

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

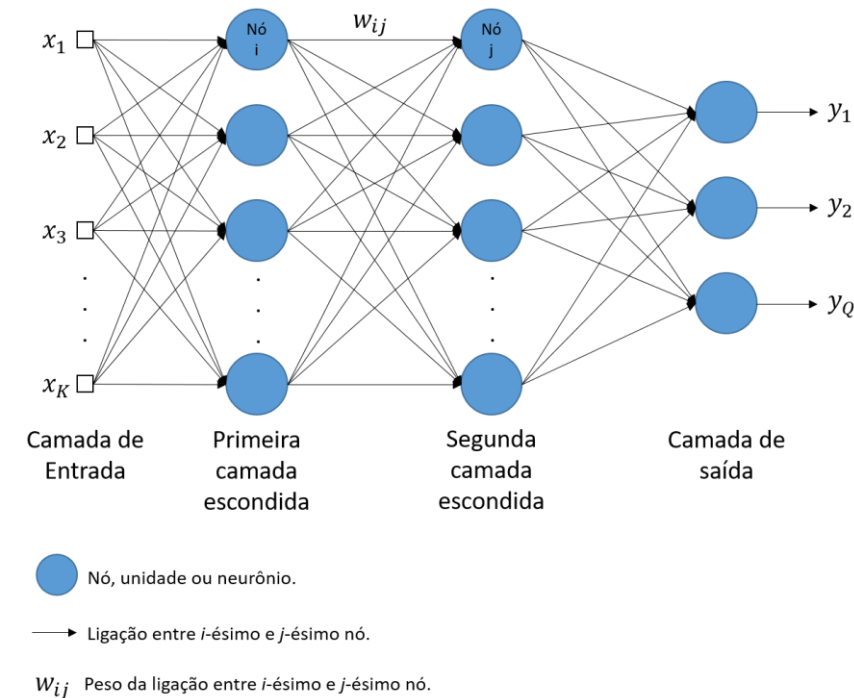


Inatel

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

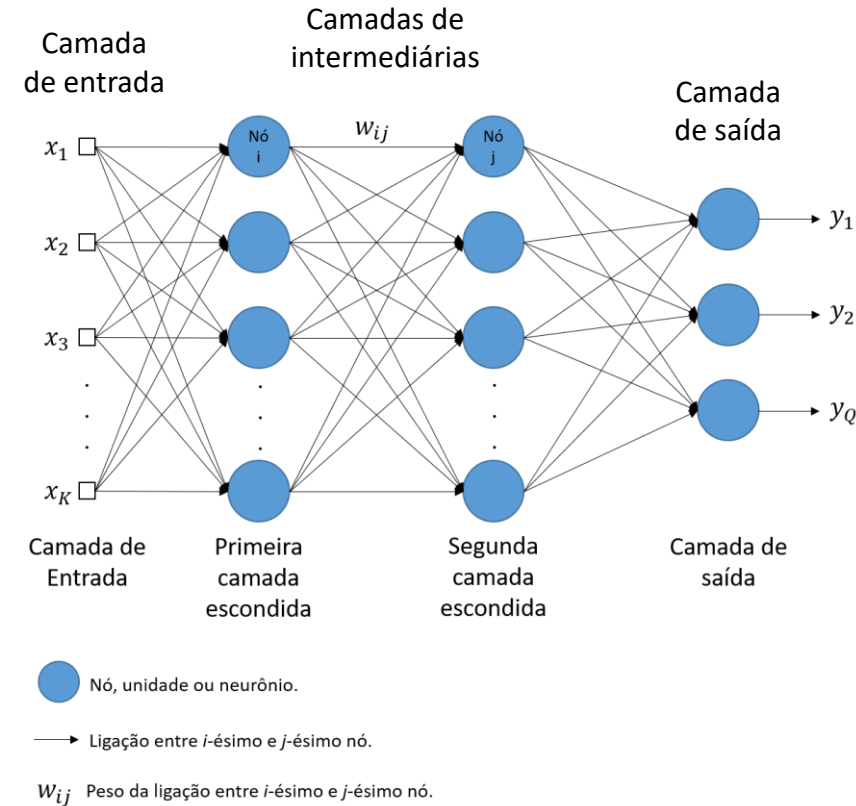
Perceptron de Múltiplas Camadas

- Em termos gerais, uma **rede neural** nada mais é do que uma coleção de **neurônios** (que também são chamados de **nós** ou **unidades**) 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** 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 de **perceptrons**.
- A RNA resultante é denominada Perceptron de Múltiplas Camadas (do inglês *Multilayer Perceptron* - MLP).



Perceptron de Múltiplas Camadas

- Um exemplo de rede MLP, com duas camadas intermediárias (ou escondidas, ocultas), 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 do inglês Deep Neural Network - DNN).
- **OBS.:** Em particular, uma MLP pode resolver o problema do XOR (lembre-se que um **perceptron** não é capaz de realizar essa tarefa).



Perceptron de Múltiplas Camadas

- A ***camada de entrada*** nada mais é que o ponto de 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 para formar as saídas.
- Redes MLPs são formadas por múltiplas camadas de ***perceptrons***: portanto, naturalmente, tais redes têm por base o ***modelo de neurônio do perceptron***.
- Esse modelo, discutido na aula anterior, é mostrado na figura seguinte.

Perceptron de Múltiplas Camadas

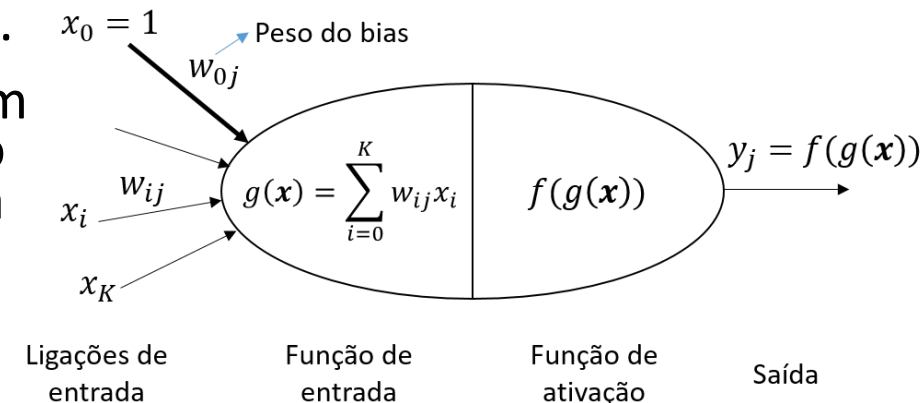
- Uma **ligação** do **nó** i para o **nó** j serve para propagar o sinal de ativação do **nó** i para o **nó** j . Cada **ligação** tem um **peso** associado, w_{ij} , que determina a **força** e o **sinal** da **ligação**.
- Assim como nos modelos de **regressão linear**, cada **nó** tem a entrada 0, i.e., x_0 , 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 inicialmente uma soma ponderada de suas entrada da seguinte forma

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

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

$$y_j = f(g(\mathbf{x})) = f\left(\sum_{i=0}^N w_{ij} x_i\right) = f(\mathbf{w}^T \mathbf{x}).$$

- Veremos a seguir que existem vários tipos de **funções de ativação**, $f(\cdot)$, que podem ser utilizadas pelos **nós** de uma rede MLP.



$$y_j = f\left(\sum_{i=0}^K w_{ij} x_i\right),$$
onde x_i é a saída da unidade i e w_{ij} é o peso conectando a saída da unidade i para esta unidade, a unidade j .

Funções de ativação

- Devido às suas características, não é comum se empregar a **função degrau** como função de ativação em MLPs pois ela possui derivada igual a 0 em todos os pontos, exceto em torno de 0, onde ela é indefinida.
- Até o surgimento das **redes neurais profundas**, a regra era se utilizar duas funções que são, em essência, versões suavizadas da **função degrau**: a **função logística** ou a **função tangente hiperbólica**.
- Essas funções possuem derivada definida e diferente de 0 em todos os pontos.
- A **função logística** tem a seguinte expressão:

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

- Sua derivada é dada por

$$\frac{dy_j}{dz_j} = py_j(1 - y_j) > 0,$$

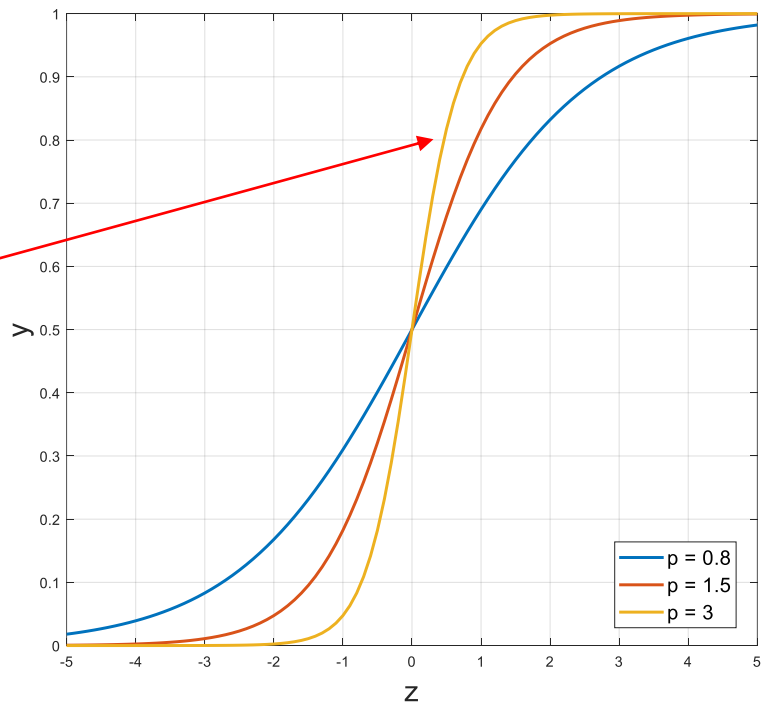
onde p é o **fator de suavização** da função de ativação logística.

- A derivada será importante, como veremos, no processo de aprendizado da rede neural.

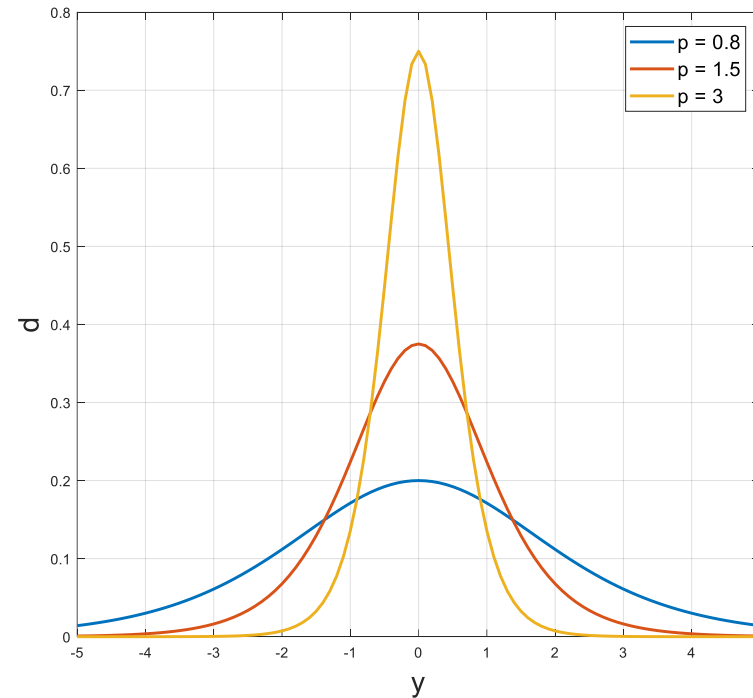
Funções de ativação

- A **função logística** e sua derivada para alguns valores do **fator de suavização** são mostradas nas figuras ao lado.

