

Redes Neurais Artificiais

Pedro H A Konzen

22 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 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

Capa	i
Licença	ii
Prefácio	iii
Sumário	v
1 Introdução	1
2 Perceptron	3
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
3 Perceptron Multicamadas	23
3.1 Modelo MLP	23
3.1.1 Treinamento	25
3.1.2 Aplicação: Problema de Classificação XOR	26
3.1.3 Exercícios	28
3.2 Aplicação: Problema de Classificação Binária	29
3.2.1 Dados	29
3.2.2 Modelo	30

CONTEÚDO

v

3.2.3	Treinamento e Teste	31
3.2.4	Verificação	33
3.2.5	Exercícios	35
3.3	Aplicação: Aproximação de Funções	35
3.3.1	Função unidimensional	35
3.3.2	Função bidimensional	38
3.3.3	Exercícios	41
3.4	Diferenciação Automática	42
3.4.1	Autograd MLP	47
3.4.2	Exercícios	50
4	Redes Informadas pela Física	53
4.1	Aplicação: Equação de Poisson	53
4.1.1	Exercícios	57
4.2	Aplicação: Equação do Calor	57
	Respostas dos Exercícios	62
	Bibliografia	63

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**

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

$$= w_1x_1 + w_2x_2 + \cdots + w_nx_n + b \quad (2.2)$$

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

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

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

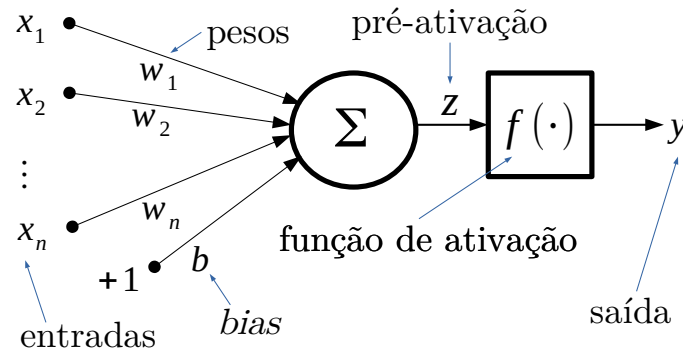


Figura 2.1: Esquema de um perceptron: unidade de processamento.

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 τ^+ e os demais pontos no semiplano negativo τ^- . Consultamos a Figura 2.2.

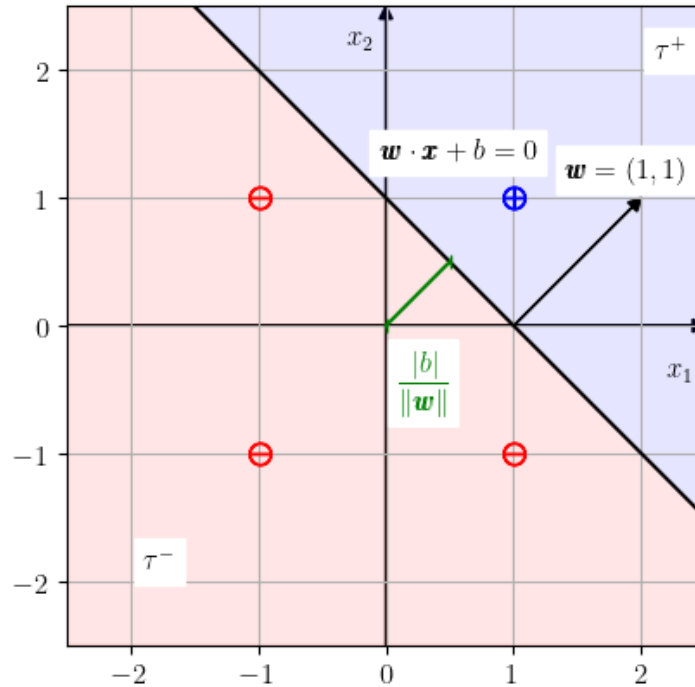


Figura 2.2: Interpretação geométrica do perceptron aplicado ao problema de classificação relacionado à operação lógica \wedge (e-lógico).

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 épocas¹.

