Redes Neurais Artificiais

1. Inspiração biológica e conceitos iniciais

A descoberta da célula, por Robert Hooke, em 1665, foi um passo de enorme importância para que houvesse uma melhor compreensão da estrutura dos seres vivos. Talvez se possa considerar, *cum grano salis*, a célula como um "átomo de vida".

As células eucariontes possuem três partes principais: membrana, núcleo e citoplasma. A membrana "delimita a célula", *i.e.*, isola seu interior do meio externo. O núcleo abriga o material genético e, no citoplasma, estão componentes como as organelas.

Os **neurônios** são células especializadas que possuem mecanismos elétricos e/ou químicos peculiares. A Figura 1 traz uma visão esquemática.

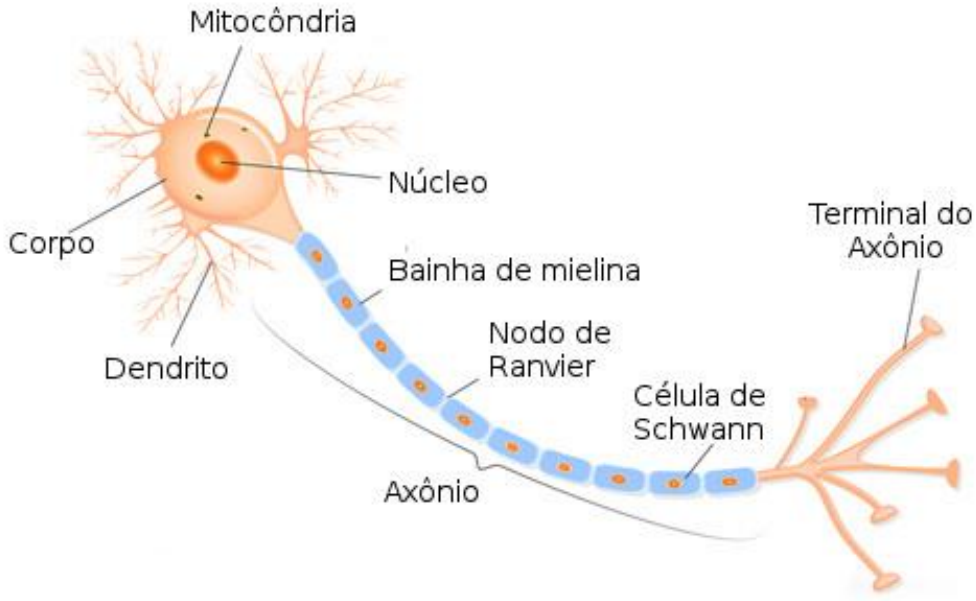


Figura 1 – Visão básica de um neurônio. Extraída de https://medium.com/@avinicius.adorno/redes-neurais-artificiais-5b65a43614a0.

Em termos muito simples, podemos afirmar (cientes de que há exceções):

- Que o neurônio recebe estímulos elétricos, basicamente a partir dos dendritos;
- Que esses estímulos são integrados;

- Que a estimulação pode levar à geração de uma resposta elétrica enviada pelo axônio: se a atividade combinada exceder certo limiar, o neurônio gera um pulso (*spike* ou potencial de ação), conforme ilustrado na Figura 2.

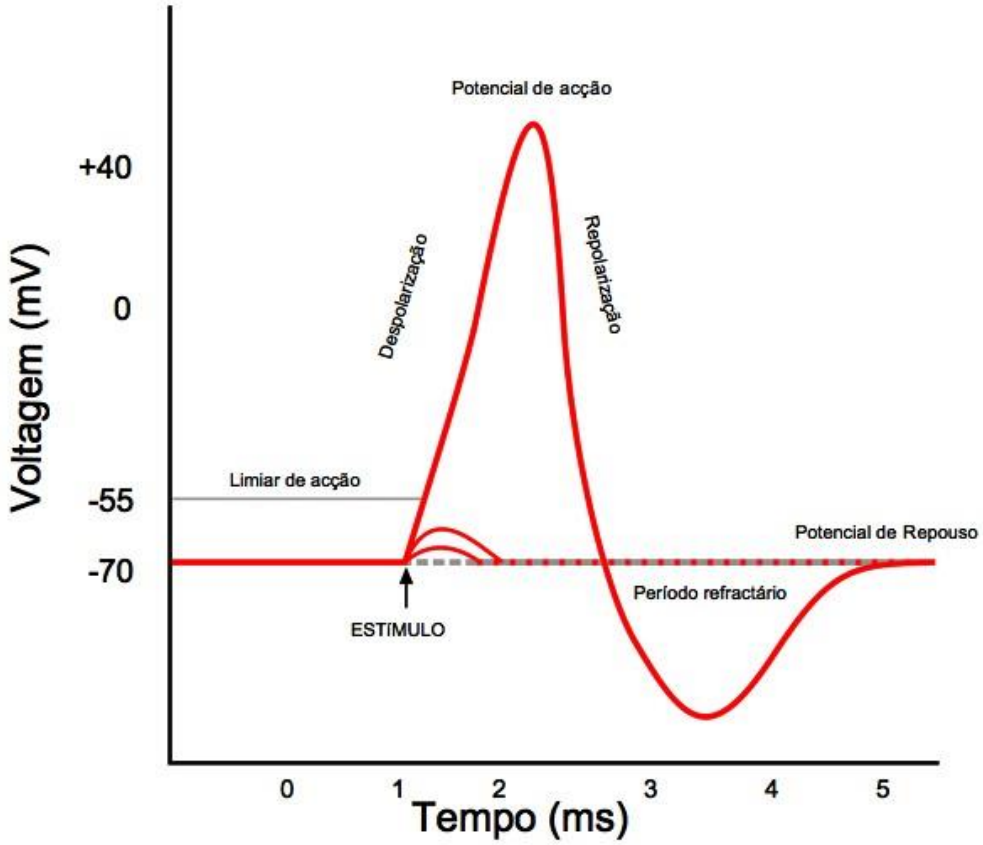


Figura 2 – Ilustração do potencial de ação.
Fonte: https://wikiciencias.casadasciencias.org/wiki/index.php/Potencial_de_Ac%C3%A7%C3%A3o.

O cérebro humano apresenta um número muito elevado – da ordem de bilhões – de neurônios massivamente interconectados.

A transmissão de informação (potenciais de ação) entre dois neurônios se dá com base em um processo eletroquímico bastante sofisticado, no qual entram em cena diversos elementos, como os canais iônicos, as bombas de sódio e potássio e os neurotransmissores.

Conforme ilustrado na Figura 3, os terminais do axônio de um neurônio se "conectam" aos dendritos ou ao corpo celular de outro neurônio através das **fendas sinápticas**.

A chamada **plasticidade sináptica**, que corresponde à capacidade de as sinapses sofrerem modificações ao longo do tempo, é o ingrediente-chave para diversos processos cognitivos, como percepção, raciocínio, memória e aprendizado.

