

Redes Neurais Artificiais

Pedro H A Konzen

21 de julho 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	13
2.2 Algoritmo de Treinamento	14
2.2.1 Método do Gradiente Descendente	15
2.2.2 Método do Gradiente Estocástico	18
2.2.3 Exercícios	21
3 Perceptron Multicamadas	22
3.1 Modelo MLP	22
3.1.1 Treinamento	23
3.1.2 Aplicação: Problema de Classificação XOR	24
3.1.3 Exercícios	27
3.2 Aplicação: Problema de Classificação Binária	27
3.2.1 Dados	27
3.2.2 Modelo	29

CONTEÚDO

v

3.2.3	Treinamento e Teste	29
3.2.4	Verificação	32
3.2.5	Exercícios	33
3.3	Aplicação: Aproximação de Funções	33
3.3.1	Função unidimensional	33
3.3.2	Função bidimensional	35
3.3.3	Exercícios	38
3.4	Diferenciação Automática	38
3.4.1	Autograd Perceptron	38
3.4.2	Autograd MLP	41
3.4.3	Exercícios	45
3.5	Aplicação: Equação de Laplace	45
3.5.1	Diferenças Finitas	45
3.5.2	Autograd	49
3.5.3	Exercícios	53

Respostas dos Exercícios

54

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** (consultemos a 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

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

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

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

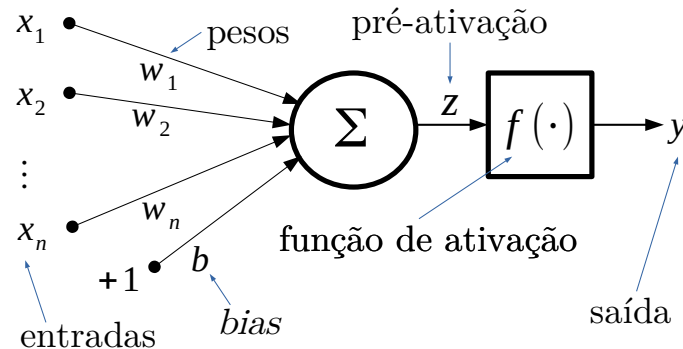


Figura 2.1: Esquema de um perceptron: unidade de processamento.

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 $R = A_1 \wedge A_2$. Consultamos a seguinte tabela verdade:

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.

$$\mathcal{N}(\mathbf{x}; (\mathbf{w}, b)) = \text{sign}(\mathbf{w} \cdot \mathbf{x} + b), \quad (2.6)$$

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-lo. 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

$$\mathbf{w} = [1, 1] \quad (2.7)$$

$$b = -1 \quad (2.8)$$

Com isso, nosso perceptron é

$$\mathcal{N}(\mathbf{x}) = \text{sign}(x_1 + x_2 - 1) \quad (2.9)$$

Verifique que ele satisfaz a tabela verdade acima!

Implementação

Código 2.1: perceptron.py

```
1 import torch
```

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

```

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
33 print(f"Valores estimados\n{y}")

```

Interpretação geométrica

Empregamos o seguinte modelo de neurônio

$$\mathcal{N}(\mathbf{x}; (\mathbf{w}, b)) = \text{sign}(w_1 x_1 + w_2 x_2 + b) \quad (2.10)$$

Observamos que

$$w_1x_1 + w_2x_2 + b = 0 \quad (2.11)$$

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

$$\tau^+ = \{\mathbf{x} \in \mathbb{R}^2 : w_1x_1 + w_2x_2 + b > 0\} \quad (2.12)$$

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

O primeiro está na direção do vetor normal a reta $\mathbf{n} = (w_1, w_2)$ e o segundo na sua direção oposta. Com isso, o problema de treinar nosso neurônio para nosso problema de classificação consiste em encontrar a reta

