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

```
mini-project-1/
├── README.md
├── requirements.txt
├── src/
│   ├── models/
│   │   ├── vector_field.py      # VectorField architectures
│   │   ├── neural_ode.py        # Neural ODE básico
│   │   ├── cnf.py               # CNF com trace exato
│   │   └── ffjord.py            # FFJORD com Hutchinson
│   ├── utils/
│   │   ├── datasets.py          # Data loading
│   │   ├── training.py          # Training loops
│   │   ├── 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

```python
import torch
import torch.nn as nn

class VectorField(nn.Module):
    """
    Parametriza dx/dt = f(x, t) usando rede neural.

    Args:
        features: dimensão dos dados
        hidden_dims: lista com dimensões das camadas ocultas
        time_embed_dim: dimensão do embedding temporal
    """
    def __init__(self, features, hidden_dims=[64, 64], time_embed_dim=16):
        super().__init__()
        self.features = features
        self.time_embed_dim = time_embed_dim

        # TODO: Implementar
        # Dicas:
        # 1. Time embedding: usar sinusoidal encoding
        # 2. Network: MLP simples ou com skip connections
        # 3. Inicialização: última camada com pesos pequenos (σ=0.01)

    def time_embedding(self, t):
        """
        Sinusoidal time embedding.

        Args:
            t: (batch,) ou escalar
        Returns:
            embedded: (batch, time_embed_dim)
        """
```

```
        # TODO: implementar
        # t_emb[2i] = sin(t / 10000^(2i/d))
        # t_emb[2i+1] = cos(t / 10000^(2i/d))
        pass

    def forward(self, t, x):
        """
        Calcula f(x, t).

        Args:
            t: tempo (escalar ou (batch,))
            x: estado (batch, features)
        Returns:
            dx_dt: (batch, features)
        """
        # TODO: implementar
        # 1. Expandir t para batch se necessário
        # 2. Time embedding
        # 3. Concatenar [x, t_emb]
        # 4. Passar pela rede
        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

```
from torchdiffeq import odeint

class NeuralODE(nn.Module):
    """
    Neural ODE: integra dx/dt = f(x,t) de t=0 até t=1.
    """
    def __init__(self, vector_field, solver='dopri5', rtol=1e-3, atol=1e-4):
        super().__init__()
        self.vf = vector_field
        self.solver = solver
        self.rtol = rtol
        self.atol = atol

    def forward(self, x0, t_span=None):
        """
        Integra ODE de t=0 até t=1.

        Args:
            x0: estado inicial (batch, features)
            t_span: tempos para avaliar (default: [0, 1])
```

```
        Returns:
            x_t: trajetória (len(t_span), batch, features)
        """
        if t_span is None:
            t_span = torch.tensor([0., 1.]).to(x0)

        # TODO: usar odeint do torchdiffeq
        # Dica: odeint(self.vf, x0, t_span, method=self.solver, ...)
        pass
```

Tarefas Milestone 1:

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

```
from sklearn.datasets import make_moons
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: $z \sim N(0, I) \to x = \varphi(z, 1)$
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 $dx/dt = f(x, t)$:

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

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

2.2.2 Divergence Computation (Trace Exato)

```python
def divergence_exact(f, x):
    """
    Calcula trace(∂f/∂x) exatamente usando autograd.
    ATENÇÃO: Custo O(d²) - só viável para dimensão baixa!

    Args:
        f: função R^d -> R^d
        x: input (batch, d)
    Returns:
        trace: (batch,)
    """
    batch_size, dim = x.shape

    # TODO: Implementar
    # Estratégia:
    # 1. Para cada dimensão i:
    #    - Compute ∂f_i/∂x_i usando torch.autograd.grad
    # 2. Somar todas as derivadas diagonais

    # Pseudo-código:
    # trace = 0
    # for i in range(dim):
    #     # Compute ∂f[i]/∂x[i]
    #     df_i = autograd.grad(f[:, i].sum(), x, create_graph=True)[0]
    #     trace += df_i[:, i]
    # return trace

    pass
```

Importante: Explique no relatório por que isso é O(d²):

- Para cada dimensão i, precisamos de 1 backward pass
- Total: d backward passes
- Cada backward é O(d) → Total O(d²)

2.2.3 CNF Implementation

