Luiz Guilherme Morais da Costa Faria

INTELIGÊNCIA ARTIFICAL

Fundamentos Teóricos, Técnicas de Aprendizado de Máquina Clássico e Aprendizado Profundo

Brasília, DF
12 de outubro de 2025

Luiz Guilherme Morais da Costa Faria

INTELIGÊNCIA ARTIFICAL

Fundamentos Teóricos, Técnicas de Aprendizado de Máquina Clássico e Aprendizado Profundo

Universidade de Brasília

Orientador: Nome do Orientador/Revisor (se aplicável)

Brasília, DF
12 de outubro 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

"Pra lá é pra baixo? É. Então eu vou pra lá."
— A filosofia de vida do gradiente descendente

Cada rede neural é composta de diferentes partes, uma rede *feedforward* é composta de camadas densas, que por sua vez são compostas por um conjunto de neurônios. Uma rede convolucional, possui camadas de achatamento, camadas de *pooling* e camadas convolucionais, que possuem *kernels* muitas vezes inteligêntes que aprendem conforme o o treino a reconhecer diferentes padrões em imagens. Já uma rede recorrente, possui camadas com neurônios recorrentes, que buscam reconhecer padrões em diferentes tipos de séries, a fim de que seja por exemplo, possível prever qual será o valor de ação de uma marca x no dia 27 de setembro de 2025.

O que não falta em uma rede neural são parâmetros, eles estão por todos os lados. Considerando isso, imagine que você está encarregado de criar uma rede convolucional para reconhecer imagens de cães e gatos e precisa de forma manual ajustar todos, o pesos das camadas densas junto com o seus vieses, além de ter que atualizar os valores os kernels presentes nas camadas convolucionais.

Certamente isso é impossível de ser feito, uma rede como essa pode ter milhares e até milhões de parâmetros. Por isso, uma das formas de contornar esse problema, e, deixar para que o computador faça as suas atualizações é a retropropagação. Ela caminha junto com o método do gradiente para que todas essas atualizações sejam feitas de forma automática, e com uma maior garantia que ela será eficáz, uma vez que estará se baseando em como o erro do modelo que está sendo treinado é medido.

Esse captítulo se inicia introduzindo primeiro o método do gradiente, o qual serve de inspiração até hoje para diversos algoritmos de otimização. Em seguida, é dedicada uma seção estudando a retropropagação, para isso são explicadas as suas fórmulas, e como elas atuam para minimizar como o erro é propagado pela rede.

Conhecendo esses dois tópicos, é possível aprofundar em métodos mais elaborados de otimização, que buscam assim como o método do gradiente encontrar o ponto de mínimo local de uma função, mas, de forma mais rápida, gastantando menos iterações, sendo um desses exemplos o método do gradiente com momento. Com novos métodos criados e apresentados para a comunidade científica, não demorou muito para que os antigos se tornassem obsoletos e novas formas de otimização fossem criadas, assim, a próxima seção explica alguns dos métodos de otimização modernos, começando pelo *AdaGrad*, o qual faz uso de taxas de aprendizado adpativas para garantir

um melhor aprendizado da rede que está sendo criada, ele serve de inspiração para a grande maioria dos métodos dessa seção, que buscam fazer pequenos incrementos em seu funcionamento.

6.1 O Método do Gradiente Descendente: O Motor da Retropropagação

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 português) do matemático francês do século XVIII Cauchy (1847), em que o autor apresenta um método que pode ser considerado um precursor para o método do gradiente atual.

Nesse texto, o autor discute 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 ilustrando o funcionamento dessa ferramenta.

6.1.1 Exemplo Ilustrativo: Cadeia de Montanhas

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

Para chegar até esse lago, você conta com uma bússola um tanto quanto diferente, ao invés dela apontar para o norte como uma bússola comum, ela aponta para a direção da subida mais íngreme daquele local. Ora, isso parece desnecessário para o que você precisa, você quer ir para o lugar com menor altitude, não o maior. Mas pensando um

pouco, você chegou a conclusão que se ir para a direção contrária a da bússola, você certamente irá chegar no lago.

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 contrária apontada pela bússola.

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 retropropagação são algumas das principais ferramentas que colaboram para que os modelos de aprendizado de máquina possam aprender com os seus erros e com isso se tornarem melhores a cada iteração.

A bússola e o lago servem como uma analogia para entender como o método do gradiente funciona, mas ele também possui uma definição matemática, a qual pode ser analisada na seção seguinte.

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

$$\theta_{t+1} = \theta_t - \eta \nabla f(\theta_t) \tag{6.1}$$

Em que:

- θ_t: representa as coordenadas do ponto atual.
- θ_{t+1}: representa as coordenadas do próximo ponto.
- η: representa o tamanho do passo, também chamado de taxa de aprendizado.
- $\nabla f(\theta_t)$: representa o vetor gradiente da função f calculado no ponto atual.

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 significa que o passo que o modelo irá dar de um ponto para outro será menor, e com isso implica que ele levará mais passos para encontrar um ponto de mínimo (essa situação pode ser vista na ilustração da esquerda da figura 6.1). 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, de forma que é possível ver isso acontecendo na ilustração da direita da figura 6.1.

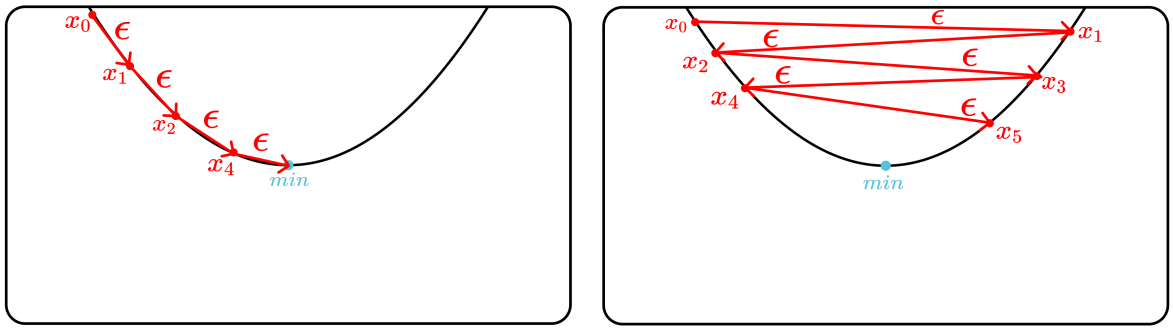


Figura 6.1 – Comparativo do tamanho de passos em uma função polinomial.

Fonte: O autor (2025).

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

Outro ponto que deve-se atentar é com relação as funções que estão sendo analisadas ao utilizar o método do gradiente mas também qualquer otimizador que seja baseado nele. Caso tenha uma função convexa (como a representada na ilustração da

esquerda da Figura 6.2), em que seu formato lembra um funil, será bem mais fácil para o modelo encontrar o ponto de mínimo global daquela função. Mas se essa função for uma função não convexa (como a representada na ilustração da direita da Figura 6.2), 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 irá afetar 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.

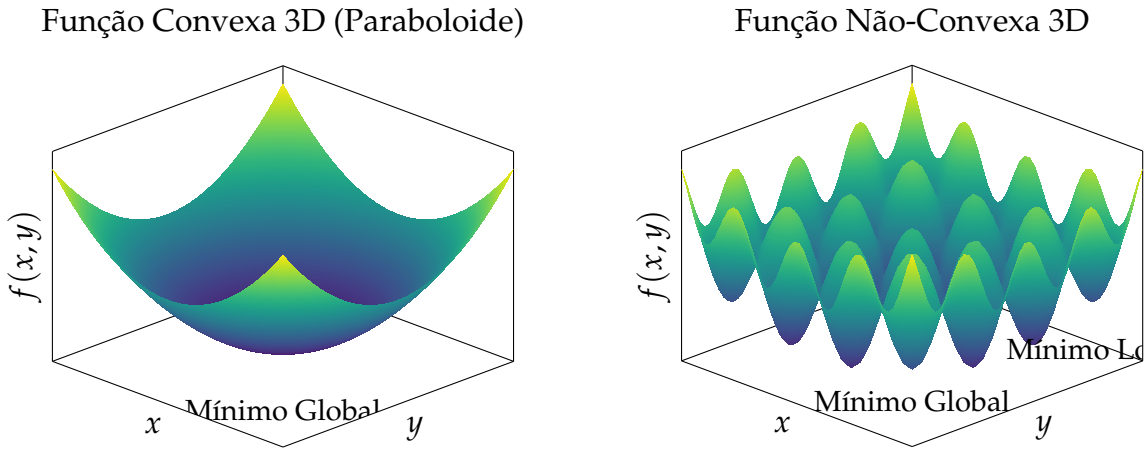


Figura 6.2 – Comparação entre funções 3D convexas e não-convexas.

Fonte: O autor (2025).

Além de ser representado utilizando equações, o método do gradiente pode ser expresso também utilizando pseudocódigo para escrever o Algortimo 1. Perceba que ele recebe como parâmetros de entrada uma taxa de aprendizado η, o vetor contendo os parâmetros iniciais θ_t e uma função de perda $f(\theta)$ a ser otimizada.

Algorithm 1 O Método do Gradiente

Requer: Taxa de aprendizado η
Requer: Vetor de parâmetros inicial θ_0
Requer: Função a ser otimizada $f(\theta)$
1: **while** não convergir **do**
2: $\quad \mathbf{g}_t \leftarrow \nabla_{\theta_{t-1}} f(\theta_{t-1})$
3: $\quad \theta_t \leftarrow \theta_{t-1} - \eta \mathbf{g}_t$
4: **end while**
5: **Retorne:** θ_t ▷ Parâmetros resultantes

Conhecendo o funcionamento do método do gradiente, é possível agora entender a retropropagação, e como ela utiliza esse método para otimizar os pesos das redes neurais garantindo assim um aprendizado eficiente e também a automatização das trocas de parâmetros presentes em uma rede neural.

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 nessa seção de forma detalhada. Para isso os autores 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 (RUMELHART; HINTON; WILLIAMS, 1986).

A primeira função, representada na Equação 6.2, explica como um neurônio de uma camada densa irá funcionar quando recebe determinadas entradas [1].

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

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_{ji}: representa o peso da conexão entre o neurônio i com o neurônio j;
- b_j: representa o viés do neurônio j.

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

[1] 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).

Equação 6.3, 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.

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

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, os autores utilizam a função do erro quadrático médio (*MSE*) (RUMELHART; HINTON; WILLIAMS, 1986). 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 pela metade. Ela é dada pela Equação 6.4.

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

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*. De forma que 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. Com isso, 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:

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

A próxima etapa 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 portanto, 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 etapa é aplicar a regra da cadeia, pois, como a entrada do neurônio x_j não aparece diretamente na equação do erro. Então, utilizando essa técnica, é possível derivar o erro em relação a saída do neurônio y_j e multiplicá-lo pela derivada da saída do neurônio em relação a sua entrada. Com base nisso, 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, $\partial E/\partial y_j$ que foi calculado e desenvolvido na expressão na primeira parte dessa demonstração, de forma que 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:

$$\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j} \cdot y_j \cdot (1 - y_j) \tag{6.5}$$

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

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

Em que:

- $\partial E / \partial x_j$: Representa como o erro varia em relação à uma entrada x_j de um neurônio j;
- $\partial E / \partial y_j$: Representa como o erro varia em relação à uma saída de uma função de ativação que recebe a saída do neurônio j;
- $\sigma'(x_j)$: Representa a derivada da função de ativação calculada para os valores de x_j.

Note que $\sigma'(x_j)$ representa a derivada de uma função de ativação qualquer, neste caso representa a sigmoide logística. Mas pode ser outra função, como a tangente hiperbólica ou a *ReLU*. Um ponto importante a ser destacado, é que como a retropropagação trabalha com o método do gradiente, as derivadas são parte essencial do algoritmo, utilizar funções de ativação que possuem derivadas complexas, ou que não podem ser derivadas em grande parte do seu domínio, acabam por dificultar o 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 à um peso w_{ji} específico da rede neural, para isso, é preciso calcular a expressão $\partial E / \partial w_{ji}$. Note mais uma vez que o peso w_{ji} não aparece diretamente na equação do erro, de

forma que é necessário mais uma vez aplicar a regra da cadeia para calcular essa derivada. Assim, a expressão inicial é:

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

Note que o primeiro termo da expressão é justamente o que foi calculado na Equação 6.6, portanto, é preciso calcular apenas o segundo termo, $\partial x_i / \partial w_{ji}$, ou seja, a derivada da entrada do do neurônio em relação a um peso específico. Obtendo então a expressão incial:

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

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

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

Em seguida, é possível abrir a soma, encontrando os termos:

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

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

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

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

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

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 dada então pelas equações 6.7

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

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial x_j} \cdot y_i \quad \text{ou} \quad \frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial y_j} \cdot \sigma'(x_j) \cdot y_i \tag{6.7}$$

Em que:

- $\partial E/\partial w_{ji}$: Representa como o erro varia em relação à variação de um peso w_{ji}, responsável por conectar os neurônios das camadas i e j;
- $\partial E/\partial x_j$: Representa como o erro varia em relação à uma entrada do de um neurônio j;
- y_i: Representa a saída da função de ativação de um neurônio da camada i;
- $\partial E/\partial y_j$: Representa como o erro varia em relação à saída de uma função de ativação de um neurônio j;
- $\sigma'(x_j)$: Representa a derivada da função de ativação calculada para os valores do neurônio x_j.

Dessa forma, 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. Com isso, a expressão inicial é dada por:

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

O primeiro termo, $\partial E/\partial x_j$, já foi calculado na Equação 6.6, assim, é preciso focar apenas na segunda parte da expressão, $\partial x_j/\partial b_j$, que é responsável por avaliar 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, $\sum_k y_k \cdot w_{jk}$, 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, a Equação 6.8.

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

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial x_j} \quad \text{ou} \quad \frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial y_j} \cdot \sigma'(x_j) \tag{6.8}$$

Em que:

- $\partial E / \partial b_j$: Representa como o erro varia em relação ao viés j;
- $\partial E / \partial x_j$: Representa como erro varia em relação à entrada x_j do neurônio j;
- $\partial E / \partial y_j$: Representa como erro varia em relação à saída da função de ativação que recebe as saídas do neurônio j;
- $\sigma'(x_j)$: Representa a derivada da função de ativação calculada para os valores de x_j.

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 6.9

Regra de Atualização de um Peso Através do Método do Gradiente

$$w_{t+1} = w_t - \eta \frac{\partial E}{\partial w} \tag{6.9}$$

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;
- η 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 6.10

Regra de Atualização de um Viés Através do Método do Gradiente

$$b_{j,t+1} = b_{j,t} - \eta \frac{\partial E}{\partial b_j} \tag{6.10}$$

- $b_{j,t+1}$ representa o viés b_j atualizado após o incremento do gradiente;
- $b_{j,t}$ representa o valor inicial do viés b_j na iteração t;
- η representa o tamanho do passo / taxa de aprendizado;
- $\partial E/\partial b_j$ representa o gradiente do erro em relação ao viés.

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 vieses, uma matriz de pesos e um vetor de neurônios. Assim, as Equações 6.11 e 6.12, 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

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

Em que:

- $\mathbf{w}_{t+1}$: Representa a matriz de pesos atualizada após o incremento do método;
- $\mathbf{w}_t$: Representa a matriz de pesos no instânte atual t;
- η: Representa a taxa de aprendizado;
- $\nabla_{\mathbf{w}} E$: Representa o vetor gradiente do erro calculado em relação aos pesos.

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

$$\mathbf{b}_{t+1} = \mathbf{b}_t - \eta \nabla_{\mathbf{b}} E \tag{6.12}$$

Em que:

- $\mathbf{b}_{t+1}$: Representa o vetor de vieses atualizado após o incremento do método;
- $\mathbf{b}_t$: Representa o vetor de vieses no instânte atual t;
- η: Representa a taxa de aprendizado;
- $\nabla_{\mathbf{b}} E$: Representa o vetor gradiente do erro calculado em relação aos vieses.

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 6.7, que calcula o gradiente do erro em relação a um peso quaquer.

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

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 6.6, 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 da camada de cima, a camada j, o qual será 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 $\partial E / \partial x_k$, 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:

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

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 6.13.

Gradiente para N Camadas

$$\delta^{(L)} = \left(\left(\mathbf{W}^{(L+1)} \right)^T \delta^{(L+1)} \right) \odot \sigma'(x^{(L)}) \tag{6.13}$$

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, com base na Equação 6.13, é possível compreender como o gradiente irá se propagar ao longo de uma rede neural. Esse é um dos pontos mais importantes, pois, qualquer alteração que mude o fluxo do gradiente em uma rede irá afetar diretamente o treinamento da mesma. Existem certos tipos de função de ativação que afetam diretamente como o gradiente é passado de camada em camada, fazendo com que ao ser propagado pelo *backward pass*, tenha seu valor reduzido consideravelmente, tornando um problema para as primeiras camadas da rede, as quais irão receber um gradiente muito pequeno e com isso as atualizações dos pesos e vieses serão mínimas, e com isso a rede sofra para aprender características mais básicas.

Além disso, existem outras funções de ativação que possuem o efeito inverso, fazendo com que o gradiente "exploda", atrapalhando diretamente como os pesos e vieses serão atualizados no aprendizado da rede.

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 *ReLUs* agonizantes.

Conhecendo como a retropropagação funciona é possível se aprofundar ainda mais em diferentes otimizadores baseados no gradiente. Os quais possuem a mesma ideia de utilizar o vetor gradiente como uma "bússola"indicando os pontos de mínimo da função, porém neste caso, esses outros métodos buscam acelerar o processo de otimização, garantindo uma quantidade menor de iterações para encontrar o mínimos.

6.3 Otimizadores Baseados em Gradiente: Melhorando o Gradiente Descendente

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 momento com intuito de acelerar a convergência a um custo computacional menor que o de implementar derivadas de segunda ordem (RUMELHART; HINTON; WILLIAMS, 1986).

Além disso, foram surgindo também outras variantes do método do gradiente, como a sua versão estocástica, trazendo uma natureza aleatória para as iterações do método, bem como o gradiente em *mini-batch*, que permite separar os parâmetros do modelo em lotes, podendo ser treinados de forma incremental. Uma outra variante é o gradiente acelerado de Nesterov, ele se diferencia dos outros otimizadores por adicionar o conceito de "*lookahead*", servindo de inspiração para uma variante do *Adam*, um dos algoritmos de otimização modernos.

Assim, essa seção busca introduzir esses novos métodos, de forma que sejam destacadas as suas diferenças ao método do gradiente tradicional, bem como quais as melhorias eles apresentam. Primeiro será visto o gradiente com momento.

6.3.1 Método do Gradiente com Momento

Uma boa analogia para entender como o método do gradiente com momento 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 ganhando 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 com a pedra rolando morro abaixo.

Da mesma forma que o método do gradiente não surgiu originalmente no contexto do aprendizado de máquina, a sua variante com momento não foi diferente. Um os artigos que trabalha com esse conceito de utilizar o momento 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 da URSS 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.

O método do gradiente com momento de Polyak pode ser visto na Equação 6.14.

Método do Gradiente com Momentum de Polyak

$$\theta_{t+1} = \theta_t - \eta \nabla f(\theta_t) + \beta(\theta_t - \theta_{t-1}) \tag{6.14}$$

Em que:

- θ_{t+1}: Representa as coordenadas do ponto após a iteração do método;
- θ_t: Representa as coordenadas do ponto na iteração n;
- η: Representa a taxa de aprendizado / tamanho do passo
- β: Representa a "inércia", controlando a influência do passo anterior no passo atual;
- θ_{t-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, equanto m é o menor valor da matriz hessiana, neste caso, ambos M e m são calculados no ponto de mínimo da função. Isso significa que esses valores não são conhecidos de antemão. Entretanto 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 momento: a inércia. Ela é dada pela Equação 6.15

Método do Gradiente com Momentum de Rumelhart et. al.

$$\Delta w(t) = \eta \nabla f(x) + \alpha \Delta w(t-1) \tag{6.15}$$

Em que:

- $\Delta w(t)$: Representa a variação nos pesos;
- η: 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.

De forma semelhante ao método do gradiente, o método do gradiente com momento utilizando no artigo da retropropagação também pode ser expresso em forma de pseudocódigo como no Algoritmo 2. Perceba que ele recebe como parâmetros os

mesmos do método do gradiente, a taxa de aprendizado η, o vetor contendo os parâmetros iniciais θ_0, a função de perda $f(\theta)$, mas também recebe o coeficiente de momento α. Em seguida, calculá-se o vetor gradiente para o instante $t-1$ e com base nele é feita a atualização do vetor de parâmetros seguindo a Equação 6.15.

Algorithm 2 O Método do Gradiente com Momento (versão de (RUMELHART; HINTON; WILLIAMS, 1986))

Requer: Taxa de aprendizado η
Requer: Coeficiente de momento α
Requer: Vetor de parâmetros inicial θ_0
Requer: Função a ser otimizada $f(\theta)$
$\Delta\theta \leftarrow 0$ ▷ Inicializa a atualização anterior
while não convergir **do**
    $\mathbf{g}_t \leftarrow \nabla_{\theta_{t-1}} f(\theta_{t-1})$
    $\theta_t \leftarrow \theta_{t-1} - \eta\mathbf{g}_t + \alpha\Delta\theta$
end while
Retorne: θ_t ▷ Parâmetros resultantes

Sabendo o método do gradiente como momento, é possível também conhecer outras soluções que buscam fazer uso do vetor gradiente para encontrar os pontos de mínimo, neste caso, na seção seguinte será analisado o método do gradiente estocástico ou *stochastic gradient descent (SGD)*.

6.3.2 Método do Gradiente Estocástico (SGD)

Assim como o método do gradiente que faz uso do momento para acelerar a convergência e com isso diminuir um número de iterações gastas para se encontrar o ponto de mínimo, o método do gradiente estocástico também não surgiu no contexto do aprendizado de máquina. Esse método fez sua primeira aparição no trabalho *A Stochastic Approximation Method*, dos autores Robbins e Monro (1951), em que eles introduzem uma forma de encontrar as raízes de uma função quando esta não pode ser observada de forma direta, mas apenas através de medições com ruído.

Contudo, apenas nos anos 60 métodos que se baseiam no cálculo de funções de erro com ruido passaram a aparecer no cenário de aprendizado de máquina, como foi o caso dos pesquisadores Widrow e Hoff (1960) com *ADALINE (Adaptive Linear Neuron)*. Nele, os autores buscam otimizar os valores dos pesos dos neurônios de uma rede em que ao calcular o erro do modelo, este era afetado por ruído (WIDROW; HOFF, 1960)

Conforme o tempo foi passando, o *SGD* provou ser uma excelente alternativa para encontrar pontos de mínimo em uma função de perda quando comparado com o método do gradiente tradicional. Isso se dá ao fato da sua natureza estocástica, que o permite "saltar"por pontos de mínimos locais e também por pontos de sela, contudo,

ele pode não garantir uma convergência tão boa quando comparado ao método tradicional, podendo chegar muito próximo do ponto de mínimo, mas escapando dele, devido a suas prioriedades aleatórias (GÉRON, 2019).

O *SGD* funciona da seguinte forma: ele seleciona uma instância aleatória do conjunto de treinamento a cada etapa, e com base nessa instância, ele calcula o gradiente (GÉRON, 2019). De tal forma que de um lado existe o gradiente descendente vetorizado, que foi visto nas Equações 6.11 (para a regra de atualização de um vetor de pesos) e 6.12 (para a regra de atualização de um vetor de vises), fazendo as atualizações em todos os parâmetros de uma vez só, enquanto de de outro lado existe o método do gradiente estocástico que utiliza apenas um parâmetro por vez. O SGD certamente será mais rápido, pois precisa que menos parâmetros estejam na memória principal para que as atualizações possam ser feitas, mas por outro lado acaba adicionando aleatoriedade para o processo de otimização.

Método do Gradiente Estocástico (SGD)

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta; x^{(i)}, y^{(i)}) \tag{6.16}$$

Em que:

- θ_{t+1}: representa o vetor de parâmetros do modelo após a atualização (na iteração $t+1$).
- θ_t: representa o vetor de parâmetros do modelo na iteração atual t.
- η: representa a taxa de aprendizado.
- $\nabla_\theta J(\theta; x^{(i)}, y^{(i)})$: representa o gradiente da função de custo J em relação aos parâmetros θ. É calculado utilizando apenas uma única amostra $(x^{(i)}, y^{(i)})$ do conjunto de dados.

Por ser um método que depende da aleatoriedade para calcular os parâmetros a serem atualizados, o *SGD* possui um comportamento mais imprevisível. Como pode ser visto na Figura 6.3, enquanto o gradiente em *batch* segue um caminho suave encontrar o ponto de mínimo global, o gradiente estocástico segue uma trajetória irregular, justamente pelo fato de não considerar todos os parâmetros de uma vez. Na prática, isso pode acabar afetando o desempenho do *SGD*, o qual irá demorar mais para convergir, contudo, ele acaba compensando essa demora ao ser mais rápido para calcular as iterações do método.

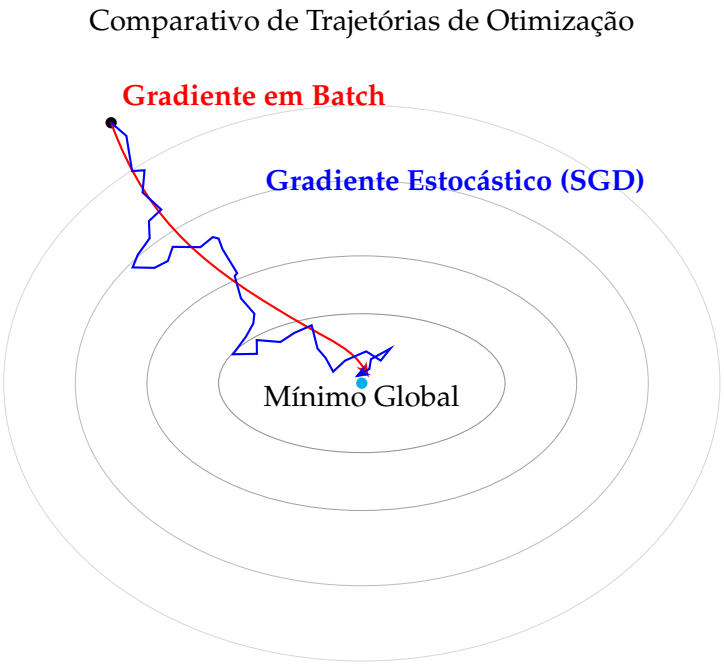


Figura 6.3 – Comparação visual entre a trajetória suave do Gradiente em Batch e a trajetória ruidosa do Gradiente Estocástico (SGD) para encontrar o mínimo de uma função de custo.

Fonte: O autor (2025).

6.3.3 Método do Gradiente em Mini-Batch (GD mini-batch)

Entendendo como funciona o *SGD*, o próximo passo é conhecer o método do gradiente em *mini-batch*, que busca ser uma alternativa entre o gradiente estocástico e o gradiente em *batch*. Como explica Géron (2019), o *GD mini-batch* calcula os gradientes para pequenos conjuntos aleatórios, chamados *mini-batches*, que fazem parte do conjunto total de parâmetros.

Por utilizar mais parâmetros de uma vez para calcular o gradiente, o gradiente em *mini-batch* possui uma vantagem quando comparado com o gradiente estocástico, ele irá apresentar um progresso de parâmetros menos irregular (GÉRON, 2019). Dessa forma, o gradiente em *mini-batch* consegue chegar um pouco mais perto dos pontos de mínimo, ajudando na convergência.

Uma boa vantagem do *GD mini-batch* está no fato que esses *batches* podem ter tamanhos escolhidos pelo programador, de forma que caso a máquina que esteja usando possua uma quantidade maior de memória *ram*, você pode utilizar *batches* maiores, ajudando na convergência do modelo para melhores pontos de mínimo.

6.3.4 Gradiente Acelerado de Nesterov (NAG)

Uma outra variante do método do gradiente é o gradiente acelerado de Nesterov (*NAG*). Ele foi introduzido no artigo *A method for solving the convex programming pro-*

blem with convergence rate $O(1/k^2)$ do autor Nesterov (1983), em que é apresentado como uma forma computacionamente mais barata de resolver problemas de programação convexa.

No texto, o autor trás a adição de um novo conceito para encontrar os pontos de mínimos, o chamado ponto de *lookahead* (olhar para frente em português), o qual serve de base para os cálculos do gradiente (NESTEROV, 1983). Assim o *NAG* possui dois passos: o primeiro consiste no cálculo da predição futura (no ponto de *lookahead*), dando um passo na direção do momento anterior, em seguida, com base nesse novo ponto calculado, é possível encontrar o valor do gradiente e com isso, atualizar os parâmetros da rede neural.

A sua expressão é mostrada na Equação 6.17.

Gradiente Acelerado de Nesterov (NAG)

$$\begin{aligned} \theta_{\text{lookahead}} &= \theta_t + \beta(\theta_t - \theta_{t-1}) \\ \theta_{t+1} &= \theta_{\text{lookahead}} - \eta \nabla_\theta J(\theta_{\text{lookahead}}) \end{aligned} \tag{6.17}$$

Em que:

- $\theta_{\text{lookahead}}$: Representa a posição futura estimada, que serve como ponto de "lookahead".
- θ_{t+1}: Representa o vetor de parâmetros atualizado após o passo de otimização.
- θ_t: Representa o vetor de parâmetros na iteração atual.
- θ_{t-1}: Representa o vetor de parâmetros da iteração anterior, usado para calcular o momento.
- β: Representa o coeficiente de momento.
- η: Representa a taxa de aprendizado.
- $\nabla_\theta J(\theta_{\text{lookahead}})$: Representa o gradiente da função de custo J.

Note, que o *NAG* é também uma variante do método do gradiente que faz uso do momento ou inércia para atualizar os parâmetros do modelo que está sendo treinado. Essa ideia de utilizar o momento como facilitador, permitindo encontrar pontos de mínimo de forma mais rápida, é utilizada até hoje em alguns otimizadores, o *NAdam*, uma variante do otimizador *Adam*, busca justamente implementar conceitos do gradiente acelerado de Nesterov no *Adam* e com isso conseguir alcançar convergências mais rápidas.

Para melhor entender o funcionamento do ponto de *lookahead* do *NAG*, é possível visualizar a Figura 6.4. Perceba que primeiro o modelo dá um passo em direção ao

lookahead utilizando o momento, em seguida, é feita a correção da direção utilizando como base o vetor gradiente. Dessa forma, o gradiente acelerado de Nesterov, ao utilizar o momento, consegue acelerar a convergência e com isso, encontrar pontos de mínimo mais rapidamente.

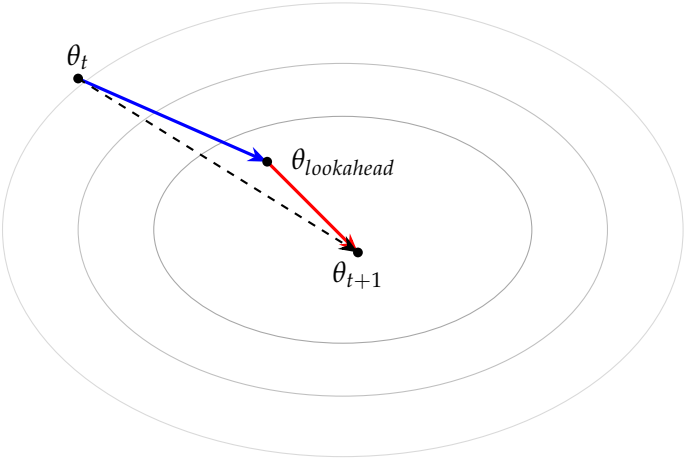


Figura 6.4 – Representação visual do funcionamento do gradiente acelerado de Nesterov. Em azul está representado o primeiro passo em direção ao ponto de *lookahead* utilizando o momento. Em vermelho está o segundo passo, com a atualização utilizando o vetor gradiente para corrigir a direção.

Fonte: O autor (2025).

Sabendo o funcionamento do *NAG*, também é possível desenvolver o seu pseudocódigo como no Algoritmo 3. Ele irá receber como entradas uma taxa de aprendizado η, o coefiente de momento μ, o vetor contendo os parâmetros inicias θ_0 e a função a ser otimizada $f(\theta)$. Com base nisso, se inicia o loop do gradiente acelerado de Nesterov, que começa calculando o vetor gradiente no ponto de *lookahead*, que é seguido da atualização dos parâmetros $\mathbf{m}_t$ os quais são utilizandos para fazer a atualização do vetor de parâmetros θ. Dessa forma, o loop termina após convergir para o ponto de mínimo, ou até atingir um número máximo de iterações.

Algorithm 3 Gradiente Acelerado de Nesterov (*NAG*)

Requer: Taxa de aprendizado η
Requer: Coeficiente de momento μ
Requer: Vetor de parâmetros inicial θ_0
Requer: Função a ser otimizada $f(\theta)$
1: **while** não convergir **do**
2: $\quad \mathbf{g}_t \leftarrow \nabla_{\theta_{t-1}} f(\theta_{t-1} - \eta\mu\mathbf{m}_{t-1})$
3: $\quad \mathbf{m}_t \leftarrow \mu\mathbf{m}_{t-1} + \mathbf{g}_t$
4: $\quad \theta_t \leftarrow \theta_{t-1} - \eta\mathbf{m}_t$
5: **end while**
6: **Retorne:** θ_t ▷ Parâmetros resultantes

6.3.5 Comparativo de Otimizadores Clássicos

Por fim, é possível compilar as características desses otimizadores vistos na Tabela 6.1. Dessa forma, é destacado a principal característica do otimizador, bem como suas vantagens e desvantagens.

Tabela 6.1 – Comparativo de otimizadores clássicos

Otimizador	**Característica principal**	**Vantagem**	**Desvantagem**
Gradiente descendente (*GD*)	Otimizador clássico que dá origem aos otimizadores baseados em gradiente	Apresenta fácil implementação	Precisa que todo o conjunto de parâmetros esteja na memória para fazer as atualizações e calcular o gradiente.
Gradiente descendente estocástico (*SGD*)	Calcula o vetor gradiente para apenas um item do conjunto de parâmetros e faz a atualização.	É mais rápido que o *GD* além de conseguir escapar de pontos de sela e mínimos locais.	Apresenta uma trajetória ruidosa
Gradiente descendente em *mini-batch* (*GD mini-batch*)	Calcula o vetor gradiente para pequenos lotes (*batches*) e faz a atualização.	Possui o parámetro tamanho do lote para ser escolhido também.	
Gradiente descendente com momento	Faz uso do momento para acelerar a convergência do método.	Consegue encontrar pontos de mínimo mais rapidamente e com menos iterações.	Possui o parâmetro do momento para ser escolhido também
Gradiente acelerado de Nesterov (*NAG*)	Faz uso de um ponto de *lookahead* para calcular o vetor gradiente e a partir desse fazer a atualização dos parâmetros	Possui uma convergência mais acelerada que o método tradicional	Apresenta um algoritmo mais custoso para ser implementado.

Fonte: O autor (2025).

Visto uma série de otimizadores mais antigos porém muito úteis para a tarefa de encontrar pontos de mínimo, é possível agora discutir os otimizadores mais recentes. Para isso, eles continuam seguindo a mesma ideia, utilizar o vetor gradiente como o

parâmetro de indicação para encontrar pontos de mínimo. Porém, fazendo uso de técnicas mais avançadas da matemática e da programação, de forma a garantir não só uma convergência mais rápida, mas também conseguir escapar de pontos de sela e mínimos locais.

6.4 Otimizadores Modernos Baseados em Gradiente: A Era das Taxas de Aprendizado Adaptativas

Conforme os anos foram passando, foram surgindo novos otimizadores, também inspirados em métodos mais clássicos, como o *SGD* e as variantes com momento. Mas, a partir dos anos 2010, uma nova família de otimizadores, que se originou a partir do *AdaGrad*, surgiu. Esses otimizadores tinham a mesma proposta dos outros baseados em gradiente, utilizá-lo como uma bússsola para encontrar os pontos de mínimo, mas ao mesmo tempo apresentavam uma diferença notável quando comparados com os otimizadores clássicos: o uso de diferentes taxas de aprendizado.

Fazer uso de diferentes taxas de aprendizado permitiu que os modelos criados com esses algoritmos apresentassem um desempenho maior. Pois, para caracterísitcas que apareciam menos dos *datasets* era possível atribuir taxas de aprendizados mais altas, como uma forma do modelo prestar mais atenção nessas caracterísitcas. Ao mesmo tempo que atribuir taxas de aprendizado menores para características mais comuns, permitia uma melhor convergência.

