Towards Learned Anticipation in Complex Stochastic Environments

Christian J. Darken

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. It is particularly relevant to environments that are constructed specifically so as to be able to support high-performance software agents, such as video games. We present results gathered from a first prototype of our approach. The technique presented may have applications that range beyond improving agent performance, in particular to user modeling in the service of automated game testing.

Introduction

One of the challenges on the long term agenda of AI for interactive entertainment is producing software agents that are proactive, i.e. that understand their environment to the degree that they are able to predict what is likely to happen next and can therefore take appropriate measures. Unlike other software agent construction tasks where the information available to the agent via its virtual sensors is fixed in advance, for a computer game the environment and agent are built together, thus opening up the possibility of adjusting the nature of agent perception to improve the system as a whole. In contexts where agent perception can be adjusted to suit, we propose that very simple, transparent learning schemes may be a practical approach to allow an agent to predict the likely course of events in its environment.

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.

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 & 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.

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 progenitor 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

```
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.

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.

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.

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

```
(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.

'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.

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 timestamp 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.

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.

Example

Consider the following percept sequence:

```
[+ 0.0 loc Ed road]
[+ 1.2 loc Fox1 road]
[E 2.5 go Fox1 east]
[- 2.5 loc Fox1 road]
[+ 5.1 loc Fox2 road]
[E 5.1 go Fox2 east]
[- 6.5 loc Fox2 road]
```

After the last percept is received at time 6.5, assuming the recency threshold $T = 0.1$, the predictive table looks like the following:

```
{}                      1| (loc+ Ed road)   1
----------------------------------------
{[loc+ Ed road]}        2| (loc+ Fox1 road) 1
                         | (loc+ Fox2 road) 1
----------------------------------------
{[loc+ Ed road]          |
 [loc+ Fox1 road]} 1| (go Fox1 east)   1
----------------------------------------
{[loc+ Ed road]          |
 [loc+ Fox1 road]        |
 [goE Fox1 east]}  1| (loc- Fox1 road) 1
----------------------------------------
{[loc+ Ed road]          |
 [loc+ Fox2 road]} 1| (go Fox2 east)   1
----------------------------------------
{[loc+ Ed road]          |
 [loc+ Fox2 road]        |
 [goE Fox2 east]}  1| (loc- Fox2 road) 1
```

The situation at time 6.5 would be

```
{[loc+ Ed road]}
```

which matches the second row of the table. The next percept would be predicted to be '(loc+ Fox1 road)' with probability $1/2$ and '(loc+ Fox2 road)' with probability $1/2$. After training the predictor on the new percept, a new row

```
{[loc+ Ed road]     |
 [loc+ Fox2 road]   |
 [goE Fox2 east]} 1| (loc- Fox2 road) 1
```

would be added to the table.

Patterns with Variables

The exact matching technique makes predictions that are
specific to specific objects in the environment. In environ-
ments where an object may be encountered only once and
never again, for example, this is not very useful. By replac-
ing 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 re-
striction 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 distri-
bution 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 con-
stant 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.

Experimental Results

We created a software agent that takes random actions (one
every 0.25 seconds) and connected it to the benchmark en-
vironment. 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, per-
cepts were defined as relevant if they had been received in
the last 0.1 seconds or if they were in the agent's current sen-
sation. 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 consid-
ered 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 pre-
dicted 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 ran-
domness in the environment caused by random movements
of monsters and outcomes of each attempted strike in com-
bat. A significant number of mistakes seemed to be caused
by forgetting of important percepts caused by the severe re-
cency threshold used (0.1 sec). We have found that the sim-
ple 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 al-
gorithm 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 rounds are spaced at intervals of about two seconds.

For the exact matching algorithm, the most common er-
rors 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 pred-
icates for indicating object type in the benchmark environ-
ment 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 be-
tween 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.

