Redes Neurais Artificiais

Pedro H A Konzen

29 de junho 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**

$$\begin{aligned} z &= \boldsymbol{w} \cdot \boldsymbol{x} + b && (2.1)\\ &= w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b && (2.2) \end{aligned}$$

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

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

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.

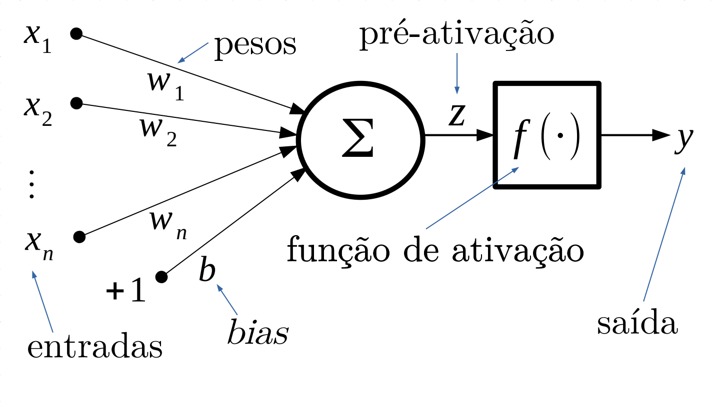

Figura 2.1: Esquema de um perceptron: unidade de processamento.

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.

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

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

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

Verifique que ele satisfaz a tabela verdade acima!

Implementação

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

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.

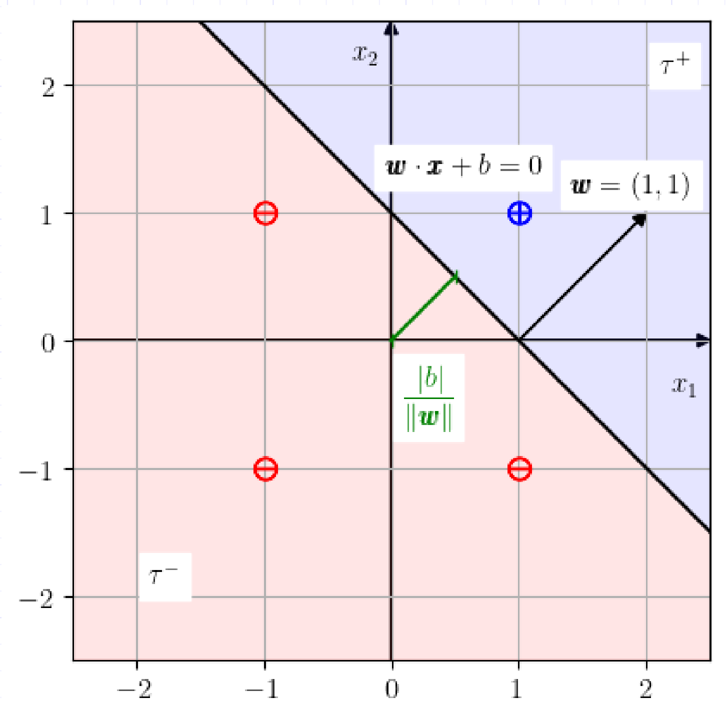

Figura 2.2: Interpretação geométrica do perceptron aplicado ao problema de classificação relacionado à operação lógica $\wedge$ (e-lógico).

Algoritmo de treinamento: perceptron

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].

```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.sign(z)
        return y

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

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

```
    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)
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
```

```python
# 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

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

```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,
```

[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.

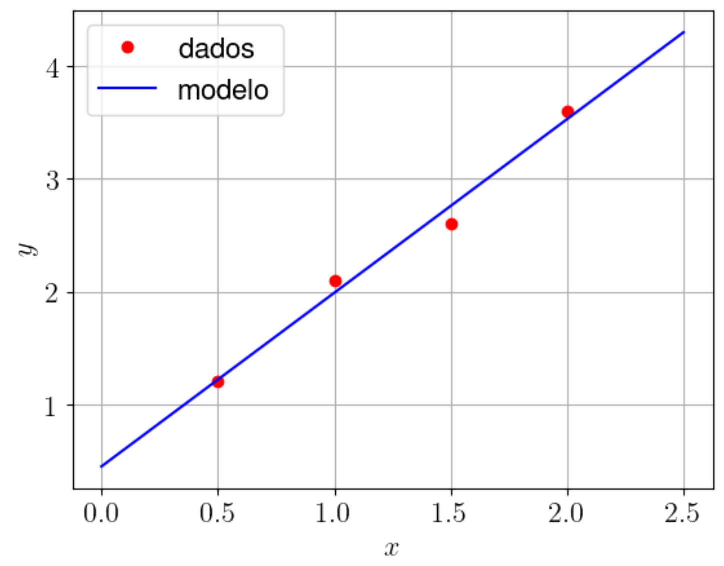

Figura 2.3: Interpretação geométrica do perceptron aplicado ao problema de regressão linear.