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)

```

¹Número 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 perceptron²

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

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 normal³

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

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

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

²Escolhendo $f(z) = z$ como função de ativação.

³Consulte 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.

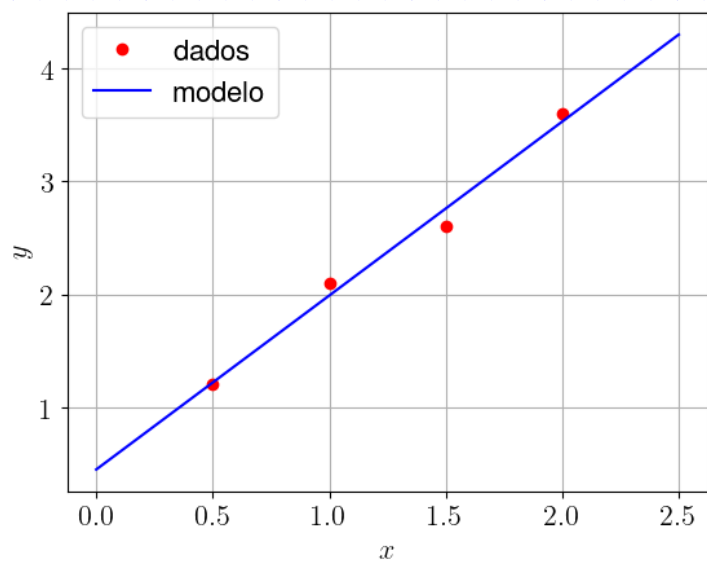


Figura 2.3: Interpretação geométrica do perceptron aplicado ao problema de regressão linear.

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 gerais⁴, 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

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

⁴Aqui, 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 segue⁵

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

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

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

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

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

⁵Aqui, 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 amostras⁶). 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 $e \leftarrow 1, \dots, n_e$:

⁶Neste 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

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.

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

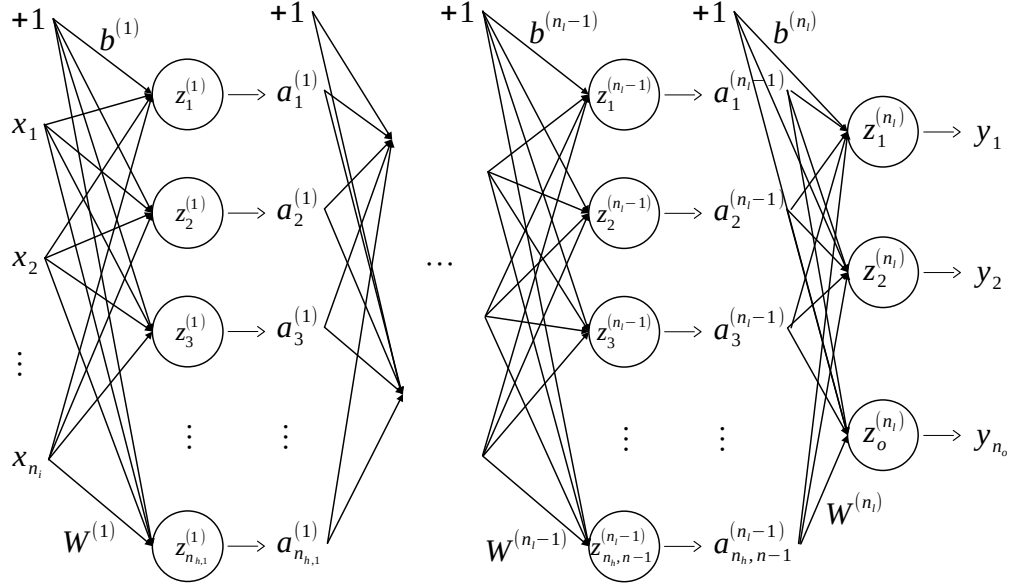


Figura 3.1: Estrutura de uma rede do tipo Perceptron Multicamadas (MLP).

Denotamos uma **MLP de n_l camadas** por

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

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

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

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

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

3.1.1 Treinamento

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 ε é uma dada **função erro** (em inglês, *loss function*) e $\tilde{\mathbf{y}}^{(s)}, \mathbf{y}^{(s)}$ são as saídas estimada e esperada da s -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 $\mathbf{d} = \mathbf{d}(\nabla_{\mathbf{W}, \mathbf{b}} \varepsilon)$ é o vetor direção, 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 por **retropropagação** (em inglês, *backward*). Para os pesos da última camada, temos

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

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

Para os pesos da penúltima, temos

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

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

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

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

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*)

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

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
17 ## dados de treinamento
18 X_train = torch.tensor([[1., 1.],
19                          [1., -1.],
20                          [-1., 1.],
21                          [-1., -1.]])
22 y_train = torch.tensor([-1., 1., 1., -1.]).reshape(-1,1)
23
24 print("\nDados de treinamento")
25 print("X_train = ")
26 print(X_train)
27 print("y_train = ")
28 print(y_train)
29
30 ## num max épocas
31 nepochs = 5000
32 tol = 1e-3
33
34 for epoch in range(nepochs):
```



```
35
36     # forward
37     y_est = model(X_train)
38
39     # função erro
40     loss = torch.mean((y_est - y_train)**2)
41
42     print(f'{epoch}: {loss.item():.4e}')
43
44     # critério de parada
45     if (loss.item() < tol):
46         break
47
48     # backward
49     optim.zero_grad()
50     loss.backward()
51     optim.step()
52
53
54 # verificação
55 y = model(X_train)
56 print(f'y_est = {y}')
```

3.1.3 Exercícios

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

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

Exercício 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.

