

ARTIFICIAL INTELLIGENCE

LEI/3, LMA/3, MBE/1

University of Beira Interior, Department of Informatics

Hugo Pedro Proença

hugomcp@di.ubi.pt, 2022/23

Linear Discriminants: Exercise

- Consider the following truth tables, corresponding to the classical “**AND**”, “**OR**” and “**XOR**” problems:
 - Suppose we want to *learn* three **logistic regression** classifiers that **appropriately discriminate between** the “0” | “1” classes

AND

X1	X2	Y
0	0	0
0	1	0
1	0	0
1	1	1



OR

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	1



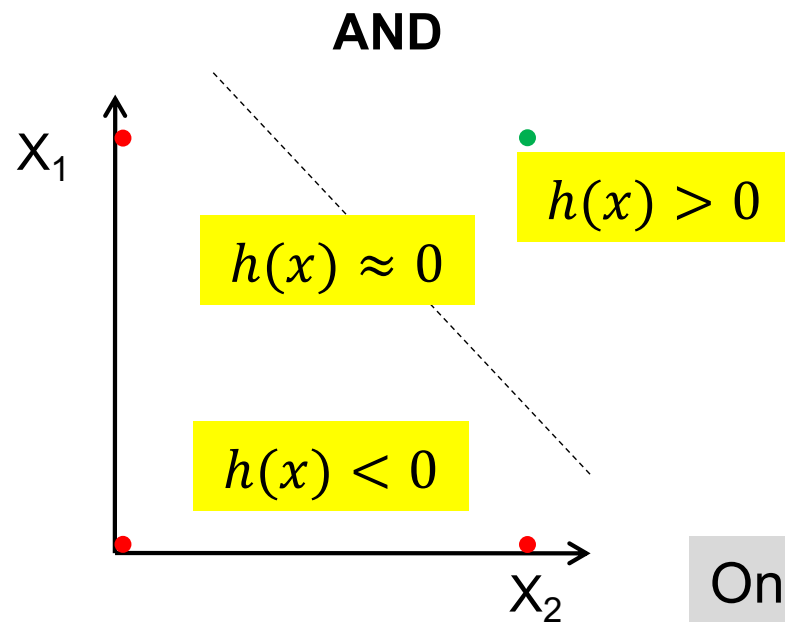
XOR

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0



Linear Discriminants: Exercise

- As we previously saw, the logistic regression is only able **to find hyperplanes (straight lines, in 2D data)** that separate the subspaces of each class, which happens in the “AND/OR” problems.
- These are called **linear discriminants**



$$g(h_{\theta}(x)) = g(\theta_1 x_1 + \theta_2 x_2 + \theta_0)$$

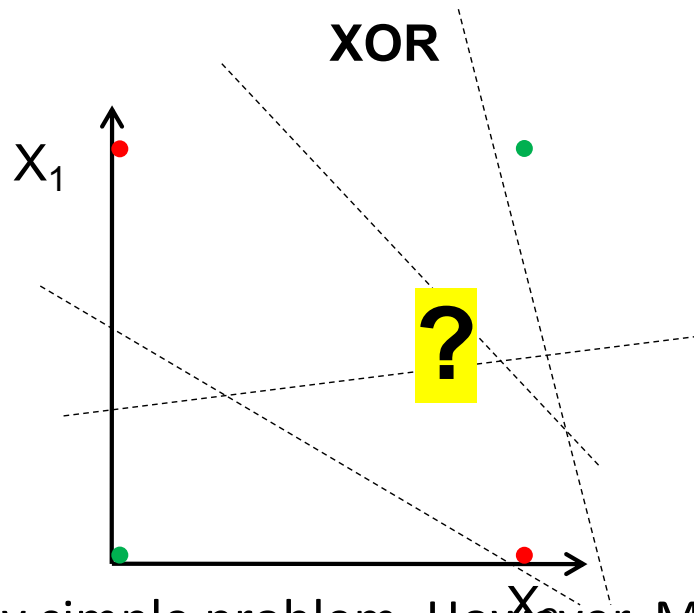
$$\frac{1}{1 + e^{-x}}$$

$$= \frac{1}{1 + e^{-(\theta_1 x_1 + \theta_2 x_2 + \theta_0)}}$$

One appropriate “AND” solution **could be**: $(\theta_1, \theta_2, \theta_0) = (0.8, 0.8, -1.5)$

Linear Discriminants: Exercise

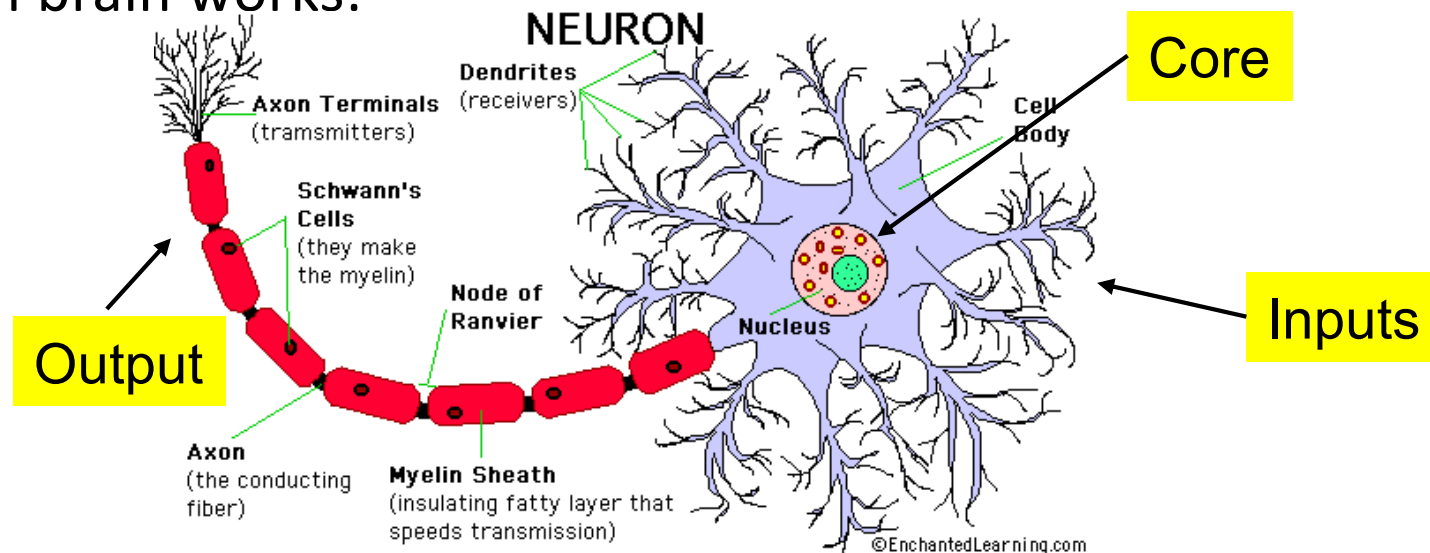
- However, for the “**XOR**” problem, there is no possible configurations for θ that satisfy the requirements:



