Connectionist Environment Modelling In A Real Robot

William Chesters and Gillian Hayes

Department of Artificial Intelligence, University of Edinburgh
5 Forrest Hill, Edinburgh EH1 2QL, Scotland
031-650 3079 & 3082 Fax: 031-225 9370
{williamc,gmh}@uk.ac.ed.aifh

This paper describes some experiments with an adaptive controller, based on multi-layer perceptrons, which tries to solve a simple reinforcement learning task for a real robot vehicle. One neural network (the 'model') is trained to predict how the robot's sensor readings will change if it performs a given action; another learns, with the aid of the model, to evaluate sensory states according to how close the robot is to receiving a reward when it experiences them. Two kinds of model which exploit context information were evaluated in robot runs, as well as a 'flat' (memoryless) model. The results confirm that backprop can provide the learning mechanism needed to solve simple adaptive control tasks, and point up some problems which will need to be addressed before it can help with more complicated skills.

Introduction

Nearly all robot learning experiments have been implemented in computer simulations. This paper describes an attempt to implement a simple reinforcement learning architecture, based on multi-layer perceptrons, in a real robot.

The core of the controller is an adaptive world model, which tries to predict what will happen at the next time-step if the robot performs a particular action. In many simulated learning experiments, the robot has easy access to information (such as its location) isomorphic to the space of different situations it might be in—its world has no 'hidden state'. A real robot, especially when it has only a few simple sensors at its disposal, must take account of circumstances which are not made explicit in its sense vector. This paper examines in particular two simple ways to give the world model some memory so that it can exploit context information in making its predictions: the use of a 'history' which tells the model what happened at the previous time step, and the incorporation of recurrent 'context' units into the model network. Both history and context models, as well as memory-less 'flat' models, were evaluated in experiments in which the robot was let loose in the arena with an initially untrained controller and allowed to learn what it could.

The Robot And Its Task

The Arena

The idea is for a robot vehicle to learn to find its way across a simple pen to some 'food', marked by a patch of black paper taped to the floor, while not bumping into the walls—see figure. The lamp is there to induce a light gradient across the pen, which it is intended that the robot learn to use to guide it to the food.

The Robot

The robot is constructed out of Technical Lego™. It carries ten simple sensors. Two microswitches on the vehicle's forward bumper detect collisions with the walls. Four infra-red proximity sensors (IRs) report objects in their field of view; their range readings are non-linear and rise sharply as the robot approaches an obstacle, until they saturate at about 25cm. The ones pointing roughly north-west and north-east serve to warn the robot of impending collisions, while the ones pointing backwards are mounted so as to saturate when the robot is actually in contact with a wall, or nearly so. Four wide-field (~70°) light-dependent resistors (LDRs) read the light intensity ahead, to the left and right, and underneath the vehicle; they give the robot the means to observe the light gradient at the bottom of which the food is placed. All sensors are normalised to read between 0 and 1.

By turning its two main drive wheels together or separately, the robot can support a limited repertoire of four actions: move ahead 30cm or astern 20cm, and turn left or right through 60°.