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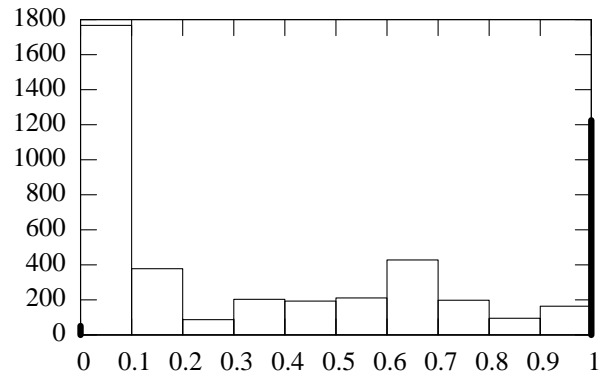
Type	Avg. Probability	Occurrences	Error
A	7.82%	14488	65.3%
E	66.39%	14905	24.5%
+	65.12%	4563	7.8%
-	89.32%	4563	2.4%

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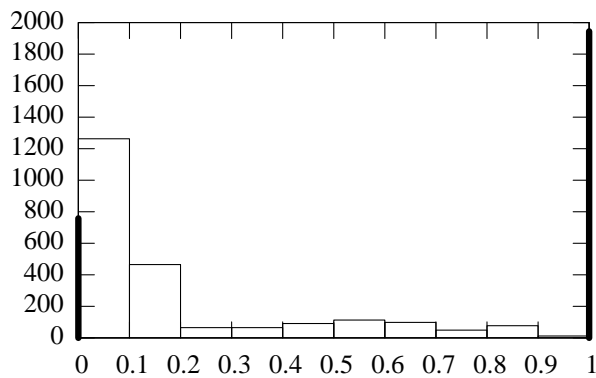

Figure 5: Prediction probability for the last 5000 percepts of the run with constants. The black bars represent the predictions of exactly 0 or 1.
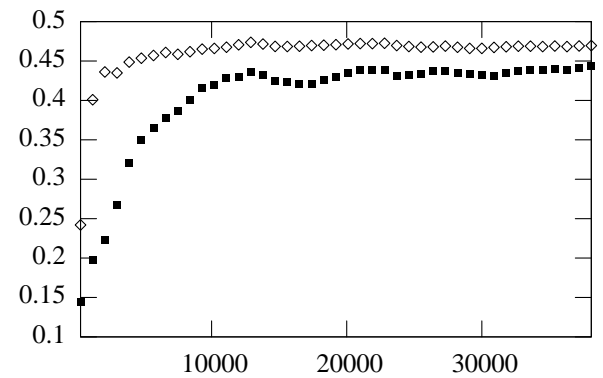

Figure 7: Average prediction probability as a function of the number of percepts received. White diamonds represent the algorithm with variables and black squares the algorithm with constants.

Discussion

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

One very positive characteristic of these techniques 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.

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.

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 membership in 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. Likewise, one could adapt the methods we present to be sensitive to percept order. 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.

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 & Darken 2005).

Two key directions for further investigation are 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 behav-

ior 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 & Lee 1997). It has been explored within the computer science literature as well (Sutton & Barto 1981).

Finally, we believe that the predictive model may have potential for improving the automated testing of computer games. It is already the case that automated "button mashing", i.e. random control inputs, are used to test the stability of computer and video games. The model described could be applied to capture, if crudely, typical user behavior as a function of game state. The model could then be simulated Monte Carlo style to generate large numbers of random tests that are more focused on the types of behavior that users actually have.

References

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

Laird, J. 2001. It knows what you're going to do: adding anticipation to a quakebot. In Müller, J. P.; Andre, E.; Sen, S.; and Frasson, C., eds., *Proceedings of the Fifth International Conference on Autonomous Agents*, 385–392. Montreal, Canada: ACM Press.

Mitchell, T. M. 1997. *Machine Learning*. Boston: McGraw-Hill.

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

R. Duda, P. H., and Stork, D. 2001. *Pattern Classification*. New York: John Wiley & Sons.

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