Redes Neurais

Prof. Danilo Silva

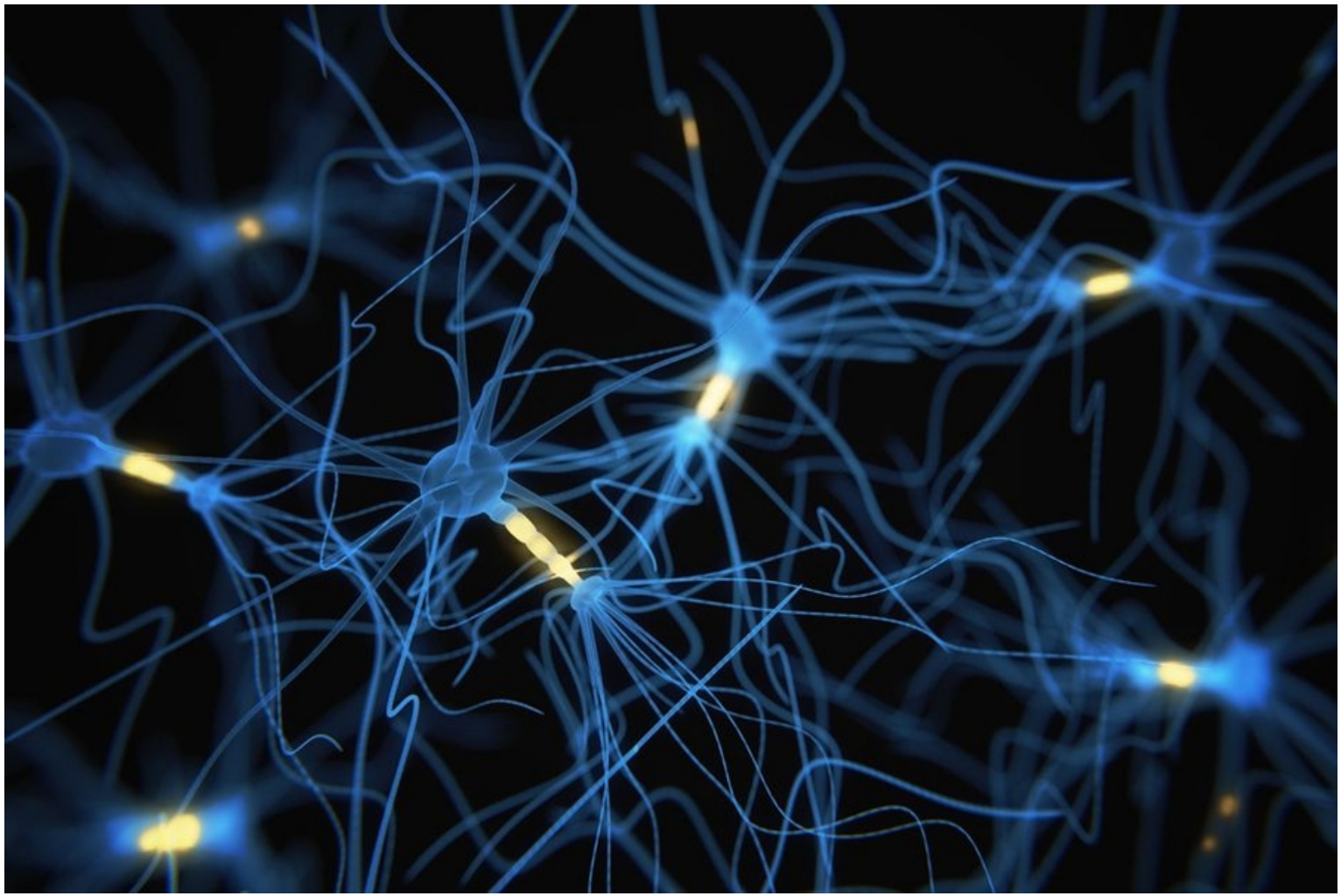
EEL7514/EEL7513 - Tópico Avançado em Processamento de Sinais
EEL410250 - Aprendizado de Máquina

EEL / CTC / UFSC

Introdução

Inspiração Biológica

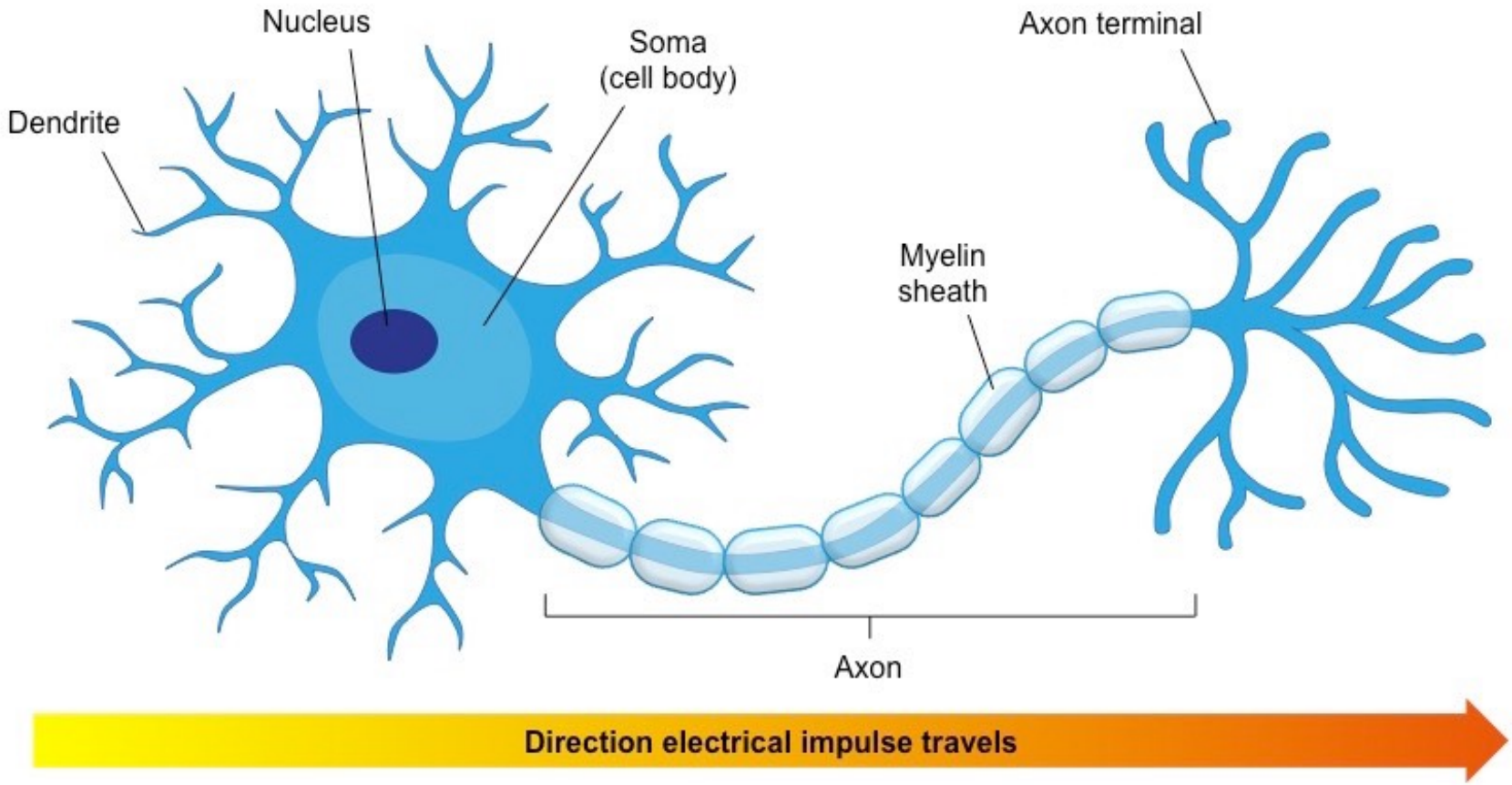
- Redes neurais (artificiais) são inspiradas nas redes neurais biológicas



