Unit 4: Artificial Neural Networks

F. J. Martín Mateos
A. Riscos Núñez
J. L. Ruiz Reina

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Artificial neurons: biological inspiration

- *Artificial neural networks* are a computational model inspired on their biological counterpart
- However, do not forget it is just a formal model:
  - Some features/properties of the biological systems are not captured in the computational model and viceversa
- Our approach is to use them as a mathematical model that is the base of powerful automated learning algorithms
  - Structure: directed graph (nodes are *artificial neurons*)

How a neural network works

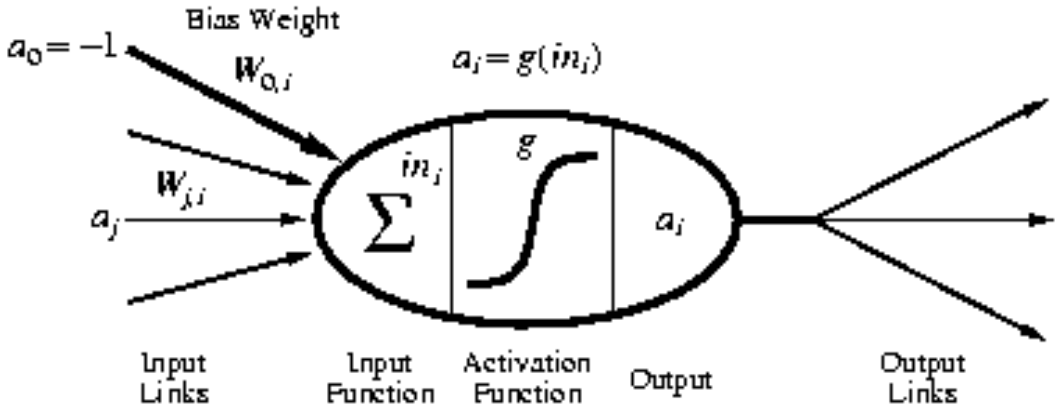
- Each node or *unit* (*artificial neuron*), is connected to other units via *directed arcs* (modeling the *axon* $\rightarrow$ *dendrites* connection)
- Each arc $j \rightarrow i$ propagates the *output* of unit j (denoted a_j) which in turn is one of the *inputs* for unit i. The inputs and outputs are *numbers*
- Each arc $j \rightarrow i$ has an associated numerical weight w_{ji} which determines the strength and the sign of the connection (simulating a *synapse*)

How a neural network works

- Each unit calculates its output according to the received inputs
- The output of each unit is, in turn, used as one of the inputs of other neurons
  - The calculation performed by each unit is very simple, as we will see later on
- The network receives a series of external inputs (*input units*) and returns the output of some of its neurons, called *output units*

Calculation at each unit

- The output of a unit is: $a_i = g(\sum_{j=0}^{n} w_{ji} a_j)$



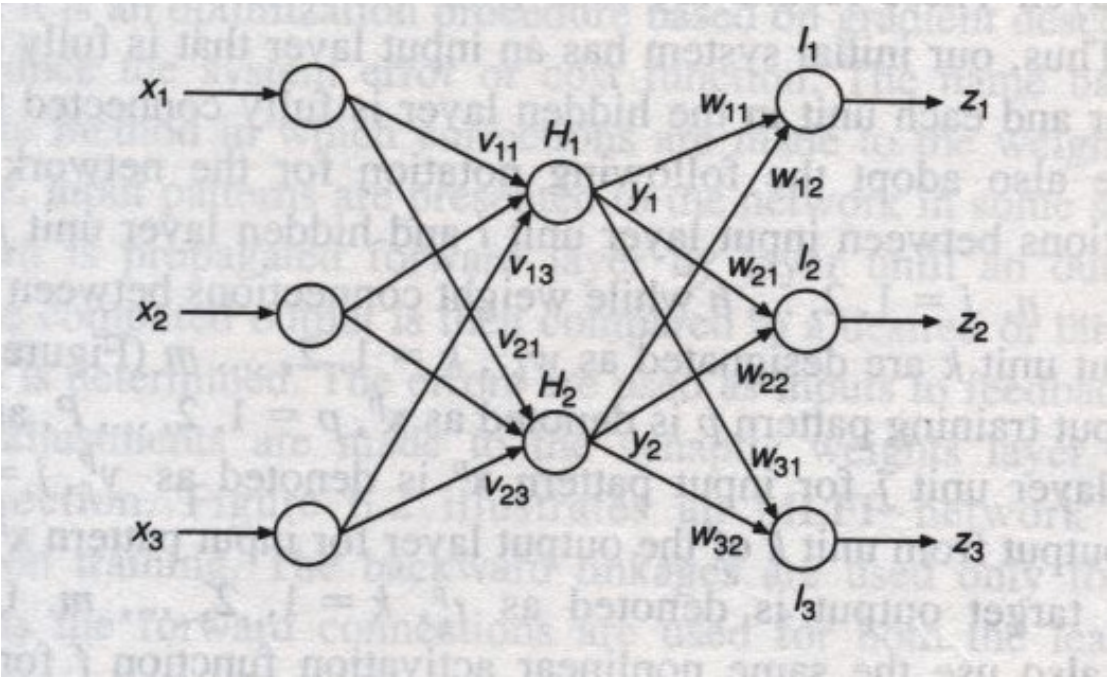
- Where:
  - g is the *activation function*
  - The summation $\sum_{j=0}^{n} w_{ji} a_j$ (denoted by in_i) gathers all the outputs of units j connected with unit i
  - Except for $j = 0$, which is considered as a virtual input $a_0 = -1$ and has a weight w_{0i} called *bias*

