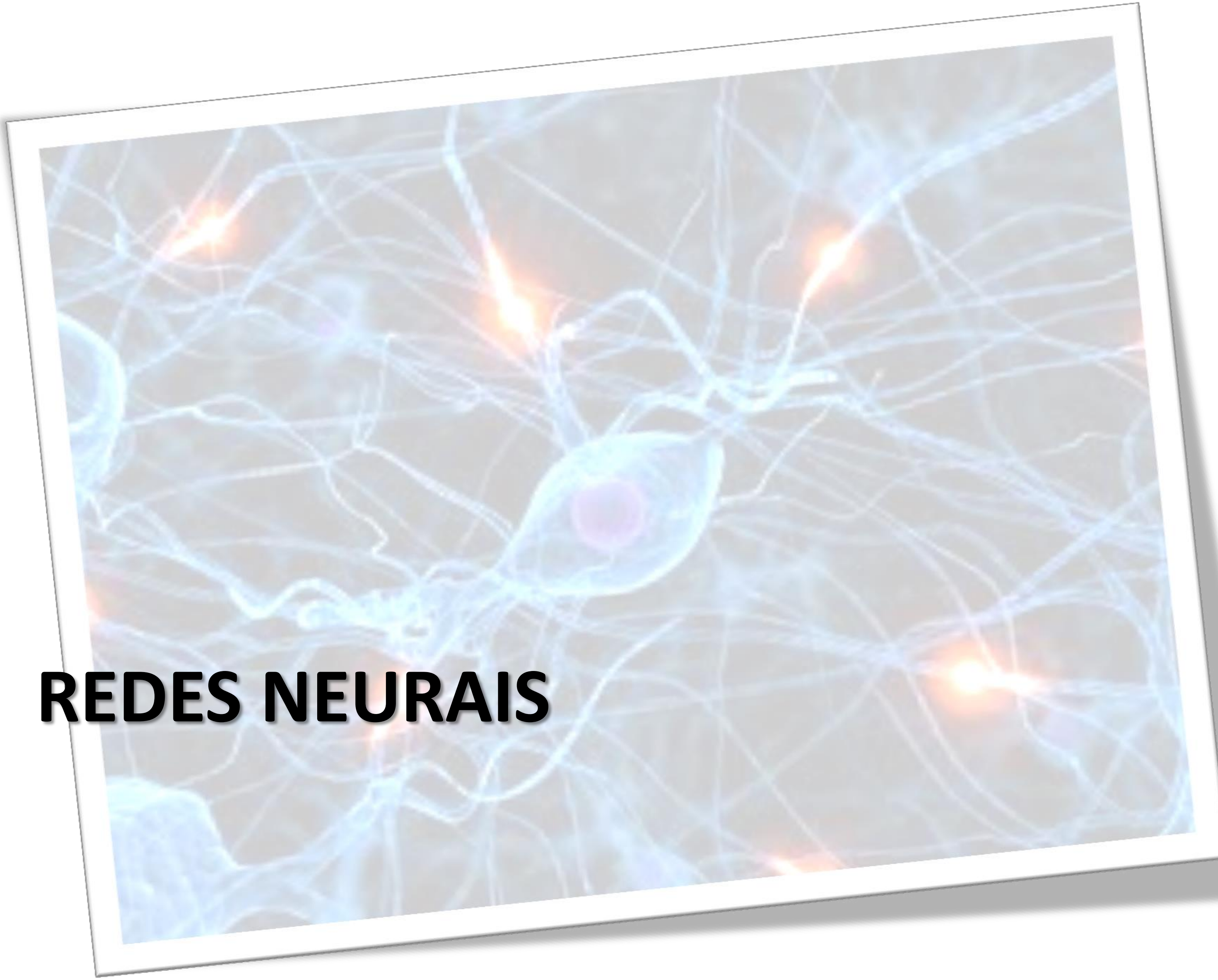
# Aprendizado de Máquina

Sistemas de Informação Inteligente

Prof. Leandro C. Fernandes

Adaptação dos materiais de:
Thiago A. S. Pardo, Daniel Honorato e Bianca Zadrozny

# REDES NEURAIS



# Redes Neurais

- Modelos inspirados no cérebro humano, criadas em analogia a sistemas neurais biológicos, que são capazes de aprendizagem.
  - Compostas por várias unidades de processamento ("neurônios")
  - Interligadas por um grande número de conexões ("sinapses")

- Criadas com o objetivo de entender sistemas neurais biológicos através de modelagem computacional.
  - Entretanto existe uma grande divergência entre os modelos biológicos neurais estudados em neurociência e as redes neurais usadas em aprendizagem de máquina.

# Redes Neurais

- O caráter “distribuído” das representações neurais permite robustez e degradação suave.

- Comportamento inteligente é uma propriedade “emergente” de um grande número de unidades simples ao contrário do que acontece com regras e algoritmos simbólicos.

# Cérebro Humano vs. Computadores

- Neurônios “ligam” e “desligam” em alguns milissegundos, enquanto o hardware atual faz essa mesma operação em nanossegundos.
- Entretanto, os sistemas neurais biológicos realizam tarefas cognitivas complexas (visão, reconhecimento de voz) em décimos de segundo.
- Sistema neural utiliza um “paralelismo massivo”
  - Cérebro humano tem $10^{11}$ neurônios com uma média de $10^{4}$ conexões cada.
  - Lentidão compensada por grande número de neurônios massivamente conectados.

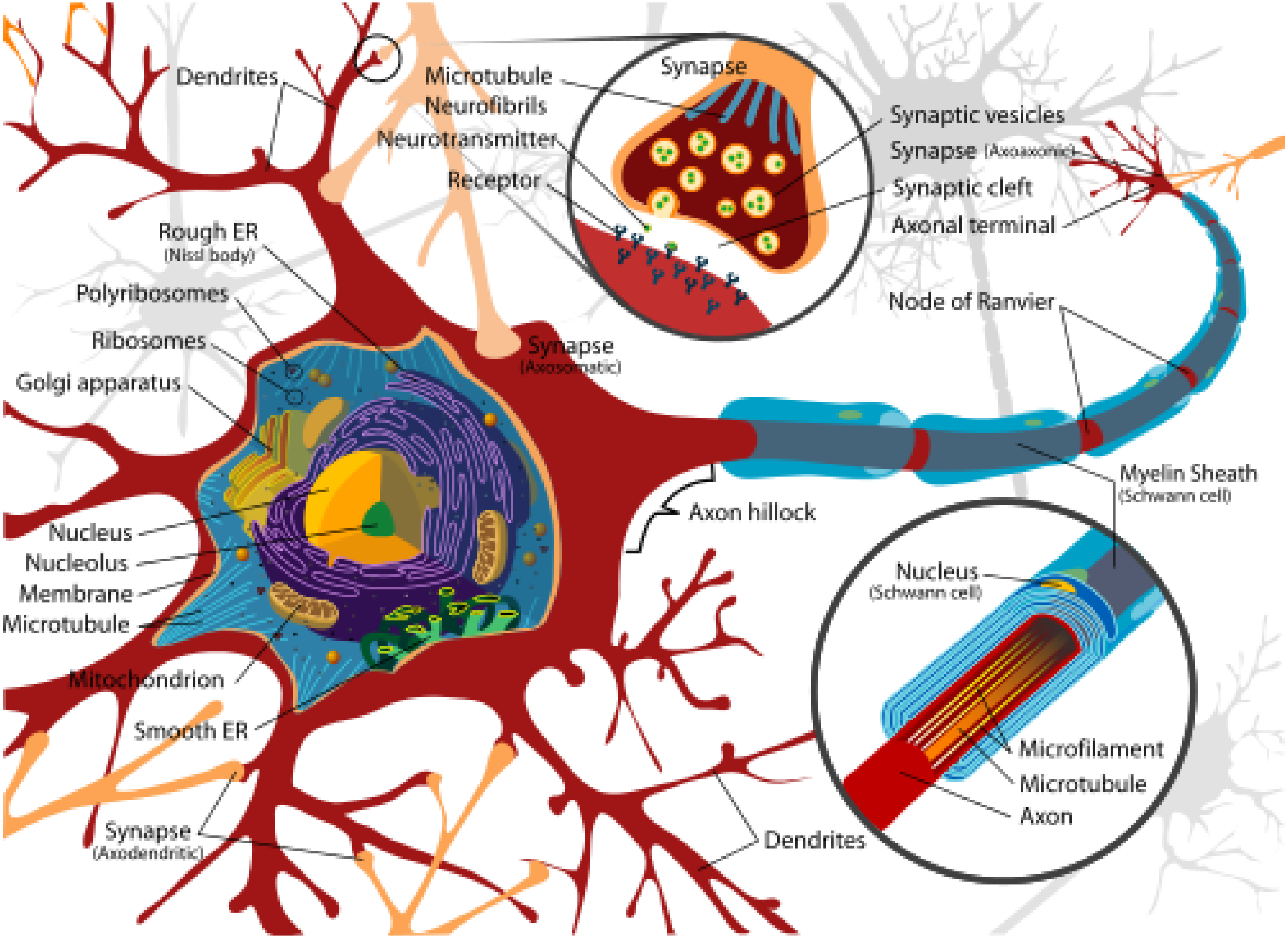
# Redes Neurais Artificiais

- Redes Neurais Artificiais (RNAs) são tentativas de produzir sistemas de aprendizado biologicamente realistas.
  - São baseadas em modelos abstratos de como pensamos que o cérebro (e os neurônios) funcionam
  - RNAs aprendem por exemplo
  - RNA = arquitetura (modelo/topologia) + processo de aprendizado