- **XOR** appears to be a very simple problem. However, Minsky and Papert (1969) showed that this was a big problem for neural network architectures of the 1960s, known as perceptrons.
 - The inefficiency of Perceptron networks to solve this problem caused the “*NN winter*” (period up to the early 90s, when NN were almost abandoned by the ML community)

Neural Networks

- Among the three classical approaches for machine learning (pattern recognition) models, this kind of methods aims at replicate the way the human brain works:



- In practice terms, this functioning model has remarkable similarities to the way our previous models were defined:
 - “*Mixing*” the values from a set of inputs, followed by one non-linear activation function”.

Neural Networks

- A logistic regression classifier is defined by:

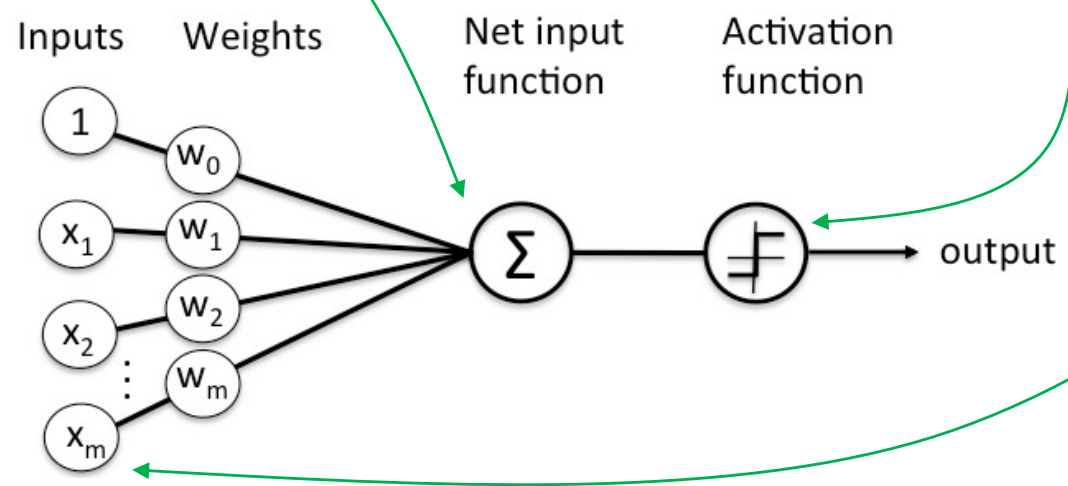
$$f_{\theta}(x) = \frac{1}{1 + e^{-(\theta_1 x_1 + \theta_2 x_2 + \theta_0)}}$$

