Introdução à Inteligência Artificial

Módulo 04: Aprendizagem Profunda (Deep Learning - DL)

Professor: Dr. João Paulo R. R. Leite

UNIFEI Softex MCTI FUTURO FUTURO DO TRABALHO, TRABALHO DO FUTURO GOVERNO FEDERAL BRASIL UNIÃO E RECONSTRUÇÃO

Sumário do Módulo 04

Aprendizagem Profunda (Deep Learning)

UNIFEI Softex MCTI FUTURO

FUTURO DO TRABALHO, TRABALHO DO FUTURO

GOVERNO FEDERAL BRASIL UNIÃO E RECONSTRUÇÃO

Objetivo principal

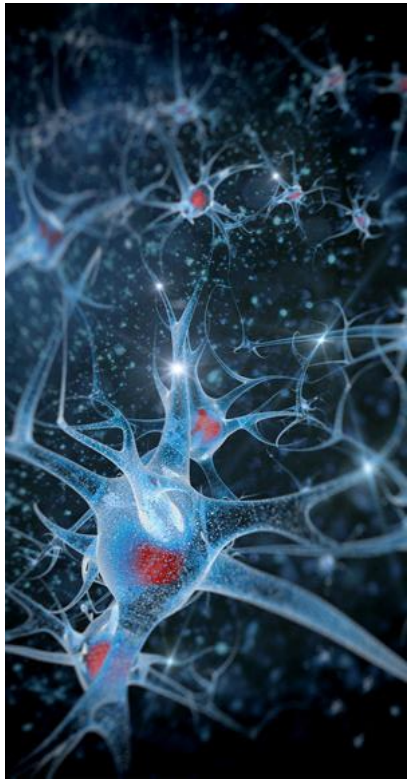
- O **objetivo** deste módulo é descrever os conceitos básicos sobre **redes neurais** e **aprendizado profundo**, incluindo seu desenvolvimento histórico, componentes de redes neurais artificiais profundas, seu método de treinamento, e soluções para problemas comuns em projetos que as utilizam.

Introdução

Deep Learning é um campo específico de *Machine Learning*, que se apresenta como um desenvolvimento moderno das redes neurais artificiais, surgidas em meados do século XX.

O termo *deep*, do inglês profundo, na verdade não tem relação com um conhecimento realmente "profundo" do sistema acerca de algum assunto, mas simplesmente reflete uma quantidade grande de camadas incluídas no modelo neural. **Quanto mais camadas, mais profundo**.

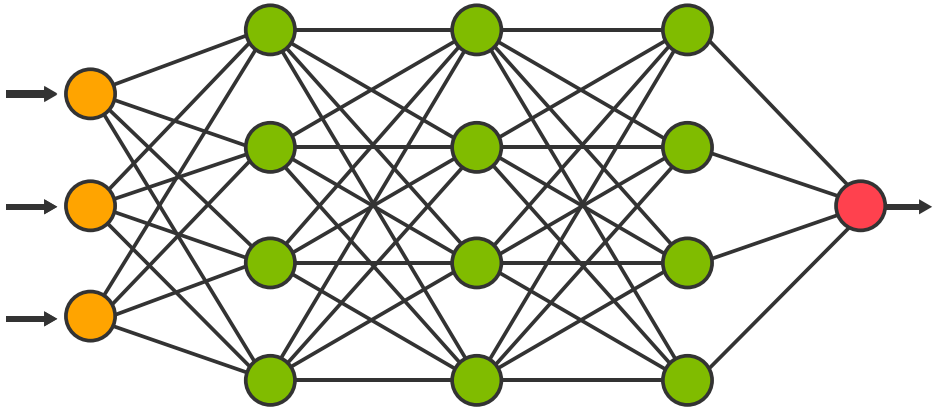
O termo "rede neural" é uma referência à neurobiologia, uma vez que esses sistemas são geralmente **inspirados em sistemas biológicos**, como o próprio cérebro humano. Não são modelos do cérebro, mas tem inspiração no processamento paralelo, distribuído e multicamadas do cérebro.



Introdução

Uma rede neural artificial pode ser definida como um sistema computacional composto de elementos processadores simples e **altamente interconectados**, que processam informação através de uma resposta dinâmica a entradas externas (estímulos).

Como são “emprestados” muitos termos da biologia, podemos dizer que a rede neural artificial é formada por neurônios artificiais conectados uns aos outros, visando simular a inteligência humana e refletir funções humanas básicas como **aprendizado**, **associação**, **classificação** de objetos e processos de **memória**. As conexões entre neurônios podem também ser chamadas de sinapses. Veja um exemplo:



História

A história do desenvolvimento das redes neurais artificiais (RNA) tem muitos **altos e baixos**, alternando momentos de empolgação e otimismo com outros em que limitações técnicas frearam o desenvolvimento.

1943: RNA foram introduzidas como modelos computacionais pela primeira vez pelos cientistas Warren McCulloch (neurofisiologista) e pelo Walter Pitts (matemático). Modelo McCulloch-Pitts.

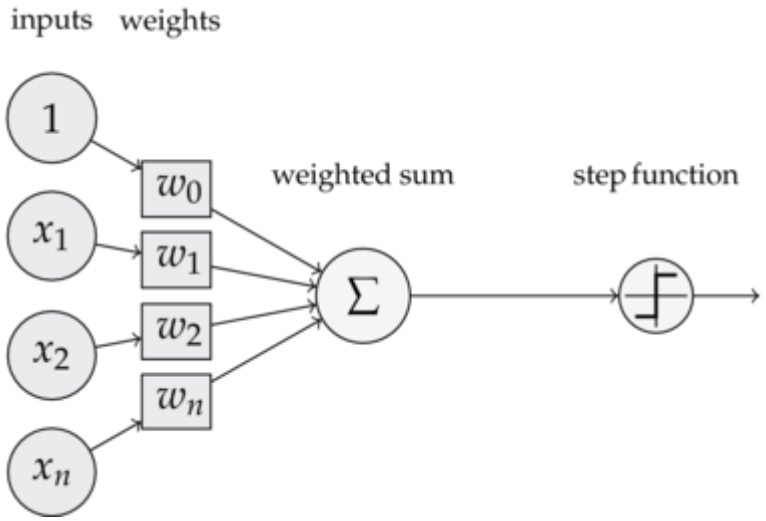
1958: Criação do modelo Perceptron por Frank Rosenblatt, que funcionava como um classificador binário. Seguiu-se um período de otimismo, em que se acreditava em uma rápida evolução. No entanto, não foi o que ocorreu.

1969: O matemático americano Marvin Minsky prova que o Perceptron é essencialmente um modelo linear, que somente consegue resolver problemas de classificação linear mas não processa dados não-lineares.

Como funciona o **Perceptron?**

O Perceptron em sua versão mais simples, de camada única, funciona como um classificador.

Ele toma como entrada um vetor multidimensional $X = [x_0, x_1, ..., x_n]$ e realiza uma classificação binária, dividindo as entradas em dois grupos, 0 ou 1. Veja:



Vetor de entrada: $X = [x_0, x_1, x_2, ... x_n]^T$
Pesos: $W = [w_0, w_1, w_2, ..., w_n]^T$
Função de ativação (degrau):

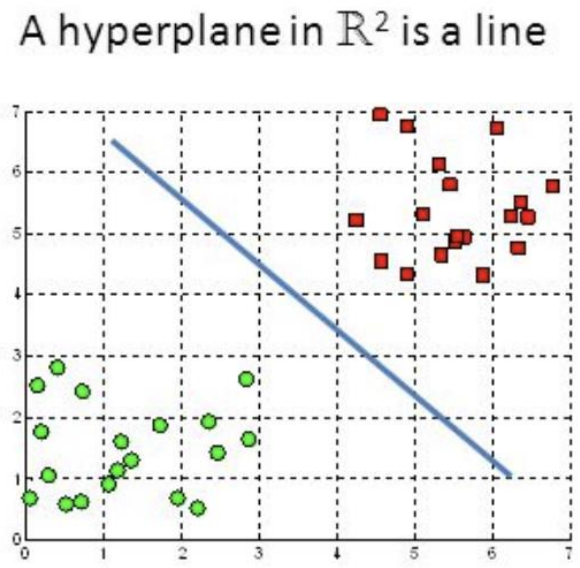
$$step(net) = \begin{cases} 1, se\ net > 0 \\ -1, \quad senão \end{cases}$$