OBS.: Quanto maior p , mais próxima ela fica da função degrau.



Função Logística.



Derivada da Função Logística.

OBS.: tende ao impulso conforme p aumenta.

Funções de ativação

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

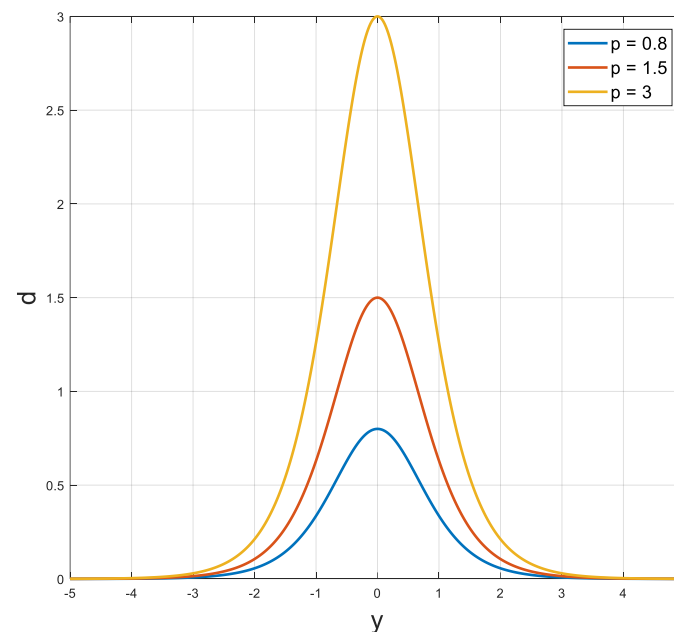
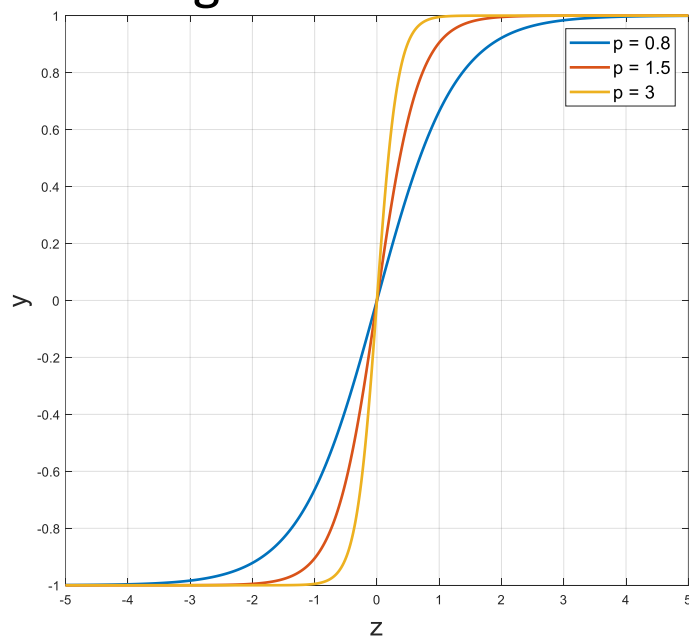
$$y_j = f(z_j) = \tanh(pz_j) = \frac{e^{pz_j} - e^{-pz_j}}{e^{pz_j} + e^{-pz_j}}.$$

- Sua derivada é dada por

$$\frac{dy_j}{dz_j} = p \left(1 - \tanh^2(pz_j) \right) > 0,$$

onde mais uma vez, o parâmetro p controla a suavidade da função. Essa função e sua derivada são mostradas nas figuras abaixo.

Função
Tangente
Hiperbólica.



Derivada da
Tangente
Hiperbólica.

Funções de ativação

- Embora as duas funções apresentadas anteriormente sejam clássicas, com o surgimento das **redes neurais profundas**, uma outra função, conhecida como **função retificadora**, passou a ser a bastante utilizada por uma série de questões numéricas e computacionais.

- A **função retificadora** tem sua expressão dada por

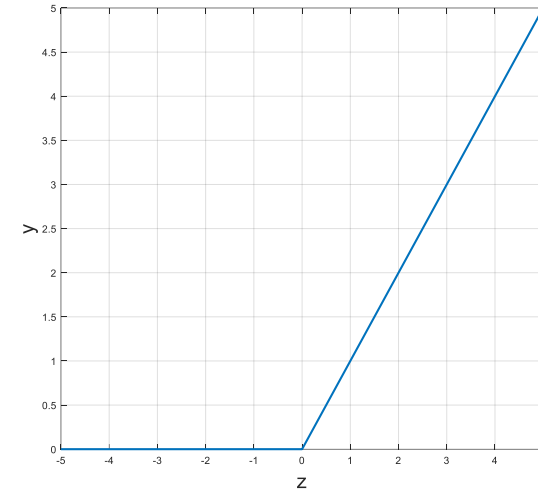
$$y_j = f(z_j) = \max(0, z_j).$$

- Sua derivada é dada por

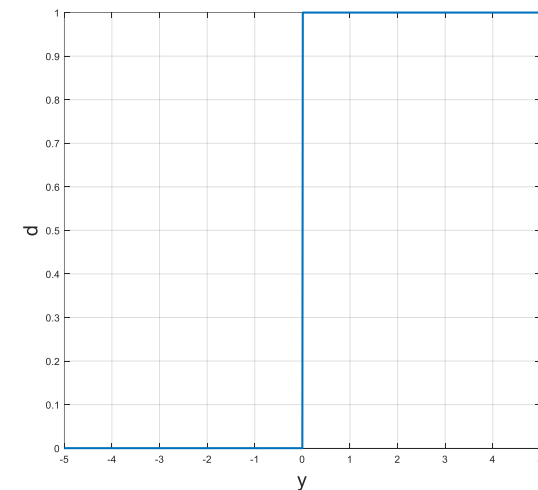
$$\frac{dy_j}{dz_j} = \begin{cases} 0, & \text{se } y_j < 0 \\ 1, & \text{se } y_j > 0 \end{cases}$$

e indefinido em $y_j = 0$, 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 ReLU (rectified linear unit).
- A **função retificadora** e sua derivada são mostradas nas figuras ao lado.



Função Retificadora.



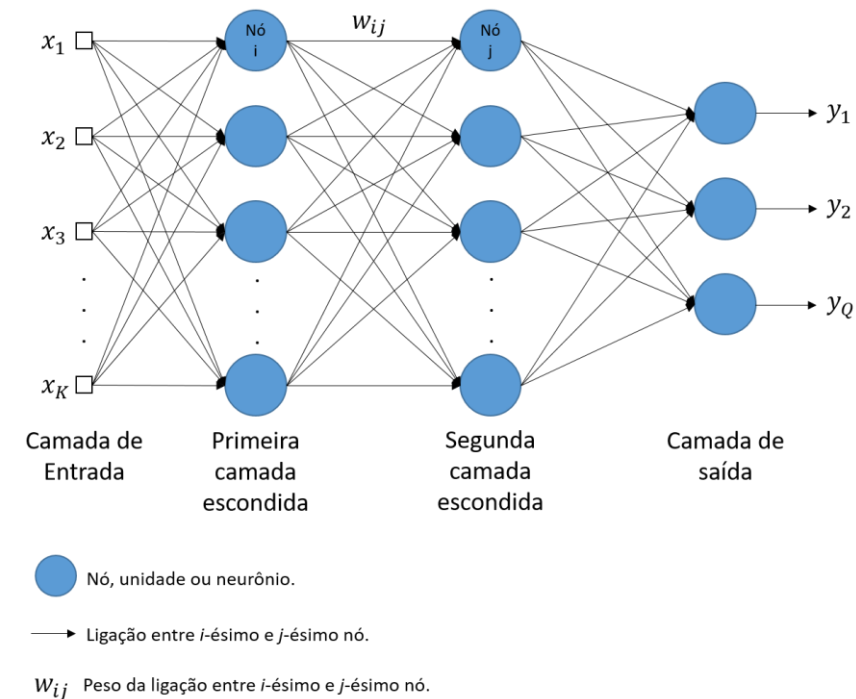
Derivada da Função Retificadora.

Funções de ativação

- Vantagens da ***função retificadora***:
 - A função e sua derivada são mais rápidas de calcular do que as funções logística e tangente hiperbólica.
 - Não sofre com o problema da diminuição do gradiente: o gradiente é multiplicado várias vezes com o algoritmo da ***retropropagação***, o que faz com que o gradiente se torne menor para as camadas inferiores, levando a uma mudança muito pequena ou até mesmo nenhuma mudança nos pesos das camadas inferiores.
- Outras funções de ativação são:
 - Identidade ou linear
 - Gaussian Error Linear Unit (GELU)
 - Leaky rectified linear unit (Leaky ReLU)
 - Gaussiana

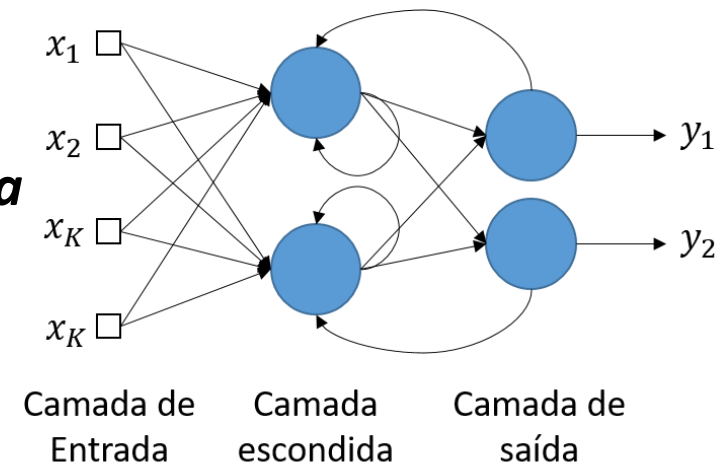
Conectando Neurônios

- Existem basicamente duas maneiras distintas para se conectar os **nós** (ou **neurônios**) de uma rede.
- 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** (*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.
- Esse tipo de rede representa uma função de suas entradas atuais e, portanto, não possui um estado interno além dos próprios pesos.



Conectando Neurônios

- Na figura ao lado, os **nós** da rede tem 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 de alguns **nós** alimentam **nós** da mesma camada (inclusive o próprio **nó**) ou de camadas anteriores.
- Isso significa que os níveis de ativação da rede formam um **sistema dinâmico** que pode atingir um estado estável, exibir oscilações ou mesmo um comportamento caótico, ou seja, divergir.
- Além disso, a resposta da rede a uma determinada entrada depende do seu estado inicial, que pode depender das entradas anteriores.
- Portanto, **redes recorrentes** podem suportar memória de curto prazo.
- Essas redes são úteis para o processamento de dados sequenciais, como som, dados de séries temporais ou linguagem natural (escrita e fala).



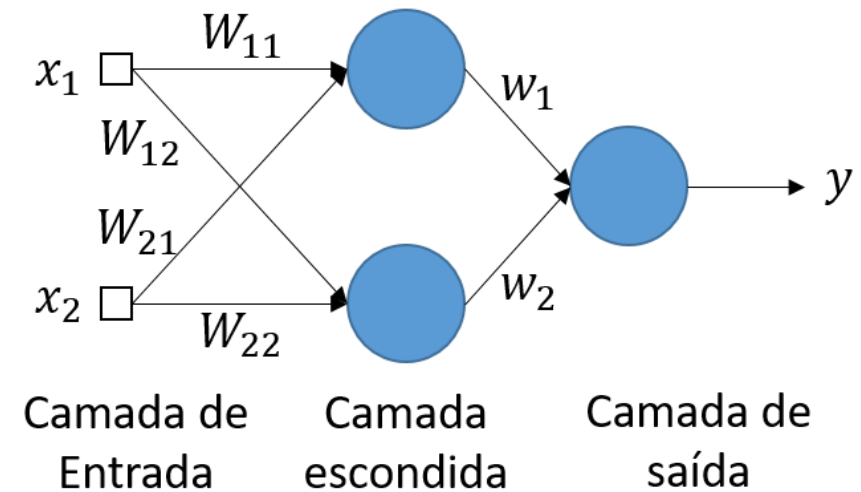
Regressão Não-Linear

- A rede MLP ao lado tem sua saída definida por

$$y = f(f(Wx)w),$$

onde f é a **função de ativação** escolhida.