$$w_1x_1 + w_2x_2 + b = 0 \quad (2.14)$$

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

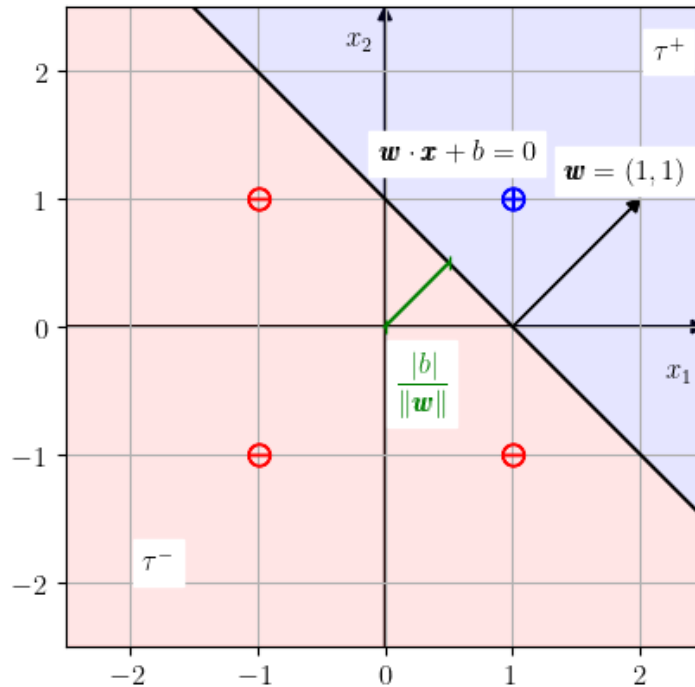


Figura 2.2: Interpretação geométrica do perceptron aplicado ao problema de classificação relacionado à operação lógica \wedge (e-lógico).

Algoritmo de treinamento: perceptron

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

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

1. $\mathbf{w} \leftarrow \mathbf{0}$, $b \leftarrow 0$.
2. Para $e \leftarrow 1, \dots, n_e$:
 - (a) Para $s \leftarrow 1, \dots, n_s$:
 - i. Se $y^{(s)}\mathcal{N}(\mathbf{x}^{(s)}) \leq 0$:
 - A. $\mathbf{w} \leftarrow \mathbf{w} + y^{(s)}\mathbf{x}^{(s)}$
 - B. $b \leftarrow b + y^{(s)}$

onde, n_e é um dado número de épocas¹.

```

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

```

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

```
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)
33 print("y_train = ")
34 print(y_train)
35
36 # treinamento
37
38 ## num max épocas
39 nepochs = 100
40
350 for epoch in range(nepochs):
41
42     # update
43     not_updated = True
44     for s in range(ns):
45         y_est = model(X_train[s:s+1,:])
46         if (y_est*y_train[s] <= 0.):
47             with torch.no_grad():
48                 W += y_train[s]*X_train[s,:]
49                 b += y_train[s]
50                 not_updated = False
51
52     if (not_updated):
53         print('Training ended.')
54         break
55
56
57
```

```

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

```

2.1.2 Problema de regressão

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

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

Modelo

Vamos determinar o perceptron²

$$\tilde{y} = \mathcal{N}(x; (w, b)) = wx + b \quad (2.15)$$

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.

$$\min_{w,b} \frac{1}{n_s} \sum_{s=1}^{n_s} (\tilde{y}^{(s)} - y^{(s)})^2 \quad (2.16)$$

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

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

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

$$= \frac{1}{n_s} \sum_{s=1}^{n_s} (wx^{(s)} + b - y^{(s)})^2 \quad (2.18)$$

Observamos que o problema (2.16) é equivalente a um problema linear de mínimos quadrados. A solução é obtida resolvendo-se a equação normal³

$$M^T M \mathbf{c} = M^T \mathbf{y}, \quad (2.19)$$

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

$$M = \begin{bmatrix} \mathbf{x} & \mathbf{1} \end{bmatrix} \quad (2.20)$$

Implementação

Código 2.2: perceptron_mq.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(1,1)
9
10    def forward(self, x):
11        z = self.linear(x)
12        return z
13
14 model = Perceptron()
15 with torch.no_grad():
16     W = model.linear.weight
17     b = model.linear.bias
18
19 # dados de treinamento
20 X_train = torch.tensor([0.5,
21                          1.0,
22                          1.5,
```

³Consulte o Exercício 2.1.4.


