Luiz Guilherme Morais da Costa Faria

APRENDIZADO DE MÁQUINA

Brasília, DF
3 de outubro de 2025

Luiz Guilherme Morais da Costa Faria

APRENDIZADO DE MÁQUINA

Universidade de Brasília

Orientador: Nome do Orientador/Revisor (se aplicável)

Brasília, DF
3 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

A filosofia de vida do gradiente descendente: 'Pra lá é pra baixo? É. Então eu vou pra lá.'
— A Sabedoria do Algoritmo

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. Agora 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 explicando a retropropagação, e como ela aliada ao método do gradiente permite que resolver o problema de otimizar milhares de parâmetros de uma rede neural, 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 conhecer 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: A Inspiração de Todos

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

Esse trabalho explica bem como aplicar o método do gradiente para se calcular mínimos de funções, mas para facilitar o entendimento do leitor, em seguida está um exemplo ilutrativo 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 incrivel ideia de ir para o menor ponto dessa cadeia de montanhas, pois, no guia que você estava seguindo, falava que lá havia um lago com uma água cristalina, perfeito para tirar fotos.

Para chegar até esse lago, você conta com uma 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.

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

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

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

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

Em que:

- x: representa as coordenadas do próximo ponto;
- x_0: representa as coordernadas do ponto atual;

- ϵ: representa o tamanho do passo, também chamado de taxa de aprendizado;
- $\nabla f(x_0)$: representa o vetor gradiente calculado na posição do ponto atual (x_0) para função que se deseja otimizar.

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

Um ponto a ser destacado nesse metodo é na hora de escolher uma taxa de aprendizado para ser utilizada no método. Uma taxa de aprendizado muito pequena signifca que o passo que o modelo irá dar de um ponto para outro será menor, e com isso implica que ele levará mais passos para encontrar um ponto de mínimo (essa situação pode ser vista na ilustração da esquerda da figura 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 1.

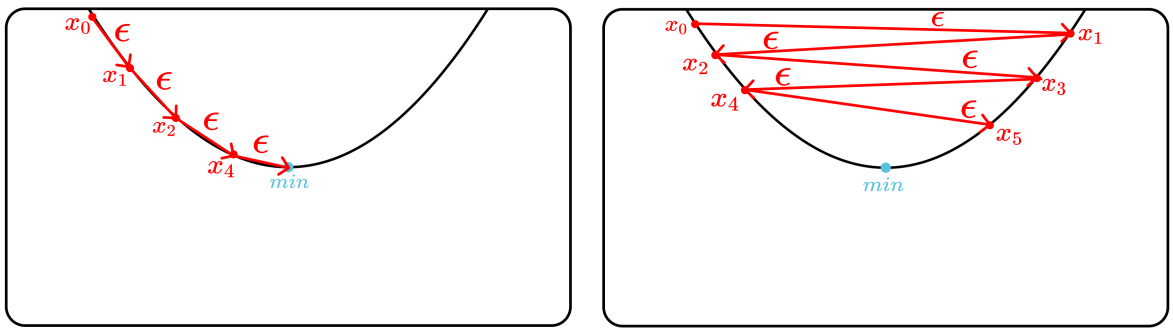


Figura 1 – Comparativo do tamanho de passos em uma função polinomial.

Na prática, escolher o valor da taxa de aprendizado é uma tarefa que irá depender de modelo em modelo, também irá variar com os diferentes métodos de otimização além da topologia da rede neural que está sendo construída. É sempre recomendado

então experimentar diferentes tamanhos de passo, de forma que seja encontrado um que melhor se ajusta ao cenário que está sendo trabalhado.

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

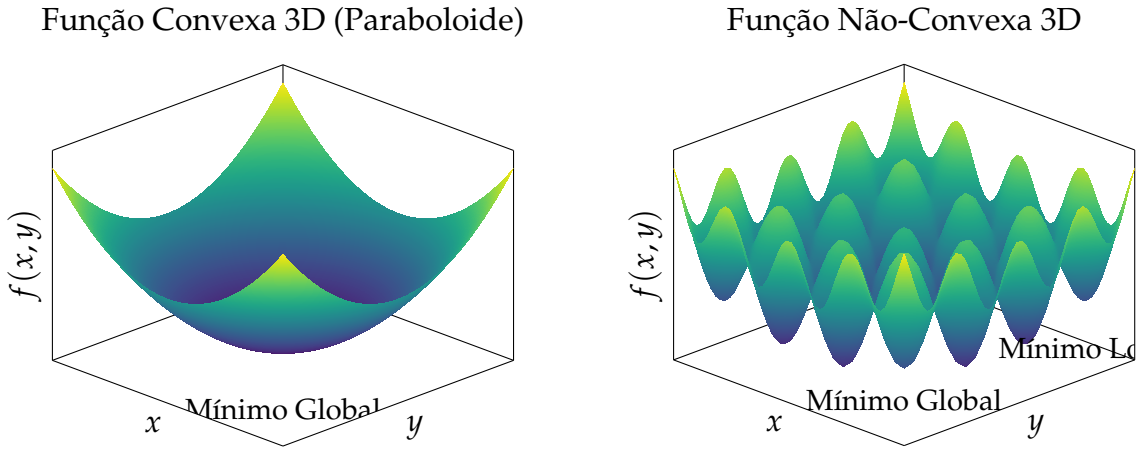


Figura 2 – Comparação entre funções 3D convexas e não-convexas.

Implementação em Python

Algorithm 1 O Método do Gradiente (Descida do Gradiente)

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

Para implementar o método do gradiente utilizando Python e a biblioteca de cálculos Numpy, deve-se seguir como base a equação 6.1, criando uma classe implenta essa ferramenta, recebendo como parâmetros de entrada a taxa de aprendizado (`learning_rate`),

a função que se quer encontrar o ponto de mínimo (`function`), e um ponto inicial (`initial_point`), que pode ser um conjunto de coordenadas aleatórias ou escolhidas pelo programador.

