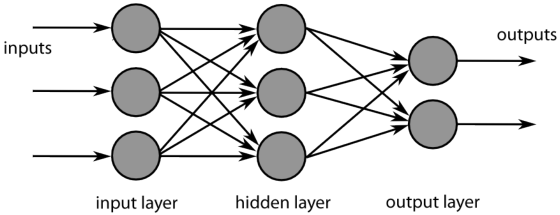
Neural Networks

Alvaro Soto

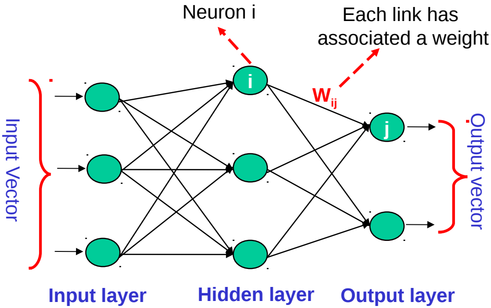
Computer Science Department (DCC), PUC

Neural Networks

- Highly practical and general approach to model data.
- There are variants that can be used for supervised and unsupervised machine learning problems.
- Here, we will focus on supervised neural network techniques.
- In particular, we will focus on feed-forward models.



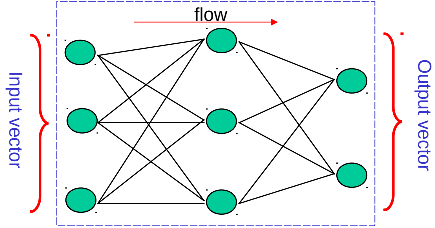
Neural Networks: Structure



Main components of a NN are:

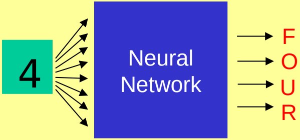
- Structure: number of units in input, hidden, and output layers. Also, number of hidden layers.
- Neuron type: mainly activation function used to regulate the output of each neuron.
- Learning method: technique used to adjust the internal weights (w_{ij}) of the network.

Neural Networks: Learning



Goal: given the NN structure, we need to adjust the weights in the network, such that they model the input-output relation in the training data.

Example:
when the NN receives an image of the digit 4, it should output a code that indicates 4.

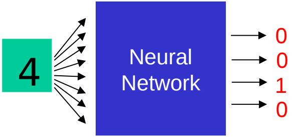


How to adjust the number of neurons in each layer?

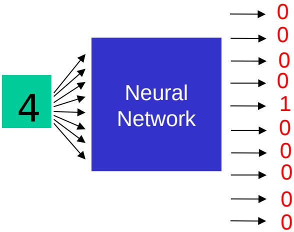
Input layer: in general, the number of units in the input layer is equal to the number of input features.

Hidden layer: the number of units in the hidden layers (and the number of hidden layers), depend on the complexity of the problem. More difficult classification or prediction problems require more complex hypothesis spaces.

Output layer: the number of units in the output layer depends on the codification selected. As an example, consider the following cases:



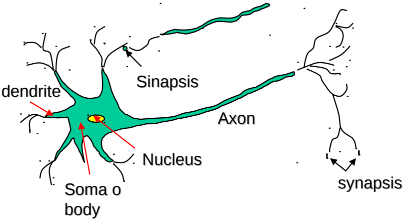
Binary codification



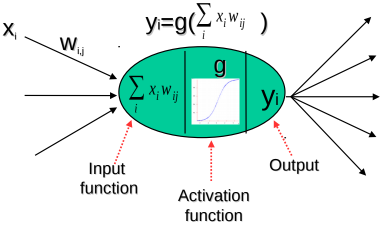
One-hot codification

Biological versus artificial neurons

Biological NN



Artificial NN

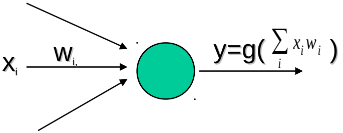


Relevant history

- 1943: McCulloch and Pitts. Binary threshold neurons.
- 1949: Hebb. Hebb learning rule.
- 1959: Rosemblatt. Perceptron.
- 1969: Minsky. Perceptrons are highly limited.
- 1982: Hopfield. Hopfield nets.
- 1986: Rumelhart. Backpropagation.
- 2000s: Hinton, Lecun. Deep convolutional networks.
- Today: Deep Learning.