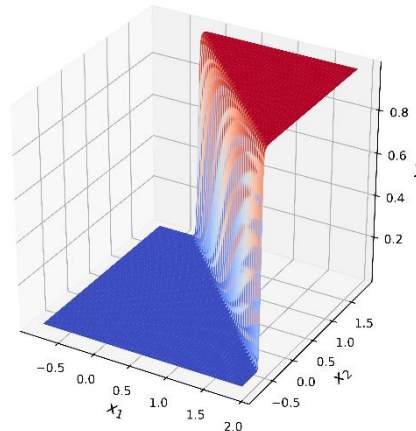
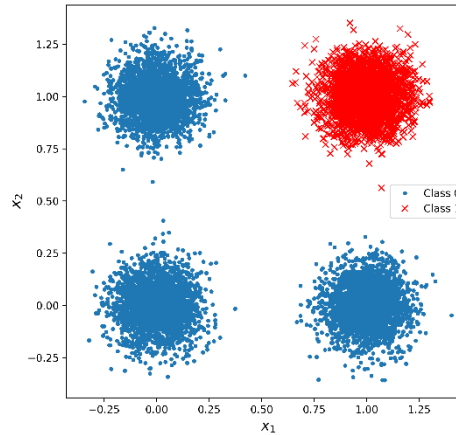
- Percebam que a saída da rede é dada pelo aninhamento das saídas de **funções de ativação não-lineares**.
- Sendo assim, as funções que uma rede pode representar podem ser altamente não-lineares dependendo da quantidade de camadas e nós.
- Portanto, redes neurais podem ser vistas como ferramentas para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



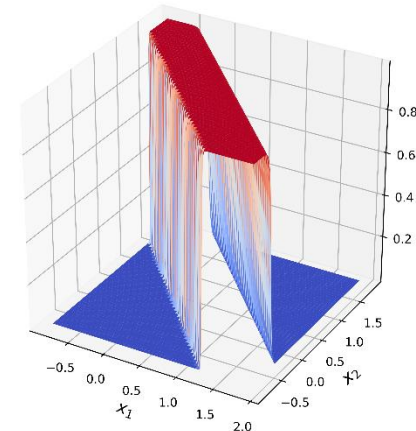
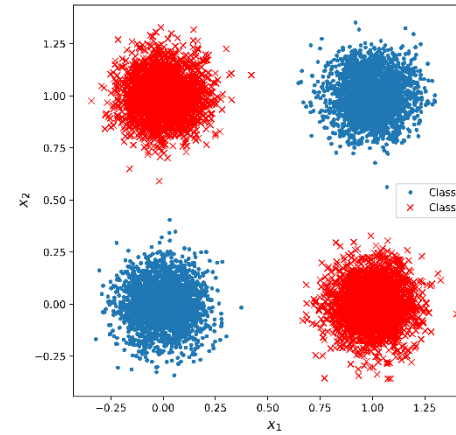
Aproximação universal de funções

- Um nó aproxima uma função de limiar suave.
- Combinando duas funções de limiar suave com direções opostas, podemos obter uma função em formato de onda.
- Combinando duas ondas perpendiculares, nós obtemos uma função em formato de cilindro.

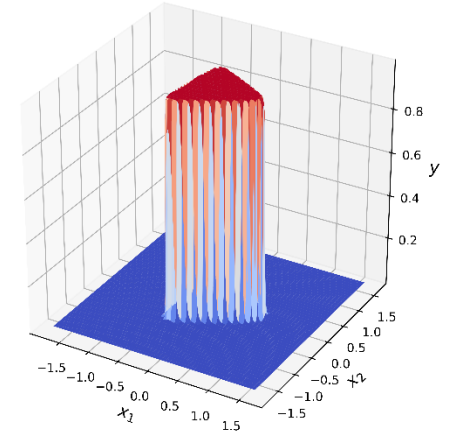
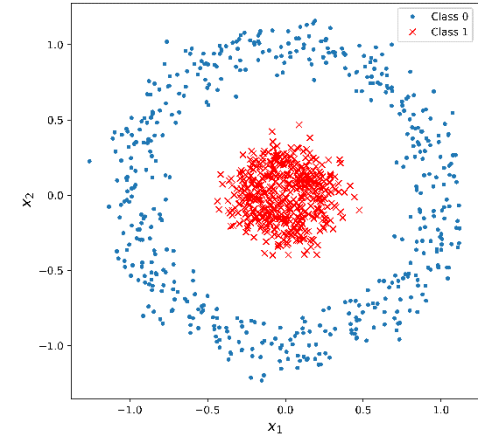
Função AND: MLP com 0 camadas escondidas, apenas um neurônio na camada de saída.
Total: 1 nó.



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.



Aprendizado em Redes Neurais

- Consideramos agora, o processo de otimização, ou seja, de adaptação dos ***pesos sinápticos***.
- Vamos considerar que o processo de otimização corresponde a uma tarefa de minimização de uma ***função custo***, $J(\mathbf{w})$, com respeito a um vetor de pesos \mathbf{w} .
- Portanto, o problema de aprendizado em redes neurais pode ser formulado como

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

- Normalmente, esse processo de otimização é conduzido de forma iterativa, o que dá um sentido mais natural à noção de aprendizado (como um processo gradual).
- Existem vários métodos de otimização aplicáveis, mas, sem dúvida, os mais utilizados são aqueles baseados nas derivadas da função custo, $J(\mathbf{w})$.

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***, geralmente agrupadas em um vetor chamado de ***vetor gradiente***:

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

- Como já vimos, o gradiente aponta na direção e sentido 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**

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

onde α é o **passo de aprendizagem**.

- Como já discutido anteriormente, a escolha do **passo de aprendizagem** é muito importante. Lembrem-se que passos muito grandes podem levar à instabilidade, enquanto passos muito pequenos podem levar a uma convergência muito lenta.
- 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 H** :

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

Aprendizado em Redes Neurais

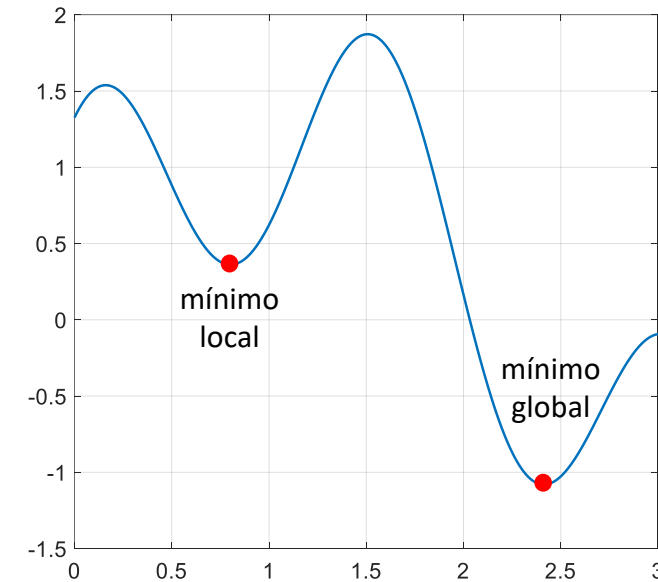
- De posse da **matriz hessiana**, é possível fazer uma aproximação de Taylor de segunda ordem da **função de custo**, o que leva à seguinte expressão para adaptação dos pesos:

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

- Essa expressão requer que a **matriz Hessiana** seja inversível, e também que ela seja **definida positiva** a cada iteração, $\mathbf{z}^T \mathbf{H} \mathbf{z} > 0 \ \forall \mathbf{z} \neq \mathbf{0}$ (vetor nulo).
- Uma vez que a aproximação de Taylor com informação de segunda ordem é mais ampla que aquela fornecida por métodos de primeira ordem, a tendência é que um método de **segunda ordem** convirja em menos passos que um método de **primeira ordem**.
- Entretanto, o cálculo exato da **matriz Hessiana** pode ser complicado em vários casos práticos. Porém, há um conjunto de métodos de segunda ordem que evitam esse cálculo direto, como os métodos **quase-Newton** ou os métodos de **gradiente escalonado**, que representam uma espécie de compromisso entre complexidade e desempenho.

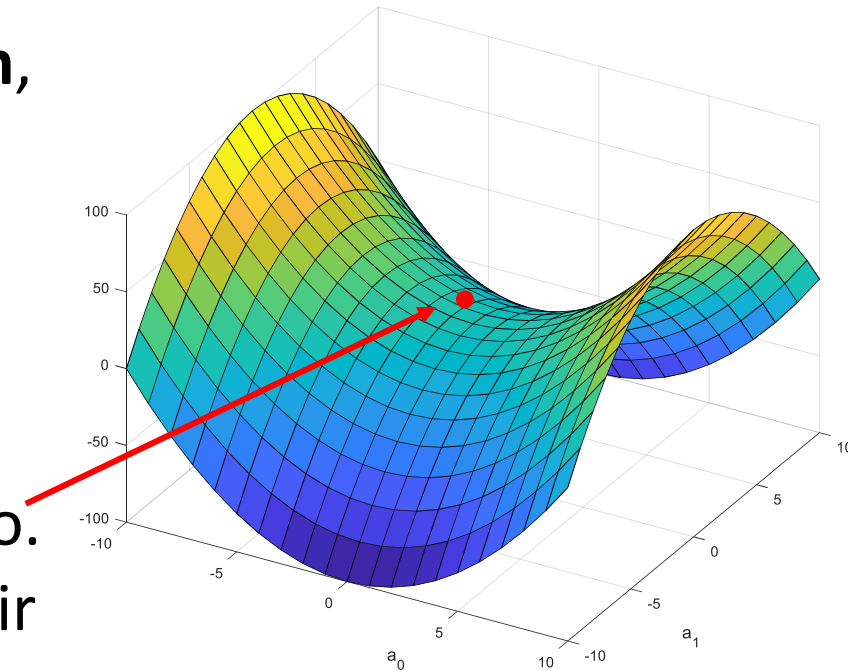
Mínimos Locais, Mínimos Globais e Pontos de Sela

- É importante ressaltarmos que todos esses métodos são métodos de **busca local**, ou seja, eles têm convergência assegurada para **mínimos locais**.
- Para lembrarmos o que é um mínimo local, vejamos a figura ao lado onde existem dois mínimos:
 - 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 (**mínimo local**), mas também em relação a todo o domínio considerado. Este é um **mínimo global**.



Mínimos Locais, Mínimos Globais e Pontos de Sela

- Para valores adequados do **passo de aprendizagem**, α , um **mínimo local** tende a atrair o vetor de pesos quando este se encontra em sua vizinhança.
- De maneira mais rigorosa, podemos dizer que cada mínimo tem sua **bacia de atração**.
- Outro ponto que deve ser mencionado são 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 de tempo sendo atraído por eles, o que prejudica seu desempenho.
- A figura ao lado mostra um exemplo de um ponto de sela.



