Course on

Neural Networks

Abdallah Zemirline[1], Pascal Ballet[1], Lionel Marcé[1], Gilles Bernot[2], Franck Delaplace[2], Jean-Louis Giavitto[2], Olivier Michel[2], Jean-Marc Delosme[2], Patrick Amar[3], Roberto Incitti[4], Paul Bourgine[5], Christophe Godin[6], François Képès[7], Philippe Tracqui[8], Vic Norris[9], Janine Guespin[9], Maurice Demarty[9], Camille Ripoll[9]

[1] *EA2215, Département d'Informatique, Université de Bretagne Occidentale, Brest*
[2] *Laboratoire de Méthodes Informatiques, CNRS UMR 8042, Université d'Evry, 91025 Evry*
[3] *Laboratoire de Recherche en Informatique, Université Paris-Sud, Orsay*
[4] *Lacl, Université de Marne La Vallée*
[5] *CREA - Ecole Polytechnique*
[6] *UMR Cirad/Inra modélisation des plantes, TA40/PSII, Montpellier*
[7] *Atelier de Génomique Cognitive CNRS ESA8071/genopole, Evry*
[8] *Lab. TIMC-IMAG, Equipe DynaCell , CNRS UMR, 5525, Faculté de Médecine, La Tronche*
[9] *Laboratoire des Processus Intégratifs Cellulaires, UPRESA CNRS 6037, Faculté des Sciences & Techniques , Université de Rouen, 76821, Mont-Saint-Aignan*
France

The development of neural networks had initially as objective the modeling information processing and learning in the brain, in order to understand how a population of interconnected biological neurons performs a cerebral function. Now, neural networks are used in several practical applications, in various fields including computational molecular biology [23, 24], and the artificial neurons are quite remote from biological neurons.

1. Biological neural networks

A neuron [22] is a nervous cell having a cytoplasm body and several cytoplasm extensions (axons and dendrites) that allow it to dispatch (axons) and to receive (dendrites) signals. The exchanged information by two neurons is accomplished by means of electrical signals, which are the result of potassium-sodium ion exchanges. The electrical signal exchanges are made at the level of the synapses, which link the axons of neurons to the dendrites of other neurons. A neuron may have 1 000 to 10 000 synapses and can receive information from 1 000 other neurons. Besides, although the synapses are often constituted between axons of cells and dendrites of other cells, there are other types of synaptic junctions : between axon and axon, between dendrite and dendrite, between axon and cellular body. The human brain may contain until 10^{11} neurons.

The complexity of biological neural networks (BNNs) is very variable. There are some BNNs like the ganglions that are constituted of heaps of neurons, as there exist sophisticated BNNs like the complex BNNs of the neocortex. These ones are able to modify their functioning and even their structures as well as they are capable of computing, memorizing and learning. Memorizing and

learning of the BNNs are made by means of some modifications at the synaptic level. The synapses may modulate their activity, as exciting or inhibiting a neuron, and in this way to let possible the writing of an information in a memory area. In 1949, Hebb [2] made the hypothesis that the abilities of BNNs are the result of the self-organization of their connections : The efficiency of a synapse increases when the neurons that it connects are at the same time either all active or all inactive; otherwise the efficiency lessens.

