Luiz Guilherme Morais da Costa Faria

APRENDIZADO DE MÁQUINA

Brasília, DF
26 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
26 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

$$x = x_0 - \epsilon \nabla f(x) \tag{6.1}$$

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.

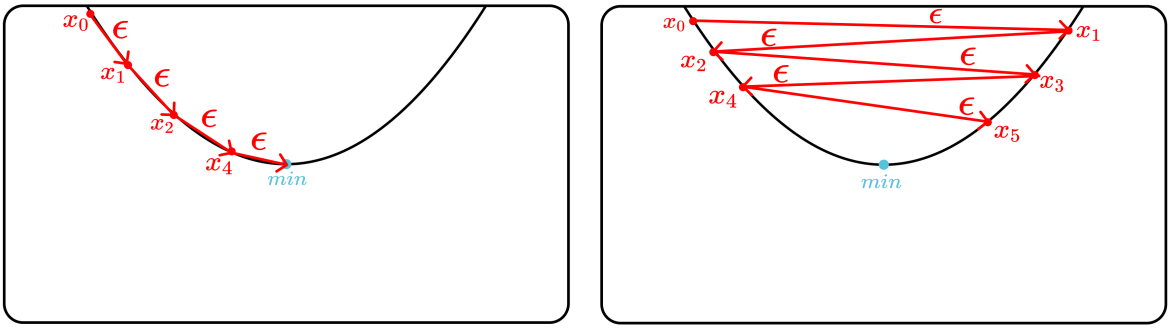


Figura 1 – Comparativo do tamanho de passos em uma função polinomial.

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.

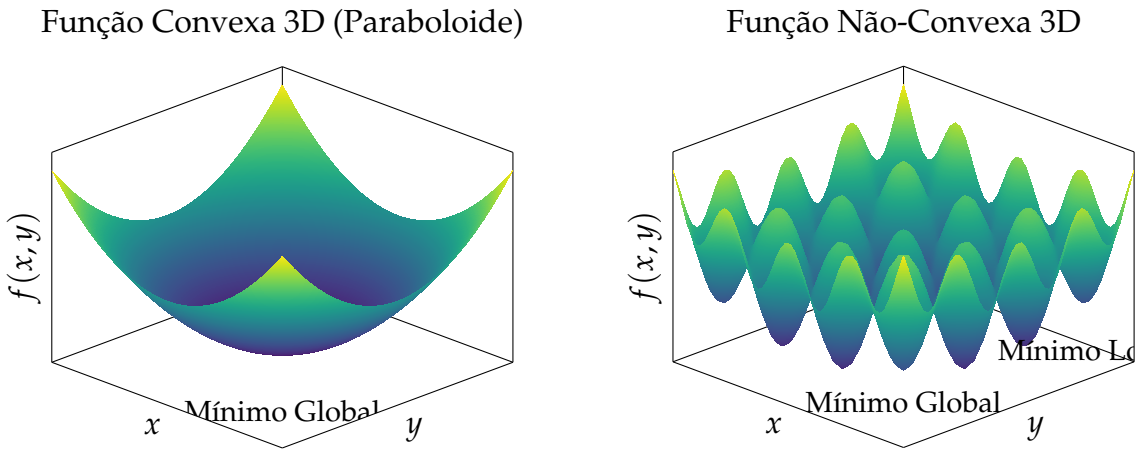


Figura 2 – Comparação entre funções 3D convexas e não-convexas.

6.1.3 Implementação em Python

Algorithm 1 O Método do Gradiente (Descida do Gradiente)

Require: Taxa de aprendizado global ϵ
Require: Parâmetros iniciais $\boldsymbol{\theta}$
1: Inicialize os parâmetros $\boldsymbol{\theta}$
2: **while** critério de parada não for atingido **do**
3: Calcule o gradiente $\mathbf{g} \leftarrow \nabla_\theta L(\boldsymbol{\theta})$, usando **todo** o conjunto de treinamento
4: Calcule a atualização: $\Delta\boldsymbol{\theta} \leftarrow -\epsilon\mathbf{g}$
5: Aplique a atualização: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
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 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 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 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*, e será explicada nesse capítulo de forma detalhada. Para issom Rumelhart, Hinton e Williams (1986) comecam introduzindo três diferentes funções que serão utilizadas ao longo do texto para deduzir o 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

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

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 $x_i = \sum_i y_i \cdot w_{ji}$,

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 logísitca, 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 efciente.

Sigmoide Logística

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

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 imbute 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)

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

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:

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

Note que todos os termos que não possuem o índice k, como a parte $(y_1 - d_1)^2$, serão valores constantes, e que estarão fora do obejtivo 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 pela 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 etepa 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 resultamente 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 $\partial y_i / \partial x_j$ 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:

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

Sendo assim, a expressão final fica:

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

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 σ' em relação a sua entrada x_j.

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

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

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 difultar o algortimo. 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, $\partial x_i / \partial w_{ij}$, ou seja, a derivada do neurônio em relação a um peso específico. Assim, tem-se a expressão incial:

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

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 simplicar 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:

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

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

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

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

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

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:

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

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:

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

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 entrada total desse neurônio, ou seja:

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

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

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 parâmetros, 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

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

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;
- ϵ 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

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

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 estrão distribuidos de forma vetorizada, que que terá um vetor de viéses, 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 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

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

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

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

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

Por fim, com base no que foi explicado até agora, é possível entender melhor como o gradiente passa de uma acamada para a outra durante a retropropagação, para isso será analisada um rede com quatro camadas densas, sendo elas:

- **Camada 1 (Entrada):**: Ela possui neurônios com o indice i;
- **Camada 2 (Oculta 1):**: Ela possui neurônios com o indice j;
- **Camada 3 (Oculta 3):**: Ela possui neurônios com o indice k;
- **Camada 4 (Oculta 3):**: Ela possui neurônios com o indice l;

O objetivo é calcular o gradiente de um peso da primeira camada de pesos, w_{ji}, ou seja, é um peso que conecta um neurônio da camada i com um neurônio da camada j. Assim, o primeiro passo é utilizar a fórmula 6.6, que calcula o gradiente do erro em relação a um peso quaquer.

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

Para encontrar esse gradiente, é preciso desenvolver a expressão $\partial E/\partial x_j$, que representa o gradiente do erro da entrada de um neurônio j da primeira camada oculta. Para isso, é possível utilizar como base a equação 6.5, obtendo então:

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

Nessa expressão, $\sigma'(x_j)$ representa a função de ativação, já o termo $\partial E/\partial y_j$ representa o gradiente que está vindo ca camada de cima, o qual é dado por:

$$\frac{\partial E}{\partial x_j} = \sum_k \frac{\partial E}{\partial x_k} \cdot w_{kj}$$

Juntando os dois termos:

$$\frac{\partial E}{\partial x_j} = \left(\sum_k \frac{\partial E}{\partial x_k} \cdot w_{kj} \right) \sigma'(x_j)$$

Agora, é preciso calcular o termo $\partial E/\partial x_j$, de forma que será possível trazer o gradiente da camada de saída l, assim temos:

$$\frac{\partial E}{\partial x_k} = \left(\sum_k \frac{\partial E}{\partial x_l} \cdot w_{lk} \right) \cdot \sigma'(x_k)$$

Em que $\partial E/\partial x_l$ representa o primeiro gradiente, ele é calculado na camada de saída.

