



# Regressão e redes neurais rasas

Juvenal J. Duarte



# Multilayer Perceptron (MLP)

*Introdução*

Juvenal J. Duarte

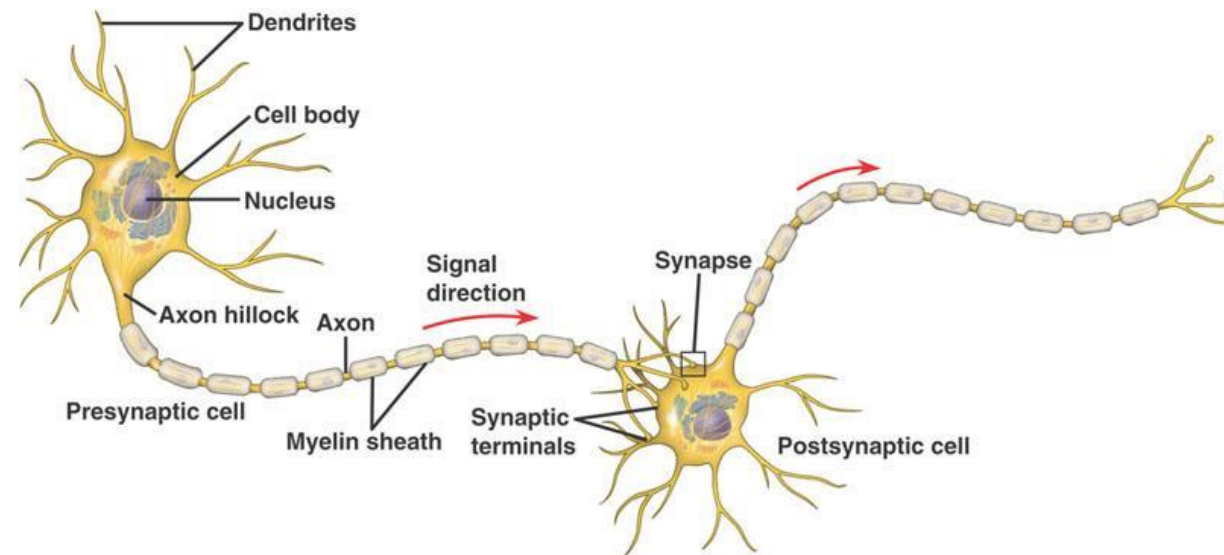
# Redes Neurais Artificiais (RNAs)

1940 – O surgimento dos primeiros computadores dá início a busca pelos primeiros modelos matemáticos do sistema nervoso.

1943 – Proposta de um neurônio artificial por McCulloch e Pitts, mostrando que funções complexas podiam ser modeladas por redes de neurônios simples.

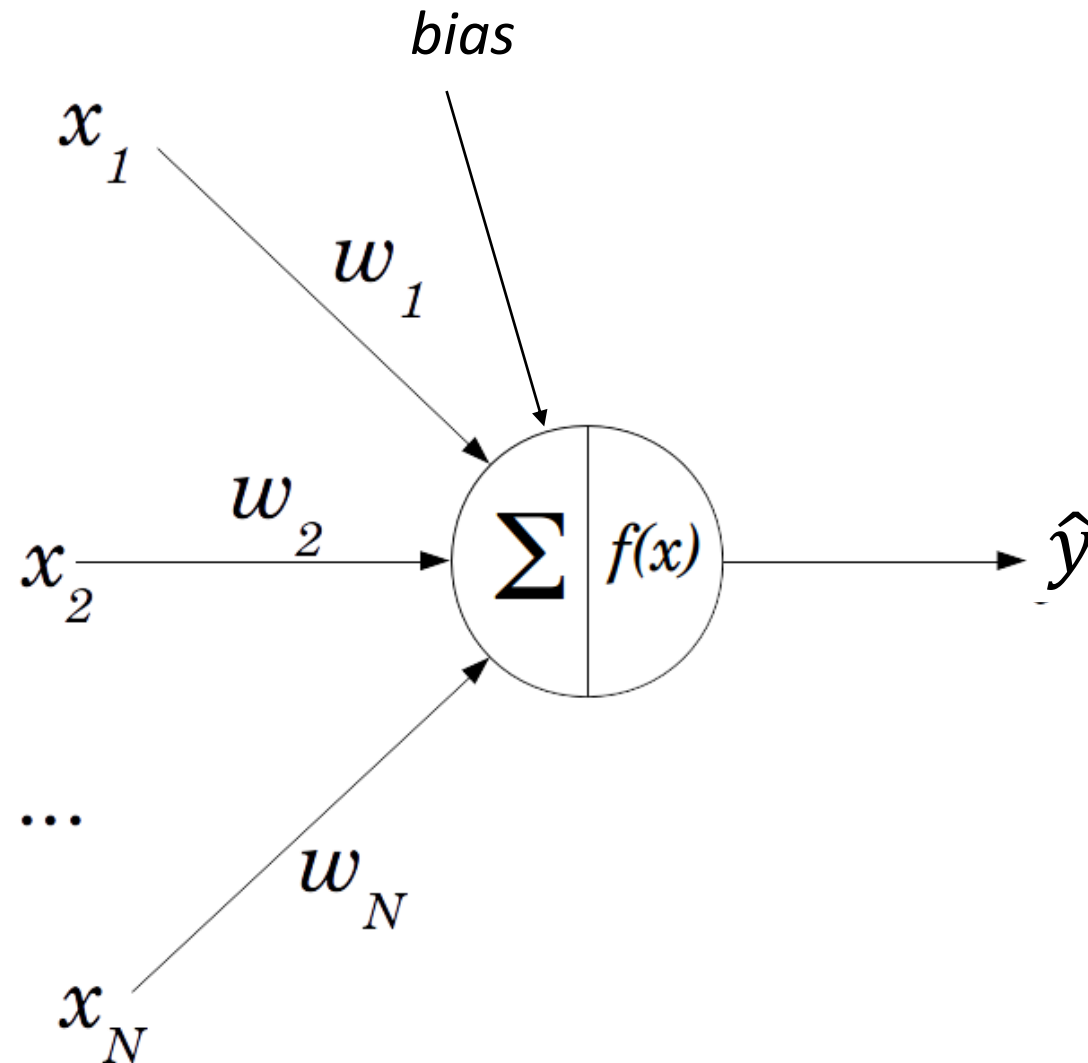
1958 – Primeira RNA proposta por Rosenblatt: o perceptron. Nessa rede foi proposta um dos primeiros algoritmos de aprendizado. Estudos posteriores demonstraram que o modelo possuía desempenho limitado apenas a dados linearmente separáveis.

1989 – Segundo Cybenko uma rede com uma camada intermediária pode representar qualquer função contínua, com duas camadas intermediárias qualquer função.

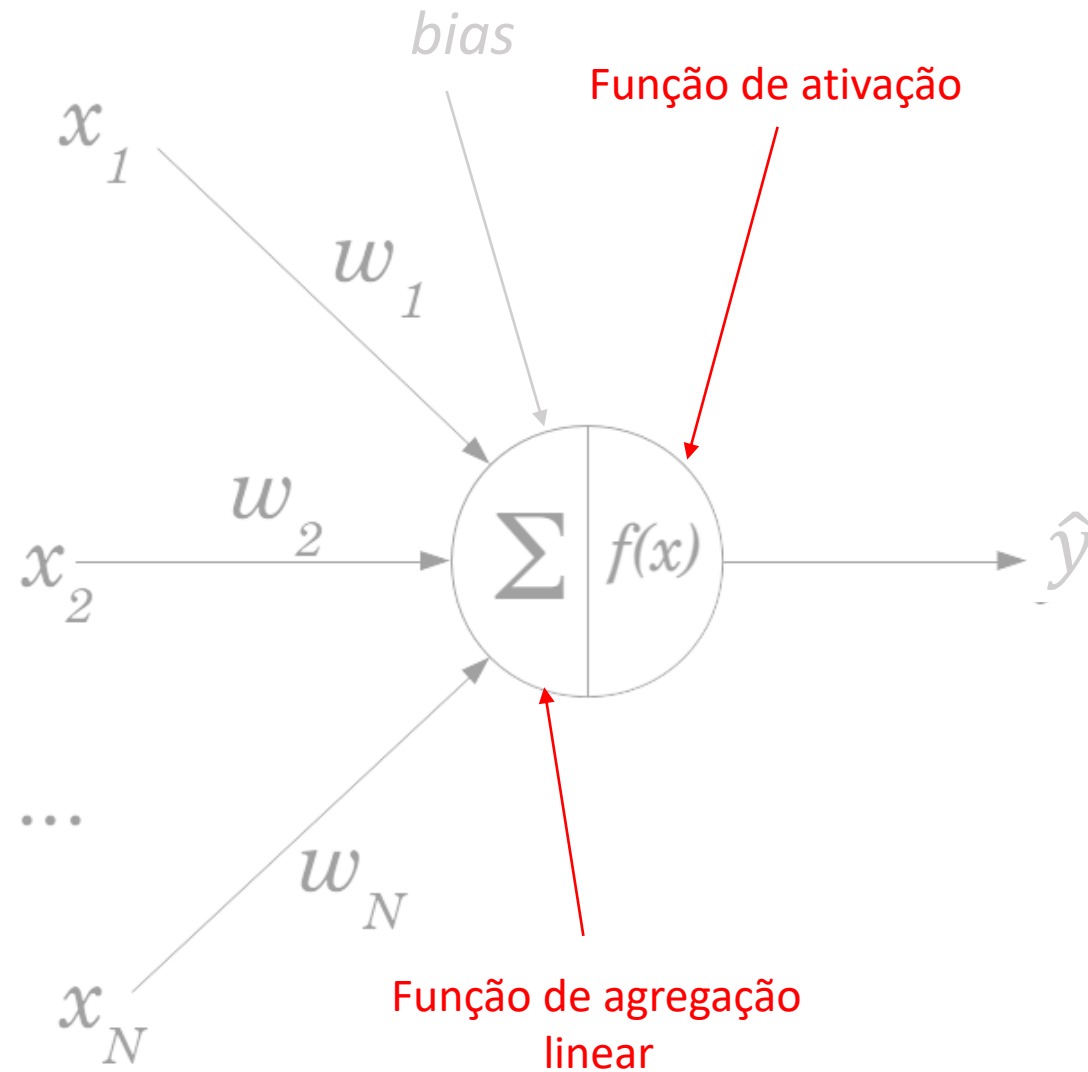


2015, Katti Facelli et al, *Inteligência Artificial Uma Abordagem de Aprendizado de Máquina*