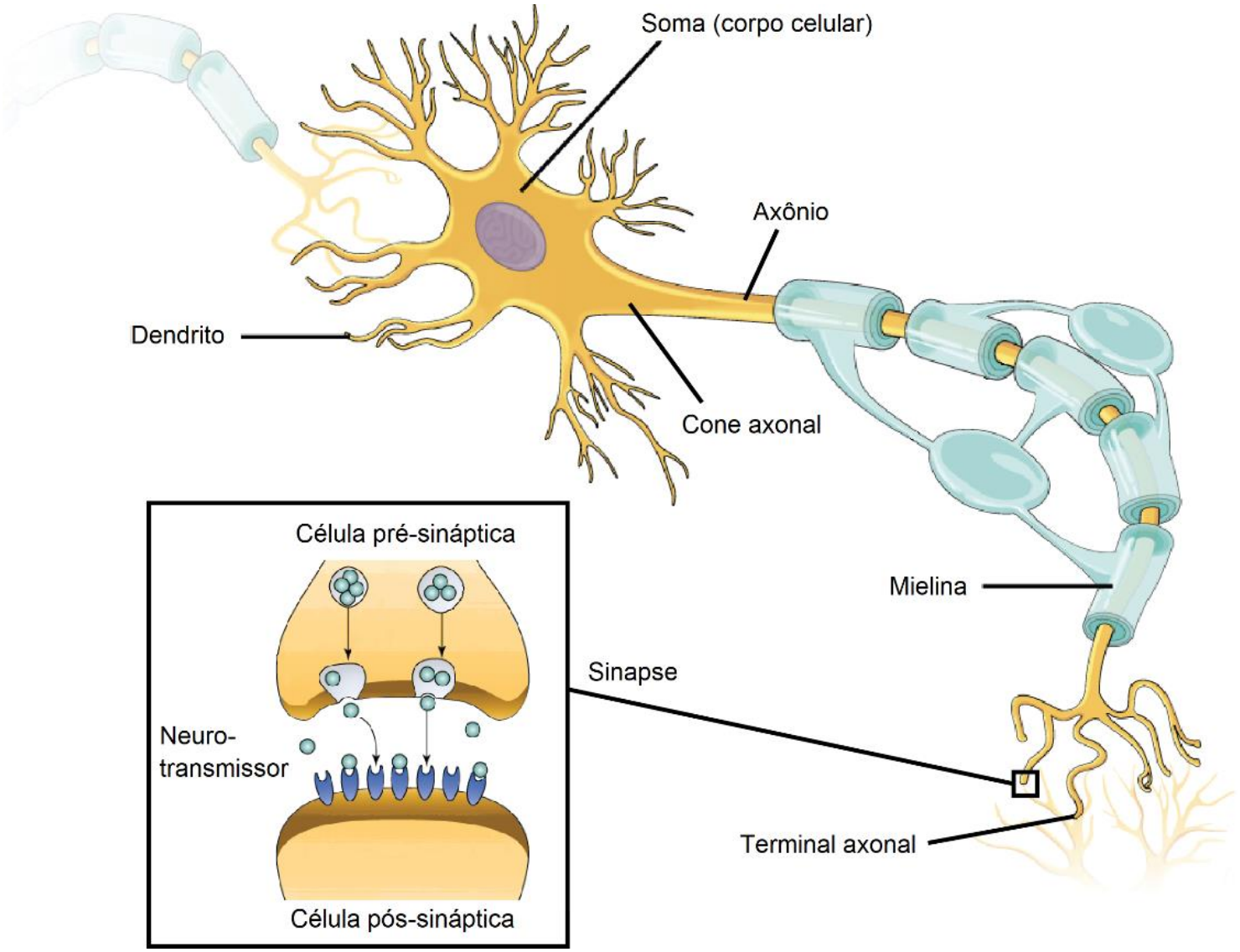


Figura 3 – Visão geral da conexão entre dois neurônios através de uma sinapse. Extraída de https://pt.khanacademy.org/science/biology/human-biology/neuron-nervous-system/a/overview-of-neuron-structure-and-function.

Do ponto de vista de nosso curso, o neurônio será interpretado como um sistema que possui entradas e gera uma saída. Deste modo, o neurônio (tanto o biológico quanto o artificial) representa uma unidade que **processa informação**.

Uma **rede neural artificial** (ANN, do inglês *artificial neural network*) é uma estrutura composta de múltiplas unidades de processamento, denominadas neurônios artificiais, interligadas segundo um padrão de conexões.

2. Modelo de neurônio

A Figura 4 exibe um dos modelos de neurônio artificial mais utilizados, o qual é conhecido como *perceptron* (ROSENBLATT, 1956).

Em essência, os sinais de entrada $x_i, i = 1, \ldots, m$, junto com um sinal de polarização (*bias*), são ponderados linearmente pelos pesos sinápticos w_{kj}, em que k representa o índice do neurônio e j indica o sinal de entrada ao qual o peso está associado, e

somados, gerando v_k, o qual passa por uma função de ativação $\varphi(\cdot)$, usualmente de caráter não-linear, produzindo a saída y_k.

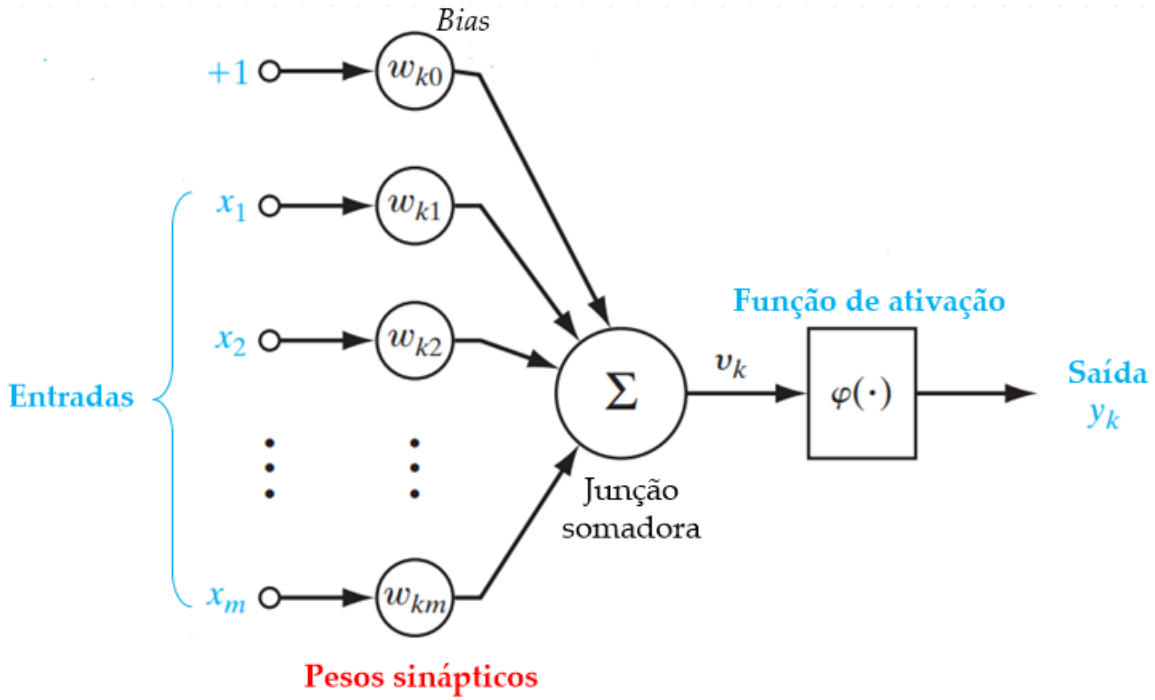


Figura 4 – Modelo de neurônio artificial (*perceptron*). Adaptada de (HAYKIN, 2008).

Em termos matemáticos, a saída do neurônio pode ser expressa da seguinte forma:

$$y_k = \varphi(v_k) = \varphi\left(\sum_{j=1}^{m} w_{kj}x_j + w_{k0}\right) = \varphi\left(\sum_{j=0}^{m} w_{kj}x_j\right), \qquad (1)$$

ou, explorando uma notação vetorial,

$$y_k = \varphi(\mathbf{w}_k{}^T \mathbf{x}), \tag{2}$$

onde $\mathbf{w}_k = [w_{k0} \ \cdots \ w_{km}]^T$ e $\mathbf{x} = [1 \ \cdots \ x_m]^T$.

Observações:

- O primeiro modelo algébrico de um neurônio artificial, proposto por McCulloch e Pitts (1943), é um caso particular de (1), no qual as entradas são binárias e a função de ativação é do tipo degrau. Além disso, o modelo admitia a presença de sinapses inibitórias.
- O *perceptron* de Rosenblatt (1956) também segue o modelo em (1) para sinais reais, mas adota a função de ativação do tipo degrau. Neste caso,

$$y_k = \varphi(v_k) = \varphi\left(\sum_{j=1}^{m} w_{kj}x_j + w_{k0}\right) = \begin{cases} 1, \text{se} \sum_{j=1}^{m} w_{kj}x_j + w_{k0} \geq 0 \\ 0, \text{se} \sum_{j=1}^{m} w_{kj}x_j + w_{k0} < 0 \end{cases}.$$

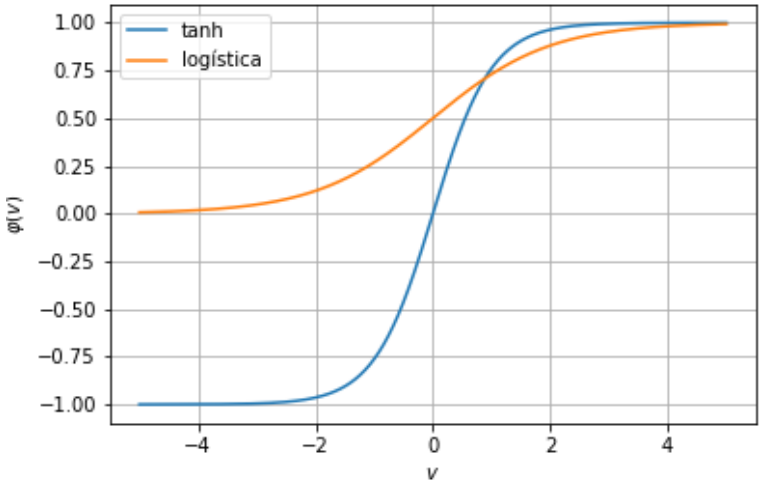
Deste modo, o *perceptron* atua como um discriminante linear, sendo capaz de classificar corretamente apenas padrões que podem ser separados por um hiperplano no espaço de atributos.

2.1. Funções de ativação

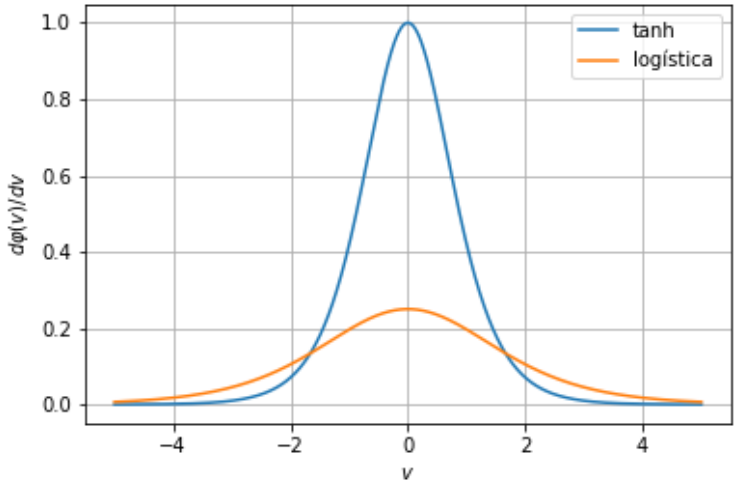
A escolha da função de ativação dos neurônios tem influência sobre alguns aspectos da rede neural, tais como sua flexibilidade para aproximar mapeamentos entrada-saída complexos, bem como a eficiência de seu treinamento.

