

# Redes Neurais Artificiais

Pedro H A Konzen

13 de novembro de 2023

# 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

Nestas notas de aula são abordados tópicos introdutórios sobre redes neurais artificiais. Como ferramenta computacional de apoio, vários exemplos de aplicação de códigos `Python+PyTorch` são apresentados.

Agradeço a todas e todos que de modo assíduo ou esporádico contribuem com correções, sugestões e críticas. :)

Pedro H A Konzen

# Conteúdo

<b>Capa</b>	<b>i</b>
<b>Licença</b>	<b>ii</b>
<b>Prefácio</b>	<b>iii</b>
<b>Sumário</b>	<b>v</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 . . . . .	10
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 . . . . .	19
2.2.3 Exercícios . . . . .	22
<b>3 Perceptron Multicamadas</b>	<b>23</b>
3.1 Modelo MLP . . . . .	23
3.1.1 Treinamento . . . . .	24
3.1.2 Aplicação: Problema de Classificação XOR . . . . .	26
3.1.3 Exercícios . . . . .	28
3.2 Aplicação: Problema de Classificação Binária . . . . .	28
3.2.1 Dados . . . . .	29
3.2.2 Modelo . . . . .	30

3.2.3	Treinamento e Teste . . . . .	31
3.2.4	Verificação . . . . .	33
3.2.5	Exercícios . . . . .	34
3.3	Aplicação: Aproximação de Funções . . . . .	34
3.3.1	Função unidimensional . . . . .	34
3.3.2	Função bidimensional . . . . .	36
3.3.3	Exercícios . . . . .	39
3.4	Diferenciação Automática . . . . .	39
3.4.1	Autograd Perceptron . . . . .	42
3.4.2	Autograd MLP . . . . .	45
3.4.3	Exercícios . . . . .	48
<b>4</b>	<b>Redes Informadas pela Física</b>	<b>49</b>
4.1	Problemas de Valores Iniciais . . . . .	49
4.1.1	Euler PINN . . . . .	49
4.1.2	AD-PINN . . . . .	53
4.1.3	Exercícios . . . . .	56
4.2	Aplicação: Equação de Laplace . . . . .	56
4.2.1	Preprocessamento . . . . .	61
4.2.2	Exercícios . . . . .	66
4.3	Aplicação: Equação do Calor . . . . .	66
4.3.1	Diferenças Finitas . . . . .	66
4.3.2	Diferenciação Automática . . . . .	70
	<b>Respostas dos Exercícios</b>	<b>75</b>
	<b>Bibliografia</b>	<b>76</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*).

tron), 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

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

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



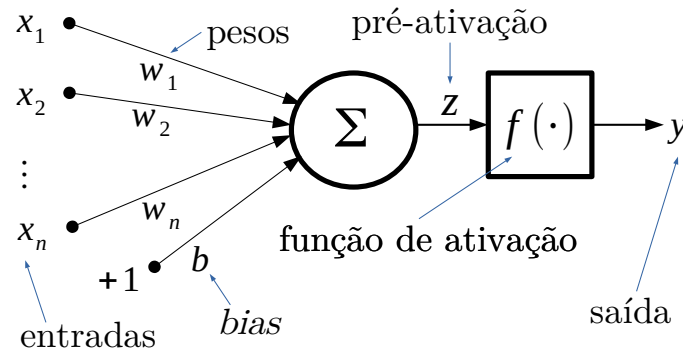


Figura 2.1: Esquema de um perceptron: unidade de processamento.

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
650 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.

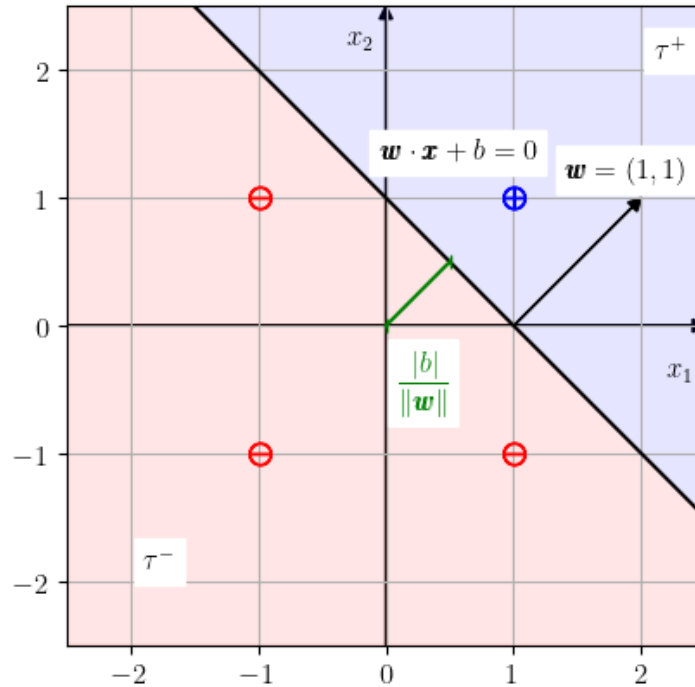


Figura 2.2: Interpretação geométrica do perceptron aplicado ao problema de classificação relacionado à operação lógica  $\wedge$  (e-lógico).

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

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

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

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.]])
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)

```

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

```
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 # verificação
58 print(f'W =\n{W}')
59 print(f'b =\n{b}')
60 y = model(X_train)
61 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

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

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>

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

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

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

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

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



## 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 =")
33 print(X_train)
34 print("y_train = ")
35 print(y_train)
36
37 # treinamento
```

```
38
650 39 ## matriz
40 M = torch.hstack((X_train,
41                    torch.ones((ns,1))))
42 ## solução M.Q.
600 43 c = torch.linalg.lstsq(M, y_train)[0]
44 with torch.no_grad():
45     W = c[0]
550 46     b = c[1]
47
48 # verificação
49 print(f'W =\n{W}')
500 50 print(f'b =\n{b}')
51 y = model(X_train)
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.

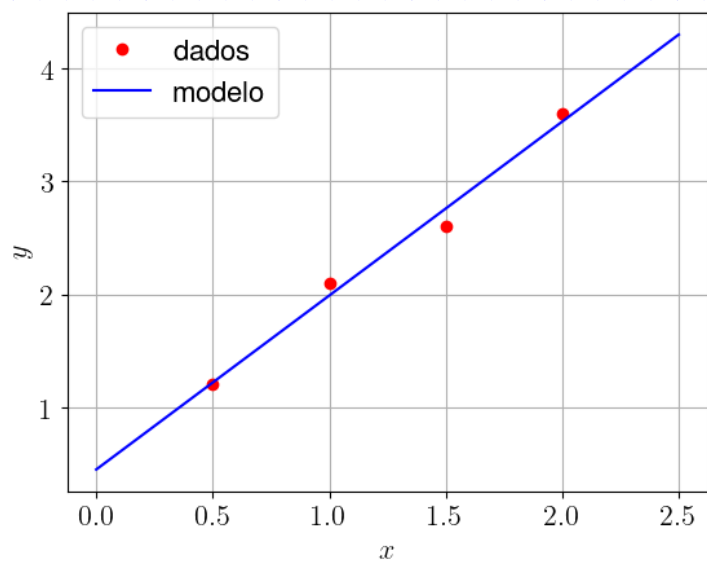


Figura 2.3: Interpretação geométrica do perceptron aplicado ao problema de regressão linear.

### 2.1.3 Exercícios

**Exercício 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

**Exercício 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.

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

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

**Exercício 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.

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

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

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

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

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

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

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

### 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

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

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

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
26 X_train = torch.tensor([[1., 1.],
27                          [1., -1.],
28                          [-1., 1.],
29                          [-1., -1.]])
30 y_train = torch.tensor([1., -1., -1., -1.]).reshape(-1,1)
31
32 print("\nDados de treinamento")
33 print("X_train =")
34 print(X_train)
35 print("y_train = ")
36 print(y_train)
37
38 ## num max épocas
39 nepochs = 1000
```

```
40 tol = 1e-3
41
42 for epoch in range(nepochs):
43
44     # forward
45     y_est = model(X_train)
46
47     # erro
48     loss = loss_fun(y_est, y_train)
49
50     print(f'{epoch}: {loss.item():.4e}')
51
52     # critério de parada
53     if (loss.item() < tol):
54         break
55
56     # backward
57     optim.zero_grad()
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 iniciais.
2. Para  $e \leftarrow 1, \dots, n_e$ :

---

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



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
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
21 optim = torch.optim.SGD(model.parameters(), lr=5e-1)
22
23 ## função erro
24 loss_fun = torch.nn.MSELoss()
25
26 ## dados de treinamento
27 X_train = torch.tensor([[1., 1.],
28                          [1., -1.],
29                          [-1., 1.],
30                          [-1., -1.]])
31 y_train = torch.tensor([1., -1., -1., -1.]).reshape(-1,1)
32
```

```
33  ## num de amostras
34  ns = y_train.size(0)
35
36  print("\nDados de treinamento")
37  print("X_train =")
38  print(X_train)
39  print("y_train = ")
40  print(y_train)
41
42  ## num max épocas
43  nepochs = 5000
44  tol = 1e-3
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

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

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

**Exercício 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.

**Exercício 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.

**Exercício 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

[[tag:construcao]]

### 3.1 Modelo MLP

[[tag:construcao]]

Uma Perceptron Multicamadas (MLP, do inglês, *Multilayer Perceptron*) é um tipo de Rede Neural Artificial formada por composições de camadas de perceptrons. Consulte a Figura 3.1.

