IIE Transactions (2002) **34**, 273–282

Neural network prediction in a system for optimizing simulations

MANUEL LAGUNA[1] and RAFAEL MARTÍ[2]

[1]*Graduate School of Business Administration, University of Colorado, Boulder, CO 80309-0419, USA*
E-mail: laguna@colorado.edu
[2]*Departament D'Estadística i Investigació Operativa, Universitat de València, Burjassot 46100, Spain*
E-mail: rafael.marti@uv.es

Received June 2000 and accepted February 2001

Neural networks have been widely used for both prediction and classification. Back-propagation is commonly used for training neural networks, although the limitations associated with this technique are well documented. Global search techniques such as simulated annealing, genetic algorithms and tabu search have also been used for this purpose. The developers of these training methods, however, have focused on accuracy rather than training speed in order to assess the merit of new proposals. While speed is not important in settings where training can be done off-line, the situation changes when the neural network must be trained and used on-line. This is the situation when a neural network is used in the context of optimizing a simulation. In this paper, we describe a training procedure capable of achieving a sufficient accuracy level within a limited training time. The procedure is first compared with results from the literature. We then use data from the simulation of a jobshop to compare the performance of the proposed method with several training variants from a commercial package.

1. Introduction

Neural network training is typically considered an off-line activity. For example, banks use neural network technology to predict the probability that a credit card applicant will become a delinquent cardholder. Since the decision of granting a credit card is not made instantly and is not based on data that is generated in real time, the neural networks embedded in credit card application software can be trained off-line with large datasets containing historical records of past applicants. It is not unusual for these training runs to be done overnight with the purpose of achieving a desired level of prediction accuracy. In such an environment, the training procedures may be run from different starting points while applying a myriad of search strategies in order to achieve the best possible results.

There are, however, situations where an extensive training that requires minutes or perhaps hours of CPU time is not possible. This situation is one in which the necessary training data are being generated in real time. That is, the data do not reside in large databases from which training sets can be constructed. Instead, the same process that will make use of the neural network output generates the data. We refer to this situation as on-line training of the neural network. One important application of on-line training arises in the context of optimizing a simulation.

The rationale behind the optimization of simulations is as follows (Glover and Kelly, 1998). Simulation, as a computer-based tool, is widely used by many decision-makers in business and industry to improve operating and organizational efficiency. The basic idea of simulation is to model a physical process on the computer, incorporating the uncertainties that are inherent in all real systems. The model is then executed to simulate the effects of the physical process and to determine their consequences. For example, a factory can be modeled and simulated by incorporating the relationships among production times, demands, tolerances, and breakdowns, including provision for the uncertain nature of each. The process variation produced by the actual design of the product is also incorporated into the model.

Such a simulation may model the flow of individual products through the factory over a period of hours, days, or even months. The analyst typically makes virtual changes to the factory and/or products, to predict the impacts without ever changing a piece of real equipment or manufacturing a new product. Similarly, simulation models are often applied to analyze the consequences of alternative scenarios in financial planning and marketing

strategy. These uses of simulation have produced widespread benefits in industry, reducing costs and increasing profits through improved decisions.

In spite of its acknowledged benefits, simulation has suffered a limitation that has prevented it from uncovering the best decisions in critical practical settings. This limitation arises out of an inability to evaluate more than a fraction of the immense range of options available. Practical problems in areas such as manufacturing, marketing, logistics and finance typically pose vast numbers of interconnected alternatives to consider. As a consequence, the decision-making goal of identifying and evaluating the best (or near best) options has been impossible to achieve in many applications.

Theoretically, the issue of identifying best options falls within the realm of optimization. Until quite recently, however, the methods available for finding optimal decisions have been unable to cope with the complexities and uncertainties posed by many real world problems, particularly those approached by simulation. In fact, these complexities and uncertainties are the primary reason that simulation is chosen as a basis for handling such problems. Consequently, decision makers have been faced with the "Catch 22" that many important types of real world optimization problems can only be treated by the use of simulation models, but once these problems are submitted to simulation there are no optimization methods that can adequately cope with the challenge of searching for the best solutions. In short, there does not exist any type of search process that is capable of effectively integrating simulation and optimization. The same shortcoming is also encountered outside the domain of simulation, as situations increasingly arise where complex (realistic) models cannot be analyzed using traditional "closed-form" optimization tools.

Recent developments have changed this picture. Advances in the field of metaheuristics — the domain of optimization that incorporates artificial intelligence and analogs to physical, biological or evolutionary processes — have led to the creation of new approaches that successfully integrate simulation and optimization. Two of these approaches, Evolver and OptQuest, are commercial products that have been integrated with simulation software. *Evolver* is an implementation of genetic algorithms while *OptQuest* is an implementation of scatter search. (See Laguna and Martí (2000) for a description of the optimization engine of OptQuest.)

Since simulations are generally computationally expensive, the optimization process would be able to search the solution space more extensively if it were able to quickly eliminate from consideration low-quality solutions, where quality is based on the performance measure being optimized. This is where a neural network becomes useful. Specifically, a neural network can be used to filter out solutions that are likely to perform poorly when the simulation is executed. In this way, the neural network becomes the prediction model for the simulation in the same way that the simulation is the prediction model for the real system.

In the remainder of the paper, we first formalize the use of a neural network as a prediction model for the simulation and then describe an on-line training procedure capable of producing fairly accurate predictions within a reasonable time. The paper ends with computational experiments and associated conclusions. We do not provide a literature review of neural networks and their applications in operations research, and instead refer the reader to Sharda and Rampal (1995) for a comprehensive annotated bibliography. This introduction to neural networks and the associated bibliography is an extended and updated version of Sharda (1994).

