

Luiz Guilherme Moraes da Costa Faria

# APRENDIZADO DE MÁQUINA

Brasília, DF  
26 de setembro de 2025



Luiz Guilherme Moraes da Costa Faria

# APRENDIZADO DE MÁQUINA

Universidade de Brasília

Orientador: Nome do Orientador/Revisor (se aplicável)

Brasília, DF  
26 de setembro de 2025



# Sumário

Sumário . . . . .	3
I	HISTÓRIA DA IA E DO COMPUTADOR 11
1	UMA BREVE HISTÓRIA DO COMPUTADOR . . . . . 13
1.1	A Necessidade de Contar ao Longo das Eras . . . . . 13
1.1.1	Ábaco . . . . . 13
1.1.2	Régua de Cálculo . . . . . 13
1.1.3	Bastões de Napier . . . . . 13
1.1.4	Pascalina . . . . . 13
2	UMA BREVE HISTÓRIA DA INTELIGÊNCIA ARTIFICIAL . . . 15
2.1	Os Anos 1900 . . . . . 15
2.2	Os Anos 1910 . . . . . 15
2.3	Os Anos 1920 . . . . . 15
2.4	Os Anos 1930 . . . . . 15
2.5	Os Anos 1940 . . . . . 15
2.6	Os Anos 1950 . . . . . 15
2.7	Os Anos 1960 . . . . . 15
2.8	Os Anos 1970 . . . . . 15
2.9	Os Anos 1980 . . . . . 15
2.10	Os Anos 1990 . . . . . 15
2.11	Os Anos 2000 . . . . . 15
2.12	Atualidade . . . . . 15
II	CONCEITOS MATEMÁTICOS 17
3	CÁLCULO PARA APRENDIZADO DE MÁQUINA . . . . . 19
3.1	Funções: A Base do Cálculo . . . . . 19
3.2	Derivadas Ordinárias . . . . . 19
3.3	Integrais Simples . . . . . 19
3.4	Derivadas Parciais . . . . . 19
4	ÁLGEBRA LINEAR PARA APRENDIZADO DE MÁQUINA . . . 21
4.1	A Unidade Fundamental: Vetores e Espaços Vetoriais . . . . . 21

4.2	Organizando Dados: Matrizes e Suas Operações . . . . .	21
4.3	Tensores: A Estrutura de Dados do Deep Learning . . . . .	21
4.4	Resolvendo Sistemas e Encontrando Propriedades: Autovalores e Autovetores . . . . .	21
4.5	Decomposição de Matrizes (SVD e PCA) . . . . .	21
5	<b>PROBABILIDADE E ESTATÍSTICA PARA APRENDIZADO DE MÁQUINA . . . . .</b>	<b>23</b>
5.1	Medindo a Incerteza: Probabilidade Básica e Condicional . . . . .	23
5.2	O Teorema de Bayes: Aprendendo com Evidências . . . . .	23
5.3	Descrevendo os Dados: Estatística Descritiva: Média, mediana, variância, desvio padrão . . . . .	23
5.4	Variáveis Aleatórias e Distribuições de Probabilidade . . . . .	23
5.5	A Função de Máxima Verossimilhança (Maximum Likelihood Estimation - MLE) . . . . .	23
<b>III</b>	<b>PILARES DAS REDES NEURAIS</b>	<b>25</b>
6	<b>O ALGORITMO DA REPROPROPAGAÇÃO E OS OTIMIZADORES BASEADOS EM GRADIENTE . . . . .</b>	<b>27</b>
6.1	<b>O Método do Gradiente Descendente . . . . .</b>	<b>27</b>
6.1.1	Exemplo Ilustrativo: Cadeia de Montanhas . . . . .	27
6.1.2	O Método em Si . . . . .	28
6.1.3	Implementação em Python . . . . .	30
6.2	<b>A Retropropagação: Aprendendo com os Erros . . . . .</b>	<b>32</b>
6.2.1	Utilizando o Gradiente Descendente para Atualizar os Pesos e Vieses . . . . .	37
6.2.2	Entendendo Como o Gradiente É Propagado ao Longo de Muitas Camadas . . . . .	39
6.3	<b>Otimizadores Baseados em Gradiente . . . . .</b>	<b>39</b>
6.3.1	Método do Gradiente Estocástico . . . . .	39
6.3.1.1	Implementação em Python . . . . .	39
6.3.2	Método do Gradiente com Momentum . . . . .	39
6.3.2.1	Implementação em Python . . . . .	39
6.4	<b>Otimizadores Modernos Baseados em Gradiente . . . . .</b>	<b>39</b>
6.4.1	Nesterov . . . . .	39
6.4.1.1	Implementação em Python . . . . .	39
6.4.2	AdaGrad . . . . .	39
6.4.2.1	Implementação em Python . . . . .	39
6.4.3	RMSPprop . . . . .	39
6.4.3.1	Implementação em Python . . . . .	39

6.4.4	Adam . . . . .	39
6.4.4.1	Implementação em Python . . . . .	39
6.4.5	Nadam . . . . .	39
6.4.5.1	Implementação em Python . . . . .	39
<b>6.5</b>	<b>O Método de Newton: Indo Além do Gradiente . . . . .</b>	<b>39</b>
6.5.1	Implementação em Python . . . . .	39
<b>7</b>	<b>FUNÇÕES DE ATIVAÇÃO SIGMOIDAIS . . . . .</b>	<b>41</b>
7.1	Teoremas da Aproximação Universal . . . . .	41
7.2	Exemplo Ilustrativo: Empurrando para extremos . . . . .	43
7.3	A Sigmoid Logística . . . . .	44
7.4	Contexto Histórico: Popularização da Sigmoid em Redes Neurais	44
7.5	Expressões Matemáticas . . . . .	47
7.5.1	Implementação em Python . . . . .	47
7.6	Tangente Hiperbólica . . . . .	48
7.6.1	Implementação em Python . . . . .	50
7.7	Softsign: Uma Sigmoidal Mais Barata . . . . .	50
7.7.1	Implementação em Python . . . . .	52
7.8	Hard Sigmoid e Hard Tanh: O Sacrifício da Suavidade em Prol do Desempenho . . . . .	53
7.8.1	Implementação em Python . . . . .	56
7.9	O Desaparecimento de Gradientes . . . . .	57
7.10	Comparativo de Desempenho das Sigmoidais . . . . .	57
<b>8</b>	<b>FUNÇÕES DE ATIVAÇÃO RETIFICADORAS . . . . .</b>	<b>59</b>
8.1	Exemplo Ilustrativo . . . . .	59
8.2	Rectified Linear Unit e Revolução Retificadora . . . . .	59
8.2.1	Implementação em Python . . . . .	60
8.3	Dying ReLUs Problem . . . . .	61
8.4	Corrigindo o Dying ReLUs Problem: As Variantes com Vazamento	61
8.4.1	Leaky ReLU . . . . .	61
8.4.1.1	Implementação em Python . . . . .	62
8.4.2	Parametric ReLU . . . . .	62
8.4.3	Randomized Leaky ReLU . . . . .	63
8.5	Em Busca da Suavidade . . . . .	64
8.5.1	Exponential Linear Unit . . . . .	64
8.5.2	Scaled Exponential Linear Unit . . . . .	65
8.5.3	Noisy ReLU . . . . .	66
8.6	O Problema dos Gradientes Explosivos . . . . .	67
8.7	Comparativo de Desempenho das Funções Retificadoras . . . . .	67

<b>9</b>	<b>FUNÇÕES DE ATIVAÇÃO MODERNAS E OUTRAS FUNÇÕES DE ATIVAÇÃO . . . . .</b>	<b>69</b>
<b>10</b>	<b>FUNÇÕES DE PERDA PARA CLASSIFICAÇÃO BINÁRIA . . . .</b>	<b>71</b>
10.1	A Intuição da Perda: Medindo o Erro do Modelo . . . . .	71
10.2	Entropia Cruzada Binária (Binary Cross-Entropy): A função de perda padrão . . . . .	71
10.3	Perda Hinge (Hinge Loss) . . . . .	71
10.4	Comparativo Visual e Prático . . . . .	71
<b>11</b>	<b>FUNÇÕES DE PERDA PARA CLASSIFICAÇÃO MULTILABEL .</b>	<b>73</b>
11.1	Softmax e a Distribuição de Probabilidades . . . . .	73
11.2	Entropia Cruzada Categórica (Categorical Cross-Entropy) . . . . .	73
11.3	Entropia Cruzada Categórica Esparsa (Sparse Categorical Cross-Entropy) . . . . .	73
<b>12</b>	<b>METAHEURÍSTICAS: OTIMIZANDO REDES NEURAI SEM O GRADIENTE . . . . .</b>	<b>75</b>
12.1	Algoritmos Evolutivos . . . . .	75
12.2	Inteligência de Enxame . . . . .	75
<b>IV</b>	<b>APRENDIZADO DE MÁQUINA CLÁSSICO</b>	<b>77</b>
<b>13</b>	<b>TÉCNICAS DE REGRESSÃO . . . . .</b>	<b>79</b>
13.1	Exemplo Ilustrativo . . . . .	79
13.2	Regressão Linear . . . . .	79
13.2.1	Função de Custo MSE . . . . .	79
13.2.2	Equação Normal . . . . .	79
13.2.3	Implementação em Python . . . . .	79
13.3	Regressão Polinomial . . . . .	79
13.3.1	Implementação em Python . . . . .	79
13.4	Regressão de Ridge . . . . .	79
13.4.1	Implementação em Python . . . . .	79
13.5	Regressão de Lasso . . . . .	79
13.5.1	Implementação em Python . . . . .	79
13.6	Elastic Net . . . . .	79
13.6.1	Implementação em Python . . . . .	79
13.7	Regressão Logística . . . . .	79
13.7.1	Implementação em Python . . . . .	79
13.8	Regressão Softmax . . . . .	79



13.8.1	Implementação em Python . . . . .	79
13.9	Outras Técnicas de Regressão . . . . .	79
14	<b>ÁRVORES DE DECISÃO E FLORESTAS ALEATÓRIAS . . . . .</b>	<b>81</b>
14.1	Exemplo Ilustrativo . . . . .	81
14.2	Entendendo o Conceito de Árvores . . . . .	81
14.2.1	Árvores Binárias . . . . .	81
14.3	Árvores de Decisão . . . . .	81
14.3.1	Implementação em Python . . . . .	81
14.4	Florestas Aleatórias . . . . .	81
14.4.1	Implementação em Python . . . . .	81
15	<b>MÁQUINAS DE VETORES DE SUPORTE . . . . .</b>	<b>83</b>
15.1	Exemplo Ilustrativo . . . . .	83
16	<b>ENSAMBLE . . . . .</b>	<b>85</b>
16.1	Exemplo Ilustrativo . . . . .	85
17	<b>DIMENSIONALIDADE . . . . .</b>	<b>87</b>
17.1	Exemplo Ilustrativo . . . . .	87
17.2	A Maldição da Dimensionalidade . . . . .	87
17.3	Seleção de Características (Feature Selection) . . . . .	87
17.4	Extração de Características (Feature Extraction) . . . . .	87
17.4.1	Análise de Componentes Principais (PCA) . . . . .	87
17.4.2	t-SNE (t-Distributed Stochastic Neighbor Embedding) e UMAP . . . . .	87
18	<b>CLUSTERIZAÇÃO . . . . .</b>	<b>89</b>
18.1	Exemplo Ilustrativo . . . . .	89
18.2	Aprendizado Não Supervisionado: Encontrando Grupos nos Dados	89
18.3	Clusterização Particional: K-Means . . . . .	89
18.4	Clusterização Hierárquica . . . . .	89
18.5	Clusterização Baseada em Densidade: DBSCAN . . . . .	89
V	<b>REDES NEURAIIS PROFUNDAS (DNNS)</b>	<b>91</b>
19	PERCEPTRONS MLP - REDES NEURAIIS ARTIFICIAIS . . . . .	93
20	REDES FEEDFORWARD (FFNS) . . . . .	95
21	REDES DE CRENÇA PROFUNDA (DBNS) E MÁQUINAS DE BOLTZMANN RESTRITAS . . . . .	97

