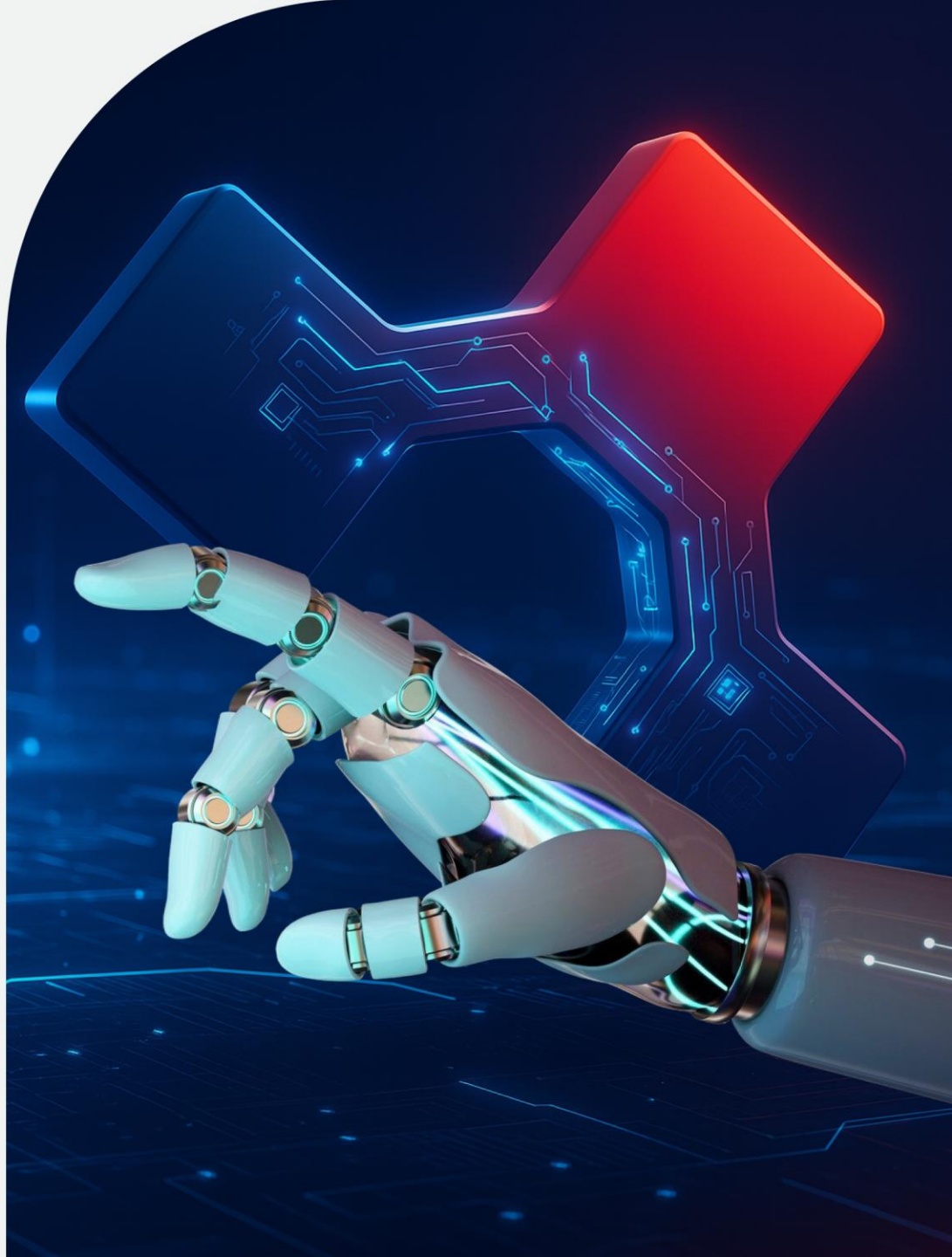
NCIA

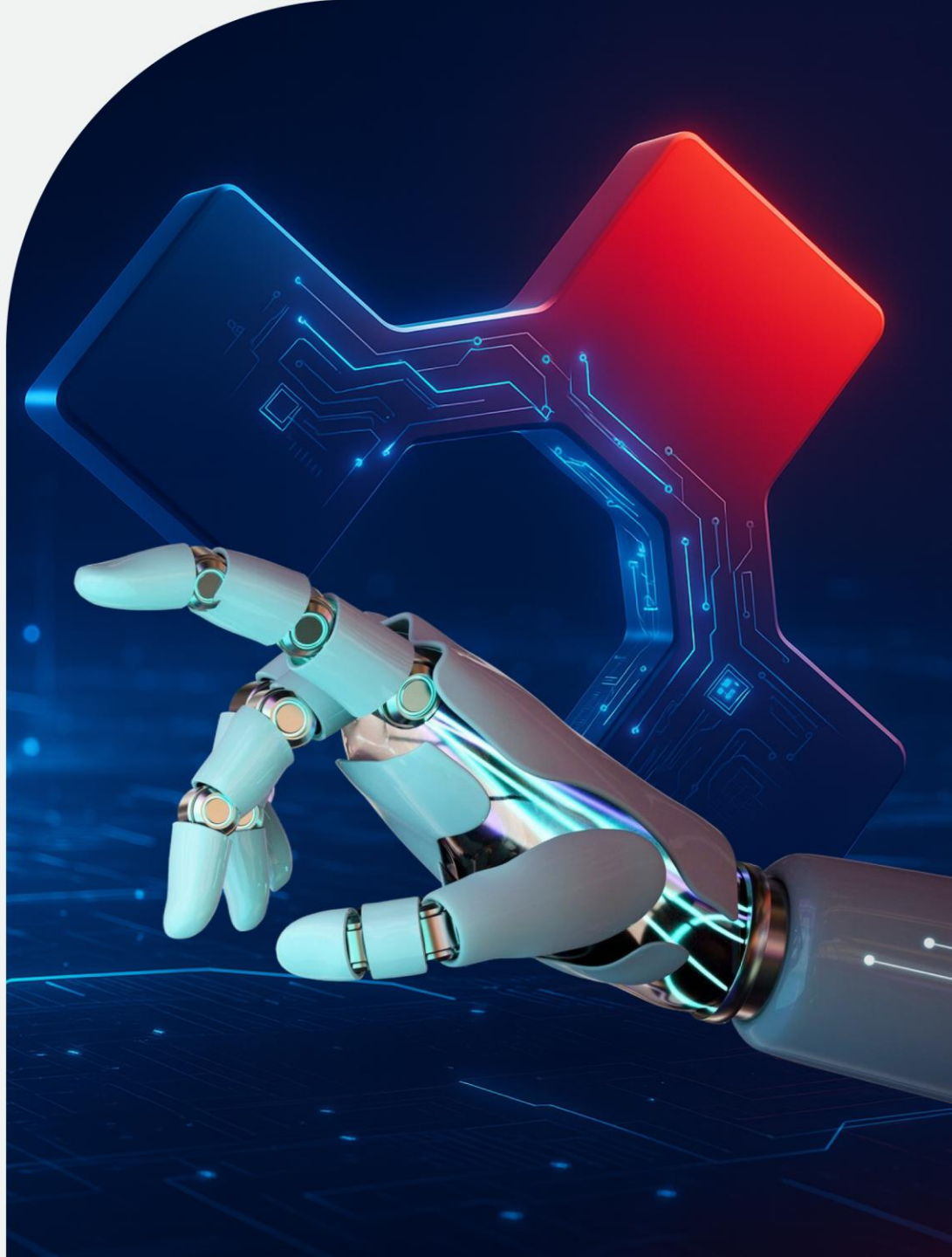
Núcleo de Capacitação em Inteligência Artificial



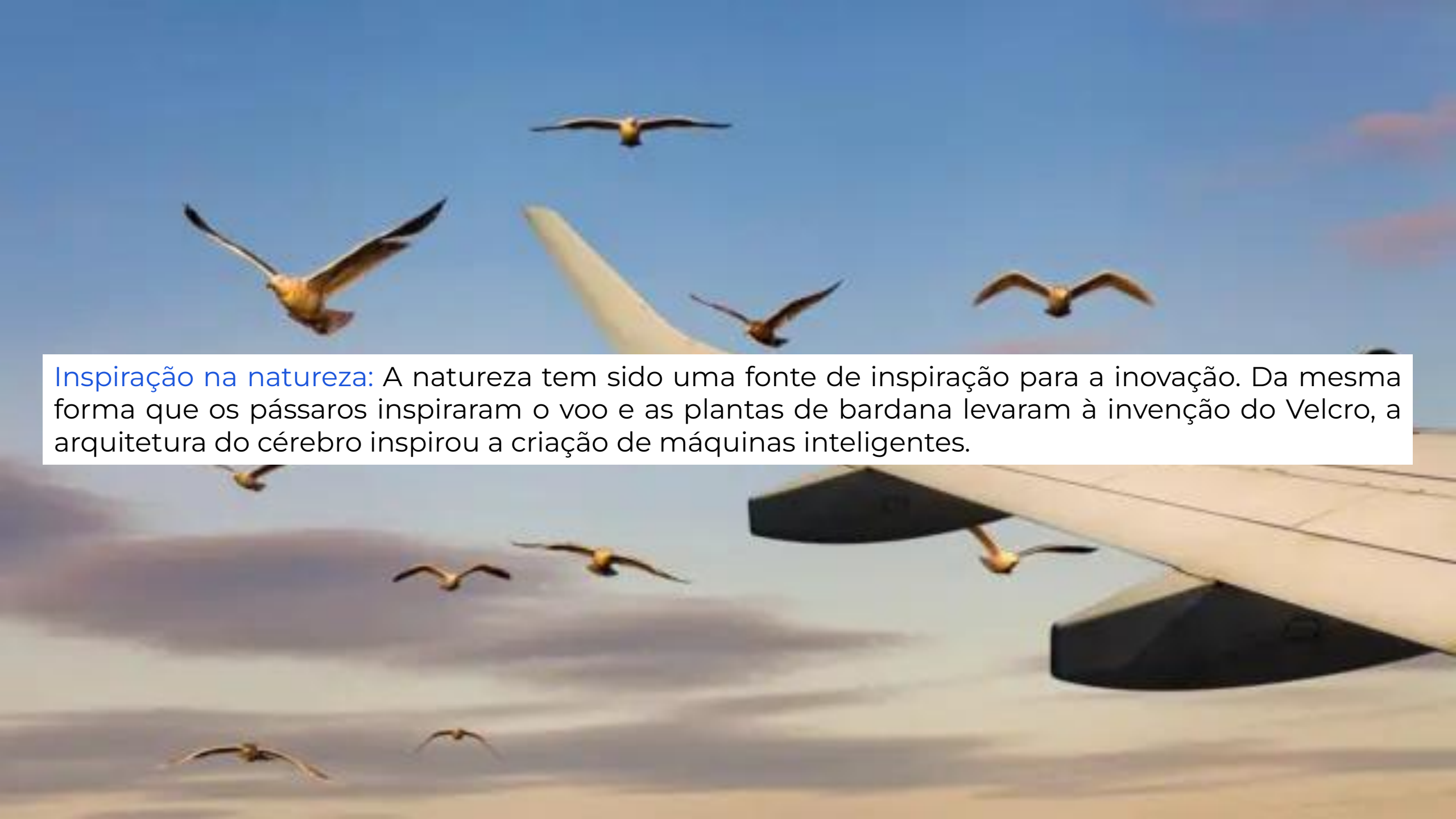
NCIA | FOXCONN FPF tech

Introduction to Artificial Neural Networks

From Biological to Artificial Neurons; Biological Neurons; Logical Computations with Neurons; The Perceptron; The Multilayer Perceptron and Backpropagation; Regression MLPs;

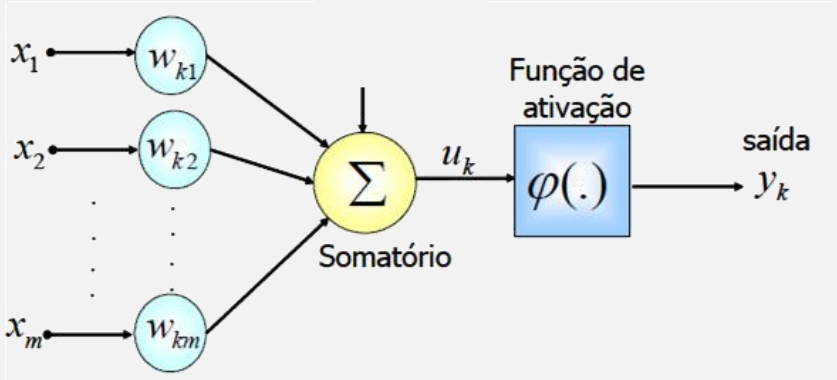


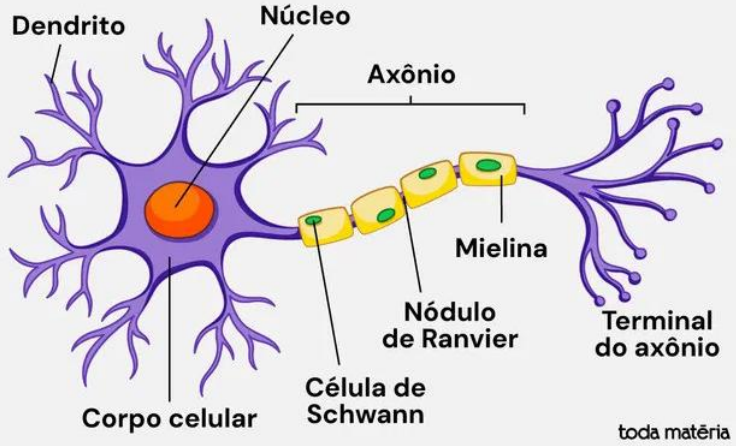
NCIA | FOXCONN FPF tech



Inspiração na natureza: A natureza tem sido uma fonte de inspiração para a inovação. Da mesma forma que os pássaros inspiraram o voo e as plantas de bardana levaram à invenção do Velcro, a arquitetura do cérebro inspirou a criação de máquinas inteligentes.

