Redes Neurais Artificiais

Pedro H A Konzen

20 de outubro 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

[[tag:construcao]]

2.1 Unidade de Processamento

[[tag:construcao]]

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**

$$\begin{align} z &= \boldsymbol{w} \cdot \boldsymbol{x} + b \tag{2.1}\\ &= w_1x_1 + w_2x_2 + \cdots + w_nx_n + b \tag{2.2}\end{align}$$

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

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

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.

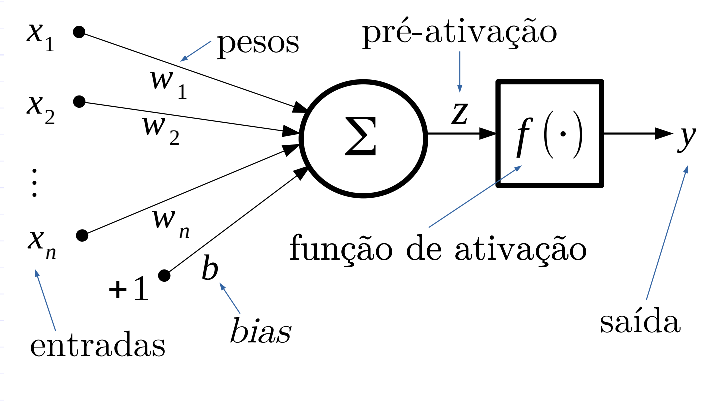

Figura 2.1: Esquema de um perceptron: unidade de processamento.

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

[[tag:construcao]]

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

[[tag:construcao]]

Nosso **modelo de neurônio** será um perceptron com duas entradas $\boldsymbol{x} \in \{-1,1\}^2$

e a função sinal

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

como função de ativação, i.e.

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

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

Pré-processamento

[[tag:construcao]]

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

[[tag:construcao]]

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

$$\boldsymbol{w} = [1, 1] \quad (2.7)$$
$$b = -1 \quad (2.8)$$

Com isso, nosso perceptron é

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

Verifique que ele satisfaz a tabela verdade acima!

Implementação

[[tag:construcao]]

Código 2.1: perceptron.py

```
import torch

# modelo
class Perceptron(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = torch.nn.Linear(2,1)

    def forward(self, x):
        z =  self.linear(x)
        y = torch.sign(z)
        return y

model = Perceptron()
W = torch.Tensor([[1., 1.]])
b = torch.Tensor([-1.])
with torch.no_grad():
    model.linear.weight = torch.nn.Parameter(W)
    model.linear.bias = torch.nn.Parameter(b)

# dados de entrada
X = torch.tensor([[1., 1.],
                  [1., -1.],
                  [-1., 1.],
                  [-1., -1.]])

print(f"\nDados de entrada\n{X}")


# forward (aplicação do modelo)
y = model(X)

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

Interpretação geométrica

[[tag:construcao]]

Empregamos o seguinte modelo de neurônio

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

Observamos que

$$w_1x_1 + w_2x_2 + b = 0 \tag{2.11}$$

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

$$\tau^+ = \{\boldsymbol{x} \in \mathbb{R}^2 : w_1x_1 + w_2x_2 + b > 0\} \tag{2.12}$$
$$\tau^- = \{\boldsymbol{x} \in \mathbb{R}^2 : w_1x_1 + w_2x_2 + b < 0\} \tag{2.13}$$

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

$$w_1x_1 + w_2x_2 + b = 0 \tag{2.14}$$

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.

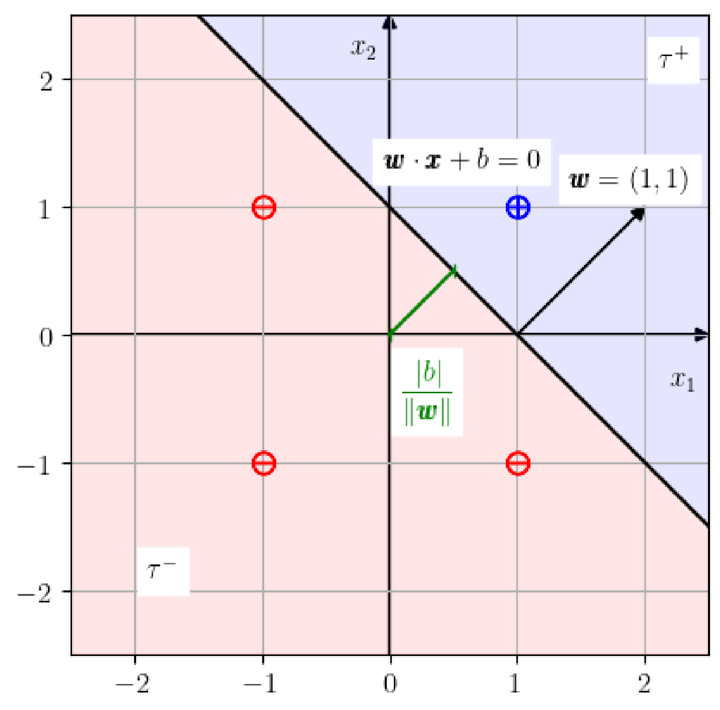

Figura 2.2: Interpretação geométrica do perceptron aplicado ao problema de classificação relacionado à operação lógica $\wedge$ (e-lógico).

Algoritmo de treinamento: perceptron

[[tag:construcao]]

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].

```
import torch

# modelo

class Perceptron(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = torch.nn.Linear(2,1)

    def forward(self, x):
        z =  self.linear(x)
        y = torch.sign(z)
        return y

model = Perceptron()
with torch.no_grad():
    W = model.linear.weight
    b = model.linear.bias

# dados de treinamento
X_train = torch.tensor([[1., 1.],
                        [1., -1.],
                        [-1., 1.],
                        [-1., -1.]])
y_train = torch.tensor([1., -1., -1., -1.]).reshape(-1,1)

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

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

[1] Número de vezes que as amostrar serão percorridas para realizar a correção dos pesos.

```python
print("y_train = ")
print(y_train)

# treinamento

## num max épocas
nepochs = 100

for epoch in range(nepochs):

    # update
    not_updated = True
    for s in range(ns):
        y_est = model(X_train[s:s+1,:])
        if (y_est*y_train[s] <= 0.):
            with torch.no_grad():
                W += y_train[s]*X_train[s,:]
                b += y_train[s]
                not_updated = False

    if (not_updated):
        print('Training ended.')
        break


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

2.1.2 Problema de regressão

[[tag:construcao]]

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

[[tag:construcao]]

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

[[tag:construcao]]

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

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

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

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

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

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

[3] Consulte o Exercício 2.1.4.

Implementação

[[tag:construcao]]

Código 2.2: perceptron_mq.py

```python
import torch

# modelo

class Perceptron(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = torch.nn.Linear(1,1)

    def forward(self, x):
        z =  self.linear(x)
        return z

model = Perceptron()
with torch.no_grad():
    W = model.linear.weight
    b = model.linear.bias

# dados de treinamento
X_train = torch.tensor([0.5,
                        1.0,
                        1.5,
                        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

[[tag:construcao]]

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.