Por último, é preciso combinar essas expressões, encontrando como resultado a expressão:

$$\frac{\partial E}{\partial w_{ij}} = \left(\sum_k \left[\left(\sum_l \frac{\partial E}{\partial x_l} \cdot w_{lk} \right) \sigma'(x_k) \right] \cdot w_{kj} \right) \sigma(x_j) \cdot y_i$$

Com base nessa expressão, é possível concluir que o gradiente de uma camada inicial é proporcional a uma cadeia de multiplicações dos pesos e das derivadas das funções de ativação das camadas posteriores, de forma que é possível simplificar isso para:

$$\frac{\partial E}{\partial w_{\text{primeira camada}}} \propto (\text{gradiente da saída}) \cdot (w_{\text{camada 3}} \cdot \sigma'_{\text{camada 3}}) \cdot (w_{\text{camada 2}} \cdot \sigma'_{\text{camada 2}})$$

Generalizando essa expressão, é possível concluir que o gradiente para uma camada é dado ela equação 6.12.

Gradiente para N Camadas

$$\delta^{(L)} = \left(\left(\mathbf{W}^{(L+1)} \right)^T \delta^{(L+1)} \right) \odot \sigma'(x^{(L)}) \tag{6.12}$$

Em que:

- L: Representa o índice de uma camada, podendo ser um valor entre 1 (indicando que é uma camada de entrada) ou n (indicando que é uma camada de saída);
- $\mathbf{W}^{(L)}$: Representa a matriz dde pesos que conecta a camada $L-1$ à camada L;
- $b^{(L)}$: Representa o vetor de viés da camada L;
- $x^{(L)}$: Representa o vetor de entradas totais para os neurônios da camada L antes da ativação;
- $y^{(L)}$: Representa o vetor de saídas da camada L
- $\delta^{(L)}$: Representa o vetor do gradienye na camada L;
- $\sigma'(x^{(L)})$: Representa o vetor contendo a derivada da função de ativação para cada neurônio da camada L;
- $\odot$: O produto de Hadamard, que significa multiplicação elemento a elemento.

Assim, é possível entender como o gradiente irá se propagar ao longo de uma rede neural. Esse ponto é um dos mais importantes, pois é possível considerar que ele terá uma grande relação com as funções de ativação que são utilizadas. Essa é uma relação

complicada, pois, enquanto um determinado tipo de função possa causar um tipo de problema no cálculo do gradiente, e por isso não deve ser uma opção viável para ser escolhida para resolver um problema, optar por outra família de função também pode acarretar outro tipo de problema com o gradiente.

Para isso, o capítulo 7 foca em entender as pricipais funções sigmoidais, como a sigmoide e a tangente hiperbólica, e como essas possuem uma forte relação com o problema do desaparecimento de gradientes. Em seguida, o capítulo 8 busca introduzir as funções retificadoras, como a ReLU, e como elas podem ser uma alternativa para contornar esse problema. Contudo, isso não as torna perfeitas, elas ainda são suceptíveis à outro tipo de problema: a explosão de gradientes; e, no caso da ReLU: o problema dos neurônios agonizantes.

6.3 Otimizadores Baseados em Gradiente

Como Rumelhart, Hinton e Williams (1986) explicam logo ao terminar a dedução do cálculo da atualização dos pesos utilizando o método do gradiente, esse processo pode ser bastante lento quando comparado com métodos que fazem uso de derivadas de segunda ordem, que certamente são mais caras para serem implementadas no computador. Assim, os autores recomendam uma variação do método do gradiente que faz uso de momentum com intuito de acelerar a convergência a um custo computacional menor que o de implementar derivadas de segunda ordem (Rumelhart; Hinton; Williams, 1986).

6.3.1 Método do Gradiente com Momentum

Uma boa analogia para entender como o método do gradiente com momentum funciona é pensar é uma pedra rolando morro abaixo. No começo, quando ela não havia pegado muita velocidade, ela demorava para rolar e dependendo podia quase parar em algum pedaço do morro, mas conforme foi descendo, foi ganahando mais velocidade, de forma que quando chegar no final do morro, a sua velocidade será bem maior do que quando estava no topo do mesmo. Essa variante funciona de forma parecida a esse cenário.

Da mesma forma que o método do gradiente não surgiu originalmente no contexto do aprendizado de máquina, a sua variante com momentum não foi diferente. Um os artigos que trabalha com esse conceito de utilizar o momentum para acelerar a velocidade do método do gradiente e com isso obter uma convergência mais rápida é o *Some methods of speeding up the convergence of iteration methods*, do pesquisador Polyak (1964), nele o autor busca criar métodos iterativos mais rápidos para resolver equações funcionais. Para isso, ele introduz um método de "dois passos", que consiste na junção do

método do gradiente com um segundo termo que dá inércia ao método, por último, ele justifica que esse método trás de fato uma convergência mais veloz quando comparado com métodos como o do gradiente.

Contudo, assim como a grande maioria dos conceitos matemáticos, essa ideia demorou a ser implementada para o aprendizado de máquina, e com o surgimento da retropropagação, como demonstrado por (Rumelhart; Hinton; Williams, 1986), em seu artigo, o método do gradiente com momentum se tornou uma excelente opção para garantir uma convergência mais rápida. Esse método é representado pela equação 6.13

Método do Gradiente com Momentum de Polyak

$$x_{n+1} = x_n - \epsilon \nabla f(x_n) + \beta(x_n - x_{n-1}) \tag{6.13}$$

Em que:

- x_{n+1}: Representa as coordenadas do ponto após a iteração do método;
- x_n: Representa as coordenadas do ponto na iteração n;
- ϵ: Representa a taxa de aprendizado / tamanho do passo
- β: Representa a "inércia", controlando a influência do passo anterior no passo atual;
- x_{n-1}: Representa as coordenadas do ponto na iteração anterior.

Para calcular β, Polyak (1964) determina que será utilizada a expressão:

$$\beta = \left(\frac{\sqrt{M} - \sqrt{m}}{\sqrt{M} + \sqrt{m}} \right)^2$$

Em que M representa o maior valor da matriz hessiana m é o menor valor no ponto de mínimo da função. Isso significa que esses valores não são conhecidos de antemão. Contudo o maior problema da fórmula proposta por Polyak é que ela exige muitos cálculos, imagine ter que calcular a matriz hessiana de uma função de perda para uma rede que possui milhares ou até milhões de parâmetros, isso é muita coisa e também muito demorado. Assim, no artigo da retropropagação, Rumelhart, Hinton e Williams (1986), apresentam uma fórmula diferente, com intuito dimunuir a quantidade de cálculos a serem feitas, mas mantendo a principal caracterísitca do método do gradiente com momentum: a inércia; ela é dada pela equação 6.14

Método do Gradiente com Momentum Usado na Retropropagação

$$\Delta w(t) = \epsilon \nabla f(x) + \alpha \Delta w(t-1) \tag{6.14}$$

Em que:

- $\Delta w(t)$: Representa a variação nos pesos;
- ϵ: Representa a taxa de aprendizado;
- $\nabla f(x)$: Representa o gradiente;
- α: Representa um fator de decaimento exponencial entre 0 e 1 que determina a contribuição do gradiente passado para o gradiente futuro.

