Some Explorations in Reinforcement Learning Techniques Applied to the Problem of Learning to Play Pinball

Nathaniel Scott Winstead
Tulane University Department of Computer Science
New Orleans, Louisiana 70118
winstead@cs.tulane.edu

Abstract

Historically, the accepted approach to control problems in physically complicated domains has been through machine learning, due to the fact that knowledge engineering in these domains can be extremely complicated. When the already physically complicated domain is also continuous and dynamical (possibly with composite and/or sequential goals), the learning task becomes even more difficult due to ambiguities of reward assignment in these domains. However, these continuous, complicated, dynamical domains can effectively be modeled discretely as *Markov Decision Processes*, which would suggest using a *Temporal Difference* learning approach on the problem. This is the traditional method of approaching these problems. In Temporal Difference learning, the value of a discrete action is defined to be the difference in some value (usually an expected reward) between the current state and and its predecessor state. However, in the problem of playing pinball, the traditional Temporal Difference methods converge slowly and perform poorly. This leads to the addition of knowledge engineering elements to the traditional Temporal Difference methods, which was previously considered difficult to do. However, by making straightforward, simple changes to the basic Temporal Difference algorithm to incorporate knowledge engineering I was able to both speed convergence of the algorithm, and greatly improve performance.

Composite and/or sequential tasks may similarly be modeled as Markov Decision Processes, which again suggests the use of Temporal Difference learning. However, applying Temporal Difference learning to composite and/or sequential continuous, dynamical tasks has been traditionally viewed as a burdensome task, involving complex changes to the basic learning architecture. Again, one goal was to keep the learning architecture simple, despite the complexity of the environment. Starting with the existing composite Temporal Difference methods, I have constructed a new composite technique which maintains the elegance and simplicity of the Temporal Difference architecture, while enabling for learning of composite tasks.

Why Pinball?

Motivations

Previous research on learning problems in robot manipulation (Christiansen, Mason, & Mitchell 1991) has focused on tasks with dynamic behavior of short duration, where dynamic aspects could be abstracted from the learned models. The research reported in this paper considers tasks with a significant temporal dependency, rather than the (apparently) more static tasks that have been previously considered.

One can argue that a pinball machine, when interfaced to a computer and a vision system to track the ball, is in fact a robot. The system has sensors (vision and contact sensors for the game targets) and effectors (the flippers). The pinball task has a goal (score lots of points), and automated planning to achieve the goal is required for performance above the novice level. The pinball task is also very dynamic.

Pinball is also interesting as a control task in that the player has only intermittent opportunities to affect the system state, unlike other well-studied control tasks such as pole balancing (Moriarty & Miikkulainen 1996), (Hougsen, Fischer, & Johnam 1994), *et cetera*. Only when the ball is close to the flippers is there any chance to change the ball's velocity, as "bumping" the table has not been implemented. When the ball is away from the flippers, the player is "at the mercy" of the environment, which is complex and not completely deterministic. Thus, credit assignment for good and bad actions becomes difficult.

A final advantage of the pinball task is that the system's performance can be compared directly to human performance. Novice and expert players may play the game, and those scores can be compared to the average score achieved by the pinball agent. Though it would be interesting to construct the best agent possible, based on human knowledge of the game, the current research focuses on *learning* agents, which can improve their performance via practice in the task.

Markovian Decision Tasks

Many problems for reinforcement learning tasks are Markovian Decision Tasks (MDT's). MDT's are finite-state, discrete-time, stochastic dynamical systems with some important attributes. Namely, there exists a finite set of states S, and a finite set of actions A available to the agent. At each time step t, the agent performs an action $a_t \in A$ based on the observation of the current state $x_t \in S$. The agent receives some real-valued payoff, $R(x_t, a_t)$ and the state makes a transition to state $x_{t+1} \in S$ with probability $P_{x_t x_{t+1}}(a_t)$. The environment satisfies the Markovian Property: the current state and future actions determine the expected future sequence of states and payoffs independently of the state trajectory prior to the current state.

The goal of any development within the MDT framework is to create agents which are capable of establishing closed-loop policies which will maximize the total payoff over time.

We say that a control policy is *optimal* when it achieves the agent's goal or goals. Optimal policies derived by RL techniques are usually able to perform the agent's goal or goals without failure. Because of the fact that the agent in the pinball environment only has intermittent control over the ball, is is unclear what an optimal control policy would be. Would it mean that the ball could be kept in play forever, and continue to score points? Therefore, we will not use the term "optimal" with regards to control policies in the pinball environment.