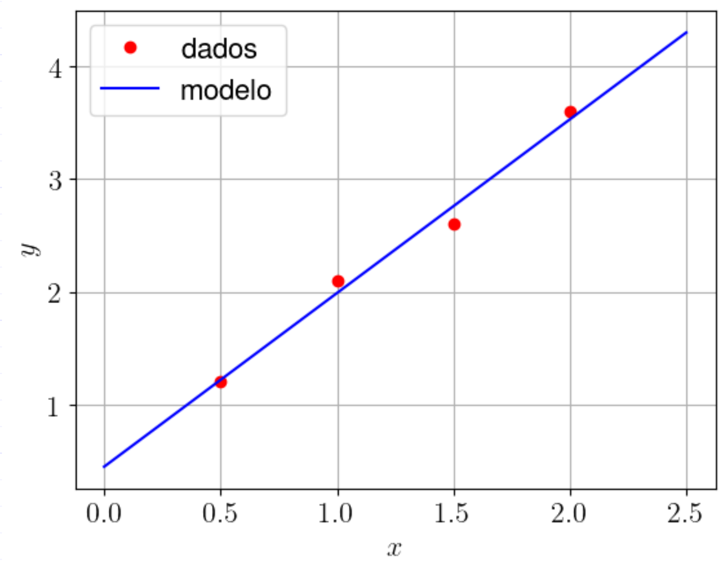

Figura 2.3: Interpretação geométrica do perceptron aplicado ao problema de regressão linear.

2.1.3 Exercícios

[[tag:construcao]]

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

[[tag:construcao]]

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.

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

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

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

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

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

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**

$$\begin{align}
\varepsilon(\boldsymbol{w}, b) &:= \frac{1}{n_s}\sum_{s=1}^{n_s}\left(\tilde{y}^{(s)} - y^{(s)}\right)^2 \tag{2.24}\\
&= \frac{1}{n_s}\sum_{s=1}^{n_s}\varepsilon^{(s)} \tag{2.25}
\end{align}$$

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

[[tag:construcao]]

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

[[tag:construcao]]

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

```python
import torch

# modelo

class Perceptron(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = torch.nn.Linear(2,1)

    def forward(self, x):
        z =  self.linear(x)
        y = torch.tanh(z)
        return y

model = Perceptron()

# treinamento

## optimizador
optim = torch.optim.SGD(model.parameters(), lr=1e-1)

## função erro
loss_fun = torch.nn.MSELoss()

## dados de treinamento
X_train = torch.tensor([[1., 1.],
                        [1., -1.],
                        [-1., 1.],
                        [-1., -1.]])
```

```
y_train = torch.tensor([1., -1., -1., -1.]).reshape(-1,1)

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

## num max épocas
nepochs = 5000
tol = 1e-3

for epoch in range(nepochs):

    # forward
    y_est = model(X_train)

    # erro
    loss = loss_fun(y_est, y_train)

    print(f'{epoch}: {loss.item():.4e}')

    # critério de parada
    if (loss.item() < tol):
        break

    # backward
    optim.zero_grad()
    loss.backward()
    optim.step()


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

2.2.2 Método do Gradiente Estocástico

[[tag:construcao]]

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. **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

[[tag:construcao]]

Código 2.4: perceptron_sgd.py

```python
import torch
import numpy as np

# modelo

class Perceptron(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = torch.nn.Linear(2,1)

    def forward(self, x):
        z =  self.linear(x)
        y = torch.tanh(z)
        return y

model = Perceptron()

# treinamento

## optimizador
```

```
optim = torch.optim.SGD(model.parameters(), lr=1e-1)

## função erro
loss_fun = torch.nn.MSELoss()

## dados de treinamento
X_train = torch.tensor([[1., 1.],
                        [1., -1.],
                        [-1., 1.],
                        [-1., -1.]])
y_train = torch.tensor([1., -1., -1., -1.]).reshape(-1,1)

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

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

## num max épocas
nepochs = 5000
tol = 1e-3

for epoch in range(nepochs):

    # forward
    y_est = model(X_train)

    # erro
    loss = loss_fun(y_est, y_train)

    print(f'{epoch}: {loss.item():.4e}')

    # critério de parada
    if (loss.item() < tol):
        break

    # backward
```

```
    for s in torch.randperm(ns):
        loss_s = (y_est[s,:] - y_train[s,:])**2
        optim.zero_grad()
        loss_s.backward()
        optim.step()
        y_est = model(X_train)


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

2.2.3 Exercícios

[[tag:construcao]]

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

[[tag:construcao]]

3.1 Modelo MLP

[[tag:construcao]]

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

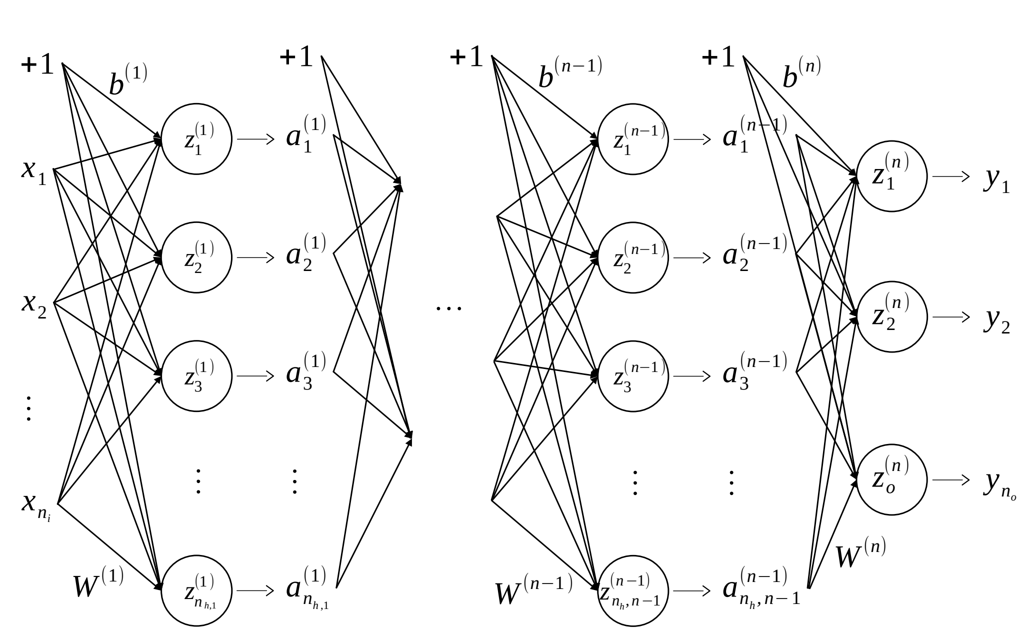

Figura 3.1: Estrutura de uma rede do tipo Perceptron Multicamadas (MLP).

Denotamos uma MLP de n camadas por

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

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, $l = 1, 2, \dots, n$.

