On Behavior Classification in Adversarial Environments

Patrick Riley[1] and Manuela Veloso[1,2]

[1] {pfr,veloso}@cs.cmu.edu, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213
[2] Currently a visiting professor at Massachusetts Institute of Technology, Boston, MA

Abstract. In order for robotic systems to be successful in domains with other agents possibly interfering with the accomplishing of goals, the agents must be able to adapt to the opponents' behavior. The more quickly the agents can respond to a new situation, the better they will perform. We present an approach to doing adaptation which relies on classification of the current adversary into predefined adversary classes. For feature extraction, we present a windowing technique to abstract useful but not overly complicated features. The feature extraction and classification steps are fully implemented in the domain of simulated robotic soccer, and experimental results are presented.

Keywords: group behaviors, adversary classification, and adaptation

1 Introduction

In order for robotic systems to be successful in complex domains, they must be able to adapt to the environment, especially to the current behavior of other agents. This adaptation should occur at all levels of strategy, from individual reactive behaviors to team strategy. In highly complex domains with no clear optimal policy, agents which can adapt more quickly and effectively will perform better.

We consider "complex domains" to be those with enormous, or possibly continuous, state and action spaces. Further, we consider dynamic environments, i.e. those with limited time to reason before each agent must choose an action. We are further interested in domains with noisy perceptions and actions.

Reinforcement learning is a common technique for having an agent automatically discover its optimal policy. However, as discussed in [3], reinforcement learning often results in poor performance in the types of domains in which we are interested. Further, reinforcement learning will generally require trying every possible strategy of the agent many times. We would like to be able to adapt much faster than this.

Systems such as Carmel and Markovitch's [1] succesfully adapt by building up models of the opponents (in this case finite automata) and then computing the best strategy given that opponent. However, the environment they

consider is a simple two-person repeated matrix game. It is not clear how this sort of technique will scale to the sort of complex environments mentioned above, and we propose that other techniques will be needed for these environments.

Another approach by Sen and Sekaran[8] allows agents to learn to cooperate with agents who are willing to cooperate, and simultaneously avoid exploitation from hostile agents. However, in their environment, it is clear what action to take when another agent is hostile. In more complex domains, there may be many possible strategies to deal with opposing agents.

The system we propose is quite different. The hypothesis underlying this research is that effective adaptation can be done by analogy to previous opponents. By noting similarities in "style of behavior," the agent team can employ the most effective strategy against that behavior.

This is partially inspired by human behavior. Human beings are often quite good at the sort of adaptation in which we are interested. Good sports players will carefully watch how their opponents play and change strategy based on their opponents' actions. We would like to better understand this ability and be able to map it onto autonomous agents.

The long-term goal of this research has several aspects:

1. Abstracting useful features from the raw sensor data and world state available to the agents. An approach to feature extraction in complex domains is introduced in Section 3
2. Constructing "adversary classes" from the information in the features. The adversary classes should embody strategic information about how this type of adversary behaves. Closely related to this is the work Ramoni, *et al.* have done on unsupervised Bayesian clustering of world dynamics[6]. Using this technique, clusters are formed which describe general features about what is going on in the world. For example, they describe one cluster as cases "in which the blue team won despite a large mass deficit." In this research, however, creating adversary classes is done by averaging over many observations with *a priori* distinctions (see Section 4.2).
3. Matching what is observed of a current adversary to the adversary classes. This gives a "classification" of the current adversary into the predefined classes. This step is addressed in Section 4.
4. Affecting an effective change in behavior based on the classification. There should be some mapping from adversary classes onto strategies that our agents' may use. This final change in behavior is the goal of the classification process. However, the other aspects may be analyzed in the absence of this change in strategy, so this step is not addressed here.

This research focuses mostly on abstracting features and classifying the adversary, numbers 1 and 3 above.

