Chapter 2

SIMULATED ANNEALING

E. Aarts, P. van der Horn, J. Korst, W. Michiels, and H. Sontrop
Philips Research Laboratories, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands

Abstract: Simulated Annealing is a meta-heuristic that performs a randomized local search to reach near-optimal solutions of combinatorial as well as continuous optimization problems. In this chapter we show how it can be used to train artificial neural networks by examples. Experimental results indicate that good results can be obtained with little or no tuning.

Key words: Simulated annealing; neural networks.

1. INTRODUCTION

In this chapter we give an introduction to simulated annealing and show how it can be used to train artificial neural networks. Training neural networks can be seen as adapting the neurons' weights to realize a desired input-output behavior.

We can distinguish two types of artificial neural networks, namely feed-forward and recurrent networks. A feed-forward network is characterized by a partial ordering $<$ on the nodes (neurons), such that the value of node i can only influence the value of node j, whenever $i < j$. Usually feed-forward networks are modeled to consist of multiple layers: one input layer, one or more hidden layers, and one output layer. Viewed from input to output, the value of a node in one layer only influences directly the value of the node in the next layer.

In a recurrent network the value of a given node can, via other nodes, again influence its own value. An interesting class of recurrent networks assumes bidirectional connections: if there is a connection between nodes i and j, then the value of i can influence the value of j and vice versa. Hopfield networks (Hopfield, 1982, 1984) and Boltzmann machines (Ackley, Hinton and Sejnowski, 1985) belong to this class of recurrent networks. In these networks there is no inherent difference between input and output nodes.

In this chapter, we restrict ourselves to feed-forward networks. The desired input-output behavior is characterized by a training set, i.e., a set of input-output samples. The goal of training neural networks is to adapt the nodes' weights such that after training, the neural network is able to find the corresponding output for each of the inputs in the training set. In addition, one tries to avoid overfitting, i.e., that also for inputs that are not in the training set the network is able to find the corresponding output.

Given the space limitations of this chapter, we do not discuss the use of simulated annealing in training recurrent networks. For details on how statistics on the equilibrium distribution associated with simulated annealing can be used to train Boltzmann machines, we refer to Aarts and Korst (1989).

The organization of this chapter is as follows. In Section 2 we present the basic simulated annealing algorithm. Section 3 explains how the behavior of simulated annealing can be modeled by Markov chains and gives details on its asymptotic convergence properties. Section 4 gives details on how simulated annealing can be used to train feed-forward neural networks. Implementation details and experimental results are given in Section 5.

2. BASIC SIMULATED ANNEALING ALGORITHM

In the early 1980s Kirkpatrick, Gelatt, and Vecchi (1983) and independently Cerny (1985) introduced the concepts of annealing in combinatorial optimization. In a combinatorial optimization problem we are given a finite or countably infinite set of solutions S and a cost function f that assigns a cost to each solution. The problem is to find a solution $i^* \in S$ for which $f(i^*)$ is either minimal or maximal, depending on whether the problem is a minimization or a maximization problem. Such a solution i^* is called a (globally) optimal solution. Without loss of generality, we restrict ourselves in this chapter to minimization problems.

Many interesting combinatorial optimization problems are NP-hard. For such problems it is generally believed that no algorithms exist that solve each instance in polynomial time. When confronted with an NP-hard combinatorial optimization problem, we have two options for tackling it. The first option is to aim for an optimal solution, despite the NP-hardness of the problem. A second option is to use a heuristic algorithm. Solutions found by such an algorithm are not necessarily optimal, but they are found within an acceptable amount of time. Hence, heuristic algorithms trade off optimality against computing time.