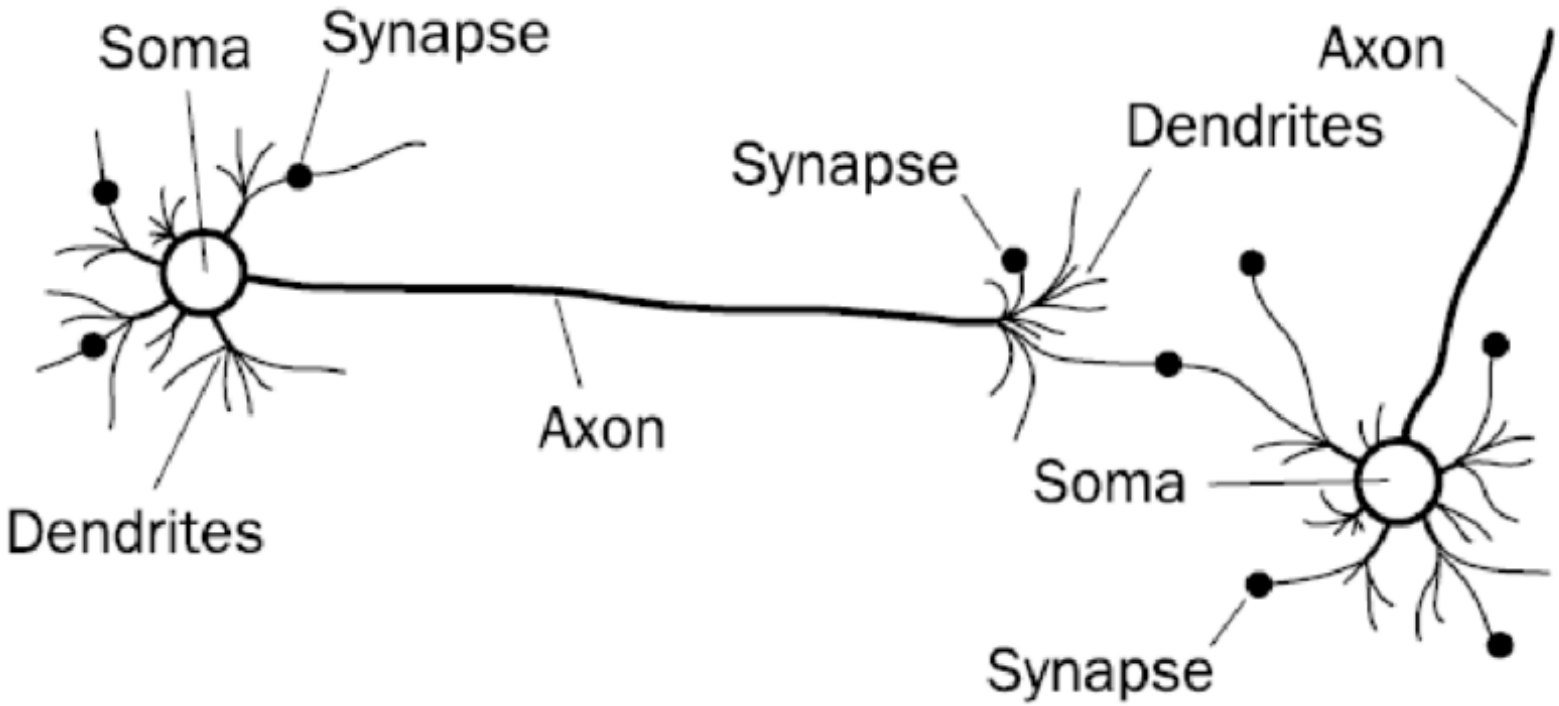
Inspiração Biológica

- Apesar da inspiração, os modelos de maior sucesso na prática diferem dos modelos biológicos em muitos aspectos importantes