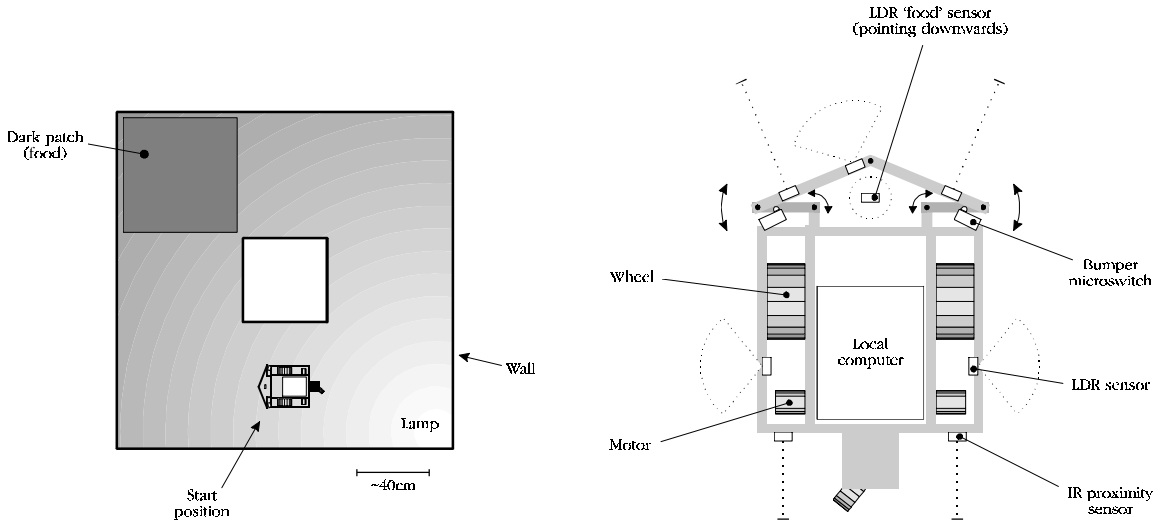
Reinforcement

A reward accrues whenever the robot is positioned on the food (whenever the light level measured by the ventral LDR drops below a threshold, indicating that it is looking at the black paper), and a penalty when it is jammed up against a wall (the bumper is depressed or the rear IRs are triggered). Most of the time, the robot receives no feedback.

$$\textsf{reinforcement}\ v = \begin{cases} 0 & \text{if } v\text{[bumper or rear IR]} > 0.75 \\ 1 & \text{if } v\text{[ventral LDR]} < 0.15 \\ \textit{undefined} & \text{otherwise} \end{cases}$$

The Controller

The controller comprises two MLPs (multi-layer perceptrons), called the **model** and the **heuristic**, and a procedure for choosing an action at each step, called the **policy**. The **model** aims to foresee what will happen if the robot carries out a particular action; the **heuristic** rates sense states according to how near the food they indicate the robot to be; and the **policy** draws on the two nets in choosing the best



The arena and the robot

action to perform. The nets are trained using the backpropagation algorithm.

Model

The model is trained at step $t+1$ to map the sense readings from step t (v_t), and the action that was performed (a_t), to the new sense readings. It is also allowed some context information (c_t), which is updated at each step according to one of the rules discussed below.

$$\text{model } c_t\ v_t\ a_t \rightarrow v_{t+1}$$
$$c_{t+1} = \text{update } c_t\ v_t$$

Simply training a feedforward network to predict the sense vector a robot will see next, given its current sense vector and the action it intends to perform, will not in general do. This is because in the real world—as opposed to experimental simulations such as that presented in Sutton 1990—the sense data available to the robot do not map one-to-one onto its state with respect to the world[1]. The problem is particularly acute if the robot has only a few simple sensors: it might be possible for insects, with vast amounts of information coming in from their eyes, to use reactive strategies such as they are reputed to employ; but if one is trying to employ sparse data effectively to solve a simple task—or rich data to solve a complex task—then some kind of memory is called for.

Williams 1990 suggests three classes of solution to this problem[2]. Two involve the use of "tapped delay lines" to make information about the past available to the model in the most direct way; but he comes down firmly on the side of the third, "radical" approach: the use of partially recurrent networks. In this paper, simple implementations of both ideas are evaluated.

Heuristic

The purpose of the heuristic is to interpolate between the sparse positive (non-*undefined*) readings provided by reinforcement. It is not trained directly on sensor readings, but rather indirectly using the model, in a series of pseudo-experiences. Broadly, it is taught to value a state at the value of the best state the robot thinks it can reach in one step starting there, scaled down by a constant discount rate: reinforcement is attenuated backwards along (supposed) robot trajectories[3].