Heuristic algorithms can be classified into two categories: constructive algorithms and local search algorithms. A constructive algorithm generates a solution through a number of steps, where in each step the partial solution obtained so far is extended until in the last step a complete solution is obtained. Local search algorithms try to find high-quality solutions by searching through the solution space S. More specifically, a local search algorithm starts with an initial solution and then iteratively generates a new solution that is in some sense near to it. A neighborhood function $N:S \rightarrow 2^S$ defines for any given solution s the solutions $N(s)$ that are near to it. The set $N(s)$ is called the neighborhood of solution s. The process of searching through the solution space can be modeled as a walk through the (neighborhood) graph $G=(V,E)$, where node set V is given by the solution space S and arc set E contains arc (i,j) if and only if $j \in N(i)$.

The solution space of combinatorial optimization problems can typically be formulated in terms of discrete structures, such as sequences, permutations, graphs, and partitions. Local search uses these representations by defining neighborhood functions in terms of local rearrangements, such as moving, swapping, and replacing items, that can be applied to a representation to obtain a neighboring solution.

The simplest form of local search is iterative improvement. An iterative improvement algorithm continuously explores neighborhoods for a solution with lower cost. If such a solution is found, then the current solution is replaced by this better solution. The procedure is repeated until no better solutions can be found in the neighborhood of the current solution. By definition, iterative improvement terminates in a local optimum, which is a solution having a cost at least as good as all of its neighbors.

A disadvantage of using iterative improvement is that it easily gets trapped in poor local optima. To avoid this disadvantage, simulated annealing accepts in a limited way neighboring solutions with a cost that is worse than the cost of the current solution.

Originally the use of annealing in combinatorial optimization was heavily inspired by an analogy between the physical annealing process of solids and the problem of solving large combinatorial optimization problems. Since this analogy is quite appealing we use it here as a background for introducing simulated annealing.

In condensed matter physics, annealing is known as a thermal process for obtaining low energy states of a solid in a heat bath. The process consists of following two steps (Kirkpatrick, Gelatt, Vecchi, 1983).

- Increase the temperature of the heat bath to a maximum value at which the solid melts.
- Decrease carefully the temperature of the heat bath until the particles arrange themselves in the ground state of the solid.

In the liquid phase all particles arrange themselves randomly, whereas in the ground state of the solid, the particles are arranged in a highly structured lattice, for which the corresponding energy is minimal. The ground state of the solid is obtained only if the maximum value of the temperature is sufficiently high and the cooling is done sufficiently slow. Otherwise the solid will be frozen into a meta-stable state rather than into the true ground state.

As far back as 1953, Metropolis et al. introduced a simple algorithm for simulating the evolution of a solid in a heat bath to thermal equilibrium. Their algorithm is based on Monte Carlo techniques (Binder, 1978), and generates a sequence of states of the solid in the following way. Given a current state i of the solid with energy E_i, a subsequent state j is generated by applying a perturbation mechanism which transforms the current state into a next state by a small distortion, for instance, by displacement of a particle. The energy of the next state is E_j. If the energy difference, E_j - E_i, is less than or equal to 0, the state j is accepted as the current state. If the energy difference is greater than 0, the state j is accepted with a probability given by

$$\exp\left(\frac{E_i - E_j}{k_B T}\right),$$

where T denotes the temperature of the heat bath and k_B a physical constant known as the Boltzmann constant. The acceptance rule described above is known as the Metropolis criterion and the algorithm that goes with it is known as the Metropolis algorithm. It is known that, if the lowering of the temperature is done sufficiently slow, the solid can reach thermal equilibrium at each temperature. In the Metropolis algorithm this is achieved by generating a large number of transitions at a given temperature value. Thermal equilibrium is characterized by the Boltzmann distribution, which gives the probability of the solid to be in a state i with energy E_i at temperature T, and which is given by

$$\text{Prob}_T\{X = i\} = \frac{\exp\left(-E_i/(k_B T)\right)}{\sum_j \exp\left(-E_j/(k_B T)\right)} \tag{1}$$

where X is a random variable denoting the current state of the solid and the summation extends over all possible states. As we show below, the Boltzmann distribution plays an essential role in the analysis of the convergence of simulated annealing.