# Aprendizagem de Redes Neurais

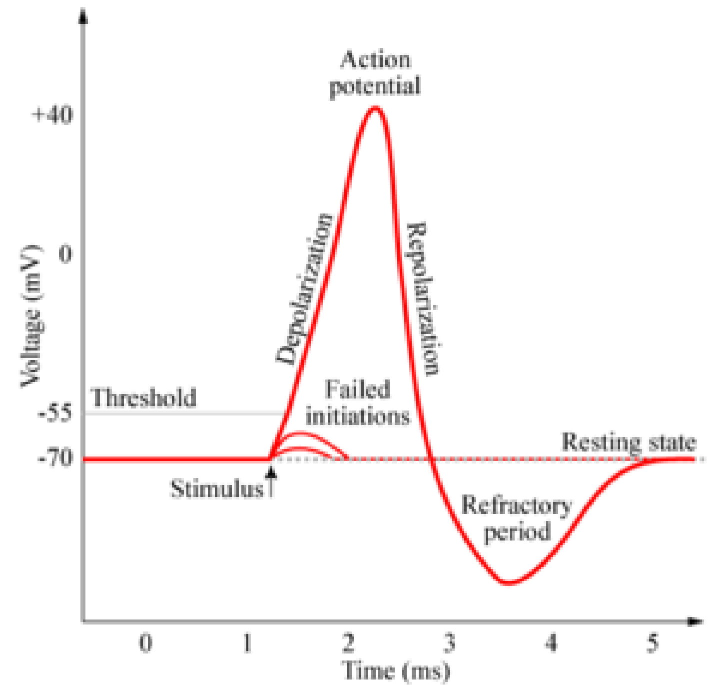
- Abordagem baseada numa adaptação do funcionamento de sistemas neurais biológicos.
- Perceptron: Algoritmo inicial pra aprendizagem de redes neurais simples (uma camada) desenvolvido nos anos 50.
- Retropropagação: Algoritmo mais complexo para aprendizagem de redes neurais de múltiplas camadas desenvolvido nos anos 80.

# Neurônios “Naturais”

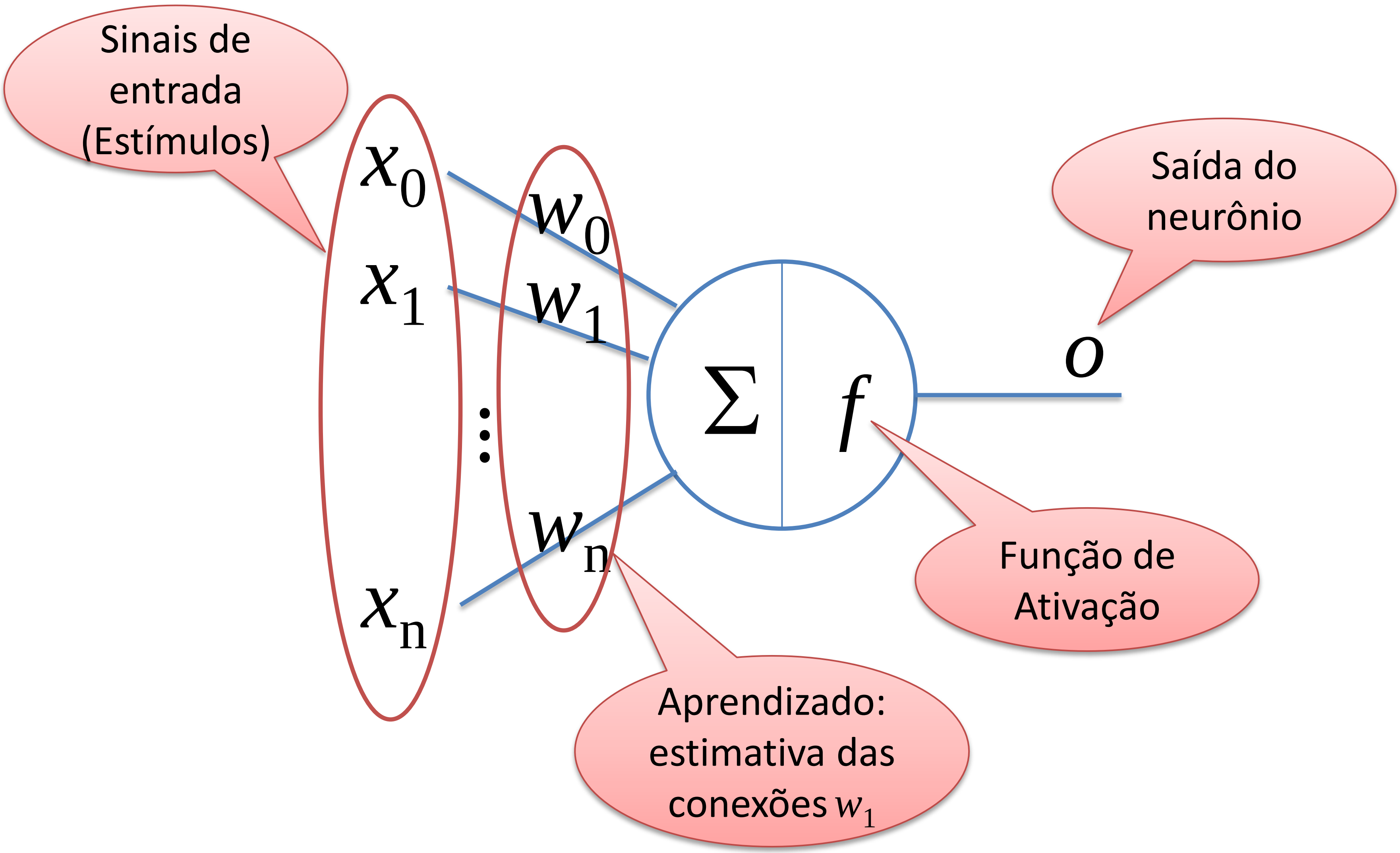


# Comunicação Neural

- Potencial elétrico através da membrana da célula exibe picos.
- Pico se origina no corpo celular, passa pelo axônio, e faz com que os terminais sinápticos soltem neurotransmissores.
- Neurotransmissores passam através das sinapses para os dendritos de outros neurônios.
- Neurotransmissores podem ser excitadores ou inibidores.
- Se a entrada total de neurotransmissores para um neurônio é excitatória e ultrapassa um certo limite, ele dispara (tem um pico).



# Neurônios “Artificiais”



# Aprendizagem de Redes Neurais

- Aprendizagem Hebbiana: Quando dois neurônios conectados disparam ao mesmo tempo, a conexão sináptica entre eles aumenta.
  - *"Neurons that fire together, wire together."*
- Sinapses mudam de tamanho e força com experiência

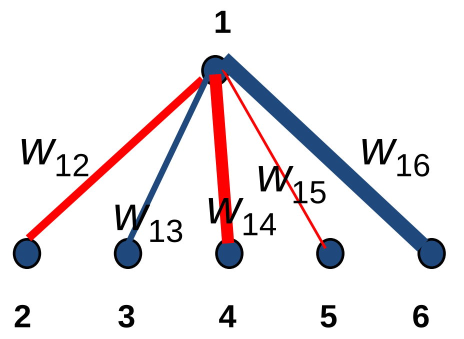
# Redes Neurais Artificiais

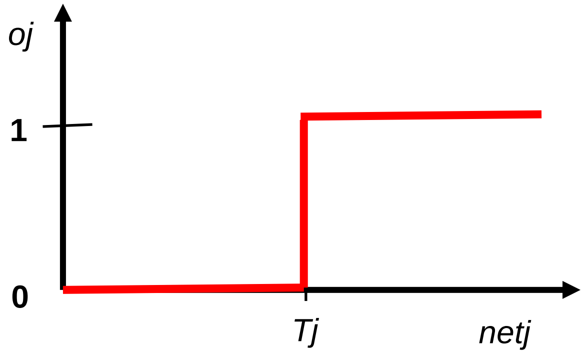
- A rede é modelada com um grafo onde as células são nós e as conexões sinápticas são arestas de um nó $i$ para um nó $j$, com pesos $w_{ij}$
- Entrada na célula:

$$net_j = \sum_i w_{ij} x_i + w_0$$

- Saída da célula:

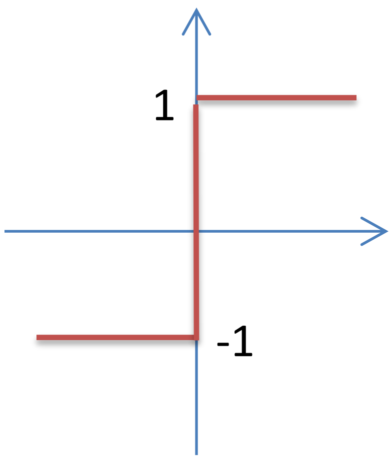
$$o_j = \begin{cases} 0, se\ net_j < T_j \\ 1, se\ net_j \geq T_j \end{cases}$$



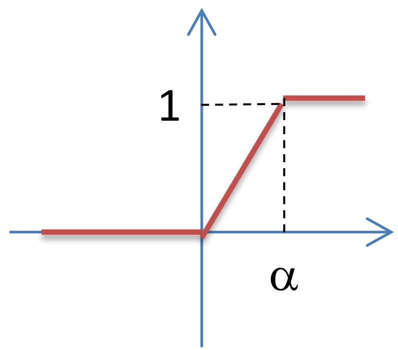



# Funções de Ativação

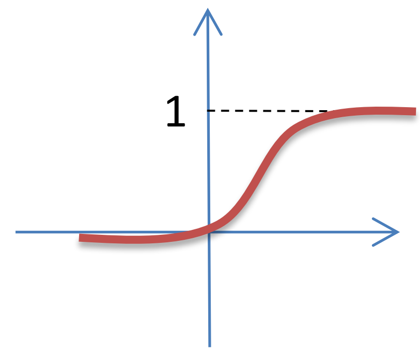
- Possíveis funções de ativação:




*Hard Limiter*
Degrau




*Threshold Logic*




*Sigmoid*

# PERCEPTRON

# Computação Neural

- **McCollough** e **Pitts** (1943) mostraram como neurônios simples desse tipo (chamados *perceptrons*) poderiam calcular funções lógicas e serem usados como máquinas de estado.
  - Podem ser usados para simular portas lógicas:
    - AND: Todos $w_{ji}$ são $T_i/n$, onde $n$ é o número de portas.
    - OR: Todos $w_{ji}$ são $T_i$
    - NOT: O limite é 0, entrada única com peso negativo
  - Podemos construir qualquer circuito lógico, máquina sequencial e computadores com essas portas.
    - Podemos representar qualquer função booleana usando uma rede com duas camadas de neurônios (AND-OR).

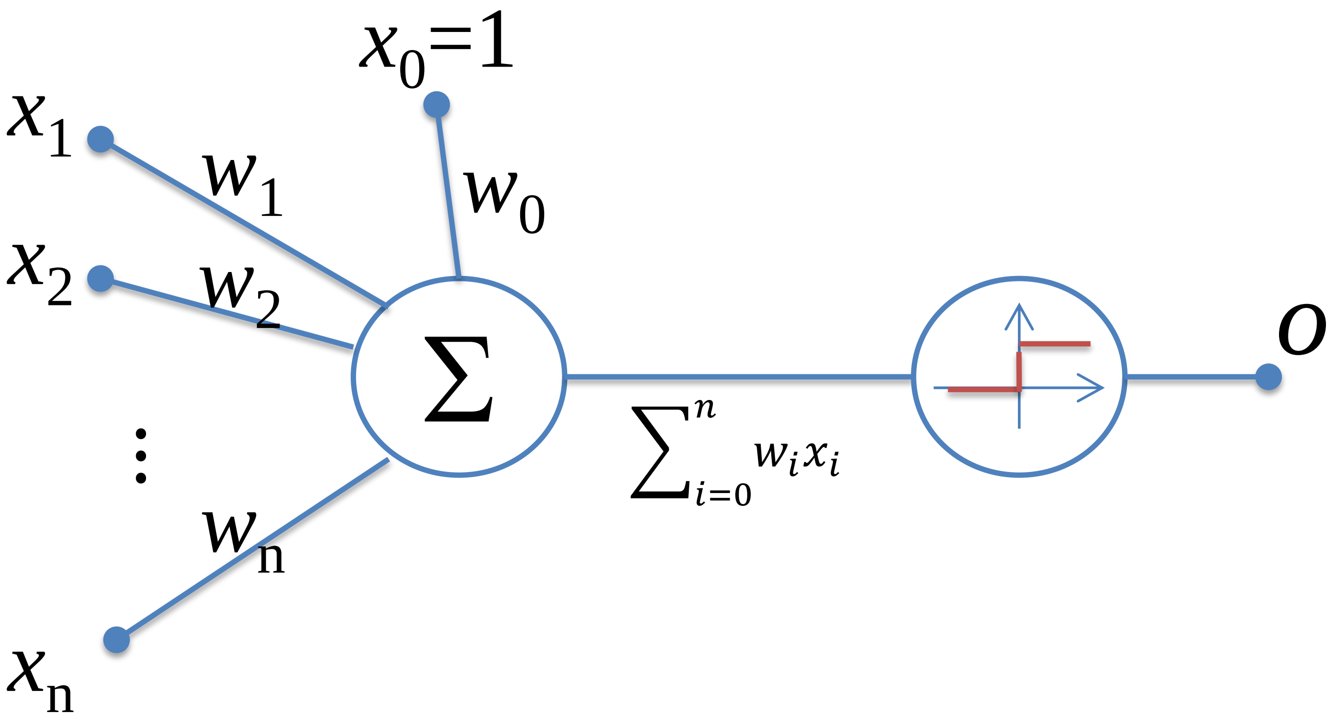
# Aprendizagem de Perceptrons

- Uma rede neural deve produzir, para cada conjunto de entradas apresentado, o conjunto de saídas desejado.
- O objetivo é aprender pesos sinápticos de tal forma que a unidade de saída produza a saída correta pra cada exemplo.
  - Quando a saída produzida é diferente da desejada, os pesos da rede são modificados

$$w_{t+1} = w_t + fator\_de\_correção$$

  - O algoritmo faz atualizações iterativamente até chegar aos pesos corretos.

# Modelo do Neurônio




Saída = 1, se $w_0 + w_1x_1 + \ldots + w_nx_n > 0$
Saída = 0, caso contrário

$w_0$ é o bias/threshold

# Algoritmo de Aprendizado

- Iterativamente atualizar pesos até a convergência.

```
Inicialize os pesos com valores aleatórios, pequenos ou iguais a zero
Até que as saídas de todos os exemplos de treinamento estejam corretos
  Para cada par de treinamento E
    Aplica-se um padrão com o seu respectivo valor desejado de saída
                                  (tᵢ) e verifica-se a saída da rede (oᵢ)
    Calcula-se o erro na saída, E = tᵢ - oᵢ
    Se E ≠ 0, atualize os pesos sinápticos e o threshold com o fator de
                                                        correção Δwᵢⱼ
```

- Cada execução do loop externo é tipicamente chamada de *época*.

# Regra de Aprendizagem de Perceptrons

- Atualizar pesos usando:

$$w_{ij} = w_{ij} + \Delta w_{ij}$$

$$\Delta w_{ij} = \eta x_i (t_j - o_j)$$

onde $\eta$ é a "taxa de aprendizagem"

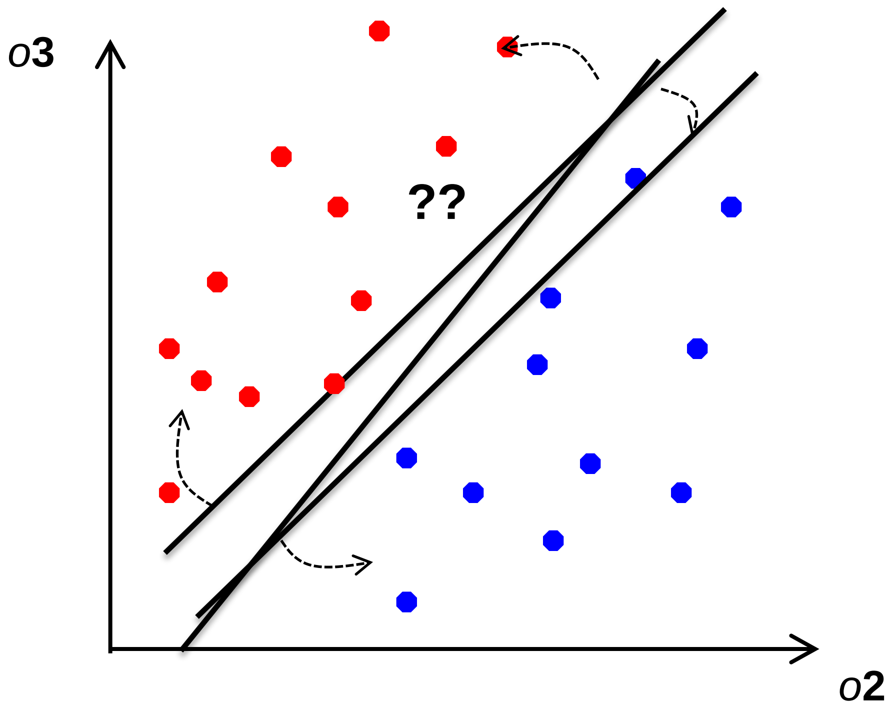
$t_j$ é a saída especificada para a unidade $j$

