Perceptron

tags: #python_full_ai #perceptron

Perceptron: Fundamentos Teóricos e matemáticos

O Perceptron constitui a base fundamental para o desenvolvimento de redes neurais mais complexas. Este material explora sua teoria, formulação matemática e aplicações práticas.

Fundamentos Históricos e Conceituais

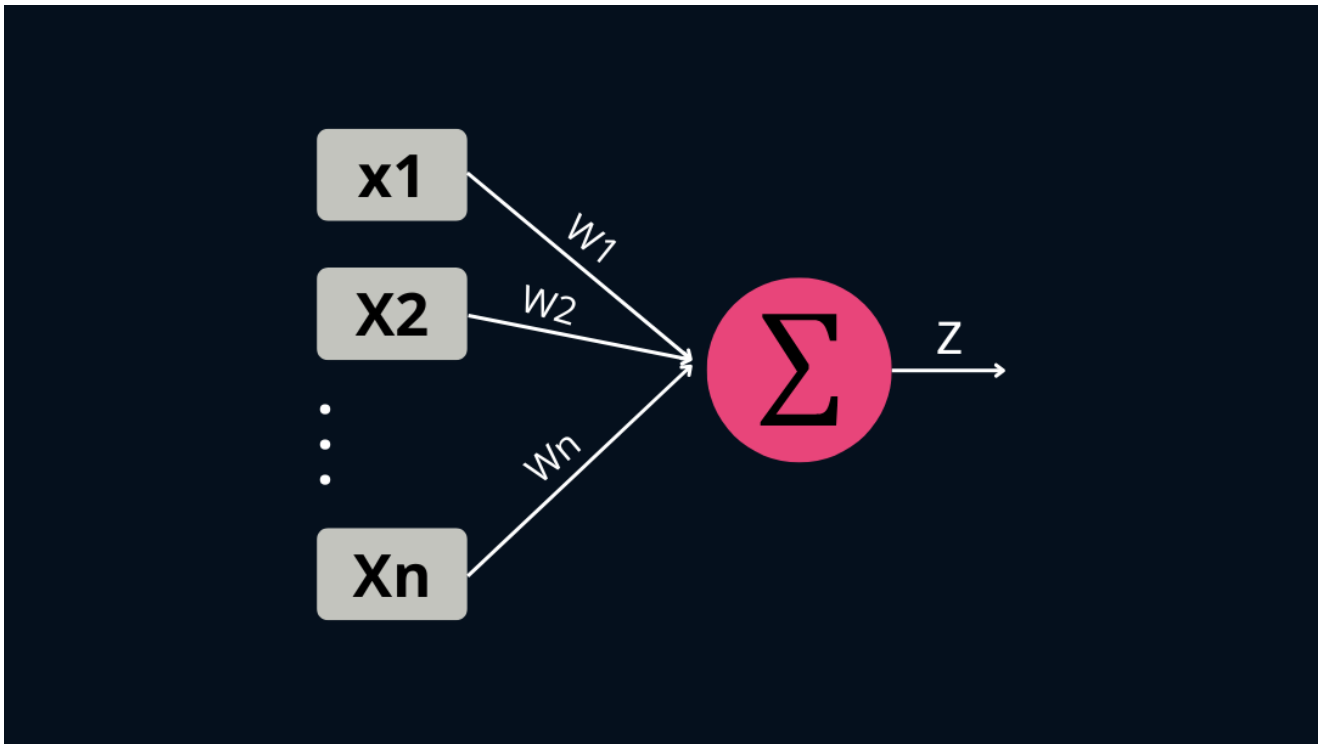
Em 1943, Warren McCulloch e Walter Pitts publicaram o artigo que revolucionou todo o nosso entendimento moderno sobre inteligências artificiais como as conhecemos hoje, desde o ChatGPT, MidJourney, ClaudeAI, entre outros.

"A Logical Calculus of the Ideas Immanent in Nervous Activity" nos deu a ideia do primeiro neurônio artificial, baseado nos neurônios biológicos do nosso próprio cérebro, chamado de **Perceptron**.

O Perceptron é um classificador binário de problemas linearmente separáveis. Ele é a forma mais elementar de uma rede neural e, por isso, individualmente, não é capaz de resolver muitos problemas, já que seria equiparado a um único neurônio do nosso cérebro.

Contudo, ele nos permite a criação do **Multilayer Perceptron**, um conjunto de Perceptrons ligados em camadas, que dá origem às redes neurais.

Funcionamento do Perceptron



Onde:

- x_1, x_2, ..., x_n são as entradas
- w_1, w_2, ..., w_n são os pesos que representam o aprendizado do modelo
- Σ representa a soma ponderada das entradas pelos pesos
- Z é o resultado da somatória

A imagem acima serve apenas como uma representação humanizada do Perceptron para fins didáticos. No entanto, para o computador, o Perceptron nada mais é do que uma função matemática.