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

3.1.1 Treinamento

[[tag:construcao]]

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

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

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

Para os pesos da penúltima, temos

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

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

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

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

[[tag:construcao]]

Vamos desenvolver uma MLP que faça a operação `xor` (ou exclusivo). I.e, receba como entrada dois valores lógicos A_1 e A_2 (V, verdadeiro ou F, falso) e forneça como saída o valor lógico $R = A_1 \texttt{xor} A_2$. 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

[[tag:construcao]]

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

[[tag:construcao]]

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

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

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

[[tag:construcao]]

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

Código 3.1: mlp_xor.py

```python
import torch

# modelo

model = torch.nn.Sequential(
    torch.nn.Linear(2,2),
    torch.nn.Tanh(),
    torch.nn.Linear(2,1)
    )

# treinamento

## optimizador
optim = torch.optim.SGD(model.parameters(), lr=1e-2)

## função erro
loss_fun = torch.nn.MSELoss()

## dados de treinamento
X_train = torch.tensor([[1., 1.],
                        [1., -1.],
                        [-1., 1.],
                        [-1., -1.]])
y_train = torch.tensor([-1., 1., 1., -1.]).reshape(-1,1)

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

```
print(y_train)

## num max épocas
nepochs = 5000
tol = 1e-3

for epoch in range(nepochs):

    # forward
    y_est = model(X_train)

    # erro
    loss = loss_fun(y_est, y_train)

    print(f'{epoch}: {loss.item():.4e}')

    # critério de parada
    if (loss.item() < tol):
        break

    # backward
    optim.zero_grad()
    loss.backward()
    optim.step()


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

3.1.3 Exercícios

[[tag::construcao]]

3.2 Aplicação: Problema de Classificação Binária

[[tag:construcao]]

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

3.2.1 Dados

[[tag:construcao]]

Vamos desenvolver uma rede do tipo Perceptron Multicamadas (MLP) para a classificação binária de pontos, com base nos seguintes dados.

```
from sklearn.datasets import make_circles
import matplotlib.pyplot as plt

plt.rcParams.update({
     "text.usetex": True,
     "font.family": "serif",
     "font.size": 14
     })

# data
print('data')
n_samples = 1000
print(f'n_samples = {n_samples}')
# X = points, y = labels
X, y = make_circles(n_samples,
                    noise=0.03, # add noise
                    random_state=42) # random seed

fig = plt.figure()
ax = fig.add_subplot()
ax.scatter(X[:,0], X[:,1], c=y, cmap=plt.cm.coolwarm)
ax.grid()
ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
plt.show()
```

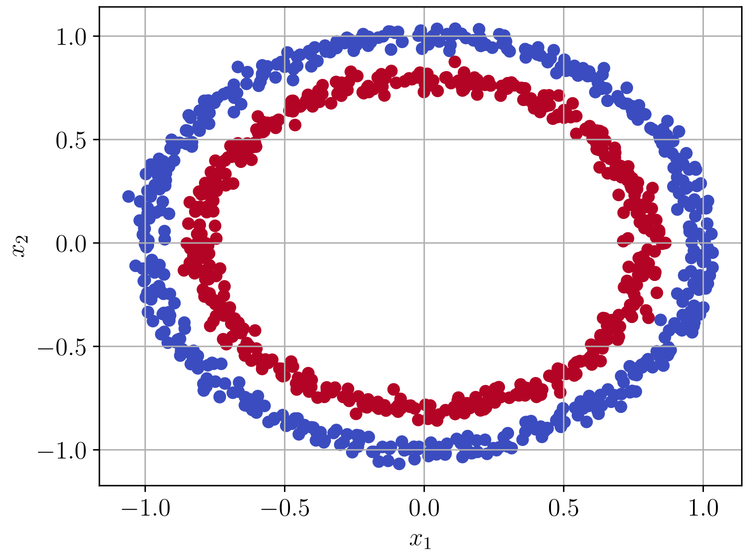

Figura 3.2: Dados para a o problema de classificação binária não linear.

3.2.2 Modelo

[[tag:construcao]]

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

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

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 $\{\boldsymbol{x}_{\text{train}}^{(k)}, y_{\text{train}}^{(k)}\}_{k=1}^{n_{\text{train}}}$ e um conjunto de teste $\{\boldsymbol{x}_{\text{test}}^{(k)}, y_{\text{test}}^{(k)}\}_{k=1}^{n_{\text{test}}}$, com $y = 0$ para os pontos azuis e $y = 1$ para os pontos vermelhos.

3.2.3 Treinamento e Teste

[[tag:construcao]]

Código 3.2: mlp_classbin.py

```
import torch
from sklearn.datasets import make_circles
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# data
print('data')
n_samples = 1000
print(f'n_samples = {n_samples}')
# X = points, y = labels
X, y = make_circles(n_samples,
                    noise=0.03, # add noise
                    random_state=42) # random seed

## numpy -> torch
X = torch.from_numpy(X).type(torch.float)
y = torch.from_numpy(y).type(torch.float).reshape(-1,1)

## split into train and test datasets
print('Data: train and test sets')
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.2,
                                                    random_state=42)
print(f'n_train = {len(X_train)}')
print(f'n_test = {len(X_test)}')
plt.close()
plt.scatter(X_train[:,0], X_train[:,1], c=y_train,
            marker='o', cmap=plt.cm.coolwarm, alpha=0.3)
plt.scatter(X_test[:,0], X_test[:,1], c=y_test,
            marker='*', cmap=plt.cm.coolwarm)
plt.show()

# model
model = torch.nn.Sequential(
```

```
    torch.nn.Linear(2, 10),
    torch.nn.ELU(),
    torch.nn.Linear(10, 1),
    torch.nn.Sigmoid()
    )

# loss fun
loss_fun = torch.nn.BCELoss()

# optimizer
optimizer = torch.optim.SGD(model.parameters(),
                            lr = 1e-1)

# evaluation metric
def accuracy_fun(y_pred, y_exp):
    correct = torch.eq(y_pred, y_exp).sum().item()
    acc = correct/len(y_exp) * 100
    return acc

# train
n_epochs = 10000
n_out = 100

for epoch in range(n_epochs):
    model.train()

    y_pred = model(X_train)

    loss = loss_fun(y_pred, y_train)

    acc = accuracy_fun(torch.round(y_pred),
                       y_train)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    model.eval()

    #testing
```

```
    if ((epoch+1) % n_out == 0):
        with torch.inference_mode():
            y_pred_test = model(X_test)
            loss_test = loss_fun(y_pred_test,
                                 y_test)
            acc_test = accuracy_fun(torch.round(y_pred_test),
                                    y_test)

        print(f'{epoch+1}: loss = {loss:.5e}, accuracy = {acc:.2f}%')
        print(f'\ttest: loss = {loss:.5e}, accuracy = {acc:.2f}%\n')
```

3.2.4 Verificação

[[tag:construcao]]

