Transactions on Information and Communications Technologies vol 8, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

Application of AI based reinforcement learning to robot vehicle control

M.G.M. Madden, P.J. Nolan
Department of Mechanical Engineering, University College, Galway, Ireland

Abstract

Reinforcement learning is a form of artificial intelligence in which an agent acquires and improves skills based on receiving positive and negative rewards when it performs actions within an environment. This paper describes a system which uses an extended reinforcement learning algorithm to generate reactive control strategies. It is applied to the control of a vehicle in a simulated traffic environment.

1 Reinforcement Learning

1.1 Introduction

Reinforcement learning is an artificial intelligence methodology whereby an autonomous agent, the learner, can acquire and improve skills within an environment without having an explicit teacher. It has been defined by Sutton [12] as the learning of a mapping from situations to actions so as to maximize a reward or reinforcement signal. The learner is not told which action to take but instead must discover which actions yield the highest reward by trying them. Reinforcements (reward or punishment signals) are received by the learner from the environment in which it operates; following Russell [8], the agent's task can be formulated as maximizing the long-term reward it receives. Actions may affect not only the reinforcements received immediately but also reinforcements received later.

Reinforcement learning can be regarded as a form of dynamic programming [2]. Barto *et al.* [1] trace the relationship between reinforcement learning and dynamic programming for solving problems in optimal control. They state that such algorithms provide an appropriate basis for generating reactive strategies for real-time control and for learning reactive strategies when the system being controlled is incompletely known.

Transactions on Information and Communications Technologies vol 8, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

The particular algorithm used in this work is based on tabular Q-learning [13] with planning. This has been found by several researchers (e.g. refs. [3, 10]) to be an effective form of reinforcement learning. Moore and Atkeson [5] found Q-learning algorithms which use planning to be more effective than the basic Q-learning algorithm and they have offered some convincing arguments why planning should be expected to improve performance. The mathematical basis of Q-Learning is outlined in the following section.

1.2 Q-Learning Algorithm

In reinforcement learning, the agent's aim is to maximise its *expected total discounted reward.* The expected total discounted reward at a particular time step k can be expressed as:

$$E\left\{ \sum_{j=0}^{\infty} \gamma^j r(k+j) \right\} \tag{1}$$

where, at any time step l, $r(l)$ is the immediate reward received from the environment. γ is a fixed discount factor, ranging of 0 to 1, which serves to make less immediate rewards less valuable. Therefore, the expected total discounted reward is the sum of the immediate reward and the current value of all future rewards. Watkins [13] defined a function called the Q-function, which assigns to each state-action pair a measurement of the expected total discounted reward which would be obtained if the given action was carried out in the given state and the optimal policy followed thereafter. If the current state is x, the current action is a, the resulting immediate reward is r and the next state is y, then:

$$Q(x,a) = E\{r + \gamma V(y) \mid x,a\} = R(x,a) + \gamma \sum_{y} P_{xy}(a) V(y) \tag{2}$$

where $R(x,a) = E\{r|x,a\}$, $V(x) = \max_a Q(x,a)$ and $P_{xy}(a)$ is the probability of making a state transition from x to y as a result of performing action a. From this Q-function, the optimal policy is $\arg\max_a Q(x,a)$. Watkins' Q-learning algorithm is based on calculating an estimate $\hat{Q}$ of the Q-function, as follows:

$$\hat{Q}(x,a) \leftarrow \hat{Q}(x,a) + \alpha(r + \gamma \hat{V}(y) - \hat{Q}(x,a)) \tag{2}$$

where α is a learning rate parameter in the range of 0 to 1 and $\hat{V}(y) = \max_b \hat{Q}(y,b)$. Then the optimal action can be *estimated* as $\arg\max_a \hat{Q}(x,a)$.

The algorithm used here, shown in Figure 1, is derived from those described by Moore & Atkeson [5] and Peng & Williams [6]. The following notes apply:

1. In step 3.2, the action is chosen according to the following procedure:
   - Select a random number, n.
   - If n is greater than some threshold, choose the action a which maximizes $\hat{Q}(x,a)$ over all a. Ties are broken randomly.
   - Alternatively, if n is less than the threshold, choose an action a at random from those available. This encourages exploration of the action space.
2. In step 3.5, the prediction difference is defined as $e = |r + \gamma \hat{V}(y) - \hat{Q}(x,a)|$.
3. In step 3.7, the predecessors of a state p are any other states q which have at least once in the history of the system have been involved in the one-step transition $q \rightarrow p$.

