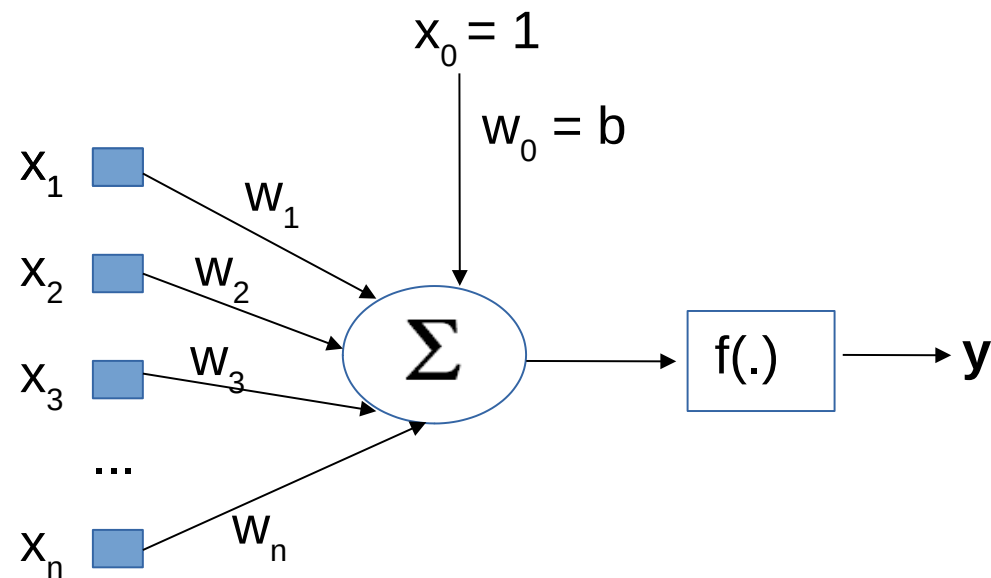


Ciência das Redes

Redes Neurais de Grafos

Ricardo Luders
Thiago H Silva





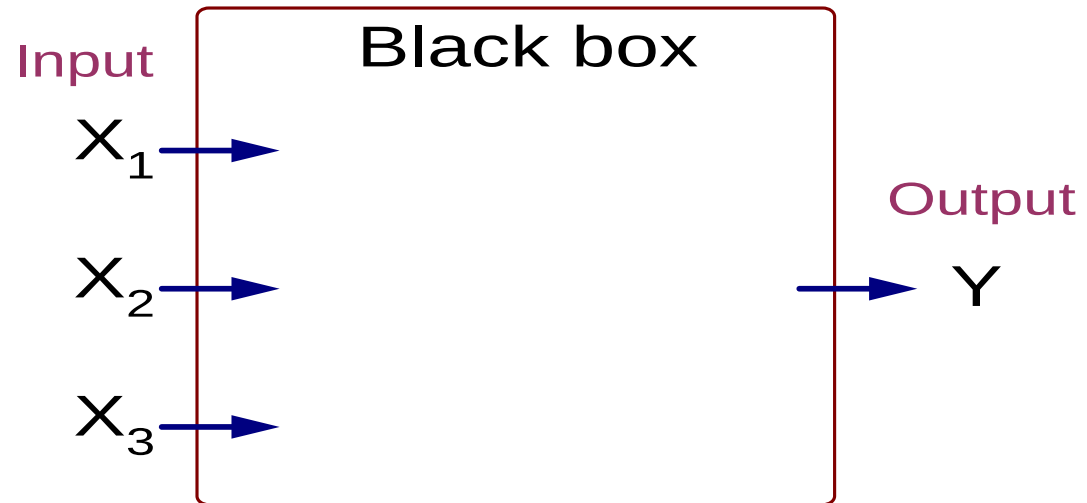
Neurônio artificial do tipo perceptron

\vec{x} = entrada (dendrito) e \vec{w} = peso sináptico

função somatória = corpo celular

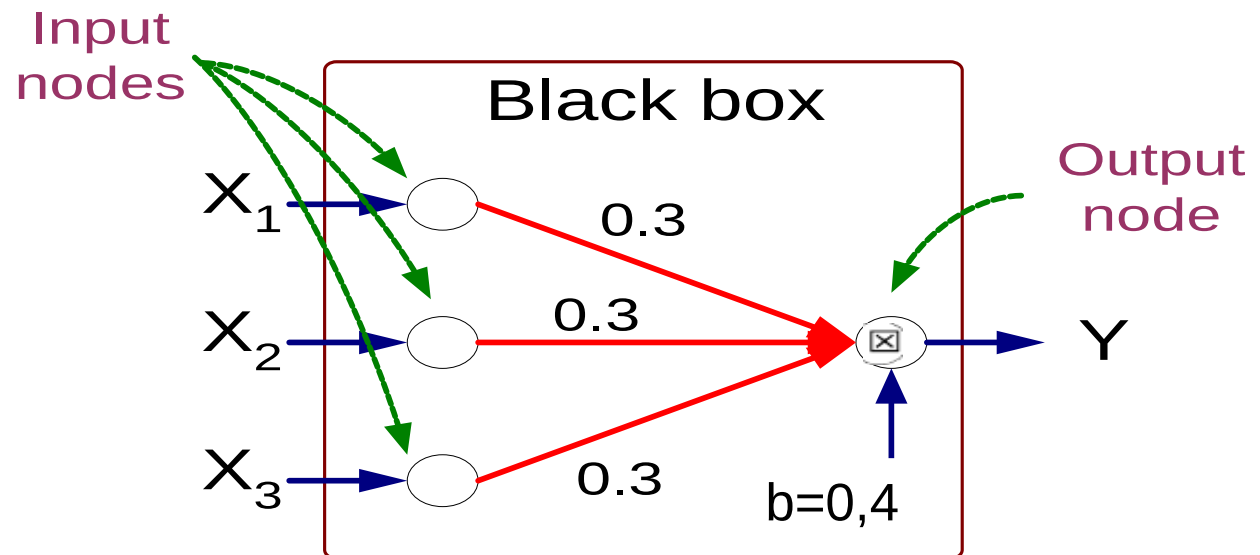
$f(\cdot)$ = função de ativação que gera a saída no axônio y

X_1	X_2	X_3	Y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1



Saída Y é 1 se pelo menos duas das três entradas são iguais a 1

X_1	X_2	X_3	Y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1



$$Y = \text{degrau}(0,3X_1 + 0,3X_2 + 0,3X_3 - 0.4)$$

$$\text{onde degrau}(v) = \begin{cases} 1 & \text{se } v \geq 0 \\ -1 & \text{se } v < 0 \end{cases}$$

