Aula 02

Redes Neurais Profundas de Alimentação Direta (Feedforward)

Prof. Dr. Denis Henrique Pinheiro Salvadeo

Aula Anterior

- O que é Aprendizado Profundo?
- Perspectiva Histórica
- O Básico de Aprendizado de Máquina
- Desafios/Motivações para o Aprendizado Profundo
- Aplicações de Aprendizado Profundo

Aula de Hoje

- Redes FF
- Gradiente descendente
- Camadas de saída
- Camadas ocultas
- Backpropagation

DEFINIÇÃO

- Também chamadas de Redes Neurais de Alimentação Direta (*Feedforward*) ou Perceptrons Multicamadas (MLPs)

- Têm como objetivo **aproximar alguma função** f^*
  - Esta função pode representar qualquer tarefa

- Ex. para um classificador:

f^* mapeia uma entrada $\boldsymbol{x}$ em uma classe y

$$y = f^*(\boldsymbol{x})$$

- Define um mapeamento $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ e aprende o valor dos parâmetros $\boldsymbol{\theta}$ dos dados, de tal forma que resulte na melhor aproximação da função (critério)

$$f \approx f^*$$

DEFINIÇÃO

- Estes modelos são chamados FF, pois a informação flui apenas no sentido $\boldsymbol{x} \rightarrow \boldsymbol{y}$
  - A função é avaliada a partir das entradas $\boldsymbol{x}$, com a realização de computações intermediárias que definem f, gerando a saída $\boldsymbol{y}$.

- Não há conexões de **retroalimentação** (*feedback*)
  - As saídas não servirão como entrada para nenhuma parte do modelo
  - Extensão para incluir feedback definem as redes recorrentes (a serem vistas mais adiante no curso)

- Redes FF são de extrema importância, sendo a base para muitas aplicações comerciais
  - Redes convolucionais são uma tipo especializado de rede FF usadas para reconhecimento de objetos em imagens (p. ex.)
  - São uma pedra fundamental conceitual para a definição das redes recorrentes, que empoderaram as aplicações de PLN (p. ex.)

DEFINIÇÃO

- São chamadas redes porque são representadas pela composição de diferentes funções

- Modelo FF é associado a dígrafos acíclicos descrevendo como as funções são compostas

- P. ex. pode-se ter três funções $f^{(1)}, f^{(2)}$ e $f^{(3)}$ conectadas em uma cadeia para formar $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$

- Estruturas de cadeias são comuns em NN:
  - $f^{(1)}$ seria a primeira camada da rede
  - $f^{(2)}$ seria a segunda camada da rede
  - ...

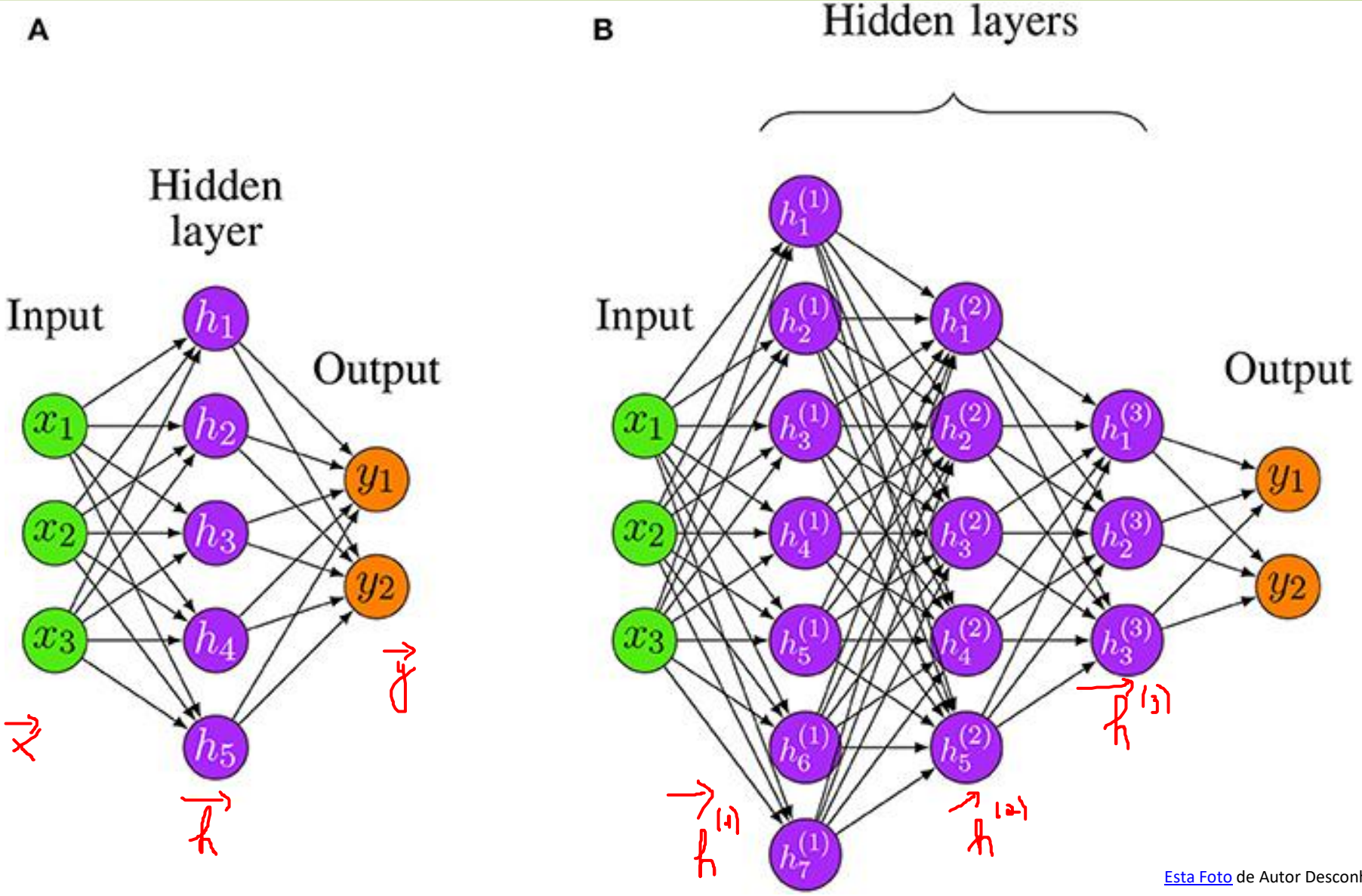
DEFINIÇÃO

- O comprimento da cadeia define a **profundidade** (D) do modelo
  - Daí surge o nome **deep learning**
  - $D = n - 1$ (número de camadas – uma camada de entrada)

- No treinamento da rede temos exemplos aproximados, ruidosos de $f^*(\boldsymbol{x})$, avaliados em pontos de treinamento diferentes (caso supervisionado)
  - Todo $\boldsymbol{x}$ é acompanhando por um rótulo $y \approx f^*(\boldsymbol{x})$

- O conjunto de treinamento não especifica o que cada camada deve fazer (apenas a camada de saída, dada uma entrada)
  - O algoritmo de treinamento deve decidir como usar $f^{(2)}, \ldots, f^{(n-1)}$ para produzir a saída desejada
    - i.e, como usar estas camadas para melhor implementar uma aproximação de f^*
    - Por isso, $f^{(2)}, \ldots, f^{(n-1)}$ são chamadas camadas ocultas ou escondidas

Representação Visual



Relação com a Neurociência

- São chamadas de **neurais** por serem inspiradas pela neurociência

- Camadas ocultas são normalmente vetorizadas
  - A **largura do modelo** é definida pela dimensionalidade das camadas ocultas (maior camada)
  - Cada elemento do vetor representa uma função análoga a um **neurônio**
  - Constituídas de **unidades** que agem em paralelo representando uma função de um vetor para escalar
    - Isto se assemelha a um neurônio que recebem entrada de muitas outras unidades e computa seu próprio valor de ativação
  - A escolha de $f^{(i)}(\boldsymbol{x})$ é inspirada por observações neurocientíficas de como os neurônios biológicos computam

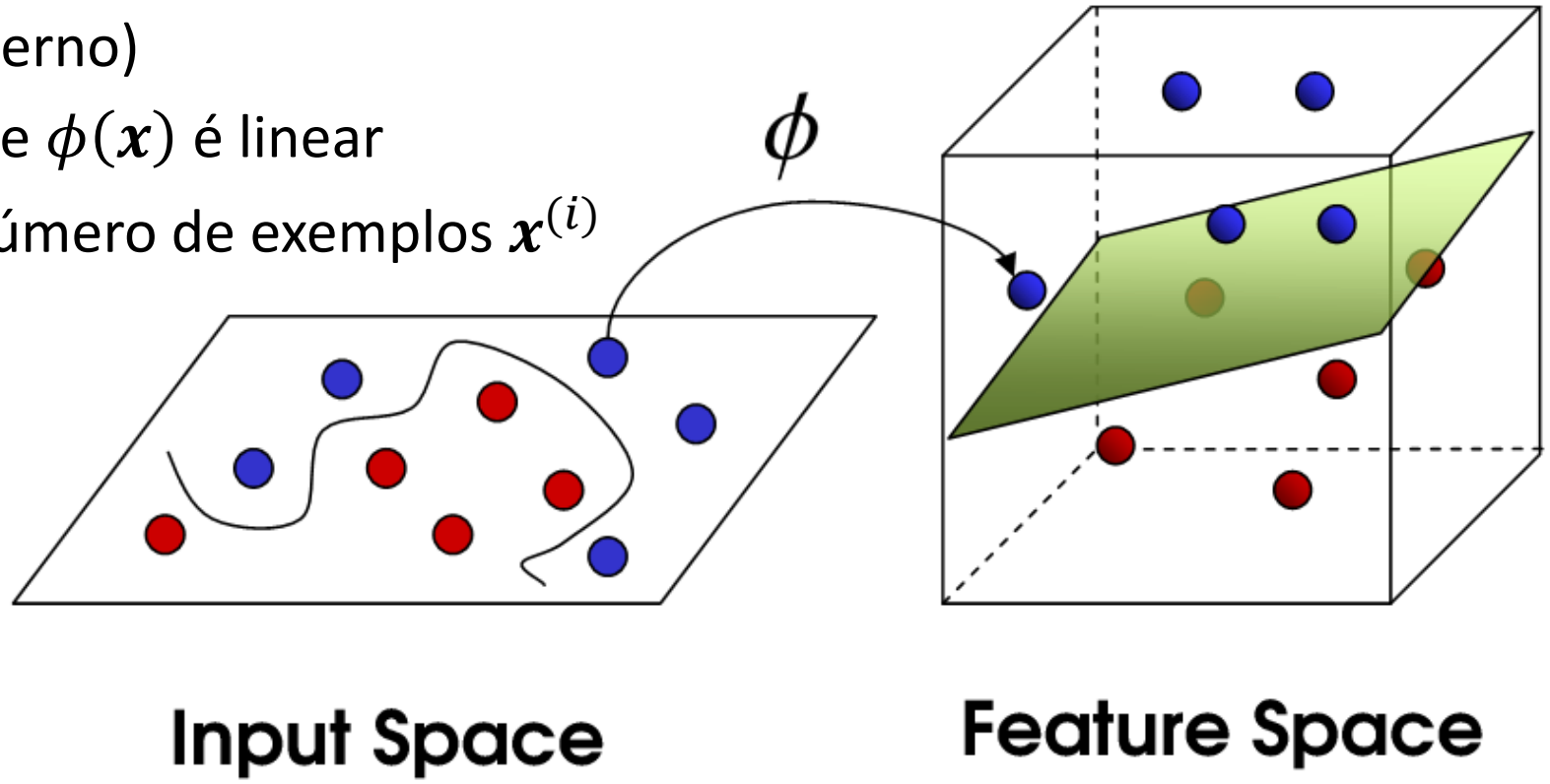
- Mas o objetivo não é modelar perfeitamente o cérebro
  - É obter generalização estatística das funções de aproximação, baseado no funcionamento biológico

Modelos Lineares

- Redes FF podem ser modeladas como modelos lineares
- Modelos lineares tal como regressão logística ou regressão linear podem se ajustar de modo confiável e eficiente, usando uma fórmula fechada ou por meio de uma otimização convexa
- Mas tem capacidade limitada para funções lineares
  - Não podem entender qualquer interação entre duas variáveis de entrada
- Para estender os modelos lineares para representar **funções não lineares** de $\boldsymbol{x}$, podemos aplicar o modelo linear para uma entrada transformada $\phi(\boldsymbol{x})$, onde ϕ é uma **transformação não linear**
  - Podemos pensar ϕ como fornecendo um conjunto de atributos descrevendo $\boldsymbol{x}$, ou como fornecendo uma nova representação para $\boldsymbol{x}$

- ϕ pode ser definido implicitamente usando um truque do kernel (como faz o SVM)
  - P. ex. em uma regressão linear, uma função $f(\boldsymbol{x}) = b + \sum_{i=1}^{m} \alpha_i \boldsymbol{x}^T \boldsymbol{x}^{(i)}$, em que podemos substituir $\boldsymbol{x}^T \boldsymbol{x}^{(i)}$ por um kernel $\mathrm{k}\left(\boldsymbol{x}, \boldsymbol{x}^{(i)}\right) = \phi(\boldsymbol{x}).\phi(\boldsymbol{x}^{(i)})$

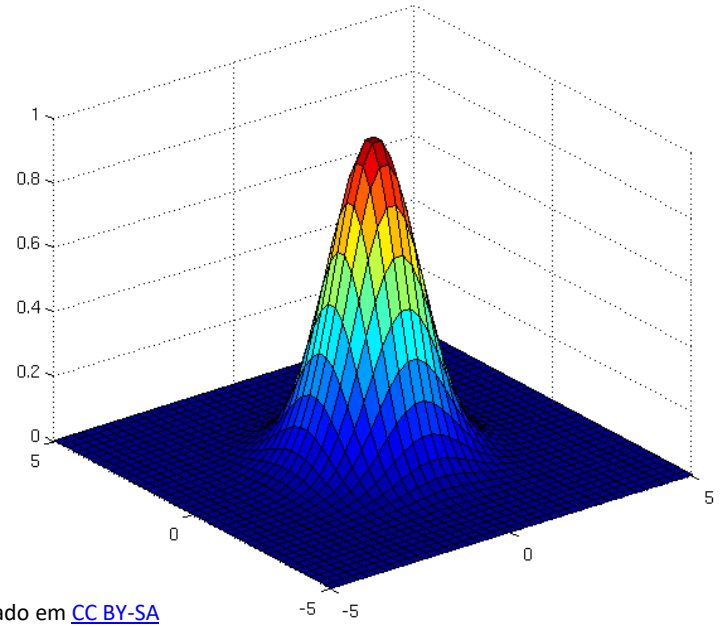
    (produto interno)
  - Relacionamento entre $f(\boldsymbol{x})$ e $\phi(\boldsymbol{x})$ é linear
  - Custo elevado, com maior número de exemplos $\boldsymbol{x}^{(i)}$



Como Escolher o Mapeamento ϕ?