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

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

Bloco de Código : Classe completa do otimizador GradientDescent

```
class GradientDescent:

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

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

Note que a classe apresenta apenas dois métodos, o primeiro sendo o `__init__`,

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

6.2 A Retropropagação: Aprendendo com os Erros

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

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

Essa ferramenta foi introduzida para a comunidade científica pelos pesquisadores Rumelhart, Hinton e Williams (1986) no texto *Learning Representations by Back-Propagating Errors*, e será explicada nessa seção de forma detalhada. Para isso Rumelhart, Hinton e Williams (1986) comecam introduzindo três diferentes funções que serão utilizadas ao longo do texto para deduzir o os conceitos da retropropagação do gradiente, sendo elas: a equação do neurônio, a função de ativação sigmoide e a função de custo do erro quadrático médio.

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

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

Sendo que:

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

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

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

A segunda fórmula diz respeito a função de ativação que será utilizada, neste caso, Rumelhart, Hinton e Williams (1986) utilizam a sigmoide logísitca, representada na fórmula 7.2, 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, Rumelhart, Hinton e Williams (1986) utilizam a função do erro quadrático médio (MSE), a qual calcula o erro entre a saída atual do modelo e o valor real desejado, depois ela eleva ao quadrado esse valor, e por último, divide por dois. Ela é dada pela equação 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:

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

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

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

Note que todos os termos que não possuem o índice k, como a parte $(y_1 - d_1)^2$, serão valores constantes, e 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 etepa 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 a um peso w_{ij} específico da rede neural, para isso, é preciso calcular a expressão $\partial E/\partial w_{ij}$, note mais uma vez que o peso w_{ij} não aparece diretamente na equação do erro, de forma que é necessário mais uma vez aplicar a regra da cadeia para calcular essa derivada. Assim, a expressão inicial é:

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

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

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

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

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

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

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

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

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

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

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

Agora é possível voltar para a expressão inicial, obtendo então a expressão final para o gradiente do erro em relação a um peso específico da rede neural, sendo ela dada então pelas expressões 6.7

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

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

Em que:

- $\partial E/\partial w_{ij}$: Representa como o erro varia em relação à variação de um peso w_{ij}, 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.

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

O próximo passo não está no artigo original, ele envolve entender como o erro varia em relação a um viés de um neurônio específico, ou seja, é preciso calcular a expressão $\partial E/\partial b_j$. Note que mais uma vez o viés b_j não aparece diretamente na equação do erro, ele está dentro da equação do neurônio, assim, novamente é preciso aplicar a regra da cadeia para encontrar essa derivada. 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

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

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;
- ϵ representa o tamanho do passo / taxa de aprendizado;
- $\partial E/\partial w$ representa o gradiente do erro em relação ao peso.

Analogamente, a regra de atualização de um viés pelo método do gradiente é dada pela equação 6.10

Regra de Atualização de um Viés Através do Método do Gradiente

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

- $b_{j,t+1}$ 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;
- ϵ 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 viéses, uma matriz de pesos e um vetor de neurônios. Assim, as expressões eq:regra-de-atualizacao-do-vetor-de-pesos-atraves-do-metodo-do-gradiente e 6.12, 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

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

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;
- ϵ: 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

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

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;
- ϵ: 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.

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

Para encontrar esse gradiente, é preciso desenvolver a expressão $\partial E / \partial x_j$, que representa o gradiente do erro da entrada de um neurônio j da primeira camada oculta. Para isso, é possível utilizar como base a equação 6.6, 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:

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

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

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

Generalizando essa expressão, é possível concluir que o gradiente para uma camada é dado ela equação 6.13.

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 erão 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 neurônios agonizantes.

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

$$x_{n+1} = x_n - \epsilon \nabla f(x_n) + \beta(x_n - x_{n-1}) \tag{6.14}$$

Em que:

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

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

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

Em que M representa o maior valor da matriz hessiana, equanto m é o menor valor no ponto de mínimo da função também da matriz hessiana. 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.

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

Em que:

- $\Delta w(t)$: Representa a variação nos pesos;
- ϵ: Representa a taxa de aprendizado;
- $\nabla f(x)$: Representa o gradiente;

- α: Representa um fator de decaimento exponencial entre 0 e 1 que determina a contribuição do gradiente passado para o gradiente futuro.

Implementação em Python

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

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

Implementação em Python

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 vimos 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

$$\theta_{\text{novo}} = \theta_{\text{antigo}} - n\nabla J(\theta; x^{(i)}; y^{(i)}) \tag{6.16}$$

Em que:

- θnovo: representa os parâmetros do modelo já atualizados;
- θ_{antigo}: representa os parâmetros do modelo na iteração atual;
- n: representa a taxa de aprendizado;
- $J(\theta; x^{(i)}; y^{(i)})$: representa o vetor gradiente.

Implementação em Python

6.3.3 Método do Gradiente em 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.

Implementação em Python

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 problem 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 consistete 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

$$
\begin{aligned}
y_k &= x_k + \beta(x_k - x_{k-1}) && \text{1. Calcula uma predição futura (lookahead) } y_k, \\
& && \text{dando um passo na direção do momento anterior.} \\
x_{k+1} &= y_k - \epsilon \nabla f(y_k) && \text{2. Calcula o gradiente em } y_k \text{ (na predição)} \\
& && \text{e atualiza os pesos } x_k \text{ a partir de } y_k.
\end{aligned}
\tag{6.17}
$$

Em que:

- y_k: Representa o ponto de "lookahead";
- x_k: Representa os parâmetros na iteração k;
- x_{k-1}: Representa os parâmetros na iteração $k-1$;
- β: Representa o coeficiente de inércia ou momento;
- x_{k+1}: Representa os parâmetros na iteração $k+1$;
- $epislon$: Representa a taxa de aprendizado;
- $\nabla f(y_k)$: Representa o vetor gradiente calculado no ponto de "*lookahead*"

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.

Implementação em Python

Algorithm 3 Gradiente Acelerado de Nesterov

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

6.4 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), Duchi, Hazan e Singer (2011) o descrevem metafóricamente 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 bloco de algoritmo 4.

O AdaGrad 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
$\mathbf{n}_0 \leftarrow 0$ ▷ Inicializar o vetor acumulador
while não convergir **do**
  $\mathbf{g}_t \leftarrow \nabla_{\theta_{t-1}} f(\theta_{t-1})$
  $\mathbf{n}_t \leftarrow \mathbf{n}_{t-1} + \mathbf{g}_t \odot \mathbf{g}_t$
  $\theta_t \leftarrow \theta_{t-1} - \eta \frac{\mathbf{g}_t}{\sqrt{\mathbf{n}_t + \epsilon}}$
end while
Retorne: θ_t ▷ Parâmetros resultantes

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 1

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

Tabela 1 – 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).

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 **??**

Tabela 2 – 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 3, 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.

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,

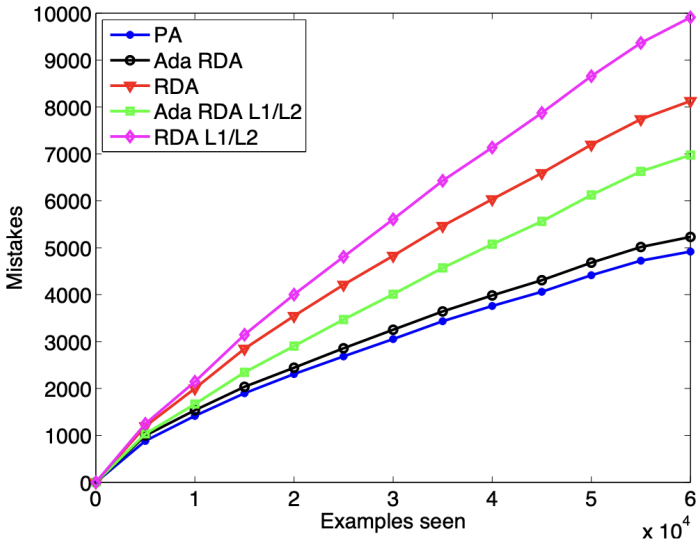


Figura 3 – Curvas de aprendizado no dataset MNIST.

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.

Implementação em Python

6.4.2 RMSProp

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
