Discovering Efficient Learning Rules for Feedforward Neural Networks using Genetic Programming

Amr Radi
formerly with the
School of Computer Science,
University of Birmingham, UK.
Amr_Radi@hotmail.com

Riccardo Poli
Department of Computer Science
University of Essex,
Wivenhoe Park
Colchester, CO4 3SQ, UK
rpoli@essex.ac.uk

Techichal Report CSM-360
Department of Computer Science
University of Essex
January 2002

Abstract

The Standard BackPropagation (SBP) algorithm is the most widely known and used learning method for training neural networks. Unfortunately, SBP suffers from several problems such as sensitivity to the initial conditions and very slow convergence. Here we describe how we used Genetic Programming, a search algorithm inspired by Darwinian evolution, to discover new supervised learning algorithms for neural networks which can overcome some of these problems. Comparing our new algorithms with SBP on different problems we show that these are faster, are more stable and have greater feature extracting capabilities.

1 Introduction

Supervised learning algorithms are by far the most frequently used methods to train Artificial Neural Networks (ANNs). The Standard Back Propagation (SBP) algorithm [1] was the first method to be discovered that is capable of training multilayer networks. It has been applied to a number of learning tasks in science, engineering, finance and other disciplines [2, 3]. Indeed, the SBP learning algorithm has emerged as the standard algorithm for the training of multilayer networks, and hence the one against which other learning algorithms are usually benchmarked [2, 4, 5, 6].

Unfortunately, SBP presents several drawbacks [7, 2, 8, 9, 10, 11, 12, 13]: it is extremely slow; training performance is sensitive to the initial conditions; it may become trapped in local minima before converging to a solution; oscillations may occur during learning (this usually happens when users increase the learning rate in an unfruitful attempt to speed up convergence); and, if the error function is shallow, the gradient is very small leading to small weight changes.

Consequently, in the past few years, a number of improvements to SBP have been proposed in the literature (see [11] for a survey). These algorithms are generally significantly faster than the SBP (being up to one order of magnitude quicker) but still suffer from some of the problems mentioned above. We will review the SBP algorithm and discuss some of these recent improvements in the next section.

Efforts continue in the direction of solving these problems to produce faster supervised learning algorithms and to improve their reliability. However, progress is extremely slow because any new rule has to be designed by a human expert using engineering and/or mathematical principles and then tested extensively to verify its functionality and efficiency. In addition, most newly proposed algorithms are neither very different from nor much better than the previous ones. This is because scientists tend to search the space of possible learning algorithms for neural nets using a kind of "gradient descent", i.e. by only marginally modifying pre-existing algorithms. This method of searching may take a long time to lead to significant breakthroughs in the field. Indeed, looking critically at the vast literature on this topic, it can be inferred that only a handful of really novel algorithms with demonstrably significantly better performance (in terms of speed and stability) than SBP have been produced in the last 15 years [14, 15, 16, 17, 18, 19, 20].

The process of discovering new learning rules for ANNs is a search process. As such, there is no reason to believe that it cannot be somehow automatised. So, instead of carrying out extensive trial-and-error experimentation to optimise neural network learning rules, optimal or near-optimal learning rules for a particular application could be sought for with search and optimisation methods. This has led some researchers to use optimisation algorithms to explore, at least locally, the space of possible learning rules for ANNs. Because of the limited knowledge we have on this space, the tools of choice have been Evolutionary Algorithms (EAs), such as Genetic Algorithms (GAs) [21, 22] and Genetic Programming (GP) [23], which, although not optimal for some domains, offer the broadest possible applicability [24].

GAs [21, 22] are search algorithms which emulate the mechanics of natural evolution. GAs operate on a population of individuals, each of which consists of a string, called a chromosome, that encodes a solution to the problem being solved. Different individuals may explore different regions of the search space of the problem in parallel. At each iteration, called a generation, a new population is created using probabilistic rules which produce new individuals using the information contained in pairs of parent chromosomes (this is called crossover or mating). Parents of above average quality (normally called fitness) are given more chances to mate.

A number of researchers have already applied GAs to the synthesis of application specific ANNs [25]. Some have used GAs to determine the connection weights in ANNs (e.g. [26]), while others have used them to find the optimal topology and parameters of a certain learning rule for ANNs (see for example [27, 28, 29]).

A few researchers (e.g. [30]) have used GAs to produce optimum ANN learning rules, by finding appropriate coefficients for rules with certain prefixed functional forms. We will review this work later in Section 3. However, by replicating some of this work ourselves, we have soon realised that fixing the class of rules that can be explored biases the search and prevents the evolutionary algorithm from exploring the much larger space of rules which we, humans, have not thought about. So, in line with some work by Bengio [15], which is also summarised in Section 3, we decided to use Genetic Programming [23] to perform this search as GP allows the direct evolution of symbolic learning rules with their coefficients (if any) rather than the simpler evolution of parameters for a fixed learning rule.

We describe our approach in Section 4. Section 5 reports the experimental results obtained on several benchmark problems. We discuss these results in Section 6 and we draw some conclusions in Section 7.

2 Standard Backpropagation Algorithm and Recent Improvements

A multilayer perceptron is a feed-forward neural network in which an arbitrary input vector is propagated forward through the network, causing an activation vector to be produced in the output layer [2]. The network behaves like a function which maps the input vector onto an output vector. This function is determined by the structure and the connection weights of the net. The objective of SBP is to tune the weights of the network so that the network performs the desired input/output mapping. In this section we briefly recall the basic concepts of multilayer feed-forward neural networks, the SBP algorithm and some of its recent improvements. More details can be found in [10, 11].

2.1 Standard Backpropagation

Let u_i^l be the i^{th} neuron in the l^{th} layer (the input layer is the 0^{th} layer and the output layer is the k^{th} layer). Let n_l be the number of neurons in the l^{th} layer. The weight of the connection between neuron u_j^l and neuron u_i^{l+1} is denoted by w_{ij}^l. Let $\{x_1, x_2, ..., x_m\}$ be the set of input patterns that the network is supposed to learn and let $\{t_1, t_2, ..., t_m\}$ be the corresponding target output patterns. The pairs (x_p,t_p) $p = 1, .., m$ are called training patterns. Each x_p is an n_o-dimensional vector with components x_{ip}. Each t_p is an n_k-dimensional vector with components t_{ip}.

The output o_{ip}^0 of a neuron u_i^0 in the input layer, when pattern x_p is presented to the network, coincides with its net input net_{ip}^o, i.e. with the i^{th} element of x_p, x_{ip}. For the other layers, the net input net_{ip}^{l+1} of neuron u_i^{l+1} (when the input pattern x_p is presented to the network) is usually computed as follows:

$$net_{ip}^{l+1} = \sum_{j=1}^{n_l} w_{ij}^l o_{jp}^l - \theta_i^{l+1},$$

where o_{jp}^l, is the output of neuron u_j^l (usually $o_{jp}^l = f(net_{jp}^l)$ with f a non-linear activation-function) and θ_i^{l+1} is the bias of neuron u_i^{l+1}. For the sake of a homogeneous representation, in the following, the bias will be interpreted as the weights of a connection to a 'bias unit' with a constant output of 1.

The error ε_{ip}^k for neuron u_i^k of the output layer for the training pair (x_p,t_p) is computed as

$$\varepsilon_{ip}^k = t_{ip} - o_{ip}^k.$$

For the hidden layers the error ε_{ip}^l is computed recursively from the errors on other layers (see [2]). The SBP rule uses these errors to adjust the weights (usually initialised randomly) in such a way that the errors gradually reduce.

The network performance can be assessed using the Total Sum of Squared (TSS) errors given by the following function:

$$E = \frac{1}{2}\sum_{p=1}^{m}\sum_{i=1}^{n_k}\left(\varepsilon_{ip}^k\right)^2.$$

The training process stops when the error E is reduced to an acceptable level, or when no further improvement is obtained.

In the batched variant of the SBP the updating of w_{ij}^l in the s^{th} learning step (often called an "epoch") is performed according to the following equations:

$$w_{ij}^l(s+1) = w_{ij}^l(s) + \Delta w_{ij}^l(s)$$

$$\Delta w_{ij}^l(s) = \eta \frac{\partial E}{\partial w_{ij}^l(s)} = \eta \delta_{ip}^{l+1}(s) o_{jp}^l(s)$$

where $\delta_{ip}^{l+1}(s)$ refers to the error signal at neuron i in layer $l+1$ for pattern p at epoch s, which is the product of the first derivative of the activation function and the error $\varepsilon_{ip}^{l+1}(s)$, and η is a parameter called learning rate.

2.2 Improvements to SBP

Many methods have been proposed to improve generalisation performance and convergence time of SBP. Current research mostly concentrates on: the optimum setting of learning rates and momentum (see below) [31, 32, 33, 9, 13, 34, 35, 36]; the optimum setting of the initial weights [37, 38, 39]; the enhancement of the contrast in the input patterns [40, 41, 42, 43, 44]; changing the error function [45, 32, 46, 47, 48, 49]; finding optimum architectures using pruning techniques [50, 51]. In the following we will describe two speed-up methods which are relevant to the work described in later sections: the Momentum method and Rprop.

