Redes Neurais Artificiais

Pedro H A Konzen

31 de julho de 2023

Licença

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

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 similiar 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 percep-*

tron), um modelo de progressão (em inglês, *feedfoward*) 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** $f : \mathbb{R} \to \mathbb{R}$ com a **pré-ativação**

$$z = \boldsymbol{w} \cdot \boldsymbol{x} + b \tag{2.1}$$
$$= w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b \tag{2.2}$$

onde, $\boldsymbol{x} \in \mathbb{R}^n$ é o **vetor de entrada**, $\boldsymbol{w} \in \mathbb{R}^n$ 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

$$y := \mathcal{N}(\boldsymbol{x}; (\boldsymbol{w}, b)) \tag{2.3}$$
$$= f(z) = f(\boldsymbol{w} \cdot \boldsymbol{x} + b) \tag{2.4}$$

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

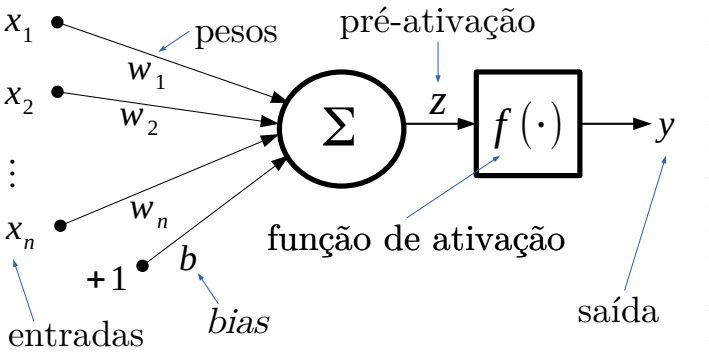

Figura 2.1: Esquema de um perceptron: unidade de processamento.

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 $\boldsymbol{x} \in \{-1,1\}^2$ e a função sinal

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

como função de ativação, i.e.

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

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

Pré-processamento

Uma vez que nosso modelo recebe valores $\boldsymbol{x}\in\{-1,1\}^2$ 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 $\boldsymbol{x}$. Isso consiste em um método para escolhermos os parâmetros $(\boldsymbol{w},b)$ que sejam adequados para esta tarefa. Vamos explorar mais sobre isso na sequência do texto e, aqui, apenas escolhemos

$$\boldsymbol{w} = [1,1] \tag{2.7}$$
$$b = -1 \tag{2.8}$$

Com isso, nosso perceptron é

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

Verifique que ele satisfaz a tabela verdade acima!

Implementação

Código 2.1: perceptron.py

```
1 import torch
```

```
2
3  # modelo
4  class Perceptron(torch.nn.Module):
5      def __init__(self):
6          super().__init__()
7          self.linear = torch.nn.Linear(2,1)
8
9      def forward(self, x):
10         z =  self.linear(x)
11         y = torch.sign(z)
12         return y
13
14 model = Perceptron()
15 W = torch.Tensor([[1., 1.]])
16 b = torch.Tensor([-1.])
17 with torch.no_grad():
18     model.linear.weight = torch.nn.Parameter(W)
19     model.linear.bias = torch.nn.Parameter(b)
20
21 # dados de entrada
22 X = torch.tensor([[1., 1.],
23                   [1., -1.],
24                   [-1., 1.],
25                   [-1., -1.]])
26
27 print(f"\nDados de entrada\n{X}")
28
29
30 # forward (aplicação do modelo)
31 y = model(X)
32
33 print(f"Valores estimados\n{y}")
```

Interpretação geométrica

Empregamos o seguinte modelo de neurônio

$$\mathcal{N}(\boldsymbol{x};(\boldsymbol{w},b)) = \text{sign}(w_1x_1 + w_2x_2 + b) \tag{2.10}$$

Observamos que

$$w_1 x_1 + w_2 x_2 + b = 0 \tag{2.11}$$

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

$$\tau^+ = \{\boldsymbol{x} \in \mathbb{R}^2 : w_1 x_1 + w_2 x_2 + b > 0\} \tag{2.12}$$
$$\tau^- = \{\boldsymbol{x} \in \mathbb{R}^2 : w_1 x_1 + w_2 x_2 + b < 0\} \tag{2.13}$$

O primeiro está na direção do vetor normal a reta $\boldsymbol{n} = (w_1, w_2)$ 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

$$w_1 x_1 + w_2 x_2 + b = 0 \tag{2.14}$$

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.

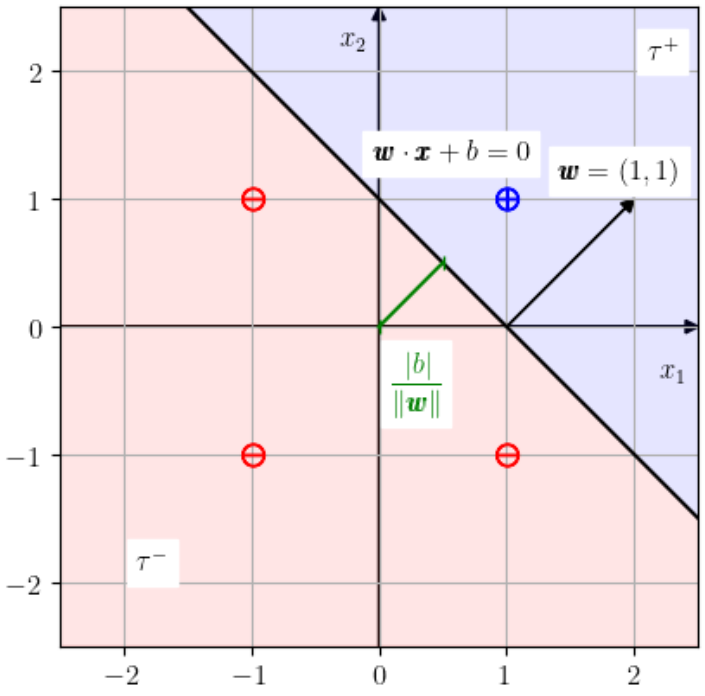

Figura 2.2: Interpretação geométrica do perceptron aplicado ao problema de classificação relacionado à operação lógica $\wedge$ (e-lógico).