Each pseudo-experience proceeds as follows. A state[4] s is picked (generated) at random, and the policy procedure is asked what the robot would do if it found itself in that situation. Then the model determines what state s' the robot would get into as a result of the chosen action, and the reinforcement oracle (or failing that the heuristic) provides a value for the new state. Finally the heuristic is 'backpropped' to rate s like s', but scaled down. Symbolically

$$\text{heuristic } s \rightarrow \frac{\text{evaluate } (\text{newState } s\ (\text{policy } s))}{\text{step penalty}}$$

where evaluate supplies the best known (pseudo-)reinforcement for a given state (c, v)

$$\text{evaluate } (c, v) = \begin{cases} \text{reinforcement } v & \text{if defined} \\ \text{heuristic } (c, v) & \text{otherwise} \end{cases}$$

---

[1]We ignore completely the issues that arise when things in the world are allowed to change.

[2]See also Hertz, Krogh & Palmer 1991, p. 176ff.

[3]Compare, for instance, Sutton 1990.

[4]i.e., context-sense vector pair

and newState is just a wrapper for model which works on states rather than sense vectors:

$$\text{newState}\ (c,\ v)\ a = (\text{update}\ c\ v\ ,\ \text{model}\ c\ v\ a)$$

By a process of relaxation, the heuristic comes to learn a function satisfying its recursive training equation and the boundary conditions provided by the reinforcement oracle.

Policy

Actions are chosen stochastically, the likelihood that a particular action will be performed in a particular state being proportional to its predicted payoff. That is to say, the model is used to predict what new experience will result from each of the four possible actions; these supposedly neighbouring experiences are then evaluated according to the reinforcement rule (for collisions or 'found the food' states) or using the heuristic (for other states); and these values are used as weights in deciding what to do.

Actually, there is a further twist in that the action weightings are scaled by an action-specific constant; and zero reinforcement predictions are adjusted upwards.

$$P[\text{policy}\ h = a] = \frac{\text{actionweight}\ h\ a}{\sum_{x \in action} \text{actionweight}\ h\ x}$$

where $action = \{\ forward,\ backward,\ left,\ right\ \}$, and

$$\text{actionweight}\ h\ a = (\max\{\ \text{evaluate}\ (\text{predict}\ h\ a),\ 0.05\ \})\ \times\ \text{preference}\ a$$

$$\text{preference} \begin{matrix} forward \\ backward \\ left \\ right \end{matrix} = \begin{matrix} 1.0 \\ 0.5 \\ 0.6 \\ 0.6 \end{matrix}$$

The point of incorporating an element of randomness into the policy is to ensure that the robot explores its environment thoroughly (by a kind of random walk) and doesn't get into a rut. Occasionally, for instance, it comes to believe that going forwards *always* results in a collision, and for a while it will avoid that action entirely[5]. But it will eventually move forwards anyway—simply because it tries even things it thinks are stupid sometimes (note the forcing of zero reinforcement predictions to 0.05)—and then its overly pessimistic theory will be disconfirmed.

The point of the preferences is just to keep the robot moving forward and speed up its exploration of the arena (recall too that the robot moves further in response to a *forward* command than it does in response to a *backward*).

Incremental Learning

It is generally held that backprop works best when it is given a sequence of training pairs selected at random from a representative body of data. The data collected by a robot and used to train the model network is, by contrast, likely to be biased, because its character will tend to change slowly

[5]If the robot gets stuck in the confined space between the central obstacle and one of the outer walls, it can even come to learn that it will be penalised whatever it does. In this situation it shows no systematic preference for any particular action beyond that introduced by its built-in preference scalars. This behaviour is somewhat similar to what psychologists call 'learnt helplessness'.

as the robot moves around. On the face of it, it appears to be necessary to buffer a lot of experiences as a pool from which randomized samples may be selected; and that is potentially expensive if the environment is complicated[6].

The technique investigated as a way of circumventing this problem is based on the idea that a network can be used as a device for regenerating something like the training data that went into its construction. If a randomly chosen vector r could plausibly have been an input vector in the previous training data[7], then applying the network to r ought—to the extent that the network has been satisfactorily trained—to yield the corresponding output vector. So we should have a fair substitute for a library of training data available to us, at the cost only of storing an old version of the network.

