Hands-on Deep Learning - Aula 2

Introdução às Redes Neurais

Camila Laranjeira[1,2], **Hugo Oliveira**[1,2], Pedro H. Barros[1,3]

[1]Programa de Pós-Graduação em Ciência da Computação (PPGCC)
Universidade Federal de Minas Gerais

[2]Interest Group in Pattern Recognition and Earth Observation (PATREO)
Universidade Federal de Minas Gerais

[3]Wireless Informational Sensing Embedded systems Modeling Algorithms and Protocols (WISEMAP)
Universidade Federal de Minas Gerais

08 de Fevereiro, 2020

DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

UFMG

Agenda

Agenda

O Cérebro

- Redes Neurais foram originalmente inspiradas no cérebro humano
- Sua unidade básica é o neurônio, composto por 3 partes principais:
  - Dendritos: Recebem sinais de outros neurônios conectados a ele
  - Corpo: Responsável por “processar” a informação
  - Axônio: Transmite o sinal processado para outros neurônios. A ativação do axônio depende da força sináptica do sinal.

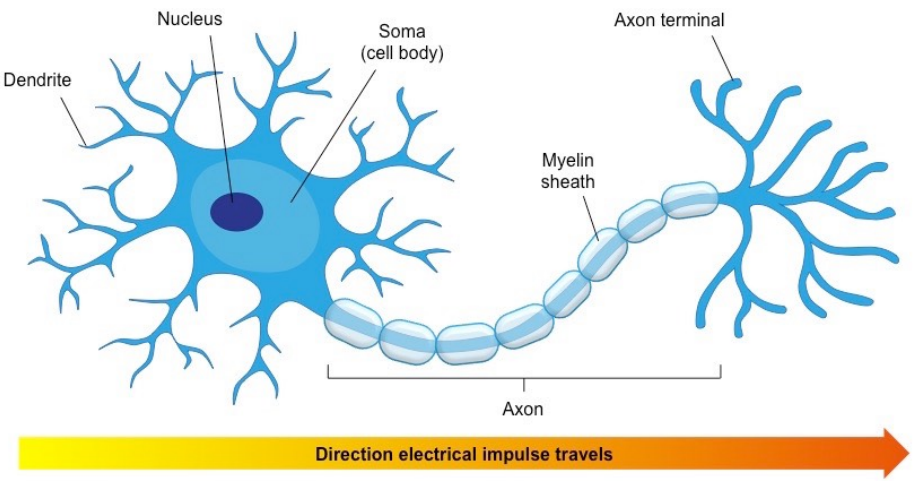


Figura: Representação de um neurônio [1]

O Cérebro

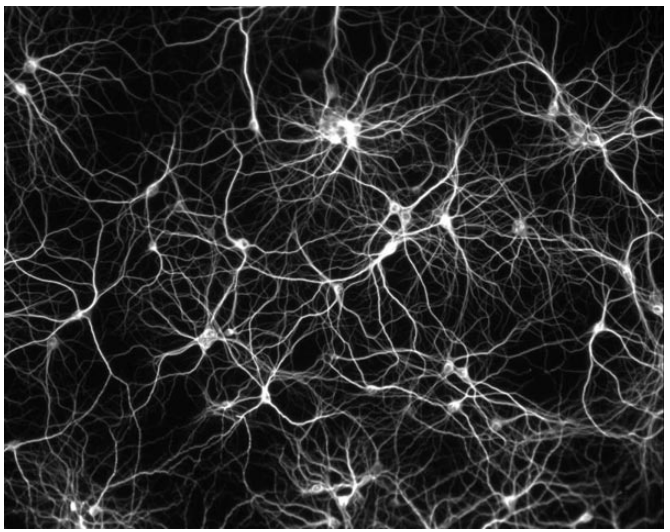


Figura: Representação da malha neural [2]

- Os neurônios se conectam uns aos outros, criando uma grande malha.
- Responsável pelo processamento de todas as funções do corpo.

Representação Artificial do Cérebro

- Redes neurais artificiais são sistemas compostos de neurônios artificiais **inspirados** no neurônio biológico.

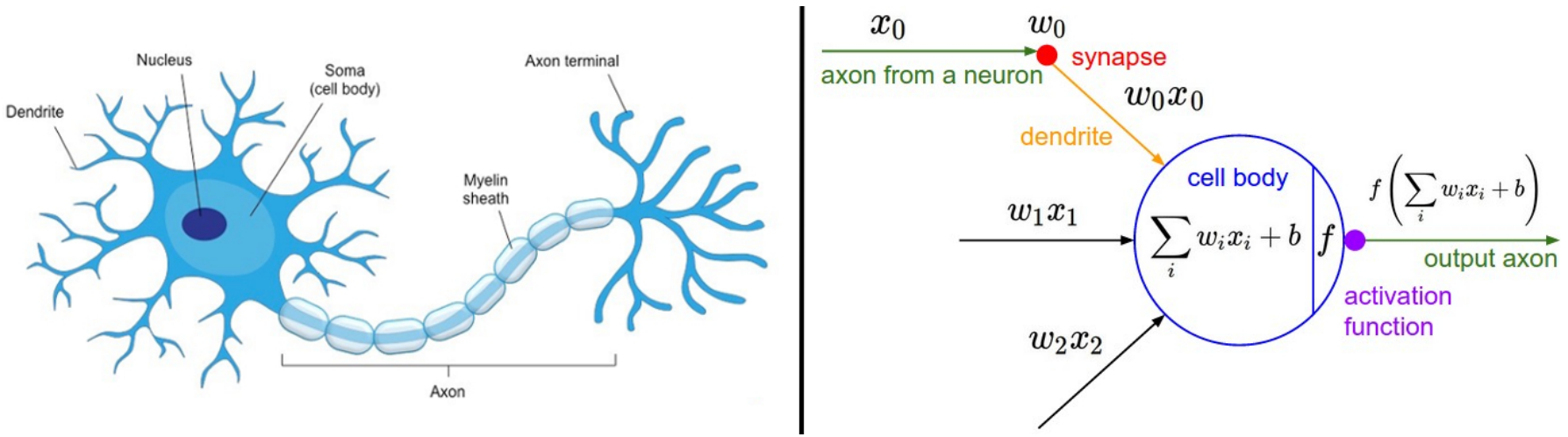
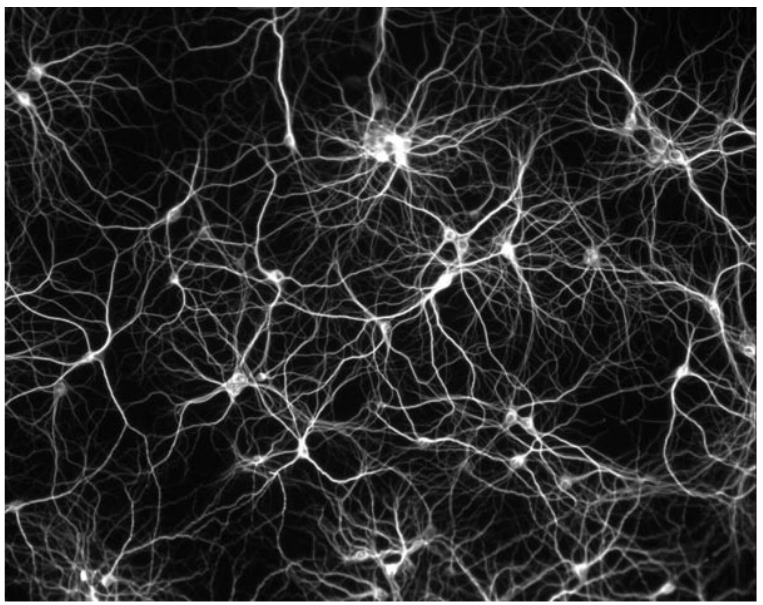
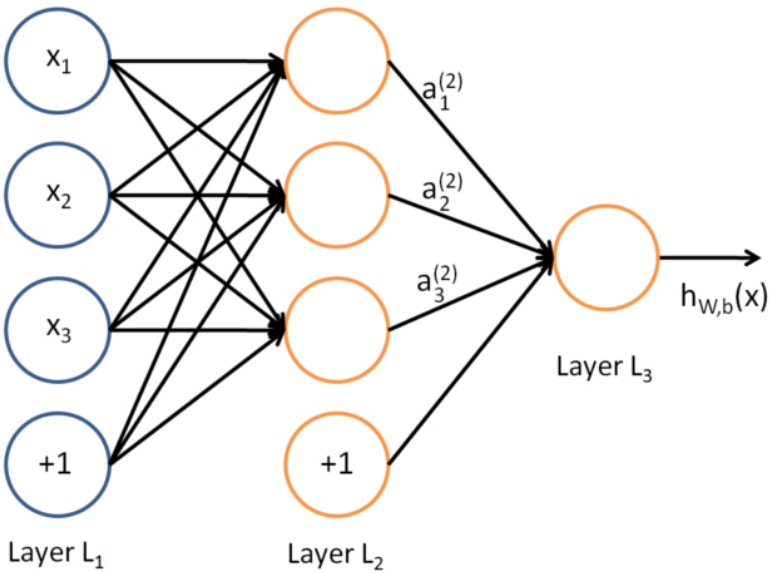


Figura: Comparando os neurônios biológico e artificial (Perceptron).

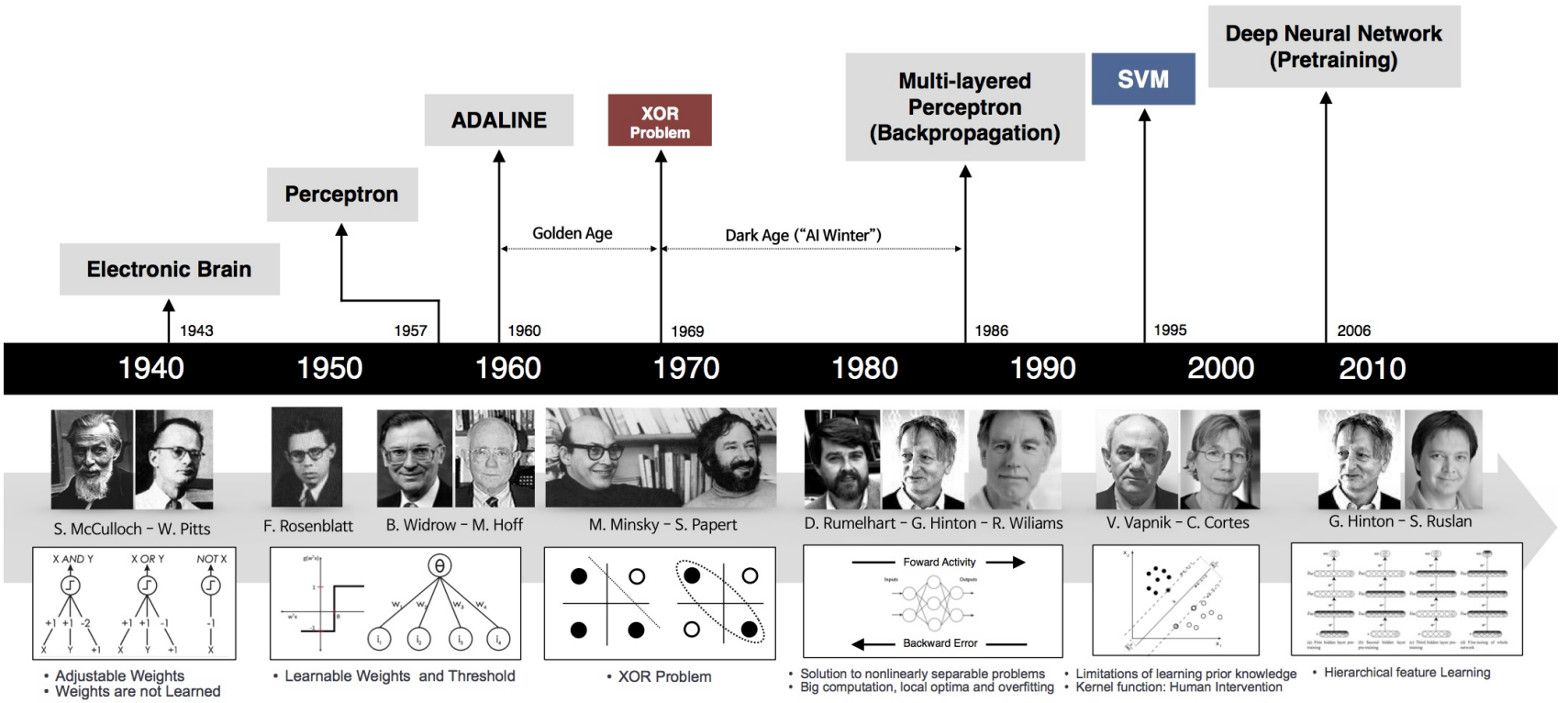
Representação Artificial do Cérebro

- Redes neurais artificiais são sistemas compostos de neurônios artificiais **inspirados** no neurônio biológico.
  - Similarmente combinados na forma de uma malha neural.
  - Esse modelo é **muito limitado** se comparado com o cérebro real.





Representação Artificial do Cérebro



Agenda

Classificação Linear

- **Mapeamento**: Uma regra que, para cada elemento do conjunto X, provê o elemento correspondente do conjunto Y.

  $f : X \rightarrow Y$

- Um mapeamento nada mais é do que uma **função**.

  $y = f(x)$

Classificação Linear

- **Mapeamento**: Uma regra que, para cada elemento do conjunto X, provê o elemento correspondente do conjunto Y.

  $f : X \rightarrow Y$

- Um mapeamento nada mais é do que uma **função**.

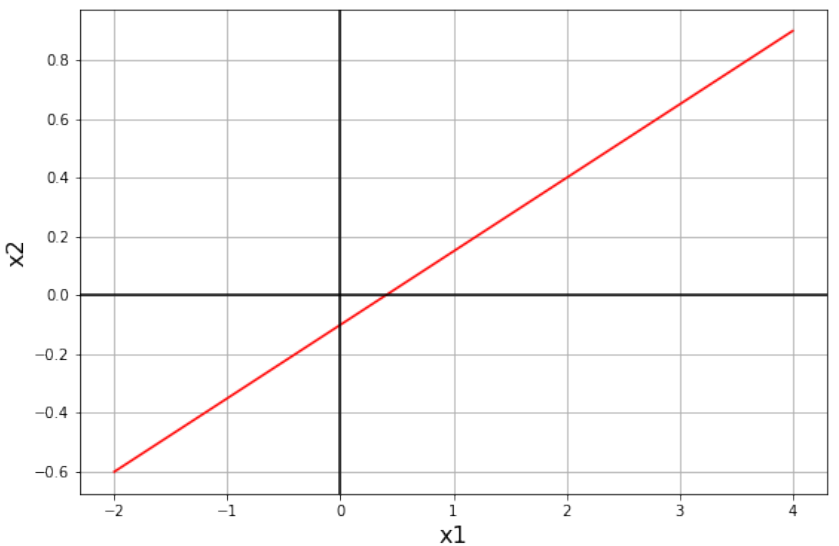
  $y = f(x)$

Classificação Linear

- Equação da Reta

  $w_1 x_1 + w_2 x_2 + b = 0$

- Um peso w_i por dimensão
- Um viés b



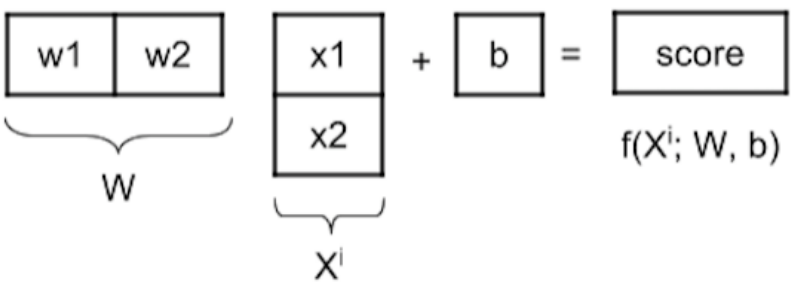
Classificação Linear

- Equação da Reta: Pode servir como uma função de mapeamento.

  $f(x; W, b) = w_1 x_1 + w_2 x_2 + b$

- Em outras palavras: Dados os **parâmetros** $W = \{w_1, w_2\}$ e b de uma reta, é possível mapear uma entrada $X = \{x_1, x_2\}$ para a saída $f(x; W, b)$.

