ANA RAQUEL

PÓS TECH

DATA ANALYTICS

DEEP & REINFORCEMENT LEARNING

# AULA 01

**SUMÁRIO**

## O QUE VEM POR AÍ?

A natureza inspirou muitas criações para os seres humanos, tais como as aves nos inspiraram a voar com os aviões, a planta bardana inspirou a criação do velcro e o cérebro humano inspirou a criação das chamadas redes neurais artificiais (RNA), consideradas como aprendizado profundo. As redes neurais artificiais são poderosas e escaláveis, sendo muito utilizadas para lidar com grandes tarefas altamente complexas do aprendizado de máquina, tais como classificação de imagens, reconhecimento de fala, criação de imagens, chats poderosos e até mesmo aprender a jogar videogames. Nessa aula, você aprenderá como funcionam as redes neurais, incluindo a arquitetura da rede neural de múltiplas camadas (Multilayer Perceptron). Vamos lá?

## HANDS ON

Não iremos apenas entender o funcionamento das redes multicamadas, vamos aprender, na prática, com Python, como ela funciona e toda a sua arquitetura! Utilizaremos a aplicação da rede usando a biblioteca do Keras. Tenho certeza que você usufrui de diversos recursos no seu dia a d que utilizam a biblioteca Keras, tais como YouTube, Netflix, Uber, Yelp, Instacart, Zocdoc, Twitter, Square/Block e muitos outros. Na pesquisa de 2022 "State of Data Science and Machine Learning" da Kaggle, Keras teve uma taxa de adesão de 61% entre as pessoas cientistas de dados. Vamos para o hands-on?

O código desta aula você encontra aqui: códigos da aula 01.

A base de dados utilizada nas aulas também está disponível: base de dados da aula 01.

## SAIBA MAIS

### Entendendo a inspiração das redes neurais artificiais

As redes neurais artificiais foram introduzidas há muito tempo, em 1943, pelo neurofisiologista Warren McCulloch e pelo matemático Walter Pitts, em um artigo com o título “Logical Calculus of Ideas Immanent in Nervous Activity”. Você pode checá-lo, na íntegra, no link a seguir.

**Artigo “Logical Calculus of Ideas Immanent in Nervous Activity”**

Nesse documento, McCulloch e Pitts apresentam um modelo computacional simplificado utilizando lógica proposicional de como os neurônios biológicos podem trabalhar juntos em cérebros de animais na realização de cálculos complexos.

### Neurônios biológicos

Para entendermos como funcionam as redes neurais artificiais, primeiro iremos entender como funciona um neurônio biológico. O neurônio é uma célula encontrada principalmente no córtex cerebral, composto por um corpo celular contendo o núcleo, a maioria dos componentes complexos da célula e muitas extensões de ramificação chamadas de dendritos, além de uma extensão muito longa chamada axônio.

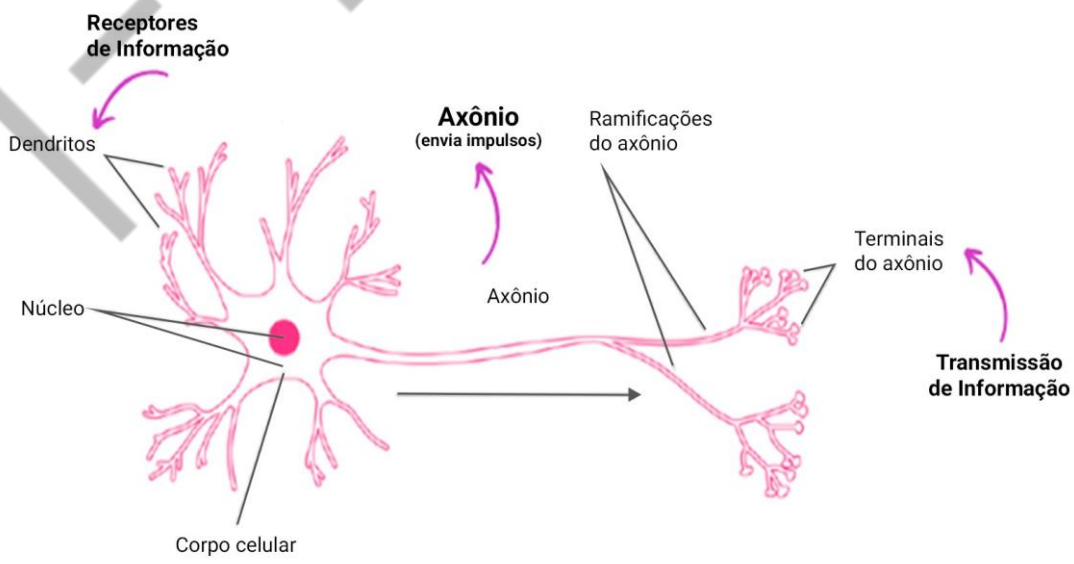


Figura 1 - Neurônio biológico
Fonte: elaborado pela autora (2023)

Perto de sua extremidade, o axônio divide-se em muitos ramos chamados de **telodendros,** e na ponta desses ramos estão estruturas minúsculas chamadas **terminais sinápticos** (ou **sinapses**) que são conectados aos dendritos de outros neurônios.

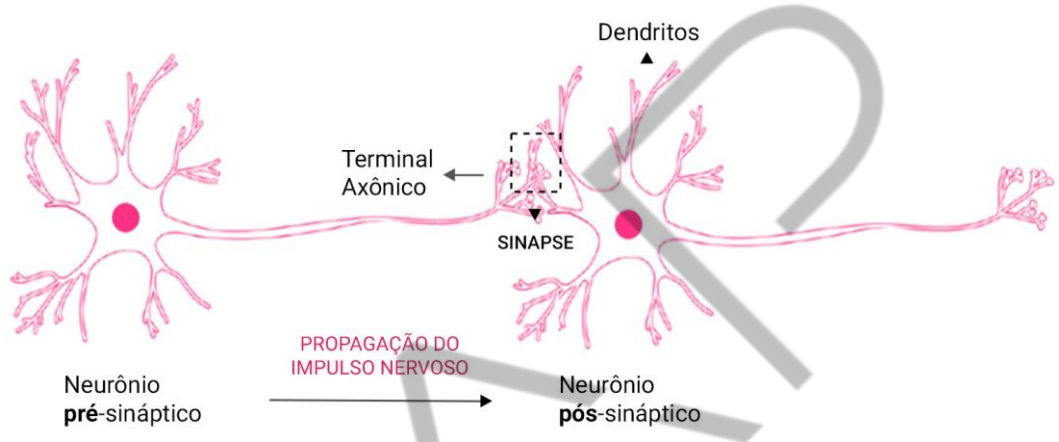


Figura 2 - Sinapse – Comunicação entre neurônios.
Fonte: (2023)

Os neurônios biológicos recebem curtos impulsos elétricos de outros neurônios através dessas sinapses chamadas sinais. É por meio das sinapses que os neurônios transmitem as informações de um para outro. Agora, imagine uma rede vasta de bilhões de neurônios conectados a milhares de outros neurônios? Essa é a ideia da rede neural!

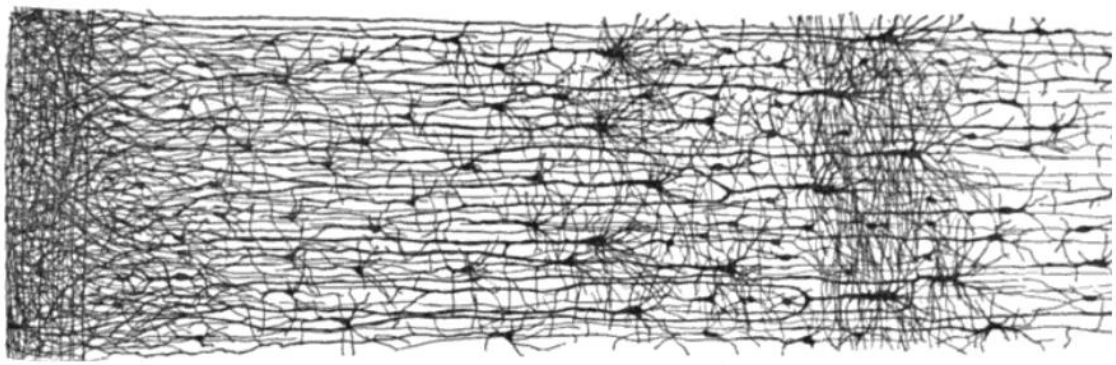

Figura 3 - Múltiplas camadas em uma rede neural biológica (córtex humano).
Fonte: Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow (2023)

Com base no comportamento do neurônio, McCulloch e Pitts propuseram um modelo simples conhecido como **neurônio artificial**, possuindo uma ou mais entradas binárias (ligado/desligado).

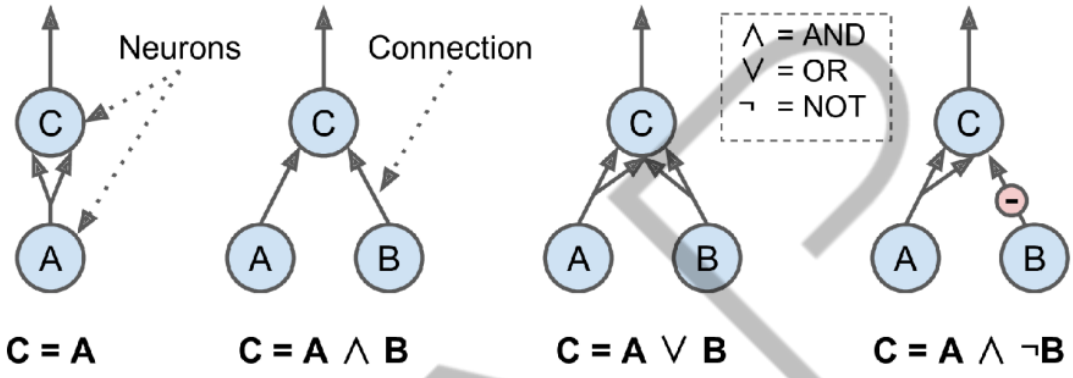


Figura 4 - Representação do funcionamento do neurônio artificial com lógica simples.
Fonte: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2023)

Logo em 1957, inspirado nos trabalhos anteriores, Frank Rosenblatt inventou o modelo de **neurônio Perceptron**, uma arquitetura bem simples de RNA:

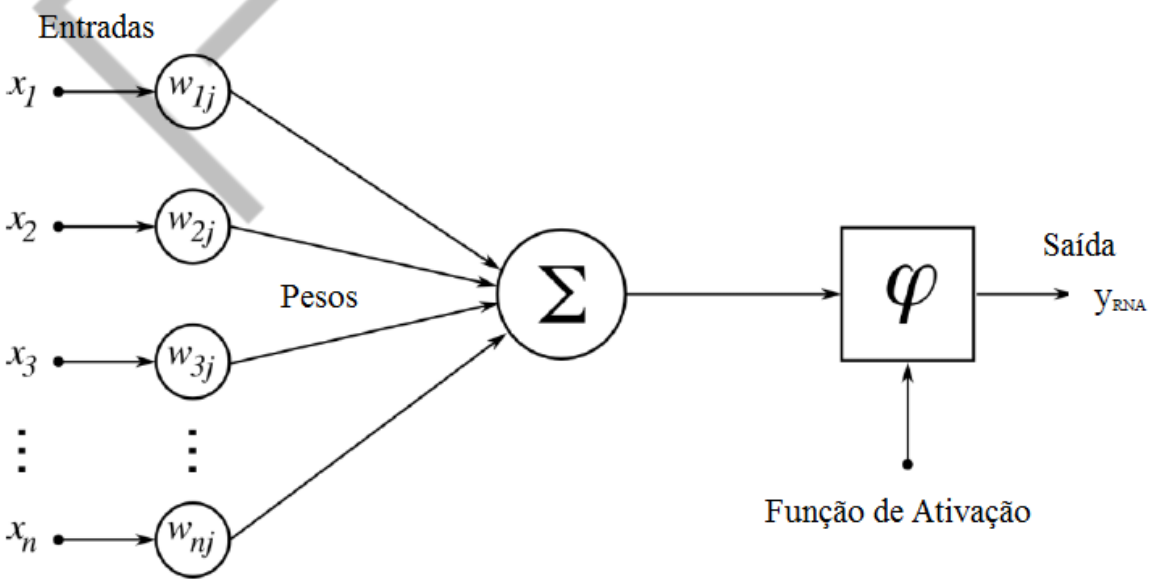


Figura 5 - Representação do neurônio artificial.
Fonte: Ricardo P. Ferreira (2016)

**Entendendo o neurônio Perceptron**

Iremos começar a entender o comportamento no neurônio Perceptron com a imagem proposta por Frank Rosenblatt, presente na figura 6.