6.3.1.1 Implementação em Python

Algorithm 2 O Método do Gradiente com Momentum (versão de Rumelhart et al.)

Require: Taxa de aprendizado global ϵ
Require: Coeficiente de momentum α
Require: Parâmetros iniciais $\boldsymbol{\theta}$
Inicialize a atualização anterior: $\Delta\boldsymbol{\theta} \leftarrow \mathbf{0}$
Inicialize os parâmetros $\boldsymbol{\theta}$
while critério de parada não for atingido **do**
    Calcule o gradiente $\mathbf{g} \leftarrow \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$
    Calcule a atualização atual: $\Delta\boldsymbol{\theta} \leftarrow -\epsilon\mathbf{g} + \alpha\Delta\boldsymbol{\theta}$
    Aplique a atualização: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

6.3.2 Método do Gradiente Estocástico

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 quil^metros 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:

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

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.

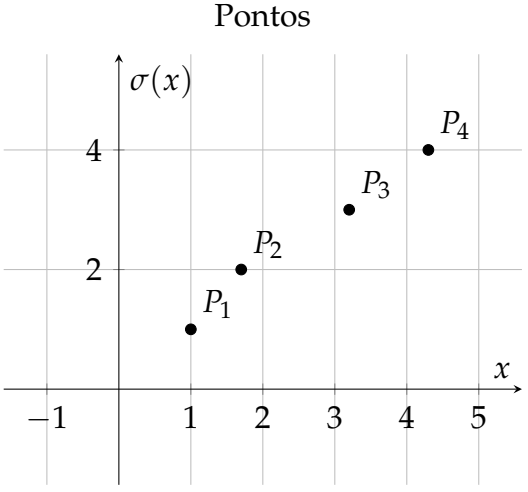


Figura 3 – Conjunto de pontos dispostos no plano cartesiano.

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

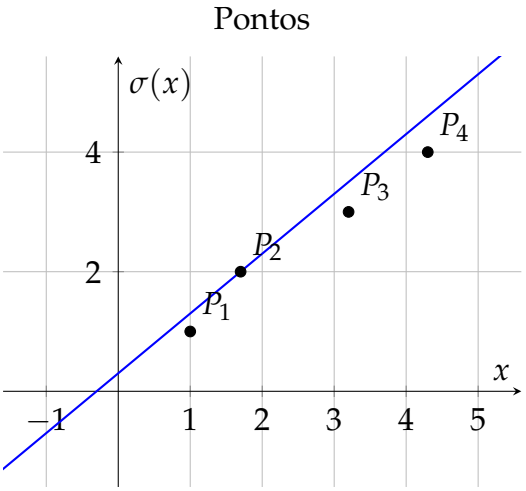


Figura 4 – Conjunto de pontos dispostos no plano cartesiano.

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 possuíssem 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 matematica, 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 logísitca, 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 analise de crescimento populacional e demografia, ela nao surgiu em um artigo em especifico, 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 area da ciência da computação.

7.4 Contexto Histórico: Popularização da Sigmoide em Redes Neurais

A primeira familia de funções que iremos conhecer são as sigmodais, sendo a principal delas, e que da 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 Daviel 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 possui uma derivada limitada.

Mas esse não foi o único fator que fez com que a sigmoide e sua familia 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 uma variação da função sigmoide, representada para expressão **??**, que era capaz de descrever o comportamento dos neurônios a certos estímulos.

Neurônio de Wilson e Cowan

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

Nessa equação, o parâmetro a representa a inclinação, a qual foi ajustada para passar pela origem ($\mathcal{S}(0) = 0$) e θ 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", caraterística da função sigmoide.

Com isso, ao consideramos 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 varias 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.

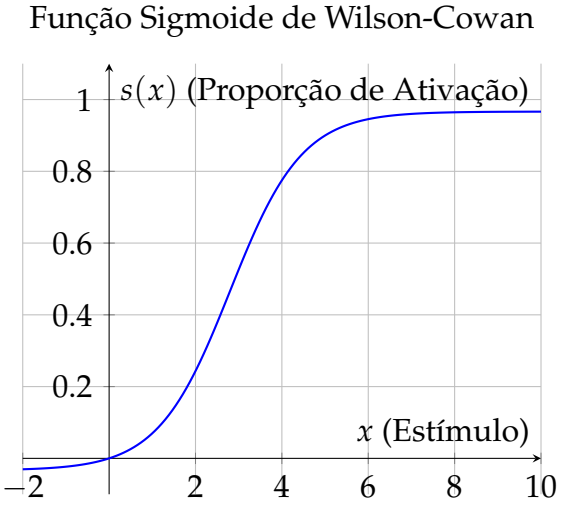


Figura 5 – Gráfico da função sigmoide conforme proposta por Wilson e Cowan (1972), com parâmetros de exemplo $a = 1.2$ e $\theta = 2.8$.

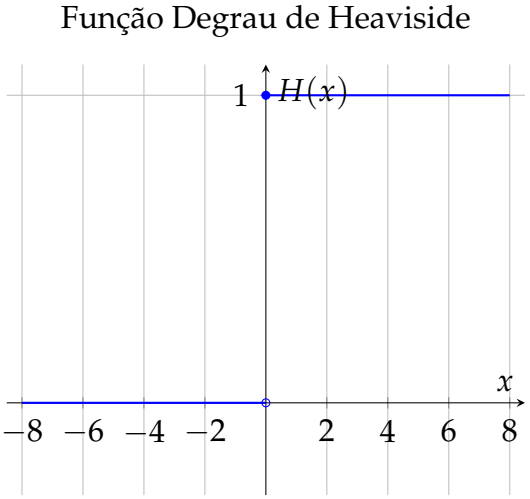


Figura 6 – Gráfico da função degrau de Heaviside.

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 nao 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 novamos 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 informations 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.

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 analise de crescimento populacional e demografia, ela nao surgiu em um artigo em especifico, 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 area da ciência da computação.

Sigmoide Logística

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

Função Sigmoide

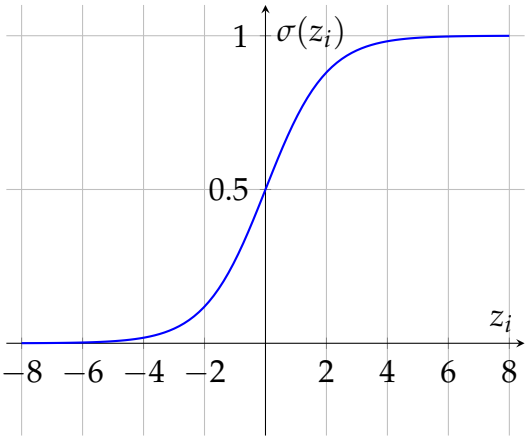


Figura 7 – Gráfico da função sigmoide logística.

