Genetic Algorithms for Neural Network Training on Transputers

Bernhard Ömer

May 24, 1995

Supervisor: Dr. Graham M. Megson

Department of Computing Science

University of Newcastle upon Tyne

Abstract

The use of both, genetic algorithms and artificial neural networks, was originally motivated by the astonishing success of these concepts in their biological counterparts. Despite their totally different approaches, both can merely be seen as optimisation methods which are used in a wide range of applications, where traditional methods often prove to be unsatisfactory.

This project deals about how genetic methods can be used as training strategies for neural networks and how they can be implemented on a distributed memory system. The main point of interest is, whether they provide a reasonable alternative to the mainstream backpropagation algorithm, because of their inherently wide scope for parallelism.

To answer this questions, both algorithms have been tested on a variety of – mainly high nonlinear – problems, using a backpropagation as well as a serial and parallel genetic version for comparative performance measurements.

Moreover, a combined algorithm with both, genetic and backpropagation features has been developed and found out to be better suited for a distributed memory system than the corresponding uncombined versions.

Contents

1 Introduction

1.1 Motivation

Since the very first days of artificial intelligence (AI) in the forties and fifties of our century, there have been two main approaches on how to model and simulate intelligent behaviour. While the symbolic approach, as favoured e.g. by Alan Turing, saw intelligence a process relating discrete concepts and predicates according to certain rules, the connectivistic approach, as supported by e.g. Warren McCulloch and Walter Pitts claimed that the connection and interaction of many small and simple units could show intelligent behaviour; a theory that is strongly encouraged by the fact, that this concept has already shown itself extraordinary successful; in the neural system of the human brain.

The evolutionary principle of mutation and surviving of the fittest, first formulated by Charles Darwin, has also proven to be obviously a rather successful one. Despite of the discovery of the genetic encoding in DNA-strings and the cellular reproduction mechanism, it took rather long, until scientists like e.g. John Holland and Davis Goldberg took up the idea to use the same principle as an optimisation algorithm in computers.

While both methods didn't in fact come up to the high expectations that their biological counterparts might suggest, both have left behind their image of rather academic research and play tools and are nowadays generally accepted and used in a wide range of applications, where traditional methods often prove to be unsatisfactory.

1.2 Principles of Neuronal Networks

In the most general case, neural networks consist of an (often very high) number of neurones, each of which has a number of inputs which are mapped via a relatively simple function to its output. Networks differ in the way their neurones are interconnected (topology), in the way the output of a neurone determined out of its inputs (propagation function) and in their temporal behaviour (synchronous, asynchronous or continuous).

While the temporal behaviour is normally determined by the simulation hard- and software used and the topology remains very often unchanged, the propagation function is associated with a set of variable parameters witch refer to the relative importance of the different inputs (weights) or to describe a threshold-value of the output (bias).

The most striking difference between neural networks and traditional programming is, that neural networks are in fact not programmed at all, but are able to "learn" by example, thus extracting and generalising features of a presented training set and correlating them to the desired output. After a certain training period, the net should be able to produce the right output also for new input values, which are nor part of the training set.

This learning process is accomplished by a training algorithm, which successively changes the parameters (i.e. the weights and the bias) of each neurone until the desired result, typically expressed by the maximum distance between the actual and the training output, is achieved. Those algorithms cant be subdivided into two major groups according to the data used to update the neurone parameters. Local algorithms (e.g. Perceptron Learn Algorithm (PLA), Backpropagation) restrict themselves to the local data at the in- and outputs of each neurone, while global methods (e.g. Simulated Annealing) also use overall data (e.g. statistical information).

The main application areas of neural networks are pattern recognition, picture and speech processing, artificial intelligence and robotics. As a scientific concept; they also play a role in other disciplines as e.g. theoretical physics and chaos theory.

1.3 Principles of Genetic Algorithms

As neural networks, genetic algorithms also rely one a specific representation of the problem to solve, but instead of the rather restricting network-parameters, any representation which can be expressed as a fixed length string (genotype) over a finite (typically the binary) alphabet can be used. The interpretation (i.e. the phenotype) of a strings (genotype) is of no importance to the algorithm, no matter whether they represent the design parameters of a jet, the strategy in a cooperative game or, as for this project, the weights and biases of a neural net.

The genetic algorithm will generate a (generally very high) number of those chromosome strings at random, each representing an individual in the initial (or parent) population, to which the evolutionary principles of selection and mutation are applied. For the selection mechanism, the user has to provide means to determine the relative fitness of the individuals. This can be done by comparing two individuals and deciding which one is better (tournament selection, rank selection) or by providing an fitness or error function which allows to classify each individual in relation to the average fitness or error of the population (normalised fitness). The algorithm will then favour individual with higher fitness (lower error) to be selected to be propagated into the next generation. Typically, this evaluation process consumes most of the execution time, no matter whether the fitness is determined by calculation or by experiment.

A new generation is then produced by applying genetic operators to the selected individuals. The basic operators are mutation, where one or more digits of the chromosome string are changed, and crossover, where two strings are cut and crosswise recombined to form two new strings, which contain features of both parents. Other operators like reproduction (copy) or inversion (swapping of substrings) are of minor importance.

The two steps of selection and propagation are repeated, until an individual matches the termination criterion. It is worth stressing the fact, that nothing has to be known about the actual solution of the problem and not even sample solutions (as with neural networks) have to be given. However, the more knowledge is put into the fitness function by making it more accurate and more "linear" (i.e. proportional to the actual distance to the solution string) the faster the algorithm will converge.

This directed stochastic search makes genetic algorithms a very robust and universal tool for almost any optimisation problem which can be expressed in a reasonably small set of parameters. Thus, they are e.g. often used in aero- and hydrodynamic design where, in lack of a practicable computer model, the fitness is evaluated by experiment. More academic applications are found e.g. in game theoretic simulations and artificial life.

1.4 About this Project

This project deals with the use of genetic algorithms for the training of neural networks. While gradient descend methods, of which backpropagation is the most popular, can be very fast and, since they normally don't need global data, are well apt to run specialised hardware as neural chips, they also have certain drawbacks: They are "greedy" (i.e. only search in the momentarily best direction) which can lead to convergence problems with highly nonlinear problems, and they are hard to parallelize on a distributed memory system due to their high interdependence which raises communication costs.

Genetic algorithms, on the other hand, are very robust and explore the search space more uniformly. And, since every individual is evaluated independently, they are perfectly suited to run on a distributed memory machine like the 16-transputer-network, for which the parallel versions of the simulations have been written.

This report should give a brief description of the genetic, backpropagation and combined algorithms used and present their serial and parallel implementations, the usage of the corresponding programs, the statistical data, gathered from test runs and the overall conclusions that can be drawn from them.

2 Neuronal Networks

2.1 Topology

In the most general point of view, artificial neural networks can be seen as function networks of a certain topology. A topology can be defined as a directed *graph* G, which consists of a set of *nodes* N (the Neurones) and a set of *transitions* T, which represent directed connections between the nodes. A graph is called *cyclic*, if there exists a series of transitions which begins and end at the same node.

$$G = (\mathbf{N}, \mathbf{T}), \qquad \mathbf{N} = \{N_1, N_2, \ldots N_n\}, \qquad \mathbf{T} \subseteq \mathbf{N}^2$$

Each Neurone N_i in the topology graph is associated with a state $s(N_i) = s_i$, which represents its current activation i.e. its output state O_i, while the vector I_i of the states of all nodes from which a transition leads to N_i is called the input state of N_i. The set of possible states S can be any real interval or any finite set. If it happens to be $\mathbf{S} = \{0, 1\}$, the network is called Boolean or logical. The propagation function f_i of the neurone N_i associates the input state I_i with the output state O_i. If the I_i is of dimension 0, then f_i and O_i are constant.

$$I_i = (x_1, x_2, \ldots x_k), \qquad O_i = s_i, \qquad f_i : \mathbf{S}^k \to \mathbf{S}$$

$$\text{with} \quad x_j = s_{l_j}, \quad k = |\mathbf{M}_i|, \quad l_j \in \mathbf{M}_i, \quad \mathbf{M}_i = \{l \,|\, (N_l, N_i) \in \mathbf{T}\}$$

A set $\mathbf{I}$ of p Nodes are defined as the input nodes, the vector I of their states is the input state or input vector of the network. (Normally it is also demanded that $\mathbf{I}$ satisfies $\{A \,|\, (A, B) \in \mathbf{T}\} \cap \mathbf{I} = \emptyset$.) A set $\mathbf{O}$ of q Nodes are defined as output nodes, their state vector O is the output vector of the network.

$$\mathbf{I} = \{X_1, X_2, \ldots X_p\}, \qquad X_i \in \mathbf{N}, \qquad I = (s(X_1), s(X_2), \ldots s(X_p))$$

$$\mathbf{O} = \{Y_1, Y_2, \ldots Y_q\}, \qquad Y_i \in \mathbf{N}, \qquad O = (s(Y_1), s(Y_2), \ldots s(Y_p))$$

2.2 Temporal Behaviour

2.2.1 Cyclic Networks

The relation between the input and output can be described by a set of equations, where the input states I are known, and all other states are variable. If the network graph is cyclic, there exists a least one series of transitions

$$\langle (A, A_1), (A_1, A_2), \ldots (A_n, A) \rangle$$

which begins and end at the same node A. Thus, the equation for a, the state of A, will contain a itself and may therefore have no solution.

2.2.2 Networks as Dynamic Systems

Cyclic networks are *dynamic systems* which must be described by their temporal behaviour, thus all states s_k become a function of time $s_k(t)$ (or $s_{k,t}$ for discrete time). In the case $\mathbf{S}$ is a real interval, the propagation function $f_i(I_i)$ must be replaced by a corresponding *temporal operator* $F_i(I_i)$. If the evaluation takes place in continuous time, $F_i(I_i)$ would be a differential operator, if discrete timesteps are assumed, $F_i(I_i)$ would be a difference operator, using the Δ-operator ($\Delta x_t = x_t - x_{t-1}$) to refer to previous values of states. The dynamic behaviour of the network could then be described by a set of differential or difference equations with the following boundary conditions, of which the last one is optional, depending whether the input is hold fixed during the simulation.

$$s_k = s_k(t), \qquad (\forall k)\ s_k(0) = s_k^{(0)}, \qquad (\forall X_i \in \mathbf{I})\ s_i(t) = s_i$$

Another possibility of dealing with discrete time is the use of recursion, where the new states is calculated using the old states. The temporal operator F_i of a neurone can be defined as:

$$F_i(I_i)\, O_{i,t} = f_i(I_{i,t-1}), \quad \text{with} \quad I_{i,t} = (x_{1,t}, x_{2,t}, \ldots x_{k,t})$$

The description of the network would then result in a system in a recursive formula.

2.3 Common Network Types

2.3.1 Feed Forward Networks

A network with the topology $G = (\mathbf{N}, \mathbf{T})$, the input nodes $\mathbf{I}$ and the output nodes $\mathbf{O}$ is called a *feed forward network* if it satisfies the following three conditions:

$$\neg\, \mathrm{cyclic}\,(\mathbf{G}), \qquad \{A \,|\, (A,B) \in \mathbf{T}\} \cap \mathbf{I} = \emptyset, \qquad \{B \,|\, (A,B) \in \mathbf{T}\} \cap \mathbf{O} = \emptyset$$

Since the network is supposed to be acyclic, the output state is a direct function of the input state. This network function f defines the functionality of the network.

$$O = f(I) \quad \text{with} \quad I \in \mathbf{S}^p, \quad O \in \mathbf{S}^q, \quad f : \mathbf{S}^p \to \mathbf{S}^q$$

2.3.2 n-Layer Networks

A feed forward network with the topology $G = (\mathbf{N}, \mathbf{T})$, the input nodes $\mathbf{I}$ and the output nodes $\mathbf{O}$ is called a *n-layer network* if it satisfies the following conditions:

$$\mathbf{N} = \bigcap_{k=0}^{n} \mathbf{N}_k, \qquad \mathbf{N}_i \cap \mathbf{N}_j = \emptyset \iff i \neq j$$

$$\mathbf{T} = \bigcap_{k=1}^{n} \mathbf{T}_k, \qquad \mathbf{T}_i \cap \mathbf{T}_j = \emptyset \iff i \neq j$$

$$\mathbf{I} = \mathbf{N}_0, \qquad \mathbf{O} = \mathbf{N}_n, \qquad \mathbf{T}_k \subseteq \mathbf{N}_{k-1} \times \mathbf{N}_k$$

Note that the input layer $\mathbf{N}_0$ is not counted as a real layer, since their propagation functions are merely constants set to the components of the input vector.

The partitions $\mathbf{T}_k$ can also be expressed my an adjacence matrix M_k which is defined follows:

$$\mathbf{N}_j = \{N_{j,i} \,|\, 1 \leq i \leq |\mathbf{N}_j|\} \qquad s = |\mathbf{N}_{k-1}|, \qquad t = |\mathbf{N}_k|$$

$$M_k = \begin{pmatrix} m_{11}^{(k)} & \cdots & m_{1t}^{(k)} \\ \vdots & \ddots & \vdots \\ m_{s1}^{(k)} & \cdots & m_{st}^{(k)} \end{pmatrix}, \qquad m_{ij}^{(k)} = \begin{cases} 0, & (N_{k,i}, N_{k,j}) \notin \mathbf{T}_k \\ 1, & (N_{k,i}, N_{k,j}) \in \mathbf{T}_k \end{cases}$$

If $(\forall i, j)\ m_{ij}^{(k)} = 1$, the layers $\mathbf{N}_{k-1}$ and $\mathbf{N}_k$ are *fully connected.*

2.4 The Neurone

2.4.1 The Propagation Function

Since neural networks can merely be seen a function networks, neurone types are classified by the type of their propagation function. The propagation function f_k of a neurone N_k is normally of the form

$$f_k(x) = g(h_k(x)), \qquad f_k : \mathbf{S}^n \to \mathbf{S}, \quad h_k : \mathbf{S}^n \to \mathbf{R}, \quad g : \mathbf{R} \to \mathbf{S}, \qquad n = |\mathbf{I}_k|$$

The function h_k is called the netto input function of the neurone N_k, and maps the input states onto a single real value. The function g is called activation function, is usually the same for all neurones and maps the real netto input back onto $\mathbf{S}$.

2.4.2 The Nettoinput Functions

Since most learn algorithm train the network by iteratively changing the netto input functions of the neurones, they can be written as

$$h_k = h(P_k) \quad \text{and} \quad h_k(I_k) = h(P_k; I_k)$$

where P_k is a vector of function parameters, which is to be determined by the learn algorithm. Normally, each input node $X_i^{(k)}$ of the neurone N_k is associated with a parameter $w_i^{(k)}$ called its weight. Often, there is an extra parameter, called threshold or bias.

A very common definition of the netto input, is a weighted sum (a dot product) to which the bias is added.

$$h(I) = h(P, I) = \theta + \sum_{i=1}^{n} x_i w_i \quad \text{with} \quad I = (x_1, x_2, \ldots x_n), \quad P = (w_i, w_2, \ldots w_n; \theta)$$

A convenient way to store the weights for n-layer networks is by replacing the adjacence matrix M_k by the *weight matrix* W_k, which contains the weights the input nodes or 0 if there is no connection.

2.4.3 The Activation Function

The activation function g is normally a monotone, nonlinear function to rescale the netto input to $\mathbf{S}$, which is usually a limited interval. If $\sup \mathbf{S} = 1$ and $\inf \mathbf{S} = 0$, then g uses binary logic, if $\sup \mathbf{S} = 1$ and $\inf \mathbf{S} = -1$, then g uses bipolar logic.

The two most commonly used activation functions for binary logic are the step- or Θ-function and the sigmoide function σ_c.

$$\Theta(x) = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases}, \qquad \sigma_c(x) = \frac{1}{1 + \mathrm{e}^{-cx}}$$

2.4.4 Common Neurone Types

In the following examples, $\mathbf{B}$ stands for the set $\{0, 1\}$, $\mathbf{Z}$ is the set of integer numbers and $\mathbf{R}$ the set of real numbers. All neurones are assumed to have n inputs. $I \cdot W$ stands for $\sum_{i=1}^{n} x_i w_i$.

McCulloch-Pitts-Cells

$$\mathbf{S} = \mathbf{B}, \quad P = (W;\theta), \quad W \in \mathbf{B}^n$$

$$h(P;I) = \begin{cases} I \cdot W - \theta, & (\forall i)\ x_i = 0 \vee w_i = 1 \geq 0 \\ 0, & (\exists i)\ x_i = 1 \wedge w_i = 0 \end{cases}, \quad g(x) = \Theta(x)$$

Perceptron

$$\mathbf{S} = \mathbf{R}, \quad P = (W;\theta), \quad W \in \mathbf{R}^n \quad h(P;I) = I \cdot W + \theta, \quad g(x) = \Theta(x)$$

Standard Backpropagation Neurone This type of neurone is very often used with the backpropagation algorithm and also the main type for this project. No explicit θ is defined, however, the effect of the bias can be achieved by simply adding an extra neurone to each layer with a constant propagation function of 0.

$$\mathbf{S} = \mathbf{R}, \quad P = W, \quad W \in \mathbf{R}^n \quad h(P;I) = I \cdot W, \quad g(x) = \sigma_c(x)$$

3 The Genetic Algorithm

3.1 Chromosome Strings

A Chromosome String is a fixed length string of a genetic alphabet $\mathbf{A}$.

$$S = (d_0, d_1, \ldots d_n), \qquad S \in \mathbf{A}^n, \qquad d_i \in \mathbf{A}$$

The function len returns the length of a string, the operator $\oplus$ concatenates two strings.

$$S = (d_0, d_1, \ldots d_n) \Longrightarrow \mathrm{len} S = n$$

$$A = (a_1, a_2, \ldots a_n),\ B = (b_1, b_2, \ldots b_m) \Longrightarrow A \oplus B = (a_1, a_2, \ldots a_n, b_1, b_2, \ldots b_m)$$

If $A = \{0, 1\}$ then the Hamming distance H between two string of the same length is define as

$$H(A, B) = \sum_{i=1}^{n} |A_i - B_i| \quad \text{with} \quad \mathrm{len} A = \mathrm{len} B = n$$

From now on, we will assume that the genetic alphabet is always $\{0, 1\}$.

3.2 The Individual

3.2.1 Genotype and Phenotype

The genotype S of an individual I is its chromosome string, while its phenotype P is the representation of the problem, the user originally wants to solve. Since the genetic algorithm operates only on the genotype, a function must be provided to translate P into S.

How this encoding is done, depends on the type of the problem parameters P and no fixed rules can be given. A encoding is optimal if the relation between the fitness $f(I)$ of a individual I is in a linear relation to the Hamming distance between S and S_{opt}, when S_{opt} represents the genotype of the optimal solution:

$$f(I) = -k\, H(S, S_{opt}) + c, \quad k > 0$$

Since the S_{opt} in normally not known, the general strategy should be, that similar phenotypes should also have a small Hamming distance between their corresponding genotypes. Moreover, related groups of parameters should be encoded in subsequent substrings of the genotype.

3.2.2 The Grey Code

While Boolean parameters of the phenotype P can be directly encoded by mapping them onto the chromosome string S, the encoding of integer and real values is not so straightforward.

Using binary numbers for the encoding of integers of the range $[0, 2^n - 1]$, leads to the problem, that the encoding of two successive numbers a and $a+1$ may have a Hamming distance up to $n-1$.

$$\max\{H(\mathrm{bin}_n a, \mathrm{bin}_n(a+1)) \mid a \in \mathbf{Z}_{2^n-1}\} = H(\mathrm{bin}_n(2^{n-1}-1), \mathrm{bin}_n(2^{n-1})) = n-1$$

The Grey code encodes the successor $a+1$ of an integer a by inverting one digit of $\mathrm{grey}_n a$, and is defined by the following conditions.

$$\mathrm{grey}_n : \mathbf{Z}_{2^n} \to \mathbf{B}^n, \quad a, b \in \mathbf{Z}_{2^n}, \quad \mathbf{Z}_k = \{0, 1, \ldots k-1\}$$

$$\mathrm{grey}_n a = \mathrm{grey}_n b \iff a = b$$

$$\mathrm{grey}_1 0 = (0), \qquad \mathrm{grey}_{n+1} a = (0) \oplus \mathrm{grey}_n a$$

$$H(\mathrm{grey}_n a, \mathrm{grey}_n(a + 1 \bmod 2^n)) = 1$$

A real value $x \in [a, b)$ can be mapped onto $\mathbf{Z}_{2^n}$ and then be Grey encoded as n-bit strings. The mapping function ϕ must be strictly monotonous and its return values should be uniformly distributed. For already uniformly distributed parameters, ϕ should be defined as

$$\phi : [a, b) \to \mathbf{Z}_{2^n}, \qquad \phi(x) = \left\lfloor 2^n \, \frac{x-a}{b-a} \right\rfloor$$

3.2.3 Encoding Neuronal Networks

If the genetic algorithm is to be used for the training of neural networks, the phenotype P is the set of network parameters which are to be optimised, namely the set of the parameter vectors P_k of all neurones in $\mathbf{N}$.