2. Artificial neural networks

2.1. Neural network models

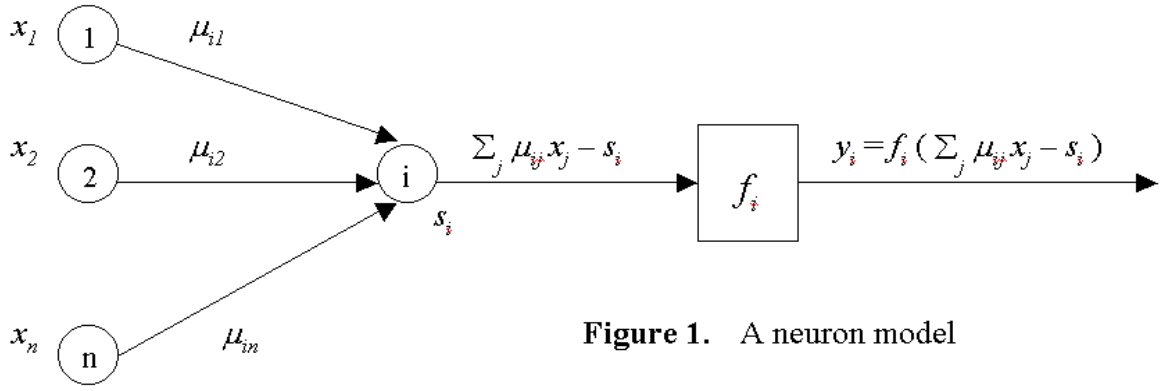


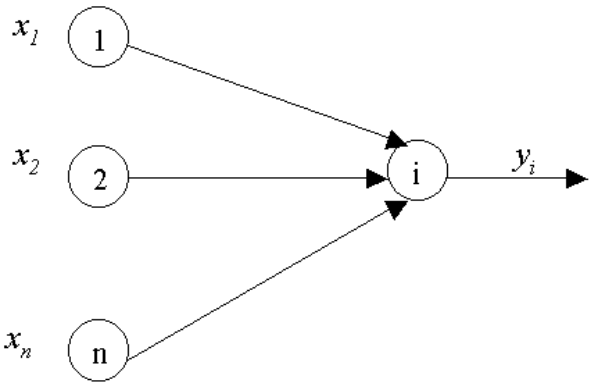
Figure 1. A neuron model

An *artificial neural network* (ANN) can be described as a set of interconnected units evolving in time and operating in parallel; the units represent axons and dendrites and each connection (*j*,*i*) from unit *j* to unit *i* has a *weight* μ_{ij} that modulates the influence of unit *j* on unit *i*. Thus, an ANN is a weight-directed graph in which to each node *i* are associated a *bias* or *threshold* s_i and a *transfer function* f_i, so that unit *i* will produce an output y_i of the form : $y_i = f_i (\sum_j \mu_{ij} x_j - s_i)$, where x_j is the *j*th input of this unit and $\sum_j \mu_{ij} x_j$ is the sum of all its weighted inputs. If this sum is greater than the threshold s_i, unit *i* is activated for producing the output y_i; otherwise unit *i* is in an inactive state (Figure 1). The parameters μ_{ij} and s_i can be adjusted so that the neural network produces some desired behavior. Namely, the neural network can be trained to achieve some particular job by adjusting the weight and bias parameters.

The transfer functions widely used are nonlinear, smooth, increasing and bounded such as sigmoid functions (so called from their “S” shape). However, sometimes the transfer function is linear like the identity function. When $f_i(x) = 1$ if $x > 0$ and $f_i(x) = 0$ otherwise, unit *i* is called a *threshold gate*. As threshold functions are discontinuous, they are often replaced by sigmoidal

transfer functions that are continuous and differentiable, such as $f(x) = arctan(x)$ and $f(x) = tanh(x)$, or by other transfer functions such as $f(x) = 1 / (1 + e^{-x})$.

One drawback of this neuron model appeared when it was used to describe what electro-chemical triggering phenomena takes place at the active cell membranes of biological neurons. It was noticed that the description of signal transformations in complicated neural networks needs an analysis computationally too heavy. Whereat T. Kohonen [13] suggested the following simple nonlinear dynamic model for a neuron (Figure 2) :



$$dy_i / dt = I_i - \gamma(y_i), \quad y_i \geq 0 \qquad (1)$$

Figure 2. A nonlinear dynamic neuron model

In Figure 2, the x_j and y_i are nonnegative scalar variables, the input activation I_i is some function of the x_j and of some internal parameters. The function $\gamma(y_i)$ is the leakage term, a nonlinear function of output activity. In order to guarantee good stability in feedback networks γ must be convex (i.e., its second derivative with respect to y_i must be positive). The leakage term $\gamma(y_i)$ takes in account all different losses and dead-time effects in the neuron, as a progressive function of activity.

2.2. Network architectures

Usually, three important architectures are considered for the ANNs : *layered* architecture, *feedforward* architecture and *recurrent* or *feedback* architecture. A *recurrent* architecture contains directed cycles; therefore, the signal paths can return to a same node. The feedback ANNs are difficult to implement. A *feedforward* architecture is devoid of directed cycles, thus the signal paths never return to a same node. A *layered* architecture is an architecture where the units are partitioned into classes, called *layers*, and where the connectivity patterns are defined between the classes.

Besides, the unit set is partitioned into *visible* units (those in contact with the external world such as *input* and *output units*) and *hidden* units. Often, the input units are grouped in an *input layer* and the output units in an *output layer*. A *hidden layer* is constituted of hidden units.

2.3. Three main categories of ANNs

It is customary to distinguish three categories of ANNs : *adaptive signal transfer networks*, *state transfer networks*, and *competitive-learning* or *self-organizing networks*.

The *signal transfer networks* have their output signals depending uniquely on input signals. These are often layered feedforward networks such as the *multilayer Perceptron* [3], the *Madaline* [4], the feedforward network in which learning is defined by means of an *error propagation algorithm* [5], and the *radial-basis-function networks* [6].

The *state transfer networks* are recurrent ANNs in which the feedbacks and nonlinearities are very strong so that the activity state quickly converges to one of its equilibrium points (attractors). Indeed, input information sets the initial activity state and once the network is in operation the output is fed back as the input until the network output will settle on one of its stable values. Typical representatives of these ANNs are the *Hopfield network* [7] and the *Boltzmann machine* [8].

The cells of the *competitive-learning* or *self-organizing networks*, which generally receive identical input information, compete in their activities by means of lateral interactions. Each cell or cell group is sensitized to a different domain of vectorial input signal values, and acts as a decoder of that domain [9, 10]. Besides, both of the *adaptive-resonance-theory models* of Grossberg and Carpenter [11,12] and the *Self-Organizing Maps* of T. Kohonen [14] belong of course to this category.

2.4. Phases of development of neural models

Three phases of development of models in ANN theory are distinguished : *memoryless* models, *adaptive* models and *plasticity-control* models

Memoryless Models : In this first modeling phase, which starts with the classical *McCulloch-Pitts* network [1], the transfer properties of the network were assumed fixed. And when feedback connections were added, such as in some interconnected networks [3] and also in some state transfer models [7, 8], only the relaxation of activity distributions was considered. There, the dynamic state equation is written as : $dA/dt = f(I, A)$; where signal activity A is a function of location, I is the external input acting on the same locations, and f is a general function of I and A, and of location.

Adaptive Models : These models take in account the *adaptation* and *memory* properties that result from parametric changes in the network. The equations, which describe the adaptive signal-transfer circuits, are : $dA/dt = f(I, A, M)$, $dM/dt = g(I, A, M)$; where : M denotes the set of system parameters (M may be a function of location and represent an adaptive bias), and f and g are general functions of I, A, and M . These equations were used in the first endeavors to model emergence of feature sensitive cells and elementary forms of self-organizing mappings.