2. Neural networks and the optimization–simulation problem

The optimization of simulations must be carried out within a somewhat small search horizon. When search procedures such as genetic algorithms, simulated annealing and tabu search are applied to problems for which the evaluation of the objective function is instantaneous, they tend to employ thousands and even millions of objective function evaluations during a single optimization run. However, this is not practical when the objective function evaluation consists of executing a computer simulation. Consider, for example, a situation where a company desires to optimize the performance of a system, whose simulation model requires 2 minutes to run. Even if the search is carried out over a weekend, the simulation could not be run for more than 1440 times (i.e., 48 hours $\times$ 30 runs per hour) on a machine with one processor. Therefore, the optimization procedure that calls the simulator must be very selective in terms of the number of times that a solution is evaluated.

If instead of directly evaluating a solution, a neural network is used to estimate the outcome from the simulation, some solutions could be disregarded thereby saving simulation time. In other words, the neural network would act as a filter to eliminate solutions that can be predicted with certain accuracy to be inferior to the best-known solution. Let us introduce the following notation that will help us formalize this process:

x = a solution to the optimization–simulation problem;
$f(x)$ = the output of the simulation when solution x is used as the set of inputs;
$p(x,w)$ = the output of the simulation as predicted by the neural network with weights w when the solution x is used as the inputs;
x^* = the best-known solution to the optimization–simulation problem (i.e., $f(x^*) < f(x)\ \forall\ x$);

l = the set of lower bounds for the decision variables x;
u = the set of upper bounds for the decision variables x.

We define the optimization–simulation problem as:

$$\text{Min } f(x),$$

subject to

$$l \leq x \leq u,$$

and some variables x may be restricted to be discrete. Note that some optimization systems, such as OptQuest, are capable of handling linear constraints as well as bounds on simulation outputs. For the purpose of this paper, however, we assume without loss of generality that the optimization–simulation problem consists of minimizing a simulation output by selecting the values of input variables within a given range.

Parallel to the optimization–simulation problem, there is a training problem, which consists of finding the set of weights w that minimize an aggregate error measure. The most common error measure, which we also use in this study, is MSE (i.e., the Mean Squared Errors). Suppose that during the search for the optimal values of x, the procedure applied to the optimization–simulation problem generates a set ALL of solutions x. Note that since ALL is the set of solutions generated during the search, $x^* \in$ ALL. Let TRAIN be a random sample of solutions in ALL, such that $|\text{TRAIN}| \leq |\text{ALL}|$. Then, we define the training problem as:

$$\text{Min } g(w) = \frac{1}{|\text{TRAIN}|} \sum_{x \in \text{TRAIN}} (f(x) - p(x,w))^2$$

where w is the set of optimization variables of the training problem. Since the training problem cannot be solved until there are at least |TRAIN| solutions in ALL, the search procedure for the optimization–simulation problem must initially operate without the help of the neural network by evaluating trial solutions x with the simulation. As the optimization–simulation search advances, the set of ALL solutions becomes large enough so that a suitable training set can be constructed. The training problem is then periodically solved with new TRAIN sets in order to improve the accuracy of the predictions generated by the associated neural network.

Suppose that the neural network has been trained and that w^* are the corresponding best set of weights. Also, suppose that the search procedure for the optimization–simulation problem generates the trial solution x. Before evaluating $f(x)$, the following test is performed:

$$\text{if } p(x,w) > f(x^*) + \alpha \text{ then discard } x.$$

The value of α accounts for the variability of the objective function values and it might also include error information from the training process. The test indicates that if the trial solution is not likely to improve the best-known solution, then the trial solution should be discarded. A convenient definition of α is:

$$\alpha = 2\hat{\sigma}_{f(x)},$$

where $\hat{\sigma}_{f(x)}$ is the sample standard deviation of $f(x)$ for $x \in$ TRAIN. Note that this definition is moderate with respect to the rejection criterion. While a conservative definition would add a factor to consider the training error, a more aggressive definition would reduce the multiplier from two to one standard deviation(s). Note that an aggressive definition may reject a new best solution to the optimization–simulation problem before its evaluation. More elaborate definitions of α could be based on the standard deviation of the errors.

3. On-line training procedure

Our on-line training procedure is based on the scatter search methodology. Scatter Search (SS) is a novel instance of evolutionary methods, because it violates the premise that evolutionary approaches must be based solely on randomization — though they likewise are compatible with randomized implementations like the one we describe here. SS is also novel, in comparison to the well-known Genetic Algorithm (GA) class of evolutionary methods, by being founded on strategies that only piecemeal came to be proposed as augmentations to GAs more than a decade after their debut in scatter search. Scatter search embodies principles and strategies that are still not emulated by other evolutionary methods, and that prove advantageous for solving a variety of complex optimization problems, including neural network training. More about the origin and multiple applications of scatter search can be found in Glover (1998), Glover *et al.* (1999) and Laguna (2000).