Bias and activation functions

- Intuitively, the bias weight w_{0i} of each unit is interpreted as the minimum amount that the sum of the received input signals has to reach in order to activate the unit
- The role of the activation function g is to "normalize" the output (usually to 1) upon activation. Besides, this ingredient allows that the network offers something more than a simple lineal function
- Frequently used activation functions:
  - Bipolar: $sgn(x) = \begin{cases} 1 & \text{si} \quad x > 0 \\ -1 & \text{si} \quad x \leq 0 \end{cases}$
  - Threshold: $threshold(x) = \begin{cases} 1 & \text{si} \quad x > 0 \\ 0 & \text{si} \quad x \leq 0 \end{cases}$
  - Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
  - The sigmoid function is derivable and $\sigma'(x) = \sigma(x)(1 - \sigma(x))$

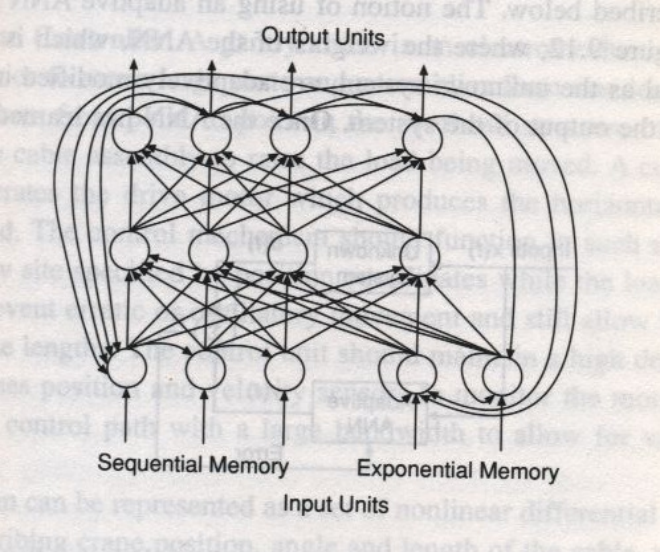
Feed-forward neural networks

- A *feed-forward* neural network is an artificial neural network where connections between the units do not form a directed cycle (we focus on them in this unit)



Feed-forward neural networks

- Units in a feed-forward neural network are usually structured into *layers*, in such a way that each layer receives its inputs from units at the layer immediately before it
  - input layer, *hidden* layers and output layer
  - *Multi-layer* networks
- Other architectures: *recurrent* networks, where the output units provide feedback to the input units



Neural networks as classifiers

- A feed-forward neural network with n units on the input layer and m units on the output layer computes a function from R^n into R^m
- Thus, it can be used as a classifier for sets in R^n:
  - For Boolean classification, take $m = 1$ and:
    - If threshold or bipolar activation functions are used, then one of the outputs (e.g. 1) is interpreted as "YES" and the other one as "NO"
    - If sigmoid is used, then consider all output values over 0.5 as "YES" and any lower value as "NO"
  - In general, for classifiers with m possibles values, each output unit corresponds to a classification value; and then the unit having the higher output provides the classification