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 $f(x) = \operatorname{sign}(x)$ 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

```
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
```

```python
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

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

```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)

```

```python
## 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

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.

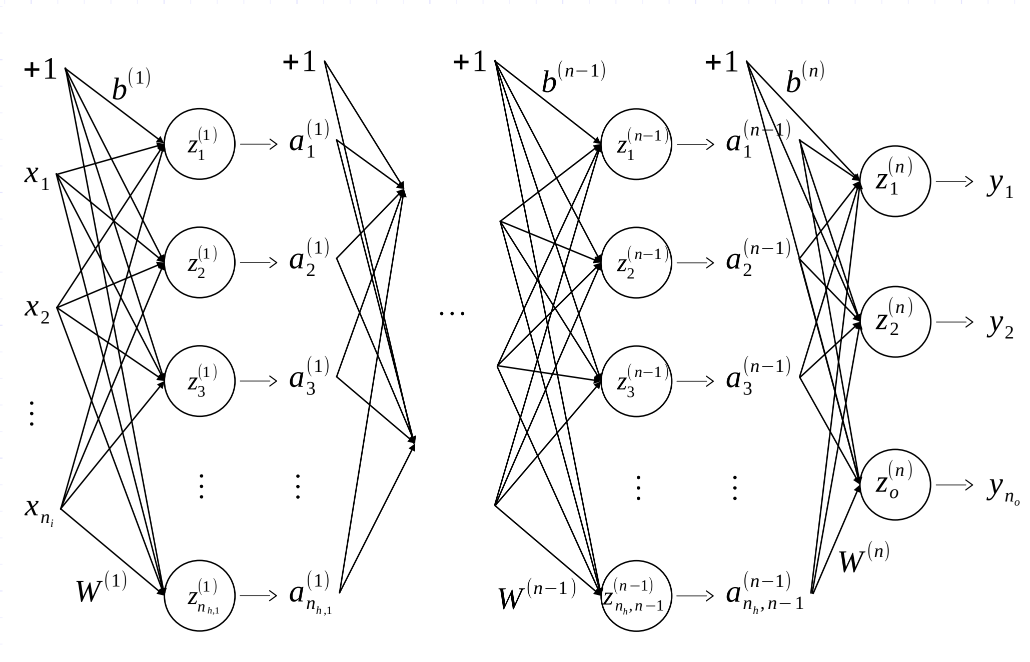

Figura 3.1: Estrutura de uma rede do tipo Perceptron Multicamadas (MLP).

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

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

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

$$\begin{aligned} \frac{\partial \varepsilon}{\partial W^{(n)}} &= \frac{\partial \varepsilon}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{z}^{(n)}} \frac{\partial \boldsymbol{z}^{(n)}}{\partial W^{(n)}}, &(3.5)\\ &= \frac{\partial \varepsilon}{\partial \boldsymbol{y}} f'\left(W^{(n)} \boldsymbol{a}^{(n-1)} + \boldsymbol{b}^{(n)}\right) \boldsymbol{a}^{(n-1)}. &(3.6) \end{aligned}$$

Para os pesos da penúltima, temos

$$\begin{aligned} \frac{\partial \varepsilon}{\partial W^{(n-1)}} &= \frac{\partial \varepsilon}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{z}^{(n)}} \frac{\partial \boldsymbol{z}^{(n)}}{\partial W^{(n-1)}}, &(3.7)\\ &= \frac{\partial \varepsilon}{\partial \boldsymbol{y}} f'\left(\boldsymbol{z}^{(n)}\right) \frac{\partial \boldsymbol{z}^{(n)}}{\partial \boldsymbol{a}^{(n-1)}} \frac{\partial \boldsymbol{a}^{(n-1)}}{\partial \boldsymbol{z}^{(n-1)}} \frac{\partial \boldsymbol{z}^{(n-1)}}{\partial W^{(n-1)}} &(3.8)\\ &= \frac{\partial \varepsilon}{\partial \boldsymbol{y}} f'\left(\boldsymbol{z}^{(n)}\right) W^{(n)} f'\left(\boldsymbol{z}^{(n-1)}\right) \boldsymbol{a}^{(n-2)} &(3.9) \end{aligned}$$

