

TP555 - Inteligência Artificial e Machine Learning: *Redes Neurais Artificiais (Parte II)*

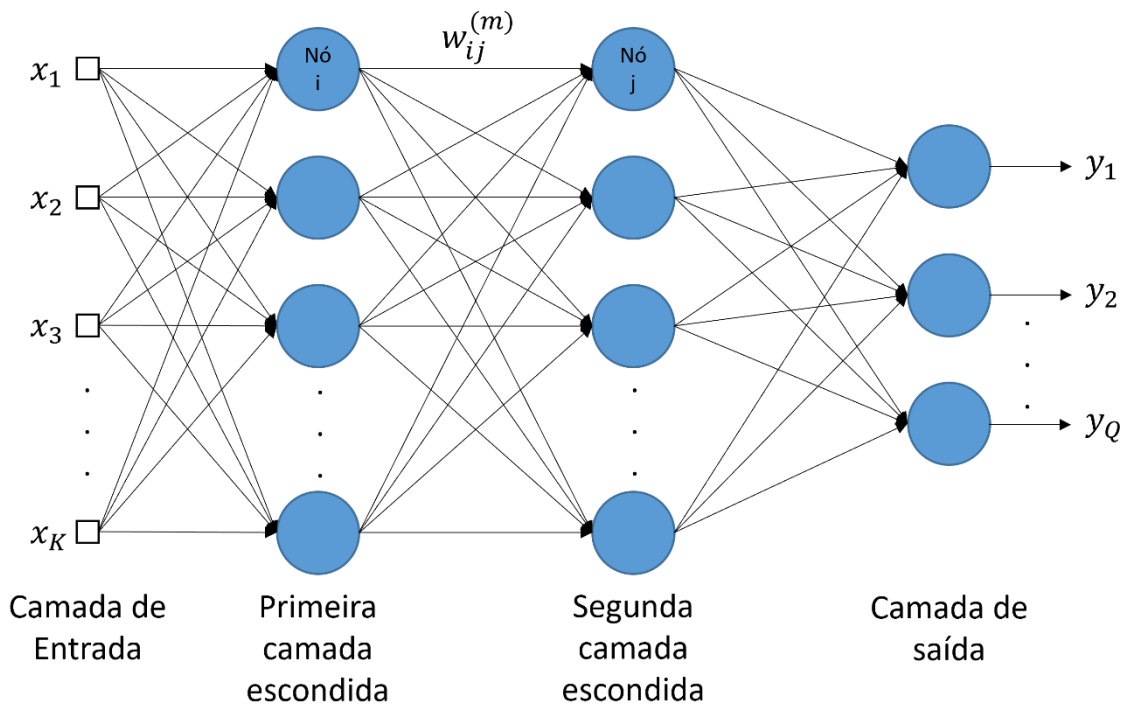


Inatel

Felipe Augusto Pereira de Figueiredo
felipe.figueiredo@inatel.br

Redes neurais artificiais

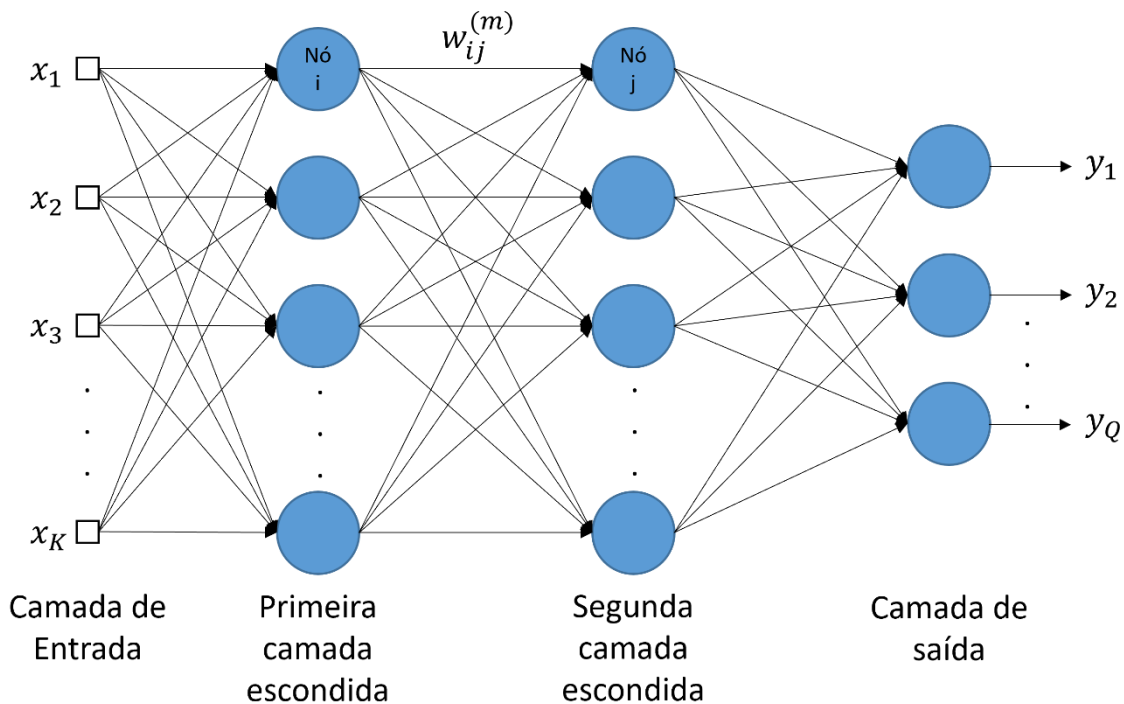
Cada ligação tem um peso (sináptico) associado.



- Uma **rede neural artificial (RNA)** nada mais é do que uma **combinação de neurônios** conectados entre si através de **ligações direcionadas** (i.e., as conexões têm uma direção associada).
 - Neurônios também são chamados de **nós** ou **unidades**.
 - **Cada ligação** entre nós **possui um peso (sináptico) associado**.
- As RNAs são formadas por uma camada de entrada, zero ou mais camadas intermediárias e uma camada de saída.

Redes neurais artificiais

Cada ligação tem um peso (sináptico) associado.



● Nó, unidade ou neurônio.

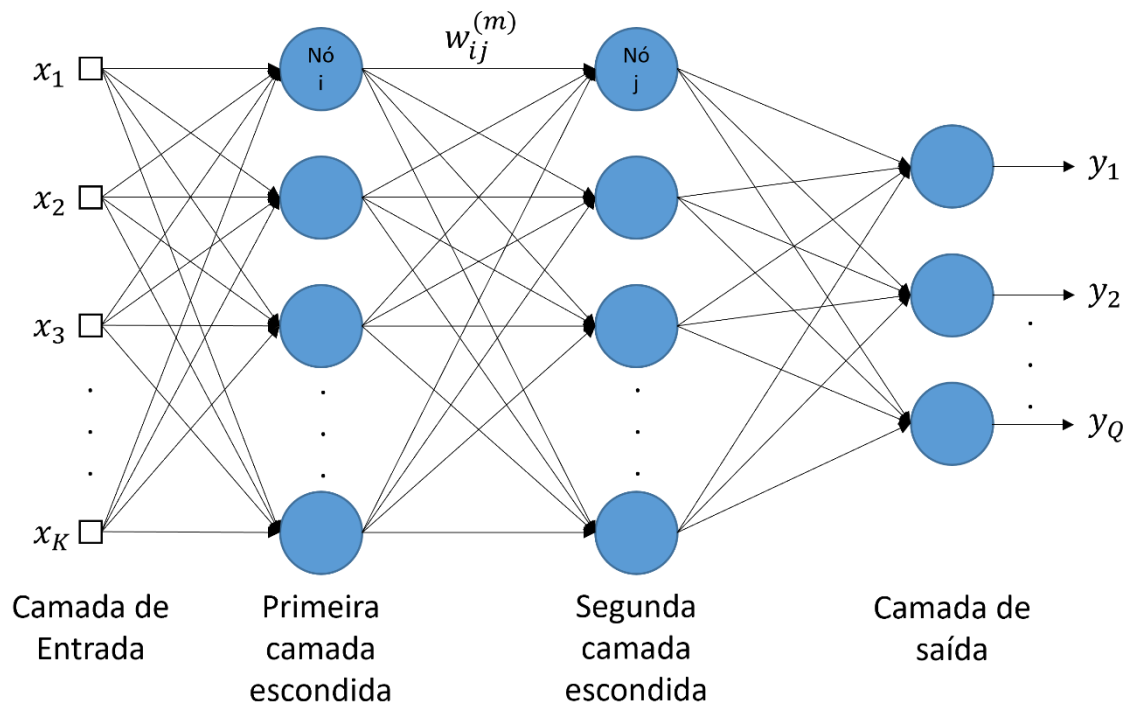
→ Ligação entre i -ésimo e j -ésimo nó.

w_{ij} Peso da ligação entre i -ésimo e j -ésimo nó.

- As camadas intermediárias são também chamadas de **ocultas** ou **escondidas**.
- Algumas das **limitações dos perceptrons** (e.g., classificação apenas de classes linearmente separáveis) podem ser **superadas adicionando-se camadas intermediárias de perceptrons**.
- O primeiro tipo de rede neural proposto foi o **perceptron de múltiplas camadas** (do inglês, *Multilayer Perceptron* - MLP).

Redes neurais artificiais

Cada ligação tem um peso (sináptico) associado.



● Nó, unidade ou neurônio.

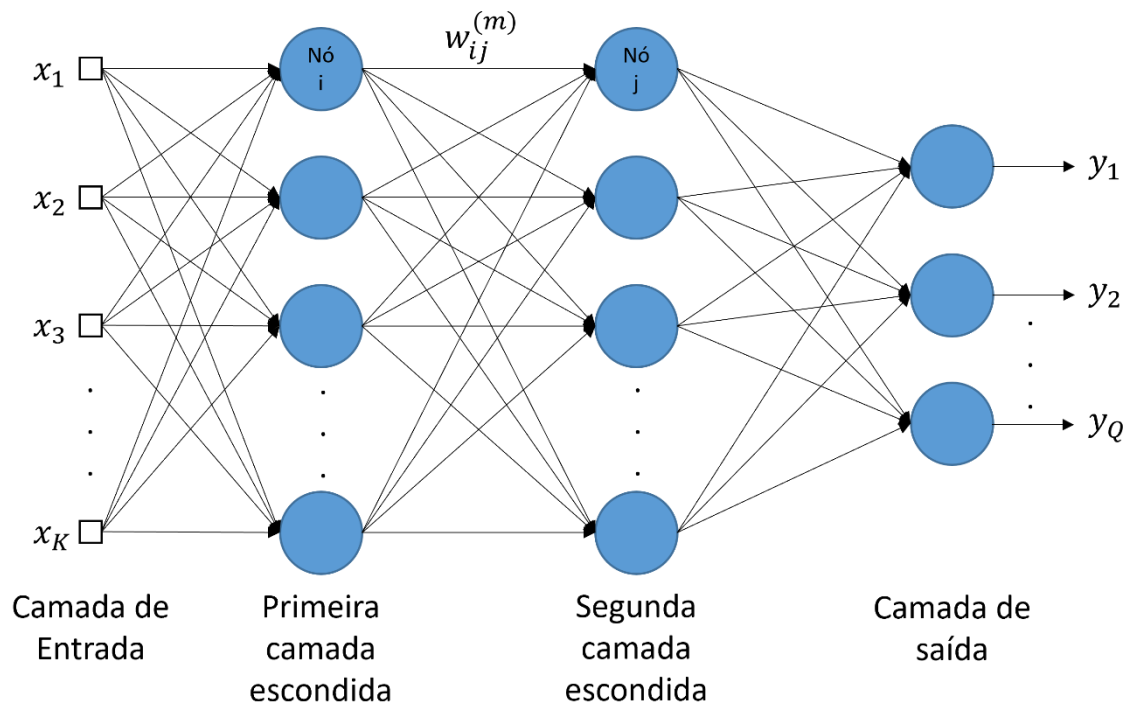
→ Ligação entre i -ésimo e j -ésimo nó.

w_{ij} Peso da ligação entre i -ésimo e j -ésimo nó.

- Ela também é chamada de **rede densamente conectada e de alimentação direta** (do inglês, *Dense Neural Network - DNN*).
 - Cada uma das saídas de uma camada **se conecta a todos os nós** da camada seguinte através de pesos sinápticos.
 - Os **dados fluem através da rede em uma única direção**, da camada de entrada para a camada de saída, **sem ciclos de realimentação**.

Redes neurais artificiais

Cada ligação tem um peso
(sináptico) associado.



Nó, unidade ou neurônio.

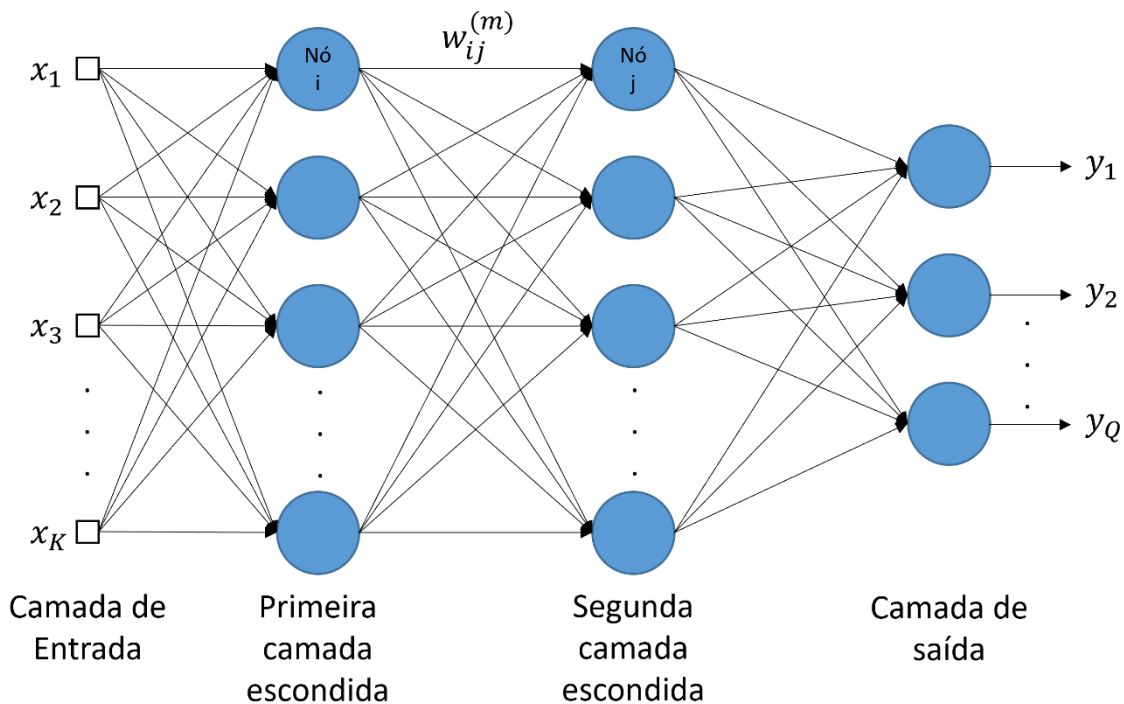
Ligação entre i -ésimo e j -ésimo nó.

w_{ij} Peso da ligação entre i -ésimo e j -ésimo nó.

- As *propriedades de uma rede neural* são determinadas por sua *arquitetura*,
 - como os neurônios estão conectados (forma direta ou recursiva),
 - quantidade neurônios,
 - quantidade de camadas escondidas,
 - função de ativação,
 - etc.

Redes neurais artificiais

Cada ligação tem um peso
(sináptico) associado.



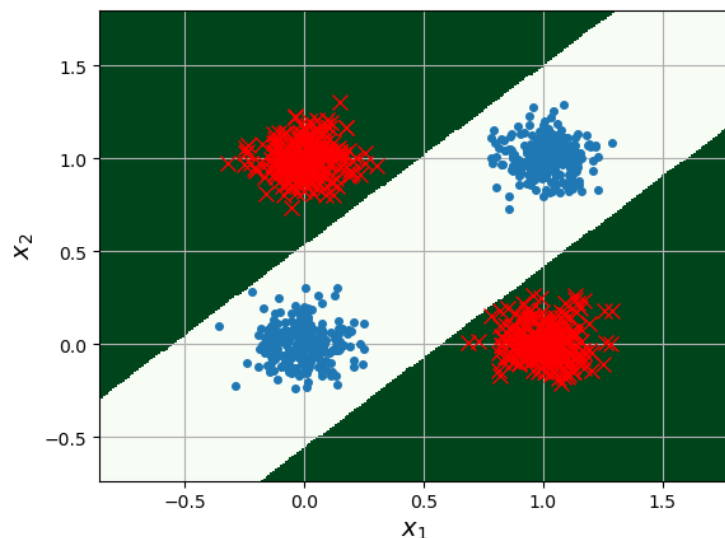
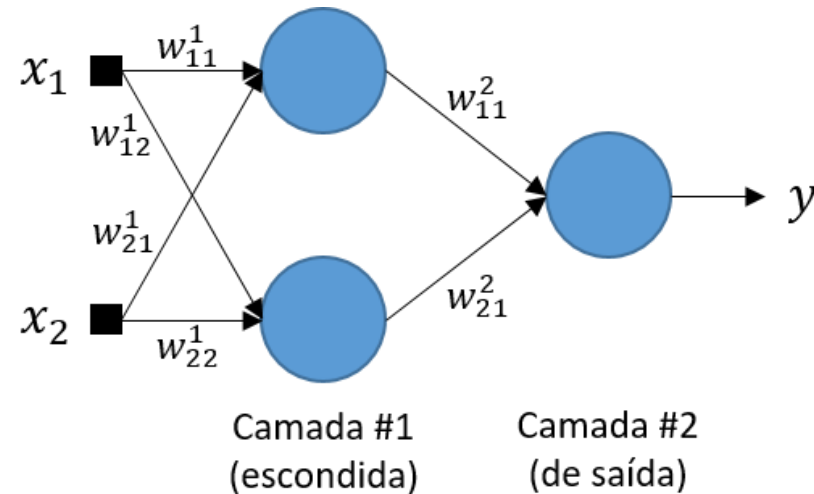
● Nó, unidade ou neurônio.

→ Ligação entre i -ésimo e j -ésimo nó.

w_{ij} Peso da ligação entre i -ésimo e j -ésimo nó.

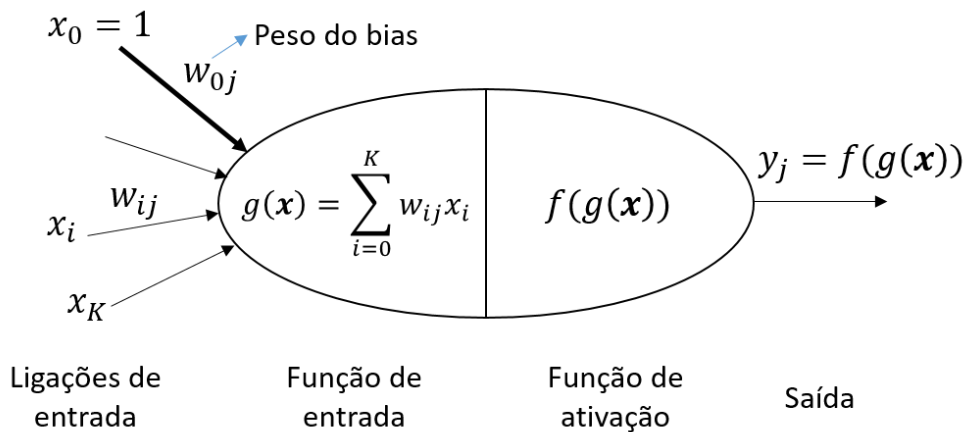
- As RNAs são o coração do ***deep learning*** ou ***aprendizado profundo***.
- O termo "***profundo***" vem fato de que essas redes podem possuir ***muitas camadas ocultas***.
- Em geral, quando uma RNA tem duas ou mais camadas ocultas, ela pode ser chamada de ***rede neural profunda*** (ou em inglês, ***Deep Neural Network - DNN***).
- A rede MLP ao lado possui duas camadas ocultas e, portanto, poderia ser chamada de DNN.

Redes neurais artificiais



- Em particular, uma MLP com **uma camada oculta com dois nós** e **uma camada de saída com um nó** pode resolver o problema da lógica XOR.
- Lembrem-se que um único **perceptron** não é capaz de realizar essa tarefa.
- Os dois nós da camada oculta **aprendem separadores lineares** que são **combinados** para obter a **separação não linear** resultante.

Os nós das redes neurais artificiais



- **Cada nó** tem a entrada x_0 (i.e., o atributo de bias) sempre com valor igual a 1 e um peso associado w_{0j} , chamado de **peso de bias**.
 - Ou seja, a entrada x_0 **não está conectada a nenhum outro nó**.
- O j -ésimo **nó** calcula a **soma ponderada** de suas entradas, x_i

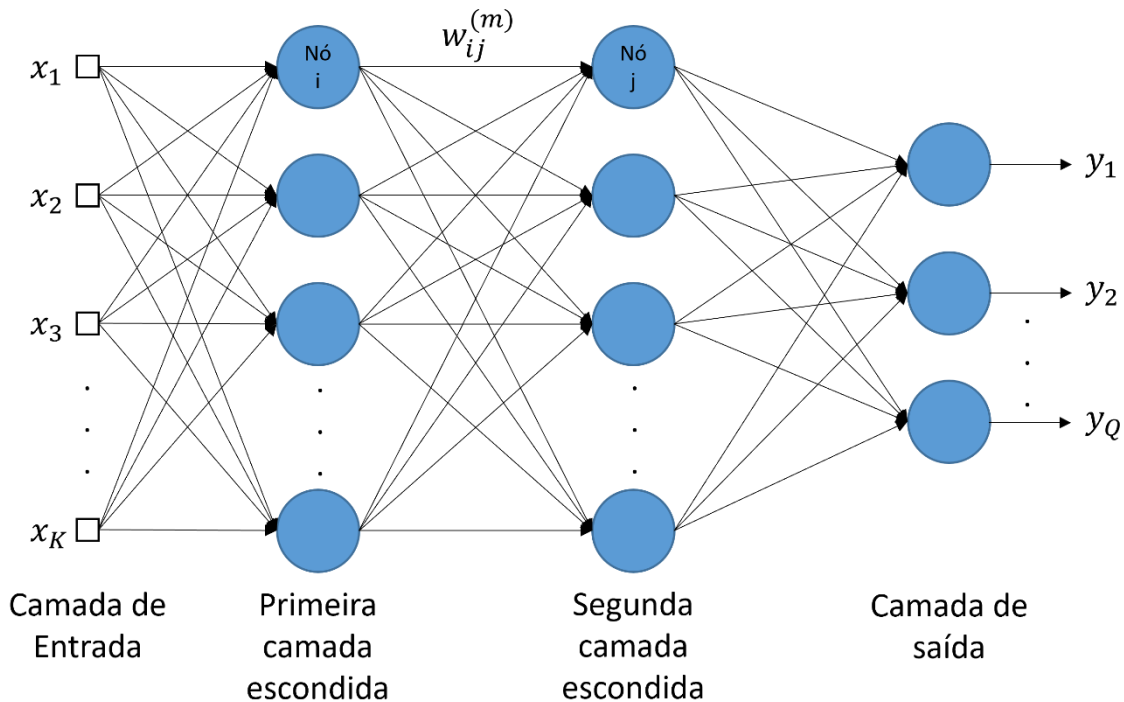
$$g(x) = \sum_{i=0}^K w_{ij}x_i = \mathbf{w}^T \mathbf{x},$$

e, em seguida, aplica uma **função de ativação** (i.e., de limiar), $f(\cdot)$, à soma para gerar sua saída

$$y_j = f(g(x)).$$

As ligações dos nós das redes neurais artificiais

Cada ligação tem um peso (sináptico) associado.



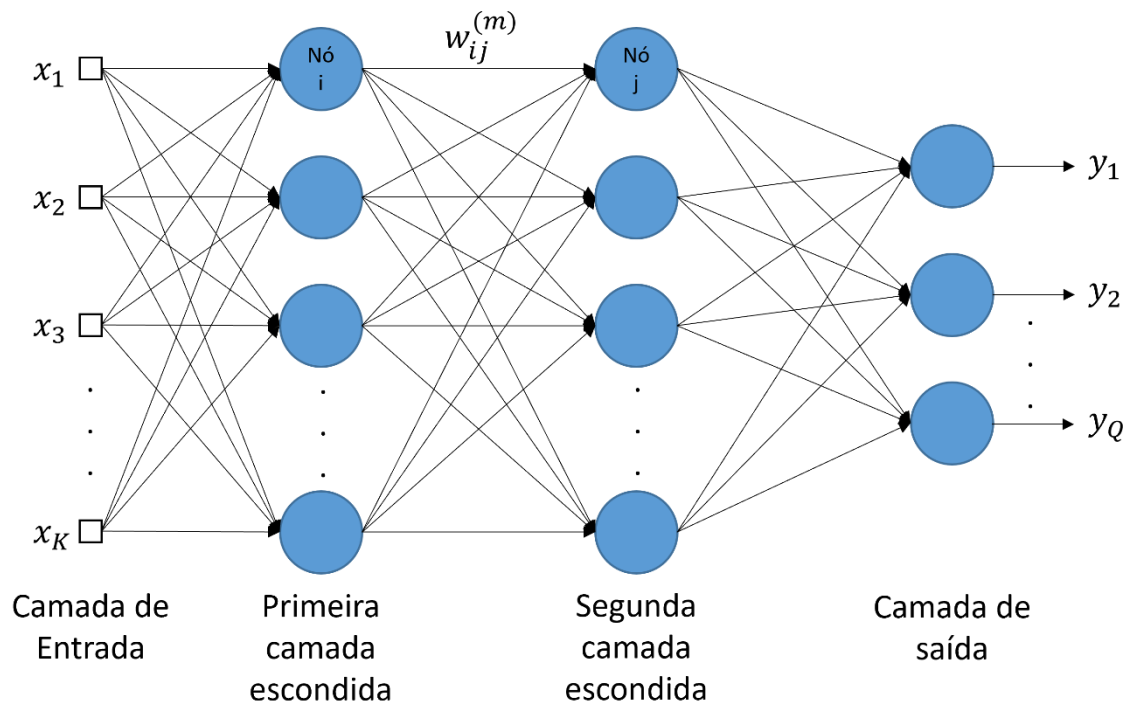
● Nó, unidade ou neurônio.


→ Ligação entre i -ésimo e j -ésimo nó.


w_{ij} Peso da ligação entre i -ésimo e j -ésimo nó.

- Considerando **qualquer dois nós da rede**, a **ligação** do i -ésimo **nó** da $m - 1$ -ésima camada para o j -ésimo **nó** da m -ésima é feita através do **peso** $w_{ij}^{(m)}$.
- A ligação **propaga** o **signal de saída** do i -ésimo **nó** para o j -ésimo **nó**.
 - O **signal de saída** do i -ésimo nó é denotado por x_i .
- O valor do **peso** determina a **força** e o **signal** da **ligação**.
- A ligação pode ser **excitatória** ou **inibitória** dependendo dos sinais do peso e de saída do nó anterior.

Funções de ativação



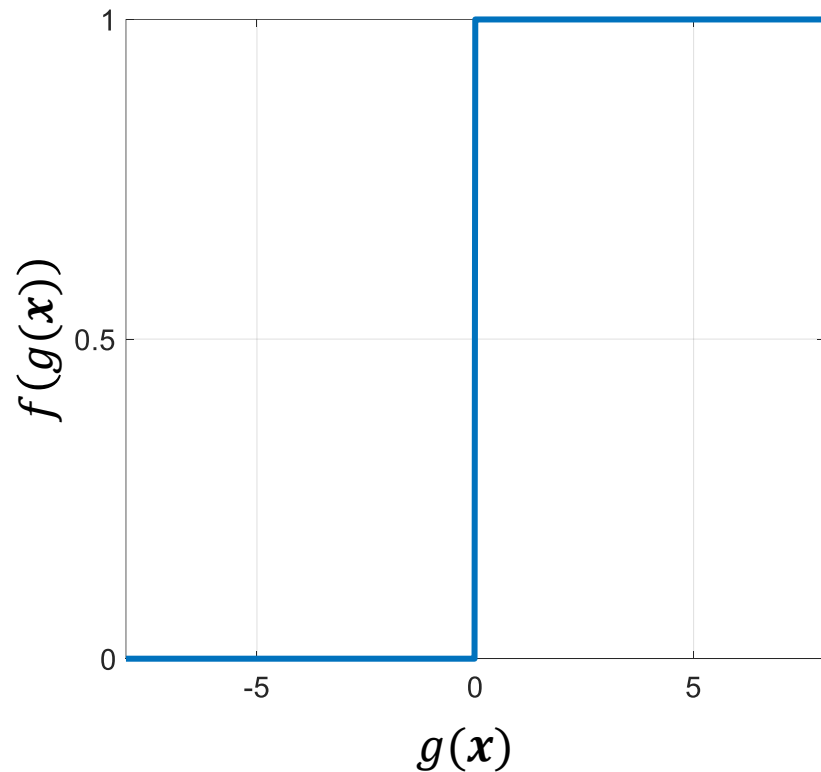
 Nó, unidade ou neurônio.

