Heuristic Speed-Ups for Learning in Complex Stochastic Environments

Christian J. Darken
MOVES Institute
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943

Abstract

We describe a novel methodology by which a software agent can learn to predict future events in complex stochastic environments together with an important heuristic-based acceleration technique for computing the prediction. This speed-up enables us to use much more context in our predictions than was previously possible [Darken, 2005]. We present results gathered from a first prototype of our approach.

1 Introduction

A significant challenge for intelligent software agents is making them proactive, i.e. able to understand their environment to the degree that they are able to predict what is likely to happen next and can therefore take appropriate measures. The ability to predict likely next events can in principle be converted into intelligent action selection along the lines suggested by [Sutton and Barto, 1981]. We propose that simple, transparent learning schemes may enable agents to predict the likely course of events. The prediction algorithm has been previously described in [Darken, 2005], but the acceleration techniques and results they enable are new.

In order to explore our hypothesis, we have created a simple game in the RPG (role-playing game) family. We then implemented a sensory interface that passes percepts coded in a first-order logic subset to the agent. The agent then attempts to predict the next percept that it will see. This environment is both stochastic and complex. "Stochastic" implies that future percepts are not a function of the sequence of previous ones. This environment may be considered complex in many senses, beginning with the fact that, although it is a small and simple game as such games go, its state space is very large. More significant, we believe, is the fact that there is no obvious way for the agent to sum up its information about the world in a representation of fixed dimension, i.e. that some aspects of first-order logic are apparently needed in order to accomplish the task. Our impression is that learning algorithms that can succeed in stochastic domains without obvious representations of fixed dimension are of interest for many domains stretching far beyond interactive entertainment applications.

2 Related Work

Anticipation of hostile unit behavior in the context of computer games has previously been addressed in [Laird, 2001], who had the agent apply its own action selection procedure based on the information probably possessed by the hostile unit in order to guess what the hostile would do. In this work, we are attempting to learn to anticipate without hand-coded rules. Further, while hostile unit behavior is one of the things we would like to predict, it is not the only thing.

We have not been successful in finding known algorithms that we can productively compare with our approach. Logical rules, including some types of predictive rules, can be learned by algorithms such as FOIL [Mitchell, 1997]. However, these algorithms assume a deterministic domain. Hidden Markov Models [R. Duda and Stork, 2001] are well suited to stochastic domains, but assume a finite state space, and in practice state spaces that are finite but large are problematic. We are more optimistic about the scaling of variable order Markov models [R. Begleiter and Yona, 2004], but these also assume a finite state space.

After submitting this paper, the reviewers suggested that several recent models may be related to the one presented in this paper. We have not been able to follow up these suggestions in as much detail as we would have liked, but we offer the following preliminary comments. Predictive State Representations [Singh *et al.*, 2003] and Schema Learning [Holmes, 2005] are recent approaches to prediction in stochastic environments. Both are focused on predicting the results of agent actions. We believe both approaches are currently limited to finite state spaces, and we are aware of tests on only very small domains (tens of states). Relational Reinforcement Learning (for example, [Gretton and Thiebaux, 2004]) also considers relational, stochastic domains like the approach described in this work, though it appears to be focused on action selection (as is conventional reinforcement learning) rather than prediction.

3 Benchmark Environment

Our benchmark environment is a simple virtual environment with a text interface modeled after the DikuMUD family of combat oriented MUD's. This family of games is instantly comprehensible to a player of World of Warcraft or Everquest 2, to name two current exemplars, and is arguably a progen-

```
Paperville
Terrified eyes peer from every window
  of this besieged hamlet.
Contents: pitchfork, wand, Conan

get pitchfork

You get the pitchfork.

equip pitchfork

You equip the pitchfork.

w

The Eastern Meadow
All the grass has been trampled into
  the dirt, and tiny footprints are
  everywhere.
Contents: Conan
```

Figure 1: The beginning of a session with the benchmark environment as it appears to a human player named Conan.

itor of these systems. Players of this type of game assume the role of a young adventurer. The goal of the game is to expand the power of one's in-game avatar to the maximum extent possible. This goal is primarily accomplished by slaying the monsters that roam the virtual environment. Slaying monsters results in improvements to the avatar's capabilities through an abstracted model of learning ("experience points") and also though the items ("loot") that the slain monsters drop or guard, which either consist of or may be traded for more powerful combat gear.