<b>22</b>	<b>REDES NEURAIS CONVOLUCIONAIS (CNN)</b>	<b>100</b>
22.1	Exemplo Ilustrativo	100
22.2	Camadas Convolucionais: O Bloco Fundamental para as CNNs	100
22.2.1	Implementação em Python	100
22.3	Camadas de Pooling: Reduzindo a Dimensionalidade	102
22.3.1	Max Pooling	102
22.3.1.1	Implementação em Python	102
22.3.2	Average Pooling	104
22.3.2.1	Implementação em Python	104
22.3.3	Global Average Pooling	105
22.3.3.1	Implementação em Python	105
22.4	Camada Flatten: Achatando os Dados	106
22.4.1	Implementação em Python	106
22.5	Criando uma CNN	107
22.6	Detecção de Objetos	107
22.7	Redes Totalmente Convolucionais (FCNs)	107
22.8	You Only Look Once (YOLO)	107
22.9	Algumas Arquiteturas de CNNs	107
22.9.1	LeNet-5	107
22.9.2	AlexNet	107
22.9.3	GoogLeNet	107
22.9.4	VGGNet	107
22.9.5	ResNet	107
22.9.6	Xception	107
22.9.7	SENet	107
<b>23</b>	<b>REDES RESIDUAIS (RESNETS)</b>	<b>109</b>
<b>24</b>	<b>REDES NEURAIS RECORRENTES (RNN)</b>	<b>111</b>
24.1	Exemplo Ilustrativo	111
24.2	Neurônios e Células Recorrentes	111
24.2.1	Implementação em Python	111
24.3	Células de Memória	111
24.3.1	Implementação em Python	111
24.4	Criando uma RNN	111
24.5	O Problema da Memória de Curto Prazo	111
24.5.1	Células LSTM	111
24.5.2	Conexões Peephole	111
24.5.3	Células GRU	111

25	<b>TÉCNICAS PARA MELHORAR O DESEMPENHO DE REDES NEURAIS . . . . .</b>	<b>113</b>
25.1	Técnicas de Inicialização . . . . .	113
25.2	Regularização L1 e L2 . . . . .	113
25.3	Normalização . . . . .	113
25.3.1	Normalização de Camadas . . . . .	113
25.3.2	Normalização de Batch . . . . .	113
25.4	Clipping do Gradiente . . . . .	113
25.5	Dropout: Menos Neurônios Mais Aprendizado . . . . .	113
25.6	Data Augmentation . . . . .	113
26	<b>TRANSFORMERS . . . . .</b>	<b>115</b>
26.1	As Limitações das RNNs: O Gargalo Sequencial . . . . .	115
26.2	A Ideia Central: Self-Attention (Query, Key, Value) . . . . .	115
26.3	Escalando a Atenção: Multi-Head Attention . . . . .	115
26.4	A Arquitetura Completa: O Bloco Transformer . . . . .	115
26.5	Entendendo a Posição: Codificação Posicional . . . . .	115
26.6	As Três Grandes Arquiteturas . . . . .	115
26.6.1	Encoder-Only (Ex: BERT): Para tarefas de entendimento . . . . .	115
26.6.2	Decoder-Only (Ex: GPT): Para tarefas de geração . . . . .	115
26.6.3	Encoder-Decoder (Ex: T5): Para tarefas de tradução/sumarização . . . . .	115
26.7	Além do Texto: Vision Transformers (ViT) . . . . .	115
27	<b>REDES ADVERSÁRIAS GENERATIVAS (GANS) . . . . .</b>	<b>117</b>
28	<b>MIXTURE OF EXPERTS (MOE) . . . . .</b>	<b>119</b>
29	<b>MODELOS DE DIFUSÃO . . . . .</b>	<b>121</b>
30	<b>REDES NEURAIS DE GRAFOS (GNNS) . . . . .</b>	<b>123</b>
VI	<b>APÊNDICES . . . . .</b>	<b>125</b>
	Referências . . . . .	127



## 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

2.1 Os Anos 1900

2.2 Os Anos 1910

2.3 Os Anos 1920

2.4 Os Anos 1930

2.5 Os Anos 1940

2.6 Os Anos 1950

2.7 Os Anos 1960

2.8 Os Anos 1970

2.9 Os Anos 1980

2.10 Os Anos 1990

2.11 Os Anos 2000

2.12 Atualidade



## Parte II

### Conceitos Matemáticos



## 3 Cálculo para Aprendizado de Máquina

3.1 Funções: A Base do Cálculo

3.2 Derivadas Ordinárias

3.3 Integrais Simples

3.4 Derivadas Parciais



## 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 Repropagação 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 simultâneas em português) 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 valores de cada uma das variáveis sejam ligeiramente incrementados por valores  $(\alpha, \beta, \gamma)$  (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 ilustrativo 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 incrível 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 algoritmo da retropropagação 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

$$x = x_0 - \epsilon \nabla f(x) \quad (6.1)$$

Em que:

- $x'$ : representa as coordenadas do próximo ponto;
- $x$ : representa as coordenadas do ponto atual;
- $\epsilon$ : 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 método é na hora de escolher uma taxa de aprendizado para ser utilizada no método. Uma taxa de aprendizado muito pequena significa 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 é grande demais, são ilustradas na figura 1.

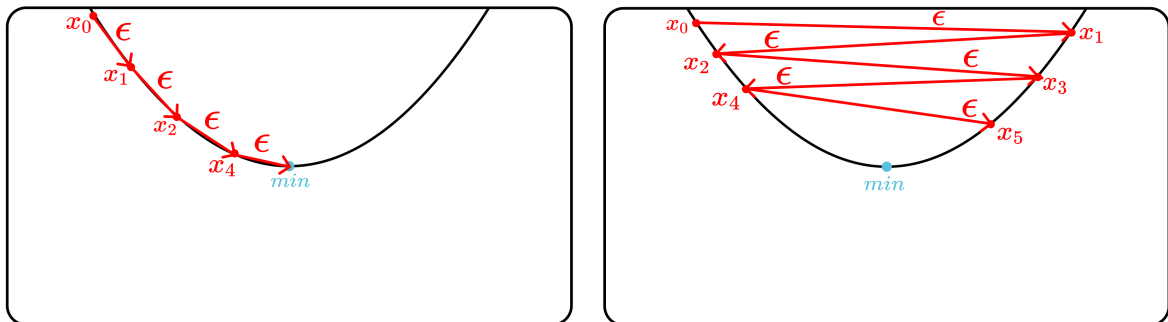


Figura 1 – Comparativo do tamanho de passos em uma função polinomial.

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 analisadas 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 desempenho 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.

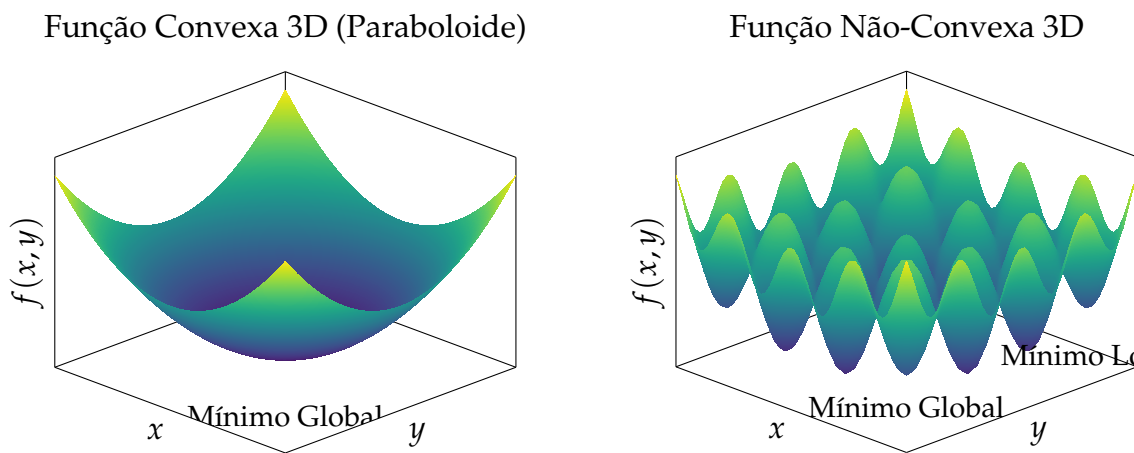


Figura 2 – Comparação entre funções 3D convexas e não-convexas.

### 6.1.3 Implementação em Python

---

#### Algorithm 1 O Método do Gradiente (Descida do Gradiente)

---

**Require:** Taxa de aprendizado global  $\epsilon$

**Require:** Parâmetros iniciais  $\theta$

- 1: Inicialize os parâmetros  $\theta$
  - 2: **while** critério de parada não for atingido **do**
  - 3:   Calcule o gradiente  $\mathbf{g} \leftarrow \nabla_{\theta} L(\theta)$ , usando **todo** o conjunto de treinamento
  - 4:   Calcule a atualização:  $\Delta\theta \leftarrow -\epsilon \mathbf{g}$
  - 5:   Aplique a atualização:  $\theta \leftarrow \theta + \Delta\theta$
  - 6: **end while**
- 