1. Usando um ϕ muito genérico,
   - Ex. kernel de base radial (RBF): $k(\boldsymbol{u}, \boldsymbol{v}) = \mathcal{N}(\boldsymbol{u} - \boldsymbol{v}; 0, \sigma^2 \boldsymbol{I})$
   - Se $\phi(\boldsymbol{x})$ tem dimensão bastante alta, possuem boa capacidade de ajuste ao conjunto de treinamento, mas generalização pobre ao conjunto de teste
   - Muitos mapeamentos genéricos são baseados somente no princípio de suavização local, não codificando informação *a priori* para resolver problemas mais complexos

2. Projetar manualmente
   - Prática dominante antes do deep learning
   - Esforço humano especializado e específico de domínio
   - Pouca transferência entre os domínios



Como Escolher o Mapeamento ϕ?

3. Aprender ϕ (estratégia do deep learning)
   - Modelo: $y = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{\omega}) = \phi(\boldsymbol{x}; \boldsymbol{\theta})^T \boldsymbol{\omega}$
   - $\boldsymbol{\theta}$ são parâmetros que usamos para aprender ϕ de uma classe ampla de funções
   - $\boldsymbol{\omega}$ são parâmetros que mapeiam de $\phi(\boldsymbol{x})$ para a saída desejada
   - Em redes FF, ϕ define uma camada oculta
   - Única abordagem que abandona a questão da convexidade do problema de treinamento
     - Mas os benefícios superam os danos
   - A representação é parametrizada como $\phi(\boldsymbol{x}; \boldsymbol{\theta})$, sendo que um algoritmo de otimização é usado para encontrar $\boldsymbol{\theta}$ que correspondem a uma boa representação
   - Se beneficia de ambas as abordagens anteriores
     - É altamente genérica: $\phi(\boldsymbol{x}; \boldsymbol{\theta})$
     - Pode codificar o conhecimento de especialistas para ajudar na generalização, por projetar funções $\phi(\boldsymbol{x}; \boldsymbol{\theta})$ que são esperadas funcionar bem (necessita encontrar a família certa de funções gerais, mais do que a função certa)
   - Seguem o princípio geral de melhorar os modelos por aprender atributos

Como Projetar uma Rede FF?

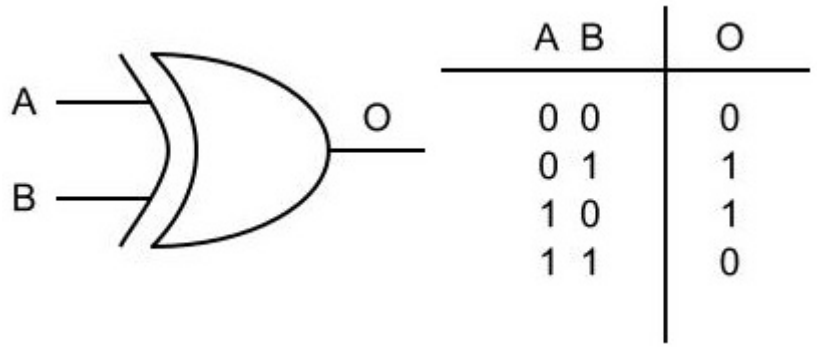
- Para treinar uma rede FF devemos escolher (mesmas decisões para um modelo linear):
  - Otimizador (método para encontrar os parâmetros adequados)
  - Função de custo (medida de erro de saída)
  - Forma das unidades de saída (depende do tipo do problema ou tipo de dado esperado)

- Redes FF introduziram o conceito de camada oculta
  - Exigirá a escolha de **funções de ativação** para realizar o cálculo dos valores destas camadas

- Definir a arquitetura da rede
  - Quantas camadas na rede
  - Como estas camadas estão conectadas
  - Quantas unidades em cada camada

- Para aprender em redes profundas, precisaremos calcular gradientes de funções complicadas
  - Algoritmo de **backpropagation** e suas generalizações ajudarão nesta tarefa, fazendo estes cálculos de modo eficiente

EXEMPLO: APRENDENDO A OPERAÇÃO XOR (OU EXCLUSIVO)



A	B	O
0	0	0
0	1	1
1	0	1
1	1	0

- Lembrando que o modelo tenta tornar $f(\boldsymbol{x};\boldsymbol{\theta}) \approx f^*(\boldsymbol{x})$
- Mais do que generalização estatística vamos focar em que a rede gere exatamente os resultados esperados
- Vamos usar uma função de custo (função de perda) MSE, avaliada sobre o conjunto todo

$$J(\boldsymbol{\theta}) = \frac{1}{4}\sum_{\boldsymbol{x}\in\mathbb{X}}(f^*(\boldsymbol{x}) - f(\boldsymbol{x};\boldsymbol{\theta}))^2$$

Exemplo: Aprendendo a Operação XOR (ou exclusivo)

- Vamos escolher o modelo $f(\boldsymbol{x}; \boldsymbol{\theta})$ como linear, com $\boldsymbol{\theta} = \{\boldsymbol{\omega}, b\}$

$$f(\boldsymbol{x}; \boldsymbol{\omega}, b) = \boldsymbol{x}^T \boldsymbol{\omega} + b$$

- Usando equações normais (**mostrar**), nós podemos minimizar $J(\boldsymbol{\theta})$ em uma fórmula fechada com relação a $\boldsymbol{\omega}$ e b
  - $\boldsymbol{\omega} = \mathbf{0}$ e $b = \frac{1}{2}$
  - O modelo sempre resulta em 0,5

(para qualquer entrada)

$$X = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

$$w = (X^T X)^{-1} X^T y = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \frac{1}{2} \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \\ b \end{bmatrix}$$

XOR is not linearly separable

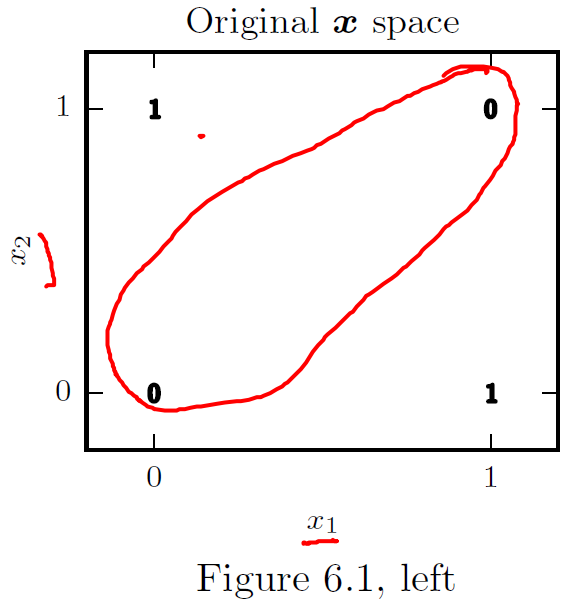


Figure 6.1, left

(Goodfellow 2017)

EXEMPLO: APRENDENDO A OPERAÇÃO XOR (OU EXCLUSIVO)

- Uma forma de resolver este problema é usar um modelo que aprende em um espaço de atributos diferente no qual um modelo linear permite representar a solução

- Utilizaremos uma rede FF com 1 camada oculta com 2 unidades ocultas

$$\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$$

- A camada de saída (regressão linear aplicada a $\boldsymbol{h}$)

$$y = f^{(2)}(\boldsymbol{h}; \boldsymbol{\omega}, b)$$

- Funções encadeadas resultando em

$$f(\boldsymbol{x}) = \boldsymbol{x}^T \boldsymbol{W} \boldsymbol{\omega}$$

$f(\boldsymbol{x}) = \boldsymbol{x}^T \boldsymbol{\omega}'$, em que $\boldsymbol{\omega}' = \boldsymbol{W}\boldsymbol{\omega}$

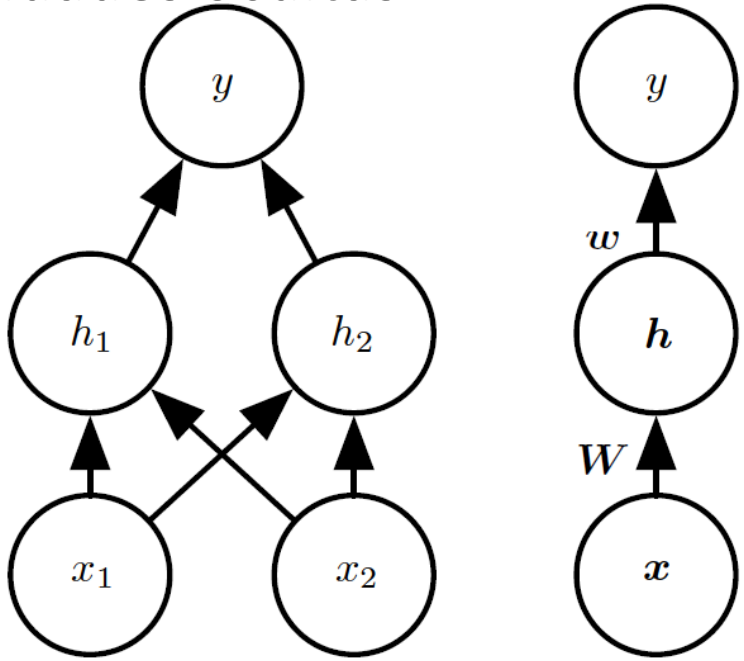


Figure 6.2

EXEMPLO: APRENDENDO A OPERAÇÃO XOR (OU EXCLUSIVO)

- Devemos usar uma função não linear para descrever os atributos

- Em NN é comum usarmos transformações afins controladas por parâmetros aprendidos, seguido por uma função não linear fixa g (função de ativação)

$$\boldsymbol{h} = g(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c})$$

  - $\boldsymbol{W}$ fornece os pesos de uma transformação linear e $\boldsymbol{c}$ são os vieses (bias)

- Normalmente, a função de ativação é aplicada por elementos

$$h_i = g\left(\boldsymbol{x}^T\boldsymbol{W}_{:,i} + c_i\right)$$

- Atualmente, uma escolha padrão é usar as **rectified linear unit (ReLU)**

$$g(z) = \max\{0, z\}$$

- O modelo de rede FF completa neste exemplo seria

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{\omega}, b) = \boldsymbol{\omega}^T \max\{0, \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c}\} + b$$

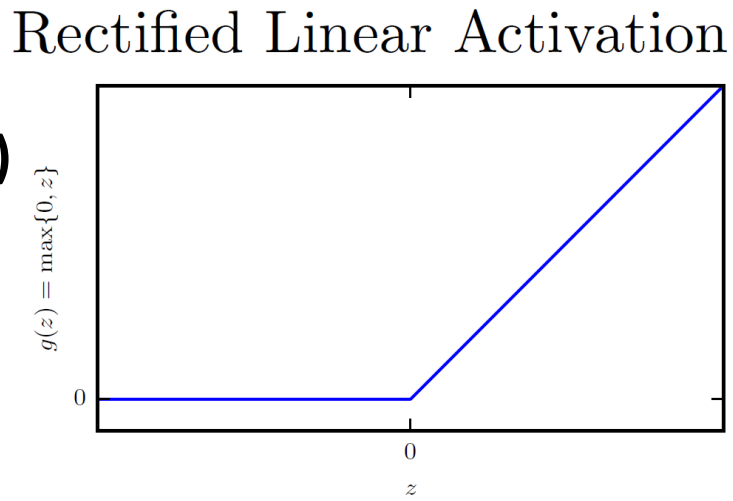


Figure 6.3

Exemplo: Aprendendo a Operação XOR (ou exclusivo)