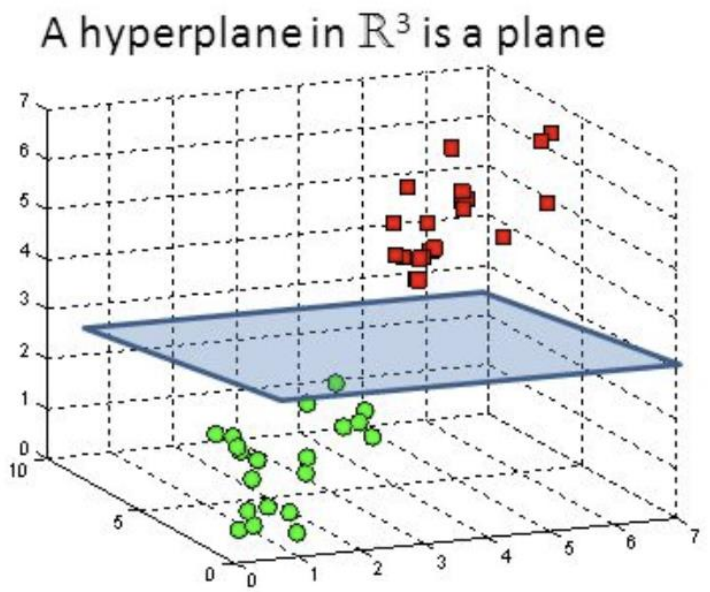
Soma ponderada (net):

$$net = \sum_{i=0}^{n} w_i x_i = \boldsymbol{W^T X}$$

Como funciona o **Perceptron?**

Portanto, o Perceptron constrói um hiperplano que separa os valores do universo de entrada em **duas classes**. O limite das classes é dado por $W^TX = 0$, que é um hiperplano com a quantidade de dimensões igual ao universo de entrada menos 1. Veja:

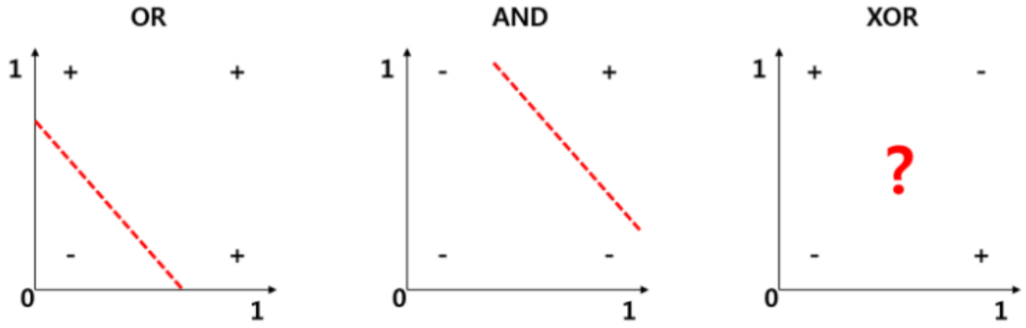




O Problema do **XOR**

Como um exemplo simples de problema, podemos tomar as operações lógicas **e**, **ou** e **ou-exclusivo**.

Repare que para as operações “e” e “ou”, para entradas x_1 e x_2, as saídas y são separáveis por uma linha no gráfico. As saídas com valor 0 ficam abaixo da linha, enquanto 1 ficam acima. No caso “**xor**”, onde deveria ser traçada essa linha? O Perceptron é incapaz de resolver este problema, por se tratar de um modelo linear.



x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

História

1986: Rumelhart, Hinton and Williams inventam o Perceptron de Múltiplas Camadas (MLP, de *MultiLayer Perceptron*), que utiliza uma função sigmoide para realizar o mapeamento não-linear e resolve o problema da classificação não-linear. Os pesquisadores também propõe o revolucionário algoritmo de retropropagação (***backpropagation***) para treinamento da rede com múltiplas camadas.

1995: Vapnik e Cortes propõem a criação da Máquina de Vetor de Suporte (SVM, de *Support Vector Machine*), um modelo neural com fortes bases teóricas e ótimo desempenho na prática.

2006: Início do advento da aprendizagem profunda. Neste ano, George Hinton propôs uma solução para o problema do treinamento de redes neurais profundas.

2010s: Em 2012, a rede AlexNet, projetada pela equipe de Hinton vence a principal competição de reconhecimento de imagens do mundo, o **ImageNet**. Em 2016, o AlphaGo, um programa de inteligência artificial de aprendizado profundo do Google, derrotou o campeão mundial de Go. A vitória trouxe a atenção do mundo para o aprendizado profundo a um novo nível.

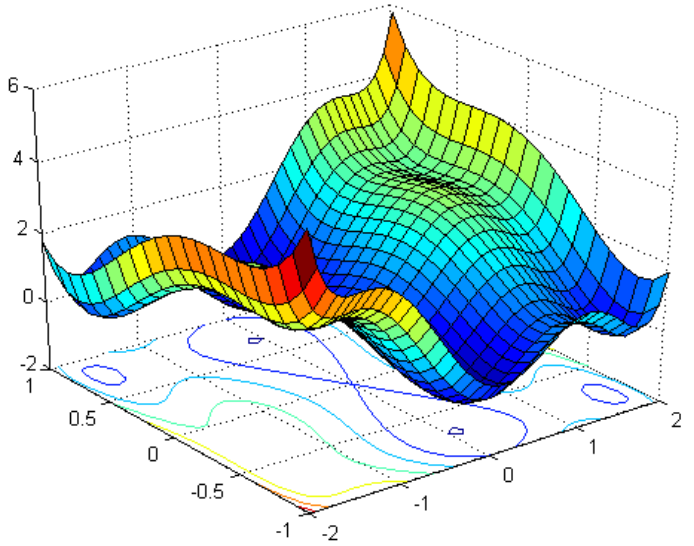
Qual a diferença?

O aprendizado profundo apresenta vantagens com relação a outros modelos mais simples, devido ao fato de ser um modelo com aprendizado de características não supervisionado. Ou seja, um modelo profundo geralmente possui a capacidade de **selecionar e extrair características (*features*) automaticamente** a partir do dado puro (*raw*), sem esforço do engenheiro humano. Isso a leva a ter grandes resultados em visão computacional, reconhecimento de voz e processamento de linguagem natural, por exemplo.

Aprendizado de Máquina Tradicional	Aprendizado Profundo
Baixo requisito de ***hardware***. Geralmente não necessita de GPUs para processamento paralelo.	Alto requisito de **hardware**. Demanda muito processamento por realizar operações de matrizes em dados massivos geralmente necessitando de GPUs para processamento paralelo.
Modelo pode ser treinado com relativamente **poucos dados**, e o desempenho nem sempre depende do aumento de dados.	O desempenho depende de uma **grande quantidade de dados** de treinamento.
Extração e seleção de ***features* manual**.	Extração e seleção de ***features* automáticas**, a partir do próprio algoritmo
Features **mais intuitivas** e fáceis de entender.	Features **menos intuitivas**, de significado obscuro.

Como é realizado o **Treinamento?**

O cerne do treinamento dos modelos neurais, profundos ou não, é a **função de perda** (*loss function*), também conhecida como função de erro (*error function*). Uma vez definida essa função, nosso papel é a aplicação de algoritmos que sejam capazes de encontrar os valores de parâmetros (pesos) que a minimizem.



A função de perda é utilizada para detectar o erro do modelo com relação ao resultado esperado (*target*). Ela deve refletir **a diferença entre o valor obtido e esperado**. Uma das funções mais utilizadas para isso é a de erro quadrático médio (MSE, de *Mean Squared Error*):

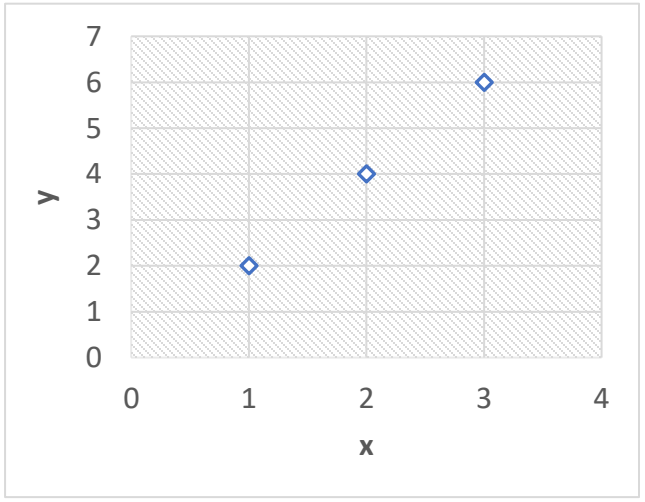
$$J(w) = \frac{1}{2n}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2$$

Em que n é a quantidade de valores da entrada (X), $\hat{y}$ é o valor esperado e y o valor obtido pelo modelo.