# *O Perceptron*

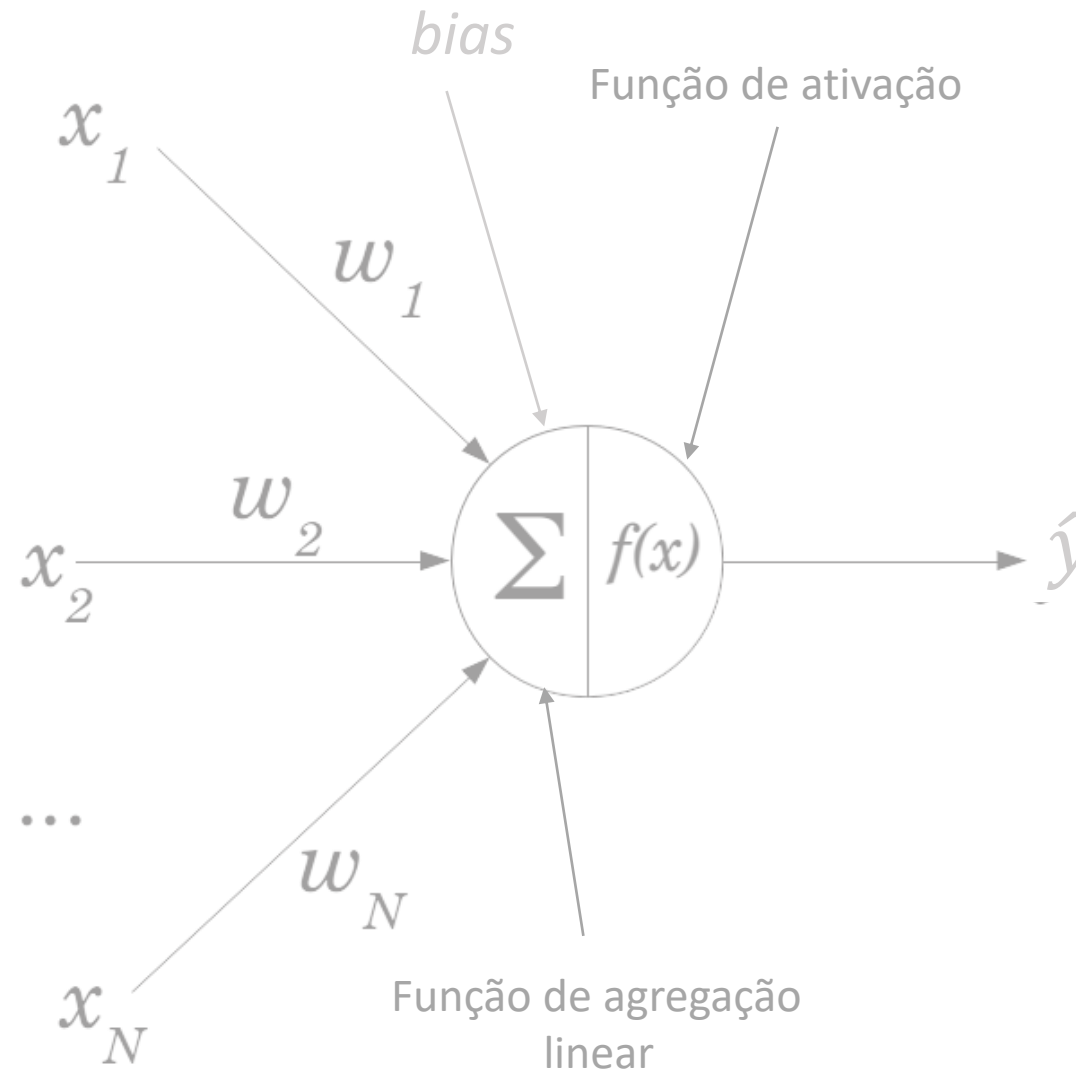


# O Perceptron



# O Perceptron

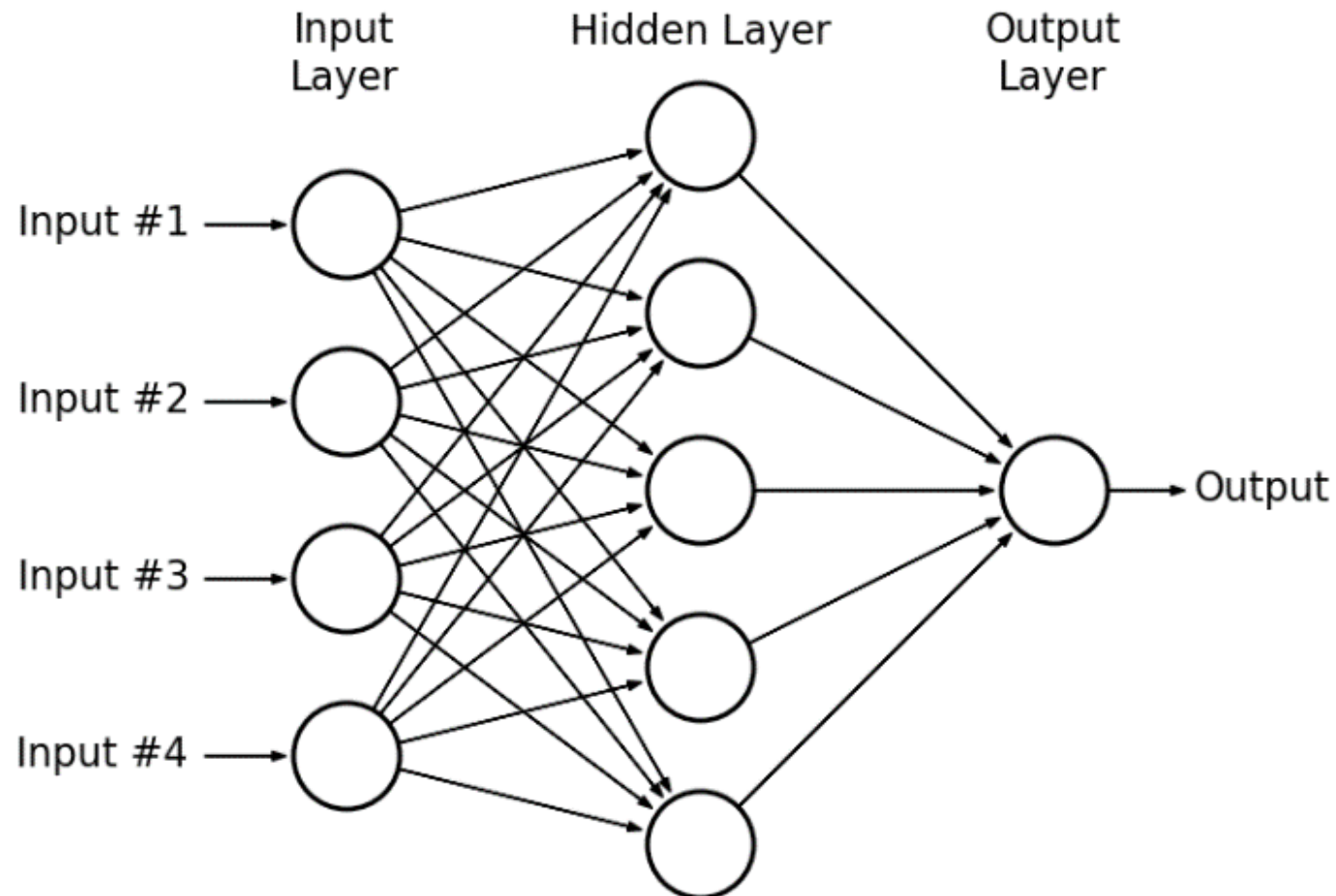
O perceptron, um único neurônio, possui as mesmas características de um classificador por regressão logística.



**E se estendermos a arquitetura do classificador para múltiplos perceptrons interconectados?**

# *O Multilayer Perceptron*

A arquitetura de Redes Neurais Artificiais (RNA) é organizada em múltiplos neurônios distribuídos em camadas.



# *O Multilayer Perceptron: Arquitetura*

## ***Camada de entrada:***

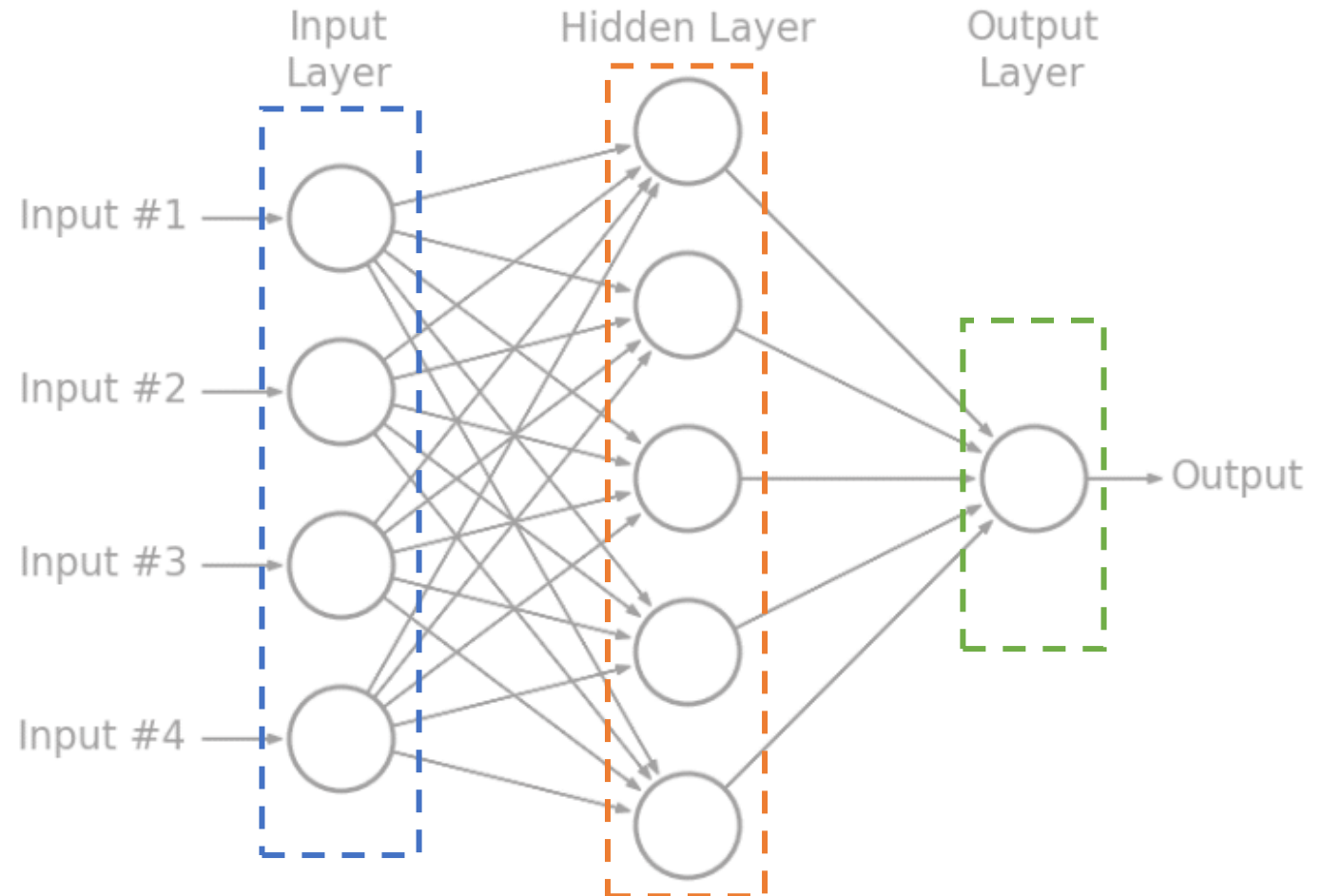
Cada neurônio na camada de entrada representa um atributo no dataset.

## ***Camada(s) intermediárias:***

Pode ter tantos neurônios e camadas quanto desenvolvedor desejar. Quanto maior a arquitetura, mais complexa é a função de decisão/regressão.

## ***Camada de saída:***

O(s) neurônio(s) na camada de saída apresentam a predição da rede. Usa-se um neurônio para problemas binários ou um neurônio por classe para classificação multi-classe (One Hot Encoding).





# O Multilayer Perceptron: Arquitetura

## **Camada de entrada:**

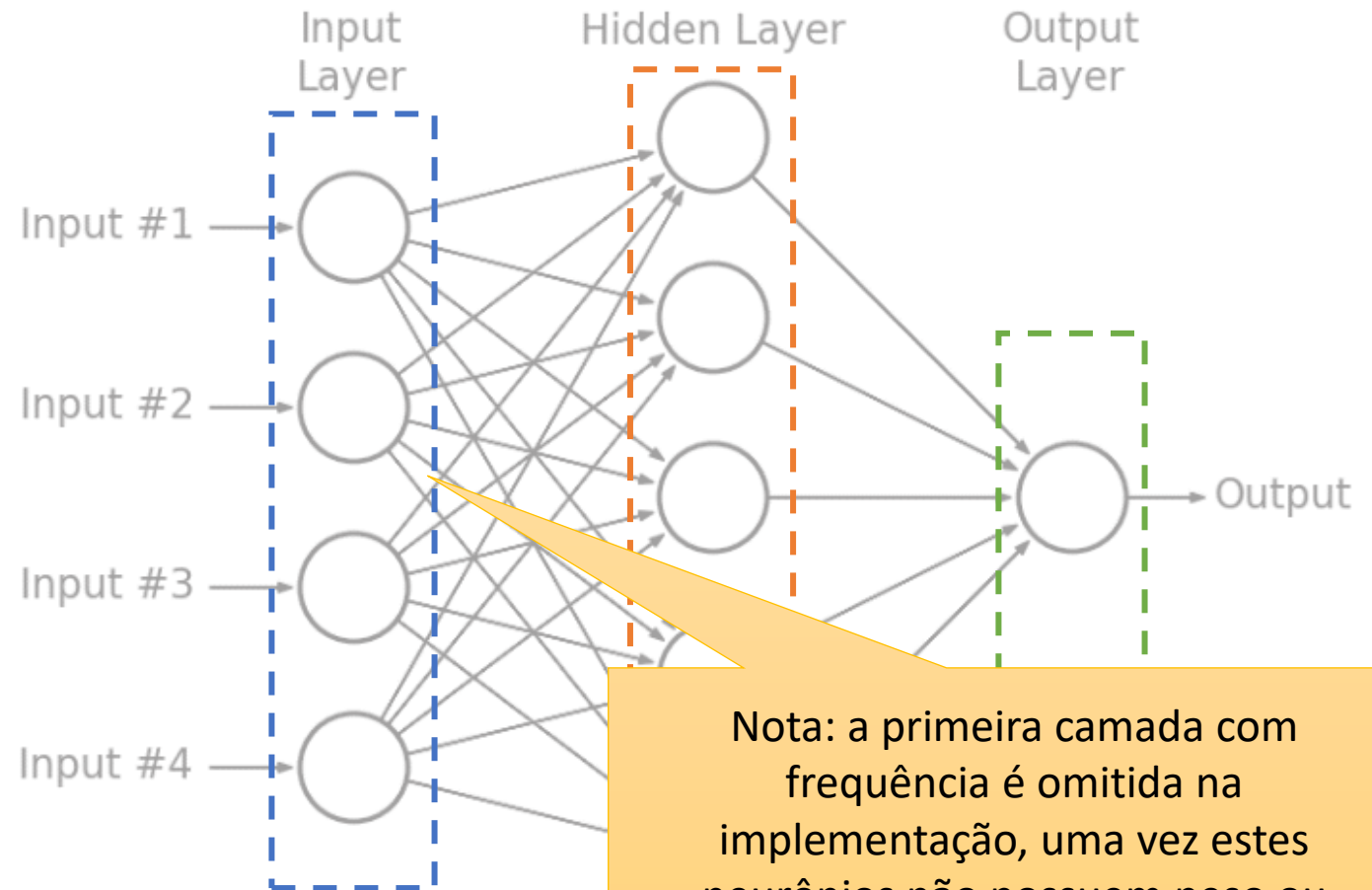
Cada neurônio na camada de entrada representa um atributo no dataset.

## **Camada(s) intermediárias:**

Pode ter tantos neurônios e camadas quanto desenvolvedor desejar. Quanto maior a arquitetura, mais complexa é a função de decisão/regressão.

## **Camada de saída:**

O(s) neurônio(s) na camada de saída apresentam a predição da rede. Usa-se um neurônio para problemas binários ou um neurônio por classe para classificação multi-classe (One Hot Encoding).



Nota: a primeira camada com frequência é omitida na implementação, uma vez estes neurônios não possuem peso ou ativação (nada é calculado!).