Funções sigmoidais, como a tangente hiperbólica, foram, por muito tempo, o padrão em ANNs.

- Tangente hiperbólica: $\varphi(v_k) = \tanh(pv_k)$ e $\frac{d\varphi(v_k)}{dv_k} = p(1 - \varphi(v_k)^2)$.
- Função logística: $\varphi(v_k) = \frac{1}{1+e^{-pv_k}}$ e $\frac{d\varphi(v_k)}{dv_k} = p\varphi(v_k)\big(1 - \varphi(v_k)\big)$.
- Em ambos os casos, a função é diferenciável em todos os pontos e a saída do neurônio apresenta saturação, conforme mostra a Figura 5.



(a)



(b)

Figura 5 – Funções de ativação sigmoidais (a) e suas derivadas (b), com $p = 1$.

Com o advento das técnicas de *deep learning*, funções lineares por partes ganharam espaço por apresentarem uma melhor relação custo-benefício entre eficiência no ajuste de pesos e desempenho de aproximação. A função retificadora, que dá origem ao famoso modelo ReLU (*Rectified Linear Unit*), é definida como:

$$y_k = \max(0, v_k) \tag{3}$$

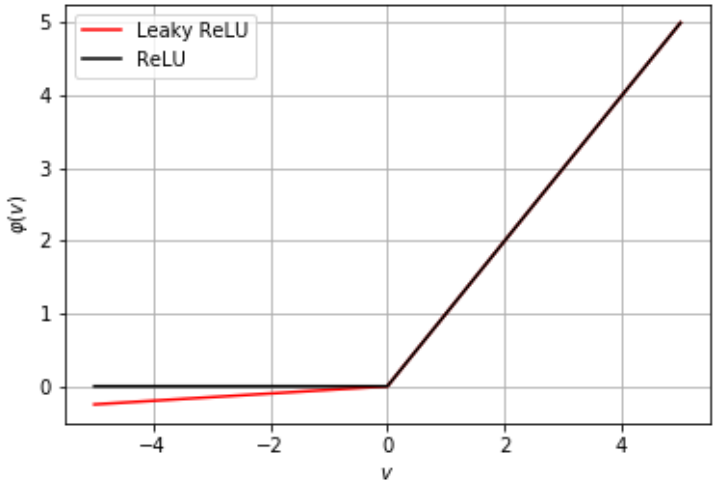
Embora a ReLU seja muito utilizada em modelos bem-sucedidos de ANNs, inclusive em redes profundas, ela apresenta uma potencial desvantagem: caso muitos neurônios apresentem valores negativos como resposta a vários estímulos de entrada, eles praticamente deixam de participar do processamento da rede e não sofrem ajustes em seus parâmetros, pois a derivada da função de ativação é nula.

Uma alternativa sugerida na literatura é a função *Leaky* ReLU, definida como:

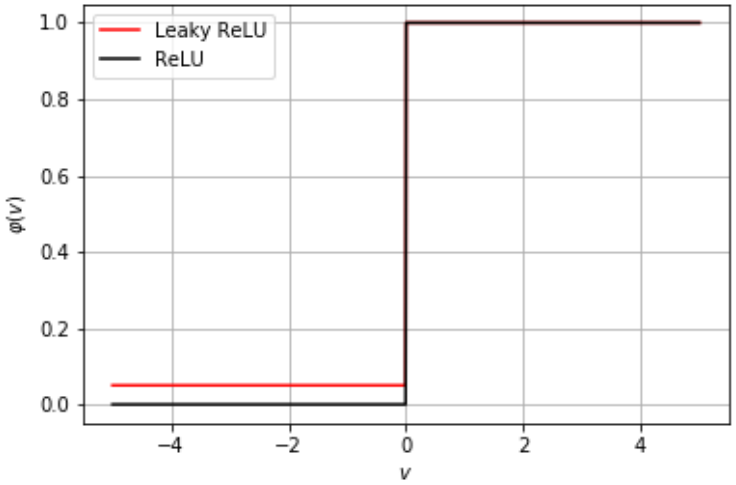
$$y_k = \begin{cases} v_k, & \text{se } v_k \geq 0 \\ \alpha v_k, & \text{se } v_k < 0' \end{cases} \tag{4}$$

onde α é um valor não-negativo próximo de zero. A Figura 6 ilustra estas duas funções de ativação, juntamente com as respectivas derivadas.

Outras variantes como PReLU, ELU e SELU também têm sido consideradas (PEDAMONTI, 2018; GÉRON, 2019).



(a)



(b)

Figura 6 – Funções de ativação lineares por partes (a) e suas derivadas (b): ReLU e *Leaky* ReLU ($\alpha = 0{,}05$).

2.2. Arquiteturas

O tipo e a quantidade de neurônios artificiais presentes em uma rede, junto com o padrão de conectividade estabelecido entre os neurônios, definem a **arquitetura** da rede neural. De forma simples, é possível dividir as diferentes arquiteturas em dois grupos básicos:

- **Redes *feedforward*** (FNNs, do inglês *feedforward neural networks*): os sinais recebidos pela rede são propagados em um único sentido até que as saídas sejam determinadas. Tipicamente, as FNNs estão organizadas em camadas, sendo que cada camada possui uma quantidade de neurônios própria e realiza um mapeamento entrada-saída. Neste caso, podemos resumir o comportamento de uma rede *feedforward* à seguinte expressão:

$$\mathbf{y}(n) = \Phi(\mathbf{x}(n); \boldsymbol{\theta}) \tag{5}$$

  Ou seja, uma FNN implementa um mapeamento da entrada $\mathbf{x}$(n) na saída $\mathbf{y}(n)$, sendo $\boldsymbol{\theta}$ o conjunto de parâmetros da rede. Para um $\boldsymbol{\theta}$ fixo, sempre que a entrada for $\mathbf{x}(n)$, a saída observada será $\mathbf{y}(n)$. Neste sentido, dizemos que o mapeamento é estático.

- **Redes recorrentes** (RNNs, do inglês *recurrent neural networks*): existem laços de realimentação que transmitem as saídas de neurônios de uma determinada

camada para neurônios pertencentes à mesma camada ou a camadas anteriores. Por conta disto, o mapeamento entrada-saída de uma RNN depende não apenas da entrada atual, mas também de uma memória interna que reflete o contexto do processamento. Assim, uma RNN configura um sistema dinâmico. Por exemplo, uma RNN poderia ter a seguinte relação entrada-saída:

$$\mathbf{y}(n) = \Phi(\mathbf{x}(n), \mathbf{y}(n-1); \boldsymbol{\theta}) \tag{6}$$

3. *Perceptron* de múltiplas camadas (MLP)

Um dos principais expoentes da classe de FNNs é a rede conhecida como *perceptron* de múltiplas camadas (MLP, do inglês *multilayer perceptron*).

Características:

- Existe um número arbitrário de camadas ocultas, também chamadas de camadas escondidas ou intermediárias, entre a entrada e a saída da rede.

- Os neurônios de uma camada l estão conectados a todos os neurônios da camada seguinte $(l+1)$. Por isso, essa estrutura é também chamada de *totalmente conectada* (ou *densa*).

A Figura 7 exibe a arquitetura geral de uma MLP.

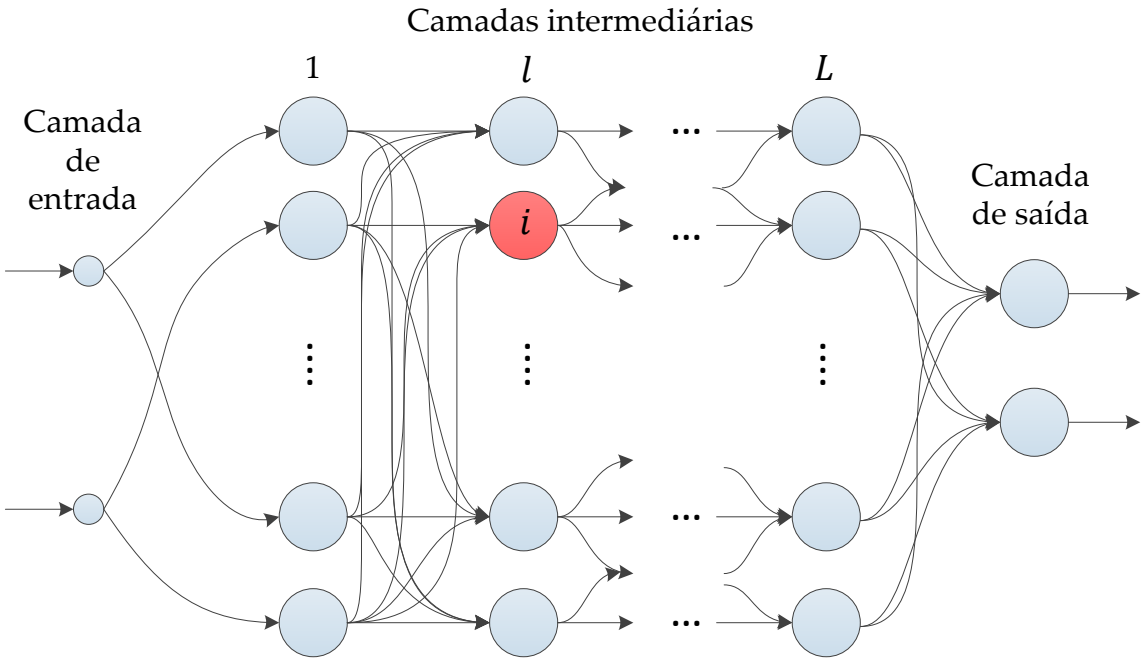


Figura 7 – Arquitetura de uma rede MLP.