Como é realizado o **Treinamento?**

Imagine um exemplo bem simplificado, de um sistema para regressão, em que temos apenas uma entrada x e uma saída y, mapeadas como **y = wx + b**, onde w é o único parâmetro do modelo, ou peso. Temos a seguinte tabela de dados, através da qual é possível criar um gráfico bidimensional. Assumindo que a componente de *bias* b = 0, qual o melhor valor de w para minimizar a função de perda?

x	y
1	2
2	4
3	6



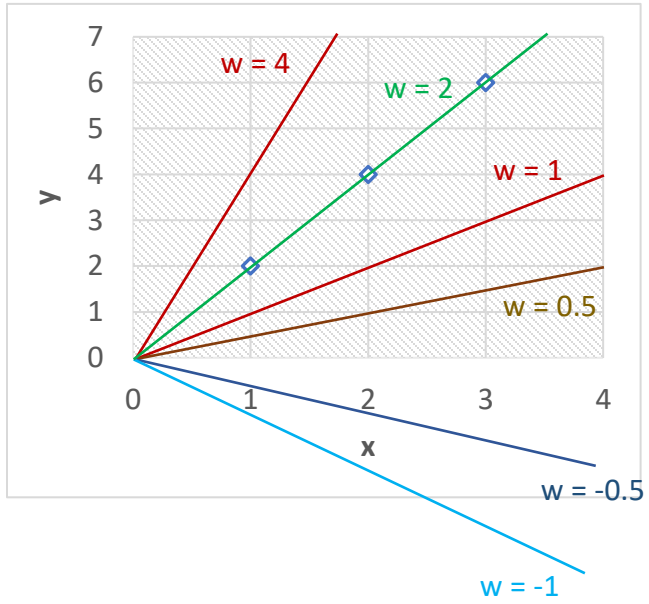
Que valor de w seria responsável por uma linha capaz de **representar bem o relacionamento** entre x e y expresso nos dados?

y = wx + b

Como é realizado o **Treinamento?**

No gráfico abaixo, cada uma das linhas foi traçada a partir de um valor diferente de w. O modelo regressor precisa **ajustar-se aos dados**, funcionando praticamente como um aproximador de função.

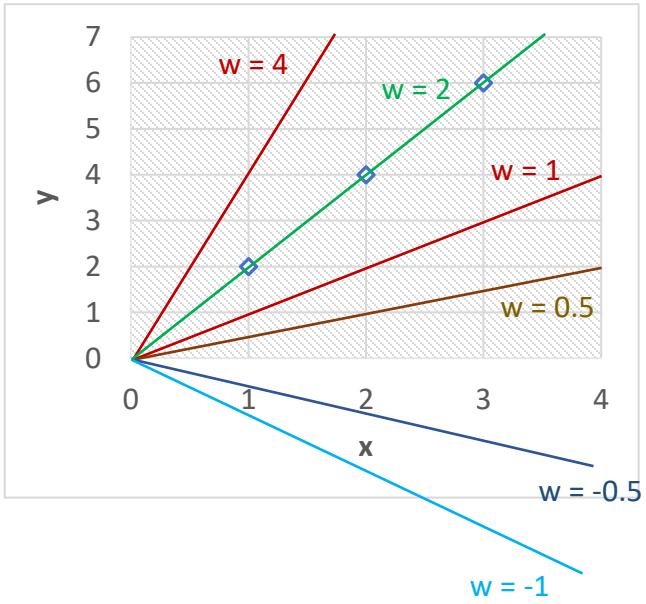
x	y
1	2
2	4
3	6



Claramente, o valor de w = 2 é a melhor opção, gerando a linha verde que passa exatamente sobre os pontos. Mas porque é a melhor? Faz com que a função de perda atinja seu menor valor (**minimiza**).

Como é realizado o **Treinamento?**

Podemos plotar um gráfico da função de perda em função de w, para verificar qual o seu comportamento para cada valor que testamos. No gráfico, temos uma "superfície" de erro no formato de uma parábola (já que a função J é quadrática). Precisamos encontrar o **ponto mais baixo** dessa superfície.



Calculando a função de perda J(w) para cada um dos valores possíveis de w, obtemos o seguinte gráfico:

$$J(w) = \frac{1}{2n}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2$$

Como é realizado o **Treinamento?**

$$J(w) = \frac{1}{2n}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2$$

Para w = 1, por exemplo:

$$J(w) = \frac{1}{2 * 3}\left((1 * 1 - 2)^2 + (1 * 2 - 4)^2 + (1 * 3 - 6)^2\right)$$

$$J(w) = \frac{1}{6}\left((1 - 2)^2 + (2 - 4)^2 + (3 - 6)^2\right)$$

$$J(w) = \frac{1}{6}\left((-1)^2 + (-2)^2 + (-3)^2\right)$$

$$J(w) = \frac{1}{6}(1 + 4 + 9) = \frac{1}{6}(14) = 2{,}333$$

Como é realizado o **Treinamento?**

$$J(w) = \frac{1}{2n}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2$$

Já para w = 2, obtemos o valor mínimo para a função de perda:

$$J(w) = \frac{1}{2*3}\left((2*1-2)^2 + (2*2-4)^2 + (2*3-6)^2\right)$$

$$J(w) = \frac{1}{6}\left((2-2)^2 + (4-4)^2 + (6-6)^2\right)$$

$$J(w) = \frac{1}{6}\left(0^2 + 0^2 + 0^2\right)$$

$$J(w) = \frac{1}{6}(0) = 0$$

E assim por diante...

Como é realizado o **Treinamento?**

Portanto, nosso processo de treinamento das redes neurais consistirá em buscar no espaço dos parâmetros da rede (pesos) os valores de W que seriam responsáveis pela **minimização da função de perda** J(W), e fariam com que nosso sistema, consequentemente, tivesse o **menor erro possível**.

E como fazer a busca?

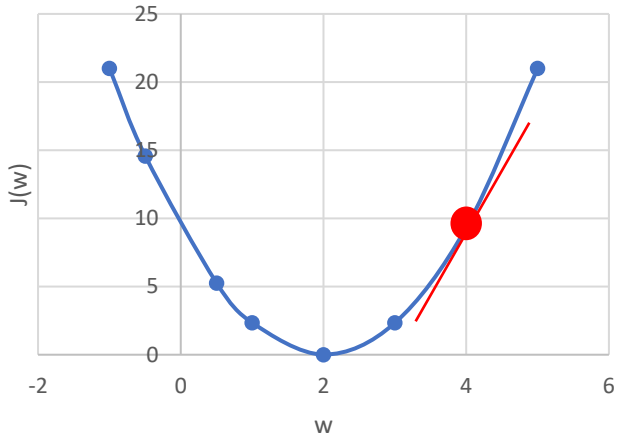
Podemos começar com um **valor aleatório de w** e ir caminhando lentamente na direção do valor mínimo de J(w), atualizando a cada passo o valor de w. A atualização se dá através da seguinte função:

$$w = w - \alpha \frac{\partial}{\partial w} J(w)$$

Agora complicou, hein!? **Derivada**? O que é esse α??

Como é realizado o **Treinamento?**

Nessa equação, simplesmente estamos dizendo que o novo valor de w será o valor antigo decrescido (ou acrescido) de um bocadinho a cada passo do treinamento. A quantidade desse "bocadinho" é dada pelo multiplicador α, que é a **taxa de aprendizagem** (*learning rate*) e a "direção" do incremento é dada pela **derivada** da função de perda com relação a w. Olhe o seguinte cenário:



Imagine que o peso inicial seja **w = 4**.

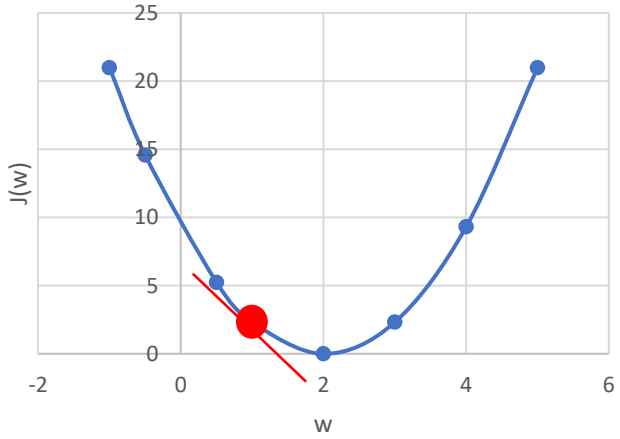
A derivada da função J(w) é dada pela **inclinação da reta que passa tangente** àquele ponto. Como a inclinação é positiva, o valor de $\partial J(w)$ é positivo e w será decrescido, para que se aproxime do mínimo.