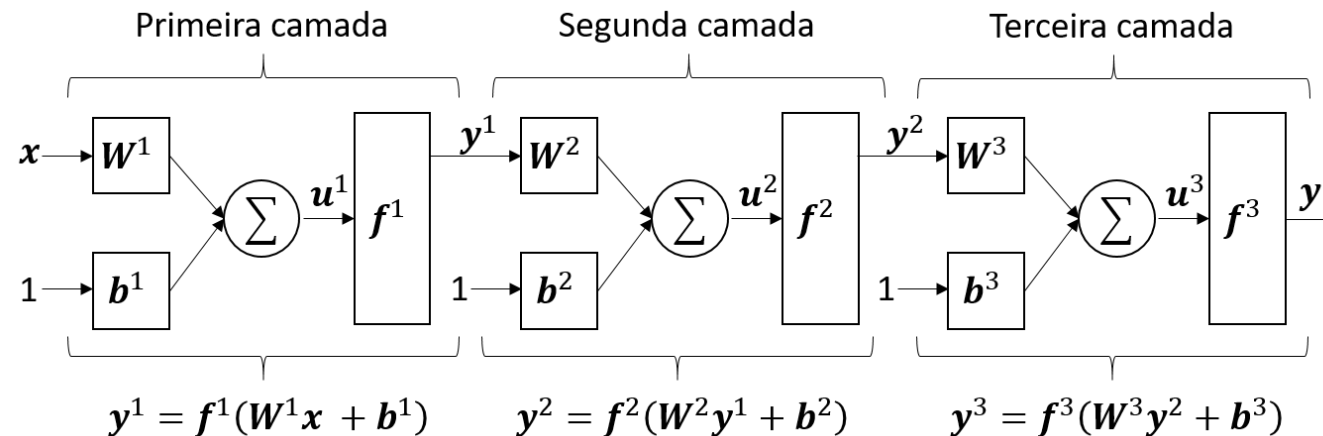
Ponto de sela: um ponto que é um mínimo ao longo de um eixo mas um máximo ao longo de outro eixo.

Retropropagação do Erro (Error Backpropagation)

- Conforme nós discutimos anteriormente, os métodos fundamentais de **aprendizado** em **redes neurais** são baseados no cálculo de derivadas parciais da **função custo** com relação aos **pesos sinápticos**.
- Esses métodos buscam, fundamentalmente, encontrar o **conjunto de pesos** que minimize a **medida de erro** escolhida.
- A ideia é encontrar uma maneira de calcular o **vetor gradiente** da **função de custo** com respeito aos **pesos sinápticos**.
- Essa tarefa pode parecer óbvia, mas não é o caso. Foram necessários 17 anos desde a criação do Perceptron para se “descobrirem” uma forma de treinar as RNAs.
- Para que entendamos melhor o porquê, nós iremos considerar uma notação que será muito útil a seguir:
 - O peso $w_{i,j}^m$ corresponde ao j -ésimo peso do i -ésimo **nó** (ou **neurônio**) da m -ésima camada da **rede neural** e W^m é a matriz com todos os pesos da m -ésima camada.
 - $f^m(\cdot)$ é a função de ativação da m -ésima camada da **rede neural**.
 - Com essa notação, obter o **vetor gradiente** significa calcular, de maneira genérica, $\frac{\partial J(\mathbf{w})}{\partial w_{i,j}^m}$, ou seja, calcular essa derivada para todos os pesos de todos os **nós**.

Retropropagação do Erro (Error Backpropagation)

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



- O mapeamento realizado pela rede MLP acima é dado por:

$$y^3 = f^3(W^3 f^2(W^2 f^1(W^1 x + b^1) + b^2) + b^3).$$

- 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 (Error Backpropagation)

- Nós vamos assumir que a última camada da rede MLP (denotada como a M -ésima camada) tenha uma quantidade genérica, N_M , de **nós**.

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

onde $d_j(n)$ é o valor desejado da j -ésima saída (rótulo) correspondente ao n -ésimo exemplo de entrada.

- Devemos derivar a **função custo** com respeito aos **pesos**, mas estes não aparecem de maneira explícita na expressão de $J(\mathbf{w})$.
- Para fazer com que a dependência dos pesos apareça de maneira clara na expressão acima, nós precisamos recorrer a aplicações sucessivas da **regra da cadeia**.
- Usando a notação de **Leibniz**, essa regra nos mostra que:

$$\frac{df(g(x))}{dx} = \frac{df(g(x))}{dg(x)} \frac{dg(x)}{dx}.$$

- Por exemplo, considerem que temos $f(g(x)) = e^{x^2}$ e queremos obter $\frac{df(g(x))}{dx}$. Nós podemos fazer $g(x) = x^2$ e usar a regra da cadeia:

$$\frac{df(g(x))}{dx} = \frac{df(g(x))}{dg(x)} \frac{dg(x)}{dx} = e^{g(x)} 2x = 2x e^{x^2}.$$

Retropropagação do Erro (Error Backpropagation)

- Agora voltamos à equação do MSE e vemos que as saídas da última camada aparecem de maneira direta.
- Isso significa que é simples se obter as derivadas com respeito aos pesos da camada de saída.
- Porém, quando se busca avaliar as derivadas com respeito aos pesos das camadas anteriores, a situação fica mais complexa, pois não existe uma dependência direta.
- Como podemos atribuir a cada **nó** de uma camada anterior sua devida influência na composição da saída e do erro?

Retropropagação do Erro (Error Backpropagation)

- Essa “caminhada de trás para a frente”, da saída (na qual se calcula o erro) para a entrada, tendo por base a **regra da cadeia**, corresponde ao processo conhecido como **retropropagação do erro** (**error backpropagation**, ou simplesmente **backpropagation**).
- A seguir, nós 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:

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

OBS.1: Operação da derivada parcial é distributiva.

OBS.2: A divisão pelo número de amostras foi omitida pois isso não afeta a otimização.

- A equação acima mostra que é necessário calcular a expressão do gradiente apenas para o n -ésimo dado, pois o gradiente médio será uma média de **gradientes particulares** (ou **locais**) associados a cada amostra.

Retropropagação: Algumas noções básicas

- Considerando novamente a derivada geral $\frac{\partial J}{\partial w_{i,j}^m}$ (i.e., um elemento genérico do gradiente). Usando a **regra da cadeia**, podemos escrever o seguinte:

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

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

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

- O termo δ_i^m é único para cada **nó**.
- O outro termo, por sua vez, varia ao longo das entradas do **nó** em questão. Como adotamos o modelo do **tipo perceptron**, a ativação 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

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

- 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 com respeito aos pesos sinápticos são produtos de um valor delta, δ_i^m , por uma entrada (ou, no caso dos termos de bias, pela unidade).***

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

ou

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

- São os valores de **delta**, 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).

Retropropagando o erro

- Portanto, a estratégia de otimização adotada é a seguinte: começa-se pela saída (onde o erro é gerado) e encontra-se uma **regra recursiva** que gere os valores de **delta** para os **nós** das camadas anteriores até a primeira camada intermediária. Esse processo é chamado de **retropropagação do erro**.
- Para facilitar a **retropropagação do erro**, nós vamos inicialmente agrupar todos os valores δ_i^m de uma camada em um vetor δ^m .
- Em seguida, vamos encontrar uma regra que fará a transição $\delta^m \rightarrow \delta^{m-1}$.
- Ao final, iremos calcular o vetor da última camada δ^M e, de maneira recursiva, vamos obter os vetores de todas as camadas e portanto, esse é o processo conhecido como **retropropagação** (ou **backpropagation**).
- Para calcular δ^M nós iremos considerar N_M saídas e assim, temos que o i -ésimo elemento é dado por:

$$\delta_i^M = \frac{\partial e_j^2}{\partial u_i^M} = \frac{\partial (d_j - y_j^M)^2}{\partial u_i^M} = -2(d_j - y_j^M) \frac{\partial y_j^M}{\partial u_i^M} = -2(d_j - y_j^M) f'^M(u_i^M)$$

Retropropagando o erro

- Matricialmente podemos expressar δ^M como:

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

onde a matriz $\mathbf{F}'^M(\mathbf{u}^M)$ é definida como

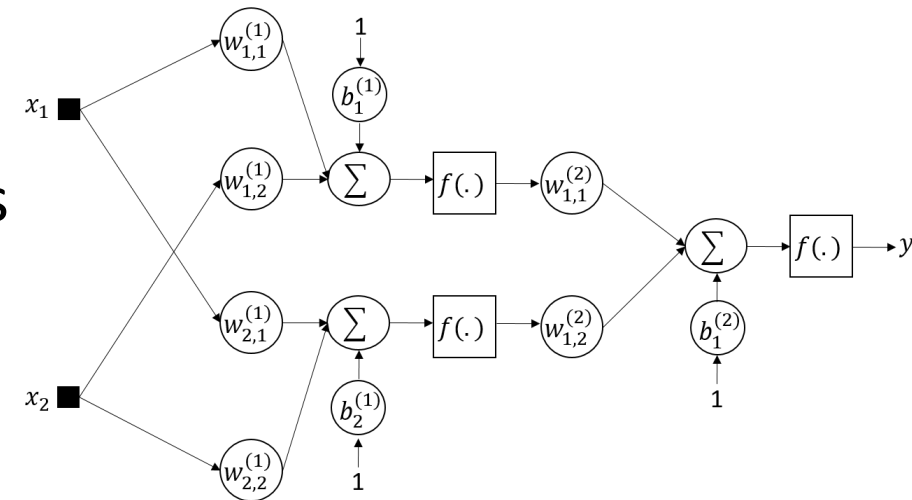
$$\mathbf{F}'^M(\mathbf{u}^M) = \begin{bmatrix} f'^M(u_1^M) & 0 & \cdots & 0 \\ 0 & f'^M(u_2^M) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f'^M(u_{N_M}^M) \end{bmatrix}.$$

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

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

Exemplo da retropropagação do erro

- Considere uma rede MLP com uma camada intermediária e apenas um **nó** na camada de saída, como a mostrada na figura ao lado.
- Temos neste exemplo $M = 2$.
- Devemos começar calculando δ^2 . Perceba que essa **sensibilidade** é um escalar porque há apenas um **nó** na camada de saída.
- Vamos considerar um único dado com entrada $\mathbf{x} = [x_1, x_2]$ e saída desejada d .
- Inicialmente, supomos que a rede terá uma certa configuração de pesos, de modo que, quando a entrada for apresentada à rede, será possível calcular todos os sinais pertinentes 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 o delta 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 recursão para **retropropagar** o erro até a camada anterior. A fórmula nos diz:

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

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

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

OBS.: Note que o $.^2$ 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,

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

- Observem que, para calcular o gradiente, não basta apenas calcular os **deltas**: é necessário multiplicá-los pelas entradas correspondentes (observando que os **bias** estão ligados a entradas com valores constantes iguais a 1). As derivadas parciais com relação aos pesos do **nó** $i = 1$ da camada $m = 1$ são mostrados abaixo.

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

Exemplo da retropropagação do erro

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

$$\begin{aligned}\frac{\partial J}{\partial w_{1,1}^1} &= \frac{\partial e^2}{\partial w_{1,1}^1} = \frac{\partial \left(d - f^2(u_1^2) \right)^2}{\partial f^2(u_1^2)} \frac{\partial f^2(u_1^2)}{\partial u_1^2} \frac{\partial u_1^2}{\partial f^1(u_1^1)} \frac{\partial f^1(u_1^1)}{\partial u_1^1} \frac{\partial u_1^1}{\partial w_{1,1}^1} \\ \frac{\partial J}{\partial w_{1,2}^1} &= \frac{\partial e^2}{\partial w_{1,2}^1} = \frac{\partial \left(d - f^2(u_1^2) \right)^2}{\partial f^2(u_1^2)} \frac{\partial f^2(u_1^2)}{\partial u_1^2} \frac{\partial u_1^2}{\partial f^1(u_1^1)} \frac{\partial f^1(u_1^1)}{\partial u_1^1} \frac{\partial u_1^1}{\partial w_{1,2}^1} \\ \frac{\partial J}{\partial b_1^1} &= \frac{\partial e^2}{\partial b_1^1} = \frac{\partial \left(d - f^2(u_1^2) \right)^2}{\partial f^2(u_1^2)} \frac{\partial f^2(u_1^2)}{\partial u_1^2} \frac{\partial u_1^2}{\partial f^1(u_1^1)} \frac{\partial f^1(u_1^1)}{\partial u_1^1} \frac{\partial u_1^1}{\partial b_1^1}\end{aligned}$$

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 - **Estimação: Online, Batch e Minibatch**

- Conforme vimos nos slides anteriores, 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*** que minimizem a ***função de custo***.
- Vimos que a obtenção do ***vetor gradiente*** se dá através de um processo de ***retropropagação*** em que há uma parte direta (***forward***) de apresentação de um exemplo e obtenção da resposta da rede e uma etapa de ***retropropagação*** em que se calculam as derivadas parciais necessárias.