O surgimento das ANNs: Essa busca por inspiração biológica deu origem às redes neurais artificiais (ANNs), que são modelos de machine learning inspirados pelas redes de neurônios do nosso cérebro.





Importância no Deep Learning: As ANNs são o coração do deep learning. Elas são poderosas e escaláveis, o que as torna ideais para tarefas complexas como classificar imagens, alimentar assistentes de voz (como a Siri), recomendar vídeos e até mesmo vencer campeões em jogos como Go.

Diferença da biologia: Apesar da inspiração inicial, as ANNs evoluíram e se tornaram diferentes de seus "primos" biológicos. Assim como um avião não precisa bater as asas para voar, as ANNs não imitam perfeitamente o cérebro. Alguns pesquisadores defendem até mesmo abandonar a analogia biológica para evitar limitar a criatividade.

1.1 Dos neurônios biológicos aos neurônios artificiais

Surpreendentemente, as ANNs já existem há bastante tempo: foram introduzidas pela primeira vez em 1943 pelo neurofisiologista Warren McCulloch e pelo matemático Walter Pitts. Em seu artigo histórico “Um Cálculo Lógico de Ideias Imanentes à Atividade Nervosa”, McCulloch e Pitts apresentaram um modelo computacional simplificado de como neurônios biológicos podem trabalhar juntos em cérebros de animais para realizar cálculos complexos usando lógica proposicional. Esta foi a primeira arquitetura de rede neural artificial. Desde então, muitas outras arquiteturas foram inventadas.

Os primeiros sucessos das ANNs levaram à crença generalizada de que em breve estaríamos conversando com máquinas verdadeiramente inteligentes. Quando ficou claro, na década de 1960, que essa promessa não se cumpriria (pelo menos por um bom tempo), o financiamento foi direcionado para outros lugares, e as ANNs entraram em um longo inverno.

O Inverno da IA

No início da década de 1980, novas arquiteturas foram inventadas e melhores técnicas de treinamento foram desenvolvidas, reacendendo o interesse pelo conexionismo, o estudo de redes neurais. Mas o progresso foi lento e, na década de 1990, outras técnicas poderosas de aprendizado de máquina foram inventadas, como máquinas de vetores de suporte. Essas técnicas pareciam oferecer melhores resultados e fundamentos teóricos mais sólidos do que as ANNs, então, mais uma vez, o estudo de redes neurais foi suspenso.

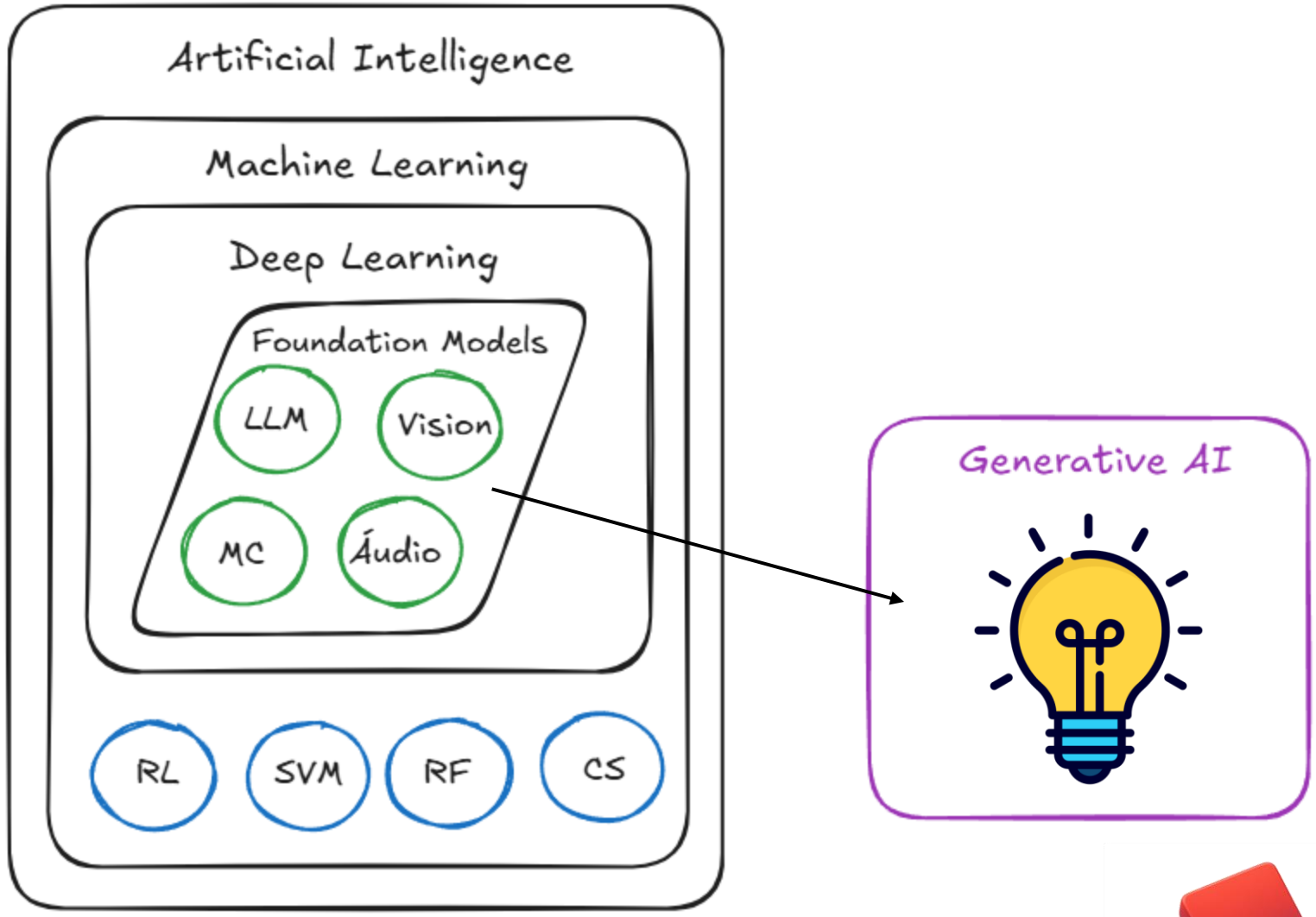
Estamos testemunhando mais uma onda de interesse em ANNs. Será que essa onda vai acabar como as anteriores? Bem, aqui estão alguns bons motivos para acreditar que desta vez é diferente e que o interesse renovado em NNAs terá um impacto muito mais profundo em nossas vidas:

1. Atualmente, há uma enorme quantidade de dados disponíveis para treinar redes neurais, e as ANNs frequentemente superam outras técnicas de ML em problemas muito grandes e complexos.

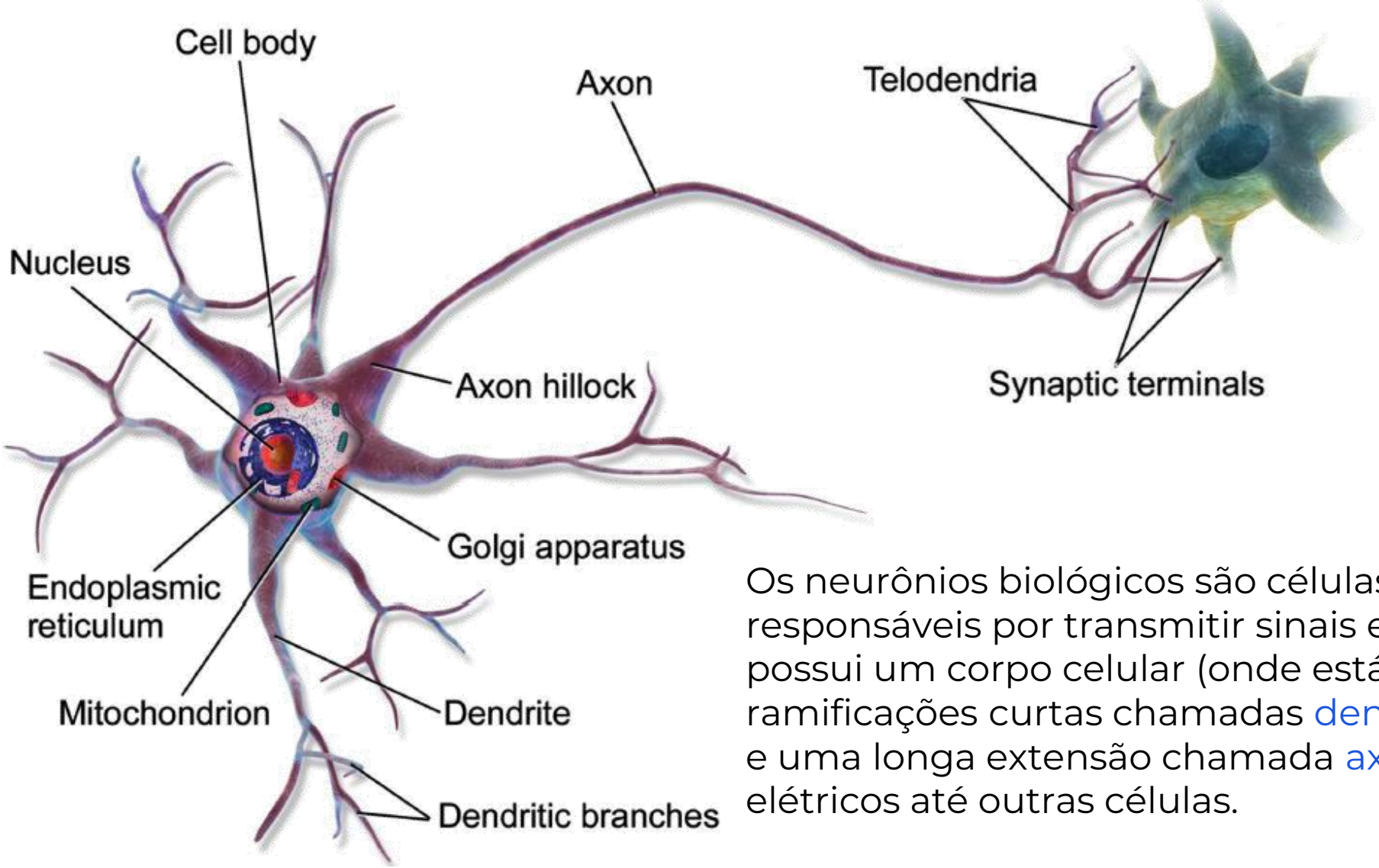
2. O enorme aumento no poder computacional desde a década de 1990 agora torna possível treinar grandes redes neurais em um tempo razoável. Além disso, as plataformas em nuvem tornaram esse poder acessível a todos.

3. Os algoritmos de treinamento foram aprimorados. Para ser justo, eles são apenas ligeiramente diferentes dos usados na década de 1990, mas esses ajustes relativamente pequenos tiveram um enorme impacto positivo.

4. Algumas limitações teóricas das NNAs se mostraram benignas na prática. Por exemplo, muitas pessoas pensavam que os algoritmos de treinamento de NNAs estavam condenados porque provavelmente ficariam presos em ótimos locais, mas acontece que isso não é um grande problema na prática, especialmente para redes neurais maiores: os ótimos locais frequentemente têm um desempenho quase tão bom quanto o ótimo global.