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{\omega}, b) = \boldsymbol{\omega}^T \max\{0, \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{c}\} + b$$

- Considerando os seguintes parâmetros temos uma solução para o XOR

$$b = 0$$

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix},$$

$$\boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix},$$

$$\boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix},$$

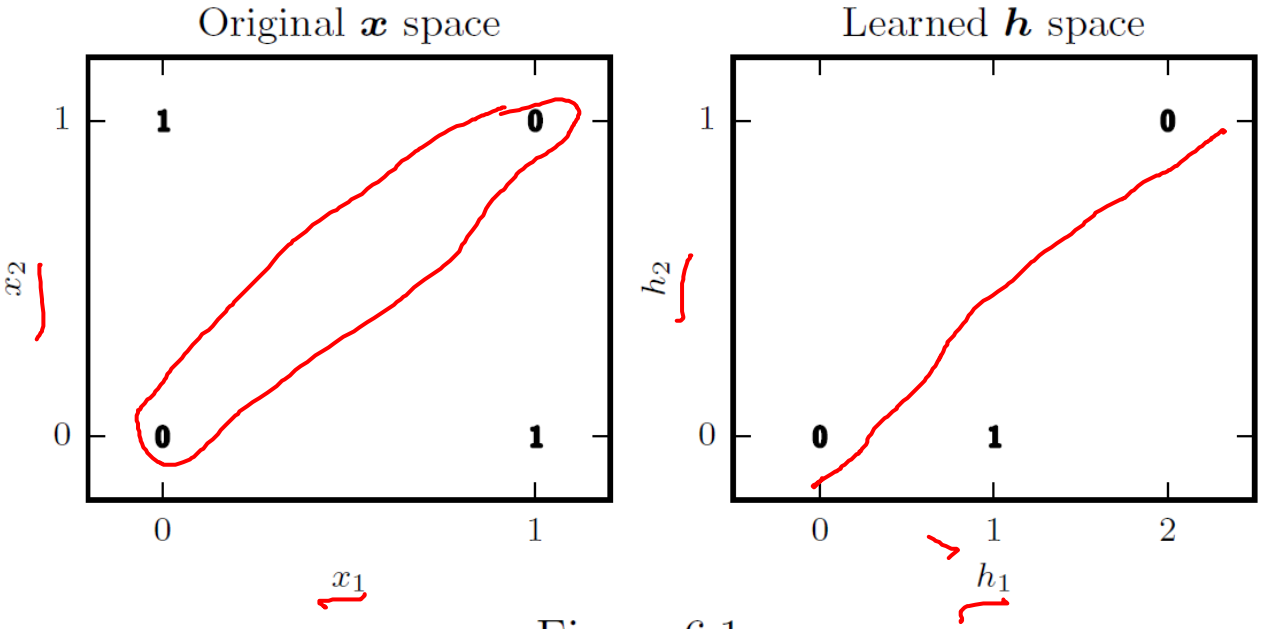


Figure 6.1

Exemplo: Aprendendo a Operação XOR (ou exclusivo)

- Neste exemplo simples, cuja solução foi especificada diretamente, o erro obtido foi 0

- No mundo real, podemos ter bilhões de parâmetros do modelo e bilhões de exemplos de treinamento
  - Isto inviabiliza enxergar uma solução direta

- Na prática, um algoritmo de otimização baseado em gradiente pode encontrar parâmetros que produzam pouco erro
  - No exemplo do XOR, a solução atinge o mínimo global da função de perda
  - O gradiente descendente poderia convergir para este ponto
  - O ponto de convergência do gradiente depende os valores iniciais dos parâmetros
  - Gradiente descendente não encontra solução normalmente clara, de fácil entendimento, avaliada em valores inteiros como visto no exemplo do XOR

Gradiente Descendente

- Muitos algoritmos de aprendizado profundo envolvem otimização
  - Otimização refere-se à tarefa de minimizar ou maximizar uma função $f(\boldsymbol{x})$ por alterar $\boldsymbol{x}$
  - Maximização pode ser executada via um algoritmo de minimização por minimizar $-f(\boldsymbol{x})$
  - $f(\boldsymbol{x})$ é chamada de **função objetivo** ou **critério**

- Normalmente estamos interessados em minimizar $f(\boldsymbol{x})$
  - Neste caso, podemos chamá-la também como **função de custo** (cost function), **função de perda** (loss function) ou **função de erro**

- O interesse é encontrar $\boldsymbol{x}^* = \arg\min f(\boldsymbol{x})$

- Suponha que temos $y = f(x)$
  - A **derivada** da função (denotada $f'(x)$ ou $\frac{dy}{dx}$) fornecem a inclinação de $f(x)$ no ponto x
  - Nos permite medir como uma pequena mudança na entrada afeta a saída $f(x + \epsilon) \approx f(x) + \epsilon f'(x)$
  - Útil para minimização, pois nos diz como mudar x para fazer um pequeno melhoramento em y
  - $f(x - \epsilon \text{sign}(f'(x))) < f(x)$ para um ϵ bastante pequeno
    - Assim, podemos reduzir $f(x)$ movendo x em pequenos passos com o sinal oposto da derivada (**gradiente descente**)

Gradiente Descendente

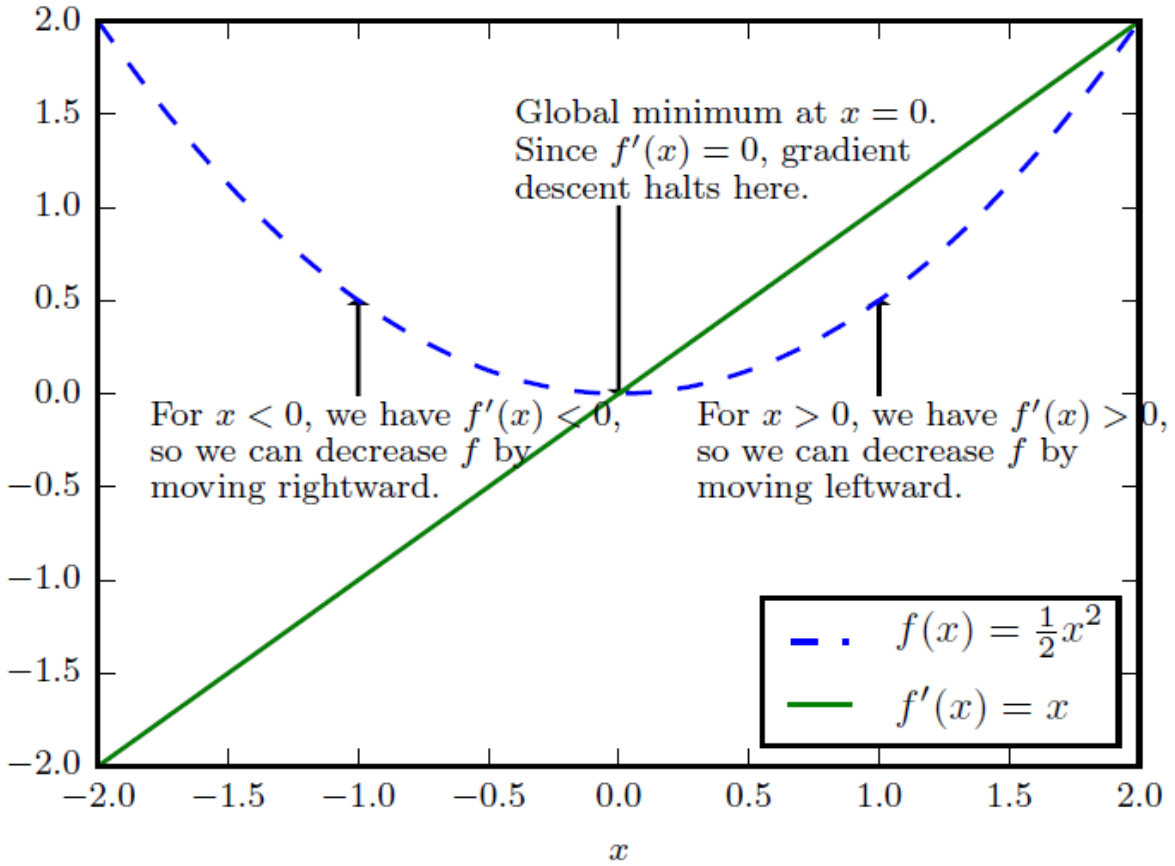
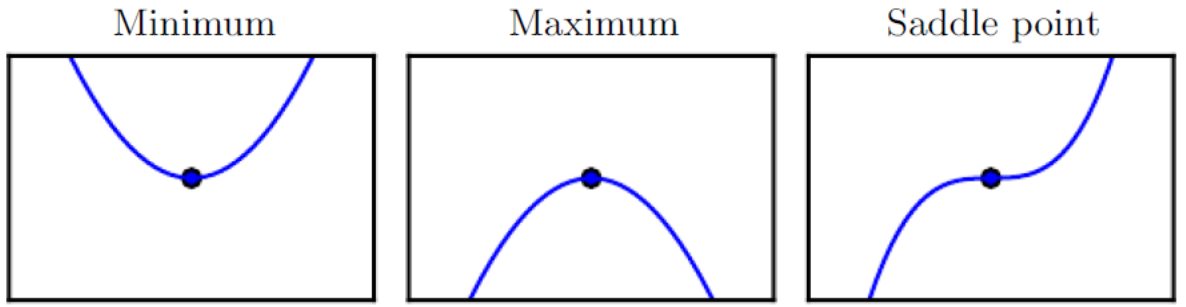
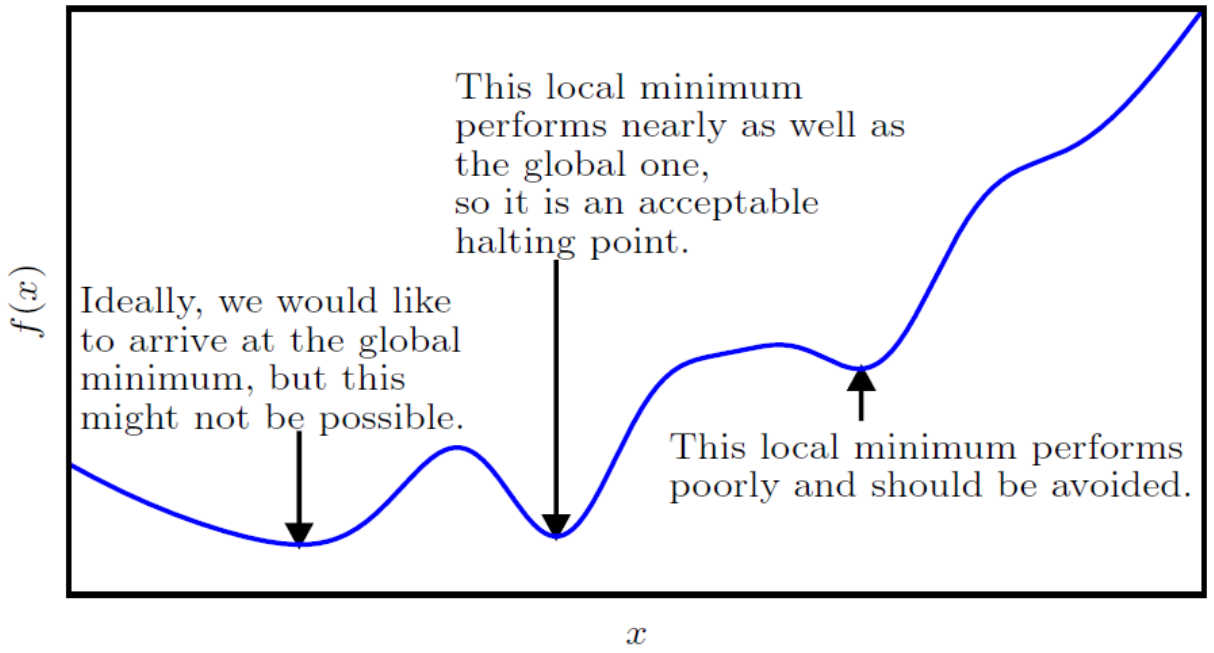