Decrescido de quanto? Aí depende de quanto a curva está inclinada e do valor escolhido para α. Se α for muito pequeno, o sistema demora demais para convergir. Se for grande demais, o valor de w pode ficar pulando de um lado para outro e nunca convergir. Busque equilíbrio.

$$w = w - \alpha \frac{\partial}{\partial w} J(w)$$

Como é realizado o **Treinamento?**

E se o valor de w estiver na **"descida" da curva**?



Imagine que o peso em determinado momento seja **w = 1**.

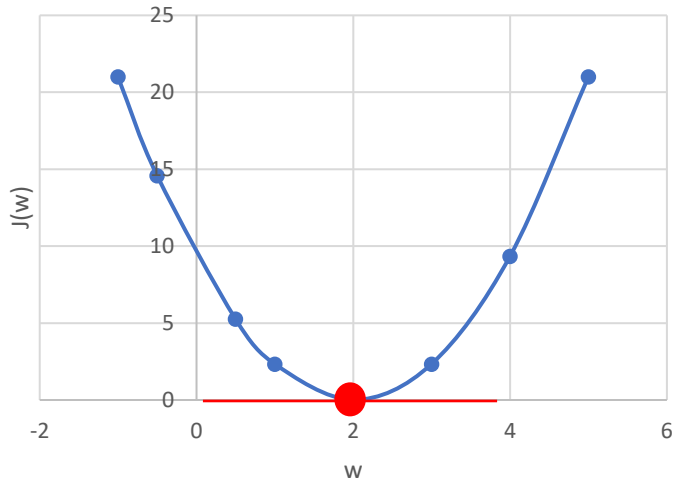
A derivada da função J(w) é dada pela **inclinação da reta que nesse caso é negativa**. A curva está descendo. Portanto, o valor de $\partial J(w)$ é negativo e w será acrescido, para que se aproxime também do mínimo.

$$w = w - \alpha \frac{\partial}{\partial w} J(w)$$

Portanto, o valor de w é atualizado na direção contrária ao valor da derivada. Se for positiva (subida), o valor de w precisa diminuir. Se for negativa (descida), o valor de w aumenta. Daí vem o termo "descendente".

Como é realizado o **Treinamento?**

Como saber se foi **atingido o mínimo** da função e perda?



Quando o peso atingir o valor ótimo, **w = 2**.

A derivada da função J(w), que é dada pela **inclinação da reta, será igual a zero**. A curva está paralela ao eixo. Portanto, o valor de $\partial J(w)$ é nulo e o valor de w será mantido.

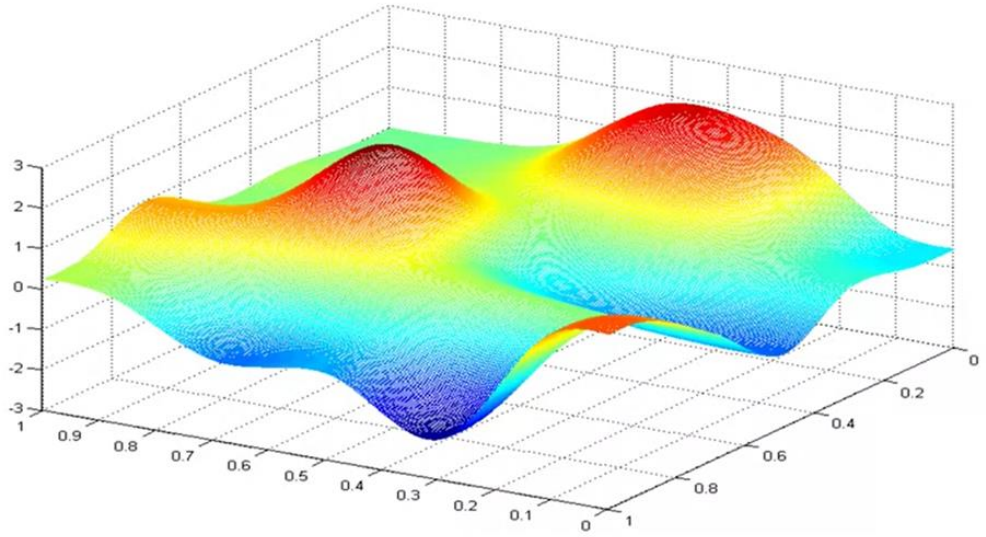
$$w = w - \alpha * 0$$

$$w = w$$

Quando o valor de w parar de ser alterado, ou sofrer muitas poucas alterações, dizemos que o sistema "convergiu", ou seja, encontrou um mínimo na superfície de erro.

Como é realizado o **Treinamento?**

No entanto, o exemplo anterior é bastante simplificado. Nossos sistemas dificilmente terão apenas um valor de entrada e um único parâmetro w. Geralmente estaremos trabalhando com **vetores**. Nesse caso, a superfície de erro pode ser muito mais complexa do que a forma de sela da parábola:



Além disso, a função de perda MSE não é a única. Outras funções também podem ser utilizadas, por serem mais adequadas a outros contextos. **MSE é utilizada geralmente em problemas de regressão**.

Por exemplo, temos a função de perda chamada ***cross entropy***, utilizada para modelas a distância entre duas distribuições de probabilidade, e muito utilizada em problemas de **classificação**.

$$J(w) = -\frac{1}{n}\sum_{x}\sum_{d\in D}[y\, ln\hat{y} + (1-y)\ln(1-\hat{y})]$$

Gradiente Descendente

Quando lidamos com **funções de perda multivariáveis**, como $J(w_1, w_2, ..., w_n)$, ao invés de apenas uma único parâmetro, é necessário substituir a derivada pelo **gradiente**, denotado por **∇J**. O gradiente da função contém todas as informações sobre suas derivadas parciais em um vetor $\nabla J = \left[\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, \dots, \frac{\partial J}{\partial w_n}\right]^T$.

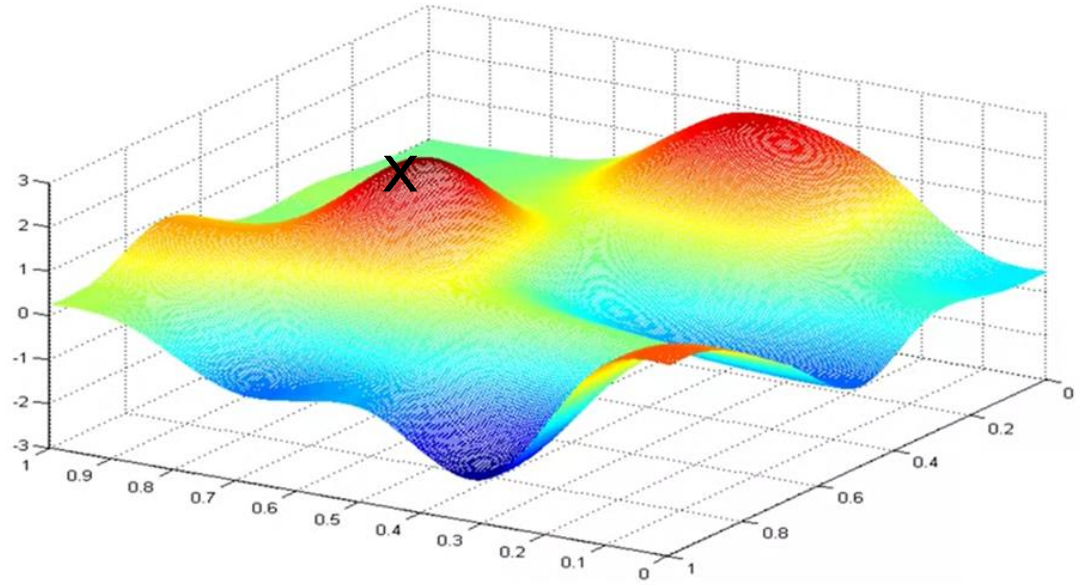
O que isso significa para nosso processo de **treinamento**?

- Imagine que você esteja parado em um ponto qualquer ($w_{1(0)}, w_{2(0)}, ..., w_{n(0)}$) no domínio da função J (na superfície de erro). Ao calcular o gradiente, o valor de ∇f diz a você em que direção você deve caminhar para aumentar o valor de f mais rapidamente.

- No entanto, como estamos tentando minimizar o erro, **andaremos na direção contrária ao gradiente**! Por isso, o algoritmo é chamado de **gradiente descendente**.