Clearly, then, Pinball is *not* an MDT. It fulfills all of the criteria, including the Markov Property. However, the game of Pinball is *neither* discrete-time *nor* finite-state. Fortunately, a problem which has infinite or countably infinite states may be modeled as an MDT by simply specifying a set of discrete values onto which real values (or a subset of them) will be mapped. Similarly, we specify intervals at which the agent can exert control over the environment, instead of allowing control at any time. This allows us to apply all of the properties of an MDT to the learning task we are considering.

The Pinball Simulator

Currently, all trials are done with a simulated pinball machine. The simulator software was written in C++ for a Unix/X-Windows environment. Currently, the simulator comprises approximately 1800 lines of code.

Some simplifying assumptions were made for practical purposes in creating the simulator:

- Flippers move infinitely fast (i.e., at the time the flipper button is pressed, the flipper moves immediately to the position where it would contact the ball and then to its fully extended position).
- All collisions in the simulator are modeled as elastic collisions, with various coefficients of restitution.
- Collisions are essentially deterministic, but include small random factors.
- A ball is put into play by a fixed mechanism—the player does not have any control over this initial ball velocity. (A small random factor is introduced into the initial ball velocity.)

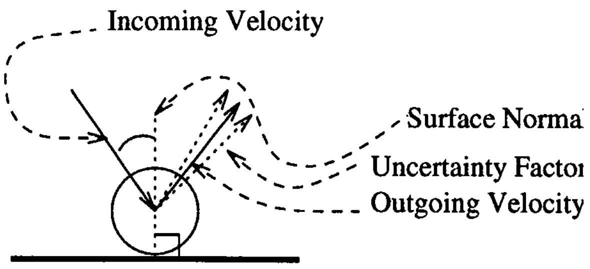

Figure 1: *A simple model of how a collision occurs in the simulator.*

Reinforcement Learning

Reinforcement learning seeks to create control policies by performing repetitive gradient descent in order to match the control policy as closely as possible to the optimal control policy. Dynamic programming methods have been developed to solve optimal control policies (Bellman 1957). However, these methods require a model of the environment *a priori*, and many of them are off-line learning algorithms.

Reinforcement learning (RL) methods, on the other hand, do not require a model of the environment to find an optimal control policy, and are frequently used in and on-line fashion. However, as Singh (Singh 1994) points out, RL methods have not been used to solve very complex tasks on-line. In fact, the most successful RL applications have been neural networks trained to mimic policies developed by RL agents (Tesauro (Tesauro 1993) and Boyan's (Boyan 1992) Backgammon-players are examples of this scheme). This is partly due to the conception that RL methods are very slow.

Unfortunately, because of RL's independence from environment models, they are often stigmatized as *weak* methods (i.e. they do not require domain knowledge, nor do they adapt well to including domain knowledge). This has been an area of great interest lately, and I will demonstrate some effective ways of incorporating domain knowledge into the pinball problem.

Further, RL methods have long been believed to scale poorly to tasks which are *composite* (i.e. tasks which require the completion of several smaller, atomic tasks). Due to the fact that these methods have adapted poorly to other tasks in the past, I have chosen to implement my own composite learning architecture. This will be discussed in more detail in Section .

The TD-Learning Paradigm

A useful paradigm to use to illustrate Reinforcement Learning and Q-Learning was invented by Sutton (Sutton 1988). This method is called Temporal Difference (TD or $TD(\lambda)$) learning. The goal of any system in the TD-learning paradigm is to take a series of observations (usually of a MDT), and produce predictive estimates of the outcome based on these observations.

Consider an agent which makes the following observations:

$$x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_{T-1} \rightarrow x_T,$$

culminating in some scalar reward, Z. The goal of the system is to produce a prediction function $P(x_t)$ which is updated at each observation.