Returning to simulated annealing, the Metropolis algorithm can be used to generate a sequence of solutions of a combinatorial optimization problem by assuming the following equivalences between a physical many-particle system and a combinatorial optimization problem.

- Solutions in the combinatorial optimization problem are equivalent to states of the physical system.
- The cost of a solution is equivalent to the energy of a state.

Furthermore, we introduce a control parameter, which plays the role of the temperature. Simulated annealing can be viewed as a sequence of Metropolis algorithms, evaluated at decreasing values of the control parameter.

We now formulate simulated annealing in terms of a local search algorithm. For an instance (S,f) of a combinatorial optimization problem and a neighborhood function N, Figure 2-1 describes simulated annealing in pseudo-code.

```
procedure SIMULATED ANNEALING;
begin
   i := initial solution
   c := initial value
   repeat
      for l:=1 to L do
      begin
       probabilistically generate neighbor
       j of i
       if f(j) ≤ f(i) then accept j
       else accept j with probability
                    exp( (f(i) - f(j)) / c )
      end;
      update L
      update c
   until stopcriterion
end;
```

Figure 2-1. The simulated annealing algorithm in pseudo code.

The algorithm generates neighbors randomly. If a neighbor j has lower cost than the current solution i, then j is always accepted. If neighbor j has higher cost than i, then j is still accepted with a positive probability of

$$\exp\left(\frac{f(i)-f(j)}{c}\right).$$

The probability of accepting a deterioration in cost depends on the value of the control parameter c: the higher the value of the control parameter, the higher the probability of accepting the deterioration. The value of the control parameter is decreased during the execution of the algorithm. In Figure 1 the value L specifies the number of iterations that the control parameter is kept constant before it is decreased. The values of c and L and the stop criterion are specified by the 'cooling schedule'.

Initially, at large values of c, large deteriorations will be accepted; as c decreases, only smaller deteriorations will be accepted and finally, as the value of c approaches 0, no deteriorations will be accepted at all. Furthermore, there is no limitation on the size of deterioration with respect to its acceptance. In simulated annealing, arbitrarily large deteriorations are accepted with positive probability; for these deteriorations the acceptance probability is small, however. This feature means that simulated annealing, in contrast to iterative improvement, can escape from local minima while it still exhibits the favorable features of iterative improvement, namely simplicity and general applicability. The speed of convergence of simulated annealing is determined by the cooling schedule. In the next section we will indicate that under certain mild conditions on the choice of the cooling schedule simulated annealing converges asymptotically to global optima.

Comparing simulated annealing to iterative improvement it is evident that simulated annealing can be viewed as a generalization. Simulated annealing becomes identical to iterative improvement in the case where the value of the control parameter is taken equal to zero. With respect to a comparison between the performances of both algorithms we mention that for most problems simulated annealing performs better than iterative improvement, repeated for a number of different initial solutions (Fox 1994).

3. MATHEMATICAL MODELING

Simulated annealing can be mathematically modeled by means of Markov chains (Feller, 1950; Isaacson and Madsen, 1976; Seneta, 1981). In this model, we view simulated annealing as a process in which a sequence of Markov chains is generated, one for each value of the control parameter. Each chain consists of a sequence of trials, where the outcomes of the trials correspond to solutions of the problem instance.

Let (S,f) be a problem instance, N a neighborhood function, and $X(k)$ a random variable denoting the outcome of the kth trial. Then the transition probability at the kth trial for each pair $i,j \in S$ of outcomes is defined as

$$P_{ij}(k) = \text{Prob}\{X(k) = j \mid X(k-1) = i\}$$
$$= \begin{cases} \mathrm{G}_{ij}(c_k) A_{ij}(c_k) & \text{if } i \neq j \\ 1 - \sum_{l \in S, l \neq i} G_{il}(c_k) A_{il}(c_k) & \text{if } i = j \end{cases} \tag{2}$$