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 não será exatamente 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

```
1 class GradientDescent:
2
3     def __init__(self, function, function_prime, initial_point
4         , learning_rate=0.001, epochs=100, tolerance=1e-6):
5         self.f = function
6         self.fp = function_prime
7         self.ip = initial_point
8         self.lr = learning_rate
9         self.ep = epochs
10        self.tol = tolerance
11        self.path = []
12
13    def update_step(self):
14        for i in range(self.ep):
15            self.path.append(self.ip)
16            grad = self.fp(self.ip)
17            if abs(grad) < self.tol: break
18            self.ip = self.ip + self.lr * (-grad)
19        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 imple-

mentar 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 esse 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*, e será explicada nesse capítulo de forma detalhada. Para isso Rumelhart, Hinton e Williams (1986) começam introduzindo três diferentes funções que serão utilizadas ao longo do texto para deduzir os conceitos da retropropagação do gradiente, sendo elas: a equação do neurônio, a função de ativação sigmoide e a função de custo do erro quadrático médio.

A primeira função, representada na equação xx, explica como um neurônio de uma camada densa irá funcionar quando recebe determinadas entradas.

### Equação do Neurônio

$$x_j = (\sum_i y_i w_{ji}) + b_j \quad (6.2)$$

Sendo que:

- $x_j$ : representa a saída de um neurônio  $j$ ;
- $y_i$ : representa a entrada do neurônio  $j$ , a qual é resultado da saída de um neurônio  $i$  da camada anterior;
- $w_{ij}$ : representa o peso da conexão entre o neurônio  $i$  com o neurônio  $j$ ;
- $b_j$ : representa o viés do neurônio  $j$ .

Caso você leitor decida olhar o artigo original, verá que os autores utilizaram notações diferentes na fórmula do neurônio, eles utilizam algo na forma  $x_i = \sum_j y_j w_{ji}$ ,

sem o viés, mas na prática, eles adicionam o viés como um peso extra que terá valor fixo igual a 1, na prática, ele pode ser tratado com um peso normal, por isso a notação mais simplificada (Rumelhart; Hinton; Williams, 1986).

A segunda fórmula diz respeito a função de ativação que será utilizada, neste caso, Rumelhart, Hinton e Williams (1986) utilizam a sigmoide logística, representada na fórmula xx, mas eles explicam que essa função pode variar conforme o problema, porém recomendam ter uma derivada limitada além de ser não linear, para que o modelo possa aprender de forma mais eficiente.

### Sigmoide Logística

$$y_j = \sigma(x_j) = \frac{1}{1 + e^{x_j}} \quad (6.3)$$

Assim, a sigmoide é a função que irá transformar a saída do neurônio  $x_j$  em uma saída  $y_j$  que vai ser passada para o neurônio da camada seguinte. Além das duas notações da equação do neurônio, existe uma terceira, que já imbuí a função de ativação na equação do neurônio, nesse livro, utilizaremos uma implementação de camadas, em que elas não possuem a função embutida, assim, caso decida que a saída da camada densa deva passar por uma função de ativação, ela deverá ser passada como uma camada separada, sendo algo da forma: camada densa, seguida de camada de ativação.

Por fim, a última equação que os autores discutem para entender a retropropagação é a da função de custo, ou também chamada de erro. No texto, Rumelhart, Hinton e Williams (1986) utilizam a função do erro quadrático médio (MSE), a qual calcula o erro entre a saída atual do modelo e o valor real desejado, depois ela eleva ao quadrado esse valor, e por último, divide por dois. Ela é dada pela equação:

### Erro Quadrático Médio (MSE)

$$E = \frac{1}{2} \sum_c \sum_j (y_{j,c} - d_{j,c})^2 \quad (6.4)$$

Em que:

- $E$ : representa o valor do erro;
- $y_{j,c}$ : representa o valor atual da saída do neurônio  $j$  para o caso  $c$ ;
- $d_{j,c}$ : representa o valor desejado para o neurônio  $j$  para o caso  $c$

Considerando essas equações, Rumelhart, Hinton e Williams (1986) explicam que o objetivo é reduzir o valor do erro  $E$ , para isso, eles aplicam o método do gradiente para

encontrar o ponto de mínimo da função MSE. Para isso, o primeiro passo é diferenciar o valor da função de erro em relação a cada um dos pesos da rede neural, assim, deve-se calcular  $\partial E / \partial y_k$  para um caso específico  $k$ , e depois generalizar a situação. Assim, é possível encontrar uma expressão da forma:

$$\frac{\partial E}{\partial y_k} = \frac{\partial}{\partial y_k} \left[ \frac{1}{2} \sum_c \sum_k (y_{k,c} - d_{j,c}^2) \right]$$

O primeiro passo é suprimir a soma sobre os casos  $c$ , pois, consideramos apenas de um caso específico  $k$ , então a expressão se simplifica para:

$$\frac{\partial E}{\partial y_k} = \frac{\partial}{\partial y_k} \left[ \frac{1}{2} \sum_k (y_{k,c} - d_{j,c}^2) \right]$$

O próximo passo é abrir a soma, para isso, será considerado que existem  $n$  neurônios na camada, assim, a expressão fica:

$$\frac{\partial E}{\partial y_k} = \frac{\partial}{\partial y_k} \left[ \frac{1}{2} \left( (y_1 - d_1)^2 + (y_2 - d_2)^2 + \dots + (y_k + d_k)^2 + \dots + (y_n + d_n)^2 \right) \right]$$

Note que todos os termos que não possuem o índice  $k$ , como a parte  $(y_1 - d_1)^2$ , serão valores constantes, e que estão fora do objetivo de calcular a derivada para o neurônio  $k$ , como eles são constantes, a sua derivada será igual a zero. Com base nisso, é possível obter uma versão ainda mais simplificada, sendo ela:

$$\frac{\partial E}{\partial y_k} = \frac{\partial}{\partial y_k} \left[ \frac{1}{2} (y_k + d_k)^2 \right]$$

Agora, o último passo é aplicar a regra da cadeia na expressão que sobrou para poder calcular a derivada, neste caso, será considerado que  $u = (y_k + d_k)$ , assim a expressão final fica:

$$\frac{\partial E}{\partial y_k} = \frac{1}{2} \cdot 2u \cdot \frac{\partial u}{\partial y_k} = (y_k - d_k) \cdot 1 = (y_k - d_k)$$

Voltando para o índice  $j$  da notação inicial, é possível concluir que o gradiente do erro (para a função MSE) em relação a uma saída específica de um neurônio é dado pela diferença da saída da unidade pelo resultado desejado, ou seja

$$\frac{\partial E}{\partial y_j} = y_j - d_j$$

O próximo passo proposto pelos autores, consiste em calcular o gradiente do erro em relação a entrada do neurônio  $j$ , para entender como o erro total muda em relação a uma variação na entrada do neurônio. Para isso, é preciso encontrar então a expressão  $\partial E / \partial x_j$  (Rumelhart; Hinton; Williams, 1986).

Assim, a primeira etapa é aplicar a regra da cadeia, pois, como a entrada do neurônio  $x_j$  não aparece diretamente na equação do erro, é preciso aplicar a regra da cadeia, derivando o erro em relação a saída do neurônio  $y_j$  e multiplicando esse resultado pela derivada da saída do neurônio em relação a sua entrada, assim, a expressão inicial é:

$$\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial x_j}$$

Perceba duas coisas nessa expressão, a primeira é que o primeiro termo é resultante o primeiro que foi calculado e desenvolvido na expressão ??, com isso, é possível substituir esse termo na expressão por  $y_j - d_j$ . A segunda informação, é de que o termo  $\partial y_j / \partial x_j$  é justamente a derivada da função de ativação em relação a sua entrada, ou seja,  $\sigma'(x_j)$ , assim, a derivada da sigmoide será dada por:

$$\frac{dy_j}{dx_j} = y_j(1 - y_j)$$

Sendo assim, a expressão final fica:

$$\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j} (1 - y_j)$$

Note que é possível generalizar essa expressão, dessa forma o gradiente do erro da entrada de um neurônio é o gradiente do erro da saída do neurônio multiplicado pela derivada da função de ativação  $\sigma'$  em relação a sua entrada  $x_j$ .

#### Cálculo do Gradiente do Erro em Relação à Entrada de um Neurônio

$$\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j} \cdot \sigma'(x_j) \quad (6.5)$$

em que  $\sigma'(x_j)$  representa a derivada de uma função de ativação qualquer, neste caso representa a sigmoide logística. Mas pode ser outra função, como a tangente hiperbólica ou a ReLU. Um ponto importante a ser destacado, é que como a retropropagação trabalha com o método do gradiente, as derivadas são parte essencial do algoritmo, utilizar funções de ativação que possuem derivadas complexas, ou que não podem ser derivadas em grande parte do seu domínio, acabam por dificultar o algoritmo. Caso a função possua uma derivada complexa, o cálculo do gradiente irá demorar mais, pois levará uma quantidade maior de operações para computar o valor da derivada.

O próximo passo vai mais além ainda, agora, como Rumelhart, Hinton e Williams (1986) explicam, o seu objetivo é entender como o gradiente muda em relação a um peso  $w_{ij}$  específico da rede neural, para isso, é preciso calcular a expressão  $\partial E / \partial w_{ij}$ , note mais uma vez que o peso  $w_{ij}$  não aparece diretamente na equação do erro, assim, é necessário mais uma vez aplicar a regra da cadeia para calcular essa derivada. Assim, a expressão inicial é:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial x_j} \cdot \frac{\partial x_j}{\partial w_{ij}}$$

Note que o primeiro termo da expressão é justamente o que foi calculado na equação 6.5, assim, é preciso calcular apenas o segundo termo,  $\partial x_j / \partial w_{ij}$ , ou seja, a derivada do neurônio em relação a um peso específico. Assim, tem-se a expressão inicial:

$$\frac{\partial x_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \left[ \left( \sum_i y_i \cdot w_{ji} \right) + b_j \right]$$

Note que o viés, dado por  $b_j$  é uma constante, e por isso, não depende do peso  $w_{ij}$ , então a sua derivada será igual a zero, isso nos permite simplificar a expressão para:

$$\frac{\partial x_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \left[ \sum_i y_i \cdot w_{ji} \right]$$

Em seguida, é possível abrir a soma, assim, temos a expressão:

$$\frac{\partial x_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} [(y_1 \cdot w_{j1}) + (y_2 \cdot w_{j2}) + \dots (y_i \cdot w_{ji}) + \dots (y_n \cdot w_{jn})]$$

Note que para qualquer termo que o índice não é  $w_{ji}$  a derivada será igual a zero, pois eles serão constantes em relação ao peso  $w_{ji}$ , dessa forma, é possível simplificar mais uma vez a expressão obtendo:

$$\frac{\partial x_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} [y_i \cdot w_{ji}]$$

Como  $y_i$  é uma constante em relação a  $w_{ij}$  a derivada final será dada por:

$$\frac{\partial x_j}{\partial w_{ij}} = y_i$$

Agora é possível voltar para a expressão inicial, obtendo então a expressão final para o gradiente do erro em relação a um peso específico da rede neural, sendo ela:

#### Cálculo do Gradiente do Erro em Relação a um Peso de um Neurônio

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial x_j} \cdot y_i = \frac{\partial E}{\partial y_j} \cdot \sigma'(x_j) \cdot y_i \quad (6.6)$$

Assim, é possível concluir que o gradiente do erro em relação a um peso específico de um neurônio é dado pelo gradiente do erro da saída do neurônio multiplicado pela derivada da função de ativação em relação a sua entrada e, por fim, multiplicado pela entrada do neurônio.

O próximo passo não está no artigo original, ele envolve entender como o erro varia em relação a um viés de um neurônio específico, ou seja, é preciso calcular a expressão

$\partial E / \partial b_j$ . Note que mais uma vez o viés  $b_j$  não aparece diretamente na equação do erro, ele está dentro da equação do neurônio, assim, novamente é preciso aplicar a regra da cadeia para encontrar essa derivada, assim, a expressão inicial é dada por:

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial x_j} \cdot \frac{\partial x_j}{\partial b_j}$$

A primeira expressão,  $\partial E / \partial x_j$ , já foi calculada na equação 6.5, assim, é preciso focar apenas na segunda parte da expressão,  $\partial x_j / \partial b_j$ , que avalia como a saída do neurônio  $x_j$  varia em relação ao viés  $b_j$ . Com base, nisso, é possível chegar na expressão:

$$\frac{\partial E}{\partial b_j} = \frac{\partial}{\partial b_j} \left[ \left( \sum_k y_k \cdot w_{jk} \right) + b_j \right]$$

Note que os termos que não possuem o viés, como a parte que faz a soma da multiplicação das entradas pelos pesos,  $\sum_l y_k \cdot w_{jk}$ , são constantes em relação ao viés, portanto, a sua derivada será igual a zero, de tal forma que é possível simplificar a expressão para:

$$\frac{\partial E}{\partial b_j} = \frac{\partial}{\partial b_j} [b_j]$$

Com base nessa expressão, é possível concluir que a derivada da saída do neurônio em relação ao viés é igual a 1. Assim, voltando para a expressão inicial, temos que o gradiente do erro em relação ao viés de um neurônio específico é dado pelo gradiente do erro da entrada total desse neurônio, ou seja:

#### **Cálculo do Gradiente do Erro em Relação a um Viés de um Neurônio**

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial x_j} \cdot 1 = \frac{\partial E}{\partial y_j} \cdot \sigma'(x_j) \quad (6.7)$$

##### **6.2.1 Utilizando o Gradiente Descendente para Atualizar os Pesos e Vieses**

Já que é possível calcular o gradiente do erro em relação a um peso específico de um neurônio, e também em relação a um viés específico de um neurônio, o próximo passo proposto pelos autores é utilizar o método do gradiente como uma forma de atualizar os pesos (e também os vieses no nosso caso) da rede neural, de forma a minimizar o valor do erro com pequenas atualizações em cada um desses parâmetros (Rumelhart; Hinton; Williams, 1986).

Note que o método do gradiente, explicado na seção anterior diz que para atualizar um parâmetro, é preciso pegar o valor atual do parâmetro, e subtrair dele o valor do gradiente multiplicado pelo tamanho do passo (ou taxa de aprendizado). dessa forma, a regra de atualização para um peso específico é dada pela equação 6.8

**Regra de Atualização de um Peso Através do Método do Gradiente**

$$w_{t+1} = w_t - \epsilon \frac{\partial E}{\partial w} \quad (6.8)$$

Em que:

- $w_{t+1}$  representa o valor do peso atualizado após o incremento do método do gradiente;
- $w_t$  representa o valor inicial do peso na iteração  $t$ ;
- $\epsilon$  representa o tamanho do passo / taxa de aprendizado;
- $\partial E / \partial w$  representa o gradiente do erro em relação ao peso.

Analogamente, a regra de atualização de um viés pelo método do gradiente é dada pela equação 6.9

**Regra de Atualização de um Viés Através do Método do Gradiente**

$$b_{j,t+1} = b_{j,t} - \epsilon \frac{\partial E}{\partial b_j} \quad (6.9)$$

Neste caso,  $b_{j,t+1}$  representa o valor final de um viés  $j$  após o incremento do método e  $b_{j,t}$  o valor inicial do viés  $j$ .

