Luiz Guilherme Morais da Costa Faria

APRENDIZADO DE MÁQUINA

Brasília, DF
24 de setembro de 2025

Luiz Guilherme Morais da Costa Faria

APRENDIZADO DE MÁQUINA

Universidade de Brasília

Orientador: Nome do Orientador/Revisor (se aplicável)

Brasília, DF
24 de setembro de 2025

Sumário

Parte I

História da IA e do Computador

1 Uma Breve História do Computador

1.1 A Necessidade de Contar ao Longo das Eras

1.1.1 Ábaco

1.1.2 Régua de Cálculo

1.1.3 Bastões de Napier

1.1.4 Pascalina

2 Uma Breve História da Inteligência Artificial

Parte II

Conceitos Matemáticos

3 Cálculo para Aprendizado de Máquina

4 Álgebra Linear para Aprendizado de Máquina

4.1 A Unidade Fundamental: Vetores e Espaços Vetoriais

4.2 Organizando Dados: Matrizes e Suas Operações

4.3 Tensores: A Estrutura de Dados do Deep Learning

4.4 Resolvendo Sistemas e Encontrando Propriedades: Autovalores e Autovetores

4.5 Decomposição de Matrizes (SVD e PCA)

5 Probabilidade e Estatística para Aprendizado de Máquina

5.1 Medindo a Incerteza: Probabilidade Básica e Condicional

5.2 O Teorema de Bayes: Aprendendo com Evidências

5.3 Descrevendo os Dados: Estatística Descritiva: Média, mediana, variância, desvio padrão

5.4 Variáveis Aleatórias e Distribuições de Probabilidade

5.5 A Função de Máxima Verossimilhança (Maximum Likelihood Estimation - MLE)

Parte III

Pilares das Redes Neurais

6 O Algoritmo da Repropropagação e Os Otimizadores Baseados em Gradiente

6.1 O Método do Gradiente Descendente

O Método do Gradiente faz parte de uma série de métodos numéricos que possuem como função otimizar diferentes funções. Métodos dessa forma veem sendo estudados a séculos, um exemplo disso é o trabalho *Méthode générale pour la résolution des systèmes d'équations simultanées* (Método geral para resolução de sistemas de equações simultâneos em portguês) do matemático francês do século XVIII Cauchy (1847), em que o autor apresenta um método que pode ser considerado um precursor para o método do gradiente atual.

Nesse texto, o autor apresenta uma forma de minimizar uma função de múltiplas variáveis ($u = f(x, y, z)$) que não assume valores negativos, para fazer isso, ele faz uso do cálculo de derivadas parciais dessa função de cada um dos seus componentes ($D_x u, D_y u, D_z u$), em seguida, ele realiza um passo de atualização, de forma que os os valores de cada uma das variáveis sejam ligeiramente incrementados por valores (α, β, γ) (Cauchy, 1847). Um ponto importante destacado por Cauchy (1847) é de que esses incrementos devem ser proporcionais ao negativo das suas respectivas derivadas parciais, ele descreve que esse processo de calcular as derivadas e fazer pequenos incrementos deve ser feito de forma iterativa, assim, calculá-se as derivadas, faz-se os incrementos, e o passo é repetido até convergir para o valor mínimo de u.

Esse trabalho explica bem como aplicar o método do gradiente para se calcular mínimos de funções, mas para facilitar o entendimento do leitor, em seguida está um exemplo ilutrativo explicando o funcionamento dessa ferramenta.

6.1.1 Exemplo Ilustrativo: Cadeia de Montanhas

Imagine que você adora aventuras, e por isso, decidiu fazer uma trilha em uma floresta que fica em uma cadeia de montanhas que podem ser escaladas. Então, você teve a incrivel ideia de ir para o menor ponto dessa cadeia de montanhas, pois, no guia que você estava seguindo, falava que lá havia um lago com uma água cristalina, perfeito para tirar fotos.

Para chegar até esse lago, você conta com uma bússula um tanto quanto diferente, ao invés dela apontar para o norte como uma bússola comum, ela aponta para a direção do lugar com menor altitude de uma região. Isso é perfeito para o que você precisa, pois ela irá apontar justamente para o lago de você quer ir.

Com isso em mente, você criou um plano de como irá chegar a esse lago, ele é método que segue dois passos diferentes, sendo eles:

1. Olha na bússola qual a direção ela está apontando;
2. Anda um metro na direção apontada pela bússola.

Você chegou na conclusão que se seguir essa estratégia repetidas vezes, em algum momento, você inevitavelmente irá chegar no lago que está querendo tirar as suas fotos.