Neural networks and Learning

- *Learning* or *training*, in the framework of artificial neural networks, means searching adequate weights for the arcs, in order to get a desired behaviour (given by a training set)
- More precisely, in feed-forward neural networks, we usually follow the following *supervised learning* scheme
  - Given a training set $D = \{(\vec{x_d}, \vec{y_d}) : \vec{x_d} \in R^n, \vec{y_d} \in R^m, d = 1, \ldots, k\}$
  - And given a structure of a network (number of layers and units per layer)
  - Find a set of weight values w_{ij} such that the function from R^n into R^m represented by the network provides the *best fit* with the examples in the training set
- We need a precise definition of "best fit"

Practical applications of neural networks

- For problems that can be expressed in terms of numbers (discrete or continuous)
- Usually suitable for domains with a huge amount of data, possibly with *noise*: cameras, microphones, digitalized images, etc
- We only care about the solution, not *why* it is so
- Problems where we can afford a long training time for the network
- And we want fast evaluation of new instances

ALVINN: an example of an application

- ANN trained to drive a car, at 70 Km/h, according to the visual perception received as input from its sensors
- Input of the network: The image of the road digitalized as an array of 30×32 pixels. That is, 960 input data
- Output of the network: Indication about turning the wheel, encoded as a 30 component vector (from *turn completely to the left*, to *keep straight*, and then all the way to *turn completely to the right*)
- Structure: feed-forward network, input layer having 960 units, one hidden layer having 4 units and an output layer with 30 units

ALVINN: an example of an application

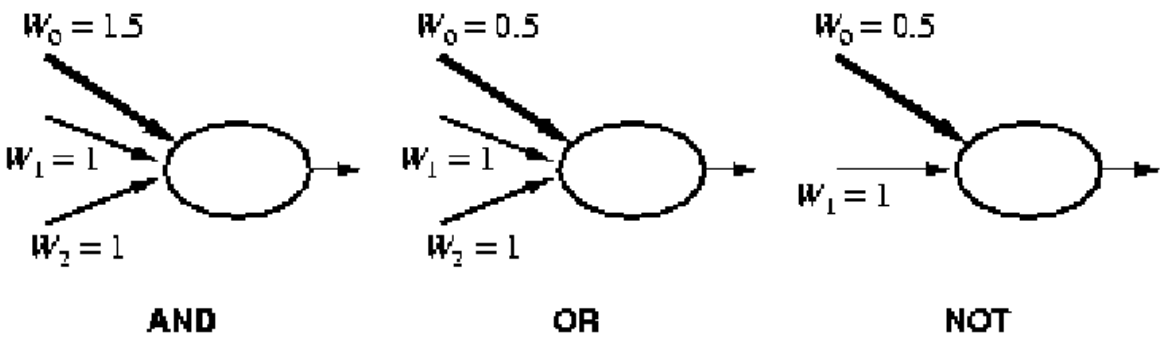
- Training: using a human driver, that drives the car again and again
- The visual sensors record the image seen by the driver (sequences of 960 data each)
- Other sensors record simultaneously the movements of the wheel
- After properly encoding all the gathered info, we have a number of different pairs of the form $(\vec{x}, \vec{y})$, where $\vec{x} = (x_1, x_2, \ldots, x_{960})$ and $\vec{y} = (y_1, y_2, \ldots, y_{30})$, constitute examples of *input/output* for the network
- Goal: find the best values for w_{ji} associated to arcs $j \to i$ such that when the network receives $\vec{x}$, its output matches the corresponding value $\vec{y}$ (or is the *best possible approximation*)

Examples of practical applications

- Classification
- Pattern recognition
- Optimization
- Prediction: weather, audience, etc
  - Speech recognition
  - Artificial vision, image recognition
- Constraint satisfaction
- Control (robots, cars, etc)
- Data compression
- Diagnosis

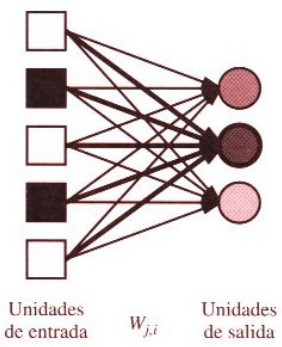
Perceptrons

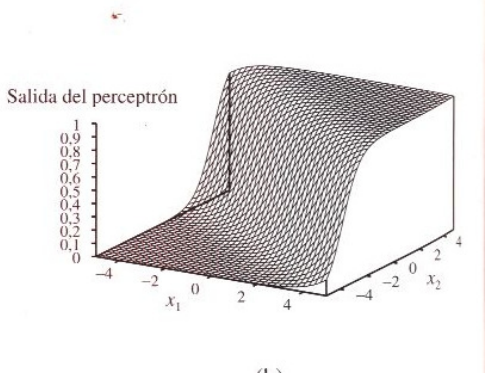
- Let us first focus on the simplest case of neural network: feed-forward, just one input layer and one output layer.
  - Since each of the output units is independent, without loss of generality we can consider just one unit in the output layer
- This type of network is called *perceptron*
- A perceptron using threshold activation function is able to represent the basic Boolean functions:



Perceptrons: limitations

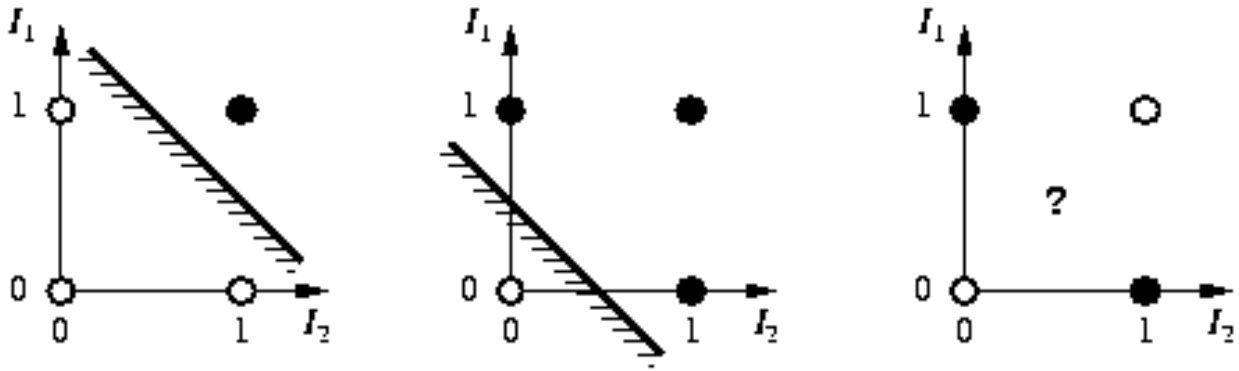
- A perceptron with n input units, weights $w_i (i = 0, \ldots, n)$ and threshold (or bipolar) activation function, accepts those $(x_1, \ldots, x_n)$ such that $\sum_{i=0}^{n} w_i x_i > 0$ (where $x_0 = -1$)
  - The equation $\sum_{i=0}^{n} w_i x_i = 0$ represents a *hyperplane* in R^n
  - That is, a Boolean function can only be represented by a threshold perceptron if there exists an hyperplane separating positive elements from negative elements (*linearly separable*)
- Perceptrons with sigmoid activation function have similar expressive limitations





Perceptrons: limitations

- For example, functions AND and OR are linearly separable, while XOR is not:



(a) I_1 **and** I_2 (b) I_1 **or** I_2 (c) I_1 **xor** I_2

- Despite their expressive limitations, perceptrons have the advantage that there exists a simple training algorithm for perceptrons with threshold activation function
  - Able to find the suitable perceptron for any linearly separable training set

Algorithm for (threshold) Perceptron training

- Input: A training set D (with examples of the form $(\vec{x}, y)$, with $\vec{x} \in R^n$ and $y \in \{0, 1\}$), and a learning factor η

Algorithm

1) Consider randomly generated initial weights $\vec{w} \leftarrow (w_0, w_1, \ldots, w_n)$
2) Repeat until halting condition
   1) For each $(\vec{x}, y)$ in the training set do
      1) Calculate $o = \mathit{threshold}(\sum_{i=0}^{n} w_i x_i)$ (with $x_0 = -1$)
      2) For each w_i do: $w_i \leftarrow w_i + \eta(y - o)x_i$
3) Return $\vec{w}$

Comments about the algorithm

- η is a positive constant, usually very small (e.g. 0.1), called *learning factor*, that moderates the weights update process
- At each iteration, if $y = 1$ and $o = 0$, then $y - o = 1 > 0$, and therefore w_i corresponding to positive x_i will increase (and those corresponding to negative x_i will decrease). Thus, o (real output) will come closer to y (expected output)
- Analogously for $o = 1$ and $y = 0$
- When $y = o$, no w_i is modified
- For perceptrons with bipolar activation function, the algorithm is analogous

