

Primeiros passos de Deep Learning em Python

Antonio Abello

<https://github.com/Abello966>

Felipe Salvatore

<https://felipessalvatore.github.io/>

October 6, 2017

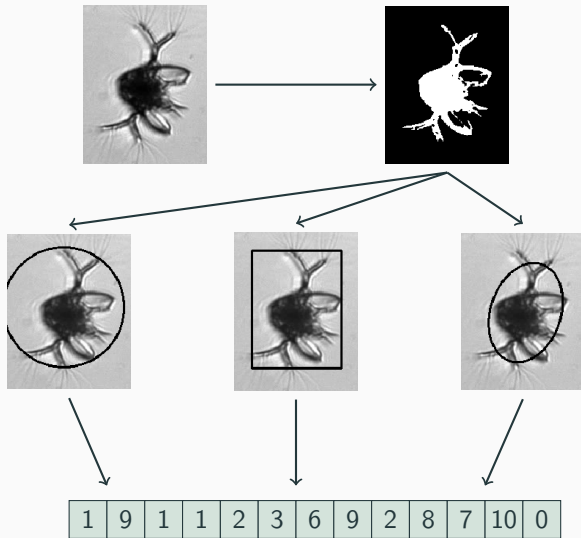
IME-USP: Instituto de Matemática e Estatística, Universidade de São Paulo

Introdução

Por que Deep Learning?

- **Engenharia de Features** (**Feature Engineering**): como extrair informação relevante dos dados?
- **Representação de features**: como representar a informação para facilitar as tarefas?

Feature Engineering - Imagens



- Conhecimento especializado
- Geralmente não aplicável a diferentes domínios
- Difícil de avaliar
- "Humano, demasiado humano"

Espaço de representação - Problemas

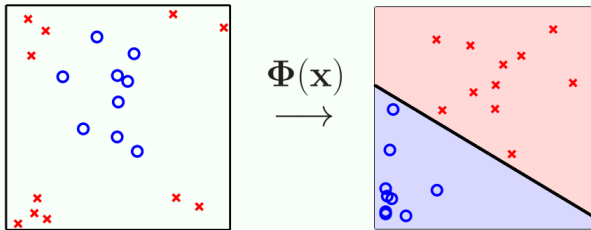


Figure 1: Transformação Φ possibilita classificação por classificador linear [1]

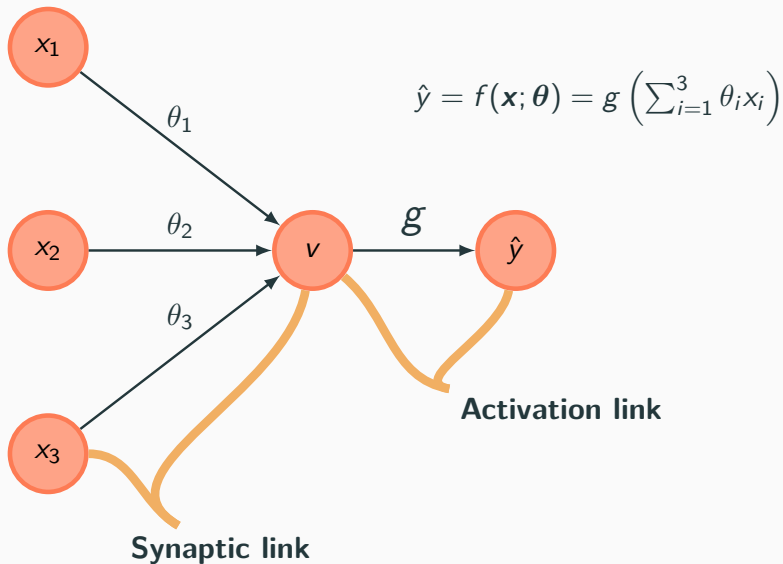
- Relação entre features: compor diferentes features matematicamente leva a informações úteis não aproveitadas
- Transformações não-lineares: podem ajudar, mas como escolher?

"The solution is to allow computers to **learn from experience** and understand the world in terms of a hierarchy of concepts, with each concept defined through its relation to simpler concepts. (...) This approach **avoids the need for human operators to formally specify** all the knowledge that the computer needs. (...) If we draw a graph showing how these concepts are built on top of each other, the **graph is deep**, with many layers. For this reason, we call this approach **deep learning**."

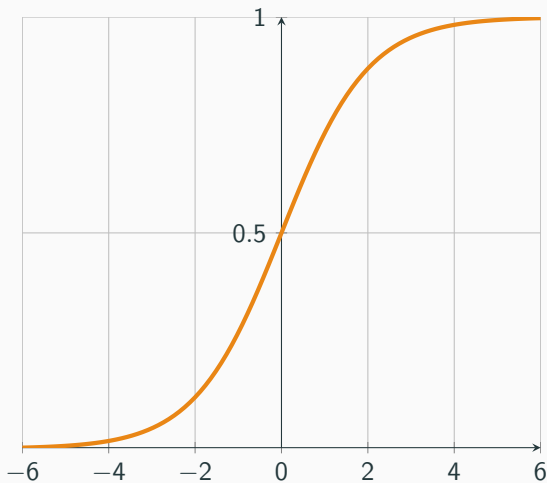
I. Goodfellow, Y. Bengio, A. Courville – **Deep Learning** (2017)

Representação gráfica

Perceptron

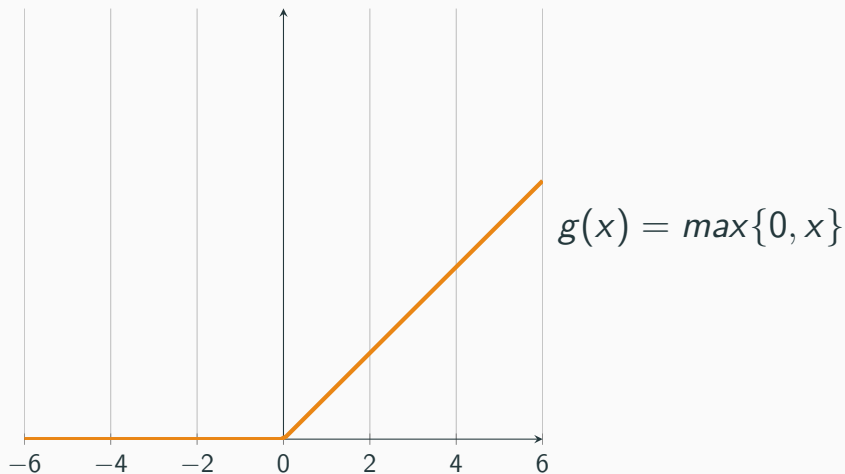


Revisão: função sigmoide



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

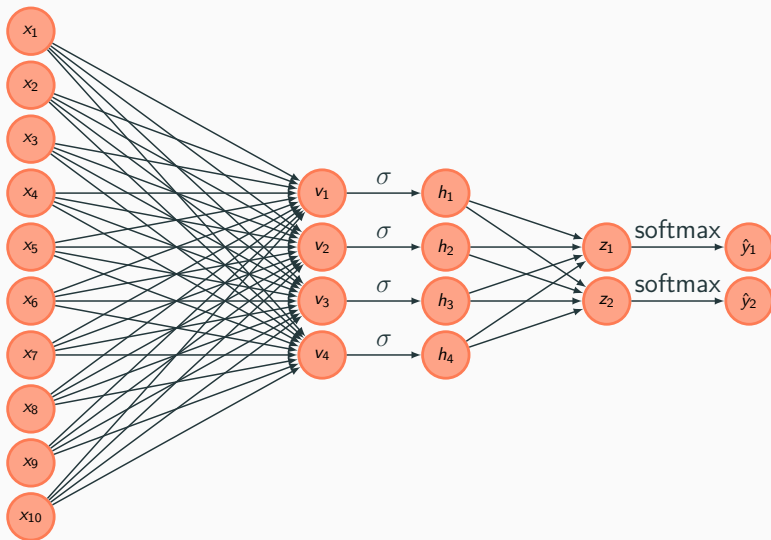
ReLU: Rectified Linear Units



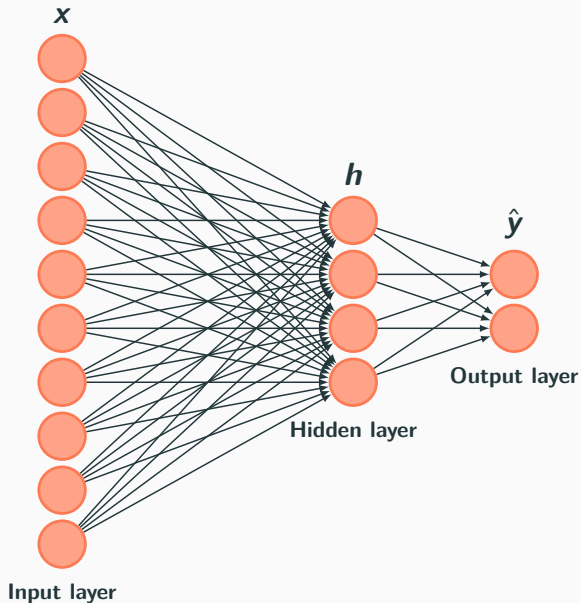
$$\begin{bmatrix} 3.82 \\ 5.35 \\ 1.44 \\ -1.26 \\ 2.71 \\ 1.98 \end{bmatrix} \xrightarrow{\text{softmax}} \begin{bmatrix} 0.16115195 \\ 0.74422819 \\ 0.01491471 \\ 0.00100235 \\ 0.05310907 \\ 0.02559374 \end{bmatrix}$$

$$\text{softmax}(x) = \frac{e^x}{\sum e^x}$$

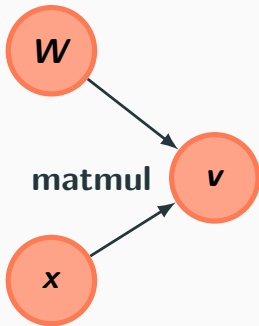
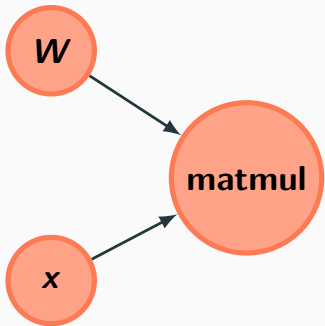
Rede neural: versão antiga