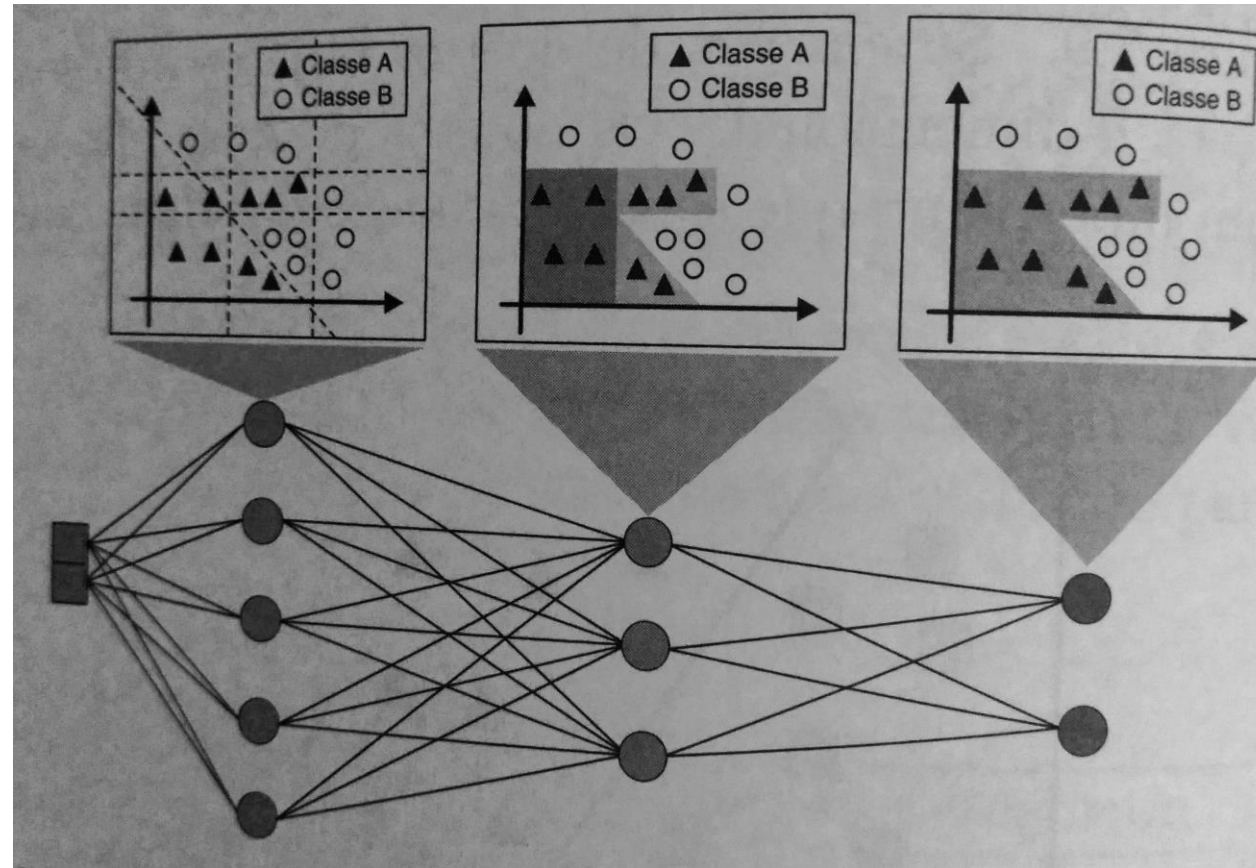


# Redes Neurais Artificiais

*Poder de representatividade*

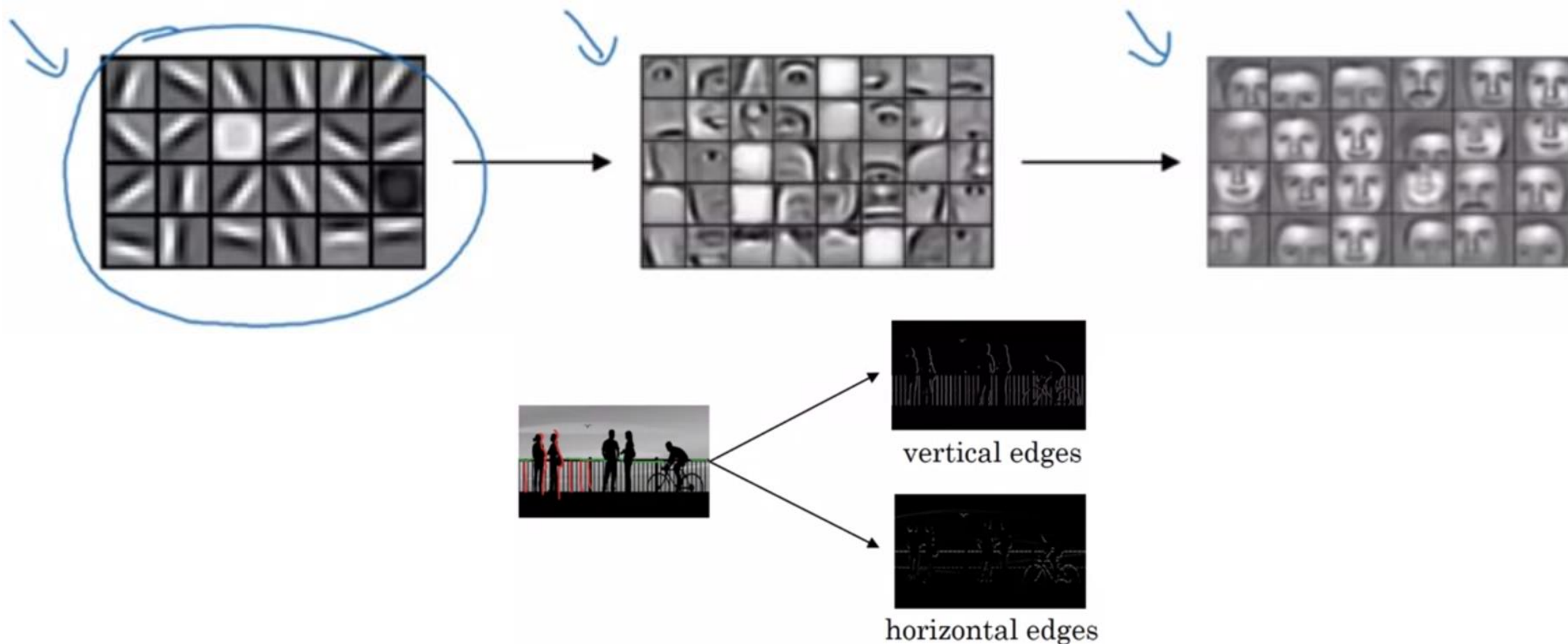
Juvenal J. Duarte

# *O Multilayer Perceptron: Interpretação*



2015, Katti Facelli et al, *Inteligência Artificial Uma Abordagem de Aprendizado de Máquina*

# Visão Computacional: Interpretação

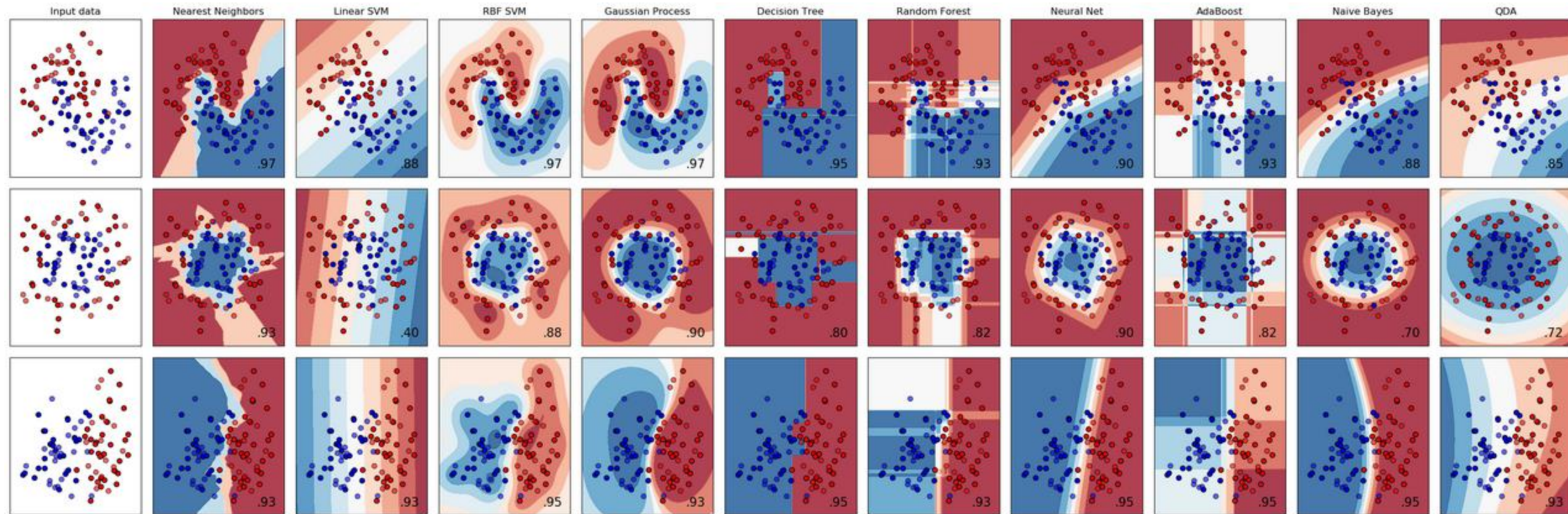


Exemplos extraídos de notas de aula do professor Andrew NG  
Facens - Especialização em Inteligência Artificial Aplicada



# Curiosidade:

*Comparação de funções de decisão, manual do SKLearn*





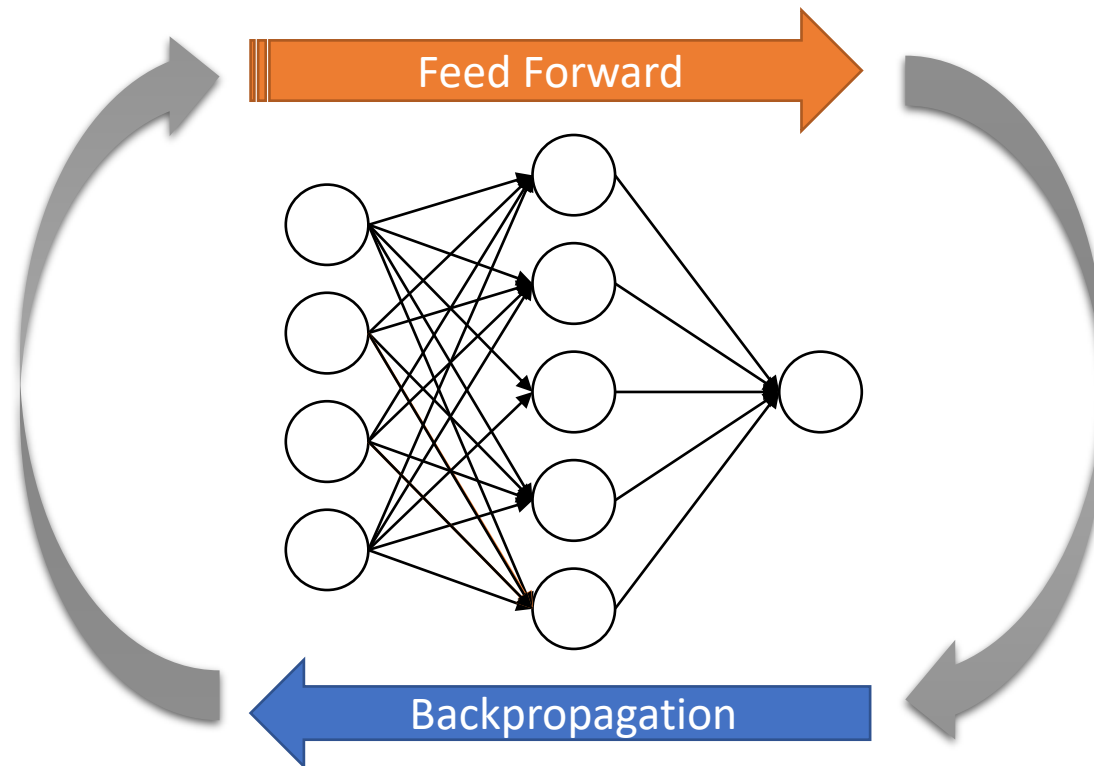
# Multilayer Perceptron (MLP)

*Funcionamento*

Juvenal J. Duarte

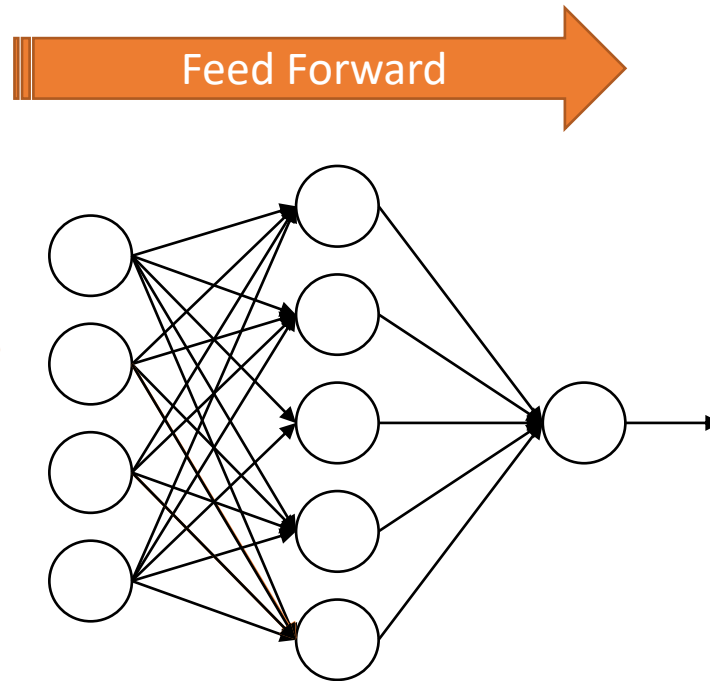
# *O Multilayer Perceptron: Fluxo*

O ciclo de *feed forward* e *backpropagation* se repete por tantas épocas quanto parametrizadas.