6.4.1 Adadptive Gradient Algorithm (AdaGrad)

Apresentado para a comunidade científica no texto *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*, *o Adadptive Gradient Algorithm (AdaGrad)*, foi descrito por Duchi, Hazan e Singer (2011) como uma adpatação dos métodos iterativos baseados em gradiente, que nos permite encontrar "agulhas em palheiros"em forma de características muito preditivas, mas raramente vistas.

O que os autores se referem ao ato de encontrar "agulhas em palheiros", está ligado diretamente na forma como o *AdaGrad* atua para melhorar o desempenho de um modelo tanto para reconhecer caracteríticas que são muito frequentes nos dados de treino quando características que são mais raras. Como Duchi, Hazan e Singer (2011) explicam, esse método faz uso de múltiplas taxas de aprendizado, assim caracterítiscas muito frequentes recebem uma taxa de aprendizado pequena, enquanto caraterísticas menos frequentes recebem uma taxa de aprendizado maior.

Dessa forma, o modelo que está sendo treinado conseque garantir melhores métricas durante o seu aprendizado, pois estará aprendendo sobre aquilo que aparece

muito nos dados de treino, mas, quando aparecer algo diferente e menos comum, ele irá "prestar mais atenção"devido à maior taxa de aprendizado nessas características.

O *AdaGrad* veio como uma forma de aplicar técnicas de aprendizado-teórico para problemas online e aprendizado estocástico, focando concretamente no métodos de (sub)gradiente, de forma que a representação desse algoritmo pode ser vista no Algoritmo 4.

Esse algoritmo recebe como entradas uma taxa de aprendizado η, o vetor de parâmetros a serem atualizados θ_0, a função a ser otimizada $f(\theta)$, e um parâmetros de estabilidade ϵ, o qual é adicionado no denominador para evitar que divisões por zero ocorram. O primeiro passo feito pelo *AdaGrad* é iniciar um vetor acumulador $\mathbf{n}_0$ que terá a mesma dimensão dos parâmetros θ_0 e será responsável por armazenar a soma dos quadrados dos gradientes passados para cada parâmetro.

Com isso feito, é então iniciado o loop do *AdaGrad*, o qual irá terminar somente quando o modelo convergir. Primeiro, é calculado o vetor gradiente $\mathbf{g}_t$ da função de custo f em relação aos parâmetros atuais da rede, como ele é um vetor gradiente, cabe destacar que ele estará apontando para a direção de maior crescimento da mesma função de custo. Em seguida, é calculado o termo t do vetor acumulador $\mathbf{n}_0$, para isso é somado o valor da interação anterior $\mathbf{n}_{t-1}$ ao quadrado elemento a elemento do gradiente atual $\mathbf{g}_t$. Por fim, o último passo do loop do *AdaGrad* consiste em atualizar os parâmetros do vetor θ_t, para isso é subtraído do parâmetros da iteração anterior θ_{t-1} a taxa de aprendizado η multiplicada pela divisão do vetor acumulador pela raiz quadrada do acumulador $\mathbf{n}_t$.

Algorithm 4 AdaGrad

Requer: Taxa de aprendizado η
Requer: Vetor de parâmetros inicial θ_0
Requer: Função a ser otimizada $f(\theta)$
Requer: Parâmetro de estabilidade ϵ
1: $\mathbf{n}_0 \leftarrow 0$ ▷ Inicializar o vetor acumulador
2: **while** não convergir **do**
3: $\quad \mathbf{g}_t \leftarrow \nabla_{\theta_{t-1}} f(\theta_{t-1})$
4: $\quad \mathbf{n}_t \leftarrow \mathbf{n}_{t-1} + \mathbf{g}_t \odot \mathbf{g}_t$
5: $\quad \theta_t \leftarrow \theta_{t-1} - \eta \frac{\mathbf{g}_t}{\sqrt{\mathbf{n}_t + \epsilon}}$
6: **end while**
7: **Retorne:** θ_t ▷ Parâmetros resultantes

Note que η quando está sendo utilizada para atualizar os parâmetros do modelo, passa a ser divida de elemento a elemento, isso faz com que elementos com um gradiente muito alto, recebam uma taxa de aprendizado baixa, enquando elementos menos comuns, e consequentemente com o gradiente menor, recebam uma taxa de aprendizado mais alta.

Para comprovar a eficácia do novo algoritmo desenvolvido, Duchi, Hazan e Singer (2011) utilizam uma série de *datasets* a fim de entender melhor em quais cenários, a aplicação do *AdaGrad* pode ser valiosa, para isso, os autores utilizam os *datasets*: *ImageNet* (para classificação de imagens), *MNIST* (para classificação de dígitos manuscritos), o *UCI repository* (para censo de income data), e *Reuters RCV1* (para classificação de textos).

Para a classificação de textos utilizando o *Reuters RCV1*, foram comparados o *AdaGrad-RDA*, o *RDA*, o *FB*, o *AdaGrad-FB*, o *PA* e o *ARROW*; de forma que são analisadas as taxas de erro de cada um desses modelos após o treino, bem como a proporção não-nula, responsável por indicar a quantidade de parâmetros do modelo que são diferentes de zero no final do treinamento, sendo uma medida para avaliar a esparsidade do modelo (quanto menor o número, maior a esparsidade, e potencialmente mais simples é o modelo) (DUCHI; HAZAN; SINGER, 2011). Analisando os resultados apresentados por Duchi, Hazan e Singer (2011) é possível ver uma superioridade dos algoritmos *AdaGrad* perante aos outros otimizadores avaliados, além disso, nota-se que o *AdaGrad-RDA* produz modelos mais esparsos quando comparado com a variante *AdaGrad-FB*.

A tabela com as taxas de erro no conjunto de testes para o dataset *Reuters RCV1*, apresentando as métricas encontradas pelos autores pode ser vista na Tabela 6.2

Tabela 6.2 – Taxas de erro no conjunto de teste e proporção de não-nulos (entre parênteses) no dataset *Reuters RCV1*

	RDA	FB	ADAGRAD-RDA	ADAGRAD-FB	PA	AROW
ECAT	.051 (.099)	.058 (.194)	**.044 (.086)**	**.044** (.238)	.059	.049
CCAT	.064 (.123)	.111 (.226)	**.053 (.105)**	**.053** (.276)	.107	.061
GCAT	.046 (.092)	.056 (.183)	**.040 (.080)**	**.040** (.225)	.066	.044
MCAT	.037 (.074)	.056 (.146)	.035 (**.063**)	**.034** (.176)	.053	.039

Fonte: Adaptado de (DUCHI; HAZAN; SINGER, 2011).

Já para o *ImageNet*, foram escolhidos apenas 4 algoritmos: o *AdaGrad-RDA*, o *AROW*, o *PA* e o *RDA*, e para avaliar esses otimizadores, foi utilizada como métricas a precisão média (Avg. Prec.), P@K (precisão em k, indicando a porcentagem das vezes em que a resposta correta esteve entre as k primeiras previsões), e também a proporção não-nula (DUCHI; HAZAN; SINGER, 2011). Com base nas análises feitas por Duchi, Hazan e Singer (2011), é possível tirar duas conclusões, a primeira delas é que o *AdaGrad* é o melhor modelo em precisão média além de atingir uma maior esparsidade, contudo o *AROW* ainda é uma melhor alternativa para as precisões em k, o que indica que ele é capaz de atingir as respostas corretas nas primeiras previsões.

Para comparar a precisão dos modelos utilizando o dataset *ImageNet* você pode analisar a Tabela 6.3

Tabela 6.3 – Precisão no conjunto de teste para o *ImageNet*

Alg.	Avg. Prec.	P@1	P@3	P@5	P@10	Prop. nonzero
ADAGRAD RDA	**0.6022**	0.8502	0.8307	0.8130	0.7811	**0.7267**
AROW	0.5813	**0.8597**	**0.8369**	**0.8165**	**0.7816**	1.0000
PA	0.5581	0.8455	0.8184	0.7957	0.7576	1.0000
RDA	0.5042	0.7496	0.7185	0.6950	0.6545	0.8996

Fonte: Adaptado de (DUCHI; HAZAN; SINGER, 2011).

O último comparativo que será visto é com relação ao *MNIST*, nele, Duchi, Hazan e Singer (2011) escolheram cinco algoritmos: o *PA*, o *Ada RDA*, o *RDA*, o *Ada RDA l1/l2* e o *RDA l1/l2*, sendo analisada a quantidade de erros pela quantidade de exemplos vistos. Nessa análise, os algoritmos que performaram melhor foram o PA seguido do Ada RDA, apresentando um comportamento parecido, enquanto isso, nota-se que utilizar as normalizações l1/l2 não trouxeram muitas vantagens para os algoritmos (DUCHI; HAZAN; SINGER, 2011).

As curvas de aprendizado dos diferentes modelos treinados pelos autores podem ser vistas na Figura 6.5, note que o *Ada RNA* apresenta uma performance muito parecida com o *PA*, apresentando pouco mais de 5.000 erros ao ser treinado com todo o *dataset*.

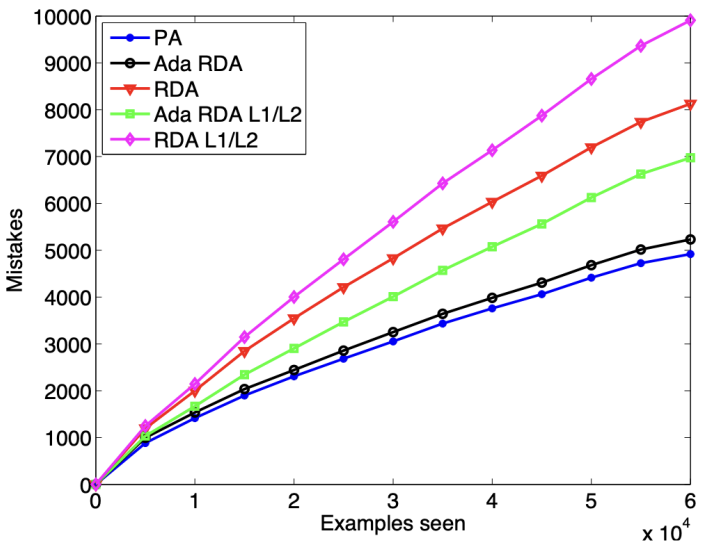


Figura 6.5 – Curvas de aprendizado no *dataset* MNIST.
Fonte: (DUCHI; HAZAN; SINGER, 2011).

O *AdaGrad* simbolizou não somente um avanço para os algoritmos de otimização, como também introduziu o conceito de diferentes taxas de aprendizado, permitindo com que um modelo possa aprender bem tanto características muito frequentes como características mais incomuns. Dessa forma, um novo periódo de otimizadores surgiu, seguindo os conceitos do *AdaGrad* mas buscando incrementar o seu algoritmo a fim

de corrigir falhas que ficaram para trás. Ao longo dessa seção, serão vistos mais otimizadores que atuam de forma parecida ao *AdaGrad*, adicinando melhorias para o seu desempenho.

6.4.2 RMSProp

Diferente dos outros métodos de otimização que surgiram em artigos, o *RMSProp* tem uma concepação diferente, sendo apresentado por Geoffrey Hinton nas suas notas de aula do curso *Neural Networks for Machine Learning* como um método que semelhante ao AdaGrad, dividia o gradiente (TIELEMAN; HINTON, 2012). Para isso, o *RMSProp* combina a ideia de somente usar o sinal do gradiente com a ideia de adptar o tamanho do passo separadamente para diferentes pesos, de forma que para funcionar em *mini-batches* é mantido uma média móvel do quadrado do gradiente para cada peso (TIELEMAN; HINTON, 2012).

Para inicilizar o *RMSProp* é preciso de cinco parâmetros: a taxa de aprendizado η, o vetor de parâmetros iniciais θ_0, a função de perda a ser otimizada $f(\theta)$, o parâmetro de estabilidade ϵ, para evitar divisões por zero, e também a taxa de decaimento ν, a qual irá controlar a média móvel dos gradientes. Antes de iniciar o loop do RMSProp é feita a inicialização do vetor acumulador com zeros, ele irá atuar como um acumulador para a média móvel do quadrados dos gradientes.

O primeiro passo ao se iniciar o loop do *RMSProp* é calcular $\mathbf{g}_t$, para isso ele irá receber o vetor gradiente da função de custo calculado para os parâmetros da iteração anterior θ_{t-1}, em seguida é atualizado o vetor acumulador da forma que $\mathbf{n}_t$ será dado pela multiplicação da taxa de decaimento ν pelo valor do vetor acumulador na iteração anterior, esse resultado é então somado ao quadrado produto a produto do gradiente $\mathbf{g}_t$ multiplicado por $(1 - \nu)$.

O último passo do *RSMProp* consistem em atualizar os parâmetros de θ_0 para isso é subtraído dos parâmetros anteriores θ_{t-1} a multiplicação da taxa de aprendizado pelo vetor gradiente $\mathbf{g}_t$ que é dividido pela raíz do vetor acumulador de parâmetros. Assim, esse loop se repete até atingir o número máximo de épocas escolhido na hora da implementação, ou no caso do pseudocódigo 5, ele irá repetir até o modelo convergir, ou seja, quando a norma do vetor gradiente for zero.

Algorithm 5 RMSProp

Requer: Taxa de aprendizado η
Requer: Vetor de parâmetros inicial θ_0
Requer: Função a ser otimizada $f(\theta)$
Requer: Parâmetro de estabilidade ϵ
Requer: Taxa de decaimento ν
$\mathbf{n}_0 \leftarrow 0$ ▷ Inicializar o vetor acumulador
while não convergir **do**
  $\mathbf{g}_t \leftarrow \nabla_{\theta_{t-1}} f(\theta_{t-1})$
  $\mathbf{n}_t \leftarrow \nu \mathbf{n}_{t-1} + (1-\nu)(\mathbf{g}_t \odot \mathbf{g}_t)$
  $\theta_t \leftarrow \theta_{t-1} - \eta \frac{\mathbf{g}_t}{\sqrt{\mathbf{n}_t + \epsilon}}$
end while
Retorne: θ_t ▷ Parâmetros resultantes

Perceba que o *RMSProp* consegue corrigir um dos problemas do *AdaGrad* ao atualizar o vetor acumulador utilizando uma taxa de decaimento ν. No *AdaGrad* o vetor acumalor só cresce, enquanto no RMSProp os termos mais antigos vão tendo seu valor reduzido gradativamente pelo coeficiente ν. Dessa forma, é possível melhor focar nos termos mais recentes que fazem parte do vetor acumulador.

6.4.3 Adaptive Moment Estimation (Adam)

Uma evolução natural dos algortitmos de otimização *AdaGrad* e *RMSProp* é o *Adam* ou *Adaptive Moment Estimation*. Esse algortimo foi apresentado pelos pesquisadores Kingma e Ba (2017), no artigo *Adam: A Method for Stochastic Optimization*, o qual veio para ser um método de otimização estocástica que requere apenas gradientes de primeira ordem com poucos requisitos de memória.

Para isso, o *Adam* computa taxas de aprendizado individuais para diferentes parâmetros para estimações do primeiro e segundo momento do gradiente, de forma que ele é capaz de combinar as vantagens do *AdaGrad*, que trabalha bem com gradientes esparsos, e o *RMSProp*, que trabalha bem *on-line* e em configurações não-estacionárias (KINGMA; BA, 2017).

O *Adam* recebe como parâmetros uma taxa de aprendizado η, os coeficientes de decaimento exponencial β_1 e β_2 para as estimativas do momento em um conjunto de intervalos $[0, 1)$. Além disso, recebe também o vetor contendo os parâmetros iniciais θ_0, a função de perda $f(\theta)$ para qual será encontrado o ponto de mínimo, e também o parâmetro de estabilidade ϵ para evitar divisões por zero.

Com todos esses parâmetros, o primeiro passo consiste em inicializar os vetores de primeiro $\mathbf{m}_0$ e segundo $\mathbf{v}_0$ momento bem como o passo do tempo t, para isso, os vetores são zerados e o tempo t também. Feito isso, o loop do Adam se inicia, terminando apenas quando o modelo convergir. Primeiro, é incrementado o tempo, e em seguida

calcula-se o gradiente $\mathbf{g}_t$ que irá receber o gradiente da função de perda $f(\theta)$ em relação aos parâmetros da iteração anterior.

A próxima etapa do Adam consiste em atualizar as estimativas des momentos, o primeiro momento $\mathbf{m}_t$ recebe o coeficiente de decaimento β_1 multiplicado pelo vetor do primeiro momento na iteração anterior, que é então somado a multiplicação de $(1 - \beta_1)$ pelo vetor gradiente $\mathbf{g}_t$. O segundo momento é calculado de forma semelhante, com a diferença de que são utilizados o vetor do segundo momento para fazer as incrementações e o uso do coeficiente β_2, além disso, na segunda parte da soma é feito o produto termo o termo dos vetores gradientes.

Seguindo adiante, deve-se calcular a correção do viés, para isso $\hat{\mathbf{m}}_t$ irá calcular a divisão do vetor do primeiro momento por $(1 - \beta_1)$, enquanto $\hat{\mathbf{v}}_t$ irá calcular a divisão do vetor do primeiro momento por $(1 - \beta_2)$.

Por fim, é feita a incrementação dos parâmetros de θ_0 de forma semelhante ao *AdaGrad*, eles serão dados pela diferença do vetor de parâmetros na iteração anterior pela multiplicação da taxa de aprendizado η com a divisão dos vetores do primeiro e segundo momento.

Esse método pode ser visto no bloco de algoritmo 6.

Algorithm 6 Adaptive Moment Estimation (Adam)

Requer: Taxa de aprendizado η (ex: 0.001)
Requer: Coeficientes de decaimento β_1, β_2 (ex: 0.9, 0.999)
Requer: Vetor de parâmetros inicial θ_0
Requer: Função a ser otimizada $f(\theta)$
Requer: Parâmetro de estabilidade ϵ (ex: 1e-7)

$\mathbf{m}_0 \leftarrow 0$ ▷ Inicializar vetor de 1º momento (média)
$\mathbf{v}_0 \leftarrow 0$ ▷ Inicializar vetor de 2º momento (variância)
$t \leftarrow 0$ ▷ Inicializar passo de tempo
while não convergir **do**
    $t \leftarrow t + 1$
    $\mathbf{g}_t \leftarrow \nabla_{\theta_{t-1}} f(\theta_{t-1})$
    $\mathbf{m}_t \leftarrow \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1)\mathbf{g}_t$
    $\mathbf{v}_t \leftarrow \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2)(\mathbf{g}_t \odot \mathbf{g}_t)$
    $\hat{\mathbf{m}}_t \leftarrow \frac{\mathbf{m}_t}{1-\beta_1^t}$
    $\hat{\mathbf{v}}_t \leftarrow \frac{\mathbf{v}_t}{1-\beta_2^t}$
    $\theta_t \leftarrow \theta_{t-1} - \eta \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}$
end while
Retorne: θ_t ▷ Parâmetros resultantes

Para testar esse novo algoritmo, os pesquisadores Kingma e Ba (2017) prepararam uma série de experimentos, sendo alguns deles envolvendo uma regressão logística, uma rede neural multi-camadas e uma rede convolucional.

No primeiro dos experimentos, os autores avaliam esse otimizador utilizado uma regularização L2 criando uma regressão logística para analisar o *dataset* de dígitos manuscritos *MNIST* (KINGMA; BA, 2017). Nos testes, Kingma e Ba (2017) comparam a sua técnica de otimização com o gradiente estocástico acelereado de Nesterov (*SGDNesterov*) e também como o *AdaGrad*; de forma que eles foram capazes de concluir que o *Adam* possui uma convergência similar ao *SGDNesterov*, equanto o *AdaGrad* apresenta uma convergência mais demorada e com um maior custo de treino. Esse compartivo pode ser visto na Figura 6.6.

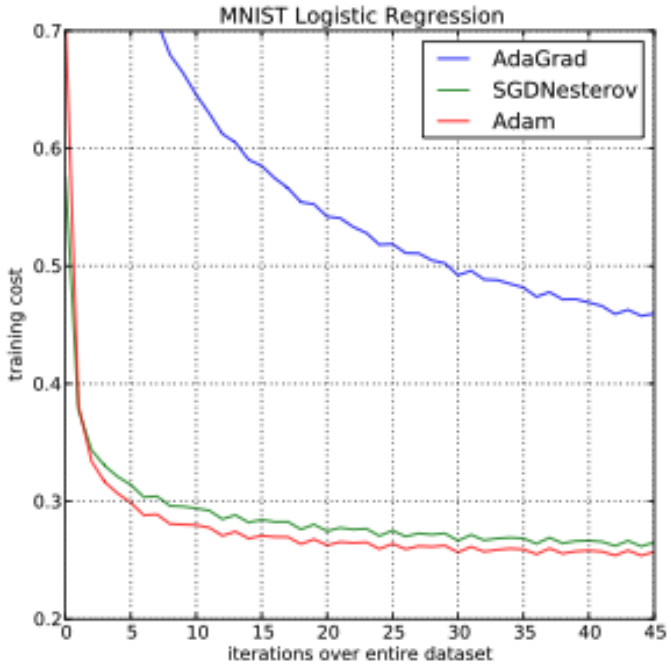


Figura 6.6 – Curvas de aprendizado no dataset MNIST.
Fonte: (KINGMA; BA, 2017).

No segundo experimento, os cientistas foram capazes de descobrir que mesmo em redes neurais multi-camadas (as quais são modelos com funções objetivas não-convexas), mesmo a análise de convergência feita não se aplicando a problemas não-convexos, o *Adam* foi capaz de performar melhor que outros métodos (KINGMA; BA, 2017). Para o teste então, Kingma e Ba (2017) criaram uma rede neural com duas camadas totalmente conectadas com 1000 unidades de neuônio utilizando a função de atiavção *ReLU* com *mini-batch* de tamanho 128, com base nos testes, eles foram capazes de perceber que o *Adam* consegui ser de 5 a 10 vezes mais rápido por iteração que o método *SFO* (*sum-of-functions*), o qual é um método *quasi-Newton* que trabalha também com dados em *mini-batches*. O comportamento desses diferentes otimizadores para uma rede neural multicamadas está retratado nos gráficos da Figura 6.7.

Já para a rede convolucional, Kingma e Ba (2017) criaram uma *CNN* com três cama-

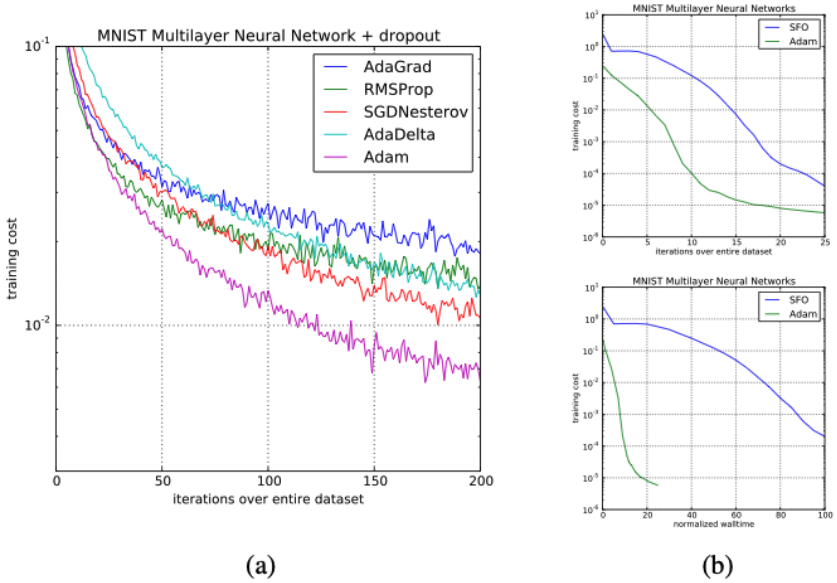


Figura 6.7 – Treinamento de redes neurais multicamadas em imagens *MNIST*. (a) Redes neurais utilizando regularização estocástica com *dropout*. (b) Redes neurais com função de custo determinística, em comparação com o otimizador soma de funções (*SFO*).
Fonte: (KINGMA; BA, 2017).

das, a primeira com filtros de dimensões 5x5, seguida de uma camada de *max pooling* de dimensões 3x3 com *stride* 2 que se liga a uma camada totalmente conectada de 1000 unidades ocultas que faz uso da função de ativação *ReLU*. Com base nas análises, os autores concluíram que tanto o *Adam* quando o *AdaGrad* fazem um progresso rápido no custo inicial do treino, em seguida, o *Adam* e o *SGD* convergem consideravelmente mais rápido que o *AdaGrad* (KINGMA; BA, 2017). É possível ver como o *Adam* e os outros otimizadores performam nessa tarefa na Figura 6.8.

Assim como os métodos *AdaGrad* e *RMSProp* foram grandes inspirações para o desenvolvimento do otimizador *Adam*, é inegável que o mesmo também passou a ser fonte de inspiração para pesquisas futuras, as quais o utilizam como base e faziam pequenos incrementos em seu método a fim de corrigir falhas. Dessa forma, a comunidade científica vive em um constante avanço incremental, pegando as partes boas e de um projeto e melhorando o que pode não estar tão bom. Os próximos otimizadores, são variantes do Adam, de forma que seguem esse mesmo princípio, de pegar algo que estava bom, mas falho em algumas partes e melhorá-lo.

6.4.4 AdaMax

Ainda em *Adam: A Method for Stochastic Optimization*, os autores Kingma e Ba (2017), além de apresentarem o *Adam* como um novo otimizador, eles também apresentam

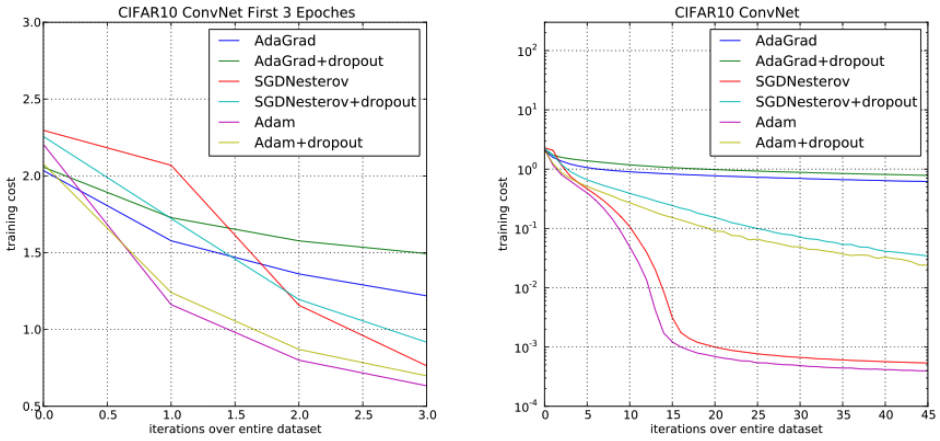


Figura 6.8 – Custo de treinamento de redes neurais convolucionais. À esquerda, o custo nas três primeiras épocas. À direita, o custo ao longo de 45 épocas.
Fonte: (KINGMA; BA, 2017).

uma variante dele, o *AdaMax*, que ao invés de utilizar a norma L^2 para calcular o segundo momento, utilizam a norma L^p, em que $p \to \infty$, provando ser um algoritmo surpreendemente simples e estável. Dessa forma, os autores foram chegram no otimizador listado no Algortimo 7.

Note que o *AdaMax* recebe como parâmetros de entrada os mesmos parâmetros do *Adam* tradicional, com a diferença sendo na inicilização do vetores, que agora, ao invés de inicilizar o vetor do segundo momento, é inicilizadada a norma infinita u_0. No loop do *AdaMax*, é possível ver um funcionamento semelhante ao do Adam, ele começa incrementando o passo de tempo t e e calculando o vetor de gradientes g_t para aquela iteração, que é então utilizado para atualizar o vetor do primeiro momento m_0.

A diferença do *AdaMax* está na próxima etapa, em que é atualizada a norma infinita utilizando como base $\max(\beta_2 \cdot u_{t-1}, |g_t|)$. Com base em todos esses parâmetros o *AdaMax* atualiza o vetor de parâmetros θ. Essas iterações são repetidas ate o modelo convergir para um ponto de mínimo.

Algorithm 7 AdaMax, uma variante do Adam baseada na norma infinita

Requer: Taxa de aprendizado η
Requer: Taxas de decaimento exponencial $\beta_1, \beta_2 \in [0, 1)$
Requer: Função objetivo estocástica com parâmetros θ, $f(\theta)$
Requer: Vetor de parâmetros inicial θ_0
$m_0 \leftarrow 0$ ▷ Inicializar vetor de 1º momento
$u_0 \leftarrow 0$ ▷ Inicializar a norma infinita exponencialmente ponderada
$t \leftarrow 0$ ▷ Inicializar passo de tempo
while não convergir **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ ▷ Obter gradientes em relação ao objetivo no passo t
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ ▷ Atualizar estimativa viciada do primeiro momento
    $u_t \leftarrow \max(\beta_2 \cdot u_{t-1}, |g_t|)$ ▷ Atualizar a norma infinita exponencialmente ponderada
    $\theta_t \leftarrow \theta_{t-1} - (\eta / (1 - \beta_1^t)) \cdot m_t / u_t$ ▷ Atualizar parâmetros
end while
Retorne: θ_t ▷ Parâmetros resultantes

6.4.5 Nesterov-accelerated Adaptive Moment Estimation (Nadam)

Com o *Nadam*, ou *Nesterov-accelerated Adaptive Moment Estimation*, ocorreu a mesma ideia de incremento presente nos outros métodos de otimização vistos até o momento, só que dessa vez incluindo uma técnica já conhecida. Neste caso, Dozat (2016), decidiu apresentar no trabaho *Incorporating Nesterov Momentum Into Adam*, uma modificação do componente de momento do Adam utilizando como base o método de otimização do gradiente acelerado de Nesterov (*NAG*). Ao fazer essa mudança, o autor conseguiu comprovar que foi possível aumentar a velocidade de convergência, assim como a qualidade do aprendizado do modelo (DOZAT, 2016).

O pseudocódigo que representa o otimizador *Nadam* está representado no Algortimo 8

O *Nadam* recebe como parâmetros de entrada uma taxa de aprendizado η, os coeficientes de decaimento μ e ν, o vetor contendo os parâmetros iniciais θ_0, a função de perda que será utilizada para otimizar o modelo $f(\theta)$ e o parâmetro de estabilidade ϵ para evitar que divisões por zero ocorram. Em seguida, são inicilizados os os vetores do primeiro $\mathbf{m}_0$ e segundo $\mathbf{v}_0$ momento junto com o passo de tempo t.

Com tudo isso feito, é possível começar o loop do *Nadam* que irá terminar quando o modelo convergir. O primeiro passo é incrementar o passo de tempo t bem como calcular o vetor gradiente $\mathbf{g}_t$ que irá receber os resultados do vetor gradiente da função de perda calculados para os parâmetros da interação anterior θ_{t-1}. Em seguida, calcula-se as estimativas dos momentos $\mathbf{m}_t$ que irá receber a multiplicação do coeficiente de dacaimento μ multiplicado pela estimativa do momento $\mathbf{m}_t$ na iteração anterior somado

a multiplicação do vetor gradiente $\mathbf{g}_t$ com $(1-\mu)$. De forma pareciada, é calculada da estimativa do segundo momento $\mathbf{v}_t$ que será dada pela multiplicação do coeficiente de dacaimento ν com a sua versão na iteração anterior $\mathbf{v}_{t-1}$ somada ao produto termo a termo do vetor gradiente $\mathbf{g}_t$ que é então multiplicado por $(1-\nu)$.

O próximo passo do *NAdam* consiste em corrigir o viés, é nessa parte que ele aplica o momento de Nesterov. O viés do primeiro momento $\hat{\mathbf{m}}_t$ recece o produto do coeficiente de dacaimento μ multiplicado com a estimativa do primeiro momento $\mathbf{h}_t$ que é dividida por $(1-\mu^t)$ e então somada por $(1-\mu)$ multiplicada pelo vetor gradiente $\mathbf{g}_t$ dividido por $(1-\mu^t)$. Também é corrigido o viés do segundo momento $\hat{\mathbf{v}}_t$ que recebe a divisão do coeficiente de dacaimento ν multiplicado pelo vetor do segundo momento com $1-\nu^t$. A partir desses vetores, é feita por último a atualização do vetor de parâmetros θ.

Algorithm 8 Nesterov-accelerated Adaptive Moment Estimation (Nadam)

Requer: Taxa de aprendizado η (pode ser agendada, ex: η_t)
Requer: Coeficientes de decaimento μ, ν (ex: 0.9, 0.999)
Requer: Vetor de parâmetros inicial θ_0
Requer: Função a ser otimizada $f(\theta)$
Requer: Parâmetro de estabilidade ϵ (ex: 1e-7)

$\mathbf{m}_0 \leftarrow 0$ ▷ Inicializar vetor de 1º momento (média)
$\mathbf{v}_0 \leftarrow 0$ ▷ Inicializar vetor de 2º momento (variância)
$t \leftarrow 0$ ▷ Inicializar passo de tempo
while não convergir **do**
  $t \leftarrow t+1$
  $\mathbf{g}_t \leftarrow \nabla_{\theta_{t-1}} f_t(\theta_{t-1})$
  $\mathbf{m}_t \leftarrow \mu\mathbf{m}_{t-1} + (1-\mu)\mathbf{g}_t$
  $\mathbf{v}_t \leftarrow \nu\mathbf{v}_{t-1} + (1-\nu)(\mathbf{g}_t \odot \mathbf{g}_t)$
  $\hat{\mathbf{m}}_t \leftarrow (\mu\mathbf{m}_t/(1-\mu^t)) + ((1-\mu)\mathbf{g}_t/(1-\mu^t))$
  $\hat{\mathbf{v}}_t \leftarrow \frac{\nu\mathbf{v}_t}{1-\nu^t}$
  $\theta_t \leftarrow \theta_{t-1} - \eta\frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t}+\epsilon}$
end while
Retorne: θ_t ▷ Parâmetros resultantes

Utilizando os otimizadores *SGD*, *Momentum*, *Nesterov*, *RMSProp*, *Adam* e *Nadam*, Dozat (2016) foi capaz de criar um *autoencoder* convolucional com três camadas de convolução e duas camadas densas em cada encoder e o decoder para comprimir as imagens do *dataset MNIST* em um espaço vetorial de 16 dimensões e com isso reconstruir a imagem original. Assim, os resultados da pesquisa do autor podem ser vistos nos gráficos da Figura 6.9, note que comparando com o clássico gradiente estocástico, o *NAdam* consegue uma vantagem de cerca de 0.010 de diferença na taxa de erro utilizando a *MSE* para calcular o erro. Não somente isso, mas ele é o modelo que apresenta uma convergência mais acelerada quando comparado com os outros, perceba que que

o decaimento da taxa de erro do *NAdam* é consideravelmente maior nas primeiras épocas, enquanto a partir das 10 épocas ele já começa a se estabilizar.

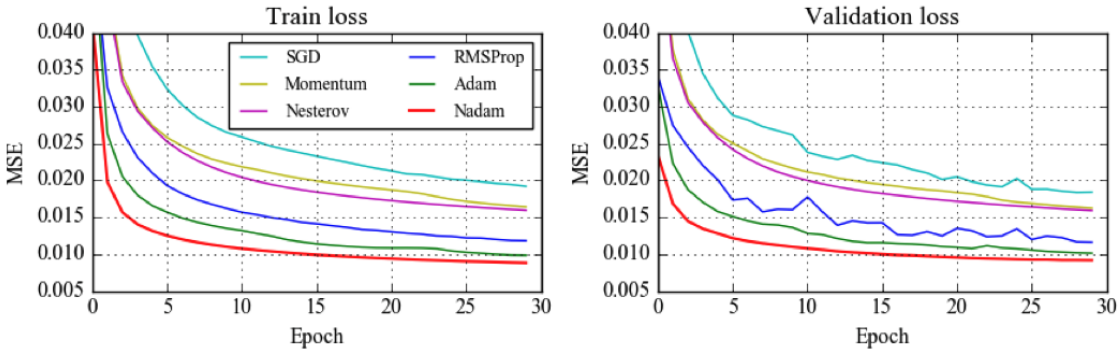


Figura 6.9 – Perda de treinamento e validação de diferentes otimizadores no dataset *MNIST*.
Fonte: (DOZAT, 2016).

Comparando com o *Adam* tradicional, a principal diferença do *NAdam* com esse outro otimizador está justamente relacionada ao decaimento do erro, que é consideravelmente maior, enquanto o *NAdam* já estava se estabilizando com cerca de 10 épocas, o *Adam* parece demorar mais. De fato como apresentado por Dozat (2016), o *NAdam* é capaz de acelerar as taxas de convergência do modelo que está sendo treinado.