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

```
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)
```

```python

    # 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}')
```

3.2 Aplicação: Aproximação de Funções

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

3.2.1 Função unidimensional

Vamos criar uma MLP para aproximar a função gaussiana

$$y = e^{-x^2}, \tag{3.11}$$

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

```python
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()
```

3.2.2 Função bidimensional

Vamos criar uma MLP para aproximar a função gaussiana

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

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

```
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 Aplicação: Equação de Laplace

Vamos criar uma MLP para resolver

$$-\Delta u = 0, \quad \boldsymbol{x} \in D = (0,1)^2, \tag{3.13a}$$

$$u = 0, \quad \boldsymbol{x} \in \partial D. \tag{3.13b}$$

Como exemplo, vamos considerar um problema com solução manufaturada

$$u(\boldsymbol{x}) = x_1(1 - x_1) - x_2(1 - x_2). \tag{3.14}$$

3.3.1 Diferenças Finitas

Código 3.2: mlp_eqlaplace_df.py

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

# modelo
model = torch.nn.Sequential(
    torch.nn.Linear(2,50),
    torch.nn.Tanh(),
    torch.nn.Linear(50,10),
    torch.nn.Tanh(),
    torch.nn.Linear(10,5),
    torch.nn.Tanh(),
    torch.nn.Linear(5,1)
```

```
)

# SGD - (Stochastic) Gradient Descent
optim = torch.optim.SGD(model.parameters(),
                        lr = 1e-3,
                        momentum = 0.9,
                        dampening = 0.)

# Solução esperada
def u(x, y):
    return a*x*(1-x) - a*y*(1-y)


def laplace_loss(X, U, h2, n, uc=u, p=1.):
    # num de amostras
    nc = 2*n + 2*(n-2)
    ni = n**2 - nc

    # loss interno
    lin = 0.
    for i in range(1,n-1):
      for j in range(1,n-1):
        s = j + i*n
        l = (U[s-n, 0] - 2 * U[s, 0] + U[s+n, 0])/h2 # x
        l += (U[s-1, 0] - 2 * U[s, 0] + U[s+1, 0])/h2 # y
        lin += l**2
    lin /= ni

    # loss contorno
    lc = 0.
    # 0 <= x <= 1 e y == 0
    for i in range(n):
        s = i*n
        x = M[s,0]
        y = M[s,1]
        lc += (U[s,0] - uc(x,y))**2
    # 0 <= x <= 1 e y == 1
    for i in range(n):
        s = n-1 + i*n
        x = M[s,0]
```

```
        y = M[s,1]
        lc += (U[s,0] - uc(x,y))**2
    # 0 == x e 0 < y < 1
    for j in range(1, n-1):
        s = j
        x = M[s,0]
        y = M[s,1]
        lc += (U[s,0] - uc(x,y))**2
    # 1 == x e 0 < y < 1
    for j in range(1, n-1):
        s = j + n*(n-1)
        x = M[s,0]
        y = M[s,1]
        lc += (U[s,0] - uc(x,y))**2
    lc *= p/nc

    loss = lin + lc
    return loss


# dados do problema

# collocation points
a = 1
n = 11
ns = n**2
h = 1./(n-1)
h2 = h**2

# malha
x = torch.linspace(0, 1, n)
y = torch.linspace(0, 1, n)

M = torch.empty((ns, 2))
s = 0
for i, xx in enumerate(x):
  for j, yy in enumerate(y):
    M[s,0] = xx
    M[s,1] = yy
    s += 1
```

```

# gráfico
X, Y = np.meshgrid(x, y)
U_esp = u(X, Y)

# training
nepochs = 10000
nout_loss = 100
nout_plot = 500

for epoch in range(nepochs):

    # forward
    U_est = model(M)

    # loss function
    loss = laplace_loss(M, U_est, h2, n, u, p=10.)

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

    # output current solution
    if ((epoch) % nout_plot == 0):
        # verificação
        fig = plt.figure()
        ax = fig.add_subplot()

        ns = 50
        x1 = torch.linspace(0., 1., ns)
        x2 = torch.linspace(0., 1., ns)
        X1, X2 = torch.meshgrid(x1, x2)
        # exact
        Z_esp = torch.empty_like(X1)
        for i,x in enumerate(x1):
            for j,y in enumerate(x2):
                Z_esp[i,j] = u(x, y)
        c = ax.contour(X1, X2, Z_esp, levels=10, colors='white')
        ax.clabel(c)

        X_plot = torch.cat((X1.reshape(-1,1),
```

```
                            X2.reshape(-1,1)), dim=1)
        Z_est = model(X_plot)
        Z_est = Z_est.reshape((ns,ns))
        cf = ax.contourf(X1, X2, Z_est.detach(), levels=10, cmap='cool
        plt.colorbar(cf)

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

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

3.3.2 Autograd

Código 3.3: mlp_eqlaplace_ag.py

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

# modelo
model = torch.nn.Sequential(
    torch.nn.Linear(2,50),
    torch.nn.Tanh(),
    torch.nn.Linear(50,10),
    torch.nn.Tanh(),
    torch.nn.Linear(10,5),
    torch.nn.Tanh(),
    torch.nn.Linear(5,1)
)

# SGD - (Stochastic) Gradient Descent
optim = torch.optim.SGD(model.parameters(),
                        lr = 1e-3,
                        momentum = 0.9,
```

```
                              dampening = 0.)

# Solução esperada
def u(x, y):
    return a*x*(1-x) - a*y*(1-y)


def laplace_loss(X, U, h2, n, uc=u, p=1.):
    # num de amostras
    nc = 2*n + 2*(n-2)
    ni = n**2 - nc

    # loss interno
    lin = 0.
    for i in range(1,n-1):
      for j in range(1,n-1):
        s = j + i*n
        x = X[s:s+1,:].detach()
        x.requires_grad = True
        u = model(x)
        grad_u = torch.autograd.grad(u, x,
                                     create_graph = True,
                                     retain_graph = True)[0]
        u_x = grad_u[0,0]
        u_y = grad_u[0,1]

        u_xx = torch.autograd.grad(u_x, x,
                                   create_graph = True,
                                   retain_graph = True)[0][0,0]
        u_yy = torch.autograd.grad(u_y, x,
                                   create_graph = True,
                                   retain_graph = True)[0][0,1]
        lin = torch.add(lin, (u_xx + u_yy)**2)
    lin /= ni

    # loss contorno
    lc = 0.
    # 0 <= x <= 1 e y == 0
    for i in range(n):
        s = i*n
```

```
            x = M[s,0]
            y = M[s,1]
            lc += (U[s,0] - uc(x,y))**2
        # 0 <= x <= 1 e y == 1
        for i in range(n):
            s = n-1 + i*n
            x = M[s,0]
            y = M[s,1]
            lc += (U[s,0] - uc(x,y))**2
        # 0 == x e 0 < y < 1
        for j in range(1, n-1):
            s = j
            x = M[s,0]
            y = M[s,1]
            lc += (U[s,0] - uc(x,y))**2
        # 1 == x e 0 < y < 1
        for j in range(1, n-1):
            s = j + n*(n-1)
            x = M[s,0]
            y = M[s,1]
            lc += (U[s,0] - uc(x,y))**2
        lc *= p/nc

        loss = lin + lc
        return loss


# dados do problema

# collocation points
a = 1
n = 11
ns = n**2
h = 1./(n-1)
h2 = h**2

# malha
x = torch.linspace(0, 1, n)
y = torch.linspace(0, 1, n)