```

23         2.0]).reshape(-1,1)
24 y_train = torch.tensor([1.2,
25         2.1,
26         2.6,
27         3.6]).reshape(-1,1)
28
29 ## número de amostras
30 ns = y_train.size(0)
31
32 print("\nDados de treinamento")
33 print("X_train =")
34 print(X_train)
35 print("y_train = ")
36 print(y_train)
37
38 # treinamento
39
40 ## matriz
41 M = torch.cat((X_train,
42         torch.ones((ns,1))), dim=1)
43 ## solução M.Q.
44 c = torch.linalg.lstsq(M, y_train)[0]
45 with torch.no_grad():
46     W = c[0]
47     b = c[1]
48
49 # verificação
50 print(f'W = \n{W}')
51 print(f'b = \n{b}')
52 y = model(X_train)
53 print(f'y = \n{y}')
```

Resultado

Nosso perceptron corresponde ao modelo

$$\mathcal{N}(x; (w, b)) = wx + b \quad (2.21)$$

com os pesos treinados $w = 1.54$ e $b = 0.45$. Ele corresponde à reta que melhor se ajusta ao conjunto de dados de $\{x^{(s)}, y^{(s)}\}$. Consulte a Figura 2.3.

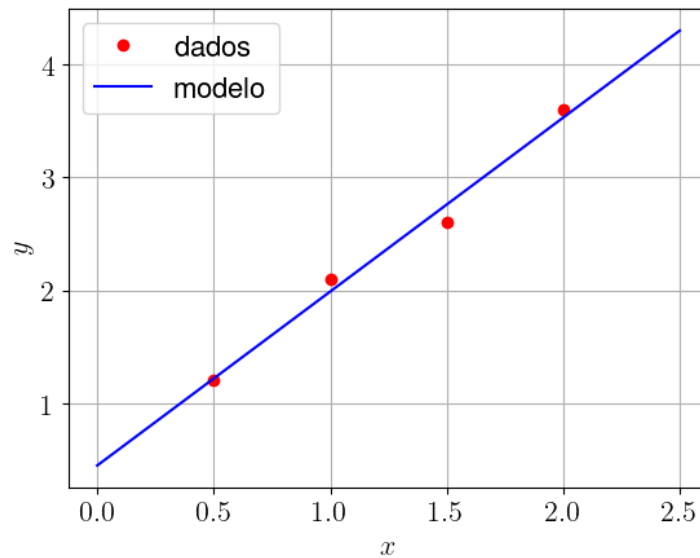


Figura 2.3: Interpretação geométrica do perceptron aplicado ao problema de regressão linear.

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 (2.15), mostre que (2.17) é função convexa.

Exercício 2.1.4. Mostre que a solução do problema (2.16) é dada por (2.19).

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

$$\tilde{y} = \mathcal{N}(\mathbf{x}; (\mathbf{w}, b)) = f(\underbrace{\mathbf{w} \cdot \mathbf{x} + b}_z), \quad (2.22)$$

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

$$z := \mathbf{w} \cdot \mathbf{x} + b \quad (2.23)$$

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

$$\varepsilon(\mathbf{w}, b) := \frac{1}{n_s} \sum_{s=1}^{n_s} (\tilde{y}^{(s)} - y^{(s)})^2 \quad (2.24)$$

$$= \frac{1}{n_s} \sum_{s=1}^{n_s} \varepsilon^{(s)} \quad (2.25)$$

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

$$\varepsilon^{(s)} := (\tilde{y}^{(s)} - y^{(s)})^2. \quad (2.26)$$

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

$$\min_{(\mathbf{w}, b)} \varepsilon(\mathbf{w}, b) \quad (2.27)$$

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

$$\frac{\partial \varepsilon}{\partial (\mathbf{w}, b)} := \left(\frac{\partial \varepsilon}{\partial w_1}, \dots, \frac{\partial \varepsilon}{\partial w_{n_{in}}}, \frac{\partial \varepsilon}{\partial b} \right) \quad (2.28)$$

O cálculo do gradiente para os pesos \mathbf{w} pode ser feito como segue

$$\frac{\partial \varepsilon}{\partial \mathbf{w}} = \frac{\partial}{\partial \mathbf{w}} \left[\frac{1}{n_s} \sum_{s=1}^{n_s} \varepsilon^{(s)} \right] \quad (2.29)$$

$$= \frac{1}{n_s} \sum_{s=1}^{n_s} \frac{\partial \varepsilon^{(s)}}{\partial \tilde{y}^{(s)}} \frac{\partial \tilde{y}^{(s)}}{\partial \mathbf{w}} \quad (2.30)$$

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

Observando que

$$\frac{\partial \varepsilon^{(s)}}{\partial \tilde{y}^{(s)}} = 2 \left(\tilde{y}^{(s)} - y^{(s)} \right) \quad (2.32)$$

$$\frac{\partial \tilde{y}^{(s)}}{\partial z^{(s)}} = f' \left(z^{(s)} \right) \quad (2.33)$$

$$\frac{\partial z^{(s)}}{\partial \mathbf{w}} = \mathbf{x}^{(s)} \quad (2.34)$$

obtemos

$$\frac{\partial \varepsilon}{\partial \mathbf{w}} = \frac{1}{n_s} \sum_{s=1}^{n_s} 2 \left(\tilde{y}^{(s)} - y^{(s)} \right) f' \left(z^{(s)} \right) \mathbf{x}^{(s)} \quad (2.35)$$

$$\frac{\partial \varepsilon}{\partial b} = \frac{1}{n_s} \sum_{s=1}^{n_s} \frac{\partial \varepsilon^{(s)}}{\partial \tilde{y}^{(s)}} \frac{\partial \tilde{y}^{(s)}}{\partial z^{(s)}} \frac{\partial z^{(s)}}{\partial b} \quad (2.36)$$

$$\frac{\partial \varepsilon}{\partial b} = \frac{1}{n_s} \sum_{s=1}^{n_s} 2 \left(\tilde{y}^{(s)} - y^{(s)} \right) f' \left(z^{(s)} \right) \cdot 1 \quad (2.37)$$

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

$$f(x) = \tanh(x). \quad (2.38)$$

Código 2.3: 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=1e-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 = 5000
```

```
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). A estocasticidade é obtida da randomização com que as amostras são escolhidas a cada época. O algoritmo consiste no seguinte:

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

1.1. Para $s \leftarrow \text{random}(1, \dots, n_s)$:

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - l_r \frac{\partial \varepsilon^{(s)}}{\partial (\mathbf{w}, b)} \quad (2.39)$$

Aplicação: Problema de Classificação

Código 2.4: 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=1e-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

$$f(x) = \tanh(x). \quad (2.40)$$

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. Consulte a Figura 3.1.

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

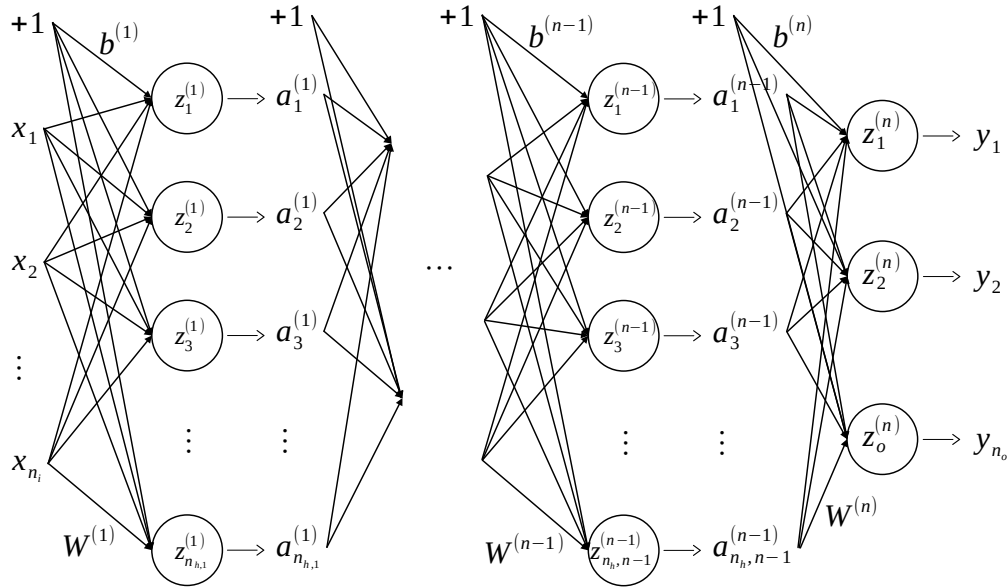


Figura 3.1: Estrutura de uma rede do tipo Perceptron Multicamadas (MLP).

Denotamos uma MLP de n 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 tripla 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.

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

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

3.1.1 Treinamento

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

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

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 l -ésima amostra, respectivamente.

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

1. W, \mathbf{b} aproximações iniciais.

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

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

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

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

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

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

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

Para os pesos da penúltima, temos

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

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

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

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

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

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*)

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

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

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
450 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

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

3.2.1 Dados

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()
```