Um ponto a ser destacado, é que quando estiver trabalhando com uma camada densa de neurônios, você não irá lidar com um neurônio específico, e sim como uma conjunto inteiro deles. Dessa forma, eles estarão distribuídos de forma vetorizada, que terá um vetor de vieses, uma matriz de pesos e um vetor de neurônios. Assim, as expressões eq:regra-de-atualizacao-do-vetor-de-pesos-atraves-do-metodo-do-gradiente e 6.11, resumem a as regras de atualização de pesos e vieses, respectivamente, para casos em que estiver lidando com um conjunto de vetores ao invés de somente um dado.

**Regra de Atualização do Vetor de Pesos Através do Método do Gradiente**

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \epsilon \nabla_{\mathbf{w}} E \quad (6.10)$$

**Regra de Atualização do Vetor de Vieses Através do Método do Gradiente**

$$\mathbf{b}_{t+1} = \mathbf{b}_t - \epsilon \nabla_{\mathbf{b}} E \quad (6.11)$$



## 6.2.2 Entendendo Como o Gradiente É Propagado ao Longo de Muitas Camadas

# 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

# 6.4 Otimizadores Modernos Baseados em Gradiente

## 6.4.1 Nesterov

### 6.4.1.1 Implementação em Python

## 6.4.2 AdaGrad

### 6.4.2.1 Implementação em Python

## 6.4.3 RMSProp

### 6.4.3.1 Implementação em Python

## 6.4.4 Adam

### 6.4.4.1 Implementação em Python

## 6.4.5 Nadam

### 6.4.5.1 Implementação em Python

# 6.5 O Método de Newton: Indo Além do Gradiente

## 6.5.1 Implementação em Python



## 7 Funções de Ativação Sigmoidais

### 7.1 Teoremas da Aproximação Universal

Pense que você tem uma função matemática, como  $f(t) = 40t + 12$ , a qual representa o deslocamento em quilômetros de um carro em uma cidade, onde  $t$  é o tempo em horas. Caso você queira encontrar o valor de deslocamento quando o carro tiver andando por 3 horas, basta substituir a variável  $t$  por 3 e resolver a expressão. Assim temos:

$$\begin{aligned} f(t) &= 40t + 12 \\ f(3) &= 40 \cdot 3 + 12 = 132km \end{aligned} \quad \left. \vphantom{\begin{aligned} f(t) &= 40t + 12 \\ f(3) &= 40 \cdot 3 + 12 = 132km \end{aligned}} \right\} \text{Quando } t = 3$$

Esse cenário é o mais comum quando estamos estudando, contudo existe um segundo cenário que também é possível de acontecer. Isso ocorre quando temos um conjunto de pontos e, com base neles, queremos encontrar uma função que descreva o comportamento desses pontos.

Pense que temos os pontos dispostos no gráfico da figura 3 e queremos encontrar uma reta que tente passar o mais próximo de cada um deles.

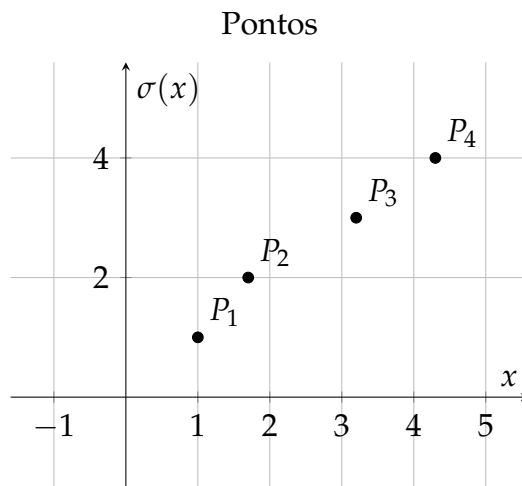


Figura 3 – Conjunto de pontos dispostos no plano cartesiano.

Existe uma técnica que permite que nos façamos isso, ela se chama **regressão linear** (a qual é um dos tópicos discutidos no capítulo 13), e com base nela, é possível dado um conjunto de pontos em um plano, traçar uma reta que se aproxime igualmente de cada um desses pontos. Existem diferentes técnicas de regressão linear, neste caso aplicaremos a dos mínimos quadrados e encontramos a expressão:

$$y = 0.8623x + 0.3011$$

Com base nessa função que encontramos, podemos desenhá-la junto ao gráfico dos pontos e vermos se ela é realmente uma boa aproximação, assim temos a figura 4

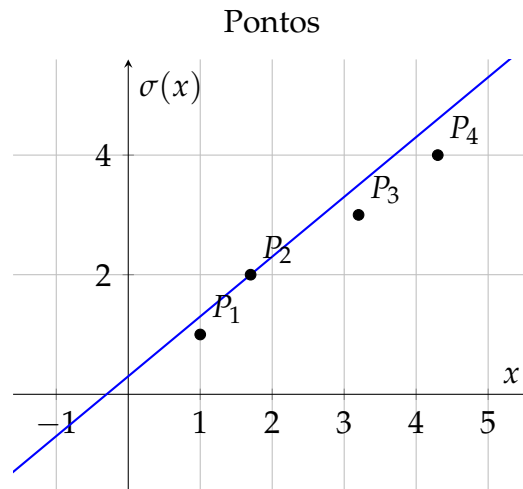


Figura 4 – Conjunto de pontos dispostos no plano cartesiano.

Existem diversas aproximações além da regressão linear, se quisermos, podemos tentar aproximar esses pontos utilizando uma função quadrática, cúbica ou até mesmo exponencial.

Esse tema parece não ter uma conexão com esse capítulo de funções de ativação, mas na realidade, o que muitas das vezes é feito por uma rede neural é justamente esse trabalho de encontrar uma função que aproxima o comportamento de um conjunto de pontos. Só que neste caso, não teremos um conjunto de pontos, vamos ter várias informações em uma base de dados, como imagens de exames médicos ou informações sobre o valor de imóveis e queremos encontrar de alguma forma uma conexão entre esses dados.

Para isso, existe um conjunto de teoremas que servem justamente para provar que uma determinada rede neural criada será capaz de encontrar uma função que descreva o comportamento que você esteja estudando. Eles são os teoremas da aproximação universal.

No livro *Deep Learning*, Goodfellow, Bengio e Courville (2016) dedicam uma seção explicando esses teoremas. Segundo os autores, o teorema da aproximação universal, introduzido Cybenko (1989) para a comunidade científica no texto *Approximation by Superpositions of a Sigmoidal Function*, afirma que uma rede feedforward com uma camada de saída linear e no mínimo uma camada oculta com qualquer função que possui a propriedade de "esmagamento", como a sigmoide logística, é capaz de aproximar qualquer função mensurável de Borel de um espaço de dimensão finita para outro com qualquer quantidade de erro diferente de zero desejada desde que essa rede neural possua unidades ocultas suficientes.

Para entendermos esse teorema, devemos primeiro entender o conceito de mensurabilidade de Borel, segundo Goodfellow, Bengio e Courville (2016) uma função contínua em um subconjunto fechado e limitado de  $\mathcal{R}^N$  é mensurável por Borel. Assim esse tipo de função pode ser aproximada por uma rede neural. Além disso, os autores ressaltam que por mais que o teorema original tenha conseguido provar apenas para as funções que saturam tanto para termos muito negativos ou termos muito positivos, diversos outros autores como Leshno *et al.* (1993), no texto *Multilayer feedforward networks with a nonpolynomial activation function can approximate any function*, foram capazes de provar que o teorema da aproximação universal pode funcionar para outras funções, no caso de Leshno, eles provaram para funções não polinomiais, como a Rectified Linear Unit (ReLU), a qual é o tema central do capítulo 8.

Basicamente os teoremas da aproximação universal reforçam o uso de funções de ativação para permitir que as redes neurais resolvam os problemas propostos por meio da aproximação de uma função. Isso acontece porque essas funções introduzem a não linearidade para a rede, como nos vimos na equação do neurônio de uma rede neural, uma RNA é composta por diferentes camadas de neurônios que são capazes de pegar valores de entrada, multiplicar por um peso dado e somar com um viés, esse resultado passa então por uma função de ativação.

$$y = x \cdot w + b$$

Se nos tivéssemos uma rede neural em que os neurônios não possuísem uma função de ativação, ou, fosse uma função linear, mesmo juntando todas essas camadas de neurônios que trazem expressões lineares, a junção disso, ainda seria uma expressão linear. Mas quando introduzimos uma função não linear, como a sigmoide, o teorema da aproximação universal, nos garante que somos capazes de encontrar qualquer função que estivermos procurando, desde que ela seja mensurável de Borel.

## 7.2 Exemplo Ilustrativo: Empurrando para extremos

Imagine que você está trabalhando para uma empresa na área de marketing e precisa analisar como foi a recepção do público para um novo produto anunciado. Para isso, você ficou responsável por classificar os comentários do público sobre esse produto. Você precisa colocar eles em duas categorias: avaliação positiva ou negativa. Então você precisa ler cada comentário e colocar ele em uma dessas categorias.

No começo foi fácil, mas depois de um tempo foi ficando repetitivo, então você teve a ideia de automatizar esse processo. Assim, você decide criar um diagrama de uma “caixa-preta” responsável por classificar automaticamente esses comentários da mesma forma que estava fazendo. Essa “caixa” irá receber uma entrada, neste caso,

o texto do comentário sobre o produto, e irá retornar uma saída, uma classificação positiva ou negativa sobre o comentário.

Na matemática, as funções do tipo sigmoide são excelentes para esse tipo de problema, pois possuem uma propriedade muito interessante: dado um valor de entrada, elas são capazes de "empurrar" esse valor para dois diferentes extremos. No caso da sigmoide logística, a função que dá nome a essa família, ela é capaz de empurrar essa entrada para valores próximos de zero ou um. Se nós consideramos que zero é uma avaliação negativa e um é uma avaliação positiva, essa função se torna perfeita para resolver o seu problema de classificar comentários.

### 7.3 A Sigmoide Logística

Por mais que a sigmoide hoje em dia seja bem comum em redes neurais, seu uso não começou nesse cenário. A sigmoide tem suas origens a análise de crescimento populacional e demografia, ela não surgiu em um artigo em específico, sendo mais uma evolução presente em vários artigos do matemático belga Pierre François Verhulst dentre os anos de 1838 e 1847. Contudo, existe um artigo desse matemático, intitulado "**Recherches mathématiques sur la loi d'accroissement de la population**" (Pesquisas matemáticas sobre a lei de crescimento da população), em que Verhulst propõem a função logística como um modelo para descrever o crescimento populacional, levando em consideração a capacidade de suporte de um ambiente, isso gerou a curva em "S" característica da sigmoide. Contudo, foi somente no próximo século, que sigmoide passou a ser utilizada na área da ciência da computação.

### 7.4 Contexto Histórico: Popularização da Sigmoide em Redes Neurais

A primeira família de funções que iremos conhecer são as sigmodais, sendo a principal delas, e que dá nome a essa família a sigmoide logística. Para isso, vamos entender o contexto em que elas se popularizaram.

Nos anos 1980, estavam ocorrendo mudanças com as funções que eram utilizadas para construir uma rede neural, um desses motivos foi a introdução da retropropagação pelos pesquisadores David Rumelhart, Geoffrey Hinton e Ronald Williams. A retropropagação era uma técnica que permitia que um modelo aprendesse com base nos seus erros, ajustando automaticamente os seus parâmetros em busca de conseguir uma melhor acurácia.

Além disso, na pesquisa que introduz a retropropagação, "**Learning representations by back-propagating errors**" de 1986, os cientistas propõem o uso da função

sigmoide logística como uma das candidatas para ser utilizada junto com a retropropagação como uma função de ativação. Como justificativa, eles citam o fato dela ser uma função que é capaz de introduzir a não-linearidade para o modelo, permitindo que ele aprenda padrões mais complexos, e também por possuir uma derivada limitada.

Mas esse não foi o único fator que fez com que a sigmoide e sua família se tornassem funções populares para a época. Pouco antes da criação da retropropagação, na década passada, haviam cientistas estudando o comportamento dos neurônios humanos como inspiração para a criação de redes neurais artificiais. Um exemplo desse caso foi o dos cientistas Wilson Hugh e Jack Cowan, em 1972 eles publicaram um artigo intitulado **"excitatory and Inhibitory interactions in localized populations of model neurons"**, em que buscam estudar como os neurônios respondiam a determinados estímulos.

No artigo, Hugh e Cowan buscam analisar o comportamento de populações localizadas de neurônios excitatórios (denotados pela função  $E(t)$ ) e inibitórios (representados por  $I(t)$ ) e como as duas interagem entre si. Para isso, eles utilizam como variável a proporção de células em uma subpopulação que dispara/reage por unidade de tempo. Para modelar essa atividade eles fizeram o uso de uma variação da função sigmoide, representada pela expressão ??, que era capaz de descrever o comportamento dos neurônios a certos estímulos.

#### Neurônio de Wilson e Cowan

$$S(x) = \frac{1}{1 + e^{-a(x-\theta)}} - \frac{1}{1 + e^{a\theta}} \quad (7.1)$$

Nessa equação, o parâmetro  $a$  representa a inclinação, a qual foi ajustada para passar pela origem ( $S(0) = 0$ ) e  $\theta$  é o limiar. Para o plotar o gráfico da figura 5, foi utilizado os mesmos valores escolhidos pelos cientistas na pesquisa, assim  $a = 1.2$  e  $\theta = 2.8$ .

Os cientistas demonstram que a população de neurônios reage de formas distintas quando sofrem determinados estímulos. Os níveis baixos de excitação não conseguem ativar a população, porém, existe uma região de alta sensibilidade, na qual pequenos aumentos no estímulo geram um grande aumento na atividade. Além disso, existe um terceiro nível, o de saturação, em que níveis muito altos de estímulos são capazes de ativar todas as células e a partir disso, a resposta da população atinge o comportamento de uma função constante, indicando que ela saturou. Ao juntar todos esses três níveis, temos a famosa curva em "S", característica da função sigmoide.

Com isso, ao considerarmos esses dois cenários: a criação da retropropagação e busca na natureza para inspiração na criação de redes neurais. A sigmoide, junto com a sua família, se tornaram funções muito populares para a época, estando presentes em várias redes neurais criadas. Um desses exemplos é a rede de Elman, um tipo de rede neural recorrente criada para aprender e representar estruturas em dados sequenciais.

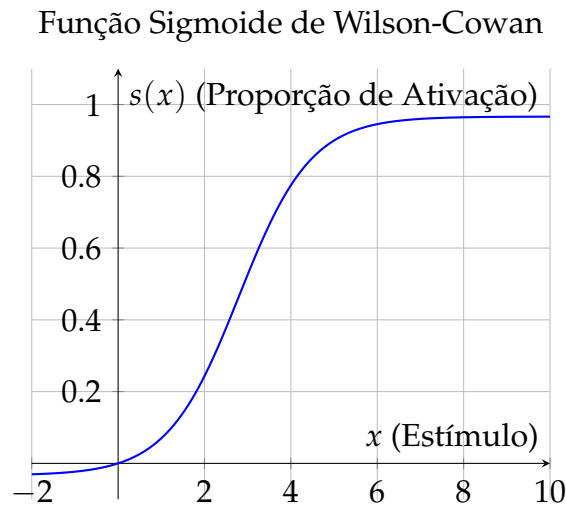


Figura 5 – Gráfico da função sigmoide conforme proposta por Wilson e Cowan (1972), com parâmetros de exemplo  $a = 1.2$  e  $\theta = 2.8$ .

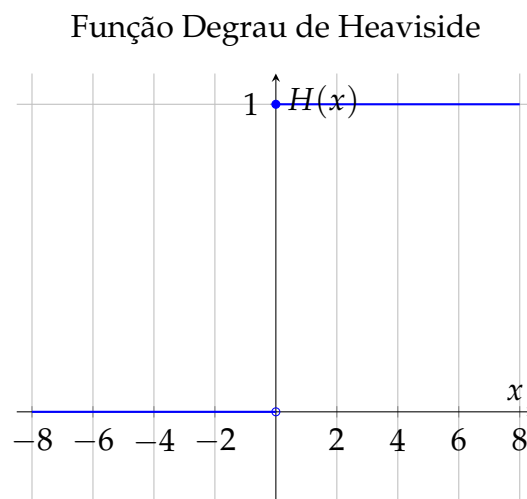


Figura 6 – Gráfico da função degrau de Heaviside.

Elman cita em seu estudo "**Finding structure in time**" de 1990 que era ideal o uso de uma função de ativação com valores limitados entre zero e um, um cenário ideal para o uso da sigmoide.

Cabe destacar também que, as sigmodais não foram as primeiras funções de ativação a serem utilizadas na criação de uma rede neural. Nesse cenário de redes neurais, existem sempre funções que são mais populares, e que com o tempo e o surgimento de novas pesquisas, são deixadas de lado para novas funções mais interessantes.

Uma função que era muito utilizada era a heavside, ou degrau em português. Ela inclusive esteve presente na produção da rede neural Perceptron criada por Frank Rosenblatt e introduzida para a comunidade científica no artigo "**The Perceptron: A probabilistic model for information storage and organization in the brain**" em 1958.



Quando a comparamos a degrau unitário com a sigmoide, notamos uma diferença crucial, a sigmoide é uma função contínua em todos os pontos, podemos dizer que para desenhar seu gráfico não precisamos tirar o lápis do papel nenhuma vez, algo que não ocorre com a heavside. Além disso, a derivada da heavside é zero em quase todos os seus pontos, por esse motivo, ela impossibilitava a retropropagação do erro, uma vez que quando fossemos calcular o gradiente para uma parte de um modelo que usasse essa função, ele provavelmente seria zero.

## 7.5 Expressões Matemáticas

### Sigmoide Logística

$$\sigma(z_i) = \frac{1}{1 + e^{-z_i}} \quad (7.2)$$

### Derivada da sigmoide logística

$$\frac{d}{dz_i} \sigma(z_i) = \frac{e^{-z_i}}{(1 + e^{-z_i})^2} \quad (7.3)$$

### 7.5.1 Implementação em Python

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

