Robot Learning with Neural Self-Organization *

Anthony H. Dekker
dekker@ACM.org

Pushkar K. Piggott
Intelligent Robotics Laboratory, Department of Computer Science,
University of Wollongong, Wollongong NSW 2522

Abstract. In this paper we describe a learning technique for reactive path-planning and guidance of autonomous robots using a Kohonen self-organizing neural network. The intended application is mineral exploration or agricultural monitoring using simple low-cost robots which do not contain a complete stored map of the environment, and may have unreliable sensor information. The network learns based on a simple sense of *value*, and our experiments with a simulated world show that this performs significantly better than rule-based learning.

1 Introduction

The central problem for an autonomous robot is how to respond in real time to events taking place around it. In particular, the robot must combine its own internal state with the current sensor input, and infer the most appropriate choice of *action*. There are two possible approaches to navigation of a robot: one is to have an internal *map*, and to navigate based on the current location; the second is to *react* to the current sensor input without knowledge of the current location. The reactive approach requires a less complex internal state, and avoids the problem of the information in the map becoming outdated.

Our interest lies in low-cost robots for mineral exploration and agricultural monitoring. For such robots a detailed map suitable for navigation would be expensive in terms of memory required, initial definition of the geographical features, and technology for measuring an accurate position. Instead we envisage a reactive robot which carries out simple tasks, and halts (activating a radio beacon) should it discover a situation requiring human intervention. A further problem for low-cost robots operating in an open-air environment is that sensor input is likely to be uncertain. For example, sonar ranging can give unreliable distances when the sonar beam is not normal to the target surface. The robot should be robust in the face of these uncertainties.

* **This paper appeared in proceedings of *Robots for Australian Industries*, National Conference of the Australian Robot Association, Melbourne, July 1995, pages 369–381.**

Considerable work has been done on reactive robot architectures, e.g. by Brooks [2] (for a survey see [15]). However, the design of appropriate robot behaviours can be very complex, and this complexity can be avoided by allowing the robot to *learn* the best behaviour [14, 18]. In previous work by the second author [15], the Lion learning algorithm for Q-learning of behaviour rules [18] was investigated and improved. However, the learning of fixed rules has disadvantages in the face of uncertain sensor input. The patterns on which the rules match may not be clearly defined, and unusual occurrences may result in the learning of an inappropriate rule. The use of *fuzzy* rules [7] may provide a solution to this problem, although it is not clear how fuzzy rules may be learned.

In this paper we investigate the learning of appropriate behaviours using a biologically plausible technique: Kohonen self-organizing neural networks [5, 6, 9, 10, 16]. We wish our (simulated) robot to use reinforcement learning based only on a simple sense of *value*. The Kohonen neural network operates by fitting itself to the distribution of sensor data. The network is flexible in the face of change, and allows behaviour to be represented in a way which effectively corresponds to fuzzy rules. We examine the performance of our learning algorithm using a simple testbed, and compare it with the rule-based Lion algorithm of [15] and with a simple fixed behaviour.

2 The Testbed

The testbed we use is taken from [15]. It simulates an autonomous mobile robot, and in spite of its simplicity, clearly illustrates our learning technique. The simulated environment is a 40×40 square grid of discrete locations, connected toroidally (i.e. opposite boundaries are considered to be joined). Rather than having four boundary walls, a number of walls of random length and orientation are placed on the grid. There are initially 40 *target items* distributed randomly, and the task is to capture these as quickly as possible. A training run terminates when all 40 target items have been captured.

The simulated robot is capable of three basic movements: moving ahead one unit, or turning in place 45° to the left or right. When a target item is directly in front of the robot, moving ahead has the effect of capturing the item. The robot has three sensors which view along a line of sight. The sensors are oriented to view straight ahead and 45° to the left and right. The sensors report whether a wall or a target item is visible along the line of sight, as well as the distance to this object. If nothing is visible up to the diameter of the world (40 units), then an 'empty' indication is reported, to avoid circular lines of sight. Figure 1 shows an example simulated world. The black areas are walls, and the grey circles are target items (10 of the target items have already been captured). The robot is shown as a triangle, with the three lines of sight shown. Walls are visible to the left and right, and a target item is visible 24 units ahead (wrapping around the toroid). The second triangle shows the position of the robot after turning right twice and moving ahead twice. It is now directly in front of a target item, ready to capture it.