The main network type used in this project is a 2-layer network which is fully interconnected except for one extra neurone in the input and the hidden (the second) layer which has no inputs and is constantly set to 1. The other neurones are either input or standard backpropagation neurones as described in Section 2.4.4.

Thus, the parameters of a n-m-o-network can be described by two real weight matrices $W^{(1)}$ and $W^{(2)}$ of the dimensions $(n+1) \times m$ and $(m+1) \times o$. If we decide to use Grey code with a precision of b bit, restrict the possible weight values to the interval $[-w, w]$ and assume them to be uniformly distributed, then the encoding function e_{num} for weights can be defined as

$$e_{num} : [-w, w] \to \mathbf{B}^b, \qquad e_{num}(x) = \mathrm{grey}_b\, \phi(x)$$

$$\text{with} \quad \phi : [-w, w] \to \mathbf{Z}_{2^b}, \qquad \phi(x) = \left\lfloor (2^b - 1)\, \frac{x+w}{2w} \right\rfloor$$

The encoding function e_{net} for the whole network is very straightforward and simply concatenates all parameters. The input weights of one neurone are encoded together as one coherent substring.

$$e_{net} : \mathbf{R}^{(n+1)m} \times \mathbf{R}^{(m+1)o} \to \mathbf{B}^l, \quad l = b\,((n+1)m + (m+1)o)$$

$$e_{net}(W^{(1)}, W^{(2)}) = \bigoplus_{j=1}^{m} \bigoplus_{i=1}^{n+1} e_{num}(w_{ij}^{(1)}) \;\oplus\; \bigoplus_{k=1}^{o} \bigoplus_{j=1}^{m+1} e_{num}(w_{ij}^{(2)})$$

3.3 The Population

The genetic algorithm doesn't work on a single individual but on a whole population $\mathbf{P}$ of p individuals which undergoes an evolutionary process starting with the initial Population $\mathbf{P}_0$.

3.3.1 The Initial Population

The simplest way to create $\mathbf{P}_0$, is by simply generating p random strings of length l. However, it is also possible to generate the phenotypes of the individual and then translating them into their corresponding genotypes. If certain parameter-combinations of the phenotype tend to be more successful, this knowledge can be used to improve the quality of $\mathbf{P}_0$ and lead to a faster convergence of the algorithm. However this will cut down the probability of finding totally different combination which might perform even better.

3.3.2 Decimation

A better method of improving the quality of the initial population is *decimation*. A population $\bar{\mathbf{P}}_0$ of the size pd is created at random of which the best (i.e. the fittest) p individuals are selected into $\mathbf{P}_0$. The factor d is called *decimation factor*.

3.4 Fitness

The fitness function f is used to direct the evolutionary process into a certain direction. The genetic algorithm is in fact merely a method of approximating the global maximum of f in the search space of chromosome strings. The actual interpretation of this search space (the phenotypes) is packed into f and doesn't concern the algorithm itself.

3.4.1 Linearity

As mentioned in Section 3.2.1, the optimal choice for f is in a linear relation to the Hamming distance H_{opt} to the optimal solution S_{opt}. In a more general definition of linearity, problems are also refered to as linear, when the following condition applies.

$$H(A, S_{opt}) < H(B, S_{opt}) \Rightarrow f(A) > f(B), \qquad A, B \in \mathbf{B}^l, \quad f : \mathbf{B}^l \to \mathbf{R}$$

Nonlinearities in f result in the slower convergence of the algorithm. Due to its stochastic nature, the algorithm will always eventually find a solution, but in some cases the number of necessary evaluations of f can be greater than 2^l and the performance is worse then in a simple systematic search of $\mathbf{B}^l$.

3.4.2 Fitness and Error

In many cases, it is more natural to refer to the quality of an individual by its error instead of its fitness and use the genetic algorithm to minimises the error. As this is the case with neural networks, we will occasionally replace the fitness function f by the error function E.

3.4.3 Error Function of a Neural Network

A training set $\mathbf{T}$ of a neural network is a set of pairs of a sample input vectors $I^{(k)}$ and its associated output vector $O^{(k)}$, which are called training *patterns*. If f is the network function, then the error E of the network for the pattern $(I^{(k)}, O^{(k)})$ is defined as

$$E_k = \frac{1}{2}\sum_{i=1}^{q}(O_i^{(k)} - O_i)^2 \quad \text{with} \quad O = f(I^{(k)}), \quad f : \mathbf{S}^p \to \mathbf{S}^q$$

To use this definition for the genetic algorithm, the network (the phenotype) must be decoded from the chromosome string. The decoding function d_{net} is inverse function of e_{net} as defined in Section 2.4.4. [1]

To calculate an error for all patterns, a mean value of all E_k must be calculated. If all patterns are of equal importance and a low sum of all errors is more important than a small maximum error for each pattern, then the arithmetic mean of all E_k should be returned.

$$E = \frac{1}{t}\sum_{j=1}^{t} E_k(d_{net}(S)), \qquad t = |\mathbf{T}|, \quad S \in \mathbf{T}^l$$

If the training set is very large and highly redundat, it is possibel to estimate the error by evaluating only a subset of $\mathbf{T}$ which is changed for each generation. (This can be seen as the genetic eqivalent to the backpropagation online learning described in Section 4.1.1.)

3.5 Evolution

The key part of the genetic algorithm if the evolutionary step of producing a child-population $\mathbf{P}'$ out of a parent-population $\mathbf{P}$ and thereby generating new generations of the initial population $\mathbf{P}$. This process consists of two main parts: *selection* and genetic transformation by applying *genetic operators* to the selected individuals.

[1]Since e_{net} maps real numbers onto strings, its inverse can in fact only return the corresponding class of networks which are mapped onto the same string, but we will gently ignore this subtlety and be content if any network of the class is returned.

3.5.1 Selection

The selection operator $\mathcal{S}$ selects an individual of the given population according to its fitness. The operator $\mathcal{S}_n$ selects a set of n individuals and is defined as follows.

$$\mathcal{S}_0\mathbf{P} = \emptyset, \qquad \mathcal{S}_{n+1}\mathbf{P} = \{\mathcal{S}\mathbf{P}\} \cup \mathcal{S}_n\mathbf{P}$$

In the following examples, $\mathbf{P} = \{S_1, S_2, \ldots S_p\}$, $f_k = f(S_k)$ and the function rnd() returns a random number of the interval $[0, 1)$.

Random Selection Since this method ignores fitness, it leads to a blind search unless used in combination with another method.

$$\mathcal{S}_{RD}\,\mathbf{P} = S_r, \quad r = \lfloor p\,\mathrm{rnd}()\rfloor$$

Roulette Wheel Selection This method selects a individual with a probability directly proportional to its fitness. The disadvantage of this method is, that is strongly depends on the actual definition of f. A simple transformation $f' = \phi(f)$ with a strictly monotonously increasing function ϕ would result in a different distribution.

$$\mathcal{S}_{RW}\,\mathbf{P} = \begin{cases} S, & S = \mathcal{S}_{RD}\,\mathbf{P} \wedge \hat{f}(S) > \mathrm{rnd}() \\ \mathcal{S}_{RW}\,\mathbf{P}, & \text{otherwise} \end{cases} \qquad \text{with} \quad \hat{f}(S) = \frac{f(S)}{\sum_{k=1}^{p} f_k}$$

Rank Selection This method avoids the disadvantage of the Roulette Wheel selection by selecting individuals according to their rank in the whole population and thereby gaining a fixed distribution. In the case of a linear distribution with a uniform fraction of u, $\mathcal{S}_{RS}$ is defined as

$$\mathcal{S}_{RS} = \mathcal{S}_{RW}[f'], \qquad f'(S) = \frac{(1-u)}{p}\,|\{S' \in \mathbf{P} \mid f(S) \geq f(S')\}| + u$$

This selection mechanism is used in this project, but to avoid the calculation of the ranks, the following equivalent method is used. [2]

[2] In fact, the method is only equivalent if $f(A) = f(B) \iff A = B$. They are, however, totally equivalent, if the ranks are determined by sorting.

$$\mathcal{S}_{RS}\,\mathbf{P} = \begin{cases} \mathcal{S}_{RD}\,\mathbf{P}, & \text{rnd}() \leq u \\ \mathcal{S}_{TS}\,\mathbf{P}, & \text{otherwise} \end{cases}$$

Tournament Selection This method is often used in optimising game strategies, where two individuals play and the winner is selected. This method can be used even without knowing f.

$$\mathcal{S}_{TS}\,\mathbf{P} = \begin{cases} A, & f(A) \geq f(B) \\ B, & f(A) < f(B) \end{cases}, \qquad A = \mathcal{S}_{RD}\,\mathbf{P}, \quad B = \mathcal{S}_{RD}\,\mathbf{P}$$

3.5.2 Genetic Operators

The most commonly used genetic operators are *n-point mutation* ($\mathcal{M}_n$), *crossover* ($\mathcal{C}$) and *reproduction* ($\mathcal{R}$). The simulation parameters m, c and r declare on how many selected individuals of the parent generation $\mathbf{P}$ each operator is applied to produce the child generation $\mathbf{P}'$.

$$\mathbf{P}_{k+1} = \mathbf{P}'_k \quad \mathbf{P}' = \mathbf{P}'_M \cup \mathbf{P}'_C \cup \mathbf{P}'_R$$

$$\mathbf{P}'_M = \{\mathcal{M}\,S \,|\, S \in \mathbf{P}_M\},\ \mathbf{P}'_C = \{\mathcal{C}\,(A,B) \,|\, (A,B) \in \mathbf{P}_C\},\ \mathbf{P}'_R = \{\mathcal{R}\,S \,|\, S \in \mathbf{P}_R\}$$

$$\mathbf{P}_M = \mathcal{S}_m\,\mathbf{P}, \quad \mathbf{P}_C = \bigcup_{k=1}^{c/2} \{(\mathcal{S}\,\mathbf{P}, \mathcal{S}\,\mathbf{P})\}, \quad \mathbf{P}_R = \mathcal{S}_r\,\mathbf{P}$$

$$\text{with} \quad m + c + r = p, \quad c \bmod 2 = 0, \quad p = |\mathbf{P}| = |\mathbf{P}'|$$

In the following definitions, the function $\text{rint}(l)$ returns a random integer between 0 and $l-1$ and A, B and S are individuals with $A, B, S \in \mathbf{B}^l$.

Mutation The operator $\mathcal{M}$ inverts one bit of the individual, the n-point mutation operator $\mathcal{M}_n$ applies $\mathcal{M}$ n times. The parameter n can be fix for the whole simulation, or be chosen randomly each time the operator is applied. (In this project $n = 1 + \text{rint}(N-1)$, where N is a simulation parameter.)

$$S' = \mathcal{M}\,S, \quad S'_i = \begin{cases} 1 - S_i, & i = r \\ S_i, & i \neq r \end{cases} \quad r = \text{rint}(l) \quad \mathcal{M}_n = \mathcal{M}^n$$

Crossover The operator $\mathcal{C}$ exchanges the chromosome strings of two individuals starting from a random index.

$$(A', B') = \mathcal{C}\,(A, B)$$

$$(\forall\, i < r)\,(A'_i = A_i \,\wedge\, B'_i = B_i), \quad (\forall\, i \geq r)\,(A'_i = B_i \,\wedge\, B'_i = A_i), \quad r = \mathrm{rint}(l)$$

Reproduction The operator $\mathcal{R}$ simply reproduces the individual.

$$S' = \mathcal{R}\, S, \quad S' = S$$

4 Backpropagation

A very popular learn algorithm for feed forward networks with linear netto input functions and differentiatabel activation functions is the backpropagation algorithm.

4.1 The Error Gradient

As in Section 3.4.3, the error function of the training pattern $(I^{(k)}, O^{(k)})$ is once again defined as

$$E_k = \frac{1}{2}\sum_{i=1}^{q}(O_i^{(k)} - O_i)^2 \quad \text{with} \quad O = f(I^{(k)}), \quad f : \mathbf{R}^p \to \mathbf{R}^q$$

Since the network function f is itself a function of the network parameters $\mathbf{P}$ and $\mathbf{P}$ is the set $\mathbf{W}$ of the weight vectors W_j of the n neurones, the error function E_k is also a function of $\mathbf{W}$ and a gradient can be defined as

$$\nabla E_k = \nabla E_k(W_1, W_2, \ldots W_n) = \left(\frac{\partial E_k}{\partial W_1}, \frac{\partial E_k}{\partial W_1}, \ldots \frac{\partial E_k}{\partial W_n}\right)$$

$$\frac{\partial E_k}{\partial W_j} = \left(\frac{\partial E_k}{\partial w_{j1}}, \frac{\partial E_k}{\partial w_{j2}}, \ldots \frac{\partial E_k}{\partial w_{jp_j}}\right), \quad p_j = \dim W_j$$

The backpropagation algorithm is a gradient descent method and thus, the weights are updated with the negative gradient of the error function.

4.1.1 Online and Batch Learning

The weights can be updated immediately after $\Delta\mathbf{W}$ is determined for a pattern. This is called *online learning.*

$$\mathbf{W}^{(i+1)} = \mathbf{W}^{(i)} + \Delta\mathbf{W}^{(i)}, \quad \Delta\mathbf{W}^{(i)} = -\gamma\nabla E_k(\mathbf{W}^{(i)})$$

Batch Learning updates the weights with the arithmetic mean of the corrections for all t patterns. This can lead to better results with small and very heterogeneous learn sets.

$$\Delta\mathbf{W}^{(i)} = -\frac{\gamma}{p}\sum_{k=1}^{t}\nabla E_k(\mathbf{W}^{(i)})$$

The constant γ is called *learn rate.* A high value of γ leads to greater learn steps at the cost of lower accuracy.

4.1.2 Learning with Impulse

In regions where the error function is very flat, the resulting gradient vector will be very short and lead to very small learn steps. A solution to this problem is the introduction of an impulse term which is added to the update $\Delta\mathbf{W}$ and steadily becomes greater if the direction of $\Delta\mathbf{W}$ remains stable.

$$\Delta\mathbf{W}^{(i)} = -\gamma\nabla E(\mathbf{W}^{(i)}) + \alpha\Delta\mathbf{W}^{(i-1)}, \quad \alpha \in [0, 1)$$

The *impulse constant* α reflects the "acceleration" a point gets on descending the error function. If we assume ∇E as constant, the maximum acceleration factor a is given by

$$a = \frac{\Delta\mathbf{W}^{(\infty)}}{\Delta\mathbf{W}^{(0)}} = \sum_{i=0}^{\infty}\alpha^i = \frac{1}{1-\alpha}$$

Fig. 1 shows a training process for the XOR-problem (Section 6.2.2) with $\alpha = 0$ and $\alpha = 0.9$.

4.2 Evaluation and Backpropagation

The main feature of backpropagation in comparison with other gradient descent methods is, that, provided that all netto input functions are linear, the weight update $\Delta\mathbf{W}_j$ of the neurone N_j can be found by using only local information, thus information passed through the incoming and outgoing transitions of the neurone. This process consists of the evaluation step, where the error is calculated and the

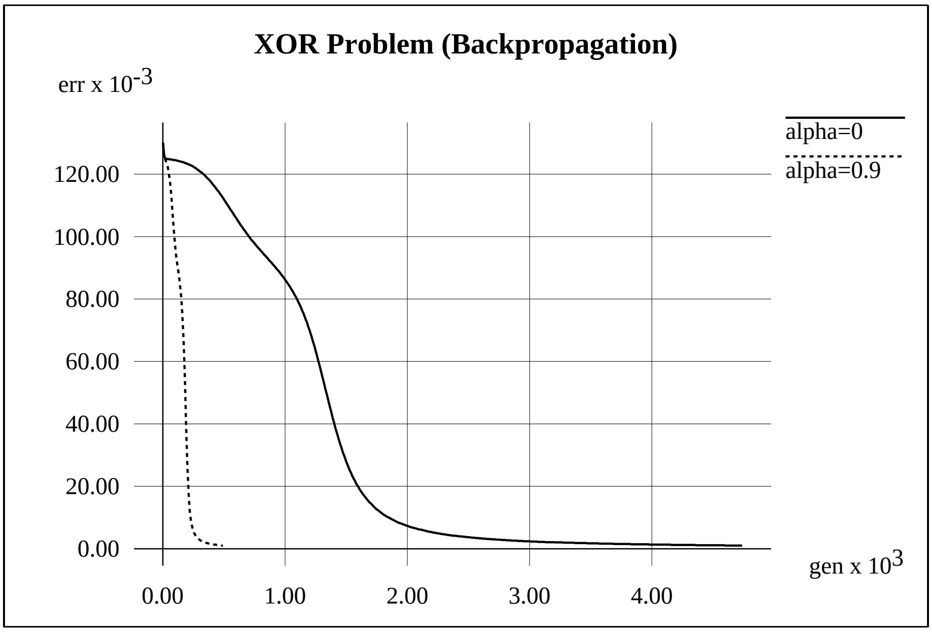

Figure 1: **Error Graph for the XOR Problem**

backpropagation of the error in the inverse direction form the output back to the input neurones.

4.2.1 The Network Function

Due to the linearity of the netto input function, the overall network function f consists merely of additions, scalar multiplications and compositions of the activation functions. The partial derivations are thus calculated as follows:

$$\frac{\partial f_1(x) + f_2(x)}{\partial x} = \frac{\partial f_1(x)}{\partial x} + \frac{\partial f_2(x)}{\partial x}, \quad \frac{\partial k\, f(x)}{\partial x} = k\frac{\partial f(x)}{\partial x}$$

$$\frac{\partial f_2\left(f_1(x)\right)}{\partial x} = \frac{\partial f_1(x)}{\partial x}\left[\frac{\partial f_2(y)}{\partial y}\right]_{y=f_1(x)}$$

4.2.2 Calculating the Error Gradient

During the evaluation step, not only the value of the activation function $g(x)$ but also the value of its derivation $g'(x)$ is calculated for the netto input x. If $g = \sigma_1 = \sigma$, the derivation has a very simple form.

$$\sigma(x) = \frac{1}{1+\mathrm{e}^{-x}}, \quad \frac{\partial \sigma(x)}{\partial x} = \frac{\mathrm{e}^{-x}}{(1+\mathrm{e}^{-x})^2} = \sigma(x)(1-\sigma(x))$$

Since E_k depends of the output vector O (calculated by the network function f) and only indirectly on the weights, ∇E_k can be written as

$$\nabla E_k = \frac{\partial E_k}{\partial \mathbf{W}} = \frac{\partial E_k}{\partial O}\frac{\partial O}{\partial \mathbf{W}}, \quad O = f(I^{(k)})$$

$$\text{and} \quad \frac{\partial E_k}{\partial O_i} = \frac{1}{2}\frac{\partial}{\partial O_i}\sum_{i=1}^{q}(O_i^{(k)} - O_i)^2 = O_i - O_i^{(k)} = \Delta O_i$$

To calculate the partial derivation for each element of the weight vector for each node, the output nodes are set to ΔO_i and $\frac{\partial O}{\partial \mathbf{W}}$ is calculated by successively stepping backward in opposite direction of the transition in $\mathbf{T}$ and applying the above listed derivation rules. Composition is handled by multiplying the stored outer derivation $g'(x)$ onto the sum of the inner derivations δ_j received via the q inverted output transitions.

$$\text{input } \delta_i, \quad \delta = \frac{\partial E_k}{\partial x} = g'(x)\sum_{j=1}^{q}\delta_j$$

Then, δ is propagated to the p input nodes by multiplying it with the corresponding weight w_i. Then the weight is updated.

$$\delta_i' = w_i\delta, \quad \text{output } \delta_i', \quad \Delta w_i = I_i\delta$$

4.3 Backpropagation for a 2-Layer Network

As shown in Section 3.2.3, for standard backpropagation neurones (Section 2.4.4) the parameters of a n-m-o-network can be described by two real weight matrices $W^{(1)}$ and $W^{(2)}$ of the dimensions $(n+1)\times m$ and $(m+1)\times o$. Let the vectors I, H and O refer to the states of the input, hidden and output neurones and $I^{(p)}$ and $O^{(p)}$ to the actual training pattern. The weight updates $\Delta W^{(1)}$ and $\Delta W^{(2)}$ without impulse can then be calculated as follows.

$$\Delta w_{ij}^{(1)} = -\gamma\, I_i\delta_j^{(1)}, \quad \Delta w_{jk}^{(2)} = -\gamma\, H_j\delta_k^{(2)}$$

$$\delta_j^{(1)} = H_j(1-H_j)\sum_{k=1}^{o} w_{jk}^{(2)}\delta_k^{(2)}, \quad \delta_k^{(2)} = O_k(1-O_k)\,(O_k - O_k^{(p)})$$

5 Parallelism

The genetic as well as the backpropagation algorithm are both inherently parallel, in the first case due to the independent calculations of the fitness function, in the latter, due to the local nature of the error propagation. However, parallelism is provided at totally different levels, which has great impact on possible parallel implementations on different architectures, especially on distributed memory systems.