Gradiente Descendente

Imagine que a função J(W) seja responsável pela superfície de erro abaixo. Como faremos para encontrar os melhores valores de parâmetros para minimizar o erro? Primeiramente, precisamos partir de algum ponto inicial aleatório, como por exemplo aquele marcado com um X no gráfico:

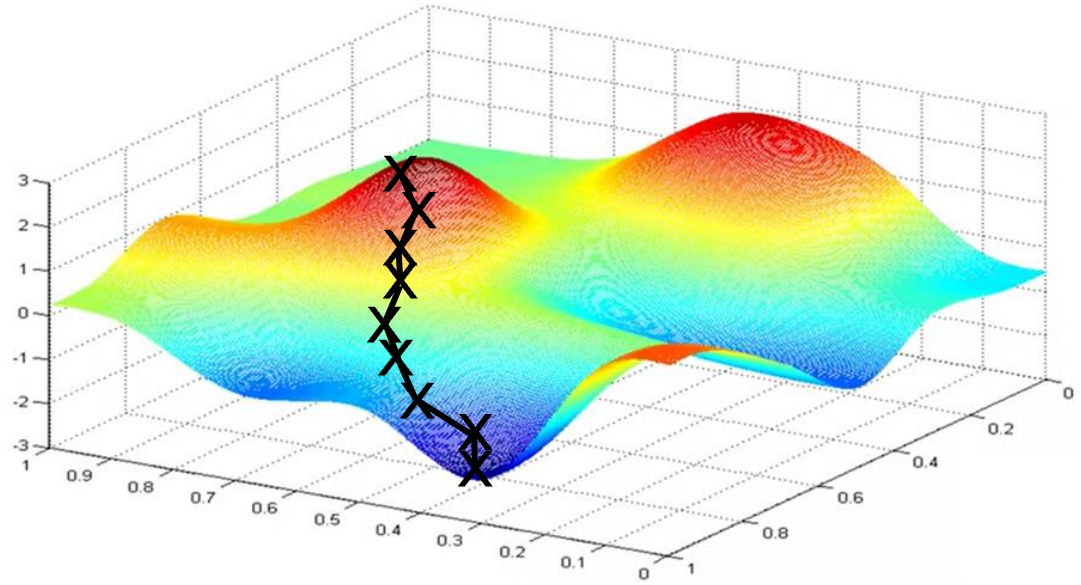


A partir do ponto inicial, iremos atualizar nosso vetor de pesos passo a passo, caminhando na direção contrária ao do gradiente calculado.

Faremos isso até encontrar um vale na superfície de erro, ou um **mínimo local**. Repare que o caminho será diferente para cada escolha de ponto inicial (podendo inclusive chegar em outro mínimo local).

Gradiente Descendente

Imagine que a função J(W) seja responsável pela superfície de erro abaixo. Como faremos para encontrar os melhores valores de parâmetros para minimizar o erro? Primeiramente, precisamos partir de algum ponto inicial aleatório, como por exemplo aquele marcado com um X no gráfico:



A partir do ponto inicial, iremos atualizar nosso vetor de pesos passo a passo, caminhando na direção contrária ao do gradiente calculado.

Faremos isso até encontrar um vale na superfície de erro, ou um **mínimo local**. Repare que o caminho será diferente para cada escolha de ponto inicial (podendo inclusive chegar em outro mínimo local).

Gradiente Descendente

Portanto, o algoritmo do gradiente descendente é como se você estivesse em **um campo de golfe** bem extenso. Seu objetivo seria encontrar o ponto mais baixo de todo o campo. Para isso, você sempre anda na **direção oposta à maior inclinação**.

Gradiente Descendente

Existem algumas maneiras de se computar o gradiente descendente durante o treinamento.

1) Batch Gradient Descent (BGD):

No conjunto de treinamento D, cada padrão de entrada é representado por <X, t> em que X é o vetor de entrada e t é a saída esperada. O algoritmo consiste em inicializar o vetor de pesos W com um valor aleatório. Para cada <X, t> em D, o vetor X é inserido no modelo e obtém-se um valor o como saída. Para cada valor w_i do vetor de pesos W, é calculado um incremento para o valor de Δw_i. Ao final, os pesos são todos atualizados com o valor acumulado após a computação de todos os padrões de entrada.

Portanto, **para cada valor de entrada, há uma atualização do valor dos pesos do modelo**.

Na prática, este algoritmo é pouco utilizado, pois a convergência é muito lenta: todos os padrões de entrada do treinamento precisam ser calculados todas as vezes que os pesos são atualizados.

Gradiente Descendente

2) Stochastic Gradient Descent (SGD):

Nesse caso, ao invés de passar por todos os padrões de entrada, o SGD realiza a atualização dos parâmetros em cada exemplo <X, t> de forma **aleatória** (que é o significado de estocástico, pra falar a verdade).

Uma das suas implementações é chamada de ONLINE-GRADIENT-DESCENT. Nela, os pesos são inicializados aleatoriamente e, até que a condição final seja satisfeita, o algoritmo **seleciona um <X, t> aleatório de D**, computa a saída para esse padrão e, baseado no erro obtido, atualiza o valor dos pesos w_i.

Quantidade de computação necessária diminui dramaticamente com relação ao BGD. No entanto, como é utilizado apenas um exemplo de dado para o cálculo, não é garantido que a direção seguida é a **direção real** do gradiente, e, portanto, também **não é garantido que alcancemos um mínimo** local da superfície.

Gradiente Descendente

3) Mini-Batch Gradient Descent (MBGD):

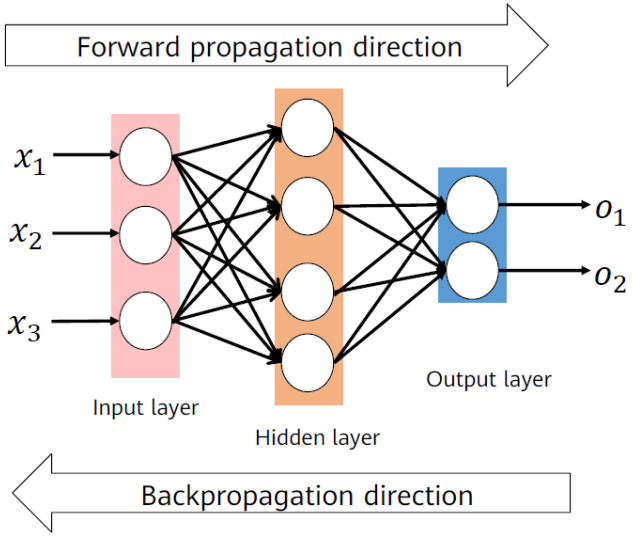
Para minimizar os problemas das duas abordagens anteriores, o algoritmo MBGD foi proposto e é o **mais amplamente utilizado**. Nele, um pequeno número de padrões de entrada do conjunto D é selecionado e forma aleatoriamente um mini lote (*mini-batch*). O valor de ΔW é calculado para este pequeno lote e os pesos são atualizados. A seguir, um novo mini lote é selecionado, e o peso novamente atualizado. E assim por diante, até o sistema convergir.

Dessa maneira, não é necessário passar pelo modelo todos os exemplos do conjunto de dados D para atualizar os pesos (demora demais), mas também não passamos apenas um (pouco representativo - não chegamos ao mínimo).

É um meio termo entre as abordagens anteriores.

Backpropagation

Em alguns casos, como em uma regressão linear ou em uma SVM, o cálculo do gradiente descendente e a minimização da função de perda podem ser feitos com relativa tranquilidade. No entanto, **nas redes neurais com várias camadas (MLP), a questão fica mais complexa**. É impossível calcular o gradiente da função de perda para todos os pesos da rede em uma única fórmula. Os valores de pesos **em cada camada** possuem uma posição diferente com relação à saída final do modelo, e contribuem também de forma diferente.



Com isso em mente, foi proposto o algoritmo de retropropagação de erro, ou simplesmente, *backpropagation* (BP). Nele, os pesos são atualizados camada por camada, o que viabiliza um treinamento eficiente da rede neural.

Sinais são propagados na direção normal (forward).
Erros são propagados na direção oposta (backwards)

Backpropagation

De acordo com o algoritmo de *backpropagation*, os erros das camadas de entrada, ocultas e de saída **se acumulam para gerar o erro total** da função de perda. Portanto, a contribuição de cada camada é diferente e a atualização de seus pesos precisa ser calculada separadamente. Se os pesos entre a camada de entrada e oculta forem W_c, e da oculta para a saída forem W_b, utilizamos as fórmulas:

$$\Delta W_c = -\alpha \frac{\partial J}{\partial W_c} \qquad \Delta W_b = -\alpha \frac{\partial J}{\partial W_b}$$

Se houver várias camadas ocultas, a **regra da cadeia** é utilizada para obtermos as derivadas para cada camada e obter a atualização dos parâmetros a cada iteração.

