AI for Astrophysics: Simple Neural Networks

David Cornu

LERMA, Observatoire de Paris, PSL

Doctoral course – ED AAIF
2024/2025

The brain as model

« There is a fantastic existence proof that learning is possible, which is the bag of water and electricity (together with a few trace chemicals) sitting between your ears [...] which is the squishy thing that your skull protects »

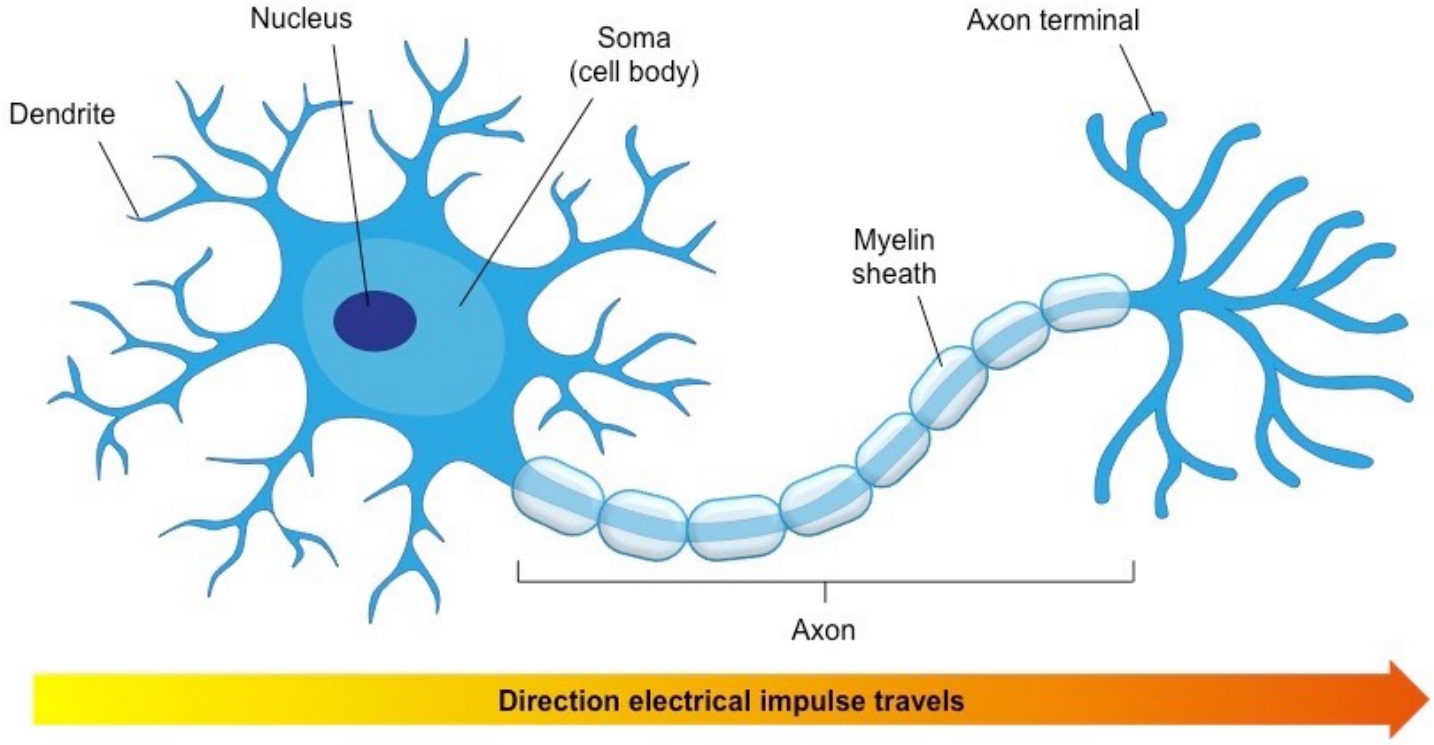
**Stephen Marsland*

The brain does exactly what we want for data analysis:

- Extract complex information in a compressed form
- Deal with noisy and/or inconsistent data
- Work in highly-dimensional spaces
- Give the appropriate answer most of the time
- Provide results very quickly
- Remain robust through aging (neuron loss)

The biological neuron

Elementary brick of a biological brain (10^{11} in the human brain).
The idea will be to use it as a model to emulate learning capabilities.



Perform the sum of various electrochemical input signals. If this total signal is sufficient, it sends a new signal through its axon to transfer information (toward other neurons).

Biological Neural Networks

A connection between two neurons is called a synapse (10^{14} in the human brain).
A neuron is a binary compute unit that either “fire” or “not-fire” in response to a signal.

➡ **In this view, a brain is a massively parallel super-computer of 10^{11} processing units.**

A simplified view of how it learns

The synapses represent the “strength” of the connection between two neurons.

Learning = modifying this connection (either positive or negative and changing its intensity)

➡ **This is called plasticity**

The Hebb law defines a simplified rule for learning:

If two neurons “fire” at the same time, there must be some **correlation** between them, and their connection must be strengthened.

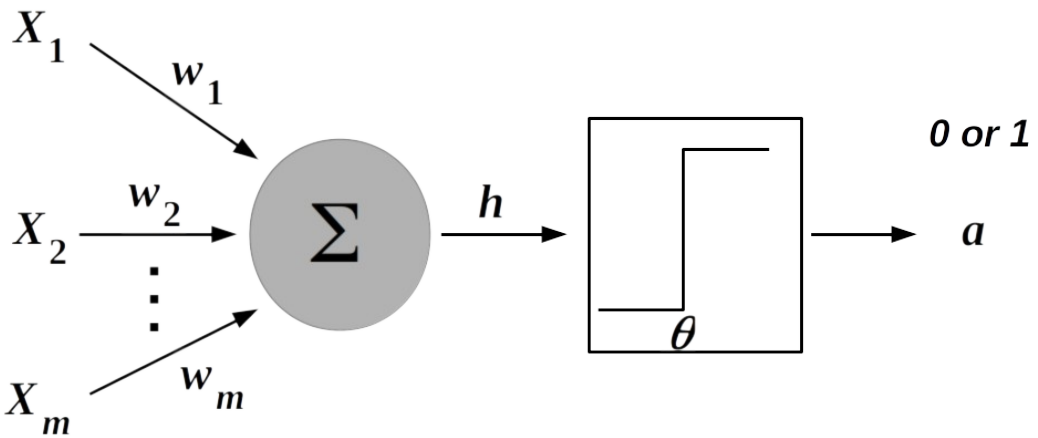
➡ **This is called conditioning**

**These rules are not enough to train a neural system but illustrate the processes that occur in the biological brain.*

To create an algorithm from these biological concepts, one first need to create a mathematical model.

Model of a Neuron

Mathematical model from McCulloch and Pitts, inspired by the biological neuron.

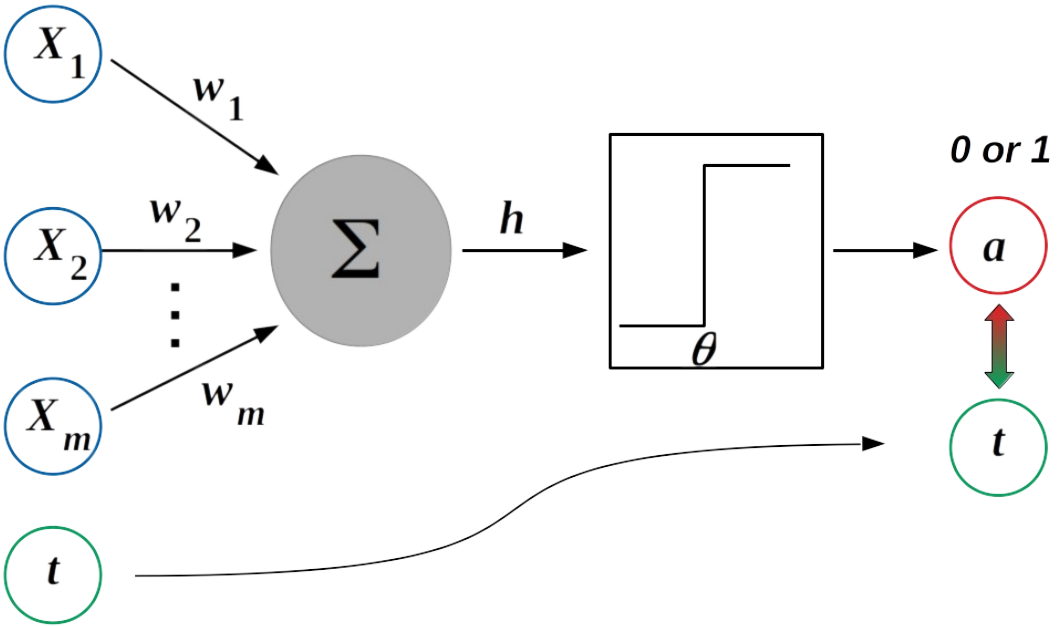


Its main components are:

- An input vector X_i that represents the dimensions of a given object
- A set of weights w_i that links the various input dimensions to the neuron
- A sum function h that defines how these weights are combined with the input dimensions
- An activation function $g(h)$ that defines if the network should remain in a “0” state or should be activated to a “1” state, depending on the results of the sum.

$$h = \sum_{i=1}^{m} X_i w_i \qquad a = g(h) = \begin{cases} 1 & \text{if} \quad h > \theta \\ 0 & \text{if} \quad h \leq \theta \end{cases}$$

Training a Neuron



$$h = \sum_{i=1}^{m} X_i w_i \qquad a = g(h) = \begin{cases} 1 & \text{if} \quad h > \theta \\ 0 & \text{if} \quad h \leq \theta \end{cases}$$

The learning process for this model is supervised
→ Each input vector is associated to a target value

So what is the purpose of the neuron if the expected value is already known ?

➡ **Generalization**

Find patterns in the data distribution in the parameter space so it can perform prediction on vectors with unseen (but close) values.