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

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

onde, n_e é um dado número de épocas[1].

```
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
```

[1] Número de vezes que as amostrar 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
41  for epoch in range(nepochs):
42
43      # update
44      not_updated = True
45      for s in range(ns):
46          y_est = model(X_train[s:s+1,:])
47          if (y_est*y_train[s] <= 0.):
48              with torch.no_grad():
49                  W += y_train[s]*X_train[s,:]
50                  b += y_train[s]
51                  not_updated = False
52
53      if (not_updated):
54          print('Training ended.')
55          break
56
57
```

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

2.1.2 Problema de regressão

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

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

Modelo

Vamos determinar o perceptron[2]

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

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

Treinamento

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

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

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

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

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

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

Observamos que o problema (2.16) é equivalente a um problema linear de mínimos quadrados. A solução é obtida resolvendo-se a equação normal[3]

$$M^T M \boldsymbol{c} = M^T \boldsymbol{y}, \tag{2.19}$$

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

$$M = \begin{bmatrix} \boldsymbol{x} & \mathbf{1} \end{bmatrix} \tag{2.20}$$

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,
```

[3] Consulte o Exercício 2.1.4.

```
                          2.0]).reshape(-1,1)
y_train = torch.tensor([1.2,
                        2.1,
                        2.6,
                        3.6]).reshape(-1,1)

## número de amostras
ns = y_train.size(0)

print("\nDados de treinamento")
print("X_train =")
print(X_train)
print("y_train = ")
print(y_train)

# treinamento

## matriz
M = torch.cat((X_train,
               torch.ones((ns,1))), dim=1)
## solucão M.Q.
c = torch.linalg.lstsq(M, y_train)[0]
with torch.no_grad():
    W = c[0]
    b = c[1]

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

Resultado

Nosso perceptron corresponde ao modelo

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

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

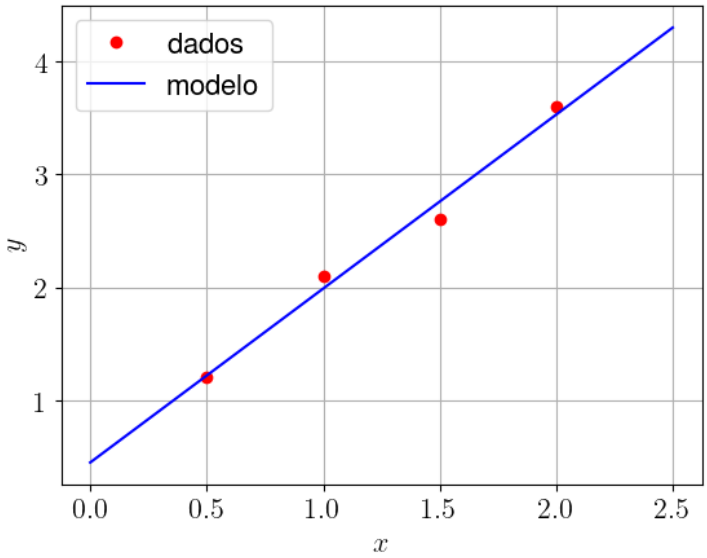

Figura 2.3: Interpretação geométrica do perceptron aplicado ao problema de regressão linear.

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 `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 gerais[4], que podem ser aplicados a ambos os problemas.

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

$$\tilde{y} = \mathcal{N}(\boldsymbol{x}; (\boldsymbol{w}, b)) = f(\underbrace{\boldsymbol{w} \cdot \boldsymbol{x} + b}_{z}), \tag{2.22}$$

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

$$z := \boldsymbol{w} \cdot \boldsymbol{x} + b \tag{2.23}$$

[4] Aqui, vamos explorar apenas algoritmos de treinamento supervisionado.

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

$$\varepsilon(\boldsymbol{w}, b) := \frac{1}{n_s}\sum_{s=1}^{n_s}\left(\tilde{y}^{(s)} - y^{(s)}\right)^2 \tag{2.24}$$

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

onde $\tilde{y}^{(s)} = \mathcal{N}\left(\boldsymbol{x}^{(s)}; (\boldsymbol{w}, b)\right)$ 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

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

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

$$\min_{(\boldsymbol{w},b)} \varepsilon(\boldsymbol{w}, b) \tag{2.27}$$

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, *Gradiente Descent Method*) é um método de declive. Aplicado ao nosso modelo de Perceptron consiste no seguinte algoritmo:

1. $(\boldsymbol{w}, b)$ aproximação inicial.
2. Para $e \leftarrow 1, \ldots, n_e$:
   (a) $(\boldsymbol{w}, b) \leftarrow (\boldsymbol{w}, b) - l_r \dfrac{\partial \varepsilon}{\partial(\boldsymbol{w}, b)}$

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

$$\frac{\partial \varepsilon}{\partial(\boldsymbol{w}, b)} := \left(\frac{\partial \varepsilon}{\partial w_1}, \ldots, \frac{\partial \varepsilon}{\partial w_{n_{in}}}, \frac{\partial \varepsilon}{\partial b}\right) \tag{2.28}$$

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

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