- Representação matricial



Classificação Linear

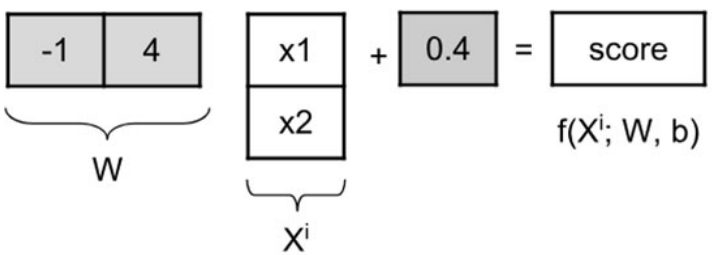
- Equação da Reta:

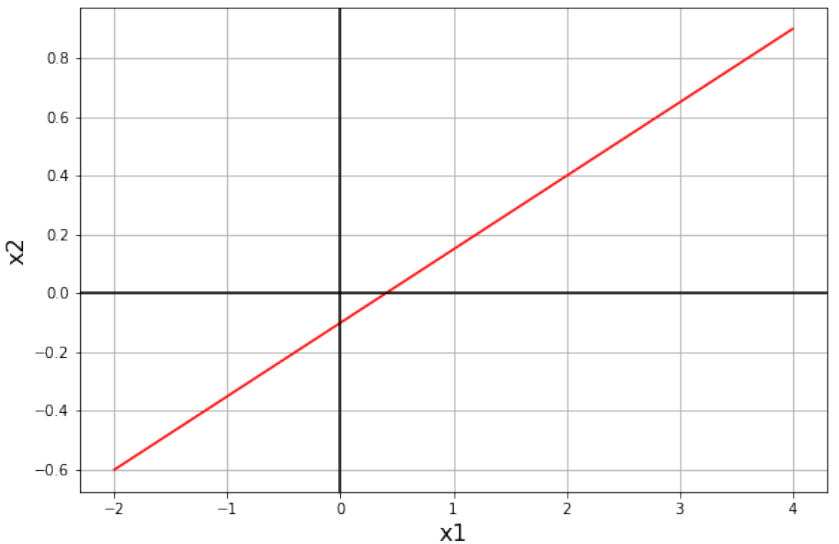
$$f(x; W, b) = w_1 x_1 + w_2 x_2 + b$$

- Vamos supor uma reta fixa:

$$f(x; W, b) = -1x_1 + 4x_2 + 0.4$$

- Representação matricial

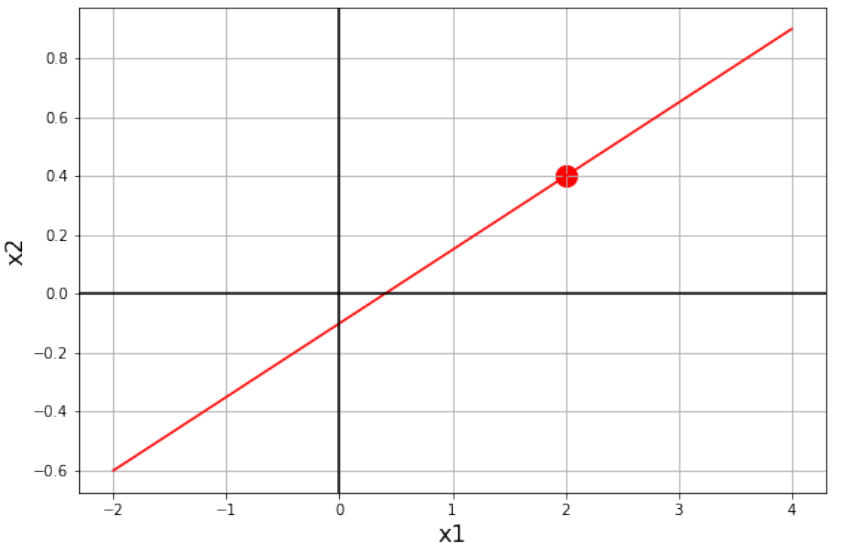




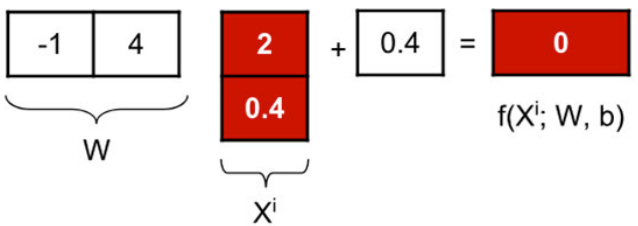
Classificação Linear

- Vamos supor uma reta fixa:

$$f(x; W, b) = -1x_1 + 4x_2 + 0.4$$



- Para $X = \{2.0, 0.4\}$

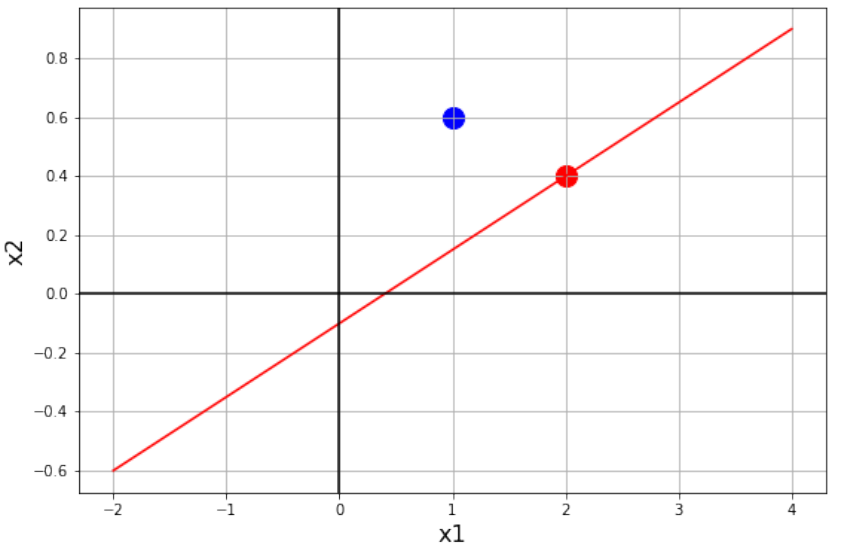


- $f(x; W, b) = 0$: ponto sobre a reta.

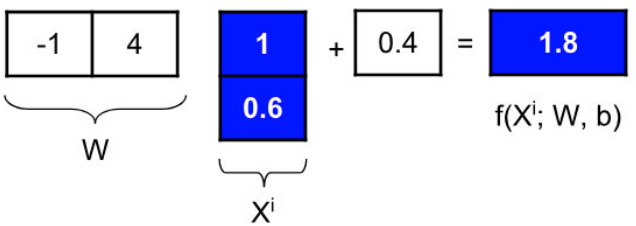
Classificação Linear

- Vamos supor uma reta fixa:

$$f(x; W, b) = -1x_1 + 4x_2 + 0.4$$



- Para $X = \{1.0, 0.6\}$

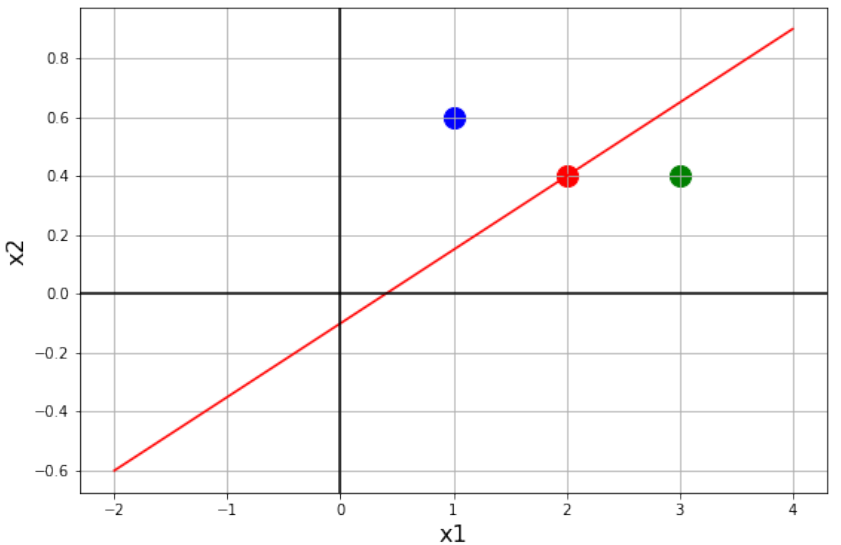


- $f(x; W, b) > 0$: ponto a cima da reta.

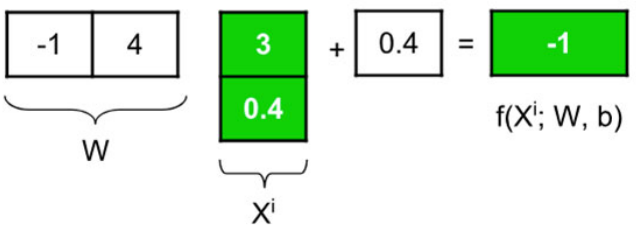
Classificação Linear

- Vamos supor uma reta fixa:

$$f(x; W, b) = -1x_1 + 4x_2 + 0.4$$



- Para $X = \{3.0, 0.4\}$



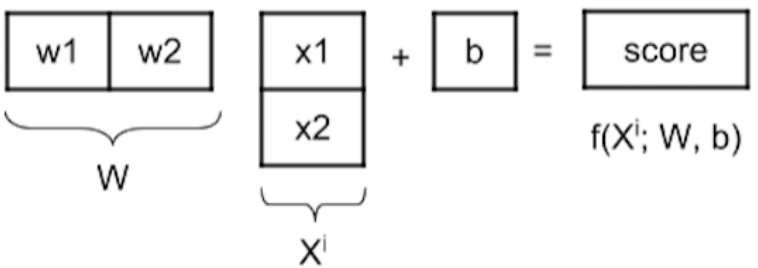
- $f(x; W, b) < 0$: ponto abaixo da reta.

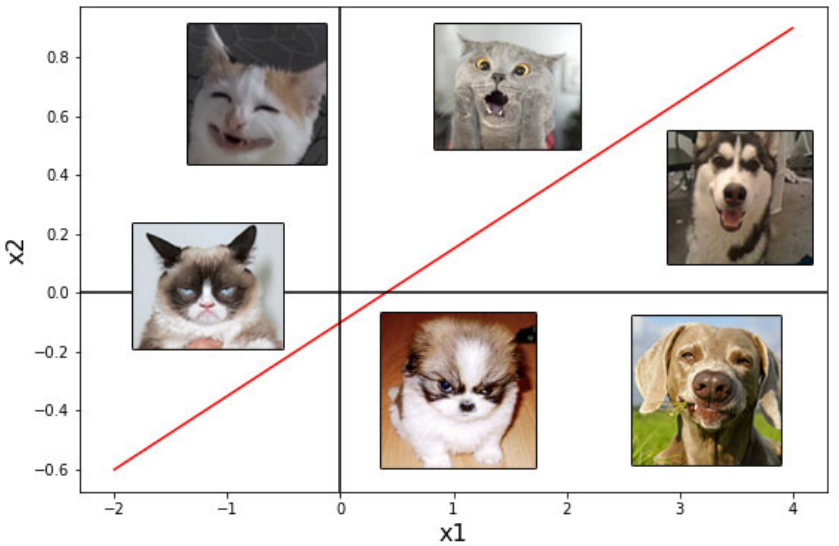
Classificação Linear

- Equação da Reta

$$f(x; W, b) = w_1 x_1 + w_2 x_2 + b$$

- Representação matricial

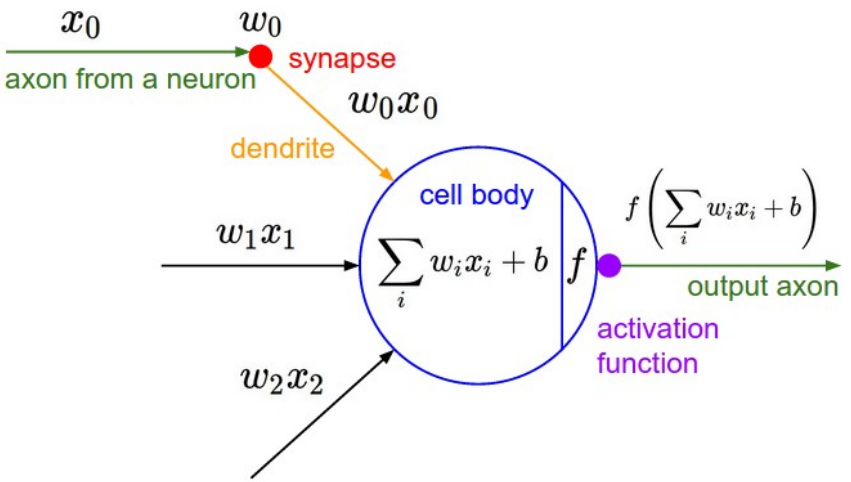




- Em termos gerais, o treinamento de um classificador linear busca os **parâmetros** *W* e *b* para compor a **fronteira de decisão** entre as classes.

Perceptron

- Unidade elementar das redes neurais.
- Modelo matemático que mapeia suas entradas para uma saída.
- Exemplo: um ponto em 3 dimensões $X = \{x_0, x_1, x_2\}$ vai compor um perceptron com 3 entradas e 1 saída.

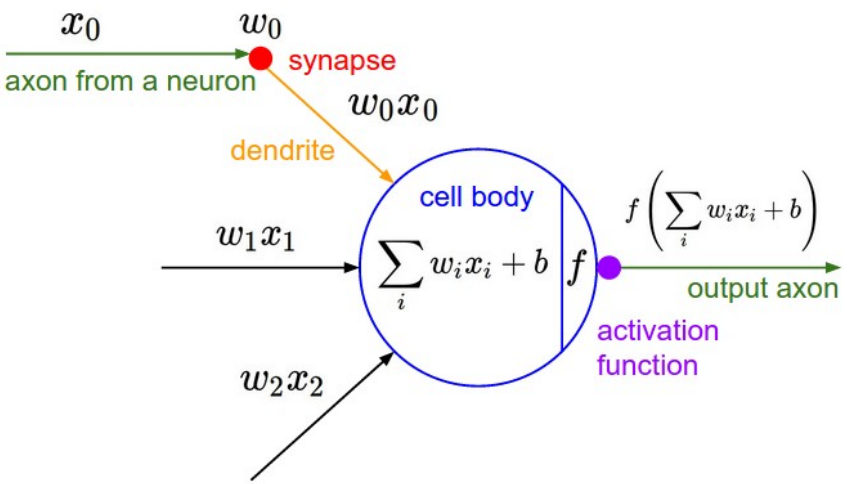


Perceptron

- Mapeamento linear

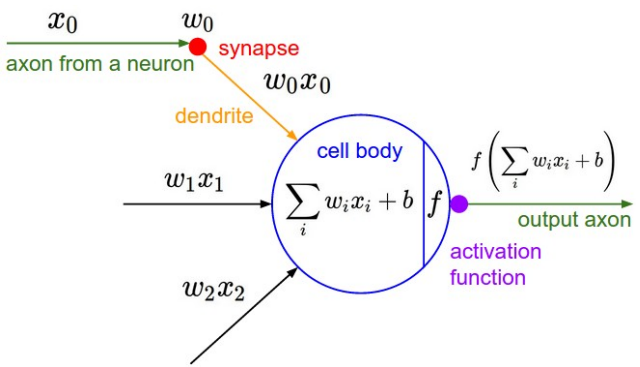
$$z = \sum_{i=0}^{N} w_i x_i + b$$

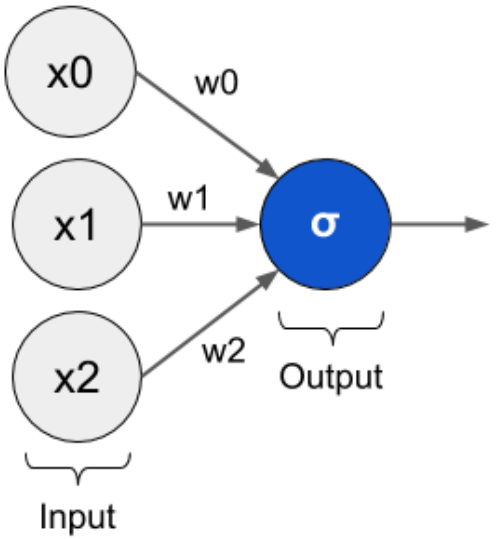
- Seguido de uma função de ativação f



Perceptron

- Pode ser representado como uma rede neural de 2 camadas
  - Camada 1: Entrada. A dimensão dessa camada é definida de acordo com o dado.
  - Camada 2: Saída. Dimensão diretamente ligada ao problema. Por exemplo, para um problema de classificação a dimensão equivale ao número de classes.