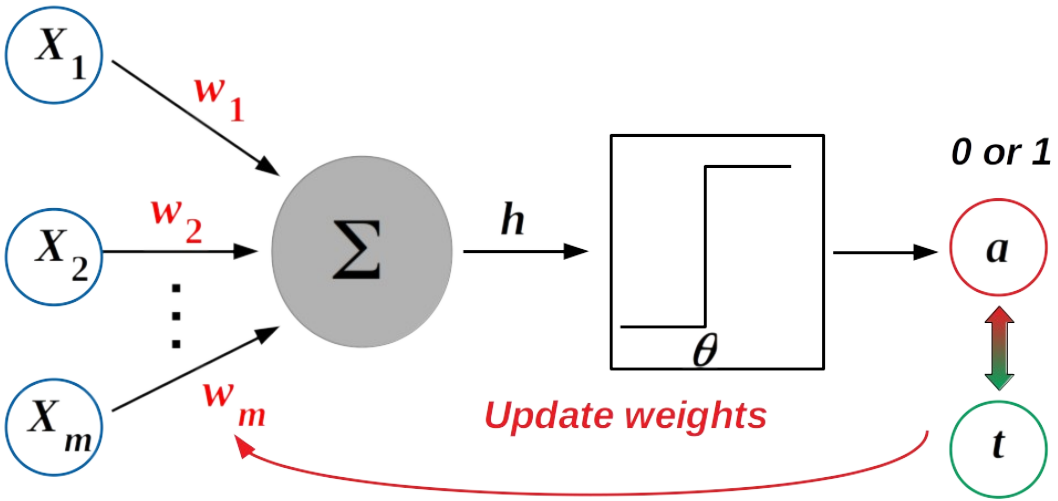
In practice, such a simple neuron optimizes a linear separation in the parameter space (**hyperplane**).

How does this model learn ?

Which parameters can or cannot be modified ?

The input, output, and targets are fixed, so the learning relies on modifying the **weights and the activation threshold**.

Training a Neuron



$$\omega_i \leftarrow \omega_i - \eta\ (a - t)\ x_i$$

$$h = \sum_{i=1}^{m} X_i w_i \qquad a = g(h) = \begin{cases} 1 & \text{if} \quad h > \theta \\ 0 & \text{if} \quad h \leq \theta \end{cases}$$

How to proceed when the output does not correspond to the target ?

There are m weights w_i associated with the network corresponding to the input dimensions.

How to modify the weights?

- If the output state is 1 while it should be 0, the weights need to be lowered
- If the output state is 0 while it should be 1, the weights need to be increased

To quantify this modification, one needs to choose an error function E.

$$E = 0.5 \times (a - t)^2$$

In the end, the weight update is defined as proportional to the input value, to the derivative of the error function for each dimension, and scaled by a learning rate* factor.

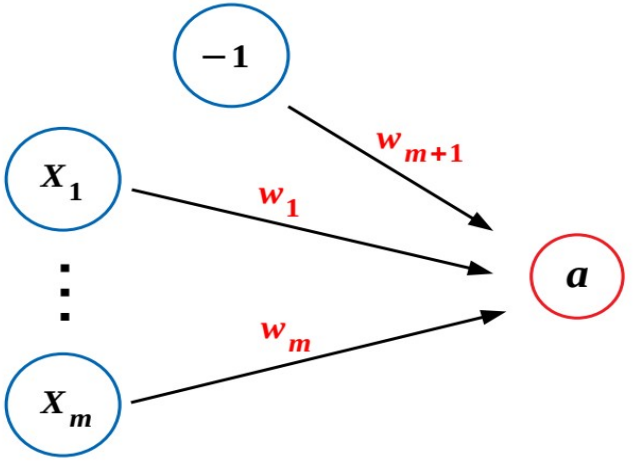
$$\omega_i \leftarrow \omega_i - \eta\ (a - t)\ x_i$$

Typically between 0.1 and 0.4, details will be given later.

The bias node

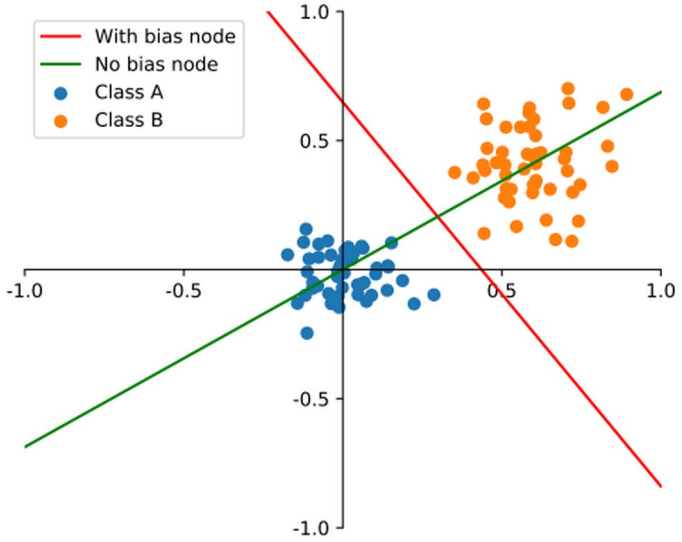
Problem with the previous formalism

The linear separation presents a fixed *f(0) = 0 point*.
The weight correction is also 0, regardless of the input value.



Solution

Add a bias input node that acts as an additional input with a constant value (usually -1). It has its own **variable weight** so it can learn the shift of the intercept position. The size of both the input vector and the weights vector is now *m+1*.



To plot the found linear separation of a neuron, we need to find where in the parameter space the neuron changes state, which corresponds to:

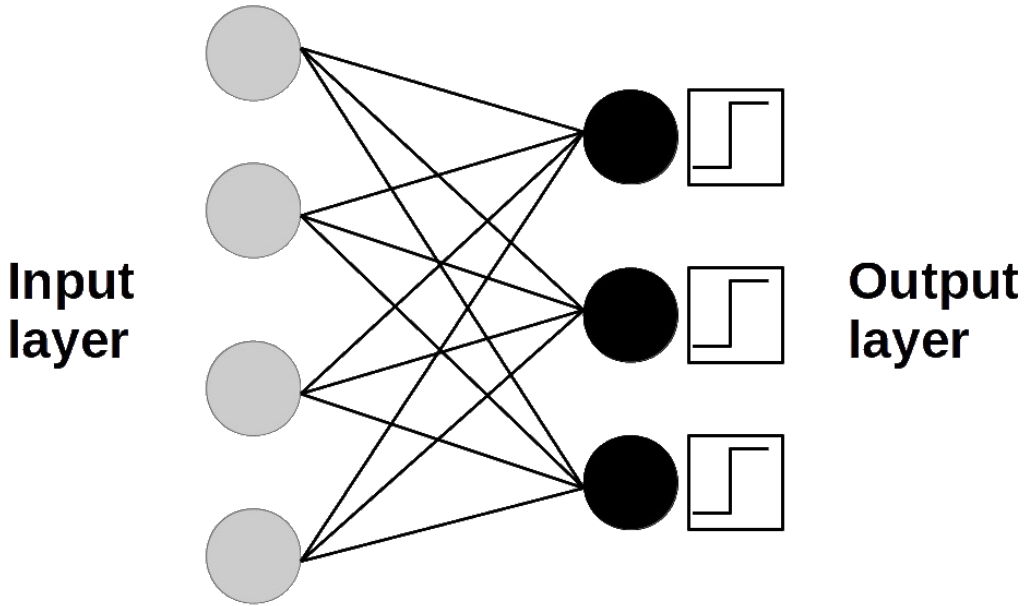
$$\sum_{i=1}^{m} X_i w_i = h = 0$$

For a 2D parameter space it results in:

$$X_1 = (W_b - X_0 W_0)/W_1$$

Note that the direction of activation is orthogonal to the linear separation.

The Perceptron algorithm



A single neuron only performs a linear separation, which is insufficient for many applications.

The simplest way to combine neurons on a single problem is to **stack them independently**. Each neuron is connected to the input vector with its own set of weights.

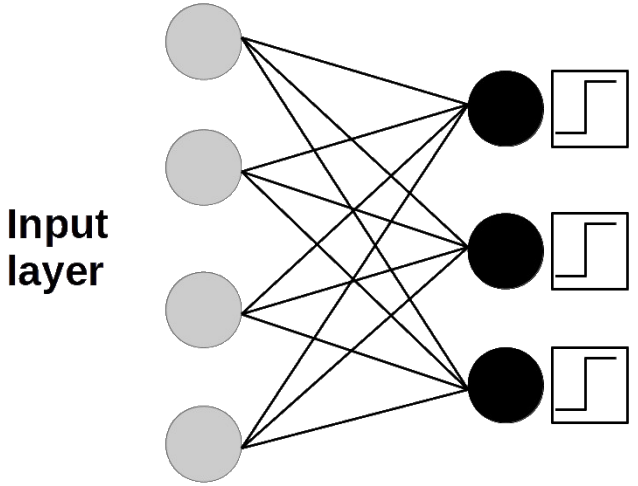
The training procedure is identical, but this time there is an index j to represent the neurons, and the weights are now in the form of a 2D matrix.

$$h_j = \sum_{i=1}^{m+1} x_i \omega_{ij} \qquad a_j = g(h_j) = \begin{cases} 1 & \text{if} \quad h_j > \theta \\ 0 & \text{if} \quad h_j \leq \theta \end{cases}$$

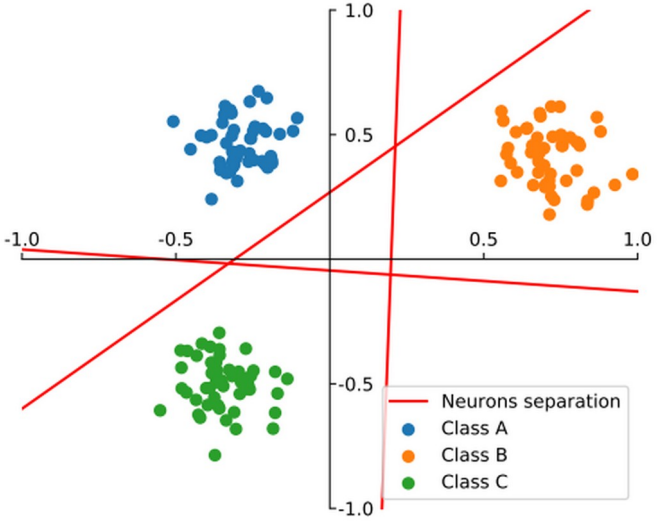
$$\omega_{ij} \leftarrow \omega_{ij} - \eta \left(a_j - t_j \right) \times x_i$$

The combination of this training procedure and this neuron connection scheme is called the single layer "Perceptron" (Rosenblatt 1958).

The Perceptron algorithm



C1	C2	C3
1	0	0
0	1	0
0	0	1



How to have multiple neurons work on the same problem ?

1) Encode the output vector (e.g in binary format)
2) Have one neuron per output class and target a format with a 1 only in the appropriate class, e.g [1,0,0]-[0,1,0]-[0,0,1]

**Warning: Since each neuron only performs a linear separation, each class must be linearly separable from all the others.*

The Perceptron algorithm

- **Initialization**
  - Set the starting weights to small random values (positive and negative). Can be drawn from a uniform or Gaussian distribution centered on zero.