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

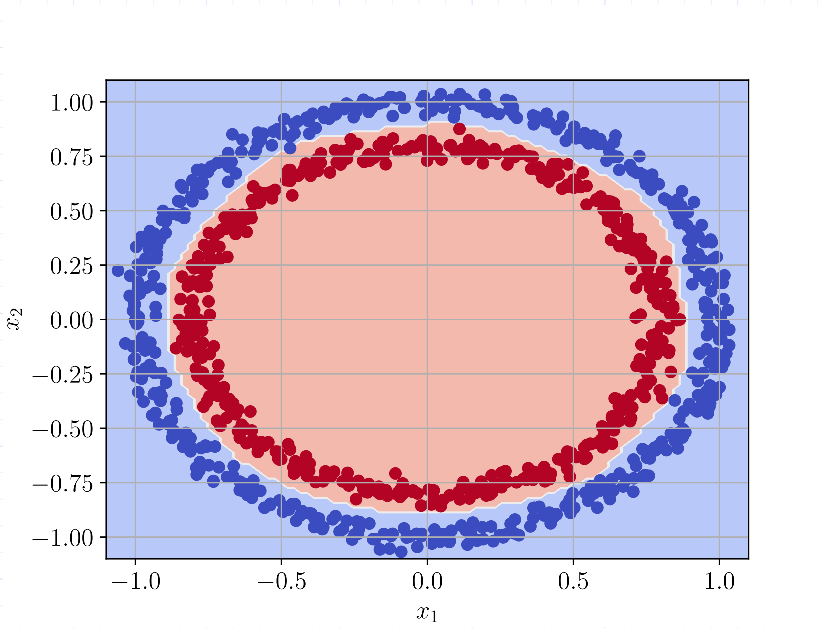

Figura 3.3: Verificação do modelo de classificação binária.

```
# malha de pontos
```

```
xx = torch.linspace(-1.1, 1.1, 100)
Xg, Yg = torch.meshgrid(xx, xx)

# valores estimados
Zg = torch.empty_like(Xg)
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

[[tag:construcao]]

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

3.3.1 Função unidimensional

[[tag:construcao]]

Vamos criar uma MLP para aproximar a função gaussiana

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

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)
    )

# treinamento

## fun obj
fobj = lambda x: torch.exp(-x**2)
a = -1.
b = 1.

## optimizador
optim = torch.optim.SGD(model.parameters(),
                        lr=1e-2, momentum=0.9)

## função erro
loss_fun = torch.nn.MSELoss()

## num de amostras por época
ns = 100
## num max épocas
nepochs = 5000
## tolerância
tol = 1e-5

for epoch in range(nepochs):

    # amostras
    X_train = (a - b) * torch.rand((ns,1)) + b
    y_train = fobj(X_train)

    # forward
    y_est = model(X_train)

    # erro
```

```
    loss = loss_fun(y_est, y_train)

    print(f'{epoch}: {loss.item():.4e}')

    # critério de parada
    if (loss.item() < tol):
        break

    # backward
    optim.zero_grad()
    loss.backward()
    optim.step()


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

x = torch.linspace(a, b,
                   steps=50).reshape(-1,1)

y_esp = fobj(x)
ax.plot(x, y_esp, label='fobj')

y_est = model(x)
ax.plot(x, y_est.detach(), label='model')

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

3.3.2 Função bidimensional

[[tag:construcao]]

Vamos criar uma MLP para aproximar a função gaussiana

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

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

```python
import torch
import matplotlib.pyplot as plt

# modelo

model = torch.nn.Sequential(
    torch.nn.Linear(2,50),
    torch.nn.Tanh(),
    torch.nn.Linear(50,25),
    torch.nn.Tanh(),
    torch.nn.Linear(25,5),
    torch.nn.Tanh(),
    torch.nn.Linear(5,1)
    )

# treinamento

## fun obj
a = -1.
b = 1.
def fobj(x):
    y = torch.exp(-x[:,0]**2 - x[:,1]**2)
    return y.reshape(-1,1)

## optimizador
optim = torch.optim.SGD(model.parameters(),
                        lr=1e-1, momentum=0.9)

## função erro
loss_fun = torch.nn.MSELoss()

## num de amostras por eixo por época
ns = 100
## num max épocas
nepochs = 5000
## tolerância
tol = 1e-5

for epoch in range(nepochs):
```

```

    # amostras
    x0 = (a - b) * torch.rand(ns) + b
    x1 = (a - b) * torch.rand(ns) + b
    X0, X1 = torch.meshgrid(x0, x1)
    X_train = torch.cat((X0.reshape(-1,1),
                         X1.reshape(-1,1)),
                        dim=1)
    y_train = fobj(X_train)

    # forward
    y_est = model(X_train)

    # erro
    loss = loss_fun(y_est, y_train)

    print(f'{epoch}: {loss.item():.4e}')

    # critério de parada
    if (loss.item() < tol):
        break

    # backward
    optim.zero_grad()
    loss.backward()
    optim.step()


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

n = 50
x0 = torch.linspace(a, b, steps=n)
x1 = torch.linspace(a, b, steps=n)
X0, X1 = torch.meshgrid(x0, x1)
X = torch.cat((X0.reshape(-1,1),
               X1.reshape(-1,1)),
              dim=1)

```

```
y_esp = fobj(X)
Y = y_esp.reshape((n,n))
levels = torch.linspace(0., 1., 10)
c = ax.contour(X0, X1, Y, levels=levels, colors='white')
ax.clabel(c)

y_est = model(X)
Y = y_est.reshape((n,n))
ax.contourf(X0, X1, Y.detach(), levels=levels)

ax.grid()
ax.set_xlabel('x_1')
ax.set_ylabel('x_2')
plt.show()
```

3.3.3 Exercícios

[[tag::construcao]]

3.4 Diferenciação Automática

[[tag:construcao]]

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

[1] Para mais detalhes, consulte a Subseção 3.1.1.
[2] Medida da diferença entre o valor estimado e o valor esperado.

RNAs, a computação do gradiente da saída da rede em relação a sua entrada pode ser feita de forma similar à computação do gradiente do erro em relação aos seus parâmetros.

3.4.1 Autograd Perceptron

[[tag:construcao]]

Para um Perceptron[3]

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

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]

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

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

Portanto, espera-se que

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

Código 3.3: autograd_percep.py

```python
import torch

# modelo
model = torch.nn.Linear(1,1)

# treinamento

## optimizador
optim = torch.optim.SGD(model.parameters(),
                        lr=1e-1)

## função erro
loss_fun = torch.nn.MSELoss()

## dados de treinamento
X_train = torch.tensor([[0.5],
                        [1.0],
                        [1.5],
                        [2.0]])
y_train = torch.tensor([[1.2],
                        [2.1],
                        [2.6],
                        [3.6]])

## num max épocas
nepochs = 5000
nstop = 10

cstop = 0
loss_min = torch.finfo().max
for epoch in range(nepochs):

    # forward
    y_est = model(X_train)
```

```

    # erro
    loss = loss_fun(y_est, y_train)

    # critério de parada
    if (loss.item() >= loss_min):
        cstop += 1
    else:
        loss_min = loss.item()
        cstop = 0

    print(f'{epoch}: {loss.item():.4e}, '\
          + f'cstop = {cstop}/{nstop}')

    if (cstop == nstop):
        break

    # backward
    optim.zero_grad()
    loss.backward()
    optim.step()


# verificação
print(f'w = {model.weight}')
print(f'b = {model.bias}')

# autograd dy/dx

## forward
x = torch.tensor([[1.]],
                 requires_grad=True)
y = model(x)

## backward
y.backward()
dydx = x.grad
print(f'dy/dx = {dydx}')
```

3.4.2 Autograd MLP

[[tag:construcao]]

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

Exemplo 3.4.2. 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 $y = \operatorname{sen}(\pi x)$ para $x \in [-1, 1]$

Código 3.4: autograd_fun1d.py

```python
import torch
import matplotlib.pyplot as plt

# modelo

model = torch.nn.Sequential(
    torch.nn.Linear(1,50),
    torch.nn.Tanh(),
    torch.nn.Linear(50,25),
    torch.nn.Tanh(),
    torch.nn.Linear(25,1)
    )