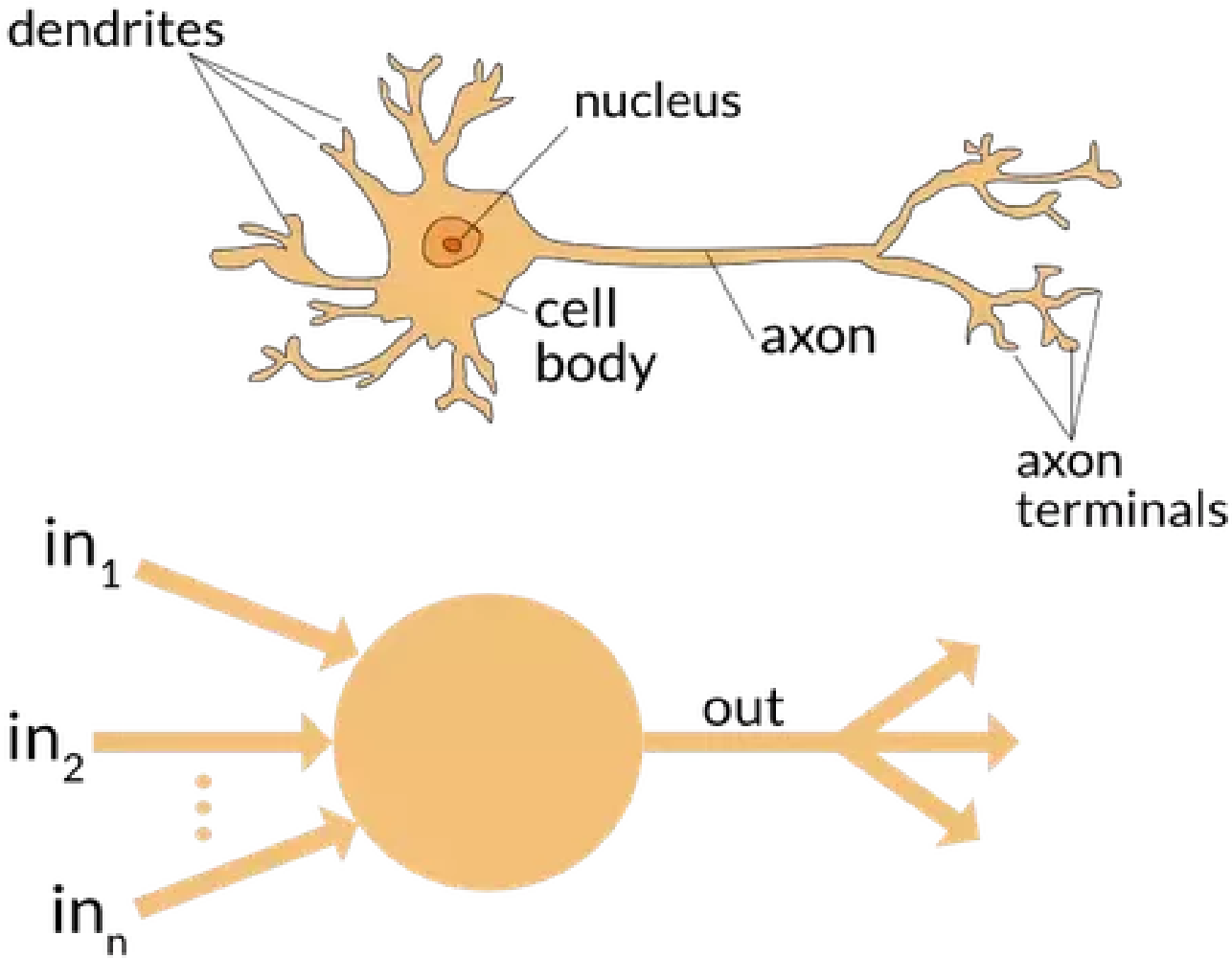
Neurônio Biológico



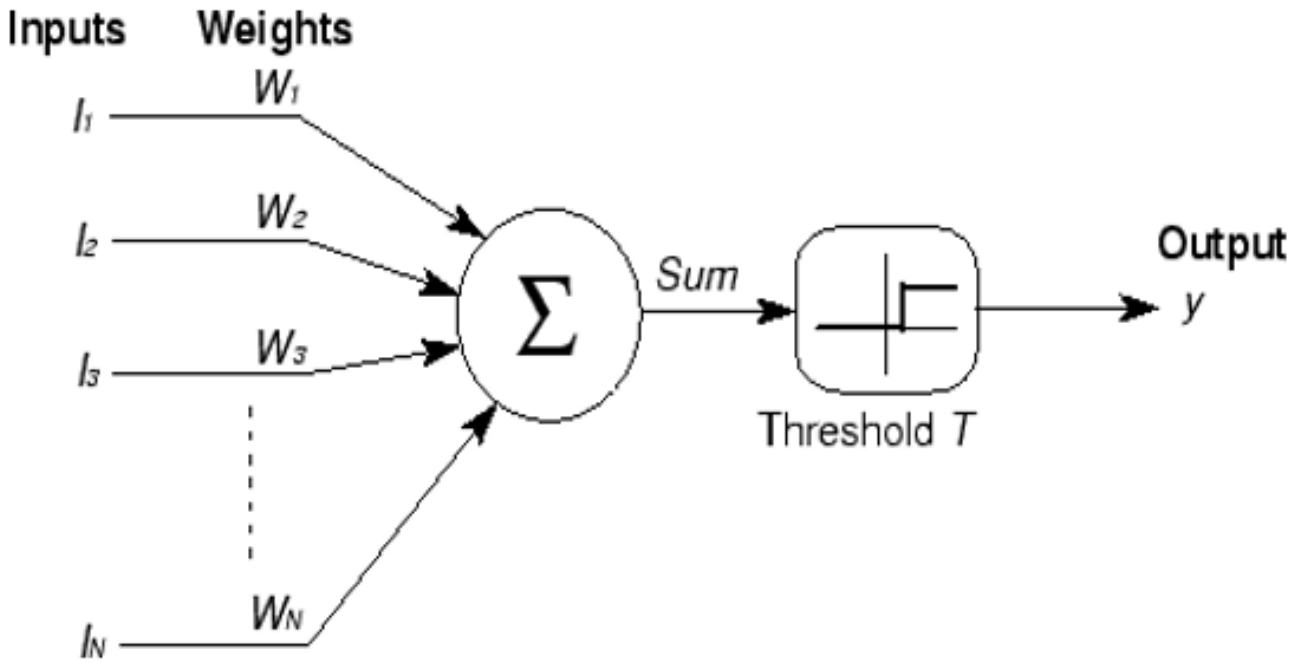
Neurônio Biológico



Neurônio Biológico



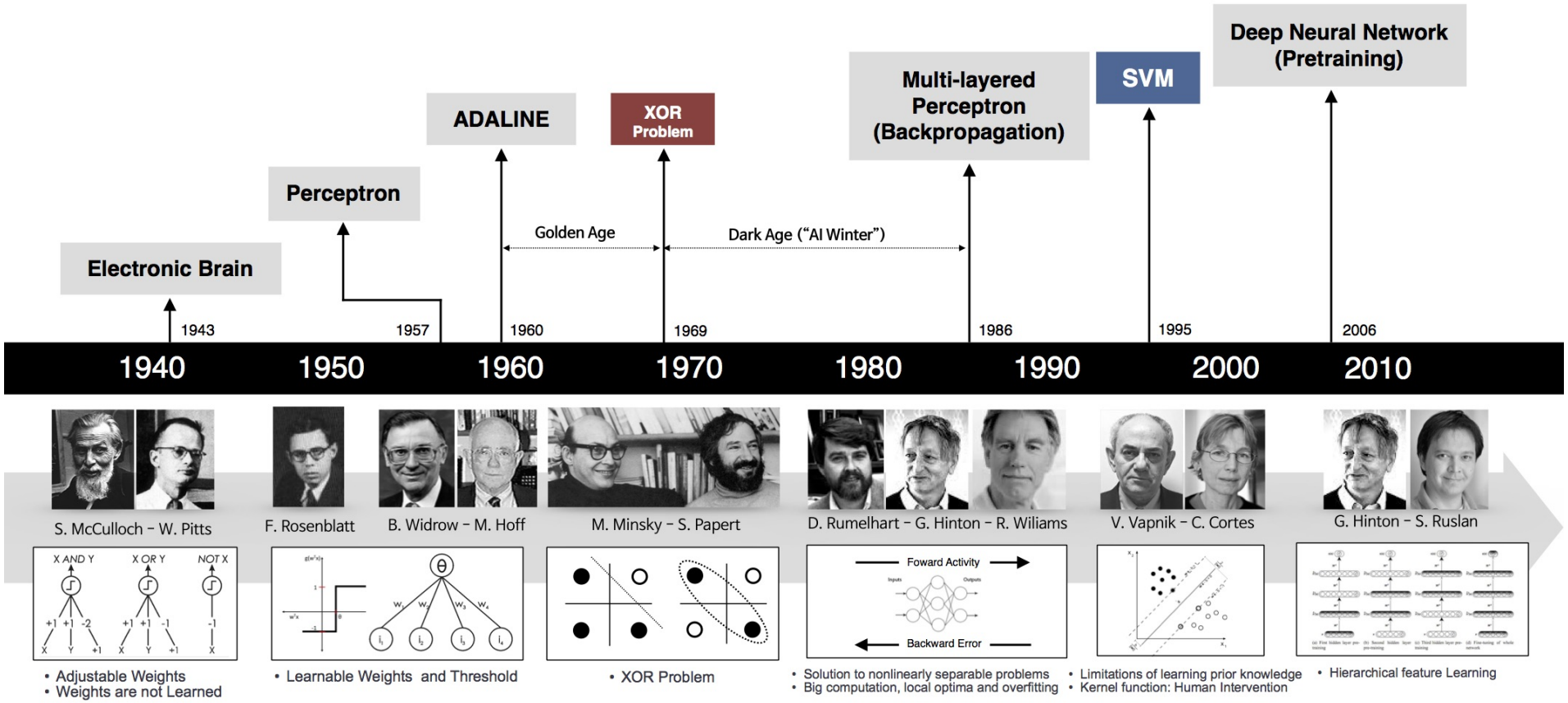
Modelo matemático de McCulloch & Pitts



Histórico

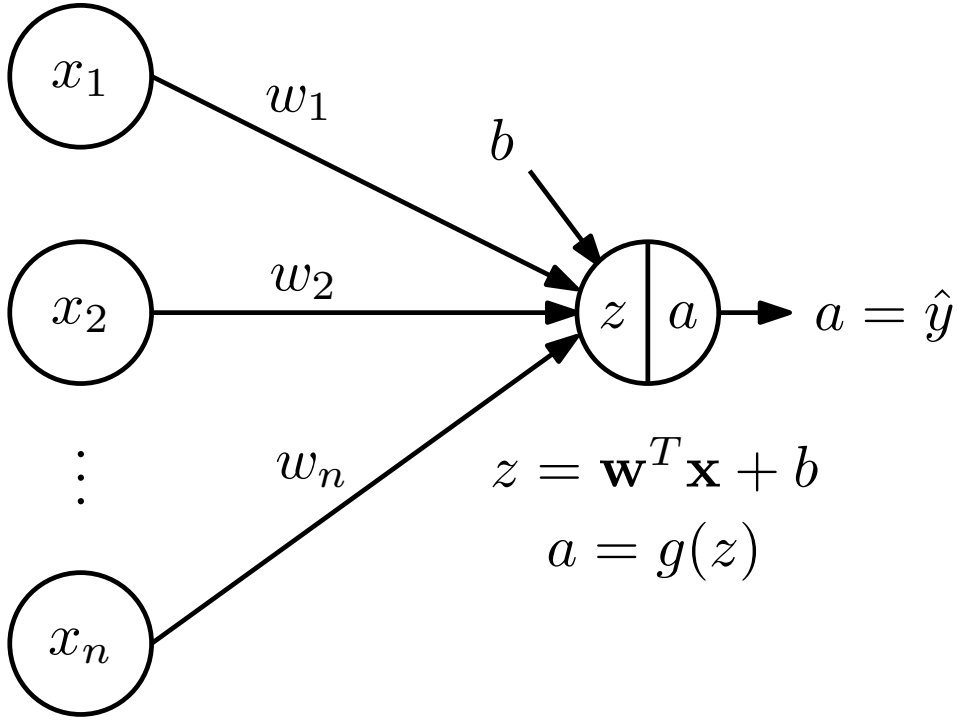
- *1st wave* (1943-1969): cybernetics
  - Modelo matemático de McCulloch & Pitts
  - Algoritmo perceptron (treinamento de um único neurônio)
- *2nd wave* (1986-1995): connectionism & neural networks
  - Rede neural = “multi-layer perceptron”
  - Algoritmo *backpropagation*
  - Aplicação prática: reconhecimento de códigos postais nos EUA
- *3rd wave* (2006-): deep learning
  - Pré-treinamento não-supervisionado
  - Treinamento utilizando GPU (*Graphical Processing Unit*)
  - Vitória na competição ImageNet 2012 com uma rede neural convolucional

Histórico


Deep Neural Network (Pretraining)
SVM
Multi-layered Perceptron (Backpropagation)
XOR Problem
ADALINE
Perceptron
Electronic Brain
Golden Age
Dark Age ("AI Winter")
1943
1957
1960
1969
1986
1995
2006
1940
1950
1960
1970
1980
1990
2000
2010
S. McCulloch - W. Pitts
F. Rosenblatt
B. Widrow - M. Hoff
M. Minsky - S. Papert
D. Rumelhart - G. Hinton - R. Wiliams
V. Vapnik - C. Cortes
G. Hinton - S. Ruslan
X AND Y
X OR Y
NOT X
Foward Activity
Backward Error
• Adjustable Weights
• Weights are not Learned
• Learnable Weights and Threshold
• XOR Problem
• Solution to nonlinearly separable problems
• Big computation, local optima and overfitting
• Limitations of learning prior knowledge
• Kernel function: Human Intervention
• Hierarchical feature Learning

Conceitos Básicos

Modelo Matemático de um Neurônio



- Unidade computacional que realiza duas tarefas:
  - Pondera linearmente as entradas
  - Produz uma saída (ativação) pela aplicação de uma função não-linear (função de ativação)
    - Exemplo: $g(z) = \sigma(z)$ (sigmóide logística)
- Modernamente chamado de unidade para evitar o termo neurônio

Modelo Matemático de um Neurônio

- Modela a função:

$$\hat{y} = a = g(\mathbf{w}^T\mathbf{x} + b) = g(z), \qquad z = \mathbf{w}^T\mathbf{x} + b$$

- Generaliza:
  - Regressão linear:

$$\hat{y} = g(z) = z \qquad \text{(identidade)}$$
$$L(y, \hat{y}) = (y - \hat{y})^2$$

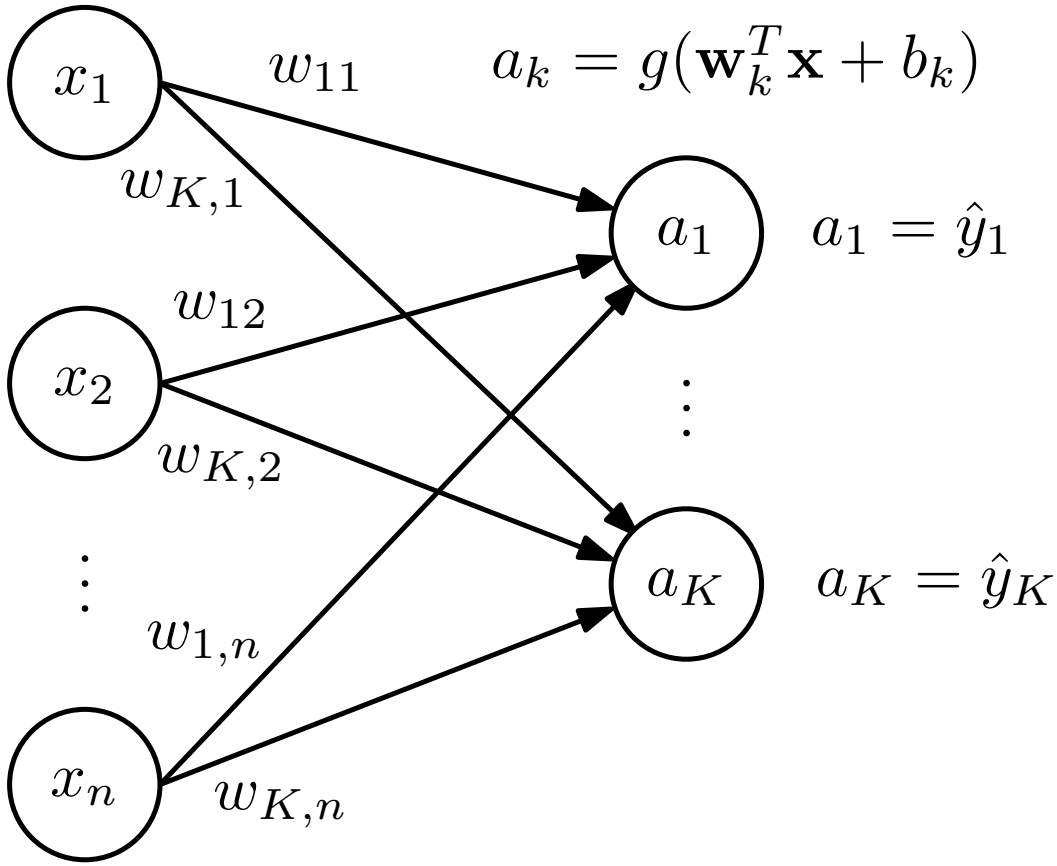
  - Regressão logística:

$$\hat{y} = g(z) = \sigma(z)$$
$$L(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

