

Neural Networks: History and foundation

Léopold Maytié

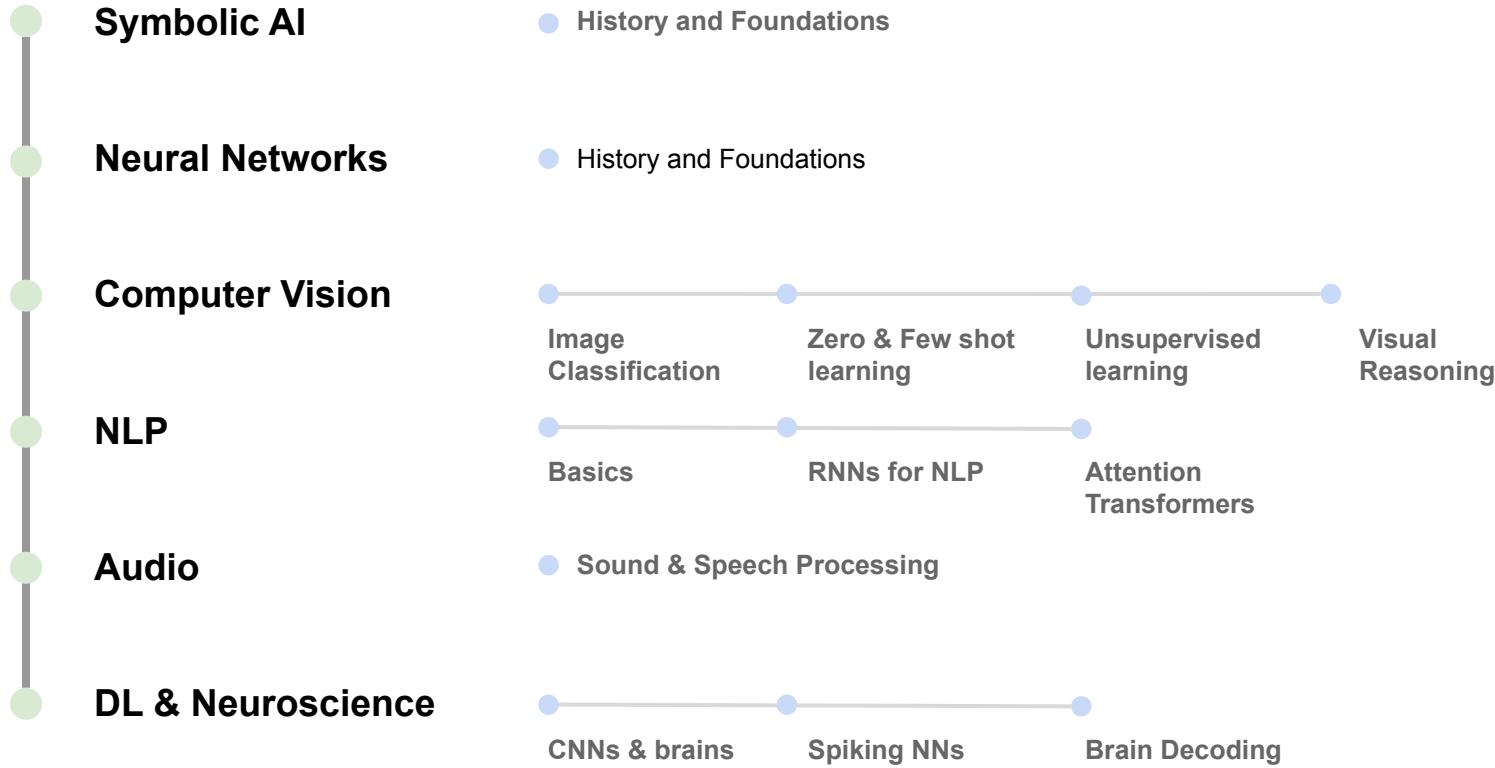
Artificial and Natural Intelligence Toulouse Institute (ANITI)
September 11, 2023



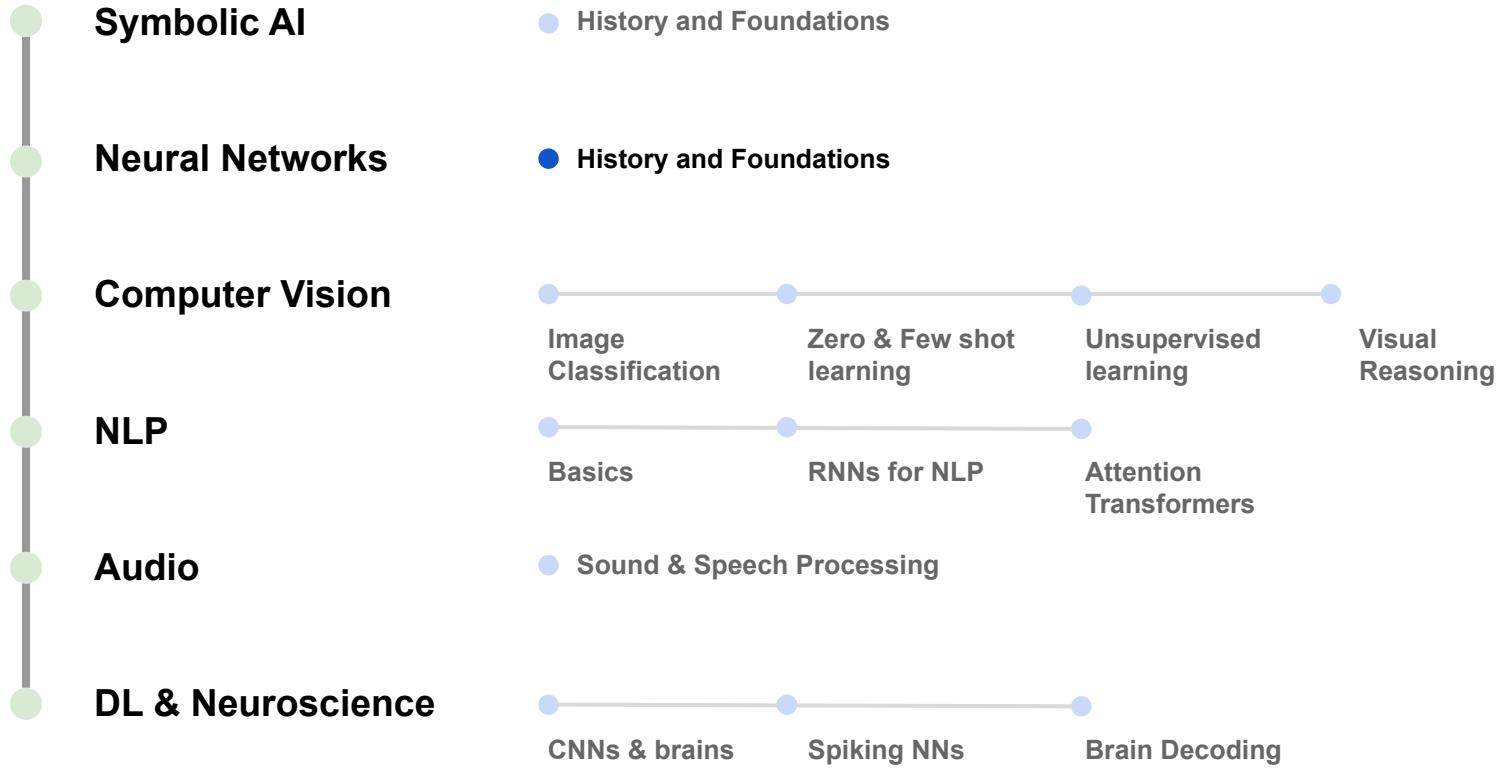
Course Intro

- **Symbolic AI**
- **Neural Networks**
- **Computer Vision**
- **NLP**
- **Audio**
- **DL & Neuroscience**

Course Intro



Course Intro



- Introduction
- Part 1: Theory of Deep Learning
 - I. The Formal Neuron
 - II. Perceptron: A learning algorithm for the neuron model
 - III. Multilayer perceptron
 - IV. Neural Network learning as optimization
 - A. Cost Function
 - B. Gradient Descent
 - C. Backpropagation algorithm
- Part 2 : Implementation of a Neural Network
 - I. How to choose the architecture
 - II. How to choose the loss
 - III. Training and evaluation
 - IV. Optimizer
 - V. Regularization
 - VI. Hyperparameters

Introduction

Artificial Intelligence

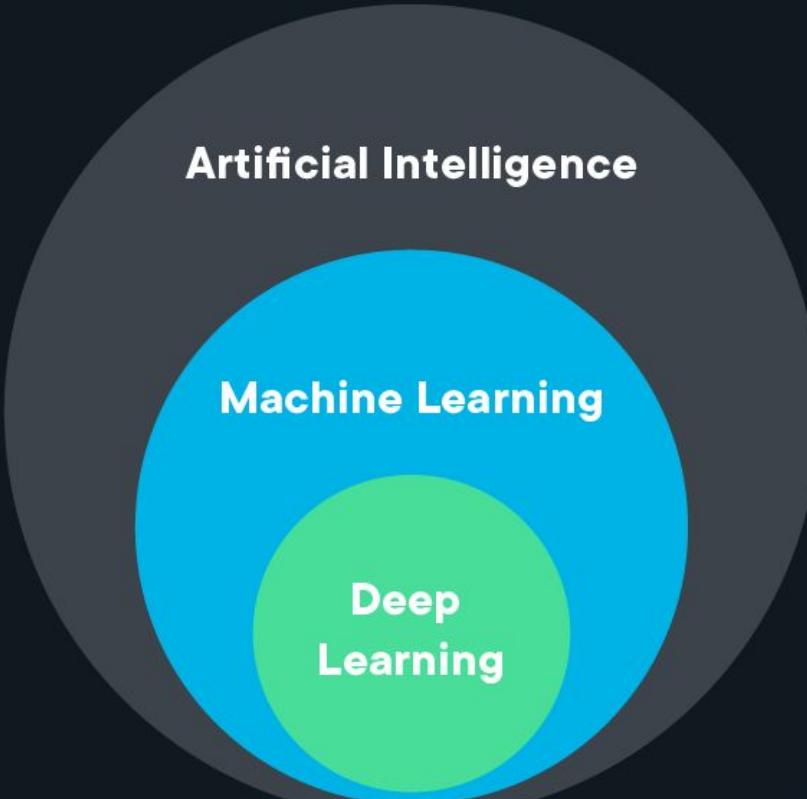
A science devoted to making machines think and act like humans.

Machine Learning

Focuses on enabling computers to perform tasks without explicit programming.

Deep Learning

A subset of machine learning based on artificial neural networks.



Artificial Intelligence

Machine Learning

Deep
Learning

Introduction

Artificial Intelligence

Symbolic Artificial Intelligence / "Good Old-Fashioned AI"

Expert Systems

Machine Learning

Neural Networks

Deep Learning

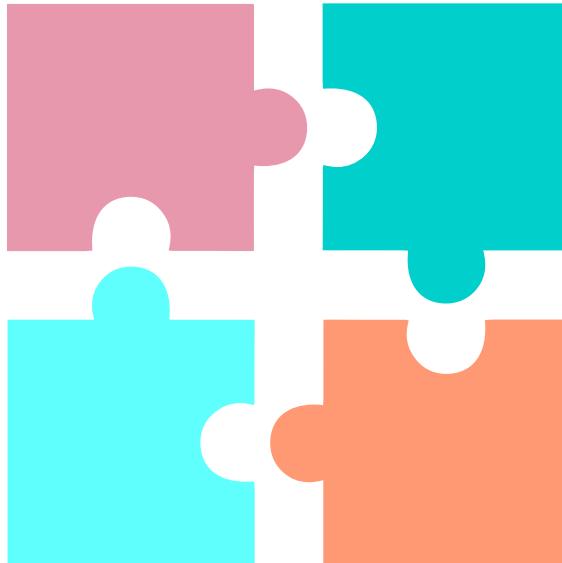
Supervised
Learning

Reinforcement
Learning

Unsupervised
Learning

Introduction

THE PARTS



01

Linear algebra / calculus

Matrix operations, derivatives ...

02

Statistics / Probability

Accuracy measurement tools, central tendency and variability.

03

Optimization

Learning algorithms (optimizers).

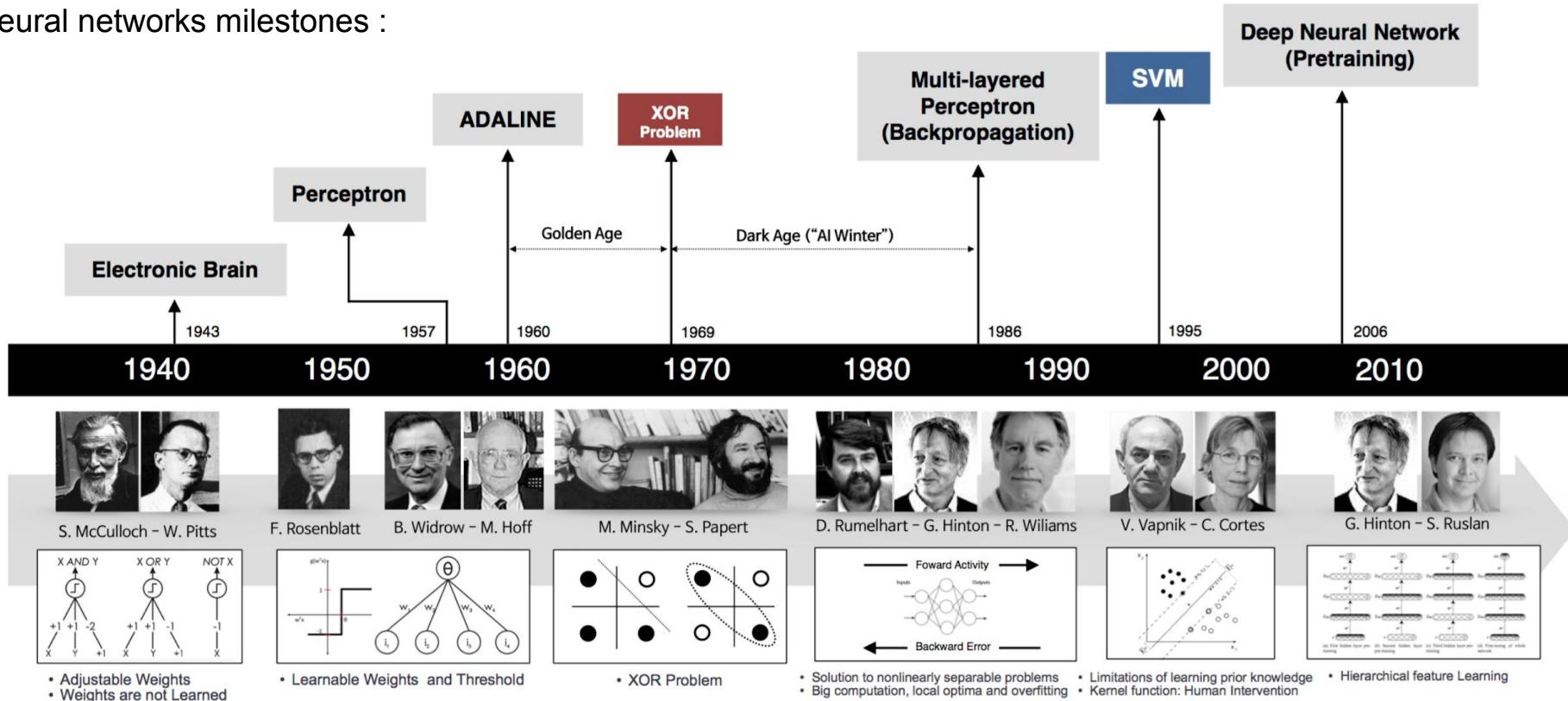
04

Coding skills

DL softwares (PyTorch, TensorFlow ...). Data engineering.

