Hands-on Deep Learning - Aula 2

Redes Neurais e Redes Convolucionais

Camila Laranjeira[1], Hugo Oliveira[12], Keiller Nogueira[12]

[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

28 de Julho, 2018

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.

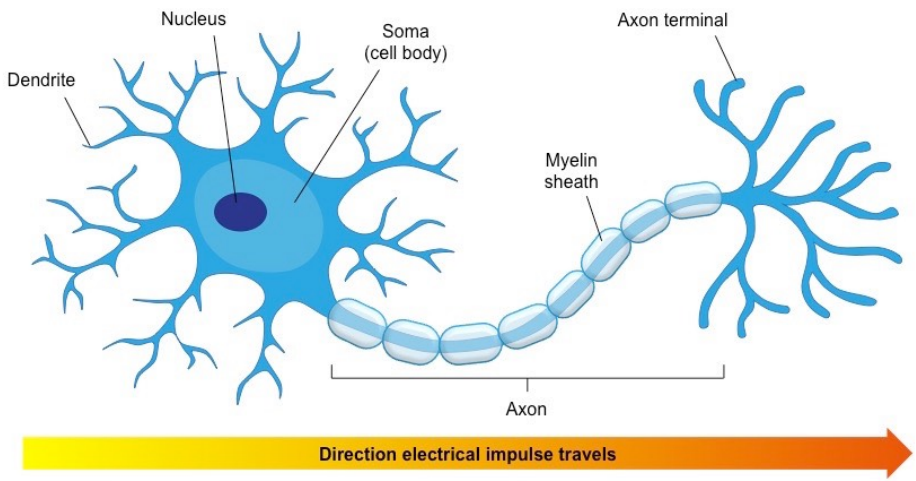


Figura: Representação de um neurônio [1]

O Cérebro

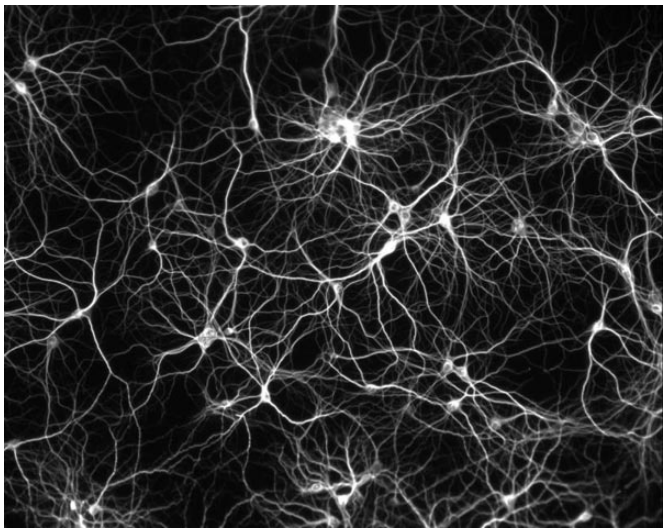


Figura: Representação da malha neural [2]

- 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.

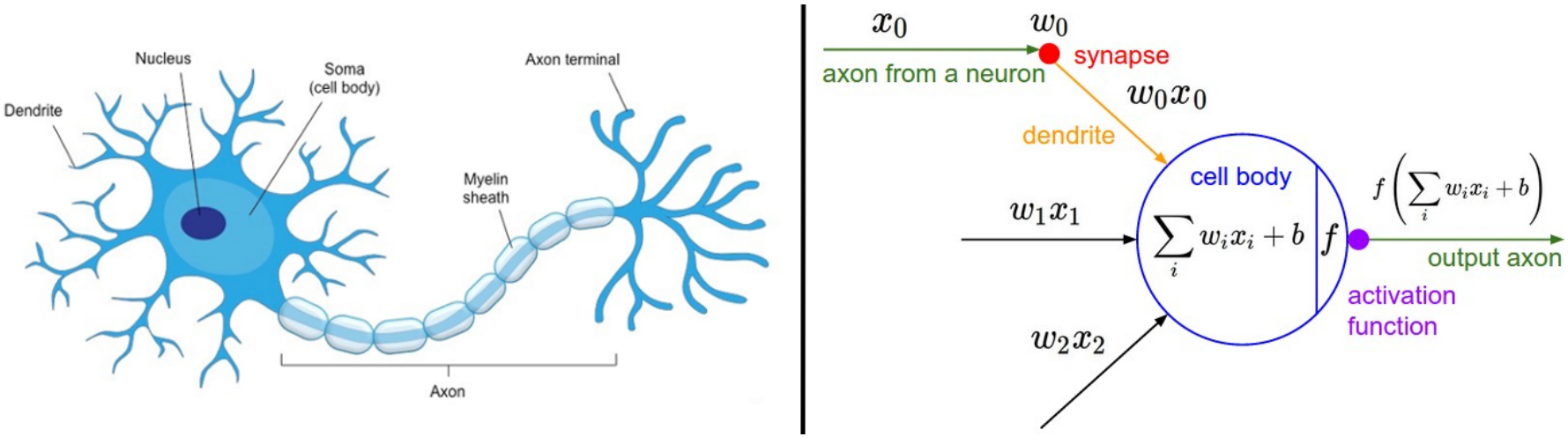
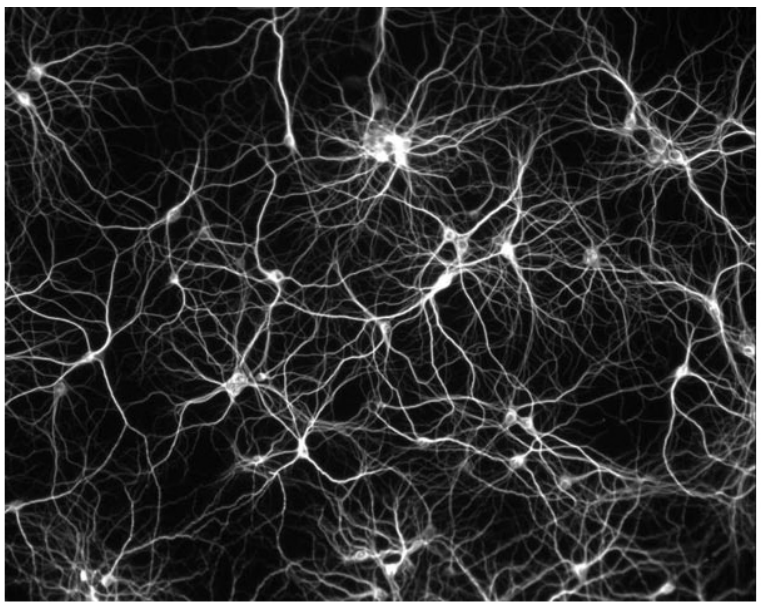
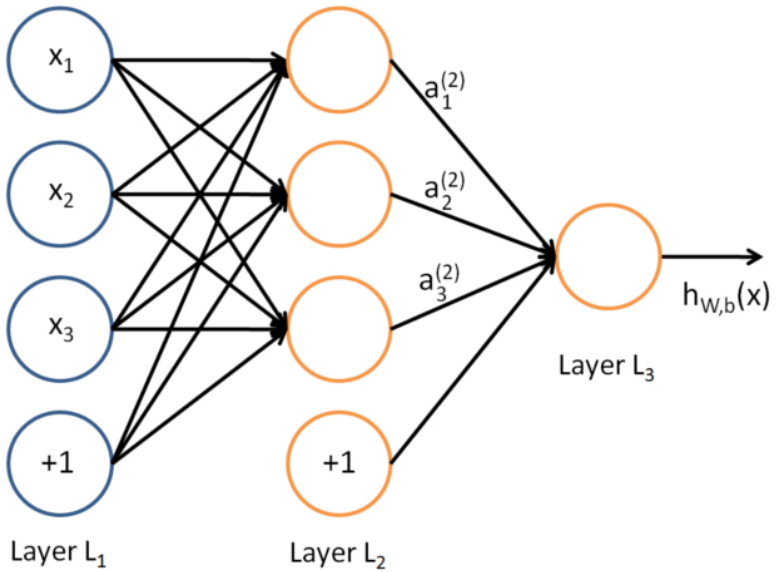


Figura: Comparando os neurônios biológico e artificial (Perceptron).

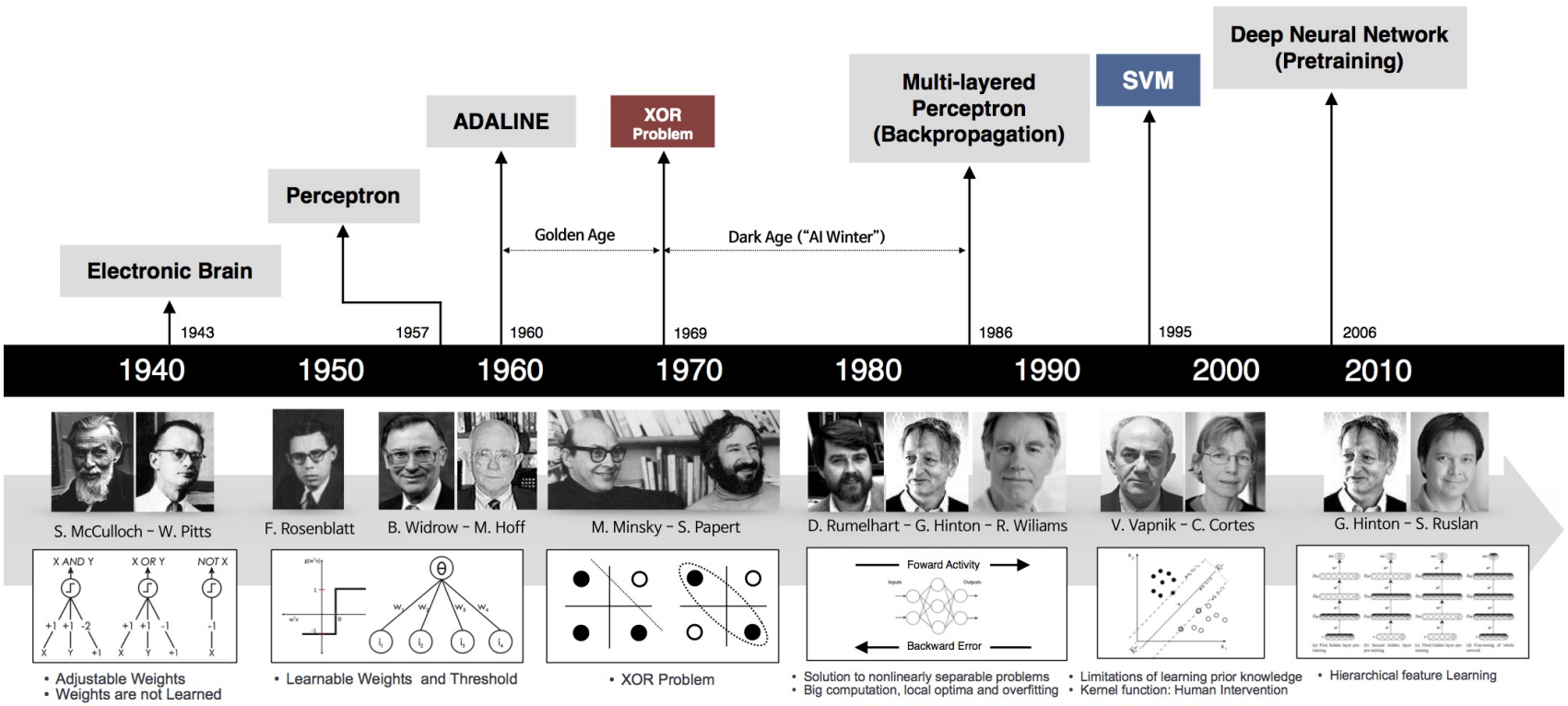
Representação Artificial do Cérebro

- 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.





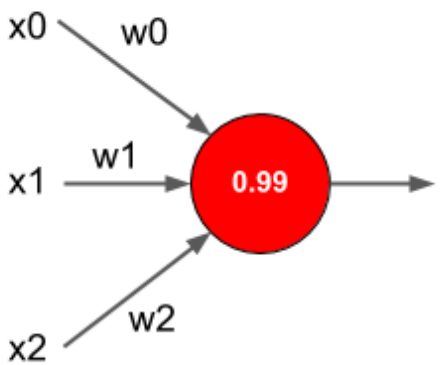
Representação Artificial do Cérebro

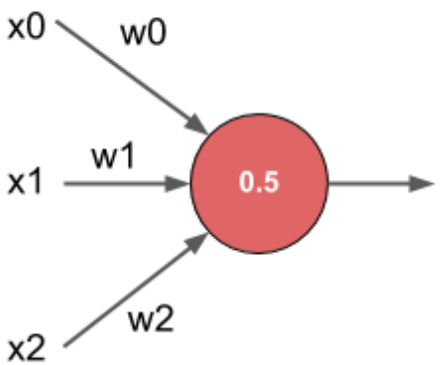


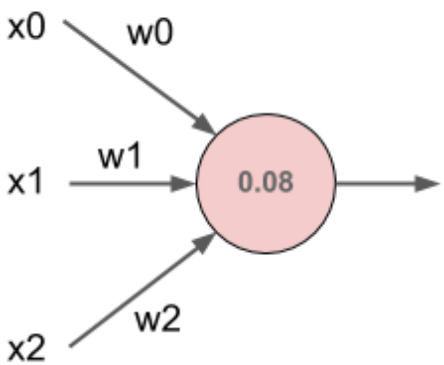
Agenda

Perceptron

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





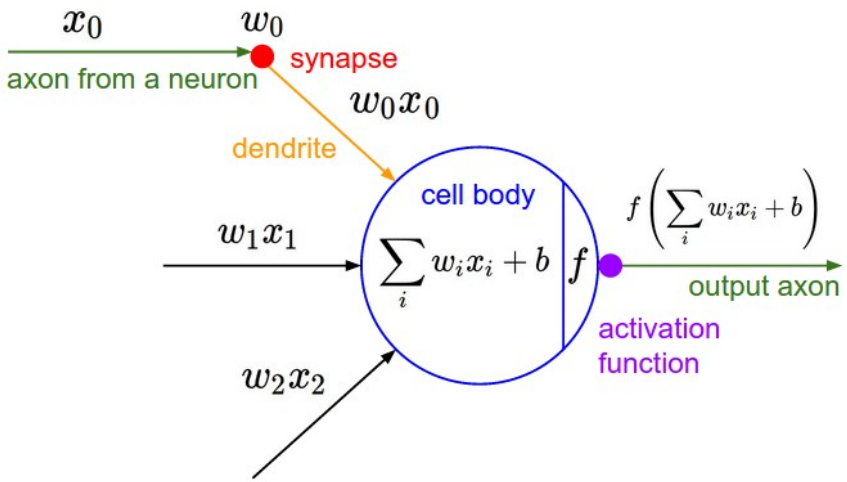


Perceptron

- Função que mapeia uma entrada $x = \{x_0, ..., x_N\}$ para uma saída z
  - Mapeamento linear

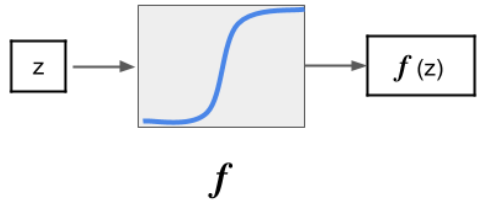
$$z = \sum_{i=0}^{N} w_i x_i + b$$

- Seguido de uma função de ativação f

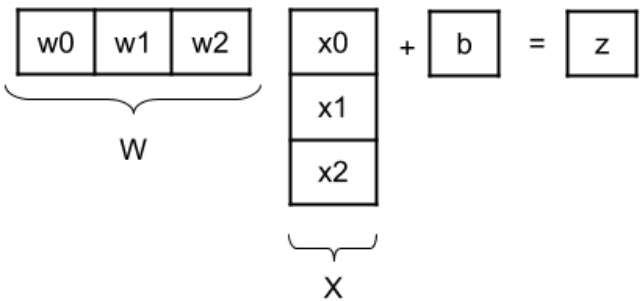


Perceptron

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



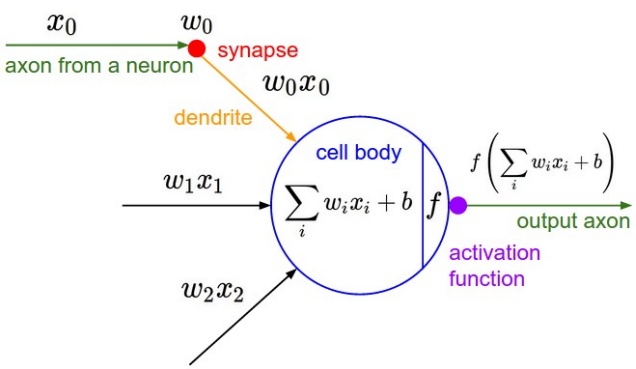
- W é o peso de cada ligação
- b é o bias
- f é a função de ativação
- Por exemplo:

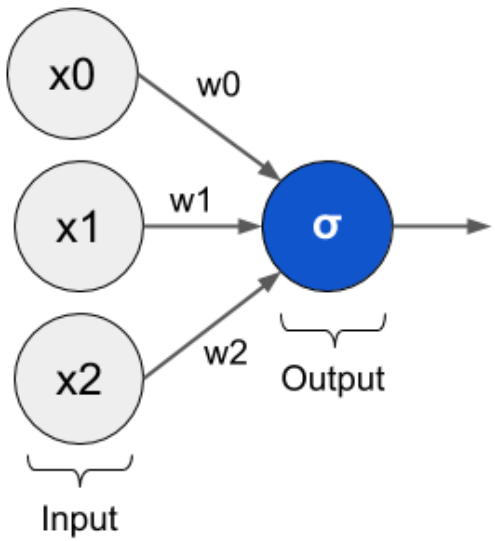


$$f(z) = \begin{cases} 1, if\, z \geqslant 0 \\ 0, otherwise \end{cases}$$

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.





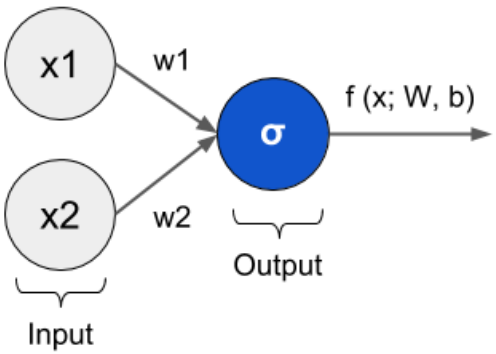
Perceptron como Classificador Linear

- Função de mapeamento do neurônio

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

- Como seria essa função para $x \in \mathbb{R}^2$?

$$z = w_1 x_1 + w_2 x_2 + b$$

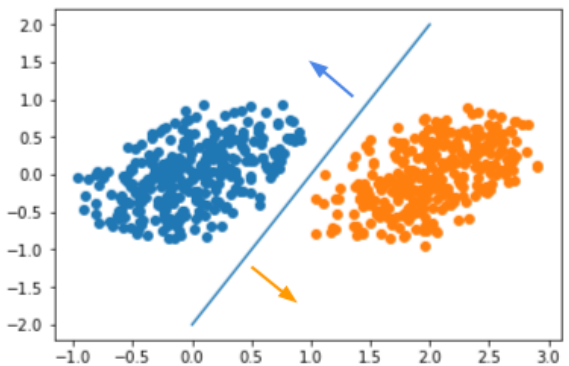


Perceptron como Classificador Linear

- Como seria essa função para $X \in \mathbb{R}^2$?

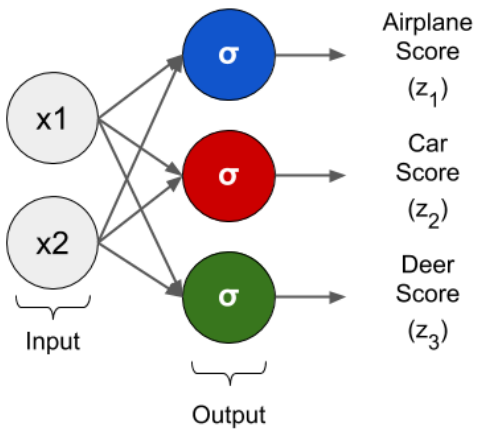
  $z = \mathbf{w_1} x_1 + \mathbf{w_2} x_2 + \mathbf{b}$

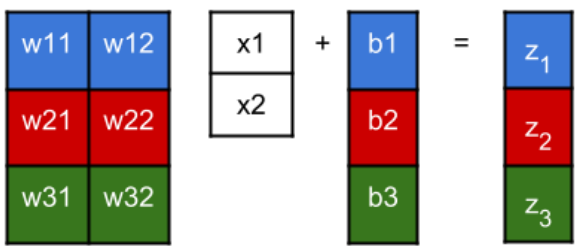
- Treinado da forma certa, um perceptron pode ser um classificador linear para um problema de classificação binária $Y = \{0, 1\}$
- $\mathbf{w_1}$, $\mathbf{w_2}$ e $\mathbf{b}$ sendo os parâmetros da fronteira de decisão
- $$Y \begin{cases} 1, w_1 x_1 + w_2 x_2 + b \geqslant 0 \\ 0, otherwise \end{cases}$$

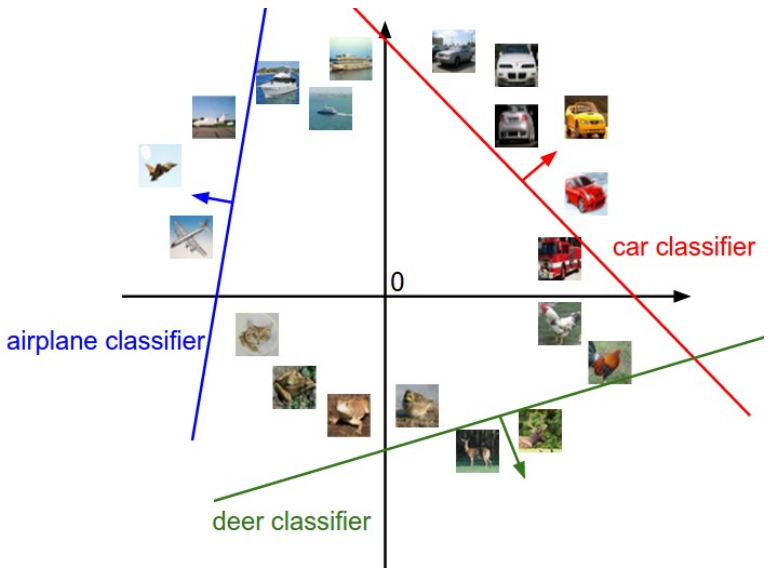


Perceptron como Classificador Linear

- Para problemas com múltiplas classes

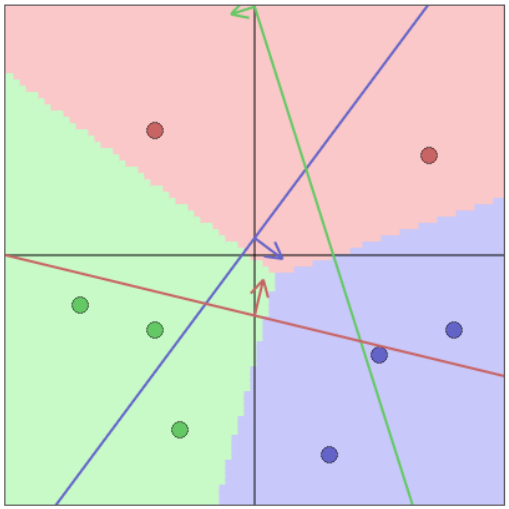






Perceptron como Classificador Linear

Datapoints are shown as circles colored by their class (red/gree/blue). The background regions are colored by whichever class is most likely at any point according to the current weights. Each classifier is visualized by a line that indicates its zero score level set. For example, the blue classifier computes scores as $W_{0,0}x_0 + W_{0,1}x_1 + b_0$ and the blue line shows the set of points (x_0, x_1) that give score of zero. The blue arrow draws the vector $(W_{0,0}, W_{0,1})$, which shows the direction of score increase and its length is proportional to how steep the increase is.
Note: you can drag the datapoints.



Parameters W, b are shown below. The value is in **bold** and its gradient (computed with backprop) is in *red, italic* below. Click the triangles to control the parameters.