The Momentum method implements a variable learning rate coefficient implicitly by adding to the weight change a fraction of the last weight change as follows:

$$\Delta w_{ij}^l(s) = \eta \frac{\partial E(s)}{\partial w_{ij}^l(s)} + \mu \frac{\partial E(s-1)}{\partial w_{ij}^l(s-1)}$$

where μ is a parameter called momentum. This method decreases the oscillation which may occur with large learning rates and accelerates the convergence. For a more detailed discussion see [32, 33, 34].

Rprop is one of the fastest variations of the SBP algorithm [9, 11, 52]. Rprop stands for 'Resilient backpropagation'. It is a local adaptive learning scheme, performing supervised batch learning. The basic principle of Rprop is to eliminate the harmful influence of the magnitude of the partial derivative $\frac{\partial E}{\partial w^l_{ij}}$ on the weight changes. The sign of the derivative is used to indicate the direction of the weight update while the magnitude of the weight change is exclusively determined by a weight-specific update-value $\Delta^l_{ij}(s)$ as follows:

$$\Delta w^l_{ij}(s) = \begin{cases} -\Delta^l_{ij}(s) & \text{if } \frac{\partial E(s)}{\partial w^l_{ij}(s)} > 0, \\ +\Delta^l_{ij}(s) & \text{if } \frac{\partial E(s)}{\partial w^l_{ij}(s)} < 0, \\ 0 & \text{otherwise.} \end{cases}$$

The update-values $\Delta^l_{ij}(s)$ are modified according to the following equation:

$$\Delta^l_{ij}(s) = \begin{cases} \eta^+ \Delta^l_{ij}(s-1) & \text{if } \frac{\partial E(s-1)}{\partial w^l_{ij}(s-1)} \cdot \frac{\partial E(s)}{\partial w^l_{ij}(s)} > 0, \\ \eta^- \Delta^l_{ij}(s-1) & \text{if } \frac{\partial E(s-1)}{\partial w^l_{ij}(s-1)} \cdot \frac{\partial E(s)}{\partial w^l_{ij}(s)} < 0, \\ \Delta^l_{ij}(s-1) & \text{otherwise.} \end{cases}$$

where η^- and η^+ are constants such that $0 < \eta^- < 1 < \eta^+$.

The Rprop algorithm has three parameters: the initial update-value $\Delta^l_{ij}(0)$, which directly determines the size of the first weight step (default setting $\Delta^l_{ij}(0)$=0.1), and the limits for the step update values Δ_{max} and Δ_{min} which prevent the weights from becoming too large or too small. Typically Δ_{max}=50.0 and Δ_{min}=0.0000001. Convergence with Rprop is rather insensitive to the choice of these parameters. Nevertheless, for some problems it can be advantageous to allow only very cautious (namely small) steps, in order to prevent the algorithm from getting stuck too quickly in local minima. For a detailed discussion see also [9, 10].

Although the Momentum method and Rprop are considerably faster than SBP, they still suffer from same of the problems mentioned in Section 1 [53, 9].

3 Previous Work on the Evolution of Neural Network Learning Rules

A considerable amount of work has been done on the evolution of the weights and/or the topology of neural networks. See for example [54, 27, 28, 29, 26, 25]. However, only a relatively small amount of previous work has been reported on the evolution of learning rules for neural networks [30, 55, 56, 57, 58, 15].

Some researchers have used GAs to perform the adaptive adjustment of the SBP algorithm's parameters, such as the learning rate and the momentum. For example, Belew *et al.* [59] used GAs to find parameters for the SBP algorithm while the architecture was fixed. Harp *et al.* [60] evolved both the SBP algorithm's modification parameters and its architecture simultaneously. Chalmers [30] applied GAs to discover learning rules for single-layer feed-forward ANNs with sigmoid output units. Being an important example, we describe this approach in more detail.

Chalmers fixed the changes in the weight of a given connection to be a function only of information local to that connection (different kinds of learning task were considered: supervised learning, reinforcement learning and unsupervised learning). The same function was employed for every connection. The search was limited to quadratic functions of $\{w^l_{ij}, o^l_{jp}, t_{ip}, o^l_{ip}\}$. The functions had the form:

$$\begin{aligned} \Delta w_{ij} &= k_o(k_1 w_{ij} + k_2 o_{jp} + k_3 o_{ip} + k_4 t_{ip} \\ &\quad + k_5 w_{ij} o_{jp} + k_6 w_{ij} o_{ip} + k_7 w_{ij} t_{ip} \\ &\quad + k_8 o_{jp} o_{ip} + k_9 o_{jp} t_{ip} + k_{10} t_{ip} o_{ip}). \end{aligned}$$

The chromosomes encoded the eleven coefficients k_c. Each chromosome consisted of 35 bits. The first five bits coded the scale parameter k_o. The other 30 bits encoded the other ten coefficient in groups of three. In each parameter, the first bit encoded the sign, and the other two bits encoded the magnitude. Note that when the k_c coeeficients take the values 4, 0, 0, 0, 0, 0, 0, 0, -2, 2, 0, one obtains the formula

$$\Delta w_{ij} = 4(k - 2o_{jp}o_{ip} + 2o_{jp}t_{ip})$$

which represents a variant of the well known Delta rule. The fitness of a learning rule was calculated as the mean error over a number (typically 20) of tasks randomly selected at the start of every evolutionary run. The GA used a population of 40 individuals, two-point crossover and elitist selection and was run for 1000 generations. As a results of the evolutionary runs, evolution discovered the well-known Delta rule (shown above) and some of its variants.

Similar experiments on the evolution of learning rules were also carried out by others [58, 55]. Mitchell *et al.* [55] used Chalmers' approach to evolve learning rules for binary perceptrons. Dasdan [58] also used Chalmers' approach to evolve learning rules for unsupervised learning algorithms. He used sixteen parameters in his function with similar variables as in [30]. Results of the evolutionary runs have been compared with the classical self-organizing map learning algorithm [61] and have been shown to provide better performance.

All the GA-based methods mentioned above are limited as they impose a fixed number of parameters and a rigid form for the learning rules. Chalmers noticed that discovering complex learning rules using GAs is not easy because either a) one uses a highly complex genetic coding which makes the search space very large and hard to explore, or b) one uses a simpler coding which allows known learning rules as a possibility, making the search very biased.

Genetic Programming is an extension of GAs in which the structures that make up the population under optimisation are not fixed-length character strings that encode possible solutions to a problem, but programs that, when executed, are the candidate solutions to the problem [23, 62]. Typically GP uses a variable size representation in which programs are represented as trees with functions as internal nodes and variables or constants (terminals) as leaves. It is arguable that GP, thanks to its variable size and shape representation for individuals, may be a good way of getting around the limitations shown by GAs when tackling the problem of evolving ANN learning rules. Indeed Chalmers suggested that GP might have an advantage over GA in that, under evolutionary pressures, syntax trees might become arbitrarily complex if needed.

GP has been applied successfully to a large number of difficult problems like automatic design, pattern recognition, robotics control, synthesis of neural networks, symbolic regression, music and picture generation, etc [63]. However, only one attempt to use GP to induce new learning rules for neural networks has been reported before our own work. This is briefly discussed below.

Bengio [15] used GP to find ANN learning rules. He used the output of the input neuron o^l_{jp}, the error of the output neuron ε^{l+1}_{ip} and the first derivative of the activation function of the output neuron as terminals for GP and algebraic operators as functions. Bengio used a very strong search bias towards a certain class of SBP-like learning rules as only the ingredients to rediscover the SBP algorithm were used. GP found a better learning rule compared to the rules discovered by simulated annealing and GAs. Bengio's rule [15] (that we will term NLR_B) is similar to the SBP learning rule but it includes the cube of the first derivative of the activation function. In the case of a logistic activation function NLR_B can be represented as follows:

$$\mathrm{NLR}_B = \eta o^l_{jp}[o^{l+1}_{ip}(1 - o^{l+1}_{ip})]^3 \varepsilon^{l+1}_{ip}.$$

In the case of a hyperbolic tangent activation function, this becomes:

$$\mathrm{NLR}_B = \frac{1}{8}\eta o^l_{jp}({o^{l+1}_{ip}}^2 - 1)^3 \varepsilon^{l+1}_{ip}.$$

Unfortunately, the new learning rule suffered from the same problems as SBP and was only tested on a very specific problem.

4 Our Approach to Evolving Learning Rules with GP

In this section we will describe our approach to discovering learning rules based on GP.