6.4.6 Adam With Decoupled Weight Decay (AdamW)

Uma outra variante do *Adam* tradicional é o *AdamW*. Essa variante foi apresentada no trabalho *Decoupled Weight Decay Regularization* dos pesquisadores Loshchilov e Hutter (2019), em que eles propuseram uma forma diferente de corrigir como a relgularização L^2, era feita no método original. Para isso, o *AdamW* desacopla o decaimento do peso da atualização principal do do gradiente, e o aplica como um passo separado e final na atualização dos parâmetros (LOSHCHILOV; HUTTER, 2019).

Esse algoritmo pode ser visto no psudeocódigo apresentado no bloco 9.

o *AdamW* recebe como entradas a taxa de aprendizado η, os coeficientes de decaimento β_1 e β_2, o fato de decaimento de peso λ, o vetor contendo os parâmetros iniciais do modelo θ_0, a função de perda a ser otimizada $f(\theta)$ e o parâmetro de estabilidade ϵ. Semelhante ao *Adam*, o *AdamW* começa inicializando os vetores de primeiro $\mathbf{m}_0$ e segundo $\mathbf{v}_0$ momento bem como o passo de tempo t.

Feito isso, é iniciado o loop do *AdamW* que irá terminar quando o modelo convergir. Ele começa da mesma forma que os outros otimizadores vistos até agora, incrementa o passo de tempo t, e calcula o vetor gradiente da função de perda no passo anterior e o adiciona em $\mathbf{g}_t$. Seguindo adiante são feitas as estimativas dos momentos de forma

semelhante ao *Adam*, primeiro calcula-se o primeiro momento $\mathbf{m}_t$ que irá receber a multiplicação do coeficiente de decaimento β_1 com o momento anterior $\mathbf{m}_{t-1}$ somado a multiplicação do vetor gradiente $\mathbf{g}_t$ com $(1-\beta_1)$. De forma semelhante é possível calcular o segundo momento $\mathbf{v}_t$ o qual será dado pela multiplicação do coeficiente de decaimento β_2 com o momento anterior $\mathbf{v}_{t-1}$ soamado ao produto termo da termo do vetor gradiente $\mathbf{g}_t$ com $(1-\beta_2)$.

O próximo passo do *AdamW* é a correção do viés. Começando com $\hat{\mathbf{m}}_\mathbf{t}$ que irá receber a divisão do vetor de momentos $\mathbf{m}_t$ com $(1-\beta_1^t)$. Já $\hat{\mathbf{v}}_t$ é calculado de forma semelhante, ele irá receber a divisão do vetor de segundo momento $\mathbf{v}_t$ com $(1-\beta_2^t)$.

A diferença do *AdamW* está nessa próxima parte, o cálculo dos parâmetros θ para a próxima iteração. Para isso, eles serão dados pela diferença dos parâmetros na iteração anterior θ_{t-1} com a multiplicação da taxa de aprendizado η com a divisão do vetor do primeiro momento já corrigido pela raiz do vetor de segundo momento também já corrigido, isso é então somado ao produto do fator de dacaimento de peso λ pelos parâmetros da iteração enterior θ_{t-1}.

Algorithm 9 Adam com Decaimento de Peso Desacoplado (AdamW)

Requer: Taxa de aprendizado η (ex: 0.001)
Requer: Coeficientes de decaimento β_1, β_2 (ex: 0.9, 0.999)
Requer: Fator de decaimento de peso λ (ex: 0.01)
Requer: Vetor de parâmetros inicial θ_0
Requer: Função a ser otimizada $f(\theta)$
Requer: Parâmetro de estabilidade ϵ (ex: 1e-8)

$\mathbf{m}_0 \leftarrow 0$ ▷ Inicializar vetor de 1º momento (média)
$\mathbf{v}_0 \leftarrow 0$ ▷ Inicializar vetor de 2º momento (variância)
$t \leftarrow 0$ ▷ Inicializar passo de tempo
while não convergir **do**
  $t \leftarrow t+1$
  $\mathbf{g}_t \leftarrow \nabla_{\theta_{t-1}} f(\theta_{t-1})$ ▷ O gradiente NÃO inclui o termo de decaimento
  $\mathbf{m}_t \leftarrow \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t$
  $\mathbf{v}_t \leftarrow \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)(\mathbf{g}_t \odot \mathbf{g}_t)$
  $\hat{\mathbf{m}}_t \leftarrow \frac{\mathbf{m}_t}{1-\beta_1^t}$
  $\hat{\mathbf{v}}_t \leftarrow \frac{\mathbf{v}_t}{1-\beta_2^t}$
  $\theta_t \leftarrow \theta_{t-1} - \eta \left(\frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t}+\epsilon} + \lambda \theta_{t-1} \right)$
end while
Retorne: θ_t ▷ Parâmetros resultantes

Para compravar a eficácia do desacoplamento de peso, Loshchilov e Hutter (2019) decidem availiar o *Adam* com reguralização L2 com a nova variante *AdamW* utilizando uma 26 2x64d *ResNet* no *CIFAR-10* medida depois de 100 épocas, além disso, os autores avaliam também o *SGD* e sua variante com peso desacoplado, o *SGDW*, de forma que os resultados podem ser vistos na Figura 6.10.

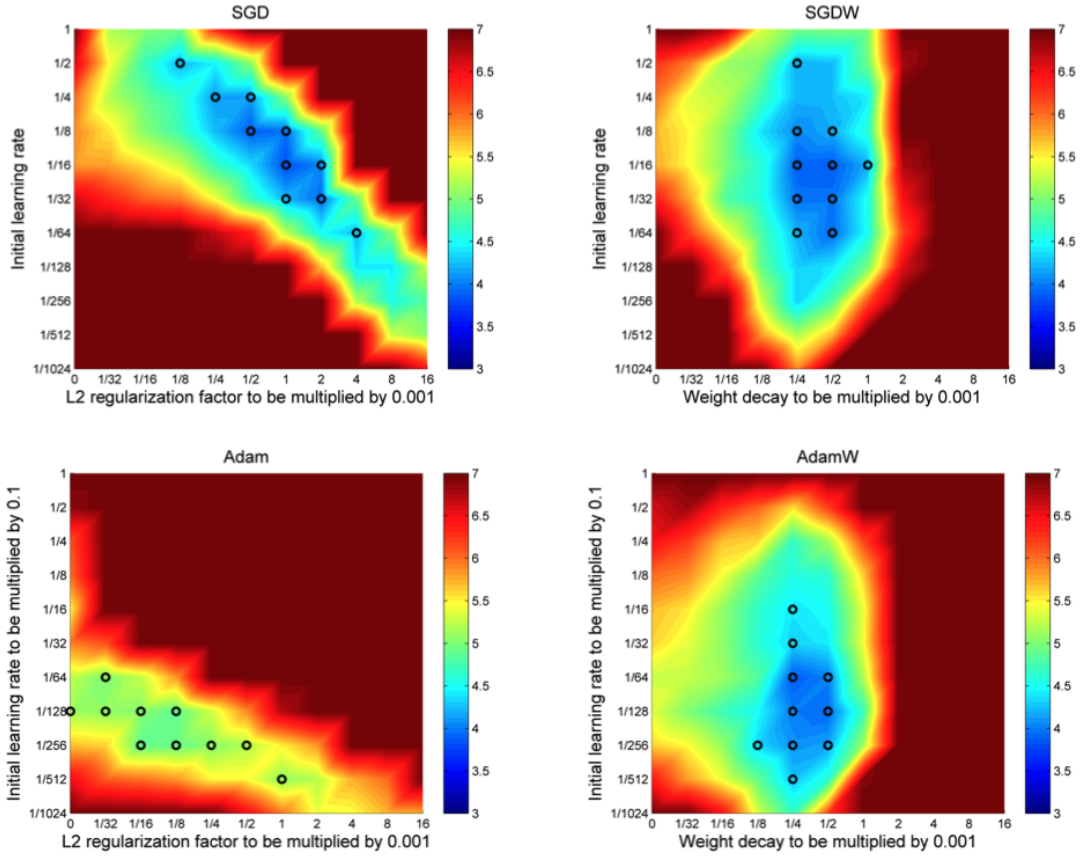


Figura 6.10 – Erro de teste Top-1 de uma ResNet 26 2x64d no CIFAR-10, medido após 100 épocas. Os métodos propostos SGDW e AdamW (coluna da direita) possuem um espaço de hiperparâmetros mais separável.
Fonte: (LOSHCHILOV; HUTTER, 2019).

Nessas figuras, o eixo vertical representa a taxa de aprendizado, enquando o eixo horizontal representa a forca de regulariação ou do decaimento do peso. Enquanto isso, as cores indicam o desempenho do momedo, azul e ciano representam um bom desempenho, com baixo erro, enquanto vermelho representa um desemepho pior, com erro mais alto.

Note que ambas as variantes do gradiente estocástico apresentam partes azuis, no *SGD* (a esquerda), a área de de bom desempenho é pequena, além de formar uma faixa diagonal estreita, o que indica que a taxa de aprendizado ideal e a força de reguralização são altamentes dependentes uma da outra. Isso significa que se você escoler um valor da taxa de aprendizado fora dessa faixa, utilizar um modelo com o *SGD* seria difícil para otimizar.

Por outro lado, no *SGDW* a área azul é muito maior além de ser mais vertical, indicando que existe uma ampla variante de taxas de aprendizado que você pode escolher e ainda sim conseguir um modelo bem otimizado. De forma que a escola de um bom decaimento de peso e de uma taxa de aprendizado são mais independentes, e com isso o modelo pode ser mais fácil de otimizar.

Quando comparamos agora as duas variantes do *Adam* vemos um cenário diferente. Note que o resultado do *Adam* é bem ruim, o gráfico é quase todo vermelho, o que mostra que o *Adam* com a reguralização L2 falha em convergir para melhores pontos de mínimo para a maioria das taxas de aprendizado apresentadas.

Agora se analisar o *AdamW* o resultado é bem mais positivo. Assim como o *SGDW* existe uma linha vertical em azul no *AdamW*, o que nos mostra que é possível escolher uma quantidade grande de taxas de aprendizado que permitam ao modelo uma boa convergência, além é claro dela ser mais independente com relação do decaimento de peso.

Assim, utilizar o *AdamW* pode ser uma excelente alternativa caso queira tentar taxas de aprendizado diferentes e ainda sim obter uma boa convergência do modelo. Por exemplo, se quiser você pode escolher uma taxa de aprendizado mais alta, permitindo com que o modelo dê passos mais largos e assim consiga convergir de forma mais rápida. Mas você ainda tem a oportunidade de utilizar taxas de aprendizado menores e provavelmente irá encontrar um ponto de mínimo.

6.4.7 Comparativo dos Otimizadores Modernos

Por fim, com base nos otimizadores vistos nessa seção, assim como foi feito na seção de otimizadores clássicos, é possível fazer uma tabela contendo as suas principais características. Para isso, tem-se então a Tabela 6.4

6.4.8 Outros Otimizadores

Rectified Adam (RAdam)

Uma outra variante do *Adam* que também busca melhorar o otimizador original é o *Rectified Adam* ou *RAdam*. Ele surgiu no trabalho *On the Variance of the Adaptive Learning Rate and Beyond* dos autores Liu *et al.* (2021) como uma atualização do Adam que buscava corrigir um problema identificado com a taxa de aprendizado adaptativa do otimizador: a sua variância é grande problemática nos primeiros estágios; de forma que eles propusaram o *RAdam*, uma variante que introduz um termo para retificar a variância da taxa de aprendizado adpativa.

Lion

Tabela 6.4 – Comparativo de otimizadores modernos

Otimizador	Inovação principal
AdaGrad	Faz o uso de múltiplas taxas de aprendizado, garantindo que tanto características comuns, quanto incomuns possam ser aprendidas pelo modelo
RMSProp	Utiliza um vetor com a média móvel do quadrado dos gradientes que é multiplicado por uma taxa de decaimento, impedindo que o vetor acumulador só cresça.
Adam	Combina as ideias do *AdaGrad* e *RMSProp* utilizando médias móveis, uma para os gradientes (primeiro momento), e outra para o quadrado dos gradientes (segundo momento).
AdaMax	Uma variante do Adam baseado na norma infinita. Para isso, ao invés de utilizar a média dos quadrados dos gradientes, ele utiliza o máximo entre o gradiente atual e a média anterior.
Nadam	Adiciona as ideias do *NAG* no *Adam*, para isso, ele aplica a ideia de *lookahead* ao cálculo da média móvel dos gradientes (primeiro momento).
AdamW	Desacopla o decaimento de peso da etapa de atualização do gradiente, aplicando-o diretamete aos pesos, dessa forma, ele consegue corrigir a forma como a regularização do decaimento de peso (L2) é aplicado no *Adam*.

Fonte: O autor (2025).

6.5 Comparativo de Desempenho: Otimizadores

6.6 O Método de Newton: Indo Além do Gradiente

O método do gradiente se enquadra no tipo de **método de primeira ordem**, pois faz o uso de apenas derivadas de primeira ordem na sua implementação. Existem métodos que vão além das derivadas de primeira ordem buscando soluções que fazem uso de **derivadas de segunda ordem**. Um desses métodos é o **método de Newton**.

Essa técnica foi inicialmente proposta por Isaac Newton nos trabalhos "*De analysi per aequationes numero terminorum infinitas*" em 1669 e "*De metodis fluxionum et serierum infinitarum*" escrito em 1671. Entretanto, mais tarde Joseph Raphson apresentou uma versão aprimorada em seu livro "*Aequationum Universalis*" em 1690. Devido às contribuições de ambos os matemáticos, essa técnica também é conhecida como método de Newton-Raphson.

6.6.1 Conceitos iniciais: Matrizes Jacobianas e Hessianas

Como dito anteriormente, o método de Newton faz uso de derivadas de segunda ordem em sua fórmula. Nisso, surgem dois conceitos que devem ser explicados para entendermos melhor esse método: as matrizes jacobianas e hessianas.

Chamamos de **matriz jacobiana** a matriz que contém todos as derivadas parciais de uma função de n variáveis $f(x, y, ..., n)$. Com base nas derivadas de primeira ordem, podemos ter uma melhor noção de como a função cresce ou decresce em intervalos do eixo.

$$J_f(x, y, ..., n) = \begin{bmatrix} \frac{\partial F_1}{\partial x} & \frac{\partial F_1}{\partial y} & \cdots & \frac{\partial F_1}{\partial n} \\ \frac{\partial F_2}{\partial x} & \frac{\partial F_2}{\partial y} & \cdots & \frac{\partial F_2}{\partial n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial x} & \frac{\partial F_m}{\partial y} & \cdots & \frac{\partial F_m}{\partial n} \end{bmatrix} \tag{6.18}$$

Já a **matriz hessiana** é uma matriz que contém todas as derivadas de segunda ordem da função $f(x, y, ..., n)$. Essas derivadas nos dão informações sobre a curvatura a função.

$$H_f(x, y, ..., n) = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} & \cdots & \frac{\partial^2 f}{\partial x \partial n} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} & \cdots & \frac{\partial^2 f}{\partial y \partial n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial n \partial x} & \frac{\partial^2 f}{\partial n \partial y} & \cdots & \frac{\partial^2 f}{\partial n^2} \end{bmatrix} \tag{6.19}$$

Assim, essas matrizes, por nos mostrarem como a função varia, podem ser muito úteis para nos auxiliar a encontrar pontos mínimos, máximos e de sela da função e assim otimizá-la.

Com isso, podemos entender o método de newton.

6.6.2 O Método

O método de Newton é baseado na expansão de uma função $f(x)$ em uma **série de Taylor de segunda ordem** em torno de um ponto $x(0)$. Com essa aproximação, podemos encontrar de forma mais rápida pontos críticos da função.

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^T(\mathbf{x} - \mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^T H_f(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) \tag{6.20}$$

$$x' = x - H_f(x)^{-1}\nabla f(x) \tag{6.21}$$

Quando estamos trabalhando com uma função que pode ser aproximada por uma função quadrática no ponto analisado, o método de Newton pode nos retornar o resultado de forma mais rápida que o método do gradiente. Contudo, se isso não for o caso, ele levará mais iterações para encontrar um ponto crítico e assim convergir.

Por mais que possa ser mais rápido que o método do gradiente em algumas situações, o método de Newton não é perfeito. Se estivermos próximo tanto de ponto de sela quanto de um ponto de mínimo, o método de Newton pode fornecer direções equivocadas e levar para o ponto de sela ao invés do ponto de mínimo. Assim, a não será encontrado um valor ideal para otimizar aquela função de custo.

Além disso, pelo fato do método de Newton envolver o cálculo das derivadas de segunda ordem da função de custo, pois, diferente do método do gradiente que calcula somente as derivadas de primeira ordem, no método de Newton-Raphson são calculadas também as derivadas parciais de segunda ordem para a matriz hessiana.

Esse custo de poder de processamento será proporcional a quantidade de variáveis que a função de custo recebe, se f for uma função de várias variáveis, pode ser inviável o uso do método de newton devido à maior quantidade de cálculos envolvidos.

Portanto ao criar uma rede neural devemos escolher com sabedoria qual método de otimização devemos usar. O método do gradiente pode ser eficiente para minimizar uma função de custo de uma rede neural, mas em casos que essa convergência seja lenta, o método de Newton-Raphson pode ser uma melhor alternativa.

Implementação em Python

7 Funções de Ativação Sigmoidais

"Ué, cadê o gradiente que estava aqui?"
— Estagiário descobrindo o problema do desaparecimento de gradientes

Conhecendo como uma rede neural aprende, é possível agora entender outros pontos que são essenciais para o seu funcionamento, sendo um deles as funções de ativação. O principal objetivo de se adicionar uma função de ativação como a sigmoide após a camada densa de modelo está voltado a introdução da não-lineridade. Dessa forma, um modelo de rede neural torná-se mais capaz de aproximar uma gama de funções que antes não eram possíveis com funções de ativação lineares.

Esse capítulo se inicia justamente discutindo a importância de introduzir a não-linearidade para uma rede que está sendo criada, como justificativa são citados alguns dos diferentes teoremas da aproximação universal. O restante do captítulo é dedicado as funções sigmoidais, começando pela sigmoide, e como elas surgiram no cenário de aprendizado de máquina como uma forma de replicar o comportamento dos neurônios humanos em redes artificiais.

Além disso, são discutidas outras funções, como a tangente hiperbólica (tanh) e a *softsign*. Seguindo adiante é explicada uma das principais desvantagens de se utilizar funções sigmoidais em uma rede: o problema do desaparecimento do gradiente. E como ele se tornou um empecilho para a criação de redes neurais mais profundas. No fim do capítulo, está uma tabela resumindo as principais características das funções de foram apresentadas.

7.1 Teoremas da Aproximação Universal: Introduzindo a Não-Linearidade

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

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

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 7.1 e queremos encontrar uma reta que tente passar o mais próximo de cada um deles.

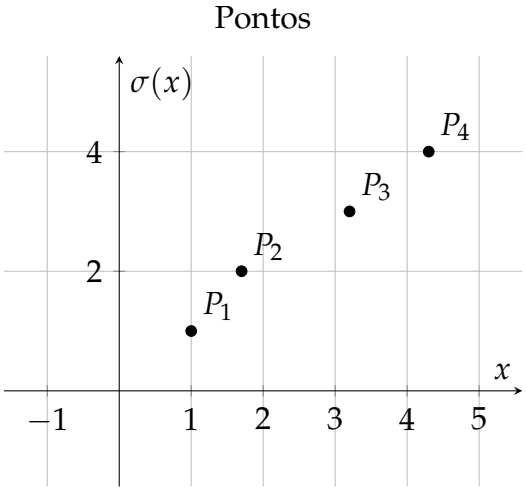


Figura 7.1 – Conjunto de pontos dispostos no plano cartesiano.

Fonte: O autor (2025).

Existe uma técnica que permite que nos façamos isso, ela se chama regressão linear (a qual é um dos tópicos discutidos no Capítulo 12), 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 7.2

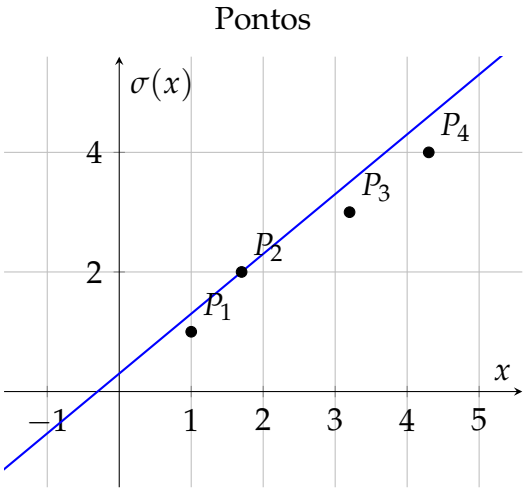


Figura 7.2 – Conjunto de pontos dispostos no plano cartesiano.

Fonte: O autor (2025).

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 $\mathbb{R}^N$ 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 rede neural 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.

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

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 Propriedades das Funções de Ativação: Escolhendo Uma Boa Função de Ativação

Visto um dos motivos de se utilizar funções de ativação em uma rede neural, cabe agora entender um pouco dessa classe de funções. Em *A Survey on Activation Functions and their relation with Xavier and He Normal Initialization*, Datta (2020) discute algumas das propriedades das funções de ativação, dizendo que é esperado que essas funções sejam: não-lineares, diferenciáveis, contínuas, limitadas, centradas em zero. Além disso, o autor discute também que é interessante considerar também o custo computacional dessas funções que estão sendo aplicadas em uma rede neural (DATTA, 2020).

Dessa forma, é possível discutir um pouco de cada uma dessas propriedades em seguida.

Não-linearidade

Como foi visto na seção anterior, o principal objetivo de uma função de ativação ser não linear é justamente relacionado ao fato delas permitirem que um modelo seja capaz de aproximar diversos tipos de funções desde que sejam mensuradas por Borel. Além disso, Datta (2020) justifica que caso tenha uma rede como um *Perceptron* com múltiplas camadas ocultadas pode ser facilmente comprimido em um *percetron* de apenas uma camada. Isso acontece porque a várias transformações lineares (uma para cada camada) em cima de outras também lineares continuam formando uma transformação linear quando serem consideradas por inteiro. Com isso, o principal benefício de uma rede neural, que é a profundidade, acaba por não ter tanto efeito em casos em que são aplicadas funções de ativação lineares.

Diferenciabilidade

O principal motivo de considerar a diferenciabilidade ao se escolher uma função de ativação está intrisicamente ligado em como a retropropagação do erro. Como visto no Capítulo 6, a retropropagação utiliza das derivadas parciais, do vetor gradiente e

também da derivada da função de ativação para calcular o erro e com base neste, as atualizações dos pesos e vieses da rede. Perceba que escolher uma função que tenha uma derivada complexa ou não seja derivada em grande parte do seu domínio, acaba por atrasar o algoritmo da retropropagação, que levará mais tempo tendo que calcular a derivada da função de ativação ou terão que ser feitas adaptações na derivada da função original para adequá-la ao novo algoritmo.

Continuidade

A continuidade está ligada com a diferencibilidade. Pela definição de derivada, não é possível derivar uma função em pontos de descotinuidade. Dessa forma, é importante considerar como o a função é representada graficamente, garantindo que não haja "quinas"no desenho da função e que o gráfico pode ser desenhado "sem tirar o lápis do papel". Técnicas como essas ajudam a descobrir ao olhar o gráfico, se existem pontos de descontinuidade os quais podem afetar a derivada da função de ativação.

Limitada

Considerar que uma função de ativação seja limitada ao escolher uma função de ativação para construir uma rede neural está ligado com os problemas do gradiente. Neste caso, uma função que não é limitada pode sofrer do problema do gradiente explosivo, o qual é explicado com maiores detalhes no Capítulo 8. Nesse sentido, o problema do gradiente explosivo o gradiente de uma rede neural começa a crescer de forma acelerada afetando o aprendizado da rede e as atualizações de pesos e vieses.

Centrada em zero

Uma função que não é centrada em zero, que é sempre positiva ou sempre negativa, afeta como a saída ocorre (DATTA, 2020). Como explica Datta (2020), um cenário em que isso acontece, significa que a saída está sendo sempre movida para os valores positivos ou negativos, e como resultado, o vetor de pesos precisa de uma quantidade maior de iterações para ser melhor ajustado. Dessa forma, ao escolher uma função que é centrada em zero, é possível garantir um treinamento com menos iterações mas ainda assim garantir uma boa convergência do modelo que está sendo treinado.

Custo computacional

Cabe considerar também o custo computacional de uma função de ativação. Funções mais simples, como a degrau unitário e a *ReLU*, as quais fazem apenas comparações com o valor da entrada, apresentam um custo computacional bem menor quando comparadas com funções mais complexas, como aquelas que fazem uso de muitos exponenciais em sua fórmula. Com isso, ao utilizar uma função de ativação muito "cara"em termos computacionais, isso acaba por gerar um gargalo no tempo de treino do modelo em desenvolvimento.

Vale destacar também que muitas dessas propriedades não estão presentes em todas as funções de ativação. A sigmoide por exemplo é uma função que não está centrada em zero, possuindo apenas saídas positivas. Já a *ReLU* apresenta um ponto de

descontinuidade na origem, mas isso não atrapalha essas funções de ativação de serem utilizadas ao construir uma rede neural. As propriedades servem mais como um guia, destacando vantagens que se pode ter ao utilizar determinada função de ativação.

Considerando isso, é possível finalmente entrar no tópico principal desse capítulo: as funções sigmoidais. Para isso, será explicado antes um exemplo ilustrativo destacando como essas funções atuam ao receberem suas entradas.

7.3 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.

7.4 A Sigmoide Logística: Ótima para Classificações Binárias

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 (1845) 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.

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 (RUMELHART; HINTON; WILLIAMS, 1986).

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, Rumelhart, Hinton e Williams (1986) 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 e Cowan (1972), 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 (WILSON; COWAN, 1972). Para modelar essa atividade, Wilson e Cowan (1972) fizeram o uso uma variação da função sigmoide, representada para Equação 7.1, 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 7.3, foi utilizado os mesmos valores escolhidos pelos cientistas na pesquisa, assim $a = 1.2$ e $\theta = 2.8$.

Wilson e Cowan (1972) demonstram que a população de neurônios reage de formas distintas quando sofrem determinados estímulos, os níveis baixos de excitação

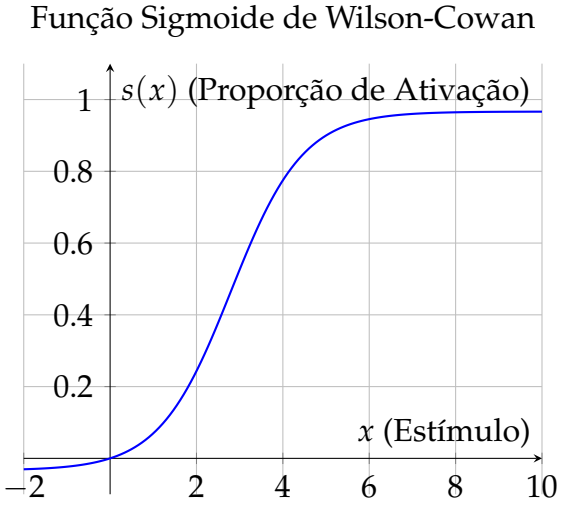


Figura 7.3 – Gráfico da função sigmoide conforme proposta por Wilson e Cowan (1972), com parâmetros de exemplo $a = 1.2$ e $\theta = 2.8$.

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 (WILSON; COWAN, 1972). Ao juntar todos esses três níveis, tem-se 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, Elman (1990) cita em seu estudo *Finding structure in time* 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 logística.

Cabe destacar também que, as sigmoidais 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 unitário em português, ela está representada na Figura 7.4. Essa função inclusive esteve presente na produção da rede neural *Perceptron* criada por Rosenblatt (1958) e introduzida para a comunidade científica no artigo *The Perceptron: A probabilistic model for informations storage and organization in the brain* no final dos anos 50.

Ao comparar a degrau unitário com a sigmoide, é possível notar uma diferença

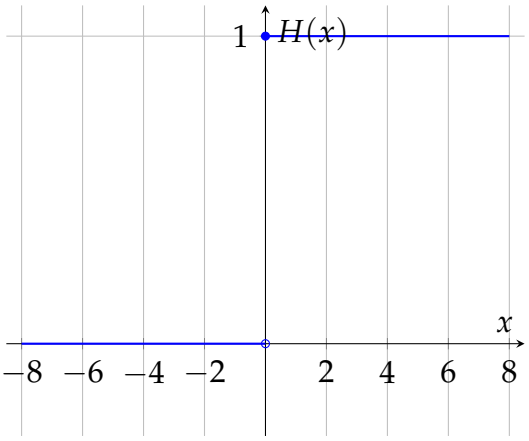


Figura 7.4 – Gráfico da função degrau unitário (*heaviside*).

Fonte: O autor (2025).

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.

A função sigmoide é escrita com uma exponencial, como na fórmula 7.2. Como explica Lederer (2021), a sigmoide 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.

Sigmoide Logística

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

O gráfico da sigmoide está presente na Figura 7.5, note que ele possui o formato de um "S"deitado. Assim, também é possível 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 (2021) 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. Essa propriedade de retornar valores em um intervalo de 0 a 1 é bem útil quando queremos fazer uma classificação binária, para isso, utilizamos a sigmoide aplicada na última camada densa de neurônios, limitando os intervalos, de forma que podemos interpretá-los como probabilidades, em que quanto mais próximo de 1, mais próximo aquele resultado está de uma classe A, por exemplo, já valores mais distantes pertenceriam a uma classe B.

Um ponto a ser destacado nas funções de ativação, e que começou a ser visto no ca-

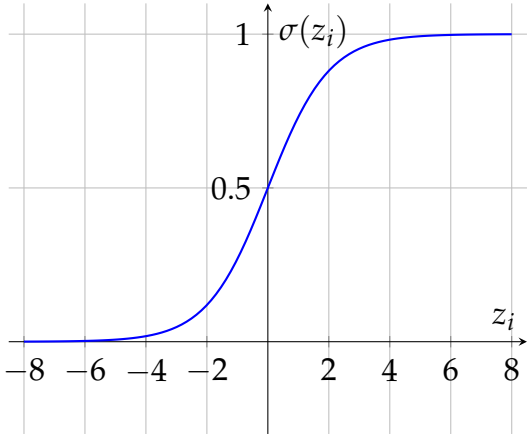


Figura 7.5 – Gráfico da função de ativação sigmoide logística.

Fonte: O autor (2025).

pítulo anterior (Capítulo 6), é que se estamos construindo um modelo que aprende por meio da retropropagação, nós estamos também interessados em entender como essas funções de ativação se comportam em suas derivadas, pois ela é um dos componentes básicos para se calcular o gradiente retropropagado.

Assim, ao derivar a sigmoide logística é possível encontrar a Equação 7.3.

Derivada da sigmoide logística

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

Um ponto a ser destacado é que a sigmoide também pode ser expressa de uma forma recurssiva, algo que é muito útil pois ajuda a poupar cálculos a serem feitos. Dessa forma, também tem-se a Equação 7.4.

Derivada da sigmoide logística recursivamente

$$\frac{d}{dz_i}\sigma(z_i) = \sigma(z_i)(1-\sigma(z_i)) \tag{7.4}$$

Tendo a sua derivada, cabe também plotar o seu gráfico para ver o seu comportamento. Com isso, a derivada da sigmoide logística pode ser vista na Figura 7.6. Perceba que o valor máximo que a derivada da sigmoide retorna é pouco maior que 0.2, esse é um dos pontos que mais atrapalha as redes que fazem uso de muitas funções sigmoidais, porque acaba por gerar um problema conhecido por desaparacimento do gradiente. Mas é importante destacar o quão importante são as derivadas das funções de ativação, pois muitas vezes elas acabam por gerar diversos problemas.

Conhecendo a sigmoide logística, a função que dá nome a essa família de funções de ativação, é possível conhecer agora outras funções dessa mesma família. Essas

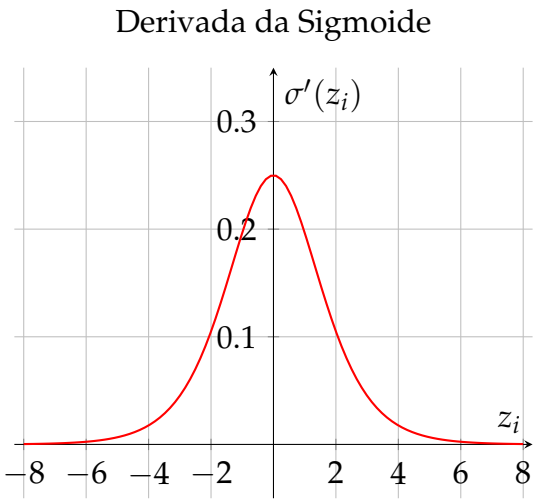


Figura 7.6 – Gráfico da derivada da função de ativação sigmoide logística.

Fonte: O autor (2025).

funções apresentam propriedades semelhantes com a sigmoide, como a caracterítisca curva em "S"mas com algumas variações em sua estrutura ou fórmula, as quais podem ser úteis em alguns cenários.

Para isso, será vista primeira a tangente hiperbólica, e como ela melhora uma das desvantagens da sigmoide, com o fato dessa nova função ser centrada em zero.

7.5 Tangente Hiperbólica: A Pioneira nas Redes Convolucionais

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 (LAMBERT, 2004).

Passado 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 (LECUN *et al.*, 1998). No artigo acadêmico *Gradient-based learning applied to document recognition* de 1998, os cientistas LeCun *et al.* (1998) explicam a criação dessa rede além de destacar suas métricas alcançadas.

É possível escrever a tangente hiperbólica utilizando a definição de tangente, que é o quociente a função seno com a função cosseno, só que neste caso usando as funções hiperbólicas. Assim ela é representada pela Equação 7.5.

Tangente Hiperbólica (tanh)

$$\tanh(z_i) = \frac{\sinh(z_i)}{\cosh(z_i)} = \frac{e_i^z - e^{-z_i}}{e_i^z + e^{-z_i}} \tag{7.5}$$

Semelhante a sigmoide, a tangente hiperbólica possui várias propriedades parecidas. Como afirma Lederer (2021), a tangente hiperbólica é infinitamente diferenciável, sendo uma versão escalada e rotacionada da sigmoide logística. Assim, veja no gráfico da Figura 7.7 que 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.

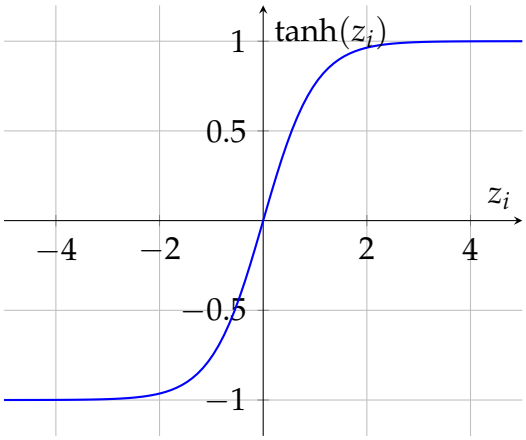


Figura 7.7 – Gráfico da função de ativação tangente hiperbólica (tanh).

Fonte: O autor (2025).

Como foi visto em seções anteriores, é interessante ao escolher uma função de ativação verificar se ela é centrada em zero, como no caso da tangente hiperbólica. Isso acontece pois funções desse tipo permitem uma convergência mais rápida do modelo para os pontos de mínimo, garantindo um melhor desempenho com menos iterações, quando comparadas com funções que não são centradas em zero, como a sigmoide.

De forma análoga a feita na sigmoide logística, vale a pena calcular a derivada da tangente hiperbólica para entender como o gradiente é propagado para as camadas anteriores ao utilizar essa função de ativação. Dessa forma, a sua derivada está representada na Equação 7.6.

Derivada da tangente hiperbólica (tanh)

$$\frac{d}{dz_i}\tanh(z_i) = \operatorname{sech}^2(z_i) \tag{7.6}$$

Assim como a sigmoide logística, a tangente hiperbólica possui uma vantagem, ela pode ser escrita de forma recursiva, o que ajuda a poupar cálculos para a o *backward pass*, pois é possível simplemente armazenar o resultado dessa função que estava calculado no *forward pass* e reutilizá-lo ao calcular a sua derivada. Isso é muito útil pois

quanto menos calculos forem feitos, maior é a tendência que o modelo será mais rápido e com isso irá convergir em menos tempo [1]. Dessa forma, tem-se que a derivada da tangente hiperbólica escrita de forma recursiva pode ser expressa pela Equação 7.7

Derivada da tangente hiperbólica (tanh) recursivamente

$$\frac{d}{dz_i}\tanh(z_i) = 1 - \tanh^2(z_i) \tag{7.7}$$

Tendo as equações da derivada da tangente hiperbólica, o próximo passo é plotar também o seu gráfico, o qual está presenta na Figura 7.8. Note mais uma vez que a tangente hiperbólica possui o mesmo problema que a sigmoide, os valores máximos que ela retorna para a derivada são muito pequenos, esse é um fator recorrente nas funções sigmoidais.