 Ligação entre i -ésimo e j -ésimo nó.

w_{ij} Peso da ligação entre i -ésimo e j -ésimo nó.

- Existem vários tipos de **funções de ativação** que podem ser utilizadas pelos **nós** de uma rede neural.
- Cada camada pode usar funções de ativação diferentes.
- Porém, em geral, todos os nós de uma camada usam o mesmo tipo de função de ativação.

Funções de ativação



- Devido a suas características, não se utiliza a ***função degrau*** como função de ativação em redes neurais.
 - Derivada sempre igual a zero, exceto na origem, onde ela é indeterminada.
- Até o surgimento das ***redes neurais profundas***, a regra era utilizar as ***funções logística*** ou ***tangente hiperbólica***, que são ***versões suavizadas da função degrau***.
 - Essas funções ***são contínuas e possuem derivada definida e diferente de 0 em todos os pontos***.

Função de ativação logística

- A saída de um nó com **função de ativação logística** (ou sigmoide) tem a seguinte expressão

$$y_j = f(g(\mathbf{x})) = \frac{1}{1 + e^{-g(\mathbf{x})}},$$

onde $g(\mathbf{x})$ é a **combinação linear das entradas do nó**.

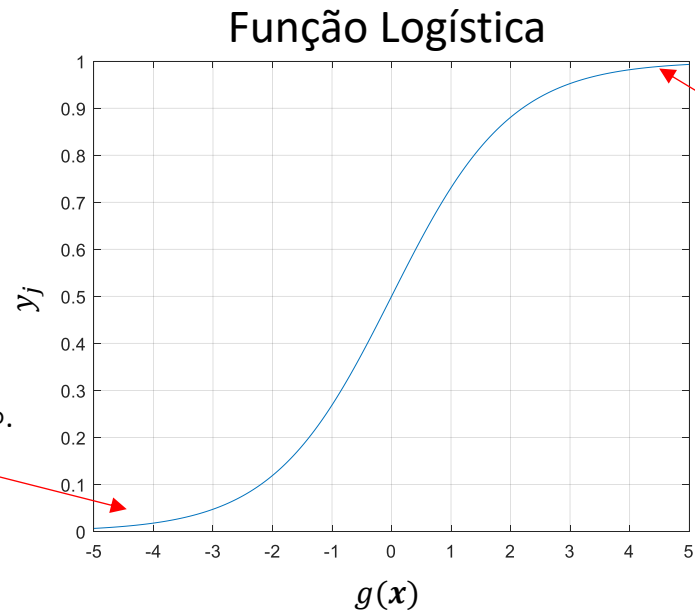
- Sua derivada é dada por

$$\frac{dy_j}{dg(\mathbf{x})} = \frac{df(g(\mathbf{x}))}{dg(\mathbf{x})} = y_j(1 - y_j) \geq 0.$$

- A derivada será importante durante o processo de aprendizado da rede neural.

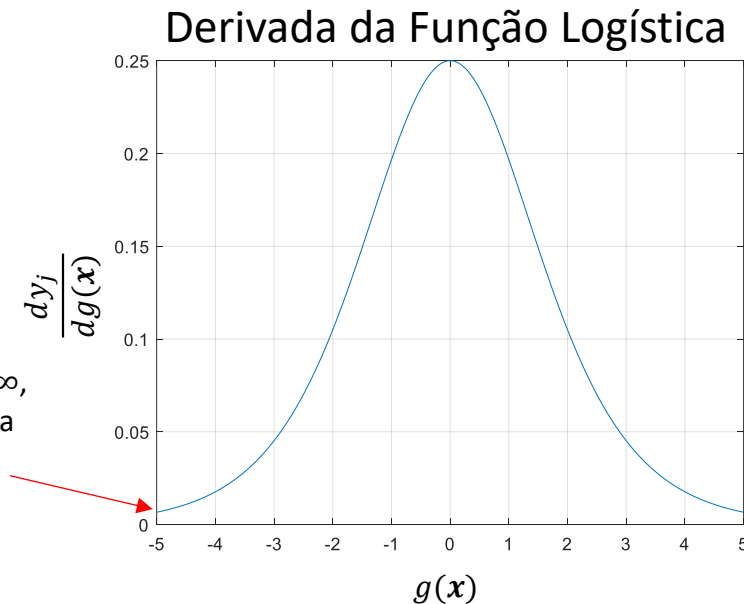
Função de ativação logística e sua derivada

- Percebam que o valor da derivada **sempre será menor do que 1, sendo no máximo igual a 0.25 quando $g(x) = 0$.**



saturação:
valor de $y_j \rightarrow 1$
quando $g(x) \rightarrow \infty$.

Quando $g(x) \rightarrow -\infty$,
 $y_j \rightarrow 0$ e a derivada
tende a 0.



Quando $g(x) \rightarrow \infty$,
 $y_j \rightarrow 1$ e a derivada
tende a 0.

Função de ativação tangente hiperbólica

- A saída de um nó com **função de ativação tangente hiperbólica** tem sua expressão dada por

$$y_j = f(g(\mathbf{x})) = \tanh(g(\mathbf{x})) = \frac{e^{g(\mathbf{x})} - e^{-g(\mathbf{x})}}{e^{g(\mathbf{x})} + e^{-g(\mathbf{x})}}.$$

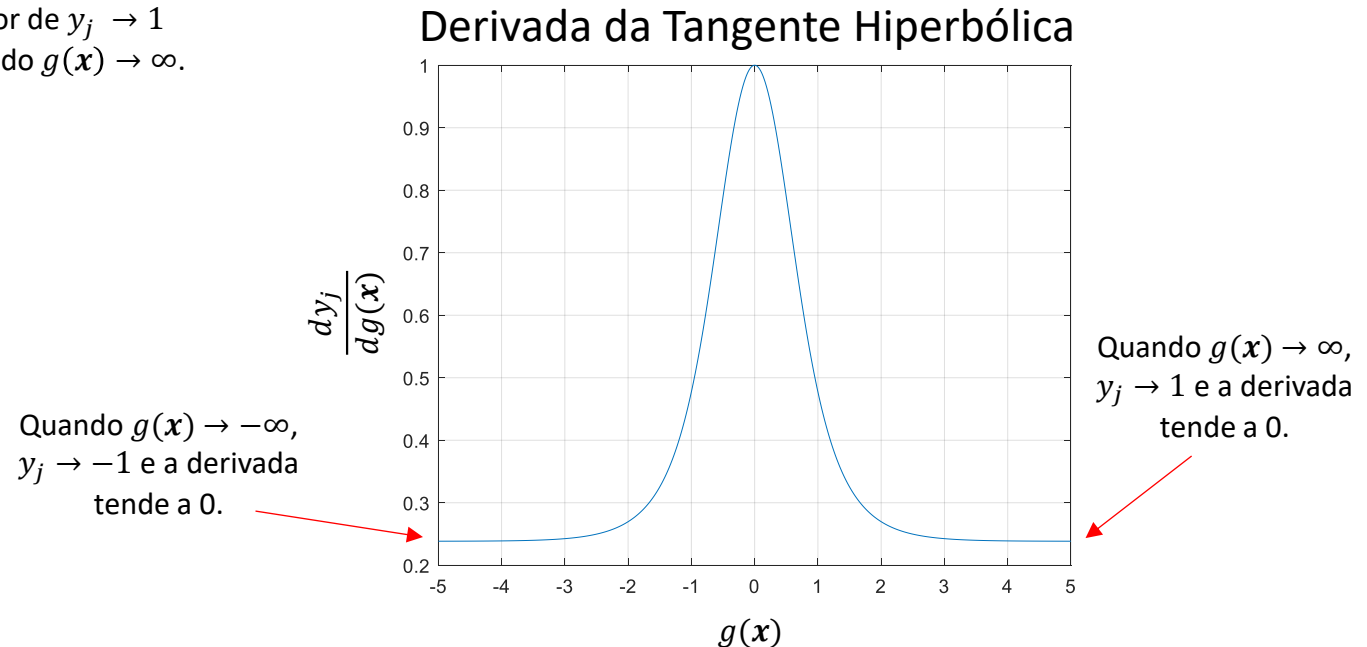
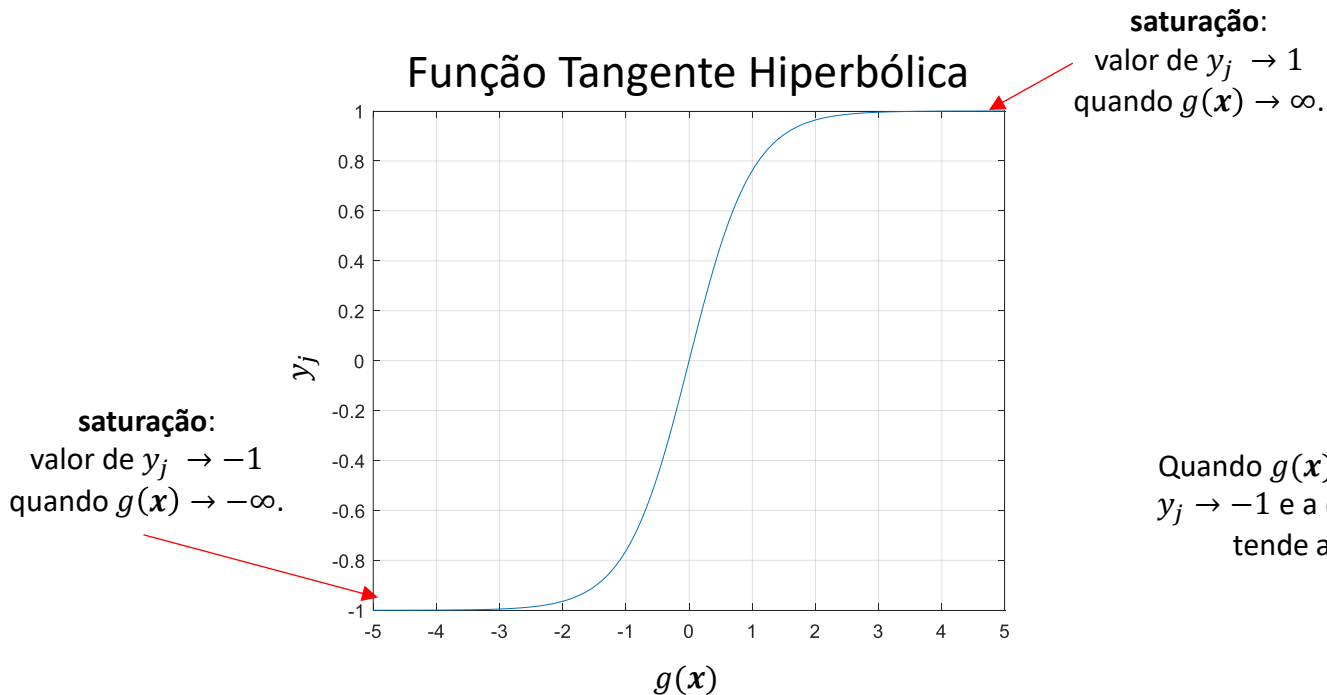
onde $g(\mathbf{x})$ é a **combinação linear das entradas do nó**.

- Sua derivada é dada por

$$\frac{dy_j}{dg(\mathbf{x})} = \frac{df(g(\mathbf{x}))}{dg(\mathbf{x})} = 1 - \tanh^2(g(\mathbf{x})) \geq 0.$$

Função de ativação tangente hiperbólica e sua derivada

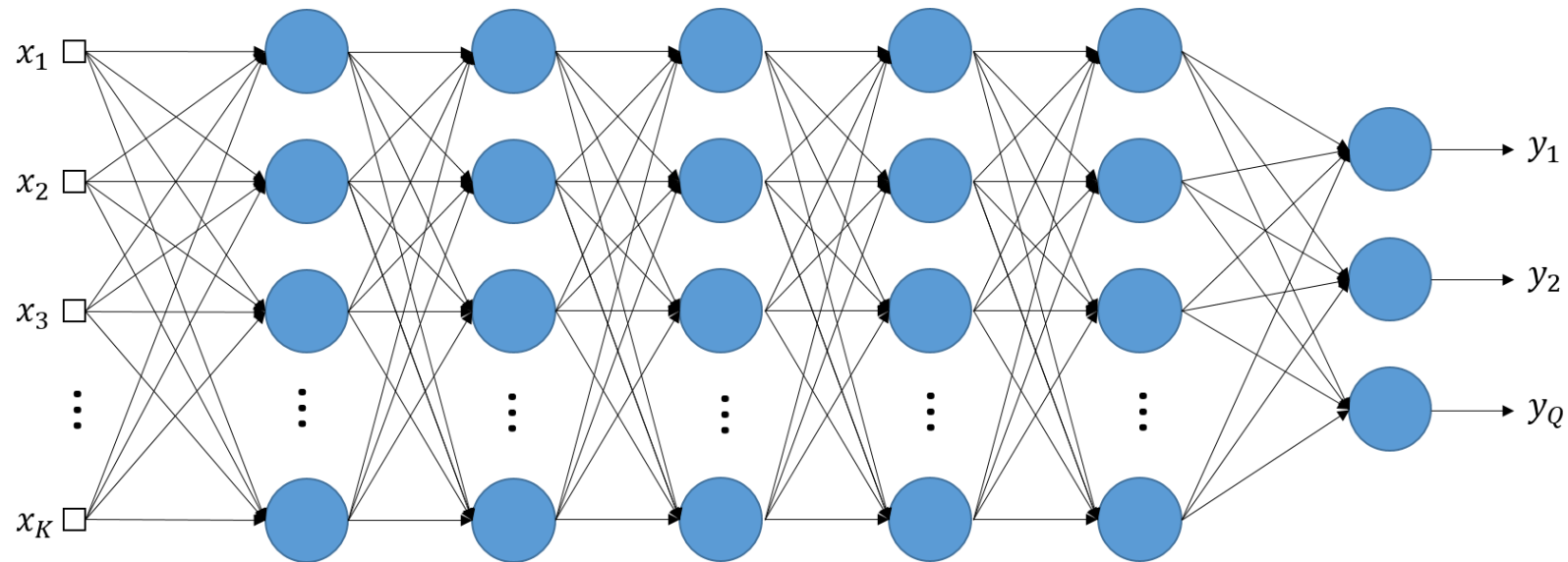
- A derivada é no máximo igual a 1 **exatamente** quando **quando** $g(x) = 0$, sendo menor do que 1 para todos os outros valores de $g(x)$.



Na sequência, veremos que esses valores de derivadas menores do que 1 causam um problema no aprendizado de redes com muitas camadas, i.e., redes profundas.

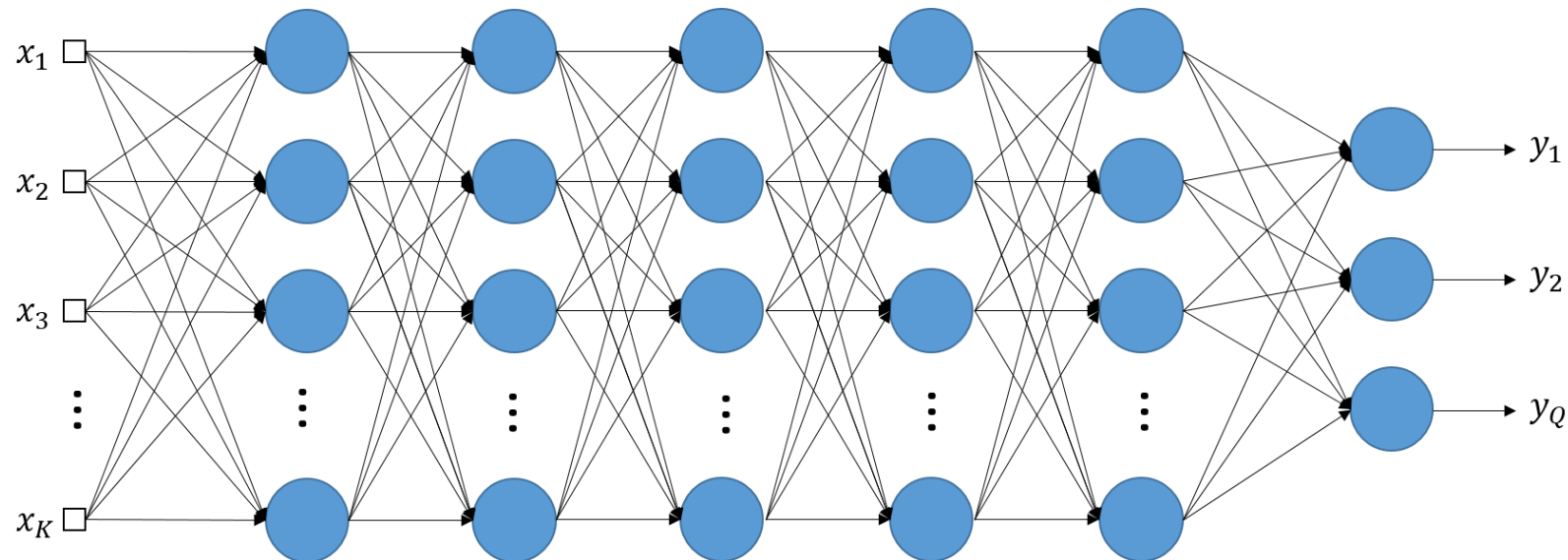
O problema da dissipação do gradiente

- É um problema encontrado quando treinamos *redes neurais profundas*, ou seja, com muitas camadas ocultas, com *métodos de aprendizado baseados no gradiente descendente* e nós usando *funções de ativação sigmoide ou tangente hiperbólica*.



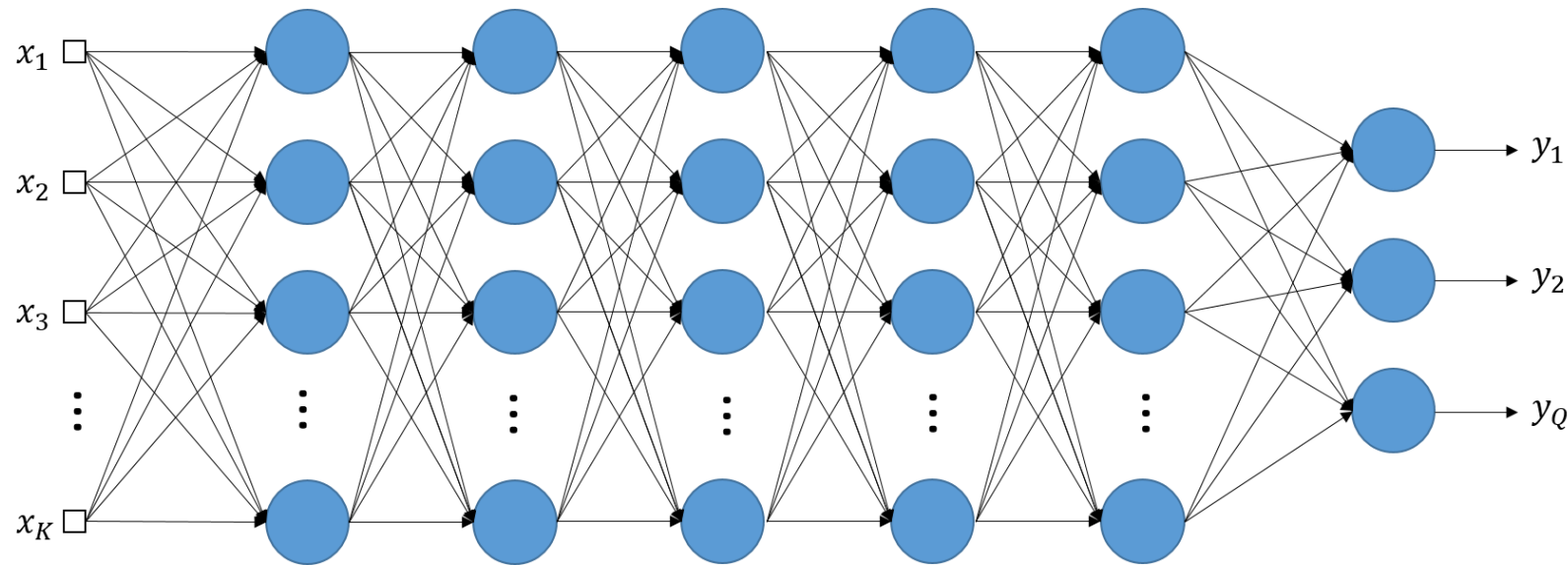
O problema da dissipação do gradiente

- Ocorre devido à natureza do **algoritmo de retropropagação**, que é usado para treinar a rede neural.
 - Para atualizar os pesos de nós das camadas ocultas, calcula-se a derivada do erro de saída em relação àqueles pesos e, para isso, usamos a **regra da cadeia**.
 - Ou seja, o algoritmo **propaga o erro de saída para as camadas ocultas** usando a **regra da cadeia**.



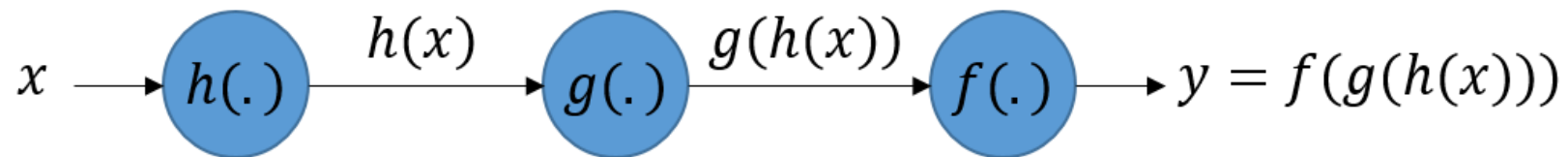
O problema da dissipação do gradiente

- Em suma, problema da dissipação do gradiente faz com que os *elementos do vetor gradiente se tornem cada vez menores* conforme ele é calculado para as *camadas próximas à entrada da rede*, levando a uma *atualização muito pequena ou até inexistente* dos pesos destas camadas.



Regra da cadeia

- Durante o treinamento, para **atualizar os pesos dos nós de cada camada** da rede, o **algoritmo de retropropagação calcula os vetores gradiente em relação aos pesos dessas camadas** através da **regra da cadeia**.
- Vejamos o exemplo abaixo com 3 nós e pesos das ligações iguais a 1.
 - **OBS.:** As funções $f(\cdot)$, $g(\cdot)$, e $h(\cdot)$ podem ser interpretadas como sendo as funções de ativação dos nós.



- Como calculamos a derivada de y em relação à x ?

$$\frac{\partial y}{\partial x} = \frac{\partial f(g(h(x)))}{\partial x} = \frac{\partial f(g(h(x)))}{\partial g(h(x))} \frac{\partial g(h(x))}{\partial h(x)} \frac{\partial h(x)}{\partial x}.$$

Regra da cadeia

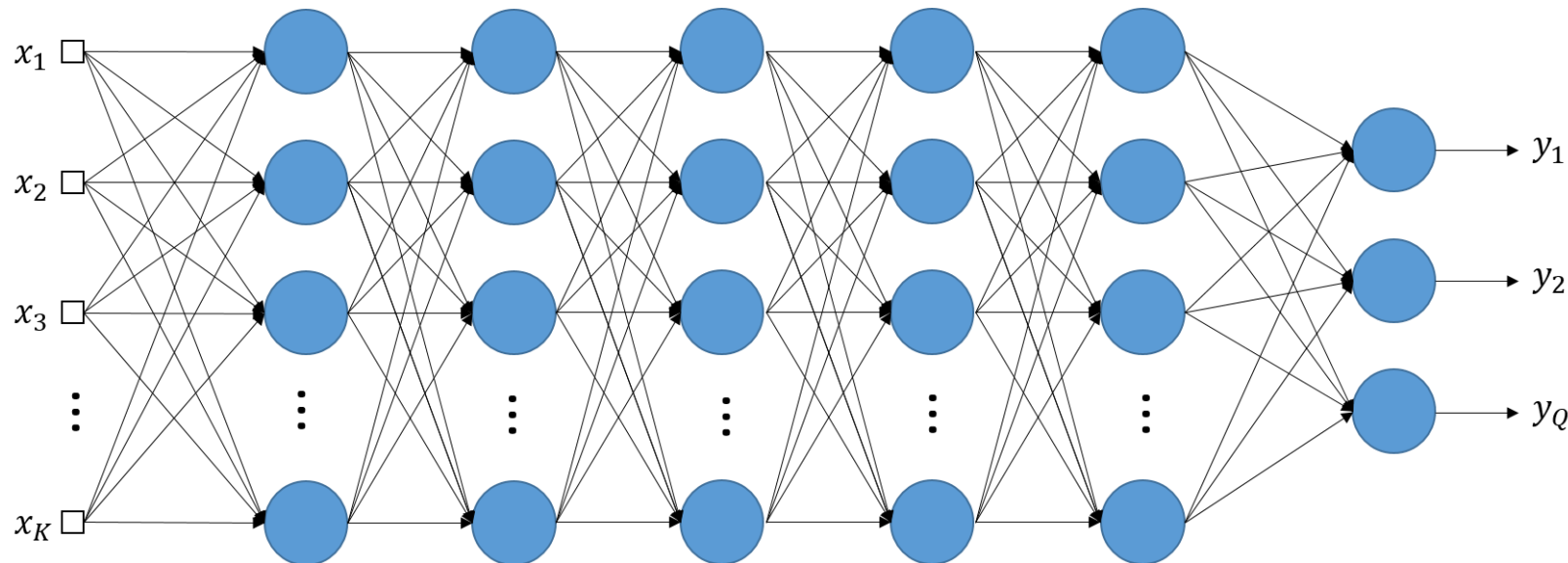
- Em outras palavras, devido à regra da cadeia, o **vetor gradiente** para a **atualização dos pesos de uma dada camada** da rede **inclui o produto das derivadas das funções de ativação dos nós desde a camada de saída até a camada desejada**.

$$\frac{\partial y}{\partial x} = \frac{\partial f(g(h(x)))}{\partial x} = \frac{\partial f(g(h(x)))}{\partial g(h(x))} \frac{\partial g(h(x))}{\partial h(x)} \frac{\partial h(x)}{\partial x}.$$

- Lembrem-se que as **funções de ativação**, como **tangente hiperbólica** ou **logística**, têm derivadas no intervalo de 0 até 1.
- Portanto, a multiplicação de vários termos menores do que 1 tende a 0 conforme o número de camadas da rede aumenta.