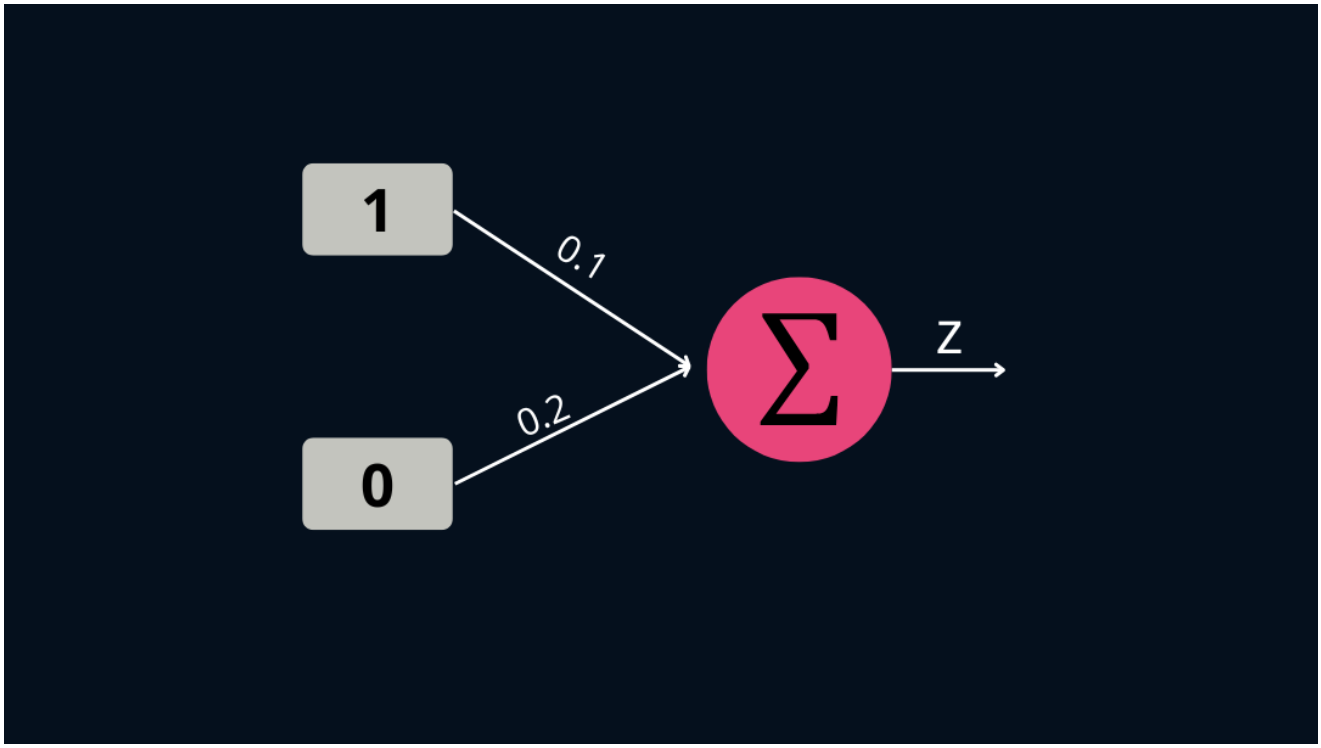
$$z = f(x_1, x_2)$$
$$f(x_1, x_2) = (x_1 \cdot w_1) + (x_2 \cdot w_2)$$

O problema de representar o Perceptron dessa maneira é que a função resultante só funciona com dois valores de entrada. Para que ele seja mais útil e escalável, precisamos expressá-lo de forma dinâmica, capaz de lidar com múltiplas entradas.

$$Z = \sum_{i=1}^{N} x_i w_i$$

Os pesos, no início do Perceptron, são definidos aleatoriamente e, conforme o treinamento avança, vão se ajustando aos dados.

Vamos adicionar dois valores de entrada e definir pesos aleatórios para ilustrar esse processo.