Rede neural: versão antiga

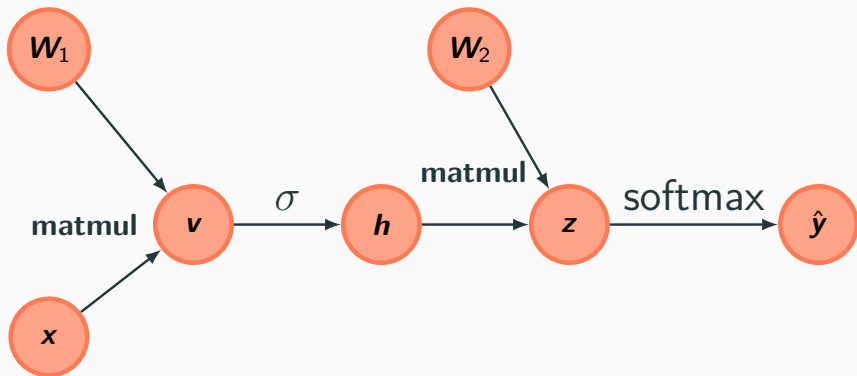


Grafo de computação



$$v = Wx$$

Grafo de computação



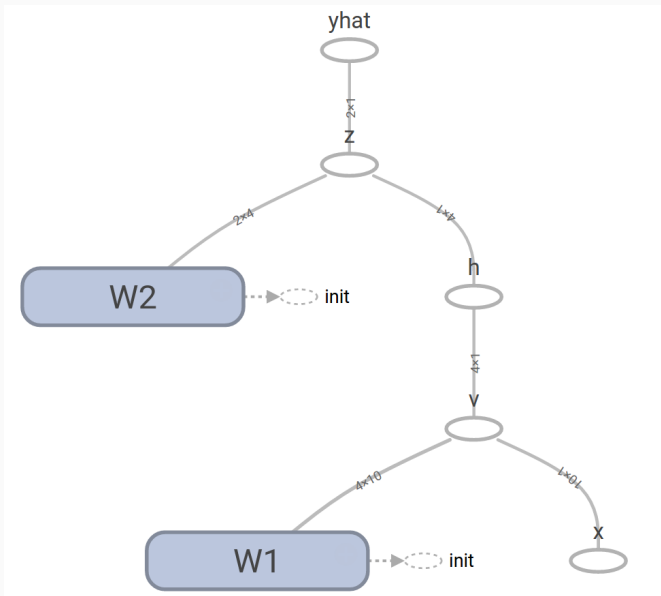
Grafo no Tensorflow

```
1  import tensorflow as tf
2  import numpy as np
3
4  input_shape = [10,1]
5  input_to_hidden_shape = [4,10]
6  hidden_to_output_shape = [2,4]
7
8  W1init = np.zeros(input_to_hidden_shape,
9                    dtype="float32")
10 W2init = np.zeros(hidden_to_output_shape,
11                   dtype="float32")
```

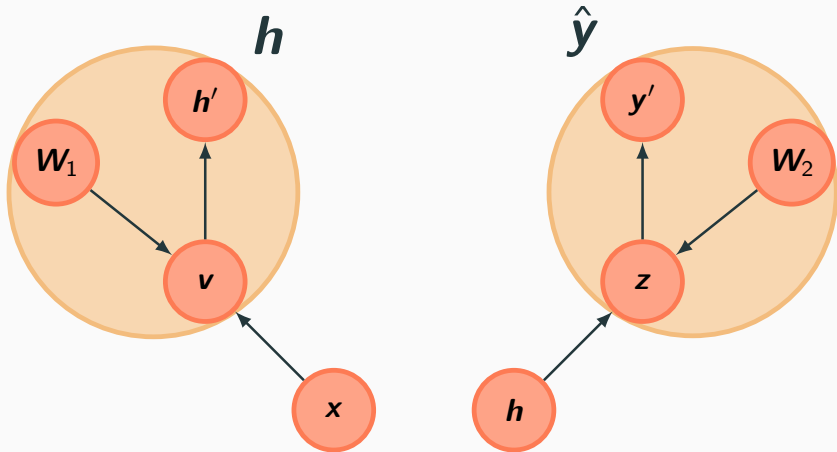
Grafo no Tensorflow

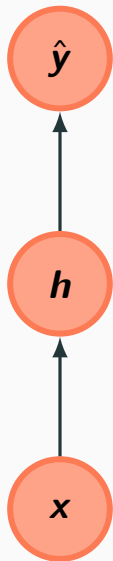
```
1 graph = tf.Graph()
2 with graph.as_default():
3     x = tf.placeholder(shape=input_shape,
4                        dtype="float32")
5     W1 = tf.get_variable(initializer=W1init)
6     v = tf.matmul(W1, x)
7     h = tf.sigmoid(v)
8     W2 = tf.get_variable(initializer=W2init)
9     z = tf.matmul(W2, h)
10    yhat = tf.nn.softmax(z)
```

Visualizando o grafo no Tensorboard

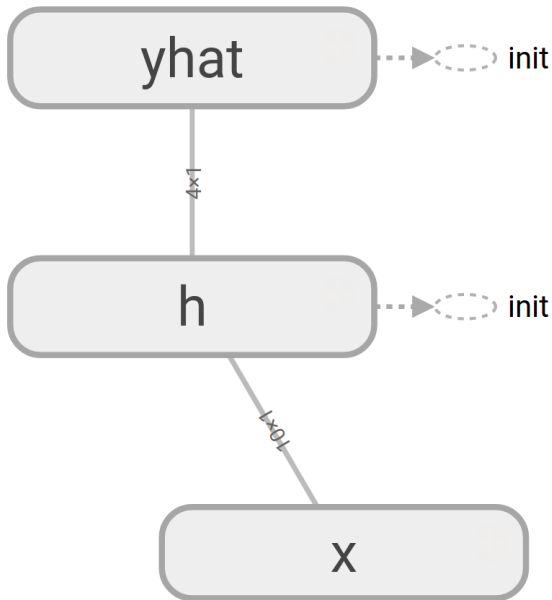


Grafo de computação

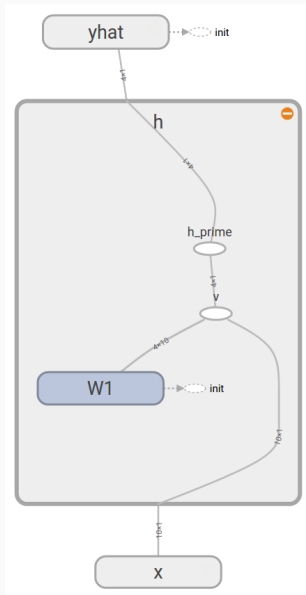




Visualizando o grafo no Tensorboard



Visualizando o grafo no Tensorboard



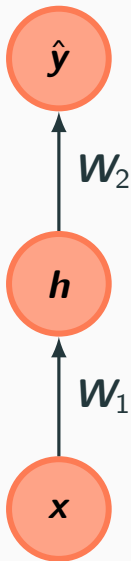
DFN - Deep Feedforward Networks

Deep Feedforward Networks

Deep Feedforward Networks são também chamadas de feedforward neural networks, multilayer perceptrons, ou, em bom português, redes neurais.

Uma rede neural $\hat{y} = f(\mathbf{x}; \theta)$ é um modelo parametrizado de aprendizado de máquina. Esse modelo pode ser descrito como uma composição de funções, por exemplo:

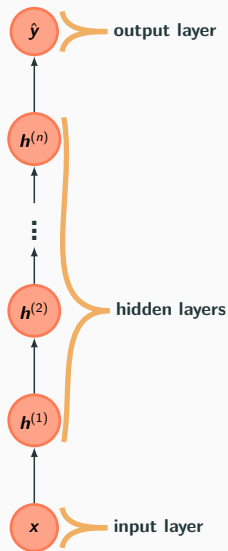
$$f(\mathbf{x}; \theta) = f^{(2)}(f^{(1)}(\mathbf{x}; \theta); \theta)$$



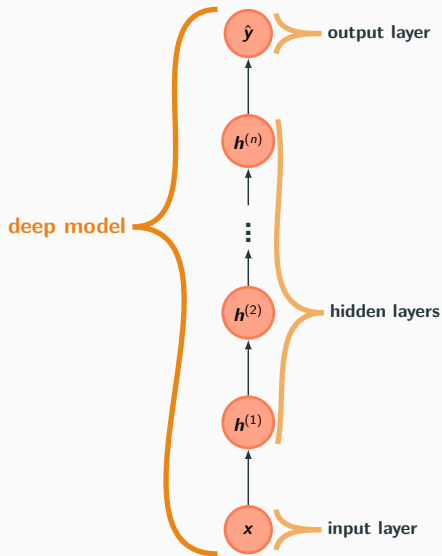
$$\hat{y} = f^{(2)}(f^{(1)}(x; W_1); W_2)$$

$$\hat{y} = \text{softmax}(W_2(\sigma(W_1 x)))$$

Rede neural profunda



Rede neural profunda



Redes neurais são aproximadores universais

Teorema da aproximação universal (**universal approximation theorem**): qualquer função Borel mensurável pode ser aproximada por uma rede neural com pelo menos uma camada escondida, dado que as camadas escondidas tenham um tamanho suficiente.

Toda função definida num subconjunto fechado e limitado de \mathbb{R}^n é Borel mensurável, então um grande número de funções são aproximáveis por uma rede neural.

Redes neurais são aproximadores universais

Uma rede neural com apenas uma camada escondida é suficiente para representar qualquer função.

Mas essa única camada pode ser muito grande e apresentar problemas para generalização. **Empiricamente redes com profundidade maior apresentam melhores resultados para generalização.**

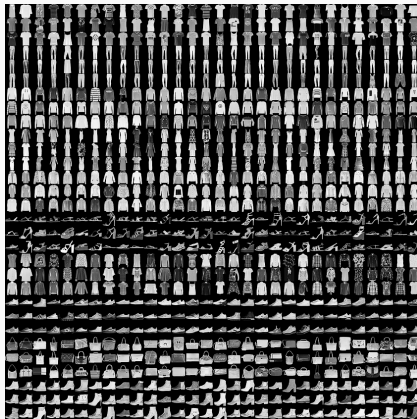
Receita para aprendizado de máquina

- Uma tarefa de aprendizado sobre algum dataset.
- Uma família de modelos.
- Uma função de custo.
- Um algoritmo de otimização.