- **Training**
  - For a given number of steps T, or until the output is “correct”
    - For each input vector
      - Compute the activation of each neuron *j* using the activation function *g :*

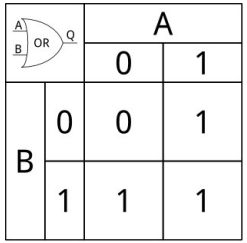
$$a_j = g\left(\sum_{i=0}^{m+1} w_{ij}x_i\right) = \begin{cases} 1 & \text{if} \quad \sum_{i=0}^{m+1} w_{ij}x_i > 0 \\ 0 & \text{if} \quad \sum_{i=0}^{m+1} w_{ij}x_i \leq 0 \end{cases}$$

      - Update each weight individually using :

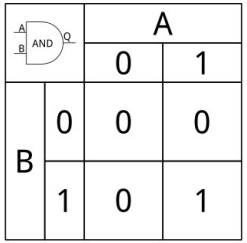
$$\omega_{ij} \leftarrow \omega_{ij} - \eta\left(a_j - t_j\right) \times x_i$$

- **Inference**
  - Compute the final activation of each neuron *j* for each input vector to test

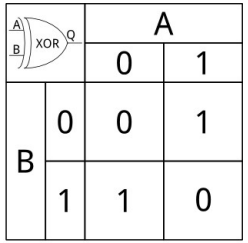
First application : logical gates



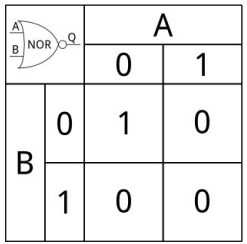
B \ A	0	1
0	0	1
1	1	1



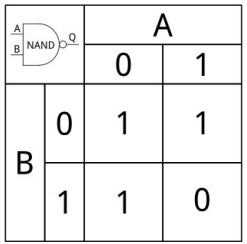
B \ A	0	1
0	0	0
1	0	1



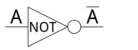
B \ A	0	1
0	0	1
1	1	0



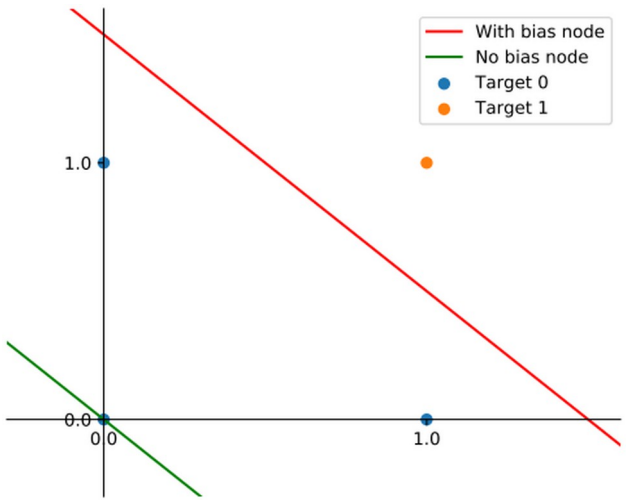
B \ A	0	1
0	1	0
1	0	0



B \ A	0	1
0	1	1
1	1	0



A	0	1
	1	0



A straightforward application of this algorithm is to make it learn a logical door, here the OR or AND gate. For this simple application, the Perceptron has a two-dimensional input vector and 4 possibilities as input-target pairs.

The output is made of a single binary neuron to predict either the 0 or 1 state of the gate.

As a first exercise:

- Write a simple program that declares an array with all the possible inputs and the associated targets.
- Identify all the necessary variables and initialize the weights with small values.
- Try to train this neuron over a few iterations following the Perceptron algorithm.

Do not forget the bias node!

- Display the output at each iteration to watch the network converge toward a stable solution.

Why is the **XOR gate** problematic with this algorithm?
How to solve the problem?

The Pima Indian dataset

Dataset from the UCI Machine Learning repository.

Provides 8 physiological measurements for 768 female native Americans and provides a class depending on whether the individual has developed diabetes.

This exercise aims to implement a Perceptron to **predict if a person has diabetes** based solely on the 8 input measurements.

Despite its limitations, a simple Perceptron can reach up to **70% accuracy** on this dataset.

- Modify your program to read the Pima dataset.
- Adapt the input dimension (including bias).
- Adapt the corresponding computations.
- Find a method to compute the "accuracy" and its evolution during training.
- Question the limits of your accuracy definition, and find a way to ensure that you are testing the generalization capabilities of the method.

Attribute Number	Attribute
1	Patient age
2	Body mass index (kg/m2)
3	Concentration of plasma glucose
4	2-h serum insulin (mu U/mL)
5	Thickness of triceps skin-fold (mm)
6	Pedigree function of diabetes
7	Number of times patient pregnant
8	Diastolic blood pressure (mmHg)
9	Class 0 or 1

Input dimension : 8 (+1)

Output dimension : 1
(or 2 if using one neuron per class)

Number of example : 768

Class distribution : 500 class 0; and 268 class 1

The Iris dataset

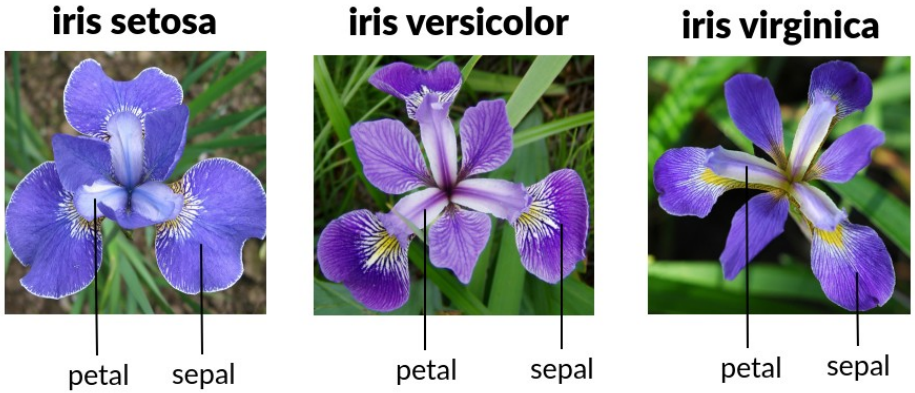
Dataset from the UCI Machine Learning repository.

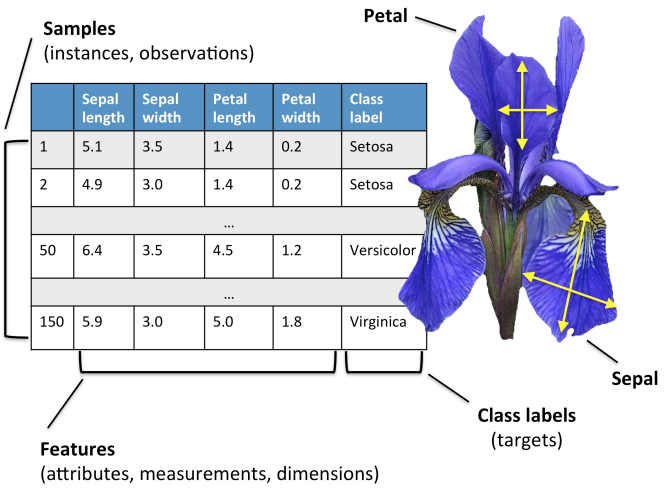
Provides 4 sizes for a set of 150 Iris flowers and provides membership to one of 3 classes.

This exercise aims to implement a Perceptron to **predict the corresponding class** based on the 4 input measurements for each flower.

Despite its limitations, a simple Perceptron can reach up to **80% accuracy** on this dataset.

- Modify your program to read the Pima dataset.
- Adapt the input dimension (including bias).
- Adapt the corresponding computations.
- Find a method to compute the "accuracy" and its evolution during the training phase.
- Question the limits of your accuracy definition, and find a way to ensure that you are testing the generalization capabilities of the method.





	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

Input dimension : 4 (+1)
Output dimension : 3
Number of example : 150
Class distribution : 50 examples per class

The Spectra dataset

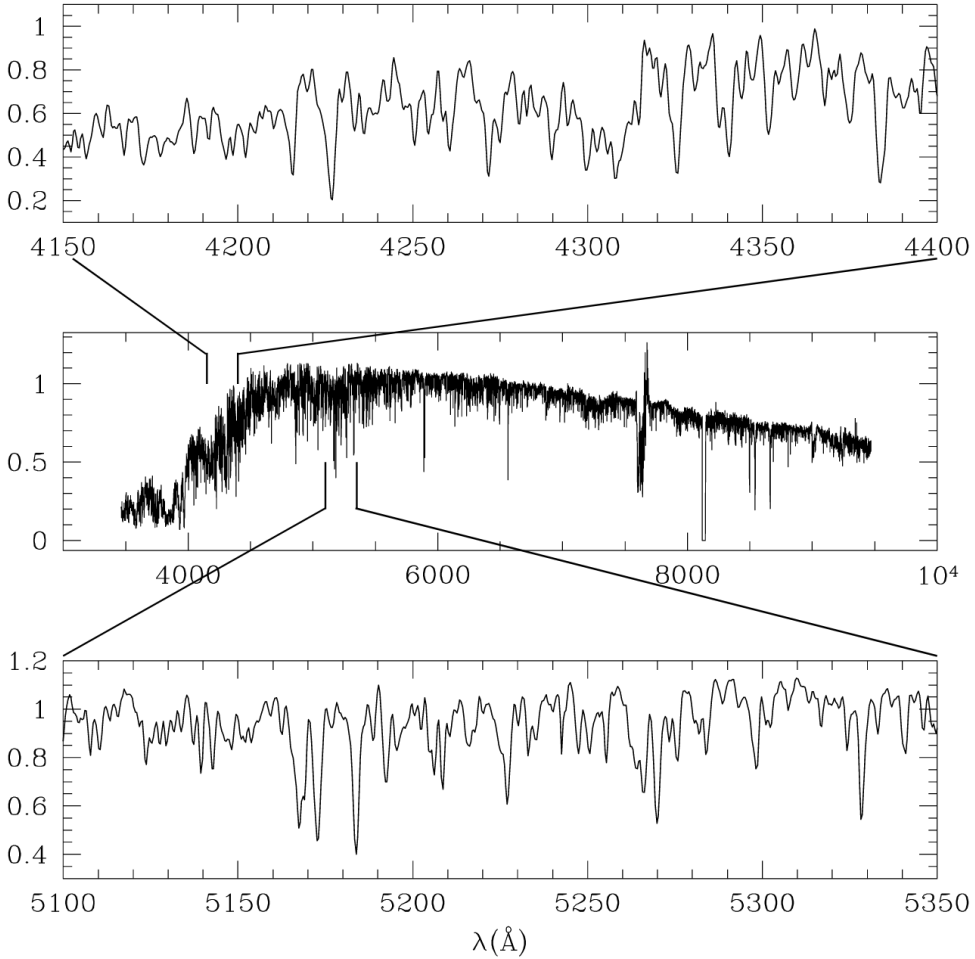
This dataset contains stellar spectra obtained with the 0.9m Coudé Feed telescope at Kitt Peak National Observatory, classified into various spectral types.