The benchmark environment consists of 19 locations, four monsters of three different types, and four different weapon types, of which there may be any number of instantiations. Growth of combat capabilities through experience has not been modeled, therefore, improved capability comes only by acquiring more powerful weapons. The environment as a whole may be conceived of as a discrete event system with a state that consists of the Cartesian product of some number of variables. The system remains in a state indefinitely until an event is received, at which time it may transition to a new state.

The benchmark environment together with networking and multiplayer infrastructure was coded from scratch in Python. The system uses a LambdaMOO-like method dispatch mechanism to determine which game object should process a player action. An unusual feature is the ability to provide output in English text and/or in a first-order logic fragment, as shown in Figures 1 and 2.

4 Perceptual Model

We have implemented text-based interfaces to allow both humans and software agents to interact with the benchmark environment. The human interface consists of English text. We describe the agent interface below.

```
(A 40.6979999542 look)
(+ 40.7079999447 location pitchfork
  Paperville)
(+ 40.7079999447 location wand
  Paperville)
(+ 40.7079999447 location Conan
  Paperville)

get pitchfork

(A 44.6440000534 get pitchfork)
(E 44.6440000534 get Conan pitchfork)
(- 44.6440000534 location pitchfork
  Paperville)
(+ 44.6440000534 location pitchfork
  Conan)

equip pitchfork

(A 47.6080000401 equip pitchfork)
(+ 47.6080000401 equipping Conan
  pitchfork)

w

(A 51.2130000591 w)
(E 51.2130000591 go Conan west)
(- 51.2130000591 location wand
  Paperville)
(- 51.2130000591 location Conan
  Paperville)
(+ 51.2130000591 location Conan
  The_Eastern_Meadow)
```

Figure 2: The beginning of the same session described in Figure 1 with the benchmark environment as it would appear to a software agent named Conan. The first four percept fields are: type, time stamp, percept name. These are followed by the percept arguments, if any.

Perception for software agents in the benchmark environment is modeled as direct access to a subset state variables and system events. The subset of visible events and variables depends upon the location of the agent in the environment, i.e. an agent receives information only about occurrences in his immediate location. The agent's own actions also generate percepts. Thus, four types of percepts are required. 'A' represents agent actions. 'E' represents events. '+' represents the beginning of a time interval in which a variable was sensed to have a particular value. When the variable changes value, or it can no longer be sensed, a '-' percept is received. We form logical atoms from percepts whenever needed by appending the percept type to the percept name to create a predicate (i.e. a percept of type 'E' with name 'location' would correspond to an atom with predicate 'locationE') and taking the remaining elements of the percept as the arguments of the predicate (the time stamp is ignored). At any given time, we define the "sensation" of the agent to be the set of all variables and their values that are currently being sensed.

5 Prediction

After the agent is turned on for the first time, and percepts start to arrive, a percept predictor is constructed on the fly, i.e. the agent learns as it goes along, just like animals do. As each percept is received, the new data is used to enhance ("train") the predictor, and the enhanced predictor is immediately put to use to predict the next percept. Prediction depends upon a few key notions. The first is the notion of a "situation".

Our statistical one-step-ahead percept predictor is a function whose input is the percept sequence up to the time of prediction and whose output is a probability distribution over all percepts that represents the probability that each percept will be the next one in the percept sequence. Of course, all percepts in the percept sequence are not equally useful for prediction. In particular, one might expect that, as a general rule, more recent percepts would be more useful than older ones. On this basis, we discriminate the "relevant" subset of the percept sequence, and ignore the rest. We define a recency threshold T. For predictions at time t, a percept in the percept sequence is relevant if either its time-stamp is in the interval $[t - T, t]$, or it is a '+' type percept whose corresponding '-' percept has not yet been received (this would indicate that the contents of the percept are still actively sensed by the agent). Given the set of relevant percepts, we produce the multiset of relevant atoms (multisets are sets that allow multiple identical members, also known as bags) by stripping off the timestamps and appending the type to the predicate to produce a new predicate whose name reflects the type. We call these relevant atom multisets "situations".

Our predictor function takes the form of a table whose left column contains a specification of a subset of situations and whose right column contains a prescription for generating a predictive distribution over percepts given a situation in the subset. The table contains counters for the number of times each left column and right column distribution element is encountered. We have investigated two different methods of specifying subsets and generating the corresponding predictions.