# *O Multilayer Perceptron: Propagação*

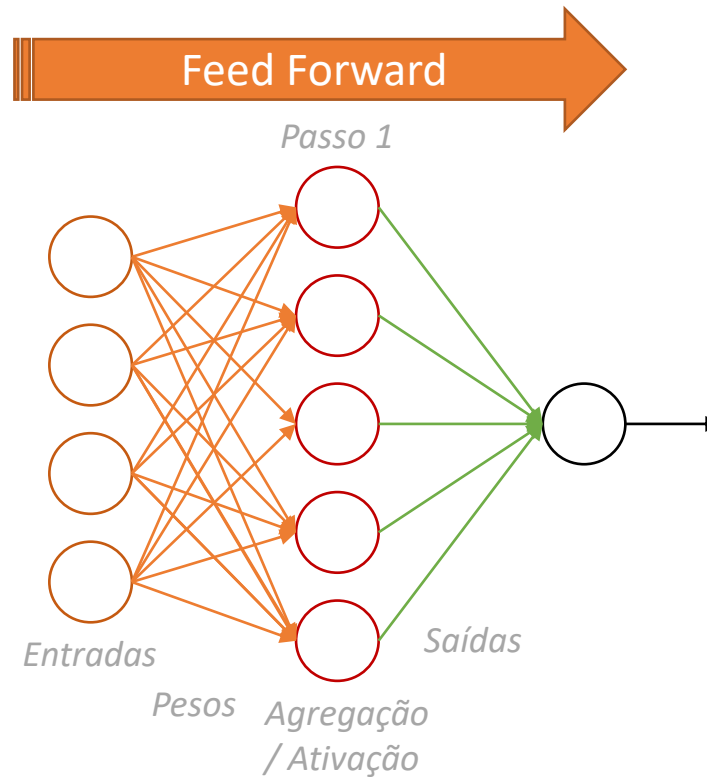
O algoritmo de Feed Forward propaga os valores de entrada por cada um dos neurônios, camada por camada





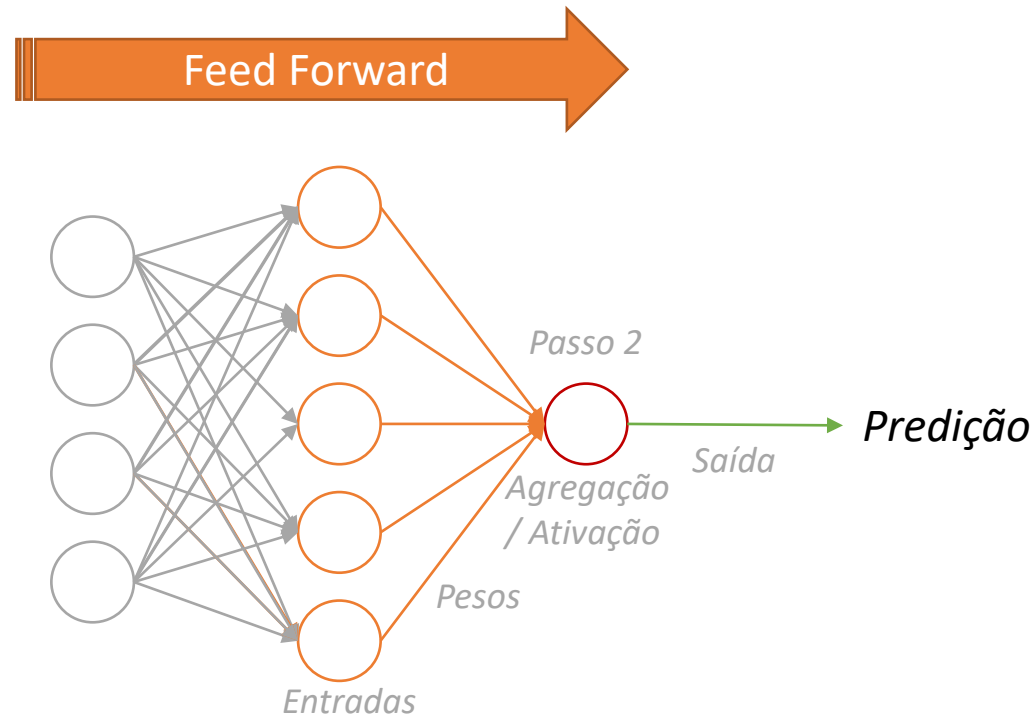
# O Multilayer Perceptron: Propagação

1. Associa cada um dos valores da camada de entrada à pesos e computa a ativação de cada um dos neurônios da segunda camada.



# *O Multilayer Perceptron: Propagação*

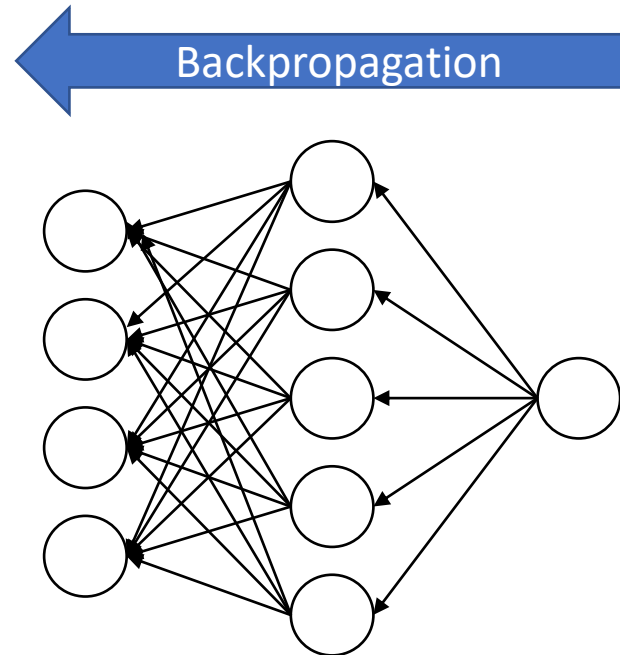
2. Associa cada uma das ativações da camada anterior à pesos e computa a ativação da camada de saída.



# *O Multilayer Perceptron: Retro-Propagação*

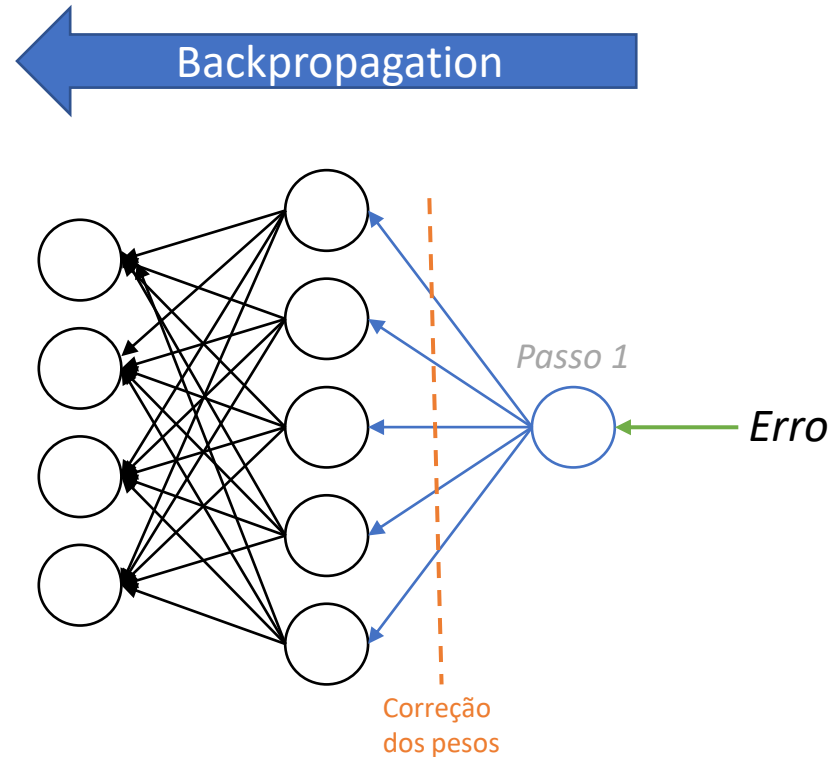
Tem como objetivo corrigir os erros cometidos pela rede através do ajuste dos pesos intermediários.

O algoritmo Backpropagation propaga o erro verificado nos outputs nos sentido contrário da rede, ajustando cada um de seus pesos.



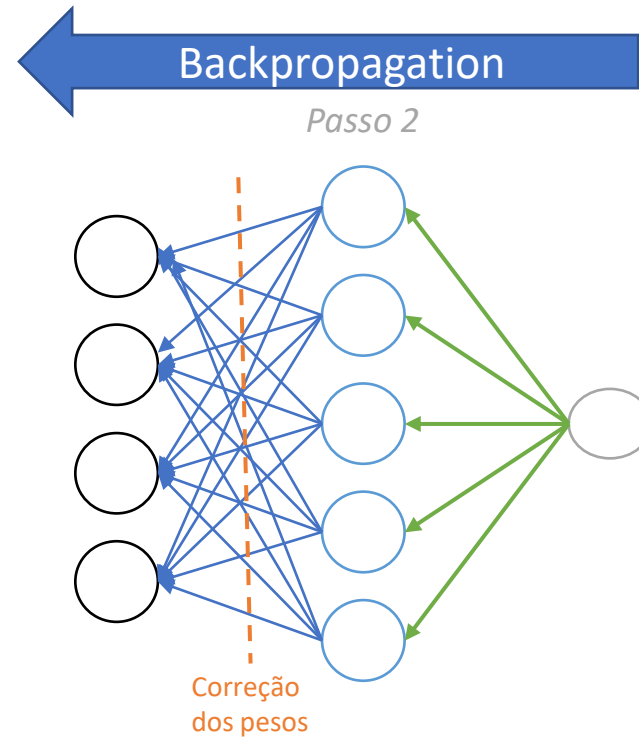
# *O Multilayer Perceptron: Retro-Propagação*

1. Calcula o gradiente em relação a cada um dos pesos conectados diretamente a camada de saída, atualizando-os em seguida.



# *O Multilayer Perceptron: Retro-Propagação*

2. Propaga o erro para a camada anterior à de saída, ajustando seus respectivos pesos.





# Multilayer Perceptron (MLP)

*Formulação Matemática*

Juvenal J. Duarte

# Notação

Variável, quando maiúscula com a seta sobreposta indica vetor/matriz, quando minúscula sem seta indica escalar.

$\vec{W}_k^{[l]}$

O valor sobreposto entre colchetes, representado por  $l$ , indica a camada da rede.

$b_k^{[l]}$

O valor abaixo da variável, representado por  $k$ , faz referência ao neurônio da camada em questão.

# *O Multilayer Perceptron: definições*

Quais as variáveis que temos por neurônio?

$\vec{A}^{[l-1]}$ : entradas vindas dos neurônios da camada anterior. Quando  $l = 0$ , representa os valores de cada um dos atributos de entrada.

$z_k^{[l]}$ : agregação linear do neurônio  $k$  na camada  $l$ .

$a_k^{[l]}$ : saída do neurônio  $k$  na camada  $l$ .

$b_k^{[l]}$ : bias usado no neurônio  $k$ , camada  $l$ .

$\vec{W}_k^{[l]}$ : vetor de pesos para cada uma das saídas camada  $l - 1$ , usado no neurônio  $k$ .



# *O Multilayer Perceptron: definições*

Comparativo com a regressão logística

$$\vec{A}^{[l-1]}: \vec{X}$$

$$z_k^{[l]}: \vec{X} \times \vec{\beta} + \alpha$$

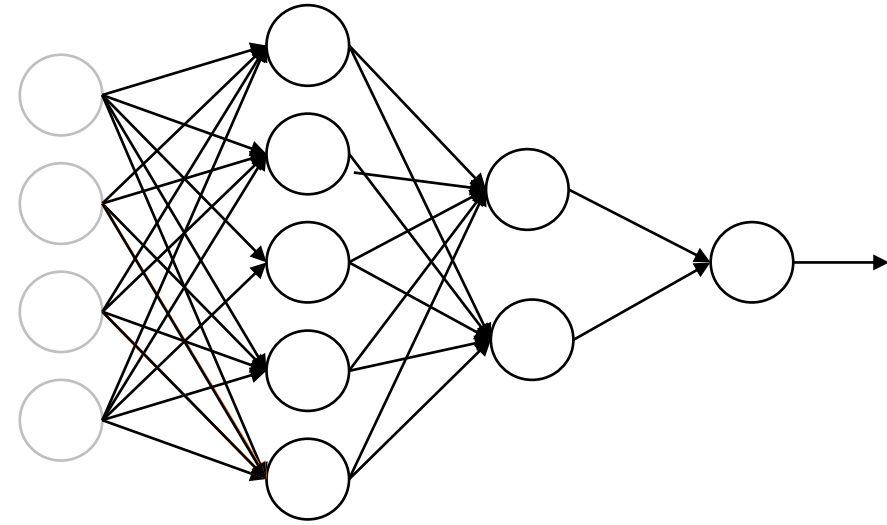
$$a_k^{[l]}: \hat{y} = \text{sigmoid}(z_n^{[l]}) = \text{sigmoid}(\vec{X} \times \vec{\beta} + \alpha)$$

$$\overrightarrow{W}_k^{[l]}: \vec{\beta}$$

$$b_k^{[l]}: \alpha$$

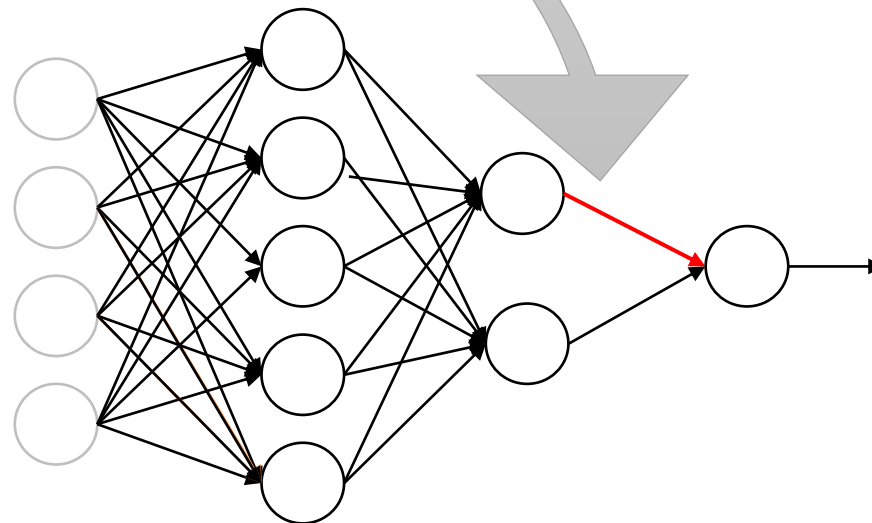
## Notação: exercício

Onde se encontra  $a_1^{[2]}$  no diagrama ao lado?