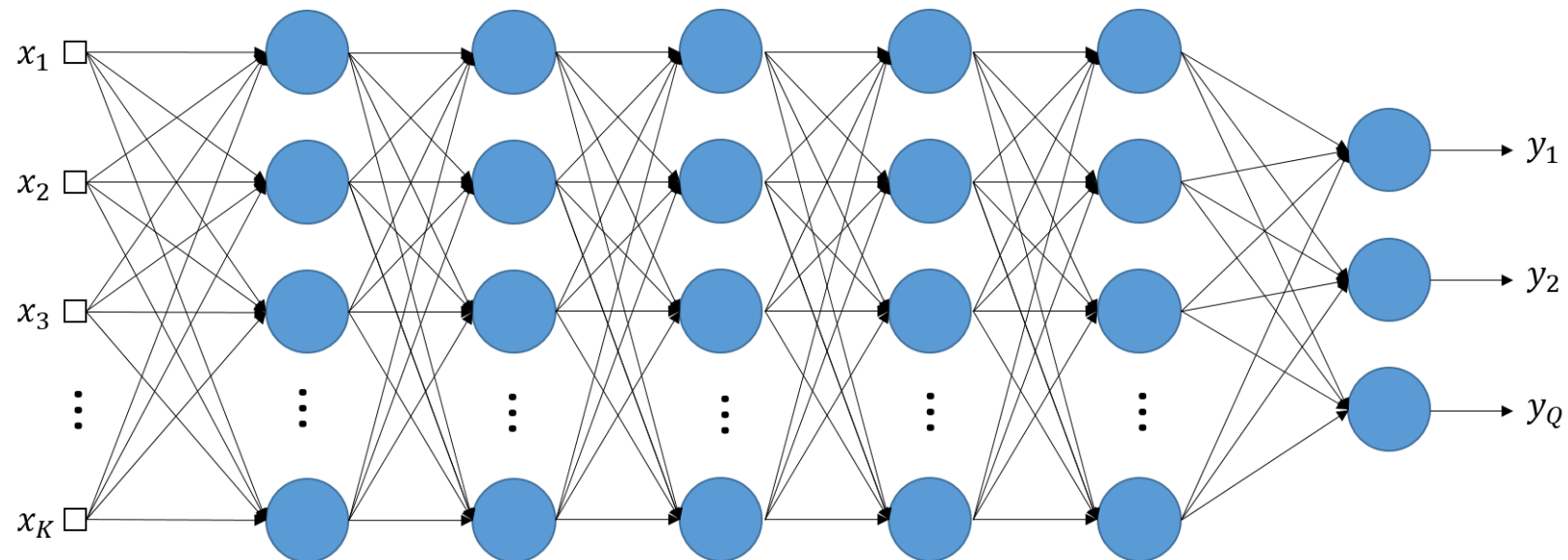
O problema da dissipação do gradiente

- Em uma rede com M camadas, a **retropropagação** tem o efeito de multiplicar até M valores pequenos (i.e., derivadas parciais das funções de ativação) para calcular os vetores gradiente das primeiras camadas.
- O que significa que o **gradiente diminui exponencialmente com M** .

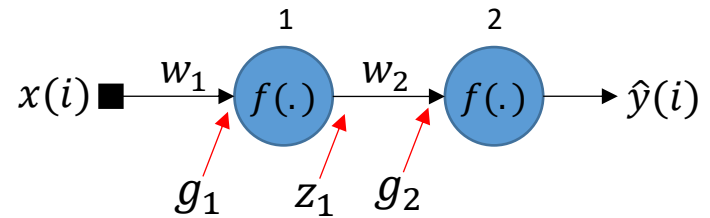


O problema da dissipação do gradiente

- Assim, os **nós das camadas iniciais** aprendem muito mais **lentamente** do que os **nós das camadas finais**, pois o **vetor gradiente** daquelas camadas é **muito pequeno**, fazendo com que a **atualização dos pesos também seja pequena**.
- Vejam um exemplo.



Dissipação do gradiente



Considerações:

- Problema de regressão.
- 2 x neurônios com função de ativação sigmoide, $f(\cdot)$.
- $g_1 = xw_1 \rightarrow$ entrada (i.e., ativação) do primeiro neurônio.
- $z_1 = f(xw_1) \rightarrow$ saída do primeiro neurônio.
- $g_2 = z_1w_2 = f(xw_1)w_2 \rightarrow$ entrada (i.e., ativação) do segundo neurônio.
- $\hat{y} = f(f(xw_1)w_2) \rightarrow$ saída do segundo neurônio.
- **Objetivo:** minimizar o erro quadrático médio, $J_e = \frac{1}{N} \sum_{i=1}^N (\hat{y}(i) - y(i))^2$.

Dissipação do gradiente

- As **regras de atualização** dos dois pesos são dadas por

$$w_2 = w_2 - \alpha \frac{\partial J_e}{\partial w_2},$$

$$w_1 = w_1 - \alpha \frac{\partial J_e}{\partial w_1}.$$

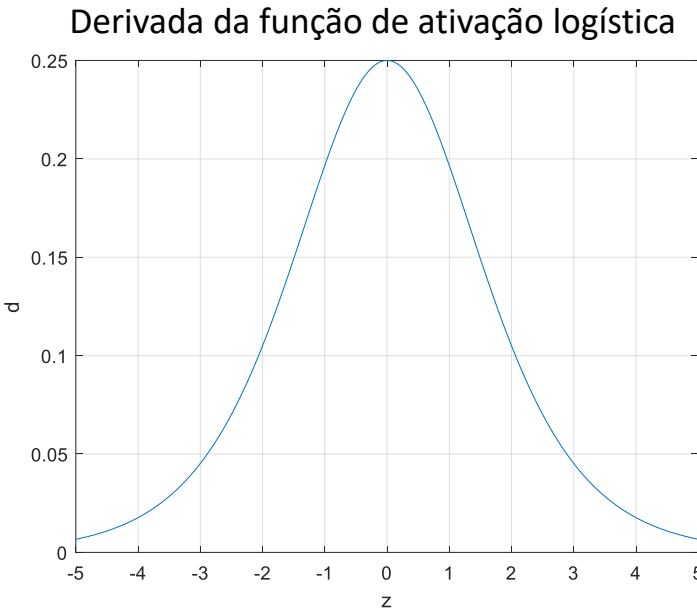
- Usando a regra da cadeia, obtemos as derivadas $\frac{\partial J_e}{\partial w_1}$ e $\frac{\partial J_e}{\partial w_2}$

$$\frac{\partial J_e}{\partial w_2} = \frac{\partial J_e}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial g_2} \frac{\partial g_2}{\partial w_2},$$

$$\frac{\partial J_e}{\partial w_1} = \frac{\partial J_e}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial g_2} \frac{\partial g_2}{\partial z_1} \frac{\partial z_1}{\partial g_1} \frac{\partial g_1}{\partial w_1}.$$

Dissipação do gradiente

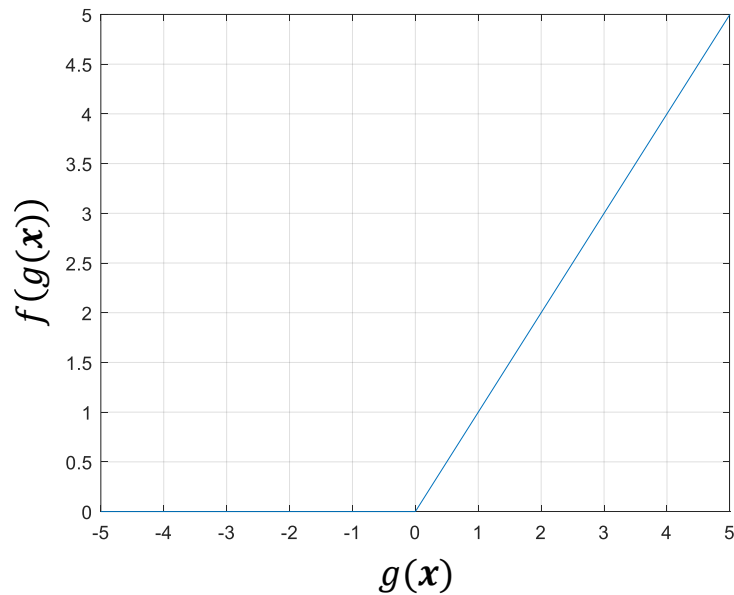
$$\frac{\partial J_e}{\partial w_2} = \frac{\partial J_e}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial g_2} \frac{\partial g_2}{\partial w_2},$$
$$\frac{\partial J_e}{\partial w_1} = \frac{\partial J_e}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial g_2} \frac{\partial g_2}{\partial z_1} \frac{\partial z_1}{\partial g_1} \frac{\partial g_1}{\partial w_1}.$$



- A derivada da função sigmoide é no máximo igual a 0.25.
- Assim, por exemplo, a **primeira camada** de uma **rede neural com M camadas**, terá as derivadas parciais da função de erro em relação a seus pesos compostas pela **multiplicação de M termos no máximo iguais a 0.25**.
- Isso faz com que as **primeiras camadas aprendam lentamente ou nem aprendam**, pois têm **derivadas muito pequenas, tendendo a zero**.

Como mitigar esse problema?

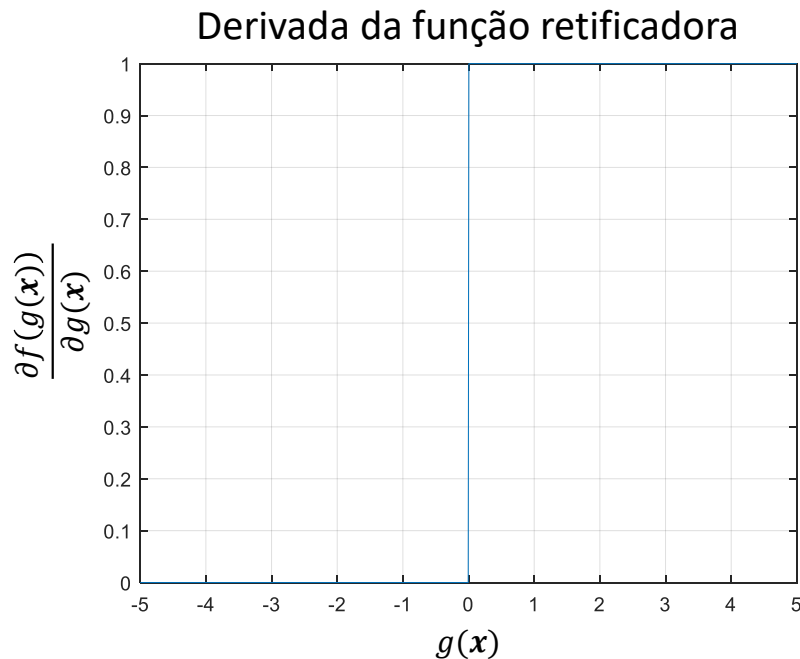
Função de ativação retificadora



$$\hat{y} = f(g(x)) = \max(0, g(x))$$

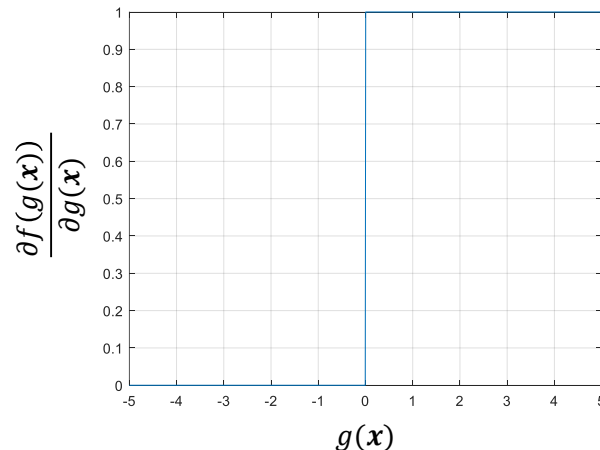
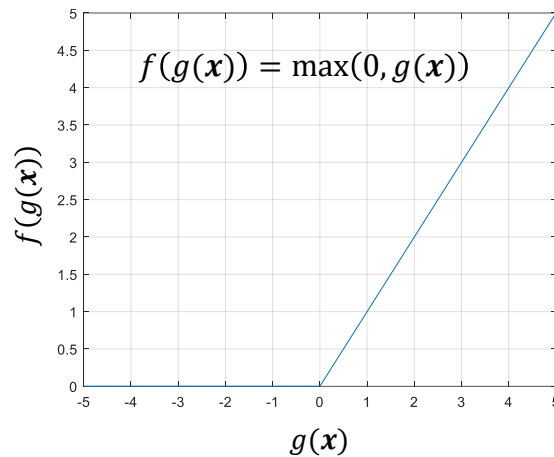
- Com o surgimento das **redes neurais profundas**, e, consequentemente, do problema do **desaparecimento do gradiente**, uma outra função de ativação, conhecida como **Rectified Linear Unit (ReLU)**, passou a ser a bastante utilizada.
- É também uma **função não-linear** onde sua saída é igual 0 quando $g(x) \leq 0$ e o próprio $g(x)$ quando $g(x) > 0$.
- É uma das funções mais amplamente utilizadas em redes neurais profundas.

Função de ativação retificadora



- Suas principais **vantagens** são a sua **simplicidade e eficiência computacional**.
 - Ela e sua derivada **são mais rápidas de se calcular** do que as funções logística e tangente hiperbólica.
- Além disso, ajuda a **minimizar o problema do desaparecimento de gradiente**, pois sua derivada é igual a 1 para $g(x) > 0$.
- Sua derivada é dada por
$$\frac{dy_j}{dg(x)} = \frac{df(g(x))}{dg(x)} = \begin{cases} 0, & \text{se } g(x) < 0 \\ 1, & \text{se } g(x) > 0 \end{cases}$$
- A derivada é indeterminada para $g(x) = 0$.










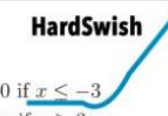



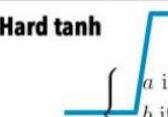
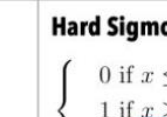



Função de ativação retificadora



- Uma desvantagem é que ela causa o problema conhecido como **ReLU agonizante**.
- Esse **problema ocorre durante o treinamento** da rede, quando a ativação do nó, $g(x)$, é **negativa**.
- Isso faz com que sua **saída e**, consequentemente, a **derivada parcial da função de ativação sejam iguais a 0**.
- Quando isso ocorre, o **nó não tem seus pesos atualizados** durante o treinamento, **permanecendo inalterados**.

Variantes da função de ativação retificadora

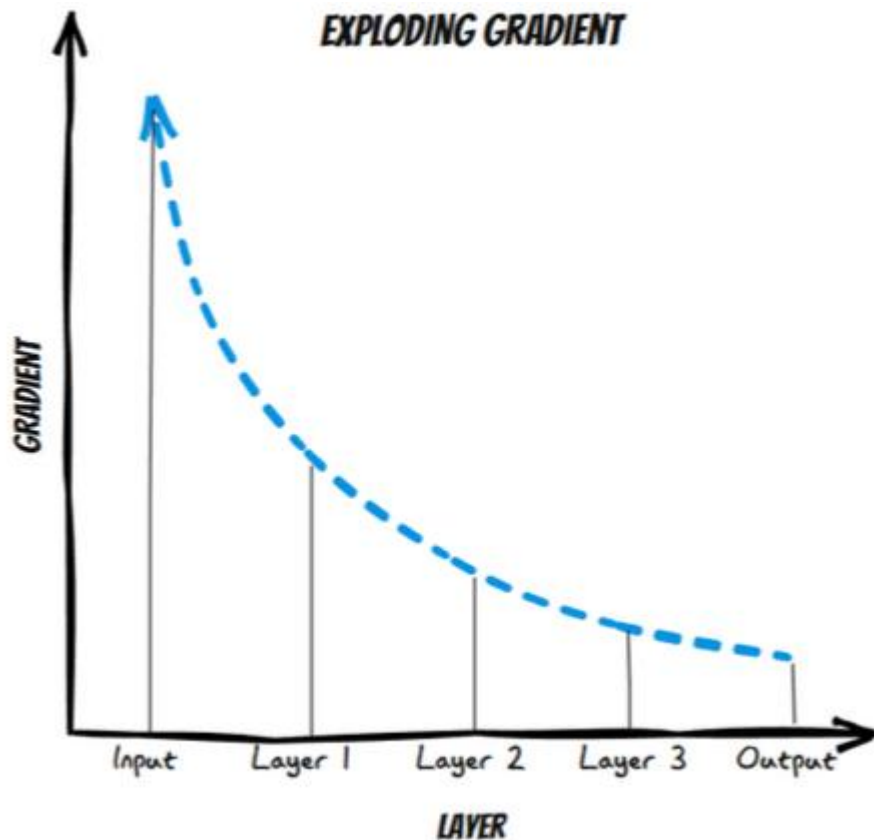
Neural Network Activation Functions: a small subset!

ReLU  $\max(0, x)$	GELU  $\frac{1}{\sqrt{2\pi}} \left(1 + \tanh \left(\sqrt{\frac{2}{\pi}} (x + ax^3) \right) \right)$	PReLU  $\max(0, x)$
ELU  $\begin{cases} x & \text{if } x > 0 \\ \alpha(x \exp x - 1) & \text{if } x < 0 \end{cases}$	Swish  $\frac{x}{1 + \exp -x}$	SELU  $\alpha(\max(0, x) + \min(0, \beta(\exp x - 1)))$
SoftPlus  $\frac{1}{\beta} \log(1 + \exp(\beta x))$	Mish  $x \tanh \left(\frac{1}{\beta} \log(1 + \exp(\beta x)) \right)$	RReLU  $\begin{cases} x & \text{if } x \geq 0 \\ ax & \text{if } x < 0 \text{ with } a \sim \mathcal{R}(l, u) \end{cases}$
HardSwish  $\begin{cases} 0 & \text{if } x \leq -3 \\ x & \text{if } -3 < x < 3 \\ (x+3)/6 & \text{otherwise} \end{cases}$	Sigmoid  $\frac{1}{1 + \exp(-x)}$	SoftSign  $\frac{x}{1 + x }$
Tanh  $\tanh(x)$	Hard tanh  $\begin{cases} a & \text{if } x \geq a \\ b & \text{if } x \leq b \\ x & \text{otherwise} \end{cases}$	Hard Sigmoid  $\begin{cases} 0 & \text{if } x \leq -3 \\ 1 & \text{if } x \geq 3 \\ x/6 + 1/2 & \text{otherwise} \end{cases}$
Tanh Shrink  $x - \tanh(x)$	Soft Shrink  $\begin{cases} x - \lambda & \text{if } x > \lambda \\ x + \lambda & \text{if } x < -\lambda \\ 0 & \text{otherwise} \end{cases}$	Hard Shrink  $\begin{cases} x & \text{if } x > \lambda \\ x & \text{if } x < -\lambda \\ 0 & \text{otherwise} \end{cases}$

- Para resolver o problema das **ReLU**s *agonizantes*, usa-se **variantes da função ReLU** que **possuam derivada diferente de zero para $g(x) < 0$** , como, por exemplo,

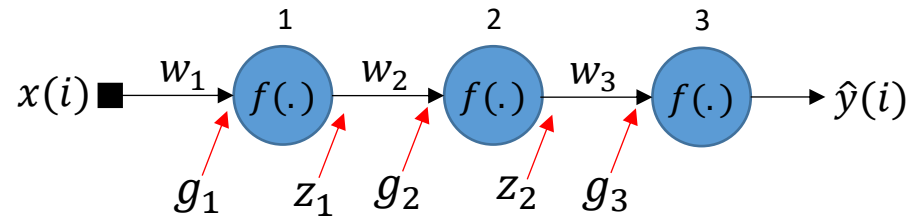
- [Leaky ReLU](#),
- [Parametric ReLU \(PReLU\)](#),
- [Gaussian Error Linear Unit \(GELU\)](#),
- [etc.](#)

Explosão do gradiente



- Usando funções de ativação ReLU, reduzimos o problema do desaparecimento do gradiente.
- Porém, um **outro problema surge** quando as **ativações são positivas** e os **pesos têm valores maiores do que 1**.
- Caso os pesos sejam inicializados (em geral, de forma aleatória) com **valores maiores do que 1**, haverá a **multiplicação de vários valores assim**, **resultando em valores de gradiente muito grandes nas camadas iniciais**.

Explosão do gradiente



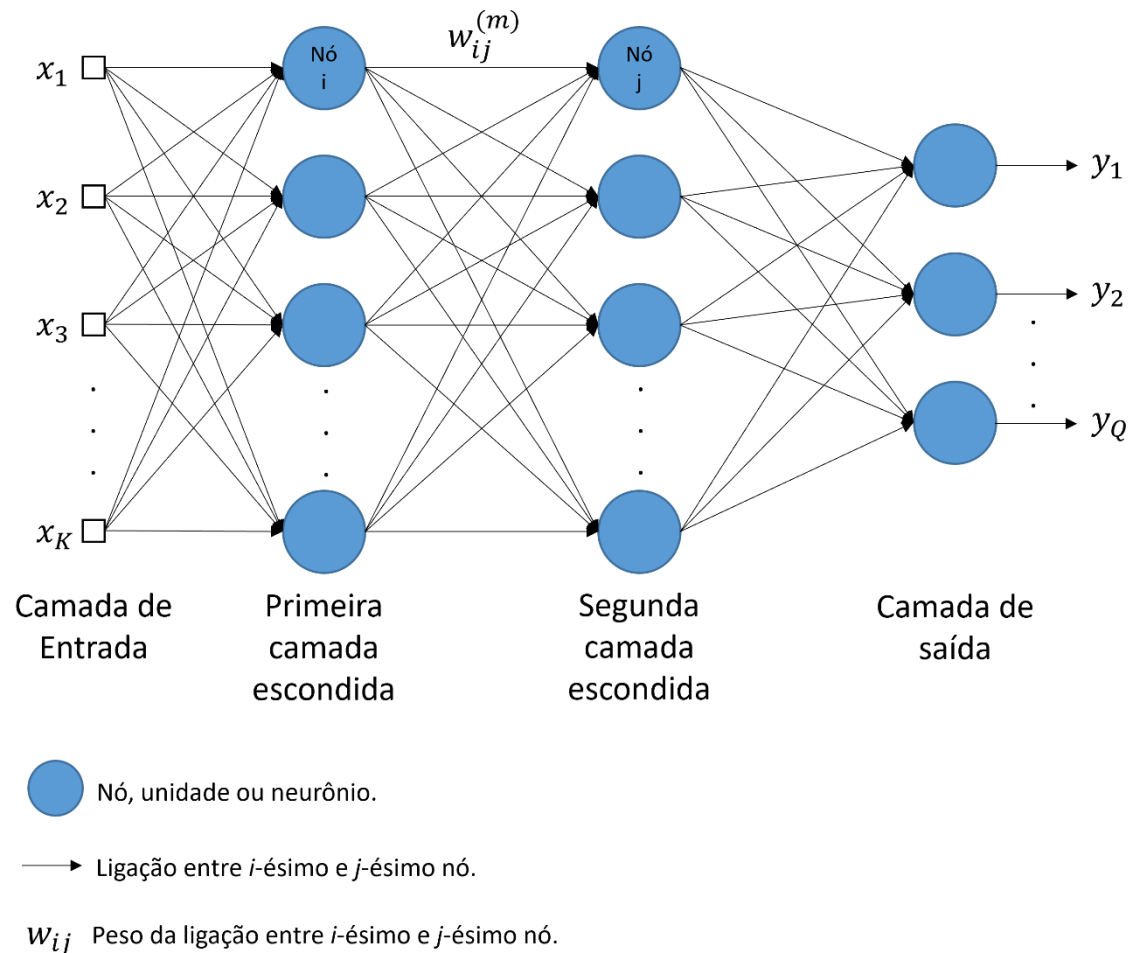
$$\frac{\partial J_e}{\partial w_1} = \frac{\partial J_e}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial g_3} \frac{\partial g_3}{\partial z_2} \frac{\partial z_2}{\partial g_2} \frac{\partial g_2}{\partial z_1} \frac{\partial z_1}{\partial g_1} \frac{\partial g_1}{\partial w_1}.$$
$$\frac{\partial g_3}{\partial z_2} = \frac{\partial f(g_2)w_3}{\partial f(g_2)} = w_3$$
$$\frac{\partial g_2}{\partial z_1} = \frac{\partial f(xw_1)w_2}{\partial f(xw_1)} = w_2$$

- Se os elementos do vetor gradiente tiverem **magnitudes muito grandes**, os **pesos da rede podem sofrer atualizações extremamente grandes**, o que leva a **instabilidades numéricas** e a um **treinamento ineficaz** ou até mesmo à **divergência**.

Formas de se minimizar a dissipação e a explosão do gradiente

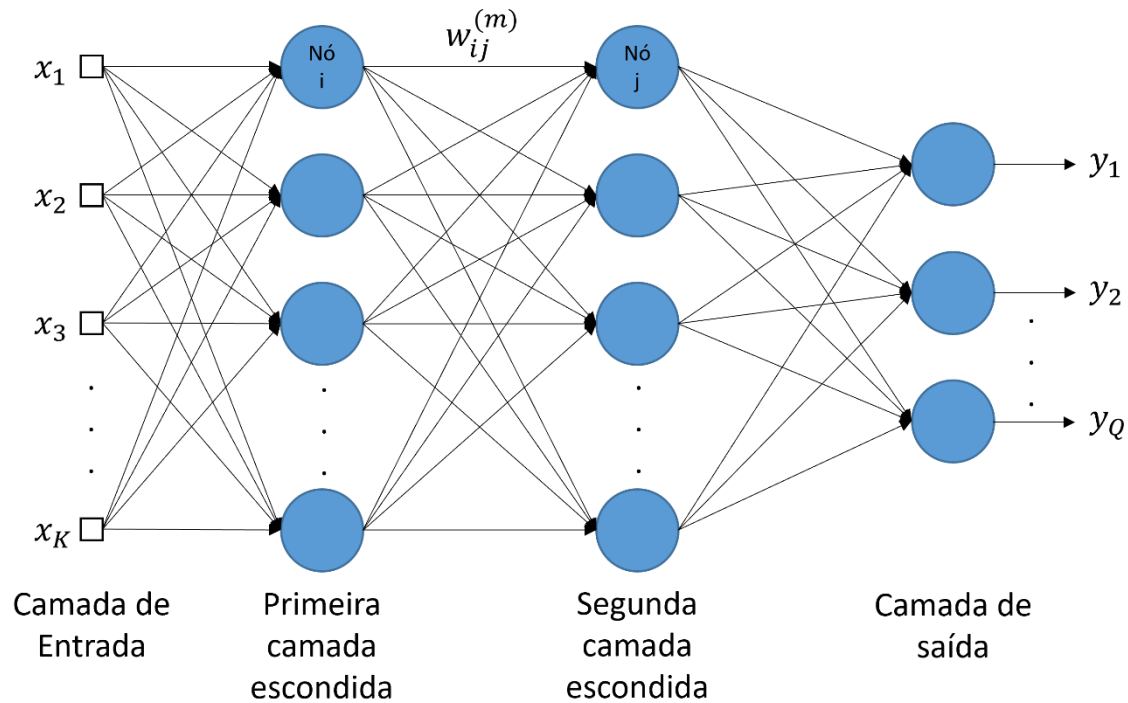
- Além do uso de funções de ativação ReLU ou de suas variantes, outras formas de se minimizar esses problemas são:
 - **Inicialização apropriada dos pesos:** garante que a *média seja zero e a variância das ativações permaneça a mesma ao longo de todas as camadas da rede*. Isso garante que o gradiente retropropagado não tenha multiplicações com valores muito pequenos ou muito grandes em qualquer camada, ajudando a *mitigar ambos os problemas*.
 - **Normalização de batch:** *padroniza as ativações* das camadas da rede e, na sequência, as *desloca e escalona*, mantendo-as dentro de intervalos que *minimizam ambos os problemas*.
 - **Poda do gradiente:** *limita (poda) os valores dos gradientes* durante o treinamento para que eles não excedam algum limite pré-definido, *mitigando apenas o problema da explosão do gradiente*.

Conectando neurônios



- Os neurônios de uma rede neural podem ser conectados de forma **acíclica** ou **cíclica**.
- O termo **acíclico** se refere a conexões **sem realimentação**.
- Isso significa que a **informação flui em uma única direção**, da camada de entrada para a camada de saída.
- A rede ao lado tem conexões **acíclicas** e é conhecida como **rede densa de alimentação direta**.