Adicionando os dados na função temos:

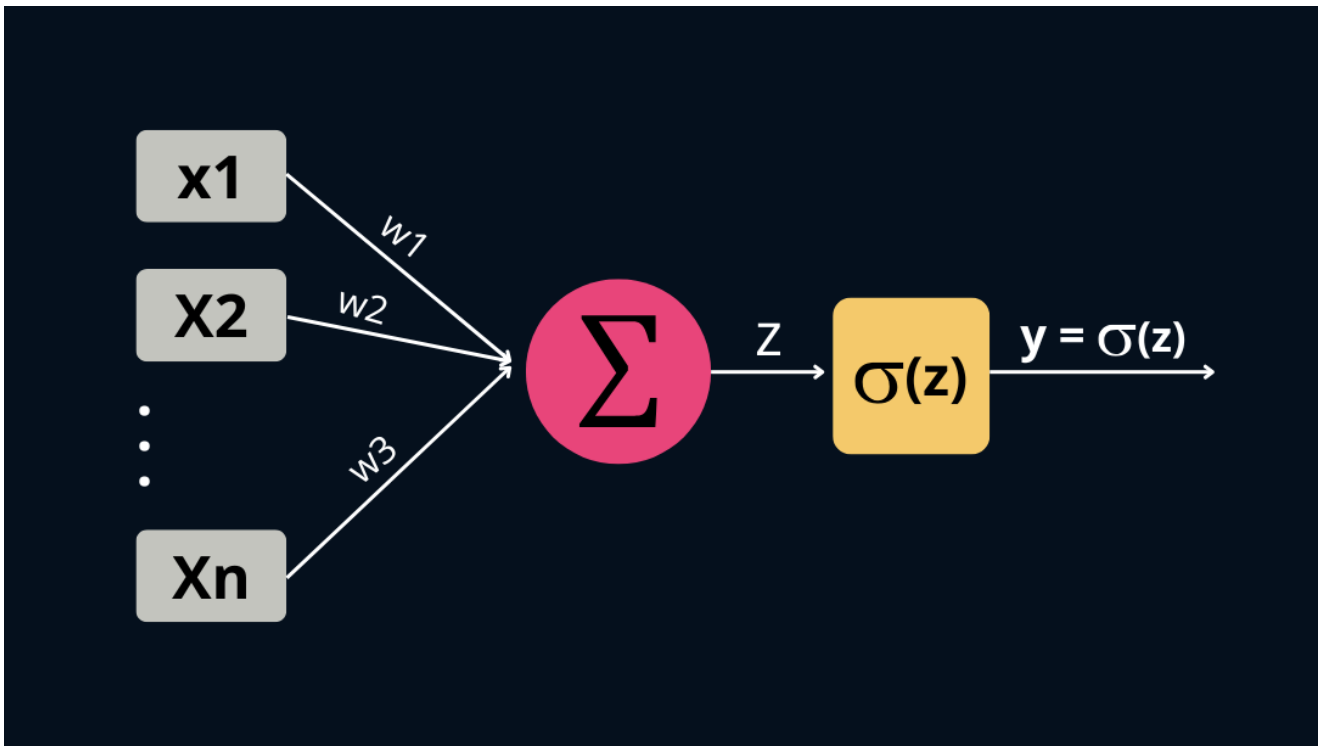
$$f(x_1, x_2) = (1 \cdot 0.1) + (0 \cdot 0.2)$$
$$Z = 0.1$$

Nesse modelo, ainda precisamos adicionar a função de ativação. Ela é responsável por transformar a soma ponderada realizada anteriormente em uma saída útil, que faça sentido para o problema.

Imagine que estamos criando uma rede neural para classificar radiografias, identificando se o osso está ou não quebrado. Resultados como 150, -20 ou 600 não fariam muito sentido.

Por isso, precisamos normalizar esses dados de forma que se encaixem no contexto do problema. Nesse exemplo, poderíamos usar como função de ativação a `step function`, que transforma resultados maiores ou iguais a 0 em 1, e valores menores que 0 em 0.

Dessa forma, teríamos a garantia de que o resultado da rede neural artificial será sempre 0 ou 1 — e poderíamos interpretar 0 como "osso não quebrado" e 1 como "osso quebrado".



Podemos perceber que a soma ponderada, representada por *Z*, passará por uma função matemática que irá processar esse valor. Agora sim, o resultado dessa função será a saída oficial do neurônio, representada por *Y*.

Matematicamente, isso pode ser representado por:

$$y = \sigma(w_1 x_1 + w_2 x_2 + \cdots + w_n x_n)$$

$$y = \sigma\left(\sum_{i=1}^{n} w_i x_i\right)$$

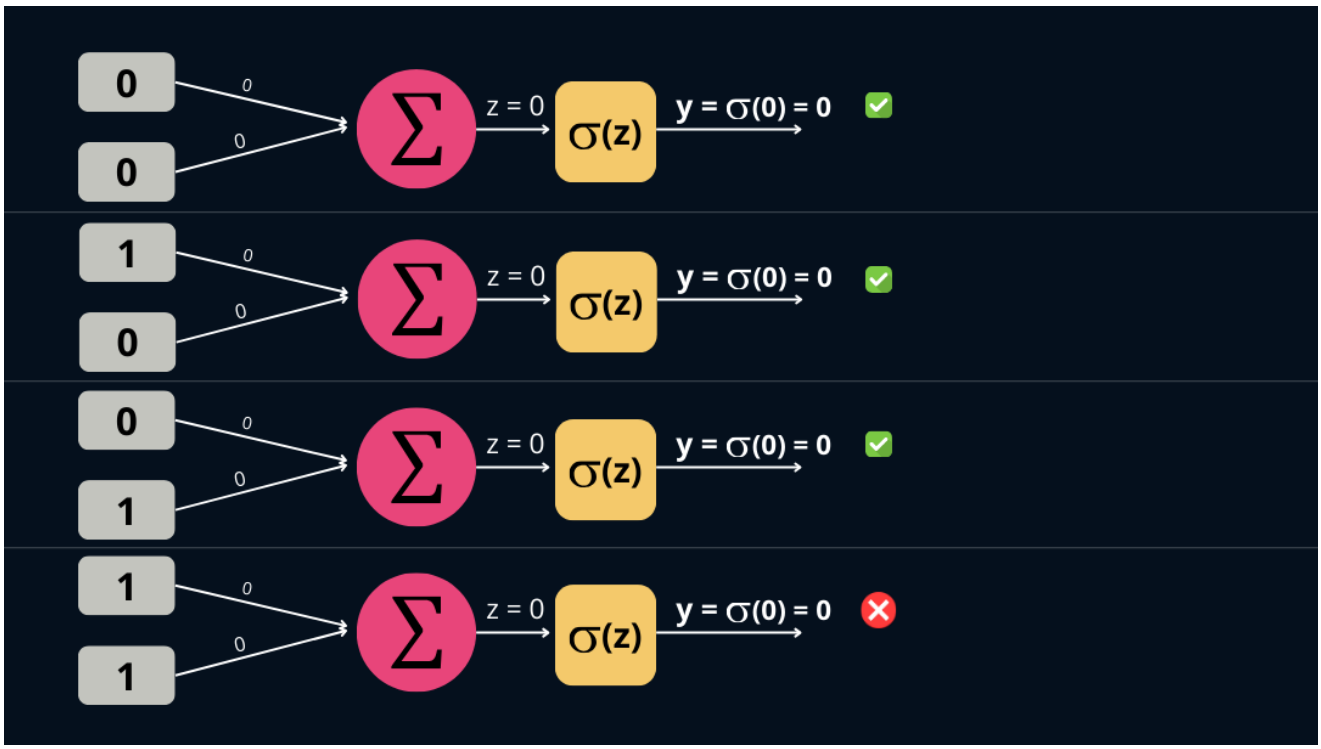
Aplicação prática do perceptron

A	B	S
0	0	1
0	1	0
1	0	0
1	1	1

Vamos assumir que desejamos "ensinar" ao nosso neurônio a responder corretamente à tabela da operação lógica AND.

Vamos iniciá-lo com pesos aleatórios e utilizar, como função de ativação, a `step function`.

$$\sigma(x) = \begin{cases} 1, & \text{se } x \geq 1 \\ 0, & \text{se } x < 1 \end{cases}$$



Na imagem acima, iniciamos os pesos aleatoriamente com o valor 0, e o Perceptron não está respondendo corretamente à tabela AND. Isso significa que precisamos reajustar os pesos do neurônio para encontrar uma combinação que forneça as respostas corretas.

Essa etapa é chamada de **“Backpropagation”** ou **“Retropropagação”**, pois envolve voltar do resultado para os pesos, ajustando-os com base no erro cometido.

O funcionamento é simples: nos casos em que a rede neural artificial fornece uma resposta incorreta, aplicamos uma equação matemática para ajustar os pesos, aproximando o neurônio da resposta esperada.

$$w_i^{(\text{novo})} = w_i^{(\text{atual})} + (\eta \cdot x_i \cdot erro)$$

- Wi é o peso atualizado
- Wi é o peso a ser atualizado
- η (eta) é a taxa de aprendizagem
- Xi é o valor da entrada n
- erro (diferença entre saída desejada e calculada) (d−y)

Vamos aplicar a fórmula para correção dos pesos no caso em que tivemos uma saída do perceptron diferente da saída desejada.