5.1 Performance Measures

5.1.1 Speedup and Efficiency

Let T_0 be the execution time of the serial version of a problem and T_p the execution time of the parallel version of the same problem to run on p processors. The speedup S_p, algorithmic speedup $\bar{S}_p$ and efficiency E_p are then given by

$$S_p = \frac{T_0}{T_p}, \quad \bar{S}_p = \frac{T_1}{T_p} \quad E_p = \frac{S_p}{p}$$

Since the parallelisation of a problem normally involves a certain overhead for synchronisation and/or communication, the efficiency is usually below 100

5.1.2 Amdahl's Law

If a fraction s of the serial problem is inherently sequential and can not be parallelised, the highest possible speedup is limited by Amdahl's law.

$$S_p \leq \frac{T_0}{T^{ser} + \frac{T^{par}}{p}} = \frac{1}{s + \frac{1-s}{p}} \leq \frac{1}{s}$$

$$\text{with} \quad T_0 = T^{ser} + T^{par}, \quad T^{ser} = sT_0, \quad T^{par} = (1-s)T_0$$

5.2 Parallel Architectures

5.2.1 Shared Memory Systems

A shared memory system consists of a number of processes with small local memory (cache), with are connected to common (shared) memory modules via a bus or a system of crossbar switches. All communication between processes is handled via the shared memory. Synchronisation is needed to avoid simultaneous writing access and to separate interdependent program parts.

5.2.2 Distributed Memory Systems

A distributed memory system consists of a higher number of independent processors with sufficiently big local main memory, which are interconnected via hardlinks, a bus or any other interconnection system. All communication must be explicitly programmed, program data and initial parameters must be distributed and the results must be regathered.

Since the communication is significantly slower than in shared memory systems, the computation/communication ratio of a problem must be sufficiently high to allow an efficient parallel implementation.

5.2.3 Specialised Hardware

Specialised hardware as neural chips reflect the structure and the interdependencies of a certain algorithms in their design and thereby minimise any synchronisation and communication overhead. They are therefore able to apply parallelism at very low level as e.g. at the update of a single neurone where the computation/communication ratio is of the order $O(1)$.

5.3 The Parallel Genetic Algorithm

This section describes the distributed memory version of a parallel genetic algorithm for the training of neural networks as it has been implemented for a transputer network of 16 T800's.

5.3.1 Distribution of the Workload

The genetic algorithm consists of two main parts, the evaluation of the fitness and the fitness based selection and generation of a child population. The fitness evaluations for each individual are totally independent and can be thus be done in parallel with each processor holding his share of the global population.

The selections and the generation of a child population by genetic operators are also independent from each other, but there are three problems to exploit this parallelism on a distributed memory system:

1. The selection must be based on the fitness data of the whole population.
2. The crossover operator works on individuals selected form the whole population.
3. The selection is not necessarily equally distributed between the local populations on each processor and can unbalance the workload.

It is therefor necessary to collect the individuals with their associated errors from all processors to one master processor, which performs those steps and redistributes the child population again in equal portions. [3]

5.3.2 The Computation/Communication Ratio

For a n-m-o network with a training set of t patterns and an encoding in chromosome strings of the length l, the computation/communication ratio r is given by

$$r = \frac{T^{comp}}{T^{comm}} = \frac{O(t((n+1)m+(m+1)o))}{O(l)} = \frac{O(t(n+o)m)}{O((n+o)m)} = O(t)$$

The ratio between the (parallel) error calculation and the (serial) selection and genetic operations is also of order (t). Since t is normally sufficiently high, this should allow an efficient parallel implementation. Fig. 2 gives the speedup graph for the parallel implementation of the encoder/decoder problem (Section 6.2.1).

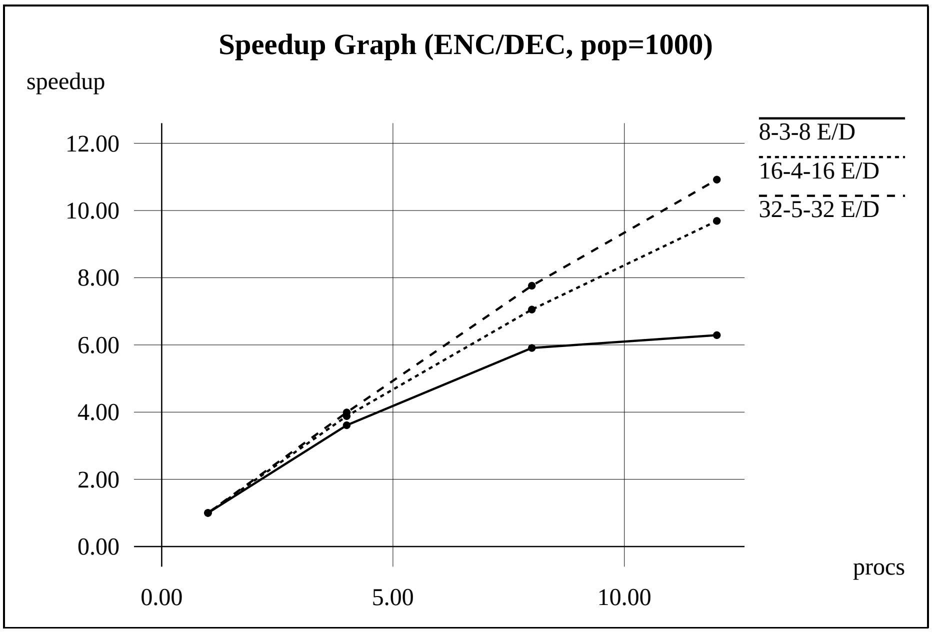

Figure 2: **Speedup for parallel ENC/DEC-problem**

To further improve the computation/communication ratio, it is possible to perform the global selection not for every new generation, but only after a certain

[3]With very low crossover rates, very long chromosome strings and a simple and fast fitness function, a more sophisticated distribution scheme with "communication on demand" might bring some additional speedup.

number of local selections. This possibility is provided as an option in the parallel implementation. However, due to the negative impact on global covergence, the use of this method brings hardly any speedup.

5.3.3 Finding a Topology

The genetic algorithm involves only communications between master and slaves, which should be reflected in an appropriate topology of the transputer network. If each processor can be directly linked to n other processor, then the optimal topology is an $(n-1)$-ary tree with the master as root. Since the used T800's have four hardlinks, a ternary tree was used. To avoid unnecessary idle times, also the master works on his own local population. (In the parallel implementation for this project even the same sourcecode is used for master and slaves.)

5.4 A Combined Algorithm

While, in opposite to backpropagation, the genetic algorithm allows very efficient parallel implementation on distributed memory system, its main disadvantage is, that, due to its stochastic nature, its superior robustness and reliability is often outweighed by its slower convergence and thus longer calculation times, which often leaves backpropagation the better choice for squential network training.

These facts suggest the conclusion, that a combined algorithm which exploits the parallel nature and robustness of the genetic algorithm without sacrificing the efficiency of backpropagation might perform better than each algorithm for itself.

5.4.1 Data Conversion

A hybrid genetic backpropagation algorithm will necessarily involve a conversion between the chromosome string and the matrix representation of a network as introduced in Section 3.2.3. This brings up two encoding problems, for which a reasonable compromise must be found.

Accuracy While backpropagation normally uses a hardware floating point representation with a length of 32 or even 64 bit per weight, the genetic algorithm normally uses a representation just long enough to guarantee a sufficient diversity of the population. Thus, too short representations might cause small weight updates to be ignored, while too long representations will lead to unnecessary long chromosome string, which slow down the genetic features of the algorithm by increasing communication times.

Range The representations of numbers as grey encodes bitstrings is limited to a certain range. If this range is too small, gradient updates exceeding the limits would be lost, if it is too large, this unnecessarily increases the searchspace for the stochastic genetic search.

5.4.2 Genetic Backpropagation

The combined algorithm used in this project works, like the standard genetic algorithm, on a population of string-encoded networks, but before the fitness is calculated, the individual is decoded, undergoes a certain number b of backpropagation steps and is encoded again. Then, the selection is performed and a new population is generated.

Initialising the Population The initial population is not generated by randomising the chromosome strings but by randomising weight matrices which are then encoded as strings. This allows the initial weights to be distributed in a smaller range than the used encoding interval and reduces the probability of starting backpropagation in a very flat region of the error function which would lead to very small gradients.

Backpropagation Parameters As introduced in Section 4.1, the main parameters of the backpropagation algorithm are the learn rate γ and the impulse rate α. Instead of fixing γ and α for all individuals as a simulation parameter, their values can also be optimised by the genetic algorithm by including them into the chromosome string. The phenotype new P'_k of the individual k is then determined by the phenotype P_k of the network (described by the weight matrices $W_k^{(1)}$ and $W_k^{(2)}$, see Section 3.2.3 for details) and the backpropagation parameters γ_k and α_k. The new encoding function e'_{net} is the defined as

$$e'_{net} : \mathbf{R}^{(n+1)m} \times \mathbf{R}^{(m+1)o} \times \mathbf{R} \times \mathbf{R} \to \mathbf{B}^l, \quad l = b\left((n+1)m + (m+1)o + a + g\right)$$

$$e'_{net}(W^{(1)}, W^{(2)}, \gamma, \alpha) = e_{net}(W^{(1)}, W^{(2)}) \oplus \mathrm{grey}_g\, \phi_g(\gamma) \oplus \mathrm{grey}_a\, \phi_a(\alpha)$$

The mapping functions ϕ_g and ϕ_a map the possible values of γ and α onto the integer intervals $\{0, \ldots 2^g - 1\}$ and $\{0, \ldots 2^a - 1\}$. For the implementation in this project, ϕ_g was chosen to be logarithmic and ϕ_a to be linear over the following intervals:

$$\phi_g : [0.02, 20] \to \mathbf{Z}_{256}, \quad \phi_a : [0, 0.95] \to \mathbf{Z}_{256}$$

6 Implementations

This section describes the actual implementations of the introduced algorithms in both, serial and parallel versions, as well as the sample problems they have been tested with.

6.1 Program Layout

6.1.1 Hardware

The serial versions of the programs are designed to run under any standard UNIX environment and were developed on an Encore Multimax 520 mainframe with 14 NS32532 processors. For testing and performance measurements, also several Hewlett Packard Apollo workstations (Series 700 with 32 MB RAM) have been used.

The target system for the parallel versions was a network of 16 T800 transputers, each equipped with floating point unit, 4 MB RAM and 4 free configurable serial hardlinks, using a Sun workstation as host.

6.1.2 Developing Software

All programs are written in ANSI C using only standard UNIX libraries and include files except for the parallel versions, which also use the Meiko Computing Surface Network library CS-Tools with their according header files for transputer specific functions. The following compilers have been used:

Target System	**Compiler**
Multimax	GNU project C and C++ Compiler (v2 preliminary)
HP Apollo	HP-UX C compiler
Transputer	MEIKO c compiler driver

6.1.3 General Principles

The aim of this project was to explore and test genetic algorithms for the training of neural networks on a transputer network as a possible alternative to backpropagation. A task like this naturally involves permanent changes an enhancements in the program as well as the testing of several algorithmic alternatives and would normally result in a series of different program versions, which makes global changes and debugging very difficult and direct comparisons between programs often impossible.

To avoid this problems, all simulation parameters, procedural alternatives and new features were either added as options to the existing program or linked to it at an object code level, keeping the changes to the original code as minimal as possible. This requires a strict modularity of the code with well defined, standardised interfaces. The class concept and the inheritance features of an object oriented programming language like C++ would be perfectly suited to meet with these requirements, however no C++ compiler was available for the transputer system, so the modularity had to be reflected on an object code level, using header files for the declaration of global interface variables and procedures.

The definition of the training problems follows the same strategy. Since the problem size is a variable in many performance measurements, the usage of explicit training files including the definition of a certain network topology and the corresponding training patterns would lead to many similar definition files varying in just one parameter (e.g. the number of input nodes) and new network options might require numerous file updates.

In order to avoid this, all simulation parameters including network definitions and problem sizes are passed to the programs as command line option and different test problems are defined by a small C-file which can interpret its own command line options and then define the network topology and the training set. These definition files are linked to the rest of the program without requiring its recompilation.

6.2 Sample Problems

All sample problems deal with 2 layer networks of standard backpropagation neurones and use Boolean logic. Since the aim of this project lies mainly in quantitative comparisons, all problems are abstract, well defined and can be adapted for arbitrary problem sizes.

The variables n, m and o refer to the number of input, hidden and output nodes of an n-m-o-layer network. p is the number of training patterns $(I^{(k)}, O^{(k)})$. The problem parameters are set by upper-case command line options of the corresponding programs; if omitted, the standard values given in the right column are assumed. Only the problem specific options are given here; for general genetic, backpropagation, sequential and parallel options, please refer to the corresponding sections.

The program names of the sequential backpropagation version have the postfix `-back`, the sequential and parallel versions of the genetic and the combined algorithm end in `-seq` and `-par`.

6.2.1 N-M-N Encoder/Decoder

Source: `enc.c` problem definition

Programs:
`encseq` sequential genetic implementation
`encpar` parallel genetic implementation
`encback` sequential backpropagation implementation

Options: `-N`n number of input and output nodes $n = 3$
`-M`m number of hidden nodes $m = \lceil \mathrm{ld} n \rceil$

An n-m-n encode/decoder reproduces input unit vectors at the output by finding a a compressed binary intermediate representation in the hidden layer.

$$p = n, \quad I_i^{(k)} = O_i^{(k)} = \delta_{jk}$$

The encoder/decoder problem allows big networks to be trained by a relatively small training set since the number of training patters is equal to the number of nodes.

6.2.2 1-Norm of a Vector

Source: `cnt.c` problem definition

Programs: `cntseq` sequential genetic implementation
`cntpar` parallel genetic implementation
`cntback` sequential backpropagation implementation

Options: `-N`n number of input nodes $n = 3$
`-M`m number of hidden nodes $m = n$
`-O`o number of output nodes $o = \lceil \mathrm{ld} n \rceil$

Counts all input nodes set to 1 and produces the number binary encoded at the output. If the number of output nodes o is set to 1, then 1 will be produced, if an odd number of inputs is set to 1, and 0 will be produced otherwise. The 1-norm can thus be seen as a generalisation of the **n-parity problem**. In the case of 2 input and 1 output node, this results in the 2-parity or **exclusive or (XOR) problem**.

$$p = 2^n, \quad I^{(k)} = \mathrm{bin}_n k, \quad I^{(k)} = \mathrm{bin}_o \sum_{i=1}^{n} I_i^{(k)}$$

$$\text{with} \quad (\mathrm{bin}_b s)_j = \left\lfloor \frac{s}{2^{j-1}} \right\rfloor \bmod 1, \quad j = 1, \ldots b$$

The 1-norm and the n-parity problem have both highly nonlinear error functions and are therefor a good test for the robustness of the training algorithm.

6.2.3 2×N Comparator

Source: `cmp.c` problem definition

Programs: `cmpseq` sequential genetic implementation
`cmppar` parallel genetic implementation
`cmpback` sequential backpropagation implementation

Options: `-N`N length of one operand $N = 3$
`-M`m number of hidden nodes $m = 1 + \left\lfloor \frac{N}{3} \right\rfloor$
`-Q`q test for $=$ if $q \neq 0$, else test for $\leq$ $q = 0$

Compares the binary encoded operands a and b and sets the output node according to the operator determined by q.

$$p = 2^{2N}, \quad n = 2N, \quad o = 1, \quad I^{(k)} = \text{bin}_n k, \quad a_k = k \bmod 2^N, \quad b_k = \left\lfloor \frac{s}{2^N} \right\rfloor$$

$$O_1^{(k)} = \begin{cases} 1, & a_k \circ_{OP} b_k \\ 0, & \text{otherwise} \end{cases} \qquad \circ_{OP} = \begin{cases} \leq, & q = 0 \\ =, & q \neq 0 \end{cases}$$

The 2×N comparator is a relatively simple problem (at least for $q = 0$) which very large ($p = 2^{2N}$) and highly redundant training sets and is therefor a good test for online learning (Section 4.1.1) or error estimation (Section 3.4.3).

6.3 Program Modules

According to the modular programming concept, all executables are generated by linking together separate specialised modules (object files). A module without object file, thus merely containing definitions is called *virtual*. The following table lists all modules and shows their hierarchic dependencies. The files *problem*`.c` and *problem*`.o` refer to the source and object file of the problem definition (e.g. to `enc.c` and `enc.o`). For a more detailed description, please refer to the `Makefile`.

Module	File	Source	Using
Definitions	`defs.o`	`defs.c defs.h`	
Sequential	`seq.o`	`seq.c seq.h`	`defs.h`
Parallel	`par.o`	`par.c par.h`	`defs.h` CS-Tools
Simulation	`sim.o`	`sim.c sim.h`	`defs.h`
Individual	*virtual*	`ind.h`	`defs.h`
Standard Net	`stdnet.o`	`stdnet.c stdnet.h`	`defs.h ind.h`
Problem Def.	*problem*`.o`	*problem*`.c`	`defs.h ind.h stdnet.h`
Genetic Alg.	`gen.o`	`gen.c gen.h`	`defs.h ind.h gen.h`
Backpropagation	`back.o`	`back.c back.h`	`defs.h ind.h`
Gen. Backprop.	`genback.o`	`genback.c genback.h`	`defs.h ind.h back.h`
Main Sequential	`mainseq.o`	`mainseq.c`	`defs.h ind.h back.h seq.h gen.h genback.h`
Main Parallel	`mainpar.o`	`mainpar.c`	`defs.h ind.h back.h par.h gen.h genback.h`
Main Backprop.	`mainback.c`	`mainback.c`	`defs.h ind.h back.h sim.h`

6.3.1 Parameter Handling and Initialisation

Most modules require an initialisation to allocate arrays, to setup local variables or hardware, or to process command line options passed to the program. For this purpose, the following functions and variables are declared in the interface (i.e. the header file):

- `char *`*Mod*`OptStr();` returns a pointer to the option string of the module.
- `char *`*Mod*`Usage();` returns a pointer to the usage message of the module.
- `int handle`*Mod*`Opt(char opt,char* arg);` returns 0 if the option `opt` with the argument `arg` was successfully handled by the module.
- `int init`*Mod*`();` returns 0 if the Module has been successfully initialised.
- `char` *Mod*`ParamStr[256];` is set by `init`*Mod*`()` and contains a verbal description of the module parameters.

Mod stands for the module names `Seq`, `Par`, `Sim`, `Ind`, `Net` (also declared in `ind.h` for network initialisation), `Std` (declared in `stdnet.h` and implemented in the problem definition), `Gen`, `Back` and `Out` (defined by the main programs for output handling). These routines are either called directly form the main program or by the corresponding routines of the parent module reflecting the object oriented concept of inheritance.

In the following module descriptions, only variables and functions directly related to the implemented algorithms are mentioned. For auxiliary and system or hardware specific functions, please refer to the source code.

6.3.2 Module: Definitions

Source: defs.h declarations of standard types and functions
defc.c implementation

Programs: *all*

Options: *none*

Parent: *none*

This module contains commonly used constants, type declaration and auxiliary functions for bit manipulation and random numbers.

6.3.3 Module: Sequential

Source: `seq.h` declarations of sequential features
`seq.c` implementation

Programs: `encseq` sequential genetic ENC/DEC problem
`cntseq` sequential genetic 1-norm problem
`cmpseq` sequential genetic comparator

Options:

`-p`p	population size	p=100
`-g`N_{max}	maximum number of generations	$N_{max} = 100000$
`-e`E_{max}	maximum error for success	$E_{max} = 0.01$
`-s`s_{rnd}	random seed value	$s_{rnd} =$ time

Parent: *none*

This module contains all functions, specific to the sequential execution of the genetic and combined algorithm and is, like the module *Simulation* a mere placeholder, since no network functions or special initialisations have to be defined.

Variables and Functions

- `int PopSize;` population size p
- `int MaxGen;` maximum number N_{max} of generations
- `errtyp MaxErr;` maximum error E_{max} for success
- `int SeedRand;` random seed value s_{rnd}

- `int initSeq`; initialises the sequential parameters
- `void errorexit(char *msg)`; prints the error message `msg` and aborts the program

6.3.4 Module: Parallel

Source:	`defs.h`	declarations of parallel features
	`sefc.c`	implementation
Programs:	`encseq`	parallel genetic ENC/DEC problem
	`cntseq`	parallel genetic 1-norm problem
	`cmpseq`	parallel genetic comparator

Options:	`-p`p	population size	p=100
	`-g`N_{max}	maximum number of generations	$N_{max} = 100000$
	`-e`E_{max}	maximum error for success	$E_{max} = 0.01$
	`-s`s_{rnd}	random seed value	$s_{rnd} =$ time
	`-t`N_{tr}	global selection every N_{tr} generations	$N_{tr} = 1$

Parent: *none*

This module contains all functions specific to the parallel execution of the genetic and combined algorithm on the transputer network and uses the Meiko CS-Tools library.