A função sigmoide é escrita com uma exponencial, como na fórmula **??**. Johannes Lederer, no texto **"Activations Functions in Artificial Neural Networks: A Systematic Overview"**, explica sobre a sigmoide, segundo o autor, ela é uma função limitada,

diferenciável e monotônica, o que significa que conforme os valores de x aumentam os valores de $f(x)$ também aumentam.

Como vemos na figura 7, o gráfico da sigmoide possui o formato de um "S"deitado. Assim, também podemos dizer que a função sigmoide é uma função suave, contínua (o que a possibilita de ser derivável em todos os pontos) e também é não linear. Lederer explica que uma das propriedades interessantes da sigmoide está no fato dela empurrar os valores de entrada para dois extremos, neste caso, 0 e 1.

Note que, conforme os valores crescem/decrescem muito, essa curva passa a assumir o comportamento de quase uma reta, tendo pouca variação entre esses valores, esse ponto será discutido mais em seguida.

Agora podemos discutir a sua derivada, para resolvê-la, podemos aplicar uma regra do quociente obtendo a expressão **??** e o gráfico 8

Derivada da sigmoide logística

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

Derivada da Sigmoide

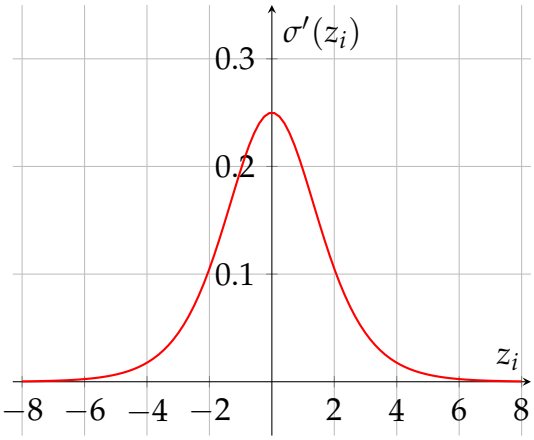


Figura 8 – Gráfico da derivada da sigmoide.

Notamos que sua derivada é contínua em todos os pontos e que ela atinge um ponto máximo entre 0.2 e 0.3 e que para valores que passam de ± 8 os resultados serão muito próximos de zero, isso acontece pois, como vemos no gráfico 7, a própria função sigmoide, não apresenta muita variação de valores conforme eles ultrapassam ± 8. Guarde essa informação, pois ela sera necessária para entender o conceito de gradiente evanescente, que será explicado no fim do capitulo. Contudo, a derivada da sigmoide também pode ser expressa em termos da própria sigmoide como a equação **??** nos mostra.

Ao analisarmos a derivada da sigmoide na expressão **??** vemos que ela possui uma derivada bem elegante e atrativa para as os cientistas da computação da época, pois pelo fato dela poder ser escrita em termos de sua própria função, isso permite que vários cálculos possam ser reaproveitados. Isso é algo interessante para a criação de uma rede neural no milênio passado pois, devemos lembrar que os computadores da época não possuíam uma poder computacional tão alto como os da atualidade. Assim, qualquer calculo que pudesse ser reaproveitado e até mesmo simplificado, já era de muita ajuda para os cientistas além de poupar grande tempo de processamento.

7.4.1 Implementação em Python

Com base nesses dados, podemos escrever a sua implementação em código. Para isso, devemos apenas seguir a fórmula da sigmoide logística e sua derivada. Temos então o bloco de código **??**.

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
```

7.5 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 transcendantes 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 11, 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 11.

Tangente Hiperbólica

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

Agora que temos a expressão da tangente hiperbólica e seu gráfico podemos também calcular sua derivada. Note na expressão **??**, que podemos escrever a derivada da tangente utilizando a expressão da secante hiperbólica, mas também podemos utilizar a própria tangente hiperbólica se quisermos, assim ela também passa a ser bastante util para reaproveitar códigos e cálculos.

Semelhante a sigmoide, os valores da tangente hiperbólica também se aproximam de zero conforme aumentam ou diminuem na sua derivada. Eles atingem um pico de 1, e conforme as entradas se aproximam de ± 4 a saída da derivada da tangente hiperbólica também fica próxima de zero. Além disso, como podemos ver na expressão **??**, a derivada da tangente hiperbólica também pode ser expressas em termos da sua própria função, permitindo que cálculos possam ser reaproveitados.

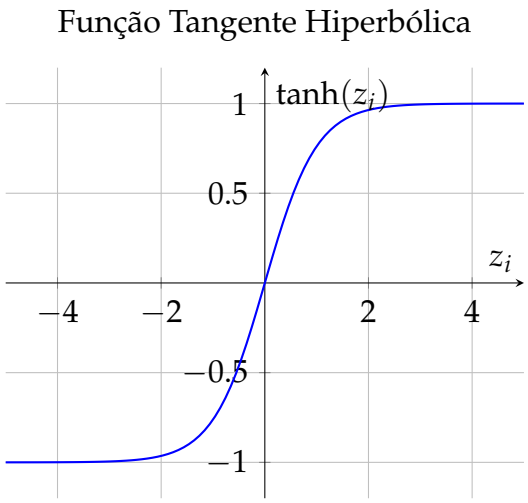


Figura 9 – Gráfico da tangente hiperbólica.

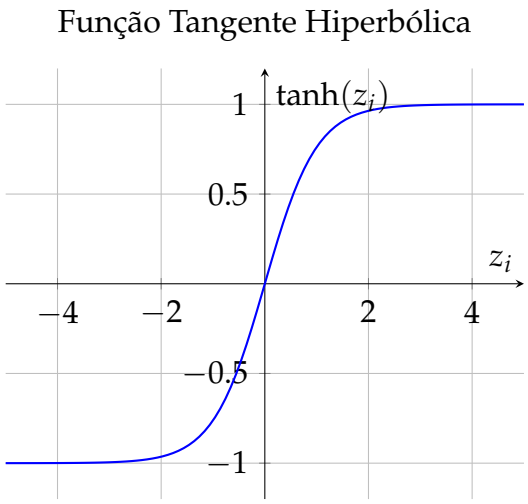


Figura 10 – Gráfico da tangente hiperbólica.

Derivada da tangente hiperbólica

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

7.5.1 Implementação em Python

Agora que sabemos como a tangente hiperbólica se comporta, além de conhecermos as suas fórmulas, podemos implementar a sua função em Python, utilizando o numpy para auxiliar nos cálculos.

Para isso, devemos implementar as expressões **??** e **??**, obtendo o bloco de código **??**.

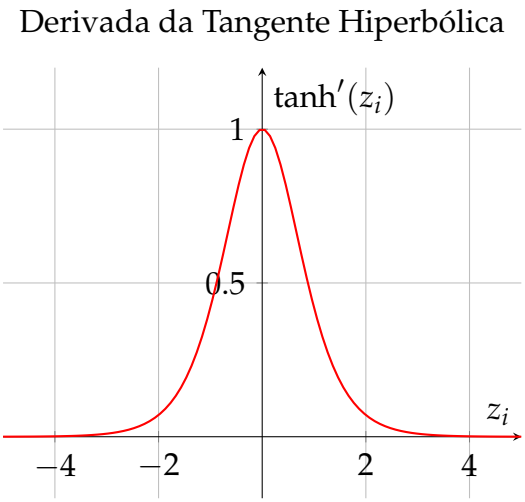


Figura 11 – Gráfico da derivada da tangente hiperbólica.

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