  - SVM linear:

$$\hat{y} = g(z) = z$$
$$L(y_s, z) = \max\{0, 1 - y_s z\}$$

Rede Neural de Uma Camada (com K Saídas)



Rede Neural de Uma Camada (com K Saídas)

- Corresponde ao caso em que $\mathbf{y} = (y_1, \ldots, y_K)^T \in \mathbb{R}^K$
  - Exemplo: regressão com K saídas
  - Exemplo: classificação multi-classe com codificação 1-de-K
  - Não confundir a notação com o vetor de rótulos de treinamento
- Para cada y_k, temos parâmetros $\mathbf{w}_k = (\mathbf{w}_{k,1}, \ldots, \mathbf{w}_{k,n})^T$ e $b_k \in \mathbb{R}$
- Predição:

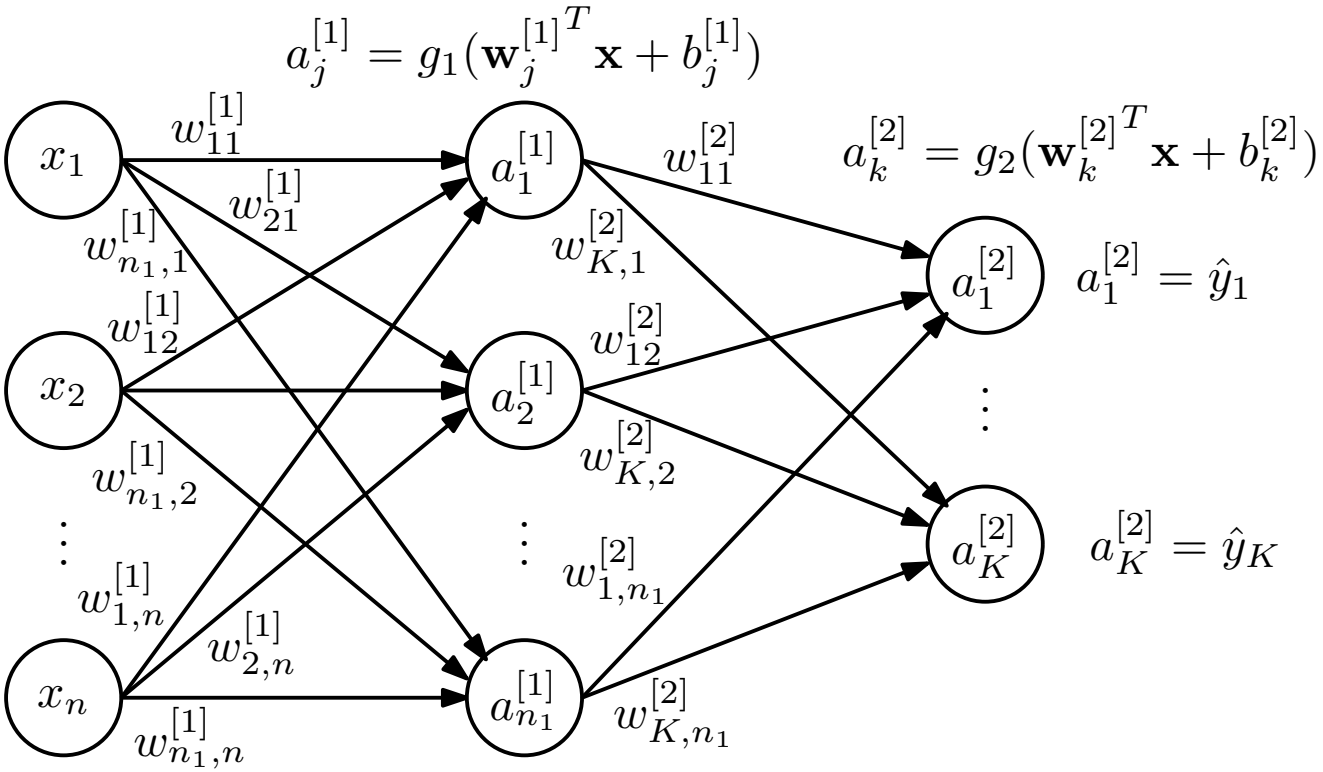
$$\hat{y}_k = g(\mathbf{w}_k^T \mathbf{x} + b_k)$$

$$\hat{\mathbf{y}} = (\hat{y}_1, \ldots, \hat{y}_K)^T = g(\mathbf{W}\mathbf{x} + \mathbf{b})$$

onde

$$\mathbf{W} = \begin{bmatrix} — \ \mathbf{w}_1^T \ — \\ \vdots \\ — \ \mathbf{w}_K^T \ — \end{bmatrix} \in \mathbb{R}^{K \times n} \quad \text{e} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_K \end{bmatrix} \in \mathbb{R}^K$$

Rede Neural de Duas Camadas



Rede Neural de Duas Camadas

- Modela a função:

$$\hat{\mathbf{y}} = f(\mathbf{x}) = \mathbf{a}^{[2]} = g_2(\mathbf{W}^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]})$$
$$\mathbf{a}^{[1]} = g_1(\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]})$$

onde $\mathbf{W}^{[1]}_{n_1 \times n}$, $\mathbf{W}^{[2]}_{K \times n_1}$ e n_1 é o número de unidades ocultas

Redes Neurais *Feedforward* (sem realimentação)

- Em geral, a função $f(\mathbf{x})$ é construída através da composição de L funções vetoriais $f_\ell : \mathbb{R}^{n_{\ell-1}} \to \mathbb{R}^{n_\ell}$, com $n_0 = n$ e $n_L = K$:

$$f(\mathbf{x}) = f_L(f_{L-1}(\cdots f_2(f_1(\mathbf{x})))) = \hat{\mathbf{y}}$$

onde

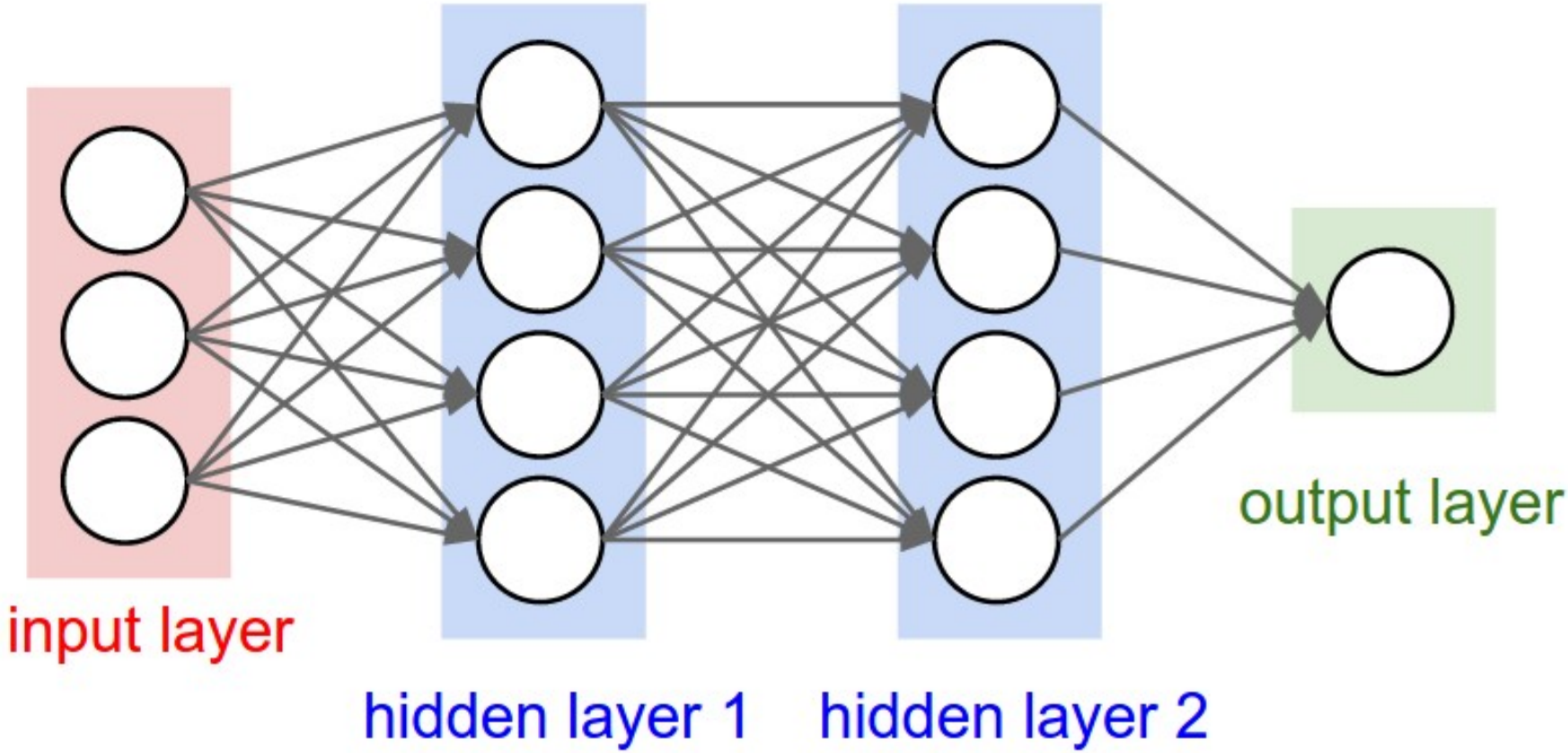
$$f_\ell(\mathbf{a}^{[\ell-1]}) = g_\ell(\mathbf{W}^{[\ell]}\mathbf{a}^{[\ell-1]} + \mathbf{b}^{[\ell]}) = \mathbf{a}^{[\ell]}$$