We now outline our adaptation of scatter search to the training problem

Step 0. Normalize input and output data.
Step 1. Start with $P = \emptyset$. Use the diversification method to construct a solution w between *wlow* and *whigh*. If $w \notin P$ then add w to P (i.e., $P = P \cup w$), otherwise, discard w. Repeat this step until $|P| =$ *PSize*. Apply the improvement method to the best $b/2$ solutions in P to generate $w^{(1)}, \ldots, w^{(b/2)}$. Generate $b/2$ more solutions, where $w^{(b/2+i)} = w^{(i)}(1 + \text{U}[-0.3, 0.3])$ for $i = 1, \ldots, b/2$. Build $RefSet = \{w^{(1)}, \ldots, w^{(b)}\}$.
Step 2. Order *RefSet* according to their objective function value such that $w^{(1)}$ is the best solution and $w^{(b)}$ the worst.
while (*NumEval* < *TotalEval*) **do**
Step 3. Generate *NewPairs*, which consists of all pairs of solutions in *RefSet* that include at least one new solution. Make *NewSolutions* = $\emptyset$.

for (all *NewPairs*) **do**

Step 4. Select the next pair $(w^{(i)}, w^{(j)})$ in *NewPairs*.

Step 5. Obtain new solutions w as linear combinations of $(w^{(i)}, w^{(j)})$ and add them to *NewSolutions*.

end for

Step 6. Select the best b solutions in *NewSolutions* and apply the improvement method.

for (each improved w) **do**

if (w is not in *RefSet* and $g(w) < g(w^{(b)})$ **then** make $w^{(b)} = w$ and reorder *RefSet*.

end for

while (*IntCount* < *IntLimit*)

Step 7. Make $IntCount = IntCount + 1$ and $w = w^{(1)}$.

Step 8. Make $w = w(1 + \mathrm{U}[-0.05, 0.05])$ and apply improvement method.

if $(g(w) < g(w^{(1)}))$ **then** Make $w^{(1)} = w$ and $IntCount = 0$.

end while

Step 9. Apply the improvement method to $w^{(i)}$ for $i = 1, \ldots, b/2$ in *RefSet*.

Step 10. Make $w^{(b/2+i)} = w^{(i)}(1 + \mathrm{U}[-0.01, 0.01])$ for $i = 1, \ldots, b/2$ in *RefSet*.

end while

We refer to w as a solution to the training problem. The procedure starts with the input and output data normalization. Let $y = f(x)$ and also let $x_{\max}$, $y_{\max}$, $x_{\min}$ and $y_{\min}$ be the maximum and the minimum values for x and y in the TRAIN set, respectively. The normalization of the data is done as follows, where x' and y' are the normalized values:

$$x' = \left(\frac{high - low}{x_{\max} - x_{\min}}\right)x + \frac{low \times x_{\max} - high \times x_{\min}}{x_{\max} - x_{\min}}$$

$high = 1.0$ and $low = -1.0$,

$$y' = \left(\frac{high - low}{y_{\max} - y_{\min}}\right)y + \frac{low \times y_{\max} - high \times y_{\min}}{y_{\max} - y_{\min}}$$

$high = 0.8$ *and* $low = -0.8$.

In this normalization, the multiplier is the scaling factor and the added constant is the offset value. These normalization functions and the specific values for *low* and *high* are defined as recommended and used in NeuralWare (*www.neuralware.com*). The training procedure operates on the normalized data set.

After the data normalization, an initial reference set (*RefSet*) of b solutions is created. A set P of *PSize* solutions w (bounded between *wlow* and *whigh*) is built with the following diversification method, based on a controlled randomization scheme (Glover *et al.*, 1999). The range (*wlow, whigh*) is subdivided into four sub-ranges of equal size. Then, a solution w is constructed in two steps. First a sub-range is randomly selected. The probability of selecting a sub-range is inversely proportional to its frequency count. Then a value is randomly generated within the selected sub-range. The number of times sub-range j has been chosen to generate a value for w_i is accumulated in the frequency counter $freq(i, j)$. The main goal of the diversification generator is to create solutions that are diverse with respect to those solutions that have already been generated in the past. That is, the frequency counts work as "seed solutions" from which the diversification attempts to move away.

The reference set *RefSet* is filled with the best $b/2$ solutions in P to which an improvement method is applied (see below). The *RefSet* is completed with $b/2$ more solutions generated as perturbations of the first $b/2$. The perturbation consists of multiplying each weight by $1 + \mathrm{U}[a, b]$, where $\mathrm{U}[a, b]$ is the uniform distribution with parameters a and b. In Step 2, the solutions in *RefSet* are ordered according to quality, where the best solution is the first one in the list. In Step 3, the *NewPairs* set is constructed. *NewPairs* consists of all the new pairs of solutions that can be obtained from *RefSet*, where a "new pair" contains at least one new solution. Since all the solutions are new in the initial *RefSet*, the initial *NewPairs* consists of $(b^2 - b)/2$ pairs. The pairs in *NewPairs* are selected one at a time in lexicographical order to create linear combinations in Step 5. We consider the following three types of linear combinations, where we assume that the reference solutions are $w^{(i)}$ and $w^{(j)}$, $d = r(w^{(i)} - w^{(j)}/2)$ and r is a random number in the range $(0, 1)$:

$$\text{C1:}\quad w = w^{(i)} - d,$$

$$\text{C2:}\quad w = w^{(i)} + d,$$

$$\text{C3:}\quad w = w^{(j)} + d.$$

The following rules are used to generate solutions with these three types of linear combinations:

- If $i \leq b/2$ and $j \leq b/2$ then four solutions are generated by applying C1 and C3 once and C2 twice.
- If $i \leq b/2$ and $j > b/2$ then three solutions are generated by applying C1, C2 and C3 once.
- If $i > b/2$ and $j > b/2$ then two solutions are generated by applying C2 once and randomly choosing between applying C1 or C3.