Introduction

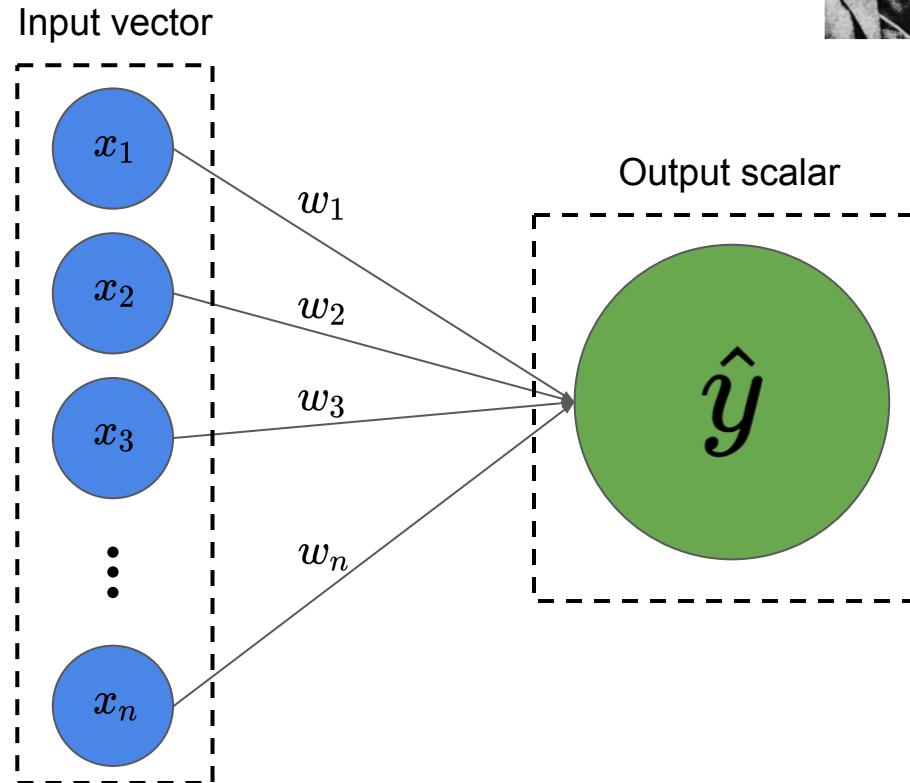
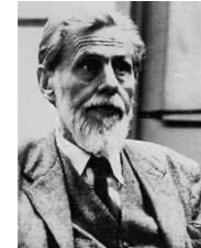
Neural networks milestones :



Part 1 : Theory of Deep Learning

I- The Formal Neuron

Artificial neuron: McCulloch & Pitt's neuron model (1943)



Recall (Linear Algebra / Calculus)

Dot product

Two vectors :

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

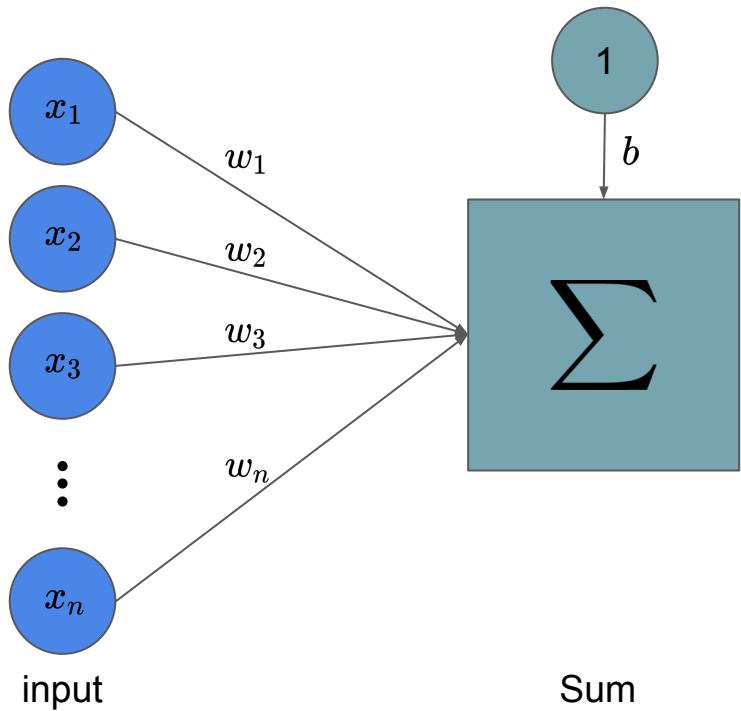
Transpose operator :

$$\mathbf{x}^T = [x_1 \quad x_2 \quad x_3]$$

$$\mathbf{x}^T \mathbf{y} = [x_1 \quad x_2 \quad x_3] \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = x_1 \times y_1 + x_2 \times y_2 + x_3 \times y_3 = \vec{\mathbf{x}} \cdot \vec{\mathbf{y}}$$

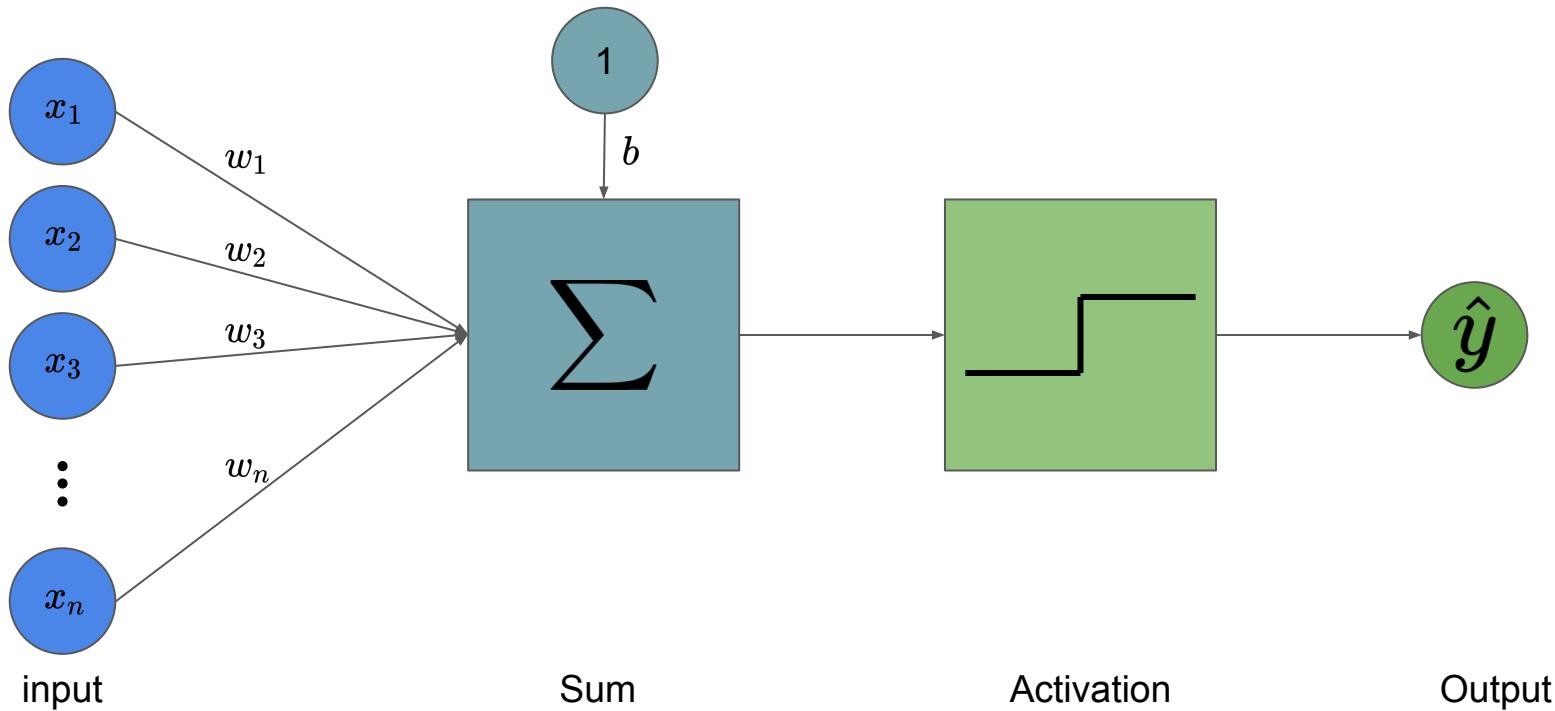
I- The Formal Neuron

- 1st step : Linear transformation : $s = \mathbf{w}^T \mathbf{x} + b$



I- The Formal Neuron

- 1st step : Linear transformation : $s = \mathbf{w}^T \mathbf{x} + b$
- 2nd step : Non-linear activation function : $\hat{y} = f(s)$



I- The Formal Neuron

Similarities with biological neurons:

- Choose f as Heaviside function : $H(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$
- $\hat{y} = 1 \Leftrightarrow \mathbf{w}^T \mathbf{x} \geq -b$ activated
- $\hat{y} = 0 \Leftrightarrow \mathbf{w}^T \mathbf{x} < -b$ unactivated
- Biological neurons : produce output if input weighted by synaptic weight exceeds the threshold

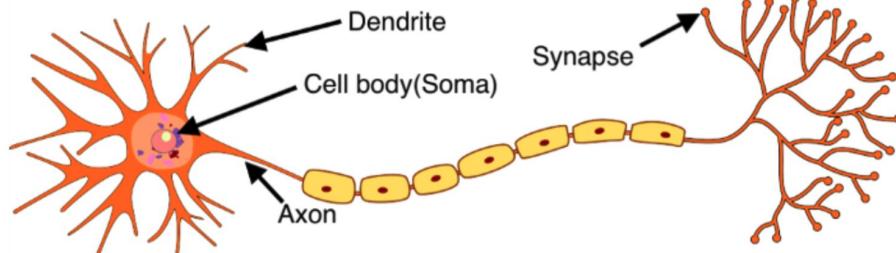
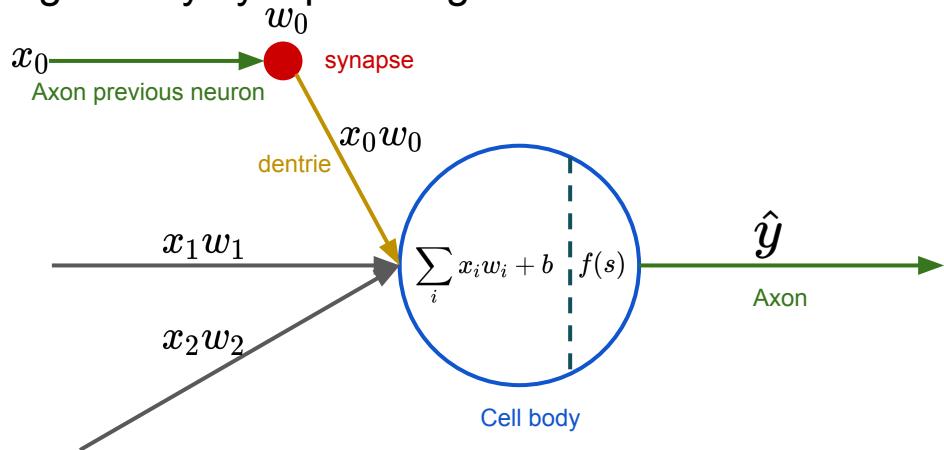
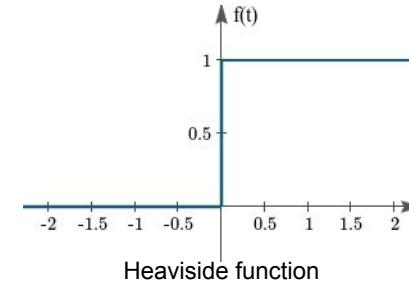


Image source: Wikimedia Commons



I- The Formal Neuron

- An artificial neuron could solve **linear** logical problems: AND, OR, NOT

AND

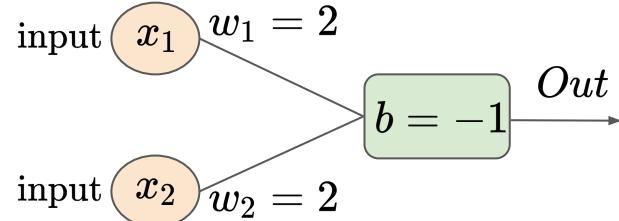
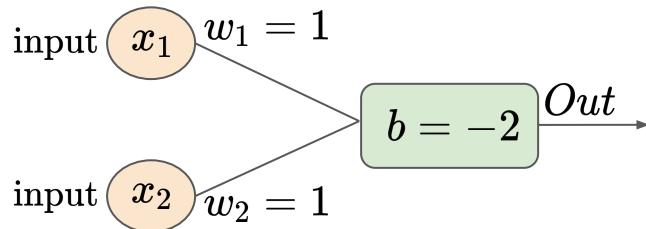
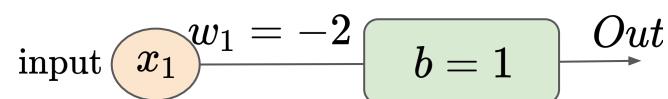
x_1	x_2	Out
0	0	0
0	1	0
1	0	0
1	1	1

OR

x_1	x_2	Out
0	0	0
0	1	1
1	0	1
1	1	1

NOT

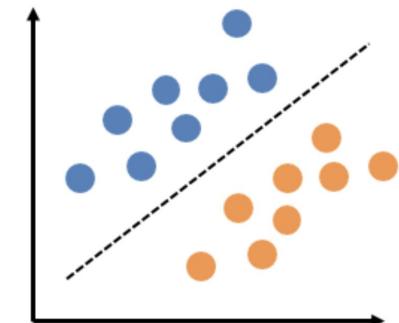
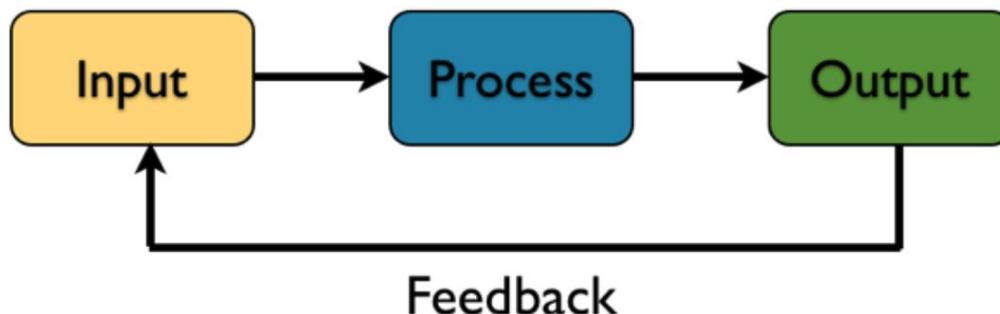
x_1	Out
0	1
1	0



II- Perceptron: A learning algorithm for the neuron model

Rosenblatt, F. (1957). *The perceptron, a perceiving and recognizing automaton Project Para.*

- Automatic learning of weights
- Supervised learning of binary classifiers
- Could recognize letters and numbers



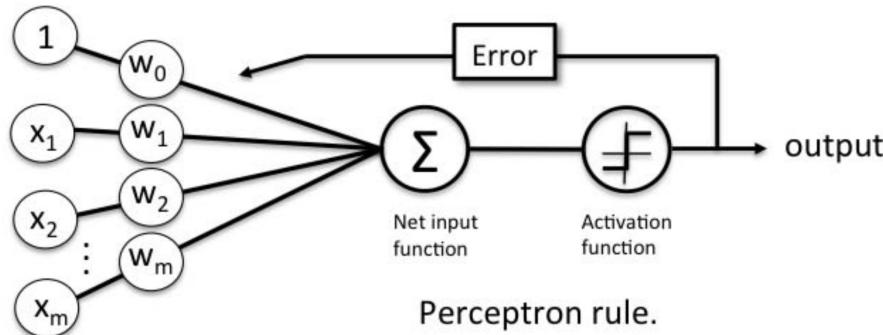
II- Perceptron: A learning algorithm for the neuron model

Let $D = (\langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle, \dots, \langle x_n, y_n \rangle) \in (\mathbb{R}^m \times \{0, 1\})^n$

1. Initialise w_i with random small values
2. For every training epoch:

For every sample $\langle x_i, y_i \rangle \in D$:

- (a) $\hat{y} := \sigma(x_i \times w)$ ← Compute output (prediction)
- (b) $err := (y_i - \hat{y}_i)$ ← Compute error
- (c) $w := w + err \times x_i$ ← Update parameters



II- Perceptron: A learning algorithm for the neuron model

Example: learning OR logical gate

Start : $(w_0, w_1, w_2) = (0, 2, 2)$

- First pass:

Input : $(1, x_1, x_2) = (1, 0, 0)$

a/ $y_{\text{hat}} = H(0x_1 + 2x_0 + 2x_0) = H(0) = 1$

b/ $\text{err} = (0 - 1) = -1$

c/ $(w_0, w_1, w_2) = (0 - 1x_1, 2 - 1x_0, 1 - 2x_0) = (-1, 2, 2)$ Good weights !

