

Redes Neurais Artificiais

Pedro H A Konzen

28 de junho 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	25
3.1.3 Exercícios	27
3.2 Aplicação: Aproximação de Funções	27
3.2.1 Função unidimensional	27
3.2.2 Função bidimensional	30

<i>CONTEÚDO</i>	v
3.2.3 Exercícios	32
3.3 Aplicação: Equação de Laplace	32
3.3.1 Exercícios	37
Respostas dos Exercícios	38
Referências Bibliográficas	39

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.

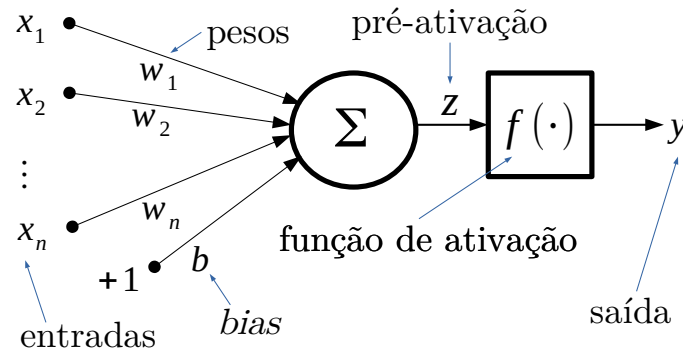


Figura 2.1: Esquema de um perceptron: unidade de processamento.

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.

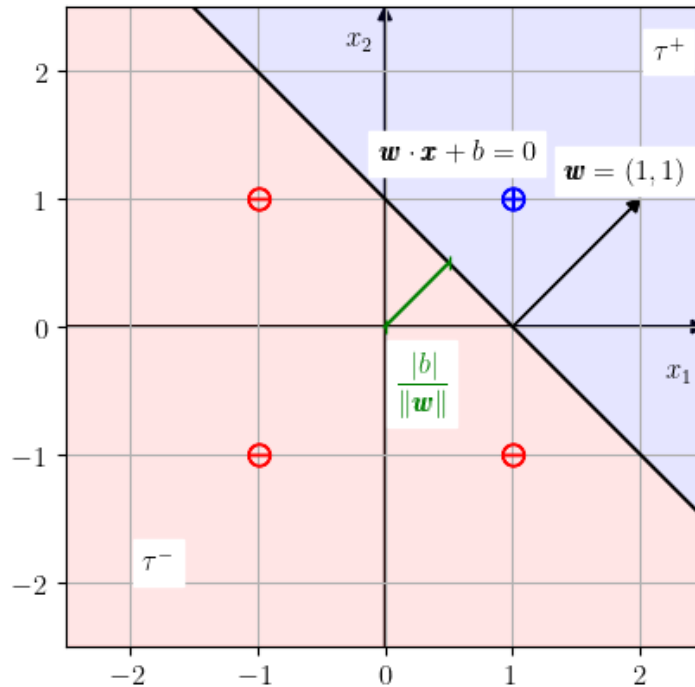


Figura 2.2: Interpretação geométrica do perceptron aplicado ao problema de classificação relacionado à operação lógica \wedge (e-lógico).

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
300     for s in range(ns):
44         y_est = model(X_train[s:s+1,:])
45         if (y_est*y_train[s] <= 0.):
46             with torch.no_grad():
47                 W += y_train[s]*X_train[s,:]
48                 b += y_train[s]
250                 not_updated = False
49
200     if (not_updated):
50         print('Training ended.')
51         break
150
52
53
54
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.

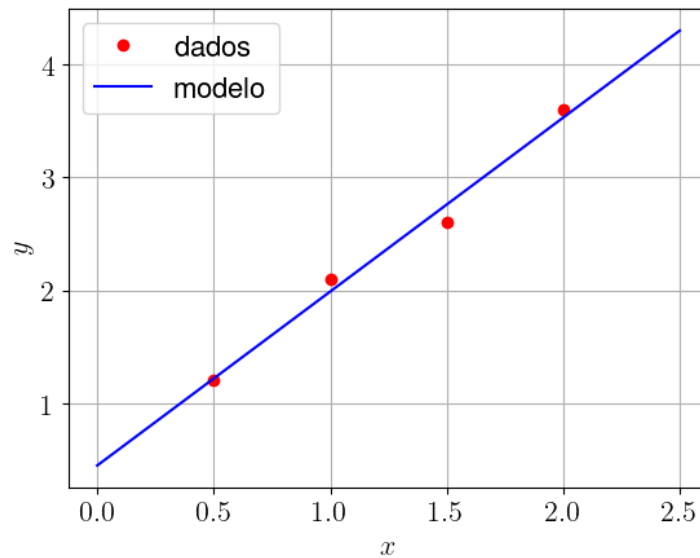


Figura 2.3: Interpretação geométrica do perceptron aplicado ao problema de regressão linear.

