A PULSE-BASED REINFORCEMENT ALGORITHM FOR LEARNING CONTINUOUS FUNCTIONS

D Gorse
Department of Computer Science
University College, Gower Street, London WC1E 6BT, UK

J G Taylor
Department of Mathematics
T G Clarkson
Department of Electrical and Electronic Engineering
King's College, Strand, London WC2R 2LS, UK

An algorithm is presented which allows continuous functions to be learned by a neural network using spike-based reinforcement learning. Both the mean and the variance of the weights are changed during training; the latter is accomplished by manipulating the lengths of the spike trains used to represent real-valued quantities. The method is here applied to the probabilistic RAM (pRAM) model, but it may be adapted for use with any pulse-based stochastic model in which individual weights behave as random variables.

1. Introduction

Reinforcement learning [1] has many attractions for neural networks; it is more widely applicable than supervised techniques (since target values are not needed for any of the processing nodes), it is biologically plausible, and it is less prone to being trapped in local minima than classical methods like error backpropagation. However standard reinforcement techniques have a major disadvantage in that they were developed to apply to systems with only a finite number of behavioural responses, which makes them unsuitable for learning real-valued functions. In the case of the A_{RP} neural model [1] there are two such responses: output 0 (off, not firing) and output 1 (on, firing). The A_{RP} reinforcement learning rule has the inevitable effect of driving neurons into saturation, changing weight parameters so that neurons will eventually output 0 or 1 with certainty for almost any input pattern. This behaviour is adequate for some control applications, such as the "bang-bang" approach to controlling an inverted pendulum [2], and for classification problems, since the user can define a different binary output code for each pattern class, but not in more general learning situations (such as process control, time series prediction, robot arm movement and so on) for which the desired outputs are continuous functions of the inputs.

The algorithm to be presented here makes use of the noise associated with the representation of real-valued quantities by finite spike trains in order to provide the necessary variation about the mean for weight parameters during training. If a variable x in the interval [0,1] is represented by a spike train of length R, its measured value

$$\hat{x}(R) = \frac{1}{R}\sum_{r=1}^{R} a(r)$$

(where the a(r) are spikes $\in \{0,1\}$, with Prob(a=1) = x) has a variance

$$\sigma^2(x,\ R) = \frac{x(1-x)}{R} \tag{1}$$

As training progresses, and performance improves, the spike trains representing the weights are lengthened so as to decrease the amount of stochastic variation in their measured values. The advantage of using spike trains in this way, rather than choosing values from a distribution function such as in [3], is that one may then hope for a simple implementation in digital hardware.

2. The pRAM Model

An N-input pRAM [4] has 2^N memory locations, indexed by the N-bit binary vector $\underline{u} = (u_1, u_2, ...u_N)$. A binary signal $\underline{i} = (i_1, i_2, ...i_N)$ on the input lines will access just one of these locations, that one for which $\underline{u} = \underline{i}$. The variable stored at this location, $\alpha_{\underline{i}}$, gives the probability that a spike value a=1 will be produced on the output line, given an input $\underline{i}$. Thus in this binary input case the mean value of the output is given by $\alpha_{\underline{i}}$. If there is a real-valued signal $\underline{x} = (x_1, x_2, ...x_N)$ on the input lines each x_j is represented by a stream of pulses i_j in which $\mathrm{Prob}(i_j = 1) = x_j$. Over r=1..R time steps a spread of locations (as in general each $\underline{i}(r)$ will be different) will be accessed and contribute stochastically a spike value a=1 or a=0 to the output stream. In this case the mean value of the output is

$$y = \sum_{\underline{u}} \alpha_{\underline{u}} \prod_{j=1}^{N} [u_j x_j + (1-u_j)(1-x_j)] \equiv \sum_{\underline{u}} \alpha_{\underline{u}} X_{\underline{u}}(\underline{x}) \tag{2}$$

where the variable $X_{\underline{u}}$ is the probability location $\underline{u}$ is accessed.

When hardware is used [5] the output of a pRAM is not computed from (2) directly, but as an approximation by averaging over the bits in the output stream:

$$\hat{y}(R) = \frac{1}{R}\sum_{r=1}^{R} a(r) \tag{3}$$

As $R \to \infty$, $\hat{y}(R) \to y$, the mean value of the output. In practice a value of $R = R_{max}$ in the order of hundreds of spikes gives a suitably close approximation to the mean.

