

# Redes Neurais Artificiais

Pedro H A Konzen

15 de fevereiro de 2024

# Licença

Este trabalho está licenciado sob a Licença Atribuição-CompartilhaIgual 4.0 Internacional Creative Commons. Para visualizar uma cópia desta licença, visite [http://creativecommons.org/licenses/by-sa/4.0/deed.pt\\_BR](http://creativecommons.org/licenses/by-sa/4.0/deed.pt_BR) ou mande uma carta para Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

# Prefácio

O site [notaspedrok.com.br](https://notaspedrok.com.br) é uma plataforma que construí para o compartilhamento de minhas notas de aula. Essas anotações feitas como preparação de aulas é uma prática comum de professoras/es. Muitas vezes feitas a rasciscos em rascunhos com validade tão curta quanto o momento em que são concebidas, outras vezes, com capricho de um diário guardado a sete chaves. Notas de aula também são feitas por estudantes - são anotações, fotos, prints, entre outras formas de registros de partes dessas mesmas aulas. Essa dispersão de material didático sempre me intrigou e foi o que me motivou a iniciar o site.

Com início em 2018, o site contava com apenas três notas incipientes. De lá para cá, conforme fui expandido e revisando os materiais, o site foi ganhando acessos de vários locais do mundo, em especial, de países de língua portuguesa. No momento, conta com 13 notas de aula, além de minicursos e uma coleção de vídeos e áudios.

As notas de **Redes Neurais Artificiais** fazem uma introdução às redes neurais artificiais com ênfase na resolução de problemas de matemática. Como ferramenta de apoio computacional, códigos exemplos são trabalhos em linguagem [Python](#), mais especificamente, com o pacote de aprendizagem de máquina [PyTorch](#).

Aproveito para agradecer a todas/os que de forma assídua ou esporádica contribuem com correções, sugestões e críticas! ;)

Pedro H A Konzen

<https://www.notaspedrok.com.br>

# Conteúdo

<b>Capa</b>	<b>i</b>
<b>Licença</b>	<b>ii</b>
<b>Prefácio</b>	<b>iii</b>
<b>Sumário</b>	<b>vi</b>
<b>1 Introdução</b>	<b>1</b>
<b>2 Perceptron</b>	<b>3</b>
2.1 Unidade de Processamento . . . . .	3
2.1.1 Um problema de classificação . . . . .	4
2.1.2 Problema de regressão . . . . .	11
2.1.3 Exercícios . . . . .	14
2.2 Algoritmo de Treinamento . . . . .	15
2.2.1 Método do Gradiente Descendente . . . . .	16
2.2.2 Método do Gradiente Estocástico . . . . .	20
2.2.3 Exercícios . . . . .	22
<b>3 Perceptron Multicamadas</b>	<b>24</b>
3.1 Modelo MLP . . . . .	24
3.1.1 Treinamento . . . . .	26
3.1.2 Aplicação: Problema de Classificação XOR . . . . .	27
3.1.3 Exercícios . . . . .	30
3.2 Aplicação: Problema de Classificação Binária . . . . .	31

3.2.1	Dados . . . . .	31
3.2.2	Modelo . . . . .	32
3.2.3	Treinamento e Teste . . . . .	33
3.2.4	Verificação . . . . .	35
3.2.5	Exercícios . . . . .	37
3.3	Aplicação: Aproximação de Funções . . . . .	37
3.3.1	Função unidimensional . . . . .	37
3.3.2	Função bidimensional . . . . .	40
3.3.3	Exercícios . . . . .	43
3.4	Diferenciação Automática . . . . .	44
3.4.1	Autograd MLP . . . . .	49
3.4.2	Exercícios . . . . .	52
<b>4</b>	<b>Redes Informadas pela Física</b>	<b>55</b>
4.1	Aplicação: Equação de Poisson . . . . .	55
4.1.1	Exercícios . . . . .	59
4.2	Aplicação: Equação do Calor . . . . .	60
4.3	PINN com Parâmetro a Determinar . . . . .	64
4.3.1	Exercícios . . . . .	70
	<b>Respostas dos Exercícios</b>	<b>72</b>
	<b>Bibliografia</b>	<b>73</b>

# Capítulo 1

## Introdução

Uma rede neural artificial é um modelo de aprendizagem profunda (**deep learning**), uma área da aprendizagem de máquina (**machine learning**). O termo tem origem no início dos desenvolvimentos de inteligência artificial, em que modelos matemáticos e computacionais foram inspirados no cérebro biológico (tanto de humanos como de outros animais). Muitas vezes desenvolvidos com o objetivo de compreender o funcionamento do cérebro, também tinham a intensão de emular a inteligência.

Nestas notas de aula, estudamos um dos modelos de redes neurais usualmente aplicados. A **unidade básica de processamento** data do modelo de neurônio de McCulloch-Pitts (McCulloch and Pitts, 1943), conhecido como **perceptron** (Rosenblatt, 1958, 1962), o primeiro com um algoritmo de treinamento para problemas de classificação linearmente separável. Um modelo similar é o ADALINE (do inglês, *adaptive linear element*, Widrow and Hoff, 1960), desenvolvido para a predição de números reais. Pela questão histórica, vamos usar o termo **perceptron** para designar a unidade básica (o neurônio), mesmo que o modelo de neurônio a ser estudado não seja restrito ao original.

Métodos de aprendizagem profunda são técnicas de treinamento (calibração) de composições em múltiplos níveis, aplicáveis a problemas de aprendizagem de máquina que, muitas vezes, não têm relação com o cérebro ou neurônios

biológicos. Um exemplo, é a rede neural que mais vamos explorar nas notas, o **perceptron multicamada** (MLP, em inglês *multilayer perceptron*), um modelo de progressão (em inglês, *feedforward*) de rede profunda em que a informação é processada pela composição de camadas de perceptrons. Embora a ideia de fazer com que a informação seja processada através da conexão de múltiplos neurônios tenha inspiração biológica, usualmente a escolha da disposição dos neurônios em uma MLP é feita por questões algorítmicas e computacionais. I.e., baseada na eficiente utilização da arquitetura dos computadores atuais.



## Capítulo 2

# Perceptron

### 2.1 Unidade de Processamento

A **unidade básica de processamento** (neurônio artificial) que exploramos nestas notas é baseada no **perceptron** (Fig. 2.1). Consiste na composição de uma **função de ativação**  $f : \mathbb{R} \rightarrow \mathbb{R}$  com a **pré-ativação**

$$z := \mathbf{w} \cdot \mathbf{x} + b \quad (2.1)$$

$$= w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b \quad (2.2)$$

onde,  $\mathbf{x} \in \mathbb{R}^n$  é o **vetor de entrada**,  $\mathbf{w} \in \mathbb{R}^n$  é o **vetor de pesos** e  $b \in \mathbb{R}$  é o **bias**. Escolhida uma função de ativação, a **saída do neurônio** é dada por

$$y = \mathcal{N}(\mathbf{x}; (\mathbf{w}, b)) \quad (2.3)$$

$$:= f(z) = f(\mathbf{w} \cdot \mathbf{x} + b) \quad (2.4)$$

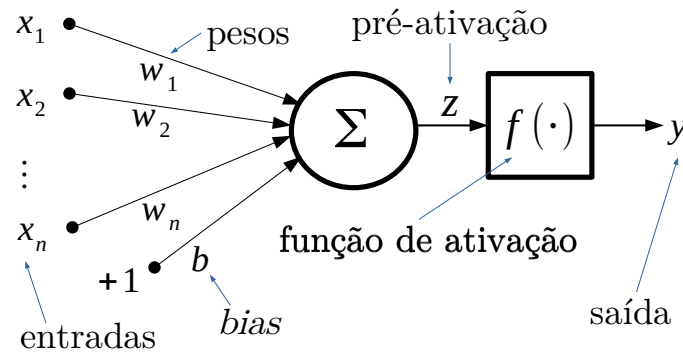


Figura 2.1: Esquema de um perceptron: unidade de processamento.

O **treinamento** (calibração) consiste em determinar os parâmetros  $(\mathbf{w}, b)$  de forma que o neurônio forneça as saídas  $y$  esperadas com base em um critério predeterminado.

Uma das vantagens deste modelo de neurônio é sua generalidade, i.e. pode ser aplicado a diferentes problemas. Na sequência, vamos aplicá-lo na resolução de um problema de classificação e noutro de regressão.

### 2.1.1 Um problema de classificação

Vamos desenvolver um perceptron que emule a operação  $\wedge$  (e-lógico). I.e, receba como entrada dois valores lógicos  $A_1$  e  $A_2$  (V, verdadeiro ou F, falso) e forneça como saída o valor lógico  $R = A_1 \wedge A_2$ . Segue a tabela verdade do  $\wedge$ :

$A_1$	$A_2$	$R$
V	V	V
V	F	F
F	V	F
F	F	F

## Modelo

Nosso **modelo de neurônio** será um perceptron com duas entradas  $\mathbf{x} \in \{-1, 1\}^2$  e a função sinal

$$f(z) = \text{sign}(z) = \begin{cases} 1 & , z > 0 \\ 0 & , z = 0 \\ -1 & , z < 0 \end{cases} \quad (2.5)$$

como função de ativação, i.e.

$$y = \mathcal{N}(\mathbf{x}; (\mathbf{w}, b)), \quad (2.6)$$

$$= \text{sign}(\mathbf{w} \cdot \mathbf{x} + b), \quad (2.7)$$

onde  $\mathbf{w} \in \mathbb{R}^2$  e  $b \in \mathbb{R}$  são parâmetros a determinar.

## Pré-processamento

Uma vez que nosso **modelo recebe valores  $\mathbf{x} \in \{-1, 1\}^2$**  e retorna  $y \in \{-1, 1\}$ , precisamos (pre)processar os dados do problema de forma a utilizá-los. Uma forma, é assumir que todo **valor negativo está associado ao valor lógico  $F$  (falso) e positivo ao valor lógico  $V$  (verdadeiro)**. Desta forma, os dados podem ser interpretados como na tabela abaixo.

$x_1$	$x_2$	$y$
1	1	1
1	-1	-1
-1	1	-1
-1	-1	-1

## Treinamento

Agora, nos falta **treinar nosso neurônio para fornecer o valor de  $y$  esperado para cada dada entrada  $\mathbf{x}$** . Isso **consiste em um método para escolhermos os parâmetros  $(\mathbf{w}, b)$**  que sejam adequados para esta tarefa. Vamos explorar mais sobre isso na sequência do texto e, aqui, apenas escolhemos

$$\mathbf{w} = (1, 1), \quad (2.8)$$

$$b = -1. \quad (2.9)$$

Com isso, nosso perceptron é

$$\mathcal{N}(\mathbf{x}) = \text{sign}(x_1 + x_2 - 1) \quad (2.10)$$

Verifique que ele satisfaz a tabela verdade acima!

### Implementação

Código 2.1: perceptron.py

```
1 import torch
2
3 # modelo
4 class Perceptron(torch.nn.Module):
5     def __init__(self):
6         super().__init__()
7         self.linear = torch.nn.Linear(2,1)
8
9     def forward(self, x):
10         z = self.linear(x)
11         y = torch.sign(z)
12         return y
13
14 model = Perceptron()
15 W = torch.Tensor([[1., 1.]])
16 b = torch.Tensor([-1.])
17 with torch.no_grad():
18     model.linear.weight = torch.nn.Parameter(W)
19     model.linear.bias = torch.nn.Parameter(b)
20
21 # dados de entrada
22 X = torch.tensor([[1., 1.],
23                  [1., -1.],
24                  [-1., 1.],
25                  [-1., -1.]])
26
27 print(f"\nDados de entrada\n{X}")
28
29
30 # forward (aplicação do modelo)
```

```

31 y = model(X)
32
220 33 print(f"Valores estimados\n{y}")

```

## Interpretação geométrica

Empregamos o seguinte modelo de neurônio

$$\mathcal{N}(\mathbf{x}; (\mathbf{w}, b)) = \text{sign}(w_1x_1 + w_2x_2 + b) \quad (2.11)$$

Observamos que

$$w_1x_1 + w_2x_2 + b = 0 \quad (2.12)$$

corresponde à equação geral de uma reta no plano  $\tau : x_1 \times x_2$ . Esta reta divide o plano em dois semiplanos

$$\tau^+ = \{\mathbf{x} \in \mathbb{R}^2 : w_1x_1 + w_2x_2 + b > 0\} \quad (2.13)$$

$$\tau^- = \{\mathbf{x} \in \mathbb{R}^2 : w_1x_1 + w_2x_2 + b < 0\} \quad (2.14)$$

O primeiro está na direção do vetor normal à reta  $\mathbf{n} = (w_1, w_2)$  e o segundo no sentido oposto. Com isso, o problema de treinar nosso neurônio para o problema de classificação consiste em encontrar a reta

$$w_1x_1 + w_2x_2 + b = 0 \quad (2.15)$$

de forma que o ponto  $(1, 1)$  esteja no semiplano positivo  $\tau^+$  e os demais pontos no semiplano negativo  $\tau^-$ . Consultamos a Figura 2.2.

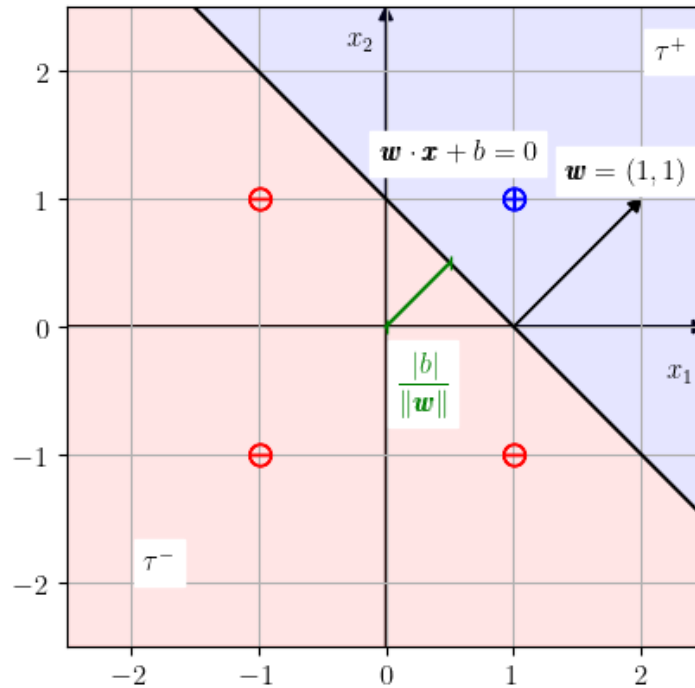


Figura 2.2: Interpretação geométrica do perceptron aplicado ao problema de classificação relacionado à operação lógica  $\wedge$  (e-lógico).