When a new training pair (u, v) comes in for network N,

```
make a copy of the network as it is now (call it N0)
repeat learnings times
    backprop N to associate u with v
    repeat rememberings times
        pick a vector r at random
        let s be the output to which N0 maps r
        backprop N to associate r with s
```

The idea is to use the training data regenerated from the old network N_0 as a constraint to ensure that new mappings don't obliterate important old ones.

It is clearly desirable to run through enough r vectors ('old training pairs') to cover a significant proportion of N's input space—otherwise there is no reason to suppose that interesting old mappings will be protected—and for a medium-sized network, this is going to mean rather many ('*rememberings*'). There is then a real danger that the new data will simply get lost in amongst them, and will not have the desired influence, which is why the procedure is repeated several times ('*learnings*').

Modelling

The robot has two goals set for it through the **reinforcement** function: it is supposed to avoid obstacles, and find the food. Obstacle avoidance can be learnt by attending to the IR and bumper sensors, and the 'characteristic timescale' is short: actions have immediate large effects on these sensor readings, and feedback from **reinforcement** is received straight away. To learn to find the food, on the other hand, the robot needs to model the effect of its actions on the LDR readings, which change gradually as it moves around, and reinforcement is delayed (so that guidance from the **heuristic** is needed). We consider the two problems separately.

Obstacles

Some experiments with driving the robot manually demonstrated that (at least) the following fairly stable phenomena were there to be learnt:

(1) moving forwards, left or right when already jammed head-on against a wall (or backwards when jammed tail-on) does not change anything

---

[6]But see Myers 1992 on the subject of "attention-directed buffering".

[7]This will not always be the case, because the encoding of the robot's sensor data is redundant. What is the **model** to make of a random pseudo-sense vector which includes a bumper switch reading of 0.37, when the only values encountered in its training data are 0 and 1? It would be easy to generate only legal pseudo-sense-vectors, but for the experiments described below, purely random vectors were used.

(2) moving backwards always releases the bumpers, and reduces the forward IR readings; driving forwards always reduces the rear IR readings to near their minimum

(3) moving forwards when both the forward IRs are strongly activated often results in a forward collision: these sensors were in fact carefully placed so that this would in general (if the robot was not approaching a wall at a shallow angle) be true

(4) backing away from a forward collision, and then immediately moving forwards, results in another collision (because the robot moves further in response to a *forward* command than it does for a *backward*)

(5) moving forwards three times is quite likely to run the robot against a wall (because the arena is fairly cramped)

Obviously some context information beyond the bare v_t and a_t is required for the model to be able to spot the last two situations. All three kinds of model (flat, history and recurrent) were tried out in robot runs; the results are summarised in the following table.

Type of model	*Learnings, Rememberings*	*Steps to plateau**	*Collisions thereafter†*	*Number of robot runs*	*Steps per run*
Flat	1, 0	50	44%	10	300
	10, 60	50	26%	10	300
History	7, 40	50	24%	10	300
Recurrent	–	450	26%	5	700

*The number of robot steps needed on average for the controller to reach its best performance (in multiples of 50)
†The proportion of robot steps (after the controller's performance had flattened off) at which a collision occurred

'Flat' Models

A 'flat' model[8] which makes no use of context information can learn (1), (2) and (3) within forty to 100 steps. It does not automagically generalise 'correctly': for instance, most head-on collisions are characterised by both bumper switches and both forward IRs being activated simultaneously, and when the net is first presented with a situation in which perhaps only one side of the bumper is depressed, or only one or neither of the IRs is triggered, it does not correctly predict the result of moving forwards (namely, continued collision). And not surprisingly, given that its training data is presented in a most un-random order, a flat model has trouble remembering everything it learns. For instance, if the robot happens to spend some time in 'free space' out of contact with the walls of the arena, it can learn that the bumper switches are *never* quite activated[9]. If learning is disabled at any time, the model will most likely be frozen in a version which fails in some class of situations; with learning enabled, it only needs a few steps to re-learn (for instance) about (1), and the overall behaviour of the robot is reasonably satisfactory.