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

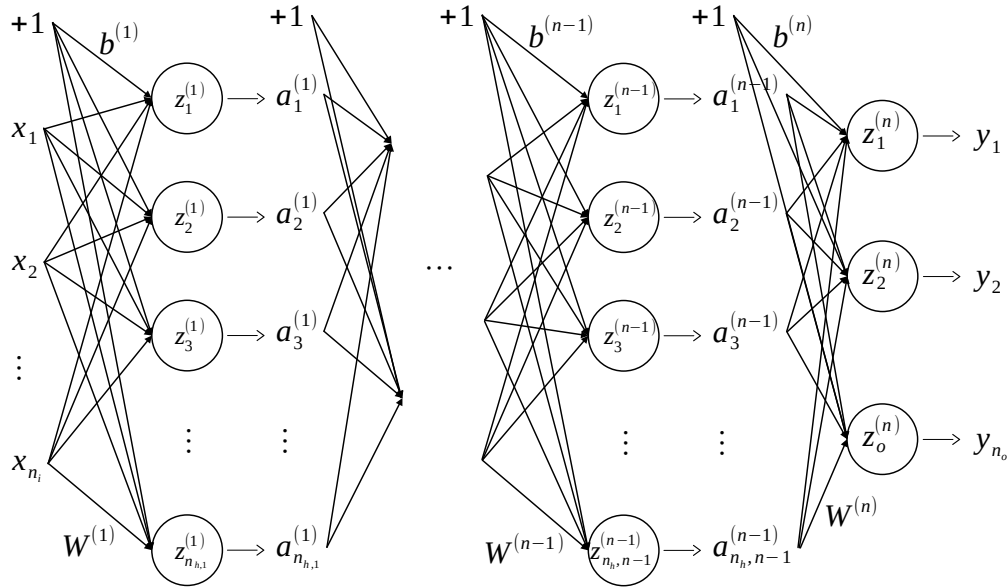


Figura 3.1: Estrutura de uma rede do tipo Perceptron Multicamadas (MLP).

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 tripa de pesos, bias 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 $\{\mathbf{x}^{(s)}\}_{s=1}^{n_s}$, com n_s amostras, o treinamento da rede consiste em resolver o problema de minimização

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

onde ε é uma dada função erro.

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 $e \leftarrow 1, \dots, n_e$:

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

onde, n_e é o **número de épocas**, l_r é uma dada **taxa de aprendizagem**¹ e o vetor direção \mathbf{d} depende dos gradientes

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

O cálculo dos gradientes pode ser feito **de trás para frente**, 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.

¹Em inglês, *learning rate*.

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 unidade de saída (função de ativação identidade).

Treinamento

Para o treinamento, vamos usar a função erro quadrático médio

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

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
56 # verificação
57 y = model(X_train)
58 print(f'y_est = {y}')
```

3.1.3 Exercícios

[[tag::construcao]]

3.2 Aplicação: Aproximação de Funções

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

3.2.1 Função unidimensional

Vamos criar uma MLP para aproximar a função gaussiana

$$y = e^{-x^2}, \quad (3.11)$$

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

Código 3.2: mlp_gaussiana_1d.py

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