Plasticity-Control Models : T. Kohonen [14, 15] was not convinced that a model with adaptive connectivity parameters is accurate enough to capture all aspects of self-organization, such as, for instance, the learning rate of a synaptic connection, which is called *plasticity* in neurophysiology. And in 1993, he [15] advanced the idea that the *plasticity* should be described and controlled by a third group of state variables called *P* and wrote the system equations as :

$dA/dt = f(I, A, M)$, $dM/dt = g(I, A, M, P)$, $dP/dt = h(I, A, M, P)$; where *f*, *g*, and *h* are general functions and where *P* does not take part in the control of activity *A*.

3. Learning and Evolution

A*daptation* refers to a control of parameters in order to optimize some performance measure, or to a behavioral modification that depends on experiences and that improves the performance of a system. In classical ANNs *adaptation* is called *learning* or also *training*. Besides, in *evolution*, *adaptation* is the adjusting of species to environment by natural selection or by behavioral change. Hence in *evolutionary artificial neural networks* (EANNs), which are a special class of ANNs, *adaptation* is called *evolution*. Thus, in ANNs *adaptation* takes two fundamental forms : *Learning* and *Evolution*.

3.1. Learning

Following the Hebb's assumption and in order that the ANNs may develop an associative memory, it is necessary that the efficiency of the connections, which link the artificial neurons, may be computed. Since the fifties, several rules appeared, especially the Perceptron rule [3] and the Widrow-Hoff learning rule [4]. These rules put the ANN on a supervised learning, which can be summarized as follows: After having presented to the input units what it must be memorized, the ANN answer is scanned. Since the correct answer is known then it is attempted to reduce the gap between these two answers by acting on the efficiencies of the connections that link the artificial neurons, more particularly on the thresholds s_i and the weights μ_{ij}. When these efficiencies stabilize, the learning phase ends.

More generally, Learning in ANNs can roughly be partitioned onto *supervised*, *unsupervised*, and *reinforcement learning* :

Supervised learning makes a direct comparison between the current output of an ANN and the correct output, which is known. This comparison is often made by means of a minimization of an error function such as the total mean square error between the actual output and the desired output. In order to minimize this error, a gradient descent-based optimization algorithm such as backpropagation [4] can then be used to adjust connection weights in the ANN interactively. *Reinforcement learning* is a special case of supervised learning where the only known information

is whether or not the current output is correct (the desired output is unknown). In this learning mode adaptive changes of the parameters due to reward or punishment depend on the final outcome of a whole sequence of behaviour.

Unsupervised learning works only on the correlations among input data; there is not any other information for learning. It is without a priory knowledge about the classification of samples.

Sect. 3.1.2. describes the Perceptron learning algorithm. Sect. 3.1.3. is devoted to *competitive-learning* networks and to an *unsupervised learning* which is used to get a representation of high-dimensional nonlinearly related data items in a illustrative two-dimensional display [14].

Finally, notice that the essence of a learning algorithm is certainly its learning rule (i.e., for example, a weight-updating rule which determines how the signals should modify the adaptive connection input weights or other parameters of the neurons in learning) and that its correctness needs to make clear what the ANN submitted to learning is supposed to do (for instance, its function is associative memory or detection of elementary patterns).

3.1.1. Some Learning Laws

3.1.1.1 Hebb's Law

Consider first the simplest classical learning law for neurons like the one defined in Figure 1. If the ANNs made of such neurons are supposed to reflect simple *memory effects,* especially those of *associative* or *content-addressable memory,* a model law that describes changes in the connections is based on *Hebb's hypothesis* [2] :

> "When an axon of cell *A* is near enough to excite a cell *B* and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells, such that *A's* efficiency, as one of the cells firing *B,* is increased"

This means that the weight μ_{ij} is varying according to $d\mu_{ij} / dt = \alpha\, y_i\, x_j$ (2) ;
where x_j is the jth input (the presynaptic "activity") of unit i, y_i is the output of unit i (the postsynaptic "activity"), and α is a scalar parameter named *learning-rate factor*. This law, generally called *Hebb's law,* has given rise to some elementary associative memory models, named *correlation matrix memories* [16-18]. In vector form, it can be written as :

$dm_i / dt = \alpha\, y_i\, x$ (2') ; where $m_i = (\mu_{i1}, \ldots, \mu_{in})^T$; $y_i = \sum_j \mu_{ij} x_j = m_i^T x = x^T m_i$; $x^T = (x_1, \ldots, x_n)$ and n the number of inputs of each unit.

Notice that with this law the associative memory function is omitted. Moreover, as *feature-sensitive cells* have central roles in the classification functions both at the input layers of the neural networks,

as weIl as inside them, some modifications of Hebb's law were considered : the *perceptron* learning law, the *Riccati* learning law, and the *principal-component-analyzer* (*PCA*) law.

3.1.1.2. Perceptron Learning Law

The perceptron learning rule is a modified form of Hebb's learning law. It was proposed by F. Rosenblatt [3] in the late 1950s. It is the following : $dm_i / dt = \alpha (y_i^c - y_i) x$ (3) ;

where y_i^c is the desired output (i.e., the correct output).

This rule is also known as back-propagation rule, LMS (least mean squares) rule, or as delta rule.