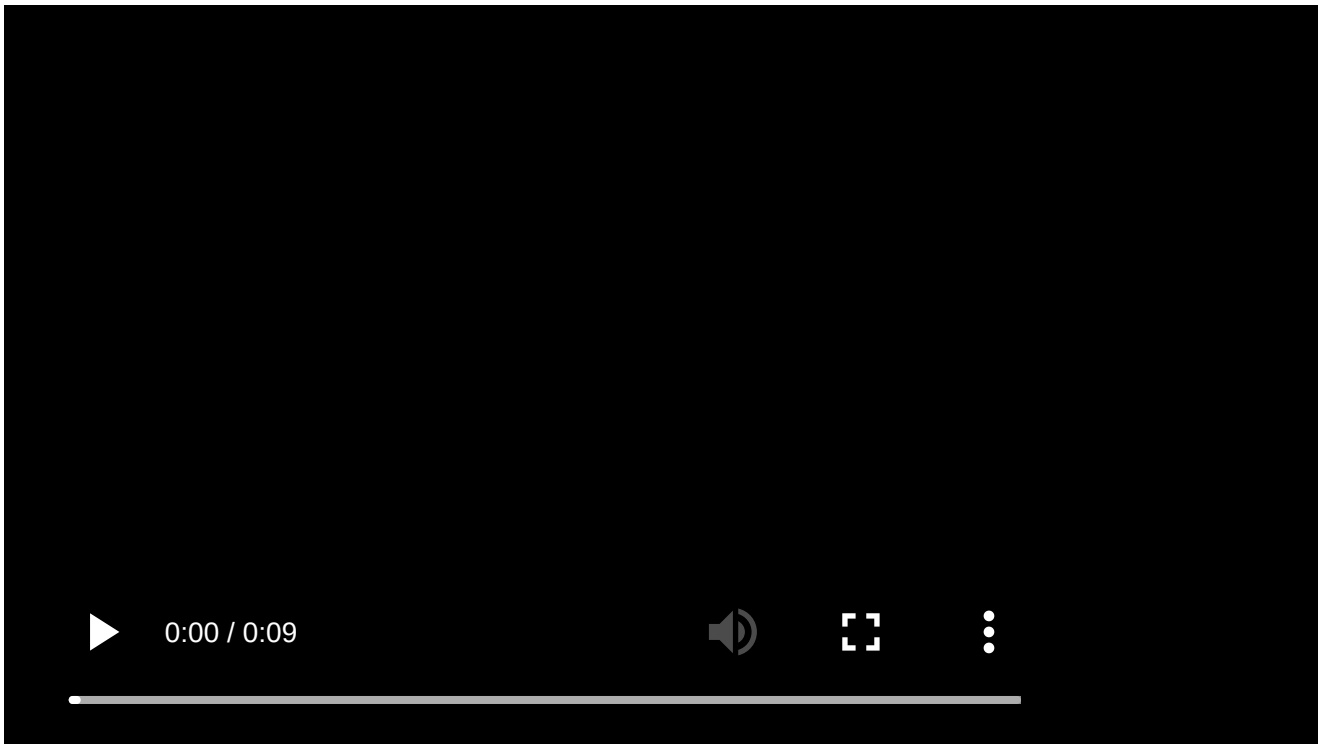
$$w_1 = 0 + (0.5 \cdot 1 \cdot 1)$$

$$w_1 = 0.5$$

$$w_1 = 0 + (0.5 \cdot 1 \cdot 1)$$

$$w_2 = 0.5$$

Com essa correção nos pesos o perceptron consegue responder corretamente para todas as possibilidades da tabela AND.