A camada de entrada nada mais é que a ilustração da passagem 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. Por fim, os neurônios da camada de saída combinam a informação que lhes é oferecida pela última camada intermediária, produzindo as respostas da rede para aquele padrão de entrada.

Considere o i-ésimo neurônio da l-ésima camada intermediária, como destacado na Figura 7, em que $i = 1, \dots, n_l$ e $l = 1, \dots, L$. Sua saída ${y_i}^l$ é obtida da seguinte forma:

$$ {y_i}^l = \varphi^l \left(\sum_{j=1}^{n_{l-1}} {w_{ij}}^l {y_j}^{l-1} + {w_{i0}}^l \right), \tag{7} $$

onde ${w_{ij}}^l$ representa o peso sináptico da conexão que liga o j-ésimo neurônio da camada $l-1$ ao i-ésimo neurônio da camada l. Para a primeira camada

intermediária, os sinais de entrada são os próprios atributos do dado, *i.e.*, $y_j{}^0 = x_j, j = 1, \dots, m$.

Percebemos que a função de ativação $\varphi(\cdot)$ determina o "perfil" do mapeamento realizado pelo neurônio. Por sua vez, os pesos sinápticos $w_{ij}{}^l$ determinam a inclinação, a orientação e a posição do mapeamento gerado. A Figura 8 ilustra isso.

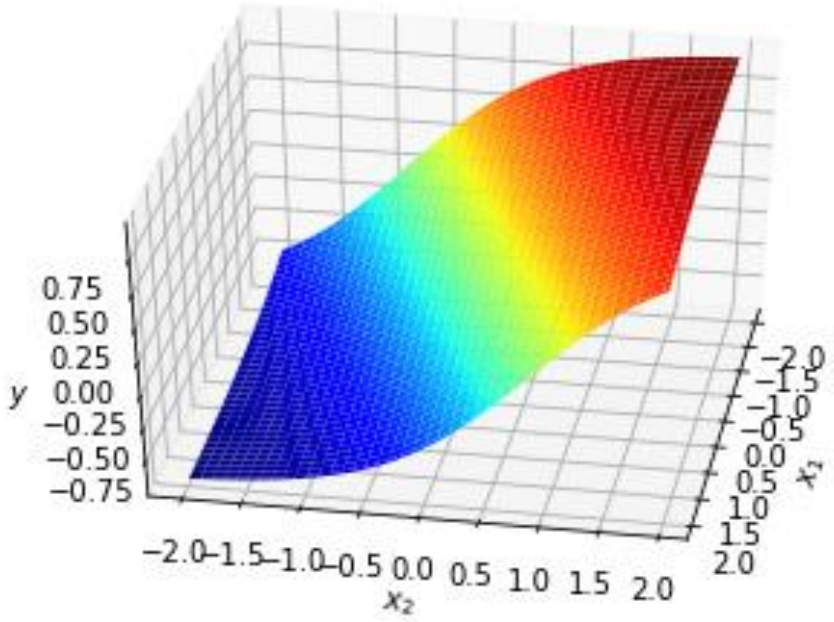


Figura 8 – Mapeamento gerado por um neurônio do tipo *perceptron* com $\varphi(\cdot) = \tanh(\cdot)$.

Observação: funções desse tipo são chamadas de funções de expansão ortogonal (*ridge functions*).

A estrutura a ser usada na camada de saída de uma MLP depende do problema abordado. Em regressão, os neurônios de saída usualmente têm função de ativação igual à identidade, de modo que cada saída da rede é obtida por meio de uma combinação linear das ativações dos neurônios da última camada oculta. Em classificação binária ou multi-rótulo, recorre-se à função logística, enquanto a *softmax* é utilizada no cenário multi-classe.

3.1. Capacidade de aproximação universal

As redes MLP possuem o apelo da inspiração biológica, mas não seriam amplamente utilizadas se não tivessem significativo potencial de aproximar mapeamentos entrada-saída.

Com efeito, essas redes possuem *capacidade de aproximação universal*, ou seja, são capazes de aproximar qualquer mapeamento contínuo num domínio compacto com um nível de erro arbitrariamente pequeno. Interessantemente, mesmo uma rede MLP com uma única camada intermediária (com função de ativação tangente hiperbólica, por exemplo) e camada de saída linear já possui essa capacidade (CYBENKO, 1989; HORNIK ET AL., 1989; HORNIK ET AL., 1990).

Seja $\varphi(\cdot)$ uma função contínua, não-constante, limitada e monotonicamente crescente (tanto a tangente hiperbólica quanto a função logística são possibilidades). Consideremos que a camada intermediária da rede seja composta de n neurônios do tipo *perceptron* com essa função de ativação. Imaginemos que se deseje aproximar uma função contínua $g(x_1, x_2, \ldots, x_m)$ que tenha por domínio o hipercubo unitário, m-dimensional $\mathbf{I}_m = (0,1)^m$. Se denominarmos o mapeamento realizado pela rede como $\Phi(x_1, x_2, \ldots, x_m)$, o resultado fundamental é que haverá um valor mínimo n e

valores de pesos sinápticos tais que $|\Phi(x_1, x_2, \ldots, x_m) - g(x_1, x_2, \ldots, x_m)| < \varepsilon$, para qualquer $\varepsilon > 0$.

3.2. Processo de treinamento

O treinamento de uma rede MLP consiste no processo de ajuste dos pesos sinápticos ${w_{ij}}^l$ de todas as camadas, em que buscamos os valores que levem ao melhor mapeamento entrada-saída possível para a rede.

Isto dá origem a um problema de otimização não-linear irrestrito, no qual, sem perda de generalidade, desejamos minimizar uma função custo $J(\mathbf{w})$ que expressa uma medida de erro entre as saídas fornecidas pela rede e as saídas desejadas, onde $\mathbf{w}$ representa o vetor com todos os parâmetros da rede.

Problema: $\min_{\mathbf{w}} J(\mathbf{w})$

Geralmente, esse processo de otimização é conduzido de maneira iterativa, o que dá sentido mais natural à noção de aprendizado (como um processo gradual). Há

vários métodos de otimização aplicáveis, mas, sem dúvida, os mais usuais são aqueles baseados nas derivadas da função custo.

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

$$\nabla J(\mathbf{w}) = \left[\frac{\partial J(\mathbf{w})}{\partial w_1} \cdots \frac{\partial J(\mathbf{w})}{\partial w_n}\right]^T.$$

O gradiente aponta na direção de maior crescimento da função: caminhar em direção contrária a ele é uma forma adequada de buscar iterativamente a minimização. Destarte, temos a seguinte forma básica para a regra de atualização dos pesos:

$$\mathbf{w}(k+1) \leftarrow \mathbf{w}(k) - \eta \nabla J[\mathbf{w}(k)]$$

sendo η o passo de adaptação (*learning rate*) e $\nabla(\cdot)$ o operador gradiente.

Existe uma literatura bastante vasta de métodos de otimização não-linear (LUENBERGER, 2003). Com efeito, veremos mais adiante algumas técnicas recentemente exploradas em *deep learning* que tentam acelerar a convergência em relação ao método de gradiente descendente.

Um ponto importante é que todos estes métodos são *métodos de busca local*, ou seja, têm convergência esperada para *mínimos locais*. Para recordarmos o que é um mínimo local, vejamos a Figura 9. Nela, há dois mínimos:

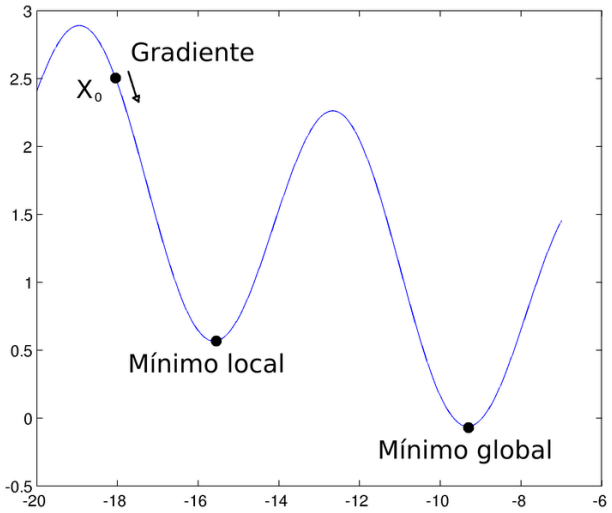


Figura 9 – Mínimo local e mínimo global.

- Um deles é uma solução ótima em relação a seus vizinhos, ou seja, um *mínimo local*.
- O outro também é uma solução ótima em relação a seus vizinhos, mas é, ademais, a solução ótima em relação a todo o domínio considerado. É, dessa forma, um *mínimo global*.