Comments about the algorithm

- Theorem: The previous algorithm *converges* on a finite number of steps to a weights vector $\vec{w}$ that classifies correctly all training examples, provided that they are *linearly separable* and η is *small enough* (Minsky and Papert, 1969)
- Therefore, for the case of linearly separable training sets, the halting condition can be to have all examples correctly classified

Another training algorithm: the Delta rule

- When the training set is not linearly separable, the convergence of the previous algorithm is not guaranteed
- In this case, it will not be possible to find a perceptron able to return the expected output for *every* element of the training set
- We aim at least to minimize the *squared error*:
  $E(\vec{w}) = \frac{1}{2}\sum_d (y_d - o_d)^2 = \frac{1}{2}\sum_d [y_d - g(w_0x_0 + w_1x_1 + \cdots + w_nx_n)]^2$
  - where g is the activation function, y_d is the expected output for the instance $(\vec{x}_d, y_d) \in D$, and o_d is the output returned by the perceptron
  - Note that E is a function over $\vec{w}$, and the goal is to find a particular $\vec{w}$ minimizing E
- In what follows, we refer to perceptrons having an activation function g being derivable (e.g. sigmoid)
- Thus, we discard threshold and bipolar perceptrons

Training algorithm by gradient descent

- Input: A training set D (with examples of the form $(\vec{x}, y)$, where $\vec{x} \in R^n$ and $y \in R$), a learning factor η and a derivable activation function g

Algorithm

1) Consider randomly generated initial weights $\vec{w} \leftarrow (w_0, w_1, \ldots, w_n)$
2) Repeat until halting condition
   1) Initialize Δw_i to zero, for $i = 0, \ldots, n$
   2) For each $(x, y) \in D$,
      1) Calculate $in = \sum_{i=0}^{n} w_i x_i$ and $o = g(in)$
      2) For each $i = 0, \ldots, n$, do
         $\Delta w_i \leftarrow \Delta w_i + \eta(y - o)g'(in)x_i$
   3) For each w_i do: $w_i \leftarrow w_i + \Delta w_i$
3) Return $\vec{w}$

The Delta rule

- It is a variant of the gradient descent method
- Instead of trying to minimize the squared error related to *all* examples in D , performs *iterative improvements* trying to reduce the squared error $E_d(\vec{w}) = \frac{1}{2}(y - o)^2$, with respect to each single example $(\vec{x}, y) \in D$
  - In this way, $\frac{\partial E_d}{\partial w_i} = (y - o)g'(in)(-x_i)$, and since $\Delta w_i = -\eta \frac{\partial E_d}{\partial w_i}$, we deduce that $\Delta w_i = \eta(y - o)g'(in)x_i$, and therefore $w_i \leftarrow w_i + \eta(y - o)g'(in)x_i$
  - This method for iteratively improving weights is known as *Delta rule*

Training perceptrons with the Delta rule

- Input: A training set D (with examples of the form $(\vec{x}, y)$, where $\vec{x} \in R^n$ and $y \in R$), a learning factor η and a derivable activation function g

Algorithm

1) Consider randomly generated initial weights
   $\vec{w} \leftarrow (w_0, w_1, \ldots, w_n)$
2) Repeat until halting condition
   1) For each $(\vec{x}, y) \in D$
      1) Calculate $in = \sum_{i=0}^{n} w_i x_i$ and $o = g(in)$
      2) For each w_i do
         $w_i \leftarrow w_i + \eta(y - o)g'(in)x_i$
3) Return $\vec{w}$

Particular cases of the Delta rule

- Perceptrons having a *linear* activation function:
  - In this case $g'(in) = C$ (constant)
  - Therefore, the Delta rule is simply (adjusting η conveniently):

$$w_i \leftarrow w_i + \eta(y - o)x_i$$

- Perceptrons having sigmoid activation function:
  - In this case $g'(in) = g(in)(1 - g(in)) = o(1 - o)$
  - Therefore, the Delta rule is simply:

$$w_i \leftarrow w_i + \eta(y - o)o(1 - o)x_i$$

Some comments on the Delta rule

- Both gradient descent and the Delta rule are local search algorithms, which converge towards local minima for the error between obtained output and expected output
  - In the case of gradient descent, we decrease at each step following the gradient of the squared error associated to *all* examples
  - In the Delta rule, we decrease at each step following the gradient of the squared error associated to *a single* example
- Having a small enough value for η, the gradient descent method converges (in general asymptotically) towards a local minimum of the global squared error

Some comments on the Delta rule (contd.)