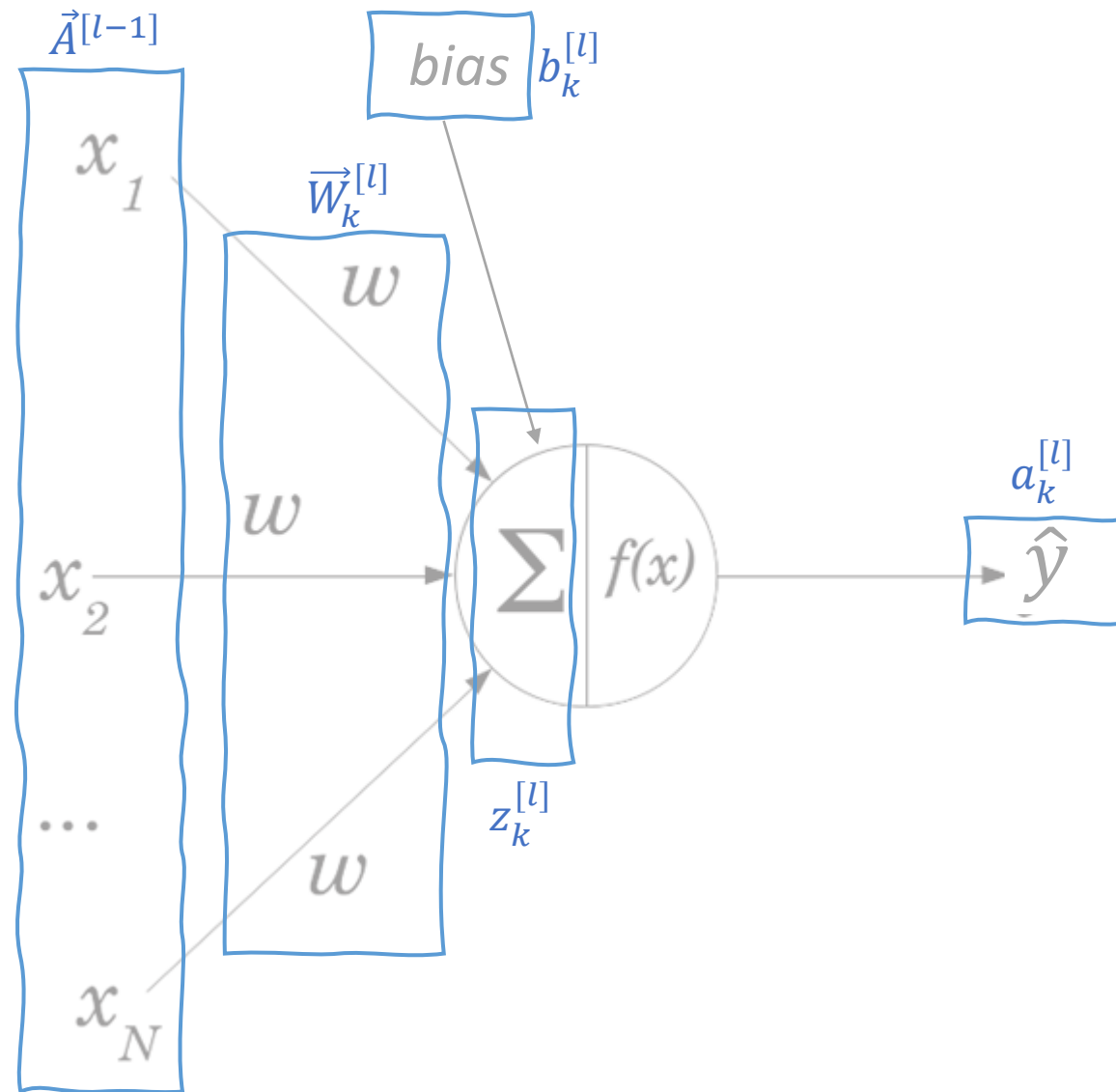
## Notação: exercício

Onde se encontra  $a_1^{[2]}$  no diagrama ao lado?



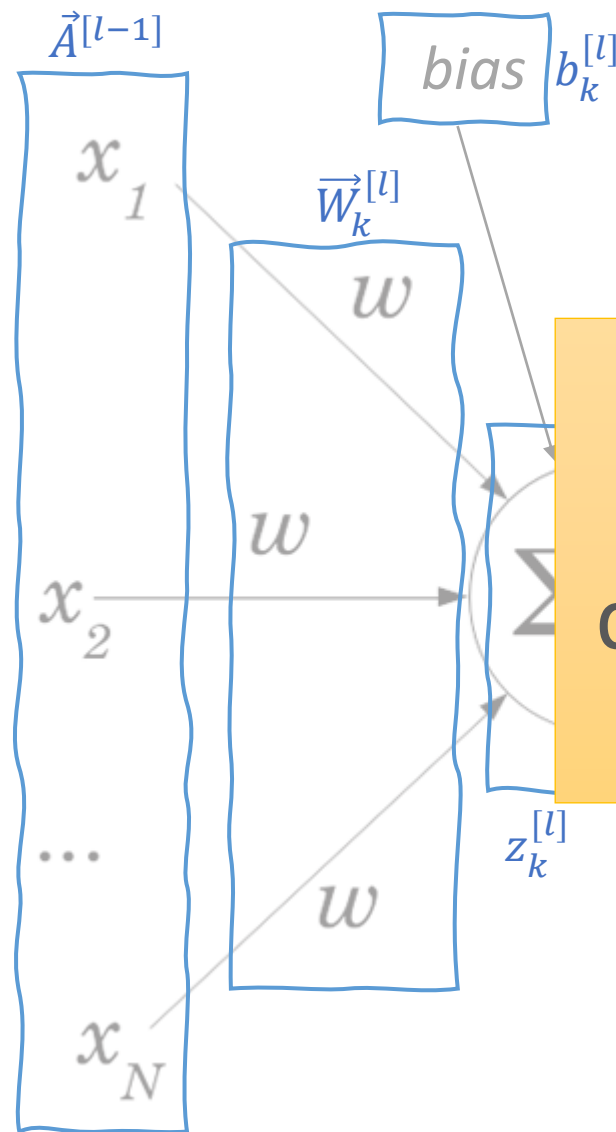
# Definições:

Um único neurônio



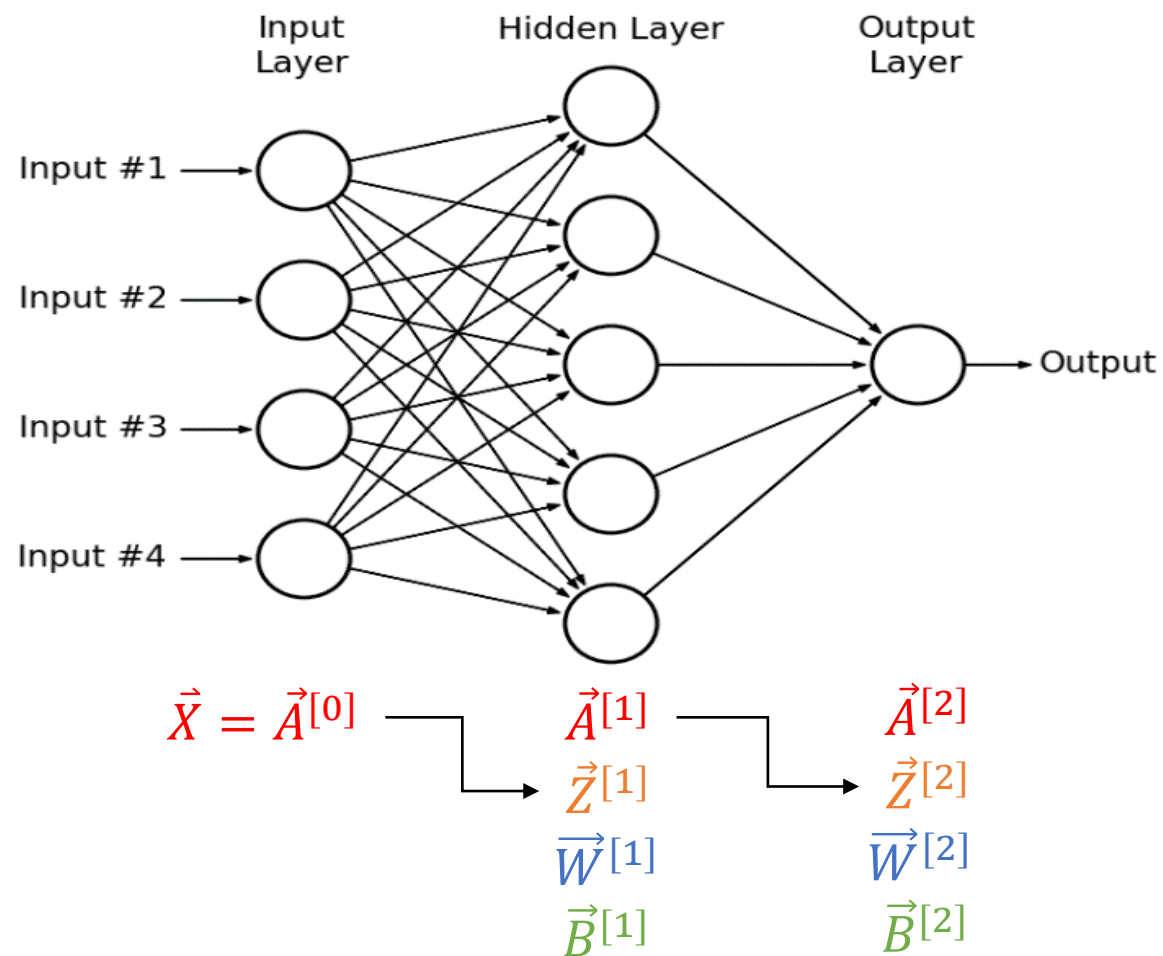
# Definições:

Um único neurônio

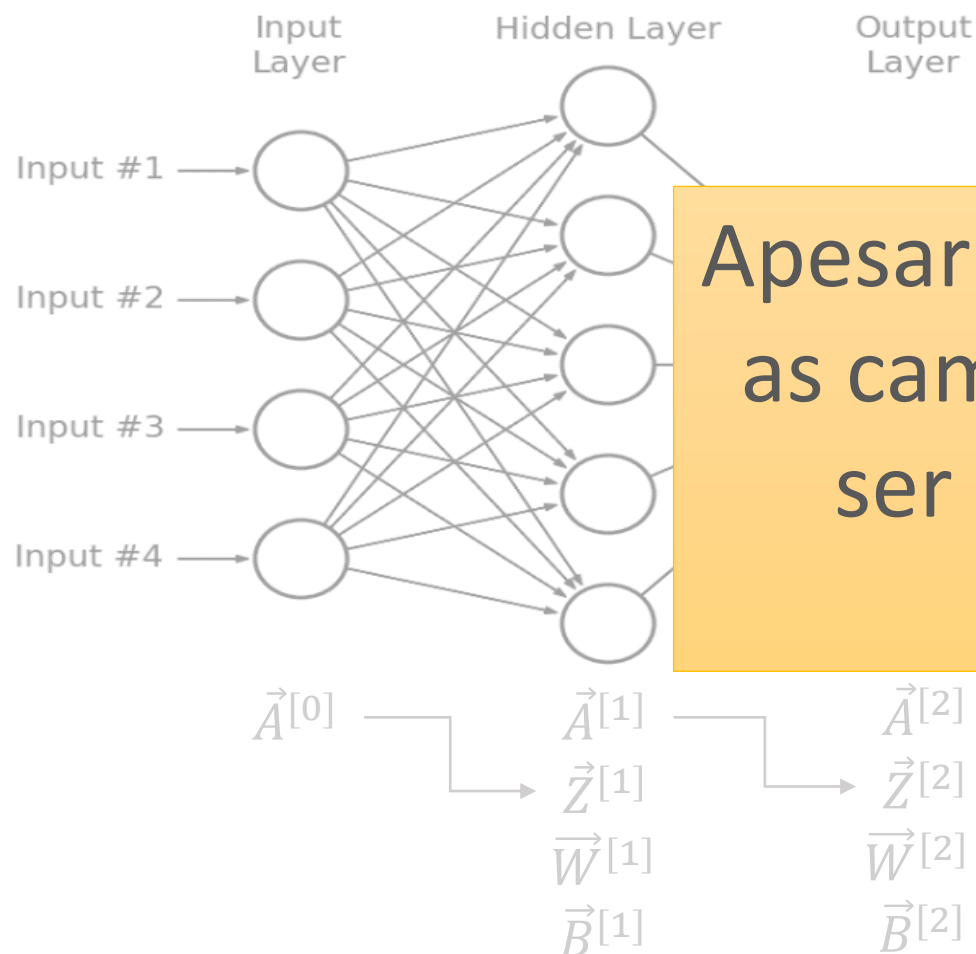


Pelas vantagens do cálculo vetorial, as variáveis são definidas por camadas e não por neurônio.