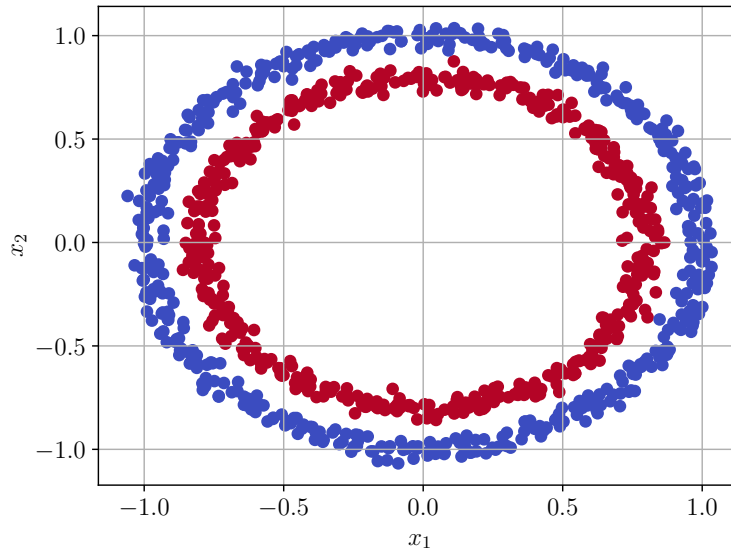


Figura 3.2: Dados para a o problema de classificação binária não linear.