Inputs: x_1, x_2, \dots

Phase 1:
Convolution
between \mathbf{x} and θ

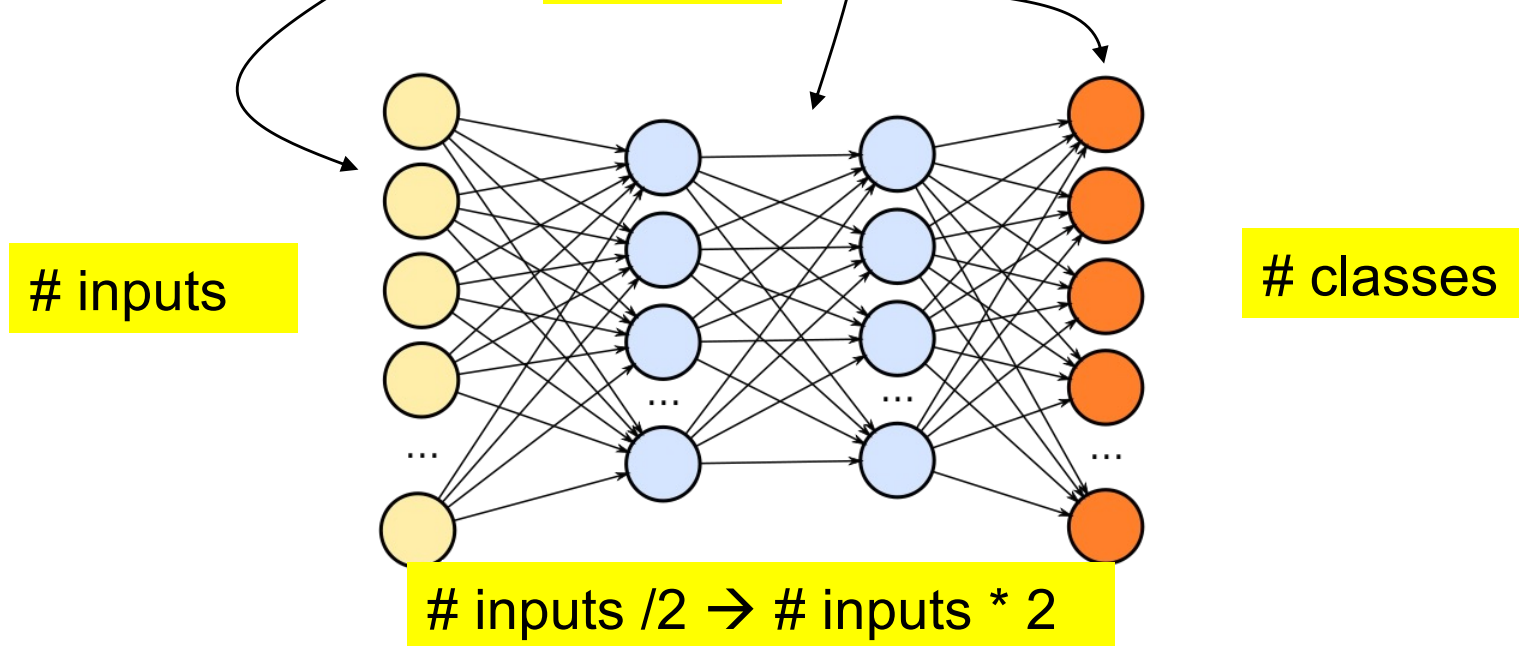
Phase 2: Non-linearity

- A Rosenblatt's perceptron is defined as:



Neural Networks: MLP Architecture

- The key concept of the most classical kind of neural networks (**feed-forward**) is to define **multiple layers**, in which neurons of one layer receive the input **of all neurons in the previous layer**.
 - These are called neurons in **hidden layers**
 - Neurons in the first layer receive the **x** input
 - They are called neurons in the **input layer**
 - Neurons in the last layer provide the result of the model
 - They are called neurons in the **output layer**



Machine Learning: Python MLP

- Let's start by the easiest part:

- How can I create one “Multi-Layer Perceptron” (MLP) network in Python and apply it to my problem?
- **Step 1:** Import the corresponding library:

```
from sklearn.neural_network import MLPClassifier
```

- **Step 2:** Have a **X** data set with shape (n, 2) and **y** with shape (n,)
 - In practice, **X** will be a “list of lists” and **y** will be a list.

```
X = [[0., 0.], [1., 1.]]  
y = [0, 1]
```

- **Step 3:** Create the network:

```
clf = MLPClassifier(solver='lbfgs', alpha=1e-5,  
                  hidden_layer_sizes=(5, 2), random_state=1)
```

- **Step 4:** Start learning:

```
clf.fit(X, y)
```

- **Step 5:** Use it, to predict on new instances:

```
clf.predict([[2., 2.], [-1., -2.]])
```


Neural Networks

- When designing a neural network, there are different parameterizations that have to be chosen, with might determine the system effectiveness:

- The **number of neurons** in the input/output layers result directly of the problem considered:

- **Input Layer = Dimension of the Feature Space**

- **Output Layer = Number of classes (hot encoded)**

1	→	0 0 1
2	→	0 1 0
3	→	1 0 0

- In the hidden layers, the number of neurons can vary:

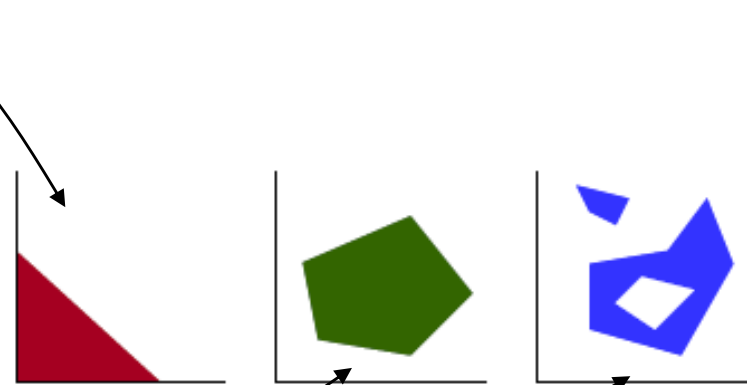
- A **too short** number might not be enough to model the decision surface desired;
- A **too high** value might lead to **overfitting**
- In practice, values between half and the double of the number of neurons in the input layer are tested

- Regarding the number of hidden layers:

Networks with **one layer** have the ability to approximate any linear decision surface

Networks with **two layers** approximate any continuous decision surface

Networks with **three layers** approximate any decision surface

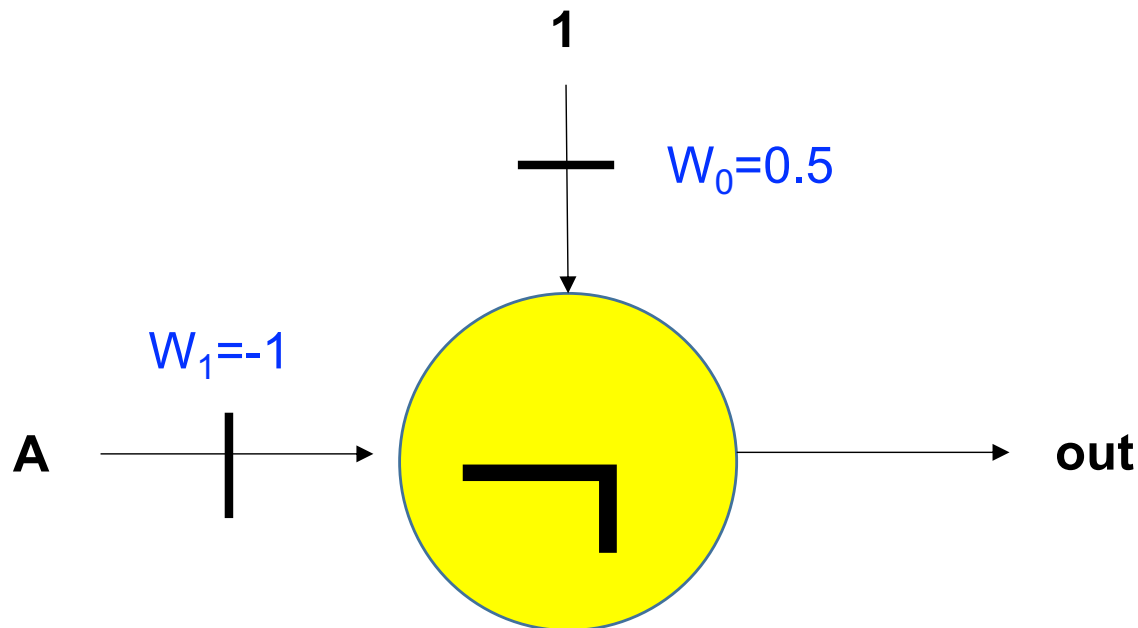


Machine Learning: NN Exercise

- Considering that:

$$A \otimes B = \neg ((A \wedge B) \vee (\neg A \wedge \neg B))$$

- For example, how to infer the weights for a “NOT” neuron, i.e., a neuron that replicates the functioning of a logical “NOT” operation.
 - In this simple case, there are various weight configurations that will work

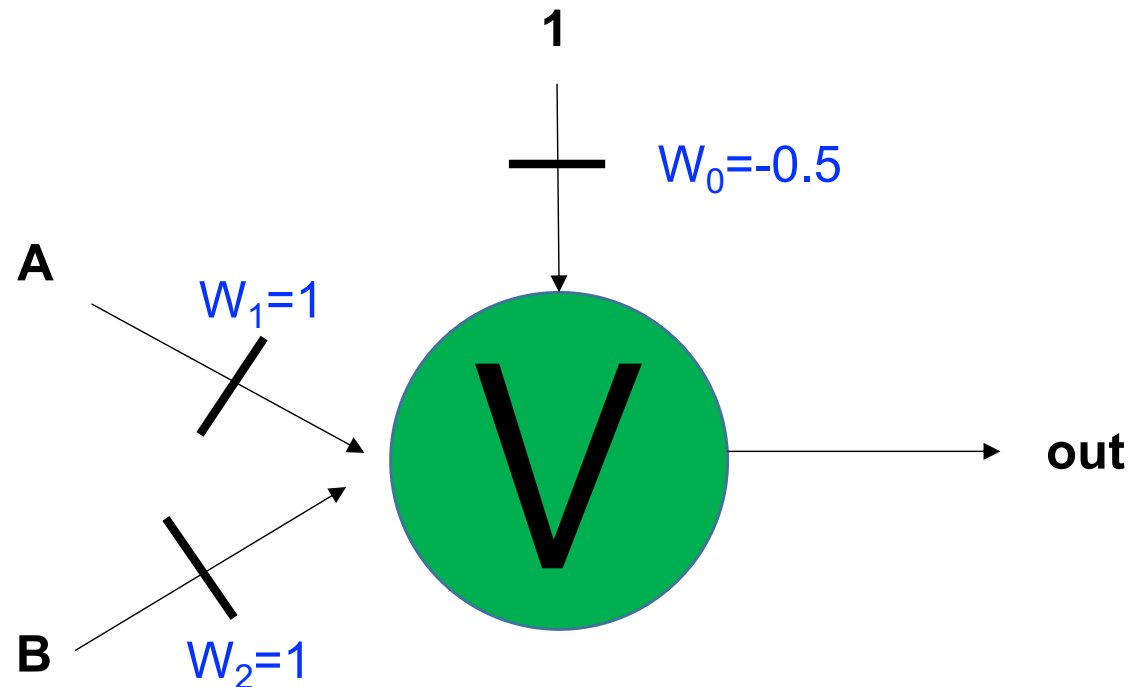


Machine Learning: NN Example

- Considering that:

$$A \otimes B = \neg((A \wedge B) \vee (\neg A \wedge \neg B))$$

- Now, how to infer the weights for a “OR” neuron, i.e., a neuron that replicates the functioning of a logical “OR” operation.
 - Again, there are various weight configurations that will work:

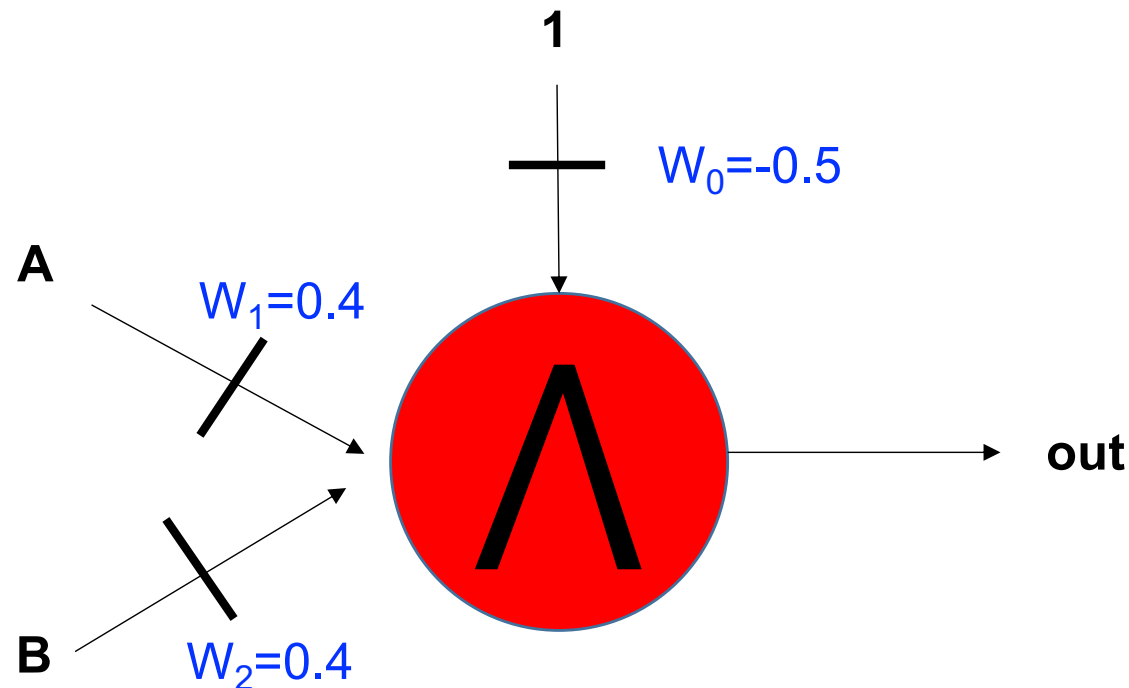


Machine Learning: NN Example

- Considering that:

$$A \otimes B = \neg((A \wedge B) \vee (\neg A \wedge \neg B))$$

- Next, in a similar way, if we want to infer the weights for a “AND” neuron, i.e., a neuron that replicates the functioning of a logical “AND” operation.
 - As in the previous cases, there are various weight configurations that will work:

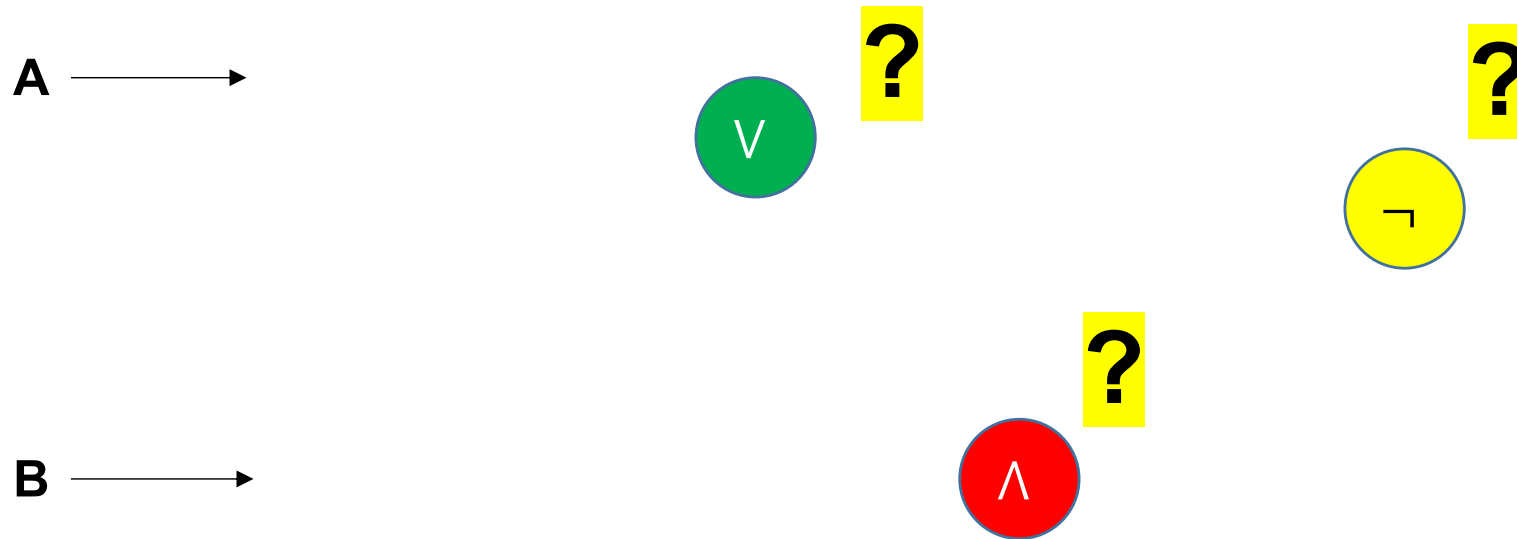


Machine Learning: NN Exercise

- Considering that:

$$A \otimes B = \neg ((A \wedge B) \vee (\neg A \wedge \neg B))$$

- Design a multi-layer network, with the corresponding weights θ , able to solve the "XOR" problem.

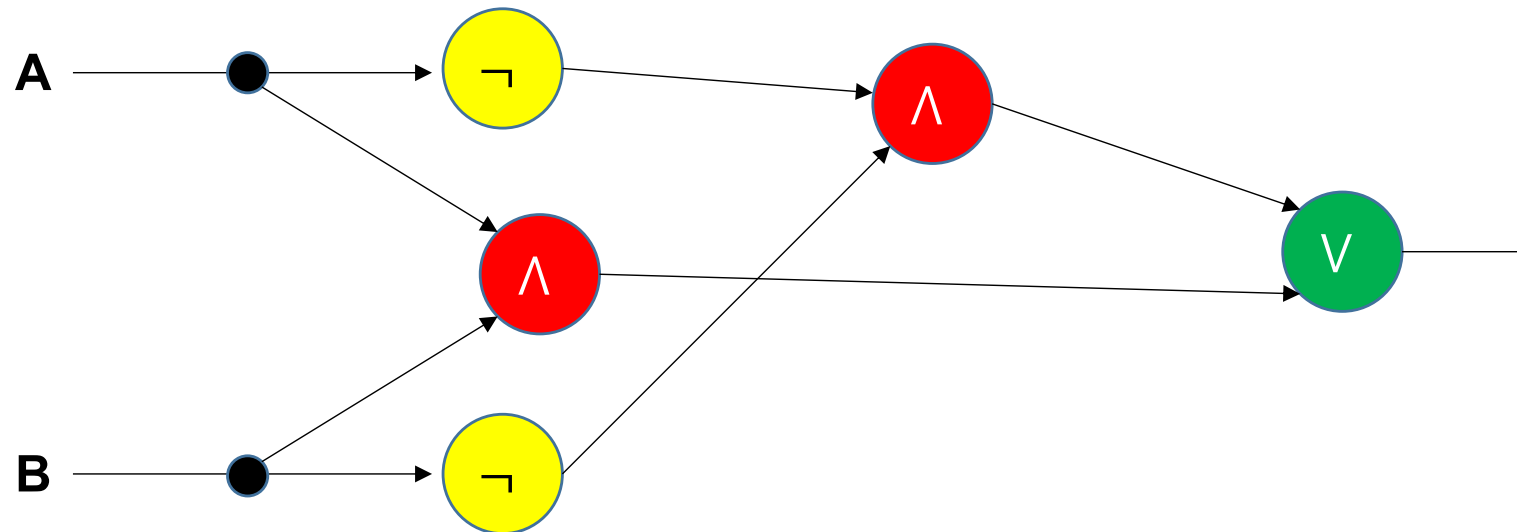


Machine Learning: NN Exercise

- Considering that:

$$A \otimes B = \neg ((A \wedge B) \vee (\neg A \wedge \neg B))$$

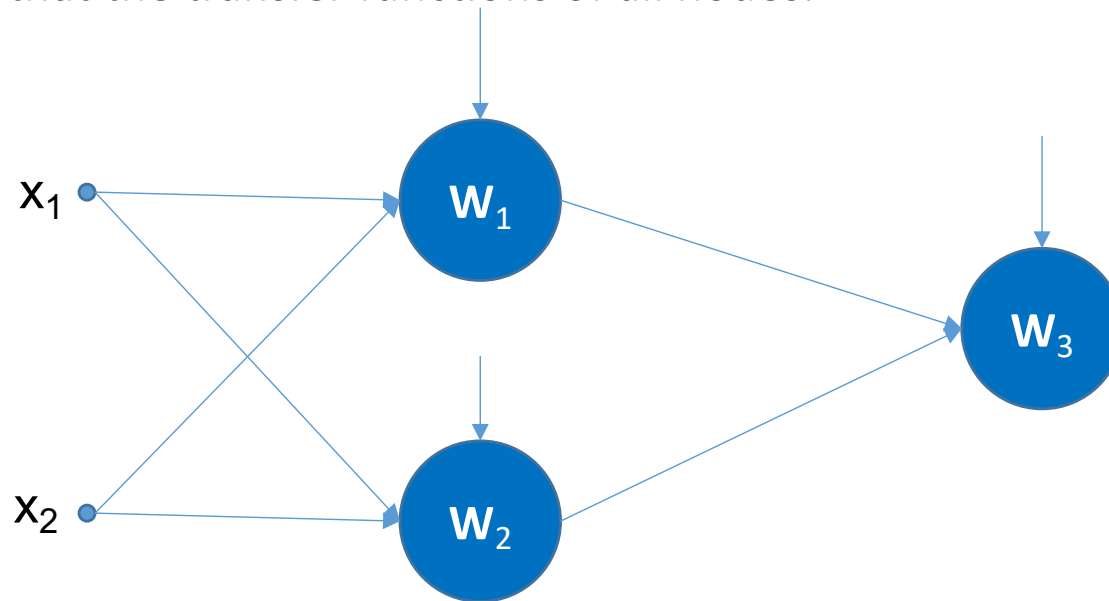
- Design a multi-layer network, with the corresponding weights θ , able to solve the “XOR” problem.



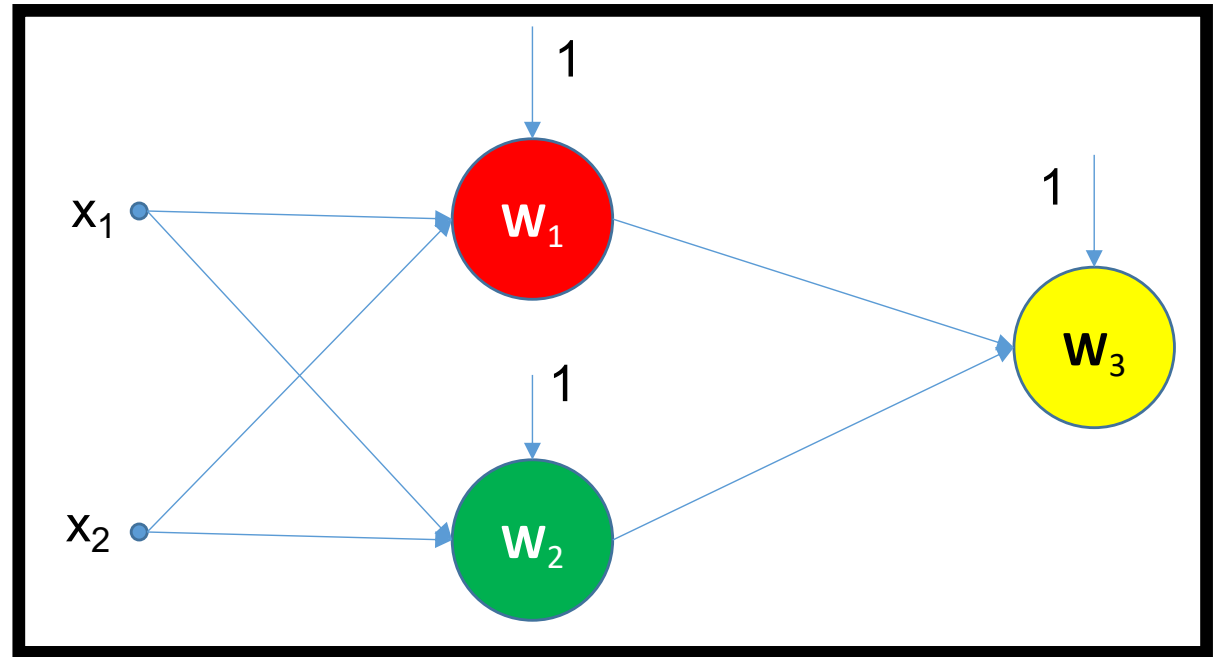
- This will be a network “specific” to reproduce this function.
- However, the big question remains: **How to automatically obtain the θ values?**

Neural Networks: Learning

- In case of multilayered networks, the closed-form equation for the whole network, the cost function and the corresponding derivatives might not be easy to obtain.
- **Exercise:**
 - Obtain the function that describes the functioning of the following network, considering that the transfer functions of all nodes.