The pRAM is not a 'weightless' model; the 2^N variables $\alpha_{\underline{u}}$ are the weights of the system. It is not a look-up table, since these variables represent probabilities continuous in the range [0,1] and a pRAM output in general contains spikes that have been contributed by a wide range of locations. The way that noise is introduced into the pRAM model, as an independent random variable for each input channel, is closer to biological reality than models which inject noise only at the threshold; one of the ideas underpinning the pRAM is the biologically realistic synaptically noisy neuron model presented by Taylor in 1972 [6].

3. Reinforcement Learning for pRAM Networks

A binary output version of this algorithm has been described in detail elsewhere [7]. It is a non-linear extension of Barto's A_{RP} rule [1] which can be implemented in self-contained hardware [5]. The learning system interacts with the outside world only through the global reward and penalty signals r and p. The update to an addressed memory location $\underline{u}$ in an N-input pRAM is given by

$$\Delta\alpha_{\underline{u}} = \rho[\,(a - \alpha_{\underline{u}})r + \lambda(1 - a - \alpha_{\underline{u}})p\,] \tag{4}$$

The other $2^N - 1$ non-addressed locations do not have their weight parameters updated. The variable a is the spike output of the pRAM on the addressing of location $\underline{u}$; r and p have values in [0,1] but may themselves be carried by spike trains. The effect of the rule (4) is to move the addressed memory content $\alpha_{\underline{u}}$ in the direction of a 'successful' output if a reward signal is received, to move $\alpha_{\underline{u}}$ away from that output value (toward 1-a) if a penalty is received. It is clear that application of (4) will eventually lead to addressed locations containing binary values, since the values toward which the $\alpha_{\underline{u}}$ are shifted (a or 1-a) are themselves binary. This is the output saturation problem [3] in the context of the pRAM model.

In order for outputs not to become saturated, the system needs to be able to experiment with values for the $\alpha_{\underline{u}}$ which are not binary, but which may be seen to lead to better or worse (encouraged or discouraged) performance. Thus any continuous output reinforcement rule will have the general form

$$\Delta\alpha_{\underline{u}} = \rho[\,(\hat{\alpha}_{\underline{u}} - \alpha_{\underline{u}})r + \lambda(1 - \hat{\alpha}_{\underline{u}} - \alpha_{\underline{u}})p\,]\hat{X}_{\underline{u}} \tag{5}$$

In this rule the variable $\hat{\alpha}_{\underline{u}}$ is the result of some stochastic experiment performed by the system; it is a 'trial value' for $\alpha_{\underline{u}}$, the effect of which is then assessed by the environment, which delivers appropriate reward and penalty signals. $\hat{X}_{\underline{u}} \in [0,1]$, which scales the weight change, reflects the relative contribution of address u during this 'experiment'; it is a random variable which approximates $X_{\underline{u}}$, the address access probability defined in (2).

The variables $\hat{\alpha}_{\underline{u}}$ and $\hat{X}_{\underline{u}}$ can be defined for a pulse-based simulation (or a hardware implementation) by using counters $S_{\underline{u}}$, $A_{\underline{u}}$ to monitor the contributions of individual locations to the R-bit output stream of (3). It is the fact that R is finite (and possibly small) which gives rise to the 'stochastic experiment' needed by a reinforcement training technique, since by (1) real numbers represented by an R-bit pulse stream have a variance inversely proportional to R. Let $S_{\underline{u}}$ (an integer from 0 to R) be the total number of times address u is accessed whilst the output $\hat{y}(R)$ is being accumulated according to (3), and let $A_{\underline{u}}$ (also an integer from 0 to R) be the number of 1s output by location $\underline{u}$ when accessed during this process. It is evident that

$$\sum_{\underline{u}} S_{\underline{u}} = R, \quad \frac{1}{R}\sum_{\underline{u}} A_{\underline{u}} = \hat{y}(R)$$

$\hat{X}_{\underline{u}}$, the proportion of times address $\underline{u}$ is accessed, is then defined by

$$\hat{X}_{\underline{u}} = \frac{S_{\underline{u}}}{R}$$

and the 'trial value' $\hat{\alpha}_{\underline{u}}$ by

$$\hat{\alpha}_{\underline{u}} = \frac{A_{\underline{u}}}{R}, \quad S_{\underline{u}} \neq 0$$