The solutions created as linear combinations of solutions in the reference set are added to the *NewSolutions* set. Once all combinations have been made, the best b solutions in *NewSolutions* are subjected to the improvement method in Step 6. Each improved solution w is then tested for admission into *RefSet*. If a newly created solution improves upon the worst solution currently in *RefSet*, the new solution replaces the worst and *RefSet* is reordered.

The procedure now intensifies the search around the best-known solution. In Step 7, the counter *IntCount* is increased and the best solution is copied to a temporary memory location w. The solution is perturbed and the improvement method is applied in Step 8. The best solution is updated if the perturbation plus improvement generates a better solution. When the best solution is improved, the intensification count *IntCount* is reset. If *IntLimit* intensification iterations are performed without improving the best solution, the procedure abandons the intensification phase.

Finally, Steps 9 and 10 operate on the entire *RefSet*. Step 9 applies the improvement method to the best $b/2$ solutions in *RefSet* and Step 9 replaces the worst $b/2$ solutions with perturbations of the best $b/2$. The training procedure stops when the number of objective function evaluations (*NumEval*) reaches the total allowed (*TotalEval*). Note that the evaluation of the objective function $g(w)$ consists of the calculation of the mean squared error.

Our procedure employs the well known Nelder and Mead (1965) optimizer as an improvement method. Given a set of weights w, the Nelder and Mead method starts by perturbing each weight to create an initial simplex from which to begin the local search. We use the implementation of the Nelder–Mead method in Press *et al.* (1992) with the following parameters:

$NMAX = 500$, $ALPHA = 1.0$, $BETA = 0.5$, $GAMMA = 2.0$.

Note that the improvement method is used in four different situations during the search: (i) to improve upon the best $b/2$ solution in the initial *RefSet*; (ii) to improve upon the b best solutions that result from the linear combinations; (iii) to improve upon the perturbed solutions generated during the intensification; and (iv) to improve upon the $b/2$ best solutions when rebuilding *RefSet* in Steps 9 and 10.

4. Neural network architecture

The training procedure is applied to a feedforward network with a single hidden layer. We assume that there are n decision variables in the optimization–simulation problem and that the neural network has m hidden neurons with a bias term in each hidden neuron and an output neuron. A schematic representation of the network appears in Fig. 1.

Note that the weights in the network are numbered sequentially starting with the first input to the first hidden neuron. Therefore, the weights for all the inputs to the first hidden neuron are w_1 to w_n. The bias term for the first hidden neuron is w_{n+1}. We test two activation functions for the hidden neurons, where $k = 1, \ldots, m$:

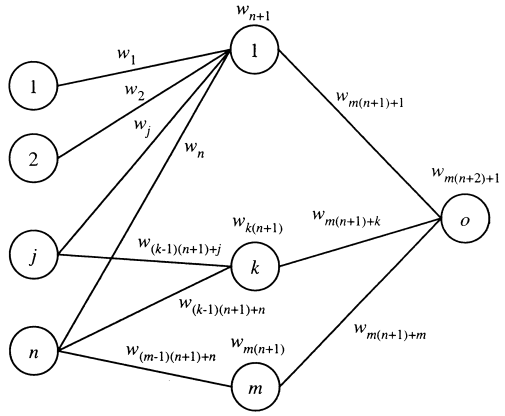

Fig. 1. Neural network with one hidden layer and one output.

$$a_k = \tanh\left(1.5\left(w_{k(n+1)} + \sum_{j=1}^{n} w_{(k-1)(n+1)+j}x_j\right)\right), \quad (1)$$

$$a_k = \frac{1}{1 + \exp\left(-w_{k(n+1)} - \sum_{j=1}^{n} w_{(k-1)(n+1)+j}x_j\right)}. \quad (2)$$

The activation function for the output layer is the identity, which is defined as follows:

$$p(w,x) = w_{m(n+2)+1} + \sum_{k=1}^{n} w_{m(n+1)+k}a_k.$$

Given this architecture and activation functions, we also test two schemes for optimizing w. The first scheme is simply the application of the training procedure listed in Section 3. The second scheme consists of applying the training method to the set of weights associated with hidden neurons and then using linear regression to find the weights associated with the output neuron. In other words, use the training method to find the best set of values for w_1 to $w_{m(n+1)}$ and then apply linear regression to minimize the sum of squares associated with $p(w,x)$. The advantage of the second scheme is that the number of weights that the training procedure needs to adjust is reduced by $m + 1$. The disadvantage, on the other hand, is that the regression model needs to be solved every time any of the first $m(n + 1)$ weights is changed in order to calculate the mean squared error.

Preliminary computational testing showed that better results could be obtained using activation function (2) and linear regression than with any other of the three possible combinations. Therefore, we conducted the rest of our experimentation using activation function (2), i.e., the traditional sigmoid function, and linear regression for the weights associated with the arcs connecting the hidden layer to the output layer.