3.2.2 Modelo

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

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

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

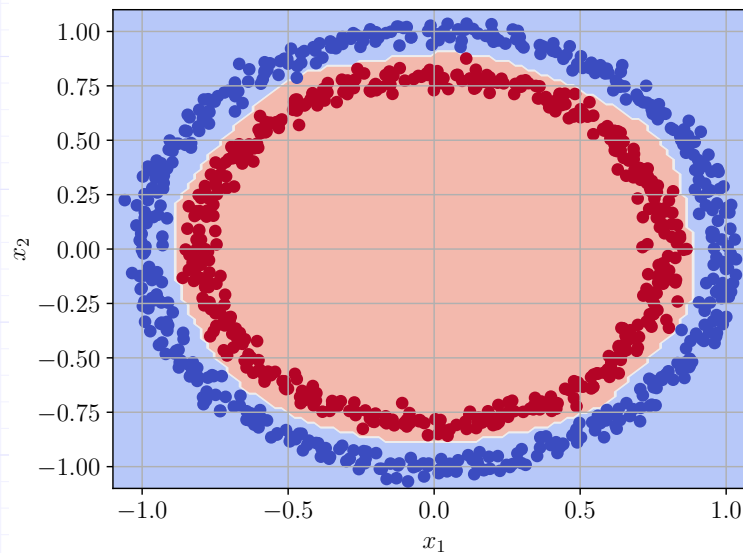


Figura 3.3: Verificação do modelo de classificação binária.

```

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):
650     z = model(torch.tensor([[xg, yg]])).detach()
10     Zg[i, j] = torch.round(z)
11
12 # visualização
600 13 fig = plt.figure()
14 ax = fig.add_subplot()
15 ax.contourf(Xg, Yg, Zg, levels=2, cmap=plt.cm.coolwarm, alpha=0.5)
550 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 (MLP) 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 gaussiana

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

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

```

1 import torch
250 2 import matplotlib.pyplot as plt
3
4 # modelo
5
200 6 model = torch.nn.Sequential(
7     torch.nn.Linear(1,25),
8     torch.nn.Tanh(),
150 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
600 57 # verificação
58 fig = plt.figure()
59 ax = fig.add_subplot()
60
550 61 x = torch.linspace(a, b,
62                       steps=50).reshape(-1,1)
63
500 64 y_esp = fobj(x)
65 ax.plot(x, y_esp, label='fobj')
66
450 67 y_est = model(x)
68 ax.plot(x, y_est.detach(), label='model')
69
70 ax.legend()
400 71 ax.grid()
72 ax.set_xlabel('x')
73 ax.set_ylabel('y')
350 74 plt.show()

```

3.3.2 Função bidimensional

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
200 4  # modelo
5
150 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
650 51     y_est = model(X_train)
52
53     # erro
600 54     loss = loss_fun(y_est, y_train)
55
56     print(f'{epoch}: {loss.item():.4e}')
57
580 58     # critério de parada
59     if (loss.item() < tol):
60         break
61
500 62     # backward
63     optim.zero_grad()
64     loss.backward()
450 65     optim.step()
66
67
68     # verificação
400 69     fig = plt.figure()
70     ax = fig.add_subplot()
71
72     n = 50
350 73     x0 = torch.linspace(a, b, steps=n)
74     x1 = torch.linspace(a, b, steps=n)
75     X0, X1 = torch.meshgrid(x0, x1)
76     X = torch.cat((X0.reshape(-1,1),
300 77                     X1.reshape(-1,1)),
78                     dim=1)
79
250 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')
200 84     ax.clabel(c)
85
86     y_est = model(X)
87     Y = y_est.reshape((n,n))
150 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

Uma RNA é uma composição de funções definidas por parâmetros (pesos e *biases*). O treinamento de uma RNA 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.

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

Para um Perceptron³

$$\tilde{y} = \mathcal{N}(\mathbf{x}, (\mathbf{w}, b)) \quad (3.15a)$$

$$= f(\underbrace{\mathbf{w} \cdot \mathbf{x} + b}_z) \quad (3.15b)$$

¹Para mais detalhes, consulte a Subseção 3.1.1.

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

³Consulte o Capítulo 2 para mais informações sobre o Perceptron.

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

$$\frac{\partial \tilde{y}}{\partial \mathbf{x}} = \frac{\partial f}{\partial z} \frac{\partial z}{\partial \mathbf{x}} \quad (3.16a)$$

$$= f'(z) \mathbf{w} \quad (3.16b)$$

Exemplo 3.4.1. Vamos treinar um Perceptron com função de ativação $f(z) = z$

$$\tilde{y} = \mathcal{N}(x; (w, b)) \quad (3.17a)$$

$$= wx + b \quad (3.17b)$$

que se ajusta ao conjunto de pontos⁴

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

Uma vez treinado com função erro MSE⁵, espera-se que o Perceptron corresponda a reta de mínimos quadrados⁶

$$y = 1.54x + 0.45 \quad (3.18)$$

Portanto, espera-se que

$$\frac{\partial \tilde{y}}{\partial x} = 1.54. \quad (3.19)$$

Código 3.3: autograd_percep.py