1. Initialise all values in the Q-value array, $\hat{Q}(x,a)$, to 0.
2. $y \leftarrow$ initial state.
3. Repeat indefinitely:
   - 3.1 $x \leftarrow y$.
   - 3.2 Choose an action a. (See note 1 below.)
   - 3.3 Execute action a. $y \leftarrow$ new state; $r \leftarrow$ reward received.
   - 3.4 Update the world model with the tuple $\tau = (x, a, y, r)$.
   - 3.5 Calculate the prediction difference, e. (See note 2 below.)
   - 3.6 If $e < \delta$:
     If (x, a) are not already on the queue, insert τ with priority e;
     otherwise, if e exceeds its current priority, promote it to priority e.
   - 3.7 While the priority queue is not empty and the number of updates in this iteration does not exceed limit:
     - $(x', a', y', r') \leftarrow$ tuple removed from top of priority queue.
     - Update Q-value table: $\hat{Q}(x',a') \leftarrow \hat{Q}(x',a') + \alpha(r' + \gamma\hat{V}(y') - \hat{Q}(x',a'))$.
     - For each predecessor x'' of x' (see note 3 below), calculate the prediction difference and if appropriate insert in the priority queue following the rules of steps 3.5 and 3.6 above.

Figure 1: Steps in Q-Learning Algorithm

1.3 The Icarus System

The learning system which is being used in this work, named Icarus, is the authors' re-implementation of Sutton's Dyna architecture [10, 11]. It consists of the following four main components, as shown in Figure 2:

- A reinforcement learner.
- A policy function.
- An interface with the external environment.
- The system's internal model of the environment.

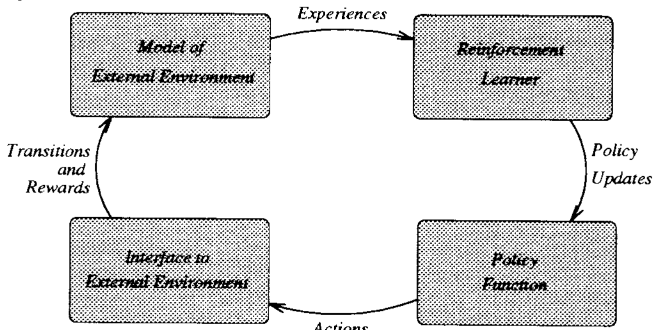

Figure 2: The Icarus Architecture

The *reinforcement learner* is based on Q-learning (described in the previous section) and uses the algorithm described in the previous section. The *policy function* determines what action will be selected in any state. The *interface* to the external environment executes the chosen action, transitions to a new state in the environment and receives a reinforcement from the environment. The *internal*

model is a simply a record of tuples, where each represents a state, action taken, new state and immediate reward; these are built up from the system's experiences of taking actions in the external environment.

The reinforcement learner adjusts the policy function dynamically, based on what is currently known about the external environment; this knowledge is stored in the internal model. The policy function selects an action which is sent, via the interface, to be executed in the external environment. Any immediate reward received back from the external environment is recorded in the internal model and thus may affect subsequent actions of the policy function.

Icarus has been implemented in C on a UNIX workstation. The interface to the external environment uses the Berkeley sockets mechanism [9] to pass information between Icarus and the process representing the external environment, which may be running on a different computer.

2 The Pharos Traffic Simulator

Pharos (Public Highway and Road Simulator) is a detailed simulation model of a street environment [7]. It may be used to model streets and multi-lane highways, with bends and intersections. It incorporates entities such as traffic lights, signs and road markings. It controls the behaviour of zombie vehicles. It also permits an externally-controlled vehicle, referred to as the robot vehicle, to operate within the environment. Figure 3 is a screen-capture of Pharos, with the traffic moving along a segment of a two-lane highway.