$$= \frac{1}{ns} \sum_{s=1}^{ns} \frac{\partial \varepsilon^{(s)}}{\partial \tilde{y}^{(s)}} \frac{\partial \tilde{y}^{(s)}}{\partial \boldsymbol{w}} \tag{2.30}$$

$$\frac{\partial \varepsilon}{\partial \boldsymbol{w}} = \frac{1}{ns} \sum_{s=1}^{ns} \frac{\partial \varepsilon^{(s)}}{\partial \tilde{y}^{(s)}} \frac{\partial \tilde{y}^{(s)}}{\partial z^{(s)}} \frac{\partial z^{(s)}}{\partial \boldsymbol{w}} \tag{2.31}$$

Observando que

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

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

$$\frac{\partial z^{(s)}}{\partial \boldsymbol{w}} = \boldsymbol{x}^{(s)} \tag{2.34}$$

obtemos

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

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

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

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

$$f(x) = \tanh(x). \tag{2.38}$$

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 ## optimizador
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 um 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 algoritmos consiste no seguinte:

1. $\mathbf{w}$, b aproximações inicial.

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

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

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

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 ## optimizador
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

$$f(x) = \tanh(x). \tag{2.40}$$

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.

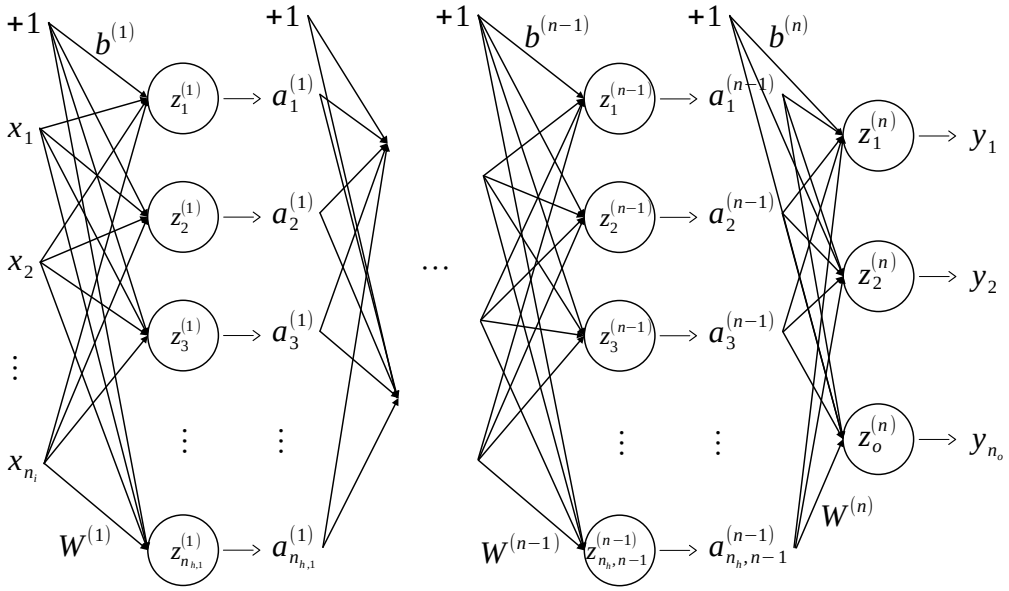

Figura 3.1: Estrutura de uma rede do tipo Perceptron Multicamadas (MLP).

Denotamos uma MLP de n camadas por

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

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

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

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

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

3.1.1 Treinamento

Fornecido um **conjunto de treinamento** $\{\boldsymbol{x}^{(s)}, \boldsymbol{y}^{(s)}\}_{s=1}^{n_s}$, com n_s amostras, o treinamento da rede consiste em resolver o problema de minimização

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

onde ε é uma dada **função erro** (em inglês, *loss function*) e $\tilde{\boldsymbol{y}}^{(s)}$, $\boldsymbol{y}^{(s)}$ 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. $W, \boldsymbol{b}$ aproximações iniciais.
2. Para $e \leftarrow 1, \ldots, n_e$:
   (a) $(W, \boldsymbol{b}) \leftarrow (W, \boldsymbol{b}) - l_r \boldsymbol{d}\left(\nabla_{W,\boldsymbol{b}}\varepsilon\right)$

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

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

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

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

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

Para os pesos da penúltima, temos

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

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

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

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 $R = A_1 \texttt{xor} A_2$. 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 $\boldsymbol{x} = (x_1, x_2)$ 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 $f^{(1)}(\boldsymbol{x}) = \tanh(\boldsymbol{x})$ e $f^{(2)}(\boldsymbol{x}) = id(\boldsymbol{x})$. 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*)

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

onde os valores estimados $\tilde{y}^{(s)} = \mathcal{N}\left(\boldsymbol{x}^{(s)}\right)$ e $\left\{\boldsymbol{x}^{(s)}, y^{(s)}\right\}_{s=1}^{n_s}$, $n_s = 4$, 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 ## optimizador
14 optim = torch.optim.SGD(model.parameters(), lr=1e-2)
15
16 ## função erro
17 loss_fun = torch.nn.MSELoss()
18
19 ## dados de treinamento
20 X_train = torch.tensor([[1., 1.],
21                         [1., -1.],
22                         [-1., 1.],
23                         [-1., -1.]])
24 y_train = torch.tensor([-1., 1., 1., -1.]).reshape(-1,1)
25
26 print("\nDados de treinamento")
27 print("X_train =")
28 print(X_train)
29 print("y_train = ")
30 print(y_train)
31
32 ## num max épocas
33 nepochs = 5000
34 tol = 1e-3
35
36 for epoch in range(nepochs):
37
38     # forward
39     y_est = model(X_train)
```

```
40
41      # erro
42      loss = loss_fun(y_est, y_train)
43
44      print(f'{epoch}: {loss.item():.4e}')
45
46      # critério de parada
47      if (loss.item() < tol):
48          break
49
50      # backward
51      optim.zero_grad()
52      loss.backward()
53      optim.step()
54
55
56  # verificação
57  y = model(X_train)
58  print(f'y_est = {y}')
```