# treinamento

## fun obj
fobj = lambda x: torch.sin(torch.pi*x)
a = -1.
b = 1.

## optimizador
optim = torch.optim.SGD(model.parameters(),
                        lr=1e-1, momentum=0.9)

```

```
## função erro
loss_fun = torch.nn.MSELoss()

## num de amostras por época
ns = 100
## num max épocas
nepochs = 10000
## tolerância
tol = 1e-5

for epoch in range(nepochs):

    # amostras
    X_train = (a - b) * torch.rand((ns,1)) + b
    y_train = fobj(X_train)

    # forward
    y_est = model(X_train)

    # erro
    loss = loss_fun(y_est, y_train)

    lr = optim.param_groups[-1]['lr']
    print(f'{epoch}: loss = {loss.item():.4e}, lr = {lr:.4e}')

    # critério de parada
    if ((loss.item() < tol) or (lr <= 1e-7)):
        break

    # backward
    optim.zero_grad()
    loss.backward()
    optim.step()
```

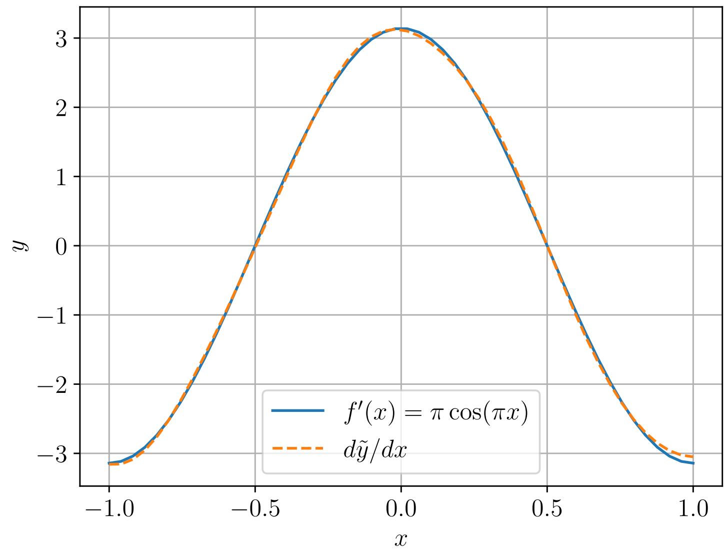

Figura 3.4: Comparação da autograd da MLP com a derivada exata $f'(x) = \pi\cos(\pi x)$ para o Exemplo 3.4.2.

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

Capítulo 4

Redes Informadas pela Física

[[tag:construcao]]

Redes neurais informadas pela física (PINNs, do inglês, *physics-informed neural networks*) são métodos de *deep learning* para a solução de equações diferenciais.

4.1 Problemas de Valores Iniciais

[[tag:construcao]]

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

$$y'(t) = g\left(t, y(t)\right), \ t_0 < t < t_f, \tag{4.1a}$$

$$y(t_0) = y_0, \tag{4.1b}$$

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

4.1.1 Euler PINN

[[tag:construcao]]

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

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

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

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

Nosso modelo PINN é uma perceptron multicamada (MLP)

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

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

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

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

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

Exemplo 4.1.1. Consideramos o seguinte IVP

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

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

Código 4.1: pyEulerPINN.py

```
import torch
from scipy.integrate import quad

# model
## num hidden layers
```

```
nh = 2
## num neurons per hidden layer
nn = 50
## activation fun in hidden layers
fh = torch.nn.Tanh()
## model architecture
model = torch.nn.Sequential()
model.add_module('layer_1', torch.nn.Linear(1,nn))
model.add_module('fun_1', fh)
for l in range(2, nh):
    model.add_module(f'layer_{l}', torch.nn.Linear(nn,nn))
    model.add_module(f'fun_{l}', fh)
model.add_module(f'layer_{nh}', torch.nn.Linear(nn,1))

# IVP params

## init time
t0 = 0.
## init condition
y0 = 0.5
## final time
tf = 1.

## num of time samples
ns = 10
## time step
ht = (tf - t0)/ns
## time samples
ts = torch.linspace(t0, tf, ns+1).reshape(-1,1)

## rhs
def g(t, y):
    return y + torch.sin(t)

# training
## num of epochs
nepochs = 10000
## output loss freq
eout = 100
## tolerance
```

```
tol = 1e-4
## early-stop
n_iter_no_change = 100

## optimizer
optim = torch.optim.Adam(model.parameters(), lr=1e-3)

# training loop
count_no_change = 0
best_loss = 1e38
for epoch in range(nepochs):

    # forward
    yy = model(ts)

    # loss
    ## t>0
    lup = torch.mean(((yy[1:] - yy[:-1])/ht \
                      - g(ts[:-1], yy[:-1]))**2)
    ## t = 0
    lic = (yy[0] - y0)**2

    loss_train = lup + lic

    # backward
    optim.zero_grad()
    loss_train.backward()
    optim.step()

    # validation
    ys = model(ts).detach()
    yv = torch.empty_like(ys)
    yv[0] = y0
    for s in range(1,ns+1):
        yv[s] = yv[s-1] + quad(lambda t: g(torch.tensor([[t]]),
                                    model(torch.tensor([[t]])).detach()),
                               ts[s-1], ts[s])[0]
    loss_valid = torch.mean((ys - yv)**2)

    if (loss_valid < best_loss):
```

```
            torch.save(model, 'model.pt')
            best_loss = loss_valid
            count_no_change = 0
        else:
            count_no_change += 1

        if ((epoch % eout == 0) or (count_no_change == 0)):
            msg = f'{epoch}: train = {loss_train.item():.4e}, valid = {los
            if (count_no_change == 0):
                msg += ' (best)'
            print(msg)

        if ((best_loss < tol) or (count_no_change > n_iter_no_change)):
            break

        if (loss_train < tol):
            break
```

4.1.2 AD-PINN

[[tag:construcao]]

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

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

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

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

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

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

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

Exemplo 4.1.2. Consideramos o seguinte IVP

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

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

Código 4.2: pyEulerPINN.py

```
import torch
from scipy.integrate import quad

# model
## num hidden layers
nh = 2
## num neurons per hidden layer
nn = 50
## activation fun in hidden layers
fh = torch.nn.Tanh()
## model architecture
model = torch.nn.Sequential()
model.add_module('layer_1', torch.nn.Linear(1,nn))
model.add_module('fun_1', fh)
for l in range(2, nh):
    model.add_module(f'layer_{l}', torch.nn.Linear(nn,nn))
    model.add_module(f'fun_{l}', fh)
model.add_module(f'layer_{nh}', torch.nn.Linear(nn,1))

# IVP params

## init time
t0 = 0.
## init condition
y0 = 0.5
## final time
tf = 1.

## num of time samples
ns = 10
## time step
ht = (tf - t0)/ns
## time samples
```

```python
ts = torch.linspace(t0, tf, ns+1).reshape(-1,1)

## rhs
def g(t, y):
    return y + torch.sin(t)

# training
## num of epochs
nepochs = 10000
## output loss freq
eout = 100
## tolerance
tol = 1e-4
## early-stop
n_iter_no_change = 100