W[0,0]	W[0,1]	b[0]
1.12 *0.02*	0.84 *0.02*	0.06 *0.22*
W[1,0]	**W[1,1]**	**b[1]**
-0.95 *0.06*	0.30 *-0.10*	0.30 *-0.11*
W[2,0]	**W[2,1]**	**b[2]**
0.35 *-0.06*	-1.44 *0.06*	0.35 *-0.11*

Step size: 0.10000

Single parameter update

Start repeated update

Stop repeated update

Randomize parameters

Visualization of the data loss computation. Each row is loss due to one datapoint. The first three columns are the 2D data x_i and the label y_i. The next three columns are the three class scores from each classifier $f(x_i; W, b) = Wx_i + b$ (E.g. s[0] = x[0] * W[0,0] + x[1] * W[0,1] + b[0]). The last column is the data loss for a single example, L_i.

x[0]	x[1]	y	s[0]	s[1]	s[2]	L
0.50	0.40	0	0.95	-0.05	-0.05	0.00
0.80	0.30	0	1.21	-0.37	0.20	0.00
0.30	0.80	0	1.06	0.26	-0.70	0.19
-0.40	0.30	1	-0.14	0.77	-0.22	0.09
-0.30	0.70	1	0.31	0.80	-0.76	0.51
-0.70	0.20	1	-0.56	1.03	-0.18	0.00
0.70	-0.40	2	0.51	-0.49	1.17	0.34
0.80	-1.15	2	-0.00	-0.81	2.28	0.00
-0.40	-0.50	2	-0.81	0.53	0.93	0.60
						mean:
						0.19

Total data loss: 0.19
Regularization loss: 0.52
Total loss: 0.71

L2 Regularization strength: 0.10000

Multiclass SVM loss formulation:
- (•) Weston Watkins 1999
- ( ) One vs. All
- ( ) Structured SVM
- ( ) Softmax

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

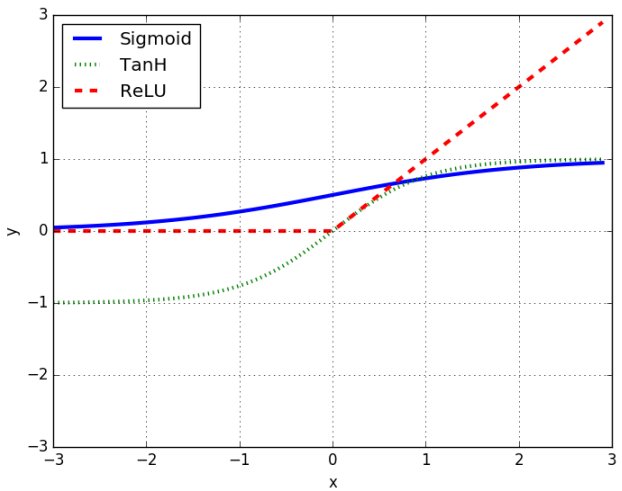
Perceptron

- Já definimos o perceptron como a função

$$f(x; \mathbf{W}, \mathbf{b}) = f(\sum_{i=1}^{N} w_i x_i + b)$$

  - $x = \{x_0, ..., x_N\}$ é um dado fixo (a entrada)
  - f é uma operação matemática fixa (a ativação)
  - $\mathbf{W} = \{w_0, ..., w_N\}$ e $\mathbf{b}$ portanto são os parâmetros a serem otimizados

Ativações



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

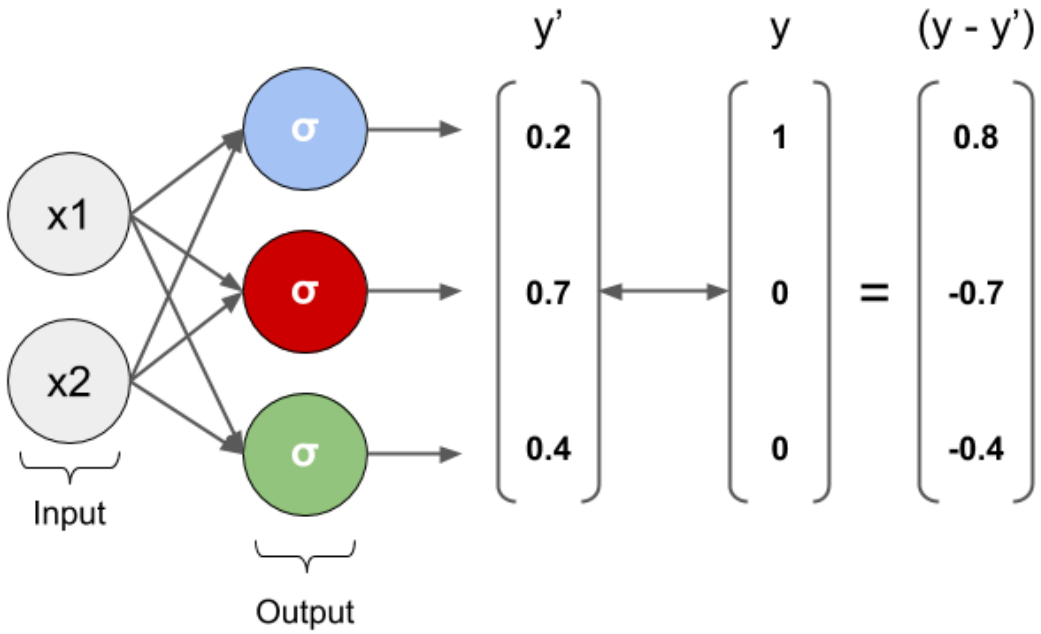
- Entre outras
  - Threshold, Leaky ReLU, Maxout, Linear, etc.
- Em redes com mais de uma camada escondida, a adição de uma função de ativação não linear permite que a rede modele dependências não lineares.

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)

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



Funções de Perda

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

1. Erro quadrático
   - $\mathcal{L}(W, b; x, y) = \frac{1}{2} \|y' - y\|^2$
2. Log-Loss (Cross-Entropy)
   - $\mathcal{L}(W, b; x, y) = -\sum_x y \log y'$
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

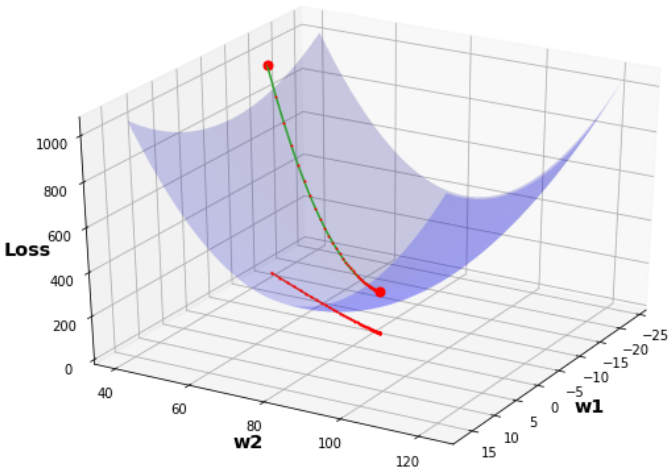
Otimização

Figura: Analogia do processo de otimização. [1]

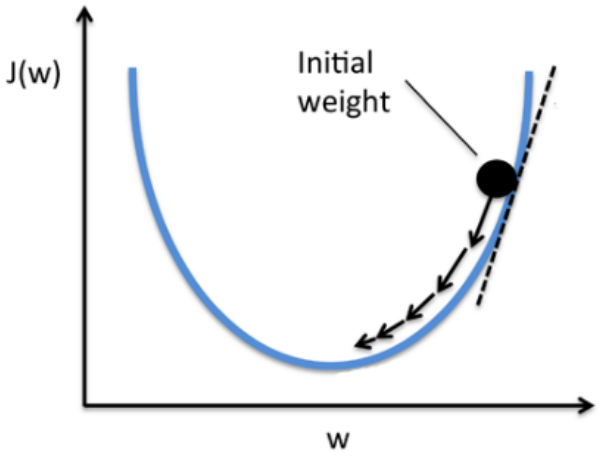
[1] http://cs231n.stanford.edu/slides/2016/winter1516_lecture3.pdf

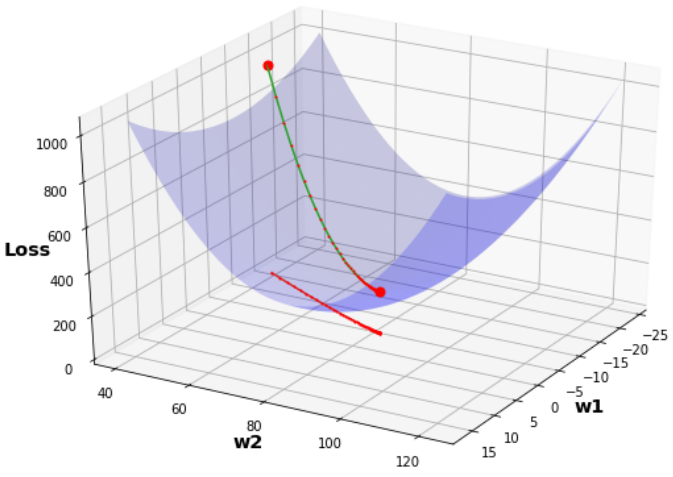
Otimização

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



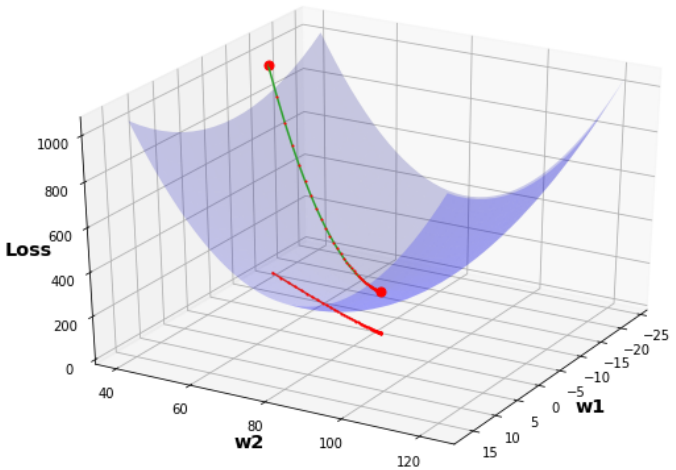
Otimização





- 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



- 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*)

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

Otimização

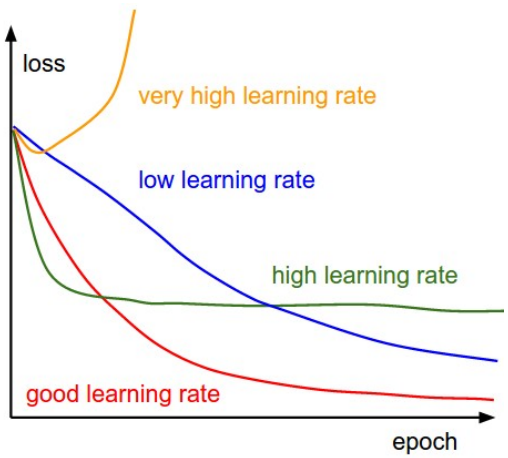
- Descida do gradiente "*vanilla*"

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

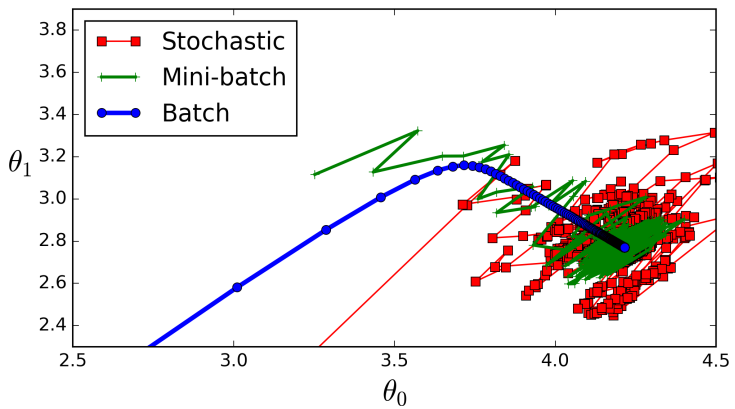
- O gradiente $\nabla_{(W,b)}\mathcal{L}$ vai indicar:
  - O sinal: se o peso deve aumentar ou diminuir
  - A magnitude: o quanto aquele peso deve ser ajustado.

Otimização

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



- Batch size
  - Ao longo de todo o treino
  - por batch
  - por amostra



Otimização

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

Otimização

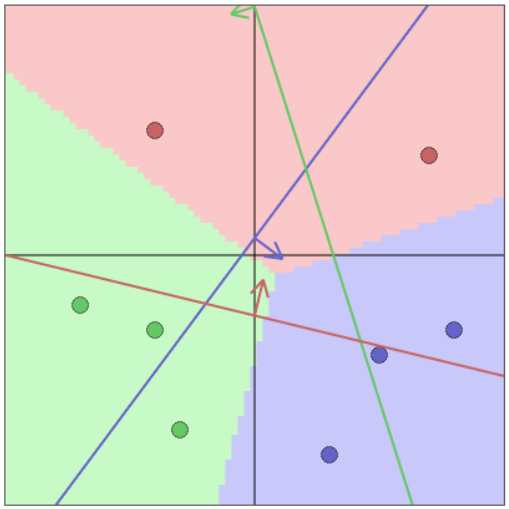
Gif - Algoritmos de Minimização

gifs/optimization-2D.gif

gifs/optimization-3D.gif

Perceptron como Classificador Linear

Datapoints are shown as circles colored by their class (red/gree/blue). The background regions are colored by whichever class is most likely at any point according to the current weights. Each classifier is visualized by a line that indicates its zero score level set. For example, the blue classifier computes scores as $W_{0,0}x_0 + W_{0,1}x_1 + b_0$ and the blue line shows the set of points (x_0, x_1) that give score of zero. The blue arrow draws the vector $(W_{0,0}, W_{0,1})$, which shows the direction of score increase and its length is proportional to how steep the increase is.
Note: you can drag the datapoints.



Parameters W, b are shown below. The value is in **bold** and its gradient (computed with backprop) is in *red, italic* below. Click the triangles to control the parameters.

W[0,0]	W[0,1]	b[0]
1.12 *0.02*	0.84 *0.02*	0.06 *0.22*
W[1,0]	**W[1,1]**	**b[1]**
-0.95 *0.06*	0.30 *-0.10*	0.30 *-0.11*
W[2,0]	**W[2,1]**	**b[2]**
0.35 *-0.06*	-1.44 *0.06*	0.35 *-0.11*

Step size: 0.10000

Single parameter update

Start repeated update

Stop repeated update

Randomize parameters

Visualization of the data loss computation. Each row is loss due to one datapoint. The first three columns are the 2D data x_i and the label y_i. The next three columns are the three class scores from each classifier $f(x_i; W, b) = Wx_i + b$ (E.g. s[0] = x[0] * W[0,0] + x[1] * W[0,1] + b[0]). The last column is the data loss for a single example, L_i.

x[0]	x[1]	y	s[0]	s[1]	s[2]	L
0.50	0.40	0	0.95	-0.05	-0.05	0.00
0.80	0.30	0	1.21	-0.37	0.20	0.00
0.30	0.80	0	1.06	0.26	-0.70	0.19
-0.40	0.30	1	-0.14	0.77	-0.22	0.09
-0.30	0.70	1	0.31	0.80	-0.76	0.51
-0.70	0.20	1	-0.56	1.03	-0.18	0.00
0.70	-0.40	2	0.51	-0.49	1.17	0.34
0.80	-1.15	2	-0.00	-0.81	2.28	0.00
-0.40	-0.50	2	-0.81	0.53	0.93	0.60
						mean:
						0.19

Total data loss: 0.19
Regularization loss: 0.52
Total loss: 0.71

L2 Regularization strength: 0.10000

Multiclass SVM loss formulation:
- (•) Weston Watkins 1999
- ( ) One vs. All
- ( ) Structured SVM
- ( ) Softmax

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

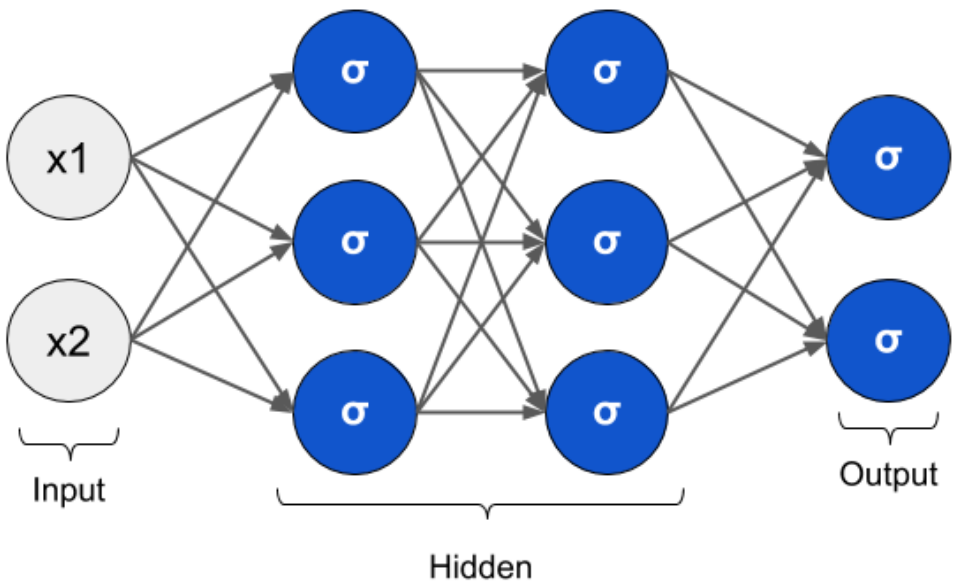
Perceptron como Classificador Linear

Demo - Classificação Linear

linear_demo.ipynb

Multi-Layer Perceptron

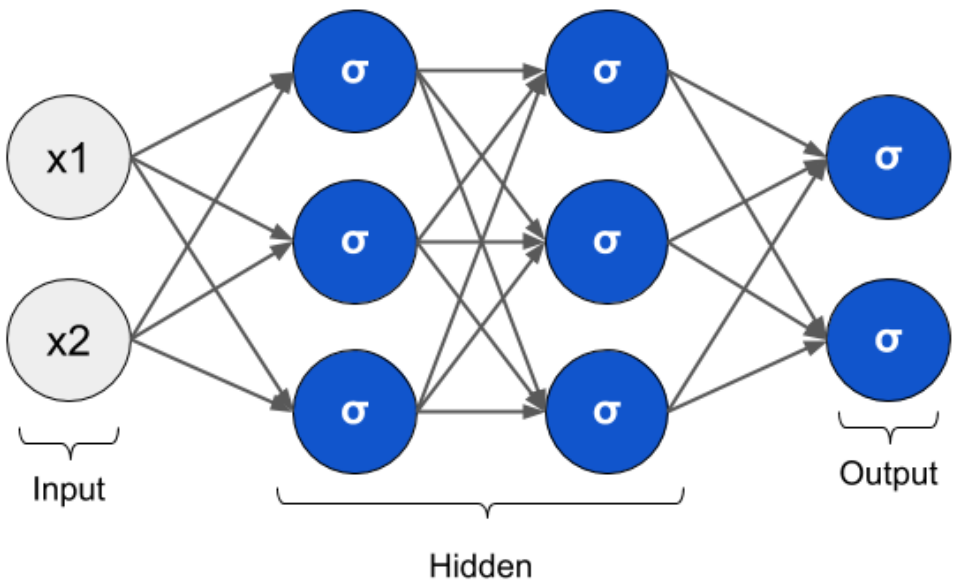
- Além das camadas de entrada e saída, temos também as camadas escondidas (*hidden layers*).



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?

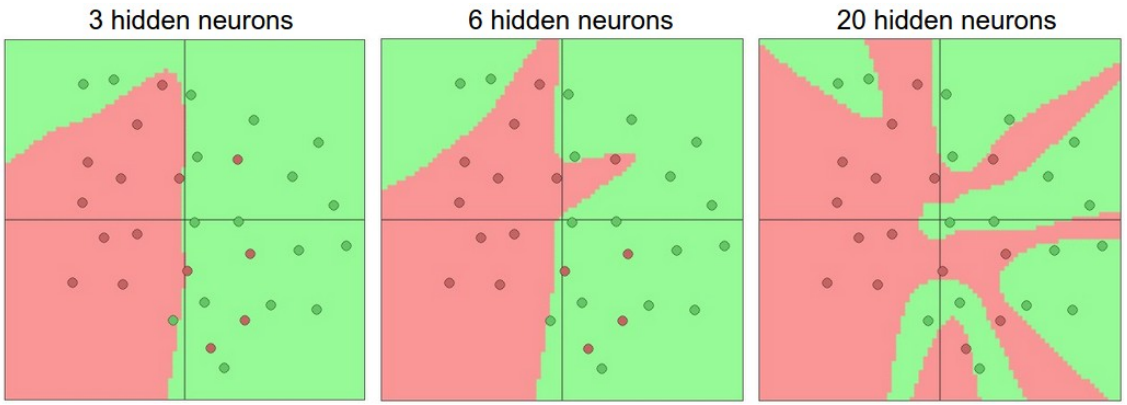


Figura: ConvnetJS demo [3]

[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**.

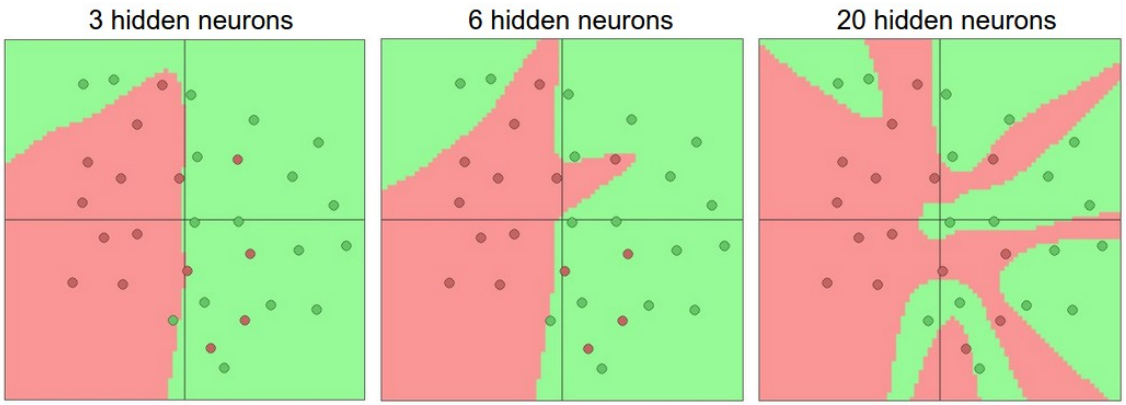


Figura: ConvnetJS demo [3]

[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

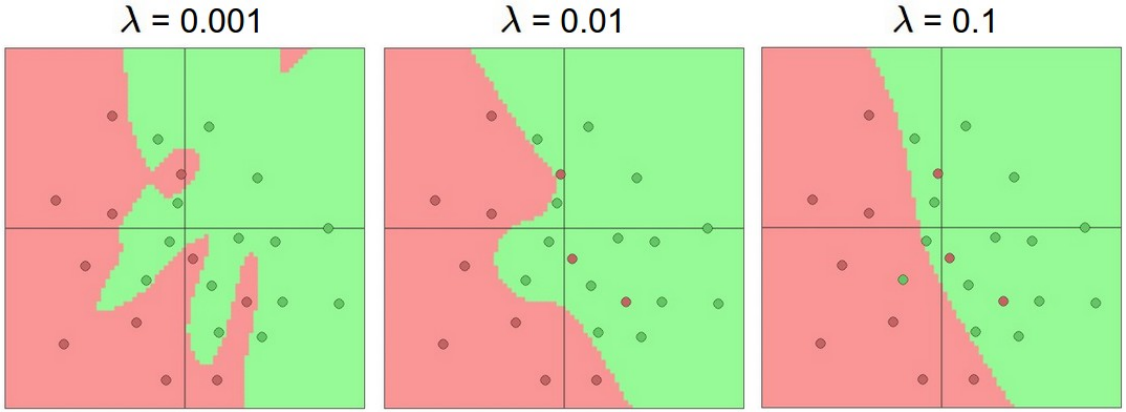


Figura: ConvnetJS demo [3]

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

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.*"
- Por que então encontramos redes com dezenas de camadas?

