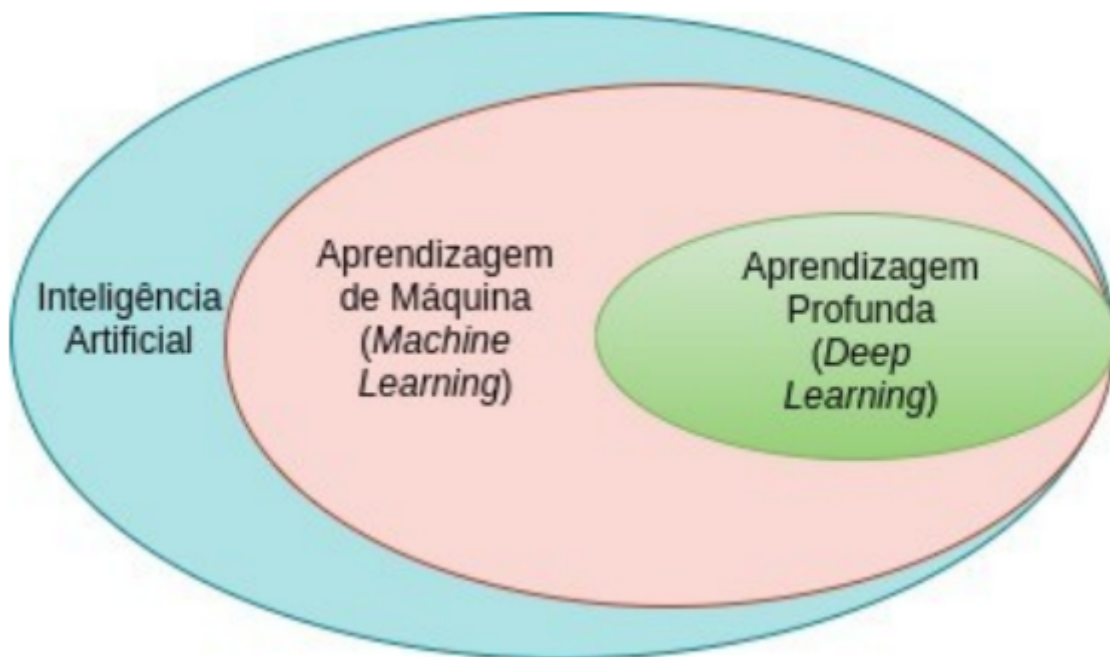


**Machine Learning:** Treinamento de modelos computacionais a partir de um conjunto de dados para simular a inteligência humana. Os algoritmos de *machine learning* aprendem a realizar tarefas específicas com base em exemplos de dados, sem serem explicitamente programados para cada etapa. Eles são amplamente usados em tarefas como classificação, regressão e agrupamento.

**Deep Learning:** A aprendizagem profunda (*deep learning*) é uma **subárea de machine learning** que **utiliza redes neurais artificiais**, inspiradas no funcionamento do cérebro humano, para realizar tarefas de aprendizado. **A principal característica do deep learning é o uso de redes neurais profundas**, ou seja, **redes com várias camadas ocultas**. Quanto mais camadas uma rede possui, mais capaz ela é de aprender representações abstratas e complexas dos dados. Esse tipo de abordagem é especialmente eficaz em problemas como reconhecimento de imagens, processamento de linguagem natural (PNL) e sistemas de recomendação.

Para contextualizar, um exemplo real de aplicação de *deep learning* seria o reconhecimento facial, onde uma rede neural profunda aprende a identificar características faciais, como formato de olhos, nariz e boca, em várias camadas, sem necessidade de intervenção humana para definir esses aspectos.



### O que é *Machine Learning*?

Em tarefas onde é difícil de se delimitar um grupo de regras específico para sua conclusão, utiliza-se o conceito de *machine learning*. Em vez de definir regras manualmente, como seria feito em algoritmos tradicionais, **os modelos de machine learning são treinados com dados e aprendem padrões a partir desses exemplos**.

Isso permite que os algoritmos façam previsões, classifiquem informações ou tomem decisões sem serem explicitamente programados para cada situação.

Por exemplo, imagine que você queira desenvolver um sistema que identifique e categorize e-mails como "spam" ou "não spam". Seria complicado criar um conjunto de regras que cubra todas as situações possíveis, já que cada e-mail pode ser escrito de forma diferente. Em vez disso, com *machine learning*, você pode treinar um modelo com milhares de exemplos de e-mails rotulados (como "spam" ou "não spam"), e o modelo aprenderá automaticamente as características que diferenciam esses e-mails.

Assim, **usamos dados para que o modelo/algoritmo aprenda padrões a partir de exemplos, permitindo que ele generalize esse conhecimento para novos dados**. Ou seja, uma vez que o modelo foi treinado adequadamente, ele poderá identificar e-mails como "spam" ou "não spam" em novos casos que nunca viu antes, com base nos padrões aprendidos durante o treinamento.

O sucesso de um modelo de *machine learning* depende muito da qualidade e da quantidade dos dados usados no treinamento, além de sua capacidade de **generalizar**, ou seja, aplicar o conhecimento aprendido a novos dados que ainda não foram apresentados ao modelo.

### Técnicas de aprendizagem

- **Aprendizagem supervisionada:** O modelo é treinado com dados rotulados, ou seja, os exemplos fornecidos já têm suas respectivas categorias ou valores corretos. O objetivo é prever as saídas corretas para novos dados com base no aprendizado dos exemplos anteriores.
  - Exemplo: Um modelo que recebe imagens de gatos e cachorros rotuladas e aprende a distinguir entre os dois.
- **Aprendizagem não supervisionada:** O modelo recebe dados **não rotulados** e precisa identificar padrões ou estruturas nos dados por conta própria. Ele tenta agrupar ou organizar os dados com base em similaridades.
  - Exemplo: Segmentação de clientes em grupos com base em comportamentos de compra.
- **Aprendizagem por reforço:** O modelo aprende através de **tentativa e erro**, recebendo **recompensas** ou **punições** com base em suas ações. O objetivo é maximizar a recompensa ao longo do tempo, encontrando a melhor estratégia possível.
  - Exemplo: Um rato que aprende a navegar por um labirinto, recebendo comida quando se move na direção correta e um choque leve quando se move na direção errada.

Alguns métodos clássicos de machine learning:

- Support Vector Machine (SVM)
- Naive Bayes
- Árvore de Decisão
- Knn
- K-Means

## O que são Redes Neurais?

**Redes neurais** são uma imitação (ainda que simplificada) do neurônio biológico humano, que recebe informações/estímulos e os processa, aplicando os devidos pesos para gerar uma saída. Uma rede neural é composta por **camadas de neurônios artificiais** conectados entre si. Cada camada realiza um processamento dos dados e passa o resultado para a próxima camada, até gerar a saída final.