Our goal in this work is to demonstrate learning from as simple a basis as possible. Consequently the robot is equipped with a simple sense of *value*: colliding with a wall is 'bad,' and capturing a target item is 'good.' The neural learning algorithm must derive a guidance technique from this basic sense of value. In particular the robot does not initially 'know' that seeing a target item directly ahead is 'good,' nor that it should move

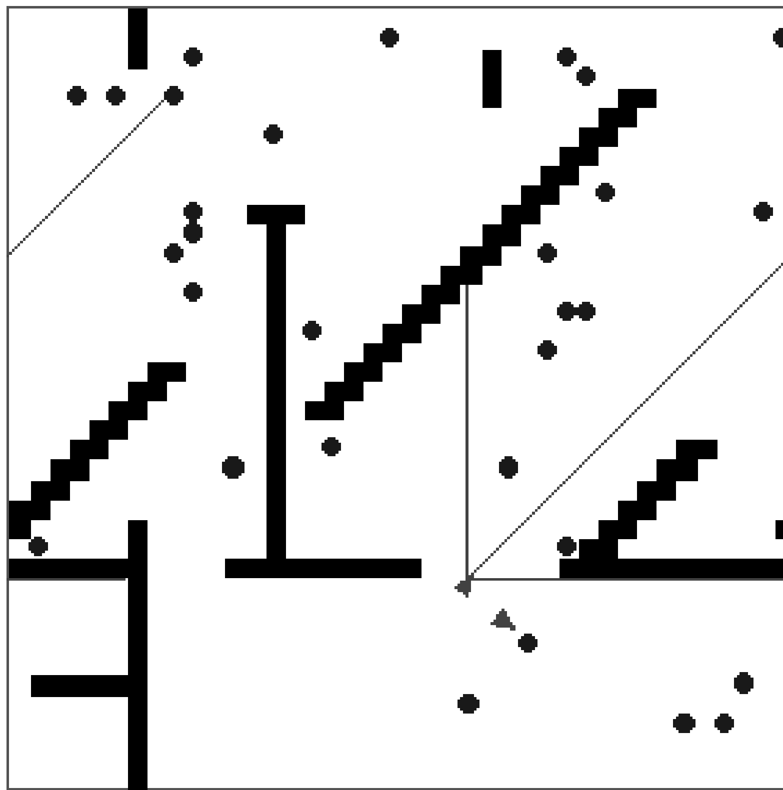

Figure 1: An example simulated world showing 30 target items

towards target items which it sees. The neural network is required to learn both of these things.

One consequence of this robot architecture is that in a given position the robot can decide to turn left (based on the sensor input), while in the new position it may decide to turn right, giving an infinitely repeatable cycle. We call this *dithering*, and the neural network must avoid such infinite loops.

We compare the behaviour of our robot with the improved Lion algorithm of [15] and with the following simple fixed behaviour:

> **if** a target item is visible ahead **then** move ahead
> **else if** a target item is visible to the left **then** turn left
> **else if** a target item is visible to the right **then** turn right
> **else** randomly choose one of {move ahead, turn left, turn right}

3 The Sensor Space

Our robot behaves as a *reactive agent*, and chooses one of the three possible moves based only on the current sensor input and the state of the neural network. In particular, the robot does not maintain a map of the world, nor does it use the network to derive a map (as do [12]). Since there are three sensors, the space of possible sensor data points is

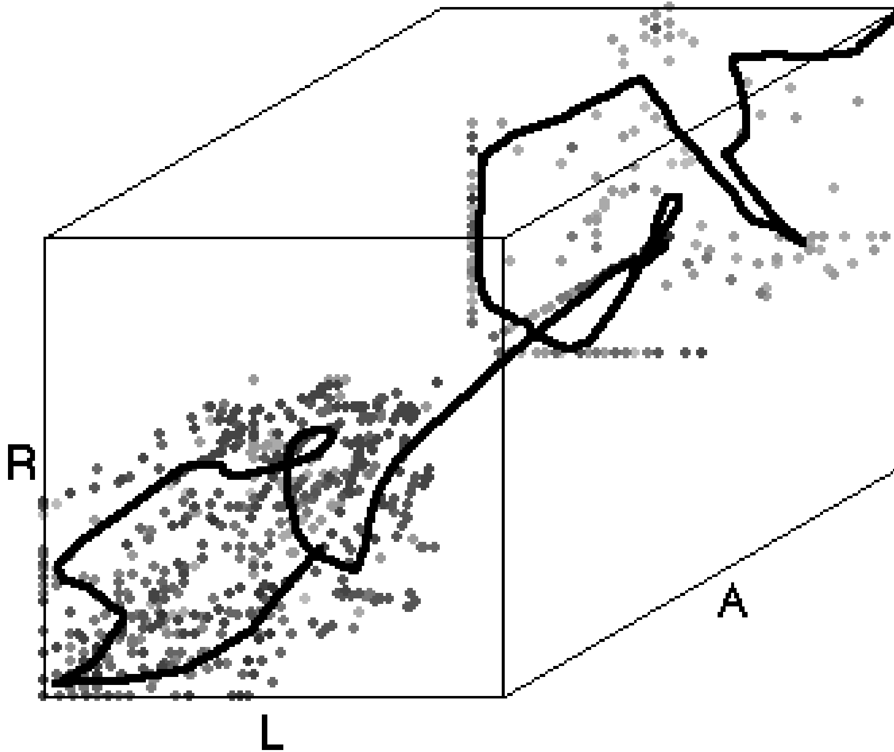

Figure 2: An example network fitted to a distribution of sensor data points

three-dimensional. We scale the sensor input s as follows:

$$
\begin{aligned}
s &= 0 \quad \text{if nothing is visible within 40 units} \\
s &= e^{\frac{-d}{15}} \quad \text{if a target item is visible } d \text{ units away } (0.07 \leq s \leq 1) \\
s &= -e^{\frac{-d}{15}} \quad \text{if a wall is visible } d \text{ units away } (-1 \leq s \leq -0.07)
\end{aligned}
$$