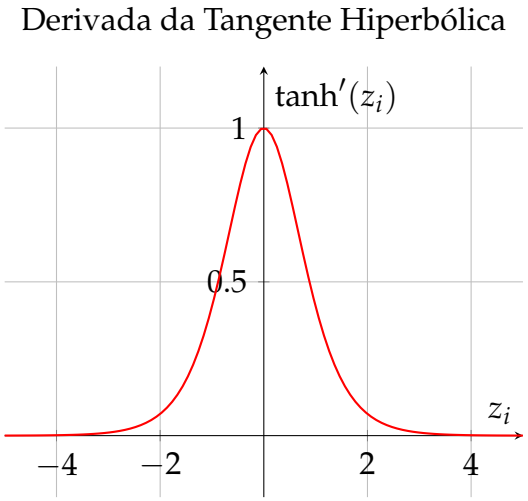


Figura 7.8 – Gráfico da derivada da função de ativação tangente hiperbólica (tanh).

Fonte: O autor (2025).

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. Mais uma vez, nota-se que a tangente hiperbólica não atinge valores muito altos em sua derivada, isso acaba por gerar um problema conhecido como desaparacimento do gradiente, o qual será explicado em seções futuras.

Por fim, perceba que a tanto a tangente hiperbólica quando a sigmoide logística são funções de ativação "caras", por utilizarem exponenciais em sua composição. Assim, a próxima função a ser vista busca justamente corrigir esse problema, possuindo a

[1] Note que tanto a sigmoide quanto a tangente hiperbólica fazem uso de exponenciais em suas fórmulas, essas operações acabam por ser caras (em termos de poder de processamento) quando comparadas com as simples operações de comparação da hard tanh, a qual será vista em seções futuras.

mesma proposta das sigmoidais, porém, sendo mais "barata"em termos de custo computacional.

7.6 Softsign: Uma Sigmoidal Mais Barata

A próxima função sigmoidal a ser analisada é a *softsign*, diferente da tangente hiperbólica e da sigmoide que tiveram suas origens em outros campos diferentes da ciência da computação, a *softsign* foi criada com o intuito de ser trabalhada em uma rede neural. Ela foi introduzida no artigo *A Better Activation Function for Artificial Neural Networks* de 1993, do cientista Foundation e Elliott (1998), no texto o autor propõe a *softsign* como uma alternativa para as funções sigmoidais tradicionais.

A principal diferença da *softsign* com as outras sigmoidais está na sua fórmula, como pode ser visto na Equação 7.8 é que ela não utiliza nenhum exponencial para compor sua função. Isso faz com que ela seja uma função mais "barata"em termos de custo computacional para ser implementada em redes neurais. Assim, é possível obter resultados parecidos porem utilizando cálculos menos complexos e com isso encontrar redes mais rápidas de serem treinadas.

Softsign

$$\text{softsign}(z_i) = \frac{z_i}{1 + |z_i|} \tag{7.8}$$

A *softsign* também possui uma representação gráfica, como vista na Figura 7.9, ela possui o formato em "S"característico das sigmoidais além de ser centrada em zero como a tangente hiperbólica. Além disso, como Foundation e Elliott (1998) destaca em seu texto, ela é uma função que é diferenciável em toda a reta possuindo também a mesma propriedade das outras sigmoidais de empurrar os valores de entrada para os seus extremos. Nota-se também pelo seu gráfico que ela também uma função contínua, suave e não-linear.

Com relação a sua diferenciabilidade, a *softsign* pode ser derivada em todos os seus pontos, e sua derivada pode ser vista na Equação 7.9. Com base nela, é possível notar 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 é aproveitado o resultado, na *softsign* isso não ocorre.

Derivada da *softsign*

$$\frac{d}{dz_i}\text{softsign}(z_i) = \frac{1}{(1 + |z_i|)^2} \tag{7.9}$$

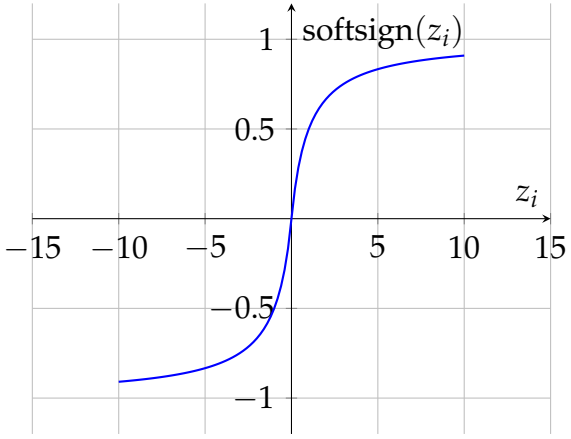


Figura 7.9 – Gráfico da função de ativação *softsign*.

Fonte: O autor (2025).

Tendo a sua derivada, cabe também plotar o gráfico da derivada da *softsign*, apresentado na Figura 7.10. Perceba 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. Com isso, nota-se que ela demora mais para saturar quando comparada com as outras sigmoidais vistas até o momento.

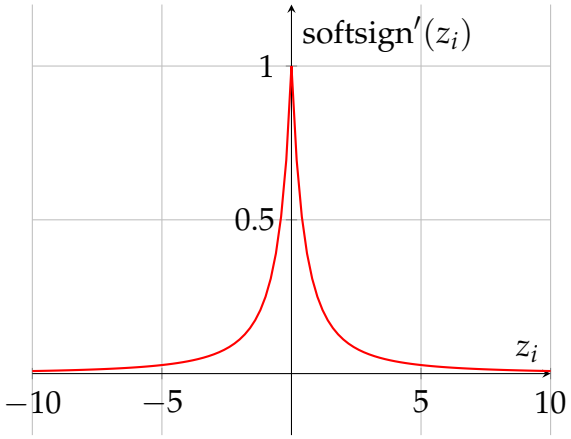


Figura 7.10 – Gráfico da derivada da função de ativação *softsign*.

Fonte: O autor (2025).

Além disso, essa função também sofre do mesmo problema das outras duas funções de ativação vistas até agora, o problema do desaparecimento do gradiente. Esse é um ponto que é corrigido com uma outra família de funções de ativação, as retificadoras, as quais são tema principal do Capítulo 8.

Seguindo na proposta de ver funções sigmoidais que são mais "baratas", a próxima seção busca explorar isso com outras duas funções: a *hard sigmoid* e a *hard tanh*. Essas funções buscam imitar o comportamento das sigmoidais porém, escrito em forma de retas, dessa forma, elas sacrificam a suavidade das sigmoidais para garantir um menor custo computacional.

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

Agora será visto duas funções sigmoidais, 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 é a *hard sigmoid*. Como pode ser visto na Equação 7.10, a qual está presente na documentação do PyTorch (2025a), 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 quando comparados com a sigmoide tradicional, não tem nenhuma exponencial para atrasar as respostas da função. Mas note que também existe um crescimento linear quando os valores estão entre -3 e 3.

Hard Sigmoid

$$\text{hard sigmoid}(z_i) = \begin{cases} 0 & \text{se } z_i < -3 \\ z_i/6 + 0.5 & \text{se } -3 \leq z_i \leq 3 \\ 1 & \text{se } z_i > 3 \end{cases} \tag{7.10}$$

Além disso, sabendo a sua fórmula, é possível também plotar o seu gráfico, o qual está representado na Figura 7.11. Ele lembra uma função sigmoide logística, porém, sem todas as curvas suaves daquela função, apresentando no lugar um conjunto de três diferentes retas, a primeira constante em zero, a segunda a função identidade e a terceira também constante em um.

Com relação a derivada da *hard sigmoid*, para obtê-lá, deve-se derivar todas as três expressões construindo a sua derivada. Note que os pontos que as retas se tocam no gráfico da Figura 7.11 são pontos de descontinuidade, pois quando calculados os limites laterais nesses pontos eles serão diferentes. Isso faz com que essa função não possa ser derivada desses locais. Considerando isso, é possível expressar a derivada da *hard sigmoid* com a Equação 7.11.

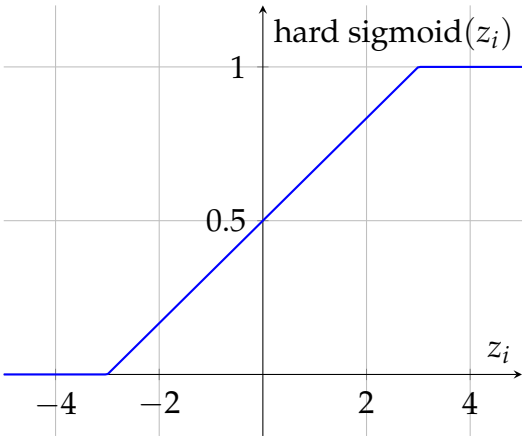


Figura 7.11 – Gráfico da função de ativação *hard sigmoid*.

Fonte: O autor (2025).

Derivada da Hard Sigmoid

$$\frac{d}{dz_i}\text{hard sigmoid}(z_i) = \begin{cases} 0 & \text{se } z_i < -3 \\ 1/6 & \text{se } -3 < z_i < 3 \\ 0 & \text{se } z_i > 3 \end{cases} \tag{7.11}$$

Tendo a sua derivada, cabe também plotar o seu gráfico, o qual está presente na figura 7.12. Perceba mais uma vez que o gráfico não é suave que nem na sigmoide logística, perceba que ele também tenta imitar o comportamento do gráfico da derivada da sigmoide logística, neste caso, com retas ao invés de curvas. Mas ainda sim, essa derivada retorna valores muito pequenos, ficando susceptível ao problema do desaparecimento do gradiente assim como a sua variante *soft*.

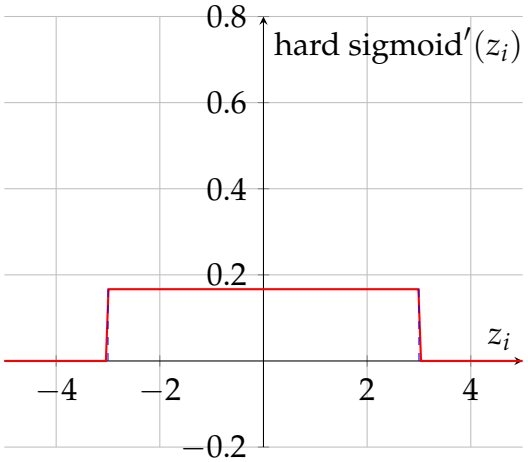


Figura 7.12 – Gráfico da derivada da função de ativação *hard sigmoid*.

Fonte: O autor (2025).

Além disso, também existe a função *hard tanh*, que apresenta a mesma proposta da

hard sigmoid, mas, nesse sentido, busca imitar o comportamento da função de ativação tangente hiperbólica. Para isso, ela segue uma ideia parecida com a da equação da *hard sigmoid*, escrevendo a sua equação em forma de condicionais, mas ajustando os seus valores para corresponder ao funcionamento da tanh. Essa função está expressa na Equação 7.12.

Hard Tanh

$$\text{hard tanh}(z_i) = \begin{cases} -1 & \text{se } z_i < -1 \\ z_i & \text{se } -1 \leq z_i \leq 1 \\ 1 & \text{se } z_i > 1 \end{cases} \tag{7.12}$$

O seu gráfico também é parecido com o da *hard sigmoid*. Como é possível vez na Figura 7.13, ele é composto por três retas, sendo duas delas constantes e uma terceira sendo a função identidade. Note que ele também possui "bicos", o que o impede essa função de ser derivada nesses pontos, uma vez que ela não é contínua nessas regiões.

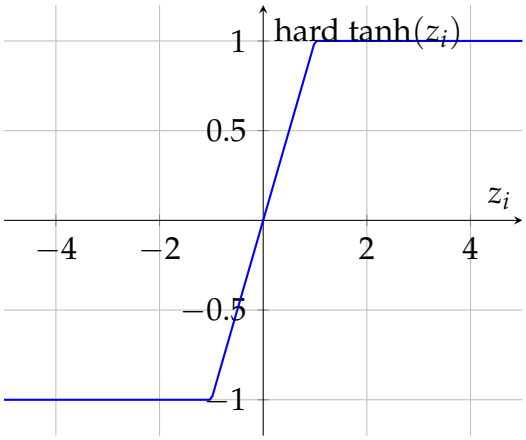


Figura 7.13 – Gráfico da função de ativação *hard tanh*.

Fonte: O autor (2025).

Perceba também que o gráfico, bem como sua equação, é bem mais simples que a tangente hiperbólica tradicional, isso faz com que a *hard tanh* seja uma função mais barata para ser empregada ao desenvolver um modelo de rede neural. Isso signica que ela pode ser uma função ideal para ser utilizada naqueles sistemas que possuem recursos limitados, como uma quantidade pequena de memória ram, ou seu a possibilidade de utilizar uma placa gráfica dedicada para o processamento dos dados. O mesmo vale para a *hard sigmoid*.

Agora é possível também considerar também a derivada da *hard tanh*, a qual é calculada de forma semelhante a da *hard sigmoid*, derivando as três expressões de sua equação separadamente. Feito isso, a derivada da *hard tanh* está expressa na Equação 7.13.

Derivada da Hard Tanh

$$\frac{d}{dz_i}\text{hard tanh}(z_i) = \begin{cases} 0 & \text{se } z_i < -1 \\ 1 & \text{se } -1 < z_i < 1 \\ 0 & \text{se } z_i > 1 \end{cases} \tag{7.13}$$

Tendo a derivada, o próximo passo é plotar o seu gráfico, que pode ser visto na Figura 7.14. Note mais uma vez que ele também tenta imitar o comportamento da derivada da tangente hiperbólica, porém, sendo composto por retas ao invés de uma curva suave. Novamente, é possível perceber que a *hard tanh* também irá sofrer do problema do desaparecimento do gradiente, uma vez que retorna valores muito pequenos para a sua derivada.

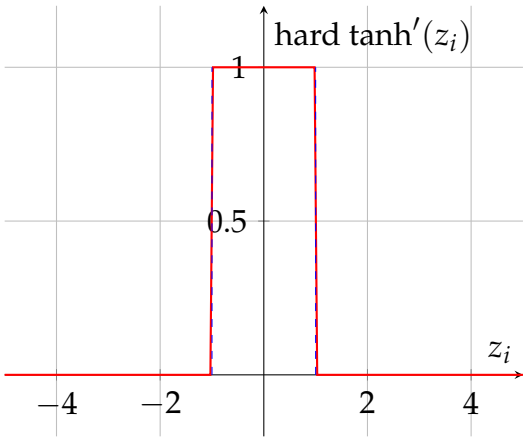


Figura 7.14 – Gráfico da derivada da função de ativação *hard-tanh*.

Fonte: O autor (2025).

Conhecendo todas essas diferentes funções de ativação, é possível finalmente entender o que é o problema do desaparecimento de gradientes e como ele ocorre em uma rede neural, além de relacionar a atuação das funções sigmoidais com o aparecimento desse fenônemo. Para isso, é possível ver essas explicações na seção a seguir.

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, é possível notar 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.

Para explicar melhor essa condição será utilizado um problema como base.

Como foi visto no capítulo anterior, o gradiente retropropagado para camadas anteriores de uma rede neural é proporcional a multiplicação da perda, com a derivada da função de ativação no ponto e o valor do resultado da camada anterior de neurônios.

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

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.

Considerando isso, imagine que temos uma rede composta por quatro camadas densas e cada camada tem apenas um neurônio com pesos iguais a 1. Dessa forma, é possível simplificar a fórmula vista para a Equação 8.20.

$$\delta^{(L)} = \delta^{(L+1)} \times \sigma'(x^{(L)}) \tag{7.14}$$

Considere também que as camadas da rede possuem a seguinte configuração:

- Sigmoide da primeira camada: tem como resultado da derivada $\sigma'(z_i) = 0.2$
- Sigmoide da camada densa 2: tem como resultado da derivada $\sigma'(z_i) = 0.05$
- Sigmoide da camada densa 3: tem como resultado da derivada $\sigma'(z_i) = 0.1$

- Sigmoide da camada de saída: tem como resultado da derivada $\sigma'(z_i) = 0.08$

Além disso, você sabe também que o gradiente inicial na camada de saída está sendo de 1. Com isso é possível calcular o gradiente retropropagado para a primeira camada, comecando pela terceira, já que já temos o valor do gradiente para a camada de saída, dessa forma temos que:

$$\left.\begin{aligned}\delta^{(3)} &= \delta^{(4)} \times \sigma'(x^{(3)}) \\ \delta^{(3)} &= 1 \times 0.1 = 0.1\end{aligned}\right\rangle \textit{Subtituindo os valores}$$

Seguindo adiante, é possível fazer o mesmo procedimento para encontrar $\delta^{(2)}$ agora já tendo $\delta^{(3)}$, dessa forma:

$$\left.\begin{aligned}\delta^{(2)} &= \delta^{(3)} \times \sigma'(x^{(2)}) \\ \delta^{(2)} &= 0.1 \times 0.05 = 0.005\end{aligned}\right\rangle \textit{Subtituindo os valores}$$

De forma semelhante, é finalmente possível encontrar o gradiente retropropagado para a primeira camada:

$$\left.\begin{aligned}\delta^{(1)} &= \delta^{(2)} \times \sigma'(x^{(1)}) \\ \delta^{(1)} &= 0.005 \times 0.2 = 0.001\end{aligned}\right\rangle \textit{Subtituindo os valores}$$

Note que o gradiente que entrou ao ser calculado pela perda era de 1, no final da camada chegou apenas 0.001, ou seja, ele diminui mil vezes. Se considerarmos uma situação como esta, 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 retropropagado naquela camada seja baixo. 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 corretamente. 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. Assim, tem-se o problema do desaparecimento do gradiente.

> **Definição:** Quando 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, fazendo com que não haja progresso nos parâmetros de aprendizado, dessa forma o **desaparecimento do gradiente** é quando esse fenômeno acontece repetidamente por várias pares de entrada e saída. (LEDERER, 2021).

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.

Nesse sentido, o Capítulo 8 busca explicar as funções retificadoras, começando pela *ReLU*, e como elas vieram como uma alternativa para contornar o problema do desaparecimento do gradiente. Contudo, elas também não foram perfeitas, e acabaram por introduzir os seus próprios problemas ao serem aplicadas em uma rede neural. Neste caso, um dos problemas que pode ocorrer ao se utilizar uma função retificadora é o problema da explosão do gradiente, e considerando a *ReLU*, ela também pode causar um problema conhecido como *ReLUs* agonizantes.

Assim, o último passo ao conhecer todas essas funções, é resumir o seu conteúdo para melhor entendimento. Este resumo pode ser visto na próxima seção.

7.9 Comparativo: Funções Sigmoidais

Tendo visto diferentes funções de ativação sigmoidais, é possível compilá-las em na Tabela 7.1, destacando as suas fórmulas, vantagens e desvantagens ao serem utilizadas em uma rede neural.

Tabela 7.1 – Comparativo das funções de ativação sigmoidais

Função	**Vantagem**	**Desvantagem**
Sigmoide logística	Pode ser interpretada como uma probabilidade, podendo ser aplicada nas camadas de saída para classificação binária.	Não é centrada em zero, fazendo com que a convergência de modelos que usem essa função seja um pouco mais lenta.
Tangente hiperbólica	Centrada em zero, garantindo uma convergência mais rápida de modelos que usem essa função.	Possui muitos exponenciais, sendo uma função "cara"em termos de custo computacional.
Softsign	É uma sigmoidal mais "barata"em termos de custo computacional, permitindo a criação de redes mais rápidas.	Sua derivada não pode ser escrita de forma recursiva.
Hard Sigmoid	É uma versão mais "barata"da sigmoide logística.	Não é uma função suave, possuindo "quinas", as quais impedem essa função de ser derivada nesses pontos.
Hard Tanh	É uma versão mais "barata"da tangente hiperbólica.	Assim como a *hard sigmoid*, a *hard tanh* apresenta "quinas", as quais impedem essa função de ser derivada nesses pontos.

Fonte: O autor (2025).

8 Funções de Ativação Retificadoras

"caramba! A perda do meu modelo está em 31.415"
— Estagiário conhecendo o problema dos gradientes explosivos

No Capítulo 7 foi apresentado as funções de ativação sigmoidais. Essas funções estiveram presentes em um grande quantidade de redes neurais criadas até os anos 2010, sendo consideradas padrões para se utilizarem ao construir uma RNA. Contudo, essas funções são susceptíveis ao problema do desaparecimento do gradiente, um problema que afetou consideravelmente como as redes neurais eram construídas ao utilizar esse tipo de função de ativação, pois não era possível construir redes muito profundas.

Nesse cenário surgem as funções retificadoras, como uma solução para contornar esse problema. Essas funções são o tópico central desse capítulo. Para isso, primeiro será vista a *rectfied linear unit*, também conhecida como *ReLU*, conhecendo as suas origens, propriedades, fórmulas e como ela permitiu a criação de redes neurais mais profundas. Em seguida, será visto um problema crônico dessa função de ativação: os *ReLUs* agonizantes.

Para contornar esse problema, surgem então variantes da *ReLU* com vazamento, permitindo que um pouco de gradiente flua pela rede para os casos em que a entrada dessa função seja negativa. Serão vistas nessa seção três funções: a *Leaky ReLU*, a *Parametric ReLU*, e a *Randomized Leaky ReLU*. Seguindo adiante serão vistas também variantes que apresentam curvas suaves em seus gráficos, como a *ELU* e a *SELU*. Para todas essas funções serão apresentados diversos comparativos para entender melhor o seu desempenho e em quais cenários elas são ideais.

Essas funções também não são perfeitas, assim como a sigmoidais, elas também acabaram por introduzir uma nova classe de problemas. Neste caso, um problema que pode ocorrer ao se utilizar uma função retificadora é o do chamado gradiente explosivo, que, assim como o desaparecimento do gradiente, impede o aprendizado dos neurônios da rede. O capítulo termina com uma tabela, compilando as principais características dessa família de funções de ativação.

8.1 Exemplo Ilustrativo: Vendendo Pipoca

Imagine que você está querendo ganhar dinheiro e decidiu vender pipoca em uma praça da sua cidade. Você comprou milho, óleo, sal e manteiga, um carrinho para poder levar e fazer as pipocas, além disso, você também comprou vários pacotes para poder colocar as pipocas para vender.

Nisso, você teve que estipular um valor para vender essas pipocas, após pensar um pouco e analisar todos os seus gastos, você estimou que um valor de R$ 5,00 seria ideal, pois conseguiria pagar os seus gastos mas você ainda ia obter lucro dos seus clientes.

Agora você está pronto para vender, começou a fritar o milho e colocou uma plaquinha com o preço ao lado do seu carrinho. Então chega uma pessoa com R$ 6,00 e decide comprar um pacote, você vende e entrega um real de troco. Logo em seguida aparece uma segunda pessoa com R$ 4,99 e decide negociar com você, ela afirma que é quase R$ 5,00, e por isso, você deveria vender a pipoca para ela, mas você explica que só vende pelo valor de R$ 5,00.

Com base nisso, nós podemos chegar em uma situação em que um pacote de pipoca será vendido somente se uma pessoa possuir R$ 5,00 no bolso, ou mais. Podemos então escrever algo como o da equação 8.1. Em casos em que uma venda ocorre, você poderá vender mais um pacote, para isso, o seu comprador deverá possuir pelo menos R$ 10,00, assim, x que indica a quantidade de pacotes vendido seguirá a lei de formação $x = 5 \mod d$, em que d é o dinheiro que a pessoa possui.

$$\text{Número de Pacotes} = \begin{cases} 0 & \text{quando } R\$ \leq 4,99 \\ x & \text{quando } R\$ > 4,99 \end{cases} \tag{8.1}$$

Saindo do assunto da pipoca e voltando para o tema deste texto, existe uma família de funções de ativação que funciona de forma semelhante a lógica de venda dos pacotes de pipoca, elas são as unidades lineares retificadoras. A ReLU, que dá nome a essa família, funciona de forma semelhante a essa venda, ela tem um comportamento de "tudo ou nada", em que irá comandar quando um neurônio de uma rede neural irá disparar seu resultado.

8.2 Rectified Linear Unit (ReLU): A Revolução Retificadora

Como foi visto anteriormente no Capítulo 7, as funções sigmoidais surgiram com inspiração nos neurônios humanos e como eles se comportam com determinados estímulos. Mas essas não foram as únicas funções que tiveram essa origem. Na década de 40, o pesquisador Alton Householder estava estudando um cenário parecido em seu trabalho *A theory of steady-state activity in nerve fiber network: I. Definition of mathematical biofysics*, nele o autor analisou o comportamento de fibras nervosas e quando elas irão assumir caráter excitatório ou inibitório, para isso ele apresentou a Equação 8.2 (HOUSEHOLDER, 1941).

$$a_{ij} = \begin{cases} 0 & \text{quando } \eta_i \leq h_{ij} \\ a_{ij}, & \text{quando } \eta_i > h_{ij} \end{cases} \tag{8.2}$$

Essa equação mostra quando uma fibra nervosa irá disparar, para isso, deve-se olhar o limiar da fibra h_{ij} e o estímulo total η_i, com base nesses valores e no que a fórmula apresenta, uma fibra irá disparar quando o estímulo total for maior que o seu limiar, quando isso não ocorrer, ela não irá disparar (HOUSEHOLDER, 1941). Além disso, Householder (1941) explica também sobre o termo a_{ij}, a saída dessa função, segundo o autor ele é utilizado para representar o parâmetro de atividade, sendo um valor diferente de zero, podendo ser positivo (quando a fibra possui ação excitatória), ou negativo (apresentando caráter inibitório).

Essa equação criada por Householder, lembra bastante a expressão da função *ReLU*, a qual é denotada pelas Equações 8.3.

Função de Ativação *Rectified Linear Unit* (*ReLU*)

$$\text{ReLU}(z_i) = \begin{cases} z_i, & \text{se } z_i > 0 \\ 0, & \text{se } z_i \leq 0 \end{cases} \text{ouReLU}(z_i) = \max(0, z_i) \tag{8.3}$$

Dito isso, mesmo com ela existindo a mais de 80 anos, ela só passou a ser amplamente utilizada nos anos 2010, antes disso, as sigmoides eram a grande maioria quando o assunto era função de ativação. Contudo, as sigmoides eram funções saturantes, e isso fazia com que sua derivada retornasse muitos valores pequenos ao longo da função. Ao multiplicar vários valores pequenos na retropropagação do gradiente, o vetor gradiente ia diminuindo até chegar um ponto em que ele não conseguia atualizar os pesos e vieses das redes neurais de forma eficiente, assim, tínhamos o problema do desaparecimento do gradiente. As funções retificadoras, sendo a principal delas a *ReLU*, surgem para corrigir esse problema crônico.

Dessa forma, antes de conhecer de fato a *ReLU* e suas propriedades, é preciso entender o cenário que ela se popularizou, com os cientistas buscando novos tipos de funções de ativação que substituísse as sigmoides, funções saturantes, por outro tipo de função que resolvesse o problema do desaparecimento do gradiente.

Nesse cenário, artigos como *Rectified Linear Units Improve Restricted Boltzmann Machines* foram essenciais para popularizar a *ReLU* como uma função de ativação interessante para se utilizar em redes neurais. No trabalho, Nair e Hinton (2010) foram responsáveis por demonstrar propriedades úteis das funções retificadoras, como a capacidade da *NReLU* de auxiliar em reconhecimentos de objetos por possuir equivariância de intensidade(*intensity equivarience*), o que significa que se a intensidade da entrada de uma função for alterada por um determinado fator a intensidade de sua saída será alterada pelo mesmo fator. Essa propriedade se torna bastante útil em casos que queremos preservar informações, como ao comparar imagens, garantindo melhor precisão por exemplo em situações de baixa luz quando comparados com cenários em

que possuem muita luz nas imagens.

Além disso, no trabalho *Deep Sparse Rectifier Neural Networks* dos autores Xavier Glorot e Bengio (2011), o uso de unidades retificadoras não lineares são propostos como alternativas para a tangente hiperbólica e sigmoide em redes neurais profundas, mas também os pesquisadores são capazes de demonstrar que as unidades retificadoras se aproximam melhor do comportamento de neurônios biológicos. Um ponto chave desse texto é que os autores destacam características importantes que a esparsidade traz para uma rede neural possibilitada pelo uso de funções retificadoras (XAVIER GLOROT; BENGIO, 2011). Entre elas estão:

- **Desembaraçamento de Informações:** Um dos principais objetivos dos algoritmos de aprendizado profundo é desembaraçar os fatores que explicam as variações nos dados, assim, existem diferentes tipos de representações, uma representação densa é altamente emaranhada porque quase qualquer mudança na entrada modifica a maior parte as entradas no vetor de representação, contudo, se tivermos uma representação esparsa e robusta a pequenas mudanças na entrada, o conjunto de características diferentes de zero é quase sempre aproximadamente conservado por pequenas mudanças na entrada (XAVIER GLOROT; BENGIO, 2011);

- **Representação eficiente de tamanho variável**. Diferentes entradas podem conter diferentes quantidades de informação e seriam mais convenientemente representadas usando uma estrutura de dados de tamanho variável, o que é comum em representações computacionais de informação, assim é interessante poder variar o número de neurônios ativos permitindo que um modelo controle a dimensionalidade efetiva da representação para uma determinada entrada e a precisão necessária (XAVIER GLOROT; BENGIO, 2011);

- **Separabilidade linear**. Representações esparsas também são mais propensas a serem linearmente separáveis, ou mais facilmente separáveis com menos maquinário não linear, simplesmente porque a informação é representada em um espaço de alta dimensão, além disso, isso pode refletir o formato original dos dados (XAVIER GLOROT; BENGIO, 2011);

- **Distribuídas, mas esparsas**. Representações densamente distribuídas são as representações mais ricas, sendo potencialmente exponencialmente mais eficientes do que as puramente locais, além disso a eficiência das representações esparsas ainda é exponencialmente maior, com a potência do expoente sendo o número de características diferentes de zero, elas podem representar uma boa compensação em relação aos critérios acima (XAVIER GLOROT; BENGIO, 2011).

Por fim, um último trabalho que colaborou para a popularização da *ReLU* foi a *AlexNet*, de Krizhevsky, Sutskever e Hinton (2012), essa rede neural convolucional (*CNN*) foi capaz ganhar o Desafio de Reconhecimento Visual em Larga Escala *ImageNet* (*ILSVRC*) sendo treinada para classificar 1.2 milhões de imagens de alta resolução e classificá-las em 1000 diferentes classes. Para isso, essa *CNN* foi construída utilizando 8 camadas com pesos, sendo as primeiras 5 camadas convolucionais, enquanto as três últimas são camadas totalmente conectadas, a última camada de neurônios faz uso da função de ativação *softmax* para fazer a distribuição em 1000 diferentes classes, por último mas não menos importante, a *AlexNet* fez uso da ReLU em sua arquitetura (KRIZHEVSKY; SUTSKEVER; HINTON, 2012).

Assim, como pode ser visto na tabela 8.1, a *AlexNet* foi capaz de alcançar uma taxa de erro de 15.3% na fase de testes, note com base na variação de camadas convolucionais, que essa é uma rede que se beneficia da sua profundidade, algo que provavelmente só foi capaz de ocorrer devido ao uso da *ReLU* como função de ativação, por não gerar o problema do desaparecimento de gradientes como nas sigmoidais. Além disso, a rede *SIFT + FVs* (*Scale-Invariant Feature Transform + Fisher Vectors*) é mostrada na tabela como base de comparativo, perceba que a *AlexNet* foi capaz de diminuir com mais de 10% dos erros que essa rede gerava.

Além disso, o modelo *SIFT + FVs* (*Scale-Invariant Feature Transform + Fisher Vectors*,) o qual é apresentado como base de comparação, apresenta uma taxa de erro de 26.2%, um aumento de 10 pontos percentuais quando comparado com o melhor modelo da *AlexNet* de 15.3%.

Tabela 8.1 – Comparação das Taxas de Erro no *AlexNet*

Modelo	**Top-1 (validação)**	**Top-5 (validação)**	**Top-5 (teste)**
SIFT + FVs	-	-	26.2%
1 CNN	40.7%	18.2%	-
5 CNNs	38.1%	16.4%	16.4%
1 CNN[a]	39.0%	16.6%	-
7 CNNs[a]	36.7%	15.4%	**15.3%**

Nota: A tabela compara as taxas de erro de diferentes modelos nos conjuntos de validação e teste do *ILSVRC-2012*. Os valores em negrito indicam o melhor resultado. [a]Modelos que foram pré-treinados para classificar todo o conjunto de dados ImageNet 2011 Fall.
Fonte: Adaptado de "ImageNet Classification with Deep Convolutional Neural Networks", por A. Krizhevsky, I. Sutskever, & G. E. Hinton, 2012, *Advances in Neural Information Processing Systems, 25*.

A definição da *ReLU* pode ser interpretada como uma pergunta, ao receber um número como entrada a *ReLU* questiona: "esse número é menor que zero?", se a resposta for sim, ela retorna como resultado o número zero, se a resposta for não, ela irá retornar o próprio número como sua saída. Neste caso estão sendo considerados números, mas

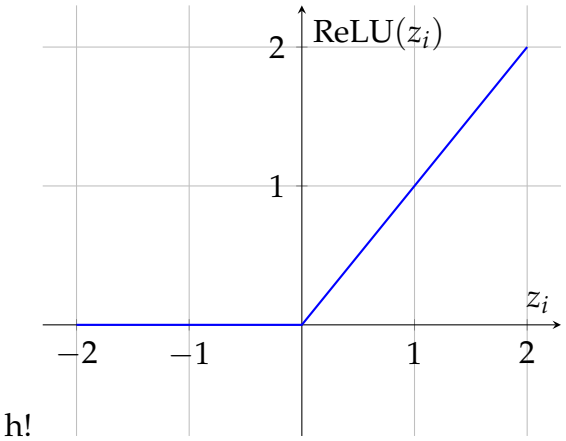


Figura 8.1 – Gráfico da função de ativação *Rectified Linear Unit* (*ReLU*).

Fonte: O autor (2025).

a analogia utilizada no início do texto em que o pacote de pipoca só é vendido caso a pessoa tenha mais de R$ 5,00 também pode ser utilizada, em que o resultado seria um valor booleano, indicando se a pessoa vende ou não a pipoca.

Além de sua fórmula, é possível plotar o seu gráfico, que está presente na Figura 8.1, ele é bem mais simples quando comparado com a sigmoide, por exemplo, sendo apenas a junção de duas retas, uma delas uma função constante que irá retornar sempre zero e a outra a função identidade. Essa simplicidade da *ReLU* é algo muito atrativo para os desenvolvedores, pois, ao utilizá-la ao invés de uma função mais complexa como a sigmoide ou a tangente hiperbólica, estamos diminuindo a complexidade da rede neural, se essa rede se torna mais simples, a tendência é de que ela possua um custo de poder de processamento menor permitindo que um volume maior de dados seja processado em menos tempo e com isso seu tempo de treinamento seja menor. Note que, antes da *ReLU* surgir, muitos das funções de ativação faziam uso de exponenciais, a *ReLU* não só resolvia o problema do desaparecimento do gradiente mas também era muito mais "barata".

ao trabalhar com redes neurais, uma das maiores vantagens destas é o fato de “aprenderem” com base na retropropagação do gradiente nas camadas da rede. Assim, ao calcular o gradiente para fazer a retropropagação do erro e ajustar os pesos e vieses das camadas, é necessário ter em mente também a derivada daquela função de ativação que que será aplicada em uma camada de neurônios da rede, dado que ela entrará no *backward pass* do modelo.

Para achar a derivada da *ReLU*, deve-se derivar as duas condicionais dela, assim, quando x for maior que zero, a saída será 1, já quando x for menor que zero, a saída será zero. Mas você vai encontrar um problema nisso, a derivada dessa função não existe quando x é 0, pois o limite lateral à esquerda dessa função é zero, enquanto o limite lateral a direita dela é um. Isso passa a ser um problema quando queremos calcular

o valor de saída justamente quando aquele valor de entrada é zero. Na prática, esse problema é fácil de resolver, ao desenvolver o código dessa função, escolher qual será o resultado da *ReLU* quando esse valor de entrada for zero. Podemos dizer que ele será um ou zero, isso irá depender somente da nossa implementação da derivada da *ReLU*.

Assim, a derivada da *ReLU* é dada pela expressão 8.4.

Derivada da Função de Ativação *Rectified Linear Unit* (*ReLU*)

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

Esse detalhe da descontinuidade da *ReLU* no ponto zero foi algo que acabou mudando em funções futuras, que buscam corrigir erros da *ReLU* e melhorá-la, assim, com o passar do tempo foram surgindo outras alternativas que também trabalhassem com os atributos da ReLU, mas que fossem contínuas em toda a reta, permitindo a sua derivação também em todos os pontos. Uma dessas funções a *ELU*, ela será explicada mais em frente.

