Artificial neural networks: Basic Algorithms

Laboratorio Calcolo Matematico Modulo Reti Neurali

Juan Rojo

INFN, Sezione di Milano

Milano, 15/12/2009

Basic algorithms

In this lecture we discuss:

1. Basic algorithms of an artificial neural network
2. Input/Output preprocessing
3. Learning strategies: Back propagation and Genetic Algorithms

BASICS OF ARTIFICIAL NETS

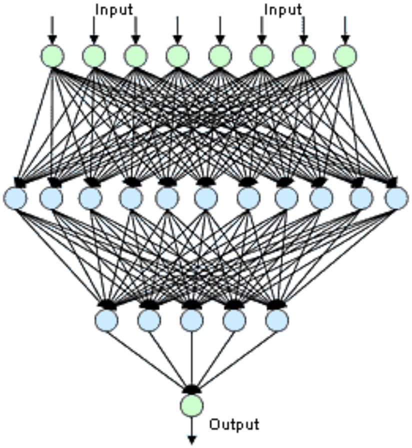
Multilayer neural networks

- An artificial neural network consists of a set of interconnected units (neurons)
- The state or activation of a given i-neuron, ξ_i, is determined as a function of the activation of the neurons connected to it
- Each pair of neurons (i,j) is connected by a synapsis, characterized by a weight ω_{ij}
- The activation of each neuron is a function g of the difference between a weighted average of input from other neurons and a threshold θ_i:

$$\xi_i = g\left(\sum_j \omega_{ij}\xi_j - \theta_i\right) .$$

where the sum in j runs over all neurons connected to neuron i

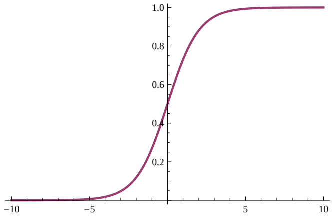
Multilayer neural networks



Multilayer neural networks - Activation function

- Activation function g is in general non-linear
- The simplest example of activation function $g(x)$ is the step function $g(x) = \Theta(x)$ (binary activation only)
- More refined choice is a continuous activation function with two distinct regimes (linear and non–linear) $\rightarrow$ the sigmoid

$$g(x) \equiv \frac{1}{1+e^{-\beta x}} .$$



Multilayer neural networks - Rosenblatt's Perceptrons

- We will focus in Rosenblatt's perceptrons (multilayer feed-forward neural networks)
- Organized in ordered layers whose neurons only receive input from a previous layer
- For L layers with $n_1, \ldots, n_L$ units, state of the multilayer neural network is established by recursive relations

$$\xi_i^{(l)} = g\left(\sum_{j=1}^{n_{l-1}} \omega_{ij}^{(l-1)} \xi_j^{(l-1)} - \theta_i^{(l)}\right) ; \quad i = 1, \ldots, n_l, \quad l = 2, \ldots, L,$$

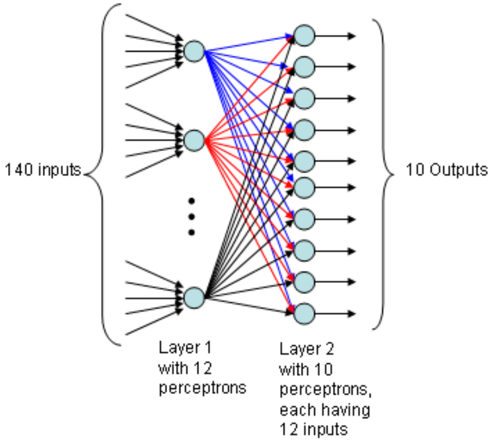
- The input is the vector $\boldsymbol{\xi}^{(1)}$ and the output the vector $\boldsymbol{\xi}^{(L)}$.
- Multilayer feed-forward neural networks can be viewed as functions F from $\mathbb{R}^{n_1} \longrightarrow \mathbb{R}^{n_L}$, parametrized by weights, thresholds and activation function,

$$\boldsymbol{\xi}^{(L)} = F\left[\boldsymbol{\xi}^{(1)}; \{w_{ij}^{(l)}\}, \{\theta_i^{(l)}\}; g\right] .$$