2 The Test Domain and Software

The test domain for this research is simulated robotic soccer, specifically, the Soccer Server System[4] as used in the Robot World Cup Initiative, an international AI and robotics research initiative. The Soccer Server System is a server-client system which simulates soccer in a discrete fashion. Clients communicate using a standard network protocol with well-defined actions. The server keeps track of the current state of the world, executes the actions which the clients request, and periodically sends each agent noisy, incomplete information about the world. Agents receive noisy information about the direction and distance of objects on the field (the ball, players, goals, etc.); information is provided only for objects in the field of vision of the agent. The Soccer Server System also allows an agent to connect as a "coach" client who has a global view of the world, but whose only action is to send short messages to the players while the ball is out of play. This is ideal for the sort of classification described here. The coach can classify based on global information, then communicate that information to the more active players. The classification thus provides a very limited but effective language for strategy communication. We have previously worked on classification without the coach agent by fusing the partial information of the agents in order to classify the adversary[7].

Each of the agents has an opportunity to act 10 times a second. Each of these action opportunities is known as a "cycle." Visual information is sent 6 or 7 times per second. Over a standard 10 minute game, this gives 6000 action opportunities and 4000 receipts of visual information. It would be impractical to attempt to classify the adversary every time new visual information arrives. Also, intuitively, some features which capture an adversary's performance are time-dependent. Therefore, we need to consider multiple cycles at once when classifying the adversary.

Further, learning from just the raw world data would also be infeasible. 10^{17} is an estimate for the size of the state space of the Soccer Server. We cannot expect to adapt as quickly as we would like without providing some abstractions to our learning algorithm.

The algorithms described here are implemented based on the successful simulated robotic soccer team CMUnited-98[9] which was the champion of RoboCup-98. All decision tree learning was done using C4.5[5].

3 Feature Extraction

When trying to learn in complex domains, some features must usually be abstracted from the raw data. It would be impractical and not very revealing to expect a learning algorithm to effectively decode all sensor information if a good model for doing so is already known. Extracting relevant features from sensor data can make learning feasible.

```
time := 0;
do {
    W := get_new_world_state;
    (D, I) := f(D, I, W);
    time += 1;
  }until (time == epoch_length);
return D;
```

Fig. 1. Pseudo-code for the use of an observation

Often, this step of feature extraction is taken somewhat lightly. However, we are working in a complex domain and trying to learn high-level strategy, so feature extraction is both more critical and complicated.

3.1 Windowing

To reduce the complexity of our features, we remove time sequencing from the feature's data. Removing time sequencing from data has been effective in other complex domains such as text classification with the "bag-of-words" approach, and music analysis and generation[10]. We refer to features of this type as "observations." An observation occurs over a fixed length of time, known as the window length.

Formally, an observation A is a 3-tuple $\langle D, I, f \rangle$. D is the data that results from the observation. I is the internal state used in the updating function $f : D \times I \times W \rightarrow D \times I$, where W represents the state of the world. The internal state I is used to capture time dependent features (see below). Every discrete cycle, the function f is called to update D and I. Pseudo-code indicating how such an observation would be used is given in Fig. 1.

Further, the definition of an observation is slightly more strict than given above. The updating function f is an indicator for some event. If that event occurs, the fact that it occurred is recorded into the data structure D; the time of the occurrence is *not* recorded. For example, say E_1 and E_2 are events of the type for which f is looking. If E_1 occurs at time 25 and E_2 at time 50, the resulting data structure D would be identical to the data structure produced if E_2 occurred at time 1 and E_1 at time 2.

Note that the observation keeps some internal state I. This allows f to detect features that involve sequences of events. For example, an observation might be for an event E which consists of an event of type E_1 followed immediately by an event of type E_2. Passing and shooting in simulated robotic soccer require this sort of description. While the internal state I may record time information, the data structure D may not. At the end of the time window, only the data structure D is returned; the internal state I is discarded.

3.2 The Observations Used

RectGrid The data structures for the observations used here are built on a data structure called a RectGrid. A RectGrid is a division of the field into

small geographic regions. The RectGrid is designed to record the occurrence of certain events in specific geographic areas. An observation is recorded by adding a count to some of these geographic areas. The counts in the geographic areas represent how many times the specified type of event (like a pass or shot) occurred in that region.

In all the experiments performed here, the RectGrid dimensions used were 8×15, with the longer dimension in the longer dimension of the field.[1]

Detailed Observations The following observations were used in all of the classification experiments.