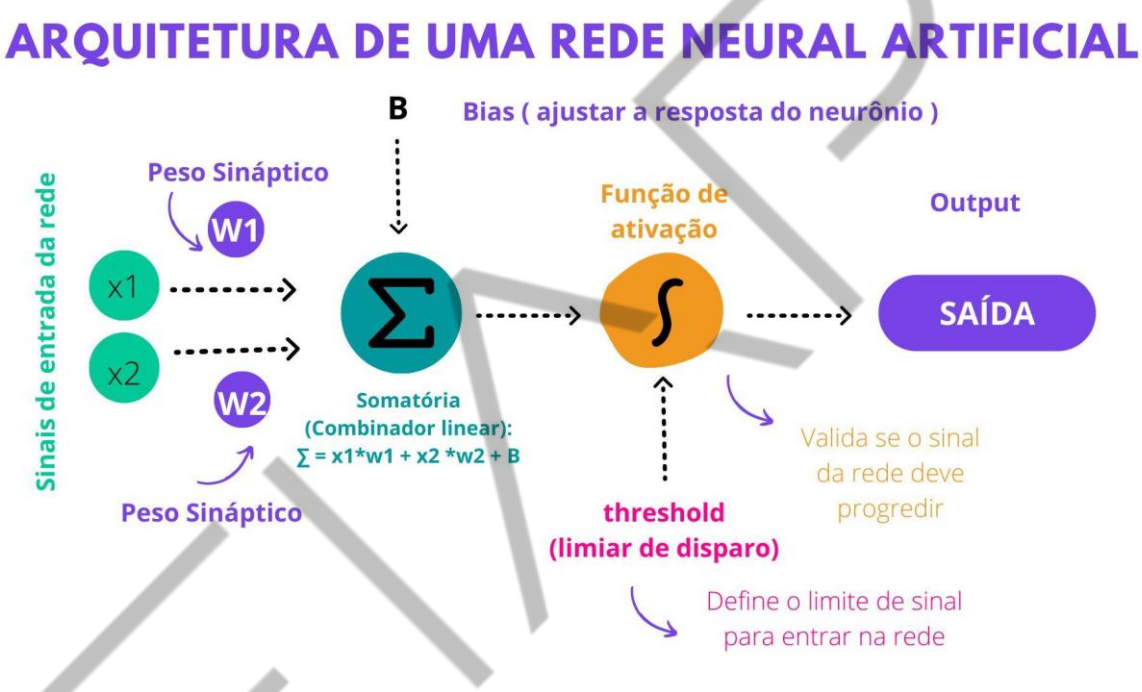


Figura 6 - Neurônio artificial perceptron.
Fonte: elaborado pela autora (2023)

O Perceptron funciona da seguinte forma: as entradas **“X1”** e **“X2”** podem ser **variáveis** (**números**) que são associados aos pesos **“W1”** e **“W2”**, que podemos nomear como **“pesos sinápticos”**. Após cada entrada ser associado a um peso, essas variáveis são inseridas na etapa de **LTU (unidade linear com threshold)**, calculando a soma ponderada das duas entradas, e aplica uma **função de ativação** que avalia se esse resultado, após a função realizada sob a somatória, deve ir para a saída da rede com base em um valor **limiar de decisão**. A função de ativação mais utilizada em Perceptrons é a função **Heaviside**.

$$\text{heaviside}\,(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

Figura 7 - Função Heaviside.
Fonte: Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow (2023)

Uma única LTU pode ser utilizada para classificação binária linear simples. Ela calcula uma combinação linear das entradas e, se o resultado exceder um limiar, exibe a classe positiva ou negativa.

Podemos listar algumas características desse modelo de Perceptron:

- O modelo possui uma natureza **binária (0 e 1)**. Tanto os sinais de entrada, quanto a saída, são valores binários. O Perceptron é um **classificador linear (binário)**.
- Os **pesos** da rede neural podem ser **ajustáveis**, inspirados em sinapses, podendo ser excitatórias ou inibitórias (positivos ou negativos).
- É utilizado na **aprendizagem supervisionada** e pode ser usado para classificar os dados de entrada fornecidos.
- Classifica a entrada separa duas categorias com uma linha reta.

Que tal aprender o comportamento do Perceptron com um exemplo simples de seu comportamento no mundo real? Vamos lá! Suponha que você quer muito ir em um show e existem 3 fatores importantes para tomar a sua decisão:

- O show é em outro estado?
- Você tem companhia para ir no show?
- Você tem dinheiro?

Imaginemos que esses fatores são nossas variáveis binárias para a entrada de nossa rede neural artificial Perceptron:

- x1 = O show é em outro estado?
- x2 = Você vai ter companhia para o show?
- x3 = Você tem dinheiro?

As variáveis X1, X2 e X3 são representadas por valores binários, por exemplo:

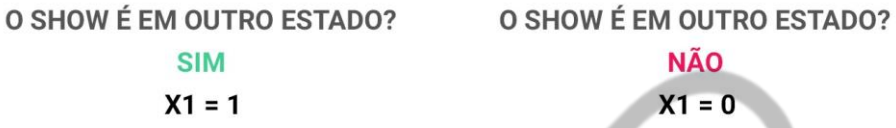


Figura 8 - Exemplo de entrada de um modelo perceptron.
Fonte: elaborado pela autora (2023)

Já adianto que os pesos de uma rede são atribuídos inicialmente de forma aleatória, e são os pesos que impulsionam a força de conexão da rede. Vamos supor, nesse nosso exemplo didático, que os pesos das variáveis foram distribuídos das seguintes formas:

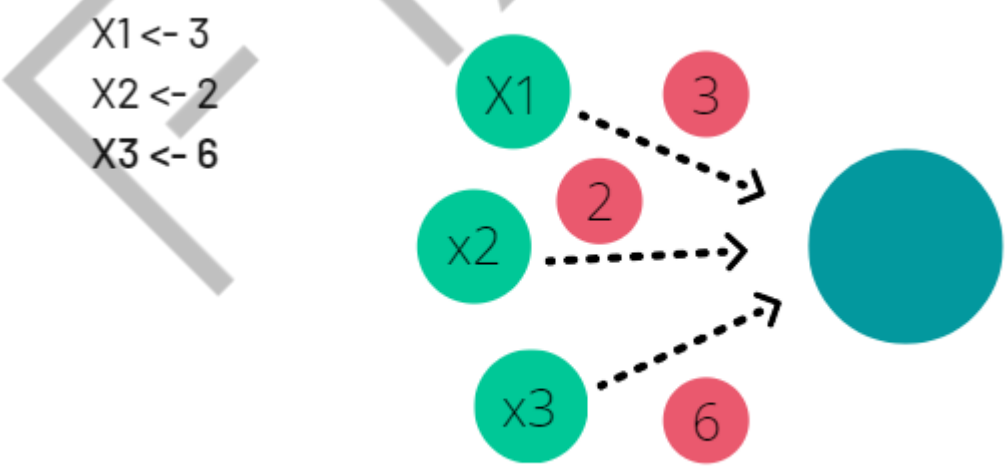


Figura 9 - Atribuição de pesos no modelo perceptron.
Fonte: elaborado pela autora (2023)

Suponha que você definiu o threshold como 5 para o Perceptron e que, inicialmente na nossa rede, temos apenas a variável “X1” = 1 , “X2” = 0 e “X3” = 0.

Quando os números das entradas são atribuídos aos pesos e passados pela função de ativação, com base no limiar de decisão da função, a rede conclui que nesse cenário você pode receber um output 0, concluindo que você não irá no show com base nos cálculos realizados na rede Perceptron.

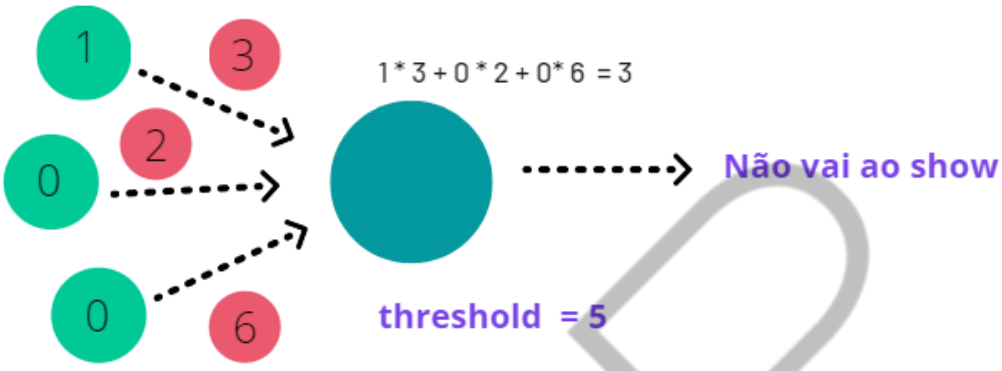


Figura 10 - Modelo Perceptron.
Fonte: elaborado pela autora (2023)

Ao alterar limiar de decisão, podemos ter várias saídas diferentes:

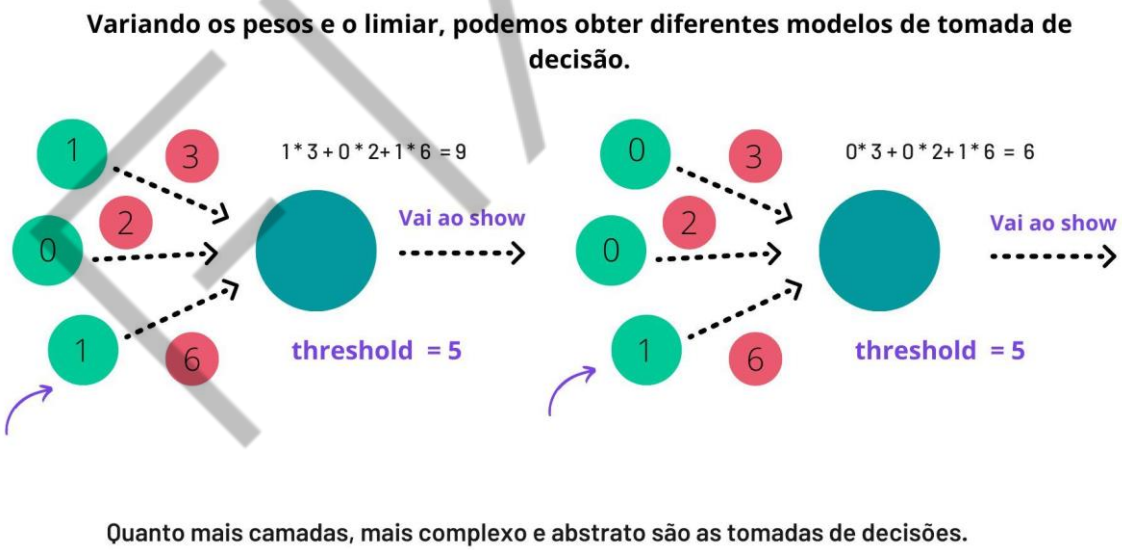


Figura 11 – Modelo Perceptron com vários limiares de decisão.
Fonte: elaborado pela autora (2023)

O Perceptron segue o modelo **feed-foward**. Nesse tipo de modelo, as entradas da rede são enviadas para o neurônio, em seguida são processadas e resultam em

uma saída. Um único Perceptron pode resolver problemas lineares, a questão é que os dados do mundo real **não são lineares**, fazendo com que esse tipo de neurônio artificial não seja muito útil para solucionar problemas não lineares.