```
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
```

7.6 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 neu-

ral. 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 computacional 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

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

Função Softsign

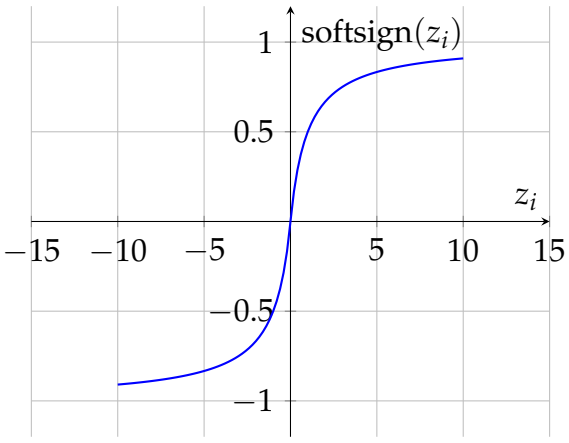


Figura 12 – Gráfico da função Softsign.

Com relação a sua diferenciabilidade, a softsign pode ser derivada em todos os seus pontos, e sua derivada pode ser vista na expressão **??**. Com base nela, notamos uma outra diferença da softsign com a tangente hiperbólica e a sigmoide. Diferente das outras duas, a derivada da softsign não pode ser expressa em termos da sua própria função.

Assim, enquanto nas outras funções é feito um cálculo complexo na função e um simples na derivada, pois aproveitamos o resultado, na softsign isso não ocorre. Podemos pensar que talvez essa função seja mais rápida que as outras duas no Forward, mas não podemos garantir a mesma coisa com relação ao Backward, para isso devemos fazer alguns testes antes.

Podemos ver também no gráfico da derivada da softsign que o valor de sua derivada começa a ficar proximo de zero quando x se aproxima de ± 10, indicando que ela começa a saturar nesse ponto.

Derivada da softsign

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

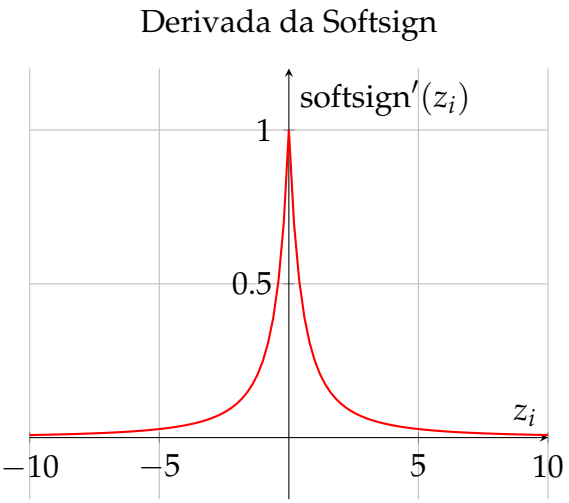


Figura 13 – Gráfico da derivada da função Softsign.

7.6.1 Implementação em Python

Assim, podemos implementar o bloco de código **??** representa a função softsign para ser utilizada uma rede neural.

Bloco de Código : Classe completa do função de ativação Softsign

```
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
```

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

Agora veremos duas funções sigmodais, criadas no contexto de redes neurais, cujo o seu intuito é ser trazer velocidade para o modelo que está sendo criado, elas são a hard sigmoid e hard tanh. Elas são inspiradas nas suas versões originais ou soft, com o mesmo intuito de variar até um certo ponto e depois saturar. Contudo, não garantem a mesma suavidade que as outras funções.

A primeira que vamos ver é a hard sigmoid. Como podemos ver na equação **??**, ela pode ser escrita juntando 3 diferentes funções, duas delas sendo funções constantes e uma terceira sendo a função identidade. Note que ela perde a suavidade da função seno, possuindo bicos que a impedem de ser derivada em todos os seus pontos, contudo, seus cálculos são bem mais simples os cálculos quando comparamos com a sigmoide tradicional, não tem nenhuma exponencial para atrasar as respostas da função. Mas note que temos um crescimento linear quando os valores estão entre -3 e 3.

Hard Sigmoid

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

Função Hard Sigmoid

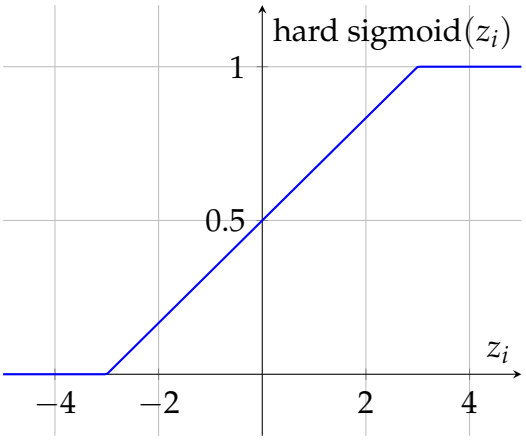


Figura 14 – Gráfico da função hard sigmoid.

Com relação a sua derivada, para obtê-lá, apenas derivamos todas as três expressões construindo a expressão da sua derivada. Note que ela também não é suave quando comparamos com a derivada da sigmoide tradicional.

Derivada da Hard Sigmoid

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

Note que o código dessa função é um pouco diferente das outras três que já vimos, ele não faz o uso de IFs, e sim opta por fazer essas operações de forma vetorizada, para garantir que os resultados sejam mais rápidos e otimizados para uma rede neural.

Também temos a função hard tanh, que tem a mesma proposta da hard sigmoid, porém busca imitar a tangente hiperbólica. Ela também é composta de uma condicional com três funções, duas constantes e a função identidade, como podemos ver na equação **??**.

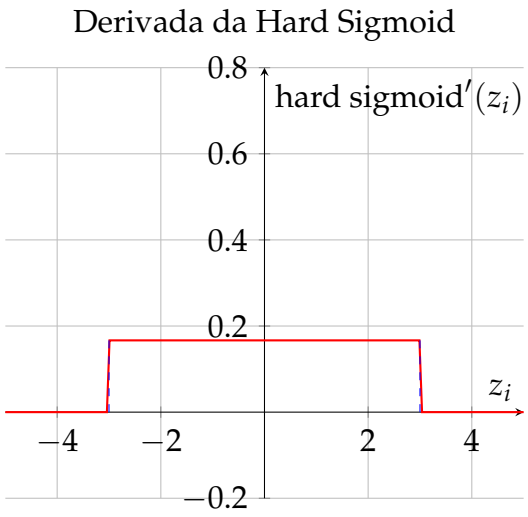


Figura 15 – Gráfico da derivada da Hard Sigmoid.

Hard Tanh

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

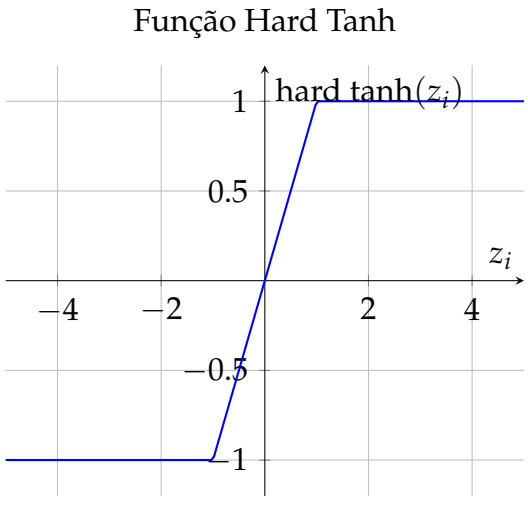


Figura 16 – Gráfico da função Hard Tanh. Fonte: PyTorch.

Utilizamos o mesmo procedimento da hard sigmoid para obter a derivada da hard tanh, derivamos as três expressões e encontramos então a equação **??**, ao lado, temos a figura 16, que representa o gráfico da derivada da hard tanh.

Derivada da Hard Tanh

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

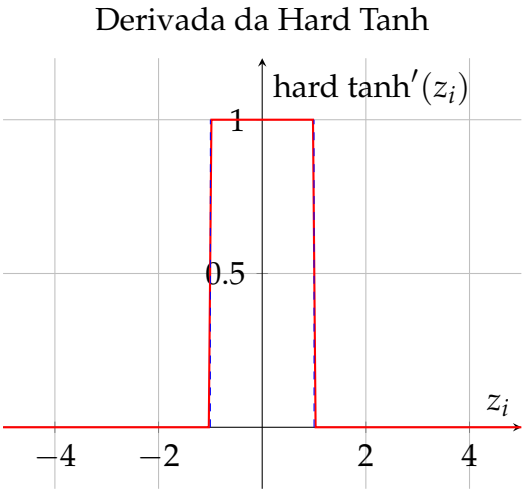


Figura 17 – Gráfico da derivada da Hard Tanh.

7.7.1 Implementação em Python

Para implementar a hard sigmoid em python, apenas devemos seguir as expressões **??** e **??** e obtemos o seu código. Note que por mais que ele seja maior que o da sigmoide, ele apresenta operações bem mais simples e "baratas"quando comparado com a sua versão soft.

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