The 'incremental learning' technique does go some way to alleviating the model's forgetfulness. When ordinary backprop is used (*learnings* = 1, *rememberings* = 0), the robot needs a lot of reminding about how to avoid walls, and ends up jammed against them (receiving negative reinforcement) at 44% of the steps. With *learnings* = 10 and *rememberings* = 60 this figure drops to 26%. Of course, this improvement is achieved at the cost of a sixhundredfold increase in

[8]Flat model nets have 14 inputs (for ten sensors plus a one-out-of-four encoded action), 14 hidden units and ten outputs (for ten sensors).

[9]Although real bumper switch readings can only ever take the values 0 and 1, the model is free to predict any value in between. If its prediction exceeds 0.75, then it is treated as an 'on' by the reinforcement function and hence by the policy.

computation. For comparison, with an untrained model the robot suffers collisions at 78% of the steps. The model requires about 50 steps to achieve its best performance.

Flat models are limited in what they can learn by their lack of memory, and this causes the robot to do stupid things, such as backing off from a wall and then immediately driving forwards into it again (cf. (4)). If the model is to make helpful predictions in these cases, it needs some context information.

'History' Models

Perhaps the simplest form of context information is a history of what the robot has recently seen—Williams 1990's "tapped delay line" on the model's input. Here we consider only one-step histories: in the notation of this paper,

$$c_t = v_{t-1} \mathbin{@} a_{t-1}$$

(@ means concatenation). The model[10] should now be able to make an appropriate warning prediction in situation (4), since it knows that the robot's last action was to move backwards from a collision.

In fact, history models do often learn about (4). They also tend to believe (fairly enough) that moving forwards twice results in the forward IRs being activated, as well as making an actual collision fairly likely[11] (compare (5)).

The robot performed very slightly better with a history model than with a flat one, spending 24% of its time up against walls when *learnings* = 7 and *rememberings* = 40. Although the extra information available to history models did allow them to make better predictions when they were working well, they were somewhat more prone to forgetting than the *learnings* = 10, *rememberings* = 60 flat models. The history models' larger input space should mean that more *rememberings* are needed for the 'incremental learning' technique to have a beneficial effect, but because they are quite a lot bigger than flat models, with 1064 weights as opposed to 360, they consume that much more CPU time per backprop pass, and it proved impractical to make robot runs with higher settings. As before, performance flattened out after 50 steps or so.

Recurrent Models

To extend a history-based context further back into the past would quickly become expensive (especially if the robot had higher-bandwidth sensors at its disposal). A tidier solution is to let the net develop a rolling context representation of its own, along the lines of an Elman or Jordan net[12]. A recurrent model was implemented with context units updated according to the rule

$$c_{t+1} = 0.5\ c_t + 0.5\ hidden_t$$

—a weighted sum of the current context and the current activations of the net's hidden units. The context units have (modifiable) connections to the net's hidden units, just like the input units.With suitable weights, as Elman 1990 points out, the model can in principle have an indefinitely long

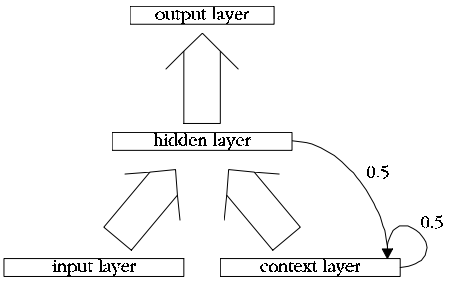
---

[10]History models have 28 inputs (twice as many as flat models), 28 hidden units and ten outputs.

[11]Although real bumper switch readings are always 0 or 1, the model can predict that the next reading will be anywhere between 0 and 1; what tends to happen is that 'risk factors' such as moving forwards, having moved forwards at the last step, or activated forward IRs all push up the predicted bumper readings, so that when several such factors come together, the prediction will pass the threshold beyond which negative reinforcement applies. MLPs are inherently prone to this kind of interpolation: if they can't say for sure whether the reading will be 0 or 1, they have to predict a weighted average in order to get the expected error down. Compare Elman 1990.