- **Camada de entrada:** Recebe os dados de entrada.
- **Camadas ocultas:** Processam os dados, ajustando os pesos e realizando cálculos.
- **Camada de saída:** Gera o resultado final do processamento.

As redes neurais existem desde 1950, no entanto, se popularizaram apenas a partir dos anos 2000 devido a alguns fatores, como o maior volume de dados disponível (*Big Data*), maior capacidade de processamento por parte do *hardware* e, consequentemente, melhores GPUs (placas de vídeo).

Entrada/saída de dados

Redes neurais costumam trabalhar com cálculo vetorial. No exemplo abaixo, temos um exemplo de uma matriz contendo valores "x", onde cada linha representa um atributo (ou *feature*) e cada coluna representa um exemplo (ou instância de treinamento/teste).

## Vetor de atributos (*features*)

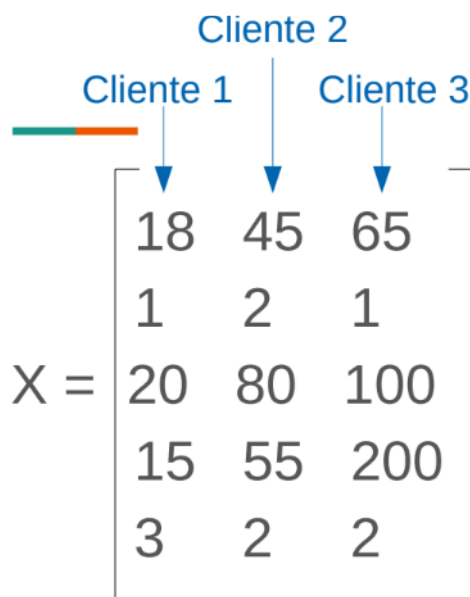
$$X = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1m} \\ x_{21} & x_{22} & \dots & x_{2m} \\ x_{31} & x_{32} & \dots & x_{3m} \\ \dots & \dots & \dots & \dots \\ x_{nm} & x_{n2} & \dots & x_{nm} \end{bmatrix}$$

$$Y = [y_{11} \ y_{12} \ \dots \ y_{1m}]$$

linha = atributo (n atributos)

coluna = exemplo de treino/teste  
(m exemplos)

Cliente 1      Cliente 2      Cliente 3


$$X = \begin{bmatrix} 18 & 45 & 65 \\ 1 & 2 & 1 \\ 20 & 80 & 100 \\ 15 & 55 & 200 \\ 3 & 2 & 2 \end{bmatrix}$$

← idade

← estado civil

← renda

← limite de crédito

← categoria cartão

$$Y = [0 \quad 1 \quad 1]$$

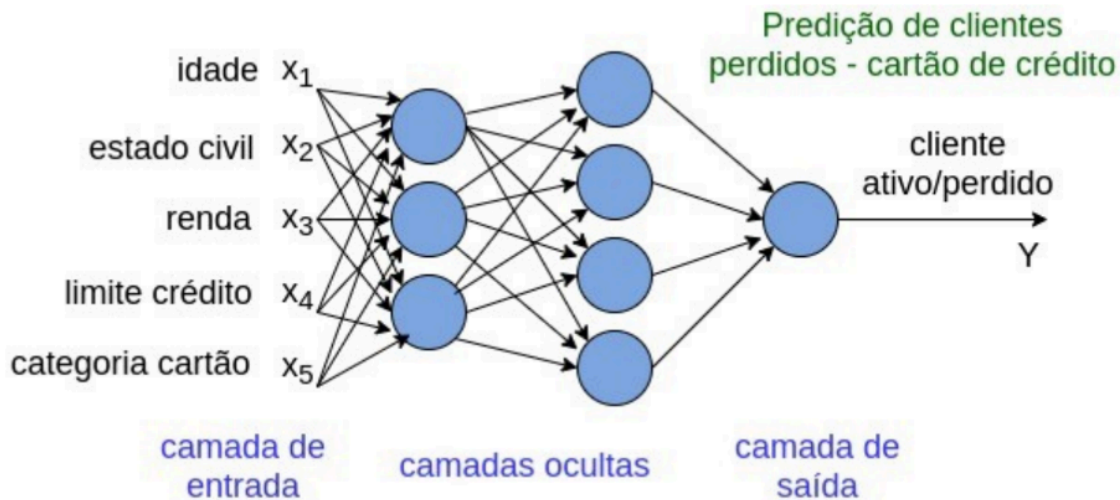
← 0 – perdido, 1 – ativo

(Imagens do slide do prof. Denilson, disponíveis em suas [videoaulas](#).)

## O que é *Deep Learning*?

É o uso de redes neurais artificiais profundas (com diversas camadas) para gerar modelos matemáticos complexos. Quanto mais camadas, mais complexo é o processamento dos modelos.

Exemplo de rede neural:



(Imagem do slide do prof. Denilson, disponível em suas [videoaulas](#)).

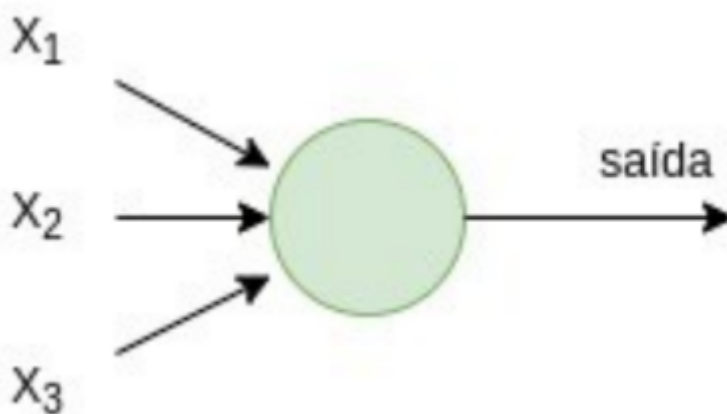
Para mais arquiteturas de redes neurais, visite: [The mostly complete chart of Neural Networks, explained | by Andrew Tch | Towards Data Science](#).

## Modelo *Perceptron*

O que é o *Perceptron*?

O ***Perceptron*** é o modelo mais simples de uma **rede neural**. Ele foi desenvolvido por Frank Rosenblatt na década de 1950 e serve como um bloco básico para redes neurais mais complexas. O *Perceptron* realiza uma tarefa de **classificação binária**, ou seja, ele tenta classificar uma (ou mais) entrada(s) em uma de duas categorias (como sim/não, verdadeiro/falso, etc.).

É o modelo mais simples de rede neural.



Na imagem acima, consideramos  $x_1$ ,  $x_2$  e  $x_3$  como atributos. (Imagem do slide do prof. Denilson, disponível em suas [videoaulas](#)).