  - L é o número de camadas ou profundidade da rede
  - n_ℓ é o número de unidades ou largura da camada ℓ
  - $\mathbf{W}^{[\ell]} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$ é matriz de pesos da camada ℓ
  - $\mathbf{b}^{[\ell]} \in \mathbb{R}^{n_\ell}$ é o vetor de *bias* da camada ℓ
  - $\mathbf{a}^{[\ell]} \in \mathbb{R}^{n_\ell}$ é o vetor de ativações da camada ℓ (com $\mathbf{a}^{[0]} = \mathbf{x}$ e $\mathbf{a}^{[L]} = \hat{\mathbf{y}}$)
  - $g_\ell : \mathbb{R} \to \mathbb{R}$ é a função de ativação (não-linear) da camada ℓ, aplicada a cada elemento de um vetor em $\mathbb{R}^{n_\ell}$, i.e.,

$$\mathbf{z} = (z_1, \ldots, z_{n_\ell})^T \quad \Longrightarrow \quad g_\ell(\mathbf{z}) = (g_\ell(z_1), \ldots, g_\ell(z_{n_\ell}))^T$$

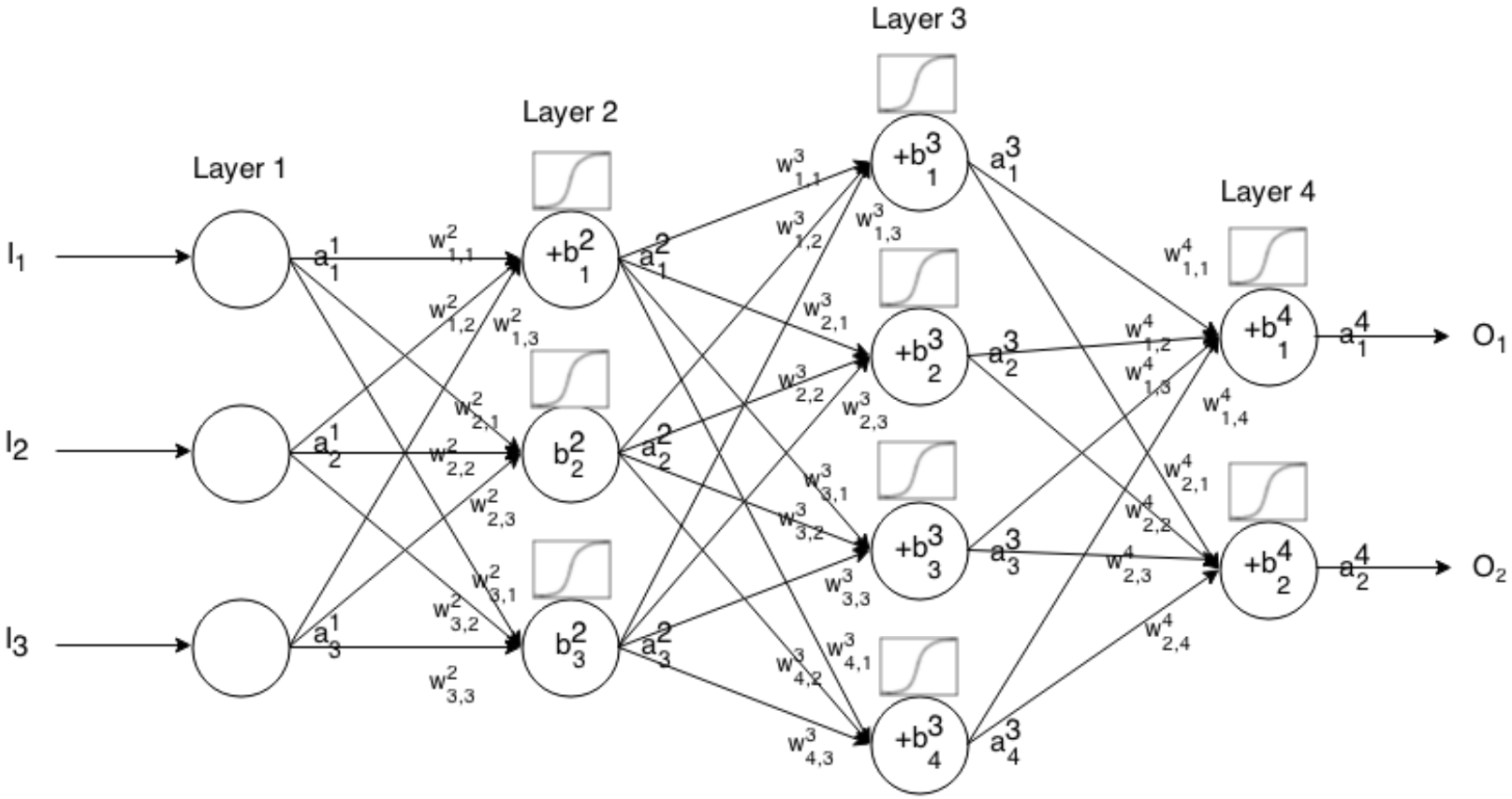
  Tipicamente escolhidas iguais, $g_\ell(z) = g(z)$, exceto possivelmente $g_L(z)$

Redes Neurais *Feedforward* (sem realimentação)



- Rede de $L = 3$ camadas (não contamos a camada de entrada)
- Ou: rede de 2 camadas ocultas

Redes Neurais *Feedforward* (sem realimentação)



- Rede de $L = 3$ camadas (não contamos a camada de entrada)
- Ou: rede de 2 camadas ocultas
- Obs: notação errada: a camada de entrada tem índice $\ell = 0$

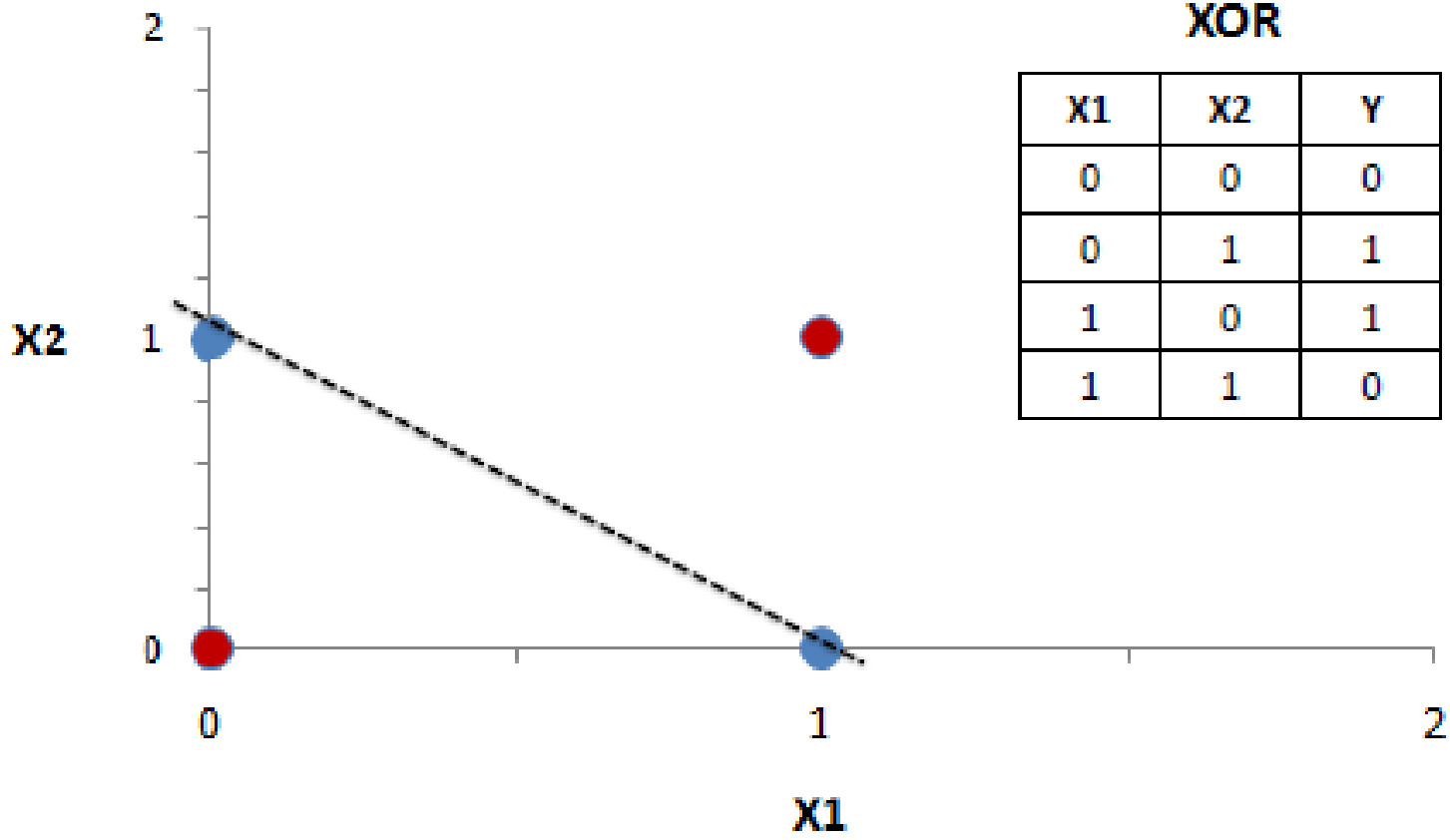
Motivação: Aumentando a Capacidade do Modelo