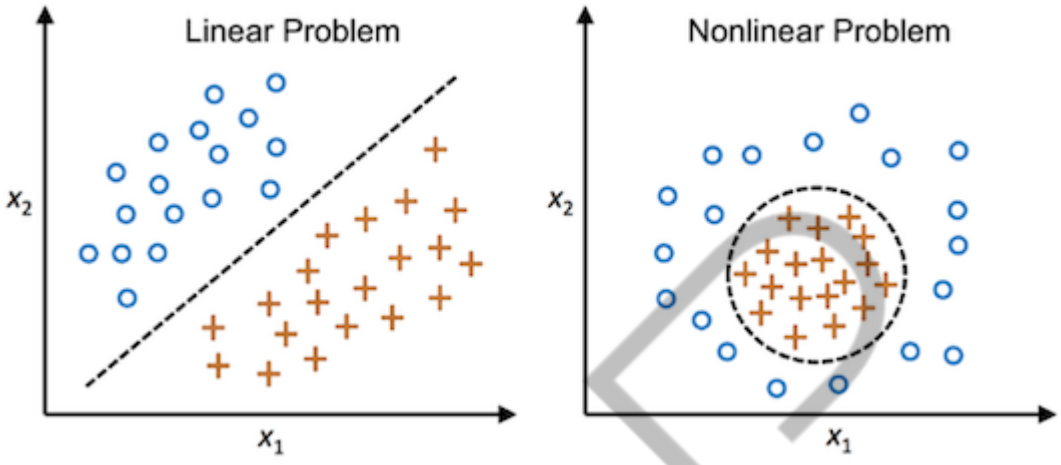


Figura 12 - Modelos lineares x não lineares.
Fonte: CMD Line Tips (2021)

Talvez você se pergunte: “Como resolvemos problemas não lineares com redes neurais?”. É aí que entram as redes artificiais multicamadas.

**Redes Neurais Multicamadas**

Podemos considerar as redes neurais multicamadas como um aprendizado supervisionado e diferente do Perceptron. Uma rede neural de multicamadas constrói várias camadas ocultas, modelando assim a correlação (ou dependências) entre os dados de entrada com os de saída. O seu treinamento envolve o ajuste dos parâmetros, ou pesos e bias do modelo, para minimizar o erro.

Basicamente, seu funcionamento consiste no algoritmo alimentar cada instância de treinamento para a rede calcular a saída de cada neurônio em cada camada consecutiva. Em seguida, ele **mede o erro de saída da rede** (isto é, a diferença entre a saída desejada e a saída real da rede) e **calcula o quanto cada neurônio contribui para o erro** de cada neurônio de saída na última camada oculta. Como próximo passo, o modelo passa a medir a quantidade dessas contribuições de erro provenientes de cada neurônio na camada oculta anterior, e assim por diante, até o algoritmo alcançar a camada de entrada. Esta passagem reversa (também

conhecida como **backpropagation**) mede a **eficiência do gradiente de erro em todos os pesos de conexão da rede**.

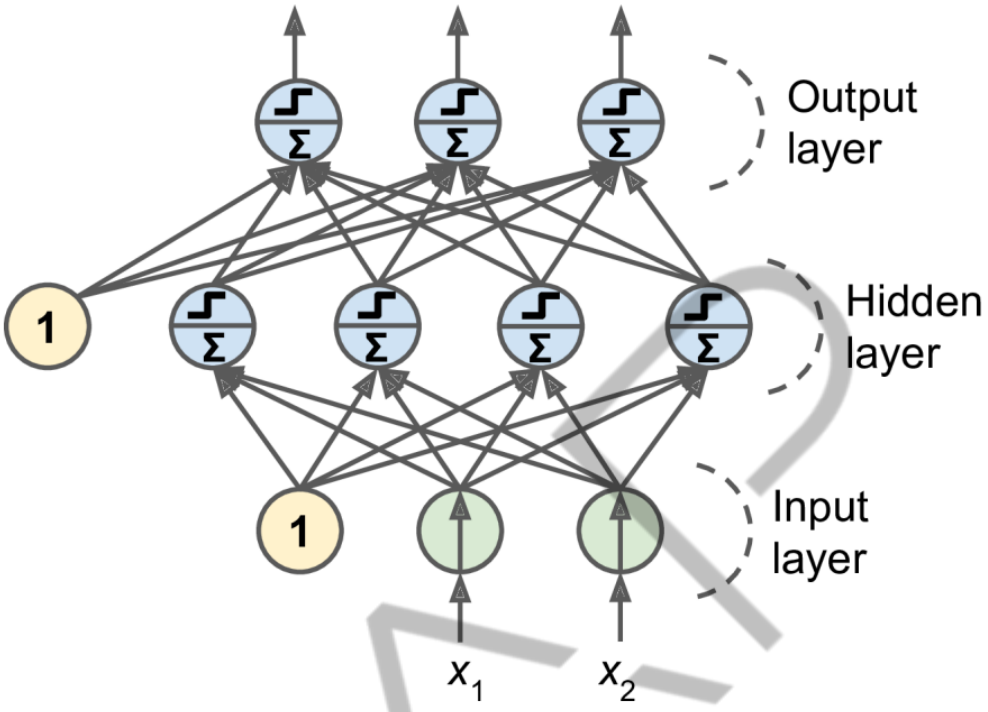


Figura 13 - Modelos de rede neural multicamadas.
Fonte: Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow (2023)

## ENTENDENDO O MODELO BACKPROPAGATION

O modelo de backpropagation é utilizado para fazer os ajustes dos pesos e de bias dos neurônios em relação ao erro. As redes neurais multicamadas basicamente funcionam utilizando a propagação e a retropropagação.

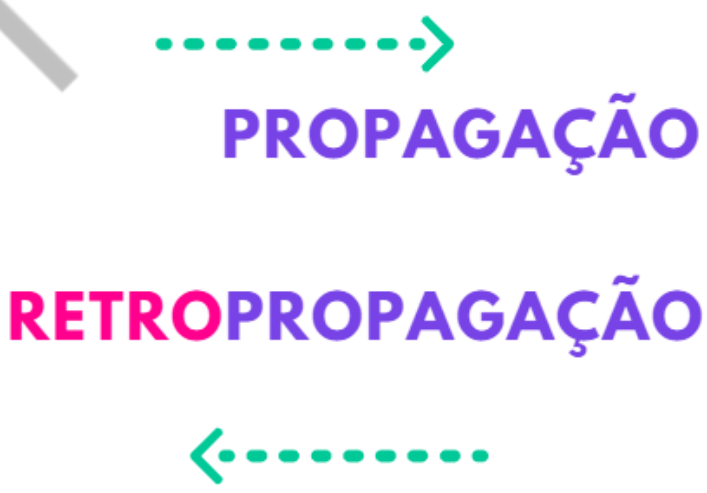

Figura 14 - Modelo Backpropagation.
Fonte: elaborado pela autora (2023)

Para você compreender melhor esse modelo, acompanhe a explicação das próximas imagens. Imagine que vamos ter a entrada das variáveis x1 e x2 na primeira camada da rede, a **Input Layer**:

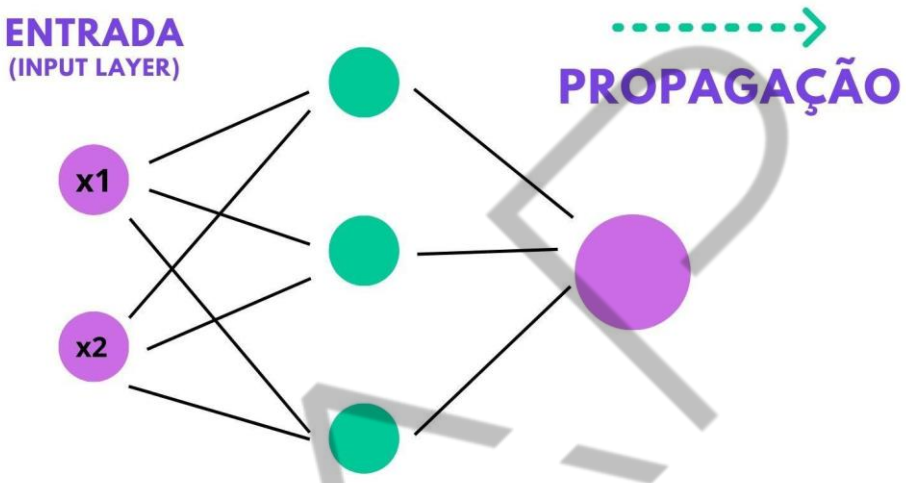


Figura 15 - Explicação Backpropagation: Camada de entrada.
Fonte: elaborado pela autora (2023)

Ainda na camada de entrada, essas variáveis vão ter pesos atribuídos a cada uma delas para o neurônio ganhar força e prosseguir para a próxima camada:

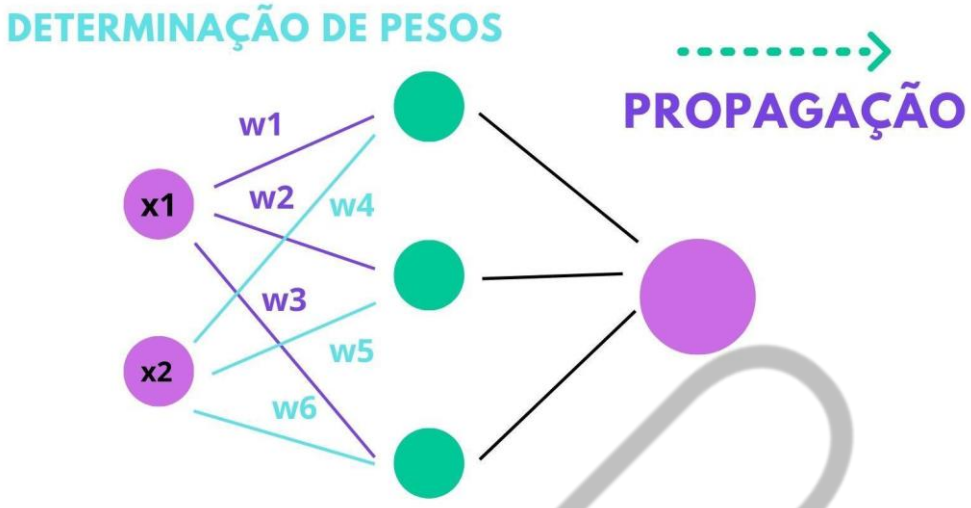


Figura 16 - Explicação Backpropagation: Atribuição de pesos.
Fonte: elaborado pela autora (2023)

Essas entradas vão passar pela camada conhecida como “Hidden Layer” (camada oculta). Nesse exemplo, é possível visualizar apenas uma camada para simplificar a explicação, porém imagine que podemos ter muitas camadas ocultas dentro de uma rede neural multicamadas. Em cada camada oculta, o cálculo do somatório, incluindo variáveis e pesos, é realizado e aplicado sobre uma função de ativação.