Degrau = função degrau

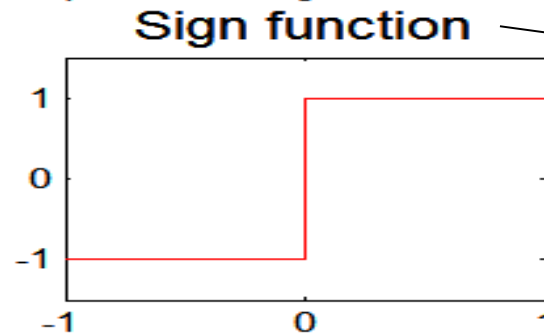
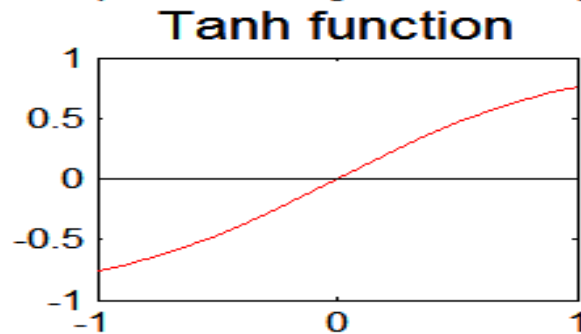
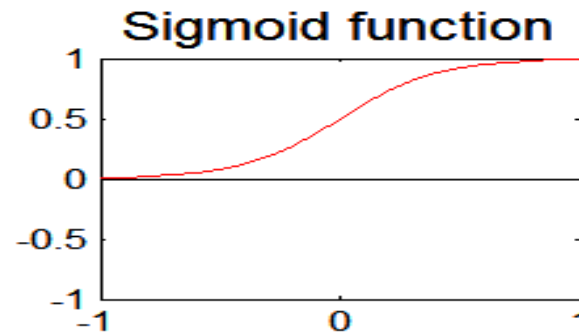
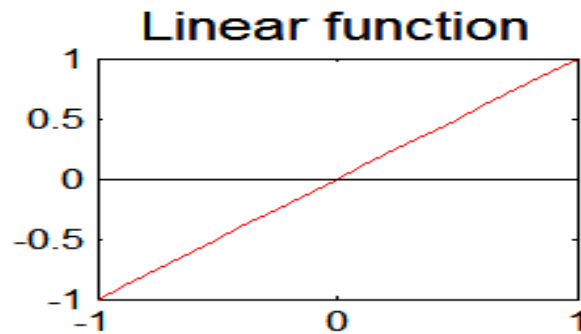
Tabela da verdade

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

O neurônio projetado para resolver o problema deve:

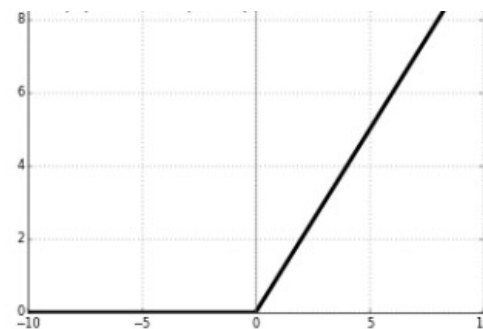
- Utilizar como entrada x_1 e x_2
- Ponderar as entradas com os pesos sinápticos
- Realizar um somatório
- Aplicar uma função de ativação para produzir uma saída \hat{y} que deve ser igual à y

Vários tipos de funções de ativação



Função
degrau

ReLU function



Para o exemplo definimos a função *sign* (degrau):

- Ela produzirá saída **1** quando o campo induzido for > 0 e **-1** caso contrário

A regra de processamento do neurônio pode ser definida como:

- cálculo do sinal que entra no neurônio o é

$$v_o = \sum_{i=1}^n (x_i * w_{oi}) + b_o$$

- x_i é o valor de entrada
- a saída do neurônio é $y_o = f(v_o)$

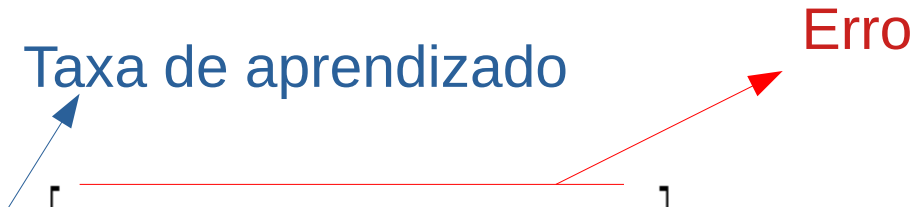
Inicializar os pesos (w_0, w_1, \dots, w_n)

Repetir

Para cada instância de treino (x_i, y_i)

Computar $f(w, x_i)$

Atualizar os pesos:

$$w^{(k+1)} = w^{(k)} + \lambda \left[y_i - f(w^{(k)}, x_i) \right] x_i$$


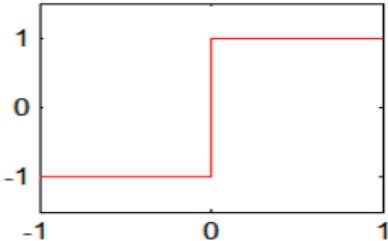
The diagram illustrates the components of the weight update equation. A blue arrow labeled "Taxa de aprendizado" points to the learning rate symbol λ . A red arrow labeled "Erro" points to the term $y_i - f(w^{(k)}, x_i)$, which represents the error.

Até que a condição de parada seja atingida

$$w^{(k+1)} = w^{(k)} + \lambda \left[y_i - f(w^{(k)}, x_i) \right] x_i$$

Taxa de aprendizado

Erro



Assuma:
 $f(x,w)$ é -1

Intuição para atualização da fórmula:

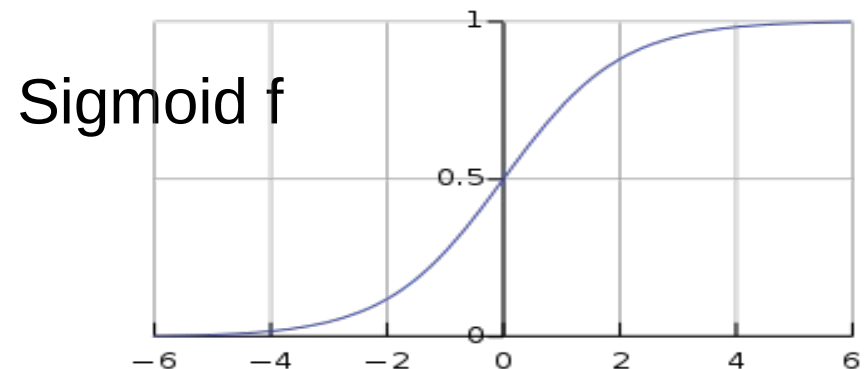
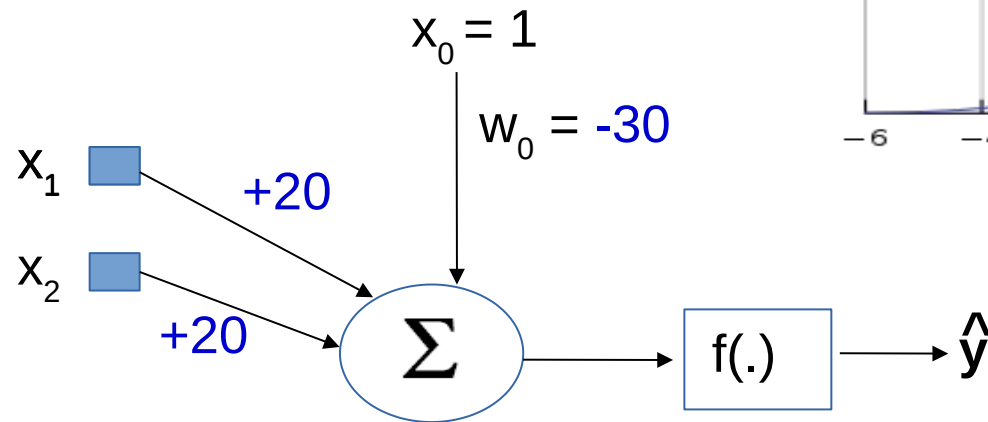
- 1- se $y=f(x,w)$: sem atualização
- 2- se $y>f(x,w)$: peso precisa aumentar assim $f(x,w)$ aumenta
- 3- se $y<f(x,w)$: peso precisa decrescer para diminuir $f(x,w)$

Exemplo, caso 2:
 $y - f(x,w) = 1 - (-1) = 2$

Outros critérios de aprendizado poderiam ser usados.

Exemplo com outra função de ativação:

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1



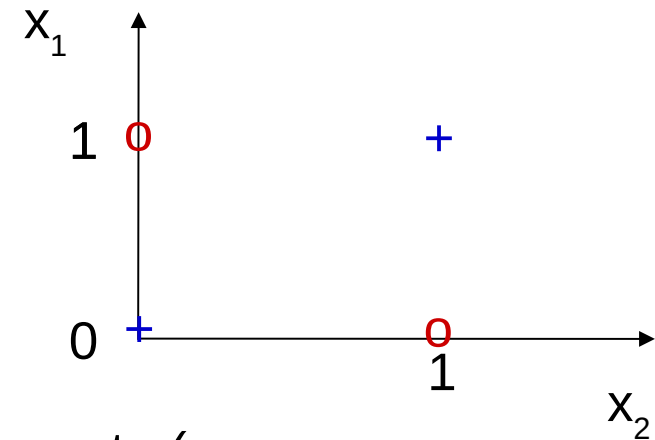
$$f(20x_1 + 20x_2 - 30)$$

x_1	x_2	\hat{y}
0	0	$f(-30) \sim 0$
0	1	$f(-10) \sim 0$
1	0	$f(-10) \sim 0$
1	1	$f(10) \sim 1$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Não é linearmente separável

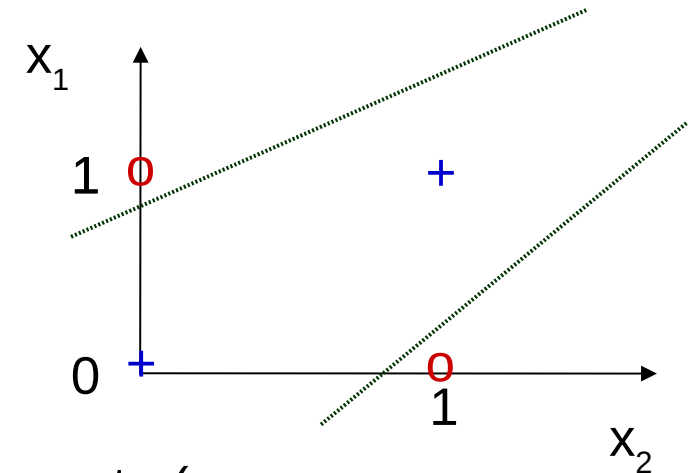
Ou seja, classes que podem ser separadas por uma reta (ou um hiperplano)



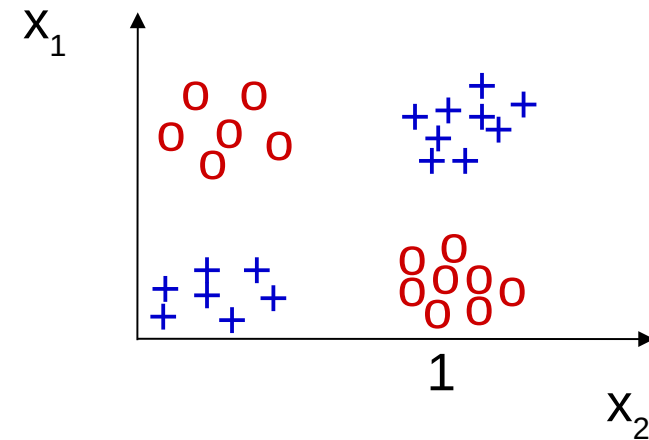
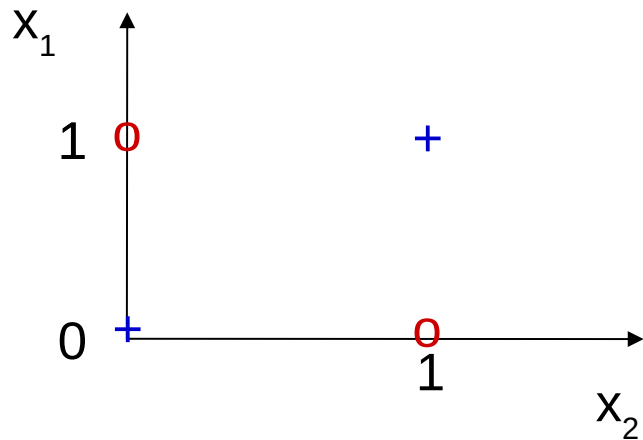
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Não é linearmente separável

Ou seja, classes que podem ser separadas por uma reta (ou um hiperplano)



Precisamos combinar mais de um neurônio
(possibilita combinar retas)