Backpropagation



$$t_1 = \frac{1}{1 + e^{-w_{1,0} - w_{1,1}x_1 - w_{1,2}x_2}}$$

$$t_2 = \frac{1}{1 + e^{-w_{2,0} - w_{2,1}x_1 - w_{2,2}x_2}}$$

$$NN = \frac{1}{1 + e^{-w_{3,0} - w_{3,1}t_1 - w_{3,2}t_2}}$$

$$NN = \frac{1}{1 + e^{-w_{3,0} - w_{3,1} \frac{1}{1 + e^{-w_{1,0} - w_{1,1}x_1 - w_{1,2}x_2}} - w_{3,2} \frac{1}{1 + e^{-w_{2,0} - w_{2,1}x_1 - w_{2,2}x_2}}}}$$

Backpropagation

$$NN = \frac{1}{1 + e^{-w_{3,0} - w_{3,1} \frac{1}{1 + e^{-w_{1,0} - w_{1,1}x_1 - w_{1,2}x_2}} - w_{3,2} \frac{1}{1 + e^{-w_{2,0} - w_{2,1}x_1 - w_{2,2}x_2}}}}$$

$$J(\mathbf{w}) = \frac{1}{N} \sum Cost(NN(\mathbf{w}, x^{(i)}, y^{(i)}))$$

$$\bullet Cost(NN(\mathbf{w}, x^{(i)}, y^{(i)})) = \begin{cases} -\log(NN(\mathbf{w}, x^{(i)})), & \text{if } y^{(i)}=1 \\ -\log(1 - NN(\mathbf{w}, x^{(i)})), & \text{if } y^{(i)}=0 \end{cases}$$

- Therefore, as we did before for the logistic regression classifier, the cost function is combined in a single function:

$$J(\mathbf{w}) = -\frac{1}{N} \sum_i y^{(i)} \log(NN(\mathbf{w}, x^{(i)})) + (1 - y^{(i)}) \log(1 - NN(\mathbf{w}, x^{(i)}))$$

Backpropagation

- Using the gradient descent (delta rule) learning strategy previously described, it will be required to obtain:

$$\frac{\partial}{\partial w_{1,0}} = ?$$

$$\frac{\partial}{\partial w_{1,1}} = ?$$

$$\frac{\partial}{\partial w_{1,2}} = ?$$

$$\frac{\partial}{\partial w_{2,0}} = ?$$

$$\frac{\partial}{\partial w_{2,1}} = ?$$

$$\frac{\partial}{\partial w_{2,2}} = ?$$

$$\frac{\partial}{\partial w_{3,0}} = ?$$

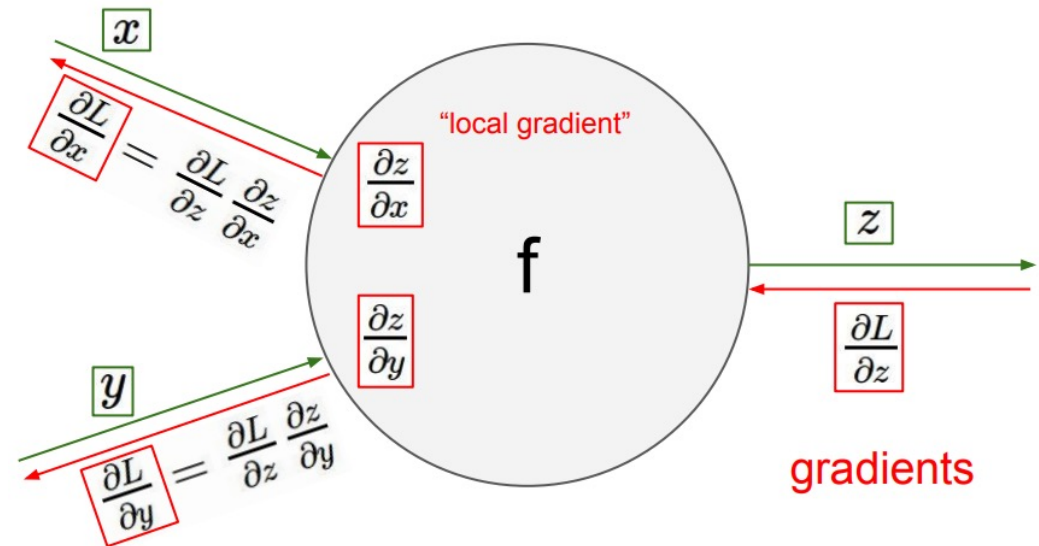
$$\frac{\partial}{\partial w_{3,1}} = ?$$

$$\frac{\partial}{\partial w_{3,2}} = ?$$

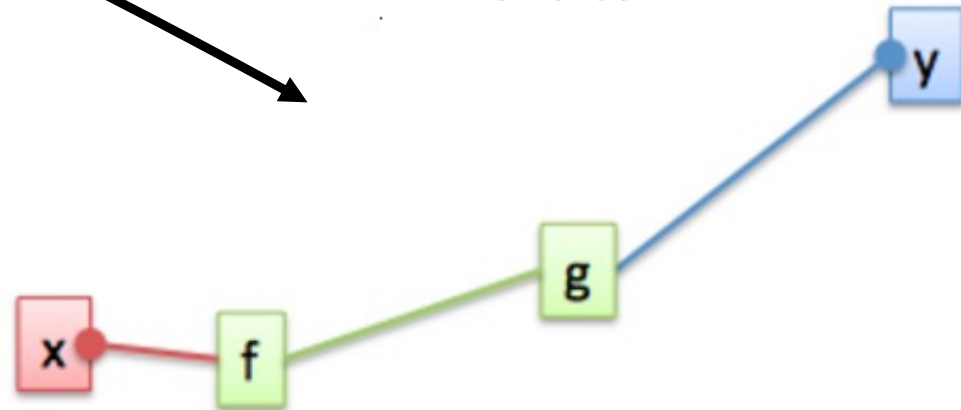
...and this is a tiny network...