- O processo equivale a:
  - Se a saída estiver correta, não fazer nada.
  - Se a saída estiver alta, baixar os pesos das saídas ativas
  - Se a saída estiver baixa, aumenta pesos das saídas ativas,

**Erro:** diferença entre o esperado e o obtido

# Perceptron como Separador Linear

- Como o perceptron usa uma função de limite linear, ele procura por um separador linear que discrimine as classes.




$$w_{12}o_2 + w_{13}o_3 > T_1$$

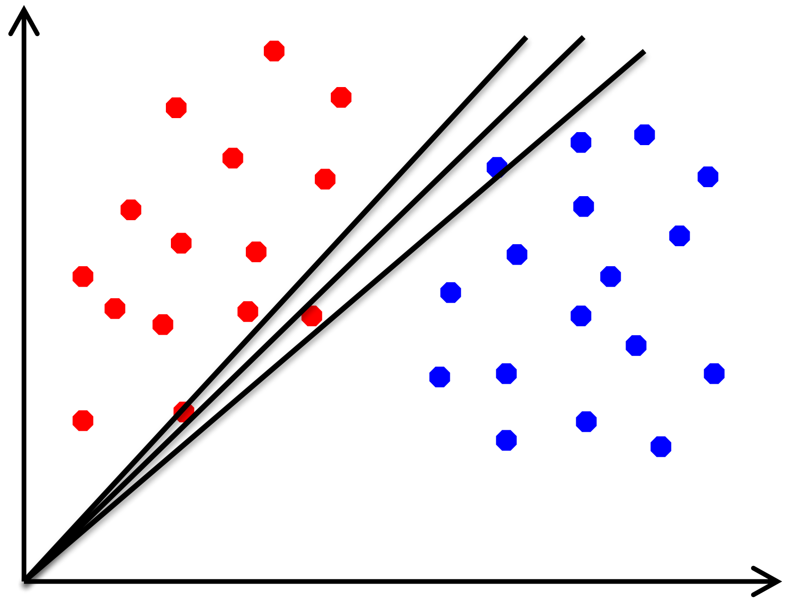
$$o_3 > -\frac{w_{12}}{w_{13}}o_2 + \frac{T_1}{w_{13}}$$