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: $\quad \mathbf{n}_t \leftarrow \nu \mathbf{n}_{t-1} + (1-\nu)(\mathbf{g}_t \odot \mathbf{g}_t)$
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

Implementação em Python

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*, ele 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 priemiro 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 consistem 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 **??**.

Algorithm 6 Adam (Adaptive Moment Estimation)

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 **??** propraram uma série de experimentos, sendo alguns deles envolvendo uma regressão logística, uma rede neural multi-camadas e uma rede convolucional.

No primeiro deles, eles avaliam esse método 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 oti-

mização com o gradiente estocástico acelereado (accelerated SGD), com o otimizador de Nesterov e também como o AdaGrad; de forma que eles foram capazes de concluir que o Adam possui uma convergência similar ao SGD e Nesterov, equanto o AdaGrad apresenta uma convergência mais demorada e com um maior custo de treino. Esse compartivo pode ser visto na figura 4.

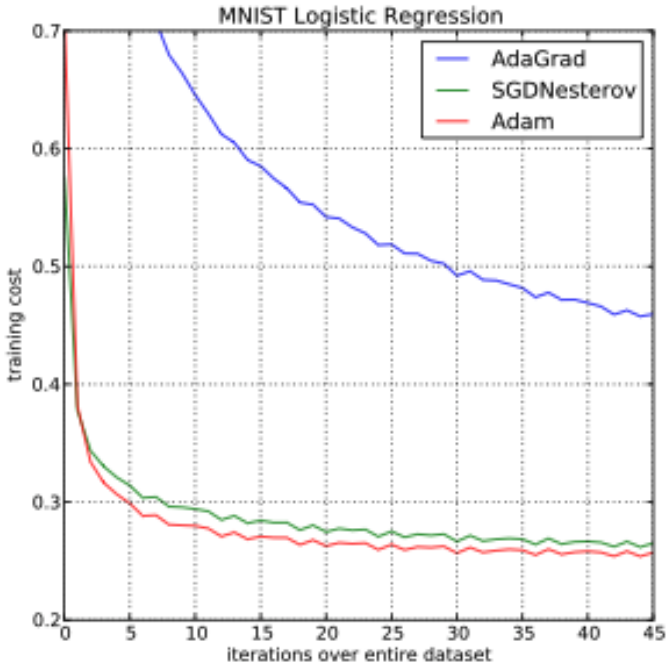


Figura 4 – Curvas de aprendizado no dataset MNIST.

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 5.

Já para a rede convolucional, Kingma e Ba (2017) criaram uma CNN com três camadas, a primeira com filtros de dimenões 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

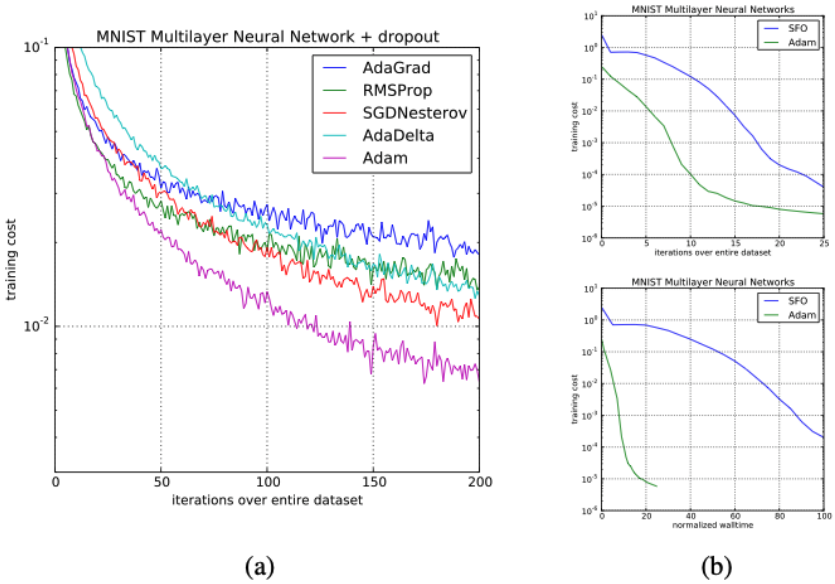


Figura 5 – Training of multilayer neural networks on MNIST images. (a) Neural networks using dropout stochastic regularization. (b) Neural networks with deterministic cost function. We compare with the sum-of-functions (SFO) optimizer.

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.

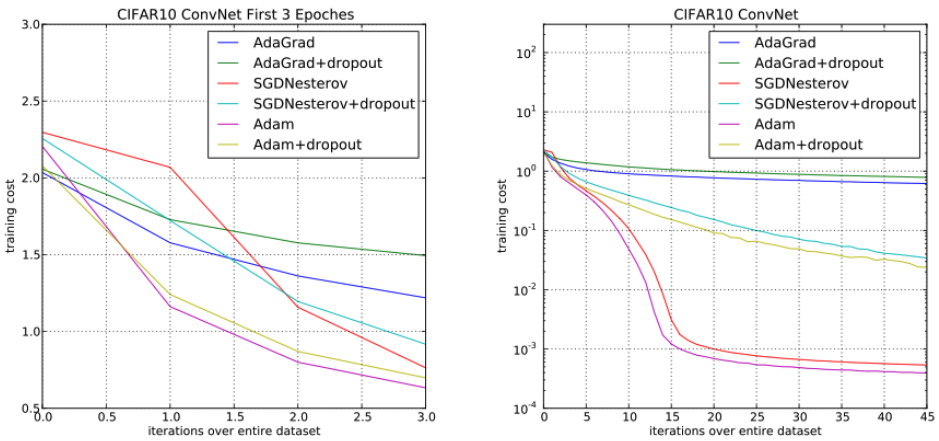


Figura 6 – Convolutional neural networks training cost. (left) Training cost for the first three epochs. (right) Training cost over 45 epochs

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.

Implementação em Python

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 uma variante dele, o AdaMax, que ao invés de utilizar a norma L^2 para calcular o segundo momentom, 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 algortimo listado no pseudocódigo 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 α
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
  $\theta_t \leftarrow \theta_{t-1} - (\alpha / (1 - \beta_1^t)) \cdot m_t / u_t$ ▷ Atualizar parâmetros
end while
Retorne: θ_t ▷ Parâmetros resultantes

Implementação em Python

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 algoritmo Nadam está representado em **??**

O Nadam recebe como parâmetros de entrada uma taxa de aprendizado α, 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 α (pode ser agendada, ex: α_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**
  $t \leftarrow t+1$
  $\mathbf{g}_t \leftarrow \nabla_{\theta_{t-1}} f_t(\theta_{t-1})$
  $\mathbf{m}_t \leftarrow \mu\mathbf{m}_{t-1} + (1-\mu)\mathbf{g}_t$
  $\mathbf{v}_t \leftarrow \nu\mathbf{v}_{t-1} + (1-\nu)(\mathbf{g}_t \odot \mathbf{g}_t)$
  $\hat{\mathbf{m}}_t \leftarrow (\mu\mathbf{m}_t/(1-\mu^t)) + ((1-\mu)\mathbf{g}_t/(1-\mu^t))$
  $\hat{\mathbf{v}}_t \leftarrow \frac{\nu\mathbf{v}_t}{1-\nu^t}$
  $\theta_t \leftarrow \theta_{t-1} - \alpha\frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t}+\epsilon}$
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 7, 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.

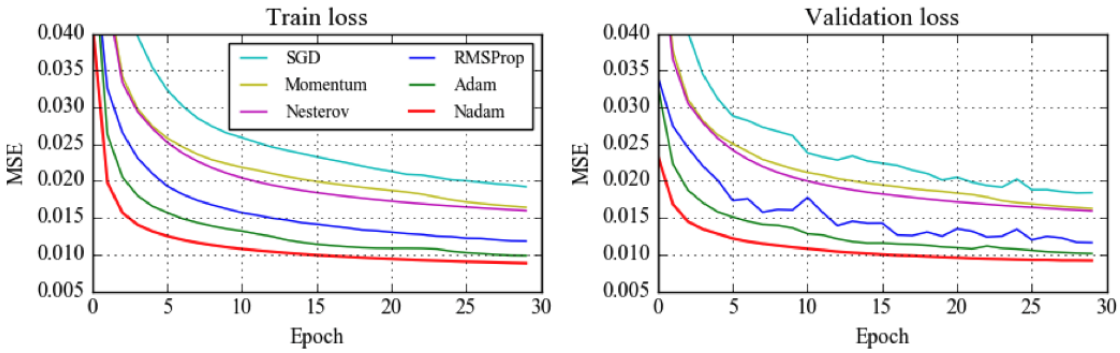


Figura 7 – Training and validation loss of different optimizers on the MNIST dataset

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.

Implementação em Python

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.

Algorithm 9 Adam com Decaimento de Peso Desacoplado (AdamW)

Dado: $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
Inicialize: passo de tempo $t \leftarrow 0$, vetor de parâmetros $\theta_0 \in \mathbb{R}^n$, vetor de 1º momento $m_0 \leftarrow 0$, vetor de 2º momento $v_0 \leftarrow 0$, multiplicador de agendamento $\eta_0 \in \mathbb{R}$
repeat
    $t \leftarrow t + 1$
    $\nabla f_t(\theta_{t-1}) \leftarrow \text{SelecionarLote}(\theta_{t-1})$ ▷ Seleciona o lote e retorna o gradiente
    $g_t \leftarrow \nabla f_t(\theta_{t-1})$ ▷ O gradiente NÃO inclui o termo de decaimento de peso
    $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ ▷ Atualizar 1º momento (operações element-wise)
    $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ ▷ Atualizar 2º momento
    $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ ▷ Corrigir viés do 1º momento
    $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ ▷ Corrigir viés do 2º momento
    $\eta_t \leftarrow \text{DefinirMultiplicadorDeAgendamento}(t)$ ▷ Pode ser fixo, decair, ou para reinícios abruptos
    $\theta_t \leftarrow \theta_{t-1} - \eta_t \left(\frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \right) - \eta_t \lambda \theta_{t-1}$ ▷ Atualização desacoplada do decaimento de peso
until critério de parada for atingido
Retorne: parâmetros otimizados θ_T

Para comprovar a performace do novo otimizador criado os autores decidem utilizar uma série de experimentos, de forma que foi possível comprovar que o Adam trás uma melhor generalização quando comparado com o Adam tradicional, de forma que foi possível encontrar um Top-5 Erro de Teste no ImageNet32x32 menor utilizando essa nova variante do Adam.

Implementação em Python

6.4.7 Rectified Adam (RAdam)

Uma outra variante do Adam que també 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.

Dessa forma, os pesquisadores chegaram a desenvolver o algoritmo que está listado no bloco de pseudocódigo **??**

Algorithm 10 Adam Retificado (RAdam)

Entrada: $\{\alpha_t\}_{t=1}^{T}$: tamanho do passo, $\{\beta_1, \beta_2\}$: taxas de decaimento, θ_0: parâmetro inicial, $f_t(\theta)$: função objetivo estocástica.
Saída: θ_T: parâmetros resultantes.
$m_0, v_0 \leftarrow 0$ ▷ Inicializar 1º e 2º momentos móveis
$\rho_\infty \leftarrow 2/(1-\beta_2) - 1$ ▷ Calcular o comprimento máximo da Média Móvel Simples (SMA) aproximada
for $t = 1$ **to** T **do**
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ ▷ Calcular gradientes no passo de tempo t
    $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$ ▷ Atualizar 2º momento móvel exponencial
    $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$ ▷ Atualizar 1º momento móvel exponencial
    $\hat{m}_t \leftarrow m_t / (1-\beta_1^t)$ ▷ Calcular média móvel com correção de viés
    $\rho_t \leftarrow \rho_\infty - 2t\beta_2^t/(1-\beta_2^t)$ ▷ Calcular o comprimento da SMA aproximada
    if a variância for tratável, i.e., $\rho_t > 4$ **then**
        $l_t \leftarrow \sqrt{(1-\beta_2^t)/\sqrt{v_t}}$ ▷ Calcular a taxa de aprendizado adaptativa
        $r_t \leftarrow \sqrt{\frac{(\rho_t-4)(\rho_t-2)\rho_\infty}{(\rho_\infty-4)(\rho_\infty-2)\rho_t}}$ ▷ Calcular o termo de retificação da variância
        $\theta_t \leftarrow \theta_{t-1} - \alpha_t r_t \hat{m}_t l_t$ ▷ Atualizar parâmetros com momentum adaptativo
    else
        $\theta_t \leftarrow \theta_{t-1} - \alpha_t \hat{m}_t$ ▷ Atualizar parâmetros com momentum não-adaptado
    end if
end for
Retorne: θ_T

Para avaliar o RAdam foi utilizado uma série de benchmarks: One Billieon Word (para modelagem linguística), CIFAR-10 e ImageNet (para classificação de imagens) e IWSLT'14 De-En/EN-DE e WMT'16 EN-De (para neural machine translation). Comparando com o Adam tradicional, as performaces de modelagem de lingaugem e de classificacao de imagens mostram que o RAdam supera o Adam nos três datasets, contudo, um ponto a ser destcado por Liu *et al.* (2021) é que o termo retificador faz com que a performance do RAdam seja masi lenta que o Adam nas primeiras épocas, permitindo que uma convergência mais rápida do RAdam deois disso. Assim, como os próprios autores apresentam, reduzir a variância da taxa de aprendizado adpativativa nos estágios iniciais, faz com que a covergência seja mais rápida além de garantir uma melhor performance (Liu *et al.*, 2021).

Implementação em Python

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

7.1 Teoremas da Aproximação Universal

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

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

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

Pense que temos os pontos dispostos no gráfico da figura 8 e queremos encontrar uma reta que tente passar o mais próximo de cada um deles.

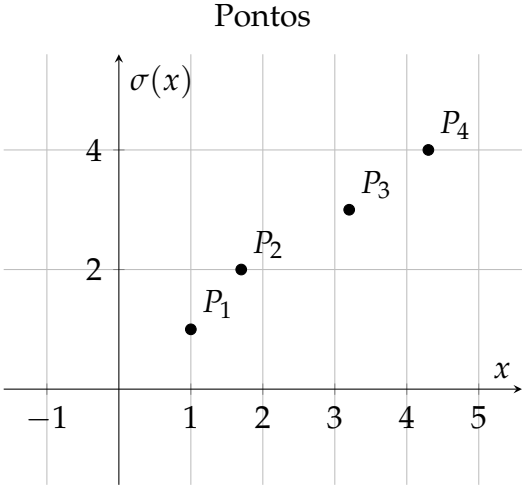


Figura 8 – Conjunto de pontos dispostos no plano cartesiano.

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

$$y = 0.8623x + 0.3011$$

Com base nessa função que encontramos, podemos desenhá-la junto ao gráfico dos pontos e vermos se ela é realmente uma boa aproximação, assim temos a figura 9

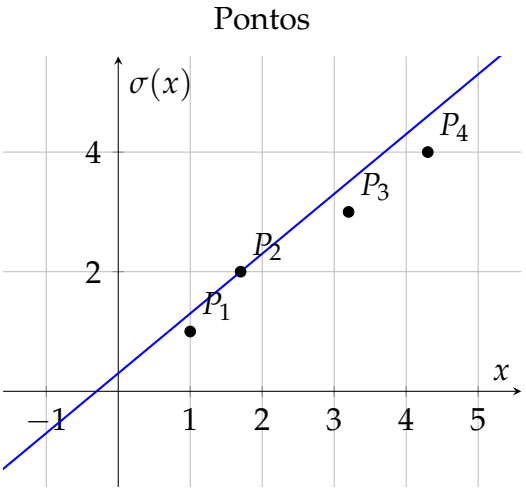


Figura 9 – Conjunto de pontos dispostos no plano cartesiano.

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

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

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

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

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

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

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

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

7.2 Exemplo Ilustrativo: Empurrando para extremos

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

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

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

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

7.3 A Sigmoide Logística

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

7.4 Contexto Histórico: Popularização da Sigmoide em Redes Neurais

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

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

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

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

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