5. Computational testing

The training procedure uses four parameters, but we have identified two as the most sensitive: b and *IntLimit*. We conducted initial experiments to test all combinations of $b = 8$, 10 and 12 and *IntLimit* = 10, 20, and 30. The best results were obtained with $b = 10$ and *IntLimit* = 20. The other two parameters were set to *PSize* = 100 and (−2, 2) for (*wlow*, *whigh*). In Sexton *et al*. (1998, 1999), the following functions are used to compare the performance of several training procedures for neural networks with one hidden layer:

$$\text{(P1):}\ y = x_1 + x_2,$$

$$\text{(P2):}\ y = x_1 x_2,$$

$$\text{(P3):}\ y = \frac{x_1}{1 + |x_2|},$$

$$\text{(P4):}\ y = x_1^2 + x_2^3,$$

$$\text{(P5):}\ y = x_1^3 + x_1^2,$$

$$\text{(P6):}\ y_t = y_{t-1} + 10.5\left(\frac{0.2y_{t-5}}{1 + (y_{t-5})^{10}} - 0.1y_{t-1}\right).$$

These functions are continuous and differentiable except for (P3) that contains an absolute value. The training set consists of 50 observations with data randomly drawn from [−100, 100] for x_1 and [−10,10] for x_2. An additional set (the testing set) of 150 observations was drawn from the same uniform distributions to test the ability of the neural network to predict y for x values that were not in the training set. The (P6) problem consists of a discrete version of the Mackey–Glass equation that has been used in neural network literature (Gallant and White, 1992; Goffe *et al*., 1994). This function was included in Sexton *et al*. (1998, 1999) because the function's apparent randomness and many local optima makes it challenging for global training algorithms. Five lagged values of the dependent variable are used as the inputs, although clearly three of these are unnecessary. The training set was generated from the starting point (1.6, 0, 0, 0, 0) and the testing set was generating starting from (−0.218 357 539, 0.055 555 36, 1.075 291 525, −1.169 494 128, 0.263 368 033). We use six nodes in the hidden layer and the training procedure 10 times on each function to be able to compare our results with those in Sexton *et al*. (1998, 1999). Table 1 shows the best training error associated with each procedure, where all procedures used a neural network architecture with one hidden layer consisting of six neurons.

In Table 1, BP refers to the backpropagation algorithm in Neural Works Profession II/Plus by NeuralWare. This was chosen among several commercial and freeware packages as the best according to the test results in Sexton *et al*. (1998). SA and GA refer to simulated annealing and genetic algorithm implementations, both in Sexton *et al*. (1999). Finally, TS refer to a tabu search implementation in Sexton *et al*. (1998) and SS refers to our adaptation of scatter search. Since we are interested in a method that can yield good results within a reasonable time, we limit the execution of our procedure to 50 000 objective function evaluations (i.e., *TotalEval* = 50 000). In comparison, BP performed 4.18 million evaluations, SA performed between 112 501 and 12.6 million evaluations, GA performed 100 000 evaluations and TS performed between 190 021 and 928 061 evalutions. According to Table 1 scatter search obtains the best results for the first three problems and problem (P6) across all methods. Problems (P4) and (P5), however, pose difficulties to our method, which is only able to improve upon the outcome of BP.

The mean squared error in Table 2, which corresponds to the testing set, show results that are in line with those obtained in the training set. Clearly, our scatter search adaptation cannot effectively handle problems (P4) and (P5) within 50 000 evaluations. We performed an additional experiment consisting of 10 runs of 500 000 evaluations for problems (P4) and (P5). The best training errors were 6.05E−02 and 7.05E+01 for problems (P4) and (P5), respectively. The best testing errors in the same experiment were 1.07E+00 and 1.81E+02, respectively, for the same problems. Note that while the errors associated with problem (P4) are now within the best, the errors associated with problem (P5) continue to lag behind the best outcomes from other procedures except BP.

The results above were obtained using six nodes in the hidden layer. We performed an additional experiment with this set of problems to test the effects of changing the number of nodes in the hidden layer on both the accuracy of the prediction and the computational time. We ran our training procedure (SS) twice for each problem using two

Table 1. Mean squared error for training set

Problem	*BP*	*SA*	*GA*	*TS*	*SS*
(P1)	5.23E−01	1.05E−05	4.16E−07	1.42E−06	8.93E−08
(P2)	1.10E+01	3.17E−02	1.27E−02	8.38E−02	5.97E−03
(P3)	8.43E+00	1.76E+00	1.82E−01	2.45E−01	2.96E−03
(P4)	1.88E+02	4.84E−01	4.09E−02	4.32E−01	2.31E+00
(P5)	8.57E+03	4.39E−01	3.06E−03	2.56E−02	2.76E+02
(P6)	1.55E−01	1.02E−02	2.53E−02	5.35E−02	3.34E−03

Table 2. Mean squared error for testing set

Problem	*BP*	*SA*	*GA*	*TS*	*SS*
(P1)	1.76E+00	1.56E−05	3.68E−07	2.79E−06	9.37E−07
(P2)	9.11E+01	1.85E−01	1.72E−02	1.96E−01	2.13E−01
(P3)	2.34E+01	5.53E+00	1.38E+00	1.69E+00	1.55E+00
(P4)	4.31E+02	2.73E+00	4.53E−02	1.15E+00	9.26E+00
(P5)	5.28E+04	1.36E+00	2.02E−02	5.68E−02	5.98E+03
(P6)	1.95E−01	2.69E−01	3.40E−02	6.75E−02	2.90E−01

Table 3. SS results when varying the number of hidden nodes

Nodes in hidden layer	*Training (MSE)*	*Testing set (MSE)*	*CPU**
2	5.67E+06	7.73E+06	6.62
6	3.39E+02	1.09E+03	16.00
12	4.59E+02	8.16E+05	61.37

* CPU time measured in seconds on a Pentium III 700 MHz machine.

different random seeds. The average training error out of 12 runs is reported in Table 3, for number of nodes in the hidden layer equal to two, six, and 12. The table reports both the Mean Square Error (MSE) for the training set and the testing set.

The results in Table 3 show that the best average accuracy is obtained with six nodes in the hidden layer. The table also shows that the CPU time increases by a factor of almost four when the number of nodes in the hidden layer is changed from six to 12. The MSE values in Table 3 are deceptively large due to the error associated with problem (P5). The use of six nodes in the hidden layer seems the most appropriate for these problems as suggested in Sexton *et al.* (1998).