```python
from torchdiffeq import odeint_adjoint

class CNF(nn.Module):
    """
    Continuous Normalizing Flow com trace exato.
    """
    def __init__(self, vector_field, base_dist=None):
        super().__init__()
        self.vf = vector_field

        if base_dist is None:
            # Prior: N(0, I)
            features = vector_field.features
            self.base_dist = torch.distributions.MultivariateNormal(
                torch.zeros(features),
                torch.eye(features)
            )
        else:
            self.base_dist = base_dist

```

```python
    def _augmented_dynamics(self, t, state):
        """
        Augmented ODE: integra [x, log_det] simultaneamente.

        dx/dt = f(x, t)
        d(log_det)/dt = trace(∂f/∂x)

        Args:
            t: tempo escalar
            state: (batch, features + 1)  # [x, log_det]
        Returns:
            d_state: (batch, features + 1)
        """
        batch_size = state.shape[0]
        x = state[:, :-1]  # (batch, features)

        # Habilitar gradientes para x
        x = x.requires_grad_(True)

        # Compute vector field
        dx_dt = self.vf(t, x)  # (batch, features)

        # Compute trace do Jacobiano
        trace = divergence_exact(lambda x: self.vf(t, x), x)  # (batch,)

        # d(log_det)/dt = -trace (note o sinal!)
        dlogdet_dt = -trace.unsqueeze(-1)  # (batch, 1)

        return torch.cat([dx_dt, dlogdet_dt], dim=-1)

    def forward(self, x):
        """
        Forward: x → z (usado para sampling)
        Integra de t=0 para t=1.
        """
        batch_size = x.shape[0]

        # Estado inicial: [x, 0]
        log_det_init = torch.zeros(batch_size, 1).to(x)
        state_0 = torch.cat([x, log_det_init], dim=-1)

        # Integrar
        t_span = torch.tensor([0., 1.]).to(x)
        state_1 = odeint_adjoint(
            self._augmented_dynamics,
            state_0,
            t_span,
            method='dopri5',
            rtol=1e-3,
            atol=1e-4
        )[-1]  # Pegar apenas t=1

        z = state_1[:, :-1]
        delta_log_det = state_1[:, -1]

        return z, delta_log_det

    def log_prob(self, x):
        """
```

```
        Calcula log p(x) usando change of variables.
        """
        # Forward pass: x → z
        z, delta_log_det = self.forward(x)

        # log p(z) do prior
        log_pz = self.base_dist.log_prob(z)

        # log p(x) = log p(z) + log |det J|
        log_px = log_pz + delta_log_det

        return log_px

    def sample(self, n_samples):
        """
        Sampling: z ~ p(z) → x = ϕ^{-1}(z)
        Integra de t=1 para t=0 (reverso).
        """
        # Sample do prior
        z = self.base_dist.sample((n_samples,))

        # Integrar de t=1 para t=0
        # Apenas precisamos de x, não de log_det durante sampling
        t_span = torch.tensor([1., 0.]).to(z)

        # Usar ODE sem augmentation (mais rápido)
        x = odeint(
            self.vf,
            z,
            t_span,
            method='dopri5',
            rtol=1e-3,
            atol=1e-4
        )[-1]

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

Distribuições comuns:

- **Gaussian**: $\varepsilon \sim N(0, I)$
- **Rademacher**: $\varepsilon_i \sim Uniform(\{-1, +1\})$