# *O Multilayer Perceptron: feed-forward*



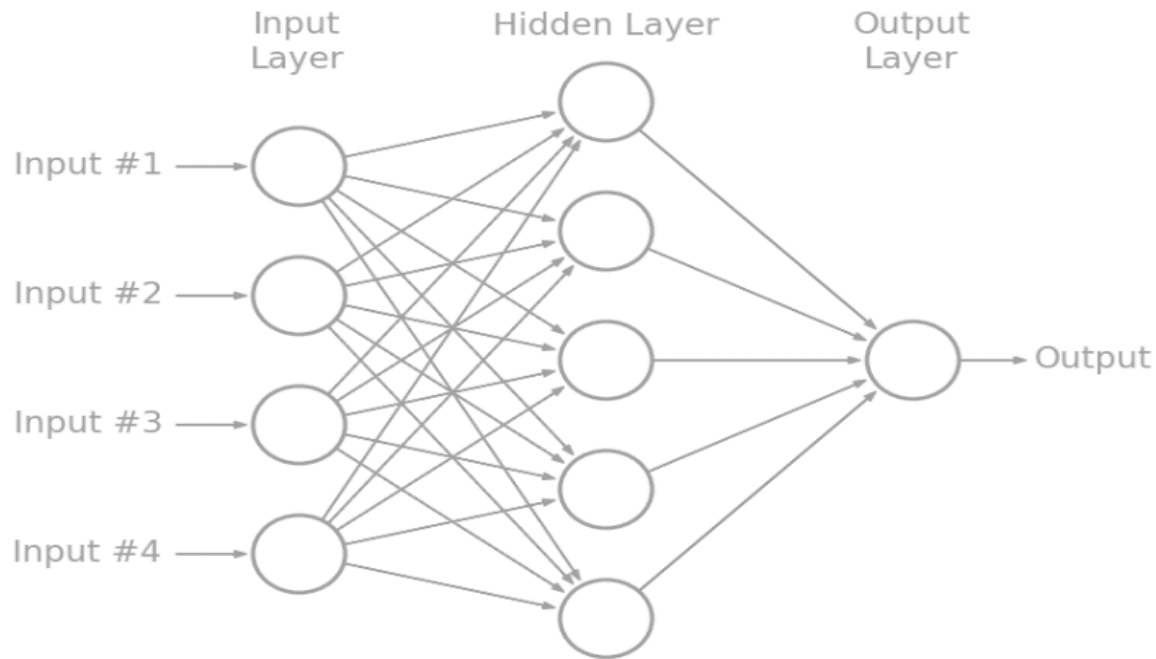
# *O Multilayer Perceptron: feed-forward*



Apesar dos cálculos vetoriais, as camadas ainda precisam ser avaliadas de forma iterativa.

# Feed-forward:

*Exemplo de notação*



Arquitetura = [4, 5, 1]

$L = 3$

$n^0 = 4; n^1 = 5; n^2 = 1$

Linhas de dados

$m = 200$



# Feed-forward: Cálculo Vetorial

$$\vec{A}^{[0]} = \#(m, n^0)$$

$$\begin{aligned}\vec{A}^{[1]} &= \text{sigmoid}(\vec{Z}^{[1]}) \\ \vec{Z}^{[1]} &= \vec{A}^{[0]} \times \vec{W}^{[1]} + \vec{B}^{[1]} \\ \vec{W}^{[1]} &= \#(n^0, n^1) \\ \vec{B}^{[1]} &= \#(1, n^1)\end{aligned}$$

$$\begin{aligned}\vec{A}^{[2]} &= \text{sigmoid}(\vec{Z}^{[2]}) \\ \vec{Z}^{[2]} &= \vec{A}^{[1]} \times \vec{W}^{[2]} + \vec{B}^{[2]} \\ \vec{W}^{[2]} &= \#(n^1, n^2) \\ \vec{B}^{[2]} &= \#(1, n^2)\end{aligned}$$

Entrada

Camada intermediária

Saída

# Feed-forward: Cálculo Vetorial

$$\vec{A}^{[0]} = \#(m, n^0)$$

$$\vec{A}^{[1]} = \text{sigmoid}(\vec{Z}^{[1]})$$

$$\vec{Z}^{[1]} = \vec{A}^{[0]} \times \vec{W}^{[1]} + \vec{B}^{[1]}$$

$$\vec{W}^{[1]} = \#(n^0, n^1)$$

$$\vec{B}^{[1]} = \#(1, n^1)$$

$$\vec{A}^{[2]} = \text{sigmoid}(\vec{Z}^{[2]})$$

$$\vec{Z}^{[2]} = \vec{A}^{[1]} \times \vec{W}^{[2]} + \vec{B}^{[2]}$$

$$\vec{W}^{[2]} = \#(n^1, n^2)$$

Regras para dimensões de matrizes:

$$W^{[l]}.shape = (\#k^{[l-1]}, \#k^{[l]})$$

$$B^{[l]}.shape = (1, \#k^{[l]})$$

$$A^{[l-1]}.shape = (m, \#k^{[l-1]})$$

$$Z^{[l]}.shape = (m, \#k^{[l]})$$

$$A^{[l]}.shape = (m, \#k^{[l]})$$

Entrada

Camada intermediária

Saída

# Feed-forward: Cálculo Vetorial

$$\vec{A}^{[0]} = \#(m, n^0)$$

$$\vec{A}^{[1]} = \text{sigmoid}(\vec{Z}^{[1]})$$

$$\vec{Z}^{[1]} = \vec{A}^{[0]} \times \vec{W}^{[1]} + \vec{B}^{[1]}$$

$$\vec{W}^{[1]} = \#(n^0, n^1)$$

$$\vec{B}^{[1]} = \#(1, n^1)$$

$$\vec{A}^{[2]} = \text{sigmoid}(\vec{Z}^{[2]})$$

$$\vec{Z}^{[2]} = \vec{A}^{[1]} \times \vec{W}^{[2]} + \vec{B}^{[2]}$$

$$\vec{W}^{[2]} = \#(n^1, n^2)$$

$$\vec{B}^{[2]} = \#(1, n^2)$$

Fórmula geral:

$$\vec{A}^{[l]} = \text{sigmoid}(\vec{Z}^{[l]})$$

$$\vec{Z}^{[l]} = \vec{A}^{[l-1]} \times \vec{W}^{[l]} + \vec{B}^{[l]}$$

Entrada

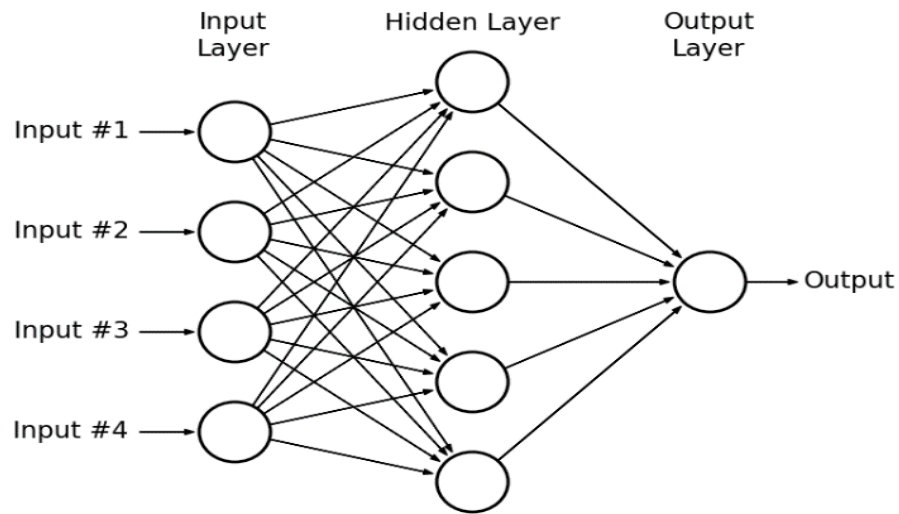
Camada intermediária

Saída

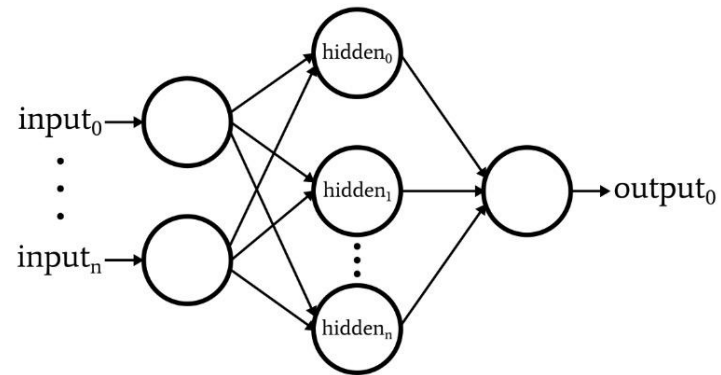
# Exercício: dimensões de dados

Calcule o valor esperado para as dimensões de cada uma das variáveis, para cada um dos exemplos, para cada uma das camadas:

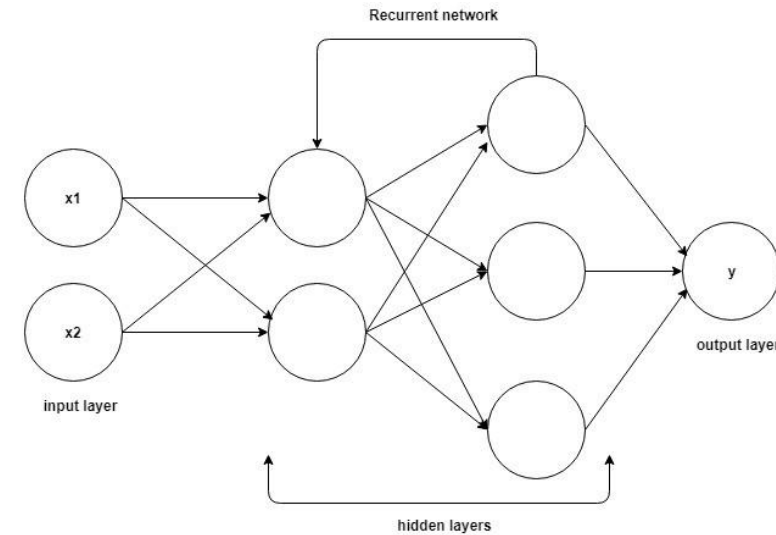
$W^{[l]}$ .shape = ?  $B^{[l]}$ .shape = ?  $A^{[l-1]}$ .shape = ?  $Z^{[l]}$ .shape = ?  $A^{[l]}$ .shape = ?



Exemplo 1



Exemplo 2



Exemplo 3



# Multilayer Perceptron (MLP)

*Funções de ativação*

Juvenal J. Duarte

# Funções de ativação

Porque usar uma função de ativação?

Exemplo 1: sem ativação

$$f(x) = 3x + 1$$

$$g(x) = x + 5$$

$$h(x) = f(x) + g(x) = 4x + 6$$

Exemplo 2: com ativação

$$f(x) = \frac{1}{1 + e^{-(3x+1)}}$$

$$g(x) = \frac{1}{1 + e^{-(x+5)}}$$

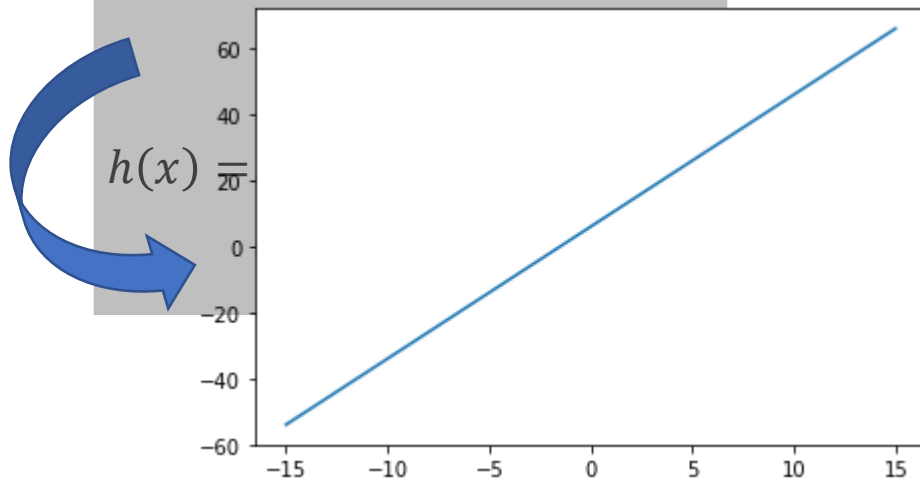
$$h(x) = f(x) + g(x) = \frac{1}{1 + e^{-(3x+1)}} + \frac{1}{1 + e^{-(x+5)}}$$

# Funções de ativação

Porque usar uma função de ativação?