Estrutura do *Perceptron*

### 1. Entrada (*input*):

Imagine que temos várias entradas, cada uma com um valor. Essas entradas são as informações que queremos classificar. Por exemplo, ao tentar prever clientes perdidos ou ativos em uma determinada companhia de crédito, precisamos informar ao modelo algumas informações sobre esses clientes.

Essas informações (características dos clientes) são representadas como  $x_1$ ,  $x_2$ , ..., até chegarmos a  $x_n$ , onde:

- $x_1$  é o valor da primeira característica,
- $x_2$  é o valor da segunda característica,
- e assim por diante até a  $n$ -ésima característica.

### 2. Pesos (*weights*):

Cada entrada tem um **peso** associado a ela, que indica a importância dessa característica na classificação. Esses pesos são representados como  $w_1$ ,  $w_2$ , ..., até  $w_n$ . Inicialmente, os pesos são atribuídos aleatoriamente, mas serão ajustados conforme o modelo aprende.

O peso funciona assim:

- Se uma entrada for mais importante, o seu peso será maior.
- Se uma entrada for menos importante, o peso será menor.

### 3. Cálculo da soma ponderada:

Para processarmos o modelo, precisamos multiplicar cada entrada pelo seu peso correspondente e, em seguida, somar todos esses valores.

**Fórmula do somatório:**

$$\sum_j x_j \cdot w_j = (x_1 \cdot w_1) + (x_2 \cdot w_2) + \dots + (x_n \cdot w_n)$$

- Se a soma ponderada das entradas (multiplicação das entradas pelos seus pesos) for menor ou igual a um *threshold* (viés/limiar), a saída será 0.
- Se a soma ponderada for maior que o *threshold*, a saída será 1.

O exemplo abaixo é a função **degrau** ou (*Heaviside*), geralmente utilizada em *perceptrons* simples.



**OBS:** O *threshold*/limiar é um valor que **define o ponto de decisão**. Se a soma ponderada for menor ou igual a esse valor, a saída será 0 (o *Perceptron* não "dispara"). Se a soma ponderada for maior, a saída será 1 (o *Perceptron* "dispara").

Se quiséssemos descrever a fórmula de maneira **vetorial** (ou simplificada), teríamos:

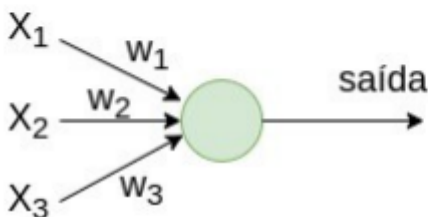
- $w$  é um **vetor de pesos**.  
 $w = (w_1, w_2, w_3, \dots)$
- $x$  é um **vetor de entradas**.  
 $x = (x_1, x_2, x_3, \dots)$

Assim, temos, a seguir, uma representação do **produto escalar** ou **interno** dos vetores  $w$  e  $x$ .

$$w^T x = (x_1 \cdot w_1) + (x_2 \cdot w_2) + \dots + (x_n \cdot w_n)$$

Usando  $b$  como viés (*bias*), temos que seu valor é o limiar (*threshold*) vezes menos um. Portanto, se o produto escalar mais o valor do viés for maior que zero, teremos um saída um, caso contrário, teremos uma saída zero.

$$\text{saída} = \begin{cases} 0 & \text{se } w^T x + b \leq 0 \\ 1 & \text{se } w^T x + b > 0 \end{cases}$$

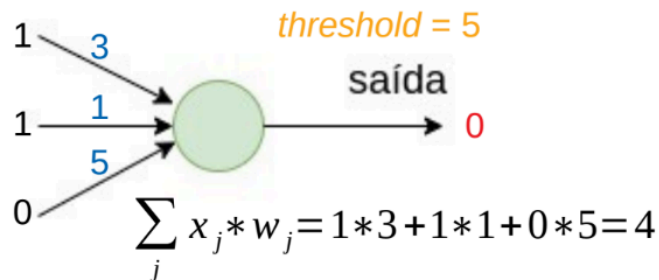


### Exemplo do modelo \*Perceptron\*\*:

No exemplo abaixo, temos como atributo de maior peso o estado do tenista (ou seja: se está ou não de bom humor) e, após o cálculo da somatória, percebemos que a decisão de jogar ou não tênis tem uma resposta negativa, visto que o cálculo não ultrapassou o limiar estabelecido de valor 5.

# Perceptron

$$\text{saída} = \begin{cases} 0 & \text{se } \sum_j x_j * w_j \leq \text{threshold} \\ 1 & \text{se } \sum_j x_j * w_j > \text{threshold} \end{cases}$$



Você joga tênis hoje?

$x_1 = 1$  (tempo bom)

$w_1 = 3$

$x_2 = 1$  (está ventando)

$w_2 = 1$

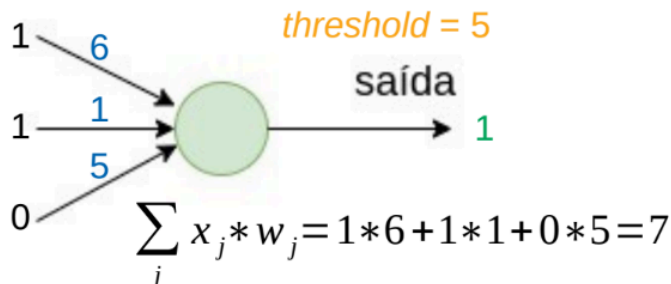
$x_3 = 0$  (estou de bom humor)

$w_3 = 5$

(Imagem do slide do prof. Denilson, disponível em suas [videoaulas](#).)

Se valorizássemos mais outro atributo, como a condição climática, teríamos um resultado diferente, com o tenista indo jogar tênis no dia de hoje, já que, nesse caso, a somatória ultrapassaria o limiar de valor 5.

$$\text{saída} = \begin{cases} 0 & \text{se } \sum_j x_j * w_j \leq \text{threshold} \\ 1 & \text{se } \sum_j x_j * w_j > \text{threshold} \end{cases}$$



Você joga tênis hoje?

$x_1 = 1$  (tempo bom)

$w_1 = 6$

$x_2 = 1$  (está ventando)

$w_2 = 1$

$x_3 = 0$  (estou de bom humor)

$w_3 = 5$

(Imagem do slide do prof. Denilson, disponível em suas [videoaulas](#).)

## Perceptron Multicamada

Uma rede de *perceptrons* de várias camadas é chamada de *Multilayer Perceptron* (MLP). Diferente de um perceptron simples, que **só pode resolver problemas linearmente separáveis**, o MLP é **capaz de resolver problemas mais complexos e não-linearmente separáveis**, graças à presença de **múltiplas camadas** e à utilização de **funções de ativação não lineares**.

Para ser considerado um *Perceptron Multicamada*, é necessário, ao menos, duas camadas além da camada de entrada (geralmente: uma camada de entrada, uma



camada oculta e uma camada de saída).