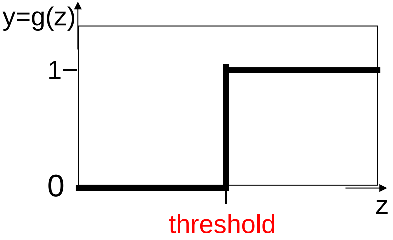
McCulloch and Pitts: Binary Threshold-neuron (1943)

- First NN model.
- Used to model logical functions.
- Weight values are manually selected. There is not a learning algorithm.

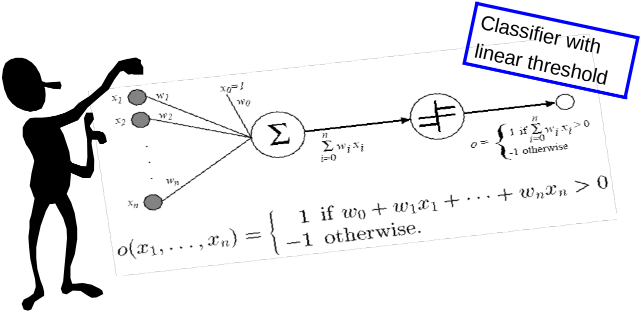


$$z=\sum_i x_i w_i$$

$$y=\begin{cases} 1 \text{ if } & z \geq \theta \\ 0 \text{ otherwise} & \end{cases}$$



Perceptron (1959)


x_1
w_1
x_2
w_2
x_n
w_n
$x_0=1$
w_0
Σ
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$
$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$

Classifier with linear threshold

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

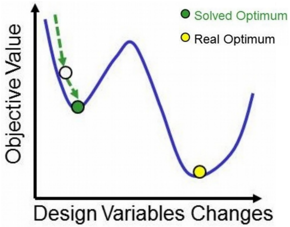
Discover of the magic:
Artificial neuronal units + **Learning Algorithm**

Training a perceptron

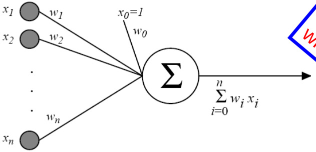
- We want to minimize the difference between real and predicted outputs over the training set.
- Loss function:

$$E[\vec{w}] = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

- We will solve this optimization problem using an iterative gradient based solution.



Training a perceptron



Linear classifier without threshold

Goal:
Find weights (hypothesis) that minimize the square error over the classification of examples from the training set.

Training set: (x_i,t_i)

x_i: input features.

t_i: target output.

o_i: predicted output.

$$O=w_0+w_1x_1+\dots w_nx_n=\sum_{i=0}^{n} w_i x_i$$

$$Error=E[\vec{w}]\equiv\frac{1}{2}\sum_{d\in D}(t_d-o_d)^2$$

Training a perceptron

Learning Rule:
Move weights in the direction
against the gradient (gradient descent)

$$w_i \leftarrow w_i + \Delta w_i$$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

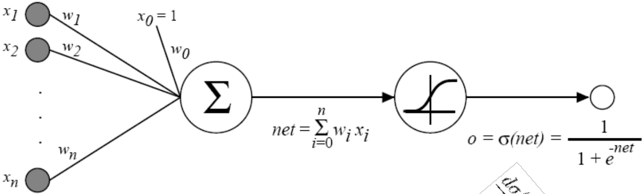
Learning rate

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

Training a perceptron with linear activation

```
- Initialize weights $w_i$ to a small random value (ex. range [0,1])
- do
    - for (x,t) in training set
        - Input x and obtain output out
        - For each weight $w_i$ compute gradient $\Delta w_i$:
            $\Delta w_i = - (\mathbf{t} - \mathbf{out})\, x_i$
      end for
    - Update weight values:
            $w_i^{new} = w_i^{old} - \eta\Delta w_i$
- while (End condition ! = TRUE).
```

Adding a sigmoidal activation


x_1
w_1
x_2
w_2
$x_0 = 1$
w_0
Σ
w_n
x_n
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

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

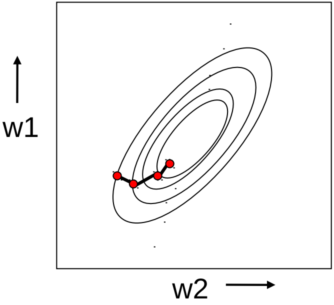
Training Rule

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

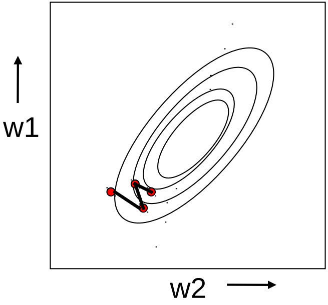
$$\frac{\partial E}{\partial w_i} = - \sum_{d \in D} (t_d - o_d) o_d (1 - o_d) x_{i,d}$$