Implementação:

```python
def hutchinson_trace_estimator(f, x, n_samples=1, noise='gaussian'):
    """
    Estima trace(∂f/∂x) usando Hutchinson's trick.
    Custo: O(d) por sample - escalável!

    Args:
        f: função R^d -> R^d
        x: input (batch, d)
        n_samples: número de samples para estimativa
        noise: 'gaussian' ou 'rademacher'
    Returns:
        trace_estimate: (batch,)
    """
    batch_size, dim = x.shape

    trace_estimates = []

    for _ in range(n_samples):
        # Sample ε
        if noise == 'gaussian':
            epsilon = torch.randn_like(x)  # N(0, I)
        elif noise == 'rademacher':
            epsilon = torch.randint(0, 2, x.shape).float() * 2 - 1  # {-1, +1}
        else:
            raise ValueError(f"Unknown noise type: {noise}")

        # Compute f(x)
        with torch.enable_grad():
            x_req = x.requires_grad_(True)
            f_x = f(x_req)

            # Compute ε^T (∂f/∂x) via vector-Jacobian product
            # Equivalente a ε^T A onde A = ∂f/∂x
            # torch.autograd.grad computa exatamente isso!
            vjp = torch.autograd.grad(
```

```
                f_x,
                x_req,
                grad_outputs=epsilon,
                create_graph=True,
                retain_graph=True
            )[0]

            # trace ≈ ε^T v onde v = (∂f/∂x)^T ε
            trace_est = (epsilon * vjp).sum(dim=-1)  # (batch,)

        trace_estimates.append(trace_est)

    # Média sobre samples
    return torch.stack(trace_estimates).mean(dim=0)
```

Análise de Variância (incluir no relatório):

Variância do estimador: Var[trace_est] = Var[ε^T A ε]

Com n samples: Var_final ≈ Var / n

Tarefas: Experimento para medir variância empírica vs teoria.

2.3.2 FFJORD Implementation

```
class FFJORD(nn.Module):
    """
    Free-Form Jacobian of Reversible Dynamics.
    CNF escalável usando Hutchinson estimator.
    """
    def __init__(self, vector_field, base_dist=None,
                 n_trace_samples=1, noise='rademacher'):
        super().__init__()
        self.vf = vector_field
        self.n_trace_samples = n_trace_samples
        self.noise = noise

        if base_dist is None:
            features = vector_field.features
            self.base_dist = torch.distributions.MultivariateNormal(
                torch.zeros(features),
                torch.eye(features)
            )
        else:
            self.base_dist = base_dist

    def _augmented_dynamics(self, t, state):
        """
        Augmented dynamics com trace estimation.
        """
        batch_size = state.shape[0]
        x = state[:, :-1]

        x = x.requires_grad_(True)

        # Vector field
        dx_dt = self.vf(t, x)

```

```
        # Trace estimation
        trace = hutchinson_trace_estimator(
            lambda x: self.vf(t, x),
            x,
            n_samples=self.n_trace_samples,
            noise=self.noise
        )

        dlogdet_dt = -trace.unsqueeze(-1)

        return torch.cat([dx_dt, dlogdet_dt], dim=-1)

    # forward, log_prob, sample: similares ao CNF
    # (copiar implementação, mudando apenas _augmented_dynamics)
```

Tarefas Milestone 3:

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

```
# Preprocessamento importante!
def preprocess_mnist(x):
    # Dequantization: x ∈ {0,...,255} → x ∈ (0, 256)
    x = x + torch.rand_like(x)
    x = x / 256.0  # → (0, 1)

    # Logit transform para evitar boundary issues
    alpha = 0.05
    x = alpha + (1 - 2*alpha) * x
    x = torch.logit(x)
    return x
```

3. **Experimentos**:

   E1: Variance do Trace Estimator

```
n_samples_list = [1, 2, 5, 10]
noise_types = ['gaussian', 'rademacher']

# Para cada combinação:
# - Treinar FFJORD
# - Medir variance empírica de trace durante treinamento
# - 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.