Na matemática, existe um método semelhante a este, que busca com base em uma bússola (chamada de vetor gradiente), encontrar um ponto de mínimo de um determinado lugar (neste caso, uma função composta por múltiplas variáveis). Esse é o método do gradiente, ele é ponto central desse capítulo, pois, ele (e suas variações) junto com o algortimo da retroprogação são algumas das principais ferramentas que colaboram para que os modelos de aprendizado de máquina possam aprender com os seus erros e com isso se tornarem melhores a cada iteração.

6.1.2 O Método em Si

A vantagem do método do gradiente é que ele é uma ferramenta matemática, e por isso pode ser representado utilizando notações mais formais e de forma enxuta. As notações utilizadas por Cauchy são diferentes das que são utilizadas hoje em dia, mas o seu significado não muda. Em *Deep Learning*, Goodfellow, Bengio e Courville (2016) explicam essa ferramenta através da equação 6.1 que deve ser repetida por múltiplos passos até o modelo convergir, ou seja, encontrar o ponto de mínimo da função estudada.

Método do Gradiente Descendente

$$x' = x - \epsilon \nabla f(x) \tag{6.1}$$

Em que:

- x': representa as coordenadas do próximo ponto;
- x: representa as coordernadas do ponto atual;
- ϵ: representa o tamanho do passo, também chamado de taxa de aprendizado;
- $\nabla f(x)$: representa o vetor gradiente calculado na posição do ponto atual (x) para função que se deseja otimizar.

Note que assim como no método proposto por Cauchy, é pego como base o inverso do vetor gradiente. Isso ocorre pois o vetor gradiente é um vetor especial que tem como principal propriedade apontar para a direção de maior crescimento de uma função no ponto que está sendo calculada. Mas no método, o objetivo não é encontrar o ponto que irá gerar os maiores valores da função, e sim o contrário. Por isso, é tomado inverso do vetor gradiente, que, dessa forma, estará então apontando para a direção de menor crescimento de uma função.

Um ponto a ser destacado nesse metodo é na hora de escolher uma taxa de aprendizado para ser utilizada no método. Uma taxa de aprendizado muito pequena signifca que o passo que o modelo irá dar de um ponto para outro será menor, e com isso implica que ele levará mais passos para encontrar um ponto de mínimo. É como se você fosse comparar a quantidade de passos que você gasta para andar do seu quarto até a sua cozinha com a quantidade de passos dados por uma formiga até lá, ambos vão chegar no local, mas a formiga certamente irá demorar bem mais. Considerando isso, surge então a hipótese de que quanto maior for o passo, mais rápido será a convergência, mas isso também não funciona muito bem, pois um passo muito largo pode ultrapassar o ponto de mínimo indo parar em outro canto da função, e ficará tentando chegar até o mínimo mas não irá conseguir pois caminha uma distância muito grande de uma só vez. Essas duas situações, em que o passo é pequeno demais e que o passo e grande demais, são ilustradas na figura 1.

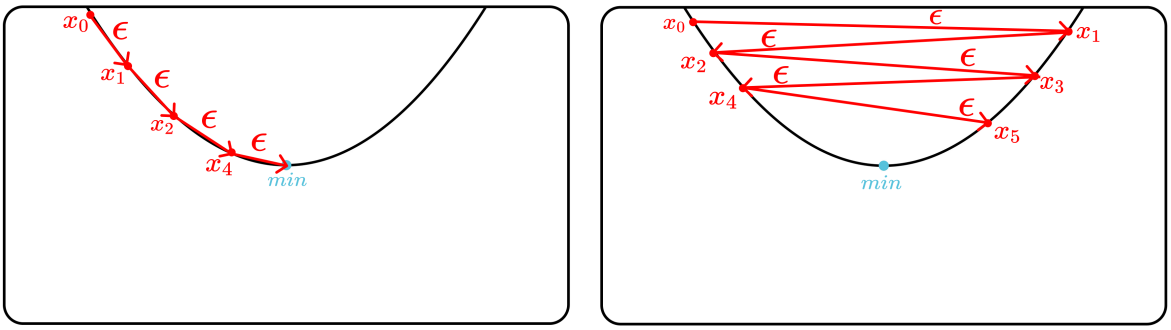

Figura 1 – Comparativo do tamanho de passos em uma função polinomial.

Na prática, escolher o valor da taxa de aprendizado é uma tarefa que irá depender de modelo em modelo, também irá variar com os diferentes métodos de otimização além da topologia da rede neural que está sendo construída. É sempre recomendado então experimentar diferentes tamanhos de passo, de forma que seja encontrado um que melhor se ajusta ao cenário que está sendo trabalhado.