5.1 Exact Matching

In this technique, each left column entry consists of a single situation. A new situation matches the entry only if it is identical (neglecting the order of the atoms). Each right column entry consists of a distribution of situations. If a new situation matches a left column entry, the predicted percept distribution is the list of atoms in the right-hand column together with probability estimates which are simply the value of the counter for the list member element divided by the value of the counter for the situation in the left column.

As each percept arrives, it is used to train the predictor function as follows. The situation as it was *at the time of the arrival of the last percept* is generated and matched against all entries in the left-hand column of the table. Because of how the table is constructed, it can match at most one. If a match is found, the counter for the entry is incremented. Then the new percept is matched against each element of the predicted percept distribution. If it matches, the counter for that element is incremented. If it fails to match any element of the distribution, it is added as a new element of the distribution with a new counter initialized to one. If the situation matches no left-hand column entry, a new entry is added.

Next, the current situation (including the percept that just arrived) is constructed and matched against the left-hand column entries to generate the predicted distribution for the next percept to arrive. If the situation does not match any entry, there is no prediction, i.e. the situation is completely novel to the agent.

An instructive example to illustrate the algorithm's function can be found in [Darken, 2005].

5.2 Patterns with Variables

The above technique makes predictions that are specific to specific objects in the environment. In environments where an object may be encountered only once and never again, for example, this is not very useful. By replacing references to specific objects by variables, we produce a technique that generalizes across objects. In this technique, left column entries contain variables instead of constants. A new situation matches the entry if there is a one-to-one substitution of the variables to constants in the situation. A one-to-one substitution is a list of bindings for the variables, that has the property that one and only one variable can be bound to one specific constant. The reason for the constraint to one-to-one substitutions is to ensure that each situation matches at most one pattern (left column entry). This restriction is not necessary, but it is convenient. Right column entries can also contain variables in this model. Given a match of a pattern to a situation, the predicted percept distribution is given by applying the substitution to the atoms in the right column distribution. Note that it may be the case that some variables remain in the prediction even after the substitution is applied.

As each percept arrives, it is used to train the predictor function as follows. The situation is generated and matched against all entries in the left-hand column of the table. It can match at most one. If a match is found, the substitution (list of variable-to-constant bindings) is kept, and the counter for the entry is incremented. Then the substitution is applied to each element of the predicted percept distribution, and the

percept is matched against it. If it matches, the counter for that element is incremented. If it fails to match any element of the distribution, it is "variablized" by replacing each constant with the corresponding variable from the substitution, and replacing each remaining constant with a new variable, and then added as a new element of the distribution with a new counter initialized to one. If the situation matches no left-hand column entry, a new entry is added, consisting of the situation with each constant replaced by a variable.

Note that one can conceive of interesting schemes that are combinations of the two presented techniques. For example, one might try to predict the next percept with an exact matching model first, but if no prediction was available (or if the prediction was based on too little data), one might revert to a simultaneously developed variable-based predictor. Alternatively, one might design the environment so that percept references to objects were either existentially quantified variables or constants. A hybrid model could be developed which would then produce patterns with variables or constants based on what was present in the percept. This places the burden of deciding how the predictor should behave onto the percept designer.

6 Accelerated Search

Initially we implemented a back-tracking depth-first search to match situations to table entries. Using back-tracking search and progressing linearly through the predictor table proved too slow. We wanted to experiment with higher recency thresholds. But a higher recency threshold corresponds directly to larger situations, and a great deal more time performing backtracking search.

For the exact matching algorithm, it is the case that each situation corresponds to a unique string which is the constituent atoms (taken as lists of strings which are the predicate and constant arguments) put into lexical order. These strings are then placed in a hash table. Now a new situation can be tested against the table by constructing its string and checking the hash table.

For the variable pattern approach, simply sorting the atoms will not work, as they contain variables whose names are not significant. Our approach is to compute an invariant of the situation pattern that does not depend on the names of the variables. For each variable, we construct two lists of predicates, the list of predicates where the variable appears as the first argument and the list of predicates where the variable appears as the second. All of our predicates are binary. Were this not the case, more lists could be used, or the higher degree atoms reduced to a semantically equivalent set of binary atoms. We then put this list of list pairs into lexical order and then hash them into a table. Two situations that are identical up to substituting the names of variables must hash to the same location in the table. Unfortunately, situations that are different in more than just variable names can nonetheless hash to the same location, so a backtracking search must be performed on each situation in the hash cell to determine whether the match is genuine or not. Still, hash collisions occur relatively rarely, and this approach is very much faster than backtracking search over every row of the predictor table.