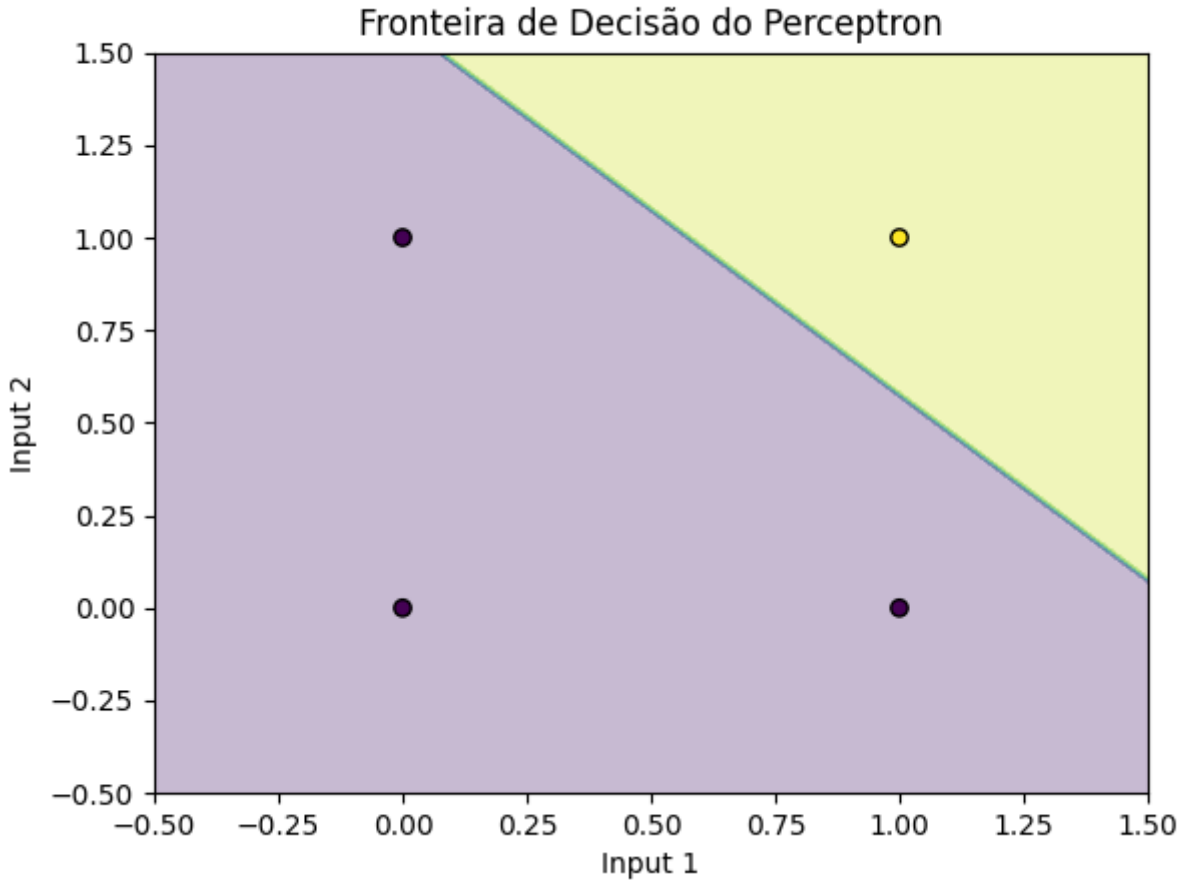


Na taxa de aprendizagem, devemos tomar cuidado com o valor que iremos definir. Um valor muito baixo traz maior precisão à rede, mas, consequentemente, torna o treinamento mais demorado.

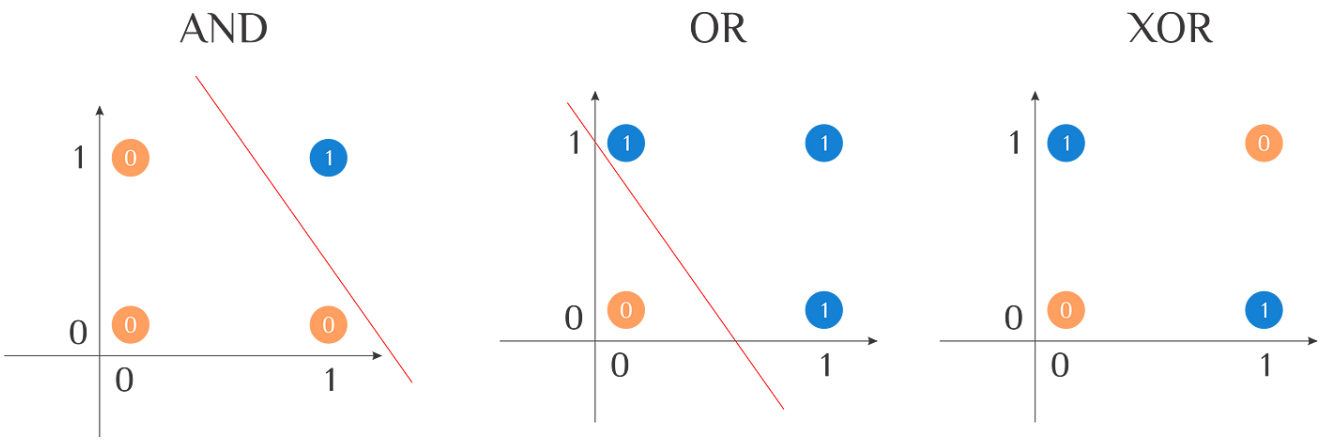
Por outro lado, valores muito altos aceleram o treinamento, porém podem comprometer a performance da rede, resultando em menor precisão.

Limitação do Perceptron

O Perceptron até aqui aparentou ser um algoritmo muito poderoso, mas a verdade é que ele por si só é capaz de resolver apenas problemas linearmente separáveis (O que não são os casos de problemas reais).



A imagem abaixo mostra a fronteira de decisão do perceptron que treinamos para resolver uma tabela AND, repare que ele gera uma linha reta para classificar os dados.



Repare que no caso do XOR não é possível resolver o problema com uma reta, e nesse caso precisaríamos de técnicas mais avançadas como o MLP que aprenderemos no próximo módulo desse curso.

