FFJORD aprende  $p(x)$  unconditional, integrando de  $z \sim N(0, I)$ .

**Pergunta:** E se quiséssemos modelar **explicitamente** o caminho  $z \rightarrow x$ ?

Isto é, definir uma família de distribuições  $p_t(x)$  para  $t \in [0, 1]$  onde:

- $p_0 = N(0, I)$
- $p_1 = p_{data}$

Como construir esse caminho? Vector field correspondente?

---

#### Q4: Optimal Transport

Em CNF/FFJORD, trajetórias de diferentes  $z$  podem "cruzar".

**Pergunta:** E se garantíssemos **transporte ótimo** (no sentido de Monge)?

- Mínimo custo de transporte:  $\int ||x(1) - x(0)||^2 dx$
- Não-cruzamento de trajetórias

Como isso afeta aprendizado e qualidade?

---

## Parte 5: Datasets e Protocolo

### 5.1 Datasets Obrigatórios

**2D Sintético** (debugging e visualização):

```
1 import numpy as np
2 from sklearn.datasets import make_moons, make_circles
3 import torch
4
5 def make_spirals(n_samples=1000, noise=0.05):
6     """Two intertwined spirals."""
7     n = n_samples // 2
8     theta = np.sqrt(np.random.rand(n)) * 2 * np.pi
9     r = theta / (2 * np.pi)
10    x = r * np.cos(theta) + noise * np.random.randn(n)
11    y = r * np.sin(theta) + noise * np.random.randn(n)
12
13    # Second spiral (rotated)
14    x2 = -r * np.cos(theta) + noise * np.random.randn(n)
15    y2 = -r * np.sin(theta) + noise * np.random.randn(n)
16
17    X = np.vstack([np.column_stack([x, y]),
18                  np.column_stack([x2, y2])])
19    return torch.FloatTensor(X)
20
21 # Datasets
22 datasets_2d = {
23     'moons': lambda: make_moons(n_samples=5000, noise=0.05)[0],
24     'circles': lambda: make_circles(n_samples=5000, noise=0.05, factor=0.5)[0],
25     'spirals': lambda: make_spirals(n_samples=5000, noise=0.05),
26 }
```

**MNIST** (avaliação quantitativa):

```
1 from torchvision import datasets, transforms
2
3 def get_mnist_loaders(batch_size=128):
4     """MNIST com préprocessamento adequado para CNF."""
5
6     def preprocess(x):
7         # x é tensor [0, 1]
8         # 1. Dequantization
9         x = x + torch.rand_like(x) / 256.0
10
11        # 2. Rescale para evitar 0 e 1
12        alpha = 0.05
```



```

13         x = alpha + (1 - 2*alpha) * x
14
15     # 3. Logit transform
16     x = torch.logit(x)
17
18     # 4. Flatten
19     return x.view(-1)
20
21     transform = transforms.Compose([
22         transforms.ToTensor(),
23         transforms.Lambda(preprocess)
24     ])
25
26     train_dataset = datasets.MNIST(
27         root='./data', train=True,
28         download=True, transform=transform
29     )
30     test_dataset = datasets.MNIST(
31         root='./data', train=False,
32         transform=transform
33     )
34
35     train_loader = DataLoader(
36         train_dataset, batch_size=batch_size,
37         shuffle=True, num_workers=4
38     )
39     test_loader = DataLoader(
40         test_dataset, batch_size=batch_size,
41         shuffle=False, num_workers=4
42     )
43
44     return train_loader, test_loader

```

## 5.2 Métricas de Avaliação

### 1. Log-Likelihood (bits/dim):

```

1  def bits_per_dim(log_prob, n_dims):
2      """
3      Converte log p(x) para bits por dimensão.
4      Métrica padrão para density estimation.
5      """
6      return -log_prob / (n_dims * np.log(2))
7
8  # Uso:
9  log_px = model.log_prob(x)
10 bpd = bits_per_dim(log_px, x.shape[1])

```

### 2. Number of Function Evaluations (NFE):

```

1  class NFECOUNTER:
2      def __init__(self):
3          self.nfe = 0
4
5      def reset(self):
6          self.nfe = 0
7

```

```

8     def __call__(self, module, input, output):
9         self.nfe += 1
10
11 # Uso:
12 nfe_counter = NFECounter()
13 model.vf.register_forward_hook(nfe_counter)
14
15 # Durante treinamento:
16 nfe_counter.reset()
17 loss = -model.log_prob(x).mean()
18 print(f"NFE: {nfe_counter.nfe}")