Exercício 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.

Exercício 3.1.5. Considere uma MLP com $n_l = 3$ camadas escondidas. Sendo ε 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

[[tag:construcao]]

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()

```

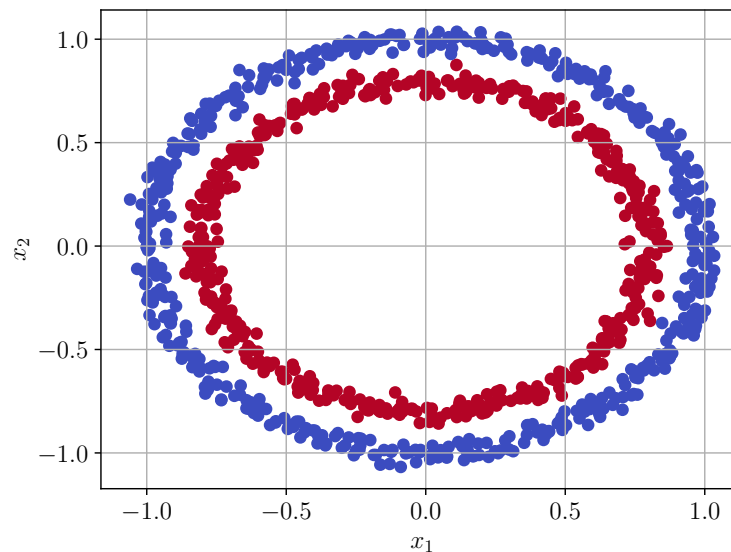


Figura 3.2: Dados para a o problema de classificação binária não linear.

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 100×100 pontos no domínio $[-1, 1]^2$. Consulte a Figure 3.3.

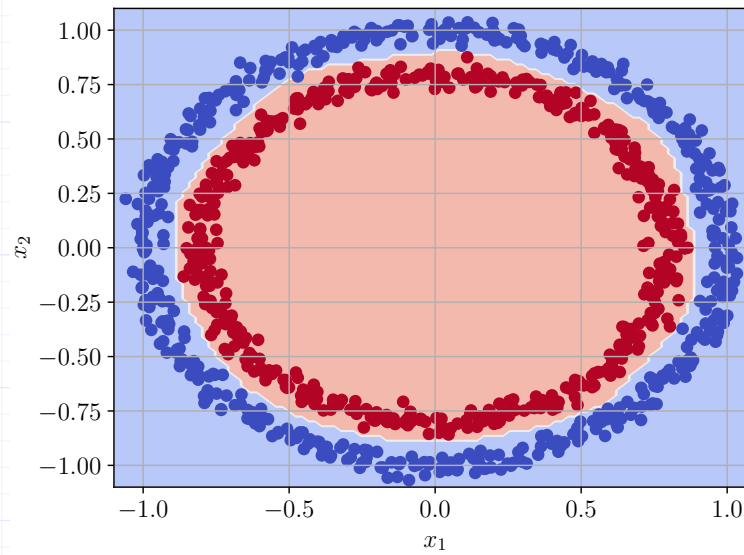


Figura 3.3: Verificação do modelo de classificação binária.

```

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

[[tag:construcao]]

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

$$y = \sin(\pi x), \quad (3.13)$$

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

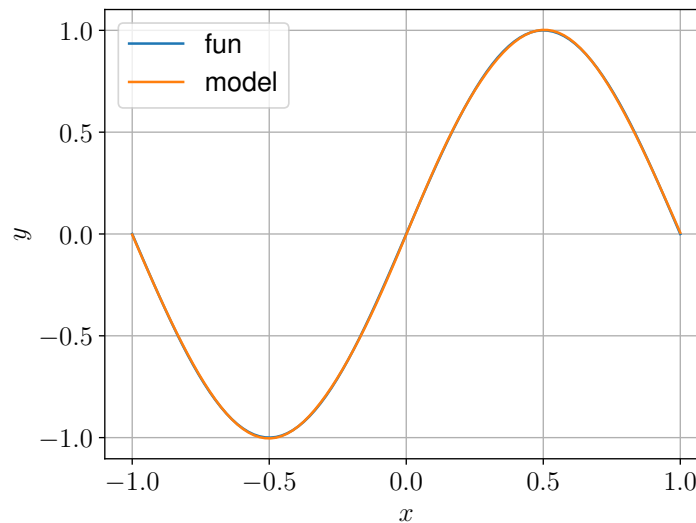


Figura 3.4: Aproximação da MLP da função $y = \sin(\pi x)$.