### Algoritmo de treinamento: perceptron

O algoritmo de treinamento perceptron permite calibrar os pesos de um neurônio para fazer a classificação de dados linearmente separáveis. Trata-se de um algoritmo para o **treinamento supervisionado** de um neurônio, i.e. a calibração dos pesos é feita com base em um dado **conjunto de amostras de treinamento**.

Seja dado um **conjunto de treinamento**  $\{\mathbf{x}^{(s)}, y^{(s)}\}_{s=1}^{n_s}$ , onde  $n_s$  é o número de amostras. O algoritmo consiste no seguinte:

1.  $\mathbf{w} \leftarrow \mathbf{0}$ ,  $b \leftarrow 0$ .
2. Para  $e \leftarrow 1, \dots, n_e$ :

(a) Para  $s \leftarrow 1, \dots, n_s$ :

i. Se  $y^{(s)} \mathcal{N}(\mathbf{x}^{(s)}) \leq 0$ :

A.  $\mathbf{w} \leftarrow \mathbf{w} + y^{(s)} \mathbf{x}^{(s)}$

B.  $b \leftarrow b + y^{(s)}$

onde,  $n_e$  é um dado número de épocas<sup>1</sup>.

Código 2.2: perceptron\_train.py

```
1 import torch
2
3 # modelo
4
5 class Perceptron(torch.nn.Module):
6     def __init__(self):
7         super().__init__()
8         self.linear = torch.nn.Linear(2,1)
9
10    def forward(self, x):
11        z = self.linear(x)
12        y = torch.sign(z)
13        return y
14
15 model = Perceptron()
16 with torch.no_grad():
17     W = model.linear.weight
18     b = model.linear.bias
19
20 # dados de treinamento
21 X_train = torch.tensor([[1., 1.],
22                          [1., -1.],
23                          [-1., 1.],
24                          [-1., -1.]])
```

<sup>1</sup>Número de vezes que as amostras serão percorridas para realizar a correção dos pesos.

```
25 y_train = torch.tensor([1., -1., -1., -1.]).reshape(-1,1)
26
27 ## número de amostras
28 ns = y_train.size(0)
29
30 print("\nDados de treinamento")
31 print("X_train =")
32 print(X_train)
33 print("y_train = ")
34 print(y_train)
35
36 # treinamento
37
38 ## num max épocas
39 nepochs = 100
40
41 for epoch in range(nepochs):
42
43     # update
44     not_updated = True
45     for s in range(ns):
46         y_est = model(X_train[s:s+1,:])
47         if (y_est*y_train[s] <= 0.):
48             with torch.no_grad():
49                 W += y_train[s]*X_train[s,:]
50                 b += y_train[s]
51                 not_updated = False
52
53     if (not_updated):
54         print('Training ended.')
55         break
56
57
58 # verificação
59 print(f'W =\n{W}')
60 print(f'b =\n{b}')
61 y = model(X_train)
62 print(f'y =\n{y}')
```



### 2.1.2 Problema de regressão

Vamos treinar um perceptron para resolver o problema de regressão linear para os seguintes dados

s	$x^{(s)}$	$y^{(s)}$
1	0.5	1.2
2	1.0	2.1
3	1.5	2.6
4	2.0	3.6

#### Modelo

Vamos determinar o perceptron<sup>2</sup>

$$\tilde{y} = \mathcal{N}(x; (w, b)) = wx + b \quad (2.16)$$

que melhor se ajusta a este conjunto de dados  $\{(x^{(s)}, y^{(s)})\}_{s=1}^{n_s}$ ,  $n_s = 4$ .

#### Treinamento

A ideia é que o perceptron seja tal que minimize o erro quadrático médio (MSE, do inglês, *Mean Squared Error*), i.e.

$$\min_{w, b} \frac{1}{n_s} \sum_{s=1}^{n_s} (\tilde{y}^{(s)} - y^{(s)})^2 \quad (2.17)$$

Vamos denotar a **função erro** (em inglês, *loss function*) por

$$\varepsilon(w, b) := \frac{1}{n_s} \sum_{s=1}^{n_s} (\tilde{y}^{(s)} - y^{(s)})^2 \quad (2.18)$$

$$= \frac{1}{n_s} \sum_{s=1}^{n_s} (wx^{(s)} + b - y^{(s)})^2 \quad (2.19)$$

Observamos que o problema (2.17) é equivalente a um problema linear de mínimos quadrados. A solução é obtida resolvendo-se a equação normal<sup>3</sup>

$$M^T M \mathbf{c} = M^T \mathbf{y}, \quad (2.20)$$

<sup>2</sup>Escolhendo  $f(z) = z$  como função de ativação.

<sup>3</sup>Consulte o Exercício 2.1.4.

onde  $\mathbf{c} = (w, p)$  é o vetor dos parâmetros a determinar e  $M$  é a matriz  $n_s \times 2$  dada por

$$M = \begin{bmatrix} \mathbf{x} & 1 \end{bmatrix} \quad (2.21)$$

### Implementação

Código 2.3: perceptron\_mq.py

```

1 import torch
2
3 # modelo
4 class Perceptron(torch.nn.Module):
5     def __init__(self):
6         super().__init__()
7         self.linear = torch.nn.Linear(1,1)
8
9     def forward(self, x):
10         z = self.linear(x)
11         return z
12
13 model = Perceptron()
14 with torch.no_grad():
15     W = model.linear.weight
16     b = model.linear.bias
17
18 # dados de treinamento
19 X_train = torch.tensor([0.5,
20                          1.0,
21                          1.5,
22                          2.0]).reshape(-1,1)
23 y_train = torch.tensor([1.2,
24                          2.1,
25                          2.6,
26                          3.6]).reshape(-1,1)
27
28 ## número de amostras
29 ns = y_train.size(0)
30

```

```
31 print("\nDados de treinamento")
32 print("X_train =")
220 33 print(X_train)
34 print("y_train = ")
35 print(y_train)
36
200 37 # treinamento
38
39 ## matriz
40 M = torch.hstack((X_train,
180 41                  torch.ones((ns,1))))
42 ## solução M.Q.
43 c = torch.linalg.lstsq(M, y_train)[0]
44 with torch.no_grad():
160 45     W = c[0]
46     b = c[1]
47
48 # verificação
140 49 print(f'W =\n{W}')
50 print(f'b =\n{b}')
51 y = model(X_train)
120 52 print(f'y =\n{y}')
```

## Resultado

Nosso perceptron corresponde ao modelo

$$\mathcal{N}(x; (w, b)) = wx + b \quad (2.22)$$

com pesos treinados  $w = 1.54$  e  $b = 0.45$ . Ele corresponde à reta que melhor se ajusta ao conjunto de dados de  $\{x^{(s)}, y^{(s)}\}_{s=1}^4$  dado na tabela acima. Consultamos a Figura 2.3.

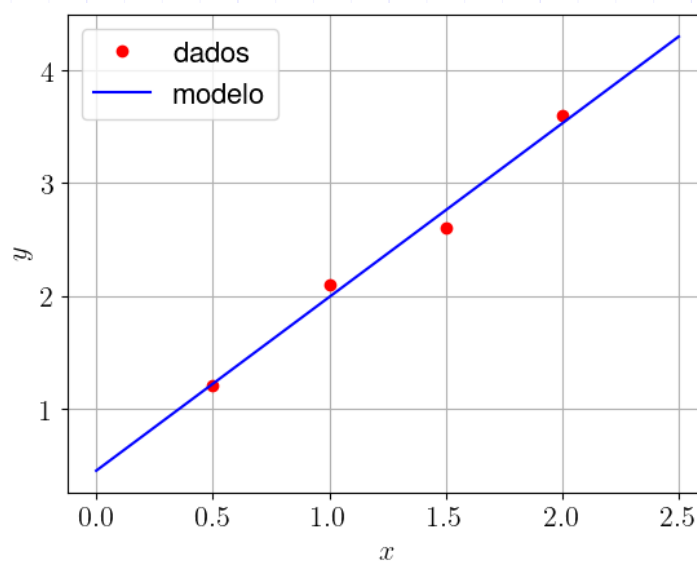


Figura 2.3: Interpretação geométrica do perceptron aplicado ao problema de regressão linear.

### 2.1.3 Exercícios

**E.2.1.1.** Crie um perceptron que emule a operação lógica do  $\vee$  (ou-lógico).

$A_1$	$A_2$	$A_1 \vee A_2$
V	V	V
V	F	V
F	V	V
F	F	F

**E.2.1.2.** Busque criar um perceptron que emule a operação lógica do **xor**.

$A_1$	$A_2$	$A_1 \text{ xor } A_2$
V	V	F
V	F	V
F	V	V
F	F	F

É possível? Justifique sua resposta.

**E.2.1.3.** Assumindo o modelo de neurônio (2.16), mostre que (2.18) é função convexa.

**E.2.1.4.** Mostre que a solução do problema (2.17) é dada por (2.20).

**E.2.1.5.** Crie um perceptron com função de ativação  $f(x) = \tanh(x)$  que melhor se ajuste ao seguinte conjunto de dados:

s	$x^{(s)}$	$y^{(s)}$
1	-1,0	-0,8
2	-0,7	-0,7
3	-0,3	-0,5
4	0,0	-0,4
5	0,2	-0,2
6	0,5	0,0
7	1,0	0,3

## 2.2 Algoritmo de Treinamento

Na seção anterior, desenvolvemos dois modelos de neurônios para problemas diferentes, um de classificação e outro de regressão. Em cada caso, utilizamos algoritmos de treinamento diferentes. Agora, vamos estudar algoritmos de treinamentos mais gerais<sup>4</sup>, que podem ser aplicados a ambos os problemas.

<sup>4</sup>Aqui, vamos explorar apenas algoritmos de treinamento supervisionado.

Ao longo da seção, vamos considerar o **modelo** de neurônio

$$\tilde{y} = \mathcal{N}(\mathbf{x}; (\mathbf{w}, b)) = f(\underbrace{\mathbf{w} \cdot \mathbf{x} + b}_z), \quad (2.23)$$

com dada função de ativação  $f: \mathbb{R} \rightarrow \mathbb{R}$ , sendo os vetores de entrada  $\mathbf{x}$  e dos pesos  $\mathbf{w}$  de tamanho  $n_{in}$ . A pré-ativação do neurônio é denotada por

$$z := \mathbf{w} \cdot \mathbf{x} + b \quad (2.24)$$

Fornecido um **conjunto de treinamento**  $\{(\mathbf{x}^{(s)}, y^{(s)})\}_1^{n_s}$ , com  $n_s$  amostras, o objetivo é calcular os parâmetros  $(\mathbf{w}, b)$  que minimizam a **função erro quadrático médio**

$$\varepsilon(\mathbf{w}, b) := \frac{1}{n_s} \sum_{s=1}^{n_s} (\tilde{y}^{(s)} - y^{(s)})^2 \quad (2.25)$$

$$= \frac{1}{n_s} \sum_{s=1}^{n_s} \varepsilon^{(s)} \quad (2.26)$$

onde  $\tilde{y}^{(s)} = \mathcal{N}(\mathbf{x}^{(s)}; (\mathbf{w}, b))$  é o **valor estimado** pelo modelo e  $y^{(s)}$  é o **valor esperado** para a  $s$ -ésima amostra. A função erro para a  $s$ -ésima amostra é

$$\varepsilon^{(s)} := (\tilde{y}^{(s)} - y^{(s)})^2. \quad (2.27)$$

Ou seja, o treinamento consiste em resolver o seguinte **problema de otimização**

$$\min_{(\mathbf{w}, b)} \varepsilon(\mathbf{w}, b) \quad (2.28)$$

Para resolver este problema de otimização, vamos empregar o Método do Gradiente Descendente.

### 2.2.1 Método do Gradiente Descendente

O **Método do Gradiente Descendente** (GD, em inglês, *Gradient Descent Method*) é um **método de declive**. Aplicado ao nosso modelo de Perceptron consiste no seguinte algoritmo:

1.  $(\mathbf{w}, b)$  aproximação inicial.