```
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 Tanh

```
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
```

7.8 O Desaparecimento de Gradientes

Mesmo possuindo muitas propriedades atrativas para a utilização da familia sigmoidal em redes neurais, como a continuidade em todos os pontos e suavidade, além de que suas derivadas podem ser feitas com as próprias funções (no caso da sigmoide e da tangente hiperbólica), essa familia de funções trouxe um problema para os cientistas da época.

Como foi destacado ao discutir o gráfico das derivadas dessas funções, nos notamos que para valores extremos, seja eles positivos ou negativos, a derivada dessas funções fica bem próxima de zero. Isso significa que quando essas funções recebem como entrada um valor alto no foward pass, na retropropagação, por pegarmos esse valor e calcularmos a derivada da função de ativação naquele ponto, irá retornar um valor baixo.

Como nós vimos em seções anteriores, a fórmula para encontrar o gradiente retropropagado para camadas anteriores de uma rede neural é dada pela multiplicação da perda, com a derivada da função de atiavção no ponto e o valor do resultado da camada anterior de neurônios.

$$\frac{\partial E}{\partial w_{ik}} = \left(\sum_j \frac{\partial E}{\partial x_j} \cdot w_{ji} \right) \cdot \sigma'(z_i) \cdot a_k^{\text{ant}}$$

Se considerarmos uma situação em que a derivada função de ativação irá retornar uma valor baixo, esse valor será multiplicado com os outros termos da expressão fazendo com que o valor total do gradiente retrorpoagado naquela camada seja baixo.

$$\delta w = -\epsilon \frac{\partial E}{\partial w}$$

Nós também vimos que redes em que é utilizada a retropropagação, cálculo do gradiente é utilizado como forma de fazer com que os pesos e vieses dos neurônios se atualizem e com isso a rede aprenda. E se os pesos são atualizados com uma variação muito pequena quando comparamos com seus valores anteriores, isso significa que essa rede estaria dando pequenos passos para encontrar a sua função.

Ao passar valores muito extremos para uma função de ativação sigmoide, estamos prejudicando o aprendizado de uma rede neural, pois isso implica em derivadas com valores pequenos e consequentemente gradientes retropropagados pequenos. Agora imagine que em uma rede neural pode existir diversas camadas densas que usem a função sigmoide, se cada vez que o gradiente passar para a camada anterior ele diminuir, isso signica que na primeira camada, a última a ter seus pesos atualizados, o gradiente será tão pequeno que pode ser que não contribua para que a rede aprenda corredamente. É como se toda vez que passasse um valor muito extremo para a rede, o tamanho do passo que ela dá em direção a função procurada diminuísse.

Em **"Regularization and Reparametrization Avoid Vanishing Gradients in Sigmoid-Type Networks"**, Leni Ven e Johannes Lederer explicam que se o gradiente é muito pequeno em valor absoluto, ou até mesmo igual a zero, a atualização do gradiente apresenta quase nenhum impacto nos parâmetros de uma rede neural, o que faz com que não há progresso nos parâmetros de aprendizado, assim o gradiente evanescente é quando esse fenômeno acontece repetidademente por vários parares de entrada e saída.

Isso se torna um problema, pois, quando criamos uma rede neural, utilizamos as primeiras camadas para que elas sejam responsáveis por aprender características básicas/simples de uma determinada amostra de dados. Se o gradiente é próximo de zero, o calculo da atualização dos pesos e dos vises irá gerar valores muito próximos dos originais. Como esses valores não irão atualizar corretamente, a rede neural não irá aprender características básicas de um cenário.

Porém, as retificadoras também não foram perfeitas, e por possuírem uma saída que não era limitada, elas acabaram trazendo o problema inverso, o gradiente explosivo. Assim, escolher uma função de ativação para compor uma rede neural é uma etapa

trabalhosa, pois estaremos sempre lidando com vantagens e desvantagens, sendo necessária uma escolha minuciosa a fim de encontrar o que melhor nos ajuda.

7.9 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)

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

h!

Figura 18 – Gráfico da função ReLU.