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)
```

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

$$y = \text{sen}(\pi x_1) \text{sen}(\pi x_2), \quad (3.14)$$

para $(x_1, x_2) \in [-1, 1]^2$.

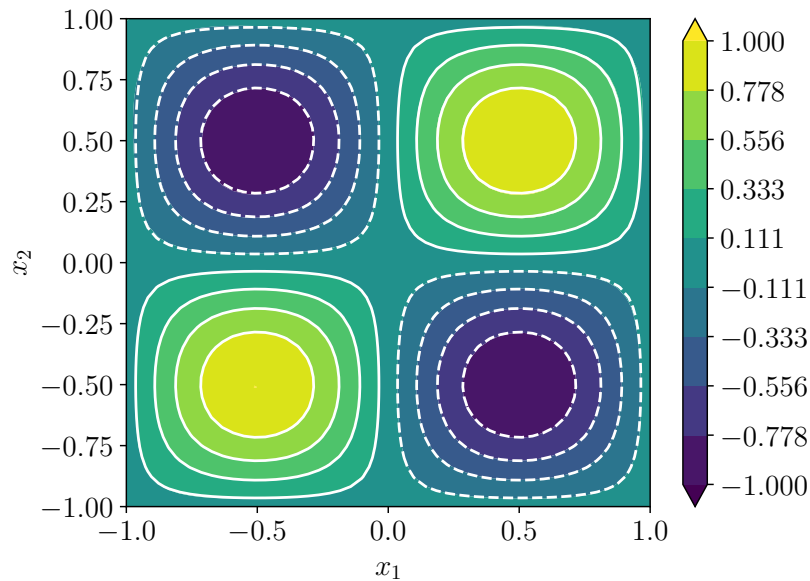


Figura 3.5: Aproximação MLP da função $y = \text{sen}(\pi x_1) \text{sen}(\pi x_2)$. Linhas: isolinhas da função. Mapa de cores: MLP.

Código 3.4: mlp_apfun_2d

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

```
11 model.add_module('layer_3', torch.nn.Linear(nh,nh))
12 model.add_module('fun_3', torch.nn.Tanh())
13 model.add_module(f'layer_4', torch.nn.Linear(nh,1))
14
15 # treinamento
16
17 ## fun obj
18 def fun(x1, x2):
19     return torch.sin(torch.pi*x1) * \
20         torch.sin(torch.pi*x2)
21
22 x1_a = -1.
23 x1_b = 1
24
25 x2_a = -1.
26 x2_b = 1.
27
28
29 ## otimizador
30 optim = torch.optim.SGD(model.parameters(),
31                           lr=5e-2, momentum=0.9)
32
33 ## num de amostras por época
34 ns = 10
35 ## num max épocas
36 nepochs = 50000
37 ## tolerância
38 tol = 1e-4
39
40 ## amostras de validação
41 n_val = 50
42 x1 = torch.linspace(x1_a, x1_b, steps=n_val)
43 x2 = torch.linspace(x2_a, x2_b, steps=n_val)
44 X1_val, X2_val = torch.meshgrid(x1, x2, indexing='ij')
45 X_val = torch.hstack((X1_val.reshape(n_val**2,1),
46                           X2_val.reshape(n_val**2,1)))
47 Y_vest = fun(X1_val, X2_val).reshape(-1,1)
48
49 for epoch in range(nepochs):
50
```