The provided dataset here is a simplification that only keeps 1115 homogeneous spectra with half the resolution (3753 “pixels” remains).

The target only provides the 7 regular spectral types for classification to increase the number of examples per class. Still, the dataset remains highly imbalanced!

Infos about the dataset and the reference paper (*F. Valdes et al. 2004*) are in the info file.

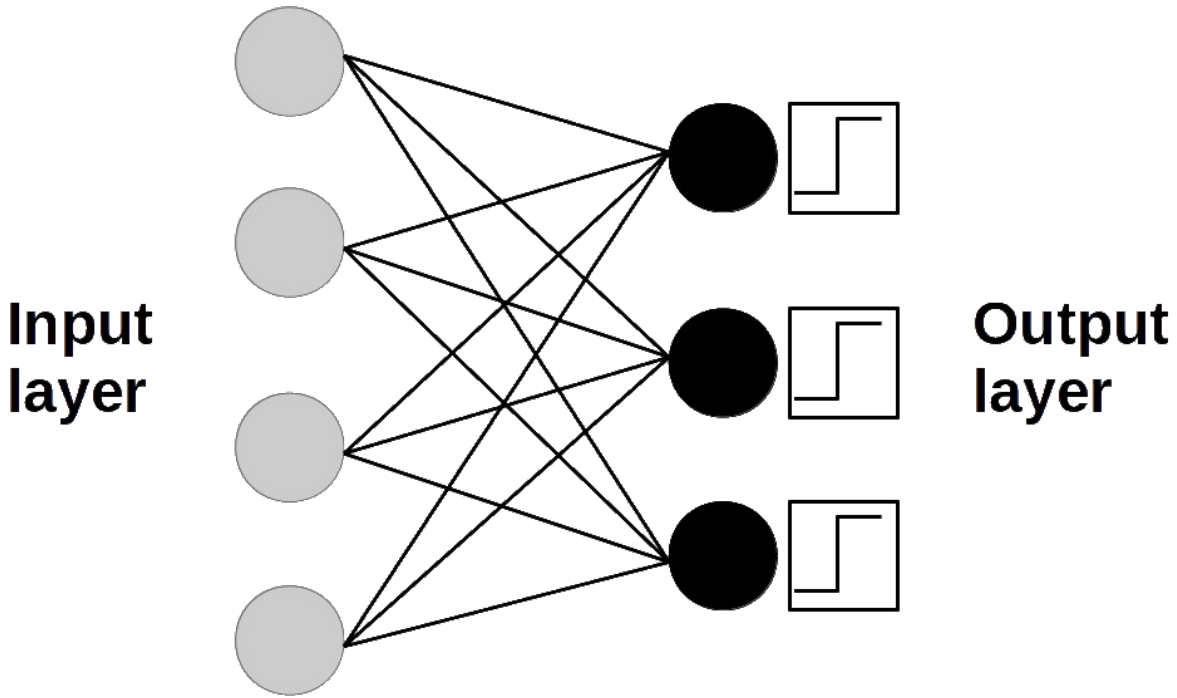
Input dimension : 3753 (+1)
Output dimension : 7
Number of example : 1115
Class distribution : strong imbalance



Constructing a “Deep” Neural Network

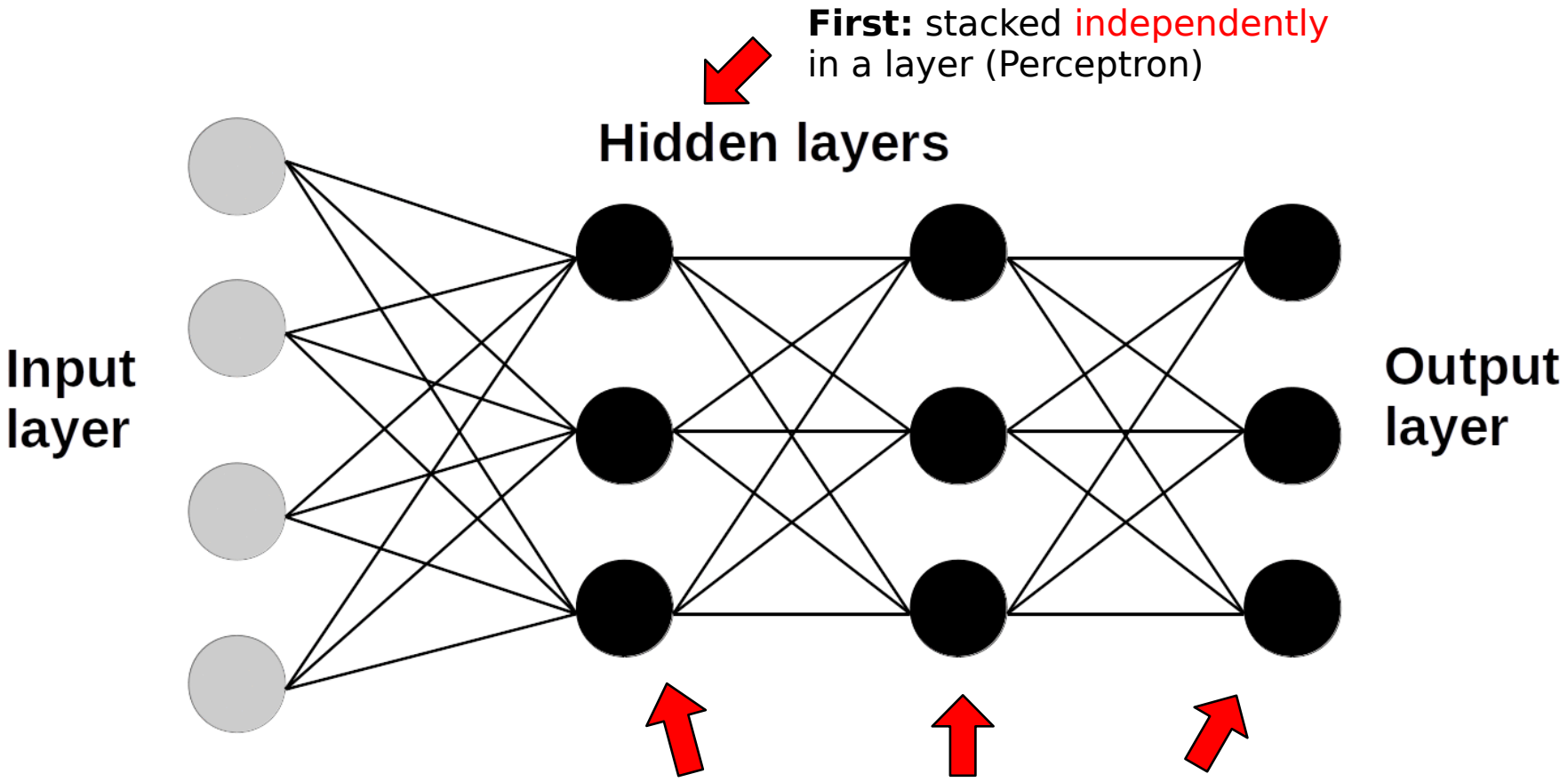
For more complex problems more neurons must be used.

First: stacked independently in a layer (Perceptron)



Constructing a “Deep” Neural Network

For more complex problems more neurons must be used.



Constructing a “Deep” Neural Network

In addition to the MLP architecture, it is necessary to change the activation to allow non-linear combinations

➡ **Change for the Sigmoid (logistic) function**

$$g(h) = \frac{1}{1 + \exp(-\beta h)}$$

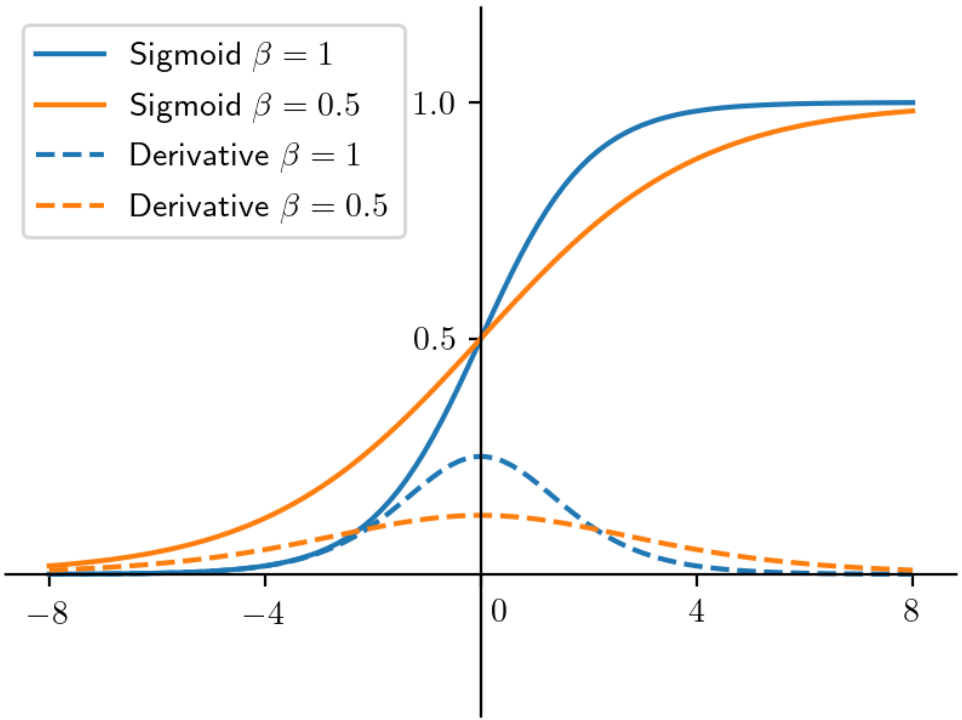
Beta represents the slope of the function.

The sigmoïd is continuous while preserving the desired global binary behavior in its extremity.