- It has been proved that by using small enough values for η, the Delta rule gives an arbitrarily close approximation of the gradient descent
- Delta rule updates the weights in a simpler way, although smaller values of η are needed. Besides, it is more likely to escape from local minima

Delta rule and threshold perceptrons

- The training rule for threshold perceptron and the Delta rule for training linear perceptrons seem apparently identical: $w_i \leftarrow w_i + \eta(y - o)x_i$, however:
  - Their activation functions are different
  - The **convergence** properties are different too:
    - Threshold: converges in a finite number of steps and reaches a perfect adjustment, provided that the training set is linearly separable
    - Delta rule: always converges asymptotically towards a local minimum of the squared error
  - The **separation** properties are different as well:
    - Threshold: looks for a hyperplane that fully separates the data
    - Delta rule: looks for a regression model, the hyperplane (possibly smoothed with the sigmoid) that lies closest to the training data

(Feed-forward) Multi-layer networks

- As mentioned before, perceptrons have a limited expressive power. Let us therefore study multi-layer networks

  figuras/rn-png-converted-to.png

- In a multi-layer network, units are arranged into *layers*, in such a way that each unit of each layer (except input layer) receive inputs from all units from the previous layer
  - *Input* layer is formed by input units
  - *Output* layer is formed by units who send their output outside
  - *Hidden* layers are all the remaining layers different from the input and output layers

Multi-layer networks: expressive power

- Combining multiple layers increases the network expressive power (provided that the activation function is not linear)
- That is, the amount of functions $f : R^n \rightarrow R^m$ which can be represented is increased

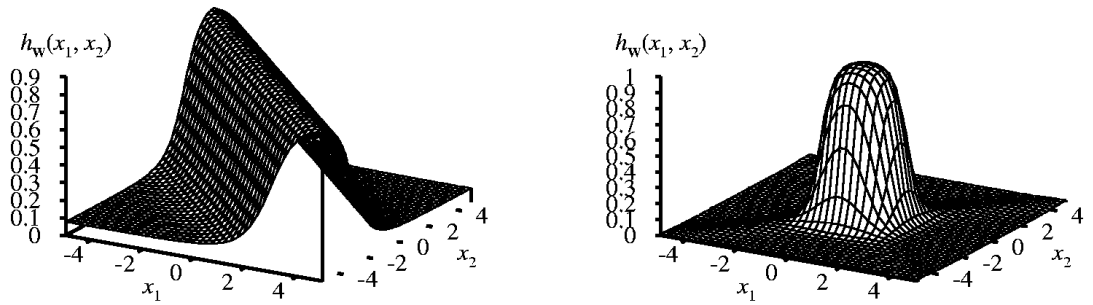


Figure 20.23 (a) The result of combining two opposite-facing soft threshold functions to produce a ridge. (b) The result of combining two ridges to produce a bump.

- Usually, a single hidden layer is enough for most of real applications

Multi-layer networks training

- Similarly to the perceptron case, we have a training set D such that each $(\vec{x}, \vec{y}) \in D$ shows the expected output $\vec{y} \in R^m$ for each input $\vec{x} \in R^n$
- We assume that the structure of the multi-layer network is given, the goal is to find the appropriate weights for the network in such a way that the function computed by the net fits in the best way the given examples
- *Retropropagation algorithm*: successive updates of the weights, following the same approach as the gradient descent for the perceptron

Multi-layer networks: notation

- Consider a neural network having L layers, with n units at the input layer, and m at the output layer
  - Layer 1 is the input one, and layer L is the output one
  - Each unit of layer l is connected with all units from layer $l+1$
- We assume that the activation function g is differentiable (usually the sigmoid)
- The weight associated with the arc from unit i to unit j is denoted by w_{ij}
- Given a training example $(\vec{x}, \vec{y}) \in D$:
  - Each component x_i from $\vec{x}$ is associated with its corresponding input unit
  - For each unit k within the output layer, we denote by y_k the corresponding component of $\vec{y}$ associated to this unit

Multi-layer networks: notation

- When computing the real output produced by the network when receiving as input an example $\vec{x}$, for each unit i we shall denote the input received by it as in_i, and the output of the unit as a_i.