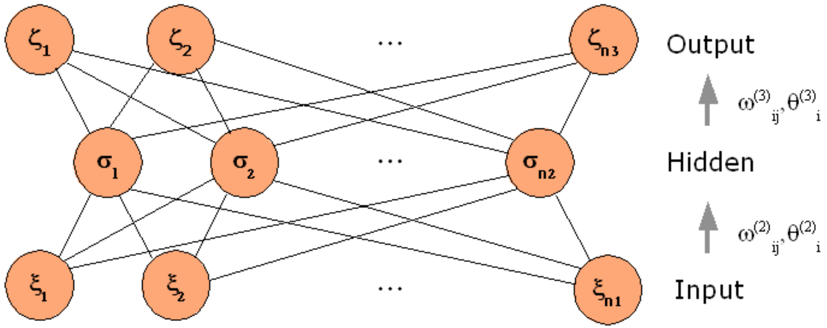
- The number of neurons and hidden layers define the architecture of the NN.
- Theorem: a multilayer perceptron with a single hidden layer can learn arbitrarily complex patters

Multilayer neural networks - Rosenblatt's Perceptrons

Perceptrons can be nested to obtain a *Meta Artificial Neural Network*


140 inputs
10 Outputs
Layer 1 with 12 perceptrons
Layer 2 with 10 perceptrons, each having 12 inputs

Multilayer neural networks



Multilayer neural networks - Preprocessing

- Preprocessing of the input/output patterns allows to speed up the NN learning and use smaller architectures
  $\rightarrow$ NNs learn more efficiently in the linear regime
- For example, if the input patterns span a wide range, logarithmic preprocessing should be used

$$\xi^{(1)} \rightarrow \widetilde{\xi}^{(1)} = \ln\left(\xi^{(1)}\right)$$

- Or if we know that the function the NN is learning vanishes at some value of the input pattern $\xi^{\hat{(1)}}$, then

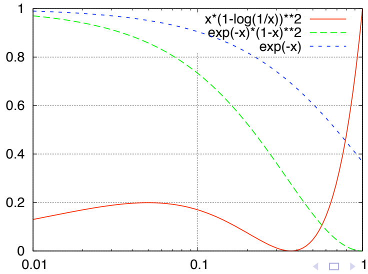
$$\begin{aligned} \xi^{(L)} &= F\left[\xi^{(1)}; \{w_{ij}^{(l)}\}, \{\theta_i^{(l)}\}; g\right] \rightarrow \\ \xi^{(L)} &= \left(\xi^{\hat{(1)}} - \xi^{(1)}\right) \hat{F}\left[\xi^{(1)}; \{w_{ij}^{(l)}\}, \{\theta_i^{(l)}\}; g\right] \end{aligned}$$

Multilayer neural networks - Preprocessing

How to use preprocessing to improve learning

$$\begin{aligned} y(x) &= x\left(1-\ln(1/x)\right)^2 \\ y\left(z=\ln(1/x)\right) &= e^{-z}\left(1-z\right)^2 \\ y(z)/(1-z)^2 &= e^{-z} \end{aligned}$$

The smoother the function to learn, with less extremes and changes of curvature, the easier the NN learning



Learning strategies

- The usefulness of neural networks is due to the availability of a training algorithm
- This algorithm allows one to select the values of weights and thresholds such that the neural network reproduces a given set of input–output data (or patterns)
- ‘Learning’ because, unlike a standard fitting procedure, there is no need to know in advance the underlying rule which describes the data
- Neural network generalizes the examples used to train it.

Supervised vs. unsupervised learning

Types of NN training:

- Unsupervised learning: the aim is to determine how data is organized, where the artificial NNet is given only unlabeled examples. Example: Self-Organizing maps
- Supervised learning: the aim is to deduce a function from training data. The task of artificial NNet is to predict the value of the function for any valid input object after having seen a number of training examples. Example: Perceptron

In this lectures we concentrate on supervised learning

- Backpropagation $\longrightarrow$ Deterministic learning
- Genetic Algorithms $\longrightarrow$ Stochastic learning

BACK-PROPAGATION

Learning strategies - Backpropagation

- Consider a set of input-output patterns $(\mathbf{x},\mathbf{z})$ which we want the neural network to learn
- The state of the neural network is generically given by

$$\xi_i^{(l)} = g(h_i^{(l)}), \qquad h_i^{(l)} = \sum_{j=1}^{n_{l-1}} \omega_{ij}^{(l-1)}\,\xi_j^{(l-1)} - \theta_i^{(l)}; \qquad i=1,\ldots,n_l, \quad l=1,\ldots,L.$$