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

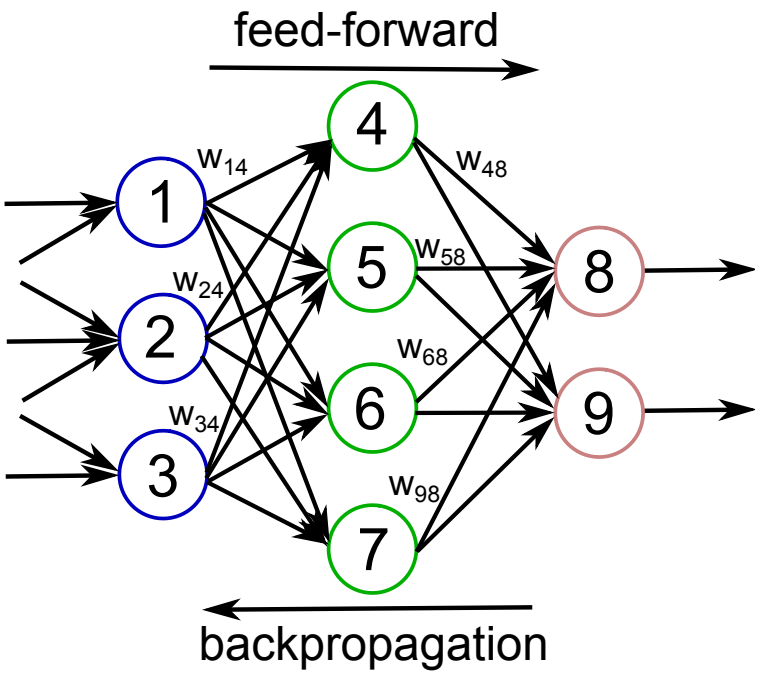
Multi-Layer Perceptron

Demo - Classificação Linear

nonlinear_demo.ipynb

Redes Neurais Artificiais

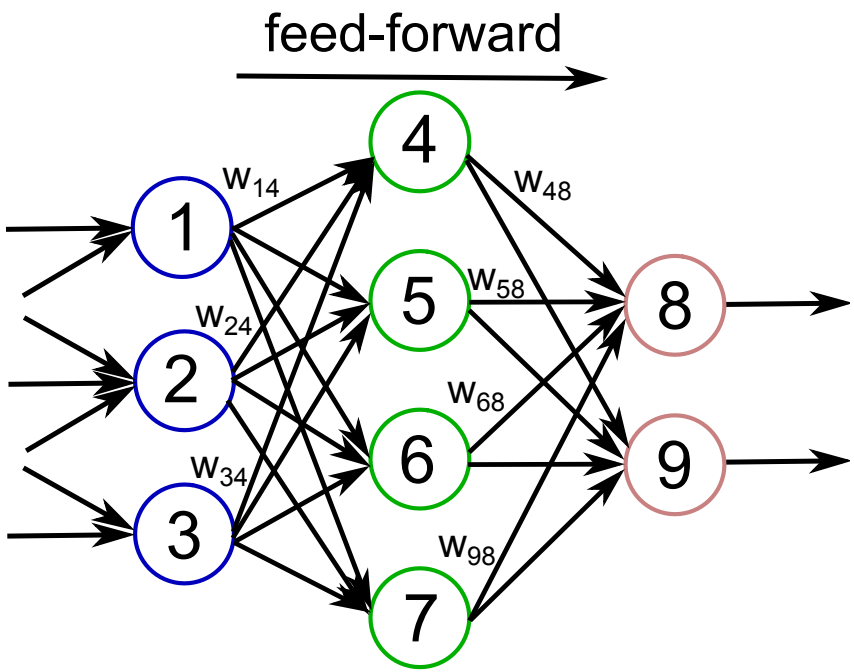
- O processo de aprendizado pode ser dividido em dois passos:
  - Feed-forward: recebe os dados e gera a saída final da rede
  - Backpropagation: ajusta os pesos de acordo com o gradiente



Feed-forward

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

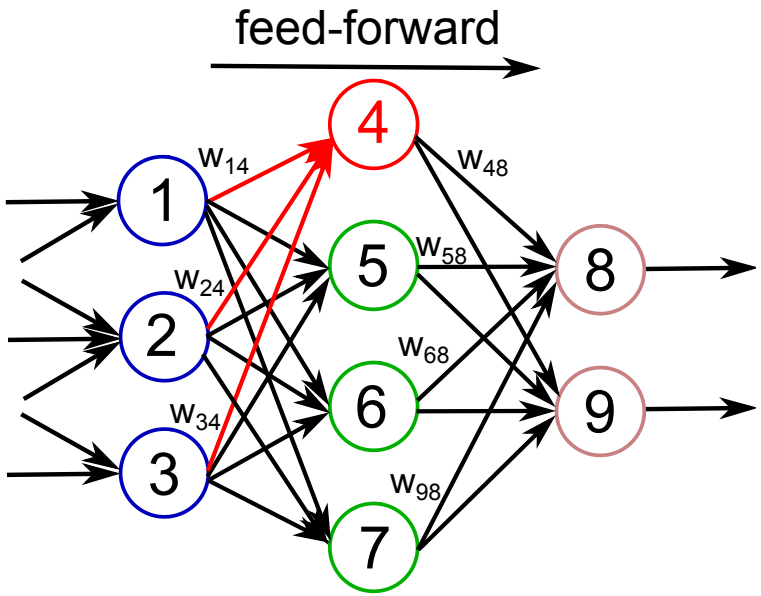
$$a = f(z)$$



Feed-forward

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

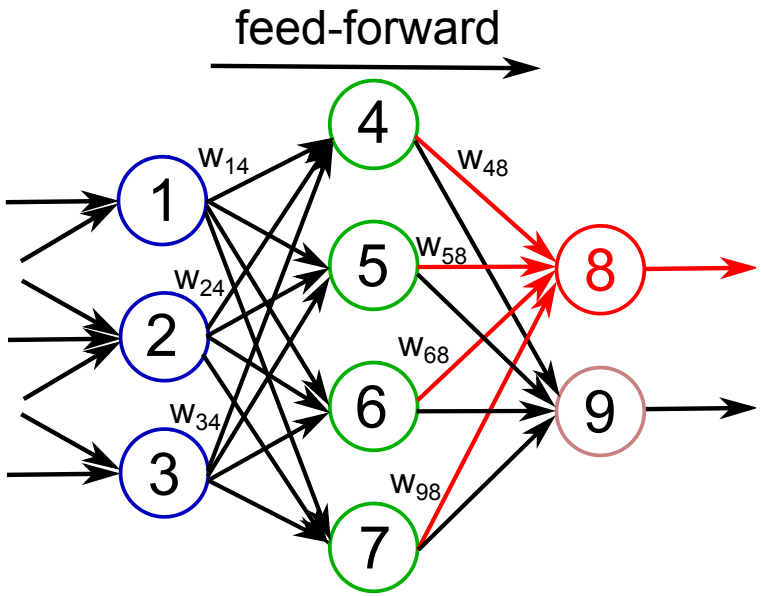
$$a^4 = f(z^4)$$



Feed-forward

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

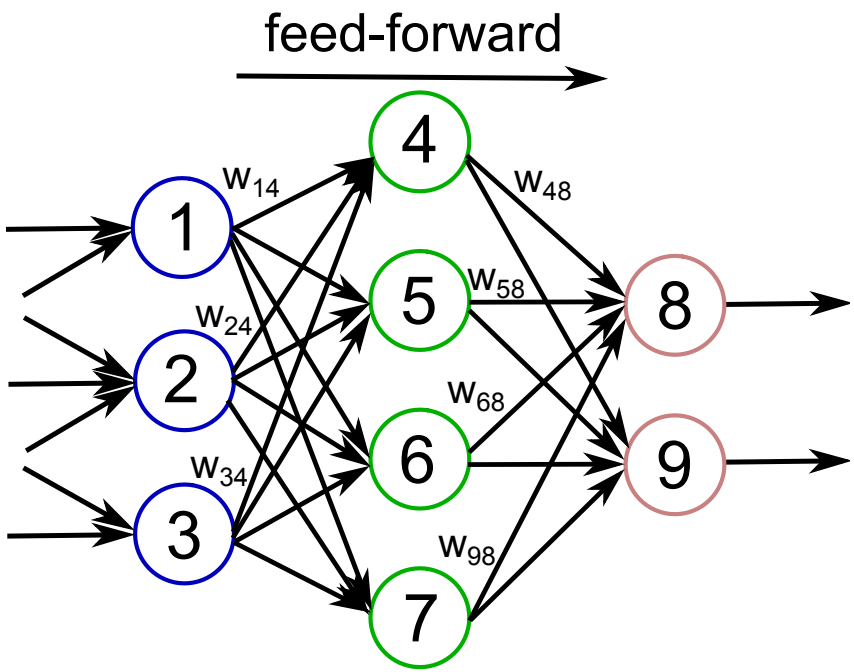
$$a^8 = f(z^8)$$



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

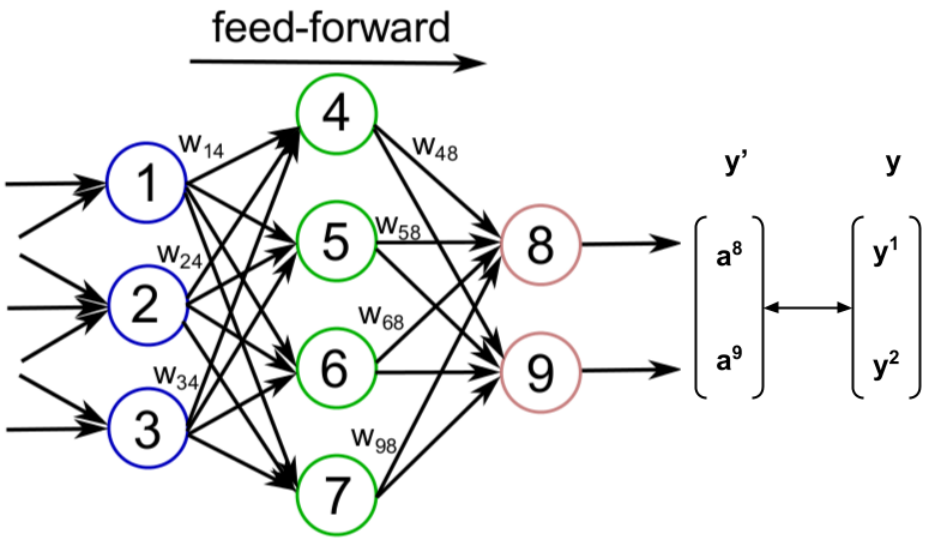
- Suponhamos:
  - Erro quadrático: $\mathcal{L}(W, b; x, y) = \frac{1}{2}\|y - a^{(L)}\|^2$
  - Sigmóide como função de ativação: $f(z) = \sigma(z) = \frac{1}{1+\exp^z}$

Backpropagation

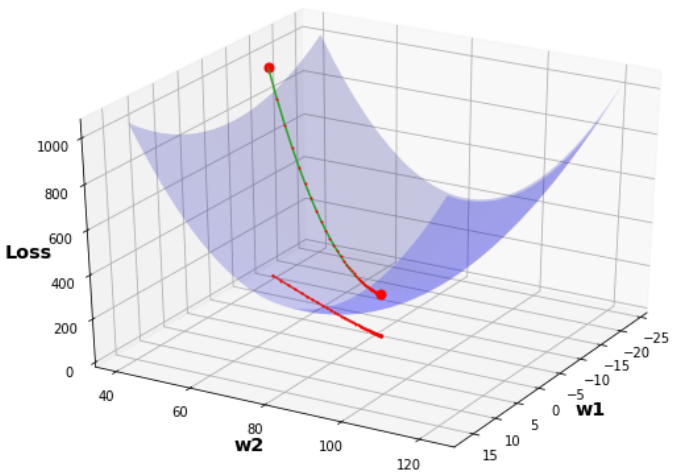
- Passo a passo, temos:

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]

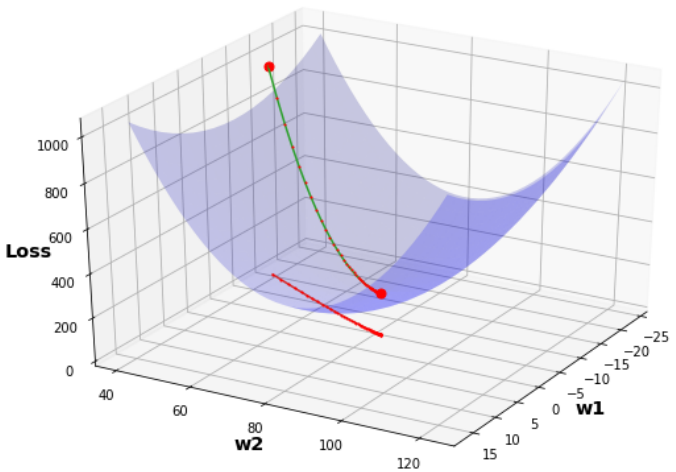
- $\frac{\partial \mathcal{L}}{\partial w^{(L)}}$
- $\mathcal{L} = \frac{1}{2}\|y - a^{(L)}\|^2$
- $a^{(L)} = \sigma(z^{(L)})$
- $z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$

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

Backpropagation

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

- Regra da cadeia

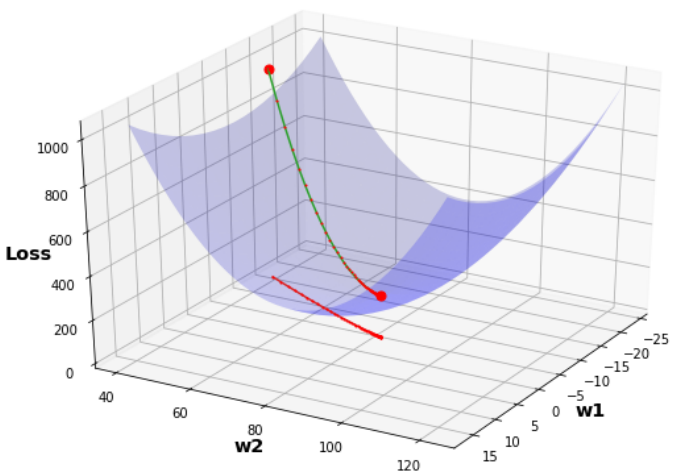


- $\frac{\partial \mathcal{L}}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial \mathcal{L}}{\partial a^{(L)}}$
- $\mathcal{L} = \frac{1}{2} \| y - a^{(L)} \|^2$
- $a^{(L)} = \sigma(z^{(L)})$
- $z^{(L)} = W^{(L)} a^{(L-1)} + b^{(L)}$

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

Backpropagation

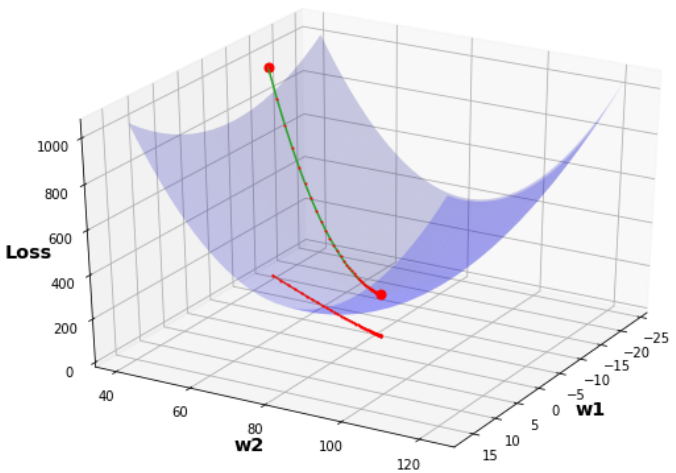
2 Calcula o erro dos neurônios da última camada [4]



- $\frac{\partial \mathcal{L}}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial \mathcal{L}}{\partial a^{(L)}}$
- $\frac{\partial \mathcal{L}}{\partial a^{(L)}} = (y - a^{(L)})$
- $\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)})$
- $\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$

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

Backpropagation



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

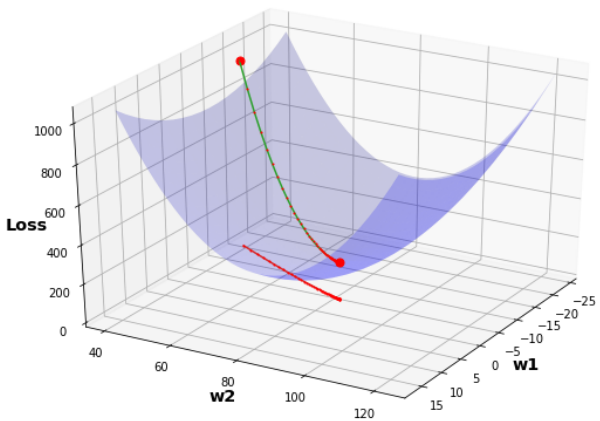
- $\frac{\partial \mathcal{L}}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial \mathcal{L}}{\partial a^{(L)}}$
- $\frac{\partial \mathcal{L}}{\partial w^{(L)}} = a^{(L-1)} \sigma'(z^{(L)})(y - a^{(L)})$

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

Backpropagation

- Sigmóide como função de ativação: $f(z) = \sigma(z) = \frac{1}{1+\exp^{z}}$

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

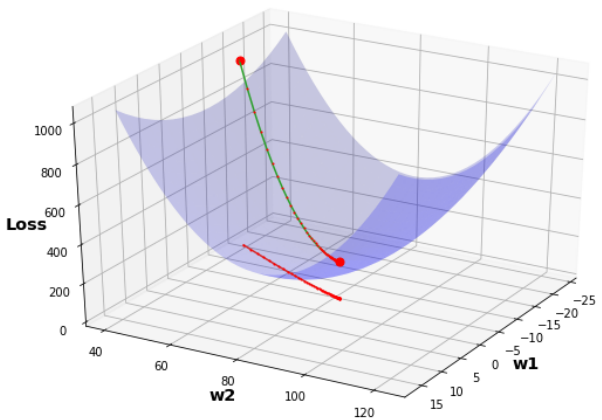


- $\sigma'(z^{(L)}) = \sigma(z^{(L)})(1 - \sigma(z^{(L)}))$
- $\sigma'(z^{(L)}) = a^{(L)}(1 - a^{(L)})$
- $\frac{\partial \mathcal{L}}{\partial W^{(L)}} = a^{(L-1)}\sigma'(z^{(L)})(y - a^{(L)})$
- $\frac{\partial \mathcal{L}}{\partial W^{(L)}} = a^{(L-1)}a^{(L)}(1 - a^{(L)})(y - a^{(L)})$

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

Backpropagation

2 Calcula o erro dos neurônios da última camada

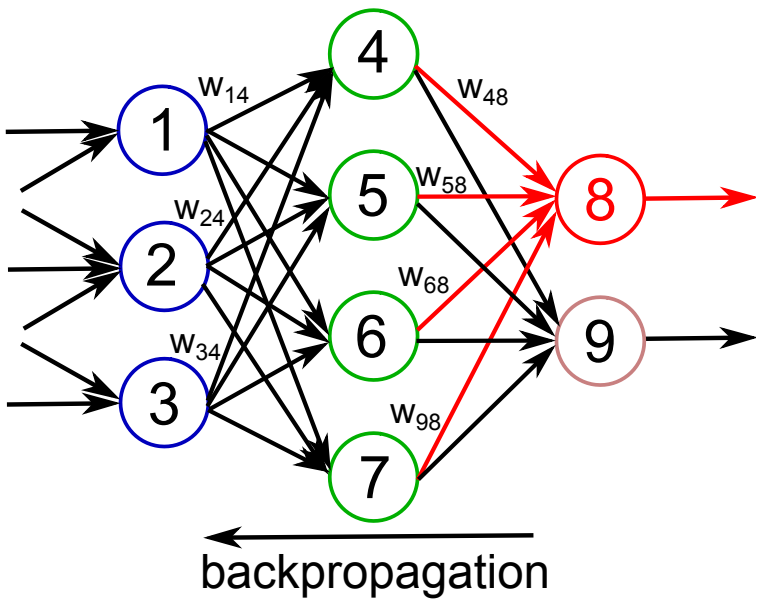


- $\frac{\partial \mathcal{L}}{\partial W^{(L)}} = a^{(L-1)} a^{(L)} (1 - a^{(L)})(y - a^{(L)})$
- $\delta^{(L)} = a^{(L)} (1 - a^{(L)})(y - a^{(L)})$
- $\frac{\partial \mathcal{L}}{\partial W^{(L)}} = a^{(L-1)} \delta^{(L)}$

Backpropagation

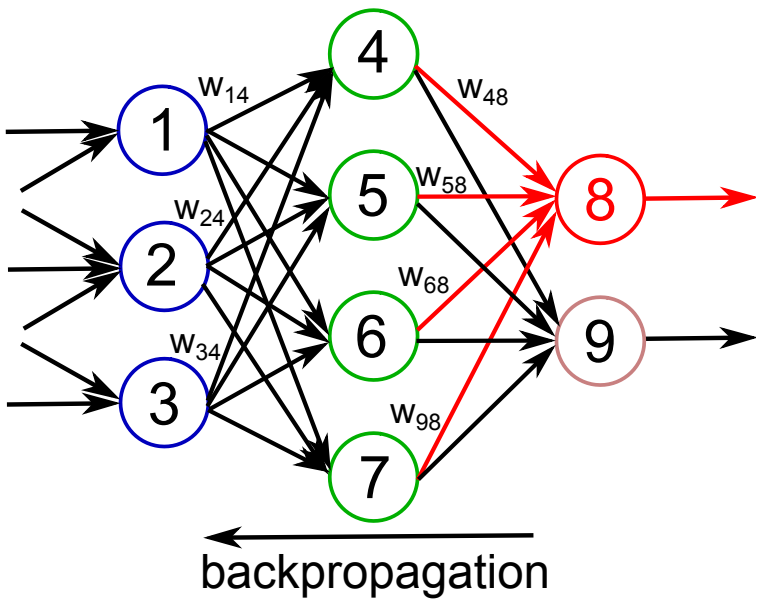
- Propagando o erro para neurônios na última camada

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



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:

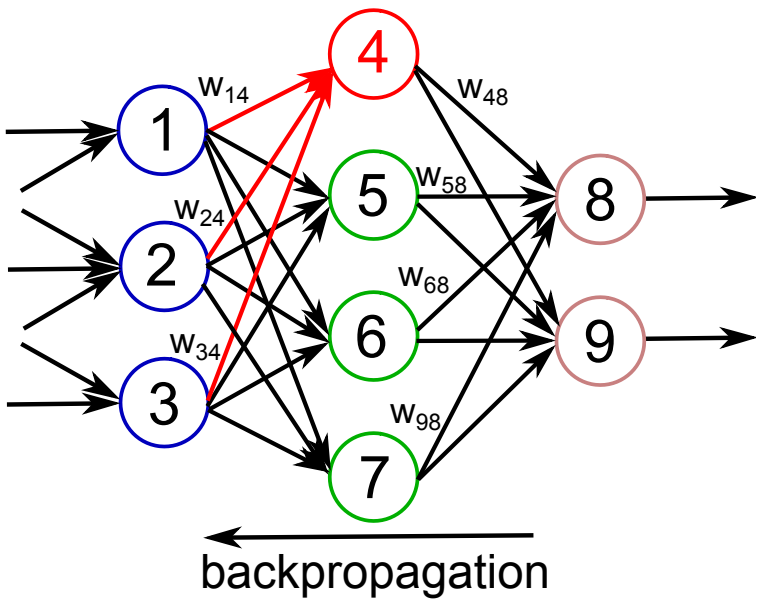
$$\delta^{(L-1)} = (\sum_i \delta_i^{(L)} W_i^{(L)}) f'(z^{(L-1)})$$

Backpropagation

- Propagando para um neurônio de camada escondida

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

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

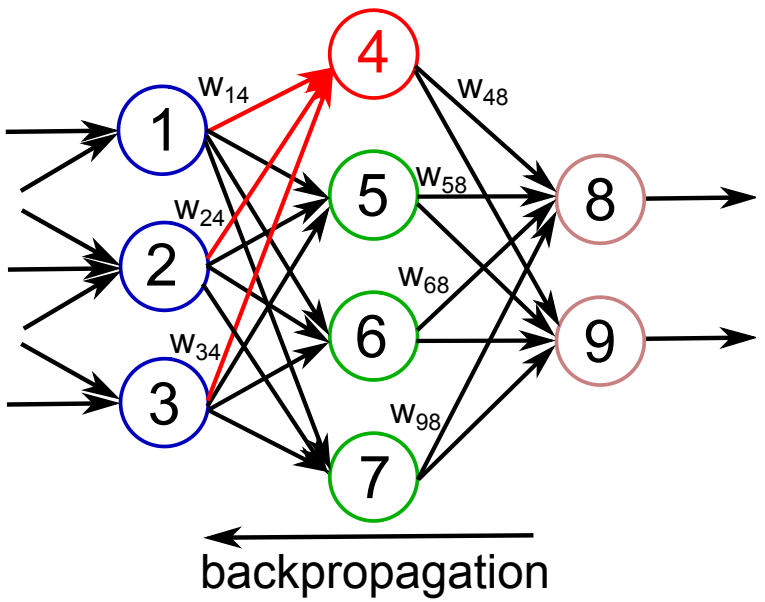


Backpropagation

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

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

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



Em resumo

- Para definir uma rede neural, precisamos de:
  - Arquitetura - número de camadas, número de neurônios em cada camada, a função de ativação
  - Função de Custo
  - Algoritmo de otimização
- Tudo isso é definido experimentalmente (conhecimento prévio pode ajudar bastante) e é diretamente ligado ao problema em questão

Redes Neurais

Atividade - Hiperparâmetros

hyperparameters.ipynb

Agenda

Frameworks

Software	Creator	Software license[k]	Open source	Platform	Written in	Interface	OpenMP support	OpenCL support	CUDA support	Parallel execution (multi node)	Automatic differentiation[l]	Has pretrained models	Recurrent nets	Convolutional nets	RBM/DBNs	Metal support
roNNie.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[c]	Yes	Yes	Yes[c]	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[f][g]	Computational Graph	Yes[f]	Yes	Yes	Yes	Yes[f]	
Chainer	Preferred Networks	MIT license	Yes	Linux, macOS, Windows		Python	No	No[h][i]	Yes	Yes	Yes	Yes	Yes			
Darknet	Joseph Redmon	Public Domain	Yes	Cross-Platform	C	C, Python	Yes	No[b]	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[33]	Yes	Yes	Yes					
Intel Data Analytics Acceleration Library	Intel	Apache License 2.0	Yes	Linux, macOS, Windows on Intel CPU[84]	C++, Python, Java	C++, Python, Java[84]	Yes	No	No	Yes	No		Yes		Yes	
Intel Math Kernel Library	Intel	Proprietary	No	Linux, macOS, Windows on Intel CPU[85]		C[86]	Yes[87]	No	No	Yes	No	Yes[88]	Yes[88]		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[89]	Yes	Yes	Yes	Yes[89]	
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

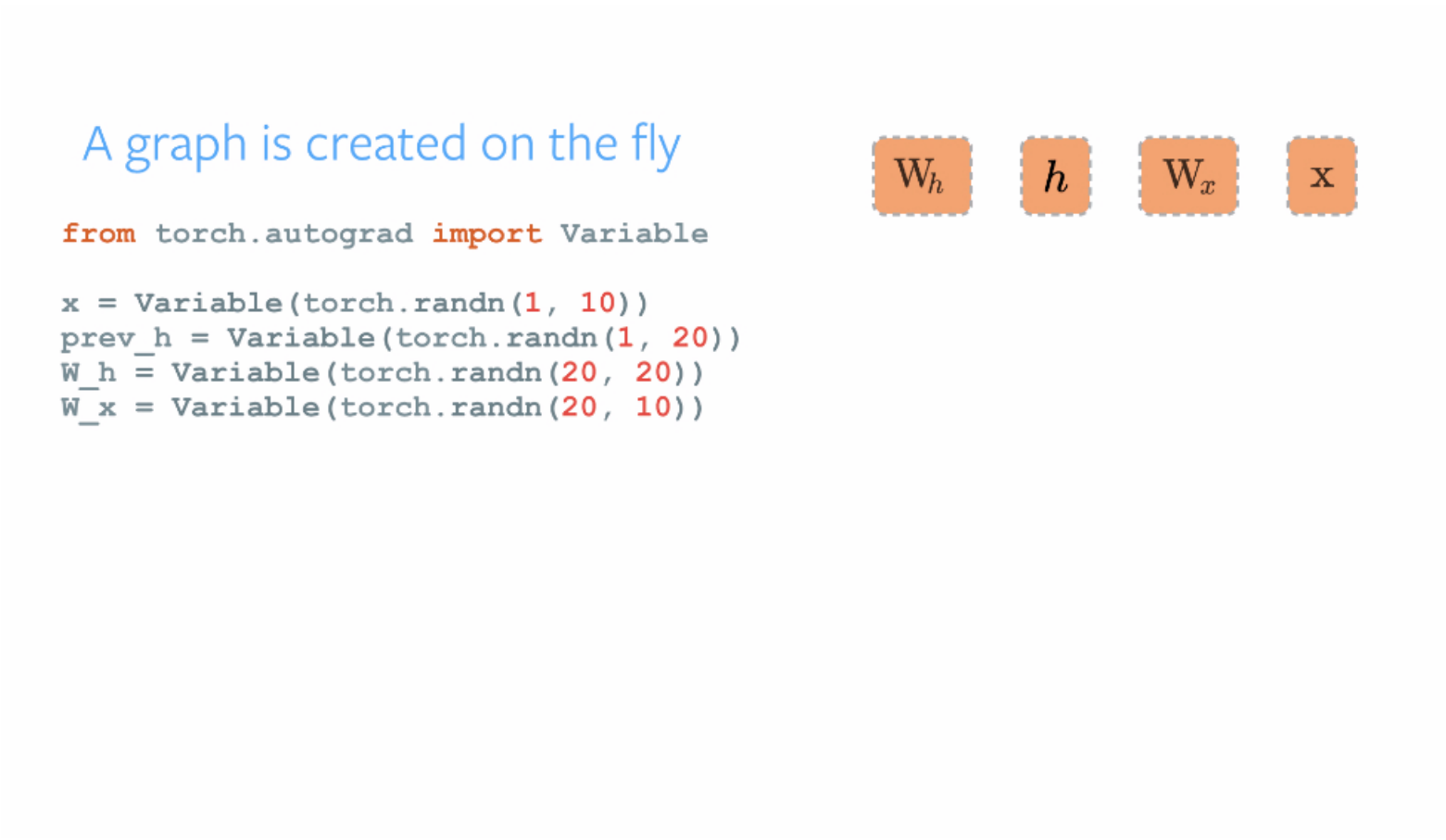


Figura: Grafos Dinâmicos.