Backpropagation

Em resumo:

1. O algoritmo alimenta cada instância de treinamento para a rede e calcula a saída de cada neurônio em cada camada consecutiva (*forward pass*). Ao final, é obtido o resultado da rede para aquela instância (valor previsto) e **calculado o erro** total da camada de saída (com base no valor esperado).

2. A seguir, o algoritmo propaga ao contrário (***backpropagation***) o erro total através da rede (*reverse pass*), passando por cada camada medindo a contribuição do erro em cada conexão (para cada peso).

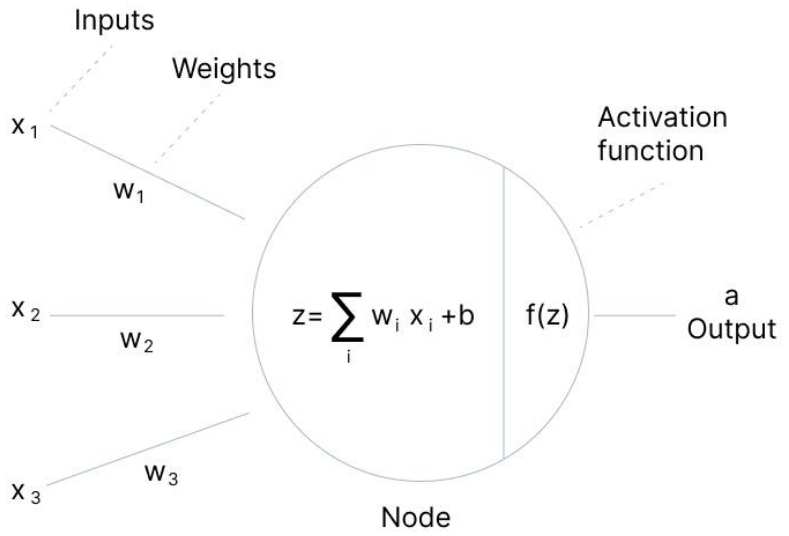
3. Finalmente, o algoritmo **ajusta os pesos** de cada conexão para reduzir o erro do modelo (através de uma versão mais complicada do gradiente descendente).

Funções de Ativação

O papel primário de uma função de ativação é transformar a soma ponderada da entrada de um neurônio em um valor de saída que alimentará a próxima camada da rede ou será a própria saída da rede.

Portanto, elas servem como um passo adicional que acontece em cada camada na etapa de propagação do sinal (forward). Cada neurônio:

1) Toma as **entradas** (X) e multiplica pelos **pesos** (W);
2) Adiciona um termo de ***bias*** (b), que é uma constante capaz de "movimentar" a função de ativação em um eixo (*offset*);
3) Aplica uma **função de ativação não-linear** que transforma os dados;
4) **Transmite o resultado** para a próxima camada de neurônios.



Funções de Ativação

As funções de ativação têm um papel muito importante no aprendizado de modelos neurais. São elas que tornam o modelo capaz de compreender e modelar relacionamentos não-lineares. Sem uma função de ativação, as conexões entre neurônios consistiriam em **operações lineares** – um **produto escalar** e uma **soma**:

$$output = W^T X + b$$

Portanto, cada camada seria capaz de aprender somente transformações lineares dos dados de entrada (X), mesmo que muitas camadas fossem acrescentadas em sequência. O aprendizado é armazenado em W.

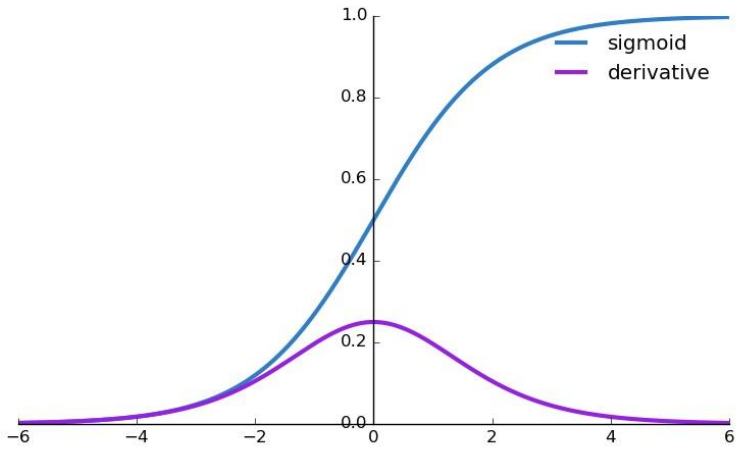
Para **acessar relacionamentos mais ricos e complexos** presentes nos dados de entrada, o modelo se beneficia do acréscimo de uma componente não-linear, na forma de uma **função de ativação "f"**:

$$output = f(w_1 x_1 + w_2 x_2 + w_3 x_1 + \ \ldots) = f(W^T X + b)$$

Funções de Ativação

1) Sigmóide ou Logística:

Uma das funções mais populares é a **sigmóide**, dada pela equação abaixo e representada pela **curva azul**.



$$f(x) = \frac{1}{1 + e^{-x}}$$

A função é semelhante à função degrau, Pode ser utilizada em **camadas de saída** de modelos neurais, para obter uma **classificação binária** (0 ou 1).

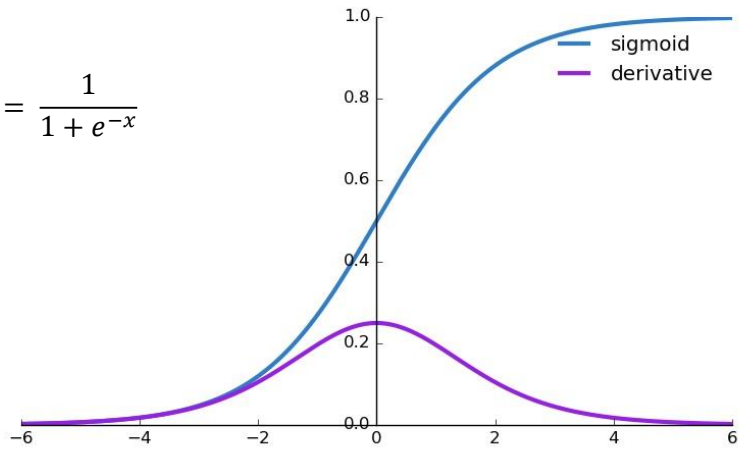
No entanto, ela é **contínua** e sua derivada é fácil de calcular (curva roxa). Isso **facilita a convergência** da rede.

Funções de Ativação

1) Sigmoide ou Logística:

Uma das funções mais populares é a **sigmoide**, dada pela equação abaixo e representada pela **curva azul**.

$$f(x) = \frac{1}{1 + e^{-x}}$$



No entanto, é possível perceber que os valores de gradiente somente são significativos para entradas com valores entre -4 e 4. Para valores fora desse intervalo, o gradiente calculado teria valores muito próximos de zero. Isso torna a rede difícil de treinar, se tiver muitas camadas. É o problema da **dissipação de gradiente** (*vanishing gradient*).

Além disso, a função logística não é simétrica com relação a zero. Isso significa que a saída de todos os neurônios terá o mesmo sinal (positivo). Isso também dificulta o treinamento da rede, tornando-o menos estável e mais demorado.

Funções de Ativação

O algoritmo de *backpropagation* propaga um sinal de erro da rede, partindo da saída até as camadas iniciais. Se esse sinal tiver que ser propagado através de um número grande de camadas, ele pode se tornar fraco ou mesmo ser perdido, tornando a atualização dos pesos e o treinamento da rede impossíveis. Esse problema é conhecido como dissipação do gradiente (***vanishing gradient***).

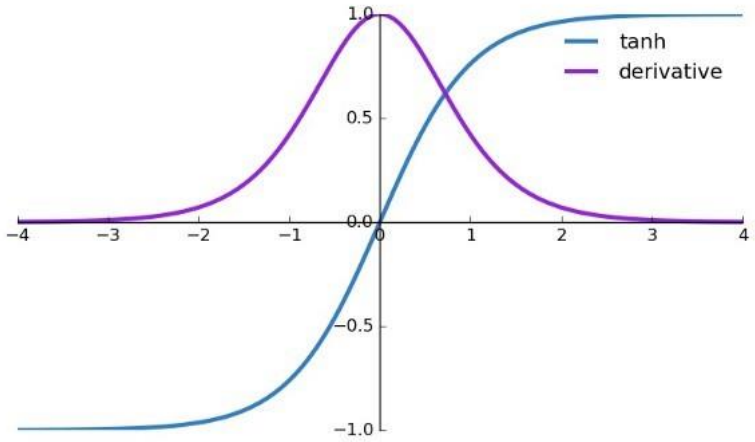
Algumas funções de ativação, como a sigmoide, fazem com que grandes mudanças na entrada reflitam em pequenas mudanças nas saídas dos neurônios (0 a 1). As derivadas são ainda menores, especialmente quando longe da origem (fica próxima de zero).