Our work is an extension of Bengio's work with the objective to explore a larger space of rules using different parameters and different rules for the hidden and output layers. Our long term objective is to obtain a rule which is general, like SBP, but is faster and more stable than SBP. We want to discover learning rules of the following form:

$$\Delta w^l_{ij}(s) = \begin{cases} F_o(w^l_{ij}, o^l_{jp}, t_{ip}, o^{l+1}_{ip}) & \text{for the output layer,} \\ F_h(w^l_{ij}, o^l_{jp}, o^{l+1}_{ip}, \varepsilon^{l+1}_{ip}) & \text{for the hidden layers} \end{cases}$$

where o^l_{jp} is the output of neuron u^l_j when pattern p is presented to the network and $\varepsilon^{l+1}_{ip} = \sum_i w^{l+1}_{ji}\delta^{l+2}_{ip}$. So, in our approach we use two different learning rules one for the output layer and one for the hidden layers, like in the SBP learning rule.

In preliminary tests in which we tried to evolve both rules at the same time we obtained relatively poor results. This has to be attributed to the huge size of the search space and to the limited memory and CPU power of current workstations. So, we decided to proceed in two stages. In the first stage, we used GP to evolve rules for the output layer, while the hidden layers were trained with the SBP rule [64]. In the second stage, we used GP to evolve rules for the hidden layers, while the output layer was trained with the best rule discovered in the first stage.

5 Experiments

The investigations were performed using our own SBP and GP simulators. The simulators are written in POP11. The experiments were run on a Digital Alpha machine with a 233 MHz processor.

5.1 Stage One: Learning Rules for Output Layers

In this stage we used GP to evolve rules for the output layer, while the hidden layers were trained with SBP.

5.1.1 GP Parameters

In our approach, the fitness of each learning rule was computed using the TSS error E for the XOR problem:

$$F = \begin{cases} \lambda(E_{max} - E) & \text{if the network does not learn,} \\ C_{max} - C_{min} & \text{otherwise,} \end{cases}$$

where C_{min} is the average minimum number of epochs needed for convergence and E_{max} and C_{max} are constants such that $F \geq 0$. λ is a factor that makes the value of $(E_{max} - E)$ greater than $C_{max} - C_{min}$ in any condition. The value of E is measured at the maximum number of learning epochs (1000).

GP was run for 500 generations with a population size of 1000 and crossover probability 0.9. After crossover, mutation was applied to all of the population with a probability of 0.01. The function set was $\{+, -, \times\}$, and the terminal set was $\{w^l_{ij}, o^l_{jp}, t_{ip}, o^{l+1}_{ip}, 1, 2\}$. The "full" initialisation method [23] with an initial maximum depth of 3, and tournament selection with a tournament size of 4 were used.

5.1.2 Testing Problems

In this set of experiments we considered the following four problems: 1) the XOR problem; 2) the family of N-M-N encoder problems which force the network to generalise and to map input patterns onto similar output activations [65]; 3) a character recognition problem with 7 inputs representing the state of a 7-segment light emitting diode (LED) display and 4 outputs representing the digits 1 to 9 encoded in binary [15]; 4) the display problem with 4 inputs which represent a digit from 1 to 9 in binary and 7 outputs which represent the LED configuration to visualise the digit.

For the XOR problem, we used a three-layer network consisting of 2 input, 1 output and 2 hidden neurons, with hyperbolic tangent activation functions having output in the range [-1,1]. The weights were randomly initialised within the range [-1,1]. The maximum number of learning epochs was 1000. For the encoder problems, we used a three-layer network consisting of 10 input, 5 hidden, and 10 output neurons. Here, logistic activation functions with output in the range of [0,1] were used. The maximum number of learning epochs was 500. For the character recognition problems, we used a three-layer network consisting of 7 input, 10 hidden and 4 output neurons and logistic activation functions with output in the range of [-1,1]. The maximum number of learning epochs was 500. For the display problems we used a three-layer network consisting of 4 input, 10 hidden and 7 output neurons and we used logistic activation functions with output in the range of [0,1]. The maximum number of learning epochs was 500. A threshold criterion was used to produce binary outputs similar to those used by digital logic designers: if the output range was [0,1], any value below 0.4 was considered to be to be a zero, any value above 0.6 to be a one and all others were discounted (if the output was in range from 0.4 to 0.6, it would not count).

These parameters and network topologies were determined experimently with different configurations (we varied number of layers, number of hidden layers, learning rate, and range of random initial weights). The ones on which the SBP algorithm worked best were adopted. Each problem was tested with 100 independent runs (i.e. each run with different initial weights).

5.1.3 Results

In the first stage, GP has discovered a useful way of using the Delta learning rule, originally developed for single-layer neural networks, to speed up learning. The rule is:

$$\mathrm{NLR}_o = \Delta w_{ij}^l(s) = \eta o_{jp}^l \varepsilon_{ip}^{l+1}$$

In a set of tests we compared the convergence behaviour of NLR_o against SBP with and without the Rprop speed-up algorithm. Figures 1 - 4 show typical runs of SBP, NLR_o, SBP with Rprop and NLR_o with Rprop. The runs in each plot used the same initial random weights.

More precisely, Figure 1 shows the TSS error for SBP and NLR_o, with and without Rprop, on the character recognition problem. The results obtained indicate that $\mathrm{NLR}_o + \mathrm{Rprop}$ achieves its target output in approximately 30 epochs while $SPB + \mathrm{Rprop}$ takes 40 epochs. Figure 2 shows the same data for the display problem. SBP achieves its target value after 500 epochs and NLR_o after 150 epochs. This is a three-fold speed-up. In addition, $\mathrm{NLR} + \mathrm{Rprop}$ achieves its target error at approximately 40 epochs while $\mathrm{SBP} + \mathrm{Rprop}$ takes 60 epochs. Figure 3 confirms that NLR_o also outperforms SBP on the XOR problem. In this case $\mathrm{SBP} + \mathrm{Rprop}$ required 110 epochs to converge, while NLR_o+Rprop took 40 epochs. Figure 4 shows that NLR_o outperforms SBP on the encoder problem as well.

So, NLR_o appears to be much faster than SBP. Also, NLR_o remains much faster than SBP when both are applied with Rprop. Interestingly, the $\mathrm{NLR}_o + \mathrm{Rprop}$ algorithm seems to be able to solve all problems in nearly constant time, despite the considerable differences in complexity of such problems.

To clarify how these speedups are achieved in Figure 5 we show the ratio between the Δw_{ij}^l of NLR_o and that of SBP for the logistic activation function, $\frac{1}{o_{ip}^{l+1}(1-o_{ip}^{l+1})}$, where o_{ip}^{l+1} is between 0 and 1. Figure 6 shows the ratio between NLR and SBP for a hyperbolic activation function, $\frac{1}{2(1-(o_{ip}^{l+1})(o_{ip}^{l+1}))}$, where o_{ip}^{l+1} is between -1 and 1. In both cases effectively NLR_o increases the learning rate for neurons which are close to saturation, thus correcting a well-known problem of SBP.

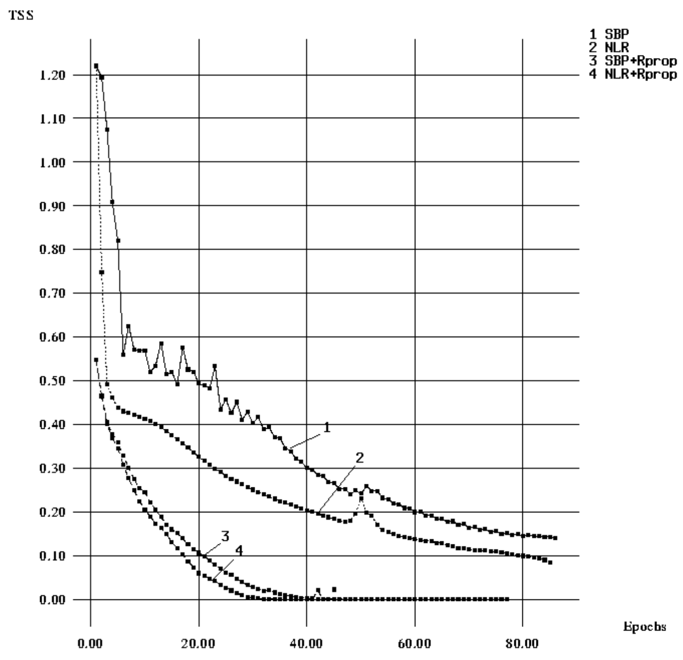

Figure 1: Plots of the TSS error for SBP and NLR_o in the character recognition problem.