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

Neurônio de Wilson e Cowan

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

Nessa equação, o parâmetro a representa a inclinação, a qual foi ajustada para passar pela origem ($\mathcal{S}(0) = 0$) e θ é o limiar. Para o plotar o gráfico da figura 10, foi utilizado os mesmos valores escolhidos pelos cientistas na pesquisa, assim $a = 1.2$ e $\theta = 2.8$.

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

Com isso, ao consideramos esses dois cenários: a criação da retropropagação e busca na natureza para inspiração na criação de redes neurais. A sigmoide, junto com a sua família, se tornaram funções muito populares para a época, estando presentes em varias redes neurais criadas. Um desses exemplos é a rede de Elman, um tipo de rede

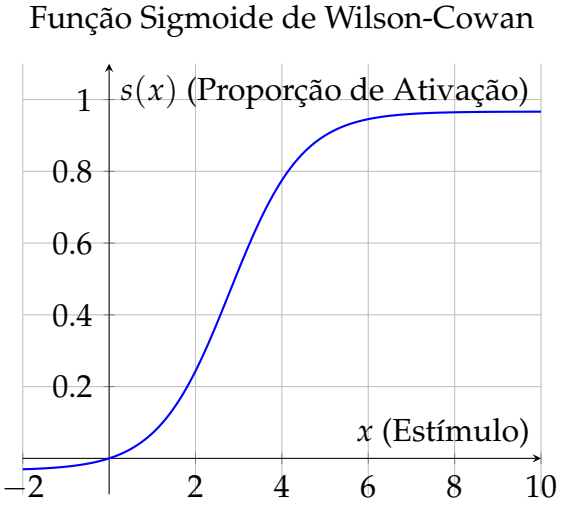


Figura 10 – Gráfico da função sigmoide conforme proposta por Wilson e Cowan (1972), com parâmetros de exemplo $a = 1.2$ e $\theta = 2.8$.

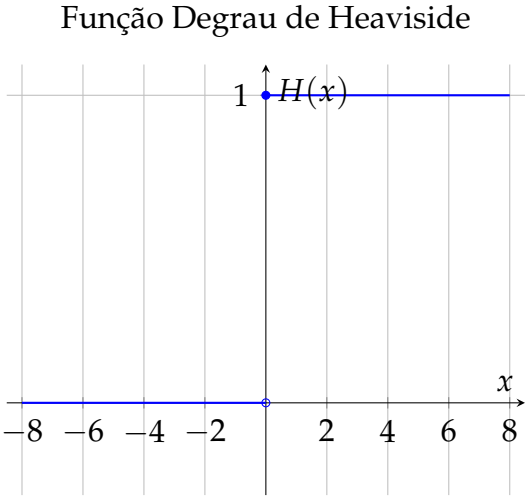


Figura 11 – Gráfico da função degrau de Heaviside.

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

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

Uma função que era muito utilizada era a heavside, ou degrau em português. Ela inclusive esteve presente na produção da rede neural Perceptron criada por Frank Rosenblatt e introduzida para a comunidade científica no artigo **"The Perceptron: A pro-**

babilistic model for informations storage and organization in the brain" em 1958.

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

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

Sigmoide Logística

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

Função Sigmoide

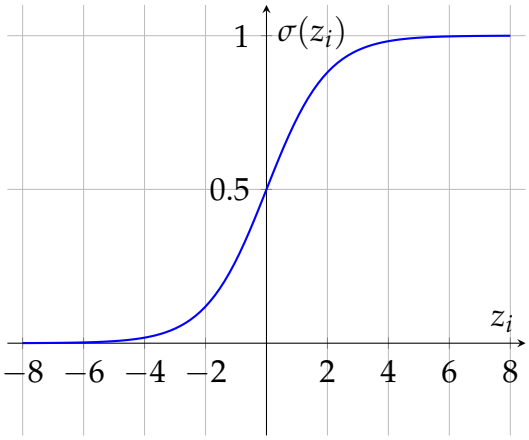


Figura 12 – Gráfico da função sigmoide logística.

A função sigmoide é escrita com uma exponencial, como na fórmula **??**. Johannes Lederer, no texto **"Activations Functions in Artificial Neural Networks: A Systematic**

Overview", explica sobre a sigmoide, segundo o autor, ela é uma função limitada, diferenciável e monotônica, o que significa que conforme os valores de x aumentam os valores de $f(x)$ também aumentam.

Como vemos na figura 12, o gráfico da sigmoide possui o formato de um "S"deitado. Assim, também podemos dizer que a função sigmoide é uma função suave, contínua (o que a possibilita de ser derivável em todos os pontos) e também é não linear. Lederer explica que uma das propriedades interessantes da sigmoide está no fato dela empurrar os valores de entrada para dois extremos, neste caso, 0 e 1.

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

Agora podemos discutir a sua derivada, para resolvê-la, podemos aplicar uma regra do quociente obtendo a expressão **??** e o gráfico 13

Derivada da sigmoide logística

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

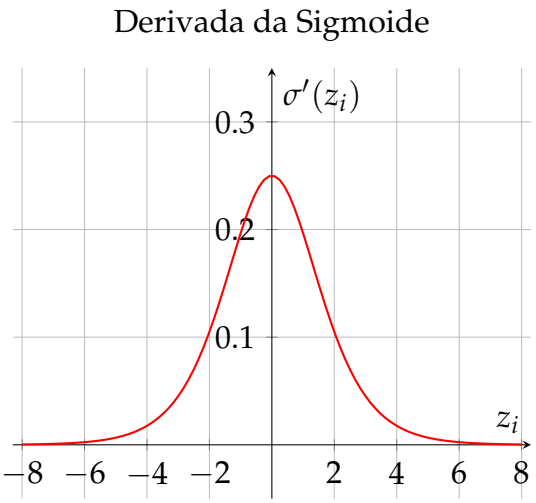


Figura 13 – Gráfico da derivada da sigmoide.

Notamos que sua derivada é contínua em todos os pontos e que ela atinge um ponto máximo entre 0.2 e 0.3 e que para valores que passam de ± 8 os resultados serão muito próximos de zero, isso acontece pois, como vemos no gráfico 12, a própria função sigmoide, não apresenta muita variação de valores conforme eles ultrapassam ± 8. Guarde essa informação, pois ela sera necessária para entender o conceito de gradiente evanescente, que será explicado no fim do capitulo. Contudo, a derivada da sigmoide também pode ser expressa em termos da própria sigmoide como a equação **??** nos mostra.

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

Implementação em Python

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

Bloco de Código : Classe completa do função de ativação Sigmoid

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

7.5 Tangente Hiperbólica

Assim como a função sigmoide, a tangente hiperbólica não possui suas origens voltadas para o uso em redes neurais. Neste caso, um dos matemáticos que ficou reconhecido por criar a notação das funções hiperbólicas, seno, cosseno e tangente, foi o suíço Johann Heinrich Lambert no trabalho de 1769 **"Mémoire sur quelques propriétés remarquables des quantités transcendantes circulaires et logarithmiques"** (Memória

sobre algumas propriedades notáveis de grandezas transcendentais circulares e logarítmicas), em que provou que muitas das identidades trigonométricas possuíam suas equivalentes hiperbólicas.

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

Semelhante a sigmoide, a tangente hiperbólica possui várias propriedades parecidas. Como afirma Lederer em **"Activation Functions in Artificial Neural Networks: A Systematic Overview"** a tangente hiperbólica é infinitamente diferenciável, sendo uma versão escalada e rotacionada da sigmoide logística. Assim, como podemos ver no gráfico da figura 16, ela é uma função que está centrada em zero, e seus valores variam agora em um intervalo de -1 a 1, diferente da sigmoide, que varia somente de 0 a 1.

Podemos escrever a tangente hiperbólica utilizando a definição de tangente, que é o quociente a função seno com a função cosseno, só que neste caso usaremos as funções hiperbólicas. Assim temos a expressão **??** e seu gráfico 16.

Tangente Hiperbólica

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

Função Tangente Hiperbólica

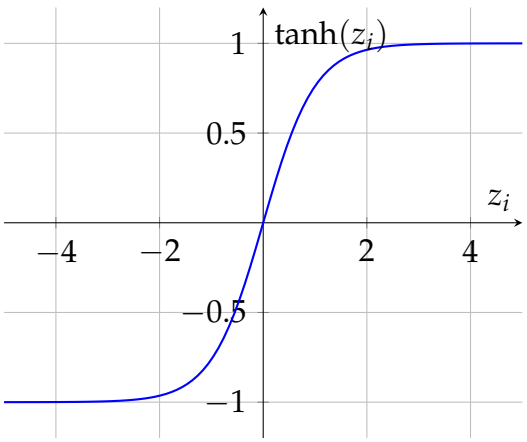


Figura 14 – Gráfico da tangente hiperbólica.

Agora que temos a expressão da tangente hiperbólica e seu gráfico podemos tam-

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

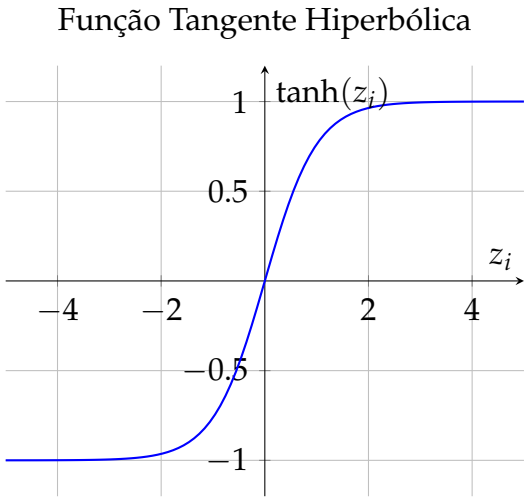


Figura 15 – Gráfico da tangente hiperbólica.

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

Derivada da tangente hiperbólica

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

Implementação em Python

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

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

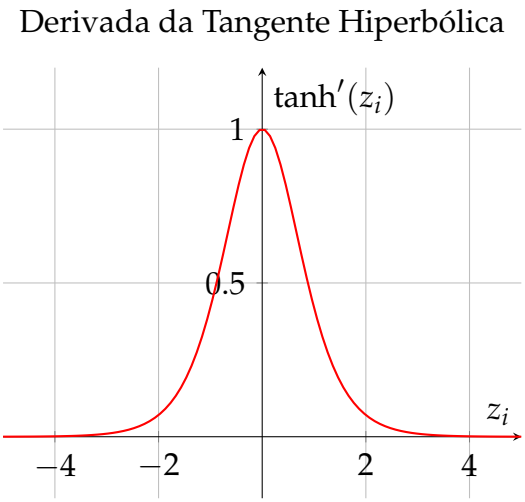


Figura 16 – Gráfico da derivada da tangente hiperbólica.

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

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

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

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

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

7.6 Softsign: Uma Sigmoidal Mais Barata

A próxima função que vamos conhecer é a softsign, diferente da tangente hiperbólica e da sigmoide que tiveram suas origens em outros campos diferentes da ciência da computação, a softsign foi criada com o intuito de ser trabalhada em uma rede neu-

ral. Ela foi introduzida no artigo **"A Better Activation Function for Artificial Neural Networks"** de 1993, do cientista D.L. Elliott. No texto ele propõem a softsign como uma alternativa para as funções sigmodais tradicionais.

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

Contudo, a principal diferença dela com as outras sigmodais está na sua fórmula, como podemos ver na equação **??** ela não utiliza nenhum exponencial para compor sua função. Isso faz com que ela seja uma função mais "barata"em termos de poder computacional para ser implementada em redes neurais. Assim, podemos obter resultados parecidos porem utilizando cálculos menos complexos e com isso teremos redes mais rápidas de serem treinadas. Um comparativo com as funções sigmodais desse texto e como elas reagem poderá ser visto em uma seção futura.

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

Softsign

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

Função Softsign

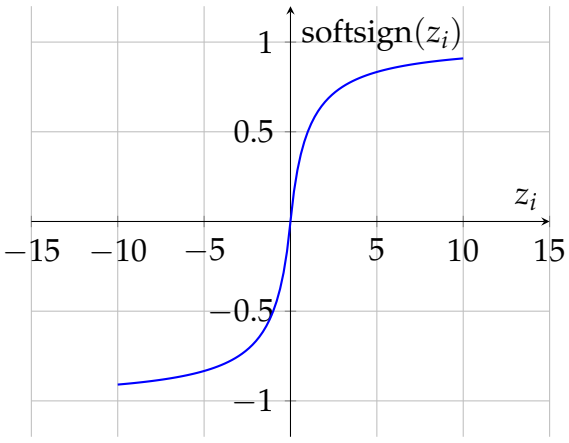


Figura 17 – Gráfico da função Softsign.

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

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

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

Derivada da softsign

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

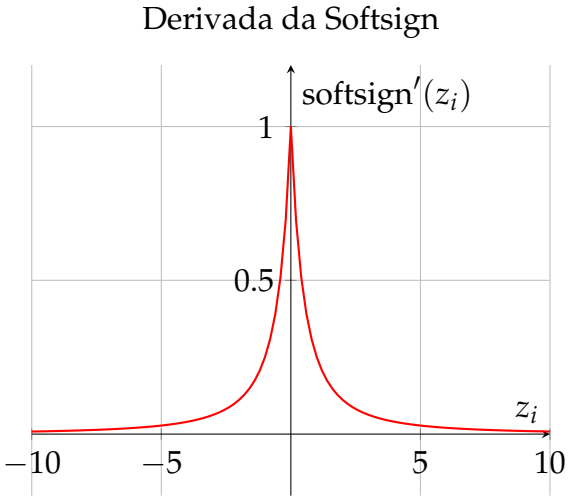


Figura 18 – Gráfico da derivada da função Softsign.

Implementação em Python

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

Bloco de Código : Classe completa do função de ativação Softsign

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

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

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

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

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

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

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

Hard Sigmoid

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

Função Hard Sigmoid

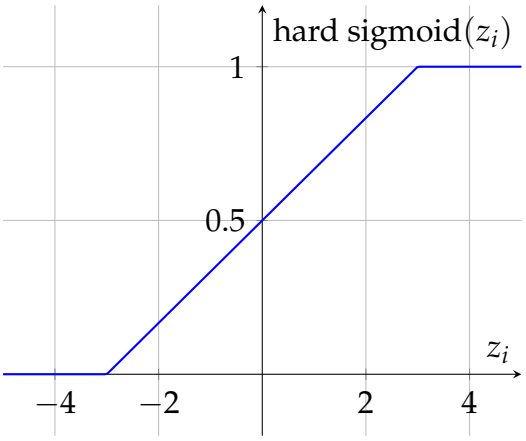


Figura 19 – Gráfico da função hard sigmoid.

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

Derivada da Hard Sigmoid

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

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

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

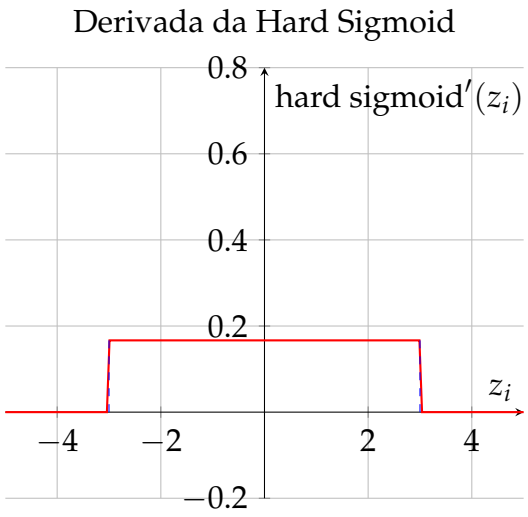


Figura 20 – Gráfico da derivada da Hard Sigmoid.

Hard Tanh

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

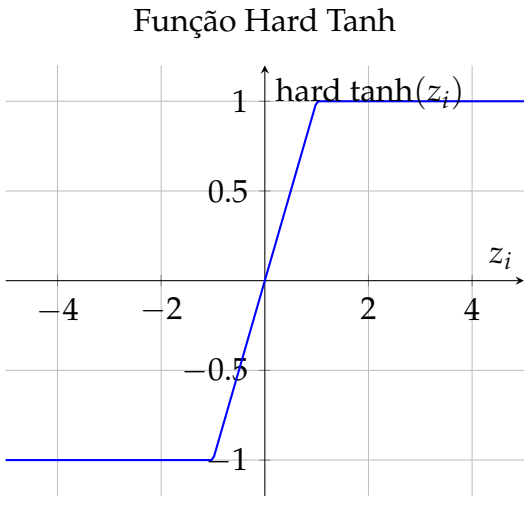


Figura 21 – Gráfico da função Hard Tanh. Fonte: PyTorch.

Utilizamos o mesmo procedimento da hard sigmoid para obter a derivada da hard tanh, derivamos as três expressões e encontramos então a equação **??**, ao lado, temos a figura 21, que representa o gráfico da derivada da hard tanh.

Derivada da Hard Tanh

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

Derivada da Hard Tanh

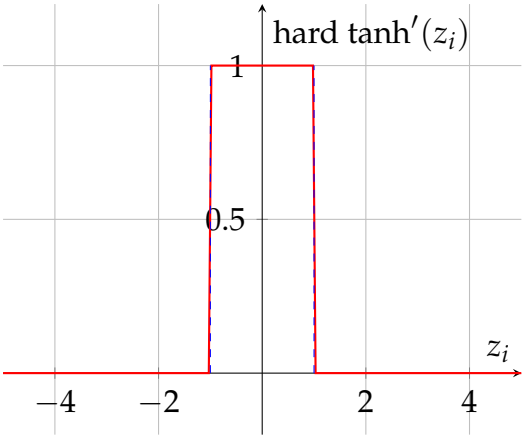


Figura 22 – Gráfico da derivada da Hard Tanh.

Implementação em Python

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

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

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

class HardSigmoid(Layer):

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

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

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

        return output

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

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

        return grad_output * hard_sigmoid_grad, None
```