Figure 4.1

Pontos críticos ou estacionários: $f'(x) = 0$





Gradiente Descendente

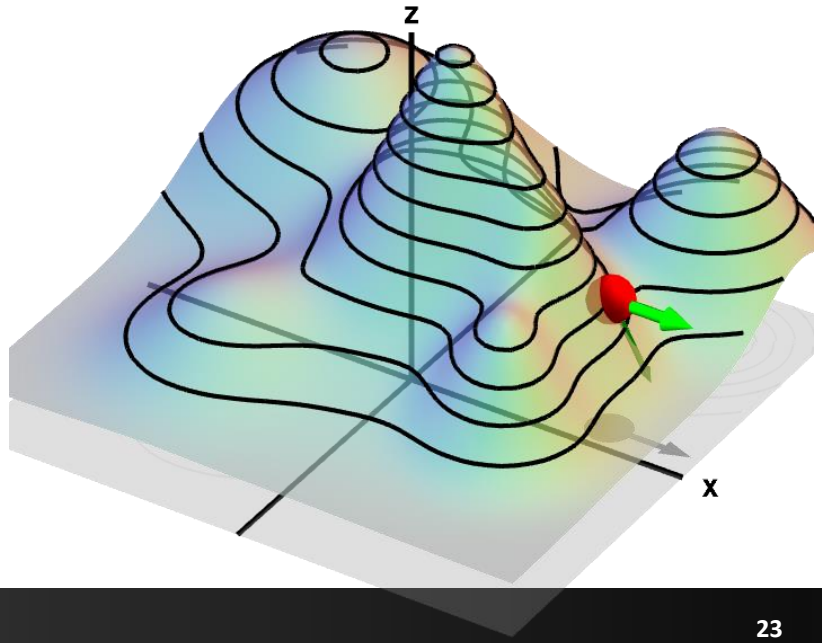
- Para funções com múltiplas entradas, devemos usar o conceito de **derivada parcial**
  - A derivada parcial $\frac{\partial}{\partial x_i} f(\boldsymbol{x})$ mede como f muda quando a variável x_i muda no ponto $\boldsymbol{x}$

- O gradiente generaliza para a noção de derivada com relação a um vetor
  - O gradiente de f é o vetor que contém todas as derivadas parciais, denotado por $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$
  - Em múltiplas dimensões, pontos críticos são pontos onde todo elemento do gradiente é igual a 0

- Pelo **método da descida do gradiente**, um novo ponto é obtido <u>iterativamente</u> a cada passo

$$\boldsymbol{x}' = \boldsymbol{x} - \in \nabla_{\boldsymbol{x}} f(\boldsymbol{x})$$

  - Em que $\in$ é a taxa de aprendizado (tamanho do passo)



GRADIENTE DESCENDENTE ESTOCÁSTICO (SGD)

- É uma extensão do algoritmo do gradiente descendente
- Um problema comum em aprendizado de máquina é que grandes conjuntos de treinamento são necessários para boa generalização
  - Isto implica em maior custo computacional para treinamento

- Tradicionalmente, as funções de custo são decompostas em uma soma sobre as amostras de treinamento de alguma função de perda aplicada para cada exemplo de treinamento
  - P. ex: $$J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x},y\sim\hat{p}_{\boldsymbol{dados}}} L(\boldsymbol{x}, y, \boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$
    $$L(\boldsymbol{x}, y, \boldsymbol{\theta}) = -\log p(y|\boldsymbol{x}; \boldsymbol{\theta})$$
  - Para estes exemplos aditivos, o gradiente descente é calculado como
    $$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$
- SGD acredita que o gradiente está em uma esperança, que pode ser estimada aproximadamente usando um pequeno conjunto de exemplo (**minibatch** de tamanho m' << m) escolhido aleatoriamente
  $$\boldsymbol{g} = \frac{1}{m'}\sum_{i=1}^{m'} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$
  $$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \in \boldsymbol{g}$$

Aprendizado baseado em Gradiente

- Projetar e treinar uma rede neural, não é muito diferente de qualquer outro modelo de aprendizado de máquina com gradiente descendente
- Basicamente, para construir um algoritmo de aprendizado de máquina precisamos definir:
  - Conjunto de dados
  - Função de custo
  - Procedimento de otimização
  - Família de Modelo

- A maior diferença entre NN e os modelos lineares é que a não linearidade das NN causa que as funções de perda mais interessantes tornam-se não convexas
  - NN são normalmente treinadas iterativamente
  - Otimizadores baseados em gradiente dirigem a função de custo para um valor muito baixo
  - Não garantem o mínimo global e são sensíveis aos valores dos parâmetros iniciais
  - Otimização convexa (modelos lineares) não dependem dos valores iniciais dos parâmetros (mas na prática podem sofrer com problemas numéricos)
  - Em redes FF é importante inicializar todos os pesos para pequenos valores aleatórios e os vieses para zero ou pequeno inteiro

Funções de Custo

- Basicamente as mesmas usadas para outros modelos paramétricos, tais como modelos lineares
- Normalmente, o modelo paramétrico define uma distribuição $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ e usamos o princípio da Máxima Verossimilhança.

$$\boldsymbol{\theta}_{MV} = \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{modelo}(\boldsymbol{x}^{(i)}|\boldsymbol{\theta})$$

  - Aplicar o logaritmo não afeta o arg max

$$\boldsymbol{\theta}_{MV} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{\boldsymbol{m}} \log p_{modelo}(\boldsymbol{x}^{(i)}|\boldsymbol{\theta})$$

  - Dividindo por 1/m

$$\boldsymbol{\theta}_{MV} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\boldsymbol{x}\sim\hat{p}_{dados}} \log p_{modelo}(\boldsymbol{x};\boldsymbol{\theta})$$

  - Podemos interpretar o MaxVer como minimizar a dissimilaridade entre a distribuição empírica (pelo conjunto de treinamento) e distribuição do modelo, medido pela divergência de Kullback-Leibler (KL)

$$D_{KL}(\hat{p}_{dados}||p_{modelo}) = \mathbb{E}_{\boldsymbol{x}\sim\hat{p}_{dados}}[\log \hat{p}_{dados}(\boldsymbol{x}) - \log p_{modelo}(\boldsymbol{x})]$$

  - Parte esquerda não depende do modelo

$$D_{KL}(\hat{p}_{dados}||p_{modelo}) = -\mathbb{E}_{\boldsymbol{x}\sim\hat{p}_{dados}}[\log p_{modelo}(\boldsymbol{x})]$$

  - Isso significa que nós utilizamos a **entropia cruzada** entre os dados de treinamento e as predições do modelo como função de custo

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{dados}} \log p_{modelo}(\boldsymbol{y}|\boldsymbol{x})$$

  - Funções de custo total envolvem ainda um termo de regularização

Funções de Custo

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{dados}} \log p_{modelo}(\boldsymbol{y}|\boldsymbol{x})$$

- Por exemplo, se usamos $p_{modelo}(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; f(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{I})$

$$J(\boldsymbol{\theta}) = \frac{1}{2}\mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{dados}}||\boldsymbol{y} - f(\boldsymbol{x};\boldsymbol{\theta})||^2 + \text{const}$$

- Uma vantagem é que remove o custo de projetar funções de custo para cada modelo
  - Especificando um modelo $p(\boldsymbol{y}|\boldsymbol{x})$ automaticamente determina uma função de custo $\log p(\boldsymbol{y}|\boldsymbol{x})$

- O gradiente da função de custo deve ser largo e previsível para servir como um bom guia para o algoritmo de aprendizado
  - Funções que saturam (tornam-se muito planos) tornam o gradiente muito pequeno
  - Muitas vezes ocorre devido às funções de ativação saturarem
  - Negativo do log da verossimilhança ajuda a evitar este problema (várias funções envolvem um exp que satura quando o argumento é muito negativo)

Unidades de Saída

- A escolha da função de custo está ligada com a escolha da unidade de saída
  - Determina a forma da função de entropia cruzada

- Qualquer tipo de unidade NN que pode ser usado como uma saída pode também ser usado como uma unidade oculta

- Vamos supor que uma rede FF fornece um conjunto de atributos ocultos $\boldsymbol{h} = f(\boldsymbol{x}; \boldsymbol{\theta})$
  - O papel da camada de saída é fornecer alguma transformação adicional dos atributos para completar a tarefa que a rede deve realizar

1. Unidades lineares para distribuições de saída gaussiana (contínuas)
   - Um tipo de saída baseado em transformação afim sem não linearidade
   - Uma camada de unidades de saída linear produz

$$\hat{\boldsymbol{y}} = \boldsymbol{W}^T \boldsymbol{h} + \boldsymbol{b}$$

   - Frequentemente são usadas para produzir a média de uma distribuição Gaussiana condicional

$$p(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I})$$

2. Unidades sigmóide para distribuições de saída Bernoulli (binárias)

$P(x=1)=\phi$
$P(x=0)=1-\phi$

– Bernoulli:

$$P(\mathrm{x}=x)=\phi^{x}(1-\phi)^{1-x}$$

– Uma camada de unidades de saída sigmóide para definir o parâmetro ϕ de Bernoulli → intervalo [0,1]

$$\hat{\boldsymbol{y}}=\sigma(\boldsymbol{\omega}^{T}\boldsymbol{h}+b)$$

$$\sigma(x)=\frac{1}{1+\exp(-x)}$$

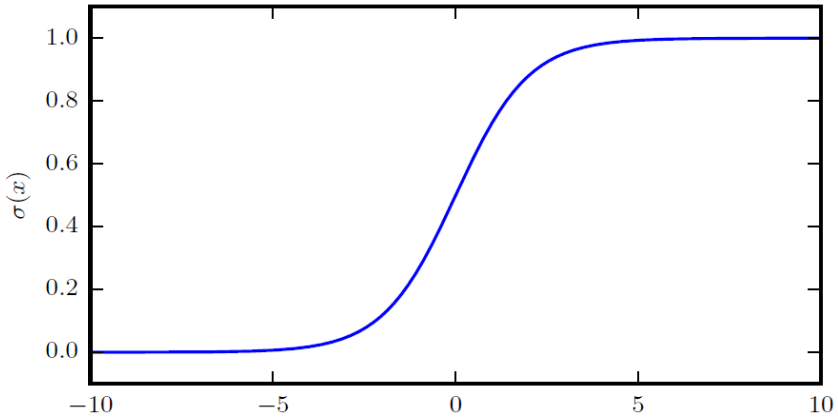
– Considerando $\mathrm{z}=\boldsymbol{\omega}^{T}\boldsymbol{h}+b$ como uma camada linear

$$\begin{aligned}J(\boldsymbol{\theta}) &= -\log P(y|\boldsymbol{x})\\ &= -\log\sigma\big((2y-1)z\big)\\ &= \varsigma\big((1-2y)z\big)\end{aligned}$$

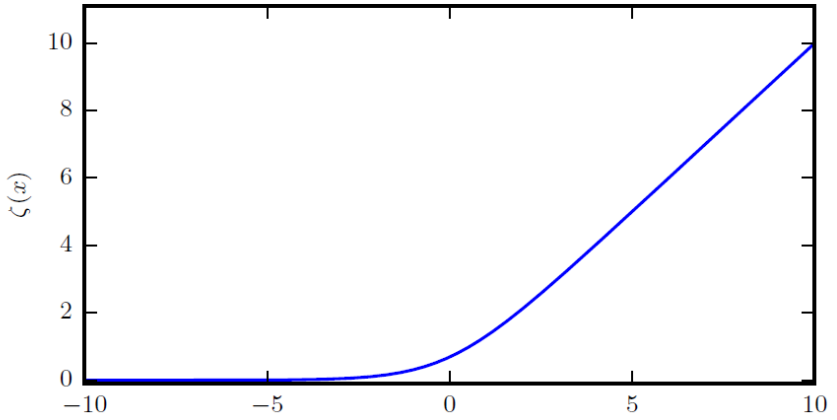
– Função softplus → intervalo $(0,\infty)$

$$\varsigma(x)=\log(1+\exp(x))$$

Logistic Sigmoid



Softplus Function



3. Unidades softmax para distribuições de saída Multinoulli (discretas em n classes)
   - Generalização da função sigmóide