```

1 import numpy as np
2
3 class Sigmoid(Layer):
4     def __init__(self):
5         super().__init__()
6         self.input = None
7         self.sigmoid = None
8
9     def forward(self, input_data):
10        self.input = input_data
11        self.sigmoid_output = 1/ (1 + np.exp(-input_data))
12        return self.sigmoid_output
13
14    def backward(self, grad_output):
15        sigmoid_grad = self.sigmoid_output * (1 - self.
sigmoid_output)
16        return grad_output * sigmoid_grad, None

```

## 7.6 Tangente Hiperbólica

Assim como a função sigmoide, a tangente hiperbólica não possui suas origens voltadas para o uso em redes neurais. Neste caso, um dos matemáticos que ficou reconhecido por criar a notação das funções hiperbólicas, seno, cosseno e tangente, foi o suíço Johann Heinrich Lambert no trabalho de 1769 "**Mémoire sur quelques propriétés remarquables des quantités transcendentes circulaires et logarithmiques**" (Memória sobre algumas propriedades notáveis de grandezas transcendentais circulares e logarítmicas), em que provou que muitas das identidades trigonométricas possuíam suas equivalentes hiperbólicas.

Passando mais de dois séculos, a tangente hiperbólica já estava sendo utilizada em diversas redes neurais, ela inclusive fez parte da primeira rede neural convolucional criada, estando presente na Le-Net-5, uma rede neural criada para identificar e classificar imagens de cheques em caixas eletrônicos. No artigo acadêmico "**Gradient-based learning applied to document recognition**" de 1998, os cientistas Yann LeCun, Léon Bottou, Yousha Bengio e Patrick Haffner explicam a criação dessa rede além de destacar suas métricas alcançadas.

Semelhante a sigmoide, a tangente hiperbólica possui várias propriedades parecidas. Como afirma Lederer em "**Activation Functions in Artificial Neural Networks: A Systematic Overview**" a tangente hiperbólica é infinitamente diferenciável, sendo uma versão escalada e rotacionada da sigmoide logística. Assim, como podemos ver no gráfico da figura 8, ela é uma função que está centrada em zero, e seus valores variam agora em um intervalo de -1 a 1, diferente da sigmoide, que varia somente de 0 a 1.

Podemos escrever a tangente hiperbólica utilizando a definição de tangente, que é o quociente a função seno com a função cosseno, só que neste caso usaremos as funções hiperbólicas. Assim temos a expressão ?? e seu gráfico 8.

### Tangente Hiperbólica

$$\tanh(z_i) = \frac{\sinh(z_i)}{\cosh(z_i)} = \frac{e_i^z - e^{-z_i}}{e_i^z + e^{-z_i}} \quad (7.4)$$

### Derivada da tangente hiperbólica

$$\frac{d}{dz_i} \tanh(z_i) = \text{sech}^2(z_i) \quad (7.5)$$

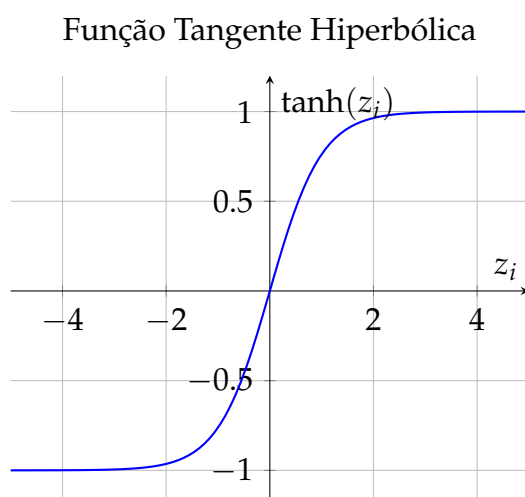


Figura 7 – Gráfico da tangente hiperbólica.

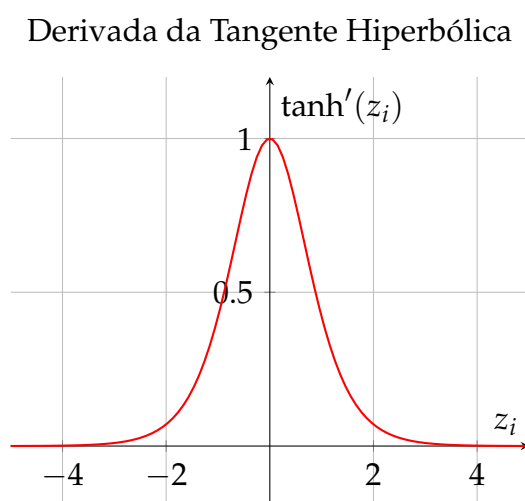


Figura 8 – Gráfico da derivada da tangente hiperbólica.

### 7.6.1 Implementação em Python

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