Para valores adequados do passo de adaptação η, um mínimo local tende a atrair o vetor de parâmetros quando este se encontra em sua vizinhança. De maneira mais rigorosa, dizemos que cada mínimo tem sua *bacia de atração*.

Devemos ainda mencionar os chamados *pontos de sela*, que são pontos que, em algumas direções são atratores, mas em outras não. Embora, a longo prazo, o algoritmo não vá convergir para esses pontos, ele pode passar um longo período sendo atraído por eles, o que prejudica seu desempenho. A Figura 10 mostra um exemplo.

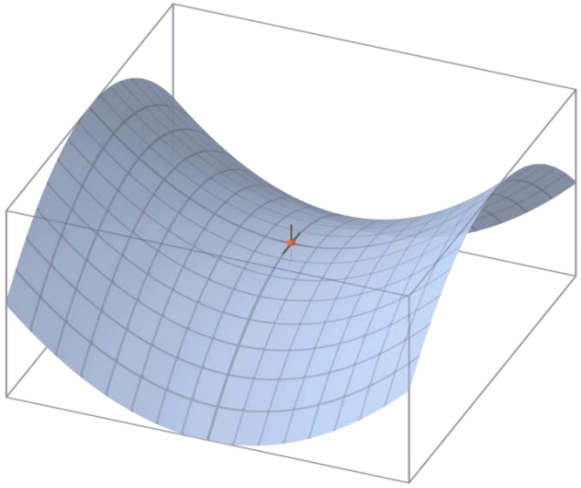


Figura 10 – Exemplo de ponto de sela. Extraída de Wikipedia.

3.3. Retropropagação do erro (*Error backpropagation*)

Conforme já exposto, os principais métodos de aprendizado em redes neurais são baseados no cálculo de derivadas da função custo com respeito aos pesos sinápticos. Busca-se, fundamentalmente, encontrar o conjunto de pesos que minimize a medida de erro escolhida. A chave, portanto, é encontrar uma maneira de calcular o vetor gradiente da função custo.

Um obstáculo imediato é que os pesos sinápticos não aparecem de maneira explícita na expressão de $J(\mathbf{w})$. Veja, por exemplo, o caso do erro quadrático médio (MSE, do inglês *mean-squared error*):

$$J_{\text{MSE}}(\cdot) = \frac{1}{N}\sum_{n=1}^{N}\sum_{j=1}^{r} e_j^2(n) = \frac{1}{N}\sum_{n=1}^{N}\sum_{j=1}^{r}\left(d_j(n) - y_j^{L+1}(n)\right)^2$$

Para fazer com que a dependência de $J(\cdot)$ em relação aos pesos fique em evidência, é preciso recorrer a aplicações sucessivas da *regra da cadeia*. Na elegante notação de Leibniz, essa regra nos informa que $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$.

Voltando ao exemplo do MSE, vemos que as saídas da última camada aparecem de maneira direta. Isso significa que é simples obter as derivadas de $J(\cdot)$ com respeito aos pesos dos neurônios desta camada. No entanto, quando se busca avaliar as derivadas com respeito aos pesos associados às camadas anteriores, a situação é um

pouco mais complexa, pois não há mais uma dependência direta. Como atribuir a cada neurônio "longínquo" seu devido crédito na composição da saída e do erro?

Essa "caminhada de trás para a frente", *i.e.*, da saída – na qual se observa o erro – para a entrada, tendo por base a regra da cadeia, corresponde ao processo conhecido como **retropropagação de erro** (*error backpropagation*, ou simplesmente *backpropagation*) (HAYKIN, 2008), que é utilizado até hoje nas redes profundas (GOODFELLOW ET AL., 2015).

Faremos a dedução desta importante técnica considerando uma rede MLP com duas camadas intermediárias, cuja estrutura é ilustrada na Figura 10. Embora tratemos deste caso particular, ele é suficiente para formalizar o procedimento recursivo de cálculo das derivadas, podendo ser estendido com naturalidade para um número arbitrário de camadas intermediárias.

Considerações iniciais:

- A rede possui m entradas, $L = 2$ camadas intermediárias, e r saídas.
- Denotamos por ${w_{ij}}^{l}$ o peso sináptico que pondera a j-ésima entrada do i-ésimo neurônio da l-ésima camada. A saída produzida por este neurônio é denotada por ${y_i}^{l} = \varphi^{l}({v_i}^{l})$.
- As camadas intermediárias possuem n_1 e n_2 neurônios, respectivamente.

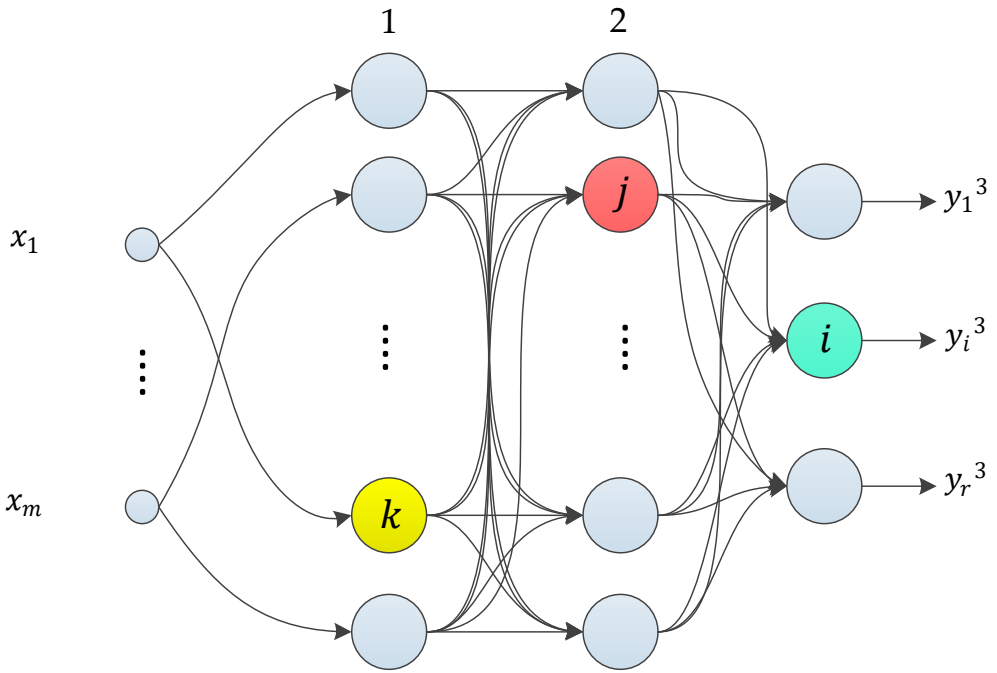
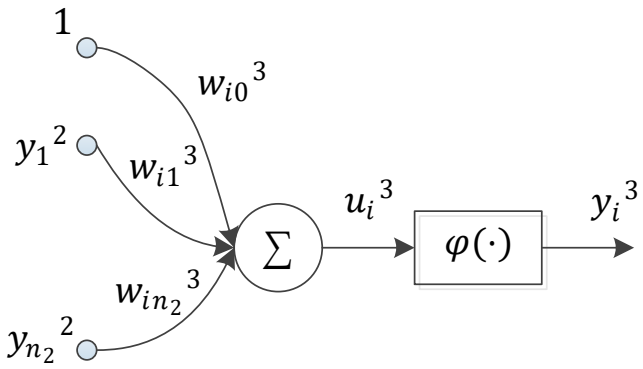


Figura 10 – MLP com duas camadas intermediárias.

3.3.1 Derivada – Pesos da camada de saída

Vamos olhar com mais atenção para o diagrama que mostra o i-ésimo neurônio da camada de saída:



Utilizando a regra da cadeia, vamos determinar a expressão de $\frac{dJ(\cdot)}{dw_{ij}{}^3}$:

$$\frac{dJ(\cdot)}{dw_{ij}{}^3} = \underbrace{\frac{dJ(\cdot)}{dy_i{}^3}\frac{dy_i{}^3}{du_i{}^3}}_{\frac{dJ(\cdot)}{du_i{}^3}}\frac{du_i{}^3}{dw_{ij}{}^3} \tag{8}$$

Ora, $\frac{dy_i{}^3}{du_i{}^3} = \dot{\varphi}^3(u_i{}^3)$, *i.e.*, é a derivada da função de ativação com respeito a seu argumento. Ademais, $\frac{du_i{}^3}{dw_{ij}{}^3} = y_j{}^2$, *i.e.*, é a entrada associada ao peso $w_{ij}{}^3$.

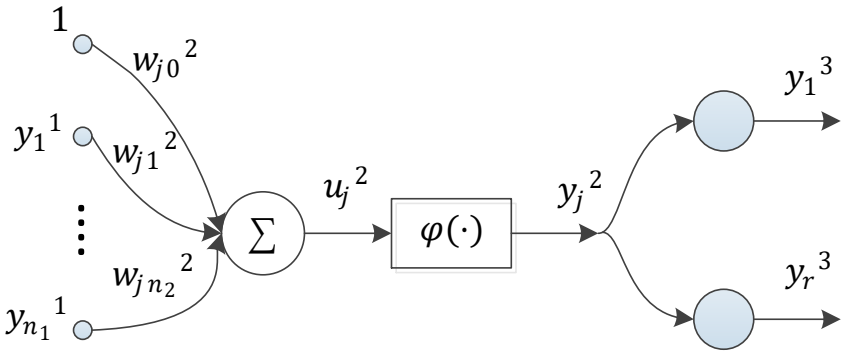
Assim,

$$\frac{dJ(\cdot)}{dw_{ij}{}^3} = \frac{dJ(\cdot)}{dy_i{}^3}\dot{\varphi}^3(u_i{}^3)y_j{}^2 \qquad (9)$$

A expressão do primeiro termo em (9) depende diretamente da função custo escolhida para o treinamento.