It has a nice derivative form :

$$\frac{\partial a_j}{\partial h_j} = \beta a_j (1 - a_j)$$

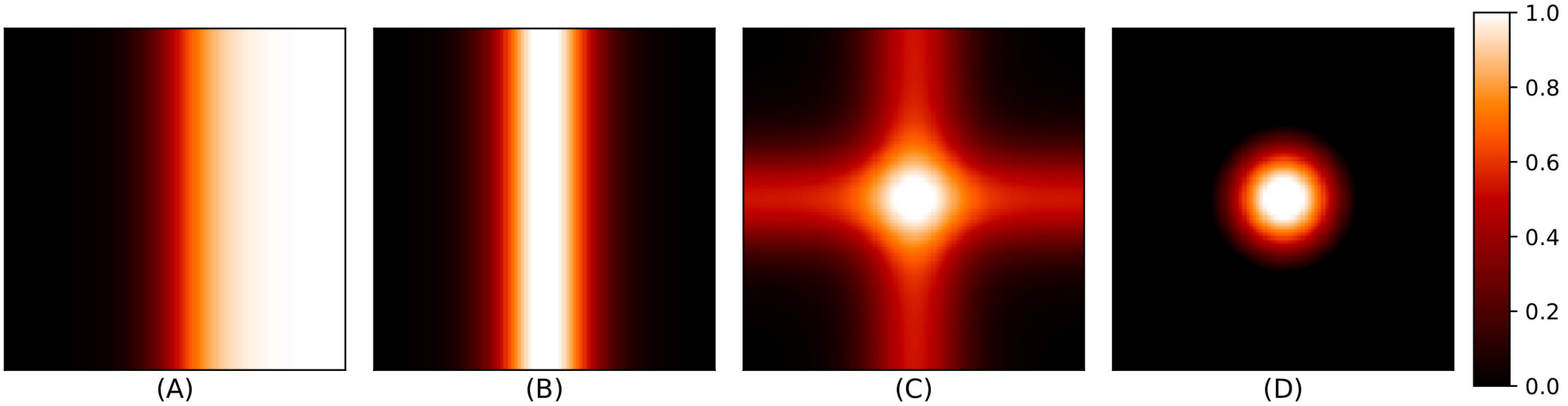
For regression problems, the output layer can either use sigmoïds or simple linear activation functions since the previous layers will handle non-linearity.



Constructing a “Deep” Neural Network

Stacking layers that use a sigmoid activation allows the construction of more and more complex functions. The image below demonstrates that it is possible to reconstruct hill shapes (B), bumps (C), and point-like functions.

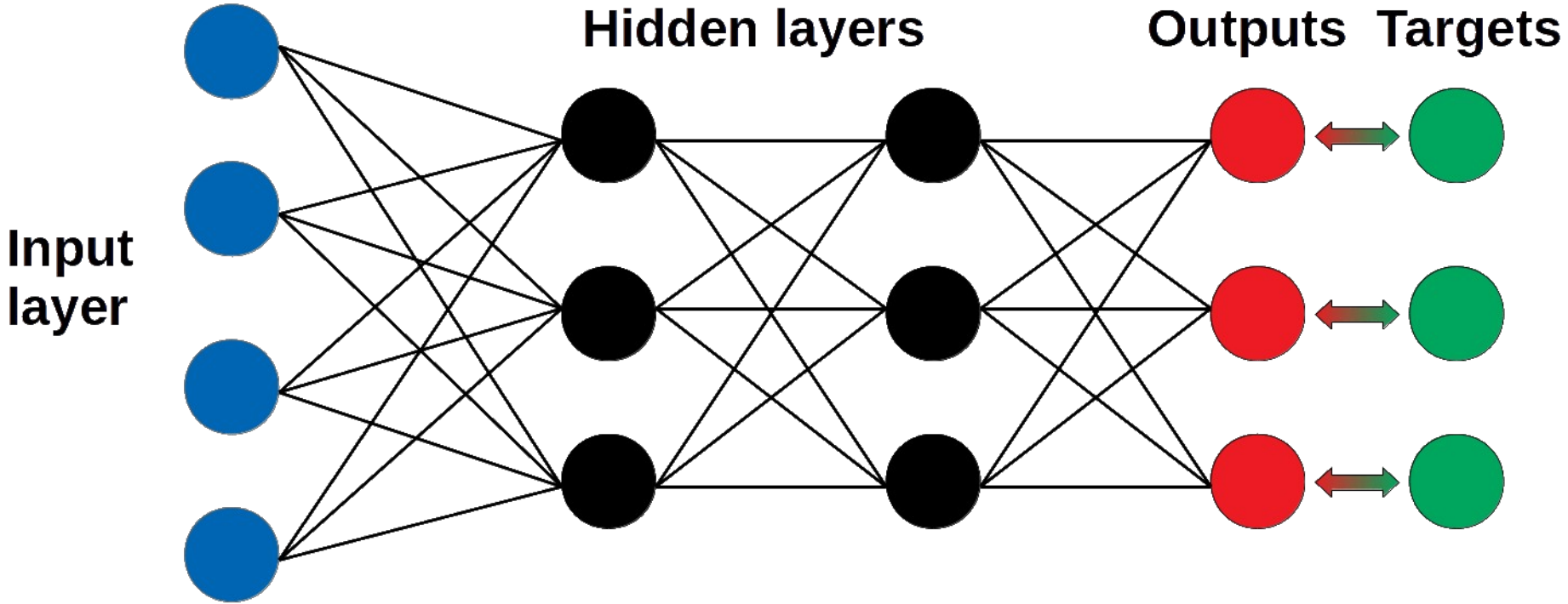
➡ With these improvements, the MLP is a **Universal Function Approximator**



In practice, based on the universal approximation theorem, a MLP with a single hidden layer remains a UFA at the condition of having a large enough number of neurons.

Error gradient backpropagation

Problem : how to correct the output of neurons in the hidden layers ?



$$\omega_{ij} \leftarrow \omega_{ij} - \eta \frac{\partial E}{\partial \omega_{ij}} \quad \Rightarrow \quad \frac{\partial E}{\partial \omega_{ij}} = \delta_l(j) \frac{\partial h_j}{\partial \omega_{ij}} \quad \text{with} \quad \delta_l(j) \equiv \frac{\partial E}{\partial h_j} = \frac{\partial E}{\partial a_j} \frac{\partial a_j}{\partial h_j} = \frac{\partial a_j}{\partial h_j} \sum_k \omega_{kj} \delta_{l+1}(k)$$

Actual Network equations for an MLP with a single hidden layer

Changes in the Perceptron algorithm required to obtain the Multi-Layer Perceptron algorithm :

- Change the activation function of all neurons, with $\beta \geq 1$

$$g(h) = \frac{1}{1 + \exp(-\beta h)}$$

- Add a hidden layer with its own bias node (connect input → hidden, and hidden → output)
- Implement the "back-propagation" with $E(a^o, t) = \frac{1}{2} \sum_{k=1}^{N} (a_k^o - t_k)^2$ using the following equations:

$$\delta_o(k) = \beta a_k^o (1 - a_k^o)(a_k^o - t_k)$$

$$\delta_h(j) = \beta a_j^h (1 - a_j^h) \sum_{k=1}^{o} \delta_o(k) \omega_{jk}$$

$$\omega_{jk} \leftarrow \omega_{jk} - \eta \delta_o(k) a_j^h$$

$$v_{ij} \leftarrow v_{ij} - \eta \delta_h(j) x_i$$

Example with classification

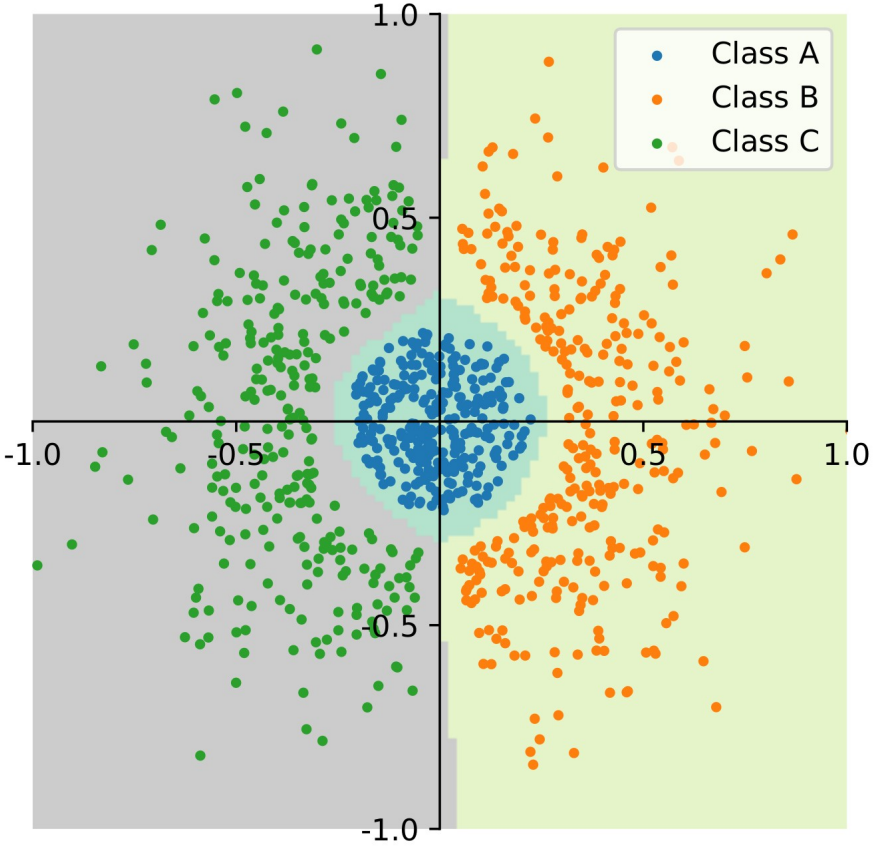


Illustration of a three-class separation in a two-dimensional feature space using a trained MLP with 3 output neurons and 8 hidden neurons, all with sigmoid activations. The light background colors indicate the regions of the feature space that the network has attributed to each class.

How to measure classification performance ?

Confusion Matrix

- Uses "**observational proportions**"
- Provides confusion between classes
- Is much more precise than the average accuracy

$$\text{Recall} = \frac{TP}{TP + FN}$$

Predicted

Actual	Positive	Negative	Recall
Positive	93	7	93.0%
Negative	46	954	95.4%
Precision	66.9%	99.27%	95.18%

$$\text{Precision} = \frac{TP}{TP + FP}$$

$TP \equiv$ True Positive $\quad TN \equiv$ True Negative

$FP \equiv$ False Positive $\quad FN \equiv$ False Negative

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

ML methods usually learn better on balanced dataset **BUT**

the results must always be represented using "observational proportions", with imbalance.