Outro ponto que deve-se atentar é com relação as funções que estão sendo analisas ao utilizar o método do gradiente mas também qualquer otimizador que seja baseado nele. Se tivermos uma função convexa, em que seu formato lembra um funil, será bem mais fácil para o modelo encontrar o ponto de mínimo global daquela função.

Mas se tivermos uma função não convexa, cheia de ondas e com muitos pontos de mínimos locais e pontos de sela, a convergência do modelo será pior, pois existe a chance de que ele fique preso em um ponto de mínimo local ou em um ponto de sela. Isso afeta diretamente o desempenho da rede neural que estará sendo criada, fazendo com que ela tenha métricas piores. O problema é que muitas das vezes a função $f(x)$ que estaremos interessados para calcular o desemepenho do modelo será não convexa, dificultando o seu aprendizado.

Na figura 2 é possível ver o gráfico de duas funções diferentes, a primeira sendo uma função convexa e a segunda uma função não convexa.

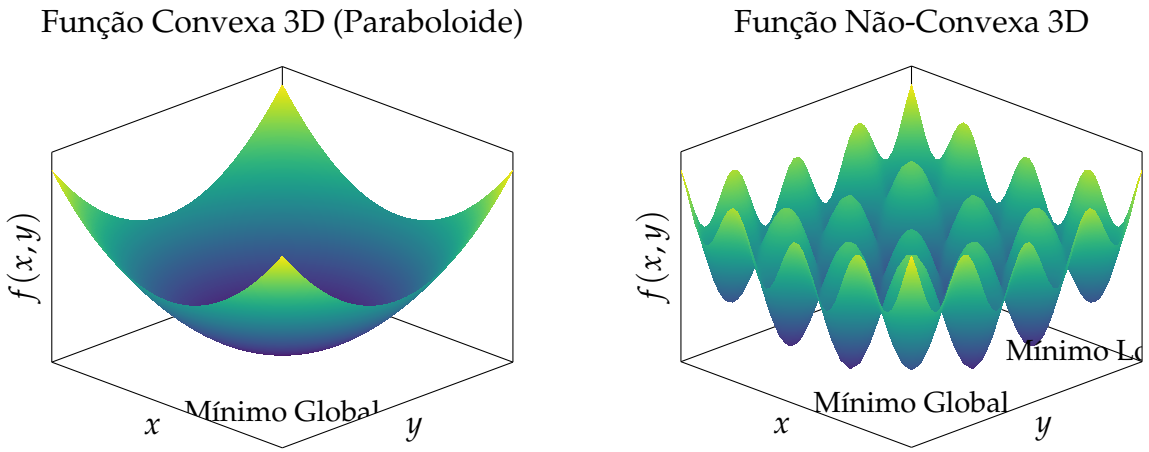

Figura 2 – Comparação entre funções 3D convexas e não-convexas.

6.1.3 Implementação em Python

Para implementar o método do gradiente utilizando Python e a biblioteca de cálculos Numpy, deve-se seguir como base a equação 6.1, criando uma classe implenta essa ferramenta, recebendo como parâmetros de entrada a taxa de aprendizado (`learning_rate`), a função que se quer encontrar o ponto de mínimo (`function`), e um ponto inicial (`initial_point`), que pode ser um conjunto de coordenadas aleatórias ou escolhidas pelo programador.

Outro ponto que deve ser destacado nas informações dessa função é de que ela também irá receber a derivada da função (`function_prime`) que se quer descobrir o ponto de mínimo, pois não será feito cálculo simbólico para calcular o vetor gradiente. Dessa forma os cálculos serão mais rápidos. Além disso, também é preciso definir outros parâmetros auxiliares, como a quantidade máxima de iterações que o modelo irá seguir (`tolerance`), que são chamadas de épocas `epochs`, também será definida um grau de tolerância para o modelo, pois, podem existir casos em que a norma do vetor gradiente são será exatemente zero, mas um valor muito próximo de zero, assim a

tolerância será responsável por definir qual valor dessa diferença será aceitável para o problema.

Por fim, um ponto interessante que é possível adicionar nessa função é uma lista, que irá armazenar todos os pontos que o modelo passou em cada uma de suas iterações, indicando o seu caminho pela função que está sendo estudada.

