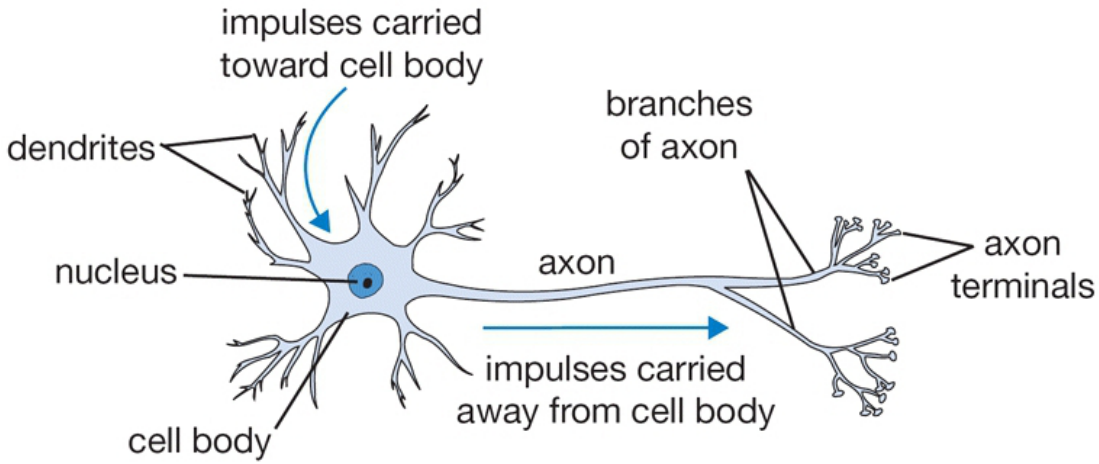
# Artificial neural networks and backpropagation

E. Decencière

MINES ParisTech
PSL Research University
Center for Mathematical Morphology

# Contents

# Biological neuron

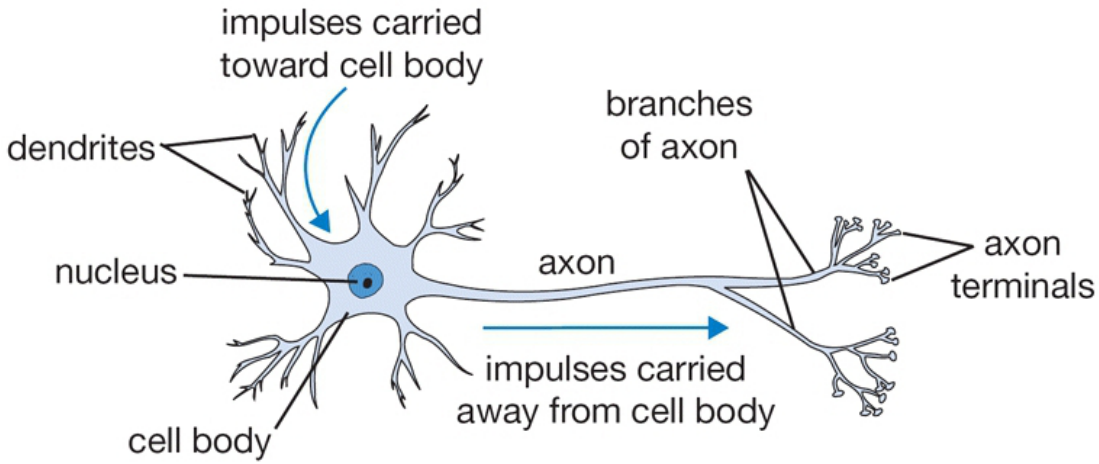



The human brain contains:

- $10^7$
- $10^9$
- $10^{11}$

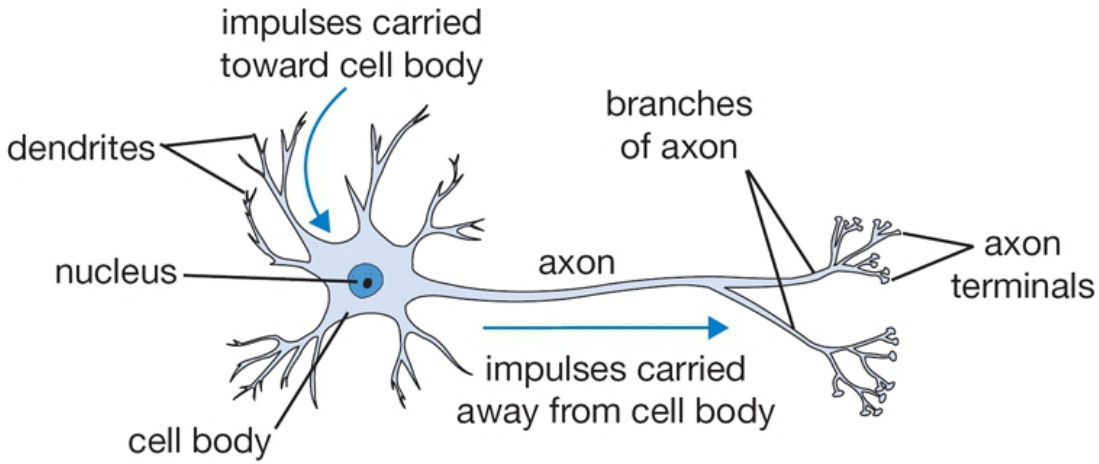
neurons

# Biological neuron



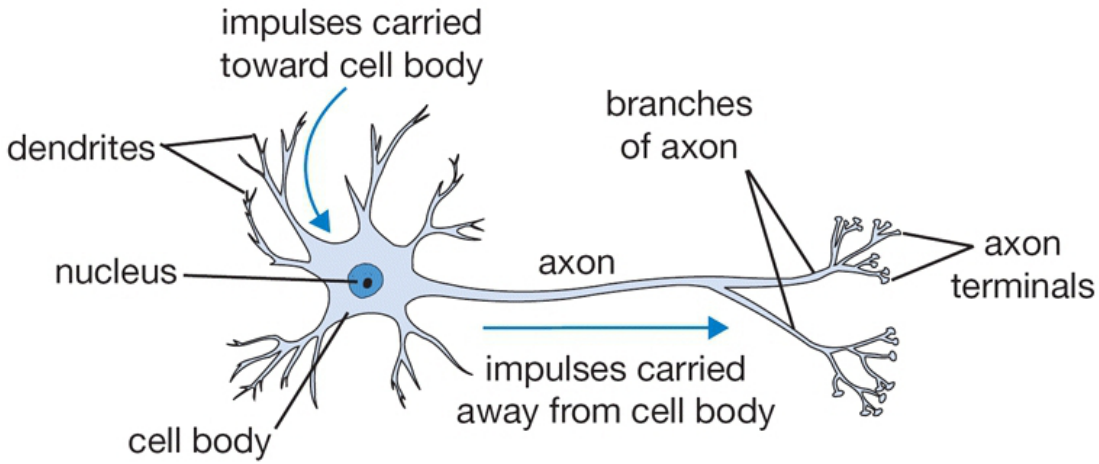

- The human brain contains 100 billion ($10^{11}$) neurons

# Biological neuron



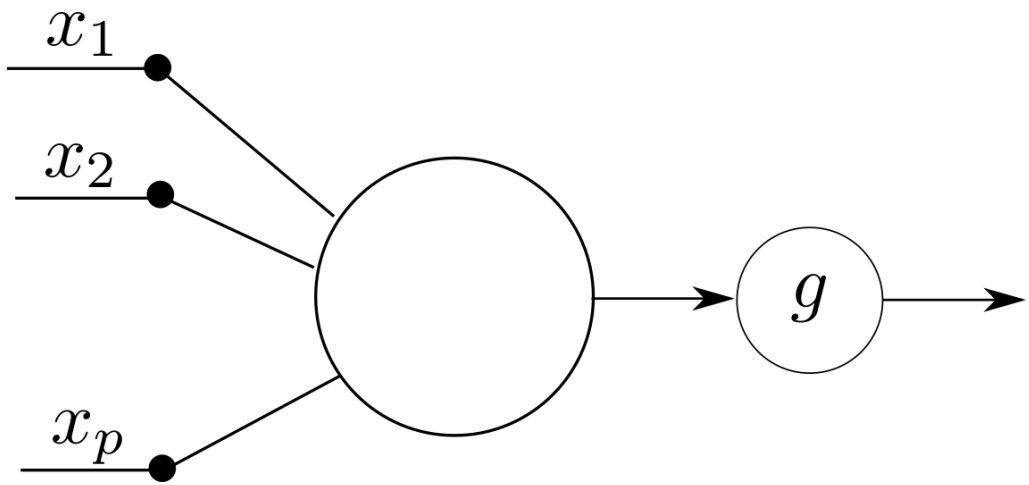

- The human brain contains 100 billion ($10^{11}$) neurons
- A human neuron can have several thousand dendrites

# Biological neuron




- The human brain contains 100 billion ($10^{11}$) neurons
- A human neuron can have several thousand dendrites
- The neuron sends a signal through its axon if during a given interval of time the net input signal (sum on excitatory and inhibitory signals received through its dendrites) is larger than a threshold.

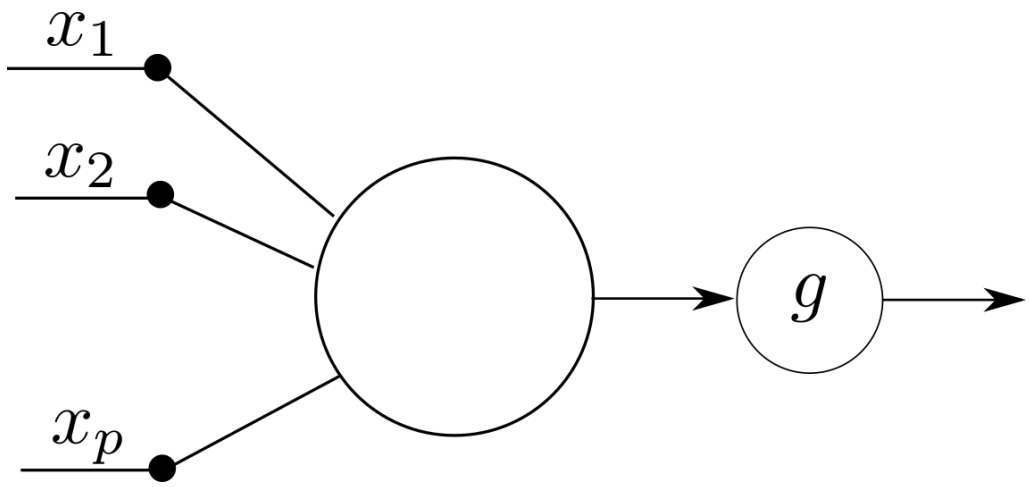
# Artificial neuron




## General principle

An artificial neuron takes $p$ inputs $\{x_i\}_{1 \leq i \leq p}$, combines them to obtain a single value, and applies an activation function g to the result.

# Artificial neuron

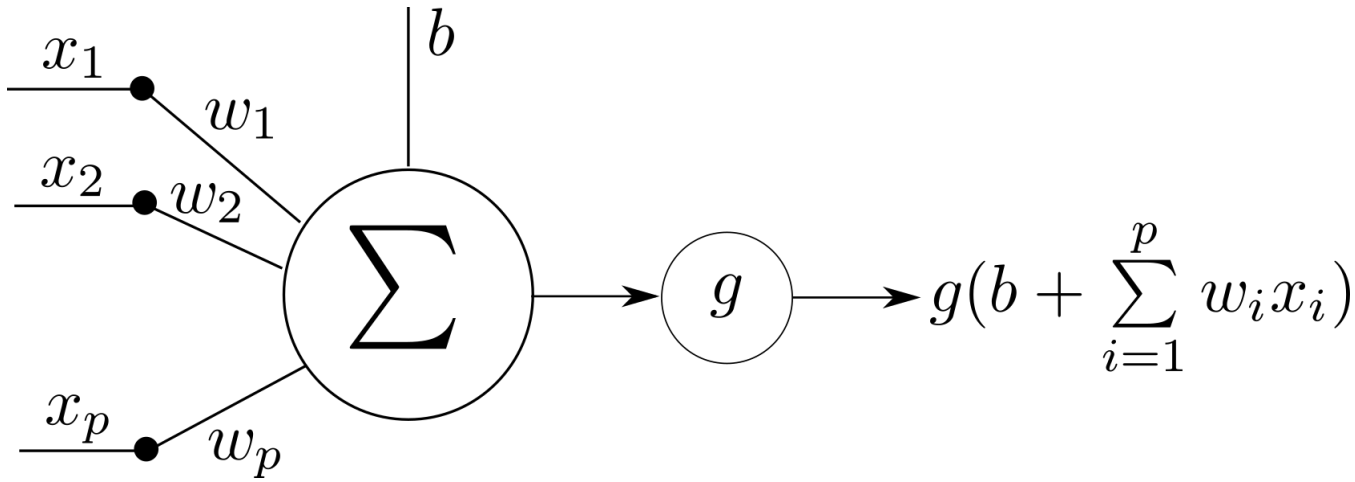



## General principle

An artificial neuron takes $p$ inputs $\{x_i\}_{1 \leq i \leq p}$, combines them to obtain a single value, and applies an activation function g to the result.

- The first artificial neuron model was proposed by [McCulloch and Pitts, 1943]
- Input and output signals were binary
- Input dendrites could be inhibitory or excitatory

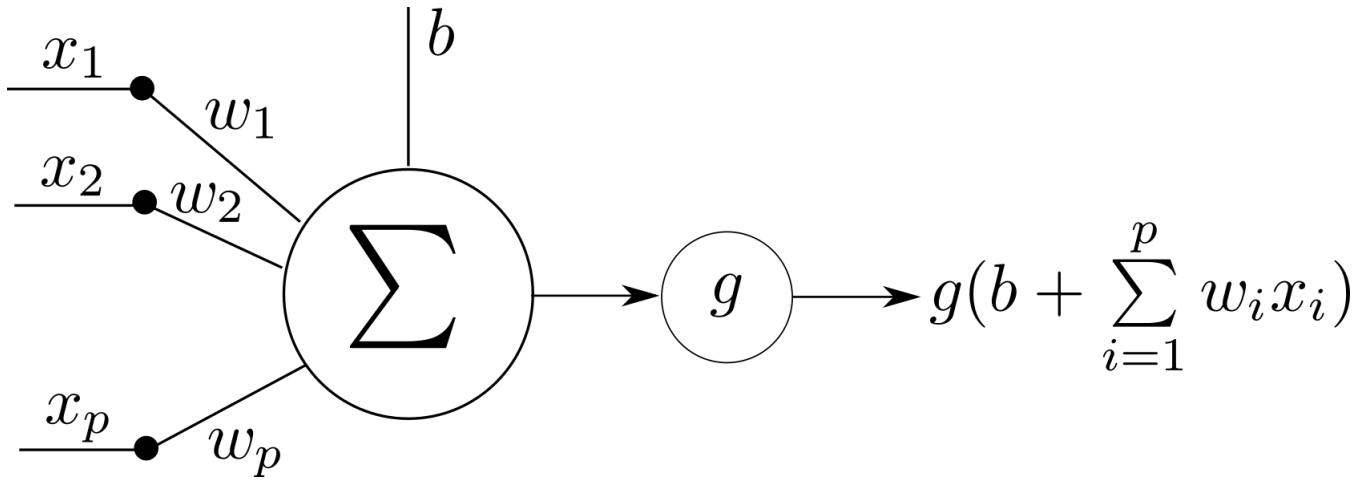
# Modern artificial neuron




- The neuron computes a linear combination of the inputs $x_i$
  - The weights $w_i$ are multiplied with the inputs
  - The bias $b$ can be interpreted as a threshold on the sum
- The activation function g somehow decides, depending on its input, if a signal (the neuron's activation) is produced

# Contents

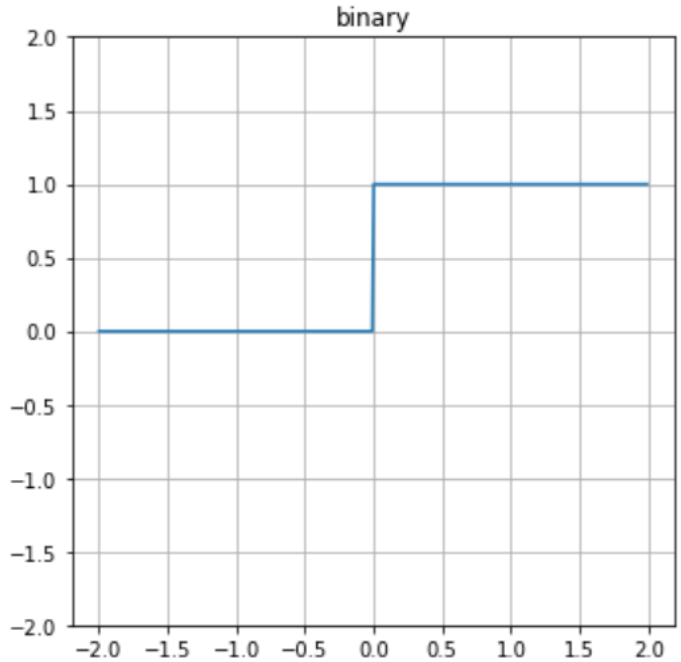
# The role of the activation function




- The initial idea behind the activation function is that it works somehow as a gate
- If its input in "high enough", then the neuron is activated, i.e. a signal (other than zero) is produced
- It can be interpreted as a source of abstraction: information considered as unimportant is ignored

# Activation: binary

$$g(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

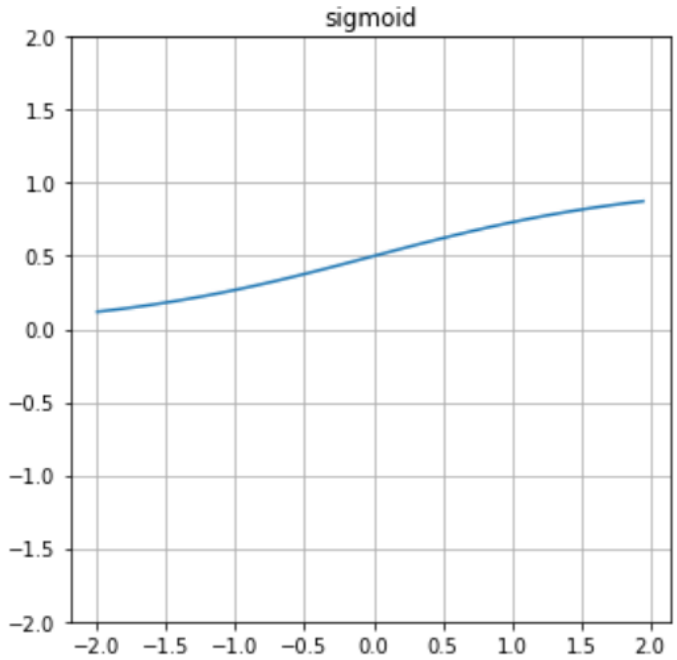



## Remarks

- Biologically inspired
- \+ Simple to compute
- \+ High abstraction
- \- Gradient nil except on one point
- In practice, almost never used

# Activation: sigmoid

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

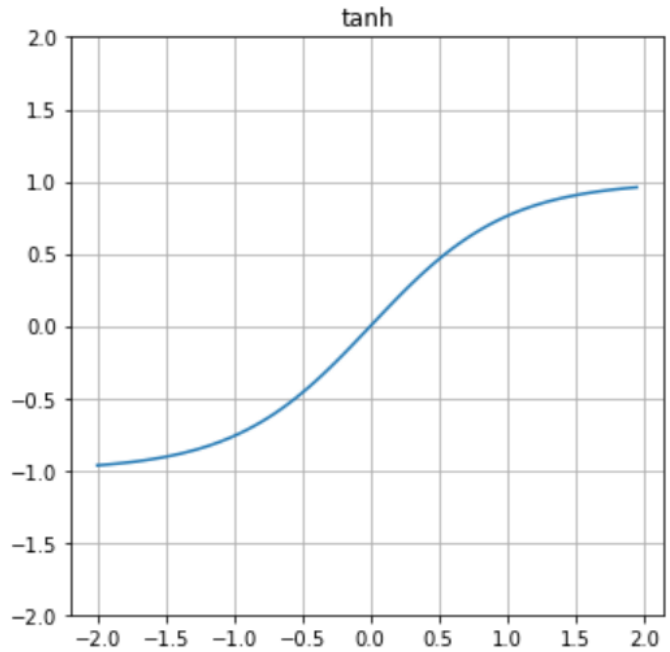



## Remarks

+ Similar to binary activation, but with usable gradient
- Bijection between $\mathbb{R}$ and $]0,1[$: no loss of information
- Gradient tends to zero when input is far from zero
- More computationally intensive