```
1 import torch
2
3 # modelo
4 model = torch.nn.Linear(1,1)
```

⁴Consulte o Exercício 2.2.4.

⁵MSE, Erro Quadrático Médio.

⁶Para mais informações sobre essa aplicação, consulte a Subseção 2.1.2.

```
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
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
650 46     print(f'{epoch}: {loss.item():.4e}, '\
47           + f'cstop = {cstop}/{nstop}')
48
49     if (cstop == nstop):
600 50         break
51
52     # backward
550 53     optim.zero_grad()
54     loss.backward()
55     optim.step()
56
500 57
58 # verificação
59 print(f'w = {model.weight}')
60 print(f'b = {model.bias}')
450 61
62 # autograd dy/dx
63
400 64 ## forward
65 x = torch.tensor([[1.]],
66                   requires_grad=True)
67 y = model(x)
350 68
69 ## backward
70 y.backward()
71 dydx = x.grad
300 72 print(f'dy/dx = {dydx}')

```

3.4.2 Autograd MLP

Os conceitos de diferenciação automática (**autograd**) são diretamente entendidos 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 3.4.2. Vamos criar uma MLP

$$\tilde{y} = \mathcal{N}\left(x; \left(W^{(l)}, \mathbf{b}^{(l)}, f^{(l)}\right)_{l=1}^n\right), \quad (3.20)$$

que aproxima a função $y = \sin(\pi x)$ 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()
```

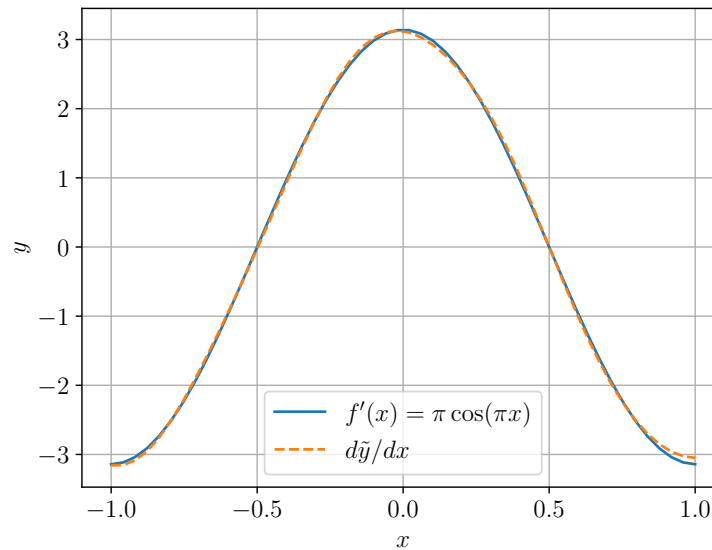



Figura 3.4: Comparação da autograd da MLP com a derivada exata $f'(x) = \pi \cos(\pi x)$ para o Exemplo 3.4.2.

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]]

3.5 Aplicação: Equação de Laplace

Vamos criar uma MLP para resolver

$$-\Delta u = 0, \quad \mathbf{x} \in D = (0, 1)^2, \quad (3.21a)$$

$$u = 0, \quad \mathbf{x} \in \partial D. \quad (3.21b)$$

Como exemplo, vamos considerar um problema com solução manufaturada

$$u(\mathbf{x}) = x_1(1 - x_1) - x_2(1 - x_2). \quad (3.22)$$

3.5.1 Diferenças Finitas

Código 3.5: mlp_eqlaplace_df.py

```

1 import torch
2 import matplotlib.pyplot as plt
3 import random
4 import numpy as np
5
6 # modelo
7 model = torch.nn.Sequential(