3.1.1.3. The Widrow-Hoff Learning Law

This law, which stems from Widrow [4] , was introduced for multilayer feedforward networks. It can be also written as (3) and where the least mean of square error criterion is applied and the optimization is performed by Robbins-Monro stochastic approximation.

3.1.1.4. The Riccati-Type Learning Law

A major revision [14] made to Hebb's law introduces a *scalar-valued plasticity-control function P* that may depend on many factors (activities, diffuse chemical control, etc …) and that shall have a time-dependent *sampling effect* on the *learning* of the signals x_j. On the other hand, it was assumed that the weights μ_{ij} are affected proportionally to x_j. In this way, the first term of the learning equation is written as $P\, x_j$, where *P* is a general functional that describes the effect of activity in the surroundings of neuron *i*.

The second major revision is inclusion of an "active forgetting" term that guarantees that the μ_{ij} remains finite. This involves the introduction of a scalar-valued *forgetting rate functional Q*, which is some function of synaptic activities of neuron *i*. Therefore, the equation, which describes a kind of "active learning and forgetting" and where the plasticity control *P* affects the total learning rate, is the following : $d\mu_{ij} / dt = P\,(x_j - Q\,\mu_{ij})$. In this equation, *P* can be seen as describing extracellular effects and *Q* intracellular effects. Moreover, it seems proper to assume that the "active forgetting" effect at synapse *j* is proportional to $\sum_k \mu_{ik} x_k$, where the sum extends over the whole cell, including synapse *j* itself. Then the latter equation can be written as the Ricatti-type equation :

$d\mu_{ij} / dt = P\,(x_j - \mu_{ij} \sum_k \mu_{ik} x_k)$; or in vector form with $\alpha = P$ and $\beta = P\,Q$ as

$$dm_i / dt = \alpha\, x - \beta\, m_i m_i^T x \qquad (4) \quad .$$

3.1.1.5. The PCA-Type Learning Law

This learning law, which was introduced by E. Oja [19], is analogous to (4), except that its right-hand side is multiplied by the expression $y_i = \sum_j \mu_{ij} x_j = x^T m_i$.

The differential equation of this law is the following : $dm_i / dt = \alpha y_i x - \beta y_i^2 m_i$ or

$dm_i / dt = \alpha x^T m_i x - \beta (m_i^T x x^T m_i) m_i$ (5) .

3.1.2. Perceptron Learning Algorithm [3] [5] [20]

The perceptron learning algorithm obeys perceptron learning rule (3). It applies to feedforward neural networks where the neuron model is the one of Figure 1. Training patterns x are presented to the neural network; the output y_i is computed. Then the weights μ_{ij} are modified according to :

$$m_i(t + 1) = m_i(t) + \alpha (y_i^c - y_i) x \quad \text{where } m_i = (\mu_{i1}, \ldots, \mu_{in})^T.$$

Hereafter, the single-layer perceptron learning algorithm and the back-propagation perceptron learning algorithm are described.

1) A single-layer perceptron neural network comprises one or more artificial neuron in parallel. Like in Figure 1 each neuron has n inputs and one output. The perceptron learning algorithm for a single-layer perceptron neural network is the following :

(0) Initialize the weights μ_j and threshold s to small random numbers; $t = 0$;

(1) Present an input vector $x = (x_1, \ldots, x_n)^T = x(t)$ and the desired output y^c, (where n is the number of input units), and calculate the output $y = y(t)$ according to

$y = f(\sum_j \mu_j x_j - s)$, where f is a given transfer function

(f can be the sigmoid function : $f(x) = 1 / (1 + e^{-x})$);

(2) Update the weights μ_j according to : $\mu_j(t + 1) = \mu_j(t) + \alpha (y^c(t) - y(t)) x_j(t)$

$j = 1, \ldots, n$; where $0.0 < \alpha < 1.0$) ; $t = t + 1$;

(3) Repeat steps (1) and (2) until the iteration error is less than a user-specified error threshold or a predetermined number of iterations have been completed.

2) Multi-layer Perceptron Learning Algorithm, or Back-Propagation Learning Algorithm :

The algorithm for multi-layer perceptron learning is based on the back-propagation rule (3) and on a gradient descent in error space. The error is defined as $E = \sum_p E_p$ (6)

where $E_p = (\sum_i (y_i^c - y_i)^2) / 2$ (7) where y_i is the actual output and y_i^c is the desired output and where the sum is over the output units of the network.

A change of weights can be made according to the gradient of the error : $\Delta\mu = - \alpha \nabla E$ (8)

where α is a constant scaling and ∇ is the gradient operator. The weight change for the connection from unit j to unit i, of this error gradient can be written as : $\Delta\mu_{ij} = - \alpha \nabla_{ij} E = - \partial E / \partial \mu_{ij}$ (9)

But $\partial E / \partial \mu_{ij} = (\partial E / \partial y_j)\,(\partial y_j / \partial z)\,(\partial z / \partial \mu_{ij})$ (10) with $z = \sum_k \mu_{kj}\, y_k$. Hence

$\partial z / \partial \mu_{ij} = \partial \sum_k \mu_{kj}\, y_k / \partial \mu_{ij} = \sum_k \partial\, (\mu_{kj}\, y_k) / \partial \mu_{ij} = \sum_k ((\partial \mu_{kj} / \partial \mu_{ij})\; y_k + \mu_{kj}\, (\partial y_k / \partial \mu_{ij}))$ (11) .