- **Ball Position**: Every cycle, the ball's position is stored in a RectGrid
- **Opponent Position**: Every cycle, each opponent's position is stored in a RectGrid, one RectGrid per opponent number
- **Opponent Passing**: This observation records all opponent passes in a RectGrid. All bins that a pass goes through are marked. A pass is defined as follows: Some opponent is in control of the ball at some cycle. Within 50 cycles, a different opponent controls the ball, with no significantly long control by another player in the middle.
- **Opponent Dribbling**: This observation records all opponent dribbles in a RectGrid. All bins which an opponent goes through while dribbling are marked. A dribble is defined as follows: Some opponent is in control of the ball continuously (i.e. no more than 4 cycles in a row where it does not control the ball) and his position changes by at least 3m.
- **Opponent Shooting**: This observation records in a RectGrid from where all shots are taken by the opponents.

An Example To further demonstrate the structure and use of observations, we will examine the Ball Position observation more closely. The data structure D is just a RectGrid as described in Section 3.2. There is no internal state I. The updating function f takes the current RectGrid D, adds a count to whichever geographic region the ball is currently in, and then returns the updated RectGrid D'.

A result of the ball position observation is pictured in Fig. 2. The field is divided into 6 rectangular regions. As the ball moves from left to right over 10 cycles (part (a)), the agent adds counts to the geographic regions of the RectGrid, resulting in the counts shown in part (b).

4 Construction and Classification

The goal here is to use the observations to *classify* the current adversary. That is, given a predefined set of adversary types, $C_1, \ldots, C_n$, determine into

[1] A variety of numerical values are provided for the observations described here in order to allow others to reproduce these results. Further, all software developed for this research is available at http://www.cs.cmu.edu/~pfr/adv_class.html

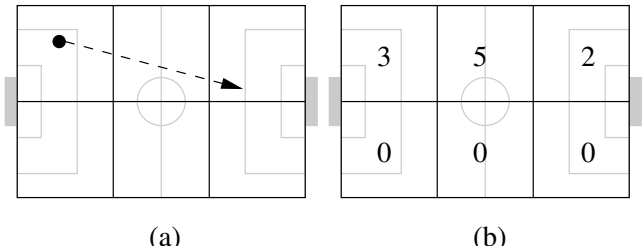


Fig. 2. An Example of the Ball Position Observation

which class the current adversary fits. On a conceptual level, these adversary classes should represent some strategic qualities about the adversary, which can then be used to improve the agent team's performance. Also, these classes provide a compact way for the agents to describe this adversary to each other.

In this framework, an adversary class is a set of target configurations for observations. For example, one adversary class may consist partly of a target configuration for "Opponent Dribbling" which indicates that the opponent only dribbles on the sides of the midfield.

4.1 Experimental Setup

The issue of creating an effective strategy change was not addressed in this research. However, it is important to verify that some useful information can be obtained from this sort of windowing approach to feature extraction.

As discussed in Section 2, we used a "coach" agent who has a global view, but whose actions are limited to communication. In this version of the software, there is no strategic change based on the classification. Consequently, the coach is entirely passive. This allows the coach to look at a logfile of a previously played game exactly as if the game was currently in progress.

Hence, the logfiles from all of the games at RoboCup-98 and RoboCup-99 were used as the data-sets. The algorithms and data structures were developed while analyzing the RoboCup-98 logfiles, and once completed, the analysis of the RoboCup-99 logfiles was done.

4.2 Generating Adversary Classes

In each of the data sets, one adversary class was created for each team. The class was created by observing the team on the data set. There is an underlying assumption that the strategy of a team does not vary significantly over the course of the competition. There are several known exceptions to this, especially in RoboCup-99[2]. However, most teams did not make significant

[2] Specifically, the team HCIII from Linköping University has a nice strategy editor which they used to design/select strategies for each game. Also, some teams made significant revisions to their software during the competition after seeing how opponents played

changes during the competition. The instances of the observation were then averaged together. A RectGrid is averaged by averaging all of the counts for each geographic region.

After creating these adversary classes, the goal was to correctly identify which teams were playing based on the observations. In RoboCup-98, there were 34 teams, making the accuracy of random guessing to be 2.9%. In RoboCup-99, there were 37 teams, making the accuracy of random guessing to be 2.7%. The window length varied between 25 cycles and 1000 cycles. With 6000 cycles in a regular length 10-minute game, this gives between 6 and 240 windows in a game. The difference in the amounts of data at these extremes can be seen in the results (Section 5).