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())
```

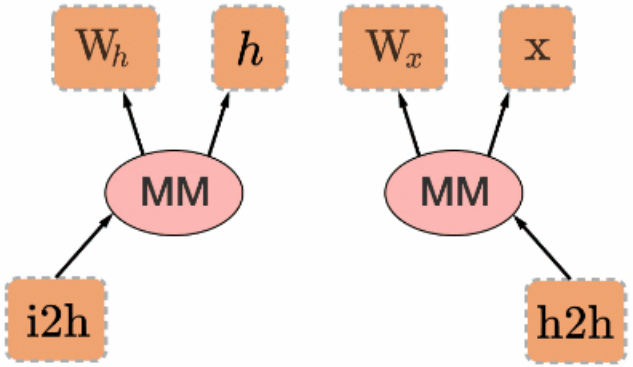


Figura: Grafos Dinâmicos.

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
```

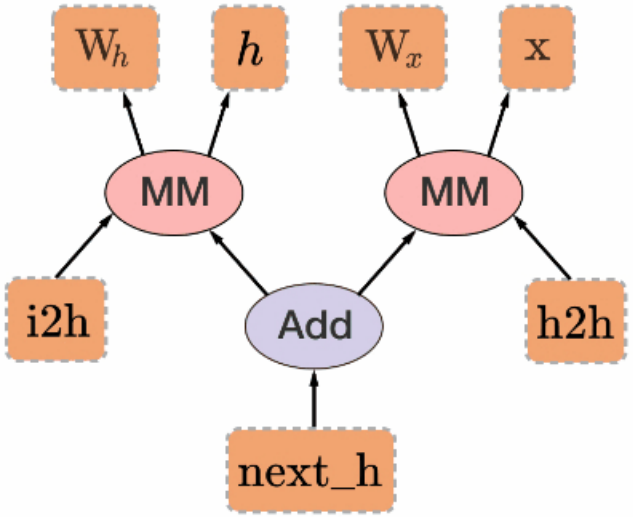


Figura: Grafos Dinâmicos.

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()
```

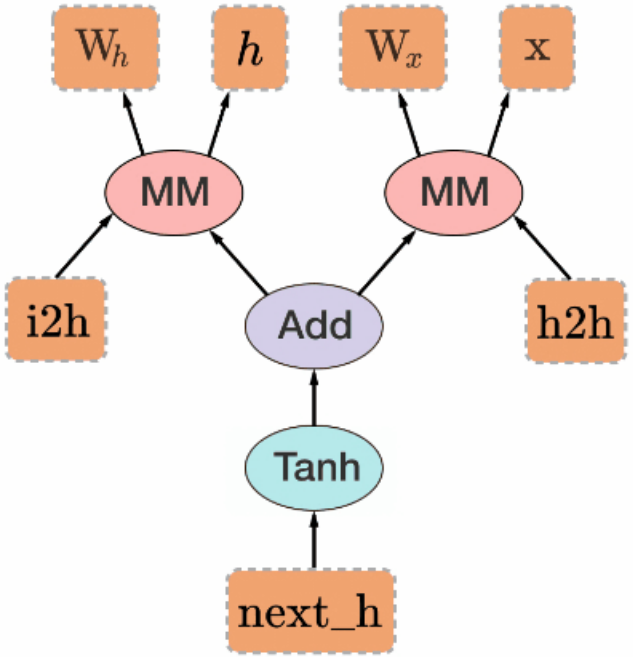


Figura: Grafos Dinâmicos.

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))
```

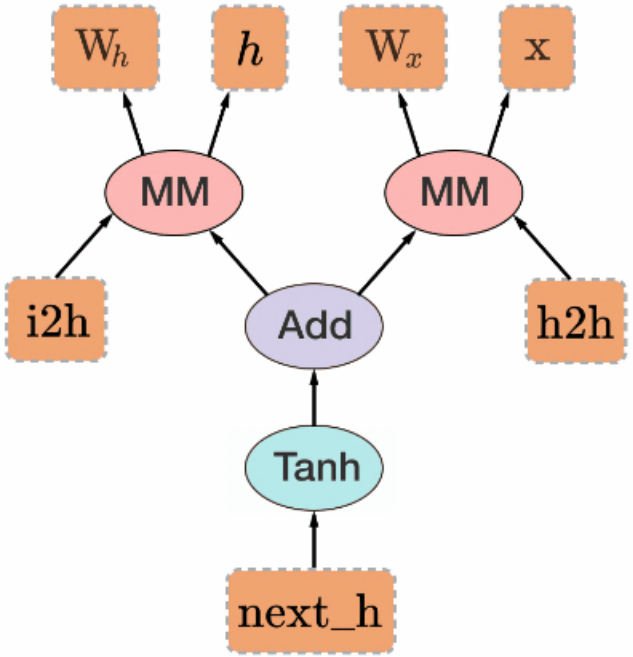


Figura: Grafos Dinâmicos.

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 para imagens: Convoluções, Pooling, Batch Normalization 2D...
  - Input: $B \times C_{in} \times H_{in} \times W_{in}$
    - B = batch size
    - C_{in} = número de canais da imagem de entrada
    - H_{in} = height da imagem de entrada
    - W_{in} = width da imagem de entrada
  - Output: $B \times C_{out} \times H_{out} \times W_{out}$
    - C_{out} = número de canais da imagem de saída
    - H_{out} = height da imagem de saída
    - W_{out} = width da imagem de saída

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

Procedimento de Treinamento

- Arquitetura de DNN
- Otimizador
- Data Loader
- Loss Function

Otimizadores

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

Otimizadores

- Gradient Descent (GD)
  - Calcula o forward e o backward pass para o dataset todo
  - Computa o gradiente conjunto para todas as amostras
  - Extremamente caro

Otimizadores

- Stochastic Gradient Descent (SGD)
  - Computa o gradiente para apenas um mini-batch em cada iteração
  - Produz resultados menos precisos que o GD
  - Suscetível a mínimos locais

SGD

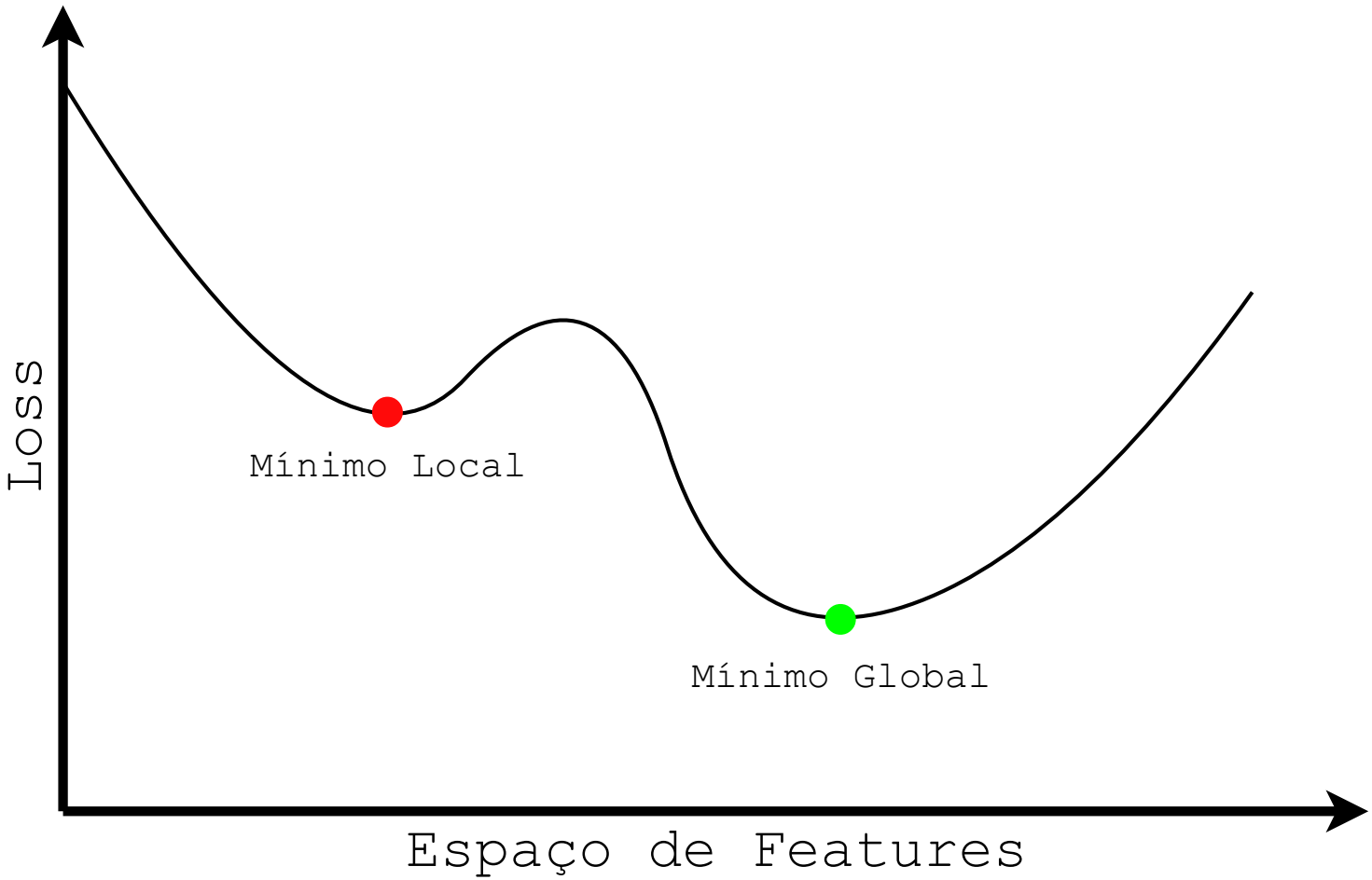


Figura: Exemplo de SGD.

SGD

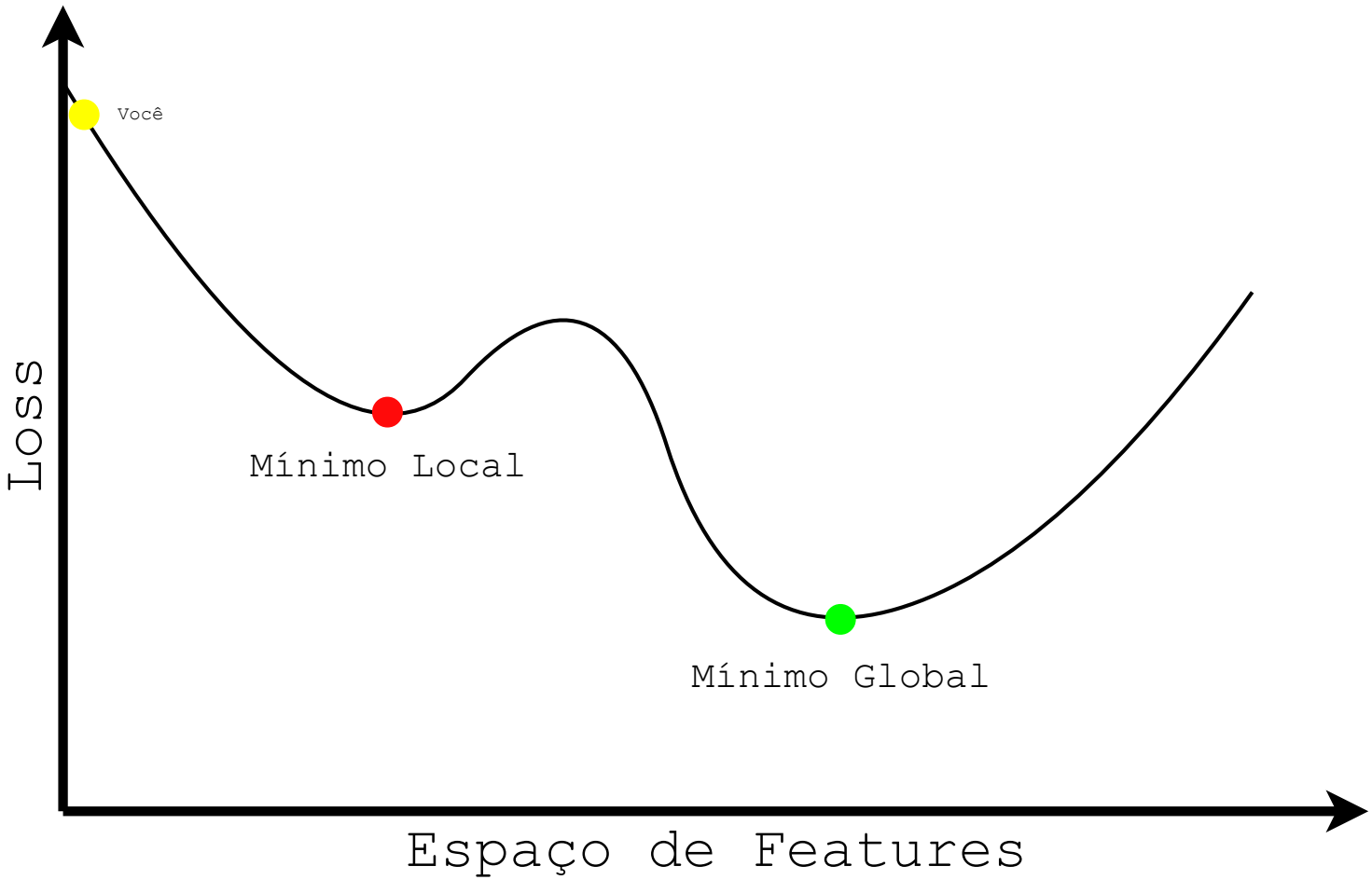


Figura: Exemplo de SGD.

SGD

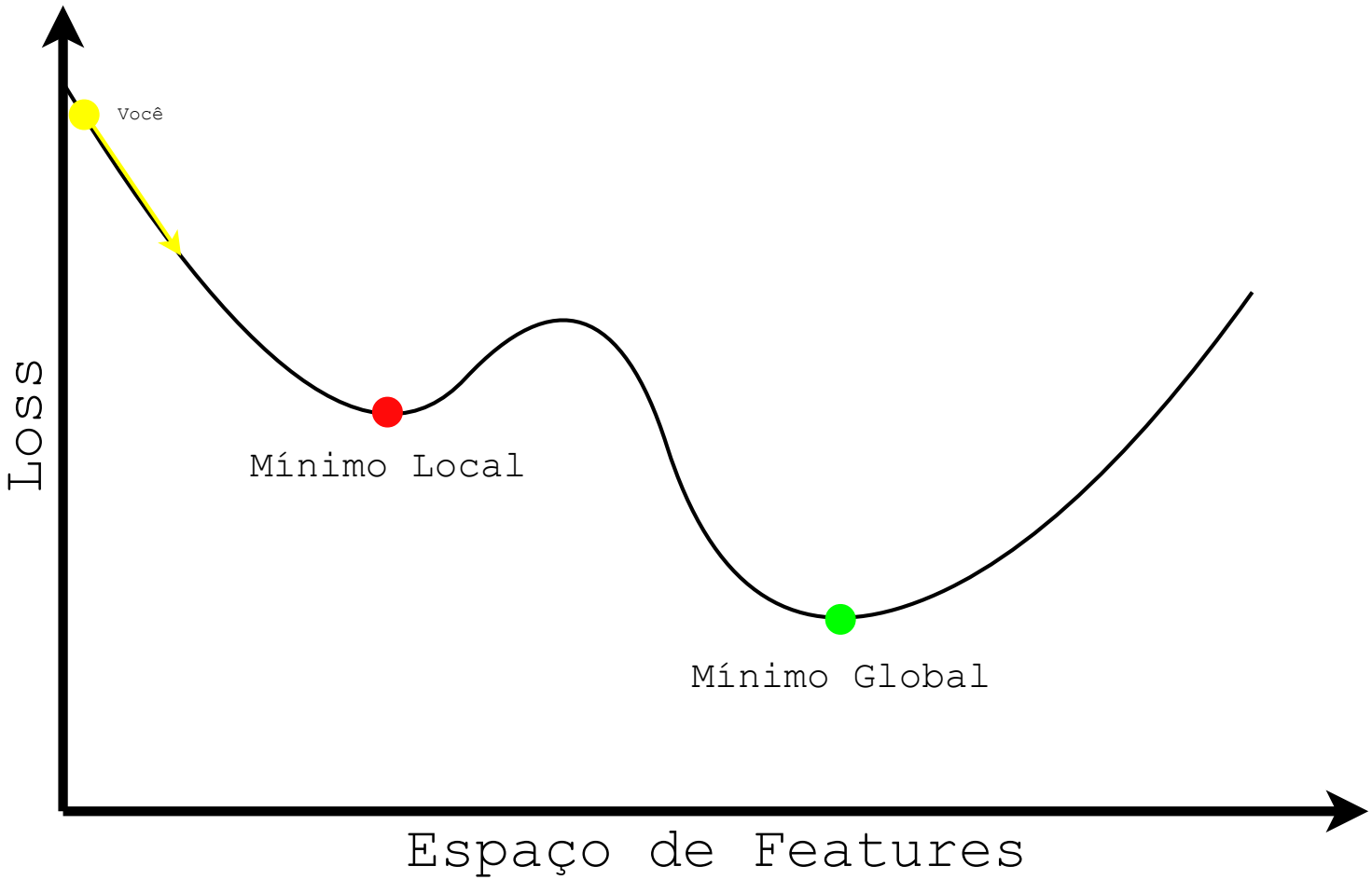


Figura: Exemplo de SGD.

SGD

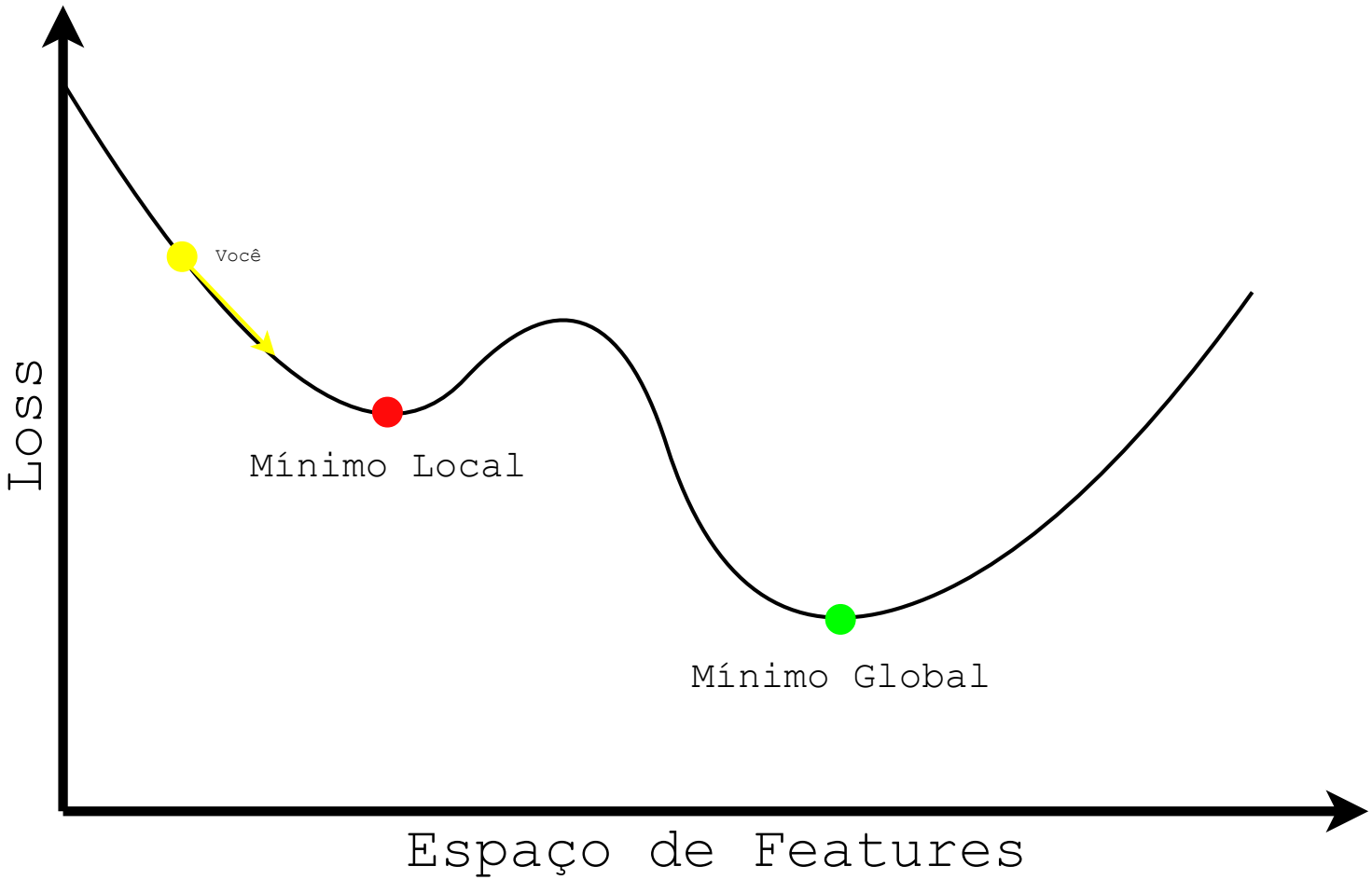


Figura: Exemplo de SGD.

SGD

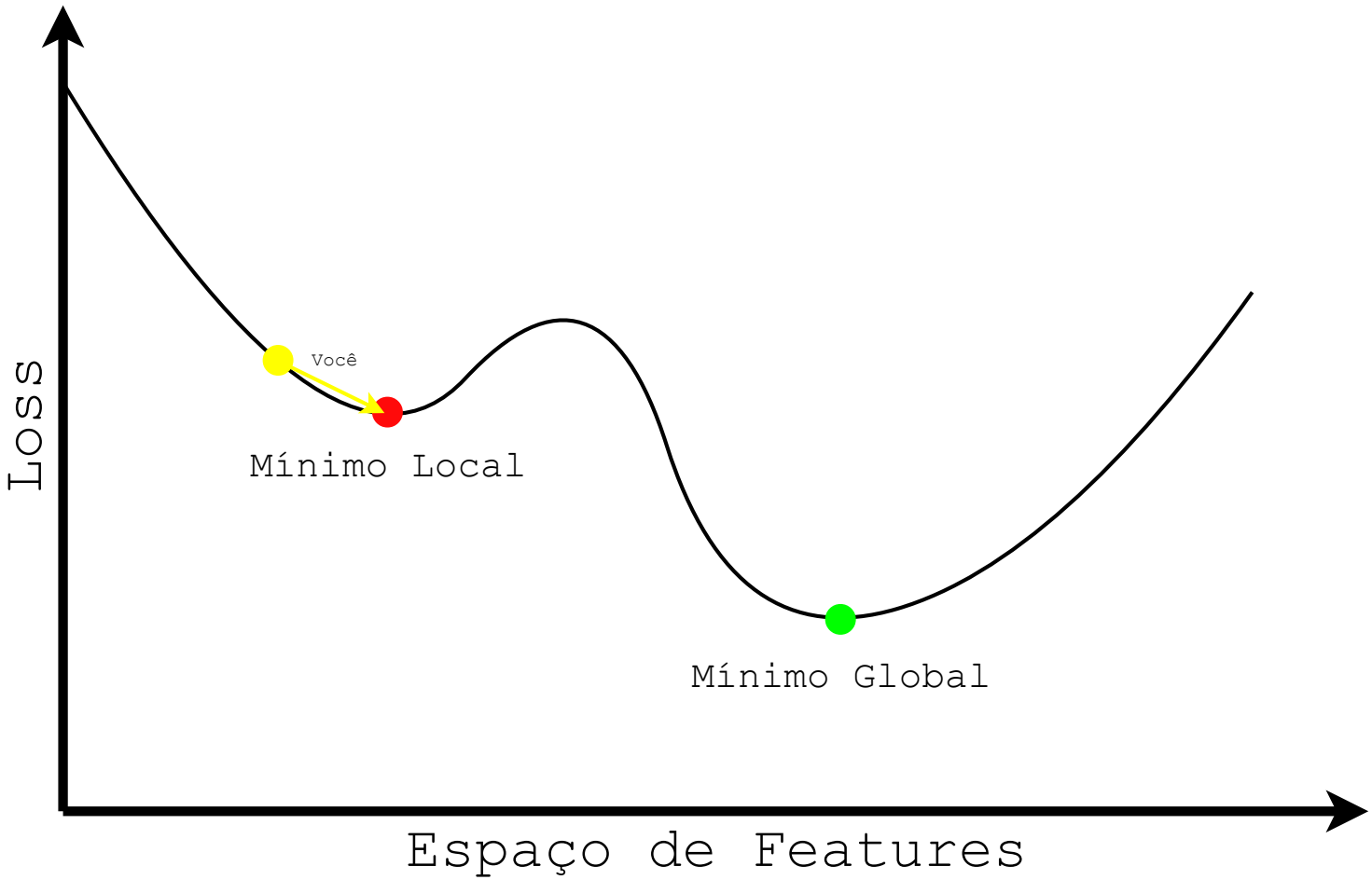


Figura: Exemplo de SGD.