Além da sua representação em forma de equação, é possível fazer também o seu gráfico na Figura 8.4, note que ele é ainda mais simples que a própria função de ativação, são só duas retas constantes que irão retornar zero quando o número for menor que zero, ou irão retornar 1 quando a entrada for um número maior que zero.

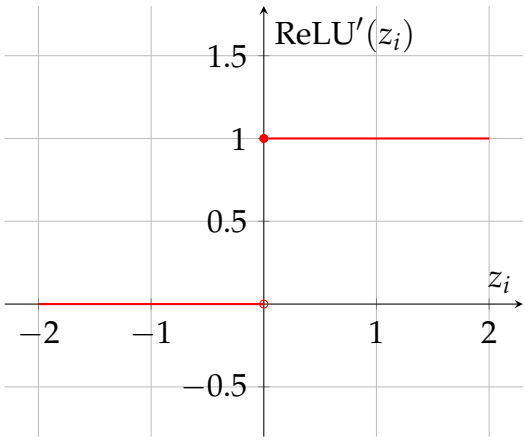


Figura 8.2 – Gráfico da derivada da função de ativação Rectified Linear Unit (ReLU).

Fonte: O autor (2025).

Assim, com toda essa simplicidade e versatilidade, a *ReLU* se tornou uma função que é considerada padrão para a maioria das redes neurais *feedforward* (GOODFELLOW; BENGIO; COURVILLE, 2016). Contudo, ela também apresenta problemas assim como as sigmoidais, sendo um desses problemas o dos *ReLUs* agonizantes, o qual será explicado em sequência.

8.3 O Problema dos ReLUs Agonizantes

Mesmo apresentando tantas propriedades úteis, como o fato de impedir o problema do desaparecimento de gradientes além de ser uma função computacionalmente barata, a *ReLU* não é perfeita. Ela é responsável por causar um problema conhecido como *ReLUs* agonizantes (*Dying ReLUs problem*), que será explicado nessa seção. Para isso, é preciso lembrar primeiro da equação da camada densa:

$$y = W^T X + b$$

Imagine que a *ReLU* é a função de ativação que está sendo utilizada para introduzir a não-linearidade após essa camada, quando uma variável y passar para uma função $\max(0, y)$, e condição mais comum dessa comparação for os casos em que $y < 0$ isso pode afetar negativamente o aprendizado do modelo na etapa do *backward pass*, uma vez que a derivada será utilizada nessa parte, ela também será zero para os casos em que $y < 0$, e como foi visto no capítulo 6, a derivada será utilizada para a multiplicação do gradiente, se a derivada é zero, o gradiente também é zero, e se o gradiente é zero, isso resulta em cenários nos quais os neurônios não vão ter seus pesos atualizados e portanto não irão aprender (DOUGLAS; YU, 2018). Como explica Douglas e Yu (2018), os neurônios irão morrer, passando apenas a retornar zeros independente de sua entrada.

Essa condição de vários neurônios morrendo causada pela *ReLU* acabou por gerar um novo conjunto de funções, as quais possuem propriedades comuns da *ReLU*, como a não linearidade e a simplicidade nos cálculos mas que buscam resolver ou amenizar esse problema em uma rede neural. Uma das funções que busca resolver esse problema é a *Leaky ReLU* (DOUGLAS; YU, 2018).

8.4 As Variantes com Vazamento: Corrigindo o Problema do ReLUs agonizantes

Diferente da *ReLU* tradicional que retorna zero para os casos em que sua entrada é negativa e por isso na sua derivada irá também retornar zero nestes casos, as variantes com vazamento atuam de outra forma, elas retornam um valor muito pequeno como 0.1, multiplicado pela entrada da função quando ela é negativa. Por isso, a sua derivada será algo também 0.1 (ou valores muito pequenos), isso permite um "vazamento"do gradiente em cenários nos quais a entrada do neurônio será negativa.

Como foi visto, que a causa do *ReLUs* agonizantes era justamente isso: muitas situações em que a entrada era negativa, que gerava um gradiente nulo e consequentemente

impedia os neurônios de terem seus pesos e vieses ajustados, e futuramente morrendo, retornando zero independente de qual fosse a sua entrada.

Assim, essas variantes, como a *Leaky ReLU* e a *PReLU* buscam tentar corrigir um amenizar esse problema da *ReLU* mas mantendo algumas de suas principais propriedades, como a não linearidade, a capacidade de ser escrita compondo duas retas permitindo a criação de uma função simples e rápida de ser computada em uma rede neural.

8.4.1 Leaky ReLU (LReLU)

Seguindo adiante, é possível analisar agora a *Leaky ReLU*, ela é uma variante da *ReLU* que foi criada com intuito de corrigir o problema do *ReLUs* agonizantes. Assim como a *ReLU*, que foi explicada com a analogia do vendedor de pipoca, é possível extender essa explicação para essa nova função, antes o limiar para comprar um pacote de pipoca era de R$ 5,00, quem tivesse menos que isso não comprava nada. Mas agora, para garantir que todos possam comprar pipoca, você como vendedor definiu que quando uma pessoa tiver menos que R$ 5,00 ela também será capaz de comprar pipoca, só que neste caso ela comprará um punhado de pipoca que será proporcional ao dinheiro que ela tem multiplicado por uma constante α. Assim, uma pessoa com um valor próximo de R$ 5,00 pode sair com um punhado de pipoca quase igual ao do pacote original se essa constante α for um valor muito proximo de um. Com isso, você como vendedor consegue obter lucro com uma nova clientela além de não perder clientes por não possuírem o valor total do pacote de pipoca. A *Leaky ReLU* traz uma proposta parecida para resolver com o problema do *ReLUs* agonizantes.

Essa função de ativação foi apresentada no artigo *Recfier Nonlinearites Improve Neural Networks Acustic Models*, em que os autores exploram o uso de redes retificadoras profundas como modelos acústicos para a tarefa de reconhecimento de fala conversacional *switchboard* (ANDREW L. MASS; NG, 2013). Além disso, a sua principal diferença, como explicam Andrew L. Mass e Ng (2013), está no fato dela permitir que um pequeno gradiente diferente de zero flua quando a unidade está saturada e não ativa. Esse gradiente diferente de zero que flui quando a unidade está saturada e não ativa são os seus compradores de pipoca que não possuem o valor total mas são capazes de comprar um punhado dela, neste caso a unidade estará não ativa pois o valor de entrada é negativo mas irá retornar um valor diferente de zero, algo que não acontecia na *ReLU*.

Também é possível discutir a expressão matemática da *Leaky ReLU*, a qual é dada pela Equação 8.5, que é bem parecida com a *ReLU*, porém, ela também irá retornar valores negativos quando a sua entrada for um valor negativo, diferente da *ReLU*, que iria retornar como saída zero. A constante α, no texto original é dada por 0.1

fazendo com que os valores negativos sejam pequenos mas ainda sim, diferentes de zero quando passam pela entrada (ANDREW L. MASS; NG, 2013) [1].

Função de Ativação *Leaky ReLU* (*LReLU*)

$$\text{LReLU}(z_i) = \begin{cases} z_i, & \text{se } z_i \geq 0 \\ \alpha \cdot z_i, & \text{se } z_i < 0 \end{cases} \text{ou} \text{LReLU}(z_i) = \max(0, \alpha z_i) \tag{8.5}$$

Em que a constante α representa uma constante pré-definida pelo programador ao desenvolver a rede neural. Por padrão, em bibliotecas como o PyTorch (2025b) essa constante tem valor de 0.1.

Já para a sua representação gráfica, ela está presente na Figura 8.3. Perceba que a *Leaky ReLU* possui características muito semelhantes com a *ReLU*, como o fato dela assumir o comportamento de uma função identidade para para valores positivos em sua entrada, mas, quando é analisado os seus valores negativos é possível ver uma diferença, agora eles são dados por um gráfico de uma função do primeiro grau, diferente da *ReLU* que era uma função constante em zero. Além disso, a *LReLU*, também é uma função assimétrica e não linear, bem como apresenta um ponto de descontinuidade em zero, pois ao traçar os seus limites laterais, eles apresentam valores diferentes, por isso ela não pode ser derivada nesse ponto, assim como a *ReLU* vista anteriormente.

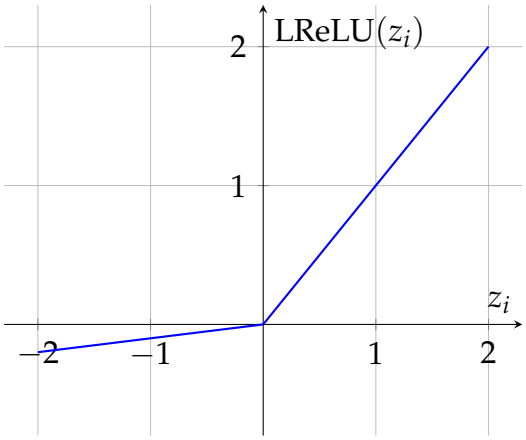


Figura 8.3 – Gráfico da função de ativação *Leaky ReLU* (*LReLU*) com $\alpha = 0.1$.

Fonte: O autor (2025).

Sabendo de sua expressão e seu gráfico, é possível agora calcular sua derivada, para isso, deve-se derivar as duas condicionais que estão na função da *Leaky ReLU*. Assim, quando a entrada dessa função for maior que zero, essa função será x que derivada é 1, já quando a entrada for menor que zero, a função será αx, que quando derivada

[1] Cabe destacar que, essa constante α pode ser ajustada para diferentes cenários, podendo ser valores diferentes de 0.1 como foram propostos no texto original, é possível ver isso acontecendo em comparativos ao longo desse capítulo, em que diferentes autores optam por valores diferentes de α para melhor ajustar ao problema que está sendo analisado.

tem como resultado a própria constante α. Contudo, como dito anteriormente, a derivada da *Leaky ReLU* não existe quando a entrada é exatamente zero, mas na prática, ao trabalhar com a sua definição na retropropagação, é possível definir um valor para a derivada nesse ponto, assim como foi feito com a *ReLU* tradicional. Com isso em mente, tem-se então a Equação 8.6, a qual representa a derivada da *Leaky ReLU*.

Derivada da Função de Ativação *Leaky ReLU* (*LReLU*)

$$\frac{d}{dz_i}[LReLU](z_i) = \begin{cases} 1, & \text{se } z_i > 0 \\ \alpha, & \text{se } z_i \leqslant 0 \end{cases} \tag{8.6}$$

Já que a sua derivada é conhecida, pode-se também plotar o seu gráfico, o qual é dado pela figura 8.4. Ele também é parecido com o gráfico da *ReLU* visto anteriormente, sendo composto por duas retas constantes, para valores positivos ele retorna 1 (assim como a *ReLU*), e para valores negativos ou nulos ele irá sempre retornar a constante α, diferente da *ReLU*, que iria retornar zero, indicando que neste caso o neurônio não está passando nenhuma informação na na retropropagação do gradiente. Por esse motivo que e a *Leaky ReLU* tem esse nome, pois *leaky* em inglês significa "vazamento", e neste caso, como a derivada dela é diferente de zero, mesmo quando a entrada for negativa, ela irá passar informações durante a retropropagação, isso permite que o neurônio não morra, como acontecia em algumas redes que faziam uso da *ReLU* tradicional, e por esse motivo continue aprendendo pelo fato do gradiente continuar fluindo pela rede e consequentemente atualizando os pesos e vieses dos neurônios.

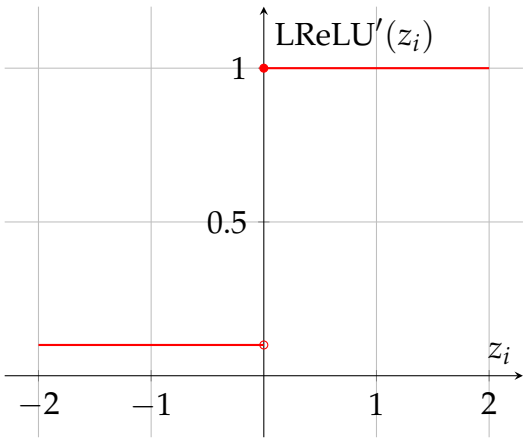


Figura 8.4 – Gráfico da derivada da função de ativação *Leaky ReLU* (*LReLU*) com $\alpha = 0.1$.

Fonte: O autor (2025).

Conhecendo a *leaky ReLU* e suas propriedades, é possível agora entender como essa função se comporta quando comparada com outras em testes.

Em testes de desempenho realizados por Andrew L. Mass e Ng (2013) em seu trabalho, eles foram capazes de analisar como uma rede neural que faz uso dessa função pode performar quando comparada com a *ReLU* tradicional e também com redes que fazem uso da tangente hiperbólica, esse comparativo pode ser visto na Tabela 8.2. Note que as redes neurais que fizeram uso da *Leaky ReLU* como função de ativação obtiveram melhores resultandos quando comparadas com as redes que utilizaram a *ReLU* tradicional ou mesmo a tangente hiperbólica, veja que a rede que foi construída com 3 camadas utilizando a *LReLU* foi capaz de obter a menor taxa de erro de palavra (*WER*) no conjunto *SWBD*, com 17.8%, esse resultado 0.3 pontos percentuais menor quando comparado com uma mesma rede de três camadas que utilizou a *ReLU* tradicional.

Além disso, ainda na tabela 8.2, nas redes compostas por três camadas, a rede que fez uso da *LReLU* também foi melhor que suas outras concorrentes que fizeram uso da *ReLU* e da tanh, sendo capaz de ter a menor taxa de erro de palavra no conjunto de avaliação (*EV*), com 24.3%, uma diferença de 0.1 pontos percentuais quando comparada com a *ReLU* de 24.4%, já quando essa rede é comparada com a tangente hiperbólica, a diferença é ainda maior, sendo de 2.1 pontos percentuais, indicando que a *LReLU* traz resultados melhores quando comparada com essas duas funções de ativação.

Tabela 8.2 – Comparativo de Desempenho de Redes Neurais para Reconhecimento de Fala

Modelo	**Dev CrossEnt**	**Dev Acc (%)**	**SWBD WER**	**CH WER**	**EV WER**
GMM Baseline	N/A	N/A	25.1	40.6	32.6
2 Camadas Tanh	2.09	48.0	21.0	34.3	27.7
2 Camadas ReLU	1.91	51.7	19.1	32.3	25.7
2 Camadas LReLU	1.90	51.8	19.1	32.1	25.6
3 Camadas Tanh	2.02	49.8	20.0	32.7	26.4
3 Camadas ReLU	1.83	53.3	18.1	30.6	24.4
3 Camadas LReLU	1.83	53.4	**17.8**	30.7	**24.3**
4 Camadas Tanh	1.98	49.8	19.5	32.3	25.9
4 Camadas ReLU	1.79	53.9	17.3	29.9	23.6
4 Camadas LReLU	**1.78**	53.9	17.3	29.9	23.7

Nota: Comparação de métricas de erro para sistemas de redes neurais profundas (DNN) em reconhecimento de fala. As métricas de quadro a quadro (frame-wise) foram avaliadas em um conjunto de desenvolvimento, e as taxas de erro de palavra (WER) no conjunto de avaliação Hub5 2000 e seus subconjuntos. Abreviações: Dev CrossEnt = Entropia Cruzada no conjunto de desenvolvimento; Dev Acc = Acurácia no conjunto de desenvolvimento; WER = Taxa de Erro de Palavra (Word Error Rate); SWBD = Switchboard; CH = CallHome; EV = Evaluation set. Valores em negrito indicam os melhores resultados para modelos de 3 e 4 camadas.
Fonte: Adaptado de "Rectifier Nonlinearities Improve Neural Network Acoustic Models", por A. L. Maas, A. Y. Hannun, & A. Y. Ng, 2013, *In Proceedings of the 30th International Conference on Machine Learning, Workshop on Deep Learning for Audio, Speech and Language Processing*.

Agora comparando as redes que fazem uso de quatro camadas, cabe destacar os

resultados da entropia cruzada no conjunto de desenvolvimento (*Dev CrossEnt*), que é uma métrica responsável por medir a diferença entre duas distribuições de probabilidade, neste cenário: a distrubição de probabilidade prevista pelo modelo e a distribuição de probabilidade real, com base nesses dois valores, a entropia cruzada consegue medir o quão bem o modelo de rede neural criado pelos pesquisadores está prevendo a transcrição correta da fala durante a fase de treinamento e ajuste, para isso, é utilizado o conjunto de dados de desenvolvimento (*Dev Set*). Tendo isso em mente, o modelo de quatro camadas que fez uso da *Leaky ReLU* em sua arquitetura obteve o melhor resultado dos seus outros dois concorrentes, sendo assim, ele teve como resultado uma entropia cruzada de 1.78, 0.01 menor que o modelo que fez uso da *ReLU* tradicional (que obteve 1.79) e 0.2 menor que o modelo que fez uso da tangente hiperbólica (que obteve 1.98).

Ainda no grupo de redes que possuem quatro camadas, é possível ver um empate ao analisar a acurácia no conjunto de desenvolvimento (*Dev Acc*), que é uma métrica responsável por medir a proporção das previsões corretas feitas pelo modelo quando comparadas com o total de previsões feitas. Assim, perceba que na tabela 8.2, as redes que fizeram uso tanto da *ReLU* quanto da *Leaky ReLU* obtiveram a mesma acurácia de 53.9%, já quando comparadas com a tangente hiperbólica, é possível ver uma diferença de 4.1 pontos percentuais, indicando as funções retificadoras acabam sendo mais precisas para essa análise. Não somente elas são mais precisas, mas quando comparadas com a *Leaky ReLU*, nota-se outros ganhos também, como menores taxas de erro de palavra (*WER*) tanto no conjunto *Switchboard* (*SWBD*) quanto no conjunto de avaliação (*EV*).

Assim, a *Leaky ReLU* já é uma evolução quando comparamos com a ReLU tradicional, mas, é possível ir além e encontrar funções ainda mais complexas que também buscam assim como a *LReLU* resolver o problema dos *ReLUs* agonizantes com um vazamento de gradiente nos casos negativos, uma dessas funções é a *PReLU*, a qual será vista em seguida.

8.4.2 Parametric ReLU (PReLU)

Continuando nas analogias do vendedor de pipoca para explicar as funções retificadoras, podemos também pensar uma para a *Parametric ReLU*. Na *LReLU* nós tínhamos uma constante fixa, que valia para todos os valores de entrada e não mudava, era como se o vendedor de pipoca definisse um valor para a proporção de pipoca que a pessoa irá receber quando tiver com uma quantia menor de dinheiro que o limiar da venda. Mas agora, este vendedor está mais experiente, e sabe que pode ajustar essa constante sempre que quiser, assim, quando estiverem muitas pessoas na praça em que está vendendo pipoca, ele poderá colocar uma constante que será capaz de dar

uma quantidade ainda maior de pipoca para aqueles que não possuem o valor total de um pacote, o que incentivaria a venda para as pessoas. Já quando estivesse em um lugar mais vazio, colocaria uma constante que daria menos pipoca, para maximizar o seu lucro. A Diferença da *PReLU* para a *LReLU* está justamente nessa constante e como ela irá se comportar.

Proposta por He *et al.* (2015) no artigo *Delving Deep into Rectifiers: Surpassing Human Level Performance on Image Net Classification*, a *PReLU* surgiu como uma variação não somente da *ReLU*, mas também uma evolução da *Leaky ReLU* que foi vista anteriormente, isso ocorre, pois diferente da *LReLU* que possuía uma constante α fixa que multiplicava o valor da entrada nos casos negativos, a *PReLU* trás essa mesma constante, mas neste caso ela é adaptável, se ajustando as particularidades de cada problema que uma rede neural está tentando resolver. Assim, a *PReLU* é como o pipoqueiro mais experiente, que ajusta como vai vender o seu punhado de pipoca em cada uma das situações para poder maximizar os seus lucros mas ao mesmo tempo garantir mais clientes para si.

A fórmula matemática da *PReLU* é dada pela Equação 8.7. Como explicam He *et al.* (2015), a *PReLU* generaliza a tradicional *ReLU*, além de melhorar o *model fitting* apresentando quase nenhum custo computacional extra e com um baixo risco de sobreajuste (*overfitting*). Este coeficiente α, que ela apresenta assim como a *Leaky ReLU* que foi visto anteriormente, é aprendível, e não uma constante fixa, isso indica ser otimizado utilizando a retropropagação do gradiente de forma simultânea com as outras camadas da rede neural criada (HE *et al.*, 2015). Assim, por esse fato tem-se uma melhor eficiência no aprendizado e no tempo da rede, dado que não precisamos criar uma nova etapa só para ajustar os valores de *alpha* das camadas densas que fazem o uso da *Parametric ReLU*.

Função de Ativação *Parametric ReLU* (*PReLU*)

$$\text{PReLU}(z_i) = \begin{cases} z_i, & \text{se } z_i \geq 0 \\ \alpha_i \cdot z_i, & \text{se } z_i < 0 \end{cases} \text{ouPReLU}(z_i) = \max(0, \alpha z_i) \qquad (8.7)$$

Com base em sua equação, tem-se o gráfico da *PReLU* na Figura 8.5, note que caso este coeficiente for igual a zero, nos temos então a *ReLU* tradicional, já quando ele for igual a 0.1, tem-se a *LReLU*, no gráfico α está com valor de 0.2, mas ele irá variar conforme a rede aprende e para cada problema, podendo apresentar diferentes valores em diferentes situações. Com isso, é possível perceber que a *PReLU* é composta por duas funções do primeiro grau, sendo assimétrica, e também apresentando um ponto de descontinuidade em zero, o que impede de ser derivada neste ponto.

Conhecendo a sua fórmula e como ela se comporta, cabe também derivar a *PReLU*, para isso, deve-se derivar cada uma das expressões dos condicionais de forma sepa-

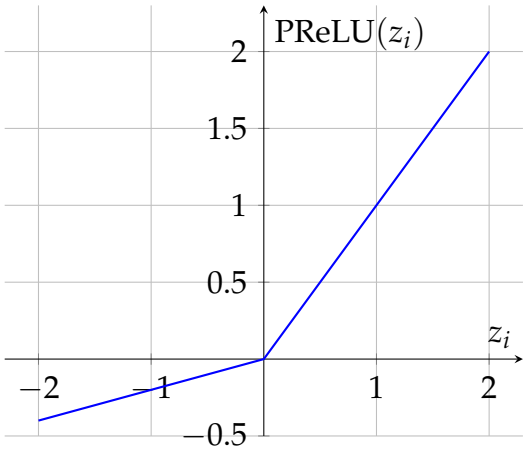


Figura 8.5 – Gráfico da função de ativação *Parametric ReLU* (*PReLU*) com $\alpha = 0.2$.

Fonte: O autor (2025).

rada, assim como foi feito com a *Leaky ReLU* e a *ReLU* anteriormente. Como a fórmula da *PReLU* é igual a *LReLU*, podemos apenas nos lembrar dela e citar a Equação 8.8 como sua derivada. Note que, essa derivada também não existe quando a entrada é exatamente zero, mas assim como na *Leaky ReLU*. É possível "corrigir"isso dizendo que ela será igual a α nesse ponto, para que o gradiente continue fluindo durante o *backward pass*. Vale lembrar, que essa reta em α irá variar conforme a rede neural aprende as características e se ajusta ao problema que está tentando resolver.

Derivada da Função de Ativação *Parametric ReLU* (*PReLU*)

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

Sabendo a fórmula da derivada da *PReLU*, é possível plotar o seu gráfico, o qual é dado pela Figura 8.6, ele é semelhante ao gráfico da *Leaky ReLU* visto na seção anterior, mas agora, a constante α está com valor em 0.2. Note que, o gráfico da derivada é também muito simples, sendo apenas duas funções constantes, uma que vale 1 para os valores de entrada positivos e outra que irá valer 0,2 para os outros valores. Assim, a *PReLU* consegue manter a simplicidade da *ReLU*, mas ao mesmo tempo fazendo pequenos ajustes garantindo melhorias de desempenho em redes mais profundas, como as que foram apresentas por He *et al.* (2015).

Ainda em *Delving Deep into Rectifiers: Surpassing Human Level Performance on Image Net Classification*, os autores realizam testes comparando a *ReLU* tradicional com a *PReLU* utilizando como base o *dataset* de 1000 classes do *ImageNet* 2012, o qual contêm cerca de 1.2 milhões de imagens de treino, 50.000 imagens de validação e 100.000

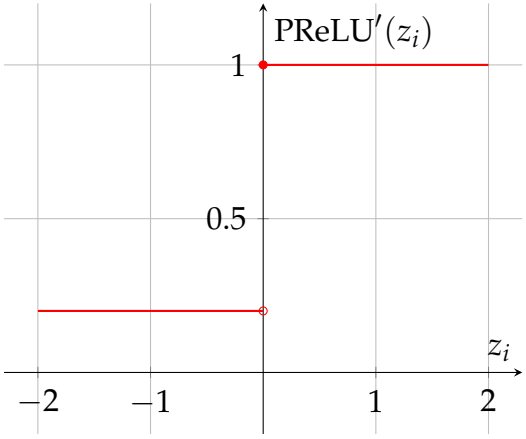


Figura 8.6 – Gráfico da derivada da função de ativação *Parametric ReLU* (*PReLU*) com $\alpha = 0.2$.

Fonte: O autor (2025).

imagens de teste sem rótulos publicados (HE *et al.*, 2015). Para isso, He *et al.* (2015) criaram três modelos diferentes (A, B e C), baseados na arquitetura *VGG-16* mas com variações entre si como diferentes números de camadas convolucionais e consequentemente complexidades distintas para cada algoritmo. Esses modelos podem ser vistos na Tabela 8.3.

Com isso, é possível ver na Tabela 8.4 as métricas utilizadas pelos autores. É medido o erro *Top-1* e o erro *Top-5*, o erro *Top-1* mostrando quão preciso é o modelo em seu melhor chute, já o erro *Top-5* mostra se a resposta correta estava entre os top 5 melhores chutes feito pelo modelo. Assim, conhecendo esses parâmetros de medida, é possível chegar na conclusão de que quanto menor esses valores, melhor o modelo está performando, seguindo essa lógica, nota-se que todos os modelos que fazem uso de funções retificadoras, seja a *PReLU* ou mesmo a *ReLU* tradicional, são capazes de performar melhor que os modelos *VGG-16* e *GoogleLet* que fazem uso de outras funções de ativação.

Ao comparar o modelo C, que faz uso da *PReLU* e possui mais camadas convolucionais, com o modelo A que faz uso da *ReLU*, é possível notar uma diferença de 2,21 pontos percentuais no erro *Top-1*, já ao analisar o erro *Top-5*, essa diferença é de 1,21 pontos percentuais, o que indica que a *PReLU* é capaz de trazer melhores resultados quando comparada com redes que fazem uso da *ReLU* tradicional. Já ao comparar com o *VGG-16*, essa diferença de desempenho é ainda maior, sendo de 3,8 pontos percentuais no erro *Top-1* e 1,95 pontos percentuais no erro *Top-5*, note que a *VGG-16*, a qual é indicada na Tabela 8.3 possui bem menos camadas convolucionais que o modelo C, é possível notar notar também a sua complexidade computacional, que é menos da metade da do modelo C, isso indica que a *PReLU*, por ser uma função não saturante e consequentemente corrigir o problema do desaparecimento do gradiente, é capaz de criar redes neurais que se beneficiam melhor com uma maior profundidade, sendo

Tabela 8.3 – Arquiteturas dos modelos grandes

Input Size	VGG-19 [25]	Model A	Model B	Model C
224	3×3, 64 3×3, 64 2×2 maxpool, /2	7×7, 96, /2	7×7, 96, /2	7×7, 96, /2
112	3×3, 128 3×3, 128 2×2 maxpool, /2	2×2 maxpool, /2	2×2 maxpool, /2	2×2 maxpo
56	3×3, 256 3×3, 256 3×3, 256 3×3, 256 2×2 maxpool, /2	3×3, 256 3×3, 256 3×3, 256 3×3, 256 2×2 maxpool, /2	3×3, 256 3×3, 256 3×3, 256 3×3, 256 2×2 maxpool, /2	3×3, 384 3×3, 384 3×3, 384 3×3, 384 2×2 maxpo
28	3×3, 512 3×3, 512 3×3, 512 3×3, 512 2×2 maxpool, /2	3×3, 512 3×3, 512 3×3, 512 3×3, 512 2×2 maxpool, /2	3×3, 512 3×3, 512 3×3, 512 3×3, 512 2×2 maxpool, /2	3×3, 768 3×3, 768 3×3, 768 3×3, 768 2×2 maxpo
14	3×3, 512 3×3, 512 3×3, 512 3×3, 512 2×2 maxpool, /2	3×3, 512 3×3, 512 3×3, 512 3×3, 512 spp, {7, 3, 2, 1}	3×3, 512 3×3, 512 3×3, 512 3×3, 512 spp, {7, 3, 2, 1}	3×3, 896 3×3, 896 3×3, 896 3×3, 896 spp, {7, 3, 2
fc_1 fc_2 fc_3	4096 4096 1000			
depth (conv+fc)	19	19	22	22
complexity (ops., $\times 10^{10}$)	1.96	1.90	2.32	5.30

Nota: Comparação detalhada das arquiteturas de rede. A notação "/2"denota um stride de 2. Cada col presenta um modelo diferente. A tabela descreve a sequência de camadas convolucionais (no formato × `kernel, n° de filtros`), de pooling e totalmente conectadas (fc). Os modelos A, B e C são variações trutura VGG-19, propostas pelos autores para avaliar a função de ativação PReLU. As linhas finais comp profundidade total (conv+fc) e a complexidade computacional de cada modelo. Fonte: Adaptado de "D Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", por K. He, X. Zh Ren, & J. Sun, 2015, *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pp. 1026-103

Tabela 8.4 – Resultados de Erro Top-1 e Top-5 no Conjunto de Validação ImageNet 2012

Modelo	Erro Top-1 (%)	Erro Top-5 (%)
MSRA	29.68	10.95
VGG-16	28.07[a]	9.33
GoogleLeNet	-	9.15
A, ReLU	26.48	8.59
A, PReLU	25.59	8.23
B, PReLU	25.53	8.13
C, PReLU	**24.27**	**7.38**

Nota: Resultados de erro (%) para um único modelo com a técnica de 10-view no conjunto de validação do ImageNet 2012. Os modelos A, B e C são variações da arquitetura VGG-16 modificada pelos autores. Os valores em negrito indicam os melhores resultados. [a]Resultado baseado em testes realizados pelos autores do artigo original.
Fonte: Adaptado de "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", por K. He, X. Zhang, S. Ren, & J. Sun, 2015, *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pp. 1026-1034.

capazes de extrair mais informações e com isso performar melhor.

Um feito importante que deve ser destacado sobre a *PReLU*, que inclusive é o nome do artigo que foi responsável por introduzir essa função para a comunidade científica, é de que durante a pesquisa do texto *Delving Deep into Rectifiers: Surpassing Human Level Performance on Image Net Classification*, He *et al.* (2015) foram capazes de criar uma rede neural capaz de superar a capacidade humana de reconhecer diferentes conjuntos de imagens no *ImagneNet* Classification, isso ocorreu porque a um humano ao analisar o *ImageNet* apresenta uma taxa de erro *Top-5* de 5.1% e em um dos testes realizados pelos autores, uma rede neural alcançou uma taxa de erro *Top-5* de 4.94%, superando assim a capacidade humana de reconhecimento de imagens.

Assim, é nítido destacar que não somente a *PReLU*, mas as funções retificadoras de forma geral, foram capazes de trazer melhorias significativas para os modelos de aprendizado profundo quando comparadas com as funções sigmoides, as quais foram o padrão da indústria por muitos anos. De fato, a resolução do problema do desaparecimento do gradiente, possibilitou a criação de redes neurais ainda mais profundas, e com isso, sendo capazes de extrair mais informações e consequente melhores métricas, sendo capazes até de superar a capacidade humanas em algumas tarefas como no artigo de introdução da *PReLU*.

8.4.3 Randomized Leaky ReLU (RReLU)

Anteriormente, na *Parametric ReLU*, existia uma padrão aprendível que era atualizado ao longo da retropropagação da rede, nós o comparamos com o caso do vendedor de pipoca ficando mais experiente para as vendas. No caso da *RReLU* temos um vendedor um tanto quanto instável, ele se baseia na sorte/aleatoriedade para definir qual será o punhado de pipoca que cada pessoa irá receber ao comprar com um valor abaixo do limiar de venda. Para isso, não existe mais um padrão, algo como o horário ou a quantidade de pessoas na praça para fazer aumentar o diminuir a quantidade de pipoca que tera em um punhado. Essa estratégia parece caótica, mas pode ser interessante caso voce queira instigar as vendas e deixa-las divertidas, você pode comprar um punhado de pipoca, mas nao ira saber quanto irá receber, é uma grande aposta.

Segundo Xu *et al.* (2015), em *Empirical Evaluation of Rectified Activations in Convolutional Network*, a *Randomized Leaky ReLU* foi proposta pela primeira vez em uma competição do *Kaggle NDSB*, ela era uma função semelhante a *leaky ReLU* mas que o seu coeficiente α é um número aleatório dado pela distribuição normal da forma $U(l,u)$, nessa mesma competição os valores escolhidos para essa distribuição foram de $U(3,8)$.

A fórmula da *RReLU* é dada pela Equação 8.9, note que é a mesma expressão da *Leaky ReLU* e da *PReLU*, mas o que muda é o significado do termo α em cada uma delas. Neste caso: $\alpha \sim U(l,u)$ em que $l < u$ e $l,u \in [0,1)$

função de Ativação *Randomized Leaky ReLU* (*RReLU*)

$$\text{RReLU}(z_i) = \begin{cases} z_i, & \text{se } z_i > 0 \\ \alpha_i z_i, & \text{se } z_i \leq 0 \end{cases} \tag{8.9}$$

Na fase de testes, deve-se calcular a média de todos os valores de α durante o treino, e com isso α se torna uma constante fixa do tipo $(l+u)/2$ de forma que com isso seja possível obter um resultado determinístico, no artigo, os autores utilizam a fórmula 8.10 para calcular a *RReLU* durante o teste do modelo (XU *et al.*, 2015).

$$\text{RReLU}(z_i) = \frac{z_i}{\frac{l+u}{2}} \tag{8.10}$$

O gráfico da *RReLU* está presente na Figura 8.7. Na representação, é possível ver várias retas, isso ocorre pois elas irão variar de caso a caso, e como a *RReLU* é uma função que utiliza de conceitos probabilísticos, não é possível garantir um gráfico exato de como ela seria pois não temos os valores de α até que a distribuição seja feita. Note que mesmo com essa particularidade, ela ainda continua sendo uma função bem simples, sendo a construção de duas retas originarias de equações do primeiro grau, a primeira delas sendo a própria função identidade, para os casos em que a entrada é positiva, e a outra é dada pela variável da entrada multiplicada pela constante α, para os casos em

que a saída é negativa. Além disso, deve-se atentar também para a sua descontinuidade no ponto zero, é o mesmo problema que acontece com outras variantes, como a *ReLU* e a *Leaky ReLU*.

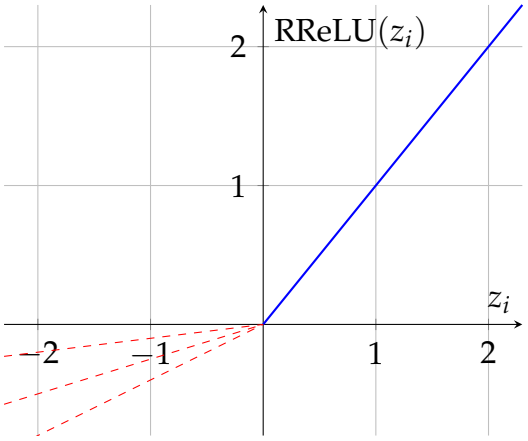


Figura 8.7 – Gráfico da função de ativação *Randomized Leaky ReLU* (*RReLU*) com diferentes inclinações aleatórias para a parte negativa.

Fonte: O autor (2025).

Conhecendo como a *RReLU* se comporta, é possível também calcular a sua derivada, a qual será de extrema utilidade durante a retropropagação, fazendo que os pesos e vieses do modelo sejam ajustados e com base nisso ele consiga aprender melhor o problema que está sendo analisado. Como a *RReLU* utiliza a mesma fórmula que funções como a *Leaky ReLU* e a *PReLU*, pode-se apenas repetir a expressão de sua derivada novamente, a qual será dada pela Equação 8.11. Note que mesmo compartilhando a mesma fórmula, o termo α possui significado distintos em cada uma dessas funções, neste caso, ele é um valor aleatório dado pela distribuição $U(l, u)$.