Thus the sensor space is the cube $[-1, 1]^3$. In addition, if a target item is visible on any sensor, we *mask* those sensors which detect walls, by setting $s = 0$. Masking means that when target items are perceived, the action taken does not need to depend on the location of the walls, i.e. if a target item is visible ahead, the presence of a wall on the left is not relevant. This reduces the amount of learning required. As a consequence of masking, the sensor data points occur only in the two sub-cubes $[-1, 0]^3$ and $[0, 1]^3$. Figure 2 shows the distribution of sensor data points within the cube $[-1, 1]^3$ over the course of a training run. The letters L, A, and R label axes showing s values from the left, ahead, and right sensors. The oldest sensor data points are shown in light grey, and the more recent sensor data points are darker. Note that data points near $(1, 1, 1)$ occur only at the beginning of the training run, since they indicate target items visible at close range in all three directions. As target items are captured, sensor data points with walls visible in all three directions become more common. Because of masking, many of the sensor data points in the upper sub-cube $[0, 1]^3$ are located on the axes (e.g. points like $(0.5, 0, 0)$) or the faces of the sub-cube (e.g points like $(0, 0.5, 0.5)$). Figure 2 also shows the neural network at the end of the training run, when it has learned a close fit to the sensor data.

Our goal is that all states where target items are visible become labelled 'good,' and that an appropriate choice of move be made in each case (moving ahead if a target item is

visible ahead, and turning towards a visible target item otherwise). In addition we hope that if only walls are visible an appropriate action will be taken which reduces the time required to find the next target item.

4 Kohonen Neural Networks

Kohonen neural networks [10, 16] and [9, sections 3.4 and 4.4] are a form of self-organizing neural network which define a mapping from a subset of $\mathbf{R}^n$ to $\mathbf{R}^m$, where $m \leq n$ is the dimension of the network. In other words, an m-dimensional abstraction is made of the n-dimensional input space. The mapping is observed to have three important properties. First, it is continuous almost everywhere on its domain, i.e. nearby data points usually map to nearby neurons. Second, the reverse map is continuous, i.e. nearby neurons map to nearby data points. Finally, the output of the map provides close to the maximum possible information about the input, although a *dimensional reduction* from n to m dimensions has taken place. There is, however, no general proof that these properties hold for every network. An analytical solution for the final network can be given for the case $n = m = 1$ [10, 16], or for $m = 1$ and a *neighbourhood* of fixed radius 0 or 2 [1]. For other cases, no analytical solution has been found, although the properties are observed to hold experimentally. The theoretical properties of Kohonen neural networks are also discussed in [3] and [4]. Kohonen neural networks are based on the behaviour of topological maps in the cerebral cortex of the brain. They have been applied to areas such as speech recognition [10], pattern recognition [11], creativity in theorem-proving [6], the learning of ballistic movements for a robot arm [17], modelling aerodynamic flow [8], and colour quantization of graphics images [5]. Our approach to Kohonen neural networks extends the technique presented in the latter work.

We use a Kohonen neural network consisting of a one-dimensional array of 256 neurons, each containing a weight vector (L_i, A_i, R_i). The network learns a mapping $\mathcal{M}$ from a sensor data point (L, A, R) to the index i of the closest weight vector. This mapping induces a function $\mathcal{F}$ from that subset of the cube $[-1, 1]^3$ which is occupied by data points, to the interval $[0, 255]$. The function $\mathcal{F}$ is defined by connecting adjacent weight vectors by line segments, as shown in figure 2, and mapping each point in the cube to the closest point on the line. The function $\mathcal{F}$ is continuous on its domain except for a finite number of surfaces which are equidistant from two parts of the line, i.e. which bisect 'loops' in the network. The continuity of $\mathcal{F}$ almost everywhere means that nearby sensor data points are almost always mapped to nearby indexes i. Consequently, what is learned about one sensor state can be generalised to similar states by propagation along the network. The inverse function $\mathcal{F}^{-1}$ is trivially continuous, but since the line segments between neurons tend to be relatively short, adjacent weight vectors (L_i, A_i, R_i) will usually represent similar sensor data points. Finally the mapping has the properties that the average mapping error $|L - L_i| + |A - A_i| + |R - R_i|$ is small, and that each index i occurs with approximately equal frequency.