Conectando neurônios



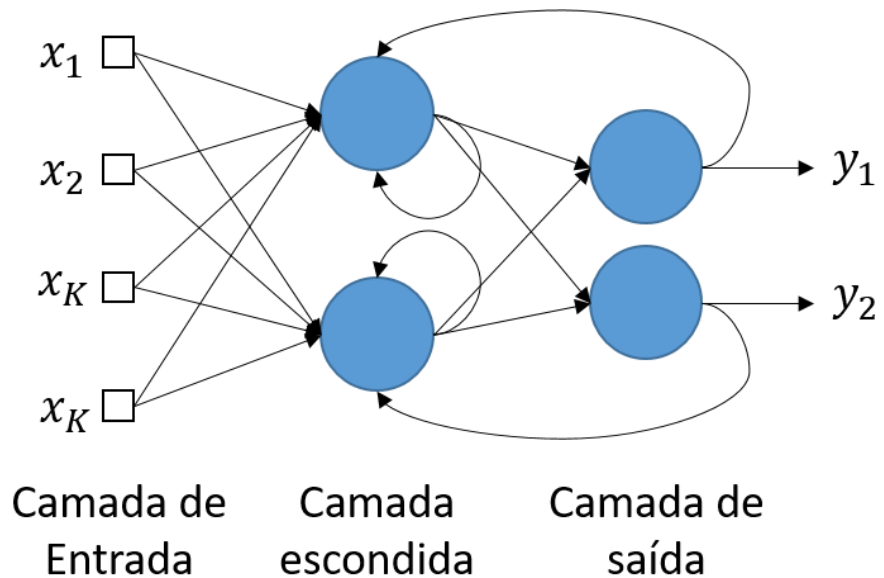
● Nó, unidade ou neurônio.

→ Ligação entre i -ésimo e j -ésimo nó.

w_{ij} Peso da ligação entre i -ésimo e j -ésimo nó.

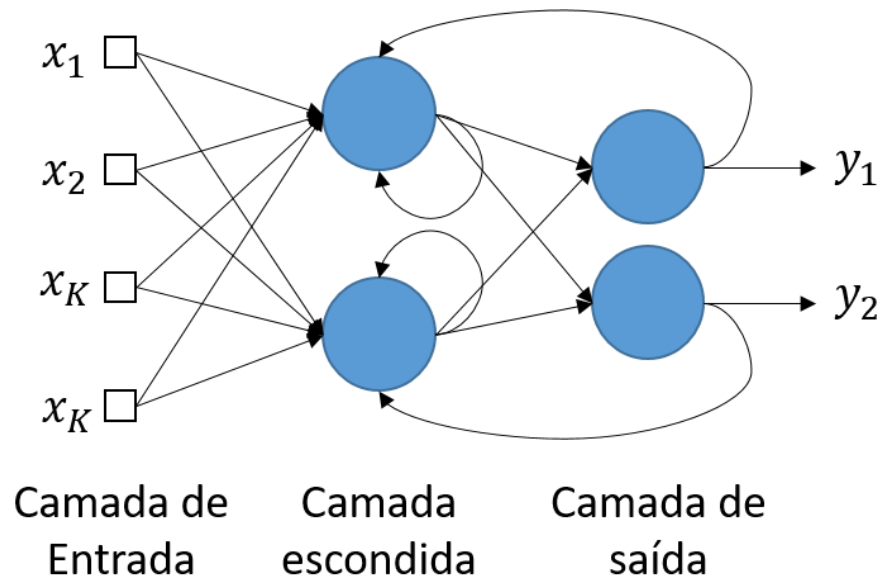
- Esse tipo de rede representa uma ***função de suas entradas e pesos atuais***
$$\mathbf{y} = f(\mathbf{x}; \mathbf{W}),$$
onde \mathbf{W} é a matriz contendo todos os pesos da rede.
- Portanto, este tipo de rede ***não possui um estado interno, ou seja, não tem memória.***

Conectando neurônios



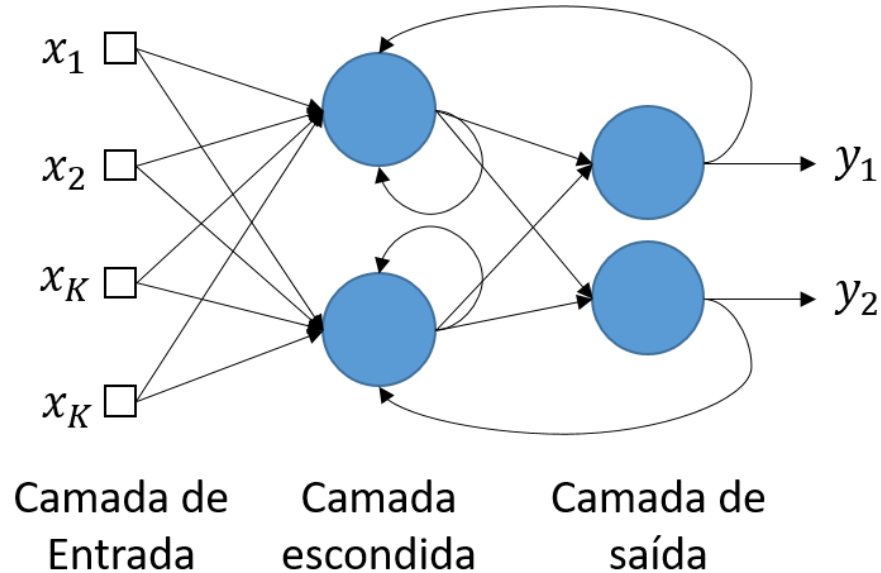
- Já o termo **cíclico** se refere a **conexões que formam ciclos**, permitindo a **realimentação de informações**.
- Redes com esse tipo de conexão são conhecidas como **redes recorrentes** ou **redes com realimentação**.
- A figura mostra que os nós da rede têm **conexões em duas direções**, desta forma, o **sinal percorre a rede nas direções direta e reversa**.

Conectando neurônios



- A saída da rede é *função de suas entradas e pesos atuais e de seus estados anteriores*, ou seja, de saídas anteriores.
- Esse tipo de rede forma um *sistema dinâmico* que pode atingir
 - um estado estável,
 - exibir oscilações
 - ou mesmo um comportamento caótico e divergir.

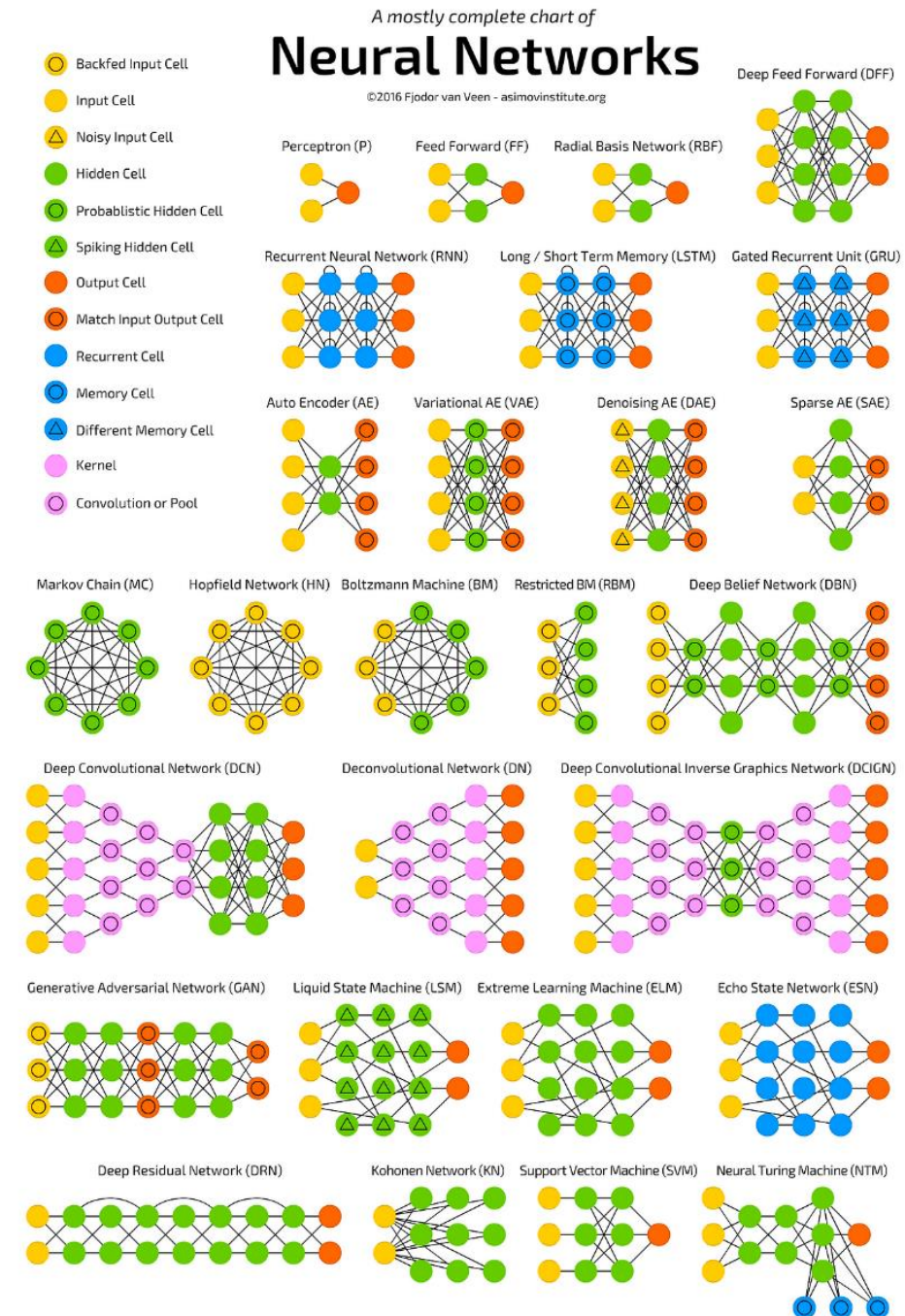
Conectando neurônios



- Portanto, **redes recorrentes** possuem **memória**.
- Essas redes são úteis em tarefas que envolvem **dependências temporais** como
 - Previsões de séries temporais (e.g., monitoramento de sinais vitais, preço de ações, etc.) e
 - Processamento de linguagem natural (e.g., conversão de fala em texto, reconhecimento de palavras, respostas a perguntas, etc.).

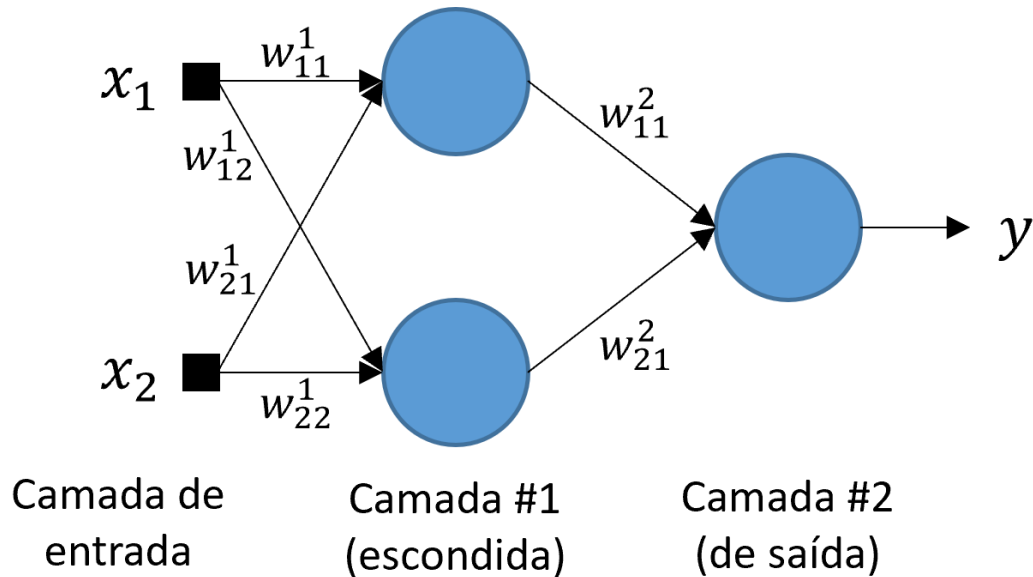
Arquiteturas de redes neurais

- Hoje em dia, existem diversas outras formas de conexão entre camadas e nós que dão origem a uma gama imensa de arquiteturas de redes neurais.
- Um compilado dessas arquiteturas pode ser encontrado em
 - <https://www.asimovinstitute.org/author/fjodorvanveen/>



Aproximação de funções com redes neurais

Aproximação de funções com redes neurais



- A rede MLP da figura ao lado tem sua saída definida por

$$y = f(\mathbf{w}^T f(\mathbf{W}^T \mathbf{x})),$$

onde $f(\cdot)$ é a **função de ativação** escolhida para todos os nós, $\mathbf{W} = \begin{bmatrix} w_{11}^1 & w_{12}^1 \\ w_{21}^1 & w_{22}^1 \end{bmatrix}$,

$$\mathbf{w} = \begin{bmatrix} w_{11}^2 \\ w_{21}^2 \end{bmatrix} \text{ e } \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

- Percebam que a saída da rede é dada pelo **aninhamento** das saídas de **funções de ativação não-lineares**.

Aproximação de funções com redes neurais

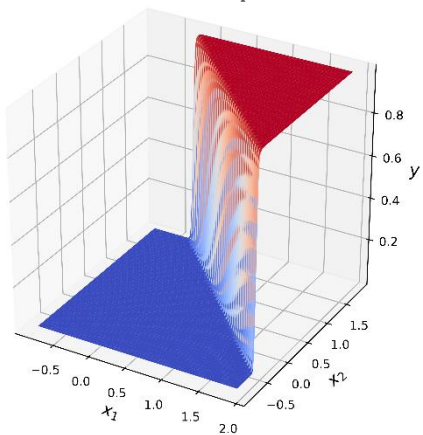
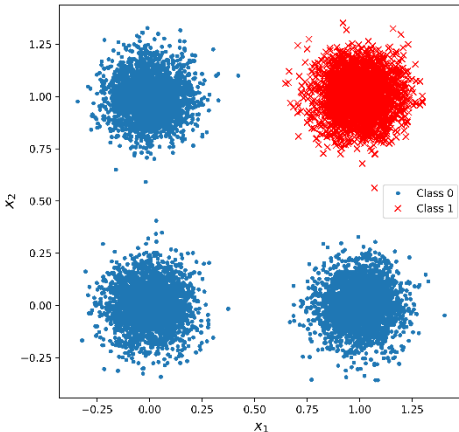
- Portanto, as redes neurais têm a **capacidade de aproximar funções altamente não-lineares**.
- Essa capacidade **depende da sua arquitetura**, incluindo o número de camadas, o número de nós (que corresponde à quantidade de pesos) e as funções de ativação empregadas.
 - A **quantidade de pesos** de uma rede está associada aos seus **graus de liberdade**, ou seja, a **capacidade da rede de aproximar diferentes tipos de funções**.
- Portanto, assim como polinômios, que podem **aproximar qualquer tipo de função** (linear ou não linear) devido a seus **graus de liberdade**, as **redes neurais podem fazer o mesmo**, bastando apenas que **encontremos sua complexidade ideal**, ou seja, sua arquitetura.

Aproximação de funções com redes neurais

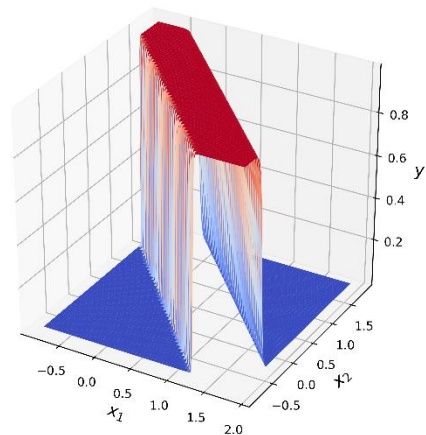
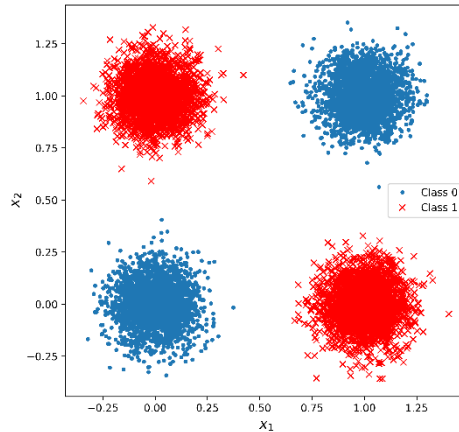
- Por exemplo, uma rede neural com *uma camada oculta* com um número *suficientemente grande de nós* pode *aproximar* praticamente qualquer *função contínua*.
- Com *duas camadas ocultas*, até *funções descontínuas* podem ser *aproximadas*.
- Portanto, dizemos que as redes neurais possuem *capacidade de aproximação universal* de funções.
- Desta forma, as redes neurais podem resolver *problemas de regressão e classificação* e uma grande gama de outros problemas.
- O desafio é encontrar a arquitetura ideal para a aproximação.
- Veremos alguns exemplos desta capacidade de aproximação a seguir.

Aproximação universal de funções em problemas de classificação

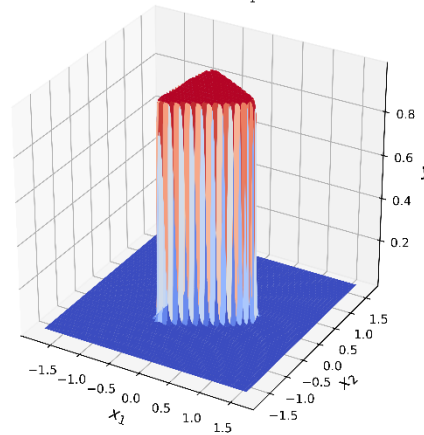
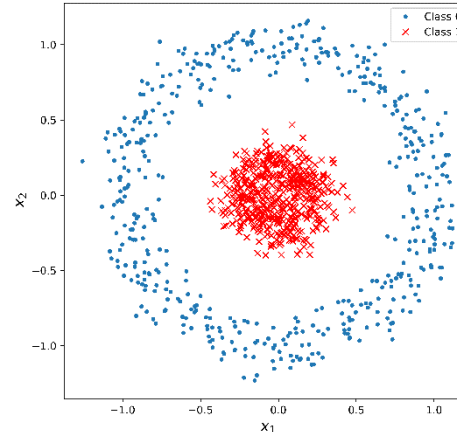
Função AND: MLP sem camada escondida, com apenas um neurônio na camada de saída.
Total: 1 nó.



Função XOR: MLP com 1 camada escondida com 2 nós mais 1 nó na camada de saída.
Total: 3 nós.



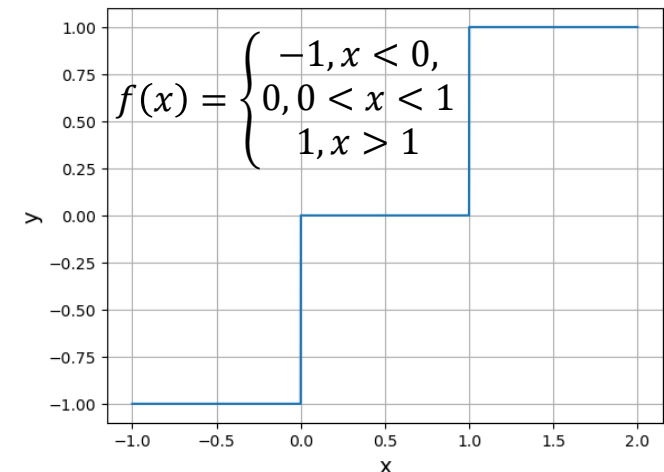
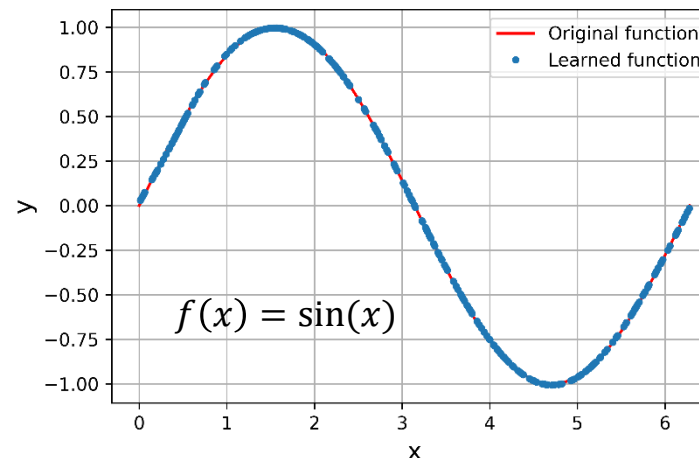
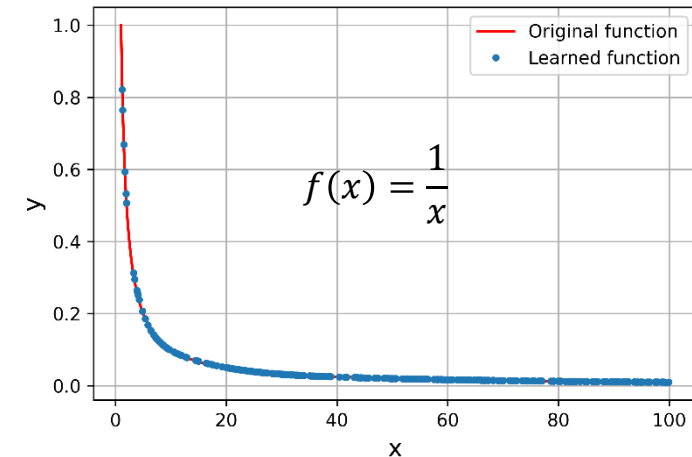
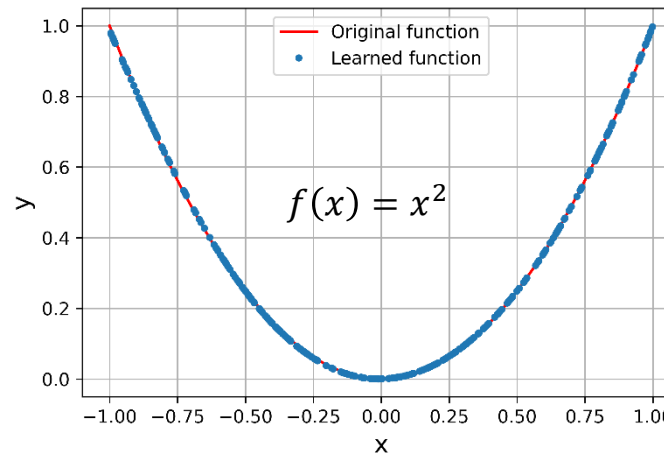
Círculos concêntricos: MLP com 1 camada escondida com 4 nós mais 1 nó na camada de saída.
Total: 5 nós.



- Fig. 1: Um nó aproxima uma função de limiar suave.
- Fig. 2: Combinando duas funções de limiar suave com direções opostas, podemos obter uma função com formato de onda.
- Fig. 3: Combinando duas ondas perpendiculares, nós obtemos uma função com formato triangular.

Aproximação universal de funções em problemas de regressão

- Redes neurais podem ser usadas para aproximar funções como as apresentadas abaixo:
 - $f(x) = x^2, -1 \leq x \leq 1,$
 - $f(x) = \frac{1}{x}, 1 \leq x \leq 100,$
 - $f(x) = \sin(x), 1 \leq x \leq 2\pi,$
 - $f(x) = \begin{cases} -1, & x < 0, \\ 0, & 0 < x < 1 \\ 1, & x > 1 \end{cases}.$



Para casa

1. Use as classes [MLPRegressor](#) e [GridSearchCV](#) da biblioteca SciKit-Learn para encontrar o número de camadas escondidas e nós necessários para que uma rede neural aproxime as funções apresentadas no slide anterior.
 - OBS.: Uma única camada oculta é capaz de aproximar qualquer função contínua. Já duas ou mais conseguem aproximar funções com descontinuidade.
2. Use as classes [MLPClassifier](#) e [GridSearchCV](#) e a função [load_digits](#) da biblioteca SciKit-Learn para encontrar o número de camadas escondidas e nós necessários para classificar imagens de dígitos escritos à mão.
 - OBS.: Ao invocar a função ***load_digits***, configure o parâmetro ***return_X_y*** como ***True***.

Como as redes neurais
aprendem?

Aprendizado em redes neurais

- O **processo de atualização dos pesos** de uma rede neural corresponde a um **problema de minimização** de uma **função de erro** (de perda ou de custo), $J(W)$, com relação aos pesos da rede neural.
 - W representa uma *array* contendo todos os pesos da rede neural.
- Assim, o problema do aprendizado em redes neurais pode ser formulado como

$$\min_W J(W)$$

- Esse processo de otimização é **conduzido de forma iterativa**, o que dá um **sentido mais natural à noção de aprendizado** (i.e., um processo gradual).
- Os **métodos de otimização mais utilizados** são os **baseados nas derivadas da função de erro**, $J(W)$.

Aprendizado em redes neurais

- Dentre esses métodos, existem os de **primeira** e os de **segunda ordem**.
- Métodos de **primeira ordem** são baseados nas **derivadas parciais de primeira ordem** da **função de erro** e usam versões da seguinte **equação de atualização dos pesos**

$$\mathbf{w}(k + 1) \leftarrow \mathbf{w}(k) - \alpha \nabla J(\mathbf{w}(k)),$$

onde \mathbf{w} é o vetor de pesos, $\nabla J(\mathbf{w}(k))$ é o vetor gradiente da forma

$\left[\frac{\partial J(\mathbf{w}(k))}{\partial w_0} \quad \frac{\partial J(\mathbf{w}(k))}{\partial w_1} \quad \dots \quad \frac{\partial J(\mathbf{w}(k))}{\partial w_{K+1}} \right]^T \in \mathbb{R}^{K+1 \times 1}$, α é o passo de aprendizagem e k é a iteração de atualização.

- O gradiente descente e suas várias versões, além das variantes adaptativas e do termo momentum, são exemplos de métodos de primeira ordem.

Aprendizado em redes neurais

- Já os métodos de *segunda ordem*, além das informações de primeira ordem, utilizam informações fornecidas pelas *derivadas parciais de segunda ordem* da *função de erro*.
- Essa informação está contida na *matriz Hessiana*, $H(\mathbf{w})$:

$$H(\mathbf{w}(k)) = \nabla^2 J(\mathbf{w}(k)) = \begin{bmatrix} \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_0^2} & \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_0 \partial w_1} & \dots & \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_0 \partial w_{K+1}} \\ \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_1 \partial w_0} & \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_1^2} & \dots & \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_1 \partial w_{K+1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_{K+1} \partial w_0} & \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_{K+1} \partial w_1} & \dots & \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_{K+1}^2} \end{bmatrix} \in \mathbb{R}^{K+1 \times K+1}.$$

Aprendizado em redes neurais

- Usando uma aproximação de Taylor de segunda ordem da **função de erro**, resulta na seguinte **equação de atualização dos pesos**

$$\mathbf{w}(k + 1) \leftarrow \mathbf{w}(k) - \alpha \mathbf{H}^{-1}(\mathbf{w}(k)) \nabla J(\mathbf{w}(k)).$$

- Essa expressão requer que a **matriz Hessiana** seja **invertível** e **definida positiva** a cada iteração, k , i.e., $\mathbf{z}^T \mathbf{H} \mathbf{z} > 0, \forall \mathbf{z} \neq \mathbf{0}$ (vetor nulo).
- A **atualização dos pesos** utilizando informações de primeira e de segunda ordem **é mais precisa** do que a fornecida por métodos de primeira ordem.
- Portanto, métodos de **segunda ordem convergem mais rapidamente** do que métodos de **primeira ordem**.