2. Para  $e \leftarrow 1, \dots, n_e$ :

$$(a) \quad (\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - l_r \frac{\partial \varepsilon}{\partial (\mathbf{w}, b)}$$

onde,  $n_e$  é o **número de épocas**,  $l_r$  é uma dada **taxa de aprendizagem** ( $l_r$ , do inglês, *learning rate*) e o **gradiente** é

$$\frac{\partial \varepsilon}{\partial (\mathbf{w}, b)} := \left( \frac{\partial \varepsilon}{\partial w_1}, \dots, \frac{\partial \varepsilon}{\partial w_{n_{in}}}, \frac{\partial \varepsilon}{\partial b} \right) \quad (2.29)$$

O cálculo do gradiente para os pesos  $\mathbf{w}$  pode ser feito como segue<sup>5</sup>

$$\frac{\partial \varepsilon}{\partial \mathbf{w}} = \frac{\partial}{\partial \mathbf{w}} \left[ \frac{1}{n_s} \sum_{s=1}^{n_s} \varepsilon^{(s)} \right] \quad (2.30)$$

$$= \frac{1}{n_s} \sum_{s=1}^{n_s} \frac{\partial \varepsilon^{(s)}}{\partial \tilde{y}^{(s)}} \frac{\partial \tilde{y}^{(s)}}{\partial \mathbf{w}} \quad (2.31)$$

$$\frac{\partial \varepsilon}{\partial \mathbf{w}} = \frac{1}{n_s} \sum_{s=1}^{n_s} \frac{\partial \varepsilon^{(s)}}{\partial \tilde{y}^{(s)}} \frac{\partial \tilde{y}^{(s)}}{\partial z^{(s)}} \frac{\partial z^{(s)}}{\partial \mathbf{w}} \quad (2.32)$$

Observando que

$$\frac{\partial \varepsilon^{(s)}}{\partial \tilde{y}^{(s)}} = 2 \left( \tilde{y}^{(s)} - y^{(s)} \right) \quad (2.33)$$

$$\frac{\partial \tilde{y}^{(s)}}{\partial z^{(s)}} = f' \left( z^{(s)} \right) \quad (2.34)$$

$$\frac{\partial z^{(s)}}{\partial \mathbf{w}} = \mathbf{x}^{(s)} \quad (2.35)$$

obtemos

$$\frac{\partial \varepsilon}{\partial \mathbf{w}} = \frac{1}{n_s} \sum_{s=1}^{n_s} 2 \left( \tilde{y}^{(s)} - y^{(s)} \right) f' \left( z^{(s)} \right) \mathbf{x}^{(s)} \quad (2.36)$$

<sup>5</sup>Aqui, há um abuso de linguagem ao não se observar as dimensões dos operandos matriciais.

$$\frac{\partial \varepsilon}{\partial b} = \frac{1}{n_s} \sum_{s=1}^{n_s} \frac{\partial \varepsilon^{(s)}}{\partial \tilde{y}^{(s)}} \frac{\partial \tilde{y}^{(s)}}{\partial z^{(s)}} \frac{\partial z^{(s)}}{\partial b} \quad (2.37)$$

$$\frac{\partial \varepsilon}{\partial b} = \frac{1}{n_s} \sum_{s=1}^{n_s} 2 \left( \tilde{y}^{(s)} - y^{(s)} \right) f' \left( z^{(s)} \right) \cdot 1 \quad (2.38)$$

### Aplicação: Problema de Classificação

Na Subseção 2.1.1, treinamos um perceptron para o problema de classificação do e-lógico. A função de ativação  $f(x) = \text{sign}(x)$  não é adequada para a aplicação do Método GD, pois  $f'(x) \equiv 0$  para  $x \neq 0$ . Aqui, vamos usar

$$f(x) = \tanh(x). \quad (2.39)$$

Código 2.4: perceptron\_gd.py

```
1 import torch
2
3 # modelo
4
5 class Perceptron(torch.nn.Module):
6     def __init__(self):
7         super().__init__()
8         self.linear = torch.nn.Linear(2,1)
9
10    def forward(self, x):
11        z = self.linear(x)
12        y = torch.tanh(z)
13        return y
14
15 model = Perceptron()
16
17 # treinamento
18
19 ## otimizador
20 optim = torch.optim.SGD(model.parameters(), lr=5e-1)
21
22 ## função erro
23 loss_fun = torch.nn.MSELoss()
```



```
24
25 ## dados de treinamento
220 26 X_train = torch.tensor([[1., 1.],
27                           [1., -1.],
28                           [-1., 1.],
29                           [-1., -1.]])
200 30 y_train = torch.tensor([1., -1., -1., -1.]).reshape(-1,1)
31
32 print("\nDados de treinamento")
33 print("X_train =")
180 34 print(X_train)
35 print("y_train = ")
36 print(y_train)
37
160 38 ## num max épocas
39 nepochs = 1000
40 tol = 1e-3
41
140 42 for epoch in range(nepochs):
43
44     # forward
45     y_est = model(X_train)
120 46
47     # erro
48     loss = loss_fun(y_est, y_train)
49
100 50     print(f'{epoch}: {loss.item():.4e}')
51
52     # critério de parada
53     if (loss.item() < tol):
80 54         break
55
56     # backward
57     optim.zero_grad()
60 58     loss.backward()
59     optim.step()
60
61
```

```

62 # verificação
63 y = model(X_train)
64 print(f'y_est = {y}')
```

### 2.2.2 Método do Gradiente Estocástico

O **Método do Gradiente Estocástico** (SGD, do inglês, *Stochastic Gradient Descent Method*) é uma variação do Método GD. A ideia é atualizar os parâmetros do modelo com base no gradiente do erro de cada amostra (ou um subconjunto de amostras<sup>6</sup>). A estocasticidade é obtida da randomização com que as amostras são escolhidas a cada época. O algoritmo consiste no seguinte:

1.  $\mathbf{w}$ ,  $b$  aproximações inicial.
2. Para  $e \leftarrow 1, \dots, n_e$ :
  - 1.1. Para  $s \leftarrow \text{random}(1, \dots, n_s)$ :

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - l_r \frac{\partial \varepsilon^{(s)}}{\partial (\mathbf{w}, b)} \quad (2.40)$$

#### Aplicação: Problema de Classificação

Código 2.5: perceptron\_sgd.py

```

1 import torch
2 import numpy as np
3
4 # modelo
5
6 class Perceptron(torch.nn.Module):
7     def __init__(self):
8         super().__init__()
9         self.linear = torch.nn.Linear(2,1)
10
```

<sup>6</sup>Nest caso, é conhecido como Batch SGD.

```
11     def forward(self, x):
12         z = self.linear(x)
13         y = torch.tanh(z)
14         return y
15
16 model = Perceptron()
17
18 # treinamento
19
20 ## otimizador
180 21 optim = torch.optim.SGD(model.parameters(), lr=5e-1)
22
23 ## função erro
24 loss_fun = torch.nn.MSELoss()
160 25
26 ## dados de treinamento
27 X_train = torch.tensor([[1., 1.],
28                         [1., -1.],
140 29                         [-1., 1.],
30                         [-1., -1.]])
31 y_train = torch.tensor([1., -1., -1., -1.]).reshape(-1,1)
32
120 33 ## num de amostras
34 ns = y_train.size(0)
35
36 print("\nDados de treinamento")
100 37 print("X_train =")
38 print(X_train)
39 print("y_train = ")
40 print(y_train)
80 41
42 ## num max épocas
43 nepochs = 5000
44 tol = 1e-3
60 45
46 for epoch in range(nepochs):
47
48     # forward
```

```
49     y_est = model(X_train)
50
51     # erro
52     loss = loss_fun(y_est, y_train)
53
54     print(f'{epoch}: {loss.item():.4e}')
55
56     # critério de parada
57     if (loss.item() < tol):
58         break
59
60     # backward
61     for s in torch.randperm(ns):
62         loss_s = (y_est[s,:] - y_train[s,:])**2
63         optim.zero_grad()
64         loss_s.backward()
65         optim.step()
66         y_est = model(X_train)
67
68
69 # verificação
70 y = model(X_train)
71 print(f'y_est = {y}')
```

### 2.2.3 Exercícios

**E.2.2.1.** Calcule a derivada da função de ativação

$$f(x) = \tanh(x). \quad (2.41)$$

**E.2.2.2.** Crie um perceptron para emular a operação lógica  $\wedge$  (e-lógico). No treinamento, use como otimizador:

- a) Método GD.
- b) Método SGD.

**E.2.2.3.** Crie um perceptron para emular a operação lógica  $\vee$  (ou-lógico). No treinamento, use como otimizador:

- a) Método GD.
- b) Método SGD.

**E.2.2.4.** Crie um perceptron que se ajuste ao seguinte conjunto de dados:

s	$x^{(s)}$	$y^{(s)}$
1	0.5	1.2
2	1.0	2.1
3	1.5	2.6
4	2.0	3.6

No treinamento, use como otimizador:

- a) Método GD.
- b) Método SGD.

## Capítulo 3

# Perceptron Multicamadas

### 3.1 Modelo MLP

Uma perceptron multicamadas (MLP, do inglês, *multilayer perceptron*) é um tipo de rede neural artificial formada por composições de camadas de perceptrons. Consultamos a Figura 3.1.

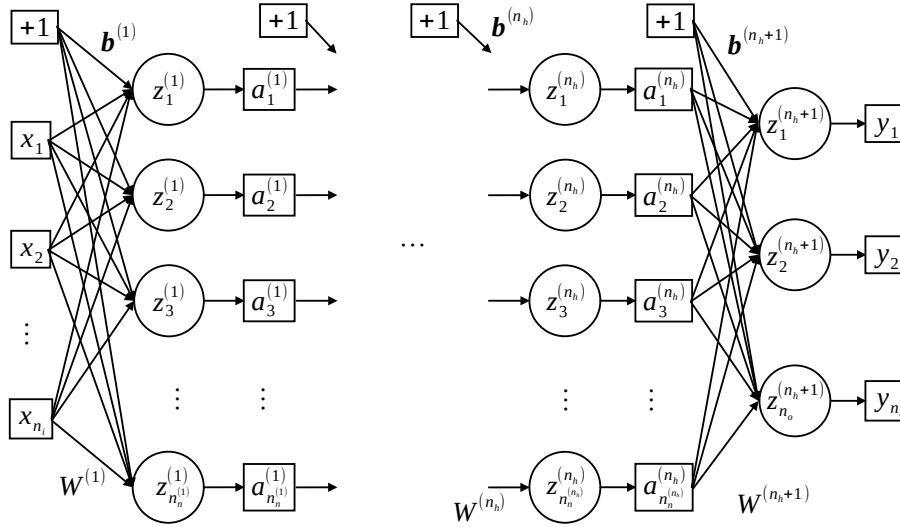


Figura 3.1: Arquitetura de uma rede do tipo perceptron multicamadas (MLP).

Denotamos uma MLP de  $n_l$  camadas por

$$\mathbf{y} = \mathcal{N} \left( \mathbf{x}; \left( W^{(l)}, \mathbf{b}^{(l)}, f^{(l)} \right)_{l=1}^{n_h+1} \right), \quad (3.1)$$

onde  $(W^{(l)}, \mathbf{b}^{(l)}, f^{(l)})$  é a tripa de **pesos**, **biases** e **função de ativação** da  $l$ -ésima camada da rede,  $l = 1, 2, \dots, n_h + 1$ . Uma rede com essa arquitetura é dita ter uma **camada de entrada**,  $n_h$  **camadas escondidas** e uma **camada de saída**.

A **saída** da rede é calculada por iteradas composições das camadas, i.e.

$$\mathbf{a}^{(l)} = f^{(l)} \left( \underbrace{W^{(l)} \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}}_{\mathbf{z}^{(l)}} \right), \quad (3.2)$$

para  $l = 1, 2, \dots, n_h + 1$ , denotando a **entrada** por  $\mathbf{x} =: \mathbf{a}^{(0)}$  e a **saída** por  $\mathbf{y} =: \mathbf{a}^{(n_h+1)}$ .

### 3.1.1 Treinamento

Em um treinamento supervisionado, tem-se um dado **conjunto de treinamento**  $\{\mathbf{x}^{(s)}, \mathbf{y}^{(s)}\}_{s=1}^{n_s}$ , com  $n_s$  amostras. O treinamento da rede consiste em resolver o problema de minimização

$$\min_{(W, \mathbf{b})} \left\{ \varepsilon := \frac{1}{n_s} \sum_{s=1}^{n_s} \varepsilon^{(s)}(\tilde{\mathbf{y}}^{(s)}, \mathbf{y}^{(s)}) \right\} \quad (3.3)$$

onde  $\varepsilon$  é uma dada **função erro** (em inglês, *loss function*) e  $\varepsilon^{(s)}$  é uma medida do erro da **saída estimada**  $\tilde{\mathbf{y}}^{(s)}$  da **saída esperada**  $\mathbf{y}^{(s)}$ .

O problema de minimização pode ser resolvido por um **método de declive** e, de forma geral, consiste em:

1.  $W, \mathbf{b}$  aproximações iniciais.
2. Para  $e \leftarrow 1, \dots, n_e$ :
  - (a)  $(W, \mathbf{b}) \leftarrow (W, \mathbf{b}) - l_r \mathbf{d}(\nabla_{W, \mathbf{b}} \varepsilon)$

onde,  $n_e$  é o **número de épocas**,  $l_r$  é uma dada **taxa de aprendizagem** (em inglês, *learning rate*) e  $\mathbf{d} = \mathbf{d}(\nabla_{W, \mathbf{b}} \varepsilon)$  é o vetor direção, onde

$$\nabla_{W, \mathbf{b}} \varepsilon := \left( \frac{\partial \varepsilon}{\partial W}, \frac{\partial \varepsilon}{\partial \mathbf{b}} \right) \quad (3.4)$$

$$= \frac{1}{n_s} \sum_{s=1}^{n_s} \left( \frac{\partial \varepsilon^{(s)}}{\partial W}, \frac{\partial \varepsilon^{(s)}}{\partial \mathbf{b}} \right) \quad (3.5)$$

O cálculo dos gradientes pode ser feito por **retropropagação** (em inglês, *backward*). Para os pesos da última camada, temos<sup>1</sup>

$$\frac{\partial \varepsilon^{(s)}}{\partial W^{(n_h+1)}} = \frac{\partial \varepsilon^{(s)}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{z}^{(n_h+1)}} \frac{\partial \mathbf{z}^{(n_h+1)}}{\partial W^{(n_h+1)}} \quad (3.6)$$

<sup>1</sup>Com um certo abuso de linguagem devido à álgebra matricial envolvida.



$$= \frac{\partial \varepsilon^{(s)}}{\partial \mathbf{y}} f' \left( W^{(n_h+1)} \mathbf{a}^{(n_h)} + \mathbf{b}^{(n_h+1)} \right) \mathbf{a}^{(n_h)}. \quad (3.7)$$

Para os pesos da penúltima camada, temos

$$\frac{\partial \varepsilon^{(s)}}{\partial W^{(n_h)}} = \frac{\partial \varepsilon}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{z}^{(n_h+1)}} \frac{\partial \mathbf{z}^{(n_h+1)}}{\partial W^{(n_h)}}, \quad (3.8)$$

$$= \frac{\partial \varepsilon^{(s)}}{\partial \mathbf{y}} f' \left( \mathbf{z}^{(n_h+1)} \right) \frac{\partial \mathbf{z}^{(n_h+1)}}{\partial \mathbf{a}^{(n_h)}} \frac{\partial \mathbf{a}^{(n_h)}}{\partial \mathbf{z}^{(n_h)}} \frac{\partial \mathbf{z}^{(n_h)}}{\partial W^{(n_h)}} \quad (3.9)$$

$$= \frac{\partial \varepsilon^{(s)}}{\partial \mathbf{y}} f' \left( \mathbf{z}^{(n_h+1)} \right) W^{(n_h+1)} f' \left( \mathbf{z}^{(n_h)} \right) \mathbf{a}^{(n_h-1)} \quad (3.10)$$

e assim, sucessivamente para as demais camadas da rede. Os gradientes em relação aos *biases* podem ser calculados de forma análoga.