Examining the first partial derivative, notice that $\partial \mu_{kj} / \partial \mu_{ij}$ is zero unless $k = i$. And examining the second partial derivative $\partial y_k / \partial \mu_{ij}$ for observing that if μ_{kj} is not zero then there exists a connection from unit k to unit j, which implies that $\partial y_k / \partial \mu_{ij}$ must be zero, otherwise the network would not be feedforward. Therefore, we get from (11) : $\partial z / \partial \mu_{ij} = y_i$ (12)

We now consider the middle partial derivative of (10) : $\partial y_j / \partial z$. Since $y_j = f(z)$ then $f(z) = 1 / (1 + e^{-z})$ would imply that $\partial y_j / \partial z = \partial\, (1 + e^{-z})^{-1} / \partial z = (1 + e^{-z})^{-2}\, e^{-z}$

$= (1 - y_j)\; y_j$. In this way : $\partial y_j / \partial z = (1 - y_j)\; y_j$ (13)

Now, return to the first derivative of (10) : $\partial E / \partial y_j$ And recall that $E = \sum_p E_p$ and

$E_p = (\sum_i (y_i^c - y_i)^2) / 2$ where the sum is over the output units of the network.

Two cases can be distinguished : j is an output unit ; j is not an output unit.

- If j is an output unit then the derivative $\partial E / \partial y_j$ can be computed as :

$\partial E / \partial y_j = \partial (\sum_i (y_i^c - y_i)^2) / 2 / \partial y_j = \sum_i (y_i^c - y_i)\, \partial (y_i^c - y_i) / \partial y_j = -(y_j^c - y_j)$ (14)

- If j is not an output unit then we need to rely on the chain rule, applied over the units k connected to unit j :

$\partial E / \partial y_j = \sum_k (\partial E / \partial y_k)\; (\partial y_k / \partial z)\, \mu_{kj}$ (15)

where : $\partial y_k / \partial z$ is given by (13) and $\partial E / \partial y_k$ is computed recursively.

We are now in position to describe the back-propagation learning algorithm :

(0) Initialize the weights μ_{ij} and threshold s to small random values; $t = 0$;

Present an input vector $x = (x_1 , \ldots , x_n)^T$ and a target output y^c

where n is the number of input units and m the number of output units y^c;

(x and y^c represent the patterns to be associated) ;

(1) Calculate the actual output : Each layer calculates $y^k = f(\sum_j \mu_{ij} x_j - s)$;

(where f is defined by : $f(z) = 1 / (1 + e^{-z})$)

This is then passes this to the next layer as an input. The final layer outputs value y^v .

(2) Adapts weights : Starting from the output y^v and working backwards, do

$\mu_{ij}(t + 1) = \mu_{ij}(t) + \alpha y^v E_j$; where E_j is an error term corresponding to the input x_j of node j ;

such that : for output units : $E_j = \sigma y^v (1 - y^v) (y^c - y^v)$

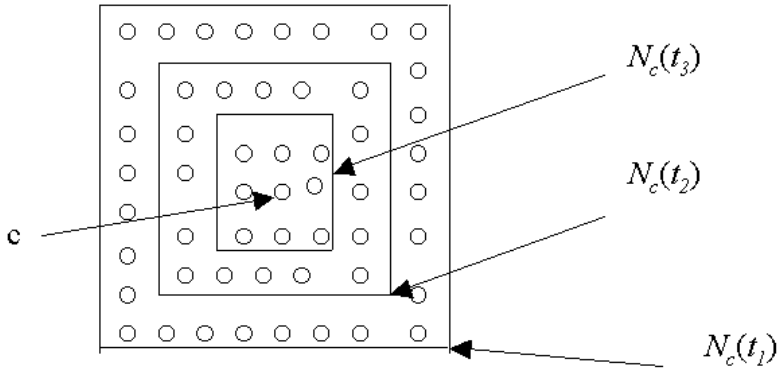
for hidden units : $E_j = \sigma y^v (1 - y^v) \sum_{k}{}_{1}^{q} E_k \mu_{kj}$ where the sum is over all the q nodes in the layer above node j .

(σ is the steepness parameter in the sigmoidal function)

3.1.3. The Self-Organizing Map (SOM) algorithm [14] [21]

The SOM algorithm, which stems from Kohonen [14], deals with the competitive learning and self-organizing networks. It operates as a nonparametric *regression* which involves fitting a number of reference vectors to the distribution of vectorial input samples (In *regression* some simple mathematical function is fitted to the distribution of sample values). The reference vectors m_i are considered here to approximate the probability distribution of the input signals x and are also used to define the nodes of a kind of "hypothetical elastic network". Indeed, the distribution of the vectors m_i should reflect the probability distribution of the input signals x, which is not given explicitly but only through the sample of vectors x.