Artificial Intelligence
Machine Learning
Deep Learning
Foundation Models
LLM
Vision
MC
Áudio
RL
SVM
RF
CS
Generative AI

1.1 Neurônio Biológico



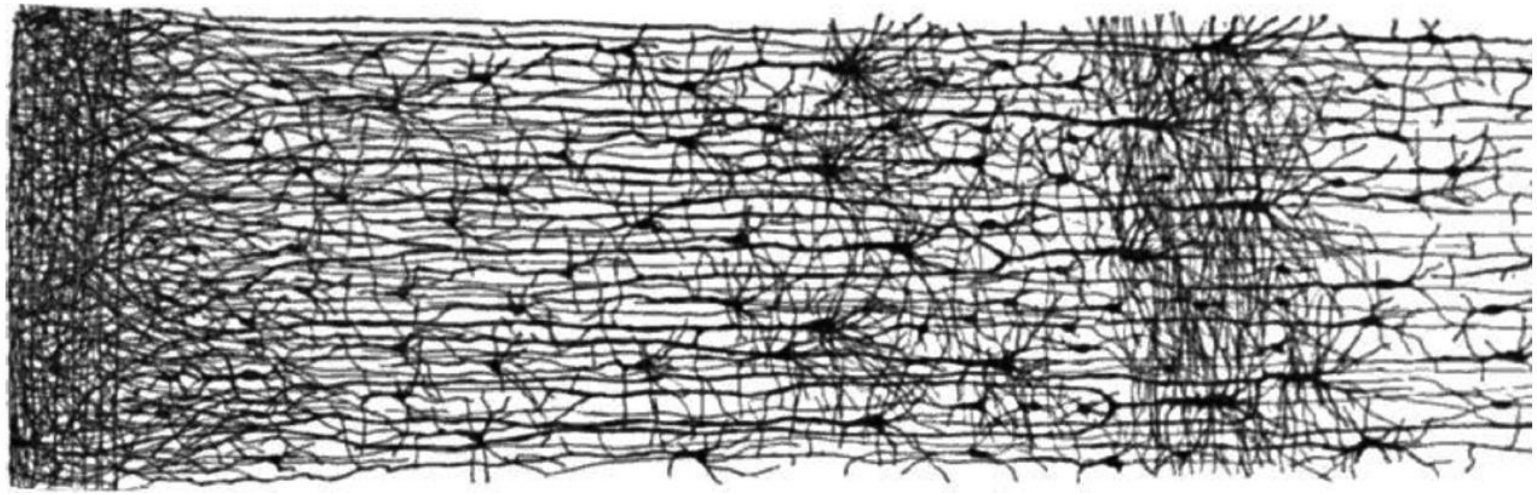
Os neurônios biológicos são células especializadas do cérebro responsáveis por transmitir sinais elétricos. Cada neurônio possui um corpo celular (onde está o núcleo), várias ramificações curtas chamadas dendritos, que recebem sinais, e uma longa extensão chamada axônio, que conduz impulsos elétricos até outras células.

Na extremidade do axônio, há ramificações chamadas telodendros, que terminam em sinapses, pequenas junções responsáveis por liberar substâncias químicas conhecidas como neurotransmissores, permitindo a comunicação entre neurônios.

Quando um neurônio recebe uma quantidade suficiente de neurotransmissores em poucos milissegundos, ele dispara um novo impulso elétrico, que percorre o axônio e ativa as sinapses seguintes. Esse processo contínuo forma a base da comunicação neural no cérebro.
Apesar de cada neurônio agir de forma simples, a interação entre bilhões deles permite a execução de tarefas altamente complexas — desde movimentos motores até o raciocínio abstrato.

Os neurônios estão organizados em extensas redes interconectadas, conhecidas como redes neurais biológicas. Essas redes são compostas por camadas sucessivas de neurônios, especialmente evidentes no córtex cerebral humano, que é a camada externa do cérebro.

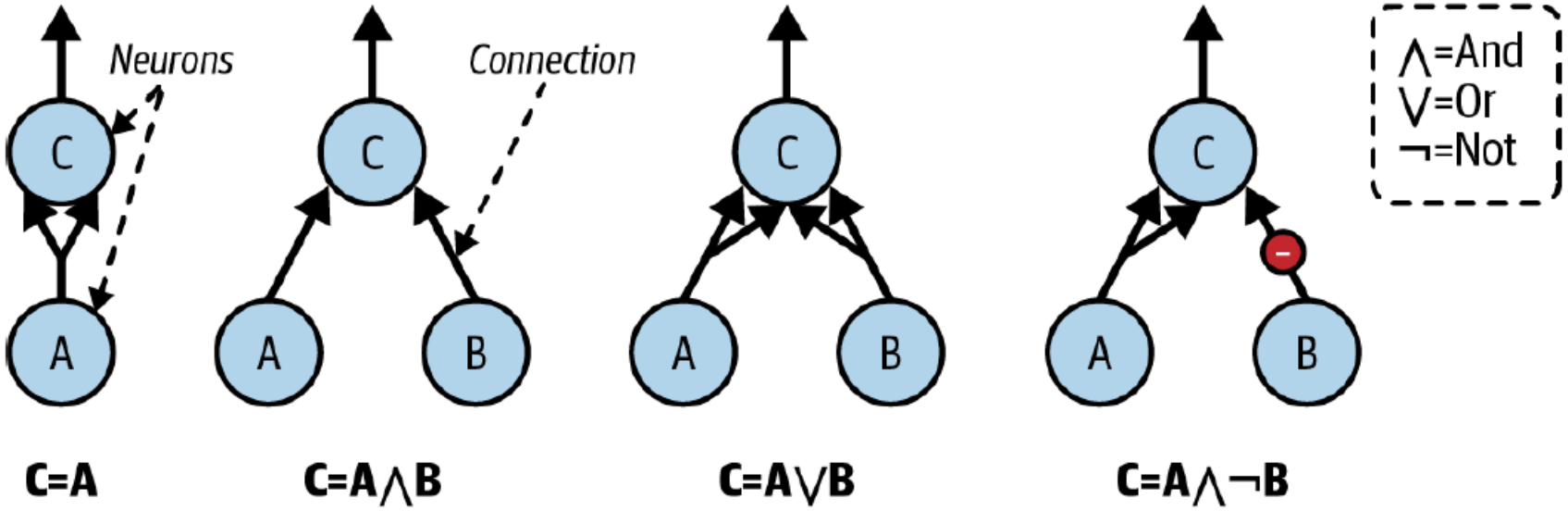
Essa disposição em camadas permite o processamento hierárquico de informações, inspirando a estrutura das redes neurais artificiais usadas em inteligência artificial.



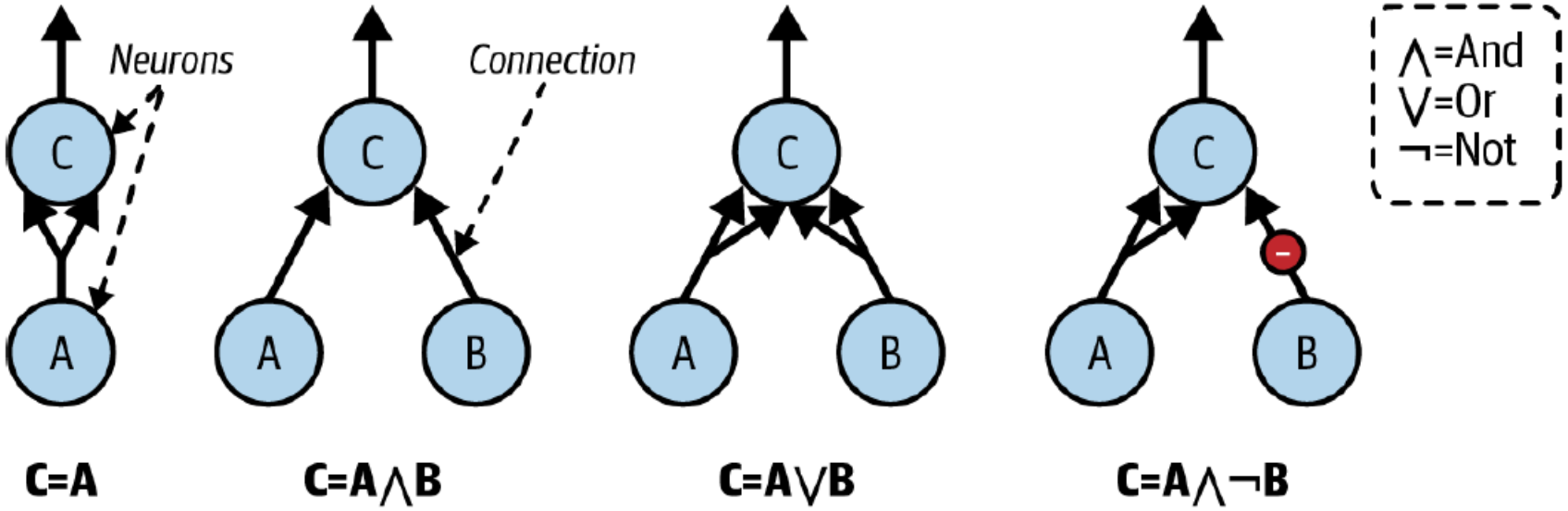
Essas pequenas redes mostram que, combinando unidades simples (neurônios artificiais), é possível construir sistemas capazes de representar qualquer proposição lógica complexa. Esse conceito foi o ponto de partida para o desenvolvimento das redes neurais modernas, nas quais os neurônios artificiais foram generalizados para processar valores contínuos e aprender a partir de dados — abrindo caminho para o aprendizado profundo (deep learning).

1.2 Cálculos lógicos com neurônios

Em 1943, McCulloch e Pitts propuseram um modelo matemático simples inspirado nos neurônios biológicos — o neurônio artificial. Nesse modelo, cada neurônio possui entradas binárias (ligado/desligado) e uma saída também binária.
O neurônio é ativado quando um número mínimo de suas entradas está ativo, simulando o disparo de um neurônio biológico. Mesmo com essa simplicidade, os autores demonstraram que uma rede de tais neurônios pode realizar qualquer computação lógica.



A figura mostra quatro redes neurais artificiais básicas:
Identidade: quando o neurônio A é ativado, o neurônio C também se ativa; se A está desligado, C permanece desligado.
AND (E lógico): C é ativado apenas se A e B estiverem ativos simultaneamente.
OR (OU lógico): C é ativado se A ou B (ou ambos) estiverem ativos.
Combinação com inibição: se uma das conexões puder inibir o neurônio, C será ativado apenas se A estiver ativo e B estiver desligado — criando uma operação equivalente a A AND NOT B.



1.4 o Perceptron

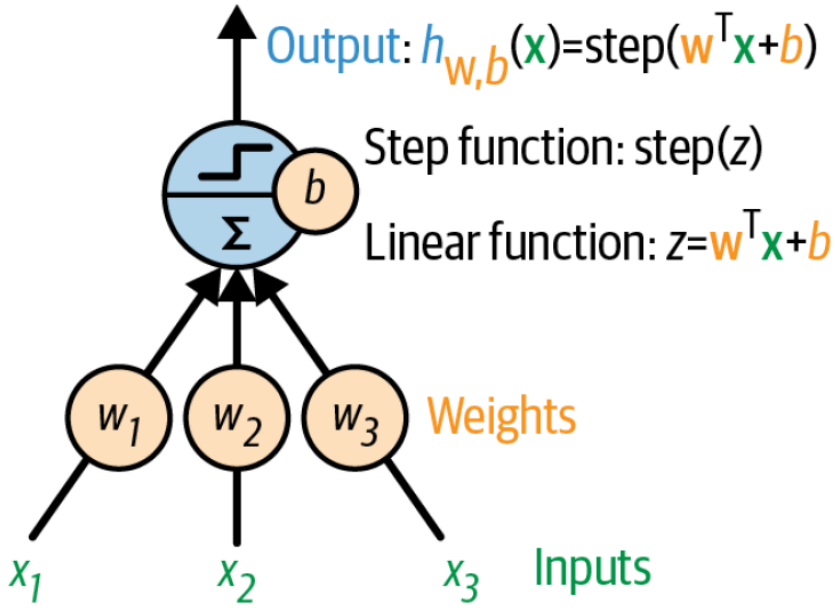
O Perceptron, criado em 1957 por Frank Rosenblatt, foi uma das primeiras arquiteturas de rede neural artificial (RNA). Ele é baseado em uma unidade chamada TLU (Threshold Logic Unit), ou LTU (Linear Threshold Unit) — um neurônio artificial que recebe entradas numéricas, aplica pesos a cada uma e adiciona um viés (bias).
O TLU calcula uma função linear:

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b = \mathbf{w}^\top \mathbf{x} + b$$

Depois, aplica uma função degrau:

$$h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z)$$

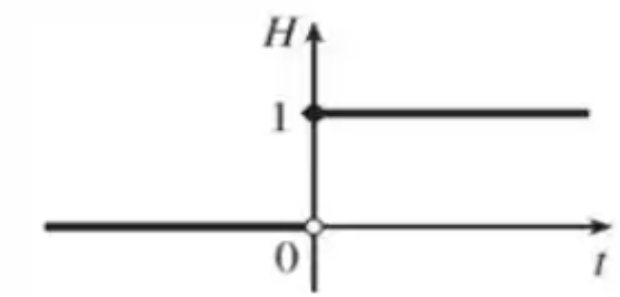
É semelhante à regressão logística, mas usa uma função degrau em vez da função logística.