Receita para aprendizado de máquina

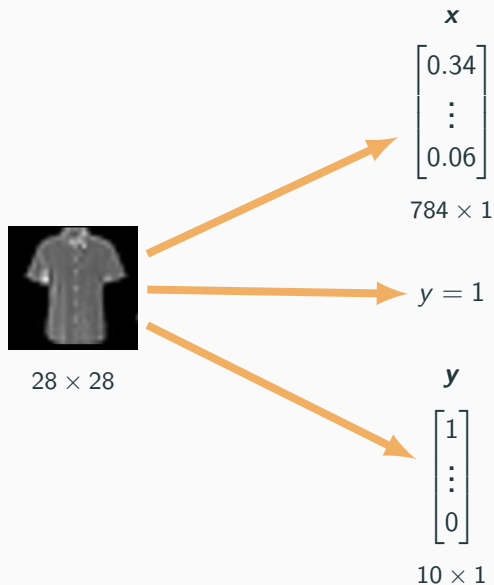
- Uma tarefa de aprendizado sobre algum dataset. **Fashion MNIST**
- Uma família de modelos. **DFN**
- Uma função de custo. **Entropia cruzada**
- Um algoritmo de otimização. **SGD**

Fashion MNIST



- Conjunto de treinamento: 60k
- Conjunto de teste: 10k
- Imagens: 28×28 tons de cinza.
- 10 classes: (camiseta, vestido, sandália, ...)

Fashion MNIST

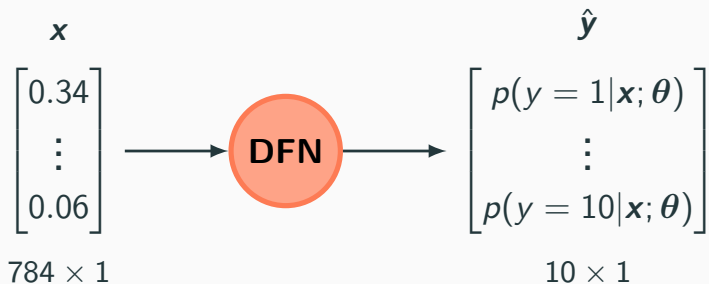


Vamos usar uma DFN para definir a distribuição $p(y|\mathbf{x}; \theta)$.

Os parâmetros θ vão ser adaptados de modo que $p(y|\mathbf{x}; \theta)$ seja a distribuição mais adequada para os dados

$$(\mathbf{x}^{(1)}, y^{(1)}), \dots, (\mathbf{x}^{(N)}, y^{(N)})$$

Classificação com uma rede neural



A função que queremos maximizar é

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) &= \mathbb{E}_{\mathbf{x}, y \sim p_{data}} \log p(y|\mathbf{x}; \boldsymbol{\theta}) \\ &= \frac{1}{N} \sum_{i=1}^N \log p(y^{(i)}|\mathbf{x}^{(i)}; \boldsymbol{\theta})\end{aligned}$$

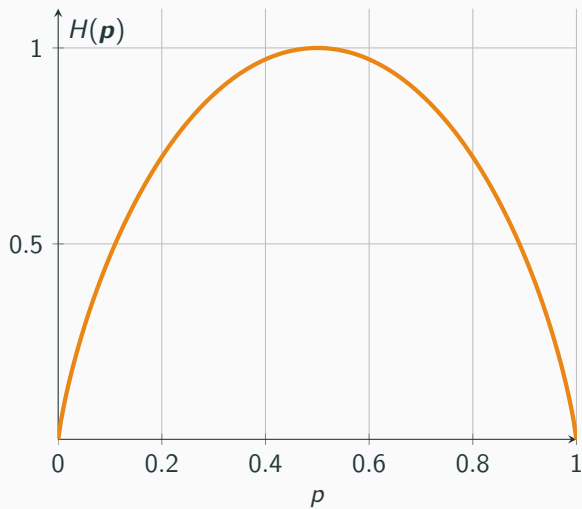
$$\mathbf{p} \quad \begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix}$$

$$\mathbf{q} \quad \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

$$H(\mathbf{p}) = 0.72 \quad H(\mathbf{q}) = 1$$

$$H(\mathbf{p}) = \sum_i \mathbf{p}_i \log \frac{1}{\mathbf{p}_i}$$

Revisão: entropia



$$\begin{matrix} p \\ \begin{bmatrix} p \\ 1 - p \end{bmatrix} \end{matrix}$$

Revisão: divergência Kullback-Leibler

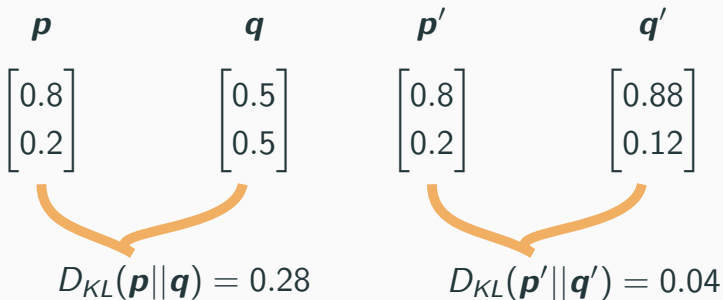


Diagram illustrating the Kullback-Leibler (KL) divergence calculation for two pairs of probability distributions.

For the first pair, p and q :

$$p = \begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix}, \quad q = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$
$$D_{KL}(p||q) = 0.28$$

For the second pair, p' and q' :

$$p' = \begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix}, \quad q' = \begin{bmatrix} 0.88 \\ 0.12 \end{bmatrix}$$
$$D_{KL}(p'||q') = 0.04$$

$$D_{KL}(p||q) = \sum_i p_i \log \frac{p_i}{q_i}$$

$$\begin{aligned}CE(\mathbf{p}, \mathbf{q}) &= H(\mathbf{p}) + D_{KL}(\mathbf{p}||\mathbf{q}) \\ &= -\sum_i \mathbf{p}_i \log(\mathbf{q}_i)\end{aligned}$$

$$\arg \min_{\mathbf{q}} CE(\mathbf{p}, \mathbf{q}) = \arg \min_{\mathbf{q}} D_{KL}(\mathbf{p}, \mathbf{q})$$

$$\begin{aligned} L(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}) &= CE(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}) \\ &= - \sum_{k=1}^{10} \mathbf{y}_k^{(i)} \log p(y = k | \mathbf{x}^{(i)}; \boldsymbol{\theta}) \\ &= - \log p(\mathbf{y}^{(i)} | \mathbf{x}^{(i)}; \boldsymbol{\theta}) \end{aligned}$$

Entropia cruzada e verossimilhança

E a função que queremos minimizar é

$$\begin{aligned} J(\boldsymbol{\theta}) &= \frac{1}{N} \sum_{i=1}^N L(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}) \\ &= -\frac{1}{N} \sum_{i=1}^N \log p(y^{(i)} | \mathbf{x}^{(i)}; \boldsymbol{\theta}) \\ &= -\mathcal{L}(\boldsymbol{\theta}) \end{aligned}$$

$$\arg \max_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \arg \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

Seja $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Chamamos de descida do gradiente (**gradient descent**) o método para minimizar f .

$$\mathbf{x}^{novo} = \mathbf{x}^{velho} - \alpha \nabla_{\mathbf{x}} f(\mathbf{x})$$

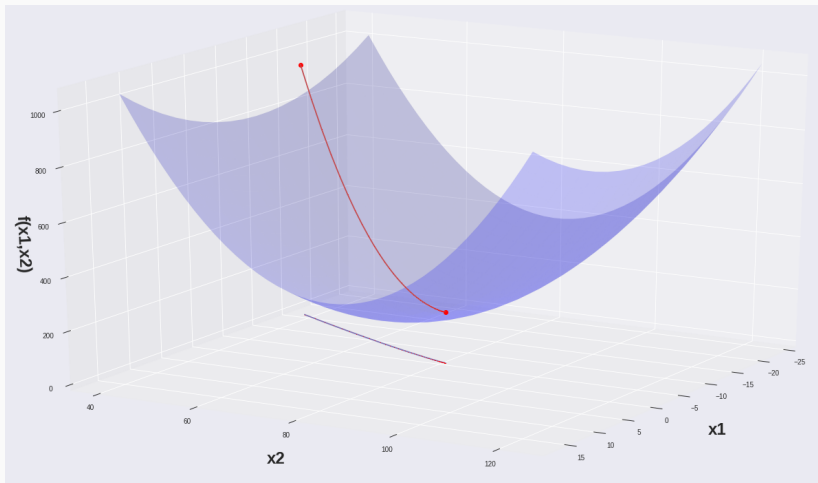
.

Seja $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Chamamos de descida do gradiente (**gradient descent**) o método para minimizar f .

$$\mathbf{x}^{novo} = \mathbf{x}^{velho} - \alpha \nabla_{\mathbf{x}} f(\mathbf{x})$$

α é chamado de taxa de aprendizado (**learning rate**).

Descida do gradiente



Na tarefa de aprendizado temos:

$$J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^N L(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)})$$

$$\boldsymbol{\theta}^{novo} = \boldsymbol{\theta}^{velho} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

Descida do gradiente estocástica

Na prática nunca lidamos com todos os dados pois N pode ser muito grande (por exemplo, 60k).

Uma alternativa é o algoritmo de **descida do gradiente estocástica** (**stochastic gradient descent**):

- Escolhemos uma observação arbitrária $(\mathbf{x}^{(i)}, y^{(i)})$.
- $J(\theta) = L(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)})$
- $\theta^{novo} = \theta^{velho} - \alpha \nabla_{\theta} J(\theta)$

Descida do gradiente em lote

Uma variação desse algoritmo é conhecido como **descida do gradiente em lote** (**mini-batch gradient descent**):

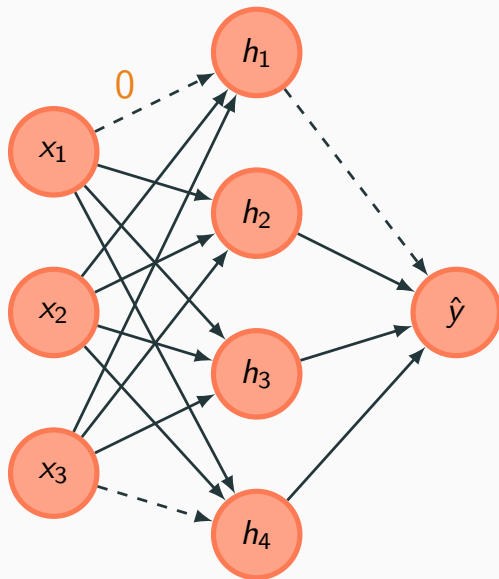
- Escolhemos um conjunto de m ($m \ll N$) observações arbitrárias $(\mathbf{x}^{(1)}, y^{(1)}), \dots, (\mathbf{x}^{(m)}, y^{(m)})$.
- $J(\theta) = \frac{1}{m} \sum_{i=1}^m L(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)})$
- $\theta^{novo} = \theta^{velho} - \alpha \nabla_{\theta} J(\theta)$

Quando se processa todos (ou um número equivalente de) exemplos se diz que se passou uma **época** (**epoch**).