Given an ANN constituted of N neurons, where to every node (neuron) i , $i = 1, \ldots, N$, is associated a weight-vector $m_i = (\mu_{i1}, \ldots, \mu_{in})^T \in \mathbb{R}^n$. Between the units of the ANN there exists a set C, possibly empty, $C \subseteq \{1, \ldots, N\}^2$ of neighborhood connections supposed unweighted and symetric. Besides, from the connection set C, construct a two-dimensional grid G, having N nodes, so that two nodes i, j are neighbors in G if and only if $(i,j) \in C$. Let $x = (x_1, \ldots, x_n)^T \in \mathbb{R}^n$ be an input vector supposed connected to each neuron i, via the weight-vector $m_i = (\mu_{i1}, \ldots, \mu_{in})^T$. Vector x is compared with all the m_i in some metric, the euclidean metric for instance, in order to determine a node $c \in \{1, \ldots, N\}$ such that $\|x - m_c\| = \min\{\|x - m_i\| ; i = 1, \ldots, N\}$; unit c is called the *winner*. In this grid, a decaying topological neighborhood $N_c = \{ i \in \{1, \ldots, N\}; (c,i) \in C\}$ of node c is defined such that $c \in N_c(t)$ for every t and $N_c(t + 1)$ is strictly contained in $N_c(t)$; where t = 0, 1, 2, … is the discrete time coordinate (see Figure 3). During learning at time t, those nodes of the grid that are in N_c(t) will activate each other to learn something from the same input x.



An example of topological neighborhood

Figure 3.

Indeed, the following learning process is proposed in [14] :

$m_i(0)$ is arbitrary; and for t = 0, 1, 2, …

$$m_i(t+1) = m_i(t) + h_{ci}(t)\,(\,x(t) - m_i(t)\,) \qquad (16)$$

where $h_{ci}(t)$ must $\to 0$ when $t \to \infty$; otherwise the sequence $(m_i(t))_{t \geq 0}$ does not converge. The form of $h_{ci}(t)$ and its average width characterize the "stiffness" of the elastic surface defined by the points m_i of $\mathbb{R}^n$.

Frequently, $h_{ci}(t)$ is taken equal to 0 if $i \notin N_c(t)$ and $h_{ci}(t) = \alpha(t)$ if $i \in N_c(t)$. $\alpha(t)$ is called a *learning-rate factor* and is such that $0 < \alpha(t) < 1$. Furthermore, both $\alpha(t)$ and the radius of $N_c(t)$ are decreasing monotonically in time.

Another choice for $h_{ci}(t)$ which widely occurs is the following :

$$h_{ci}(t) = \alpha(t)\, exp\,(\, - \|r_c - r_i\|^2 / 2\,\sigma^2(t)\,) \qquad (17)$$

where : $\alpha(t)$ is another valued learning rate factor ;

r_c and r_i *belong to* $\mathbb{R}^2$ and are respectively the location vectors of nodes c and i in the grid ;

$\sigma(t)$ denotes the width of $N_c(t)$.

The <u>self-organizing feature map algorithm</u> is the following, where *Nit* is a predetermined number of iterations to be completed :

(0) Initialize the ANN to contain N units. Each unit i has n entries and an associated reference vector $m_i = (\mu_{i1}, \ldots, \mu_{in})^T \in \mathbb{R}^n$ chosen randomly ;

Initialize the connection set C to form a rectangular or a squared grid G ; $t = 0$;

(1) While ($t <$ *Nit*) do

(1.1) Generate at random an input signal $x \in \mathbb{R}^n$ according to a continuous probability density function $p(\xi)$, $\xi \in \mathbb{R}^n$;

(1.2) Determine a unit $c \in \{1, \ldots, N\}$ such that

$\|x - m_c\| = \min\{\|x - m_i\| ; \ i = 1, \ldots, N\}$;

and consider the topological neighborhood $N_c(t)$;

(1.3) Adapt each unit $i \in N_c(t)$ according to (16) and (17) ; $t = t + 1$;

End while

3.2. Evolution

Evolutionary artificial neural networks (EANNs) denote a special class of ANNs, where another form of adaptation, called *evolution* and distinct from learning, takes a prominent part. This evolutionary approach of adaptation applies *evolutionary algorithms* to ANNs for evolving weight training, evolution of architectures, evolution of learning rules, evolution of input features, etc.

3.2.1. Evolutionary algorithms

An *evolutionary algorithm* (EA) refers to a population-based stochastic search algorithm inspired by natural evolution. Three mechanisms drive natural evolution (*reproduction*, *mutation* and *selection*) by acting on the chromosomes containing the genetic information of the individual (the genotype), rather than on the individual itself (the phenotype) : By the *reproduction* mechanism new individuals are introduced into a population, these offspring chromosomes inherit from their both parents a mixture of genetic information (*crossover*). The *mutation* process brings small changes into the inherited chromosomes. And the *selection* mechanism allows only the fittest individuals (the best adapted to their environment) to survive and reproduce.

To solve a problem by means of an EA makes use of a metaphor of natural evolution : All the possible solutions constitute a population living in an environment that is the problem itself. The phenotype of each individual (each candidate solution) is encoded in some manner into its genome (genotype). The adaptability of each individual is measured by means of a *fitness function*. And the natural evolutionary mechanisms are modeled by appropriate genetic operators. Starting from an initial population and by applying genetic operators to introduce progressively “niece genetic material” into the successive populations, an EA produces step by step better solutions to the problem.

The EAs comprise several types : evolution strategies [25, 26], evolutionary programming [27, 28, 29], and genetic algoritms [30, 31]. All proceed as follows :

(0) t = 0; Generate the initial population G(0) at random;

(1) While (termination criterion is not satisfied) do

(1.1) Evaluate each individual of G(t);

(1.2) From G(t) select parents P(t) based on their fitness in G(t);

(1.3) Apply genetic operators to P(t) to generate offspring which constitute G(t + 1);

(1.4) t = t + 1;

End while.