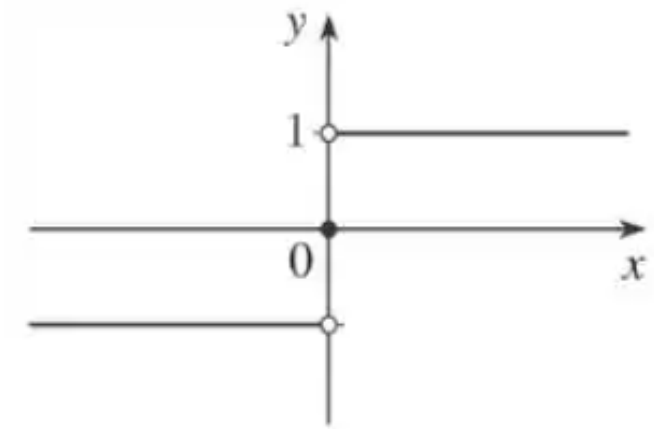


As funções degrau controlam quando o neurônio é ativado. As duas mais usadas são:

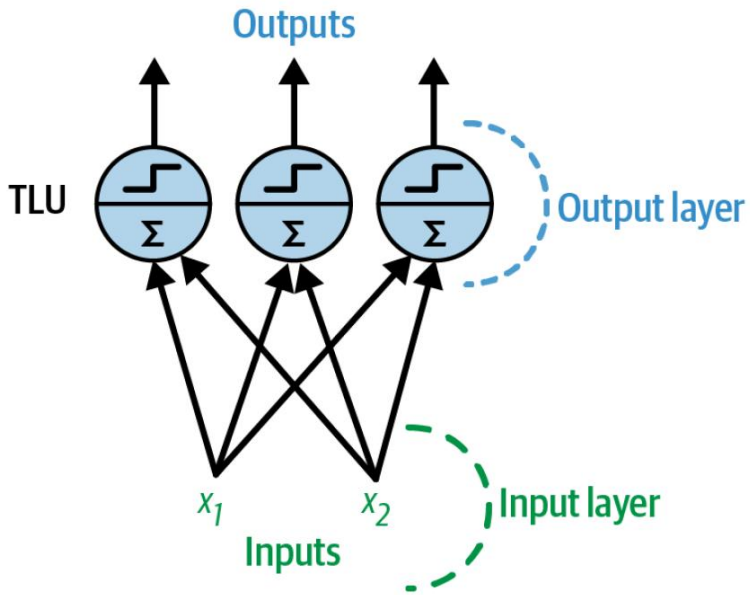
$$\text{heaviside}\,(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \qquad \text{sgn}\,(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

Essas funções permitem ao Perceptron realizar classificação binária linear — ativando uma saída positiva se o valor ultrapassa um limiar e negativa caso contrário.





Um perceptron é formado por uma ou mais TLUs organizadas em uma única camada totalmente conectada (fully connected layer).
Cada TLU recebe todas as entradas, e o conjunto dessas saídas forma a camada de saída.



Esse modelo pode classificar múltiplas categorias ao mesmo tempo (classificação multilabel), e pode também ser adaptado a tarefas multiclasse.

Graças à álgebra linear, é possível calcular todas as saídas de uma camada de uma vez, para várias instâncias:

$$\hat{\mathbf{Y}} = \varphi\left(\mathbf{X}\mathbf{W} + \mathbf{b}\right)$$

Ŷ é a matriz de saída. Ela possui uma linha por instância e uma coluna por neurônio.
X é a matriz de entrada. Ela possui uma linha por instância e uma coluna por atributo de entrada.
A matriz de pesos W contém todos os pesos de conexão. Ela possui uma linha por atributo de entrada e uma coluna por neurônio.
O vetor de polarização b contém todos os termos de polarização: um por neurônio.
A função ϕ é chamada de função de ativação: quando os neurônios artificiais são TLUs, ela é uma função degrau (discutiremos outras funções de ativação em breve).

O algoritmo de treinamento do perceptron foi inspirado na regra de Hebb (1949): *"neurônios que disparam juntos se conectam mais fortemente"*.
Durante o treinamento, para cada amostra, o perceptron compara sua predição com o alvo e ajusta os pesos de forma a reduzir o erro.

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta\left(y_j - \hat{y}_j\right)x_i$$

Onde:
$w_{i,j}$:peso entre a entrada *i* e o neurônio *j*.
x_i :valor da entrada *i*.
$\hat{y}_j$:saída prevista.
y_j :valor alvo.
η: taxa de aprendizado.
Essa regra reforça conexões corretas e enfraquece as erradas, permitindo que a rede aprenda padrões a partir dos dados.

Cada neurônio de saída em um perceptron define uma fronteira de decisão linear. Isso significa que o perceptron só consegue separar classes linearmente separáveis, sendo incapaz de aprender padrões complexos — assim como os classificadores de regressão logística.
Rosenblatt demonstrou que, se os dados forem linearmente separáveis, o algoritmo do perceptron sempre converge para uma solução, resultado conhecido como o Teorema da Convergência do Perceptron.

O Scikit-Learn fornece a classe Perceptron, que pode ser usada de forma semelhante a outros classificadores.
Exemplo com o dataset Iris:

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = (iris.target == 0)  # Iris setosa