SGD

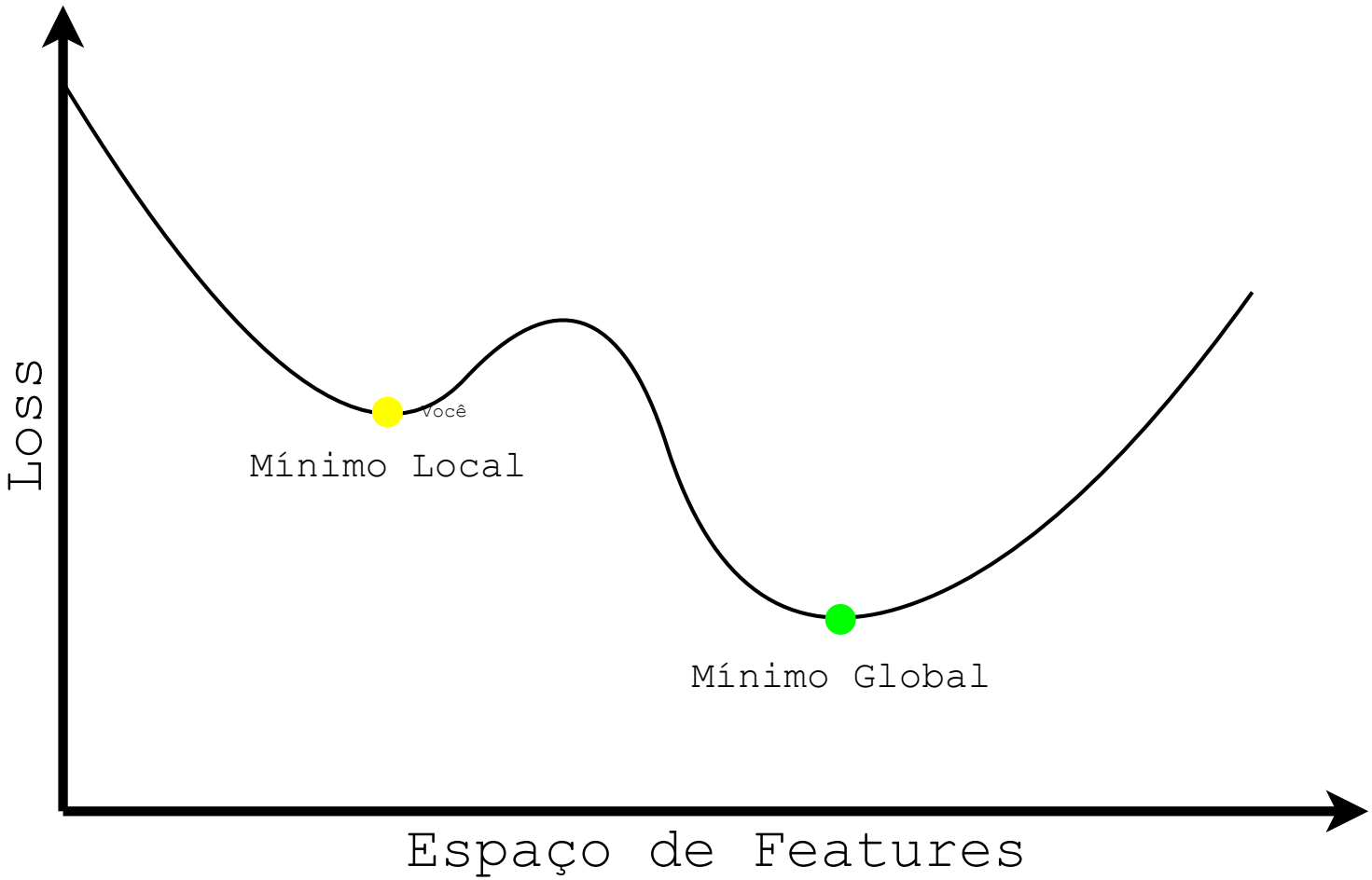


Figura: Exemplo de SGD.

Otimizadores

- Momentum
  - Considera a “velocidade” dos steps da rede
  - Maior robustez a mínimos locais

Momentum

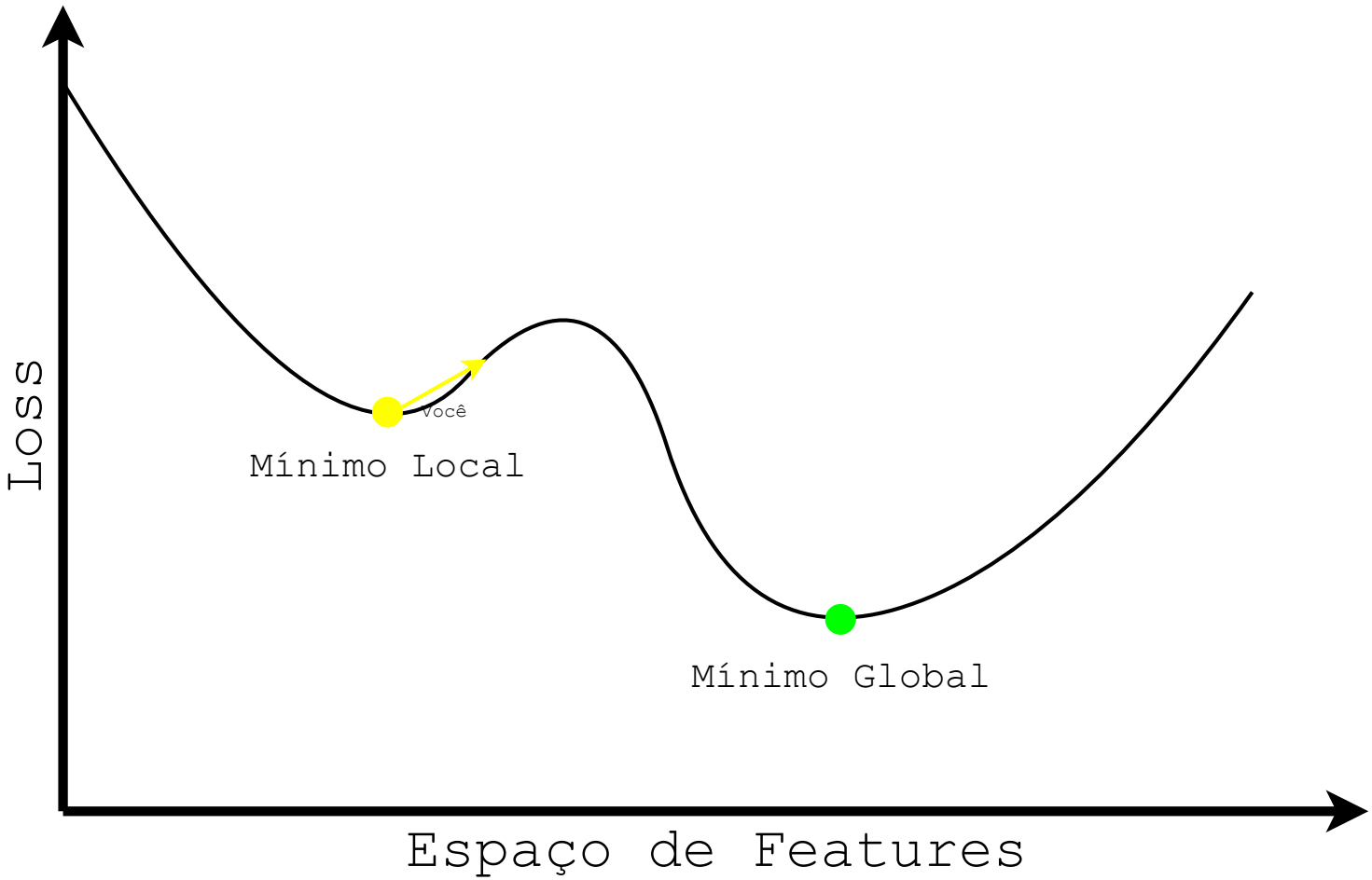


Figura: Exemplo de Momentum.

Momentum

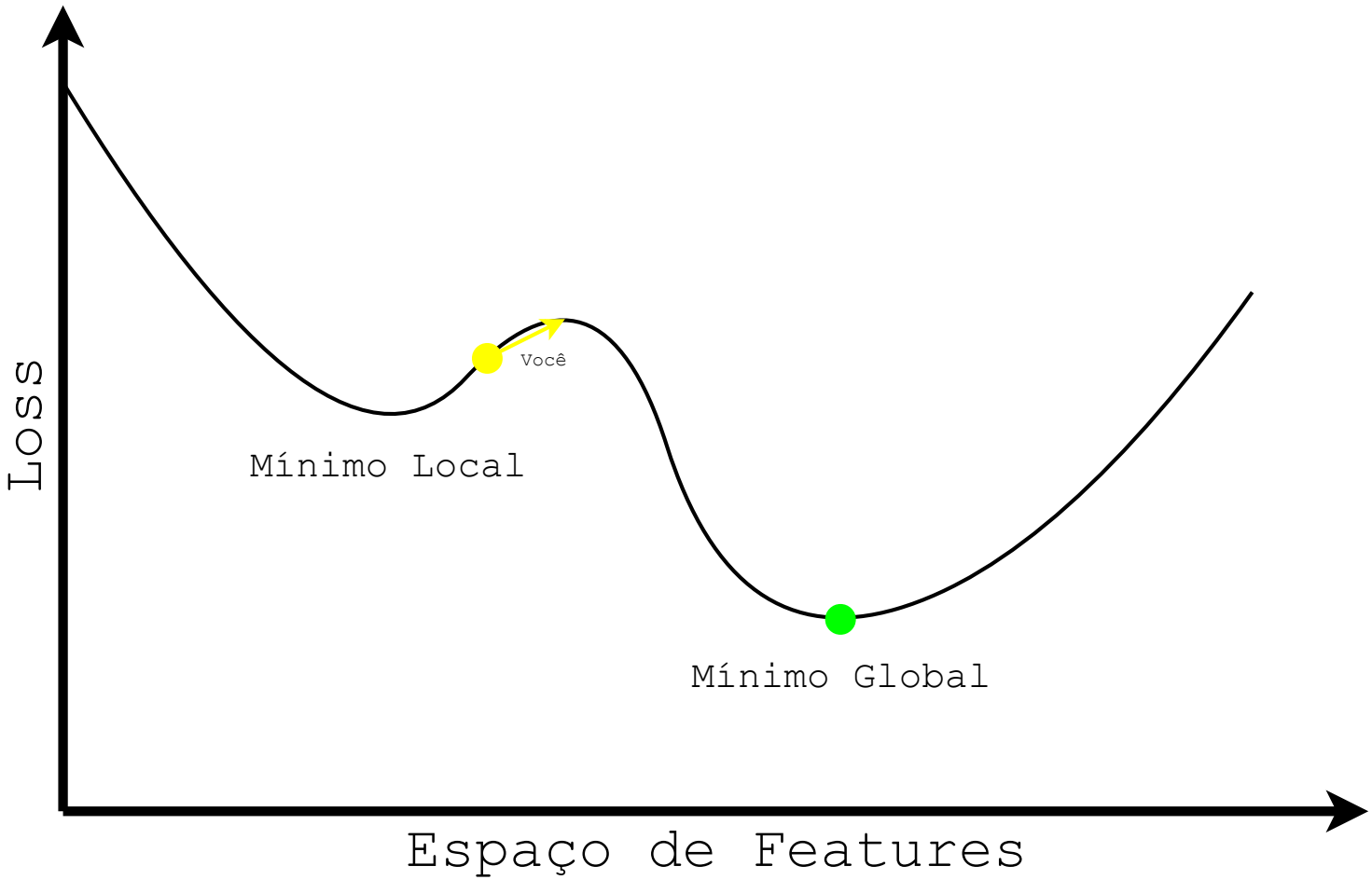


Figura: Exemplo de Momentum.

Momentum

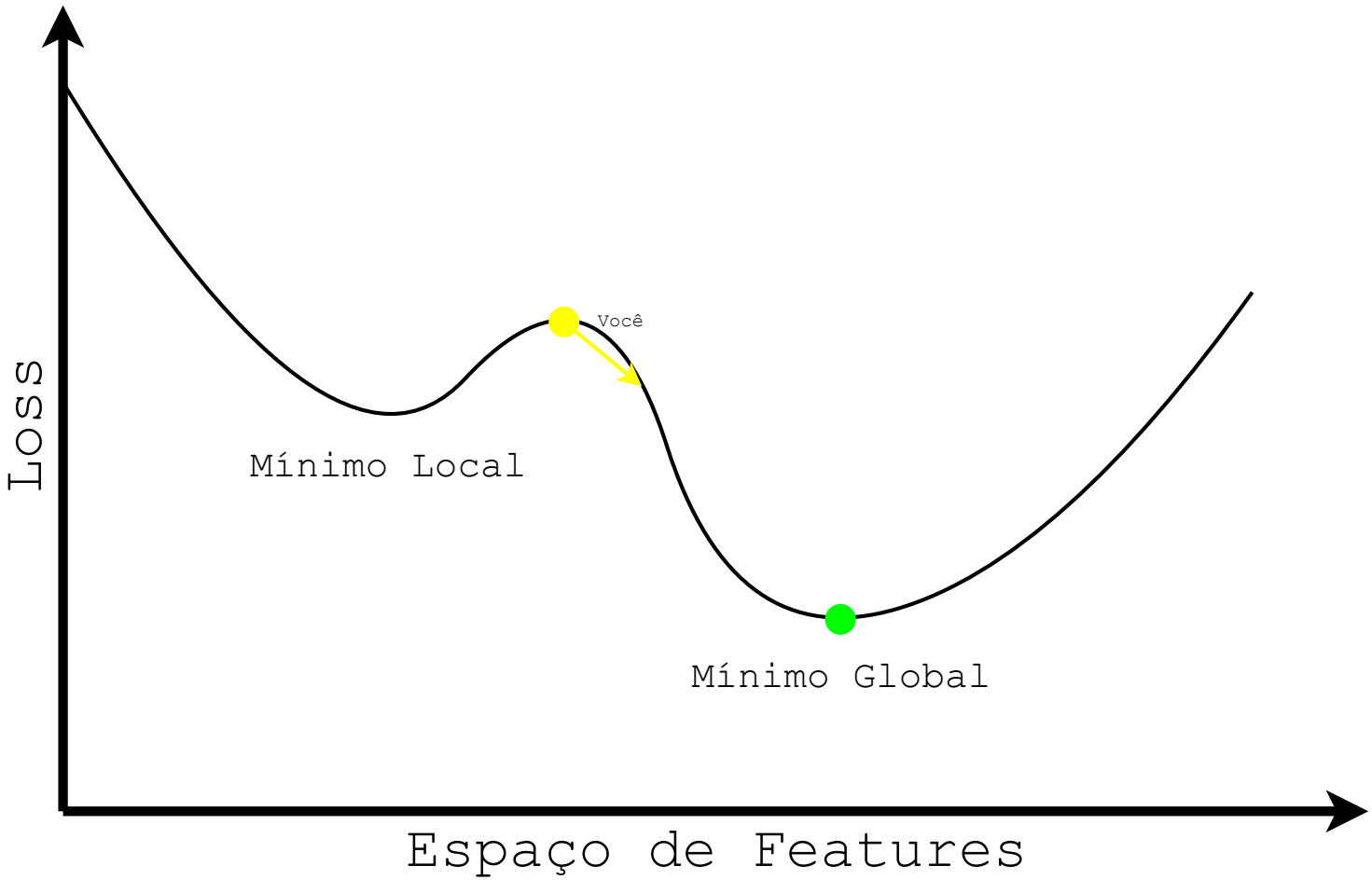


Figura: Exemplo de Momentum.

Momentum

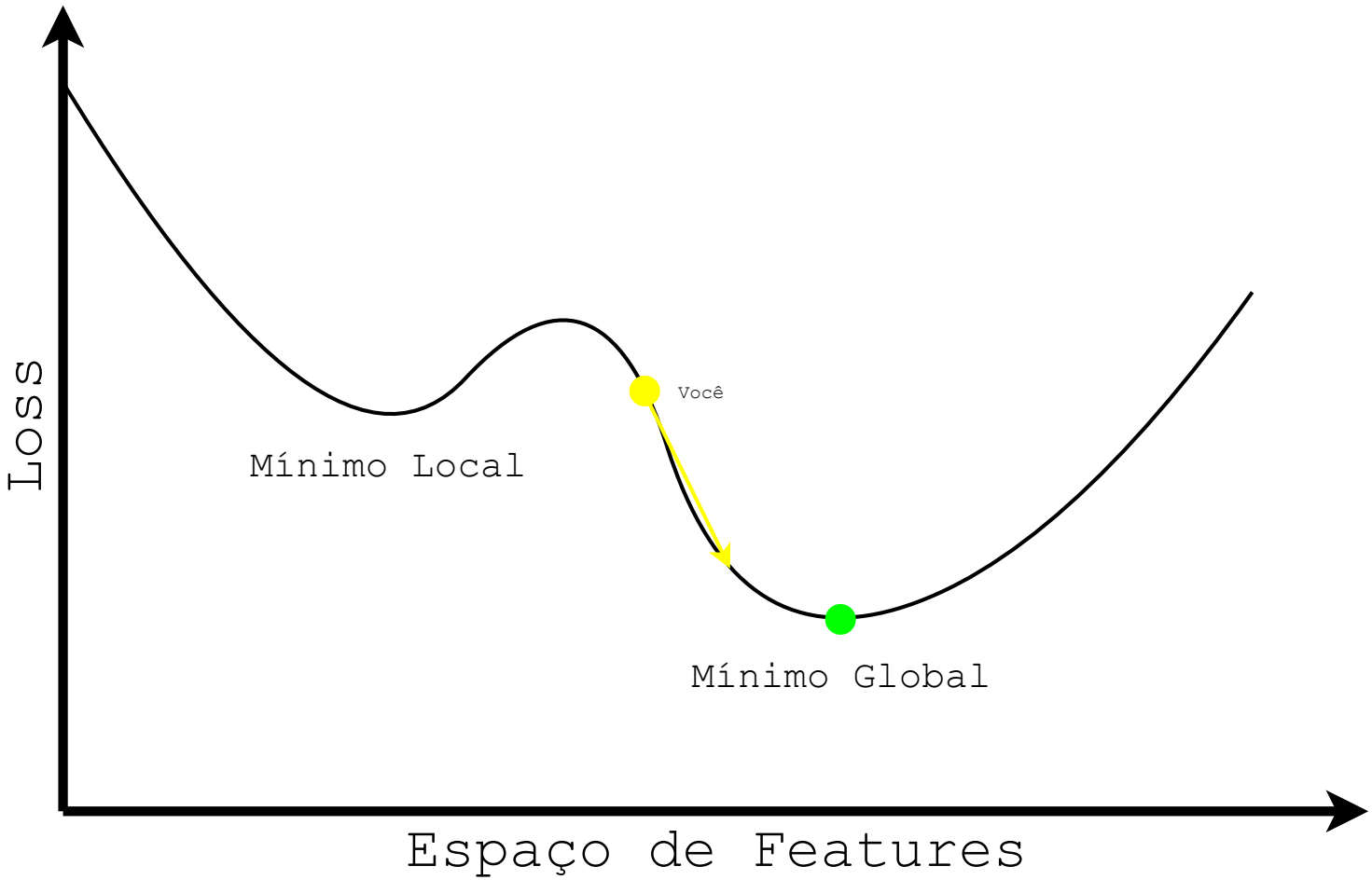


Figura: Exemplo de Momentum.

Momentum

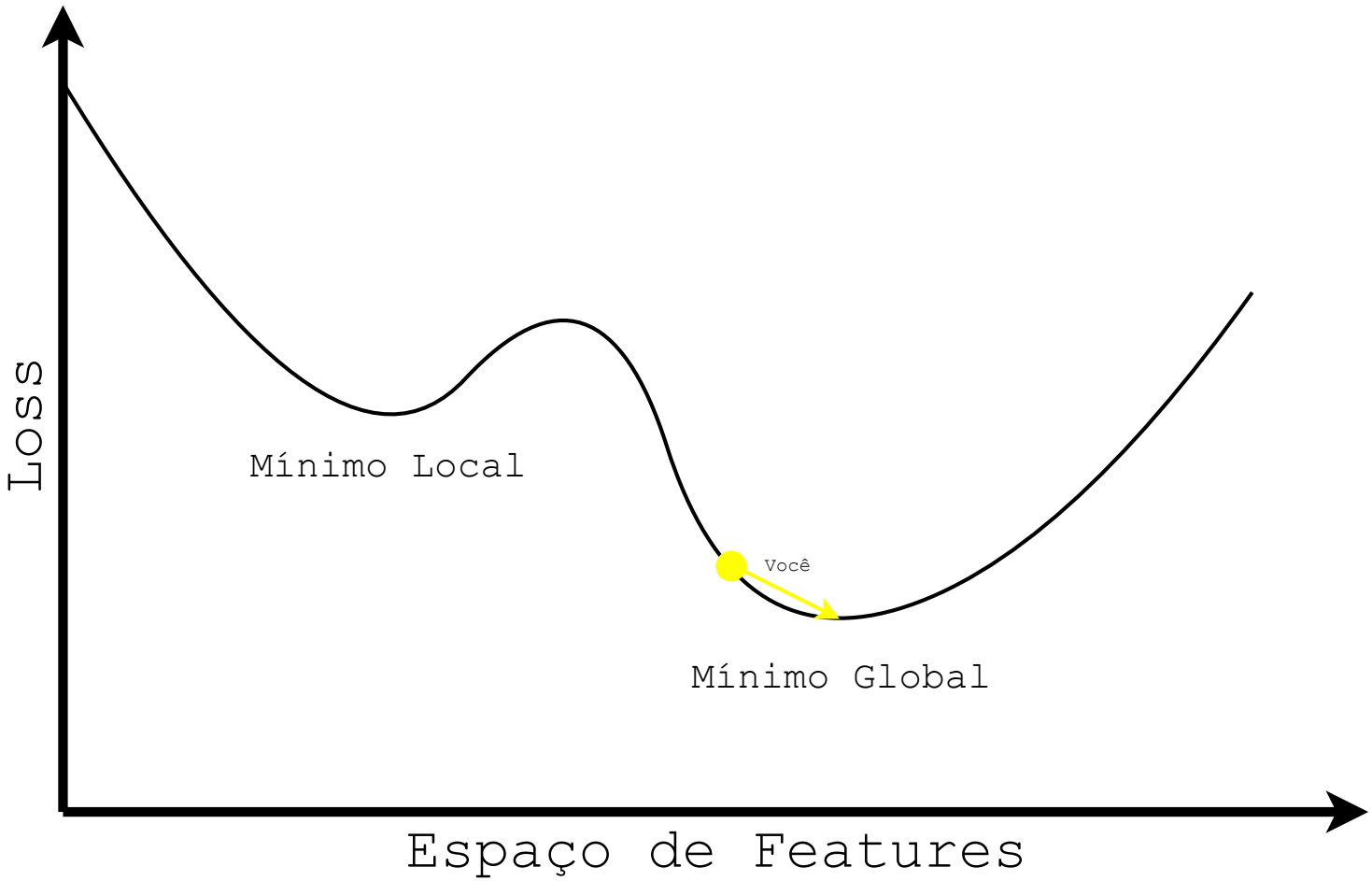


Figura: Exemplo de Momentum.

Momentum

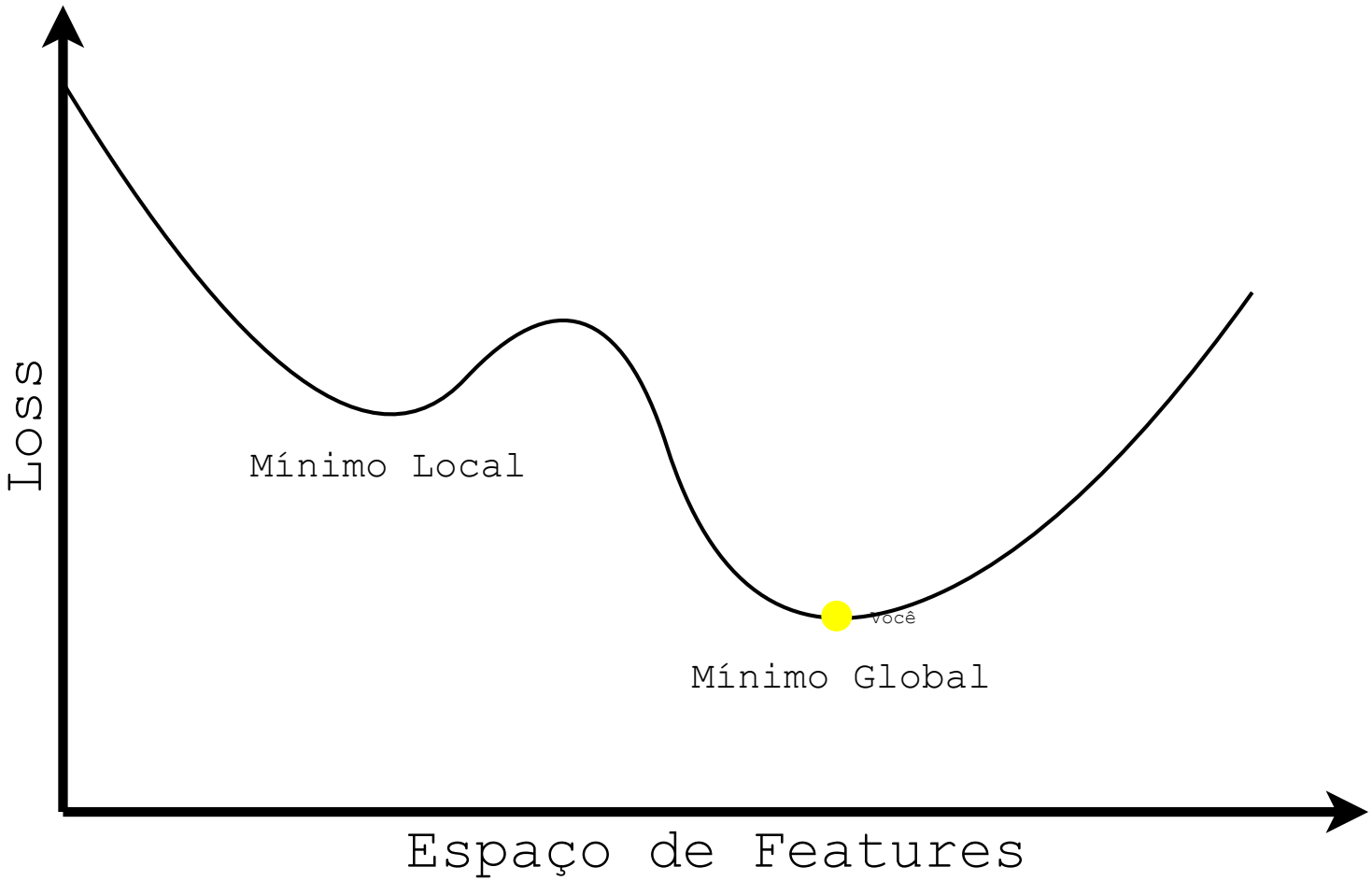


Figura: Exemplo de Momentum.