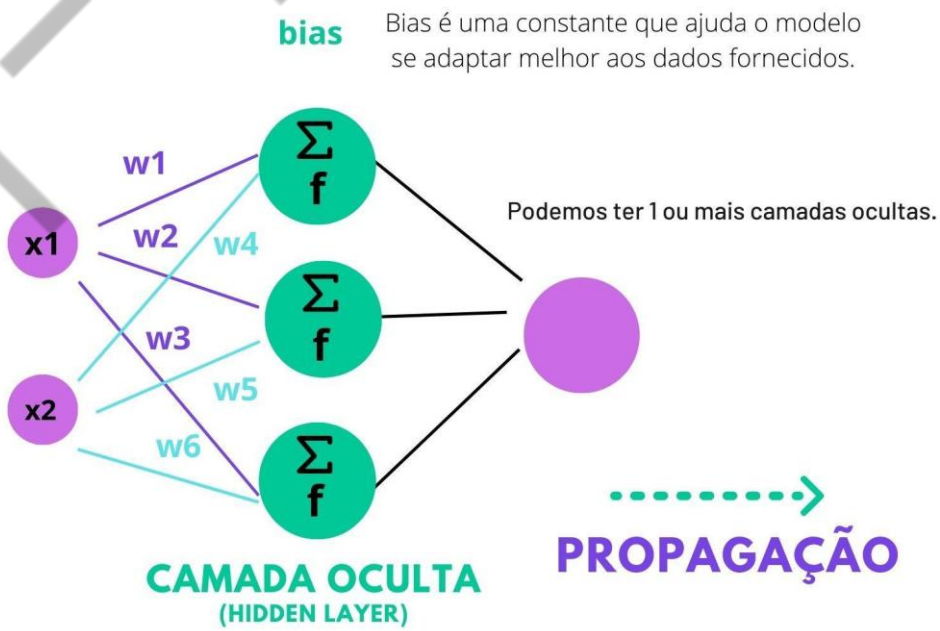

Figura 17 - Explicação Backpropagation: Camada oculta.
Fonte: elaborado pela autora (2023)

Depois de todos os cálculos serem realizados por todas as camadas ocultas, vamos ter uma saída da rede, a chamada “Output Layer” (camada de saída). Essa saída irá retornar um valor de erro dado a comparação do valor estimado versus o valor real.

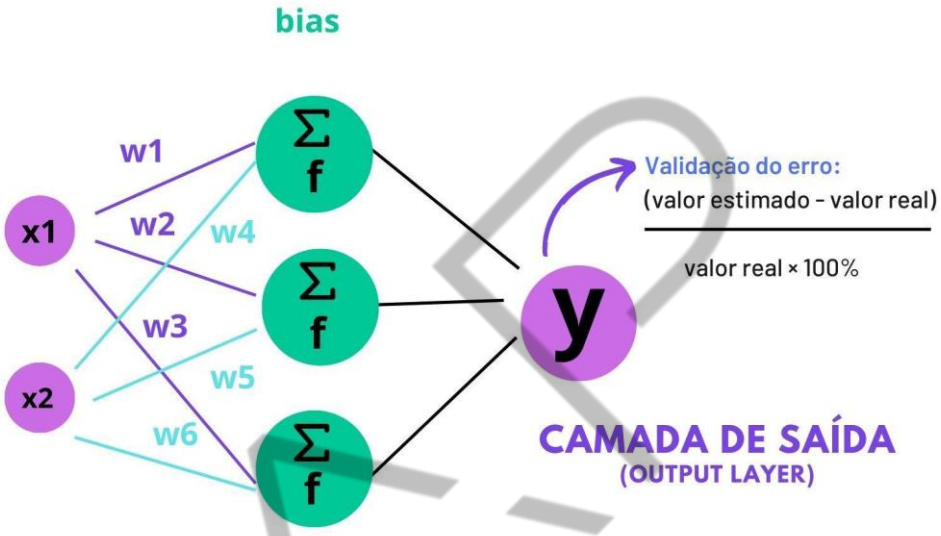


Figura 18 - Explicação Backpropagation: Camada de saída.
Fonte: elaborado pela autora (2023)

Agora veremos o diferencial das redes multicamadas: o backpropagation! Com o erro calculado na primeira passagem da rede, o backpropagation é responsável por voltar o processo da rede desde o início das camadas e reprocessar todos os dados novamente, porém com pesos diferentes. Essa etapa é responsável por atualizar os pesos da rede e gerar um novo valor de saída. A ideia aqui é minimizar os erros gerados na camada de saída.

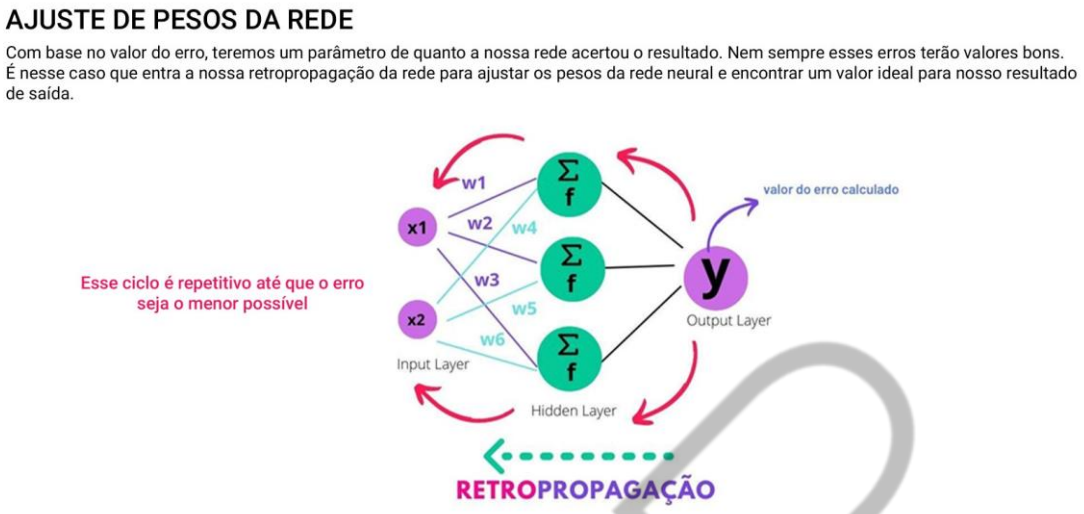


Figura 19 - Explicação Backpropagation: Funcionamento.
Fonte: elaborado pela autora (2023)

Os hiperparâmetros responsáveis pela atualização de pesos e do “vai e vem” da redesão **epochs**. Podemos definir as epochs (Épocas de processamento) como o **total de vezes que são executadas as redes neurais** (Número máximo de épocas).

## OS HIPERPARÂMETROS DAS REDES NEURAIS MULTICAMADAS

Podemos listar muitos hiperparâmetros a serem configurados em uma rede neural multicamadas, como:

**Número de camadas ocultas:** uma única camada de neurônio na rede nem sempre será o suficiente. As redes multicamadas possuem uma arquitetura composta por **várias camadas ocultas**, e essas hierarquias de neurônios auxiliam a rede a **convergir mais rapidamente** para uma boa solução, também ajudando a melhorar a capacidade de **generalizar** para novos conjuntos de dados. Para uma boa partida, inicie sua construção de redes multicamadas com uma ou duas camadas ocultas de neurônios e, se for necessário, é possível construir mais camadas para solucionar o problema. Normalmente, problemas complexos necessitam de mais camadas ocultas. Neste caso, você pode aumentar gradualmente o número de camadas até começar a sobreajustar o conjunto de treinamento. Tarefas muito complexas, como classificação de imagens ou reconhecimento de imagens, necessitam de mais dezenas de camadas ocultas.

**Número de neurônios dentro das camadas:** o número de neurônios nas camadas de entrada e saída é determinado pelo tipo de entrada e saída que sua tarefa requer. Assim como para o número de camadas, você pode tentar aumentar o número de neurônios gradualmente até que a rede comece a se sobreajustar. Em geral, é possível obter mais vantagem aumentando o número de camadas do que o número de neurônios por camada.

**Funções de ativações:** o objetivo de uma função de ativação é **realizar a transformação não linear dos neurônios**. Aliás, uma rede neural sem a função de ativação, é essencialmente um modelo de regressão linear. A função de ativação permite que as mudanças realizadas nos pesos e bias **causem uma alteração na saída final do modelo** (output).

$$Y = \text{Activation}(\Sigma(weight * input) + bias)$$

Figura 20 - Função de ativação.
Fonte: Deep Learning Book [s.d].

Existem diversos tipos de funções de ativações, tais quais:

**Função baseada em limiar (threshold):** a ideia desse tipo de função de ativação é muito simples. Se o valor Y estiver acima de um limite determinado, ative o neurônio; senão, deixe-o desativado.

- Retorna valores binários.
- Se x < 0, então 0. Se x > 0, então 1.
- Pode ser utilizada para classificadores binários (Sim/Não).

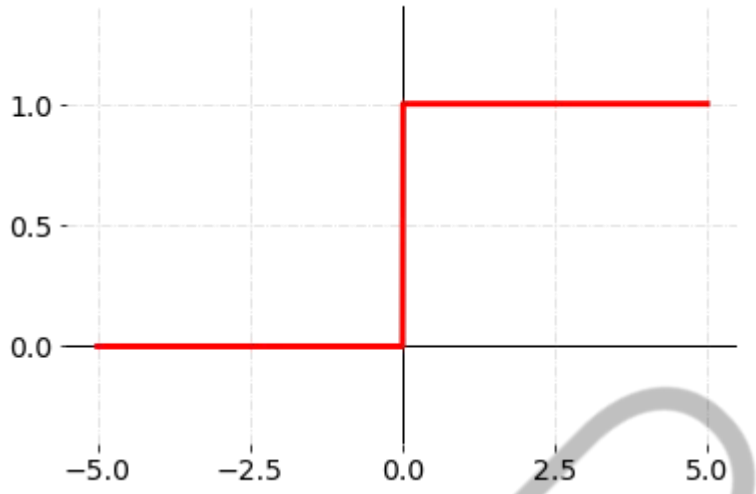


Figura 21 - Função de ativação baseada em threshold.
Fonte: Hasara Samsom (2020)

**Sigmoide:** essa função de ativação tenta empurrar os valores de Y para os extremos. Esta é uma qualidade muito desejável quando tentamos classificar os valores para uma classe específica.

- Saídas com 0 ou 1 (retorna dados binários).
- Possui uma suavização.
- Função não linear.
- A função, essencialmente, tenta empurrar os valores de Y para os extremos.
- Trabalha apenas com valores positivos.
- O resultado da função pode ser uma combinação de diversas variáveis.
- Funciona muito bem para classificar os valores para até duas classes específicas.
- Pode apresentar problemas quando os gradientes se tornam muito pequenos. Isso significa que o gradiente está se aproximando do zero e a rede não está realmente aprendendo.