per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

X_new = [[2, 0.5], [3, 1]]
y_pred = per_clf.predict(X_new)  # retorna True e False
```

Esse perceptron realiza uma classificação binária entre *Iris setosa* e as demais espécies.

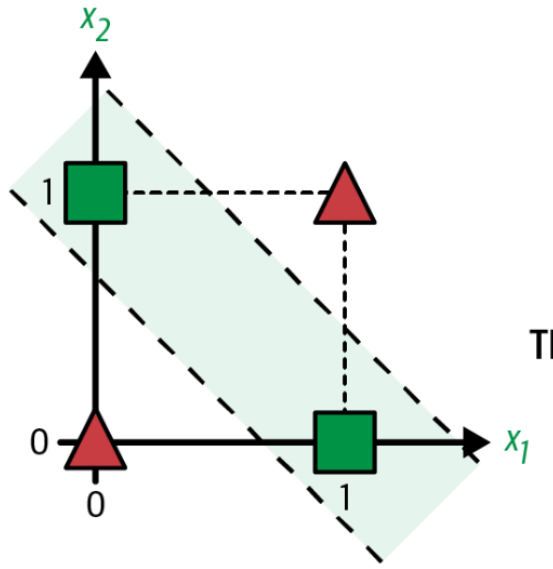
O algoritmo de aprendizado do perceptron é muito semelhante ao Stochastic Gradient Descent (SGD), visto no capítulo 4.
Na prática, o Perceptron do Scikit-Learn é equivalente a usar:

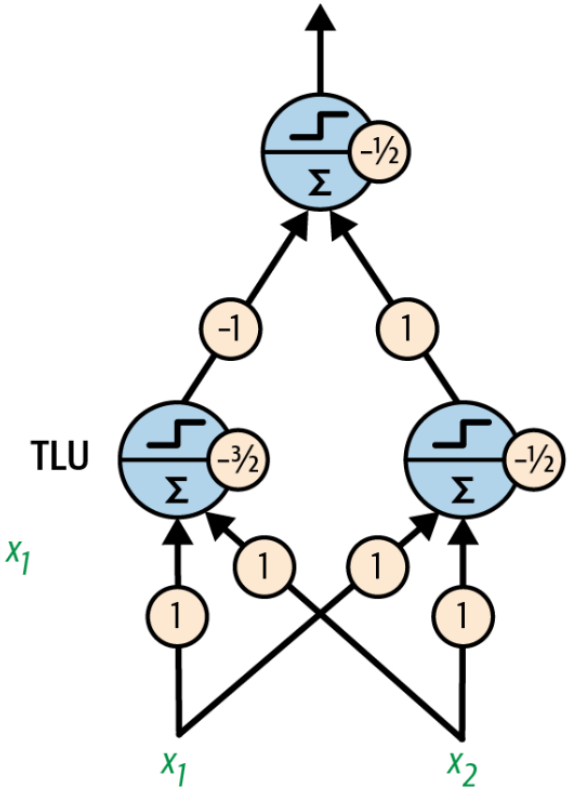
```
SGDClassifier(
    loss="perceptron",
    learning_rate="constant",
    eta0=1,
    penalty=None
)
```

Ou seja, ele é uma forma específica de SGD sem regularização e com taxa de aprendizado constante.

Em 1969, Marvin Minsky e Seymour Papert publicaram o livro *Perceptrons*, destacando limitações graves do modelo.
Eles mostraram que o perceptron não consegue resolver problemas simples como o XOR, pois a saída desejada não pode ser separada por uma linha reta.

Essa descoberta causou grande frustração na comunidade científica — muitos pesquisadores abandonaram o estudo das redes neurais, migrando para abordagens mais formais, como lógica simbólica e busca.

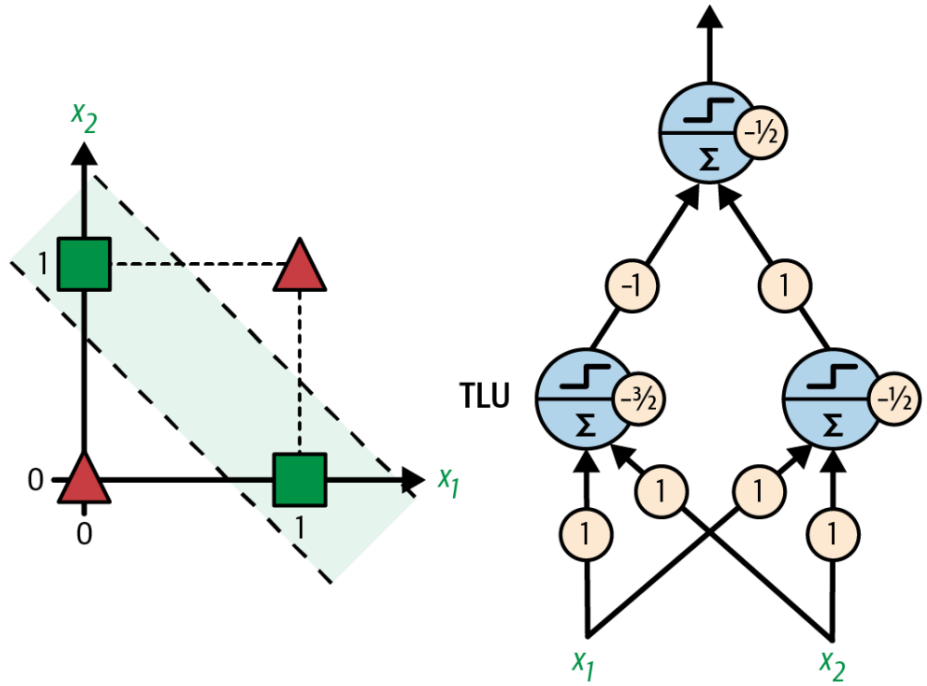




Alguns anos depois, descobriu-se que era possível superar as limitações do perceptron empilhando várias camadas de neurônios. Essa nova estrutura passou a ser chamada de Perceptron Multicamadas (MLP – Multilayer Perceptron).
O MLP introduz camadas ocultas (hidden layers) e funções de ativação não lineares, permitindo que a rede aprenda padrões complexos e não linearmente separáveis — tornando-se a base das redes neurais modernas.

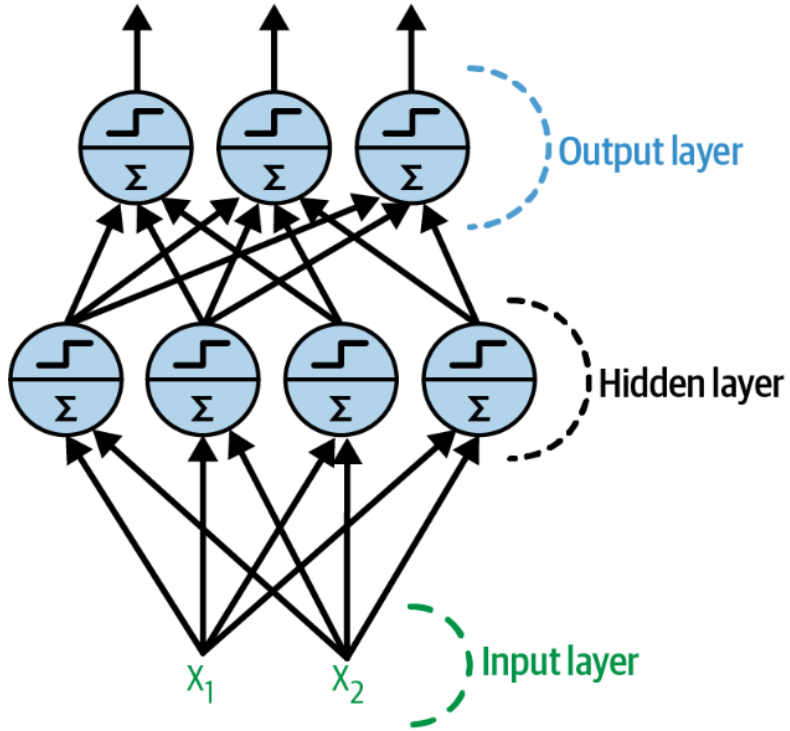
1.5 The Multilayer Perceptron and Backpropagation

O Perceptron Multicamadas (MLP) é capaz de resolver problemas que o perceptron simples não consegue — como o clássico problema do XOR.
Na figura ao lado, vemos que o MLP à direita produz a saída 0 para as entradas (0, 0) e (1, 1), e a saída **1** para (0, 1) e (1, 0). Isso mostra como uma rede com camadas intermediárias (ocultas) pode aprender relações não lineares.



Um MLP é composto por:

Camada de entrada, que recebe os dados.
Uma ou mais camadas ocultas, compostas por neurônios artificiais (originalmente TLUs).
Camada de saída, que gera a predição final. As camadas próximas à entrada são chamadas de camadas inferiores, e as próximas à saída, camadas superiores.
O sinal flui em uma única direção (entrada → saída), caracterizando uma rede neural feedforward (FNN).

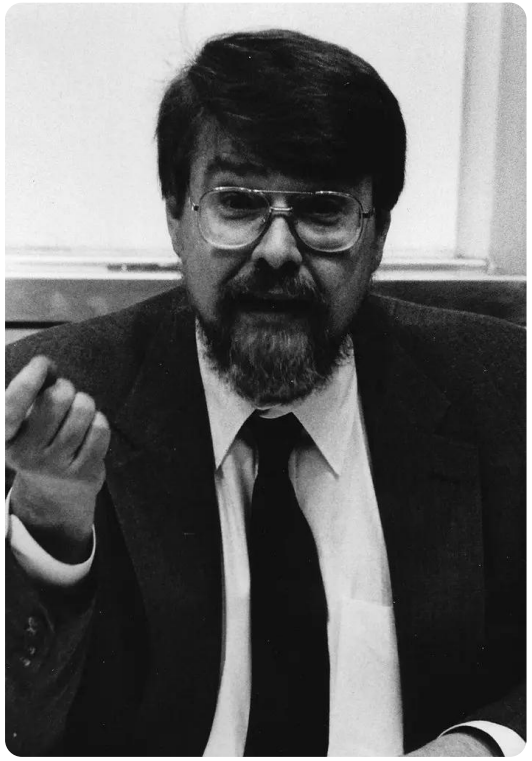


Quando um MLP possui muitas camadas ocultas, chamamos de rede neural profunda (DNN), base do campo de deep learning.
Durante anos, pesquisadores não sabiam como treinar eficientemente essas redes, pois o cálculo dos gradientes (necessários para o aprendizado) era inviável nos computadores da época.

Em 1970, Seppo Linnainmaa propôs um método chamado reverse-mode automatic differentiation (autodiff reverso), capaz de calcular todos os gradientes de forma eficiente em duas passagens pela rede:
Forward pass: calcula as saídas e armazena os resultados intermediários.
Backward pass: aplica a regra da cadeia para calcular a contribuição do erro de cada parâmetro.
Repetindo esse processo com gradiente descendente, a rede ajusta seus pesos até minimizar o erro — esse ciclo é o que chamamos de backpropagation.

Imagine que você está aprendendo a acertar uma bola na cesta:
O arremesso é o *forward pass* (você faz uma predição).
O erro é ver onde a bola caiu (muito à direita, por exemplo).
O ajuste é o *backward pass* — você analisa como mudar sua postura para errar menos.
O novo arremesso é o passo de gradiente descendente.
Repetindo o ciclo, seus erros diminuem até você acertar consistentemente — o mesmo acontece com o MLP durante o treinamento.

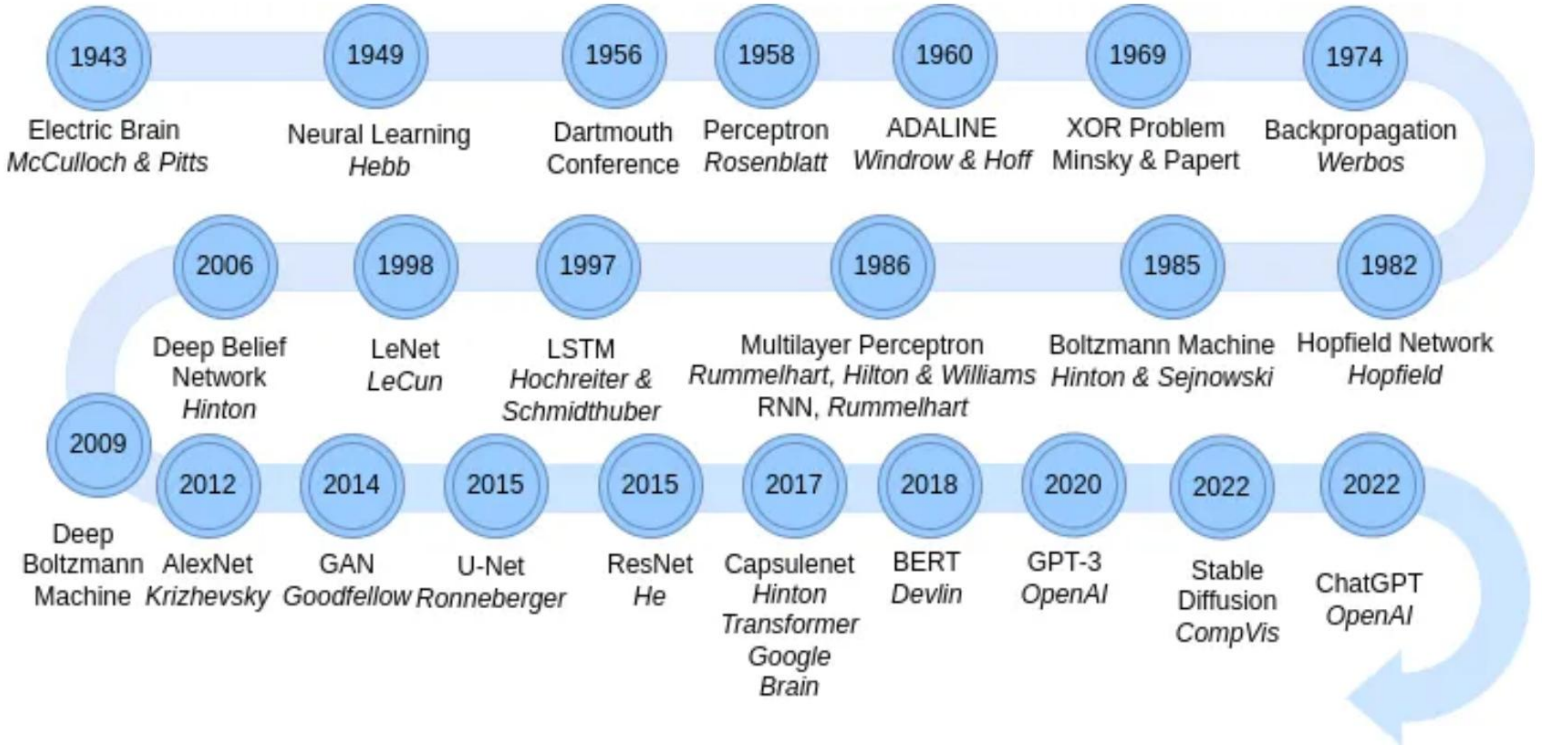
O backpropagation é aplicável a qualquer grafo computacional, não apenas a redes neurais. Mas foi em 1985, com o trabalho de Rumelhart, Hinton e Williams, que o método ganhou destaque. Eles mostraram que o algoritmo permite que redes aprendam representações internas úteis, revolucionando a IA. Mais de 40 anos depois, o backprop ainda é a principal técnica de treinamento em redes neurais.



Funcionamento passo a passo do Backpropagation:

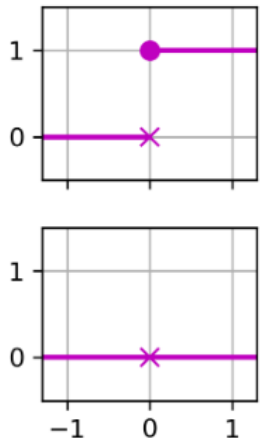
1.Divide o conjunto de dados em mini-batches (por exemplo, 32 amostras × 100 features).
2.Realiza a passagem direta (forward pass) calculando as saídas camada a camada: $\hat{\mathbf{Y}} = \varphi(\mathbf{X}\mathbf{W} + \mathbf{b})$
3.Mede o erro da rede usando uma função de perda.
4.Calcula a contribuição do erro de cada parâmetro, aplicando a regra da cadeia.
5.Propaga o erro de trás para frente até a entrada (reverse pass).
6.Atualiza pesos e vieses via gradiente descendente.

⚠️ Atenção: é fundamental inicializar os pesos de forma aleatória — se todos começarem iguais, os neurônios se comportarão da mesma forma, e o aprendizado não ocorrerá.


1943
Electric Brain
McCulloch & Pitts
1949
Neural Learning
Hebb
1956
Dartmouth
Conference
1958