Variables and Functions

- `int Procs`; number of processors (determined automatically)
- `int ProcId`; process Id (0 is master)
- `int PopGlobal`; global population size p
- `int PopLocal`; local population size p_{loc}
- `int MaxGen`; maximum number N_{max} of generations
- `errtyp MaxErr`; maximum error E_{max} for succes
- `int SeedRand`; random seed value s_{rnd}
- `int Ntrans`; global selection every N_{trans} generations
- `int initPar()`; initialises parallel parameters and communication network and communicates the random seed values.
- `void initNetwork()`; network setup

- send and receive functions for network communication
- `void errorexit(char *msg);` prints the error mesage `msg` and aborts all processes

6.3.5 Module: Simulation

Source: `defs.h` declarations of sequential backpropagation features
`sefc.c` implementation

Programs: `encback` backpropagation ENC/DEC problem
`cntback` backpropagation 1-norm problem
`cmpback` backpropagation comparator

Options:

`-g`N_{max}	maximum number of iterations	$N_{max} = 100000$
`-e`E_{max}	maximum error for success	$E_{max} = 0.01$
`-s`s_{rnd}	random seed value	$s_{rnd} =$ time

Parent: *none*

This module is the backpropagation equivalent to the module *Sequential* and is also merely a placeholder.

Variables and Functions

- `int MaxIter;` maximum number N_{max} of iterations
- `errtyp MaxErr;` maximum error E_{max} for success
- `int SeedRand;` random seed value s_{rnd}
- `int initSim();` initialises parameters

6.3.6 Module: Individual

Source: `ind.h` declaration of individual and network features

Programs: *all*

Options: *none*

Parent: *none*

This virtual module contains variable and function declaration for the individual definition in the genetic algorithm including network topology and training sets, which are also used for backpropagation. The actual implementation are left to the derived modules. This allows to use the genetic programs for any kind of phenotypes.

The actual implementations for 2-layer neural networks is left to the derived module *Standard Net*.

Declarations

- `int CrBits;` length l of one chromosome string in bits
- `int Nin, Nhid, Nout;` network topology for a 2-layer n-m-o-network
- `float **TrainIn, **TrainOut;` input and output patters $I^{(k)}$ and $O^{(k)}$ of the training set $\mathbf{T}$.
- `int Ntrain;` number of patterns t
- `int Nback;` number of backpropagation steps b for the combined genetic-backpropagation algorithm
- `int NoTrain;` number of training patters t' used for error estimation (Section 3.4.3)
- `float Estimate` the initial information ratio r_{est}, which is defined as $r_{est} = \frac{t'(n+o)}{m(n+1)+o(m+1)}$ and used to calculate t'.
- `int initInd();` and `int initNet();` initialise the individual or the network parameters
- `errtyp calcerr(ind x);` returns the error (i.e. the negative fitness) $E(x)$ of the individual `x`.
- `void printind(ind x);` prints a description of the phenotype of `x`.

6.3.7 Module: Standard Network

Source: `stdnet.h` declarations for 2-layer n-m-o-networks
`stdnet.c` implementation

Programs: *all*

Options:

`-B`B	number of bits B used for weight encoding	$B = 8$
`-W`W	weights in interval $[-W, W]$	$W = 10$
`-E`r_{est}	error estimation factor	$r_{est} = 0$
`-b`b	no. of backprop. steps for combined alg.	$b = 0$

Parent: *Individual*

This module contains the implementation for 2-layer networks of the functions declared in the module *Individual*. The actual topology, as well as the training set, is, however, determined by the problem definition via the "virtual" functions `initStd` and `initTrain`.

The individual options `-B`, `-W`, `-E` and `-b` are recognised by `initInd` and ignored, when only the backpropagation `initNet` is called.

Variables and Functions

- `int Nbits;` number of bits B used for weight encoding
- `float Width;` range W for weights in interval $[-W, W]$
- `int initStd();` and `void initTrain();` forward declarations for the actual problem definition. The interface variables `Nin`, `Nhid`, `Nout` and `Ntrain` must be set according to the problem.
- `void initTrain();` forward declarations for the actual problem definition. The arrays `TrainIn` and `TrainOut` must be set to the training patters of the problem.
- various auxiliary constants, macros and functions for decoding chromosome strings

6.3.8 Module: Genetic Algorithm

Source: `gen.h` declaration of the genetic algorithm
`gen.c` implementation

Programs: `encseq, cntseq, cmpseq` sequential versions
`encpar, cntpar, cmppar` parallel versions

Options:

`-m`p_m	percentage of mutations	$p_m = 60\%$
`-n`N_m	perform $(1, \ldots N)$-point mutations	$N_m = \left\lfloor \frac{l}{50} \right\rfloor + 1$
`-c`p_c	percentage of reproduction (copies)	$p_c = 0\%$
`-u`u	percentage of uniform selections	$u = 0\%$
`-d`d	decimation factor	$d = 0$
`-h`H	size of internal hashtable	$H = 0$

Parent: *none*

This modules contains the variables and functions for the genetic algorithm. The chromosome strings (type `ind`) are arrays of unsigned integers; all genetic operators work on this representation by directly manipulating the bits.

Variables and Functions

- `int Ncopy, Nmutate, Ncrossover` the numbers r, m and c of individuals on which to perform reproduction, mutation and crossover (Section 3.5.2)
- `int Decimation;` initial decimation factor d (Section 3.3.2)

- `ind *Pop`; array of the chromosome strings S_k of the population $\mathbf{P}$.
- `errtyp *Err`; array of the errors $E(S_k)$ of all individuals in $\mathbf{P}$
- `int initGen(int popsize,int popmem)`; initialises parameters and generates $\mathbf{P}_0$ by randomising strings.
- `void randomPop(int popsize)`; generates `popsize` random individuals.
- `void mutate(ind x0,ind x1)`; perform n-point mutation on `x0` where n is a random number in $\{1,\dots N_m\}$. The result is written to `x1`.
- `void crossover(ind x0,ind y0,ind x1,ind y1)`; performs crossover of `x0` and `y0`; the results are written to `x1` and `y1`.
- `void copy(ind x0,ind x1)`; copies `x0` onto `x1`.
- `void calcerrors(int popsize)`; calculates the error of all individuals using `calcerr` declared in the module *Individual*.
- `void selection(int popsize)`; generates the next generation, using the rank selection mechanism defined in Section 3.5.1.

6.3.9 Module: Backpropagation

Source: `back.h` declaration of the backpropagation algorithm
`back.c` implementation

Programs: *all*

Options:

`-o`O	use online learning (0/1)	$O = 1$
`-k`γ	learn rate	$\gamma = 1$
`-a`α	impulse factor	$\alpha = 0$
`-w`w	range of initial weights $[-w, w]$	w=1

Parent: *none*

This module contains the backpropagation algorithm as described in Section 4.3.

Variables and Functions

- `float NetErr`; residual Error $E(W^{(1)}, W^{(2)})$
- `float LearnRate`; learn rate γ
- `float Impulse`; impulse constant α
- `float InitWeight`; initial weight range w

- `float **Wih, **Who;` weight matrices $W^{(1)}$ and $W^{(2)}$
- `float **Dih,**Dho;` first order updates $\Delta W^{(1)}$ and $\Delta W^{(2)}$
- `float **DDih,**DDho;` second order updates $\Delta^2 W^{(1)}$ and $\Delta^2 W^{(2)}$ (impulse term)
- `int initBack();` initialise parameters and weights
- `void randomNet(float w);` randomise weights (interval $[-w, w]$)
- `float calcNetErr();` calculates the average network error per pattern ($E(W^{(1)}, W^{(2)})$) and updates weights in case of online learning
- `void updateNet();` updates weights in case of batch learning
- `void printnet();` prints $W^{(1)}$ and $W^{(2)}$

6.3.10 Module: Genetic Backpropagation

Source: `back.h` declarations for the combined algorithm
`back.c` implementation

Programs: `encseq, cntseq, cmpseq` sequential versions
`encpar, cntpar, cmppar` parallel versions

Options: *none*

Parent: *none*

This module implements special function for the combined genetic backpropagation algorithm (Section 5.4).

Variables and Functions

- `void randomNetPop(int popsize,float w);` randomises a network population with initial weights in $[-w, w]$.
- `void gen2back(ind x);` decoding function d'_{net}
- `void back2gen(ind x);` encoding function e'_{net}
- `void backsteps(ind x,int n);` performs n backpropagation steps on the individual `x`.

6.3.11 Main Modules

Source:	`mainseq.c`	main program for the sequential genetic alg.
	`mainpar.c`	main program for the parallel genetic alg.
	`mainback.c`	main program for sequential backpropagation

Programs:	`encseq, cntseq, cmpseq`	sequential versions (`mainseq.c`)
	`encpar, cntpar, cmppar`	parallel versions (`mainpar.c`)
	`encback, cntback, cmpback`	backpropagation (`mainback.c`)

Options:	`-l`L	frequency of log-output	*automatic*
	`-f`F	log momentarily best network	$F = 0$
	`-i`I	show parameter info	$I = 1$
	`-r`R	show calculation statistics	$R = 1$

Parent: *none*

These modules contain the `main`-procedures for all executables. They mainly consist of a loop which performs the genetic or backpropagation iterations and produces log-output. The output options are the same for all three modules.

6.4 Example Session: Parallel XOR-Problem

The command

```
mrun xor.par
```

with the following configuration file `xor.par`

```
par processor 0 for 4 cntpar -N2 -O1 -p200 -b5 -e0.001
-f1 -l2 networkis ternarytree endpar
```

starts a parallel simulation of the XOR-problem (1-norm with `-N2` and `-O1`) on a ternary tree of 4 transputers, using the combined genetic backpropagation algorithm with a population size of $p = 200$ (`-p200`) and 5 backpropagation steps per generation (`-b10`). The simulation stops when the average error per pattern of the best individual is lower than 0.001 (`-e0.001`).

Log output is produced every second generation (`-l2`) and the resulting network is printed (`-f1`):

```
Simulation Parameters:
Network:    2 bit Counter (counts 1s in input) (4 patterns)
            Topology 2-2-1, 3 Neurons, 88 bits (Weights 8)
            Weights in [-10.00, 10.00]
            5 backpropagation steps per generation
Simulation: Procs = 4, PopSize = 200, MaxGen = 10000, MaxErr =  0.0010
```

```
            RandomSeed = 802499780
Genetic:    Pcopy = 0 %, Pmutate(1..2 pt.) = 60 %, Pcrossover = 40 %
            Selection 100 % linear, 0 % uniform
Backprop.:  Method: Batch, InitWeight = [ -1.000,  1.000]

Gen      1: t=      0, MinErr=  0.1242, AvgErr=  0.1335
Gen      2: t=      1, MinErr=  0.1225, AvgErr=  0.1330
Gen      4: t=      2, MinErr=  0.1193, AvgErr=  0.1335
Gen      6: t=      3, MinErr=  0.1087, AvgErr=  0.1286
Gen      8: t=      4, MinErr=  0.0890, AvgErr=  0.1252
Gen     10: t=      5, MinErr=  0.0574, AvgErr=  0.1134
Gen     12: t=      6, MinErr=  0.0202, AvgErr=  0.1022
Gen     14: t=      7, MinErr=  0.0090, AvgErr=  0.0684
Gen     16: t=      8, MinErr=  0.0040, AvgErr=  0.0378
Gen     18: t=      9, MinErr=  0.0014, AvgErr=  0.0252
Gen     20: t=     10, MinErr=  0.0012, AvgErr=  0.0195
Gen     21: t=     11, MinErr=  0.0006, AvgErr=  0.0160

Statistic                total    per sec.        time
--------------------------------------------------------
Generations:                21    1.909091    0.523810 s
Individuals global:       4200  381.818176    0.002619 s
Individuals local:        1050   95.454544    0.010476 s
Backprop. steps:         21000 1909.090942    0.000524 s
evaluated Patterns:      25200 2290.909180    0.436508 ms

(-8.75, 7.80,b:-3.18)( 3.88,-3.80,b:-1.53)
( 9.69, 9.14,b:-5.14)

program succeeded.
```

7 Performance

This section contains performance measurements of various aspects of the implemented algorithms as well as the conclusions for their practical use. Of course, only a small fraction of all test runs can be presented; every example was chosen to illustrate a different aspect of the relative performance of the algorithms used, and to highlight the advantages and disadvantages of every method.

7.1 Efficiency of Parallelisation

As shown in Section 5.3.2, the genetic algorithm provides a large scope for distributes memory parallelism (see also Fig. 2).

The following table gives the speedup S_p and the efficiency E_p as defined in Section 5.1.1 for three different network sizes of the genetic Encoder/Decoder problem using three different population sizes. The ENC/DEC problem was chosen because the number of training patterns t is equal to the number of input nodes n, which results in a computation/communication ratio r of order $O(n)$. For all other problems, r is of order $O(2^n)$ which would soon lead to efficiencies of nearly 100% and make comparisons difficult for greater network sizes.

The simulations were executed in parallel on 4, 8 and 12 processors (`encpar`) and the optained performance was compared to the sequential version `encseq` on a single transputer.

$\mathbf{E}_p$,$\mathbf{S}_p$		8-3-8 ENC/DEC			16-4-16 ENC/DEC			32-5-32 ENC/DEC		
pop.		100	1000	10000	100	1000	10000	100	1000	10000
4	S_4	3.22	3.61	3.56	3.88	3.88	3.83	3.90	3.99	3.93
	E_4	81%	90%	89%	97%	97%	96%	98%	100%	98%
8	S_8	2.61	5.91	6.00	5.30	7.05	7.21	6.88	7.76	7.66
	E_8	33%	74%	75%	66%	88%	90%	86%	97%	96%
12	S_12	2.72	6.29	7.68	6.12	9.69	9.69	9.75	10.92	11.13
	E_12	23%	52%	64%	51%	81%	81%	81%	91%	93%

The efficiency increases with the problem size, due to larger training sets an therefore longer error calculations. It also increases with larger population sizes which reduce the relative impact of the overhead involved for setting up the connections. Higher numbers of processes reduce efficiency, which can be explained by Amdahl's Law (Section 5.1.2) and by the fact, that the overhead for routing information between the master and the slaves increases with the number of transputers in the network. This explains the very high efficiency of the 4 transputer networks, because they require no routing at all.

7.2 The XOR-Problem

One of the most common test problems to evaluate the performance of network training algorithms is the XOR-problem. To allow algorithmic comparisons, only the sequential versions of the genetic (`cntseq -N2 -O1`) and the backpropagation algorithm (`cntback -N2 -O1`) have been used. (Since the training set consists only of the 4 patterns $((0,0),(0))$, $((0,1),(1))$, $((1,0),(1))$ and $((1,1),(0))$, a parallel algorithm on a shared memory system would be of rather low efficiency, anyway.)

For the following measurements, the problem was considered to be solved, when the average error per pattern is less than 0.001. A program fails, after 100000 backpropagation steps or 10000 generations.

7.2.1 Backpropagation

Four series of 10 simulations varying in learn method (online or batch) and impulse factor ($\alpha = 0$ or $\alpha = 0.9$) have been started and produced the following results:

γ	α	method	succeeded	avg. iterations	avg. time
1	0	batch	8/10	5188.0	9.0 s
1	0.9	batch	9/10	538.7	1.1 s
1	0	online	6/10	1294.8	2.2 s
1	0.9	online	7/10	138.0	0.3 s

This figures show the limitation of backpropagation or any other gradient descend method: One quarter of the started simulations failed i.e. got trapped in a local minimum of the error functions. Also, the data illustrates the effects of the impulse term with speedups about 900%, the value of the acceleration factor a (see Section 4.1.2). Online learning is about 4 times faster than batch learning, but leads to more than the double number of failures.

7.2.2 Genetic Algorithm

Six series of 10 simulations varying in the numbers of bits b used for the encoding of the weights have been started. The standard genetic parameters (60% mutation, 40% crossover, population size 100, linear rank selection) have been used.

p	succeeded	avg. gen.	avg. time
4	10/10	20.7	2.5 s
6	10/10	62.8	8.6 s
8	10/10	51.7	6.7 s
10	10/10	40.4	5.8 s
12	10/10	33.0	4.7 s

The most striking difference to backpropagation is, that all programs have succeeded, which illustrates the robustness of the genetic method, while backpropagation normally converges faster if it converges at all.

The impact of b shows, that a more accurate error function normally leads to better convergence, except for very short encodings, where the element of random search might outweigh this effect. Fig. 3 shows the error graph for two simulations with $b = 4$ and $b = 12$.

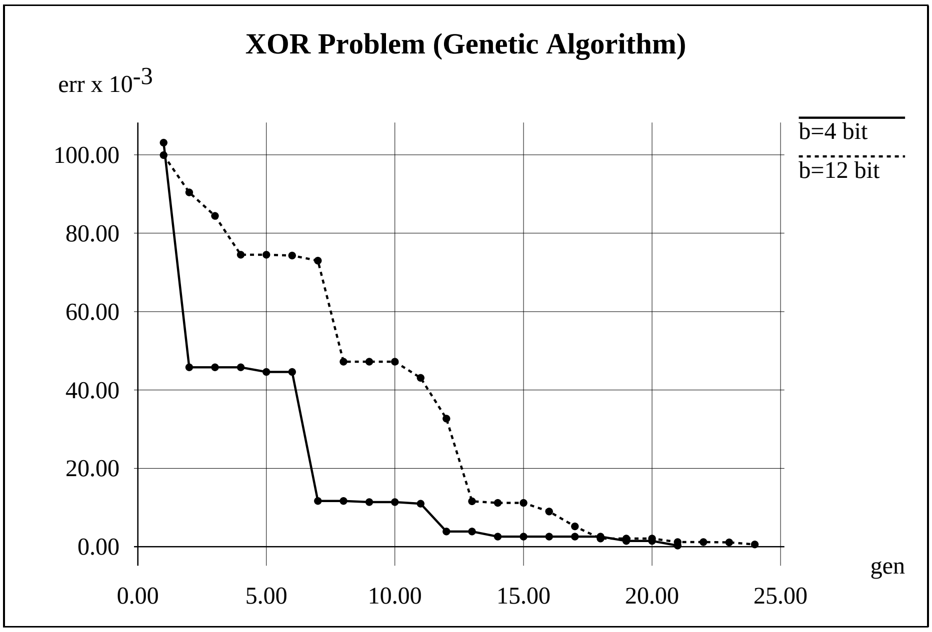

Figure 3: **Error Graph for the XOR Problem**

7.3 The 1-Norm Problem

To compare the performance of the backpropagation and the combined algorithm on large and highly nonlinear problems, both algorithms were used to train a network for the 1-Norm of a 7-dimensional binary vector to an error of 0.001. 4 runs of the sequential versions of both algorithms have been performed.

7.3.1 Backpropagation

All 4 programs used online learning at a learn rate of 1. The impulse constants 0 and 0.8 were used on two programs each. The simulations were started by the following commands.

```
cntback -N7 -O1 -e0.001 -f1 -g250000 -o1
cntback -N7 -O1 -e0.001 -f1 -g250000 -o1 -a0.8
```

Both simulations with $\alpha = 0.8$ failed, while the other two programs succeeded after 55097 and 45974 iterations (11661 s and 9800 s)

7.3.2 Genetic Backpropagation

The combined algorithm used a population size of 50 with 5 backpropagation steps per generation and was started by the following command

```
cntseq -N7 -O1 -e0.001 -f1 -g1000 -p50 -b5
```

All 4 programs succeeded after an average of 240 generations and an average computation time of 18797.5 s. The following weight matrices $W^{(1)}$ and $W^{(2)}$ were found by one of the programs:

$$W^{(1)} = \begin{pmatrix} 10.00 & 9.06 & -6.94 & 5.29 & 7.65 & -5.22 & -5.45 & -7.33 \\ 9.61 & -7.02 & -8.75 & -8.12 & -7.49 & -8.27 & 9.61 & 10.00 \\ 8.75 & -7.33 & -8.35 & 8.90 & -7.25 & -9.29 & -4.90 & 9.61 \\ -5.22 & -8.98 & -7.88 & -8.90 & -7.33 & 8.75 & 9.53 & 9.84 \\ 7.10 & 4.75 & -7.41 & 7.57 & -6.94 & -6.71 & -7.02 & 4.51 \\ 8.35 & 7.02 & -9.29 & -8.90 & -9.92 & -8.51 & 8.27 & 6.55 \\ 3.10 & -10.00 & -4.27 & -2.94 & -3.57 & -3.73 & 2.86 & 9.45 \end{pmatrix}$$

$$W^{(2)} = \begin{pmatrix} 5.22 & 10.00 & 8.67 & 6.16 & -6.16 & -9.29 & -10.00 & -2.31 \end{pmatrix}$$

Since the encoding range of the weights is by default defined as $[-10, 10]$, the occurrences of 10 and -10 indicate range overflows caused by the backpropagation steps.