# Activation: hyperbolic tangent

$$\mathrm{g}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

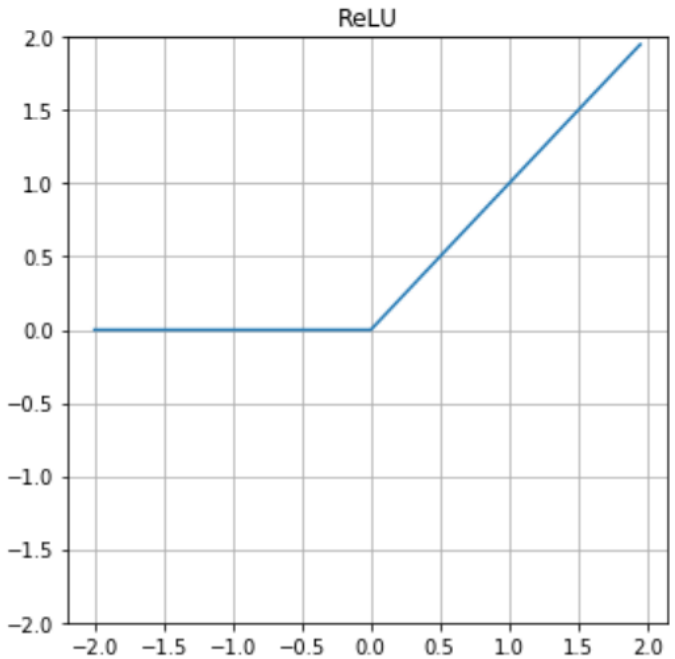



## Remarks

- Similar to sigmoid

# Activation: rectified linear unit (ReLU)

$$\mathrm{g}(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

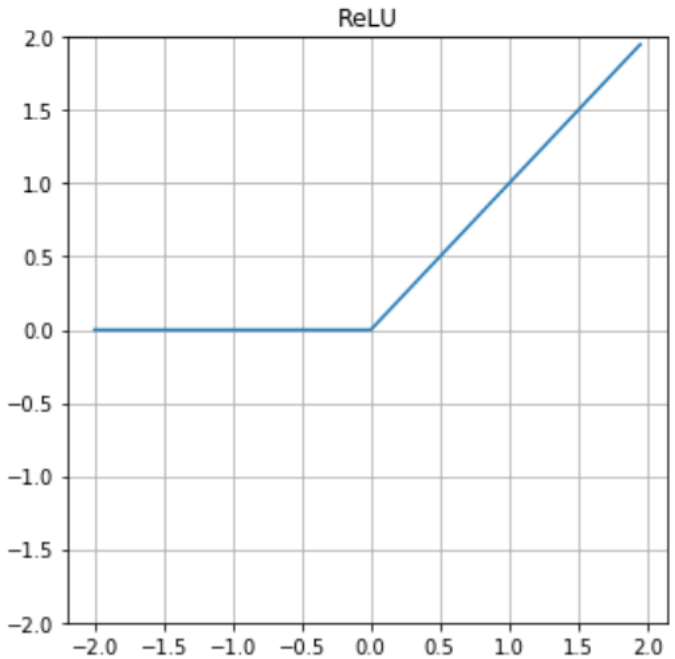



## Remarks

+ Usable gradient when activated
+ Fast to compute
+ High abstraction

# Activation: rectified linear unit (ReLU)

$$\mathrm{g}(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$



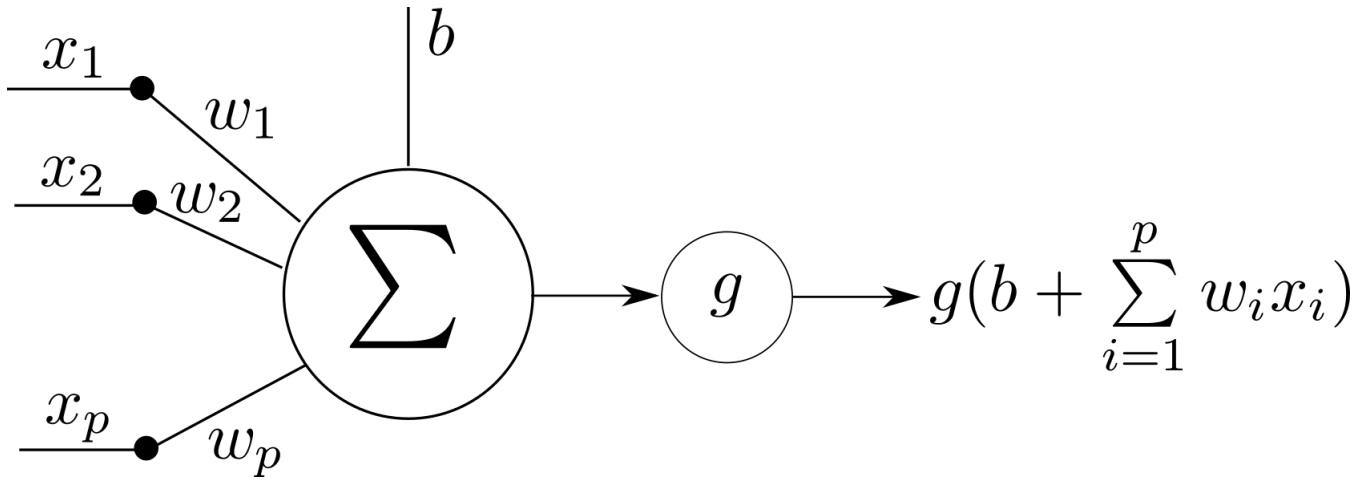

## Remarks

+ Usable gradient when activated
+ Fast to compute
+ High abstraction

ReLU is the most commonly used activation function.

# Contents

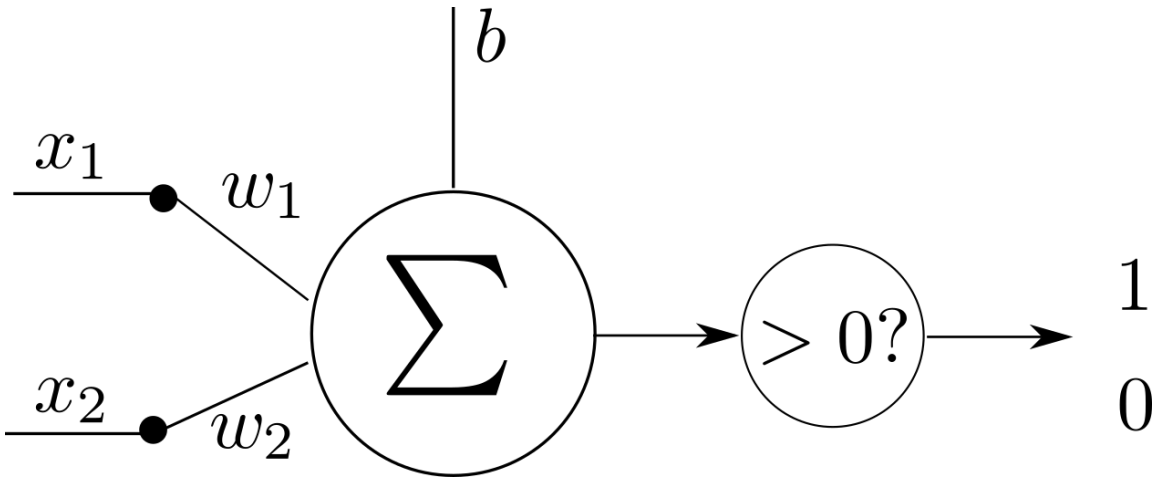
# What can an artificial neuron compute?




In $\mathbb{R}^p$ , $b + \sum_{i=1}^{p} w_i x_i = 0$ corresponds to a hyperplane $H$. For a given point $\mathbf{x} = \{x_1, \ldots, x_p\}$, decisions are made according to the side of the hyperplane it belongs to.

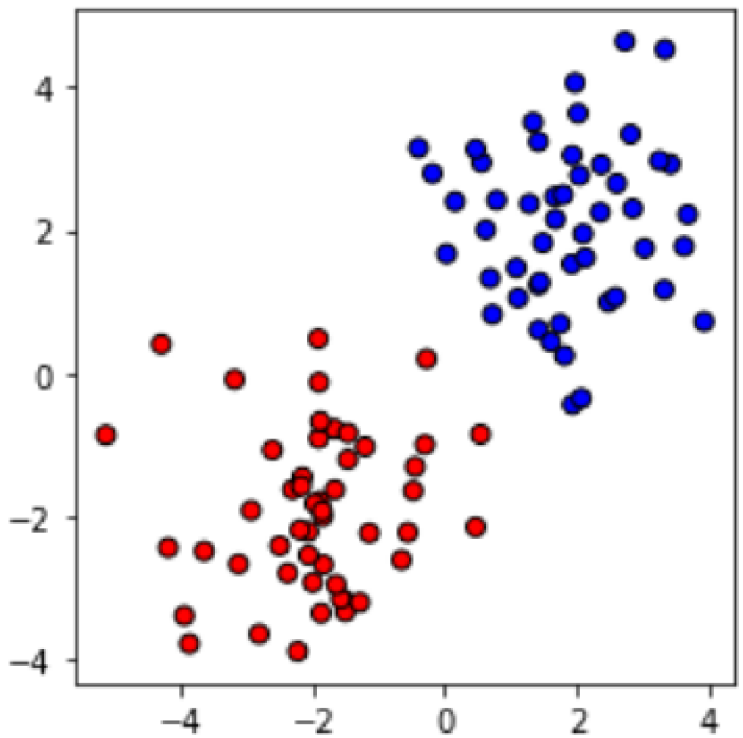
When the activation function is binary, we obtain a perceptron

# Example of what we can do with a neuron

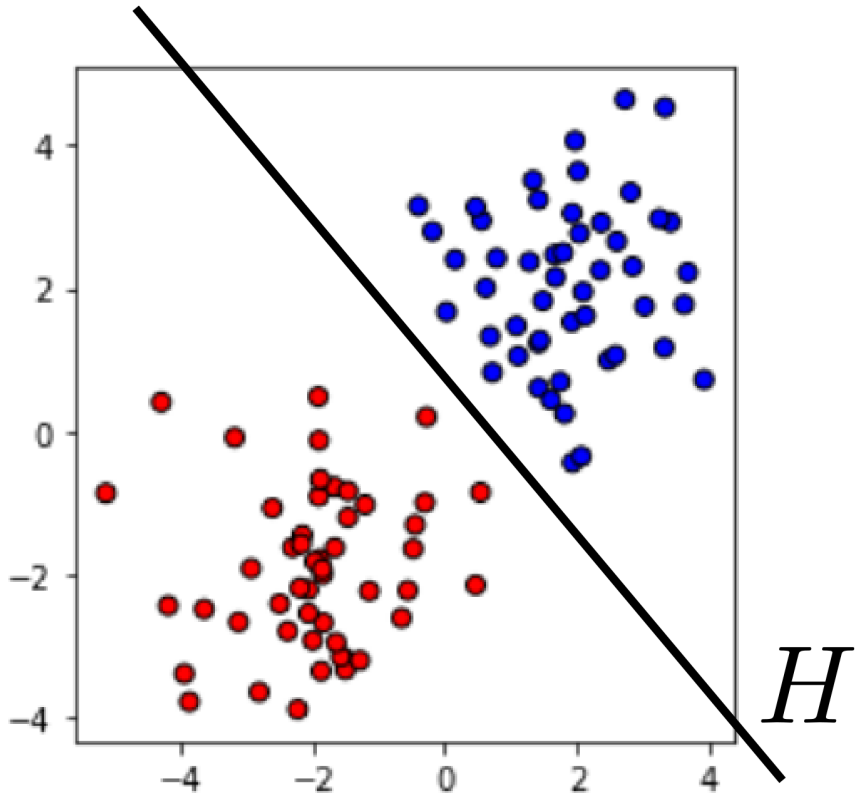



- $p = 2$ : 2-dimensional inputs (can be represented on a screen!)
- Activation: binary
- Classification problem

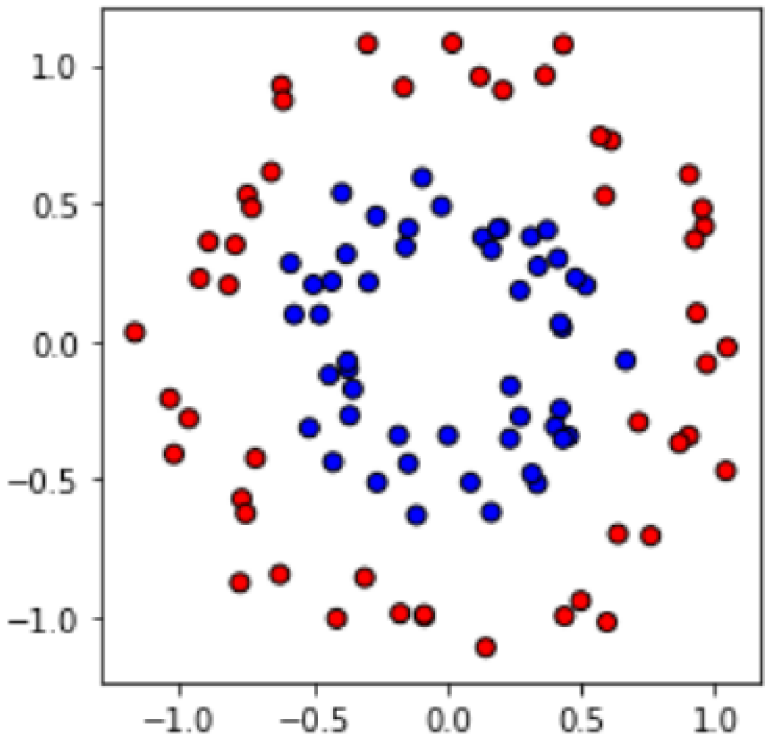
# Gaussian clouds



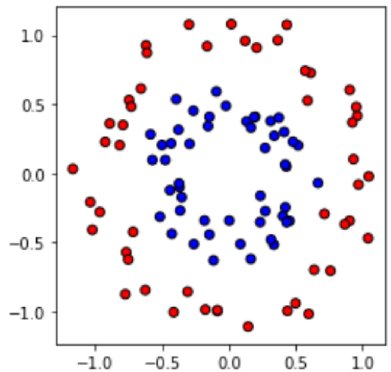
# Gaussian clouds



# Circles

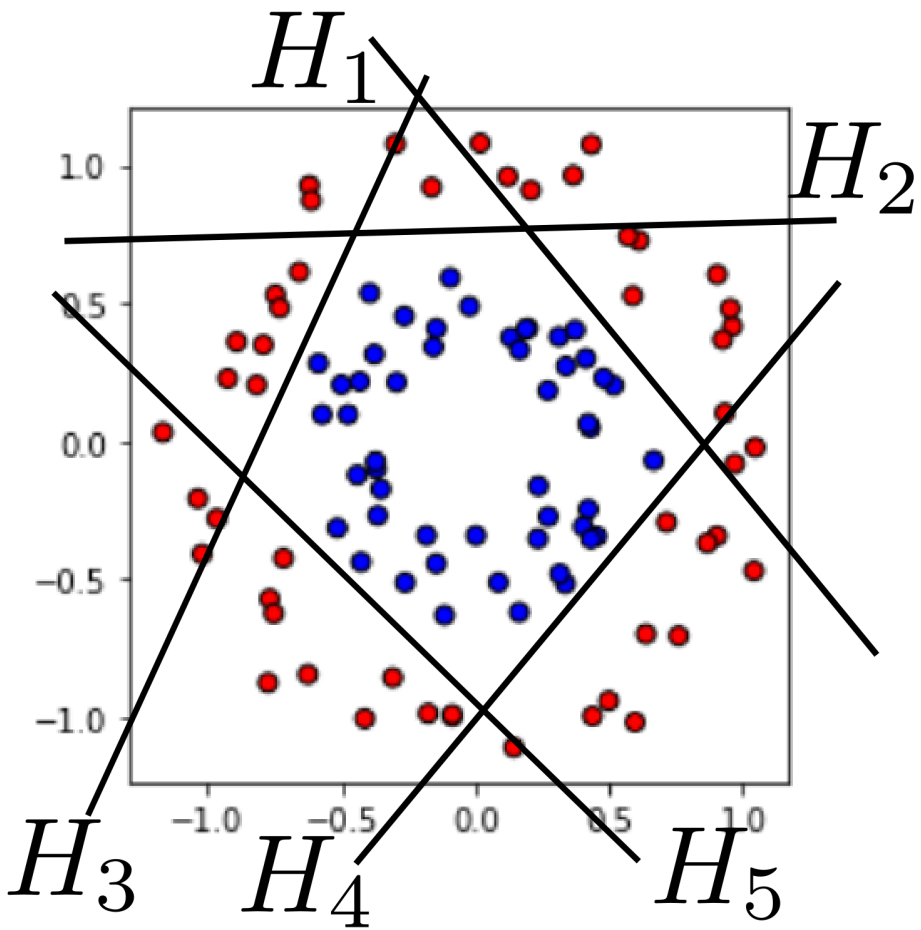


# Solution?

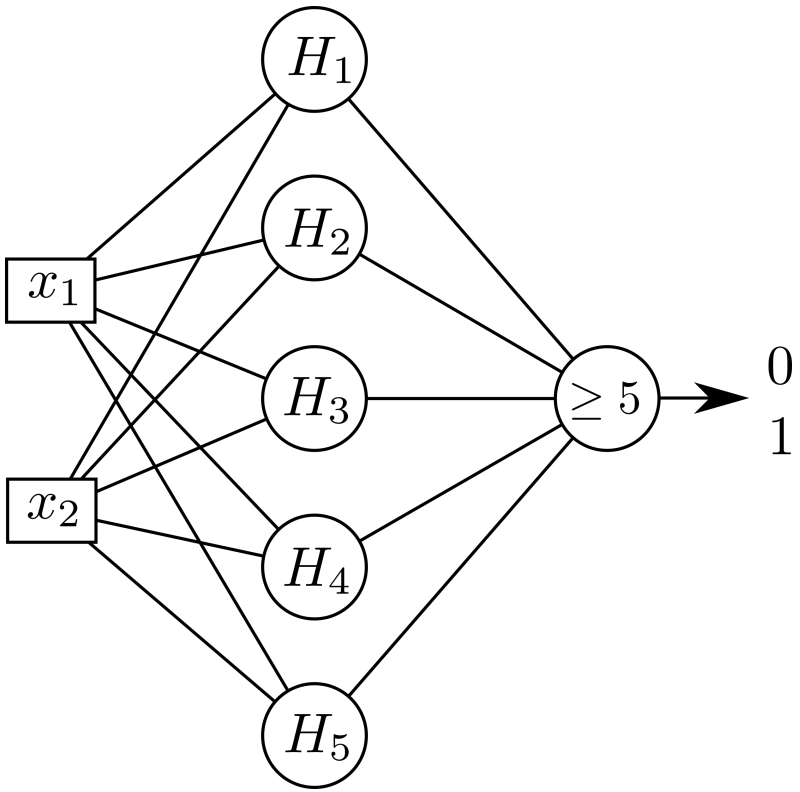


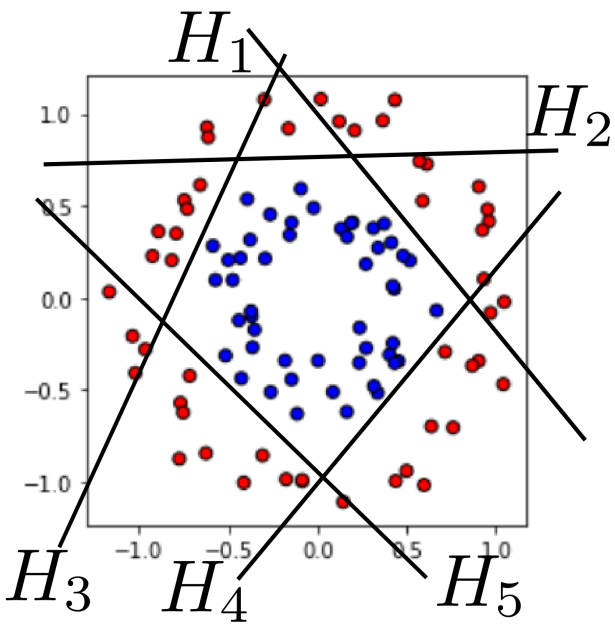

- Transform the input space, for example use a polar transformation
- Increase the number of input dimensions: for example add $x_1^2, x_2^2, x_1 x_2$ to the initial features $x_1, x_2$
- Combine several neurons

# Circles



# Solution

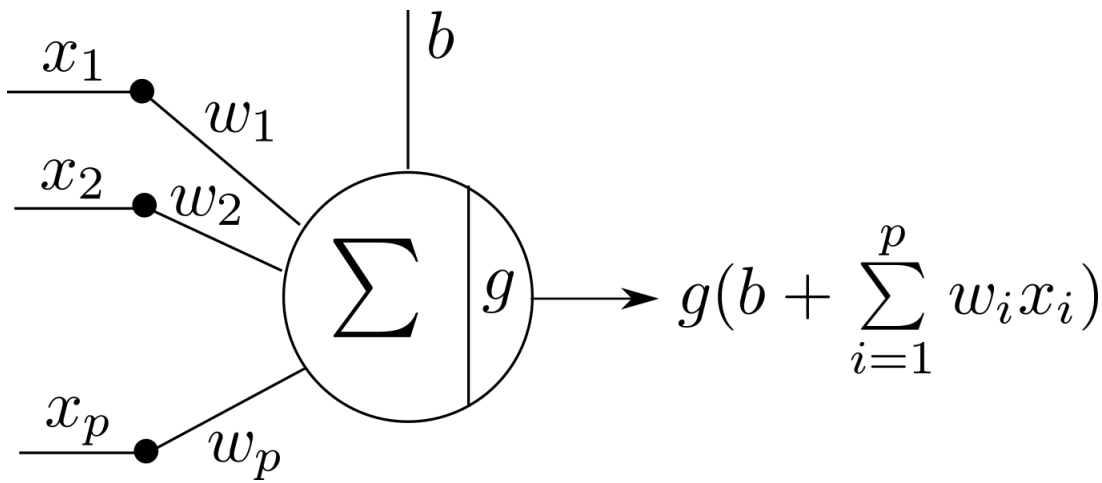


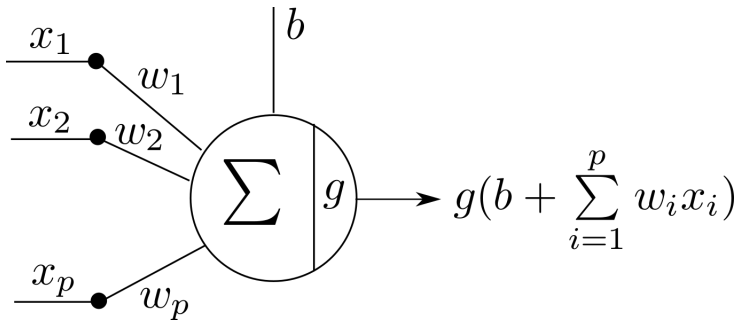

## Intuition

Combining several neurons one can build complex classifiers.