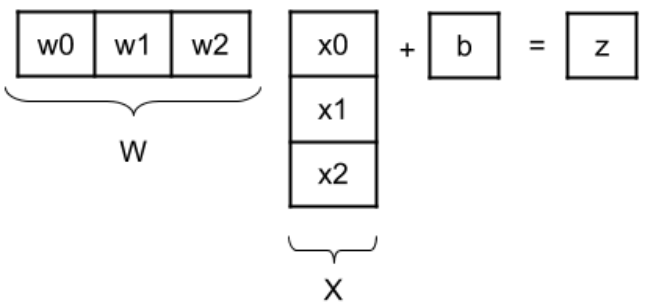


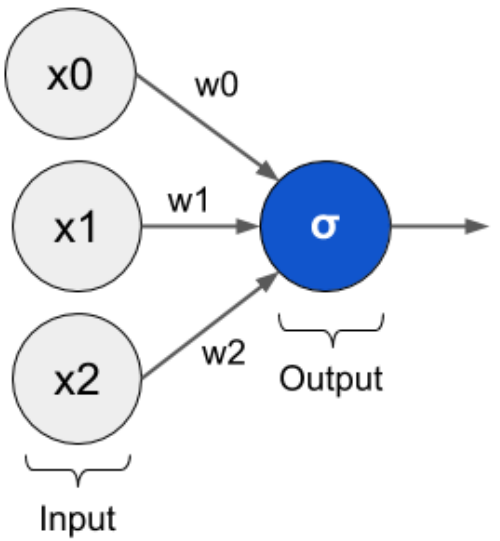


Perceptron

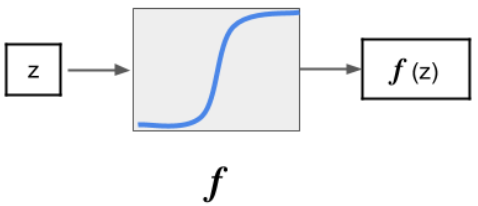
$$z = \sum_{i=1}^{N} w_i x_i + b$$

- W é o peso de cada ligação
- b é o bias

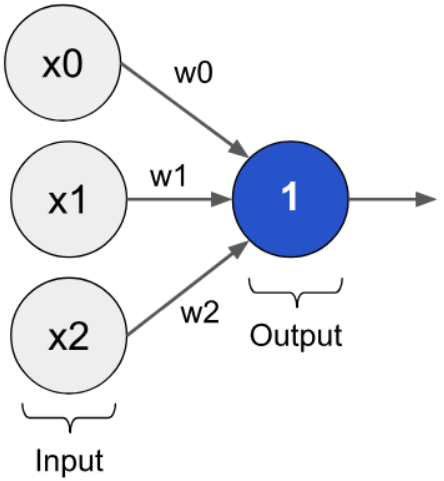




Ativações

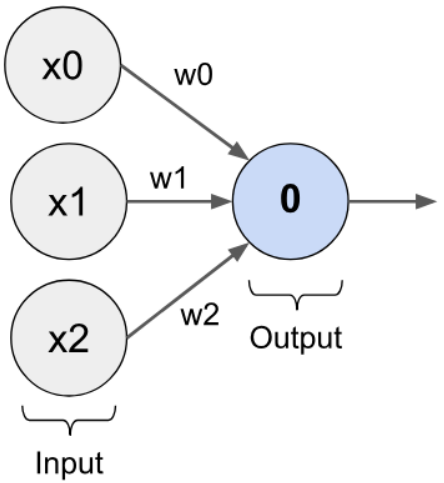


f

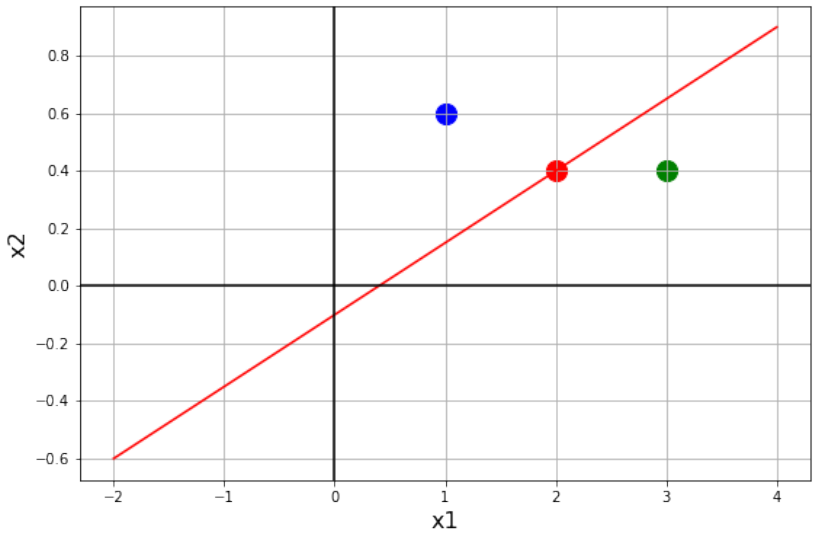


- f é a função de ativação
- Por exemplo:

$$f(z) = \begin{cases} 1, if\, z \geqslant 0 \\ 0, otherwise \end{cases}$$



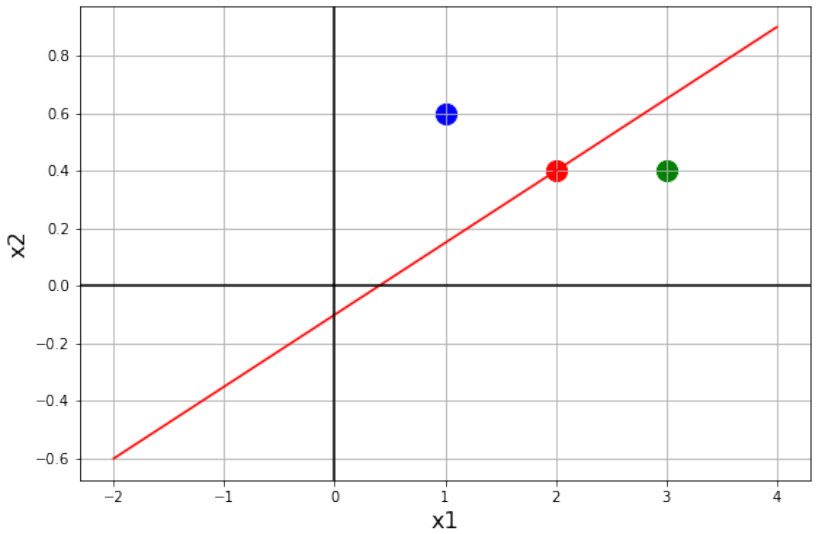
Ativações



- $f(z) = \begin{cases} 1, if\, z \geqslant 0 \\ 0, otherwise \end{cases}$
- $z = -1x_1 + 4x_2 + 0.4$
- $z = -1 \times 1 + 4 \times 0.6 + 0.4 = 1.8$
- $f(\mathbf{z}) = \mathbf{1}$
- $z = -1 \times 3 + 4 \times 0.4 + 0.4 = -1$
- $f(\mathbf{z}) = \mathbf{0}$

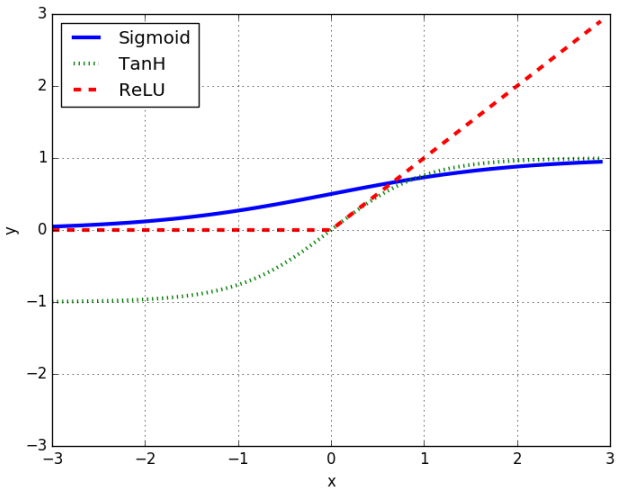
Ativações

- $f(z) = \begin{cases} 1, if\, z \geqslant 0 \\ 0, otherwise \end{cases}$
- $z = -1x_1 + 4x_2 + 0.4$



- Essa função de ativação permite maior controle sobre a saída do perceptron. A saída z que antes poderia assumir qualquer valor $z = \{-\infty, ..., \infty\}$ agora está limitada a $z = \{0, 1\}$

Ativações



- Sigmoid (σ)
  $f(x) = \frac{1}{1+e^{-x}}$
- TanH
  $f(x) = 2\sigma(2x) - 1$
- ReLU
  $f(x) = max(0, x)$

- Entre outras
  - Threshold, Leaky ReLU, Maxout, Linear, etc.
- Em redes com mais de uma camada escondida, a adição de uma função de ativação não linear permite que a rede modele dependências não lineares.

Perceptron como Classificador Linear

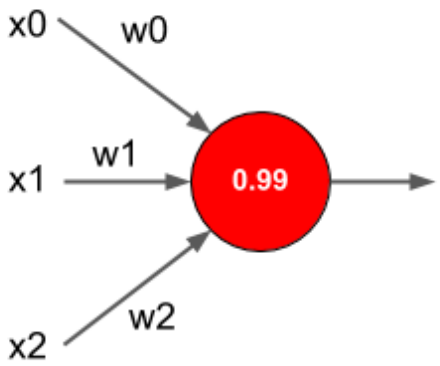
- Função de mapeamento do neurônio

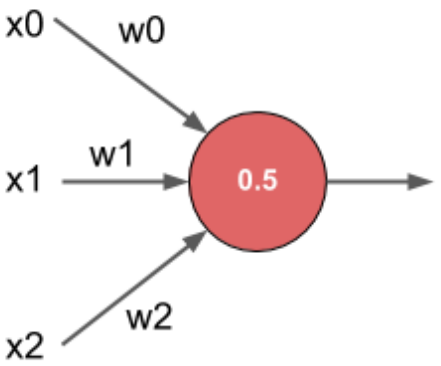
$$f(\sum_{i=0}^{N} w_i x_i + b)$$

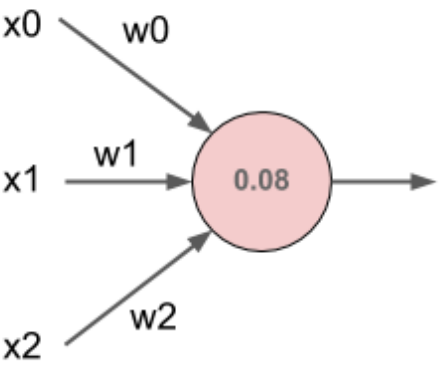
- O neurônio faz um mapeamento linear, permitindo que seja possível treiná-lo para encontrar os parâmetros da fronteira de decisão tal qual um classificador linear para um problema de classificação binária $Y = \{0, 1\}$.

Perceptron como Classificador Linear

- De forma abstrata, um perceptron interpreta as suas entradas, e libera uma ativação com uma determinada força.

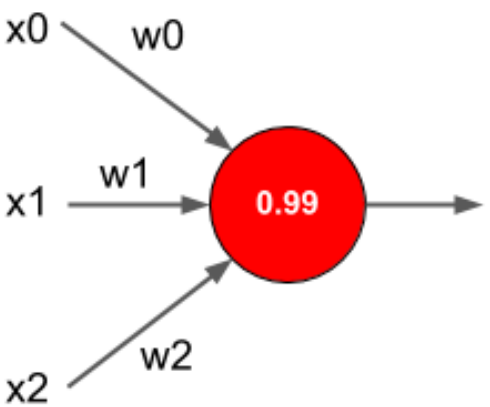


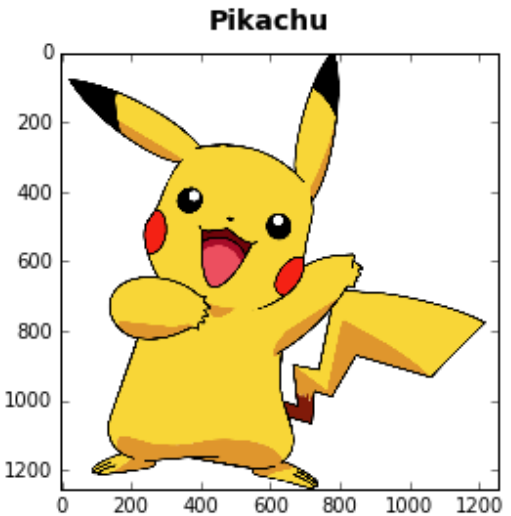




Perceptron como Classificador Linear

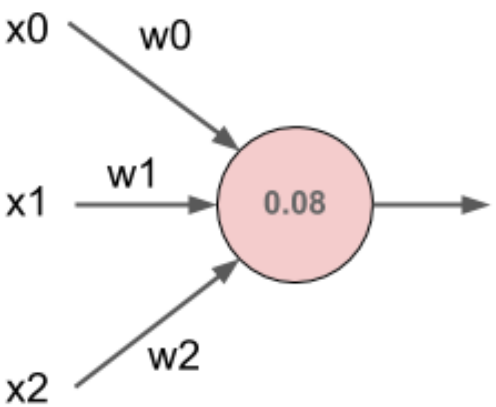
- Exemplo: você treinou um perceptron para fazer uma classificação binária, associando um rótulo numérico para cada classe $Y = \{1 : \textit{Pikachu}, 0 : \textit{Bulbasauro}\}$. Quanto mais próxima a ativação for de 1, mais certeza o perceptron tem da classe "Pikachu".

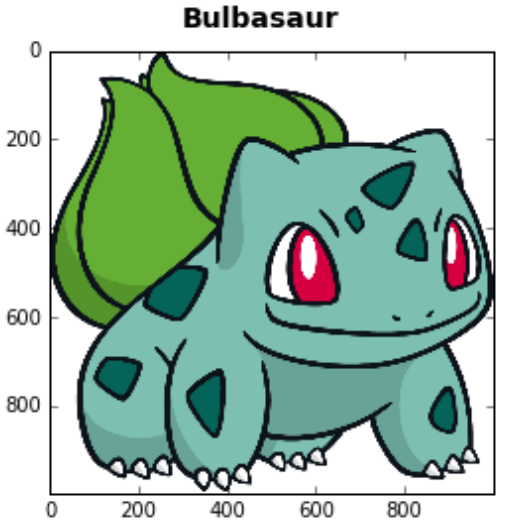




Perceptron como Classificador Linear

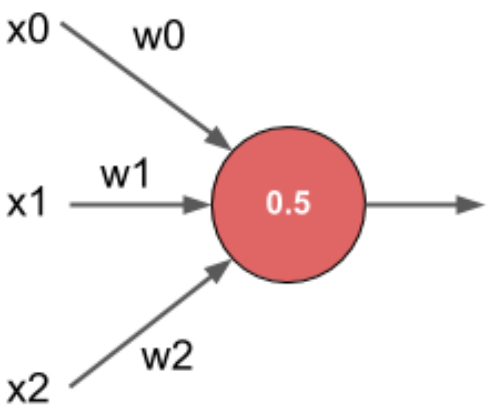
- Exemplo: você treinou um perceptron para fazer uma classificação binária, associando um rótulo numérico para cada classe $Y = \{1 : \textit{Pikachu}, 0 : \textit{Bulbasauro}\}$. Quanto mais próxima a ativação for de 0, mais certeza o perceptron tem da classe "Bulbasauro".