In order to avoid the need for special action when $S_{\underline{u}} = 0$, it is convenient both for simulation and hardware realisation to write the learning rule (5) in the alternative form

$$\Delta\alpha_{\underline{u}} = \frac{\rho}{R}[\,(A_{\underline{u}} - \alpha_{\underline{u}}S_{\underline{u}})r - \lambda(A_{\underline{u}} - (1 - (1-\alpha_{\underline{u}})S_{\underline{u}})p\,] \tag{6}$$

The rule (6) relates to the adaptation of the mean values of the weight parameters; a further rule is required which gives the change in their variance (determined by, and inversely proportional to, the spike train length R) as a function of the performance of the system. This further rule requires in general that R increases when performance improves and decreases when it worsens.

In order to proceed further we will need to define 'performance' more precisely. Although the method described here can be applied in any situation in which it is possible to define a cost or utility function for the purpose of reinforcement, we will from this point on restrict the discussion to the learning of a function from $[0,1]^N$ to $[0,1]^M$, from a set of P training patterns $\underline{x}_p \in [0,1]^N$ and their corresponding desired outputs $\underline{d}_p \in [0,1]^M$ (to be learned either by a set of M N-pRAMs or by a multilayer structure, as appropriate). We will use an 'on-line' or pattern-by-pattern updating strategy. A broad overview of the function learning algorithm is as follows:

while mean error over training set > tolerance

1. Pick a pattern p from the training set.
2. Make an accurate assessment of the performance of the network in classifying this pattern.
3. Use this measure to update R for the next 'stochastic experiment'.
4. Make an assessment of the performance of the network during this stochastic procedure.
5. Use this new measure to calculate reward r and penalty p.
6. Update weights using r and p.
7. Go to step 1.

In this case a suitable performance measure is the Euclidean distance D between the actual and desired outputs. Depending on the value of R used in the output computation, this measure may itself show significant stochastic variation. It is best to measure the error as accurately as possible when R is to be updated (step 2.). We define

$$E_p = D(\underline{d}_p, \hat{\underline{y}}_p(R_{max})) \approx D(\underline{d}_p, \underline{y}_p) \tag{7}$$

where R_{max} is a spike train length sufficiently long that the output $\hat{\underline{y}}_p(R_{max})$ is very close to that which would be obtained by computation using the mean values of the variables (inputs, weights, outputs) of the system. R may then be calculated, for example, from

$$R(E_p) = \mathit{ceiling}\left(\frac{c_R}{E_p^{k_R}}\right) \tag{8}$$

where the 'ceiling' function delivers the smallest integer greater than or equal to its argument, and c_R, k_R are positive real numbers.

In calculating the reward and penalty, the performance measure D will be a random variable whose degree of variation depends on the current (updated by step 3., equation (8)) value of R:

$$\hat{E}_p = D(\underline{d}_p, \hat{\underline{y}}_p(R)) \tag{9}$$

It is clear that (9), not (7), should be used in the calculation of r and p, since reinforcement should reflect the performance level achieved when $R < R_{max}$ by using 'trial values' $\underline{\alpha}_u$ of the weight parameters.

The reward and penalty signals can be given in terms of $\hat{E}_p$ by

$$r = 1 - \hat{E}_p^{k_r}, \quad p = \hat{E}_p^{k_p} \tag{10}$$

where k_r, k_p are positive real numbers. It can be shown (see Appendix) that the above algorithm converges provided that

$$k_R + k_r \geq 2, \quad k_p \geq 3, \quad \lambda > 0 \tag{11}$$

These conditions are easy to satisfy and convergence has been observed over a wide region of parameter space.

A construction using Bernstein polynomials can be used to show that any function $F:[0,1] \to [0,1]$ can be approximated arbitrarily well by a single N-input pRAM of suitably high order. The simulation to be described below shows how the approximation power of a single N-pRAM improves with N. The function to be learned was the chaotic logistic series $x(t+1) = 3.97x(t)(1 - x(t))$; the training set was $P = 100$ consecutive points of this series, starting from $x(0) = 0.5$. Figure 1 shows how the mean absolute error decreases with epoch (one epoch is a presentation of the complete set of P patterns) for single pRAMs of input dimension $N = 4$, 6 and 8 (note that in a pulse-based output computation it is necessary to use independent spike trains on the N input lines carrying the single variable x). The parameters used in (8) and (10) were $c_R = 0.25$, $k_R = 1.5$, $k_r = 0.5$, $k_p = 2.0$. The choice $k_R + k_r = 2.0$ gives exponential convergence, and is the optimal choice for a spike train length function given by (8) and reward function given by (10). The penalty parameter λ was 0.01 for each case, but the training rate ρ differed so as to facilitate the plotting of the results on the same scale: for N=4, $\rho = 0.05$; for N=6, $\rho = 0.20$; for N=8, $\rho = 0.50$. It can clearly be seen how the approximation ability improves with input order N.