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

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


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

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

    def backward(self, grad_output):

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

        return grad_output * hard_tanh_grad, None
```

7.8 O Desaparecimento de Gradientes

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

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

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

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

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

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

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

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

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

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

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

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

7.9 Comparativo de Desempenho das Sigmoidais

8 Funções de Ativação Retificadoras

8.1 Exemplo Ilustrativo: 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.

As funções retificadoras, como a ReLU, são o tópico principal deste texto, para isso, antes de conhecê-las vamos entender um pouco do cenário que elas surgiram, com uso das funções sigmoides e o problema do gradiente em fuga. Em seguida veremos a ReLU os problemas que ela pode causar em uma rede neural. Seguindo adiante, conheceremos as suas variantes com vazamento, como a Leaky ReLU, depois as variantes

suaves, como a ELU. Para conhecermos todas essas funções, serão utilizados gráficos, equações, comparativos de pesquisas e citações que explicam sobre suas propriedades. No final, veremos um comparativo com essas funções utilizando a construção de uma rede neural convolucional com o dataset CIFAR-10.

8.2 Rectified Linear Unit e Revolução Retificadora

Como vimos anteriormente, as funções sigmoides 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 (**Householder1941**).

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

Essa equação nos mostra quando uma fibra nervosa irá disparar, para isso, devemos 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 (**Householder1941**). Além disso, **Householder1941** 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, nos lembra bastante a expressão da função ReLU, a qual é denotada pela fórmula 8.3.

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

A ReLU é o tópico principal desse texto e será a primeira função de ativação que iremos estudar. 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 pro-

blema do gradiente em fuga. As funções retificadoras, sendo a principal delas a ReLU, surgem para corrigir esse problema crônico.

Dessa forma antes de conhecermos de fato a ReLU e suas propriedades, vamos antes 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 gradiente em fuga.

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, **Nair2010** 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 texto *Deep Sparse Rectifier Neural Networks* dos autores **Bordes**, 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 (**Bordes**). 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 (**Bordes**);

- **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 dimensi-

onalidade efetiva da representação para uma determinada entrada e a precisão necessária (**Bordes**);

- **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 (**Bordes**);

- **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 (**Bordes**).

Por fim, um último trabalho que colaborou para a popularização da ReLU foi a *AlexNet*, de **AlexNet**, 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, a *AlexNet* 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, além disso a *AlexNet* fez uso da ReLU em sua arquitetura (**AlexNet**). Podemos ver um esquema de sua arquitetura na figura **??**.

Assim, como podemos ver na tabela 3, a *AlexNet* foi capaz de alcançar uma taxa de erro de 15.3% na fase de testes, podemos notar 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 gradiente em fuga como nas sigmoides. Além disso, a rede SIFT + FVs (*Scale-Invariant Feature Transform + Fisher Vectors*) é mostrada na tabela como base de comparativo, podemos notar 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%.

Agora que conhecemos um pouco de como foi o contexto em que a ReLU surgiu, podemos de fato conhecê-la. Para isso, primeiro conhecemos sua fórmula representada pela equação 8.3, mas também podemos representá-la com a expressão reduzida

Tabela 3 – 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. 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*.

$\max(0, z_i)$. Nós podemos interpretar essa definição como uma pergunta em que a função ReLU recebe um número como entrada e faz uma pergunta "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 estamos falando de números, mas 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, temos também o seu gráfico, que está presente na figura **??**, podemos ver que ele é bem mais simples quando comparamos com a sigmoide, por exemplo, ele é a apenas a junção de duas retas, sendo 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 tendencia 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 gradiente em fuga mas também era muito mais barata.

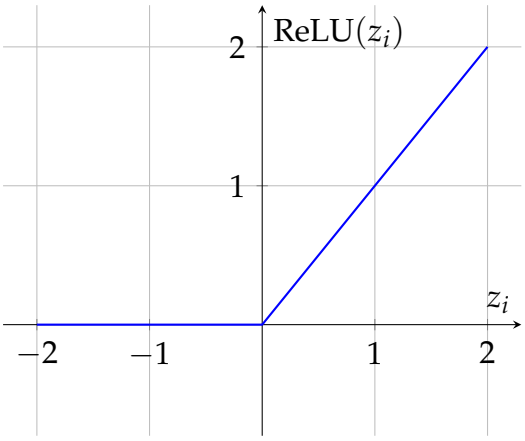
Rectified Linear Unit (ReLU)

$$\text{ReLU}(z_i) = \begin{cases} z_i, & \text{se } z_i > 0 \\ 0, & \text{se } z_i \leq 0 \end{cases} \tag{8.4}$$

Como estamos trabalhando com redes neurais, uma das maiores vantagens dessas redes é o fato delas “aprenderem” com base na retropropagação do gradiente nas ca-

h!

Figura 23 – Gráfico da função ReLU.



madas da rede. Assim, ao calcularmos 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 vamos aplicar em uma camada de neurônios da rede, dado que ela entrará no backward pass do modelo.

Para achar a derivada da ReLU, podemos simplesmente 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 encontramos 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 quando estamos trabalhando com o código dessa função, basta escolhermos qual será o resultado da ReLU quando esse valor de entrada for zero. Podemos dizer que ele será 1 ou zero, isso irá depender somente da nossa implementação da derivada da ReLU. Note que, essa descontinuidade da ReLU, se torna um problema quando estamos trabalhando com sua derivada.

Assim, temos a equação **??**.

Derivada Rectified Linear Unit (ReLU)

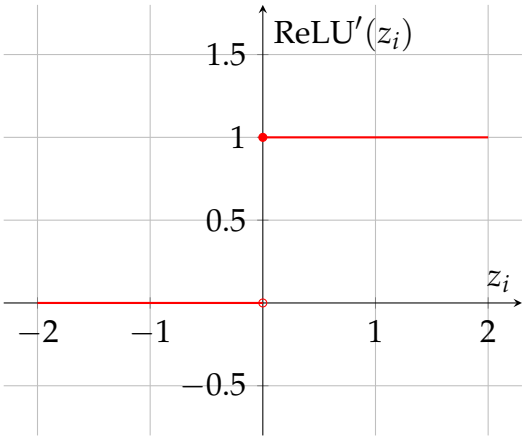
$$\frac{d}{dz_i}[ReLU](z_i) = \begin{cases} 1, & \text{se } z_i > 0 \\ 0, & \text{se } z_i \leqslant 0 \end{cases} \tag{8.5}$$

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 de termos a sua derivada em equação, podemos fazer também o seu gráfico na figura **??**, note que ela 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 24 – Gráfico da Derivada da Função ReLU



8.2.1 Implementação em Python

Agora que sabemos qual é o comportamento da ReLU, quais são suas fórmulas e quais são seus gráficos, podemos trabalhar em uma implementação dessa função utilizando Python. Para fazer isso, vamos criar uma classe, com duas funções: a *forward* e a *backward*. A função *forward* é responsável por implementar a ReLU, já a *backward* implementa a sua derivada. Para criar a *forward*, podemos utilizar uma função já pronta da biblioteca numpy, a maximum, essa função é responsável por comparar dois valores e escolher o maior deles. Neste caso, um dos valores será sempre zero, enquanto o outro será a entrada da função. Já a *backward*, irá receber o gradiente retropropagado, e irá aplicar na fórmula da derivada da ReLU, comparando se aquele valor é negativo e retornando zero, caso for, ou um caso seja positivo.

Assim, conhecemos a ReLU, uma função que é considerada padrão para a maioria das redes neurais *feedforward* (**Goodfellow2016**). Vimos que ela veio para ser uma alternativa para as funções sigmoides, sendo mais simples e resolvendo o problema do gradiente em fuga, contudo, a ReLU também apresenta problemas, como o do neurônios agonizantes a o explosão de gradientes, uma versão diferente do problema do gradiente em fuga. Dito isso, eles serão explicados em seguida para entendermos também as desvantagens de se utilizar essa função em uma rede neural.

Implementação em Python

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

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

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

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

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

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

8.3 Dying ReLUs Problem

8.4 Corrigindo o Dying ReLUs Problem: As Variantes com Vazamento

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.

Nós vimos, que a causa do neurônios 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 per-

mitindo a criação de uma função simples e rápida de ser computada em uma rede neural.

8.4.1 Leaky ReLU (LReLU)

Seguindo adiante, a próxima função que iremos ver é a Leaky ReLU, ela é uma variante da ReLU que foi criada com intuito de corrigir o problema do neurônios agonizantes. Assim como a ReLU, que foi explicada com a analogia do vendedor de pipoca, podemos 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 neurônios agonizantes.

Ela 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* (**Mass2013**). Além disso, a sua principal diferença, como explicam **Mass2013**, 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.

Em testes de desempenho realizados por **Mass2013**, 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 4. Nós podemos nessa tabela 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 a Tangente Hiperbólica, vemos que a rede que foi construída com 3 camadas utilizando a LReLU foi capaz de obter a taxa de erro de palavra (WER) mais baixa 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 **??**, 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 trás resultados melhores quando comparada com essas duas funções de ativação.

Tabela 4 – 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. 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, para isso, 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, podemos ver um empate quando analisamos 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, quando vemos a tabela 4, as redes que fizeram uso tanto da ReLU quanto da Leaky ReLU obtiveram a mesma acurácia de 53.9%, já quando comparamos com a tangente hiperbólica, vemos 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 comparamos com a Leaky ReLU, vemos outros ganhos também, como menores taxas de erro de palavra (WER) tanto no conjunto Switchboard (SWBD) quanto no conjunto de avaliação (EV).

Seguindo adiante, podemos discutir a expressão matemática da Leaky ReLU, a qual é dada pela equação **??**, 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 (**Mass2013**). 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 texto, em que diferentes autores optam por valores diferentes de α para melhor ajustar ao problema que está sendo analisado.

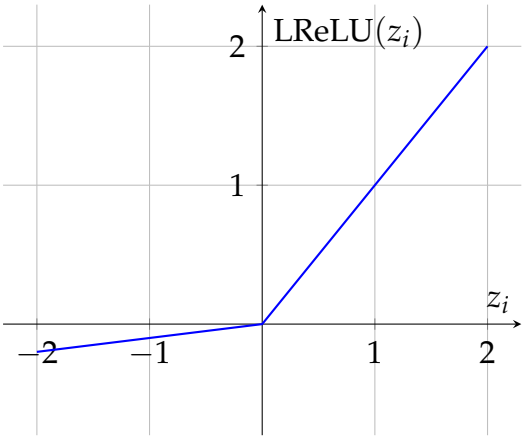
Leaky ReLU (LReLU)

$$\text{LReLU}(z_i) = \begin{cases} z_i, & \text{se } z_i \geq 0 \\ \alpha \cdot z_i, & \text{se } z_i < 0 \end{cases} \tag{8.6}$$

Cabe destacar, que assim como a ReLU tradicional, a Leaky ReLU também possuí uma expressão compacta, dada por $\max(\alpha z_i, z_i)$, sendo uma forma mais simples de escrever a expressão **??** com condicionais mas ainda sim manter o seu sentido. Já para a sua representação gráfica, podemos plotar o seu gráfico na figura **??**, analisando ele, vemos 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 analisamos seus valores negativos vemos 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 traçarmos os seus limites laterais, eles apresentam valores diferentes, por isso ela não pode ser derivada nesse ponto, assim como a ReLU que vimos anteriormente.

Figura 25 – Gráfico da Função Leaky ReLU (LReLU) com $\alpha = 0.1$



Sabendo de sua expressão e seu gráfico, podemos nos preocupar agora a calcular sua derivada, para isso, devemos 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 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, quando estamos trabalhando com a sua definição na retropropagação, podemos definir um valor para a derivada nesse ponto, assim como fizemos com a ReLU tradicional. Com isso em mente, temos a equação **??** que representa a derivada da Leaky ReLU, note que para a sua derivada nós não temos uma expressão reduzida quando comparada com a expressão original.

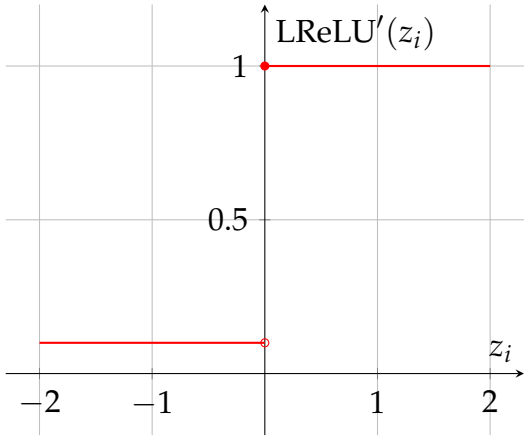
Derivada Leaky ReLU (LReLU)

$$\frac{d}{dz_i}[LReLU](z_i) = \begin{cases} 1, & \text{se } z_i > 0 \\ \alpha, & \text{se } z_i \leqslant 0 \end{cases} \tag{8.7}$$

Já que agora conhecemos a sua derivada, podemos também plotar o seu gráfico, o qual é dado pela figura **??**. Ele também é parecido com o gráfico da ReLU que vimos 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 26 – Gráfico da Derivada da Função Leaky ReLU (LReLU) com $\alpha = 0.1$



Voltando para *Recfier Nonlinearites Improve Neural Networks Acustic Models*, **Mass2013** explicam sobre outras propriedades da LReLu, como o fato dela sacrificar a esparsidade de zero duros (*hard-zero sparsity*) para ter um gradiente que é potencialmente mais robusto durante a otimização da rede neural. Além disso, os autores também explicam que ambas as funções retificadoras, tanto a ReLU quando a Leaky ReLU se saem bem nos testes de performance quando comparadas com a tangente hiperbólica além de se beneficiarem melhor da profundidade de rede quando comparadas com as sigmoides (**Mass2013**).

A Leaky ReLU já é uma evolução quando comparamos com a ReLU tradicional, mas, podemos ir além e encontrar funções ainda mais complexas que também buscam assim como a LReLU resolver o problema do neurônios agonizantes com um vazamento de gradiente nos casos negativos, uma dessas funções é a PReLU, a qual veremos em seguida.

Implementação em Python

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

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


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

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

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

8.4.2 Parametric ReLU

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 **He2015** 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 vimos anteriormente, isso ocorre, pois diferente da LReLU que possuía uma constante α fixa que multipli-

cava 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.

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 imagens de teste sem rótulos publicados (**He2015**). Para isso, **He2015** 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 **??**.

Conhecendo cada um dos modelos apresentados por **He2015**, podemos analisar como eles se comportaram nos testes de performance utilizando o dataset do ImageNet 2012, para isso, os autores realizaram testes tanto utilizando a nova função de ativação PReLU, mas também a ReLU tradicional, podendo ter uma base melhor para suas comparações. Além disso, como podemos ver na tabala **??**, é medido o erro Top-1 e o erro Top-5, o erro Top-1 nos mostra quão preciso é o modelo em seu melhor chute, já o erro Top-5 nos mostra se a resposta correta estava entre os top 5 melhores chutes feito pelo modelo. Assim, conhecendo esses parâmetros de medida, podemos chegar na conclusão de que quanto menor esses valores, melhor o modelo está performando, seguindo essa lógica, podemos ver 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.