where G_{ij} (c_k) denotes the generation probability, i.e., the probability of generating a solution j when being at solution i, and $A_{ij}(c_k)$ denotes the acceptance probability, i.e., the probability of accepting solution j once it is generated from solution i. The most frequently used choice for these probabilities is the following (Aarts and Korst, 1989):

$$G_{ij}(c_k) = \begin{cases} |N(i)|^{-1} & j \in N(i) \\ 0 & j \notin N(i) \end{cases} \tag{3}$$

and

$$A_{ij}(c_k) = \begin{cases} 1 & \text{if } f(j) \leq f(i) \\ \exp\left((f(i) - f(j))/c_k\right) & \text{if } f(j) > f(i) \end{cases} \tag{4}$$

For fixed values of c_k, the probabilities do not depend on k, in which case the resulting Markov chain is time-independent or homogeneous. Using the theory of Markov chains it is fairly straightforward to show that, under the condition that the neighborhoods are strongly connected - in which case the Markov chain is irreducible and aperiodic - there exist a unique stationary distribution of the outcomes. This distribution is the probability distribution of the solutions after an infinite number of trials and assumes the following form (Aarts and Korst, 1989).

Theorem 1. *Given an instance (S,f) of a combinatorial optimization problem and a suitable neighborhood function. Then, after a sufficiently large number of transitions at a fixed value c of the control parameter, applying the transition probabilities of (2), (3), and (4), simulated annealing will find a solution $i \in S$ with a probability given by*

$$\text{Prob}_c\{X = i\} \stackrel{\text{def}}{=} q_i(c) = \frac{1}{N_0(c)} \exp\left(-\frac{f(i)}{c}\right), \tag{5}$$

where X is a stochastic variable denoting the current solution obtained by simulated annealing and

$$N_0(c) = \sum_{j \in S} \exp\left(-\frac{f(j)}{c}\right)$$

denotes a normalization constant.

A proof of this theorem is considered beyond the scope of this chapter. For those interested we refer to Aarts and Korst (1989). The probability distribution of (5) is called the stationary or equilibrium distribution and it is the equivalent of the Boltzmann distribution of (1). We can now formulate the following important result.

Corollary 1. *Given an instance (S,f) of a combinatorial optimization problem and a suitable neighborhood function. Furthermore, let the stationary distribution $q_i(c)$ that simulated annealing finds solution i after an infinite number of trials at value c of the control parameter be given by (5). Then*

$$\lim_{c \downarrow 0} q_i(c) \stackrel{\text{def}}{=} q_i^* = \frac{1}{|S^*|} C_{(S^*)}(i),$$

where S^* denotes the set of globally optimal solutions and where for any two sets A and $A' \subseteq A$ the characteristic function C is defined such that $C_{(A')}(a) = 1$ if $a \in A'$ and $C_{(A')}(a) = 0$ if $a \in A \setminus A'$.

The result of this corollary is quite interesting since it guarantees asymptotic convergence of the simulated annealing algorithm to the set of globally optimal solutions under the condition that the stationary distribution of (5) is attained at each value of *c*. More specifically it implies that asymptotically optimal solutions are obtained which can be expressed as

$$\lim_{c \downarrow 0} \lim_{k \to \infty} \text{Prob}_c\{X(k) \in S^*\} = 1.$$

We end this section with some remarks.

- It is possible to formulate a more general class of acceptance and generation probabilities than the ones we considered above, and prove asymptotic convergence to optimality in that case. The probabilities we used above are imposed by this more general class in a natural way and used in practically all applications reported in the literature.