In $TD(\lambda)$ systems, λ is used for exponential weighting of predictions of observations within k steps of the most recent action, according to the rule:

$$\Delta w_t = \alpha(P_{t+1} - P_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w P_k,$$

where $0 \leq \lambda \leq 1$. It should be pointed out that when $\lambda = 1$, the update rule will associate each observation's predictive value with the final scalar outcome. When $\lambda = 0$, only the λ^0 term contributes to the sum, giving the Q-Learning update rule:

$$\Delta w_t = \alpha(P_{t+1} - P_t) \nabla_w P_t$$

The goal of these TD methods is to produce predictive functions which preserve the temporal nature of the problem. They do this by "exponential weighting with recency" i.e. observations which are temporally close to the current state affect the adjustment of the current state's prediction more than other observations (Boyan 1992), (Sutton 1988).

Compositional Q-Learning

Singh (Singh 1994) proposed a method for learning complex, composite and/or sequential tasks within the Q-Learning framework. His system does this successfully by combining maps of Q-values for each of the sub-tasks with a *learning* stochastic switch which learns the map of Q-values to associate with the current state of the composite and/or sequential task, and a bias module that compensates for differences between the Q-values for the current state space location and the desired Q-values for the composite and/or sequential task (thus allowing for transfer of learned information across tasks). In most cases, the stochastic switch and bias module are implemented as learning functions.

Singh's methods were an expansion over the previous concept of competing domain experts, described by Jacobs (Jacobs 1990), which only included a purely stochastic switch (rather than a learning one) and maps of Q-values for each sub-task of the composite or sequential task. I chose to only use Singh's notion of the learning stochastic switch, to allow the maps of Q-values to compete to learn the sub-task knowledge.

The results of these methods are discussed in detail in the following section.

```
$Q \leftarrow a$ set of values for the action-value function (e.g., all zeroes).

For each $x \in S$ : $f(x) \leftarrow a$ such that $Q(x,a) = \max_{b \in A} Q(x,b)$.

Repeat forever:
1. $x \leftarrow$ the current state.
2. Select an action $a$ to execute that is usually consistent with $f$ but occasionally an alternate. For example, one might choose to follow $f$ with probability $p$ and choose a random action otherwise.
3. Execute action $a$ and let $y$ be the next state and $r$ be the reward received.
4. Update $Q(x,a)$, the action-value estimate for the state-action pair $(x,a)$:
       $Q(x,a) \leftarrow (1-\alpha)Q(x,a) + \alpha[r + \gamma U(y)]$
   where $U(y) = Q(y, f(y))$.
5. Update the policy $f$:
       $f(x) \leftarrow a$ such that $Q(x,a) = \max_{b \in A} Q(x,b)$.
```

Figure 2: *Pseudo-code for the Q-Learning algorithm, taken from Whitehead et al.*

Experimental Results

Initially, I developed some useful calibration agents:

- **"Do-Nothing"** which never performs any action.
- **"Flail"** which oscillates the flippers whenever the ball enters the area near the flippers.
- **"Cheat"** which is a simple knowledge-engineering method which knows where the obstacle (or obstacles) which score points are located, and uses this knowledge to determine when the ball should be struck with the flipper.

The results for these agents are illustrated in Figure 4. It should be explained that there are various factors for the poor performance of humans in the simulator environment, such as network delays, slow keyboard input under X Windows, and lack of training of the human test subjects.

Two different table configurations were tested in our learning trials. These table layouts are displayed in Figure 3. The first of the two (on the left) is the layout used in the "Single Obstacle Results" section and the second of the two (on the right) is used in the "Composite Task Results" sections. Both tables are arranged so that the scoring obstacles are surrounded by obstacles which prevent agents or players from scoring points inadvertently. This has been shown to allow for simpler

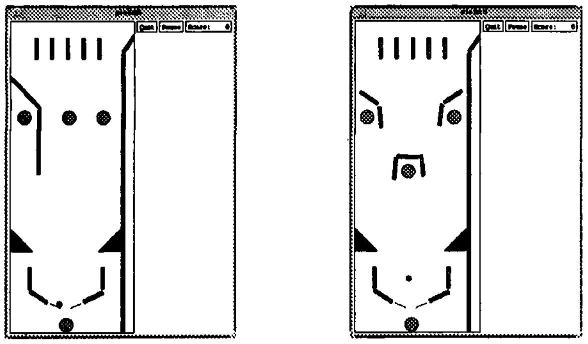
Figure 3: *The pinball simulator table layouts used for the tests described in this subsection.*