Essa rede de neurônios é completamente conectada, todos os neurônios "conversam" uns com os outros, então cada saída de um *perceptron* é "replicada" como entrada para todos os neurônios subsequentes.

O que é um problema "linearmente separável" e como esse tipo de problema se relaciona com *perceptrons*?

Imagine que você está organizando dois tipos de objetos em uma mesa: **maças** e **laranjas**. Se você conseguir **desenhar uma linha reta** no meio da mesa, de modo que todas as maçãs fiquem de um lado da linha e todas as laranjas fiquem do outro lado, isso significa que esses dois grupos de objetos são **linearmente separáveis**.

Representando as maçãs como **o** e as laranjas como **x**, teríamos:

Exemplo linearmente separável:

o o o o o <-- Maças

-----

x x x x x <-- Laranjas

Exemplo não linearmente separável:

x o x o x

-----

o x o x o

De ambos os lados da fronteira imaginária existem elementos tanto da classe laranja, como da classe maçã.

**Linearmente separável** significa simplesmente que é possível separar dois grupos de coisas com uma linha reta (ou, em casos mais complexos, com uma divisão "reta", como um plano ou hiperplano).

Um **Perceptron Simples** faz uma coisa bem específica: ele tenta encontrar uma **reta** (em 2D) ou um **hiperplano** (em mais dimensões) que separe os dados em duas classes.

Vamos ver o processo básico:

1. O **perceptron** recebe dados de entrada (que podem ter 2, 3 ou mais características).
2. Ele combina essas entradas de uma forma linear (soma ponderada) e, com base nisso, toma uma decisão.
3. A "decisão" do *perceptron* pode ser vista como traçar uma linha reta (ou um hiperplano) para separar as classes.

Se os dados forem linearmente separáveis (como no exemplo das maçãs e laranjas organizadas perfeitamente), o *perceptron* consegue encontrar essa reta. Mas se os dados **não forem linearmente separáveis**, com as frutas misturadas, por exemplo, ele não conseguirá encontrar uma solução.

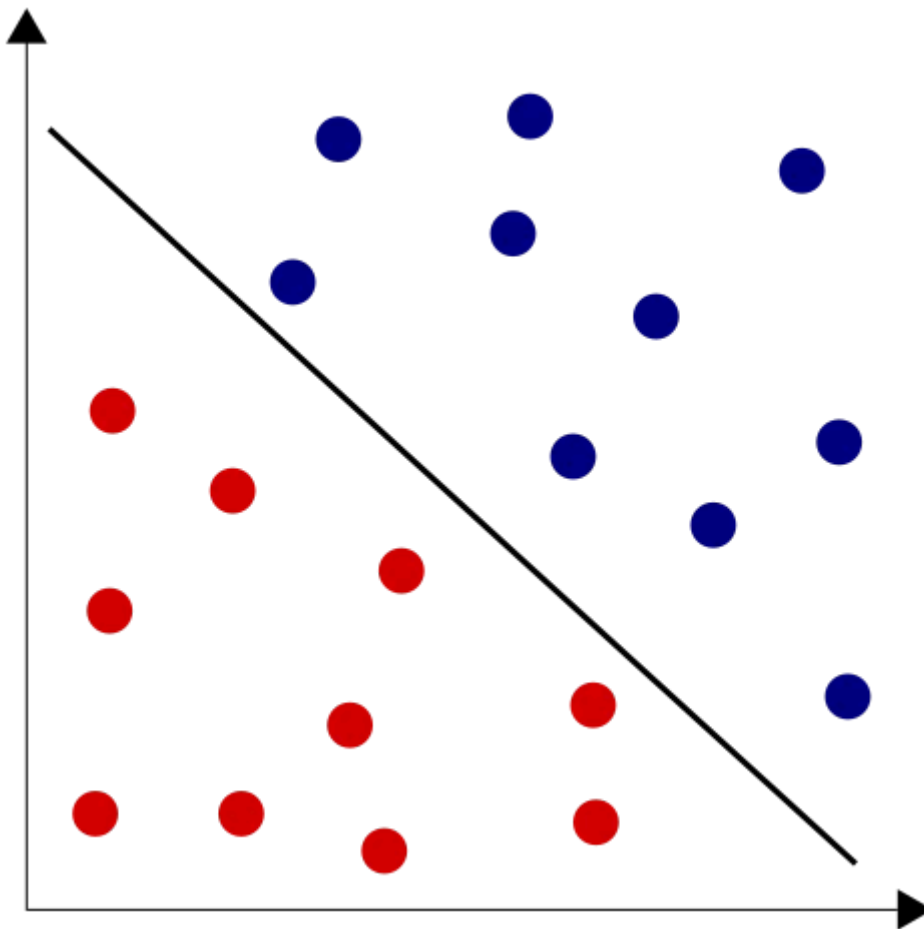


Gráfico de um problema linearmente separável.

Já um **Perceptron Multicamada** adiciona mais camadas de neurônios, permitindo que a rede combine informações de formas mais complexas. Em vez de apenas desenhar uma linha reta para separar as classes, ele pode **aprender curvas e fronteiras não lineares** para fazer uma separação mais sofisticada.

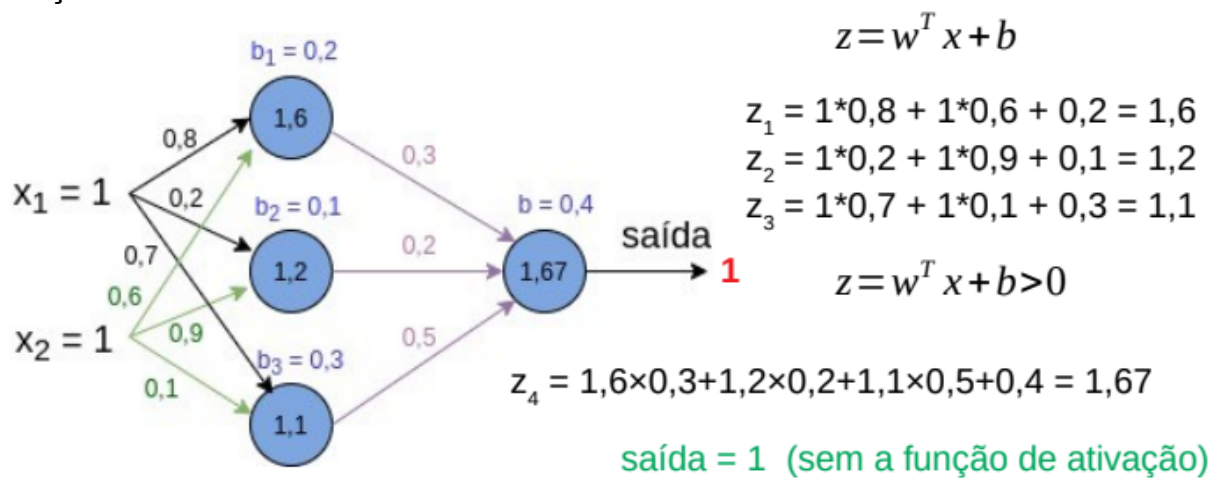
Um exemplo de problema que o *Perceptron* Simples não consegue resolver, mas um MLP consegue é a operação XOR (*Exclusive Or*).