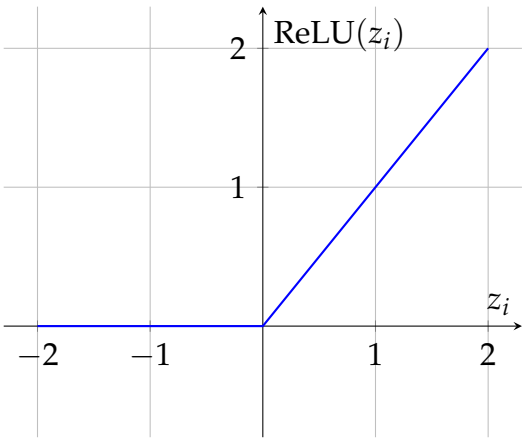
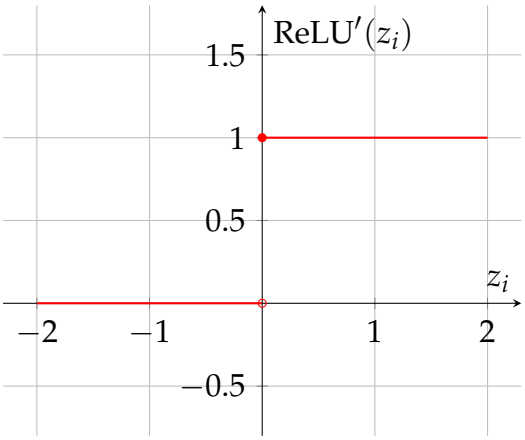


Figura 19 – Gráfico da Derivada da Função ReLU



8.2.1 Implementação em Python

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

```
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

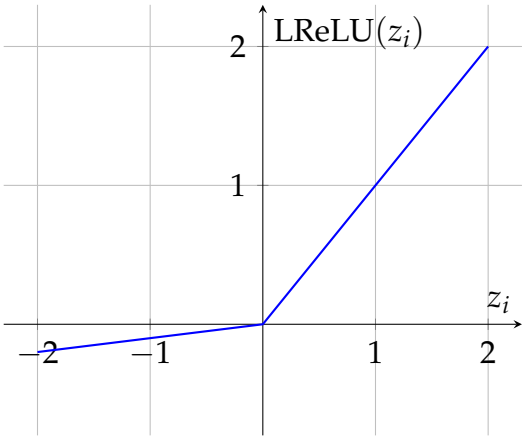
8.4 Corrigindo o Dying ReLUs Problem: As Variantes com Vazamento

8.4.1 Leaky ReLU

Leaky ReLU (LReLU)

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

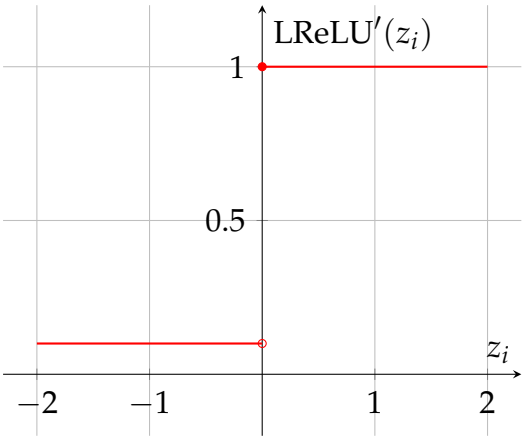
Figura 20 – Gráfico da Função Leaky ReLU (LReLU) com $\alpha = 0.1$



Derivada Leaky ReLU (LReLU)

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

Figura 21 – 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

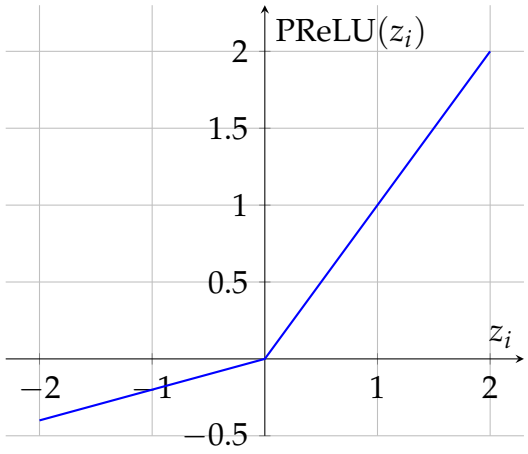
```
import numpy as np
from layers.base import Layer


class LeakyReLU(Layer):
    def __init__(self, alpha=0.01):
        super().__init__()
        self.input = None
        self.alpha = alpha