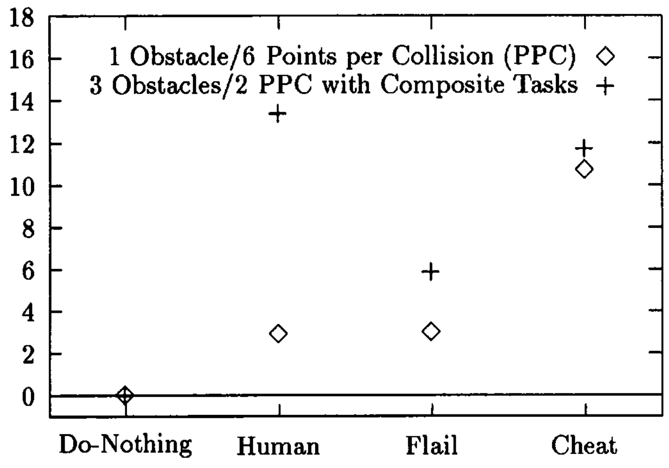

Figure 4: *Mean scores for calibration techniques and human performance.*

agent design, as the agent does not have to discern between intentional and inadvertent scoring increases.

Furthermore, consistency in the results has been ensured by making the scoring obstacle in the "Single Obstacle Results" score six points, while each of the three scoring obstacles in the "Multiple Obstacle Results" scores two points.

Single Obstacle Results

Standard Reinforcement Learning Agent code has been developed in C++ which has been linked into the simulator. Thus far, the agent has been limited to acting only by flipping both flippers simultaneously, and always returning them instantaneously to the "resting" position. For this reason, "catching" the ball with the flippers is not a possible action for the agent.

The pseudo-code for our reinforcement learning agent is shown in Figure 2. I have found that discretizing the state space into a region 50 by 25 cells in the x and y positions, and 50 by 25 cells for the velocity components (the V_x and V_y directions) provides a

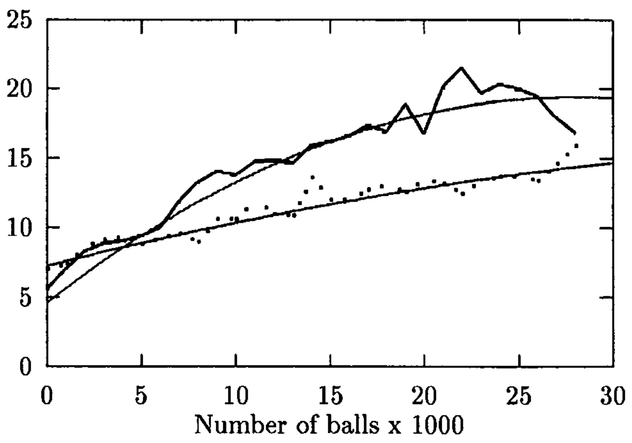

Figure 5: *Results for directed RL versus standard RL over 30000 trials. The directed RL plot is dark, and the standard RL plot is dotted.*

reasonable trade-off between optimality of the obtained control policy and the time that the agent requires (in terms of number of trials) to converge on that policy. I have also discretized and mapped Q-values only for a region near the flippers corresponding to where the agent has control over the ball.

The results of a Q-Learning agent described above in this environment are shown in Figure 5. It appears that the agent quickly achieves a the best observed policy, then oscillates around it over the remainder of the trials.

Incorporating Domain Knowledge Into RL Knowledge Engineering provides quick and consistent results, while RL possibly provides steadily improving results. However, it is never immediately apparent whether the RL agent is producing a control policy which will converge to some acceptable performance level. We desire the exploration of the RL agent to be non-random and *directed* by a Knowledge Engineering method. Interestingly, this approach did not produce any noticeable advantage over the previous RL agent over the first few thousand trials. However, after that time, the directed RL technique showed significantly faster improvement. These results are shown in Figure 5.

Composite Task Results

Next a composite task was created by allowing three obstacles to score two points each, and giving a score bonus of one-hundred points for striking each of the three obstacles. The straight RL technique was applied to this composite task table. These results are shown in Figure 6 as the lower of the two histograms.