- The simulated annealing algorithm can also be formulated as an inhomogeneous algorithm, viz. as a single inhomogeneous Markov chain, where the value of the control parameter c_k is decreased in between subsequent trials. In this case, asymptotic convergence again can be proved. However an additional condition on the sequence c_k of values of the control parameter is needed, namely

$$c_k \geq \frac{\Gamma}{\log(k+2)}, k = 0,1,...,$$

  for some constant Γ that can be related to the neighborhood function that is applied.
- Asymptoticity estimates of the rate of convergence show that the stationary distribution of simulated annealing can only be approximated arbitrarily closely if the number of transitions is proportional to $|S|^2$. For hard problems $|S|$ is necessarily exponential in the size of the problem instance, thus, implying that approximating the asymptotic behavior arbitrarily close results in an exponential-time execution of simulated annealing. Similar results have been derived for the asymptotic convergence of the inhomogeneous algorithm.

Summarizing, simulated annealing can find optimal solutions with probability 1 if it is allowed an infinite number of transitions and it can get arbitrarily close to an optimal solution if at least an exponential amount of transitions is allowed. These results are merely of theoretical interest. The real strength of simulated annealing lies in the good practical results that can be obtained by applying more efficient finite-time implementations.

4. TRAINING NEURAL NETWORKS

In the remainder of this chapter we focus on applying simulated annealing to the problem of training a neural network. We consider a multilayer feed-forward neural network with a set V_{in} of n input nodes, a set V_{h} of m hidden nodes, and a single output node y. The neural network computes a function $h: \Re^n \rightarrow \Re$.

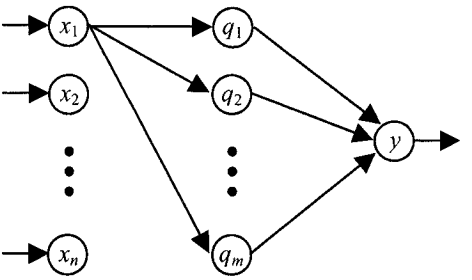

Figure 2-2. Neural network considered in this chapter. To simplify the picture, we omit the arcs from input nodes x_2 and x_n to the hidden nodes.

The neural network we consider has an arc from each input node to each hidden node and from each hidden node to the output node; see Figure 2-2. The weight of an arc from node i to node j is given by w_{ij}. Except for the input nodes, each node i has a bias value w_i. The output out_j of hidden node q_j is computed by

$$out_j = \text{sigmoid}\left(w_j + \sum_{x_i \in V_{\text{in}}} v(x_i) w_{ij} \right),$$

where sigmoid(x) is the non-linear sigmoid function $1/(1+e^{-x})$ and $v(x_i)$ is the value of input node x_i. Unlike the hidden nodes, output node y has a linear activation function. More specifically, this means that the output of node y, i.e., the output of the neural network, is computed from the outputs of the hidden layer via

$$out_y = w_y + \sum_{q_j \in V_{\text{h}}} out_j w_{jy} .$$

Suppose that we want the neural network to compute some function $g : \Re^n \rightarrow \Re$. The problem that we now face is to choose the weights in the neural network, i.e., the weights of the arcs and the biases, such that the function h computed by the neural network approximates g as good as possible. We consider the following version of this training problem.

Let $\mathcal{T}=\{T_1, T_2,..., T_r\}$ be a training set consisting of r possible inputs for the function g. Furthermore, let h_w be the function computed by the neural network of Figure 2 for weight vector w, where a weight vector specifies all

the weights in the neural network. The problem is now to find a weight vector w, such that the Root Mean Square Error (RMSE) defined by

$$\sqrt{\frac{\sum_{T_i \in \mathcal{T}} error(T_i, w)}{r}}$$

is minimal, where

$$error(T_i, w) = \left(g(T_i) - h_w(T_i)\right)^2.$$

The problem of choosing the weight vector based on a training set is that the corresponding neural network may only show a good performance on the inputs in this set, but not on the entire domain of the function g that has to be learned. Therefore, after training a neural network, we do not evaluate the network by its performance on the inputs in the training set, but by its performance on the inputs in some other set, called the validation set.