```
51     # amostras
52     x1 = (x1_b - x1_a) * torch.rand(ns) + x1_a
53     x2 = (x2_b - x2_a) * torch.rand(ns) + x2_a
54     X1, X2 = torch.meshgrid(x1, x2, indexing='ij')
55     X_train = torch.hstack((X1.reshape(-1,1),
56                             X2.reshape(-1,1)))
57     Y_train = fun(X1, X2).reshape(-1,1)
58
59
60     # forward
61     Y_est = model(X_train)
62
63     # erro
64     loss = torch.mean((Y_est - Y_train)**2)
65
66     if (epoch % 100 == 0):
67         print(f'{epoch}: {loss.item():.4e}')
68
69     # backward
70     optim.zero_grad()
71     loss.backward()
72     optim.step()
73
74     # validação
75     if (epoch % 100 == 0):
76         Y_val = model(X_val)
77         loss_val = torch.mean((Y_val - Y_vest)**2)
78
79         print(f"\tloss_val = {loss_val.item():.4e}")
80
81     # critério de parada
82     if (loss_val.item() < tol):
83         break
84
85
86     # # verificação
87     fig = plt.figure()
88     ax = fig.add_subplot()
89
90     Y_vest = Y_vest.reshape((n_val, n_val))
```

```
91 Y_val = Y_val.detach().reshape((n_val, n_val))
92
93 levels=10
94 ax.contour(X1_val, X2_val, Y_val, levels=levels, colors='white')
95 cb = ax.contourf(X1_val, X2_val, Y_val, levels=levels)
96 plt.colorbar(cb)
97
98 ax.set_xlabel('$x_1$')
99 ax.set_ylabel('$x_2$')
100 plt.show()
```

3.3.3 Exercícios

Exercício 3.3.1. Crie uma MLP para aproximar a função gaussiana

$$y = e^{-x^2} \quad (3.15)$$

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

Exercício 3.3.2. Crie uma MLP para aproximar a função $y = \sin(x)$ para $x \in [-\pi, \pi]$.

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

Exercício 3.3.4. Crie uma MLP para aproximar a função gaussiana

$$z = e^{-(x^2+y^2)} \quad (3.16)$$

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

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

Exercício 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

$$\nabla_{\mathbf{x}} f(\mathbf{x}_0) = \left. \frac{df}{d\mathbf{x}} \right|_{\mathbf{x}=\mathbf{x}_0}, \quad (3.17)$$

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 $f(x) = \sin(\pi x)$ e vamos computar

$$f'(x_0) = \left. \frac{df}{dx} \right|_{x=0} \quad (3.18)$$

por diferenciação automática.

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

$$\frac{df}{dx} = \frac{d}{du} \sin(u) \cdot \frac{du}{dx} \quad (3.19)$$

$$= \cos(u) \cdot \pi \quad (3.20)$$

$$= \pi \cos(\pi x) \quad (3.21)$$

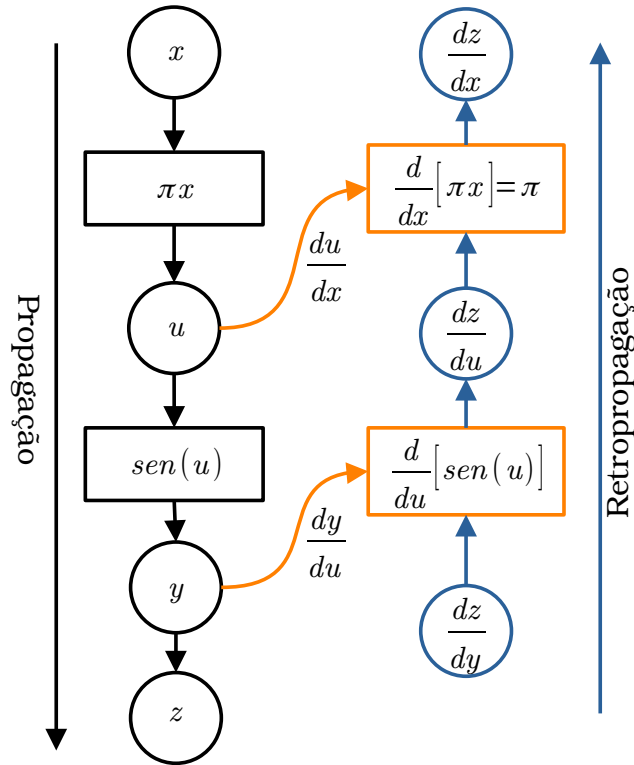


Figura 3.6: Grafo computacional da diferenciação automática de $f(x) = \text{sen}(\pi x)$.

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

$$\text{a. } \frac{dz}{dy} = 1 \quad (3.22a)$$