### 3.1.2 Aplicação: Problema de Classificação XOR

Vamos desenvolver uma MLP que faça a operação **xor** (ou exclusivo). A rede recebe como entrada dois valores lógicos  $A_1$  e  $A_2$  (V, verdadeiro ou F, falso) e fornece como saída o valor lógico  $R = A_1 \text{ xor } A_2$ . Consultamos a tabela verdade:

$A_1$	$A_2$	$R$
V	V	F
V	F	V
F	V	V
F	F	F

Assumindo  $V = 1$  e  $F = -1$ , podemos modelar o problema tendo entradas  $\mathbf{x} = (x_1, x_2)$  e saída  $y$  como na seguinte tabela:

$x_1$	$x_2$	$y$
1	1	-1
1	-1	1
-1	1	1
-1	-1	-1

## Modelo

Vamos usar uma MLP de estrutura  $2 - 2 - 1$  e com funções de ativação  $f^{(1)}(\mathbf{x}) = \tanh(\mathbf{x})$  e  $f^{(2)}(\mathbf{x}) = id(\mathbf{x})$ . Ou seja, nossa rede tem duas entradas, uma **camada escondida** com 2 unidades (função de ativação tangente hiperbólica) e uma camada de saída com uma unidade (função de ativação identidade).

## Treinamento

Para o treinamento, vamos usar a função **erro quadrático médio** (em inglês, *mean squared error*)

$$\varepsilon := \frac{1}{n_s} \sum_{s=1}^{n_s} |\tilde{y}^{(s)} - y^{(s)}|^2, \quad (3.11)$$

onde  $\tilde{y}^{(s)} = \mathcal{N}(\mathbf{x}^{(s)})$  são os valores estimados e  $\{\mathbf{x}^{(s)}, y^{(s)}\}_{s=1}^{n_s}$ ,  $n_s = 4$ , o conjunto de treinamento conforme na tabela acima.

## Implementação

O seguinte código implementa a **MLP com Método do Gradiente Descendente (DG)** como otimizador do algoritmo de treinamento.

Código 3.1: mlp\_xor.py

```
1 import torch
2
3 # modelo
4
5 model = torch.nn.Sequential()
6 model.add_module('layer_1', torch.nn.Linear(2,2))
7 model.add_module('fun_1', torch.nn.Tanh())
8 model.add_module('layer_2', torch.nn.Linear(2,1))
9
10
11 # treinamento
12
13 ## otimizador
14 optim = torch.optim.SGD(model.parameters(),
```

```
15         lr=5e-1)
16
220 17 ## dados de treinamento
18 X_train = torch.tensor([[1., 1.],
19                          [1., -1.],
20                          [-1., 1.],
200 21                          [-1., -1.]])
22 y_train = torch.tensor([-1., 1., 1., -1.]).reshape(-1,1)
23
24 print("\nDados de treinamento")
180 25 print("X_train =")
26 print(X_train)
27 print("y_train = ")
28 print(y_train)
160 29
30 ## num max épocas
31 nepochs = 5000
32 tol = 1e-3
140 33
34 for epoch in range(nepochs):
35
36     # forward
120 37     y_est = model(X_train)
38
39     # função erro
40     loss = torch.mean((y_est - y_train)**2)
100 41
42     print(f'{epoch}: {loss.item():.4e}')
43
44     # critério de parada
80 45     if (loss.item() < tol):
46         break
47
48     # backward
60 49     optim.zero_grad()
50     loss.backward()
51     optim.step()
40 52
```

```
53  
54 # verificação  
55 y = model(X_train)  
56 print(f'y_est = {y}')
```

### 3.1.3 Exercícios

**E.3.1.1.** Faça uma nova versão do Código , de forma que a MLP tenha tangente hiperbólica como função de ativação na sua saída.

**E.3.1.2.** Faça uma nova versão do Código usando o método do gradiente estocástico (SGD) como otimizador no algoritmo de treinamento.

**E.3.1.3.** Crie uma MLP para emular a operação lógica  $\wedge$  (e-lógico). No treinamento, use como otimizador:

- a) Método GD.
- b) Método SGD.

**E.3.1.4.** Crie uma MLP para emular a operação lógica  $\vee$  (ou-lógico). No treinamento, use como otimizador:

- a) Método GD.
- b) Método SGD.

**E.3.1.5.** Considere uma MLP com  $n_l = 3$  camadas escondidas. Sendo  $\varepsilon$  uma dada função erro, calcule:

1.  $\frac{\partial \varepsilon}{\partial W^{n_l-2}}.$

$$2. \frac{\partial \varepsilon}{\partial \mathbf{b}^{n_l-2}}.$$

## 3.2 Aplicação: Problema de Classificação Binária

Em construção

Vamos estudar uma aplicação de redes neurais artificiais em um problema de classificação binária não linear.

### 3.2.1 Dados

Em construção

Vamos desenvolver uma rede do tipo Perceptron Multicamadas (MLP) para a classificação binária de pontos, com base nos seguintes dados.

```

1 from sklearn.datasets import make_circles
2 import matplotlib.pyplot as plt
3
4 plt.rcParams.update({
5     "text.usetex": True,
6     "font.family": "serif",
7     "font.size": 14
8 })
9
10 # data
11 print('data')
12 n_samples = 1000
13 print(f'n_samples = {n_samples}')
14 # X = points, y = labels
15 X, y = make_circles(n_samples,
16                     noise=0.03, # add noise
17                     random_state=42) # random seed
18

```

```

19 fig = plt.figure()
20 ax = fig.add_subplot()
21 ax.scatter(X[:,0], X[:,1], c=y, cmap=plt.cm.coolwarm)
22 ax.grid()
23 ax.set_xlabel('$x_1$')
24 ax.set_ylabel('$x_2$')
25 plt.show()

```

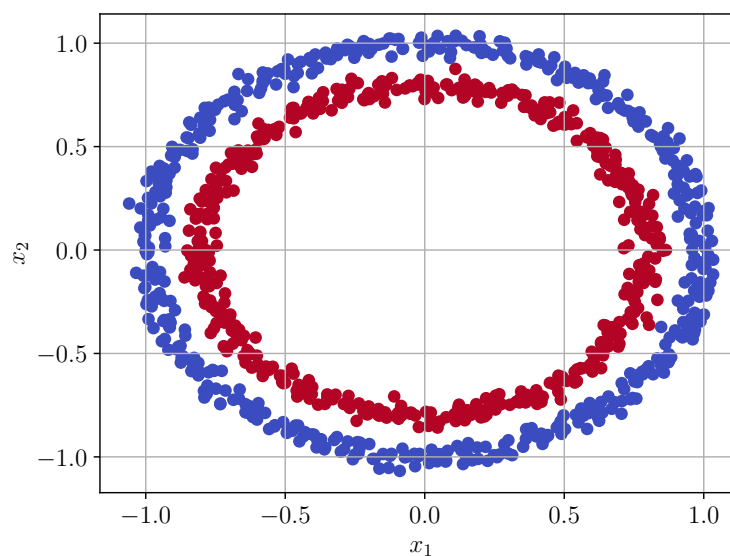


Figura 3.2: Dados para a o problema de classificação binária não linear.

### 3.2.2 Modelo

Em construção

Vamos usar uma MLP de estrutura 2-10-1, com função de ativação

$$\text{elu}(x) = \begin{cases} x & , x > 0 \\ \alpha(e^x - 1) & , x \leq 0 \end{cases} \quad (3.12)$$

na camada escondida e

$$\text{sigmoid}(x) = \frac{1}{1 + e^x} \quad (3.13)$$

na saída da rede.

Para o treinamento e teste, vamos randomicamente separar os dados em um conjunto de treinamento  $\{\mathbf{x}_{\text{train}}^{(k)}, y_{\text{train}}^{(k)}\}_{k=1}^{n_{\text{train}}}$  e um conjunto de teste  $\{\mathbf{x}_{\text{test}}^{(k)}, y_{\text{test}}^{(k)}\}_{k=1}^{n_{\text{test}}}$ , com  $y = 0$  para os pontos azuis e  $y = 1$  para os pontos vermelhos.

### 3.2.3 Treinamento e Teste

Em construção

Código 3.2: mlp\_classbin.py

```
1 import torch
2 from sklearn.datasets import make_circles
3 from sklearn.model_selection import train_test_split
4 import matplotlib.pyplot as plt
5
6 # data
7 print('data')
8 n_samples = 1000
9 print(f'n_samples = {n_samples}')
10 # X = points, y = labels
11 X, y = make_circles(n_samples,
12                     noise=0.03, # add noise
13                     random_state=42) # random seed
14
15 ## numpy -> torch
16 X = torch.from_numpy(X).type(torch.float)
17 y = torch.from_numpy(y).type(torch.float).reshape(-1,1)
18
19 ## split into train and test datasets
20 print('Data: train and test sets')
21 X_train, X_test, y_train, y_test = train_test_split(X,
22                                                     y,
23                                                     test_size=0.2,
```

```

24 random_state=
25 print(f'n_train = {len(X_train)}')
26 print(f'n_test = {len(X_test)}')
27 plt.close()
28 plt.scatter(X_train[:,0], X_train[:,1], c=y_train,
29             marker='o', cmap=plt.cm.coolwarm, alpha=0.3)
30 plt.scatter(X_test[:,0], X_test[:,1], c=y_test,
31             marker='*', cmap=plt.cm.coolwarm)
32 plt.show()
33
34 # model
35 model = torch.nn.Sequential(
36     torch.nn.Linear(2, 10),
37     torch.nn.ELU(),
38     torch.nn.Linear(10, 1),
39     torch.nn.Sigmoid()
40 )
41
42 # loss fun
43 loss_fun = torch.nn.BCELoss()
44
45 # optimizer
46 optimizer = torch.optim.SGD(model.parameters(),
47                               lr = 1e-1)
48
49 # evaluation metric
50 def accuracy_fun(y_pred, y_exp):
51     correct = torch.eq(y_pred, y_exp).sum().item()
52     acc = correct/len(y_exp) * 100
53     return acc
54
55 # train
56 n_epochs = 10000
57 n_out = 100
58
59 for epoch in range(n_epochs):
60     model.train()
61

```



```
62     y_pred = model(X_train)
63
220 64     loss = loss_fun(y_pred, y_train)
65
66     acc = accuracy_fun(torch.round(y_pred),
67                           y_train)
200 68
69     optimizer.zero_grad()
70     loss.backward()
71     optimizer.step()
180 72
73     model.eval()
74
75     #testing
160 76     if ((epoch+1) % n_out == 0):
77         with torch.inference_mode():
78             y_pred_test = model(X_test)
79             loss_test = loss_fun(y_pred_test,
140 80                                 y_test)
81             acc_test = accuracy_fun(torch.round(y_pred_test),
82                                     y_test)
83
120 84     print(f'{epoch+1}: loss = {loss:.5e}, accuracy = {acc:.2f}%')
85     print(f'\ttest: loss = {loss:.5e}, accuracy = {acc:.2f}%\n')
```

### 3.2.4 Verificação

Em construção

Para a verificação, testamos o modelo em uma malha uniforme de  $100 \times 100$  pontos no domínio  $[-1, 1]^2$ . Consulte a Figure 3.3.

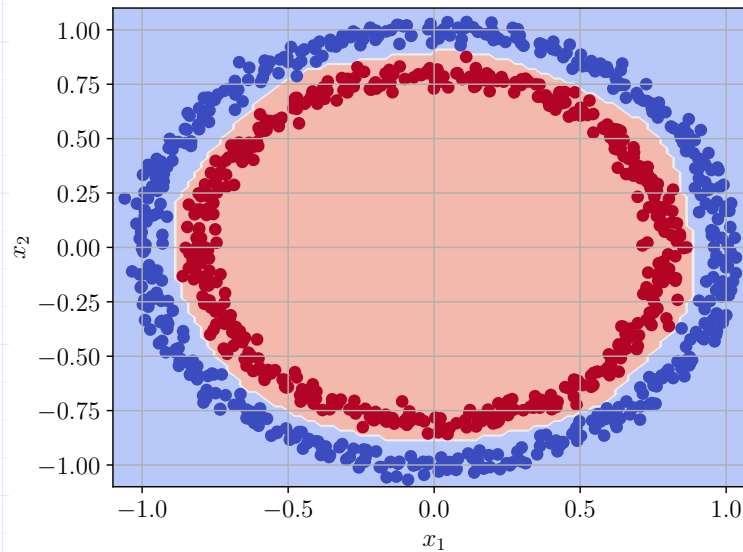


Figura 3.3: Verificação do modelo de classificação binária.

```
1 # malha de pontos
2 xx = torch.linspace(-1.1, 1.1, 100)
3 Xg, Yg = torch.meshgrid(xx, xx)
4
5 # valores estimados
6 Zg = torch.empty_like(Xg)
7 for i, xg in enumerate(xx):
8     for j, yg in enumerate(xx):
9         z = model(torch.tensor([[xg, yg]])).detach()
10        Zg[i, j] = torch.round(z)
11
12 # visualização
13 fig = plt.figure()
14 ax = fig.add_subplot()
15 ax.contourf(Xg, Yg, Zg, levels=2, cmap=plt.cm.coolwarm, alpha=0.5)
16 ax.scatter(X[:,0], X[:,1], c=y, cmap=plt.cm.coolwarm)
17 plt.show()
```

### 3.2.5 Exercícios

Em construção

## 3.3 Aplicação: Aproximação de Funções

Redes Perceptron Multicamadas (MLPs) são aproximadoras universais. Nesta seção, vamos aplicá-las na aproximação de funções uni- e bidimensionais.

### 3.3.1 Função unidimensional

Vamos criar uma MLP para aproximar a função

$$y = \sin(\pi x), \quad (3.14)$$

para  $x \in [-1, 1]$ .

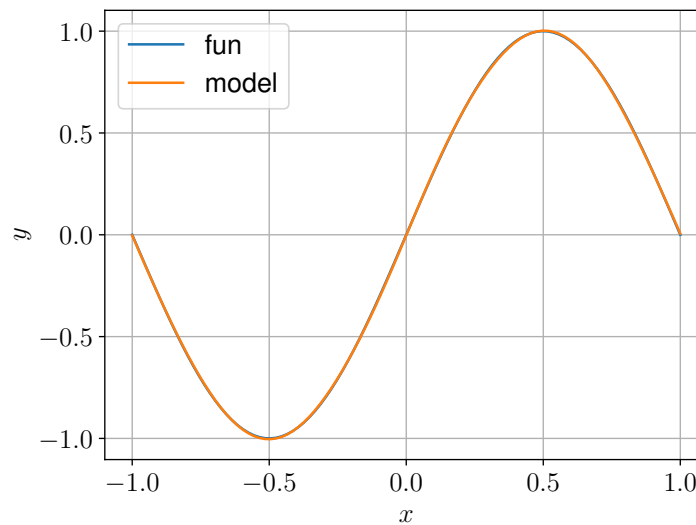


Figura 3.4: Aproximação da MLP da função  $y = \sin(\pi x)$ .