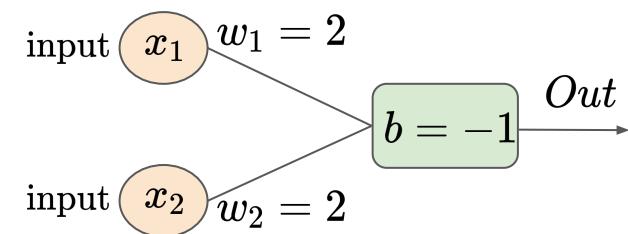
- Second pass:

Input : $(1, x_1, x_2) = (1, 1, 1)$

a/ $y_{\text{hat}} = H(-1x_1 + 2x_1 + 2x_1) = H(3) = 1$

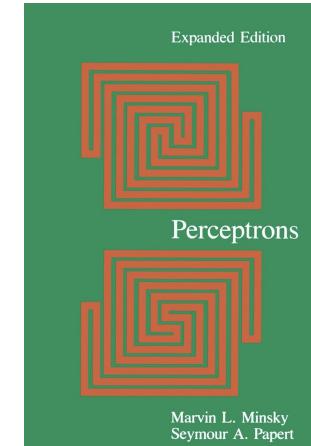
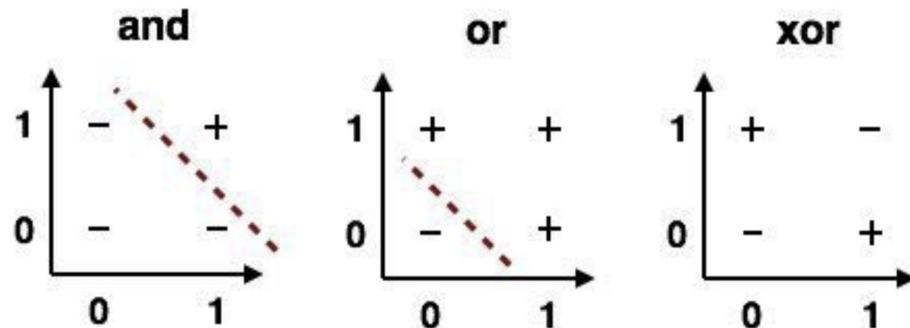
b/ $\text{err} = (1 - 1) = -0$

c/ $(w_0, w_1, w_2) = (-1 - 0x_1, 2 - 0x_1, 2 - 0x_1) = (-1, 2, 2)$ No change



The first AI winter

Minsky and Papert (1969) show that the perceptron can't even solve the XOR problem

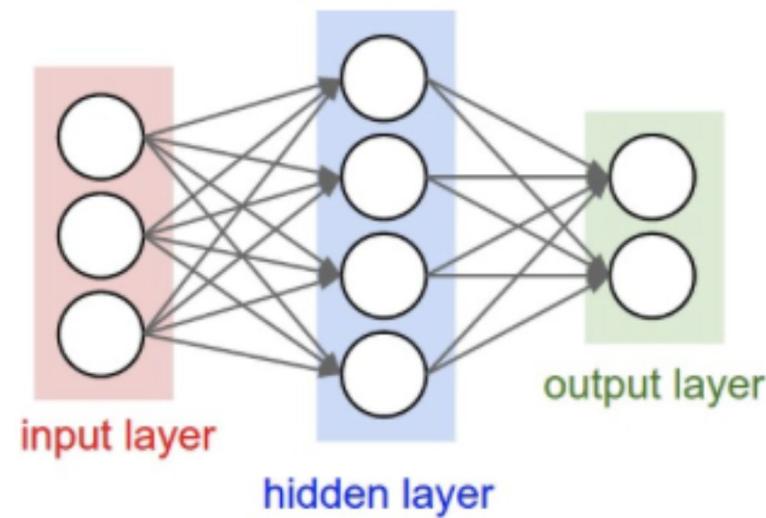
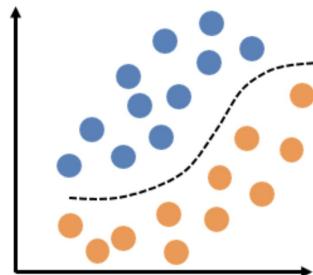


⇒ Kills research on neural nets for the next 15-20 years

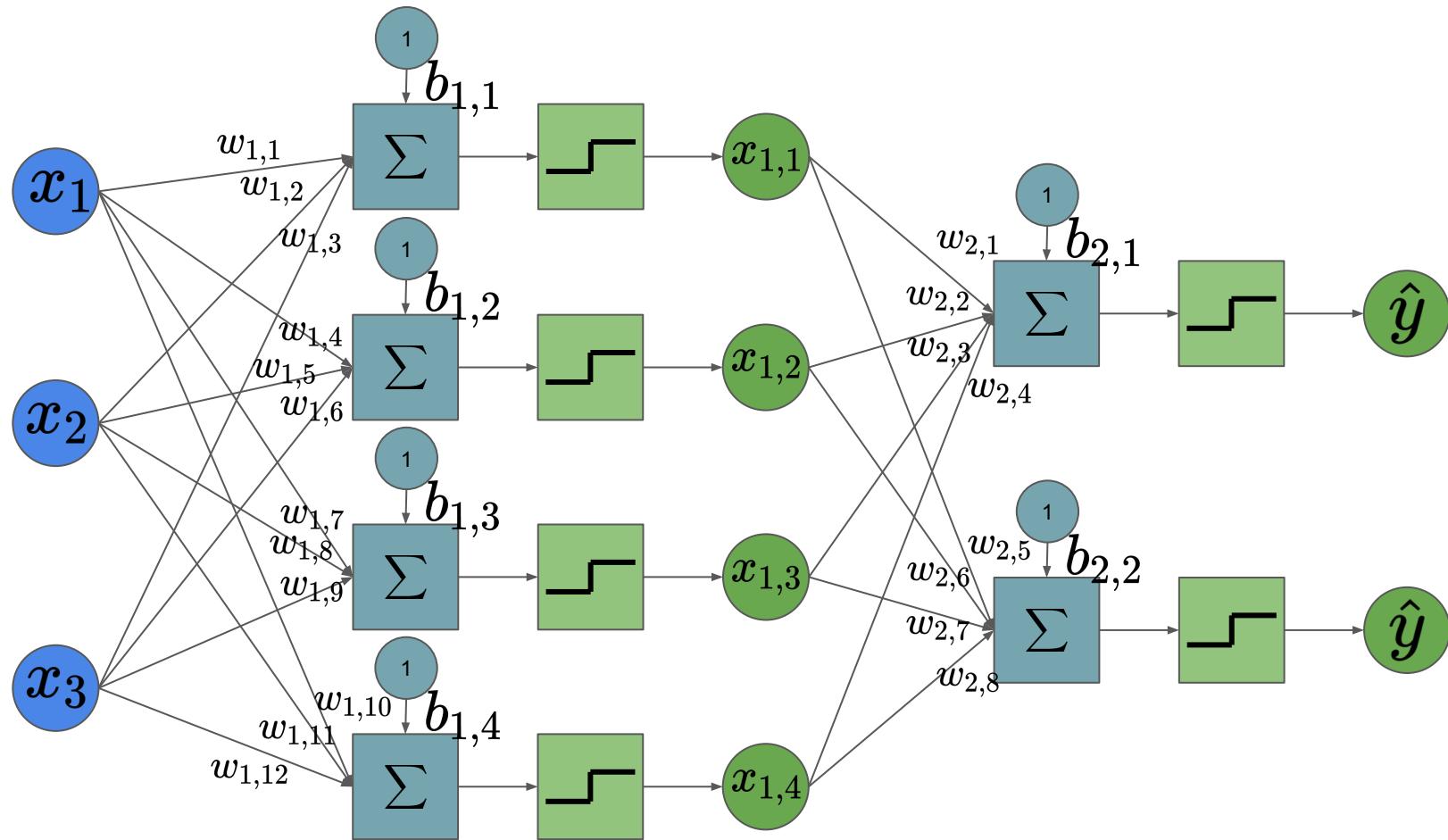
III- Multilayer perceptrons (1980's)

Solution to the XOR problem: **Multilayer perceptrons**

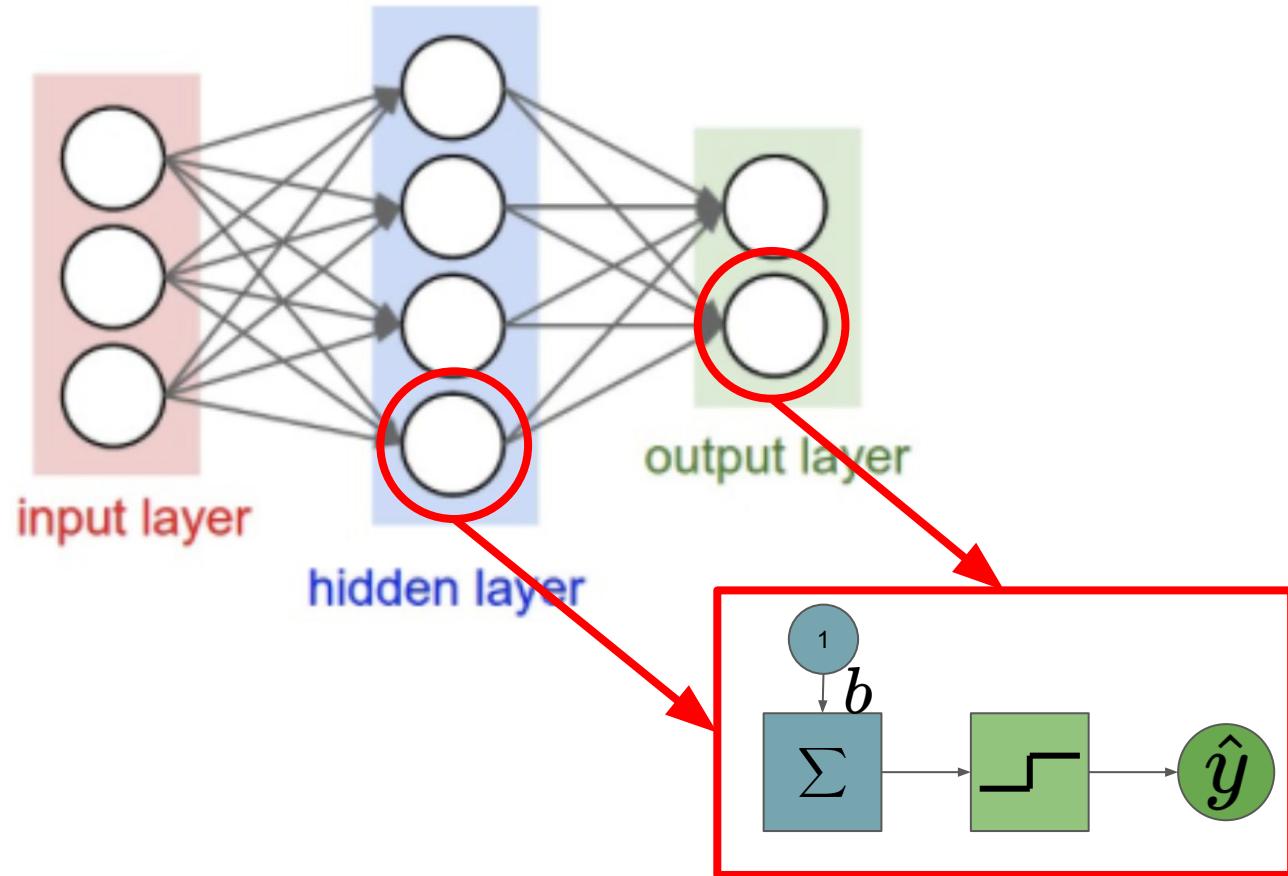
- Composed of: **input layer**, **hidden layer(s)** and an **output layer**.
- Each node (of hidden and output layers) is a neuron that uses a **nonlinear** activation function.
- It can distinguish data that is **not linearly separable**.



III- Multilayer perceptrons (1980's)



III- Multilayer perceptrons (1980's)



Recall (Linear Algebra / Calculus)

Matrix product :

$$\begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} = \begin{bmatrix} a_1a_2 + b_1c_2 & a_1b_2 + b_1d_2 \\ c_1a_2 + d_1c_2 & c_1b_2 + d_1d_2 \end{bmatrix}$$

$$\begin{array}{c|c} \text{1 neuron} & \text{4 neurons} \\ \hline \boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_{41} \end{bmatrix} & \boldsymbol{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{1,4} & w_{1,5} & w_{1,6} \\ w_{1,7} & w_{1,8} & w_{1,9} \\ w_{1,10} & w_{1,11} & w_{1,12} \end{bmatrix} \end{array}$$

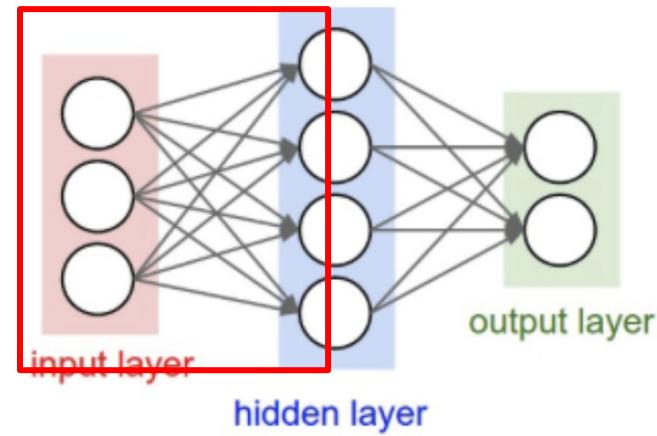
III- Multilayer perceptrons (1980's)

Mathematics behind a Multilayer perceptron :

1- Multiply the input by the weights

$$\mathbf{s} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

$$\mathbf{s} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{1,4} & w_{1,5} & w_{1,6} \\ w_{1,7} & w_{1,8} & w_{1,9} \\ w_{1,10} & w_{1,11} & w_{1,12} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} w_{1,1}x_1 + w_{1,2}x_2 + w_{1,3}x_3 + b_1 \\ w_{1,4}x_1 + w_{1,5}x_2 + w_{1,6}x_3 + b_2 \\ w_{1,7}x_1 + w_{1,8}x_2 + w_{1,9}x_3 + b_3 \\ w_{1,10}x_1 + w_{1,11}x_2 + w_{1,12}x_3 + b_4 \end{bmatrix}$$



III- Multilayer perceptrons (1980's)

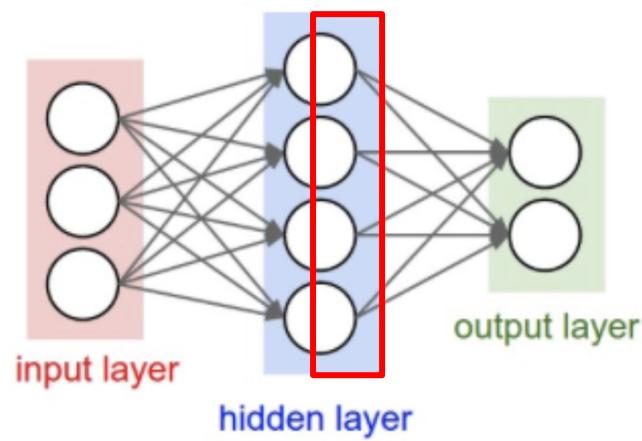
Mathematics behind a Multilayer perceptron :

1- Multiply the input by the weights

$$\mathbf{s} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

2- Apply non-linearity function

$$\begin{bmatrix} x_{11} \\ x_{12} \\ x_{13} \\ x_{14} \end{bmatrix} = f(\mathbf{s})$$



III- Multilayer perceptrons (1980's)

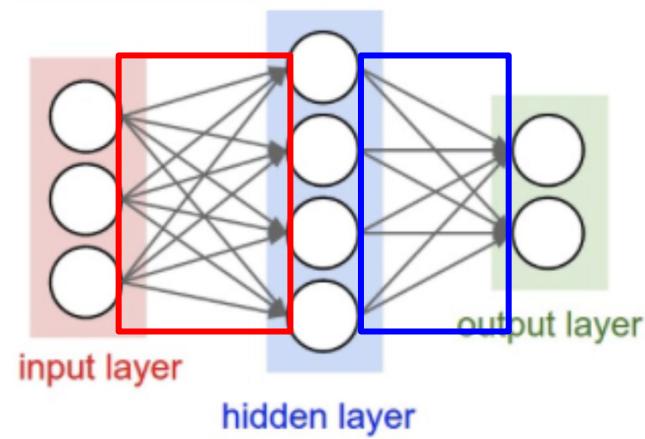
Mathematics behind a Multilayer perceptron :

1- Multiply the input by the weights $s = \mathbf{W}x + b$

2- Apply non-linearity function $\begin{bmatrix} x_{11} \\ x_{12} \\ x_{13} \\ x_{14} \end{bmatrix} = f(s)$

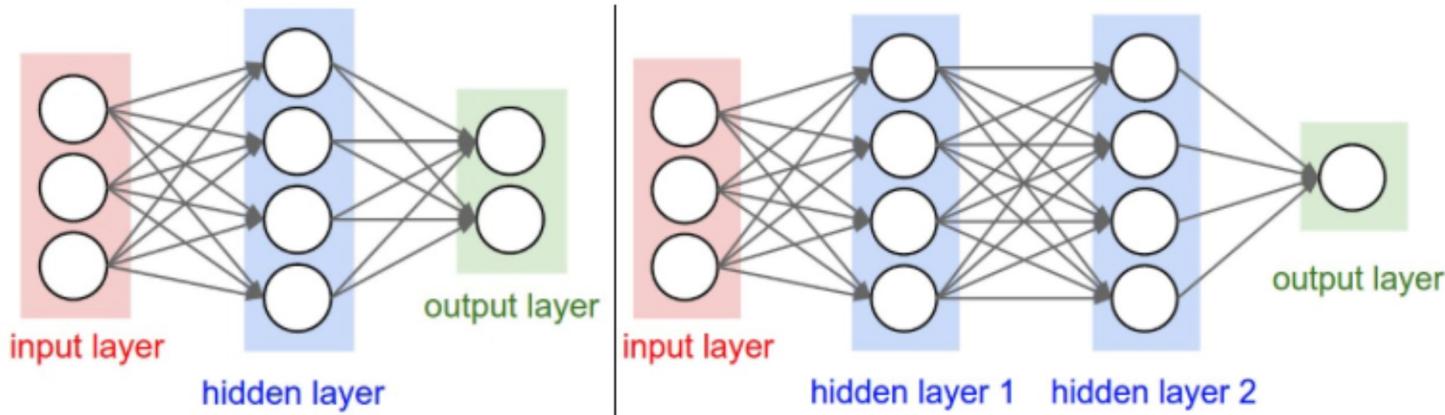
3- Do it for all the layers :

$$\hat{y} = f(\mathbf{W}_2 f(\mathbf{W}_1 x_1 + b_1) + b_2)$$



III- Multilayer perceptrons (1980's)