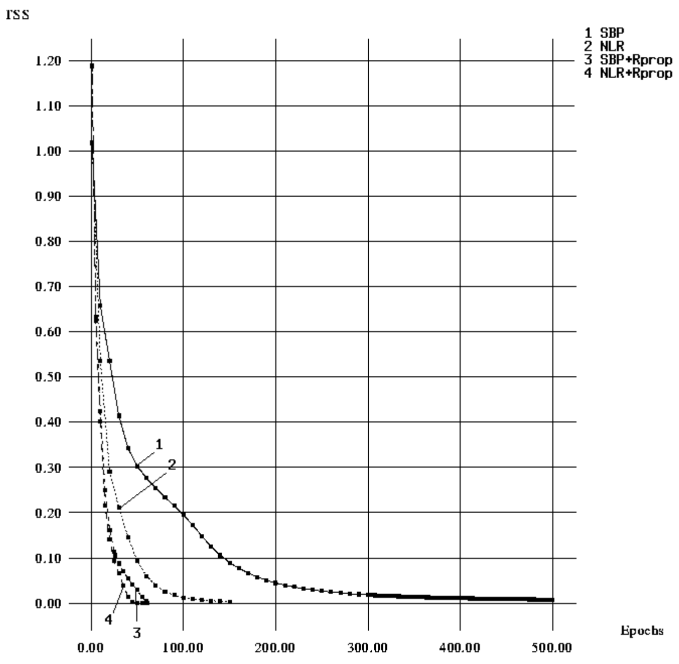

Figure 2: Plots of the TSS error for SBP and NLR_o in the display problem.

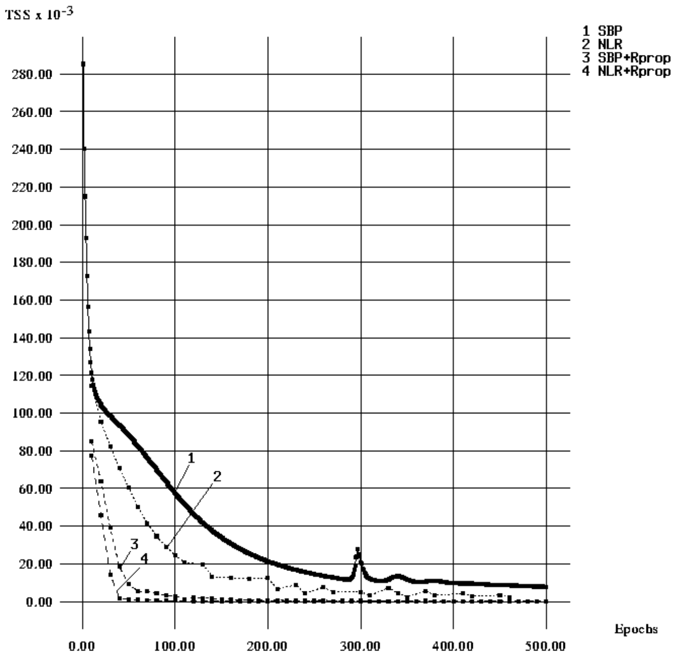

Figure 3: Plots of the TSS error for SBP and NLR_o in the XOR problem.

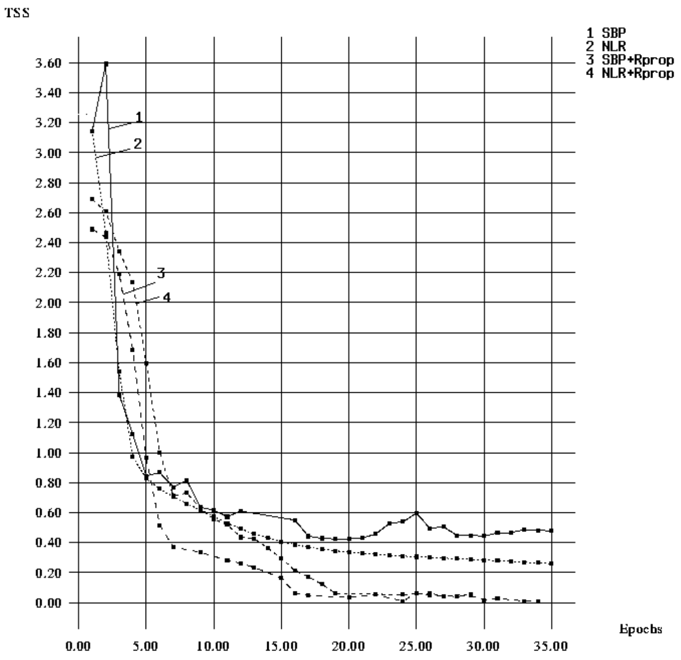

Figure 4: Plots of the TSS error for SBP and NLR_o in the encoder problem.

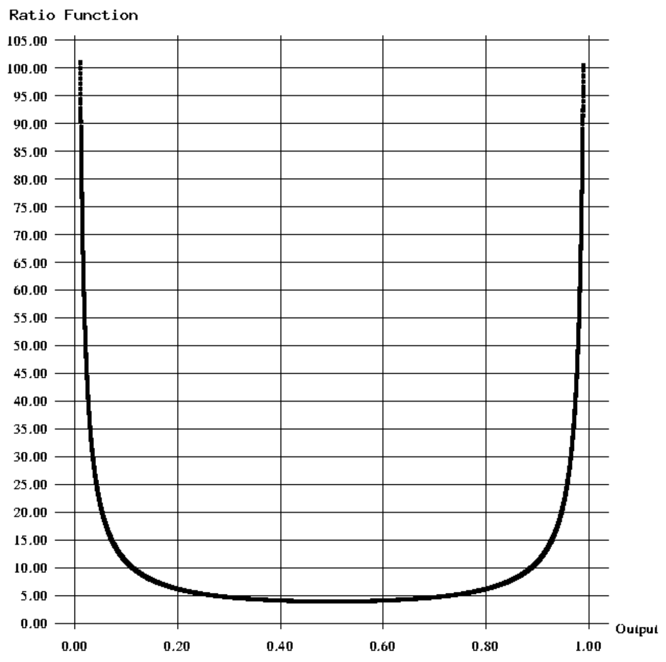

Figure 5: The ratio between NLR_o and SBP for a logistic activation function.

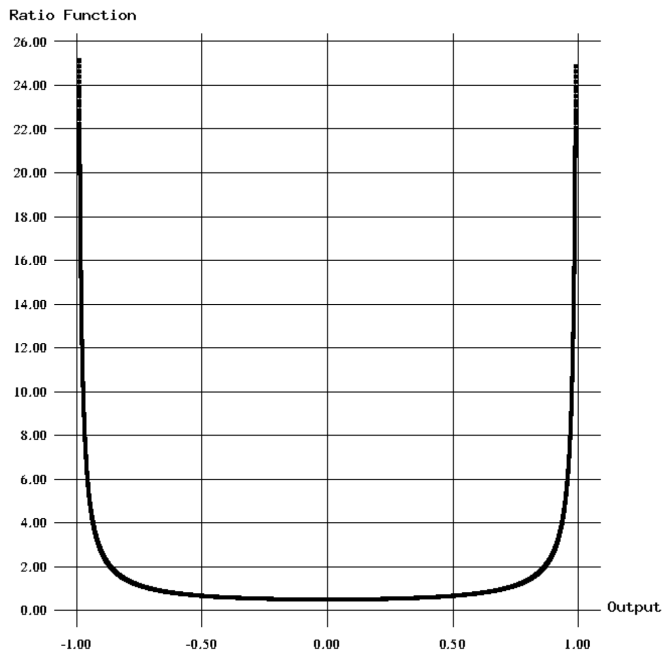

Figure 6: The ratio between NLR_o and SBP for a hyperbolic tangent activation function.

5.2 Stage Two: Learning Rules for Hidden Layers

5.2.1 GP Parameters

The fitness of each learning rule was computed using the TSS error E for both the XOR and encoder problems, using the same fitness function as described in Section 5.1.1. GP was run for 1000 generations with a population size of 1000 and a crossover probability of 0.9. After applying crossover, subtree mutation was applied with a probability of 0.01. We used the function set $\{+, -, \times\}$ and the terminal set $\{w_{ij}^l, o_{jp}^l, o_{ip}^{l+1}, \varepsilon_{ip}^{l+1}, 1, 0.5, 0.1\}$, with the "full" initialisation method (with initial maximum depths from 3 to 5) and tournament selection (with a tournament size of four).

5.2.2 Testing Problems

We considered the following six problems (the first four of which are the same as in Section 5.1.2): 1) the 'exclusive or' (XOR) problem and its more general form, the N-input parity problem; 2) the N-M-N encoder problem; 3) the character recognition problem with 7 inputs [15]; 4) the display problem; 5) the vowel recognition problem for independent speakers of the eleven steady state vowels of British English (for more information see [66]) where each training pattern has 10 input coefficients for each vowel and 4 outputs which represent the eleven different vowels; 6) the classification of sonar signals problem [67], where the task is to discriminate between the sonar signals (60 inputs) bounced off a metal cylinder and those bounced off a roughly cylindrical rock. The last two are hard, real-world benchmark problems.