3.2.2. The Evolution of Connection Weights

Most learning algorithms, such as backpropagation [5], are based on gradient descent. This use of gradient descent let these algorithms have drawbacks : They are often incapable of finding a global minimum of the error function and get trapped in local minima. One way to overcome these shortcomings is to formulate the training process as the evolution of connection weights in the environment defined by the architecture and the associated learning rule. Indeed, EAs can be used in the evolution to find a near-optimal set of connection weights globally. Unlike the case in gradient-descent-based learning algorithms, the fitness (or error) function of an ANN does not have to be differentiable or even continuous.

The evolutionary approach to weight training in EANNs comprises two phases.

The first phase deals with the choice of a representation of connection weights, either the binary representation or the real-number representation. In a binary representation, each connection weight is represented by a number of bits with a given length; then the concatenation of all the connection weights of the network encodes the ANN in the chromosome. In a real number representation, each connection weight is represented by a real number; in this way each individual (i.e., ANN) in an evolving population is encoded by a real vector.

The second phase is the evolutionary process simulated by an EA, in which genetic operators such as crossover and mutation have to be decided in conjunction with the representation scheme. The evolution stops when the fitness is greater than a predefined value (i.e., the training error is smaller than a certain value) or the population has converged.

A typical cycle of the evolution of connection weights is the following [32] :

(1) Decode each individual (genotype) in the current generation into a set of connection weights and construct with the weights a corresponding ANN.

(2) Evaluate each ANN by computing its total mean square error between actual and target outputs. The fitness of an individual is determined by the error. The higher the error, the lower the fitness. The optimal mapping from the error to the fitness is problem

dependent. A regularization term may be included in the fitness function to penalize large weights.

(3) Select parents for reproduction based on their fitness.

(4) Apply genetic operators, such as crossover and/or mutation, to parents to generate offspring, which form the next generation.

3.2.3. The Evolution of Architectures

The architecture of an ANN includes its topological structure (connectivity, and the transfer function of each node in the ANN). Architecture design is crucial in the successful application of ANNs because the architecture has significant impact on a network's information processing capabilities.

Like in the evolution of connection weights, two major phases involved in the evolution of architectures are the genotype representation scheme of architectures and the EA used to evolve ANN architectures. Encoding an ANN architecture implies deciding how much information about this architecture should be encoded in the chromosome. At one extreme, all the details, i.e., every connection and node of an architecture can be specified by the chromosome; this kind of representation scheme is called *direct encoding*. At the other extreme, only the most important parameters of an architecture, such as the number of hidden layers and hidden nodes in each layer are encoded; more details about the architecture are left to the training process to decide; this kind of representation scheme is called *indirect encoding*. The indirect encoding is used in order to reduce the length of the genotypical representation of architectures.

After a representation scheme has been chosen, the evolution of architectures can progress according to the cycle shown hereafter; the cycle stops when a satisfactory ANN is found [32].

(1) Decode each individual in the current generation into an architecture. If the indirect encoding scheme is used, further detail of the architecture is specified by some developmental rules or a training process.

(2) Train each ANN with the decoded architecture by a predefined learning rule (some parameters of the learning rule could be evolved during training) starting from different sets of random initial connection weights and, if any, learning rule parameters.

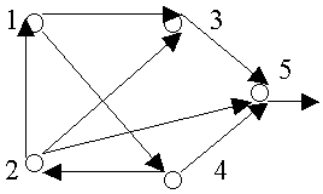
(3) Compute the fitness of each individual (encoded architecture) according to the above training result and other performance criteria such as the complexity of the architecture.

(4) Select parents from the current generation based on their fitness.

(5) Apply search operators to the parents and generate offspring which form the next

generation.

An example of the direct encoding of a feedback ANN is the following (Figure 4) :



0 0 1 1 0
1 0 1 0 1
0 0 0 0 1
0 1 0 0 1
0 0 0 0 0

Figure 4.

00110 10101 00001 01001 00000

architecture | connectivity matrix | binary representation

3.2.4. The Evolution of Learning Rules

Designing an efficient learning rule is very difficult when there is little prior knowledge about the ANN's architecture, which is often the case in practice. Besides, what is often expected from an ANN is its ability to adjust its learning rule adaptively according to the task to be performed and also to its architecture. These two reasons, and certainly several others, let the evolution of learning rules be introduced into ANNs in order to learn their learning rules.

But, as the evolution of learning rules has to work on the dynamic behavior of an ANN, then one key issue is how to encode the dynamic behavior of a learning rule into static chromosomes. The answer to this requires the two following assumptions :

(i) Weight-updating depends only on local information such as the current connection weight, the activation of the input node, the activation of the output node, etc;

(ii) The learning rule in an ANN is the same for all its connections.

Thus, a learning rule can be expressed by the function [9'] :

$$\Delta w(t) = \sum_{k=1}^{n} \sum_{i_1, i_2, \ldots, i_k=1}^{n} \left(\theta_{i_1, i_2, \ldots, i_k} \prod_{j=1}^{k} x_{i_j}(t-1) \right) \qquad (18)$$

where t is the time, $\Delta w(t)$ is the weight change, x_1 , x_2 , … , x_n are local variables, and the θ's are real-valued coefficients witch will be determined by evolution.

In this way, the evolution of learning rules amounts to the evolution of real-valued vectors of θ's.

On the other hand, the evolution of learning rules raises three questions [32] :

(i) determination of a subset of terms described in Eq. (18) ;

(ii) representation of their coefficients as chromosomes;

(iii) the EA used to evolve these chromosomes.