```

```
8     torch.nn.Linear(2,50),
9     torch.nn.Tanh(),
10    torch.nn.Linear(50,10),
11    torch.nn.Tanh(),
12    torch.nn.Linear(10,5),
13    torch.nn.Tanh(),
14    torch.nn.Linear(5,1)
15 )
16
17 # SGD - (Stochastic) Gradient Descent
18 optim = torch.optim.SGD(model.parameters(),
19                          lr = 1e-3,
20                          momentum = 0.9,
21                          dampening = 0.)
22
23 # Solução esperada
24 def u(x, y):
25     return a*x*(1-x) - a*y*(1-y)
26
27
28 def laplace_loss(X, U, h2, n, uc=u, p=1.):
29     # num de amostras
30     nc = 2*n + 2*(n-2)
31     ni = n**2 - nc
32
33     # loss interno
34     lin = 0.
35     for i in range(1,n-1):
36         for j in range(1,n-1):
37             s = j + i*n
38             l = (U[s-n, 0] - 2 * U[s, 0] + U[s+n, 0])/h2 # x
39             l += (U[s-1, 0] - 2 * U[s, 0] + U[s+1, 0])/h2 # y
40             lin += l**2
41     lin /= ni
42
43     # loss contorno
44     lc = 0.
45     # 0 <= x <= 1 e y == 0
46     for i in range(n):
47         s = i*n
```

```
48         x = M[s,0]
650 49         y = M[s,1]
50         lc += (U[s,0] - uc(x,y))**2
51         # 0 <= x <= 1 e y == 1
52         for i in range(n):
600 53             s = n-1 + i*n
54             x = M[s,0]
55             y = M[s,1]
550 56             lc += (U[s,0] - uc(x,y))**2
57             # 0 == x e 0 < y < 1
58             for j in range(1, n-1):
59                 s = j
500 60                 x = M[s,0]
61                 y = M[s,1]
62                 lc += (U[s,0] - uc(x,y))**2
63                 # 1 == x e 0 < y < 1
450 64                 for j in range(1, n-1):
65                     s = j + n*(n-1)
66                     x = M[s,0]
400 67                     y = M[s,1]
68                     lc += (U[s,0] - uc(x,y))**2
69         lc *= p/nc
70
350 71         loss = lin + lc
72         return loss
73
74
300 75 # dados do problema
76
77 # collocation points
250 78 a = 1
79 n = 11
80 ns = n**2
81 h = 1./(n-1)
200 82 h2 = h**2
83
84 # malha
150 85 x = torch.linspace(0, 1, n)
86 y = torch.linspace(0, 1, n)
87
```

```
88 M = torch.empty((ns, 2))
89 s = 0
90 for i, xx in enumerate(x):
91     for j, yy in enumerate(y):
92         M[s,0] = xx
93         M[s,1] = yy
94         s += 1
95
96 # gráfico
97 X, Y = np.meshgrid(x, y)
98 U_esp = u(X, Y)
99
100 # training
101 nepochs = 10000
102 nout_loss = 100
103 nout_plot = 500
104
105 for epoch in range(nepochs):
106
107     # forward
108     U_est = model(M)
109
110     # loss function
111     loss = laplace_loss(M, U_est, h2, n, u, p=10.)
112
113     if ((epoch % nout_loss) == 0):
114         print(f'{epoch}: loss = {loss.item():.4e}')
115
116     # output current solution
117     if ((epoch) % nout_plot == 0):
118         # verificação
119         fig = plt.figure()
120         ax = fig.add_subplot()
121
122         ns = 50
123         x1 = torch.linspace(0., 1., ns)
124         x2 = torch.linspace(0., 1., ns)
125         X1, X2 = torch.meshgrid(x1, x2)
126         # exact
127         Z_esp = torch.empty_like(X1)
```

```
128     for i,x in enumerate(x1):
650 129         for j,y in enumerate(x2):
130             Z_esp[i,j] = u(x, y)
131     c = ax.contour(X1, X2, Z_esp, levels=10, colors='white')
132     ax.clabel(c)
600 133
134     X_plot = torch.cat((X1.reshape(-1,1),
135                        X2.reshape(-1,1)), dim=1)
550 136     Z_est = model(X_plot)
137     Z_est = Z_est.reshape((ns,ns))
138     cf = ax.contourf(X1, X2, Z_est.detach(), levels=10, cmap='coolwarm')
139     plt.colorbar(cf)
500 140
141     ax.grid()
142     ax.set_xlabel('$x_1$')
143     ax.set_ylabel('$x_2$')
450 144     plt.show()
145
146     # backward
400 147     optim.zero_grad()
148     loss.backward()
149     optim.step()
```