Otimizadores

- Adaptive (Sub)Gradient (AdaGrad) [7]
  - Velocidade de mudança diferente para cada parâmetro na NN
  - Em geral, para de convergir consideravelmente antes de chegar no Mínimo Global
  - Tende a resultar em underfitting

Otimizadores

- RMSprop
  - Acaba com o problema de underfitting do AdaGrad
  - Fórmula de otimização não publicada
  - Retirada do curso no Coursera do Geoffrey Hinton [6]

[6] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Otimizadores

- Adam
  - Junta o melhor do Momentum com o melhor do AdaGrad/RMSprop
  - Estado da Arte atualmente

Otimizadores

Comparação de Otimizadores

Para mais informações sobre otimizadores, visitar o blog post “An overview of gradient descent optimization algorithms”[7]

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

Procedimento de Treinamento

Exercício Prático - Training Procedure

Training_Procedure.ipynb

Agenda

Convoluções 2D

- Operação que mede a “semelhança” entre dois sinais
- Imagem: $f(.)$
- Kernel: $h(.)$
- $f(x,y) * h(x,y) = \frac{1}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m,n) h(x-m, y-n)$,
  para $x = 0, 1, ..., X-1$ e $y = 0, 1, ..., Y-1$

Convoluções e Correlações

Convoluções e Correlações

Convoluções e Correlações podem ser usadas para armazenar informação visual

Convoluções e Correlações

Convoluções e Correlações

Convoluções e Correlações podem ser usadas para fazer casamento de padrões visuais

Convoluções e Correlações

Convoluções e Correlações

As respostas geradas por Convoluções e Correlações serão mais fortes nas áreas da imagem que casarem melhor com o conteúdo do kernel

Convoluções e Correlações

Demo - Correlation and Convolution

Correlation_Convolution.ipynb

CNNs

- Redes Neurais Convolucionais (CNNs) surgiram:
  - há bastante tempo [8]
  - originalmente para imagens
  - mas já foram adaptadas para texto [9] e informação temporal/vídeo [10]

CNNs

- CNNs ficaram mais conhecidas a partir 2012
  - quando foram base do método usado para ganhar uma competição de classificação muito conhecida

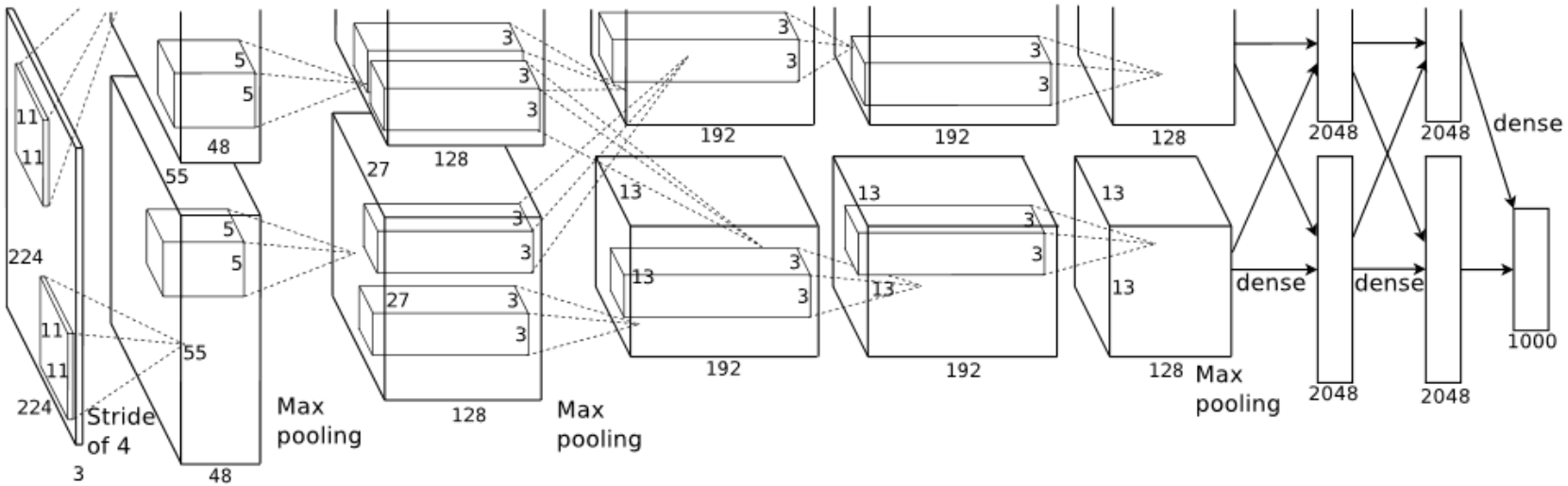
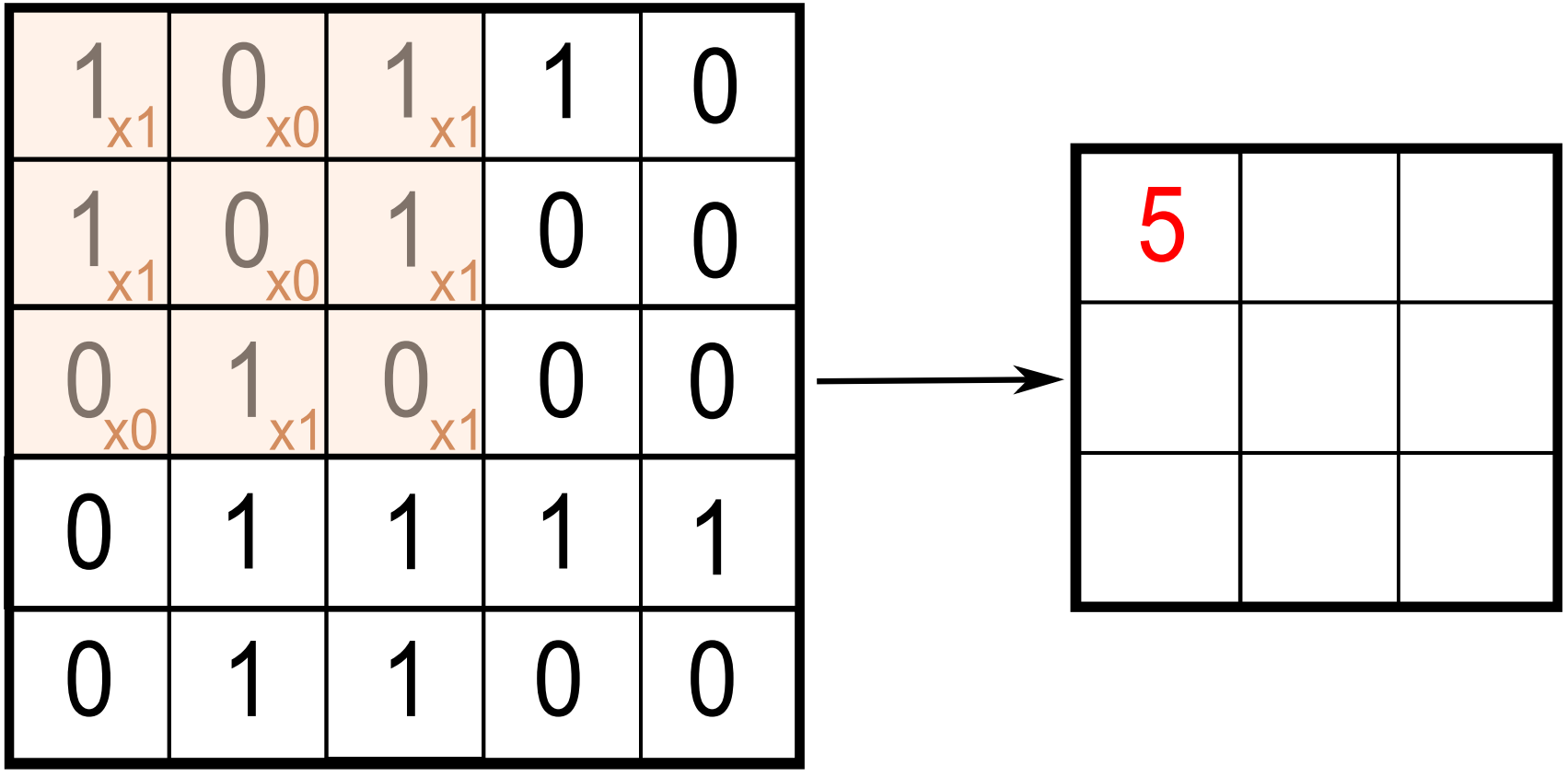
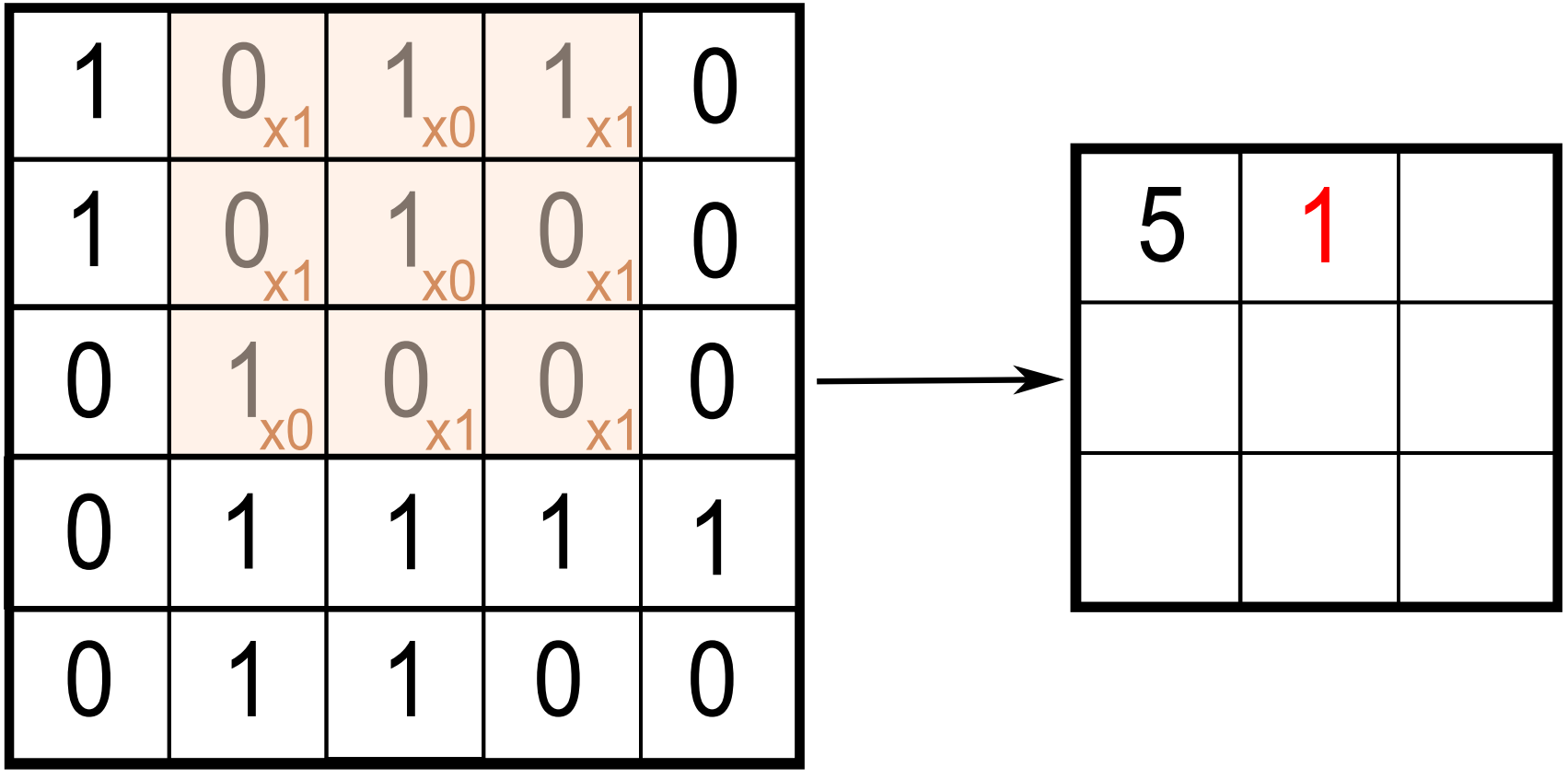


Figura: Arquitetura da AlexNet [11], CNN vencedora de importante competição em 2012

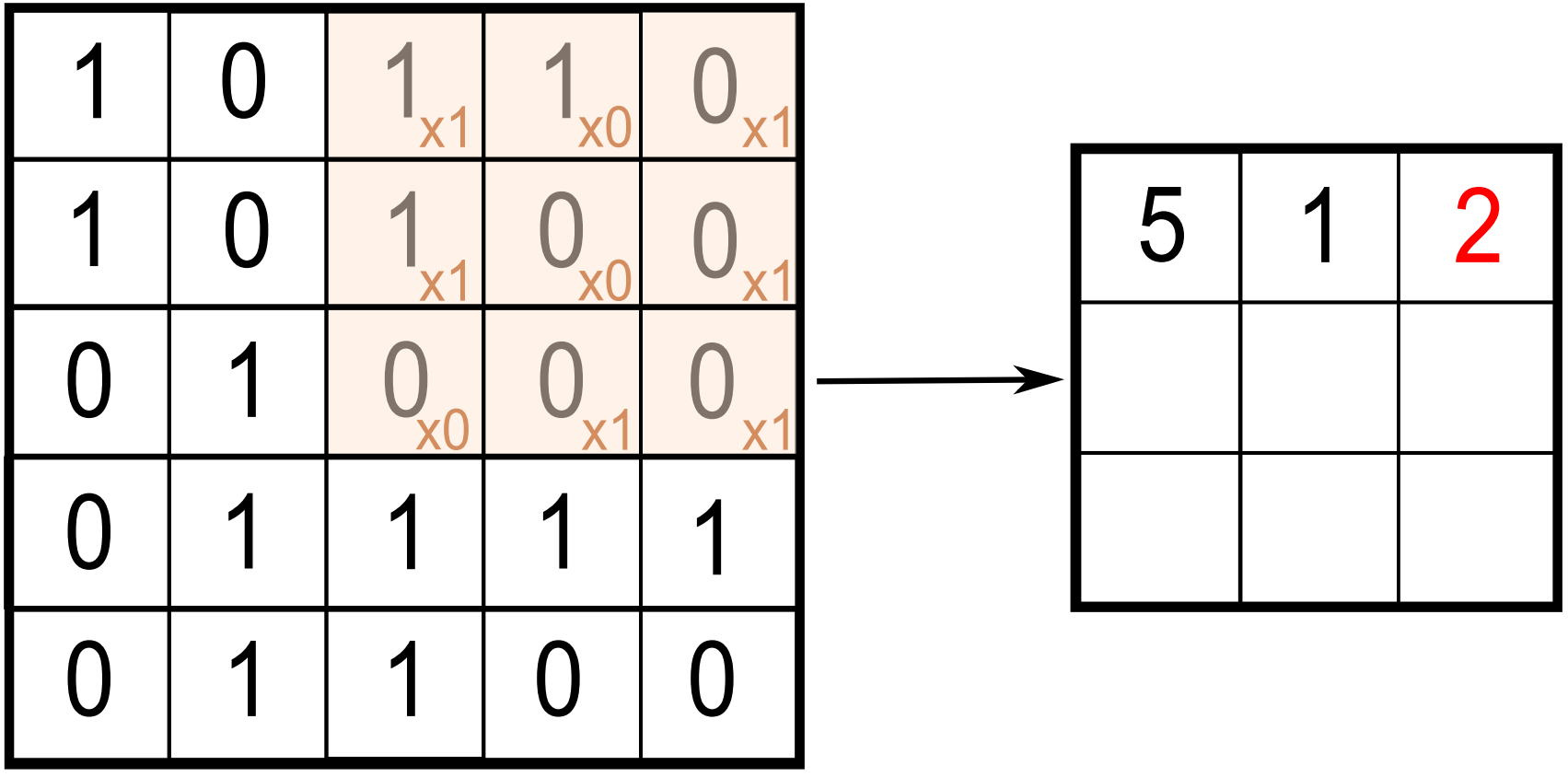
Camadas Convolucionais



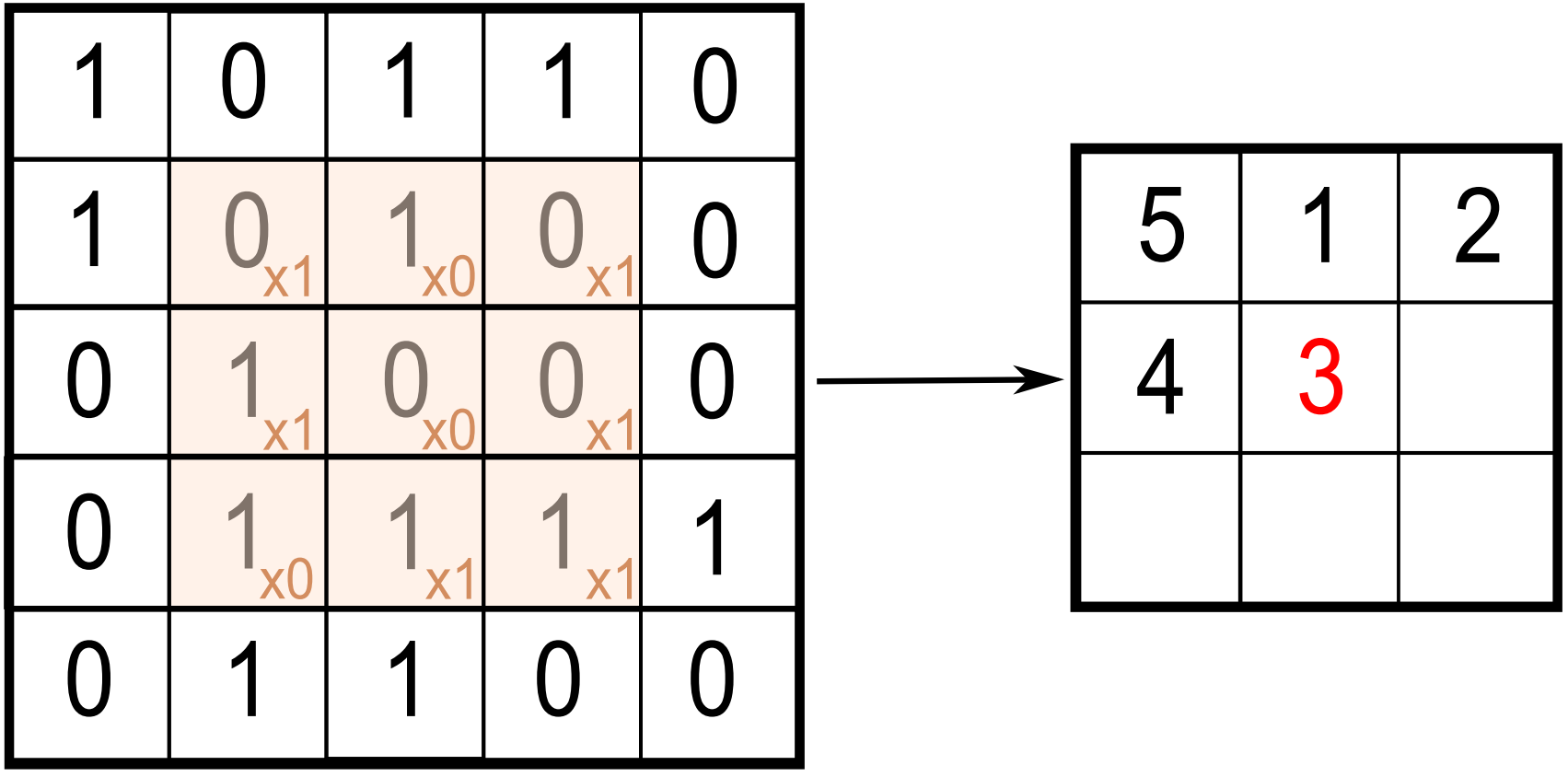
Camadas Convolucionais



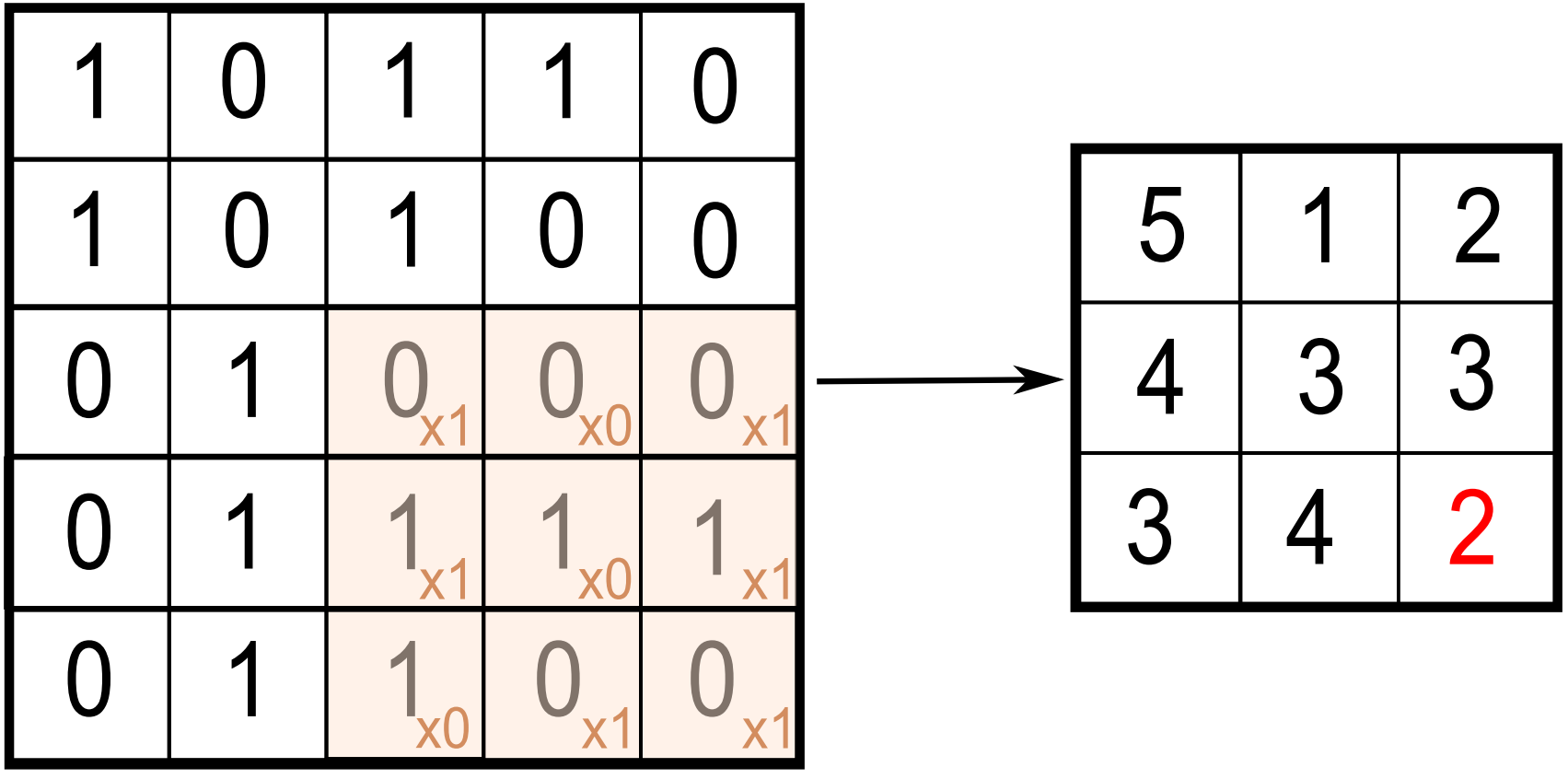
Camadas Convolucionais



Camadas Convolucionais



Camadas Convolucionais



Convoluções

Demo - 2D Convolutions

2D_Convolutions.ipynb

Redes Convolucionais

- Nessas camadas, o feed-forward acontece da mesma forma
- Mas considera a operação de convolução para definir os valores (vizinhança) considerados como entrada para o neurônio

Camadas Convolucionais

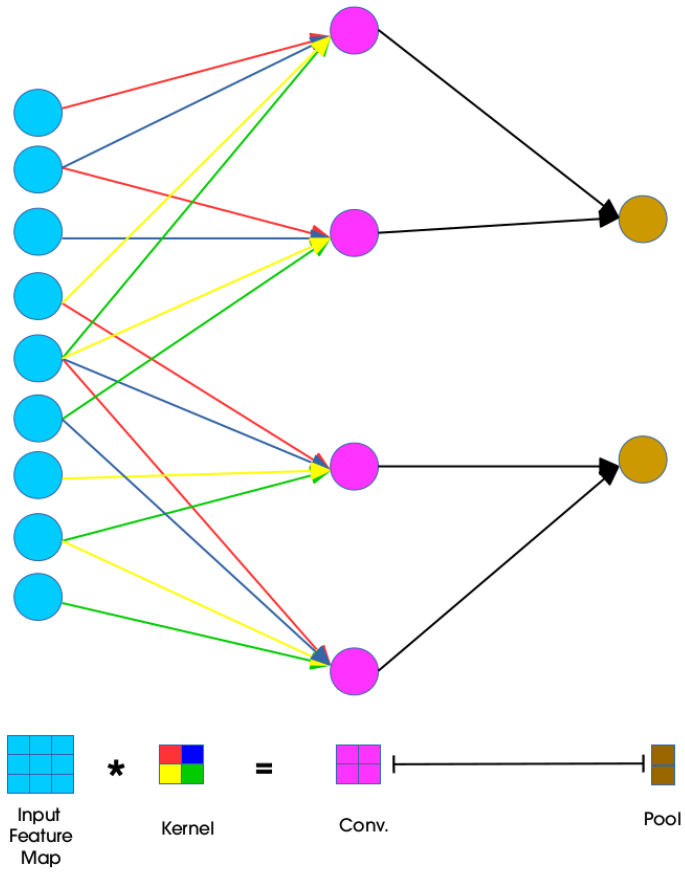


Figura: Exemplo [12] de feed-forward para uma CNN.