The answers to these issues lead to the following cycle of the evolution of learning rules [32] :

(1) Decode each individual in the current generation into a learning rule.

(2) Construct a set of ANNs with randomly generated architectures and initial

connection weights, and train them using the decoded learning rule.

(3) Compute the fitness of each individual (encoded learning rule) according to the average training result.

(4) Select parents from the current generation according to their fitness.

(5) Apply genetic operators to the parents and generate offspring which form the next generation.

3.2.5. Conclusion

Thus, evolution can be used in ANNs at several levels. The evolution of connection weights is quite competitive with regard to the gradient-based training algorithms. It can be also used to find quickly an efficient architecture as well as an efficient learning rule according to some architecture and to the task to be performed.

Furthermore, as it was noticed in [32], in many practical problems, the possible inputs to an ANN can be quite large. (There may be some redundancy among different inputs; a large number of inputs to an ANN increase its size and thus require more training data and longer training times). Preprocessing is often needed to reduce the number of inputs to an ANN. Given a large set of potential inputs, finding a subset, which has the fewest number of features but the performance of the ANN using this subset is no worse than that of the ANN using the whole input set, is not trivial. However, this problem can be implemented using a binary chromosome whose length is the same as the total number of input features; each bit in the chromosome corresponds to a feature : "1" indicates presence of a feature, while "0" indicates absence of the feature. The evaluation of an individual is carried out by training an ANN with these inputs and using the result to calculate its fitness value.

References

[1] W.S. McCulloch, W.A. Pitts, Bull. Math. Biophys. 5, 115, (1943).

[2] D. Hebb, *Organization of Behaviour*, Wiley, N.Y. (1949).

[3] F. Rosenblatt, *Principles of Neurodynamics : Perceptrons and the Theory of the Brain Mechanisms*, Spartan Books, Washington D.C. (1961).

[4] B. Widrow, In *Self-Organizing Systems 1962*, ed. by M. Yovits, G. Jacobi, G. Goldstein, Spartan Books, Washington D.C., 435, (1962).

[5] D.E. Rumelhart, G.E. Hinton, R.J. Williams, *Learning internal representations by error propagation*, in Parallel Distributed Processing, D. E. Rumelhart and J.L. McClelland eds, Vol. 1, *chap. 8*, MIT Press, Cambridge, MA, 318-362, (1986).

[6] T. Poggio, F. Girosi, Science 247, 978, (1982).

[7] J. Hopfield, Proc. Natl. Acad. Sci. USA 79, 2554, (1982).

[8] D. Ackley, G. Hinton, T. Sejnowski, Cognitive Science 9, p 147, (1985).

[9] R. Didday, Math. Biosci. 30, 169, (1976).

[10] S. Amari, M.A. Arbib, In Systems Neuroscience, ed. by J. Metzler, Academic N.Y., p 119, (1977)

[11] G. Carpenter, S. Grossberg, Computer 21, 77, (1988).

[12] G. Carpenter, S. Grossberg, Neural Networks 3, 129, (1990).

[13] T. Kohonen, Neural Networks 1, 3, (1988).

[14] T. Kohonen, *Self-Organizing Maps*, 2nd ed. Springer-Verlag, (1997).

[15] T. Kohonen, Neural Networks 6, 895, (1993).

[16] T. Kohonen, IEEE Trans. C-21, 353, (1972).

[17] K. Nakano, J. Nagumo, in Advance Papers, 2nd Int. Joint Conf. On Artificial Intelligence (The British Computer Society, London, UK 1971) p 101.

[18] J. Anderson, Math. Biosci. 14, 197, (1972).

[19] E. Oja, J. Math. Biol. 15, 267, (1982).

[20] F. Corbett, *Web Applets for Interactive Tutorials on Artificial Neural Learning*, Computer Engineering, University of Manitoba, Canada.

[21] B. Fritzke, *Some competitive Learning methods, Self-organizing Feature Map,* http://www.neuroinformatik.ruhr-uni-bochum.de

[22] S. W. Kuffler, J. C. Nicholls, *From Neuron to Brain : A cellular approach to the Function of the Nervous System,* Sinauer Associates Inc. Publishers, (1976).

[23] P. Baldi, S. Brunak, *Bioinformatic, The machine learning approach,* MIT Press, (1998).

[24] J. M. Renders, *Algorithmes génétiques et réseaux de neurones*, Hermes, (1995).

[25] H.-P. Schwefel, *Numerical Optimization of Computer Models*, J. Wiley & Sons, (1981).

[26] H.-P. Schwefel, *Evolution and Optimum Seeking*, J. Wiley & Sons, (1995).

[27] L. J. Fogel, A. J. Owens, M. J. Walsh, *Artificial Intelligence Through Simulated Evolution*, J. Wiley & Sons, (1966).

[28] D. B. Fogel, *System Identification Through Simulated Evolution: A Machine Learning Approach to Modeling*, Ginn Press, (1991).

[29] D. B. Fogel, *Evolutionary Computation: Towards a New Philosophy of Machine Intelligence*, IEEE Press, (1995).

[30] J. H. Holland, *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, (1975).

[31] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, (1989).

[32] X. Yao, *Evolving Artificial Neural Networks*, Proceedings of the IEEE, 87 (9), 1423-1447, (1999).

[33] X. Yao, *Evolutionary artificial neural networks*, in Encyclopedia of Computer Science and Technology (A. Kent and J. G. Williams, eds.), vol. 33, 137-170, NY 10016 : M. Dekker Inc., (1995).