Aprendizado em redes neurais

$$\mathbf{H}(\mathbf{w}(k)) = \begin{bmatrix} \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_0^2} & \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_0 \partial w_1} & \dots & \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_0 \partial w_{K+1}} \\ \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_1 \partial w_0} & \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_1^2} & \dots & \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_1 \partial w_{K+1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_{K+1} \partial w_0} & \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_{K+1} \partial w_1} & \dots & \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_{K+1}^2} \end{bmatrix}$$

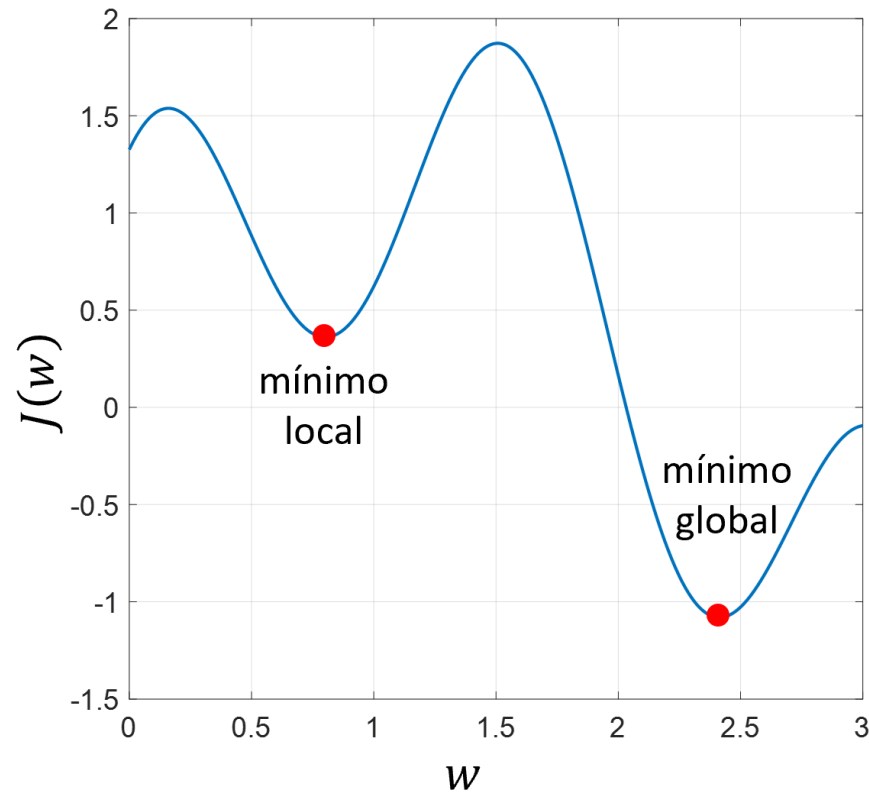
- Entretanto, o cálculo exato da **matriz Hessiana** pode ser **custoso computacionalmente** em vários casos práticos.
 - Por exemplo, se tivermos $K = 10$ pesos para otimizar, precisamos calcular $10 \times 10 = 100$ derivadas parciais para formar a matriz Hessiana.
 - Além disso, ela precisa ser invertida, o que tem complexidade cúbica, $O(K^3)$.
 - Portanto, essa abordagem direta não é eficiente se o número de pesos for muito grande, o que é o caso quando se usa redes neurais profundas.

Aprendizado em redes neurais

$$\mathbf{H}(\mathbf{w}(k)) = \begin{bmatrix} \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_0^2} & \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_0 \partial w_1} & \cdots & \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_0 \partial w_{K+1}} \\ \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_1 \partial w_0} & \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_1^2} & \cdots & \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_1 \partial w_{K+1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_{K+1} \partial w_0} & \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_{K+1} \partial w_1} & \cdots & \frac{\partial^2 J(\mathbf{w}(k))}{\partial w_{K+1}^2} \end{bmatrix}$$

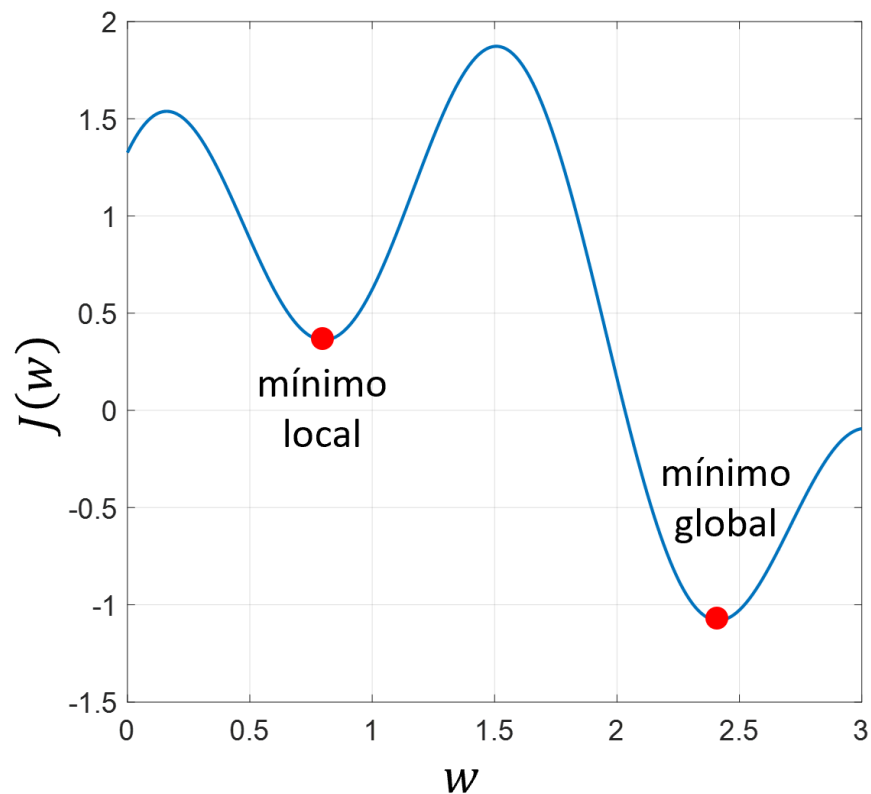
- Porém, há um **conjunto de métodos** de segunda ordem que **evitam esse cálculo direto**, como os métodos **quasi-Newton** ou os métodos de **gradiente escalonado**, os quais **aproximam a matriz Hessiana**.
- O algoritmo *limited-memory Broyden-Fletcher-Goldfarb-Shanno* (LBFGS) é um exemplo de método **quasi-Newton** implementado pela biblioteca *SciKit-Learn* em algumas de suas classes.

Superfícies de erro irregulares



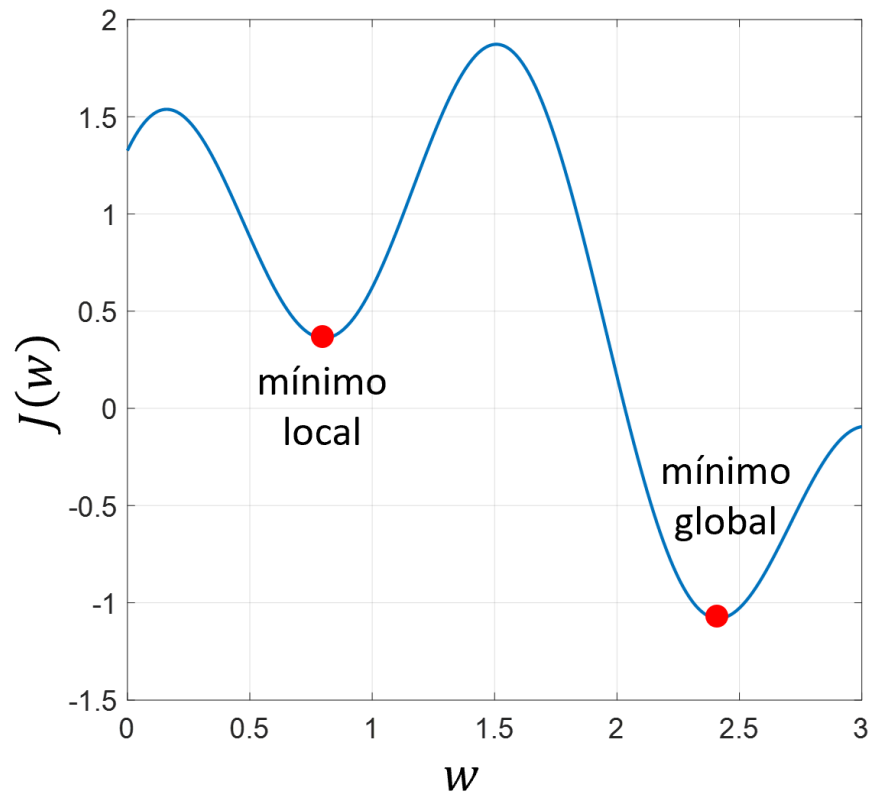
- Todos os métodos que acabamos de discutir são métodos de *busca local*, ou seja, eles *buscam uma solução nas proximidades de onde se encontram*.
- Consequentemente, a *convergência para um mínimo global não é assegurada*.

Superfícies de erro irregulares



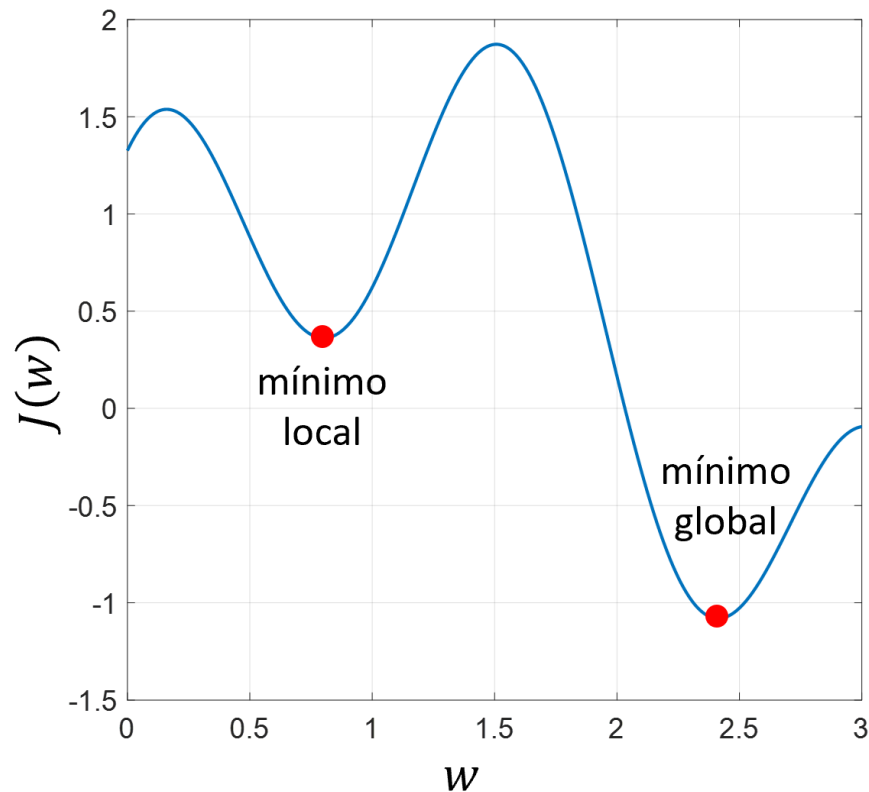
- Portanto, dependendo de onde o algoritmo é ***inicializado***, ele pode ***convergir para um mínimo local***.
- A figura apresenta dois mínimos:
 - ***Mínimo local***: é uma ***solução ótima apenas em relação aos seus vizinhos***.
 - ***Mínimo global***: é uma ***solução ótima em relação a todo o domínio da função de erro***.

Superfícies de erro irregulares



- Por serem formadas pela **combinação de vários nós com funções de ativação não-lineares**, as superfícies de erro de redes neurais **não são convexas**, ou seja, são **altamente irregulares**, **podendo conter vários mínimos locais**.

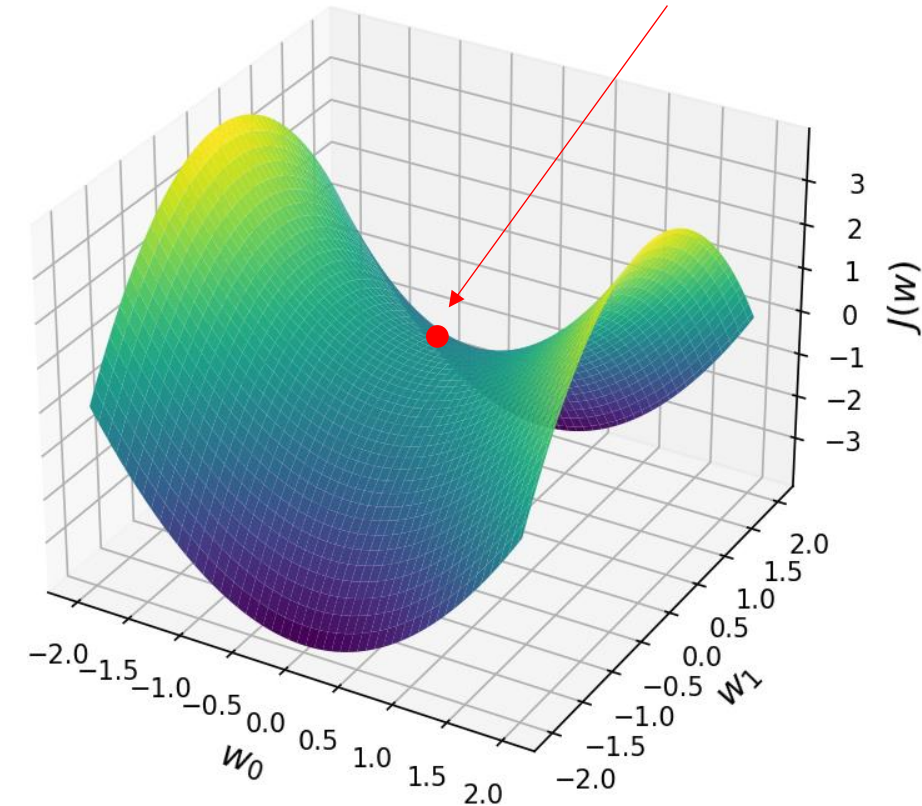
Superfícies de erro irregulares



- Entretanto, felizmente, em muitos problemas envolvendo redes neurais, *quase todos os mínimos locais têm valor de erro próximo ao do mínimo global* e, portanto, encontrar um mínimo local já é bom o suficiente para um dado problema.
- Além dos mínimos locais e global, as superfícies de erro de redes neurais podem apresentar outras *irregularidades que dificultam seu aprendizado*.

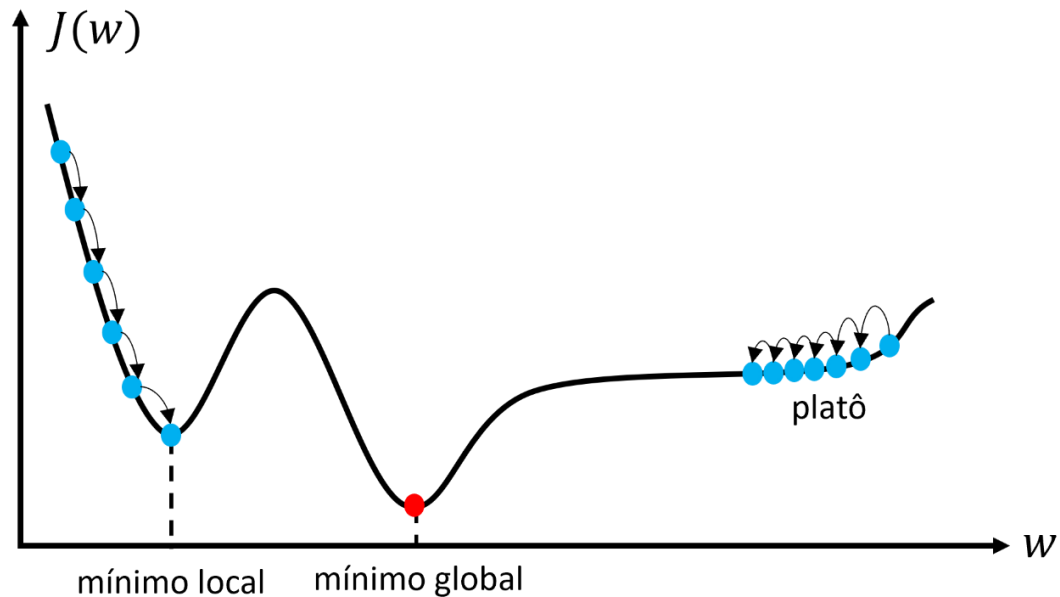
Superfícies de erro irregulares

Superfície com Ponto de Sela



- Uma irregularidade que pode ser encontrada são os ***pontos de sela***:
 - É um ponto que é um **mínimo ao longo de um eixo**, mas um **máximo ao longo de outro**.
 - Em algumas direções são **atratores** (i.e., alta declividade), mas em outras não.
- O algoritmo de otimização pode passar um longo período de tempo sendo atraído por eles, o que prejudica seu desempenho.
- Para escapar destes pontos, usa-se métodos de ***segunda ordem, versões estocásticas (i.e., ruidosas) do gradiente descendente ou reinicialização dos pesos***.

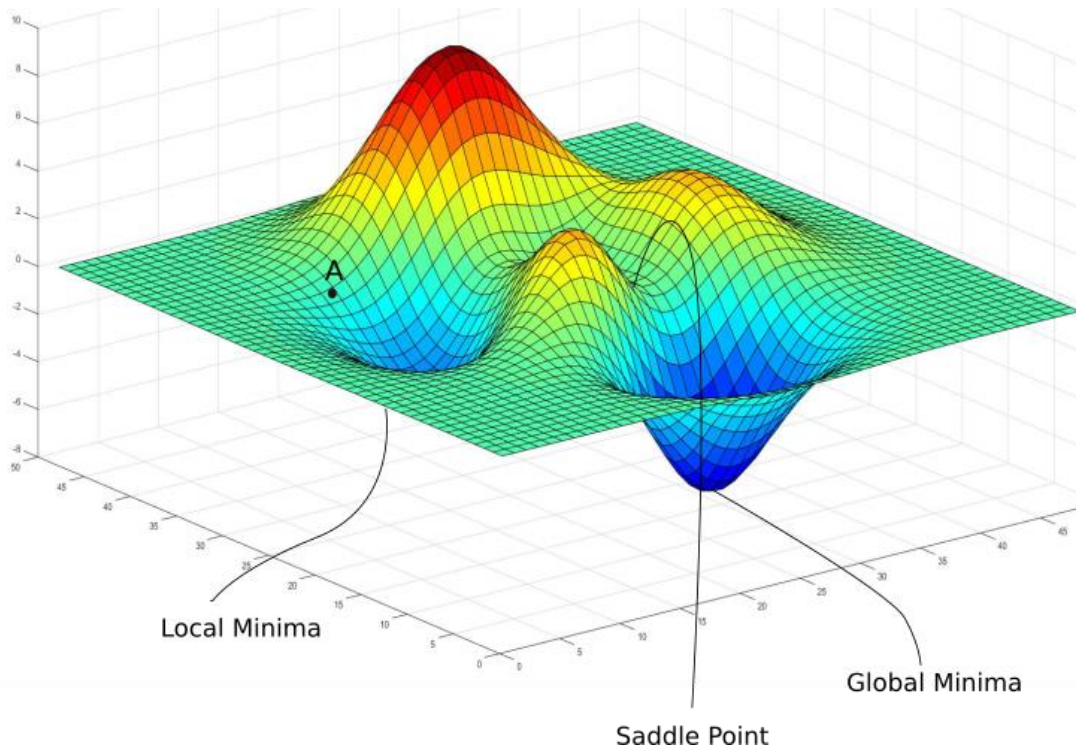
Superfícies de erro irregulares



- Outro tipo de irregularidade são os **platôs**.
- Eles são **regiões planas e com erro elevado**.
- Como a **inclinação da superfície** nessa região é **próxima de zero** (i.e., o gradiente é próximo de zero) o algoritmo pode levar muito tempo para atravessá-la.
- Métodos com **termo momentum** ou de **aprendizado adaptativo**, como AdaGrad, RMSProp, Adam, podem escapar destas regiões.

Superfícies de erro irregulares

Exemplo da superfície de erro de uma rede neural



- Portanto, como garantir que o mínimo encontrado é bom o suficiente?
- Treina-se o modelo várias vezes, sempre *inicializando os pesos de forma aleatória*, com a esperança de que em alguma dessas vezes ele inicialize mais *próximo do mínimo global ou de um bom mínimo local*.
- Adicionalmente, pode-se usar a técnica da parada antecipada para armazenar o melhor conjunto de pesos.

Como atualizamos os pesos dos neurônios de uma RNA?

Retropropagação do erro

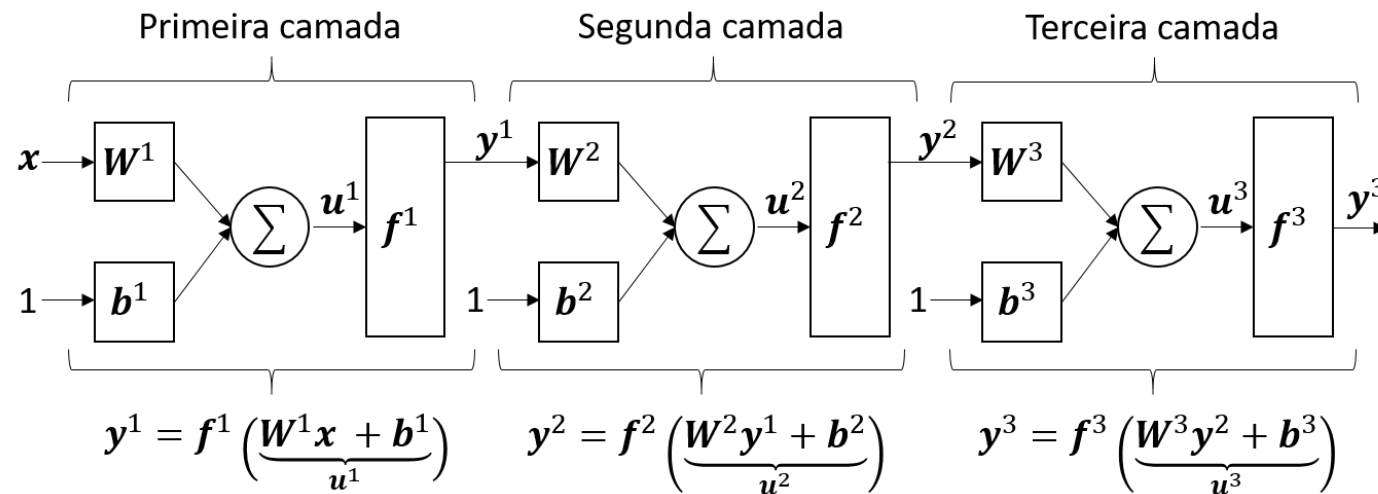
- Conforme nós discutimos antes, os métodos principais de *aprendizado para redes neurais* são *baseados no cálculo das derivadas parciais* da *função de erro* com relação aos seus *pesos* (sinápticos e de bias).
- Esses métodos têm como *objetivo* encontrar o *conjunto de pesos* que *minimiza a função de erro* escolhida.
- Assim, é necessário encontrar uma maneira de se calcular o *vetor gradiente* da *função de erro com respeito aos pesos das várias camadas de uma rede neural*.
- Essa tarefa pode parecer *trivial*, mas não é o caso.
 - Como podemos calcular a influência dos pesos das camadas ocultas no erro da camada de saída?
- Foram necessários 17 anos desde a criação do *Perceptron* até que se “*descobrisse*” uma forma de treinar redes neurais.

Retropropagação do erro

- Para que entendamos melhor o motivo desta tarefa não ser trivial, nós iremos considerar as notações abaixo, as quais serão úteis a seguir.
 - O peso sináptico, $w_{i,j}^m$, corresponde ao j -ésimo peso do i -ésimo **nó** da m -ésima camada da **rede neural** e W^m é a matriz (ou vetor) com todos os pesos da m -ésima camada.
 - O peso de *bias*, b_i^m , corresponde ao peso do i -ésimo **nó** da m -ésima camada da **rede neural** e b^m é o vetor com todos os pesos de *bias* da m -ésima camada.
 - A **ativação**, u_i^m , corresponde à **combinação linear** das entradas do i -ésimo **nó** da m -ésima camada da **rede neural** e u^m é o **vetor de ativações** com as **combinações lineares** das entradas de todos os nós da m -ésima camada.
 - $f^m(.)$ é a função de ativação da m -ésima camada da **rede neural**.
- Essas notações nos ajudarão a obter os vetores gradiente para atualizar os pesos de todos os nós da rede neural.

Retropropagação do erro

- Usando as notação definidas, podemos representar uma MLP como



OBS.: Para facilitar nossa análise, não vamos considerar as entradas como uma camada, apenas as camadas ocultas e de saída.

- O mapeamento realizado pela rede MLP acima é dado pela expressão

$$y^3 = f^3 \left(\underbrace{W^3 \underbrace{f^2 \left(W^2 \underbrace{f^1 (W^1 x + b^1)}_{y^1} + b^2 \right)}_{y^2} + b^3 \right)$$

Retropropagação do erro

- Para facilitar a análise, iremos supor, sem nenhuma perda de generalidade, que a **função de erro** escolhida é a função do **erro quadrático médio** (MSE).
- Assumiremos que a **última camada da rede MLP** (definida como a M -ésima camada) tem uma quantidade genérica de **nós**, N_M . Assim, o MSE é dado por

$$\begin{aligned} J &= \frac{1}{N_{\text{dados}} N_M} \sum_{n=1}^{N_{\text{dados}}} \sum_{j=1}^{N_M} e_j^2(n) \\ &= \frac{1}{N_{\text{dados}} N_M} \sum_{n=1}^{N_{\text{dados}}} \sum_{j=1}^{N_M} \left(d_j(n) - y_j^M(n) \right)^2, \end{aligned}$$

onde N_{dados} é o número de exemplos, $d_j(n)$ e $y_j^M(n)$ são o valor desejado da j -ésima saída (i.e., rótulo) e a saída do j -ésimo nó da M -ésima camada, respectivamente, ambos correspondentes ao n -ésimo exemplo de entrada.

Retropropagação do erro

- Para treinar a rede (i.e., atualizar os pesos), devemos derivar a **função de erro** com relação aos **pesos** (sinápticos e de bias) de todas suas camadas.
- Como as **saídas dos nós da M -ésima camada** e, conseqüentemente, **seus pesos, aparecem de forma direta na equação do MSE**, é simples se obter as derivadas parciais com relação aos pesos desta camada.

$$J = \frac{1}{N_{\text{dados}} N_M} \sum_{n=1}^{N_{\text{dados}}} \sum_{j=1}^{N_M} \left(d_j(n) - \underbrace{f_j^M \left((\mathbf{w}_j^M)^T \mathbf{y}^{M-1} + b_j^M \right)}_{y_j^M(n)} \right)^2,$$

onde \mathbf{w}_j^M é o vetor de pesos e f_j^M a função de ativação do j -ésimo nó da M -ésima camada.

Retropropagação do erro

- Porém, percebiam que os ***pesos dos nós das camadas ocultas não aparecem explicitamente*** na expressão do erro, J .
- Assim, quando precisamos avaliar as ***derivadas parciais com relação aos pesos das camadas ocultas***, a situação fica mais complexa, pois ***não existe uma dependência direta***.
- Portanto surge a pergunta: Como podemos atribuir aos pesos dos nós das camadas ocultas sua influência no cálculo dos valores de saída da RNA e, conseqüentemente, do erro?

Retropropagação do erro

- Resposta: Propaga-se o erro calculado na saída da rede neural para suas camadas anteriores até a primeira camada oculta usando-se um *algoritmo, baseado na regra da cadeia*, conhecido como *backpropagation* ou *retropropagação do erro*.
- A *dependência dos pesos das camadas ocultas aparece de maneira clara* na expressão do erro através de *aplicações sucessivas da regra da cadeia*.
- Portanto, na sequência, veremos de maneira *sistemática* como a *retropropagação do erro* é realizada para treinar uma rede neural.

Retropropagação do erro

- Inicialmente, nós devemos observar um fato fundamental.
- O cálculo da derivada do erro com relação a um peso qualquer é dado por

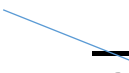
$$\frac{\partial J}{\partial w_{i,j}^m} = \frac{\partial \sum_{n=1}^{N_{\text{dados}}} \sum_{k=1}^{N_M} e_k^2(n)}{\partial w_{i,j}^m} = \sum_{n=1}^{N_{\text{dados}}} \sum_{k=1}^{N_M} \frac{\partial e_k^2(n)}{\partial w_{i,j}^m}.$$

OBS.: mudei o índice do erro de j para k para não haver confusão com o índice j do peso.

- **OBS.1:** A operação da derivada parcial é ***distributiva***.
- **OBS.2:** A divisão pelo número de amostras e saídas é omitida, pois não afeta a otimização por ser um valor constante.
- A equação mostra que é necessário se calcular a derivada parcial apenas do quadrado do erro associado ao n -ésimo exemplo de entrada da k -ésima saída, pois o gradiente será a ***média destes gradientes particulares*** (ou ***locais***).

Algumas noções básicas da retropropagação

- Considerando a **derivada parcial da função de erro em relação a um peso qualquer** e usando a **regra da cadeia**, podemos reescrevê-la como

Peso de qualquer camada. 

$$\frac{\partial J}{\partial w_{i,j}^m} = \frac{\partial J}{\partial u_i^m} \frac{\partial u_i^m}{\partial w_{i,j}^m}.$$

OBS.: Para deixar a notação mais concisa, não vamos incluir o índice das amostras, n .