Training a perceptron: batch vs incremental modes

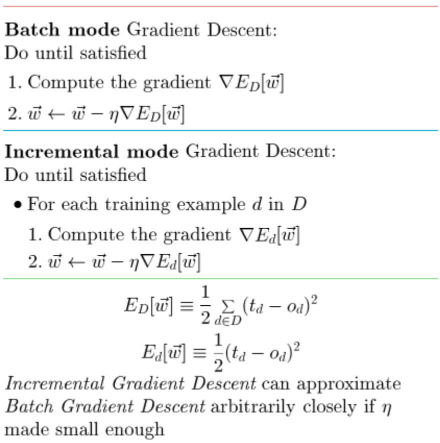
Batch mode



Incremental mode



Training a perceptron batch vs incremental modes



Batch mode Gradient Descent:

```
Do until satisfied
 1. Compute the gradient $\nabla E_D[\vec{w}]$
 2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$
```

Incremental mode Gradient Descent:

```
Do until satisfied
 • For each training example $d$ in $D$
   1. Compute the gradient $\nabla E_d[\vec{w}]$
   2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$
```

$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$E_d[\vec{w}] \equiv \frac{1}{2} (t_d - o_d)^2$$

Incremental Gradient Descent can approximate *Batch Gradient Descent* arbitrarily closely if η made small enough

Incremental mode: Stochastic gradient descent (SGD)

- Most popular way to implement the incremental mode is using a stochastic gradient descent approach.
- SGD main steps:
  - Randomly select a small subset (mini-batch) of the training examples.
  - Estimate direction of the gradient to minimize error over the mini-batch, i.e, estimate gradient of:

$$E[\vec{w}] = \frac{1}{2} \sum_{batch \in D} (t_d - o_d)^2$$

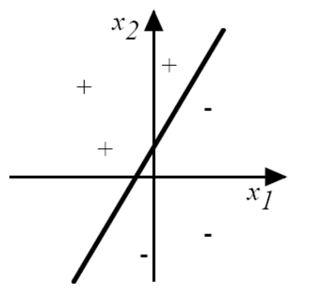
  - Update weights using direction of the gradient over the mini-batch.

Training a perceptron: Sthocastic gradient descent (SGD)

- SGD is fast and effective in large datasets. It effectively uses the redundancy in the data.
- Ex. Consider a dataset with 1M examples, where each example is repeated 100 times, i.e., the number of different data is 10K.
- In average, after each epoch, SGD has made the equivalent of 100 iterations over the 10K different data, why?.
- Batch mode will be 100 times slower (why?).

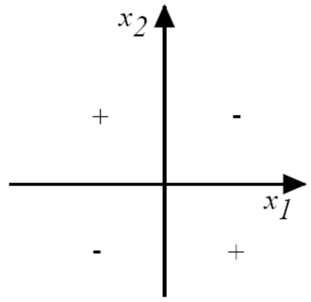
What can a perceptron learn?

A perceptron can learn linearly separable functions (why?).



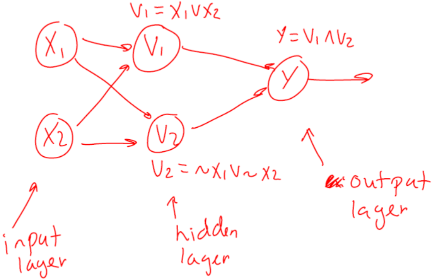
What can a perceptron learn?

What about non-linear functions, like the exclusive OR?



What can a perceptron learn?

$$x_1 \oplus x_2 = (x_1 \vee x_2) \wedge (\sim x_1 \vee \sim x_2)$$

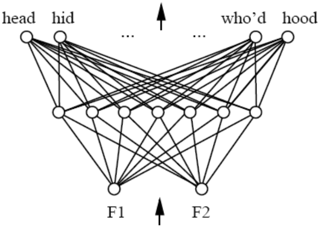


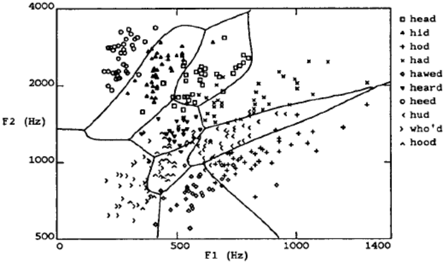
New problem:
How can we train a NN with hidden layers?