Perceptron
Rosenblatt
1960
ADALINE
Windrow & Hoff
1969
XOR Problem
Minsky & Papert
1974
Backpropagation
Werbos
1982
Hopfield Network
Hopfield
1985
Boltzmann Machine
Hinton & Sejnowski
1986
Multilayer Perceptron
Rummelhart, Hilton & Williams
RNN, Rummelhart
1997
LSTM
Hochreiter &
Schmidthuber
1998
LeNet
LeCun
2006
Deep Belief
Network
Hinton
2009
Deep
Boltzmann
Machine
2012
AlexNet
Krizhevsky
2014
GAN
Goodfellow
2015
U-Net
Ronneberger
2015
ResNet
He
2017
Capsulenet
Hinton
Transformer
Google
Brain
2018
BERT
Devlin
2020
GPT-3
OpenAI
2022
Stable
Diffusion
CompVis
2022
ChatGPT
OpenAI

Funções de ativação

Heaviside (Step Function)

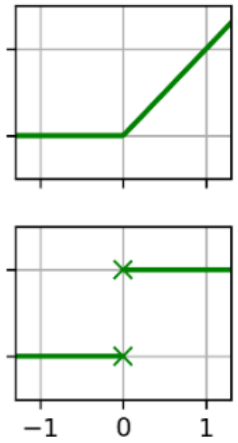


$$\text{heaviside}(z) = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases}$$

- Ativa o neurônio de forma binária
- Representa o tudo ou nada
- Sem gradiente → não permite aprendizado com backpropagation

A função Heaviside, ou função degrau, foi a primeira usada nos neurônios artificiais — lá nos perceptrons de Rosenblatt.
Ela representa um comportamento binário: o neurônio está ligado (1) quando o estímulo passa do limiar, e desligado (0) caso contrário.
Isso imitava bem o disparo dos neurônios biológicos, mas trazia um grande problema para o aprendizado de máquina:
a função não tem gradiente — ela é completamente plana antes e depois do limiar.
Sem gradiente, o modelo não consegue ajustar os pesos durante o treinamento, tornando impossível usar backpropagation.
Por isso, ela é historicamente importante, mas praticamente obsoleta nos modelos modernos.

Sigmoid (Logística)

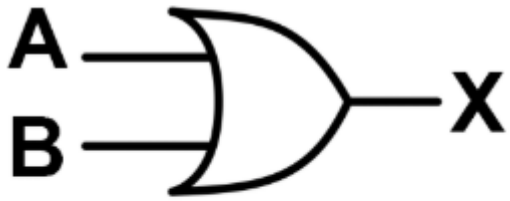


$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- Saída entre 0 e 1
- Interpretação como probabilidade
- Suaviza o degrau → permite gradiente

A sigmoid foi o primeiro grande avanço após a Heaviside. Ela “suaviza” o degrau, tornando a transição contínua e diferenciável.
Agora, o neurônio não é simplesmente ligado ou desligado — ele responde gradualmente ao estímulo.
Essa suavização permite usar gradiente descendente para ajustar os pesos.
Além disso, a saída entre 0 e 1 é intuitiva: podemos interpretá-la como probabilidade.
O problema é que, para valores extremos de entrada, o gradiente fica muito pequeno, gerando o problema do vanishing gradient — o que dificulta o aprendizado em redes profundas.
Mesmo assim, a sigmoid foi essencial para o desenvolvimento do backpropagation.

Símbolo

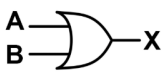


$X = A+B$

Tabela Verdade

Porta OR		
A	**B**	**Saída**
0	0	0
0	1	1
1	0	1
1	1	1

Símbolo



X = A+B

Tabela Verdade

Porta OR		
A	B	Saída
0	0	0
0	1	1
1	0	1
1	1	1

Em cada época:
1. Cálculo (forward pass):
Ele calcula:

$$z = w_1 x_1 + w_2 x_2 + b \qquad \hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}$$

2. Compara com a saída real (y) e calcula o erro:

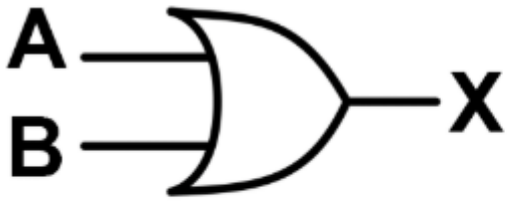
$$E = (y - \hat{y})$$

3. Ajusta os pesos e o bias (backpropagation):

$$w_i \leftarrow w_i + \eta \cdot E \cdot x_i$$
$$b \leftarrow b + \eta \cdot E$$

onde η é a taxa de aprendizado (learning rate).

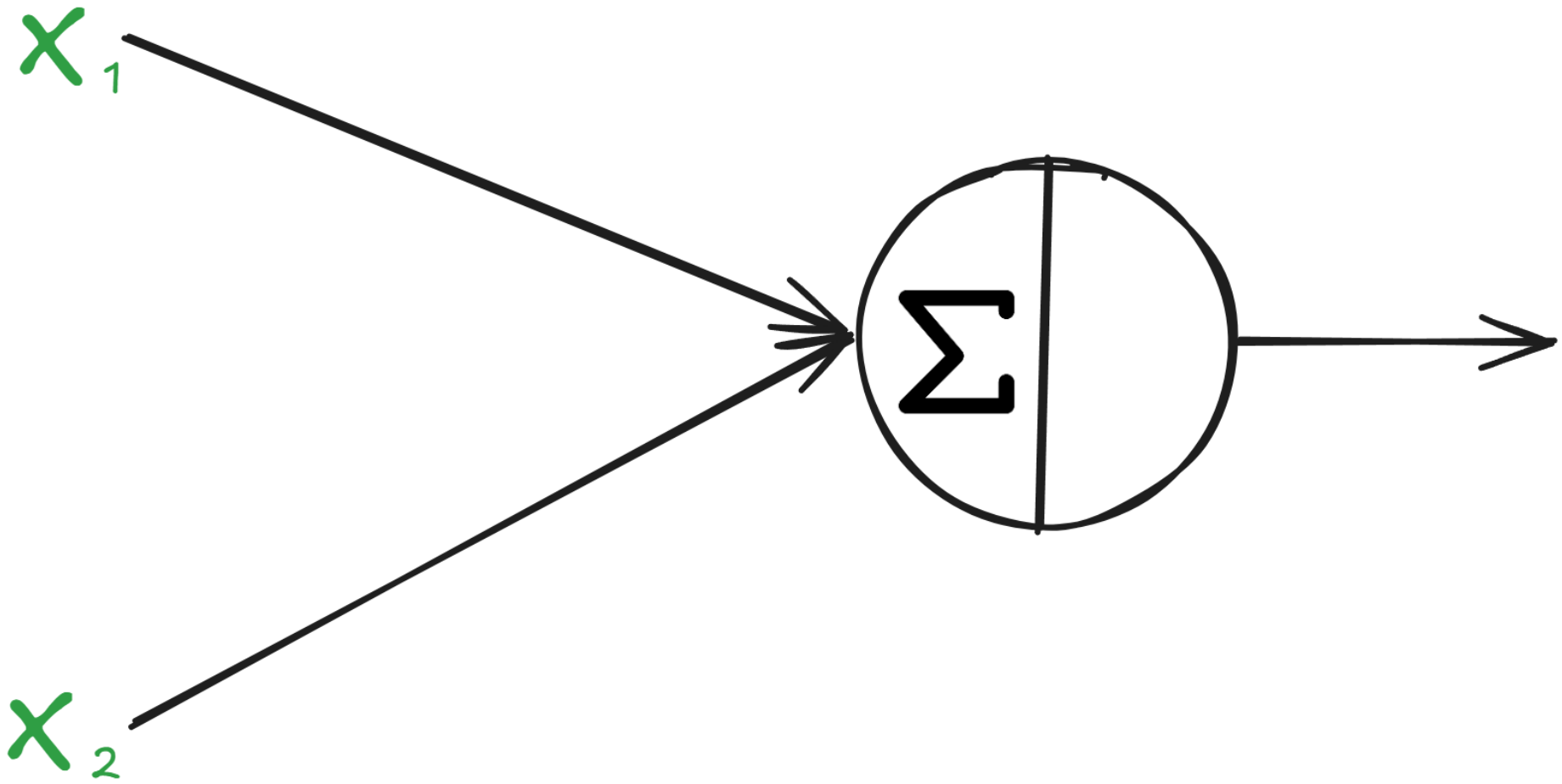
Símbolo

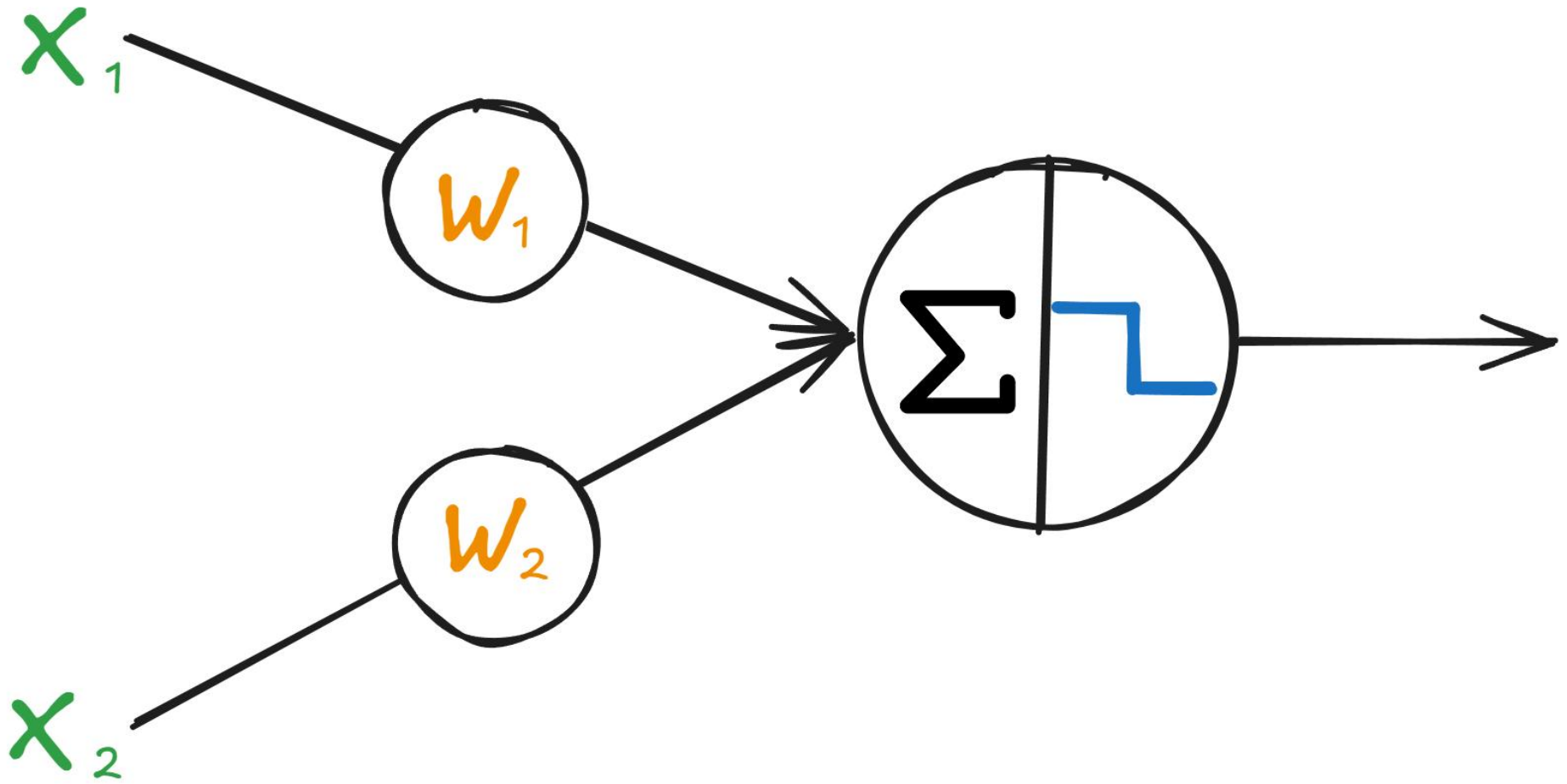


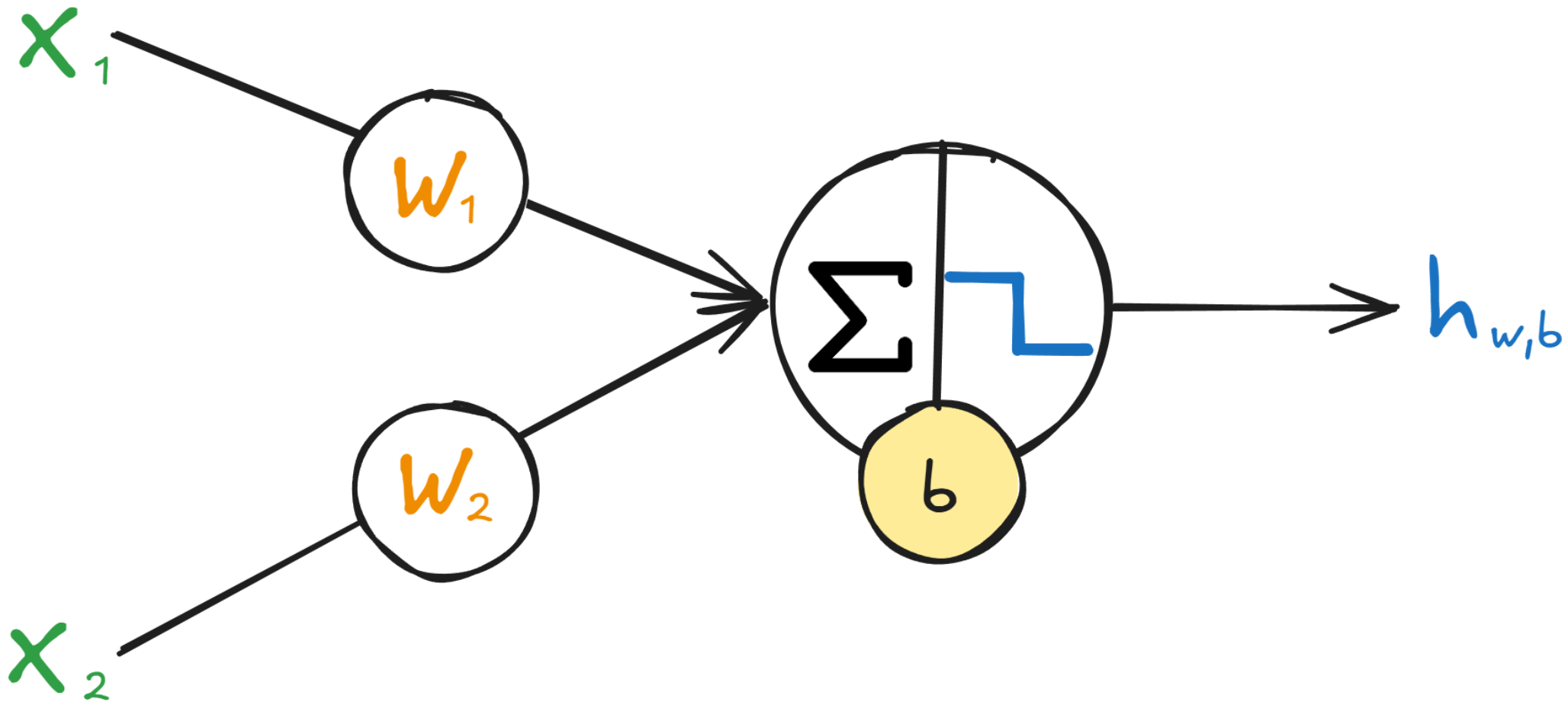
$X = A+B$

Tabela Verdade

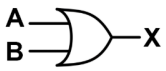
Porta OR		
A	**B**	**Saída**
0	0	0
0	1	1
1	0	1
1	1	1


X 1
X 2
Σ


X 1
W 1
W 2
X 2


x 1
w 1
w 2
x 2
b
h w,b

Símbolo



X = A+B

Tabela Verdade

Porta OR		
A	B	Saída
0	0	0
0	1	1
1	0	1
1	1	1

Pesos e bias começam aleatórios:

$$w_1 = 0.2, w_2 = -0.1, b = 0.0$$

Taxa de aprendizado:

$$\eta = 0.1$$

Função de ativação (sigmoide):

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

Regra de atualização:

$$w_i \leftarrow w_i + \eta \cdot (y - \hat{y}) \cdot x_i$$
$$b \leftarrow b + \eta \cdot (y - \hat{y})$$

Época 1 — Primeira amostra (x_1=1, x_2=0, y=1)
Cálculo da saída:

$$z = (0.2)(1) + (-0.1)(0) + 0 = 0.2$$

$$\hat{y} = \sigma(0.2) = \frac{1}{1 + e^{-0.2}} = 0.55$$

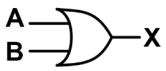
Erro:

$$E = y - \hat{y} = 1 - 0.55 = 0.45$$

Atualização dos pesos e bias:

$$w_1 = 0.2 + 0.1 \times 0.45 \times 1 = 0.245$$
$$w_2 = -0.1 + 0.1 \times 0.45 \times 0 = -0.1$$
$$b = 0 + 0.1 \times 0.45 = 0.045$$

Símbolo



X = A+B

Tabela Verdade

Porta OR		
A	B	Saída
0	0	0
0	1	1
1	0	1
1	1	1

Novos valores:

$$w_1 = 0.245, w_2 = -0.1, b = 0.045$$

Época 1 — Segunda amostra (x_1=0, x_2=1, y=1)
Cálculo da saída:

$$z = (0.245)(0) + (-0.1)(1) + 0.045 = -0.055$$

$$\hat{y} = \sigma(-0.055) = \frac{1}{1 + e^{0.055}} = 0.486$$

Erro:

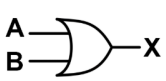
$$E = y - \hat{y} = 1 - 0.486 = 0.514$$

Atualização dos pesos e bias:

$$w_1 = 0.245 + 0.1 \times 0.514 \times 0 = 0.245$$
$$w_2 = -0.1 + 0.1 \times 0.514 \times 1 = -0.0486$$
$$b = 0.045 + 0.1 \times 0.514 = 0.0964$$

$$z = w_1 x_1 + w_2 x_2 + b$$
$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}$$
$$E = (y - \hat{y})$$
$$w_i \leftarrow w_i + \eta \cdot E \cdot x_i$$
$$b \leftarrow b + \eta \cdot E$$

Símbolo



X = A+B

Tabela Verdade

Porta OR		
A	B	Saída
0	0	0
0	1	1
1	0	1
1	1	1

Novos valores:

$$w_1 = 0.245, w_2 = -0.0486, b = 0.0964$$

