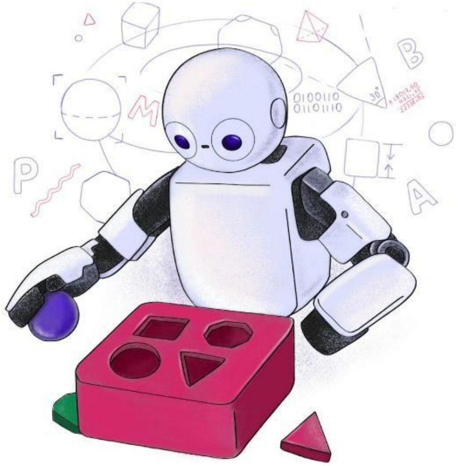
TP555 - Inteligência Artificial e Machine Learning: *Redes Neurais Artificiais (Parte II)*



Esse material foi desenvolvido e gentilmente cedido pelo Prof. Dr. Felipe Augusto Pereira de Figueiredo, do Inatel.(felipe.figueiredo@inatel.br)

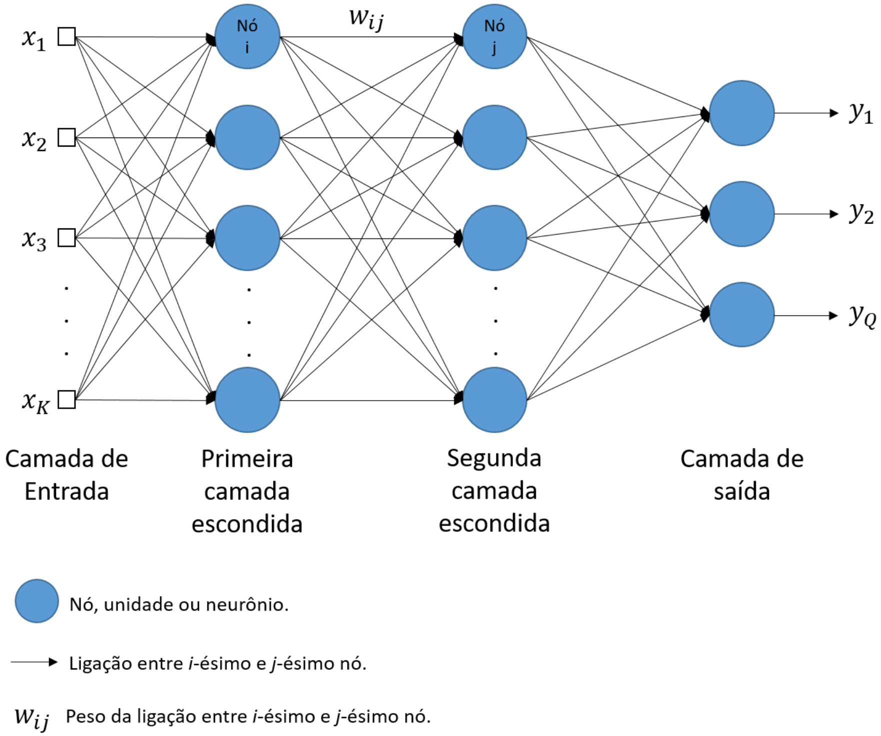
Inatel

Prof. Dr. Luiz Augusto Melo Pereira
luiz.melo@inatel.br

Perceptron de Múltiplas Camadas

- Em termos gerais, uma ***rede neural*** nada mais é do que uma ***combinação de neurônios*** conectados entre si através de ***ligações direcionadas*** (ou seja, as conexões têm uma direção associada).
- As ***propriedades da rede neural*** são determinadas por sua ***topologia*** (i.e., como os neurônios estão conectados, camadas, etc.) e pelas ***propriedades dos neurônios*** (e.g., função de ativação e pesos).
- Algumas das ***limitações dos perceptrons*** (e.g., classificação apenas de classes linearmente separáveis) podem ser ***eliminadas adicionando-se camadas intermediárias*** (também chamadas de ocultas ou escondidas) de ***perceptrons***.
- A RNA resultante é denominada ***Perceptron de Múltiplas Camadas*** (do inglês, *Multilayer Perceptron* - MLP).

Cada ligação tem um peso (sináptico) associado.

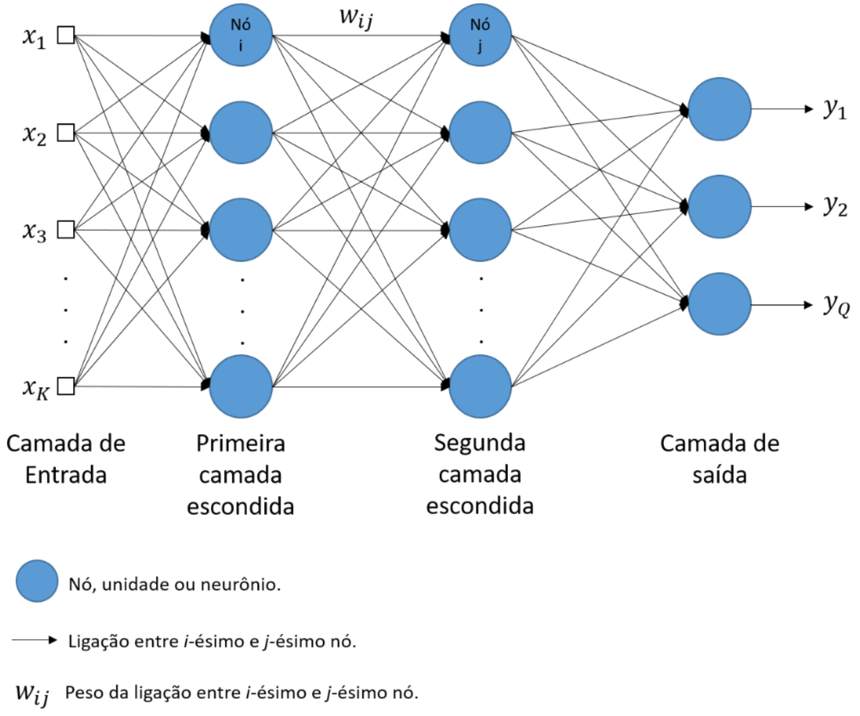


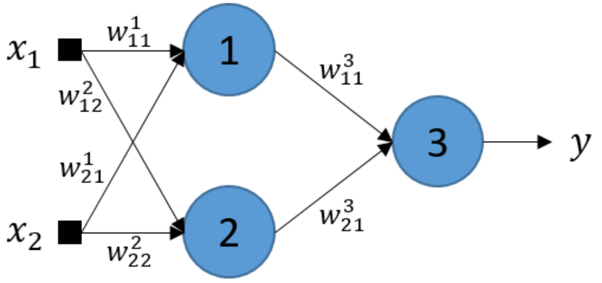
OBS.: Neurônios também são chamados de ***nós*** ou ***unidades***.

Perceptron de Múltiplas Camadas

- Uma rede MLP é sempre ***densamente*** conectada.
  - Cada saída de um nó em uma camada se conecta a todos os nós da camada seguinte através de pesos sinápticos.
- Um exemplo de rede ***MLP com duas camadas intermediárias*** é mostrado na figura ao lado.
- As RNAs são o coração do ***Deep Learning***.
  - Quando uma RNA tem duas ou mais camadas escondidas, ela é chamada de ***rede neural profunda*** (ou em inglês *Deep Neural Network* - DNN).
- **OBS.**: Em particular, uma MLP pode resolver o problema da lógica XOR.
  - Lembrem-se que um único ***perceptron*** não é capaz de realizar essa tarefa.

Cada ligação tem um peso (sináptico) associado.





Perceptron de Múltiplas Camadas

- A ***camada de entrada*** é o ponto de transferência dos ***atributos*** à rede.
- As ***camadas intermediárias*** realizam ***mapeamentos não-lineares*** que, idealmente, vão tornando a informação contida nos dados mais ***"explícita"*** do ponto de vista da tarefa que se deseja realizar.
  - Os mapeamentos são ***não-lineares devido às funções de ativação*** utilizadas não serem lineares, e.g., função logística, tangente hiperbólica, etc.
- Por fim, os ***neurônios*** da ***camada de saída combinam a informação*** que lhes é ***oferecida pela última camada intermediária*** para formar as saídas.
- Redes MLPs são formadas por ***múltiplas camadas de Perceptrons***:
  - Portanto, tais redes têm por base o ***modelo de neurônio do Perceptron***.
- Esse modelo, discutido anteriormente, é mostrado na figura seguinte.

Perceptron de Múltiplas Camadas

- A ***ligação*** do ***nó*** i para o ***nó*** j é feita através do ***peso*** w_{ij} e serve para ***propagar o sinal de ativação*** do ***nó*** i para o ***nó*** j.
- O valor do ***peso*** determina a ***força*** e o ***sinal*** da ***ligação***.
- 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}.
  - Ou seja, esta entrada ***não está conectada a nenhum outro nó***.
- Cada ***nó*** j, calcula a ***soma ponderada*** de suas entrada da seguinte forma

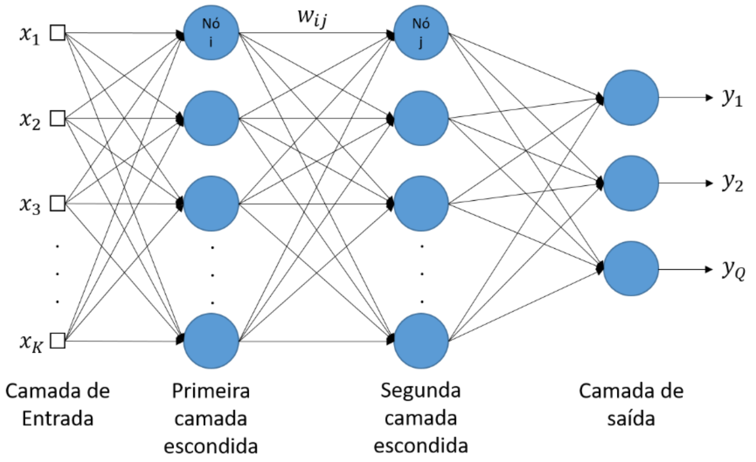
$$g(\boldsymbol{x}) = \sum_{i=0}^{K} w_{ij} x_i .$$

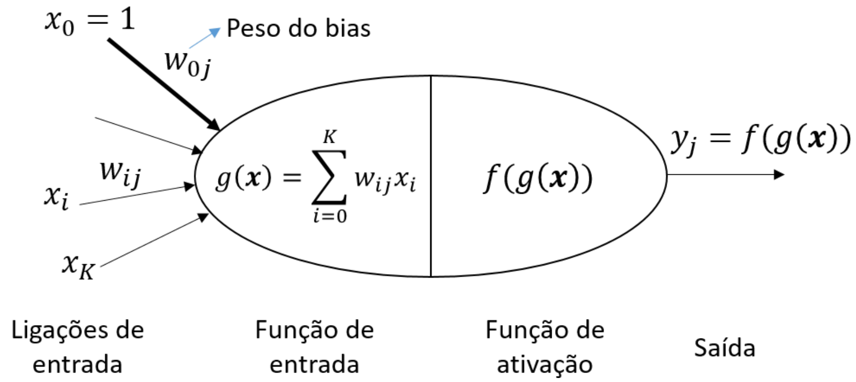
$g(\boldsymbol{x})$ é também chamada de **ativação** do nó.

- Em seguida, o ***nó*** aplica uma ***função de ativação*** (i.e., de limiar), $f(.)$, ao somatório acima para obter sua saída

$$y_j = f\big(g(\boldsymbol{x})\big) = f\left(\sum_{i=0}^{K} w_{ij} x_i\right) = f\left(\boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}\right).$$

- Existem vários tipos de ***funções de ativação*** que podem ser utilizadas pelos ***nós*** de uma rede MLP.
- Cada camada pode usar funções de ativação diferentes, mas, em geral, a mesma camada usa a mesma função.





$$y_j = f\big(g(\boldsymbol{x})\big) = f\left(\sum_{i=0}^{K} w_{ij} x_i\right).$$

onde x_i é a saída do nó i e w_{ij} é o peso conectando a saída do nó i para este nó, o nó j.

Funções de ativação

- Devido a suas características, não se utiliza a ***função degrau*** como função de ativação em MLPs.
  - Derivada sempre igual a zero, exceto na origem, onde 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***.
- A ***função logística*** tem a seguinte expressão:

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

onde z_j é a ***combinação linear das entradas do nó***, i.e., $g(\boldsymbol{x})$.

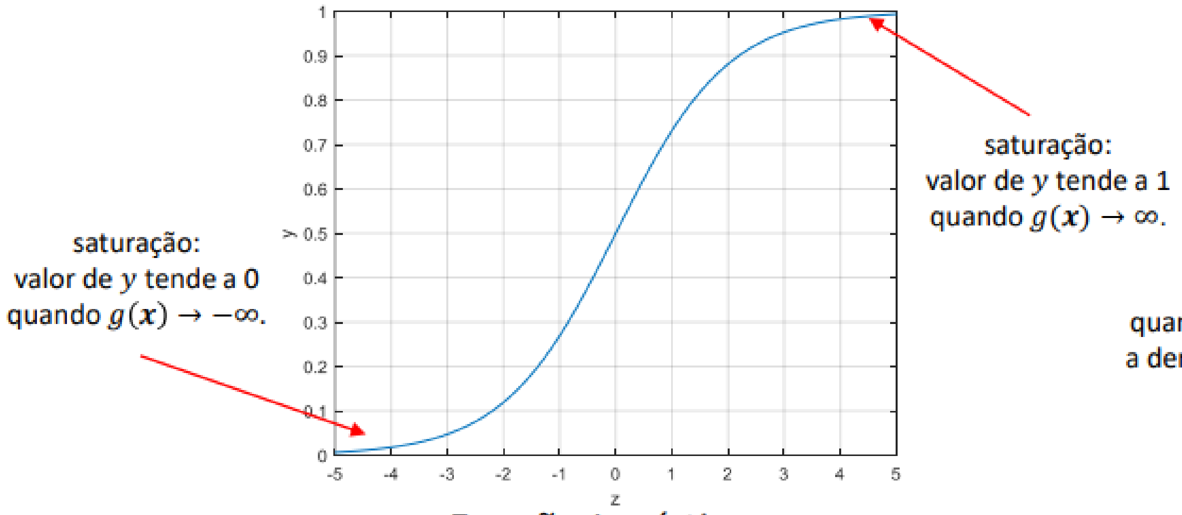
- Sua derivada é dada por

$$\frac{dy_j}{dz_j} = \frac{df(z_j)}{dz_j} = y_j(1 - y_j) \geq 0$$

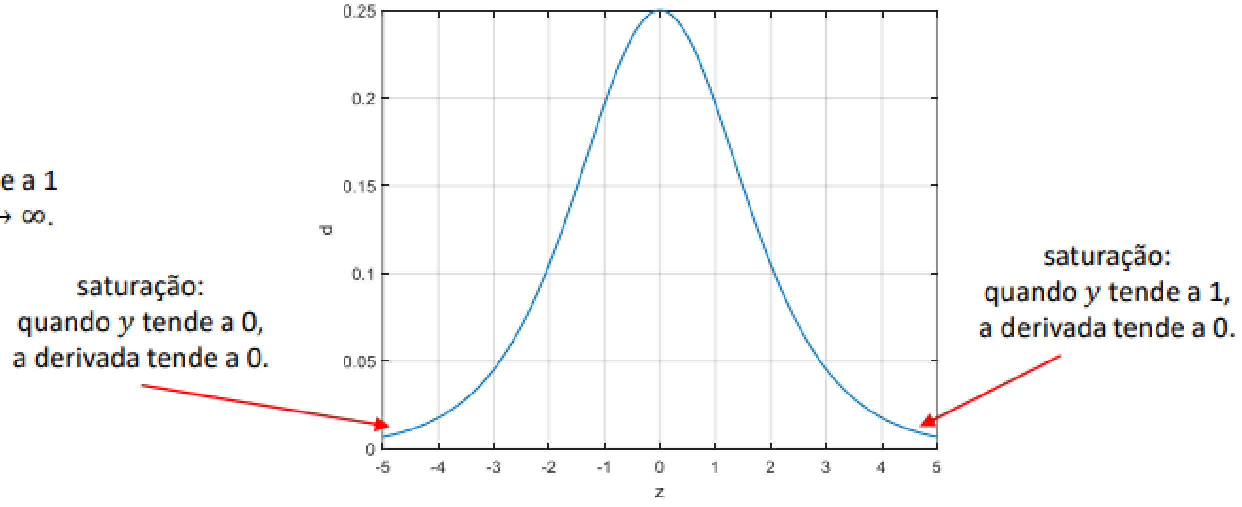
- A derivada será importante durante o processo de aprendizado da rede neural.

Funções de ativação

- A ***função logística*** e sua derivada são mostradas nas figuras abaixo.
- O valor da derivada, d, ***sempre será menor do que 1***, ***sendo no máximo igual a 0.25 quando*** $g(\boldsymbol{x}) = 0$.
  - Isso causa um problema no aprendizado de redes com muitas camadas, i.e., redes profundas, chamado de ***dissipação do gradiente***.
- Quando z se torna muito grande (negativo ou positivo), a função satura em 0 ou 1, e o valor da derivada tende a 0.



Função Logística



Derivada da Função Logística

Funções de ativação

- A ***função tangente hiperbólica*** tem sua expressão dada por:

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

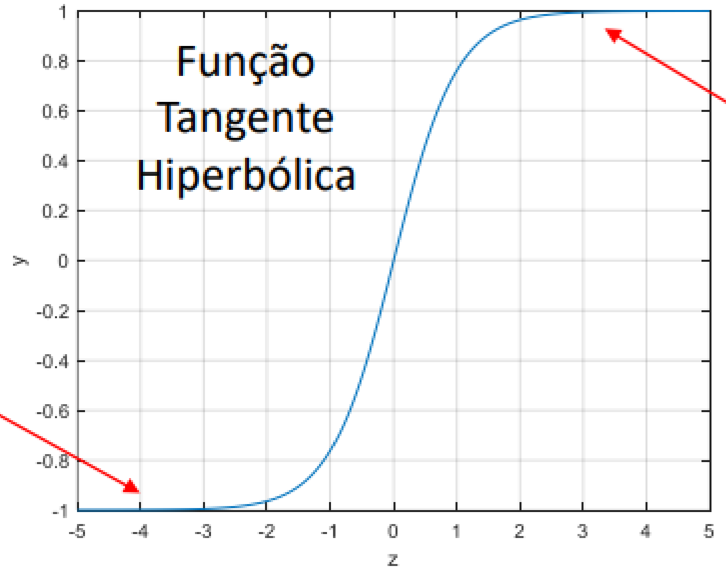
onde z_j é a ***combinação linear das entradas do nó***, i.e., $g(\boldsymbol{x})$.