Bloco de Código : Classe completa do otimizador GradientDescent

```
class GradientDescent:

    def __init__(self, function, function_prime, initial_point
    , learning_rate=0.001, epochs=100, tolerance=1e-6):
        self.f = function
        self.fp = function_prime
        self.ip = initial_point
        self.lr = learning_rate
        self.ep = epochs
        self.tol = tolerance
        self.path = []

    def update_step(self):
        for i in range(self.ep):
            self.path.append(self.ip)
            grad = self.fp(self.ip)
            if abs(grad) < self.tol: break
            self.ip = self.ip + self.lr * (-grad)
        return self.ip, self.path
```

Note que a classe apresenta apenas dois métodos, o primeiro sendo o `__init__`, que inicializa os parâmetros da classe, e a `update_step` a qual é responsável por implementar de fato o método do gradiente, ela irá retornar o ponto de mínimo e também a lista com os pontos pelos quais o modelo passou ao longo das iterações.

6.2 A Retropropagação: Aprendendo com os Erros

Ainda no contexto de utilizar com o vetor gradiente para otimizar um modelo de rede neural, existe uma ferramenta que trabalha justamente com essse processo, ela é a retropropagação ou *backpropagation* em inglês.

> **Definição:** A **retropropagação** é uma ferramenta que veio para permitir que redes que fazem o uso de unidades de neurônios possam aprender, para isso, o procedimento ajusta repetidamente os pesos das conexões da rede para minimizar a diferença entre o valor atual da saída do vetor da rede neural com o valor real desejado (Rumelhart; Hinton; Williams, 1986).

Essa ferramenta foi introduzida para a comunidade científica pelos pesquisadores Rumelhart, Hinton e Williams (1986) no texto *Learning Representations by Back-Propagating Errors*,

6.3 Otimizadores Baseados em Gradiente

6.3.1 Método do Gradiente Estocástico

6.3.1.1 Implementação em Python

6.3.2 Método do Gradiente com Momentum

6.3.2.1 Implementação em Python

6.3.3 Nesterov

6.3.3.1 Implementação em Python

6.3.4 AdaGrad

6.3.4.1 Implementação em Python

6.3.5 RMSProp

6.3.5.1 Implementação em Python

6.3.6 Adam

6.3.6.1 Implementação em Python

6.3.7 Nadam

6.3.7.1 Implementação em Python

6.4 O Método de Newton: Indo Além do Gradiente

6.4.1 Implementação em Python

7 Funções de Ativação Sigmoidais

7.1 Teoremas da Aproximação Universal

7.2 Exemplos Ilustrativo

7.3 A Sigmoide Logística

Sigmoide Logística

$$\sigma(z_i) = \frac{1}{1 + e^{-z_i}} \tag{7.1}$$

Derivada da sigmoide logística

$$\frac{d}{dz_i}\sigma(z_i) = \frac{e^{-z_i}}{(1 + e^{-z_i})^2} \tag{7.2}$$

7.3.1 Implementação em Python

Bloco de Código : Classe completa do função de ativação Sigmoid

```
import numpy as np

class Sigmoid(Layer):
    def __init__(self):
        super().__init__()
        self.input = None
        self.sigmoid = None

    def forward(self, input_data):
        self.input = input_data
        self.sigmoid_output = 1/ (1 + np.exp(-input_data))
        return self.sigmoid_output

    def backward(self, grad_output):
        sigmoid_grad = self.sigmoid_output * (1 - self.
    sigmoid_output)
        return grad_output * sigmoid_grad, None
```

7.4 Tangente Hiperbólica

Tangente Hiperbólica

$$\tanh(z_i) = \frac{\sinh(z_i)}{\cosh(z_i)} = \frac{e_i^z - e^{-z_i}}{e_i^z + e^{-z_i}} \tag{7.3}$$

Derivada da tangente hiperbólica

$$\frac{d}{dz_i}\tanh(z_i) = \operatorname{sech}^2(z_i) \tag{7.4}$$

7.4.1 Implementação em Python

Bloco de Código : Classe completa do função de ativação Tangente Hiperbólica

```python
import numpy as np
from layers.base import Layer  # Assuming your base class is here

class Tanh(Layer):
    def __init__(self):
        super().__init__()
        self.input = None
        self.tanh_output = None

    def forward(self, input_data):
        self.input = input_data
        self.tanh_output = np.tanh(self.input)
        return self.tanh_output

    def backward(self, grad_output):
        tanh_grad = 1 - self.tanh_output**2
        return grad_output * tanh_grad, None
```

7.5 Softsign: Uma Sigmoidal Mais Barata

Softsign

$$\operatorname{softsign}(z_i) = \frac{z_i}{1 + |z_i|} \tag{7.5}$$

Derivada da softsign

$$\frac{d}{dz_i}\text{softsign}(z_i) = \frac{1}{(1+|z_i|)^2} \tag{7.6}$$

7.5.1 Implementação em Python

Bloco de Código : Classe completa do função de ativação Softsign

```python
from layers.base import Layer
import numpy as np

class Softsign(Layer):
    def __init__(self):
        super().__init__()
        self.input = None

    def forward(self, input_data):
        self.input = input_data
        return self.input / (1 + np.abs(self.input))

    def backward(self, grad_output):
        grad =  (1 / (1 + np.abs(self.input))**2)
        return grad_output * softsign_grad, None
```