# Compact representation



# Notations




With

$$\mathbf{w} = \begin{pmatrix} w_1 \\ \vdots \\ w_p \end{pmatrix} = (w_1, \ldots, w_p)^T$$

and

$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_p \end{pmatrix} = (x_1, \ldots, x_p)^T$$

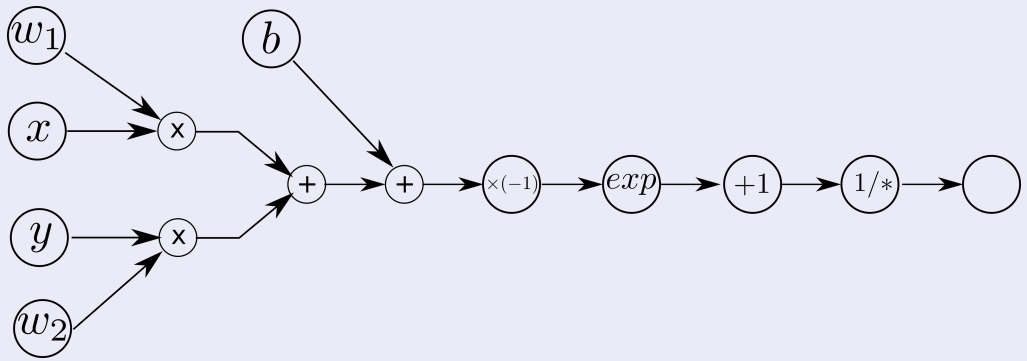
We can simply write:

$$\mathrm{g}(b + \sum_{i=1}^{p} w_i x_i) = \mathrm{g}(b + \mathbf{w}^T \mathbf{x})$$

# Contents

# Contents

# Computational graph

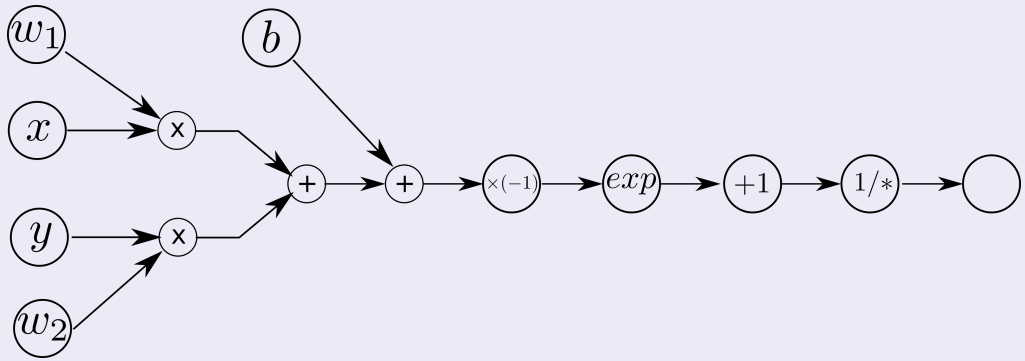



## Definition

A computational graph is an acyclic direct graph such that:

- A node is a mathematical operator
- To each edge is associated a value
- Each node can compute the values of its output edges from the values of its input edges
  - Nodes without input edges are *input nodes*. They represent the input values of the graph.
  - Similarly, output values can be held in the *output nodes*.

# Computational graph




- In this course, we will only consider *acyclic* computational graphs.
- Computing a *forward pass* through the graph means choosing its input values, and then progressively computing the values of all edges.

# Computational graph example

Computational graph of:

$$\sigma(w_1 x + w_2 y + b)$$

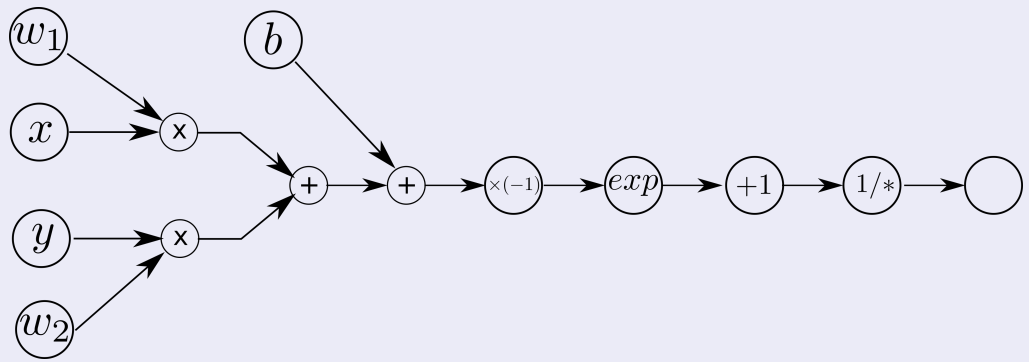
where $\sigma$ is the sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}}$

# Computational graph example

Computational graph of:

$$\sigma(w_1 x + w_2 y + b)$$

where $\sigma$ is the sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}}$

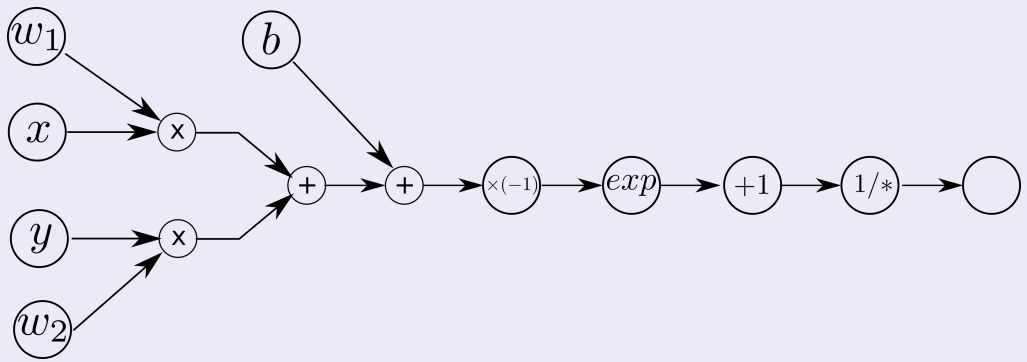


# Computational graph example

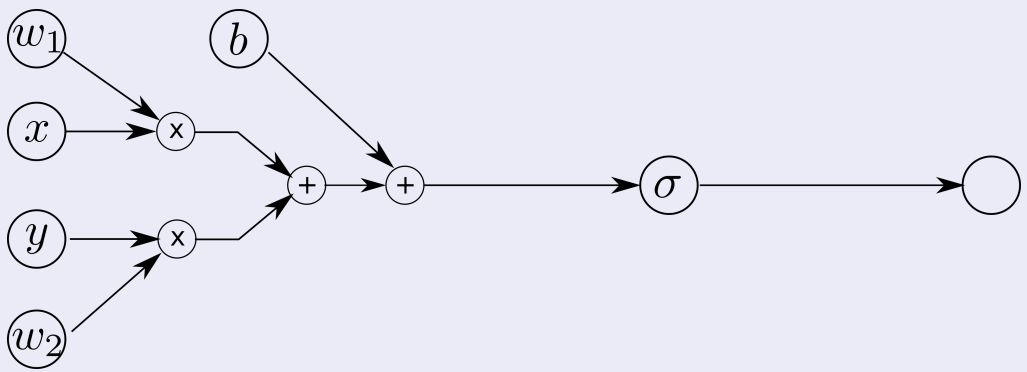
Computational graph of:

$$\sigma(w_1 x + w_2 y + b)$$

where $\sigma$ is the sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}}$




The graph can be represented at different levels of detail:



# Differentiable programming

**Definition**

Differentiable programming refers to the use of computational graphs such that all operators are differentiable. Automatic differentiation then allows to use gradient descent for optimization.

# Contents

# Neural network (NN)

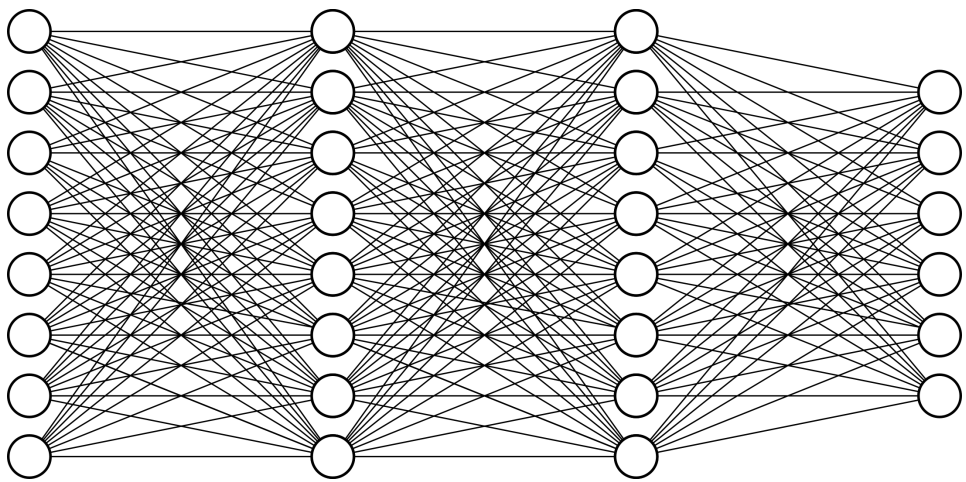
## Definitions

- An artificial neural network is a computational graph, where the nodes are artificial neurons
- The input layer is the set of neurons without incoming edges.
- The output layer is the set of neurons without outgoing edges.

# Feed-forward neural networks

## Definition

- A feed-forward neural networks is a NN without cycles
- Neurons are organized in layers
- Any layers other than input and output layers are called hidden layers



# Feed-forward neural networks

In the following of this course, except when otherwise specified, all NNs will be feed-forward. Indeed, this is the preferred type of NN for image processing.

## What about other architectures?

- Recurrent neural networks (RNN)
- Long short-term memory networks (LSTM)

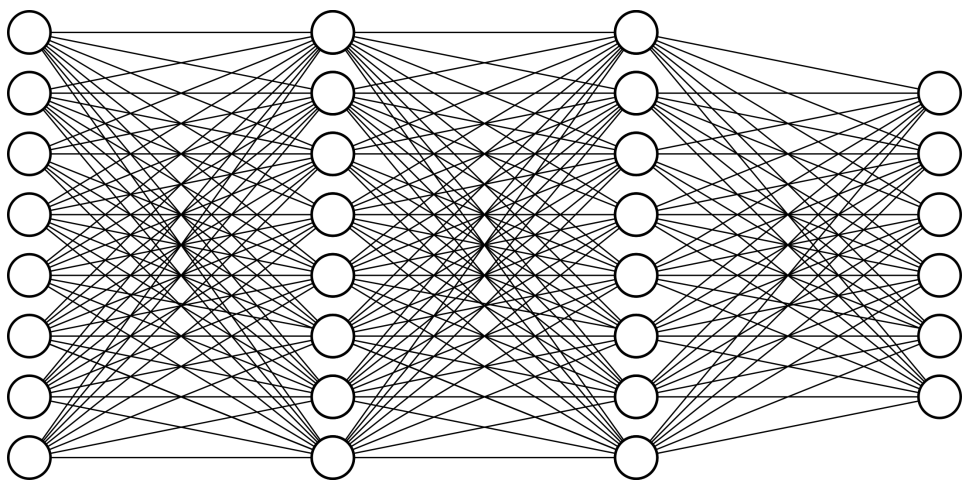
\+ More powerful than feed-forward NNs

\- Complex dynamics; more difficult to train

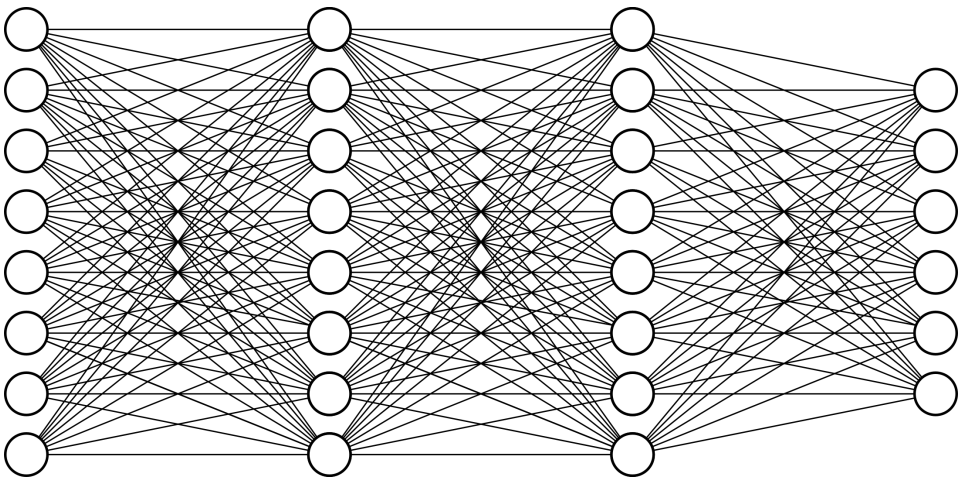
- Mainly used for processing temporal data

# Fully-connected layer

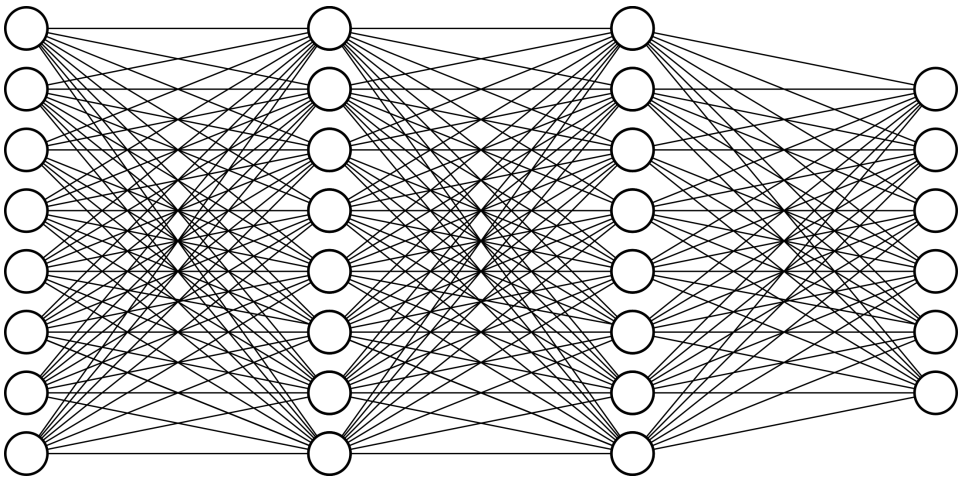
- A layer is said to be fully-connected (FC) if each of its neurons is connected to all the neurons of the previous layer
- If a FC layer contains $r$ neurons, and the previous layer $q$, then its weights are a 2D dimensional array (a matrix) of size $q \times r$



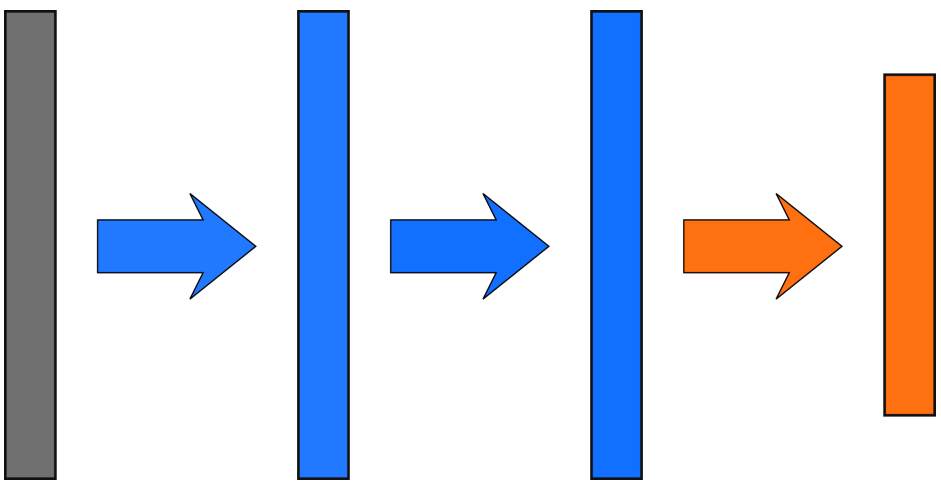
# Graphical representation of NNs



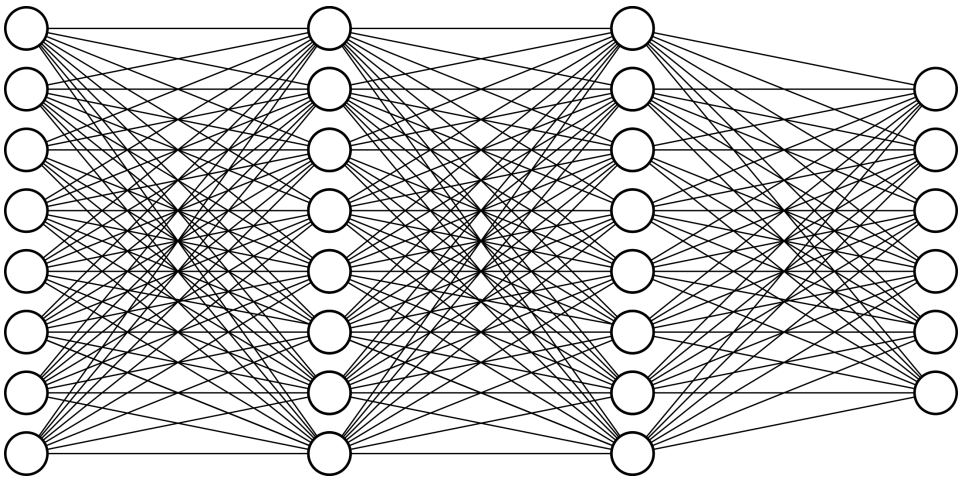
# Graphical representation of NNs

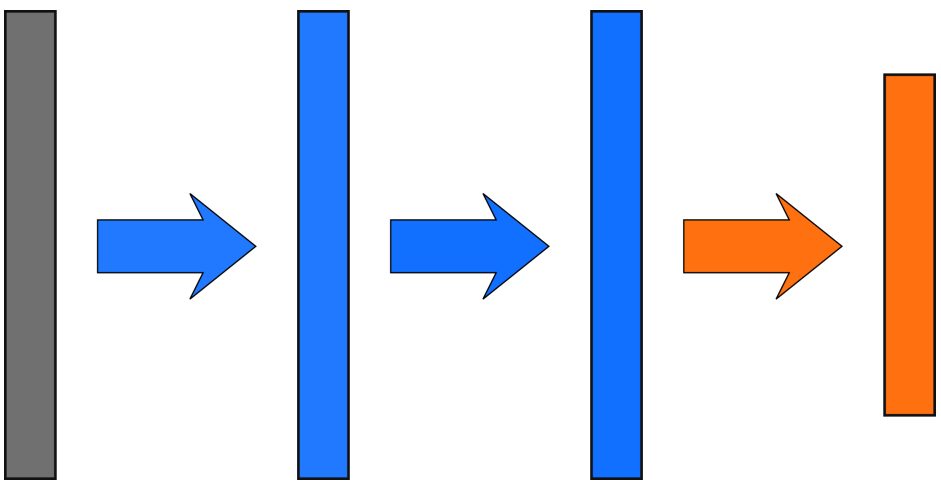


- Data is organized into arrays, linked with operators



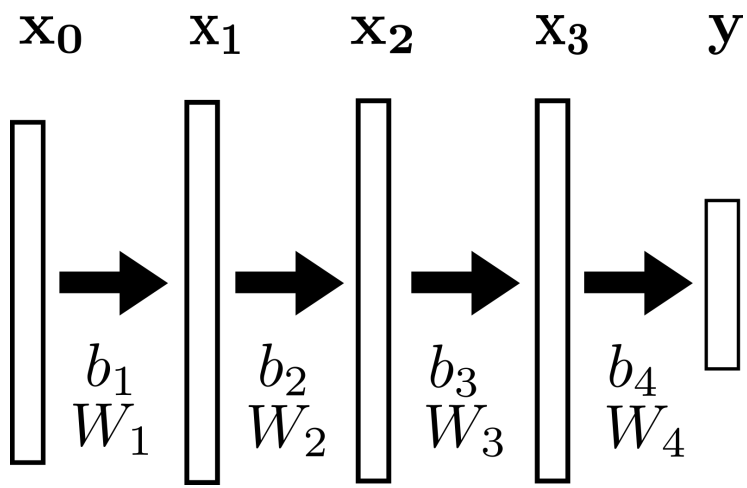
# Graphical representation of NNs





- Data is organized into arrays, linked with operators
- A layer corresponds to an operator between arrays.

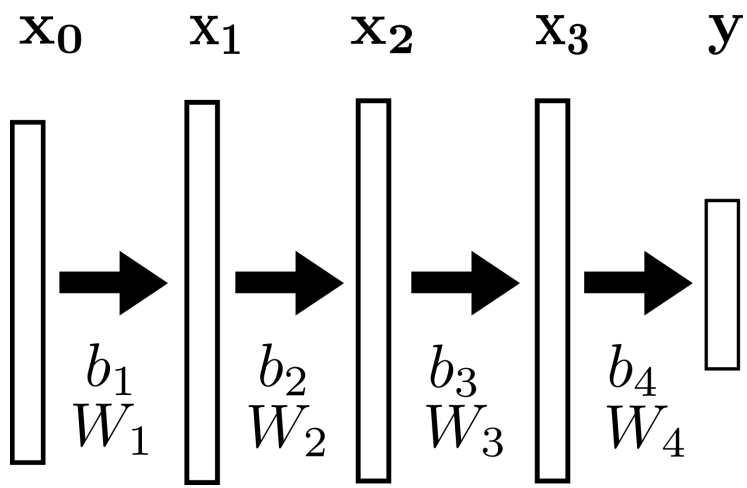
# The equations of a fully connected neural network




$$\mathbf{x}^i = \mathbf{g}_i(\mathbf{x}_{i-1}^t \mathbf{W}_i + \mathbf{b}_i),\ i = 1, 2, 3$$

$$\mathbf{y} = \mathbf{g}_4(\mathbf{x}_4^t \mathbf{W}_4 + \mathbf{b}_4)$$

# The equations of a fully connected neural network

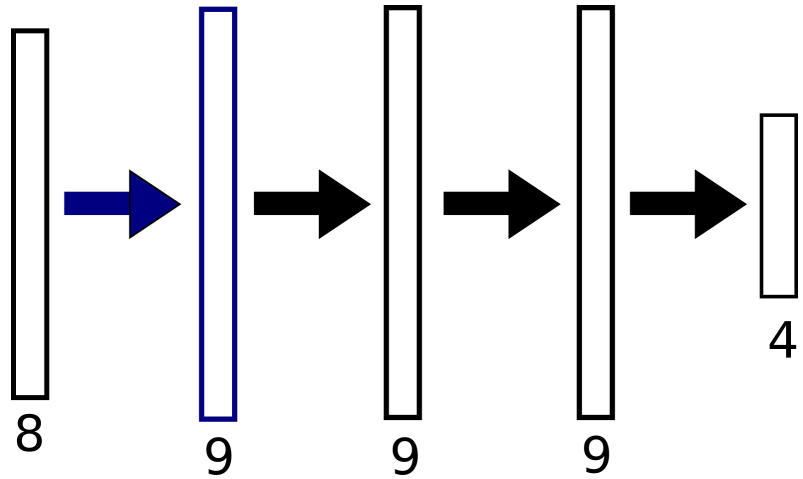



$$\mathbf{x}^i = \mathrm{g}_i(\mathbf{x}_{i-1}^t \mathbf{W}_i + \mathbf{b}_i),\ i = 1, 2, 3$$

$$\mathbf{y} = \mathrm{g}_4(\mathbf{x}_4^t \mathbf{W}_4 + \mathbf{b}_4)$$

What would happen if all activation functions $\mathrm{g}_i$ were equal to the identity function?