Com relação ao problema da descontinuidade no ponto zero, é possível apenas escolher para qual valor essa função irá retornar neste caso, assim, vamos considerar que quando a sua entrada for zero, ela irá retornar o segundo caso, em que é a própria constante α

Derivada da Função de Ativação *Randomized Leaky ReLU* (*RReLU*)

$$\frac{d}{dz_i}[RReLU](z_i) = \begin{cases} 1, & \text{se } z_i > 0 \\ \alpha_i, & \text{se } z_i \leqslant 0 \end{cases} \tag{8.11}$$

Esse detalhe da aleatoriedade da constante α afetou o desenho do gráfico da *RReLU*, e com isso, ele também afeta a plotagem de sua derivada. Como pode ser visto na Figura 8.8, existem um conjunto de retas em um intervalo, neste caso, está sendo considerado a distribuição como sendo de $U(0.1, 0.3)$, mas para cada uma dessas distribu-

ições, terá um conjunto de retas diferentes e com isso gráficos distintos para cada um dos problemas.

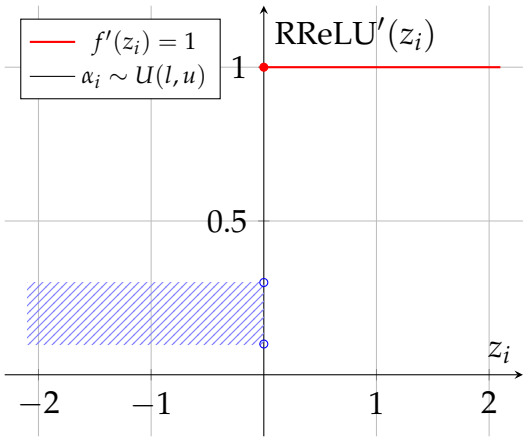


Figura 8.8 – Gráfico da derivada da função de ativação *Randomized Leaky ReLU* (*RReLU*) com $l = 0.1, u = 0.3$.

Fonte: O autor (2025).

Ainda no artigo, Xu *et al.* (2015) investigam a performance de diferentes funções retificadoras em uma rede neural convolucional, cuja arquitetura pode ser vista na Tabela 8.5, para a classificação de imagens, os autores analisaram a *ReLU* tradicional, a *Leaky ReLU*, a *Parametric ReLU* e a *Randomized Leaky ReLU*. Para fazer a análise dos modelos, foram escolhidos os datasets *CIFAR-10* e *CIFAR-100*, e o desempenho dessas funções nos respectivos datasets pode ser visto na Tabela **??** e na Tabela **??**.

Analisando a Tabela 8.6, que mostra a taxa de erro das funções retificadoras na rede *NIN* para o *dataset CIFAR-10*, é possível notar que a *Randomized Leaky ReLU* foi a função que performou melhor, com um total de 11.19% de erro nos casos de teste, enquanto a *leaky ReLU* com $a = 100$ obteve o pior resultado. Contudo, essa diferença de resultado é pequena, indicando que caso essas funções sejam muito mais complexas quando comparadas com a *ReLU* na hora de treinar o modelo, pode ser melhor optar por uma função mais "barata"mas com uma taxa de erro um pouco maior.

Já ao analisar a Tabela 8.7, que mostra a taxa de erro dessas funções no *dataset CIFAR-10*, é possível ver que assim como no *CIFAR-10*, a *RReLU* foi a função que obteve melhor resultado, neste caso há uma diferença de 2.65 pontos percentuais, quando comparada com a *ReLU* tradicional. Outro ponto interessante a ser destacado ao analisar essa tabela é de que provavelmente a rede que utilizou a *Parametric ReLU* sofreu um sobreajuste (*overfitting*) fazendo com que ela decorasse o dados de treino e com isso conseguisse uma taxa de erro consideravelmente menor, já quando ela foi apresentada para o conjunto de testes houve uma grande disparidade das taxas de erro.

Ainda em *Empirical Evaluation of Rectified Activations in Convolutional Network* Xu *et al.* (2015) explicam que a *RReLU* é uma função que ajuda a combater o sobreajuste (overfitting) do modelo, mas ainda devem ser feitos mais testes para descobrir como a

Tabela 8.5 – Estrutura da Rede "Network in Network"(NIN) para CIFAR-10/100

Tamanho da Entrada	Camada / Operação
32×32	Conv 5x5, 192 canais
32×32	Conv 1x1, 160 canais
32×32	Conv 1x1, 96 canais
32×32	Max Pooling 3x3, stride /2
16×16	Dropout, taxa 0.5
16×16	Conv 5x5, 192 canais
16×16	Conv 1x1, 192 canais
16×16	Conv 1x1, 192 canais
16×16	Avg Pooling 3x3, stride /2
8×8	Dropout, taxa 0.5
8×8	Conv 3x3, 192 canais
8×8	Conv 1x1, 192 canais
8×8	Conv 1x1, 10 ou 100 canais[a]
8×8	Global Avg Pooling 8x8
10 ou 100	Softmax

Nota: Descrição da arquitetura da rede convolucional "Network in Network"(NIN). A coluna "Tamanho da Entrada"indica a dimensão espacial dos mapas de características em cada estágio. A coluna "Camada / Operação"detalha a sequência de operações da rede. [a]O número de canais na última camada convolucional corresponde ao número de classes do dataset (10 para CIFAR-10 ou 100 para CIFAR-100), funcionando como uma etapa de classificação antes do Global Average Pooling. Fonte: Adaptado de "Empirical Evaluation of Rectified Activations in Convolutional Network", por B. Xu, N. Wang, T. Chen, & M. Li, 2015, *arXiv preprint arXiv:1505.00853*.

aleatoriedade afeta os processos de treino e teste (XU *et al.*, 2015). Essa característica de ajudar a combater o sobreajuste é uma vantagem que a *RReLU* possui, permitindo com que modelos maiores e mais profundos possam ser criados e mesmo assim obtenham resultados significativos. Provavelmente, um dos motivos dela possuir essa função está no fato de que ela introduz uma maior aleatoriedade para o modelo, ajudando a impedir que ele decore os padrões, como em imagens dos conjuntos *CIFAR-10* e *CIFAR-100*.

Tabela 8.6 – Taxa de Erro de Diferentes Funções de Ativação na Rede NIN para o CIFAR-10

Função de Ativação	Erro de Treino	Erro de Teste (%)
ReLU	0.00318	12.45
Leaky ReLU ($a = 100$)	0.00310	12.66
Leaky ReLU ($a = 5.5$)	0.00362	11.20
PReLU	0.00178	11.79
RReLU[a]	0.00550	**11.19**

Nota: Comparação da taxa de erro da rede "Network in Network"(NIN) treinada no conjunto de dados CIFAR-10 com diferentes funções de ativação retificadoras. Os valores de erro de teste são apresentados em porcentagem (%). O valor em negrito indica o melhor resultado (menor erro de teste). [a]Randomized Leaky ReLU (RReLU), onde o coeficiente de vazamento é amostrado de uma distribuição uniforme durante o treino. A fórmula na tabela original representa uma parametrização específica testada. Fonte: Adaptado de "Empirical Evaluation of Rectified Activations in Convolutional Network", por B. Xu, N. Wang, T. Chen, & M. Li, 2015, *arXiv preprint arXiv:1505.00853*.

Tabela 8.7 – Taxa de Erro de Funções de Ativação na Rede NIN para o CIFAR-100

Função de Ativação	Erro de Treino (%)	Erro de Teste (%)
ReLU	13.56	42.90
Leaky ReLU ($a = 100$)	11.55	42.05
Leaky ReLU ($a = 5.5$)	8.54	40.42
PReLU	**6.33**	41.63
RReLU[a]	11.41	**40.25**

Nota: Comparação da taxa de erro (%) da rede "Network in Network"(NIN) treinada no conjunto de dados CIFAR-100. Os valores em negrito indicam o melhor resultado (menor erro) em cada coluna. [a]Randomized Leaky ReLU (RReLU), onde o coeficiente de vazamento é amostrado de uma distribuição uniforme durante o treino. Fonte: Adaptado de "Empirical Evaluation of Rectified Activations in Convolutional Network", por B. Xu, N. Wang, T. Chen, & M. Li, 2015, *arXiv preprint arXiv:1505.00853*.

8.5 As Variantes Não Lineares: Em Busca da Suavidade

Seguindo adiante, agora será visto um novo conjunto de variantes da *ReLU* tradicional, elas incluem funções que apresentam gráficos com curvas mais suaves, como é o caso da *ELU*, que faz uso de funções exponenciais para a sua composição e com isso consegue não só resolver o problema dos *ReLUs* agonizantes, mas também sendo uma

função contínua na origem e portanto derivável em todo o seu domínio.

Além disso, é possível conhecer também uma variante da *ELU*, a *Scaled Exponetial Linear Unit*, uma função que é utilizada para construir redes capazes de se autonormalizarem, ademais será visto a *Noisy ReLU*, outra variante da *ReLU*, mas que dessa vez adiciona ruído em sua saída a fim de garantir uma melhor performance quando comparada com a sua função original.

8.5.1 Exponential Linear Unit (ELU)

Continuando com as analogias do vendedor de pipoca, o vendedor de pipoca que faz uso da *ELU* para as suas vendas trabalha de forma diferente. Ao invés de vender um punhado de pipoca de forma linear com base no dinheiro que o cliente tem quando ele não quer comprar o pacote inteiro por não possuir o valor total, ele adota uma curva exponencial como base, assim, clientes com valores muito próximos de R$ 5,00 recebem uma quantia muito grande de pipoca, quase equivalente ao pacote total, enquanto aqueles que possuem valores pequenos irão receber um punhado pequeno de pipoca. Talvés seja uma forma de incentivar aqueles que quase tem o valor total para comprar uma pipoca, mas ainda sim garantir a clientela dos que possuem pouco dinheiro e aumentando o seu lucro como vendendor.

A *ELU* ou *Exponential Linear Unit* foi introduzida no artigo *Fast and Accurate Deep Networks Leaning By Exponential Linear Units (ELUs)*, sendo uma variação que acelera o aprendizado de uma rede neural densa e apresentando uma maior acurácia em problemas de classificação (CLEVERT; UNTERTHINER; HOCHREITER, 2016).

A *Exponential Linear Unit* pode é descrita utilizando a Equação 8.12. Note que há uma grande diferença dela quando comparamos com a *ReLU*, a *ELU* faz uso de funções exponenciais, algo que é computacionalmente mais "caro"para um computador quando comparado com apenas cálculos simples como uma função identidade.

Função de Ativação *Exponential Linear Unit* (*ELU*)

$$\text{ELU}(z_i) = \begin{cases} z_i, & \text{se } z_i \geq 0 \\ \alpha \cdot (e^{z_i} - 1), & \text{se } z_i < 0 \end{cases} \tag{8.12}$$

Conhecendo como é a fórmula da *ELU*, é possível também plotar o seu gráfico, o qual está presente na Figura 8.9. Ao analisar, é possível ver uma diferença notável quando o comparada com as funções vistas anterioemente, a *ELU* não apresenta um bico no ponto de origem, ela é uma função bem mais suave. Além disso, note que ela segue o mesmo padrão das outras funções: ela retorna a função identidade nos casos em que a entrada é maior que zero, assim como as outras variantes, mas quando é tratado dos casos em que a entrada é negativa, nota-se que ela utiliza uma funcão ex-

ponencial, o que garante a suavidade vista no gráfico. Perceba também que ela possui valores negativos, assim como as variantes com vazamento, indicando que ela também pode ser capaz de lidar com o problema do *ReLUS* agonizantes causado pela *ReLU* tradicional e que vem sendo mitigado com outras variantes como a *Leaky ReLU*.

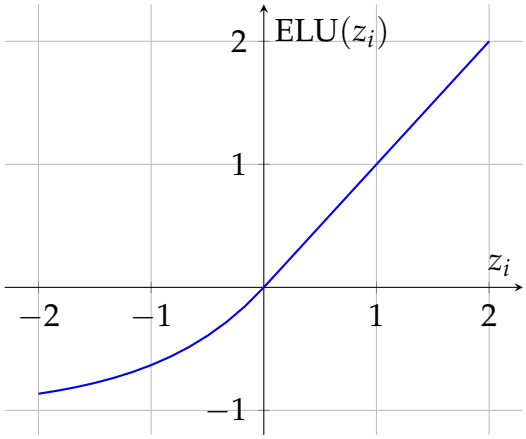


Figura 8.9 – Gráfico da função de ativação *Exponential Linear Unit* (*ELU*) com $\alpha = 1$.

Fonte: O autor (2025).

Agora que a sua fórmula e seu gráfico são conhecidos, cabe também calcular a sua derivada, a qual será útil na retropropagação do modelo. Para isso, é possível seguir a mesma estratégia vista até agora, derivando a função em cada um dos casos, gerando assim a sua derivada. Nos cenários em que a entrada é positiva, a derivada será sempre 1, pois quando derivamos a expressão x, ela nos irá retornar 1. Já quando temos o cenário negativo, teremos como resultado da derivação da expressão $alpha \cdot (e^{z_i} - 1)$ o termo $alpha \cdot e^{z_i}$. Um ponto a ser destacado é de que a *ELU* é contínua na origem, assim, não tendo que preocupar em escolher um valor da derivada quando o seu valor de entrada for zero. Assim, tem-se como resultado final a Equação 8.13

Derivada da Função de Ativação *Exponential Linear Unit* (*ELU*)

$$\frac{d}{dz_i}[ELU](z_i) = \begin{cases} 1, & \text{se } z_i > 0 \\ \alpha \cdot e^{z_i}, & \text{se } z_i \leq 0 \end{cases} \tag{8.13}$$

Sabendo a sua derivada, pode-se também plotar o seu gráfico, para isso, ele está representado na Figura 8.10. Note que ele é composto de duas partes diferentes, sendo a primeira delas, para os casos em que a entrada é negativa, a função constante em um, e para os casos em que a entrada é negativa, tem-se uma curva exponencial. Perceba também que a sua derivada irá sempre retornar valores positivos quando é calculada para qualquer ponto do seu domínio.

Um dos testes realizados pelos autores para analisar o desempenho na *ELU*, foi na criação de uma *CNN* com 18 camadas convolucionais para fazer a classifição dos

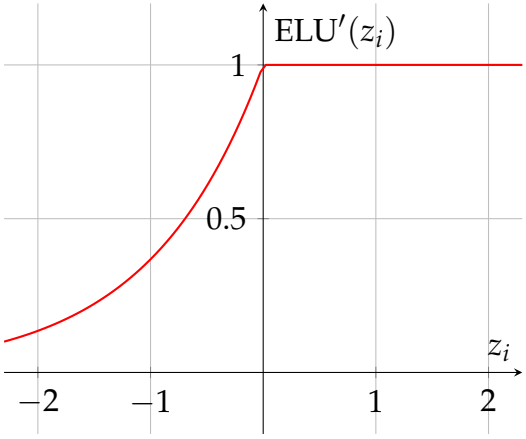


Figura 8.10 – Gráfico da derivada da função de ativação *Exponential Linear Unit* (*ELU*) com $\alpha = 1$.

Fonte: O autor (2025).

datasets *CIFAR-10* e *CIFAR-100*, para isso, outras técnicas foram utilizas em conjunto como o decaimento do peso L2 e reduções das taxas de aprendizado (CLEVERT; UNTERTHINER; HOCHREITER, 2016).

Com base nessa *CNN* e seus experimentos, é possível ver os resultados na Tabela 8.8, em que Clevert, Unterthiner e Hochreiter (2016) comparam a *ELU* com outras redes no mesmo problema, como a *AlexNet* que foi vista anterioremente na explicação do surgimento da *ReLU*. Ao analisar esses resultados, nota-se que a *ELU* obteve um desempenho excelente no dataset *CIFAR-100*, com uma diferença de 21.52 pontos percentuais quando comparada com a *AlexNet*, que ficou em último lugar. Já ao considerar o seu desempenho para um problema de classificação mais simples, como o *CIFAR-10*, ela ficou em segundo lugar, estando atrás apenas da *Fract. Max-Pooling*, mas ainda sim, apresentando uma diferença considerável de 2.05 pontos percentuais a mais de erro.

Isso indica que a *ELU* é uma excelente opção para problemas de classificação, especialmente se há um grande número de classes a ser analisadas. Contudo, uma rede neural convolucional que apresenta 18 camadas de convolução pode ser um tanto quanto custosa para ser processada por um computador, assim, faz se necessário o uso de unidades de *GPUs* para o processamento de uma rede como essa, para que mesmo sendo pesada para ser processada, os resultados possam sair um pouco mais rápidos.

Por fim, cabe destacar algumas afirmações realizadas pelos autores ainda em *Fast and Accurate Deep Networks Leaning By Exponential Linear Units (ELUs)*, segundo Clevert, Unterthiner e Hochreiter (2016), ao comparar a *ELU* com funções como a *ReLU tradicional* e *Leaky ReLU*, pode-se notar um melhor e mais rápido aprendizado, além de que a *Exponetial Linear Unit* é capaz de garantir uma melhor generalização quando passa a ser utilizada em redes com mais de cinco camadas. Outro ponto destacado pelos autores, está no fato da *ELU* garantir *noise rebust deactivation states*, algo que mesmo

Tabela 8.8 – Comparativo da Taxa de Erro de Redes Neurais nos Datasets CIFAR

Arquitetura da Rede	CIFAR-10 Erro (%)	CIFAR-100 Erro (%)	Augmentation
AlexNet	18.04	45.80	
DSN	7.97	34.57	v
NiN	8.81	35.68	v
Maxout	9.38	38.57	v
All-CNN	7.25	33.71	v
Highway Network	7.60	32.24	v
Fract. Max-Pooling	**4.50**	27.62	v
ELU-Network	6.55	**24.28**	v

Nota: Comparação da taxa de erro de classificação (%) no conjunto de teste para diversas arquiteturas de redes neurais convolucionais (CNNs). Os valores em negrito indicam o melhor resultado em cada dataset. A coluna "Augmentation"indica se foram utilizadas técnicas de aumento de dados (data augmentation) durante o treinamento, marcado com "v". Fonte: Adaptado de "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)", por D. Clevert, T. Unterthiner, & S. Hochreiter, 2015, *arXiv preprint arXiv:1511.07289*.

com a *Leaky ReLU* e *Parametric ReLU* possuindo valores negativos, não são capazes de garantir ao serem utilizadas para construir uma rede neura (CLEVERT; UNTERTHINER; HOCHREITER, 2016).

Mesmo apresentando um grande salto, quando comparada com a *ReLU* tradicional, a *ELU* também pode ser modificada para atender outros casos. Para isso, ela também tem variações, sendo uma delas a *SELU*, a qual adiciona a *ELU* um termo λ para garantir uma autonormalização da rede que está sendo criada. Essa função será vista em seguida.

8.5.2 Scaled Exponential Linear Unit (SELU)

A próxima função é uma variante da *ELU*, a *Scaled Exponential Linear Unit*, ou *SELU*. Ela se distingue da ELU tradicional pelo fato de que ela é capaz de implemtar propriedades auto-normalizadoras em uma rede que faz uso dessa função, como explicam os autores no artigo de sua introdução *Self-Normalizing Neural Networks* (KLAMBAUER *et al.*, 2017).

Os autores apresentam a Equação 8.14 para calcular essa função, em que o termo λ é uma constante que será maior que 1 (KLAMBAUER *et al.*, 2017). Note que essa fórmula é bem parecida com a da *Exponential Linear Unit*, a única diferença é que ela estará sendo multiplicada pela constante λ, assim pode-se escrever também que $\text{SELU}(z_i) = \lambda\text{ELU}(z_i)$.

Função de Ativação *Scaled Exponential Linear Unit* (*SELU*)

$$\text{SELU}(z_i) = \lambda \begin{cases} z_i, & \text{se } z_i > 0 \\ \alpha \cdot (e^{z_i} - 1), & \text{se } z_i \leq 0 \end{cases} \tag{8.14}$$

Com relação ao seu gráfico, ele está presente na Figura 8.11, neste caso, está sendo considerado que as constantes α e λ são dadas por 1.7 e 1.05 respectivamente. Note que é um gráfico que lembra bastante a *Leaky ReLU*, mas que neste caso, quando a função recebe valores negativos, ela não estará mais assumindo o comportamento de uma reta, e sim o de uma curva exponencial, já para os cenários em que a entrada é postiva os resultados serão próximos os de uma função identidade, mas com uma reta um pouco mais inclinada.

Pelo fato da *SELU* ter um comportamento que também retorna valores para a saída quando a sua entrada é negativa, ela consegue combater o problema dos *ReLUs* agonizantes, causado pela *ReLU*, além de que também não é uma função saturante, como a sigmoide, o que também ajuda a resolver o problema no desaparecimento do gradiente. Mas, por não ser uma função saturante, e pelo fato de que sua saída vai para valores infinitos conforme os valores de sua entrada aumentam, ela está sujeita ao problema da explosão de gradientes.

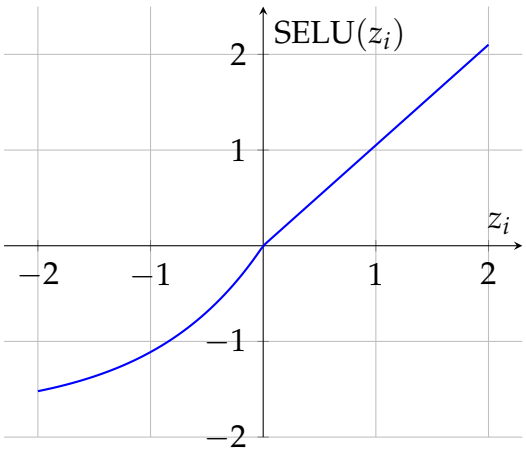


Figura 8.11 – Gráfico da função de ativação *Scaled Exponential Linear Unit* (*SELU*) com $\alpha \approx 1.67 e \lambda \approx 1.05$.

Fonte: O autor (2025).

Considerando agora como é a equação da *SELU* e qual é o seu comportamento no gráfico, é possível também calcular a sua derivada para ser utilizada na retroproapagação do gradiente, para isso, deve-se derivar a Equação 8.14, considerando os dois cenários, em que a sua entrada será positiva e quando sua entrada for negativa ou zero. Um ponto que ajuda bastante ao calcular a derivada da *SELU* está no fato dela ser composta pela função *ELU* multiplicada por uma constante, se utilizarmos regras

de derivação para esse cenário, precisaremos apenas derivar a *ELU* e depois adicionar a constante λ multiplicando-a. Como a derivada da *ELU* já foi calculada, ela pode ser aproveitada agora. Você pode ver então a derivada da *Scaled Exponential Linear Unit* na Equação 8.15.

Derivada da Função de Ativação *Scaled Exponential Linear Unit* (*SELU*)

$$\frac{d}{dz_i}[SELU](z_i) = \lambda \begin{cases} 1, & \text{se } z_i > 0 \\ \alpha \cdot e^{z_i}, & \text{se } z_i \leq 0 \end{cases} \tag{8.15}$$

Sabendo a sua derivada, é possível também plotar o seu gráfico, para isso, ele está na Figura 8.12. Note, que o gráfico da derivada da *SELU* também possui grandes similaridades com a *ELU* original mas também com as outras retificadoras, pois também pode ser dividido em duas partes principais. A primeira parte, para os casos em que a entrada é negativa segue o comportamento de uma curva exponencial, enquanto a segunda parte é semelhante a uma reta constante com inclinação zero.

Um ponto interessante dessa reta da segunda parte é que ela retorna justamente um valor bem próximo de um, assim como nas retificadoras, isso trás como benefício uma menor chance para ocorrer casos de desaparecimento do gradiente, pois ele não estará sendo constantemente sendo multiplicado por valores pequenos e com isso reduzindo o seu valor. Mas, por outro lado, isso também acaba colaborando para que gradientes explosivos possam ocorrer.

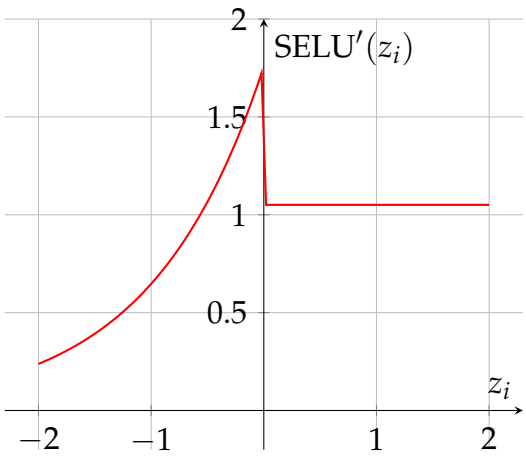


Figura 8.12 – Gráfico da derivada da função de ativação Scaled Exponential Linear Unit (SELU) com $\alpha \approx 1.67$, $\lambda \approx 1.05$.

Fonte: O autor (2025).

No texto da introdução da *SELU*, Klambauer *et al.* (2017), comparam as redes neurais criadas por eles, as quais são chamadas de *Self-Normalizing Neural Networks* (*SNN*), com outras redes *feedforward*, como *MSRAinit* (uma *FNN* que não possui técnicas de

normalização, com funções de ativação *ReLU*, e que faz uso do *Microsoft weight initialization*), a *BatchNorm* (uma *FNN* com normalização em lote), a *LayerNorm* (uma *FNN* com normalização nas camadas), a *WightNorm* (uma *FNN* com normalização nos pesos), a *Highway* e também com redes residuais *ResNet*. Para comparar essas redes, os autores escolhem 121 *datasets* do *UCI*, em que são apresentadas áreas de aplicação diversas como física e biologia, nesses *datasets* o seus tamanhos podem variar de 10 até 130.000 pontos de dados com o número de features variando de 4 a 250. Na Tabela 8.9, é possível ver o ranking médio entre as *SNNs* e as outras diferentes arquiteturas em 75 tarefas de classificação.

Tabela 8.9 – Comparativo do Rank Médio entre SNNs e Outras Arquiteturas de Redes Neurais

Grupo do Método	Método	Rank Médio	Valor-p
SNN	SNN	9.6	3.8×10^{-1}
MSRAinit	MSRAinit	11.0	4.0×10^{-2}
LayerNorm	LayerNorm	11.3	7.2×10^{-2}
Highway	Highway	11.5	8.9×10^{-3}
ResNet	ResNet	12.3	3.5×10^{-3}
BatchNorm	BatchNorm	12.6	4.9×10^{-4}
WeightNorm	WeightNorm	13.0	8.3×10^{-5}

Nota: Comparação do rank médio de diferentes arquiteturas de redes neurais em 75 tarefas de classificação do repositório UCI. O "Rank Médio"é a média das classificações de acurácia entre as tarefas. O "Valor-p"corresponde ao teste de Wilcoxon pareado para avaliar se a diferença para o método de melhor desempenho é significativa. Fonte: Adaptado de "Self-Normalizing Neural Networks", por G. Klambauer, T. Unterthiner, A. Mayr, & S. Hochreiter, 2017, *arXiv preprint arXiv:1706.02515*.

Para analisar essa tabela, pode-se primeiro olhar o ranking médio de cada uma desses modelos, que é dado pela média de como esses modelos performaram nos diferentes datasets, considerando isso, note que as redes que fazem uso da *SELU* em sua composição, as *SNNs*, são as melhores, por uma diferença de 1.4 pontos quando comparadas com o segundo colocado, isso indica que a *SELU* pode ser uma ótima alternativa quando ainda não se sabe exatamente qual será o conjunto de dados que será trabalhado, se ele será de conceitos como física ou geologia, assim, elas garantem uma maior versatilidade para encarar diversos problemas. Além disso, ao olhar também o seu *p-value*, que vem de um teste de Wilcoxon pareado para verificar se a diferença em relação ao melhor colocado é signifitiva, percebe-se que as *SNNs* continuam se destacando, com o *p-value* mais alto, indicando que existe uma diferença que não é estatisticamente significativa quando ela é comparada com o modelo *SVM*, mostrando que podemos considerar as *SNNs* como se tivessem empatadas com o campeão.

O interessante dessa comparação é analisar ela considerando aquelas redes que fazem uso de técnicas de normalização para conseguir uma maior desempenho, como a *BatchNorm* e a *LayerNorm*, cada uma delas utiliza uma técnica de normalização diferente de forma a garantir que que problemas como os gradientes explosivos e desaparacimento do gradiente não ocorra com tanta frequência e com isso permitindo um melhor convergência do modelo que está sendo treinado. Ao olhar por essa ótica, pode-se chegar a conclusão que criar uma rede neural utilizando a *SELU* não só irá garantir uma maior versatilidade para a resolução de problemas, como também você não terá que se preocupar em aplicar técnicas de normalização ao construir essa rede.

Voltando para o seu texto de introdução, os autores destacam propriedades importantes dessa nova função de atiavção criada, como o fato de que de que elas possbilitam a criação de redes neurais mais profundas além de serem capazes de aplicar fortes esquemas de regularização (KLAMBAUER *et al.*, 2017). Por favorecer a criação de RNAs mais profundas, como consequência, a *SELU* se torna uma excelente alternativa para ser utilizada em problemas complexos, que possuem muitas características e por essa razação necessitam de que mais camadas sejam construídas a fim de garantir um melhor processamento e aprendizado dos dados e com base nisso, alcançar métricas maiores, como uma maior acurácia indicando uma gerenalização maior e também uma perda menor, indicando que o gradiente conseguiu uma convergência melhor.

8.5.3 Noisy ReLU (NReLU)

Seguindo adiante, é possível conhecer a *Noisy ReLU*, também conhecida como *NReLU*. Um dos trabalhos que explora as característica dessa função e como ela pode ser aplicada em uma RNA é o *Rectified Linear Units Improve Restricted Boltzmann Machines* dos autores Nair e Hinton (2010). Nesse texto, os autores comparam o desenpenho dessa função com a função binária, dada pela equação **??**, que era a opção mais comum para ser utilizada na construção de máquinas restritas de Boltzmann.

Função de Ativação Binária (*Binary*)

$$f(z_i) = \begin{cases} 1 & \text{se } z_i \geq \theta \\ 0 & \text{se } z_i < \theta \end{cases} \tag{8.16}$$

No texto, Nair e Hinton (2010), apresentam a *NReLU* como sendo dada pela equação $\max 0, z_i + \mathcal{N}(0, \sigma(z_i))$, em que $\mathcal{N}(0, V)$ representa o ruído Gaussiano (*Gaussian noise* em inglês), com méida zero e vairância dada por V. Também é possível expressar a Noisy ReLU com a Equação 8.17.

Função de Ativação Noisy ReLU (*NReLU*)

$$\text{NReLU}(z_i) = \begin{cases} 0 & \text{se } z_i \leq 0 \\ z_i + \mathcal{N}(0, \sigma(z_i)) & \text{se } z_i > 0 \end{cases} \tag{8.17}$$

Antes de seguir em frente, é útil entende primeiro o que é ruído gaussiano, e para isso, é preciso entender antes a distribuição gaussiana. Segundo Goodfellow, Bengio e Courville (2016), a distribuição gaussiana é a distribuição mais utilizada para números reais, ela também é conhecida por ser chamada de distribuição normal, ela é dada pela Equação 8.18.

$$\mathcal{N}(x; \mu, \sigma^2) = \sqrt{\frac{1}{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right) \tag{8.18}$$

Nessa equação, os dois parâmetros $\mu \in \mathbb{R}$ e $\sigma(0, \infty)$ controlam como a distribuição normal funciona; o termo μ é responsável por dar as coordenadas para o pico do centro, que é também a média da distribuição $\mathbb{E}[x] = \mu$, já o desvio padrão é dado por σ, equanto a variância é denotada por σ^2 (GOODFELLOW; BENGIO; COURVILLE, 2016). Para chegarmos no ruído gaussiano, é preciso então adicionar como parâmetros da equação da distribução gaussiana (Equação 8.18) os termos que são dados pela Equação 8.17.

A distribuição normal, com os parâmetros $\mu = 0$ e $\sigma = 1$, é responsável por gerar um gráfico em formato de sino, como é mostrado na Figura 8.13. Esse gráfico indica quais são os casos que possuem uma maior probabilidade de acontecer, os casos que estão no centro, onde, $p(x)$ possuem uma maior probabilidade de acontecer, já conforme eles se distânciam desse centro essa propabilidade diminui.

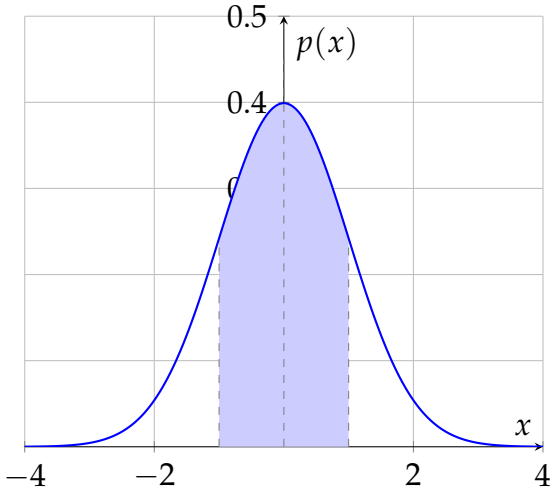


Figura 8.13 – Gráfico da Distribuição Gaussiana (ou Normal) para o caso padrão, com média 0 ($\mu = 0$) e desvio padrão 1 ($\sigma = 1$).

Fonte: O autor (2025).

Entendendo a estrutura e como funciona a *Noisy ReLU*, é possível plotar o seu gráfico, o qual é dado pela Figura 8.14. Note que ela compartilha a suavidade das funções *ELU* e *SELU*, apresentando uma curva para os casos em que a entrada é negativa, o que é bom, pois ela permite um vazamento de gradiente, evitando assim o o problema do *ReLUs* agonizantes. Já para os cenários que a entrada é positiva ela assume o comportamento de uma função identidade, lembrando bastante as outras variantes da *ReLU*. Contudo, como os autores destacam no texto, ela não é capaz de resolver o problema da descontinuidade em zero, para isso, em sua derivada que é utilizada na retropropagação, os casos em que sua entrada é zero irão retornar zero como saída (NAIR; HINTON, 2010).

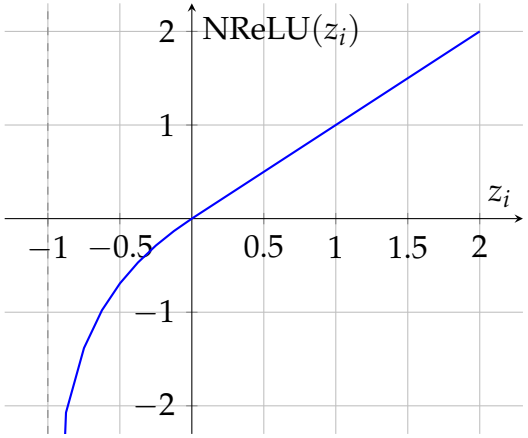


Figura 8.14 – Gráfico da função de ativação *Noisy ReLU* (*NReLU*).

Fonte: O autor (2025).

Ao calcular a derivada da *NReLU* você encontrará um problema que não havia aparecido nas outras funções. O termo de ruído $\mathcal{N}$ é um termo não determísitico, o que significa que mesmo que tivéssemos a mesma entrada para a função *Noisy ReLU* duas ou mais vezes, não poderíamos afirmar com certeza de que essas saídas seriam iguais. Para resolver esses problemas, os autores consideram para a *NReLU* que sua função para o *backward pass* será irá retornar zero quando o valor de entrada for negativo ou nulo, e irá retornar um, quando o valor de entrada for positivo (NAIR; HINTON, 2010). Então a expressão que representa a *NReLU* para a retropropagação é dada pela Equação 8.19, note que ela é a mesma expressão da derivada da *ReLU* tradicional.

Derivada Noisy ReLU (NReLU)

$$\frac{d}{dz_i}[\text{NReLU}](z_i) = \begin{cases} 0 & \text{se } z_i \leq 0 \\ 1 & \text{se } z_i > 0 \end{cases} \tag{8.19}$$

Se a função da *Noisy ReLU* no *backward pass* será a mesma que a derivada da *ReLU*, então é possível utilizar como base o gráfico da derivada da *ReLU*. Para isso, tem-se

então a Figura 8.15.

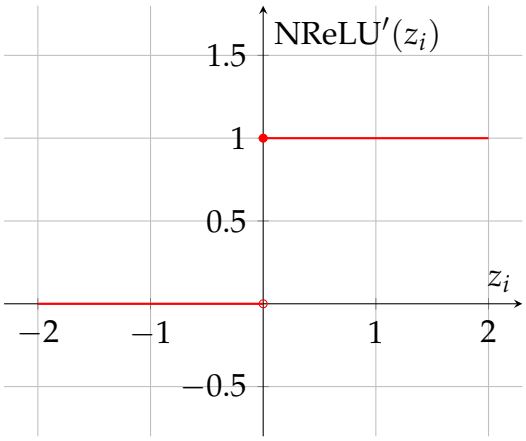


Figura 8.15 – Gráfico da função *Backward Pass* da função de ativação *Noisy ReLU*.