- Input–output patterns correspond to pairs of states of the first and last layers, which we shall denote as

$$\mathbf{x} = \boldsymbol{\xi}^{(1)}\,, \qquad \mathbf{o}(\mathbf{x}) = \boldsymbol{\xi}^{(L)}.$$

- The goal of the training is to learn from a given set of data patterns, which consist of associations of a given input with a desired output
- For any given values of the weights and thresholds, define the error function

$$E[\omega,\theta] \equiv \frac{1}{2}\sum_{A=1}^{n_p}\sum_{i=1}^{n_L}\left(o_i(\mathbf{x}^A) - z_i^A\right)^2$$

(This is the χ^2 function for data without uncertainties)

Learning strategies - Backpropagation

- Error function minimized by looking for the direction of steepest descent

$$\delta\omega_{ij}^{(l)} = -\eta \frac{\partial E}{\partial \omega_{ij}^{(l)}} \,, \quad \delta\theta_{i}^{(l)} = -\eta \frac{\partial E}{\partial \theta_{i}^{(l)}} \,,$$

  where η fixes the learning rate.
- Steepest descent direction

$$\begin{aligned} \frac{\partial E}{\partial \omega_{ij}^{(l)}} &= \sum_{A=1}^{n_p} \Delta_i^{(l)A} \xi_j^{(l-1)A}; \qquad i = 1, \ldots, n_l, \quad j = 1, \ldots, n_{l-1} \,, \\ \frac{\partial E}{\partial \theta_{i}^{(l)}} &= -\sum_{A=1}^{n_p} \Delta_i^{(l)A}, \qquad i = 1, \ldots, n_l, \end{aligned}$$

  where information on goodness of fit fed to last layer through

$$\Delta_i^{(L)A} = g'\left(h_i^{(L)A}\right)\left[o_i(\mathbf{x}^A) - z_i^A\right]$$

  and then it is back-propagated to the rest of the network by

$$\Delta_j^{(l-1)A} = g'\left(h_j^{(l-1)A}\right) \sum_{i=1}^{n_l} \Delta_i^{(l)A} \omega_{ij}^{(l)} \,.$$

Learning strategies - Backpropagation

The back-propagation algorithm then consists of the following steps:

1. initialize all the weights and thresholds randomly, and choose a small value for the learning rate η;
2. run a pattern $\mathbf{x}^A$ of the training set and store the activations of all the units;
3. calculate $\Delta_i^{(L)A}$, and then back-propagate the error ;
4. compute the contributions to $\delta\omega_{ij}^{(l)}$ and $\delta\theta_i^{(l)}$;
5. update weights and thresholds.

Procedure iterated until a suitable convergence criterion is fulfilled

Learning strategies - Backpropagation

Several modifications have been proposed to improve the algorithm so that local minima are avoided.
One of the most successful is the introduction of a 'momentum' term:

$$\begin{aligned}
\delta\omega_{ij}^{(l)} &= -\eta \frac{\partial E}{\partial \omega_{ij}^{(l)}} + \alpha\ \delta\omega_{ij}^{(l)}(\mathrm{last}) \\
\delta\theta_{i}^{(l)} &= -\eta \frac{\partial E}{\partial \theta_{i}^{(l)}} + \alpha\ \delta\theta_{i}^{(l)}(\mathrm{last})
\end{aligned}$$

where "last" denotes the values of the $\delta\omega_{ij}^{(l)}$ and $\delta\theta_{i}^{(l)}$ used in previous updating of the weights and thresholds.
The parameter α ('momentum') must be a positive number smaller than 1.