Quando comparamos o modelo C, que faz uso da PReLU e possui mais camadas convolucionais, com o modelo A que faz uso da ReLU, vemos uma diferença de 2,21 pontos percentuais no erro Top-1, já quando analisamos o erro Top-5, essa diferença é de 1,21 pontos percentuais, o que indica que a ReLU é capaz de trazer melhores resultados quando comparada com redes que fazem uso da ReLU tradicional. Já quando comparamos 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 **??** possui bem menos camadas convolucionais que o modelo C, podemos 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 gradiente em fuga, é capaz de criar redes neurais que se beneficiam melhor com uma maior profundidade, sendo

capazes de extrair mais informações e com isso performar melhor.

Agora que conhecemos a Parametric ReLU e sabemos de seu potencial, podemos estudar a sua fórmula matemática, a qual é dada pela expressão **??**. Como explicam **He2015**, 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 vimos 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 (**He2015**). Assim, por esse fato temos 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.

Parametric ReLU (PReLU)

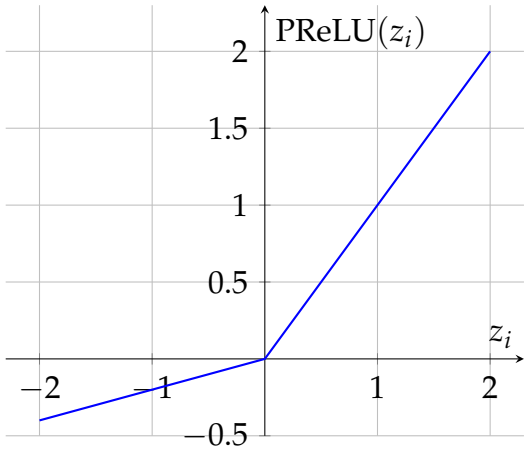
$$\text{PReLU}(z_i) = \begin{cases} z_i, & \text{se } z_i \geq 0 \\ \alpha_i \cdot z_i, & \text{se } z_i < 0 \end{cases} \tag{8.8}$$

Assim como a LReLU e a ReLU, também podemos expressar a PReLU como uma função reduzida ao invés das condicionais vistas na expressão **??**, para isso, ela utiliza a mesma forma $\max \alpha x, x$, que vimos na Leaky ReLU, mas agora, a constante possui outro significado, indicando que ela é um parâmetro aprendível, e não mais fixo como na Leaky ReLU.

Com base em sua equação, podemos plotar o gráfico da PReLU na figura **??**, note que caso este coeficiente for igual a zero, nos temos então a ReLU tradicional, já quando ele for igual a 0.1, teremos 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, nos notamos 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.

Se conhecemos a sua fórmula e como ela se comporta, podemos também derivar a PReLU, para isso, derivamos cada uma das expressões dos condicionais de forma separada, assim como fizemos 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 expressão **??** como sua derivada. Note que, essa derivada também não existe quando a entrada é exatamente zero, mas assim como na Leaky ReLU, "corrigimos"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.

Figura 27 – Gráfico da Função Parametric ReLU (PReLU) com $\alpha = 0.2$

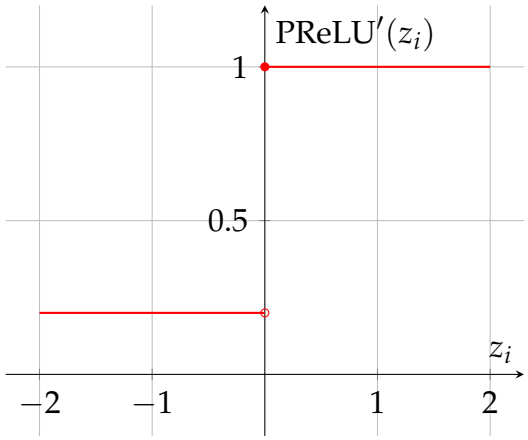


Derivada Parametric ReLU (PReLU)

$$\frac{d}{dz_i}[PReLU](z_i) = \begin{cases} 1, & \text{se } z_i > 0 \\ \alpha_i, & \text{se } z_i < 0 \\ \nexists, & \text{se } z_i = 0 \end{cases} \tag{8.9}$$

Sabendo a fórmula da derivada da PReLU, podemos também plotar o seu gráfico, o qual é dado pela figura **??**, 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 vimos apresentas por **He2015**.

Figura 28 – Gráfico da Derivada da Função Parametric ReLU (PReLU) com ($\alpha = 0.2$)



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*, **He2015** 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 gradiente em fuga, 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 nós vimos no texto.

Implementação em Python

8.4.3 Randomized Leaky ReLU

Anteriormente, na Parametric ReLU, nós tínhamos 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 **XuRReLU**, 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 $U(3, 8)$.

Ainda no artigo, **XuRReLU** investigam a performance de diferentes funções retificadoras em uma rede neural convolucional, cuja arquitetura pode ser vista na tabela **??**, para a classificação de imagens, os autores analisaram a ReLU tradicional, a Le-

aky 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 nas tabelas **??** e **??**.

Analisando a tabela **??**, que mostra a taxa de erro das funções retificadoras na rede NIN para o dataset CIFAR-10, nós podemos 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 analisarmos a tabela **??**, que nos mostra a taxa de erro dessas funções no dataset CIFAR-10, vemos que assim como no CIFAR-10, a RReLU foi a função que obteve melhor resultado, neste caso nós vemos 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.

Agora que entendemos um pouco de como a Randomized Leaky ReLU funciona, podemos entender melhor a sua fórmula, a qual é dada pela expressão **??**, 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 temos que $\alpha \sim U(l, u)$ em que $l < u$ e $l, u \in [0, 1)$

Randomized Leaky ReLU (RReLU)

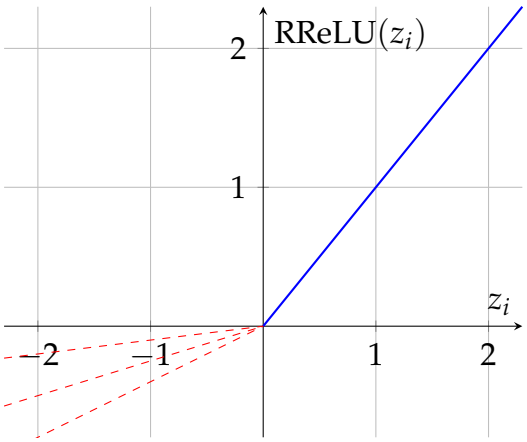
$$\text{RReLU}(z_i) = \begin{cases} z_i, & \text{se } z_i > 0 \\ \alpha_i z_i, & \text{se } z_i \leq 0 \end{cases} \tag{8.10}$$

Na fase de testes, devemos 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 **??** para calcular a RReLU durante o teste do modelo (**XuRReLU**).

Conhecendo como a Randomized Leaky ReLU se comporta, podemos plotar o seu gráfico, o qual está presente na figura **??**. 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 podemos 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, devemos nos 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 29 – Gráfico da Função Randomized Leaky ReLU (RReLU) com Diferentes Inclinações Aleatórias para a Parte Negativa



Se conhecemos com a RReLU se comporta podemos 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, nós podemos apenas repetir a expressão de sua derivada novamente, a qual será dada pela equação **??**. 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, nós podemos 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

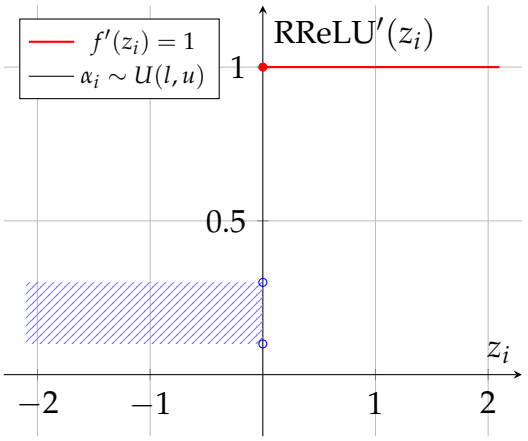
Derivada Randomized Leaky ReLU (RReLU)

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

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 nós podemos ver na

figura **??**, existem um conjunto de retas em um intervalo, neste caso, estamos considerando a distribuição como sendo de $U(0.1, 0.3)$, mas para cada uma dessas distribuições, teremos um conjunto de retas diferentes e com isso gráficos distintos para cada um dos problemas.

Figura 30 – Gráfico da Derivada da Função Randomized Leaky ReLU (RReLU) com $l = 0.1, u = 0.3$



Ainda em *Empirical Evaluation of Rectified Activations in Convolutional Network* **XuRReLU** 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 aleatoriedade afeta os processos de treino e teste (**XuRReLU**). 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.

Considerando as fórmulas da RReLU, podemos construir o seu bloco de código **??** que é capaz de implementar uma classe em python para atuar como a função Randomized Leaky ReLU.

Implementação em Python

8.5 Em Busca da Suavidade: As Variantes Não Lineares

Seguindo adiante, agora veremos 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 do neurônios agonizantes, mas também sendo uma função contínua em na origem e portanto derivável em todo o seu domínio.

Além disso, veremos uma variante da ELU, a Scaled Exponetial Linear Unit, uma função que é utilizada para construir redes capazes de se autonormalizarem, ademais

veremos 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 alogias 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 (**Djork**). 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 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 (**Djork**).

Com base nessa CNN e seus experimentos, é possível ver os resultados na tabela **??**, em que **Djork** comparam a ELU com outras redes no mesmo problema, como a AlexNet que vimos anterioremente na explicação do surgimento da ReLU. Ao analisarmos esses resultados, vemos 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 quando analisamos 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 nos indica que a ELU é uma excelente opção para problemas de classificação, especialmente se tivermos um grande número de classes a ser analisados. 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. Em uma das seções a frente, será possível comparar o desempenho das funções retificadoras as quais estamos vendo nesse texto, e a partir desses comparativos, veremos que talvez não seja uma opção interessante utilizar funções que fazem uso de exponenciais em suas fórmulas para resolver problemas de classificação mais simples, como o caso do dataset CIFAR-10.

Agora que entendendo que a ELU pode ser uma boa alternativa para problemas de classifacação de múltiplas classes, podemos conhecer como ela é escrita e como se comporta graficamente. A Exponential Linear Unit pode é descrita utilizando a expressão **??**. Note que temos uma grande diferença dela quando comparamos com a ReLU, a Exponential Linear Unit faz uso de funções exponenciais, algo que é computacionalmente mais pesado para um computador quando comparado com apenas cálculos simples como uma função identidade, podemos esperar que ela será mais lenta quando comparamos com a ReLU, mas para ter certeza precisaremos fazer compratívos, os quais serão vistos futuramente.

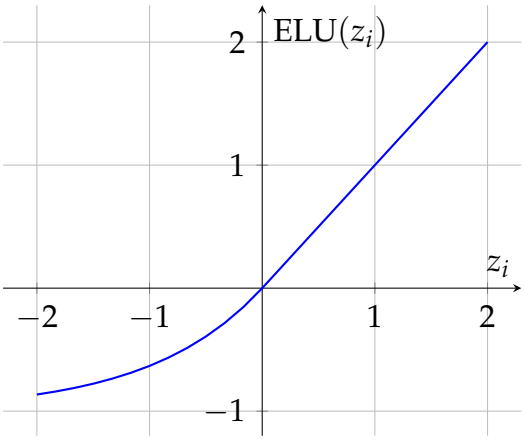
Exponential Linear Unit (ELU)

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

Conhecendo como é a fórmula da ELU, podemos também plotar o seu gráfico, o qual está presente na figura **??**. Ao analisarmos, vemos uma diferença notável quando o comparamos com as funções vistas anterioemente, a ELU não apresenta um bico no ponto de origem, ela é uma função bem mais suave. Além disso, vemos 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 analisamos a os casos em que a entrada é negativa, vemos que ela utiliza uma funcão exponencial, o que garante a suavidade vista no gráfico. Podemos notar 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 neurônios agonizantes causado pela ReLU tradicional e que vem sendo mitigado com outras variantes como a Leaky ReLU.

Já que sabemos a sua fórmula e o seu gráfico, podemos também calcular a sua derivada, a qual será útil na retropropagação do modelo. Para isso, seguimos a mesma estratégia vista até agora, derivamos 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 temos que nos preocupar em escolher um valor da derivada quando o seu valor de

Figura 31 – Gráfico da Função Exponential Linear Unit (ELU) com $\alpha = 1$



entrada for zero. Assim, temos como resultado final a expressão **??**

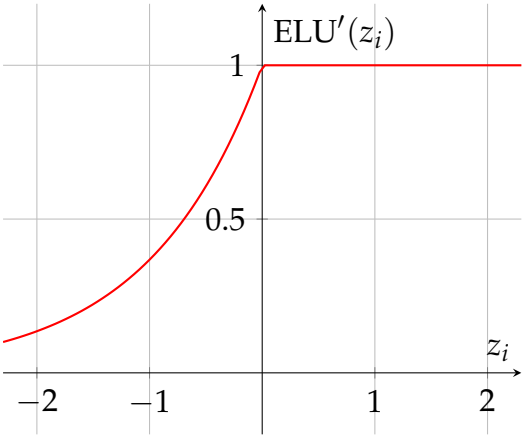
Para calcularmos sua derivada, basta derivamos as duas expressões da equação **??** encontrando então a função **??**.

Derivada Exponential Linear Unit (ELU)

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

Sabendo a sua derivada, podemos também plotar o seu gráfico, para isso, temos a figura **??**. 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, nós temos uma curva exponencial. Podemos notar também que a sua derivada irá sempre retornar valores positivos quando é calcula para qualquer ponto do seu domínio.

Figura 32 – Gráfico da Derivada da Função Exponential Linear Unit (ELU) com $\alpha = 1$



Por fim, cabe destacar algumas afirmações realizadas pelos autores ainda em *Fast and Accurate Deep Networks Leaning By Exponential Linear Units (ELUs)*, segundo **Djork**,

quando nós comparamos a ELU com funções como a ReLU tradicional e Leaky ReLU, nós podemos 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 com a Leaky ReLU e Parametric ReLU possuindo valores negativos, não são capazes de garantir ao serem utilizadas para construir uma rede neural.

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. Nós veremos essa função em seguida.

Implementação em Python

8.5.2 Scaled Exponential Linear Unit (SELU)

A próxima função que iremos conhecer 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 aritofo de sua introdução *Self-Normalizing Neural Networks* (**SELU**).

No texto, **SELU**, 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 dadatasets 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 **??**, é possível ver o ranking médio entre as SNNs e as outras diferentes arquiteturas em 75 tarefas de classificação.

Para analisar essa tabela, podemos 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, nós notamos 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 nos 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, se olharmos 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, vemos 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, nos mostrando que podemos considerar as SNNs como se tivessem empatadas com o campeão.

O interessante dessa comparação é olharmos 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 fuga do gradiente não ocorra com tanta frequência e com isso permitindo um melhor convergência do modelo que está sendo treinado. Se olharmos por essa ótica, podemos chegar a conclusão que criar uma rede neural utilizando a SELU não só ura garantir uma maior versatilidade para a resolução de problemas, como também não teremos que nos preocupar em aplicar técnicas de normalização ao construir essa rede.

Agora que temos uma noção maior de como a SELU pode ser uma opção interessante para criar uma RNA, podemos conhecer como ela funciona de fato. Para isso, os autores apresentam a fórmula **??** para calcular essa função, em que o termo λ é uma constante que será maior que 1 (**SELU**). 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 podemos escrever também que $\text{SELU}(z_i) = \lambda \text{ELU}(z_i)$.

Scaled Exponential Linear Unit (ELU)

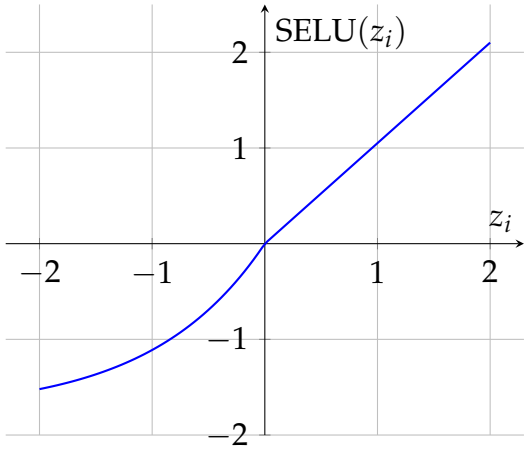
$$\text{SELU}(z_i) = \lambda \begin{cases} z_i, & \text{se } z_i > 0 \\ \alpha \cdot (e^{z_i} - 1), & \text{se } z_i \leq 0 \end{cases} \tag{8.14}$$

Conhendo sua equação, podemos também plotar o seu gráfico, o qual está presente na figura **??**, neste caso, estamos considerando que as constantes α e λ são dados por 1.7 e 1.05 respectivamente. Podemos notar 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 Dying ReLUs, causado pela ReLU, além de que também não é uma função saturante, como a sigmoide, o que também ajuda a resolver o problema da fuga dos gradientes. 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 33 – Gráfico da Função Scaled Exponential Linear Unit (SELU) com $\alpha \approx 1.67 e \lambda \approx 1.05$



Considerando agora que sabemos como é a equação da SELU e qual é o seu comportamento no gráfico, podemos também calcular a sua derivada para utilizarmos na retroproapagação do gradiente, para isso, devemos derivar a expressão **??**, considerando os dois cenários, em que a sua entrada será positiva e quando sua entrada for negativa ou zero. Um ponto que nos 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 nós já calculamos a derivada da ELU, podemos ver então a derivada da Scaled Exponential Linear Unit na equação **??**.

Derivada Scaled Exponential Linear Unit (ELU)

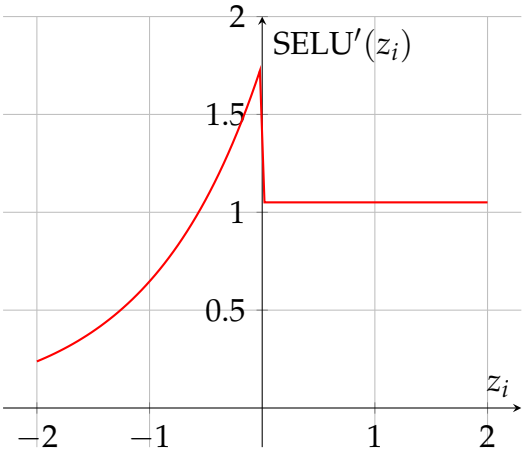
$$\frac{d}{dz_i}[SELU](z_i) = \lambda \begin{cases} 1, & \text{se } z_i > 0 \\ \alpha \cdot e^{z_i}, & \text{se } z_i \leq 0 \end{cases} \tag{8.15}$$

Se sabemos a sua derivada, podemos também plotar o seu gráfico, para isso, temos a representação na figura **??**. 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 fuga de 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 34 – Gráfico da Derivada da Função Scaled Exponential Linear Unit (SELU) com $\alpha \approx 1.67, \lambda \approx 1.05$



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 (**SELU**). 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.

Implementação em Python

8.5.3 Noisy ReLU

Seguindo adiante, a última variante da ReLU que iremos ver nessa seção das variantes suaves, é 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 **Nair2010**. Nesse texto, os autores comparam o desenpenho dessa função com a função binária, dada pela equação 8.16, que era a opção mais comum para ser utilizada na construção de máquinas restritas de Boltzmann.