3.3.2 Derivada – pesos da segunda camada intermediária

À semelhança da seção anterior, vamos concentrar a atenção no j-ésimo neurônio da última camada intermediária.



Observe que a saída deste neurônio, $y_j{}^2$, afeta todas as saídas da rede neural. Interessantemente, uma vez que as funções custo normalmente envolvem uma soma dos erros para cada saída da rede, no cálculo do gradiente aparecerá uma soma das derivadas de $J(\cdot)$ partindo de cada saída.

Além disso, como o algoritmo *backpropagation* parte da camada de saída, já foram calculadas as derivadas $\frac{dJ(\cdot)}{du_k{}^3}, k = 1, \ldots, r$. Assim, vamos criar uma relação entre as $\frac{dJ(\cdot)}{du_j{}^2}$ e $\frac{dJ(\cdot)}{du_k{}^3}, k = 1, \ldots, r$.

$$\frac{dJ(\cdot)}{du_j{}^2} = \frac{dJ(\cdot)}{du_1{}^3}\frac{du_1{}^3}{dy_j{}^2}\frac{dy_j{}^2}{du_j{}^2} + \frac{dJ(\cdot)}{du_2{}^3}\frac{du_2{}^3}{dy_j{}^2}\frac{dy_j{}^2}{du_j{}^2} + \cdots + \frac{dJ(\cdot)}{du_r{}^3}\frac{du_r{}^3}{dy_j{}^2}\frac{dy_j{}^2}{du_j{}^2} \qquad (10)$$

Simplificando um pouco a equação, temos que:

$$\frac{dJ(\cdot)}{du_j{}^2} = \left(\sum_{k=1}^{r} \frac{dJ(\cdot)}{du_k{}^3} \frac{du_k{}^3}{dy_j{}^2} \right) \frac{dy_j{}^2}{du_j{}^2} \tag{11}$$

Ora, $\frac{du_k{}^3}{dy_j{}^2} = w_{kj}{}^3$ e $\frac{dy_j{}^2}{du_j{}^2} = \dot{\varphi}^2(u_j{}^2)$. Assim, percebemos que é possível obter a derivada com respeito a $u_j{}^2$ conhecendo as derivadas da função com relação a $u_k{}^3, k = 1, \ldots, r$.

Com isso, a derivada da função custo com respeito ao q-ésimo peso $w_{jq}{}^2$ deste neurônio é dada por:

$$\frac{dJ(\cdot)}{dw_{jq}{}^2} = \frac{dJ(\cdot)}{du_j{}^2} \frac{du_j{}^2}{dw_{jq}{}^2} = \frac{dJ(\cdot)}{du_j{}^2} y_q{}^1 \tag{12}$$

3.3.3. Derivada – pesos da primeira camada intermediária

Neste momento, as derivadas $\frac{dJ(\cdot)}{du_j{}^2}, j = 1, \ldots, n_2$, já foram calculadas. Então, seguindo um raciocínio muito similar ao da seção anterior, é possível concluir que:

$$\frac{dJ(\cdot)}{du_k{}^1} = \left(\sum_{j=1}^{n_2} \frac{dJ(\cdot)}{du_j{}^2} \frac{du_j{}^2}{dy_k{}^1} \right) \frac{dy_k{}^1}{du_k{}^1} \tag{13}$$

Então,

$$\frac{dJ(\cdot)}{dw_{kp}{}^1} = \frac{dJ(\cdot)}{du_k{}^1} \frac{du_k{}^1}{dw_{kp}{}^1}, \tag{14}$$

onde $\frac{du_k{}^1}{dw_{kp}{}^1} = y_p{}^0 = x_p, p = 0, \ldots, m.$

3.4. Metodologias de treinamento

Vimos que a obtenção do gradiente pode se dar num processo de retropropagação em que há uma parte direta (*forward*) de apresentação de um dado e obtenção da resposta da rede, bem como uma etapa de retropropagação (*backward*) em que se calculam as derivadas necessárias.

O vetor gradiente utilizado no ajuste dos parâmetros pode ser definido segundo uma estimação *online* (ou padrão-a-padrão), uma estimação em batelada (*batch*), ou

com base em pequenos lotes de dados (*mini-batches*). Vejamos primeiramente a noção geral de adaptação com estimação *online*, como expressa no seguinte algoritmo.

- Defina uma condição inicial para o vetor de pesos $\mathbf{w}$ e um passo η pequeno;
- Faça $k = 0, t = 0$ e calcule $J(\mathbf{w}(k))$;
- Enquanto o critério de parada não for atendido, faça:
  - Ordene aleatoriamente os padrões de entrada-saída;
  - Para l variando de 1 até N, faça:
    - Apresente o padrão l de entrada à rede;
    - Calcule $J_l(\mathbf{w}(t))$ e $\nabla J_l(\mathbf{w}(t))$;
    - $\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \nabla J_l(\mathbf{w}(t))$; $t = t + 1$;
  - $k = k + 1$;
  - Calcule $J(\mathbf{w}(k))$;

Algoritmo 1 – Adaptação padrão-a-padrão.

O outro extremo seria, como dito, utilizar todo o conjunto de dados para estimar o gradiente antes de dar o passo do processo iterativo. O Algoritmo 2 ilustra o *modus operandi* correspondente (novamente considerando uma metodologia de primeira ordem).

- Defina uma condição inicial para o vetor de pesos $\mathbf{w}$ e um passo η pequeno;
- Faça $k = 0$ e calcule $J(\mathbf{w}(k))$;
- Enquanto o critério de parada não for atendido, faça:
    - Para l variando de 1 até N, faça:
        - Apresente o padrão l de entrada à rede;
        - Calcule $J_l(\mathbf{w}(k))$ e $\nabla J_l(\mathbf{w}(k))$;
    - $\mathbf{w}(k+1) = \mathbf{w}(k) - \frac{\eta}{N}\sum_{l=1}^{N} \nabla J_l(\mathbf{w}(k))$;
    - $k = k + 1$;
    - Calcule $J(\mathbf{w}(k))$;

Algoritmo 2 – Adaptação em esquema *batch*.

A terceira opção, bastante comum no contexto de *deep learning*, estima o vetor gradiente a partir de um meio-termo entre usar uma única amostra e usar todas as amostras. Cada *mini-batch* é formado por amostras aleatoriamente tomadas do conjunto de dados. O Algoritmo 3 ilustra isso.

- Defina uma condição inicial para o vetor de pesos $\mathbf{w}$ e um passo η pequeno;
- Faça $k = 0$ e calcule $J(\mathbf{w}(k))$;
- Enquanto o critério de parada não for atendido, faça:
  - Para l variando de 1 até N_B, faça:
    - Apresente o padrão l de entrada, amostrado para compor um *mini-batch*, à rede;
    - Calcule $J_l(\mathbf{w}(k))$ e $\nabla J_l(\mathbf{w}(k))$;
  - $\mathbf{w}(k+1) = \mathbf{w}(k) - \frac{\eta}{N_B}\sum_{l=1}^{N_B} \nabla J_l(\mathbf{w}(k))$;
  - $k = k + 1$;
  - Calcule $J(\mathbf{w}(k))$;

Algoritmo 3 – Adaptação Baseada em *Minibatch*.

Dizemos que uma *época* (em inglês, *epoch*) se encerra quando todos os dados do conjunto de treinamento são oferecidos uma vez para a rede. Tipicamente, o processo de aprendizado da rede ocorre até que um número máximo de épocas seja atingido.

3.5. Algoritmos de otimização

Existem muitos desafios associados ao treinamento supervisionado de redes neurais, especialmente em *deep learning*. A não-convexidade da superfície de erro

impõe alguns obstáculos para o progresso dos algoritmos de busca, tais como a presença de platôs, pontos de sela e mínimos locais de baixa qualidade.

Isso levou à proposição de algoritmos de otimização mais eficientes, no sentido de promoverem, em média, avanços expressivos na exploração da superfície de erro a cada iteração. Nesta seção, vamos considerar alguns métodos que se credenciam como alternativas promissoras ao gradiente descendente (SGD, do inglês *stochastic gradiente descent*). Tais métodos, embora também sejam de primeira ordem, diferem na forma como o passo de adaptação e a direção de ajuste são estimados a cada iteração (RUDER, 2017; GÉRON, 2019).

3.5.1 Gradiente com momento

A direção de ajuste não é mais determinada somente pelo gradiente atual, mas também por um termo de momento, o qual traz a informação de gradientes

anteriores acumulados. Esta ideia pode, em certas situações, reduzir oscilações durante a exploração da superfície de erro, bem como acelerar o progresso sempre que gradientes subsequentes preservarem direções de busca.