Sigmoid

A função sigmoid é responsável por introduzir não linearidade no sistema. Ela transforma uma saída linear em valores entre 0 e 1, os quais são comumente interpretados como probabilidades em tarefas de classificação. Por exemplo, em uma rede neural treinada para classificar uma radiografia e identificar se um osso está ou não fraturado, um resultado de 0,82 pode ser interpretado como 82% de chance de o osso realmente estar quebrado.

Matematicamente representada por:

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

o e na função é uma constante matemática conhecida como número de Euler, equivale a aproximadamente 2,71828.

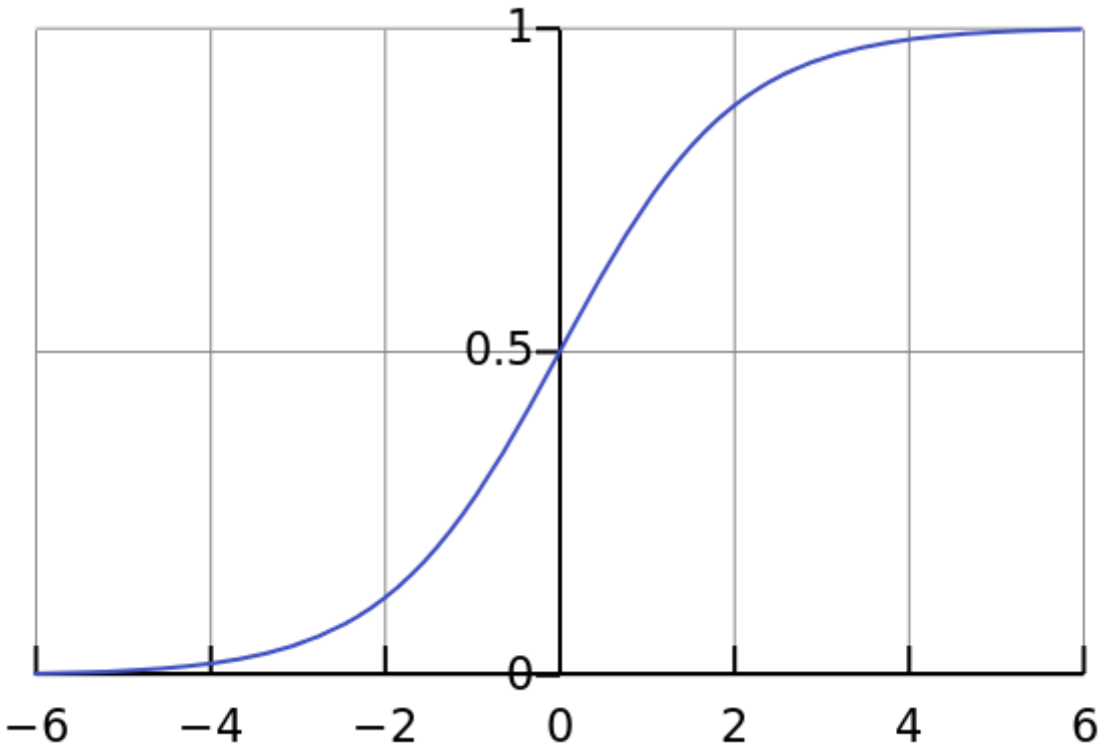
Esse valor é calculado pela seguinte somatória:

$$e = \sum_{k=0}^{\infty} \frac{1}{k!}$$

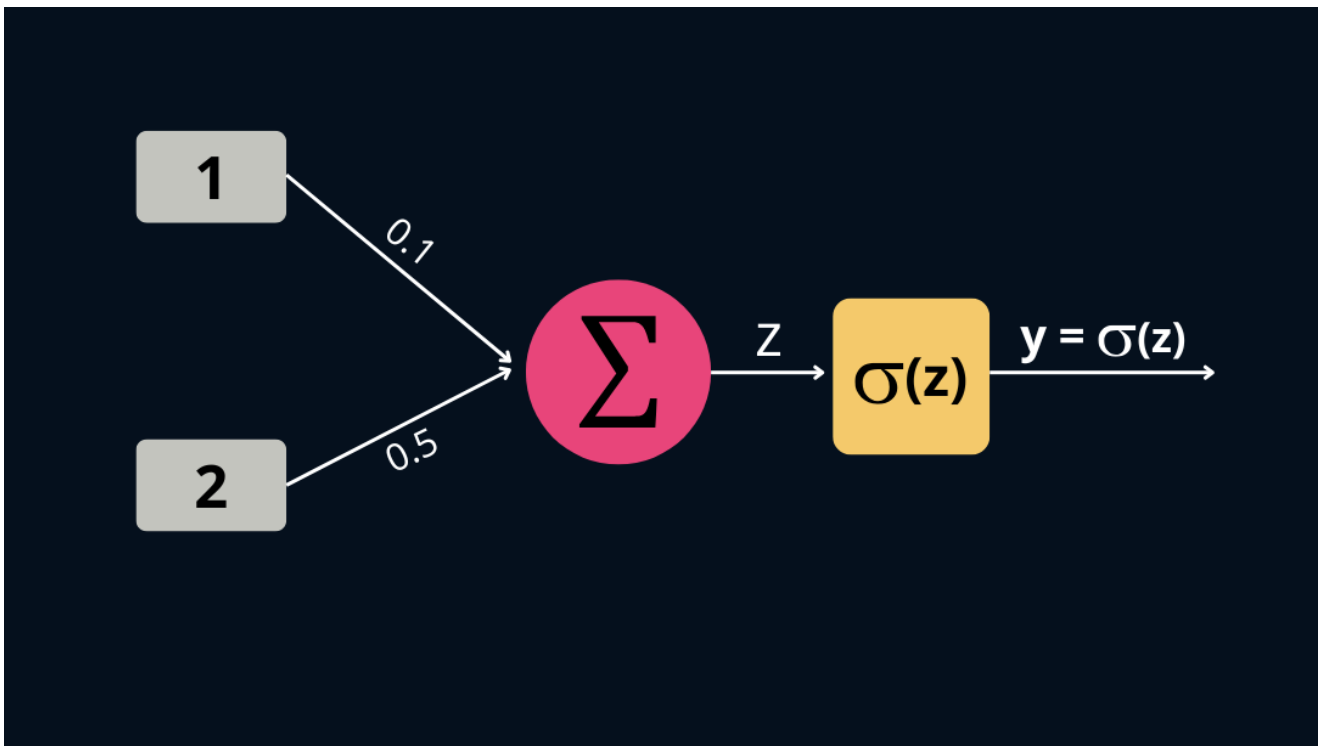
$$e \approx 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \frac{1}{6!} + \frac{1}{7!} \dots$$

O uso do e é fundamental porque a função exponencial tem propriedades matemáticas que tornam a sigmoid suave, continuamente diferenciável e adequada para cálculos de gradiente durante o treinamento de redes neurais. Além disso, a presença do e garante que pequenas variações em x resultem em mudanças suaves na saída, o que é essencial para o aprendizado eficiente em modelos de machine learning

Resultando em um gráfico como:



Quanto maior o valor de entrada, mais próxima de 1 será a saída; quanto menor, mais próxima de 0. No entanto, é importante observar que a função sigmoid apenas **tende** a esses extremos — ela **nunca atinge exatamente** os valores 0 ou 1, apenas se aproxima infinitamente deles.



$$y = \sigma(1 \cdot 0,1 + 2 \cdot 0.5)$$

$$y = \sigma(1.1)$$

$$y = 0.75$$

Representação por álgebra linear

Agora que você já entendeu o funcionamento do perceptron, podemos começar a pensar em como implementá-lo. Para isso, utilizaremos **álgebra linear**, pois em Python as entradas e os pesos são representados por **vetores e matrizes**.

As entradas serão representadas da seguinte forma:

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

E os pesos:

$$\vec{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_n \end{bmatrix}$$

Nesta etapa, já sabemos que precisamos multiplicar as entradas pelos pesos (um **produto escalar**), como vimos no apêndice de Álgebra Linear. Para isso, como estamos lidando com a multiplicação de um **vetor linha pelo vetor coluna**, é necessário realizar a **transposição do vetor de pesos $\vec{w}$**.

$$\vec{w}^\top = \begin{bmatrix} w_1 & w_2 & w_3 & \cdots & w_n \end{bmatrix}$$

Agora sim, podemos realizar o produto escalar com:

$$z = \vec{w}^\top \vec{x}$$

$$\vec{w}^\top = [1{,}5 \quad -0{,}5], \quad \vec{x} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

$$z = \vec{w}^\top \vec{x} = (1{,}5 \times 2) + (-0{,}5 \times 3) = 3 - 1{,}5 = 1{,}5$$

Agora que realizamos a somatória ponderada, basta aplicar a **função de ativação**. Neste exemplo, utilizaremos a **sigmoid**, que transformará o valor escalar em uma saída entre 0 e 1.

$$\vec{w}^\top = [1{,}5 \quad -0{,}5], \quad \vec{x} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

$$z = \vec{w}^\top \vec{x} = (1{,}5 \times 2) + (-0{,}5 \times 3) = 3 - 1{,}5 = 1{,}5$$

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-1{,}5}} \approx \frac{1}{1 + 0{,}2231} \approx \frac{1}{1{,}2231} \approx 0{,}777$$