- Sua derivada é dada por

$$\frac{dy_j}{dz_j} = \frac{df(z_j)}{dz_j} = 1 - \tanh^2(z_j) \geq 0$$

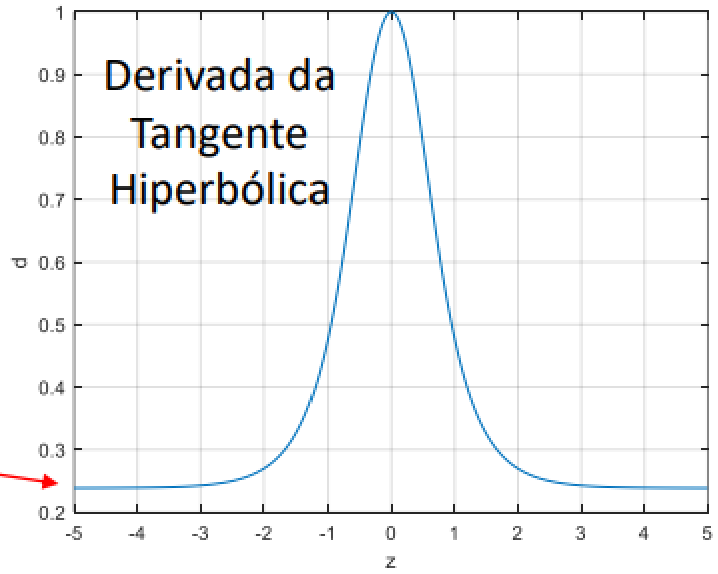
- A função e sua derivada são mostradas nas figuras abaixo.

A derivada é no máximo igual a 1 quando z, $g(\boldsymbol{x})$, é exatamente igual a 0.



saturação: valor de y tende a 1 quando $g(\boldsymbol{x}) \rightarrow \infty$.

saturação: valor de y tende a -1 quando $g(\boldsymbol{x}) \rightarrow -\infty$.

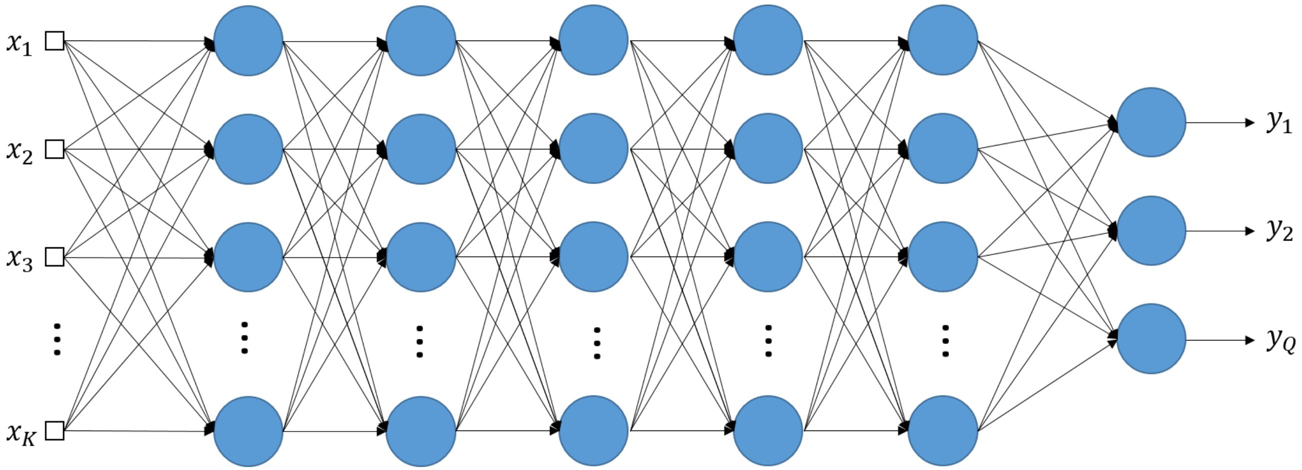


A derivada é igual a 0 quando $g(\boldsymbol{x}) \rightarrow -\infty$.

A derivada é igual a 0 quando $g(\boldsymbol{x}) \rightarrow \infty$.

O Problema da Dissipação do Gradiente

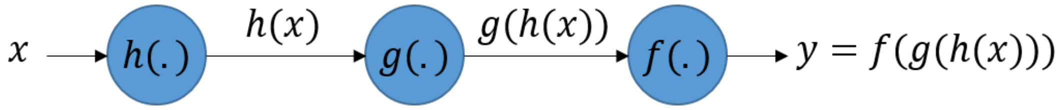
- É um problema encontrado quando treinamos ***redes neurais profundas***, ou seja, com muitas camadas escondidas, com ***métodos de aprendizado baseados no gradiente*** e ***funções de ativação sigmoide ou tangente hiperbólica***.
- Ocorre devido à natureza do ***algoritmo de retropropagação***, que 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 àquele peso 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.



Em suma, o gradiente se torna cada vez menor nas camadas próximas à entrada, levando a uma atualização muito pequena ou até inexistente nos pesos destas camadas.

O Problema da Dissipação do Gradiente

- Lembrem-se que as ***funções de ativação***, como ***tangente hiperbólica*** ou ***logística***, têm derivadas parciais no intervalo de 0 até 1.
- Durante o treinamento, para atualizar os pesos de cada camada da ***rede neural***, o ***algoritmo de retropropagação*** calcula os gradientes dos pesos das camadas ocultas através do uso da ***regra da cadeia*** (exemplo abaixo).



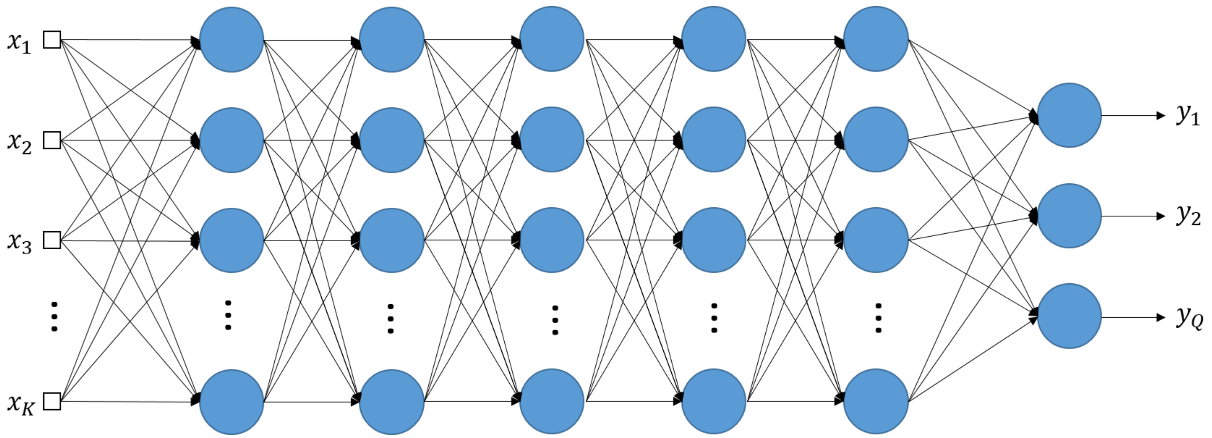
$$\frac{\partial y}{\partial x} = \frac{\partial f\left(g\left(h(x)\right)\right)}{\partial x} = \frac{\partial f\left(g\left(h(x)\right)\right)}{\partial g\left(h(x)\right)} \frac{\partial g\left(h(x)\right)}{\partial h(x)} \frac{\partial h(x)}{\partial x}$$

OBS.: As funções $f(.), g(.)$ e $h(.)$ podem ser interpretadas como sendo as funções de ativação dos nós.

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

O Problema da Dissipação do Gradiente

- Em uma rede com $\boldsymbol{M}$ camadas, a ***retropropagação*** tem o efeito de multiplicar até $\boldsymbol{M}$ valores pequenos (i.e., derivadas parciais) para calcular os gradientes das primeiras camadas.
- O que significa que o ***gradiente diminui exponencialmente*** com $\boldsymbol{M}$.
- Isso significa que os ***nós das camadas iniciais aprendem muito mais lentamente do que os nós das camadas finais***, pois o valor do gradiente é muito pequeno, fazendo com que a ***atualização dos pesos também seja pequena*** (i.e., lenta).



Exemplo: vanishing_gradientsv3.ipynb

O Problema da Dissipação do Gradiente

- Esse problema foi uma das razões pelas quais as ***redes neurais profundas*** foram ***abandonadas por um longo tempo***, voltando à cena em ***2010, quando se fez um progresso significativo em sua compreensão*** [1].
- Os autores de [1] mostraram que com ***funções de ativação sigmoide ou tangente hiperbólica*** e um ***esquema de inicialização usando distribuição normal com média zero e variância unitária***, ***a variância das saídas de cada camada é muito maior do que a variância de suas entradas***.
- Indo em direção à saída da rede, a ***variância continua aumentando após cada camada*** até que as ***funções de ativação*** de camadas posteriores ***saturem***.
- ***Um dos insights*** de [1] foi que os problemas da ***dissipação e explosão dos gradientes*** são em parte causados pela ***escolha inadequada da função de ativação***.
- ***Funções de ativação que não saturem*** e ***inicialização adequada dos pesos*** são formas de mitigar o problema.

[1] Xavier Glorot and Yoshua Bengio, "Understanding the Difficulty of Training Deep Feedforward Neural Networks", January 2010.

Funções de ativação

- Com o surgimento das ***redes neurais profundas***, uma outra função, conhecida como ***função retificadora***, passou a ser a bastante utilizada por questões ***numéricas*** e ***computacionais***.
- A ***função retificadora*** tem sua expressão dada por

$$y_j = f(z_j) = \max(0, z_j).$$

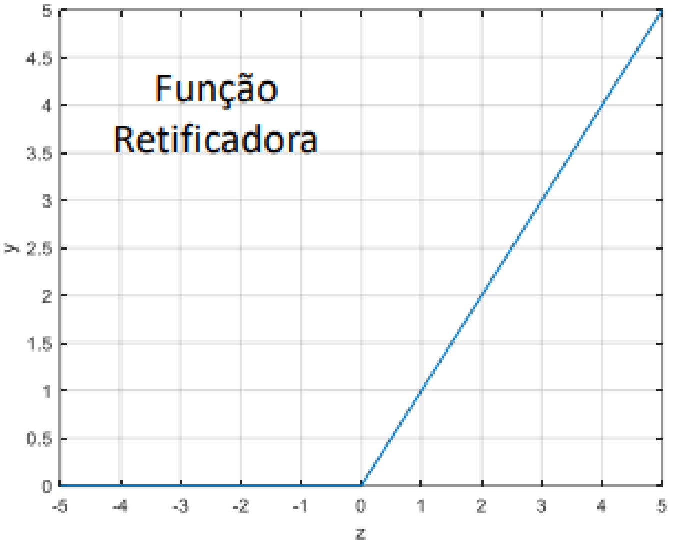
- Sua derivada é dada por

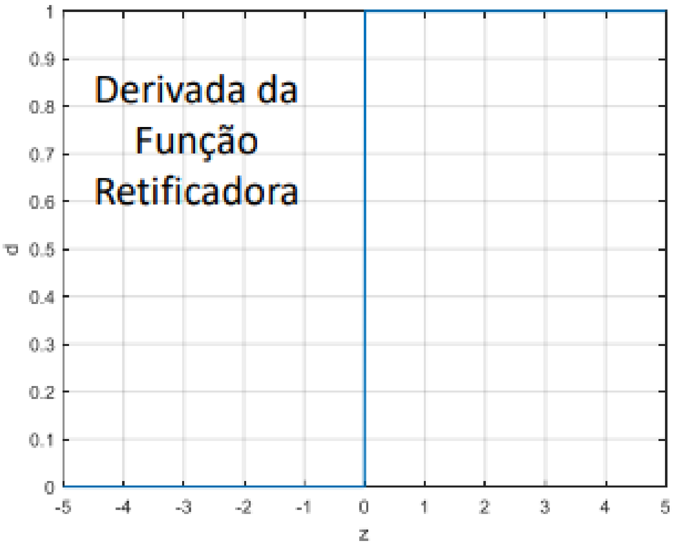
$$\frac{dy_j}{dz_j} = \frac{df(z_j)}{dz_j} = \begin{cases} 0, & \text{se } z_j < 0 \\ 1, & \text{se } z_j > 0 \end{cases}$$

Função degrau

e é indefinida para $z_j = 0$, porém o valor da derivada em zero pode ser arbitrariamente escolhido como 0 ou 1.

- Um ***nó*** que emprega uma ***função de ativação retificadora*** é chamado de ***rectified linear unit*** (ReLU)
- A ***função retificadora*** e sua derivada são mostradas nas figuras ao lado.





Funções de ativação

- Vantagens da ***função retificadora***:
  - A função e sua derivada são ***mais rápidas de se calcular*** do que as funções logística e tangente hiperbólica.
  - Não satura para ativações, z_j, positivas, minimizando o problema da dissipação do gradiente.
    - O gradiente para valores positivos é sempre igual a 1, assim, se vários gradientes de várias camadas forem multiplicados, não haverá diminuição do seu valor.
- Infelizmente, a função ReLU não é perfeita. Ele sofre de um problema conhecido como ***ReLUs agonizantes***:
  - Durante o treinamento, alguns nós com função de ativação ReLU “morrem”, ou seja, seus pesos não são mais atualizados, permanecendo inalterados.
  - Isso ocorre porque a ativação, z_j, tem valor negativo, fazendo com que a derivada parcial seja igual a 0.

Funções de ativação

- Para resolver o problema das ***ReLUs agonizantes***, usa-se variantes da função ReLU que possuem gradiente diferente de zero para $z_j < 0$:
  - **Leaky ReLU**: $f(z_j) = \max(0.01 z_j, z_j)$.
  - **Randomized leaky ReLU:** $f(z_j) = \max(\alpha z_j, z_j)$ onde α é um valor aleatório.
  - **Parametric leaky ReLu:** $f(z_j) = \max(\alpha z_j, z_j)$ onde α deve ser aprendido durante o treinamento.

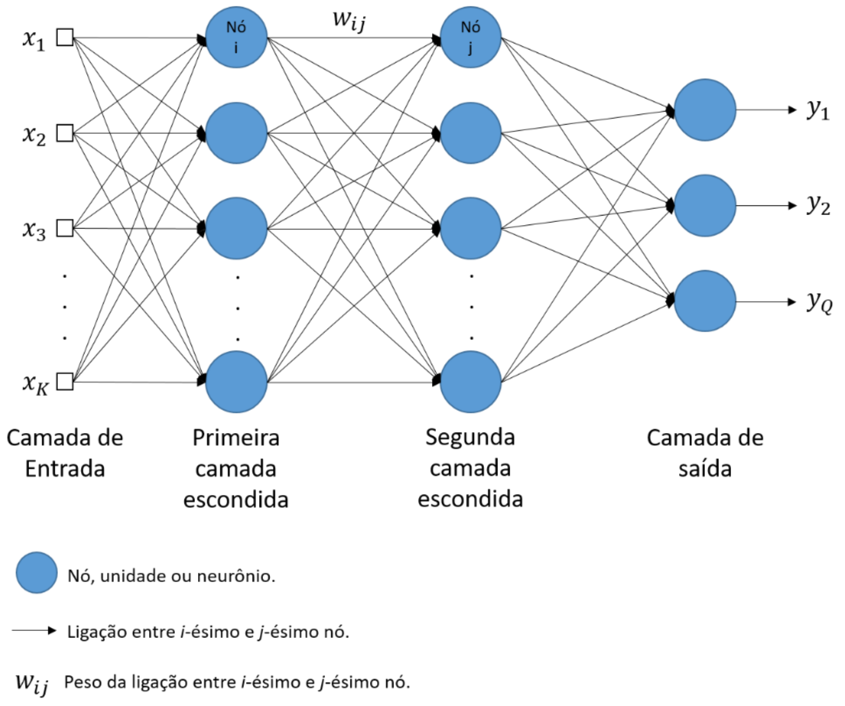
- Outras funções de ativação são:
  - **Exponential linear unit (ELU)**: supera ReLU e suas variantes em vários experimentos.
  - **Scaled ELU (SELU)**: possui a propriedade de auto-normalização, onde a saída de cada camada tende a preservar a média e o desvio padrão dos sinais de entrada, minimizando os problemas da dissipação e da explosão do gradiente.
  - https://en.wikipedia.org/wiki/Activation_function#Table_of_activation_functions

Evitando a dissipação e explosão do gradiente

- Algumas formas de se evitar os problemas da dissipação e explosão do gradiente são:
  - **Inicialização dos pesos**: heurísticas de inicialização criadas para ***garantir que a variância da saída de cada camada seja similar à variância de sua entrada***. As heurísticas também ***devem garantir que os gradientes tenham a mesma variância antes e depois de fluírem através de uma camada na direção reversa*** (mitiga ambos os problemas).
  - **Normalização de mini-batches**: consiste em adicionar uma operação imediatamente antes ou depois da função de ativação de cada camada oculta, que padroniza (remove média e divide pelo desvio padrão) cada entrada e, em seguida, escalona e desloca o resultado usando dois novos parâmetros, γ e β, por camada (mitiga ambos os problemas).
  - **Limitar/podar o gradiente**: consiste em ***limitar/podar os gradientes*** durante a retropropagação do erro para que eles nunca excedam algum limite pré-definido (resolve apenas o problema da explosão do gradiente).

Conectando Neurônios

- Existem basicamente duas maneiras distintas para se conectar os ***nós*** de uma rede neural, ***direta*** e ***reversa***.
- Na figura ao lado, os ***nós*** da rede têm conexões em apenas uma única direção.
- Esse tipo de rede é conhecida como ***rede de alimentação direta*** (do inglês, *feedforward*) ou ***sem realimentação***.
- O sinal percorre a rede em uma única direção, da entrada para a saída.
- Os ***nós*** da mesma camada ***não são conectados entre si***.
- Esse tipo de rede representa uma ***função de suas entradas atuais*** e, portanto, ***não possui um estado interno além dos próprios pesos***.