In our next experiment, we use data from a discrete event simulation. The simulation consists of a jobshop in which three types of jobs randomly arrive and are processed through a maximum of five machine groups (drills, grinders, lathes, punches and saws). Each job is routed according to its type and the queues in each machine group follow a first-in-first-out discipline. The optimization–simulation problem is to find the optimal number of machines to put in each group in order to minimize the makespan. Clearly, for each combination of machines, the simulation must be executed in order to calculate the makespan. An additional constraint to the problem is that the total number of machines in all groups must not exceed 15. We use the Micro Saint 3.0[1] simulation package to set-up and run the optimization-simulation problem with the embedded optimizer OptQuest.[2] Figure 2 shows the Task Network associated with the simulation model of the jobshop.

A total of 150 simulations were performed and a data file was created in order to use 50 observations for training and 100 observations for testing. (The training set is given in the Appendix.) We use 15 neurons in the hidden layer for this experiment. We run our scatter search adaptation and compare the results with the seven training procedures in Masters (1995). The results of this comparison appears in Table 4, where the labels refer to the following training procedures:

AN1 = simulated annealing 1;
AN2 = simulated annealing 2;
AN1_CJ = simulated annealing 1 with a conjugate gradient search;
AN2_CJ = simulated annealing 2 with a conjugate gradient search;
AN1_LM = simulated annealing 1 with Levenberg–Marquardt search;
AN2_LM = simulated annealing 2 with Levenberg–Marquardt search;
SM = stochastic smoothing.

The number of restarts for these methods was set to one for all the combined methods, to 100 for the simulated annealing versions 1 and 2 and to 200 for stochastic smoothing.

The results in Table 4 show that AN1, AN2 and SM are fast procedures that by an appropriate choice of the number of restarts could be used for on-line training. However, they consistently yield a mean squared error that is larger than alternative approaches. The combined methods (i.e., simulated annealing and a direct descent approach) tend to find solutions of higher quality but employ a training time that is not practical in the on-line context. Note that the combined methods with Levenberg–Marquardt direct descent learning do not perform many objective function evolutions, however, they consume a significant amount of computer time applying a singular value decomposition procedure with the corresponding back-substitution for solving a linear system of equations. Our scatter search implementation is capable of finding a set of weights with a MSE value comparable

[1] Micro Saint is a trademark of Micro Analysis and Design (*www.maad.com*).

[2] OptQuest is a trademark of OptTek Systems, Inc. (*www.opttek. com*).