```

```
M = torch.empty((ns, 2))
s = 0
for i, xx in enumerate(x):
  for j, yy in enumerate(y):
    M[s,0] = xx
    M[s,1] = yy
    s += 1

# gráfico
X, Y = np.meshgrid(x, y)
U_esp = u(X, Y)

# training
nepochs = 10000
nout_loss = 100
nout_plot = 500

for epoch in range(nepochs):

    # forward
    U_est = model(M)

    # loss function
    loss = laplace_loss(M, U_est, h2, n, u, p=10.)

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

    # output current solution
    if ((epoch) % nout_plot == 0):
        # verificação
        fig = plt.figure()
        ax = fig.add_subplot()

        ns = 50
        x1 = torch.linspace(0., 1., ns)
        x2 = torch.linspace(0., 1., ns)
        X1, X2 = torch.meshgrid(x1, x2)
        # exact
        Z_esp = torch.empty_like(X1)
```

```
        for i,x in enumerate(x1):
            for j,y in enumerate(x2):
                Z_esp[i,j] = u(x, y)
        c = ax.contour(X1, X2, Z_esp, levels=10, colors='white')
        ax.clabel(c)

        X_plot = torch.cat((X1.reshape(-1,1),
                            X2.reshape(-1,1)), dim=1)
        Z_est = model(X_plot)
        Z_est = Z_est.reshape((ns,ns))
        cf = ax.contourf(X1, X2, Z_est.detach(), levels=10, cmap='cool
        plt.colorbar(cf)

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

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

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.