## optimizer
optim = torch.optim.Adam(model.parameters(), lr=1e-3)

# training loop
count_no_change = 0
best_loss = 1e38
for epoch in range(nepochs):

    # forward
    yy = model(ts)

    # loss
    ## t>0
    lup = torch.mean(((yy[1:] - yy[:-1])/ht \
                      - g(ts[:-1], yy[:-1]))**2)
    ## t = 0
    lic = (yy[0] - y0)**2

    loss_train = lup + lic

    # backward
    optim.zero_grad()
    loss_train.backward()
    optim.step()
```

```

    # validation
    ys = model(ts).detach()
    yv = torch.empty_like(ys)
    yv[0] = y0
    for s in range(1,ns+1):
        yv[s] = yv[s-1] + quad(lambda t: g(torch.tensor([[t]]),
                                          model(torch.tensor([[t]])).detach()),
                               ts[s-1], ts[s])[0]
    loss_valid = torch.mean((ys - yv)**2)

    if (loss_valid < best_loss):
        torch.save(model, 'model.pt')
        best_loss = loss_valid
        count_no_change = 0
    else:
        count_no_change += 1

    if ((epoch % eout == 0) or (count_no_change == 0)):
        msg = f'{epoch}: train = {loss_train.item():.4e}, valid = {loss_vali
        if (count_no_change == 0):
            msg += ' (best)'
        print(msg)

    if ((best_loss < tol) or (count_no_change > n_iter_no_change)):
        break

    if (loss_train < tol):
        break
```

4.1.3 Exercícios

[[tag:construcao]]

4.2 Aplicação: Equação de Laplace

[[tag:construcao]]

Vamos criar uma MLP para resolver

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

$$u = u_{\text{bc}},\ \boldsymbol{x} \in \partial D, \tag{4.11b}$$

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

Como exemplo, vamos considerar um problema com solução manufaturada

$$u(\boldsymbol{x}) = x_1(1 - x_1) - x_2(1 - x_2). \tag{4.12}$$

Código 4.3: pyEqLaplace

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

# modelo
## n camadas escondidas
nh = 3
## n neurônios por camada
nn = 50
## fun de ativação
fh = torch.nn.Tanh()
## arquitetura
model = torch.nn.Sequential()
model.add_module('layer_1', torch.nn.Linear(2,nn))
model.add_module('fun_1', fh)
for layer in range(2, nh):
    model.add_module(f'layer_{layer}', torch.nn.Linear(nn,nn))
    model.add_module(f'fun_{layer}', fh)
model.add_module(f'layer_{nh}', torch.nn.Linear(nn,1))

# SGD - (Stochastic) Gradient Descent
optim = torch.optim.SGD(model.parameters(),
                        lr = 1e-2,
                        momentum = 0.9)
optim = torch.optim.Adam(model.parameters(),
                         lr = 1e-2)
```

```
# params treinamento
## n épocas
nepochs = 10001
## freq output loss
nout_loss = 100
## stop criterion
tol = 1e-4

## n amostras por eixo
ns = 101

lloss = []
for epoch in range(nepochs):

    # forward

    ## internal pts samples
    Xin = torch.rand((ns, 2), requires_grad=True)
    Uin = model(Xin)

    ## loss internal pts
    D1Uin = torch.autograd.grad(
        Uin, Xin,
        grad_outputs=torch.ones_like(Uin),
        retain_graph=True,
        create_graph=True)[0]
    D2Uin = torch.autograd.grad(
        D1Uin, Xin,
        grad_outputs=torch.ones_like(D1Uin),
        retain_graph=True,
        create_graph=True)[0]

    lin = torch.mean((D2Uin[:,0] + D2Uin[:,1])**2)

    ## bc 1
    xx = torch.rand((ns, 1))
    yy = torch.zeros((ns,1))
    Xbc1 = torch.hstack((xx, yy))
```

```
    Ubc1 = model(Xbc1)

    ## loss bc 1
    Uexp = xx*(1. - xx)
    lbc1 = torch.mean((Ubc1 - Uexp)**2)

    ## bc 3
    xx = torch.rand((ns, 1))
    yy = torch.ones((ns,1))
    Xbc3 = torch.hstack((xx, yy))
    Ubc3 = model(Xbc3)

    ## loss bc 3
    Uexp = xx*(1. - xx)
    lbc3 = torch.mean((Ubc3 - Uexp)**2)

    ## bc 2
    xx = torch.ones((ns, 1))
    yy = torch.rand((ns,1))
    Xbc2 = torch.hstack((xx, yy))
    Ubc2 = model(Xbc2)

    ## loss bc 2
    Uexp = yy*(yy - 1.)
    lbc2 = torch.mean((Ubc2 - Uexp)**2)

    ## bc 4
    xx = torch.zeros((ns, 1))
    yy = torch.rand((ns,1))
    Xbc4 = torch.hstack((xx, yy))
    Ubc4 = model(Xbc4)

    ## loss bc 3
    Uexp = yy*(yy - 1.)
    lbc4 = torch.mean((Ubc4 - Uexp)**2)

    # loss function
    loss = lin + lbc1 + lbc2 + lbc3 + lbc4

    lloss.append(loss.item())
```

```

    if (((epoch % nout_loss) == 0) or (loss.item() < tol)):
        print(f'{epoch}: loss = {loss.item():.4e}')

        # gráfico
        fig = plt.figure()
        ax = fig.add_subplot()

        npts = 50
        xx = torch.linspace(0., 1., npts)
        yy = torch.linspace(0., 1., npts)
        X, Y = torch.meshgrid(xx, yy)
        # exact
        Uexp = X*(1. - X) - Y*(1. - Y)
        c = ax.contour(X, Y, Uexp, levels=10, colors='white')
        ax.clabel(c)

        M = torch.hstack((X.reshape(-1,1),
                          Y.reshape(-1,1)))
        Uest = model(M).detach()
        Uest = Uest.reshape((npts, npts))
        cf = ax.contourf(X, Y, Uest, levels=10, cmap='coolwarm')
        plt.colorbar(cf)

        ax.grid()
        ax.set_xlabel('$x$')
        ax.set_ylabel('$y$')
        plt.title(f"epoch = {epoch}, loss = {loss.item():.4e}")
        plt.savefig(f'results/sol_{epoch:0>6}.png', bbox_inches='tight')
        plt.close()

    if (loss.item() < tol):
        break

    # backward
    optim.zero_grad()
    loss.backward()
    optim.step()
```

```
fig = plt.figure()
ax = fig.add_subplot()
ax.plot(lloss)
ax.set_yscale('log')
plt.show()
```

4.2.1 Preprocessamento

[[tag:construcao]]

Vamos assumir as seguintes mudanças de variáveis

$$\bar{x} = 2x - 1 \tag{4.13a}$$

$$\bar{y} = 2y - 1. \tag{4.13b}$$

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

Então, segue que

$$\begin{aligned} \frac{\partial \bar{u}}{\partial \bar{x}} &= \frac{\partial}{\partial \bar{x}} u\left(\bar{\boldsymbol{x}}(\boldsymbol{x})\right) \\ &= \frac{\partial u}{\partial x}\frac{\partial x}{\partial \bar{x}} \\ &= \frac{1}{2}\frac{\partial u}{\partial x}. \end{aligned} \tag{4.14}$$

Também, temos

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

Analogamente, temos

$$\frac{\partial \bar{u}}{\partial \bar{y}} = \frac{1}{2}\frac{\partial u}{\partial y} \tag{4.16}$$

e

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

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

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

$$\bar{u} = \bar{u}_0,\ \bar{\boldsymbol{x}} \in \Gamma = \partial D. \tag{4.18b}$$

Código 4.4: pyEqLaplacePP

Exemplo 4.2.1.