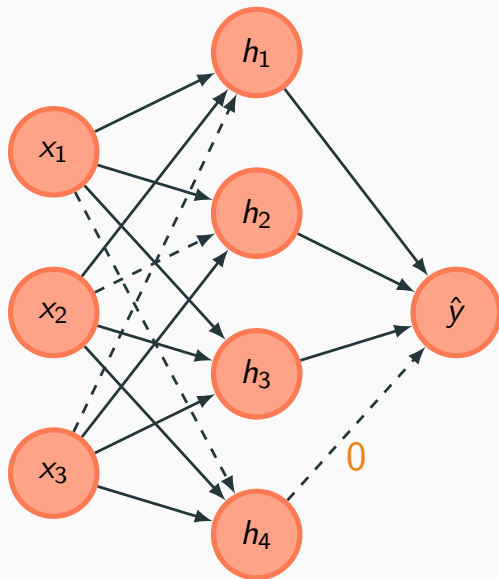
Estratégias para impor a navalha de Occam. Elas melhoram a performance no teste às custas de uma piora no treino.

- Penalizações à Norma do Parâmetro (L_2 , L_1)
- Early Stopping
- Dropout

Dropout



Dropout



Back Propagation

- $\hat{\mathbf{y}} = f(\mathbf{x}; \boldsymbol{\theta})$
- $J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m L(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)})$
- **SGD:**

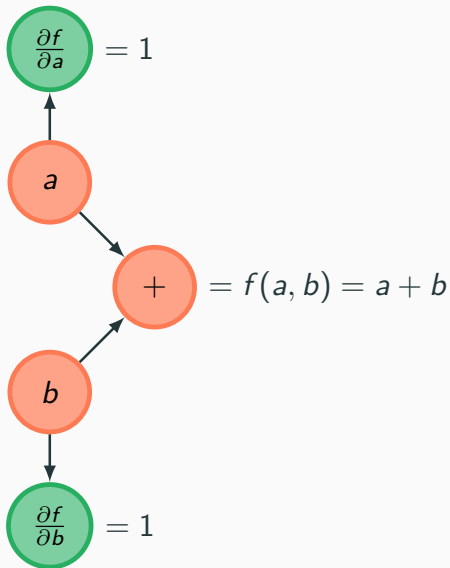
$$\boldsymbol{\theta}^{novo} = \boldsymbol{\theta}^{velho} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

- $\hat{\mathbf{y}} = f(\mathbf{x}; \boldsymbol{\theta})$
- $J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m L(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)})$
- **SGD:**

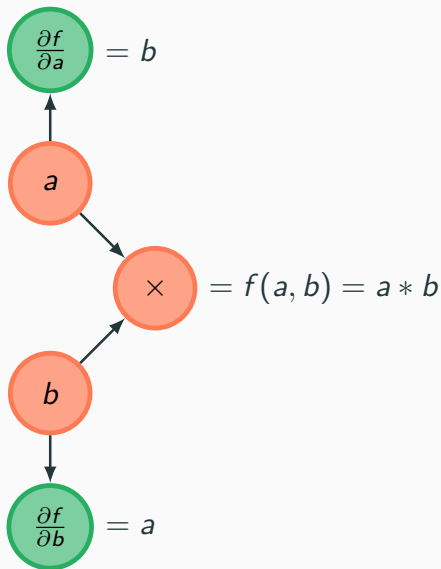
$$\boldsymbol{\theta}^{novo} = \boldsymbol{\theta}^{velho} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

- Retro propagação (back propagation) é um modo de computar $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ usando aplicações recursivas da **regra da cadeia**.

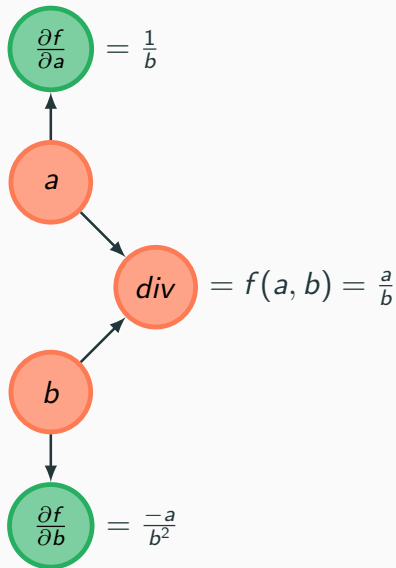
Operações simples: soma



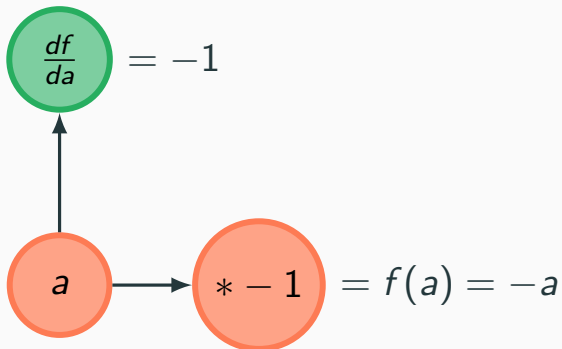
Operações simples: multiplicação



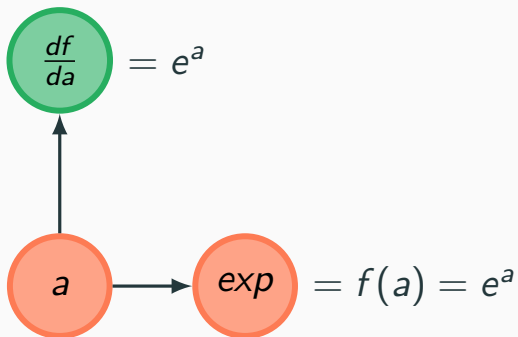
Operações simples: divisão



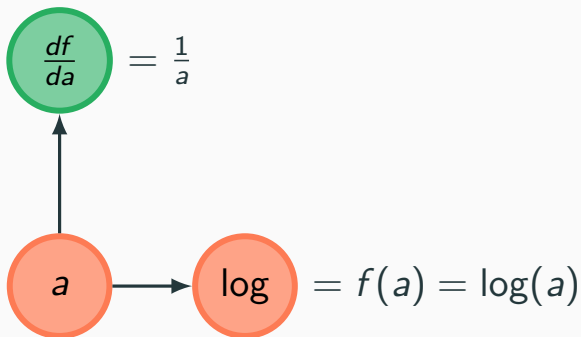
Operações simples: negativo



Operações simples: exponenciação



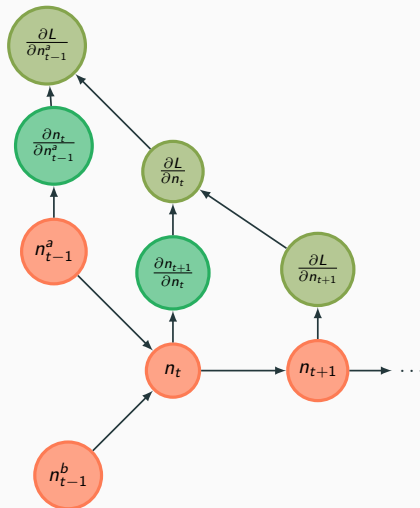
Operações simples: logarítimo



- $f : \mathbb{R} \rightarrow \mathbb{R}, g : \mathbb{R} \rightarrow \mathbb{R}.$
- $y = g(x)$
- $z = f(g(x)) = f(y)$

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$$

Aplicando a regra da cadeia

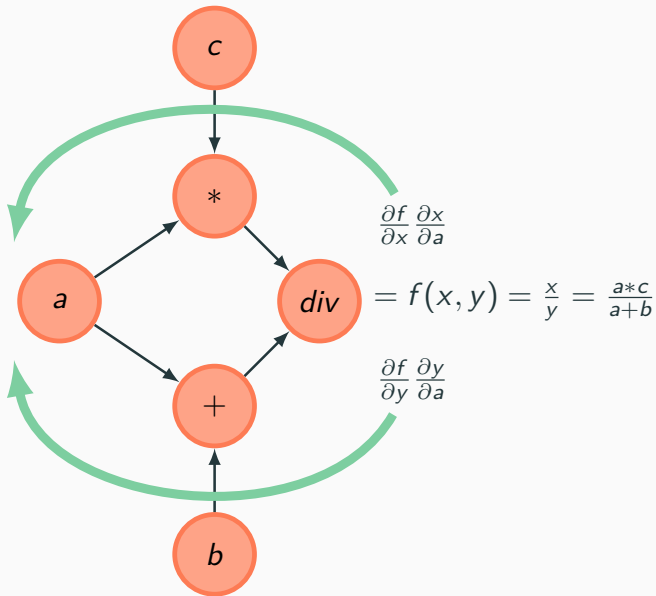


Regra da cadeia para várias variáveis

- $z = f(x, y)$
- $x = f_1(a)$.
- $y = f_2(a)$

$$\frac{\partial z}{\partial a} = \frac{\partial z}{\partial x} \frac{\partial x}{\partial a} + \frac{\partial z}{\partial y} \frac{\partial y}{\partial a}$$

Exemplo



Exemplo: regressão logística

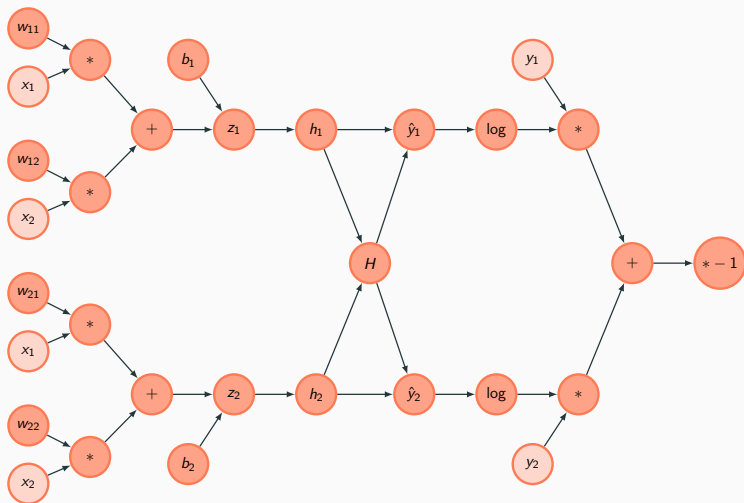
$$\hat{\mathbf{y}} = \textit{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$L(\mathbf{y}, \hat{\mathbf{y}}) = CE(\mathbf{y}, \hat{\mathbf{y}})$$