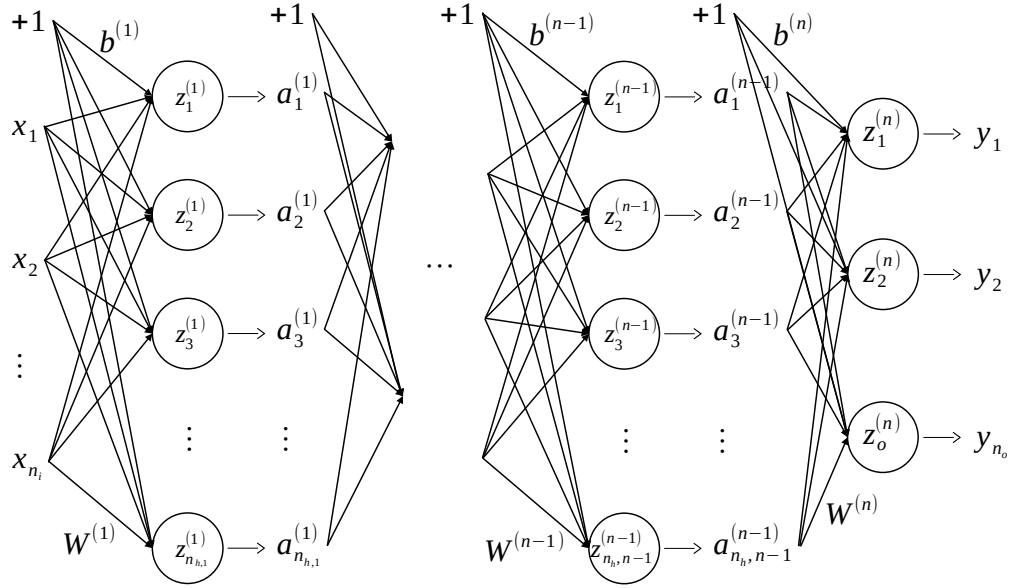


Figura 3.1: Estrutura de uma rede do tipo Perceptron Multicamadas (MLP).

Denotamos uma MLP de  $n$  camadas por

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

onde  $(W^{(l)}, \mathbf{b}^{(l)}, f^{(l)})$  é a tripa de **pesos**, **biases** e **função de ativação** da  $l$ -ésima camada da rede,  $l = 1, 2, \dots, n$ .

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  $l = 1, 2, \dots, n$ , denotando  $\mathbf{a}^{(0)} := \mathbf{x}$  e  $\mathbf{a}^{(n)} := \mathbf{y}$ .

### 3.1.1 Treinamento

[[tag:construcao]]

Fornecido um **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

$$\min_{(\mathbf{W}, \mathbf{b})} \varepsilon(\tilde{\mathbf{y}}^{(s)}, \mathbf{y}^{(s)}) \quad (3.3)$$

onde  $\varepsilon$  é uma dada **função erro** (em inglês, *loss function*) e  $\tilde{\mathbf{y}}^{(s)}, \mathbf{y}^{(s)}$  são as saídas estimada e esperada da  $l$ -ésima amostra, respectivamente.

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

1.  $\mathbf{W}, \mathbf{b}$  aproximações iniciais.
2. Para  $e \leftarrow 1, \dots, n_e$ :

$$(a) \quad (\mathbf{W}, \mathbf{b}) \leftarrow (\mathbf{W}, \mathbf{b}) - l_r \mathbf{d}(\nabla_{\mathbf{W}, \mathbf{b}} \varepsilon)$$

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

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

O cálculo dos gradientes pode ser feito **de trás para frente** (em inglês, *backward*), i.e. para os pesos da última camada, temos

$$\frac{\partial \varepsilon}{\partial \mathbf{W}^{(n)}} = \frac{\partial \varepsilon}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{z}^{(n)}} \frac{\partial \mathbf{z}^{(n)}}{\partial \mathbf{W}^{(n)}}, \quad (3.5)$$

$$= \frac{\partial \varepsilon}{\partial \mathbf{y}} f' \left( \mathbf{W}^{(n)} \mathbf{a}^{(n-1)} + \mathbf{b}^{(n)} \right) \mathbf{a}^{(n-1)}. \quad (3.6)$$

Para os pesos da penúltima, temos

$$\frac{\partial \varepsilon}{\partial \mathbf{W}^{(n-1)}} = \frac{\partial \varepsilon}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{z}^{(n)}} \frac{\partial \mathbf{z}^{(n)}}{\partial \mathbf{W}^{(n-1)}}, \quad (3.7)$$

$$= \frac{\partial \varepsilon}{\partial \mathbf{y}} f' \left( \mathbf{z}^{(n)} \right) \frac{\partial \mathbf{z}^{(n)}}{\partial \mathbf{a}^{(n-1)}} \frac{\partial \mathbf{a}^{(n-1)}}{\partial \mathbf{z}^{(n-1)}} \frac{\partial \mathbf{z}^{(n-1)}}{\partial \mathbf{W}^{(n-1)}} \quad (3.8)$$

$$= \frac{\partial \varepsilon}{\partial \mathbf{y}} f' \left( \mathbf{z}^{(n)} \right) \mathbf{W}^{(n)} f' \left( \mathbf{z}^{(n-1)} \right) \mathbf{a}^{(n-2)} \quad (3.9)$$

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

### 3.1.2 Aplicação: Problema de Classificação XOR

[[tag:construcao]]

Vamos desenvolver uma MLP que faça a operação **xor** (ou exclusivo). 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  $R = A_1 \text{ xor } A_2$ . Consultamos a seguinte 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

[[tag:construcao]]

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

[[tag:construcao]]

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

$$\varepsilon := \frac{1}{n_s} \sum_{s=1}^{n_s} |\tilde{y}^{(s)} - y^{(s)}|^2, \quad (3.10)$$

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

### Implementação

[[tag:construcao]]

O seguinte código implementa a MLP e usa o Método do Gradiente Descendente (DG) no algoritmo de treinamento.

Código 3.1: mlp\_xor.py

```
1 import torch
2
3 # modelo
4
5 model = torch.nn.Sequential(
6     torch.nn.Linear(2,2),
7     torch.nn.Tanh(),
8     torch.nn.Linear(2,1)
9 )
10
11 # treinamento
12
13 ## otimizador
14 optim = torch.optim.SGD(model.parameters(), lr=1e-2)
15
16 ## função erro
17 loss_fun = torch.nn.MSELoss()
18
19 ## dados de treinamento
20 X_train = torch.tensor([[1., 1.],
21                          [1., -1.],
22                          [-1., 1.],
23                          [-1., -1.]])
24 y_train = torch.tensor([-1., 1., 1., -1.]).reshape(-1,1)
25
26 print("\nDados de treinamento")
27 print("X_train =")
28 print(X_train)
29 print("y_train = ")
```



```
30 print(y_train)
31
32 ## num max épocas
33 nepochs = 5000
34 tol = 1e-3
35
36 for epoch in range(nepochs):
37
38     # forward
39     y_est = model(X_train)
40
41     # erro
42     loss = loss_fun(y_est, y_train)
43
44     print(f'{epoch}: {loss.item():.4e}')
45
46     # critério de parada
47     if (loss.item() < tol):
48         break
49
50     # backward
51     optim.zero_grad()
52     loss.backward()
53     optim.step()
54
55
56 # verificação
57 y = model(X_train)
58 print(f'y_est = {y}')
```

### 3.1.3 Exercícios

[[tag::construcao]]

## 3.2 Aplicação: Problema de Classificação Binária

[[tag:construcao]]

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

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

### 3.2.1 Dados

[[tag:construcao]]