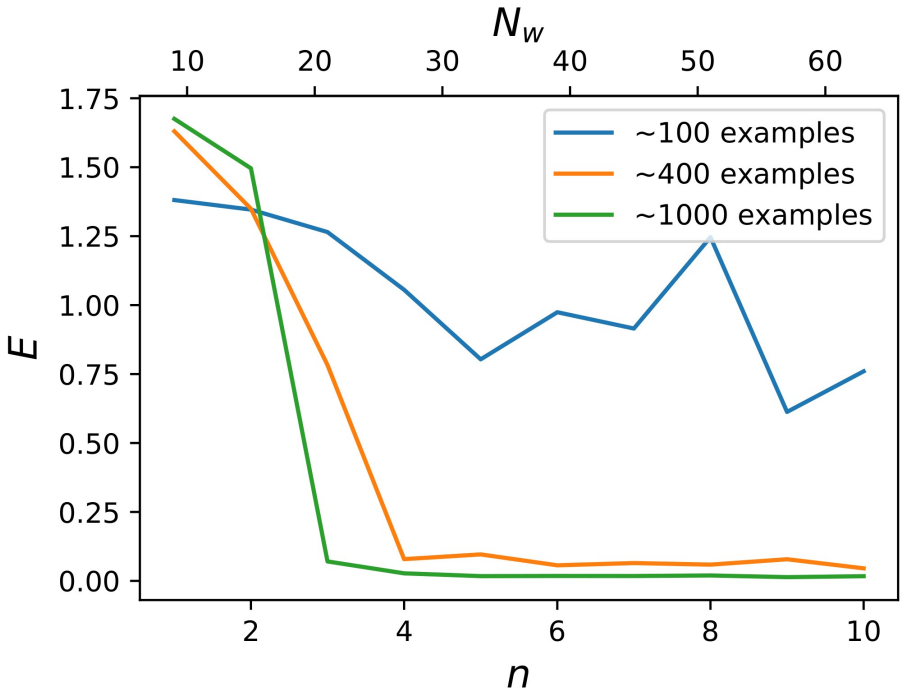
Network depth and size

How to choose the network architecture?
In the case of the MLP, how many layers and neurons should be used per layer?

In this context, the objective is to find a **trade-off between:**
The strength of the network (its expressive power) and the difficulty of training the model

A few ideas that help explore the architectures:

- More data allows to constrain more parameters
  Rule of thumb: need roughly 10 times the number of weights as training examples
- More neurons increase the strength but add parameters to fit.
  Reminder: a neuron is a linear separation in the parameter space → how many would you need by eye?
  (starting point before exploration)
- More layers allow the construction of complex functions with fewer parameters, but increases training difficulty.
  Reminder: a single hidden layer can approximate any function, but it is not always the optimal solution
  (nb. of parameters, time to converge, etc.)



Finding the appropriate architecture is always an exploratory and iterative process.
Training different networks on the same data is almost always necessary to obtain good results.

Learning rate

Choosing the appropriate learning rate for the task is very important and must consider several aspects of the error space.

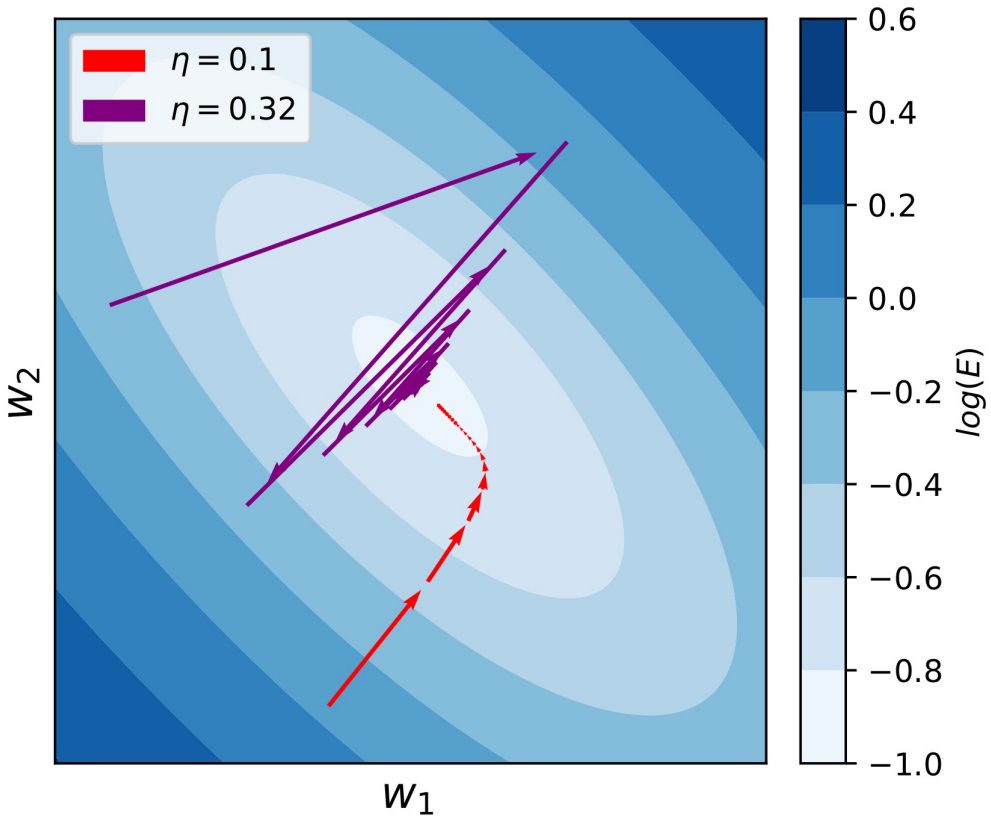
	Small η	**Large η**
Pros	• Increase training stability • Can find narrower minima	• Faster training • Easily switch from a local minima to another
Cons	• Slow training • Might get stuck on local minima	• Might fail to find a narrow minima • Training can be very unstable

The learning rate does not have to be a single value, it can change during the training phase.

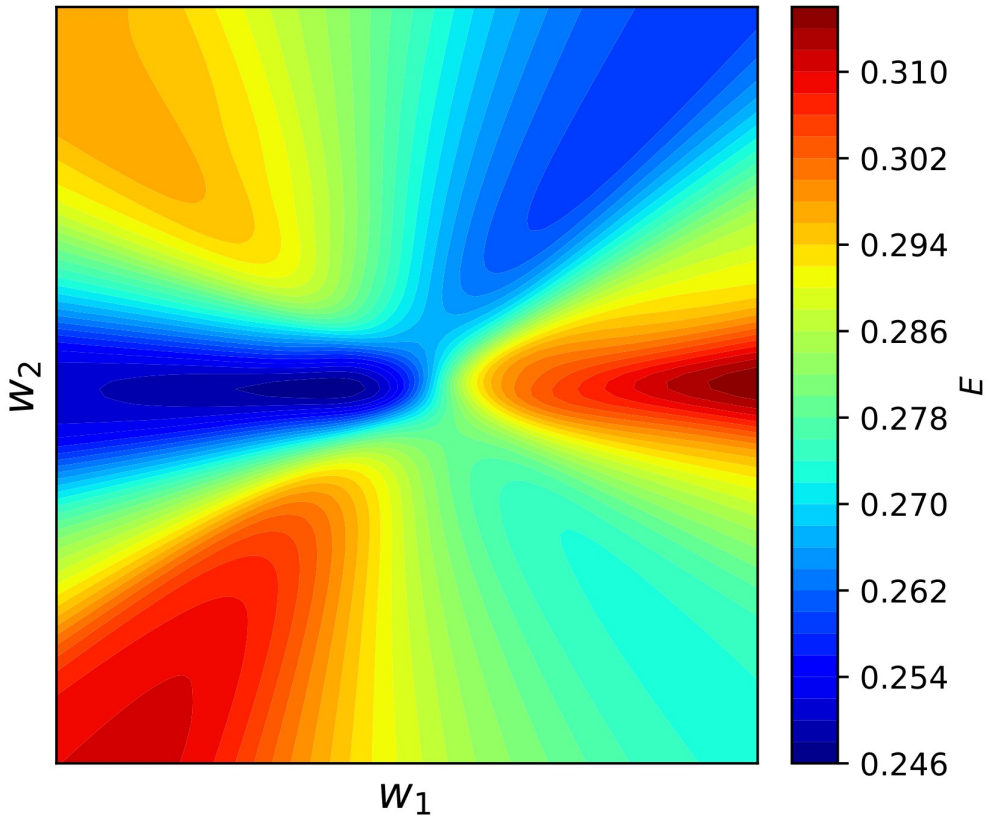
The simple approaches are: pre-established fixed values, linear decay, exponential decay, ...

Most frameworks allow the use of advanced gradient descent scheme that also automatically adapts the learning rate: Adam, RMSprop, Adagrad, etc ...

Learning rate



Logarithm of the error term in the weight space of a simple binary neuron with a two dimensional input parameter. The arrows represent the correction after each epoch for two different learning rates.



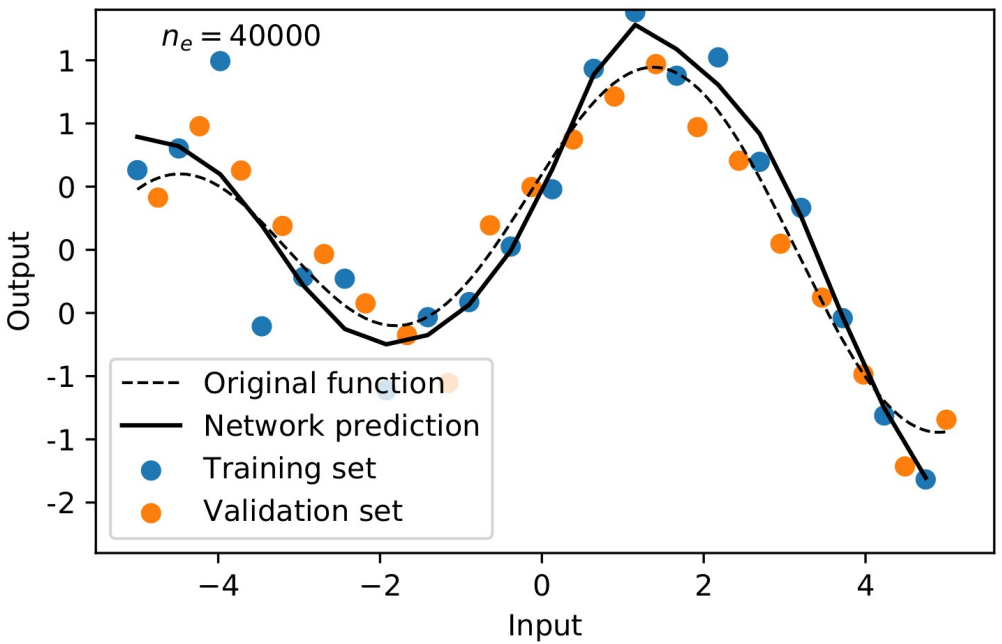
Error term in a 2D weight space of neuron in the hidden layer of a 3 class output MLP. The other weights of the network are frozen while exploring this 2D weight space.