# Number of parameters

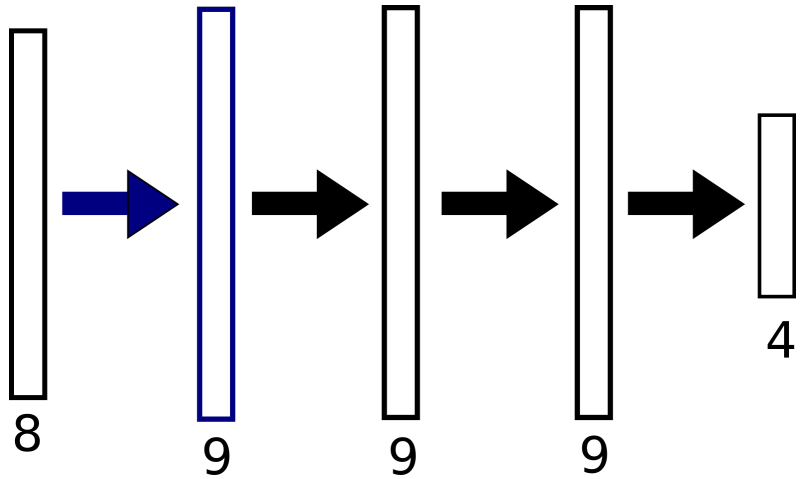



- How many parameters does the above network contain?

First layer:

- 64
- 72
- 81
- 90

# Number of parameters




- How many parameters does the above network contain?

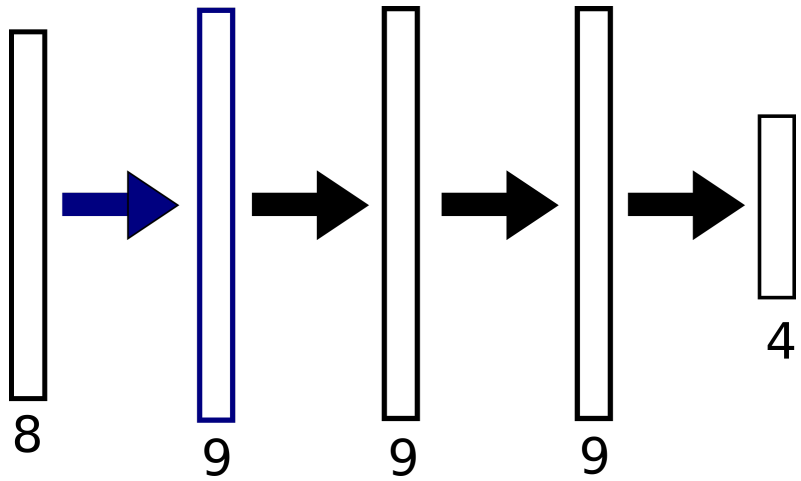
First layer:

- 64
- 72
- 81
- 90

Second and third layers:

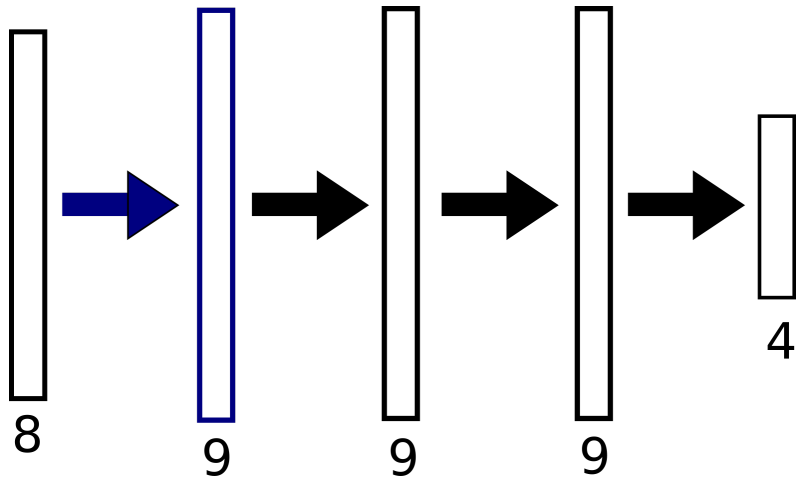
- 64
- 72
- 81
- 90

# Number of parameters



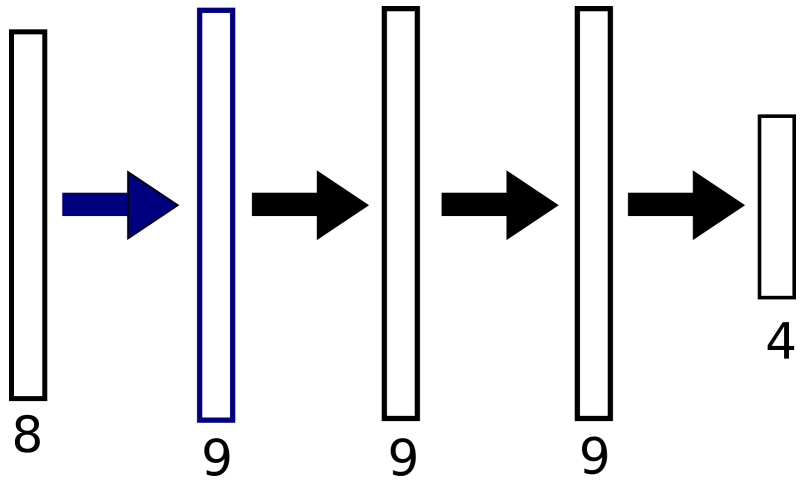

- How many parameters does the above network contain?

# Number of parameters



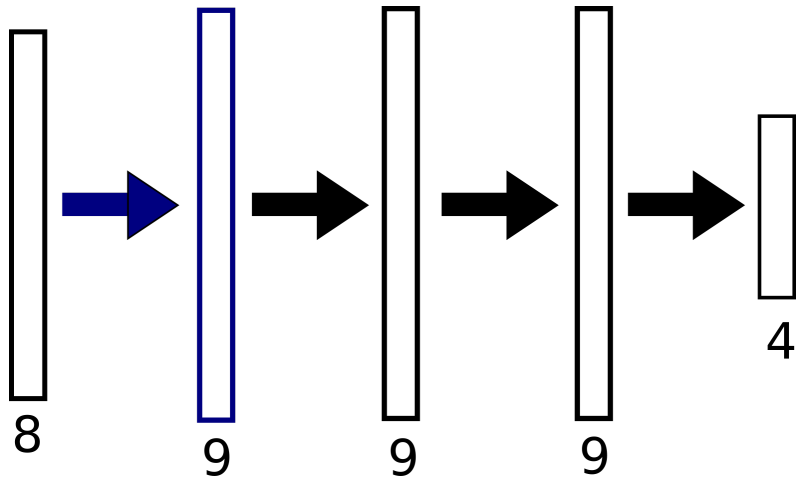

- How many parameters does the above network contain?
- First hidden layer:

# Number of parameters



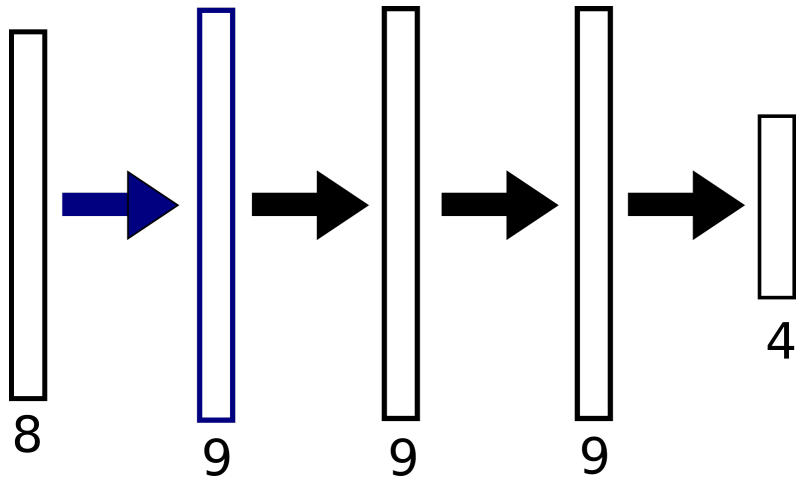

- How many parameters does the above network contain?
- First hidden layer:
  - $9$ neurons $\times 8$ neurons in the previous layer $+9$ biases $= 81$

# Number of parameters



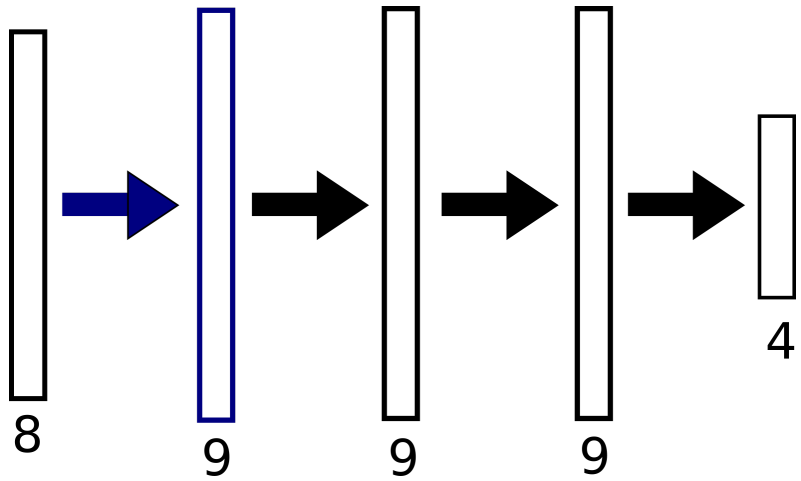

- How many parameters does the above network contain?
- First hidden layer:
  - $9$ neurons $\times 8$ neurons in the previous layer $+9$ biases $= 81$
- Second and third layers:

# Number of parameters



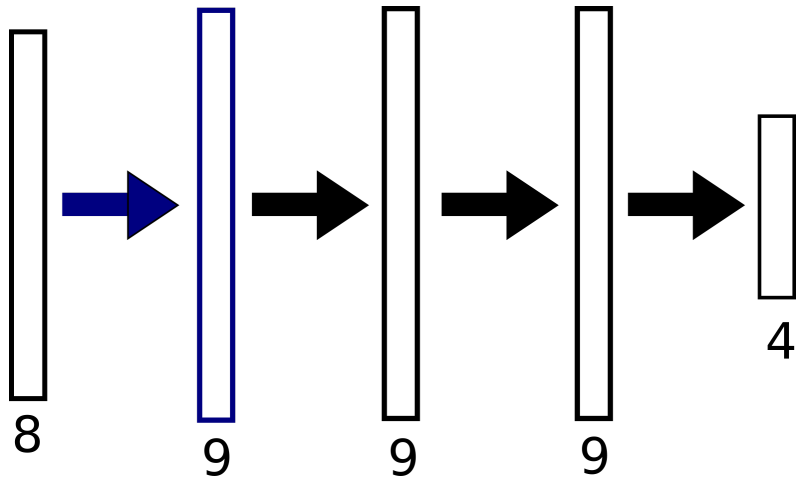

- How many parameters does the above network contain?
- First hidden layer:
  - $9$ neurons $\times 8$ neurons in the previous layer $+9$ biases $= 81$
- Second and third layers: $9 \times 9 + 9 = 90$

# Number of parameters



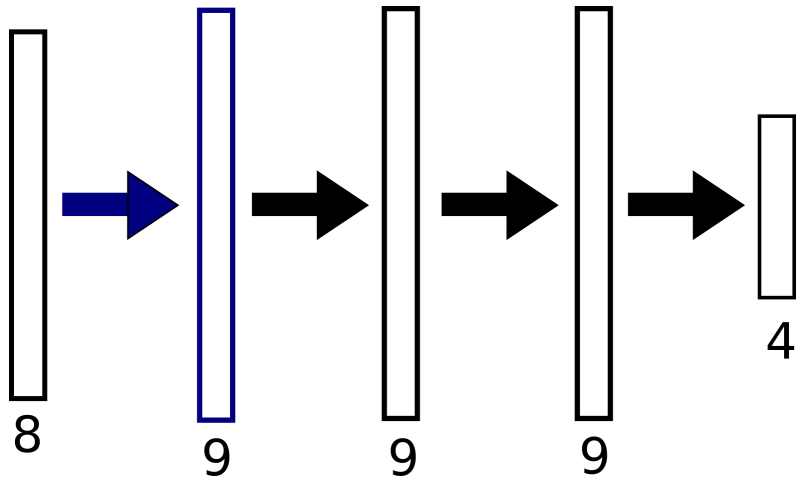

- How many parameters does the above network contain?
- First hidden layer:
  - $9$ neurons $\times 8$ neurons in the previous layer $+9$ biases $= 81$
- Second and third layers: $9 \times 9 + 9 = 90$
- Output layer:

# Number of parameters




- How many parameters does the above network contain?
- First hidden layer:
  - $9$ neurons $\times 8$ neurons in the previous layer $+9$ biases $= 81$
- Second and third layers: $9 \times 9 + 9 = 90$
- Output layer: $4 \times 9 + 4 = 40$

# Number of parameters




- How many parameters does the above network contain?
- First hidden layer:
  - $9$ neurons $\times 8$ neurons in the previous layer $+9$ biases $= 81$
- Second and third layers: $9 \times 9 + 9 = 90$
- Output layer: $4 \times 9 + 4 = 40$
- Total: $301$ parameters

# Batch processing

In a learning context, one may want to process $n$ vectors of length $p$ at the same time. They can be grouped into a matrix $\mathbf{X}$ of size $n \times p$. The $n$ corresponding outputs $\mathbf{y}_i$ can also be grouped into a matrix $\mathbf{Y}$. The resulting equations are:

$$\mathbf{X}_i = \mathrm{g}_i(\mathbf{X}_{i-1}\mathbf{W}_i + \mathbf{b}_i),\ i = 1, 2, 3$$

$$\mathbf{Y} = \mathrm{g}_4(\mathbf{X}_4\mathbf{W}_4 + \mathbf{b}_4)$$

This can accelerate processing thanks to hardware architectures such a Graphical Processing Units (GPUs) but can also play an important role in optimization.

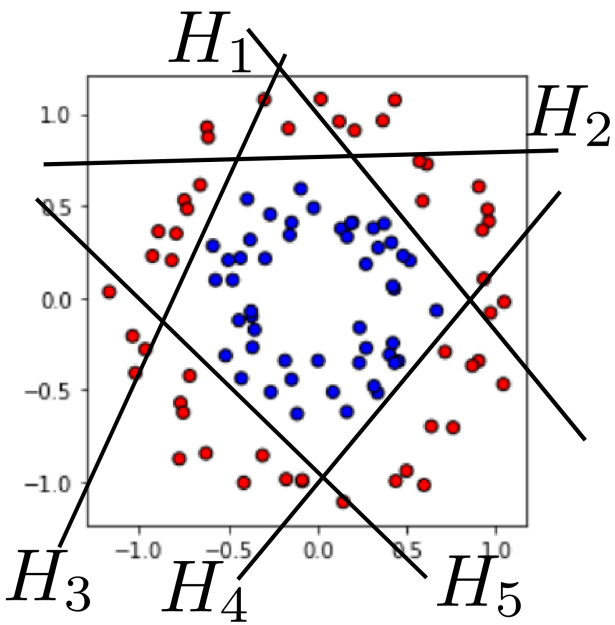
# From neurons to arrays

- Neurons are organized into arrays (0-D, 1-D, 2-D, 3-D ...)
- Artificial neural networks can be seen as computational graphs processing arrays

# Contents

# Universal approximation theorem

- We have previously seen that a neuron can be used as a linear classifier and that combining several of them one can build complex classifiers
- We will see that this observation can be generalized



# Universal approximation theorem

Let $f$ be a continuous real-valued function of $[0,1]^p$ ($p \in \mathbb{N}^*$) and $\epsilon$ a strictly positive real. Let g be a non-constant, increasing, bounded real function (*the activation function*).
Then there exists an integer $q$, real vectors $\{\mathbf{w}_i\}_{1 \leq i \leq q}$ of $\mathbb{R}^p$, and reals $\{b_i\}_{1 \leq i \leq q}$ and $\{v_i\}_{1 \leq i \leq q}$ such that for all $\mathbf{x}$ in $[0,1]^p$:

$$\left| f(\mathbf{x}) - \sum_{i=1}^{q} v_i \mathrm{g}(\mathbf{w}_i \mathbf{x} + b_i) \right| < \epsilon$$

A first version of this theorem, using sigmoidal activation functions, was proposed by [Cybenko, 1989]. The version above was demonstrated by [Hornik, 1991].

# Universal approximation theorem: what does it mean?

$$\left| f(\mathbf{x}) - \sum_{i=1}^{q} v_i \mathrm{g}(\mathbf{w}_i \mathbf{x} + b_i) \right| < \epsilon$$

This means that function $f$ can be approximated with a neural network containing:

- an input layer of size $p$;
- a hidden layer containing $q$ neurons with activation function g, weights $\mathbf{w}_i$ and biases $b_i$;
- an output layer containing a single neuron, with weights $v_i$ (and an identity activation function).

# Universal approximation theorem in practice

- The number of neurons increases very rapidly with the complexity of the function
- Empirical evidence has shown that multi-layer architectures give better results
- The theorem does not say how to optimize the architecture or the parameters for a given learning task

# Universal approximation theorem in practice

- The number of neurons increases very rapidly with the complexity of the function
- Empirical evidence has shown that multi-layer architectures give better results
- The theorem does not say how to optimize the architecture or the parameters for a given learning task

A NN can potentially have a lot of parameters. How can we set them?

# Contents

# Introduction

- We have seen that NNs have a lot of potential. However, how can the parameters $\boldsymbol{\theta} = (\mathbf{W}_i, \mathbf{b}_i)$ be set?
- What is our objective ?

# Supervised learning problem

We recall that our training set contains $n$ samples:

$$(\mathbf{x}_i, y_i) \in \mathbb{R}^p \times \mathcal{Y}$$

Where $\mathcal{Y} = \mathbb{R}$ in the regression case and $\mathcal{Y} = \{0, 1\}$ in the binary classification case.

# Supervised learning problem

We recall that our training set contains $n$ samples:

$$(\mathbf{x}_i, y_i) \in \mathbb{R}^p \times \mathcal{Y}$$

Where $\mathcal{Y} = \mathbb{R}$ in the regression case and $\mathcal{Y} = \{0, 1\}$ in the binary classification case.
We choose a loss function $l$ and a family $f_{\boldsymbol{\theta}}$ of functions from $\mathbb{R}^p$ into $\mathbb{R}$, depending on a set of parameters $\boldsymbol{\theta}$, and find the value $\boldsymbol{\theta}^*$ of $\boldsymbol{\theta}$ that minimizes:

$$\frac{1}{n} \sum_{i=1}^{n} l(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i)$$

# Supervised learning problem

We recall that our training set contains $n$ samples:

$$(\mathbf{x}_i, y_i) \in \mathbb{R}^p \times \mathcal{Y}$$

Where $\mathcal{Y} = \mathbb{R}$ in the regression case and $\mathcal{Y} = \{0, 1\}$ in the binary classification case.
We choose a loss function $l$ and a family $f_{\boldsymbol{\theta}}$ of functions from $\mathbb{R}^p$ into $\mathbb{R}$, depending on a set of parameters $\boldsymbol{\theta}$, and find the value $\boldsymbol{\theta}^*$ of $\boldsymbol{\theta}$ that minimizes:

$$\frac{1}{n} \sum_{i=1}^{n} l(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i)$$

For the sake of simplicity, we have dropped the regularization term.

# Contents

# Choosing a loss function

- The choice of the loss function depends on the type of problem (regression or classification) and is tightly linked to the application.
- In the following slides we will see two classical loss functions used respectively for regression and classification problems: the squared error loss and the cross-entropy loss.

# The standard loss for regression problems: Squared error loss

In the regression case, we have $\mathcal{Y} = \mathbb{R}$.

**Squared error loss**

$$l(f_{\boldsymbol{\theta}}(x), y) = (f_{\boldsymbol{\theta}}(x) - y)^2$$

# Binary cross-entropy

In the simplest classification case, we have $\mathcal{Y} = \{0, 1\}$.