Código 3.3: mlp\_apfun\_1d

```
1 import torch
2 import matplotlib.pyplot as plt
```

Notas de Aula - Pedro Konzen \*/\* Licença CC-BY-SA 4.0

```
3
4 # modelo
5
6 model = torch.nn.Sequential()
7 model.add_module('layer_1', torch.nn.Linear(1,25))
8 model.add_module('fun_1', torch.nn.Tanh())
9 model.add_module('layer_2', torch.nn.Linear(25,25))
10 model.add_module('fun_2', torch.nn.Tanh())
11 model.add_module('layer_3', torch.nn.Linear(25,1))
12
13 # treinamento
14
15 ## fun obj
16 fun = lambda x: torch.sin(torch.pi*x)
17 a = -1.
18 b = 1.
19
20 ## otimizador
21 optim = torch.optim.SGD(model.parameters(),
22                           lr=1e-1, momentum=0.9)
23
24 ## num de amostras por época
25 ns = 100
26 ## num max épocas
27 nepochs = 5000
28 ## tolerância
29 tol = 1e-5
30
31 ## amostras de validação
32 X_val = torch.linspace(a, b, steps=100).reshape(-1,1)
33 y_vest = fun(X_val)
34
35 for epoch in range(nepochs):
36
37     # amostras
38     X_train = (a - b) * torch.rand((ns,1)) + b
39     y_train = fun(X_train)
40
```

```
41     # forward
42     y_est = model(X_train)
220 43
44     # erro
45     loss = torch.mean((y_est - y_train)**2)
46
200 47     print(f'{epoch}: {loss.item():.4e}')
48
49     # backward
50     optim.zero_grad()
180 51     loss.backward()
52     optim.step()
53
54     # validação
160 55     y_val = model(X_val)
56     loss_val = torch.mean((y_val - y_val)**2)
57     print(f"\tloss_val = {loss_val.item():.4e}")
58
140 59     # critério de parada
60     if (loss_val.item() < tol):
61         break
62
120 63
64 # verificação
65 fig = plt.figure()
66 ax = fig.add_subplot()
100 67
68 x = torch.linspace(a, b,
69                     steps=100).reshape(-1,1)
70
80 71 y_esp = fun(x)
72 ax.plot(x, y_esp, label='fun')
73
74 y_est = model(x)
60 75 ax.plot(x, y_est.detach(), label='model')
76
77 ax.legend()
40 78 ax.grid()
```

```

79 ax.set_xlabel('x')
80 ax.set_ylabel('y')
81 plt.show()

```

### 3.3.2 Função bidimensional

Vamos criar uma **MLP** para aproximar a função bidimensional

$$y = \text{sen}(\pi x_1) \text{sen}(\pi x_2), \quad (3.15)$$

para  $(x_1, x_2) \in \mathcal{D} := [-1, 1]^2$ .

Vamos usar uma **arquitetura de rede**  $2 - n_n \times 3 - 1$  (duas entradas, 3 camadas escondidas com  $n_n$  neurônios e uma saída). Nas  $n_h = 3$  camadas escondidas, vamos usar a **tangente hiperbólica** como função de ativação.

Para o treinamento, vamos usar o **erro médio quadrático** como função erro

$$\varepsilon = \frac{1}{n_s} \sum_{s=1}^{n_s} |\hat{y}^{(s)} - y^{(s)}|^2, \quad (3.16)$$

onde, a cada época,  $n_s$  pontos randômicos<sup>2</sup>  $\{\mathbf{x}^{(s)}\} \subset \mathcal{D}$  são usados para gerar o conjunto de treinamento  $\{(\mathbf{x}^{(s)}, y^{(s)})\}_{s=1}^{n_s}$ .

---

<sup>2</sup>Em uma distribuição uniforme.

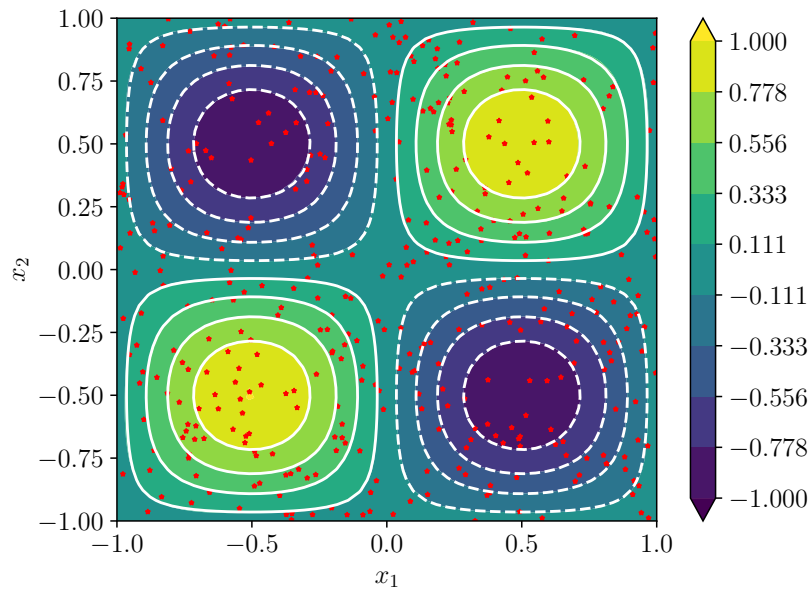


Figura 3.5: Aproximação MLP da função  $y = \sin(\pi x_1) \sin(\pi x_2)$ . Linhas: isolinhas da função. Mapa de cores: MLP. Estrelas: pontos de treinamentos na última época.

Código 3.4: mlp\_apfun\_2d

```

1  import torch
2
3  # modelo
4  nn = 50
5  model = torch.nn.Sequential()
6  model.add_module('layer_1', torch.nn.Linear(2, nn))
7  model.add_module('fun_1', torch.nn.Tanh())
8  model.add_module('layer_2', torch.nn.Linear(nn, nn))
9  model.add_module('fun_2', torch.nn.Tanh())
10 model.add_module('layer_3', torch.nn.Linear(nn, nn))
11 model.add_module('fun_3', torch.nn.Tanh())
12 model.add_module('layer_4', torch.nn.Linear(nn, 1))
13
14 # treinamento
15
16 ## fun obj

```

```
17 def fun(x1, x2):
18     return torch.sin(torch.pi*x1) * \
19         torch.sin(torch.pi*x2)
20
21 x1_a = -1.
22 x1_b = 1
23
24 x2_a = -1.
25 x2_b = 1.
26
27
28 ## otimizador
29 optim = torch.optim.SGD(model.parameters(),
30                          lr=1e-1, momentum=0.9)
31
32 ## num de amostras por época
33 ns = 20
34 ## num max épocas
35 nepochs = 50000
36 ## tolerância
37 tol = 1e-4
38
39 ## amostras de validação
40 n_val = 50
41 x1 = torch.linspace(x1_a, x1_b, steps=n_val)
42 x2 = torch.linspace(x2_a, x2_b, steps=n_val)
43 X1_val, X2_val = torch.meshgrid(x1, x2, indexing='ij')
44 X_val = torch.hstack((X1_val.reshape(n_val**2,1),
45                        X2_val.reshape(n_val**2,1)))
46 Y_val = fun(X1_val, X2_val).reshape(-1,1)
47
48 for epoch in range(nepochs):
49
50     # amostras
51     X1 = (x1_b - x1_a) * torch.rand(ns**2, 1) + x1_a
52     X2 = (x2_b - x2_a) * torch.rand(ns**2, 1) + x2_a
53     # X1, X2 = torch.meshgrid(x1, x2, indexing='ij')
54     X_train = torch.hstack((X1, X2))
```



```
55     Y_train = fun(X1, X2).reshape(-1,1)
56
220 57
58     # forward
59     Y_est = model(X_train)
60
200 61
62     # erro
63     loss = torch.mean((Y_est - Y_train)**2)
64
180 65     if (epoch % 100 == 0):
66         print(f'{epoch}: {loss.item():.4e}')
67
68     # backward
69     optim.zero_grad()
160 70     loss.backward()
71     optim.step()
72
140 73     # validação
74     if (epoch % 100 == 0):
75         Y_val = model(X_val)
76         loss_val = torch.mean((Y_val - Y_vest)**2)
120 77
78         print(f"\tloss_val = {loss_val.item():.4e}")
79
79     # critério de parada
80     if (loss_val.item() < tol):
100 81         break
```

### 3.3.3 Exercícios

**E.3.3.1.** Crie uma MLP para aproximar a função gaussiana

$$y = e^{-x^2} \quad (3.17)$$

para  $x \in [-1, 1]$ .

**E.3.3.2.** Crie uma MLP para aproximar a função  $y = \sin(x)$  para  $x \in$

$[-\pi, \pi]$ .

**E.3.3.3.** Crie uma MLP para aproximar a função  $y = \sin(x) + \cos(x)$  para  $x \in [0, 2\pi]$ .

**E.3.3.4.** Crie uma MLP para aproximar a função gaussiana

$$z = e^{-(x^2+y^2)} \quad (3.18)$$

para  $(x, y) \in [-1, 1]^2$ .

**E.3.3.5.** Crie uma MLP para aproximar a função  $y = \sin(x_1) \cos(x_2)$  para  $(x_1, x_2) \in [0, \pi] \times [-\pi, 0]$ .

**E.3.3.6.** Crie uma MLP para aproximar a função  $y = \sin(x_1) + \cos(x_2)$  para  $(x_1, x_2) \in [-2\pi, 2\pi]$ .

## 3.4 Diferenciação Automática

**Diferenciação automática** é um conjunto de técnicas para a computação de derivadas numéricas em um programa de computador. Explora-se o fato de que um programa computacional executa uma sequência de operações aritméticas e funções elementares, podendo-se computar a derivada por aplicações da **regra da cadeia**.

PyTorch computa o **gradiente** (derivada) de uma função  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  a partir de seu **grafo computacional**. Os gradientes são computados por retropropagação. Por exemplo, para a computação do gradiente

$$\nabla_{\mathbf{x}} f(\mathbf{x}_0) = \left. \frac{df}{d\mathbf{x}} \right|_{\mathbf{x}=\mathbf{x}_0}, \quad (3.19)$$

primeiramente, propaga-se a entrada  $\mathbf{x}_0$  pela função computacional  $f$ , obtendo-se  $y = f(\mathbf{x}_0)$ . Então, o gradiente é computado por retropropagação.

**Exemplo 3.4.1.** Consideramos a função  $f(x) = \text{sen}(\pi x)$  e vamos computar

$$f'(x_0) = \left. \frac{df}{dx} \right|_{x=0} \quad (3.20)$$

por diferenciação automática.

Antes, observamos que, pela regra da cadeia, denotamos  $u = \pi x$  e calculamos

$$\frac{df}{dx} = \frac{d}{du} \text{sen}(u) \cdot \frac{du}{dx} \quad (3.21)$$

$$= \cos(u) \cdot \pi \quad (3.22)$$

$$= \pi \cos(\pi x) \quad (3.23)$$

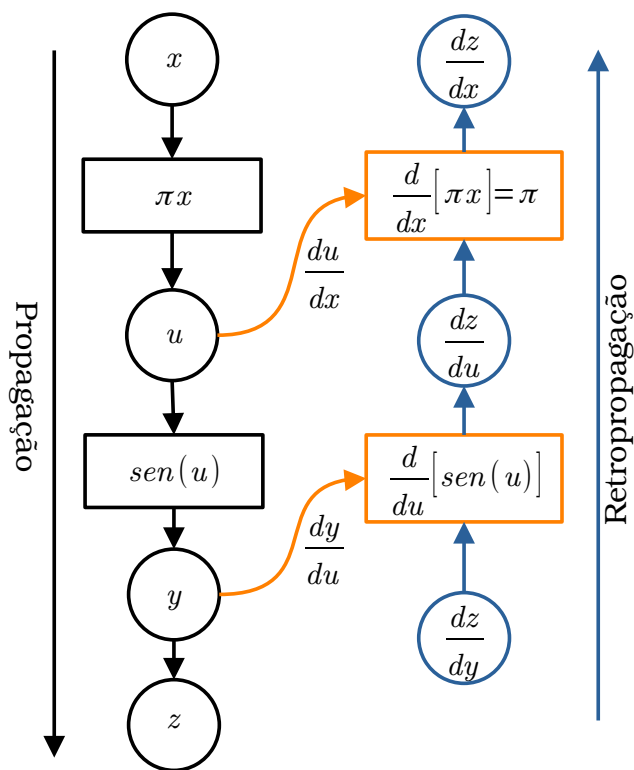


Figura 3.6: Grafo computacional da diferenciação automática de  $f(x) = \text{sen}(\pi x)$ .

Agora, observamos que a computação de  $f(x)$  pode ser representada pelo grafo de propagação mostrado na Figura 3.6. Para a computação do gradiente, adicionamos uma variável fictícia  $z = y$ . Na retropropagação, computamos

$$\text{a. } \frac{dz}{dy} = 1 \quad (3.24a)$$

$$\begin{aligned} \text{b. } \frac{dz}{du} &= \frac{dy}{du} \frac{dz}{dy} \\ &= \frac{d}{du} [\text{sen}(u)] \cdot 1 \\ &= \cos(u) \end{aligned} \quad (3.24b)$$

$$\text{c. } \frac{dz}{dx} = \frac{du}{dx} \frac{dz}{du} \quad (3.24c)$$

$$= \frac{d}{dx} [\pi x] \cos(u) \quad (3.24d)$$

$$= \pi \cos(\pi x) = \frac{dy}{dx}. \quad (3.24e)$$

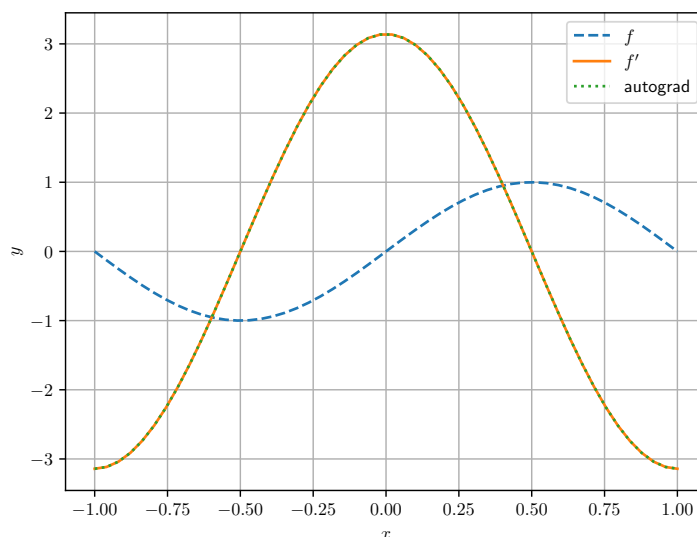


Figura 3.7: Comparação entre as diferenciações analítica ( $f'$ ) e automática (autograd).