A operação XOR retorna 1 **somente quando** uma das entradas é 1 e a outra é 0 (ou seja, quando as entradas são diferentes).

Entrada $x_1$	Entrada $x_2$	Saída (XOR)
0	0	0
0	1	1
1	0	1
1	1	0

Repare que, mesmo que tentemos, não vamos conseguir traçar uma linha que divida as classes entre "0" e "1" de maneira exata. O *Perceptron* Simples só pode traçar uma **reta** (ou plano) para separar os dados, e isso não é suficiente para resolver o XOR, que exige uma separação **não linear**, que o MLP consegue resolver.

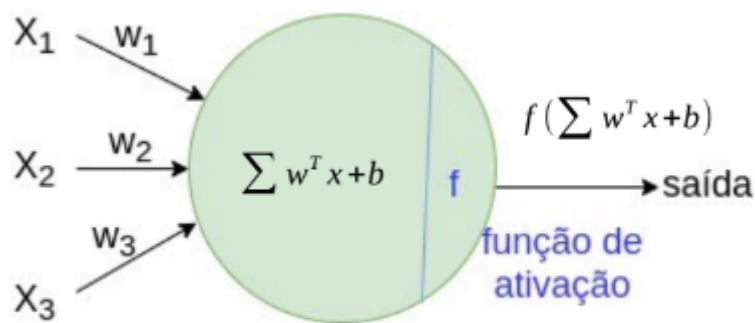
Exemplo de cálculo de um *Perceptron* Multicamada sem considerarmos a função de ativação:



(Imagem do slide do prof. Denilson, disponível em suas [videoaulas](#)).

### Função de ativação

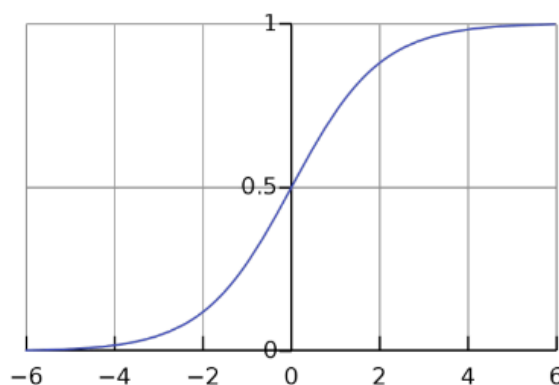
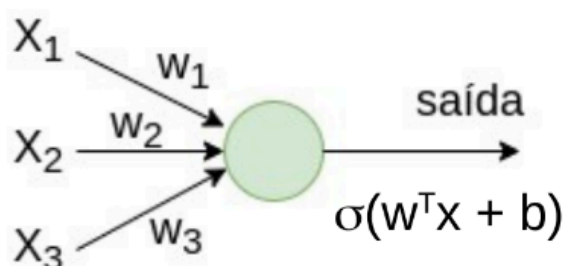
Cada neurônio é caracterizado pelo peso, *bias* (viés) e a função de ativação. Enquanto os neurônios fazem uma transformação linear na entrada pelos pesos e *bias*, a função de ativação faz uma transformação **não linear**. Isso é o que torna o MLP tão diferente do SLP (*Single Layer Perceptron*).



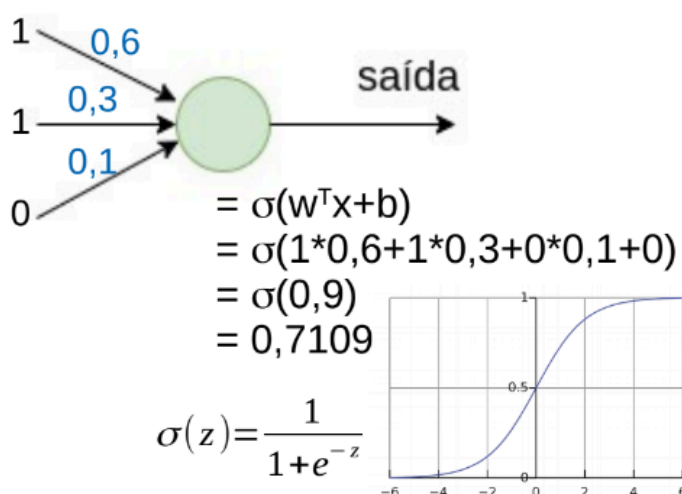
(Imagem do slide do prof. Denilson, disponível em suas [videoaulas](#)).

### Função Sigmóide

É um exemplo de função de ativação, mapeando a saída  $z$  para o intervalo  $(0,1)$  e é usada frequentemente em redes neurais para introduzir não linearidade. Além disso, aparece, principalmente, na camada de saída para problemas de **classificação binária** e não é muito eficaz quando os gradientes se tornam muito pequenos.



$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



Você joga tênis hoje?

$x_1 = 1$  (tempo bom)

$w_1 = 0,6$

$x_2 = 1$  (está ventando)

$w_2 = 0,3$

$x_3 = 0$  (estou de bom humor)

$w_3 = 0,1$

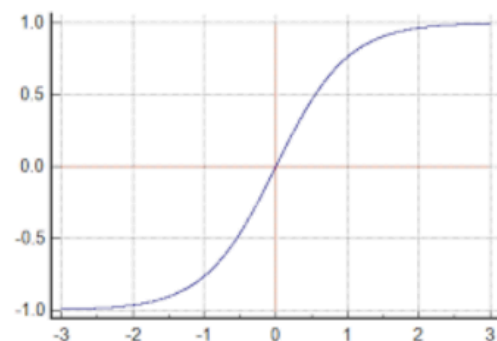
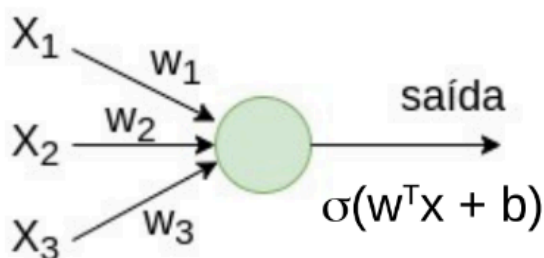
$b = 0$

(Imagens do slide do prof. Denilson, disponíveis em suas [videoaulas](#)).

Função TanH (tangente hiperbólica)

A função tangente hiperbólica ( $\tanh(x)$ ) é uma derivação da função sigmoide. No entanto, ela é **simétrica em relação à origem** (o ponto  $\tanh(0)$  está no centro do gráfico). A função  $\tanh(x)$  tem um **intervalo de saída entre -1 e 1**, enquanto a sigmoide tem saída entre 0 e 1. Isso permite que a  **$\tanh(x)$  lide melhor com valores negativos e positivos em entradas de redes neurais**, ajudando a centrar os dados em torno de zero, o que pode melhorar o desempenho e a convergência durante o treinamento.