Backpropagation algorithm

Backpropagation algorithm

Backpropagation allows us to train multilayer networks

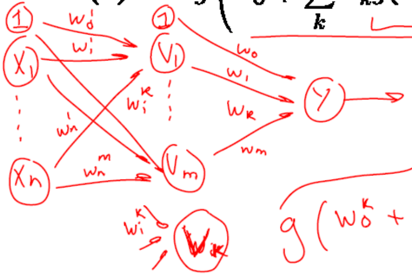




Backpropagation algorithm

Backpropagation allows us to train multilayer networks

$$out(\mathbf{x}) = g\left(w_0 + \sum_k w_k \underline{g(w_0^k + \sum_i w_i^k x_i)}\right)$$



$$\overbrace{g(w_0^k + \sum_i w_i^k x_i)}$$

Training NNs with Backpropagation

Useful Training tips

Backpropagation algorithm useful training tips

When to stop iterating?

- Several possible criteria:
  - After a fixed number of epochs.
  - After error converges to some value.
  - When error reaches a desired value.
  - When we detect overfitting

Backpropagation useful training tips

How to initialize the weights?

- Several possible criteria:
  - Random small values.
  - Values from a related problem (transfer learning).
- To avoid local optimum, start the network with different sets of initial weights (why?).

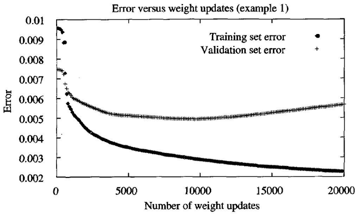
Backpropagation useful training tips

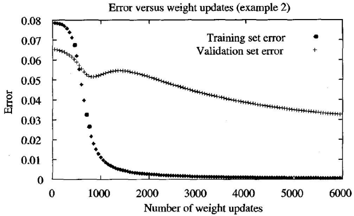
How to manage the learning rate?

- In general the rule is:
    - Far from the optimum, use a large learning rate $\rightarrow \approx 1$.
    - Close to the optimum, use a small value $\rightarrow \approx 0$.
- Adjust the learning rate during training using an annealing strategy:
    - Reduce learning rate according to a pre-defined schedule or when the change in objective between epochs falls below a threshold.

Backpropagation useful training tips

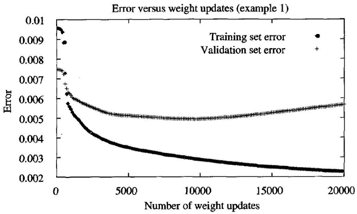
Is there overfitting in NNs?

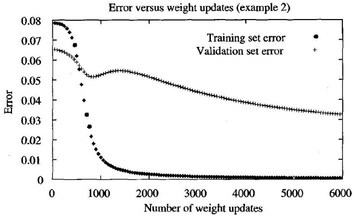




Backpropagation useful training tips

How can I avoid the overfitting?





Backpropagation useful training tips

How to accelerate or improve learning?

- Use momentum factor to avoid noisy stochastic gradient steps.
- Change the order of the training data after each epoch.
- Normalize input data (ex. $\mu = 0, \sigma = 1$).

More tips: “Neural Networks, Tricks of the Trade”, ed. G. Montavon, G. B. Orr, and K-R Müller, 2012, Springer.

NN example: NETtalk (Sejnowski & Rosenberg, 1986)

- Task: learning to pronounce written English words.
- Training set: 1024 words with their corresponding phonemes.
- Input: 7 consecutive characters using a sliding window over a given input sentence.
- Output: phonen codes corresponding to input text.
- Structure: 3 layers. 7x29 inputs (26 chars + special signs), 80 neurons in hidden layer, 26 neurons in output layer (phonen codes).
- Sigmoidal activation functions.
- Results: 95% accuracy in training set after 50 epochs using backpropagation.
- Results: 78% accuracy in test set.

Why people stop using NNs?

- In general, NNs are difficult to train, several parameters to adjust.
- During the 90s, techniques such as SVMs and ensemble methods (random forest, adaboost, etc) provided easier training and better results.
- Therefore, over that time, NNs were not so popular.
- Furthermore, there was a notion that convergence to local optimum was a severe problem for gradient descent based training methods.

Neural networks are back!

- Today, big data is the key new engine to move NN models to new level of performance.
- Big data allows gradient optimization methods to reach excellent performance, in terms of efficiency and accuracy, why?,
- What happen with local optimums?
- Furthermore, as an improved modeling scheme, deep learning schemes add efficiency in terms of the number of parameters needed to fit a input-output relation.
- While, NNs with 1 or 2 hidden layers can model all possible functions, they are highly inefficient.

We will cover deep learning in short!