GENETIC ALGORITHMS

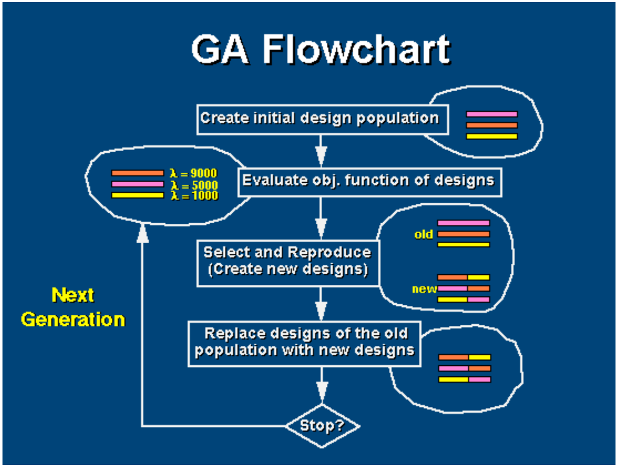
Learning strategies - Genetic algorithms

- They simultaneously work on populations of solutions, rather than tracing the progress of one point through the problem space. Check many regions of the parameter space at the same time.
- They only use payoff information directly associated with each investigated point. No outside knowledge such as the local gradient behaviour around the point is necessary.
- They have a built-in mix of stochastic elements applied under deterministic rules, which improves their behaviour in problems with many local extrema, without the serious performance loss that a purely random search would bring.

All the power of genetic algorithms lies in the repeated application of three basic operations onto successive generations of points in the problem space. These are

1. Selection
2. Crossover
3. Mutation

GA flowchart


GA Flowchart
Create initial design population
Evaluate obj. function of designs
λ = 9000
λ = 5000
λ = 1000
Select and Reproduce (Create new designs)
old
new
Replace designs of the old population with new designs
Stop?
Next Generation

Genetic Algorithms - An example

Maximisation of $f(x) = x^2$ on the interval $x \in [0, 31]$

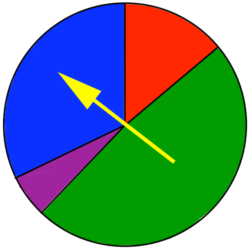
1. **Encode** our problem parameter x into a string, the *chromosome*, on which the GA can then operate. Possibility: binary encoding, $x = 1$ codes as 00001 and $x = 31$ as 11111.
2. Create at random the initial population with fixed number of individuals $i = 1, \ldots, N$. We take $N = 4$ for illustration. Fitness calculated with the function to maximise: $f(x) = x^2$

i	Genotype	Phenotype x_i	Fitness $f_i = f(x_i)$	$f_i / \sum f_i$
1	01101	13	169	0.14
2	11000	24	576	0.49
3	01000	8	64	0.06
4	10011	19	361	0.31

Genetic Algorithms - An example

Maximisation of $f(x) = x^2$ on the interval $x \in [0, 31]$

- *Selection*: individuals with higher fitness will have a larger chance of contributing offspring to the next generation.
  Example of selection method that achieves this is *roulette wheel selection*: the probability of an individual to be selected in proportional to its fitness



- Assume that in the four coin tosses of the *wheel* the following parents were chosen: 1, 2, 2 and 4.

Genetic Algorithms - An example

Maximisation of $f(x) = x^2$ on the interval $x \in [0, 31]$

- *Crossover* is necessary to obtain a child generation that is genetically different from the parents.
  The parents selected in the previous step are paired up randomly and a crossover site s within the chromosome is randomly chosen for each pair. The chromosomes of both parents are cut after that position and the ends are exchanged to form the chromosomes for the two children.

Breeding pair 1			Breeding pair 2		
i	before	after	i	before	after
1	0110|1	0110|0	2	11|000	11|011
2	1100|0	1100|1	4	10|011	10|000

The first child generation after selection and crossover:

i	Genotype	Phenotype x_i	Fitness $f_i = f(x_i)$	$f_i / \sum f_i$
5	01100	12	144	0.08
6	11001	25	625	0.36
7	11011	27	729	0.42
8	10000	16	256	0.14

The rapid increase of fitness over the very first few generations is a common feature of GAs.