This definition is sufficient to *recognise* a sensor state, but not to choose an action based on it. We add an *output* facility to the network, which extends that of [16]. A novel feature of our network is that it produces useful output, which successfully guides the robot, while learning is still in progress. To each neuron we add an *activity vector* $(A_i, A_{iL}, A_{iA}, A_{iR})$ and a *value vector* $(V_i, V_{iL}, V_{iA}, V_{iR})$. When a sensor data point (L, A, R) causes neuron i

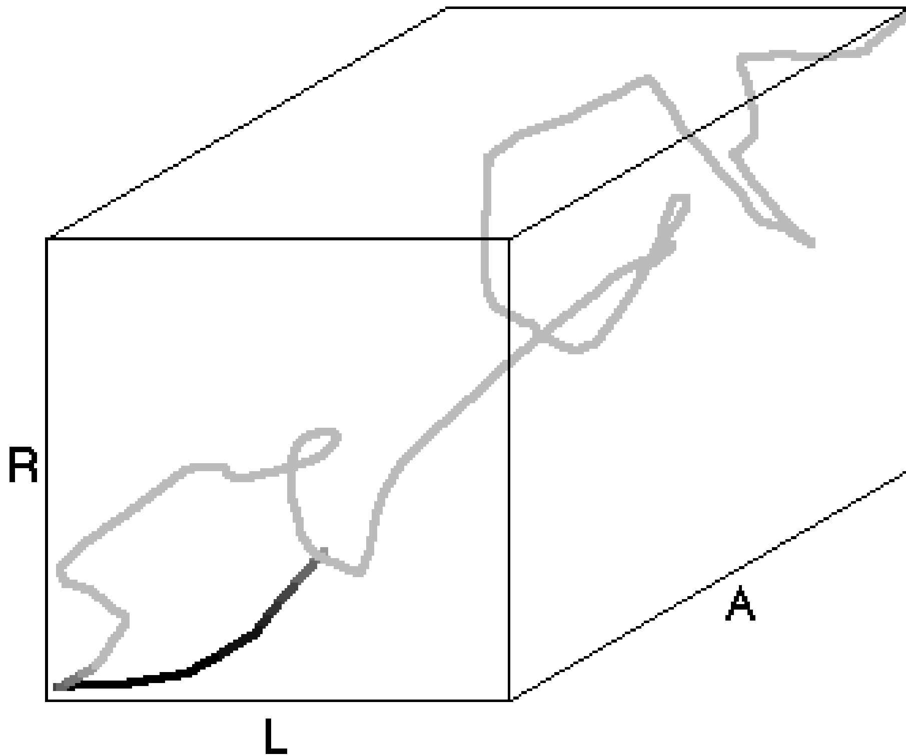

Figure 3: An example network showing an activity zone

to be activated and move j to be taken, the activities A_i and A_{ij} are set to 1, and activities in a neighbourhood of surrounding neurons are set to lesser values. These activities decay with time. This effectively provides a history mechanism: the network at any given time includes one or more *activity zones* with non-zero activity levels. These activity zones indicate neurons close to recently visited states. Figure 3 shows such an activity zone on the network, with black indicating an activity level of 1, and light grey an activity level of 0.

The *values* V_i in the range $-1 \ldots 1$ represent the perceived 'goodness' of the sensor state (L_i, A_i, R_i), while V_{ij} represents the perceived 'goodness' of making the move j from that state. Initially we set $V_{iA} = 0.5$ and $V_{iL} = V_{iR} = 0$, to give a bias towards moving ahead. The robot soon learns under which circumstances this is inappropriate. At any given time, the robot makes a probabilistic move based on the current values V_{ij}. Assuming $V_{ix} \leq V_{iy} \leq V_{iz}$, we compute:

$$q_j \quad = \quad \left(\frac{0.03 + V_{ij} - V_{ix}}{0.03 + V_{iz} - V_{ix}} \right)^2$$

for $j \in \{x, y, z\}$, and obtain three probabilities of the form:

$$p_j \quad = \quad \frac{q_j}{q_x + q_y + q_z}$$

Move j is then made with probability p_j. Figure 4 shows the value V_i for each neuron in the network after a full training run, with black indicating a value of 1, and light grey a value of -1. Line segments pointing left, ahead, or right indicate which of V_{iL}, V_{iA}, or