## Função Tanh (Tangente Hiperbólica)



$$\sigma(z) = \frac{2}{1 + e^{-2z}} - 1$$

(Imagens do slide do prof. Denilson, disponíveis em suas [videoaulas](#)).

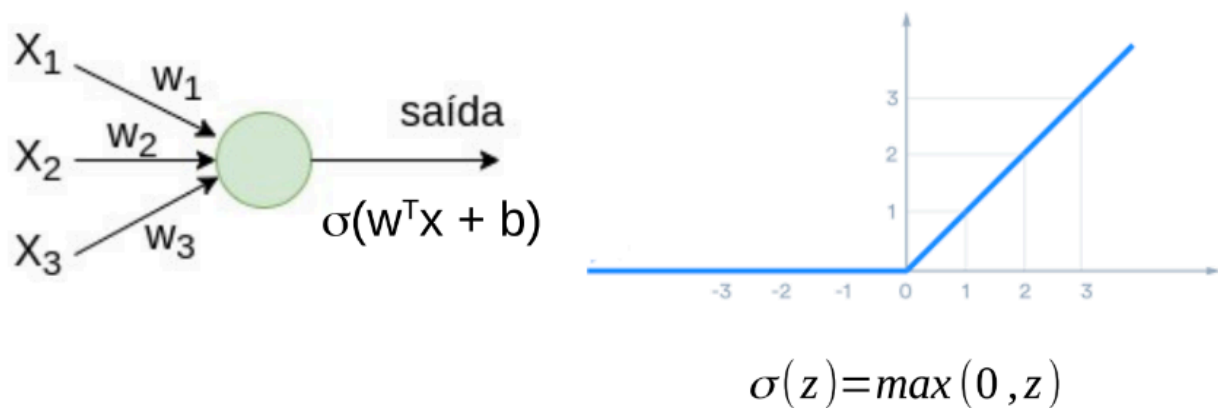
Função ReLU (*Rectified Linear Unit*)

Essa é uma das funções de ativação mais populares usadas em redes neurais profundas, especialmente em redes convolucionais (CNNs). Ela retorna o valor da entrada  $x$  se  $x$  for positivo; caso contrário, retorna 0. Graficamente, a ReLU é uma linha reta com inclinação 1 para valores positivos e colada ao eixo  $x$  para valores negativos.

Além disso, a função ReLU não ativa todos os neurônios ao mesmo tempo, o que torna a rede esparsa e a computação se torna fácil e eficiente.

Uma de suas desvantagens é que apresenta problemas quando os gradientes são muito pequenos (próximos de zero).

# Função ReLU (*Rectified Linear Unit*)



(Imagens do slide do prof. Denilson, disponíveis em suas [videoaulas](#)).

## PROBLEMA DO NEURÔNIO MORTO

Um **neurônio morto** em redes neurais ocorre quando uma unidade de ativação (ou neurônio) deixa de atualizar seu valor de saída durante o treinamento, ou seja, ela **sempre retorna zero** independentemente da entrada. Esse problema é comumente associado à função de ativação **ReLU**.

Na função **ReLU** ( $\text{ReLU}(x) = \max(0, x)$ ), qualquer valor de entrada negativo resulta em uma saída de 0. Se muitos neurônios recebem entradas negativas repetidamente, esses neurônios podem **parar de aprender** porque o gradiente para esses valores negativos também será zero. Quando isso acontece, esses neurônios se tornam "mortos" ou inativos, pois suas saídas continuam sendo 0 durante todo o treinamento, sem a capacidade de mudar.

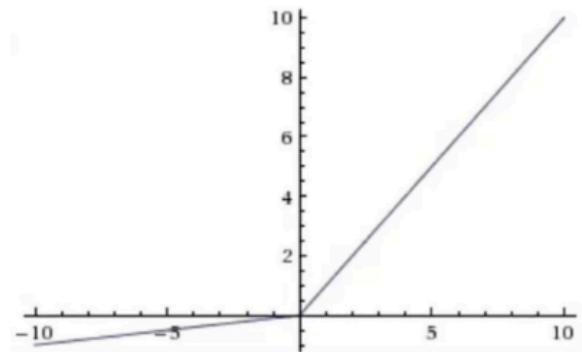
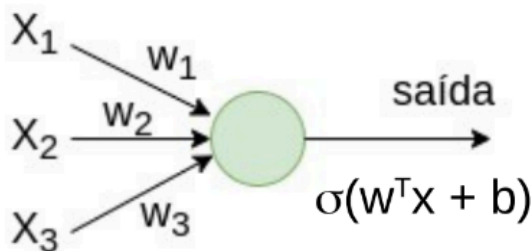
A **Leaky ReLU** (que veremos a seguir) resolve parcialmente o problema do neurônio morto ao permitir que valores negativos passem com um pequeno coeficiente  $\alpha$ . Isso garante que, mesmo quando as entradas forem negativas, a ativação não será completamente zero, mas sim um valor pequeno. Com isso, o neurônio ainda tem a chance de continuar aprendendo e ajustando seus pesos.

## Função Leaky ReLU (*Leaky Rectified Linear Unit*)

É uma versão da ReLU, que busca diminuir problemas como o do "neurônio morto", citado acima. A grande diferença entre as duas é que, para valores negativos, utiliza-se uma constante  $\alpha$  (que costuma ter um valor muito pequeno) para auxiliar no cálculo.

$$\text{Leaky ReLU}(x) = \begin{cases} x & \text{se } x \geq 0, \\ \alpha x & \text{se } x < 0 \end{cases}$$

## Função Leaky ReLU

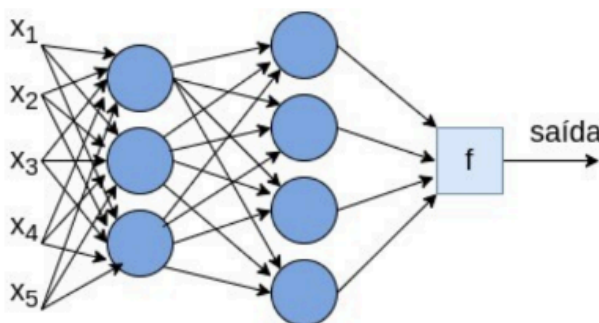


$$\sigma(z) = \begin{cases} \alpha z & \text{se } z < 0 \\ z & \text{se } z \geq 0 \end{cases}$$

### Função Softmax

É um tipo de função sigmóide usada para converter uma lista de números reais (que podem ser positivos, negativos ou zero) em probabilidades. Cada número é transformado em um valor entre 0 e 1, e todos os valores juntos somam 1. Isso é especialmente útil em problemas de classificação multiclasse, onde queremos saber a probabilidade de uma entrada pertencer a diferentes classes. Além disso, ela não é usada em cada camada oculta da rede neural, mas apenas na camada de saída.