7.6 Hard Sigmoid e Hard Tanh: O Sacrifício da Suavidade em Prol do Desempenho

Hard Sigmoid

$$\text{hard sigmoid}(z_i) = \begin{cases} 0 & \text{se } z_i < -3 \\ z_i/6 + 0.5 & \text{se } -3 \leq z_i \leq 3 \\ 1 & \text{se } z_i > 3 \end{cases} \tag{7.7}$$

Derivada da Hard Sigmoid

$$\frac{d}{dz_i}\text{hard sigmoid}(z_i) = \begin{cases} 0 & \text{se } z_i < -3 \\ 1/6 & \text{se } -3 < z_i < 3 \\ 0 & \text{se } z_i > 3 \end{cases} \tag{7.8}$$

Hard Tanh

$$\text{hard tanh}(z_i) = \begin{cases} -1 & \text{se } z_i < -1 \\ z_i & \text{se } -1 \leq z_i \leq 1 \\ 1 & \text{se } z_i > 1 \end{cases} \tag{7.9}$$

Derivada da Hard Tanh

$$\frac{d}{dz_i}\text{hard tanh}(z_i) = \begin{cases} 0 & \text{se } z_i < -1 \\ 1 & \text{se } -1 < z_i < 1 \\ 0 & \text{se } z_i > 1 \end{cases} \tag{7.10}$$

7.6.1 Implementação em Python

Bloco de Código : Classe completa do função de ativação Hard Sigmoid

```python
from layers.base import Layer
import numpy as np

class HardSigmoid(Layer):

    def __init__(self):
        super().__init__()
        self.input = None

    def forward(self, input_data):
        self.input = input_data

        output = self.input / 6 + 0.5
        output = np.clip(output, 0, 1)  # A more concise way to handle the bounds

        return output

    def backward(self, grad_output):
        hard_sigmoid_grad = np.full_like(self.input, 1 / 6)

        hard_sigmoid_grad[self.input < -3] = 0
        hard_sigmoid_grad[self.input > 3] = 0

        return grad_output * hard_sigmoid_grad, None
```

Bloco de Código : Classe completa do função de ativação Hard Sigmoid

```
from layers.base import Layer
import numpy as np


class HardTanh(Layer):
    def __init__(self):
        super().__init__()
        self.input = None

    def forward(self, input_data):
        self.input = input_data
        return np.clip(self.input, -1, 1)

    def backward(self, grad_output):

        hard_tanh_grad = np.where((self.input > -1) & (self.
    input < 1), 1, 0)

        return grad_output * hard_tanh_grad, None
```

7.7 O Desaparecimento de Gradientes

7.8 Comparativo de Desempenho das Sigmoidais

8 Funções de Ativação Retificadoras

8.1 Exemplo Ilustrativo

8.2 Rectified Linear Unit e Revolução Retificadora

Rectified Linear Unit (ReLU)

$$\text{ReLU}(z_i) = \begin{cases} z_i, & \text{se } z_i > 0 \\ 0, & \text{se } z_i \leq 0 \end{cases} \tag{8.1}$$

Derivada Rectified Linear Unit (ReLU)

$$\frac{d}{dz_i}[ReLU](z_i) = \begin{cases} 1, & \text{se } z_i > 0 \\ 0, & \text{se } z_i \leqslant 0 \end{cases} \tag{8.2}$$

8.2.1 Implementação em Python

Bloco de Código : Classe completa do função de ativação Rectified Linear Unit

```python
import numpy as np
from layers.base import Layer

class ReLU(Layer):
    def __init__(self):
        super().__init__()
        self.input = None

    def forward(self, input_data):
        self.input = input_data
        return np.maximum(0, self.input)

    def backward(self, grad_output):
        relu_grad = (self.input > 0)

        # Apply the chain rule
        return grad_output * relu_grad, None
```

8.3 Dying ReLUs Problem

8.4 Corrigindo o Dying ReLUs Problem: As Variantes com Vazamento

8.4.1 Leaky ReLU

Leaky ReLU (LReLU)

$$\text{LReLU}(z_i) = \begin{cases} z_i, & \text{se } z_i \geq 0 \\ \alpha \cdot z_i, & \text{se } z_i < 0 \end{cases} \tag{8.3}$$