5.2.3 Testing Conditions

We used the same network architectures as in Section 5.1.2 for the XOR problem, the encoder problem, the character recognition problem, and the display problem. For the vowel recognition problem we used a three-layer network consisting of 10 input, 8 hidden, and 4 output neurons. The neurons had a logistic activation function with output in the range of [-1,1]. For the classification of sonar signals we used a three-layer network consisting of 60 input, 12 hidden, and 1 output neurons. We used logistic activation functions with output in the range of [-0.3,0.3] (the same range as in [67]). Again, these parameters and network topologies were determined by experimenting with different configurations and selecting the ones on which the SBP algorithm worked best.

5.2.4 Results

In these experiments, GP was allowed to evolve learning rules for the hidden layers, while the learning rule for the output layer was NLR_o. Each GP run took six days of CPU time on average. For this reason, we were able to perform only 25 GP runs in this second stage. Four out of the twenty five end-of-run rules found by GP succeeded in learning the XOR problem. These are:

$$\mathrm{NLR}_1 = \Delta w_{ij}^l(s) = \eta[1.5 o_{ip}^{l+1}(1 - o_{ip}^{l+1})\varepsilon_{ip}^{l+1}(o_{jp}^l - 0.5) o_{jp}^l o_{jp}^l]$$

$$\mathrm{NLR}_2 = \Delta w_{ij}^l(s) = \eta[o_{jp}^l o_{ip}^{l+1} o_{ip}^{l+1}(1 - o_{ip}^{l+1})\varepsilon_{ip}^{l+1} + 0.1(o_{ip}^{l+1} - 0.5)o_{jp}^l]$$
$$\mathrm{NLR}_3 = \Delta w_{ij}^l(s) = \eta[o_{jp}^l o_{ip}^{l+1}(1 - o_{ip}^{l+1})\varepsilon_{ip}^{l+1} + 0.1(o_{ip}^{l+1} - 0.55)o_{jp}^l]$$
$$\mathrm{NLR}_4 = \Delta w_{ij}^l(s) = \eta[o_{jp}^l o_{ip}^{l+1}(1 - o_{ip}^{l+1})\varepsilon_{ip}^{l+1} + 0.1(o_{jp}^l - 0.55)o_{ip}^{l+1}]$$

We compared these rules against each other and against the learning rule found by Bengio and SBP. The results are shown in Table 1. The table reports the different learning rules with their learning rate η, and the minimum (Min.), maximum (Max.) and mean number of epochs which the algorithm needed to converge in 100 independent runs. The range of learning rates within which the algorithm converged with a probability of at least 50% (i.e. converged in at least 50 independent runs) is also shown. If a network had not converged within the maximum number of epochs, the run was declared unsuccessful.

The comparison shows that rules NLR_3 and NLR_4 are the best rules discovered in our runs. They perform very favourably with respect to SBP. Let us study the reasons for their reliability and efficiency.

Both NLR_3 and NLR_4 include two terms. The first one is the term $\eta o_{jp}^l o_{ip}^{l+1}(1 - o_{ip}^{l+1})\varepsilon_{ip}^{l+1}$ which is the SBP rule in the case of a logistic activation function (which has derivative $o_{ip}^{l+1}(1 - o_{ip}^{l+1})$). This makes sense since, in the encoder, display and character recognition problems, the logistic activation function is used. However, this makes less sense for the XOR problem, where the hyperbolic tangent activation function (which has derivative $\frac{1}{2}(1 - (o_{ip}^{l+1})^2)$) is used.

In our experiments we used two different activation function to attempt to obtain activation-function-independent learning rules. However, it is clear from Table 1, that we did not fully succeed in that: on the XOR problem both NLR_3 and NLR_4 are worse than SBP. Arguably the reason for this deficiency is the use of the incorrect form for the derivative of the activation function. Indeed, if we replace the term $o_{ip}^{l+1}(1 - o_{ip}^{l+1})$ with the term $\frac{1}{2}(1 - (o_{ip}^{l+1})^2)$ in NLR_3 and NLR_4, we obtain two rules, which we will call $\mathrm{NLR}_3(m)$ and $\mathrm{NLR}_4(m)$, that outperform SBP as shown in Table 2. This suggests that first term in each rule should be corrected to be $o_{jp}^l f'(net_{ip}^{l+1})\varepsilon_{ip}^{l+1}$, which behaves exactly like the SBP rule (ignoring the learning rate factor).

The second terms in NLR_3 and NLR_4 are $\eta 0.1(o_{ip}^{l+1} - 0.55)o_{jp}^l$ and $\eta 0.1(o_{jp}^l - 0.55)o_{ip}^{l+1}$, respectively. These two equations are similar to the Hebb-type learning rule used by Linsker [68]:

$$\mathrm{HB} = (o_{ip}^{l+1} - k_1)(o_{jp}^l - k_2) + k_3,$$

where k_1, k_2 and k_3 are constants. In NLR_3 and NLR_4 k_3 and either k_1 or k_2 are zero.

So, to summarise, our experiments suggest that a good learning rule for the hidden layers should have the form:

$$\mathrm{NLR}_h = \eta\mathrm{SBP} + \beta\mathrm{HB}$$

which, in the case of a logistic activation function, corresponds to:

$$\mathrm{NLR}_h = \eta o_{jp}^l o_{ip}^{l+1}(1 - o_{ip}^{l+1})\varepsilon_{ip}^{l+1} + \beta(o_{ip}^{l+1} - k_1)(o_{jp}^l - k_2),$$

while in the case of a hyperbolic tangent activation function corresponds to:

$$\mathrm{NLR}_h = \frac{1}{2}\eta o_{jp}^l(1 - (o_{ip}^{l+1})^2)\varepsilon_{ip}^{l+1} + \beta(o_{ip}^{l+1} - k_1)(o_{jp}^l - k_2).$$

Table 1: Performance of old and new learning rules on four different problems.

XOR							
		Learning Epochs				Successful	
Algorithm	η	Min.	Max.	Mean	Std. Dev.	Runs	Range
SBP	0.96	101	870	163.95	228.7	100	[0.13,1]
NLR_B	0.96	313	1000	580.1	N/A	73	[0.8,1]
$\mathrm{NLR}_o + \mathrm{NLR}_1$	0.96	44	1000	832.54	N/A	18	N/A
$\mathrm{NLR}_o + \mathrm{NLR}_2$	0.96	1000	1000	1000	N/A	0	N/A
$\mathrm{NLR}_o + \mathrm{NLR}_3$	0.96	16	1000	926.8	N/A	16	N/A
$\mathrm{NLR}_o + \mathrm{NLR}_4$	0.96	2	1000	688.26	N/A	17	N/A
Encoder							
		Learning Epochs				Successful	
Algorithm	η	Min.	Max.	Mean	Std. Dev.	Runs	Range
SBP	0.4	44	303	118.67	80	100	[0.05,0.5]
NLR_B	0.4	69	386	189.62	128.2	100	[0.2,0.6]
$\mathrm{NLR}_o + \mathrm{NLR}_1$	0.4	500	500	500	N/A	0	N/A
$\mathrm{NLR}_o + \mathrm{NLR}_2$	0.4	15	105	50.66	25.8	100	[0.03,1]
$\mathrm{NLR}_o + \mathrm{NLR}_3$	0.4	10	118	34.74	29.9	100	[0.01,1]
$\mathrm{NLR}_o + \mathrm{NLR}_4$	0.4	16	78	34.37	18.5	100	[0.01,1]
Character Recognition							
		Learning Epochs				Successful	
Algorithm	η	Min.	Max.	Mean	Std. Dev.	Runs	Range
SBP	0.3	179	358	226.54	49.8	100	[0.05,0.5]
NLR_B	0.3	500	500	500	N/A	0	N/A
$\mathrm{NLR}_o + \mathrm{NLR}_1$	0.3	43	94	61.26	14.6	100	[0.02,0.7]
$\mathrm{NLR}_o + \mathrm{NLR}_2$	0.3	71	331	149.35	90.4	100	[0.15,0.7]
$\mathrm{NLR}_o + \mathrm{NLR}_3$	0.3	54	160	83.21	28.3	100	[0.02,0.48]
$\mathrm{NLR}_o + \mathrm{NLR}_4$	0.3	40	108	64.31	18.9	100	[0.01,0.46]
Display							
		Learning Epochs				Successful	
Algorithm	η	Min.	Max.	Mean	Std. Dev.	Runs	Range
SBP	0.8	130	500	250.514	N/A	87	[0.3,1]
NLR_B	0.8	500	500	500	N/A	0	N/A
$\mathrm{NLR}_o + \mathrm{NLR}_1$	0.8	46	180	79.94	36.7	100	[0.03,1]
$\mathrm{NLR}_o + \mathrm{NLR}_2$	0.8	71	331	149.35	73.4	100	[0.05,0.95]
$\mathrm{NLR}_o + \mathrm{NLR}_3$	0.8	41	108	66.81	20.3	100	[0.05,1]
$\mathrm{NLR}_o + \mathrm{NLR}_4$	0.8	46	116	72.6	21.6	100	[0.06,1]

Table 2: Performance of $\mathrm{NLR}_o + \mathrm{NLR}_3(m)$ and $\mathrm{NLR}_o + \mathrm{NLR}_4(m)$ on the XOR problem.