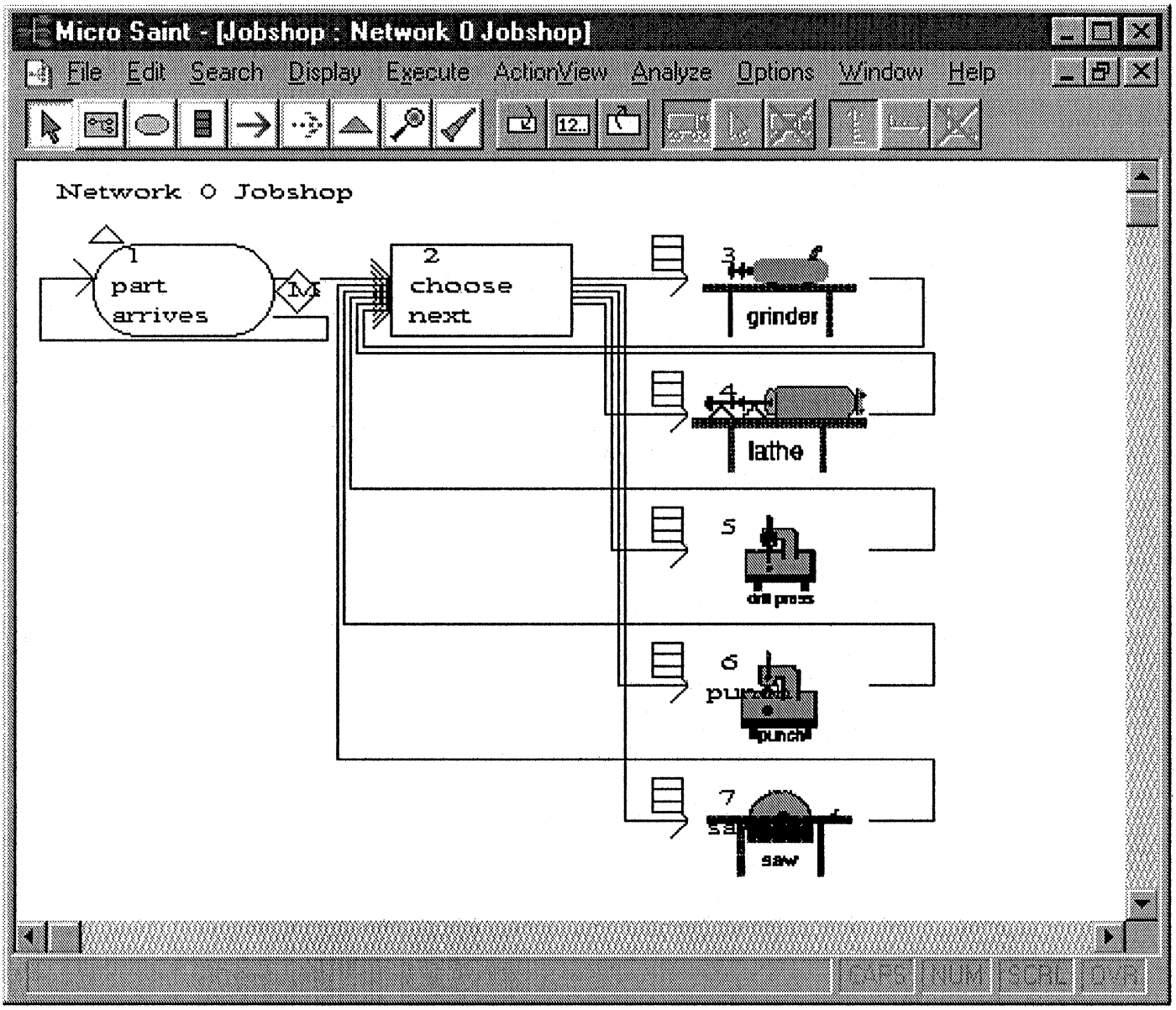

Fig. 2. Micro Saint simulation model of a jobshop.

Table 4. Comparison of training procedures using data from a discrete event simulation

Method	*MSE*	*Evaluations*	*CPU**
AN1	213.534	38,228	89.2
AN2	148.813	49,003	114.0
AN1_CJ	4.479	855,159	510.9
AN2_CJ	40.396	360,746	201.8
AN1_LM	4.714	5,588	355.6
AN2_LM	4.827	2,383	234.7
SM	397.867	52,715	116.5
SS	4.849	62,802	51.4

* CPU time measured in seconds on a Pentium III 700 MHz machine.

to the best combined methods and within a computational time that is reasonable for on-line training.

We use the best set of weights found with our SS implementation to predict the output of the simulation in the 100 observations in our testing set. We apply the criterion defined at the end of Section 2 to determine whether or not a solution should be sent to the simulator for evaluation. The best-known solution was found in simulation 23 and has a makespan of $f(x^*) = 254.710$ (see Appendix). The sample standard deviation of the makespan values in the training set is $\hat{\sigma}_{f(x)} = 47.037$, therefore any solution with a makespan of $254.710 + 2 \times 47.037 = 348.783$ will be discarded. This criterion results in the

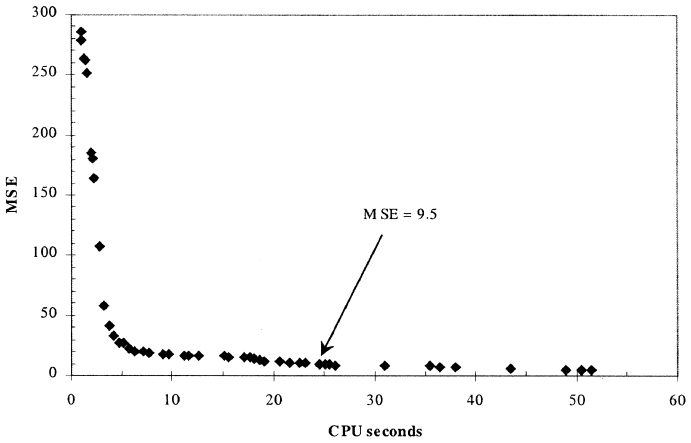

Fig. 3. Performance graph of scatter search training.

elimination of 38 out of 100 solutions in the testing set. The testing set includes a better solution than the current best-known solution. The new best solution is the 10th observation in the testing set, which corresponds to the 60th solution generated during the optimization–simulation process. This solution has a makespan of 245.818 and the neural network predicts a value of 301.06. The solution, however, is not discarded because the predicted value falls within the allowable range for the solutions that are sent to the simulator. This example shows that an aggressive definition of α could cause the rejection of good solutions before they are evaluated.

We have mentioned that a desirable feature for an on-line training procedure is the ability to quickly find good values for the set of weights w. In Fig. 3, we show the performance graph for the training of the neural network associated with the jobshop data. Note that within 5 CPU seconds, the scatter search procedure is able to find solutions to the training problem that yield a MSE of less than 30. This is quite remarkable, considering that three of the procedures in Table 4 do not achieve this level of accuracy even when employing significantly more computational time. An accuracy of 10, which given the relative magnitude of the makespan values in our simulation may be sufficient, is reached after about 25 CPU seconds (see Fig. 3).

We have also measured the convergence of AN_CJ, AN1_LM and AN2_LM and determined to be significantly slower than our scatter search implementation.

6. Conclusions

In this paper we have described the implementation of scatter search for training a single layer feed-forward neural network. Our goal was to develop a training procedure that could be used on-line. Specifically, we were interested in a procedure that could train a neural network with data generated by a simulator during an optimization–simulation search. In such a context, the training must be fast while assuring a reasonable accuracy level. Our experiments show that the scatter search implementation reaches a prediction accuracy that is sufficient for the purpose of filtering out potentially bad solutions generated during the optimization of a simulation, and it does so within a computational time that is practical for on-line training.

Acknowledgements

M.L. was partially supported by the Visiting Professor Fellowship Program of the University of Valencia (Grant Ref. No. 42743).

References

Gallant, R.A. and White, H. (1992) On learning the derivatives of an unknown mapping with multilayer feedforward networks, in *Artificial Neural Networks*, Blackwell, Cambridge, MA, 206–223.

Goffe, W.L., Ferrier, G.D. and Rogers, J. (1994) Global optimization of statistical functions with simulated annealing. *Journal of Econometrics*, **60**, 65–99.