- A primeira derivada após a igualdade é a derivada da **função de erro** em relação à **ativação** do i -ésimo **nó** da m -ésima camada.
- Esse valor é chamado de **sensibilidade** e será denotado pela letra grega δ . Desta forma a **sensibilidade** do i -ésimo nó da m -ésima camada dado por

$$\delta_i^m = \frac{\partial J}{\partial u_i^m}.$$

- O termo δ_i^m é único para cada **nó** da m -ésima camada.

Algumas noções básicas da retropropagação

- O segundo termo, por sua vez, varia ao longo das entradas do **nó** em questão.
- Lembrando que a ativação, u_i^m , é a **combinação ponderada das entradas do nó mais o peso de bias**

$$u_i^m = \left(\sum_{j \in \text{entradas}} w_{i,j}^m y_j^{m-1} \right) + b_i^m,$$

então, sua derivada em relação ao peso sináptico $w_{i,j}^m$ é dada por

$$\frac{\partial u_i^m}{\partial w_{i,j}^m} = y_j^{m-1}.$$

Saída do j-ésimo nó da camada anterior.

- Caso a derivada seja em relação ao peso de *bias*, b_i^m , temos

$$\frac{\partial u_i^m}{\partial b_i^m} = 1.$$

Algumas noções básicas da retropropagação

- Desta forma, vemos que todas as derivadas da função de erro em relação aos pesos são **produtos de uma sensibilidade**, δ_i^m , **por uma entrada do i -ésimo nó da rede**.

$$\frac{\partial J}{\partial w_{i,j}^m} = \frac{\partial J}{\partial u_i^m} \frac{\partial u_i^m}{\partial w_{i,j}^m} = \delta_i^m y_j^{m-1},$$

ou, no caso do peso de bias, b_i^m , pela unidade

$$\frac{\partial J}{\partial b_i^m} = \frac{\partial J}{\partial u_i^m} \frac{\partial u_i^m}{\partial b_i^m} = \delta_i^m.$$

- São os valores de **sensibilidade**, δ_i^m , que trazem dificuldades em seu cálculo, pois a derivada $\frac{\partial u_i^m}{\partial w_{i,j}^m}$ é trivial (ela é apenas o valor de uma entrada daquele nó).

Retropropagando o erro

- Portanto, a estratégia de otimização adotada para atualização dos pesos (sinápticos e de bias) da rede neural é a seguinte:
 1. Começa-se pela saída, onde o erro é calculado.
 - Etapa chamada de **direta**, pois aplica-se as entradas (i.e., atributos) à rede e calcula-se o erro de saída.
 2. Encontra-se uma **regra recursiva** que gere os valores de **sensibilidade** para os **nós** das camadas anteriores até a primeira camada oculta.
 - Etapa chamada de **reversa**, pois calcula-se a **contribuição de cada nó** das diversas camadas da rede no erro de saída.

Retropropagando o erro

- Esse processo é chamado de ***retropropagação do erro*** ou ***backpropagation***.
- Para facilitar a ***retropropagação do erro***, nós vamos inicialmente agrupar todas as ***sensibilidades*** da m -ésima camada, δ_i^m , $\forall i$, em um vetor, δ^m .
- Em seguida, vamos encontrar uma regra que fará a transição $\delta^m \rightarrow \delta^{m-1}$.
- Ou seja, a partir do vetor de ***sensibilidades*** da camada m , iremos encontrar o vetor de ***sensibilidades*** da camada anterior, $m - 1$.
- Em resumo, o processo de ***retropropagação do erro*** é ***iniciado calculando-se o vetor de sensibilidades da camada de saída, δ^M , e, de maneira recursiva, obtém-se os vetores de sensibilidades de todas as camadas anteriores.***

Retropropagando o erro

- Para calcular a sensibilidade da camada de saída, δ^M , consideramos N_M saídas (i.e., nós) e, assim, temos que o j -ésimo elemento do vetor δ^M é dado por:

$$\delta_j^M = \frac{\partial e_j^2}{\partial u_j^M} = \frac{\partial (d_j - y_j^M)^2}{\partial u_j^M} \stackrel{\text{Regra da cadeia}}{=} \frac{\partial (d_j - y_j^M)^2}{\partial y_j^M} \frac{\partial y_j^M}{\partial u_j^M} = -2(d_j - y_j^M) \frac{\partial y_j^M}{\partial u_j^M} \\ = -2(d_j - y_j^M) f'^M(u_j^M),$$

onde

$$y_j^M = f^M(u_j^M), \\ f'^M(u_j^M) = \frac{\partial f^M(u_j^M)}{\partial u_j^M}.$$

Função logística

$$\frac{\partial f(u)}{\partial u} = f(u)(1 - f(u))$$

Função tangente hiperbólica

$$\frac{\partial f(u)}{\partial u} = (1 - \tanh^2(u))$$

Retropropagando o erro

- Matricialmente nós podemos expressar o vetor δ^M como

$$\delta^M = -2\mathbf{F}'^M(\mathbf{u}^M)(\mathbf{d} - \mathbf{y}),$$

onde a matriz $\mathbf{F}'^M(\mathbf{u}^M)$ é uma **matriz diagonal** com as derivadas das funções de ativação em relação às ativações dos N_M nós da M -ésima camada,

$$\mathbf{F}'^M(\mathbf{u}^M) = \begin{bmatrix} f'^M(u_1^M) & 0 & \cdots & 0 \\ 0 & f'^M(u_2^M) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f'^M(u_{N_M}^M) \end{bmatrix},$$

\mathbf{d} e \mathbf{y} são vetores coluna de dimensão $N_M \times 1$ com os valores esperados e de saída da rede neural, respectivamente.

- Desta forma, a aplicação sucessiva da **regra da cadeia** leva a uma **recursão** que, em termos matriciais, é dada por

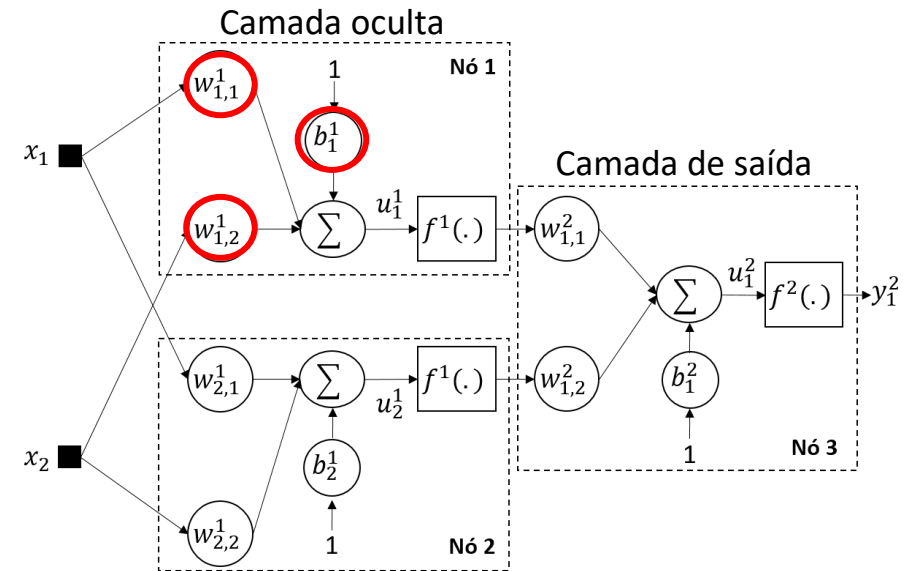
$$\delta^{m-1} = \mathbf{F}'^{m-1}(\mathbf{u}^{m-1})(\mathbf{W}^m)^T \delta^m.$$

Matriz ou vetor com os pesos que conectam a camada $m - 1$ à camada m .

Exemplo da aplicação da retropropagação

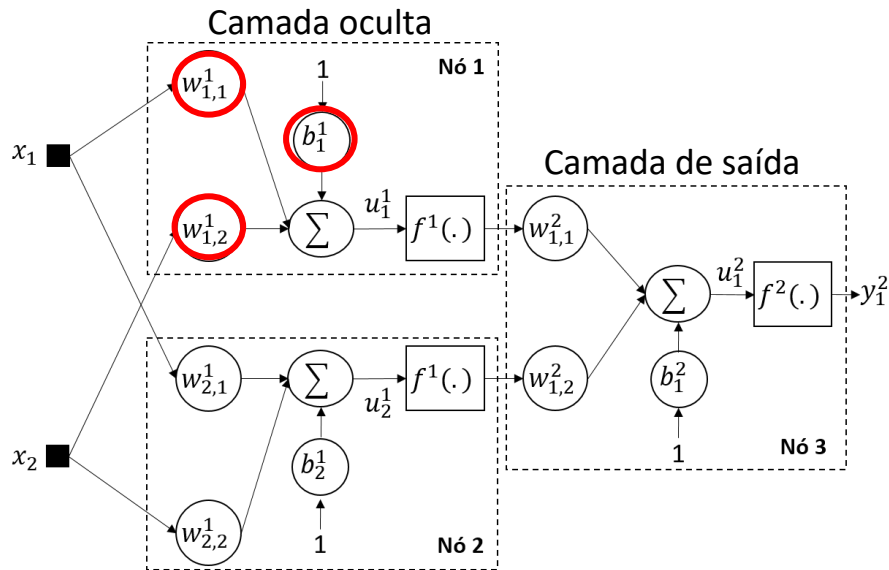
- Encontrar o ***vetor gradiente*** para todos os pesos do nó 1 (camada oculta) da rede neural MLP abaixo.

$$\begin{bmatrix} \frac{\partial J}{\partial w_{1,1}^1} \\ \frac{\partial J}{\partial w_{1,2}^1} \\ \frac{\partial J}{\partial b_1^1} \end{bmatrix} = ?$$



- OBS.:** vamos deixar as derivadas da função de ativação em relação às ativações de forma genérica, ou seja, sem assumir um tipo específico de função de ativação.

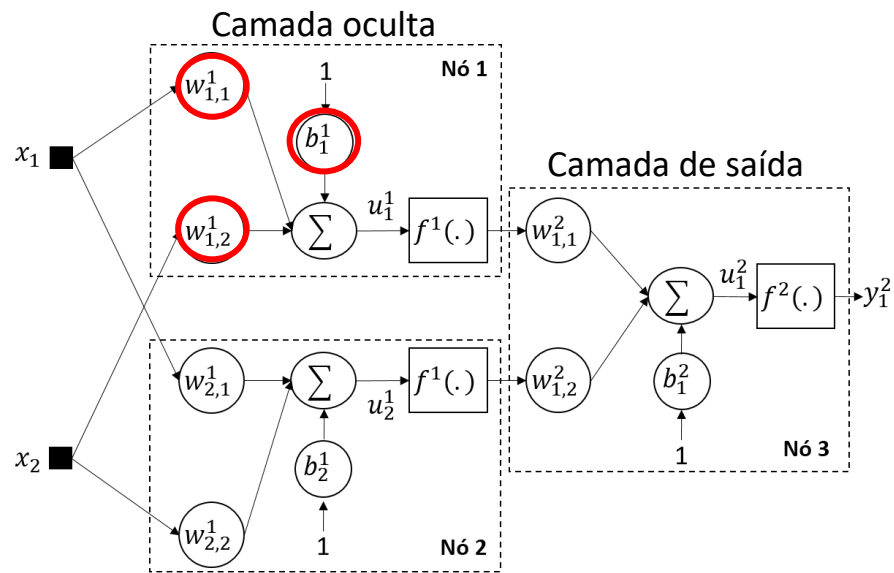
Exemplo da aplicação da retropropagação



- A rede possui uma camada oculta com dois nós e uma **camada de saída com um único nó**, portanto $M = 2$.
- Devemos começar calculando δ^2 .
- Porém, percebam que essa **sensibilidade** é na verdade um escalar, pois há apenas um **nó** na camada de saída.
- Vamos considerar um **único exemplo de entrada**, $x = [x_1, x_2]$ e a respectiva saída desejada, d . Assim

$$\frac{\partial J}{\partial w_{i,j}^m} = \frac{1}{N_{\text{dados}} N_M} \sum_{n=1}^{N_{\text{dados}}} \sum_{k=1}^{N_M} \frac{\partial e_k^2(n)}{\partial w_{i,j}^m} = \frac{\partial e_1^2(n)}{\partial w_{i,j}^m}$$

Exemplo da aplicação da retropropagação



- Vamos supor que os pesos de todos os nós têm uma certa configuração inicial.
 - Por exemplo, os pesos podem ser inicializados com valores retirados de uma distribuição Gaussiana.
- Assim, quando a entrada, \mathbf{x} , é apresentada à rede, é possível calcular todos os valores de interesse ao longo dela até sua saída.
- Consequentemente, tendo o valor de saída, conseguimos calcular o erro.
- Essa é a etapa ***direta*** (ou do inglês, ***forward***).

Exemplo da aplicação da retropropagação

- Portanto, de posse do valor de saída y_1^2 , podemos calcular o erro

$$e_1 = d - y_1^2.$$

- Com o erro, podemos calcular a sensibilidade do **nó** da camada de saída

$$\delta^2 = -2(d - y_1^2)f'^2(u_1^2).$$

- Temos, assim, nossa primeira **sensibilidade**. Agora, usando a equação de recursão para **retropropagar** o erro até a camada anterior, temos

$$\delta^1 = F'^1(\mathbf{u}^1)(\mathbf{W}^2)^T \delta^2,$$

onde $(\mathbf{W}^2)^T = [w_{1,1}^2, w_{1,2}^2]^T$ e

$$F'^1(\mathbf{u}^1) = \begin{bmatrix} f'^1(u_1^1) & 0 \\ 0 & f'^1(u_2^1) \end{bmatrix}.$$

OBS.: Notem que $.^2$ aqui não significa “ao quadrado”, mas sim a indicação de que se trata de um valor da camada $m = 2$.

Exemplo da aplicação da retropropagação

- Portanto, o vetor de sensibilidades da camada 1 é dado por

$$\boldsymbol{\delta}^1 = \begin{bmatrix} \delta_1^1 \\ \delta_2^1 \end{bmatrix} = \begin{bmatrix} w_{1,1}^2 f'^1(u_1^1) \\ w_{1,2}^2 f'^1(u_2^1) \end{bmatrix} \delta^2.$$

- Em seguida, para obtermos a matriz de vetores gradiente, multiplicamos o vetor de ***sensibilidades*** pelo ***vetor de entradas*** da camada.

$$\begin{bmatrix} \frac{\partial J}{\partial w_{1,1}^1} & \frac{\partial J}{\partial w_{2,1}^1} \\ \frac{\partial J}{\partial w_{1,2}^1} & \frac{\partial J}{\partial w_{2,2}^1} \\ \frac{\partial J}{\partial b_1^1} & \frac{\partial J}{\partial b_2^1} \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} [\delta_1^1 \quad \delta_2^1].$$

Exemplo da aplicação da retropropagação

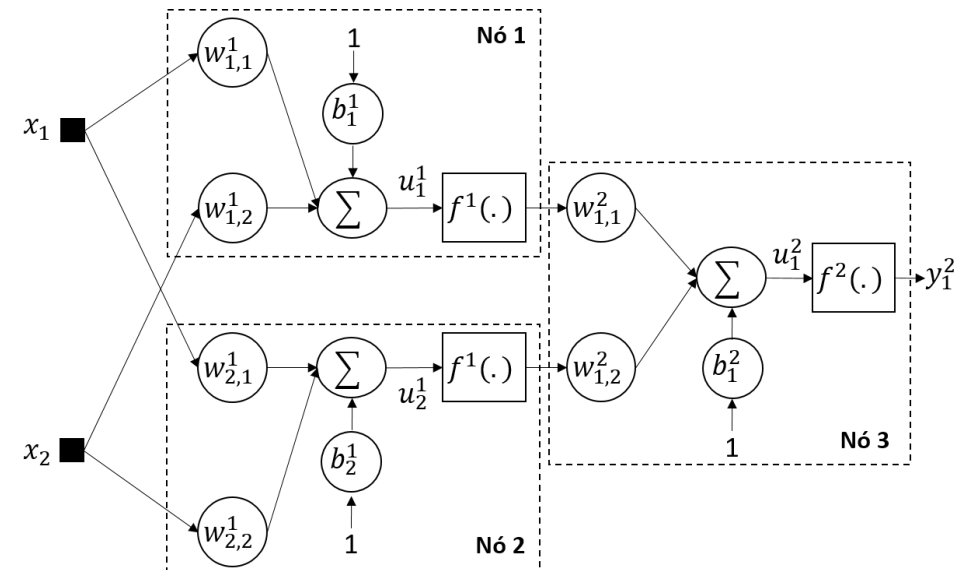
- Assim, as derivadas parciais com relação aos pesos do **nó** $i = 1$ da camada $m = 1$ são dados por

$$\begin{bmatrix} \frac{\partial J}{\partial w_{1,1}^1} \\ \frac{\partial J}{\partial w_{1,2}^1} \\ \frac{\partial J}{\partial b_1^1} \end{bmatrix} = \overset{\text{Escalar}}{\delta_1^1} \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} = \delta^2 w_{1,1}^2 f'^1(u_1^1) \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}$$

Os pesos de **bias** estão ligados às entradas com valores constantes iguais a 1.

$$= -2(d - y_1^2) f'^2(u_1^2) w_{1,1}^2 f'^1(u_1^1) \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}.$$

Exemplo da aplicação da retropropagação



- Se nós fôssemos calcular as derivadas parciais ***aplicando a regra da cadeia diretamente***, elas seriam calculadas como mostrado abaixo.
- Por exemplo, a derivada parcial do erro em relação ao peso $w_{1,1}^1$ é dada por

$$\frac{\partial J}{\partial w_{1,1}^1} = \underbrace{\frac{\partial \left(d - f^2(u_1^2) \right)^2}{\partial f^2(u_1^2)} \frac{\partial f^2(u_1^2)}{\partial u_1^2}}_{\delta_1^1} \underbrace{\frac{\partial u_1^2}{\partial f^1(u_1^1)} \frac{\partial f^1(u_1^1)}{\partial u_1^1}}_{x_1} \underbrace{\frac{\partial u_1^1}{\partial w_{1,1}^1}}_{x_1}$$

- Resolvendo as derivadas parciais, temos

$$\frac{\partial J}{\partial w_{1,1}^1} = -2(d - y_1^2) f'^2(u_1^2) w_{2,1}^2 f'^1(u_1^1) x_1$$

Exemplo da aplicação da retropropagação

- Aplicando-se o mesmo procedimento aos outros pesos, obtemos

$$\frac{\partial J}{\partial w_{1,1}^1} = \frac{\partial e^2}{\partial w_{1,1}^1} = \frac{\partial \left(d - f^2(u_1^2) \right)^2}{\partial f^2(u_1^2)} \frac{\partial f^2(u_1^2)}{\partial u_1^2} \frac{\partial u_1^2}{\partial f^1(u_1^1)} \frac{\partial f^1(u_1^1)}{\partial u_1^1} \frac{\partial u_1^1}{\partial w_{1,1}^1}$$

$$\frac{\partial J}{\partial w_{1,2}^1} = \frac{\partial e^2}{\partial w_{1,2}^1} = \frac{\partial \left(d - f^2(u_1^2) \right)^2}{\partial f^2(u_1^2)} \frac{\partial f^2(u_1^2)}{\partial u_1^2} \frac{\partial u_1^2}{\partial f^1(u_1^1)} \frac{\partial f^1(u_1^1)}{\partial u_1^1} \frac{\partial u_1^1}{\partial w_{1,2}^1}$$

$$\frac{\partial J}{\partial b_1^1} = \frac{\partial e^2}{\partial b_1^1} = \frac{\partial \left(d - f^2(u_1^2) \right)^2}{\partial f^2(u_1^2)} \frac{\partial f^2(u_1^2)}{\partial u_1^2} \frac{\partial u_1^2}{\partial f^1(u_1^1)} \frac{\partial f^1(u_1^1)}{\partial u_1^1} \frac{\partial u_1^1}{\partial b_1^1}$$

Para casa

- Façam o exercício #7 da lista 12.
- Vocês já podem fazer até o exercício #9 da lista 12.

Algumas questões práticas sobre algoritmos de aprendizado

- Podemos dizer que os ***elementos básicos do aprendizado de máquina*** através de ***redes neurais*** foram apresentados até aqui.
- Porém, existem alguns aspectos práticos que nós precisamos discutir.
- Portanto, começamos lembrando sobre a questão do ***cálculo do vetor gradiente***.

Cálculo do vetor gradiente

- Conforme vimos anteriormente, a base para o aprendizado de redes MLP é a obtenção do ***vetor gradiente*** e o estabelecimento de um ***processo iterativo de busca*** dos ***pesos*** que minimizem a ***função de erro***.
- Vimos que a obtenção do ***vetor gradiente*** se dá através do processo de ***retropropagação do erro***, o qual é dividido em duas etapas:
 - Etapa direta (***forward***) onde se apresenta um exemplo de entrada, x , e obtém-se a resposta da rede e, conseqüentemente, o ***erro de saída***.
 - Etapa reversa (***retropropagação***) em que se calculam as derivadas parciais necessárias ao longo das camadas da rede.

Cálculo do vetor gradiente

- Vimos que a derivada parcial do erro em relação a um peso qualquer é a média de ***gradientes particulares (ou locais)***

$$\frac{\partial J(\mathbf{W})}{\partial w_{i,j}^m} = \frac{1}{N_{\text{dados}} N_M} \sum_{n=1}^{N_{\text{dados}}} \sum_{j=1}^{N_M} \underbrace{\frac{\partial e_j^2(n)}{\partial w_{i,j}^m}}_{\text{Gradiente local}} = \frac{1}{N_{\text{dados}}} \sum_{n=1}^{N_{\text{dados}}} \nabla J_n(\mathbf{W}).$$

- O ***gradiente local*** é a derivada parcial do erro da j -ésima saída da rede para o n -ésimo exemplo de entrada em relação ao peso $w_{i,j}^m$.
- $\nabla J_n(\mathbf{W})$ é a média dos N_M ***gradientes locais*** para o n -ésimo exemplo de entrada.
- No entanto, aqui surge um questionamento importante:
 - O que é melhor, usar a ***média dos N_M gradientes locais, $\nabla J_n(\mathbf{W})$, e já dar um passo de otimização***, ou seja, atualizar os pesos, ***reunir o gradiente completo e então dar um passo único e mais preciso*** ou ***um meio termo***?

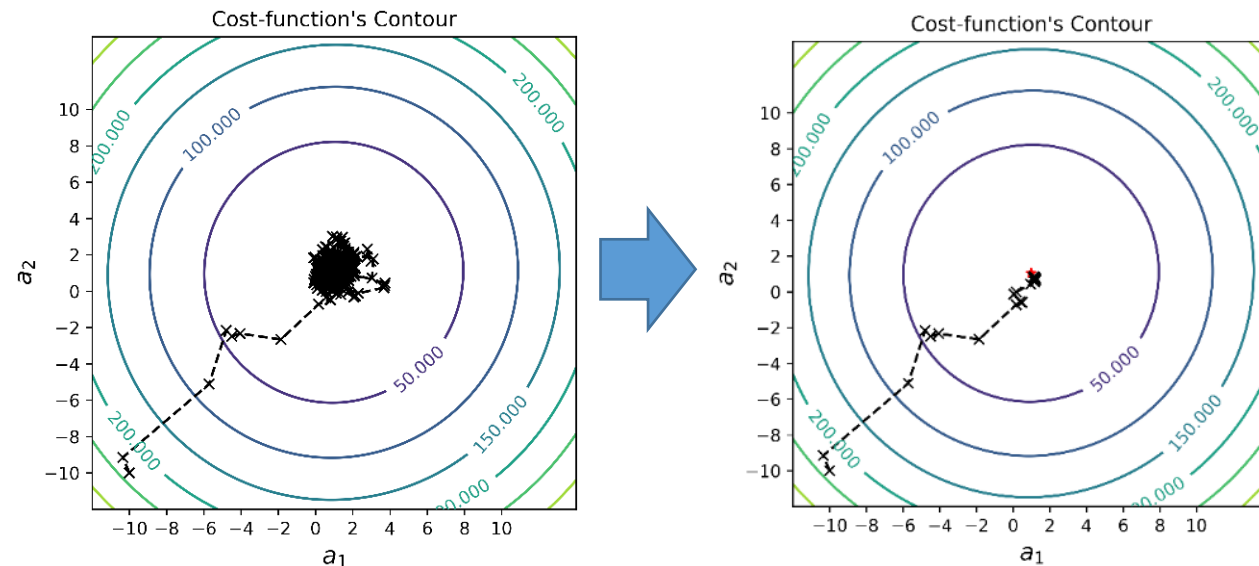
Cálculo do vetor gradiente

- Esse questionamento gera três abordagens possíveis para o cálculo do vetor gradiente.
 - O cálculo usando todos os exemplos (batelada).
 - O cálculo (i.e., estimativa) usando um único exemplo (estocástica).
 - O cálculo usando um subconjunto de exemplos (mini-*batches*).
- Nas ***redes neurais profundas*** (ou ***deep learning***), usadas com muita frequência em problemas possuem enormes conjuntos de dados, usa-se a abordagem com ***mini-batches***, pois com ela, podemos controlar a complexidade computacional necessária para o treinamento.
- **OBS.:** Os exemplos para estimativa do vetor gradiente com as versões ***estocástica*** e ***mini-batch*** devem ser ***aleatoriamente*** escolhidos a partir do conjunto de treinamento.

Variações dos algoritmos de otimização dos pesos

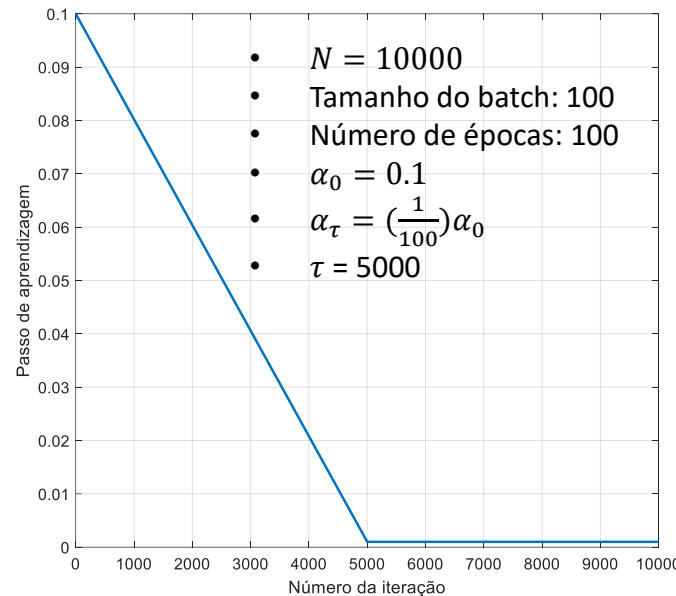
- Existem algumas **modificações** que podem ser aplicadas às versões estocásticas (mini-*batch* e estocástica) para **melhorar seu desempenho sem aumentar muito sua complexidade computacional**.
- As modificações mais usadas são:
 - Redução gradual do passo de aprendizagem,
 - Adição do termo momentum,
 - Adição do termo momentum de Nesterov,
 - Adição de passos de aprendizagem adaptativos.