3.1.3 Exercícios

[[tag::construcao]]

3.2 Aplicação: Problema de Classificação Binária

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()
```

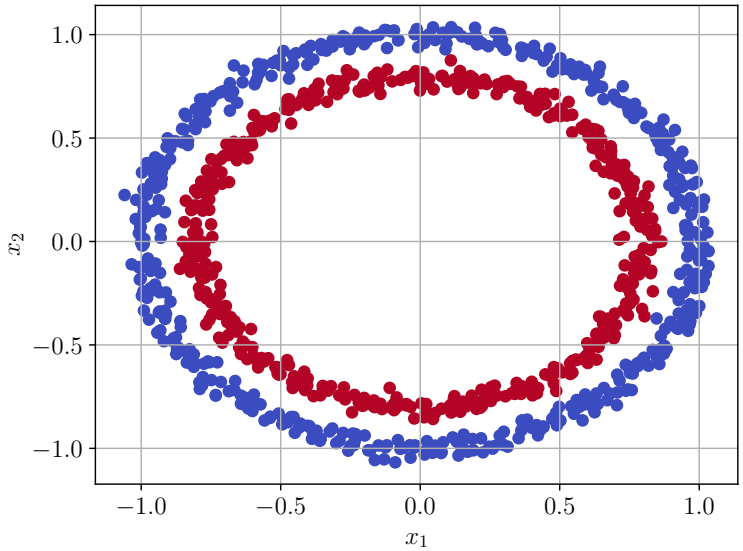

Figura 3.2: Dados para a o problema de classificação binária não linear.

3.2.2 Modelo

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

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

na camada escondida e

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

na saída da rede.

Para o treinamento e teste, vamos randomicamente separar os dados em um conjunto de treinamento $\{\boldsymbol{x}^{(k)}_{\text{train}}, y^{(k)}_{\text{train}}\}_{k=1}^{n_{\text{train}}}$ e um conjunto de teste $\{\boldsymbol{x}^{(k)}_{\text{test}}, y^{(k)}_{\text{test}}\}_{k=1}^{n_{\text{test}}}$, 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'\ttest: 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.

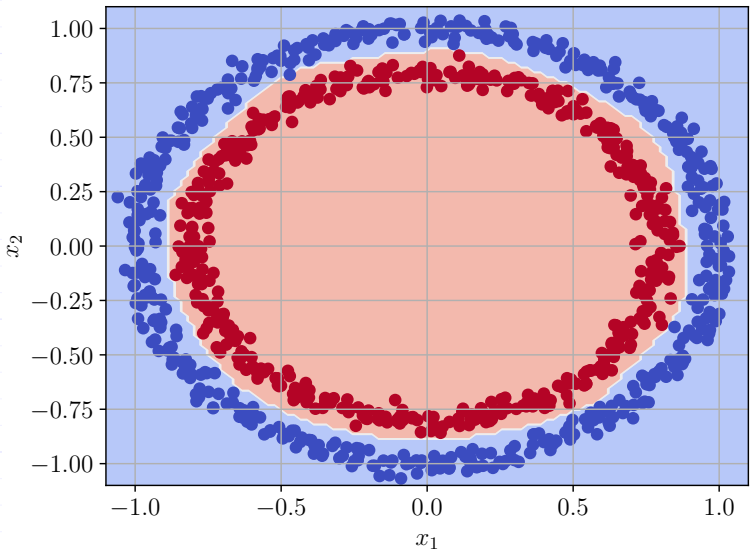

Figura 3.3: Verificação do modelo de classificação binária.