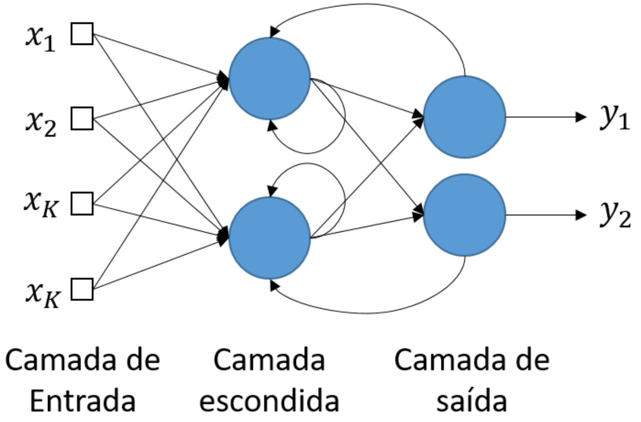


$$\boldsymbol{y} = f(\boldsymbol{x}, \boldsymbol{W})$$

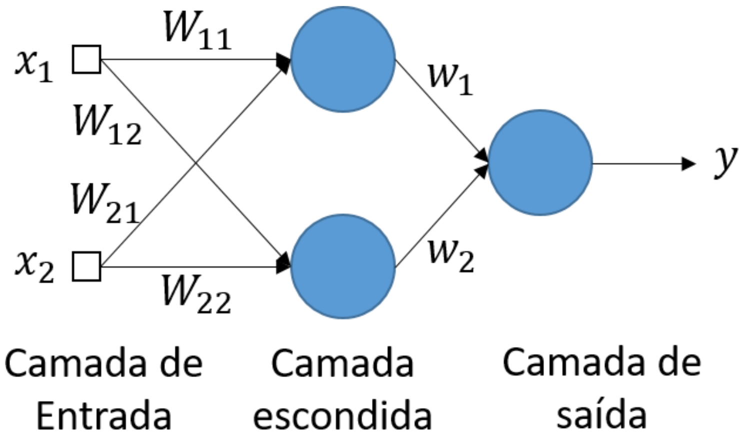
OBS.: A informação se move em apenas uma direção: da entrada, passando pelos nós ocultos indo em direção aos nós de saída. Não há ciclos ou loops neste tipo de rede.

Conectando Neurônios

- Na figura ao lado, os ***nós*** da rede têm conexões em 2 direções, desta forma, o sinal percorre a rede nas direções ***direta e reversa***.
- Este tipo de rede é conhecida como ***rede recorrente*** ou ***rede com realimentação***.
- Nessas redes, a saída dos ***nós*** alimentam ***nós*** da mesma camada (inclusive o próprio ***nó***) ou de camadas anteriores.
- Isso significa que a rede forma um ***sistema dinâmico*** que pode atingir um ***estado estável, exibir oscilações ou mesmo um comportamento caótico, ou seja, divergir***.
- Além disso, a saída da rede é ***função da entrada atual e de seu estado interno***, ou seja, de entradas anteriores.
- Portanto, ***redes recorrentes*** possuem memória.
- Essas redes são úteis para o ***processamento de dados sequenciais***, como séries temporais (e.g., sons, preços de ações, padrões cerebrais, etc.) ou linguagem natural (e.g., escrita e fala).



Regressão Não-Linear



- A rede MLP ao lado tem sua saída definida por

$$y = f\left(\boldsymbol{w}^T f(\boldsymbol{W}^{\mathrm{T}}\boldsymbol{x})\right),$$

onde $f(.)$ é a ***função de ativação*** escolhida, $W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}$ e $\boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$,

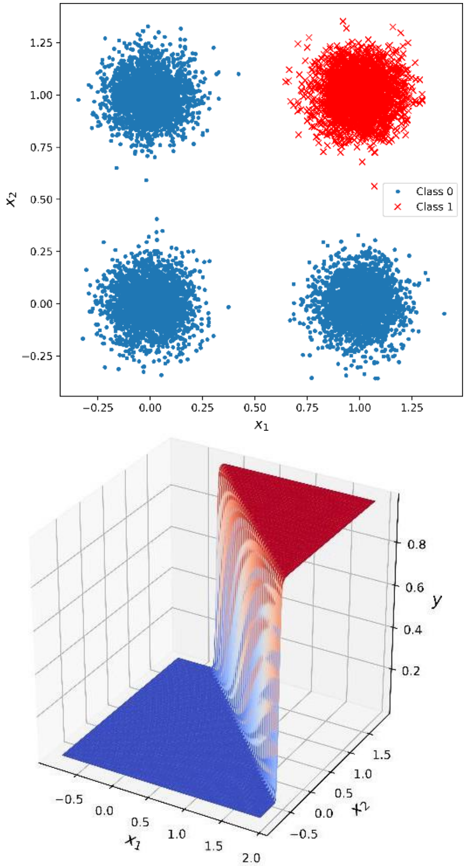
- Percebam que a saída da rede é dada pelo ***aninhamento*** das saídas de ***funções de ativação não-lineares***.
- Sendo assim, as funções que uma rede neural pode representar podem ser ***altamente não-lineares*** dependendo da quantidade de camadas e nós.
- Portanto, redes neurais podem ser vistas como ferramentas para a realização de ***regressão não-linear***, mas também podemos resolver problemas de classificação.
- Com uma única camada oculta suficientemente grande, é possível representar ***qualquer função contínua*** das entradas com uma precisão arbitrária (depende da topologia).
- Com duas camadas ocultas, até ***funções descontínuas*** podem ser representadas.
- Portanto, dizemos que as redes neurais possuem ***capacidade de aproximação universal*** de funções.
- Veremos alguns exemplos desta capacidade de aproximação a seguir.

Aproximação universal de funções

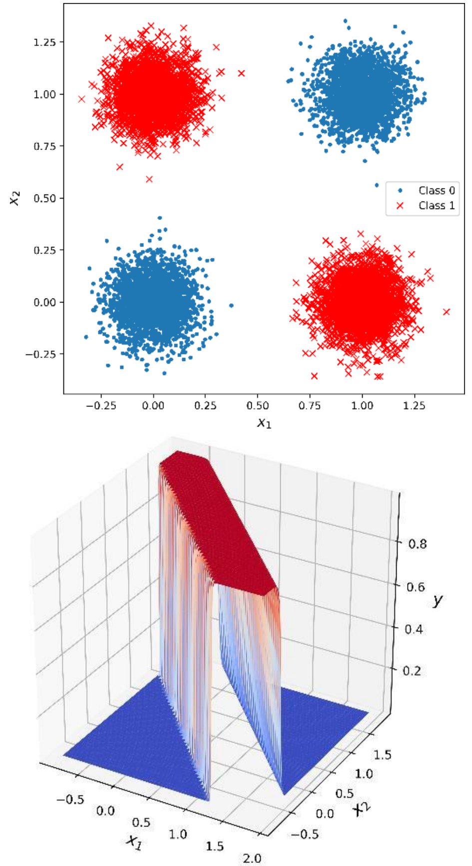
[Exemplo: FunctionApproximationWithMLP.ipynb]

- 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 em formato de onda.
- Fig. 3: Combinando duas ondas perpendiculares, nós obtemos uma função em formato cilíndrico.

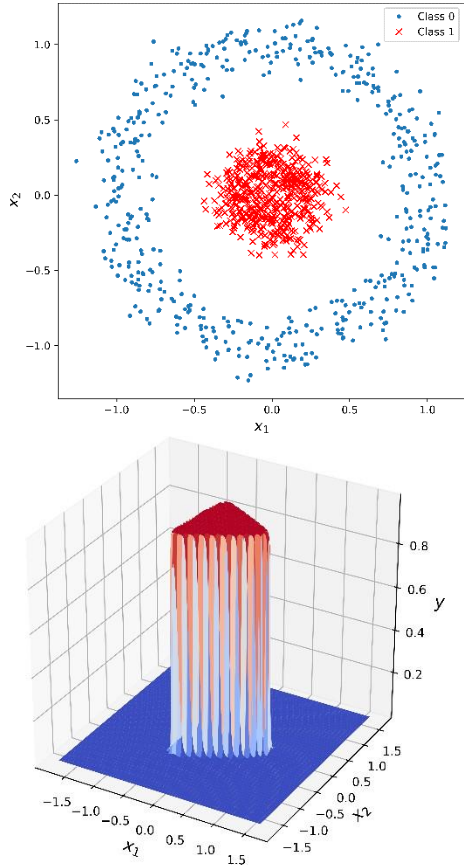
Função AND: MLP com 0 camadas escondidas, apenas um neurônio na camada de saída. Total: 1 nó.



Função XOR: MLP com 1 camada escondida com 2 nós. Total: 3 nós.



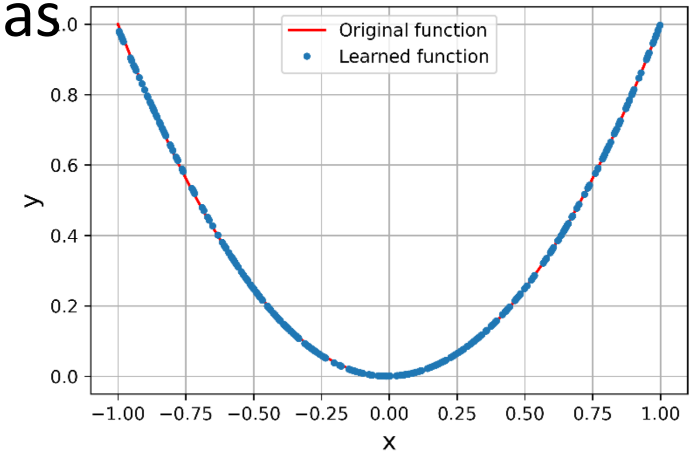
Círculos concêntricos: MLP com 1 camada escondida com 4 nós. Total: 5 nós.



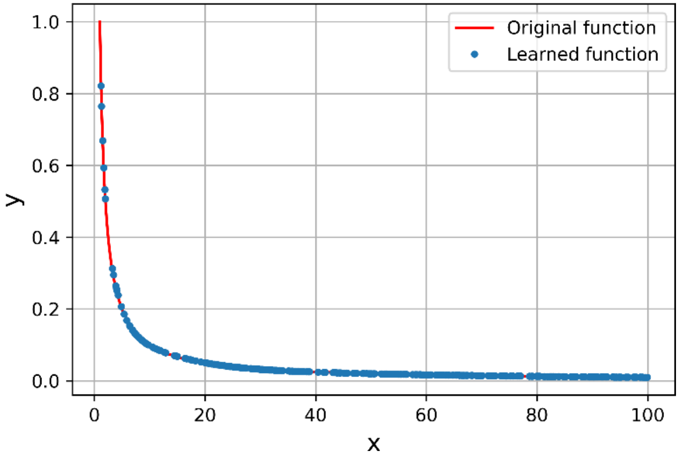
Aproximação universal de funções

- Redes neurais podem ser usadas para aproximar funções como as mostradas abaixo:
  - $f(x) = x^2, \ -1 \leq x \leq 1,$
  - $f(x) = \frac{1}{x}, \ 1 \leq x \leq 100,$
  - $f(x) = \sin(x),, \ 1 \leq x \leq 2\pi,$
- **Exercício**: usar as classes [MLPRegressor]() e [GridSearchCV]() da biblioteca SciKit-Learn para encontrar o número de nós necessários na camada escondida para que uma rede neural aproxime estas funções.

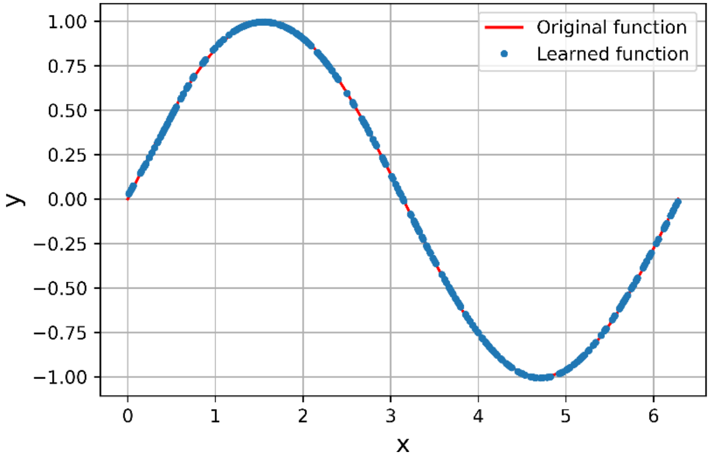
$$f(x) \ = \ x^2$$



$$f(x) \ = \frac{1}{x}$$



$$f(x) \ = \sin(x)$$



Aprendizado em Redes Neurais

- Consideramos agora, o ***processo de otimização***, ou seja, de ***atualização dos pesos sinápticos***.
- Assim como vimos anteriormente, o processo de otimização corresponde a um ***problema de minimização*** de uma ***função de erro (ou de custo ou perda)***, $J(\boldsymbol{w})$, ***com respeito a um vetor de pesos*** $\boldsymbol{w}$.
- Portanto, o problema de aprendizado em redes neurais pode ser formulado como

$$\min_{\boldsymbol{w}} J(\boldsymbol{w})$$

- Normalmente, 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).
- Existem ***vários métodos de otimização*** aplicáveis, mas, sem dúvida, ***os mais utilizados são aqueles baseados nas derivadas da função custo***, $J(\boldsymbol{w})$.

Aprendizado em Redes Neurais

- Dentre esses métodos, existem os de ***primeira ordem*** e os de ***segunda ordem***.
- Os métodos de ***primeira ordem*** são baseados nas derivadas parciais de primeira ordem da ***função custo***, agrupadas no ***vetor gradiente***:

$$\nabla J(\boldsymbol{w}) = \begin{bmatrix} \frac{\partial j(\boldsymbol{w})}{\partial w_1} \\ \frac{\partial J(\boldsymbol{w})}{\partial w_2} \\ \vdots \\ \frac{\partial j(\boldsymbol{w})}{\partial w_K} \end{bmatrix}$$

- Como já vimos, o ***gradiente aponta na direção de maior crescimento da função*** e portanto, ***caminhar em sentido contrário*** a ele é uma forma adequada de se ***buscar iterativamente a minimização*** da ***função de custo***.

Aprendizado em Redes Neurais

- Desta maneira, temos a seguinte ***equação de atualização dos pesos***

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

onde α é o ***passo de aprendizagem*** e k é a iteração de atualização.

- Já os métodos de ***segunda ordem***, são baseados na informação trazida pela ***derivada parcial de segunda ordem da função custo***. Essa informação está contida na ***matriz Hessiana***, $\boldsymbol{H}$:

$$\boldsymbol{H}(\boldsymbol{w}) = \nabla^2 J(\boldsymbol{w}) = \begin{bmatrix} \frac{\partial^2 J(\boldsymbol{w})}{\partial w_1^2} & \frac{\partial^2 J(\boldsymbol{w})}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 J(\boldsymbol{w})}{\partial w_1 \partial w_K} \\ \frac{\partial^2 J(\boldsymbol{w})}{\partial w_2 \partial w_1} & \frac{\partial^2 J(\boldsymbol{w})}{\partial w_2^2} & \cdots & \frac{\partial^2 J(\boldsymbol{w})}{\partial w_2 \partial w_K} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 J(\boldsymbol{w})}{\partial w_K \partial w_1} & \frac{\partial^2 J(\boldsymbol{w})}{\partial w_K \partial w_2} & \cdots & \frac{\partial^2 J(\boldsymbol{w})}{\partial w_K^2} \end{bmatrix}$$

OBS.: A matriz Hessiana é uma matriz quadrada com dimensões $K \times K$.

Aprendizado em Redes Neurais

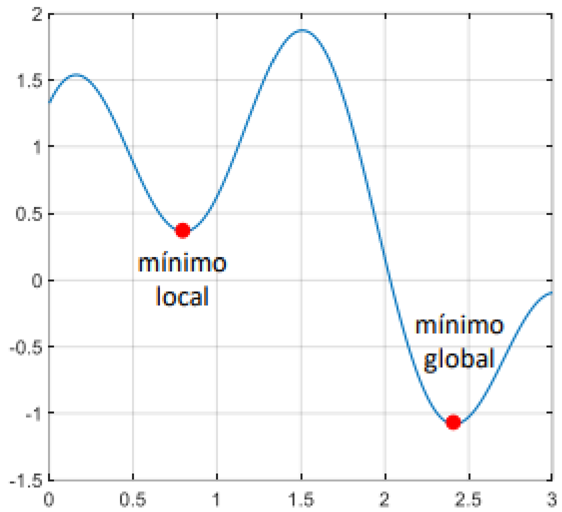
- De posse da ***matriz Hessiana***, é possível fazer uma ***aproximação de Taylor de segunda ordem*** da ***função de custo***, o que leva à seguinte expressão para adaptação dos pesos:

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

- Essa expressão requer que a ***matriz Hessiana*** seja ***inversível*** e ***definida positiva*** a cada iteração, k, i.e., $\boldsymbol{z}^{\boldsymbol{T}}\boldsymbol{H}\boldsymbol{z} > 0, \forall \boldsymbol{z} \neq \boldsymbol{0}$ (vetor nulo).
- A aproximação de Taylor com ***informação de segunda ordem é mais precisa*** que a fornecida por métodos de primeira ordem.
- Portanto, a tendência é que métodos de ***segunda ordem*** convirjam em ***menos passos que métodos de primeira ordem***.
- Entretanto, o cálculo exato da ***matriz Hessiana*** pode ser complicado em vários casos práticos.
  - Por exemplo, se tivermos 10 pesos para otimizar, a matriz Hessiana teria 10x10 elementos. Portanto, essa abordagem direta não é eficiente se o número de pesos for muito grande.
- 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.