XOR							
		Learning Epochs				Successful	
Algorithm	η	Min.	Max.	Mean	Std. Dev.	Runs	Range
$\mathrm{NLR}_o + \mathrm{NLR}_3(m)$	0.96	49	1000	161.68	N/A	89	[0.12,1]
$\mathrm{NLR}_o + \mathrm{NLR}_4(m)$	0.96	58	88	70.95	7.3	100	[0.07,1]

The complete learning rule is NLR_o for the output layer and NLR_h for the hidden layers. Table 3 shows the results of SBP and $\mathrm{NLR}_o + \mathrm{NLR}_h$ for the vowel and sonar problems. In the tests, we found that $\beta = 0.01, k_1 = 0.5$ and $k_2 = 0$ give the best results on the vowel problem, while for the sonar problem, the best parameters are $\beta = 0.1, k_1 = 0.5$ and $k_2 = 0$.

To test the generality of $\mathrm{NLR}_o + \mathrm{NLR}_h$, additional experiments were carried out on the vowel recognition problem and the classification of sonar signals where we changed the architecture of the neural networks to be trained. Table 4 shows a comparisons between SBP and NLR_o+NLR_h when the number of neurons in the hidden layer is changed. In the comparison, $\mathrm{NLR}_o + \mathrm{NLR}_h$ is the clear winner. This can only be attributed to the presence of a Hebbian term and to its feature-extraction properties. Table 5 reports the performance of SBP and of the NLR_o+NLR_h learning rule in networks with different architectures on the XOR problem. For each rule the table shows the best learning rate η, the minimum (Min.), maximum (Max.) and mean number of epochs which the algorithm needs to converge in 100 independent training runs, and the range of learning rates within which the algorithm converges with probability at least 95% (i.e., in 95 out of 100 independent runs). Again, the advantages of the feature extracting Hebbian term in the NLR_h are clear. When we move from networks with one hidden layer to networks with a second hidden layer, the successful-convergence range for SBP shrinks and the mean number of epochs increases. On the contrary, the performance of $\mathrm{NLR}_o + \mathrm{NLR}_h$ improves and its learning rate range either expands or stays the same.

6 Discussion

The most interesting result of our experiments is that GP has discovered a learning rule in which the weaknesses of a supervised learning rule (SBP) are removed by combining it with an unsupervised learning rule (HB).

An explanation for the success of this combination can be as follows. The rule for the output layer,

$$\mathrm{NLR}_o = \Delta w_{ij}^l(s) = \eta o_{jp}^l \varepsilon_{ip}^{l+1},$$

is identical to the SBP rule except that the factor representing the derivative of the activation function, $f'(net_{ip}^{l+1})$, is missing. As discussed previously, this is an improvement since it leads to higher convergence speed when the neurons are near to saturation.

Table 3: Performance of $\mathrm{NLR}_o + \mathrm{NLR}_h$ and SBP on two hard real-life problems.

Vowel						
		Learning Epochs				Successful
Algorithm	η	Min.	Max.	Mean	Std. Dev.	Runs
SBP	0.08	124	273	256.9	43.3	100
$\mathrm{NLR}_o + \mathrm{NLR}_h$	0.08	120	200	165.8	26.1	100
Sonar						
		Learning Epochs				Successful
Algorithm	η	Min.	Max.	Mean	Std. Dev.	Runs
SBP	0.1	373	420	396.7	13.3	100
$\mathrm{NLR}_o + \mathrm{NLR}_h$	0.1	117	167	138.1	27.24	100

Table 4: Performance of $\mathrm{NLR}_o + \mathrm{NLR}_h$ and SBP as a function of the number of neurons in the hidden layer N.

Vowel			
Algorithm	N	% of success	Std. Dev.
SBP	8	32	2.1
SBP	12	39	1.8
SBP	16	45	1.6
SBP	24	49	1.3
SBP	32	50	1.1
$\mathrm{NLR}_o + \mathrm{NLR}_h$	8	48	1.5
$\mathrm{NLR}_o + \mathrm{NLR}_h$	12	50	1.2
$\mathrm{NLR}_o + \mathrm{NLR}_h$	16	58	1.1
$\mathrm{NLR}_o + \mathrm{NLR}_h$	24	63	1.0
$\mathrm{NLR}_o + \mathrm{NLR}_h$	32	65	1.0
Sonar			
Algorithm	N	% of success	Std. Dev.
SBP	8	70	1.4
SBP	12	78	1.1
SBP	16	80	1.0
SBP	24	81	0.8
$\mathrm{NLR}_o + \mathrm{NLR}_h$	8	74	2.1
$\mathrm{NLR}_o + \mathrm{NLR}_h$	12	84	1.5
$\mathrm{NLR}_o + \mathrm{NLR}_h$	16	88	1.3
$\mathrm{NLR}_o + \mathrm{NLR}_h$	24	90	1.0

Table 5: Performance of NLR_o with NLR_h and SBP in networks with different architectures on the XOR problem.

2-2-2-1							
		Learning Epochs				Successful	
Algorithm	η	Min	Max	Mean	Std. Dev.	Runs	Range
SBP	1	97	1000	231.55	258	100	[0.55-1]
$\text{NLR}_o + \text{NLR}_h$	0.95	21	126	34.48	30	100	[0.1-1]
2-2-3-1							
		Learning Epochs				Successful	
Algorithm	η	Min	Max	Mean	Std. Dev.	Runs	Range
SBP	0.95	86	602	175.61	146.2	100	[0.3-1]
$\text{NLR}_o + \text{NLR}_h$	1	19	70	26.34	14.5	100	[0.05-1]
2-3-3-1							
		Learning Epochs				Successful	
Algorithm	η	Min	Max	Mean	Std. Dev.	Runs	Range
SBP	1	54	315	126.52	74.2	100	[0.3-1]
$\text{NLR}_o + \text{NLR}_h$	0.9	17	62	26.51	12.6	100	[0.05-1]
2-3-1							
		Learning Epochs				Successful	
Algorithm	η	Min	Max	Mean	Std. Dev.	Runs	Range
SBP	1	68	157	104.35	29.1	100	[0.15-1]
$\text{NLR}_o + \text{NLR}_h$	1	36	61	47.23	6.8	100	[0.1-1]

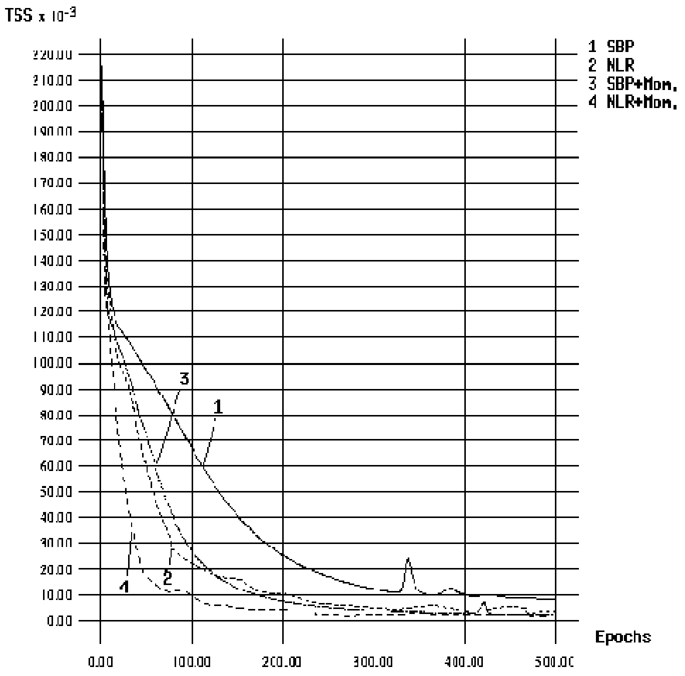

Figure 7: Plots of the TSS error for SBP, NLR, SBP with Momentum, and NLR with Momentum on the character recognition problem.

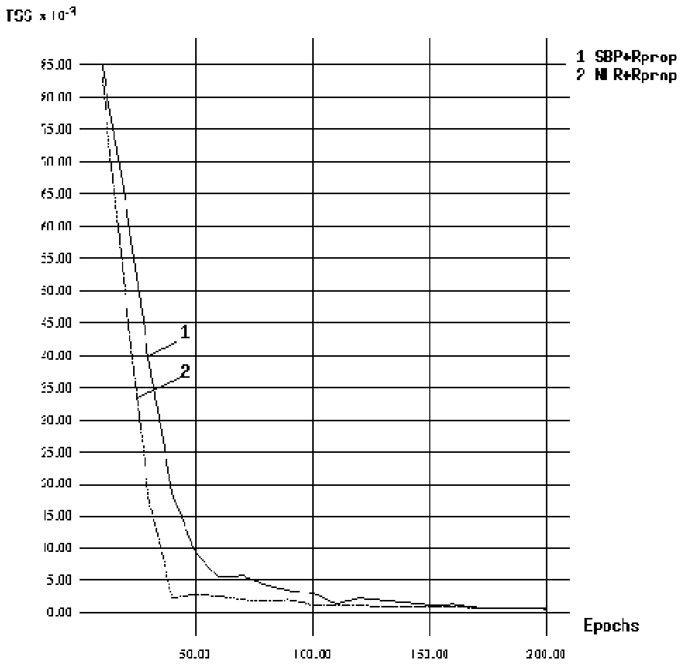

Figure 8: Plots of the TSS error for SBP with Rprop and NLR with Rprop on the character recognition problem.