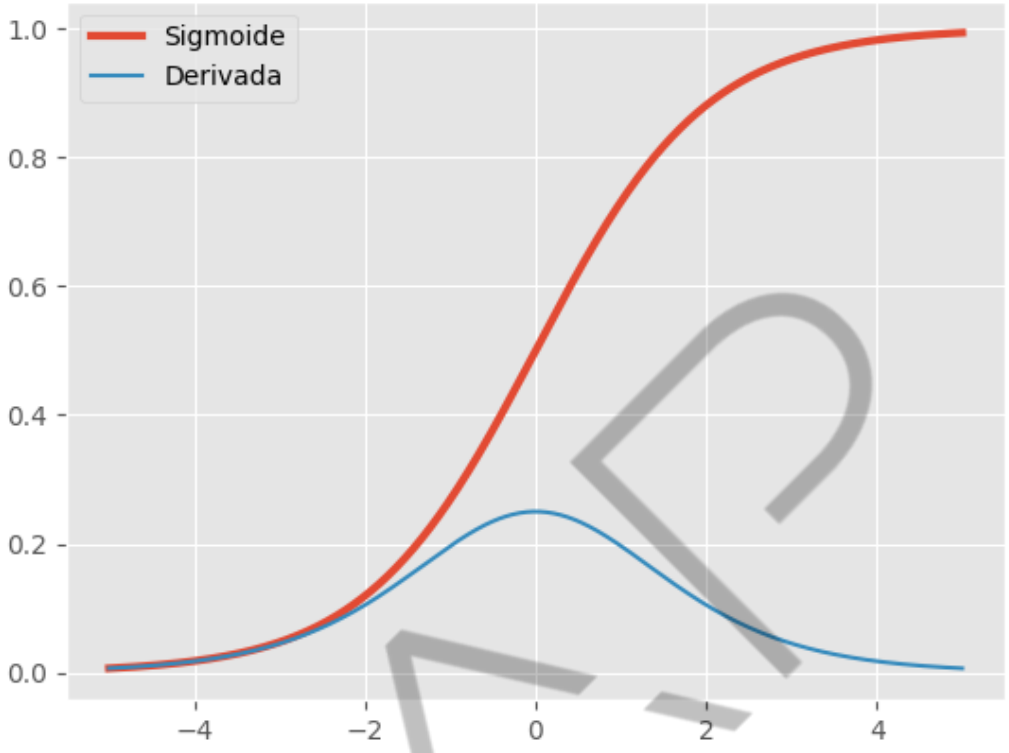


Figura 22 - Função de ativação sigmóide.
Fonte: Matheus Facure (2017)

**Tangente Hiperbólica (tanh):** a função tangente hiperbólica funciona de forma similiar à sigmoide, porém varia de -1 a 1.

- A função tanh é uma versão escalonada da função sigmóide.
- Varia de -1 a 1.
- Função não linear.
- A TanH se aproxima mais da identidade, sendo assim uma alternativa mais atraente do que a sigmóide para servir de ativação às camadas ocultas das RNAs.

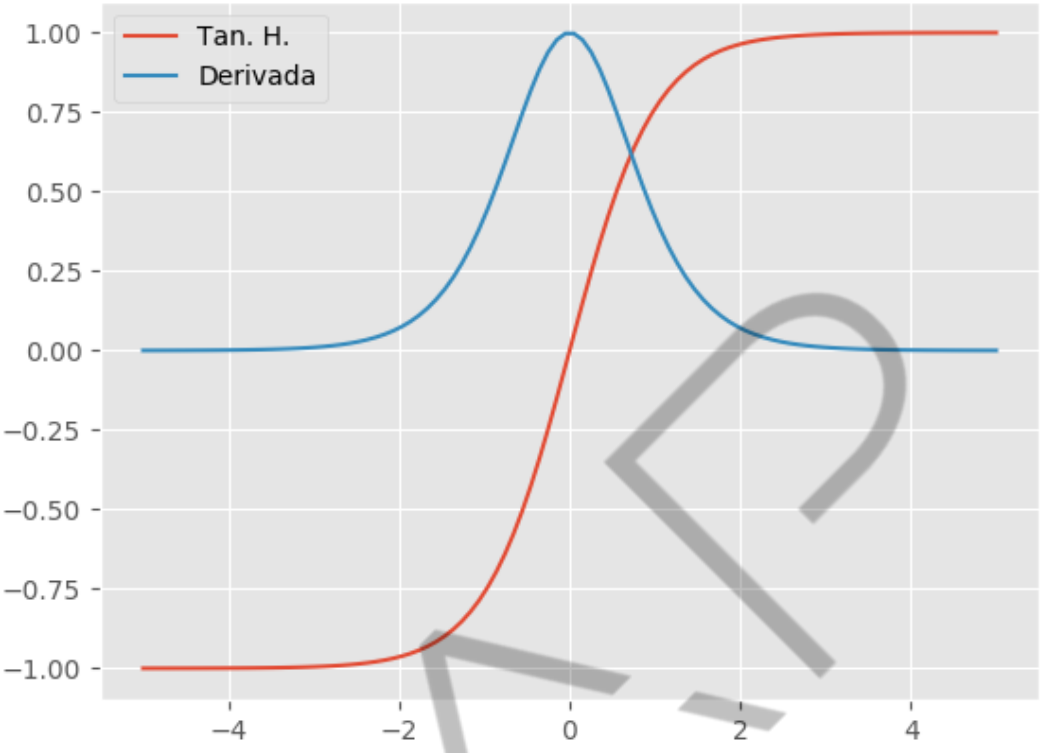


Figura 23 - Função de ativação tanh.
Fonte: Matheus Facure (2017)

**ReLU (unidade linear retificada):** a função ReLU é uma das mais utilizadas em redes neurais, pois uma das suas principais funções é não ativar todos os neurônios ao mesmo tempo (Isso traz mais performance para a rede).

- A função ReLU é não linear.
- Não ativa todos os neurônios ao mesmo tempo. Se você olhar para a função ReLU e a entrada for negativa, ela será convertida em zero e o neurônio não será ativado.
- Menor complexidade computacional.
- ReLU também pode ter problemas com os gradientes que se deslocam em direção a zero.
- A função ReLU é uma função de ativação geral e é usada na maioria dos casos atualmente.

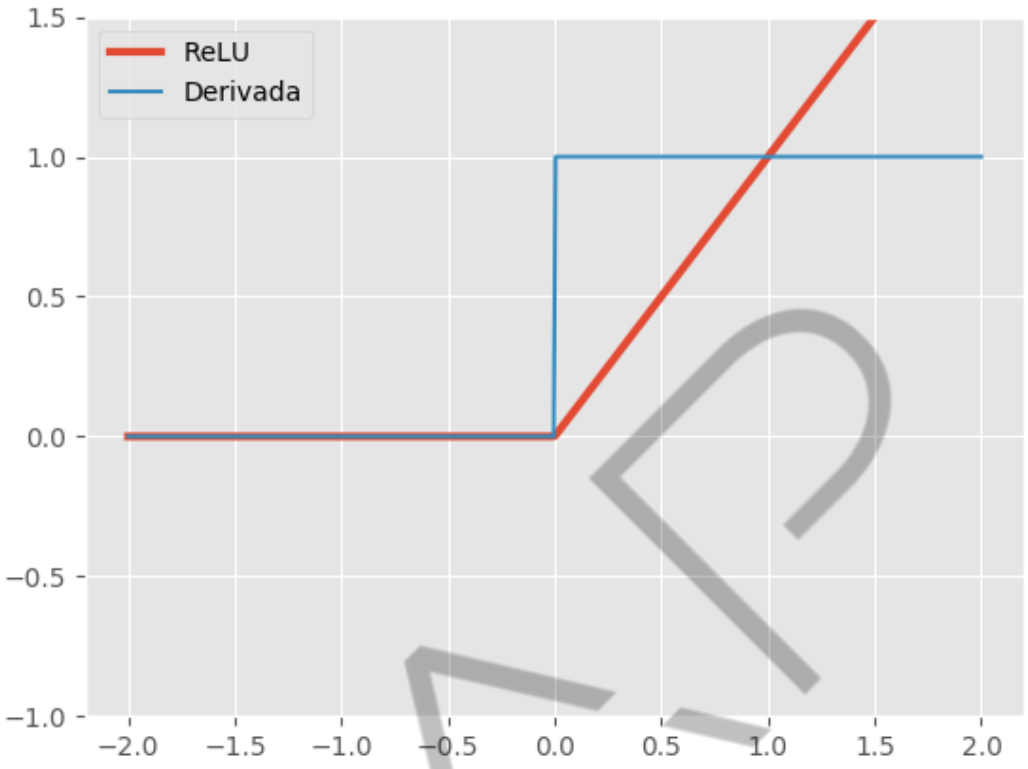


Figura 24 - Função de ativação ReLU.
Fonte: Matheus Facure (2017)

**Leaky ReLU:** a função Leaky ReLU é uma versão melhorada da ReLU, com a diferença de que valores menores que 0 recebem um componente linear de ativação.

- Versão melhorada da ReLU.
- Faz a ativação para valores x<0.
- Para valores menores que 0, é definido um pequeno componente linear (0,01).
- A principal vantagem de substituir a linha horizontal é remover o gradiente zero.

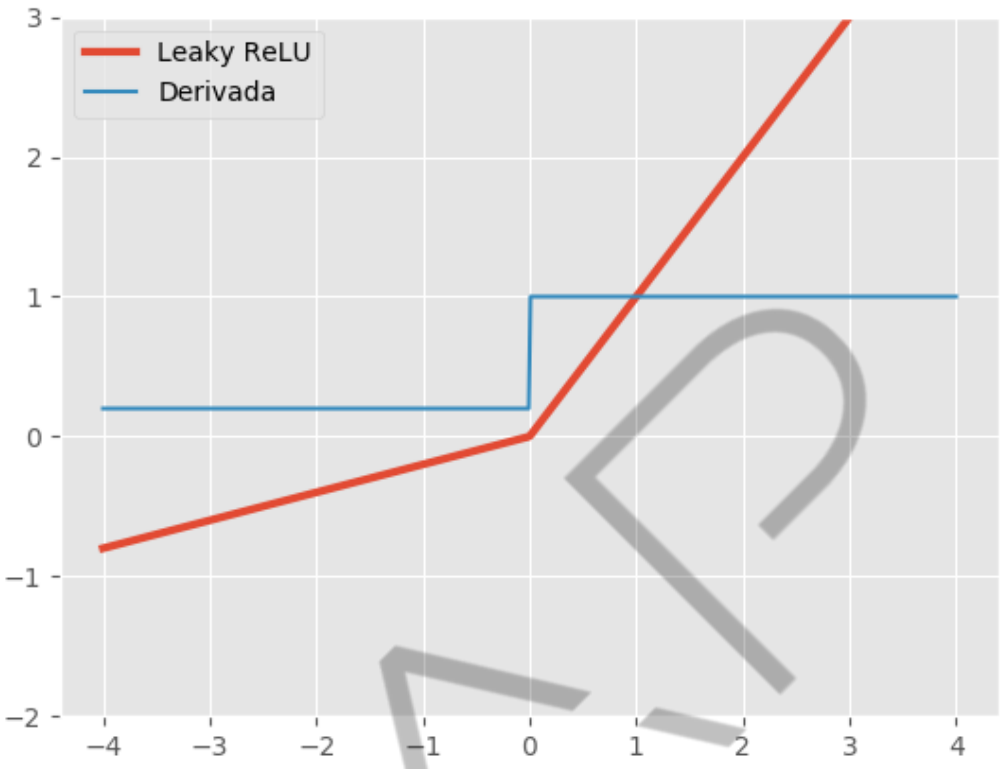


Figura 25 - Função de ativação Leaky ReLU.
Fonte: Matheus Facure (2017)

**Softmax:** a função softmax é útil quando queremos lidar com problemas de multiclasses. Ela transforma as saídas de cada classe para valores entre 0 e 1 e também divide pela soma das saídas.

- É um tipo de função sigmóide.
- Além de transformar as saídas para 0 e 1, também divide pela soma das saídas.
- Atribuí a probabilidade de resultado pertencer a uma determinada classe.
- É idealmente usada na camada de saída do classificador, no qual estamos tentando gerar as probabilidades para definir a classe de cada entrada.
- Ideal quando estamos tentando lidar com várias classes.


Figura 26 - Função de ativação Softmax.
Fonte: Acervo Lima (s.d.)

Falando da aplicação das funções de ativações em redes neurais multicamadas, na maioria dos casos é possível utilizar a função ReLU nas camadas ocultas, pelo fato de ser mais rápida para calcular do que outras funções de ativações. Outra vantagem é que o gradiente descendente não fica tão preso em platôs, graças ao fato de que ela não satura para grandes valores de entrada.