$$\boldsymbol{z} = \boldsymbol{W}^T \boldsymbol{h} + \boldsymbol{b}$$

$$z_i = \log \tilde{P}(y = i|\boldsymbol{x})$$

   - Exponenciar e normalizar z para obter a saída desejada $\hat{\boldsymbol{y}}$

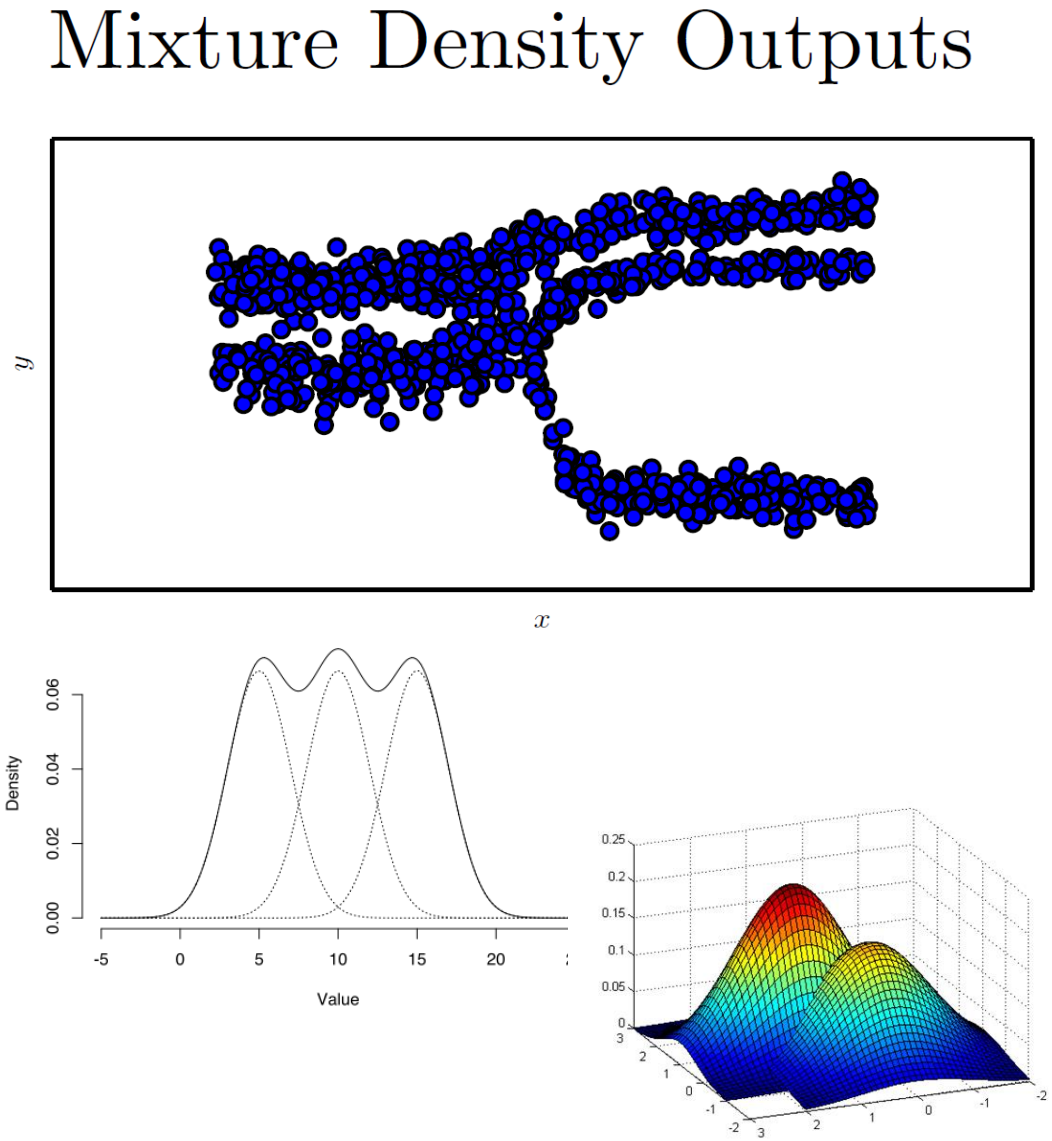
$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

   - Funciona bem com o MaxVer
   - Deve somar 1

4. Outros tipos (contínuas)
   - P. ex. regressão multimodal (pode ter vários picos em $\boldsymbol{y}$ para o mesmo valor de $\boldsymbol{x}$)
   - Saída de uma mistura de Gaussianas com n componentes

$$p(\boldsymbol{y}|\boldsymbol{x}) = \sum_{i=1}^{n} p(c = i|\boldsymbol{x}) \mathcal{N}\left(\boldsymbol{y}; \boldsymbol{\mu}^{(i)}(\boldsymbol{x}), \boldsymbol{\Sigma}^{(i)}(\boldsymbol{x})\right)$$

   - A rede neural deve gerar:
     - $p(c = i|\boldsymbol{x})$ é o peso dos componentes associados à variável latente c, forma uma distribuição multinoulli
     - $\boldsymbol{\mu}^{(i)}(\boldsymbol{x})$
     - $\boldsymbol{\Sigma}^{(i)}(\boldsymbol{x})$



Tipos de Saída

Output Type	Output Distribution	Output Layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary cross-entropy
Discrete	Multinoulli	Softmax	Discrete cross-entropy
Continuous	Gaussian	Linear	Gaussian cross-entropy (MSE)
Continuous	Mixture of Gaussian	Mixture Density	Cross-entropy
Continuous	Arbitrary	See part III: GAN, VAE, FVBN	Various

Unidades Ocultas

- Representam uma escolha que é específica das redes FF
  - Como escolher o tipo de unidade oculta para usar nas camadas ocultas do modelo
  - É uma área extremamente ativa de pesquisa

- Por padrão, ReLU é uma escolha excelente
  - Predizer qual funcionará melhor previamente é impossível
  - Tentativa e erro (intuição, treinamento e avaliação)

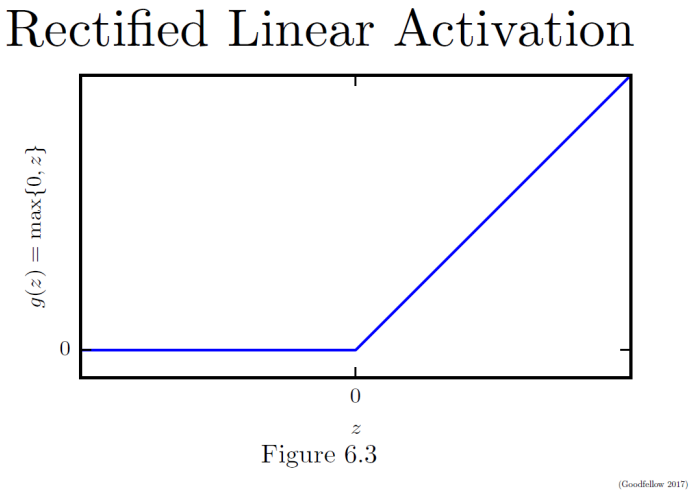


Figure 6.3
(Goodfellow 2017)

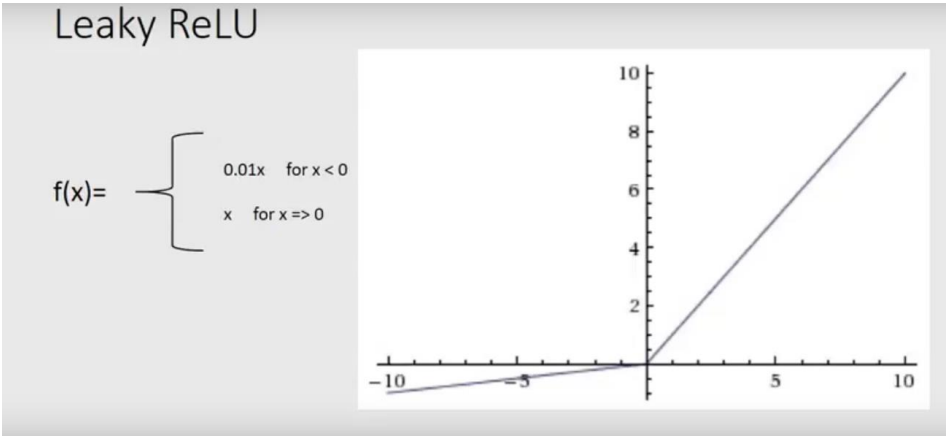
- Algumas não são diferenciáveis em todos os pontos de entrada
  - P. ex, a ReLU ($g(z) = \max\{0, z\}$) no ponto $z = 0$
  - Isto parece invalidar o uso com algoritmo de aprendizado baseado em gradiente
  - Na prática, o gradiente descendente ainda executa bem (não atinge o mínimo na função de custo, mas reduz o seu valor significativamente)
    - É improvável que o treinamento atinja um ponto onde o gradiente é **0**
    - São não diferenciáveis normalmente em somente um pequeno número de pontos
    - Funções usadas para NN normalmente tem derivadas esquerda e direita definidas
    - Métodos computacionais estão sujeitos a erros numéricos de qualquer modo ($g(0)$ é provável ser em $g(\epsilon)$, com ϵ pequeno)

- Maioria das unidades ocultas aceita entradas $\boldsymbol{x}$, calcula uma transformação afim $\boldsymbol{z} = \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b}$ e aplica uma função não linear por elemento $g(\boldsymbol{z})$
  - Se distinguem pela escolha da forma da função de ativação $g(\boldsymbol{z})$

ReLU

$$g(z) = \max\{0, z\}$$

- Similares a unidades lineares
  - Difere por metade de seu domínio resultar em zero
  - Derivada e gradientes restam grandes quando a unidade está ativa ($g'(x) = 1$)
  - $g''(x) = 0$ em quase todo lugar



$$\boldsymbol{h} = g(\boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b})$$

- Inicializar **b** para um pequeno valor positivo (ex. $0{,}1$) faria as unidades inicialmente ativas (permitindo a passagem das derivadas)

- Uma desvantagem de ReLU é não podem aprender de exemplos onde sua ativação é zero
  - Algumas generalizações garantem sempre receber um gradiente
    - Ex. $g(z) = \max\{0, z\} + \alpha \min\{0, z\}$ (α é uma pequena inclinação quanto z < 0)

- Se baseiam no princípio de que modelos são mais fáceis otimizar se o seu comportamento é próximo do linear

ReLU

- Unidades Maxout
  - Generalização da ReLU
  - Divide **z** em grupos de tamanho k (conjuntos de índices) $\mathbb{G}^{(i)} = \{(i-1)k+1, \dots, ik\}$

$$g(\boldsymbol{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j$$

  - Reduz o número de pesos para a próxima camada em k
    - Semelhante a um max pooling?

Sigmóide Logística e Tangente Hiperbólica

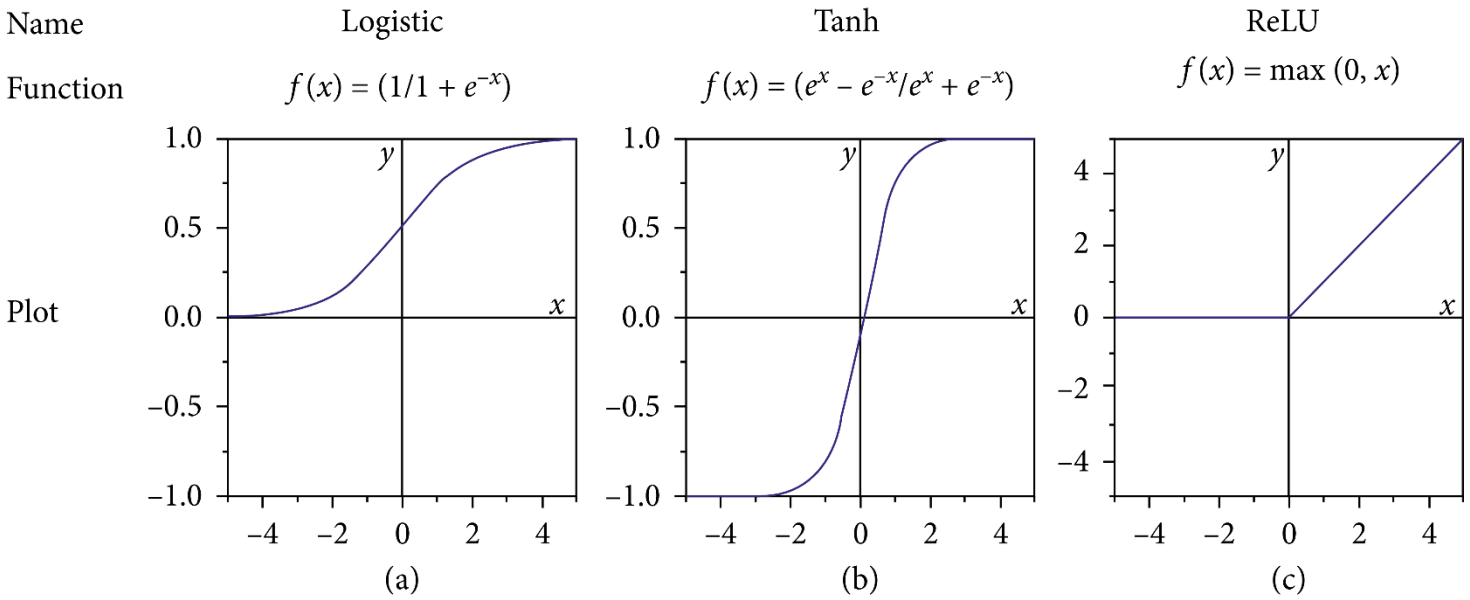
- Eram as mais usadas antes de ReLU
- Sigmóide logística