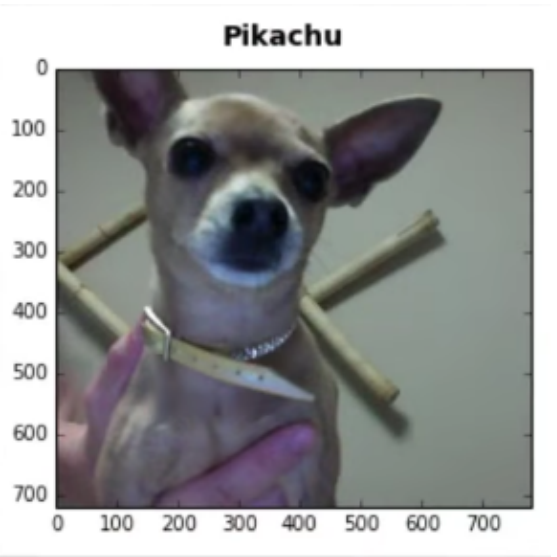




Perceptron como Classificador Linear

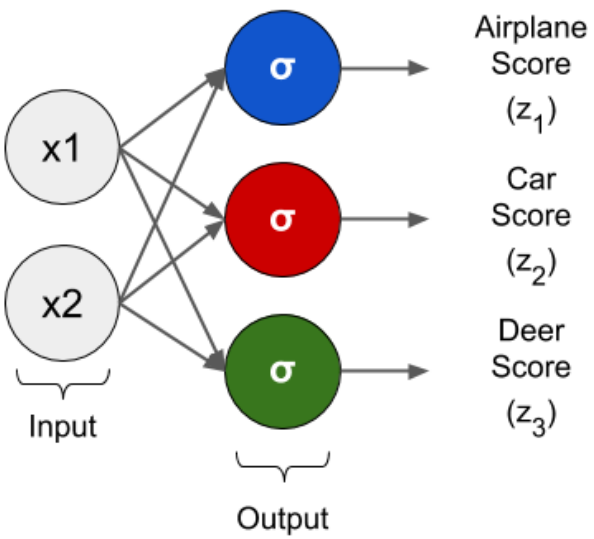
- Exemplo: você treinou um perceptron para fazer uma classificação binária, associando um rótulo numérico para cada classe $Y = \{1 : \textit{Pikachu}, 0 : \textit{Bulbasauro}\}$. Valores intermediários indicam que a imagem está na **zona de dúvida** da classificação.

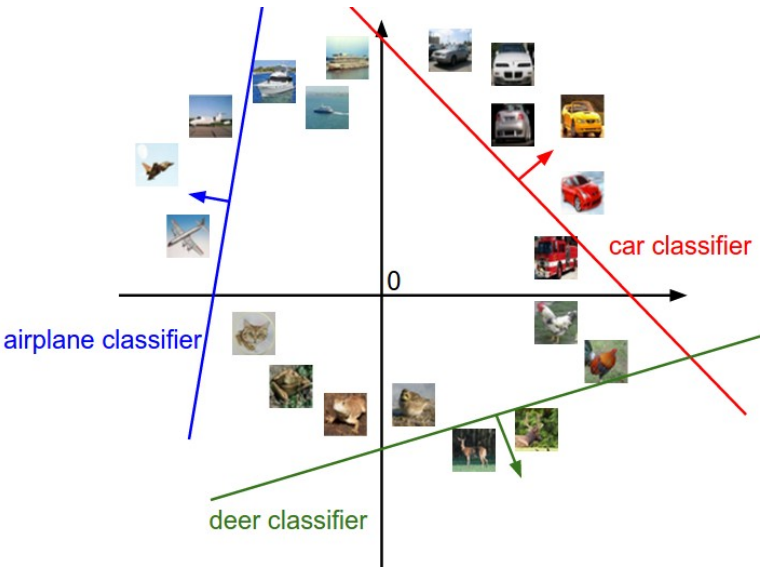




Perceptron como Classificador Linear

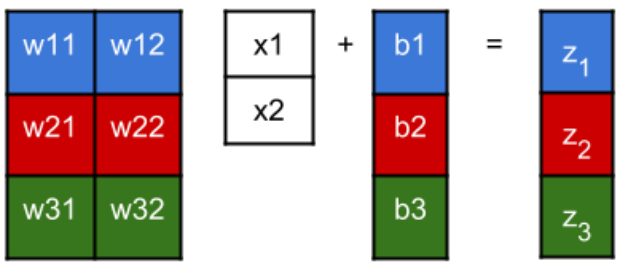
- Para problemas com múltiplas classes, é necessário especializar cada perceptron em uma classe específica.
- Cada perceptron vai otimizar os parâmetros da fronteira de decisão da classe que ele deve se especializar.

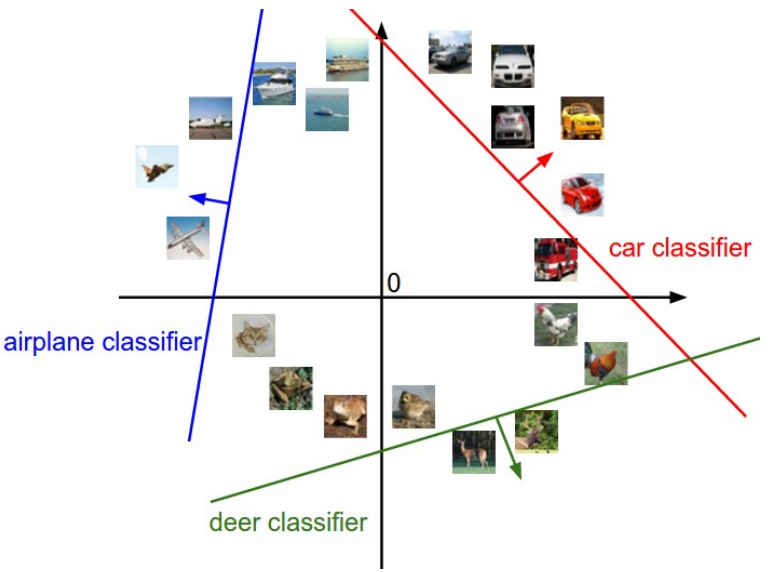




Perceptron como Classificador Linear

- Na representação matricial, isso significa que a matriz de pesos W tem a forma $K \times D$, sendo K o número de classes e D a dimensão do dado. E o vetor de bias b possui a forma $K \times 1$, apenas um bias por classe.
- Em outras palavras, cada linha das matrizes W e b produzem o score de uma classe específica.





Perceptron como Classificador Linear

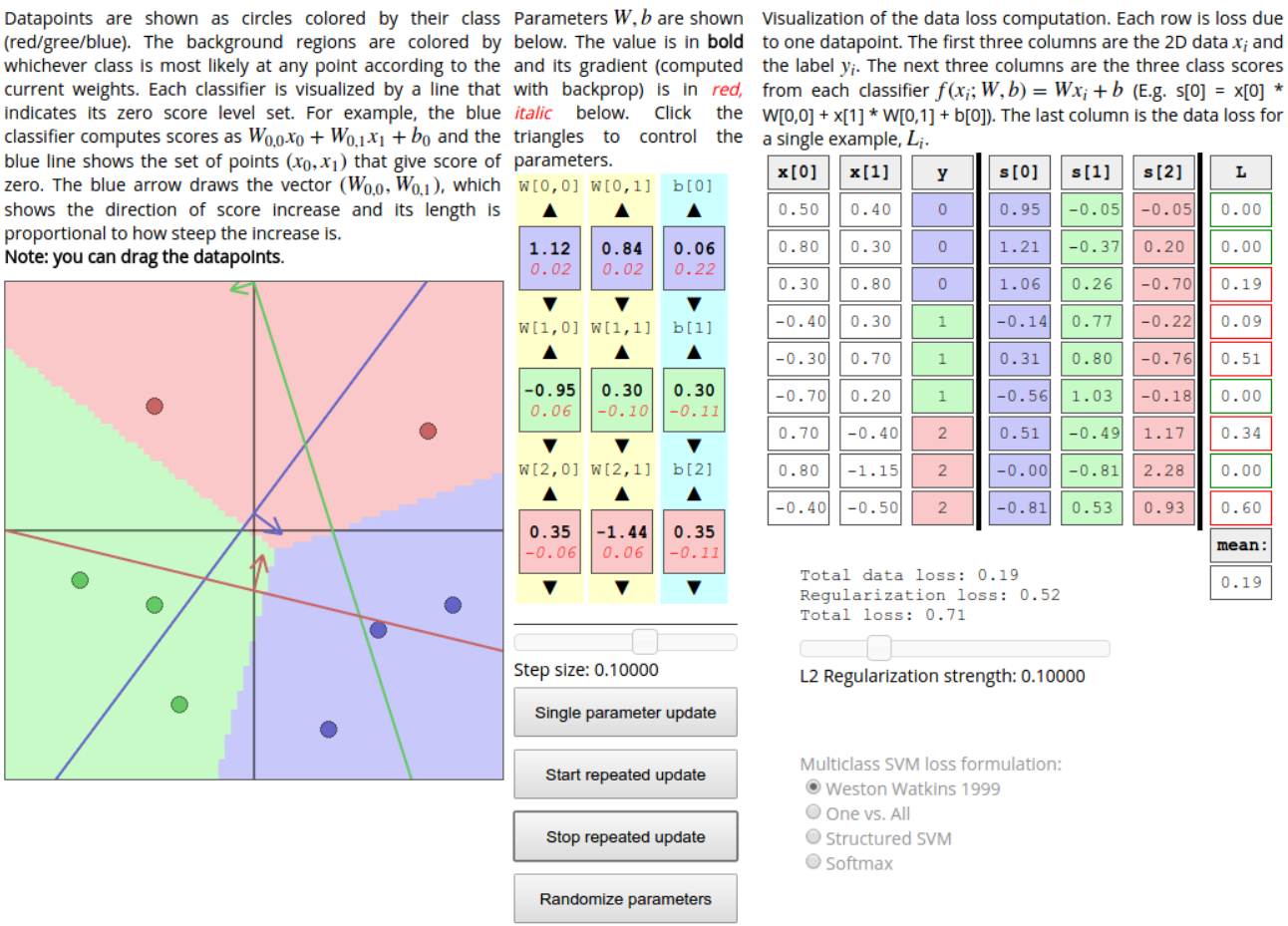


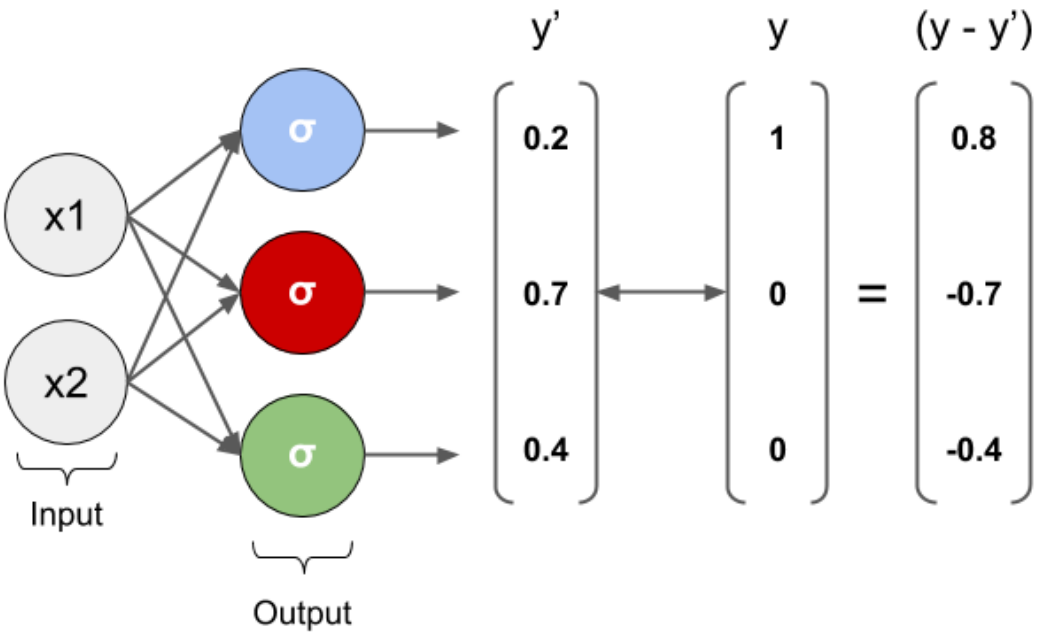
Figura: Demo interativa da Universidade de Stanford (CS231n):
http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/

Funções de Perda

- Primeiro precisamos definir uma medida de qualidade
- Isso é obtido através da função de perda (*loss function*)
  - Também conhecida como função de custo
- A escolha da *loss function* vai depender do problema sendo abordado

Funções de Perda (Exemplo)

- $\hat{y}$ representa a saída da rede (predição) dada a entrada x
  - $f(x; \mathbf{W}, \mathbf{b})$
- y é o rótulo da amostra x
- $(y - \hat{y})$ é a função de custo



Funções de Perda

- $\hat{y}$ representa a saída da rede (predição) dada a entrada x
  - $f(x; \mathbf{W}, \mathbf{b})$
- y é o rótulo da amostra x

1. Erro quadrático
   - $\mathcal{L}(W, b; x, y) = \frac{1}{2} \|\hat{y} - y\|^2$
2. Log-Loss (Cross-Entropy)
   - $\mathcal{L}(W, b; x, y) = -\sum_x y \log \hat{y}$
3. Entre (inúmeras) outras
   - Kullback-Leibler, Hinge, Huber, Mean Absolute Error, ...
   - Você pode customizar a sua *loss* (os frameworks dão suporte a isso)

Otimização

- O nosso objetivo então é minimizar essa função de custo
- Como pôde ser visto, a função de custo $\mathcal{L}(W, b; x, y)$ é baseada nas variáveis da rede:
  - pesos W
  - bias b
- Logo, a minização dessa função se dá ao mudar essas variáveis baseado no erro que ela gera

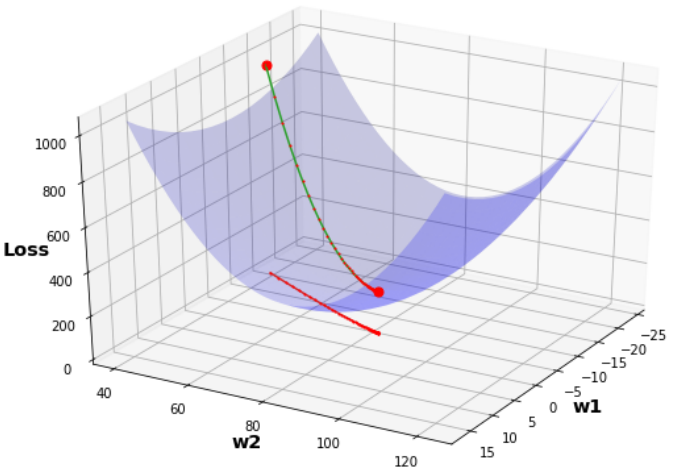
Otimização

Figura: Analogia do processo de otimização[1].

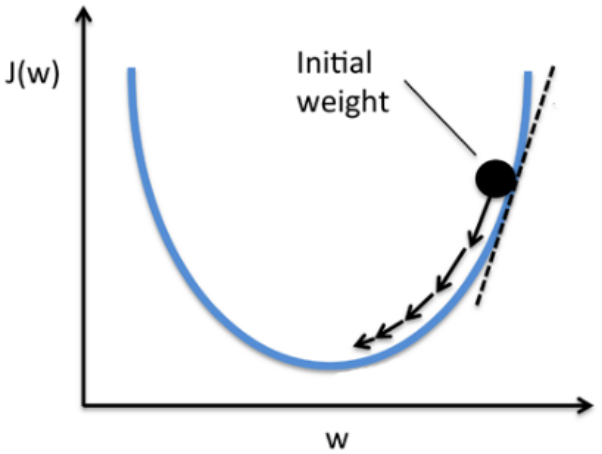
[1] http://cs231n.stanford.edu/slides/2016/winter1516_lecture3.pdf

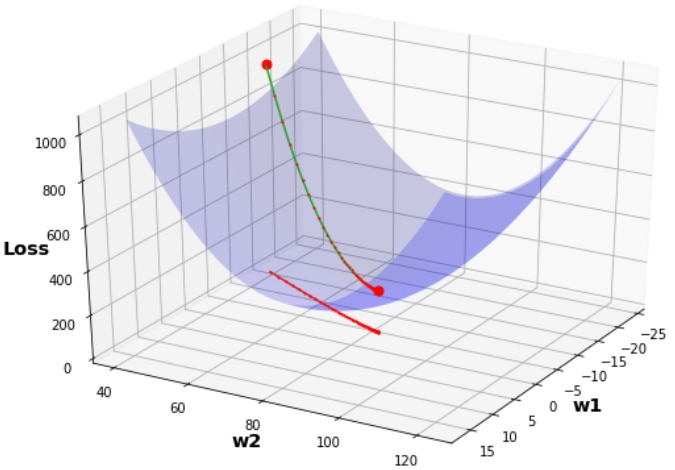
Otimização

- Intuitivamente, o nosso "medidor de altura"é uma função da *loss* em relação aos pesos.