In order to compare observations of the current adversary to the adversary classes, there must be a notion of similarity between different values for an observation. Since all of the observations are built on the RectGrid data structure, we need a similarity metric for a RectGrid. We previously developed such a metric to take into account the spatial locality of differences in observations[7].

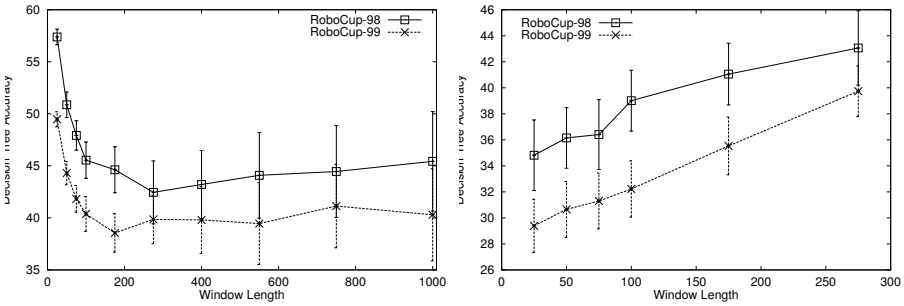
4.3 Classification

Our first approach to classification was a simple nearest neighbor approach. This performed quite poorly, not doing signifigantly better than random. One problem we perceived with the nearest-neighbor approach was the small amount of information available for making the classification decision. In generating a similarity (distance) between instances of observations and an adversary class, you have a single real number in $[0, 1]$ for each type of observation (5 types in this case).

This led us to two changes: using decision trees to deal with the large amount of data, and disassociating the target configurations with a specific adversary class. Consequently, even though the target configurations were generated for a specific team, classification would occur based on all similarities to all target configurations. All of the similarities are fed into the decision tree learning as features on which to split.

Therefore, when distinguishing between class C_1 and C_2, the best feature to distinguish on may be the similarity to a target configuration from class C_3. Why should the similarity to a target configuration in C_3 affect whether the adversary is of type C_1 or C_2? Consider the classes to be two rectangular regions on the plane (not necessarily axis-parallel), with their centers on the x-axis. Just measuring the similarity between the centers of the two regions allows you to split the plane only with lines parallel to the y-axis. Using other centers can allow you the split the plane in other directions.

Decision trees also allow us to capture disjunctive concepts. For example, a team may have several modes of behavior, like "defensive", "offensive" or "goal kicks". Comparing to a single target configuration does not allow us to effectively capture these different modes. Having a greater variety of



(a) All Data (b) Normalized Amount of Data

Fig. 3. Classification Accuracy (note that accuracy of random guessing $< 3\%$)

configurations allows us to express that adversary type A may have different distinct modes of behavior.

We are also interested in how varying the window length affects classification accuracy. Too short of a window presumably does not allow the agent time to capture the occurrence of important events. Using too long of a window negatively affects the goal of quick adaptation to an adversary, as well as causing unnecessary smoothing of the features.

5 Results

In the experiments, all of the data was used to generate the target configurations. Then, each time window was observed again, and the similarities to all of the target configurations were recorded. 90% of the data was then used for training of the decision tree and 10% was reserved for testing. All accuracy rates reported are for the testing data. In order to better ascertain the accuracy, 50 decision trees were generated for each window length, using different randomly selected training and test data sets.

The results are shown in Fig. 3(a). The results are somewhat surprising: the smallest windows have the highest accuracy rates. It was expected that too small a window would not allow many important features to be captured.

Notice that as window length decreases, the amount of data available for decision tree training increases like $1/x$. Increasing the data available for training often greatly increases accuracy in decision tree training.

Therefore, we trained decision trees for all window lengths less than 275 with reduced data and testing sets. That is, the data sets for all of these window lengths were reduced to the size of the data set for a window length of 275. The data sets were reduced uniformly in the sense that if the data needed to be reduced by a factor of n, 1 out of every n consecutive windows was kept.

The results for these trials is shown in Fig. 3(b). This error curve is a bit more as expected, with the smaller epochs performing slightly worse.