Considering that the parallel version at this problem size ($t = 256$) would run with an efficiency of effectively 100%, the combined algorithm would definitely be the better choice for this problem.

7.4 The Comparator Problem

The comparator problem with the operator $\leq$ is, in comparison with the n-parity problem, very linear in the sense that its error function has no significant local minima. This property makes it a perfect choice for all gradient descend methods like backpropagation and will illustrate the performance gains that can be optained by adding backpropagation features to the standard genetic algorithm.

7.4.1 Backpropagation

To get some reference values for comparisons, two series of 10 sequential backpropagation programs (online learning with $\gamma = 1$), one of which used learning with impulse ($\alpha = 0.9$), were started on single transputers. The test problem was a 2×4-comparator with the operator $\leq$, the maximal error was 0.001. All programs succeeded and produced the following results ($\overline{x}$ refers to the arithmetic men, $\overline{\Delta x}$ to average deviation from $\overline{x}$ and $\widehat{x}$ to the geometric mean of the samples x_i):

impulse	iterations			time		
α	$\overline{N}$	$\overline{\Delta N}$	$\widehat{N}$	$\overline{T}$	$\overline{\Delta T}$	$\widehat{T}$
0	55.9	9.0	55.3	10.2 s	1.8 s	10.1 s
0.9	235.4	318.0	138.1	53.9 s	72.6 s	31.6 s

7.4.2 Genetic Backpropagation

Six series of 10 programs have been started, the first using the standard genetic algorithm with a population size p of 256, the other five using $b = 1$, 2, 4, 8 and 16 backpropagation steps per generation with proportional reduced population sizes of 256, 128, 64, 32 and 16. All simulations were run in their parallel versions on a 16 transputer ternary tree using the following configuration file:

```
par
processor 0 for 16 cmppar -N4 -B12 -e0.001 -g100 -pp -bb
networkis ternarytree
endpar
```

While all 10 programs with the standard genetic algorithm ($b = 0$) failed to train the network to the goal of $E \leq 0.001$ within 100 generations, all programs using the combined algorithm succeeded.

backprop. steps	pop. size	generations			time		
b	p	$\overline{N}$	$\overline{\Delta N}$	$\widehat{N}$	$\overline{T}$	$\overline{\Delta T}$	$\widehat{T}$
1	256	19.9	5.4	19.3	143.3 s	38.9 s	138.9 s
2	128	11.5	3.5	11.1	63.0 s	19.6 s	60.7 s
4	64	4.9	1.0	4.8	22.6 s	4.3 s	22.2 s
8	32	2.9	0.6	2.8	12.3 s	2.6 s	12.0 s
16	16	2.0	0.0	2.0	8.1 s	0.3 s	8.1 s

Fig. 4 shows the average computation time $\overline{T}$ in relation to the number of backpropagation steps in the combined algorithm. The two horizontal lines indicate the average times of the serial backpropagation programs.

While for sequential implementations, the standard backpropagation algorithm would certainly be the better choice for this very problem, the example shows that a parallel implementation of the combined algorithm with a sufficiently high number of backpropagation steps can nevertheless lead to faster execution times even if, as in this case, due to the high linearity of its error function, the problem is perfectly suited for a simple gradient descend method.

7.5 Conclusion

The given examples are only a small but representative selection of all test runs and measurements performed for this project, and illustrate the advantages and disadvantages of genetic methods for the training of neuronal networks. Theoretic analysis and the optained practical results suggest the following conclusions:

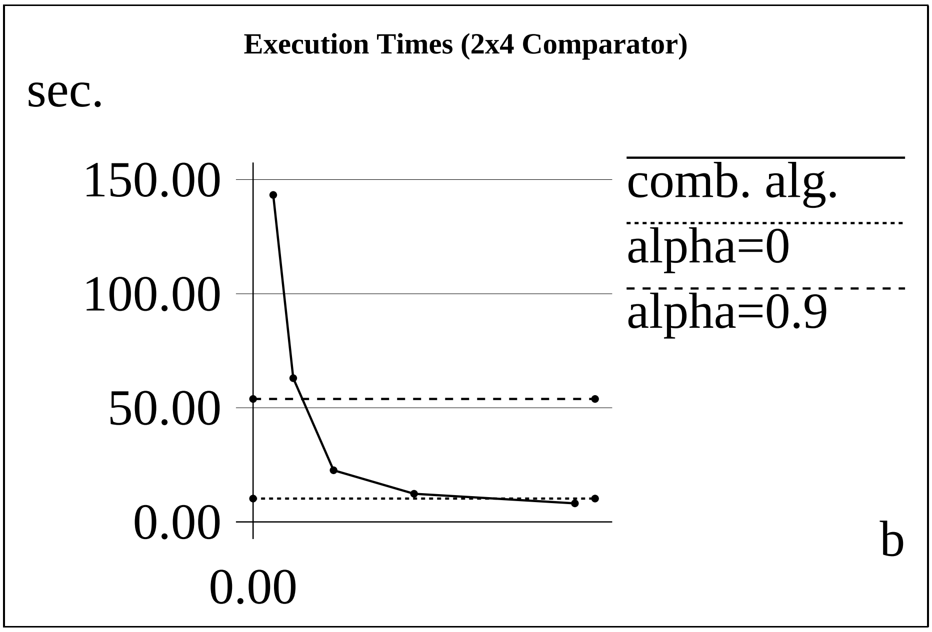

Figure 4: **Execution Time for 2×4 CMP**

7.5.1 Parallelism

- Despite its inherently parallel nature, an efficient parallel implementation of the backpropagation algorithm on distributed memory systems for small and medium network sizes is impossible due to its high interdependencies and therefore low computation/communication ratio.
- An efficient parallel implementation of the genetic algorithm on distributed memory systems is possible if the number of training patterns is sufficiently high.

7.5.2 Reliability

- Due to its stochastic nature, the standard genetic algorithm can handle highly nonlinear problems and usually finds the global minimum of the error function if the population is sufficiently large.
- Backpropagation is a gradient descend method and therefore very sensible to nonlinearities in the error function. Even with very small nonlinear problems like XOR, the algorithm may converge against a local minimum and fail.

7.5.3 Convergence Speed

- The necessary population size and number of generations for the standard genetic algorithm dramatically increase with the problem size and render the method impracticable for bigger networks.

- Provided that the problem is sufficiently linear, the backpropagation algorithm normally converges reasonably fast. However, the actual speed depends very much on the simulation parameters γ and α and on the initial weight values.

7.5.4 Genetic Backpropagation

- Unlike standard backpropagation, the combined genetic backpropagation algorithm is relatively robust against nonlinearities, can dynamically adapt learn and impulse rates and can efficiently be parallelised on distributed memory systems.

- Unlike the standard genetic algorithm, the required population sizes and the number of necessary generations are smaller, which allows the use of the combined algorithm for bigger networks and larger training sets.

References

[Rojas] Raúl Rojas. *1993 Theorie der Neuralen Netze [Theory of Neural Nets]*

[Gold] David E. Goldberg. *1989 Genetic Algorithms in Search, Optimization, and Machine Learning*

[Davis] Lawrence Davis. *1991 Handbook of Genetic Algorithms*

A Makefile

```

## Transputers (Bowes)
CC = mcc
CCOPT = -c -O
LNKOPT = -O -lm -lcs -lcsn
STDLNKOPT = -O
all: seq par back

## Multimax (Newton)
#CC = gcc
#CCOPT = -c -pedantic
#LNKOPT = -lm
#STDLNKOPT = -r -nostdlib
#all: seq back

seq: encseq cntseq cmpseq

par: encpar cntpar cmppar

back: encback cntback cmpback

# executables

STDPAR = mainpar.o par.o gen.o back.o genback.o stdnet.o defs.o
STDSEQ = mainseq.o seq.o gen.o back.o genback.o stdnet.o defs.o
STDBACK = mainback.o sim.o back.o genback.o stdnet.o defs.o

stdpar.o: mainpar.o par.o gen.o back.o genback.o stdnet.o defs.o

stdseq.o: mainseq.o seq.o gen.o back.o genback.o stdnet.o defs.o

stdback.o: mainback.o sim.o back.o genback.o stdnet.o defs.o

encpar: enc.o stdpar.o
        $(CC) $(STDPAR) enc.o -o encpar $(LNKOPT)

encseq: enc.o stdseq.o
        $(CC) $(STDSEQ) enc.o -o encseq $(LNKOPT)

encback: enc.o stdback.o
        $(CC) $(STDBACK) enc.o -o encback $(LNKOPT)

cntpar: cnt.o stdpar.o
        $(CC) $(STDPAR) cnt.o -o cntpar $(LNKOPT)

cntseq: cnt.o stdseq.o
        $(CC) $(STDSEQ) cnt.o -o cntseq $(LNKOPT)

cntback: cnt.o stdback.o
```

```
	$(CC) $(STDBACK) cnt.o -o cntback $(LNKOPT)

cmppar: cmp.o stdpar.o
	$(CC) $(STDPAR) cmp.o -o cmppar $(LNKOPT)

cmpseq: cmp.o stdseq.o
	$(CC) $(STDSEQ) cmp.o -o cmpseq $(LNKOPT)

cmpback: cmp.o stdback.o
	$(CC) $(STDBACK) cmp.o -o cmpback $(LNKOPT)

# object files

mainpar.o: mainpar.c gen.h back.h genback.h par.h ind.h defs.h
	$(CC) mainpar.c -o mainpar.o $(CCOPT)

mainseq.o: mainseq.c gen.h back.h genback.h seq.h ind.h defs.h
	 $(CC) mainseq.c -o mainseq.o $(CCOPT)

mainback.o: mainback.c back.h ind.h defs.h
	$(CC) mainback.c -o mainback.o $(CCOPT)

enc.o: enc.c stdnet.h ind.h defs.h
	$(CC) enc.c -o enc.o $(CCOPT)

cnt.o: cnt.c stdnet.h ind.h defs.h
	$(CC) cnt.c -o cnt.o $(CCOPT)

cmp.o: cmp.c stdnet.h ind.h defs.h
	$(CC) cmp.c -o cmp.o $(CCOPT)

gen.o: gen.c gen.h ind.h defs.h
	$(CC) gen.c -o gen.o $(CCOPT)

par.o: par.c par.h defs.h
	$(CC) par.c -o par.o $(CCOPT)

seq.o: seq.c seq.h defs.h
	$(CC) seq.c -o seq.o $(CCOPT)

sim.o: sim.c sim.h defs.h
	$(CC) sim.c -o sim.o $(CCOPT)

back.o: back.c back.h ind.h defs.h
	$(CC) back.c -o back.o $(CCOPT)

defs.o: defs.c defs.h
	$(CC) defs.c -o defs.o $(CCOPT)

stdnet.o: stdnet.c stdnet.h defs.h ind.h
	$(CC) stdnet.c -o stdnet.o $(CCOPT)
```

```

genback.o: genback.c genback.h back.h ind.h defs.h
	$(CC) genback.c -o genback.o $(CCOPT)

clear:
	rm -f *.o
	rm -f encpar encseq encback
	rm -f cntpar cntseq cntback
	rm -f cmppar cmpseq cmpback

```

B Source Code

B.1 Problem Definitions

B.1.1 File: enc.c

```

/* Implementaion of the Encode/Decoder-Problem */

#include "defs.h"
#include "ind.h"
#include "stdnet.h"

/* Parameter Handling */

#define DEFNIN          AUTO
#define DEFNHID         AUTO

#define MAXIN           0x7fff
#define MAXHID          15

char *StdOptStr() {return  "N:M:\0";}

char *StdUsage() {return
  "Network (N-M-N-ENC/DEC) Parameters:\n"
  "-N <no. of inputs/outputs>:       4\n"
  "-M <no. of hidden units>:         auto (>= ld M)\n\0";}

/* set default values */

int Nin   = DEFNIN;     /* no. of inputs */
int Nhid  = DEFNHID;    /* no. of hidden units */

int handleStdOpt(char opt,char* arg)
{
  switch(opt)
  {
    case 'N': return (Nin       =getint(arg,1,MAXIN))<0;
    case 'M': return (Nhid      =getint(arg,1,MAXHID))<0;
```

```
    default: return 1;
  };
}

int initStd()
{
  if(Nin==AUTO) Nin=4;
  Nout=Nin;
  if(Nhid==AUTO) Nhid=duallog(Nin);
  if(Nhid<duallog(Nin)) return 1;
  Ntrain=Nin;
  sprintf(StdName,"%d-%d-%d ENC/DEC",Nin,Nhid,Nout);
  return 0;
}

void initTrain()
{
  int p,i;

  for(p=0;p<Ntrain;p++)
    for(i=0;i<Nin;i++)
      TrainIn[p][i]=TrainOut[p][i]= p==i ? 1.0 : 0.0;
}
```

B.1.2 File: cnt.c

```

/* Implementation of 1-norm of a binary vector */
/* out[]=sum(i,in[i]) mod 2^O */

#include "defs.h"
#include "ind.h"
#include "stdnet.h"

/* Parameter Handling */

#define DEFNIN          3
#define DEFNHID         AUTO
#define DEFNOUT         AUTO

#define MAXIN           31
#define MAXHID          255
#define MAXOUT          5

char *StdOptStr() {return "N:M:O:\0";}

char *StdUsage() {return
  "Network (Counter):\n"
  "-N <no. of input units>:          3\n"
  "-M <no. of hidden units>:         auto (M)\n"
  "-O <no. of output units>:         auto (>= ld M)\n\0";}
```

```

int Nin=DEFNIN;
int Nhid=DEFNHID;
int Nout=DEFNOUT;

const int grey2dec[16]={0,1,3,2,7,6,4,5,15,14,12,13,8,9,11,10};

int handleStdOpt(char opt,char* arg)
{
  switch(opt)
  {
    case 'N': return (Nin      =getint(arg,1,MAXIN))<0;
    case 'M': return (Nhid     =getint(arg,1,MAXHID))<0;
    case 'O': return (Nout     =getint(arg,1,MAXOUT))<0;
  };
  return 1;
}

int initStd()
{
  if(Nhid==AUTO) Nhid=Nin;
  if(Nout==AUTO) Nout=duallog(Nin+1);
  Ntrain=1<<Nin;
  sprintf(StdName,"%d bit Counter (counts 1s in input)",Nin);
  return 0;
}

void initTrain()
{
  int p,z,i;

  for(p=0;p<Ntrain;p++)
  {
    z=0;
    for(i=0;i<Nin;i++)
      z+=TrainIn[p][i]=(float) ((p>>i) & 1);
    for(i=0;i<Nout;i++)
      TrainOut[p][i]=(float) ((z>>i) & 1);
  };
}

```

B.1.3 File: cmp.c

```

/* Implementation of 2*N-bit Comperator */

#include "defs.h"
#include "ind.h"
#include "stdnet.h"

```

```
/* Parameter Handling */

#define DEFNIN          3
#define DEFNHID         AUTO

#define MAXIN           8
#define MAXHID          255

char *StdOptStr() {return "N:M:\0";}

char *StdUsage() {return
  "Network (2xN Comperator) Parameters:\n"
  "-N <no. of input units>:          3\n"
  "-M <no. of hidden units>:         auto (M)\n"
  "-Q <train for = instead of =< ?>  0\n\0";}

int Nin=DEFNIN;
int Nhid=DEFNHID;
int Equality=0;

const int grey2dec[16]={0,1,3,2,7,6,4,5,15,14,12,13,8,9,11,10};
const char EqualityStr[2][40]={"less or equal","equal"};
const int stdNhid[17]={0,0,2,0,2,0,3,0,3,0,3,0,4,0,4,0,4};

int handleStdOpt(char opt,char* arg)
{
  switch(opt)
  {
    case 'N': return (Nin       =getint(arg,1,MAXIN))<0;
    case 'M': return (Nhid      =getint(arg,1,MAXHID))<0;
    case 'Q': return (Equality  =getint(arg,0,1))<0;
  };
  return 1;
}

int initStd()
{
  Nin*=2;
  if(Nhid==AUTO) Nhid=stdNhid[Nin];
  Nout=1;
  Ntrain=1<<Nin;
  sprintf(StdName,"2x%d bit Comperator (%s)",
    Nin/2,EqualityStr[Equality]);
  return 0;
}

void initTrain()
{
  int p,i;
  unsigned l,a,b,m;

```

```
  l=Nin/2; m=(1<<l)-1;
  for(p=0;p<Ntrain;p++)
  {
    a=p&m; b=(p>>l)&m;
    for(i=0;i<Nin;i++)
      TrainIn[p][i]=(float) ((p>>i) & 1);
    TrainOut[p][0]=(float)( Equality ? (a==b) : (a<=b) );
  };
}

```

B.2 Module: Definitions

B.2.1 File: defs.h

```

/* Common Definitions */

#ifndef DEFS_H
#define DEFS_H  1

/* data types */

#define errtyp          float
#define byte            unsigned char
#define word            unsigned int
#define ind             word*
#define state           int

/* data lengths */

#define INTLEN          sizeof(int)
#define ERRLEN          sizeof(int)
#define WORDLEN         sizeof(unsigned int)
#define INDLEN          sizeof(ind)
#define FLOATLEN        sizeof(float)
#define PNTLEN          sizeof(void*)

/* maximum values */

#define MAXERROR        0x7fffffff
#define MAXRANDOM       0x7fffffff

/* macros */

#define bitoffs(n)      ((n) & 31)
#define wordoffs(n)     ((n) >> 5)
#define getbit(x,i)     ((x[wordoffs(i)]>>bitoffs(i))&1)
#define min(x,y)        ((x)<(y) ? (x) : (y))
#define max(x,y)        ((x)>(y) ? (x) : (y))
#define getrand()       random()
```

```
#define seedrand(s)     srandom(s)
#define gettime()       time(0)

/* flags */

#define AUTO            (-1)
#define FAUTO           (-65536.0*65536.0)
#define NOENTRY         (-1)
#define UNDEF           (-1)

/* procedures */

word getbits(ind x,int i,int n);
word putbits(word w,ind x,int i,int n);
int duallog(int n);
int getint(char *s,int min,int max);
int getfloat(float *y,char *s,float min,float max);
int rounddec(int x);

#endif

```

B.2.2 File: defs.c

```

/* Common Procedures */

#include "defs.h"
#include "par.h"

/* isolate bitstring */

word getbits(ind x,int i,int n)
{
  byte *p;
  word *q;
  int  j,k;
  word m;

  j=bitoffs(i);
  k=wordoffs(i);
  m=(1 << n)-1;
  if(j<bitoffs(i+n))
    return (x[k] >> j) & m;
  else
    return ((x[k] >> j) | (x[k+1] << (32-j))) & m;
}

word putbits(word w,ind x,int i,int n)
{
  byte *p;
```

```
  word *q;
  int  j,k;
  word m;

  j=bitoffs(i);
  k=wordoffs(i);
  m=((1 << n)-1);
  x[k]&= ~(m << j); x[k]|=(w << j);
  if(j>=bitoffs(i+n))
  {
    x[k+1]&= ~(m >> (32-j));
    x[k+1]|= (w >> (32-j));
  };
}

/* dual logarithm, rounded up */

int duallog(int n)
{
  int l;

  for(l=0;(1<<l)<n;l++);
  return l;
}

int getfloat(float *y,char *s,float min,float max)
{
  float x;

  if(!s) return 1;
  switch(s[0])
  {
        case 0:   return 1;
        default:  sscanf(s,"%f",&x);
  };
  if(x<min || x>max) return 1;
  *y=x;
  return 0;
}

int getint(char *s,int min,int max)
{
  int n;

  if(!s) return -1;
  switch(s[0])
  {
        case '+': n=1;          break;
        case '-': n=0;          break;
        case 0:   n=1;          break;
        default:  n=atoi(s);
```

```
  };
  if(n<min || n>max) return(-1);
  return n;
}

int rounddec(int x)
{
  if(x<=1) return 1;
  if(x<=3) return 2;
  if(x<=7) return 5;
  return 10*rounddec((x+5)/10);
}
```

B.3 Module: Sequential

B.3.1 File: seq.h

```

/* Declaration of Sequential Features */

#ifndef SEQ_H
#define SEQ_H   1

#include "defs.h"

char *SeqOptStr();
char *SeqUsage();
char SeqParamStr[256];

int PopSize;    /* size of actual population */

int MaxGen;     /* Maximum no. of generations */
errtyp MaxErr;  /* Maximum error for succes */

int SeedRand;   /* random seed */

int handleSeqOpt(char opt,char* arg);
int initSeq();

void errorexit(char *msg);

#endif
```

B.3.2 File: seq.c

```

/* Implementation of sequential features */

#include "defs.h"
#include "seq.h"