3.5.2 Autograd

Código 3.6: mlp_eqlaplace_ag.py

```
300 1  import torch
2  import matplotlib.pyplot as plt
3  import random
250 4  import numpy as np
5
6  # modelo
200 7  model = torch.nn.Sequential(
8      torch.nn.Linear(2,50),
9      torch.nn.Tanh(),
10     torch.nn.Linear(50,10),
150 11     torch.nn.Tanh(),
12     torch.nn.Linear(10,5),
13     torch.nn.Tanh(),
```

```
14     torch.nn.Linear(5,1)
15 )
16
17 # SGD - (Stochastic) Gradient Descent
18 optim = torch.optim.SGD(model.parameters(),
19                           lr = 1e-3,
20                           momentum = 0.9,
21                           dampening = 0.)
22
23 # Solução esperada
24 def u(x, y):
25     return a*x*(1-x) - a*y*(1-y)
26
27
28 def laplace_loss(X, U, h2, n, uc=u, p=1.):
29     # num de amostras
30     nc = 2*n + 2*(n-2)
31     ni = n**2 - nc
32
33     # loss interno
34     lin = 0.
35     for i in range(1,n-1):
36         for j in range(1,n-1):
37             s = j + i*n
38             x = X[s:s+1,:].detach()
39             x.requires_grad = True
40             u = model(x)
41             grad_u = torch.autograd.grad(u, x,
42                                           create_graph = True,
43                                           retain_graph = True)[0]
44             u_x = grad_u[0,0]
45             u_y = grad_u[0,1]
46
47             u_xx = torch.autograd.grad(u_x, x,
48                                       create_graph = True,
49                                       retain_graph = True)[0][0,0]
50             u_yy = torch.autograd.grad(u_y, x,
51                                       create_graph = True,
52                                       retain_graph = True)[0][0,1]
53             lin = torch.add(lin, (u_xx + u_yy)**2)
```

```
54     lin /= ni
55
56     # loss contorno
57     lc = 0.
58     # 0 <= x <= 1 e y == 0
59     for i in range(n):
60         s = i*n
61         x = M[s,0]
62         y = M[s,1]
63         lc += (U[s,0] - uc(x,y))**2
64     # 0 <= x <= 1 e y == 1
65     for i in range(n):
66         s = n-1 + i*n
67         x = M[s,0]
68         y = M[s,1]
69         lc += (U[s,0] - uc(x,y))**2
70     # 0 == x e 0 < y < 1
71     for j in range(1, n-1):
72         s = j
73         x = M[s,0]
74         y = M[s,1]
75         lc += (U[s,0] - uc(x,y))**2
76     # 1 == x e 0 < y < 1
77     for j in range(1, n-1):
78         s = j + n*(n-1)
79         x = M[s,0]
80         y = M[s,1]
81         lc += (U[s,0] - uc(x,y))**2
82     lc *= p/nc
83
84     loss = lin + lc
85     return loss
86
87
200 88 # dados do problema
89
90 # collocation points
91 a = 1
150 92 n = 11
93 ns = n**2
```



```
94 h = 1./(n-1)
95 h2 = h**2
96
97 # malha
98 x = torch.linspace(0, 1, n)
99 y = torch.linspace(0, 1, n)
100
101 M = torch.empty((ns, 2))
102 s = 0
103 for i, xx in enumerate(x):
104     for j, yy in enumerate(y):
105         M[s,0] = xx
106         M[s,1] = yy
107         s += 1
108
109 # gráfico
110 X, Y = np.meshgrid(x, y)
111 U_esp = u(X, Y)
112
113 # training
114 nepochs = 10000
115 nout_loss = 100
116 nout_plot = 500
117
118 for epoch in range(nepochs):
119
120     # forward
121     U_est = model(M)
122
123     # loss function
124     loss = laplace_loss(M, U_est, h2, n, u, p=10.)
125
126     if ((epoch % nout_loss) == 0):
127         print(f'{epoch}: loss = {loss.item():.4e}')
128
129     # output current solution
130     if ((epoch) % nout_plot == 0):
131         # verificação
132         fig = plt.figure()
133         ax = fig.add_subplot()
```

```
134
135     ns = 50
136     x1 = torch.linspace(0., 1., ns)
137     x2 = torch.linspace(0., 1., ns)
138     X1, X2 = torch.meshgrid(x1, x2)
139     # exact
140     Z_esp = torch.empty_like(X1)
141     for i,x in enumerate(x1):
142         for j,y in enumerate(x2):
143             Z_esp[i,j] = u(x, y)
144     c = ax.contour(X1, X2, Z_esp, levels=10, colors='white')
145     ax.clabel(c)
146
147     X_plot = torch.cat((X1.reshape(-1,1),
148                        X2.reshape(-1,1)), dim=1)
149     Z_est = model(X_plot)
150     Z_est = Z_est.reshape((ns,ns))
151     cf = ax.contourf(X1, X2, Z_est.detach(), levels=10, cmap='coolwarm')
152     plt.colorbar(cf)
153
154     ax.grid()
155     ax.set_xlabel('$x_1$')
156     ax.set_ylabel('$x_2$')
157     plt.show()
158
159     # backward
160     optim.zero_grad()
161     loss.backward()
162     optim.step()
```

3.5.3 Exercícios

[[tag::construcao]]

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.