Otimização

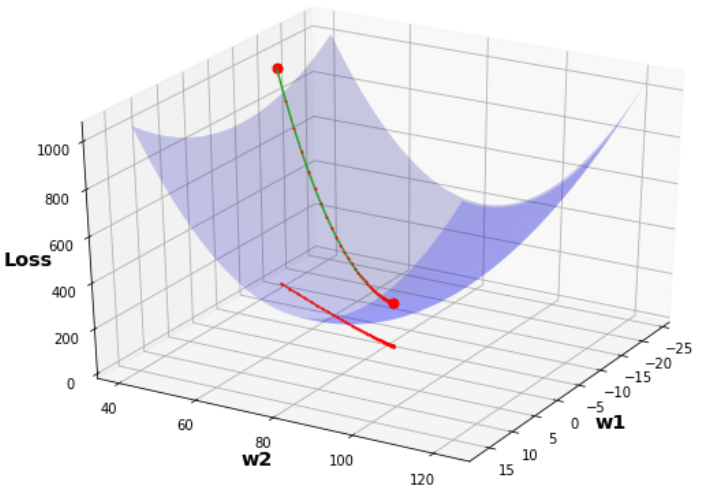




- A direção é dada pela derivada do custo em relação ao peso.
- Para múltiplas dimensões o vetor de derivadas parciais é chamado de **gradiente**.

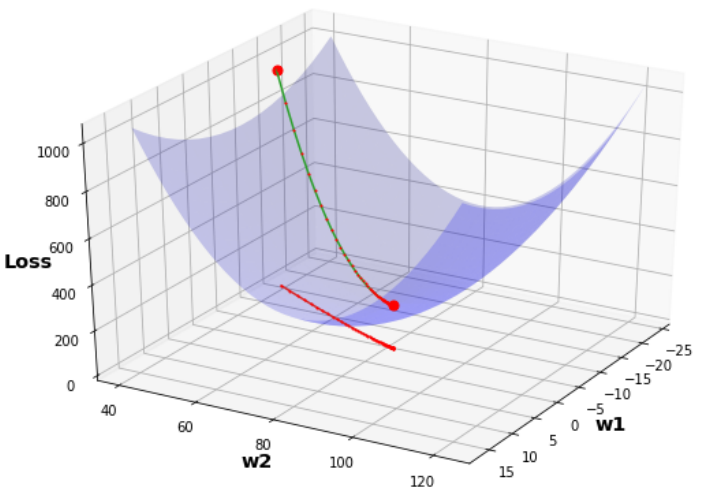
Otimização

- Em outras palavras, eu consigo alterar os pesos e detectar como a função de perda foi impactada pela minha alteração.
- $$\frac{\partial \mathcal{L}(W,b)}{\partial W_{ij}^{(L)}}$$



Otimização

- Isso vai indicar:
  - O sinal: se o peso deve aumentar ou diminuir
  - A magnitude: o quanto aquele peso deve ser ajustado.
- $\frac{\partial \mathcal{L}(W,b)}{\partial W_{ij}^{(L)}}$



Otimização

- Descida do gradiente "*vanilla*"

$$W_{ij}^{(L)} = W_{ij}^{(L)} - \alpha \frac{\partial \mathcal{L}(W,b)}{\partial W_{ij}^{(L)}}$$

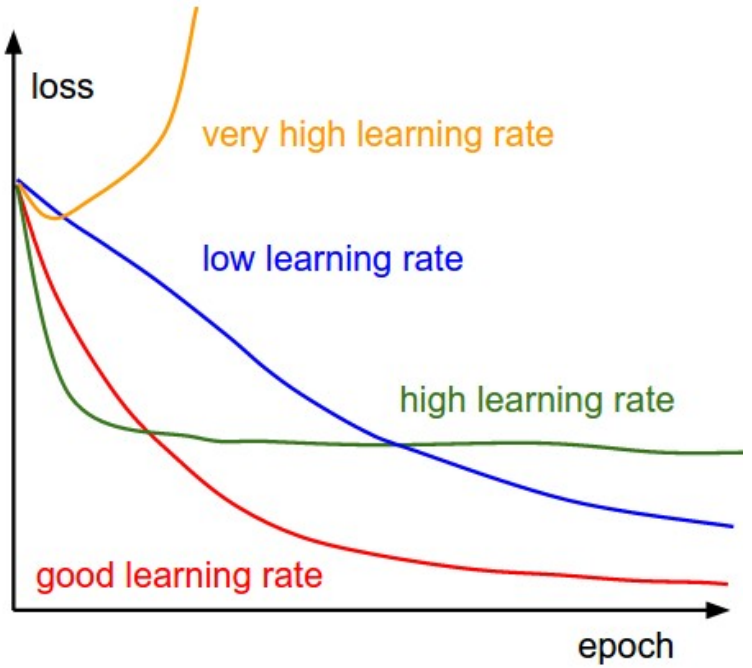
$$b_{ij}^{(L)} = b_{ij}^{(L)} - \alpha \frac{\partial \mathcal{L}(W,b)}{\partial b_{ij}^{(L)}}$$

- Sendo α a taxa de aprendizado (*learning rate*)
  - Um fator que interfere na magnitude da alteração dos pesos.

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

Otimização

- Taxa de Aprendizado: Define o tamanho do "passo"de otimização



Perceptron como Classificador Linear

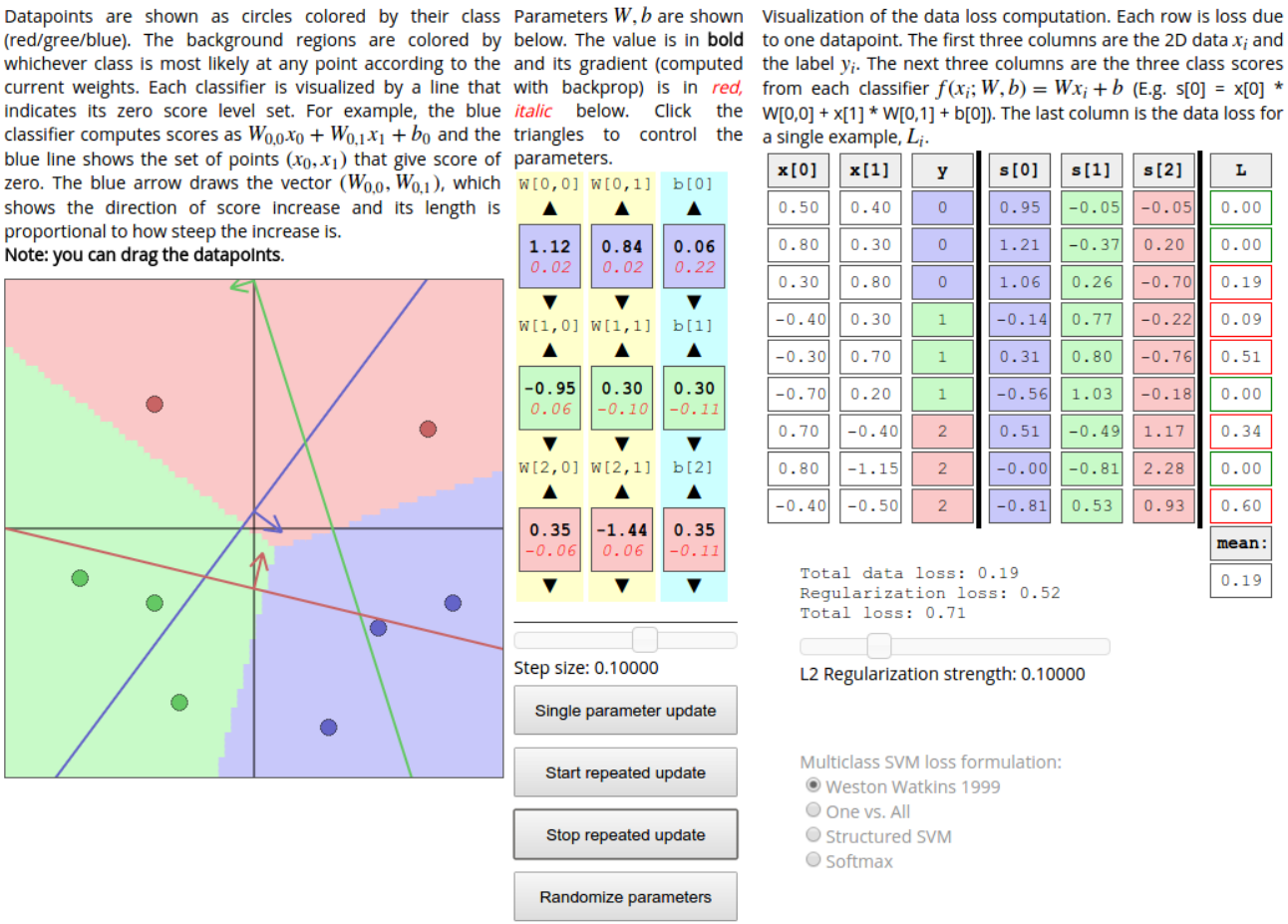


Figura: Demo interativa da Universidade de Stanford (CS231n):
http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/

Otimização

- Métodos de minimização mais populares
  - Gradiente descendente (*Stochastic Gradient Descent* (SGD)) [3]
  - Adam [4]
  - RMSProp [5]
  - Adadelta [6]
  - Entre outros

Otimização

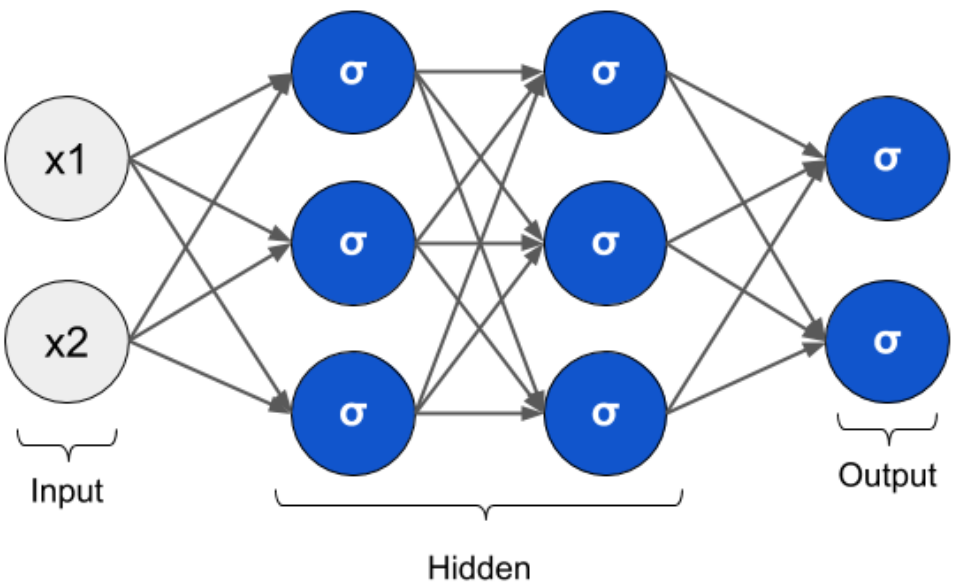
Gif - Algoritmos de Minimização

gifs/optimization-2D.gif

gifs/optimization-3D.gif

Multi-Layer Perceptron

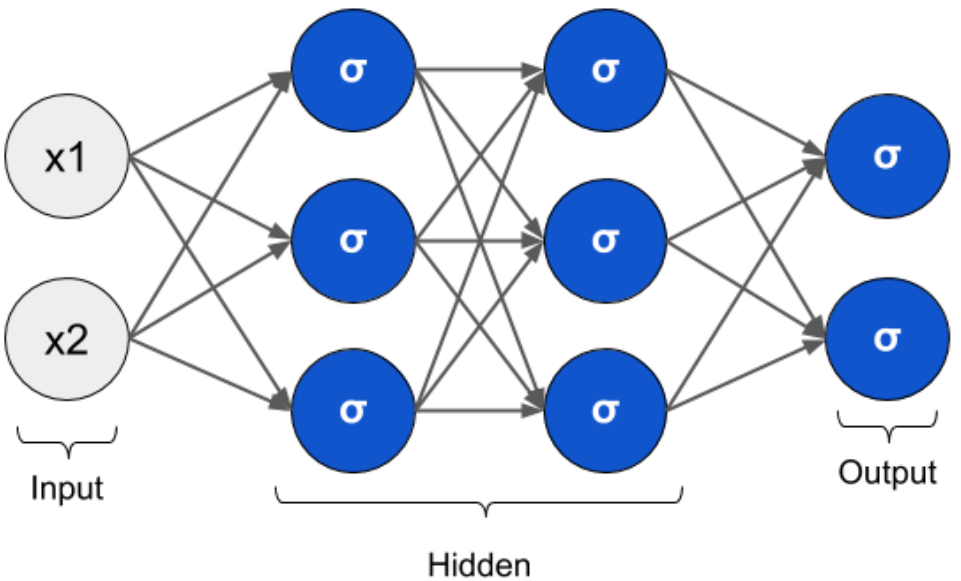
- Além das camadas de entrada e saída, temos também as camadas escondidas (*hidden layers*).



Multi-Layer Perceptron

- Considerando o que já sabemos sobre o perceptron, uma arquitetura multicamada é uma função composta

$$f(f(f(x; W^0, b^0); W^1, b^1); W^2, b^2)$$



Multi-Layer Perceptron

- Considerando o que já sabemos sobre o perceptron, uma arquitetura multicamada é uma função composta

$$f(f(f(x; W^0, b^0); W^1, b^1); W^2, b^2)$$

$$\begin{cases} a^0 = f(x; W^0, b^0) \\ a^1 = f(a^0; W^1, b^1) \\ a^2 = f(a^1; W^2, b^2) \end{cases}$$

Multi-Layer Perceptron

- Teorema de Aproximação Universal [2]
  - "*Uma rede neural feed forward com apenas uma camada (escondida) é suficiente para representar qualquer função, mas a camada pode ser inviavelmente grande e pode falhar em aprender e generalizar corretamente.*"

[2] Ian Goodfellow, Deep Learning Book. http://www.deeplearningbook.org/contents/mlp.html

Multi-Layer Perceptron

- Teorema de Aproximação Universal [2]
  - "*Uma rede neural feed forward com apenas uma camada (escondida) é suficiente para representar **qualquer função**, mas a camada pode ser inviavelmente grande e pode falhar em aprender e generalizar corretamente.*"
- **Inclusive não linearidades!** (considerando ativações não lineares na camada escondida)

[2] Ian Goodfellow, Deep Learning Book. http://www.deeplearningbook.org/contents/mlp.html

Multi-Layer Perceptron

- Teorema de Aproximação Universal

Qual desses modelos você acha melhor?