[12]See for instance Hertz, Krogh & Palmer 1991 pp. 179ff.

memory, encoded compactly in a way tailored to its task; and partially recurrent nets have indeed been found to learn such encodings for themselves in a wide variety of simulations.



A recurrent model

With a recurrent model[13], the robot spent 26% of its time stuck against obstacles. It did not require the use of 'incremental learning' to achieve these results, so that although recurrent models take 450 or so steps before they start performing at their best—as opposed to 50 for both the incrementally-learned models—they use up less computer time along the way. The greater the overlap in the network's input between two situations with different outcomes, the more likely it is that weight adjustments appropriate for one are necessarily going to interfere with the net's response in the other, and so a context memory that distinguishes between situations better than the bare sense vector will have the desirable side-effect of rendering the model more stable. Interestingly, recurrent models only started outperforming *learnings* = 1, *rememberings* = 0 flat models after about 250 steps, presumably because the changing 'meanings' of the context units interfere with the conclusions the net might draw from the sense vector alone.

Recurrent models do generally learn about situation (4), but not about (5), perhaps because the fairly low momentum parameter on the context units (0.5) means that their weights would have to be set quite sensitively to implement a suitable memory that extended over the necessary three steps. Of the various models tested, recurrent models seemed to make the best predictions when they were working well, but they were more prone to forgetting (and subsequent painful periods of relearning) than non-recurrent models with incremental learning.

Light Levels

If it is to find the food efficiently, the robot is ultimately going to have to learn how the ventral light levels it observes change in response to its actions as it moves around, so that it can find its way down the light gradient; the following regularities were found to hold most of the time, although not as robustly as those relating to obstacles ((1)–(5)):

(6) if the robot moves forwards twice in row (and does not run into a wall in the process), there is a consistent trend in the ventral light level—if it goes (e.g.) up after the first step, it will go up after the second as well

(7) if it is brighter on the right than on the left, then turning to the right and moving forwards will increase the ventral light level (and vice versa)

Direct Prediction

Modelling absolute light levels with a sequentially trained net was attempted but proved impossible. The likely reason for this was that they change rather slowly, so that models of every kind could simply keep predicting a constant

$$(\text{model } c_t \; v_t \; a_t)[\text{ventral LDR}] = k \approx v_{t+1}[\text{ventral LDR}]$$

which floats up and down as the network is trained on each new sense vector. At each particular step, the prediction is more or less right, but if one presents the model with a historical or hypothetical situation, it will in general make a very poor prediction.

[13]Recurrent models have 14 inputs like flat models, 14 hidden units with 14 context units, and ten outputs.

$$(\textsf{model}\ c_x\ v_x\ a_x)[\text{ventral LDR}] = k \neq v_{x+1}[\text{ventral LDR}]$$

Providing the **model** with context information does not help: if a network can substantially reduce its cost function by outputting a constant, it won't learn anything more sophisticated.

Generous use of 'incremental learning' (with *learnings* = 3 and *rememberings* = 100) did make a **model** trained on recorded data learn, after several thousand steps, a function whose peaks and troughs matched those of the input light levels, but whose overall range was only about a fifth as great.

Randomised Training Data

A history **model** trained on data recorded from a robot run and presented to it in random order learned something much more like an identity function:

$$(\textsf{model}\ c_t\ v_t\ a_t)[\text{ventral LDR}] \approx v_t[\text{ventral LDR}] \approx v_{t+1}[\text{ventral LDR}]$$

But it did not learn to predict the small changes which need to be controlled if the robot is to find its way down the light gradient, perhaps because they were not large enough to cause big errors and force the net to take account of them.

Predicting Differences

The LDR data has a small, fast component—because every time the robot moves, the readings change slightly—but it also has a large, slow component caused by the robot's gradual semi-random walk around the arena between light and dark areas. A sequentially trained **model** will not capture regularities whose characteristic timescale is longer than a few steps; but there is an easy way to tease the two components apart and isolate the fast one: get the **model** to predict the *difference* between this reading and the coming one, rather than its absolute value. It is after all the light *gradient* in which we would like the **model** to interest itself.