```
solvers_config = [
    {'method': 'euler', 'rtol': None, 'atol': None},  # Fixed step
    {'method': 'rk4', 'rtol': None, 'atol': None},
    {'method': 'dopri5', 'rtol': 1e-3, 'atol': 1e-4},
    {'method': 'dopri5', 'rtol': 1e-4, 'atol': 1e-5},
    {'method': 'dopri5', 'rtol': 1e-5, 'atol': 1e-6},
]

# Para cada config:
# - Treinar modelo (ou usar checkpont)
# - Medir NFE médio
# - Medir tempo de forward/backward
# - 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:

```
def compute_regularizations(vf, x, t):
    """
    Computa termos de regularização.
    """
    x = x.requires_grad_(True)
    v = vf(t, x)  # (batch, d)

    # R1: Kinetic Energy
    # Penaliza velocidades altas: E[||v||²]
    kinetic_energy = (v ** 2).sum(dim=-1).mean()

    # R2: Jacobian Frobenius Norm
    # Penaliza Jacobian complexo: E[||∂v/∂x||_F²]
    jac_frob = 0.0
    for i in range(v.shape[1]):
```

```
            grad_v_i = torch.autograd.grad(
                v[:, i].sum(), x,
                create_graph=True, retain_graph=True
            )[0]
            jac_frob += (grad_v_i ** 2).sum()
        jac_frob = jac_frob / v.shape[0]  # Average over batch

        return {
            'kinetic_energy': kinetic_energy,
            'jacobian_frobenius': jac_frob
        }

    # Loss total
    loss = -log_prob.mean() + λ_KE * KE + λ_JF * JF
```

Experimentos:

- Sem regularização (baseline)
- λ_KE = 0.01, λ_JF = 0
- λ_KE = 0, λ_JF = 0.01
- λ_KE = 0.01, λ_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. SimpleMLP (OBRIGATÓRIA - baseline do paper FFJORD)
class SimpleMLP(VectorField):
    """
    MLP simples como no paper FFJORD original.
    Architecture: Linear → ReLU → ... → Linear
    """
    def __init__(self, features, hidden_dims=[64, 64], time_embed_dim=16):
        super().__init__()
        self.features = features
        self.time_embed_dim = time_embed_dim

        # TODO: implementar
        # Input: features + time_embed_dim
        # Hidden: len(hidden_dims) camadas com ReLU
        # Output: features
        # IMPORTANTE: última camada com inicialização pequena (std=0.01)

    def forward(self, t, x):
        # TODO:
```

```
        # 1. Time embedding
        # 2. Concatenar [x, t_emb]
        # 3. Passar pela rede
        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:

```
# 2. ResNetVF (OPCIONAL - não está no FFJORD original)
class ResNetVectorField(VectorField):
    """
    Vector field com skip connections (inspirado em ResNet).
    Pode ajudar com gradient flow e estabilidade.
    """
    def __init__(self, features, n_blocks=3, hidden_dim=64, time_embed_dim=16):
        super().__init__()
        self.features = features

        # TODO: implementar
        # Architecture:
        # x_0 = [x, t_emb]
        # Para cada block i:
        #   h_i = ResidualBlock(x_{i-1})
        #   x_i = x_{i-1} + h_i  # skip connection

    def forward(self, t, x):
        # TODO: implementar
        pass

# 3. TimeConditionedVF (OPCIONAL - usa FiLM layers)
class TimeConditionedVF(VectorField):
    """
    FiLM: Feature-wise Linear Modulation (Perez et al., 2018).
    Ao invés de concatenar tempo, usa-o para modular features.
    """
    def __init__(self, features, hidden_dims=[64, 64], time_embed_dim=32):
        super().__init__()
        self.features = features

        # TODO: implementar
        # FiLM: h = γ(t) ⊙ W(x) + β(t)
```

```
        # Onde γ e β são pequenas redes que dependem de t

    def forward(self, t, x):
        # TODO: implementar
        # Para cada camada:
        #   h = layer(x)
        #   γ, β = film_generator(t)
        #   h = γ * h + β
        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 ResNetVF
   - 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 $z \to x$?

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):

```python
import numpy as np
from sklearn.datasets import make_moons, make_circles
import torch

def make_spirals(n_samples=1000, noise=0.05):
    """Two intertwined spirals."""
    n = n_samples // 2
    theta = np.sqrt(np.random.rand(n)) * 2 * np.pi
    r = theta / (2 * np.pi)
    x = r * np.cos(theta) + noise * np.random.randn(n)
    y = r * np.sin(theta) + noise * np.random.randn(n)