Algumas visões práticas de algoritmos de aprendizado - **Estimação: Online, Batch e Minibatch**

- Vimos também que se calcula o gradiente associado a cada dado de entrada e que a combinação de todos esses ***gradientes locais*** leva ao gradiente estimado para o conjunto de dados inteiro.

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

- No entanto, surge aqui um questionamento interessante: o que é melhor, usar o gradiente local e já dar um passo de otimização ou reunir o gradiente completo e então dar um passo único e mais preciso?

Algumas visões práticas de algoritmos de aprendizado - **Estimação: Online, Batch e Minibatch**

- Nesse questionamento, existem duas abordagens: o cálculo **online** do gradiente (exemplo-a-exemplo) e o cálculo em batelada (**batch**) do gradiente.
- Vejamos inicialmente a noção geral de **adaptação dos pesos sinápticos** com cálculo **online** do gradiente, como expressa o seguinte algoritmo, um método clássico de **primeira ordem**.

- Defina valores iniciais para o vetor de pesos \mathbf{w} e um passo de aprendizagem α pequeno.
- Faça $k = 0, t = 0$ e calcule $J(\mathbf{w}(k))$.
- Enquanto o critério de parada não for atendido, faça:
 - Ordene aleatoriamente os exemplos de entrada/saída.
 - Para l variando de 1 até N , faça:
 - Apresente o exemplo l de entrada à rede.
 - Calcule $J_l(\mathbf{w}(t))$ e $\nabla J_l(\mathbf{w}(t))$.
 - $\mathbf{w}(t + 1) = \mathbf{w}(t) - \alpha \nabla J_l(\mathbf{w}(t)); t = t + 1$.
 - $k = k + 1$.
 - Calcule $J(\mathbf{w}(k))$.

Algumas visões práticas de algoritmos de aprendizado - **Estimação: Online, Batch e Minibatch**

- O outro extremo seria utilizar todo o conjunto de dados para estimar o gradiente antes de dar o passo do processo iterativo de aprendizagem.
- Essa é a ideia por trás da abordagem em **batelada (batch)**. O algoritmo abaixo ilustra a operação correspondente (novamente considerando uma metodologia de **primeira ordem**).

- Defina valores iniciais para o vetor de pesos \mathbf{w} e um passo de aprendizagem α 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 exemplo l de entrada à rede.
 - Calcule $J_l(\mathbf{w}(k))$ e $\nabla J_l(\mathbf{w}(k))$.
 - $\mathbf{w}(k + 1) = \mathbf{w}(k) - \frac{\alpha}{N} \sum_{l=1}^N \nabla J_l(\mathbf{w}(k))$.
 - $k = k + 1$.
 - Calcule $J(\mathbf{w}(k))$.

Algumas visões práticas de algoritmos de aprendizado - Estimação: Online, Batch e Minibatch

- Nas modernas **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**).
- As amostras que devem compor o **mini-batch** são **aleatoriamente** tomadas do conjunto de dados. O algoritmo abaixo ilustra isso.

- Defina valores iniciais para o vetor de pesos \mathbf{w} e um passo de aprendizagem α 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 m , faça:
 - Apresente o exemplo l de entrada, amostrado aleatoriamente para compor um **minibatch**, à rede.
 - Calcule $J_l(\mathbf{w}(k))$ e $\nabla J_l(\mathbf{w}(k))$.
 - $\mathbf{w}(k + 1) = \mathbf{w}(k) - \frac{\alpha}{m} \sum_{l=1}^m \nabla J_l(\mathbf{w}(k))$.
 - $k = k + 1$.
 - Calcule $J(\mathbf{w}(k))$.

Variações dos algoritmos de otimização dos pesos: **Método do Gradiente Estocástico**

- Existem vários algoritmos baseados no ***gradiente*** que podem ser empregados para otimizar os ***pesos sinápticos*** de uma rede neural.
- Aqui, vamos nos ater a alguns métodos muito usuais na literatura moderna, que se encontra bastante focada em ***apredizado profundo***.
- **Método do Gradiente Estocástico (Stochastic Gradient Descent, SGD)**
 - Nos slides anteriores, nós vimos que o método ***online*** utiliza um único exemplo (tomado aleatoriamente) para estimar o gradiente da ***função custo***.
 - Este tipo de estimador é o que gera a noção de ***gradiente estocástico***. Caso utilizemos ***mini-batches***, também teremos uma estimativa do ***gradiente***, o qual, a rigor, seria determinístico apenas se usássemos todos os dados (no caso do ***batch***).
 - Por esse motivo, esses métodos de ***primeira ordem***, como ***online***, são conhecidos como métodos de ***stochastic gradient descent*** (SGD).

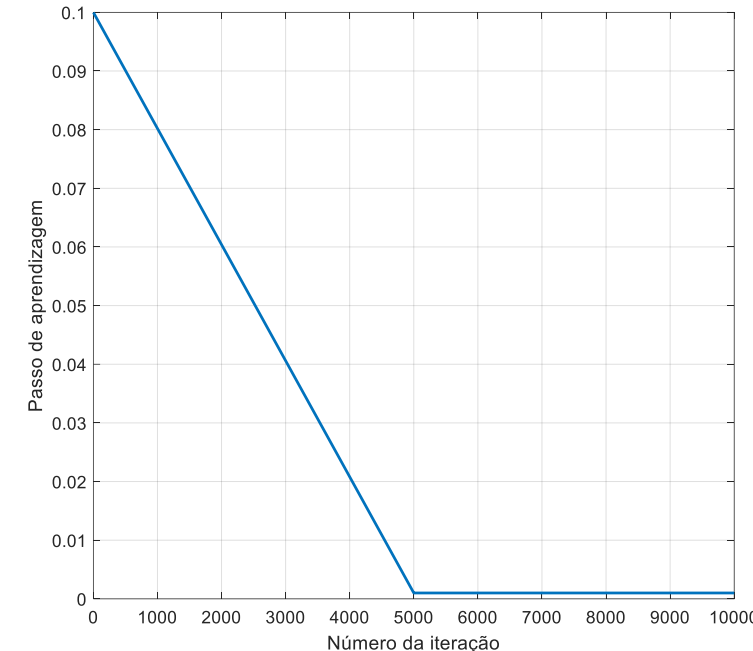
Variações dos algoritmos de otimização dos pesos: **Método do Gradiente Estocástico**

- A tarefa de escolha do ***passo de aprendizagem*** é complicada e nos remete ao conhecido compromisso entre velocidade de convergência e estabilidade/precisão.
- Pode-se usar um valor fixo, mas geralmente, se adota um método de variação linear decrescente de um valor α_0 a um valor α_τ (i.e., da iteração 0 à iteração τ):

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

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.
- Naturalmente, a definição dos valores necessários (i.e., α_0 e α_τ) é mais um problema ***a ser tratado caso-a-caso***.



- $N = 10000$
- Tamanho do batch: 100
- Número de épocas: 100
- $\alpha_0 = 0.1$
- $\alpha_\tau = \left(\frac{1}{100}\right)\alpha_0$
- $\tau = 5000$

Variações dos algoritmos de otimização dos pesos: **Momento**

➤ **Momento** (ou **Momentum**)

- O uso de um ***termo de momento*** numa metodologia de gradiente descendente pode ser interessante por trazer, para o ***ajuste de pesos*** em determinada iteração, ***informação de gradientes anteriores acumulados***. Isso, em certas situações, melhora a convergência.
- Para discutirmos o algoritmo do ***momentum***, vamos partir de um esquema de aprendizado em mini-batch.
- Seja ***g*** o ***vetor gradiente*** calculado para o mini-batch e ***v*** um ***termo de velocidade*** introduzido pelo algoritmo do ***momentum***.
- O termo ***v*** dá a ***direção*** e a ***velocidade*** na qual os pesos se movem pelo espaço de pesos.
- A ***velocidade*** é atualizada da seguinte forma:

$$v \leftarrow \varphi v - \alpha g.$$

- Momentum em física é igual a massa de uma partícula vezes sua velocidade. No algoritmo do momentum, assumimos que a massa é unitária, então o vetor velocidade ***v*** também pode ser considerado como o momentum da partícula.

Variações dos algoritmos de otimização dos pesos: **Momento**

➤ **Momento** (ou **Momentum**)

- O **hiperparâmetro** φ (*phi*) $\in [0,1)$ determina com que rapidez as contribuições de gradientes anteriores decaem exponencialmente (ou seja, φ é um termo de memória).
- A **atualização dos pesos** é dada por
$$\mathbf{w} \leftarrow \mathbf{w} + \mathbf{v}.$$
- O efeito do **termo momentum** pode ser visto como um valor que se acumula de acordo com a regra de uma **progressão geométrica**.
- Portanto, podemos pensar em seu efeito de aceleração no sentido contrário do gradiente em termos da equação $\frac{1}{1-\varphi}$. Por exemplo, $\varphi = 0.9$ corresponde à multiplicação da velocidade por 10 em relação ao algoritmo do gradiente descendente.
- Valores típicos de φ são 0.5, 0.9 e 0.99.
- Assim como a taxa de aprendizagem, φ também pode ser adaptado ao longo do tempo. Normalmente, ele começa com um valor grande e é diminuído posteriormente.

Variações dos algoritmos de otimização dos pesos:

Momento de Nesterov e Passo de Aprendizado Adaptativo

➤ Momento de Nesterov

- O método do ***momento de Nesterov*** pode ser visto, essencialmente, como uma variação do ***método do momento*** em que o cálculo do ***vetor gradiente*** não é feito sobre o vetor de pesos w , mas sim sobre $w + \varphi v$.
- Esse termo adicional funciona como um fator de correção que pode beneficiar, em alguns casos, a velocidade de convergência.

➤ Modelos com Passo de Aprendizagem Adaptativo

- Como discutimos anteriormente, o ***passo de aprendizagem*** é um hiperparâmetro difícil de se ajustar otimamente e bastante relevante para o sucesso do treinamento de uma rede neural.
- Isso motivou o surgimento de um conjunto de métodos com mecanismos capazes de modificá-lo dinamicamente. Dentre as técnicas mais populares dessa classe estão o ***AdaGrad***, o ***RMSProp*** e o ***Adam*** (de “adaptive moments”).

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 e falha completamente em convergir.
- Também pode haver variações expressivas na **velocidade de convergência**.
- Um ponto importante da inicialização é “**quebrar a simetria**” entre os **nós**, ou seja, se dois **nós** ocultos (i.e., **nós** de camadas ocultas) com a mesma **função de ativação** estiverem conectados às mesmas entradas, esses **nós** deverão ter pesos iniciais diferentes.
- Isso, portanto, sugere uma **abordagem aleatória**.