Originally, Pharos was used in conjunction with a rule-based expert system called Ulysses [7] which controlled the actions of the robot vehicle. Ulysses implements a computational model of driving. In order to make driving decisions, it sends requests to Pharos for high-level perceptual information and subsequently sends acceleration and steering commands to control the behaviour of the robot within Pharos, as determined by its rule base.

In this work, Pharos has been adapted in order to use it as a test application for reinforcement learning. This adapted version is referred to here as Pharos/R. Since Pharos was primarily intended for use with Ulysses, which has comprehensive knowledge of how to operate within the Pharos environment and which therefore never sends entirely inappropriate command requests to the simulator, it was not designed to handle such inappropriate requests gracefully. For example, requesting a change over to the next lane on the right when the robot is already in the rightmost lane on a Pharos highway can cause the simulator to crash.

In contrast with Ulysses, Icarus is meant to learn appropriate behaviour from attempting all sorts of actions and receiving positive and negative reinforcements based on whether actions are appropriate or inappropriate. Thus, Icarus might well send a request to change lanes to the right when already in the rightmost lane, but should receive a negative reinforcement for requesting that manoeuvre. If the effect of such command would be catastrophic, then it must be filtered out.

There are two categories of rewards and penalties may be incurred within Pharos/R. If the robot issues a command request which is inappropriate then an immediate penalty is incurred. Such reinforcements are calculated in the command-filtering code. Alternatively, if a command is acceptable then its repercussions may not be immediate and must be calculated after a simulation time

Transactions on Information and Communications Technologies vol 8, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

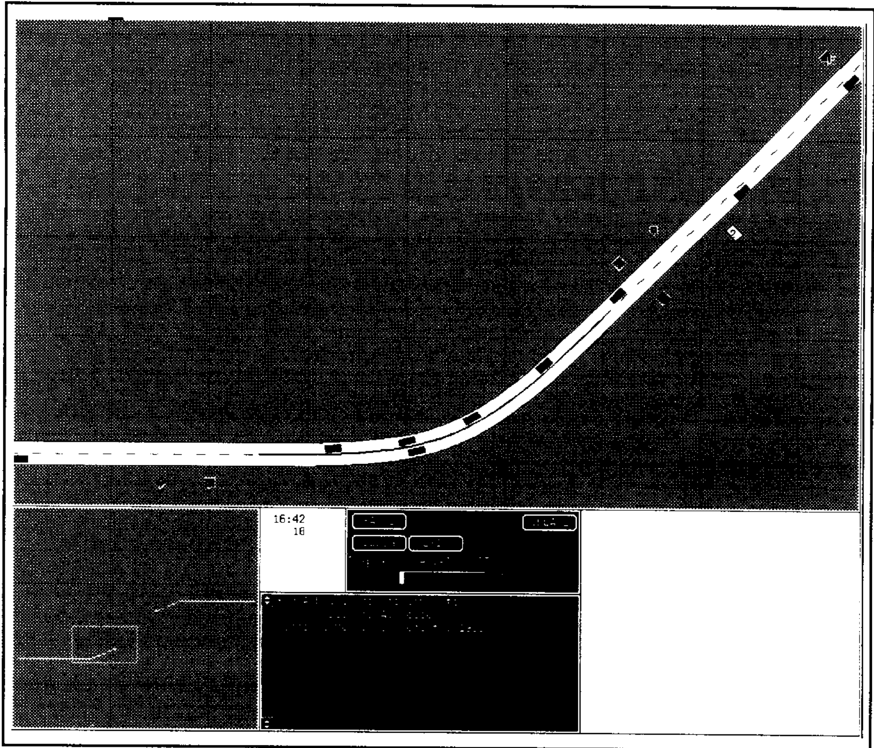

Figure 3: The Graphical Interface of Pharos

step. For example, the effect of acceleration on distance travelled is not immediate. If the robot issues an acceleration command and a reward is given to it on the basis of progress it has made along the highway, that reward must be calculated at the end of the simulation time step rather than at the beginning in order to be relevant. Pharos/R has functions to calculate the reinforcement associated both with commands which must be processed (e.g. filtered) as soon as they are received and with commands the effects of which are appreciable only after a finite time delay. Pharos/R passes the reinforcement that it calculates back to the robot's controller at an appropriate point in the simulation cycle. Slight changes were made to the message passing protocol originally used in Pharos so that it works under different hardware systems than that on which it was originally developed.