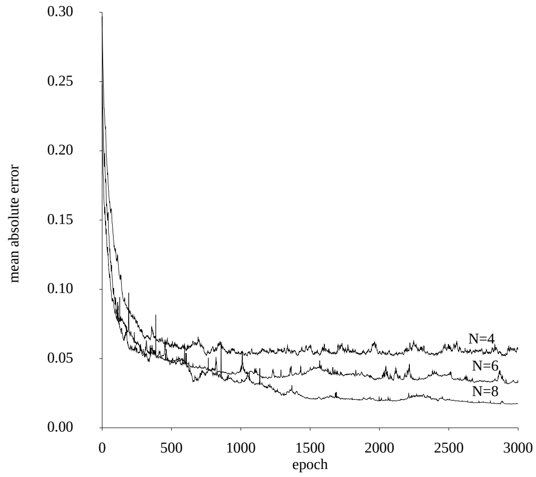


Figure 1

5. Adding Non-linear Output Transforms

Under certain circumstances, in particular when multi-pRAM networks with hidden units are used, it may be desirable to apply a non-linear 'squashing' transformation to the pRAM output function. This greatly extends the functionality of pRAM nets, and leads to more efficient function approximation. By applying a sigmoid-like output transformation a given function can usually be approximated by a more compact network which also uses pRAMs of lower input dimensionality. This reduces the number of free parameters, thereby improving generalisation. The output transform must be applied to the bit stream of (3), integration of which gives an approximation to the polynomial function (2), and applied in such a way as not to disrupt the free flow of pulses from input to final output (performance assessment), on which the stochastic reinforcement algorithm depends. The sigmoid output transform

$$\sigma(y) \;=\; \frac{1}{1 + e^{-y})} \tag{12}$$

is designed for models with output activities

$$y \;=\; \sum_j w_j x_j \;\;\in\; (-\infty,\, \infty)$$

not for the current case, where the activity y, as given by (2), is restricted to the interval [0,1]. The transform function

$$s(n,\, y) \;=\; \frac{y^n}{y^n \;+\; (1-y)^n} \tag{13}$$

illustrated for various values of n in Figure 2, preserves all the essential features of the sigmoid (12), being steepest (with a gradient equal to n) and closest to linear in its midrange, and with a gradient of zero at the limits of its argument ($y = 0, 1$ in this case). As well as being a more natural choice than the sigmoid (12) for arguments restricted to [0,1], s(n,y) has the very useful feature that it can be implemented by a simple pulse-based feedback circuit. The output bit stream (mean value y) is tapped and n+1 independently generated bits diverted to a buffer $a_0 a_1 \cdots a_n$. The n spikes $a_1 \cdots a_n$ are input to a recurrent (n+1)-input pRAM with fixed deterministic memory contents

$$\alpha_{\underline{u}} \;=\; (1-u_0) \prod_{j=1}^{n} u_j \;+\; u_0 \,(\, 1 - \prod_{j=1}^{n} (1-u_j)\,)$$

(the 0th input is the self-feedback line). The output of this recurrent pRAM has the mean value (13). By applying an output transform like (13) we are able to effectively increasing the range of the probabilistic weights (the $\alpha_{\underline{u}}$), which are of necessity restricted to the interval [0,1]. However it is commonly observed in the training of conventional neural networks that not all problems require the development of large weights (and the corresponding step-like output function). So the transform we are looking for should be one which can be applied adaptively. It is clear that the parameter n in (13) (the number of external inputs to the recurrent 'squashing' pRAM) is not a good choice for adaptation, since it must necessarily change in discrete (integer) steps, and the effect of such a change can be large (see Figure 2). A much better choice is the interpolation parameter β in the new output function