5. SIMULATED ANNEALING IMPLEMENTATIONS

A weight vector for the neural network depicted in Figure 2 contains $(n+2)m+1$ weights in total: nm weights for the arcs between the input nodes and the hidden nodes, m weights for the arcs between the hidden nodes and the output node, and $m+1$ weights for the bias values. Instead of using simulated annealing to find all these weights, we only use it to find the nm weights of the arcs from the input nodes to the hidden nodes and the m weights representing the biases of the hidden nodes. For a given value of these weights, an optimal value of the other $m+1$ weights, i.e., the weights of the arcs from the hidden nodes to the output node and the bias of the output node, can be derived by means of linear regression. Given vectors X and Y, this method looks for a vector b that minimizes the squared error $e^T e$, where $e=Y-Xb$ is the error made when approximating a vector Y by the linear relation Xb. In our case, $Y=(g(T_1), g(T_2), ..., g(T_r))^T$ is the vector containing the correct function value for each input T_i from the training set $\mathcal{T}$ and X is given by

$$X = \begin{pmatrix} 1 & out_1(1) & \dots & out_m(1) \\ \vdots & \vdots & \ddots & \vdots \\ 1 & out_1(r) & \dots & out_m(r) \end{pmatrix},$$

where $out_i(j)$ defines the output of the ith hidden node for input T_j. Linear regression finds that the vector $b=(w_t,w_{1t},w_{2t},...,w_{mt})$ is given by

$$b = (X^T X)^{-1} X^T Y .$$

In Section 5.1 we present a basic simulated annealing implementation that is based on a neighborhood function presented by Sexton et al. (1998). This basic implementation is extended in Section 5.2.

5.1 Basic Implementation

As mentioned above we do not need to apply simulated annealing on the solution space containing all possible weight vectors. The solution space S can be restricted to weight vectors that only contain the weights of the arcs between the input nodes and the hidden nodes and the bias values of the hidden nodes. We will call such a weight vector a restricted weight vector.

Given the solution space, we can now define a neighborhood function. Let $w,v \in S$ be two restricted weight vectors. Then v is a neighbor of w with respect to some given parameter α if each weight v_i in v can be obtained from the corresponding weight w_i in w by adding or subtracting αw_i. Hence, the neighborhood N_α of w is given by

$$N_\alpha(w) = \{((1+z_1)w_1,...,(1+z_{(n+1)m})w_{(n+1)m}) \mid z_1,...,z_{(n+1)m} \in \{-\alpha,\alpha\}\}.$$

5.2 Extended Implementation

We extend our basic implementation by introducing an additional level of local search. We do this by an approach called iterated local search (Lourenço et al., 2002). The approach works as follows. Let N_1 and N_2 be two neighborhood functions. Neighborhood function N_1 is based on the same solution space used by our basic implementation, i.e., on the set of restricted weight vectors, and neighborhood function N_2 is based on the set of solutions that are locally optimal with respect to N_1. We now search for a weight vector that minimizes RMSE by performing simulated annealing on neighborhood function N_2, where we use iterative improvement based on N_1 to derive an N_2-neighbor.

To be more specific, we define the neighborhood function N_1 as N_α for some parameter α and N_2 as $N_{\alpha\delta}$ for some additional parameter δ, where $N_{\alpha\delta}$ is defined as follows. Solution s' is an $N_{\alpha\delta}$-neighbor of s if and only if s' can be obtained from s by performing an N_δ-move followed by an execution of iterative improvement based on N_α. The parameter δ is chosen larger than

α. The idea of the resulting iterated local search algorithm is that it tries to find high-quality N_α-local optima by initializing iterative improvement with a (hopefully) high-quality starting solution that is obtained by 'kicking' a N_α-local optimum out of local optimality.