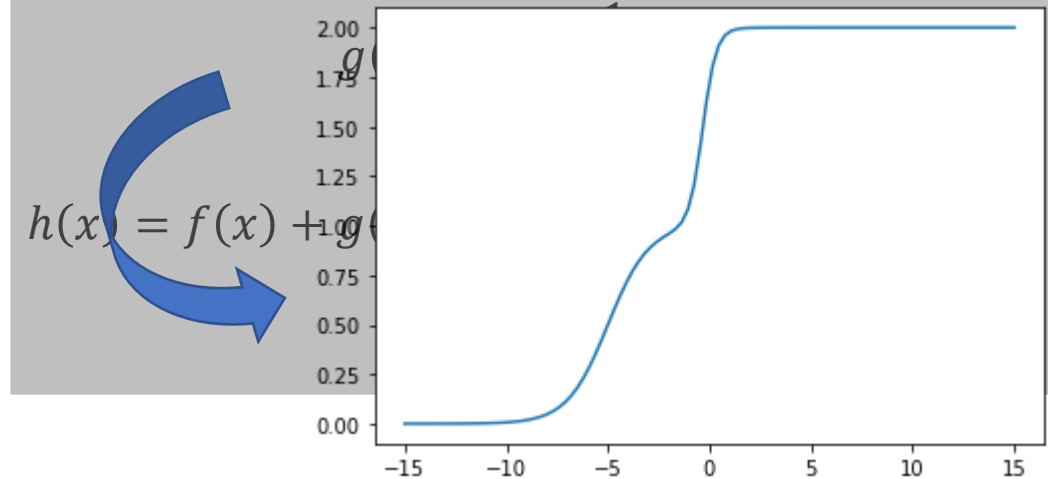
Exemplo 1: sem ativação

$$f(x) = 3x + 1$$








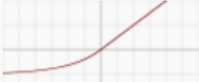



Exemplo 2: com ativação

$$f(x) = \frac{1}{1 + e^{-(3x+1)}}$$



# Funções de ativação

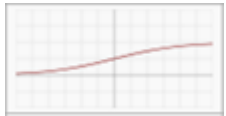
Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$



# Funções de ativação

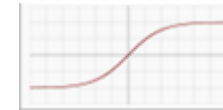
## Problemas comuns:

Sigmoid / Função logística:



- Não centrada em zero
- Aprendizado lento quando  $|X|$  é alto
- Computacionalmente cara

Tangente Hiperbólica



Rectified Linear Unit (ReLU)

- Neurônios desativados quando entradas  $< 0$



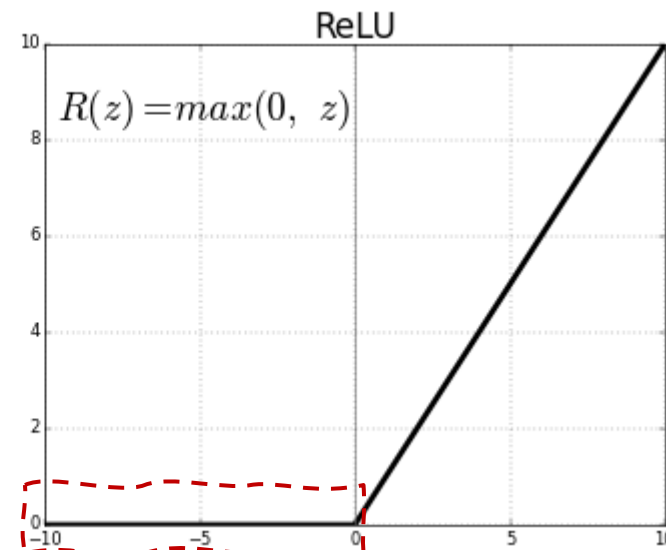
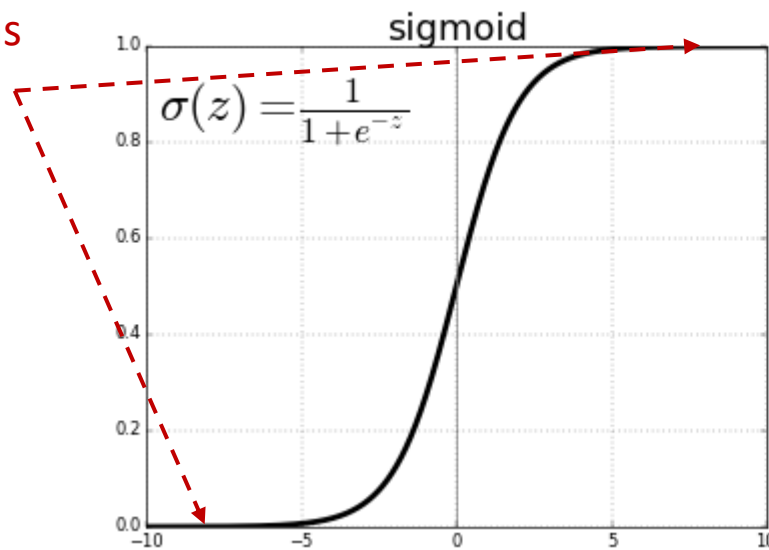
Leaky/Parametric Rectified Linear Unit (ReLU)



# Funções de ativação:

## Sigmoid Vs ReLU

A derivada da função nestas regiões é praticamente zero, logo as atualizações dos pesos pelo gradiente é praticamente nenhuma.

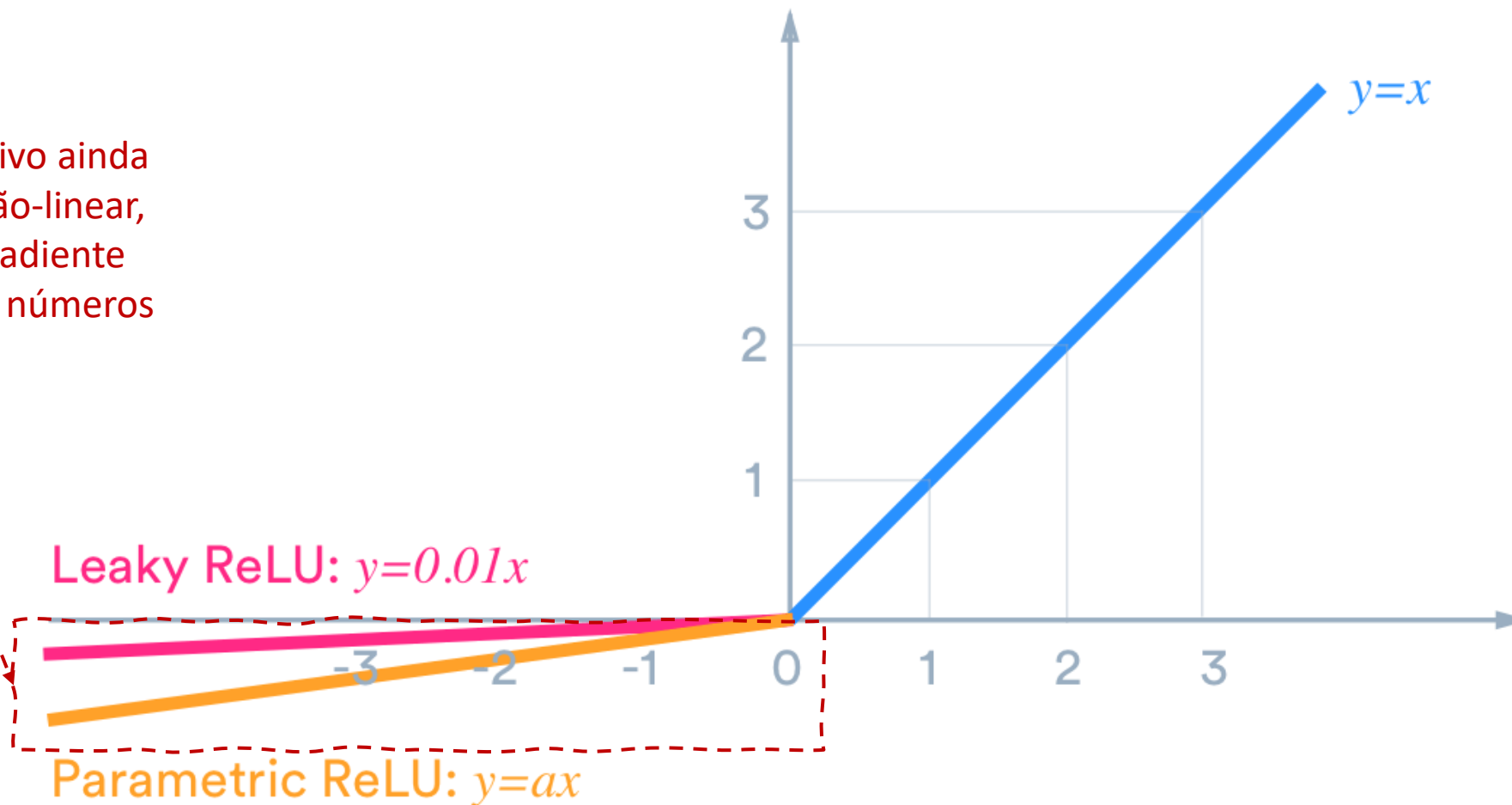


Zona da morte!!!  
Se o gradiente chegar nesta região os pesos nunca mais serão atualizados!

# Funções de ativação:

## Melhorias na função ReLU

O fator multiplicativo ainda mantém função não-linear, mas evita que o gradiente se torne zero para números negativos.



# Feed-forward: Implementação

4  
4

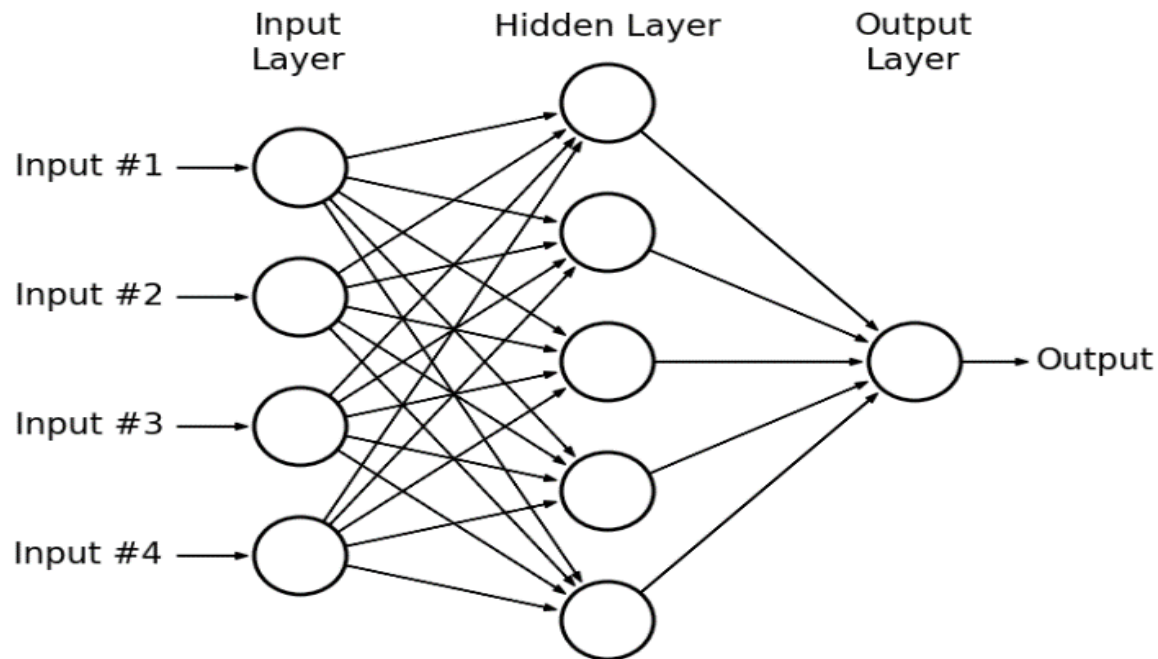


# Backpropagation

Juvenal J. Duarte

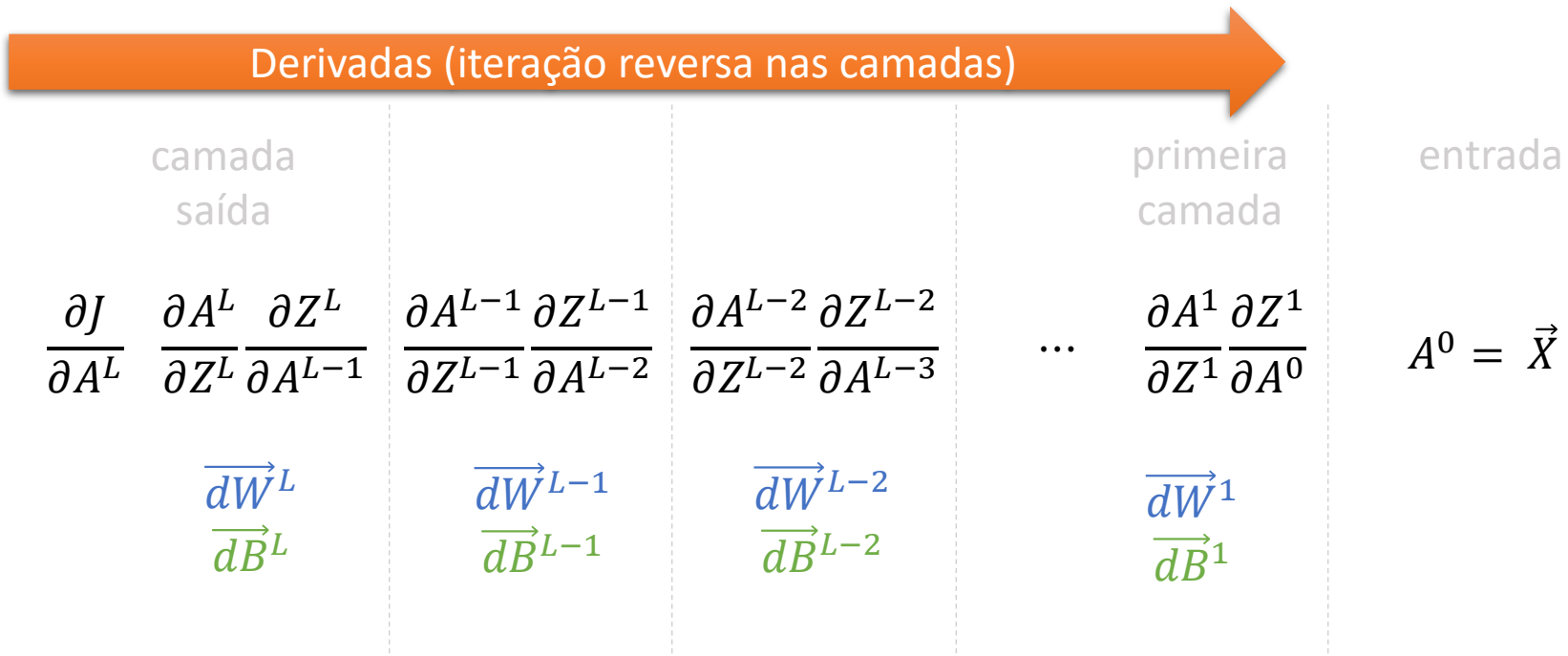


# Backpropagation



Se analisada como um todo, a saída da rede neural artificial nada mais é que uma função composta, recursiva pelas camadas da rede. Se chamamos a última camada simplesmente de  $f_1^{[L]}(x)$ , esta função é uma composição de todas as funções da camada anterior  $f_1^{[L-1]}(x) \dots f_k^{[L-1]}(x)$ , que por sua vez dependem são composições das funções da camada anterior  $f_1^{[L-2]}(x) \dots f_k^{[L-2]}(x)$ . A recursão segue até a camada de entrada, onde os valores são pré estabelecidos e não calculados.