Código 3.5: mlp\_autograd\_df1d

```
1 import torch
2
3 # input
4 x = torch.linspace(-1., 1., steps=50).reshape(-1,1)
5 # requires_grad
6 x.requires_grad = True
7
8 # output
9 y = torch.sin(torch.pi*x)
10
11 # compute gradients
12 y.backward(gradient=torch.ones_like(y))
13
14 # dy/dx
15 dydx = x.grad
```

A computação do gradiente também acaba por construir um novo grafo (consulte Figura 3.6). Este, por sua vez, pode ser usado para a computação da diferenciação automática de segunda ordem, i.e. para a derivação de segunda ordem.

**Exemplo 3.4.2.** Consideramos a função  $y = \sin(\pi x)$ . No exemplo anterior, computamos  $dy/dx = \pi \cos(\pi x)$  por diferenciação automática. No Código 3.5, os gradientes foram computados com o comando

```
1 y.backward(gradient=torch.ones_like(y))
2 dudx = x.grad
```

Alternativamente, podemos usar

```
1 dydx = torch.autograd.grad(
2     y, x,
3     grad_outputs=torch.ones_like(y),
4     retain_graph=True,
5     create_graph=True)[0]
```

Este comando computa  $dy/dx$ , mas avisa o [PyTorch](#) que os grafos computacionais sejam mantidos e que um novo grafo seja gerado da retropropagação. Com isso, podemos computar o gradiente do gradiente, como no código abaixo.

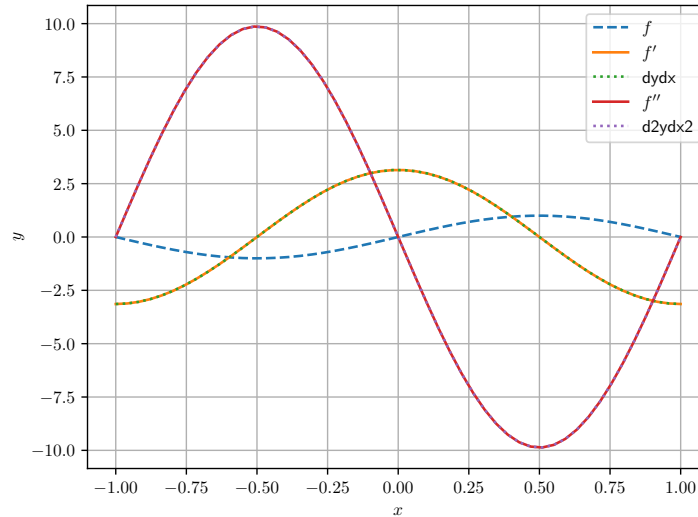


Figura 3.8: Comparação entre as diferenciações analítica ( $f'$ ,  $f''$ ) e automática ( $dydx$ ,  $d2ydx2$ ).

Código 3.6: mlp\_autograd\_d2f1d

```
1 import torch
2
3 # input
4 x = torch.linspace(-1., 1., steps=50).reshape(-1,1)
5 # requires_grad
6 x.requires_grad = True
7
8 # output
9 y = torch.sin(torch.pi*x)
10
11 # compute gradients
12 dydx = torch.autograd.grad(
13     y, x,
14     grad_outputs=torch.ones_like(y),
```

```

15     retain_graph=True,
16     create_graph=True)[0]
17
18 d2ydx2 = torch.autograd.grad(
19     dydx, x,
20     grad_outputs=torch.ones_like(dydx))[0]

```

### 3.4.1 Autograd MLP

Os conceitos de diferenciação automática (**autograd**) são diretamente estendidos para redes do tipo Perceptron Multicamadas (MLP, do inglês, *Multi-layer Perceptron*). Uma MLP é uma composição de funções definidas por parâmetros (pesos e *biases*). Seu treinamento ocorre em duas etapas<sup>3</sup>:

1. **Propagação (*forward*)**: os dados de entrada são propagados para todas as funções da rede, produzindo a saída estimada.
2. **Retropropagação (*backward*)**: a computação do gradiente do erro<sup>4</sup> em relação aos parâmetros da rede é realizado coletando as derivadas (gradientes) das funções da rede. Pela regra da cadeia, essa coleta é feita a partir da camada de saída em direção a camada de entrada da rede.

No seguinte exemplo, exploramos o fato de MLPs serem aproximadoras universais e avaliamos a derivada de uma MLP na aproximação de uma função.

**Exemplo 3.4.3.** Vamos criar uma MLP

$$\tilde{y} = \mathcal{N}\left(x; \left(W^{(l)}, \mathbf{b}^{(l)}, f^{(l)}\right)_{l=1}^n\right), \quad (3.25)$$

que aproxima a função

$$y = \text{sen}(\pi x), \quad x \in [-1, 1]. \quad (3.26)$$

Em seguida, computamos, por diferenciação automática, o gradiente

$$\frac{d\tilde{y}}{dx} = \nabla_x \mathcal{N}(x) \quad (3.27)$$

<sup>3</sup>Para mais detalhes, consulte a Subseção 3.1.1.

<sup>4</sup>Medida da diferença entre o valor estimado e o valor esperado.

e comparamos com o resultado esperado

$$\frac{dy}{dx} = \pi \cos(\pi x). \quad (3.28)$$

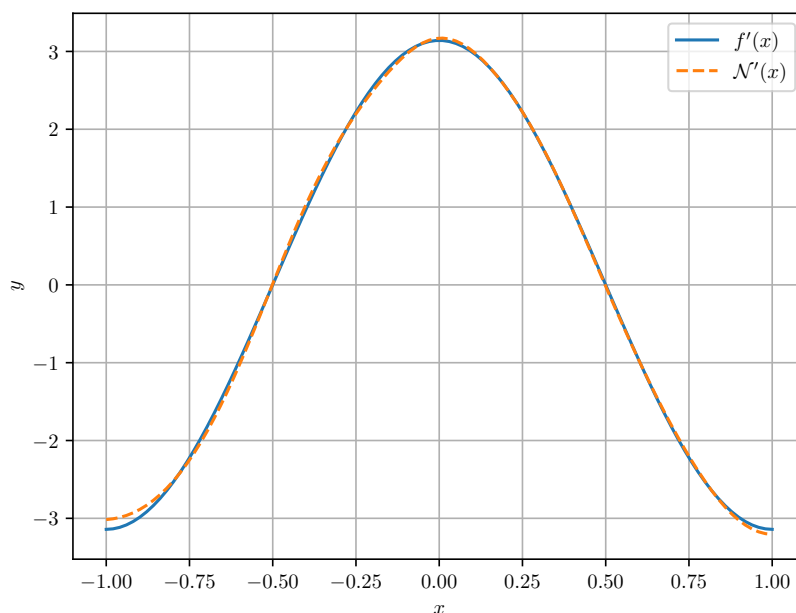


Figura 3.9: Comparação da diferenciação automática da MLP com a derivada analítica  $f'(x) = \pi \cos(\pi x)$ .

Código 3.7: mlp\_autograd\_apfun1d.py

```
1 import torch
2 from torch import nn
3 from torch import autograd
4
5 # modelo
6
7 model = torch.nn.Sequential()
8 model.add_module('layer_1', torch.nn.Linear(1,25))
9 model.add_module('fun_1', torch.nn.Tanh())
10 model.add_module('layer_2', torch.nn.Linear(25,25))
11 model.add_module('fun_2', torch.nn.Tanh())
```



```
12 model.add_module('layer_3', torch.nn.Linear(25,1))
13
14 # treinamento
15
16 ## fun obj
17 fun = lambda x: torch.sin(torch.pi*x)
18 a = -1.
19 b = 1.
20
21 ## otimizador
22 optim = torch.optim.SGD(model.parameters(),
23                           lr=1e-1, momentum=0.9)
24
25 ## num de amostras por época
26 ns = 100
27 ## num max épocas
28 nepochs = 5000
29 ## tolerância
30 tol = 1e-5
31
32 ## amostras de validação
33 X_val = torch.linspace(a, b, steps=100).reshape(-1,1)
34 y_vest = fun(X_val)
35
36 for epoch in range(nepochs):
37
38     # amostras
39     X_train = (a - b) * torch.rand((ns,1)) + b
40     y_train = fun(X_train)
41
42     # forward
43     y_est = model(X_train)
44
45     # erro
46     loss = torch.mean((y_est - y_train)**2)
47
48     print(f'{epoch}: {loss.item():.4e}')
49
```

```
50     # backward
51     optim.zero_grad()
52     loss.backward()
53     optim.step()
54
55     # validação
56     y_val = model(X_val)
57     loss_val = torch.mean((y_val - y_vest)**2)
58     print(f"\tloss_val = {loss_val.item():.4e}")
59
60     # critério de parada
61     if (loss_val.item() < tol):
62         break
63
64 # autograd MLP
65 X_val.requires_grad = True
66 # forward
67 y_val = model(X_val)
68 # gradient
69 dydx = autograd.grad(
70     y_val, X_val,
71     grad_outputs=torch.ones_like(y_val))[0]
```

### 3.4.2 Exercícios

**E.3.4.1.** Por diferenciação automática, compute o gradiente (a derivada) das seguintes funções

- a)  $f(x) = x^2 - 2x + 1$  para valores  $x \in [-2, 2]$ .
- b)  $g(x) = \cos^2(x)$  para valores  $x \in [0, 2\pi]$ .
- c)  $h(x) = \ln(x - 1)$  para valores  $x \in (-1, 2]$ .
- d)  $u(t) = e^{-t^2} \sin(t)$  para valores  $t \in [-\pi, \pi]$ .

Em cada caso, compare os valores computados com os valores esperados.

**E.3.4.2.** Em cada item do Exercício 3.4.1, faça um fluxograma dos grafos computacionais da propagação e da retropropagação na computação dos gradientes.

**E.3.4.3.** Em cada item do Exercício 3.4.1, compute a derivada de segunda ordem da função indicada. Compare os valores computados com os valores esperados.

**E.3.4.4.** Por diferenciação automática, compute os gradientes das seguintes funções:

a)  $f(x, y) = x^2 + y^2$  para valores  $(x, y) \in [-1, 1]^2$ .

b)  $g(x, y) = e^x \sin(xy)$  para valores  $(x, y) \in (-1, 2) \times (0, \pi)$ .

Em cada caso, compare os valores computados com os valores esperados.

**E.3.4.5.** Para as funções de cada item do Exercício 3.4.6, compute:

a)  $\frac{\partial^2}{\partial x^2}$ .

b)  $\frac{\partial^2}{\partial x \partial y}$ .

c)  $\frac{\partial^2}{\partial y^2}$ .

Compare os valores computados com os valores esperados.

**E.3.4.6.** Em cada item do Exercício 3.4.6, compute o laplaciano  $\Delta = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)$  da função indicada. Compare os valores computados com os valores esperados.

**E.3.4.7.** Seja a função  $\mathbf{f} : \mathbb{R}^2 \rightarrow \mathbb{R}^2$  definida por

$$\mathbf{f}(x, y) = \begin{bmatrix} xy^2 - x^2y + 6 \\ x + x^2y^3 - 7 \end{bmatrix} \quad (3.29)$$

no domínio  $\mathcal{D} = [-1, 2] \times [1, 3]$ . Por diferenciação automática e para valores no domínio da função, compute:

a)  $\nabla f_1(x, y)$ .

b)  $\nabla f_2(x, y)$ .

c)  $\frac{\partial^2 f_1}{\partial x^2}$ .

d)  $\frac{\partial^2 f_1}{\partial x \partial y}$ .

e)  $\frac{\partial^2 f_1}{\partial y^2}$ .

f)  $\frac{\partial^2 f_2}{\partial x^2}$ .

g)  $\frac{\partial^2 f_2}{\partial x \partial y}$ .

h)  $\frac{\partial^2 f_2}{\partial y^2}$ .

## Capítulo 4

# Redes Informadas pela Física

[[tag:construcao]]

**Redes neurais informadas pela física** (PINNs, do inglês, *physics-informed neural networks*) são métodos de *deep learning* para a solução de equações diferenciais.

### 4.1 Aplicação: Equação de Poisson

Vamos criar uma **MLP para resolver o problema de Poisson**<sup>1</sup>

$$-\Delta u = f, \mathbf{x} \in \mathcal{D} = (-1, 1)^2, \quad (4.1a)$$

$$u = 0, \mathbf{x} \in \partial D, \quad (4.1b)$$

com fonte dada

$$f(x_1, x_2) = \pi^2 \sin(\pi x_1) \sin(\pi x_2). \quad (4.2)$$

No treinamento, vamos usar a **função erro baseada no resíduo da equação de Poisson** (4.1a) e nas condições de contorno (4.1b). Mais especificamente,

assumimos a função erro

$$\varepsilon := \underbrace{\frac{1}{n_{s,in}} \sum_{s=1}^{n_{s,in}} |\mathcal{R}(\tilde{u}^{(s)})|^2}_{\text{resíduo}} + \underbrace{\frac{1}{n_{s,cc}} \sum_{s=1}^{n_{s,cc}} |\tilde{u}^s|^2}_{\text{c.c.}}, \quad (4.3)$$

onde o resíduo é definido por

$$\mathcal{R}(\tilde{u}^{(s)}) := f + \Delta \tilde{u}^{(s)}. \quad (4.4)$$

A cada época, conjuntos de pontos  $\{\mathbf{x}^{(s)}\}_{s=1}^{n_{s,in}} \subset \mathcal{D}$  e  $\{\mathbf{x}^{(s)}\}_{s=1}^{n_{s,cc}} \subset \partial\mathcal{D}$  são randomicamente gerados com distribuição uniforme.