Mínimos Locais, Globais, Pontos de Sela e Platôs

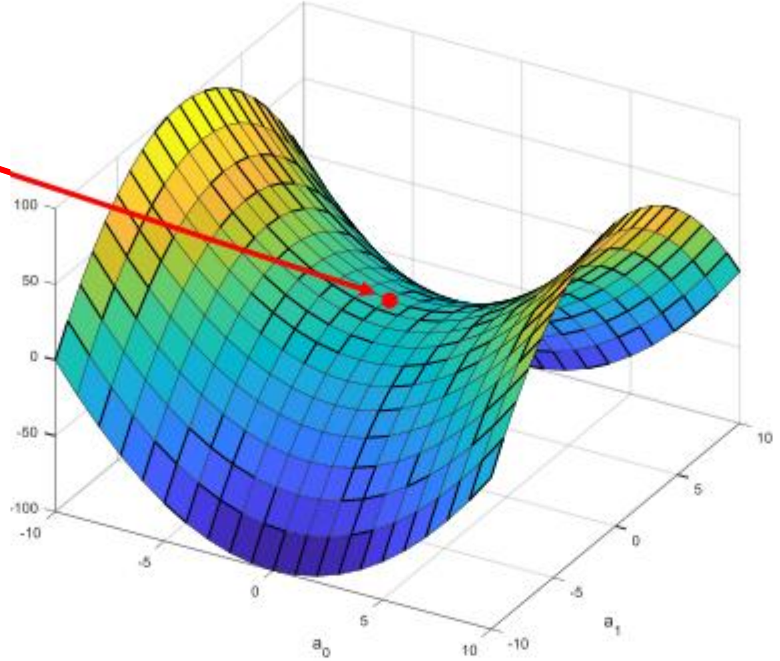
- É importante ressaltarmos que todos esses métodos são métodos de ***busca local***, ou seja, eles têm ***convergência assegurada para mínimos locais***.
- Um ***mínimo*** (local ou global) sempre ***atrai*** o vetor de pesos quando este se encontra em sua vizinhança.
- Para relembrarmos o que é um mínimo local, vejamos a figura ao lado onde existem dois mínimos:
  - Um deles é uma ***solução ótima em relação apenas a seus vizinhos***, ou seja, um ***mínimo local***.
  - O outro também é uma solução ótima em relação a seus vizinhos (***mínimo local***), mas também ***em relação a todo o domínio da função de custo***. Este é um ***mínimo global***.
- 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***.



IMPORTANTE: Para muitos problemas envolvendo redes neurais, quase todos os mínimos locais têm um valor muito semelhante ao do mínimo global e, portanto, encontrar um mínimo local já é bom o suficiente para um dada problema.

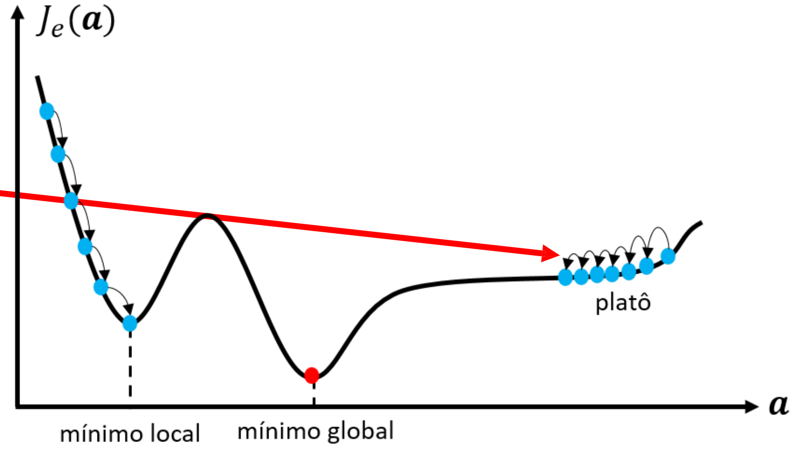
Mínimos Locais, Globais, Pontos de Sela e Platôs

- Outra irregularidade que podemos encontrar são os chamados ***pontos de sela***:
  - 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 minimização da função de custo 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*** ou ***versões ruidosas do gradiente descendente***, como, por exemplo, o ***Gradiente Descendente Estocástico***.



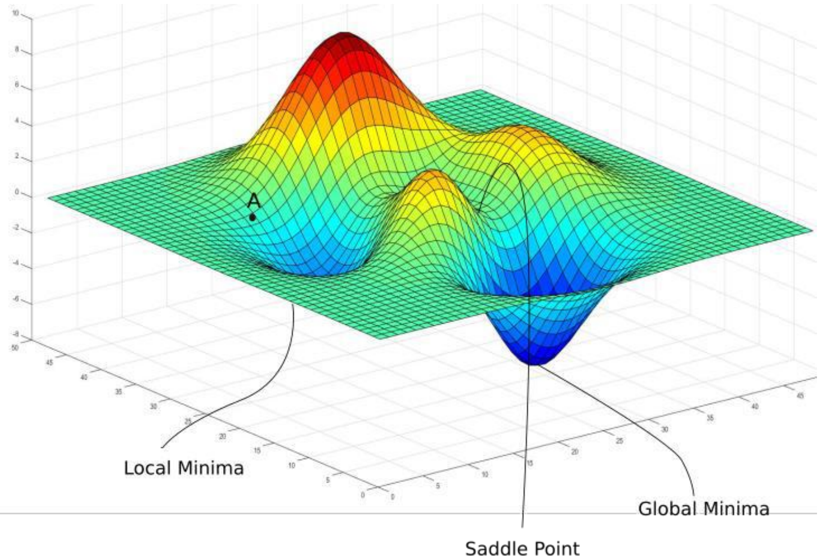
Mínimos Locais, Globais, Pontos de Sela e Platôs

- Outro tipo de irregularidade são os ***platôs***: regiões planas, mas com erro elevado.
  - Como a inclinação nesta região é próxima de zero (consequentemente o gradiente é próximo de zero) o algoritmo pode levar muito tempo para atravesá-la.



- Para se escapar destas regiões, usa-se métodos de ***aprendizado adaptativo*** como AdaGrad, RMSProp, Adam, etc.
- Portanto, como garantir que o mínimo encontrado é bom o suficiente?
  - Treina-se o modelo várias vezes, sempre inicializando os ***pesos aleatoriamente***, com a esperança de que em alguma dessas vezes ele inicialize mais próximo do mínimo global ou de um bom mínimo local.

Exemplo da superfície de erro de uma rede neural



Retropropagação do Erro

- Conforme nós discutimos anteriormente, os métodos fundamentais de ***aprendizado*** para ***redes neurais*** são baseados no cálculo das ***derivadas parciais*** da ***função de erro*** (ou de ***custo/perda***) com relação aos ***pesos sinápticos***.
- Esses métodos têm como objetivo encontrar o ***conjunto de pesos sinápticos*** que minimize a ***métrica (função) de erro*** escolhida.
- Para isso, é necessário encontrar uma maneira de se calcular o ***vetor gradiente*** da ***função de custo*** com respeito aos ***pesos sinápticos das várias camadas de uma rede neural***.
- Essa tarefa pode parecer óbvia, 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 RNAs.

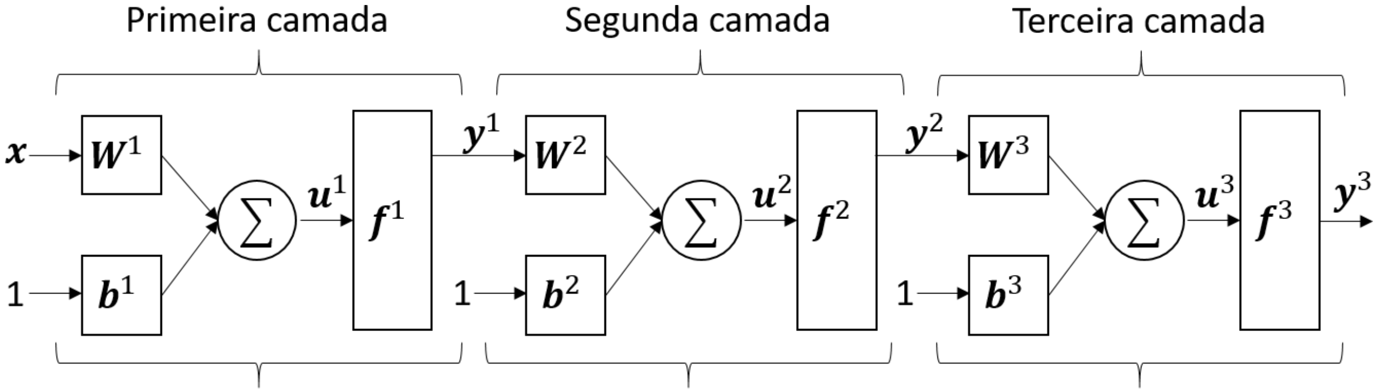
Retropropagação do Erro

- Para que entendamos melhor o porquê de não ser uma tarefa trivial, nós iremos considerar a notação abaixo, a qual será muito útil 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 $\boldsymbol{W}^{m}$ é a matriz 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 $\boldsymbol{b}^{m}$ 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 $\boldsymbol{u}^{m}$é o ***vetor de ativações*** com as ***combinações lineares*** das entradas de todos os nós da m-ésima camada.
  - $\boldsymbol{f}^{m}(.)$ é a função de ativação da m-ésima camada da ***rede neural***.
  - Com essa notação, obter o ***vetor gradiente*** significa calcular, de maneira genérica,

$\frac{\partial J(\boldsymbol{w})}{\partial w_{i,j}^{m}}$ ou seja, calcular essa derivada para todos os pesos de todos os ***nós***.

Retropropagação do Erro

- A figura abaixo apresenta um exemplo de como uma rede MLP pode ser descrita segundo essa notação.



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

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 por:

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

- Para facilitar nosso trabalho, iremos supor, sem nenhuma perda de generalidade, que a ***função de custo*** escolhida é o ***erro quadrático médio*** (MSE).

Retropropagação do Erro

- Nós vamos assumir que a ***última camada da rede MLP*** (definida como a M-ésima camada) tenha uma quantidade genérica, N_M, de ***nós***. Assim, o MSE é dado por

$$j(\boldsymbol{w}) = \frac{1}{N_{\text{dados}} N_M} \sum_{n=1}^{N_{\text{dados}}} \sum_{j=1}^{N_M} e_j^2(n)$$

$$= \frac{1}{N_{\text{dados}} N_M} \sum_{n=1}^{N_{\text{dados}}} \sum_{j=1}^{N_M} \left(d_j(n) - y_j^M(n) \right)^2$$

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.

- Para ***treinar a rede*** (i.e., atualizar os pesos), devemos **derivar a função custo com respeito aos pesos sinápticos**.
- Porém, percebam que os ***pesos das camadas ocultas não aparecem explícitamente*** na expressão do erro, $J(\boldsymbol{w})$, apenas os da camada de saída, como veremos a seguir.

Retropropagação do Erro

- Para fazer com que a dependência dos pesos apareça de maneira clara na expressão do erro, nós precisamos recorrer a aplicações sucessivas da ***regra da cadeia***.
- Usando a notação de ***Leibniz***, essa regra nos mostra que:

$$\frac{\partial f\left(g\big(h(x)\big)\right)}{\partial x} = \frac{\partial f\left(g\big(h(x)\big)\right)}{\partial g\big(h(x)\big)} \frac{\partial g\big(h(x)\big)}{\partial h(x)} \frac{\partial h(x)}{\partial x}$$

- Por exemplo, vamos considerar que $f\big(g(x)\big) = e^{x^2}$ e que queremos obter $\frac{\partial f\left(g(x)\right)}{\partial x}$.
- Nós podemos fazer $g(x) = x^2$ e usar a ***regra da cadeia***:

$$\frac{\partial f\big(g(x)\big)}{\partial x} = \frac{\partial f\big(g(x)\big)}{\partial g(x)} \frac{\partial g(x)}{\partial x} = e^{g(x)} 2x = 2xe^{x^2}.$$

Retropropagação do Erro

$$j(\boldsymbol{w}) = \frac{1}{N_{\text{dados}} N_M} \sum_{n=1}^{N_{\text{dados}}} \sum_{j=1}^{N_M} \left(d_j(n) - y_j^M(n)\right)^2$$

- Agora voltamos à equação do MSE e vemos que ***as saídas da M-ésima camada (i.e., saída) da rede aparecem de maneira direta na equação***.
- Isso significa que é ***simples se obter as derivadas com respeito aos pesos desta camada***.
- Porém, quando precisamos avaliar as ***derivadas com respeito aos pesos das camadas anteriores (i.e., ocultas)***, a situação fica mais complexa, pois não existe uma dependência direta.
- Portanto surge a pergunta, como podemos atribuir a cada ***nó*** de uma camada oculta da rede, e, consequentemente a seus pesos, sua devida influência na composição dos valores de saída e, consequentemente, do erro?
  - 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***.

Retropropagação do Erro

- A seguir, veremos de maneira mais ***sistemática*** como a ***retropropagação do erro*** é realizada.
- Inicialmente, nós devemos observar um fato fundamental. O cálculo da derivada do MSE com respeito a um peso qualquer é dada por:

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

OBS.: mudei o índice do erro de j para k.

  - **OBS.1**: Operação da derivada parcial é ***distributiva***.
  - **OBS.2**: A divisão pelo número de amostras é omitida, pois não afeta a otimização.
- A equação acima 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***).

Retropropagação: Algumas noções básicas

- Considerando a derivada geral $\frac{\partial J(\boldsymbol{w})}{\partial w_{i,j}^{m}}$ (i.e., derivada para um peso genérico) e usando a ***regra da cadeia***, podemos reescrevê-la como:

Ativação do nó ao qual o peso pertence.

$$\frac{\partial J(\boldsymbol{w})}{\partial w_{i,j}^{m}} = \frac{\partial J(\boldsymbol{w})}{\partial u_{i}^{m}} \frac{\partial u_{i}^{m}}{\partial w_{i,j}^{m}}$$

- A primeira derivada após a igualdade é a derivada da ***função de custo*** com respeito à ***ativação***,u_i^m, do i-ésimo ***nó*** da m-ésima camada.
- Essa grandeza será chamada de ***sensibilidade*** e é denotada pela letra grega δ. Desta forma:

$$\delta_i^m = \frac{\partial J(\boldsymbol{w})}{\partial u_i^m}.$$

Sensibilidade do i-ésimo nó da m-ésima camada.

- O termo δ_i^mé único para cada ***nó*** da m-ésima camada.
- O outro termo, por sua vez, varia ao longo das entradas do ***nó*** em questão. Como adotamos nós do ***tipo perceptron***, a ativação, u_i^m, é uma ***combinação linear*** das entradas

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

Retropropagação: Algumas noções básicas

- Assim

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

Saída da camada anterior conectada ao i-ésimo nó da m-ésima camada através do peso $w_{i,j}^m$

- Caso a derivada seja em relação ao termo de ***bias***, b_i^m, teremos o seguinte resultado

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

- Desta forma, vemos que ***todas as derivadas da função de custo em relação aos pesos (sinápticos/bias) são produtos de uma sensibilidade, δ_i^m, por uma entrada do i-ésimo nó da rede (ou, no caso dos termos de bias, pela unidade)***.

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

ou, para o peso de bias, b_i^m

$$\frac{\partial J(\boldsymbol{w})}{\partial b_{i,}^m} = \frac{\partial J(\boldsymbol{w})}{\partial u_i^m}\frac{\partial u_i^m}{\partial b_i^m} = \delta_i^m.$$

- São os valores de ***sensibilidade***, δ_i^m, que trazem mais 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 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 camadas ocultas no erro de saída.
- 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, $\delta_i^m, \forall i$, em um vetor, $\boldsymbol{\delta}^m$.
- Em seguida, vamos encontrar uma regra que fará a transição $\boldsymbol{\delta}^m \rightarrow \boldsymbol{\delta}^{m-1}$.
- Ou seja, a partir da ***sensibilidade*** da camada m, iremos encontrar a ***sensibilidade*** da camada anterior, $m - 1$.

Retropropagando o erro

- Em resumo, o processo de ***retropropagação do erro*** é iniciado calculando-se o ***vetor de sensibilidades*** da última camada, $\boldsymbol{\delta}^M$, e, de maneira ***recursiva***, obtém-se os ***vetores de sensibilidades*** de todas as camadas anteriores.
- Para calcular $\boldsymbol{\delta}^M$ (vetor de sensibilidades da camada de saída) consideramos N_M saídas e, assim, temos que o j-ésimo elemento de $\boldsymbol{\delta}^M$ é dado por:

$$S_j^M = \frac{\partial e_j^2}{\partial u_j^M} = \frac{\partial\left(d_j - y_j^M\right)^2}{\partial u_j^M} \overset{\text{Regra da cadeia}}{=} \frac{\partial\left(d_j - y_j^M\right)^2}{\partial y_j^M}\frac{\partial y_j^M}{\partial u_j^M} = -2\left(d_j - y_j^M\right)\frac{\partial y_j^M}{\partial u_j^M}$$

$$-2\left(d_j - y_j^M\right)f'^M\left(u_j^M\right),$$

onde

$$y_j^M = f^M\left(u_j^M\right),$$

$$f'^M\left(u_j^M\right) = \frac{\partial f^M\left(u_j^M\right)}{\partial u_j^M}.$$

Função logística

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

Função tangente hiperbólica

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

Retropropagando o erro

- Matricialmente nós podemos expressar $\boldsymbol{\delta}^M$como:

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

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

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

$\boldsymbol{d}$ e $\boldsymbol{y}$ são vetores 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 simples e dada por

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

Tarefa

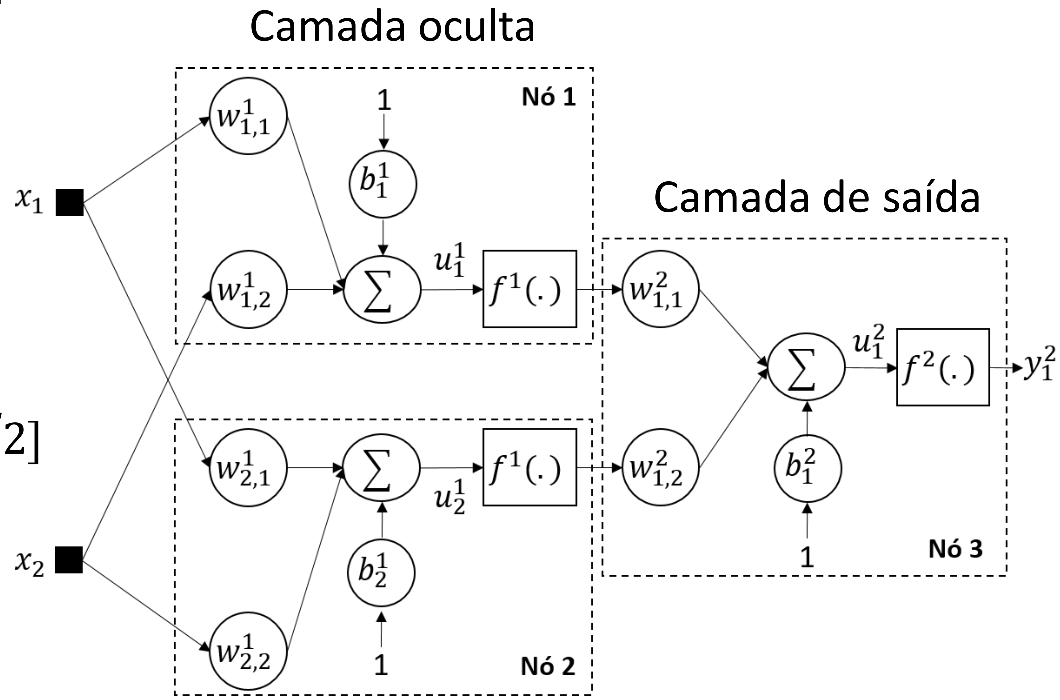
- Encontrem o vetor gradiente para todos os pesos do nó 1 (camada 1) da rede neural do próximo slide.

$$\begin{bmatrix} \dfrac{\partial J(\boldsymbol{w})}{\partial w_{1,1}^{1}} \\ \dfrac{\partial J(\boldsymbol{w})}{\partial w_{1,2}^{1}} \\ \dfrac{\partial J(\boldsymbol{w})}{\partial b_{1}^{1}} \end{bmatrix} = ?$$

- **OBS**.: Podem 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 retropropagação do erro

- Considerem uma rede MLP com uma camada oculta com dois nós e uma camada de saída com um único nó, portanto $M = 2$.
- Devemos começar calculando $\boldsymbol{\delta}^2$.
- Percebam que essa ***sensibilidade*** é um escalar pois há apenas um ***nó*** na camada de saída.
- Vamos considerar um exemplo de entrada $\boldsymbol{x} = [x_1, x_2]$ e saída desejada d.
- Supomos que os pesos de todos os nós têm uma certa configuração inicial (e.g., dist. normal).
- Assim, quando a entrada, $\boldsymbol{x}$, é apresentada à rede, é possível calcular todos os valores de interesse ao longo dela até sua saída.
- Essa é a etapa ***direta*** (ou do inglês, ***forward***).