The learning rule for the hidden layers, NLR_h, can be written as follows:

$$\Delta w^l_{ij}(s) \;=\; \frac{1}{2}\eta o^l_{jp}[f'(net^{l+1}_{ip})\varepsilon^{l+1}_{ip} + \frac{\beta}{\eta}(o^{l+1}_{ip} - k_1)]$$

where $k_1 \approx 0.5$. For the logistic activation function its operation can be interpreted as follows:

1. If neuron j is active (i.e., contributed to the error) then
   (a) If neuron i is not fully committed to be either active or inactive (o^{l+1}_{ip} is near 0.5) then the weight is updated using the SBP rule for the hidden layers (in these conditions $o^{l+1}_{ip} - k_1 \approx 0$). This makes sure correct error-reducing decisions are taken during the early stages of training.
   (b) Otherwise, if neuron i appears to be committed towards positive saturation (o^{l+1}_{ip} greater than 0.5) then the term $o^{l+1}_{ip} - k_1$ helps it to became even more saturated (in these conditions the influence of $f'(net^{l+1}_{ip})\varepsilon^{l+1}_{ip}$ can be neglected unless the error is much smaller than 0). This makes sure that neurons tend saturate quickly even if f' is very small.
   (c) Finally, if o^{l+1}_{ip} is committed towards negative saturation (o^{l+1}_{ip} less than 0.5) then the term $o^{l+1}_{ip} - k_1$ helps the neuron to saturate towards 0 (unless there is a clear indication to do the opposite). Again this speeds up convergence even if f' is very small.
2. If neuron j is inactive, do nothing.

The combination of NLR_o and NLR_h, which we will term NLR for brevity, was very efficient and stable in the problems considered. To see if this rule could be improved even more by using some of the SBP speed-up techniques, in another set of tests its convergence behaviour was compared with SBP, with and without the Momentum and Rprop speed-up algorithms. Figure 7 shows the TSS error for SBP and NLR, with and without Momentum, in a typical run on the character recognition problem. In this run NLR achieves its target output at the same epoch as SBP with Momentum, while NLR with Momentum converges much more quickly than the other algorithms. Figure 8 shows another typical run in which NLR with Rprop outperforms SBP with Rprop in the character recognition problem. In these runs all algorithms used the same initial random weights. To reiterate, all runs are typical. Similar performance improvements were obtained with different problems.

7 Conclusions

GP discovered, for the output layer, a useful way of using the Delta learning rule (originally developed for single-layer neural networks) to speed up learning. This rule has been consistently faster and more stable than the SBP learning rule on all the sample problems considered. Then, using this rule to train the output layer, GP discovered new learning rules for the hidden layer. In particular, GP discovered a useful way of using the generalised Delta rule and the Hebbian learning rule together. Combined, these two learning rules have performed much better (i.e. they have been faster, more stable, and have shown greater feature extracting capabilities) than the SBP learning rule on all the problems considered, with and without different speed up algorithms.

This study indicates that there are supervised learning algorithms that perform better and are more stable than the SBP learning rule and that GP can discover them.

References

[1] D. Rumelhart, G. Hinton, and R. Williams, *Parallel Distributed Processing*. MIT Press, Cambridge, MA, 1986.

[2] S. Haykin, *Neural Networks: A Comprehensive Foundation*. New York 10022: IEEE Society Press, Macmillan College Publishing, 1994.

[3] X. Yao and Y. Liu, "Evolving neural networks for medical applications," in *Proceedings of 1995 Australia-Korea Joint Workshop on Evolutionary Computation*, (KAIST, Taejon, Korea), pp. 1–16, September 1995.

[4] J. Taylor, *The Promise of Neural Networks*. London: Springer-Verlag, 1993.

[5] X. Yao, "A review of evolutionary artificial neural networks," tech. rep., Commonwealth Scientific and Industrial Research Organization, Division of Building, Construction and Engineering, PO Box 56, Highett, Victoria 3190, Australia, 1992.

[6] X. Yao, "Evolutionary artificial neural networks," in *Encyclopedia of Computer Science and Technology* (A. K. et al., ed.), vol. 33, pp. 137–170, Marcel Decker Inc., New York, NY 10016, USA, 1995.

[7] S. Fahlman, "An empirical study of learning speed in back propagation networks," Tech. Rep. CMU-CS-88-162, Carnegie Mellon University, Pittsburgh, PA, 1988.

[8] M. Hagan, H. Demuth, and M. Beale, *Neural Network Design*. PWS Publishing Company, Boston, MA 02116, 1996.

[9] M. Riedmiller, "Advanced supervised learning in multi-layer perceptrons from backpropagation to adaptive learning algorithms," *Computer Standards and Interfaces Special Issue on Neural Networks*, vol. 16, no. 3, pp. 265–275, 1994.

[10] M. Riedmiller and H. Braun, "A direct method for faster backpropagation learning: The RPROP Algorithm," *In IEEE International Conference on Neural Networks 1993 (ICNN93), San Francisco*, pp. 586–591, 1993.

[11] D. Sarkar, "Methods to speed up error back propagation learning algorithm," *ACM Computing Surveys*, vol. 27, no. 4, pp. 519–542, 1995.

[12] F. Silva and L. Almeida, “Acceleration techniques for the back-propagation algorithm parallel networks,” *Lecture Notes in Computer Science*, vol. 412, pp. 110–119, 1990.

[13] F. Silva and L. Almeida, “Speeding up back-propagation,” in *Advanced Neural Computers* (R. Eckmiller, ed.), (Amsterdam), pp. 151–158, Elsevier Science Publishers, 1990.

[14] P. Baldi, “Gradient learning algorithm overview: A general dynamical systems perspective,” *IEEE Transactions on Neural Networks*, vol. 6, pp. 182–195, January 1995.

[15] S. Bengio, Y. Bengio, and J. Cloutier, “Use of genetic programming for the search of a learning rule for neural networks,” in *Proceedings of the First Conference on Evolutionary Computation,IEEE World Congress on Computational Intelligence*, (Orlando, Florida, USA), pp. 324–327, 1994.

[16] J. Kruschke and J. Movellan, “Fast learning algorithms for neural networks,” *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, vol. 39, no. 7, pp. 453–473, 1992.

[17] M. Moller, “A scaled conjugate gradient algorithm for fast supervised learning,” *Neural Networks*, vol. 6, pp. 525–533, 1993.

[18] A. Parlos, B. Fermandez, A. Atiya, J. Muthusami, and W. Tsai, “An accelerated learning algorithm for multilayer perceptron networks,” *IEEE Transactions on Neural Networks*, vol. 5, no. 3, pp. 493–497, 1994.

[19] D. Karras and S. Perantonis, “An efficient constrained training algorithm for feedforward networks,” *IEEE Transactions on Neural Networks*, vol. 6, pp. 1420–1434, Nov 1995.

[20] S. Perantonis and D. Karras, “An efficient constrained learning algorithm with momentum acceleration,” *Neural Networks*, vol. 8, pp. 237–249, 1995.

[21] J. Holland, *Adaptation in Natural and Artificial Systems*. Cambridge, Massachusetts: The MIT Press, 2nd ed., 1992.

[22] D. Goldberg, *Genetic Algorithm in Search, Optimisation and Machine Learning*. Reading, Massachussets: Addison-Wesley, 1989.

[23] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[24] W. Spears, K. D. Jong, T. Bäck, D. Fogel, and H. de Garis, “An overview of evolutionary computation,” in *Proceedings of the European Conference on Machine Learning (ECML-93)* (P. B. Brazdil, ed.), vol. 667 of *LNAI*, (Vienna, Austria), pp. 442–459, Springer Verlag, Apr. 1993.

[25] X. Yao, “Evolving artificial neural networks,” *Proceedings of the IEEE*, vol. 87, pp. 1423–1447, September 1999.

[26] D. Whitley and T. Hanson, “Optimizing neural networks using faster, more accurate genetic search,” in *Third International Conference on Genetic Algorithms* (J. Schaffer, ed.), (George Mason University), pp. 391–396, Morgan Kaufmann, 1989.