Época 1 — Terceira amostra (x_1=1, x_2=1, y=1)
Cálculo:

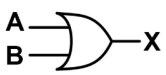
$$z = (0.245)(1) + (-0.0486)(1) + 0.0964 = 0.293$$
$$\hat{y} = \sigma(0.293) = 0.572$$
$$E = 1 - 0.572 = 0.428$$

$$z = w_1 x_1 + w_2 x_2 + b$$
$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}$$
$$E = (y - \hat{y})$$
$$w_i \leftarrow w_i + \eta \cdot E \cdot x_i$$
$$b \leftarrow b + \eta \cdot E$$

Atualização:

$$w_1 = 0.245 + 0.1 \times 0.428 \times 1 = 0.2878$$
$$w_2 = -0.0486 + 0.1 \times 0.428 \times 1 = -0.0058$$
$$b = 0.0964 + 0.1 \times 0.428 = 0.1392$$

Símbolo



X = A+B

Tabela Verdade

Porta OR		
A	B	Saída
0	0	0
0	1	1
1	0	1
1	1	1

Novos valores:

w1=0.2878,w2=−0.0058,b=0.1392

Época 1 — Quarta amostra (x_1=0, x_2=0, y=0)

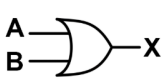
$$z = (0.2878)(0) + (-0.0058)(0) + 0.1392 = 0.1392$$
$$\hat{y} = \sigma(0.1392) = 0.5348$$
$$E = y - \hat{y} = 0 - 0.5348 = -0.5348$$

Atualização:

$$w_1 = 0.2878 + 0.1 \times (-0.5348) \times 0 = 0.2878$$
$$w_2 = -0.0058 + 0.1 \times (-0.5348) \times 0 = -0.0058$$
$$b = 0.1392 + 0.1 \times (-0.5348) = 0.0857$$

$$z = w_1 x_1 + w_2 x_2 + b$$
$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}$$
$$E = (y - \hat{y})$$
$$w_i \leftarrow w_i + \eta \cdot E \cdot x_i$$
$$b \leftarrow b + \eta \cdot E$$

Símbolo



X = A+B

Tabela Verdade

Porta OR		
A	B	Saída
0	0	0
0	1	1
1	0	1
1	1	1

Novos valores:

w1=0.2878, w2=−0.0058, b=0.0857

Agora os pesos atualizados são usados novamente com as mesmas 4 amostras:
A cada iteração, o perceptron recalcula a saída, mede o erro e ajusta os pesos.

$$w_1, w_2, b \text{ são atualizados novamente:}$$
$$w_1 = 0.33, w_2 = 0.04, b = 0.13 \Rightarrow \cdots$$
$$w_1 = 0.36, w_2 = 0.09, b = 0.17 \Rightarrow \cdots$$

...o processo continua por várias épocas...

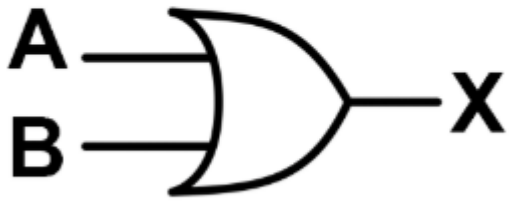
até que o erro médio fique mínimo e os pesos se estabilizem.

Após diversas atualizações, os pesos convergem para valores próximos de:

$$w_1 \approx 6, w_2 \approx 6, b \approx -3$$

O neurônio então passa a reproduzir exatamente a função OR, com saídas próximas de:

Símbolo

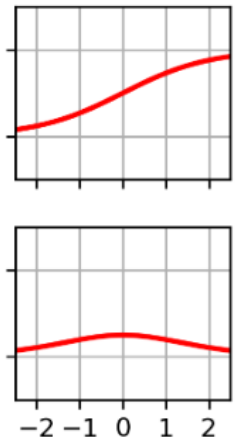


X = A+B

Tabela Verdade

Porta OR		
A	**B**	**Saída**
0	0	0
0	1	1
1	0	1
1	1	1

Tangente Hiperbólica (tanh)



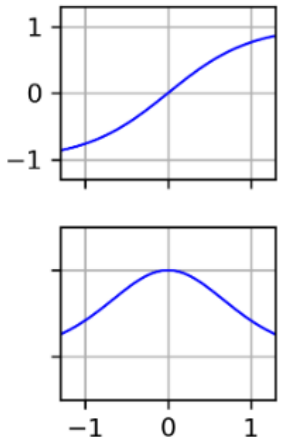
$$\tanh(z) = 2\sigma(2z) - 1$$

- Saída entre –1 e 1
- Centrada em zero → facilita o aprendizado
- Ainda sofre com gradientes pequenos

A tanh é uma versão ajustada da sigmoid:

enquanto a sigmoid varia de 0 a 1, a tanh varia de –1 a 1.
Isso faz com que as ativações fiquem centralizadas em torno de zero, o que ajuda o gradiente a se propagar melhor pelas camadas.
No início do treinamento, ter saídas positivas e negativas balanceadas acelera a convergência e evita acúmulo de vieses positivos.
Por isso, a tanh foi, por muito tempo, a função de ativação mais usada em MLPs antes da chegada da ReLU.
Ainda assim, para valores muito grandes ou muito pequenos de z, ela também sofre do vanishing gradient, o que limita sua eficiência em redes muito profundas.

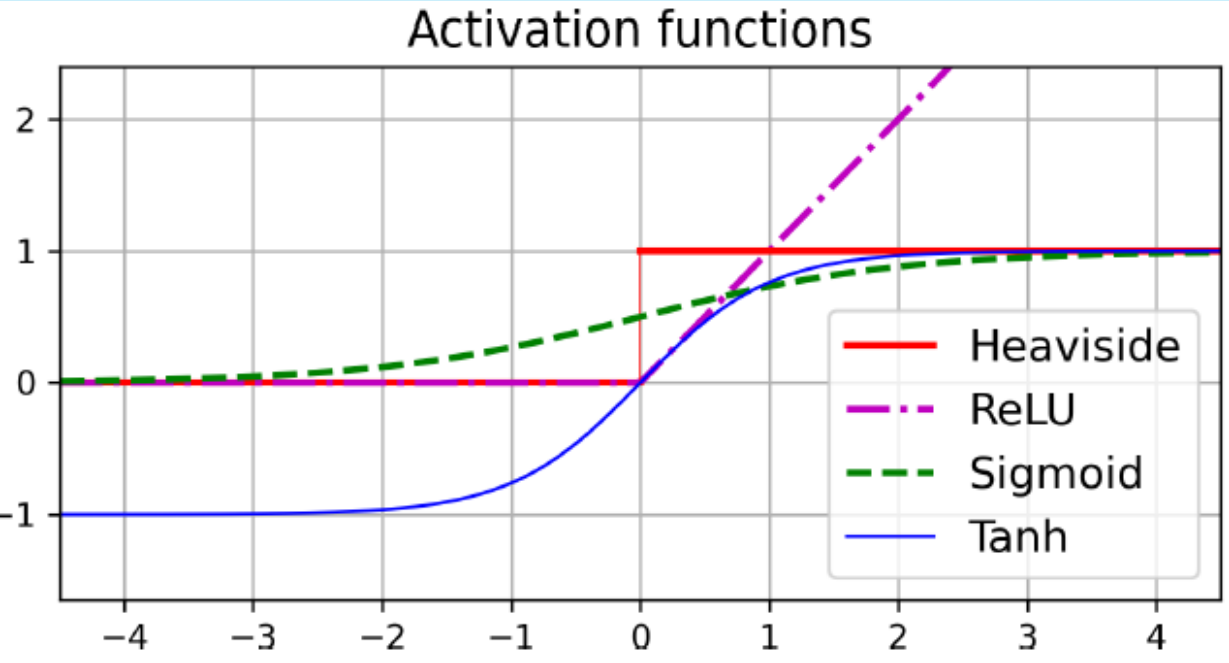
ReLU (Rectified Linear Unit)

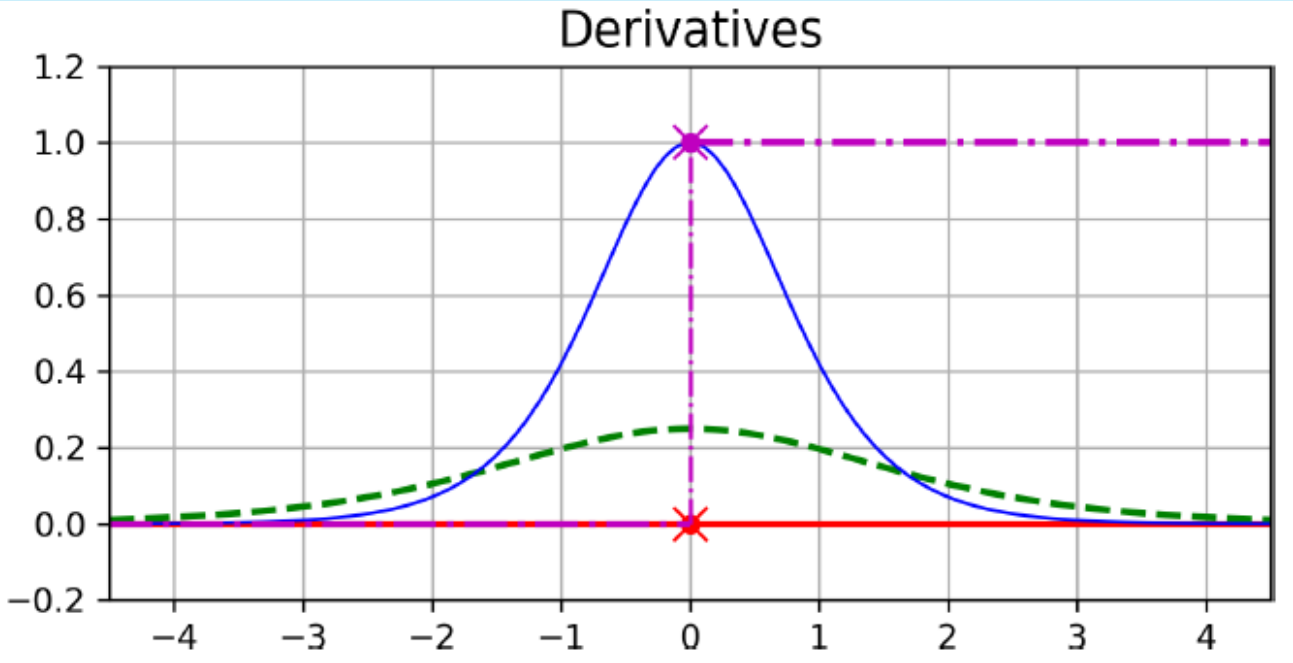


$$\mathrm{ReLU}(z) = \max(0, z)$$

- Simples e rápida de calcular
- Evita o vanishing gradient
- Função mais usada nas redes modernas

A ReLU revolucionou o treinamento de redes neurais profundas.
Sua lógica é simples: qualquer valor negativo vira zero, e qualquer valor positivo passa direto.
Isso tem duas grandes vantagens:
1.É muito rápida de calcular — basta comparar com zero.
2.Evita o desaparecimento dos gradientes, porque sua derivada é 1 para os valores positivos.
Na prática, isso faz com que o aprendizado seja muito mais estável e eficiente, mesmo em redes com dezenas de camadas.
O único cuidado é com os neurônios “mortos” — aqueles que ficam presos no valor zero e param de aprender. Mesmo assim, a ReLU é hoje a função padrão na maioria das arquiteturas modernas, como CNNs e MLPs.





OK! Você sabe de onde vieram as redes neurais, qual é a arquitetura delas e como calcular suas saídas. Você também aprendeu sobre o algoritmo de retropropagação. Mas o que exatamente você pode fazer com redes neurais?

2. Building and Training MLPs with Scikit-Learn

2.1 Regression MLPs

As redes Perceptron Multicamadas (MLPs) podem ser aplicadas a diversas tarefas, como regressão e classificação. No caso da regressão, o MLP prevê valores contínuos.
Se queremos prever um único valor (como o preço de uma casa), usamos um neurônio de saída. Para prever múltiplos valores simultaneamente, usamos um neurônio por variável de saída.

Por exemplo:
Para prever o centro de um objeto em uma imagem, precisamos de duas saídas (coordenadas x e y).
Para prever também largura e altura da caixa delimitadora, precisamos de quatro saídas no total.

O Scikit-Learn oferece a classe MLPRegressor, que facilita a criação e o treinamento de redes neurais para regressão. Aqui, usamos o dataset California Housing, que contém apenas variáveis numéricas, sem valores ausentes.

Cada unidade no alvo representa US$100.000.