Inicialização dos Pesos

- Os pesos tipicamente são obtidos de ***distribuições gaussianas*** ou ***uniformes***.
- A ordem de grandeza desses pesos levanta algumas discussões:
 - Pesos de maior magnitude criam 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.
 - Pesos de magnitude elevada podem levar os ***nós*** (no caso de ***funções de ativação*** do tipo sigmoide como a tangente hiperbólica e a função logística) a operarem numa região de saturação, comprometendo a convergência do algoritmo.

Inicialização dos Pesos

- Portanto, podemos citar algumas **heurísticas** para inicializar os pesos.
- Uma primeira seria, para uma camada com m entradas e n saídas, inicializar os pesos com valores retirados da seguinte distribuição:

$$w_{i,j} \sim U\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right),$$

onde $U(.)$ é a **distribuição uniforme**.

- Outra heurística de inicialização dos pesos seria:

$$w_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right).$$

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

Redes Neurais MLP com SciKit-Learn

- A biblioteca SciKit-Learn disponibiliza algumas classes para o treinamento de redes neurais multi-layer perceptron.
- Entretanto, as implementações desta biblioteca não se destinam a aplicações de larga escala.
- Em particular, a biblioteca scikit-learn não oferece suporte a GPUs.
- Para implementações muito mais rápidas, baseadas em GPU, bem como estruturas que oferecem muito mais flexibilidade para criar arquiteturas de aprendizado profundo, por exemplo, devemos utilizar outras bibliotecas como:
 - **Tensorflow**: biblioteca para desenvolvimento de aplicações eficientes e escaláveis de machine learning.
 - **keras**: uma biblioteca para desenvolvimento de aplicações Deep Learning capaz de rodar sobre o TensorFlow ou o Theano.
 - **skorch**: uma biblioteca de rede neural compatível com o scikit-learn que encapsula a biblioteca PyTorch.
 - Entre outras: https://scikit-learn.org/stable/related_projects.html#related-projects

Detecção de símbolos QPSK com MLPClassifier

```
from sklearn.neural_network import MLPClassifier ← Importa a classe MLPClassifier
import numpy as np
```

```
# Number of QPSK symbols to be transmitted.
N = 10000
```

```
# ***** Modulation *****
```

```
# Generate N binary symbols.
```

```
bits = np.random.randint(0,4,(N,1))
```

```
# Modulate the binary stream into QPSK symbols.
```

```
s = mod(bits)
```

Gera um sequência aleatória de bits para transmissão.

Modula os símbolos QPSK com os bits gerados.

```
# ***** AWGN Channel *****
```

```
# Generate noise vector.
```

```
np.random.seed(seed)
```

```
noise = np.sqrt(1.0/2.0)*(np.random.randn(N, 1) + 1j*np.random.randn(N, 1))
```

```
# Pass symbols through AWGN channel.
```

```
y = s + np.sqrt(0.2)*noise
```

Passa sinal modulado por canal AWGN.

```
# ***** Demodulation *****
```

```
# Instantiate Multi layer Perceptron Classifier.
```

```
clf = MLPClassifier(hidden_layer_sizes=(10,4), activation='logistic', solver='sgd', batch_size=50,
learning_rate='adaptive', random_state=seed, max_iter=4000)
```

```
# Split arrays into random train and test subsets.
```

```
s_test, s_train, y_test, y_train, b_test, b_train = train_test_split(s, y, bits, random_state=seed)
```

```
# SciKit-learn's MLPs do not support complex signals, then we split it into real and imag parts.
```

```
Y = np.c_[y_train.real, y_train.imag]
```

```
# Fit the MLP model.
```

```
clf.fit(Y, toOneHotEncoding(b_train))
```

```
# Prediction (detection) with trained MLP.
```

```
detected_mlp = clf.predict(np.c_[y_test.real, y_test.imag])
```

```
# Detection with optimum detector.
```

```
detected_opt = optimumDemod(np.c_[y_test.real, y_test.imag])
```

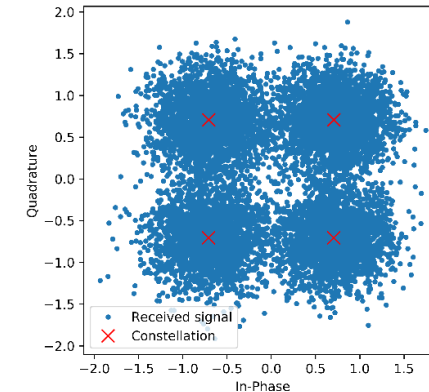
Instancia MLP com 2 camadas escondidas com 10 e 4 neurônios, respectivamente.

Divide o conjunto.

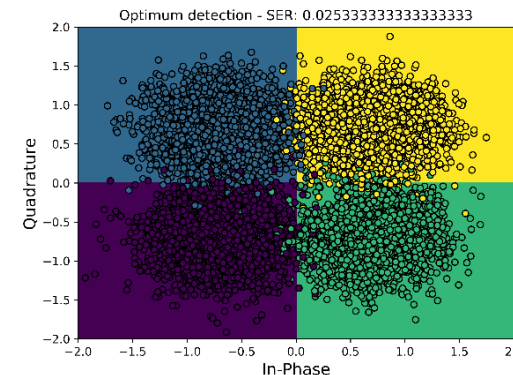
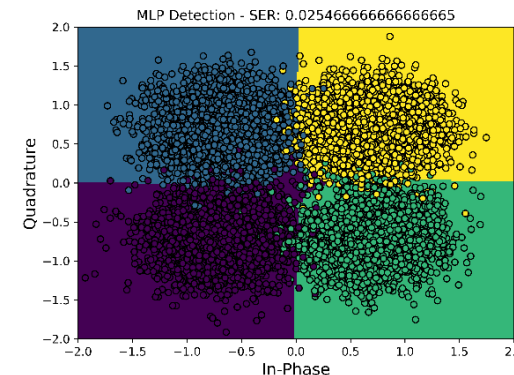
A classe MLP não suporta números complexo, portanto, dividimos y (real,imag) em 2 atributos.

Treina o modelo com codificação one-hot e faz detecção dos símbolos.

Detecção ótima dos símbolos.



$$\frac{E_s}{N_0} = 7 \text{ dB}$$



- As fronteiras de decisão do detector com classificador MLP se aproximam das fronteiras do detector ótimo.
- Qual seria a vantagem em se utilizar um detector baseado em MLP?
 - Se existe um algoritmo ótimo conhecido, uma rede neural treinada nunca poderá superá-lo.

Exemplo: [SciKitMLPQPSKClassifier.ipynb](#)

Estimação de fase com MLPRegressor

Import all necessary libraries.

import numpy as np

from sklearn.model_selection import train_test_split

from sklearn.neural_network import MLPRegressor

Importa a classe
MLPRegressor

Number of QPSK symbols to be transmitted.

N = 100000

Es/N0 = 27 dB

Define Es/N0 value in dB.

EsN0dB = 27

Transform into linear value.

EsN0Lin = 10.0**(-(EsN0dB/10.0))

Generate N binary symbols.

ip = np.random.randint(0,4,(N,1))

Modulate binary stream into QPSK symbols.

s = mod(ip)

Generate noise vector.

noise = np.sqrt(1.0/2.0)*(np.random.randn(N, 1) + 1j*np.random.randn(N, 1))

Add phase error and pass symbols through AWGN channel.

y = s*phase_rnd + np.sqrt(EsN0Lin)*noise

Phase of received signal.

theta = np.arctan(y.real/y.imag)

Gera uma sequência aleatória
de bits para transmissão.

Modula os símbolos QPSK
com os bits gerados.

Adiciona fase aleatória ao
símbolo e passa sinal
modulado por canal AWGN.

Calcula fase do símbolo
recebido.

Divide o conjunto.

Split arrays into training and validation subsets.

theta_train, theta_test, theta_orig_train, _, y_test = train_test_split(theta,

theta_orig.ravel(), y, test_size=0.2)

Instantiate MLP Regressor.

reg = MLPRegressor(hidden_layer_sizes=(10,5,4), max_iter=2000)

Instancia MLP com 3 camadas
escondidas com 10, 5 e 4
neurônios, respectivamente.

Train MLP Regressor.

reg.fit(theta_train, theta_orig_train)

Predict phase over test set.

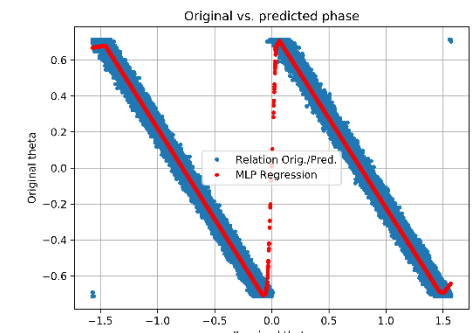
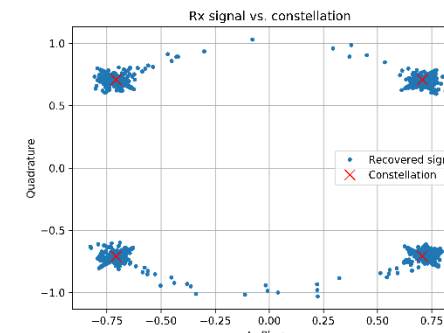
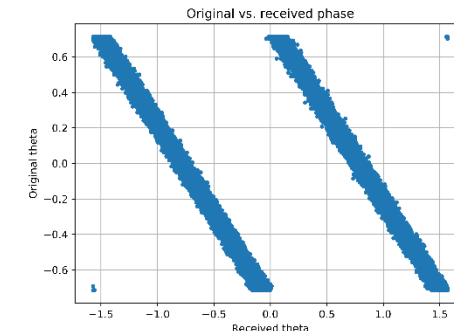
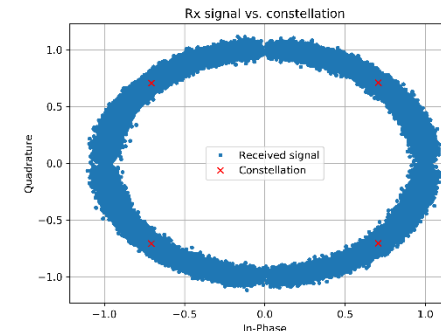
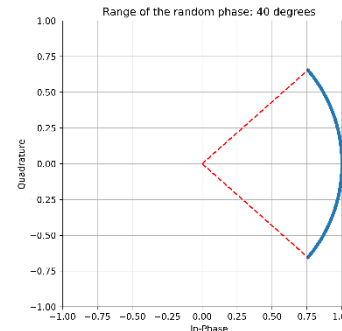
theta_pred = reg.predict(theta_test).reshape(len(theta_test), 1)

Correct phase-shift.

y_rec = np.exp(-1j*theta_pred)*y_test

Treina o modelo com fase recebida
e original e faz estimação.

Aplica inverso da fase
estimada ao símbolo recebido.



- Os símbolos QPSK tem sua fase variada por um desvio de fase aleatório.
- Fase aleatório varia entre -40 a +40 graus.
- Além disto, tem-se adição de ruído, onde a relação Es/N0 = 27 dB.
- O MLP estima a relação entre a fase do sinal recebido e a fase adicionada ao símbolo transmitido.
- De posse da relação, pode-se desfazer o efeito da fase aleatória.

Exemplo: SciKitMLPRegression_v4.ipynb

Avisos

- Material, exemplos e lista de exercícios #12 já estão disponíveis.

Obrigado!