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

# modelo
## n camadas escondidas
nh = 3
## n neurônios por camada
nn = 50
## fun de ativação
fh = torch.nn.Tanh()
## arquitetura
model = torch.nn.Sequential()
model.add_module('layer_1', torch.nn.Linear(2,nn))
model.add_module('fun_1', fh)
for layer in range(2, nh):
    model.add_module(f'layer_{layer}', torch.nn.Linear(nn,nn))
    model.add_module(f'fun_{layer}', fh)
model.add_module(f'layer_{nh}', torch.nn.Linear(nn,1))

# SGD - (Stochastic) Gradient Descent
optim = torch.optim.SGD(model.parameters(),
                        lr = 1e-2,
                        momentum = 0.9)
optim = torch.optim.Adam(model.parameters(),
                         lr = 1e-2)
```

```

# params treinamento
## n épocas
nepochs = 10001
## freq output loss
nout_loss = 100
## stop criterion
tol = 1e-4

## n amostras por eixo
ns = 101

lloss = []
for epoch in range(nepochs):

    # forward

    ## internal pts samples
    Xin = 2.*torch.rand((ns, 2)) -1.
    Xin.requires_grad=True
    Uin = model(Xin)

    ## loss internal pts
    D1Uin = torch.autograd.grad(
        Uin, Xin,
        grad_outputs=torch.ones_like(Uin),
        retain_graph=True,
        create_graph=True)[0]
    D2Uin = torch.autograd.grad(
        D1Uin, Xin,
        grad_outputs=torch.ones_like(D1Uin),
        retain_graph=True,
        create_graph=True)[0]

    lin = torch.mean((D2Uin[:,0] + D2Uin[:,1])**2)

    ## bc 1
    xx = 2.*torch.rand((ns, 1)) - 1.
    yy = -1.*torch.ones((ns,1))
    Xbc1 = torch.hstack((xx, yy))
```

```
    Ubc1 = model(Xbc1)

    ## loss bc 1
    xx = (xx + 1.)/2.;
    Uexp = xx*(1. - xx)
    lbc1 = torch.mean((Ubc1 - Uexp)**2)

    ## bc 3
    xx = 2.*torch.rand((ns, 1)) -1.
    yy = torch.ones((ns,1))
    Xbc3 = torch.hstack((xx, yy))
    Ubc3 = model(Xbc3)

    ## loss bc 3
    xx = (xx + 1.)/2.;
    Uexp = xx*(1. - xx)
    lbc3 = torch.mean((Ubc3 - Uexp)**2)

    ## bc 2
    xx = torch.ones((ns, 1))
    yy = 2.*torch.rand((ns,1)) -1.
    Xbc2 = torch.hstack((xx, yy))
    Ubc2 = model(Xbc2)

    ## loss bc 2
    yy = (yy + 1.)/2.;
    Uexp = yy*(yy - 1.)
    lbc2 = torch.mean((Ubc2 - Uexp)**2)

    ## bc 4
    xx = -1.*torch.ones((ns, 1))
    yy = 2.*torch.rand((ns,1)) -1.
    Xbc4 = torch.hstack((xx, yy))
    Ubc4 = model(Xbc4)

    ## loss bc 3
    yy = (yy + 1.)/2.;
    Uexp = yy*(yy - 1.)
    lbc4 = torch.mean((Ubc4 - Uexp)**2)

```

```
    # loss function
    loss = lin + lbc1 + lbc2 + lbc3 + lbc4

    lloss.append(loss.item())

    if (((epoch % nout_loss) == 0) or (loss.item() < tol)):
        print(f'{epoch}: loss = {loss.item():.4e}')

        # gráfico
        fig = plt.figure()
        ax = fig.add_subplot()

        npts = 50
        xx = torch.linspace(-1., 1., npts)
        yy = torch.linspace(-1., 1., npts)
        X, Y = torch.meshgrid(xx, yy)
        # exact
        Uexp = (X+1.)/2.*(1. - (X+1.)/2.) \
            - (Y+1.)/2.*(1. - (Y+1.)/2.)
        c = ax.contour(X, Y, Uexp, levels=10, colors='white')
        ax.clabel(c)

        M = torch.hstack((X.reshape(-1,1),
                          Y.reshape(-1,1)))
        Uest = model(M).detach()
        Uest = Uest.reshape((npts, npts))
        cf = ax.contourf(X, Y, Uest, levels=10, cmap='coolwarm')
        plt.colorbar(cf)

        ax.grid()
        ax.set_xlabel('$\\bar{x}$')
        ax.set_ylabel('$\\bar{y}$')
        plt.title(f"epoch = {epoch}, loss = {loss.item():.4e}")
        plt.savefig(f'results/sol_{epoch:0>6}.png', bbox_inches='tight
        plt.close()

    if (loss.item() < tol):
        break

    # backward
```

```
    optim.zero_grad()
    loss.backward()
    optim.step()

fig = plt.figure()
ax = fig.add_subplot()
ax.plot(lloss)
ax.set_yscale('log')
plt.show()
```

4.2.2 Exercícios

[[tag::construcao]]

4.3 Aplicação: Equação do Calor

[[tag:construcao]]

Consideramos o problema

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

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

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

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

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

4.3.1 Diferenças Finitas

[[tag:construcao]]

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

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

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

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

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

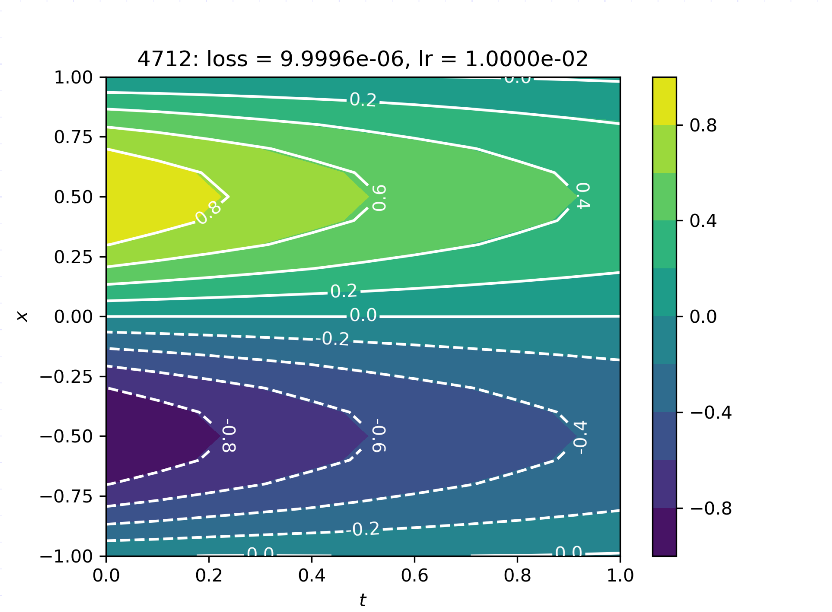

Figura 4.1: Soluções RNA (linhas brancas) *versus* analítica (cores de face) para o Problema 4.19.

Código 4.5: mlp_calor.py