3 Experiments with Pharos

Experiments to date have focussed on a limited part of the task of learning to control a car in Pharos/R. The task that has been considered is that of learning acceleration control. Icarus' interface with the external environment requests information from Pharos/R about the robot's speed and the distance to the next car. Icarus' learning module discretises these values and uses them to index an array of Q-values. The set of commands which Icarus can send to the robot car,

Transactions on Information and Communications Technologies vol 8, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

for acceleration control, consists of a discrete set of acceleration values. In the particular experiments described in this section, the commands are: accelerate at 20 ft/s^2, 10 ft/s^2 or 5 ft/s^2; decelerate at 20 ft/s^2, 10 ft/s^2 or 5 ft/s^2; no acceleration.

Some experimental results from controlling a robot car using Icarus within the Pharos/R environment are shown in Figure 4. Each plot is a graph of the robot's speed (in ft/s) versus its distance along the road segment (in ft).

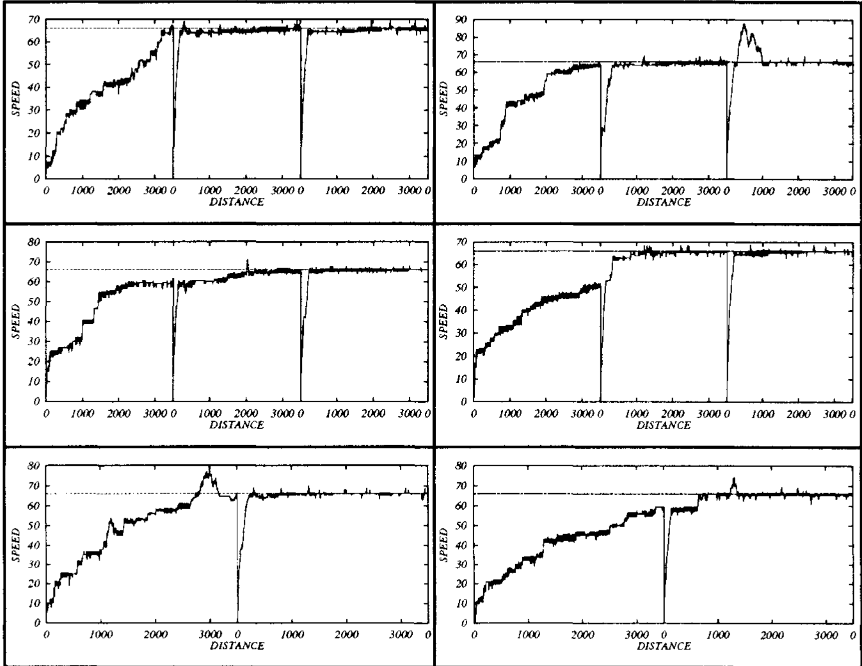

Figure 4: Plots of Speed versus Distance for Pharos/R

In each run, the controller initially has no information about how to act. It therefore begins by making random moves; thus there is a noticeable difference in the car's behaviour in the first part of each run. In each case, the car starts with an initial speed of 0 feet/second and the speed limit is 66 feet/second. The highway segment is 3500 feet long, and upon reaching the end of the segment the robot is returned to the start with speed 0 again, to repeat the task. These points, where the robot's position and speed are reset and it immediately starts to accelerate again, show up as large downward-pointing spikes on the graphs.

It can be seen that after having the robot's speed reset to zero, Icarus directs the robot to accelerate quite rapidly over the range of speeds that it has previously covered. (From the graphs, the average acceleration is in the region of 15 ft/s^2.) It can also be seen that the punishment signals for exceeding the speed limit have a pronounced effect; once the robot reaches the speed limit it oscillates about that speed. The robot's speed shows a tendency to converge to the speed limit, but it does not completely settle at the speed limit for two reasons:

- Icarus' acceleration control is coarse, thus making it often impossible for the robot's speed to reach the exact optimum speed of 66 feet/second.