Quando muitas camadas da rede utilizam uma função de ativação com essas características, derivadas pequenas são multiplicadas a cada passo (regra da cadeia). Assim, o gradiente diminui rapidamente enquanto é propagado para as camadas iniciais. **Se o gradiente for muito pequeno, os pesos das camadas iniciais são pouco ou nada atualizados**. Como essas camadas são importantes para o reconhecimento de características dos dados de entrada, o sistema inteiro fica prejudicado e obtém desempenho ruim.

Funções de Ativação

2) Tangente Hiperbólica ou Tanh:

Muito similar à função sigmoide. Também tem o formato característico da letra “S” (curva azul), mas possui um intervalo diferente, entre -1 e 1. Nessa função, quanto maior a entrada, mais próximo de 1 será seu resultado. Quando menor a entrada, a saída aproxima-se de -1.



$$f(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$$

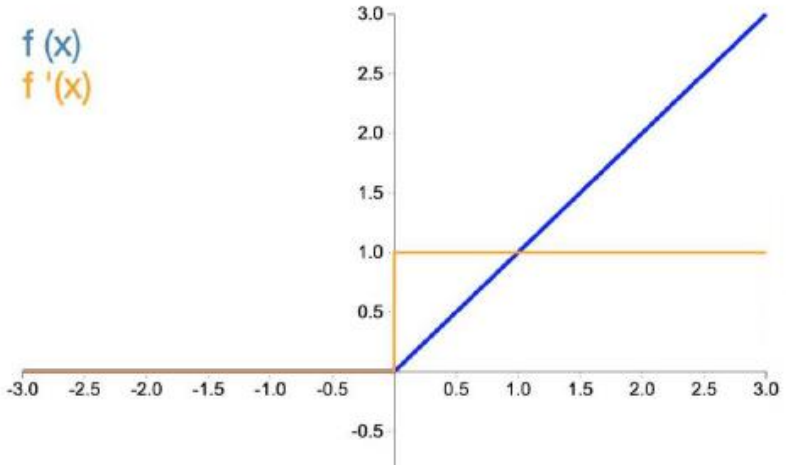
Uma vantagem com relação à função logística é que a saída da função **tanh** é centralizada em zero, o que facilita o mapeamento dos valores de saída para fortemente positivo, negativo ou neutro. Isso também **facilita o treinamento da rede**.

Ela também sofre com o problema da dissipação do gradiente, mas é geralmente **preferível com relação à sigmoide**.

Funções de Ativação

3) ReLU (*Rectified Linear Unit*):

É a função de ativação mais utilizada (curva azul). Diferente das anteriores, não há um valor limite (1, -1) para sua saída. Isso faz com que os neurônios nunca saturem e mitiga o problema da dissipação do gradiente, tornando o treinamento eficiente, e convergindo mais rapidamente o gradiente descendente.



$$f(x) = \begin{cases} x, se\ x \geq 0 \\ 0, se\ x < 0 \end{cases}$$

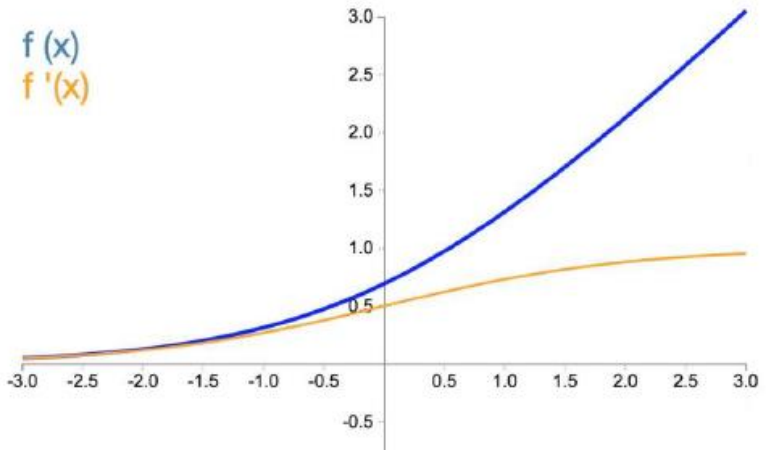
Diferente das funções anteriores, a ReLU **não necessita de operações de exponenciação**, que são computacionalmente intensas. Gradiente também é calculado muito facilmente (0 ou 1).

Para entradas negativas, o neurônio passa a não ser ativado (saída igual a 0). Com sua derivada igual a 0, os pesos não são atualizados. Outra desvantagem é que, com sua saída sem limites, valores muito altos podem ser obtidos, e a rede pode divergir.

Funções de Ativação

4) Softplus:

É vista como uma versão mais suave da ReLU. Computacionalmente é mais exigente do que a ReLU, mas possui uma derivada contínua e uma curva mais suave.



$$f(x) = \ln(e^x + 1)$$

Possui características muito semelhantes a ReLU, mas é mais difícil de computar (tanto o sinal quanto a derivada). Softplus fornece mais estabilização para redes neurais profundas do que a função ReLU, devido à curva mais suave e diferenciação quando x = 0. No entanto, ReLU é geralmente preferida devido à facilidade em calculá-lo e sua derivada. Esses cálculos são operações frequentes em redes neurais, e a ReLU fornece uma propagação mais rápida para frente e para trás quando comparado com a função softplus.

Funções de Ativação

5) Softmax:

A função softmax é uma extensão da função sigmoide utilizada em **casos multidimensionais**. Funciona como a combinação de várias funções sigmoides. É projetada para mapear um vetor arbitrário de K dimensões para uma distribuição de probabilidade também com K dimensões, Portanto, é geralmente utilizada na **camada de saída** para problemas de classificação com múltiplas classes. Sua fórmula é:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

Por exemplo, assuma que você tem um problema com três classes: cachorro, gato e rato. Agora, imagine que as saídas dos neurônios da camada de saída seja [1.8, 0.9, 0.68]. Se aplicarmos a função softmax a esses valores, para termos uma visão probabilística do resultado, teremos o vetor [0.58, 0.23, 0.19]. Isso significa que há 58% de chance de ser um cachorro, 23% de ser um gato e 19% de ser um rato. Provavelmente é um cachorro! **Por isso é tão utilizada em problemas de classificação muticlasses**.

Regularização

Um modelo e aprendizado profundo é muito maior do que um modelo comum de aprendizado de máquina e, portanto, está mais propenso ao problema do *overfitting*. Quando estamos projetando, nosso objetivo é que o modelo tenha uma boa capacidade de **generalização**, ou seja, que ele aprenda o comportamento do sistema a partir de dados, e seja capaz de aplicar esse "conhecimento" em dados desconhecidos.

Para reduzir o erro de generalização, foram introduzidas técnicas conhecidas como **regularização**, incluindo:

- Termos de penalização L1 e L2;
- *Data augmentation*;
- *Dropout;*
- *Early Stopping.*

Regularização

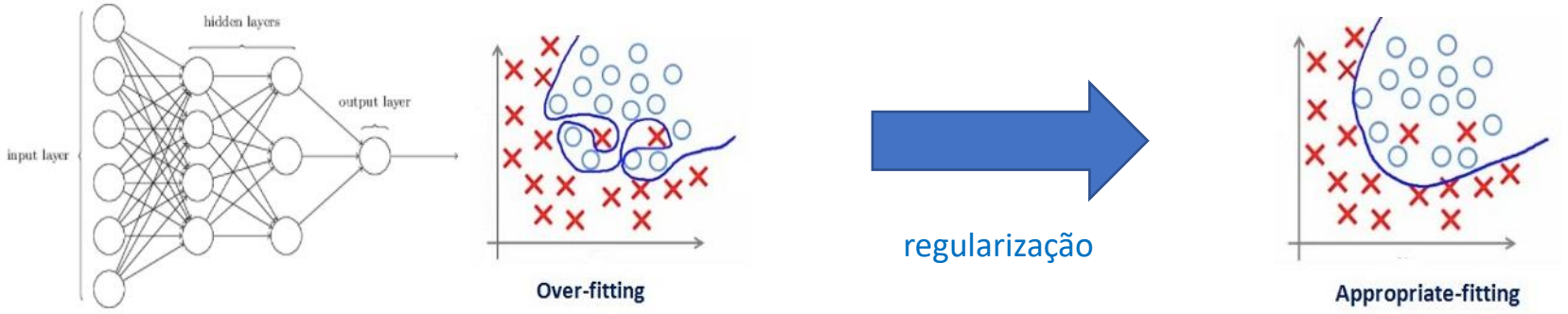
Segundo o princípio filosófico conhecido como a “**Navalha de Occam**”:

Dadas duas explicações para algo, a explicação com maior probabilidade de **estar correta é a mais simples** – aquela que faz menos suposições. Essa ideia também se aplica aos modelos aprendidos pelas **redes neurais**: dados alguns dados de treinamento e uma arquitetura de rede, vários conjuntos de valores de peso (vários modelos) podem explicar os dados. Modelos **mais simples** são menos propensos a superajustar (*overfit*) do que os complexos.