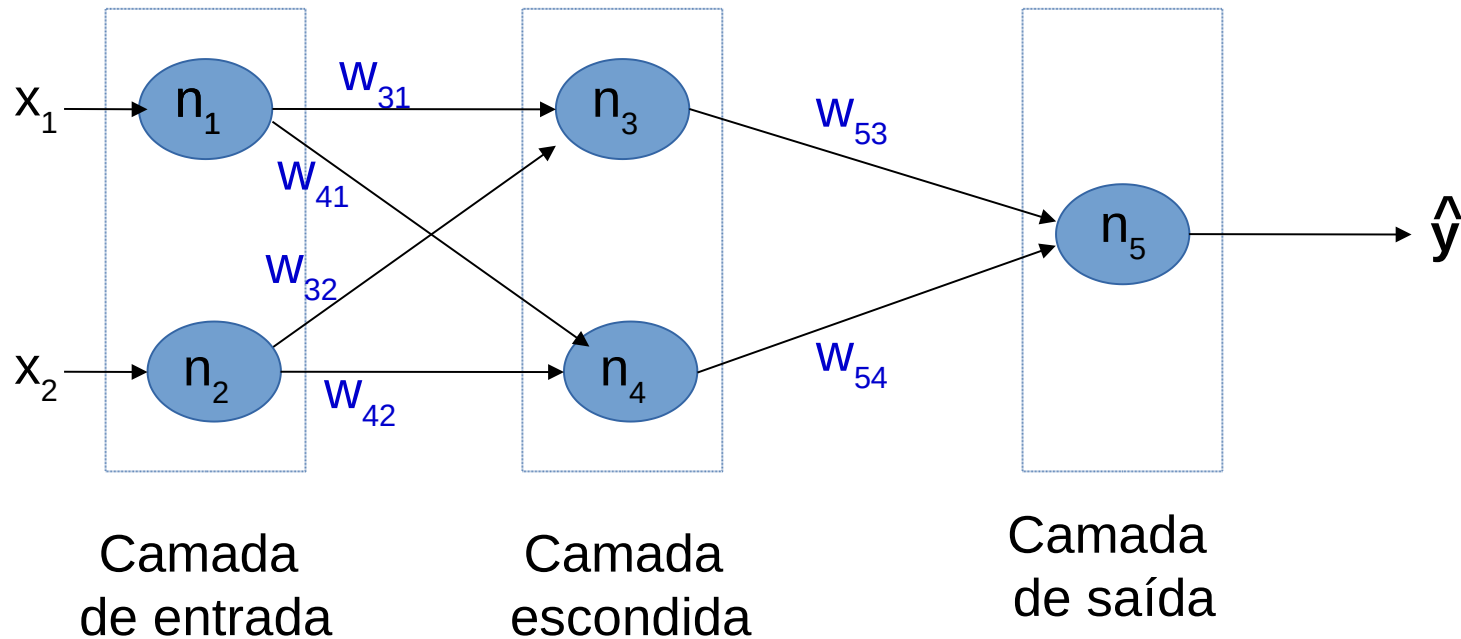
Problema à esquerda é uma versão simplificada (mais fácil de analisar) do problema a direita

Possibilidade de combinação de neurônios em uma rede de múltiplas camadas

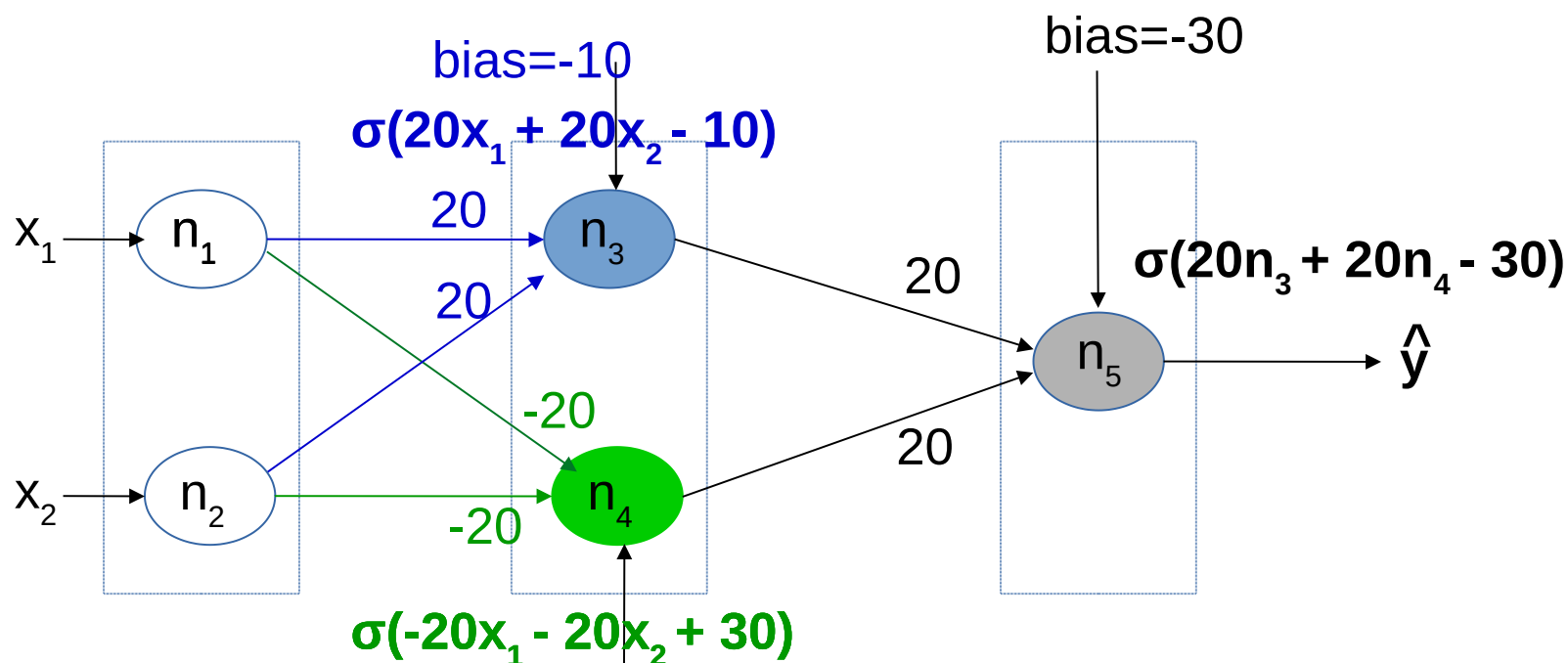
Possibilita a obtenção de estruturas mais complexas

Pode ser útil na resolução de tarefas que envolvem superfícies de decisão não-lineares