$$\begin{aligned} \text{b. } \frac{dz}{du} &= \frac{dy}{du} \frac{dz}{dy} \\ &= \frac{d}{du} [\text{sen}(u)] \cdot 1 \\ &= \cos(u) \end{aligned} \quad (3.22b)$$

$$c. \frac{dz}{dx} = \frac{du}{dx} \frac{dz}{du} \quad (3.22c)$$

$$= \frac{d}{dx} [\pi x] \cos(u) \quad (3.22d)$$

$$= \pi \cos(\pi x) = \frac{dy}{dx}. \quad (3.22e)$$

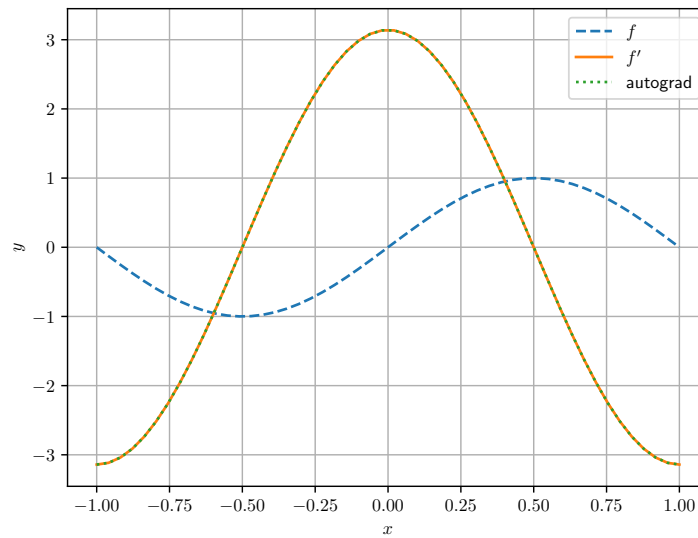


Figura 3.7: Comparação entre as diferenciações analítica (f') e automática (autograd).

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.

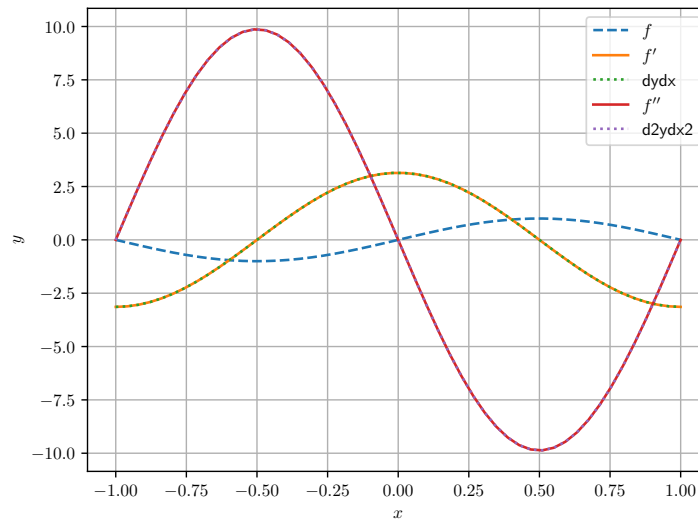


Figura 3.8: Comparação entre as diferenciações analítica (f' , f'') e automática (dydx, d2ydx2).

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, *Multilayer Perceptron*). Uma MLP é uma composição de funções definidas por parâmetros (pesos e *biases*). Seu treinamento ocorre em duas etapas¹:

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 erro² 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

$$\tilde{y} = \mathcal{N}\left(x; \left(W^{(l)}, \mathbf{b}^{(l)}, f^{(l)}\right)_{l=1}^n\right), \quad (3.23)$$

que aproxima a função

$$y = \text{sen}(\pi x), \quad x \in [-1, 1]. \quad (3.24)$$

Em seguida, computamos, por diferenciação automática, o gradiente

$$\frac{d\tilde{y}}{dx} = \nabla_x \mathcal{N}(x) \quad (3.25)$$

e comparamos com o resultado esperado

$$\frac{dy}{dx} = \pi \cos(\pi x). \quad (3.26)$$

¹Para mais detalhes, consulte a Subseção 3.1.1.

²Medida da diferença entre o valor estimado e o valor esperado.

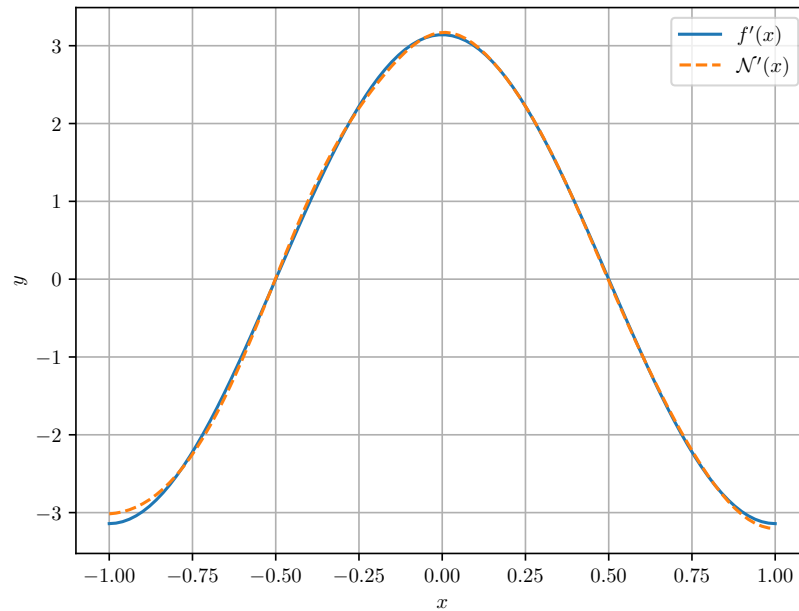


Figura 3.9: Comparação da diferenciação automática da MLP com a derivada analítica $f'(x) = \pi \cos(\pi x)$.

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

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

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

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

Exercício 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.

Exercício 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.

Exercício 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.

Exercício 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.

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

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

$$f(x,y) = \begin{bmatrix} xy^2 - x^2y + 6 \\ x + x^2y^3 - 7 \end{bmatrix} \quad (3.27)$$

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

[[tag:construcao]]

Vamos criar uma MLP para resolver

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

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

com dada fonte

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

Neste caso, a solução analítica é conhecida

$$u(x_1, x_2) = \sin(\pi x_1) \sin(\pi x_2). \quad (4.3)$$

Código 4.1: py_pinn_poisson