**ou *hiperplano* em um espaço *n*-dimensional**

# Para que serve o bias ($w_0$)?

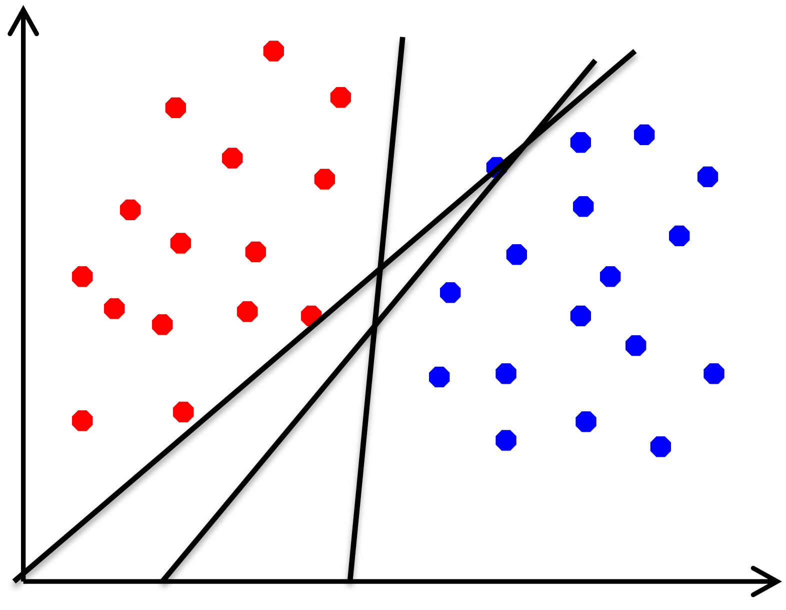
**Sem bias**

- Define um hiperplano passando pela origem

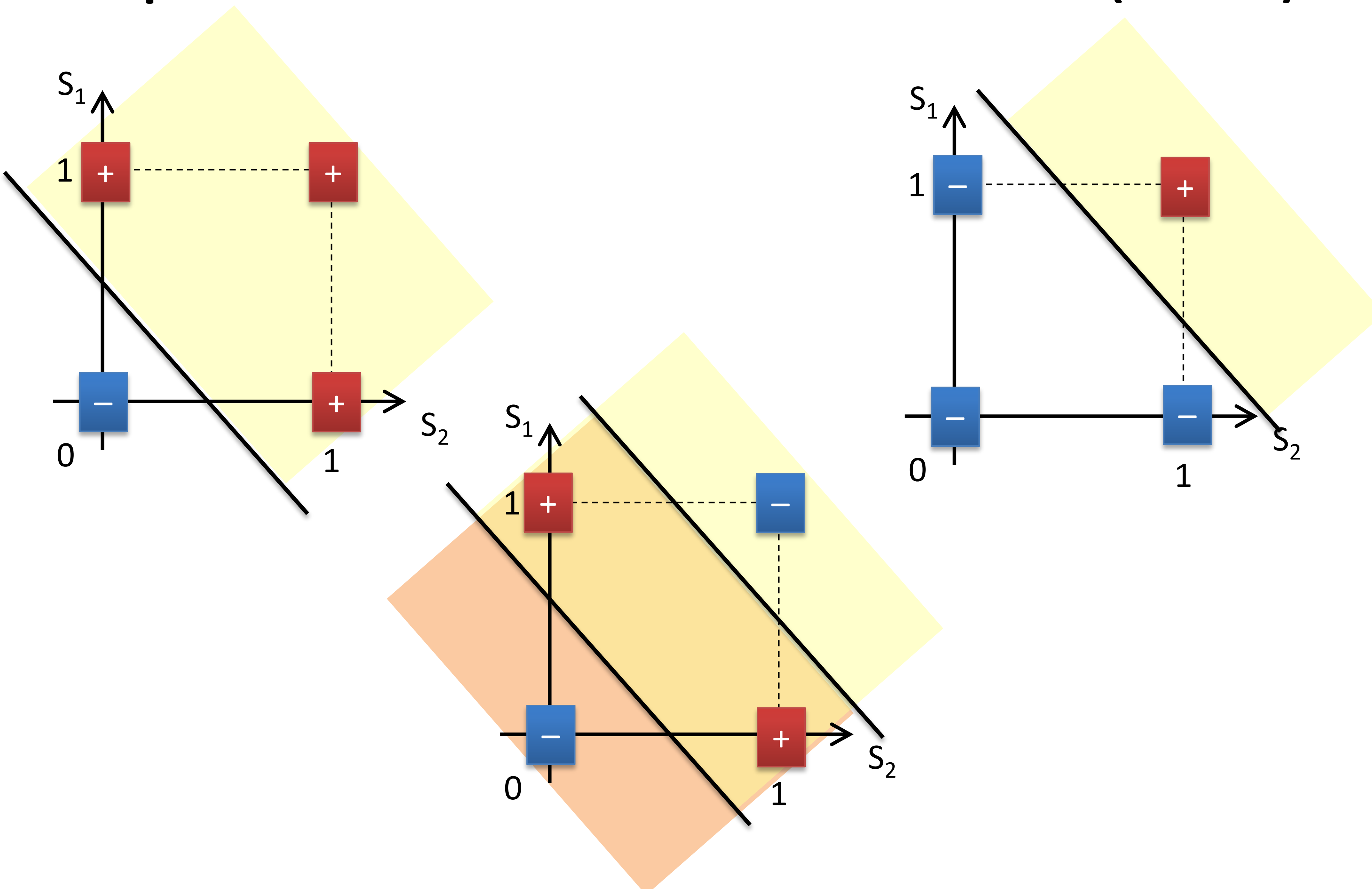


**Com bias**

- Permite que o hiperplano se desloque em relação a origem



# O problema do Ou-Exclusivo (XOR)



# Limitações do *Perceptron*

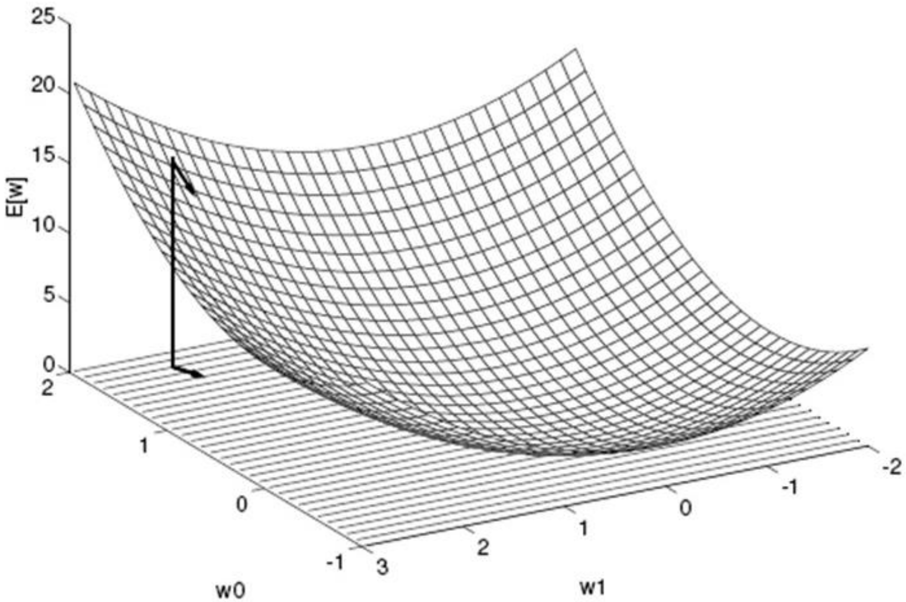
- Obviamente não pode aprender conceitos que não é capaz de representar.
- **Minksy** e **Papert** (1969) escreveram um livro analisando o perceptron e descrevendo funções que ele não podia aprender.
- Esses resultados desencorajaram o estudo de redes neurais e as regras simbólicas se tornaram o principal paradigma de IA.
  – Tempos depois, descobriu-se que as redes de uma única camada funcionam para exemplos linearmente separados, mas redes multi-camadas podem representar qualquer função, mesmo não-lineares.

# Teoremas

- **Teorema de convergência do perceptron**: Se os dados forem linearmente separáveis, então o algoritmo do perceptron irá corrigir para um conjunto consistente de pesos.
- **Teorema do ciclo do perceptron**: Se os dados não forem linearmente separáveis, o algoritmo irá repetir um conjunto de pesos e limites no final de uma época e, como consequência entra em um loop infinito.
  - Podemos garantir término do programa checando as repetições.

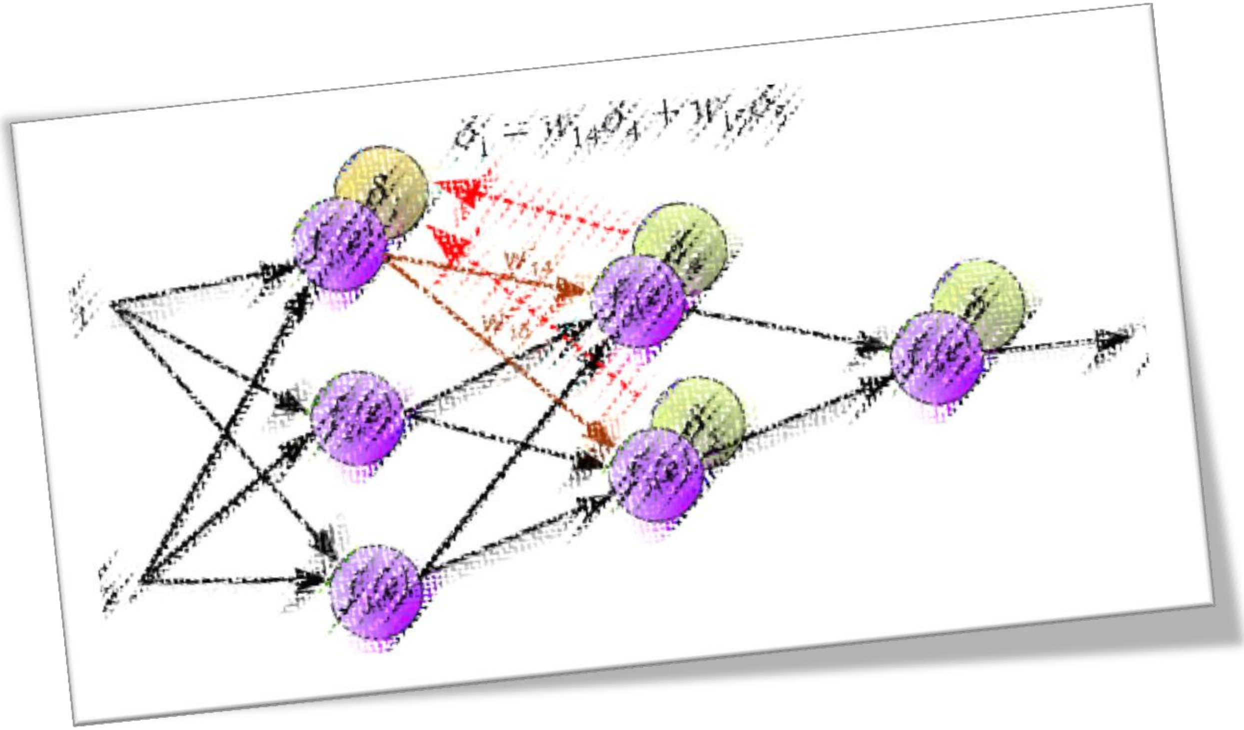
# *Perceptron* como Subida de Encosta

- O espaço de hipóteses é um conjunto de pesos e um limite.
- O objetivo é minimizar o erro de classificação no conjunto de treinamento.
- O perceptron efetivamente realiza uma subida de encosta (descida) neste espaço.
- Para um único neurônio, o espaço é bem comportado com um único mínimo.



# Desempenho do *Perceptron*

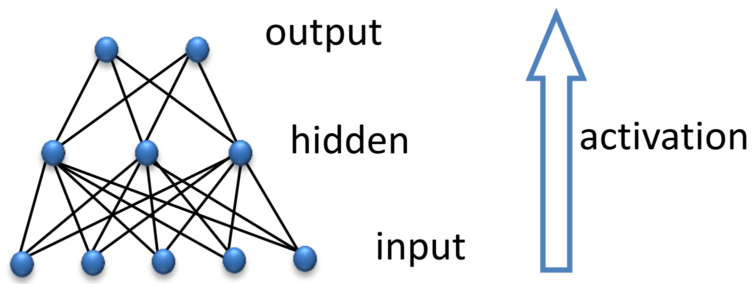
- Funções lineares são restritivas (bias ou viés alto) mas ainda razoavelmente expressivas; mais gerais que:
  - Conjuntiva pura
  - Disjuntiva pura
  - M-de-N (pelo menos M de um conjunto esperado de N características deve estar presente)
- Na prática, converge razoavelmente rápido para dados linearmente separáveis.
- Pode-se usar até resultados anteriores à convergência quando poucos *outliers* são classificados erroneamente.
- Experimentalmente, o Perceptron tem bons resultados para muitos conjuntos de dados.



# REDES MLP
# (*MULTI-LAYER PERCEPTRON)*

# Redes Multi-Camada

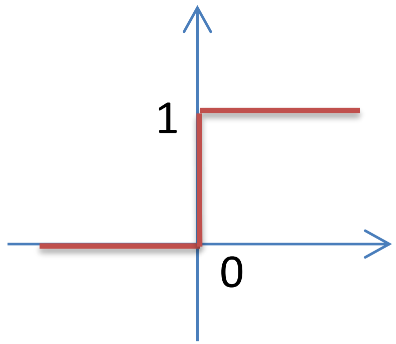
- Redes multi-camada podem representar funções arbitrárias, mas o aprendizado dessas redes era considerado um problema de difícil solução.
- Uma rede multi-camada típica consiste das camadas de  entrada, interna e saída, cada uma totalmente conectada à próxima, com a ativação indo pra frente.



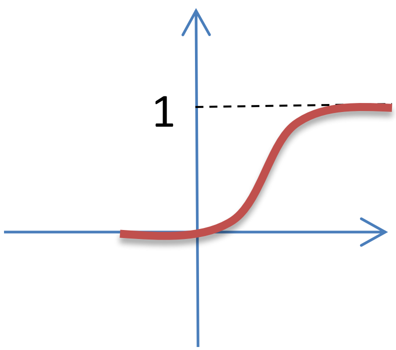

- Os pesos determinam a função calculada. Dado um número arbitrário de unidades internas, qualquer função booleana pode ser calculada com uma única camada interna.