```

1 import numpy as np
2 from layers.base import Layer # Assuming your base class is
   here
3
4 class Tanh(Layer):
5     def __init__(self):
6         super().__init__()
7         self.input = None
8         self.tanh_output = None
9
10    def forward(self, input_data):
11        self.input = input_data
12        self.tanh_output = np.tanh(self.input)
13        return self.tanh_output
14
15    def backward(self, grad_output):
16        tanh_grad = 1 - self.tanh_output**2
17        return grad_output * tanh_grad, None

```

## 7.7 Softsign: Uma Sigmoidal Mais Barata

A próxima função que vamos conhecer é a softsign, diferente da tangente hiperbólica e da sigmoide que tiveram suas origens em outros campos diferentes da ciência da computação, a softsign foi criada com o intuito de ser trabalhada em uma rede neural. Ela foi introduzida no artigo "**A Better Activation Function for Artificial Neural Networks**" de 1993, do cientista D.L. Elliott. No texto ele propõem a softsign como uma alternativa para as funções sigmodais tradicionais.

Como podemos ver no seu gráfico ??, ela possui o formato em "S" característico das sigmodais além de ser centrada em zero como a tangente hiperbólica. Além disso, como Elliot destaca em seu texto, ela é uma função que é diferenciável em toda a reta possuindo também a mesma propriedade das outras sigmodais de empurrar os valores de entrada para os seus extremos. Podemos notar também pelo seu gráfico que ela também uma função contínua, suave e não linear

Contudo, a principal diferença dela com as outras sigmodais está na sua fórmula, como podemos ver na equação ?? ela não utiliza nenhum exponencial para compor sua função. Isso faz com que ela seja uma função mais "barata" em termos de poder com-

putacional para ser implementada em redes neurais. Assim, podemos obter resultados parecidos porem utilizando cálculos menos complexos e com isso teremos redes mais rápidas de serem treinadas. Um comparativo com as funções sigmodais desse texto e como elas reagem poderá ser visto em uma seção futura.

Com base em sua expressão, também podemos plotar o seu gráfico na figura ??.

### Softsign

$$\text{softsign}(z_i) = \frac{z_i}{1 + |z_i|} \quad (7.6)$$

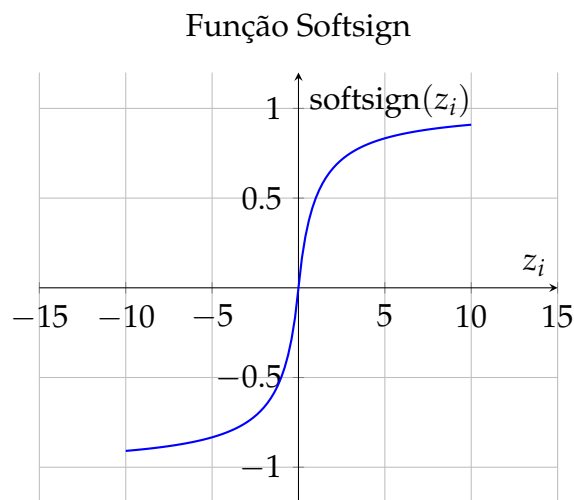


Figura 9 – Gráfico da função Softsign.

### Derivada da softsign

$$\frac{d}{dz_i} \text{softsign}(z_i) = \frac{1}{(1 + |z_i|)^2} \quad (7.7)$$

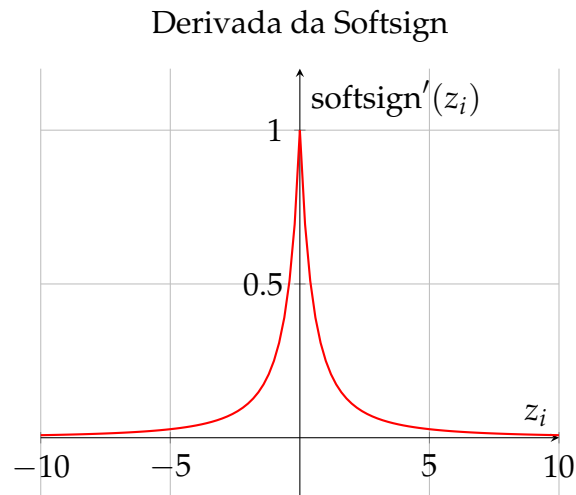


Figura 10 – Gráfico da derivada da função Softsign.

### 7.7.1 Implementação em Python

#### Bloco de Código : Classe completa do função de ativação Softsign

```

1 from layers.base import Layer
2 import numpy as np
3
4 class Softsign(Layer):
5     def __init__(self):
6         super().__init__()
7         self.input = None
8
9     def forward(self, input_data):
10        self.input = input_data
11        return self.input / (1 + np.abs(self.input))
12
13    def backward(self, grad_output):
14        grad = (1 / (1 + np.abs(self.input))**2)
15        return grad_output * softsign_grad, None

```

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

### Hard Sigmoid

$$\text{hard sigmoid}(z_i) = \begin{cases} 0 & \text{se } z_i < -3 \\ z_i/6 + 0.5 & \text{se } -3 \leq z_i \leq 3 \\ 1 & \text{se } z_i > 3 \end{cases} \quad (7.8)$$

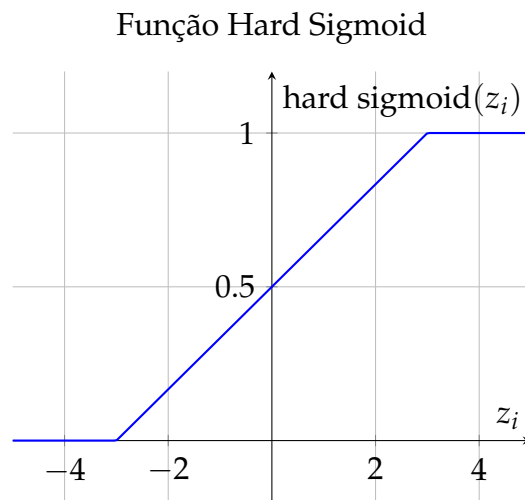


Figura 11 – Gráfico da função hard sigmoid.

### Derivada da Hard Sigmoid

$$\frac{d}{dz_i} \text{hard sigmoid}(z_i) = \begin{cases} 0 & \text{se } z_i < -3 \\ 1/6 & \text{se } -3 < z_i < 3 \\ 0 & \text{se } z_i > 3 \end{cases} \quad (7.9)$$

### Hard Tanh

$$\text{hard tanh}(z_i) = \begin{cases} -1 & \text{se } z_i < -1 \\ z_i & \text{se } -1 \leq z_i \leq 1 \\ 1 & \text{se } z_i > 1 \end{cases} \quad (7.10)$$

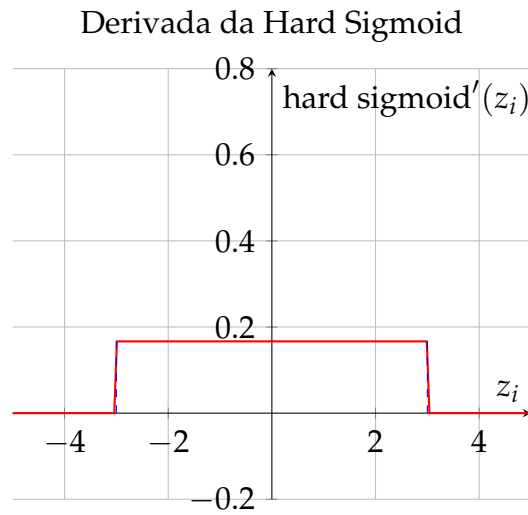


Figura 12 – Gráfico da derivada da Hard Sigmoid.

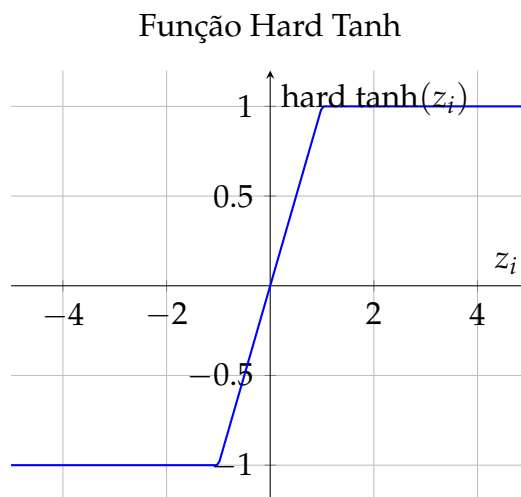


Figura 13 – Gráfico da função Hard Tanh. Fonte: PyTorch.

#### Derivada da Hard Tanh

$$\frac{d}{dz_i} \text{hard tanh}(z_i) = \begin{cases} 0 & \text{se } z_i < -1 \\ 1 & \text{se } -1 < z_i < 1 \\ 0 & \text{se } z_i > 1 \end{cases} \quad (7.11)$$



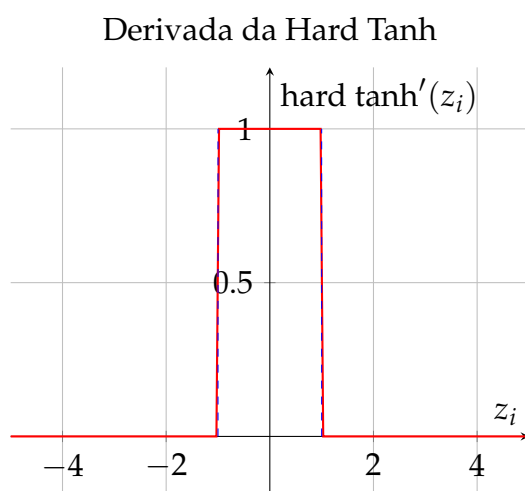


Figura 14 – Gráfico da derivada da Hard Tanh.

### 7.8.1 Implementação em Python

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