```
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):
```

```
    for j,yg in enumerate(xx):
        z = model(torch.tensor([[xg, yg]])).detach()
        Zg[i, j] = torch.round(z)

# visualização
fig = plt.figure()
ax = fig.add_subplot()
ax.contourf(Xg, Yg, Zg, levels=2, cmap=plt.cm.coolwarm, alpha=0.5)
ax.scatter(X[:,0], X[:,1], c=y, cmap=plt.cm.coolwarm)
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

$$y = e^{-x^2}, \tag{3.13}$$

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

```
import torch
import matplotlib.pyplot as plt

# modelo

model = torch.nn.Sequential(
    torch.nn.Linear(1,25),
    torch.nn.Tanh(),
    torch.nn.Linear(25,1)
    )

```

```
12 # treinamento
13
14 ## fun obj
15 fobj = lambda x: torch.exp(-x**2)
16 a = -1.
17 b = 1.
18
19 ## optimizador
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()
```

3.3.2 Função bidimensional

Vamos criar uma MLP para aproximar a função gaussiana

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

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

```
1  import torch
2  import matplotlib.pyplot as plt
3
4  # modelo
5
6  model = torch.nn.Sequential(
7      torch.nn.Linear(2,50),
8      torch.nn.Tanh(),
9      torch.nn.Linear(50,25),
```

```
10      torch.nn.Tanh(),
11      torch.nn.Linear(25,5),
12      torch.nn.Tanh(),
13      torch.nn.Linear(5,1)
14      )
15
16  # treinamento
17
18  ## fun obj
19  a = -1.
20  b = 1.
21  def fobj(x):
22      y = torch.exp(-x[:,0]**2 - x[:,1]**2)
23      return y.reshape(-1,1)
24
25  ## optimizador
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()
```

3.3.3 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 etapas[1]:

1. **Propagação (*forward*)**: os dados de entrada são propagados para todas as funções da rede, produzindo a saída estimada.
2. **Retropropagação (*backward*)**: a computação do gradiente do erro[2] 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 Perceptron[3]

$$\tilde{y} = \mathcal{N}(\boldsymbol{x}, (\boldsymbol{w}, b)) \tag{3.15a}$$
$$= f(\underbrace{\boldsymbol{w} \cdot \boldsymbol{x} + b}_{z}) \tag{3.15b}$$

[1] Para mais detalhes, consulte a Subseção 3.1.1.
[2] Medida da diferença entre o valor estimado e o valor esperado.
[3] Consulte o Capítulo 2 para mais informações sobre o Perceptron.

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

$$\frac{\partial \tilde{y}}{\partial \boldsymbol{x}} = \frac{\partial f}{\partial z}\frac{\partial z}{\partial \boldsymbol{x}} \tag{3.16a}$$

$$= f'(z)\boldsymbol{w} \tag{3.16b}$$

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

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

$$= wx + b \tag{3.17b}$$

que se ajusta ao conjunto de pontos[4]

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 MSE[5], espera-se que o Perceptron corresponda a reta de mínimos quadrados[6]

$$y = 1.54x + 0.45 \tag{3.18}$$

Portanto, espera-se que

$$\frac{\partial \tilde{y}}{\partial x} = 1.54. \tag{3.19}$$

Código 3.3: autograd_percep.py

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

[4]Consulte o Exercício 2.2.4.
[5]MSE, Erro Quadrático Médio.
[6]Para mais informações sobre essa aplicação, consulte a Subseção 2.1.2.

```
5
6  # treinamento
7
8  ## optimizador
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
46      print(f'{epoch}: {loss.item():.4e}, '\
47            + f'cstop = {cstop}/{nstop}')
48
49      if (cstop == nstop):
50          break
51
52      # backward
53      optim.zero_grad()
54      loss.backward()
55      optim.step()
56
57
58  # verificação
59  print(f'w = {model.weight}')
60  print(f'b = {model.bias}')
61
62  # autograd dy/dx
63
64  ## forward
65  x = torch.tensor([[1.]],
66                   requires_grad=True)
67  y = model(x)
68
69  ## backward
70  y.backward()
71  dydx = x.grad
72  print(f'dy/dx = {dydx}')
```

3.4.2 Autograd MLP

Os conceitos de diferenciação automática (**autograd**) são diretamente estendidos para redes do tipo Perceptron Multicamadas (MLP, do inglês, *Multilayer Perceptron*). No seguinte exemplo, exploramos o fato de MLPs serem aproximadoras universais e avaliamos a derivada de uma MLP na aproximação de uma função.

Exemplo 3.4.2. Vamos criar uma MLP

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

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

Código 3.4: autograd_fun1d.py

```
1  import torch
2  import matplotlib.pyplot as plt
3
4  # modelo
5
6  model = torch.nn.Sequential(
7      torch.nn.Linear(1,50),
8      torch.nn.Tanh(),
9      torch.nn.Linear(50,25),
10     torch.nn.Tanh(),
11     torch.nn.Linear(25,1)
12     )
13
14 # treinamento
15
16 ## fun obj
17 fobj = lambda x: torch.sin(torch.pi*x)
18 a = -1.
19 b = 1.
20
21 ## optimizador
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()
```

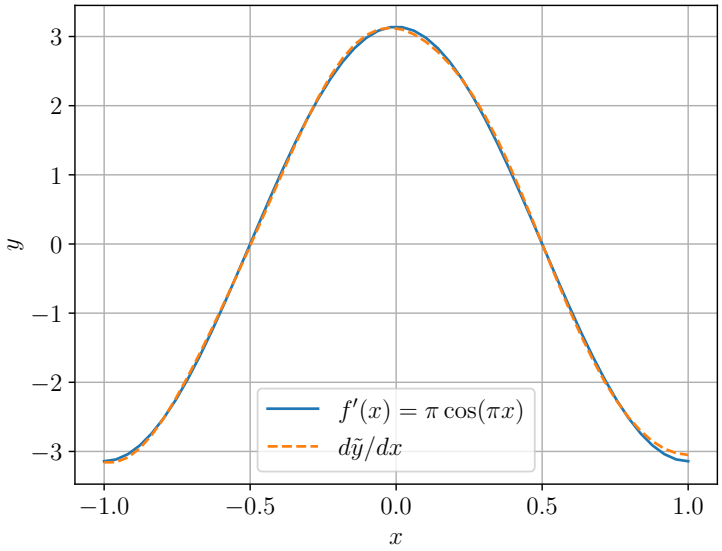

Figura 3.4: Comparação da autograd da MLP com a derivada exata $f'(x) = \pi\cos(\pi x)$ para o Exemplo 3.4.2.