Fonte: O autor (2025).

No trabalho *Rectified Linear Units Improve Restricted Boltzmann Machines*, Nair e Hinton (2010) exploram o desempenho da *NReLU* e *ReLU* utilizando o *dataset NORB*, que é um *dataset* para o reconhecimento de objetos 3D sintéticos que contém cinco classes: humanos, animais, carros, aviões e caminhões. No texto, os autores utilizam a versão *Jittered-Cluttered NORB*, uma variante que tem imagens estereoscópicas em tons de cinza com fundo desorganizado e um objeto central que é aleatoriamente instável em posição, tamanho, intensidade de pixels (NAIR; HINTON, 2010). O desempenho dessas funções nesse *dataset* pode ser visto pela Tabela 8.10 e pela Tabela 8.11.

A Tabela 8.10, mostra a taxa de erro dos diferentes modelos no *dataset*, para isso, são construídos modelos com 4000 unidades ocultas treinados com imagens de dimensões 32x32x2. Note ao analisar a tabela que é nítido que os modelos que são pré-treinados utilizando máquinas de Boltzmann restritas apresentam um melhor desempenho de forma geral, da mesma forma que a *NReLU* oferece uma perda menor quando comparada com a função binária em ambos os casos: pré-treinado ou não.

Mas, vale a pena fazer uma comparação ainda mais interessante, o modelo que faz uso da *Noisy ReLU* que não foi pré-treinado apresenta um resultado com uma diferença de 0.9 pontos quando comparado com o modelo treinado que faz uso da função binária. Isso é interessante porque indica que é possível conseguir um resultado muito melhor utilizando a *Noisy ReLU* mediante a função binária mesmo quando não tiver condições de pré-treinar uma rede.

Seguindo adiante, na Tabela 8.11 também é possível analisar as taxas de erro dos classicadores, mas, neste caso, eles possuem duas camadas ao invés de somente uma como mostrado da comparação da Tabela 8.10, assim a primeira camada é composta de 4000 unidades ocultas (assim como no primeiro caso), enquanto a segunda camada possui 2000 unidades ocultas.

Tabela 8.10 – Taxa de Erro de Classificadores no Dataset NORB Jittered-Cluttered

Pré-treinado?	NReLU (%)	Binary (%)
Não	17.8	23.0
Sim	**16.5**	**18.7**

Nota: Taxas de erro no conjunto de teste para classificadores com 4000 unidades ocultas. Os valores em negrito indicam a menor taxa de erro (melhor resultado) em cada coluna. Os modelos foram treinados com imagens de 32x32x2 do dataset NORB Jittered-Cluttered. A coluna "NReLU"refere-se a unidades de ativação ReLU com ruído (Noisy ReLU), enquanto "Binary"refere-se a unidades binárias tradicionais. A coluna "Pré-treinado?"indica se o modelo utilizou uma Máquina de Boltzmann Restrita para pré-treinamento. Fonte: Adaptado de "Rectified Linear Units Improve Restricted Boltzmann Machines", por V. Nair & G. E. Hinton, 2010, *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*.

Com base essa tabela, é possível notar que o desempenho dos modelos que fazem uso da *Noisy ReLU* melhorou tanto nos casos em que as camadas não foram pré-treinadas, quanto nos casos em que uma ou ambas foram, quando se compara com os resultados da Tabela 8.10. Um ponto interessante disso é que no caso em que somente uma das camadas foi pré-treinada no modelo que usa a *NReLU* o seu resultado foi igual ao do primeiro caso onde existia somente uma camada, o que pode indicar que talvez não seja vantajoso adicionar mais camadas em uma rede caso não esteja disposto a pré-treiná-las. Note também que esse cenário com a unidade binária foi ainda pior, mostrando que não existe um grande ganho em adicionar mais camadas em uma rede que faz uso dessa função.

Esse resultado prová-se ainda mais nítido quando é analisado o caso em que ambas as camadas foram pré-treinadas, na unidade binária não houve nenhuma diminuição na sua perda, ela inclusive é pior que a do modelo que faz uso de apenas uma camada. Já quando é observada a *Noisy ReLU* e seus resultados, nota-se um cenário bem diferente, os modelos que fazem uso dela apresentam um desempenho melhor quando são adicionadas mais camadas na rede, fazendo com que a perda da rede diminua, indicando que essa função permite a criação de redes mais profundas e com isso, desempenhos melhores possam ser alcançados.

Esse tópico de permitir a criação de redes mais profundas, que aconteceu justa-

Tabela 8.11 – Taxas de Erro de Classificadores no Dataset Jittered-Cluttered NORB

Camadas Pré-treinadas		Taxa de Erro de Teste (%)	
Camada 1	**Camada 2**	**NReLU**	**Binary**
Não	Não	17.6	23.6
Sim	Não	16.5	**18.8**
Sim	Sim	**15.2**	**18.8**

Nota: Taxas de erro (%) de teste para classificadores com duas camadas ocultas (4000 unidades na primeira, 2000 na segunda), treinados em imagens do dataset Jittered-Cluttered NORB de 32x32x2. A tabela compara modelos com unidades retificadoras ruidosas (NReLU) e unidades binárias tradicionais (Binary), avaliando o impacto do pré-treinamento de cada camada. Os valores em negrito indicam os melhores resultados para cada tipo de unidade. Fonte: Adaptado de "Rectified Linear Units Improve Restricted Boltzmann Machines", por V. Nair & G. E. Hinton, 2010, *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 807-814.

mente ao optar por funções retificadoras como a *ReLU* e a *Noisy ReLU* ao invés de funções sigmoidais, está intrinsicamente relacionado ao problema da desaparecimento do gradiente. Em redes que faziam uso de funções sigmoidais, os programadores e pesquisadores estavam constantemente correndo o risco de que ao adicionar mais camadas a fim de que essa rede pudesse alcançar melhores métricas, o problema do desaparecimento do gradiente viesse a tona. Isso acontece, porque ao adicionar mais camadas, existe uma maior chance de que esse vetor seja mais uma vez multiplicado por valores pequenos e com isso diminuísse o seu valor.

Ainda em *Rectified Linear Units Improve Restricted Boltzmann Machines*, Nair e Hinton (2010) citam que uma das propriedades interessantes da *NReLU* é a *intensity equivarience* (equivariância de intensidade), a qual é bem útil para o reconhecimento de objetos. No texto, os autores destacam que um dos principais objetivos ao contruir um sistema que faça o reconhecimento de objetos, é garantir que a saída seja invariante às propriedades da sua entrada, como localização, escala, iluminação e orientação, e a NReLU é uma das funções que quando adicionada em uma rede neural, garante que isso possa ser atingido (NAIR; HINTON, 2010).

8.6 O Problema dos Gradientes Explosivos

Anteriormente, ao conhecer as sigmoidais no Capítulo 7, foi possível ver que elas eram comumente utilizadas como as funções de ativação padrão de uma rede neu-

ral antes das retificadoras. Mas elas possuíam um problema, o do desaparacimento do gradiente. Esse problema acontecia porque essas funções retornavam sempre números muito pequenos em suas derivadas, que consequentemente eram multiplicadas no *backward pass* com o gradiente retropropagado gerando como produto um número pequeno, esse número era então novamente multiplicado por outra constante de baixo valor e por aí vai, como resultado, o gradiente retropropagado que chegava nas primeiras camadas para atualizar os pesos e vieses da rede possuía um valor tão pequeno que muitas vezes não resultava em uma atualização capaz de gerar impacto no aprendizado da rede. Assim, tínhamos o problema do desaparecimento do gradiente.

Já neste capítulo, foram conhecidas as funções retificadoras, e como elas surgiram como uma alternativa para contornar esse problema. Contudo, elas também apresentam problemas, sendo um delos o dos gradientes explosivos, o qual será explicado nessa seção.

Para explicar melhor essa condição será utilizado um exemplo como base.

Como foi visto no Capítulo 6, o gradiente retropropagado para camadas anteriores de uma rede neural é proporcional a multiplicação da perda, com a derivada da função de ativação no ponto e o valor do resultado da camada anterior de neurônios.

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

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.

Considerando isso, imagine que temos uma rede composta por quatro camadas densas e cada camada tem apenas um neurônio com pesos iguais a 1. Dessa forma, é possível simplificar a fórmula vista para a Equação 8.20.

$$\delta^{(L)} = \delta^{(L+1)} \times \sigma'(x^{(L)}) \tag{8.20}$$

Considere também que as camadas da rede possuem a seguinte configuração:

- ReLU da primeira camada: tem como resultado da derivada $\text{SELU}(z_i) = 1.5$
- ReLU da camada densa 2: tem como resultado da derivada $\text{SELU}(z_i) = 1.4$
- ReLU da camada densa 3: tem como resultado da derivada $\text{SELU}(z_i) = 1.45$
- ReLU da camada de saída: tem como resultado da derivada $\text{SELU}(z_i) = 1.5$

Além disso, você sabe também que o gradiente inicial na camada de saída está sendo de 25. Com isso é possível calcular o gradiente retropropagado para a primeira camada, comecando pela terceira, já que já temos o valor do gradiente para a camada de saída, dessa forma temos que:

$$\left.\begin{aligned} \delta^{(3)} &= \delta^{(4)} \times \text{ReLU}'(x^{(3)}) \\ \delta^{(3)} &= 25 \times 1.45 = 36.25 \end{aligned}\right\rangle \textit{Subtituindo os valores}$$

Seguindo adiante, é possível fazer o mesmo procedimento para encontrar $\delta^{(2)}$ agora já tendo $\delta^{(3)}$, dessa forma:

$$\left.\begin{aligned} \delta^{(2)} &= \delta^{(3)} \times \text{ReLU}'(x^{(2)}) \\ \delta^{(2)} &= 36.25 \times 1.4 = 50.75 \end{aligned}\right\rangle \textit{Subtituindo os valores}$$

De forma semelhante, é finalmente possível encontrar o gradiente retropropagado para a primeira camada:

$$\left.\begin{aligned} \delta^{(1)} &= \delta^{(2)} \times \text{ReLU}'(x^{(1)}) \\ \delta^{(1)} &= 50.125 \times 1 = 76.125 \end{aligned}\right\rangle \textit{Subtituindo os valores}$$

Perceba que o gradiente que antes era de 25, mais que triplicou, passando para 76.125. Caso você tivesse uma rede neural mais profunda, com 10 ou mais camadas, por exemplo, e todas essas camadas fizessem uso da *SELU*, existe uma chance de que o gradiente que chegasse para as primeiras camadas tivesse um valor muito alto. Esse valor muito elevado pode afetar diretamente como os pesos e vieses da rede são atualizados, impedindo que a rede aprenda nas primeiras camadas. E como as primeiras camadas geralmente são responsáveis por aprender características mais básicas do problema, todo o aprendizado da rede sofre com isso.

Esse é o problema do gradiente explosivo, e pode ser definido como:

> **Definição:** Quando o erro é retropropagado por uma rede neural, ele pode aumentar exponencialmente de camada para camada. Nesses casos, o gradiente em relação aos parâmetros em camadas inferiores pode ser exponencialmente maior do que o gradiente em relação aos parâmetros em camadas superiores. Isso torna a rede difícil de treinar se ela for suficientemente profunda. Tendo então o problema do **gradiente explosivo** (PHILIPP; SONG; CARBONELL, 2018).

Assim, mesmo as retificadoras corrigindo o problema do desaparecimento do gradiente, ela acabou por introduzir uma nova categoria de problemas para uma rede neural. Acontecimentos assim são comuns, muitas vezes queremos concertar algo mas acabamos por atrapalhar outra parte de um projeto de rede neural, por isso, devemos escolher com calma quais funções serão utilizadas além de realizar testes para garantir uma melhor performance do modelo que está sendo criado.

Perceba também que se você tiver uma função como a *ReLU*, em que o maior valor retornado por sua derivada é 1, ainda sim isso pode ser um problema. Pois, caso o gradiente que é calculado para a última camada seja muito alto, ele vai voltar para as primeiras camadas também com o mesmo valor, considerando que todas as funções *ReLU* retornem 1 em suas derivadas. Dessa forma, ainda sim, há um problema em como o gradiente é propagado pela rede.

Conhecidas todas essas diferentes funções de ativação retificadoras, começando pela *ReLU*, com as suas propriedades e como ela foi importante para o desenvolvimento de redes neurais mais profundas. E, terminando explicando o problema do gradiente explosivo, cabe então sumarizar o conteúdo visto. Para isso, é possível ver esse resumo na próxima seção.

8.7 Comparativo: Funções Retificadoras

Por fim, visto todas essas funções, é possível compilá-las na Tabela 8.12. A qual é responsável por destacar a principal característica de cada uma das funções retificadoras, bem como suas vantagens e desvantagens de forma resumida.

Tabela 8.12 – Comparativo das funções de ativação retificadoras

Função	Principal característica	Vantagem	Desvantagem
Rectfied Linear Unit (*ReLU*)	Retorna z_i quando $z_i > 0$, caso contrário, retorna zero.	É a função padrão para ser utilizada em uma rede *feedforwad*.	Sobre do problema dos *ReLUs* agonizantes.
Leaky ReLU (*LReLU*)	Variante da *ReLU* que apresenta um coeficiente α, permitindo um pequeno "vazamento"de gradiente em situações que $z_i < 0$	Consegue mitigar o problema dos *ReLUs* agonizantes.	Adiciona mais um hiperparâmetro que precisa ser ajustado manualmente.
Parametric ReLU (*PReLU*)	Variante da *Leaky ReLU* em que α é um parâmetro aprendível	Por possuir um coeficiente aprendível, a *PReLU* apresenta uma tendência maior de se adptar aos dados.	Apresenta mais parâmetros, aumentando o grau de complexidade da rede neural.
Randomized Leaky ReLU (*RReLU*)	Versão da *Leaky ReLU* em que α é dado por um número aleatório dado pela distribuição normal da forma $U(l, u)$	Adiciona maior aleatoriedade para a rede, ajudando a combater o sobreajuste.	Por possuir uma natureza aleatória pode deixar o treinamento menos determinístico.
Exponential Linear Unit (*ELU*)	Versão exponencial que busca imitar o comportamento da *ReLU*	É uma função suave e derivável em todos os seus pontos.	Apresenta exponenciais em sua fórmula, sendo mais "cara"em termos de custo computacional.
Scale Exponential Linear Unit (*SELU*)	Versão escalada da *ELU*.	Possui propriedades auto-normalizadoras, permitindo criar as *SNNs*	Assim como a *ELU*, a *SELU* apresenta exponenciais em sua composição, sendo mais "cara"que outras funções dessa tabela.
Noisy ReLU (*NReLU*)	Versão da *ReLU* que adiciona ruído Gaussiano em sua fórmula.	Apresenta *intensity equivariance*, ajudando a reconhecer padrões em diferentes situações.	Não pode ser derivada, sendo usada para o *backward pass* a derivada da *ReLU*.

9 Funções de Ativação Modernas e Outras Funções de Ativação

9.1 Funções Modernas: O Estado-da-Arte das Funções de Ativação

9.1.1 Gaussian Error Linear Unit (GELU)

Agora chegamos na última função que iremos ver neste texto, ela é a Gaussian Error Linear Unit, ou GELU. Ela é uma das mais diferentes quando nós passamos a analisar como é sua fórmula, a qual veremos mais em frente. A GELU foi introduzida no artigo *Gaussian Error Linear Units (GELUs)* dos autores Hendrycks e Gimpel (2023), nele, os pesquisadores apresentam uma nova função de ativação de alta performance para ser utilizada na construção de redes neurais.

No texto, os autores, avaliam a GELU e outras variantes como a Exponential Linear Unit e a ReLU tradicional no dataset MNIST, o qual apresenta 10 classes de imagens em escala de cinza sendo 60.000 imagens de treino e 10.000 de teste, os resultados de como a perda e robustez a ruído dessas funções se comporta o longo do experimento podem ser vistos nas figuras **??** e **??** respectivamente (HENDRYCKS; GIMPEL, 2023). Além disso, Hendrycks e Gimpel (2023), avaliaram essas funções em outros conjuntos como o MNIST autoencoding, Tweet part-of-speech tagging, TIMIT frame recognition além dos datasets de imagens CIFAR-10/100.

Analisando a figura **??**, podemos compreender como essas diferentes funções contribuem para a diminuição da perda no modelo criado para a classificação do dataset MNIST. Nos vemos que a GELU é a função que apresenta a menor perda, mas não somente isso, ela é a que contribui para que ela diminua mais rapidamente. Nos casos em que foi utilizado o dropout nas camadas os resultados são ainda mais mais significativos, com a ReLU apresentando a maior perda dentre as três funções, a ELU em segundo, e GELU com uma diferença considerável em relação as suas concorrentes. Isso pode nos indicar que em redes neurais cujo técnicas como o dropout de neurônios nas camadas não seja uma prática viável, pode ser interessante utilizar como estratégia a GELU como função de ativação, pois mesmo sem o dropout, ela consegue manter um bom valor para a perda do modelo que está sendo desenvolvido.

Já na figura **??**, podemos ver como a acurácia dos modelos se comporta quando é adicionado é adicionado ruído as dados de teste. Para isso, nós notamos que os todos os modelos possuem a mesma tendência de decrescer a sua acurácia ao longo do

aumento do ruído, vemos que o modelo que é mais afetado com essa transformação é o que faz uso da exponential linear unit, enquanto os que fazem uso da ReLU e da GELU, mesmo encontrando grandes dificuldades para identificar coretamente as imagens, conseguem manter uma acurácia de quase 0.1 a mais que a ELU. Ainda na figura **??**, também vemos como a perda no conjunto de testes de comporta ao aumentar o ruído, aqui vemos uma situação diferente, vemos que o modelo que menos se adequou ao que estava analisando foi o que fazia uso da ReLU, pois, encontrou difuculdades em tentar minizar o cálculo da perda, por outro lado temos a GELU, que mesmo aumentando o ruído, conseguiu manter uma diferença considerável quando comparada a essas outras duas funções.

Além disso, como dito anteriormente, os autores também fazem testes comparando a GELU com outras funções de ativação utilizando também o dataset CIFAR-10, o qual vem sendo discutido em seções anteriores deste texto, assim, temos a figura **??**, que nos mostra esse comparativo com a taxa de erro (HENDRYCKS; GIMPEL, 2023). Com base nessa análise, nós podemos concluir que a GELU é a melhor alternativa dentre essas três funções para a rede que foi criada, apresentando a menor taxa de erro, tanto no conjunto de dados de treino quanto no conjunto de dados de teste. Uma observação interessante a ser feita com base neste gráfico é de que esses modelos foram treinados por 200 épocas no total, e como a GELU é uma função bem mais complexa que a ReLU, o tempo de treino do modelo que fez uso dessa função foi provavelmente bem maior, algo que pode ser levado em consideração caso seja necessário criar uma rede que seja treinada mais rapidamente mas que ainda sim tenha uma taxa de erro baixa.

Conhecendo um pouco como a GELU atua em uma rede neural, podemos agora conhecer ela por meio da sua fórmula, a qual é dada pela expressão 9.1, a qual é um tanto diferente das outras expressões que vimos até agora. Note que nós não temos dessa vez uma expressão condicional como na ReLU e suas outras variantes, temos uma expressão única, que pode ser reescrita utilizando outras expressões diferentes. Mais a esquerda, temos o termo $\Phi(z_i)$ que representa o standard Gaussian cumalative distribution function, já na expressão mais a direita, temos o uso da função erro, uma função bem comum de ser utilizada quando estamos trabalhando com conceitos probabilísticos.

Gaussian Error Linear Unit

$$\text{GELU}(z_i) = z_i P(X \leqslant z_i) = z_i \Phi(z_i) = z_i \frac{1}{2}\left[1 + \text{erf}(z_i/\sqrt{2})\right] \tag{9.1}$$

Por apresentar cálculos mais complexos em sua composição, como o uso da função erro para encontrar o satrd gaussian cumative distribution function os autores também apresentam aproximações para a GELU, elas são dadas pelas equações **??** e **??** (HENDRYCKS; GIMPEL, 2023). Essas aproximações facilitam não somente os cálculos mas

também na hora de implementar essa função em Python, garantindo algoritmos mais curtos e fáceis de serem implementados.

Na expressão 9.2 vemos que ela pode ser aproximada utilizando a função tangente como um dos componentes usados, já na expressão 9.3, os autores utilizam como base a Sigmoid Linear Unit (SiLU), a qual é dada pela fórmula $SiLU = x\sigma(x)$ para criar uma expressão que seja capaz de aproximar como a GELU se comporta mas trazer uma maior velocidade de processamento por apresentar cálculos mais simples em sua composição.

Gaussian Error Linear Unit Aproximações

$$\text{GELU}(x) \approx 0.5x \left(1 + \tanh\left[\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right]\right) \tag{9.2}$$

Gaussian Error Linear Unit Aproximação com Sigmoide

$$\text{GELU}(x) = x\sigma(1.702x) \tag{9.3}$$

Se sabemos a sua fórmula, podemos também plotar o seu gráfico, para isso temos a figura 9.1, esse gráfico é uma aproximação, tendo como base a expressão 9.2. Note que ela é uma função assimétrica, que possui o comportamento quase que de uma função identidade para os casos em que a sua entrada é maior que zero, além disso, podemos ver que ela retorna valores não nulos quando a entrada é negativa, mas apenas até um certo ponto, depois ela assume o comportamento de um função constante. O fato dela retornar valores quando alguns valores da entrada são negativos pode acabar contribuindo para que essa função diminua o problema do neurônios agonizantes, além disso, por ser uma variante da ReLU, ela também é capaz de resolver o problema do gradiente em fuga.

Considerando as suas expressões e como a GELU se comporta, podemos também calcular a sua derivada para ser utilizada na retropropagação do modelo. Para isso, temos a expressão 9.4, a qual pode ser expandida em uma equação mais completa, resultando então na fórmula da expressão 9.5.

Derivada Gaussian Error Linear Unit

$$\frac{d}{dz_i}[\text{GELU}](z_i) = \Phi(z_i) + z_i\phi(z_i) \tag{9.4}$$

Derivada Completa Gaussian Error Linear Unit

$$\frac{d}{dx}[\text{GELU}](z_i) = \Phi(z_i) + \frac{z_i}{\sqrt{2\pi}}e^{-\frac{z_i^2}{2}} \tag{9.5}$$

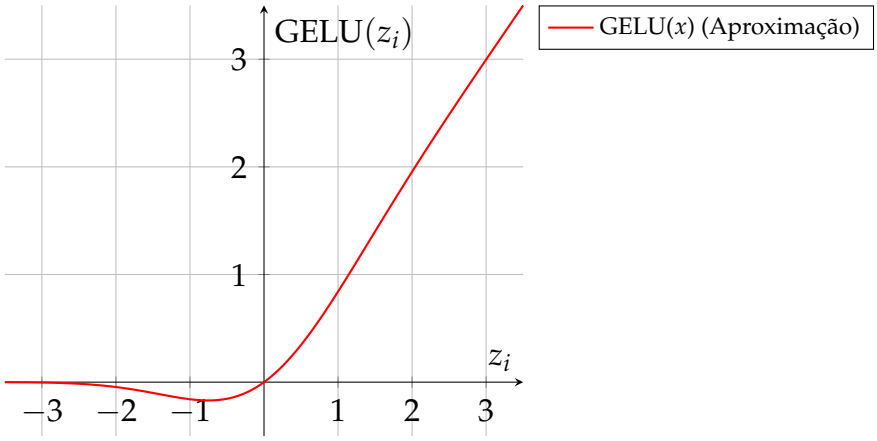


Figura 9.1 – Gráfico da função de ativação Gaussian Error Linear Unit (GELU) usando aproximação.

Fonte: O autor (2025).

Para plotarmos o gráfico de sua derivada podemos fazer uma aproximação utilizando $\phi(x) = 1/(1 + exp(-1.702 * x))$, com base nela, encontramos como resultado a figura **??**. Note que ele também é diferente dos gráficos que estávamos vendo até agora, ele é contínuo em todo o seu domínio, diferente das funções mais simples, com a ReLU e a Leaky ReLU, além de possuir um carater saturante, assim, para valores acima de três o ele irá retornar valores próximos de um, indicando que o não irá colaborar para que o problema do gradiente em fuga ocorra como no caso das funções sigmodais. Além disso, para valores abaixo de -3 a derivada da GELU retorna valores próximos de zero, algo que tem em comum com a ReLU e algumas de suas variantes.

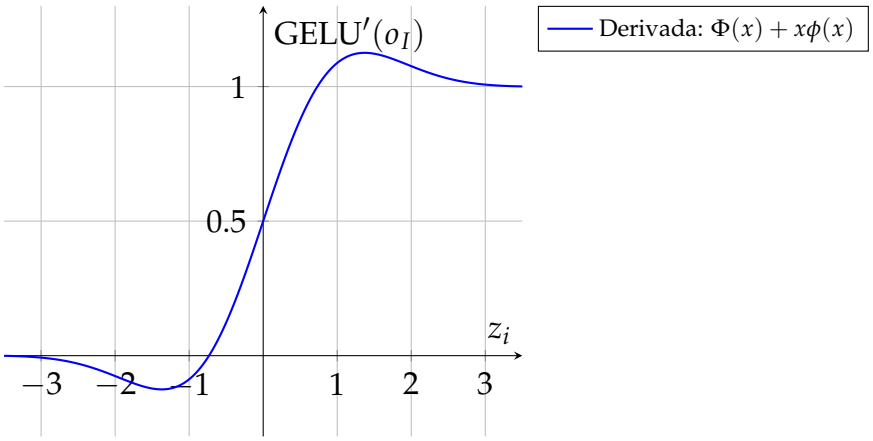


Figura 9.2 – Gráfico da derivada da função de ativação Gaussian Error Linear Unit (GELU).

Fonte: O autor (2025).

Ainda no artigo *Gaussian Error Linear Units (GELUs)*, os autores discutem outras informações úteis da Gaussian Error Linear Unit para serem considerados ao construir uma rede neural com essa função de ativação, o primeiro deles é de que é recomen-

dado o uso de um otimizador com momentum quando estiver treinando uma rede com a GELU (HENDRYCKS; GIMPEL, 2023). Em segundo lugar, Hendrycks e Gimpel (2023), destacam que é importante utilizar uma aproximação próxima da distribuição acumulativa da distribuição gaussiana, entretanto, funções como a sigmoide, são uma aproximação acumulativa da distribuição normal, contudo, a SiLU mesmo performando pior que a GELU, ainda sim, é capaz de performar melhor que outras retificadoras como a ELU e a ReLU nos testes realizados pelos autores, assim, faz necessário o uso de novas aproximações, como as vistas nas expressões 9.2 e 9.3.

9.1.2 Swish

Swish

$$\text{Swish}(z_i) = z_i \cdot \sigma(z_i) = z_i \frac{1}{1 + e^{-z_i}} \tag{9.6}$$

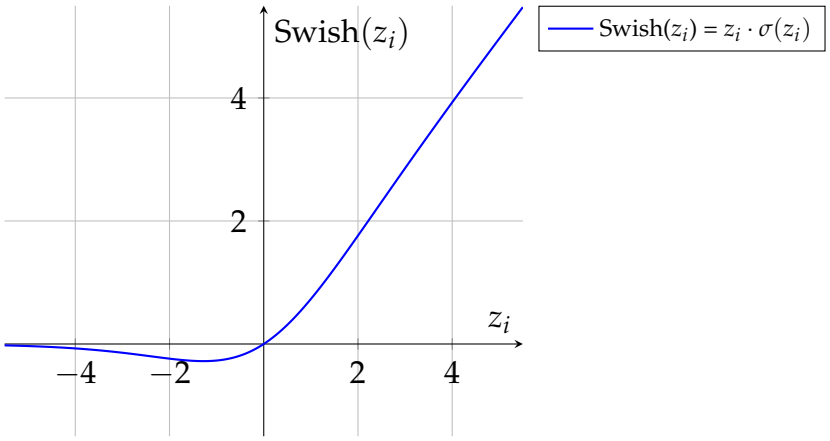


Figura 9.3 – Gráfico da função de ativação Swish.

Fonte: O autor (2025).

Derivada Swish

$$\frac{d}{dz_i}[\text{Swish}](z_i) = \text{Swish}(z_i) + \sigma(z_i)(1 - \text{Swish}(z_i)) \tag{9.7}$$

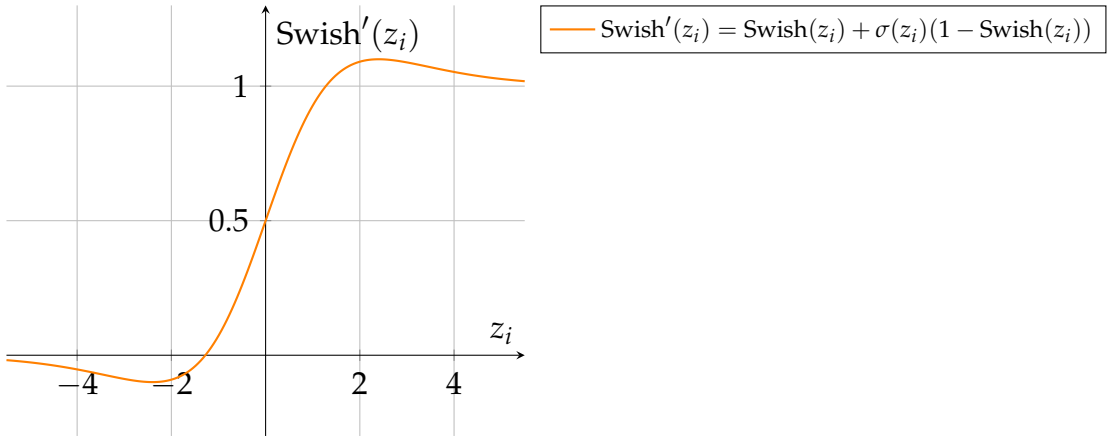


Figura 9.4 – Gráfico da derivada da função de ativação Swish.

Fonte: O autor (2025).

9.2 Funções Para Camadas de Saída

9.2.1 Softmax

Softmax

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{9.8}$$

Derivada Softmax

$$\frac{\partial \text{Softmax}(z_i)}{\partial z_j} = \begin{cases} \text{Softmax}(z_i)(1 - \text{Softmax}(z_i)) & \text{se } i = j \\ -\text{Softmax}(z_i)\text{Softmax}(z_j) & \text{se } i \neq j \end{cases} \tag{9.9}$$

10 Funções de Perda

Até agora foi visto o funcionamento da retropropagação, e como ela faz uso dos otimizadores, os quais funcionam como um barco, percorrendo a função de perda em busca de pontos de mínimos. Além disso, em seguida foram vistas diversas funções de ativação, começando pelas sigmoidais, depois pelas retificadoras, e por fim uma coletânea de diferentes funções. Contudo, está na hora de entender o outro lado da retropropagação: as funções de perda.

Para isso, esse capítulo busca explicar diversas funções de perda e suas aplicações, começando pelas funções para problemas de regressão, conhecendo as clássicas erro quadrático médio e erro absoluto médio, além da *hubber loss*, uma função que busca unir o melhor dessas duas funções de perda. Seguindo adiante, são introduzidas as funções de perda para classificação binária, como a *BCE*. Visto os problemas de classificação binária, é possível também conhecer os problemas de classifação multi com a *categorical cross entropy*.

Mais adiante, está apresentado não funções, mas esquemas de como a perda pode ser medida para problemas como o de redes adversárias. Mas as perdas não são a única forma de medir como um modelo está performando, para isso, o final do capítulo é dedicado para explicar outros diferentes métodos de medir o desempenho do modelo que está sendo construído.

10.1 A Intuição da Perda: Medindo o Erro do Modelo

10.2 Funções de Perda Para Regressão

10.2.1 Exemplo Ilustrativo: Jogando Dardos

Pense que você está jogando dardos com seus amigos e quer decidir quem está com mais pontos. Mas você não está satisfeito em considerar as marcações que estão no jogo, e decidiu inovar. Para isso, você pegou uma régua e passou a medir a distância que os dardos que você e seus amigos haviam jogado no centro. Quem chegasse mais próximo do centro, ganhava o jogo.

Essa ideia de medir o quão próximo você está do resultado desejado utilizando a disância entre esses dois pontos como parâmetro, é justamente o motivador pela criação das funções de perda para regressão. Para isso, elas utilizam diferentes fórmulas, com todas com o mesmo intuito, medir a distância em que o "chute"dado pelo modelo está do ponto real (desejado). Assim, é possível começar a ver essas funções pela *MSE*,

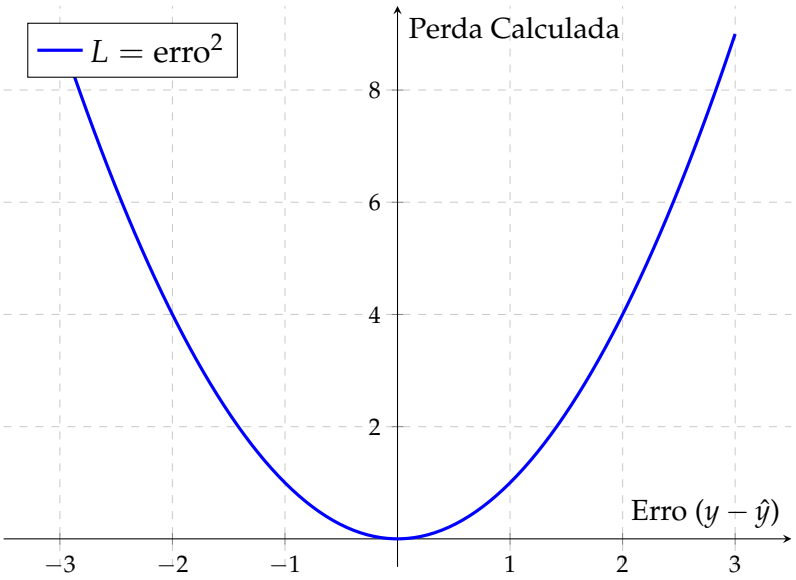
a qual é responsável por médir o erro quadrático médio.