Regra de atualização:

$$\begin{aligned}\mathbf{m}(n) &= \beta\mathbf{m}(n-1) - \eta\nabla J(\mathbf{w}(n)) \\ \mathbf{w}(n+1) &= \mathbf{w}(n) + \mathbf{m}(n),\end{aligned} \tag{15}$$

onde β exerce o papel de um fator de esquecimento. Valores típicos de β são 0,5, 0,9 e 0,99. A Figura 11 apresenta uma visão do potencial benefício do uso do termo de momento.

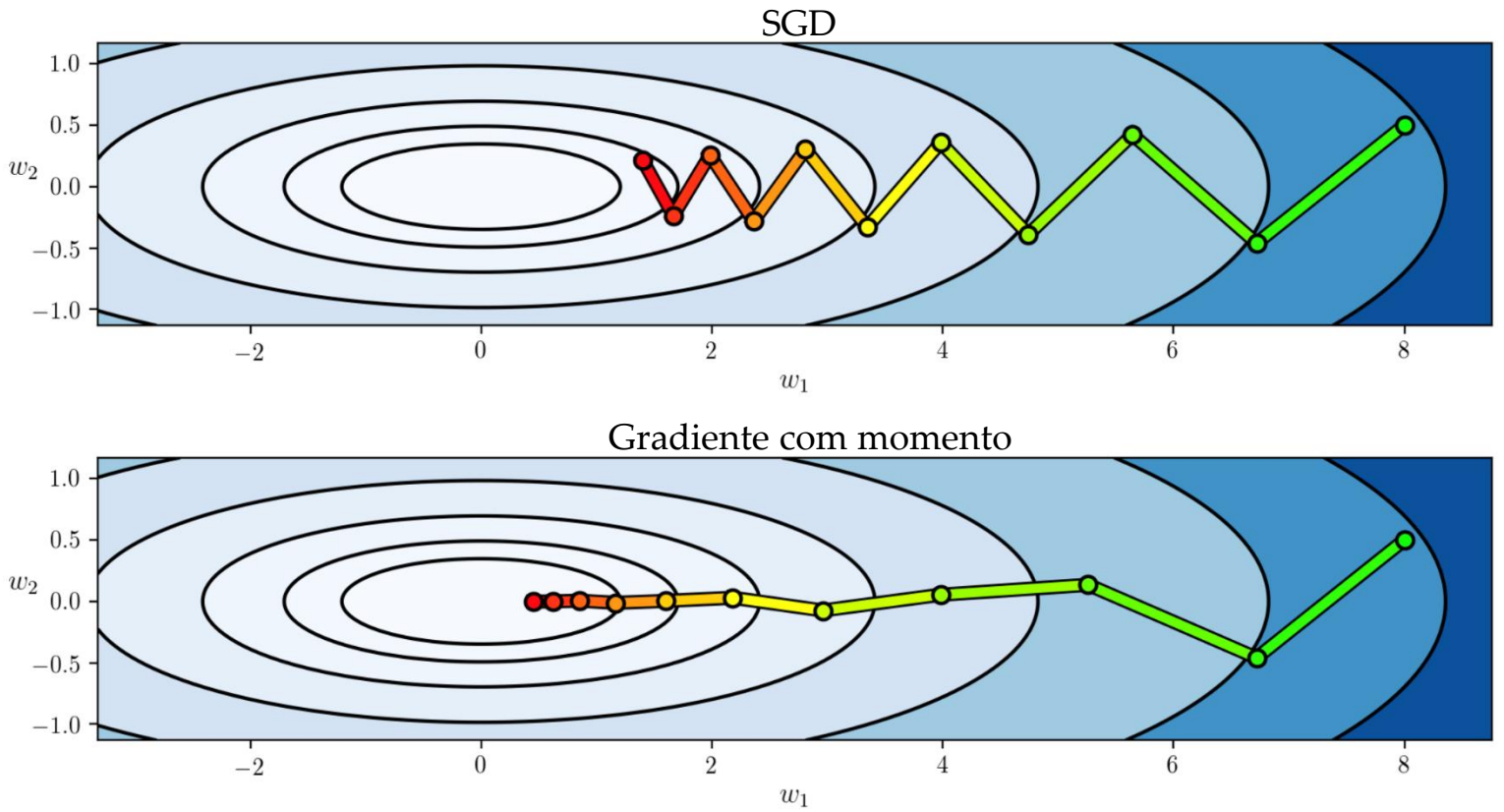


Figura 11 – Contribuição do termo de momento na redução de oscilações. Adaptada de https://www.quora.com/What-exactly-is-momentum-in-machine-learning.

3.5.2 *Nesterov Accelerated Gradient* (NAG)

O método do momento de Nesterov pode ser visto, essencialmente, como uma variação do método exposto na seção anterior em que a estimativa do vetor

gradiente não é feita na posição atual do vetor de parâmetros $\mathbf{w}(n)$, mas sim sobre $\mathbf{w}(n) + \beta\mathbf{m}(n-1)$. Essa alteração funciona como um fator de correção, podendo beneficiar a velocidade de convergência, conforme ilustra a Figura 12.

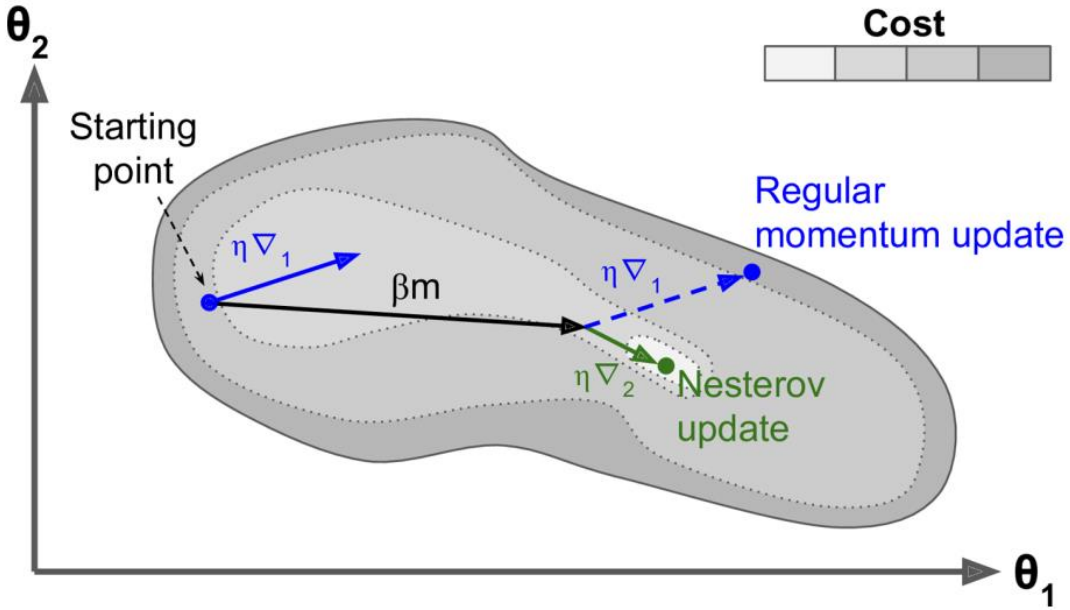


Figura 12 – Atualização do vetor de parâmetros com NAG. Extraída de (GÉRON, 2019).

3.5.3 Algoritmos com passo adaptativo

Os algoritmos adaptativos foram motivados basicamente pela limitação associada ao uso de um passo de adaptação (η) fixo para todos os pesos sinápticos e para

todas as iterações, ou, então, pela definição prévia de uma política de ajuste dessa taxa de aprendizado.

Deve-se levar em conta que uma taxa de aprendizado muito pequena produz uma convergência muito lenta do treinamento, enquanto uma taxa de aprendizado muito elevada pode impedir a convergência do processo de ajuste, seja por apresentar oscilações em torno de um ótimo local, seja por divergir.

Além disso, usar uma mesma taxa de aprendizado para todos os pesos ajustáveis pode não ser a melhor solução. Na verdade, tende a ser mais produtivo promover ajustes mais intensos nos pesos que, com menor frequência, foram modificados em um histórico recente.

3.5.4 AdaGrad

O AdaGrad realiza um ajuste do passo de aprendizado de acordo com a intensidade acumulada com que cada peso já foi ajustado ao longo da busca, até a iteração

corrente. Matematicamente, o AdaGrad implementa a seguinte regra de atualização (DUCHI ET AL., 2011):

$$\begin{aligned}\mathbf{s}(n) &= \mathbf{s}(n-1) + \nabla J(\mathbf{w}(n)) \otimes \nabla J(\mathbf{w}(n)) \\ \mathbf{w}(n+1) &= \mathbf{w}(n) - \eta \nabla J\big(\mathbf{w}(n)\big) \oslash \sqrt{\mathbf{s}(n) + \varepsilon},\end{aligned} \tag{16}$$

onde $\otimes$ e $\oslash$ denotam uma multiplicação e uma divisão elemento-a-elemento, respectivamente.

- No primeiro passo, acumulamos o quadrado dos gradientes no vetor $\mathbf{s}(n)$. Se em uma dada direção a função custo é íngreme, então o elemento em $\mathbf{s}(n)$ associado a esta coordenada vai se tornar cada vez maior.
- No segundo passo, o ajuste é feito na direção oposta ao gradiente, mas com um passo inversamente proporcional a $\mathbf{s}(n)$, de modo a frear o ajuste nas direções com maior inclinação da superfície de erro.