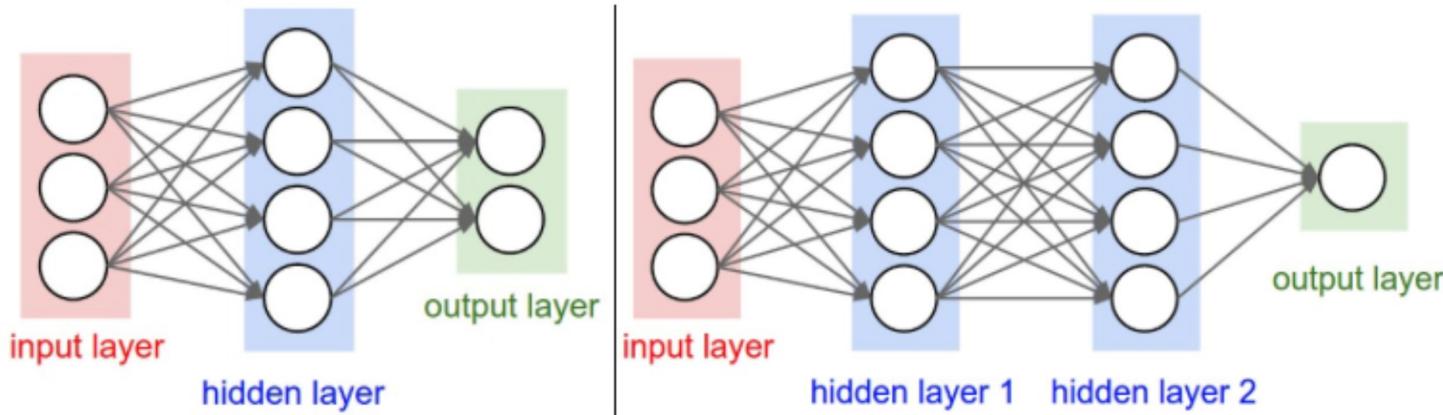
Can add more layers to increase capacity of the network



- New problem: MLPs are hard to train!

III- Multilayer perceptrons (1980's)

Can add more layers to increase capacity of the network



- New problem: MLPs are hard to train!
- ⇒ Solution: The **Backpropagation** algorithm

IV- Neural Network learning as optimization

Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton†
& Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California,
San Diego, La Jolla, California 92093, USA

† Department of Computer Science, Carnegie-Mellon University,
Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal ‘hidden’ units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure¹.

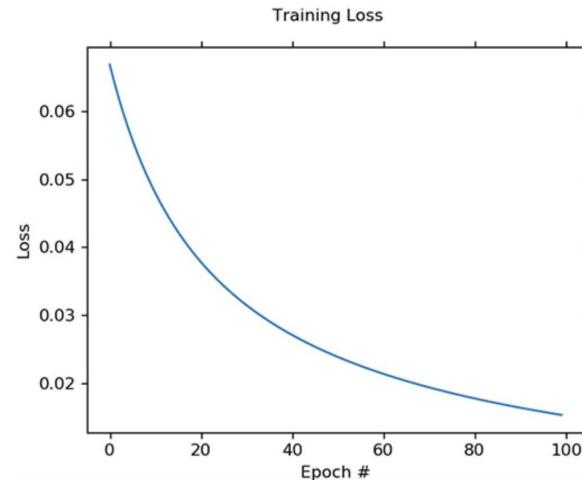
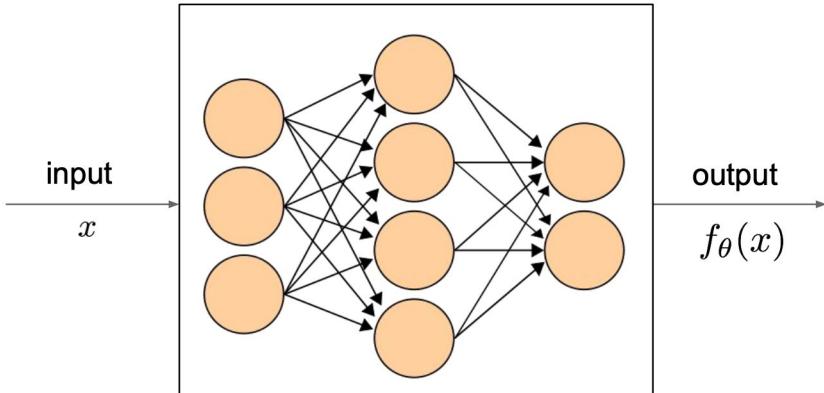


Rumelhart, Hinton, and Williams (1986) introduced Backpropagation to train MLPs

Principle: Computing the gradient of the cost function w.r.t the weights of the network

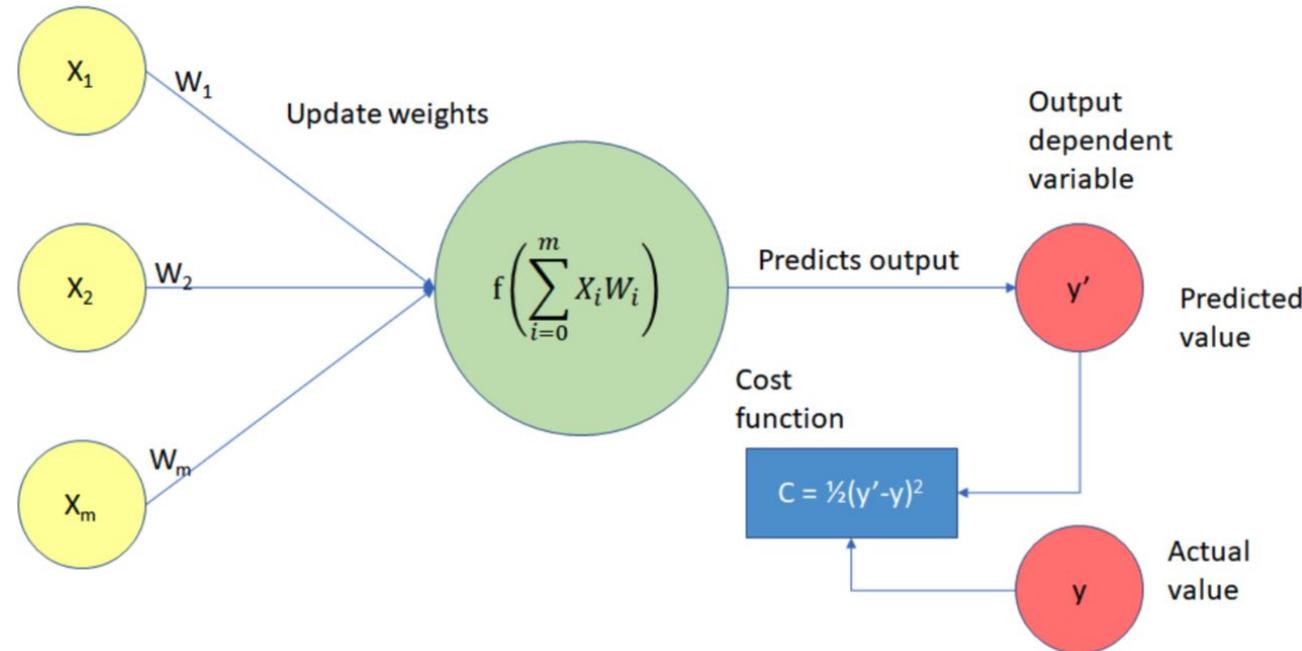
IV- Neural Network learning as optimization

- Mapping a set of inputs to a set of outputs from training data
- Learning is cast as an **optimization** problem to make good enough predictions
- Training with **gradient descent**



IV.A - Cost functions (Statistics / Probability)

- A cost function is a measure of error between predictions and true values



IV.A - Cost functions (Statistics / Probability)

- A cost function is a measure of error between predictions and true values
- Guides the training process to find a set of weights that minimizes its value

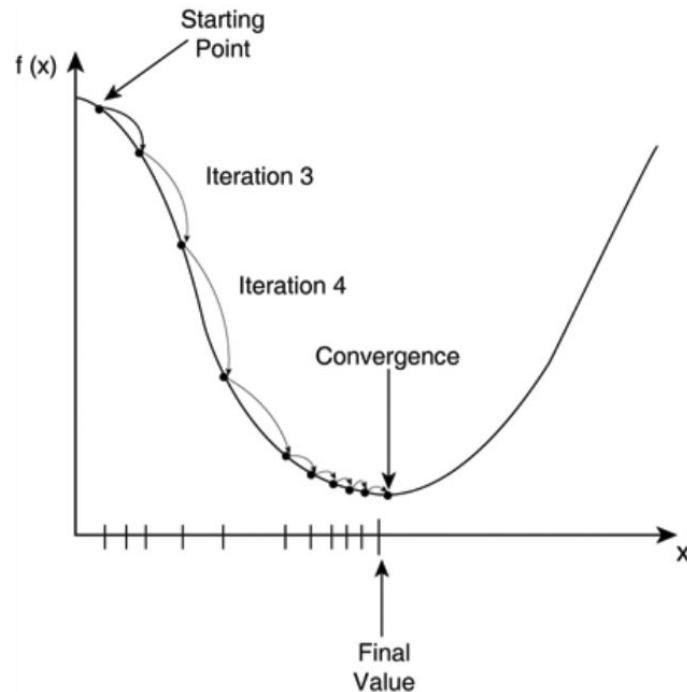
Some examples:

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

test set predicted value actual value

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

test set predicted value actual value



Recall Gradient (Optimization)

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

The **derivative** shows the sensitivity of change of a function's output with respect to the input.
It can be seen as the slope of the function at a point.

In our case :

- The function is the error computed at the end of the Neural Network.
- We want to see how the error is impact by each parameter of the model → higher dimension than 1

Solution : use the **gradient**

Gradient of a function f at point a with $f : \mathbb{R}^3 \rightarrow \mathbb{R}$

$$\nabla f(a) = \begin{bmatrix} \frac{\partial f}{\partial x}(a) \\ \frac{\partial f}{\partial y}(a) \\ \frac{\partial f}{\partial z}(a) \end{bmatrix}$$

Recall Gradient (Optimization)

Example :

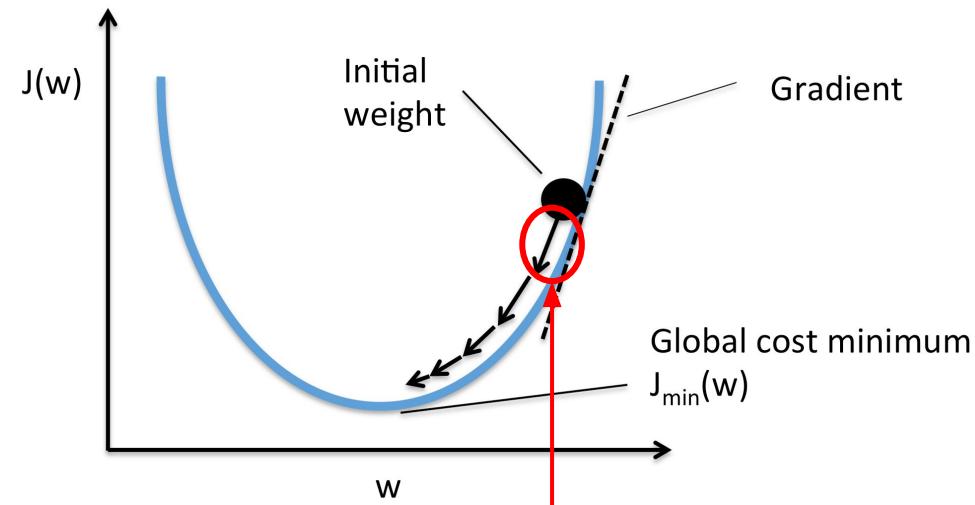
$$f(x, y, z) = x^2 + y * z \longrightarrow \nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{bmatrix} = \begin{bmatrix} 2x \\ z \\ y \end{bmatrix}$$

- We obtain the slope of the function f in each direction (x, y, z)
- The direction will be replaced by each parameter of the model (w and b)

IV.B- Gradient Descent (Optimization)

- Find the minimum of the cost function C
- The Negative gradient : $-\nabla C$ points in the direction where the function decreases most rapidly
- Calculate new weights: $W^+ = W - \eta \nabla C$

$$\begin{bmatrix} w_1^+ \\ w_2^+ \\ \vdots \\ w_n^+ \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} - \eta \begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial w_2} \\ \vdots \\ \frac{\partial C}{\partial w_n} \end{bmatrix}$$



- η is called the learning rate : increase or decrease the length of the **steps**

IV.B- Gradient Descent (Optimization)

- Find the minimum of the cost function C
- The Negative gradient : $-\nabla C$ points in the direction where the function decreases most rapidly
- Calculate new weights: $W^+ = \boxed{W} - \eta \boxed{\nabla C}$

Gradient descent algorithm

repeat until convergence {

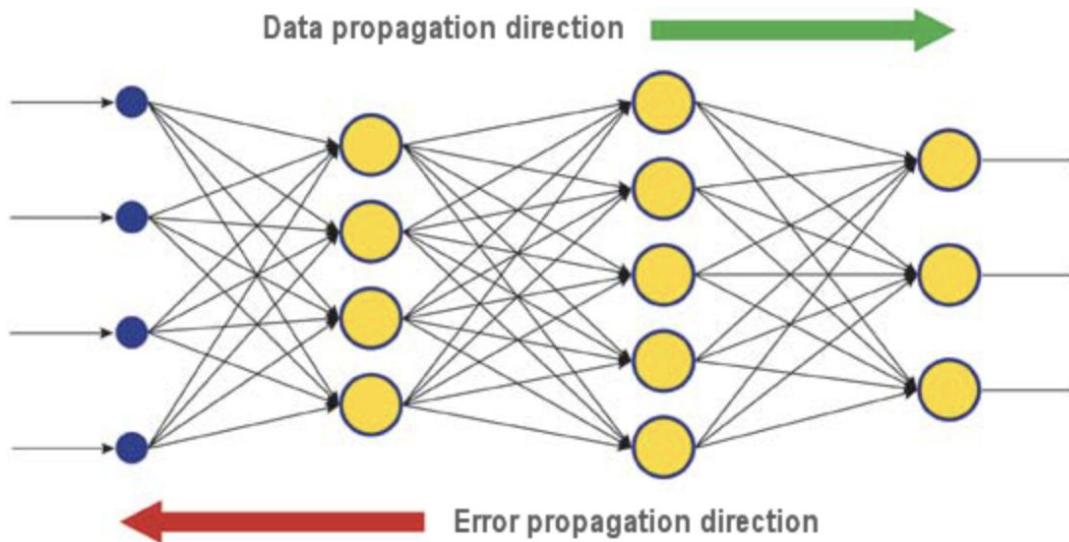
$$\theta_j := \boxed{\theta_j} - \alpha \boxed{\frac{\partial}{\partial \theta_j}} J(\theta_0, \theta_1)$$

(for $j = 1$ and $j = 0$)

}

IV.C- Backpropagation (Optimization)

- **Minimization** through **gradient descent** requires computing the gradient
- Backpropagation: way to compute the gradient by applying the **chain rule**



Recall (Optimization)