```

```
/* Parameter Handling */

#define DEFPOPSIZE      100
#define DEFMAXGEN       10000
#define DEFMAXERR       0.01
#define DEFSEED         AUTO

char *SeqOptStr() {return "p:g:e:s:\0";}

char *SeqUsage() {return
  "Simulation Parameters:\n"
  "-p <population size>:            100\n"
  "-g <max. no. of generations>:    10000\n"
  "-e <max. error for succes>:      0.01\n"
  "-s <random seed value>:          time\n\0";}

/* set default values */

int PopSize     =DEFPOPSIZE;    /* size of actual population */
int MaxGen      =DEFMAXGEN;     /* Maximum no. of generations */
errtyp MaxErr   =DEFMAXERR;     /* Maximum error for succes */
int SeedRand    =DEFSEED;       /* random seed */

int handleSeqOpt(char opt,char* arg)
{
  switch(opt)
  {
    case 'p': return (PopSize   =getint(arg,2,1000000))<0;
    case 'g': return (MaxGen    =getint(arg,1,1000000000))<0;
    case 'e': return getfloat(&MaxErr,arg,0,1000000);
    case 's': return (SeedRand  =getint(arg,0,MAXRANDOM))<0;
    default: return 1;
  };
}

void errorexit(char *msg)
{
  printf(msg);
  exit(1);
}

int initSeq()
{
  if(SeedRand==AUTO) SeedRand=gettime();
  seedrand(SeedRand);

  sprintf(SeqParamStr,
    "Simulation: serial, PopSize = %d, MaxGen = %d, "
    "MaxErr = %7.4f\n            RandomSeed = %d\n",
    PopSize,MaxGen,MaxErr,SeedRand);
  return 0;
```

```
}
```

B.4 Module: Parallel

B.4.1 File: par.h

```

/* Declaration of Parallel Features */

#ifndef PAR_H
#define PAR_H   1

#include "defs.h"

#define MAXPROCS        16

char *ParOptStr();
char *ParUsage();
char ParParamStr[256];

int Procs;      /* no. of processes */
int ProcId;     /* process Id */
int LocalId;    /* Local Id */

int PopGlobal;  /* size of global population */
int PopLocal;   /* size of local population */

int MaxGen;     /* Maximum no. of generations */
errtyp MaxErr;  /* Maximum error for succes */

int SeedRand;   /* random seed */
int Ntrans;     /* frequency of data transfer */

void initNetwork();
int handleParOpt(char opt,char* arg);
int initPar();

void errorexit(char *msg);
void send(int proc,void *p,int len);
void recv(void *p,int len);
int recvfrom(void *p,int len);
void setvect(int n,void *p,int len);
void sendvect(int proc,int n);
void recvvect(int n);
int recvvectfrom(int n);

#endif
```

B.4.2 File: par.c

```
/* Implementation of parallel features */

#include <csn/csn.h>
#include <csn/names.h>
#include <csn/csnuio.h>

#include "defs.h"
#include "par.h"

/* Parameter Handling */

#define DEFPOPSIZE      100
#define DEFMAXGEN       10000
#define DEFMAXERR       0.01
#define DEFSEED         AUTO
#define DEFTRANS        1

char *ParOptStr() {return "p:g:e:s:t:\0";}

char *ParUsage() {return
  "Simulation Parameters:\n"
  "-p <population size>:             100\n"
  "-g <max. no. of generations>:     10000\n"
  "-e <max. error for succes>:       0.01\n"
  "-s <random seed value>:           time\n"
  "-t <frequency of data transfers>  1\n\0";}

/* set default values */

int PopGlobal   =DEFPOPSIZE;    /* size of actual population */
int MaxGen      =DEFMAXGEN;     /* Maximum no. of generations */
errtyp MaxErr   =DEFMAXERR;     /* Maximum error for succes */
int SeedRand    =DEFSEED;       /* random seed */
int Ntrans      =DEFTRANS;      /* frequency of data transfers */

/* Network variables */

Transport Tr;
netid_t NetId[MAXPROCS];
char Name[MAXPROCS][2];
struct iovec Vector[CSN_MAX_IOVCNT];

int handleParOpt(char opt,char* arg)
{
  switch(opt)
  {
    case 'p': return (PopGlobal =getint(arg,2,1000000))<0;
    case 'g': return (MaxGen    =getint(arg,1,1000000000))<0;
    case 'e': return getfloat(&MaxErr,arg,0,1000000);
    case 's': return (SeedRand  =getint(arg,0,MAXRANDOM))<0;
    case 't': return (Ntrans    =getint(arg,1,10000))<0;
```

```
    default: return 1;
  };
}

void errorexit(char *msg)
{
  cs_abort(msg,-1);
}

void send(int proc,void *p,int len)
{
  if(csn_tx(Tr,0,NetId[proc],(char*)p,len)!=len)
    errorexit("send error\n");
}

void recv(void *p,int len)
{
  netid_t from;

  if(csn_rx(Tr,&from,(char*)p,len)!=len)
    errorexit("receive error\n");
}

int recvfrom(void *p,int len)
{
  netid_t from;
  int i;

  if(csn_rx(Tr,&from,(char*)p,len)!=len)
    errorexit("receive error\n");
  for(i=0;i<Procs;i++)
    if(i!=ProcId && NetId[i]==from) return i;
  errorexit("sender not found\n");
  return -1;
}

void setvect(int n,void *p,int len)
{
  if(n>=CSN_MAX_IOVCNT)
    errorexit("too many vectors\n");
  Vector[n].iov_base=(caddr_t)p;
  Vector[n].iov_len=len;
}

void sendvect(int proc,int n)
{
  csn_txv(Tr,0,NetId[proc],Vector,n);
}

void recvvect(int n)
{
```

```
  netid_t from;

  csn_rxv(Tr,&from,Vector,n);
}

int recvvectfrom(int n)
{
  netid_t from;
  int i;

  csn_rxv(Tr,&from,Vector,n);
  for(i=0;i<Procs;i++)
    if(i!=ProcId && NetId[i]==from) return i;
  errorexit("sender not found\n");
  return -1;
}

void initNetwork()
{
  csn_init();
  cs_getinfo(&Procs,&ProcId,&LocalId);
}

int initPar()
{
  int i,k,n,h;
  word *p;
  ind* pop;

  if(csn_open(-1,&Tr)!=CSN_OK)
    errorexit("Can't open transport.\n");
  for(i=0;i<Procs;i++)
  {
    Name[i][0]='A'+i;
    Name[i][1]=0;
  };
  if(csn_registername(Tr,Name[ProcId])!=CSN_OK)
    errorexit("Can't register name\n");
  for(i=0;i<Procs;i++)
  {
    if(i==ProcId)
      NetId[i]=CSN_NULL_ID;
    else
      if(csn_lookupname(&NetId[i],Name[i],1)!=CSN_OK)
        errorexit("Can't find process name\n");
  };
  PopLocal=(PopGlobal-1)/Procs+1;
  PopGlobal=PopLocal*Procs;
  if(ProcId==0)
  {
    if(SeedRand==AUTO) SeedRand=gettime();
```

```
    for(i=1;i<Procs;i++)
      send(i,&SeedRand,WORDLEN);
  } else {
    recv(&SeedRand,WORDLEN);
    SeedRand+=ProcId;
  };
  seedrand(SeedRand);

  MaxGen=((MaxGen-1)/Ntrans+1)*Ntrans;

  sprintf(ParParamStr,
    "Simulation: Procs = %d, PopSize = %d, MaxGen = %d, "
    "MaxErr = %7.4f\n            RandomSeed = %d\n",
    Procs,PopGlobal,MaxGen,MaxErr,SeedRand);
  return 0;
}
```

B.5 Module: Simulation

B.5.1 File: sim.h

```
/* Declarations of simulation features (backpropagation) */

#ifndef SIM_H
#define SIM_H   1

#include "defs.h"

char *SimOptStr();
char *SimUsage();
char SimParamStr[256];

float MaxErr;   /* Maximum error for succes */
int MaxIter;    /* Maximum no. of iterations */
int SeedRand;   /* random seed */

int handleSimOpt(char opt,char* arg);
int initSim();

#endif
```

B.5.2 File: sim.c

```
/* Declarations of simulation features (backpropagation) */

#include "defs.h"
#include "sim.h"

/* Parameter Handling */

#define DEFMAXITER      100000
```

```
#define DEFMAXERR       0.01
#define DEFSEED         AUTO

char *SimOptStr() {return "g:e:s:\0";}

char *SimUsage() {return
  "Simulation Parameters:\n"
  "-g <max. no. of iterations:>:     10000\n"
  "-e <max. error for succes>:       0.01\n"
  "-s <random seed value>:           time\n\0";
}

/* set default values */

int MaxIter     =DEFMAXITER;    /* Maximum no. of generations */
float MaxErr    =DEFMAXERR;     /* Maximum error for succes */
int SeedRand    =DEFSEED;       /* random seed */

int handleSimOpt(char opt,char* arg)
{
  switch(opt)
  {
    case 'g': return (MaxIter   =getint(arg,1,1000000000))<0;
    case 'e': return getfloat(&MaxErr,arg,0,1000000);
    case 's': return (SeedRand  =getint(arg,0,MAXRANDOM))<0;
    default: return 1;
  };
}

int initSim()
{
  if(SeedRand==AUTO) SeedRand=gettime();
  seedrand(SeedRand);

  sprintf(SimParamStr,
    "Simulation: MaxIter = %d, MaxErr = %7.4f, Seed = %d\n",
    MaxIter,MaxErr,SeedRand);
  return 0;
}
```

B.6 Module: Individual

B.6.1 File: ind.h

```

/* Declarations of individual-specific functions */

#ifndef IND_H
#define IND_H   1
```

```
#include "defs.h"

int ptrain;

/* Genetic */

char *IndOptStr();
char *IndUsage();
char IndParamStr[256];

int CrBits;     /* length of one chromosome in bits */
int CrBytes;    /* length of one chromosome in bytes */
int CrWords;    /* length of one chromosome in words */

int OffsTrain;          /* Offset of error estimate */
int NoTrain;            /* No. of patterns of err. estim. */
float Estimate;         /* initial information ratio */

int handleIndOpt(char opt,char* arg);
int initInd();

int GenCalcs;

errtyp calcerr(ind x);
void printind(ind x);

/* Backpropagation */

char *NetOptStr();
char *NetUsage();
char NetParamStr[256];

int Nin;        /* no. of inputs */
int Nhid;       /* no. of hidden units */
int Nout;       /* no. of outputs */

float **TrainIn;        /* Training Set Input */
float **TrainOut;       /* Training Set Output */
int Ntrain;             /* no. of patterns */

int Nback;      /* no. of backprop. steps for combined moethod */

int BackCalcs;

int handleNetOpt(char opt,char* arg);
int initNet();

#endif
```

B.7 Module: Standard Network

B.7.1 File: stdnet.h

```

/* Declarations for standard 2 layer networks */

#ifndef STDNET_H
#define STDNET_H

#include "defs.h"

char *StdOptStr();
char *StdUsage();
char StdName[80];

/* These and Nin,Nhid,Nout,Ntrain are to be defined */
/* in initStd() */

int Nbits;      /* bits for bias and weights (gen.alg.) */
float Width;    /* weights in [-Width,Width] */

int handleStdOpt(char opt,char* arg);
int initStd();
void initTrain();

/* Standard Decoding */

int OffW1,OffW2,IncW1,IncW2,OffLR,OffIM;
float FGmult,FGadd;
int FGmax;

float *grey2float;
float *grey2lrate;
float *grey2imp;
word *int2grey;

int initDecoding();

#define Nlrate  8
#define Nimp    8
#define Llrate  0.02
#define Hlrate  20.0
#define Limp    0.0
#define Himp    0.95

#define decode(x,i,n) grey2float[getbits(x,i,n)]
#define weight1(x,i,h) decode(x,IncW1*h+Nbits*i,Nbits)
#define weight2(x,h,o) decode(x,OffW2+IncW2*o+Nbits*h,Nbits)
#define bias1(x,h) weight1(x,Nin,h)
#define bias2(x,o) weight2(x,Nhid,o)
#define lrate(x) grey2lrate[getbits(x,OffLR,Nlrate)]
```

```
#define imp(x) grey2imp[getbits(x,OffIM,Nimp)]
#define float2int(x) ((int)(x*FGmult+FGadd))

#define sigma(y)        1/(1+exp(-(y)))

#endif

```

B.7.2 File: stdnet.c

```

/* Implementation of standard 2 layer networks */

#include <math.h>

#include "defs.h"
#include "ind.h"
#include "stdnet.h"
#include "genback.h"

char IndStdOptStr[40]="B:W:E:b:\0";
char NetStdOptStr[40]="\0";

int   Nbits     =8;
float Width     =10.0;
float Estimate  =0;
int Nback       =0;

char *IndStdUsage=
  "-B <no. of bits for weights>      8\n"
  "-W <interval for weights [-W,W]>: 10\n"
  "-E <error estimation factor>:     0.0\n"
  "-b <no. of backprop. iterations>: 0\n\0";

char *NetStdUsage="\0";

char* IndOptStr()
{
  strcat(IndStdOptStr,StdOptStr());
  return IndStdOptStr;
}
char* NetOptStr()
{
  strcat(NetStdOptStr,StdOptStr());
  return NetStdOptStr;
}
char* IndUsage()
{
  char *s;

  s=(char*)malloc(512);
```

```
  strcpy(s,StdUsage());
  strcat(s,IndStdUsage);
  return s;
}
char* NetUsage()
{
  char *s;

  s=(char*)malloc(512);
  strcpy(s,StdUsage());
  strcat(s,NetStdUsage);
  return s;
}

int handleIndOpt(char opt,char* arg)
{
  switch(opt)
  {
    case 'B': return (Nbits     =getint(arg,1,24))<0;
    case 'W': return getfloat(&Width    ,arg,0.01,100);
    case 'E': return getfloat(&Estimate ,arg,0.1,100);
    case 'b': return (Nback     =getint(arg,0,100000))<0;
  };
  return handleStdOpt(opt,arg);
}

int handleNetOpt(char opt,char* arg)
{
  return handleStdOpt(opt,arg);
}

float *grey2float= (float*)0;
float *grey2lrate= (float*)0;
float *grey2imp=   (float*)0;
word  *int2grey=   (word*)0;

float *Hidden=(float*)0;

float **TrainIn=  (float**)0;   /* Training Set Input */
float **TrainOut= (float**)0;   /* Training Set Output */

int OffW1,OffW2,IncW1,IncW2,OffLR,OffIM;

void intgrey(word *a,int n)
{
  int i;
  int sum,bit,par;

  for(i=0;i<n;i++)
  {
    bit=n>>1; sum=0; par=0;
```

```
    while(bit)
    {
      if(bit&i) par=!par;
      if(par) sum|=bit;
      bit>>=1;
    };
    a[sum]=i;
  };
}

void greyfloat(float *a,int n,float l,float h)
{
  int i;
  int sum,bit,par;

  for(i=0;i<n;i++)
  {
    bit=n>>1; sum=0; par=0;
    while(bit)
    {
      if(bit&i) par=!par;
      if(par) sum|=bit;
      bit>>=1;
    };
    a[i]=(float)sum*(h-l)/(n-1)+l;
  };
}

int getTrainSet()
{
  int i,j,k;
  float *p,*q;

  if(TrainIn) return 0;
  if(!(TrainIn=(float**)malloc(Ntrain*PNTLEN))) return 1;
  if(!(TrainOut=(float**)malloc(Ntrain*PNTLEN))) return 1;
  if(!(p=(float*)malloc((Nin+1)*Ntrain*FLOATLEN))) return 1;
  if(!(q=(float*)malloc(Nout*Ntrain*FLOATLEN))) return 1;
  for(i=0;i<Ntrain;i++)
  {
    TrainIn[i]=p; p+=(Nin+1);
    TrainOut[i]=q; q+=Nout;
    TrainIn[i][Nin]=1.0;
  };
  initTrain();
  for(i=0;i<4*Ntrain;i++)
  {
    j=getrand()%Ntrain; k=getrand()%Ntrain;
    p=TrainIn[k]; q=TrainOut[k];
    TrainIn[k]=TrainIn[j]; TrainOut[k]=TrainOut[j];
    TrainIn[j]=p; TrainOut[j]=q;
```

```
  };
  return 0;
}

int initDecoding()
{
  int i;

  OffW1=0;
  IncW1=(Nin+1)*Nbits;
  OffW2=IncW1*Nhid;
  IncW2=(Nhid+1)*Nbits;
  OffLR=CrBits-Nlrate-Nimp;
  OffIM=CrBits-Nimp;
  FGmax=(1<<Nbits)-1;
  FGmult=0.5*(float)FGmax/Width;
  FGadd=0.5*(float)FGmax+0.5;
  if(!(grey2float=(float*)malloc((1<<Nbits)*FLOATLEN))) return 1;
  greyfloat(grey2float,1<<Nbits,-Width,Width);

  if(Nback)
  {
    if(int2grey) return 0;
    if(!(int2grey=(word*)malloc((1<<Nbits)*WORDLEN))) return 1;
    intgrey(int2grey,1<<Nbits);
    if(!(grey2lrate=(float*)malloc((1<<Nlrate)*FLOATLEN))) return 1;
    greyfloat(grey2lrate,1<<Nlrate,log(Llrate),log(Hlrate));
    for(i=0;i<(1<<Nlrate);i++) grey2lrate[i]=exp(grey2lrate[i]);
    if(!(grey2imp=(float*)malloc((1<<Nimp)*FLOATLEN))) return 1;
    greyfloat(grey2imp,1<<Nimp,Limp,Himp);
  };
  return 0;
}

int initInd()
{
  char s[80]="\0";
  char t[80]="\0";

  initStd();
  if(!(Ntrain && Nin && Nhid && Nout)) return 1;

  CrBits=(Nin+1)*Nbits*Nhid+(Nhid+1)*Nbits*Nout;
  if(Nback) CrBits+= Nlrate+Nimp;
  CrBytes=(CrBits-1)/8+1;
  CrWords=(CrBytes-1)/WORDLEN+1;

  if(initDecoding()) return 1;
  if(getTrainSet()) return 1;

  if(!(Hidden=(float*)malloc(Nhid*FLOATLEN))) return 1;
```

```

  NoTrain=Ntrain;
  OffsTrain=0;

  if(Nback)
    sprintf(t,
     "            %d backpropagation steps per generation\n",Nback);
  if(Estimate!=0.0)
  {
    NoTrain=(Estimate*((Nin+1)*Nhid+(Nhid+1)*Nout))/(Nin+Nout)+1;
    NoTrain=min(NoTrain,Ntrain);
    sprintf(s,
      "Estimation: Initial factor %4.1f, %d patterns (%d %%) used\n",
      Estimate,NoTrain,(NoTrain*100)/Ntrain);
  };
  sprintf(IndParamStr,
    "Network:    %s (%d patterns)\n"
    "            Topology %d-%d-%d, %d Neurons, %d bits (Weights %d)\n"
    "            Weights in [%6.2f,%6.2f]\n%s%s",
    StdName,Ntrain,Nin,Nhid,Nout,Nhid+Nout,CrBits,Nbits,
    -Width,Width,t,s);
  return 0;
}

errtyp calcerr(ind x)
{
  int p,pp,i,j,k;
  float e,s,d;

  if(Nback)
    backsteps(x,Nback);
  p=OffsTrain; e=0.0;
  for(pp=0;pp<NoTrain;pp++)
  {
    for(j=0;j<Nhid;j++)
    {
      s=0.0;
      for(i=0;i<Nin;i++)
        s+=weight1(x,i,j)*TrainIn[p][i];
      Hidden[j]=sigma(s+bias1(x,j));
    };
    for(k=0;k<Nout;k++)
    {
      s=0.0;
      for(j=0;j<Nhid;j++) s+=weight2(x,j,k)*Hidden[j];
      d=sigma(s+bias2(x,k))-TrainOut[p][k];
      e+=d*d;
    };
    p++; if(p>=Ntrain) p=0;
  };
  GenCalcs+=NoTrain;
```

```
  return (0.5*e)/((float)NoTrain);
}

void printind(ind x)
{
  int i,j,k,c;

  c=0;
  for(j=0;j<Nhid;j++)
  {
    if(c>70-6*Nin) { printf("\n"); c=0; };
    printf("(");
    for(i=0;i<Nin;i++) printf("%5.2f,",weight1(x,i,j));
    printf("b:%5.2f)",bias1(x,j));
    c+=9+6*Nin;
  };
  printf("\n"); c=0;
  for(k=0;k<Nout;k++)
  {
    if(c>70-6*Nhid) { printf("\n"); c=0; };
    printf("(");
    for(j=0;j<Nhid;j++) printf("%5.2f,",weight2(x,j,k));
    printf("b:%5.2f)",bias2(x,k));
    c+=9+6*Nhid;
  };
  printf("\n");
}

/* Backpropagation */

int initNet()
{
  int i,j;
  float *p,*q;

  initStd();
  if(getTrainSet()) return 1;
  Nback=0;

  sprintf(NetParamStr,
    "Network:    %s\n"
    "            Topology %d-%d-%d, %d Neurons, %d patterns\n",
    StdName,Nin,Nhid,Nout,Nhid+Nout,Ntrain);
  return 0;
}
```