Uma vez treinada, nossa MLP é uma aproximadora da função seno, i.e. $\tilde{y} \approx \text{sen}(\pi x)$. 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.

```
# verificação
fig = plt.figure()
ax = fig.add_subplot()

xx = torch.linspace(a, b,
                    steps=50).reshape(-1,1)
# y' = cos(x)
dy_esp = torch.pi*torch.cos(torch.pi*xx)
ax.plot(xx, dy_esp, label="$f'(x) = \\pi\cos(\\pi x)$")

# model autograd
dy_est = torch.empty_like(xx)
for i,x in enumerate(xx):
    x.requires_grad = True
```

```
    y = model(x)
    y.backward()
    dy_est[i] = x.grad
ax.plot(xx, dy_est, label='$d\\tilde{y}/dx$')

ax.legend()
ax.grid()
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
plt.show()
```

3.4.3 Exercícios

[[tag:construcao]]

3.5 Aplicação: Equação de Laplace

Vamos criar uma MLP para resolver

$$-\Delta u = 0, \quad \boldsymbol{x} \in D = (0,1)^2, \tag{3.21a}$$
$$u = 0, \quad \boldsymbol{x} \in \partial D. \tag{3.21b}$$

Como exemplo, vamos considerar um problema com solução manufaturada

$$u(\boldsymbol{x}) = x_1(1-x_1) - x_2(1-x_2). \tag{3.22}$$

3.5.1 Diferenças Finitas

Código 3.5: mlp_eqlaplace_df.py

```
import torch
import matplotlib.pyplot as plt
import random
import numpy as np

# modelo
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]
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):
53          s = n-1 + i*n
54          x = M[s,0]
55          y = M[s,1]
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
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
64      for j in range(1, n-1):
65          s = j + n*(n-1)
66          x = M[s,0]
67          y = M[s,1]
68          lc += (U[s,0] - uc(x,y))**2
69      lc *= p/nc
70
71      loss = lin + lc
72      return loss
73
74
75  # dados do problema
76
77  # collocation points
78  a = 1
79  n = 11
80  ns = n**2
81  h = 1./(n-1)
82  h2 = h**2
83
84  # malha
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):
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)
133 
134         X_plot = torch.cat((X1.reshape(-1,1),
135                             X2.reshape(-1,1)), dim=1)
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)
140 
141         ax.grid()
142         ax.set_xlabel('$x_1$')
143         ax.set_ylabel('$x_2$')
144         plt.show()
145 
146     # backward
147     optim.zero_grad()
148     loss.backward()
149     optim.step()
```

3.5.2 Autograd

Código 3.6: mlp_eqlaplace_ag.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          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
88  # dados do problema
89
90  # collocation points
91  a = 1
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]]

3.6 Aplicação: Equação do Calor

Consideramos o problema

$$u_t = u_{xx} + f, (t,x) \in (0,1] \times (-1,1), \tag{3.23a}$$

$$u(0,x) = \text{sen}(\pi x), x \in [-1,1], \tag{3.23b}$$
$$u(t,-1) = u(t,1) = 0, t \in (t_0, tf], \tag{3.23c}$$

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

$$u(t,x) = e^{-t}\,\text{sen}(\pi x). \tag{3.24}$$

3.6.1 Diferenças Finitas

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

$$u_t\left(t^{(k)}, x_i\right) \approx \frac{u_i^{(k)} - u_i^{(k-1)}}{h_t}, \tag{3.25}$$

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

$$u_{xx}\left(t^{(k)}, x_i\right) \approx \frac{u_{i-1}^{(k)} - 2u_i^{(k)} + u_{i+1}^{(k)}}{h_x^2}, \tag{3.26}$$

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

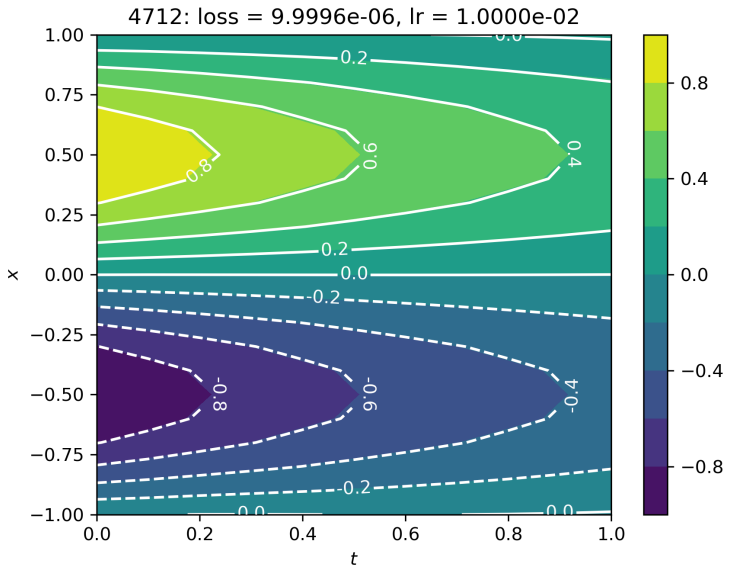

Figura 3.5: Soluções RNA (linhas brancas) *versus* analítica (cores de face) para o Problema 3.23.