    # Second spiral (rotated)
    x2 = -r * np.cos(theta) + noise * np.random.randn(n)
    y2 = -r * np.sin(theta) + noise * np.random.randn(n)

    X = np.vstack([np.column_stack([x, y]),
                   np.column_stack([x2, y2])])
    return torch.FloatTensor(X)

# Datasets
datasets_2d = {
    'moons': lambda: make_moons(n_samples=5000, noise=0.05)[0],
    'circles': lambda: make_circles(n_samples=5000, noise=0.05, factor=0.5)[0],
    'spirals': lambda: make_spirals(n_samples=5000, noise=0.05),
}
```

MNIST (avaliação quantitativa):

```python
from torchvision import datasets, transforms

def get_mnist_loaders(batch_size=128):
    """MNIST com preprocessamento adequado para CNF."""

    def preprocess(x):
        # x é tensor [0, 1]
        # 1. Dequantization
        x = x + torch.rand_like(x) / 256.0

        # 2. Rescale para evitar 0 e 1
        alpha = 0.05
```

```
            x = alpha + (1 - 2*alpha) * x

            # 3. Logit transform
            x = torch.logit(x)

            # 4. Flatten
            return x.view(-1)

        transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Lambda(preprocess)
        ])

        train_dataset = datasets.MNIST(
            root='./data', train=True,
            download=True, transform=transform
        )
        test_dataset = datasets.MNIST(
            root='./data', train=False,
            transform=transform
        )

        train_loader = DataLoader(
            train_dataset, batch_size=batch_size,
            shuffle=True, num_workers=4
        )
        test_loader = DataLoader(
            test_dataset, batch_size=batch_size,
            shuffle=False, num_workers=4
        )

        return train_loader, test_loader
```

5.2 Métricas de Avaliação

1. Log-Likelihood (bits/dim):

```
def bits_per_dim(log_prob, n_dims):
    """
    Converte log p(x) para bits por dimensão.
    Métrica padrão para density estimation.
    """
    return -log_prob / (n_dims * np.log(2))

# Uso:
log_px = model.log_prob(x)
bpd = bits_per_dim(log_px, x.shape[1])
```

2. Number of Function Evaluations (NFE):

```
class NFECounter:
    def __init__(self):
        self.nfe = 0

    def reset(self):
        self.nfe = 0

```

```
        def __call__(self, module, input, output):
            self.nfe += 1

# Uso:
nfe_counter = NFECounter()
model.vf.register_forward_hook(nfe_counter)

# Durante treinamento:
nfe_counter.reset()
loss = -model.log_prob(x).mean()
print(f"NFE: {nfe_counter.nfe}")
```

3. Tempo Computacional:

```
import time

# Training time
start = time.time()
for epoch in range(n_epochs):
    train_one_epoch(...)
elapsed = time.time() - start

# Sampling time
start = time.time()
samples = model.sample(n_samples=1000)
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:

```
baselines/
├── realnvp_mnist.pt          # Log-lik: ~-1200 nats
├── neural_ode_2d_moons.pt    # Para validação
└── 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:

```python
"""
Starter Code para Mini-Projeto 1: CNF e FFJORD
Complete as seções marcadas com TODO.
"""

import torch
import torch.nn as nn
from torchdiffeq import odeint, odeint_adjoint

# ============================================
# MILESTONE 1: Neural ODE
# ============================================

class VectorField(nn.Module):
    """
    Parametriza dx/dt = f(x, t).
    """
    def __init__(self, features, hidden_dims=[64, 64], time_embed_dim=16):
        super().__init__()
        self.features = features
        self.time_embed_dim = time_embed_dim

        # TODO: Implementar network
        # Sugestão:
        # 1. self.time_embed: Linear(1, time_embed_dim)
        # 2. self.network: MLP com input_dim = features + time_embed_dim
        # 3. Inicialização: última layer com std=0.01

    def time_embedding(self, t):
        """Sinusoidal embedding."""
        # TODO: implementar
        # freq = 1.0 / (10000 ** (torch.arange(0, self.time_embed_dim, 2) / self.time_embed_dim))
        # ...
        pass
```