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()
```

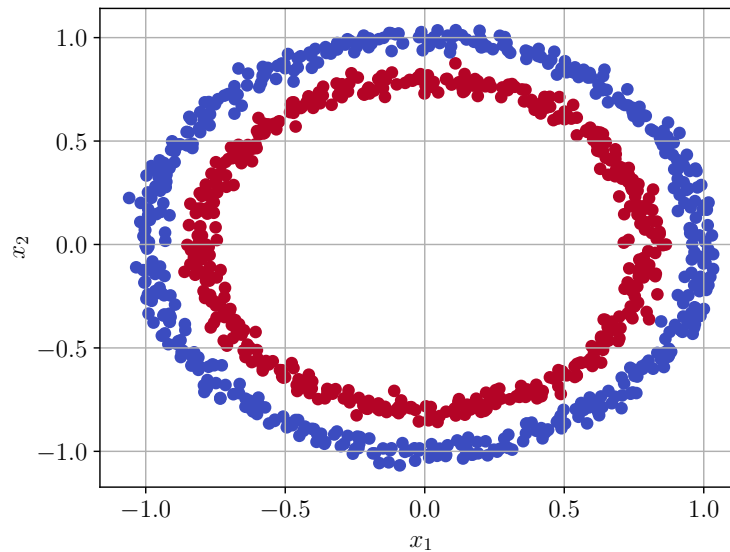


Figura 3.2: Dados para a o problema de classificação binária não linear.

### 3.2.2 Modelo

[[tag:construcao]]

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

$$\text{elu}(x) = \begin{cases} x & , x > 0 \\ \alpha(e^x - 1) & , x \leq 0 \end{cases} \quad (3.11)$$

na camada escondida e

$$\text{sigmoid}(x) = \frac{1}{1 + e^x} \quad (3.12)$$

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

[[tag:construcao]]

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=42)
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
64     loss = loss_fun(y_pred, y_train)
65
66     acc = accuracy_fun(torch.round(y_pred),
67                         y_train)
68
69     optimizer.zero_grad()
70     loss.backward()
71     optimizer.step()
72
73     model.eval()
74
75     #testing
```

```

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,
80                                 y_test)
81             acc_test = accuracy_fun(torch.round(y_pred_test),
82                                     y_test)
83
84     print(f'{epoch+1}: loss = {loss:.5e}, accuracy = {acc:.2f}%')
85     print(f'\tttest: loss = {loss:.5e}, accuracy = {acc:.2f}%\n')

```

### 3.2.4 Verificação

[[tag:construcao]]

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.

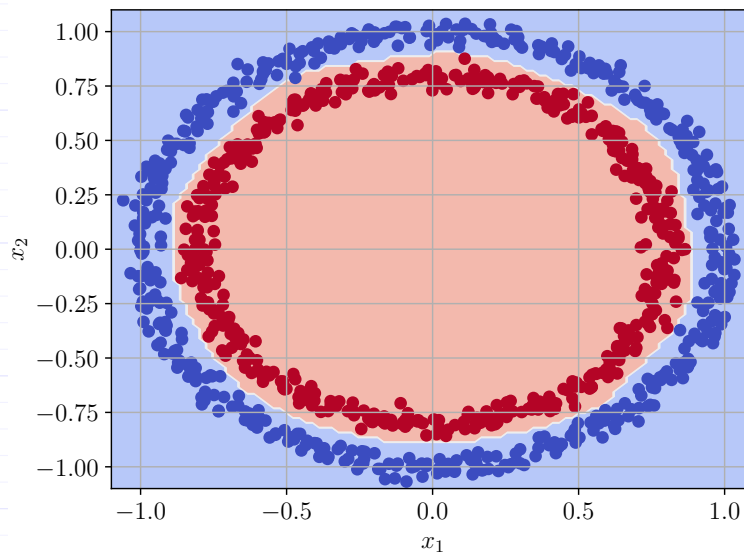


Figura 3.3: Verificação do modelo de classificação binária.

1 # malha de pontos

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

```

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

[[tag:construcao]]

## 3.3 Aplicação: Aproximação de Funções

[[tag:construcao]]

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

### 3.3.1 Função unidimensional

[[tag:construcao]]

Vamos criar uma MLP para aproximar a função gaussiana

$$y = e^{-x^2}, \quad (3.13)$$

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

```

1 import torch
2 import matplotlib.pyplot as plt

```

```
3
4 # modelo
5
6 model = torch.nn.Sequential(
7     torch.nn.Linear(1,25),
8     torch.nn.Tanh(),
9     torch.nn.Linear(25,1)
10 )
11
12 # treinamento
13
14 ## fun obj
15 fobj = lambda x: torch.exp(-x**2)
16 a = -1.
17 b = 1.
18
19 ## otimizador
20 optim = torch.optim.SGD(model.parameters(),
21                           lr=1e-2, momentum=0.9)
22
23 ## função erro
24 loss_fun = torch.nn.MSELoss()
25
26 ## num de amostras por época
27 ns = 100
28 ## num max épocas
29 nepochs = 5000
30 ## tolerância
31 tol = 1e-5
32
33 for epoch in range(nepochs):
34
35     # amostras
36     X_train = (a - b) * torch.rand((ns,1)) + b
37     y_train = fobj(X_train)
38
39     # forward
40     y_est = model(X_train)
41
42     # erro
```



```

43     loss = loss_fun(y_est, y_train)
44
45     print(f'{epoch}: {loss.item():.4e}')
46
47     # critério de parada
48     if (loss.item() < tol):
49         break
50
51     # backward
52     optim.zero_grad()
53     loss.backward()
54     optim.step()
55
56
57 # verificação
58 fig = plt.figure()
59 ax = fig.add_subplot()
60
61 x = torch.linspace(a, b,
62                    steps=50).reshape(-1,1)
63
64 y_esp = fobj(x)
65 ax.plot(x, y_esp, label='fobj')
66
67 y_est = model(x)
68 ax.plot(x, y_est.detach(), label='model')
69
300 70 ax.legend()
71 ax.grid()
72 ax.set_xlabel('x')
73 ax.set_ylabel('y')
74 plt.show()

```

### 3.3.2 Função bidimensional

[[tag:construcao]]

Vamos criar uma MLP para aproximar a função gaussiana

$$y = e^{-(x_1^2 + x_2^2)}, \quad (3.14)$$

para  $\mathbf{x} = (x_1, x_2) \in [-1, 1]^2$ .

```
1 import torch
2 import matplotlib.pyplot as plt
3
4 # modelo
5
6 model = torch.nn.Sequential(
7     torch.nn.Linear(2,50),
8     torch.nn.Tanh(),
9     torch.nn.Linear(50,25),
10    torch.nn.Tanh(),
11    torch.nn.Linear(25,5),
12    torch.nn.Tanh(),
13    torch.nn.Linear(5,1)
14 )
15
16 # treinamento
17
18 ## fun obj
19 a = -1.
20 b = 1.
21 def fobj(x):
22     y = torch.exp(-x[:,0]**2 - x[:,1]**2)
23     return y.reshape(-1,1)
24
25 ## otimizador
26 optim = torch.optim.SGD(model.parameters(),
27                           lr=1e-1, momentum=0.9)
28
29 ## função erro
30 loss_fun = torch.nn.MSELoss()
31
32 ## num de amostras por eixo por época
33 ns = 100
34 ## num max épocas
35 nepochs = 5000
36 ## tolerância
37 tol = 1e-5
38
39 for epoch in range(nepochs):
```

```
40
41     # amostras
42     x0 = (a - b) * torch.rand(ns) + b
43     x1 = (a - b) * torch.rand(ns) + b
44     X0, X1 = torch.meshgrid(x0, x1)
45     X_train = torch.cat((X0.reshape(-1,1),
46                           X1.reshape(-1,1)),
47                           dim=1)
48     y_train = fobj(X_train)
49
50     # forward
51     y_est = model(X_train)
52
53     # erro
54     loss = loss_fun(y_est, y_train)
55
450    print(f'{epoch}: {loss.item():.4e}')
57
58     # critério de parada
300    if (loss.item() < tol):
60         break
61
62     # backward
350    optim.zero_grad()
64    loss.backward()
65    optim.step()
66
67
68     # verificação
250    fig = plt.figure()
70    ax = fig.add_subplot()
71
72    n = 50
73    x0 = torch.linspace(a, b, steps=n)
200    x1 = torch.linspace(a, b, steps=n)
75    X0, X1 = torch.meshgrid(x0, x1)
76    X = torch.cat((X0.reshape(-1,1),
77                  X1.reshape(-1,1)),
150                  dim=1)
78
79
```