B.8 Module: Genetic Algorithm

B.8.1 File: gen.h

```
/* Declarations of general genetic algorithms */

#ifndef GEN_H
#define GEN_H   1

#include "defs.h"

char *GenOptStr();
char *GenUsage();
char GenParamStr[256];

/* CONSTANTS */

int Pcopy;      /* % of indiv. to copy per gen. */
int Pmutate;    /* % of indiv. to mutate per gen.*/
int Pcrossover; /* % of indiv. to crossover per gen. */

int Ncopy;      /* no. of indiv. to copy per gen. */
int Nmutate;    /* no. of indiv. to mutate per gen.*/
int Ncrossover; /* no. of indiv. to crossover per gen. */

int Decimation; /* initial decimation factor */
float DecErrMin;
float DecErrAvg;

int HashLen;    /* lenth of hashkey 9n bits */
int HashSize;   /* size of hashtable */

/* Variables set by calcerr */

int Nunique;    /* no. of different indiv. */
int Nredundant; /* no. of redundant indiv. */
int Nmismatch;  /* no. of mismatches */

float ErrMin;
float ErrAvg;
int TopInd;

/* Arrays */

ind    *Pop;    /* Poulation */
errtyp *Err;    /* errorlist of Populaton */

/* Procedures */

int handleGenOpt(char opt,char* arg);
int initGen(int popsize,int popmem);
void randomPop(int popsize);

int hashfct(ind x);
void clearerrtab();
```

```

errtyp geterr(int n);
void mutate(ind x0,ind x1);
void crossover(ind x0,ind y0,ind x1,ind y1);
void copy(ind x0,ind x1);
void calcerrors(int popsize);
void selection(int popsize);

#endif
```

B.8.2 File: gen.c

```

/* Implementation of general genetic algorithms */

#include "defs.h"
#include "gen.h"
#include "ind.h"

/* Parameter Handling */

#define DEFMUTATE       60
#define DEFNMUTATE      AUTO
#define DEFCOPY         0
#define DEFDECIMATION   0
#define DEFHASH         AUTO

char *GenOptStr() {return "m:n:c:u:d:h:\0";}

char *GenUsage() {return
  "Genetic Parameters:\n"
  "-m <percentage of mutations>:     60\n"
  "-n <n for 1..n point mutations>:  auto\n"
  "-c <percentage of copies>:        0\n"
  "-u <perc. of uniform selections>: 0\n"
  "-d <initial decimation factor>:   0\n"
  "-h <size of hashtable (0=off)>:   auto\n\0";}

/* set default values */

int Pcopy       =DEFCOPY;       /* % of indiv. to copy per gen. */
int Pmutate     =DEFMUTATE;     /* % of indiv. to mutate per gen.*/
int Puniform    =0;             /* % of indiv. for uniform selection */
int Nmutate     =DEFNMUTATE;    /* 1..Nmutate point mutations */
int Decimation  =DEFDECIMATION; /* initial decimation factor */
int HashLen     =DEFHASH;       /* lenth of hashkey 9n bits */
int HashSize    =0;             /* size of hashtable */
int Nunique     =UNDEF;         /* no. of different indiv. */
int Nredundant  =UNDEF;         /* no. of redundant indiv. */
int Nmismatch   =UNDEF;         /* no. of mismatches */

```

```
ind *OldPop;    /* old population */
int HashMask;   /* HashSize-1 */
int *HashTab;
int Nuniform;

int handleGenOpt(char opt,char* arg)
{
  switch(opt)
  {
    case 'm': return (Pmutate   =getint(arg,0,100))<0;
    case 'n': return (Nmutate   =getint(arg,1,1000))<0;
    case 'c': return (Pcopy     =getint(arg,0,100))<0;
    case 'u': return (Puniform  =getint(arg,0,100))<0;
    case 'd': return (Decimation=getint(arg,0,100))<0;
    case 'h': return (HashLen   =getint(arg,0,16))<0;
    default: return 1;
  };
}

int initGen(int popsize,int popmem)
{
  int i,j,k,n,h,t;
  word *p;
  ind* pop;
  char s[128];
  errtyp e,m;

  GenCalcs=0;
  Pcrossover=100-Pmutate-Pcopy;
  if(Pcrossover<0) return 1;
  Nuniform=Puniform*(MAXRANDOM/100);

  if(HashLen==AUTO) HashLen=Pmutate>50 ? 0 : duallog(popsize)+1;
  if(HashLen)
  {
    if(popsize>(1<<HashLen)) return 1;
    HashSize=1 << HashLen;
    HashMask=HashSize-1;
    HashTab=(int*)malloc(HashSize*WORDLEN);
  };
  if(!(Pop=(ind*)malloc(popmem*INDLEN))) return 1;
  if(!(p=(word*)malloc(popmem*CrWords*WORDLEN))) return 1;
  for(i=0;i<popmem;i++) Pop[i]=p+i*CrWords;
  if(!(OldPop=(ind*)malloc(popmem*INDLEN))) return 1;
  if(!(p=(word*)malloc(popmem*CrWords*WORDLEN))) return 1;
  for(i=0;i<popmem;i++) OldPop[i]=p+i*CrWords;
  if(!(Err=(errtyp*)malloc(popmem*ERRLEN))) return 1;
  if(Decimation<=1) Decimation=0;
  if(Decimation)
  {
    h=HashLen; HashLen=0;
```

```
    DecErrAvg=0; DecErrMin=MAXERROR;
    k=(popsize-1)/Decimation+1; n=0;
    while(n<popsize)
    {
      randomPop(popsize);
      calcerrors(popsize);
      if(DecErrMin>ErrMin) DecErrMin=ErrMin;
      DecErrAvg+=ErrAvg;
      for(i=0;i<k && n<popsize;i++)
      {
        t=0;
        for(j=1;j<popsize;j++)
          if(Err[j]<Err[t]) t=j;
        copy(Pop[t],OldPop[n++]);
        Err[t]=MAXERROR;
      };
    };
    pop=Pop; Pop=OldPop; OldPop=pop;
    DecErrAvg/=Decimation;
    HashLen=h;
  } else {
    randomPop(popsize);
  };
  if(Nmutate==AUTO) Nmutate=CrBits/50+1;

  s[0]=0;
  if(HashLen) sprintf(s,", HashSize = %d",HashSize);
  sprintf(GenParamStr,
    "Genetic:    Pcopy = %d %%, Pmutate(1..%d pt.) = %d %%, "
    "Pcrossover = %d %%%s\n"
    "            Selection %d %% linear, %d %% uniform\n",
    Pcopy,Nmutate,Pmutate,Pcrossover,s,100-Puniform,Puniform);
  if(Decimation)
  {
    sprintf(s,"Decimation: Factor = %d, PopSize = %d, "
      "MinErr=%6.2f, AvgErr=%6.2f\n",
      Decimation,popmem*Decimation,DecErrMin,DecErrAvg);
    strcat(GenParamStr,s);
  };
  return 0;
}

void randomPop(int popsize)
{
  int i,j;
  word m;
  byte *p;

  p=(byte*)Pop[0];
  for(i=0;i<popsize*CrWords*WORDLEN;i++) p[i]=getrand() & 0xff;
  j=CrBits % (8*WORDLEN);
```

```
  if(CrBits%(WORDLEN*8))
  {
    m=(1<<j)-1;
    for(i=0;i<popsize;i++)
      Pop[i][CrWords-1] &= m;
  };
}

void clearerrtab()
{
  int i;

  for(i=0;i<HashSize;i++) HashTab[i]=NOENTRY;
  Nunique=Nredundant=Nmismatch=0;
}

int hashfct(ind x)
{
  int i;
  word w=0;

  for(i=0;i<CrWords;i++) w+=x[i];
  w+=w>>HashLen;
  w+=w>>HashLen;
  w+=w>>HashLen;
  return (int)(w & HashMask);
}

errtyp geterr(int n)
{
  int i,j,h,k;
  word w;
  errtyp e;

  h=hashfct(Pop[n]);
mismatch:
  k=HashTab[h];
  if(k==NOENTRY)
  {
    e=Err[n]=calcerr(Pop[n]);
    HashTab[h]=n;
    Nunique++;
    return e;
  } else {
    for(i=0;i<CrWords;i++)
      if(Pop[n][i]!=Pop[k][i])
      {
        h=(h+1)&HashMask;
        Nmismatch++;
        goto mismatch;
      };
```

```
    e=Err[n]=Err[k];
    Nredundant++;
    return e;
  };
}

void mutate(ind x0,ind x1)
{
  int i,k,m;
  for(i=0;i<CrWords;i++) x1[i]=x0[i];
  m=getrand()%Nmutate;
  for(i=0;i<=m;i++)
  {
    k=getrand()%CrBits;
    x1[wordoffs(k)]^=1<<bitoffs(k);
  };
}

void crossover(ind x0,ind y0,ind x1,ind y1)
{
  int n,i,k;
  word w,m,xx0,xx1,yy0,yy1;

  n=getrand()%CrBits; n=67;
  k=wordoffs(n);
  for(i=0;i<k;i++)
  {
    x1[i]=y0[i]; y1[i]=x0[i];
  };
  m=(1<<bitoffs(n))-1;

  xx0=m & (x0[k]); xx1=(~m) & (x0[k]);
  yy0=m & (y0[k]); yy1=(~m) & (y0[k]);

  x1[k]=yy0|xx1;
  y1[k]=xx0|yy1;

  for(i=k+1;i<CrWords;i++)
  {
    x1[i]=x0[i]; y1[i]=y0[i];
  };

}

void copy(ind x0,ind x1)
{
  int i;

  for(i=0;i<CrWords;i++) x1[i]=x0[i];
}
```

```
void calcerrors(int popsize)
{
  int i;
  errtyp e,sum,min;

  sum=0; min=MAXERROR;
  if(HashLen)
  {
    clearerrtab();
    for(i=0;i<popsize;i++)
    {
      e=geterr(i);
      sum+=e;
      if(e<min) { min=e; TopInd=i; };
    };
  } else {
    for(i=0;i<popsize;i++)
    {
      e=Err[i]=calcerr(Pop[i]);
      sum+=e;
      if(e<min) { min=e; TopInd=i; };
    };
  };
  ErrMin=(float)min;
  ErrAvg=((float)sum)/((float)popsize);
}

ind selectind(int popsize)
{
  int p,q,n;

  if(getrand()>=Nuniform)
  {
    p=getrand()%popsize;
    q=getrand()%popsize;
    return (Err[p]<=Err[q]) ? OldPop[p] : OldPop[q];
  } else {
    return OldPop[getrand()%popsize];
  };
}

void selection(int popsize)
{
  int n,i,j;
  ind *p;

  int ncopy,nmutate,ncrossover;

  ncopy=(popsize*Pcopy)/100;
  nmutate=(popsize*Pmutate)/100;
  ncrossover=popsize-ncopy-nmutate;
```

```
  if(ncopy==0)
  {
    ncopy=1;
    if(nmutate) nmutate--; else ncrossover--;
  }
  if(ncrossover&1) { ncrossover--; nmutate++; };

  p=Pop; Pop=OldPop; OldPop=p; i=0;
  copy(OldPop[TopInd],Pop[i++]);
  for(j=1;j<ncopy;j++)
    copy(selectind(popsize),Pop[i++]);
  for(j=0;j<ncrossover;j+=2)
   crossover(selectind(popsize),selectind(popsize),Pop[i++],Pop[i++]);
  while(i<popsize)
    mutate(selectind(popsize),Pop[i++]);
}

```

B.9 Module: Backpropagation

B.9.1 File: back.h

```

/* Declarations of backpropagation algorithm */

#ifndef BACK_H
#define BACK_H  1

#include "defs.h"

char *BackOptStr();
char *BackUsage();
char BackParamStr[256];

float NetErr;           /* residual Error */
float LearnRate;        /* learn rate */
float Impulse;          /* impulse constant */
float InitWeight;       /* initial weight range */

/* Arrays */

float **Wih;    /* weight matrix input -> hidden */
float **Who;    /* weight matrix hidden -> output */
float *Hid;     /* hidden states */
float *Out;     /* output states */
float *BiasHid; /* bias of hidden neurons */
float *BiasOut; /* bias of output neurons */
float **Dih,**Dho;      /* Weight update (1st order) */
float **DDih,**DDho;    /* Weight update (2nd order) */

/* Procedures */
```

```
int handleBackOpt(char opt,char* arg);
int initBack();
void randomNet(float w);
float calcNetErr();
void updateNet();
void printnet();

#endif
```

B.9.2 File: back.c

```

/* Implementation of backpropagation algorithm */

#include <math.h>

#include "defs.h"
#include "back.h"
#include "ind.h"

/* Parameter Handling */

#define DEFLEARNRATE    1.0
#define DEFIMPULSE      0.0
#define DEFINITWEIGHT   1.0
#define DEFONLINE       1

char *BackOptStr() {return "o:w:k:a:\0";}

char *BackUsage() {return
  "Backpropagation Parameters:\n"
  "-o <use online learning ?>:       1\n"
  "-w <w for initial weights [-w,w]> 1.0\n"
  "-k <learn rate>:                  1.0\n"
  "-a <impulse factor>:              0.0\n\0";}

/* set default values */

int Online      =DEFONLINE;     /* Use online learning */
float LearnRate =DEFLEARNRATE;  /* learn rate */
float Impulse   =DEFIMPULSE;    /* impulse constant */
float InitWeight=DEFINITWEIGHT; /* initial weight range */

/* others */

float *Dout;            /* Delta update for output layer */

const char OnlineStr[2][10]={"Batch","Online"};

int handleBackOpt(char opt,char* arg)
```

```
{
  switch(opt)
  {
    case 'o': return (Online    =getint(arg,0,1))<0;
    case 'w': return getfloat(&InitWeight,arg,0.0,1000.0);
    case 'k': return getfloat(&LearnRate,arg,0.0,1000.0);
    case 'a': return getfloat(&Impulse,arg,0.0,1000.0);
    default: return 1;
  };
}

int initBack()
{
  int i,j,k;
  float *p,*q,*pp,*qq;

  BackCalcs=0;
  if(!(Hid=(float*)malloc((Nin+1)*FLOATLEN))) return 1;
  if(!(Out=(float*)malloc(Nout*FLOATLEN))) return 1;
  if(!(Dout=(float*)malloc(Nout*FLOATLEN))) return 1;
  if(!(Wih=(float**)malloc((Nin+1)*PNTLEN))) return 1;
  if(!(Who=(float**)malloc((Nhid+1)*PNTLEN))) return 1;
  if(!(Dih=(float**)malloc((Nin+1)*PNTLEN))) return 1;
  if(!(Dho=(float**)malloc((Nhid+1)*PNTLEN))) return 1;
  if(!(p=(float*)malloc((Nin+1)*Nhid*FLOATLEN))) return 1;
  if(!(q=(float*)malloc((Nhid+1)*Nout*FLOATLEN))) return 1;
  if(!(pp=(float*)malloc((Nin+1)*Nhid*FLOATLEN))) return 1;
  if(!(qq=(float*)malloc((Nhid+1)*Nout*FLOATLEN))) return 1;
  for(i=0;i<=Nin;i++)
  {
    Wih[i]=p; p+=Nhid;
    Dih[i]=pp; pp+=Nhid;
    for(j=0;j<Nhid;j++) Dih[i][j]=0.0;
  };
  BiasHid=Wih[Nin];
  for(j=0;j<=Nhid;j++)
  {
    Who[j]=q; q+=Nout;
    Dho[j]=qq; qq+=Nout;
    for(k=0;k<Nout;k++) Dho[j][k]=0.0;
  };
  randomNet(InitWeight);
  BiasOut=Who[Nhid];
  Hid[Nhid]=1.0;

  if(Nback || Impulse!=0.0)
  {
    if(!(DDih=(float**)malloc((Nin+1)*PNTLEN))) return 1;
    if(!(DDho=(float**)malloc((Nhid+1)*PNTLEN))) return 1;
    if(!(p=(float*)malloc((Nin+1)*Nhid*FLOATLEN))) return 1;
    if(!(q=(float*)malloc((Nhid+1)*Nout*FLOATLEN))) return 1;
```

```
    for(i=0;i<=Nin;i++) { DDih[i]=p; p+=Nhid; };
    for(j=0;j<=Nhid;j++) { DDho[j]=q; q+=Nout; };
  };
  if(Nback)
    sprintf(BackParamStr,
      "Backprop.:  Method: %s, InitWeight = [%7.3f,%7.3f]\n",
      OnlineStr[Online],-InitWeight,+InitWeight);
  else
    sprintf(BackParamStr,
      "Backprop.:  Method: %s, LearnRate = %7.4f, Impulse = %7.4f\n"
      "             InitWeight = [%7.3f,%7.3f]\n",
      OnlineStr[Online],LearnRate,Impulse,
      -InitWeight,+InitWeight);
  return 0;
}

#define randweight()    (((float)getrand()/(float)MAXRANDOM) \
                        *2.0*w-w)

void randomNet(float w)
{
  int i,j,k;

  for(i=0;i<=Nin;i++)
    for(j=0;j<Nhid;j++) Wih[i][j]= randweight();
  for(j=0;j<=Nhid;j++)
    for(k=0;k<Nout;k++) Who[j][k]= randweight();
}

#undef randweight

#define sigma(x)        1/(1+exp(-(x)))

float evaluate(float *in,float *out)
{
  int i,j,k;
  float s,e,d;

  for(j=0;j<Nhid;j++)
  {
    s=0.0;
    for(i=0;i<=Nin;i++) s+=in[i]*Wih[i][j];
    Hid[j]=sigma(s);
  };
  for(k=0;k<Nout;k++)
  {
    s=0.0;
    for(j=0;j<=Nhid;j++) s+=Hid[j]*Who[j][k];
    Out[k]=sigma(s);
  };
  e=0.0;
```

```
  for(k=0;k<Nout;k++)
  {
    d=(out[k]-Out[k]);
    Dout[k]=d*Out[k]*(1-Out[k]);
    e+=d*d;
  };
  return 0.5*e;
}

#undef sigma

#define swap(x,y)       {pt=x; x=y; y=pt;}

float calcNetErr()
{
  int i,j,k,p;
  float **pt;
  float *in,*out;
  float s;
  NetErr=0.0;

  BackCalcs+=Ntrain;
  if(Online)
  {
    for(p=0;p<Ntrain;p++)
    {
      if(Impulse!=0) { swap(Dih,DDih); swap(Dho,DDho); };
      in=TrainIn[p]; out=TrainOut[p];
      NetErr+=evaluate(in,out);
      for(j=0;j<Nhid;j++)
      {
        s=0.0;
        for(k=0;k<Nout;k++) s+=Who[j][k]*Dout[k];
        s*=Hid[j]*(1-Hid[j]);
        if(Impulse!=0.0)
          for(i=0;i<=Nin;i++)
            Wih[i][j]+=Dih[i][j]=
              LearnRate*s*in[i]+Impulse*DDih[i][j];
        else
          for(i=0;i<=Nin;i++)
            Wih[i][j]+=LearnRate*s*in[i];
      };
      for(j=0;j<=Nhid;j++)
      {
        if(Impulse!=0.0)
        {
          for(k=0;k<Nout;k++)
            Who[j][k]+=Dho[j][k]=
              LearnRate*Dout[k]*Hid[j]+Impulse*DDho[j][k];
        } else {
          for(k=0;k<Nout;k++)
```

```
      Who[j][k]+=LearnRate*Dout[k]*Hid[j];
    };
   };
  };
  NetErr/=Ntrain;
  return NetErr;
 } else {
  if(Impulse!=0) { swap(Dih,DDih); swap(Dho,DDho); };

  for(i=0;i<=Nin;i++)
   for(j=0;j<Nhid;j++) Dih[i][j]=0.0;
  for(j=0;j<=Nhid;j++)
   for(k=0;k<Nout;k++) Dho[j][k]=0.0;

  for(p=0;p<Ntrain;p++)
  {
   in=TrainIn[p]; out=TrainOut[p];
   NetErr+=evaluate(in,out);
   for(j=0;j<Nhid;j++)
   {
    s=0.0;
    for(k=0;k<Nout;k++) s+=Who[j][k]*Dout[k];
    s*=Hid[j]*(1-Hid[j]);
    for(i=0;i<=Nin;i++)
     Dih[i][j]+=s*in[i];
   };
   for(j=0;j<=Nhid;j++)
    for(k=0;k<Nout;k++)
     Dho[j][k]+=Dout[k]*Hid[j];
  };
  NetErr/=Ntrain;
  return NetErr;
 };
}

#undef swap

void updateNet()
{
 int i,j,k;
 float l;

 if(!Online)
 {
  l=LearnRate/Ntrain;
  if(Impulse!=0.0)
  {
   for(i=0;i<=Nin;i++)
    for(j=0;j<Nhid;j++)
    {
     Dih[i][j]*=l;
```

```
          Wih[i][j]+=Dih[i][j]+=Impulse*DDih[i][j];
        };
      for(j=0;j<=Nhid;j++)
        for(k=0;k<Nout;k++)
        {
          Dho[j][k]*=l;
          Who[j][k]+=Dho[j][k]+=Impulse*DDho[j][k];
        };
    } else {
      for(i=0;i<=Nin;i++)
        for(j=0;j<Nhid;j++) Wih[i][j]+=Dih[i][j]*=l;
      for(j=0;j<=Nhid;j++)
        for(k=0;k<Nout;k++) Who[j][k]+=Dho[j][k]*=l;
    };
  };
}

void printnet()
{
  int i,j,k,c;

  c=0;
  for(j=0;j<Nhid;j++)
  {
    if(c>70-6*Nin) { printf("\n"); c=0; };
    printf("(");
    for(i=0;i<Nin;i++) printf("%5.2f,",Wih[i][j]);
    printf("b:%5.2f)",Wih[Nin][j]);
    c+=9+6*Nin;
  };
  printf("\n"); c=0;
  for(k=0;k<Nout;k++)
  {
    if(c>70-6*Nhid) { printf("\n"); c=0; };
    printf("(");
    for(j=0;j<Nhid;j++) printf("%5.2f,",Who[j][k]);
    printf("b:%5.2f)",Who[Nhid][k]);
    c+=9+6*Nhid;
  };
  printf("\n");
}
```