```
1 import torch
2 from torch import pi, sin
3 import matplotlib.pyplot as plt
4
5 # modelo
6 nn = 25
7 model = torch.nn.Sequential()
8 model.add_module('layer_1', torch.nn.Linear(2,nn))
9 model.add_module('fun_1', torch.nn.Tanh())
10 model.add_module('layer_2', torch.nn.Linear(nn,nn))
11 model.add_module('fun_2', torch.nn.Tanh())
12 model.add_module('layer_3', torch.nn.Linear(nn,nn))
13 model.add_module('fun_3', torch.nn.Tanh())
14 model.add_module('layer_4', torch.nn.Linear(nn,1))
15
16 # otimizador
17 optim = torch.optim.SGD(model.parameters(), #lr=1e-3)
18                               lr = 1e-3, momentum=0.9)
19
20 # fonte
21 def f(x1, x2):
22     return 2.*pi**2*sin(pi*x1)*sin(pi*x2)
23
24 # treinamento
25 ns = 25
26 nepochs = 50000
27 tol = 1e-3
28
29 for epoch in range(nepochs):
30
31     # forward
32     x1 = 2.*torch.rand(ns) - 1.
33     x2 = 2.*torch.rand(ns) - 1.
34     X1, X2 = torch.meshgrid(x1, x2, indexing='ij')
35     X = torch.hstack((X1.reshape(-1,1),
36                       X2.reshape(-1,1)))
37     X.requires_grad = True
38
39     U = model(X)
```

```
40
650 41     # gradientes
42     D1U = torch.autograd.grad(
43         U, X,
44         grad_outputs=torch.ones_like(U),
600 45         retain_graph=True,
46         create_graph=True)[0]
47     D2UX1 = torch.autograd.grad(
550 48         D1U[:,0:1], X,
49         grad_outputs=torch.ones_like(D1U[:,0:1]),
50         retain_graph=True,
51         create_graph=True)[0]
500 52     D2UX2 = torch.autograd.grad(
53         D1U[:,1:2], X,
54         grad_outputs=torch.ones_like(D1U[:,1:2]),
55         retain_graph=True,
450 56         create_graph=True)[0]
57
58     # fonte
400 59     F = f(X1, X2).reshape(-1,1)
60
61     # loss
62
350 63     # pts internos
64     lin = torch.mean((F + D2UX1[:,0:1] + D2UX2[:,1:2])**2)
65
66     # contornos
300 67     ## c.c. 1
68     Xbc1 = torch.hstack((X1.reshape(-1,1),
69                         -torch.ones((ns**2,1))))
250 70     Ubc1 = model(Xbc1)
71     lbc1 = torch.mean(Ubc1**2)
72
73     ## c.c. 3
200 74     Xbc3 = torch.hstack((X1.reshape(-1,1),
75                         torch.ones((ns**2,1))))
76     Ubc3 = model(Xbc3)
77     lbc3 = torch.mean(Ubc3**2)
150 78
79     ## c.c. 4
```