- The learning system explores different possible behaviours by occasionally attempting moves other than that which has been calculated to be the best. In fact it makes the what it has calculated to be the 'best' move only 90% of the time; the rest of the time, it selects a move randomly.

4 Bias from Past Experience

As can be seen from Figure 4 above, the car is initially slow in accelerating as each new speed corresponds to a new state about which it initially has no information. It makes random moves, discovers some information about the state, and is subsequently able to select a better strategy. Thus, the first time that Icarus starts from rest it accelerates slowly, but when its speed is reset to zero (after 3500 ft) it accelerates rapidly to reach the speed it had been at before the reset.

In general, progress tends to be slow when the learner is moving into states that have not been previously explored, since it initially knows nothing about them and therefore just moves at random. In many cases, however, the optimal move in two adjoining states is the same. Hence, in the absence of any information about what is the best move to make in a given state, making the move which was best in the previous state might be better than moving at random.

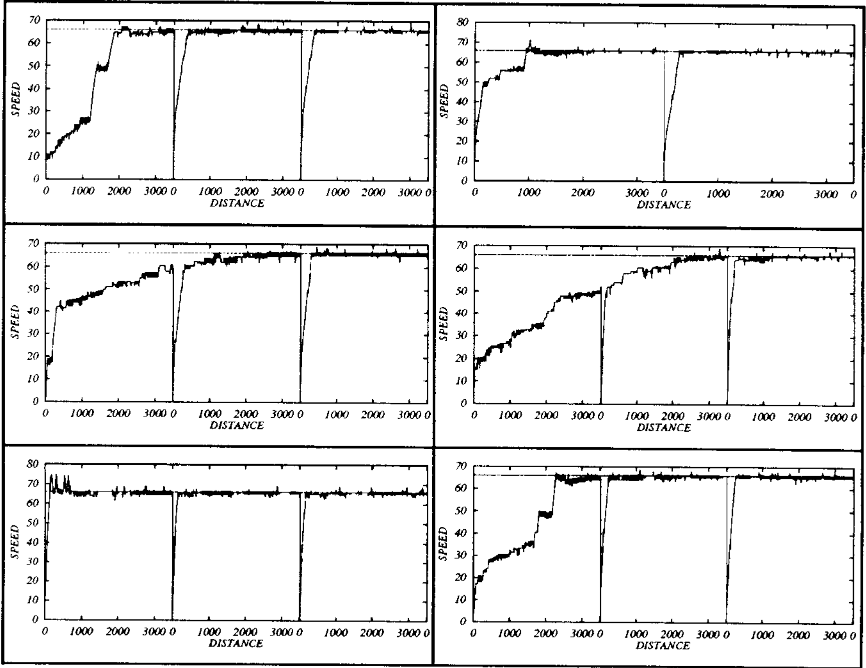

Figure 5: Plots of Speed versus Distance with Bias from Past Experience

Based on this observation, the reinforcement learning module of Icarus has been modified. If nothing is known about the current state but there is information available about the prior state, then rather than moving at random, the move which was optimal in the prior state is generally made. As before, a random move is sometimes made rather than the 'better' move, in order to facilitate exploration.

Transactions on Information and Communications Technologies vol 8, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

This strategy is termed *bias from past experience.* Figure 5 (see previous page) shows some results from the Pharos control task using bias from past experience. Comparing these with the graphs of Figure 3, it may be seen that, using of the bias from previous experience, Icarus generally tends to converge on a good policy more quickly. In a total of 12 runs, the average distance travelled before settling at the optimal speed was 4897 feet. With bias from past experience, this distance (again averaged over 12 runs) dropped to 3436 feet, representing 30% quicker convergence to an optimal control policy.

5 Discussion and Future Possibilities

The current system focusses on a single task, that of acceleration control on a highway with no lane-changing and no intersections. In order to be able to deal with different driving scenarios, such as multi-lane roads, intersections without traffic, intersections with traffic and closely-spaced intersections, it might be worthwhile to use a so-called subsumption architecture similar to that proposed by Mahadevan and Connell [4] for their robot box-pushing task. This approach decomposes the overall task to be learned into subtasks, each of which have a separate learning module. This improves reinforcement learning performance by converting the problem of learning a complex task into that of learning a simpler set of special-purpose tasks.