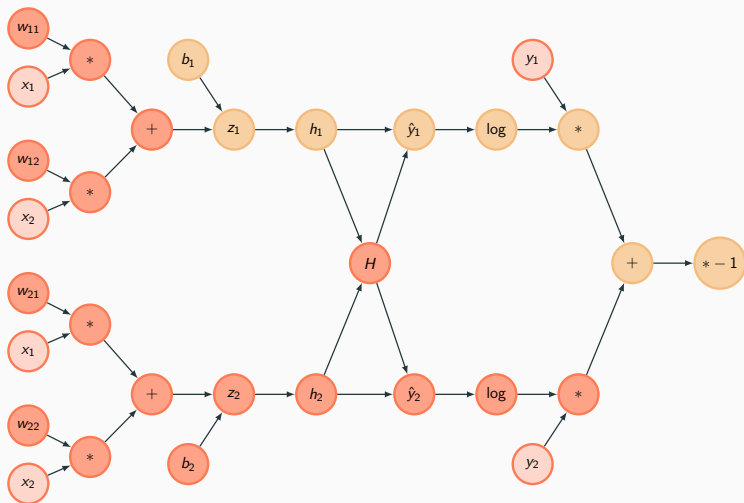
$$L(\mathbf{y}, \hat{\mathbf{y}}) = - \sum_i \mathbf{y}_i \log \left(\frac{\exp(\sum_k \mathbf{W}_{i,k} \mathbf{x}_k + \mathbf{b}_i)}{\sum_j \exp(\sum_k \mathbf{W}_{j,k} \mathbf{x}_k + \mathbf{b}_j)} \right)$$

- $$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$
- $$\begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = \begin{bmatrix} \exp(z_1) \\ \exp(z_2) \end{bmatrix}$$
- $$H = h_1 + h_2$$
- $$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \end{bmatrix} = \begin{bmatrix} \frac{h_1}{H} \\ \frac{h_2}{H} \end{bmatrix}$$

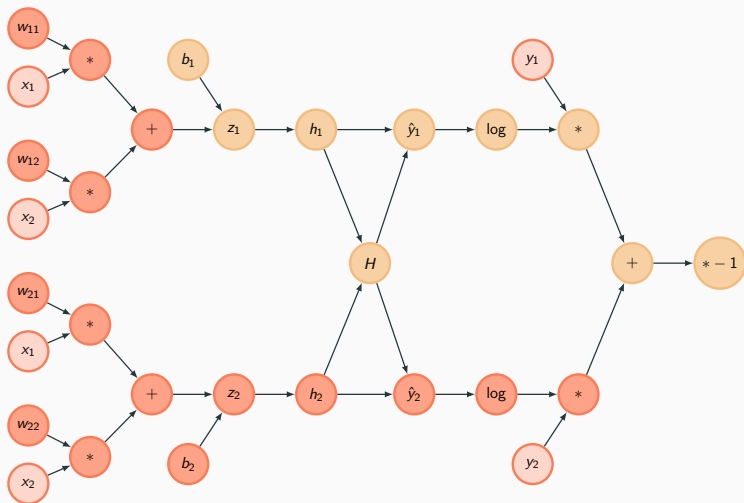
Grafo de $L(\hat{y}, y)$



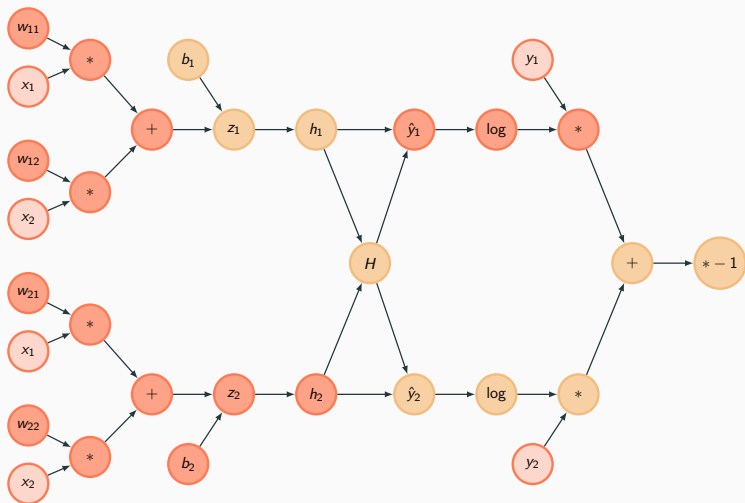
Caminho de b_1 : 1



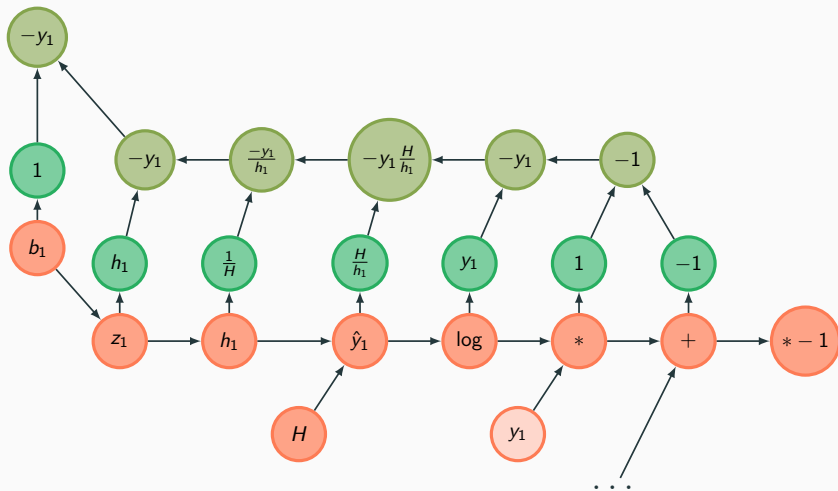
Caminho de b_1 : 2



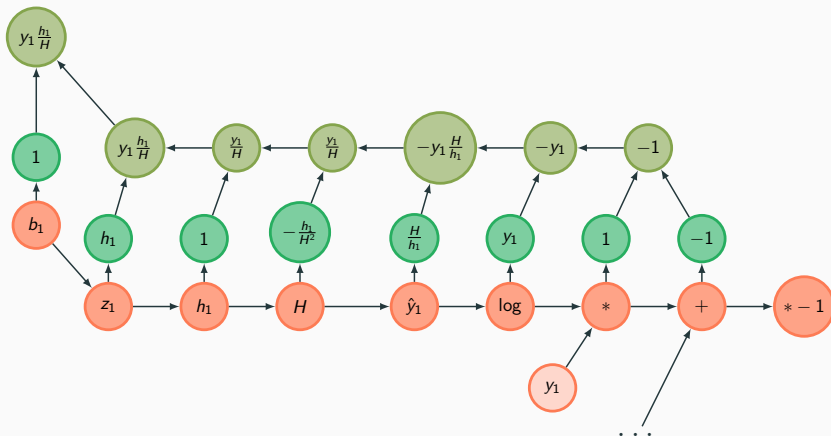
Caminho de b_1 : 3



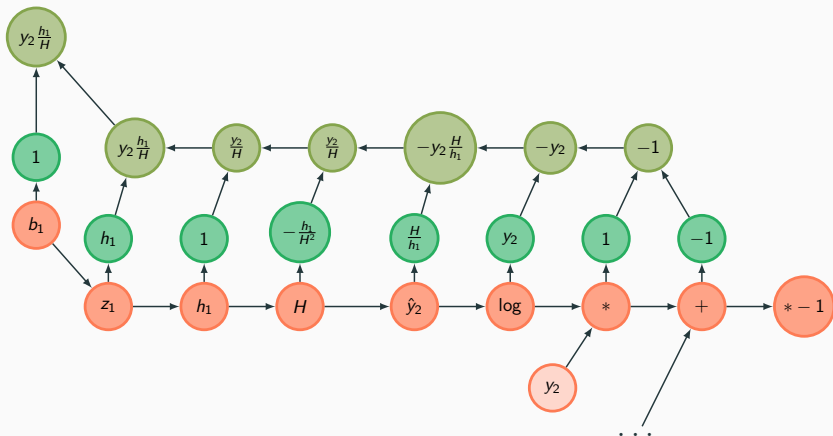
Derivada parcial de L com respeito a b_1 : 1



Derivada parcial de L com respeito a b_1 : 2



Derivada parcial de L com respeito a b_1 : 3



Derivada parcial de L com respeito a b_1

$$\frac{\partial L}{\partial b_1} = -y_1 + y_1 \frac{h_1}{H} + y_2 \frac{h_1}{H}$$

Derivada parcial de L com respeito a b_1

$$\begin{aligned}\frac{\partial L}{\partial b_1} &= -y_1 + y_1 \frac{h_1}{H} + y_2 \frac{h_1}{H} \\ &= y_1 \left(\frac{h_1}{H} - 1 \right) + y_2 \left(\frac{h_1}{H} - 0 \right)\end{aligned}$$

Derivada parcial de L com respeito a b_1

$$\begin{aligned}\frac{\partial L}{\partial b_1} &= -y_1 + y_1 \frac{h_1}{H} + y_2 \frac{h_1}{H} \\ &= y_1 \left(\frac{h_1}{H} - 1 \right) + y_2 \left(\frac{h_1}{H} - 0 \right) \\ &= y_1 (\hat{y}_1 - 1) + y_2 (\hat{y}_1 - 0)\end{aligned}$$

Derivada parcial de L com respeito a b_1

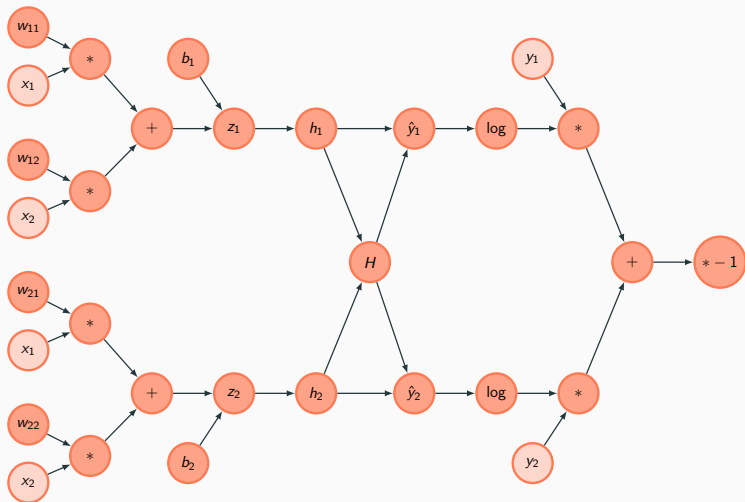
$$\begin{aligned}\frac{\partial L}{\partial b_1} &= -y_1 + y_1 \frac{h_1}{H} + y_2 \frac{h_1}{H} \\&= y_1 \left(\frac{h_1}{H} - 1 \right) + y_2 \left(\frac{h_1}{H} - 0 \right) \\&= y_1 (\hat{y}_1 - 1) + y_2 (\hat{y}_1 - 0) \\&= \hat{y}_1 - y_1 \quad (\text{quando } y \text{ é um vetor one-hot})\end{aligned}$$