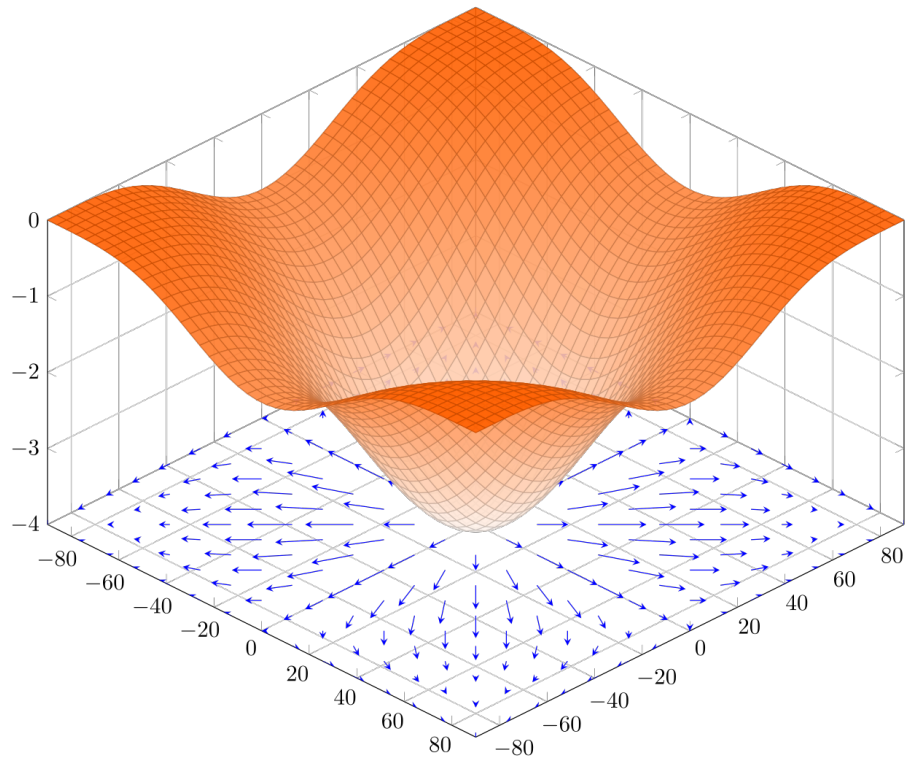
## Binary cross-entropy loss

$$l(f_{\boldsymbol{\theta}}(x), y) = -y \log(f_{\boldsymbol{\theta}}(x)) - (1 - y) \log(1 - f_{\boldsymbol{\theta}}(x))$$

- For this expression to be mathematically sound, $f_{\boldsymbol{\theta}}(x)$ must belong to $]0, 1[$. In practice, in the case of NN, this can be achieved by using a sigmoid as last activation.
- Note that the expression above is equivalent to:

$$l(f_{\boldsymbol{\theta}}(x), y) = \begin{cases} -\log(1 - f_{\boldsymbol{\theta}}(x)) & \text{if } y = 0 \\ -\log(f_{\boldsymbol{\theta}}(x)) & \text{if } y = 1 \end{cases}$$

# How to minimize the loss?




## Definition: gradient

Let $L$ be a differentiable function from $\mathbb{R}^n$ into $\mathbb{R}$. Its gradient $\nabla L$ is:

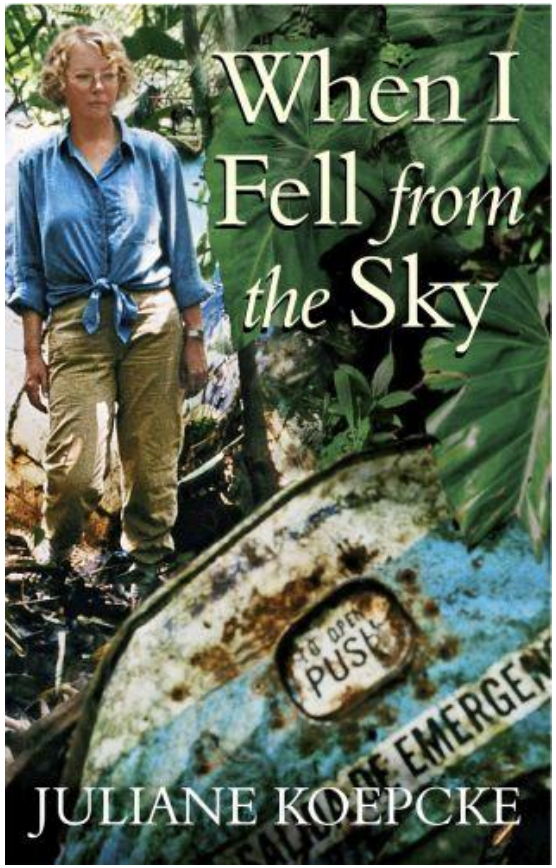
$$\nabla L(x) = \begin{pmatrix} \frac{\partial L}{\partial \mathbf{x}_1}(x) \\ \vdots \\ \frac{\partial L}{\partial \mathbf{x}_n}(x) \end{pmatrix}$$

Credits: By MartinThoma, CC0, https://commons.wikimedia.org/

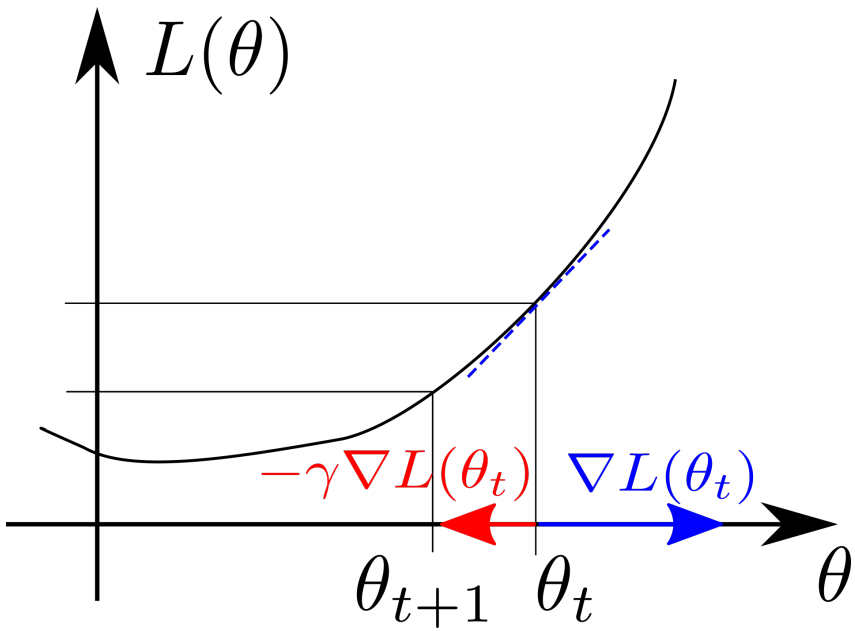
# Contents

# Gradient descent can save your life...

# Gradient descent can save your life...



# Gradient descent in the scalar case




$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla L(\boldsymbol{\theta}_t)$$

# Gradient descent

## Definition

Gradient descent is an optimization algorithm. For a differentiable function $L$, a positive real $\eta$ (the learning rate) and a starting point $\boldsymbol{\theta}_0$, it computes a sequence of values:

$$\forall t \in \mathbb{N} : \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla L(\boldsymbol{\theta}_t)$$

## Property

For a given $t$, if $\eta$ is small enough, then:

$$L(\boldsymbol{\theta}_{t+1}) \leq L(\boldsymbol{\theta}_t)$$

Gradient descent is an essential tool in optimization.

# Gradient descent: stopping criteria

In practice:

$$\forall t \in [0, \ldots, E-1]: \quad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla L(\boldsymbol{\theta}_t)$$

- Choose $E$ (the number of epochs) based on experience
- Track the quality of the model using a validation dataset and stop when the validation loss does not improve

# Towards stochastic gradient descent

The loss function we initially defined depends on the whole training set:

$$L(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} l(y_i, f(\mathbf{x}_i, \boldsymbol{\theta}))$$

# Towards stochastic gradient descent

The loss function we initially defined depends on the whole training set:

$$L(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} l(y_i, f(\mathbf{x}_i, \boldsymbol{\theta}))$$

- If $n$ is very large, computing $L$ is impractical.

# Towards stochastic gradient descent

The loss function we initially defined depends on the whole training set:

$$L(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} l(y_i, f(\mathbf{x}_i, \boldsymbol{\theta}))$$

- If $n$ is very large, computing $L$ is impractical.
- A computation on the whole training set leads to a single update of the model parameters - convergence can therefore be slow.

# Stochastic gradient descent

In stochastic gradient descent, the parameters are updated for each sample $i$.

- First, the loss is computed

$$L(\boldsymbol{\theta}_t) = l(y_i, f(\mathbf{x}_i, \boldsymbol{\theta}_t))$$

# Stochastic gradient descent

In stochastic gradient descent, the parameters are updated for each sample $i$.

- First, the loss is computed

$$L(\boldsymbol{\theta}_t) = l(y_i, f(\mathbf{x}_i, \boldsymbol{\theta}_t))$$

- The gradient $\nabla L(\boldsymbol{\theta}_t)$ is computed and

# Stochastic gradient descent

In stochastic gradient descent, the parameters are updated for each sample $i$.

- First, the loss is computed

$$L(\boldsymbol{\theta}_t) = l(y_i, f(\mathbf{x}_i, \boldsymbol{\theta}_t))$$

- The gradient $\nabla L(\boldsymbol{\theta}_t)$ is computed and
- Finally the parameters are updated:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla L(\boldsymbol{\theta}_t)$$

# Stochastic gradient descent

In stochastic gradient descent, the parameters are updated for each sample $i$.

- First, the loss is computed

$$L(\boldsymbol{\theta}_t) = l(y_i, f(\mathbf{x}_i, \boldsymbol{\theta}_t))$$

- The gradient $\nabla L(\boldsymbol{\theta}_t)$ is computed and
- Finally the parameters are updated:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla L(\boldsymbol{\theta}_t)$$

- Note that the learning rate $\eta$ can have a different value than in classic gradient descent.

# Mini-batch processing

- One can (and often does) choose and intermediate solution between the full gradient and the stochastic gradient: mini-batch gradient.

# Mini-batch processing

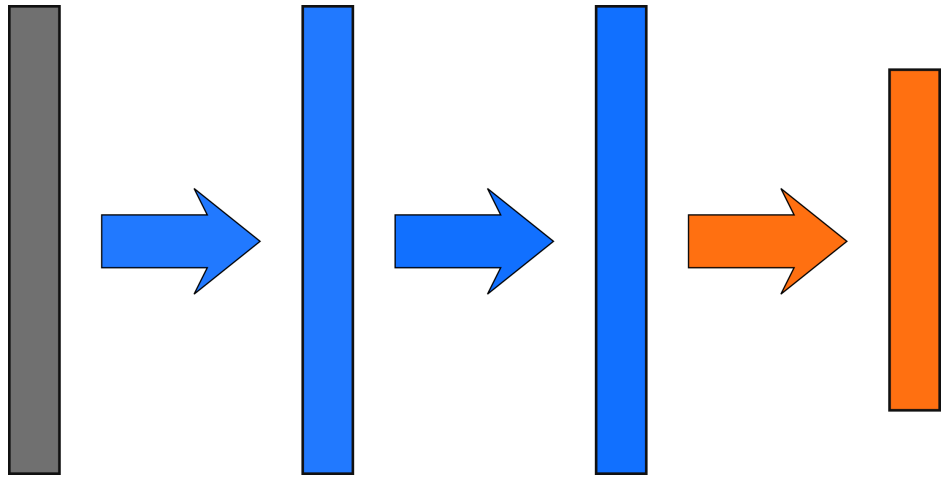
- One can (and often does) choose and intermediate solution between the full gradient and the stochastic gradient: mini-batch gradient.
- The training database is then separated into subsets containing $m$ samples ($m < n$).

# Mini-batch processing

- One can (and often does) choose and intermediate solution between the full gradient and the stochastic gradient: mini-batch gradient.
- The training database is then separated into subsets containing $m$ samples ($m < n$).
- This has a regularization effect on the optimization with respect to the stochastic gradient and speeds up computation thanks to the vectorization capacity of hardware architectures such as GPUs.

# Contents

# Gradient descent applied to neural networks



- In the case of neural networks, the loss $L$ depends on each parameter $\boldsymbol{\theta}_i$ via the composition of several functions.
- Analytical derivation is possible, but complex - and has to be re-computed when the network architecture is modified
- Using the chain rule theorem leads to an efficient solution: backpropagation.

# Chain rule theorem

Let $f_1$ and $f_2$ be two differentiable real functions ($\mathbb{R} \to \mathbb{R}$). Then for all $x$ in $\mathbb{R}$:

$$(f_2 \circ f_1)'(x) = (f_2' \circ f_1)(x).f_1'(x)$$

## Leibniz notation

Let us introduce variables $x$, $y$ and $z$:

$$x \xrightarrow{f_1} y \xrightarrow{f_2} z$$

Then:

$$\frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}y} \cdot \frac{\mathrm{d}y}{\mathrm{d}x}$$

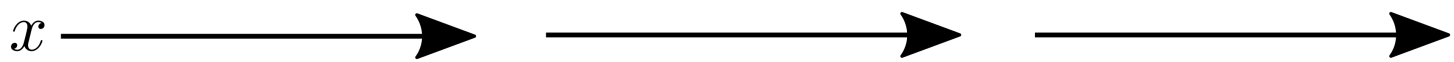
# The backpropagation algorithm

- The backpropagation algorithm is used in a neural network to efficiently compute the partial derivatives of the loss with respect to each parameter of the network.
- One can trace the origins of the method to the sixties
- It was first applied to NN in the eighties
  [Werbos, 1982, LeCun, 1985]

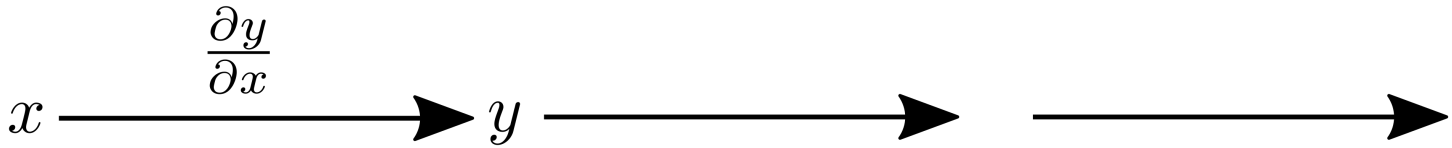
# The backpropagation algorithm: intuition

- Given a computational graph, the main idea is to compute the local derivatives during a forward pass
- Then, during a backward pass, the partial derivatives of the loss with respect to each parameter are computed