0.5 was added to the difference targets to fit them into the net's [0, 1] output range; they were also scaled up by a constant m to fill the whole interval rather than clustering around the middle:

$$\textsf{model}\ c_t\ v_t\ a_t \rightarrow m\,(l_{t+1} - l_t) + 0.5$$

where l_t is the vector of light readings from v_t. The particular action taken by the robot now has a strong effect on the signal the **model** is trying to predict.

Nevertheless, flat, history and recurrent **model**s all failed to predict light level changes accurately. They always picked up on just one salient fact about the data—that if the readings are very high, they are likely to fall, and vice versa—and learned to map even moderately high (low) readings to negative (positive) changes, while ignoring other regularities, including (6) and (7). This looks to be a case of unwanted interpolation: the net will naturally tend to 'generalise' the rule "if the reading is very high, it will fall" to "the higher the reading, the more it will fall". Initially, this is not a bad way to get the cost function down. But then it seems the net is unable to make further progress, perhaps because it cannot incrementally mutate this generalised rule into a more discriminating one—it is stuck in a 'local maximum'.

Conclusions

The controller described in this paper did learn to reduce the number collisions suffered by the robot. The versions with flat, history and recurrent **model**s were all about equally effective, though for different reasons. The recurrent **model** was computationally the cheapest, since the others required time-consuming 'incremental learning' to perform at their best; but it needed more physical robot steps before it began to work well.

There is an irreducible minimum of bumps which the robot cannot be expected to foresee; but others would be avoidable if the **model** was less prone to forget what it had learnt. 'Incremental learning' is of only limited help here.

One way to solve the forgetting problem would be by selective buffering of experiences for later reference as training data for the model. In general, though, buffering (like 'incremental learning') is incompatible with the use of adaptive context representations such as those developed by recurrent **model**s, because there is no 'neutral language' in which to store the buffered experiences' contexts. If it is known that the context information is largely determined by the preceding n readings leading up to it, then a library of n-step sequences will serve. But systems capable of supporting the acquisition of significant skills will probably have to maintain a hierarchy of contexts at different timescales, and it is not clear how buffering can be integrated into such a scheme.

The **model**s were less successful at capturing how the light levels around the robot varied as it moved around its pen. Sequentially trained **model**s would predict accurately neither absolute readings (because of the forgetting problem) nor differences between successive readings (because of over-generalisation or interpolation); randomly trained **model**s would not take account of the small action-dependent component of the signal, for reasons unknown. (This modelling failure meant that the **heuristic** could not be trained to guide the robot to the food.) If backprop nets are to learn (with or without buffering) to model awkward data such as the LDR readings discussed above, they are going to need some help from signal preprocessing.

The problem of over-eager interpolation is perhaps inherent in any adaptive learning device which works by composing continuous functions; it might be possible to find an alternative cost function which penalised persistent small systematic errors as much as occasional big surprises, and discouraged the net from quickly landing itself with a simple, but dead-end, solution.

References

Elman 1990. *Finding Structure in Time*, in *Cognitive Science* **14**, pp. 179–211.

Hertz, Krogh & Palmer 1991. *Introduction to the Theory of Neural Computation*, Addison-Wesley.

Miller, Sutton & Werbos (eds.), 1990. *Neural Networks for Control*, MIT Press.

Myers 1992. *Delay Learning in Artificial Neural Networks*, Chapman & Hall.

Sutton 1990. *First Results with Dyna*, in Miller, Sutton & Werbos 1990, p. 179.

Williams 1990. *Adaptive State Representation and Estimation Using Recurrent Connectionist Networks*, in Miller, Sutton & Werbos 1990, p. 97.

The authors would like to thank Edinburgh University, the Department of AI, and particularly everyone in the workshop for help with the hardware. William Chesters was supported by a SERC Advanced Studentship, and currently holds a Research Studentship; he would like to thank SERC.