- Suponha por hora que não estamos preocupados com generalização, apenas em representar com mínimo erro o conjunto de treinamento
- Suponha que exista uma função $f^* : \mathbb{R}^n \to \mathbb{R}^K$ e que dispomos de m amostras (possivelmente ruidosas) de pares $(\mathbf{x}, \mathbf{y})$, onde $\mathbf{y} \approx f^*(\mathbf{x})$
- Nosso objetivo é encontrar uma função f que aproxima f^*
- Regressão logística (com os atributos originais $\mathbf{x}$), isto é,

$$\hat{\mathbf{y}} = f(\mathbf{x}) = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

  funciona bem quando as classes são separáveis por hiperplanos
- Caso contrário, precisamos criar novos atributos derivados dos originais para obter regiões de decisão mais complexas

Exemplo: XOR

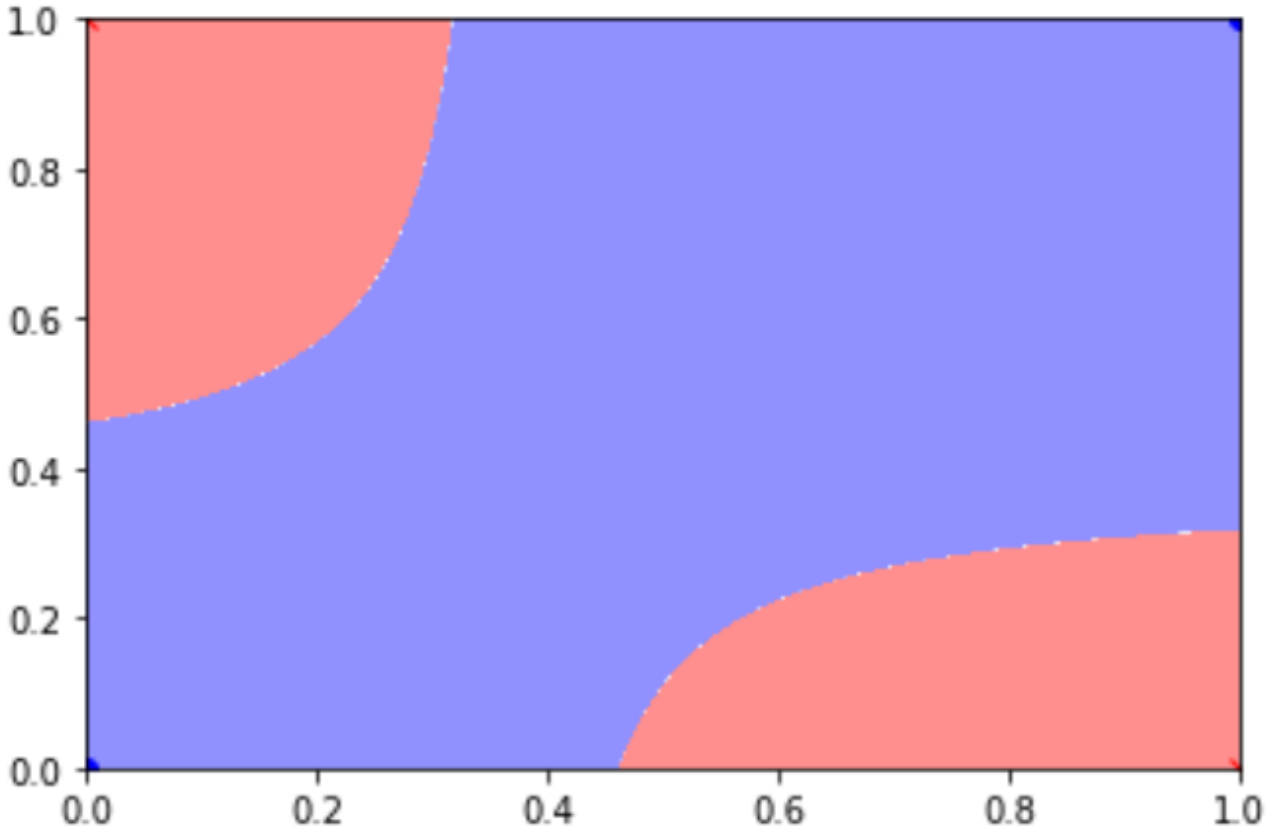


XOR

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

É impossível representar os dados com os atributos originais x_1 e x_2, mas é possível adicionando o termo $x_1 x_2$

Exemplo: XOR

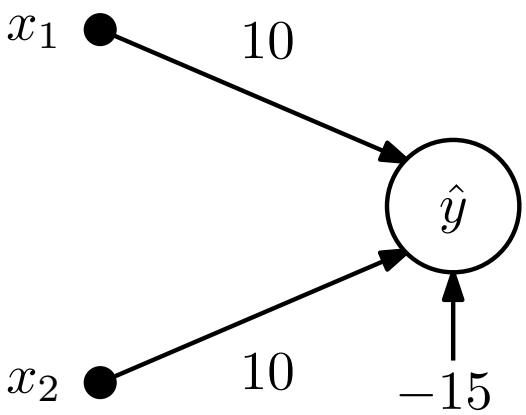


É impossível representar os dados com os atributos originais x_1 e x_2, mas é possível adicionando o termo $x_1 x_2$

Determinação de Atributos

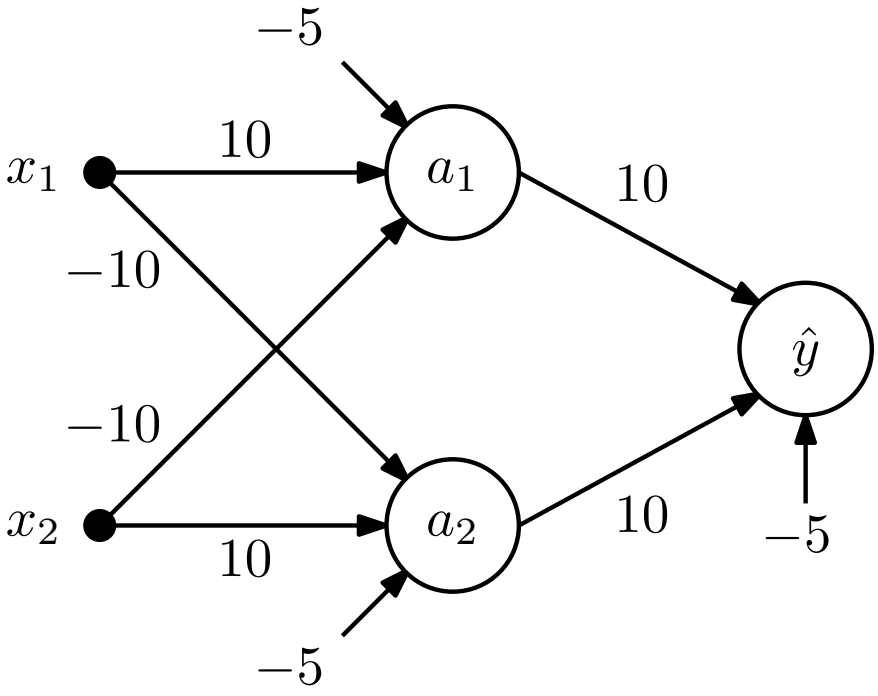
- Como escolher os atributos derivados?
  - Escolha manual:
    - requer “criatividade” e conhecimento específico do problema
    - se adicionarmos atributos polinomiais até grau d, ocorrerá uma explosão de termos: total de $\binom{n+d}{d} \geq (n/d)^d$ atributos
    - cresce rapidamente com o aumento de n
  - Escolha automática:
    - Função genérica suficientemente flexível com parâmetros que podem ser encontrados via treinamento
    - Troca “engenharia de atributos” por “aprendizagem de atributos” (*feature learning / representation learning*)