Exemplo

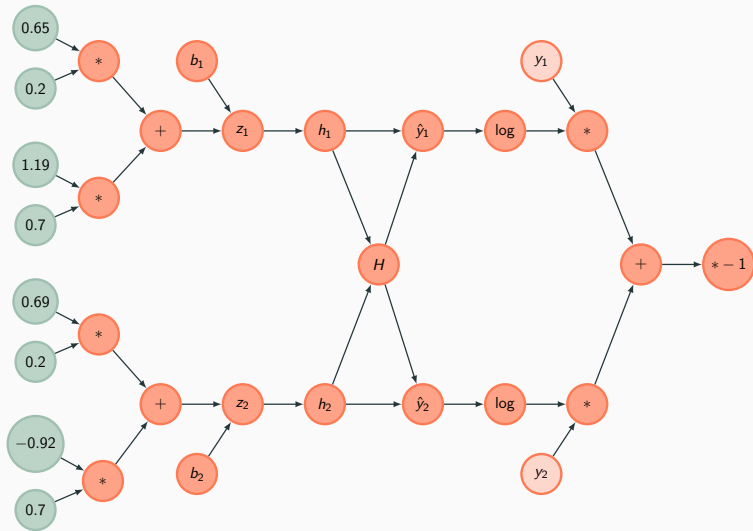
$$\mathbf{W} = \begin{bmatrix} 0.65 & 1.19 \\ 0.69 & -0.92 \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} 0.2 \\ 0.7 \end{bmatrix}$$