```

### 3. Tempo Computacional:

```

1 import time
2
3 # Training time
4 start = time.time()
5 for epoch in range(n_epochs):
6     train_one_epoch(...)
7 elapsed = time.time() - start
8
9 # Sampling time
10 start = time.time()
11 samples = model.sample(n_samples=1000)
12 sampling_time = time.time() - start

```

### 4. Qualidade Visual (apenas 2D e MNIST):

- 2D: Plot samples vs true data
- MNIST: Grid de samples (8×8)

## 5.3 Baseline para Comparação

Fornecer checkpoints:

```

1 baselines/
2 |— realnvp_mnist.pt          # Log-lik: ~-1200 nats
3 |— neural_ode_2d_moons.pt   # Para validação
4 |— training_config.json     # Hyperparameters usados

```

Alunos devem reportar comparação lado-a-lado com Real NVP.

## Parte 6: Entregáveis

### 6.1 Código (50% da nota)

Checklist:

- ☐ Código executa sem erros
- ☐ `README.md` com instruções de setup

- ☐ `requirements.txt` completo
- ☐ Implementação dos 3 Milestones
- ☐ Código bem documentado (docstrings)
- ☐ Reprodutível (seeds, configs)
- ☐ PEP 8 compliant

**Estrutura esperada:** Conforme Parte 1, Seção 1.2.

## 6.2 Relatório (40% da nota)

**Formato:** PDF, 4-6 páginas (excluindo apêndices).

**Estrutura:**

1. **Introdução** (0.5 pág)
  - Motivação: por que CNFs?
  - Objetivos do projeto
2. **Metodologia** (1 pág)
  - Implementação: torchdiffeq + próprio código
  - Architectures testadas
  - Protocolo experimental
3. **Resultados** (2-3 pág)
  - **Milestone 1:** Neural ODE 2D
    - Visualizações de trajetórias e vector field
  - **Milestone 2:** CNF com trace exato
    - Comparação com Real NVP (tabela)
    - Análise de custo computacional
  - **Milestone 3:** FFJORD
    - MNIST results
    - Experimentos E1-E3
  - **Tabela resumo** comparando todos os métodos
4. **Discussão** (1 pág)
  - Trade-offs: expressividade vs custo
  - Quando usar CNF vs discrete flow?
  - Limitações observadas
  - Dificuldades encontradas
5. **Conexão com Flow Matching** (0.5 pág)
  - Respostas às 4 questões reflexivas (Parte 4.1)
6. **Conclusão** (0.5 pág)
  - Principais aprendizados
  - Insights sobre CNFs/FFJORD

**Apêndices** (não contam no limite de páginas):

- Hyperparameters completos

- Código de funções-chave (se relevante)
- Figuras adicionais

### 6.3 Apresentação (10% da nota)

**Formato:** 5 min + 2 min Q&A

**Slides** (máximo 5):

1. **Título:** Nome, objetivo do projeto
2. **Implementação:** Milestones cumpridos
3. **Resultado visual principal:** Melhor figura (trajetórias ou samples)
4. **Análise quantitativa:** Tabela comparativa
5. **Conclusões:** Principais insights e aprendizados

## Parte 7: Rubrica Detalhada

### Código (50 pontos)

Critério	Pontos	Descrição
Execução	10	Roda sem erros, reproduzível
Milestone 1	8	Neural ODE implementado corretamente
Milestone 2	10	CNF com trace exato funcional
Milestone 3	12	FFJORD com Hutchinson
Documentação	6	Docstrings, comentários, README
Qualidade	4	PEP 8, modular, não-repetitivo

### Relatório (40 pontos)

Critério	Pontos	Descrição
Estrutura	6	Bem organizado, fácil de ler
Metodologia	6	Justificativas claras
Figuras	8	Alta qualidade, bem legendadas
Tabelas/Análise	8	Comparações completas e justas
Discussão	8	Insights profundos, crítico
Conexão FM	4	Questões reflexivas bem respondidas

### Apresentação (10 pontos)

Critério	Pontos	Descrição
Clareza	5	Compreensível, slides limpos
Timing	2.5	Respeita 5 minutos
Q&A	2.5	Respostas corretas