- More precisely:
  - If i is an input unit (i.e., from layer 1), then $in_i = a_i = x_i$
  - If i is a unit within a layer $l \neq 1$, then $in_i = \sum_j w_{ji} a_j$ and $a_i = g(in_i)$ (where the sum is made over all units j from layer $l - 1$)

Retropropagation algorithm: intuitive idea

- Given an example $(\vec{x}, \vec{y}) \in D$, for each unit i within the output layer, the weights of the incoming arcs to this unit is made similarly as with the Delta rule:
  - Let $\Delta_i = g'(in_i)(y_i - a_i)$ (*modified error* in unit i)
  - Then $w_{ji} \rightarrow w_{ji} + \eta a_j \Delta_i$
- However, it is not possible to do the same for hidden layers
  - The "expected output" is unknown

Retropropagation algorithm: intuitive idea

How to update weights of incoming arcs for hidden layers?

- **Idea**: go backwards, in order to calculate error Δ_j of a unit within layer $l-1$, we take into account errors of units within layer l (unit j is connected to all of them)

$$\Delta_j = g'(in_j) \sum_i w_{ji} \Delta_i \qquad\qquad w_{kj} \rightarrow w_{kj} + \eta a_k \Delta_j$$

- Intuitively, each unit j takes "responsibility" for the errors of the units where it sends its output
  - depending on weight of the connection

Retropropagation algorithm: intuitive idea

- The output of each unit is calculated by propagating values forward, but their errors are calculated backwards starting from the output layer (thus the name *retropropagation*)
- The retropropagation method can be formally justified as a gradient descent of the error, but we skip the proof

Retropropagation algorithm

- Input: A training set D (with examples of the form $(\vec{x}, \vec{y})$, with $\vec{x} \in R^n$ and $\vec{y} \in R^m$), a learning factor η, a differentiable activation function g, and the structure of the *net*

Algorithm

1) Initialize the weights on the *net* (randomly, usually values close to zero, either positive or negative)
2) Repeat until halting condition
   1) For each example $(\vec{x}, \vec{y}) \in D$ do:
      1) Calculate the output a_i of each unit i, propagating values forward
      2) Calculate errors Δ_i of each unit i, and update weights w_{ji}, propagating values backwards
3) Return *net*

- *In the following slides items 2.1.1) and 2.1.2) will be explained in detail*

Forward propagation

2.1.1) forward propagation for an example $(\vec{x}, \vec{y}) \in D$

Procedure

1) For each node i within the input layer, $a_i \leftarrow x_i$
2) For each l from 2 to L do:
    1) for each node i in layer l do
        $in_i \leftarrow \sum_j w_{ji} a_j$ (sum for each unit j in layer $l-1$)
        $a_i \leftarrow g(in_i)$

- Once the values in_i and a_i have been computed for an example $(\vec{x}, \vec{y}) \in D$, let us focus on the next step

Backwards propagation

2.1.2) backwards propagation of errors for an example $(\vec{x}, \vec{y}) \in D$, and weights update

Procedure

1) For each unit i within output layer,
   $\Delta_i \leftarrow g'(in_i)(y_i - a_i)$
2) For each l from $L-1$ down to 1 do:
   1) for each node j in layer l do
      1) $\Delta_j \leftarrow g'(in_j) \sum_i w_{ji} \Delta_i$ (sum for each unit i in layer $l+1$)
      2) for each node i in layer $l+1$ do
         $w_{ji} \leftarrow w_{ji} + \eta a_j \Delta_i$

- No need to calculate Δ_j for the input layer ($l = 1$)
- Bias weights are updated after calculating each Δ_i, just like any other weight: $w_{0i} \leftarrow w_{0i} + \eta a_0 \Delta_i$ (where $a_0 = -1$)

Retropropagation with sigmoid units

- The most common case where retropropagation algorithm is applied involves a network having the sigmoid activation function
- Recall $\sigma(x) = \frac{1}{1+e^{-x}}$, and $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
- Therefore, when $g(x) = \sigma(x)$, we have $g'(in_i) = g(in_i)(1 - g(in_i)) = a_i(1 - a_i)$
- In this case, the error calculation in Step 2 is:
  - For the output layer, $\Delta_i \leftarrow a_i(1 - a_i)(y_i - a_i)$
  - For the hidden layers, $\Delta_j \leftarrow a_j(1 - a_j) \sum_i w_{ji} \Delta_i$
- Notice that it is not necessary to store the values in_i from Step 1, since they are not used here

Retropropagation: a running example

- Let us consider a neural network with the following structure which uses the sigmoid as activation function:

  figuras/rn-png-converted-to.png

  - Let (x_1, x_2, x_3) be a given example with expected output (y_6, y_7)
  - Let us assume that we have *already calculated* the output a_i for every unit $i = 1, \ldots, 7$