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

    def backward(self, grad_output):
        leaky_relu_grad = np.where(self.input > 0, 1, self.
    alpha)
        return grad_output * leaky_relu_grad, None
```

8.4.2 Parametric ReLU

Parametric ReLU (PReLU)

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

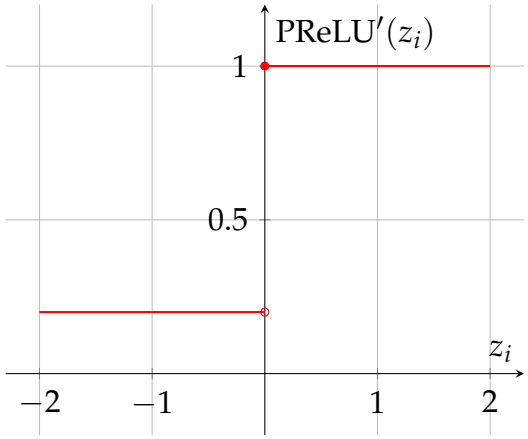
Figura 22 – Gráfico da Função Parametric ReLU (PReLU) com $\alpha = 0.2$



Derivada Parametric ReLU (PReLU)

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

Figura 23 – Gráfico da Derivada da Função Parametric ReLU (PReLU) com ($\alpha = 0.2$)

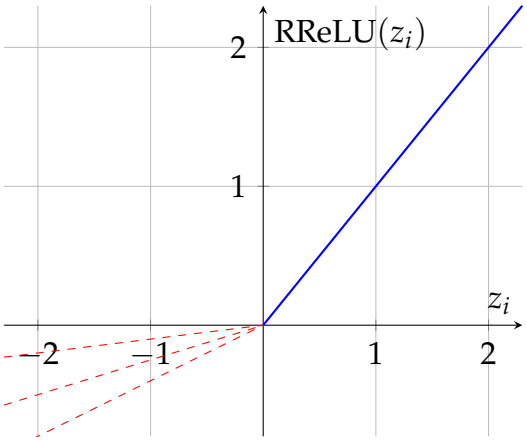


8.4.3 Randomized Leaky ReLU

Randomized Leaky ReLU (RReLU)

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

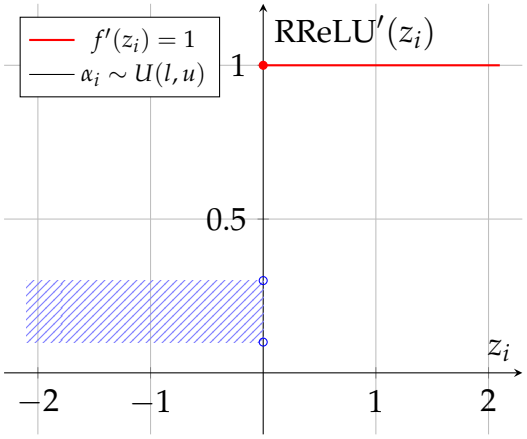
Figura 24 – Gráfico da Função Randomized Leaky ReLU (RReLU) com Diferentes Inclinações Aleatórias para a Parte Negativa



Derivada Randomized Leaky ReLU (RReLU)

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

Figura 25 – Gráfico da Derivada da Função Randomized Leaky ReLU (RReLU) com $l = 0.1, u = 0.3$



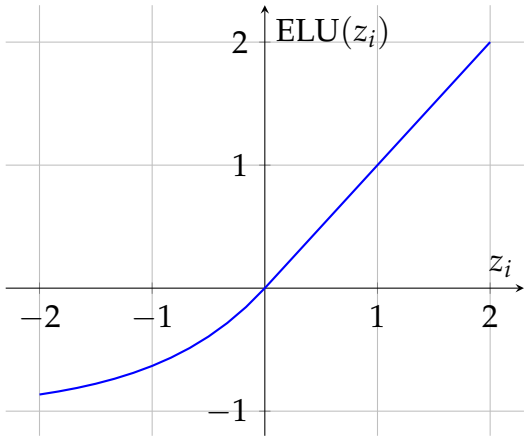
8.5 Em Busca da Suavidade

8.5.1 Exponential Linear Unit

Exponential Linear Unit (ELU)

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

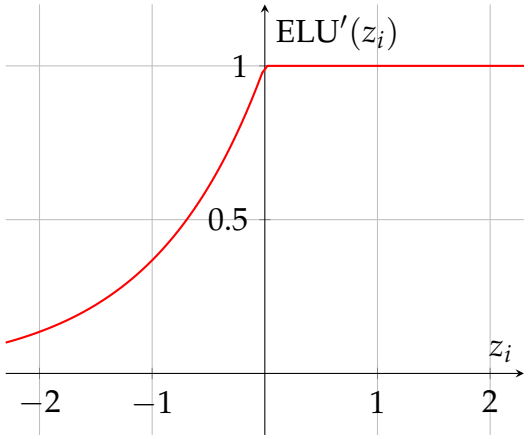
Figura 26 – Gráfico da Função Exponential Linear Unit (ELU) com $\alpha = 1$



Derivada Exponential Linear Unit (ELU)

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

Figura 27 – Gráfico da Derivada da Função Exponential Linear Unit (ELU) com $\alpha = 1$

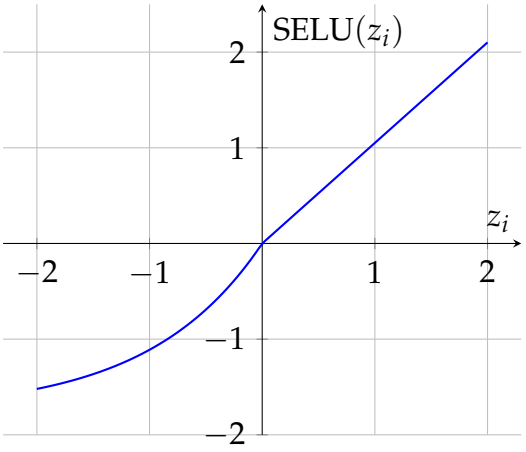


8.5.2 Scaled Exponential Linear Unit

Scaled Exponential Linear Unit (ELU)

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

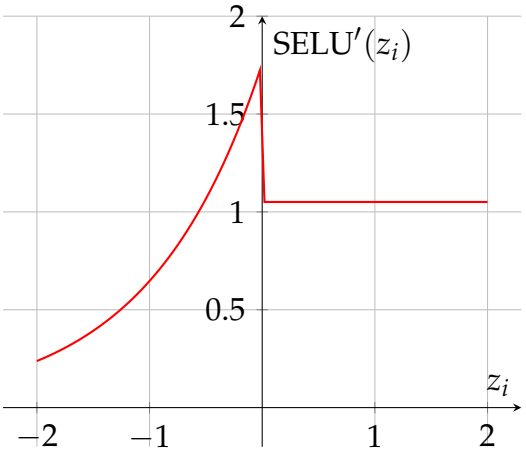
Figura 28 – Gráfico da Função Scaled Exponential Linear Unit (SELU) com $\alpha \approx 1.67 e \lambda \approx 1.05$



Derivada Scaled Exponential Linear Unit (ELU)

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

Figura 29 – Gráfico da Derivada da Função Scaled Exponential Linear Unit (SELU) com $\alpha \approx 1.67, \lambda \approx 1.05$

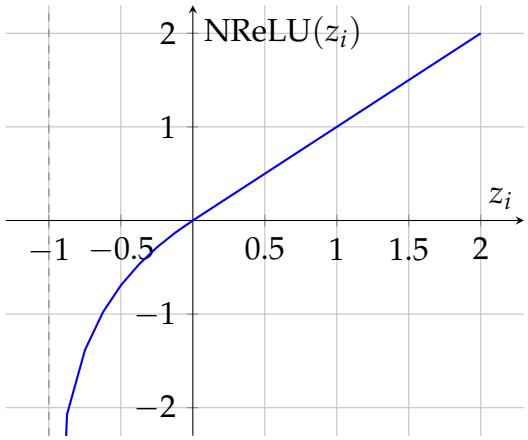


8.5.3 Noisy ReLU

Noisy ReLU (NReLU)

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

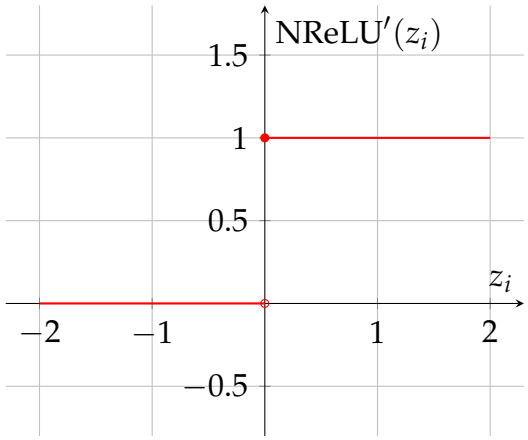
Figura 30 – Gráfico da Função Noisy ReLU (NReLU)



Derivada Noisy ReLU (NReLU)

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

Figura 31 – 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

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.

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. 46.

GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. *Deep Learning*. [*S. l.*]: MIT Press, 2016. Citado nas pp. 28, 46, 47.

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. Disponível em: https://www.sciencedirect.com/science/article/pii/S0893608005801315. Citado na p. 47.

POLYAK, Boris T. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, Elsevier, v. 4, n. 5, p. 1–17, 1964. Citado nas pp. 41, 42.

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, 41, 42.