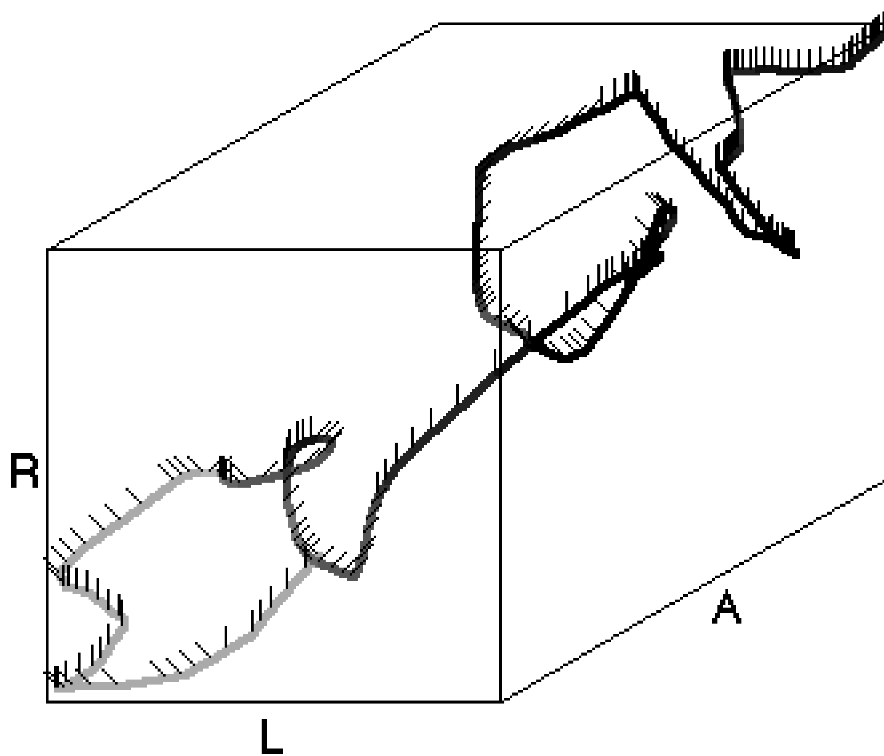

Figure 4: An example network showing values

V_{iR} is greatest. It can be seen that this generally corresponds with the direction in which a target item is visible. Where target items are not visible (at the lower left corner of the cube), there is more variation in move values. Examples of probability calculations are given in section 6.

The Kohonen neural network architecture provides both kinds of learning generalisation described by [14]. *Structural generalisation* is provided by the fact that all updates to the network apply to a *neighbourhood* of adjacent neurons, which correspond to similar sensor states. These updates are made most strongly at the central best-matching neuron, and more weakly at the edges of the neighbourhood. *Temporal generalisation* is provided by the activity mechanism, which is used to propagate 'good' and 'bad' values back in time to recently visited states.

One difficulty with the Kohonen approach is that values are recorded for *neurons*, rather than directly for sensor states. During the early phases of training, there is considerable movement of neurons through the sensor space. Consequently, the sensor state corresponding to a neuron can change with time, making learned values incorrect. In the next section we describe techniques for dealing with this problem, so that the network nevertheless provide an effective learning method.

5 The Learning Algorithm

Initially we set the weight vectors along the main diagonal of the sensor space, which provides a good first approximation to the input. At each training step, we find the 'best'

weight vector (L_i, A_i, R_i) corresponding to the input data point (L, A, R). This vector is then updated by moving it closer to the input, giving the neural network a better 'fit' to the data points:

$$(L_i, A_i, R_i) \ := \ \alpha(L, A, R) + (1 - \alpha)(L_i, A_i, R_i)$$

Here α is a parameter which is initially 1, and decreases with time. Kohonen [10, p 133] suggests decreasing α linearly, but we have found that results are improved and training time is decreased if α decreases *exponentially* down to a minimum of $\frac{1}{32}$.

As usual, we consider the network to be slightly 'elastic' in that when a weight vector is updated, the neighbouring vectors are also moved. Specifically, there is a neighbourhood of radius r, which decreases with time, and for $i - r \leq j \leq i + r$ (and $0 \leq j \leq 255$), we update the vectors in the neighbourhood by:

$$(L_j, A_j, R_j) \ := \ \alpha\rho_{(i,j,r)}(L, A, R) + (1 - \alpha\rho_{(i,j,r)})(L_j, A_j, R_j)$$

where $\rho_{(i,j,r)}$ is equal to 1 if $i = j$, decreasing as $|i - j|$ increases, down to 0 when $|i - j| = r$. We have found by experience that, as is the case with α, the values of r used significantly affect performance. The best results for this application are obtained when r decreases exponentially from 32 down to a minimum value of 6. This definition of r is combined with the following definition of $\rho_{(i,j,r)}$:

$$\rho_{(i,j,r)} = 1 - \left(\frac{|j - i|}{\lfloor r \rfloor}\right)^2$$

where $\lfloor r \rfloor$ is the integer part of r.

As well as moving the network in sensor space to make it a better abstraction of the sensor data, the activity and value vectors must also be updated. First we decay every old activity A_j and A_{jk} with $A := \frac{3}{4}A$, and then we update the activities in the neighbourhood of i by:

$$A \ := \ \max(1, A + \rho_{(i,j,r_A)})$$

where the radius used here is $r_A = 12$.