Redução gradual do passo de aprendizagem



- Assim como fizemos com as versões estocásticas do gradiente descendente quando trabalhamos com regressores lineares, podemos **reduzir o passo de aprendizagem para tornar essas versões mais comportadas e, esperançosamente, obter a convergência.**
- Podemos utilizar todas as técnicas que aprendemos antes: **redução por degraus, decaimento exponencial ou temporal.**

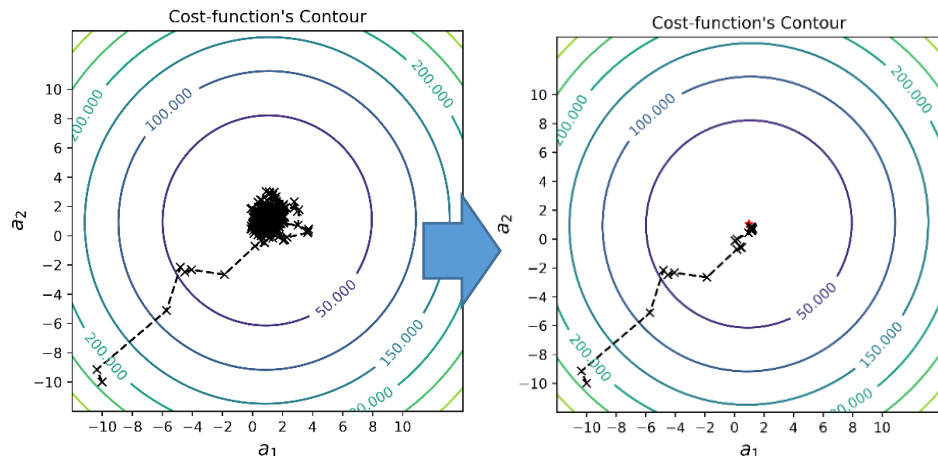
Redução gradual do passo de aprendizagem



- As figuras mostram o resultado do uso da técnica de redução temporal com a equação

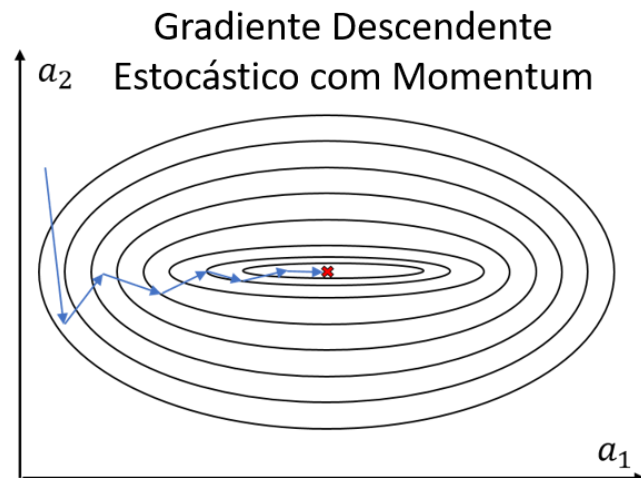
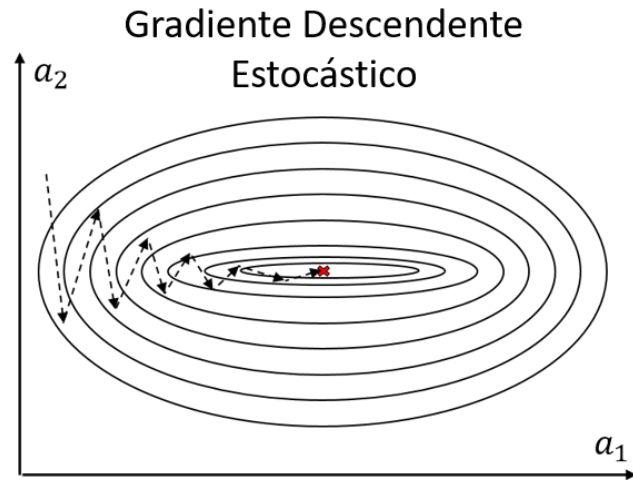
$$\alpha_j = \left(1 - \frac{j}{\tau}\right) \alpha_0 + \frac{j}{\tau} \alpha_\tau,$$

onde j é o contador de iterações, α_0 é o valor inicial do passo, τ é o número da iteração a partir da qual o passo fica constante e α_τ é o valor constante do passo após a τ -ésima iteração.



- Entretanto, percebam que ***ainda temos que encontrar os valores ideais para os hiperparâmetros***, nesse caso, α_0 , α_τ e τ .

Termo momentum

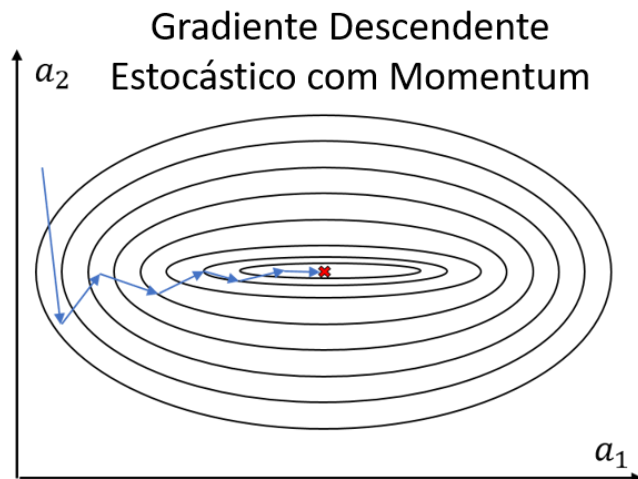
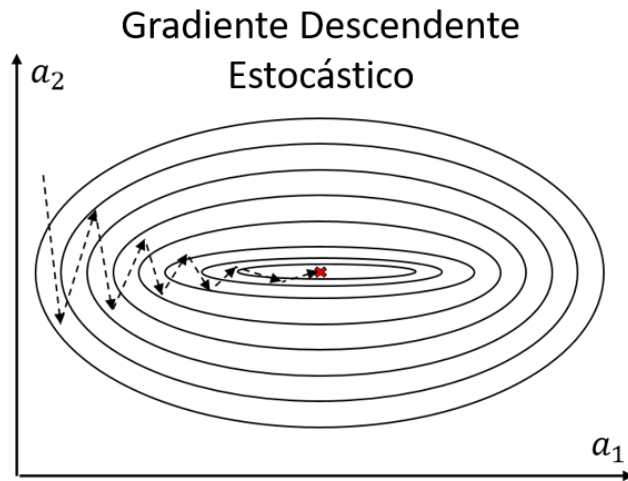


- Como vimos antes, o termo momentum adiciona uma **média movente de estimativas do vetor gradiente**, \mathbf{v} , à equação de atualização dos pesos, **tornando as atualizações menos ruidosas** e, consequentemente, **acelerando a convergência e aumentando a estabilidade** do algoritmo.

$$\mathbf{v}(i) = \mu \mathbf{v}(i-1) + (1-\mu) \nabla \hat{J}_e(\mathbf{w}(i)),$$
$$\mathbf{w}(i+1) = \mathbf{w}(i) - \alpha \mathbf{v}(i).$$

onde $\nabla \hat{J}_e(\mathbf{w}(i))$ é a **estimativa do vetor gradiente** e $\mu \in [0,1)$ (**coeficiente de momentum**) determina a quantidade de estimativas anteriores que são consideradas no cálculo da média.

Termo momentum



- O termo momentum adiciona uma média das estimativas dos gradientes anteriores à atualização corrente.
 - Quando as **estimativas apontam na mesma direção** por várias iterações, o termo faz com que o tamanho dos passos dados naquela direção aumentem, ou seja, o **modelo ganha impulso**.
 - Quando as **estimativas mudam de direção** a cada nova iteração, o termo **suaviza as variações**.
 - Como resultado, temos **convergência mais rápida e oscilação reduzida**.
- A **desvantagem** é que nós precisamos encontrar os valores ideais dos **hiperparâmetros** α e μ .

Momento de Nesterov

- O método do ***momento de Nesterov*** é uma variação do ***termo momentum*** em que o cálculo da ***estimativa do vetor gradiente*** não é feito em relação ao vetor de pesos atual, $\mathbf{w}(i)$, mas em ***relação ao próximo vetor de pesos***, ou seja, em relação ao valor do vetor de pesos após sua atualização com o termo momentum,

$$\nabla \hat{J}_e (\mathbf{w}(i + 1)) = \nabla \hat{J}_e (\mathbf{w}(i) - \alpha \mathbf{v}(i)).$$

- Essa mudança no cálculo da estimativa do vetor gradiente faz com que o ***momento de Nesterov*** apresente ***convergência mais rápida e ajustes mais precisos dos pesos, evitando passos em direções equivocadas.***
- Amortiza as oscilações, especialmente em regiões onde a ***superfície de erro se assemelha à forma de um vale.***

Passo de aprendizagem adaptativo

- Na *variação adaptativa*, o passo de aprendizagem é *ajustado adaptativamente* de acordo com a *inclinação da superfície de erro*.
- Além disso, usa *passos de aprendizagem diferentes para cada peso* do modelo, *os atualizando de forma independente* de acordo com a inclinação da superfície na direção dos pesos.
- Assim, esses métodos são adequados para redes neurais, onde a *superfície de erro é bastante irregular e diferente em diferentes dimensões, tornando a atualização dos pesos mais efetiva*.
- Uma *vantagem* é que na maioria dos casos, *não é necessário se ajustar manualmente nenhum hiperparâmetro*.
- As técnicas mais conhecidas são RMSProp, AdaGrad e Adam.

Inicialização dos pesos

- Um outro aspecto prático que é importante discutirmos é a *inicialização dos pesos de uma rede neural*.
- Como os métodos de treinamento de *redes neurais* são de *busca local*, eles dependem de uma *inicialização dos pesos*.
- Porém, a inicialização pode afetar drasticamente a qualidade da solução obtida.
- O *ponto de inicialização dos pesos* pode afetar a velocidade de convergência do algoritmo.
- Alguns *pontos de inicialização* fazem com que a rede alcance uma *boa solução mais rapidamente*, enquanto outros pontos podem levar a uma *convergência mais lenta* (e.g., algoritmo pode ser inicializado em um ponto de sela ou em uma região de platô).

Inicialização dos pesos

- Alguns ***pontos de inicialização*** são tão instáveis que o algoritmo pode encontrar dificuldades numéricas (***underflow*** e ***overflow***), falhando completamente em convergir (***desaparecimento*** ou ***explosão*** dos gradientes).
- Uma questão importante da inicialização dos pesos é ***quebrar a simetria*** entre os ***nós***, ou seja, ***nós*** com a ***mesma função de ativação*** e ***conectados aos mesmos nós***, devem ter pesos iniciais diferentes, caso contrário, eles terão os mesmos pesos ao longo do treinamento (i.e., aprendem a mesma coisa).
- Portanto, como veremos a seguir, para ***quebrar a simetria e evitar problemas de convergência***, utilizamos algumas ***heurísticas de inicialização aleatória dos pesos***.

Inicialização dos pesos

- Os pesos iniciais são tipicamente obtidos a partir de *distribuições gaussianas ou uniformes*, não importando muito qual delas é usada.
- No entanto, a *escala de variação da distribuição de inicialização dos pesos* tem um efeito significativo tanto no *resultado da otimização* quanto na *capacidade de generalização* da rede neural.
- Sendo assim, a *escala de variação* da inicialização dos pesos levanta algumas discussões.
- Distribuições com *grande escala variação* tendem a *reduzir o problema da simetria*, pois a probabilidade de valores iniciais bastante distintos é maior.

Inicialização dos pesos

- Porém, se as **magnitudes dos valores iniciais forem muito grandes**, podemos ter problemas de **instabilidade**.
- Pesos com magnitudes muito grandes podem levar os **nós** com **funções de ativação** do tipo
 - Sigmoides a operarem na região de saturação, causando o **desaparecimento do gradiente**.
 - ReLU à **explosão do gradiente**.
- Por outro lado, distribuições com **escala de variação muito pequena** têm **maiores chances causar a simetria entre nós** e também podem apresentar **instabilidade ou lentidão** durante o treinamento.
 - Por exemplo, redes com **pesos muito pequenos e com nós usando função de ativação ReLU**, podem ter problemas com o **desaparecimento do gradiente**.
- Na sequência veremos algumas **heurísticas** para inicialização dos pesos.

Heurísticas de inicialização dos pesos

- A ideia por trás destas heurísticas de inicialização dos pesos é **manter a média das ativações dos nós igual a zero e suas variâncias constantes ao longo das várias camadas da rede**, pois desta forma evita-se o desaparecimento ou a explosão do gradiente.
- Considerando uma camada com m entradas e n saídas, temos as seguintes **heurísticas** para inicializar os **pesos sinápticos*** de seus nós.

Inicialização	Funções de ativação	Distribuição Uniforme $U(-r, r)$	Distribuição Normal $N(0, \sigma^2)$
Xavier/Glorot	Linear (i.e., nenhuma), Tanh, Logística, Softmax	$r = \sqrt{\frac{6}{m+n}}$	$\sigma^2 = \frac{2}{m+n}$
He	ReLU e suas variantes	$r = \sqrt{\frac{6}{m}}$	$\sigma^2 = \frac{2}{m}$
LeCun	SELU	$r = \sqrt{\frac{3}{m}}$	$\sigma^2 = \frac{1}{m}$

*Em geral, inicializa-se os **pesos de bias** com **valores iguais a 0**, pois se mostra uma inicialização bastante eficiente na maioria dos casos.

Redes neurais com a biblioteca SciKit-Learn



- A biblioteca SciKit-Learn *disponibiliza apenas dois tipos de arquiteturas* de redes neurais, MLP e *máquina de Boltzmann restrita*.
- A *máquina de Boltzmann* é implementada através da classe BernoulliRBM, e que é usada para *extração de características de forma não supervisionada*.
- Além disso, suas implementações *não são flexíveis* e *não se destinam a aplicações de larga escala*.
 - Por exemplo, a biblioteca *SciKit-Learn* não oferece suporte a GPUs.

Redes neurais com a biblioteca SciKit-Learn

- Para implementações de ***modelos de aprendizado profundo*** escaláveis, muito mais rápidos, flexíveis e baseados em GPU, devemos utilizar bibliotecas como:
 - ***Tensorflow***: criada pela equipe *Google Brain* do *Google*.
 - ***PyTorch***: criada pela *Meta AI* (antigo *Facebook*).
 - ***MXNet***: criada pela *Apache*.
 - ***Theano***: criada pela Universidade de Montreal (primeira versão) e mantida posteriormente pela equipe de desenvolvedores do pacote PyMC sob o nome de Aesara.
 - Entre outras:
https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software

Avisos

- Vocês já podem resolver os exercícios da lista #12.
- Apresentação dos trabalhos finais: XX/YY/2024 a partir das 08:00.
- Horários das apresentações:

Dia	Horário do Início	Número do Grupo	Nome
	8:00		
	8:20		
	8:40		
	9:00		
	9:20		

Obrigado!

People with no idea about AI, telling me my AI will destroy the world

Me wondering why my neural network is classifying a cat as a dog..



Deep Learning



What society thinks I do



What my friends think I do



What other computer scientists think I do



What mathematicians think I do



What I think I do

```
In [1]:
import keras
Using TensorFlow backend.
```

What I actually do

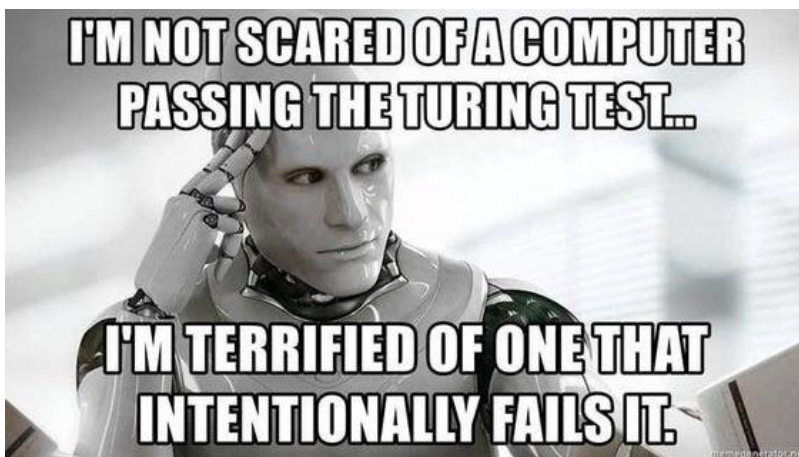
SO YOU ARE TELLING ME



TO TRAIN DEEP LEARNING MODELS IN THE BROWSER?

I'M NOT SCARED OF A COMPUTER PASSING THE TURING TEST...

I'M TERRIFIED OF ONE THAT INTENTIONALLY FAILS IT.



<p>Sigmoid</p> $y = \frac{1}{1 + e^{-x}}$	<p>Tanh</p> $y = \tanh(x)$	<p>Step Function</p> $y = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$	<p>Softplus</p> $y = \ln(1 + e^x)$
<p>ReLU</p> $y = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$	<p>Softsign</p> $y = \frac{x}{1 + x }$	<p>ELU</p> $y = \begin{cases} x, & x < 0 \\ \alpha(e^x - 1), & x \geq 0 \end{cases}$	<p>Log of Sigmoid</p> $y = \ln\left(\frac{1}{1 + e^{-x}}\right)$
<p>Swish</p> $y = \frac{x}{1 + e^{-x}}$	<p>Sinc</p> $y = \frac{\sin(x)}{x}$	<p>Leaky ReLU</p> $y = \max(0, x)$	<p>Mish</p> $y = x \cdot (\tanh(\text{softplus}(x)))$



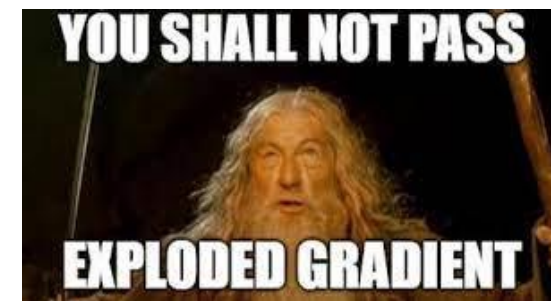
I NEED GPU FOR MY DUMB NEURAL NETWORK

ONE DOES NOT SIMPLY



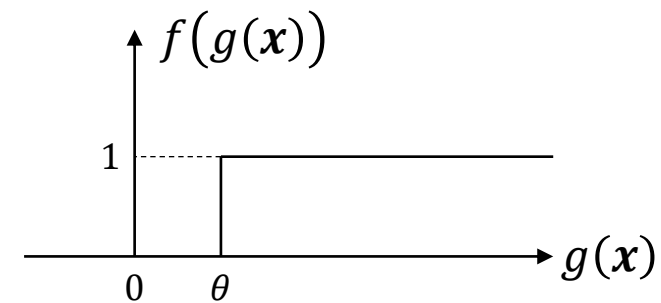
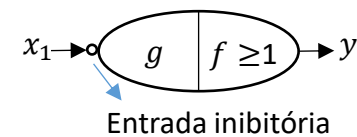
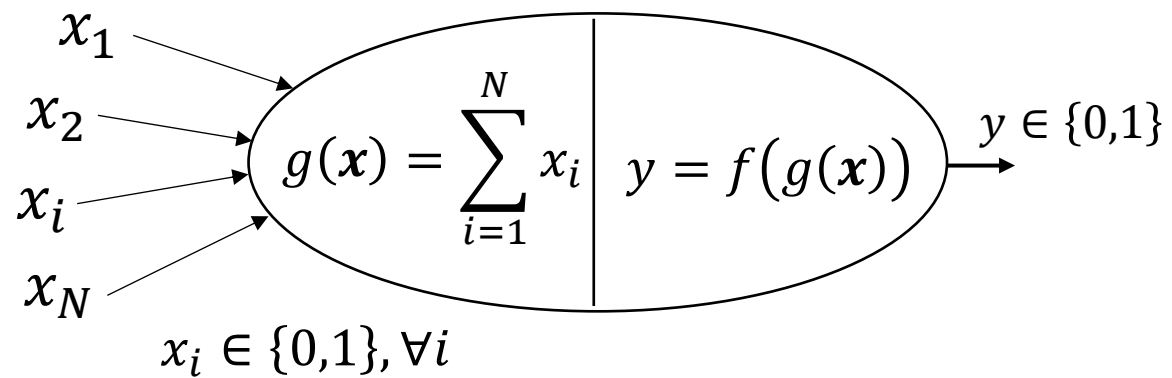
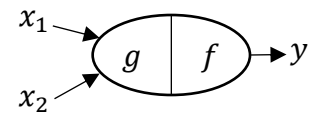
GENERATE MEMES USING DEEP LEARNING

YOU SHALL NOT PASS



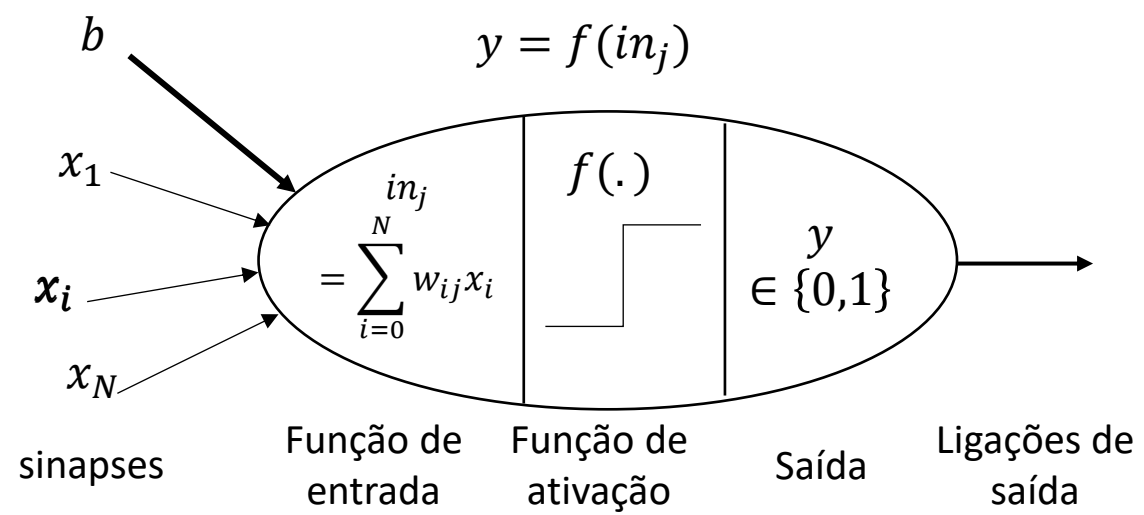
EXPLODED GRADIENT

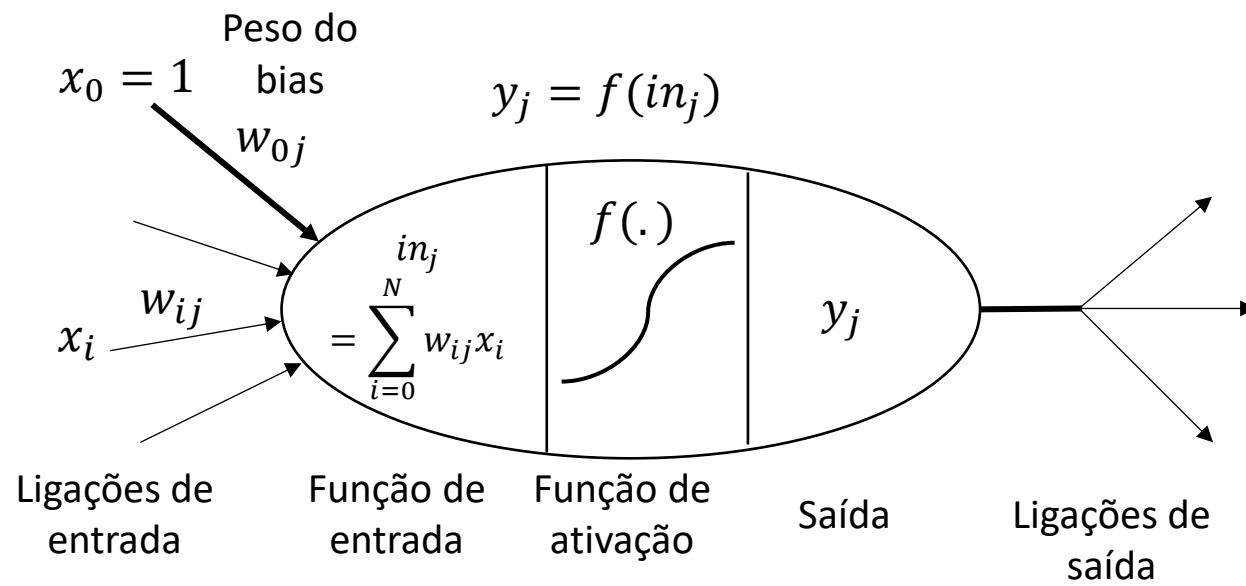
Figuras

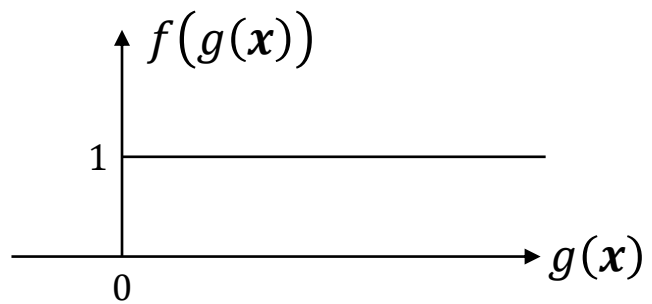
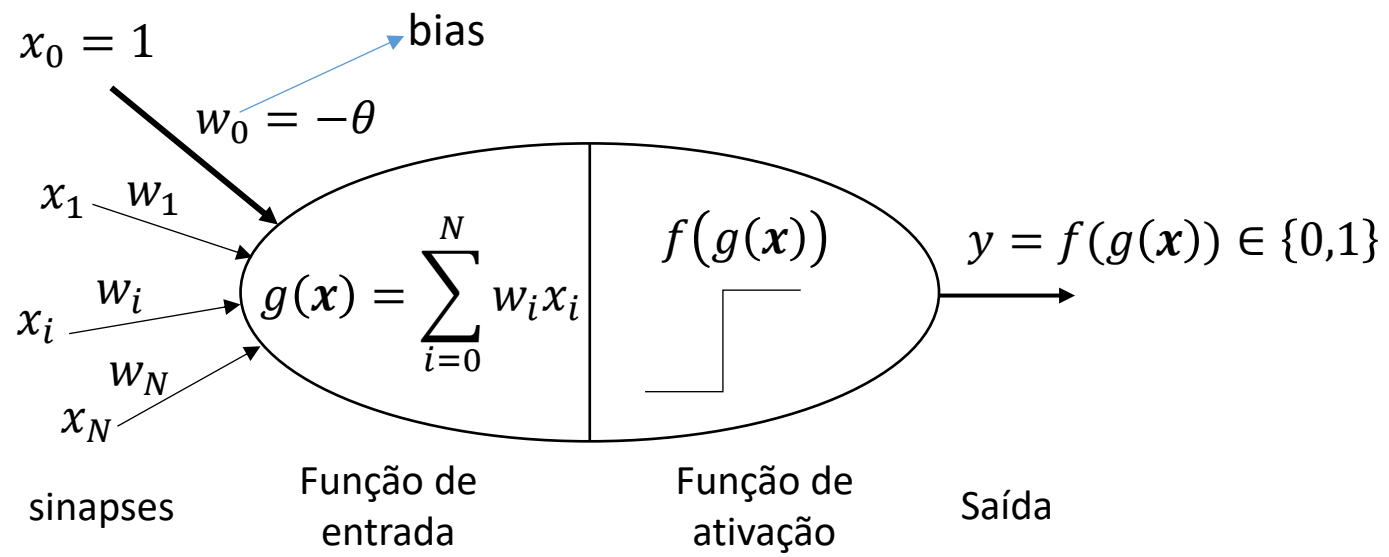


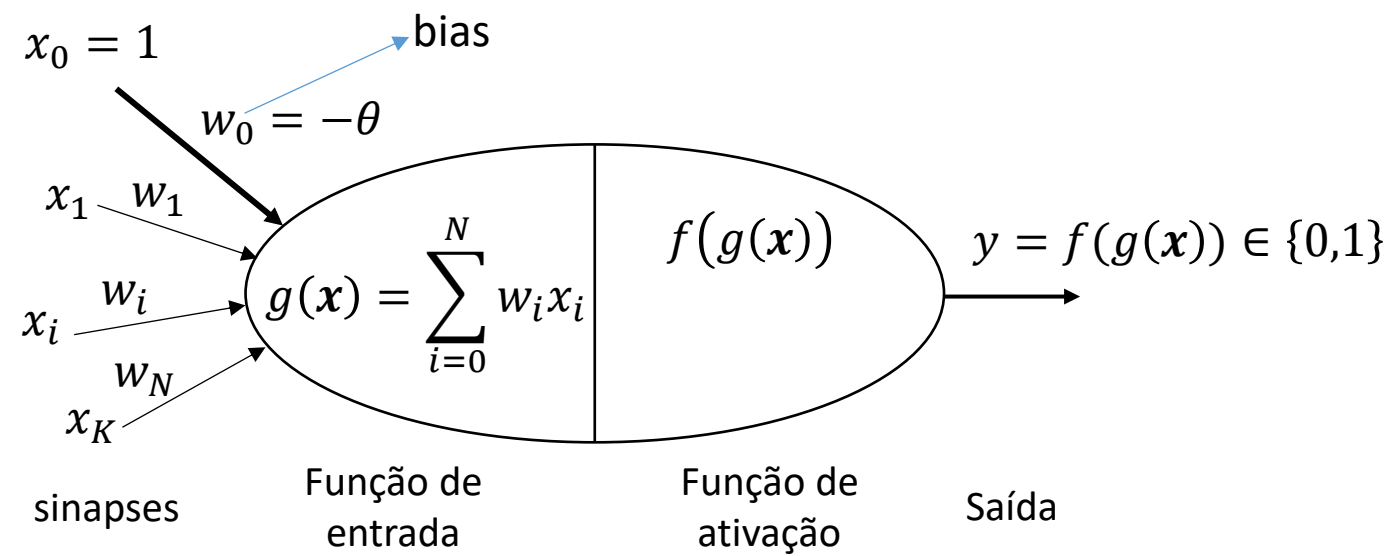
$$y = f(g(\mathbf{x})) = \begin{cases} 1 & \text{se } g(\mathbf{x}) \geq \theta \\ 0 & \text{se } g(\mathbf{x}) < \theta \end{cases}$$

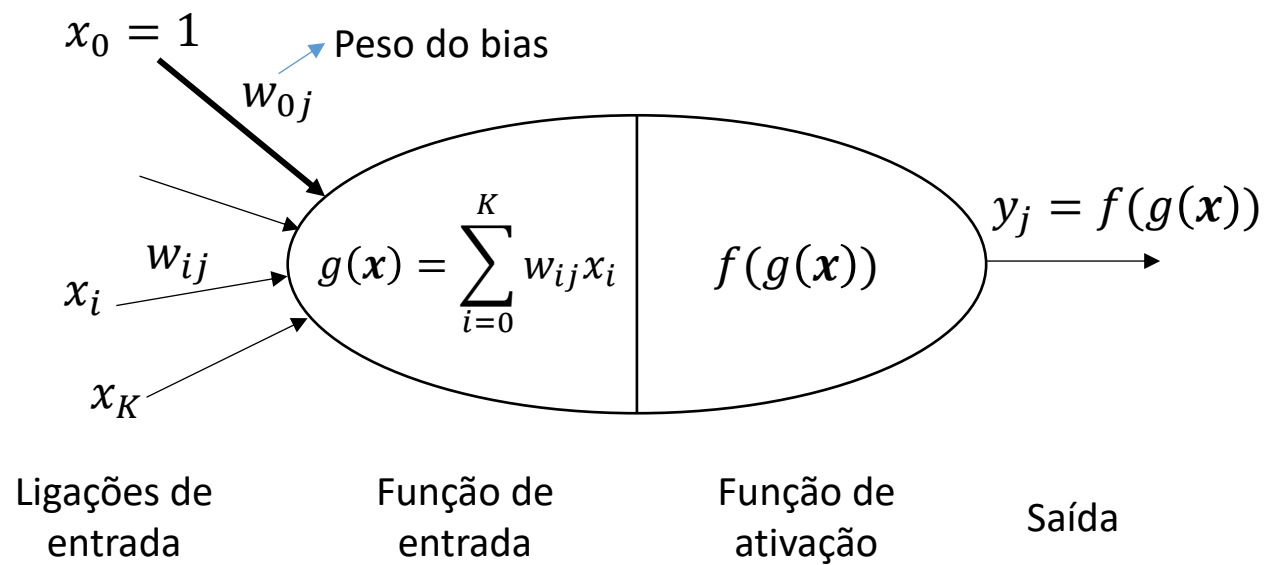
onde θ é o limiar de decisão.