"The simplest explanation is usually the best one.“
(Guilherme de Ockham, filósofo e frade do século 14)

Regularização

A maneira mais simples de prevenir o overfitting é reduzindo o tamanho do modelo. O número de parâmetros ajustáveis do modelo (pesos) é geralmente conhecido como sua “capacidade”. Se sua capacidade for grande, ele terá um poder de “**memorização**” muito elevado, e funcionará quase como um dicionário – refletindo em uma generalização pobre. A regularização pode nos ajudar a resolver este problema:



L1 e L2

Uma das maneiras mais populares de tornar o modelo mais simples é através da adição de **restrições ou limitações** (*constraints*) na complexidade da rede, forçando que seus pesos tenham apenas valores pequenos, diminuindo a entropia e tornando a distribuição de pesos mais regular através da rede. Isso é chamado de "**regularização**" dos pesos, e é realizado através das adição de um termo à função de perda que seja associado a um "custo" de se ter valores grandes de peso. A função de perda passa a ser:

Função de Perda = Perda Normal + Termo de regularização

E há dois tipos de termos de regularização: L1 e L2.

- **L1:** O custo adicionado é proporcional ao valor absoluto dos coeficientes de peso (norma L1 dos pesos).
- **L2:** O custo adicionado é proporcional a quadrado do valor dos coeficientes de peso (norma L2)

L1 e L2

Para L1, portanto: $\tilde{J}(w; X, y) = J(w; X, y) + \alpha \|w\|_1,$

Enquanto, para L2: $\tilde{J}(w; X, y) = J(w; X, y) + \frac{1}{2}\alpha \|w\|_2^2,$

A **penalização do modelo depende do valor de α**, que pode ser ajustado.

No processo, cada coeficiente na matriz de peso da rede adicionará **α*coeficiente** ao valor total da função de perda. Isso significa que, durante o cálculo do gradiente, haverá uma penalização proporcional à norma do peso da rede, e os pesos permanecerão pequenos. Como resultado, a capacidade de aprendizado da rede fica diminuída, e **a chance de sobreajuste (*overfitting*) diminui**. Lembre-se: um valor de α que seja grande demais pode gerar o problema contrário: *underfitting*.

A diferença principal entre as técnicas é que L1 diminui os coeficientes relacionados a *features* menos importantes dos dados para zero, removendo essas *features* totalmente no processo. Funciona bem para seleção de *features*, gerando um modelo mais esparso do que L2.

Data Augmentation

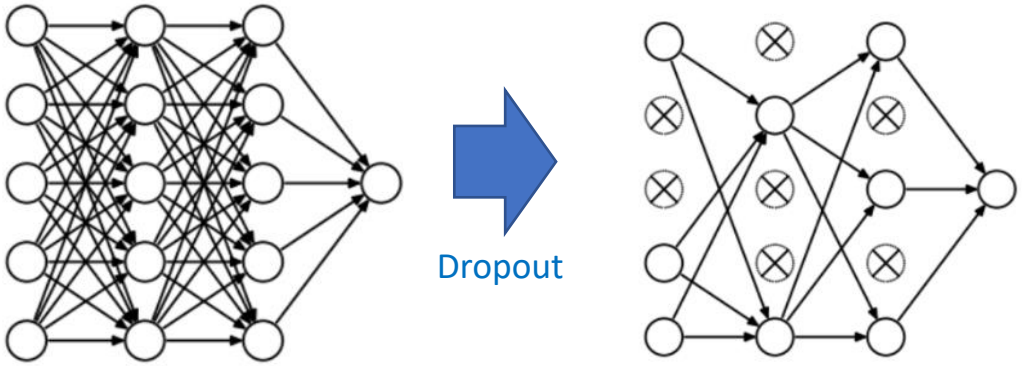
Uma das maneiras mais efetivas de se evitar o *overfitting* é **aumentar o tamanho do conjunto de dados de treinamento**. Se houver mais dados, o modelo terá mais dificuldades para memorizar com muita exatidão. No entanto, muitas vezes a coleta de novos dados pode ser difícil ou inviável. Existem técnicas para aumento artificial do *dataset* a partir dos dados já existentes, como:

- Quando o dataset é composto por **imagens**, podem ser aplicadas **transformações como rotação, escala, *blurring*,** etc. Essa transformação não pode influenciar excessivamente. Imagine por exemplo se rotacionarmos uma imagem do dígito 6 até que ele vire um 9. Atrapalharia o modelo.
- No **reconhecimento de voz**, é possível adicionar ruído nos dados de entrada.
- Em problemas de **Processamento de Linguagem Natural** (NLP), é comum substituir palavras por sinônimos.
- E assim por diante...

Essa técnica é muitas vezes referida como "*data augmentation*", e pode ser muito importante para a eficiência de um sistema com dados insuficientes.

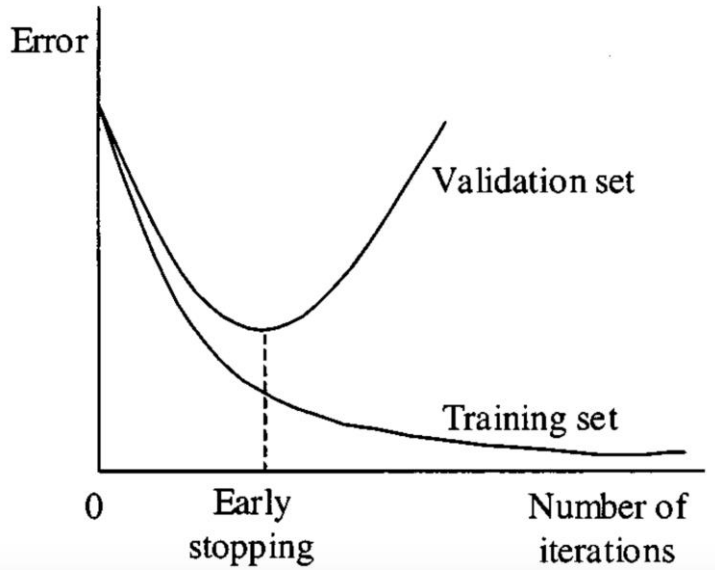
Dropout

Um dos tipos mais interessantes de regularização, produz bons resultados e é **uma das técnicas mais utilizadas em *deep learning***. Como funciona? Imagine a rede neural abaixo. A cada iteração, o algoritmo aleatoriamente seleciona alguns nós (neurônios) e os "remove" temporariamente do modelo, com todas as suas conexões e parâmetros. Os parâmetros correspondentes aos nós descartados **não são atualizados nessa rodada**. É como se fossem criadas sub-redes a cada passo, e o resultado final fosse a combinação delas, em uma técnica de *ensemble*. Precisa de uma grande quantidade de dados de treinamento.



Early Stopping

É uma técnica em que uma parte dos dados de treinamento é separada em **conjunto de validação**. A função de perda é então calculada para esse conjunto a cada etapa, refletindo uma amostra do que seria o desempenho do modelo em dados "desconhecidos". Quando o valor da função de perda começa a subir, temos um indicativo de que o modelo está começando a sobreajustar (***overfit***) e é o momento de uma **parada precoce** (*early stopping*). Como o conjunto de validação é apenas uma amostra, corremos o risco de *underfitting*.



Apoio

Este projeto é apoiado pelo Ministério da Ciência, Tecnologia e Inovações, com recursos da Lei nº 8.248, de 23 de outubro de 1991, no âmbito do [PPI-Softex| PNM-Design], coordenado pela Softex.

João Paulo Reus Rodrigues Leite
Universidade Federal de Itajubá
e-mail: joaopaulo@unifei.edu.br

UNIFEI

Softex

MCTI FUTURO
FUTURO DO TRABALHO, TRABALHO DO FUTURO

GOVERNO FEDERAL
BRASIL
UNIÃO E RECONSTRUÇÃO