Retropropagation: a running example

Trace of the error retropropagation

Layer	Unit	Operations
Output	7	$\Delta_7 = a_7(1 - a_7)(y_7 - a_7)$
		$w_{0,7} \longleftarrow w_{0,7} + \eta a_0 \Delta_7$
	6	$\Delta_6 = a_6(1 - a_6)(y_6 - a_6)$
		$w_{0,6} \longleftarrow w_{0,6} + \eta a_0 \Delta_6$
Hidden	5	$\Delta_5 = a_5(1 - a_5)[w_{5,6}\Delta_6 + w_{5,7}\Delta_7]$
		$w_{0,5} \longleftarrow w_{0,5} + \eta a_0 \Delta_5$
		$w_{5,6} \longleftarrow w_{5,6} + \eta a_5 \Delta_6$
		$w_{5,7} \longleftarrow w_{5,7} + \eta a_5 \Delta_7$
	4	$\Delta_4 = a_4(1 - a_4)[w_{4,6}\Delta_6 + w_{4,7}\Delta_7]$
		$w_{0,4} \longleftarrow w_{0,4} + \eta a_0 \Delta_4$
		$w_{4,6} \longleftarrow w_{4,6} + \eta a_4 \Delta_6$
		$w_{4,7} \longleftarrow w_{4,7} + \eta a_4 \Delta_7$
Input	3	$w_{3,4} \longleftarrow w_{3,4} + \eta a_3 \Delta_4$
		$w_{3,5} \longleftarrow w_{3,5} + \eta a_3 \Delta_5$
	2	$w_{2,4} \longleftarrow w_{2,4} + \eta a_2 \Delta_4$
		$w_{2,5} \longleftarrow w_{2,5} + \eta a_2 \Delta_5$
	1	$w_{1,4} \longleftarrow w_{1,4} + \eta a_1 \Delta_4$
		$w_{1,5} \longleftarrow w_{1,5} + \eta a_1 \Delta_5$

Momentum in retropropagation algorithm

- Retropropagation is a gradient descent method, and thus entails the problem of local minima
- A quite common variant of the retropropagation algorithm is to include an extra addend in the weight updating formula
- This new addend involves that each weight update takes into account the update performed in the previous step
- More precisely:
  - On the n-th iteration, weights are updated as follows: $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}^{(n)}$ where $\Delta w_{ji}^{(n)} = \eta a_j \Delta_i + \alpha \Delta w_{ji}^{(n-1)}$
  - $0 < \alpha \leq 1$ is a constant called *momentum*
- The momentum technique might be effective sometimes to escape from "small local minima", whereas the standard version of the algorithm could get stuck

Halting condition for retropropagation

- Notice that the algorithm might loop *several times* over the training set
  - Or could take a random example on each iteration
  - Or could even halt and then resume the training, but starting from learned weights

Examples of halting criteria

- Fixed number of iterations
- When error over the training set gets below a predetermined threshold
  - Risk of overfitting
  - independent test set, or cross-validation

Learning the network structure

- Retropropagation algorithm receives as input a fixed structure
- Which structure is suitable for each problem?
- In particular, here we need to decide the number of hidden layers, and how many units for each layer
- This problem is not fully solved yet
- Usually, the best structure is searched experimentally, measuring it over an independent test set
- Most of the times, a single hidden layer with a few units suffices to provide good results
- Larger networks involve higher risk of overfitting

Final comment .... Recurrent networks

Recurrent networks

- Short term memory
- Better modelization of the brain
- Harder to understand
- Directed graph (cycles allowed)
- Output of a unit is allowed to feed its own input
- Acts like a dynamic system
- May reach stability, or may exhibit oscillatory or chaotic behaviour

Bibliography

- Russell, S. and Norvig, P. *Artificial Intelligence (A modern approach)* (Third edition) (Prentice–Hall, 2010)
  - Ch. 18: "Learning from Examples"
- Russell, S. and Norvig, P. *Artificial Intelligence (A modern approach)* (Second edition) (Prentice Hall, 2003)
  - Ch. 20: "Statistical Learning" (available on-line at http://aima.cs.berkeley.edu/2nd-ed/newchap20.pdf)
- Mitchell, T.M. *Machine Learning* (McGraw-Hill, 1997)
  - Ch. 4: "Artificial Neural Networks"