$$g(z) = \sigma(z)$$

- Tangente hiperbólica

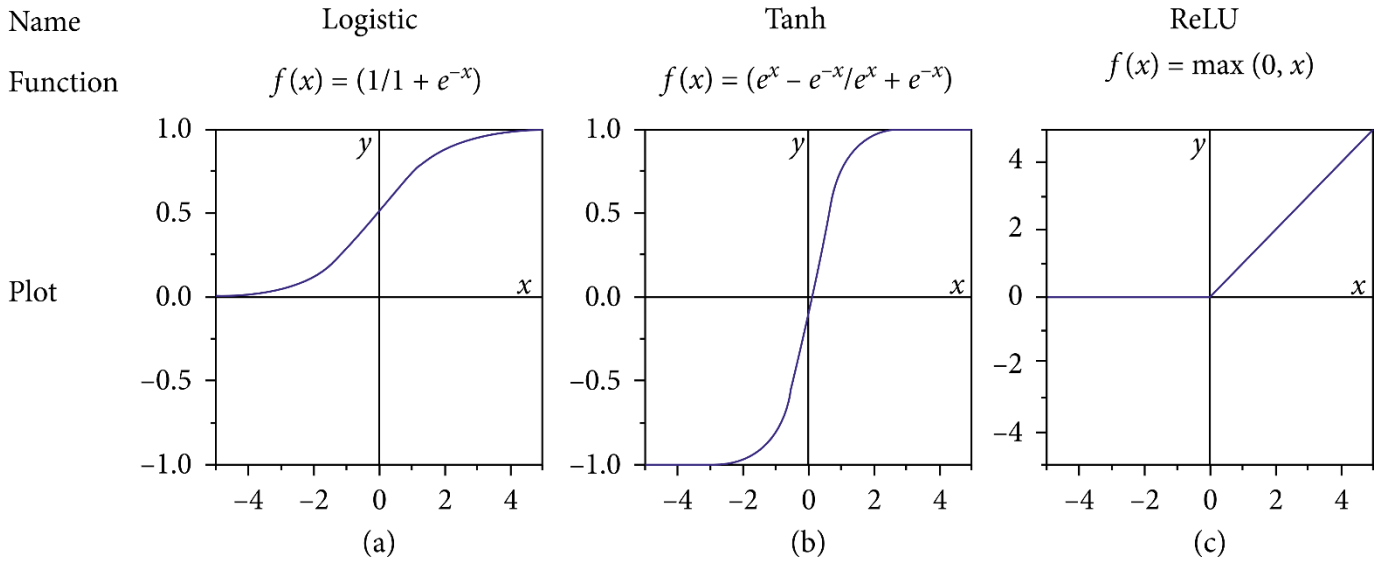
$$g(z) = tanh(z)$$

$$tanh(z) = 2\sigma(2z) - 1$$



Sigmóide Logística e Tangente Hiperbólica

- Saturam para valores altos (muito positivos ou muito negativos)
  - Muitos sensíveis à entrada quando z próximo de 0
  - Dificultam o aprendizado por gradiente
    - Atualmente, seu uso é desencorajado como camada oculta
  - Normalmente tanh funciona melhor que sigmóide (pois $\tanh(0) = 0$ e $\sigma(0) = 1/2$)
    - Treinar uma rede $\hat{y} = \boldsymbol{\omega}^T \tanh(\boldsymbol{U}^T \tanh(\boldsymbol{V}^T \boldsymbol{x}))$ se assemelha a treinar um modelo linear $\hat{y} = \boldsymbol{\omega}^T \boldsymbol{U}^T \boldsymbol{V}^T \boldsymbol{x}$ quando as ativações da rede são mantidas pequenas
  - Mais usadas em outras redes
    - Recorrentes, autoenconders...



Design de Arquitetura

- Em geral, usamos uma estrutura encadeada
  - Definir profundidade da rede e largura da rede
  - Primeira camada

$$\boldsymbol{h}^{(1)} = g^{(1)}\left(\boldsymbol{W}^{(1)T}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)$$

  - Segunda camada

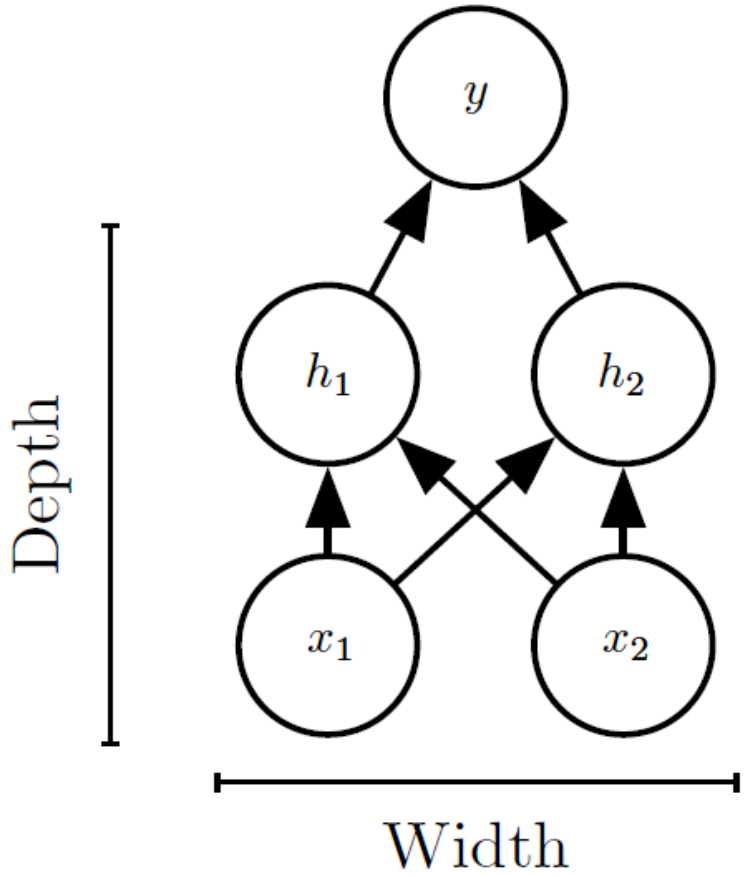
$$\boldsymbol{h}^{(2)} = g^{(2)}\left(\boldsymbol{W}^{(2)T}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right)$$

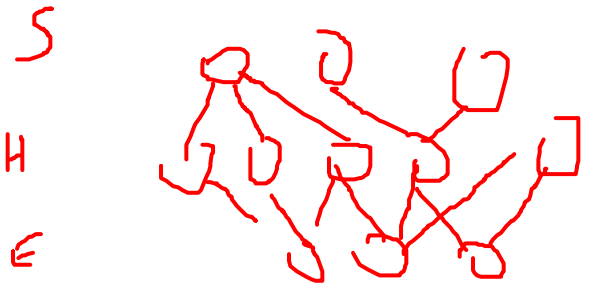
  - ...

- Como as unidades devem ser conectadas a cada outra?

- Podemos ter arquiteturas mais especializadas para tarefas específicas
  - CNN para visão computacional
  - Recorrentes para processamento de sequências

- Arquiteturas podem
  - adicionar atalhos nas conexões (skip connections)
  - Entradas estarem conectadas a um pequeno subconjunto de unidades na camada de saída
    - Reduzindo conexões, reduz parâmetros
  - Entre outras decisões de projeto (“não há receita de bolo”)





Propriedades de Aproximação Universal e Profundidade

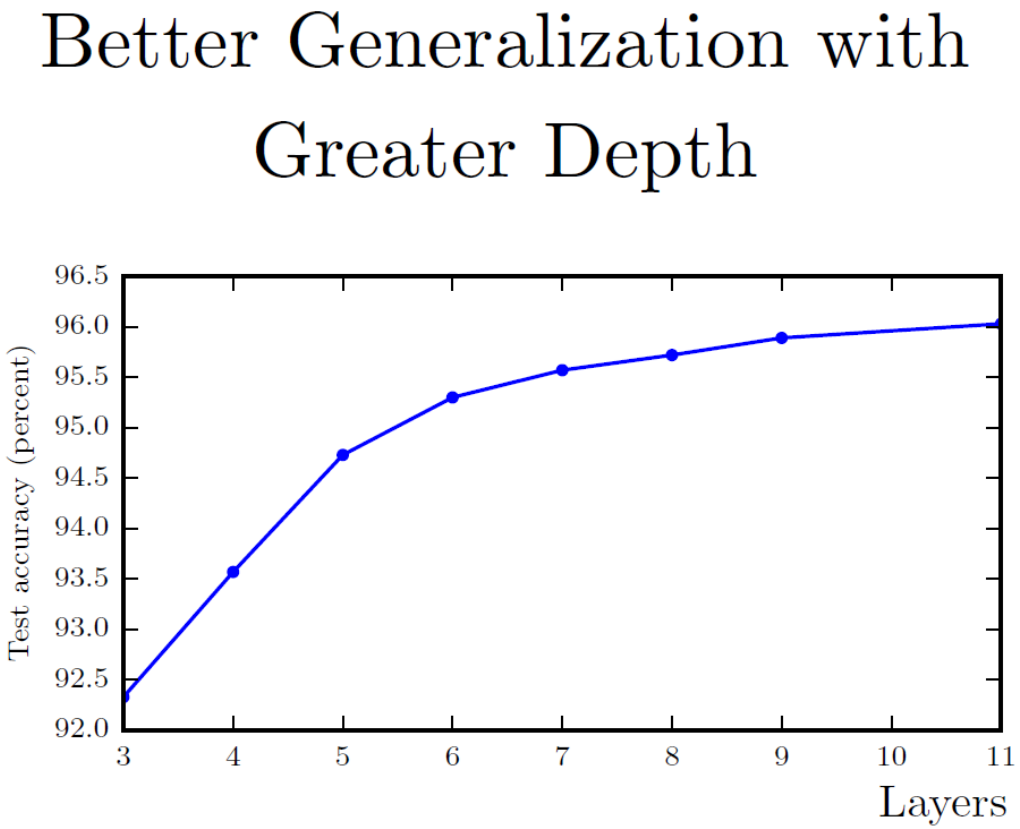
- O **teorema de aproximação universal** estabelece que, independente de qual função (contínua) nós estamos tentando apresentar, uma MLP grande seria capaz de representá-la
  - Mas não há garantia que o algoritmo de treinamento conseguirá aprender a função
    - Otimizador pode não conseguir encontrar parâmetros que representam a função desejada
    - Overfitting pode levar a uma escolha errada de parâmetros
  - Não diz o quão grande deve ser

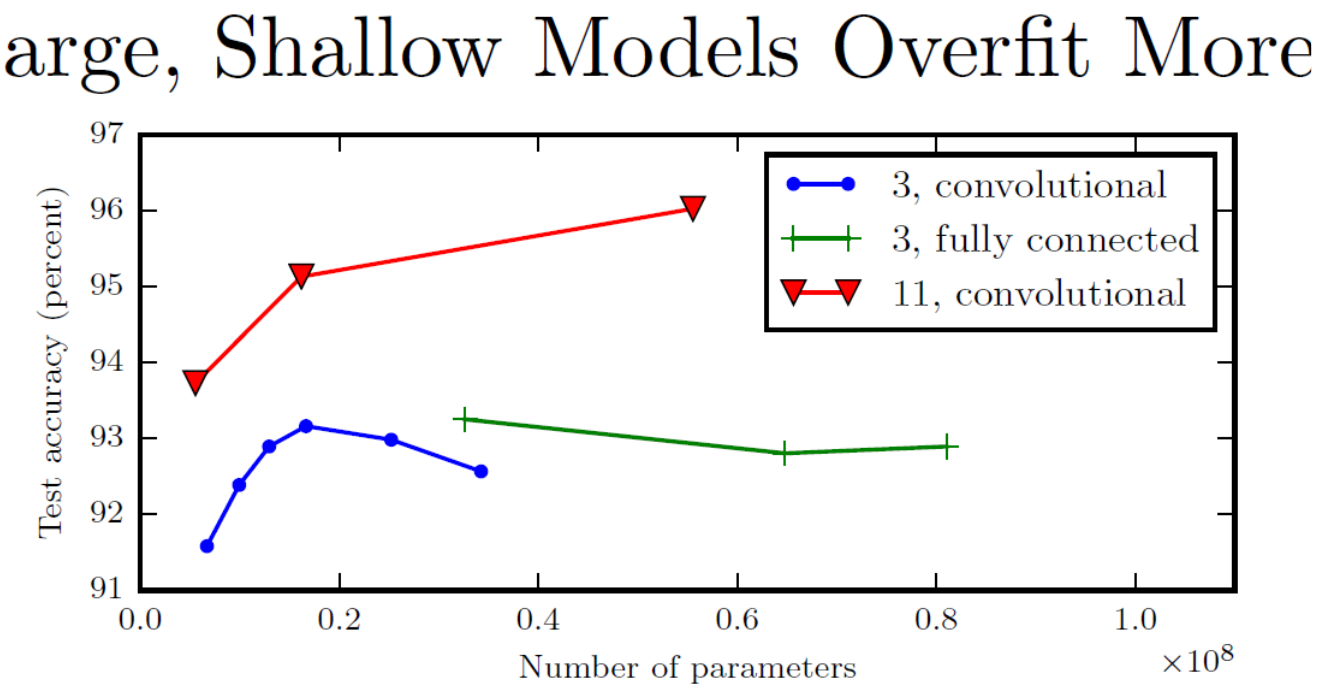
- Em resumo, uma rede FF com uma única camada é suficiente para representar qualquer função
  - Mas pode ser impraticável treiná-la adequadamente
  - O número de unidades poderia ser exponencial

- Usar modelos mais profundos pode reduzir o número de unidades exigidas para representar a função desejada e pode reduzir a quantidade de erro de generalização