Exemplo da retropropagação do erro,

- Portanto, temos então a saída y_1^2, onde o erro pode ser calculado como

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

- De posse do erro, podemos calcular a sensibilidade do ***nó*** da camada de saída

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

- Temos, portanto, nossa primeira ***sensibilidade***. Agora, usamos a equação de recursão para ***retropropagar*** o erro até a camada anterior. A fórmula nos diz:

$$\boldsymbol{\delta}^1 = \boldsymbol{F}'^1(\boldsymbol{u}^1)(\boldsymbol{W}^2)^{\mathrm{T}} \delta^2.$$

onde $(\boldsymbol{W}^2)^{\mathrm{T}} = \begin{bmatrix} w_{1,1}^2, & w_{1,2}^2 \end{bmatrix}^{\mathrm{T}}$

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

OBS.: Notem que $.^2$ aqui não significa "ao quadrado", mas sim a indicação de que se trata de uma saída da camada $m = 2$.

Exemplo da retropropagação do erro

- Portanto,

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

- Agora, para obtermos o vetor gradiente, multiplicamos as ***sensibilidades*** pelas entradas correspondentes.
- Por exemplo, as derivadas parciais com relação aos pesos do ***nó*** $i = 1$ da camada $m = 1$ são mostradas abaixo

$$\begin{bmatrix} \frac{\partial J(\boldsymbol{w})}{\partial w_{1,1}^1} \\ \frac{\partial J(\boldsymbol{w})}{\partial w_{1,2}^1} \\ \frac{\partial J(\boldsymbol{w})}{\partial b_1^1} \end{bmatrix} = \delta_1^1 \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} = \delta^2 w_{1,1}^2 f'^1(u_1^1) \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} = -2(d - y_1^2) f'^2(u_1^2) w_{1,1}^2 f'^1(u_1^1) \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}$$

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

Exemplo da retropropagação do erro

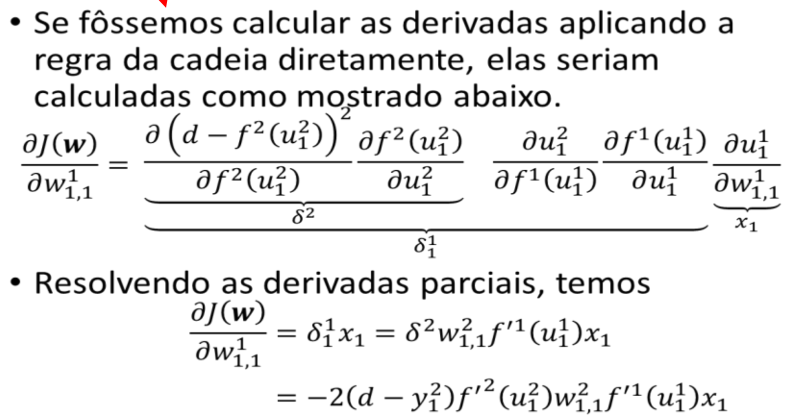
- Se fôssemos calcular as derivadas aplicando a regra da cadeia diretamente, elas seriam calculadas como mostrado abaixo.

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

- Resolvendo as derivadas parciais, temos

$$\frac{\partial J(\boldsymbol{w})}{\partial w_{1,1}^{1}} = \delta_1^1 x_1 = \delta^2 w_{1,1}^2 f'^{1}(u_1^1) x_1$$
$$= -2(d - y_1^2) f'^{2}(u_1^2) w_{1,1}^2 f'^{1}(u_1^1) x_1$$

Derivada com relação ao primeiro peso do nó 1 da camada 1.



- Se fôssemos calcular as derivadas aplicando a regra da cadeia diretamente, elas seriam calculadas como mostrado abaixo.

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

- Resolvendo as derivadas parciais, temos

$$\frac{\partial J(\boldsymbol{w})}{\partial w_{1,1}^{1}} = \delta_1^1 x_1 = \delta^2 w_{1,1}^2 f'^{1}(u_1^1) x_1$$
$$= -2(d - y_1^2) f'^{2}(u_1^2) w_{1,1}^2 f'^{1}(u_1^1) x_1$$

Exemplo da retropropagação do erro

- Aplicando-se o mesmo procedimento aos outros pesos, temos:

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

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

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

Algumas visões práticas de algoritmos de aprendizado

- Podemos dizer que os ***elementos básicos do aprendizado de máquina através de redes neurais foram apresentados até aqui***.
- Porém, existem importantes aspectos práticos que devem ser comentados de modo que vocês fiquem mais familiarizados com as práticas atuais.
- Começamos falando da questão do cálculo do ***vetor gradiente***.

Algumas visões práticas de algoritmos de aprendizado

Versões Online, Batch e Minibatch

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

Algumas visões práticas de algoritmos de aprendizado

Versões Online, Batch e Minibatch

- Vimos também que se calcula o gradiente associado a cada exemplo de entrada e que a média de todos esses ***gradientes locais*** leva ao gradiente para o conjunto total de exemplos.

Gradiente local

$$\frac{\partial J(\boldsymbol{X}|\boldsymbol{W})}{\partial w_{i,j}^{m}} = \frac{1}{N_{\text{dados}} N_M} \sum_{n=1}^{N_{\text{dados}}} \sum_{j=1}^{N_M} \boxed{\frac{\partial e_j^2(n)}{\partial w_{i,j}^{m}}} = \frac{1}{N_{\text{dados}}} \sum_{n=1}^{N_{\text{dados}}} \nabla J_n(\boldsymbol{W})$$

- O ***gradiente local***, é a derivada parcial do erro da j-ésima saída da rede neural para o n-ésimo exemplo de entrada em relação ao peso, $w_{i,j}^{m}$.
- $\nabla J_n(\boldsymbol{W})$ é a média dos N_M ***gradientes locais*** para o n-ésimo exemplo de entrada.
- No entanto, surge aqui um questionamento interessante: o que é melhor, usar o ***gradiente local 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***?

Algumas visões práticas de algoritmos de aprendizado

Versões Online, Batch e Minibatch

- Nesse questionamento, existem três abordagens: o cálculo ***online*** do gradiente (ou seja, exemplo-a-exemplo), o cálculo em batelada e um meio termo.
- Vejamos inicialmente a noção geral de ***adaptação dos pesos*** com o cálculo ***online*** do gradiente, como expressa o algoritmo abaixo (considerando um método de ***primeira ordem***).

> - Defina valores iniciais para a matriz de pesos $\boldsymbol{W}$ e um passo de aprendizagem α pequeno.
> - Faça $k = 0$ (épocas), $t = 0$ (iterações) e calcule $J\big(\boldsymbol{W}(k)\big)$.
> - Enquanto o critério de parada não for atendido, faça:
>   - Ordene aleatoriamente os exemplos de entrada e saídas correspondentes.
>   - Para n variando de 1 até N, faça:
>   - Apresente o n-ésimo exemplo de entrada à rede.
>   - Calcule $J_n(\boldsymbol{W}(t))$ e $\nabla J_n\,(\boldsymbol{W}(\,t))$.
>   - $\boldsymbol{W}\,(t+1) = \boldsymbol{W}(t) - \alpha \nabla J_n\big(\boldsymbol{W}(t)\big)$.
>   - $t = t + 1$.
>   - $k = k + 1$.
>   - Calcule $J\big(\boldsymbol{W}(k)\big)$

OBS.: $J_n(\boldsymbol{W})$ é a média do erro para as N_M saídas e *n*-ésimo exemplo.

Algumas visões práticas de algoritmos de aprendizado

Versões Online, Batch e Minibatch

- O outro extremo seria utilizar todo o conjunto de dados para calcular o gradiente antes de atualizar os pesos.
- Essa é a ideia por trás da abordagem em ***batelada*** (***batch***). O algoritmo abaixo ilustra a operação correspondente.

> - Defina valores iniciais para a matriz de pesos $\boldsymbol{W}$ e um passo de aprendizagem α pequeno.
> - Faça $k = 0$ (épocas) e calcule $J\big(\boldsymbol{W}(k)\big)$.
> - Enquanto o critério de parada não for atendido, faça:
>   - Para n variando de 1 até N, faça:
>   - Apresente o n-ésimo exemplo de entrada à rede.
>   - Calcule $J_n\big(\boldsymbol{W}(k)\big)$ e calcule e armazene $\nabla J_n\big(\boldsymbol{W}(k)\big)$.
>   - $\boldsymbol{W}\,(k+1) = \boldsymbol{W}(k) - \frac{\alpha}{N}\sum_{n=1}^{N}\nabla J_n\big(\boldsymbol{W}(k)\big)$.
>   - $k = k + 1$.
>   - Calcule $J\big(\boldsymbol{W}(k)\big)$

Algumas visões práticas de algoritmos de aprendizado

Versões Online, Batch e Minibatch

- Nas ***redes neurais profundas*** (ou ***deep learning***), usadas com muita frequência em problemas com enormes conjuntos de dados, a regra é adotar o caminho do meio, usando a abordagem com ***mini-batches***.
- Nesse caso, a adaptação dos ***pesos*** é realizada com um gradiente calculado a partir de um meio-termo entre um exemplo e o número total de exemplos (em geral, este é um valor relativamente pequeno em métodos de ***primeira ordem***).
- **OBS**.: As amostras que compõem um ***mini-batch*** são ***aleatoriamente*** tomadas do conjunto de dados. O algoritmo abaixo ilustra isso.

> - Defina valores iniciais para a matriz de pesos $\boldsymbol{W}$ e um passo de aprendizagem α pequeno e o tamanho, m, do mini-batch
> - Faça $k = 0$ (épocas) e calcule $J\big(\boldsymbol{W}(k)\big)$.
> - Enquanto o critério de parada não for atendido, faça:
>   - Para n variando de 1 até m, faça:
>   - Apresente o n-ésimo exemplo de entrada, amostrado aleatoriamente sem reposição do conjunto de treinamento, à rede.
>   - Calcule $J_n\big(\boldsymbol{W}(k)\big)$ e calcule e armazene $\nabla J_n\big(\boldsymbol{W}(k)\big)$.
>   - $\boldsymbol{W}\,(k+1) = \boldsymbol{W}(k) - \frac{\alpha}{N}\sum_{n=1}^{m} \nabla J_n\big(\boldsymbol{W}(k)\big)$.
>   - $k = k + 1$.
>   - Calcule $J\big(\boldsymbol{W}(k)\big)$

Variações dos algoritmos de otimização dos pesos

- Existem vários algoritmos baseados no ***gradiente*** que podem ser empregados para otimizar os ***pesos*** de uma rede neural.
- Aqui, vamos nos ater a alguns métodos mais usuais na literatura moderna, que se encontra bastante focada no ***apredizado profundo***.

➢ **Métodos do Gradiente Descendente Estocástico (GDE) e Mini-batch**

- Sabemos que o métodos do ***GDE*** e ***mini-batch*** utilizam, respectivamente, um único exemplo e um subconjunto de exemplos tomados aleatoriamente para ***estimar*** o gradiente da ***função custo***.
- Este tipo de estimador é o que gera a noção de ***gradiente estocástico***: ***atualizações não seguem a direção de máxima declividade da superfície de erro e, se o conjunto de treinamento contiver ruído, não convergem para o ponto de mínimo.***
- Porém, eles são amplamente empregados em aprendizado profundo devido à utilização reduzida e configurável de amostras, resultando em menor complexidade computacional.
- Além disso, existem algumas variações em cima deles que melhoram a convergência.

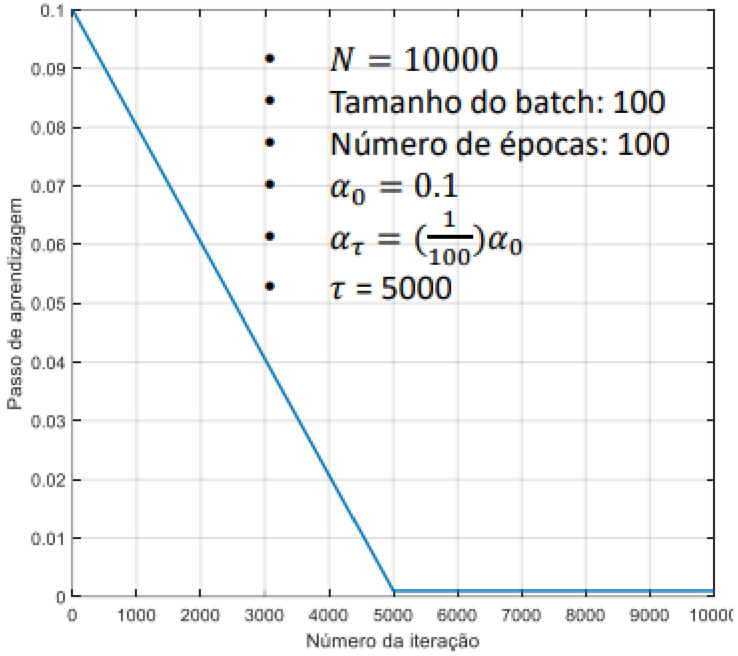
Variações dos algoritmos de otimização dos pesos

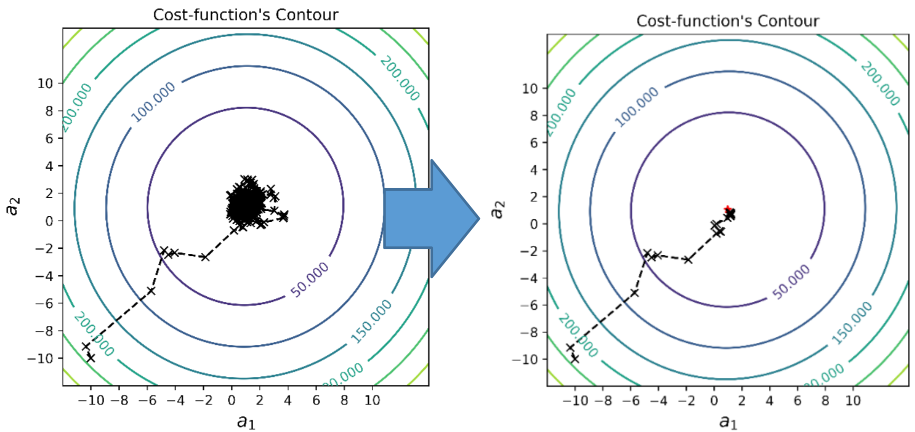
- **Redução programada do passo de aprendizagem**
  - A escolha do ***passo de aprendizagem***, α, é complicada e exige um compromisso entre velocidade de convergência e estabilidade/precisão.
  - Pode-se usar α com um valor fixo, mas, geralmente, se adota uma variação decrescente de um valor α_0 a um valor α_τ (i.e., da iteração 0 à τ-ésima iteração):

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

onde j é o número da iteração de treinamento.

  - Após a τ-ésima iteração, pode-se deixar o valor do passo de aprendizagem fixo, como mostrado na figura ao lado.
  - Porém, a definição dos hiperparâmetros, α_0 e α_τ, é mais um ***problema de otimização de hiperparâmetros***.





Variações dos algoritmos de otimização dos pesos

- **Momentum**
  - O ***termo momento*** é adicionado à ***equação de atualização dos pesos*** para incorporar ***informação do histórico de gradientes anteriores***.
  - Esse termo tem o potencial de ***aumentar a velocidade de convergência*** das versões GDE e em mini-lotes e ***deixá-las mais estáveis***.
  - A ***atualização dos pesos*** com o ***termo momento*** é dada por

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \boldsymbol{v},$$

onde $\boldsymbol{w}$ são os pesos, $\boldsymbol{v}$ é a ***velocidade***, a qual é atualizada da seguinte forma

$$\boldsymbol{v} \leftarrow \mu \boldsymbol{v} + (1 - \mu) \nabla J(\boldsymbol{w}),$$

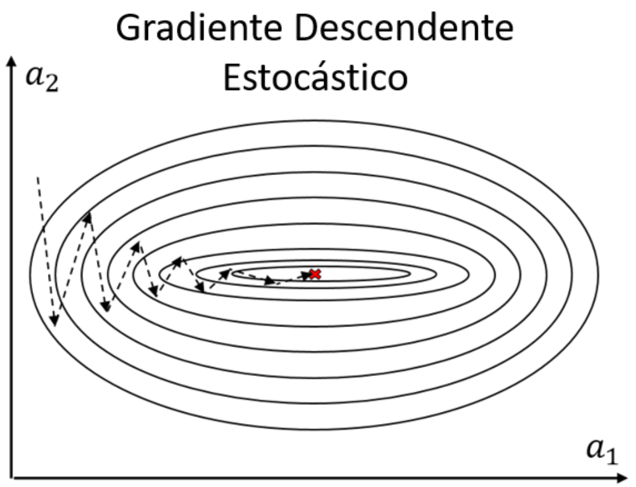
Média móvel exponencialmente decrescente.