Uma desvantagem deste algoritmo é que os passos se tornam menores a cada iteração, o que pode levar a uma parada prematura do processo de busca. Não obstante, a exposição do AdaGrad serve como preparação para os dois algoritmos seguintes.

3.5.5 RMSProp

O algoritmo RMSProp, criado por G. Hinton e T. Tieleman, em 2012, corrige o problema do AdaGrad acumulando apenas os gradientes das iterações mais recentes. Agora, há um decaimento exponencial no ajuste de $\mathbf{s}(n)$, conforme a seguinte expressão:

$$\mathbf{s}(n) = \gamma \mathbf{s}(n-1) + (1-\gamma)\nabla J(\mathbf{w}(n)) \otimes \nabla J(\mathbf{w}(n)) \qquad (17)$$

A taxa de decaimento γ usualmente assume o valor 0,9.

3.5.6 Adam

O algoritmo Adam (*Adaptive Moment Estimation*) combina as ideias de gradiente com momento e de passo adaptativo, nos moldes do RMSProp, em um único método (Kingma & Ba, 2014).

Regra de atualização:

$$\begin{aligned}
\mathbf{m}(n) &= \beta\mathbf{m}(n-1) - (1-\beta)\nabla J\big(\mathbf{w}(n)\big) \\
\mathbf{s}(n) &= \gamma\mathbf{s}(n-1) + (1-\gamma)\nabla J\big(\mathbf{w}(n)\big) \otimes \nabla J\big(\mathbf{w}(n)\big) \\
\hat{\mathbf{m}}(n) &= \frac{\mathbf{m}(n)}{1-\beta^n} \\
\hat{\mathbf{s}}(n) &= \frac{\mathbf{s}(n)}{1-\gamma^n} \\
\mathbf{w}(n+1) &= \mathbf{w}(n) + \eta\hat{\mathbf{m}}(n) \oslash \sqrt{\hat{\mathbf{s}}(n) + \varepsilon}
\end{aligned} \tag{18}$$

Uma vez que $\mathbf{m}(n)$ e $\mathbf{s}(n)$ são normalmente inicializados como vetores nulos, esses momentos têm um viés e precisam de uma correção (daí o algoritmo calcular as variáveis $\hat{\mathbf{m}}(n)$ e $\hat{\mathbf{s}}(n)$).

3.6. Inicialização dos pesos

Uma vez que os métodos de treinamento de redes neurais são iterativos, eles dependem de uma inicialização. Como os métodos são de busca local, a inicialização pode afetar drasticamente a qualidade da solução obtida. Também pode haver variações expressivas do ponto de vista da velocidade de convergência.

O padrão é usar uma abordagem aleatória para aumentar a diversidade de comportamentos dos neurônios na rede (GOODFELLOW ET AL., 2016).

Cuidado: a distribuição e a ordem de grandeza dos pesos devem ser pensadas tendo em vista o perfil da função de ativação dos neurônios; alguns esquemas foram elaborados, até mesmo em anos recentes, com o intuito de evitar que os

neurônios partam de uma condição desfavorável para o aprendizado. Por exemplo, deve-se evitar que os neurônios já comecem operando na região de saturação da função de ativação, pois, neste caso, o gradiente será praticamente nulo, dificultando o progresso do treinamento.

- *Xavier (Glorot) initialization* (GLOROT & BENGIO, 2010): considere uma camada com L entradas e M neurônios. Então, cada peso é amostrado conforme:

$$w_{ij}{}^{l} \sim N\left(0, \frac{1}{(L+M)/2}\right)$$

ou

$$w_{ij}{}^{l} \sim U\left(-\sqrt{\frac{6}{L+M}}, \sqrt{\frac{6}{L+M}}\right).$$

- *LeCun initialization*: $w_{ij}{}^{l} \sim N\left(0, \frac{1}{L}\right)$.

4. *Batch normalization*

Outra técnica desenvolvida para favorecer o treinamento de redes neurais é conhecida como *batch normalization* (BN) (IOFFE & SZEGEDY, 2015). De forma resumida, uma camada do tipo BN remove a média e normaliza a variância de cada variável de entrada, para, então, re-escalar e deslocar o resultado usando dois novos vetores de parâmetros. Em outras palavras, esta operação permite que o modelo aprenda a média e a escala ótima para cada variável.

$$\begin{aligned}
\boldsymbol{\mu}_B &= \frac{1}{N_B}\sum_{i=1}^{N_B}\mathbf{x}(i) \\
{\boldsymbol{\sigma}_B}^2 &= \frac{1}{N_B}\sum_{i=1}^{N_B}(\mathbf{x}(i)-\boldsymbol{\mu}_B)^2 \\
\hat{\mathbf{x}}(i) &= \frac{\mathbf{x}(i)-\boldsymbol{\mu}_B}{\sqrt{{\boldsymbol{\sigma}_B}^2+\varepsilon}} \\
\mathbf{z}(i) &= \boldsymbol{\rho}\otimes\hat{\mathbf{x}}(i)+\boldsymbol{\tau}
\end{aligned} \tag{19}$$

No algoritmo mostrado em (19):

- $\boldsymbol{\mu}_B$ é o vetor de média avaliado em um *mini-batch* de N_B amostras.
- $\boldsymbol{\sigma}_B{}^2$ é o vetor com as variâncias estimadas de cada variável considerando um *mini-batch* de N_B amostras.
- $\hat{\mathbf{x}}(i)$ é o vetor com variáveis normalizadas, *i.e.*, com média nula e variância unitária.
- $\mathbf{z}(i)$ é o vetor de variáveis gerado pelo BN, em que cada variável possui média e variância definidas pelos vetores $\boldsymbol{\tau}$ e $\boldsymbol{\rho}$, respectivamente.
- $\boldsymbol{\mu}_B$ e $\boldsymbol{\sigma}_B{}^2$ são estimados durante o treinamento com base nos dados presentes no mini-batch; $\boldsymbol{\tau}$ e $\boldsymbol{\rho}$, por sua vez, contêm os parâmetros ajustáveis do BN.

Do ponto de vista do método de treinamento da rede neural, BN é apenas mais uma camada com operações que permitem a propagação do vetor gradiente.

Na etapa de teste da rede, $\boldsymbol{\mu}_B$ e $\boldsymbol{\sigma}_B{}^2$ são substituídos pelos resultados de médias móveis com decaimento exponencial, as quais são calculadas durante o treinamento.

Possíveis benefícios:

- Suavização da superfície de erro, o que leva a gradientes mais estáveis e torna possível o uso de passos de adaptação um pouco maiores (SANTUNKAR ET AL., 2019).
- Reduz a sensibilidade em relação à inicialização dos pesos.
- Diminui as chances de o gradiente desaparecer (fenômeno conhecido como *vanishing gradient*) durante o treinamento, ou de ele crescer excessivamente em magnitude (*exploding gradient*).

5. Referências bibliográficas

CYBENKO, G., "Approximation by Superpositions of a Sigmoidal Function", *Mathematics of Control, Signals and Systems*, vol. 2, no. 4, pp. 303 – 314, 1989.

DUCHI, J., HAZAN, E., SINGER, Y., "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization", *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, 2011.

GÉRON, A., **Hands-on Machine Learning with Scikit-Learn, Keras & Tensorflow**, O'Reilly Media, 2ª ed., 2019.

GLOROT, X., BENGIO, Y., "Understanding the Difficulty of Training Deep Feedforward Neural Networks", Proceedings of the 13th International Conference on Artificial Intelligence and Statistics, pp. 249-256, 2010.

GOODFELLOW, I., BENGIO, Y., COURVILLE, A., **Deep Learning**, MIT Press, 2016.

HAYKIN, S. **Neural Networks and Learning Machines**, 3rd edition, Prentice-Hall, 2008.

HORNIK, K., STINCHCOMBE, M., WHITE, H., "Multi-layer feedforward networks are universal approximators", *Neural Networks*, vol. 2, no. 5, pp. 359-366, 1989.

HORNIK, K., STINCHCOMBE, M., WHITE, H., "Universal approximation of an unknown function and its derivatives using multilayer feedforward networks", *Neural Networks*, vol. 3, no. 5, pp. 551-560, 1990.

IOFFE, S., SZEGEDY, C., "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", Proceedings of the 32nd International Conference on Machine Learning, pp. 448-456, 2015.

LUENBERGER, D. G., **Linear and Nonlinear Programming**, Springer, 2ª ed., 2003.

MCCULLOCH, W., PITTS, W., "A Logical Calculus of the Ideas Immanent in Nervous Activity", *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115 – 133, 1943.

PEDAMONTI, D.,"Comparison of Non-Linear Activation Functions for Deep Neural Networks on MNIST Classification Task", *arXiv:1804.02763v1*, 2018.

RUDER, S., "An Overview of Gradient Descent Optimization Algorithms", *arXiv: 1609.04747v2*, 2017.

SANTUNKAR, S., TSIPRAS, D., ILYAS, A., MADRY, A., "How Does Batch Normalization Help Optimization?", *arXiv:1805.11604v5*, 2019.

VON ZUBEN, F. J., **Notas de Aulas do Curso "Redes Neurais" (IA353),** disponíveis em *http://www.dca.fee.unicamp.br/~vonzuben/courses/ia353.html*

WIKIPEDIA, **Artigos e Figuras Diversas**.