Propriedades de Aproximação Universal e Profundade

- Resultados empíricos

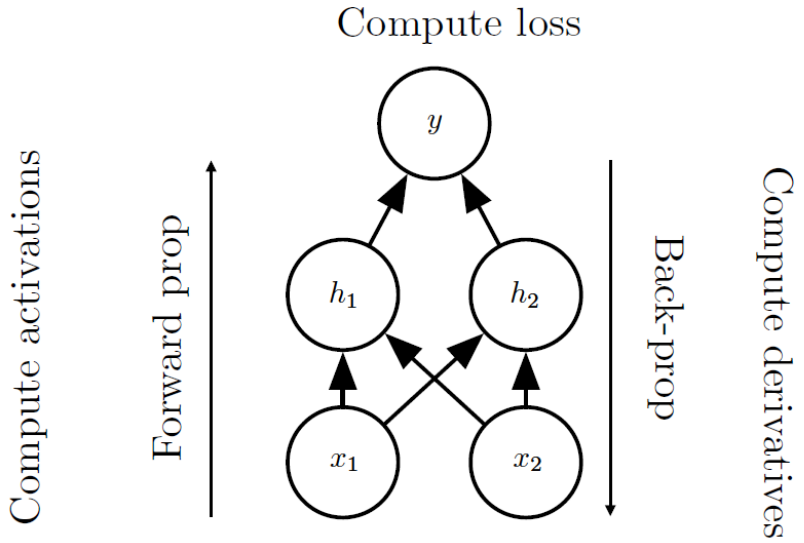




BACKPROPAGATION

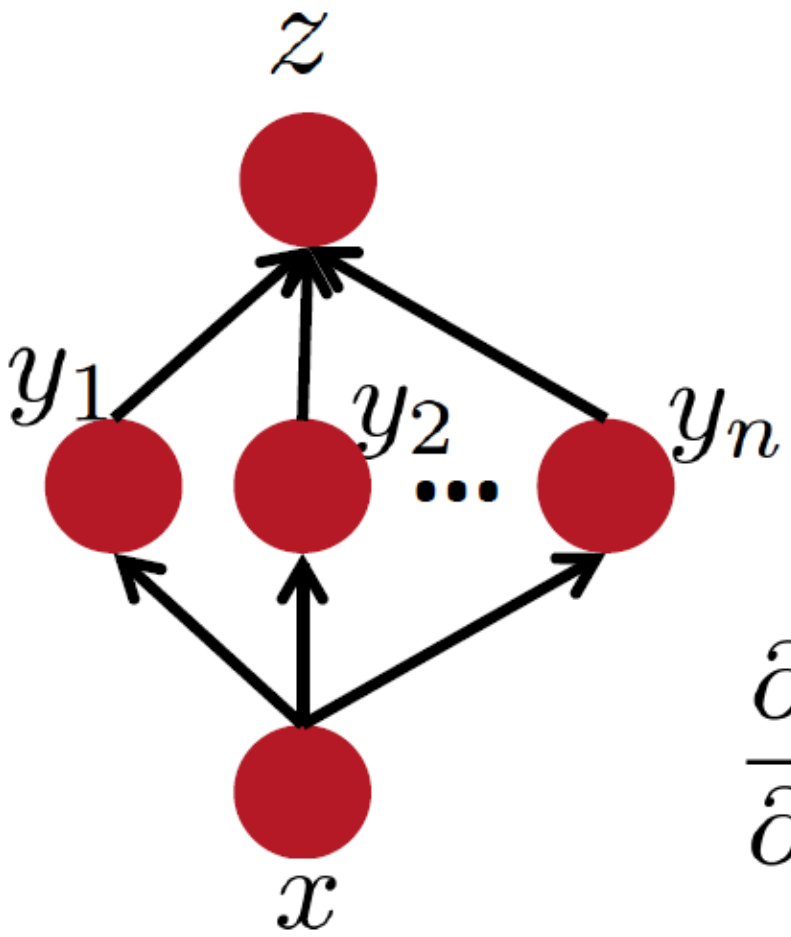
- O processamento de uma entrada $\boldsymbol{x}$ na rede produz a saída $\hat{\boldsymbol{y}}$ (propagação para frente)
  - Nesta propagação, durante o treinamento podemos calcular um custo escalar $J(\boldsymbol{\theta})$
  - O algoritmo backpropagation permite a informação do custo fluir no sentido oposto da rede a fim de calcular os gradientes da função de custo $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ (em algoritmos de aprendizado)

- Cálculo analítico de gradiente é direto, mas avaliar numericamente é computacionalmente caro
  - O algoritmo de backpropagation simplifica e barateia este procedimento



- **Não é o algoritmo de aprendizado!**
  - Gradiente descendente é um algoritmo de aprendizado que usa os gradientes calculados pelo backpropagation
  - Não é específico de redes MLP

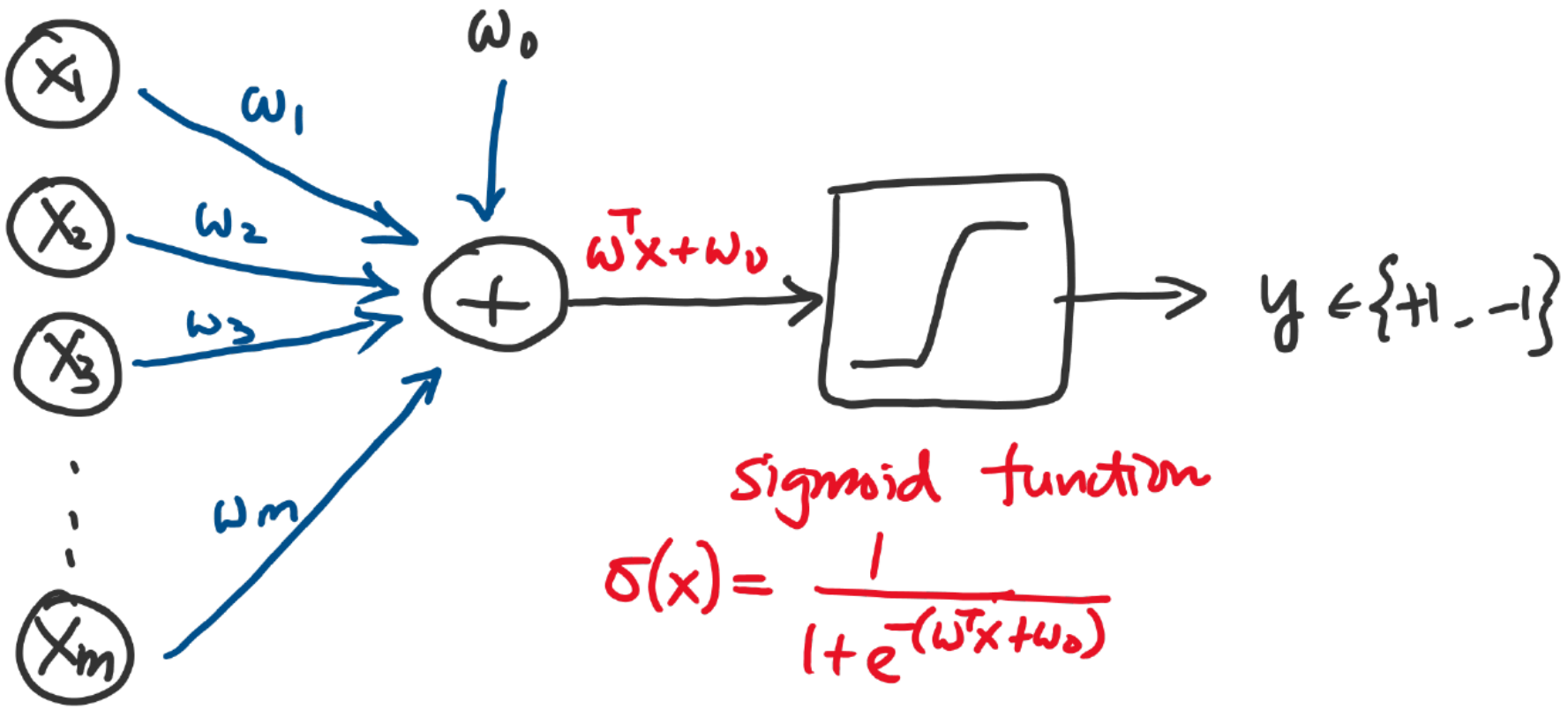
Regra da Cadeia de Cálculo



$z = f(\sum_i y_i)$
$y_i = g(x)$
$\Rightarrow z = f(\sum_i g(x_i))$

$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

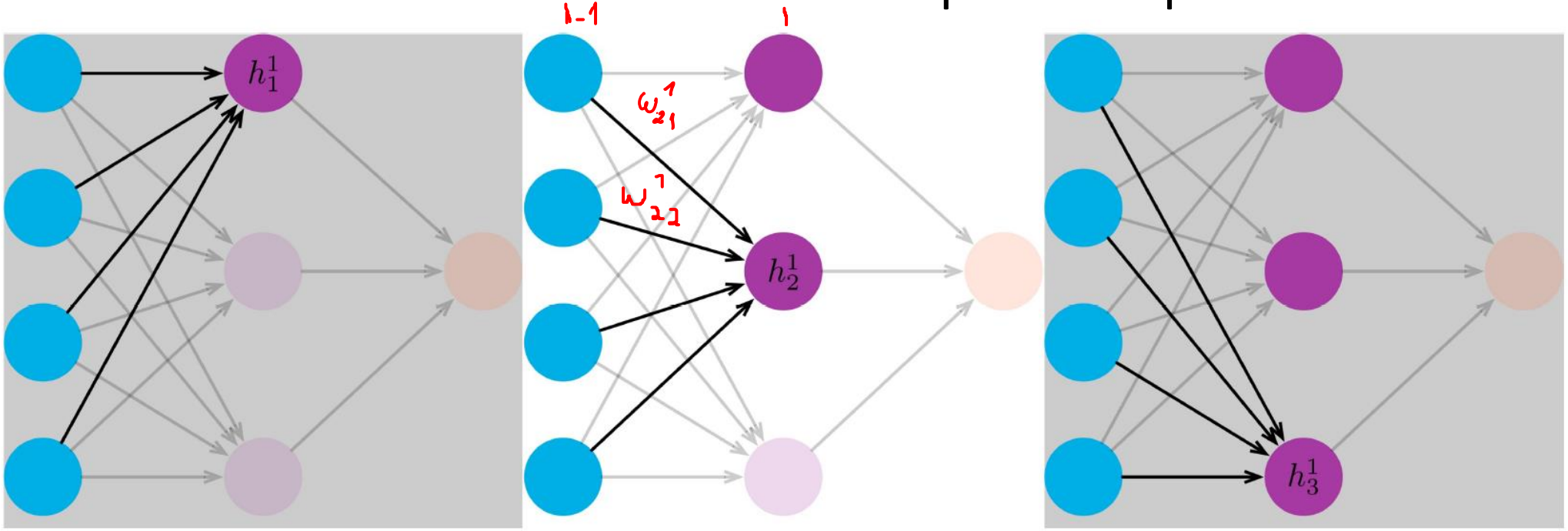
Perceptron Simples



$$y = \sigma(\boldsymbol{w}^T \boldsymbol{x} + w_0)$$

MLP

- Cada neurônio oculto é a saída de um Perceptron simples



$$\begin{bmatrix} h_1^1 \\ h_2^1 \\ \vdots \\ h_n^1 \end{bmatrix} = \begin{bmatrix} w_{11}^1 & w_{21}^1 & \dots & w_{n1}^1 \\ w_{12}^1 & w_{22}^1 & \dots & w_{n2}^1 \\ \vdots & \vdots & \ddots & \vdots \\ w_{1m}^1 & w_{2m}^1 & \dots & w_{nm}^1 \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

w_{jk}^l é o peso do k-ésimo neurônio da camada $l-1$ para o j-ésimo neurônio na camada l

MLP

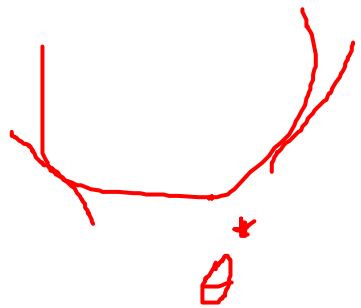
- Função de Custo

$$J(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_L) = \sum_{i=1}^{N} \|\sigma(\boldsymbol{W}_L^T \ldots \sigma(\boldsymbol{W}_2^T \sigma(\boldsymbol{W}_1^T \boldsymbol{x}_i))) - \boldsymbol{y}_i\|^2$$

y_i

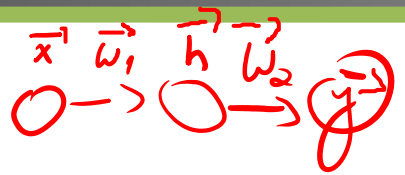
- Gradiente descendente

$$\boldsymbol{W}_1^{t+1} = \boldsymbol{W}_1^t - \alpha \nabla J(\boldsymbol{W}_1^t)$$