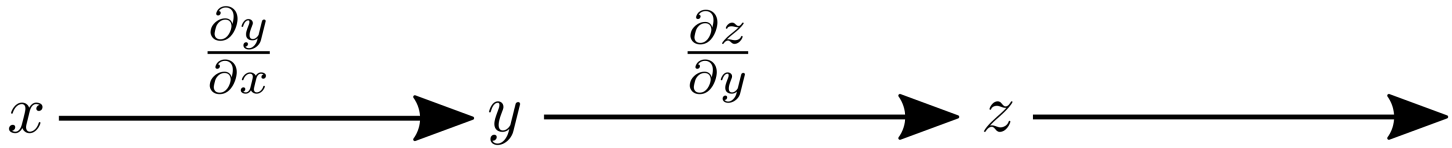
# Simple backpropagation example



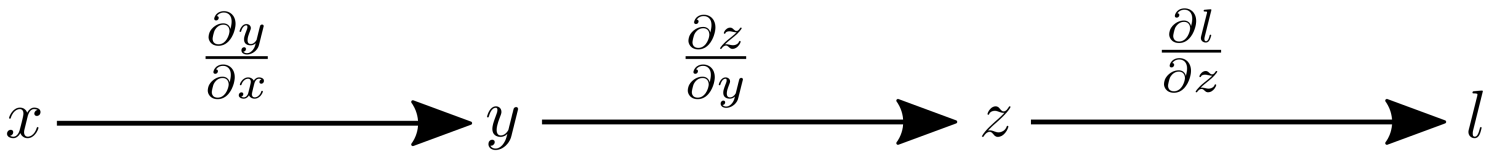
# Simple backpropagation example



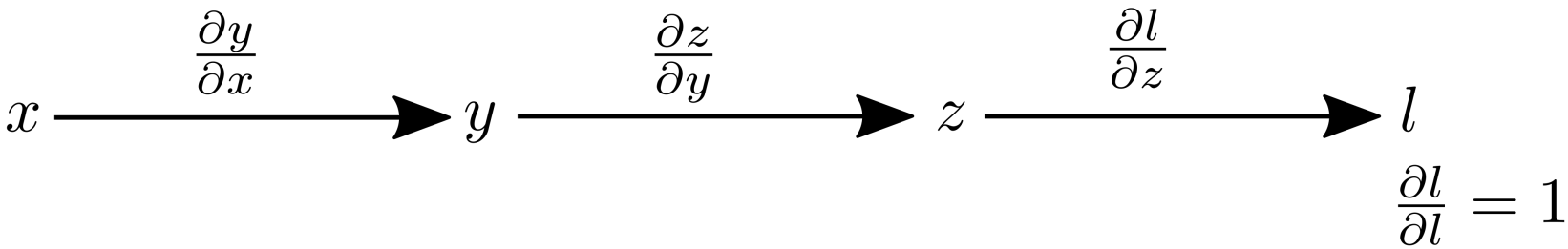
# Simple backpropagation example



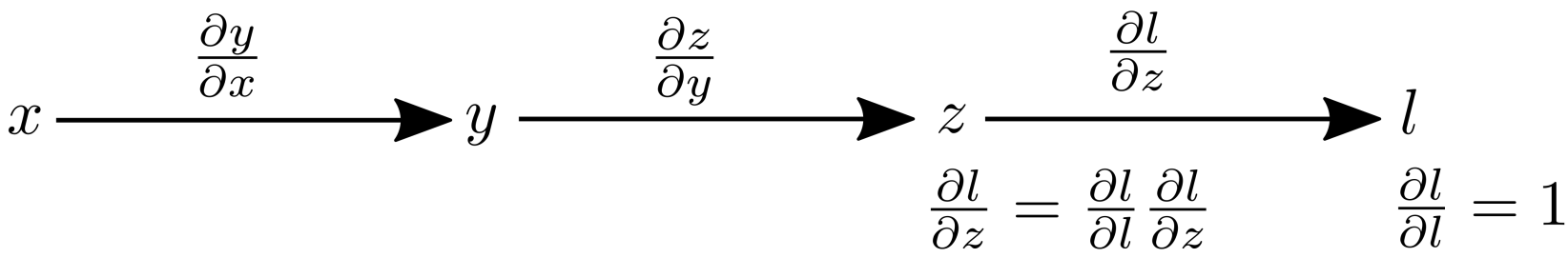
# Simple backpropagation example



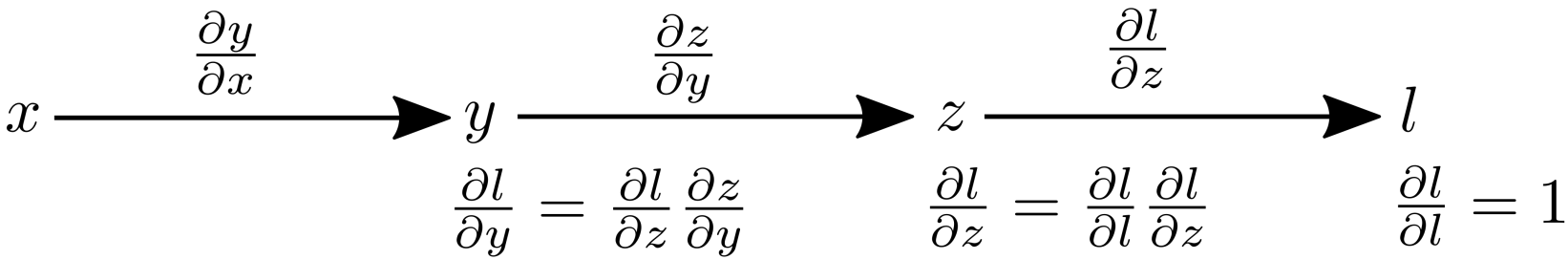
# Simple backpropagation example



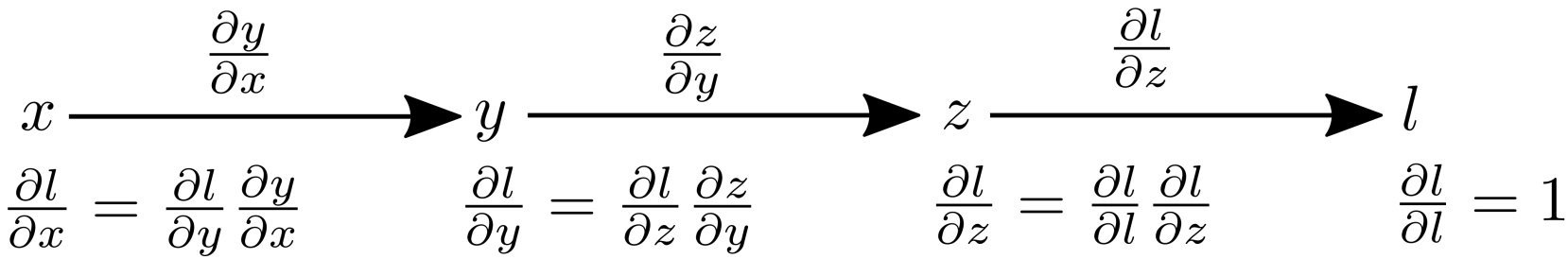
# Simple backpropagation example



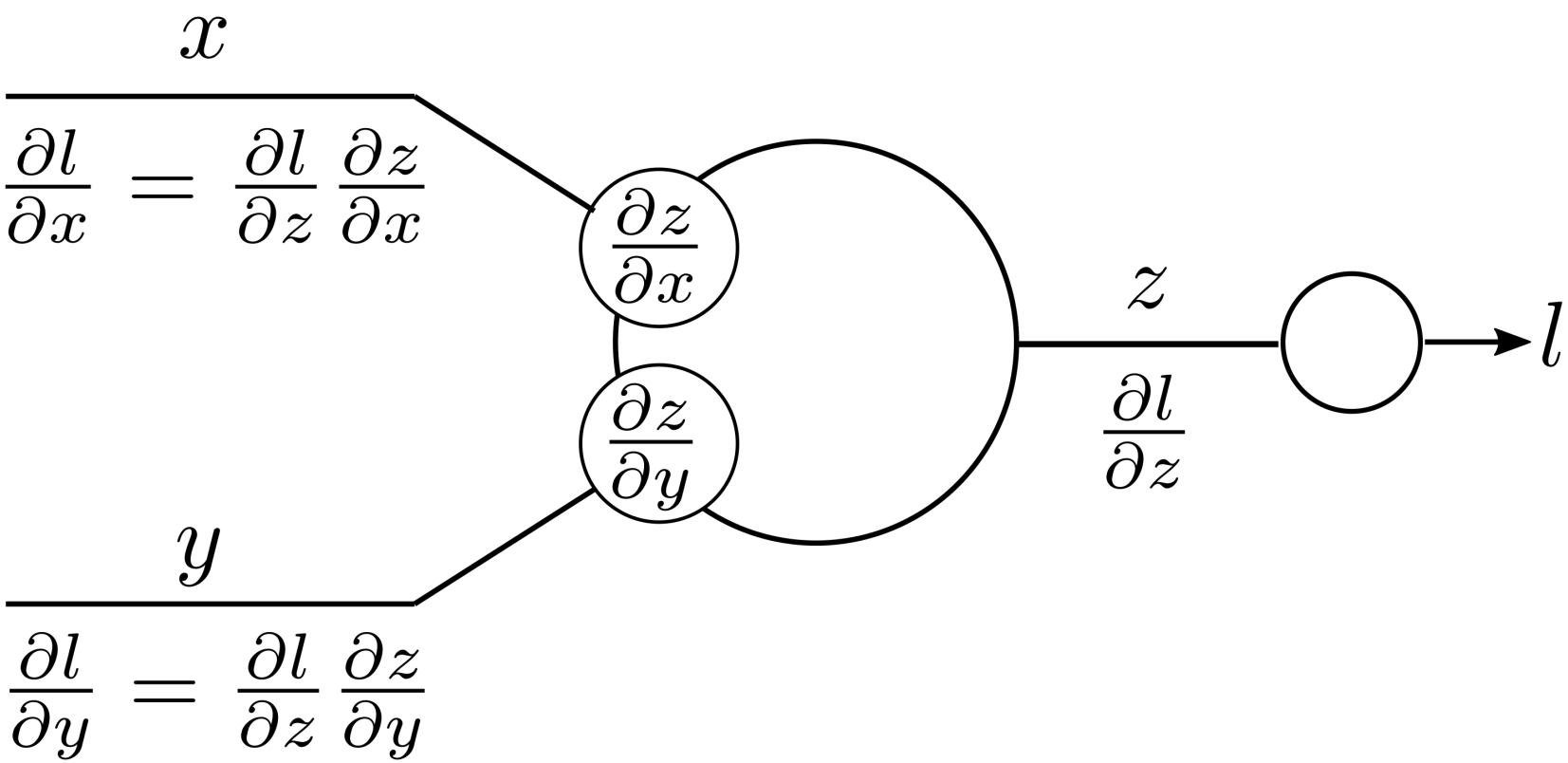
# Simple backpropagation example



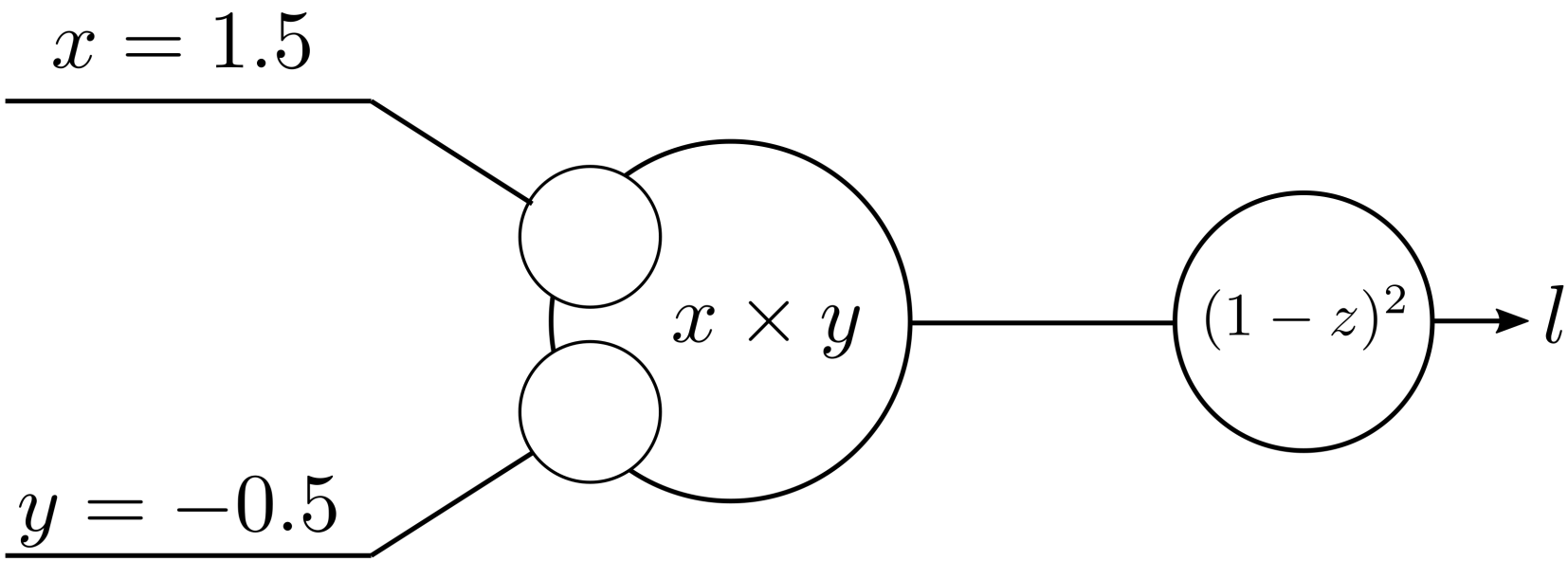
# Simple backpropagation example



# Backpropation through a neuron



# Exercise



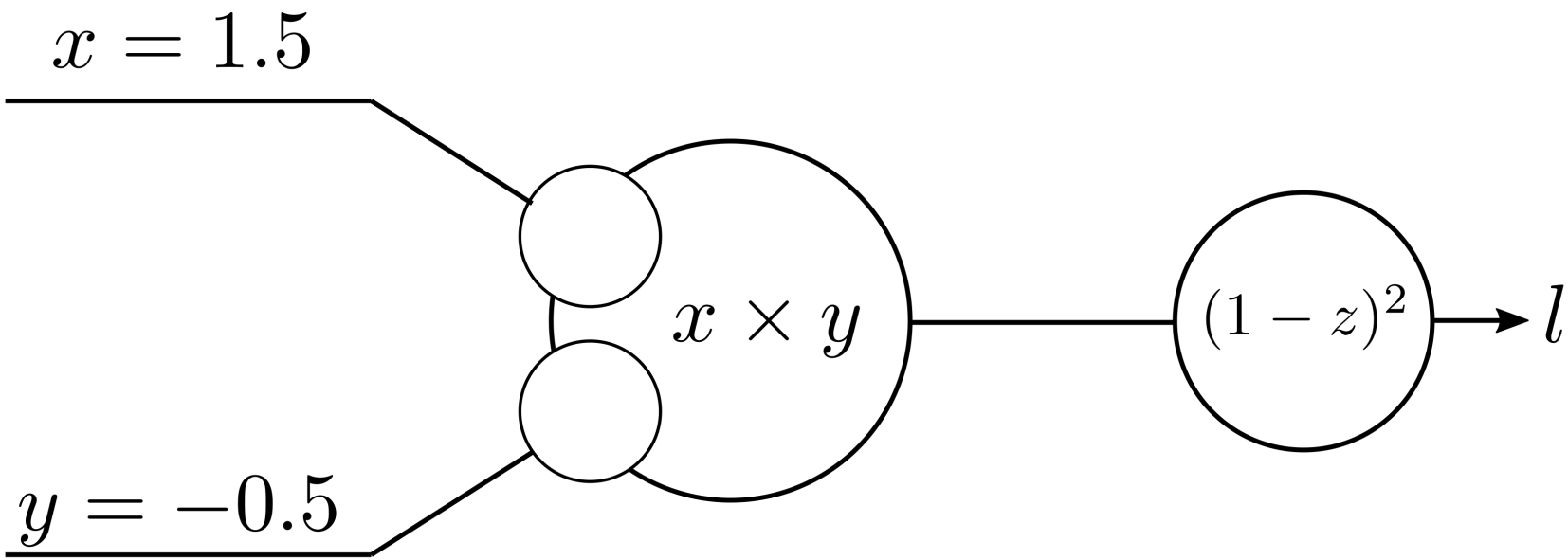

Gradient local du haut:

- $1.5$
- $-0.5$
- autre valeur

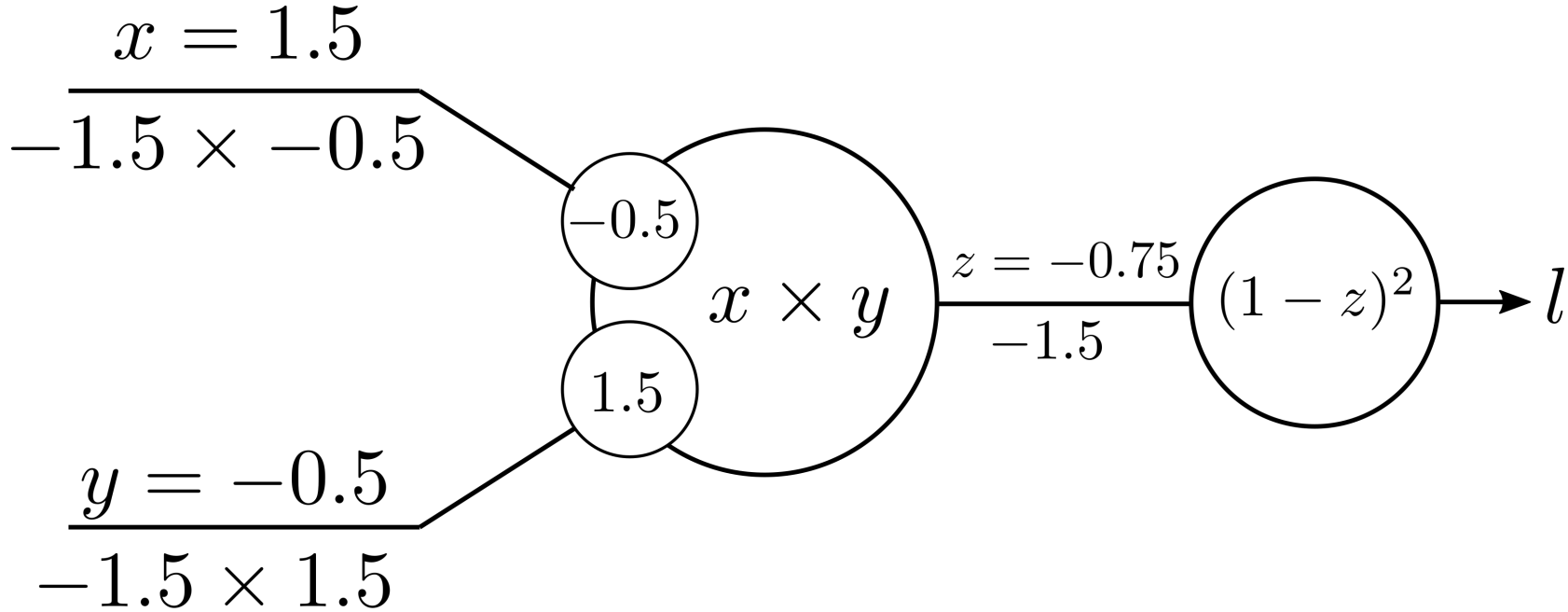
Gradient local du bas:

- $1.5$
- $-0.5$
- autre valeur

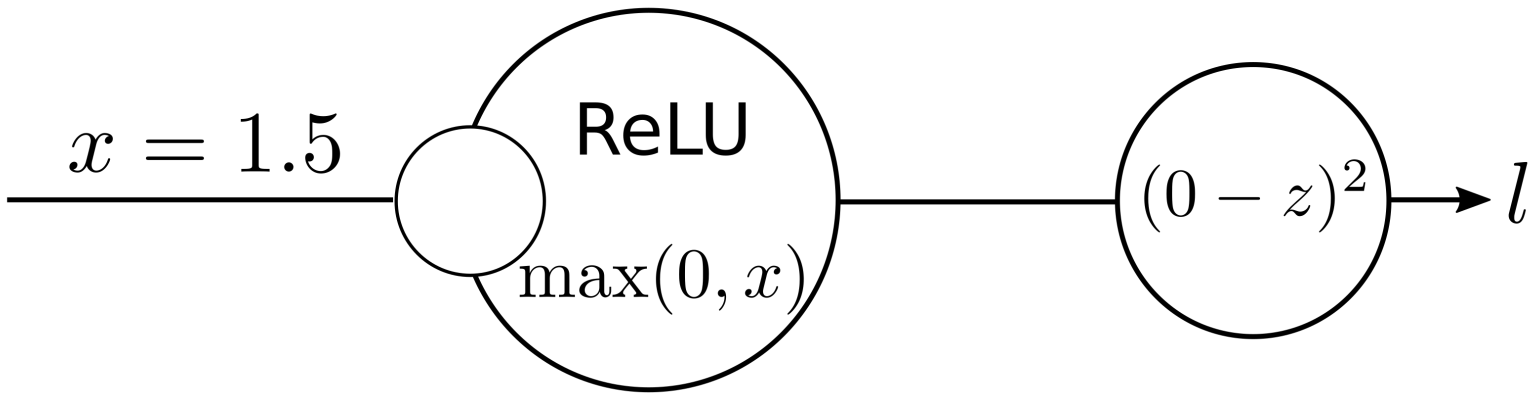
# Exercise



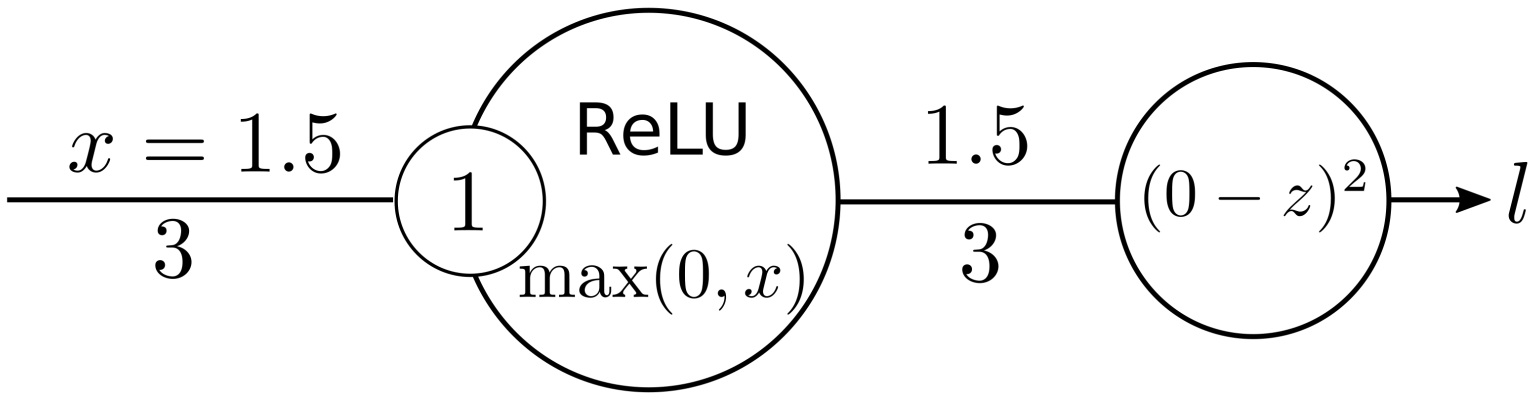
# Exercise: solution



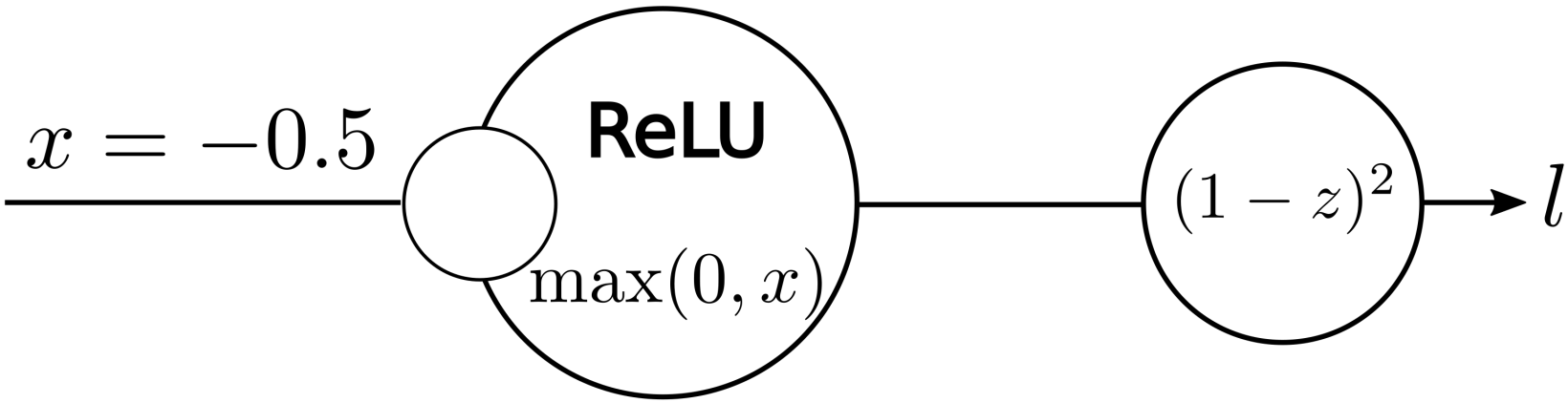
# Exercise



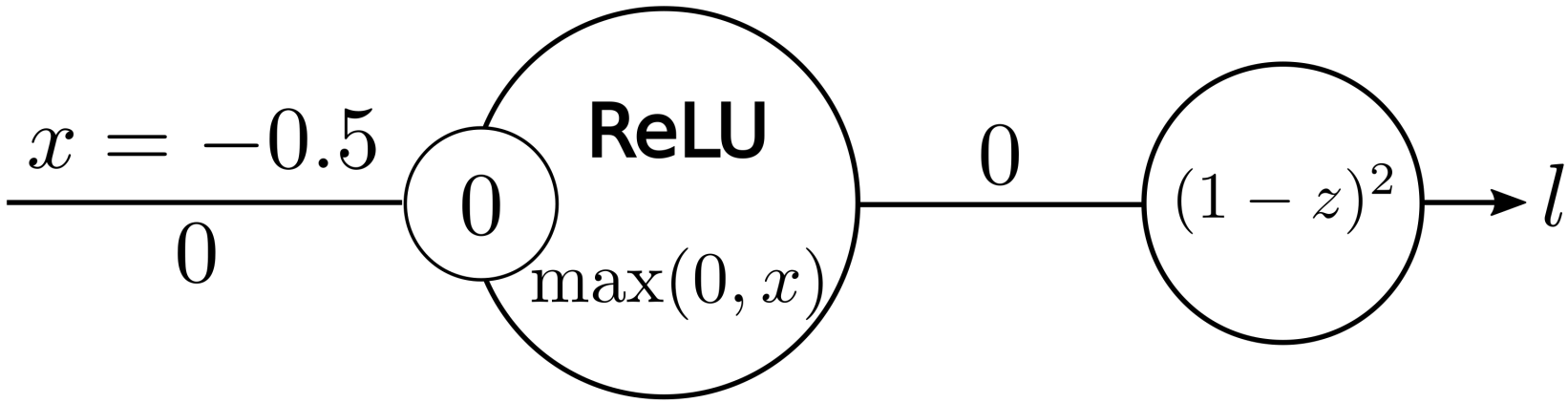
# Exercise: solution



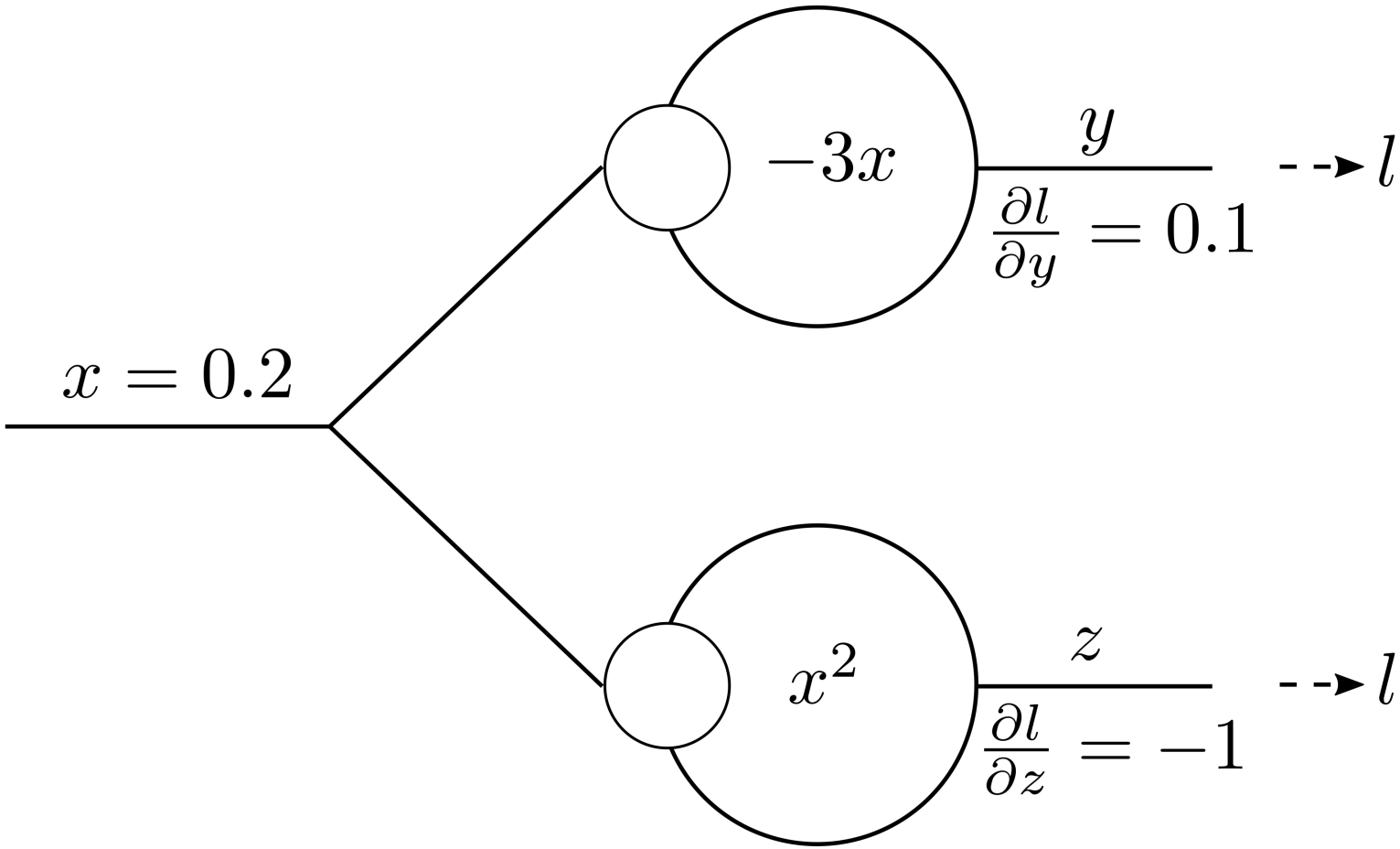
# Exercise



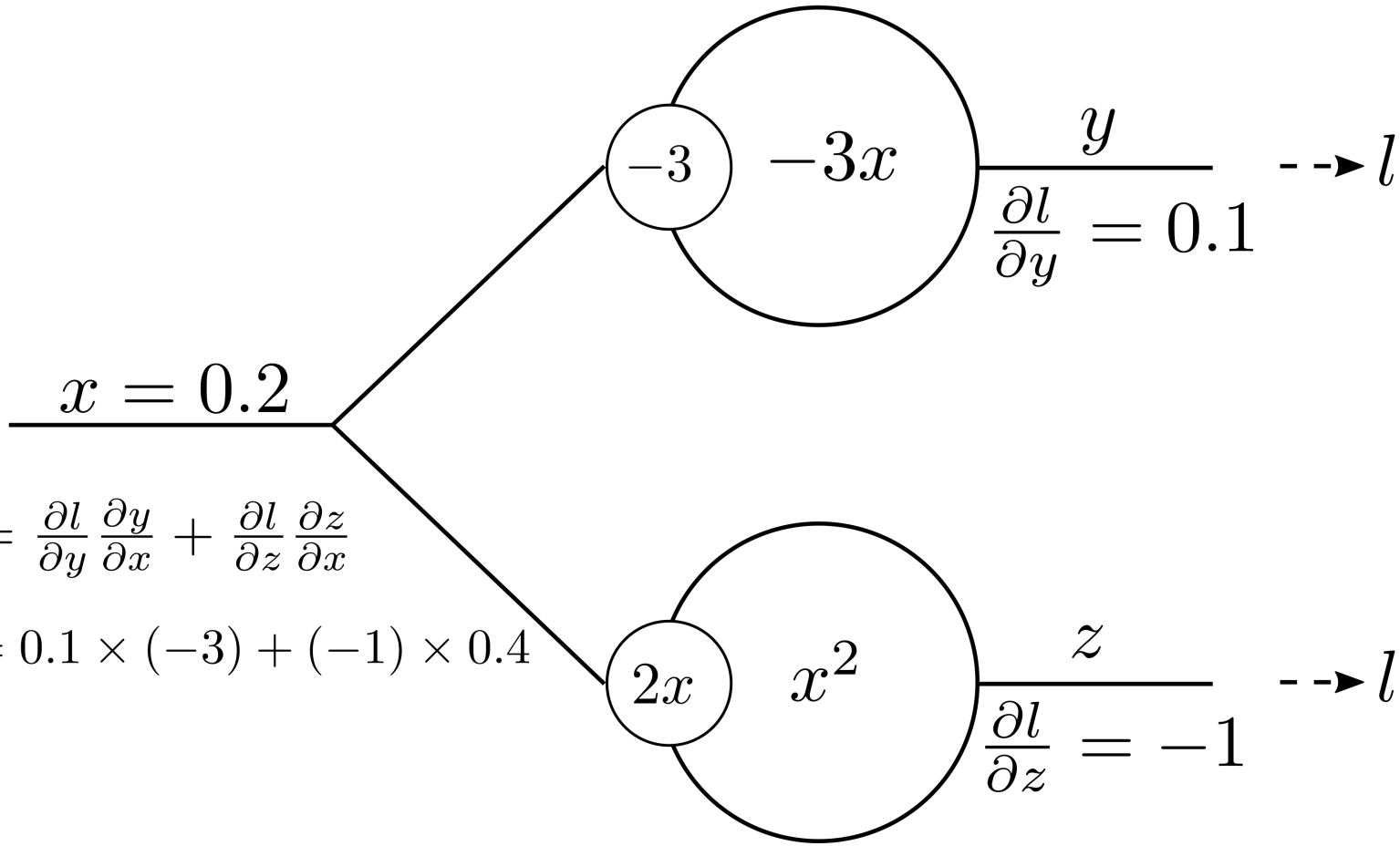
# Exercise: solution



# Exercise


$x = 0.2$
$-3x$
$y$
$\frac{\partial l}{\partial y} = 0.1$
$l$
$x^2$
$z$
$\frac{\partial l}{\partial z} = -1$
$l$