$$\mathbf{b} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

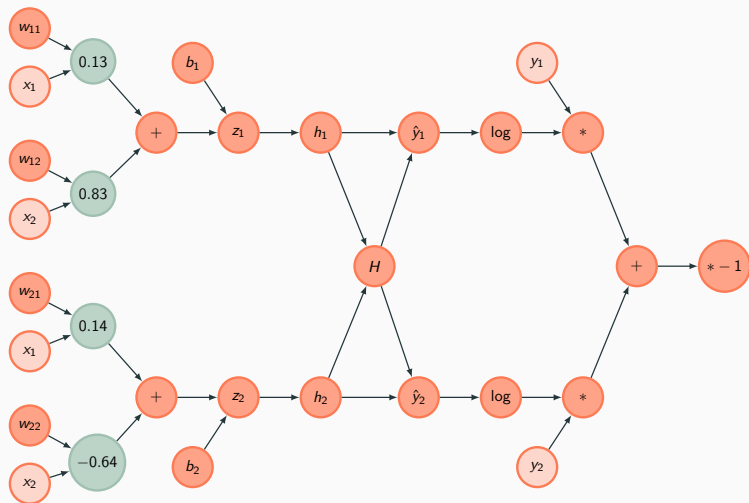
Forward



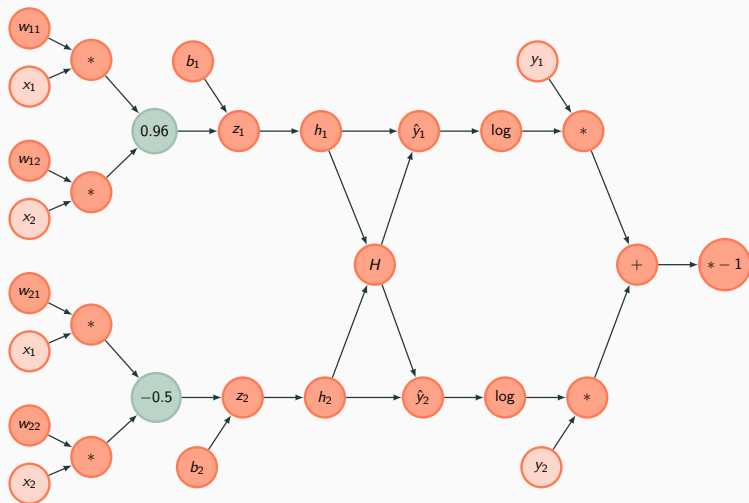
Forward



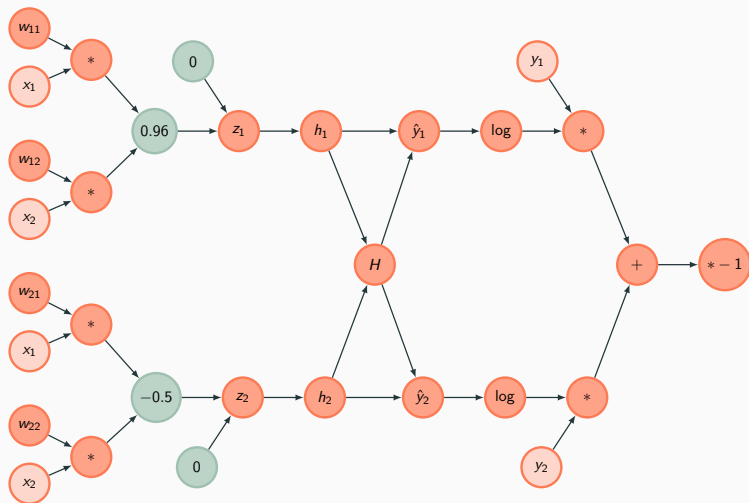
Forward



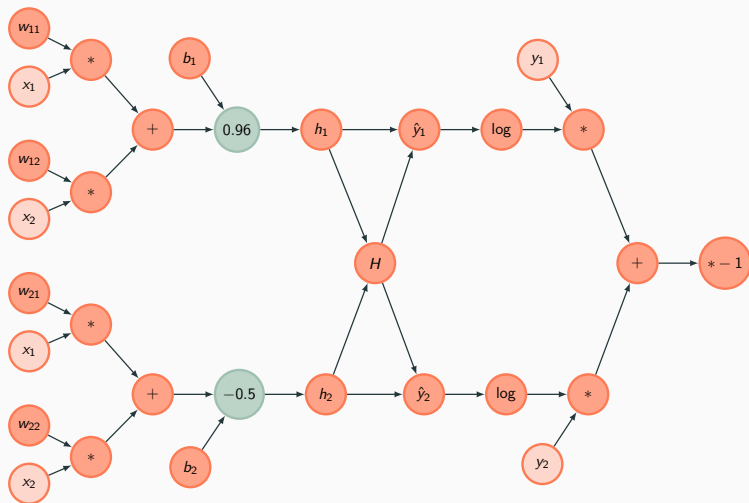
Forward



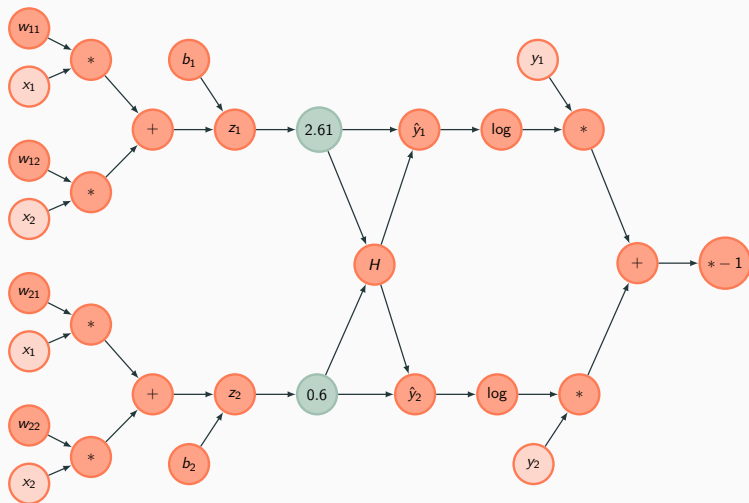
Forward



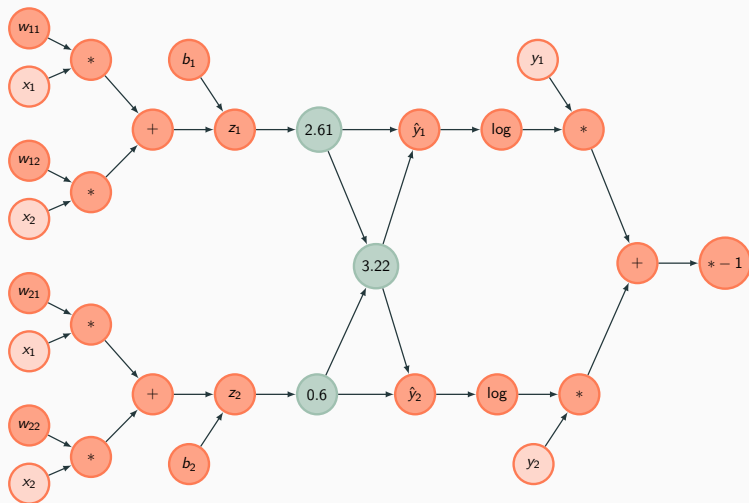
Forward



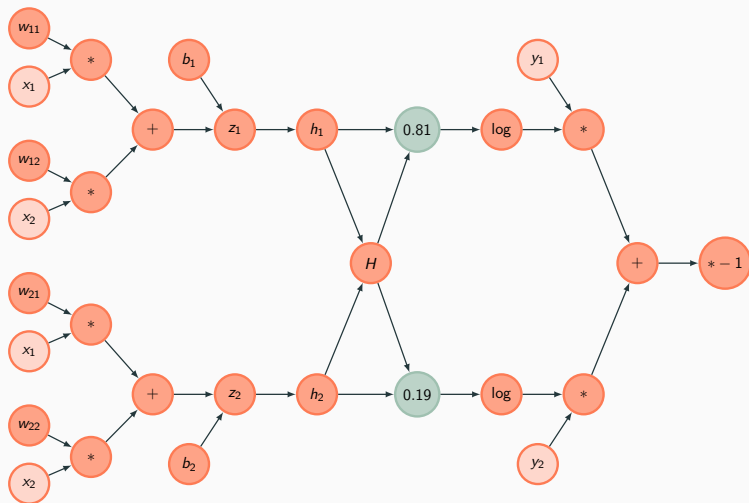
Forward



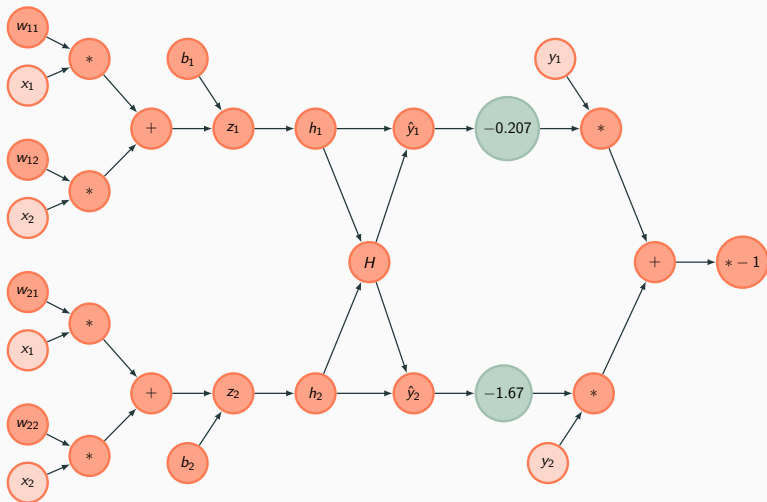
Forward



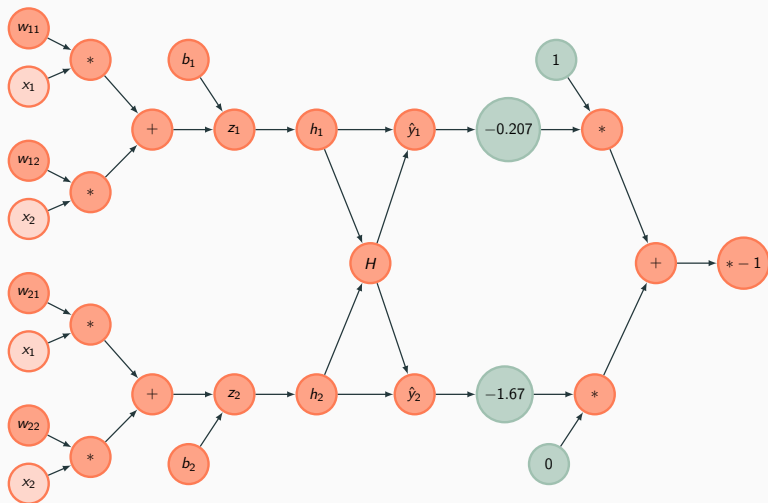
Forward



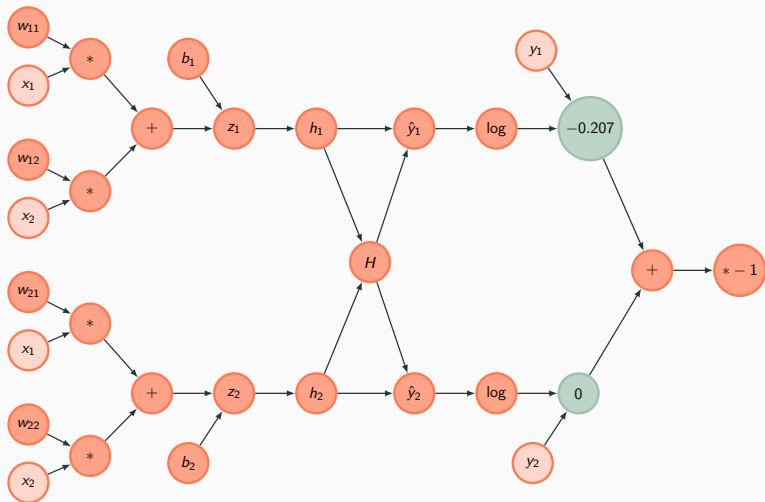
Forward



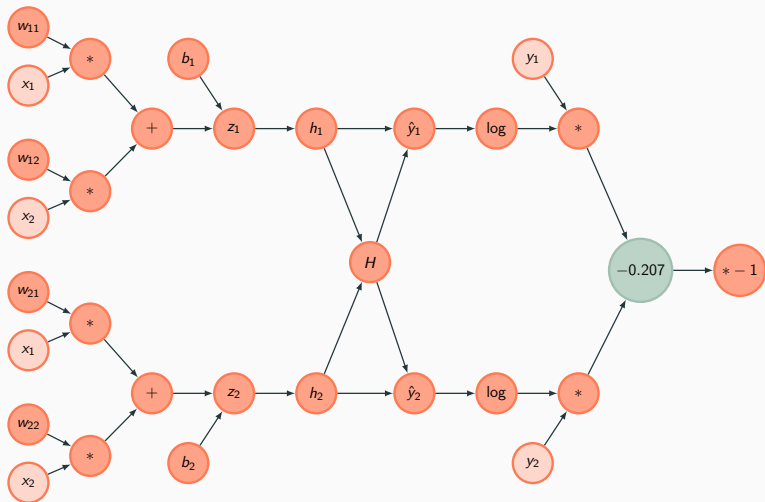
Forward



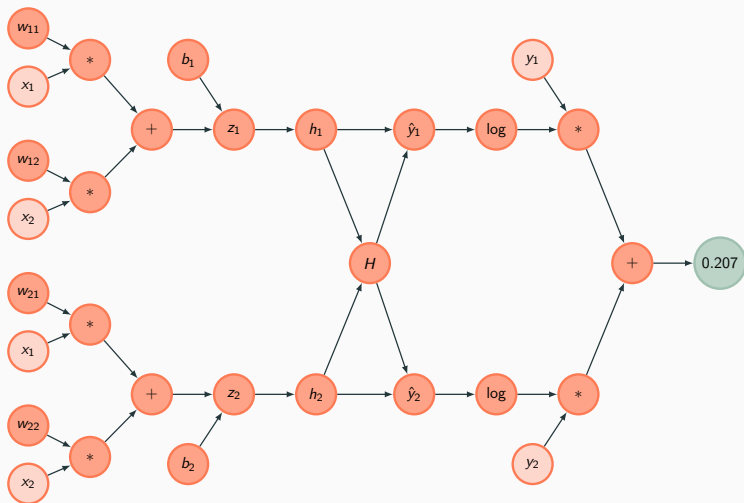
Forward



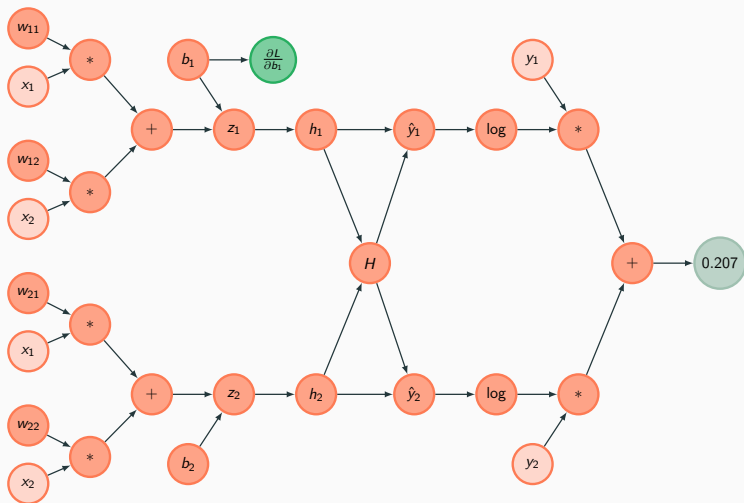
Forward



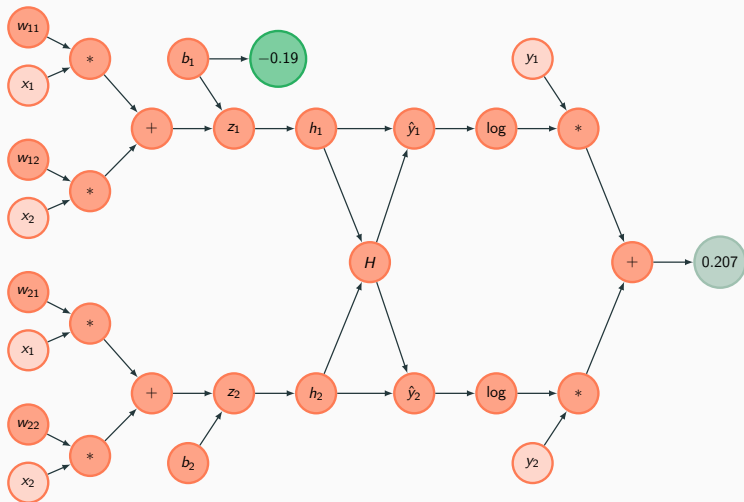
Forward



Forward



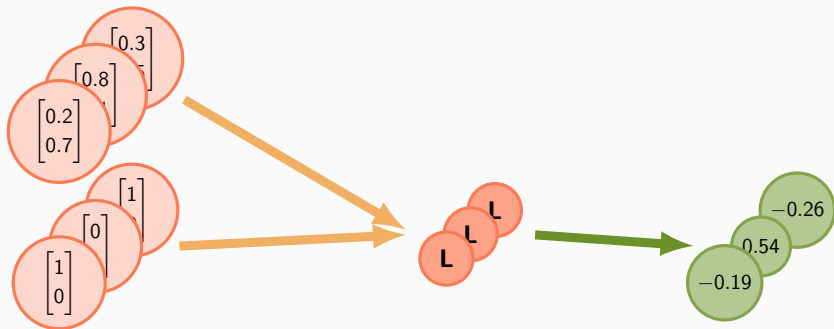
Backward



`batch_size = 3`

\mathbf{x}_1	\mathbf{x}_2	\mathbf{x}_3
$\begin{bmatrix} 0.2 \\ 0.7 \end{bmatrix}$	$\begin{bmatrix} 0.8 \\ 0.1 \end{bmatrix}$	$\begin{bmatrix} 0.3 \\ 0.5 \end{bmatrix}$
\mathbf{y}_1	\mathbf{y}_2	\mathbf{y}_3
$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$