Overtraining

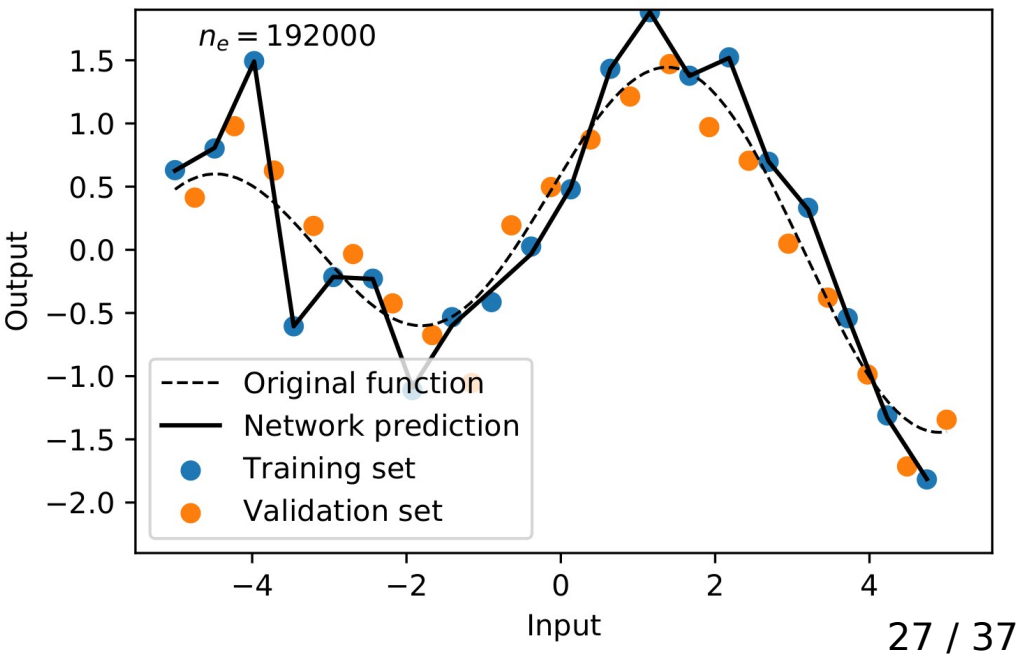
Overtraining: At some point during the training phase, the network that looks at the same data again and again, might start to over-fit!

It means that the network is learning specificities of the dataset that are no longer related to the average function that it should approximate.

Proper fit



Over-fitting



Monitoring overtraining

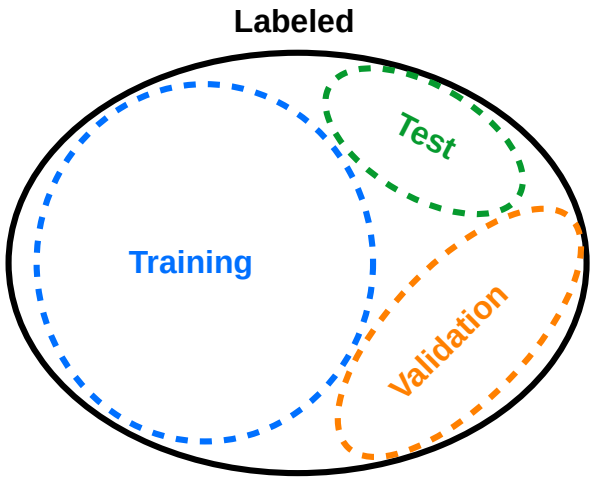
While several techniques can be used to moderate overtraining (like regularization), it is always necessary to monitor it to stop the training at the appropriate time.

Split **labeled dataset** into 3 subsets

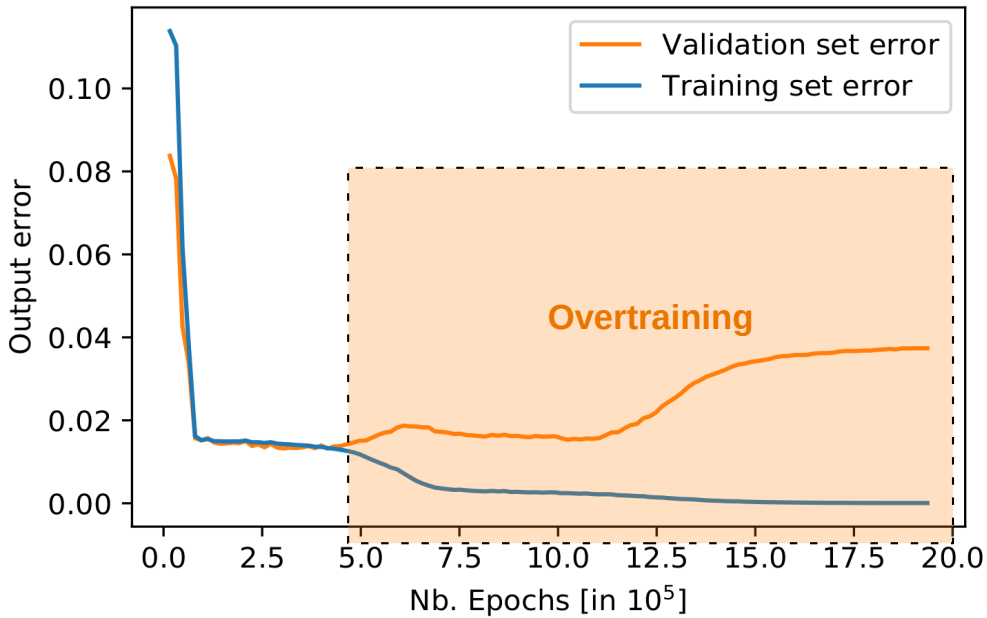
Training set ➤ To train the network

Validation set ➤ To monitor the training process

Test set ➤ To assess the quality of the prediction



Typical proportions



Improving results with data pre-processing

Improving the prediction results relies on more than just using larger networks.
Data pre-processing can be employed with two primary objectives :

Add already known information, remove irrelevant information

The way we organize and process the data can help statistical methods.

For example, there is an “age” parameter in the Pima dataset. While it is well known that most medical issues correlate to aging, a one-year resolution is unlikely to reflect the effect. Using groups based on the age parameter, e.g. [Under 30, 30-40, 40-50, 50+], tends to improve the result.

Another example is the “number of times pregnant”. While there is certainly a significant difference between 0 and 1, it is unlikely that 3+ makes more difference. So it should be possible to cap this value.

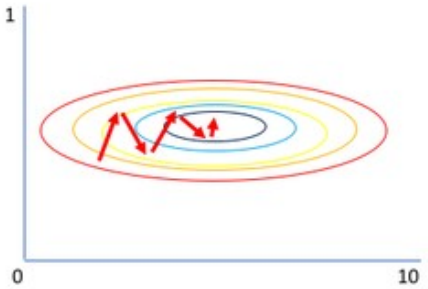
This type of pre-processing can have a significant impact on prediction performance for large models.

Normalization

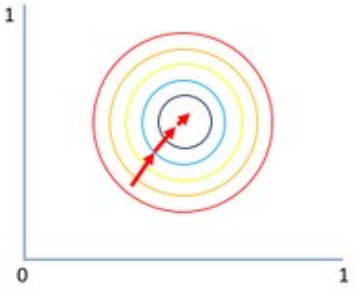
Why normalize ?

It is always necessary to **normalize each dimension of the dataset,** usually in the range [0,1] or [-1,1].

Otherwise, some dimensions might dominate the weight update gradient, eclipsing the information contained in the others



Gradient of larger parameter dominates the update



Both parameters can be updated in equal proportions

Different approaches to rescaling

Centered: $X - \text{mean}$

Standardized: $\frac{X - \text{mean}}{\text{sd}}$

Normalized: $\frac{X - \min(X)}{\max(X) - \min(X)}$

WARNING !

All labeled data must be normalized with the same values (not only with the same equation)!

This include both the training and testing samples but also the data used for prediction at deployment time!

Improving classification

While it is possible to perform classification in a binary way using neural networks, it would be much more helpful to have "probability" predictions.

It is possible to associate a floating point value representing a class membership likelihood using the "one-per-class" encoding and sigmoid activations. A probability can then be obtained by normalizing the output vector by the sum of all activations.

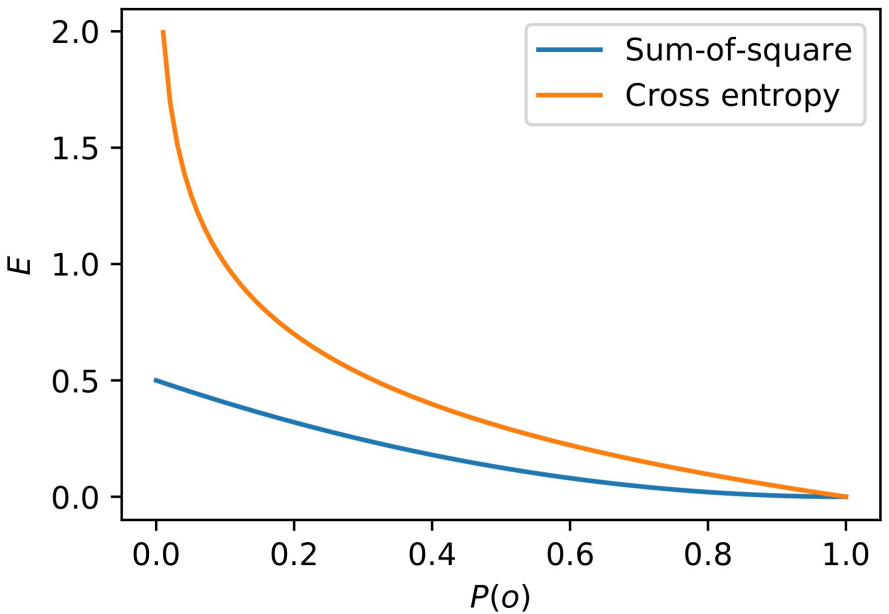
An even better way to construct efficient membership probability is to use the Softmax activation function paired with a **cross-entropy error function**.

$$a_k^o = g(h_k^o) = \frac{\exp(h_k^o)}{\sum_{k'=1}^{o} \exp(h_{k'}^o)}$$

$$E = -\sum_{k=1}^{o} t_k \log(a_k^o)$$

Using these functions, one can demonstrate that the output error terms can be simplified as

$$\delta_o(k) = a_k^o - t_k$$



Gradient Descent Schemes

The previous algorithms were presented in their sequential (or online) version, where one item is forwarded into the network at a time, and an update is performed before going to the next one. Ordering effects can degrade generalization capabilities, hence the shuffling step at each epoch.