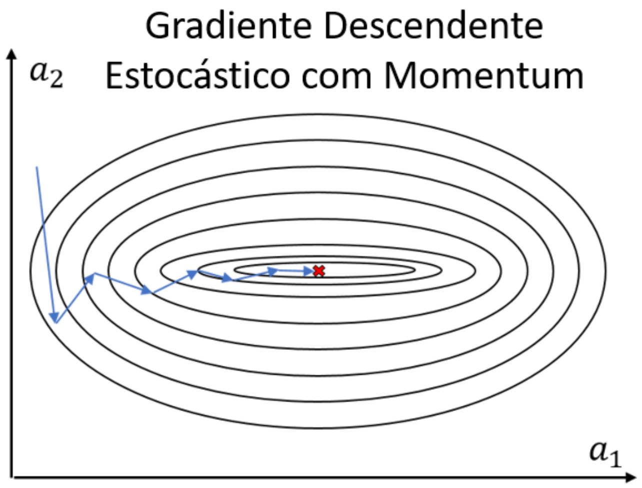
onde, $\nabla J(\boldsymbol{w})$ é o ***vetor gradiente***, α é o ***passo de aprendizagem*** e $\mu \in [0,1)$ é o ***coeficiente de momento*** e determina com que rapidez as contribuições de gradientes anteriores decaem (ou seja, μ é um termo que dita a quantidade de memória).

  - Quanto maior for μ, maior será a influência de gradientes anteriores na direção atual e quanto menor, menor a influência de gradientes anteriores.
  - $\boldsymbol{v}$ dá a ***direção*** e a ***velocidade*** na qual os pesos se movem pelo espaço de pesos.

Variações dos algoritmos de otimização dos pesos

- **Momentum**
  - Em física, ***momento*** é igual a ***massa de uma partícula vezes sua velocidade***.
  - Neste caso, a partícula é o vetor de pesos, $\boldsymbol{w}$.
  - No algoritmo do momento, assumimos que a massa é unitária, então o vetor velocidade, $\boldsymbol{v}$, também pode ser considerado como o momento da partícula.
  - O termo momento adiciona uma média dos gradientes anteriores à atualização corrente, assim:
    - Quando o gradiente aponta na mesma direção por várias iterações, o termo aumenta o tamanho dos passos dados naquela direção.
    - Quando o gradiente muda de direção a cada nova iteração, o termo momento suaviza as variações (figura ao lado).
    - Como resultado, temos ***convergência mais rápida e oscilação reduzida***.





Variações dos algoritmos de otimização dos pesos

- **Momento de Nesterov**
  - O método do ***momento de Nesterov*** é uma variação do ***método do momento*** em que o cálculo do ***vetor gradiente*** não é feito em relação ao vetor de pesos $\boldsymbol{w}$, mas em relação a $\boldsymbol{w} + \mu\boldsymbol{v}$.
  - Essa mudança no cálculo do gradiente faz com que o ***momento de Nesterov*** apresente ***convergência mais rápida e ajustes mais precisos dos pesos*** do que o momento clássico.
- **Modelos com Passo de Aprendizagem Adaptativo**
  - O passo de aprendizagem é um ***hiperparâmetro difícil de ser ajustado de forma ótima*** e ***bastante relevante para o sucesso do treinamento*** de uma rede neural.
  - Isso motivou o surgimento de métodos capazes de ajustá-lo ***dinamicamente***.
  - Esses métodos ajustam o passo de acordo com ***informações dos gradientes passados***.
  - Além disso, pode-se ter ***passos diferentes para cada peso do modelo***, os quais são atualizados de forma independente.
  - Portanto, 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***.
  - Dentre as técnicas mais populares dessa classe estão ***AdaGrad***, ***RMSProp*** e ***Adam.***

Inicialização dos Pesos

- Uma vez que os métodos de treinamento de ***redes neurais MLP*** são iterativos, eles dependem de uma ***inicialização dos pesos***.
- Como os métodos são de ***busca local***, a inicialização pode afetar drasticamente a qualidade da solução obtida.
- O ***ponto de inicialização*** pode determinar se o algoritmo converge, sendo alguns pontos iniciais tão instáveis que o algoritmo encontra dificuldades numéricas (representações numéricas: ***underflow*** e ***overflow***) e falha completamente em convergir (e.g., ***desaparecimento*** e ***explosão*** dos gradientes).
- O ponto de inicialização também pode fazer com que ocorram variações expressivas na ***velocidade de convergência*** (e.g., platôs, pontos de sela).
- 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 às mesmas entradas, devem ter pesos iniciais diferentes, caso contrário, eles terão os mesmos pesos ao longo do treinamento.
- Isso, portanto, sugere uma ***abordagem de inicialização aleatória***.

[1] The importance of effective initialization

Inicialização dos Pesos

- Os pesos iniciais são tipicamente obtidos a partir de ***distribuições gaussianas*** ou ***uniformes***, não importando muito qual é usada.
- No entanto, ***o intervalo de valores da distribuição usada para iniciar os pesos*** tem um efeito significativo tanto no resultado da otimização quanto na capacidade de generalização da rede.
- A ordem de grandeza desses pesos levanta algumas discussões:
  - Pesos de maior magnitude criam uma maior distinção entre ***nós*** (i.e., a ***quebra de simetria***). Por outro lado, isso pode causar problemas de ***instabilidade***.
  - Pesos de maior magnitude favorecem a propagação de informação, porém, por outro lado, causam preocupações do ponto de vista de regularização (***overfitting***).
  - Pesos de magnitude elevada podem levar os ***nós*** com ***funções de ativação*** do tipo sigmóide a operarem na região de saturação, comprometendo a convergência do algoritmo (***desaparecimento do gradiente***).
  - Pesos de magnitude elevada podem levar os ***nós*** com ***funções de ativação*** do tipo RELU à ***explosão do gradiente*** ou dos ***valores de saída***, deixando a rede muito sensível a mudanças dos valores de entrada.
- Portanto, na sequência listamos algumas ***heurísticas*** para inicialização dos pesos.

Inicialização dos Pesos

- A ideia por trás destas heurísticas é ***manter a média das ativações 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	$r = \sqrt{\frac{3}{m}}$	$\sigma^2 = \frac{1}{m}$

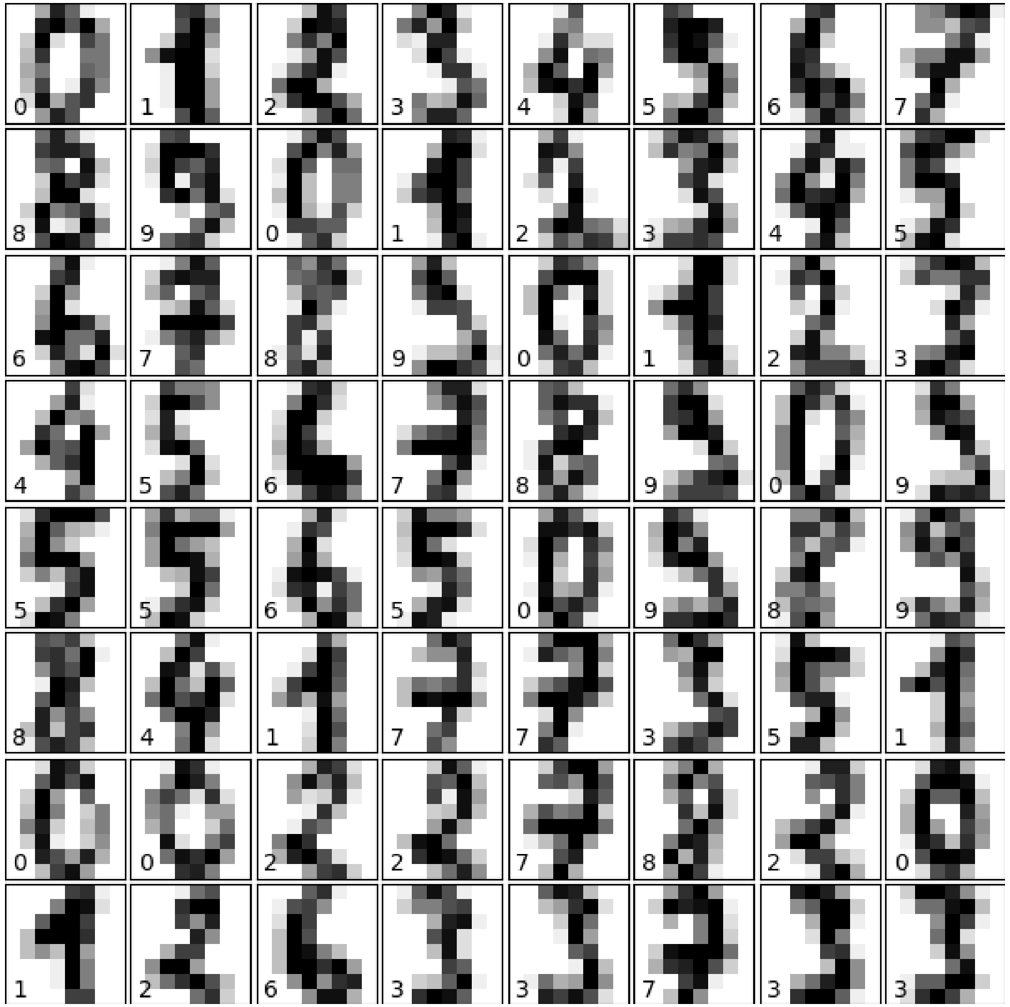
- Uma heurística para a inicialização dos ***pesos de bias*** é inicializá-los com ***valores nulos***. Esta heurística é usada pois se mostra bastante eficiente na maioria dos casos.

[2] How to find appropriate initialization values

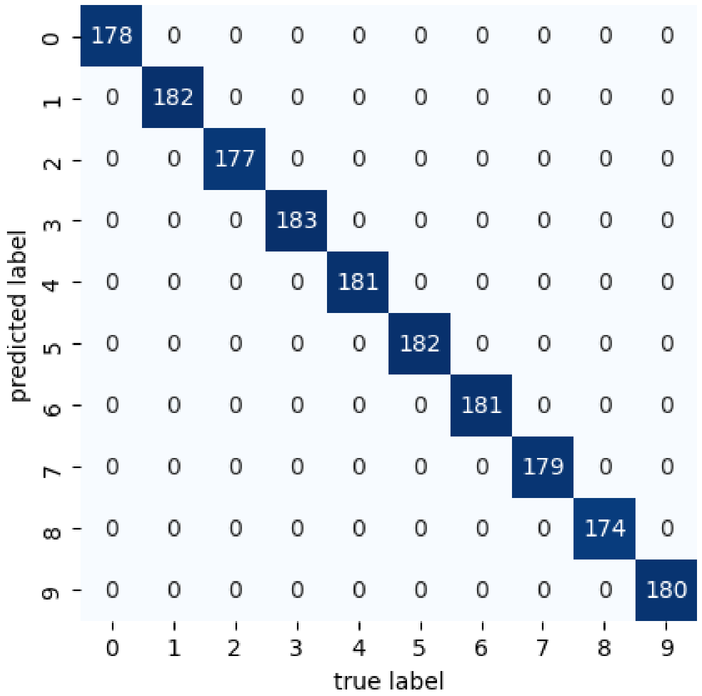
Redes Neurais MLP com SciKit-Learn

- Como vimos anteriormente, a biblioteca *SciKit-Learn* disponibiliza algumas classes para o treinamento de redes neurais *multi-layer perceptron*.
- Entretanto, suas implementações ***não são flexíveis e não se destinam a aplicações de larga escala***.
  - A biblioteca *SciKit-Learn* não oferece suporte a GPUs.
- 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://scikit-learn.org/stable/related_projects.html#related-projects

Classificação com MLPClassifier

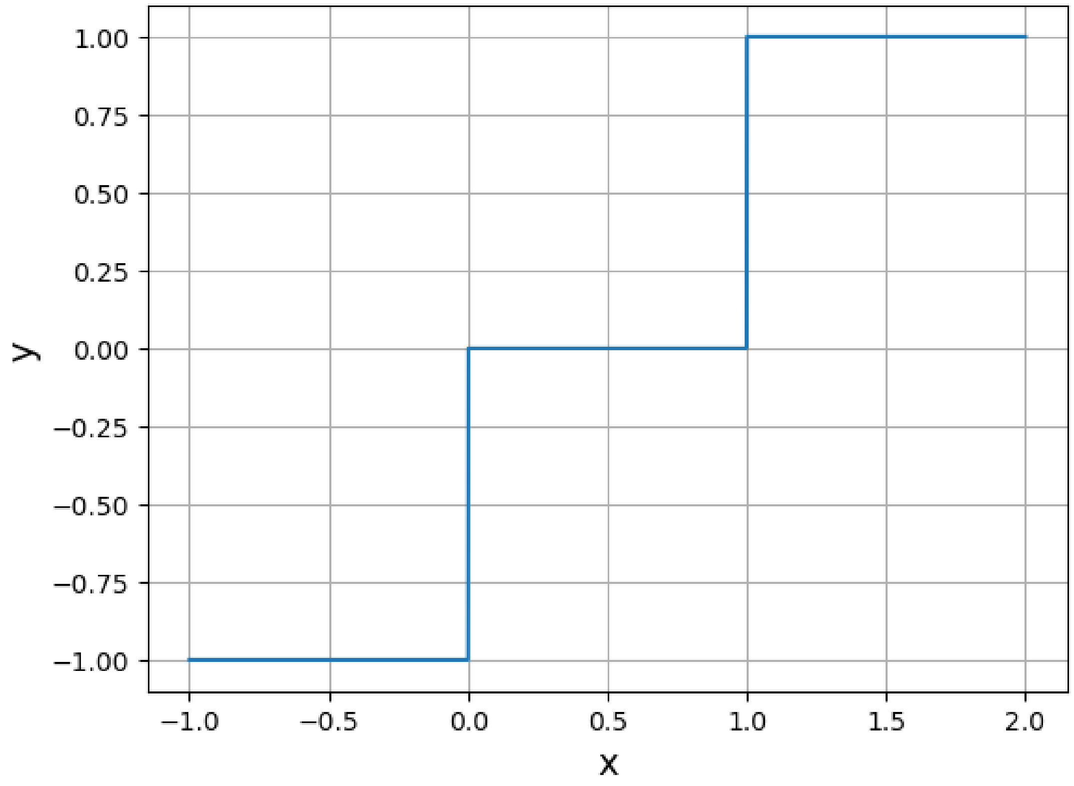


Classificação de dígitos escritos à mão com uma rede MLP.



[Exemplo: classificação_com_MLPClassifier.ipynb]

Regressão com MLPRegressor



Aproximação de função com descontinuidades com uma rede MLP.

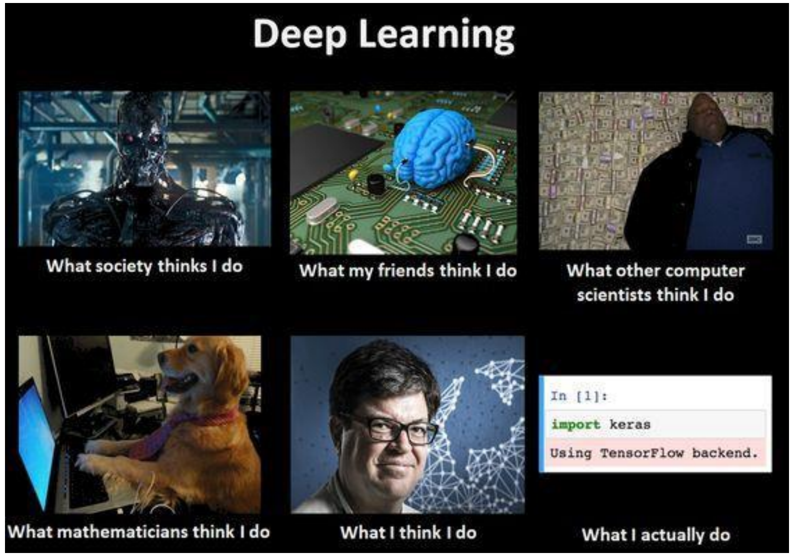
[Exemplo: aproximando_função_descontínua_com_MLPRegressor.ipynb]

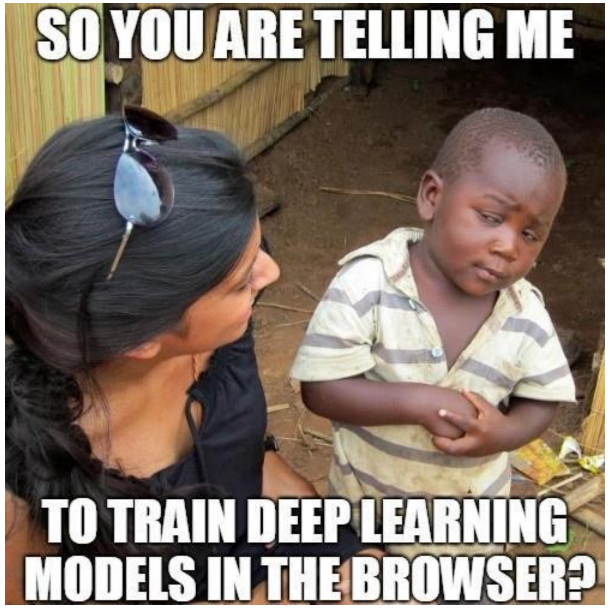
Avisos

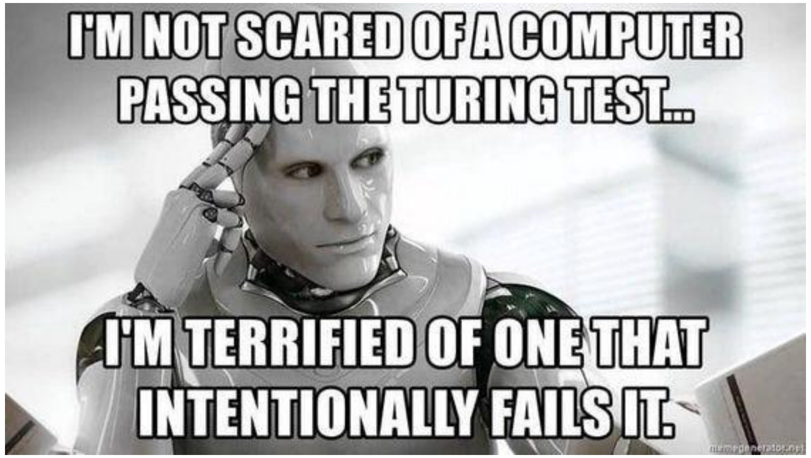
- Vocês já podem resolver os exercícios da lista #12.