- Precisamos descobrir todos os parâmetros

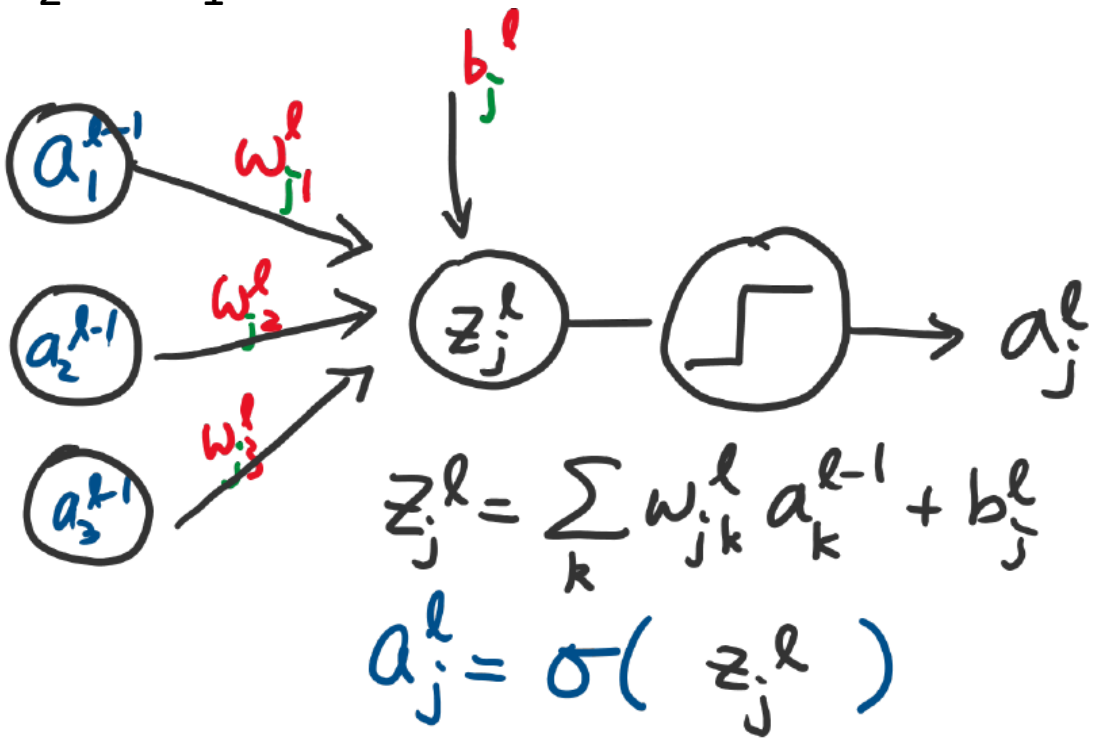
Exemplo para MLP



- Função de Custo

$$J(\boldsymbol{W}_1, \boldsymbol{W}_2) = \|\underbrace{\sigma(\boldsymbol{W}_2^T \sigma(\boldsymbol{W}_1^T \boldsymbol{x}))}_{\boldsymbol{a}_2} - \boldsymbol{y}\|^2$$

- <u>Vamos</u> obter o gradiente com relação à W_2 e W_1



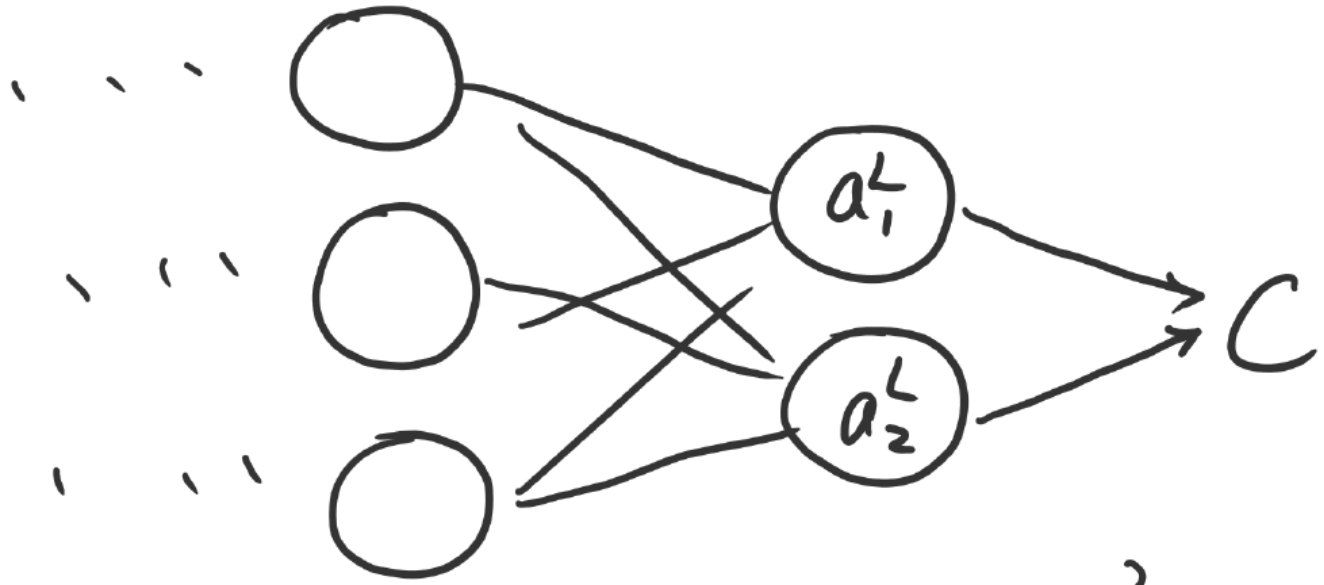
Derivando Equações Principais do Backpropagation

- Função de Custo

$$C = \sum_j (a_j^L - y_j)^2$$

$$\frac{\partial C}{\partial a_j^L} = \sum \frac{\partial (a_j^L - y_j)^2}{\partial a_j^L}$$

$$= 2(a_j^L - y_j)$$



$$C = (a_1^L - y_1)^2 + (a_2^L - y_2)^2$$

Derivando Equações Principais do Backpropagation

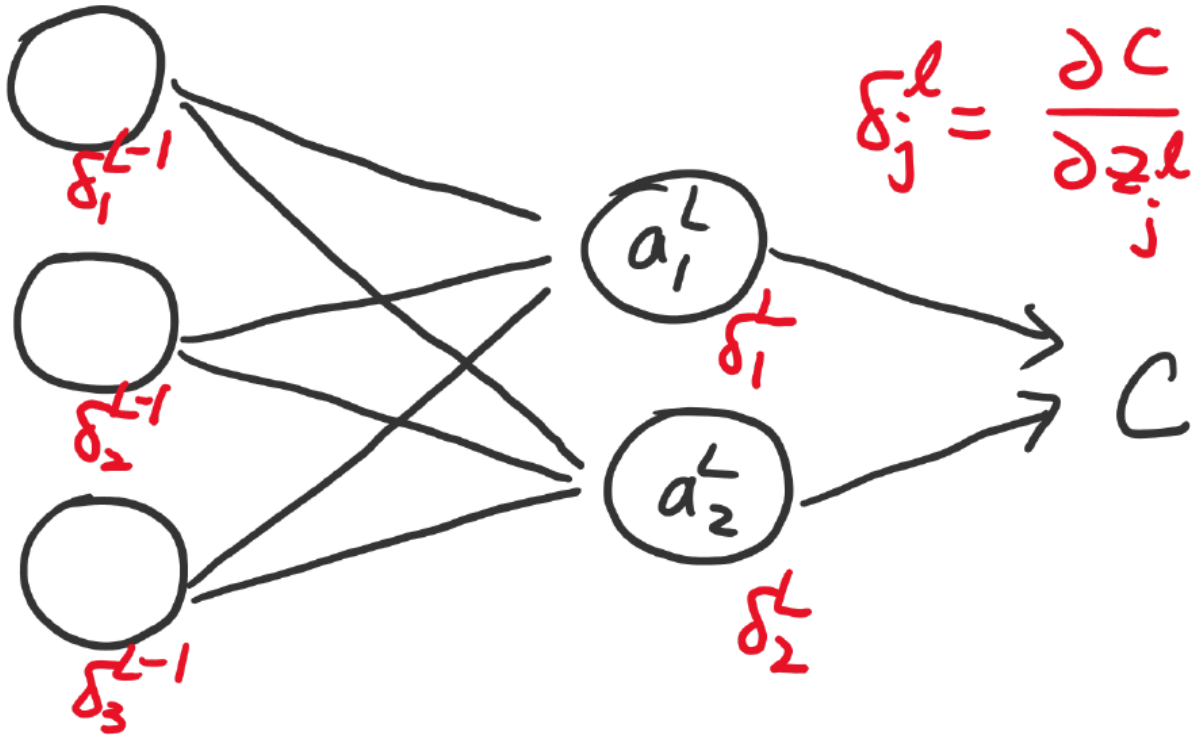
- Termo do Erro

Erro na camada de saída L

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L).$$

$$2(a_j^L - y_j)$$

$$2(a_j^L - y_j)$$



$$\vec{\delta}^L = \begin{bmatrix} \delta_1^L \\ \delta_2^L \end{bmatrix} = \begin{bmatrix} \frac{\partial C}{\partial a_1^L} \\ \frac{\partial C}{\partial a_2^L} \end{bmatrix} \odot \begin{bmatrix} \sigma'(z_1^L) \\ \sigma'(z_2^L) \end{bmatrix}$$

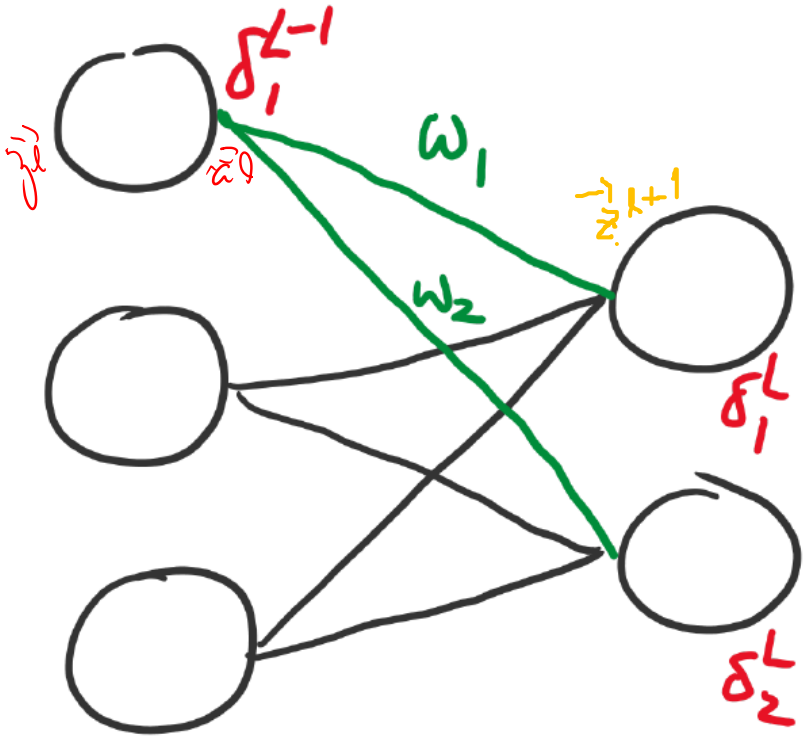
$$= \begin{bmatrix} \frac{\partial C}{\partial a_1^L} \cdot \sigma'(z_1^L) \\ \frac{\partial C}{\partial a_2^L} \cdot \sigma'(z_2^L) \end{bmatrix}$$

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

Derivando Equações Principais do Backpropagation

- Erro de uma camada l em termos da camada seguinte $(l+1)$

$$\delta^{\ell} = ((w^{\ell+1})^T \delta^{\ell+1}) \odot \sigma'(z^{\ell})$$



$$\delta_j^{\ell} = \frac{\partial C}{\partial z_j^{\ell}}$$

$$\vec{a}^{\ell} = \sigma(\vec{z}^{\ell})$$

$$\vec{z}^{\ell+1} = (w^{(\ell+1)})^T \cdot \vec{a}^{\ell}$$

$$= \frac{\partial C}{\partial \vec{z}^{\ell+1}} \cdot \frac{\partial \vec{z}^{\ell+1}}{\partial \vec{z}^{\ell}}$$

$$\vec{\delta}^{\ell+1} \qquad \frac{\partial \vec{z}^{\ell+1}}{\partial \vec{a}^{\ell}} \quad \frac{\partial \vec{a}^{\ell}}{\partial \vec{z}^{\ell}}$$

$$w^{(\ell+1)}$$

Derivando Equações Principais do Backpropagation

- Equação para o viés

$$\frac{\partial C}{\partial b_j^\ell} = \delta_j^\ell$$

$$z_j^\ell = \sum_k w_{jk}^\ell a_k^{\ell-1} + b_j^\ell$$

- Equação para os pesos (mostrar)

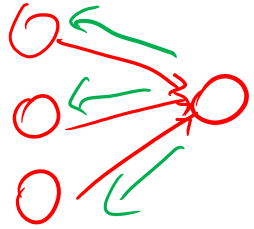
$$\frac{\partial C}{\partial w_{jk}^\ell} = a_k^{\ell-1} \delta_j^\ell$$

$$\frac{\partial C}{\partial w_{jk}^\ell} = \frac{\partial C}{\partial z_j^\ell} \cdot \frac{\partial z_j^\ell}{\partial w_{jk}^\ell}$$

δ_j^ℓ $\cdot$ $a_k^{\ell-1}$

Algoritmo BP

1. **Input x:** Set the corresponding activation a^1 for the input layer.



2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.

3. **Output error δ^L:** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.

4. **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2,1$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.

5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

Usar f ao invés de sigmóide para uma MLP com qualquer função de ativação

OBS: $a^0_k = x_k$

Lembre-se que uma rede FF pode ser vista como um dígrafo, de tal forma que podemos seguir as relações de paternidade para definir qual nó afeta outro nó.