## Bônus (até +10)

- Implementação de regularizações além de KE e JF (+2)
- Experimento adicional não especificado (+3)
- Visualização interativa (HTML/JS) (+3)
- Reprodução de resultado do paper FFJORD (+5)

## Parte 8: Starter Code

Fornecer arquivo `starter_code.py` completo com TODO's estratégicos:

```

1  """
2  Starter Code para Mini-Projeto 1: CNF e FFJORD
3  Complete as seções marcadas com TODO.
4  """
5
6  import torch
7  import torch.nn as nn
8  from torchdiffeq import odeint, odeint_adjoint
9
10 # =====
11 # MILESTONE 1: Neural ODE
12 # =====
13
14 class VectorField(nn.Module):
15     """
16     Parametriza  $dx/dt = f(x, t)$ .
17     """
18     def __init__(self, features, hidden_dims=[64, 64], time_embed_dim=16):
19         super().__init__()
20         self.features = features
21         self.time_embed_dim = time_embed_dim
22
23         # TODO: Implementar network
24         # Sugestão:
25         # 1. self.time_embed: Linear(1, time_embed_dim)
26         # 2. self.network: MLP com input_dim = features + time_embed_dim
27         # 3. Inicialização: última layer com std=0.01
28
29     def time_embedding(self, t):
30         """Sinusoidal embedding."""
31         # TODO: implementar
32         # freq = 1.0 / (10000 * (torch.arange(0, self.time_embed_dim, 2) /
33         self.time_embed_dim))
34         # ...
35         pass

```

```

35
36     def forward(self, t, x):
37         """
38         Args:
39             t: escalar ou (batch,)
40             x: (batch, features)
41         Returns:
42             dx_dt: (batch, features)
43         """
44         # TODO: implementar
45         pass
46
47     class NeuralODE(nn.Module):
48         """Neural ODE usando torchdiffeq."""
49         def __init__(self, vector_field, solver='dopri5', rtol=1e-3, atol=1e-4):
50             super().__init__()
51             self.vf = vector_field
52             self.solver = solver
53             self.rtol = rtol
54             self.atol = atol
55
56         def forward(self, x0, t_span=None):
57             if t_span is None:
58                 t_span = torch.tensor([0., 1.]).to(x0)
59
60             # TODO: usar odeint
61             # trajectory = odeint(self.vf, x0, t_span,
62             #                      method=self.solver, rtol=self.rtol,
63             #                      atol=self.atol)
64             pass
65
66         # =====
67         # MILESTONE 2: CNF com Trace Exato
68         # =====
69
70     def divergence_exact(f, x):
71         """
72         Calcula trace( $\partial f / \partial x$ ) exatamente.  $O(d^2)$ !
73         """
74         batch_size, dim = x.shape
75         # TODO: implementar
76         # for i in range(dim):
77         #     df_dx_i = torch.autograd.grad(f[:, i].sum(), x, create_graph=True)[0]
78         #     trace += df_dx_i[:, i]
79         pass
80
81     class CNF(nn.Module):
82         """CNF com trace exato."""
83         def __init__(self, vector_field, base_dist=None):
84             super().__init__()
85             self.vf = vector_field
86             # TODO: definir base_dist se None
87
88         def _augmented_dynamics(self, t, state):
89             """Integra [x, log_det]."""
90             x = state[:, :-1]
91             x = x.requires_grad_(True)
92
93             # TODO:

```

```

93         # 1. dx_dt = self.vf(t, x)
94         # 2. trace = divergence_exact(lambda x: self.vf(t, x), x)
95         # 3. dlogdet_dt = -trace
96         # 4. return torch.cat([dx_dt, dlogdet_dt.unsqueeze(-1)], dim=-1)
97         pass
98
99     def log_prob(self, x):
100         """Compute log p(x)."""
101         # TODO:
102         # 1. Forward:  $x \rightarrow z$ , get delta_log_det
103         # 2. log_pz = self.base_dist.log_prob(z)
104         # 3. return log_pz + delta_log_det
105         pass
106
107     def sample(self, n_samples):
108         """Sample  $x \sim p(x)$ ."""
109         # TODO:
110         # 1.  $z = \text{self.base\_dist.sample}((n\_samples,))$ 
111         # 2. Integrate backwards:  $t=1 \rightarrow t=0$ 
112         # 3. return x
113         pass
114
115     # =====
116     # MILESTONE 3: FFJORD com Hutchinson
117     # =====
118
119     def hutchinson_trace_estimator(f, x, n_samples=1, noise='rademacher'):
120         """
121         Estima trace usando Hutchinson. O(d)!
122         """
123         # TODO: implementar
124         # for _ in range(n_samples):
125         #     if noise == 'rademacher':
126         #         epsilon = torch.randint(0, 2, x.shape) * 2 - 1
127         #     else: # gaussian
128         #         epsilon = torch.randn_like(x)
129         #
130         #     vjp = torch.autograd.grad(f(x), x, grad_outputs=epsilon, ...)[0]
131         #     trace += (epsilon * vjp).sum(-1)
132         pass
133
134     class FFJORD(CNF):
135         """FFJORD: CNF escalável."""
136         def __init__(self, vector_field, base_dist=None,
137                     n_trace_samples=1, noise='rademacher'):
138             super().__init__(vector_field, base_dist)
139             self.n_trace_samples = n_trace_samples
140             self.noise = noise
141
142         def _augmented_dynamics(self, t, state):
143             """Augmented dynamics com Hutchinson."""
144             x = state[:, :-1]
145             x = x.requires_grad_(True)
146
147             dx_dt = self.vf(t, x)
148
149             # TODO: usar hutchinson_trace_estimator
150             # trace = hutchinson_trace_estimator(
151             #     lambda x: self.vf(t, x), x,

```

```

152         #         n_samples=self.n_trace_samples, noise=self.noise
153         # )
154         pass
155
156     # =====
157     # UTILITIES
158     # =====
159
160     def bits_per_dim(log_prob, n_dims):
161         return -log_prob / (n_dims * np.log(2))
162
163     class NFECounter:
164         """Conta function evaluations."""
165         def __init__(self):
166             self.nfe = 0
167
168         def reset(self):
169             self.nfe = 0
170
171         def __call__(self, module, input, output):
172             self.nfe += 1
173
174     # =====
175     # TRAINING LOOP (EXEMPLO)
176     # =====
177
178     def train_cnf(model, train_loader, optimizer, n_epochs, device='cuda'):
179         """Training loop básico."""
180         model.train()
181
182         for epoch in range(n_epochs):
183             total_loss = 0.0
184             for batch in train_loader:
185                 x = batch[0] if isinstance(batch, (list, tuple)) else batch
186                 x = x.to(device)
187
188                 # TODO: completar
189                 # loss = -model.log_prob(x).mean()
190                 # optimizer.zero_grad()
191                 # loss.backward()
192                 # optimizer.step()
193
194             print(f"Epoch {epoch+1}/{n_epochs}, Loss: {total_loss /
len(train_loader):.4f}")

```

## Parte 9: FAQs

### Q: FFJORD não converge, o que fazer?

Checklist:

1. Préprocessamento MNIST correto? (dequantization + logit)
2. Learning rate: tente  $1e-4$
3. Inicialização: última layer do VF com  $\text{std}=0.01$
4. Regularizações:  $\lambda_{\text{KE}} = 0.01$  ajuda



5. Batch size: maior é melhor (128-256)
6. Warm-up: linear warm-up de LR nas primeiras 5 epochs

**Q: Trace exato está muito lento, posso pular?**

Não, mas faça apenas em 2D. É importante entender  $O(d^2)$  vs  $O(d)$ .

**Q: Quantas epochs?**

- 2D: 1000-5000 (é rápido)
- MNIST com FFJORD: 50-100 (suficiente para análise)

**Q: Posso usar GPU de outra pessoa?**

Sim, mas documente no relatório.

**Q: Trabalho em dupla: como dividir?**

Sugestão:

- Pessoa A: Milestones 1-2 + Exp 1
- Pessoa B: Milestone 3 + Exp 2-3
- Ambos: Relatório + apresentação

Deixe claro as contribuições individuais.

---

## Resumo Executivo para os Alunos

**O que você vai fazer:**

1. Implementar Neural ODE, CNF, FFJORD usando `torchdiffeq`
2. Entender trace exato ( $O(d^2)$ ) vs Hutchinson ( $O(d)$ )
3. Treinar em 2D e MNIST
4. Comparar com Real NVP
5. Preparar-se para Flow Matching via questões reflexivas

**O que você vai aprender:**

- Funcionamento interno de CNFs
  - Por que são expressivos mas caros
  - Técnicas de trace estimation
  - Trade-offs práticos
-