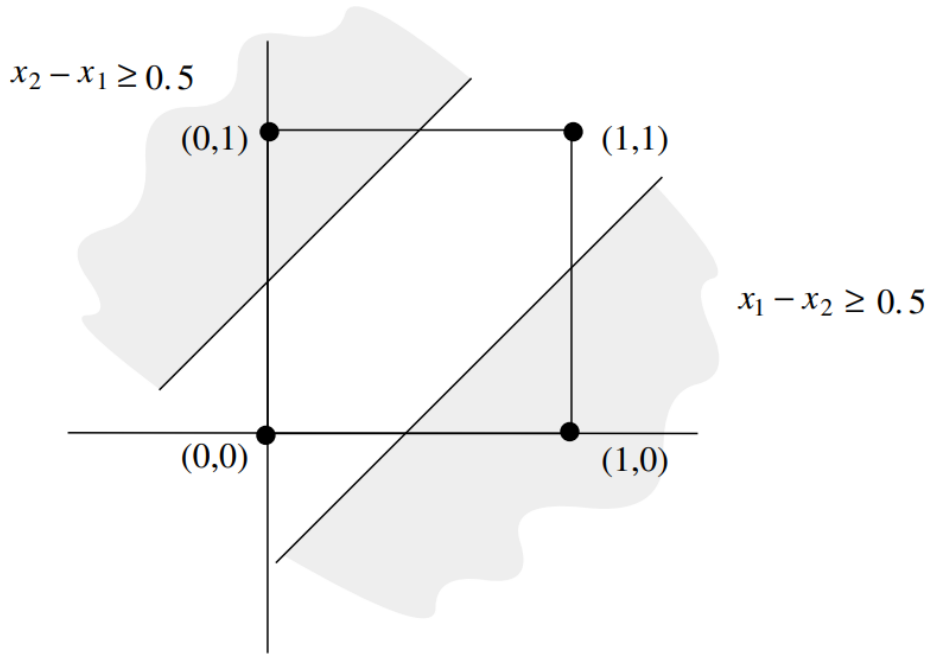
Exemplo: AND



$$\hat{y} = \sigma(10(x_1 + x_2 - 1.5)) \approx x_1 \text{ AND } \bar{x_2}$$

Exemplo: XOR





$$a_1 = \sigma(10(x_1 - x_2 - 0.5)) \approx x_1 \text{ AND } \bar{x_2}$$
$$a_2 = \sigma(10(x_2 - x_1 - 0.5)) \approx \bar{x_1} \text{ AND } x_2$$
$$\hat{y} = \sigma(10(a_1 + a_2 - 0.5)) \approx a_1 \text{ OR } \bar{a_2}$$

Flexibilidade das Redes Neurais

- Redes neurais são aproximadores universais:

Teorema

Uma rede neural de 2 camadas (1 camada oculta) com um número suficiente de unidades ocultas é capaz de aproximar qualquer função (regressão) ou região de decisão (classificação) suave.

- Também é fácil perceber que uma rede neural suficiente larga e profunda é capaz de implementar qualquer função booleana, uma vez que é capaz de implementar portas lógicas

- Mas como encontrar os pesos (treinamento)?

Treinamento

Função Custo

- Seja $\boldsymbol{\theta} = (\theta_1, \theta_2, \ldots, \theta_N) = (w_{kj}^{[\ell]},\ b_k^{[\ell]},\ \forall k, j, \ell)$ o vetor que consiste de todos os parâmetros do modelo
- Dado um conjunto de dados $\mathcal{D} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}),\ i = 1, \ldots, m\}$, desejamos minimizar a função custo:

$$J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} L(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}), \quad \text{onde } \hat{\mathbf{y}}^{(i)} = f(\mathbf{x}^{(i)} | \boldsymbol{\theta})$$

- Desafio: $J(\boldsymbol{\theta})$ é, em geral, uma função não-convexa. Possíveis abordagens incluem:
  - Múltiplas reinicializações aleatórias
  - Métodos avançados de otimização
  - Quem precisa do ótimo global?

Cálculo do Gradiente

- Otimização local eficiente requer cálculo do gradiente $\nabla J(\boldsymbol{\theta})$:
  - Métodos de 1ª ordem (*gradient descent* e variações)
  - Métodos de 2ª ordem:
    - Método de Newton: requer hessiana além do gradiente
    - Métodos quasi-Newton: utilizam uma aproximação da hessiana
- Como calcular $\nabla J(\boldsymbol{\theta})$ de forma eficiente?
- Algoritmo *backpropagation* (propagação reversa): Aplicação sucessiva da regra da cadeia reutilizando operações
  - Historicamente: "backpropagation" = backpropagation + gradient descent
  - Caso particular de reverse-mode autodifferentiation

Cálculo do Gradiente

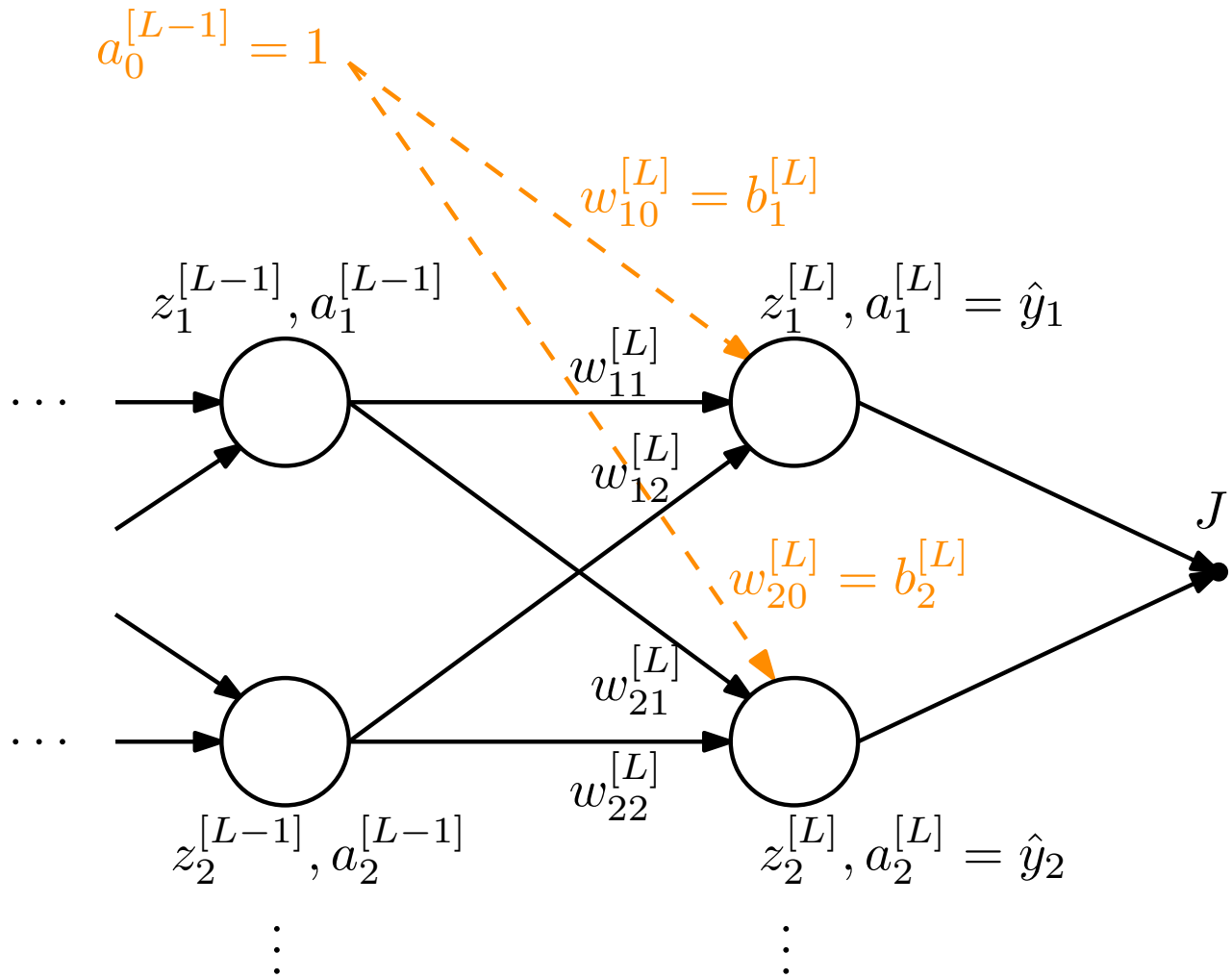
- Função custo:

$$J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} J^{(i)}(\boldsymbol{\theta}), \quad J_i(\boldsymbol{\theta}) = L(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}), \quad \hat{\mathbf{y}}^{(i)} = f(\mathbf{x}^{(i)}|\boldsymbol{\theta})$$

$$\nabla J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \nabla J^{(i)}(\boldsymbol{\theta})$$

- Para simplificar a notação, considere:
  - $\mathbf{x} = \mathbf{x}^{(i)}$, $\mathbf{y} = \mathbf{y}^{(i)}$, $\hat{\mathbf{y}} = \hat{\mathbf{y}}^{(i)}$, $J(\boldsymbol{\theta}) = J^{(i)}(\boldsymbol{\theta})$
  - $w_{k0}^{[\ell]} = b_k^{[\ell]}$ e $a_0^{[\ell-1]} = 1, \ell = 1, \ldots, L$
  - $\nabla J(\boldsymbol{\theta}) = \left(\frac{\partial J}{\partial w_{kj}^{[\ell]}}, \ \forall k, j, \ell \right)$
  - $\mathbf{a}^{[0]} = \mathbf{x}$ e $\mathbf{a}^{[L]} = \hat{\mathbf{y}}$