6.1 Example

The following example provides proof that the "list of list pairs" invariant, described above, is not sufficient to discriminate all situations that are legitimately different. Consider the following situation description, as might appear as a left column entry in the variable pattern method. Only one predicate, "P", is used.

```
P(?v,?w)
P(?w,?x)
P(?x,?y)
P(?z,?y)
```

Constructing the two lists for each variable as described above yields:

```
?v: [P] []
?w: [P] [P]
?x: [P] [P]
?y: []  [P P]
?z: [P] []
```

Here is a similar, yet different situation description.

```
P(?v,?w)
P(?w,?x)
P(?y,?x)
P(?z,?y)
```

And here is the corresponding list of list pairs.

```
?v: [P] []
?w: [P] [P]
?x: []  [P P]
?y: [P] [P]
?z: [P] []
```

After lexical sort, both cases become:

```
[]  [P P]
[P] []
[P] []
[P] [P]
[P] [P]
```

7 Results

7.1 First Experiment

We created a software agent that takes random actions (one every 0.25 seconds) and connected it to the benchmark environment. Since the action generator is not very intelligent, many actions elicit what are essentially error messages from the environment. We do not consider this a problem. In fact, we would like the agent to learn when an action will be fruitless.

We describe the results of a typical run. For this run, percepts were defined as relevant if they had been received in the last 0.1 seconds or if they were in the agent's current sensation. The agent was allowed to explore the environment for about one and one quarter hours of real time while the learning algorithm ran concurrently. 38519 percepts were received and processed during the run.

The exact matching approach produced 5695 predictors (rows in the table). The approach with variables produced only 952, much fewer, as might be expected.

Numeric results are given in Figures 3 and 4. The average predicted probability of the percepts as a function of time is presented in Figure 7. Note that by the end of the run, both curves are fairly flat. The exact match curve is lower, but increasing faster.

For the approach with variables, the prediction is considered correct if it matches the actual next percept (to within a one-to-one variable substitution). Note that the agent's own actions, being randomly generated, were the most difficult to predict. Neglecting type 'A' percepts, the average predicted probability of all remaining percepts is 66.6 percent for the exact match model and 70.5 percent for the model with variables. This strikes us as reasonably high given the fine-grained nature of the predictions, the simplicity of the algorithm and the high degree of remaining irreducible randomness in the environment caused by random movements of monsters and outcomes of each attempted strike in combat. A significant number of mistakes seemed to be caused by forgetting of important percepts caused by the severe recency threshold used (0.1 sec). We have found that the simple table-based predictive model does not scale well to the recency threshold of multiple seconds that would be seem to be necessary to solve the problem without modifying the agents perception to be more informative.

Detailed analysis of the top five types of errors for each algorithm shows that both algorithms are strongly impacted by the 0.1 sec recency threshold. The worst symptom is that the algorithms are unable to predict combat-related messages accurately because they can not tell that they are in combat. They can not tell that they are in combat because there is nothing in the sensation that indicates ongoing combat, and combat messages are spaced at intervals of one to two seconds.

For the exact matching algorithm, the most common errors stem from the simple fact that, being completely unable to generalize, many situations look completely novel, even at the end of the run. This difference can be clearly seen in the histograms of the last 5000 prediction probabilities presented as Figures 5 and 6. The exact match algorithm has more predictions with probability one than the variable-based algorithm, but it also has more with probability zero, indicating the absence of a match with any table entry.

The variable-based approach scored better that the exact matching algorithm overall. Nonetheless, the lack of predicates for indicating object type in the benchmark environment caused an interesting problem for this approach. For example, this approach was unable to predict the results of attempts to 'get X', and therefore had to hedge its bets between success and an error message. This was no issue for the exact match algorithm, as it could learn that 'get Troll' would provoke an error while 'get sword' would succeed. Note that the addition of a 'portable' predicate, for example, would mitigate this problem.