```
38
39     # forward
40     y_est = model(X_train)
41
42     # erro
43     loss = loss_fun(y_est, y_train)
44
45     print(f'{epoch}: {loss.item():.4e}')
46
47     # critério de parada
48     if (loss.item() < tol):
49         break
50
51     # backward
52     optim.zero_grad()
53     loss.backward()
54     optim.step()
55
56
57 # verificação
58 fig = plt.figure()
59 ax = fig.add_subplot()
60
61 x = torch.linspace(a, b,
62                    steps=50).reshape(-1,1)
63
64 y_esp = fobj(x)
65 ax.plot(x, y_esp, label='fobj')
66
67 y_est = model(x)
68 ax.plot(x, y_est.detach(), label='model')
69
70 ax.legend()
71 ax.grid()
72 ax.set_xlabel('x')
73 ax.set_ylabel('y')
74 plt.show()
```

3.2.2 Função bidimensional

Vamos criar uma MLP para aproximar a função gaussiana

$$y = e^{-(x_1^2 + x_2^2)}, \quad (3.12)$$

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

Código 3.3: mlp_gaussiana_2d.py

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

```
33 ns = 100
34 ## num max épocas
35 nepochs = 5000
36 ## tolerância
37 tol = 1e-5
38
39 for epoch in range(nepochs):
40
41     # amostras
42     x0 = (a - b) * torch.rand(ns) + b
43     x1 = (a - b) * torch.rand(ns) + b
44     X0, X1 = torch.meshgrid(x0, x1)
45     X_train = torch.cat((X0.reshape(-1,1),
46                          X1.reshape(-1,1)),
47                          dim=1)
48     y_train = fobj(X_train)
49
50     # forward
51     y_est = model(X_train)
52
53     # erro
54     loss = loss_fun(y_est, y_train)
55
56     print(f'{epoch}: {loss.item():.4e}')
57
58     # critério de parada
59     if (loss.item() < tol):
60         break
61
62     # backward
63     optim.zero_grad()
64     loss.backward()
65     optim.step()
66
67
68     # verificação
69     fig = plt.figure()
70     ax = fig.add_subplot()
71
72     n = 50
```

```

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),
77               X1.reshape(-1,1)),
78               dim=1)
79
80 y_esp = fobj(X)
81 Y = y_esp.reshape((n,n))
82 levels = torch.linspace(0., 1., 10)
83 c = ax.contour(X0, X1, Y, levels=levels, colors='white')
84 ax.clabel(c)
85
86 y_est = model(X)
87 Y = y_est.reshape((n,n))
88 ax.contourf(X0, X1, Y.detach(), levels=levels)
89
90 ax.grid()
91 ax.set_xlabel('x_1')
92 ax.set_ylabel('x_2')
93 plt.show()

```

3.2.3 Exercícios

[[tag::construcao]]

3.3 Aplicação: Equação de Laplace

Vamos criar uma MLP para resolver

$$-\Delta u = f, \quad \mathbf{x} \in D = (-1, 1)^2, \quad (3.13a)$$

$$u = 0, \quad \mathbf{x} \in \partial D. \quad (3.13b)$$

Para validação, vamos considerar um problema com solução manufaturada

$$u(\mathbf{x}) = \sin(\pi x_1) \sin(\pi x_2) \quad (3.14)$$

o que nos fornece

$$f = \pi^2 \sin(\pi x_1) \sin(\pi x_2). \quad (3.15)$$

Código 3.4: mlp_eqlaplace.py

```
1 import torch
2 import matplotlib.pyplot as plt
3
4 # modelo
5
6 model = torch.nn.Sequential(
7     torch.nn.Linear(2,500),
8     torch.nn.Tanh(),
9     torch.nn.Linear(500,250),
10    torch.nn.Tanh(),
11    torch.nn.Linear(250,50),
12    torch.nn.Tanh(),
13    torch.nn.Linear(50,1)
14 )
15
16 # treinamento
17
18 ## fun obj
19 a = -1.
20 b = 1.
21 def exact(x):
22     y = torch.sin(torch.pi*x[:,0])*torch.sin(torch.pi*x[:,1])
23     return y.reshape(-1,1)
24
25 def rhs(x):
26     y = torch.pi**2*torch.sin(torch.pi*x[:,0])*torch.sin(torch.pi*x[:,1])
27     return y.reshape(-1,1)
28
29 ## otimizador
30 optim = torch.optim.SGD(model.parameters(),
31                           lr=1e-2, momentum=0.85)
32
33 ## scheduler
34 scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optim,
35                                                         factor = 0.6,
36                                                         min_lr = 1e-6,)
```