```

80 y_esp = fobj(X)
81 Y = y_esp.reshape((n,n))
82 levels = torch.linspace(0., 1., 10)
83 c = ax.contour(X0, X1, Y, levels=levels, colors='white')
84 ax.clabel(c)
85
86 y_est = model(X)
87 Y = y_est.reshape((n,n))
88 ax.contourf(X0, X1, Y.detach(), levels=levels)
89
90 ax.grid()
91 ax.set_xlabel('x_1')
92 ax.set_ylabel('x_2')
93 plt.show()

```

### 3.3.3 Exercícios

[[tag::construcao]]

## 3.4 Diferenciação Automática

[[tag:construcao]]

**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 a partir de seu grafo computacional. Os gradientes são computados por retropropagação. Por exemplo, para a computação do gradiente

$$\left. \frac{df}{dx} \right|_{x=x_0}, \quad (3.15)$$

primeiramente, propaga-se a entrada  $x_0$  pela função computacional  $f$ , obtendo-se  $y = f(x_0)$ . Então, o gradiente é computado por retropropagação.

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

$$\left. \frac{df}{dx} \right|_{x=0} \quad (3.16)$$

por diferenciação automática.

Pela regra da cadeia

$$\frac{df}{dx} = \text{sen}'(\pi x) \cdot [\pi x]' \quad (3.17)$$

$$= \cos(\pi x) \cdot \pi \quad (3.18)$$

$$= \pi \cos(\pi x) \quad (3.19)$$

Primeiramente, observamos que a computação de  $f(x)$  pode ser representada pelo grafo de propagação mostrado na Figura 3.4. Para a computação do gradiente, adicionamos uma variável fictícia  $z = y$ . Na retropropagação, computamos

1.

$$\frac{dz}{dy} = 1 \quad (3.20)$$

2.

$$\frac{dz}{du} = \frac{d}{du} [\text{sen}(u)] \frac{dz}{dy} \quad (3.21)$$

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

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

3.

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

$$= \pi \cos(\pi x). \quad (3.25)$$

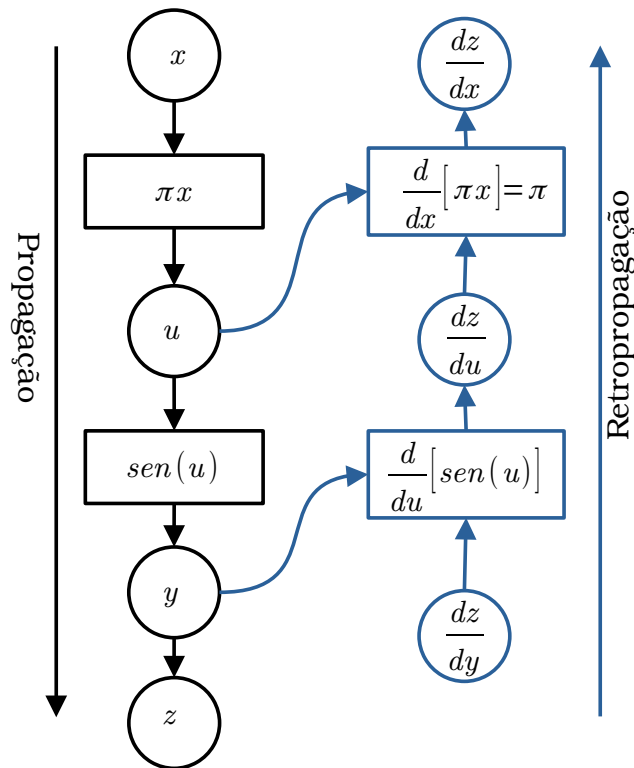


Figura 3.4: Grafo computacional para a diferenciação automática.

Uma RNA é uma composição de funções definidas por parâmetros (pesos e *biases*). O treinamento de uma RNA ocorre em duas etapas<sup>1</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>2</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.

A **Diferenciação Automática (Autograd, do inglês, *Automatic Gradient*)**

<sup>1</sup>Para mais detalhes, consulte a Subseção 3.1.1.

<sup>2</sup>Medida da diferença entre o valor estimado e o valor esperado.

consiste na computação de derivadas a partir da regra da cadeia em uma estrutura computacional composta de funções elementares. Esse é o caso em RNAs, a computação do gradiente da saída da rede em relação a sua entrada pode ser feita de forma similar à computação do gradiente do erro em relação aos seus parâmetros.

### 3.4.1 Autograd Perceptron

[[tag:construcao]]

Para um Perceptron<sup>3</sup>

$$\tilde{y} = \mathcal{N}(\mathbf{x}, (\mathbf{w}, b)) \quad (3.26a)$$

$$= f(\underbrace{\mathbf{w} \cdot \mathbf{x} + b}_z) \quad (3.26b)$$

temos que o gradiente da saída  $y$  em relação à entrada  $\mathbf{x}$  pode ser computada como segue

$$\frac{\partial \tilde{y}}{\partial \mathbf{x}} = \frac{\partial f}{\partial z} \frac{\partial z}{\partial \mathbf{x}} \quad (3.27a)$$

$$= f'(z) \mathbf{w} \quad (3.27b)$$

**Exemplo 3.4.2.** Vamos treinar um Perceptron com função de ativação  $f(z) = z$

$$\tilde{y} = \mathcal{N}(x; (w, b)) \quad (3.28a)$$

$$= wx + b \quad (3.28b)$$

que se ajusta ao conjunto de pontos<sup>4</sup>

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

<sup>3</sup>Consulte o Capítulo 2 para mais informações sobre o Perceptron.

<sup>4</sup>Consulte o Exercício 2.2.4.

Uma vez treinado com função erro MSE<sup>5</sup>, espera-se que o Perceptron corresponda a reta de mínimos quadrados<sup>6</sup>

$$y = 1.54x + 0.45 \quad (3.29)$$

Portanto, espera-se que

$$\frac{\partial \tilde{y}}{\partial x} = 1.54. \quad (3.30)$$

Código 3.3: autograd\_percep.py

```
1 import torch
2
3 # modelo
4 model = torch.nn.Linear(1,1)
5
6 # treinamento
7
8 ## otimizador
9 optim = torch.optim.SGD(model.parameters(),
10                           lr=1e-1)
11
12 ## função erro
13 loss_fun = torch.nn.MSELoss()
14
15 ## dados de treinamento
16 X_train = torch.tensor([[0.5],
17                          [1.0],
18                          [1.5],
19                          [2.0]])
20 y_train = torch.tensor([[1.2],
21                          [2.1],
22                          [2.6],
23                          [3.6]])
24
25 ## num max épocas
26 nepochs = 5000
```

<sup>5</sup>MSE, Erro Quadrático Médio.

<sup>6</sup>Para mais informações sobre essa aplicação, consulte a Subseção 2.1.2.



```
27 nstop = 10
28
29 cstop = 0
30 loss_min = torch.finfo().max
31 for epoch in range(nepochs):
32
33     # forward
34     y_est = model(X_train)
35
36     # erro
37     loss = loss_fun(y_est, y_train)
38
39     # critério de parada
40     if (loss.item() >= loss_min):
41         cstop += 1
42     else:
43         loss_min = loss.item()
44         cstop = 0
45
46     print(f'{epoch}: {loss.item():.4e}, '\
47           + f'cstop = {cstop}/{nstop}')
48
49     if (cstop == nstop):
50         break
51
52     # backward
53     optim.zero_grad()
54     loss.backward()
55     optim.step()
56
57
58     # verificação
59     print(f'w = {model.weight}')
60     print(f'b = {model.bias}')
61
62     # autograd dy/dx
63
64     ## forward
65     x = torch.tensor([[1.]],
66                       requires_grad=True)
```

```

67 y = model(x)
68
69 ## backward
70 y.backward()
71 dydx = x.grad
72 print(f'dy/dx = {dydx}')
```

### 3.4.2 Autograd MLP

[[tag:construcao]]

Os conceitos de diferenciação automática (**autograd**) são diretamente estendidos para redes do tipo Perceptron Multicamadas (MLP, do inglês, *Multilayer Perceptron*). 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

$$\tilde{y} = \mathcal{N}\left(x; \left(W^{(l)}, \mathbf{b}^{(l)}, f^{(l)}\right)_{l=1}^n\right), \quad (3.31)$$

que aproxima a função  $y = \text{sen}(\pi x)$  para  $x \in [-1, 1]$

Código 3.4: autograd\_fun1d.py

```

1 import torch
2 import matplotlib.pyplot as plt
3
4 # modelo
5
6 model = torch.nn.Sequential(
7     torch.nn.Linear(1,50),
8     torch.nn.Tanh(),
9     torch.nn.Linear(50,25),
10    torch.nn.Tanh(),
11    torch.nn.Linear(25,1)
12 )
13
14 # treinamento
15
16 ## fun obj
```

```
17 fobj = 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 ## função erro
26 loss_fun = torch.nn.MSELoss()
27
28 ## num de amostras por época
29 ns = 100
30 ## num max épocas
31 nepochs = 10000
32 ## tolerância
33 tol = 1e-5
34
35 for epoch in range(nepochs):
36
37     # amostras
38     X_train = (a - b) * torch.rand((ns,1)) + b
39     y_train = fobj(X_train)
40
41     # forward
42     y_est = model(X_train)
43
44     # erro
45     loss = loss_fun(y_est, y_train)
46
47     lr = optim.param_groups[-1]['lr']
48     print(f'{epoch}: loss = {loss.item():.4e}, lr = {lr:.4e}')
49
50     # critério de parada
51     if ((loss.item() < tol) or (lr <= 1e-7)):
52         break
53
54     # backward
55     optim.zero_grad()
56     loss.backward()
```

```
57 optim.step()
```

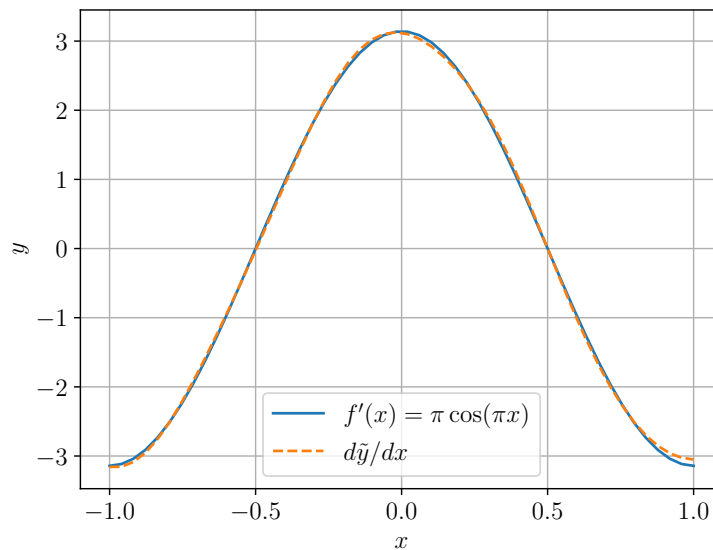


Figura 3.5: Comparação da autograd da MLP com a derivada exata  $f'(x) = \pi \cos(\pi x)$  para o Exemplo 3.4.3.

Uma vez treinada, nossa MLP é uma aproximadora da função seno, i.e.  $\tilde{y} \approx \sin(\pi x)$ . Usando de autograd podemos computar  $\tilde{y}' \approx \pi \cos(\pi x)$ . O código abaixo, computa  $d\tilde{y}/dx$  a partir da rede e produz o gráfico da figura acima.