7.2 Second Experiment

In the second experiment, a fresh run of the agent was performed with the time between actions greatly increased (from

Type	Avg. Probability	Occurrences	Error
A	7.65%	14488	65.5%
E	72.09%	14905	20.3%
+	45.92%	4563	12.1%
-	69.28%	4563	6.9%

Figure 3: Performance summary for exact matching. The average predicated probability over all percepts was 44.43%. Error is the expected fraction of the total number of prediction errors for percepts of the given type.

Type	Avg. Probability	Occurrences	Error
A	7.82%	14488	65.3%
E	66.39%	14905	24.5%
+	65.12%	4563	7.8%
-	89.32%	4563	2.4%

Figure 4: Performance summary for patterns with variables. The average predicted probability over all percepts was 46.94%. Error is the expected fraction of the total number of prediction errors for percepts of the given type.

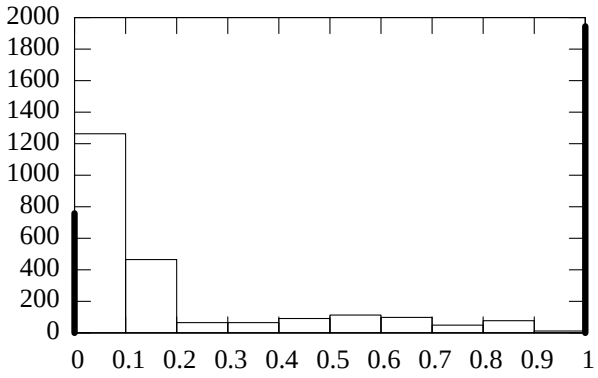

Figure 5: Prediction probability for the last 5000 percepts of the run with constants. The black bars represent the predictions of exactly 0 or 1.

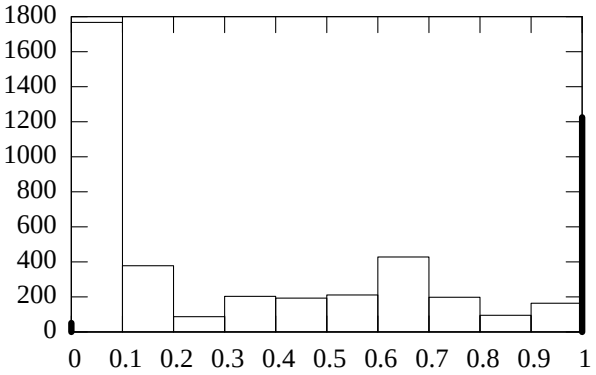

Figure 6: Prediction probability for the last 5000 percepts of the run with variables. The black bars represent the predictions of exactly 0 or 1.

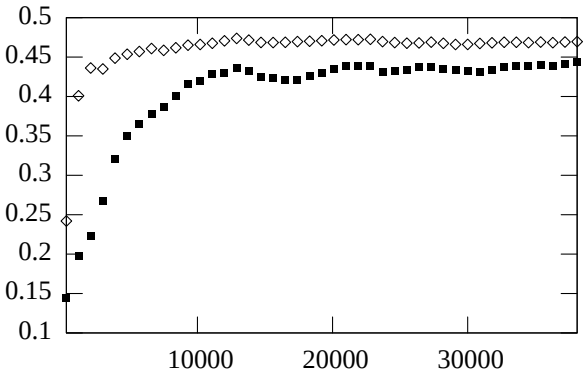

Figure 7: Average prediction probability as a function of the number of percepts received. White diamonds represent the algorithm with variables and black squares the algorithm with constants.

	0.1s	2.1s
Exact Match	62.2%	57.3%
Variable Pattern	64.8%	62.3%

Figure 8: Performance summary on the second experiment on all percepts except 'A' type percepts.

0.25 seconds to 2.5 seconds between successive actions. The reason for the increase was because at 0.25 seconds per action and two seconds per combat round, the agent would attempt up to four actions in between successive combat "blows". The combat messages were thus somewhat "buried". This run was longer than that of the previous experiment. It consisted of 170762 percepts received over 38 hours of real time.

Using the acceleration techniques described above, we tested both 0.1 second and 2.1 second recency thresholds with both the exact match and variable pattern techniques. Results on all percepts excluding 'A' type percepts are presented in Figure 8. As in the first experiment, the variable pattern approach performs better. The additional context provided by the higher recency threshold seems to hurt overall performance rather than helping. Apparently the extra information in the larger context is not enough to overcome the need for more training data. However, these results are very new, and we are still analyzing them in detail.