Calulando em lote



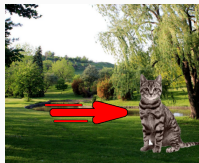
$$b_1^{novo} = b_1^{velho} - \alpha \frac{\partial J}{\partial b_1}$$

$$\begin{aligned}b_1^{novo} &= b_1^{velho} - \alpha \frac{\partial J}{\partial b_1} \\&= 0 - \alpha \frac{1}{3} (-0.19 + 0.54 - 0.26)\end{aligned}$$

$$\begin{aligned}b_1^{novo} &= b_1^{velho} - \alpha \frac{\partial J}{\partial b_1} \\&= 0 - \alpha \frac{1}{3} (-0.19 + 0.54 - 0.26) \\&= 0 - \alpha 0.03\end{aligned}$$

$$\begin{aligned}b_1^{novo} &= b_1^{velho} - \alpha \frac{\partial J}{\partial b_1} \\&= 0 - \alpha \frac{1}{3} (-0.19 + 0.54 - 0.26) \\&= 0 - \alpha 0.03 \\&= -0.003 \quad (\alpha = 0.1)\end{aligned}$$

CNN - Convolutional Neural Networks



- Propriedade interessante de imagens: invariância translacional
- Convolução: operador que passa um filtro/kernel/núcleo pequeno por uma imagem para gerar outra imagem
- Multiplicação de matrizes está para DFN assim como Convolução está para CNN

Aplicando filtros em uma imagem

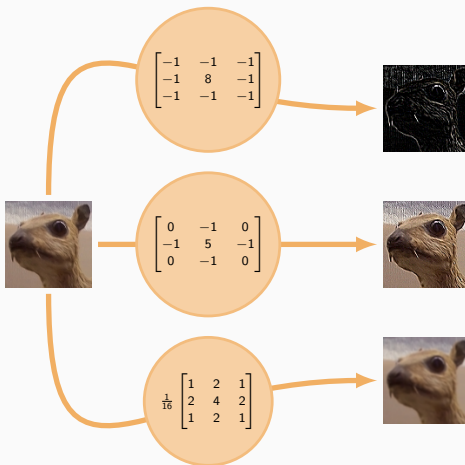


Figure 2: Exemplo de aplicação de filtros em uma imagem (extraído de [https://en.wikipedia.org/wiki/Kernel_\(image_processing\)](https://en.wikipedia.org/wiki/Kernel_(image_processing)))

Exemplo de imagem (retirado de [2])

3	3	2	1	0
0	0	1	3	1
3	1	2	2	3
2	0	0	2	2
2	0	0	0	1

Exemplo de filtro (retirado de [2])

0	1	2
2	2	0
0	1	2

Convolução (retirado de [2])

3_0	3_1	2_2	1	0
0_2	0_2	1_0	3	1
3_0	1_1	2_2	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

Convolução (retirado de [2])

3	3_0	2_1	1_2	0
0	0_2	1_2	3_0	1
3	1_0	2_1	2_2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

Convolução (retirado de [2])

3	3	2 ₀	1 ₁	0 ₂
0	0	1 ₂	3 ₂	1 ₀
3	1	2 ₀	2 ₁	3 ₂
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

Convolução (retirado de [2])

3	3	2	1	0
0_0	0_1	1_2	3	1
3_2	1_2	2_0	2	3
2_0	0_1	0_2	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

Convolução (retirado de [2])

3	3	2	1	0
0	0_0	1_1	3_2	1
3	1_2	2_2	2_0	3
2	0_0	0_1	2_2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

Convolução (retirado de [2])

3	3	2	1	0
0	0	1 ₀	3 ₁	1 ₂
3	1	2 ₂	2 ₂	3 ₀
2	0	0 ₀	2 ₁	2 ₂
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

Convolução (retirado de [2])

3	3	2	1	0
0	0	1	3	1
3_0	1_1	2_2	2	3
2_2	0_2	0_0	2	2
2_0	0_1	0_2	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

Convolução (retirado de [2])

3	3	2	1	0
0	0	1	3	1
3	1_0	2_1	2_2	3
2	0_2	0_2	2_0	2
2	0_0	0_1	0_2	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

Convolução (retirado de [2])

3	3	2	1	0
0	0	1	3	1
3	1	2_0	2_1	3_2
2	0	0_2	2_2	2_0
2	0	0_0	0_1	1_2

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

Feature map (retirado de [2])

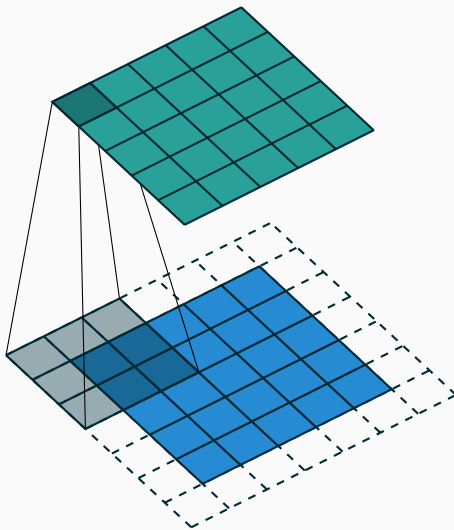
12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

Para calcular o tamanho do feature map nos usamos a equação:

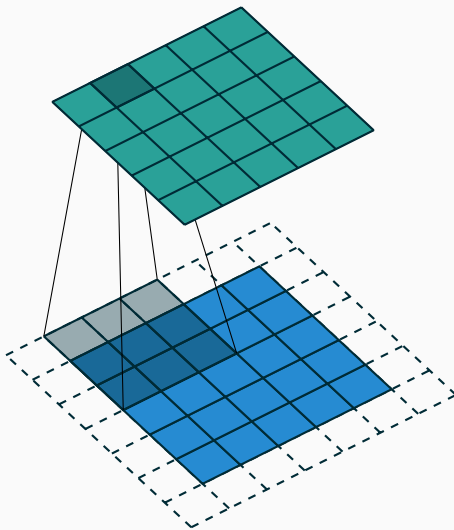
$$o = (i - k) + 1$$

Em que $o \times o$ é o tamanho do feature map, a imagem de entrada tem o tamanho $i \times i$, o tamanho do filtro é $k \times k$, andamos apenas um pixel por vez em cada eixo (**stride** = 1) e não usamos **padding**.

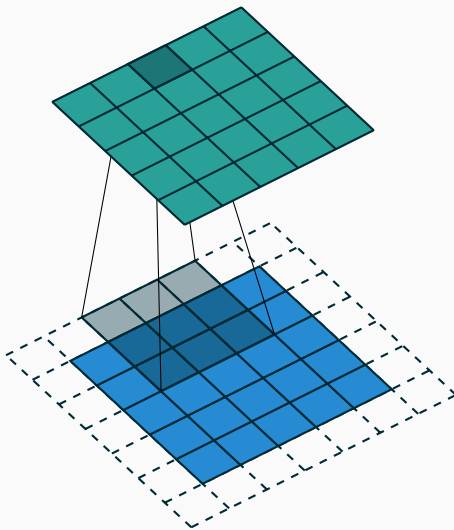
Same Padding (retirado de [2])



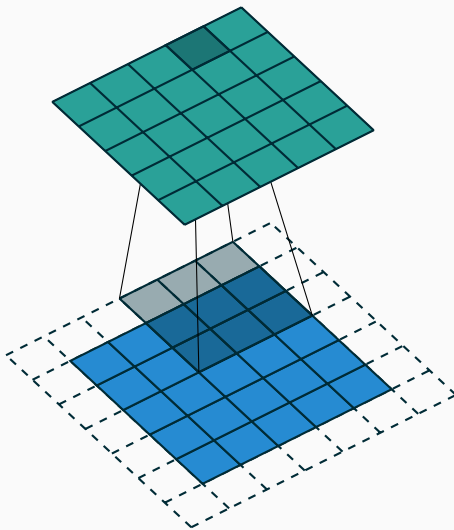
Same Padding (retirado de [2])



Same Padding (retirado de [2])



Same Padding (retirado de [2])



- Inserimos uma camada de **pooling** entre camadas de convolução.
- Fazemos isso para progressivamente diminuir o número de parâmetros.

Max Pooling (retirado de [2])

3	3	2	1	0
0	0	1	3	1
3	1	2	2	3
2	0	0	2	2
2	0	0	0	1

3.0	3.0	3.0
3.0	3.0	3.0
3.0	2.0	3.0

Max Pooling (retirado de [2])

3	3	2	1	0
0	0	1	3	1
3	1	2	2	3
2	0	0	2	2
2	0	0	0	1

3.0	3.0	3.0
3.0	3.0	3.0
3.0	2.0	3.0

Max Pooling (retirado de [2])

3	3	2	1	0
0	0	1	3	1
3	1	2	2	3
2	0	0	2	2
2	0	0	0	1

3.0	3.0	3.0
3.0	3.0	3.0
3.0	2.0	3.0

Max Pooling (retirado de [2])

3	3	2	1	0
0	0	1	3	1
3	1	2	2	3
2	0	0	2	2
2	0	0	0	1

3.0	3.0	3.0
3.0	3.0	3.0
3.0	2.0	3.0

Max Pooling (retirado de [2])

3	3	2	1	0
0	0	1	3	1
3	1	2	2	3
2	0	0	2	2
2	0	0	0	1

3.0	3.0	3.0
3.0	3.0	3.0
3.0	2.0	3.0

Max Pooling (retirado de [2])

3	3	2	1	0
0	0	1	3	1
3	1	2	2	3
2	0	0	2	2
2	0	0	0	1

3.0	3.0	3.0
3.0	3.0	3.0
3.0	2.0	3.0

Max Pooling (retirado de [2])

3	3	2	1	0
0	0	1	3	1
3	1	2	2	3
2	0	0	2	2
2	0	0	0	1

3.0	3.0	3.0
3.0	3.0	3.0
3.0	2.0	3.0

Max Pooling (retirado de [2])

3	3	2	1	0
0	0	1	3	1
3	1	2	2	3
2	0	0	2	2
2	0	0	0	1

3.0	3.0	3.0
3.0	3.0	3.0
3.0	2.0	3.0

Max Pooling (retirado de [2])

3	3	2	1	0
0	0	1	3	1
3	1	2	2	3
2	0	0	2	2
2	0	0	0	1

3.0	3.0	3.0
3.0	3.0	3.0
3.0	2.0	3.0

CONV \rightarrow POOL \rightarrow ReLU \longrightarrow (...) \longrightarrow FC

- Layers Convolucionais: filtros aprendíveis
- Pooling: reduz a dimensão da imagem
- Ativação: igual à DFN
- Fully-Connected: DFN

Parte Prática

Preparamos tutoriais básicos em **Tensorflow** e **Keras**:

<https://github.com/MLIME/12aMostra>

Esses slides também estão lá.



Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin.

Learning From Data.

AMLBook, 2012.



V. Dumoulin and F. Visin.

A guide to convolution arithmetic for deep learning, 2016.



I. Goodfellow, Y. Bengio, and A. Courville.

Deep Learning.

MIT Press, 2017.



H. Xiao, K. Rasul, and R. Vollgraf.

Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.