```
1 # verificação
2 fig = plt.figure()
3 ax = fig.add_subplot()
4
5 xx = torch.linspace(a, b,
6                     steps=50).reshape(-1,1)
7 # y' = cos(x)
8 dy_esp = torch.pi*torch.cos(torch.pi*xx)
9 ax.plot(xx, dy_esp, label="$f'(x) = \pi \cos(\pi x)$")
10
11 # model autograd
```

```
12 dy_est = torch.empty_like(xx)
13 for i,x in enumerate(xx):
14     x.requires_grad = True
15     y = model(x)
16     y.backward()
17     dy_est[i] = x.grad
18 ax.plot(xx, dy_est, label='$d\\tilde{y}/dx$')
19
20 ax.legend()
21 ax.grid()
22 ax.set_xlabel('$x$')
23 ax.set_ylabel('$y$')
24 plt.show()
```

### 3.4.3 Exercícios

[[tag:construcao]]

## 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 Problemas de Valores Iniciais

[[tag:construcao]]

Consideramos um **problema de valor inicial** (**IVP**, do inglês, *initial value problem*)

$$y'(t) = g(t, y(t)), \quad t_0 < t < t_f, \quad (4.1a)$$

$$y(t_0) = y_0, \quad (4.1b)$$

com dada  $g : \mathbb{R}^2 \rightarrow \mathbb{R}$  e dados valor inicial  $y_0 \in \mathbb{R}$ , tempos inicial  $t_0 \in \mathbb{R}$  e final  $t_f \in \mathbb{R}$ .

#### 4.1.1 Euler PINN

[[tag:construcao]]

A solução do IVP (4.2) pode ser obtida por uma **rede neural informada pela física** (PINN, do inglês, *physics-informed neural network*) assumindo

a seguinte aproximação de Euler explícita

$$y^{(s+1)} = y^{(s)} + h_t g(t^{(s)}, y^{(s)}), \quad 0 \leq s \leq n_t - 1, \quad (4.2a)$$

$$y^{(0)} = y_0, \quad (4.2b)$$

onde  $y^{(s)} \approx y(t^{(s)})$ , nos tempos discretos  $t^{(s)} = t_0 + sh_t$ , com passo  $h_t = (t_t - t_0)/n_t$ ,  $s = 0, 1, 2, \dots, n_t$ .

Nosso modelo PINN é uma **perceptron multicamada** (MLP)

$$\tilde{y} = \mathcal{N}\left(t; \{W^{(l)}, \mathbf{b}^{(l)}, \mathbf{f}^{(l)}\}_{l=1}^{n_h+1}\right), \quad (4.3)$$

de dada arquitetura  $1 - n_n \times n_h - 1$ , i.e. uma entrada,  $n_h$  camadas escondidas, cada com  $n_n$  neurônios e uma saída. A entrada é o valor do tempo  $t$  e a saída é  $\tilde{y} = \mathcal{N}(t) \approx y(t)$ , a estimativa da solução do IVP (4.2). Escolhidas as funções de ativação  $\mathbf{f}^{(l)}$ ,  $l = 1, 2, \dots, n_h + 1$ , o treinamento da PINN consiste em resolver o seguinte problema de minimização

$$\min_{\{W^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{n_h+1}} \frac{1}{n_s} \sum_{s=0}^{n_s-1} |\mathcal{R}^{(s)}|^2 + p_p |\tilde{y}^{(0)} - y_0|^2, \quad (4.4)$$

onde  $p_p > 0$  é um dado **parâmetro de penalização** e  $\mathcal{R}^{(s)}$  é o **resíduo**

$$\mathcal{R}^{(s)} := \frac{y^{(s+1)} - y^{(s)}}{h_t} - h_t g(t^{(s)}, y^{(s)}). \quad (4.5)$$

**Exemplo 4.1.1.** Consideramos o seguinte IVP

$$y'(t) = \sin(t) - y, \quad 0 < t < 1, \quad (4.6a)$$

$$y(0) = \frac{1}{2}. \quad (4.6b)$$

Código 4.1: pyEulerPINN.py

```
1 import torch
2 from scipy.integrate import quad
3
4 # model
5 ## num hidden layers
```

```
6  nh = 2
7  ## num neurons per hidden layer
8  nn = 50
9  ## activation fun in hidden layers
10 fh = torch.nn.Tanh()
11 ## model architecture
12 model = torch.nn.Sequential()
13 model.add_module('layer_1', torch.nn.Linear(1,nn))
14 model.add_module('fun_1', fh)
15 for l in range(2, nh):
16     model.add_module(f'layer_{l}', torch.nn.Linear(nn,nn))
17     model.add_module(f'fun_{l}', fh)
18 model.add_module(f'layer_{nh}', torch.nn.Linear(nn,1))
19
20 # IVP params
21
22 ## init time
23 t0 = 0.
24 ## init condition
25 y0 = 0.5
26 ## final time
27 tf = 1.
28
29 ## num of time samples
30 ns = 10
31 ## time step
32 ht = (tf - t0)/ns
33 ## time samples
34 ts = torch.linspace(t0, tf, ns+1).reshape(-1,1)
35
36 ## rhs
37 def g(t, y):
38     return y + torch.sin(t)
39
40 # training
41 ## num of epochs
42 nepochs = 10000
43 ## output loss freq
44 eout = 100
45 ## tolerance
```



```
46 tol = 1e-4
47 ## early-stop
48 n_iter_no_change = 100
49
50 ## optimizer
51 optim = torch.optim.Adam(model.parameters(), lr=1e-3)
52
53 # training loop
54 count_no_change = 0
55 best_loss = 1e38
56 for epoch in range(nepochs):
57
58     # forward
59     yy = model(ts)
60
61     # loss
62     ## t>0
63     lup = torch.mean(((yy[1:] - yy[:-1])/ht \
64                       - g(ts[:-1], yy[:-1]))**2)
65     ## t = 0
66     lic = (yy[0] - y0)**2
67
68     loss_train = lup + lic
69
70     # backward
71     optim.zero_grad()
72     loss_train.backward()
73     optim.step()
74
75     # validation
76     ys = model(ts).detach()
77     yv = torch.empty_like(ys)
78     yv[0] = y0
79     for s in range(1,ns+1):
80         yv[s] = yv[s-1] + quad(lambda t: g(torch.tensor([[t]]),
81                                           model(torch.tensor([[t]]).detach()
82                                           ts[s-1], ts[s])[0]
83     loss_valid = torch.mean((ys - yv)**2)
84
85     if (loss_valid < best_loss):
```

```

86         torch.save(model, 'model.pt')
87         best_loss = loss_valid
88         count_no_change = 0
89     else:
90         count_no_change += 1
91
92     if ((epoch % eout == 0) or (count_no_change == 0)):
93         msg = f'{epoch}: train = {loss_train.item():.4e}, valid = {loss_vali
94         if (count_no_change == 0):
95             msg += ' (best)'
96         print(msg)
97
98     if ((best_loss < tol) or (count_no_change > n_iter_no_change)):
99         break
100
101     if (loss_train < tol):
102         break

```

### 4.1.2 AD-PINN

[[tag:construcao]]

Aqui nosso modelo PINN é novamnte uma **perceptron multicamada** (MLP)

$$\tilde{y} = \mathcal{N}\left(t; \left\{W^{(l)}, \mathbf{b}^{(l)}, \mathbf{f}^{(l)}\right\}_{l=1}^{n_h+1}\right), \quad (4.7)$$

de dada arquitetura  $1 - n_n \times n_h - 1$ , i.e. uma entrada,  $n_h$  camadas escondidas, cada com  $n_n$  neurônios e uma saída. A entrada é o valor do tempo  $t$  e a saída é  $\tilde{y} = \mathcal{N}(t) \approx y(t)$ , a estimativa da solução do IVP (4.2). Escolhidas as funções de ativação  $\mathbf{f}^{(l)}$ ,  $l = 1, 2, \dots, n_h + 1$ , o treinamento da PINN consiste em resolver o seguinte problema de minimização

$$\min_{\{W^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{n_h+1}} \frac{1}{n_s} \sum_{s=0}^{n_s-1} \left| \mathcal{R}^{(s)} \right|^2 + p_p \left| \tilde{y}^{(0)} - y_0 \right|^2, \quad (4.8)$$

onde  $p_p > 0$  é um dado **parâmetro de penalização** e  $\mathcal{R}^{(s)}$  é o **resíduo**

$$\mathcal{R}^{(s)} := y'^{(s)} - h_t g\left(t^{(s)}, y^{(s)}\right), \quad (4.9)$$

com  $y'^{(s)} \approx y'\left(t^{(s)}\right)$  computada por diferenciação automática da MLP.

**Exemplo 4.1.2.** Consideramos o seguinte IVP

$$y'(t) = \sin(t) - y, \quad 0 < t < 1, \quad (4.10a)$$

$$y(0) = \frac{1}{2}. \quad (4.10b)$$

Código 4.2: pyEulerPINN.py