Derivada Leaky ReLU (LReLU)

$$\frac{d}{dz_i}[LReLU](z_i) = \begin{cases} 1, & \text{se } z_i > 0 \\ \alpha, & \text{se } z_i \leqslant 0 \end{cases} \tag{8.4}$$

8.4.1.1 Implementação em Python

Bloco de Código : Classe completa do função de ativação Leaky ReLU

```
1  import numpy as np
2  from layers.base import Layer
3
4
5  class LeakyReLU(Layer):
6      def __init__(self, alpha=0.01):
7          super().__init__()
8          self.input = None
9          self.alpha = alpha
10
11     def forward(self, input_data):
12         self.input = input_data
13         return np.maximum(self.input * self.alpha, self.input)
14
15     def backward(self, grad_output):
16         leaky_relu_grad = np.where(self.input > 0, 1, self.
       alpha)
17         return grad_output * leaky_relu_grad, None
```

8.4.2 Parametric ReLU

Parametric ReLU (PReLU)

$$\text{PReLU}(z_i) = \begin{cases} z_i, & \text{se } z_i \geq 0 \\ \alpha_i \cdot z_i, & \text{se } z_i < 0 \end{cases} \tag{8.5}$$

Derivada Parametric ReLU (PReLU)

$$\frac{d}{dz_i}[PReLU](z_i) = \begin{cases} 1, & \text{se } z_i > 0 \\ \alpha_i, & \text{se } z_i < 0 \\ \nexists, & \text{se } z_i = 0 \end{cases} \tag{8.6}$$

8.4.3 Randomized Leaky ReLU

Randomized Leaky ReLU (RReLU)

$$\text{RReLU}(z_i) = \begin{cases} z_i, & \text{se } z_i > 0 \\ \alpha_i z_i, & \text{se } z_i \leq 0 \end{cases} \tag{8.7}$$

Derivada Randomized Leaky ReLU (RReLU)

$$\frac{d}{dz_i}[RReLU](z_i) = \begin{cases} 1, & \text{se } z_i > 0 \\ \alpha_i, & \text{se } z_i \leqslant 0 \end{cases} \tag{8.8}$$

8.5 Em Busca da Suavidade

8.5.1 Exponential Linear Unit

Exponential Linear Unit (ELU)

$$\text{ELU}(z_i) = \begin{cases} z_i, & \text{se } z_i \geq 0 \\ \alpha \cdot (e^{z_i} - 1), & \text{se } z_i < 0 \end{cases} \tag{8.9}$$

Derivada Exponential Linear Unit (ELU)

$$\frac{d}{dz_i}[ELU](z_i) = \begin{cases} 1, & \text{se } z_i > 0 \\ \alpha \cdot e^{z_i}, & \text{se } z_i \leq 0 \end{cases} \tag{8.10}$$

8.5.2 Scaled Exponential Linear Unit

Scaled Exponential Linear Unit (ELU)

$$\text{SELU}(z_i) = \lambda \begin{cases} z_i, & \text{se } z_i > 0 \\ \alpha \cdot (e^{z_i} - 1), & \text{se } z_i \leq 0 \end{cases} \tag{8.11}$$

Derivada Scaled Exponential Linear Unit (ELU)

$$\frac{d}{dz_i}[SELU](z_i) = \lambda \begin{cases} 1, & \text{se } z_i > 0 \\ \alpha \cdot e^{z_i}, & \text{se } z_i \leq 0 \end{cases} \tag{8.12}$$

8.5.3 Noisy ReLU

Noisy ReLU (NReLU)

$$\text{NReLU}(z_i) = \begin{cases} 0 & \text{se } z_i \leq 0 \\ z_i + \mathcal{N}(0, \sigma(z_i)) & \text{se } z_i > 0 \end{cases} \tag{8.13}$$

Derivada Noisy ReLU (NReLU)

$$\frac{d}{dz_i}[\text{NReLU}](z_i) = \begin{cases} 0 & \text{se } z_i \leq 0 \\ 1 & \text{se } z_i > 0 \end{cases} \tag{8.14}$$

8.6 O Problema dos Gradientes Explosivos

8.7 Comparativo de Desempenho das Funções Retificadoras

9 Funções de Ativação Modernas e Outras Funções de Ativação

O texto do seu capítulo começa aqui...