Obrigado!

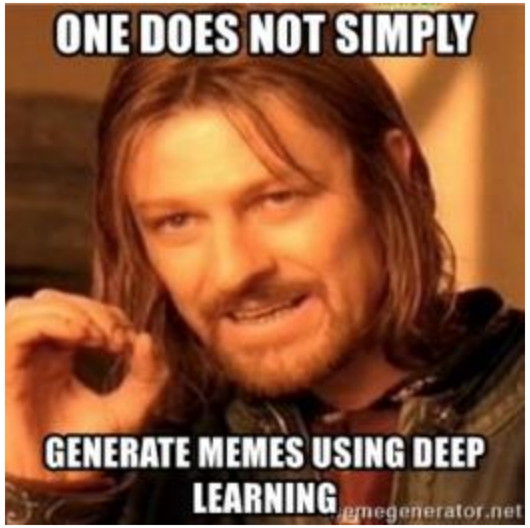

People with no idea about AI, telling me my AI will destroy the world
Me wondering why my neural network is classifying a cat as a dog..
Dog

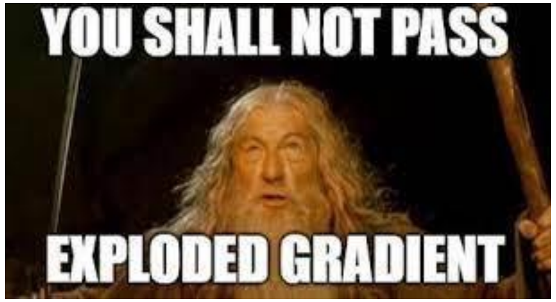

Deep Learning
What society thinks I do
What my friends think I do
What other computer scientists think I do
What mathematicians think I do
What I think I do
In [1]:
import keras
Using TensorFlow backend.
What I actually do


SO YOU ARE TELLING ME
TO TRAIN DEEP LEARNING MODELS IN THE BROWSER?

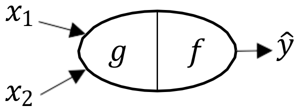

I'M NOT SCARED OF A COMPUTER PASSING THE TURING TEST...
I'M TERRIFIED OF ONE THAT INTENTIONALLY FAILS IT.

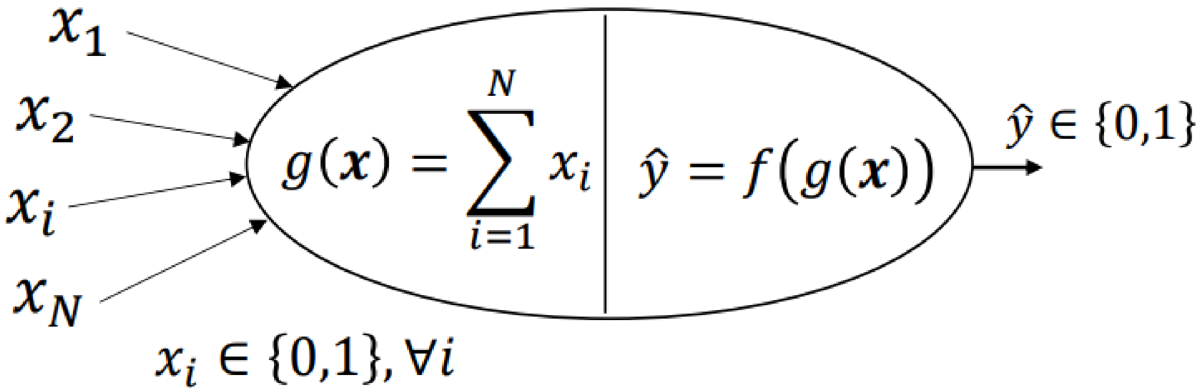

Dog
I NEED GPU
FOR MY DUMB
NEURAL NETWORK

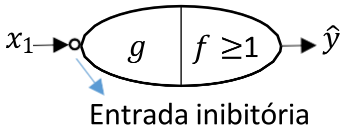

ONE DOES NOT SIMPLY
GENERATE MEMES USING DEEP LEARNING

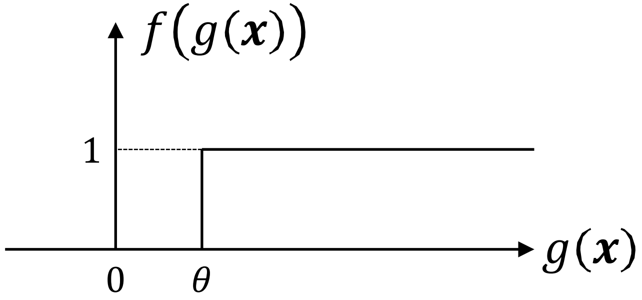

YOU SHALL NOT PASS
EXPLODED GRADIENT

Figuras



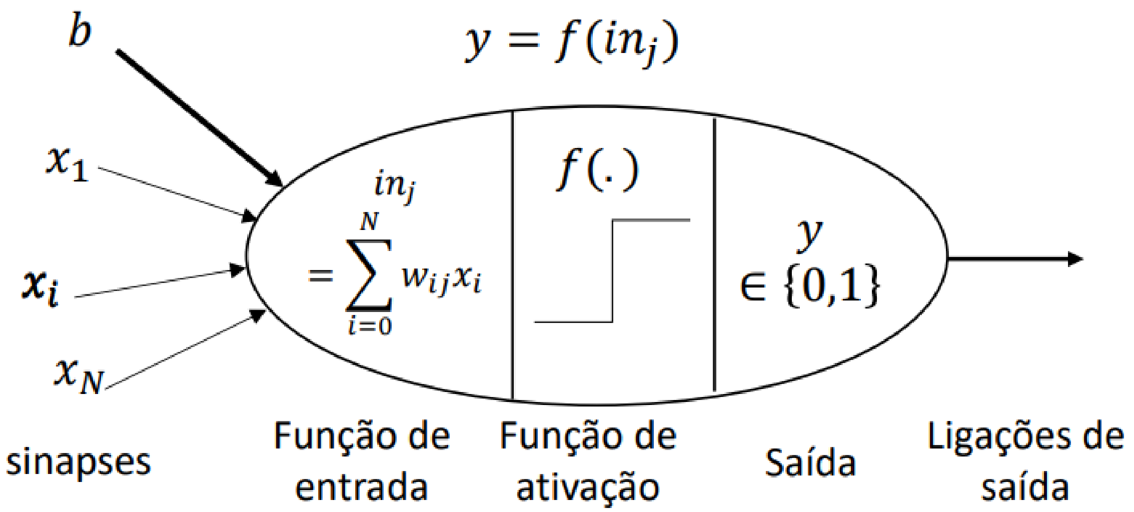




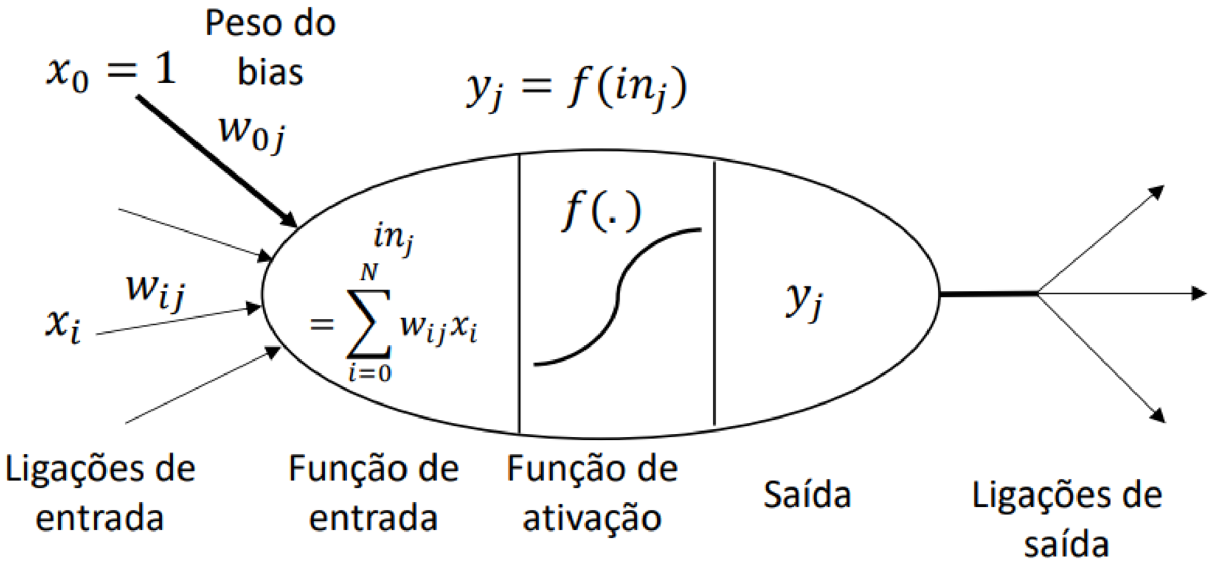


$$\hat{y} = f(g(\boldsymbol{x})) = \begin{cases} 1, & \text{se } g(\boldsymbol{x}) \geq \theta \\ 0, & \text{se } g(\boldsymbol{x}) < \theta \end{cases}$$

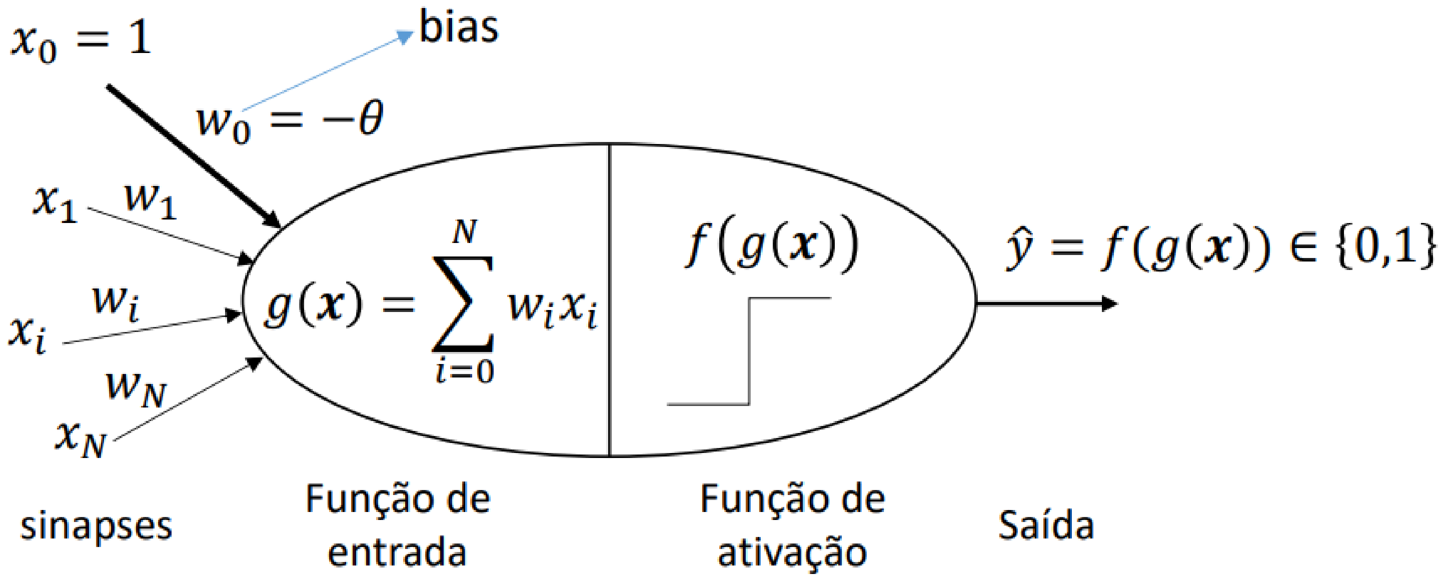
onde θ é o limiar de decisão.



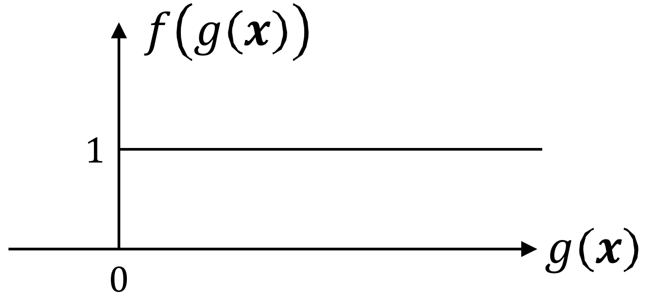
b
$y = f(in_j)$
x_1
x_i
x_N
$in_j = \sum_{i=0}^{N} w_{ij} x_i$
$f(.)$
$y \in \{0,1\}$
sinapses
Função de entrada
Função de ativação
Saída
Ligações de saída



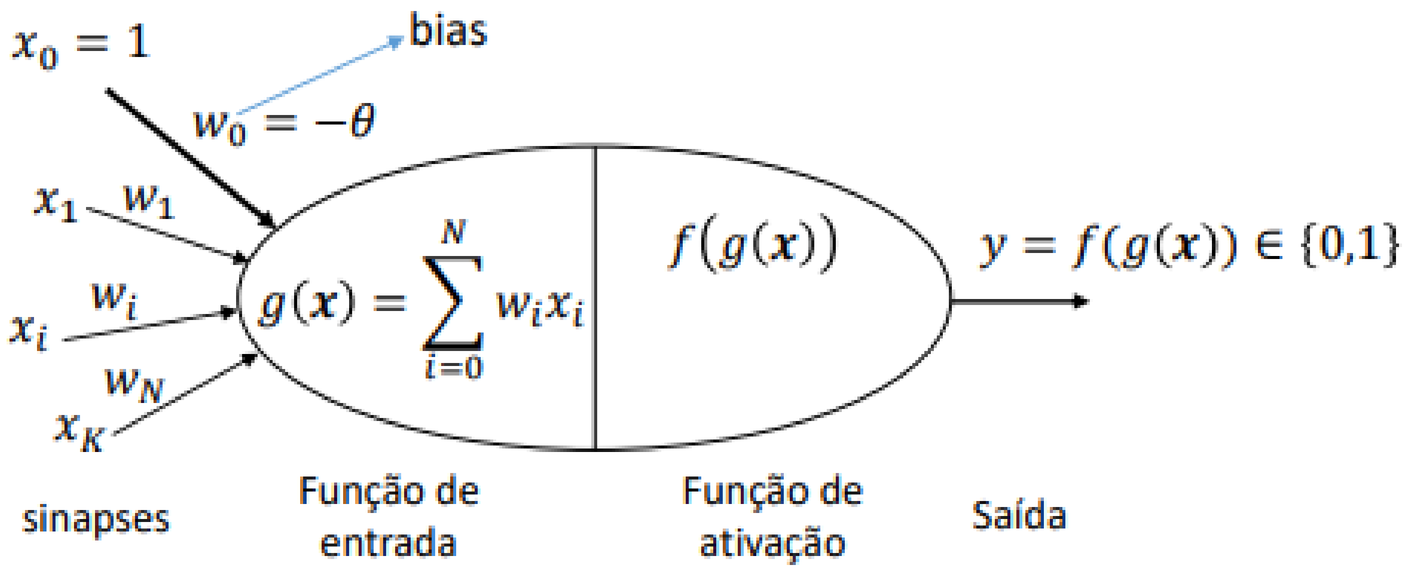
Peso do bias
$x_0 = 1$
w_{0j}
$y_j = f(in_j)$
x_i
w_{ij}
$in_j = \sum_{i=0}^{N} w_{ij} x_i$
$f(.)$
y_j
Ligações de entrada
Função de entrada
Função de ativação
Saída
Ligações de saída