```
1 import torch
2 from scipy.integrate import quad
3
4 # model
5 ## num hidden layers
6 nh = 2
7 ## num neurons per hidden layer
8 nn = 50
9 ## activation fun in hidden layers
10 fh = torch.nn.Tanh()
11 ## model architecture
12 model = torch.nn.Sequential()
13 model.add_module('layer_1', torch.nn.Linear(1, nn))
14 model.add_module('fun_1', fh)
15 for l in range(2, nh):
16     model.add_module(f'layer_{l}', torch.nn.Linear(nn, nn))
17     model.add_module(f'fun_{l}', fh)
18 model.add_module(f'layer_{nh}', torch.nn.Linear(nn, 1))
19
20 # IVP params
21
22 ## init time
23 t0 = 0.
24 ## init condition
25 y0 = 0.5
26 ## final time
27 tf = 1.
28
29 ## num of time samples
30 ns = 10
31 ## time step
32 ht = (tf - t0)/ns
33 ## time samples
```

```
34 ts = torch.linspace(t0, tf, ns+1).reshape(-1,1)
35
36 ## rhs
37 def g(t, y):
38     return y + torch.sin(t)
39
40 # training
41 ## num of epochs
42 nepochs = 10000
43 ## output loss freq
44 eout = 100
45 ## tolerance
46 tol = 1e-4
47 ## early-stop
48 n_iter_no_change = 100
49
50 ## optimizer
51 optim = torch.optim.Adam(model.parameters(), lr=1e-3)
52
53 # training loop
54 count_no_change = 0
55 best_loss = 1e38
56 for epoch in range(nepochs):
57
58     # forward
59     yy = model(ts)
60
61     # loss
62     ## t>0
63     lup = torch.mean(((yy[1:] - yy[:-1])/ht \
64                       - g(ts[:-1], yy[:-1]))**2)
65
66     ## t = 0
67     lic = (yy[0] - y0)**2
68
69     loss_train = lup + lic
70
71     # backward
72     optim.zero_grad()
73     loss_train.backward()
74     optim.step()
```

```
74
75     # validation
76     ys = model(ts).detach()
77     yv = torch.empty_like(ys)
78     yv[0] = y0
79     for s in range(1, ns+1):
80         yv[s] = yv[s-1] + quad(lambda t: g(torch.tensor([[t]]),
81                                         model(torch.tensor([[t]]).detach_(),
82                                         ts[s-1], ts[s]))[0]
83     loss_valid = torch.mean((ys - yv)**2)
84
85     if (loss_valid < best_loss):
86         torch.save(model, 'model.pt')
87         best_loss = loss_valid
88         count_no_change = 0
89     else:
90         count_no_change += 1
91
92     if ((epoch % eout == 0) or (count_no_change == 0)):
93         msg = f'{epoch}: train = {loss_train.item():.4e}, valid = {loss_valid.item():.4e}'
94         if (count_no_change == 0):
95             msg += ' (best)'
96         print(msg)
97
98     if ((best_loss < tol) or (count_no_change > n_iter_no_change)):
99         break
100
101     if (loss_train < tol):
102         break
```

### 4.1.3 Exercícios

[[tag:construcao]]

## 4.2 Aplicação: Equação de Laplace

[[tag:construcao]]

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

Vamos criar uma MLP para resolver

$$-\Delta u = 0, \mathbf{x} \in D = (0, 1)^2, \quad (4.11a)$$

$$u = u_{bc}, \mathbf{x} \in \partial D, \quad (4.11b)$$

com dada condição de contorno  $u_0 = u_0(\mathbf{x})$ .

Como exemplo, vamos considerar um problema com solução manufaturada

$$u(\mathbf{x}) = x_1(1 - x_1) - x_2(1 - x_2). \quad (4.12)$$

Código 4.3: pyEqLaplace

```

1 import torch
2 import matplotlib.pyplot as plt
3 import random
4 import numpy as np
5
6 # modelo
7 ## n camadas escondidas
8 nh = 3
9 ## n neurônios por camada
10 nn = 50
11 ## fun de ativação
12 fh = torch.nn.Tanh()
13 ## arquitetura
14 model = torch.nn.Sequential()
15 model.add_module('layer_1', torch.nn.Linear(2, nn))
16 model.add_module('fun_1', fh)
17 for layer in range(2, nh):
18     model.add_module(f'layer_{layer}', torch.nn.Linear(nn, nn))
19     model.add_module(f'fun_{layer}', fh)
20 model.add_module(f'layer_{nh}', torch.nn.Linear(nn, 1))
21
22 # SGD - (Stochastic) Gradient Descent
23 optim = torch.optim.SGD(model.parameters(),
24                           lr = 1e-2,
25                           momentum = 0.9)
26 optim = torch.optim.Adam(model.parameters(),
27                           lr = 1e-2)

```

```
28
29
30 # params treinamento
31 ## n épocas
32 nepochs = 10001
33 ## freq output loss
34 nout_loss = 100
35 ## stop criterion
36 tol = 1e-4
37
38 ## n amostras por eixo
39 ns = 101
40
41 lloss = []
42 for epoch in range(nepochs):
43
44     # forward
45
46     ## internal pts samples
47     Xin = torch.rand((ns, 2), requires_grad=True)
48     Uin = model(Xin)
49
50     ## loss internal pts
51     D1Uin = torch.autograd.grad(
52         Uin, Xin,
53         grad_outputs=torch.ones_like(Uin),
54         retain_graph=True,
55         create_graph=True)[0]
56     D2Uin = torch.autograd.grad(
57         D1Uin, Xin,
58         grad_outputs=torch.ones_like(D1Uin),
59         retain_graph=True,
60         create_graph=True)[0]
61
62     lin = torch.mean((D2Uin[:,0] + D2Uin[:,1])**2)
63
64     ## bc 1
65     xx = torch.rand((ns, 1))
66     yy = torch.zeros((ns,1))
67     Xbc1 = torch.hstack((xx, yy))
```

```
68     Ubc1 = model(Xbc1)
69
70     ## loss bc 1
71     Uexp = xx*(1. - xx)
72     lbc1 = torch.mean((Ubc1 - Uexp)**2)
73
74     ## bc 3
75     xx = torch.rand((ns, 1))
76     yy = torch.ones((ns,1))
77     Xbc3 = torch.hstack((xx, yy))
78     Ubc3 = model(Xbc3)
79
80     ## loss bc 3
81     Uexp = xx*(1. - xx)
82     lbc3 = torch.mean((Ubc3 - Uexp)**2)
83
84     ## bc 2
85     xx = torch.ones((ns, 1))
86     yy = torch.rand((ns,1))
87     Xbc2 = torch.hstack((xx, yy))
88     Ubc2 = model(Xbc2)
89
90     ## loss bc 2
91     Uexp = yy*(yy - 1.)
92     lbc2 = torch.mean((Ubc2 - Uexp)**2)
93
94     ## bc 4
95     xx = torch.zeros((ns, 1))
96     yy = torch.rand((ns,1))
97     Xbc4 = torch.hstack((xx, yy))
98     Ubc4 = model(Xbc4)
99
100    ## loss bc 3
101    Uexp = yy*(yy - 1.)
102    lbc4 = torch.mean((Ubc4 - Uexp)**2)
103
104    # loss function
105    loss = lin + lbc1 + lbc2 + lbc3 + lbc4
106
107    lloss.append(loss.item())
```