Redes Convolucionais

- O backpropagation precisa seguir a mesma lógica
- A vizinhança usada no feed-forward deve ser levada em consideração

Camadas Convolucionais

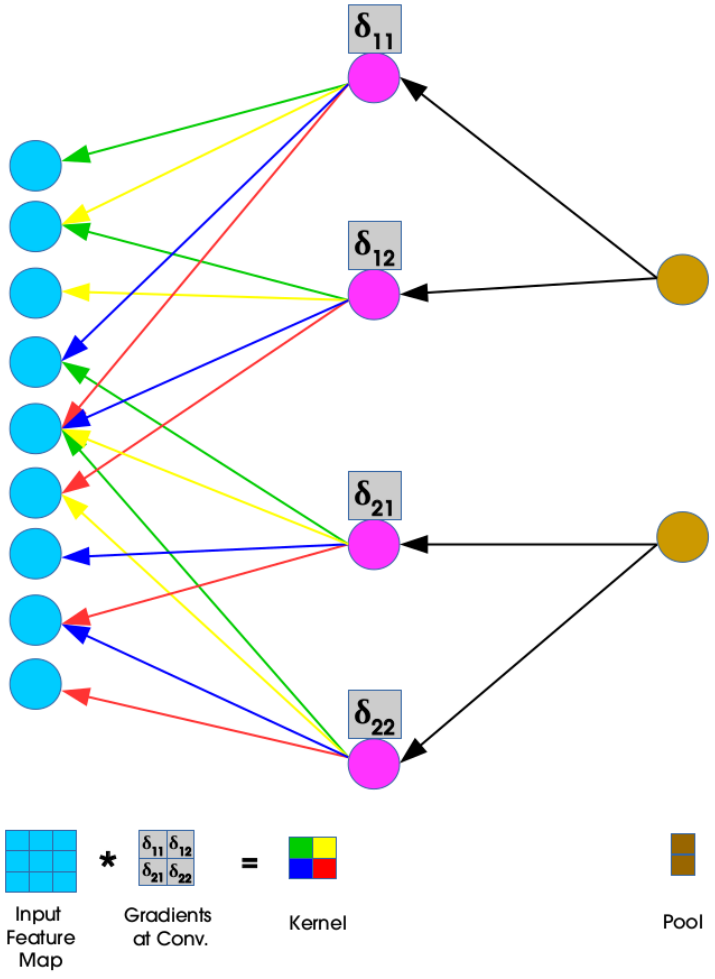


Figura: Exemplo [12] de backpropagation para uma rede convolucional

Redes Convolucionais

- Como é a mesma vizinhança, podemos também fazer uma convolução durante o backpropagation para calcular e propagar o erro
- Isso torna o algoritmo mais rápido

CNNs

- Camadas comuns em CNNs:
  - Camadas Convolucionais
  - Camadas de Pooling
  - Camadas Totalmente Conectadas (“Fully Connected”, “Linear”, ou “Dense”)

Camadas Convolucionais

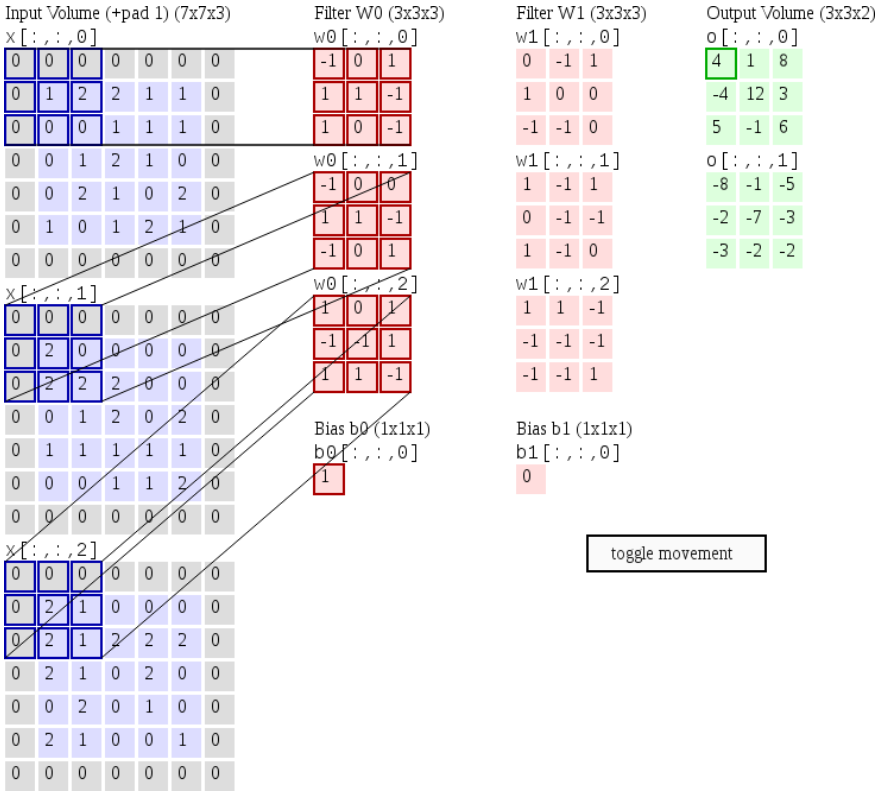


Figura: Convoluções em uma CNN.

Camadas Convolucionais

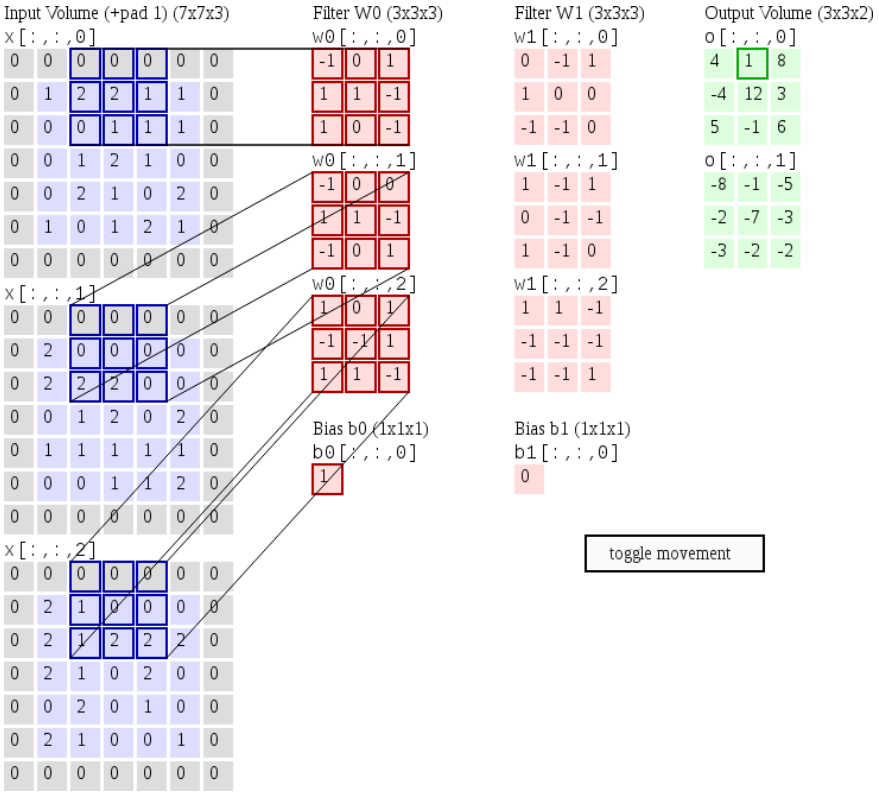


Figura: Convoluções em uma CNN.

Camadas Convolucionais

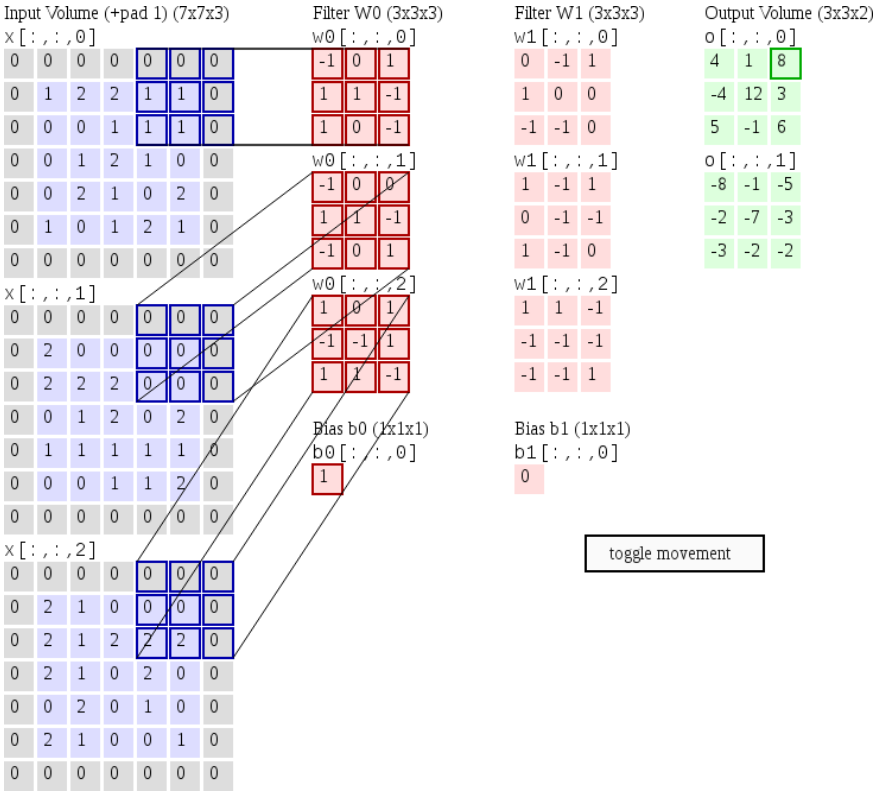


Figura: Convoluções em uma CNN.

Camadas Convolucionais

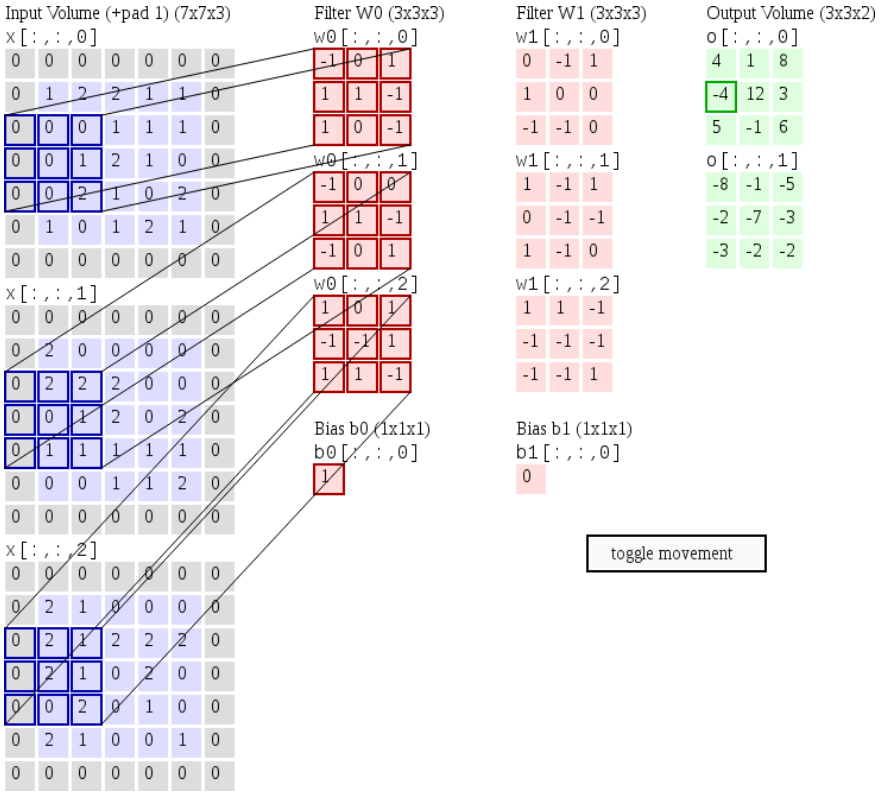


Figura: Convoluções em uma CNN.

Camadas Convolucionais

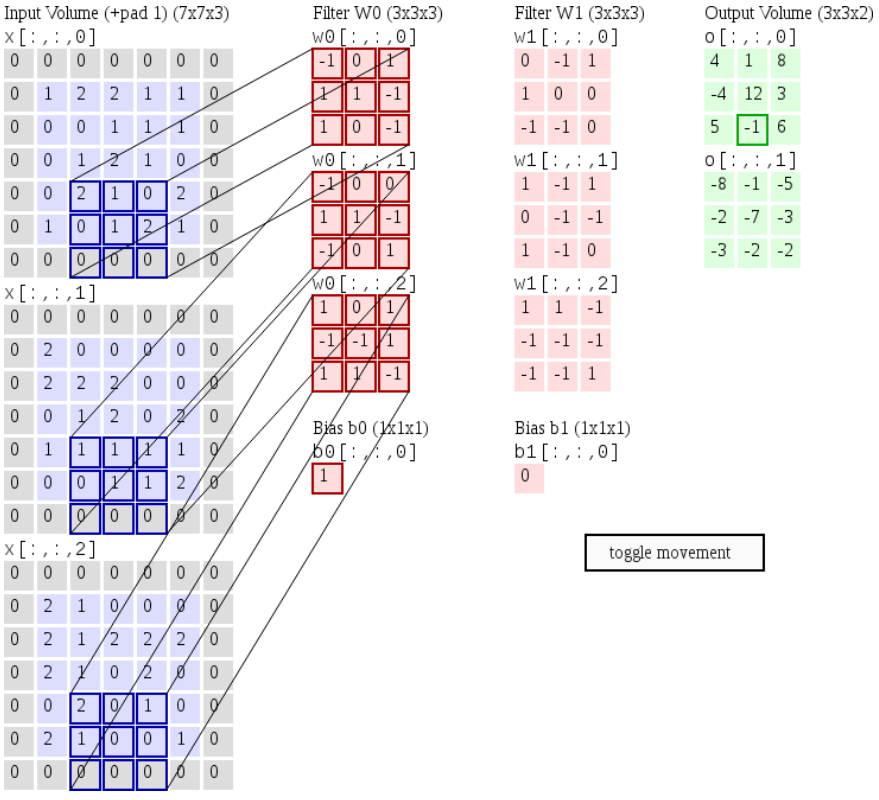


Figura: Convoluções em uma CNN.

Camadas Convolucionais

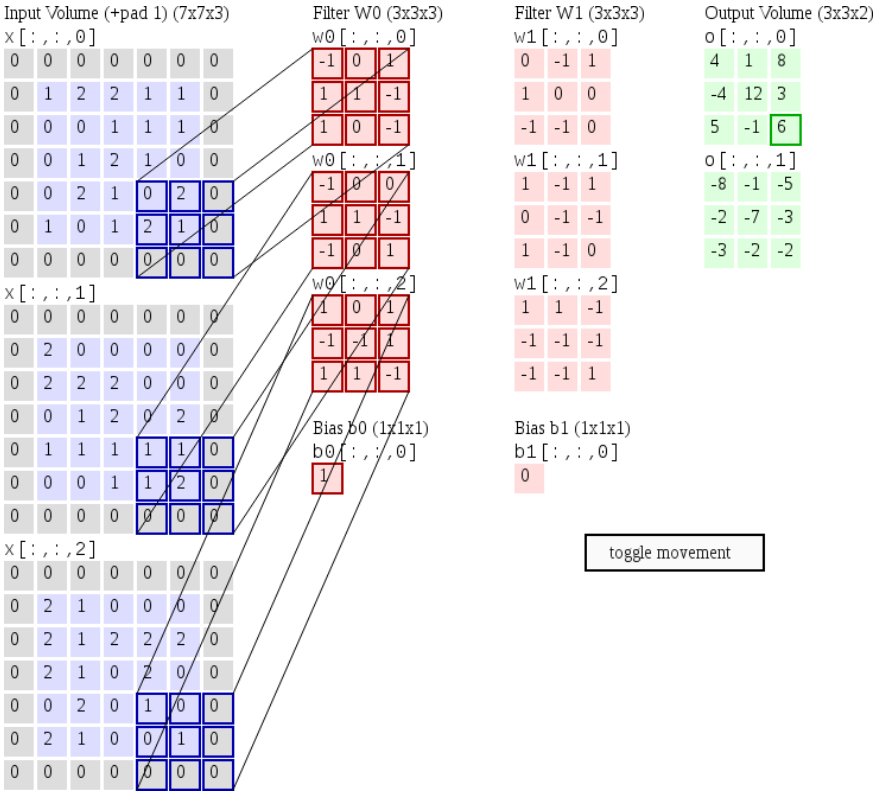


Figura: Convoluções em uma CNN.

Camadas Convolucionais

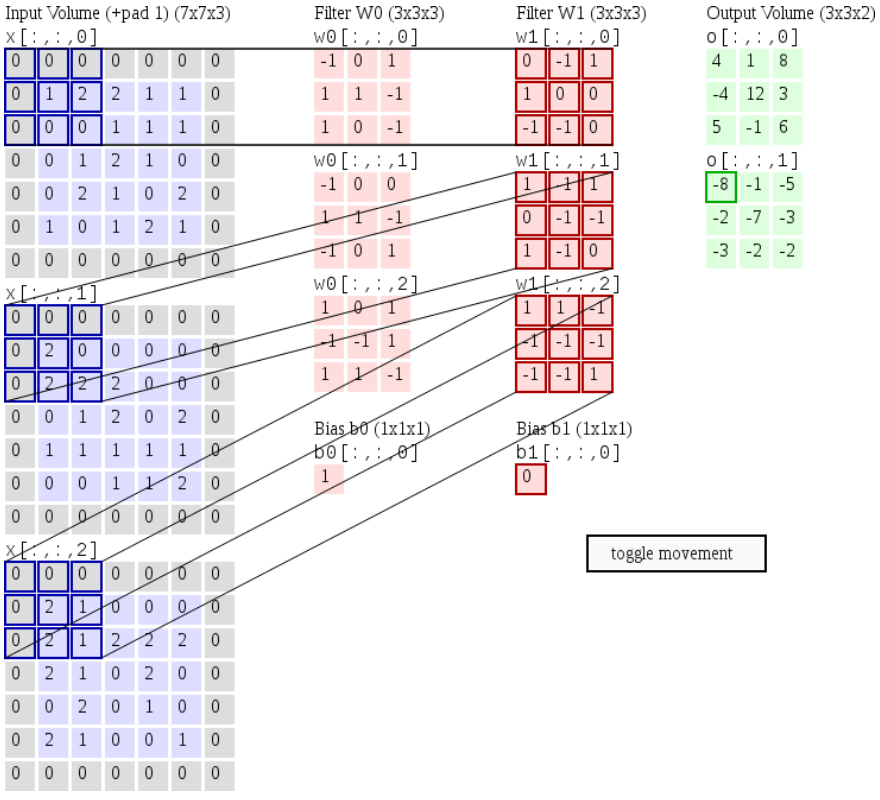


Figura: Convoluções em uma CNN.

Camadas Convolucionais

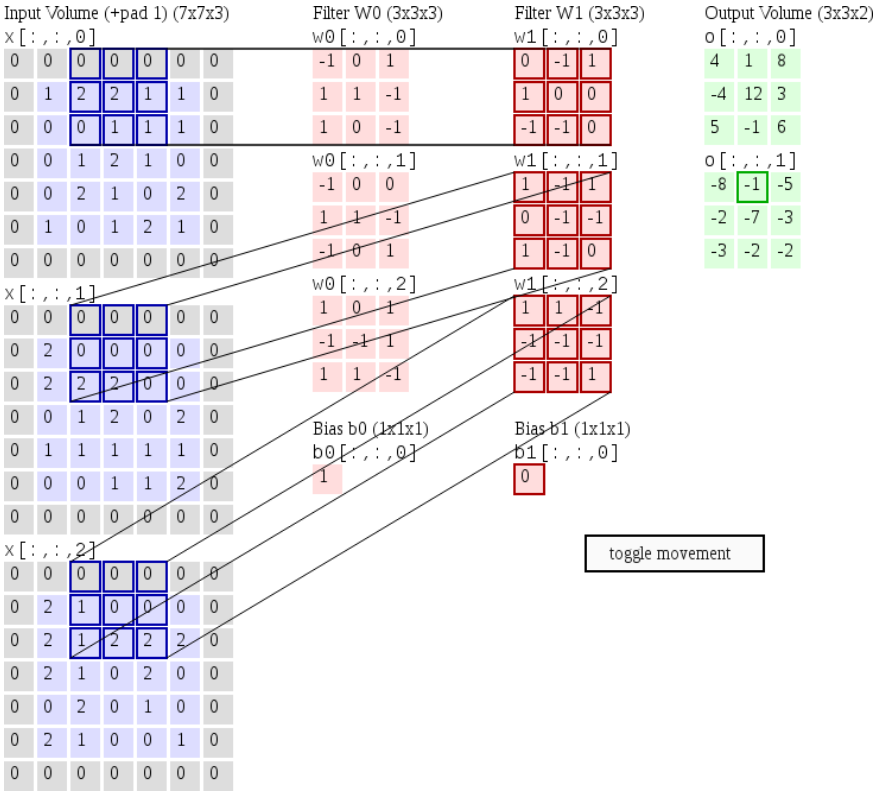


Figura: Convoluções em uma CNN.

Camadas Convolucionais

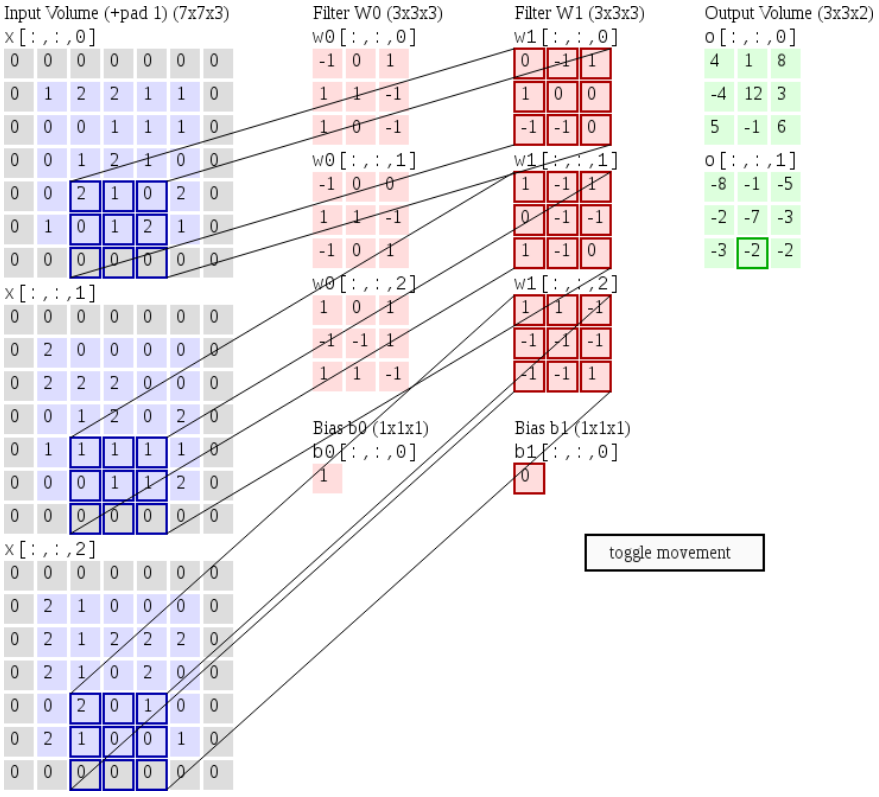


Figura: Convoluções em uma CNN.