```
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import root_mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

# Carrega e divide o conjunto de dados
housing = fetch_california_housing()
X_train, X_test, y_train, y_test = train_test_split(
    housing.data, housing.target, random_state=42)
```

O modelo terá 3 camadas ocultas, cada uma com 50 neurônios, e utilizará ReLU como função de ativação.
Nenhuma ativação é usada na camada de saída (permitindo qualquer valor real).
O parâmetro early_stopping=True ativa a parada antecipada, e verbose=True mostra o progresso do treino.

```
mlp_reg = MLPRegressor(
    hidden_layer_sizes=[50, 50, 50],
    early_stopping=True,
    verbose=True,
    random_state=42
)
```

O early stopping reserva automaticamente 10% dos dados de treino como conjunto de validação. O treino para automaticamente se o desempenho não melhora após 10 épocas.

Esse método reduz o risco de overfitting.
Além disso, o MLPRegressor aplica uma regularização L2 leve (controlada pelo hiperparâmetro alpha, padrão = 0.0001).
O otimizador usado é o Adam, uma variação moderna do gradiente descendente

Antes de treinar, é essencial padronizar os atributos, pois o gradiente descendente converge mal com escalas diferentes.
Usamos um pipeline com StandardScaler seguido pelo MLPRegressor.

```
pipeline = make_pipeline(StandardScaler(), mlp_reg)
pipeline.fit(X_train, y_train)
```

```
Iteration 1, loss = 0.85190332
Validation score: 0.534299
Iteration 2, loss = 0.28288639
Validation score: 0.651094
...
Iteration 45, loss = 0.12960481
Validation score: 0.788517
Validation score did not improve more than tol=0.000100 for 10 consecutive
epochs. Stopping.
```

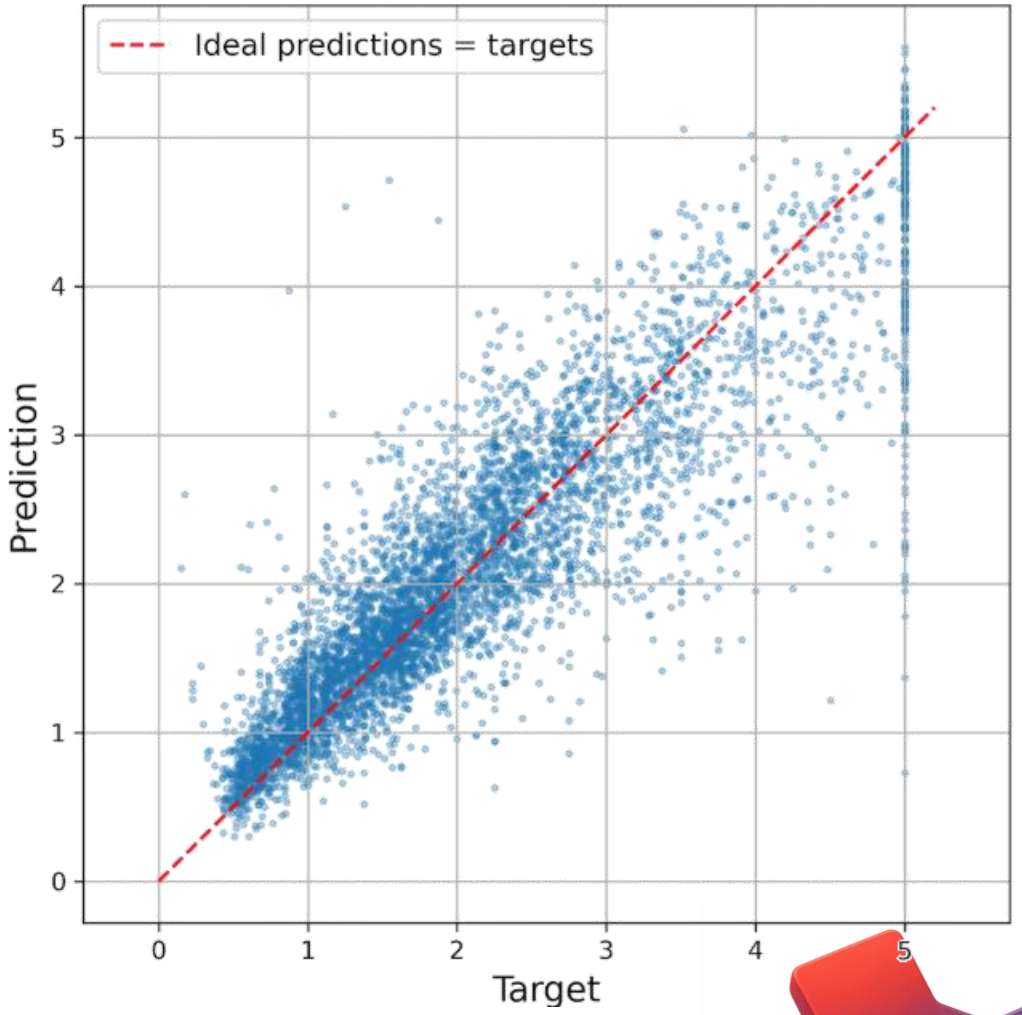
Após 45 épocas, o modelo atinge cerca de 79% de R^2 no conjunto de validação.

O método score() retorna o coeficiente R^2, que mede a proporção da variância explicada pelo modelo.
Para regressão, é comum também calcular o erro quadrático médio da raiz (RMSE).

```
y_pred = pipeline.predict(X_test)
rmse = root_mean_squared_error(y_test, y_pred)
print(rmse)
```

```
0.5327699946812925
```

O RMSE ≈ 0.53 — comparável ao desempenho de um Random Forest Regressor.



A Figura anterior mostra as previsões (eixo y) em função dos valores reais (eixo x).
A linha vermelha tracejada representa a predição ideal (y = x).
A maioria dos pontos está próxima da linha, indicando boas previsões, mas ainda há erros para valores maiores. O modelo não usa função de ativação na saída, então ele pode produzir qualquer valor.

Se quisermos restringir os valores de saída, podemos usar:
ReLU ou softplus → para garantir apenas saídas positivas,
Sigmoid → para saídas no intervalo [0, 1],
tanh → para saídas entre –1 e 1.
Fórmula – Softplus:

$$\text{softplus}(z) = \log(1 + e^z)$$

Essa função é uma versão suave da ReLU: próxima de 0 para z < 0, e próxima de z para z > 0.

O MLPRegressor do Scikit-Learn não suporta GPU, pois é totalmente baseado em CPU.
Suas opções são limitadas (por exemplo, não há ativação configurável na camada de saída).
Ainda assim, ele é extremamente prático para prototipagem rápida, permitindo treinar um MLP completo em poucas linhas de código.
Nos capítulos seguintes (a partir do 10), passaremos para PyTorch, que oferece GPU acceleration, autodiff completo e maior controle sobre as arquiteturas.

Em regressão, a perda mais comum é o Erro Quadrático Médio (MSE), ideal para dados limpos.
Quando há outliers, preferimos a Huber Loss, que é quadrática para erros pequenos e linear para erros grandes — combinando precisão e robustez.

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2, & |a| \leq \delta \\ \delta(|a| - \frac{1}{2}\delta), & |a| > \delta \end{cases}$$

(O MLPRegressor suporta apenas MSE, mas o Huber pode ser implementado em TensorFlow ou PyTorch.)

A tabela a seguir resume as configurações mais comuns para redes MLP aplicadas à regressão:

Hiperparâmetro	Valor típico
N° de camadas ocultas	1 a 5
Neurônios por camada	10 a 100
Neurônios de saída	1 por dimensão do alvo
Ativação oculta	ReLU
Ativação de saída	Nenhuma / ReLU / softplus / sigmoid / tanh
Função de perda	MSE ou Huber

Essa configuração é suficiente para a maioria dos problemas de regressão numérica, equilibrando simplicidade, desempenho e generalização.

O Scikit-Learn facilita a criação de MLPs para regressão com poucas linhas de código.
O MSE é a função de perda padrão, mas a Huber Loss é melhor em dados com outliers.
O early stopping evita overfitting e acelera o treino.
Para tarefas mais avançadas, com GPUs e arquiteturas personalizadas, PyTorch será a escolha ideal.

FPFtech