But there are other possible Gradient Descent Schemes :

Batch

All the objects from the training set are forwarded, and their contributions to the error gradient are averaged.

One weight update is then performed before going to the next epoch.

Mini-Batch SGD

The train set is randomly split into groups of identical size.

Each of these groups is then used to train in a batch way, averaging the contribution of each element.

All the data are randomly distributed to the groups at each epoch.

SGD

Stochastic Gradient Descent.

The examples are selected randomly in the full training set (with replacement).

A weight update is performed for each object.

This method replaces the shuffle at each epoch.

Modern Gradient descent schemes optimizers are all based on these different approaches (Adam, RMSprop, Adagrad, etc ...)

Momentum

Another simple way of improving the Neural Network behavior during the training phase is to use previous updates as an indication of the global direction of the error gradient.

This corresponds to the idea of inertia or momentum.

Each weight update can then incorporate a term corresponding to the previous update term, which will progressively average the few last updates.

$$\omega_{ij}^{t} \leftarrow \omega_{ij}^{t-1} - \Delta\omega_{ij}^{t}$$

$$\Delta\omega_{ij}^{t} = \eta \frac{\partial E}{\partial \omega_{ij}^{t-1}} + \alpha \Delta\omega_{ij}^{t-1}$$

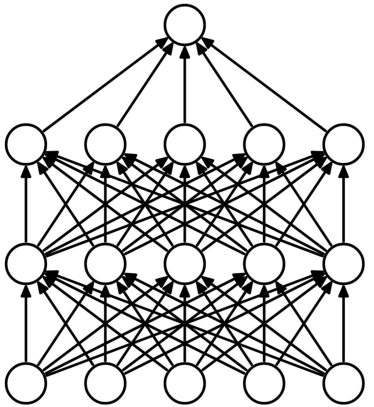
With α the momentum term, which is usually between 0.6 and 0.9.

This inertia in the gradient value will help the network to get out of local minima. It will overall speed up the training by emulating a larger value of the learning rate toward the global gradient direction during the first few epochs, while having less impact when the network oscillates around its optimal value in the last epochs.

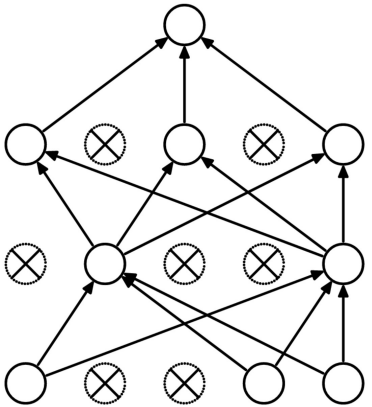
Additional regularization: dropout

One common regularization techniques is called **dropout**, which consists in randomly deactivating some neurons for each batch during the training phase.

This approach usually allow better results through different effects on the network's inner working (more salient features, increased sparsity, etc.).



(a) Standard Neural Net



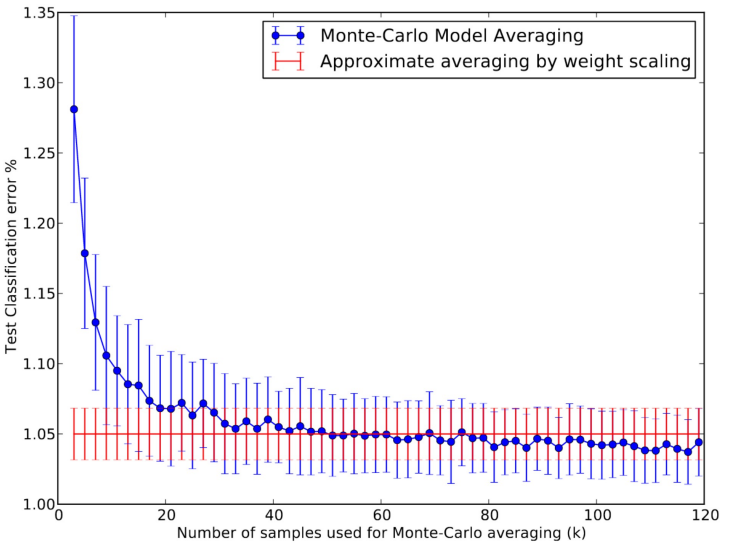
(b) After applying dropout.

Average dropout

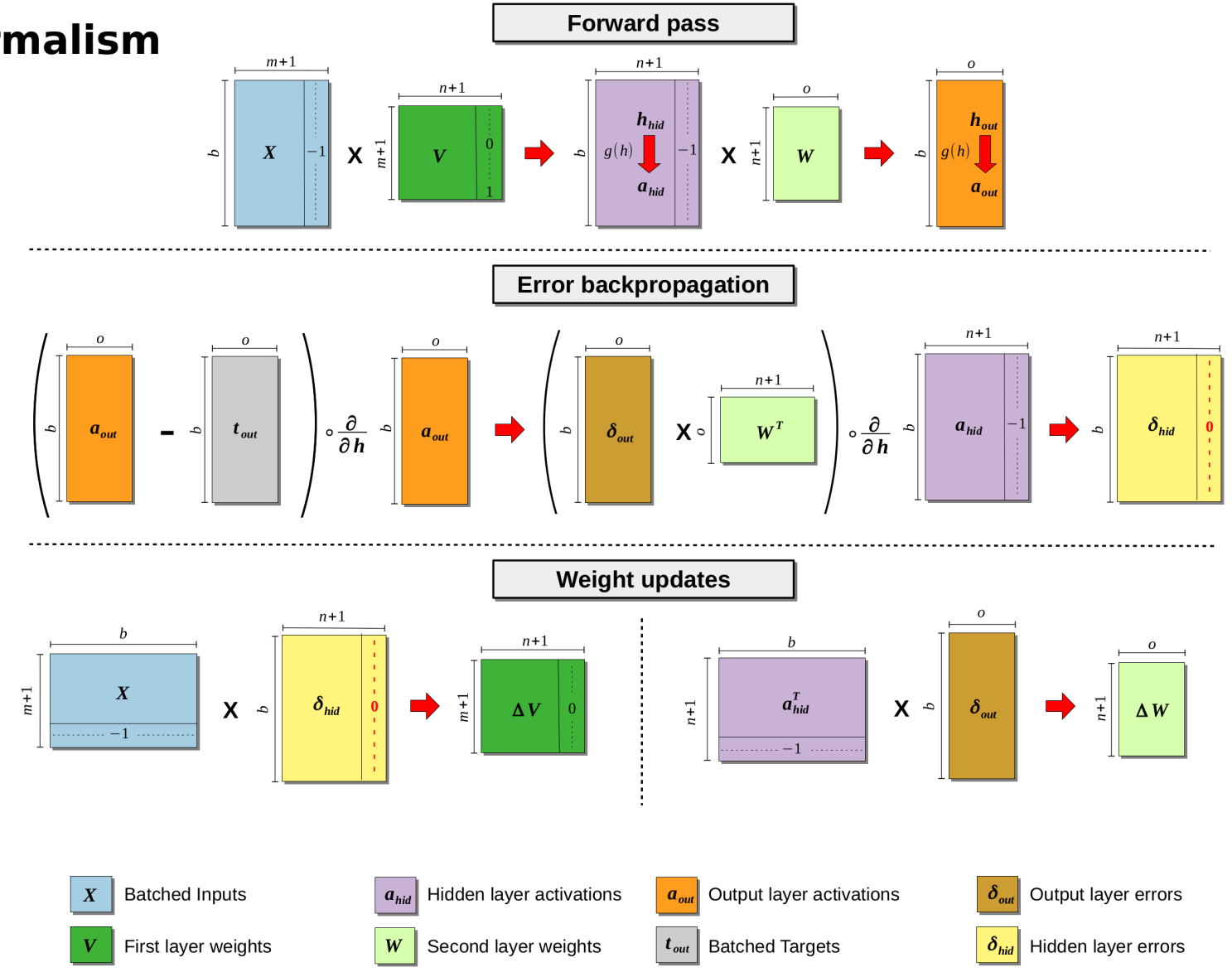
At inference time it, is possible to **average all the models** represented by the dropout selection by keeping all neurons activated but with a scaling of the weights proportional to the dropout rate at each layer.

Monte-Carlo dropout

Another approach is to keep dropping neurons but performing several predictions for each input to test.
This is a way to reconstruct a **posterior probability distribution** for each output dimension.
This approach allows to measure the network **prediction uncertainty**.



Matrix formalism



Optimized matrix operation

$$C(i, j) = \sum_k A(i, k) \times B(k, j)$$

Matrix operations are of uttermost importance

→ They appears in a lot a computing problems!

→ They represent the vast majority of the computations done by AI models.

Matrix operations have received a lot of **optimization attention** and chips manufacturer have built **dedicate hardware and instructions** for this operation!

The default algorithm **complexity scales with the cube of the matrix side size**
→ if M=N=K, it scales in $\Theta(N^3)$.

Algorithms with a lower complexity exist but they often work in a way that make them difficult to optimize for classical computing hardware.

Many libraries are dedicated to **parallel matrix multiplication** or other linear algebra operation **(BLAS) acceleration** using various types of hardware:

OpenBLAS, IntelMKL, MAGMA, CuBLAS, rocBLAS, ...

MLP application

Use the previous list of changes to implement the Multi-Layer Perceptron algorithm.

Use the previous datasets to test your algorithm, and try to optimize its prediction performance by tweaking the dataset and the accessible network hyperparameters.

Your implementation must have the following properties:

- Easy change of the learning rate, number of epochs, number of neurons in the hidden layer
- An output encoding using one neuron per class
- A display of the results in the form of a confusion matrix

In addition, try to have:

- Generic functions for the forward and backprop operations that are independent of the dataset
- A way to re-balance the proportions of classes for the training phase (bet test on true proportions)

Advanced:

- Splitting of the labeled data to avoid overtraining
- A probabilistic prediction (softmax or other)
- An alternative gradient descent scheme (either SGD or Batch)
- A matrix formalism version of the forward and backprop operations