Quando temos classes exclusivas, a função de ativação Softmax é uma boa opção para tarefas de classificação na camada de saída da rede. Para casos onde temos apenas duas classes de saídas, a função de ativação sigmoide pode ser uma boa opção.

## ENTENDENDO O GRADIENTE DESCENDENTE

É possível observar que, ao longo da explicação, o termo “gradiente descendente” foi muito citado, mas afinal, o que é um gradiente descendente?

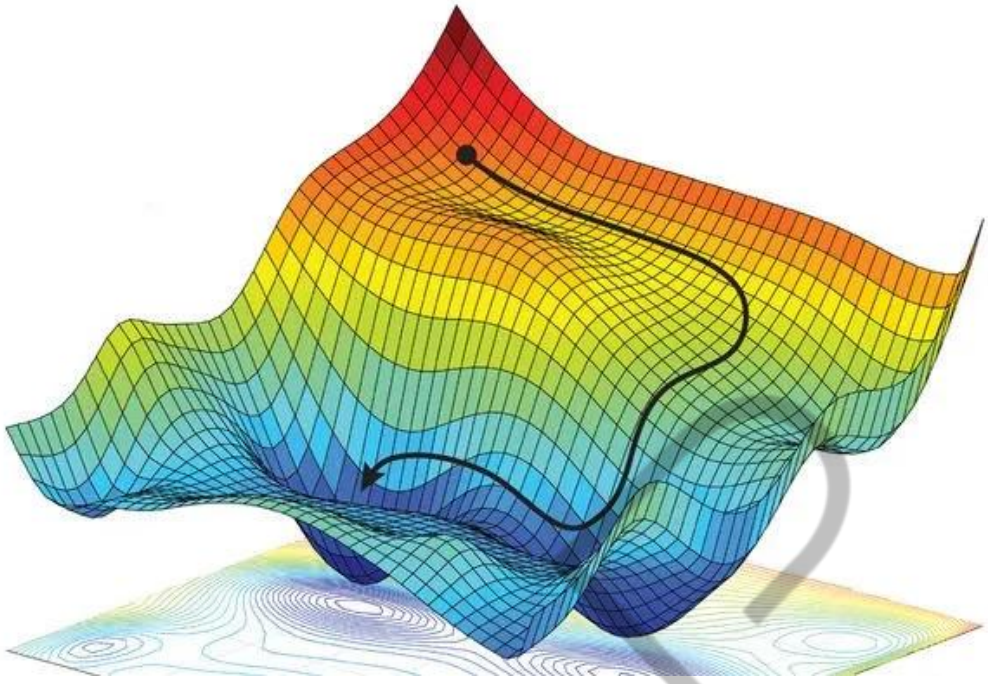

Figura 27 - Função de ativação Softmax.
Fonte: Arthur Lamblet Vaz (2020)

A descida do gradiente é um **algoritmo de otimização para encontrar os valores de parâmetros (pesos) de uma função que minimizam uma função de custo**. Você também pode pensar em um gradiente como a inclinação de uma função. Quanto maior o gradiente, mais inclinada é a subida e mais rápido o modelo pode aprender. Mas se a inclinação for zero, o modelo para de aprender. O gradiente descendente (GD) é um algoritmo utilizado para encontrar o **mínimo de uma função de forma interativa.**

Pensando na questão da matemática por trás dos gradientes, podemos dizer que o gradiente descendente é a **derivada da função do erro** em relação a nosso peso.

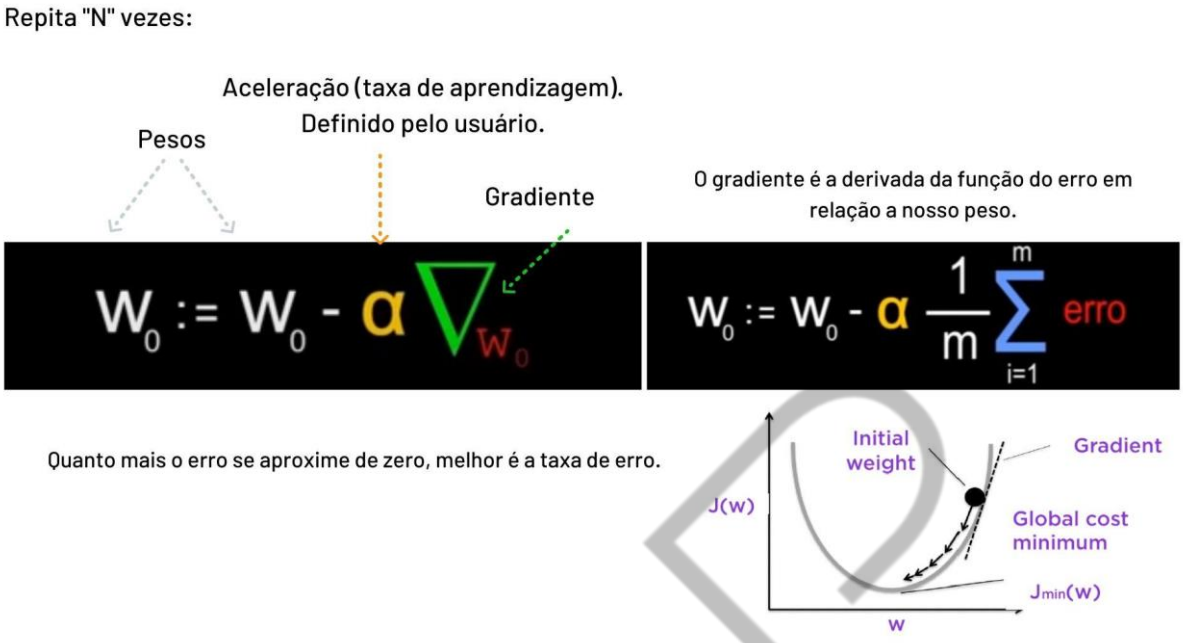


Figura 28 - Gradiente descendente.
Fonte: elaborado pela autora (2023)

Este processo é repetido até que o **custo dos coeficientes** (**Função de custo**) seja 0,0 ou próximo o suficiente de zero, indicando que as saídas da rede estão cada vez mais próximas dos valores reais (Saídas desejadas).

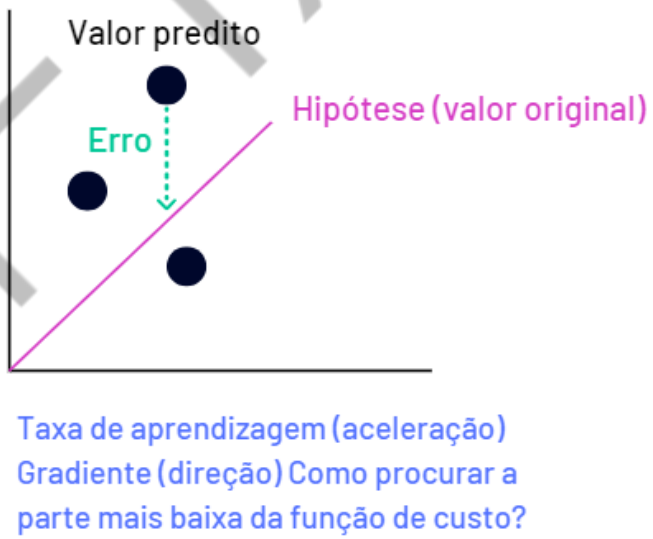


Figura 29 - Gradiente descendente.
Fonte: elaborado pela autora (2023)

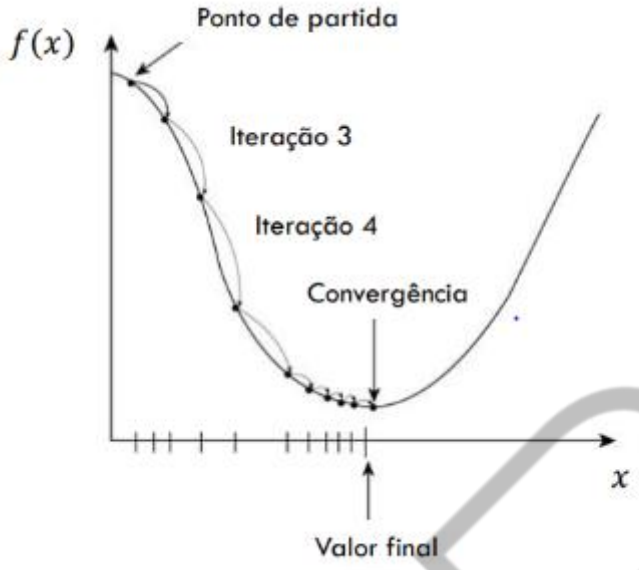


Figura 30 - Gradiente descendente.
Fonte: elaborado pela autora (2023)

A performance que ocorre durante a descida do gradiente é controlada pela taxa de aprendizagem, também conhecida como **learning rate**. Podemos dizer que a learning rate controla o tamanho dos “passos” das interações para encontrar um melhor mínimo local (menor erro). Uma taxa de aprendizagem baixa pode ser mais precisa, porém pode demorar para encontrar o mínimo local. Uma taxa de aprendizado alta pode deixar passar o melhor mínimo local.

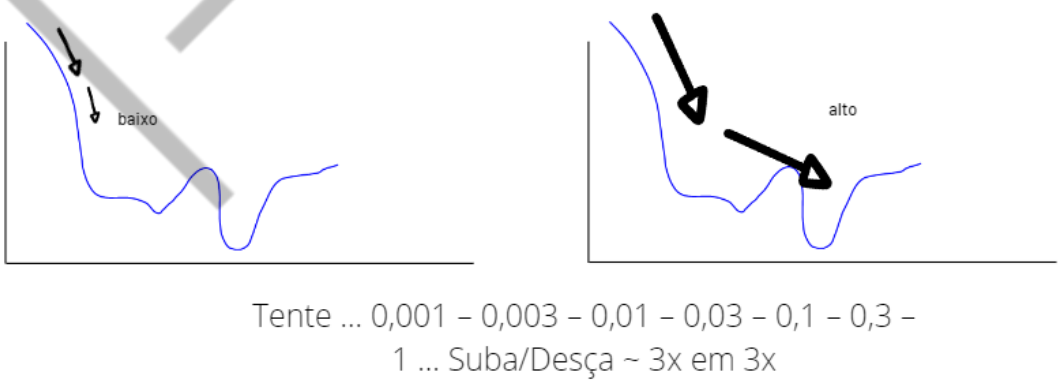


Figura 31 - Learning rate.
Fonte: elaborado pela autora (2023)

A learning rate é um hiperparâmetro muito importante a ser configurado nas redes neurais e normalmente tentamos valores muito baixos quando buscamos uma

eficiência maior da convergência do erro, casando então com um número maior de épocas de processamento, ou também podemos ter learning rates nem tão baixas assim e com épocas menores de processamento. Aqui a ideia é sempre testar qual melhor configuração para o modelo, qual irá convergir melhor.