The problem of the proposed algorithm is the running time of the iterative improvement algorithm. Because $|N_\alpha|=2^{(n+1)m}$, it takes exponential time to evaluate all neighbors of a solution. Therefore, we replace the iterative improvement algorithm by a heuristic variant of the algorithm. The heuristic algorithm randomly generates neighbors of the current solution. If a solution is found with better cost, then it replaces the current solution. If some predefined number M of neighbors has been generated without finding a better solution, then the algorithm assumes that the solution is locally optimal (although this need not be the case). The resulting algorithm is depicted in Figure 2-3.

```
procedure ITERATED LOCAL SEARCH;
begin
   i := initial solution
   c := initial value
   repeat
      for l:=1 to L do
      begin
        j := 'kick' i
        k := heuristic iterative improvement on  j
        if f(k) ≤  f(i) then accept k
        else accept k with probability
                    exp( (f(i) - f(k)) / c )
      end;
      update L
      update c
   until stopcriterion
end;
```

Figure 2-3. Iterated local search algorithm.

6. EXPERIMENTAL RESULTS

We have carried out some experiments to evaluate the two simulated annealing implementations discussed in Section 5. Our experiments are similar to the ones performed by Glover and Martí (2006) for tabu search (Chapter 4 in this book). This means that we trained a neural network with 9 hidden nodes, two input nodes x_1 and x_2, and one output node y. The domain

of x_1 is given by [-100,100] and the domain of x_2 is given by [-10,10]. The training set consists of 200 inputs. These inputs are generated uniformly at random. The validation set consists of another 100 randomly generated inputs. For a precise definition of the 15 test functions used, we refer to Glover and Martí (2006).

In our basic simulated annealing implementation we use α=0.01 and L_k=1 for all iterations k. To initialize the starting value and the end value of c_k we perform a random walk through the neighborhood graph. We define the starting value of the control parameter such that if this value would be used, then the acceptance criterion of simulated annealing would accept 90% of the moves performed in the random walk. Analogously, we define the end value of the control parameter such that if this value would be used, then the acceptance criterion would accept only 1% of the moves from the random walk that lead to a cost deterioration. After each iteration of simulated annealing, the control parameter is updated by $c_{k+1} := q \cdot c_k$, where parameter q is chosen such that the simulated annealing algorithm terminates within the maximum available amount of time $\tau_{\max}$, where the algorithm terminates if the control parameter reaches its end value. Hence, q is chosen, such that if the average time per iteration is τ_{avg}, then the end value of c_k is reached after at most $\tau_{\max}/\tau_{\text{avg}}$ iterations. In our simulations, we let $\tau_{\max}$ be 1 minute on a Pentium 4, 2.8 GHz, 1GB RAM.

In our extended simulated annealing implementation depicted in Figure 3 we use M=200. Furthermore, we let α and δ be related by $\delta=3\alpha$. While in our basic implementation α was kept constant at 0.01, we cool (decrease) α, and consequently also δ, in a similar way as we cool the control parameter. This resulted in better results than when α and δ were kept constant. The starting value of α is 0.1, and we derive the starting value of c_k and the end value of c_k in the same way as for the basic implementation. The time in which the algorithm has to terminate is again set at 1 minute.

The results of our simulations are depicted in Tables 2-1 and 2-2. We omit the functions Goldstein, Beal, and SixHumpCamelB. Similarly as the tabu search implementations of Glover and Martí (2006), our implementations were not able to train the neural network for these functions. As Glover and Martí (2006), we run both algorithms 20 times on each instance of the training problem. We report both the standard deviation and the average deviation.

Due to overfitting, the trained network sometimes shows a good performance in training, but a very poor performance on the validation set. To prevent that these results pollute the overall numbers, we only take runs of the algorithms into account in which the performance of the neural network on the training set does not differ too much from the performance on the validation set. Glover and Martí (2006) use the same strategy.