6 Conclusions and Future Work

Accuracies around 40% may not seem all that successful, but there are several points to note:

- There are 34-37 classes to be distinguished, and the accuracy of guessing is less than 3%. Therefore, our accuracy is on order of magnitude better than guessing.
- The data sets are "real" data sets. The logfiles from the RoboCup competitions are a result of many people working to produce interesting teams, and reflect a great variety of techniques and strategies.
- One class was made for each named team in the data sets. If two teams play similarly, we are still requiring that the classification methods distinguish them. It is likely that a more refined method of creating adversary classes would have higher classification accuracy.

The difference in the accuracy trends on epoch length points out an interesting tradeoff in this windowing approach. The smaller the windows used, the more training data one will get, probably improving accuracy. However, a small window also means that some necessarily time consuming events will rarely be observed because they will not occur perfectly inside of a window. Also, a small window is likely to give higher random fluctuation in behaviors.

Given the importance of choosing the correct window length, the ability to dynamically change the window length would probably improve the system. Alternatively, running several different window lengths in parallel and using a meta-learner to combine the classifications is also an interesting future direction.

The main contributions of this research are:

- A windowing approach to extracting useful features in complex domains.
- A successful classification method using decision tree learning.
- A demonstration that this approach to classification is effective in a particular complex, dynamic domain.

As indicated in the introduction, this research addresses only part of the whole problem of adversary adaptation. For example, in this research, an adversary class was generated for each named team of agents. However, creation of adversary classes should be based on playing style and not solely on *a priori* team distinctions, suggesting a clustering approach. Further, understanding how to map adversary classes to strategy changes requires a greater understanding of the different strategies our own agent team may employ.

The use of Hidden Markov Models in the updating function f of observations also seems promising in creating more complex and reliable observations. Hidden Markov Models have been used successfully in recognizing simple behaviors in robotic soccer[2].

While the use of decision trees to produce classifications gave fairly good results, having a classification method which indicates how *well* this adversary

matches to all classes would be helpful. Other statistically based machine learning algorithms could produce this kind of information. That way, truly novel adversaries would be recognized, as would adversaries who have features from several adversary classes.

We have described a general method for adapting to adversaries in complex domains. We further explored some pieces of this approach, using decision trees in the final classification step. We also presented experimental results for this system in the domain of simulated robotic soccer. We hope that this approach to adversary adaptation will further developed, tested, and improved.

References

1. D. Carmel and S. Markovitch. Opponent modeling in multi-agent systems. In G. Weiss and S. Sen, editors, *Adaptation and Learning in Multi-Agent Systems*, Lecture Notes in Artificial Intelligence, pages 40–52. Springer, 1995.
2. K. Han and M. Veloso. Automated robot behavior recognition applied to robotic soccer. In *Proceedings of IJCAI-99 Workshop on Team Behaviors and Plan Recognition*, 1999.
3. M. J. Matatić. Learning in multi-robot systems. In G. Weiss and S. Sen, editors, *Adaptation and Learning in Multi-Agent Systems*, Lecture Notes in Artificial Intelligence, pages 206–217. IJCAI'95 Workshop, Springer, 1995.
4. I. Noda, H. Matsubara, K. Hiraki, and I. Frank. Soccer server: A tool for research on multiagent systems. *Applied Artificial Intelligence*, 12:233–250, 1998.
5. J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, CA, 1993.
6. M. Ramoni, P. Sebastiani, P. Cohen, J. Warwick, and J. Davis. Bayesian clustering by dynamics. Technical Report KMi-TR-78, Knowledge Media Institute, The Open University, United Kindgdom MK7 6AA, February 1999.
7. P. Riley. Classifying adversarial behaviors in a dynamic, inaccessible, multi-agent environment. Technical Report CMU-CS-99-175, Carnegie Mellon University, 1999.
8. S. Sen and M. Sekaran. Using reciprocity to adapt to others. In G. Weiss and S. Sen, editors, *Adaptation and Learning in Multi-Agent Systems*, Lecture Notes in Artificial Intelligence, pages 206–217. IJCAI'95 Workshop, Springer, 1995.
9. P. Stone, M. Veloso, and P. Riley. The CMUnited-98 champion simulator team. In M. Asada and H. Kitano, editors, *RoboCup-98: Robot Soccer World Cup II*, pages 61–76. Springer Verlag, Berlin, 1999.
10. B. Thom. Learning models for interactive melodic improvisation. In *International Conference on Computer Music*, China, October 1999.