The value vectors are also decayed, if they are negative. This ensures that if particular neurons or moves are marked 'bad,' they will not be ignored forever. This compensates for incorrect values resulting from early movement of the network through sensor space. The decay is reduced as α is decreased:

$$V \ := \ V\left(1 - \frac{\alpha}{30}\right)$$

No decay is performed on positive values, since they will be automatically unlearned if they lead to inappropriate behaviour. Apart from possible decay, values can be updated in three ways. On capturing a target item, it can be stated with certainty that the value of the sensor state just before the capture should be 1 (i.e. maximum 'goodness'). The same applies to the capturing move. The sensor states and moves leading up to these should also have a positive value, although it is less certain that these were indeed 'good.' Our robot has no explicit memory of past sensor states or moves, but this information is provided by the activities which decay over time. Hence in capturing a target item, we

move all values V_j and V_{jk} towards 1 by a factor corresponding to the relevant activity A_j or A_{jk}:

$$V \;:=\; A + (1 - A)V$$

On colliding with a wall, we view the previous sensor states and moves as 'bad,' and move the values towards -1:

$$V \;:=\; -A + (1 - A)V$$

In addition, after colliding with a wall or capturing a target item, we set all activities to 0, to avoid interactions with subsequent events.

Collisions and captures (which provide a definite indication of 'good' or 'bad') are relatively infrequent during a training run. Consequently we also update values based on the perceived value V_i of the current sensor state. If this value is negative, then the preceding sensor states and moves were to some extent 'bad,' since they led to this poor state. We thus shift all values towards V_i by a factor $|V_i|A$, which is generally much less than the factor used for collisions and captures:

$$V \;:=\; |V_i|AV_i + (1 - |V_i|A)V$$

If the current sensor state has a positive value V_i, this means that the preceding sensor state values must have been *accurate*, but not necessarily 'good.' For example, a 'bad' state may have been visited along the way. Consequently we *reinforce* these existing values to reward their accuracy, by moving them *away* from V_i towards the extreme values of 1 or -1 (using V_i as an estimate of the average value over the network):

$$V \;:=\; -|V_i|AV_i + (1 + |V_i|A)V$$

Our results show that this learning strategy is indeed very effective. It should be noted that the update at each step takes approximately the same amount of time, which depends on the size of the neural network, but not on its values. This is useful in constructing real-time robot responses.

We have not yet defined the 'best' vector corresponding to an input. To ensure an even assignment of weight vectors to different regions of the sensor space [9, p 69], the 'best' vector for the input (L, A, R) is the weight vector (L_i, A_i, R_i) minimising:

$$|L - L_i| + |A - A_i| + |R - R_i| - b_i$$

where b_i is a bias factor which increases for less frequently chosen vectors. This allows a vector not to be chosen if it has been chosen 'too many' times already. The bias b_i is defined by:

$$b_i = \gamma\left(\frac{1}{256} - f_i\right)$$

where γ is a constant, and f_i estimates the frequency at which weight vector i is closest to the input. Initially $f_i = \frac{1}{256}$ and hence $b_i = 0$. At each training step, we first find the weight vector closest to the data point, and update f_i with $f_i := f_i - \beta f_i + \beta$, while the other weight vectors are updated with $f_i := f_i - \beta f_i$. For this application, the best performance is obtained when $\beta = \frac{1}{1024}$ and $\gamma = 8$. This technique (due to Desieno) ensures that data points are mapped evenly over the neural network, with each neuron being activated approximately equally often. This makes the maximum use of the network for learning.

6 A Training Example

Figure 1 shows an example simulated world after the first ten target items have been captured. Table 1 shows the steps taken in finding the eleventh target item. Initially a target item is visible 24 units ahead, but the value is negative, indicating that the robot has not yet learned that this state is 'good.' Eventually the learning process will propagate this information. Notice that the distances to walls in this state have been 'masked.' Both left and right turns from the initial state have values near zero, and the robot makes a random choice of a right turn, which has a probability of 0.325. In the next state, a target item is then visible to the right, and the robot has already learned that this is 'good.' The robot again chooses a right turn, which now has a probability of 0.679. The target item is then directly ahead, which has an extremely high 'goodness,' and the robot moves ahead three times (each move with probability 0.998). On the final move the target item is captured.

	Distances	Sensor State	Value	Move Values	Probabilities	Move
1	(–,24,–)	(0,0.20,0)	-0.374	(0.07,-0.71,-0.18)	(0.674,0.001,**0.325**)	R
2	(24,–,2)	(0.20,0,0.87)	0.808	(-0.03,0.60,0.90)	(0.001,0.320,**0.679**)	R
3	(–,2,5)	(0,0.87,0.72)	0.998	(0,0.96,0)	(0.001,**0.998**,0.001)	A
4	(–,1,–)	(0,0.93,0)	1.000	(0,1.00,0)	(0.001,**0.998**,0.001)	A
5	(–,0,–)	(0,1.00,0)	1.000	(0,1.00,0)	(0.001,**0.998**,0.001)	A

Table 1: An Example Training Run: The Eleventh Target Item

7 Results

Figure 5 shows a graph of the number of steps needed to capture each target item, averaged over 10 training runs. Results are shown for our neural network algorithm, the improved Lion algorithm of [15] and for the simple fixed behaviour described in section 2.