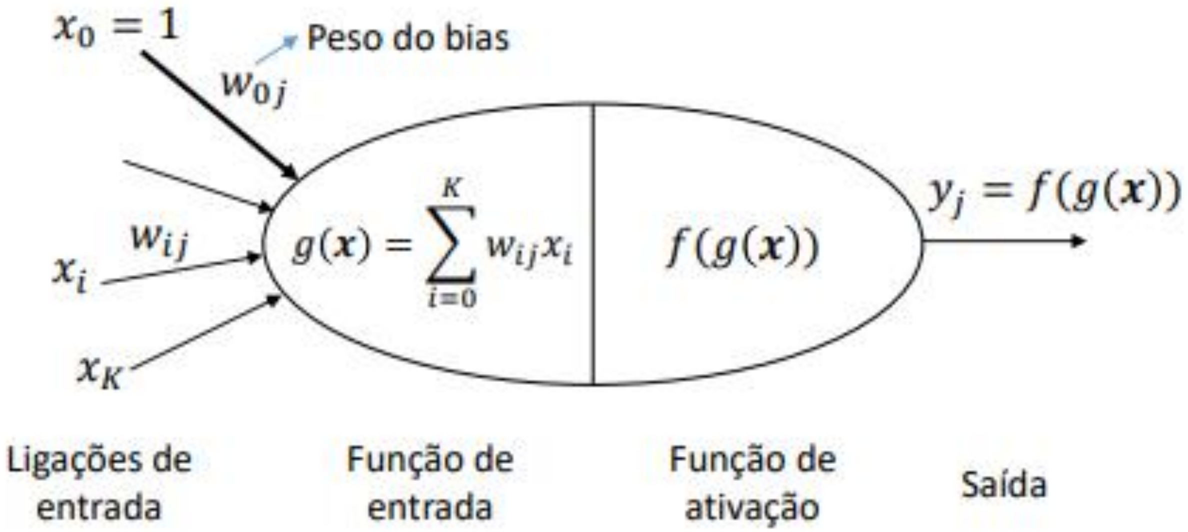


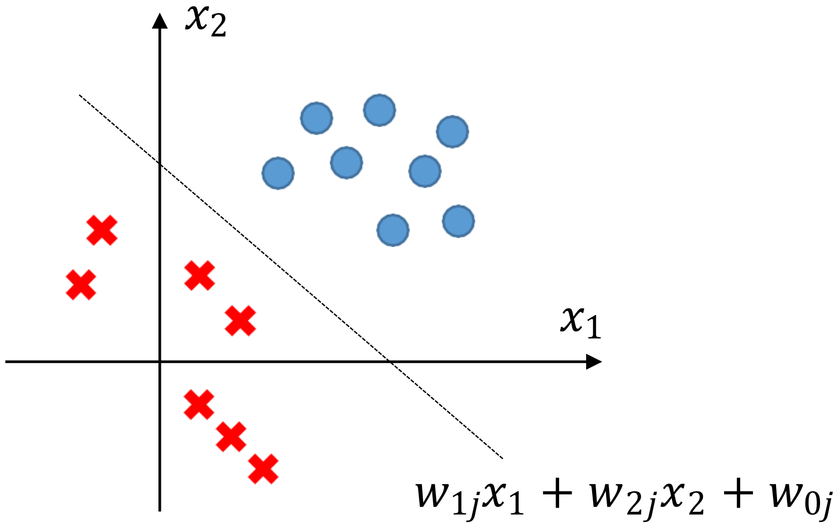
$x_0 = 1$
bias
$w_0 = -\theta$
x_1 w_1
x_i w_i
x_N w_N
$g(\boldsymbol{x}) = \sum_{i=0}^{N} w_i x_i$
$f(g(\boldsymbol{x}))$
$\hat{y} = f(g(\boldsymbol{x})) \in \{0,1\}$
sinapses
Função de entrada
Função de ativação
Saída

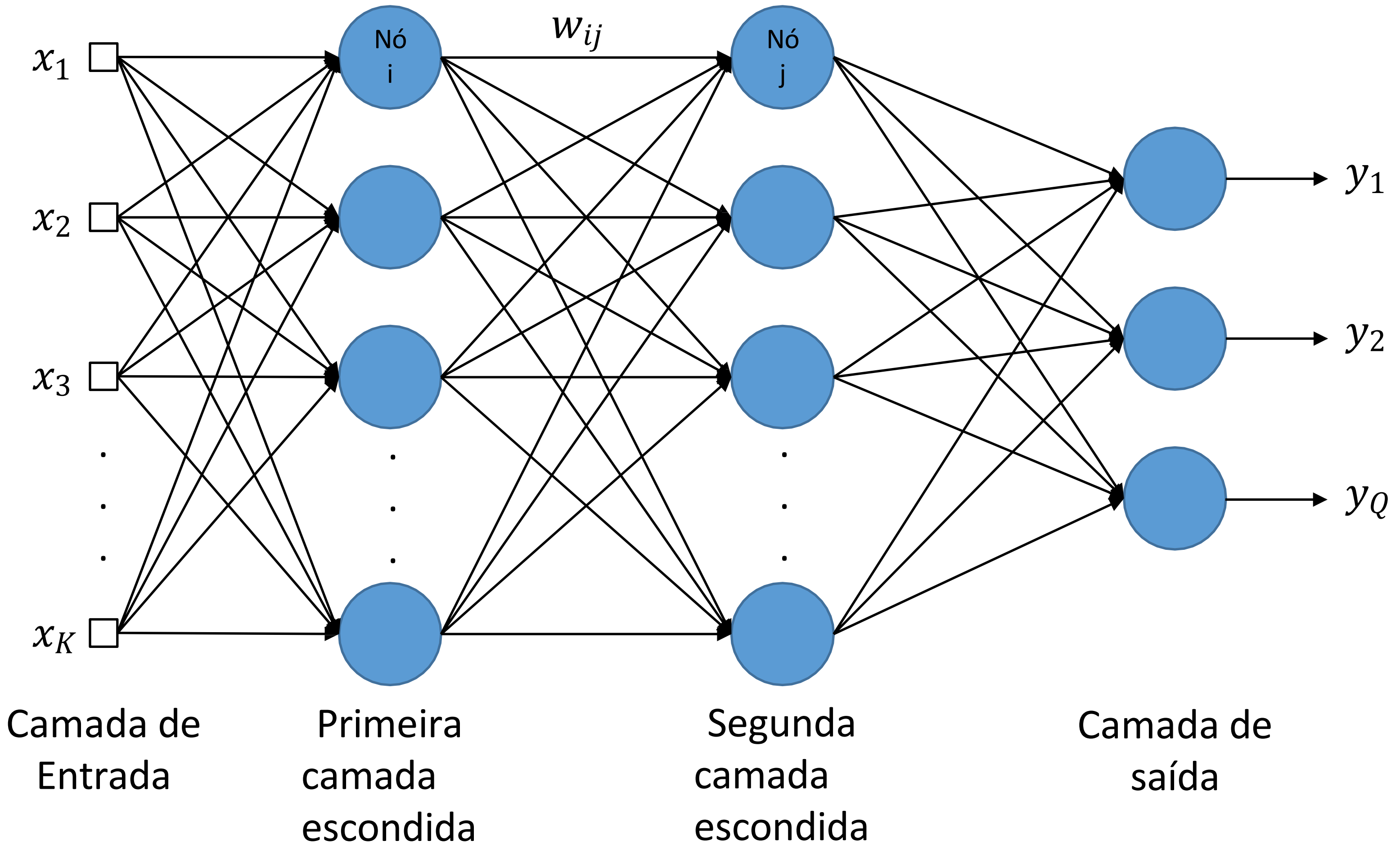


$f(g(\boldsymbol{x}))$
1
0
$g(\boldsymbol{x})$


$x_0 = 1$
bias
$w_0 = -\theta$
x_1 w_1
x_i w_i
x_K w_N
$g(\boldsymbol{x}) = \sum_{i=0}^{N} w_i x_i$
$f(g(\boldsymbol{x}))$
$y = f(g(\boldsymbol{x})) \in \{0,1\}$
sinapses
Função de entrada
Função de ativação
Saída


$x_0 = 1$
Peso do bias
w_{0j}
w_{ij}
x_i
x_K
$g(\boldsymbol{x}) = \sum_{i=0}^{K} w_{ij} x_i$
$f(g(\boldsymbol{x}))$
$y_j = f(g(\boldsymbol{x}))$
Ligações de entrada
Função de entrada
Função de ativação
Saída

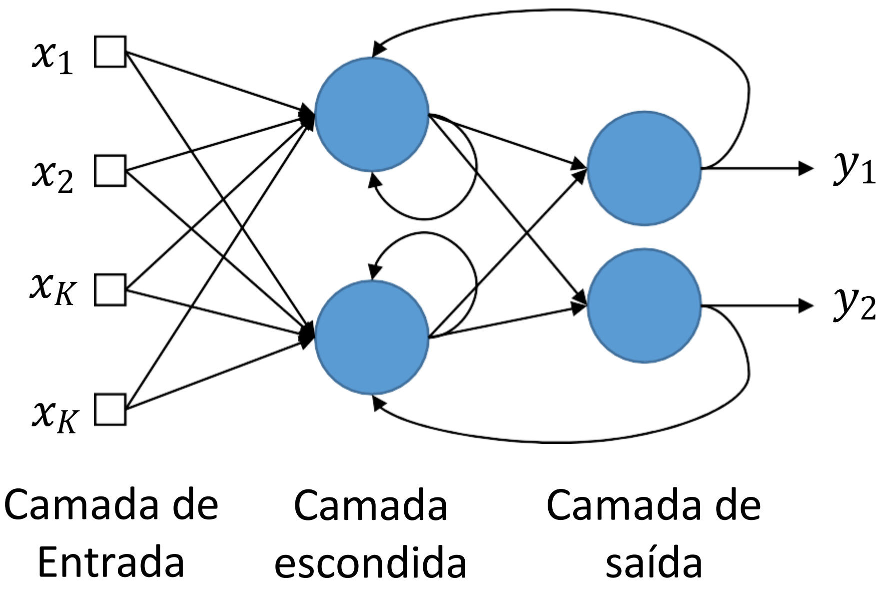

x_2
x_1
$w_{1j}x_1 + w_{2j}x_2 + w_{0j}$

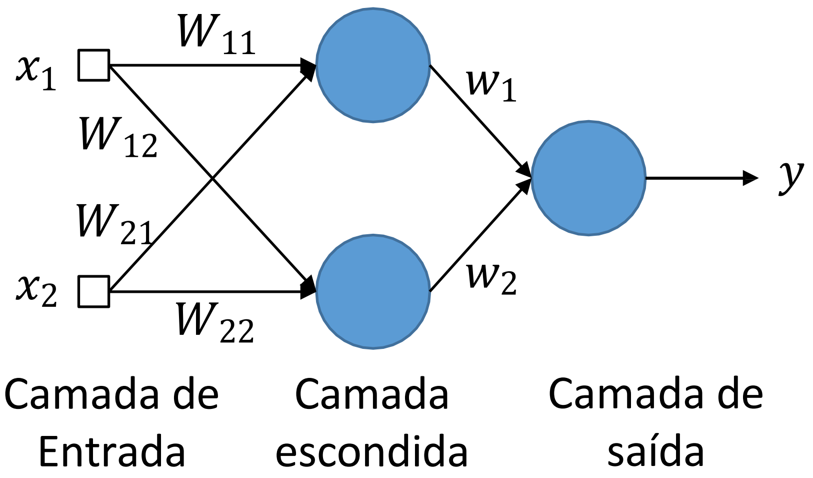


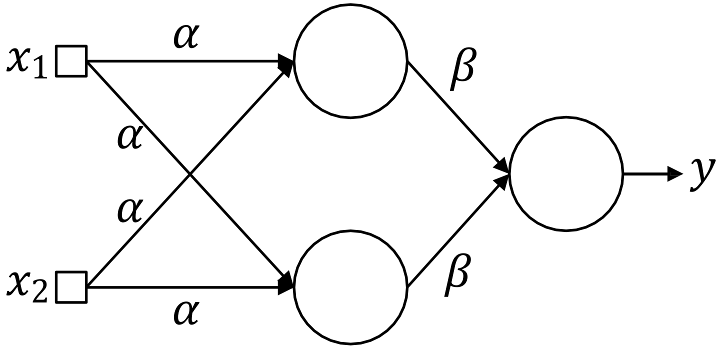
Nó, unidade ou neurônio.

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

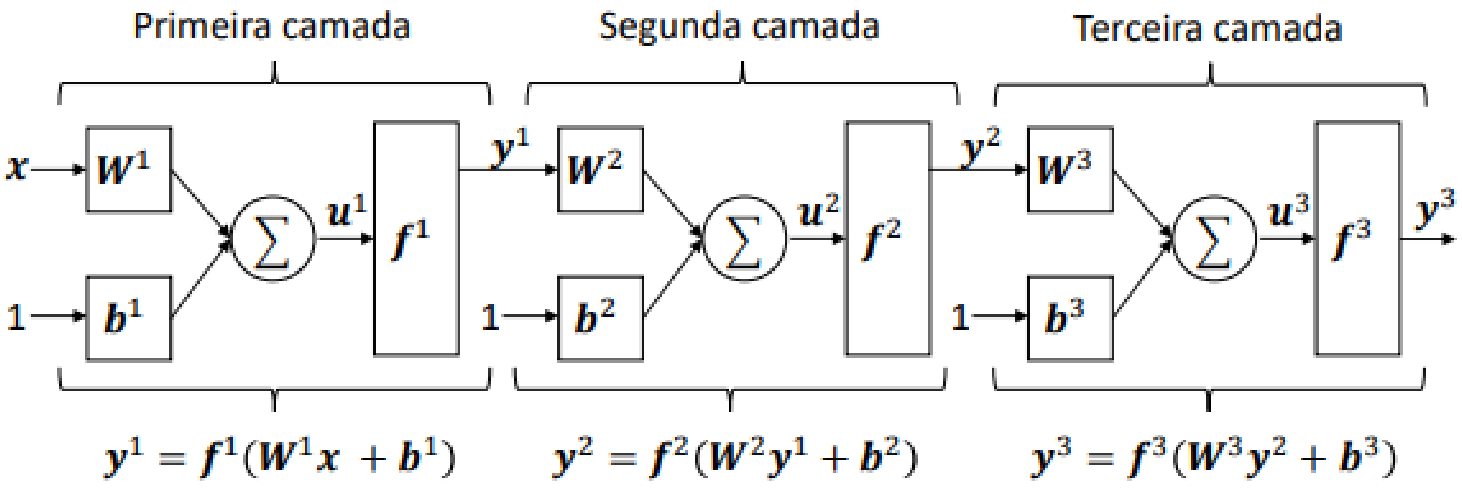
w_{ij} Peso da ligação entre *i*-ésimo e *j*-ésimo nó.

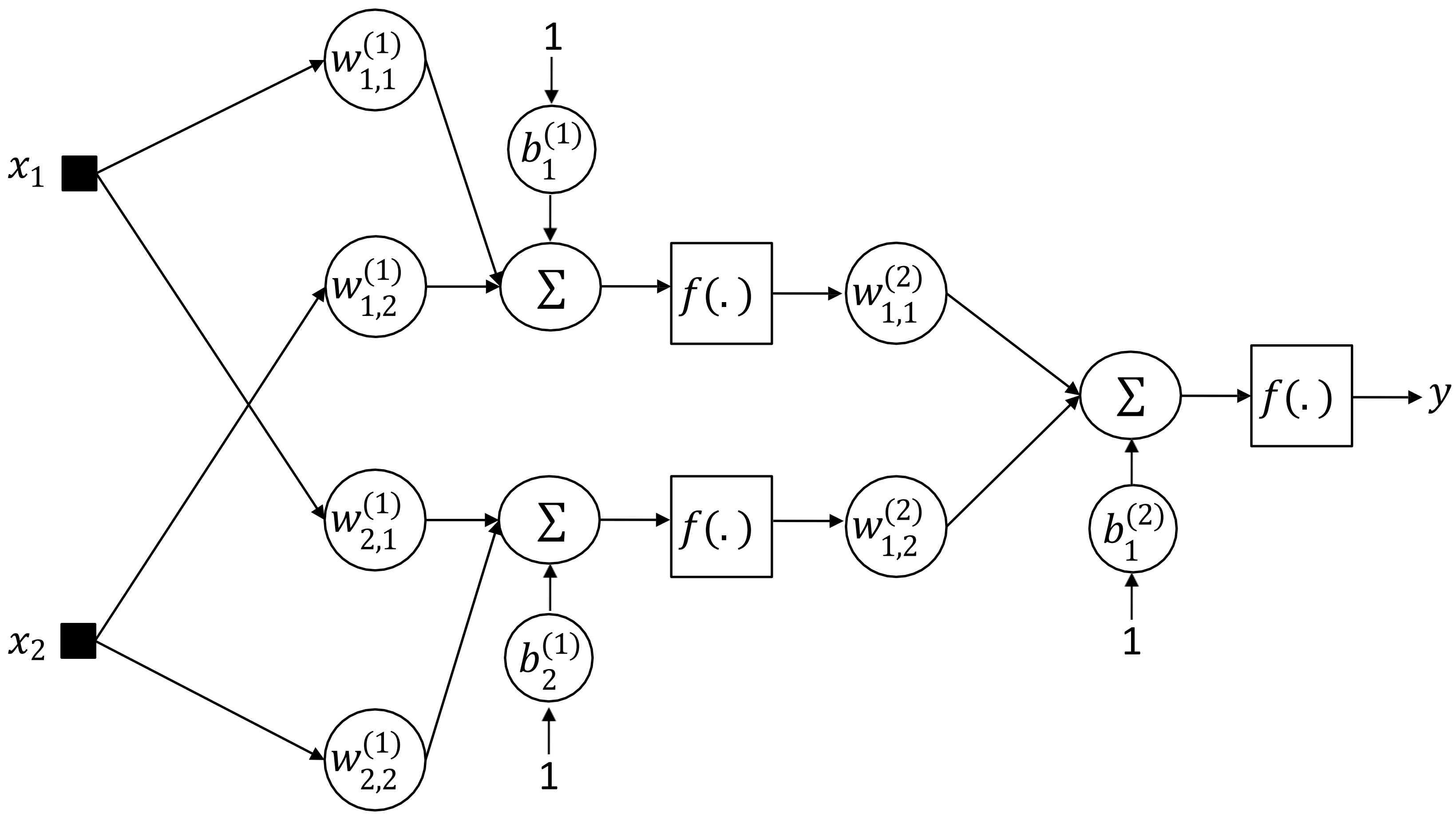

x_1
x_2
x_K
x_K
y_1
y_2
Camada de Entrada
Camada escondida
Camada de saída


x_1
x_2
W_{11}
W_{12}
W_{21}
W_{22}
w_1
w_2
y
Camada de Entrada
Camada escondida
Camada de saída



x_1
x_2
α
α
α
α
β
β
y


Primeira camada
Segunda camada
Terceira camada
x
W^1
$\sum$
u^1
f^1
y^1
1
b^1
W^2
$\sum$
u^2
f^2
y^2
1
b^2
W^3
$\sum$
u^3
f^3
y^3
1
b^3
$\boldsymbol{y}^1 = \boldsymbol{f}^1(\boldsymbol{W}^1\boldsymbol{x} + \boldsymbol{b}^1)$
$\boldsymbol{y}^2 = \boldsymbol{f}^2(\boldsymbol{W}^2\boldsymbol{y}^1 + \boldsymbol{b}^2)$
$\boldsymbol{y}^3 = \boldsymbol{f}^3(\boldsymbol{W}^3\boldsymbol{y}^2 + \boldsymbol{b}^3)$



x_1
x_2
$w_{1,1}^{(1)}$
$w_{1,2}^{(1)}$
$w_{2,1}^{(1)}$
$w_{2,2}^{(1)}$
1
$b_1^{(1)}$
$b_2^{(1)}$
1
Σ
Σ
$f(.)$
$f(.)$
$w_{1,1}^{(2)}$
$w_{1,2}^{(2)}$
Σ
$b_1^{(2)}$
1
$f(.)$
y