8 Discussion

A few comments on the structural characteristics of the methods presented in this paper are in order.

One very positive characteristic of them is that there is a clear "audit trail" that can be followed when the agent makes unexpected predictions. I.e. each row in the table can be traced to a specific set of prior experiences that are related to the predictions it makes in an obvious way. Many machine learning techniques do not share this characteristic.

Note that the situations in the left column of the table divide all possible percept sequences into a set of equivalence classes, i.e. many percept sequences can map into a single situation set. To the agent, only the sequence sets specified in the left column of the table matter. It will never be able to discriminate between different percept sequences that map into the same sequence set. The temptation naturally arises to make these sets as differentiated as possible by, for example, increasing the recency threshold or using exact matching instead of patterns with variables. But increasing the fineness of the situation sets is a two-edged sword. While it does indeed make it possible for the agent to discriminate between different percept sequences that it could not differentiate before, it also makes it increasingly rare that the agent visits situations that it knows about. Figures 5, 6, and 7 illustrate this fact.

9 Future Work

Although we have not discussed it previously, note that it is possible to extend the system as described to making predictions about *when* the next percept will be received in addition to what the next percept will be along the lines described in [Kunde and Darken, 2005].

A key direction for further investigation is improved predictive models and systematic exploitation of the predictions. The technique described in this work is very limited in its generalization capabilities. Unlike FOIL, which searches through candidate atoms and includes only the most promising in the model, the current approach takes all atoms that have passed the relevance test. It would be nice to have an approach that could perhaps learn from experience which of the relevant atoms were actually necessary to accurate prediction.

While we take for granted that many special-purpose schemes can be constructed which can improve agent behavior based on the ability to predict future percepts, it seems worth pointing out that one can search over the space of potential courses of action using the predictive model and a quality function to decide which course to adopt. This is a homogeneous and general-purpose method of exploiting prediction very similar in spirit to the model predictive control techniques that are an established part of chemical engineering [Morari and Lee, 1997]. It has been explored within the computer science literature as well [Sutton and Barto, 1981].

10 Acknowledgements

Partial funding for this work was provided by the U.S. Army TRADOC Analysis Center (TRAC) Monterey and the Naval Modeling and Simulation Management Office. The author wishes to thank the anonymous reviewers for very helpful references, comments and advice.

References

[Darken, 2005] C. Darken. Towards learned anticipation in complex stochastic environments. In *Proc. Artificial Intelligence for Interactive Digital Entertainment 2005*, Marina Del Rey, CA, 2005.

[Gretton and Thiebaux, 2004] C. Gretton and S. Thiebaux. Exploiting first-order regression in inductive policy selection - extended abstract. In *Proc. of the ICML '04 Workshop on Relational Reinforcement Learning*, 2004.

[Holmes, 2005] M. Holmes. Schema learning: Experience-based construction of predictive action models. In *Neural Information Processing Systems*, 2005.

[Kunde and Darken, 2005] D. Kunde and C. Darken. Event prediction for modeling mental simulation in naturalistic decision making. In *Proc. BRIMS 2005*, Universal City, CA, 2005.

[Laird, 2001] J. Laird. It knows what you're going to do: adding anticipation to a quakebot. In Jörg P. Müller, Elisabeth Andre, Sandip Sen, and Claude Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 385–392, Montreal, Canada, 2001. ACM Press.

[Mitchell, 1997] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, Boston, 1997.

[Morari and Lee, 1997] M. Morari and J. Lee. Model predictive control: Past, present and future, 1997.

[R. Begleiter and Yona, 2004] R. El-Yaniv R. Begleiter and G. Yona. On prediction using variable order markov models. *Journal of Artificial Intelligence Research (JAIR)*, 22:385–421, 2004.

[R. Duda and Stork, 2001] P. Hart R. Duda and D. Stork. *Pattern Classification*. John Wiley & Sons, New York, 2001.

[Singh *et al.*, 2003] S. Singh, M. Littman, N. Jong, D. Pardoe, and P. Stone. Learning predictive state representations. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML) 2003*, pages 99—106, 2003.

[Sutton and Barto, 1981] R. Sutton and A. Barto. An adaptive network that constructs and uses an internal model of its world. *Cognition and Brain Theory*, 4(3):217–246, 1981.