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

Para entendermos como a comparação dessas funções ocorreu, é interessante primeiro conhecer o conceito das máquinas restritas de boltzmann.

> *As Máquinas de Boltzmann Restritas (RBMs) têm sido usadas como modelos generativos de muitos tipos diferentes de dados, incluindo imagens rotuladas ou não rotuladas, sequências de coeficientes mel-cepstrais que representam a fala, sacos de palavras que representam documentos e classificações de usuários de filmes. Em sua forma condicional, elas podem ser usadas para modelar sequências temporais de alta dimensão, como dados de vídeo ou captura de movimento. Seu uso mais importante é como módulos de aprendizagem que são compostos para formar redes de crenças profundas.*
>
> —(Nair & Hinton, 2010, p. 1, tradução nossa)

No trabalho, **Nair2010** exploram o desempenho dessas duas funções 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 utilização a vrsão 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 (**Nair2010**). O desempenho dessas funções nesse dataset pode ser visto pelas tabelas 5 e **??**.

A tabela 5, nos 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. Podemos notar 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, podemos 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 nos indica que podemos conseguir um resultado muito melhor utilizando a Noisy ReLU mediante a função binária mesmo quando não tivermos condições de pré-treinar uma rede.

Seguindo adiante, na tabela **??** 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 5, assim a primeira camada é composta de 4000 unidades ocultas (assim como no primeiro caso), enquanto a segunda camada possui 2000 unidades ocultas.

Tabela 5 – 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. 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 nós podemos 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 5. 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 nos indicar que talvez não seja vantajoso adicionar mais camadas em uma rede caso não estejamos dispostos a pré-treiná-las. Note também que esse cenário com a unidade binária foi ainda pior, nos mostrando que não temos um grande ganho em adicionar mais camadas em uma rede que faz uso dessa função.

Esse resultado prová-se ainda mais nítido quando vemos 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 observamos a Noisy ReLU e seus resultados, vemos 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 justamente ao optar por funções retificadoras como a ReLU e a Noisy ReLU ao invés de funções sigmoide, está intrinsicamente relacionado ao problema da fuga do gradiente. Em redes que faziam uso de funções sigmoide, 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 da fuga 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.

Sabendo de como a NReLU surgiu e sua importância para permitir a criação de redes mais profundas, podemos conhecer a sua fórmula e entender como ela funciona na prática. No texto, **Nair2010**, 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. Como neste texto estamos dando maior enfoque em uma notação com condicionais para as funções retificadoras, também podemos expressar a Noisy ReLU com a equação 8.17.

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

Antes de seguir em frente, é útil entende primeiro o que é ruído gaussiano, e para isso, precisamos entender antes a distribuição gaussiana. Segundo **Goodfellow2016**, a distribuição gaussiana é a distribuição mais utilizada para números reais, ela também é conhecida também por ser chamada de distribuição normal, ela é dada pela equação 8.18.

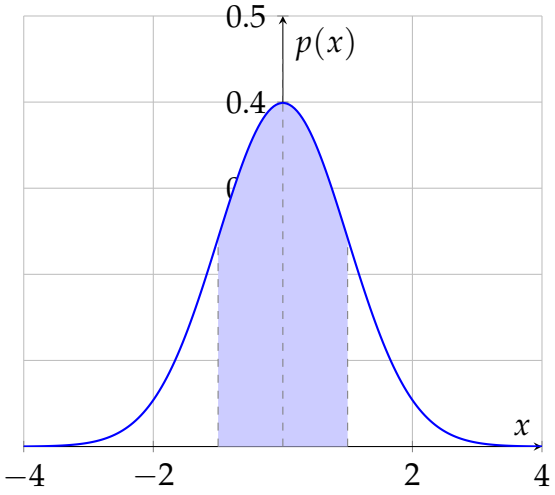
$$\mathcal{N}(x; \mu, \sigma^2) = \sqrt{\frac{1}{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right) \tag{8.18}$$

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 (**Goodfellow2016**). Para chegarmos no ruído gaussiano, precisamos 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 nos dar um gráfico em formato de sino, como é mostrado na figura 35. 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.

Entendendo a estrutura e como funciona a Noisy ReLU, podemos também plotar o sey gráfico, o qual é dado pela figura **??**. Vemos 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 Dying ReLU problem. 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 (**Nair2010**).

Figura 35 – Gráfico da Distribuição Gaussiana (ou Normal) para o caso padrão, com média 0 ($\mu = 0$) e desvio padrão 1 ($\sigma = 1$).



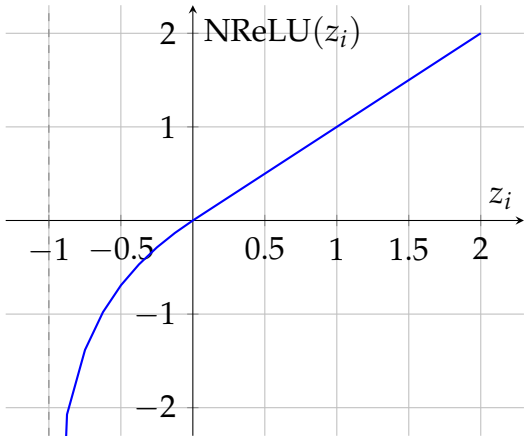
Noisy ReLU (NReLU)

$$\text{NReLU}(z_i) = \begin{cases} 0 & \text{se } z_i \leq 0 \\ z_i + \mathcal{N}(0, \sigma(z_i)) & \text{se } z_i > 0 \end{cases} \tag{8.19}$$

Entendendo a estrutura e como funciona a Noisy ReLU, podemos também plotar o sey gráfico, o qual é dado pela figura **??**. Vemos 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 Dying ReLU problem. 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 (**Nair2010**).

Quando vamos calcular a sua derivada, encontramos 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 (**Nair2010**). Então a expressão que representa a NReLU para a retroproagação é dada pela equação **??**, note que ela é a mesma expressão da derivada da ReLU tradicional.

Figura 36 – Gráfico da Função Noisy ReLU (NReLU)

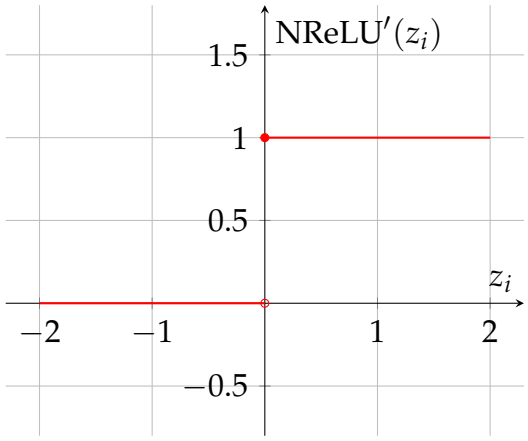


Derivada Noisy ReLU (NReLU)

$$\frac{d}{dz_i}[\text{NReLU}](z_i) = \begin{cases} 0 & \text{se } z_i \leq 0 \\ 1 & \text{se } z_i > 0 \end{cases} \tag{8.20}$$

Se a função da Noisy ReLU no backward pass será a mesma que a derivada da ReLU, podemos então utilizar como base o gráfico da derivada da ReLU. Para isso, temos então a figura **??**.

Figura 37 – Gráfico da função Backward Pass da Noisy ReLU



Ainda em *Rectified Linear Units Improve Restricted Boltzmann Machines*, **Nair2010** citam que uma das propriedades interessantes da NReLU é a intensity equivarience (), 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 fubções que quando adicionada em uma rede neural, garante que isso possa ser atingido (**Nair2010**).

Implementação em Python

8.6 O Problema dos Gradientes Explosivos

Anteriormente, nós conhecemos as funções sigmoides, vimos que elas eram comumente utilizadas como as funções de ativação padrão de uma rede neural antes das funções retificadoras. Mas elas possuíam um problema, o do gradiente em fuga. 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 gradiente em fuga.

Já nessa seção nós conhecemos a ReLU, e vimos que ela veio como uma alternativa para as funções sigmoides e que era capaz de corrigir esse problema. Contudo, ela também possui problemas, pelo fato de ser uma função que não é suturante como as sigmoides, isso faz com que ela possa gerar o problema da explosão do gradiente, para entendê-lo primeiro precisamos nos lembrar da fórmula do gradiente retropropagado.

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

O primeiro termo, o que possui o somatório, é responsável por calcular o erro total que chega para um neurônio i da camada atual, para isso temos a equação 8.21.

$$\text{Erro}_i = \left(\sum_j \frac{\partial E}{\partial x_j} \cdot w_{ji} \right) \tag{8.21}$$

$\partial E / \partial x_j$ nos indica o gradiente do erro em relação à saída de um neurônio j que está na camada seguinte da rede. Para nós calcularmos ele, devemos aplicar a regra da cadeia, assim $\partial E / \partial x_j$ será dado pelo cálculo do erro vindo depois da camada j, uma camada k por exemplo, assim, o erro será dado pela equação 8.22.

$$\text{Erro}_j = \left(\sum_k \frac{\partial E}{\partial x_k} \cdot w_{kj} \right) \tag{8.22}$$

Note que temos um padrão nos cálculos dos erros.

1. Para que nós possamos calcular o erro de uma camada $w_b a$ devemos encontrar o Erro_b;

2. Para encontrar o Erro_b, precisamos calcular o gradiente da camada seguinte: $\partial E / \partial x_c$

3. O gradiente ($\partial E/\partial x_c$) é calculado a partir do Erro$_c$

4. Para calcular o Erro$_c$, precisamos da camada seguinte e seu gradiente $\partial E/\partial x_d$ que envolve os pesos $w_d e$

5. E assim por diante até a última camada da rede.

Com base nessa análise podemos encontrar uma expressão que será capaz de nos mostrar qual será o valor do gradiente retropropagado agora considerando as atualizações de pesos que é feita na rede, ela é dada pela equação 8.23.

$$\delta^l \propto (W^{l+1})^T \cdot (W^{l+2})^T \cdots (W^L)^T \cdot \delta^L \tag{8.23}$$

Note que, com base nessa expressão, se um cenário em que sempre houver valores muito grandes nos pesos das camadas da rede, ou ate mesmo muitas camadas, o gradiente que chegará na última camada poderá ter um valor muito alto, podendo atrapalhar o desempenho da rede neural e como ela irá aprender, assim temos um sério problema ao construir uma rede.

Esse tipo de problema, é chamado de problema da explosão do gradiente e ele pode ser definido da seguinte forma:

> 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.

(**explodingGradient**)

Assim, mesmo a ReLU corrigindo o problema do gradiente em fuga, 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.

8.7 Comparativo de Desempenho das Funções Retificadoras

Além do problema do explosão de gradientes que pode ocorrer ao utilizar a ReLU em uma rede neural, essa função também sofre de outro problema crônico: o do neurônios agonizantes (ou *Dying ReLU Problem* em inglês). Esse problema pode ser definido

como uma variação do problema do gradiente em fuga, em que uma parte dos neurônios da rede morrem e passam a retornar somente zero independente de sua entrada (**dyingReLU**).

Para entendermos melhor devemos nos lembrar da estrutura básica de uma camada densa de neurônios de uma rede neural, de forma que ela pode ser vista na equação 8.24, em que W representa um vetor de pesos para os neurônios dessa camada, X é o vetor de elementos de entrada e b é o viés.

$$y = W^T X + b \tag{8.24}$$

Se estivemos utilizando a ReLU como função de ativação dessa camada, essa variável y irá passar para uma função $\max(0, y)$, podem existir conjuntos de dados que quando passados por essa camada, a condição mais comum de ocorrer será a de $y < 0$ para os diferentes padrões de treinamento, isso tem efeito no *backward pass*, em que vamos utilizar a derivada da ReLU, que também será zero para casos em que $y < 0$, e nós sabemos que com a derivada dessa função nós calculamos o gradiente que irá calcular o como o peso da rede irá mudar de uma época para outra, se essa saída for zero, quando ela for multiplicada ao calcular o gradiente ele também será zero, e consequentemente os pesos e vieses da rede neural não serão atualizados, se eles não são atualizados, o neurônio não aprende, tendo então o problema do neurônios agonizantes (**douglasDyingRelu**).

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 (**douglasDyingRelu**).

Agora que conhecemos a ReLU, vimos como ela surgiu e se popularizou, além de suas propriedades e problemas, podemos conhecer as suas variações, as quais buscam corrigir alguns problemas que a função original possui mas mesmo assim mantendo as sua essência como inspiração para a criação de uma função melhor. Primeiro, conheceremos as variações com vazamentos, começando pela Leaky ReLU ou LReLU.

9 Funções de Ativação Modernas e Outras Funções de Ativação

9.1 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 **gelu**, 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 (**gelu**). Além disso, **gelu**, 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 ru-

ído, 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 (**gelu**). 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.

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

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 9.2 e 9.3 (**gelu**). 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

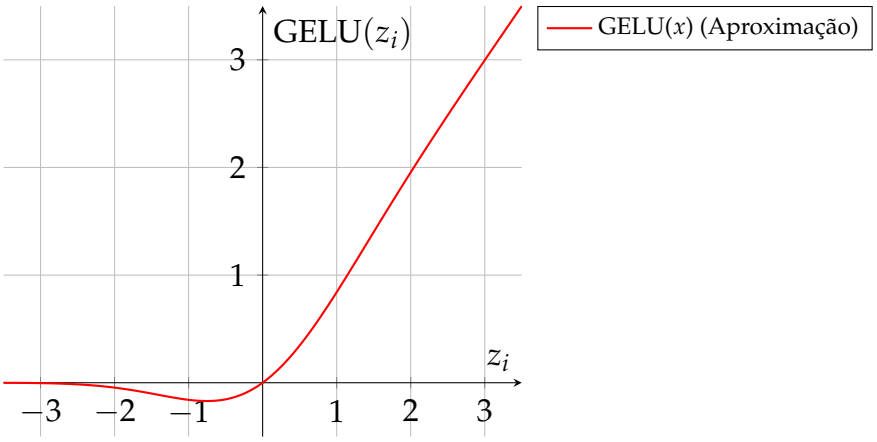
uma maior velocidade de processamento por apresentar cálculos mais simples em sua composição.

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

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

Se sabemos a sua fórmula, podemos também plotar o seu gráfico, para isso temos a figura 38, 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.

Figura 38 – Gráfico da Função de Ativação GELU (usando aproximação).



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.

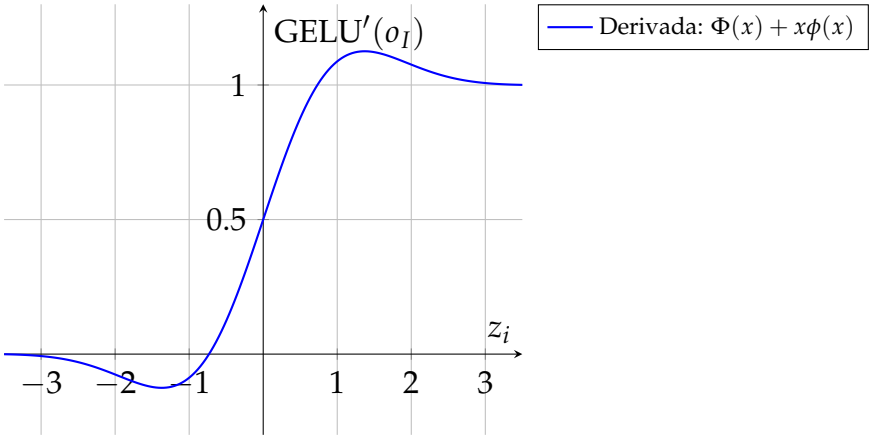
$$\frac{d}{dz_i}[\text{GELU}](z_i) = \Phi(z_i) + z_i\phi(z_i) \tag{9.4}$$

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

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 39. 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 39 – Gráfico da Derivada da Função de Ativação GELU



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 recomendado o uso de um otimizador com momentum quando estiver treinando uma rede com a GELU (**gelu**). Em segundo lugar, **gelu**, 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.

10 Funções de Perda para Classificação Binária

11 Funções de Perda para Classificação Multilabel

11.1 Softmax e a Distribuição de Probabilidades

11.2 Entropia Cruzada Categórica (Categorical Cross-Entropy)

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

12 Metaheurísticas: Otimizando Redes Neurais Sem o Gradiente

O texto do seu capítulo começa aqui...

12.1 Algoritmos Evolutivos

12.2 Inteligência de Enxame

Parte IV

Aprendizado de Máquina Clássico

13 Técnicas de Regressão

13.1 Exemplo Ilustrativo

13.2 Regressão Linear

13.2.1 Função de Custo MSE

13.2.2 Equação Normal

13.2.3 Implementação em Python

13.3 Regressão Polininomial

13.3.1 Impletanção em Python

13.4 Regressão de Ridge

13.4.1 Implementação em Python

13.5 Regressão de Lasso

13.5.1 Implementação em Python

13.6 Elastic Net

13.6.1 Implementação em Python

13.7 Regressão Logística

13.7.1 Implementação em Python

13.8 Regressão Softmax

13.8.1 Implementação em Python

13.9 Outras Técnicas de Regressão

14 Árvores de Decisão e Florestas Aleatórias

15 Máquinas de Vetores de Suporte

15.1 Exemplo Ilustrativo

16 Ensamble

16.1 Exemplo Ilustrativo

17 Dimensionalidade

17.1 Exemplo Ilustrativo

17.2 A Maldição da Dimensionalidade

17.3 Seleção de Características (Feature Selection)

17.4 Extração de Características (Feature Extraction)

17.4.1 Análise de Componentes Principais (PCA)

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

18 Clusterização

Parte V

Redes Neurais Profundas (DNNs)

19 Perceptrons MLP - Redes Neurais Artificiais

O texto do seu capítulo começa aqui...

20 Redes FeedForward (FFNs)

O texto do seu capítulo começa aqui...

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

O texto do seu capítulo começa aqui...

22 Redes Neurais Convolucionais (CNN)

22.1 Exemplo Ilustrativo

22.2 Camadas Convolucionais: O Bloco Fundamental para as CNNs

22.2.1 Implementação em Python

Bloco de Código : Classe completa de Convolution2D

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

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

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

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

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

22.3 Camadas de Poooling: Reduzindo a Dimensionalidade

22.3.1 Max Pooling

22.3.1.1 Implementação em Python

Bloco de Código : Classe completa de MaxPooling2D

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

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

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

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

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

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

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

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