10 Funções de Perda para Classificação Binária

10.1 A Intuição da Perda: Medindo o Erro do Modelo

10.2 Entropia Cruzada Binária (Binary Cross-Entropy): A função de perda padrão

10.3 Perda Hinge (Hinge Loss)

10.4 Comparativo Visual e Prático

11 Funções de Perda para Classificação Multilabel

11.1 Softmax e a Distribuição de Probabilidades

11.2 Entropia Cruzada Categórica (Categorical Cross-Entropy)

11.3 Entropia Cruzada Categórica Esparsa (Sparse Categorical Cross-Entropy)

12 Metaheurísticas: Otimizando Redes Neurais Sem o Gradiente

O texto do seu capítulo começa aqui...

12.1 Algoritmos Evolutivos

12.2 Inteligência de Enxame

Parte IV

Aprendizado de Máquina Clássico

13 Técnicas de Regressão

13.1 Exemplo Ilustrativo

13.2 Regressão Linear

13.2.1 Função de Custo MSE

13.2.2 Equação Normal

13.2.3 Implementação em Python

13.3 Regressão Polininomial

13.3.1 Impletanção em Python

13.4 Regressão de Ridge

13.4.1 Implementação em Python

13.5 Regressão de Lasso

13.5.1 Implementação em Python

13.6 Elastic Net

13.6.1 Implementação em Python

13.7 Regressão Logística

13.7.1 Implementação em Python

13.8 Regressão Softmax

13.8.1 Implementação em Python

13.9 Outras Técnicas de Regressão

14 Árvores de Decisão e Florestas Aleatórias

15 Máquinas de Vetores de Suporte

15.1 Exemplo Ilustrativo

16 Ensamble

16.1 Exemplo Ilustrativo

17 Dimensionalidade

17.1 Exemplo Ilustrativo

17.2 A Maldição da Dimensionalidade

17.3 Seleção de Características (Feature Selection)

17.4 Extração de Características (Feature Extraction)

17.4.1 Análise de Componentes Principais (PCA)

17.4.2 t-SNE (t-Distributed Stochastic Neighbor Embedding) e UMAP

18 Clusterização

Parte V

Redes Neurais Profundas (DNNs)

19 Perceptrons MLP - Redes Neurais Artificiais

O texto do seu capítulo começa aqui...

20 Redes FeedForward (FFNs)

O texto do seu capítulo começa aqui...

21 Redes de Crença Profunda (DBNs) e Máquinas de Boltzmann Restritas

O texto do seu capítulo começa aqui...

22 Redes Neurais Convolucionais (CNN)

22.1 Exemplo Ilustrativo

22.2 Camadas Convolucionais: O Bloco Fundamental para as CNNs

22.2.1 Implementação em Python

Bloco de Código : Classe completa de Convolution2D

```
import numpy as np
from layers.base import Layer

class Convolution2D(Layer):
    def __init__(self, input_channels, num_filters, kernel_size, stride=1, padding=0):
        super().__init__()
        self.input_channels = input_channels
        self.num_filters = num_filters
        self.kernel_size = kernel_size
        self.stride = (stride, stride) if isinstance(stride, int) else stride
        self.padding = padding

        kernel_height, kernel_width = self.kernel_size
        self.kernels = np.random.randn(num_filters, input_channels, kernel_height, kernel_width) * 0.01
        self.biases = np.zeros((num_filters, 1))
        self.params = [self.kernels, self.biases]
        self.cache = None

    def forward(self, input_data):
        (batch_size, input_height, input_width, input_channels) = input_data.shape
        filters, _, kernel_height, kernel_width = self.kernels.shape
        stride_height, stride_width = self.stride

        pad_config = ((0, 0), (self.padding, self.padding), (self.padding, self.padding), (0, 0))
        input_padded = np.pad(input_data, pad_config, mode='constant')
        self.cache = input_padded
```

22.3 Camadas de Poooling: Reduzindo a Dimensionalidade

22.3.1 Max Pooling

22.3.1.1 Implementação em Python

Bloco de Código : Classe completa de MaxPooling2D

```
import numpy as np
from layers.base import Layer

class MaxPooling2D(Layer):
    def __init__(self, pool_size=(2,2), stride=None):
        super().__init__()
        self.pool_size = pool_size
        self.stride = stride if stride is not None else
    pool_size
        self.cache = None

    def forward(self, input_data):
        (batches, input_height, input_width, channels) =
    input_data.shape

        pool_height, pool_width = self.pool_size
        stride_height, stride_width = self.stride

        output_height = int((input_height - pool_height) /
    stride_height) + 1
        output_width = int((input_width - pool_width) /
    stride_width) + 1

        output_matrix = np.zeros((batches, output_height,
    output_width, channels))
        self.cache = np.zeros_like(input_data)

        for b in range(batches):
            for c in range(channels):
                for h in range(output_height):
                    for w in range(output_width):

                        start_height = h * stride_height
                        start_width = w * stride_width
                        end_height = start_height +
    pool_height
                        end_width = start_width + pool_width