# Gradiente em Redes MLP

- Para fazer descida de gradiente, precisamos que a saída de uma unidade seja uma função diferenciável da entrada e dos pesos.
- A função limite padrão não é diferenciável.




*Hard Limiter*
Degrau

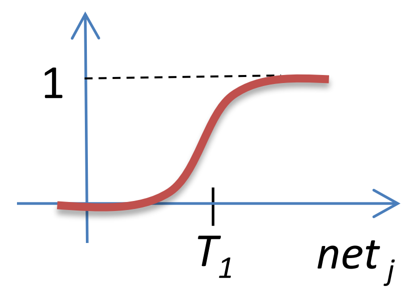



*Sigmoid*

# Função de Saída Diferenciável

- Precisamos de uma saída não-linear.
  - Uma rede multi-camada com saídas lineares só representa funções lineares (igual a um percéptron).
- Solução padrão é usar a função não-linear e diferenciável chamada de função "logística" ou sigmóide:

$$o_j = \frac{1}{1+e^{-(net_j - T_j)}}$$




- Também é possível utilizar *tanh* ou gaussiana.

# Descida de Gradiente

- Objetivo é minimizar o erro:

$$E(W) = \sum_{d \in D} \sum_{k \in K} (t_{kd} - o_{kd})^2$$

  onde $D$ é o conjunto de exemplos de treinamento, $K$ é o conjunto de unidades de saída, $t_{kd}$ e $o_{kd}$ são, respectivamente, a saída esperada e a saída atual para a unidade $k$ para o exemplo $d$.
- A derivada de uma unidade sigmoidal com relação a entrada da unidade é:

$$\frac{\partial o_j}{\partial net_j} = o_j(1 - o_j)$$

- Regra de aprendizado pra mudar os pesos e diminuir o erro é:

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$$

# *Backpropagation*

- Cada peso deve ser modificado usando: $\Delta w_{ji} = \eta \delta_j o_i$

$$\delta_j = o_j(1 - o_j)(t_j - o_j) \quad \text{se } j \text{ for uma unidade de saída}$$

$$\delta_j = o_j(1 - o_j)\sum_k \delta_k w_{kj} \quad \text{se } j \text{ for uma unidade interna}$$

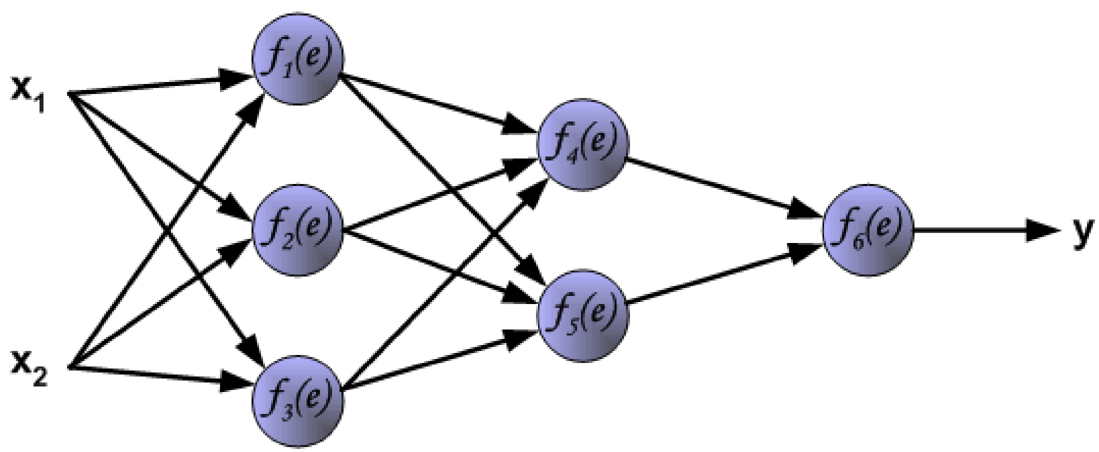
onde $\eta$ é uma constante chamada de taxa de aprendizado

$t_j$ é a saída correta para a unidade *j*

$\delta_j$ é a medida de erro para a unidade *j*

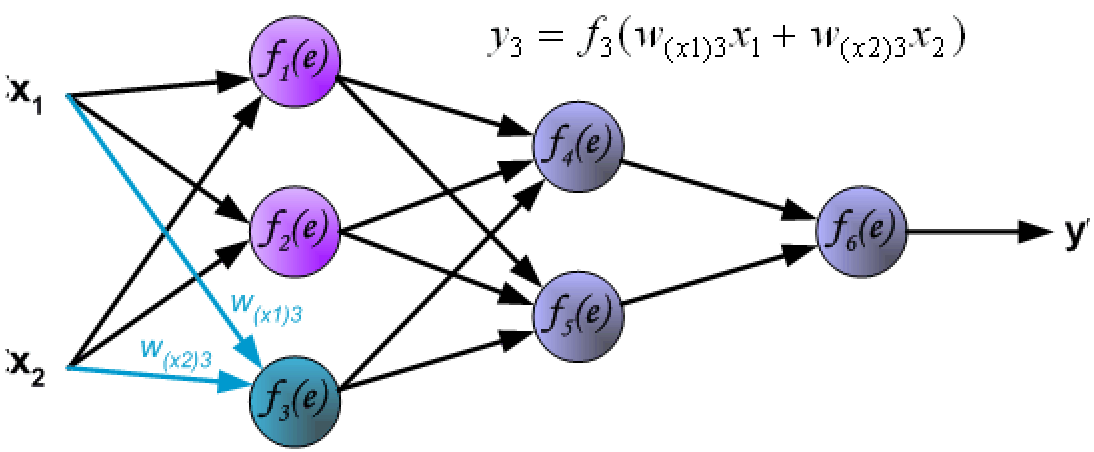
# *Backpropagation*

- Vejamos como ocorre o treinamento de uma rede neural multi-camadas usando o algoritmo *backpropagation*. Suponha uma rede de três camadas com duas entradas ($x_1$ e $x_2$) e uma saída ($y$)



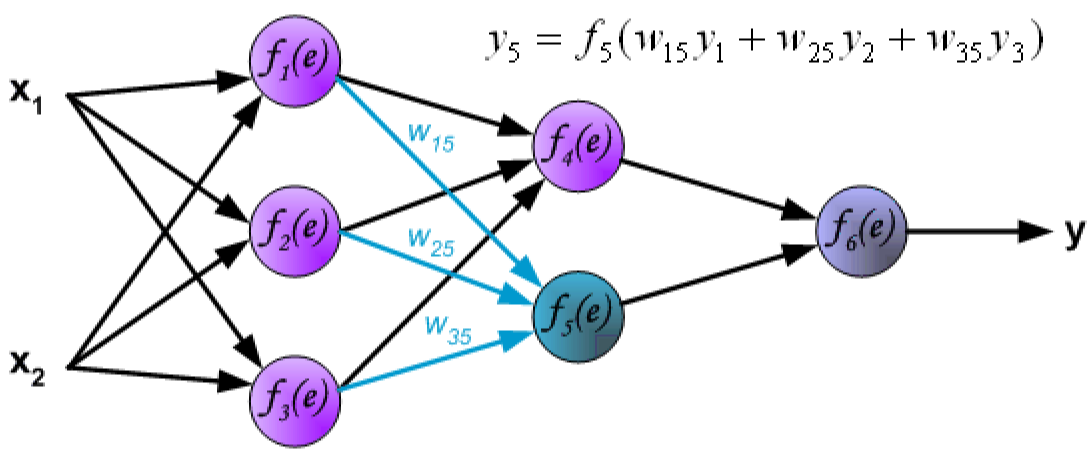
# *Backpropagation*

- Para cada caso de treinamento, calculamos os valores de saída de cada unidade da rede para os valores de entrada.