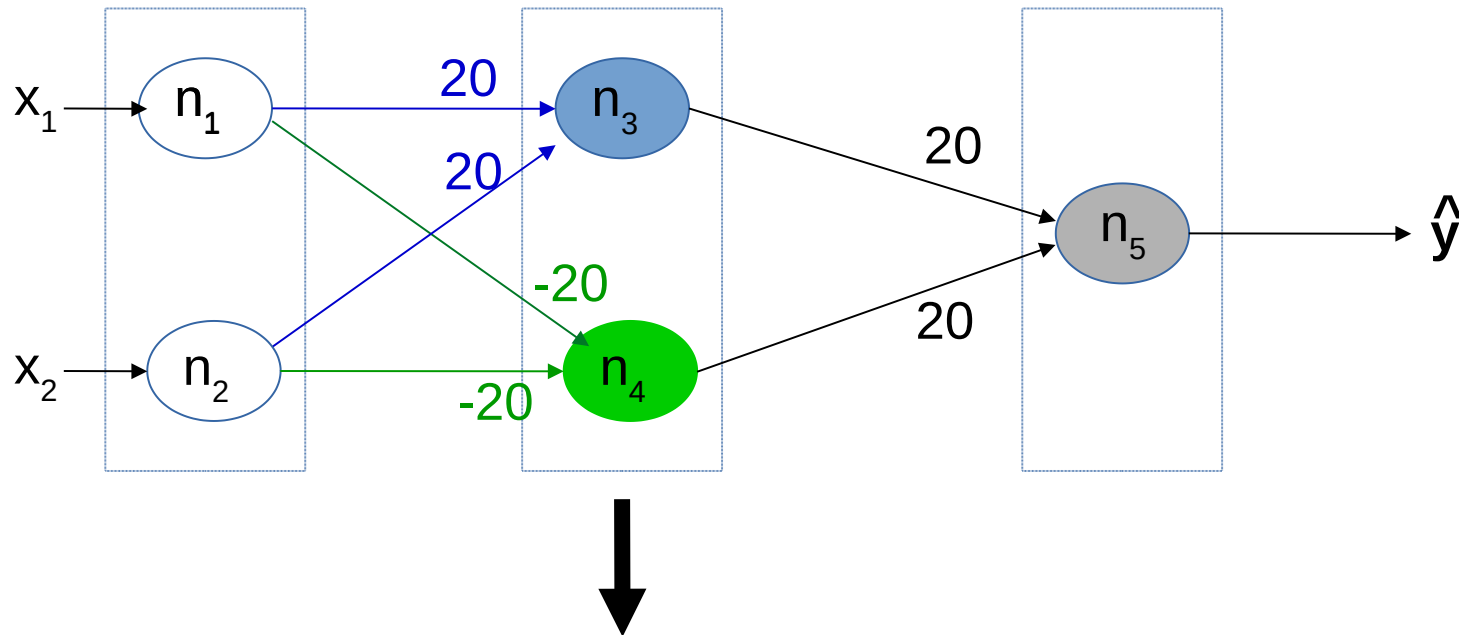
Rede neural de múltiplas camadas



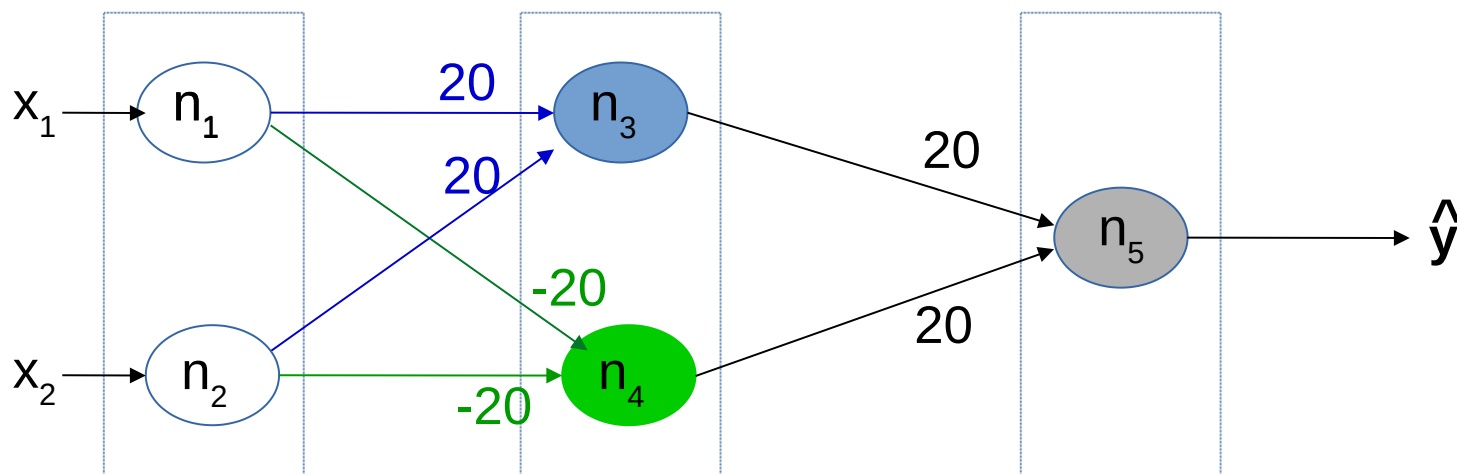
Rede neural de múltiplas camadas



x_1	x_2	n_3	n_4	\hat{y}
0	0	$\sigma(20*0 + 20*0 - 10) \sim 0$	$\sigma(-20*0 - 20*0 + 30) \sim 1$	$(20*0 + 20*1 - 30) \sim 0$
0	1	$\sigma(20*0 + 20*1 - 10) \sim 1$	$\sigma(-20*0 - 20*1 + 30) \sim 1$	$(20*1 + 20*1 - 30) \sim 1$
1	0	$\sigma(20*1 + 20*0 - 10) \sim 1$	$\sigma(-20*1 - 20*0 + 30) \sim 1$	$(20*1 + 20*1 - 30) \sim 1$
1	1	$\sigma(20*1 + 20*1 - 10) \sim 1$	$\sigma(-20*1 - 20*1 + 30) \sim 0$	$(20*1 + 20*0 - 30) \sim 0$



Não há como calcular o erro nos neurônios da camada escondida de forma direta, pois não existe a resposta desejada para tais neurônios.