Next, the learning stochastic switching agent was applied to the composite goal table configuration. These results are illustrated in Figure 6 as the upper of the two

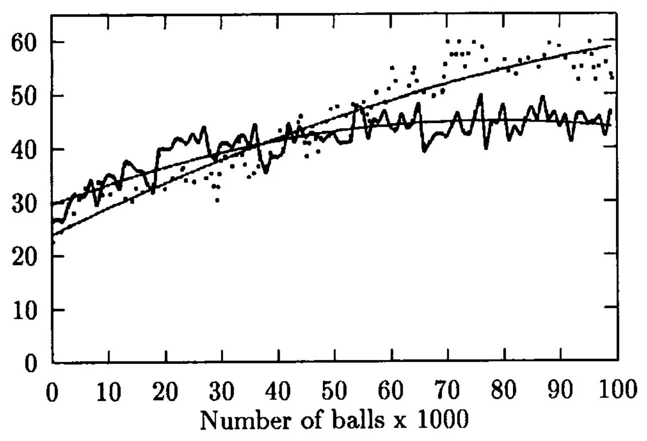

Figure 6: *Results for straight and composite reinforcement learning in a composite environment.*

histograms. The learning stochastic switching agent simply learns to associate the state of the composite task, and the desired sub-task to be performed, and the success (or failure) of each of each of the Q-maps in performing that task. This procedure (in theory) allows competition between the Q-maps to learn the various sub-tasks of the composite task.

Conclusions

This work has shown that while machine learning in physically complicated, dynamical, continuous (an possibly composite and/or sequential) domains can be difficult, they are easily modeled for *temporal difference* (TD) learning methods through a straightforward mapping into the *Markov Decision Process* (MDP) framework. From within this framework, control policies for these problems can be developed using TD methods. Further, though coding domain knowledge into TD methods was once thought to be a difficult thing to do, the author has shown a simple, straightforward method to code domain knowledge into TD architectures which preserves the elegance and simplicity of these methods.

The problem of using TD methods to find control policies in composite and/or sequential environments was previously thought to require cumbersome additions to the architecture. I have demonstrated an architecture able to solve a composite task with a minimum of overhead to the standard Q-Learning algorithm.

I have attempted to demonstrate that there exist useful, efficient, and simple methods to solve problems of substantial complexity. A recent attempt at a system to play checkers produced a system which consisted of four *gigabytes* of main memory, and *thirty-two three-hundred megahertz* processors (Schaeffer *et al.* 1996). While gargantuans like this are becoming commonplace in the AI community, they are a far, far cry from Samuel's IBM 704 of 1959 (which arguably played to roughly the same level: both systems beat world-champions of their day) (Samuel 1963). While it would be nice if the AI community could concentrate on finding the simplest solution to complex problems, it is unlikely that the community would ever unite behind one goal, no matter how sensible it seems.

References

Bellman, R. 1957. *Dynamic Programming.* Princeton, N.J.: Princeton University Press.

Boyan, J. A. 1992. Modular neural networks for learning context-dependent game strategies. Master's thesis, Computer Speech and Language Processing, Cambridge University.

Christiansen, A. D.; Mason, M. T.; and Mitchell, T. M. 1991. Learning reliable manipulation strategies without initial physical models. *Robotics and Autonomous Systems* 8:7–18.

Hougsen, D.; Fischer, J.; and Johnam, D. 1994. A neural network pole-balancer that learns and operates on a real robot in real time. In *Proceedings of the MLC-COLT Workshop on Robot Learning,* 73–80.

Jacobs, R. A. 1990. *Task decomposition through competition in a modular connectionist architecture.* Ph.D. Dissertation, COINS Department University of Massachussets.

Moriarty, D. E., and Miikkulainen, R. 1996. Efficient reinforcement learning through symbiotic evolution. *Machine Learning* 22:11–32.

Samuel, A. L. 1963. Some studies in machine learning using the game of checkers. In Feigenbaum, E. A., and Feldman, J., eds., *Computers and Thought.* McGraw-Hill. 71–105. Originally published in *The IBM Journal of Research and Development,* July, 1959 3:211–229.

Schaeffer, J.; Lake, R.; Lu, P.; and Bryant, M. 1996. Chinook: The World Man-Machine Checkers Champion. *AI Magazine* 17(1):21–29.

Singh, S. P. 1994. *Learning to solve Markovian decision processes.* Ph.D. Dissertation, University of Massechussetts.

Sutton, R. S. 1988. Learning to predict by the methods of temporal differences. *Machine Learning* 3:9–44.

Tesauro, G. 1993. TD-Gammon, A Self-Teaching Backgammon Program, Achieves Master-Level Play. *Neural Computation.*