10.2.2 Erro Quadrático Médio (Mean Squared Error - MSE)

Erro Quadrático Médio (MSE)

$$L_{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{10.1}$$

Erro Quadrático Médio (MSE)



Derivada do MSE

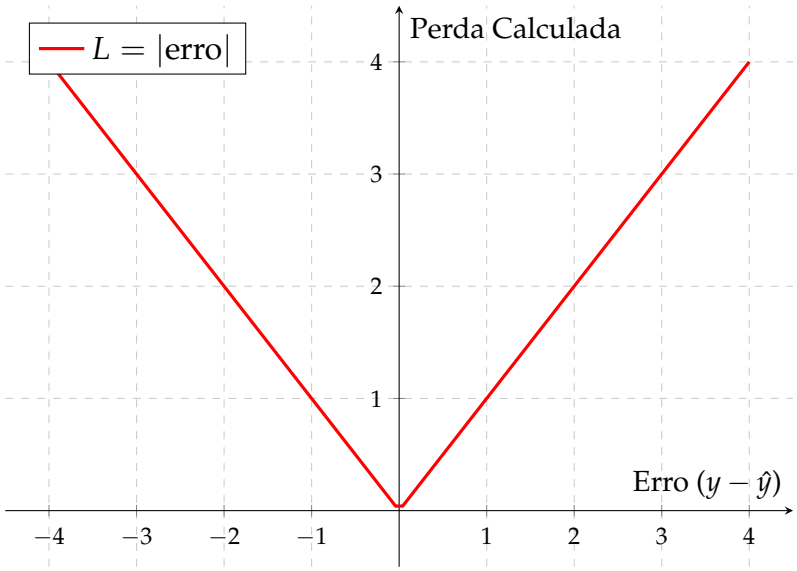
$$\frac{\partial L_{\text{MSE}}}{\partial \hat{y}_i} = \frac{2}{N}(\hat{y}_i - y_i) \tag{10.2}$$

10.2.3 Erro Absoluto Médio (Mean Absolute Error - MAE)

Erro Absoluto Médio (MAE)

$$L_{\text{MAE}} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \tag{10.3}$$

Função de Perda: Erro Absoluto Médio (MAE)



Derivada do MAE

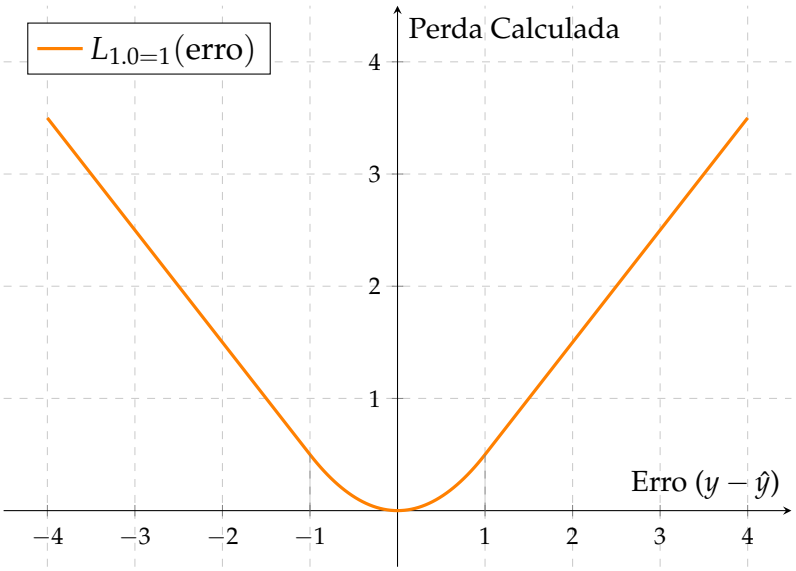
$$\frac{\partial L_{\text{MAE}}}{\partial \hat{y}_i} = \frac{1}{N} \cdot \text{sgn}(\hat{y}_i - y_i) = \begin{cases} +\frac{1}{N} & \text{se } \hat{y}_i > y_i \\ -\frac{1}{N} & \text{se } \hat{y}_i < y_i \\ 0 & \text{se } \hat{y}_i = y_i \end{cases} \tag{10.4}$$

10.2.4 Huber Loss: O Melhor de Dois Mundos

Huber Loss

$$L_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{para } |y - \hat{y}| \leq \delta \\ \delta(|y - \hat{y}| - \frac{1}{2}\delta) & \text{caso contrário} \end{cases} \tag{10.5}$$

Função de Perda: Huber Loss ($1.0 = 1$)



Derivada da Huber Loss

$$\frac{\partial L_\delta}{\partial \hat{y}} = \begin{cases} \hat{y} - y & \text{para } |\hat{y} - y| \leq \delta \\ \delta \cdot \text{sgn}(\hat{y} - y) & \text{caso contrário} \end{cases} \tag{10.6}$$

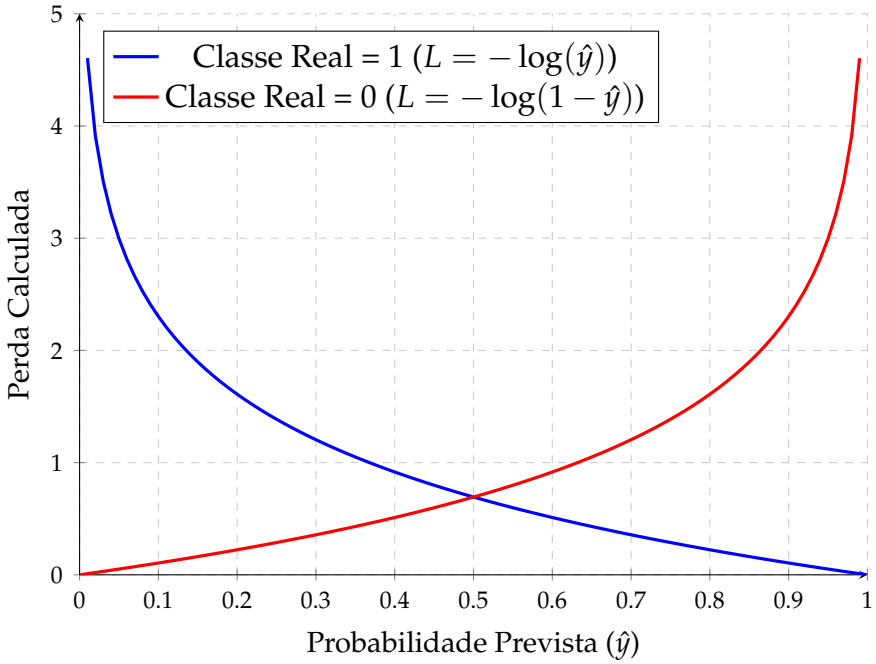
10.3 Funções de Perda para Classificação Binária

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

Entropia Cruzada Binária

$$L(y, \hat{y}) = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})] \tag{10.7}$$

Função de Perda: Entropia Cruzada Binária



Derivada da Entropia Cruzada Binária

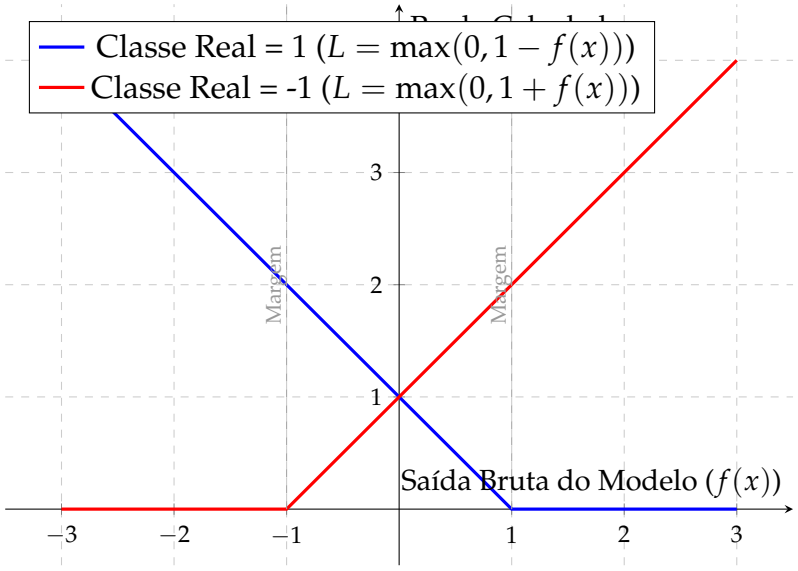
$$\frac{\partial L}{\partial \hat{y}} = \frac{\hat{y} - y}{\hat{y}(1-\hat{y})} \tag{10.8}$$

10.3.2 Perda Hinge (Hinge Loss)

Hinge Loss

$$L(y, f(x)) = \max(0, 1 - y \cdot f(x)) \tag{10.9}$$

Função de Perda: Hinge Loss



Derivada da Hinge Loss

$$\frac{\partial L}{\partial f(x)} = \begin{cases} -y & \text{se } y \cdot f(x) < 1 \\ 0 & \text{se } y \cdot f(x) \geq 1 \end{cases} \tag{10.10}$$

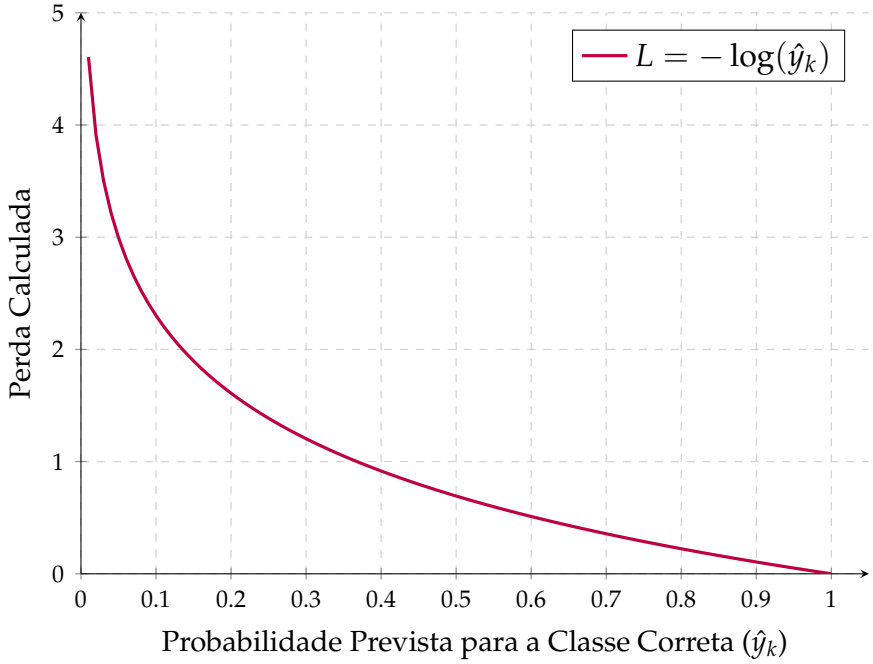
10.4 Funções de Perda para Classificação Multilabel

10.4.1 Entropia Cruzada Categórica (Categorical Cross-Entropy)

Entropia Cruzada Categórica

$$L(y, \hat{y}) = -\sum_{c=1}^{C} y_c \log(\hat{y}_c) \tag{10.11}$$

Função de Perda: Entropia Cruzada Categórica



Derivada da Entropia Cruzada Categórica

$$\frac{\partial L}{\partial z_i} = \hat{y}_i - y_i \tag{10.12}$$

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

Entropia Cruzada Categórica Esparsa

$$L_i = -\log(\hat{y}_{i,y_i}) \tag{10.13}$$

Derivada da Entropia Cruzada Categórica Esparsa

$$\frac{\partial L_i}{\partial z_{i,k}} = \hat{y}_{i,k} - y_{i,k} \tag{10.14}$$

10.5 Funções de Perda Avançadas

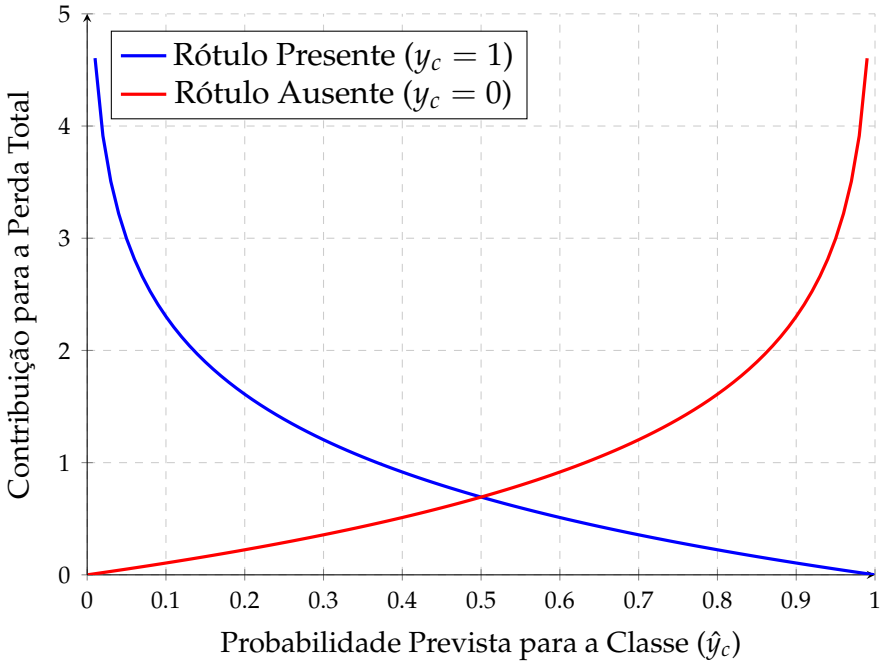
10.5.1 Perda para Classificação Multirrótulo (Multilabel)

Perda para Classificação Multirrótulo

$$\mathcal{L}_{\text{multilabel}} = -\frac{1}{C}\sum_{c=1}^{C}[y_c \log(\hat{y}_c) + (1 - y_c)\log(1 - \hat{y}_c)] \tag{10.15}$$

(Contribuição de uma única classe)

Função de Perda Multirrótulo



Derivada da Perda Multirrótulo

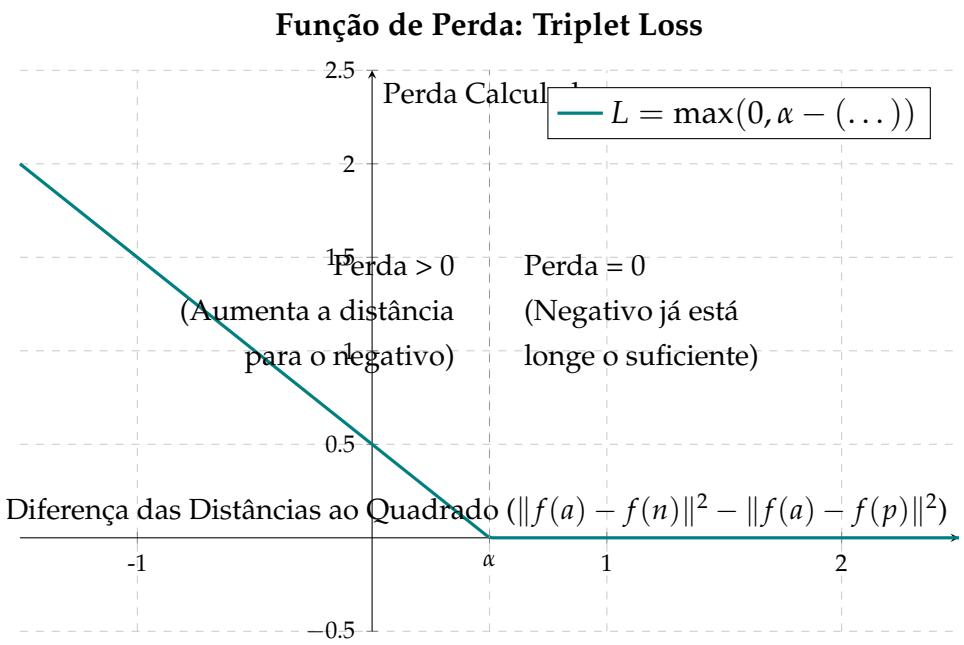
$$\frac{\partial \mathcal{L}_{\text{multilabel}}}{\partial z_c} = \frac{1}{C}(\hat{y}_c - y_c) \tag{10.16}$$

10.5.2 Perdas para Ranking e Aprendizado de Métricas

10.5.2.1 Triplet Loss (Perda Tripla)

Triplet Loss

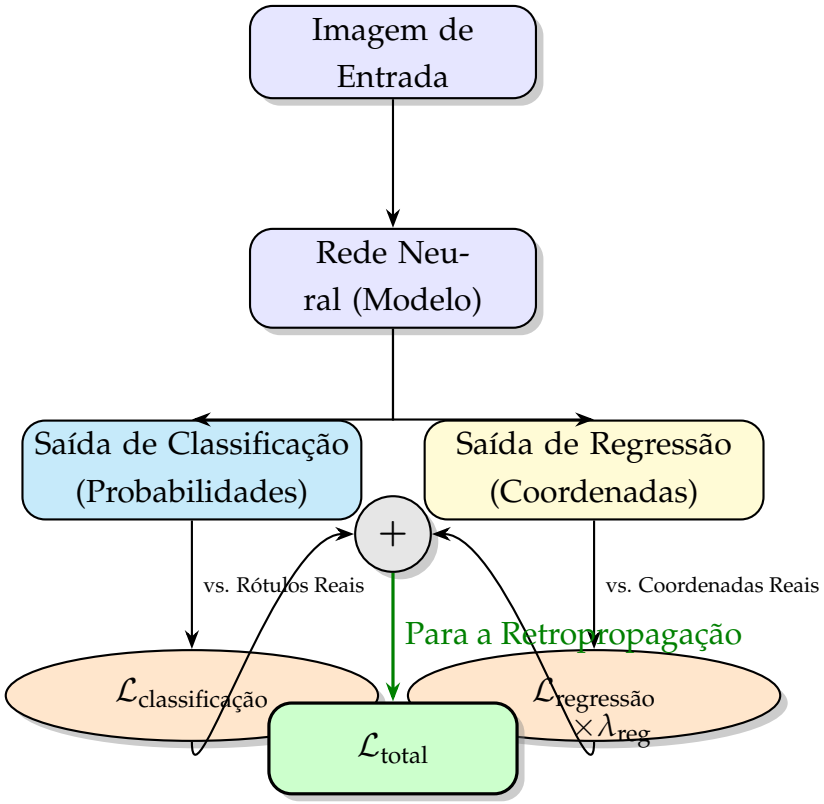
$$\mathcal{L}(a, p, n) = \max\left(\|f(a) - f(p)\|^2 - \|f(a) - f(n)\|^2 + \alpha, 0\right) \tag{10.17}$$



10.5.2.2 Contrastive Loss (Perda Contrastiva)

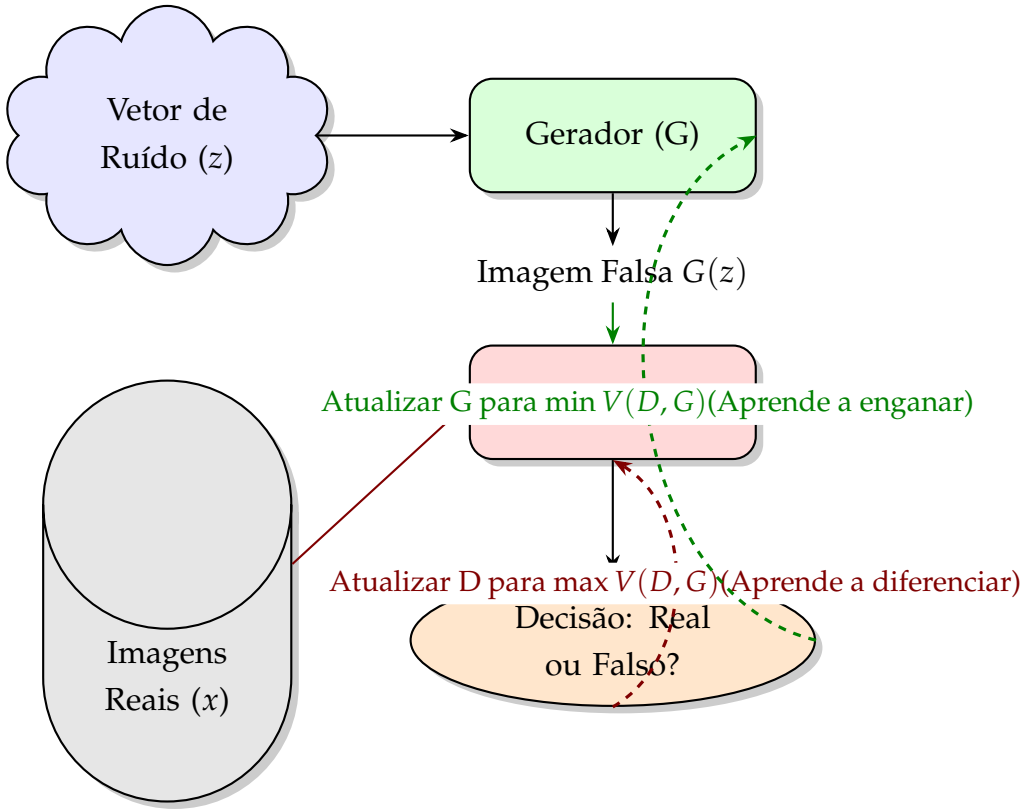
Perda Composta (Conceitual)

$$\mathcal{L}_{\text{total}} = \lambda_{\text{reg}}\mathcal{L}_{\text{regressão}} + \mathcal{L}_{\text{classificação}} \tag{10.18}$$



Função de Valor de uma GAN (Minimax)

$$\min_G \max_D V(D,G) = \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))] \quad (10.19)$$



10.6 Outras Métricas de Avaliação

10.6.1 Acurácia

Acurácia

$$\text{Acurácia} = \frac{VP + VN}{VP + VN + FP + FN} \tag{10.20}$$

10.6.2 Precisão

Precisão (Precision)

$$\text{Precisão} = \frac{VP}{VP + FP} \tag{10.21}$$

10.6.3 Revocação ou Sensibilidade

Revocação (Recall) ou Sensibilidade

$$\text{Revocação} = \frac{VP}{VP + FN} \tag{10.22}$$

10.6.4 F1-Score

F1-Score

$$\text{F1-Score} = 2 \times \frac{\text{Precisão} \times \text{Revocação}}{\text{Precisão} + \text{Revocação}} \tag{10.23}$$

10.6.5 Curva ROC e AUC

Taxa de Falsos Positivos (FPR)

$$\text{FPR} = \frac{FP}{FP + VN} \tag{10.24}$$

10.6.6 Métricas Para Regressão (R^2)

Raiz do Erro Quadrático Médio (RMSE)

$$L_{\text{RMSE}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2} \tag{10.25}$$

Coeficiente de Determinação (R^2)

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2} \tag{10.26}$$

11 Metaheurísticas: Otimizando Redes Neurais Sem o Gradiente

O texto do seu capítulo começa aqui...

11.1 Algoritmos Evolutivos

11.2 Inteligência de Enxame

Parte IV

Aprendizado de Máquina Clássico

12 Técnicas de Regressão

13 Árvores de Decisão e Florestas Aleatórias

14 Máquinas de Vetores de Suporte

14.1 Exemplo Ilustrativo

15 Ensamble

15.1 Exemplo Ilustrativo

16 Dimensionalidade

17 Clusterização

Parte V

Redes Neurais Profundas (DNNs)

18 Perceptrons MLP - Redes Neurais Artificiais

O texto do seu capítulo começa aqui...

19 Redes FeedForward (FFNs)

O texto do seu capítulo começa aqui...

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

O texto do seu capítulo começa aqui...

21 Redes Neurais Convolucionais (CNN)

21.1 Exemplo Ilustrativo

21.2 Camadas Convolucionais: O Bloco Fundamental para as CNNs

21.2.1 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
```

21.3 Camadas de Poooling: Reduzindo a Dimensionalidade

21.3.1 Max Pooling

21.3.1.1 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

```

21.3.2 Average Pooling

21.3.2.1 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

```

21.3.3 Global Average Pooling

21.3.3.1 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
```

21.4 Camada Flatten: Achatando os Dados

21.4.1 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
```

21.5 Criando uma CNN

21.6 Detecção de Objetos

21.7 Redes Totalmente Convolucionais (FCNs)

21.8 You Only Look Once (YOLO)

21.9 Algumas Arquiteturas de CNNs

21.9.1 LeNet-5

21.9.2 AlexNet

21.9.3 GoogLeNet

21.9.4 VGGNet

21.9.5 ResNet

21.9.6 Xception

21.9.7 SENet

22 Redes Residuais (ResNets)

O texto do seu capítulo começa aqui...

23 Redes Neurais Recorrentes (RNN)

O texto do seu capítulo começa aqui...

23.1 Exemplo Ilustrativo

23.2 Neurônios e Células Recorrentes

23.2.1 Implementação em Python

23.3 Células de Memória

23.3.1 Implementação em Python

23.4 Criando uma RNN

23.5 O Problema da Memória de Curto Prazo

23.5.1 Células LSTM

23.5.2 Conexões Peephole

23.5.3 Células GRU

24 Técnicas para Melhorar o Desempenho de Redes Neurais

25 Transformers

26 Redes Adversárias Generativas (GANs)

O texto do seu capítulo começa aqui...

27 Mixture of Experts (MoE)

O texto do seu capítulo começa aqui...

28 Modelos de Difusão

O texto do seu capítulo começa aqui...

29 Redes Neurais de Grafos (GNNs)

O texto do seu capítulo começa aqui...

Parte VI

Apêndices

A Tabela das Funções de Ativação

Tabela A.1 – Comparativo das famílias de funções de ativação, suas propriedades, vantagens e desvantagens.

Função	Equação e Derivada	Vantagens	Desvantagens
Sigmoide	$\frac{1}{1+e^{-z_i}}$ $\sigma(z_i)(1-\sigma(z_i))$	• Saída no intervalo (0, 1), interpretável como probabilidade. • Função suave e diferenciável.	• Não é centrada em zero. • Sofre com o desvanecimento do gradiente.
Tangente hip.	$\frac{e^{z_i}-e^{-z_i}}{e^{z_i}+e^{-z_i}}$ $1-\tanh^2(z_i)$	• Centrada em zero, acelera a convergência. • Gradiente mais forte que a sigmoide.	• Ainda sofre com o desvanecimento do gradiente.
Softsign	$\frac{z_i}{1+\lvert z_i\rvert}$ $\frac{1}{(1+\lvert z_i\rvert)^2}$	• Computacionalmente eficiente. • Satura mais lentamente que a Tanh.	• Derivada não pode ser expressa em termos da própria função.
Hard Sigmoid	$\begin{cases} 0 & \text{se } z_i < -3 \\ z_i/6 + 0.5 & \text{se } -3 \leq z_i \leq 3 \\ 1 & \text{se } z_i > 3 \end{cases}$ $\begin{cases} 0 & \text{se } z_i < -3 \\ 1/6 & \text{se } -3 < z_i < 3 \\ 0 & \text{se } z_i > 3 \end{cases}$	• Extremamente rápida e eficiente. • Ideal para hardware com poucos recursos.	• Não é suave; pode "matar"gradientes. • É uma aproximação.

(Continua na próxima página)

Tabela A.1 – Continuação

Função	Equação e Derivada	Vantagens	Desvantagens
Hard Tanh	$\begin{cases} -1 & \text{se } z_i < -1 \\ z_i & \text{se } -1 \leq z_i \leq 1 \\ 1 & \text{se } z_i > 1 \end{cases}$ $\begin{cases} 0 & \text{se } z_i < -1 \\ 1 & \text{se } -1 < z_i < 1 \\ 0 & \text{se } z_i > 1 \end{cases}$	• Extremamente rápida e centrada em zero. • Ótima para hardware de baixo consumo.	• Não é suave; derivada nula em grande parte do domínio.
ReLU	$\begin{cases} z_i, & \text{se } z_i > 0 \\ 0, & \text{se } z_i \leq 0 \end{cases}$ $\begin{cases} 1, & \text{se } z_i > 0 \\ 0, & \text{se } z_i < 0 \\ \nexists & \text{se } z_i = 0 \end{cases}$	• Computacionalmente eficiente. • Evita o desvanecimento do gradiente. • Promove esparsidade na rede.	• Não é centrada em zero. • Pode "morrer"(Dying ReLU). • Pode sofrer com a explosão de gradientes.
LReLU	$\begin{cases} z_i, & \text{se } z_i \geq 0 \\ \alpha \cdot z_i, & \text{se } z_i < 0 \end{cases}$ $\begin{cases} 1, & \text{se } z_i > 0 \\ \alpha, & \text{se } z_i < 0 \\ \nexists, & \text{se } z_i = 0 \end{cases}$	• Resolve o problema da "Dying ReLU".	• O valor de α não é aprendido. • Resultados podem ser inconsistentes.
PReLU	$\begin{cases} z_i, & \text{se } z_i \geq 0 \\ \alpha_i \cdot z_i, & \text{se } z_i < 0 \end{cases}$ $\begin{cases} 1, & \text{se } z_i > 0 \\ \alpha_i, & \text{se } z_i < 0 \\ \nexists, & \text{se } z_i = 0 \end{cases}$	• Variação da LReLU onde α é um parâmetro aprendido. • Pode melhorar a performance.	• Risco de sobreajuste (overfitting) se os dados forem poucos.
ELU	$\begin{cases} z_i, & \text{se } z_i \geq 0 \\ \alpha(e^{z_i} - 1), & \text{se } z_i < 0 \end{cases}$ $\begin{cases} 1, & \text{se } z_i > 0 \\ \alpha e^{z_i}, & \text{se } z_i < 0 \end{cases}$	• Saídas com média próxima de zero. • Mais robusta a ruído que LReLU/PReLU.	• Computacionalmente mais custosa (exponencial).

(Continua na próxima página)

Tabela A.1 – Continuação

Função	Equação e Derivada	Vantagens	Desvantagens
SELU	$\lambda \begin{cases} z_i, & \text{se } z_i > 0 \\ \alpha(e^{z_i} - 1), & \text{se } z_i \leq 0 \end{cases}$ $\lambda \begin{cases} 1, & \text{se } z_i > 0 \\ \alpha e^{z_i}, & \text{se } z_i \leq 0 \end{cases}$	• Propriedades de auto-normalização. • Evita gradientes explosivos/desvanecentes em redes muito profundas.	• Requer inicialização de pesos específica (LeCun normal). • Computacionalmente mais custosa.
GELU	$z_i \cdot \Phi(z_i)$ $\Phi(z_i) + z_i \phi(z_i)$	• Suave e diferenciável em todos os pontos. • Performance estado da arte em Transformers (BERT, GPT).	• Computacionalmente mais custosa que ReLU.

Fonte: O autor (2025).

Referências

ANDREW L. MASS, Awni Y. Hannum; NG, Andrew Y. Rectifier Nonlinearities Improve Neural Networks Acustic Models, 2013. Citado nas pp. 103, 104, 106.

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. 28.

CLEVERT, Djork-Arné; UNTERTHINER, Thomas; HOCHREITER, Sepp. *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*. [*S. l.: s. n.*], 2016. arXiv: 1511.07289 [cs.LG]. Disponível em: https://arxiv.org/abs/1511.07289. Citado nas pp. 118, 120, 121.

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. 73.

DATTA, Leonid. *A Survey on Activation Functions and their relation with Xavier and He Normal Initialization*. [*S. l.: s. n.*], 2020. arXiv: 2004.06632 [cs.NE]. Disponível em: https://arxiv.org/abs/2004.06632. Citado nas pp. 74, 75.

DOUGLAS, Scott C.; YU, Jiutian. *Why ReLU Units Sometimes Die: Analysis of Single-Unit Error Backpropagation in Neural Networks*. [*S. l.: s. n.*], 2018. arXiv: 1812.05981 [cs.LG]. Disponível em: https://arxiv.org/abs/1812.05981. Citado na p. 102.

DOZAT, Timothy. Incorporating Nesterov Momentum into Adam. *In:* ICLR 2016 Workshop Track. [*S. l.: s. n.*], 2016. Disponível em: https://openreview.net/forum?id=OM0DEvNMIxAaI-fG_O1-I. Acesso em: 15 set. 2025. Citado nas pp. 61–63.

DUCHI, John; HAZAN, Elad; SINGER, Yoram. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, v. 12, n. 61, p. 2121–2159, 2011. Disponível em: http://jmlr.org/papers/v12/duchi11a.html. Acesso em: 15 set. 2025. Citado nas pp. 51, 53, 54.

ELMAN, Jeffrey L. Finding structure in time. *Cognitive Science*, Wiley Online Library, v. 14, n. 2, p. 179–211, 1990. DOI: 10.1207/s15516709cog1402_1. Citado na p. 78.

FOUNDATION, The; ELLIOTT, David. A better Activation Function for Artificial Neural Networks, dez. 1998. Citado na p. 84.

GÉRON, Aurélien. *Mãos à Obra: Aprendizado de Máquina com Scikit-Learn e TensorFlow*. [*S. l.*]: Alta Books, 2019. Citado nas pp. 46, 47.

GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. *Deep Learning*. [*S. l.*]: MIT Press, 2016. Citado nas pp. 29, 73, 101, 126.

HE, Kaiming *et al. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. [*S. l.: s. n.*], 2015. arXiv: 1502.01852 [cs.CV]. Disponível em: https://arxiv.org/abs/1502.01852. Citado nas pp. 108–110, 112.

HENDRYCKS, Dan; GIMPEL, Kevin. *Gaussian Error Linear Units (GELUs)*. [*S. l.: s. n.*], 2023. arXiv: 1606.08415 [cs.LG]. Disponível em: https://arxiv.org/abs/1606.08415. Citado nas pp. 135, 136, 139.

HOUSEHOLDER, Alston S. A Theory of Steady-State Activity in Nerve-Fiber Networks: I. Definitions and Preliminary Theorems. *The Bulletin of Mathematical Biophysics*, v. 3, n. 2, p. 63–69, 1941. Citado nas pp. 96, 97.

KINGMA, Diederik P.; BA, Jimmy. *Adam: A Method for Stochastic Optimization*. [*S. l.: s. n.*], 2017. arXiv: 1412.6980 [cs.LG]. Disponível em: https://arxiv.org/abs/1412.6980. Acesso em: 15 set. 2025. Citado nas pp. 56–60.

KLAMBAUER, Günter *et al. Self-Normalizing Neural Networks*. [*S. l.: s. n.*], 2017. arXiv: 1706.02515 [cs.LG]. Disponível em: https://arxiv.org/abs/1706.02515. Citado nas pp. 121, 123, 125.

KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. ImageNet Classification with Deep Convolutional Neural Networks. *In:* PEREIRA, F. *et al.* (ed.). *Advances in Neural Information Processing Systems*. [*S. l.*]: Curran Associates, Inc., 2012. v. 25. Disponível em: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf. Citado na p. 99.

LAMBERT, J. H. Mémoire sur quelques propriétés remarquables des quantités transcendantes circulaires et logarithmiques. *In:* BERGGREN, Lennart; BORWEIN, Jonathan M.; BORWEIN, Peter B. (ed.). *Pi: A Source Book*. New York: Springer-Verlag, 2004. Original work published in 1768. p. 129–140. DOI: 10.1007/978-1-4757-4217-6_18. Citado na p. 81.

LECUN, Yann *et al.* Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, v. 86, n. 11, p. 2278–2324, 1998. DOI: 10.1109/5.726791. Citado na p. 81.

LEDERER, Johannes. *Activation Functions in Artificial Neural Networks: A Systematic Overview*. [*S. l.: s. n.*], 2021. arXiv: 2101.09957 [cs.LG]. Disponível em: https://arxiv.org/abs/2101.09957. Citado nas pp. 79, 82, 92.

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. 73.

LIU, Liyuan *et al. On the Variance of the Adaptive Learning Rate and Beyond*. [*S. l.: s. n.*], 2021. arXiv: 1908.03265 [cs.LG]. Disponível em: https://arxiv.org/abs/1908.03265. Acesso em: 15 set. 2025. Citado na p. 66.

LOSHCHILOV, Ilya; HUTTER, Frank. *Decoupled Weight Decay Regularization*. [*S. l.: s. n.*], 2019. arXiv: 1711.05101 [cs.LG]. Disponível em: https://arxiv.org/abs/1711.05101. Acesso em: 15 set. 2025. Citado nas pp. 63–65.

NAIR, Vinod; HINTON, Geoffrey E. Rectified Linear Units Improve Restricted Boltzmann Machines. *In:* PROCEEDINGS of the 27th International Conference on Machine Learning (ICML-10). [*S. l.: s. n.*], 2010. p. 807–814. Citado nas pp. 97, 125, 127, 128, 130.

NESTEROV, Yurii. A method for solving the convex programming problem with convergence rate $O(1/k^2)$. *Proceedings of the USSR Academy of Sciences*, v. 269, p. 543–547, 1983. Disponível em: https://api.semanticscholar.org/CorpusID:145918791. Citado na p. 48.

PHILIPP, George; SONG, Dawn; CARBONELL, Jaime G. *The exploding gradient problem demystified - definition, prevalence, impact, origin, tradeoffs, and solutions*. [*S. l.: s. n.*], 2018. arXiv: 1712.05577 [cs.LG]. Disponível em: https://arxiv.org/abs/1712.05577. Citado na p. 133.

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. 43, 44.

PYTORCH. *Hardsigmoid*. 2025. Disponível em: https://pytorch.org/docs/stable/generated/torch.nn.Hardsigmoid.html. Acesso em: 10 out. 2025. Citado na p. 86.

PYTORCH. *LeakyReLU*. 2025. Disponível em: https://docs.pytorch.org/docs/stable/generated/torch.nn.LeakyReLU.html. Acesso em: 11 out. 2025. Citado na p. 104.

ROBBINS, Herbert; MONRO, Sutton. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, v. 22, n. 3, p. 400–407, 1951. DOI: 10.1214/aoms/1177729586. Citado na p. 45.

ROSENBLATT, F. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, v. 65, n. 6, p. 386–408, 1958. DOI: 10.1037/h0042519. Citado na p. 78.

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, 38, 42–45, 77.

TIELEMAN, Tijmen; HINTON, Geoffrey. *Lecture 6.5—RMSProp: Divide the gradient by a running average of its recent magnitude*. [*S. l.: s. n.*], 2012. COURSERA: Neural Networks for Machine Learning. Disponível em: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. Acesso em: 3 out. 2025. Citado na p. 55.

VERHULST, P. F. Recherches mathématiques sur la loi d'accroissement de la population. *Nouveaux Mémoires de l'Académie Royale des Sciences et Belles-Lettres de Bruxelles*, L'Académie Royale, v. 18, p. 1–42, 1845. Citado na p. 76.

WIDROW, Bernard; HOFF, Marcian E. Adaptive Switching Circuits. *In:* 1960 IRE WESCON Convention Record. [*S. l.*]: IRE, 1960. .4, p. 96–104. Citado na p. 45.

WILSON, Hugh R.; COWAN, Jack D. Excitatory and inhibitory interactions in localized populations of model neurons. *Biophysical Journal*, Biophysical Society, v. 12, n. 1, p. 1–24, 1972. DOI: 10.1016/S0006-3495(72)86068-5. Citado nas pp. 77, 78.

XAVIER GLOROT, Antaine Border; BENGIO, Yoshua. Deep Sparse Rectifier Neural Networks, 2011. Citado na p. 98.

XU, Bing *et al.* Empirical Evaluation of Rectified Activations in Convolutional Network. *arXiv preprint arXiv:1505.00853*, 2015. Citado nas pp. 113, 115, 116.