Chain rule :

$$\frac{d}{dx} [f(g(x))] = f'(g(x))g'(x)$$

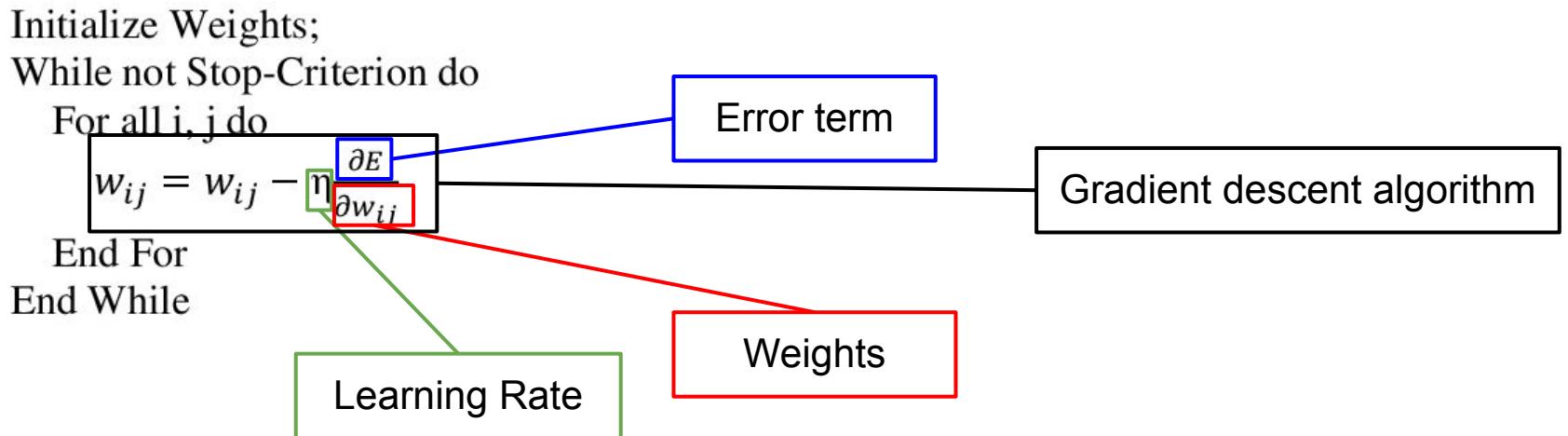
Example :

$$F(x) = f(g(x)) = \frac{1}{x^2}$$

$$F'(x) = f'(g(x))g'(x) = -\frac{1}{(x^2)^2} \cdot 2x = -\frac{2}{x^3}$$

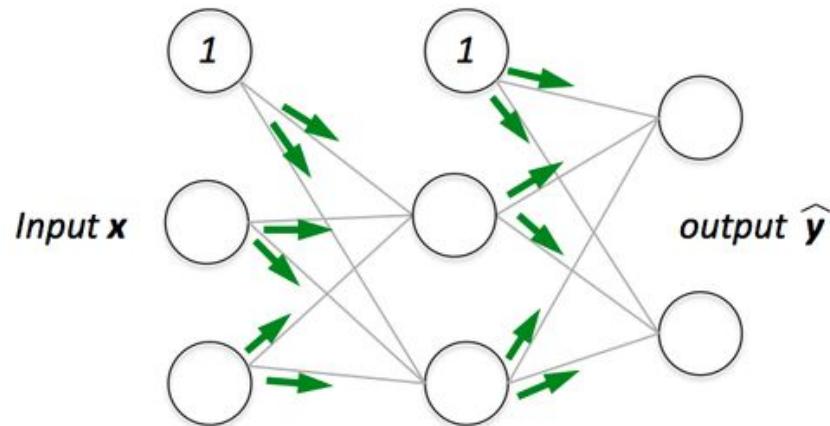
IV.C- Backpropagation (Optimization)

Backpropagation Algorithm :



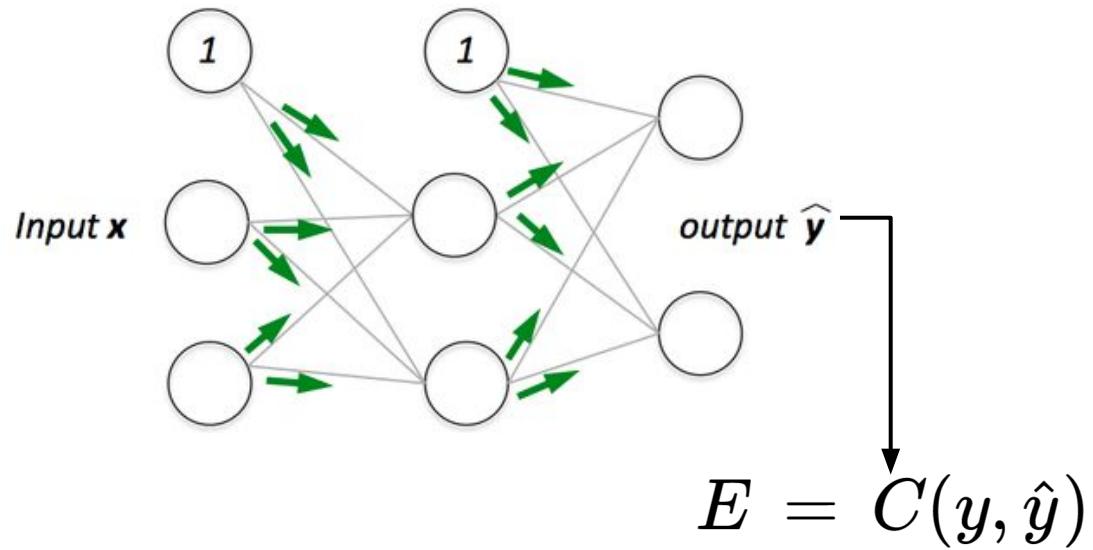
IV.C- Backpropagation (Optimization)

- 1) Forward pass: propagate data through the network to get predictions



IV.C- Backpropagation (Optimization)

- 1) Forward pass: propagate data through the network to get predictions
- 2) Calculate the total error with respect to the desired outputs



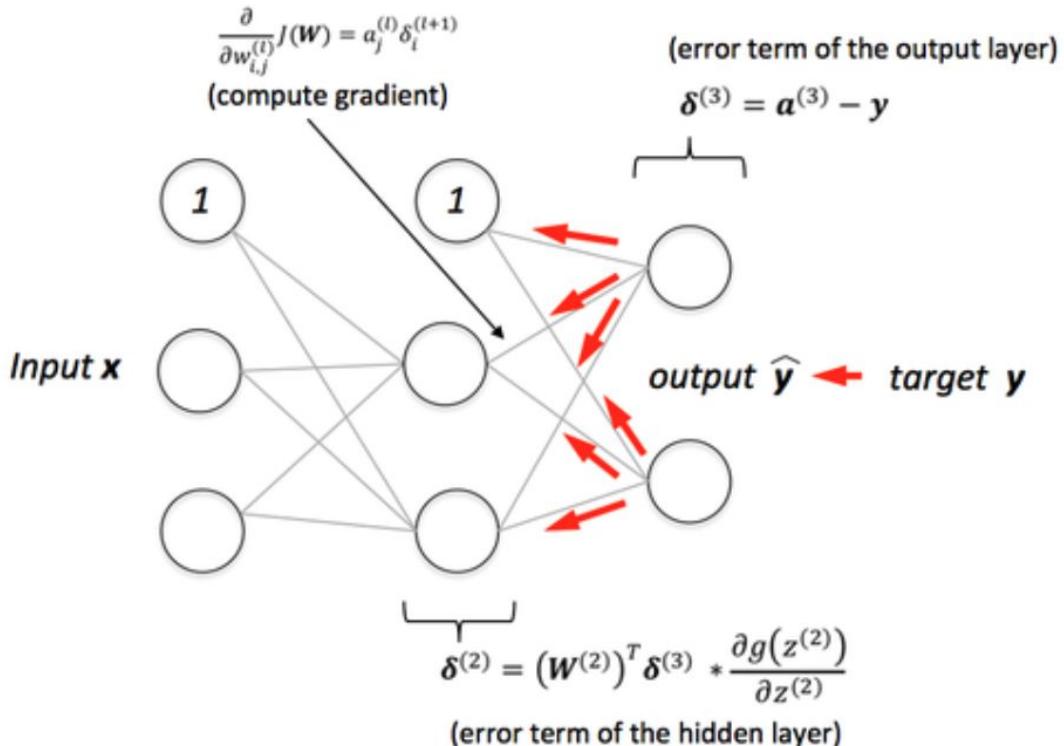
IV.C- Backpropagation (Optimization)

- 1) Forward pass: propagate data through the network to get predictions
- 2) Calculate the total error w.r.t desired outputs
- 3) Backward pass:

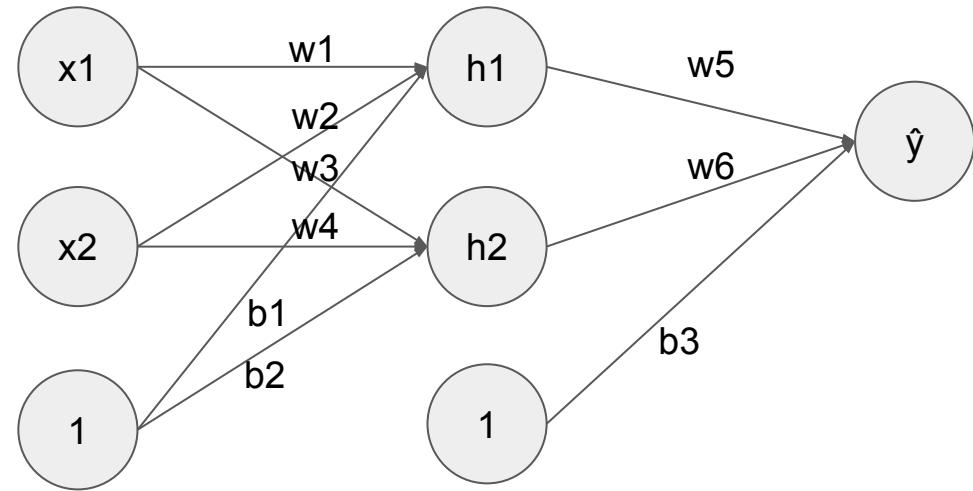
a) Compute partial derivatives of the error w.r.t each weight $\frac{dE}{dw_{ij}}$ by applying the **chain rule**

b) Update weights:

$$w_i = w_i - \eta \frac{dE}{dw_{ij}}$$

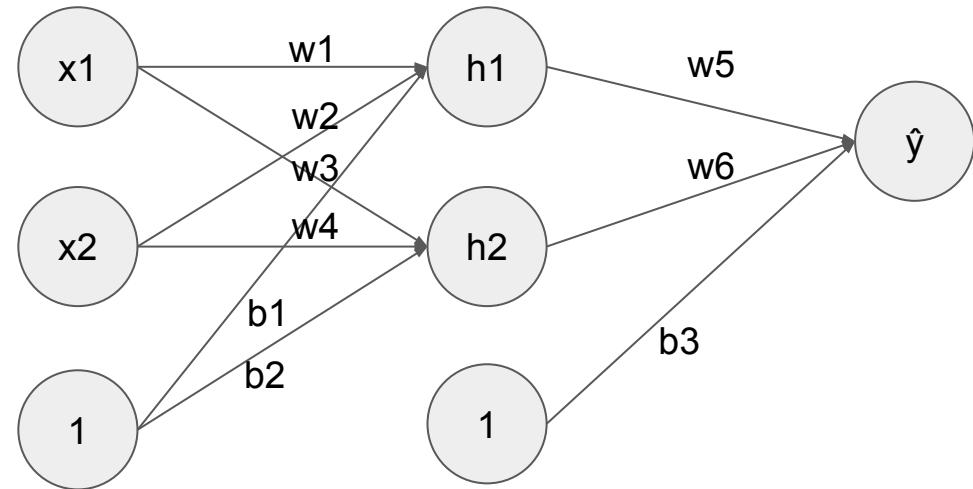


IV.C- Backpropagation (Optimization)



1- Compute error term : $E = (\hat{y} - y)^2$

IV.C- Backpropagation (Optimization)

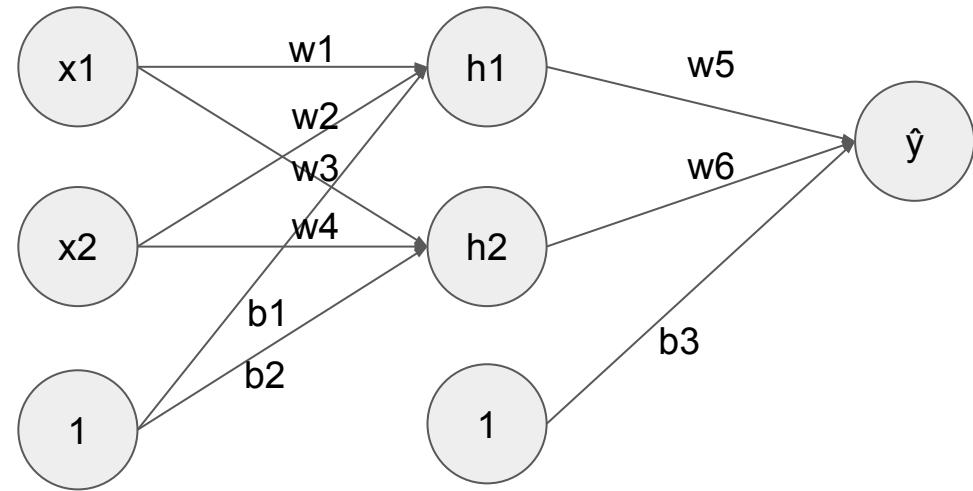


1- Compute error term : $E = (\hat{y} - y)^2$

2- Update weight : $w_5 = w_5 - \eta \frac{dE}{dw_5}$

$$\text{Error term : } E = (f(w_5 f(w_1 x_1 + w_2 x_2 + b_1) + w_6 f(w_3 x_1 + w_4 x_2 + b_2) + b_3) - y)^2$$

IV.C- Backpropagation (Optimization)



1- Compute error term : $E = (\hat{y} - y)^2$

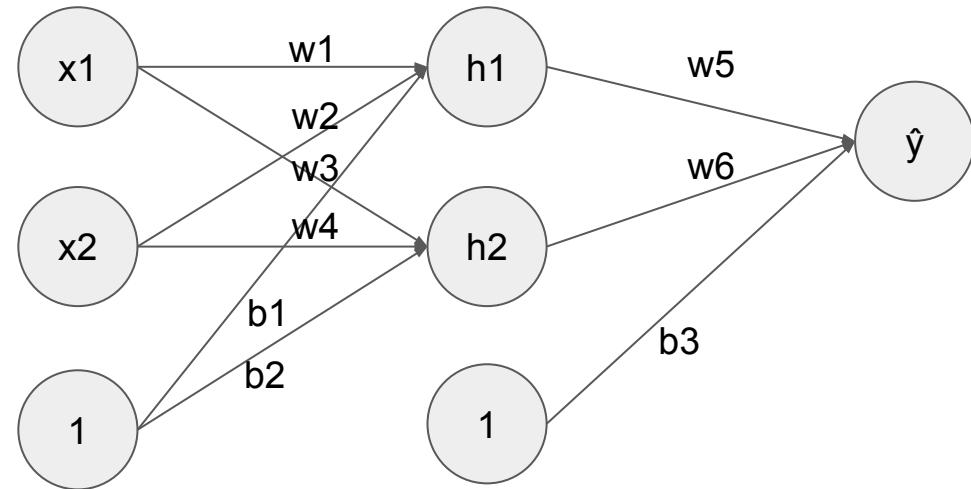
2- Update weight : $w_5 = w_5 - \eta \frac{dE}{dw_5}$

$$\text{Error term : } E = (f(w_5 f(w_1 x_1 + w_2 x_2 + b_1) + w_6 f(w_3 x_1 + w_4 x_2 + b_2) + b_3) - y)^2$$

$$\text{chain rule : } \frac{dE}{dw_5} = \frac{dE}{d\hat{y}} \cdot \frac{d\hat{y}}{ds} \cdot \frac{ds}{dw_5} \quad \text{with : } s = w_5 h_1 + w_6 h_2 + b_3$$

$$\frac{dE}{dw_5} = 2(\hat{y} - y) \cdot f'(w_5 h_1 + w_6 h_2 + b_3) \cdot f(w_1 x_1 + w_2 x_2 + b_1)$$

IV.C- Backpropagation (Optimization)



1- Compute error term : $E = (\hat{y} - y)^2$

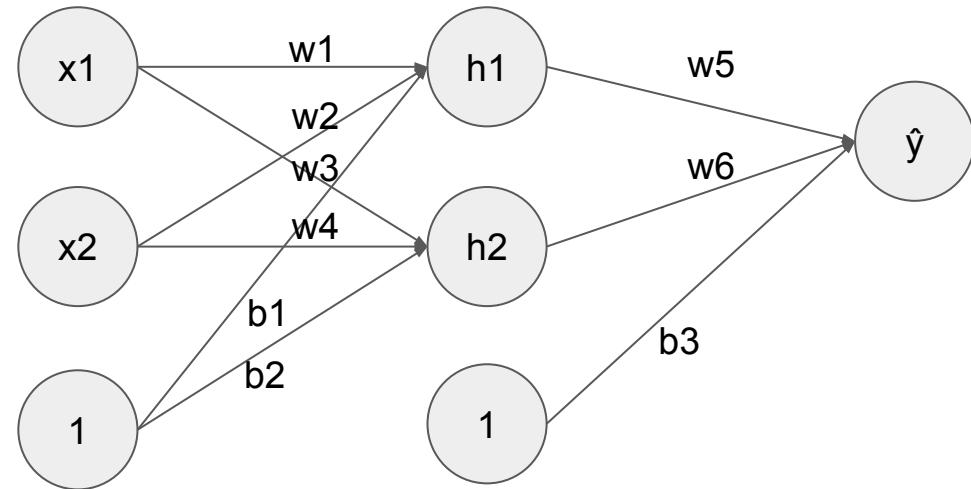
2- Update weight : $w_1 = w_1 - \eta \frac{dE}{dw_1}$

$$\frac{dE}{dw_1} = \frac{dE}{dh_1} \cdot \frac{dh_1}{ds_1} \cdot \frac{ds_1}{dw_1}$$

$$\frac{dE}{dw_1} = \frac{dE}{d\hat{y}} \cdot \frac{d\hat{y}}{ds_3} \cdot \frac{ds_3}{dh_1} \cdot \frac{dh_1}{ds_1} \cdot \frac{ds_1}{dw_1} \quad s_3 = w_5 h_1 + w_6 h_2 + b_3 \\ s_1 = w_1 x_1 + w_2 x_2 + b_1$$

$$\frac{dE}{dw_1} = 2(\hat{y} - y) \cdot f'(s_3) \cdot w_5 \cdot f'(s_1) \cdot x_1$$