```
1 from layers.base import Layer
2 import numpy as np
3
4 class HardSigmoid(Layer):
5
6     def __init__(self):
7         super().__init__()
8         self.input = None
9
10    def forward(self, input_data):
11        self.input = input_data
12
13        output = self.input / 6 + 0.5
14        output = np.clip(output, 0, 1) # A more concise way
15        to handle the bounds
16
17        return output
18
19    def backward(self, grad_output):
20        hard_sigmoid_grad = np.full_like(self.input, 1 / 6)
21
22        hard_sigmoid_grad[self.input < -3] = 0
23        hard_sigmoid_grad[self.input > 3] = 0
24
25        return grad_output * hard_sigmoid_grad, None
```

**Bloco de Código : Classe completa do função de ativação Hard Tanh**

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

## 7.9 O Desaparecimento de Gradientes

### 7.10 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)

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

#### Derivada Rectified Linear Unit (ReLU)

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

h!

Figura 15 – Gráfico da função ReLU.

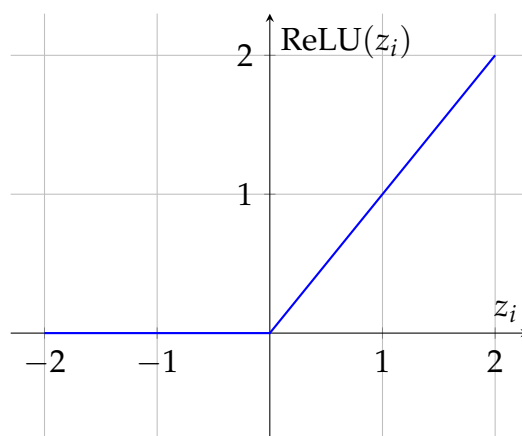
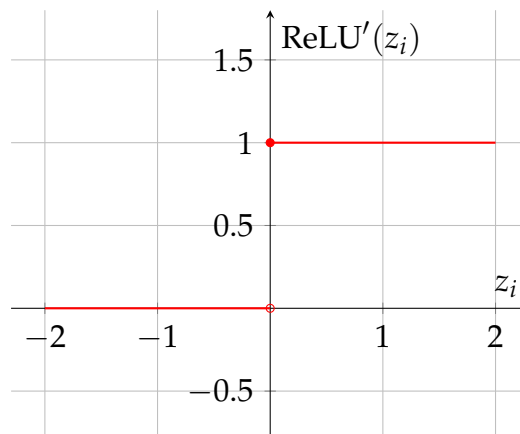


Figura 16 – Gráfico da Derivada da Função ReLU



### 8.2.1 Implementação em Python

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

```

1 import numpy as np
2 from layers.base import Layer
3
4 class ReLU(Layer):
5     def __init__(self):
6         super().__init__()
7         self.input = None
8
9     def forward(self, input_data):
10        self.input = input_data
11        return np.maximum(0, self.input)
12
13    def backward(self, grad_output):
14        relu_grad = (self.input > 0)
15
16        # Apply the chain rule
17        return grad_output * relu_grad, None

```

## 8.3 Dying ReLUs Problem

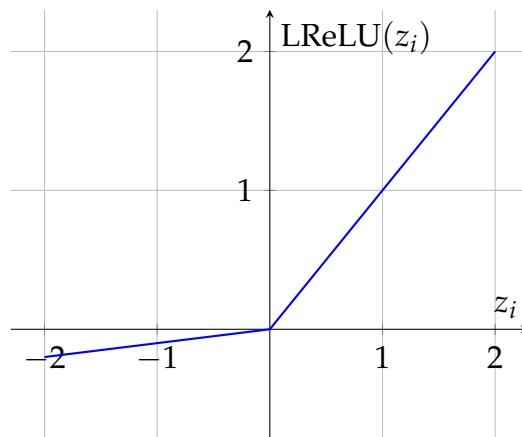
## 8.4 Corrigindo o Dying ReLUs Problem: As Variantes com Vazamento

### 8.4.1 Leaky ReLU

#### Leaky ReLU (LReLU)

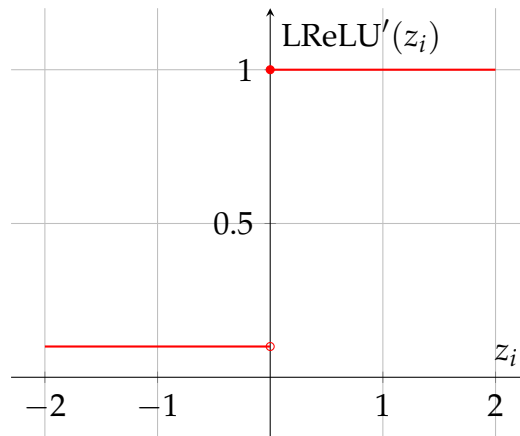
$$\text{LReLU}(z_i) = \begin{cases} z_i, & \text{se } z_i \geq 0 \\ \alpha \cdot z_i, & \text{se } z_i < 0 \end{cases} \quad (8.3)$$

Figura 17 – Gráfico da Função Leaky ReLU (LReLU) com  $\alpha = 0.1$



#### Derivada Leaky ReLU (LReLU)

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

Figura 18 – Gráfico da Derivada da Função Leaky ReLU (LReLU) com  $\alpha = 0.1$ 

## 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

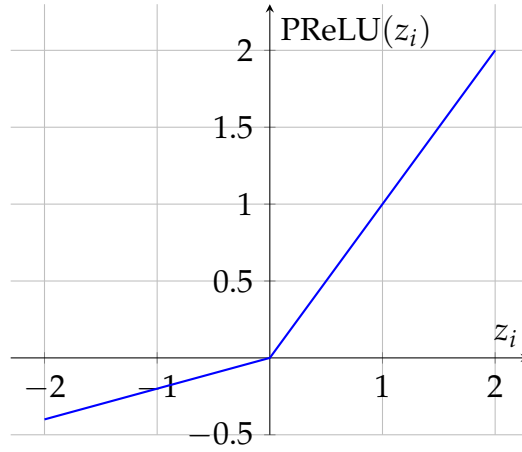
```

## 8.4.2 Parametric ReLU

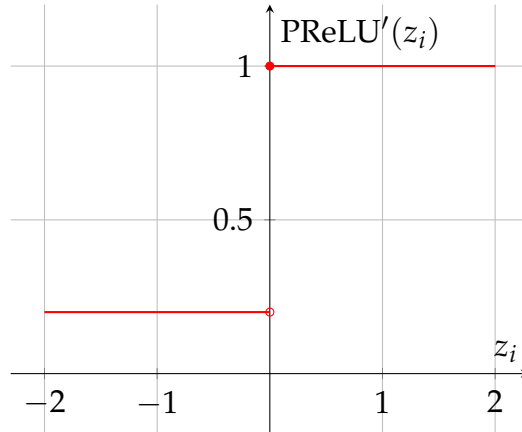
**Parametric ReLU (PReLU)**

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



Figura 19 – Gráfico da Função Parametric ReLU (PReLU) com  $\alpha = 0.2$ **Derivada Parametric ReLU (PReLU)**

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

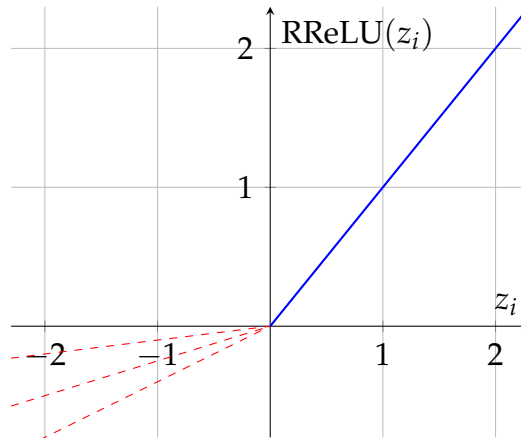
Figura 20 – Gráfico da Derivada da Função Parametric ReLU (PReLU) com ( $\alpha = 0.2$ )

## 8.4.3 Randomized Leaky ReLU

**Randomized Leaky ReLU (RReLU)**

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

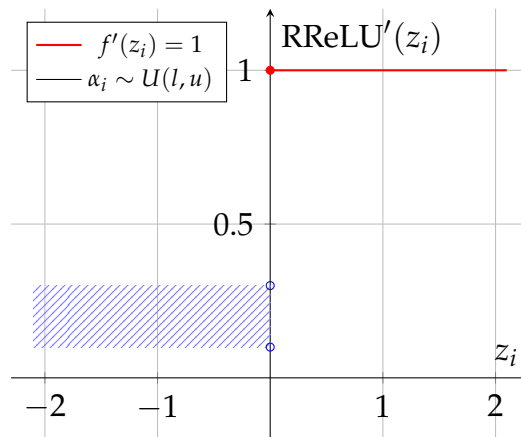
Figura 21 – Gráfico da Função Randomized Leaky ReLU (RReLU) com Diferentes Inclinações Aleatórias para a Parte Negativa



#### Derivada Randomized Leaky ReLU (RReLU)

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

Figura 22 – Gráfico da Derivada da Função Randomized Leaky ReLU (RReLU) com  $l = 0.1, u = 0.3$

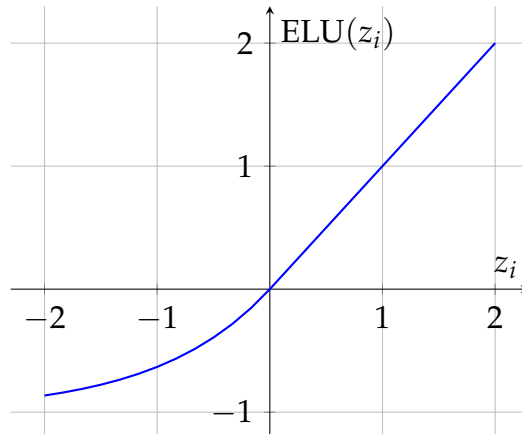


## 8.5 Em Busca da Suavidade

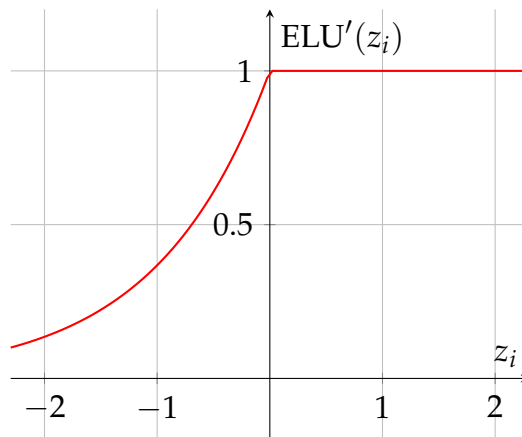
### 8.5.1 Exponential Linear Unit

#### Exponential Linear Unit (ELU)

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

Figura 23 – Gráfico da Função Exponential Linear Unit (ELU) com  $\alpha = 1$ **Derivada Exponential Linear Unit (ELU)**

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

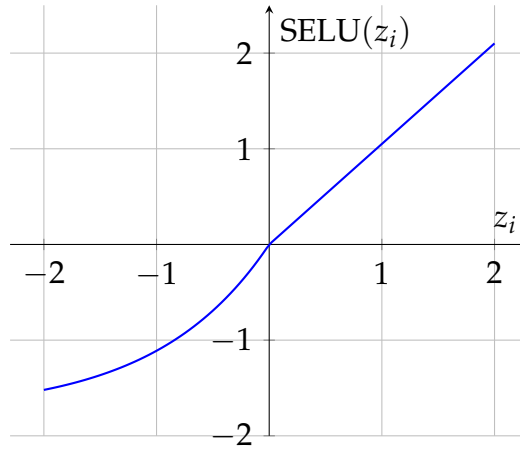
Figura 24 – Gráfico da Derivada da Função Exponential Linear Unit (ELU) com  $\alpha = 1$ 