Redes Neurais: Notação



Cálculo do Gradiente

- Propagação direta: para $\ell = 1, \ldots, L$:

$$\begin{cases} \mathbf{z}^{[\ell]} &= \mathbf{W}^{[\ell]}\mathbf{a}^{[\ell-1]} \\ \mathbf{a}^{[\ell]} &= g_\ell(\mathbf{z}^{[\ell]}) \end{cases} \iff \begin{cases} z_k^{[\ell]} = \sum_{j=0}^{n_{\ell-1}} w_{kj}^{[\ell]} a_j^{[\ell-1]} \\ a_k^{[\ell]} = g_\ell(z_k^{[\ell]}) \end{cases}$$

- Regra da cadeia:

$$\frac{\partial J}{\partial w_{kj}^{[\ell]}} = \frac{\partial z_k^{[\ell]}}{\partial w_{kj}^{[\ell]}} \cdot \frac{\partial J}{\partial z_k^{[\ell]}}, \qquad \ell = 1, \ldots, L$$

$$\delta_k^{[L]} \triangleq \frac{\partial J}{\partial z_k^{[L]}} = \frac{\partial a_k^{[L]}}{\partial z_k^{[L]}} \cdot \frac{\partial J}{\partial a_k^{[L]}}$$

$$\delta_j^{[\ell]} \triangleq \frac{\partial J}{\partial z_j^{[\ell]}} = \frac{\partial a_j^{[\ell]}}{\partial z_j^{[\ell]}} \cdot \sum_{k=1}^{n_{\ell+1}} \frac{\partial z_k^{[\ell+1]}}{\partial a_j^{[\ell]}} \cdot \frac{\partial J}{\partial z_k^{[\ell+1]}}, \qquad \ell = 1, \ldots, L-1$$

Cálculo do Gradiente

- Propagação direta: para $\ell = 1, \ldots, L$:

$$\begin{cases} \mathbf{z}^{[\ell]} &= \mathbf{W}^{[\ell]}\mathbf{a}^{[\ell-1]} \\ \mathbf{a}^{[\ell]} &= g_\ell(\mathbf{z}^{[\ell]}) \end{cases} \iff \begin{cases} z_k^{[\ell]} = \sum_{j=0}^{n_{\ell-1}} w_{kj}^{[\ell]} a_j^{[\ell-1]} \\ a_k^{[\ell]} = g_\ell(z_k^{[\ell]}) \end{cases}$$

- Propagação reversa: para $\ell = L, \ldots, 1$:

$$\delta_j^{[\ell]} = \begin{cases} g'_L(z_j^{[L]}) \dfrac{\partial J}{\partial a_j^{[L]}}, & \ell = L \\ g'_\ell(z_j^{[\ell]}) \displaystyle\sum_{k=1}^{n_{\ell+1}} w_{kj}^{[\ell+1]} \delta_k^{[\ell+1]}, & \ell < L \end{cases}$$

$$\frac{\partial J}{\partial w_{kj}^{[\ell]}} = a_j^{[\ell-1]} \delta_k^{[\ell]}$$

Cálculo do Gradiente (Notação Matricial)

- Propagação direta: para $\ell = 1, \ldots, L$:

$$\mathbf{z}^{[\ell]} = \mathbf{W}^{[\ell]}\mathbf{a}^{[\ell-1]}$$
$$\mathbf{a}^{[\ell]} = g_\ell(\mathbf{z}^{[\ell]})$$

- Propagação reversa: para $\ell = L, \ldots, 1$:

$$\boldsymbol{\delta}^{[\ell]} = \begin{cases} g'_L(\mathbf{z}^{[L]}) \odot \dfrac{\partial J}{\partial \mathbf{a}^{[L]}}, & \ell = L \\ g'_\ell(\mathbf{z}^{[\ell]}) \odot \left(\mathbf{W}^{[\ell+1]^T}\boldsymbol{\delta}^{[\ell+1]}\right), & \ell < L \end{cases}$$

$$\frac{\partial J}{\partial \mathbf{W}^{[\ell]}} = \boldsymbol{\delta}^{[\ell]}\mathbf{a}^{[\ell-1]^T} + \lambda\mathbf{W}^{[\ell]}\begin{bmatrix} 0 & 0 \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \text{ (c/ regulariz. L2)}$$

Obs: $\odot$ = multiplicação elemento a elemento

- Complexidade: $O(N)$ operações, onde N é a dimensão de $\boldsymbol{\theta}$

Casos Particulares

- Regressão: Ativação de saída linear com perda de erro quadrático:

$$a_k^{[L]} = z_k^{[L]}, \qquad J = \frac{1}{2}\sum_{k=1}^{K}(a_k^{[L]} - y_k)^2$$

- Classificação binária: Ativação de saída logística com perda de entropia cruzada:

$$a_k^{[L]} = \sigma(z_k^{[L]}) = \frac{1}{1+e^{-z_k^{[L]}}}, \quad J = -\sum_{k=1}^{K} y_k \log a_k^{[L]} + (1-y_k)\log(1-a_k^{[L]})$$

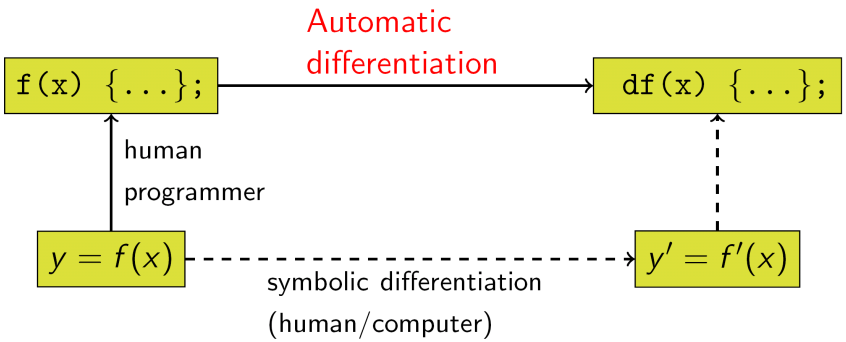
- Classificação multi-classe: Ativação de saída softmax com perda de entropia cruzada categórica:

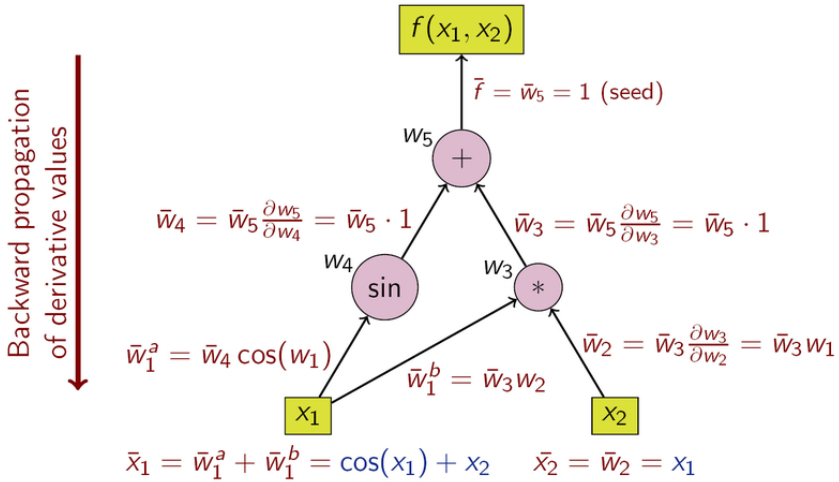
$$a_k^{[L]} = \mathrm{softmax}_k(\mathbf{z}^{[L]}) = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}, \quad J = -\sum_{k=1}^{K} y_k \log a_k^{[L]}$$

- Em todos os três casos, temos:

$$\delta_k^{[L]} = \frac{\partial J}{\partial z_k^{[L]}} = a_k^{[L]} - y_k$$

Autodiff na prática






```
>>> import autograd.numpy as np  # Thinly-wrapped numpy
>>> from autograd import grad    # The only autograd function you may ever need
>>>
>>> def tanh(x):                 # Define a function
...     y = np.exp(-2.0 * x)
...     return (1.0 - y) / (1.0 + y)
...
>>> grad_tanh = grad(tanh)       # Obtain its gradient function
>>> grad_tanh(1.0)               # Evaluate the gradient at x = 1.0
0.41997434161402603
>>> (tanh(1.0001) - tanh(0.9999)) / 0.0002  # Compare to finite differences
0.41997434264973155
```

Inicialização de pesos

- Embora os pesos da camada de saída e todos os termos de bias possam ser inicializados com zeros, os demais pesos devem ser inicializados com valores distintos para quebrar a simetria existente entre as unidades ocultas
  - Caso contrário, as unidades aprenderão os mesmos pesos, tornando-se idênticas
- Por outro lado, os valores não podem ser muito altos, caso contrário tenderão a causar uma saturação da função de ativação, o que por sua vez resulta em um aprendizado muito lento
  - Pequenas variações nos parâmetros daquela unidade praticamente não terão impacto no custo final
- Recomenda-se utilizar uma inicialização aleatória, como uniforme $[-\epsilon, \epsilon]$ ou gaussiana $\mathcal{N}(0, \epsilon^2)$, onde ϵ é um valor pequeno
  - Diversas heurísticas dependendo da função de ativação interna

Outras funções de ativação (para camadas ocultas)

- Tangente hiperbólica:

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

- ReLU (*rectified linear unit*):

$$\mathrm{relu}(x) = \max(0, x)$$