```
36
37 ## num de amostras pts internos
38 n_in = 200
39 ## num de amostras pts fronteira
40 n_bound = 50
41 ## num max épocas
42 nepochs = 5000
43 ## tolerância
44 tol = 1e-5
45 ## output freq
46 eout = 100
47
48 def loss_fun(X_in, X_bound, model=model):
49
50     ## pontos internos
51     n_in = X_in.size(0)
52     l_in = 0.
53     for s in range(n_in):
54         x = X_in[s:s+1,:].detach()
55         x.requires_grad = True
56         u = model(x)
57         grad_u = torch.autograd.grad(u, x,
58                                     create_graph = True,
59                                     retain_graph = True)[0]
60         u_x = grad_u[0,0]
61         u_y = grad_u[0,1]
62
63         u_xx = torch.autograd.grad(u_x, x,
64                                   create_graph = True,
65                                   retain_graph = True)[0][0,0]
66         u_yy = torch.autograd.grad(u_y, x,
67                                   create_graph = True,
68                                   retain_graph = True)[0][0,1]
69         l_in = torch.add(l_in, (u_xx + u_yy + rhs(x))**2)
70     l_in /= n_in
71
72
73     ## pontos de contorno
74     n_bound = X_bound.size(0)
75     l_bound = 0.
```

```
76     for s in range(n_bound):
650 77         x = X_bound[s:s+1,:]
78         u = model(x)
79         l_bound = torch.add(l_bound, u**2)
80     l_bound /= n_bound
600 81
82     return l_in + l_bound
83
84
550 85 # pts de fronteira
86 X_bound = torch.empty((4*n_bound, 2))
87 # pts internos
500 88 X_in = torch.empty((n_in, 2))
89
90 # épocas
450 91 for epoch in range(nepochs):
92
93     # amostras: pts internos
94     for s in range(n_in):
400 95         X_in[s,:] = (a-b)*torch.rand(2) + b
96     # amostras: pst fronteira
97     s = 0
98     for i in range(n_bound):
350 99         # a <= x0 <= b, x1 = 0
100         X_bound[s,0] = (a-b)*torch.rand(1) + b
101         X_bound[s,1] = a
102         s += 1
300 103
104         # x0 = b, a <= x1 <= b
105         X_bound[s,0] = b
106         X_bound[s,1] = (a-b)*torch.rand(1) + b
250 107         s += 1
108
109         # x0 = a, a <= x1 < b
200 110         X_bound[s,0] = a
111         X_bound[s,1] = (a-b)*torch.rand(1) + b
112         s += 1
113
150 114         # a <= x0 <= b, x1 = b
115         X_bound[s,0] = (a-b)*torch.rand(1) + b
```



```
116         X_bound[s,1] = b
117         s += 1
118
119     # erro
120     loss = loss_fun(X_in, X_bound)
121
122     print(f'{epoch}: loss = {loss.item():.4e}, lr = {optim.param_group
123 if ((epoch+1) % eout == 0):
124     # verificação
125     fig = plt.figure()
126     ax = fig.add_subplot()
127
128     ns = 50
129     x0 = torch.linspace(a, b, steps=ns)
130     x1 = torch.linspace(a, b, steps=ns)
131     X0, X1 = torch.meshgrid(x0, x1)
132     X = torch.cat((X0.reshape(-1,1),
133                   X1.reshape(-1,1)),
134                   dim=1)
135
136     y_esp = exact(X)
137     Y = y_esp.reshape((ns,ns))
138     c = ax.contour(X0, X1, Y, levels=10, colors='white')
139     ax.clabel(c)
140
141     y_est = model(X)
142     Y = y_est.reshape((ns,ns))
143     cf = ax.contourf(X0, X1, Y.detach(), levels=10, cmap='coolwarm
144     plt.colorbar(cf)
145
146     # amostras
147     ax.plot(X_in[:,0].detach(), X_in[:,1].detach(), ls='', marker=
148     ax.plot(X_bound[:,0].detach(), X_bound[:,1].detach(), ls='', m
149
150     ax.grid()
151     ax.set_xlabel('$x_1$')
152     ax.set_ylabel('$x_2$')
153     plt.show()
154
155
```

```
156     # critério de parada
650 157     if (loss.item() < tol):
158         break
159
160     # backward
600 161     optim.zero_grad()
162     loss.backward()
163     optim.step()
550 164     scheduler.step(loss)
```

3.3.1 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] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT Press, Cambridge, MA, 2016.