```
108
109     if (((epoch % nout_loss) == 0) or (loss.item() < tol)):
110         print(f'{epoch}: loss = {loss.item():.4e}')
111
112         # gráfico
113         fig = plt.figure()
114         ax = fig.add_subplot()
115
116         npts = 50
117         xx = torch.linspace(0., 1., npts)
118         yy = torch.linspace(0., 1., npts)
119         X, Y = torch.meshgrid(xx, yy)
120         # exact
121         Uexp = X*(1. - X) - Y*(1. - Y)
122         c = ax.contour(X, Y, Uexp, levels=10, colors='white')
123         ax.clabel(c)
124
125         M = torch.hstack((X.reshape(-1,1),
126                           Y.reshape(-1,1)))
127         Uest = model(M).detach()
128         Uest = Uest.reshape((npts, npts))
129         cf = ax.contourf(X, Y, Uest, levels=10, cmap='coolwarm')
130         plt.colorbar(cf)
131
132         ax.grid()
133         ax.set_xlabel('$x$')
134         ax.set_ylabel('$y$')
135         plt.title(f"epoch = {epoch}, loss = {loss.item():.4e}")
136         plt.savefig(f'results/sol_{epoch:0>6}.png', bbox_inches='tight')
137         plt.close()
138
139     if (loss.item() < tol):
140         break
141
142     # backward
143     optim.zero_grad()
144     loss.backward()
145     optim.step()
146
147
```

```

148 fig = plt.figure()
149 ax = fig.add_subplot()
150 ax.plot(lloss)
151 ax.set_yscale('log')
152 plt.show()

```

### 4.2.1 Preprocessamento

[[tag:construcao]]

Vamos assumir as seguintes mudanças de variáveis

$$\bar{x} = 2x - 1 \quad (4.13a)$$

$$\bar{y} = 2y - 1. \quad (4.13b)$$

Também, assumimos a notação  $\bar{u}(\bar{\mathbf{x}}) = u(\bar{\mathbf{x}}(\mathbf{x}))$ .

Então, segue que

$$\begin{aligned}
 \frac{\partial \bar{u}}{\partial \bar{x}} &= \frac{\partial}{\partial \bar{x}} u(\bar{\mathbf{x}}(\mathbf{x})) \\
 &= \frac{\partial u}{\partial x} \frac{\partial x}{\partial \bar{x}} \\
 &= \frac{1}{2} \frac{\partial u}{\partial x}.
 \end{aligned} \quad (4.14)$$

Também, temos

$$\begin{aligned}
 \frac{\partial^2 \bar{u}}{\partial \bar{x}^2} &= \frac{\partial}{\partial \bar{x}} \left( \frac{\partial \bar{u}}{\partial \bar{x}} \right) \\
 &= \frac{\partial}{\partial \bar{x}} \left( \frac{1}{2} \frac{\partial u}{\partial x} \right) \\
 &= \frac{\partial}{\partial x} \left( \frac{1}{2} \frac{\partial u}{\partial x} \right) \frac{\partial x}{\partial \bar{x}} \\
 &= \frac{1}{4} \frac{\partial^2 u}{\partial x^2}.
 \end{aligned} \quad (4.15)$$

Analogamente, temos

$$\frac{\partial \bar{u}}{\partial \bar{y}} = \frac{1}{2} \frac{\partial u}{\partial y} \quad (4.16)$$

e

$$\frac{\partial^2 \bar{u}}{\partial \bar{y}^2} = \frac{1}{4} \frac{\partial^2 u}{\partial y^2}. \quad (4.17)$$

Na nova variável  $\bar{\mathbf{x}}$  o problema de Laplace (4.11) é equivalente a

$$\frac{\partial^2 \bar{u}}{\partial \bar{x}^2} + \frac{\partial^2 \bar{u}}{\partial \bar{y}^2} = 0, \quad \bar{\mathbf{x}} = (\bar{x}, \bar{y}) \in (-1, 1)^2, \quad (4.18a)$$

$$\bar{u} = \bar{u}_0, \quad \bar{\mathbf{x}} \in \Gamma = \partial D. \quad (4.18b)$$

Código 4.4: pyEqLaplacePP

**Exemplo 4.2.1.** `import torch`

```

2 import matplotlib.pyplot as plt
3 import random
4 import numpy as np
5
6 # modelo
7 ## n camadas escondidas
8 nh = 3
9 ## n neurônios por camada
10 nn = 50
11 ## fun de ativação
12 fh = torch.nn.Tanh()
13 ## arquitetura
14 model = torch.nn.Sequential()
15 model.add_module('layer_1', torch.nn.Linear(2, nn))
16 model.add_module('fun_1', fh)
17 for layer in range(2, nh):
18     model.add_module(f'layer_{layer}', torch.nn.Linear(nn, nn))
19     model.add_module(f'fun_{layer}', fh)
20 model.add_module(f'layer_{nh}', torch.nn.Linear(nn, 1))
21
22 # SGD - (Stochastic) Gradient Descent
23 optim = torch.optim.SGD(model.parameters(),
24                             lr = 1e-2,
25                             momentum = 0.9)
26 optim = torch.optim.Adam(model.parameters(),
27                             lr = 1e-2)
```

```
28
650 29 # params treinamento
30 ## n épocas
31 nepochs = 10001
32 ## freq output loss
600 33 nout_loss = 100
34 ## stop criterion
35 tol = 1e-4
550 36
37 ## n amostras por eixo
38 ns = 101
39
500 40 lloss = []
41 for epoch in range(nepochs):
42
43     # forward
450 44
45     ## internal pts samples
46     Xin = 2.*torch.rand((ns, 2)) - 1.
47     Xin.requires_grad=True
400 48     Uin = model(Xin)
49
50     ## loss internal pts
350 51     D1Uin = torch.autograd.grad(
52         Uin, Xin,
53         grad_outputs=torch.ones_like(Uin),
54         retain_graph=True,
300 55         create_graph=True)[0]
56     D2Uin = torch.autograd.grad(
57         D1Uin, Xin,
250 58         grad_outputs=torch.ones_like(D1Uin),
59         retain_graph=True,
60         create_graph=True)[0]
61
200 62     lin = torch.mean((D2Uin[:,0] + D2Uin[:,1])**2)
63
64     ## bc 1
65     xx = 2.*torch.rand((ns, 1)) - 1.
150 66     yy = -1.*torch.ones((ns,1))
67     Xbc1 = torch.hstack((xx, yy))
```

```
68     Ubc1 = model(Xbc1)
69
70     ## loss bc 1
71     xx = (xx + 1.)/2.;
72     Uexp = xx*(1. - xx)
73     lbc1 = torch.mean((Ubc1 - Uexp)**2)
74
75     ## bc 3
76     xx = 2.*torch.rand((ns, 1)) -1.
77     yy = torch.ones((ns,1))
78     Xbc3 = torch.hstack((xx, yy))
79     Ubc3 = model(Xbc3)
80
81     ## loss bc 3
82     xx = (xx + 1.)/2.;
83     Uexp = xx*(1. - xx)
84     lbc3 = torch.mean((Ubc3 - Uexp)**2)
85
86     ## bc 2
87     xx = torch.ones((ns, 1))
88     yy = 2.*torch.rand((ns,1)) -1.
89     Xbc2 = torch.hstack((xx, yy))
90     Ubc2 = model(Xbc2)
91
92     ## loss bc 2
93     yy = (yy + 1.)/2.;
94     Uexp = yy*(yy - 1.)
95     lbc2 = torch.mean((Ubc2 - Uexp)**2)
96
97     ## bc 4
98     xx = -1.*torch.ones((ns, 1))
99     yy = 2.*torch.rand((ns,1)) -1.
100    Xbc4 = torch.hstack((xx, yy))
101    Ubc4 = model(Xbc4)
102
103    ## loss bc 3
104    yy = (yy + 1.)/2.;
105    Uexp = yy*(yy - 1.)
106    lbc4 = torch.mean((Ubc4 - Uexp)**2)
107
```

```

108     # loss function
650 109     loss = lin + lbc1 + lbc2 + lbc3 + lbc4
110
111     lloss.append(loss.item())
112
600 113     if (((epoch % nout_loss) == 0) or (loss.item() < tol)):
114         print(f'{epoch}: loss = {loss.item():.4e}')
115
550 116         # gráfico
117         fig = plt.figure()
118         ax = fig.add_subplot()
119
500 120         npts = 50
121         xx = torch.linspace(-1., 1., npts)
122         yy = torch.linspace(-1., 1., npts)
123         X, Y = torch.meshgrid(xx, yy)
450 124         # exact
125         Uexp = (X+1.)/2.*(1. - (X+1.)/2.) \
126             - (Y+1.)/2.*(1. - (Y+1.)/2.)
400 127         c = ax.contour(X, Y, Uexp, levels=10, colors='white')
128         ax.clabel(c)
129
130         M = torch.hstack((X.reshape(-1,1),
350 131                         Y.reshape(-1,1)))
132         Uest = model(M).detach()
133         Uest = Uest.reshape((npts, npts))
134         cf = ax.contourf(X, Y, Uest, levels=10, cmap='coolwarm')
300 135         plt.colorbar(cf)
136
137         ax.grid()
250 138         ax.set_xlabel('$\\bar{x}$')
139         ax.set_ylabel('$\\bar{y}$')
140         plt.title(f"epoch = {epoch}, loss = {loss.item():.4e}")
141         plt.savefig(f'results/sol_{epoch:0>6}.png', bbox_inches='tight')
200 142         plt.close()
143
144         if (loss.item() < tol):
145             break
150 146
147     # backward

```