Camadas Convolucionais

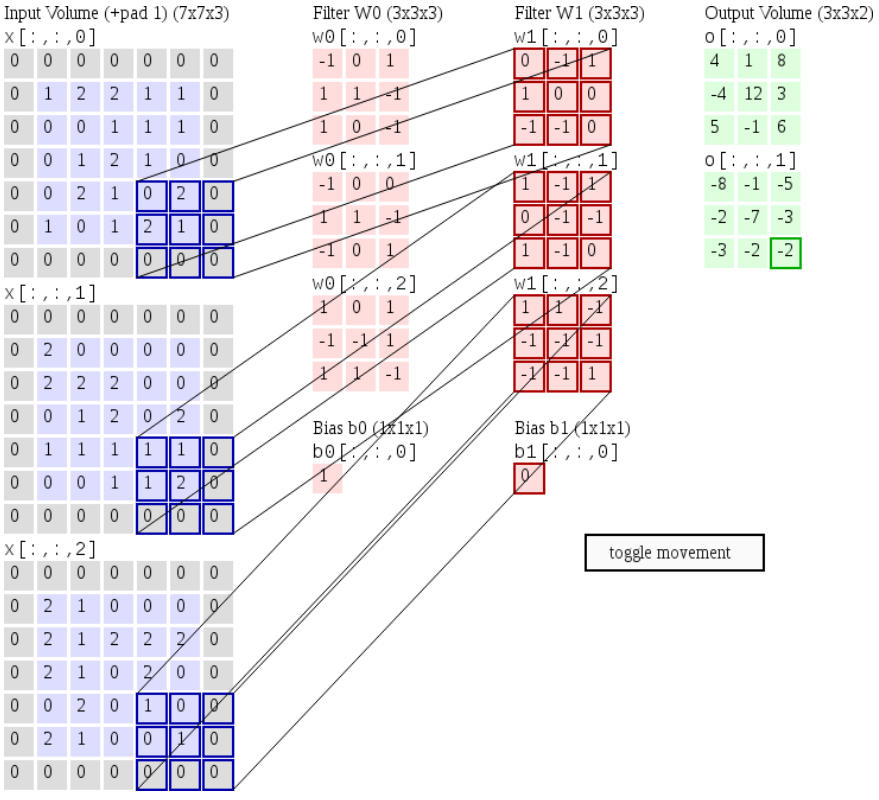


Figura: Convoluções em uma CNN.

Camadas Convolucionais

- Parâmetros de Camadas Convolucionais:
  - Field of View (F)
  - Stride (S)
  - Dilation (D)
  - Padding (P)

Receptive Field de Tamanho 3

100	150	201	202	202
105	155	202	203	203
105	155	202	203	203
110	160	203	204	204
110	160	203	204	204

Receptive Field de Tamanho 5

100	150	201	202	202
105	155	202	203	203
105	155	202	203	203
110	160	203	204	204
110	160	203	204	204

Receptive Field de Tamanho 4

100	150	201	202	202
105	155	202	203	203
105	155	202	203	203
110	160	203	204	204
110	160	203	204	204

Receptive Field de Tamanho 2x4

100	150	201	202	202
105	155	202	203	203
105	155	202	203	203
110	160	203	204	204
110	160	203	204	204

Stride de Tamanho 1

100	150	201	202	202
105	155	202	203	203
105	155	202	203	203
110	160	203	204	204
110	160	203	204	204

Stride de Tamanho 1

100	150	201	202	202
105	155	202	203	203
105	155	202	203	203
110	160	203	204	204
110	160	203	204	204

Stride de Tamanho 1

100	150	201	202	202
105	155	202	203	203
105	155	202	203	203
110	160	203	204	204
110	160	203	204	204

Stride de Tamanho 2

100	150	201	202	202
105	155	202	203	203
105	155	202	203	203
110	160	203	204	204
110	160	203	204	204

Stride de Tamanho 2

100	150	201	202	202
105	155	202	203	203
105	155	202	203	203
110	160	203	204	204
110	160	203	204	204

Dilation de Tamanho 1 com Kernel 3x3

100	150	201	202	202
105	155	202	203	203
105	155	202	203	203
110	160	203	204	204
110	160	203	204	204

Dilation de Tamanho 2 com Kernel 3x3

100	150	201	202	202
105	155	202	203	203
105	155	202	203	203
110	160	203	204	204
110	160	203	204	204

Padding de Tamanho 1

0	0	0	0	0	0	0
0	100	150	201	202	202	0
0	105	155	202	203	203	0
0	105	155	202	203	203	0
0	110	160	203	204	204	0
0	110	160	203	204	204	0
0	0	0	0	0	0	0

Padding de Tamanho 2

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	100	150	201	202	202	0	0
0	0	105	155	202	203	203	0	0
0	0	105	155	202	203	203	0	0
0	0	110	160	203	204	204	0	0
0	0	110	160	203	204	204	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

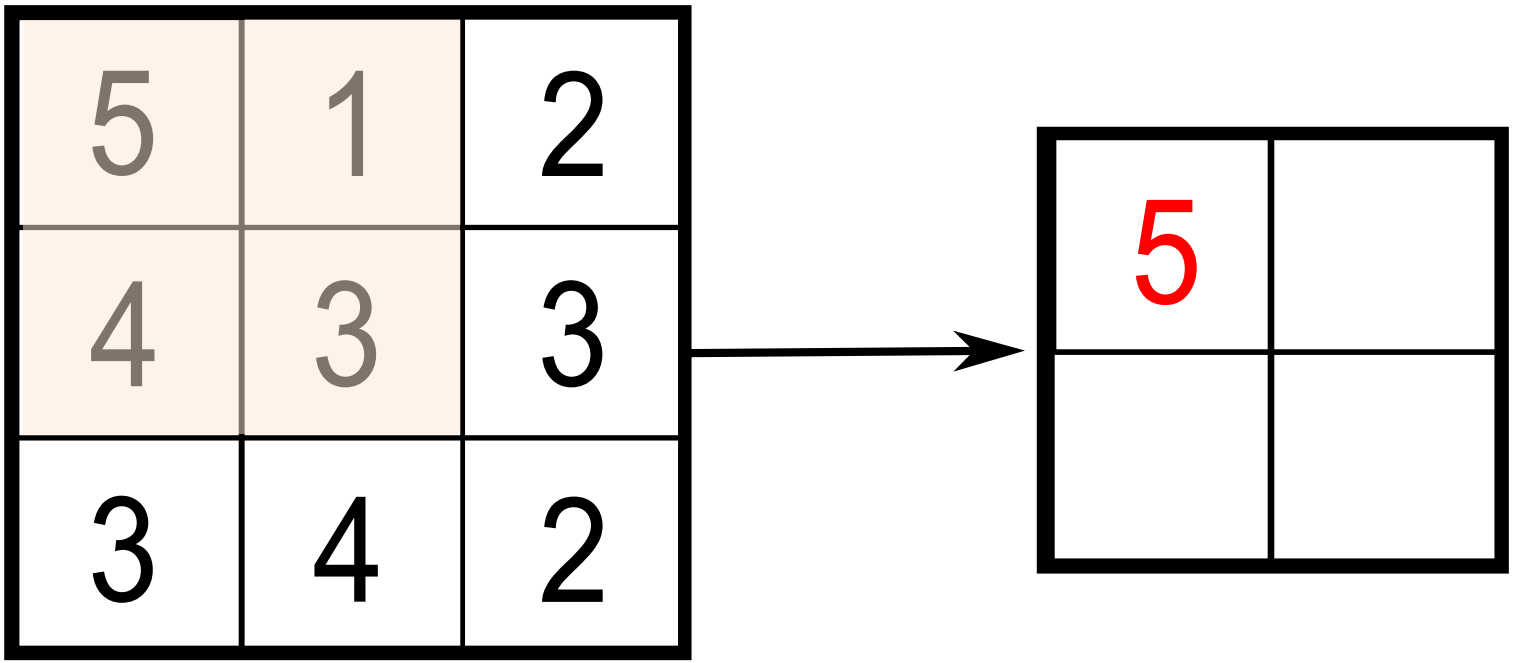
Resolução de Saída de Convolução

Calculando a Resolução Espacial de Saída

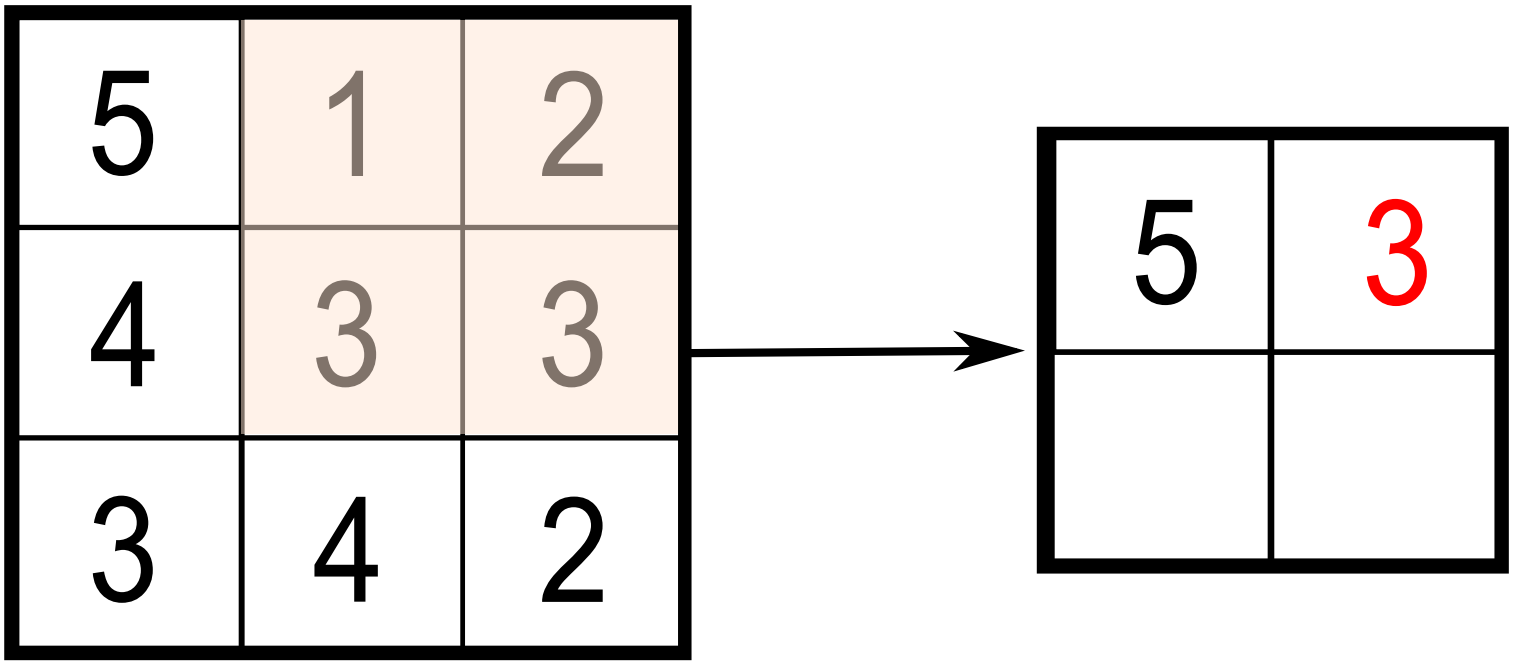
$$W_{out} = ((W_{in} - F_w + 2P_w)/S_w) + 1$$

$$H_{out} = ((H_{in} - F_h + 2P_h)/S_h) + 1$$

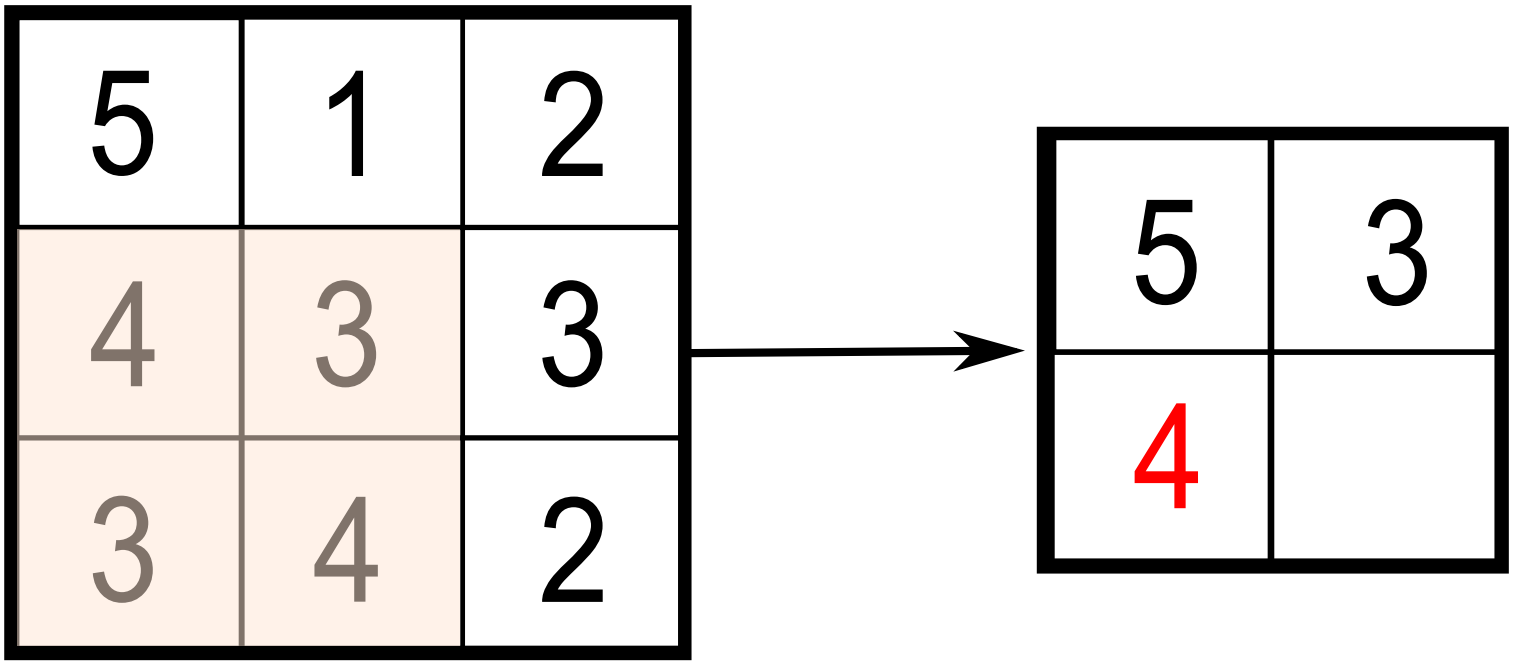
Camadas de Pooling



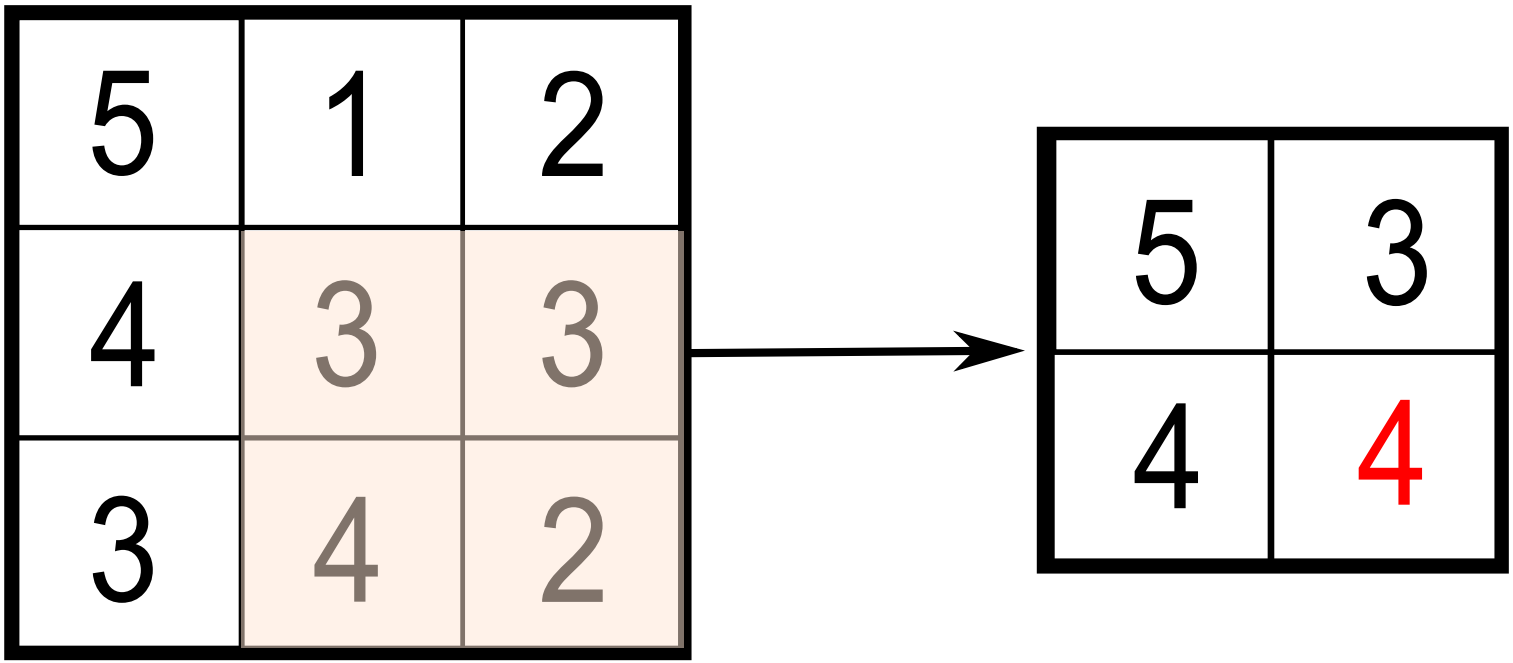
Camadas de Pooling



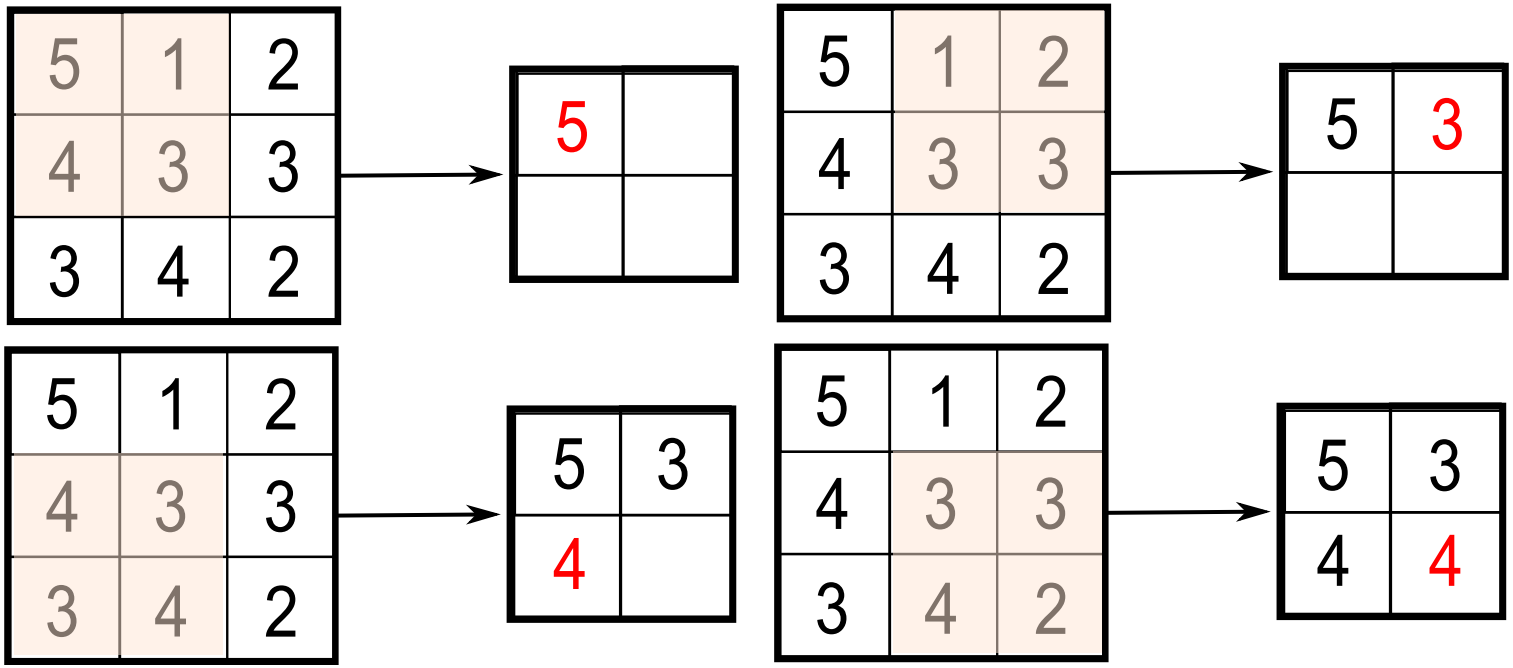
Camadas de Pooling



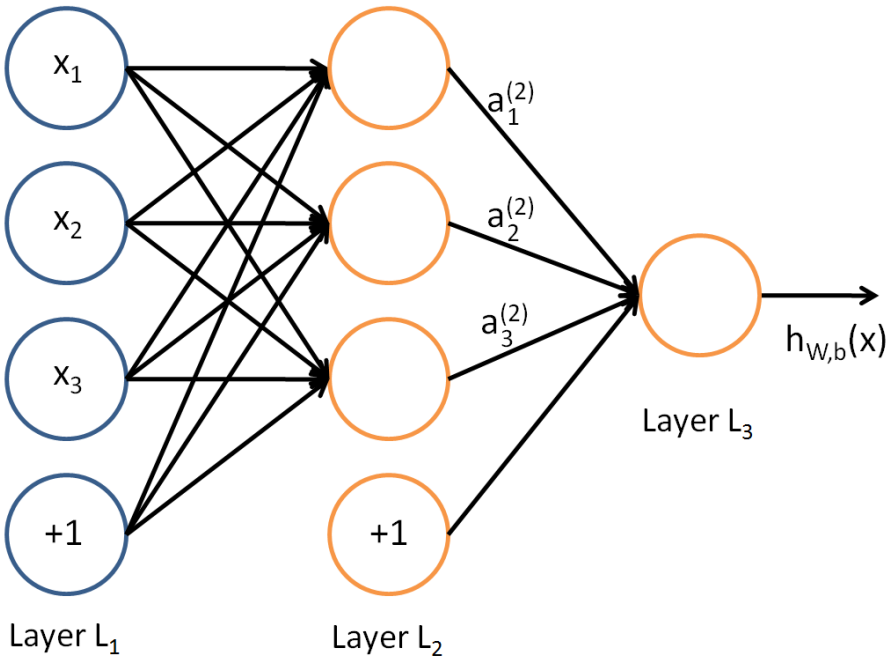
Camadas de Pooling



Camadas de Pooling



Camadas Totalmente Conectadas (MLPs)



Arquitetura Padrão de uma CNN

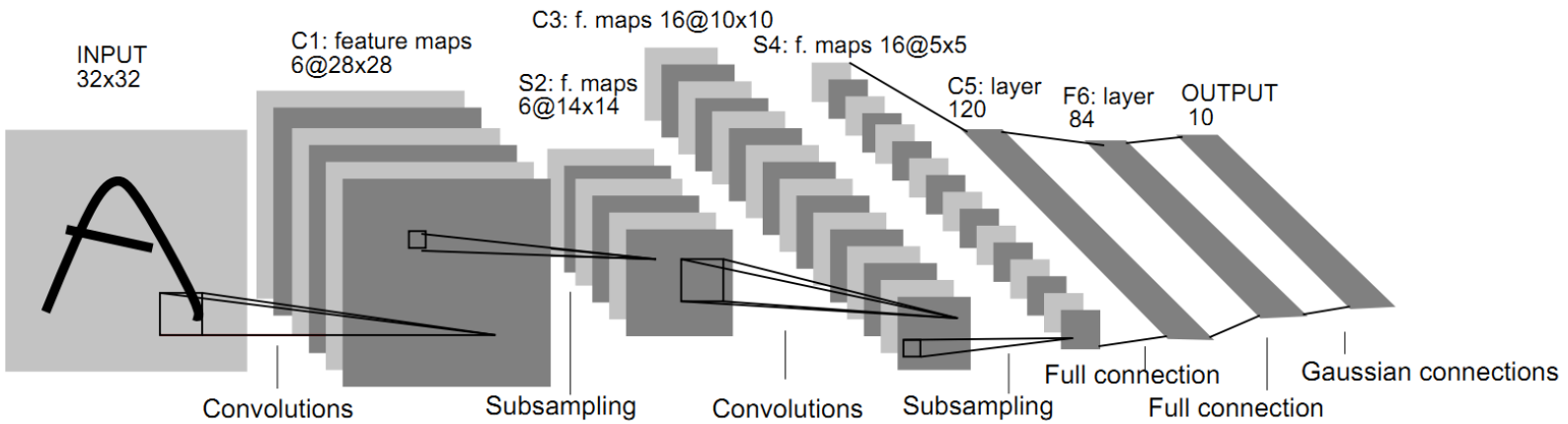


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Figura: Arquitetura da LeNet [8].

Procedimento de Treinamento de CNNs

Exercício Prático - Training Procedure

Training_Procedure.ipynb

[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.

[7] John Duchi, Elad Hazan, and Yoram Singer.
Adaptive subgradient methods for online learning and stochastic optimization.
Journal of Machine Learning Research, 12(Jul):2121–2159, 2011.

[8] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner.
Gradient-based learning applied to document recognition.
Proceedings of the IEEE, 86(11):2278–2324, 1998.

[9] Andreea Salinca.
Convolutional neural networks for sentiment classification on business reviews.
arXiv preprint arXiv:1710.05978, 2017.

[10] Ming Zeng, Le T Nguyen, Bo Yu, Ole J Mengshoel, Jiang Zhu, Pang Wu, and Joy Zhang.
Convolutional neural networks for human activity recognition using mobile sensors.
In Mobile Computing, Applications and Services (MobiCASE), 2014 6th International Conference on, pages 197–205. IEEE, 2014.

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton.

Imagenet classification with deep convolutional neural networks.
In Advances in neural information processing systems, pages 1097–1105, 2012.

[12] Jefkine.
Backpropagation in convolutional neural networks, 2016.