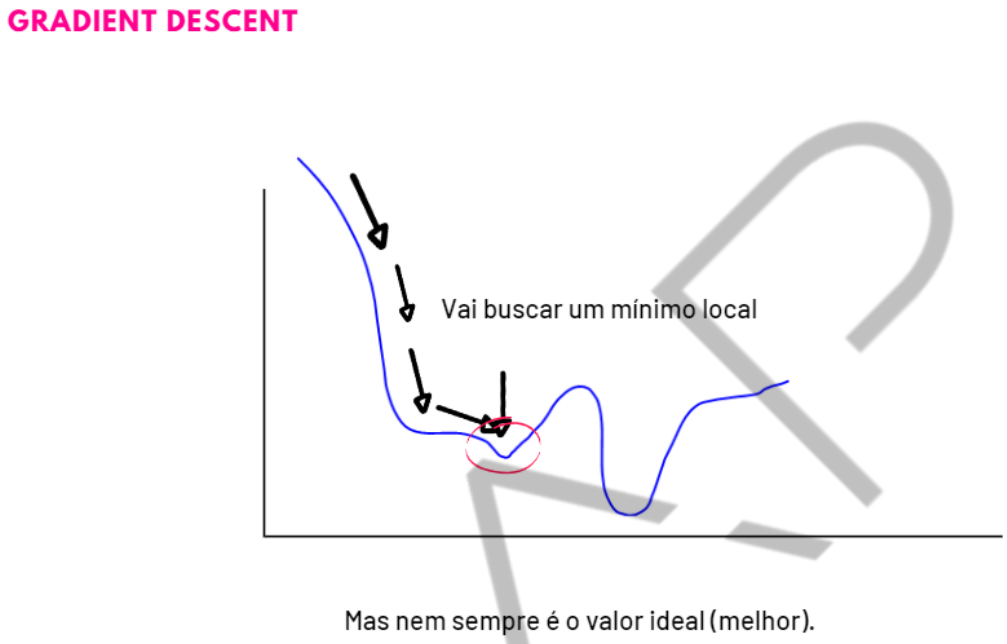


Figura 32 - Learning rate convergindo para o menor mínimo local.
Fonte: elaborado pela autora (2023)

Perceba que, nem sempre a learning rate irá atingir o menor mínimo local global do erro. Sendo assim, é necessário alterar as configurações de leraning rate e épocas de processamento (epoch) para o algoritmo convergir para o melhor mínimo local "ideal".

## O PROBLEMA DOS GRADIENTES: VANISHING/EXPLODING

Infelizmente, os gradientes geralmente ficam cada vez menores à medida que o algoritmo avança para as camadas inferiores e vai atualizando os pesos. Como resultado, a atualização do gradiente descendente deixa os pesos da conexão da camada inferior praticamente inalterados e o treinamento nunca converge para uma boa solução. Este é o famoso caso do **vanishing gradiente**. Também podemos ter o contrário acontecendo: os gradientes podem crescer cada vez mais, as camadas recebem atualizações de peso insanamente grandes e o algoritmo diverge, o que é o problema de **exploding gradiente**. No geral, as redes neurais profundas sofrem

gradientes instáveis, ou seja, diferentes camadas podem aprender em velocidades muito diferentes.

## TIPOS DE ALGORITMOS OTIMIZADORES VELOZES

Temos um conceito chamado “batchs” em deep learning, que funcionam como lotes de dados para otimizar a performance do algoritmo. Os otimizadores de gradiente descendente utilizam as chamadas mini-batches (mini lotes de dados) para otimizar a performance, passando pela rede múltiplas amostras (por exemplo: 128 amostras), calcular o erro médio delas e então realizar o backpropagation e a atualização dos pesos. Existem vários tipos de algoritmos otimizadores que podem calcular a função de custo das redes neurais, vamos listar aqui os principais tipos utilizados e ver a sua aplicação em Python.

## OTIMIZAÇÃO MOMENTUM

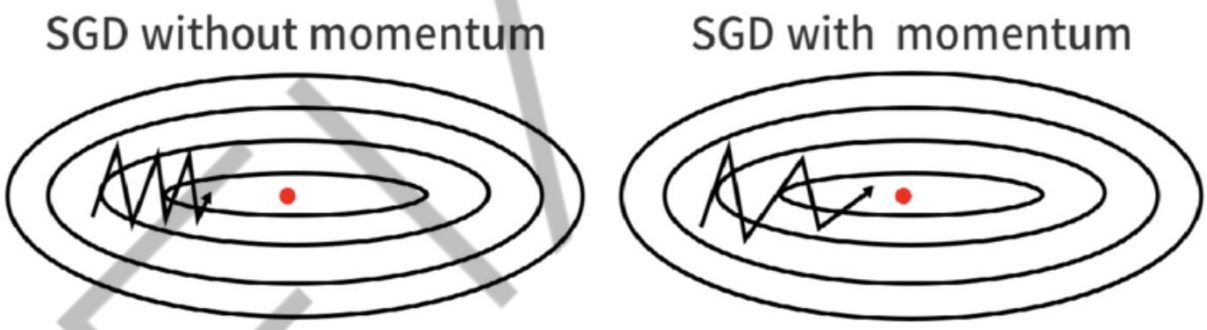


Figura 33 - Otimizador momentum
Fonte: Papers With Code (s.d.)

Podemos comparar esses tipos de otimizador como uma bola de boliche, rolando em um declive suave em uma superfície lisa (Irá começar a descer de forma lenta, porém logo pegará impulso até que chegue à velocidade final). A otimização momentum se preocupa muito com os gradientes anteriores: a cada iteração, ela subtrai o gradiente local do vetor momentum **m** (multiplicado pela taxa de aprendizado n) e atualiza os pesos adicionando este vetor momentum. O algoritmo adiciona um novo hiperparâmetro β (momentum) que deve ser ajustado entre 0 até 1. Um bom e típico valor para esse hiperparâmetro costuma ser 0,9.

$$1. \quad \mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$
$$2. \quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}$$

Figura 34 - Fórmula algoritmo otimizador momentum.
Fonte: Géron Aurélien (2019)

A biblioteca Keras fornece uma interface simples para trabalhar com redes neurais, e podemos observar no código a seguir como é fácil a configuração do otimizador momentum.

**optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)**

**Gradiente Acelerado de Nesterov**

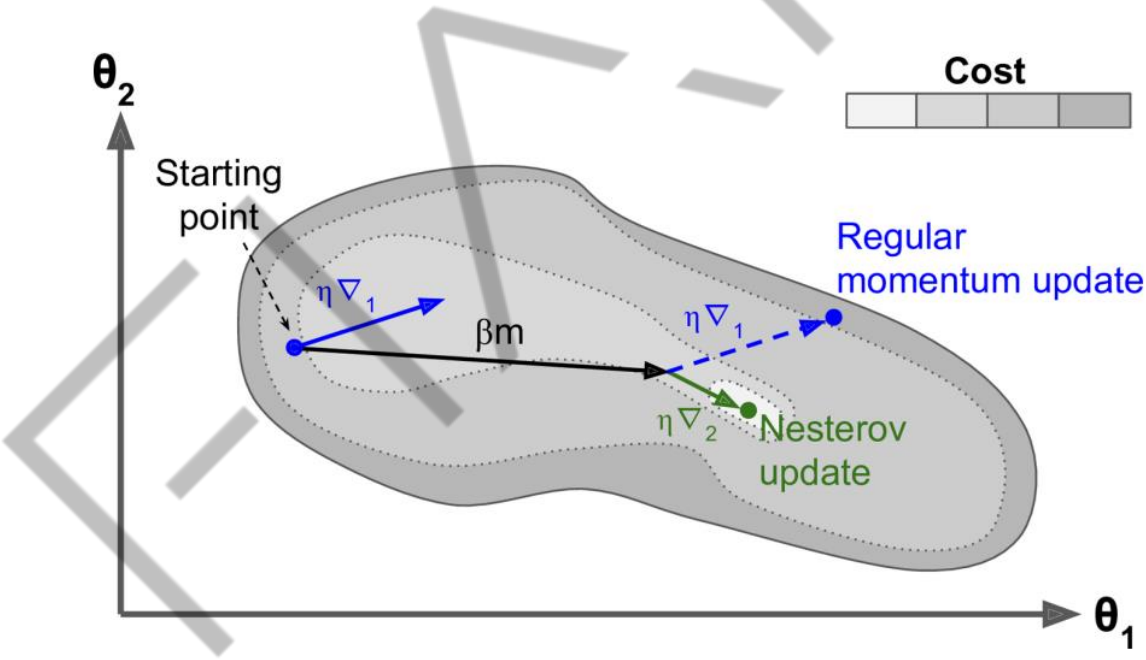


Figura 35 - Otimizador Nesterov.
Fonte: Géron Aurélien (2019)

Podemos definir o Gradiente Acelerado Nesterov uma variação do algoritmo de momentum, onde quase sempre essa opção é mais rápida do que a otimização de momentum normal. A ideia do algoritmo é medir o gradiente da função de custo, não na posição local, mas ligeiramente à frente, na direção do momentum. A única diferença da otimização momentum normal é o gradiente medido em θ + βm.

$$1. \quad \mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta} + \beta\mathbf{m})$$
$$2. \quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}$$

Figura 36 - Fórmula algoritmo otimizador Nesterov.
Fonte: Géron Aurélien (2019)

Em comparação com a otimização de momentum, o Nesterov quase sempre irá acelerar o treinamento. Veja a sua aplicação utilizando a biblioteca Keras:

**optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)**

**AdaGard**

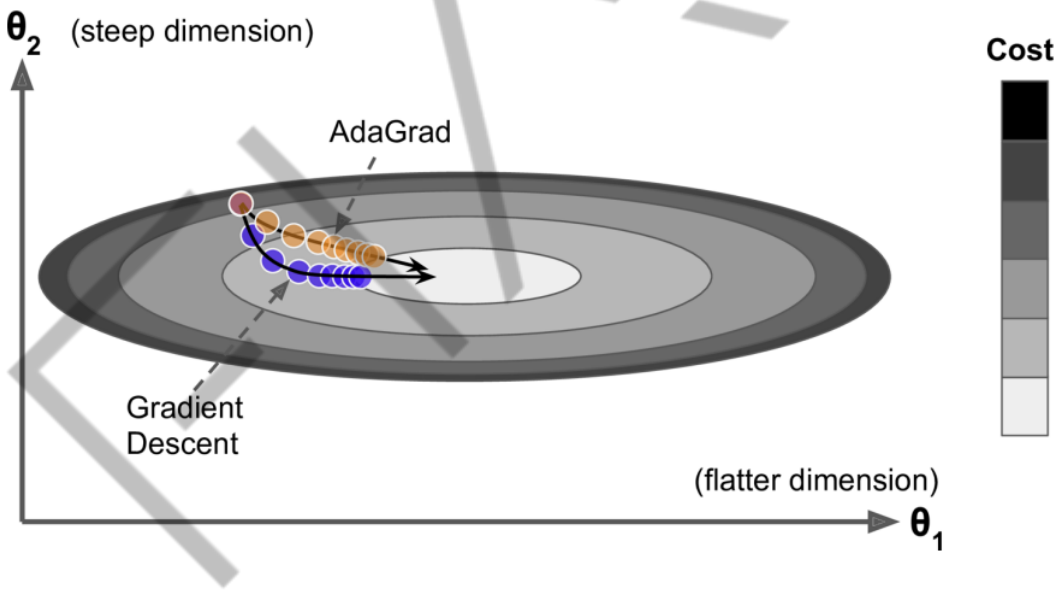


Figura 37 - Otimizador AdaGrad.
Fonte: Géron Aurélien (2019)

Considere o seguinte exemplo: imagine uma tigela alongada, onde o gradiente descendente começa a descer a encosta mais íngreme rapidamente para depois descer lentamente em direção ao fundo. Seria bom se o algoritmo detectasse isso rapidamente, e corrigisse sua direção para direcionar mais para o global optimum (melhor mínimo local). O algoritmo AdaGrad corrige essa situação! Basicamente, o AdaGrad escalona o vetor gradiente na dimensão mais íngreme.

$$\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \varepsilon}$$

Figura 38 - Fórmula algoritmo otimizador AdaGrad.
Fonte: Géron Aurélien (2019)

AdaGrad geralmente funciona bem para problemas quadráticos simples, mas infelizmente, muitas vezes ele para treinar as redes neurais muito cedo. A taxa de aprendizagem diminui tanto que o algoritmo acaba parando completamente antes de encontrar um melhor mínimo local. O Keras possui essa opção de otimizador, porém a utilização não é recomendada por conta deste problema citado.