**Observação 4.1.1.** O problema de Poisson (4.1) tem solução analítica

$$u(x_1, x_2) = \sin(\pi x_1) \sin(\pi x_2). \quad (4.5)$$

É importante observar que o treinamento da MLP não depende de conhecermos a solução. Aqui, vamos usá-la apenas para compararmos a solução MLP com a analítica.

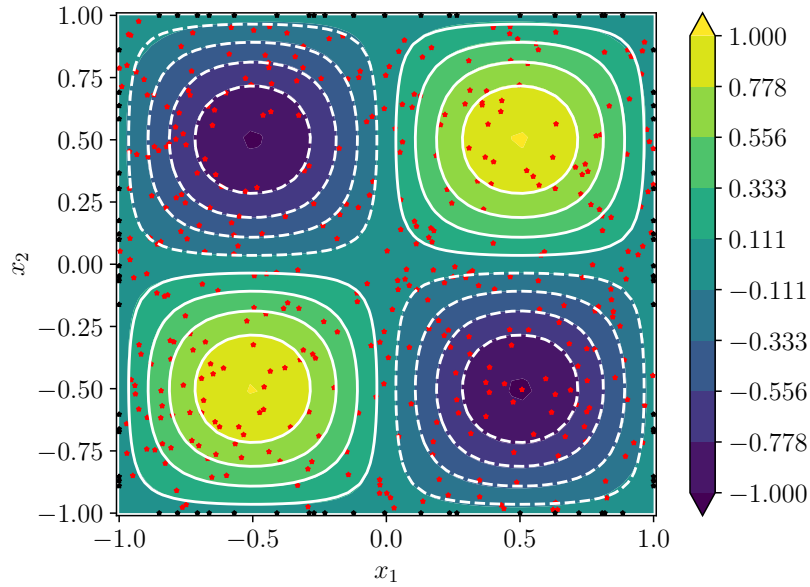


Figura 4.1: Aproximação MLP da função solução do problema de Poisson (4.1). Linhas: isolinhas da solução analítica. Mapa de cores: solução MLP. Estrelas: pontos de treinamentos na última época.

Código 4.1: py\_pinn\_poisson

```
1  import torch
2  from torch import pi, sin
3
4  # modelo
5  nn = 50
6  model = torch.nn.Sequential()
7  model.add_module('layer_1', torch.nn.Linear(2, nn))
8  model.add_module('fun_1', torch.nn.Tanh())
9  model.add_module('layer_2', torch.nn.Linear(nn, nn))
10 model.add_module('fun_2', torch.nn.Tanh())
11 model.add_module('layer_3', torch.nn.Linear(nn, nn))
12 model.add_module('fun_3', torch.nn.Tanh())
13 model.add_module('layer_4', torch.nn.Linear(nn, 1))
14
15 # otimizador
16 optim = torch.optim.SGD(model.parameters(),
17                           lr = 1e-3, momentum=0.9)
18
19 # fonte
20 def f(x1, x2):
21     return 2.*pi**2*sin(pi*x1)*sin(pi*x2)
22
23 # treinamento
24 ns_in = 400
25 ns_cc = 20
26 nepochs = 50000
27 tol = 1e-3
28
29 ## pontos de validação
30 ns_val = 50
31 x1_val = torch.linspace(-1., 1., steps=ns_val)
32 x2_val = torch.linspace(-1., 1., steps=ns_val)
33 X1_val, X2_val = torch.meshgrid(x1_val, x2_val, indexing='ij')
34 X_val = torch.hstack((X1_val.reshape(ns_val**2, 1),
35                        X2_val.reshape(ns_val**2, 1)))
36
```

```
37 for epoch in range(nepochs):
38
39     # forward
40     X1 = 2.*torch.rand(ns_in, 1) - 1.
41     X2 = 2.*torch.rand(ns_in, 1) - 1.
42     X = torch.hstack((X1, X2))
43     X.requires_grad = True
44
45     U = model(X)
46
47     # gradientes
48     D1U = torch.autograd.grad(
49         U, X,
50         grad_outputs=torch.ones_like(U),
51         retain_graph=True,
52         create_graph=True)[0]
53     D2UX1 = torch.autograd.grad(
54         D1U[:,0:1], X,
55         grad_outputs=torch.ones_like(D1U[:,0:1]),
56         retain_graph=True,
57         create_graph=True)[0]
58     D2UX2 = torch.autograd.grad(
59         D1U[:,1:2], X,
60         grad_outputs=torch.ones_like(D1U[:,1:2]),
61         retain_graph=True,
62         create_graph=True)[0]
63
64     # fonte
65     F = f(X1, X2)
66
67     # loss pts internos
68     lin = torch.mean((F + D2UX1[:,0:1] + D2UX2[:,1:2])**2)
69
70     # contornos
71     ## c.c. 1
72     X1 = 2.*torch.rand(ns_cc, 1) - 1.
73     Xcc1 = torch.hstack((X1, -torch.ones((ns_cc,1))))
74     Ucc1 = model(Xcc1)
```



```

75
76     ## c.c. 3
220 77     Xcc3 = torch.hstack((X1, torch.ones((ns_cc,1))))
78     Ucc3 = model(Xcc3)
79
80     ## c.c. 4
200 81     X2 = 2.*torch.rand(ns_cc, 1) - 1.
82     Xcc4 = torch.hstack((-torch.ones((ns_cc,1)), X2))
83     Ucc4 = model(Xcc4)
84
180 85     ## c.c. 2
86     Xcc2 = torch.hstack((torch.ones((ns_cc,1)), X2))
87     Ucc2 = model(Xcc2)
88
160 89     # loss cc
90     lcc = 1./(4.*ns_cc) * torch.sum(Ucc1**2 + Ucc2**2 + Ucc3**2 + Ucc4**2)
91
92     # loss
140 93     loss = lin + lcc
94
95     if ((epoch % 500 == 0) or (loss.item() < tol)):
96         print(f'{epoch}: loss = {loss.item():.4e}')
120 97
98         if (loss.item() < tol):
99             break
100
100 101     optim.zero_grad()
102     loss.backward()
103     optim.step()

```

### 4.1.1 Exercícios

#### E.4.1.1. Crie uma MLP para resolver

$$-\Delta u = 0, \mathbf{x} \in D = (0,1)^2, \quad (4.6)$$

$$u(x_1, 0) = x_1(1 - x_1), 0 \leq x_1 \leq 1, \quad (4.7)$$

$$u(1, x_2) = x_2(1 - x_2), 0 < x_2 \leq 1, \quad (4.8)$$

$$u(x_1, 1) = x_1(1 - x_1), 0 \leq x_1 < 1, \quad (4.9)$$

$$u(0, x_2) = x_2(1 - x_2), 0 < x_2 < 1. \quad (4.10)$$

## 4.2 Aplicação: Equação do Calor

Em construção

Consideramos o problema

$$u_t = u_{xx} + f, (t, x) \in (0, 1] \times (-1, 1), \quad (4.11a)$$

$$u(0, x) = \text{sen}(\pi x), x \in [-1, 1], \quad (4.11b)$$

$$u(t, -1) = u(t, 1) = 0, t \in (t_0, tf], \quad (4.11c)$$

onde  $f(t, x) = (\pi^2 - 1)e^{-t} \text{sen}(\pi x)$  é a fonte. Este problema foi manufaturado a partir da solução

$$u(t, x) = e^{-t} \text{sen}(\pi x). \quad (4.12)$$

Código 4.2: mlp\_calor\_autograd.py

```
1 import torch
2 from torch import pi, sin, exp
3 from collections import OrderedDict
4 import matplotlib.pyplot as plt
5
6 # modelo
7 hidden = [50]*8
8 activation = torch.nn.Tanh()
9 layerList = [('layer_0', torch.nn.Linear(2, hidden[0])),
10              ('activation_0', activation)]
11 for l in range(len(hidden)-1):
12     layerList.append((f'layer_{l+1}',
13                      torch.nn.Linear(hidden[l], hidden[l+1])))
14     layerList.append((f'activation_{l+1}', activation))
15 layerList.append((f'layer_{len(hidden)}', torch.nn.Linear(hidden[-1], 1)))
16 #layerList.append((f'activation_{len(hidden)}', torch.nn.Sigmoid()))
```

```
17 layerDict = OrderedDict(layerList)
18 model = torch.nn.Sequential(OrderedDict(layerDict))
19
20 # otimizador
21 # optim = torch.optim.SGD(model.parameters(),
22 #                           lr = 1e-3, momentum=0.85)
23 optim = torch.optim.Adam(model.parameters(),
24                           lr = 1e-2)
25 scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optim,
26                                                         factor=0.1,
27                                                         patience=100)
28
29 # treinamento
30 nt = 10
31 tt = torch.linspace(0., 1., nt+1)
32 nx = 20
33 xx = torch.linspace(-1., 1., nx+1)
34 T,X = torch.meshgrid(tt, xx, indexing='ij')
35 tt = tt.reshape(-1,1)
36 xx = xx.reshape(-1,1)
37
38 Sic = torch.hstack((torch.zeros_like(xx), xx))
39 Uic = sin(pi*xx)
40
41 Sbc0 = torch.hstack((tt[1:,:], -1.*torch.ones_like(tt[1:,:])))
42 Ubc0 = torch.zeros_like(tt[1:,:])
43
44 Sbc1 = torch.hstack((tt[1:,:], 1.*torch.ones_like(tt[1:,:])))
45 Ubc1 = torch.zeros_like(tt[1:,:])
46
47 tin = tt[1:,:]
48 xin = xx[1:-1,:]
49 Sin = torch.empty((nt*(nx-1), 2))
50 Fin = torch.empty((nt*(nx-1), 1))
51 s = 0
52 for i,t in enumerate(tin):
53     for j,x in enumerate(xin):
54         Sin[s,0] = t
```

```
55     Sin[s,1] = x
56     Fin[s,0] = (pi**2 - 1.)*exp(-t)*sin(pi*x)
57     s += 1
58 tin = torch.tensor(Sin[:,0:1], requires_grad=True)
59 xin = torch.tensor(Sin[:,1:2], requires_grad=True)
60 Sin = torch.hstack((tin,xin))
61
62 nepochs = 50001
63 tol = 1e-4
64 nout = 100
65
66 for epoch in range(nepochs):
67
68     # loss
69
70     ## c.i.
71     Uest = model(Sic)
72     lic = torch.mean((Uest - Uic)**2)
73
74     ## residual
75     U = model(Sin)
76     U_t = torch.autograd.grad(
77         U, tin,
78         grad_outputs=torch.ones_like(U),
79         retain_graph=True,
80         create_graph=True)[0]
81     U_x = torch.autograd.grad(
82         U, xin,
83         grad_outputs=torch.ones_like(U),
84         retain_graph=True,
85         create_graph=True)[0]
86     U_xx = torch.autograd.grad(
87         U_x, xin,
88         grad_outputs=torch.ones_like(U_x),
89         retain_graph=True,
90         create_graph=True)[0]
91     res = U_t - U_xx - Fin
92     lin = torch.mean(res**2)
```

```

93
94     ## c.c. x = -1
220 95     Uest = model(Sbc0)
96     lbc0 = torch.mean(Uest**2)
97
98     ## c.c. x = 1
200 99     Uest = model(Sbc1)
100     lbc1 = torch.mean(Uest**2)
101
102     loss = lin + lic + lbc0 + lbc1
180 103
104     lr = optim.param_groups[-1]['lr']
105     print(f'{epoch}: loss = {loss.item():.4e}, lr = {lr:.4e}')
106
160 107     # backward
108     scheduler.step(loss)
109     optim.zero_grad()
110     loss.backward()
140 111     optim.step()
112
113
114     # output
120 115     if ((epoch % nout == 0) or (loss.item() < tol)):
116         plt.close()
117         fig = plt.figure(dpi=300)
118         nt = 10
100 119         tt = torch.linspace(0., 1., nt+1)
120         nx = 20
121         xx = torch.linspace(-1., 1., nx+1)
122         T,X = torch.meshgrid(tt, xx, indexing='ij')
80 123         Uesp = torch.empty_like(T)
124         M = torch.empty(((nt+1)*(nx+1),2))
125         s = 0
126         for i,t in enumerate(tt):
60 127             for j,x in enumerate(xx):
128                 Uesp[i,j] = exp(-t)*sin(pi*x)
129                 M[s,0] = t
130                 M[s,1] = x
40

```

```

131         s += 1
132         Uest = model(M)
133         Uest = Uest.detach().reshape(nt+1,nx+1)
134         l2rel = torch.norm(Uest - Uesp)/torch.norm(Uesp)
135
136         ax = fig.add_subplot()
137         cb = ax.contourf(T, X, Uesp,
138                         levels=10)
139         fig.colorbar(cb)
140         cl = ax.contour(T, X, Uest,
141                        levels=10, colors='white')
142         ax.clabel(cl, fmt='%.1f')
143         ax.set_xlabel('$t$')
144         ax.set_ylabel('$x$')
145         plt.title(f'{epoch}: loss = {loss.item():.4e}, l2rel = {l2rel:.4e}')
146         plt.savefig(f'./results/sol_{{epoch//nout}:0>6}.png')
147
148     if ((loss.item() < tol) or (lr < 1e-6)):
149         break

```

### 4.3 PINN com Parâmetro a Determinar

Em construção

Vamos considerar uma **equação diferencial**

$$L(u; \lambda) = f, \quad \mathbf{x} \in D \subset \mathbb{R}^n, \quad (4.13)$$

onde  $L$  é um operador em funções  $u = u(\mathbf{x})$ ,  $\lambda \in \mathbb{R}$  é um **parâmetro a determinar** e  $f$  uma dada função fonte. Assumimos conhecidas condições inicial e de contorno, bem como um **conjunto de amostras**

$$\mathcal{D} := \left\{ (\mathbf{x}^{(s)}, u^{(s)}) \right\}_{s=1}^{n_s}, \quad (4.14)$$

com  $\mathbf{x}^{(s)} \in D$  e  $u^{(s)} = u(\mathbf{x}^{(s)})$ .