B.10 Module: Genetic Backpropagation

B.10.1 File: genback.h

```

/* Definitions for Genetic Backpropagation */

#ifndef GENBACK_H
#define GENBACK_H       1
```

```

#include "defs.h"

void randomNetPop(int popsize,float w);
void gen2back(ind x);
void back2gen(ind x);
void backsteps(ind x,int n);

#endif

```

B.10.2 File: genback.c

```

/* Implementations for Genetic Backpropagation */

#include "defs.h"
#include "ind.h"
#include "gen.h"
#include "back.h"
#include "stdnet.h"
#include "genback.h"

void randomNetPop(int popsize,float w)
{
  int i;

  for(i=0;i<popsize;i++)
  {
    randomNet(w);
    back2gen(Pop[i]);
  };
}

void gen2back(ind x)
{
  int i,j,k;
  for(i=0;i<=Nin;i++)
    for(j=0;j<Nhid;j++)
      { Wih[i][j]=weight1(x,i,j); DDih[i][j]=0.0; };
  for(j=0;j<=Nhid;j++)
    for(k=0;k<Nout;k++)
      { Who[j][k]=weight2(x,j,k); DDho[j][k]=0.0; };
  LearnRate=lrate(x);
  Impulse=imp(x);
}

void back2gen(ind x)
{
  int i,j,k;
  int w,m;
```

```
  float s;

  m=(1<<Nbits)-1;
  s=((1<<(Nbits-1))-1)/Width;
  for(i=0;i<=Nin;i++)
    for(j=0;j<Nhid;j++)
    {
      w=float2int(Wih[i][j]);
      if(w<0) w=0; else if(w>FGmax) w=FGmax;
      putbits(int2grey[w],x,IncW1*j+Nbits*i,Nbits);
    };
  for(j=0;j<=Nhid;j++)
    for(k=0;k<Nout;k++)
    {
      w=float2int(Who[j][k]);
      if(w<0) w=0; else if(w>FGmax) w=FGmax;
      putbits(int2grey[w],x,OffW2+IncW2*k+Nbits*j,Nbits);
    };
}

void backsteps(ind x,int n)
{
  int i;

  gen2back(x);
  for(i=0;i<n;i++)
  {
    calcNetErr();
    updateNet();
  };
  back2gen(x);
}

```

B.11 Main Modules

B.11.1 File: mainseq.c

```

/* Main file sequential */

#include "defs.h"
#include "seq.h"
#include "ind.h"
#include "gen.h"
#include "back.h"
#include "genback.h"

#define DEFLOG          AUTO
#define DEFFIRST        0
#define DEFINFO         1
```

```
#define DEFSTAT         1

extern char *optarg;
extern int optind;

char *OutOptStr() {return "l:f:i:r:\0";}

char *OutUsage() {return
  "Output Parameters:\n"
  "-l <frequency of log-output>:     auto\n"
  "-f <show first indiv. (0/1)>:    0\n"
  "-i <show parameter info>:         1\n"
  "-r <show statistic (0/1/2)>:     1\n\0";}

/* set default values */

int OptLog      =DEFLOG;
int OptFirst    =DEFFIRST;
int OptInfo     =DEFINFO;
int OptStat     =DEFSTAT;

int Gen;        /* current generation */
int StartTime;
int EndTime;

#define SimTime()       (gettime()-StartTime)

int Calcs;
int DecCalcs;
float CalcsPerGen;
float Diversity;

int handleOutOpt(char opt,char* arg)
{
  switch(opt)
  {
    case 'l': OptLog    =getint(arg,0,1000000000);      return 0;
    case 'f': OptFirst  =getint(arg,0,2);               return 0;
    case 'i': OptInfo   =getint(arg,0,1);               return 0;
    case 'r': OptStat   =getint(arg,0,2);               return 0;
    default: return 1;
  };
}

int initOut()
{
  StartTime=gettime();
  DecCalcs=PopSize*Decimation;
  Calcs=0;

  if(OptLog==AUTO)
```

```
  {
    if(MaxGen<=10)
      OptLog=1;
    else if(Ntrain)
      OptLog=rounddec(500000/((1+Nback)*
        ((1+Nin+Nout)*(1+Nhid)*NoTrain*PopSize)));
    else
      OptLog=rounddec(500000/((1+Nback)*
        ((1+Nin+Nout)*(1+Nhid)*Nin*PopSize)));
  };
  return 0;
}

void printinfo()
{
  printf("Simulation Parameters:\n%s%s%s%s\n",
        IndParamStr,SeqParamStr,GenParamStr,BackParamStr);
}

void printlog()
{
  printf("Gen%7d: t=%7d, MinErr=%8.4f, AvgErr=%8.4f",
        Gen,SimTime(),ErrMin,ErrAvg,ptrain);
  if(HashLen) printf(", Nuni=%6d, Nred=%6d, Nmis=%6d",
        Nunique,Nredundant,Nmismatch);
  printf("\n");
  if(OptFirst==2) printind(Pop[TopInd]);
}

void printstat()
{
  float t,g,i,b,p;
  t=SimTime(); if(t==0.0) t=1.0;
  g=Gen;
  i=g*PopSize;
  b=i*Nback;
  p= Nback ? GenCalcs+BackCalcs : GenCalcs;
  printf("\n"
    "Statistic               total    per sec.        time\n"
    "--------------------------------------------------------\n"
    "Generations:       %10.0f%12.6f%12.6f s\n"
    "Individuals:       %10.0f%12.6f%12.6f s\n",
    g,g/t,t/g,i,i/t,t/i);
  if(Nback) printf(
    "Backprop. steps:   %10.0f%12.6f%12.6f s\n",b,b/t,t/b);
  printf(
    "evaluated Patterns:%10.0f%12.6f%12.6f ms\n",p,p/t,1000*t/p);
}

int main(int argc,char **argv)
{
```

```
  int opt,i,j,k;
  char optstr[128]="\0";

  strcat(optstr,SeqOptStr());
  strcat(optstr,IndOptStr());
  strcat(optstr,GenOptStr());
  strcat(optstr,BackOptStr());
  strcat(optstr,OutOptStr());

  while((opt=getopt(argc,argv,optstr)) != -1)
  {
    if( handleSeqOpt(opt,optarg) &&
        handleIndOpt(opt,optarg) &&
        handleGenOpt(opt,optarg) &&
        handleBackOpt(opt,optarg) &&
         handleOutOpt(opt,optarg) )
      {
        printf("%s\n%s\n%s\n%s\n%s",
          SeqUsage(),IndUsage(),GenUsage(),
          BackUsage(),OutUsage());
        exit(1);
      };
  };

  if(initSeq()) errorexit("sequential init failed\n");
  if(initInd()) errorexit("individual init failed\n");
  if(initGen(PopSize,PopSize)) errorexit("genetic init failed\n");
  if(Nback && initBack()) errorexit("backpropagation init failed\n");
  if(initOut()) errorexit("output init failed\n");

  if(Nback) randomNetPop(PopSize,InitWeight);
  if(OptInfo) printinfo();

  for(Gen=1;Gen<=MaxGen;Gen++)
  {
    calcerrors(PopSize);
    Calcs+= HashLen ? Nunique : PopSize;
    if((OptLog && (Gen==1 || Gen%OptLog==0)) || (ErrMin<=MaxErr))
      printlog();
    if((ErrMin<=MaxErr) && (Ntrain==NoTrain))
    {
      CalcsPerGen=((float)Calcs)/((float)Gen);
      Diversity=100*CalcsPerGen/PopSize;
      if(OptStat) printstat();
      if(OptFirst==1) {printf("\n"); printind(Pop[TopInd]);};
      printf("\nprogram succeeded.\n");
      exit(0);
    };
    selection(PopSize);

    if((Ntrain>NoTrain) && (ErrMin*NoTrain<=MaxErr*Ntrain))
```

```
      NoTrain=min(Ntrain,NoTrain*2);
    OffsTrain++; if(OffsTrain>=Ntrain) OffsTrain=0;
  };
  if(OptStat==2) printstat();
  printf("\ntime out - program failed.\n");
  exit(0);
}
```

B.11.2 File: mainpar.c

```

/* Main file parallel */

#include <stdio.h>

#include "defs.h"
#include "par.h"
#include "ind.h"
#include "gen.h"
#include "back.h"
#include "genback.h"

#define DEFLOG          AUTO
#define DEFFIRST        0
#define DEFINFO         1
#define DEFSTAT         1

extern char *optarg;
extern int optind;

char *OutOptStr() {return "l:f:i:r:\0";}

char *OutUsage() {return
  "Output Parameters:\n"
  "-l <frequency of log-output>:     auto\n"
  "-f <show first indiv. (0/1)>:     0\n"
  "-i <show parameter info>:         1\n"
  "-r <show statistic (0/1/2)>:      1\n\0";}

/* set default values */

int OptLog      =DEFLOG;
int OptFirst    =DEFFIRST;
int OptInfo     =DEFINFO;
int OptStat     =DEFSTAT;

int Gen;        /* current generation */
int StartTime;
int EndTime;

#define SimTime()       (gettime()-StartTime)
```

```

int handleOutOpt(char opt,char* arg)
{
  switch(opt)
  {
    case 'l': OptLog    =getint(arg,0,1000000000);      return 0;
    case 'f': OptFirst  =getint(arg,0,2);               return 0;
    case 'i': OptInfo   =getint(arg,0,1);               return 0;
    case 'r': OptStat   =getint(arg,0,2);               return 0;
    default: return 1;
  };
}

int initOut()
{
  StartTime=gettime();

  if(OptLog==AUTO)
  {
    if(MaxGen<=10)
      OptLog=1;
    else if(Ntrain)
      OptLog=rounddec(500000/((1+Nback)*
        ((1+Nin+Nout)*(1+Nhid)*NoTrain*PopLocal)));
    else
      OptLog=rounddec(500000/((1+Nback)*
        ((1+Nin+Nout)*(1+Nhid)*Nin*PopLocal)));
  };
  if(OptLog%Ntrans) OptLog=(OptLog/Ntrans+1)*Ntrans;
  return 0;
}

void printinfo()
{
  printf("Simulation Parameters:\n%s%s%s%s\n",
        IndParamStr,ParParamStr,GenParamStr,BackParamStr);
}

void printlog()
{
  printf("Gen%7d: t=%7d, MinErr=%8.4f, AvgErr=%8.4f\n",
        Gen,SimTime(),ErrMin,ErrAvg,ptrain);
  if(OptFirst==2) printind(Pop[TopInd]);
}

void printstat()
{
  float t,g,i,l,b,p;
  t=SimTime(); if(t==0.0) t=1.0;
  g=Gen;
  i=g*PopGlobal;
```

```
  l=g*PopLocal;
  b=i*Nback;
  p= Nback ? GenCalcs+BackCalcs : GenCalcs;
  printf("\n"
    "Statistic               total    per sec.        time\n"
    "--------------------------------------------------------\n"
    "Generations:       %10.0f%12.6f%12.6f s\n"
    "Individuals global:%10.0f%12.6f%12.6f s\n"
    "Individuals local: %10.0f%12.6f%12.6f s\n",
    g,g/t,t/g,i,i/t,t/i,l,l/t,t/l);
  if(Nback) printf(
    "Backprop. steps:   %10.0f%12.6f%12.6f s\n",b,b/t,t/b);
  printf(
    "evaluated Patterns:%10.0f%12.6f%12.6f ms\n",p,p/t,1000*t/p);
}

int main(int argc,char **argv)
{
  int opt,i,j,k,n;
  char optstr[128]="\0";
  errtyp locErrMin,locErrAvg;
  int locTopInd;

  strcat(optstr,ParOptStr());
  strcat(optstr,IndOptStr());
  strcat(optstr,GenOptStr());
  strcat(optstr,BackOptStr());
  strcat(optstr,OutOptStr());

  initNetwork();

  while((opt=getopt(argc,argv,optstr)) != EOF)
  {
    if( handleParOpt(opt,optarg) &&
        handleIndOpt(opt,optarg) &&
        handleGenOpt(opt,optarg) &&
        handleBackOpt(opt,optarg) &&
        handleOutOpt(opt,optarg) )
      {
         if(ProcId==0) printf("%s\n%s\n%s\n%s\n%s",
          ParUsage(),IndUsage(),GenUsage(),
          BackUsage(),OutUsage());
        exit(1);
      };
  };

  if(initPar()) errorexit("parallel init failed\n");
  if(initInd()) errorexit("individual init failed\n");
  if(initGen(PopLocal,ProcId==0 ? PopGlobal : PopLocal))
    errorexit("genetic init failed\n");
  if(Nback && initBack()) errorexit("backpropagation init failed\n");
```

```
if(initOut()) errorexit("output init failed\n");

if(Nback) randomNetPop(PopLocal,InitWeight);
if(OptInfo && ProcId==0) printinfo();

setvect(0,Pop[0],PopLocal*CrWords*WORDLEN);
setvect(1,&Err[0],PopLocal*ERRLEN);
setvect(2,&ErrMin,FLOATLEN);
setvect(3,&ErrAvg,FLOATLEN);
setvect(4,&TopInd,WORDLEN);

for(Gen=1;Gen<=MaxGen;Gen++)
{
  calcerrors(PopLocal);
  if(Gen%Ntrans)
  {
    selection(PopLocal);
    continue;
  };
  if(ProcId==0)       /* Master */
  {
    for(i=1;i<Procs;i++)
    {
      k=PopLocal*i;
      setvect(0,Pop[k],PopLocal*CrWords*WORDLEN);
      setvect(1,&Err[k],PopLocal*ERRLEN);
      setvect(2,&locErrMin,FLOATLEN);
      setvect(3,&locErrAvg,FLOATLEN);
      setvect(4,&locTopInd,WORDLEN);
      n=recvvectfrom(5);
      if(ErrMin>locErrMin)
      {
        ErrMin=locErrMin;
        TopInd=locTopInd+k;
      };
      ErrAvg+=locErrAvg;
    };
    ErrAvg/=Procs;
    if(ErrMin<=MaxErr)
    {
      for(i=1;i<Procs;i++) send(i,&ErrMin,FLOATLEN);
      printlog();
      if(OptStat) printstat();
      if(OptFirst==1) {printf("\n"); printind(Pop[TopInd]);};
      printf("\nprogram succeeded.\n");
      exit(0);
    }
    if(OptLog && (Gen==1 || Gen%OptLog==0)) printlog();
    selection(PopGlobal);
    for(i=1;i<Procs;i++)
    {
```

```
        k=PopLocal*i;
        send(i,&ErrMin,FLOATLEN);
        send(i,Pop[k],PopLocal*CrWords*WORDLEN);
      };
    }
    else                /* Slaves */
    {
      sendvect(0,5);
      recv(&ErrMin,FLOATLEN);
      if(ErrMin<=MaxErr) exit(0);
      recv(Pop[0],PopLocal*CrWords*WORDLEN);
    };

    if((Ntrain>NoTrain) && (ErrMin*NoTrain<=MaxErr*Ntrain))
      NoTrain=min(Ntrain,NoTrain*2);
    OffsTrain++; if(OffsTrain>=Ntrain) OffsTrain=0;

  };
  if(ProcId==0)
  {
    if(OptStat==2) printstat();
    printf("\ntime out - program failed.\n");
  };
  exit(0);
}
```

B.11.3 File: mainback.c

```

/* Main file backpropagation */

#include "defs.h"
#include "sim.h"
#include "ind.h"
#include "back.h"

extern char *optarg;
extern int optind;

char *OutOptStr() {return "l:f:i:r:\0";}

char *OutUsage() {return
  "Output Parameters:\n"
  "-l <frequency of log-output>:     auto\n"
  "-f <show weights (0/1/2)>:        0\n"
  "-i <show parameter info (0/1)>:   1\n"
  "-r <show statistic (0/1/2)>:      1\0\n";}

#define DEFLOG          AUTO
#define DEFFIRST        0
#define DEFINFO         1
```

```
#define DEFSTAT         1

/* set default values */

int OptLog      =DEFLOG;
int OptFirst    =DEFFIRST;
int OptInfo     =DEFINFO;
int OptStat     =DEFSTAT;

int Iter;       /* current generation */
int StartTime;
int EndTime;

#define SimTime()       (gettime()-StartTime)

int handleOutOpt(char opt,char* arg)
{
  switch(opt)
  {
    case 'l': OptLog    =getint(arg,0,1000000000);      return 0;
    case 'f': OptFirst  =getint(arg,0,2);               return 0;
    case 'i': OptInfo   =getint(arg,0,1);               return 0;
    case 'r': OptStat   =getint(arg,0,2);               return 0;
    default: return 1;
  };
}

int initOut()
{
  StartTime=gettime();

  if(OptLog==AUTO)
  {
    if(MaxIter<=10)
      OptLog=1;
    else
      OptLog=rounddec(500000/((1+Nin+Nout)*
        (1+Nhid)*Ntrain));
  };
  return 0;
}

void errorexit(char *s)
{
  printf("%s",s);
  exit(1);
}

void printinfo()
{
  printf("Simulation Parameters:\n%s%s%s\n",
```

```
        SimParamStr,NetParamStr,BackParamStr);
}

void printlog()
{
  printf("Iter%7d: t=%7d, NetErr=%8.4f\n",Iter,SimTime(),NetErr);
  if(OptFirst==2) printnet();
}

void printstat()
{
  float t,i,e,u;

  t=SimTime(); if(t==0.0) t=1.0;
  i=Iter;
  e=BackCalcs;
  u=e*((Nin+1)*Nhid+(Nhid+1)*Nout);
  printf("\n"
    "Statistic               total    per sec.        time\n"
    "--------------------------------------------------------\n"
    "Iterations:        %10.0f%12.5f%12.6f s\n"
    "evaluated Patterns:%10.0f%12.5f%12.6f ms\n"
    "updated Weights:   %10.0f%12.0f%12.6f us\n",
    i,i/t,t/i,e,e/t,1000*t/e,u,u/t,1000000*t/u);
}

int main(int argc,char **argv)
{
  int opt,i,j,k;
  char optstr[128]="\0";

  strcat(optstr,SimOptStr());
  strcat(optstr,NetOptStr());
  strcat(optstr,BackOptStr());
  strcat(optstr,OutOptStr());

  while((opt=getopt(argc,argv,optstr)) != -1)
  {
    if( handleSimOpt(opt,optarg) &&
        handleNetOpt(opt,optarg) &&
        handleBackOpt(opt,optarg) &&
        handleOutOpt(opt,optarg) )
      {
        printf("%s\n%s\n%s",
          SimUsage(),NetUsage(),BackUsage(),OutUsage());
        exit(1);
      };
  };

  if(initSim()) errorexit("simulation init failed\n");
  if(initNet()) errorexit("individual init failed\n");
```

```
  if(initBack()) errorexit("backpropagation init failed\n");
  if(initOut()) errorexit("output init failed\n");
  if(OptInfo) printinfo();

  for(Iter=1;Iter<=MaxIter;Iter++)
  {

    calcNetErr();

    if(NetErr<=MaxErr)
    {
      printlog();
      if(OptStat) printstat();
      if(OptFirst==1) {printf("\n"); printnet();};
      printf("\nprogram succeeded.\n");
      exit(0);
    };
    if(Iter%OptLog==0 || Iter==1 || Iter==MaxIter) printlog();
    updateNet();
  };
  if(OptStat==2) printstat();
  printf("\ntime out - program failed.\n");
  exit(0);
}
```