$$f(\beta, y) \;=\; (1 - \beta)\, y \;+\; \beta\, s(n, y) \tag{14}$$

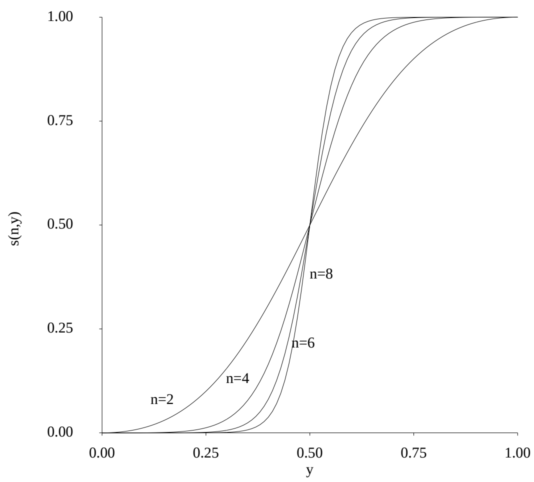
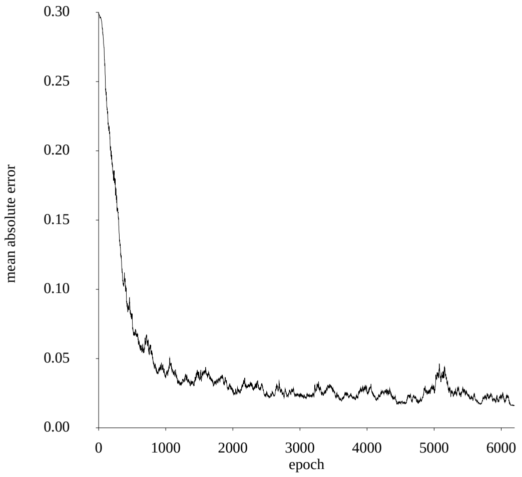


Figure 2

as this function can be accomplished by a 2-input pRAM with memory contents (0, β, $1-\beta$, 1) and inputs (i) from the untransformed bit stream with mean value y and (ii) from the spike output of the recurrent pRAM whose mean value is given by (13). (It should be noted that a different interpolation pRAM - and hence adaptive β - is to be attached to each pRAM neuron, and that if a single pRAM of high input order is replaced by a pyramidal structure then only the apex pRAM has an output transform circuit.) The variable β in (14) plays a similar role to the 'inverse temperature' parameter in the Boltzmann model. When it is smallest (highest 'temperature', $\beta = 0$) the output function is flattest, when it is largest (lowest 'temperature', $\beta = 1$) the output is most strongly compressed into a step-like form. Of course there will be a limit to the amount of squashing ultimately available which is determined by n. A suitable value for n must therefore be decided by simulation before any hardware is constructed. It appears from work done so far that n does not need to be large; a value between 2 and 4 was adequate for all applications we have investigated. The question remains as to how the interpolation parameter β is to be adapted. In fact all of the machinery is already in place, since this parameter (with its complement, 1-β) is contained in a pRAM and can be learned using the same techniques as were described in Section 3 (with a small amount of additional circuitry to ensure that the sum of the second and third locations in the interpolation pRAM always sum to 1, and that the contents of the first and fourth locations are not adapted). Thus the learning procedure is uniform over all modules; there is only one kind of adaptive circuit required to change both the parameters involved in polynomial fitting (the $\alpha_{\underline{u}}$) and the parameters which allow a non-polynomial 'tuning' of the output behaviour (the βs).