Uma rede informada pela física (**PINN**, do inglês, *Physics-informed neural network*) **com parâmetro a determinar** é uma rede neural

$$\tilde{u} = \mathcal{N}(\mathbf{x}; \lambda), \quad (4.15)$$

em que  $\tilde{u}$  é a solução estimada do modelo dado pela equação diferencial (4.13) com dadas condições inicial e de contorno, em que **o parâmetro  $\lambda$  é estimado tal que**

$$\tilde{u}^{(s)} \approx u^{(s)}, \quad (\mathbf{x}^{(s)}, u^{(s)}) \in \mathcal{D}. \quad (4.16)$$

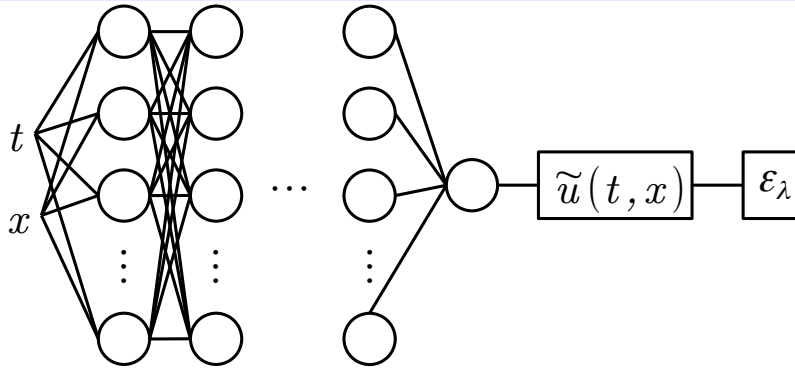


Figura 4.2: Esquema de uma PINN  $\tilde{u} = \mathcal{N}(\mathbf{x}; \lambda)$ .

Considerando uma rede do tipo perceptron multicamadas (MLP, do inglês, *multilayer perceptron*, consulte Fig. 4.2), seus pesos e *biases* são treinados em conjunto com parâmetro  $\lambda$  de forma a minimizar a **função de perda**

$$\begin{aligned} \epsilon_{\lambda} := & \underbrace{\frac{1}{n_{\text{in}}} \sum_{s=1}^{n_{\text{in}}} |\mathcal{R}_{\lambda}(\mathbf{x}_{\text{in}}^{(s)})|^2}_{\text{pts. internos}} \\ & + \underbrace{\frac{1}{n_{\text{cc}}} \sum_{s=1}^{n_{\text{cc}}} |\tilde{u}_{\text{cc}} - u_{\text{cc}}|^2}_{\text{c.i. \& c.c.}} \\ & + \underbrace{\frac{p}{n_s} \sum_{s=1}^{n_s} |\tilde{u}^{(s)} - u^{(s)}|^2}_{\text{amostras}}, \end{aligned} \quad (4.17)$$

onde  $p \geq 0$  é uma **penalidade** e

$$\mathcal{R}_\lambda(\mathbf{x}) := f - L(u; \lambda) \quad (4.18)$$

é o **resíduo** de (4.13).

**Exemplo 4.3.1.** Consideramos a equação de Fisher<sup>2</sup>

$$u_t = u_{xx} + \lambda u(1 - u), \quad (t, x) \in (0, t_f) \times (0, 1), \quad (4.19)$$

com o parâmetro  $\lambda > 0$  a determinar. Assumimos dada condição inicial

$$u(0, x) = \frac{1}{\left(1 + e^{\sqrt{\frac{\lambda}{6}}x}\right)^2}, \quad x \in [0, 1], \quad (4.20)$$

e condições de contorno

$$u_x(t, 0) = \frac{1}{\left(1 + e^{-\frac{5}{6}\lambda t}\right)^2}, \quad (4.21)$$

$$u_x(t, 1) = \frac{1}{\left(1 + e^{\sqrt{\frac{\lambda}{6}} - \frac{5}{6}\lambda t}\right)^2}. \quad (4.22)$$

Este problema tem solução analítica [1]

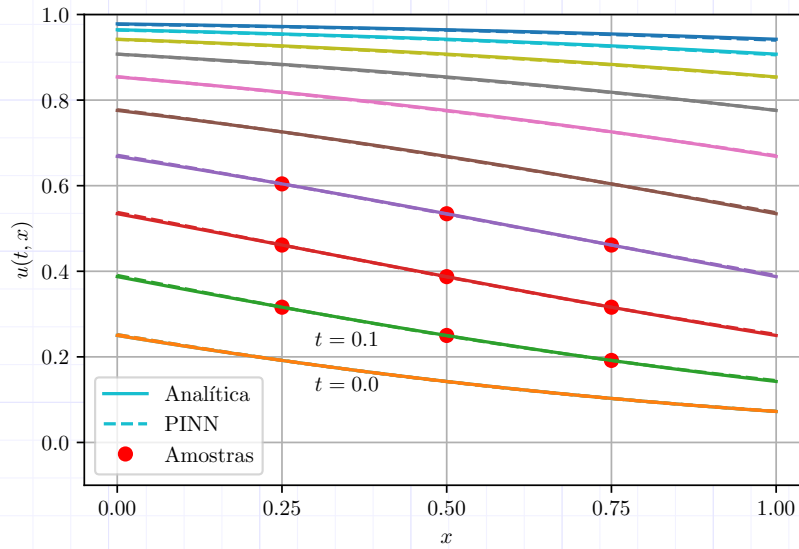
$$u_a(t, x) = \frac{1}{\left(1 + e^{\sqrt{\frac{\lambda}{6}}x - \frac{5}{6}\lambda t}\right)^2}. \quad (4.23)$$

Como exemplo de aplicação de uma PINN com parâmetro a determinar, vamos assumir o seguinte conjunto de amostras

$$\mathcal{D} = \left\{ \left( \left( t^{(s)}, x^{(s)} \right), u^{(s)} \right) \right\}_{s=1}^{n_s}, \quad (4.24)$$

com  $\left( t^{(s)}, x^{(s)} \right) \in \{0.1, 0.2, 0.3\} \times \{0.25, 0.5, 0.75\}$  e  $u^{(s)} = u_a\left(t^{(s)}, x^{(s)}\right)$ .



Figura 4.3: Solução PINN *versus* analítica para  $\lambda = 6$ .

Código 4.3: ex\_pinn\_fisher.py

```

1 import torch
2
3 # modelo
4 nh = 4
5 nn = 50
6 fun = torch.nn.Tanh()
7 model = torch.nn.Sequential()
8 model.add_module('layer_1', torch.nn.Linear(2, nn))
9 model.add_module('fun_1', fun)
10 for l in range(2, nh+1):
11     model.add_module(f'layer_{l}', torch.nn.Linear(nn, nn))
12     model.add_module(f'fun_{l}', fun)
13 model.add_module(f'layer_{nh+1}', torch.nn.Linear(nn, 1))
14
15 # parâmetro
16 rgn = [5., 7]
17 model.lambda = torch.nn.Parameter(
18     data=(rgn[1]-rgn[0])*torch.rand(1)+rgn[0])

```

```
19
20 # otimizador
21 optim = torch.optim.Adam(model.parameters(), lr=0.001)
22
23 # parâmetros do problema
24 tf = 1.
25
26 # solução analítica
27 lambda = torch.tensor([6.])
28 def ua(t,x, lambda=lambda):
29     return 1./(1.+torch.exp(torch.sqrt(lambda/6.)*x-5./6*lambda*t))
30
31 # condição inicial
32 def u0(x, lambda=lambda):
33     return 1./(1.+torch.exp(torch.sqrt(lambda/6)*x))**2
34
35 # amostras
36 ts = torch.tensor([0.1, 0.2, 0.3])
37 xs = torch.tensor([0.25, 0.5, 0.75])
38 T, X = torch.meshgrid(ts, xs, indexing='ij')
39 Ss = torch.hstack((T.reshape(-1,1), X.reshape(-1,1)))
40 Us_exp = ua(T, X).reshape(-1,1)
41
42 # treinamento
43 nepochs = 50000
44 tol = 1e-5
45
46 eout = 100
47
48 sin = 50
49 penalty = 1e1
50
51 for epoch in range(nepochs):
52
53     # forward
54
55     ## pts internos
56     tsin = tf*torch.rand(sin, 1)
```

```
57     xsin = torch.rand(sin, 1)
58     Sin = torch.hstack((tsin, xsin))
220 59     Sin.requires_grad = True
60
61     Uin = model(Sin)
62
200 63     ## loss pts internos
64     DUin = torch.autograd.grad(
65         Uin, Sin,
66         torch.ones_like(Uin),
180 67         create_graph=True,
68         retain_graph=True)[0]
69     Uin_t = DUin[:,0:1]
70     Uin_x = DUin[:,1:2]
160 71
72     Uin_xx = torch.autograd.grad(
73         Uin_x, Sin,
74         torch.ones_like(Uin_x),
140 75         create_graph=True,
76         retain_graph=True)[0][:,1:2]
77
78
120 79     lin = torch.mean((Uin_t - Uin_xx \
80                       - model.lmbda*Uin*(1-Uin))**2)
81
82     ## cond. inicial
100 83     S0 = torch.hstack((torch.zeros_like(xsin), xsin))
84
85     U0 = model(S0)
86
80 87     ## loss cond. inicial
88     l0 = torch.mean((U0 - u0(xsin))**2)
89
90     ## cond. de contorno
60 91     Sbc0 = torch.hstack((tsin, torch.zeros_like(xsin)))
92     Sbc1 = torch.hstack((tsin, torch.ones_like(xsin)))
93     Sbc = torch.vstack((Sbc0, Sbc1))
40 94
```

```

95     Ubc_exp = ua(Sbc[:,0:1],Sbc[:,1:2])
96     Ubc_est = model(Sbc)
97
98     ## loss cond. de contorno
99     lbc = torch.mean((Ubc_est - Ubc_exp)**2)
100
101     ## amostras
102     Us_est = model(Ss)
103
104     ## loss amostras
105     ls = torch.mean((Us_est - Us_exp)**2)
106
107     ## loss total
108     loss = lin + l0 + lbc + penalty*ls
109
110     if ((epoch % eout == 0) or (loss.item() < tol)):
111         print(f'epoch: {epoch}, '\
112               + f'loss={loss.item():.4e}, '\
113               + f'lmbda={model.lmbda.item():.3f}')
114
115     if (loss.item() < tol):
116         break
117
118     optim.zero_grad()
119     loss.backward()
120     optim.step()

```

### 4.3.1 Exercícios

Em construção

**Exemplo 4.3.2.** Considere o seguinte problema de valor inicial

$$-u'' = \lambda \sin(\pi x), \quad 0 < x < 1, \quad (4.25a)$$

$$u(0) = u(1) = 0, \quad (4.25b)$$

onde  $\lambda > 0$  é um parâmetro a determinar. Dadas as amostras

$$\mathcal{D} = \left\{ \left( \frac{1}{6}, \frac{1}{2} \right), \left( \frac{1}{4}, \sqrt{22} \right), \left( \frac{1}{3}, \sqrt{33} \right) \right\}, \quad (4.26)$$

crie uma PINN

$$\tilde{u} = \mathcal{N}(x; \lambda) \quad (4.27)$$

para estimar o parâmetro  $\lambda$  e a solução em todo o domínio  $0 \leq x \leq 1$ .

**Exemplo 4.3.3.** Considere o problema de Poisson<sup>3</sup>

$$-\nabla u = \lambda, \quad (x, y) \in D = (-1, 1)^2, \quad (4.28a)$$

$$u = 0, \quad (x, y) \in \partial D, \quad (4.28b)$$

onde  $\lambda > 0$  é um parâmetro a determinar. Dado que  $u(1/2, 1/2) = 1/8$ , crie uma PINN

$$\tilde{u} = \mathcal{N}(x, y; \lambda) \quad (4.29)$$

para estimar o parâmetro  $\lambda$  e a solução em todo o domínio  $D$ .

**Exemplo 4.3.4.** Considere o problema de calor

$$u_t = \lambda u_{xx} + (\pi^2 - 1)e^{-t} \sin(\pi x), \quad (t, x) \in (0, 1)^2, \quad (4.30a)$$

$$u(0, x) = \sin(\pi x), \quad x \in [0, 1], \quad (4.30b)$$

$$u(t, 0) = u(t, 1) = 0, \quad t \in [0, 1], \quad (4.30c)$$

onde o coeficiente de difusão  $\lambda > 0$  é um parâmetro a determinar. Sabendo que o problema tem solução analítica

$$u(t, x) = e^{-t} \sin(\pi x), \quad (4.31)$$

escolha um conjunto de amostras  $\mathcal{D} = \left\{ \left( t^{(s)}, x^{(s)} \right), u^{(s)} \right\}_{s=1}^{n_s}$  tal que seja possível estimar  $\lambda$  com uma PINN

$$\tilde{u} = \mathcal{N}(t, x; \lambda). \quad (4.32)$$

# Resposta dos Exercícios

**E.2.1.3.** Dica: verifique que sua matriz hessiana é positiva definida.

**E.2.1.4.** Dica: consulte a ligação [Notas de Aula: Matemática Numérica: 7.1 Problemas lineares](#).

**E.2.2.1.**  $(\tanh x)' = 1 - \tanh^2 x$

**E.4.1.1.** Dica: solução analítica  $u(x_1, x_2) = x_1(1 - x_1) - x_2(1 - x_2)$ .

**E.4.3.0.**  $\lambda = \pi^2$

**E.4.3.0.**  $\lambda = 1$

**E.4.3.0.**  $\lambda = 1$

## Bibliografia

- [1] Ağirseven, D., Öziş, T.. *An analytical study for Fisher type equations by using homotopy perturbation method*, Computers and Mathematics with Applications, vol. 60, p. 602-609, 2010. DOI: [10.1016/j.camwa.2010.05.006](https://doi.org/10.1016/j.camwa.2010.05.006)
- [2] Goodfellow, I., Bengio, Y., Courville, A.. Deep learning, MIT Press, Cambridge, MA, 2016.
- [3] Neural Networks: A Comprehensive Foundation, Haykin, S.. Pearson:Delhi, 2005. ISBN: 978-0020327615.
- [4] Raissi, M., Perdikaris, P., Karniadakis, G.E.. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. Journal of Computational Physics 378 (2019), pp. 686-707. DOI: [10.1016/j.jcp.2018.10.045](https://doi.org/10.1016/j.jcp.2018.10.045).
- [5] Mata, F.F., Gijón, A., Molina-Solana, M., Gómez-Romero, J.. Physics-informed neural networks for data-driven simulation: Advantages, limitations, and opportunities. Physica A: Statistical Mechanics and its Applications 610 (2023), pp. 128415. DOI: [10.1016/j.physa.2022.128415](https://doi.org/10.1016/j.physa.2022.128415).