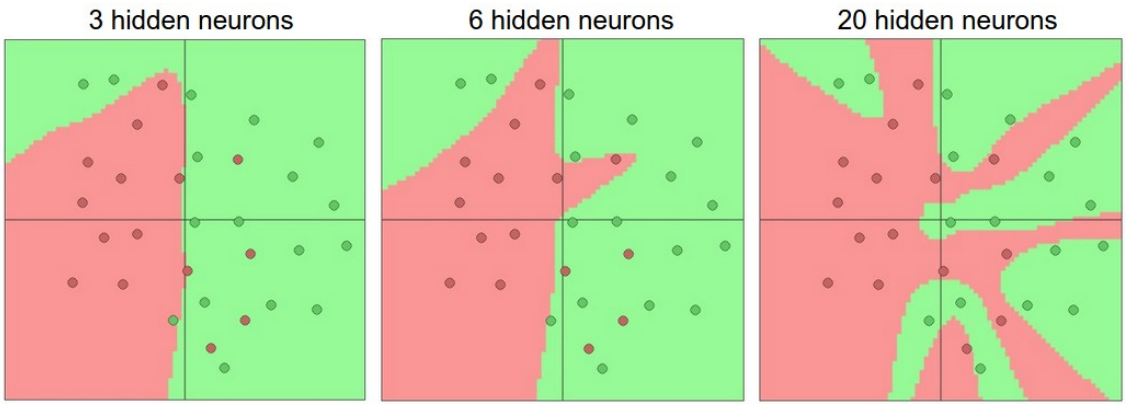


Figura: ConvnetJS demo [3]

[3] ConvnetJS demo https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

Multi-Layer Perceptron

- Teorema de Aproximação Universal
- A capacidade das redes neurais de aproximar qualquer função contínua (inclusive as mais complexas), a torna muito vulnerável a **overfitting**.

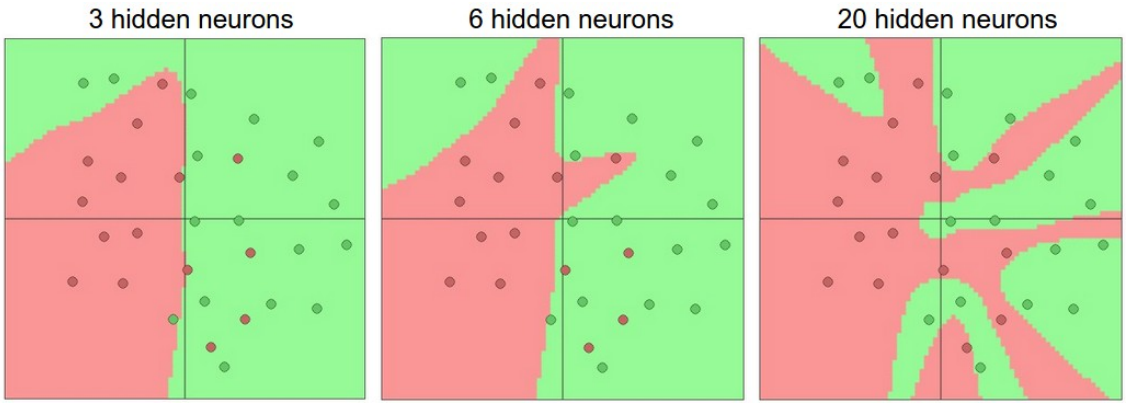


Figura: ConvnetJS demo [3]

[3] ConvnetJS demo https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

Regularização

- Existem artifícios para controlar a complexidade do modelo.

1. Regularização L2
   - Adiciona o termo $\frac{1}{2}\lambda W^2$ à função objetivo para todos os pesos da rede
   - λ controla a força da regularização
   - Pesos decaem em direção a 0

2. Dropout
   - Durante o treinamento, cada neurônio tem uma probabilidade p de estar ativo.
   - Neurônios inativos não são atualizados naquela iteração
   - Não há dropout em tempo de teste

Regularização

- Efeito de aplicar regularização L2 no modelo com 20 neurônios

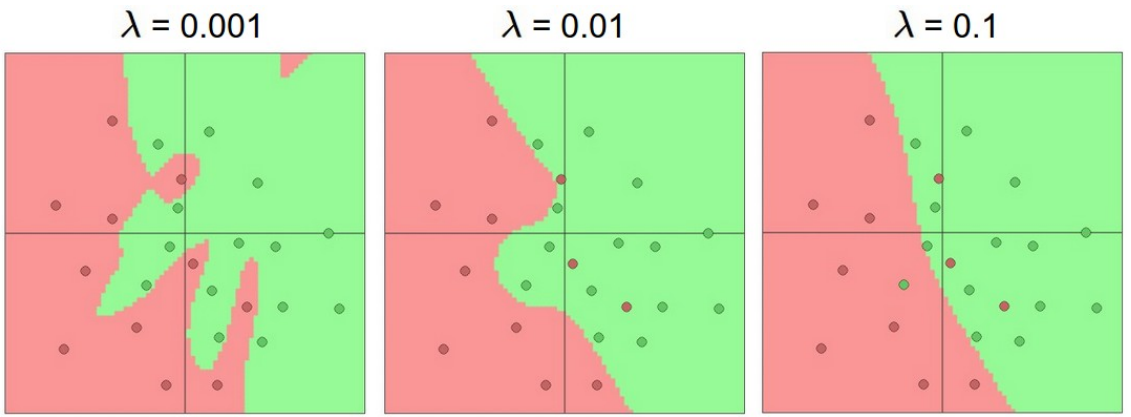
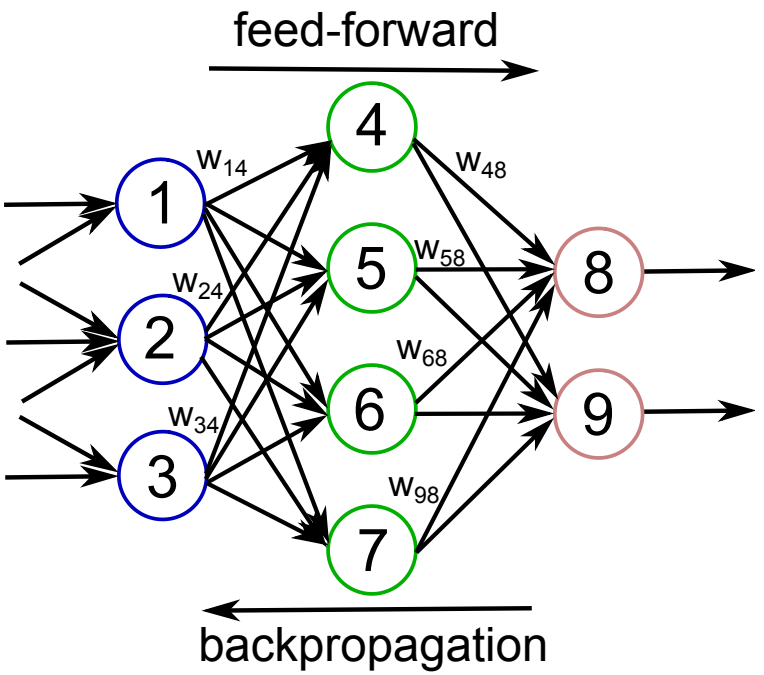


Figura: ConvnetJS demo [3]

[3] ConvnetJS demo https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

Redes Neurais Artificiais

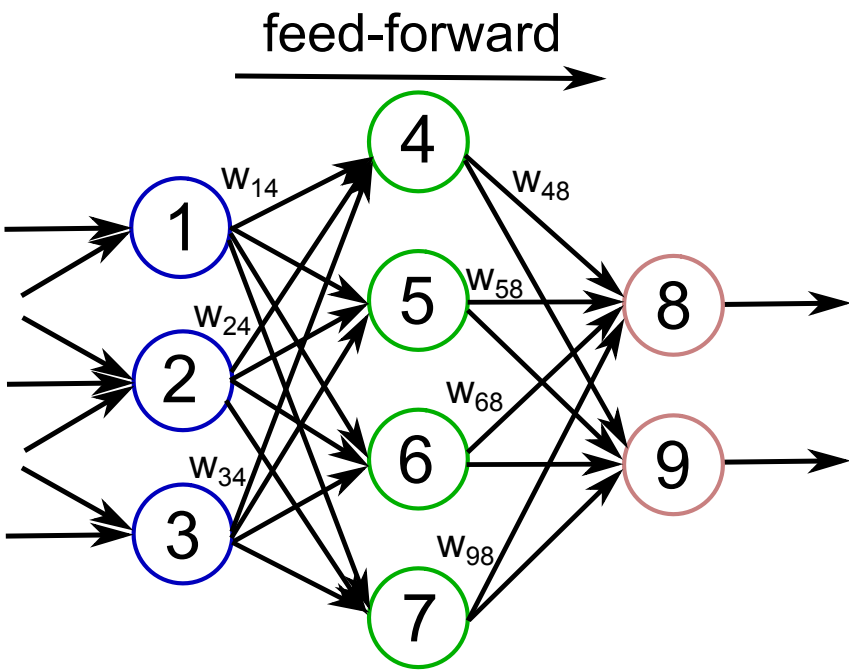
- O processo de aprendizado pode ser dividido em dois passos:
  - Feed-forward: recebe os dados e gera a saída final da rede
  - Backpropagation: ajusta os pesos de acordo com o gradiente



Feed-forward

$$z = \sum_i w_i x_i + b$$

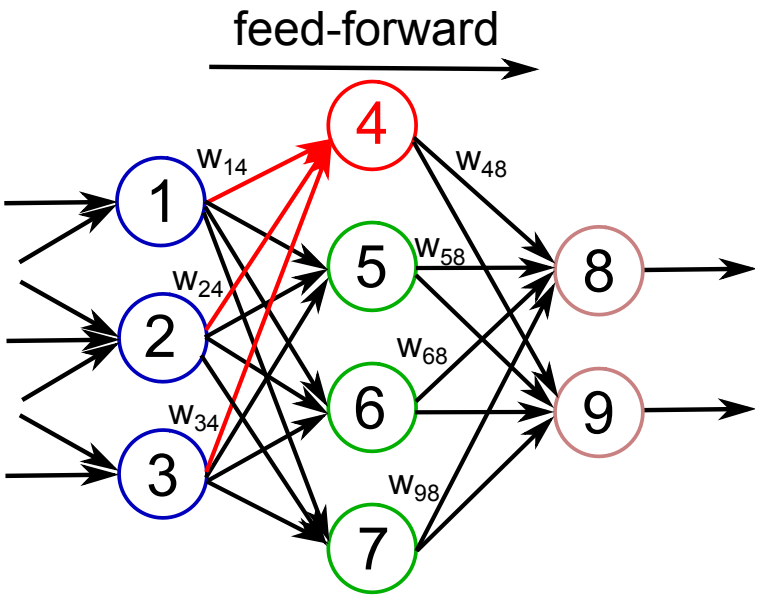
$$a = f(z)$$



Feed-forward

$$z^4 = W^{14}a^1 + W^{24}a^2 + W^{34}a^3 + b^4$$

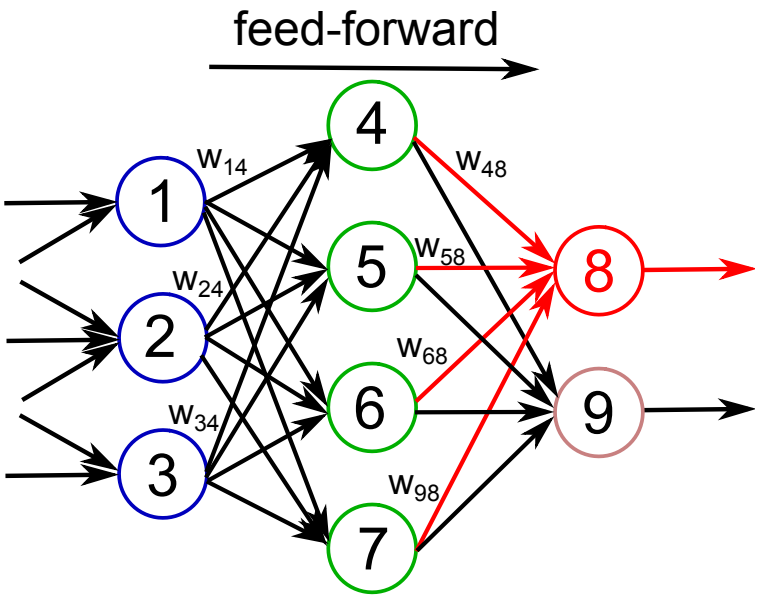
$$a^4 = f(z^4)$$



Feed-forward

$$z^8 = W^{48}a^4 + W^{58}a^5 + W^{68}a^6 + W^{78}a^7 + b^8$$

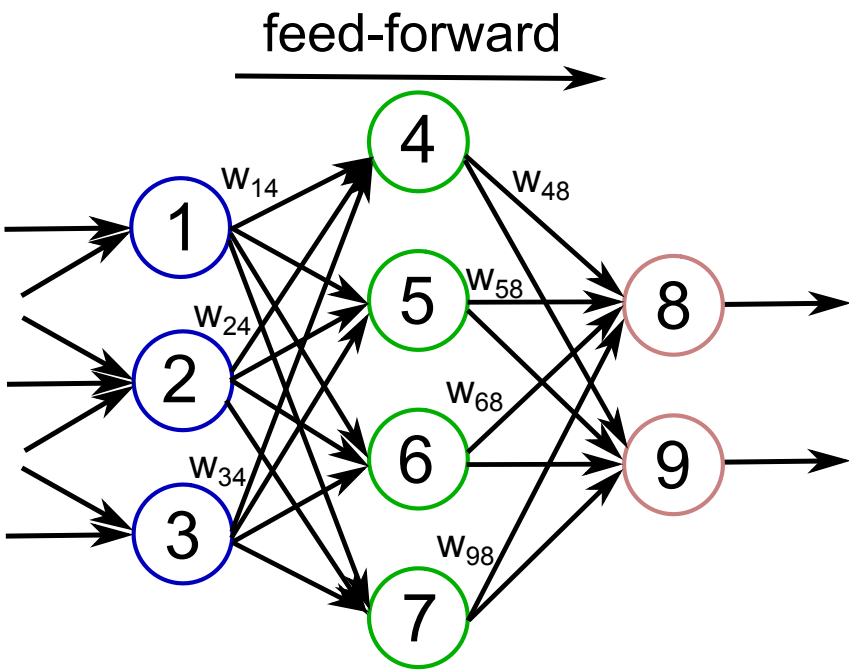
$$a^8 = f(z^8)$$



Feed-forward

$$z^{(L+1)} = W^{(L)} a^{(L)} + b^{(L)}$$

$$a^{(L+1)} = f(z^{(L+1)})$$



Backpropagation

- Podemos calcular a velocidade com que o erro muda (usando a *loss function*) à medida que mudamos as saídas dos neurônio
- Para mudar as saídas, precisamos mudar os pesos, o que é feito através do algoritmo de otimização
- Como vimos, precisamos das derivadas parciais
- Backpropagation calcula essas derivadas parciais!