```

22.3.2 Average Pooling

22.3.2.1 Implementação em Python

Bloco de Código : Classe completa de AveragePooling2D

```
import numpy as np
from layers.base import Layer

class AveragePooling2D(Layer):
    def __init__(self, pool_size=(2, 2), stride=None):
        super().__init__()
        self.pool_size = pool_size
        self.stride = stride if stride is not None else
    pool_size

    def forward(self, input_data):
        (batches, input_height, input_width, channels) =
    input_data.shape
        pool_h, pool_w = self.pool_size
        stride_h, stride_w = self.stride

        output_height = int((input_height - pool_h) / stride_h
    ) + 1
        output_width = int((input_width - pool_w) / stride_w)
    + 1

        output_matrix = np.zeros((batches, output_height,
    output_width, channels))

        for b in range(batches):
            for c in range(channels):
                for h in range(output_height):
                    for w in range(output_width):
                        start_h = h * stride_h
                        start_w = w * stride_w
                        end_h = start_h + pool_h
                        end_w = start_w + pool_w

                        pooling_window = input_data[b, start_h
    :end_h, start_w:end_w, c]
                        output_matrix[b, h, w, c] = np.mean(
    pooling_window)  # Changed to mean

        return output_matrix

```

22.3.3 Global Average Pooling

22.3.3.1 Implementação em Python

Bloco de Código : Classe completa de GlobalAveragePooling2D

```
import numpy as np
from layers.base import Layer

class GlobalAveragePooling2D(Layer):
    def __init__(self):
        super().__init__()
        self.input_shape = None

    def forward(self, input_data):
        self.input_shape = input_data.shape

        output = np.mean(input_data, axis=(1, 2), keepdims=
    True)

        return output

    def backward(self, output_gradient):
        _, input_h, input_w, _ = self.input_shape

        distributed_grad = output_gradient / (input_h *
    input_w)

        upsampled_grad = np.ones(self.input_shape) *
    distributed_grad

        return upsampled_grad, None
```

22.4 Camada Flatten: Achatando os Dados

22.4.1 Implementação em Python

Bloco de Código : Classe completa de Flatten

```
import numpy as np
from layers.base import Layer

class Flatten(Layer):
    def __init__(self):
        super().__init__()
        self.input_shape = None

    def forward(self, input_data):
        self.input_shape = input_data.shape

        flatten_output = input_data.reshape(input_data.shape
    [0], -1)

        return flatten_output

    def backward(self, output_gradient):
        input_gradient = output_gradient.reshape(self.
    input_shape)
        return input_gradient, None
```

22.5 Criando uma CNN

22.6 Detecção de Objetos

22.7 Redes Totalmente Convolucionais (FCNs)

22.8 You Only Look Once (YOLO)

22.9 Algumas Arquiteturas de CNNs

22.9.1 LeNet-5

22.9.2 AlexNet

22.9.3 GoogLeNet

22.9.4 VGGNet

22.9.5 ResNet

22.9.6 Xception

22.9.7 SENet

23 Redes Residuais (ResNets)

O texto do seu capítulo começa aqui...

24 Redes Neurais Recorrentes (RNN)

O texto do seu capítulo começa aqui...

24.1 Exemplo Ilustrativo

24.2 Neurônios e Células Recorrentes

24.2.1 Implementação em Python

24.3 Células de Memória

24.3.1 Implementação em Python

24.4 Criando uma RNN

24.5 O Problema da Memória de Curto Prazo

24.5.1 Células LSTM

24.5.2 Conexões Peephole

24.5.3 Células GRU

25 Técnicas para Melhorar o Desempenho de Redes Neurais

26 Transformers

27 Redes Adversárias Generativas (GANs)

O texto do seu capítulo começa aqui...

28 Mixture of Experts (MoE)

O texto do seu capítulo começa aqui...

29 Modelos de Difusão

O texto do seu capítulo começa aqui...

30 Redes Neurais de Grafos (GNNs)

O texto do seu capítulo começa aqui...

Parte VI

Apêndices

Referências

CAUCHY, Augustin-Louis. Méthode générale pour la résolution des systèmes d'équations simultanées. *Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences*, v. 25, p. 536–538, 1847. Citado na p. 27.

GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. *Deep Learning*. [*S. l.*]: MIT Press, 2016. Citado na p. 28.

RUMELHART, David E.; HINTON, Geoffrey E.; WILLIAMS, Ronald J. Learning Representations by Back-Propagating Errors. *Nature*, v. 323, p. 533–536, 1986. Citado na p. 32.