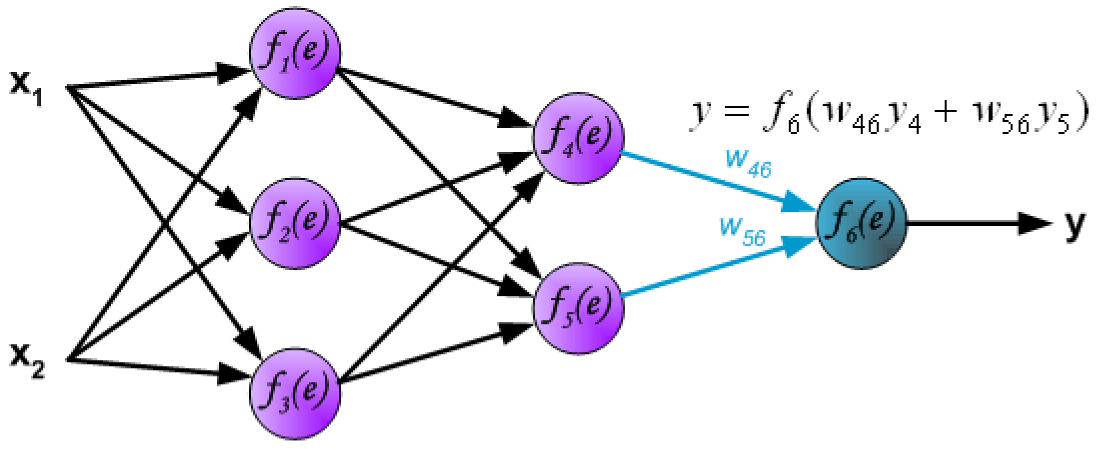
# *Backpropagation*

- Calculados os valores dos neurônios da camada de entrada, os sinais são então propagados para a camada intermediária.



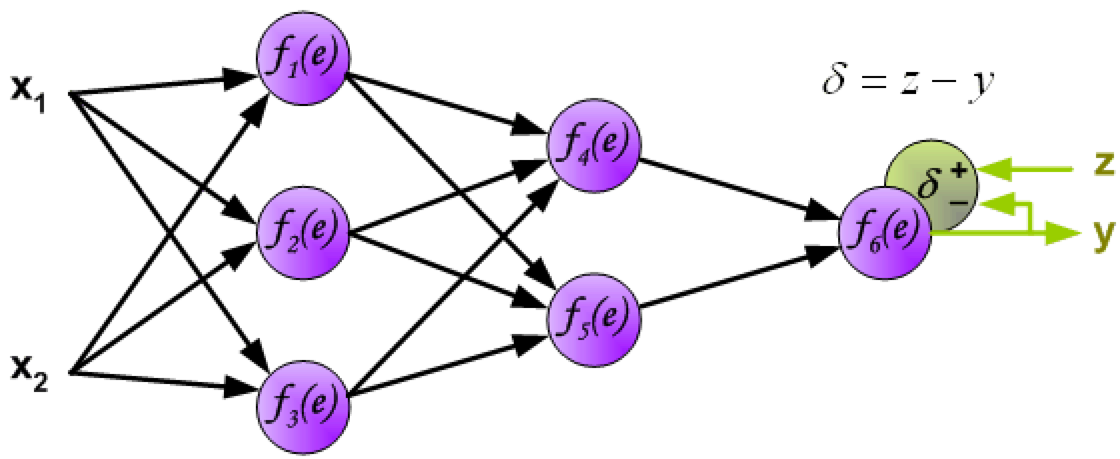
# *Backpropagation*

- Resta ainda uma camada para ser computada, assim propagamos os sinais da camada intermediária para a camada de saída.



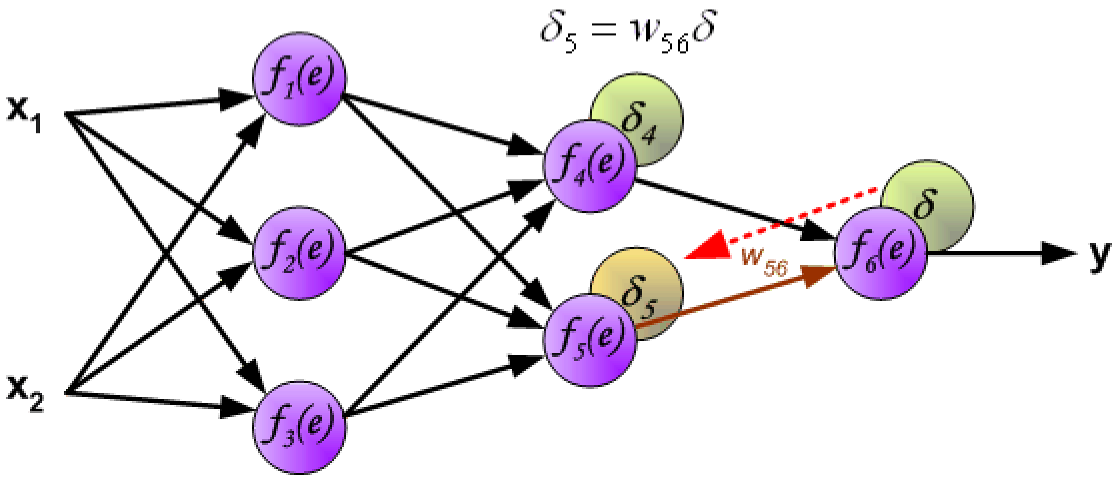
# *Backpropagation*

- O próximo passo do algoritmo é comparar a saída esperada com a saída obtida com. A diferença ($\delta$) é chamada de sinal de erro do neurônio da camada de saída.



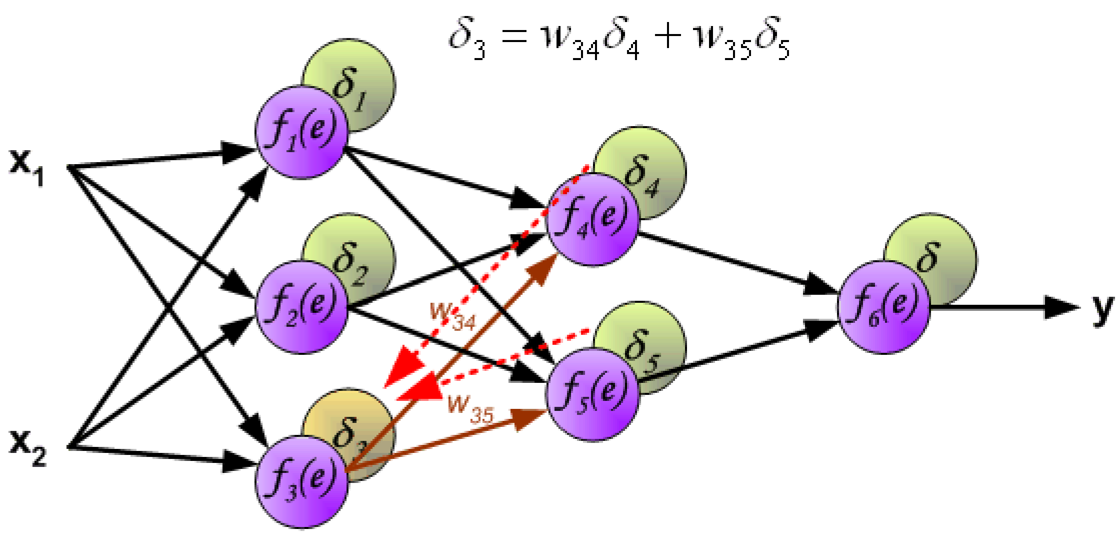
# *Backpropagation*

- Note que é impossível calcular o erro exato associado a camada interna. Assim a ideia é propagar o erro para todos os neurônios das camadas anteriores.



# *Backpropagation*

- Os coeficientes de peso ($w_i$) são os mesmos utilizados para calcular o valor de saída. A mesma técnica é utilizada para todas as camadas da rede.



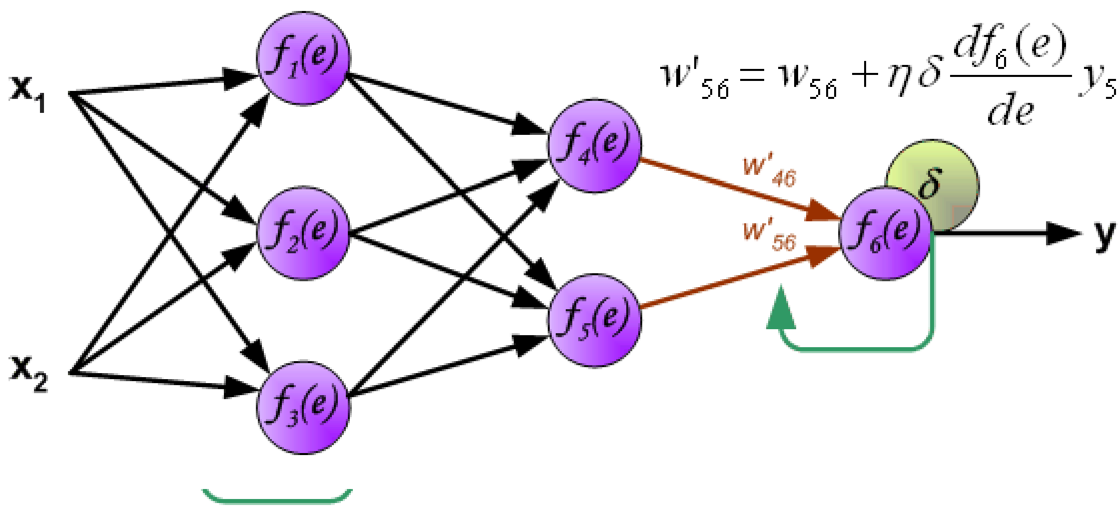
# *Backpropagation*

- Após todos os sinais de erro serem calculados, os coeficientes de entrada são ajustados para cada um dos neurônios.

$$w'_{15} = w_{15} + \eta\delta_5 \frac{df_5(e)}{de} y_1$$

$$w'_{46} = w_{46} + \eta\delta \frac{df_6(e)}{de} y_4$$

$$w'_{56} = w_{56} + \eta\delta \frac{df_6(e)}{de} y_5$$



# Algoritmo *Backpropagation*

Create the 3-layer network with *H* hidden units with full connectivity between layers. Set weights to small random real values.