# Exercise: solution




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

$$\frac{\partial l}{\partial x} = 0.1 \times (-3) + (-1) \times 0.4$$

# Vector calculus

- $L$ and $V$ are differentiable functions.

$$L : \mathbb{R}^n \longrightarrow \mathbb{R}$$
$$\mathbf{x} \longmapsto L(\mathbf{x})$$

**Gradient**

$$\nabla_{\mathbf{x}} L = \frac{\partial L}{\partial \mathbf{x}} = (\frac{\partial L}{\partial x_1}, \ldots, \frac{\partial L}{\partial x_n})$$

# Vector calculus

- $L$ and $V$ are differentiable functions.

$$L : \mathbb{R}^n \longrightarrow \mathbb{R}$$
$$\mathbf{x} \longmapsto L(\mathbf{x})$$

$$V : \mathbb{R}^p \longrightarrow \mathbb{R}^q$$
$$\mathbf{y} \longmapsto V(\mathbf{y})$$

**Gradient**

$$\nabla_{\mathbf{x}} L = \frac{\partial L}{\partial \mathbf{x}} = \left(\frac{\partial L}{\partial x_1}, \ldots, \frac{\partial L}{\partial x_n}\right)$$

**Jacobian**

$$J(V) = \frac{\partial V}{\partial \mathbf{y}} = \begin{pmatrix} \frac{\partial V_1}{\partial y_1} & \cdots & \frac{\partial V_1}{\partial y_p} \\ \vdots & \ddots & \vdots \\ \frac{\partial V_q}{\partial y_1} & \cdots & \frac{\partial V_q}{\partial y_p} \end{pmatrix}$$

# Matrix calculus

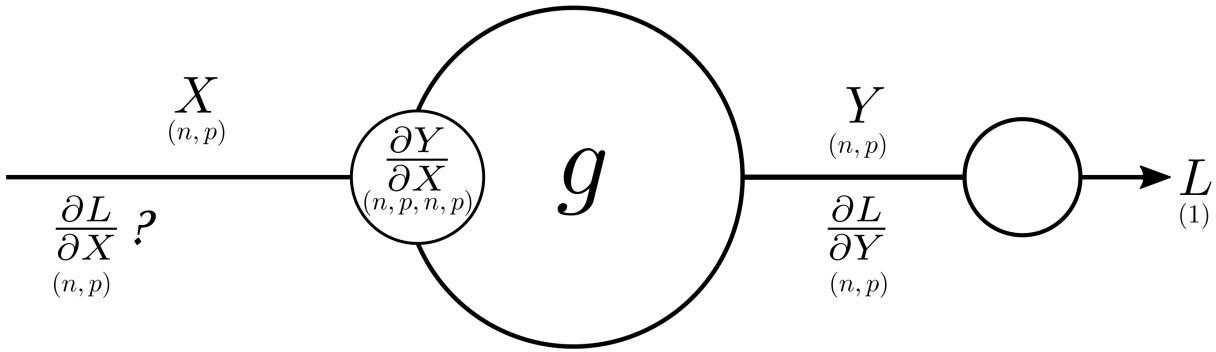
- Function $\mathcal{M}$ is differentiable.

$$\begin{aligned} \mathcal{M} : \mathbb{R}^{(m,n)} &\longrightarrow \mathbb{R}^{(p,q)} \\ \mathbf{X} &\longmapsto \mathcal{M}(M) \end{aligned}$$

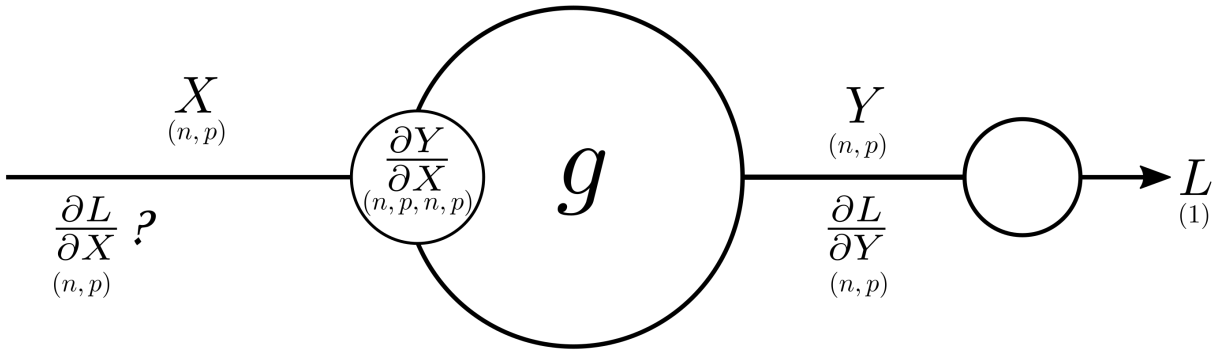
$$\frac{\partial \mathcal{M}}{\partial \mathbf{X}} = \begin{pmatrix} \frac{\partial \mathcal{M}_{1,1}}{\partial \mathbf{X}} & \cdots & \frac{\partial \mathcal{M}_{1,q}}{\partial \mathbf{X}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathcal{M}_{p,q}}{\partial \mathbf{X}} & \cdots & \frac{\partial \mathcal{M}_{p,q}}{\partial \mathbf{X}} \end{pmatrix}$$

This is an array of size $(m, n, p, q)$.

# Backpropagation through an activation function $g$

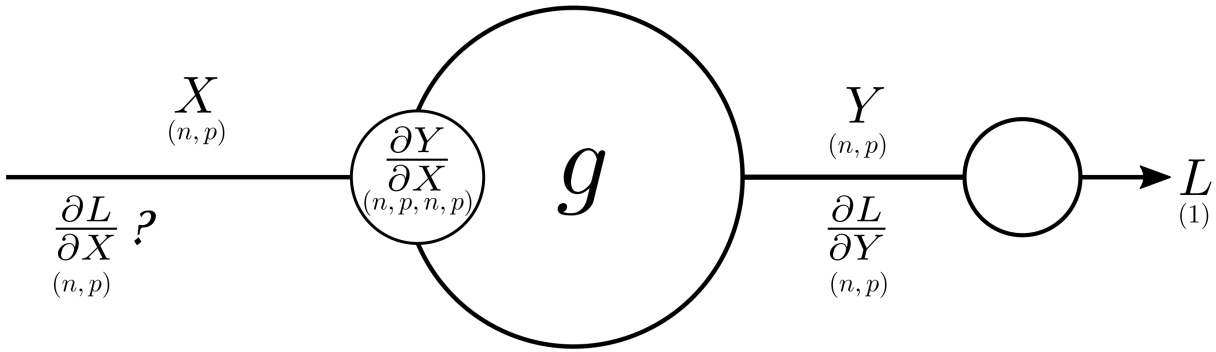


# Backpropagation through an activation function $g$



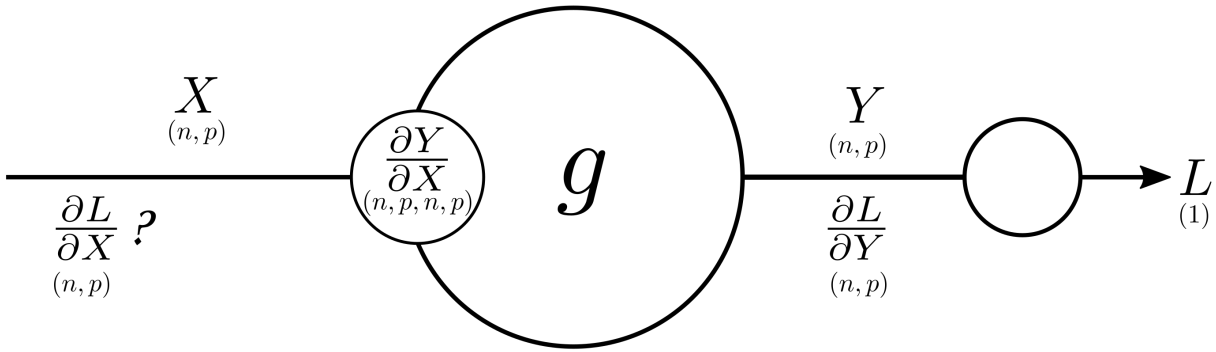

- Computing the full matrix $\frac{\partial \mathbf{Y}}{\partial \mathbf{X}}$ is impractical

# Backpropagation through an activation function $g$



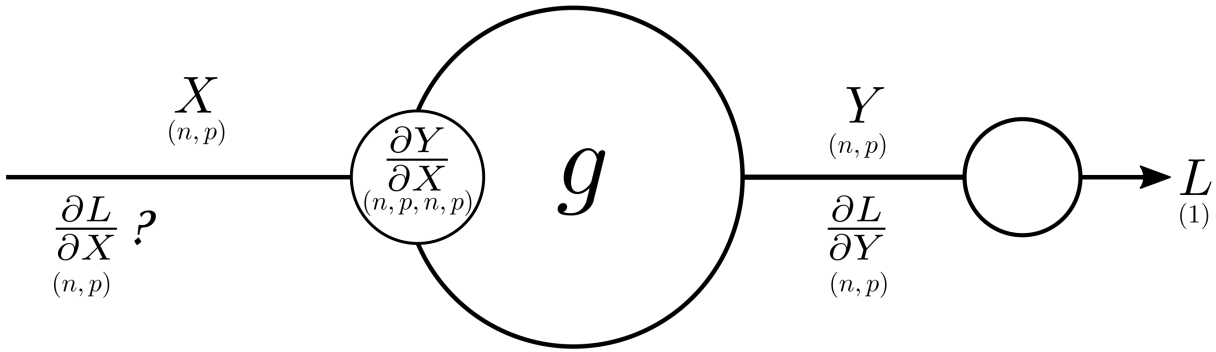

- Computing the full matrix $\frac{\partial \mathbf{Y}}{\partial \mathbf{X}}$ is impractical
- But here $Y_{i,j}$ only depends on $X_{i,j}$: $Y_{i,j} = \mathrm{g}(X_{i,j})$

# Backpropagation through an activation function $g$




- Computing the full matrix $\frac{\partial \mathbf{Y}}{\partial \mathbf{X}}$ is impractical
- But here $Y_{i,j}$ only depends on $X_{i,j}$: $Y_{i,j} = \mathrm{g}(X_{i,j})$
- Therefore: $\frac{\partial \mathbf{Y}_{i,j}}{\partial \mathbf{X}_{i,j}} = \mathrm{g}'$.

# Backpropagation through an activation function $g$

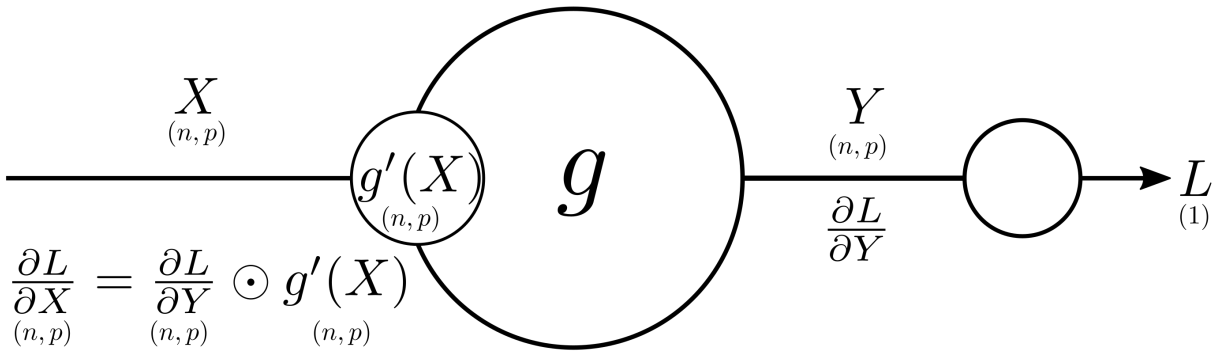



- Computing the full matrix $\frac{\partial \mathbf{Y}}{\partial \mathbf{X}}$ is impractical
- But here $Y_{i,j}$ only depends on $X_{i,j}$: $Y_{i,j} = \mathrm{g}(X_{i,j})$
- Therefore: $\frac{\partial \mathbf{Y}_{i,j}}{\partial \mathbf{X}_{i,j}} = \mathrm{g}'$.
- Finally:

$$\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{Y}} \odot \mathrm{g}'(\mathbf{X}) \ ,$$

where $\odot$ is the term by term matrix multiplication or Hadamard matrix multiplication.

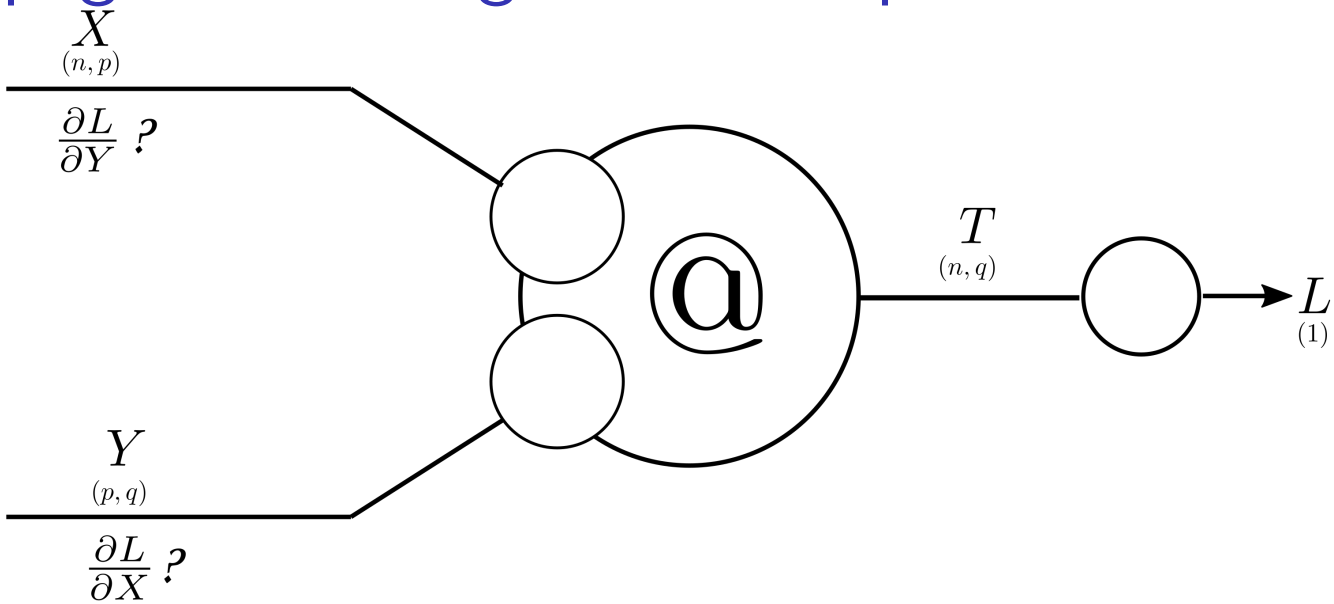
# Backpropagation through an activation function $g$



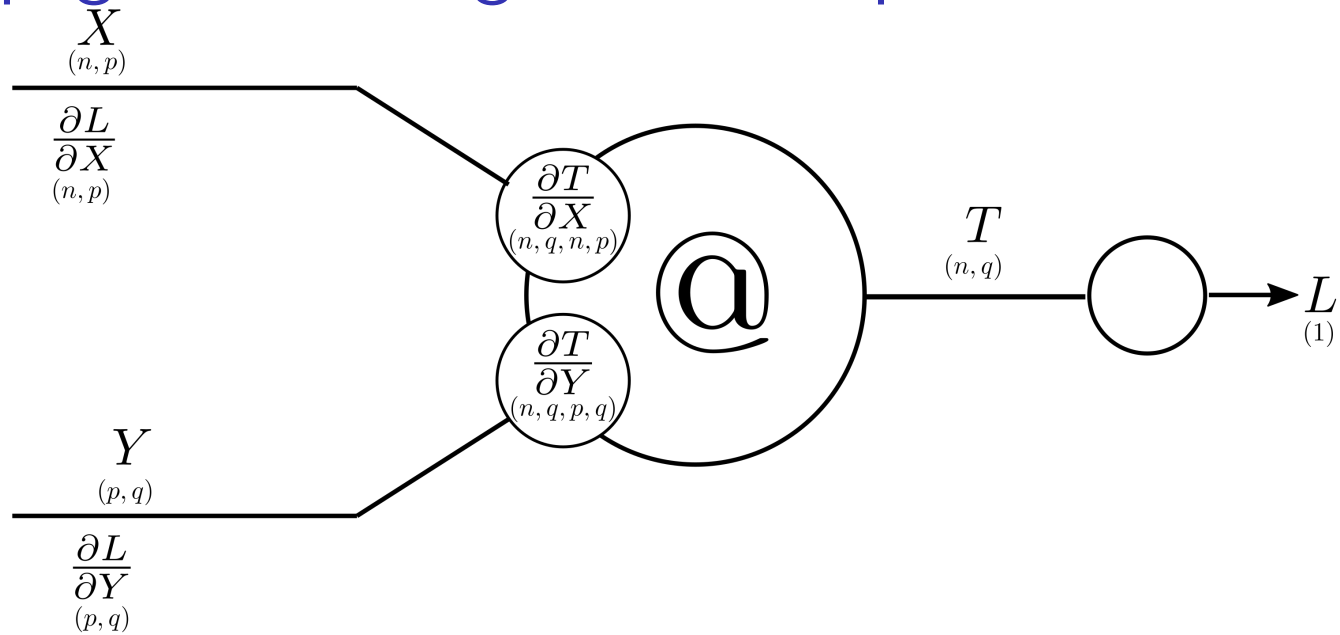

We will abusively write:

$$\frac{\partial \mathbf{Y}}{\partial \mathbf{X}} = \mathrm{g}'(\mathbf{X})$$

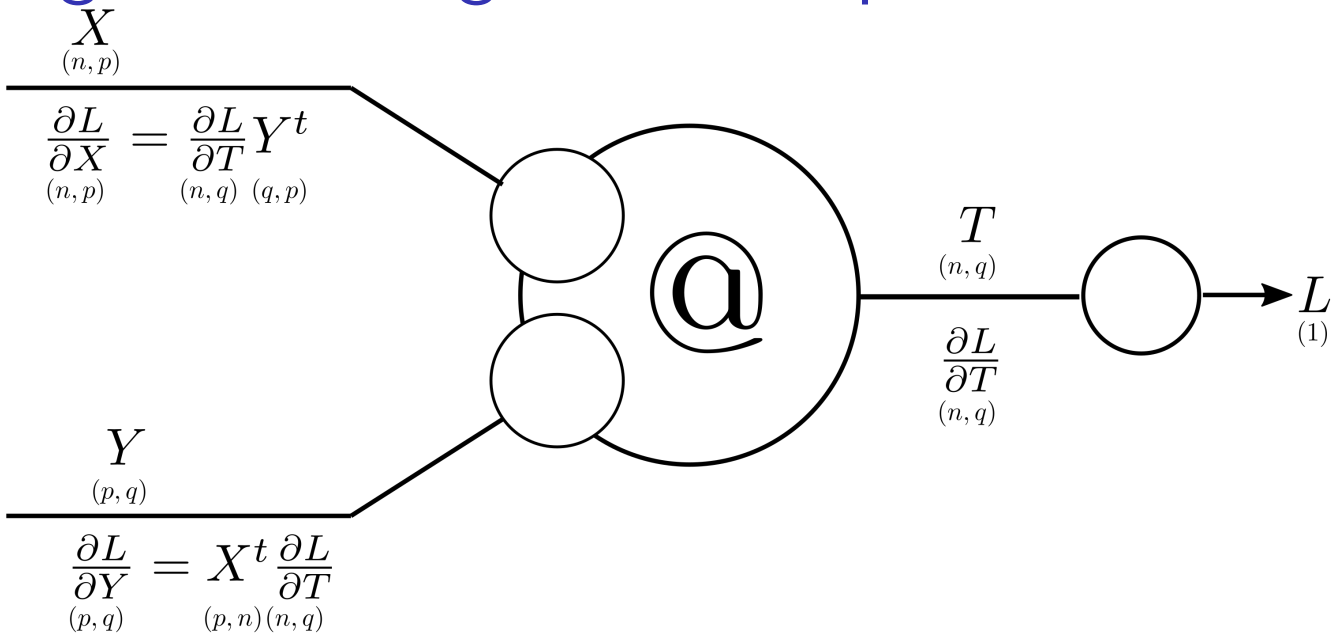
# Backpropagation through a matrix product