```
import torch
from torch import pi, sin, exp
import matplotlib.pyplot as plt

# modelo
model = torch.nn.Sequential(
    torch.nn.Linear(2,500),
    torch.nn.ELU(),
    torch.nn.Linear(500,500),
    torch.nn.ELU(),
```

```
    torch.nn.Linear(500,500),
    torch.nn.ELU(),
    torch.nn.Linear(500,1)
)

# otimizador
optim = torch.optim.SGD(model.parameters(),
                        lr = 1e-2, momentum = 0.9)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optim)

# amostras
nt = 10
ht = 1./nt
tt = torch.linspace(0., 1., nt+1)
nx = 20
hx = 2./nx
xx = torch.linspace(-1., 1., nx+1)
T, X = torch.meshgrid(tt, xx,
                      indexing='ij')
Uesp = torch.empty_like(T)
nsamples = (nt+1)*(nx+1)
M = torch.empty((nsamples, 2))
s = 0
for i,t in enumerate(tt):
    for j,x in enumerate(xx):
        Uesp[i,j] = exp(-t)*sin(pi*x)
        M[s,0] = t
        M[s,1] = x
        s += 1

# treinamento
nepochs = 10001
tol = 1e-5
nout = 100

for epoch in range(nepochs):

    # forward
    Uest = model(M)

```

```
    # loss
    ## c.i.
    lci = torch.tensor([0.])
    for j,x in enumerate(xx):
        s = j
        assert(M[s,1] == x)
        uesp = sin(pi*x)
        lci += (Uest[s] - uesp)**2
    ## pts internos
    lin = torch.tensor([0.])
    for i in range(1,nt+1):
        for j in range(1,nx):
            s = j + i*(nx+1)
            # u_t
            l = (Uest[s] - Uest[s-nx-1])/ht
            # u_xx
            l -= (Uest[s-1] - 2*Uest[s] + Uest[s+1])/hx**2
            # f
            l -= (pi**2 - 1.)*exp(-M[s,0])*sin(pi*M[s,1])
            lin += l**2
    ## c.c.
    lcc = torch.tensor([0.])
    for i,t in enumerate(tt[1:]):
        # x = 0
        s = i*(nx+1)
        lcc += Uest[s]**2
        # x = 1
        s = nx + i*(nx+1)
        lcc += Uest[s]**2

    loss = (lci + lin + lcc)/nsamples

    lr = optim.param_groups[-1]['lr']
    print(f'{epoch}: loss = {loss.item():.4e}, lr = {lr:.4e}')

    # output
    if ((epoch % nout == 0) or (loss.item() < tol)):
        plt.close()
        fig = plt.figure(dpi=300)
        ax = fig.add_subplot()
```

```
        cb = ax.contourf(T, X, Uesp,
                         levels=10)
        fig.colorbar(cb)
        cl = ax.contour(T, X, Uest.detach().reshape(nt+1,nx+1),
                        levels=10, colors='white')
        ax.clabel(cl, fmt='%.1f')
        ax.set_xlabel('$t$')
        ax.set_ylabel('$x$')
        plt.title(f'{epoch}: loss = {loss.item():.4e}, lr = {lr:.4e}')
        plt.savefig(f'./results/sol_{epoch:0>6}.png')

    if (loss.item() < tol):
        break

    # backward
    scheduler.step(loss)
    optim.zero_grad()
    loss.backward()
    optim.step()
```

4.3.2 Diferenciação Automética

[[tag:construcao]]

Código 4.6: mlp_calor_autograd.py

```
import torch
from torch import pi, sin, exp
from collections import OrderedDict
import matplotlib.pyplot as plt

# modelo
hidden = [50]*8
activation = torch.nn.Tanh()
layerList = [('layer_0', torch.nn.Linear(2, hidden[0])),
             ('activation_0', activation)]
for l in range(len(hidden)-1):
    layerList.append((f'layer_{l+1}',
                      torch.nn.Linear(hidden[l], hidden[l+1])))
    layerList.append((f'activation_{l+1}', activation))
layerList.append((f'layer_{len(hidden)}', torch.nn.Linear(hidden[-1], 1)))
```

```
#layerList.append((f'activation_{len(hidden)}', torch.nn.Sigmoid()))
layerDict = OrderedDict(layerList)
model = torch.nn.Sequential(OrderedDict(layerDict))

# otimizador
# optim = torch.optim.SGD(model.parameters(),
#                          lr = 1e-3, momentum=0.85)
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
nout = 100

for epoch in range(nepochs):

    # loss

    ## c.i.
    Uest = model(Sic)
    lic = torch.mean((Uest - Uic)**2)

    ## residual
    U = model(Sin)
    U_t = torch.autograd.grad(
        U, tin,
        grad_outputs=torch.ones_like(U),
        retain_graph=True,
        create_graph=True)[0]
    U_x = torch.autograd.grad(
        U, xin,
        grad_outputs=torch.ones_like(U),
        retain_graph=True,
        create_graph=True)[0]
    U_xx = torch.autograd.grad(
        U_x, xin,
        grad_outputs=torch.ones_like(U_x),
        retain_graph=True,
        create_graph=True)[0]
    res = U_t - U_xx - Fin
    lin = torch.mean(res**2)

    ## c.c. x = -1
    Uest = model(Sbc0)
```

```
    lbc0 = torch.mean(Uest**2)

    ## c.c. x = 1
    Uest = model(Sbc1)
    lbc1 = torch.mean(Uest**2)

    loss = lin + lic + lbc0 + lbc1

    lr = optim.param_groups[-1]['lr']
    print(f'{epoch}: loss = {loss.item():.4e}, lr = {lr:.4e}')

    # backward
    scheduler.step(loss)
    optim.zero_grad()
    loss.backward()
    optim.step()


    # output
    if ((epoch % nout == 0) or (loss.item() < tol)):
        plt.close()
        fig = plt.figure(dpi=300)
        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')
        Uesp = torch.empty_like(T)
        M = torch.empty(((nt+1)*(nx+1),2))
        s = 0
        for i,t in enumerate(tt):
            for j,x in enumerate(xx):
                Uesp[i,j] = exp(-t)*sin(pi*x)
                M[s,0] = t
                M[s,1] = x
                s += 1
        Uest = model(M)
        Uest = Uest.detach().reshape(nt+1,nx+1)
        l2rel = torch.norm(Uest - Uesp)/torch.norm(Uesp)

```

```
            ax = fig.add_subplot()
            cb = ax.contourf(T, X, Uesp,
                             levels=10)
            fig.colorbar(cb)
            cl = ax.contour(T, X, Uest,
                            levels=10, colors='white')
            ax.clabel(cl, fmt='%.1f')
            ax.set_xlabel('$t$')
            ax.set_ylabel('$x$')
            plt.title(f'{epoch}: loss = {loss.item():.4e}, l2rel = {l2rel:.4e}')
            plt.savefig(f'./results/sol_{(epoch//nout):0>6}.png')

        if ((loss.item() < tol) or (lr < 1e-6)):
            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.