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



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



Deep Learning



What society thinks I do



What my friends think I do



What other computer
scientists think I do



What mathematicians think I do

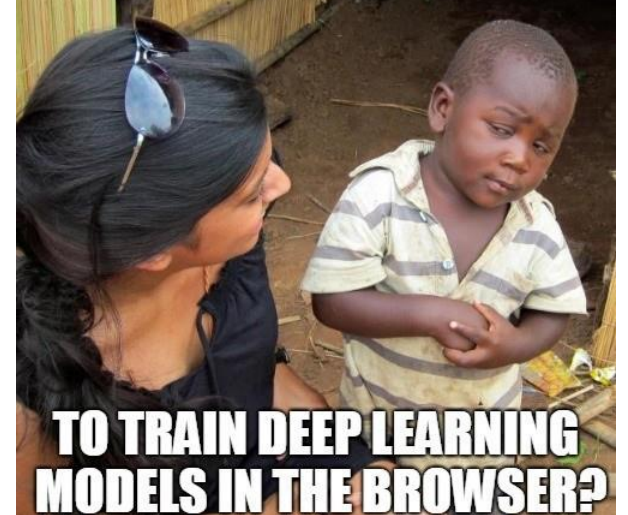


What I think I do

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

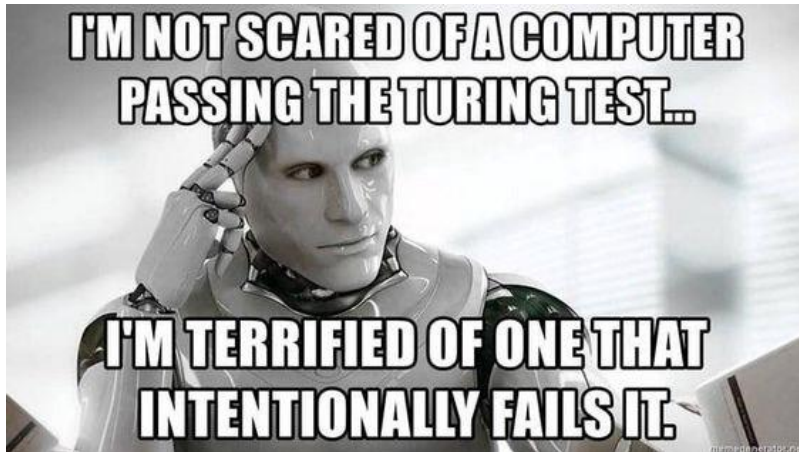
What I actually do

SO YOU ARE TELLING ME



**TO TRAIN DEEP LEARNING
MODELS IN THE BROWSER?**

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



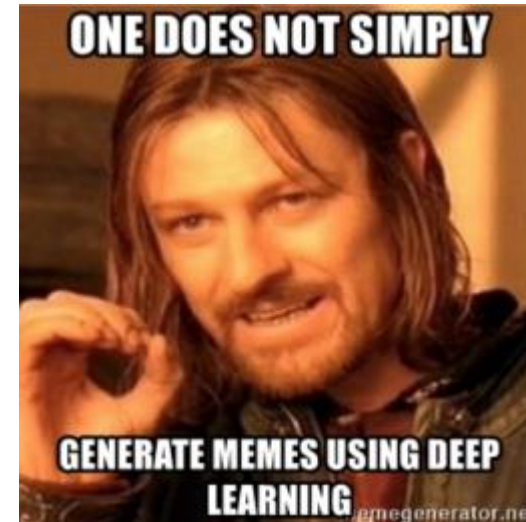
**I'M TERRIFIED OF ONE THAT
INTENTIONALLY FAILS IT.**

Dog



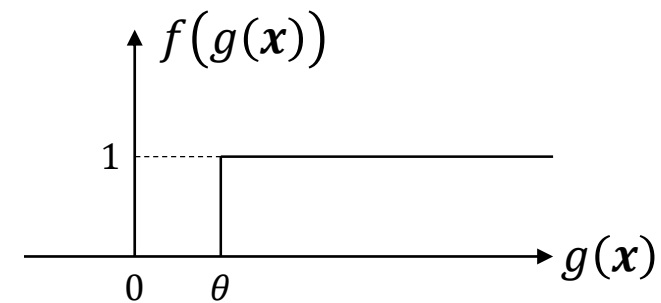
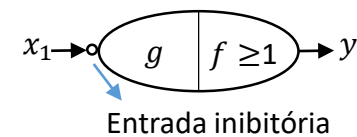
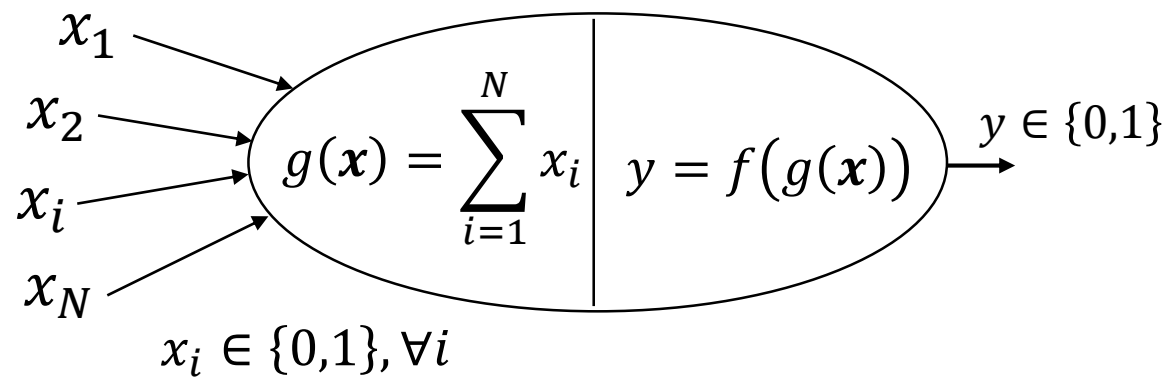
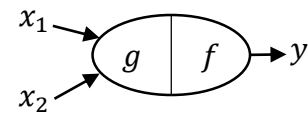
**I NEED GPU
FOR MY DUMB
NEURAL NETWORK**

ONE DOES NOT SIMPLY



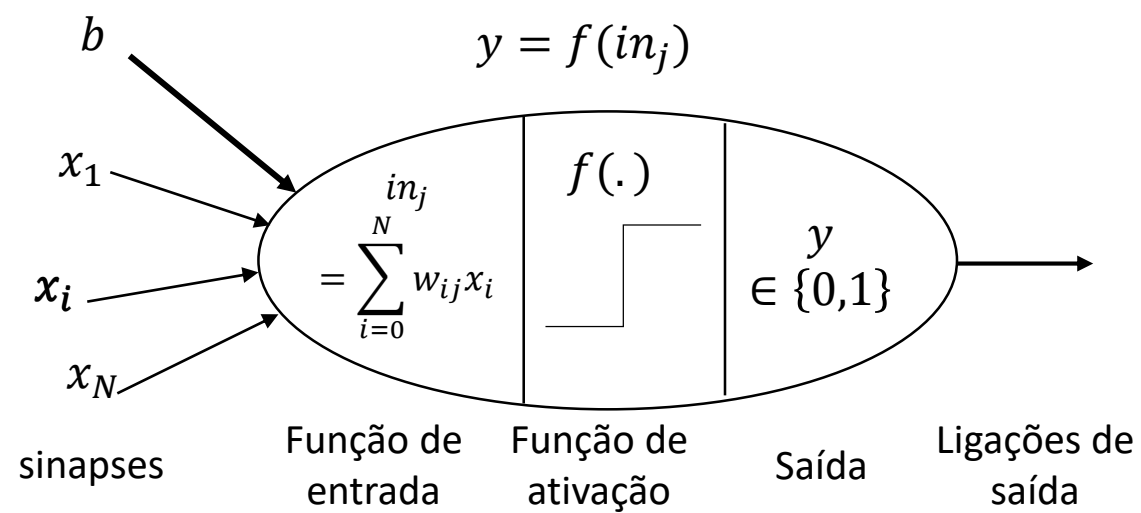
**GENERATE MEMES USING DEEP
LEARNING**

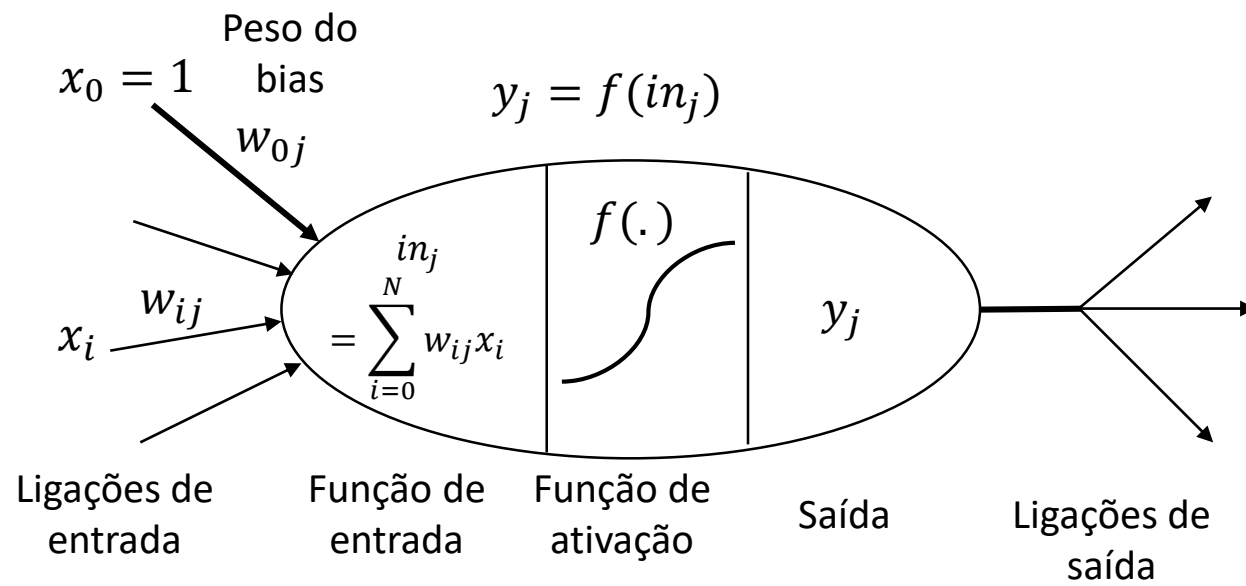
Figuras

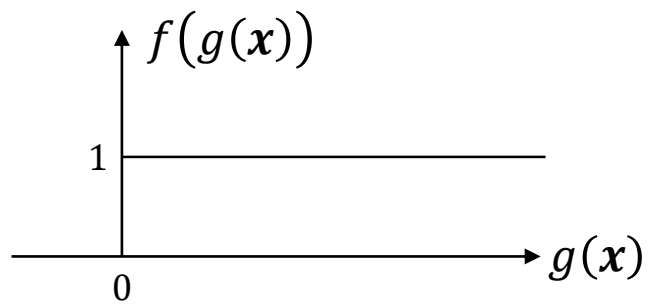
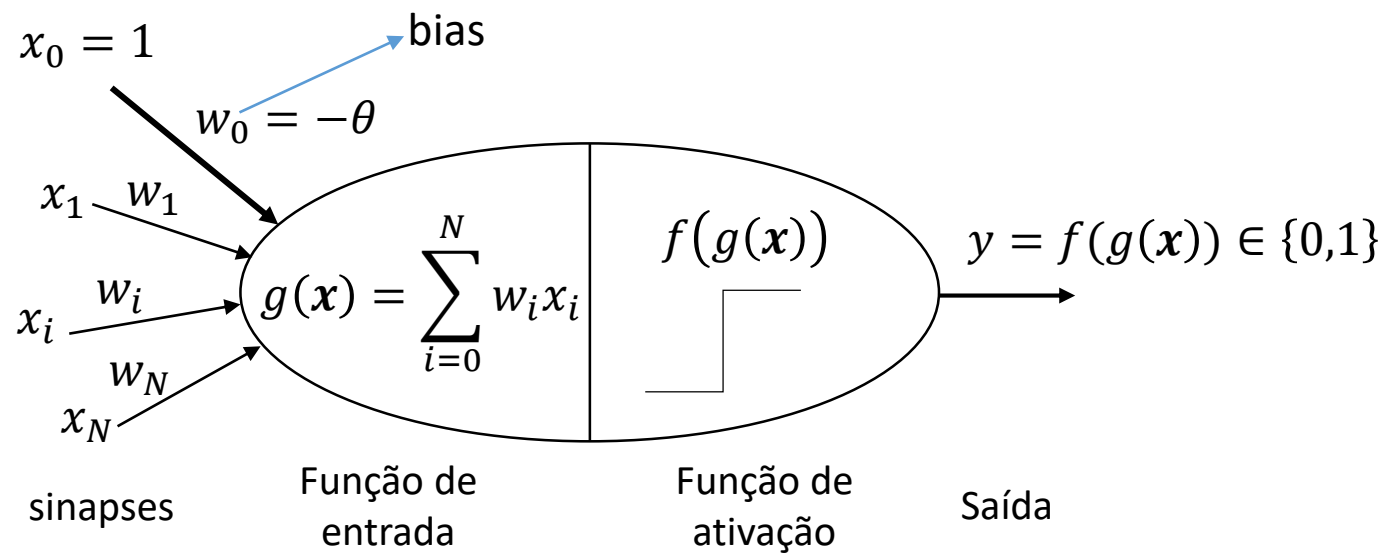