[27] J. Pujol and R. Poli, “Efficient evolution of asymmetric recurrent neural networks using a PDGP-inspired two-dimensional representation,” in *the First European Workshop on Genetic Programming(EUROGP’98),* vol. 1391, (Paris, Springer Lecture Notes in Computer Science), pp. 130–141, 1998.

[28] J. Pujol and R. Poli, “Evolving neural networks using a dual representation with a combined crossover operator,” in *the IEEE International Conference on Evolutionary Computation (ICEC’98),* (Anchorage, Alaska), pp. 416–421, 1998.

[29] J. Pujol and R. Poli, “Evolving the topology and the weights of neural networks using a dual representation,” *Special Issue on Evolutionary Learning of the Applied Intelligence Journal,* vol. 8, no. 1, pp. 73–84, 1998.

[30] D. Chalmers, “The evolution of learning: An experiment in genetic connectionsm,” in *Proceedings of the 1990 Connectionist Models Summer School* (E. Touretsky, ed.), (San Mateo, CA), pp. 81–90, Morgan-Kaufman, 1990.

[31] X.-H. Y. G.-A. Chen, “Efficient backpropagation learning using optimal learning rate and momentum,” *Neural Networks*, vol. 10, no. 3, pp. 517–527, 1997.

[32] H. Eaton and T. Oliver, “Improving the convergence of the back propagation algorithm,” *Neural Networks*, vol. 5, pp. 283–288, 1992.

[33] R. Jacobs, “Increased rates of convergence through learning rate adaptation,” *Neural Networks,* vol. 1, no. 1, pp. 295–307, 1989.

[34] T. Tollenaere, “Super SUB: Fast adaptive back propagation with good scaling properties,” *Neural Networks*, vol. 3, no. 5, pp. 561–573, 1990.

[35] T. Vogl, J. Mangis, A. Rigler, W. Zink, and D. Alkon, “Accelerating the convergence of the backpropagation method,” *Biological Cybernetics,* vol. 59, pp. 257–263, September 1988.

[36] M. Weir, “A method for self determination of adaptive learning rate in back propagation,” *Neural Networks*, vol. 4, pp. (371–379), 1991.

[37] V. Cherkassky and R. Shepherd, “Regularization effect of weight initialisation in back propagation networks,” in *1998 IEEE International Joint Conference of Neural Networks (IJCNN’98),* (Anchorage, Alaska), pp. 2258–2261, IEEE, May 1998.

[38] T. Denoeux and R. Lengelle, “Initialising back propagation networks using prototypes,” *Neural Networks*, vol. 6, no. 3, pp. 351–363, 1993.

[39] L. Wessels and E. Barnard, “Avoiding false local minima by proper initialisation of connections,” *IEEE Transactions on Neural Networks*, vol. 3, no. 6, pp. 899–905, 1992.

[40] J. Kruschke and J. Movellan, “Benefits of gain: Speeded learning and minimal hidden layers in back propagation networks,” *IEEE Transactions on Systems, Man and Cybernetics.*, vol. 21, no. 1, pp. 273–280, 1991.

[41] R. Anand, K. Mehrotra, C. Mohan, and S. Ranka, “An improved algorithm for neural network classification of imbalanced training sets,” *IEEE Transactions on Neural Networks*, vol. 4, no. 6, pp. 962–969, 1993.

[42] T. M. Kwon and H. Cheng, “Contrast enhancement for backpropagation,” *IEEE Transactions on Neural Networks*, vol. 7, no. 1, pp. 515–524, 1996.

[43] R. Vitthal, P. Sunthar, Rao, and C. Durgaprasada, “The generalized proportional-integral-derivative (PID) gradient descent back propagation algorithm,” *Neural Networks*, vol. 8, no. 4, pp. 563–569, 1995.

[44] B.-T. Zhang, “Accelerated learning by active example selection,” *International Journal of Neural Systems*, vol. 5, no. 1, pp. 67–75, 1994.

[45] R. Ahmad and F. Salam, “Error back propagation learning using the polynomial energy function,” in *Proceedings of the International Joint Conference on Neural Networks, Iizuka, Japan*, 1992.

[46] M. Holt and S. Semnani, “Convergence of back-propagation in neural networks using a log-likelihood cost function,” *Electronics Letters*, vol. 26, no. 23, pp. 1964–1965, 1990.

[47] A. Ooyen and B. Nienhuis, “Improving the convergence of the back propagation algorithm,” *Neural Networks*, vol. 5, pp. 465–471, 1992.

[48] M. Joost and W. Schiffmann, “Speeding up backpropagation algorithms by using cross-entropy combined with pattern normalization,” *International Journal of Uncertainity, Fuzziness and Knowledge–Based Systems (IJUFKS)*, vol. 6, no. 2, pp. 117–126, 1998.

[49] S. Solla, E. Levin, and M. Fleisher, “Accelerated learning in layered neural networks,” *Complex System*, vol. 2, pp. 625–640, 1988.

[50] C. Schittenkopf, G. Deco, and W. Brauer, “Two strategies to avoid overfitting in feedforward neural networks,” *Neural Networks*, vol. 10, no. 3, pp. 505–516, 1997.

[51] Y. Hirose, K. Yamashit, and S. Hijiya, “Back propagation algorithm which varies the number of hidden units,” *Neural Networks*, vol. 4, no. 1, pp. 61–66, 1991.

[52] W. Schiffmann, M. Joost, and R. Werner, “Application of genetic algorithms to the construction of topologies for multilayer perceptrons,” in *Proceedings of International Conference on Neural Networks and Genetic Algorithms (ICNNGA)*, (University of Innsbruck, Austria), pp. 675–682, 1993.

[53] M. Gori and A. Tesi, “On the problem of local minima in backpropagation,” *IEEE Transactions on PAMI*, vol. 14, no. 1, pp. 76–86, 1992.

[54] H. Kitano, “Neurogenetic learning: an integrated method of designing and training neural networks using genetic algorithms,” *Physica D*, vol. 75, pp. 225–238, 1994.

[55] R. Mitchell, J. Bishop, and W. Low, “Using a genetic algorithm to find the rules of a neural network,” in *Artificial Neural Nets and Genetic Algorithms* (R. Albrecht, C. Reeves, and N. Steele, eds.), pp. 664–669, Springer-Verlag, 1993.

[56] J. Baxter, “The evolution of learning algorithms for artificial neural networks,” in *Complex Systems* (D. Green and T. Bossomaier, eds.), pp. 313–326, IOS Press, Amsterdam, 1992.

[57] S. Bengio, Y. Bengio, and J. Cloutier, “Generalisation of a parametric learning rule,” in *Proceedings of the International Conference on Artificial Neural Networks*, (Amsterdam, Nederlands), pp. 502–502, Springer Verlag, 1993.

[58] A. Dasdan and K. Oflazer, “Genetic synthesis of unsupervised learning algorithms,” Tech. Rep. BU-CEIS-9306, Department of Computer Engineering and Information Science, Bilkent University, Ankara, Turkey, 1993.

[59] R. Belew, J. McInerney, and N. Schraudolph, “Evolving networks: Using the genetic algorithm with connectionism learning,” in *Proceeding Second Artificial Life Conference*, pp. 511–547, Addison-Wesley, 1991.

[60] S. Harp, T. Samad, and A. Guha, “Designing application-specific neural networks using the genetic algorithm,” in *Advances in Neural Information Processing Systems II* (D. Touretzky, ed.), pp. 447–454, San Mateo, California: Morgan Kaufmann, 1990.

[61] T. Kohonen, *Self-organization and Associative Memory*. Springer-Verlag, New York, 2 ed., 1988.

[62] J. Koza, *Genetic Programming II: Automatic discovery of reusable programs*. Cambridge, Massachussetts: The MIT Press, 1994.

[63] W. Banzhaf, P. Nordin, R. Keller, and F. Francone, *Genetic Programming An Introduction: On the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann Publishers, Inc. San Francisco, California, 1998.

[64] A. Radi and R. Poli, “Genetic programming can discover fast and general learning rules for neural networks,” in *Third Annual Genetic Programming Conference (GP’98), Madison, Wisconsin* (J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. Fogel, M. Garzon, D. Goldberg, H. Iba, and R. Riolo, eds.), pp. 314–323, Morgan Kaufmann, July 1998.

[65] A. Radi and R. Poli, "Discovery of optimal backpropagation learning rules using genetic programming," in *IEEE International Conference on Evolutionary Computation*, (Anchorage, Alaska), pp. 371–375, IEEE Press, May 5–9 1998.

[66] D. Deterding, *Speaker Normalisation for Automatic Speech Recognition*. PhD thesis, University of Cambridge, 1989.

[67] R. Gorman and T. Sejnowski, "Analysis of hidden units in a layered network trained to classify sonar targets," *Neural Networks*, vol. 1, pp. 75–89, 1988.

[68] R. Linsker, "From basic network principles to neural architecture: Emergence of spatial-opponent cells.," in *Proceedings of the National Acedemy of Science USA*, vol. 83, pp. 7508–7512, 1986.