```

22.3.2 Average Pooling

22.3.2.1 Implementação em Python

Bloco de Código : Classe completa de AveragePooling2D

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

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

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

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

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

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

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

        return output_matrix

```

22.3.3 Global Average Pooling

22.3.3.1 Implementação em Python

Bloco de Código : Classe completa de GlobalAveragePooling2D

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

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

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

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

        return output

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

        distributed_grad = output_gradient / (input_h *
    input_w)

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

        return upsampled_grad, None
```

22.4 Camada Flatten: Achatando os Dados

22.4.1 Implementação em Python

Bloco de Código : Classe completa de Flatten

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

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

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

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

        return flatten_output

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

22.5 Criando uma CNN

22.6 Detecção de Objetos

22.7 Redes Totalmente Convolucionais (FCNs)

22.8 You Only Look Once (YOLO)

22.9 Algumas Arquiteturas de CNNs

22.9.1 LeNet-5

22.9.2 AlexNet

22.9.3 GoogLeNet

22.9.4 VGGNet

22.9.5 ResNet

22.9.6 Xception

22.9.7 SENet

23 Redes Residuais (ResNets)

O texto do seu capítulo começa aqui...

24 Redes Neurais Recorrentes (RNN)

O texto do seu capítulo começa aqui...

24.1 Exemplo Ilustrativo

24.2 Neurônios e Células Recorrentes

24.2.1 Implementação em Python

24.3 Células de Memória

24.3.1 Implementação em Python

24.4 Criando uma RNN

24.5 O Problema da Memória de Curto Prazo

24.5.1 Células LSTM

24.5.2 Conexões Peephole

24.5.3 Células GRU

25 Técnicas para Melhorar o Desempenho de Redes Neurais

26 Transformers

27 Redes Adversárias Generativas (GANs)

O texto do seu capítulo começa aqui...

28 Mixture of Experts (MoE)

O texto do seu capítulo começa aqui...

29 Modelos de Difusão

O texto do seu capítulo começa aqui...

30 Redes Neurais de Grafos (GNNs)

O texto do seu capítulo começa aqui...

Parte VI

Apêndices

Referências

CAUCHY, Augustin-Louis. Méthode générale pour la résolution des systèmes d'équations simultanées. *Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences*, v. 25, p. 536–538, 1847. Citado na p. 28.

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. 68.

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. Citado nas pp. 59–61.

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. Citado nas pp. 50–52.

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, 68, 69.

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. Citado nas pp. 54–58.

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. 69.

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. Citado nas pp. 62, 63.

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. Citado na p. 61.

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.

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. 44, 45.

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. 46.

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. 33–35, 37, 39, 43–45.

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. 46.