# Backpropagation



# Backpropagation

Derivadas (iteração reversa nas camadas)

camada saída			
$\frac{\partial J}{\partial A^L}$	$\frac{\partial A^L}{\partial Z^L} \frac{\partial Z^L}{\partial A^{L-1}}$	$\frac{\partial A^{L-1}}{\partial Z^{L-1}} \frac{\partial Z^{L-1}}{\partial A^{L-2}}$	$\frac{\partial A^{L-2}}{\partial Z^{L-2}}$
$\overrightarrow{dW}^L$	$\overrightarrow{dW}^{L-1}$	$\overrightarrow{dW}^{L-2}$	
$\overrightarrow{dB}^L$	$\overrightarrow{dB}^{L-1}$	$\overrightarrow{dB}^{L-2}$	

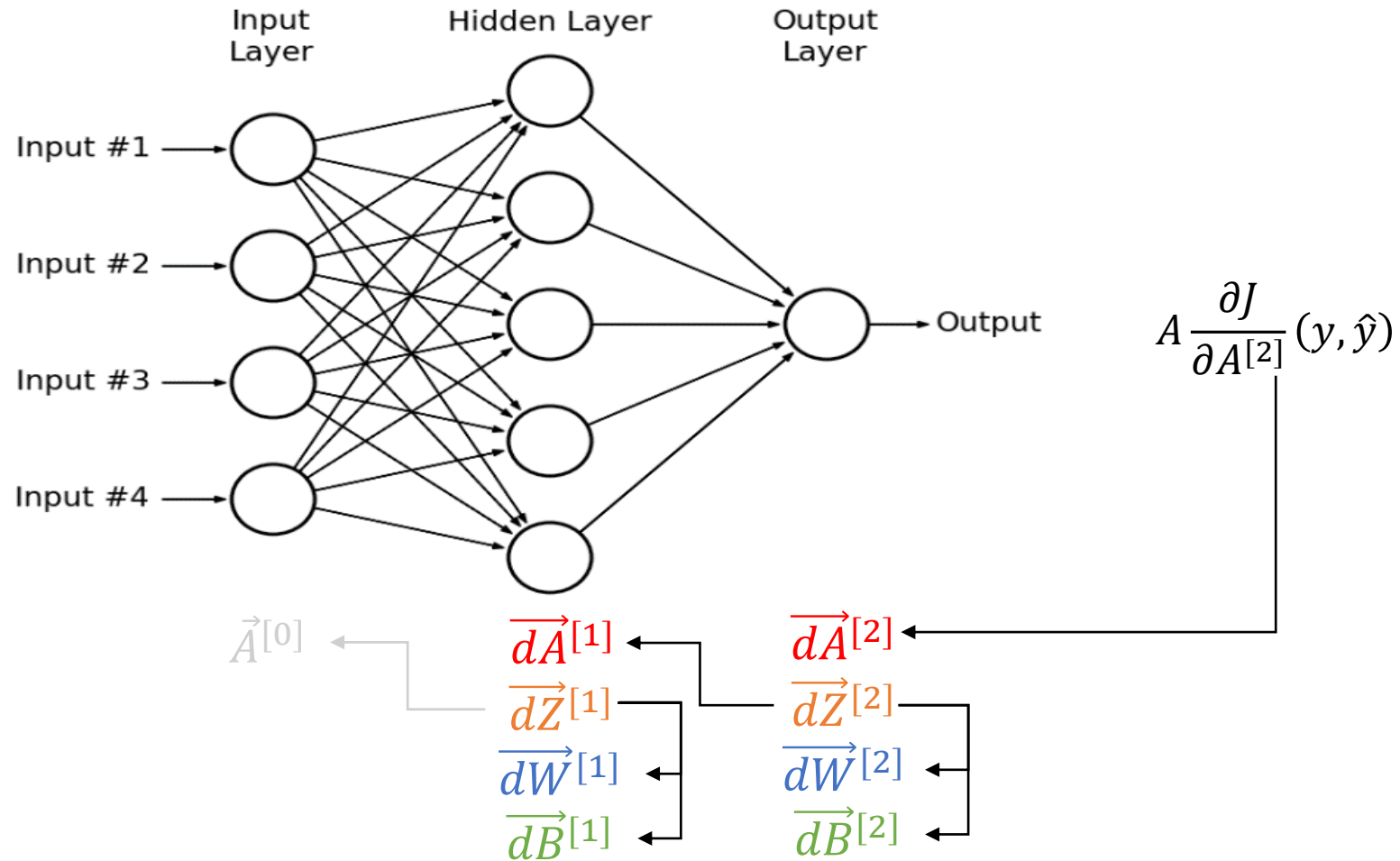
Resumindo:

- A perda (loss) é calculada a partir da predição da última camada,  $A^L$ .
- $A^L$  é calculado aplicando a função de ativação sobre  $Z^L$ .
- $Z^L$  é calculado pela soma ponderada de  $A^{L-1}$  e  $W^L$  mais o bias  $B^L$
- $A^{L-1}$  é calculado aplicando a função de ativação sobre  $Z^{L-1}$ .
- $Z^{L-1}$  é calculado pela soma ponderada de  $A^{L-2}$  e  $W^{L-1}$  mais o bias  $B^{L-1}$
- ...

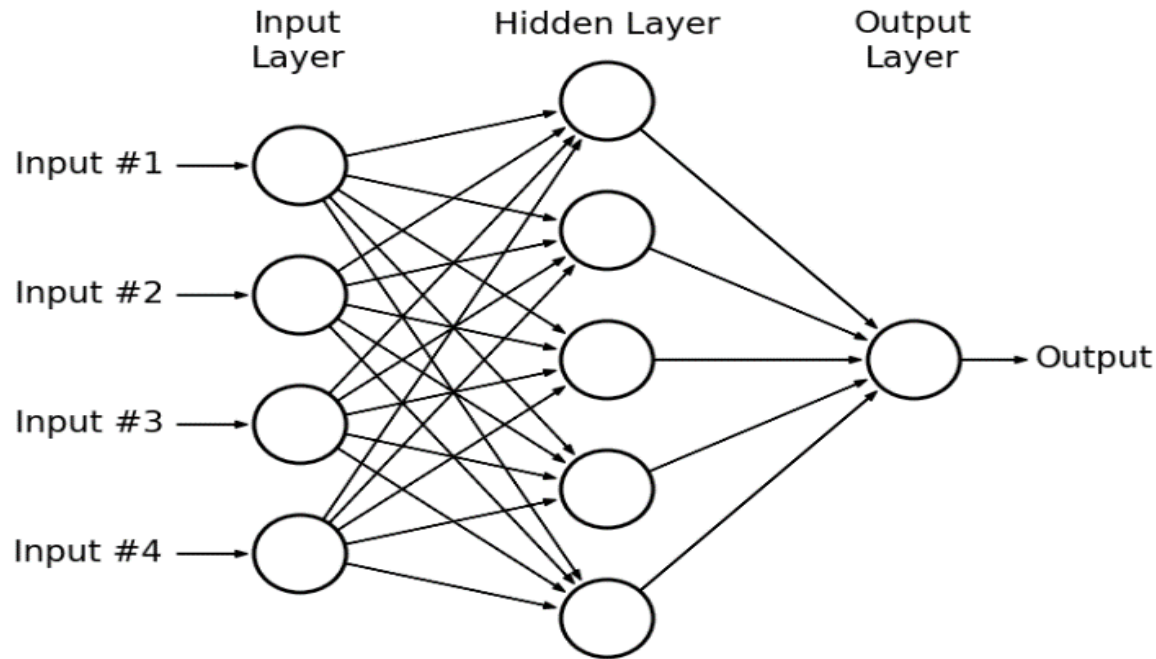
**Obtendo recursivamente as derivadas parciais através da decomposição da função é possível saber quanto cada componente contribuiu para o erro auferido!!!**



# Backpropagation



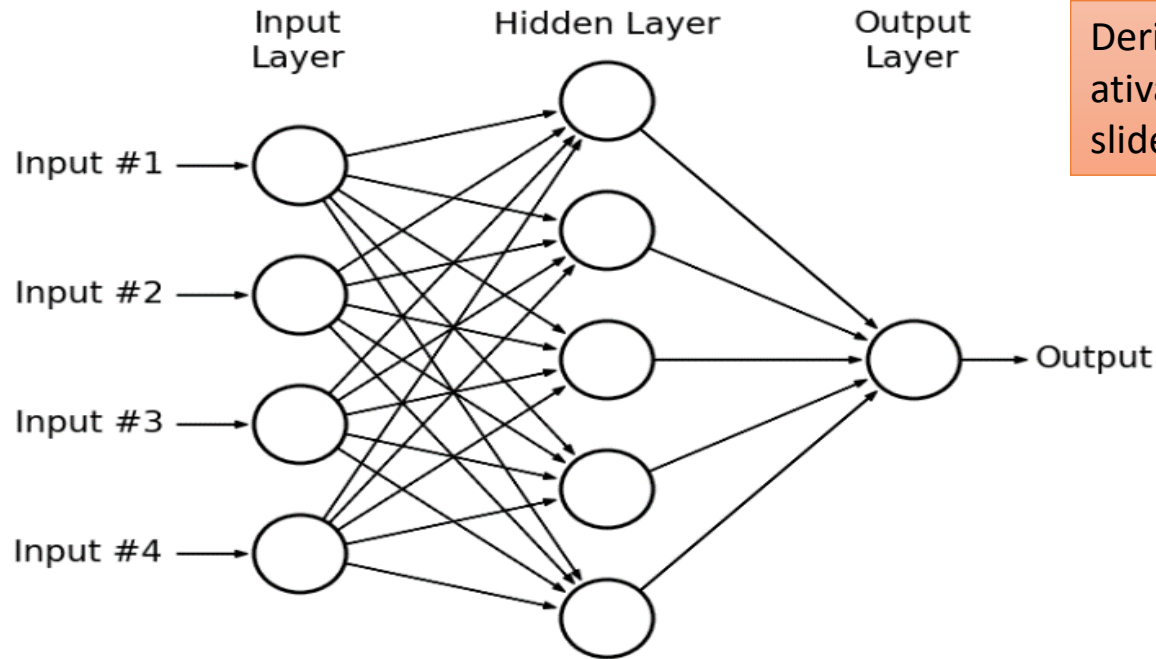
# Backpropagation



$$A \frac{\partial J}{\partial A^{[2]}}(y, \hat{y}) = -\frac{y}{\hat{y}} + \frac{(1-y)}{(1-\hat{y})}$$

$$\begin{aligned} \vec{A}^{[0]} &\leftarrow \begin{cases} \vec{dA}^{[1]} = \vec{dZ}^{[2]} \times \vec{W}^{[2].T} \\ \vec{dZ}^{[1]} \end{cases} \leftarrow \begin{cases} \vec{dA}^{[2]} \\ \vec{dZ}^{[2]} = \vec{dA}^{[2]} \times g'(\vec{Z}^{[2]}) \\ \vec{dW}^{[2]} = \frac{1}{m} \times \vec{A}^{[1].T} \times \vec{dZ}^{[2]} \\ \vec{dB}^{[2]} = \frac{1}{m} \times np.sum(\vec{dZ}^{[2]}, axis = 0, keepdims = True) \end{cases} \\ &\leftarrow \begin{cases} \vec{dZ}^{[1]} \\ \vec{dW}^{[1]} \\ \vec{dB}^{[1]} \end{cases} \end{aligned}$$

# Backpropagation



Derivada da função de ativação em relação a Z, ver slide 16.

$$A \frac{\partial J}{\partial A^{[2]}}(y, \hat{y}) = -\frac{y}{\hat{y}} + \frac{(1-y)}{(1-\hat{y})}$$

$$\begin{aligned} \vec{A}^{[0]} &\leftarrow \begin{cases} \vec{dA}^{[1]} = \vec{dZ}^{[2]} \times \vec{W}^{[2].T} \\ \vec{dZ}^{[1]} \\ \vec{dW}^{[1]} \\ \vec{dB}^{[1]} \end{cases} \\ \vec{dA}^{[2]} &\leftarrow \begin{cases} \vec{dZ}^{[2]} = \vec{dA}^{[2]} \times g'(\vec{Z}^{[2]}) \\ \vec{dW}^{[2]} = \frac{1}{n} \times \vec{A}^{[1].T} \times \vec{dZ}^{[2]} \\ \vec{dB}^{[2]} = \frac{1}{m} \times np.sum(\vec{dZ}^{[2]}, axis = 0, keepdims = True) \end{cases} \end{aligned}$$

# Backpropagation

Tantas derivadas para...

$$\vec{W}^{[l]} = \vec{W}^{[l]} - learning_{rate} * \overrightarrow{dW}^{[l]}$$

$$\vec{B}^{[l]} = B^{[l]} - learning_{rate} * \overrightarrow{dB}^{[l]}$$

...atualizar cada um dos pesos da rede, em cada camada.

# Backpropagation: Implementação