Backpropagation and the Chain Rule

- “**Backpropagation**” is the short name for “**backward propagation of errors**”
- It is an algorithm for supervised learning of multi-layer artificial neural networks, based in gradient descent
- The key concept is the **chain rule**:
 - $\delta g / \delta x = \delta g / \delta f \cdot \delta f / \delta x$
- Calculates the gradient of the error function with respect to the neural network's weights;
- It is a **generalization** of the delta rule for perceptrons to multilayer feed-forward neural networks.



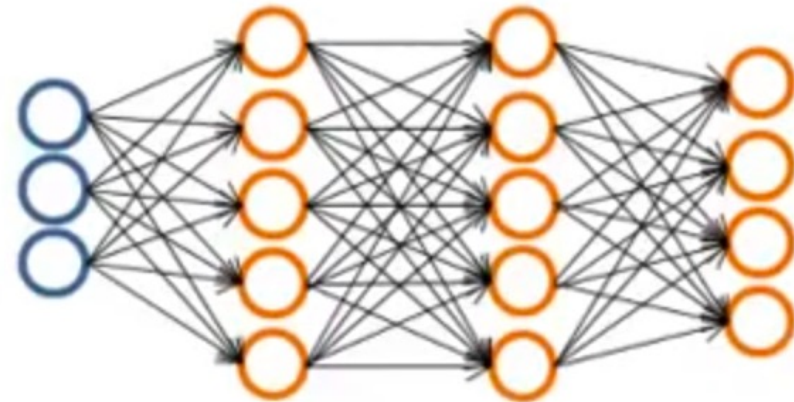
$$y = g(f(x))$$



Backpropagation: Forward Pass

- Forward Pass:

- Let $p^{(i,j)}$ denote the “inner product” between the inputs and the weights of the j^{th} neuron of the i^{th} layer of the network;
- Let $s()$ denote the sigmoid transfer function;
- Hence, the output of one neuron is given by $s^{(i,j)}(p^{(i,j)})$
- We consider that at the input layer the output is simply given by the network inputs, i.e., $s(p^{(0,*)}) = \mathbf{x}$
- $p^{(1,*)} = \mathbf{w}^{(1,*)} s^{(0,*)}$
- $s^{(1,*)} = s(p^{(1,*)})$
- $p^{(2,*)} = \mathbf{w}^{(2,*)} s^{(1,*)}$
- ...

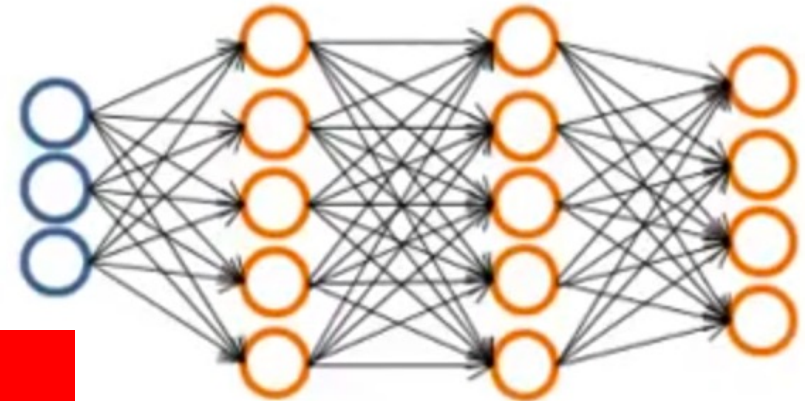


$$p^{(i,*)} = \mathbf{w}^{(i,*)} s^{(i-1,*)}$$

$$s^{(i,*)} = s(p^{(i,*)})$$

Backpropagation: Backward Pass

- We start by obtaining the error at the output layer $e^{(\text{last})}$
- Use this value to obtain the error at the previous layer ($e^{(\text{last}-1)}$) and so on...
- $e^{(\text{last},*)} = s^{(\text{last},*)} - y$
- $e^{(\text{last}-1,*)} = w^{(\text{last}-1,*)T} \cdot e^{(\text{last})} \cdot [s(p^{(\text{last}-1)})]'$
- Note that $[s(p^{(\text{last}-1)})]' = s^{(\text{last}-1)} \cdot (1 - s^{(\text{last}-1)})$



$$e^{(i,*)} = w^{(i,*)T} \cdot e^{(i+1)} \cdot [s(p^{(i)})]'$$

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

$$\frac{d\sigma(x)}{d(x)} = \sigma(x) \cdot (1 - \sigma(x))$$

Backpropagation Algorithm

- Having a learning set $\{(x^{(1)}, y^{(1)}), \dots, (x^{(n)}, y^{(n)})\}$, and a NN with “L” layers.

$u^{(l,i,j)}=0$ //update factor for the l^{th} layer, i^{th} neuron, j^{th} weight

For $i=1..n$

$s^{(1)}=x^{(i)}$ //input layer

Perform forward propagation to obtain $s^{(l)}$, $l=1,..L$

Use $y^{(i)}$ to obtain $e_i^{(L)}$

Obtain $e^{(L-1)}, \dots, e^{(2)}$

$u^{(l,i,j)} = u^{(l,i,j)} + s^{(l,j)} e_i^{(l+1)}$

$D^{(l,i,j)} = 1/n u^{(l,i,j)} + \text{STEP } w^{(l,i,j)}$ for “non-bias” neuron

$D^{(l,i,j)} = 1/n u^{(l,i,j)}$ for “bias” neurons

$$\frac{\partial J}{\partial w_{l,i,j}} = D^{(l,i,j)}$$