IV.C- Backpropagation (Optimization)



1- Compute error term : $E = (\hat{y} - y)^2$

2- Update weight : $w_1 = w_1 - \eta \frac{dE}{dw_1}$

$$\text{chain rule : } \frac{dE}{dw_1} = \frac{dE}{dh_1} \cdot \frac{dh_1}{ds_1} \cdot \frac{ds_1}{dw_1}$$

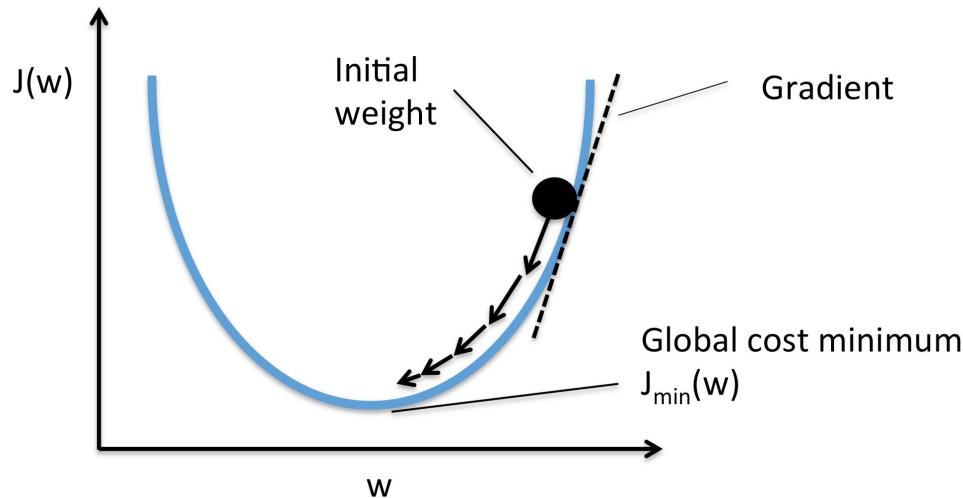
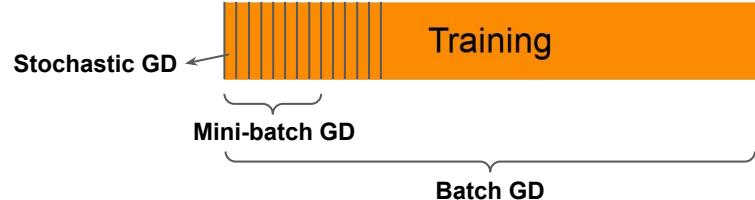
$$\frac{dE}{dw_1} = \frac{dE}{d\hat{y}} \cdot \frac{d\hat{y}}{ds_3} \cdot \frac{ds_3}{dh_1} \cdot \frac{dh_1}{ds_1} \cdot \frac{ds_1}{dw_1}$$

$$\frac{dE}{dw_1} = 2(\hat{y} - y) \cdot f'(s_3) \cdot w_5 \cdot f'(s_1) \cdot x_1$$

$$\text{with : } \begin{aligned} s_3 &= w_5 h_1 + w_6 h_2 + b_3 \\ s_1 &= w_1 x_1 + w_2 x_2 + b_1 \end{aligned}$$

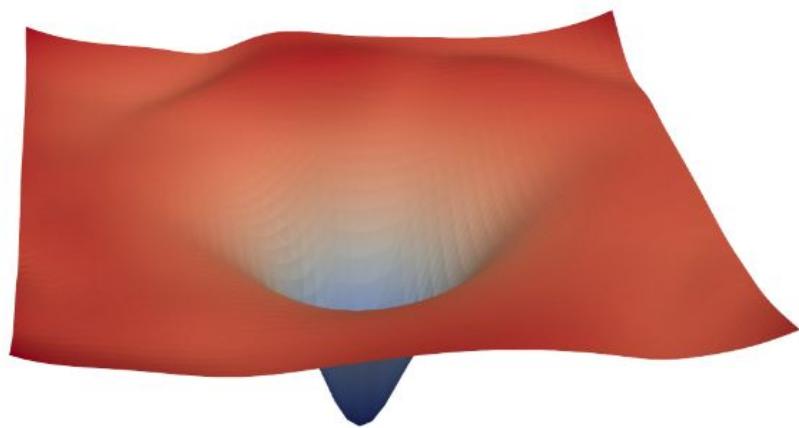
IV.C- Backpropagation (Optimization)

- **batch GD, stochastic, or mini-batch?**
- **SGD** in DL generally refers to mini-batch GD

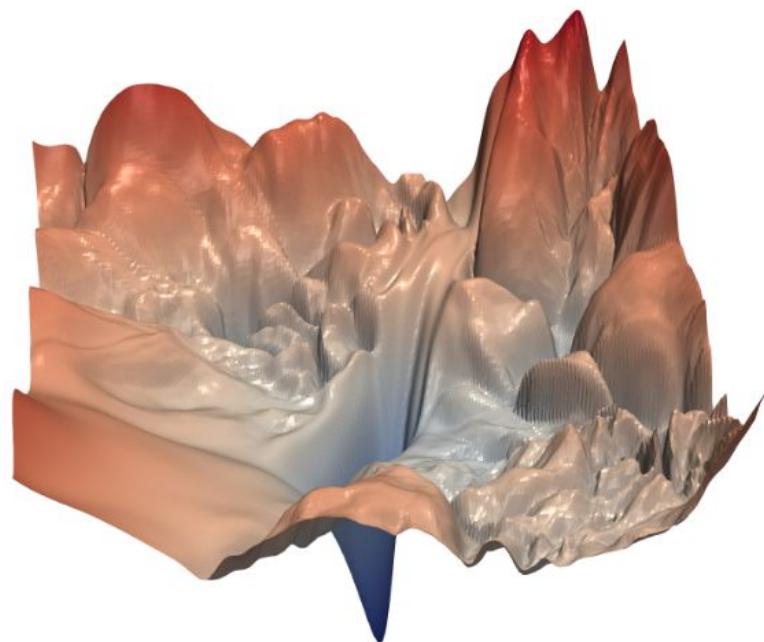


Training with Gradient descent and BP (Optimization)

Local VS global minimum



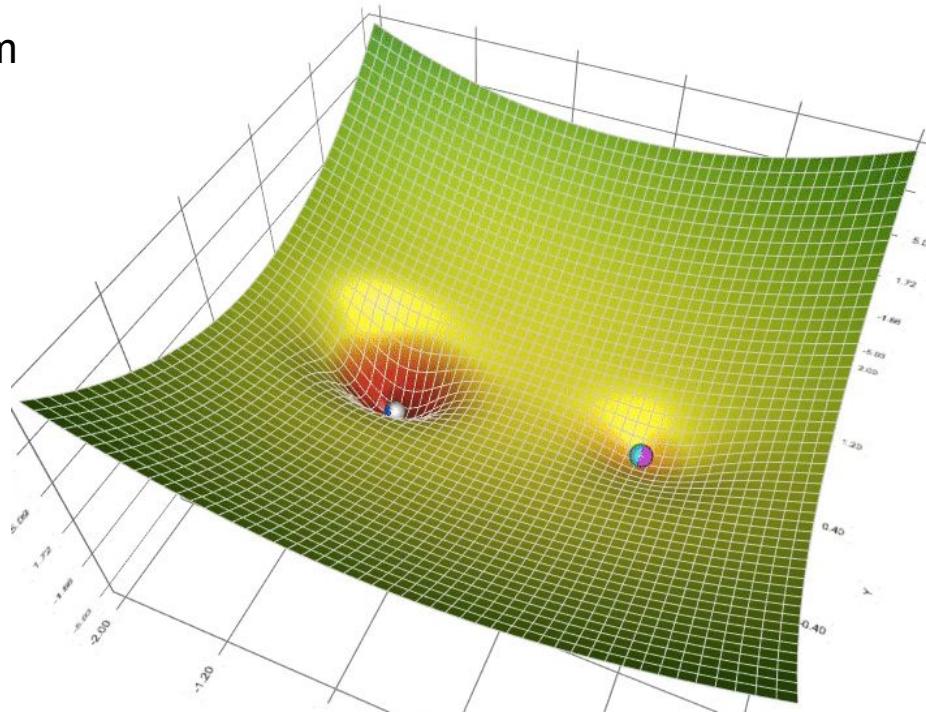
One global minimum



One global minimum & Multiple local minimum

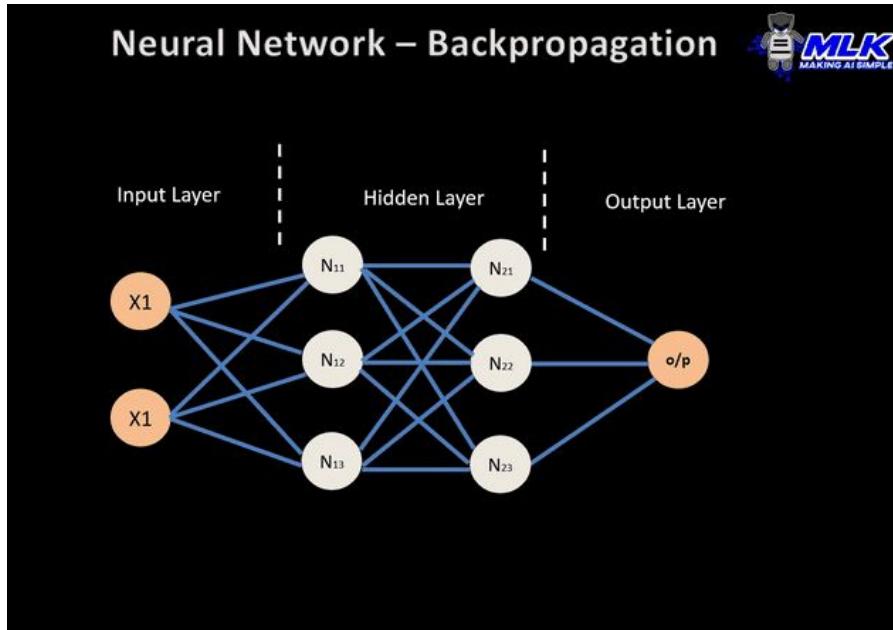
Training with Gradient descent and BP (Optimization)

Local VS global minimum



Depending on initialization the results can end up in a local or global minimum

Training with Gradient descent and BP (Optimization)



Coding skills

Not so much



```
pytorch_example.py  x
61 # define the training procedure
62 # i.e. one step of gradient descent
63 # there are lots of steps
64 # so we encapsulate it in a function
65 # Note: inputs and labels are torch tensors
66 def train(model, loss, optimizer, inputs, labels):
67     # https://discuss.pytorch.org/t/why-is-it-recommended-to-wrap-your-data-with-variable/
68     inputs = Variable(inputs, requires_grad=False)
69     labels = Variable(labels, requires_grad=False)
70
71     # Reset gradient
72     # https://discuss.pytorch.org/t/why-do-we-need-to-set-the-gradients-manually-to-zero-in-backward/
73     optimizer.zero_grad()
74
75     # Forward
76     logits = model.forward(inputs)
77     output = loss.forward(logits, labels)
78
79     # Backward
80     output.backward()
81
82     # Update parameters
83     optimizer.step()
84
85     # what's the difference between backward() and step()?
```

Some useful resources

<http://neuralnetworksanddeeplearning.com/chap1.html>

<https://towardsdatascience.com/part-2-gradient-descent-and-backpropagation-bf90932c066a>

<https://towardsdatascience.com/a-concise-history-of-neural-networks-2070655d3fec#.ekc89166m>

<https://people.idsia.ch/~juergen/who-invented-backpropagation.html>

Part 2 : Implementation of a Neural Network

Training a neural network

Data

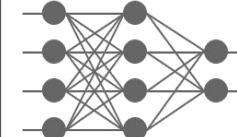
Data Samples
and Labels

$$\{(x_1, y_1), (x_2, y_2), \dots\}$$

Task

Input and output
 $x_i \rightarrow y_i$

Architecture



Loss function

$$loss = diff(prediction, label)$$

Optimization

prediction → backpropagation → gradient descent

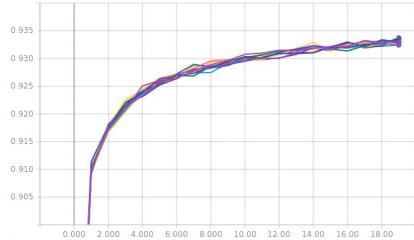


Training a neural network

Data

Handwritten Digit Dataset

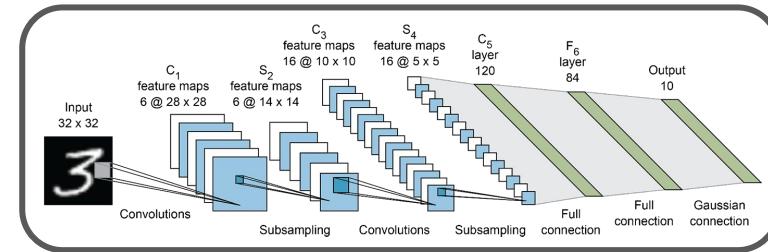
```
0 0 0 0 0 0 0 0 0  
1 1 1 1 1 1 1 1  
2 2 2 2 2 2 2 2  
3 3 3 3 3 3 3 3  
4 4 4 4 4 4 4 4  
5 5 5 5 5 5 5 5  
6 6 6 6 6 6 6 6  
7 7 7 7 7 7 7 7  
8 8 8 8 8 8 8 8  
9 9 9 9 9 9 9 9
```



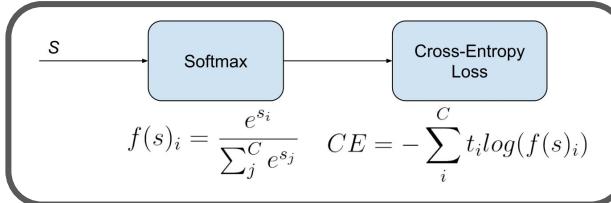
Task



Architecture



Loss function

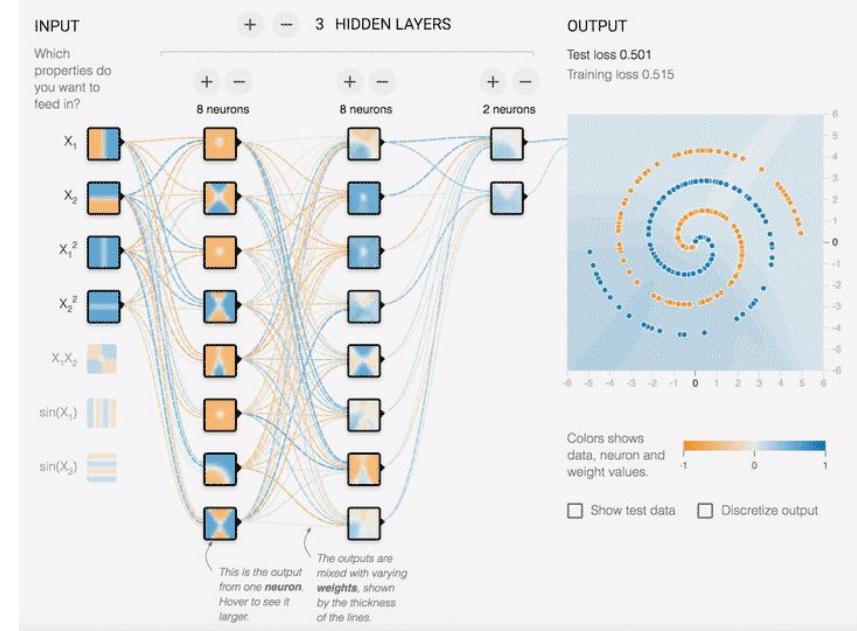
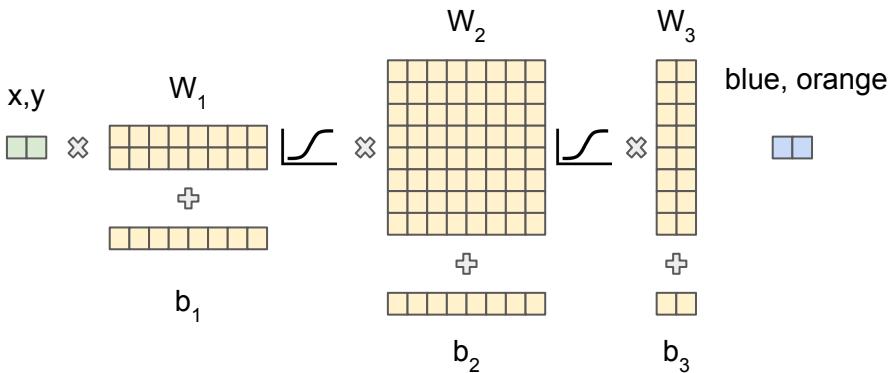


Optimization



I- How to choose the architecture

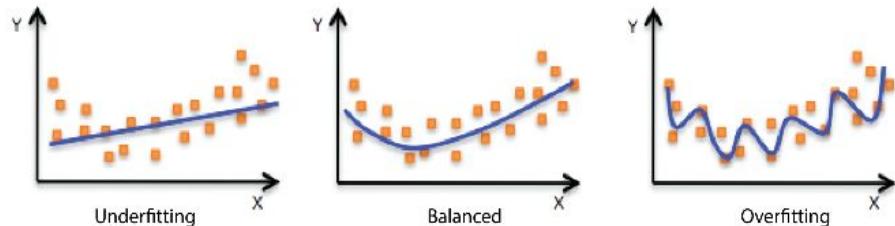
Multi-layer perceptrons **MLPs** are the standard solution for data with simple structure (ex. tabular data).