Backpropagation

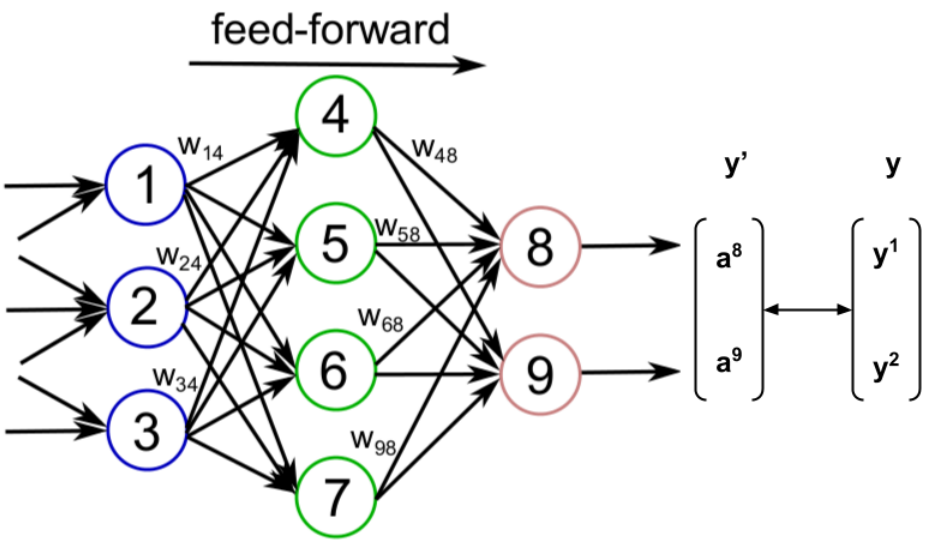
- Suponhamos:
  - Erro quadrático: $\mathcal{L}(W, b; x, y) = \frac{1}{2}\|y - a^{(L)}\|^2$
  - Sigmóide como função de ativação: $f(z) = \sigma(z) = \frac{1}{1+\exp^z}$

Backpropagation

- Passo a passo, temos:

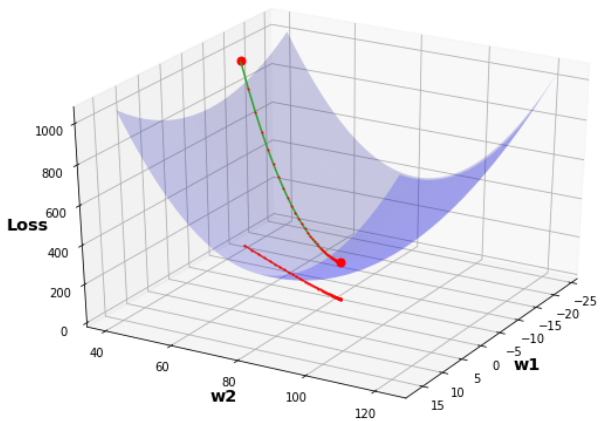
1. Calcula o erro da rede

$$\mathcal{L}(W, b; x, y) = \frac{1}{2}\|y - a^{(L)}\|^2$$



Backpropagation

2 Calcula o erro dos neurônios da última camada [4]



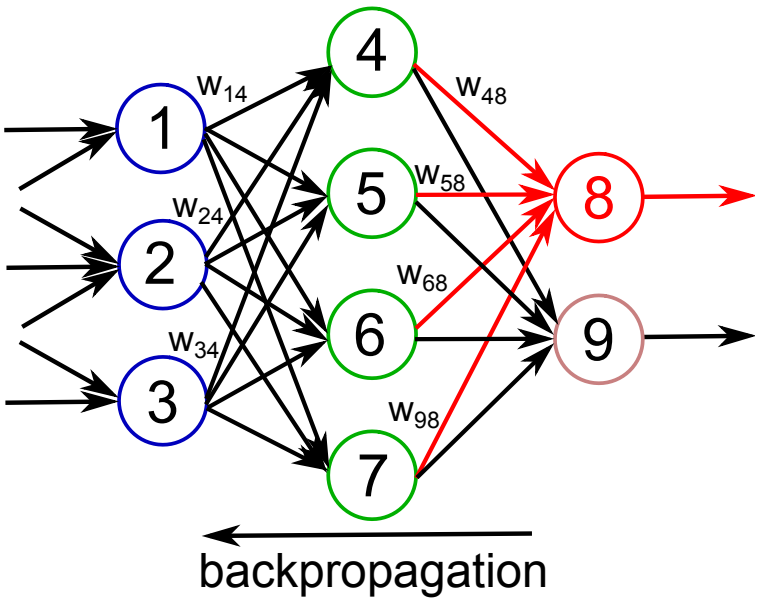
- $\frac{\partial \mathcal{L}}{\partial W^{(L)}} = a^{(L-1)} a^{(L)} (1 - a^{(L)})(y - a^{(L)})$
- $\delta^{(L)} = a^{(L)} (1 - a^{(L)})(y - a^{(L)})$
- $\frac{\partial \mathcal{L}}{\partial W^{(L)}} = a^{(L-1)} \delta^{(L)}$

[4] Backpropagation calculus | Appendix to deep learning chapter 3 https://youtu.be/tIeHLnjs5U8

Backpropagation

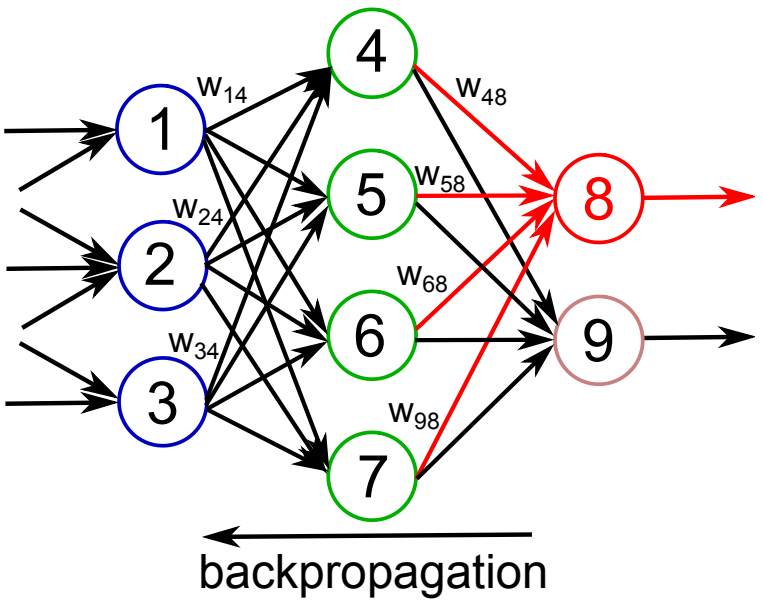
- Propagando o erro para neurônios na última camada

- $\delta^8 = a^8(1 - a^8)(y - a^8)$



Backpropagation

- Propagando o erro para neurônios na última camada
- $W^{(L)} = W^{(L)} - \alpha \frac{\partial \mathcal{L}}{\partial W^{(L)}}$
- $\frac{\partial \mathcal{L}}{\partial W^{(L)}} = a^{(L-1)} \delta^{(L)}$
- $W_{48} = W_{48} - \alpha a^4 \delta^8$
- $W_{58} = W_{58} - \alpha a^5 \delta^8$
- $W_{68} = W_{68} - \alpha a^6 \delta^8$
- $W_{78} = W_{78} - \alpha a^7 \delta^8$



Backpropagation

3 Calcula os erros das camadas escondidas

- Mais regra da cadeia!

$$\frac{\partial \mathcal{L}}{\partial W^{(L-1)}}$$

- Iterando em todos os neurônios da camada L, o $\delta^{(L-1)}$ de qualquer neurônio da camada $L-1$ é dado por:

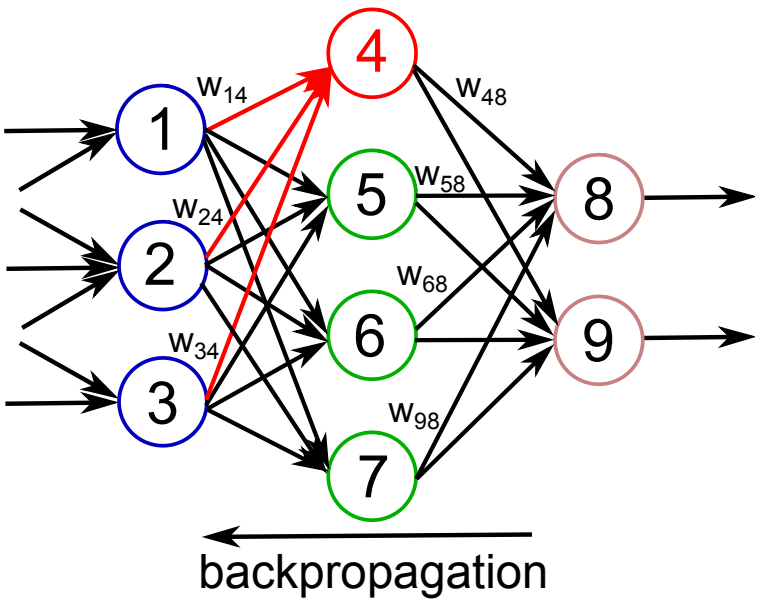
$$\delta^{(L-1)} = (\sum_i \delta_i^{(L)} W_i^{(L)}) f'(z^{(L-1)})$$

Backpropagation

- Propagando para um neurônio de camada escondida

- $\delta^4 = (\delta^8 W_{48} + \delta^9 W_{49}) a^4 (1 - a^4)$

- $W_{14} = W_{14} - \alpha \delta^4 a^1$
- $W_{24} = W_{24} - \alpha \delta^4 a^2$
- $W_{34} = W_{34} - \alpha \delta^4 a^3$

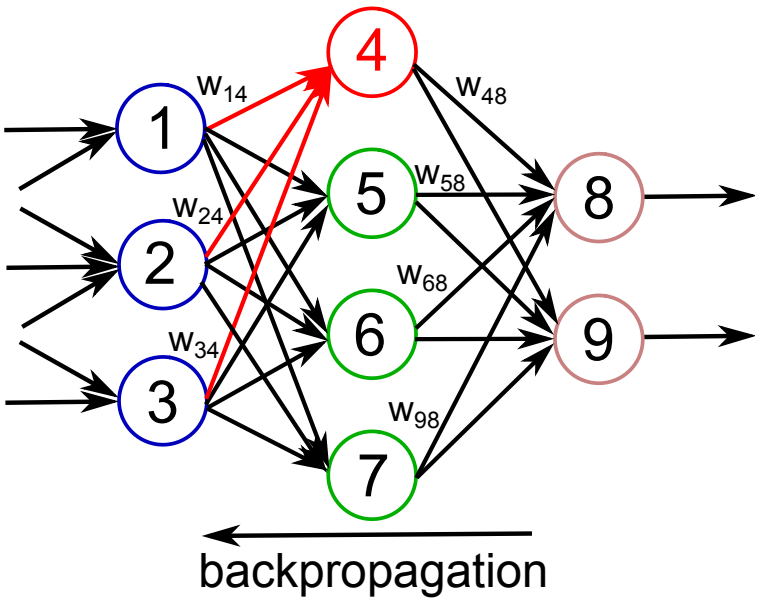


Backpropagation

- Propagando para um neurônio de camada escondida em relação ao bias

- $\delta^4 = (\delta^8 W_{48} + \delta^9 W_{49}) a^4 (1 - a^4)$

- $b_4 = b_4 - \alpha \delta^4 * 1$



Procedimento de Treinamento

- **Configurações Iniciais** (O que sabemos até então)
  - Arquitetura
  - Função de Perda
  - Regularização
  - Otimizador
- Tudo isso é definido experimentalmente (conhecimento prévio pode ajudar bastante) e é diretamente ligado ao problema em questão

Procedimento de Treinamento

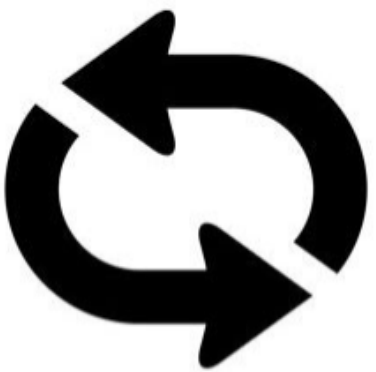
- Dinâmica **iterativa** de treinamento
  - A otimização é realizada a cada **passo** de treinamento (analogia da montanha).
  - De forma prática que é um passo de treinamento?

Procedimento de Treinamento

Batch

Iteração

Época

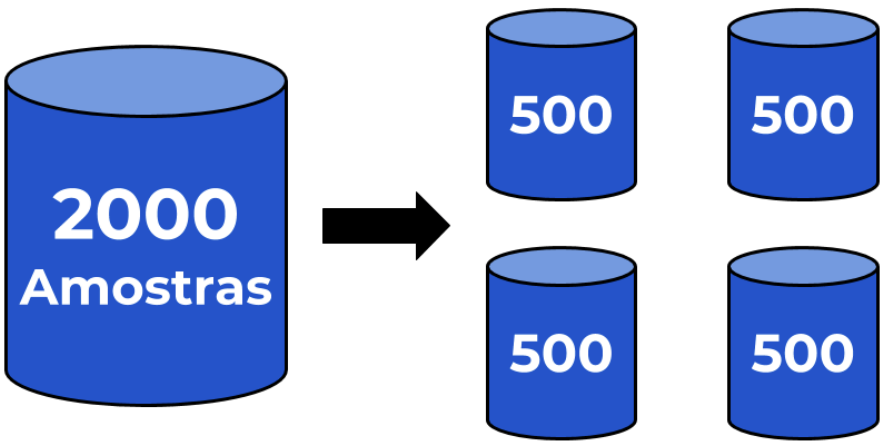


Batch

- Todas as amostras do dataset.
- Quantidade de amostras em um passo de treinamento.
  - Um passo: Forward + Backward
  - Em outras palavras, quantas amostras o modelo vai ver antes de calcular a loss e otimizar.

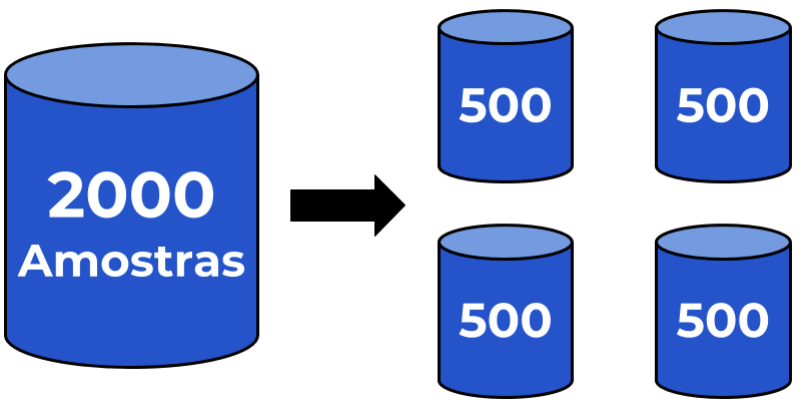
Batch

- Por exemplo, dado um dataset de 2000 amostras, poderíamos dividi-lo em mini-batches de 500 amostras.
  - *batch_size* é um hiperparâmetro do treinamento (e geralmente se refere a mini-batches)
  - Nesse caso, $batch\_size = 500$



Época

- Uma época é completada quando todas as amostras do dataset foram vistas pelo modelo.
- No nosso exemplo, 1 época se passa a cada 4 iterações.



Época

- *num_epochs* também é hiperparâmetro do treinamento.
- **Mas se ao final de uma época o modelo viu todas as amostras do dataset, por que precisamos de mais de uma época?**

Época

- Modelos profundos são *data hungry* (famintos por dados).
- Na maioria dos casos, uma época não é suficiente para o modelo convergir.
  - Quantidade insuficiente de amostras.
- Nessas condições, é necessário apresentar as amostras mais de uma vez ao modelo.