x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

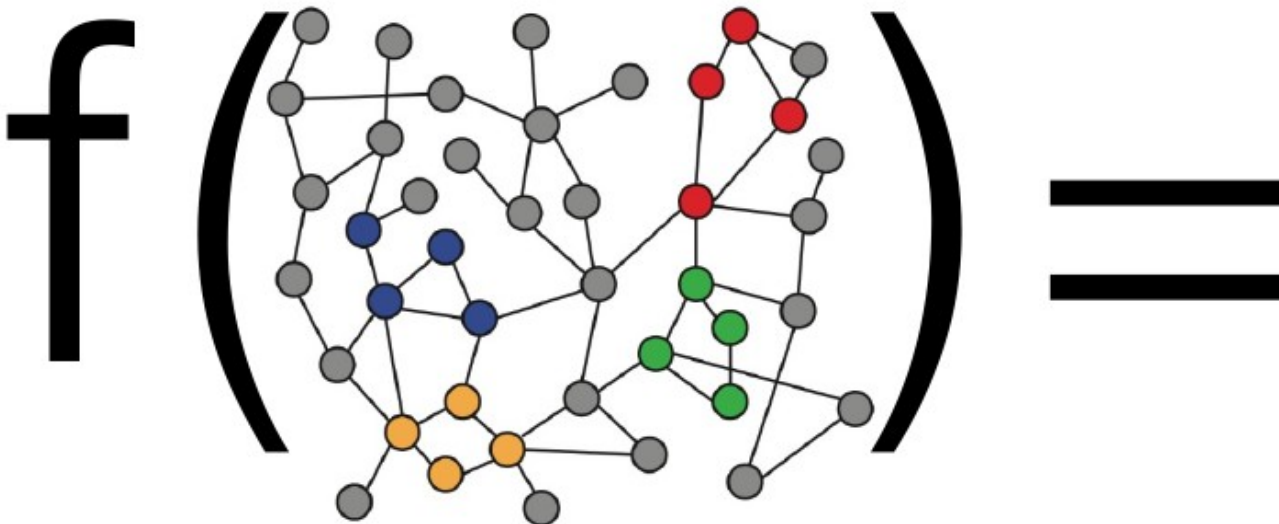
→ Erro calculado

Erro propagado

Pesos reajustados levemente

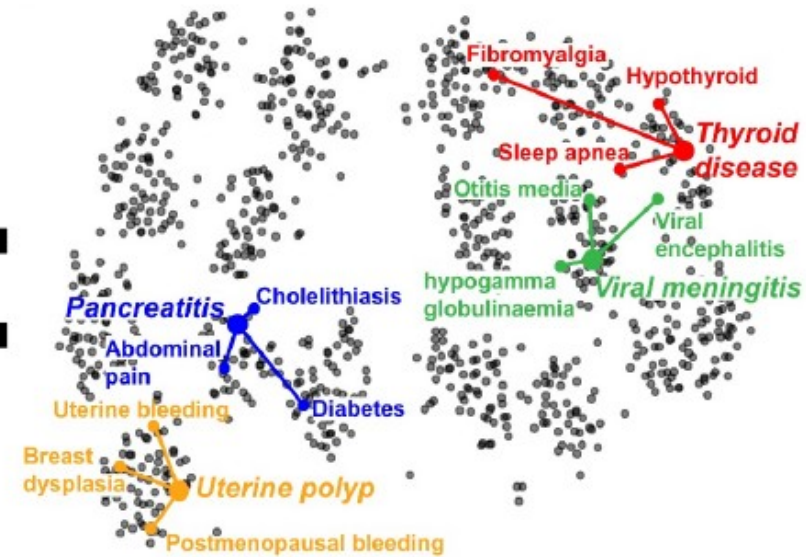
O processo é repetido para todas as entradas e saídas até que o erro seja pequeno ou outra condição imposta.
Após esse processo a rede é considerada treinada

Intuição: Mapear os nós para *d-dimensional embeddings* de modo que nós semelhantes no gráfico são próximos juntos



Grafo de entrada

=

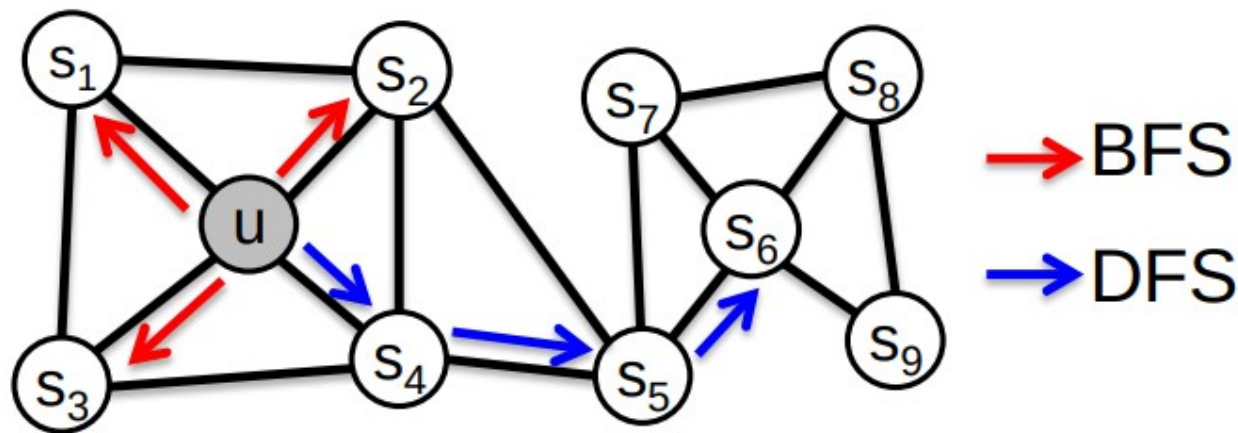


2D node embeddings

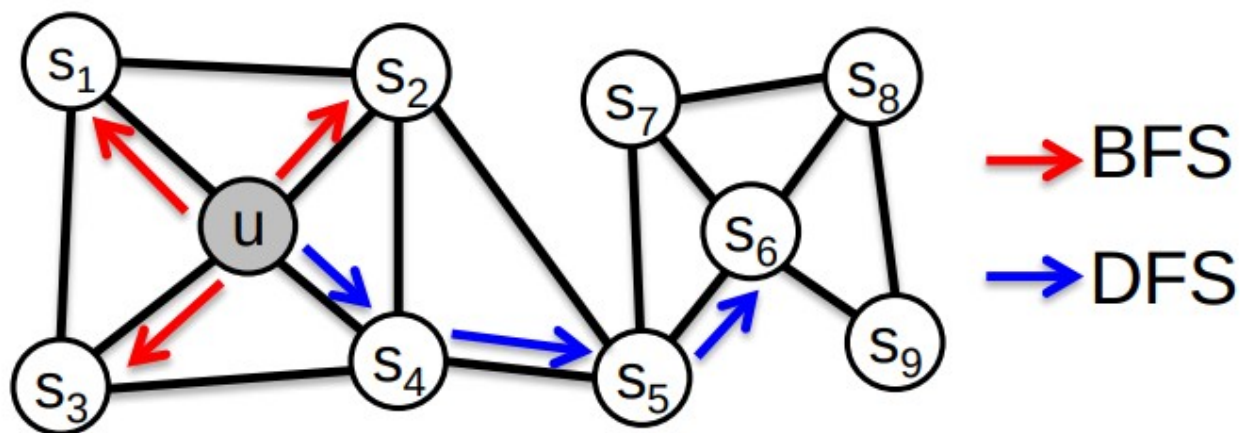
Como aprender a função de mapeamento f ?

Node2vec

Ideia: usar *random walks* flexíveis e tendenciosos que possam alternar entre as visões locais e globais da rede (Grover e Leskovec, 2016).