For the first five target items, our neural network algorithm performs better than the Lion algorithm, learning an approximate behaviour more quickly in the target-rich early environment. The neural network algorithm averages 23 steps per target item for the first five target items, compared with 77 for the Lion algorithm, and 11 for the fixed behaviour (averaged over 10 training runs). Over the next thirty target items the neural network algorithm averages 27 steps per target item, which is inferior to the Lion algorithm (20) and the fixed behaviour (17). This is due three factors. Firstly, the probabilistic nature of moves, which can occasionally result in a poor choice; secondly the fact that there is still movement of neurons through the sensor space, making some learned values incorrect; and thirdly the problem of *dithering* noted in section 2. The first two factors are the price paid for the flexibility of the neural network approach. Dithering is reduced by probabilistic nature of moves, but could be reduced further by explicitly detecting and penalising it.

On the last five target items, our neural network algorithm shows the benefits of its learning strategy, averaging 160 steps per target item, which is better than the Lion algorithm (205) and approaches that of the fixed behaviour (134). On the final target item,

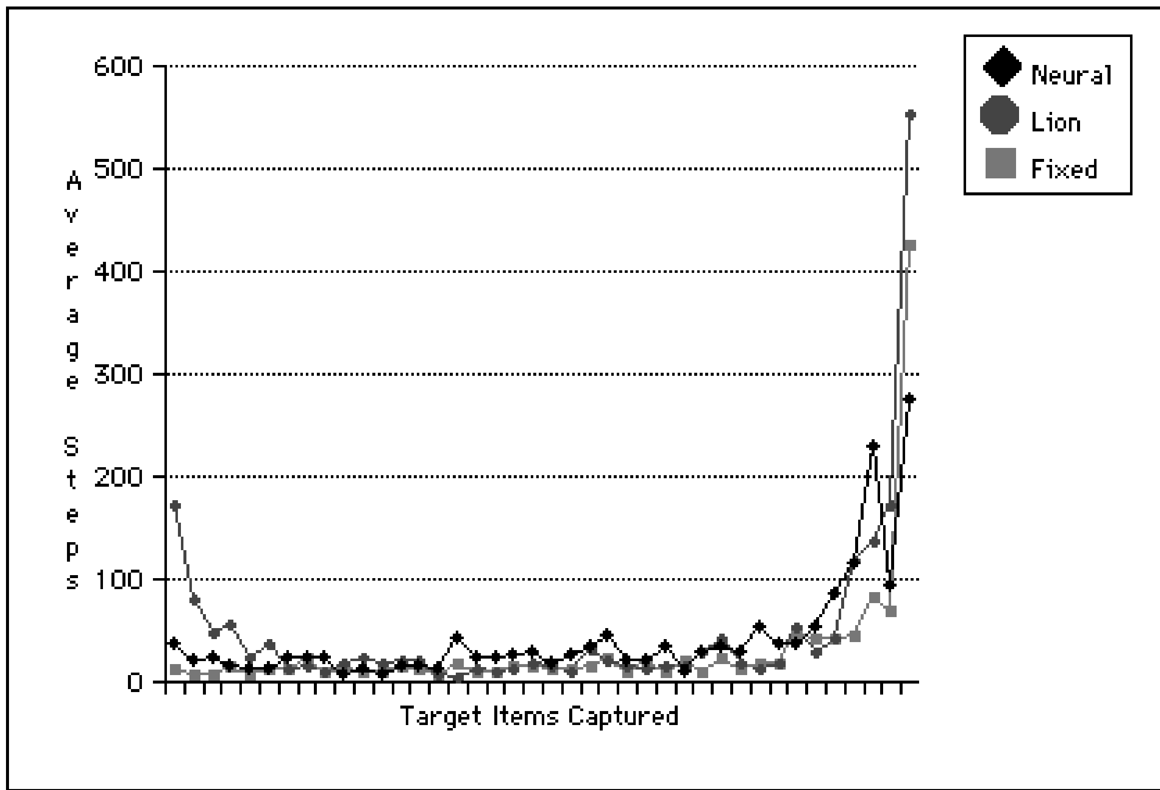

Figure 5: Performance Comparison

i.e. after learning is finally complete, the graph shows that the neural network algorithm, with 276 steps, outperforms both the Lion algorithm (554) and the fixed behaviour (426). In other words, by the end of training it has learned an exploration strategy which is better than the simple random walk in the fixed behaviour, and makes use of the distance-to-wall information to vary the robot's actions according to its location in the world. Averaging over all forty target items, the neural network algorithm (43 steps) clearly outperforms the Lion algorithm (51), coming much closer to the time taken by the fixed behaviour (31).

8 Related Work

Lin [13] uses a simulated environment slightly more complex than ours to study a number of reinforcement learning methods which combine connectionism with Q-learning. This approach differs substantially from our Kohonen neural network algorithm. Unfortunately Lin does not compare his connectionist approaches with other techniques.