Icarus with the Pharos/R environment uses what may be termed passive perception: it makes the same perceptual requests at each time step. In contrast with this, Ulysses (the original expert system for controlling the robot in Pharos) uses an active perception system: at each time step it makes perceptual requests based on the current situation in order to acquire specific information. Whitehead and Ballard [14] have done some work on active perception for a simple block-stacking domain, with a system that learns not only how to solve a task but where to focus its attention in order to gather important sensory data. There is scope for incorporating a similar approach into the system described here.

There are many engineering systems which could benefit from the use of controllers based on reinforcement learning. In general, a control system such as Icarus would be applicable to any system for which a clearly defined control strategy might not be available, but for which good and bad states can be identified.

6 Conclusions

This paper has described an adaptive controller, Icarus, which uses an extended reinforcement learning approach to generate reactive control strategies. This has been used to control the acceleration behaviour of a robot vehicle in a simulated traffic environment. The results which have been presented show that Icarus successfully develops an optimal strategy for the control task.

The extension to the reinforcement learning algorithm, in which random selection of actions in new system states is biased by previous experience in similar states, has also been presented. Inclusion of bias from past experience has been shown to significantly improve convergence of the controller to an optimal strategy for acceleration control.

Transactions on Information and Communications Technologies vol 8, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

Acknowledgements

Part of the work described here was carried out by Michael Madden while a visitor at the Department of Mechanical Engineering, University of California at Berkeley. The authors gratefully acknowledge the assistance of Prof. Alice Agogino of the Berkeley Expert Systems Technology laboratory, Prof. Stuart Russell of the department of Computer Science at Berkeley, and Prof. Stuart Dreyfus of the department of Industrial Engineering and Operations Research there, for providing access to computing facilities and for invaluable discussions.

References

1. Barto, A.G., Bradke, S.J. and Singh, A.P.; 1993. *Learning to Act using Real-Time Dynamic Programming.* Preprint: Submitted to AI Journal special issue on Computational Theories of Interaction and Agency.
2. Bellman, R.E. and Dreyfus, S.E.; 1962. *Applied Dynamic Programming.* Princeton University Press, Princeton, NJ.
3. Lin, Long-Ji; 1991. *Self-Improvement based on reinforcement learning, planning and tracking.* Proc. Eighth Int'l Conf. on Machine Learning.
4. Mahadevan, S. and Connell, J.; 1991. *Scaling Reinforcement Learning to Robotics by Exploiting the Subsumption Architecture.* Proc. Eighth Int'l Conf. on Machine Learning.
5. Moore, A. W. and Atkeson, C.G.; 1992. *Prioritized Sweeping: Reinforcement Learning with Less Data and Less Real Time.* MIT Artificial Intelligence Laboratory, Cambridge, MA.
6. Peng, Jing and Williams, Ronald J.; 1992. *Efficient Learning and Planning Within the Dyna Framework.* Proc. SAB-92.
7. Reece, Douglas and Shafer, Steven; 1991. *A computational model of Driving for Autonomous Vehicles.* Report CMU-CS-91-122, Carnegie Mellon University, Pittsburgh, PA.
8. Russell, Stuart J.; 1989. *Execution Architectures and Compilation.* Proc. IJCAI-89, 15-20.
9. Sun Microsystems, Inc.; 1990: *SunOS Network Programming Guide,* Mountain View, CA.
10. Sutton, Richard S., 1990. *Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming.* Proc. Seventh Int'l Conf. on Machine Learning.
11. Sutton, Richard S.; 1991. *Dyna, an Integrated Architecture for Learning, Planning, and Reacting.* Working Notes of AAAI Spring Symposium on Integrated Intelligent Architectures.
12. Sutton, Richard S.; 1992. *The Challenge of Reinforcement Learning.* Machine Learning, 8, 225-227.
13. Watkins, C.J.C.H., 1989. *Learning From Delayed Rewards.* Ph.D. Thesis, King's College, Cambridge University, England.
14. Whitehead, S.D. and Ballard, D.H.; 1991. *Learning to Perceive and Act by Trial and Error.* Machine Learning, 7, 45-83.