Duas estratégias clássicas para definir uma vizinhança N_u de um determinado nó u



$$N_{BFS}(u) = \{s_1, s_2, s_3\} \quad \text{Local – visão microscópica}$$

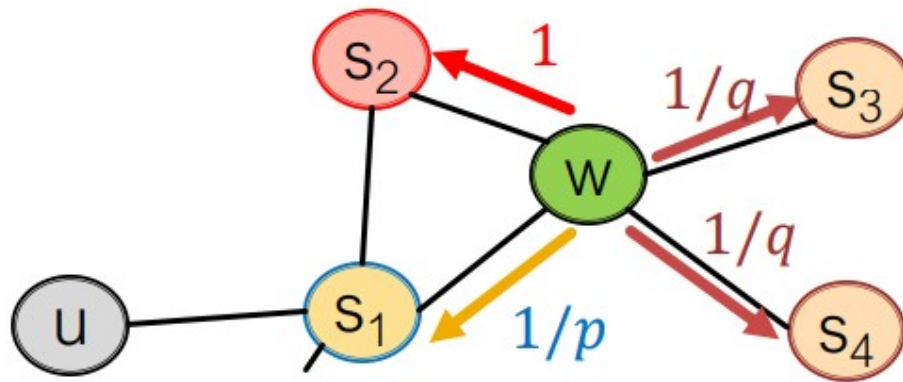
$$N_{DFS}(u) = \{s_4, s_5, s_6\} \quad \text{Global – visão macroscópica}$$

Random walks de comprimento fixo enviesado R que dado um nó u gera vizinhança Nu

Dois parâmetros:

- **Parâmetro de retorno p :**
 - Retornar ao nó anterior
- **Parâmetro de entrada-saída q :**
 - Movendo-se para fora (DFS) vs. para dentro (BFS)
 - Intuitivamente, q é a “razão” de BFS vs. DFS

Foi percorrido o caminho (S_1, w) e uma decisão precisa ser tomada em w



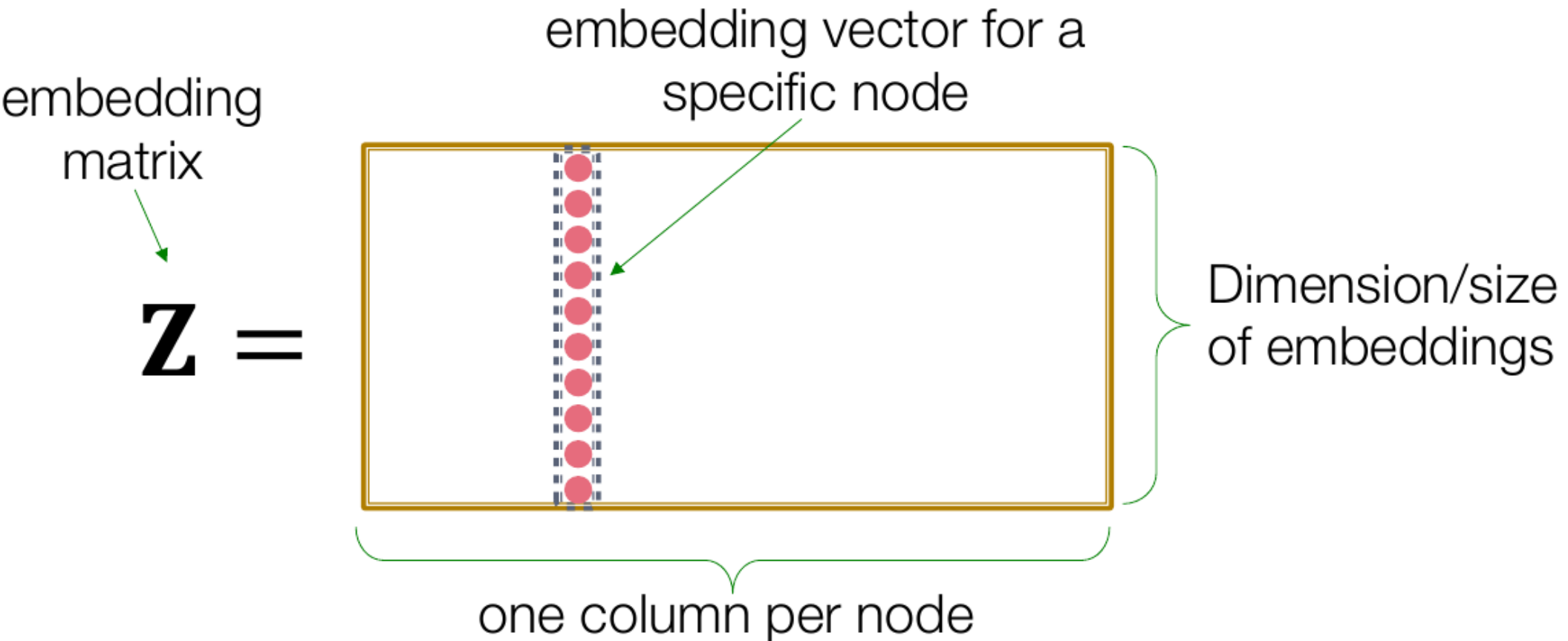
1 , $1/q$ e $1/p$ são valores de probabilidade não normalizados

p e q modelam probabilidade de transições:

p : parâmetro de retorno	—————▶	Baixo p (caminho BFS)
q : parâmetro de “afastamento”	—————▶	Baixo q (caminho DFS)