Agenda

Frameworks

Software	Creator	Software license[a]	Open source	Platform	Written in	Interface	OpenMP support	OpenCL support	CUDA support	Parallel execution (multi node)	Automatic differentiation[c]	Has pretrained models	Recurrent nets	Convolutional nets	RBM/DBNs	Metal support
ro|Nlie.ai	Kevin Lok	MIT	Yes	Linux, macOS, Windows	Python	Python			Yes			Yes	Yes	Yes		
BigDL	Jason Dai	Apache 2.0	Yes	Apache Spark	Scala	Scala, Python			No			Yes	Yes	Yes		
Caffe	Berkeley Vision and Learning Center	BSD license	Yes	Linux, macOS, Windows[c]	C++	Python, MATLAB, C++	Yes	Under development[D]	Yes	Yes	Yes[B]	Yes	Yes	No	?	
Deeplearning4j	Skymind engineering team; Deeplearning4j community; originally Adam Gibson	Apache 2.0	Yes	Linux, macOS, Windows, Android (Cross-platform)	C++, Java	Java, Scala, Clojure, Python (Keras), Kotlin	Yes	On roadmap[F]	Yes[B][7]	Computational Graph	Yes[B]	Yes	Yes	Yes	Yes[9]	
Chainer	Preferred Networks	MIT license	Yes	Linux, macOS, Windows		Python	No	No[10][11]	Yes	Yes	Yes	Yes	Yes			
Darknet	Joseph Redmon	Public Domain	Yes	Cross-Platform	C	C, Python	Yes	No[12]	Yes	Yes						
Dlib	Davis King	Boost Software License	Yes	Cross-Platform	C++	C++	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes	
DataMelt (DMelt)	S.Chekanov	Freemium	Yes	Cross-Platform	Java	Java	No	No	No	No	No	No	No	No	No	
DyNet	Carnegie Mellon University	Apache 2.0	Yes	Linux, macOS, Windows		C++, Python		No[13]	Yes	Yes	Yes					
Intel Data Analytics Acceleration Library	Intel	Apache License 2.0	Yes	Linux, macOS, Windows on Intel CPU[14]	C++, Python, Java	C++, Python, Java[14]	Yes	No	No	Yes	No		Yes		Yes	
Intel Math Kernel Library	Intel	Proprietary	No	Linux, macOS, Windows on Intel CPU[15]		C[16]	Yes[17]	No	No	Yes	No	Yes[18]	Yes[18]		No	
Keras	François Chollet	MIT license	Yes	Linux, macOS, Windows	Python	Python, R	Only if using Theano as backend	Under development for the Theano backend (and on roadmap for the TensorFlow backend)	Yes	Yes	Yes[19]	Yes	Yes	Yes	Yes[20]	
MATLAB + Neural Network Toolbox	MathWorks	Proprietary	No	Linux, macOS, Windows	C, C++, Java, MATLAB	MATLAB	No	No	Train with Parallel Computing Toolbox and generate CUDA code with GPU Coder[21]	No	Yes[22][23]	Yes[22]	Yes[22]	Yes[22]	With Parallel Computing Toolbox[24]	
Microsoft Cognitive Toolkit	Microsoft Research	MIT license[25]	Yes	Windows, Linux[26] (macOS via Docker on roadmap)	C++	Python (Keras), C++, Command line,[27] BrainScript[28] (.NET on roadmap[29])	Yes[30]	No	Yes	Yes	Yes[31]	Yes[32]	Yes[32]	Yes	Yes[33]	
Apache MXNet	Apache Software Foundation	Apache 2.0	Yes	Linux, macOS, Windows,[34][35] AWS, Android,[36] iOS, JavaScript[37]	Small C++ core library	C++, Python, Julia, Matlab, JavaScript, Go, R, Scala, Perl	Yes	On roadmap[38]	Yes	Yes[39]	Yes[40]	Yes	Yes	Yes	Yes[41]	
Neural Designer	Artelnics	Proprietary	No	Linux, macOS, Windows	C++	Graphical user interface	Yes	No	No	?	?	No	No	No	?	
OpenNN	Artelnics	GNU LGPL	Yes	Cross-platform	C++	C++	Yes	No	Yes	?	?	No	No	No	?	
PlaidML	Vertex.AI	AGPL3	Yes	Linux, macOS, Windows	C++, Python	Keras, Python, C++, C	No	Yes	Yes	Yes	Yes		Yes	Yes	?	Yes
PyTorch	Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan	BSD license	Yes	Linux, macOS, Windows	Python, C, CUDA	Python	Yes	Via separately maintained package[42][43][44]	Yes	Yes	Yes	Yes	Yes		Yes	
Apache SINGA	Apache Incubator	Apache 2.0	Yes	Linux, macOS, Windows	C++	Python, C++, Java	No	No	Yes	?	Yes	Yes	Yes	Yes	Yes	
TensorFlow	Google Brain team	Apache 2.0	Yes	Linux, macOS, Windows,[45] Android	C++, Python, CUDA	Python (Keras), C/C++, Java, Go, R,[46] Julia	No	On roadmap[47] but already with SYCL[48] support	Yes	Yes[49]	Yes[50]	Yes	Yes	Yes	Yes	
TensorLayer	Hao Dong	Apache 2.0	Yes	Linux, macOS, Windows,[51] Android	C++, Python,	Python	No	On roadmap[47] but already with SYCL[48] support	Yes	Yes[52]	Yes[53]	Yes	Yes	Yes	Yes	
Theano	Université de Montréal	BSD license	Yes	Cross-platform	Python	Python (Keras)	Yes	Under development[54]	Yes	Yes[55][56]	Through Lasagne's model zoo[57]	Yes	Yes	Yes	Yes[58]	
Torch	Ronan Collobert, Koray Kavukcuoglu, Clement Farabet	BSD license	Yes	Linux, macOS, Windows,[59] Android,[60] iOS	C, Lua	Lua, LuaJIT,[61] C, utility library for C++/OpenCL[62]	Yes	Third party implementations[63][64]	Yes[65][66]	Through Twitter's Autograd[67]	Yes[68]	Yes	Yes	Yes	Yes[69]	
Wolfram Mathematica	Wolfram Research	Proprietary	No	Windows, macOS, Linux, Cloud computing	C++, Wolfram Language, CUDA	Wolfram Language	Yes	No	Yes	Yes	Yes[70]	Yes	Yes	Yes	Under Development	
VerAI	VerAI	Proprietary	No	Linux, Web-based	C++,Python, Go, Angular	Graphical user interface, cli	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	

Figura: Comparação das características dos principais frameworks de Deep Learning[1].

[1] https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software

Frameworks

- Caffe[2]
  - Berkeley AI Research (BAIR)
  - Linguagens: Python, C++, Matlab
  - Pouca customização

[2] http://caffe.berkeleyvision.org/

Frameworks

- Torch[3]
  - Ronan Collobert, Koray Kavukcuoglu, Clement Farabet
  - Linguagem: Lua
  - Problemas com o suporte a bibliotecas externas no Lua

[3] http://torch.ch/

Frameworks

- Tensorflow[4]
  - Google Brain team
  - Linguagens: Python, C/C++, Java, Go, R, Julia
  - Grafos Estáticos
  - Comunidade vasta e participativa

[4] https://www.tensorflow.org/

Frameworks

- Pytorch[5]
  - Facebook
  - Linguagem: Python
  - Grafos Dinâmicos
  - Vasta gama de modelos “off-the-shelf”
  - Eficiente, simples e com várias funcionalidades além do treinamento de NNs

[5] https://pytorch.org/

Grafos Dinâmicos

A graph is created on the fly


```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))
```

Figura: Grafos Dinâmicos.

Grafos Dinâmicos

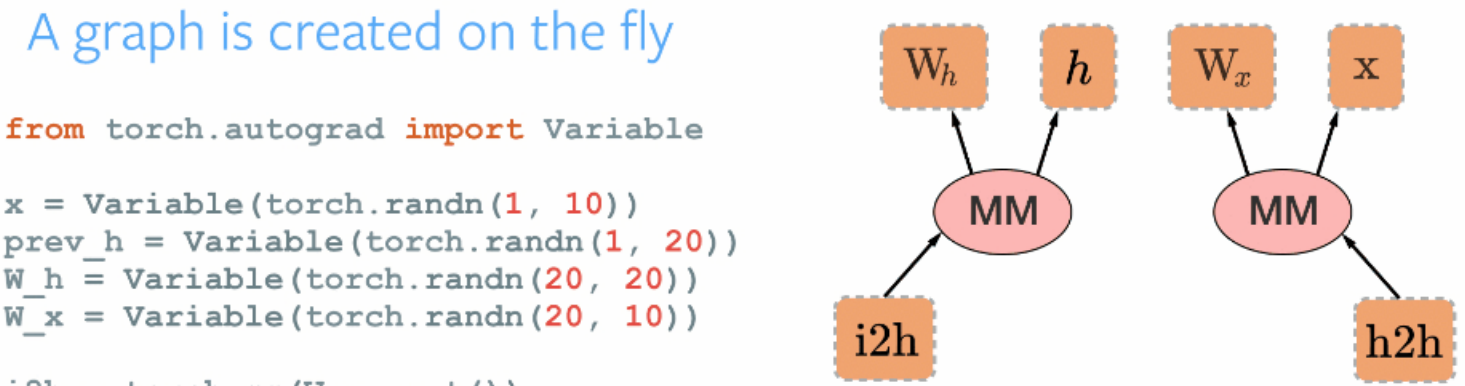


Figura: Grafos Dinâmicos.

Grafos Dinâmicos

A graph is created on the fly

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
```

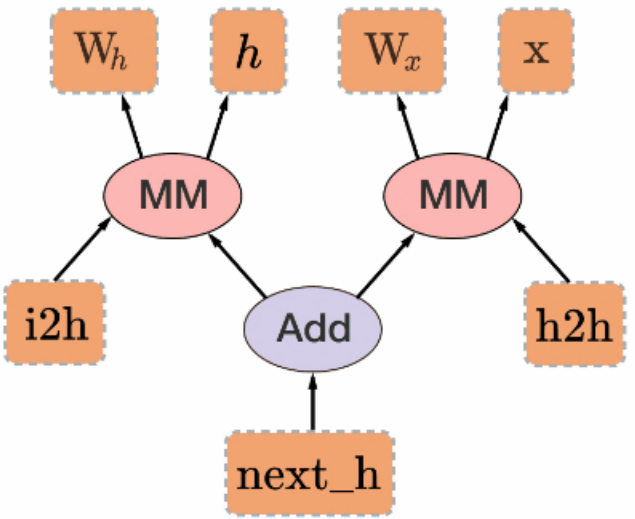


Figura: Grafos Dinâmicos.

Grafos Dinâmicos

A graph is created on the fly

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()
```

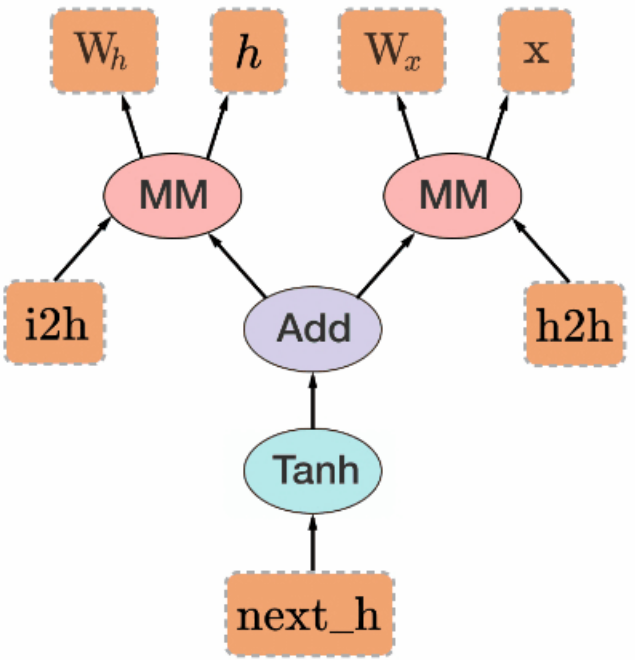


Figura: Grafos Dinâmicos.

Grafos Dinâmicos

Back-propagation
uses the dynamically built graph

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

next_h.backward(torch.ones(1, 20))
```

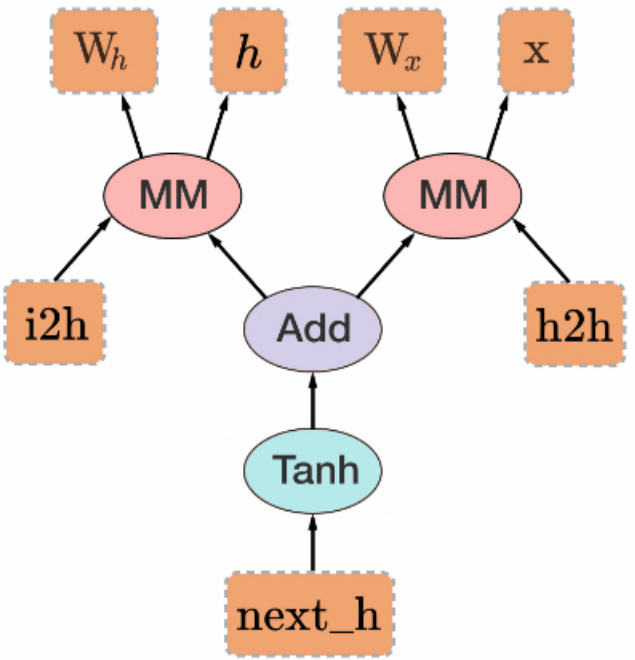


Figura: Grafos Dinâmicos.

Pytorch

- Tensores
- Data Loader
- Definindo um modelo
- Funções de Perda
- Procedimento de treinamento

Tensores

- Sintaxe parecida com ndarrays (Numpy)
- Armazena tanto os inputs e labels quanto os pesos e viéses das NNs
- Funções para tradução de ndarrays para tensores e vice-versa

Tensores

Data type	dtype	CPU tensor
32-bit floating point	`torch.float32` or `torch.float`	`torch.FloatTensor`
64-bit floating point	`torch.float64` or `torch.double`	`torch.DoubleTensor`
16-bit floating point	`torch.float16` or `torch.half`	`torch.HalfTensor`
8-bit integer (unsigned)	`torch.uint8`	`torch.ByteTensor`
8-bit integer (signed)	`torch.int8`	`torch.CharTensor`
16-bit integer (signed)	`torch.int16` or `torch.short`	`torch.ShortTensor`
32-bit integer (signed)	`torch.int32` or `torch.int`	`torch.IntTensor`
64-bit integer (signed)	`torch.int64` or `torch.long`	`torch.LongTensor`

Figura: Tipos de Tensores.

Tensores

Demo - Tensors

Tensors.ipynb

Data Loader

- Classes *Dataset* e *DataLoader*
  - *DataLoader*: normalmente usada a versão original
  - *Dataset*: normalmente customizada
- Subclasses da classe *Dataset* possuem duas funções principais:
  - make_dataset()
  - __getitem__()

Data Loader

Demo - Data Loader

Data_Loader.ipynb

Definindo uma Arquitetura

- O pacote *torch.nn* contém as principais camadas
  - **Convolução/Deconvolução:** Conv1d, Conv2d, Conv3d, ConvTranspose1d, ConvTranspose2d, ConvTranspose3d
  - **Pooling/Unpooling:** MaxPool1d, MaxPool2d, MaxPool3d, MaxUnpool1d, MaxUnpool2d, MaxUnpool3d
  - **Funções de Ativação:** ReLU, Sigmoid, Tanh, Softplus, Softmax, LogSoftmax
  - **Regularização:** Dropout, Dropout2d, Dropout3d
  - **Normalização:** BatchNorm1d, BatchNorm2d, BatchNorm3d, InstanceNorm1d, InstanceNorm2d, InstanceNorm3d
  - **Recorrência:** LSTM, GRU
  - **Densa:** Linear, Bilinear

Dimensões de Entrada e Saída

- Camadas Fully Connected: Linear, Softmax...
  - Input: $B \times * \times F_{in}$
    - B = batch size
    - $*$ = qualquer número de dimensões adicionais
    - F_{in} = features de entrada
  - Output: $B \times * \times F_{out}$
    - F_{out} = features de saída

Dimensões de Entrada e Saída

- Muitas camadas fazem operações em tensores de quaisquer dimensões e retornam um tensor do mesmo tamanho da entrada: ReLU, Dropout, Sigmoid, Tanh...
  - Input: $*$
    - $*$ = qualquer número de dimensões
  - Output: $B \times * \times F_{out}$

Definindo uma Arquitetura

Demo - Architecture

Architecture.ipynb

Funções de Perda

- CrossEntropyLoss – Classificação
- NLLLoss – Classificação
- MSELoss – Regressão/Reconstrução
- L1Loss – Regressão/Reconstrução
- KLDivLoss – Reconstrução
- HingeEmbeddingLoss – Aprendizado Semi-Supervisionado

Funções de Perda

Funções de Perda

Funções de Perda diferentes recebem inputs e targets com dimensões diferentes

Funções de Perda

Demo - Loss Functions

Loss_Functions.ipynb

Otimizadores

- Gradient Descent (GD)
- Stochastic Gradient Descent (SGD)
- Momentum
- AdaGrad (Adaptive Gradient)
- RMSprop
- Adam

Otimizadores

Comparação de Otimizadores

Para mais informações sobre otimizadores, visitar o blog post “An overview of gradient descent optimization algorithms”[6]

[6] http://ruder.io/optimizing-gradient-descent/index.html

Introdução ao Pytorch

Demo - Introdução ao Pytorch

Introduction_to_Pytorch.ipynb

Procedimento de Treinamento

Exercício Prático - Classificação de Faces

MLP_Classification_Olivetti.ipynb

Procedimento de Treinamento

Exercício Prático - Classificação de Dígitos

MLP_Classification_MNIST.ipynb

Procedimento de Treinamento

Exercício Prático - Classificação de Peças de Roupa

MLP_Classification_FashionMNIST.ipyn

[1] Aleksandra C.
How is a neuron adapted to perform its function?, 2018.

[2] Fractal Foundation.
Fractal neurons, 2018.

[3] Herbert Robbins and Sutton Monro.
A stochastic approximation method.
In Herbert Robbins Selected Papers, pages 102–109. Springer, 1985.

[4] Diederik P Kingma and Jimmy Ba.
Adam: A method for stochastic optimization.
arXiv preprint arXiv:1412.6980, 2014.

[5] Geoffrey Hinton.
Overview of mini–batch gradient descent, 2018.

[6] Matthew D Zeiler.
Adadelta: an adaptive learning rate method.
arXiv preprint arXiv:1212.5701, 2012.