I- How to choose the architecture

What's important ?

- Generalization
 - Fitting the data distribution
 - Fitting unseen examples
- Efficiency
 - Memory (how large is the model ?)
 - Time (how long does it take to train ?)
 - Data (how many training samples does it need ?)



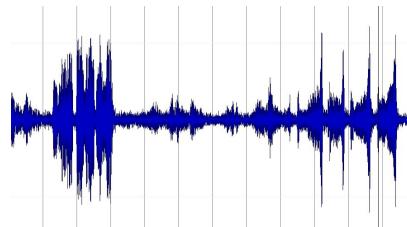
I- How to choose the architecture

Do MLPs work for all types of data ? **Yes, but not efficiently**

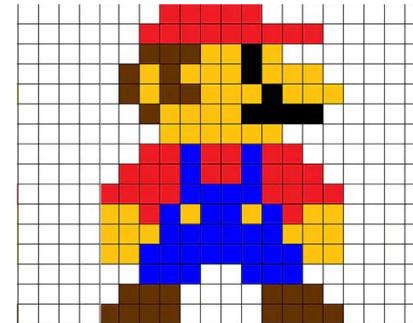
text

I am a student

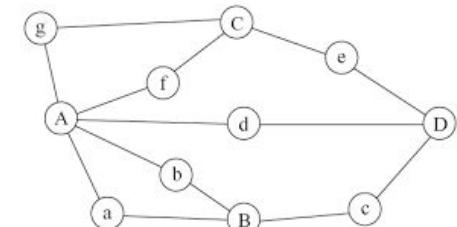
time
sequences



images

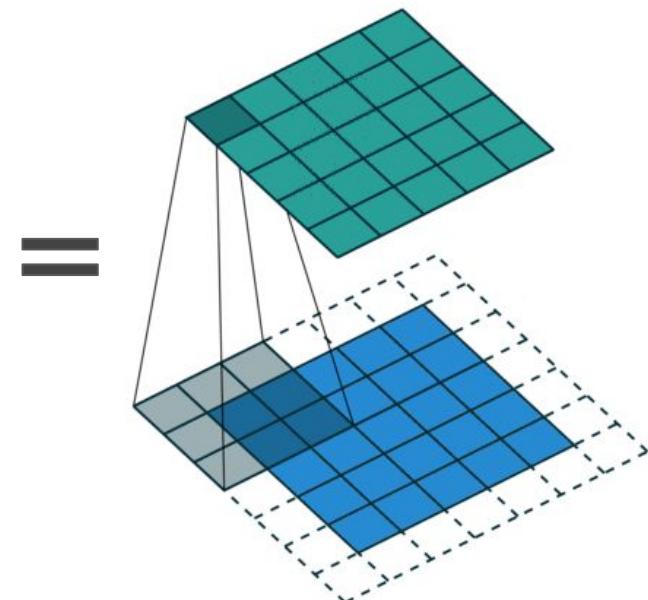
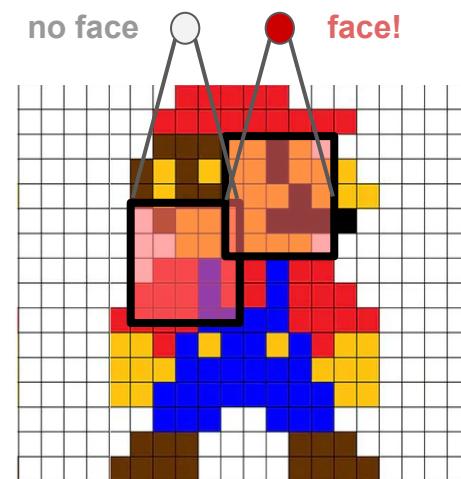
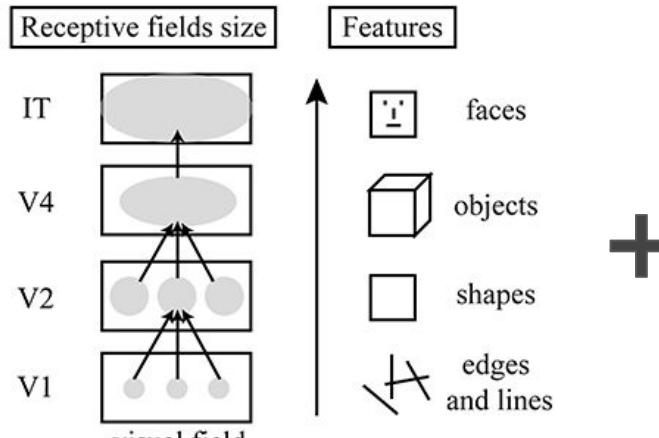


graphs



Inductive bias : an architectural assumption or constraint

I.A- Convolutional Neural Network (CNN)



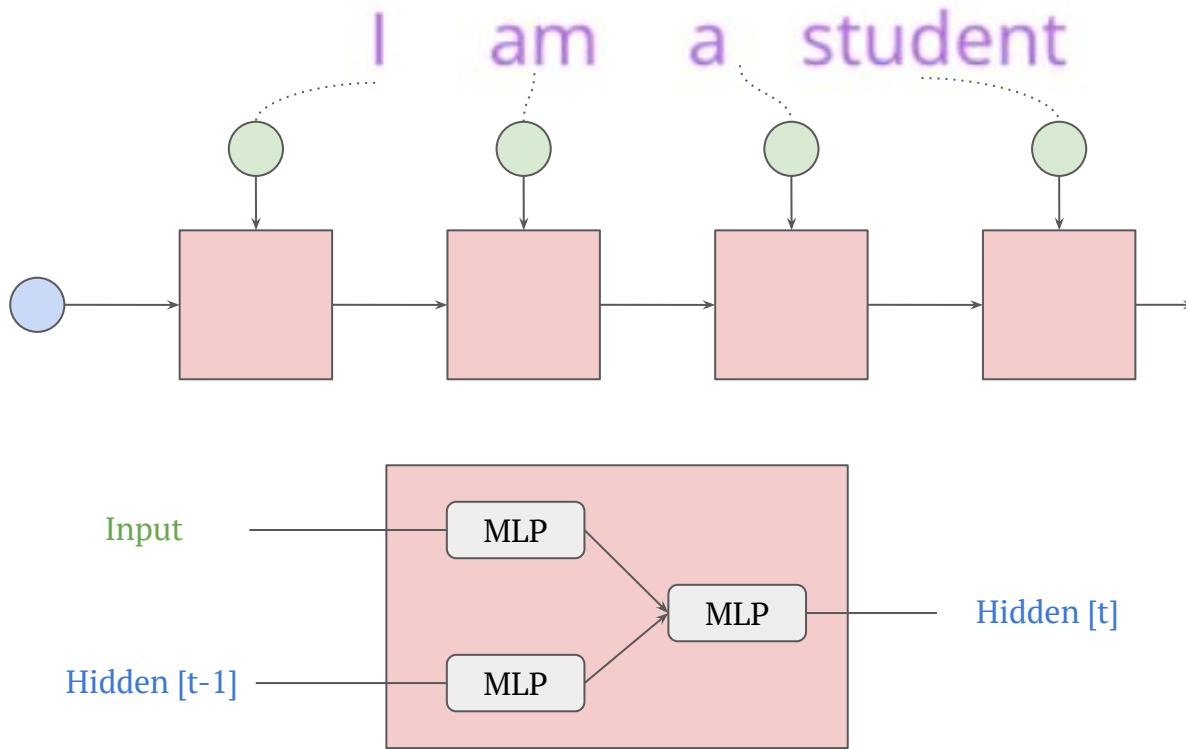
Local Connectivity

Hierarchical Processing

Weight sharing

Convolution

I.B- Recurrent neural networks (RNN)

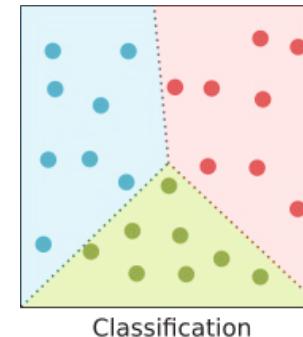
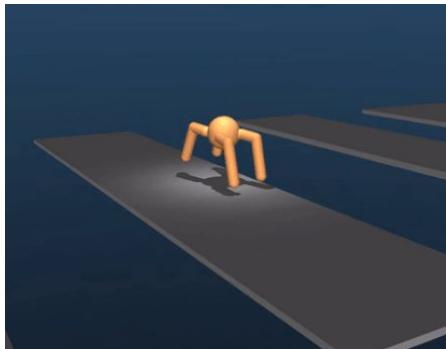


II- How to choose the loss

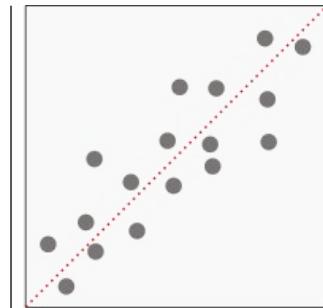
The choice of the loss is crucial !

What's important in the loss design?

- Adaptability to the problem (correlates with performance metrics)
- Continuous and differentiable
- Numerically stable



Classification

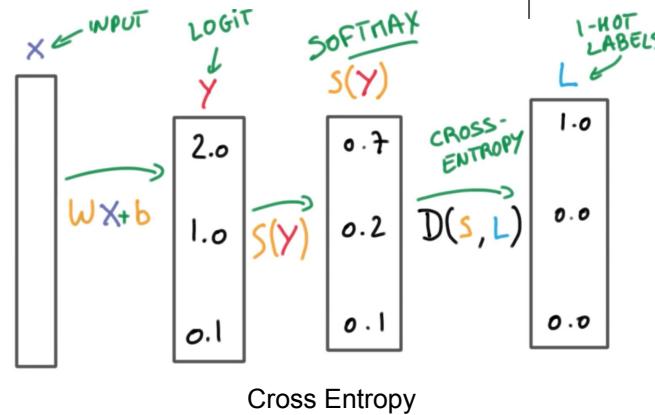


Regression

Label : Categorical (discrete)

Examples : Cross Entropy
Hinge Loss

Mean Square Error (L2)
Mean Absolute Error (L1)



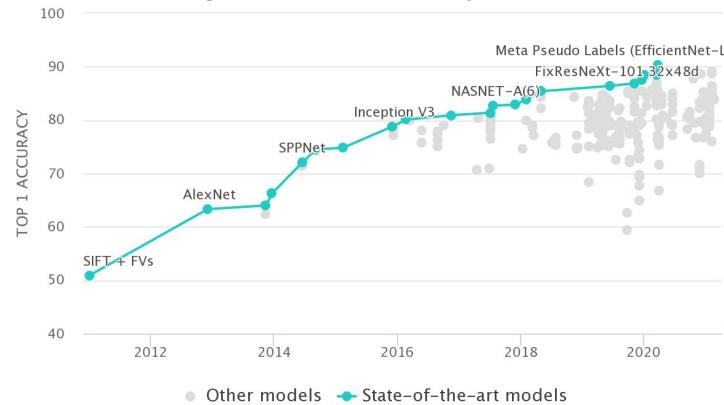
III - Training and Evaluation^A

Training: optimization of the model

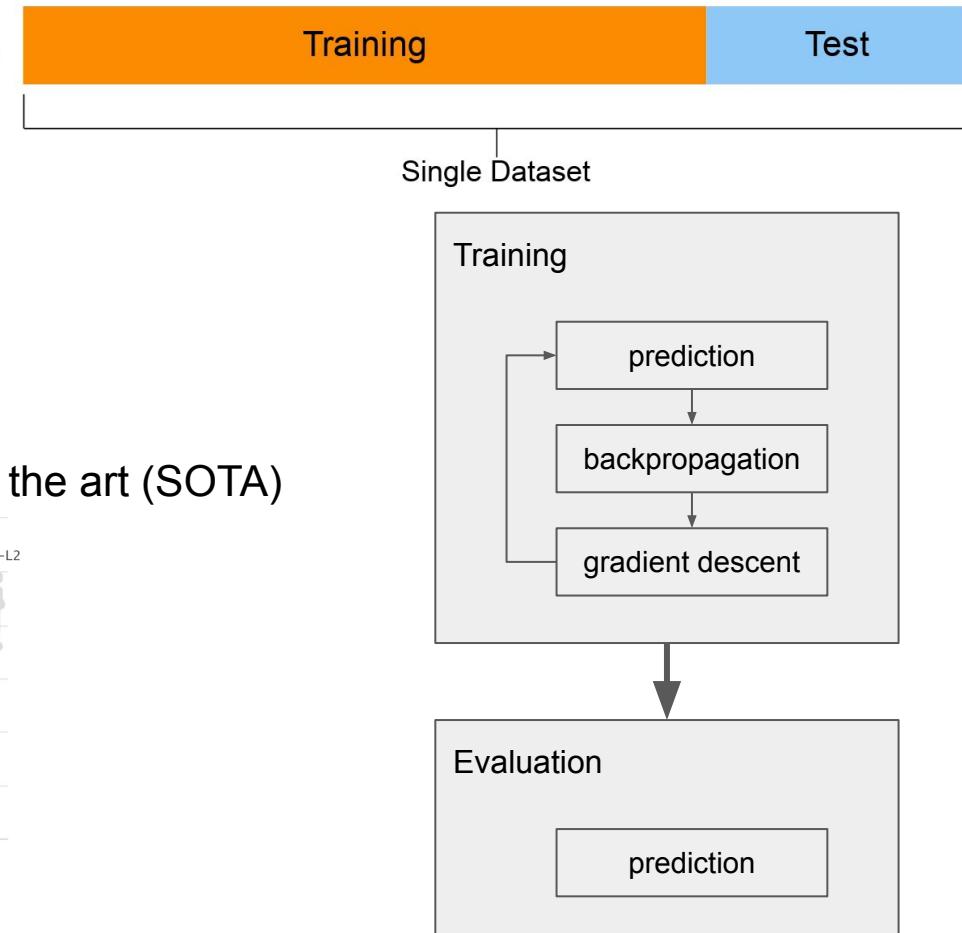
Evaluation: testing generalization

Metrics: Loss and accuracy

Models with the highest accuracy are state of the art (SOTA)



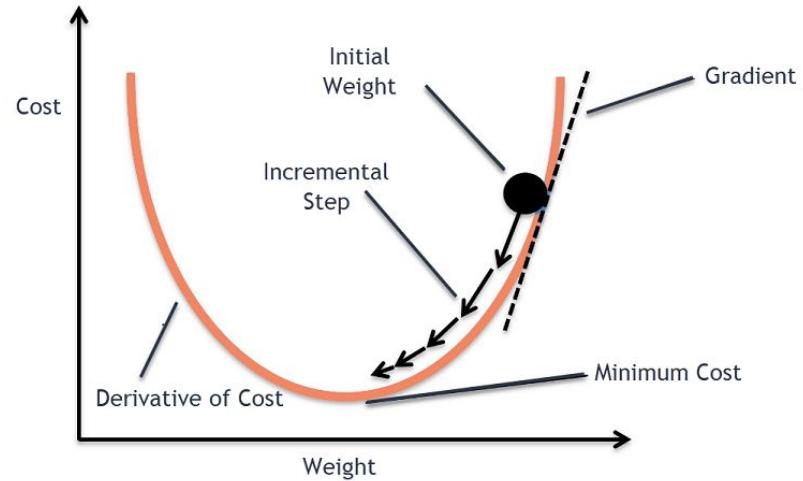
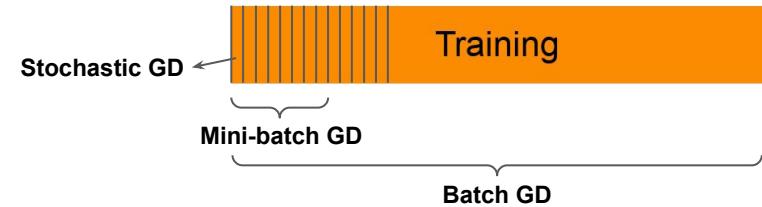
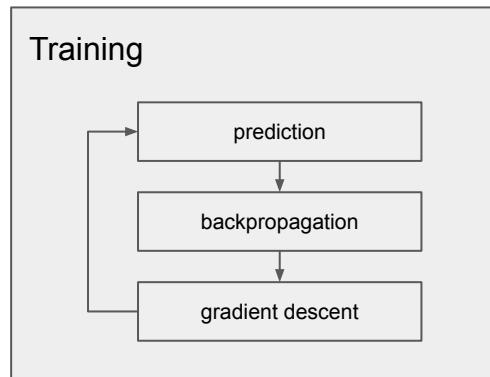
Imagenet Benchmark on paperswithcode.com