Genetic Algorithms - An example

Maximisation of $f(x) = x^2$ on the interval $x \in [0, 31]$

- **Mutation**: With a given probability every bit in every chromosome is flipped from 1 to 0 or reverse after crossover
- Mutations allow to explore to whole available parameter space
- Over the span of several GA generations then, even a stagnated chromosome position can become reactivated by mutation.
- Several improvements of the basic concept exist: mutation which depend of the GA generation, on the value of the input patter, or even mutations of random size themselves

Genetic Algorithms - Why do they work

- The theoretical concept behind the success of GAs is the concept of patterns or *schemata* within the chromosomes
- Rather than operating on only N individuals in each generation, a GA works with a much higher number of schemata that partly match the actual chromosomes.
- A chromosome like 10110 matches 2^5 schemata, such as `**11*`, `***10` or `1*1*0`, where `*` stands as a wild card for either 1 or 0. Since fit chromosomes are handed down to the next generation more often than unfit ones, the number of copies n_S of a certain schema S associated with fit chromosomes will increase from one generation to the next:

  $$n_S(t+1) = n_S(t) \cdot \frac{\bar{f}(S)}{\bar{f}_{total}},$$

  where $\bar{f}(S)$ is the average fitness of all individuals whose chromosomes match schema S, and $\bar{f}_{total}$ is the average fitness of all individuals.
- If we assume that a certain schema approximately gives all matching chromosomes a constant fitness advantage c over the average

  $$\bar{f}(S) \equiv (1+c) \cdot \bar{f}_{total},$$

  we get an exponential growth in the number of this schema from one generation to the next:

  $$n_S(t) = n_S(0) \cdot (1+c)^t.$$

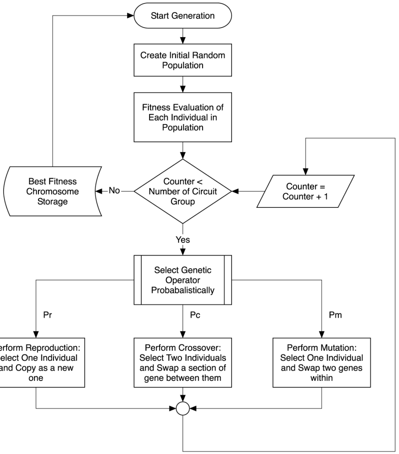
Genetic Algorithms - Why do they work

- Taking into account the effects that crossover and mutation may have, we need to define two measures on schemata:
  - The *defining length* δ is the distance between the furthest two fixed positions. In the examples above, we get $\delta = 1$ for **11* and ***10, and $\delta = 4$ for 1*1*0.
  - The *order* o of a schema is the number of fixed positions it contains. In the above example o is 2, 2 and 3 respectively.
- With these measures and L as the total length of a chromosome, we can now write

$$n_S(t+1) \geq n_S(t) \cdot \frac{\bar{f}(S)}{\bar{f}_{total}} \left[1 - \frac{\delta(S)}{L-1} - o(S) \cdot p_m \right].$$

- Final term is the effect of mutation on a schema. In a schema of order o, there is a probability of $\approx (1 - o \cdot p_{mut})$ that the schema survives mutation.
- Short, low-order schemata of high fitness are *building blocks* toward a solution of the problem.
- In a population of size N, approximately $\mathcal{O}(N^3)$ schemata are processed per generation.

Genetic Algorithms - Summary



Start Generation
Create Initial Random Population
Fitness Evaluation of Each Individual in Population
Best Fitness Chromosome Storage
No
Counter < Number of Circuit Group
Counter = Counter + 1
Yes
Select Genetic Operator Probabalistically
Pr
Pc
Pm
Perform Reproduction: Select One Individual and Copy as a new one
Perform Crossover: Select Two Individuals and Swap a section of gene between them
Perform Mutation: Select One Individual and Swap two genes within

SVN AND EXERCISES

The svn software

- Subversion (svn) is an open source version control system
- We will use in this course to share code
- Manual can be found at

  `http://svnbook.red-bean.com/`

- To check-out today's examples

```
svn co
svn+ssh://<utente-lcm>@pcteor1.mi.infn.it/home/rojo/nnet-code
```

- To check in modifications

```
svn ci -m ‘‘message explaining what I have done’’
```

- To update the working version of the repository

```
svn update
```

First exercises

- Check out the code of the course, compile with `make` and run

```
./nnet-course-example1
```

- This program trains a neural network on $\sin(x)$ using Genetic Algorithms
- Aim: to obtain the smallest possible χ^2 by modifying the neural network architecture and the parameters of GA minimization
- Check also the C++ example

```
cpp/gahelloworld
```

  which uses GA to reproduce a given target string
- Aim: change the target string to a more complex one and check that GA still work