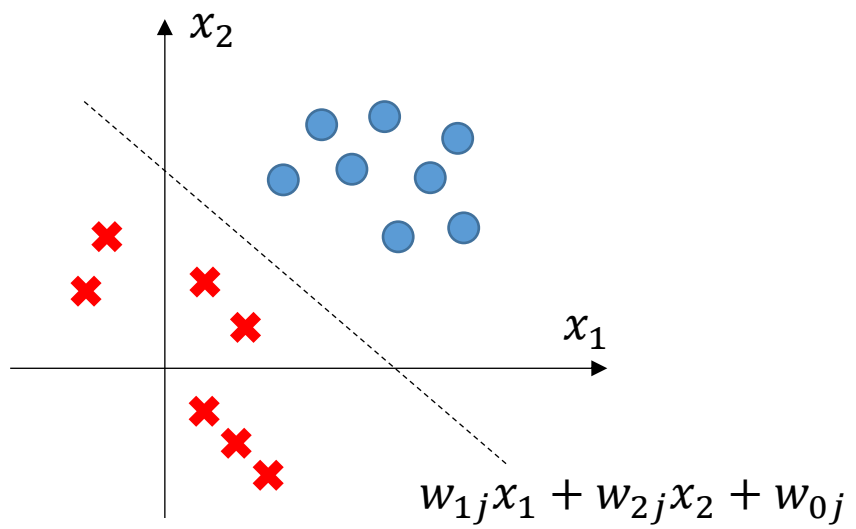


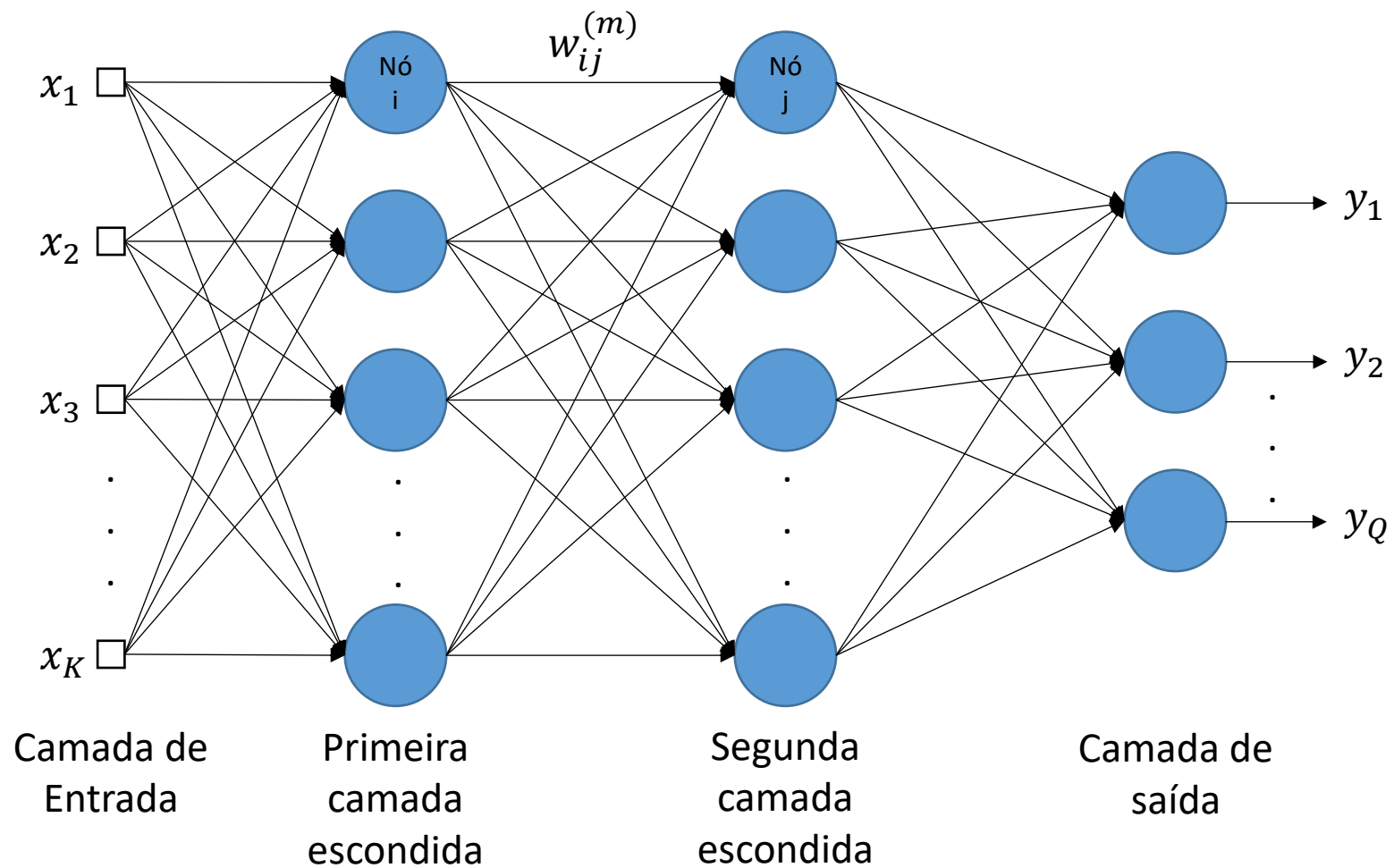










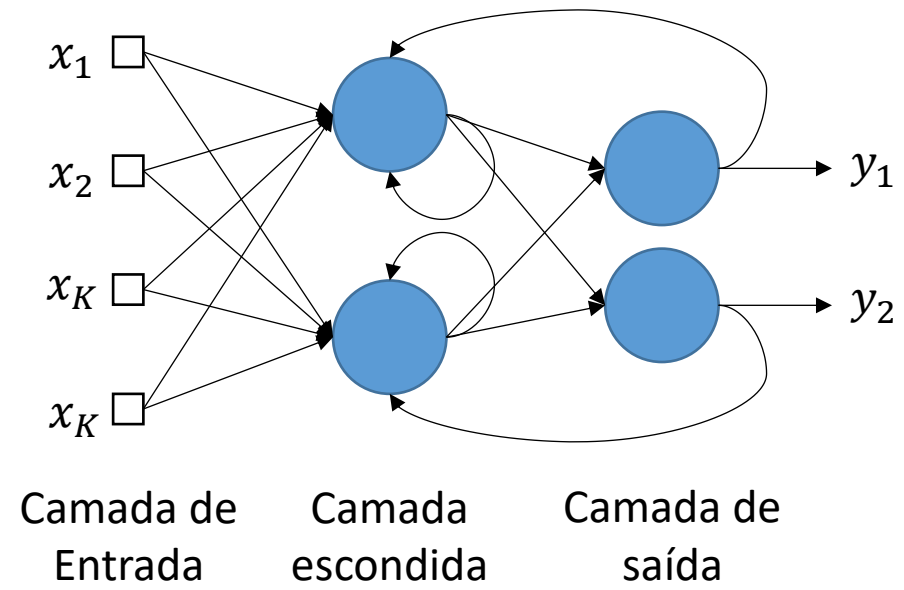


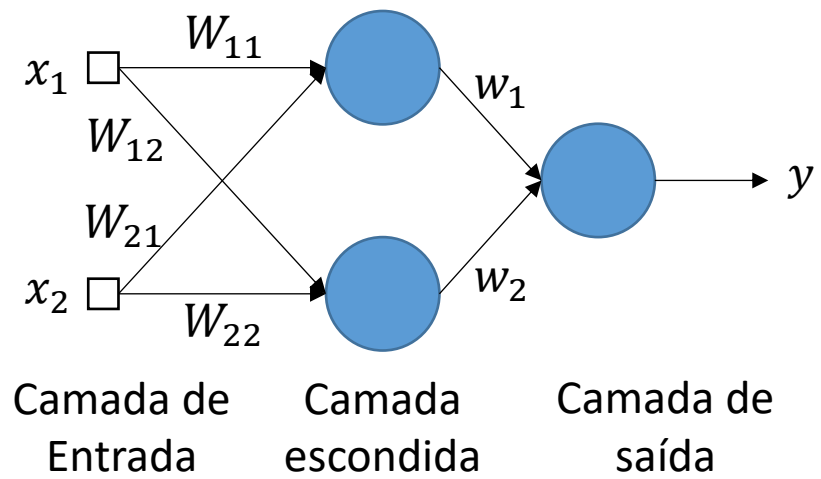


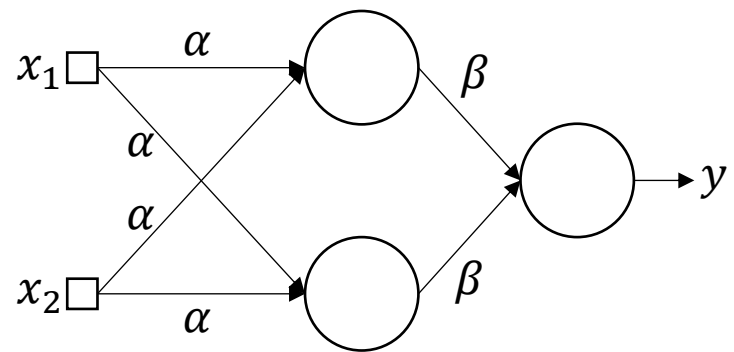
 Nó, unidade ou neurônio.

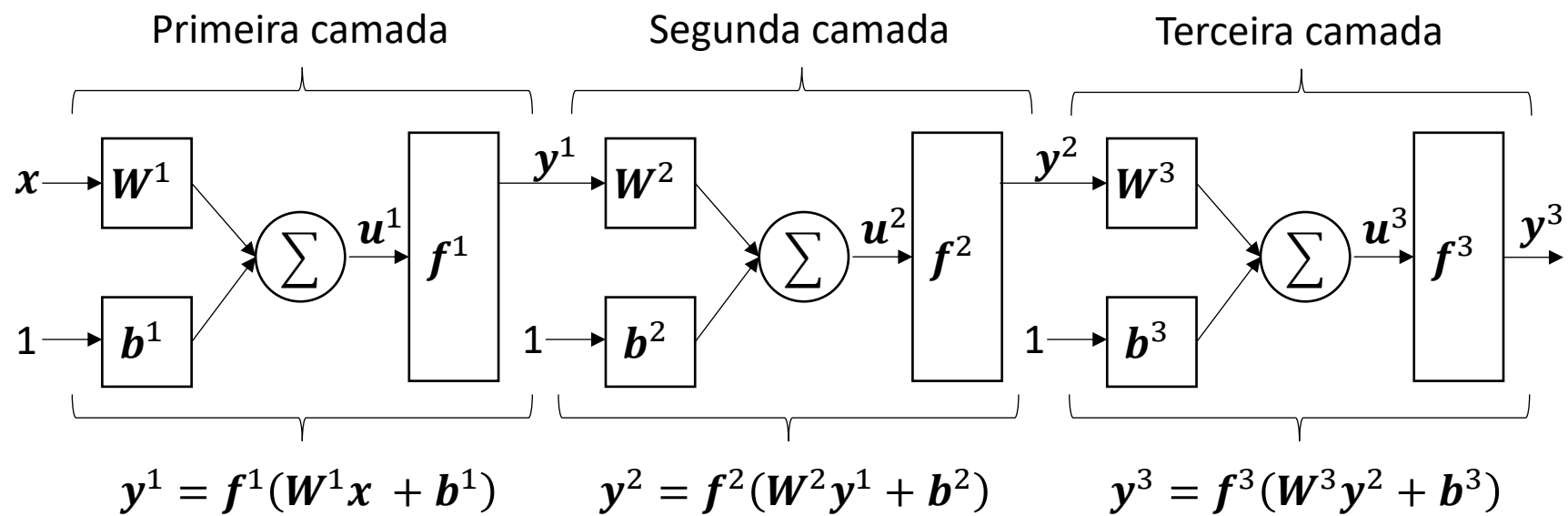
 Ligação entre i -ésimo e j -ésimo nó.

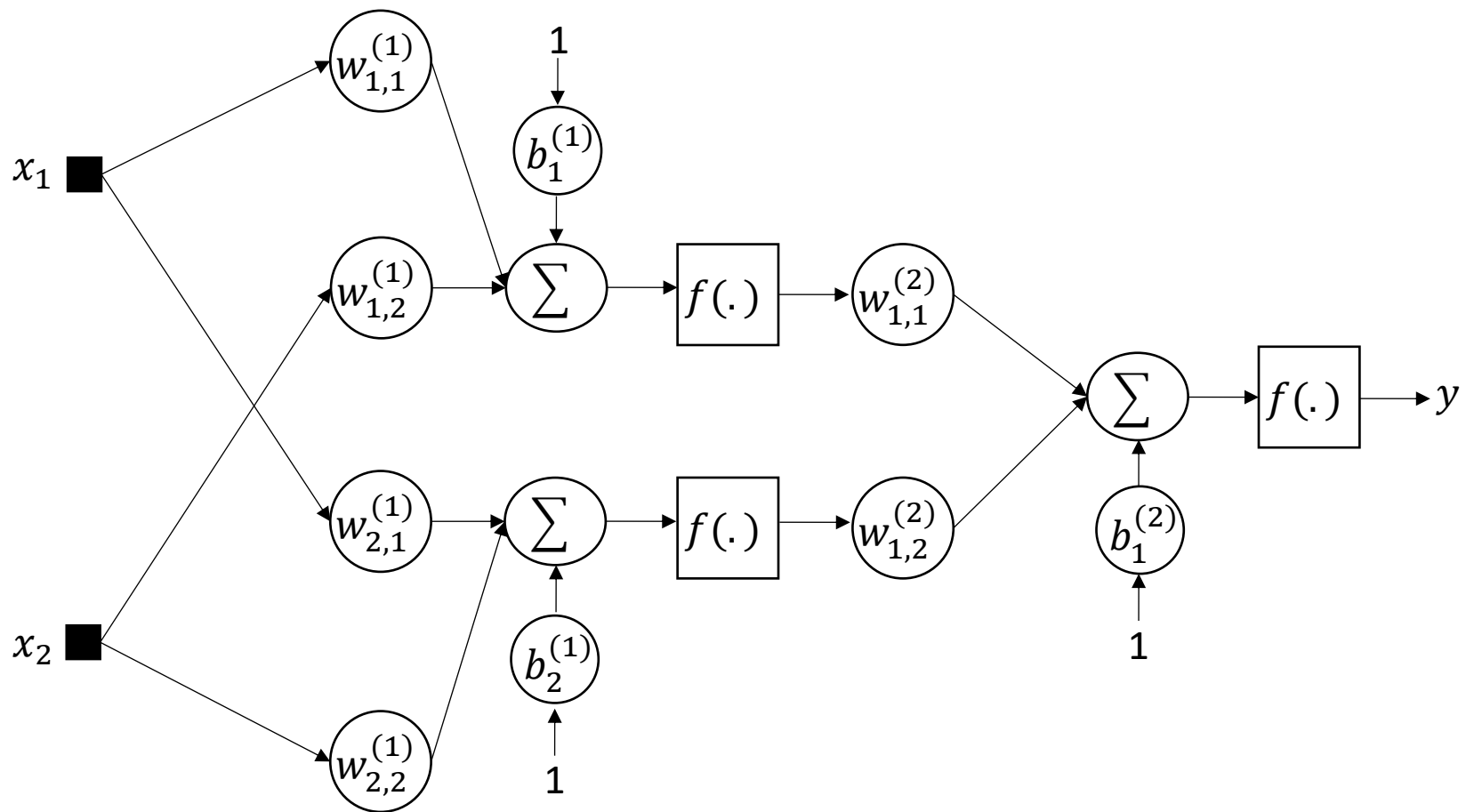
w_{ij} Peso da ligação entre i -ésimo e j -ésimo nó.

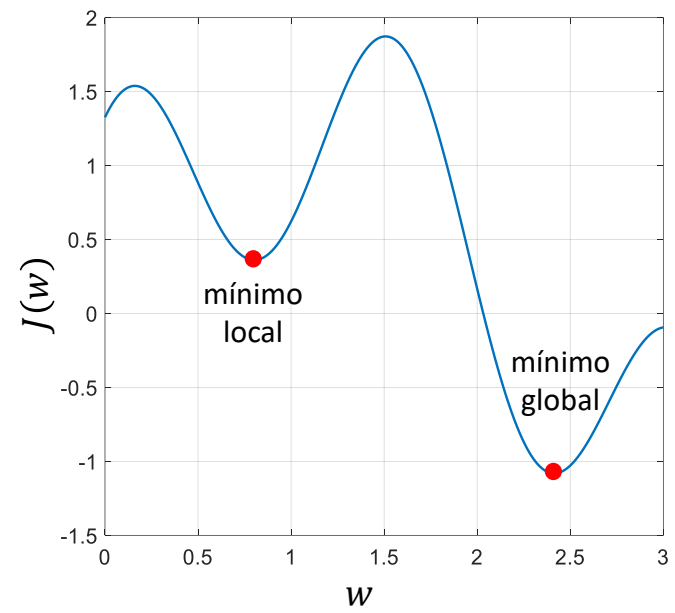






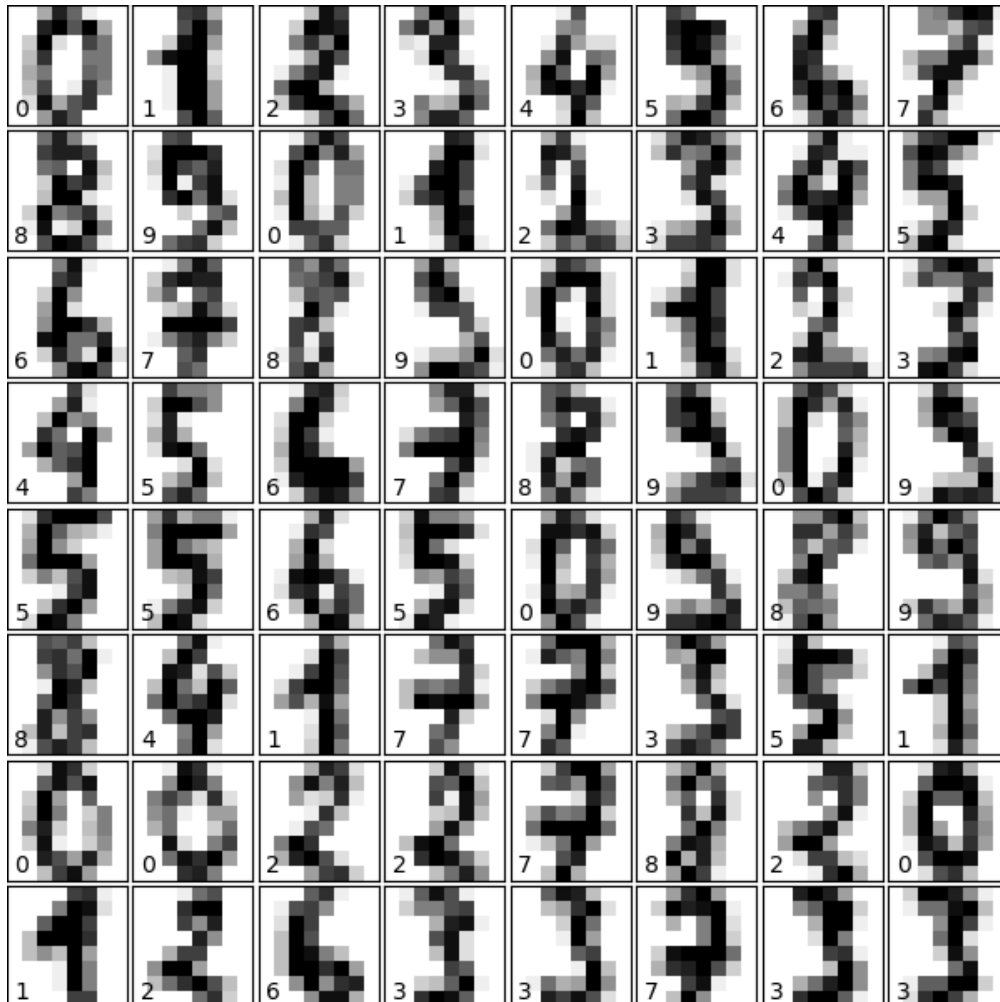




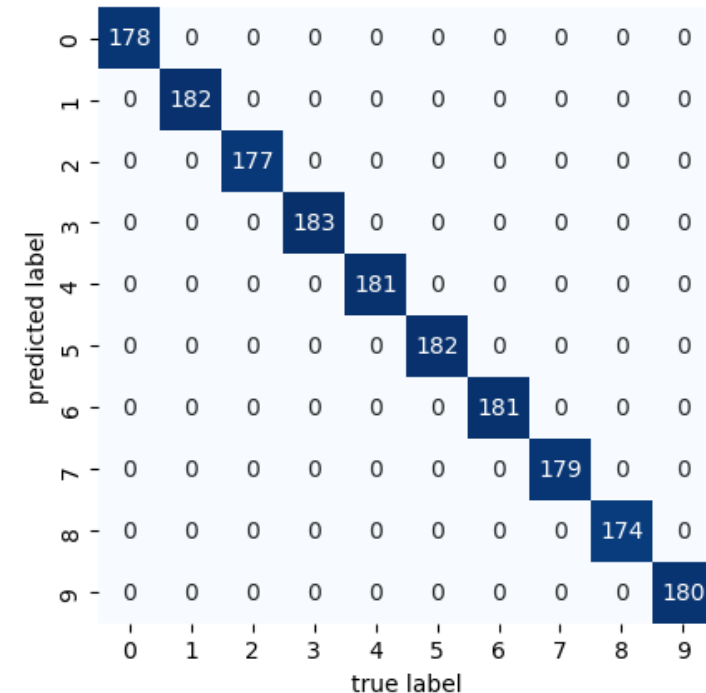


Possíveis respostas

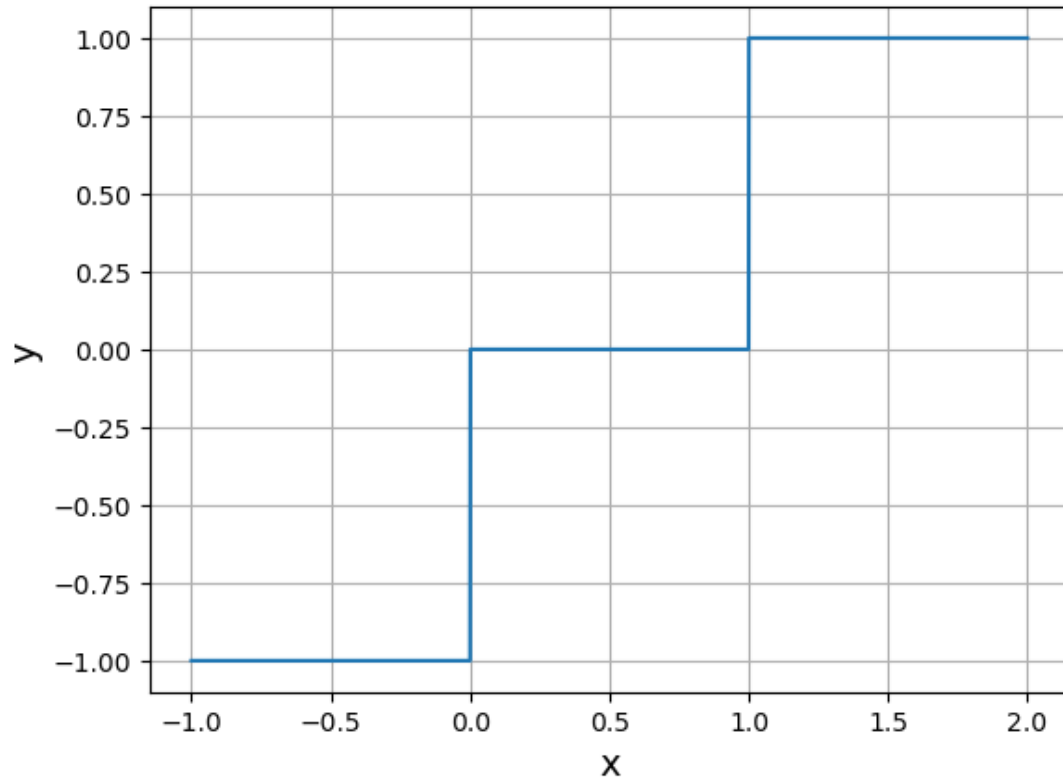
Classificação com MLPClassifier



Classificação de dígitos escritos à mão com uma rede MLP.



Regressão com MLPRegressor



Aproximação de função com
descontinuidades com uma rede MLP.