```python

    def forward(self, t, x):
        """
        Args:
            t: escalar ou (batch,)
            x: (batch, features)
        Returns:
            dx_dt: (batch, features)
        """
        # TODO: implementar
        pass

class NeuralODE(nn.Module):
    """Neural ODE usando torchdiffeq."""
    def __init__(self, vector_field, solver='dopri5', rtol=1e-3, atol=1e-4):
        super().__init__()
        self.vf = vector_field
        self.solver = solver
        self.rtol = rtol
        self.atol = atol

    def forward(self, x0, t_span=None):
        if t_span is None:
            t_span = torch.tensor([0., 1.]).to(x0)

        # TODO: usar odeint
        # trajectory = odeint(self.vf, x0, t_span,
        #                     method=self.solver, rtol=self.rtol,
atol=self.atol)
        pass

# ========================================
# MILESTONE 2: CNF com Trace Exato
# ========================================

def divergence_exact(f, x):
    """
    Calcula trace(∂f/∂x) exatamente. O(d²)!
    """
    batch_size, dim = x.shape
    # TODO: implementar
    # for i in range(dim):
    #     df_dx_i = torch.autograd.grad(f[:, i].sum(), x, create_graph=True)[0]
    #     trace += df_dx_i[:, i]
    pass

class CNF(nn.Module):
    """CNF com trace exato."""
    def __init__(self, vector_field, base_dist=None):
        super().__init__()
        self.vf = vector_field
        # TODO: definir base_dist se None

    def _augmented_dynamics(self, t, state):
        """Integra [x, log_det]."""
        x = state[:, :-1]
        x = x.requires_grad_(True)

        # TODO:
```

```
        # 1. dx_dt = self.vf(t, x)
        # 2. trace = divergence_exact(lambda x: self.vf(t, x), x)
        # 3. dlogdet_dt = -trace
        # 4. return torch.cat([dx_dt, dlogdet_dt.unsqueeze(-1)], dim=-1)
        pass

    def log_prob(self, x):
        """Compute log p(x)."""
        # TODO:
        # 1. Forward: x → z, get delta_log_det
        # 2. log_pz = self.base_dist.log_prob(z)
        # 3. return log_pz + delta_log_det
        pass

    def sample(self, n_samples):
        """Sample x ~ p(x)."""
        # TODO:
        # 1. z = self.base_dist.sample((n_samples,))
        # 2. Integrate backwards: t=1 → t=0
        # 3. return x
        pass

# ========================================
# MILESTONE 3: FFJORD com Hutchinson
# ========================================

def hutchinson_trace_estimator(f, x, n_samples=1, noise='rademacher'):
    """
    Estima trace usando Hutchinson. O(d)!
    """
    # TODO: implementar
    # for _ in range(n_samples):
    #     if noise == 'rademacher':
    #         epsilon = torch.randint(0, 2, x.shape) * 2 - 1
    #     else:  # gaussian
    #         epsilon = torch.randn_like(x)
    #
    #     vjp = torch.autograd.grad(f(x), x, grad_outputs=epsilon, ...)[0]
    #     trace += (epsilon * vjp).sum(-1)
    pass

class FFJORD(CNF):
    """FFJORD: CNF escalável."""
    def __init__(self, vector_field, base_dist=None,
                 n_trace_samples=1, noise='rademacher'):
        super().__init__(vector_field, base_dist)
        self.n_trace_samples = n_trace_samples
        self.noise = noise

    def _augmented_dynamics(self, t, state):
        """Augmented dynamics com Hutchinson."""
        x = state[:, :-1]
        x = x.requires_grad_(True)

        dx_dt = self.vf(t, x)

        # TODO: usar hutchinson_trace_estimator
        # trace = hutchinson_trace_estimator(
        #     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. Preprocessamento MNIST correto? (dequantization + logit)
2. Learning rate: tente 1e-4
3. Inicialização: última layer do VF com std=0.01
4. Regularizações: λ_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