III - Training and Evaluation

SGD in DL generally refers to mini-batch GD

Epoch: one pass through the dataset

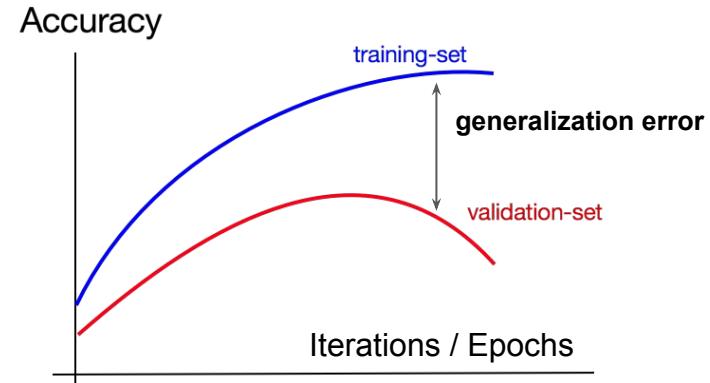
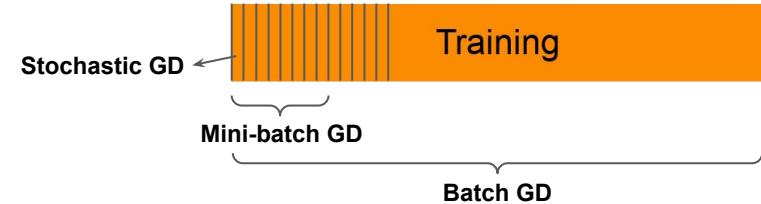


III - Training and Evaluation

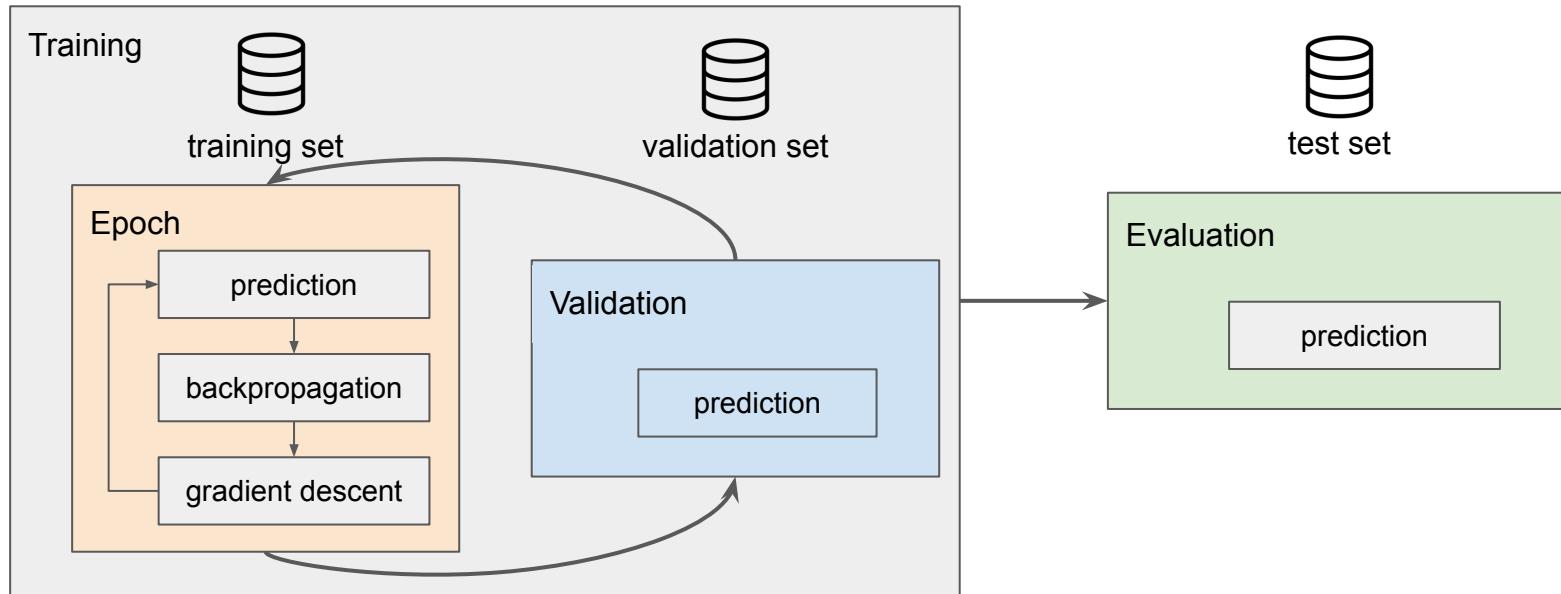
How many iterations/epochs ?

The validation set is used for early stopping

Test set \neq validation set



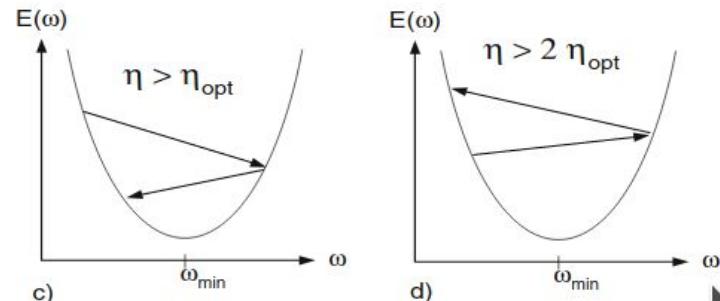
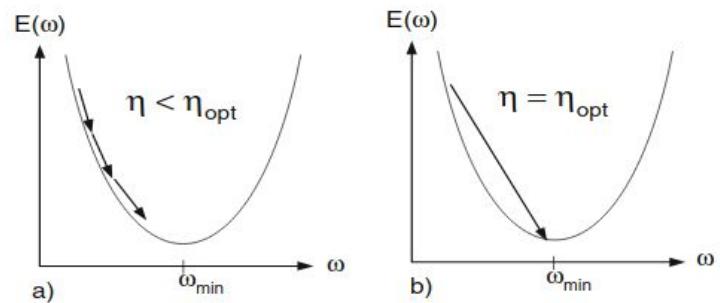
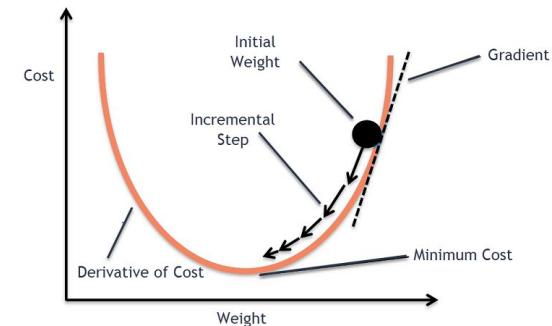
III - Training and Evaluation



III - Training and Evaluation

How to find the learning rate ? η

$$\theta \leftarrow \theta - \eta \frac{\partial \mathcal{L}}{\partial \theta}$$



IV- Optimizers

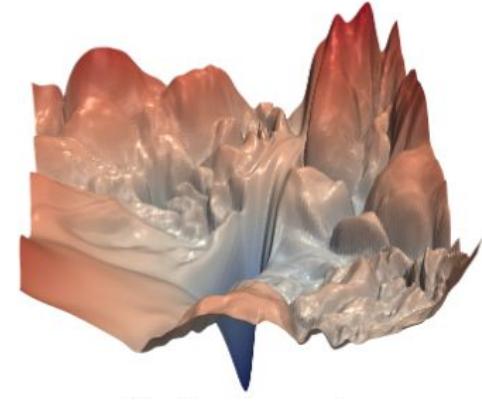
$$\theta \leftarrow \theta - \eta \frac{\partial \mathcal{L}}{\partial \theta}$$

Loss landscapes are not easy to navigate for optimizers

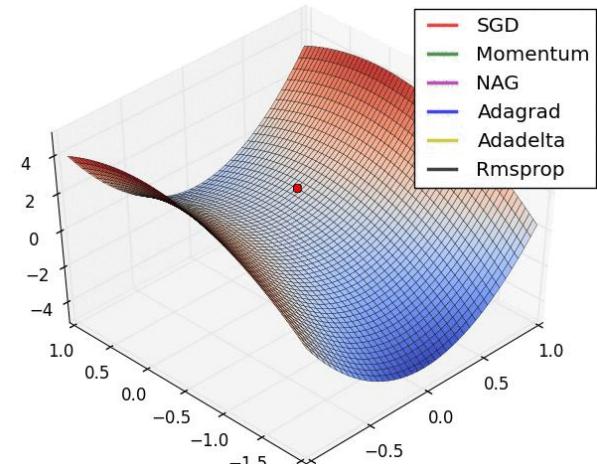
Ideas:

- Use the gradient history of previous timesteps to inform GD in future timesteps
- Adapt the learning rate to each parameter

Adam optimizer is an extension of SGD which makes use of these two ideas. It is currently the most used optimizer in DL after SGD.



This is a 2 parameter example!
Imagine millions



V- Regularization

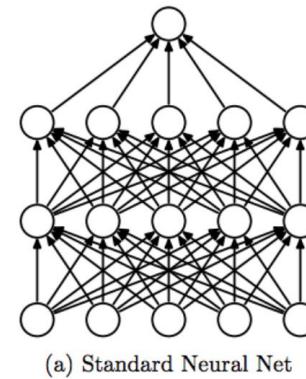
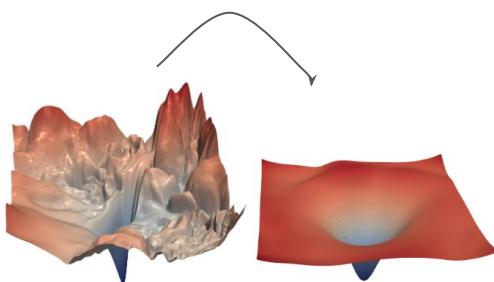
Additional constraints to reduce overfitting

Dropout: stochastically dropping weights during inference

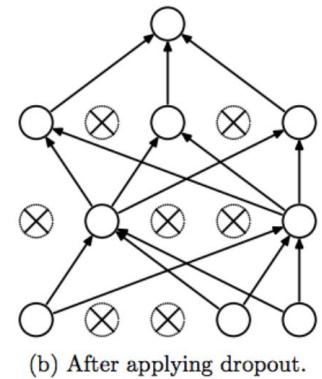
Early stopping: stopping the training as soon as the validation loss starts increasing

Weight penalties (weight decay): L1 norm (Lasso) / L2 norm (ridge). Terms added to the loss.

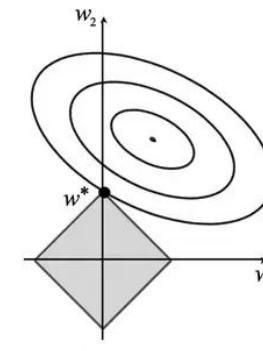
Data augmentations: artificially boosting the number of training samples



(a) Standard Neural Net

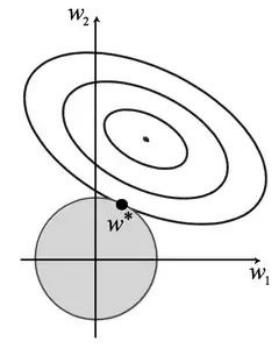


(b) After applying dropout.



L1

Sparse
weights



L2

Smaller weight
values

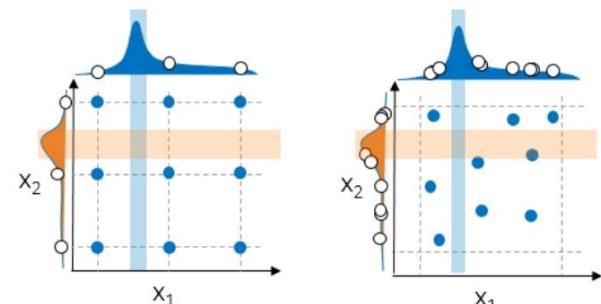
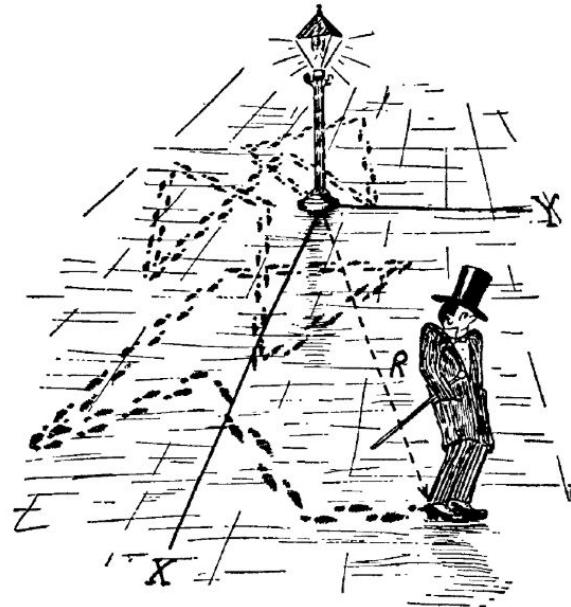
VI- Hyperparameters

All parameters and settings that are set before training:

- Architectural choices: types and number of layers, size of each layer
- Losses/regularization: weights, additional constants
- Optimization: batch size, learning rate, iterations/epochs, schedule

Hyperparameter search. Another optimization problem ?

- Often done manually.
- When resources are available, large scale search is possible



(a) Standard Grid Search

(b) Random Search

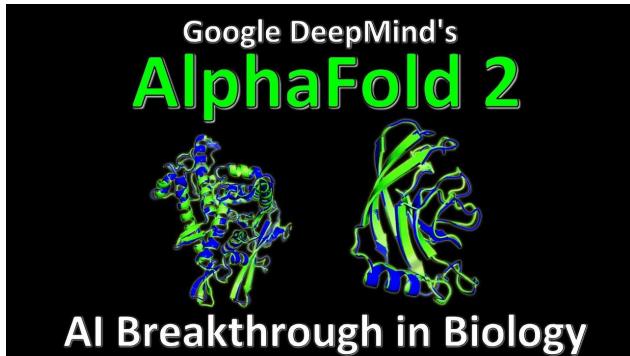
The machine learner pipeline

1. Understanding the data and the task
2. Set up the end-to-end training/evaluation skeleton
 - a. a basic architecture
 - b. a standard loss
 - c. Standard optimization pipeline
3. Complexify one thing at a time
4. Regularize
5. Tune Hyperparameters

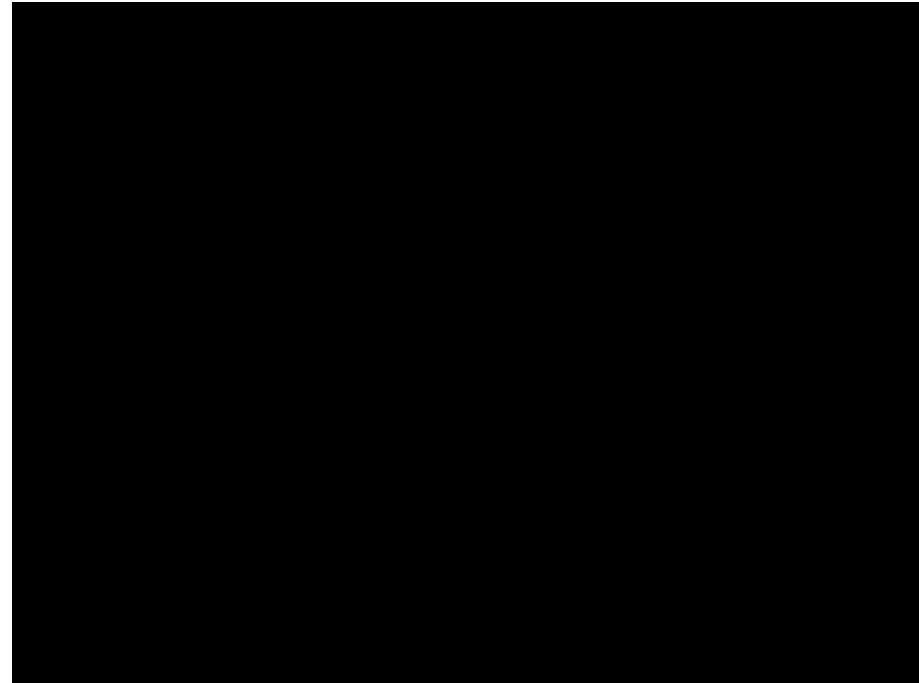
Practical examples



GPT-4 DALL·E 2



Midjourney



Palm-E

Practical Work