Table 2-1. Basic simulated annealing implementation

Function	Testing Average	Standard	Validation Average	Standard
Sexton 1	0	0	0	0
Sexton 2	0.35	0.28	0.43	0.38
Sexton 3	0.39	0.19	0.65	0.2
Sexton 4	0.52	0.35	0.58	0.38
Sexton 5	200.16	211.5	218.13	231.46
Branin	1917.46	1785.74	2372.63	2201.79
B2	0.69	0.4	0.76	0.44
Easom	0	0	0.01	0.02
Shubert	23.32	3.77	29.01	6.56
Booth	1.42	0.78	1.63	0.91
Matyas	0.24	0.43	0.26	0.44
Schwefel	0.81	0.49	1.01	0.6

Table 2-2. Extended simulated annealing implementation

Function	Testing Average	Standard	Validation Average	Standard
Sexton 1	0	0	0	0
Sexton 2	0.02	0.02	0.03	0.02
Sexton 3	1.16	0.64	2.4	0.76
Sexton 4	0.24	0.17	0.33	0.24
Sexton 5	7.41	7.93	15.56	20.04
Branin	89.45	34.11	124.02	41.38
B2	0.33	0.02	0.38	0.02
Easom	0	0	0	0
Shubert	17.46	2.01	37.68	9.38
Booth	0.3	0.1	0.38	0.16
Matyas	0.02	0.01	0.02	0.01
Schwefel	0.25	0.18	0.33	0.29

Glover and Martí (2006) ran their tabu search implementations for 10 minutes on a computer with the same characteristics as the computer we used. Although we run our simulated annealing implementations for only one minute, both algorithms outperform their Back Propagation and Extended Tabu Search implementations. The basic simulated annealing implementation shows a better performance for 8 out of the 12 test functions and the extended simulated annealing implementation shows a better performance for 10 out of the 12 test functions. The Extended Tabu Search Path Relinking algorithm discussed by Glover and Martí (2006) outperforms our algorithms for all functions, although its implementation is more involved.

7. CONCLUSIONS

Simulated annealing is a randomized local search algorithm that is generally applicable and easy to implement. In this chapter we introduced the metaheuristic and used it to train artificial neural networks. We proposed two simulated annealing implementations and evaluated them by carrying out some experiments. The experiments indicate that simulated annealing yields high-quality solutions with little or no tuning.

REFERENCES

Aarts, E. H. L., and Korst, J. H. M., 1989, *Simulated Annealing and Boltzmann Machines*, Wiley.

Ackley, D. H., Hinton, G. E., and Sejnowski, T. J., 1985, A learning algorithm for Boltzmann machines, *Cognitive Science* **9**: 147-169.

Binder, K., 1978, *Monte Carlo Methods in Statistical Physics*, Springer-Verlag.

Cerny, V., 1985, Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm, *Journal of Optimization Theory and Applications* **45**: 41-51.

Glover, F., and Martí, R., 2006, Tabu search, Chapter 4 in this book.

Hopfield, J. J., 1982, Neural networks and physical systems with emergent collective computational abilities, *Procs. of the Natl. Academy of Sciences of the USA* **79**: 2554-2558.

Hopfield, J. J., 1984, Neurons with graded response have collective computational properties like those of two-state neurons, *Proceedings of the National Academy of Sciences of the USA* **81**: 3088-3092.

Feller, W., 1950, *An Introduction to Probability Theory and Its Applications* **1**, Wiley.

Fox, B. L., 1994, Random restart versus simulated annealing, *Computers & Mathematics with Applications*, **27**:33-35.

Isaacson, D., and Madsen, R., 1976, *Markov Chains*, Wiley.

Kirkpatrick, S., Gelatt Jr., C. D., and Vecchi, M. P., 1983, Optimization by simulated annealing, *Science* **220**: 671-680.

Lourenço, H. R., Martin, O., and Stützle, T., 2002, Iterated local search, in: *Handbook of Metaheuristics*, F. Glover and G. Kochenberger, eds., Kluwer A. P., 321-353.

Seneta, E., 1981, *Non-negative matrices and Markov chains*, Springer-Verlag.

Sexton, R.S., Alidaee, B., Dorsey, E., and Johnson, J. D., 1998, Global optimization for artificial neural networks: a tabu search application, *European Journal of Operational Research* **106**: 570-584.