## Função Softmax (ou softargmax)



$$\sigma(z_j) = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \text{ for } j = 1, \dots, K$$

### Função de custo

A **função de custo** é uma métrica que avalia o quão bem o modelo está realizando suas previsões. Ela calcula a diferença entre a saída prevista pelo modelo ( $\hat{y}$ ) e o valor real esperado ( $y$ ), fornecido pelos dados de treinamento.

Quando treinamos uma rede neural, o objetivo é ajustar os **pesos** ( $w$ ) e os **bias** ( $b$ ) de tal forma que a saída prevista ( $\hat{y}(i)$ ) para uma dada entrada ( $x(i)$ ) se aproxime o máximo possível da saída correta ( $y(i)$ ). Quanto menor essa diferença, melhor o modelo se torna em suas previsões.

Função de Custo Quadrático (MSE - *Mean Squared Error*)

Se o vetor  $v$  representa os **pesos** do modelo ( $w$ ), a função de custo está diretamente relacionada ao vetor  $v$ . Em termos simples, o vetor  $v$  é um conjunto de números que ajustam como cada característica ou variável de entrada afeta a previsão final do modelo. O modelo tenta encontrar a melhor configuração desses números, ou seja, dos **pesos** ( $w$ ), que minimize o valor da função de custo  $C(w, b)$ . Isso significa que o modelo ajusta seus **pesos** para que as previsões fiquem o mais próximo possível dos valores reais.

$$C(w, b) = \frac{1}{m} \sum_{i=1}^m \|y^{(i)} - \hat{y}^{(i)}\|^2$$

- **$C(w, b)$** : Essa é a função de custo, e depende dos parâmetros do modelo, ou seja, os **pesos** ( $w$ ) e o **bias** ( $b$ ). O objetivo do treinamento do modelo é minimizar essa função.
- **$m$** : O número total de exemplos no conjunto de dados de treinamento. A função de custo média considera a soma dos erros em todos os exemplos de treinamento, dividida pelo número total de exemplos para normalizar o valor.
- **$y_i$** : O valor **real** da saída esperada para o exemplo  $i$  no conjunto de treinamento. Isso significa que cada exemplo tem um valor de referência conhecido.
- **$\hat{y}_i$** : O valor **previsto** pelo modelo para o exemplo  $i$ . Esta é a saída calculada pelo modelo com os parâmetros atuais ( $w, b$ ) aplicados ao exemplo de entrada.
- **$(y_i - \hat{y}_i)^2$** : A diferença entre a saída real ( $y_i$ ) e a saída prevista ( $\hat{y}_i$ ) é chamada de **erro**. Elevamos essa diferença ao quadrado para garantir que todos os erros sejam positivos e para dar mais peso a erros maiores (os erros maiores terão um impacto maior no valor final).

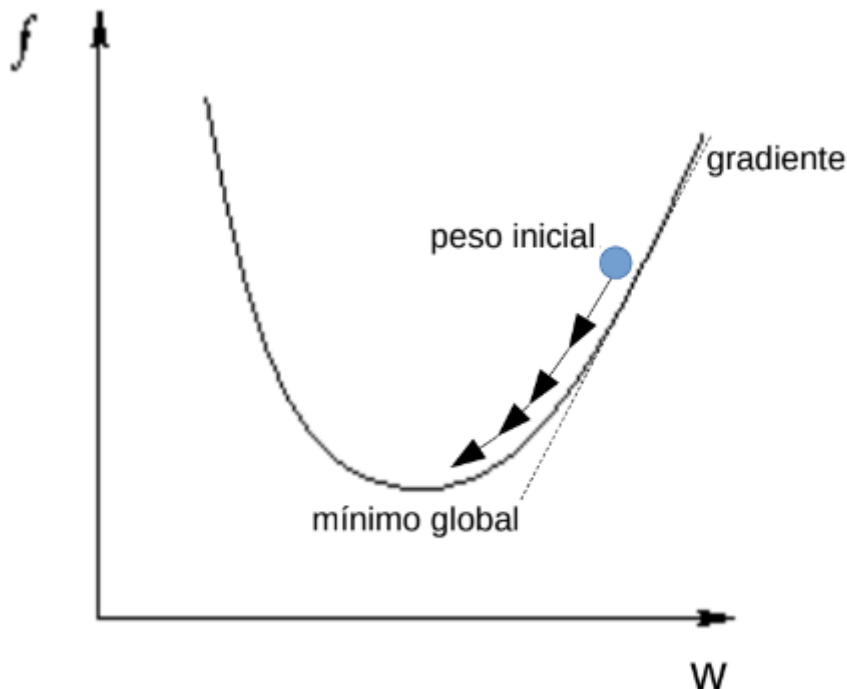
## Descida do gradiente

A **descida do gradiente** é uma técnica de otimização usada para ajustar os **pesos** ( $w$ ) e o **bias** ( $b$ ) de um modelo, de modo que ele possa fazer previsões mais precisas. Em



termos simples, é como um **processo de "tentativa e erro"** que o modelo usa para **minimizar a função de custo** — ou seja, o modelo vai ajustando seus pesos aos poucos para reduzir os erros nas previsões.

O **gradiente** é um vetor que contém todas as derivadas parciais de uma função em relação às suas variáveis. Em termos simples, ele fornece a inclinação ou a direção em que uma função está mudando em um ponto específico. O gradiente é frequentemente representado por  $\nabla$  (nabla) seguido da função que estamos considerando.



(Imagem do slide do prof. Denilson, disponível em suas [videoaulas](#)).

Como funciona?

Aqui está o processo básico:

1. **Escolha de um ponto de partida (valores iniciais de  $w$  e  $b$ ):**

O modelo começa com valores iniciais aleatórios para os pesos ( $w$ ) e o bias ( $b$ ).

2. **Cálculo do erro (função de custo):**

O modelo faz previsões usando os valores iniciais de  $w$  e  $b$  e compara essas previsões com os valores reais do conjunto de treinamento. Com base na diferença entre as previsões e os valores reais (erro), ele calcula a **função de custo**.

3. **Cálculo do gradiente:**

Aqui entra a **derivada**! O gradiente é calculado usando as **derivadas parciais** da função de custo em relação aos pesos ( $w$ ) e ao bias ( $b$ ). A derivada nos diz a

**inclinação** da função de custo em um determinado ponto, ou seja, se estamos "subindo" (erro aumentando) ou "descendo" (erro diminuindo). O objetivo da descida do gradiente é seguir sempre na direção de descida para minimizar o erro.