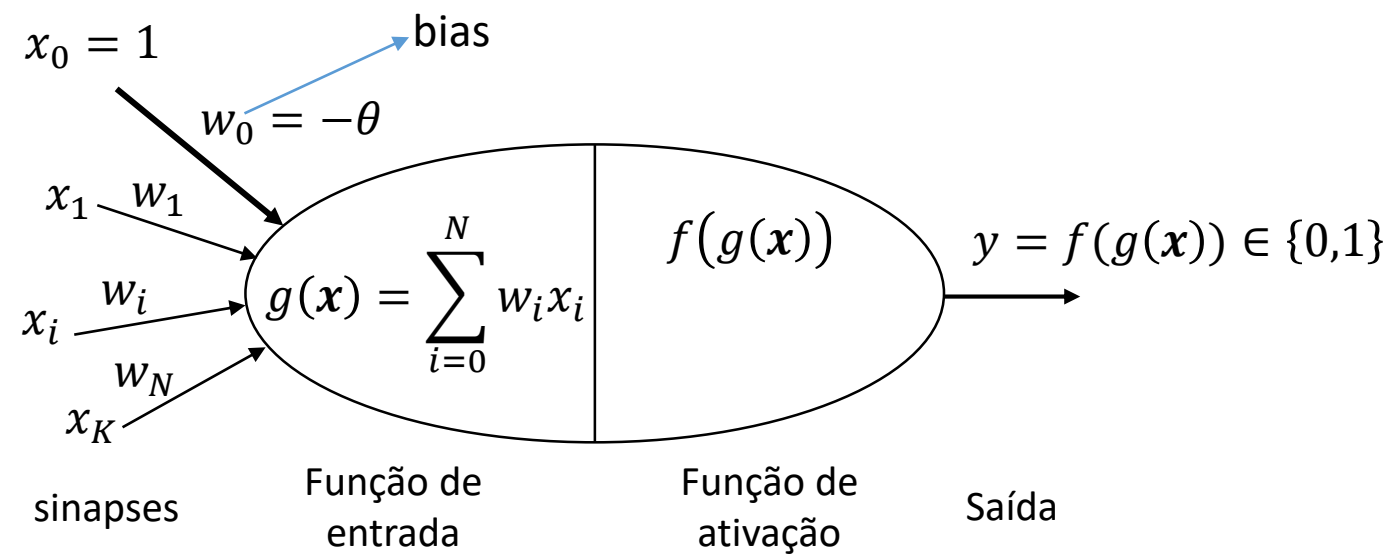
$$y = f(g(\mathbf{x})) = \begin{cases} 1 & \text{se } g(\mathbf{x}) \geq \theta \\ 0 & \text{se } g(\mathbf{x}) < \theta \end{cases}$$

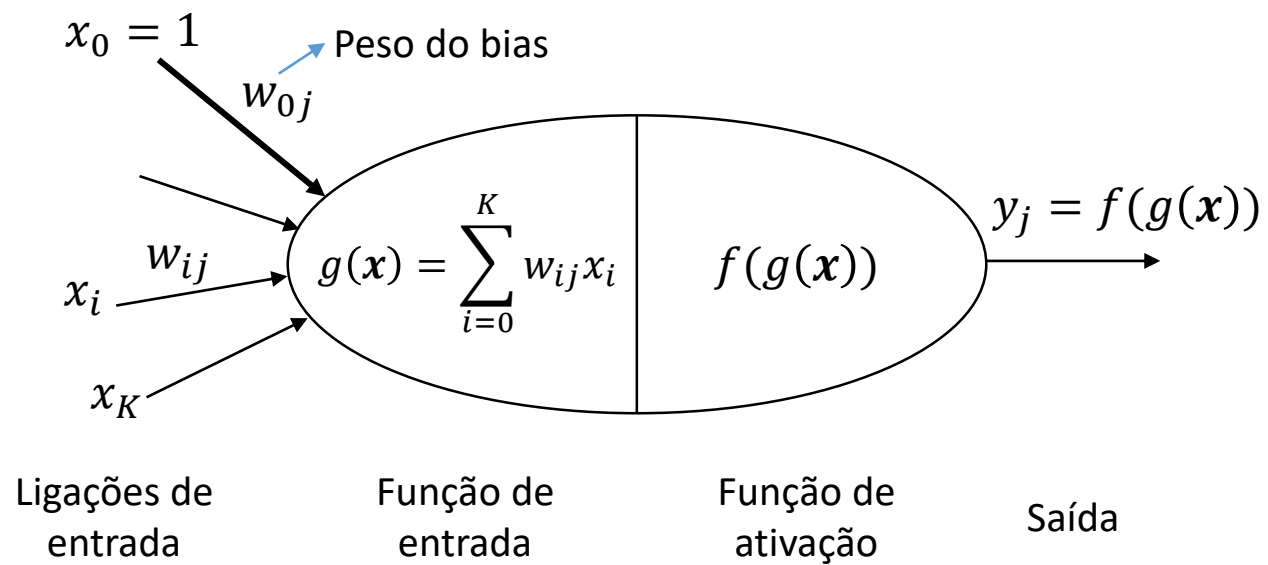
onde θ é o limiar de decisão.

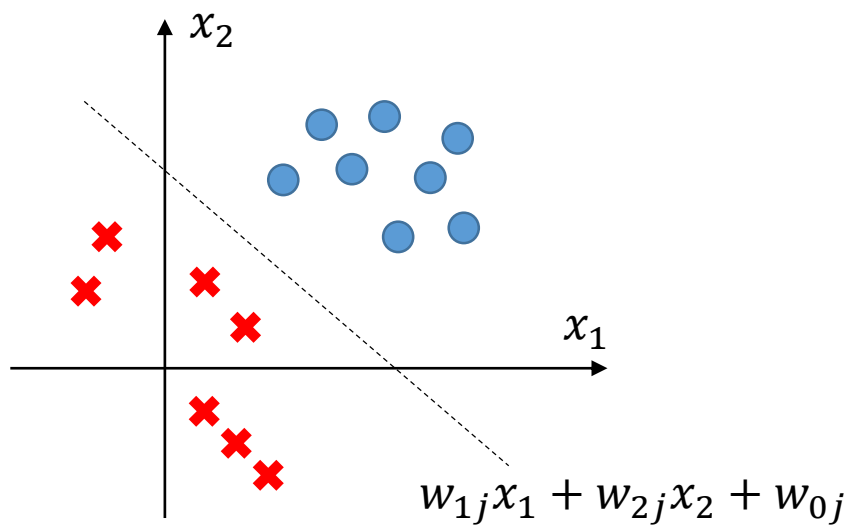


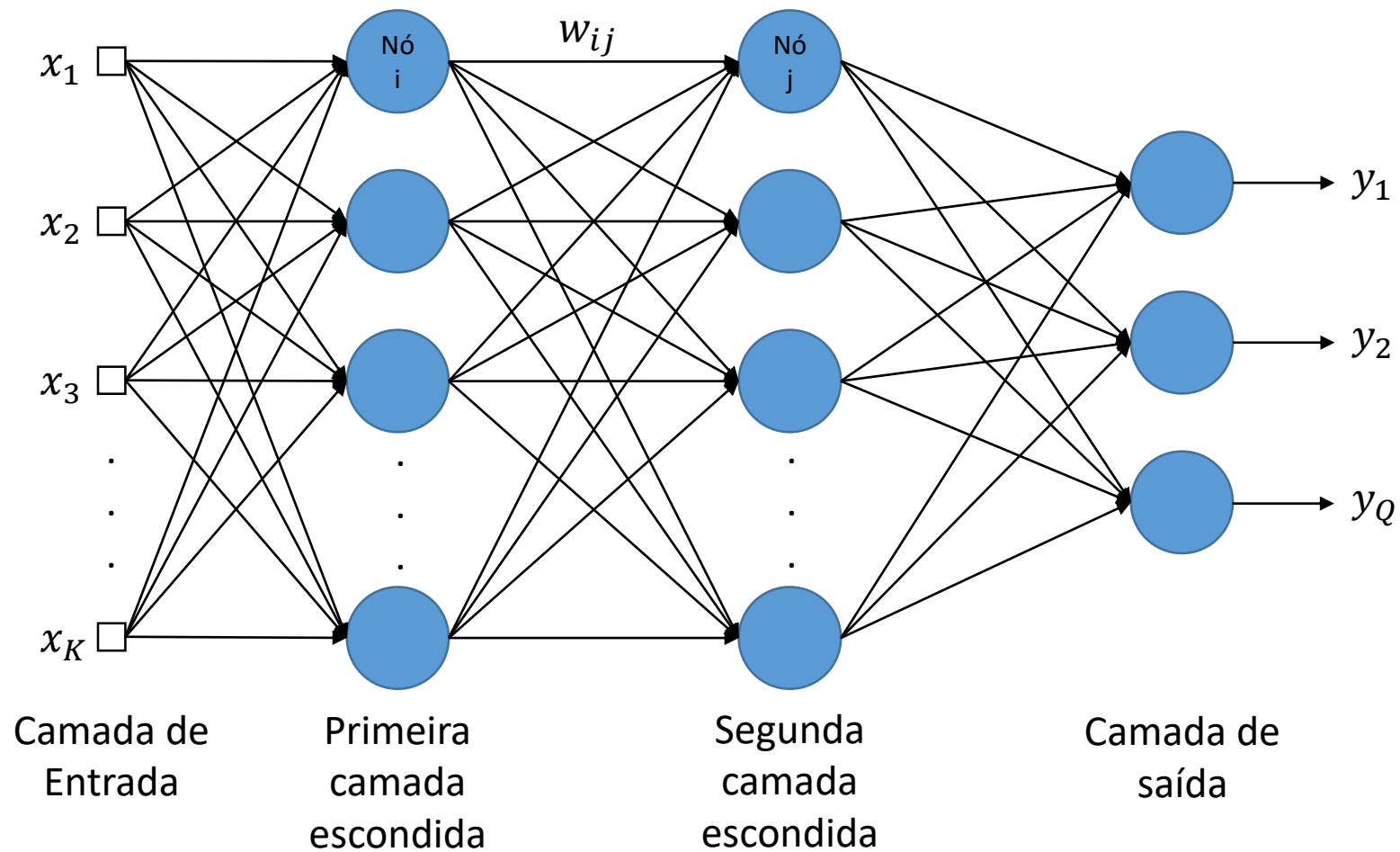





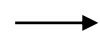








 Nó, unidade ou neurônio.

 Ligação entre i -ésimo e j -ésimo nó.

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