Código 3.7: mlp_calor.py

```
1  import torch
2  from torch import pi, sin, exp
3  import matplotlib.pyplot as plt
4
5  # modelo
6  model = torch.nn.Sequential(
7      torch.nn.Linear(2,500),
8      torch.nn.ELU(),
9      torch.nn.Linear(500,500),
10     torch.nn.ELU(),
11     torch.nn.Linear(500,500),
12     torch.nn.ELU(),
13     torch.nn.Linear(500,1)
14 )
15
16 # otimizador
17 optim = torch.optim.SGD(model.parameters(),
```

```
18                          lr = 1e-2, momentum = 0.9)
19 scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optim)
20
21 # amostras
22 nt = 10
23 ht = 1./nt
24 tt = torch.linspace(0., 1., nt+1)
25 nx = 20
26 hx = 2./nx
27 xx = torch.linspace(-1., 1., nx+1)
28 T, X = torch.meshgrid(tt, xx,
29                        indexing='ij')
30 Uesp = torch.empty_like(T)
31 nsamples = (nt+1)*(nx+1)
32 M = torch.empty((nsamples, 2))
33 s = 0
34 for i,t in enumerate(tt):
35     for j,x in enumerate(xx):
36         Uesp[i,j] = exp(-t)*sin(pi*x)
37         M[s,0] = t
38         M[s,1] = x
39         s += 1
40
41 # treinamento
42 nepochs = 10001
43 tol = 1e-5
44 nout = 100
45
46 for epoch in range(nepochs):
47
48     # forward
49     Uest = model(M)
50
51     # loss
52     ## c.i.
53     lci = torch.tensor([0.])
54     for j,x in enumerate(xx):
55         s = j
56         assert(M[s,1] == x)
57         uesp = sin(pi*x)
```

```
58          lci += (Uest[s] - uesp)**2
59      ## pts internos
60      lin = torch.tensor([0.])
61      for i in range(1,nt+1):
62          for j in range(1,nx):
63              s = j + i*(nx+1)
64              # u_t
65              l = (Uest[s] - Uest[s-nx-1])/ht
66              # u_xx
67              l -= (Uest[s-1] - 2*Uest[s] + Uest[s+1])/hx**2
68              # f
69              l -= (pi**2 - 1.)*exp(-M[s,0])*sin(pi*M[s,1])
70              lin += l**2
71      ## c.c.
72      lcc = torch.tensor([0.])
73      for i,t in enumerate(tt[1:]):
74          # x = 0
75          s = i*(nx+1)
76          lcc += Uest[s]**2
77          # x = 1
78          s = nx + i*(nx+1)
79          lcc += Uest[s]**2
80
81      loss = (lci + lin + lcc)/nsamples
82
83      lr = optim.param_groups[-1]['lr']
84      print(f'{epoch}: loss = {loss.item():.4e}, lr = {lr:.4e}')
85
86      # output
87      if ((epoch % nout == 0) or (loss.item() < tol)):
88          plt.close()
89          fig = plt.figure(dpi=300)
90          ax = fig.add_subplot()
91          cb = ax.contourf(T, X, Uesp,
92                           levels=10)
93          fig.colorbar(cb)
94          cl = ax.contour(T, X, Uest.detach().reshape(nt+1,nx+1),
95                          levels=10, colors='white')
96          ax.clabel(cl, fmt='%.1f')
97          ax.set_xlabel('$t$')
```

```
98          ax.set_ylabel('$x$')
99          plt.title(f'{epoch}: loss = {loss.item():.4e}, lr = {lr:.4e}')
100         plt.savefig(f'./results/sol_{epoch:0>6}.png')
101
102     if (loss.item() < tol):
103         break
104
105     # backward
106     scheduler.step(loss)
107     optim.zero_grad()
108     loss.backward()
109     optim.step()
```