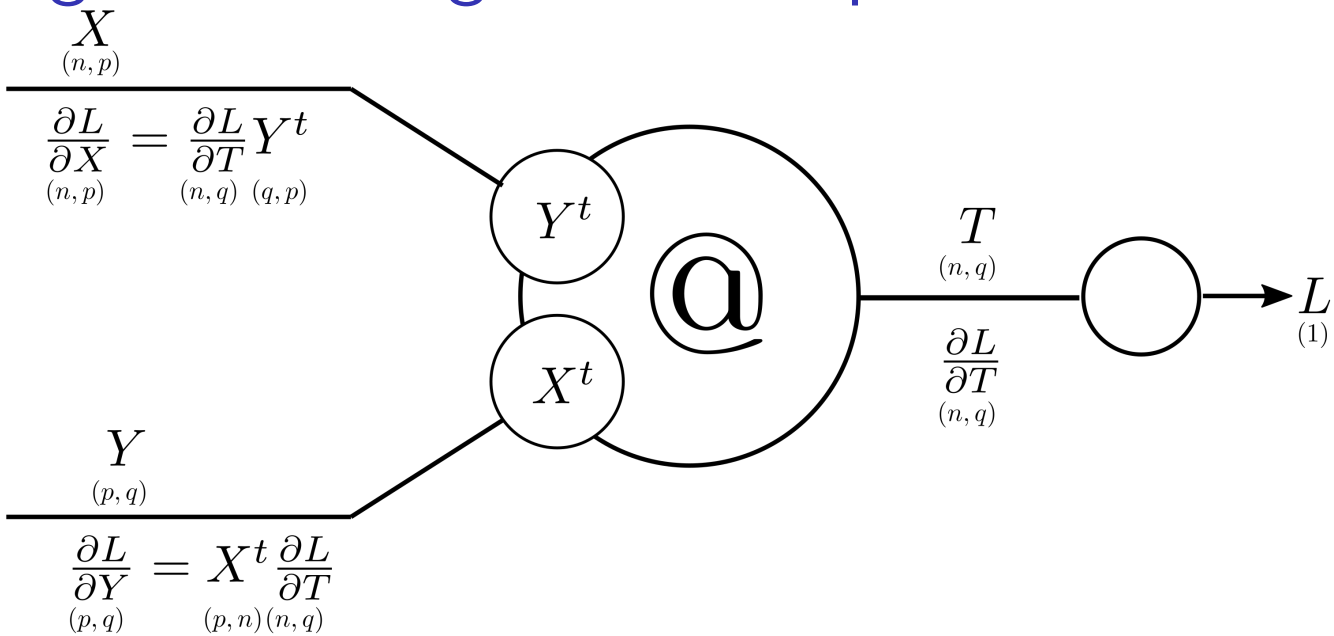
# Backpropagation through a matrix product



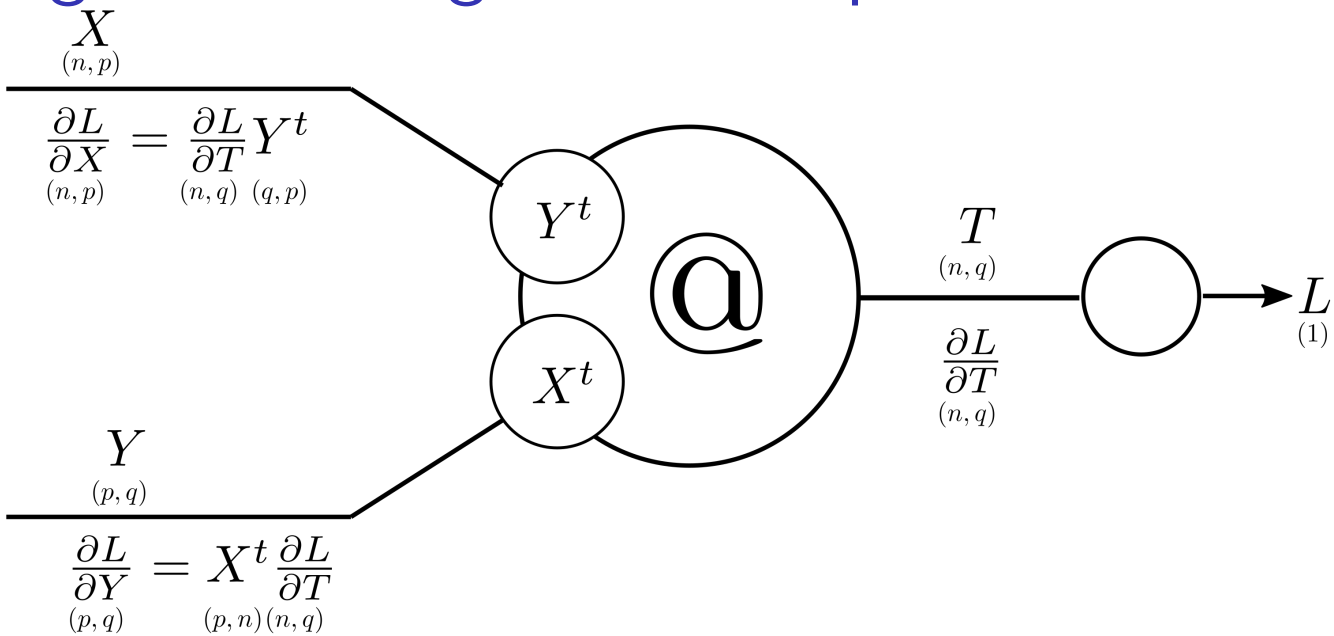
# Backpropagation through a matrix product



# Backpropagation through a matrix product

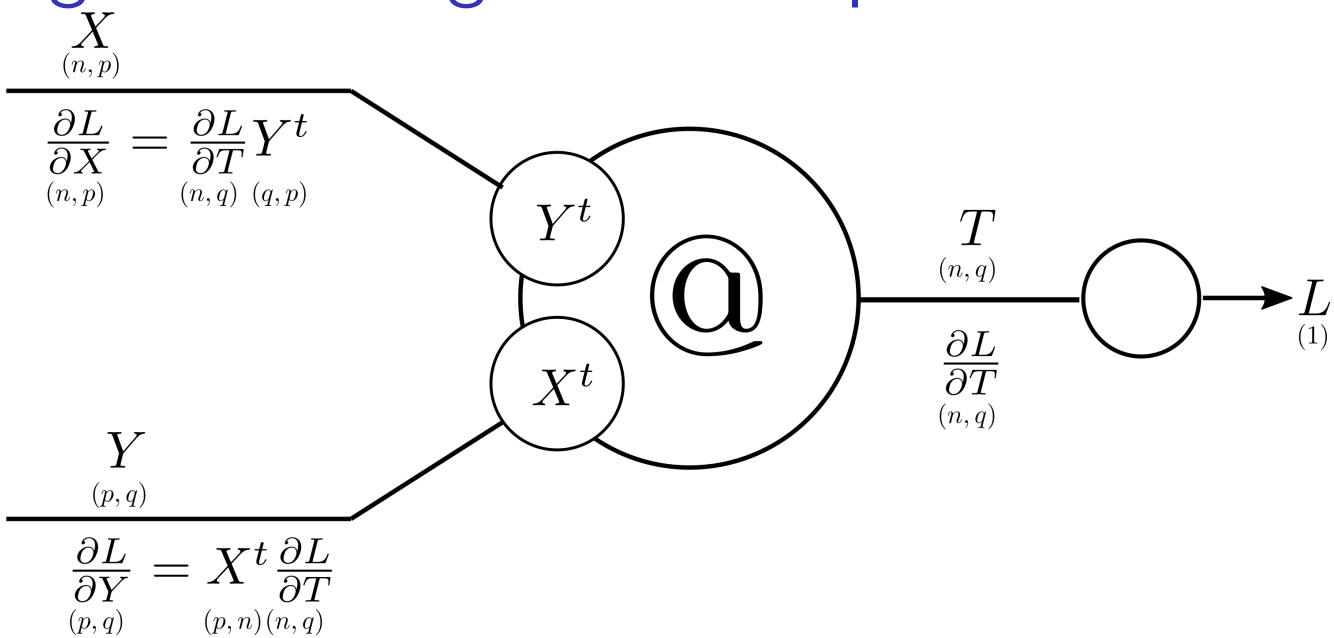


# Backpropagation through a matrix product




In fact this is the only way you can make the shapes match.

# Backpropagation through a matrix product

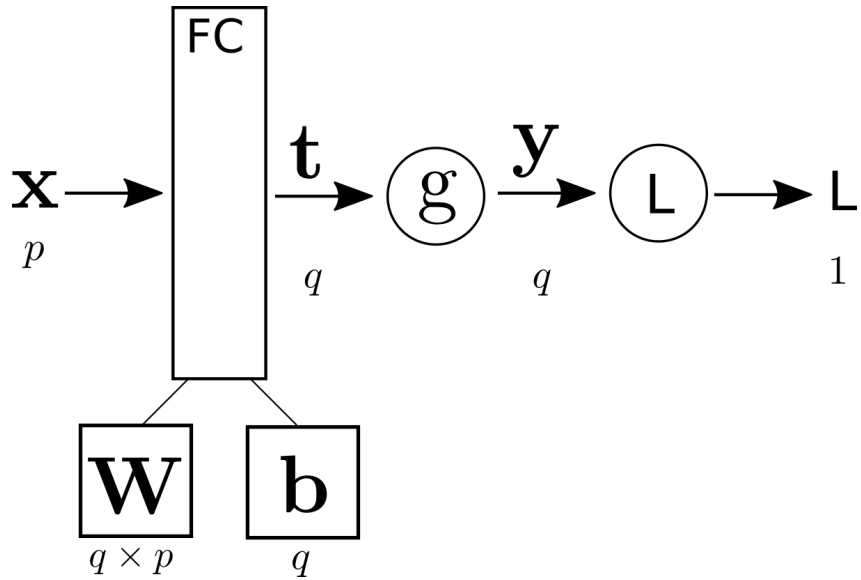



In fact this is the only way you can make the shapes match.

We will abusively write, only for matrix multiplication:

$$\frac{\partial \mathbf{T}}{\partial \mathbf{X}} = \mathbf{Y}^t$$
$$\frac{\partial \mathbf{T}}{\partial \mathbf{Y}} = \mathbf{X}^t$$

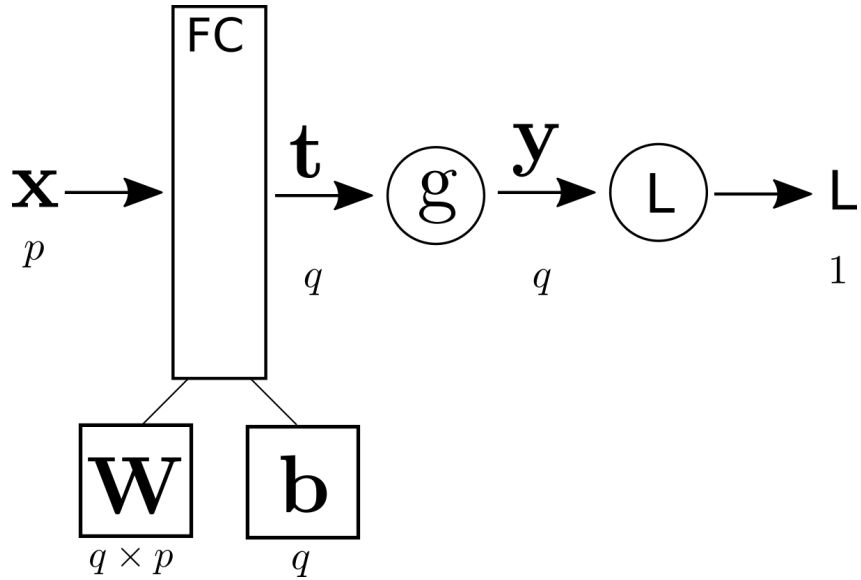
# Backpropagation through a fully connected layer




Setup:

$$p, q \in \mathbb{N}^*$$
$$\mathbf{x} \in \mathbb{R}^p$$
$$\mathbf{W} \in \mathbb{R}^q \times \mathbb{R}^p$$
$$\mathbf{b}, \mathbf{t}, \mathbf{y} \in \mathbb{R}^q$$
$$L \in \mathbb{R}$$

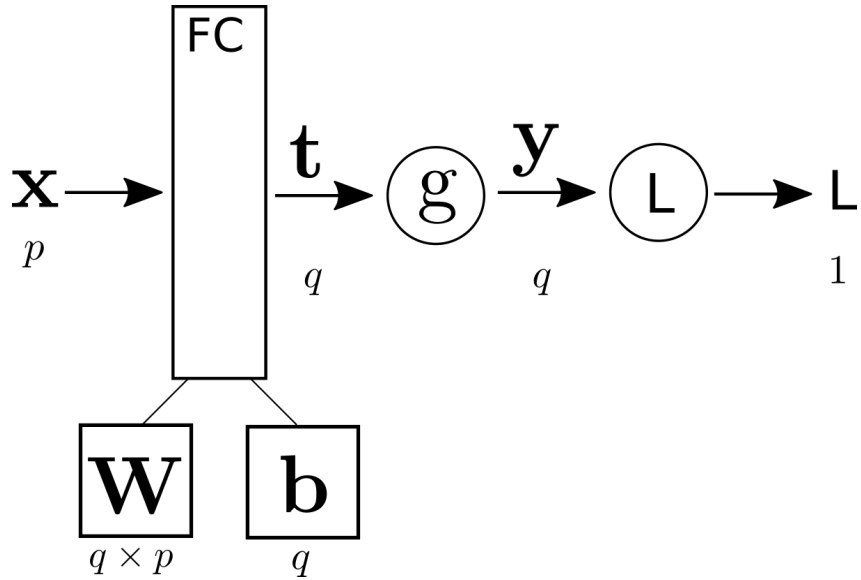
# Backpropagation through a fully connected layer




Forward pass:

$$\begin{aligned} \mathbf{t} &= \mathbf{W}\mathbf{x} + \mathbf{b} \\ \mathbf{y} &= \mathbf{g}(\mathbf{W}\mathbf{x} + \mathbf{b}) \\ L &= L(\mathbf{y}) \end{aligned}$$

# Backpropagation through a fully connected layer



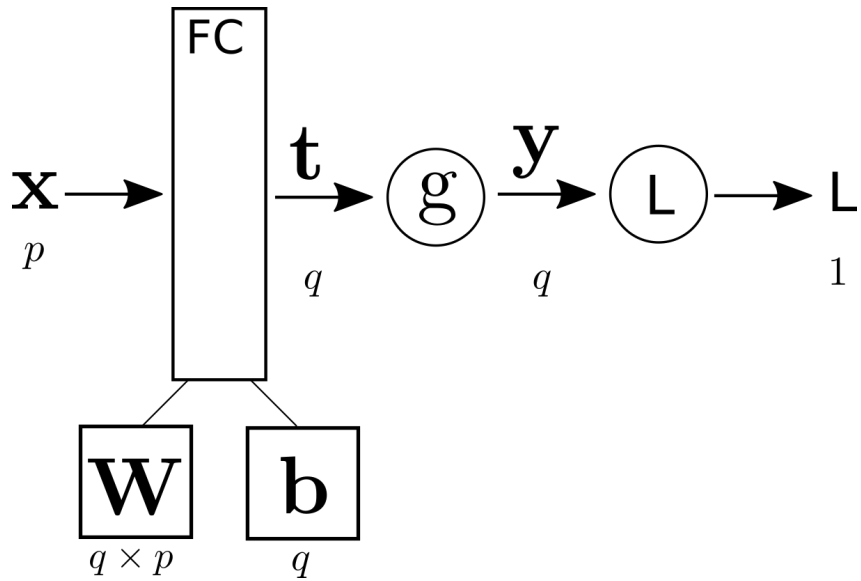

Forward pass:

$$\begin{aligned} \mathbf{t} &= \mathbf{W}\mathbf{x} + \mathbf{b} \\ \mathbf{y} &= \mathbf{g}(\mathbf{W}\mathbf{x} + \mathbf{b}) \\ L &= L(\mathbf{y}) \end{aligned}$$

Local gradients:

$$\begin{aligned} \frac{\partial \mathbf{t}}{\partial \mathbf{W}} &= \mathbf{x}^t \\ \frac{\partial \mathbf{t}}{\partial \mathbf{b}} &= Id_{(q)} \\ \frac{\partial \mathbf{y}}{\partial \mathbf{t}} &= \mathbf{g}'(\mathbf{t}) \end{aligned}$$

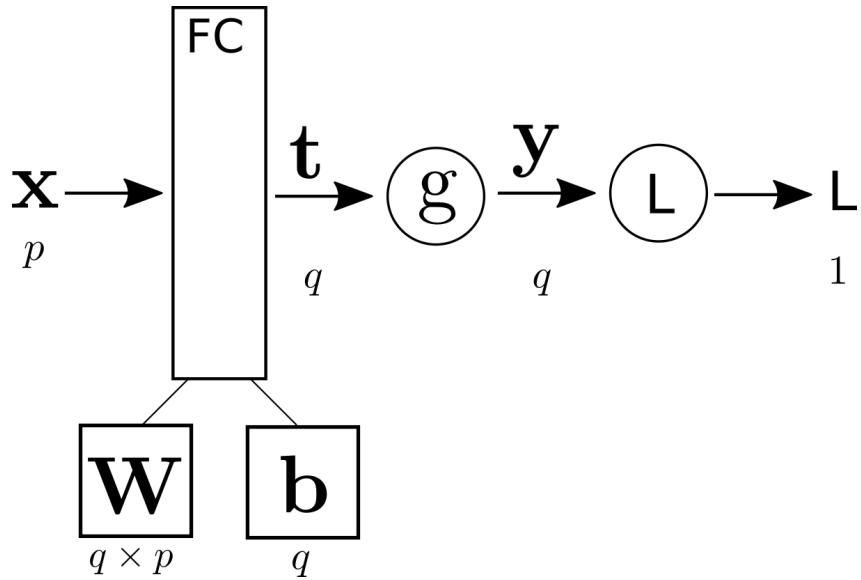
# Backpropagation through a fully connected layer




Backpropagation:

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{t}} &= \frac{\partial L}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{t}} \\ &= \frac{\partial L}{\partial \mathbf{y}} \odot \mathbf{g}'(\mathbf{t})\end{aligned}$$

# Backpropagation through a fully connected layer




Backpropagation:

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{t}} . \frac{\partial \mathbf{t}}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{y}} \odot \mathbf{g}'(\mathbf{t}) . \mathbf{x}^t$$

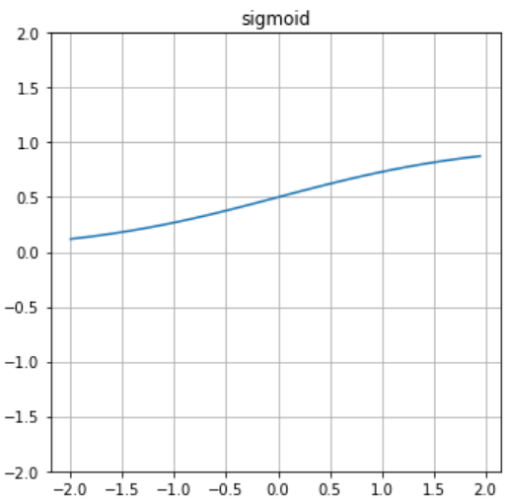
$$\frac{\partial L}{\partial \mathbf{b}} = Id^t . \frac{\partial L}{\partial \mathbf{t}} = \frac{\partial L}{\partial \mathbf{y}} \odot \mathbf{g}'(\mathbf{t})$$

# Contents

# Network parameters initialization

## General idea

Inputs of activation functions should be in a range such that gradients are high.




- Bias are set to zero
- If weights are also initialized to zero, then in each layer the activations will remain equal – symmetry will never be broken
- Empirical solutions are based on a gaussian distribution of the weights, with *small* standard deviation.

# Network parameters initialization: current practice

- [Glorot and Bengio, 2010]: they empirically show that a standard deviation of $1/\sqrt{n}$ gives good results (where $n$ is the number of inputs of a neuron)
- [He et al., 2015]: in the case of ReLU activations, they recommend a $2/\sqrt{n}$ standard deviation

# Contents

# Conclusion

We have seen:

- What is an artificial neuron and an artificial neural network (NN)
- The (potential) power of a NN
- The backpropagation algorithm
- NN learning basics

Next step:

- Application to images

# Procedure for setting up the practical sessions environment

1. Download **practical_sessions.zip** from `https://tinyurl.com/y558bmun`. Uncompress it.
2. Create (if necessary) a Google account. Log in.
3. Go to: `https://colab.research.google.com`. A notebook will open if it is your first connection. Click on Save on drive.
4. Go to: `https://drive.google.com`. A Colab Notebooks directory should be available. Go into it and upload **practical_sessions** directory.

Following this procedure you should have a `practical_sessions` directory on your Colab Notebooks containing some folders with data sets and python files. A first set of notebooks, for your first assignment, will also be available.
The following days, the new notebooks will be available from:

`https://tinyurl.com/y3x6andh`

# References I


[Cybenko, 1989] Cybenko, G. (1989). Approximations by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:183–192.

[Glorot and Bengio, 2010] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256.

[He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *arXiv:1502.01852 [cs]*. arXiv: 1502.01852.

[Hornik, 1991] Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257.

[LeCun, 1985] LeCun, Y. (1985). Une procedure d'apprentissage pour reseau a seuil asymmetrique (A learning scheme for asymmetric threshold networks). In *proceedings of Cognitiva 85*.

[McCulloch and Pitts, 1943] McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

# References II

[Werbos, 1982] Werbos, P. J. (1982). Applications of advances in nonlinear sensitivity analysis. In Drenick, R. F. and Kozin, F., editors, *System Modeling and Optimization*, Lecture Notes in Control and Information Sciences, pages 762–770. Springer Berlin Heidelberg.