```

148     optim.zero_grad()
149     loss.backward()
150     optim.step()
151
152 fig = plt.figure()
153 ax = fig.add_subplot()
154 ax.plot(lloss)
155 ax.set_yscale('log')
156 plt.show()

```

## 4.2.2 Exercícios

[[tag::construcao]]

## 4.3 Aplicação: Equação do Calor

[[tag:construcao]]

Consideramos o problema

$$u_t = u_{xx} + f, (t, x) \in (0, 1] \times (-1, 1), \quad (4.19a)$$

$$u(0, x) = \text{sen}(\pi x), x \in [-1, 1], \quad (4.19b)$$

$$u(t, -1) = u(t, 1) = 0, t \in (t_0, tf], \quad (4.19c)$$

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

$$u(t, x) = e^{-t} \text{sen}(\pi x). \quad (4.20)$$

### 4.3.1 Diferenças Finitas

[[tag:construcao]]

Assumimos a discretização no tempo  $t^{(k)} = kh_t$ ,  $k = 0, 1, 2, \dots, n_t$ , com passo  $h_t = 1/n_t$ . Para a discretização no espaço, assumimos  $x_i = -1 + ih_x$ ,  $i = 0, 1, 2, \dots, n_x$ , com passo  $h_x = 2/n_x$ . Ainda, denotando  $u_i^{(k)} \approx u(t^{(k)}, x_i)$ , usamos as seguintes fórmulas de diferenças finitas

$$u_t(t^{(k)}, x_i) \approx \frac{u_i^{(k)} - u_i^{(k-1)}}{h_t}, \quad (4.21)$$

para  $0 < k < n_t$ ,  $0 \leq i \leq n_x$  e

$$u_{xx}(t^{(k)}, x_i) \approx \frac{u_{i-1}^{(k)} - 2u_i^{(k)} + u_{i+1}^{(k)}}{h_x^2}, \quad (4.22)$$

para  $0 \leq k \leq n_t$  e  $0 < i < n_x$ .

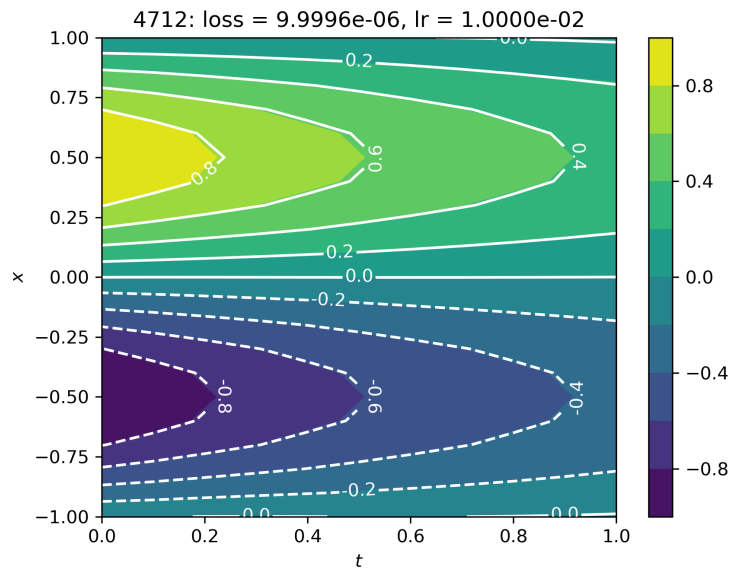


Figura 4.1: Soluções RNA (linhas brancas) *versus* analítica (cores de face) para o Problema 4.19.

Código 4.5: mlp\_calor.py

```
1 import torch
2 from torch import pi, sin, exp
3 import matplotlib.pyplot as plt
4
5 # modelo
6 model = torch.nn.Sequential(
7     torch.nn.Linear(2,500),
8     torch.nn.ELU(),
9     torch.nn.Linear(500,500),
10    torch.nn.ELU(),
```



```
11     torch.nn.Linear(500,500),
12     torch.nn.ELU(),
13     torch.nn.Linear(500,1)
14 )
15
16 # otimizador
17 optim = torch.optim.SGD(model.parameters(),
18                          lr = 1e-2, momentum = 0.9)
19 scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optim)
20
21 # amostras
22 nt = 10
23 ht = 1./nt
24 tt = torch.linspace(0., 1., nt+1)
25 nx = 20
26 hx = 2./nx
27 xx = torch.linspace(-1., 1., nx+1)
28 T, X = torch.meshgrid(tt, xx,
29                       indexing='ij')
30 Uesp = torch.empty_like(T)
31 nsamples = (nt+1)*(nx+1)
32 M = torch.empty((nsamples, 2))
33 s = 0
34 for i,t in enumerate(tt):
35     for j,x in enumerate(xx):
36         Uesp[i,j] = exp(-t)*sin(pi*x)
37         M[s,0] = t
38         M[s,1] = x
39         s += 1
40
41 # treinamento
42 nepochs = 10001
43 tol = 1e-5
44 nout = 100
45
46 for epoch in range(nepochs):
47
48     # forward
49     Uest = model(M)
50
```

```

51     # loss
52     ## c.i.
53     lci = torch.tensor([0.])
54     for j,x in enumerate(xx):
55         s = j
56         assert(M[s,1] == x)
57         uesp = sin(pi*x)
58         lci += (Uest[s] - uesp)**2
59     ## pts internos
60     lin = torch.tensor([0.])
61     for i in range(1,nt+1):
62         for j in range(1,nx):
63             s = j + i*(nx+1)
64             # u_t
65             l = (Uest[s] - Uest[s-nx-1])/ht
66             # u_xx
67             l -= (Uest[s-1] - 2*Uest[s] + Uest[s+1])/hx**2
68             # f
69             l -= (pi**2 - 1.)*exp(-M[s,0])*sin(pi*M[s,1])
70             lin += l**2
71     ## c.c.
72     lcc = torch.tensor([0.])
73     for i,t in enumerate(tt[1:]):
74         # x = 0
75         s = i*(nx+1)
76         lcc += Uest[s]**2
77         # x = 1
78         s = nx + i*(nx+1)
79         lcc += Uest[s]**2
80
81     loss = (lci + lin + lcc)/nsamples
82
83     lr = optim.param_groups[-1]['lr']
84     print(f'{epoch}: loss = {loss.item():.4e}, lr = {lr:.4e}')
85
86     # output
87     if ((epoch % nout == 0) or (loss.item() < tol)):
88         plt.close()
89         fig = plt.figure(dpi=300)
90         ax = fig.add_subplot()

```

```

91         cb = ax.contourf(T, X, Uesp,
650                             levels=10)
92         fig.colorbar(cb)
93         cl = ax.contour(T, X, Uest.detach().reshape(nt+1,nx+1),
94                             levels=10, colors='white')
600         ax.clabel(cl, fmt='%.1f')
96         ax.set_xlabel('$t$')
97         ax.set_ylabel('$x$')
550         plt.title(f'{epoch}: loss = {loss.item():.4e}, lr = {lr:.4e}')
100         plt.savefig(f'./results/sol_{epoch:0>6}.png')
101
102     if (loss.item() < tol):
500         break
103
104     # backward
105     scheduler.step(loss)
450     optim.zero_grad()
107     loss.backward()
108     optim.step()
400

```

### 4.3.2 Diferenciação Automética

[[tag:construcao]]

Código 4.6: 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
250 6  # modelo
7  hidden = [50]*8
8  activation = torch.nn.Tanh()
9  layerList = [('layer_0', torch.nn.Linear(2, hidden[0])),
200                ('activation_0', activation)]
10  for l in range(len(hidden)-1):
12      layerList.append((f'layer_{l+1}',
150                          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],

```

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

```
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
95         Uest = model(Sbc0)
```

```

96     lbc0 = torch.mean(Uest**2)
97
98     ## c.c. x = 1
99     Uest = model(Sbc1)
100     lbc1 = torch.mean(Uest**2)
101
102     loss = lin + lic + lbc0 + lbc1
103
104     lr = optim.param_groups[-1]['lr']
105     print(f'{epoch}: loss = {loss.item():.4e}, lr = {lr:.4e}')
106
107     # backward
108     scheduler.step(loss)
109     optim.zero_grad()
110     loss.backward()
111     optim.step()
112
113
114     # output
115     if ((epoch % nout == 0) or (loss.item() < tol)):
116         plt.close()
117         fig = plt.figure(dpi=300)
118         nt = 10
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')
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):
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
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}')
146         plt.savefig(f'./results/sol_{(epoch//nout):0>6}.png')
147
148     if ((loss.item() < tol) or (lr < 1e-6)):
149         break
```

# Resposta dos Exercícios

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

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

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



# Bibliografia

- [1] Goodfellow, I., Bengio, Y., Courville, A.. Deep learning, MIT Press, Cambridge, MA, 2016.
- [2] Neural Networks: A Comprehensive Foundation, Haykin, S.. Pearson:Delhi, 2005. ISBN: 978-0020327615.
- [3] 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.
- [4] 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.