Until all training examples produce the correct value (within ε), or mean squared error ceases to decrease, or other termination criteria:

- Begin epoch
- For each training example, *d*, do:
  - Calculate network output for *d*'s input values
  - Compute error between current output and correct output for *d*
  - Update weights by backpropagating error and using learning rule
- End epoch

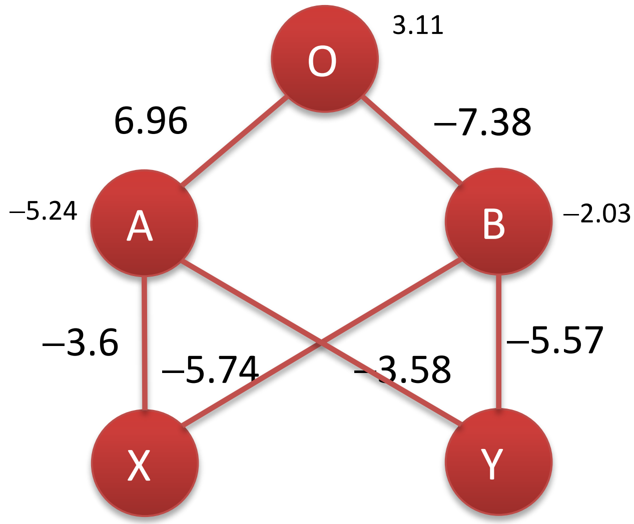
# Comentários sobre o algoritmo

- Não tem a convergência garantida – pode convergir para um ótimo local ou oscilar indefinidamente.
- Na prática, converge para um erro baixo para redes grandes com dados reais.
- Muitas épocas (milhares) podem ser necessárias, significando horas ou dias de treinamento para redes grandes.
- Para evitar problemas de mínimo local, executamos várias vezes com diferentes pesos aleatórios (*reinícios aleatórios*).
  - Pegamos resultado com menor erro de treinamento.
  - Podemos também construir um *ensemble* (possivelmente dando pesos de acordo com a acurácia).

# Poder de representação

- Funções booleanas: Qualquer função booleana pode ser representada por uma rede de duas camadas com número suficiente de unidades.
- Funções contínuas: Qualquer função contínua (limitada) pode ser aproximada arbitrariamente por uma rede de duas camadas.
  - Funções sigmoide funcionam como um conjunto de funções base.
- Funções arbitrárias: Qualquer função pode ser aproximada arbitrariamente por uma rede de três camadas.

# Exemplo: Rede XOR aprendida




Unidade interna A representa: $\neg(X \wedge Y)$

Unidade interna B representa: $\neg(X \vee Y)$

Saída O representa: $A \wedge \neg B = \neg(X \wedge Y) \wedge (X \vee Y)$

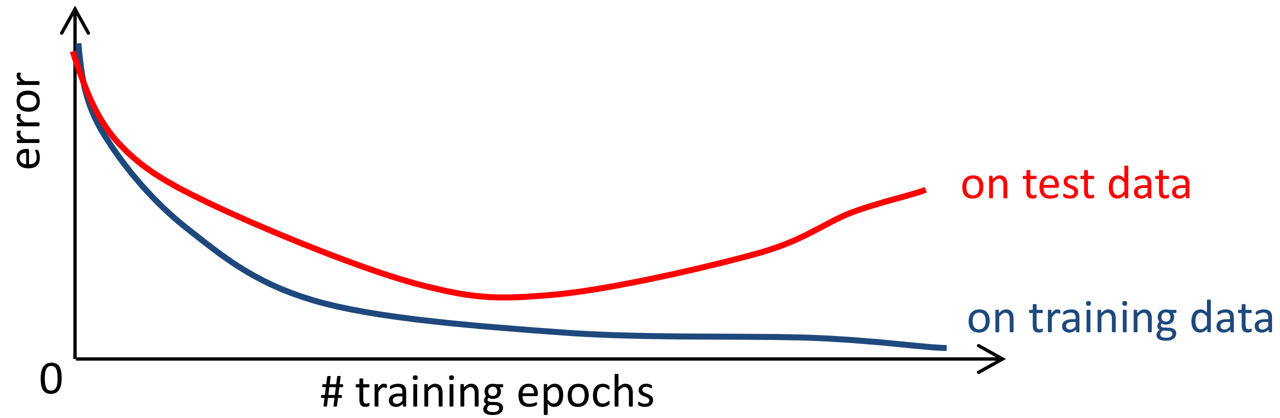
$= X \oplus Y$

# Representações nas Unidades Internas

- Unidade internas treinadas podem ser vistas como um novo conjunto de atributos que fazem o conceito ser linearmente separável.
- Em muitos domínios reais, unidades internas podem ser interpretadas como representando conceitos intermediários conhecidas como detectores de vogal ou detectores de forma, etc.
- Porém a camada interna pode ser vista também como uma representação distribuída da entrada,sem representar características conhecidas.

# Prevenção de Super-Ajuste

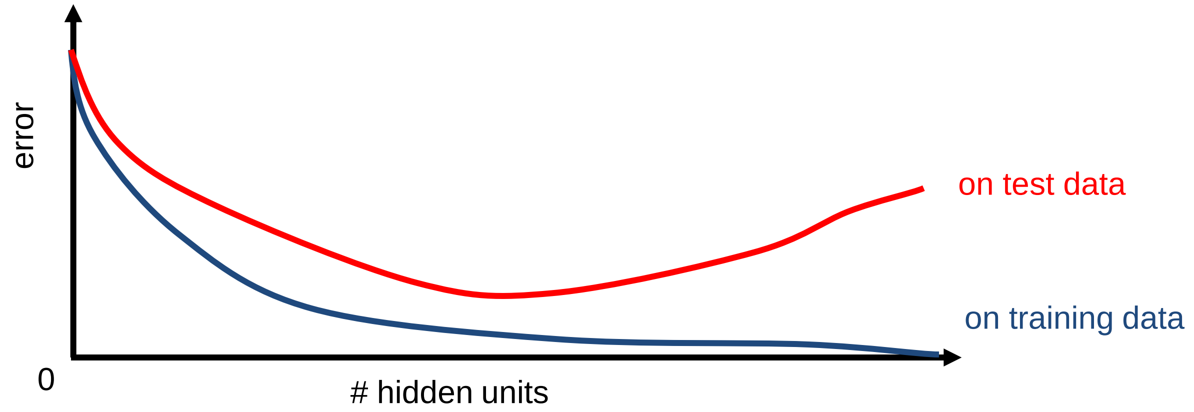
- Treinar por muitas épocas pode levar a super-ajuste.




- Usar conjunto de validação e parar quando erro começar a aumentar.
- Para não desperdiçar dados:
  - Usar validação cruzada para encontrar melhor número de épocas.
  - Treinar rede final usando todos os dados pelo mesmo número de épocas.

# Determinando o melhor número de unidades internas

- Poucas unidades impedem a rede de se adequar totalmente aos dados.
- Muitas unidades podem resultar em super-ajuste.




- Usar validação cruzada interna para determinar empiricamente o melhor número de unidades internas.

# Questões em Redes Neurais

- Métodos de treinamento mais eficientes:
  - Resilient propagation (Rprop)
  - Gradiente conjugado (usa segunda derivada)
- Aprender a melhor arquitetura:
  - Aumentar a rede até ela se ajustar os dados
    - Cascade Correlation
    - Upstart
  - Diminuir a rede até que ela não se ajuste mais aos dados
    - Optimal Brain Damage
- Redes recorrentes que usam retroalimentação podem aprender máquinas de estado finito através da "*retropropagação no tempo*"
- Algoritmos mais plausíveis biologicamente.
- Aprendizado não-supervisionado
  - Self-Organizing Feature Maps (SOMs)
- Aprendizado por reforço
  - Usa-se as redes para representar funções de valor.