**RMSPROP**

O otimizador RMSProp corrige o que o AdaGrad faz: o algoritmo acumula somente os gradientes das interações mais recentes (em vez de todos os gradientes desde o início do treinamento) por meio do decaimento exponencial no primeiro passo.

$$\mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \varepsilon}$$

Figura 39 - Fórmula algoritmo otimizador RMSProp.
Fonte: Géron Aurélien (2019)

A taxa de degradação β é ajustada para 0,9, geralmente esse valor padrão funciona bem. Exceto para problemas muito simples, o otimizador RMSProp quase sempre funciona muito bem e melhor que o AdaGrad. Esse otimizador já foi um dos favoritos da comunidade, mas perdeu seu posto para o Adam (Adaptative moment estimation).

**OTIMIZAÇÃO ADAM**

O adaptative moment estimation combina as ideias do momentum e do RMSProp: assim como a otimização de momentum, o Adam acompanha uma média exponencialmente decadente de gradientes passados, e assim como o RMSProp, controla uma média exponencialmente decadente de gradientes quadrados passados.

$$\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$
$$\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$
$$\widehat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^{\mathsf{T}}}$$
$$\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^{\mathsf{T}}}$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \widehat{\mathbf{m}} \oslash \sqrt{\hat{\mathbf{s}} + \varepsilon}$$

Figura 40- Fórmula algoritmo otimizador Adam.
Fonte: Géron Aurélien (2019)

O otimizador Adam requer menos configuração de hiperparâmetro da taxa de aprendizagem, então um valor padrão n = 0,001 pode ser uma boa opção!

**optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)**

**A IMPORTÂNCIA DE ESCALONAR DADOS EM DEEP LEARNING**

É importante termos em mente que uma rede neural aprende melhor quando os dados de entrada estão em uma escala normalizada ou padronizada. Os dados transformados encontram uma convergência mais rápida. Se os dados de entrada forem muito grandes, a função de custo terá uma dificuldade maior para encontrar o melhor mínimo local do gradiente descendente.

Outro hiperparâmetro que pode ser afetado pelas escalas dos dados são as funções de ativações, principalmente aquelas que possuem regiões de gradientes muito pequenos (Sigmóide e tangente hiperbólica, por exemplo) ocasionando o problema de vanishing gradiente.

## EVITANDO O OVERFITTING POR MEIO DA REGULARIZAÇÃO

As redes neurais profundas geralmente possuem dezenas de milhares de parâmetros e às vezes até milhões. Como tantos parâmetros, a rede tem uma incrível quantidade de liberdade e pode conter uma grande variedade de conjuntos de dados complexos. Porém, essa grande liberdade pode levar ao tão temido overfitting.

As técnicas de regularização são utilizadas para minimizar super ajustes (overfitting) dos dados. A regularização remove os ruídos e deixa o modelo com melhor performance. O que é um modelo super ajustado? Quando o algoritmo é colocado em produção, não consegue validar dados do mundo real. O algoritmo praticamente "decorou" os dados do modelo de treinamento.

As regularizações praticamente adicionam um termo de penalização na função de custo do modelo. Podemos listar algumas das principais técnicas de regularização utilizada nos modelos de deep learning.

## REGULARIZAÇÕES L1 E L2

Essas regularizações restringem os pesos das conexões da rede adicionando um termo de regularização adequado à sua função de custo.

### Regularização L1

- L1 vai aplicar uma restrição aos coeficientes menos importantes, levando-os a zero;
- Ao zerar um coeficiente, elimina-se o atributo;
- Melhor performance.

**Regularização L2**

- L2 penaliza os coeficientes que assumem valores muito grandes, levando-os a tender a zero;
- Ajuda a resolver o problema de atributos muito correlacionados;
- Por não reduzir atributos, é muito bem utilizado em datasets menores;
- A grande diferença aqui é que este tipo não chega a zerar o coeficiente, mas reduz os menos importantes a valores muito baixos e mantendo todos eles no modelo.

**DROPOUT**

Dropout é uma das técnicas de regularização mais utilizadas e seu funcionamento é muito simples: em cada etapa de treinamento, os neurônios possuem uma probabilidade de serem temporariamente “descartados”, ou seja, é possível que eles sejam totalmente ignorados durante o treinamento da rede. Essa probabilidade é chamada de taxa de dropout aplicada na rede. A técnica de remover neurônios aleatórios da rede pode inclusive ajudar na performance da rede neural.

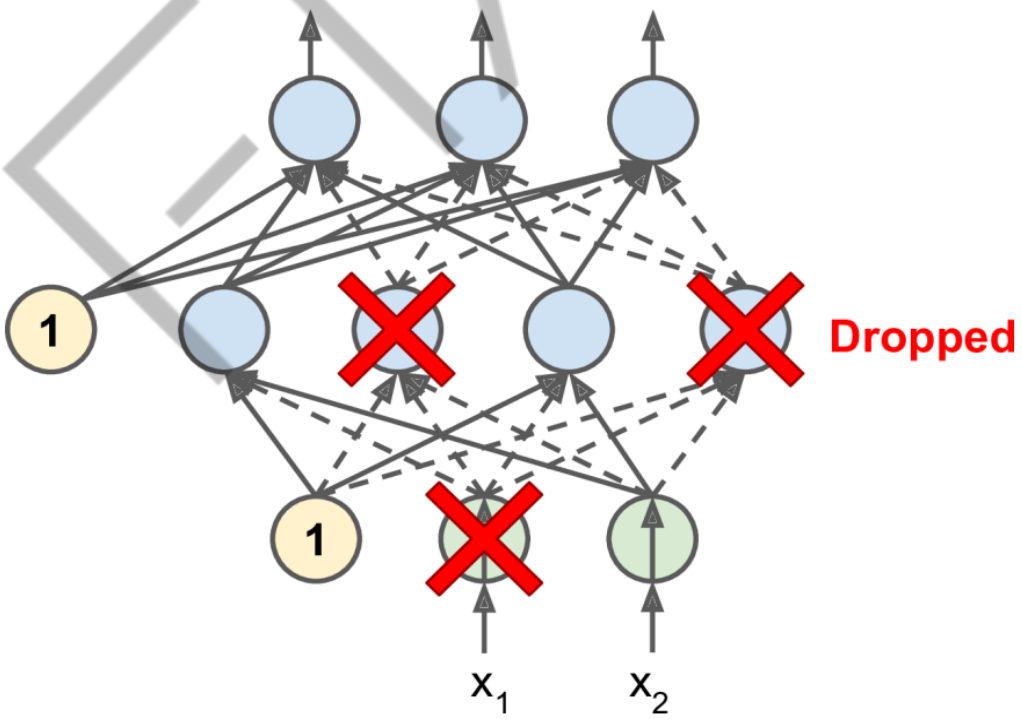


Figura 41- Fórmula algoritmo otimizador Adam.
Fonte: Géron Aurélien (2019)

## EARLY STOPPING

Com essa técnica, a rede neural é interrompida quando começa a ter uma degradação da rede. Quando é gerado muitas epochs (épocas de processamento), pode-se levar a queda da performance da rede. Para evitar esse tipo de situação, a early stopping identifica essa degradação e para as epochs antes do modelo não ficar performático.

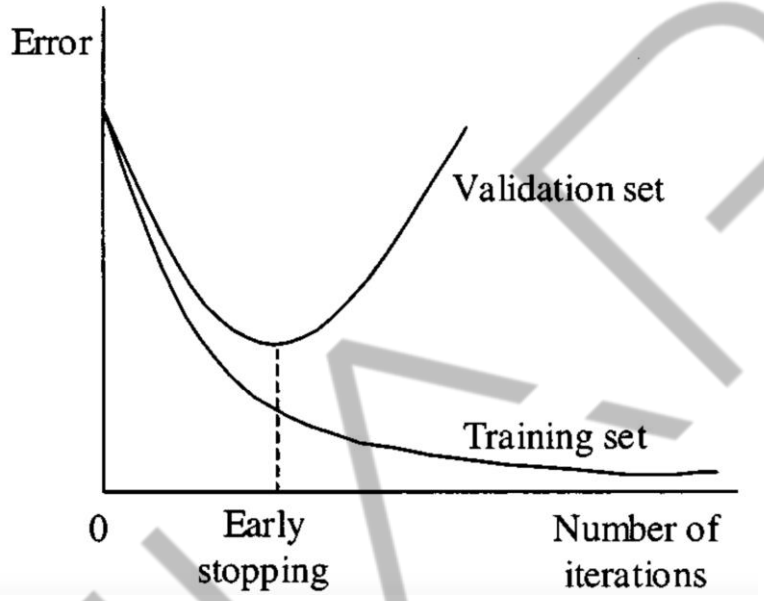


Figura 42 - Técnica Early Stopping.
Fonte: Papers With Code (s.d.)

## BIBLIOTECAS PARA REDES NEURAIS EM PYTHON

Para realizar o treinamento das redes neurais em Python, existem diversas opções de bibliotecas para você conhecer e aplicar o seu conhecimento obtido.

Figura 43 – Tensorflow
Fonte: Tensorflow (2023)

**TensorFlow:** desenvolvido pela Google Brain, o TensorFlow é uma das bibliotecas de deep learning mais populares, que utiliza grafos de fluxo de dados.

Figura 44 – Keras
Fonte: Keras (2023)

**Keras**: é uma API de alto nível que pode ser executada em conjunto com várias plataformas de deep learning, incluindo o TensorFlow, como um backend. A biblioteca é bem simples de ser trabalhada, dado que o assunto de redes neurais é complexo de entender. As redes são construídas em sequências e baseadas em grafos.

Figura 45 – PyTorch
Fonte: PyTorch (2023)

**PyTorch:** desenvolvida pelo Facebook AI Research, o PyTorch é conhecido por sua flexibilidade e é amplamente utilizado por pesquisadores. Ele é classificado como “Tensores e redes neurais dinâmicas em Python com forte aceleração de GPU”.

## O QUE VOCÊ VIU NESTA AULA?

Nessa aula, você aprendeu o que são as redes neurais e como funcionam alguns conceitos básicos para funcionamento da arquitetura de redes neurais multicamadas. Essa é apenas a pontinha do iceberg para o mundo das redes neurais artificiais!

O que achou do conteúdo? Conte-nos no Discord! Estamos disponíveis na comunidade para fazer networking, tirar dúvidas, enviar avisos e muito mais. Participe!

# REFERÊNCIAS

Deep Learning Book. [s.d.]. Disponível em: <https://www.deeplearningbook.com.br/>. Acesso em: 07 nov. 2023.

Early Stopping. Papers With Code. [s.d.]. Disponível em: <https://paperswithcode.com/method/early-stopping>. Acesso em: 07 nov. 2023.

GÉRON, Aurélien. **Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow**. 2nd Edition. [s.l.]: O'Reilly Media, Inc., 2019.

Getting started. Keras. [s.d.]. Disponível em: <https://keras.io/getting_started/>. Acesso em: 07 nov. 2023.

Introduction to Kernal PCA with Python. **CMDLineTips**. [s.d.]. Disponível em: <https://cmdlinetips.com/2021/02/kernal-pca-with-python/>. Acesso em: 08 nov. 2023.

SGD with Momentum. **Papers With Code**. [s.d.]. Disponível em: <https://paperswithcode.com/method/sgd-with-momentum>. Acesso em: 07 nov. 2023.

Utilizando uma rede neural artificial para aproximação da função de evolução do sistema de Lorentz. **Research Gate**. 2016. Disponível em: <https://www.researchgate.net/figure/Figura-1-Representacao-do-neuronio-artificial_fig1_329245206>. Acesso em: 07 nov. 2023.

Why choose Keras? **Keras**. [s.d.]. Disponível em: <https://keras.io/why_keras/>. Acesso em: 07 nov. 2023.

## PALAVRAS-CHAVE

**Palavras-chave:** Deep Learning, Neurônios, Backpropagation, Multilayer Perceptron, Otimizadores.