Glover, F. (1998) A template for scatter search and path relinking, in *Artificial Evolution*, *Lecture Notes in Computer Science* 1363, Hao, J.-K., Lutton, E., Ronald, E., Schoenauer, M. and Snyers, D. (eds.), Springer-Verlag, pp. 13–54.

Glover, F. and Kelly, J. (1998) Combining simulation and optimization for improved business decisions. *Colorado Business Review*, **LXIV**, (4), 2–3.

Glover, F., Laguna, M. and Martí, R. (1999) Scatter search, in *Theory and Applications of Evolutionary Computation: Recent Trends*, Ghosh, A. and Tsutsui, S. (eds.), Springer-Verlag, (to appear).

Laguna, M. (2000) Scatter search, in *Handbook of Applied Optimization*, Pardalos, P.M. and Resende, M.G.C. (eds.), Oxford Academic Press, (to appear).

Laguna, M. and Marti, R. (2000) "The OptQuest Callable Library", in *Optimization Software Class Libraries*, Voss, S. and Woodruff, D.L. (eds), Kluwer Academic Publishers, Boston, MA (to appear).

Masters, T. (1995) *Neural, Novel and Hybrid Algorithms for Time Series Prediction*, John Wiley.

Nelder, J.A. and Mead, R. (1965) A simplex method for function minimization. *Computer Journal*, **7**, 308–313.

Press, W.H., Teukolsky, S.A., Vetterling, W.T. and Flannery, B.P. (1992) *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press.

Sharda, R. and Rampal, R. (1995) Neural networks and management science/operations research. Oklahoma State University, *http://catt.okstate.edu/itorms/guide/nnpaper.html*.

Sharda, R. (1994) Neural networks for the MS/OR analyst: an application bibliography. *Interfaces*, **24**(2), 116–130.

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

Sexton, R.S., Dorsey, R.E. and Johnson, J.D. (1999) Optimization of neural networks: a comparative analysis of the genetic algorithm and simulated annealing. *European Journal of Operational Research*, **114**, 589–601.

Appendix

The following table shows the training set for the experiment with the discrete event simulation data.

Solution	*Drills*	*Grinders*	*Lathes*	*Punches*	*Saws*	*Makespan*
1	1	1	2	3	3	358.141
2	1	1	1	1	1	403.568
3	1	1	1	2	5	414.315
4	1	2	5	1	1	382.613
5	1	1	2	1	5	405.845
6	1	1	2	5	1	353.075
7	1	2	1	1	5	392.893
8	1	2	1	5	1	386.072
9	3	4	1	1	1	341.865
10	1	1	4	3	1	376.481
11	2	1	1	1	5	409.489
12	3	1	1	3	1	395.398
13	1	3	2	3	1	408.547
14	2	2	2	1	2	274.508
15	2	1	2	2	2	336.803
16	2	2	2	2	1	290.547
17	1	3	1	2	3	386.049
18	4	1	1	2	2	301.243
19	2	3	1	1	3	330.720
20	2	2	1	2	2	331.910
21	1	1	3	2	3	370.548
22	1	4	3	1	1	401.984
23	4	2	2	1	1	254.710
24	1	1	1	3	3	380.588
25	1	1	5	1	1	342.081
26	3	1	1	2	3	336.231
27	1	1	1	2	4	414.315
28	1	5	1	1	1	396.135
29	3	2	2	1	1	278.051
30	1	3	1	1	3	421.979
31	3	3	1	2	1	414.263
32	2	2	2	3	1	290.547
33	2	1	2	3	1	323.616
34	2	2	2	1	1	265.044
35	3	2	2	1	2	281.245
36	2	2	2	2	2	260.519
37	5	2	1	1	1	355.913
38	1	1	4	1	1	342.081
39	1	1	4	2	1	374.974
40	1	1	3	2	1	374.974
41	1	1	1	5	1	428.242
42	2	1	1	2	3	347.726
43	3	1	1	3	2	349.564
44	5	1	1	1	2	411.760
45	1	2	4	1	1	382.613
46	1	2	3	1	1	382.613
47	1	2	2	1	1	390.765
48	1	1	2	4	2	361.593
49	1	1	2	2	4	358.141
50	1	1	1	2	3	380.588

Biographies

Manuel Laguna is Associate Professor of Operations Management in the College of Business and Administration and Graduate School of Business Administration of the University of Colorado at Boulder. He received Master's and Doctoral degrees in Operation Research and Industrial Engineering from the University of Texas at Austin. He has done extensive research in the interface between computer science, artificial intelligence and operations research to develop solution methods for problems in areas such as production planning and scheduling, routing and network design in telecommunications, combinatorial optimization on graphs, and optimization of simulations. Dr. Laguna has more than 50 publications, including articles in scientific journals such as *Operations Research*, *Management Science*, *European Journal of Operational Research*, *Computers and Operations Research*, *IIE Transactions*, and the *International Journal of Production Research*. He is the co-author of *Tabu Search*; the first book devoted to this innovative optimization technology. He is editor-in-chief of the *Journal of Heuristics*, is in the editorial board of *Combinatorial Optimization*: *Theory and Practice* and has been guest editor for the *Annals of Operations research*. Dr. Laguna is a member of the Institute for Operations Research and the Management Science, the Institute of Industrial Engineering, and the International Honor Society Omega Rho.

Rafael Marti is an Associate Professor in the Department of Statistics and Operations Research at the University of Valencia in Spain. He has a doctoral degree in Mathematics from the University of Valencia. He has done extensive research in the field of metaheuristics and is currently Associate Editor of the *Journal of Heuristics*.

Contributed by the Computer Technologies and Information Systems Department