3.6.2 Diferenciação Autométicа

Código 3.8: mlp_calor_autograd.py

```
1  import torch
2  from torch import pi, sin, exp
3  from collections import OrderedDict
4  import matplotlib.pyplot as plt
5
6  # modelo
7  hidden = [50]*8
8  activation = torch.nn.Tanh()
9  layerList = [('layer_0', torch.nn.Linear(2, hidden[0])),
10              ('activation_0', activation)]
11 for l in range(len(hidden)-1):
12     layerList.append((f'layer_{l+1}',
13                       torch.nn.Linear(hidden[l], hidden[l+1])))
14     layerList.append((f'activation_{l+1}', activation))
15 layerList.append((f'layer_{len(hidden)}', torch.nn.Linear(hidden[-1],
16 #layerList.append((f'activation_{len(hidden)}', torch.nn.Sigmoid()))
17 layerDict = OrderedDict(layerList)
18 model = torch.nn.Sequential(OrderedDict(layerDict))
19
20 # otimizador
21 # optim = torch.optim.SGD(model.parameters(),
22 #                         lr = 1e-3, momentum=0.85)
23 optim = torch.optim.Adam(model.parameters(),
```

```
                            lr = 1e-2)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optim,
                                                       factor=0.1,
                                                       patience=100)

# treinamento
nt = 10
tt = torch.linspace(0., 1., nt+1)
nx = 20
xx = torch.linspace(-1., 1., nx+1)
T,X = torch.meshgrid(tt, xx, indexing='ij')
tt = tt.reshape(-1,1)
xx = xx.reshape(-1,1)

Sic = torch.hstack((torch.zeros_like(xx), xx))
Uic = sin(pi*xx)

Sbc0 = torch.hstack((tt[1:,:], -1.*torch.ones_like(tt[1:,:])))
Ubc0 = torch.zeros_like(tt[1:,:])

Sbc1 = torch.hstack((tt[1:,:], 1.*torch.ones_like(tt[1:,:])))
Ubc1 = torch.zeros_like(tt[1:,:])

tin = tt[1:,:]
xin = xx[1:-1,:]
Sin = torch.empty((nt*(nx-1), 2))
Fin = torch.empty((nt*(nx-1), 1))
s = 0
for i,t in enumerate(tin):
    for j,x in enumerate(xin):
        Sin[s,0] = t
        Sin[s,1] = x
        Fin[s,0] = (pi**2 - 1.)*exp(-t)*sin(pi*x)
        s += 1
tin = torch.tensor(Sin[:,0:1], requires_grad=True)
xin = torch.tensor(Sin[:,1:2], requires_grad=True)
Sin = torch.hstack((tin,xin))

nepochs = 50001
tol = 1e-4
```

```
64 nout = 100
65
66 for epoch in range(nepochs):
67
68     # loss
69
70     ## c.i.
71     Uest = model(Sic)
72     lic = torch.mean((Uest - Uic)**2)
73
74     ## residual
75     U = model(Sin)
76     U_t = torch.autograd.grad(
77         U, tin,
78         grad_outputs=torch.ones_like(U),
79         retain_graph=True,
80         create_graph=True)[0]
81     U_x = torch.autograd.grad(
82         U, xin,
83         grad_outputs=torch.ones_like(U),
84         retain_graph=True,
85         create_graph=True)[0]
86     U_xx = torch.autograd.grad(
87         U_x, xin,
88         grad_outputs=torch.ones_like(U_x),
89         retain_graph=True,
90         create_graph=True)[0]
91     res = U_t - U_xx - Fin
92     lin = torch.mean(res**2)
93
94     ## c.c. x = -1
95     Uest = model(Sbc0)
96     lbc0 = torch.mean(Uest**2)
97
98     ## c.c. x = 1
99     Uest = model(Sbc1)
100    lbc1 = torch.mean(Uest**2)
101
102    loss = lin + lic + lbc0 + lbc1
103
```

```
104     lr = optim.param_groups[-1]['lr']
105     print(f'{epoch}: loss = {loss.item():.4e}, lr = {lr:.4e}')
106
107     # backward
108     scheduler.step(loss)
109     optim.zero_grad()
110     loss.backward()
111     optim.step()
112
113
114     # output
115     if ((epoch % nout == 0) or (loss.item() < tol)):
116         plt.close()
117         fig = plt.figure(dpi=300)
118         nt = 10
119         tt = torch.linspace(0., 1., nt+1)
120         nx = 20
121         xx = torch.linspace(-1., 1., nx+1)
122         T,X = torch.meshgrid(tt, xx, indexing='ij')
123         Uesp = torch.empty_like(T)
124         M = torch.empty(((nt+1)*(nx+1),2))
125         s = 0
126         for i,t in enumerate(tt):
127             for j,x in enumerate(xx):
128                 Uesp[i,j] = exp(-t)*sin(pi*x)
129                 M[s,0] = t
130                 M[s,1] = x
131                 s += 1
132         Uest = model(M)
133         Uest = Uest.detach().reshape(nt+1,nx+1)
134         l2rel = torch.norm(Uest - Uesp)/torch.norm(Uesp)
135
136         ax = fig.add_subplot()
137         cb = ax.contourf(T, X, Uesp,
138                          levels=10)
139         fig.colorbar(cb)
140         cl = ax.contour(T, X, Uest,
141                         levels=10, colors='white')
142         ax.clabel(cl, fmt='%.1f')
143         ax.set_xlabel('$t$')
```

```
144             ax.set_ylabel('$x$')
145             plt.title(f'{epoch}: loss = {loss.item():.4e}, l2rel = {l2rel:
146             plt.savefig(f'./results/sol_{(epoch//nout):0>6}.png')
147
148         if ((loss.item() < tol) or (lr < 1e-6)):
149             break
```

Resposta dos Exercícios

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

Exercício 2.1.4. Dica: consulte a ligação Notas de Aula: Matemática Numérica: 7.1 Problemas lineares.

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

Bibliografia

[1] Goodfellow, I., Bengio, Y., Courville, A.. Deep learning, MIT Press, Cambridge, MA, 2016.

[2] Neural Networks: A Comprehensive Foundation, Haykin, S.. Pearson:Delhi, 2005. ISBN: 978-0020327615.

[3] Raissi, M., Perdikaris, P., Karniadakis, G.E.. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. Journal of Computational Physics 378 (2019), pp. 686-707. DOI: 10.1016/j.jcp.2018.10.045.

[4] Mata, F.F., Gijón, A., Molina-Solana, M., Gómez-Romero, J.. Physics-informed neural networks for data-driven simulation: Advantages, limitations, and opportunities. Physica A: Statistical Mechanics and its Applications 610 (2023), pp. 128415. DOI: 10.1016/j.physa.2022.128415.