Kröse and Eecen [12] extend the work of Ritter and Schulten [16, 17] on Kohonen neural networks for ballistic robot arm movements. Kröse and Eecen use a three-dimensional network to learn a map of the world based on range-sensor input. This map is then used for obstacle avoidance and path planning, using a gradient descent process. They assert that the dimensionality of the network must match the 'intrinsic dimensionality' (i.e. the degrees of freedom) of the robot. This need not be the case, however, because of the Kohonen neural network's ability to perform dimensional reduction. Their approach also

requires far more training data than our technique (although it is far more effective at avoiding collisions). However, their approach could usefully be combined with ours, in cases where it was desirable to learn map-based information.

9 Conclusions and Further Work

We have presented a biologically plausible neural network architecture for learning in an autonomous reactive robot, based on a simple sense of *value*. Because of its flexibility we expect it to be suitable for use with unreliable sensor data. Our results show that this technique performs significantly better than rule-based learning, in spite of the fact that movement of the network through sensor space can make older learned values incorrect. Neural self-organization is clearly an effective learning technique for exploration of a space such as our simulated world, where the important properties of the space, the choice of the best action, and the characteristics identifying target items from a distance are not known *a priori*, and must be learned through generalisation.

Further work is required on detecting and penalising *dithering*, without interfering with early exploratory activity. This can be expected to significantly improve performance. Simulations of a more elaborate robot would also be informative. A more complex sensor space may benefit from a two-dimensional Kohonen neural network, which would have fewer discontinuities, and could generalise more easily. The work of Kröse and Eecen [12] on learning maps could also be incorporated. Finally, we are currently investigating the possibility of extracting fuzzy expert system rules from the neural network after learning, and also the possibility of allowing the network to automatically learn to perform masking of sensor data, rather than imposing it directly.

10 Acknowledgements

The author is deeply indebted to Phillip McKerrow and members of the Intelligent Robotics Laboratory at the University of Wollongong for their useful suggestions.

References

[1] Catherine Bouton and Gilles Pagès. Self-organization and a.s. convergence of the one-dimensional Kohonen algorithm with non-uniformly distributed stimuli. *Stochastic Processes and their Applications*, 47(2):249–274, September 1993.

[2] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.

[3] Marie Cottrell and Jean-Claude Fort. Etude d'un algorithme d'auto-organisation. *Ann. Inst. Henri Poincaré*, 23(1):1–20, 1987.

[4] István Csabai, Tamás Geszti, and Gábor Vattay. Criticality in the one-dimensional Kohonen neural map. *Physical Review A*, 46(10):6181–6184, November 1992.

[5] Anthony Dekker. Kohonen neural networks for optimal colour quantization. *Network: Computation in Neural Systems*, 5:351–367, 1994.

[6] Anthony Dekker and Paul Farrow. Creativity, chaos, and artificial intelligence. In T. H. Dartnall, editor, *Artificial Intelligence and Creativity: An Interdisciplinary Approach*, pages 217–231. Kluwer Academic, Dordrecht, Holland, 1994.

[7] Dimiter Driankov, Peter W. Eklund, and Anca L. Ralescu, editors. *Fuzzy Logic and Fuzzy Control: Proceedings of the IJCAI Workshops, Sydney, Australia, August 24, 1991.* Number 833 in Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 1991.

[8] Robert Hecht-Nielsen. Neurocomputing: picking the human brain. *IEEE Spectrum*, 25(3):36–41, March 1988.

[9] Robert Hecht-Nielsen. *Neurocomputing.* Addison-Wesley, 1990.

[10] Teuvo Kohonen. *Self-Organization and Associative Memory.* Springer-Verlag, Berlin, third edition, 1989.

[11] Teuvo Kohonen. Pattern recognition by the self-organizing map. In Eduardo R. Caianiello, editor, *Parallel Architectures and Neural Networks: Third Italian Workshop, Vietri sul Mare, Salerno, 15–18 May, 1990*, pages 13–18. World Scientific, Singapore, 1990.

[12] Ben J. A. Kröse and Marc Eecen. A self-organizing representation of sensor space for mobile robot navigation. In *Proceedings of the IEEE/RSJ/GI International Conference on Intelligent Robots and Systems*, pages 9–14, 1994.

[13] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3/4):293–321, 1991.

[14] Sridhar Mahadevan and Jonathan Connell. Automatic programming of behaviour-based robots using reinforcement learning. *Artificial Intelligence*, 55:311–365, 1992.

[15] Pushkar Piggott and Abdul Sattar. Reinforcement learning of iterative behaviour with multiple sensors. *International Journal of Applied Intelligence*, 4(4):351–365, October 1994.

[16] Helge Ritter, Thomas Martinetz, and Klaus Schulten. *Neural Computation and Self-Organizing Maps: An Introduction.* Addison-Wesley, 1992.

[17] Helge Ritter and Klaus Schulten. Extending Kohonen's self-organizing mapping algorithm to learn ballistic movements. In Rolf Eckmiller and Christoph v.d. Malsburg, editors, *Neural Computers*, pages 393–406. Springer-Verlag, Berlin, 1989.

[18] Stephen D. Whitehead and Dana H. Ballard. Learning to perceive and act by trial and error. *Machine Learning*, 7(1):45–83, July 1991.