Memória da origem é mantida

Shallow encoders



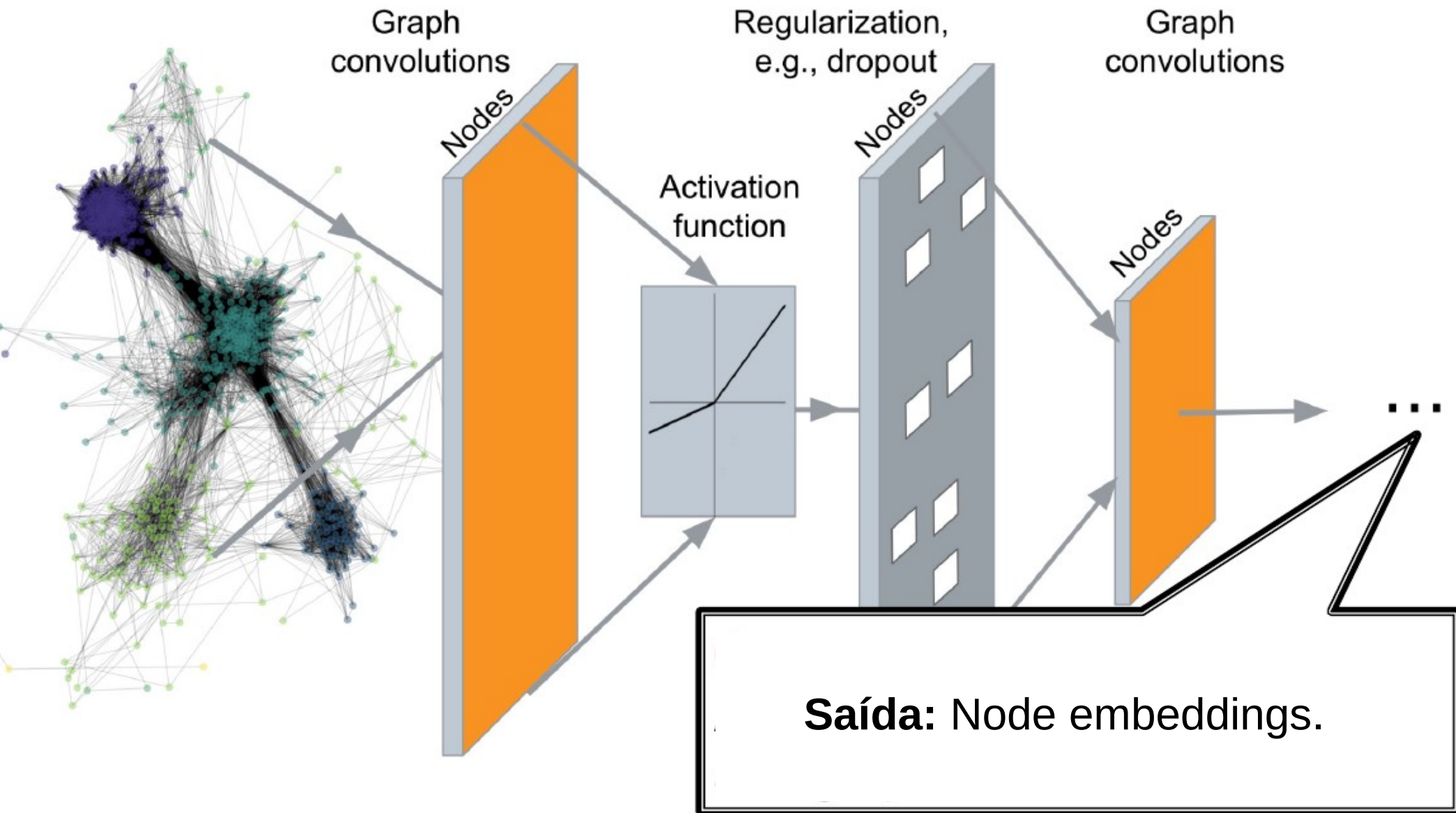
Abordagem de codificação mais simples: o codificador é apenas um *embedding lookup*

Limitações dos métodos de *shallow encoders* (como o node2vec):

- São necessários $O(|V|)$ Parâmetros:
 - Sem compartilhamento de parâmetros entre os nós
 - Cada nó tem seu próprio *embedding*
- Não incorpora *features* de nó:
 - Os nós em muitos grafos possuem recursos que podemos e devemos usar

Podemos usar métodos de aprendizagem profunda com base em redes neurais de grafos (GNNs):

$\text{ENC}(v) =$ Múltiplas camadas de transformações não lineares baseadas na estrutura do grafo



Classificação de nó

Preveja um tipo de um determinado nó

Previsão de link

Prever se dois nós estão ligados

Deteccção de comunidades

Identificar *clusters* densamente ligados de nós

Similaridade de rede

Quão semelhantes são duas (sub) redes



PyTorch é uma biblioteca de aprendizado de máquina de código aberto baseada na biblioteca Torch.

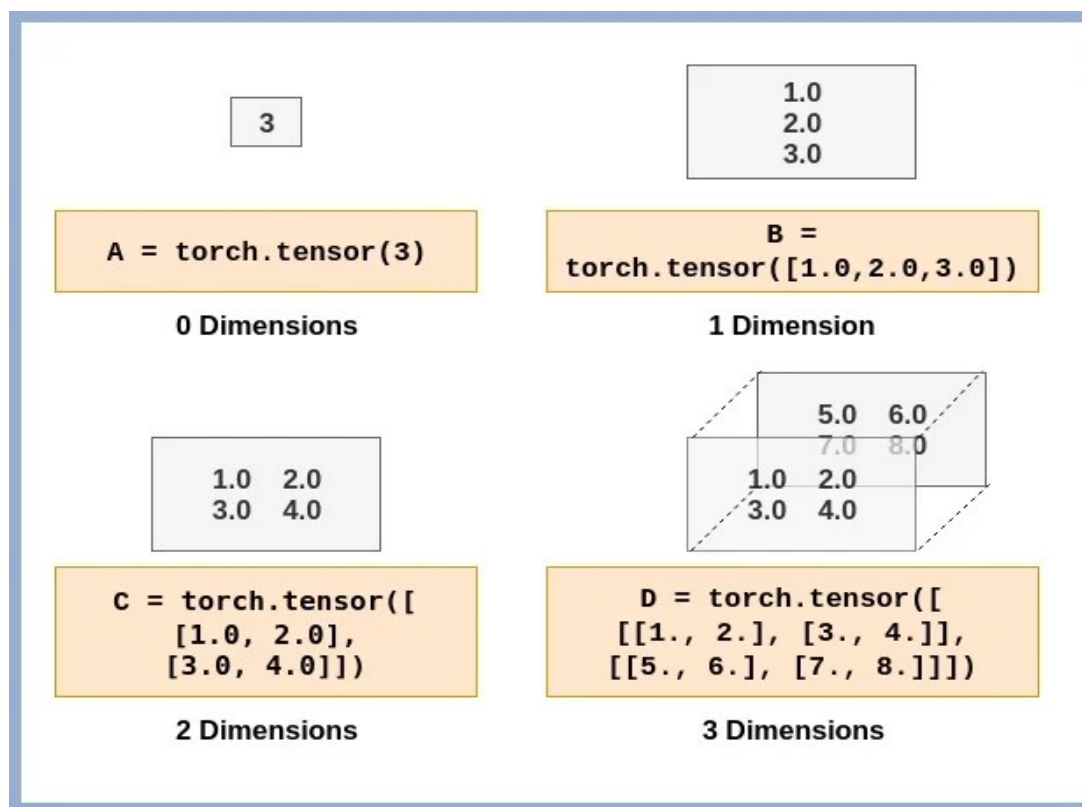
Usada para aplicações como visão computacional, processamento de linguagem natural, entre outras.

Desenvolvida principalmente pelo laboratório AI Research do Facebook.

Tensores são uma estrutura de dados especializada muito semelhante a arrays e matrizes.

No PyTorch, usamos tensores para codificar as entradas e saídas de um modelo, bem como os parâmetros do modelo.

Os tensores são semelhantes aos ndarrays do NumPy, exceto que os tensores podem ser executados em GPUs.



Mais de 100 operações de tensor, incluindo aritmética, álgebra linear, manipulação de matriz são descritas aqui de forma abrangente.

Cada uma dessas operações pode ser executada na GPU (normalmente em velocidades mais altas do que em uma CPU).

Por padrão, tensores são criados na CPU.

Precisamos mover explicitamente os tensores para a GPU usando o método `.to` (após verificar a disponibilidade da GPU).

Ver demonstração prática