- Se a derivada for **positiva**, isso significa que estamos subindo na curva da função de custo e precisamos ajustar os pesos e o bias para o lado oposto, diminuindo o erro.
- Se a derivada for **negativa**, estamos na direção certa, pois o erro está diminuindo. Continuamos ajustando até a derivada se aproximar de zero, o que indica que estamos próximos do **mínimo da função de custo**.

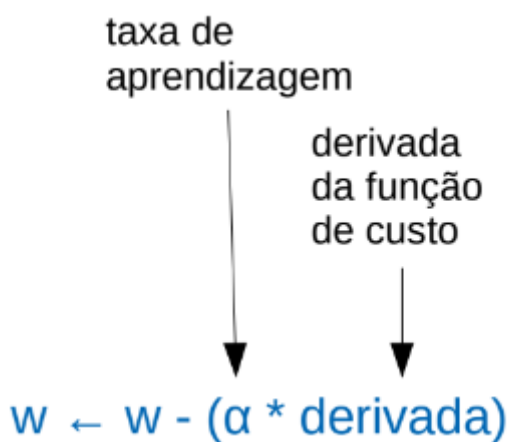
#### 4. Ajuste dos pesos e bias:

O modelo ajusta os valores de **w** e **b** na direção oposta à do gradiente (a inclinação), diminuindo aos poucos o erro. Esses ajustes são feitos multiplicando o gradiente por um número pequeno chamado **taxa de aprendizado** (learning rate), que controla o tamanho dos passos que o modelo vai dar. Passos muito grandes podem fazer com que o modelo "passe do ponto", enquanto passos muito pequenos podem demorar muito para encontrar o mínimo.

#### 5. Repetição do processo:

O processo se repete várias vezes: o modelo ajusta **w** e **b**, recalcula o erro, ajusta novamente, e assim por diante, até que a função de custo esteja o mais próximo possível do mínimo. Assim, damos seguidos passos para frente e para trás, usando **backpropagation**.

Quando chegamos ao fim, dizemos que o modelo "convergiu", ou seja, encontrou a melhor combinação de pesos para minimizar o erro nas previsões.



(Imagem do slide do prof. Denilson, disponível em suas [videoaulas](#)).

#### **Backpropagation** (retropropagação)

Algoritmo usado para calcular o gradiente (derivadas) da função de custo. Ele é dividido em duas fases.

##### 1. Fase de *Feedforward*

- **Entrada de Dados:** Os dados de entrada são passados através da rede neural.
- **Cálculo da Saída:** A rede realiza cálculos em cada camada e produz uma saída. Essa saída é comparada com os valores reais (ou esperados) usando uma função de custo (como o erro quadrático médio).

## 2. Fase de Retropropagação (Backward Pass)

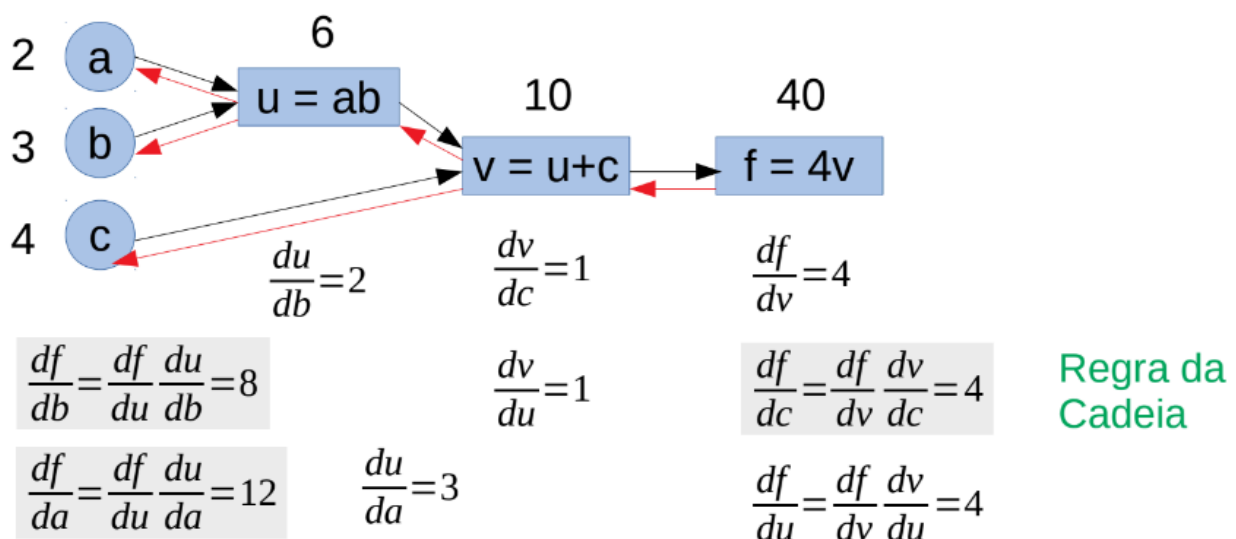
Na fase de retropropagação, seguimos os passos abaixo:

### 2.1 Cálculo do Erro na Camada Final

- **Cálculo do Gradiente da Função de Perda:** Primeiramente, **calculamos o erro na camada de saída** (camada final) da rede. Isso é **feito usando a função de custo**, que nos diz quão longe nossas previsões estão dos valores reais.

### 2.2 Aplicação da Regra da Cadeia

- **Atualização dos Pesos:** Usamos a **regra da cadeia** para **propagar esse erro de volta através da rede**, camada por camada. A ideia é que precisamos entender como o erro na saída impacta cada peso na rede.
  - Para cada camada, calculamos a contribuição do erro aos pesos dessa camada. Isso envolve calcular a derivada da função de ativação utilizada em cada neurônio, que nos diz como a saída do neurônio muda em relação à sua entrada.
- **Retropropagação do Erro:** O erro calculado na camada final é propagado para trás, atualizando os pesos com base na contribuição de cada um deles para o erro total. A quantidade que ajustamos é baseada no **gradiente** (que indica a direção e o tamanho do ajuste) multiplicado por um número pequeno chamado **taxa de aprendizado**. Essa taxa controla quão grande é o passo que estamos dando para ajustar os pesos.



(Imagem do slide do prof. Denilson, disponível em suas [videoaulas](#)).

# Descida do Gradiente

Inicialize os pesos  $w$  e  $b$

Repita (com os valores de pesos atuais)

- Passo para frente {
- passe as entradas do treinamento pela rede
  - calcule a saída (predição)

- Passo para trás {
- compare a saída com a resposta correta
  - calcule o erro
  - retroaja esse erro (*backpropagation*), ajustando os pesos  $w$  e  $b$  dos neurônios em cada camada

$$w \leftarrow w - \alpha \frac{dC}{dw}$$

$$b \leftarrow b - \alpha \frac{dC}{db}$$

Até conseguir o custo zero (ou próximo dele)

(Imagem do slide do prof. Denilson, disponível em suas [videoaulas](#)).