The task chosen to illustrate the use of non-linear output transforms is the same one which was used in Section 3 to show the approximation ability of single N-pRAMs, the chaotic logistic series $x(t+1) = 3.97x(t)(1 - x(t))$, with the same 100-pattern training set. The architecture used in this case had six 2-input pRAMs in the hidden layer and one 6-input pRAM in the output layer (there was no attempt to optimise the archtecture and it is likely that similar results could have been obtained with a somewhat smaller network). Adaptive output transform circuits (with n=2) were attached to all the pRAM neurons. The parameters c_R, k_R, k_r and k_p were set to the same values used in the single pRAM simulation. The $\alpha_{\underline{u}}$ had a training rate $\rho_\alpha = 0.025$ and penalty parameter $\lambda_\alpha = 0.01$; the β had a lower training rate $\rho_\beta = 0.0005$ and a higher penalty parameter $\lambda_\beta = 0.1$. The relatively large value chosen for λ_β was needed because training progressed most effectively when the β were initially set to zero, which is an absorbing point when $\lambda_\beta = 0$ [7]. A larger λ_β enables this initial setting to be escaped more easily should this be required. The network begins by trying to fit a polynomial approximation to the data ($\beta = 0$); if this is unsatisfactory the β then begin to increase, otherwise these parameters remain negligible. Figure 3a shows the decrease of mean absolute error with epoch for this example; it should be compared with Figure 1, where the same function is being approximated by single pRAMs of input dimension $N = 4$, 6 and 8. Notice that although the multilayer net in Figure 3a takes twice as many epochs to converge as the single N-pRAMs in Figure 1, the training rate for the $\alpha_{\underline{u}}$ in this case is only 1/8 that used for the 6-pRAM in Figure 1, and 1/20 that used for the 8-pRAM. The final error produced by the multilayer net is significantly less than that produced by the 6-pRAM, and in fact slightly lower than that produced by the 8-pRAM, despite the fact that the multilayer net has only 95 adaptive parameters (88 weights $\alpha_{\underline{u}}$, 7 transform parameters β) compared with the 256 weights of the 8-pRAM.



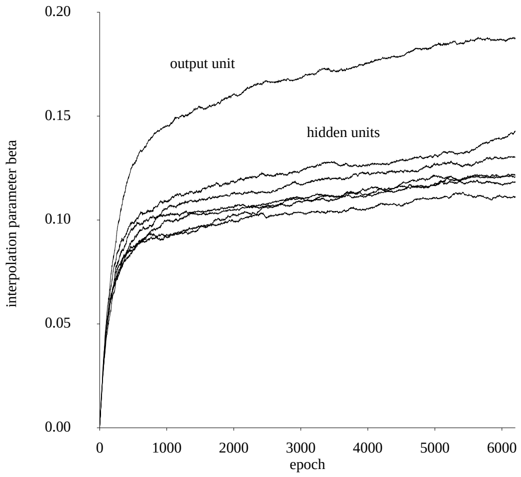


Figure 3a

Figure 3b

Figure 3b shows the evolution of the seven β in the network during training. The top curve is the transform parameter for the output unit, the cluster of six curves below it the βs associated with the hidden units. Notice that all seven β initially show a rapid increase - indicating that the network has discovered that pure polynomial approximation in a network of this architecture is not adequate - but thereafter level off. None of the β achieve a value close to the theoretical maximum of 1, indicating that a transform circuit with n=2 is easily sufficient in this example. However, although the β remain quite small, their presence is crucial to get a result of this quality - with all the β fixed at zero (no output transforms) this network converges to a final error about twice that achieved in Figure 3a.

Discussion

An algorithm has been presented which uses the noise associated with the representation of real-valued quantities by finite spike trains to drive a reinforcement algorithm which can enable a probabilistic RAM (pRAM) net to learn to approximate any function from $[0,1]^N$ to $[0,1]^M$ arbitrarily closely. Because the algorithm can in this way be directly implemented in self-contained hardware, it provides the possibility of embedded neurocircuitry with many applications in areas requiring a real-time learning capability. The underlying ideas are also applicable to any other pulse-based stochastic model in which the weights are treated as random variables.

References

[1] A G Barto, "Learning by statistical cooperation of self-interested neuron-like computing elements", *Human Neurobiol.* , 4, 229-256 (1985).

[2] A G Barto, R S Sutton and C W Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems", *IEEE Trans. Syst. Man and Cyb.* , 13, 834-846 (1983).

[3] V Gullapalli, "A stochastic reinforcement learning algorithm for learning real-valued functions", *Neural Networks* , 3, 671-692 (1990).

[4] D Gorse and J G Taylor, "An analysis of noisy RAM and neural nets", *Physica* , D34, 90-114 (1989); T G Clarkson, D Gorse and J G Taylor, "From wetware to hardware: reverse engineering using probabilistic RAMs", *Journal of Intelligent Systems* , 2, 11-30 (1992).

[5] T G Clarkson, C K Ng, D Gorse and J G Taylor, "Learning probabilistic RAM nets using VLSI structures" , *IEEE Transactions on Computers* , 41(12), 1552-1561 (1992).

[6] J G Taylor, "Spontaneous behaviour in neural networks", *J. Theor. Biol.* , 36, 513-528 (1972).

[7] D Gorse and J G Taylor, "Universal associative stochastic learning automata", *Neural Network World* , 1, 193-202 (1991).