```
80     Xbc4 = torch.hstack((-torch.ones((ns**2,1)),
650     X2.reshape(-1,1)))
81
82     Ubc4 = model(Xbc4)
83     lbc4 = torch.mean(Ubc4**2)
84
600     ## c.c. 2
85
86     Xbc2 = torch.hstack((torch.ones((ns**2,1)),
87     X2.reshape(-1,1)))
550
88     Ubc2 = model(Xbc2)
89     lbc2 = torch.mean(Ubc2**2)
90
91     loss = lin + lbc1 + lbc2 + lbc3 + lbc4
500
92
93     if ((epoch % 100 == 0) or (loss.item() == 0)):
94         print(f'{epoch}: loss = {loss.item():.4e}')
95
450     if (loss.item() < tol):
96         break
97
98
400     optim.zero_grad()
100     loss.backward()
101     optim.step()
102
350     # verificação
103
104     def exact(x1, x2):
105         return sin(pi*x1)*sin(pi*x2)
106
300
107     ns = 50
108     x1 = torch.linspace(-1, 1, steps=ns)
109     x2 = torch.linspace(-1, 1, steps=ns)
250
110     X1, X2 = torch.meshgrid(x1, x2, indexing='ij')
111
112     Uexp = exact(X1, X2)
113
200
114     X = torch.hstack((X1.reshape(-1,1),
115     X2.reshape(-1,1)))
116     Uest = model(X).detach().reshape((ns, ns))
117
150
118     levels = 10
119     cb = plt.contourf(X1, X2, Uest, levels=levels)
```

```

120 plt.contour(X1, X2, Uexp, levels=levels, colors='white')
121 plt.colorbar(cb)
122 plt.show()

```

4.1.1 Exercícios

Exercício 4.1.1. Crie uma MLP para resolver

$$-\Delta u = 0, \mathbf{x} \in D = (0, 1)^2, \quad (4.4)$$

$$u(x_1, 0) = x_1(1 - x_1), 0 \leq x_1 \leq 1, \quad (4.5)$$

$$u(1, x_2) = x_2(1 - x_2), 0 < x_2 \leq 1, \quad (4.6)$$

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

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

4.2 Aplicação: Equação do Calor

[[tag:construcao]]

Consideramos o problema

$$u_t = u_{xx} + f, (t, x) \in (0, 1] \times (-1, 1), \quad (4.9a)$$

$$u(0, x) = \text{sen}(\pi x), x \in [-1, 1], \quad (4.9b)$$

$$u(t, -1) = u(t, 1) = 0, t \in (t_0, tf], \quad (4.9c)$$

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

$$u(t, x) = e^{-t} \text{sen}(\pi x). \quad (4.10)$$

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],
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
650 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))
600 51 s = 0
52 for i,t in enumerate(tin):
53     for j,x in enumerate(xin):
550 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
500 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
450 62 nepochs = 50001
63 tol = 1e-4
64 nout = 100
400 65
66 for epoch in range(nepochs):
67
68     # loss
350 69
70     ## c.i.
71     Uest = model(Sic)
72     lic = torch.mean((Uest - Uic)**2)
300 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,
200 80         create_graph=True)[0]
81     U_x = torch.autograd.grad(
82         U, xin,
83         grad_outputs=torch.ones_like(U),
150 84         retain_graph=True,
85         create_graph=True)[0]

```



```
86     U_xx = torch.autograd.grad(
650 87         U_x, xin,
88         grad_outputs=torch.ones_like(U_x),
89         retain_graph=True,
600 90         create_graph=True)[0]
91     res = U_t - U_xx - Fin
92     lin = torch.mean(res**2)
93
550 94     ## c.c. x = -1
95     Uest = model(Sbc0)
96     lbc0 = torch.mean(Uest**2)
97
500 98     ## c.c. x = 1
99     Uest = model(Sbc1)
100     lbc1 = torch.mean(Uest**2)
101
450 102     loss = lin + lic + lbc0 + lbc1
103
104     lr = optim.param_groups[-1]['lr']
400 105     print(f'{epoch}: loss = {loss.item():.4e}, lr = {lr:.4e}')
106
107     # backward
108     scheduler.step(loss)
350 109     optim.zero_grad()
110     loss.backward()
111     optim.step()
300 112
113
114     # output
250 115     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)
200 120         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)
150 124         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:.4e}')
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. $(\tanh x)' = 1 - \tanh^2 x$

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

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.