## 8.5.2 Scaled Exponential Linear Unit

**Scaled Exponential Linear Unit (ELU)**

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

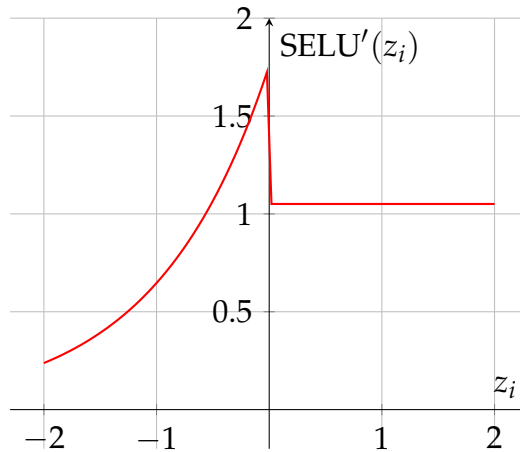
Figura 25 – Gráfico da Função Scaled Exponential Linear Unit (SELU) com  $\alpha \approx 1.67e\lambda \approx 1.05$



#### Derivada Scaled Exponential Linear Unit (ELU)

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

Figura 26 – Gráfico da Derivada da Função Scaled Exponential Linear Unit (SELU) com  $\alpha \approx 1.67, \lambda \approx 1.05$

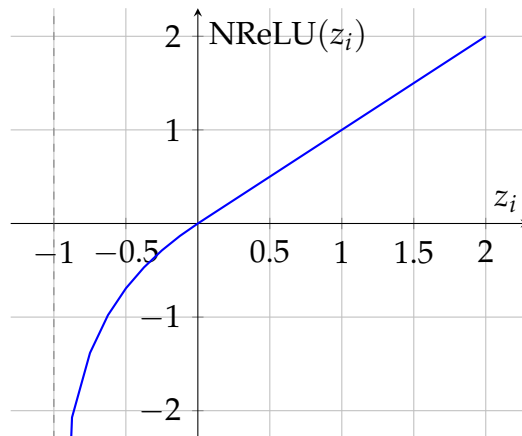


### 8.5.3 Noisy ReLU

#### Noisy ReLU (NReLU)

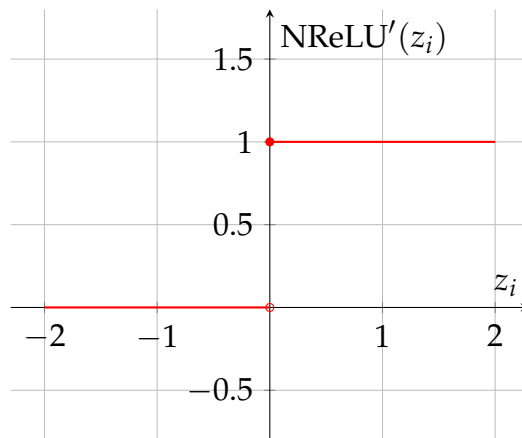
$$\text{NReLU}(z_i) = \begin{cases} 0 & \text{se } z_i \leq 0 \\ z_i + \mathcal{N}(0, \sigma(z_i)) & \text{se } z_i > 0 \end{cases} \quad (8.13)$$

Figura 27 – Gráfico da Função Noisy ReLU (NReLU)

**Derivada Noisy ReLU (NReLU)**

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

Figura 28 – Gráfico da função Backward Pass da Noisy ReLU



## 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 Polinomial

#### 13.3.1 Implementação 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

### 14.1 Exemplo Ilustrativo

### 14.2 Entendendo o Conceito de Árvores

#### 14.2.1 Árvores Binárias

### 14.3 Árvores de Decisão

#### 14.3.1 Implementação em Python

### 14.4 Florestas Aleatórias

#### 14.4.1 Implementação em Python



## 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

### 18.1 Exemplo Ilustrativo

### 18.2 Aprendizado Não Supervisionado: Encontrando Grupos nos Dados

### 18.3 Clusterização Particional: K-Means

### 18.4 Clusterização Hierárquica

### 18.5 Clusterização Baseada em Densidade: DBSCAN



## 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

```
1 import numpy as np
2 from layers.base import Layer
3
4 class Convolution2D(Layer):
5     def __init__(self, input_channels, num_filters,
6         kernel_size, stride=1, padding=0):
7         super().__init__()
8         self.input_channels = input_channels
9         self.num_filters = num_filters
10        self.kernel_size = kernel_size
11        self.stride = (stride, stride) if isinstance(stride,
12            int) else stride
13        self.padding = padding
14
15        kernel_height, kernel_width = self.kernel_size
16        self.kernels = np.random.randn(num_filters,
17            input_channels, kernel_height, kernel_width) * 0.01
18        self.biases = np.zeros((num_filters, 1))
19        self.params = [self.kernels, self.biases]
20        self.cache = None
21
22    def forward(self, input_data):
23        (batch_size, input_height, input_width, input_channels
24        ) = input_data.shape
25        filters, _, kernel_height, kernel_width = self.kernels
26        .shape
27        stride_height, stride_width = self.stride
28
29        pad_config = ((0, 0), (self.padding, self.padding), (
30            self.padding, self.padding), (0, 0))
31        input_padded = np.pad(input_data, pad_config, mode='
32            constant')
33        self.cache = input_padded
```



## 22.3 Camadas de Pooling: Reduzindo a Dimensionalidade

### 22.3.1 Max Pooling

#### 22.3.1.1 Implementação em Python

##### Bloco de Código : Classe completa de MaxPooling2D

```
1 import numpy as np
2 from layers.base import Layer
3
4 class MaxPooling2D(Layer):
5     def __init__(self, pool_size=(2,2), stride=None):
6         super().__init__()
7         self.pool_size = pool_size
8         self.stride = stride if stride is not None else
pool_size
9         self.cache = None
10
11     def forward(self, input_data):
12         (batches, input_height, input_width, channels) =
input_data.shape
13
14         pool_height, pool_width = self.pool_size
15         stride_height, stride_width = self.stride
16
17         output_height = int((input_height - pool_height) /
stride_height) + 1
18         output_width = int((input_width - pool_width) /
stride_width) + 1
19
20         output_matrix = np.zeros((batches, output_height,
output_width, channels))
21         self.cache = np.zeros_like(input_data)
22
23         for b in range(batches):
24             for c in range(channels):
25                 for h in range(output_height):
26                     for w in range(output_width):
27
28                         start_height = h * stride_height
29                         start_width = w * stride_width
30                         end_height = start_height +
pool_height
31                         end_width = start_width + pool_width
32
```





## 22.3.2 Average Pooling

### 22.3.2.1 Implementação em Python

#### Bloco de Código : Classe completa de AveragePooling2D

```
1 import numpy as np
2 from layers.base import Layer
3
4 class AveragePooling2D(Layer):
5     def __init__(self, pool_size=(2, 2), stride=None):
6         super().__init__()
7         self.pool_size = pool_size
8         self.stride = stride if stride is not None else
pool_size
9
10    def forward(self, input_data):
11        (batches, input_height, input_width, channels) =
input_data.shape
12        pool_h, pool_w = self.pool_size
13        stride_h, stride_w = self.stride
14
15        output_height = int((input_height - pool_h) / stride_h
) + 1
16        output_width = int((input_width - pool_w) / stride_w)
+ 1
17
18        output_matrix = np.zeros((batches, output_height,
output_width, channels))
19
20        for b in range(batches):
21            for c in range(channels):
22                for h in range(output_height):
23                    for w in range(output_width):
24                        start_h = h * stride_h
25                        start_w = w * stride_w
26                        end_h = start_h + pool_h
27                        end_w = start_w + pool_w
28
29                        pooling_window = input_data[b, start_h
:end_h, start_w:end_w, c]
30                        output_matrix[b, h, w, c] = np.mean(
pooling_window) # Changed to mean
31
32        return output_matrix
33
```

### 22.3.3 Global Average Pooling

#### 22.3.3.1 Implementação em Python

##### Bloco de Código : Classe completa de GlobalAveragePooling2D

```
1 import numpy as np
2 from layers.base import Layer
3
4 class GlobalAveragePooling2D(Layer):
5     def __init__(self):
6         super().__init__()
7         self.input_shape = None
8
9     def forward(self, input_data):
10         self.input_shape = input_data.shape
11
12         output = np.mean(input_data, axis=(1, 2), keepdims=
13             True)
14
15         return output
16
17     def backward(self, output_gradient):
18         _, input_h, input_w, _ = self.input_shape
19
20         distributed_grad = output_gradient / (input_h *
21             input_w)
22
23         upsampled_grad = np.ones(self.input_shape) *
24             distributed_grad
25
26         return upsampled_grad, None
```

## 22.4 Camada Flatten: Achatando os Dados

### 22.4.1 Implementação em Python

#### Bloco de Código : Classe completa de Flatten

```
1 import numpy as np
2 from layers.base import Layer
3
4 class Flatten(Layer):
5     def __init__(self):
6         super().__init__()
7         self.input_shape = None
8
9     def forward(self, input_data):
10         self.input_shape = input_data.shape
11
12         flatten_output = input_data.reshape(input_data.shape
13 [0], -1)
14
15         return flatten_output
16
17     def backward(self, output_gradient):
18         input_gradient = output_gradient.reshape(self.
19 input_shape)
20         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

### 25.1 Técnicas de Inicialização

### 25.2 Regularização L1 e L2

### 25.3 Normalização

#### 25.3.1 Normalização de Camadas

#### 25.3.2 Normalização de Batch

### 25.4 Clipping do Gradiente

### 25.5 Dropout: Menos Neurônios Mais Aprendizado

### 25.6 Data Augmentation



## 26 Transformers

26.1 As Limitações das RNNs: O Gargalo Sequencial

26.2 A Ideia Central: Self-Attention (Query, Key, Value)

26.3 Escalando a Atenção: Multi-Head Attention

26.4 A Arquitetura Completa: O Bloco Transformer

26.5 Entendendo a Posição: Codificação Posicional

26.6 As Três Grandes Arquiteturas

26.6.1 Encoder-Only (Ex: BERT): Para tarefas de entendimento

26.6.2 Decoder-Only (Ex: GPT): Para tarefas de geração

26.6.3 Encoder-Decoder (Ex: T5): Para tarefas de tradução/sumarização

26.7 Além do Texto: Vision Transformers (ViT)



## 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.

CYBENKO, George. Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals, and Systems*, v. 2, n. 4, p. 303–314, 1989. Citado na p. 42.

GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. *Deep Learning*. [S. l.]: MIT Press, 2016. Citado nas pp. 28, 42, 43.

LESHNO, Moshe *et al.* Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, v. 6, n. 6, p. 861–867, 1993. ISSN 0893-6080. DOI: [https://doi.org/10.1016/S0893-6080\(05\)80131-5](https://doi.org/10.1016/S0893-6080(05)80131-5). Disponível em: <https://www.sciencedirect.com/science/article/pii/S0893608005801315>. Citado na p. 43.

RUMELHART, David E.; HINTON, Geoffrey E.; WILLIAMS, Ronald J. Learning Representations by Back-Propagating Errors. *Nature*, v. 323, p. 533–536, 1986. Citado nas pp. 32–35, 37.