Opponent Modeling in Interesting Adversarial Environments

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Brett Jason Borghetti

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Maria Gini, Advisor

August 2008

Acknowledgments

I would like the thank the many computer science graduate students who helped me conceive of, formalize, and survive many aspects of this work. In particular, I thank Tim Woods for his willingness to discuss Game Theory, adversarial thinking and countless other computer science topics; Steven Damer for his help with developing a method for computing the equilibrium solution to the simultaneous move strategy game and encouragement in the development of other adversarial thinking; Mohamed Elidrisi for his willingness to discuss the meaning of deception within agent gameplay; Steve Jensen for introducing me to the many uses of Information Theory. To the many others that have contributed to my well being during this endeavor through companionship and conversation, I give thanks.

I thank my advisor, Professor Maria Gini, for countless hours of guidance and direction, her humor, and for enabling me to get this thing done in just three years! Thanks also to the members of my committee: Professor Arindam Banerjee for his ability to illuminate machine learning concepts and his guidance in finding related work to stand on, Professor Paul Schrater for inspiration for new directions to explore and Professor Paul Johnson for giving me a philosophical viewpoint that was rarely present in my computer science classes.

To my family, Sara, Max and Noah: You have made this possible in countless ways. I cannot begin to express my gratitude in words. So I simply thank you with all of my heart.

The views expressed in this work are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

Dedication

To Sara, Max, and Noah

Abstract

We advance the field of research involving modeling opponents in interesting adversarial environments: environments in which equilibrium strategies are intractable to calculate or undesirable to use. We motivate the need for opponent models by showing how successful opponent modeling agents can exploit non-equilibrium strategies and strategies using equilibrium approximations. We examine the requirements for an opponent modeling agent, including the desiderata of a good model.

We develop a new measurement which can be used to quantify how well our model can predict the opponent's behavior independently from the performance of the agent in which it resides. We show how this metric can be used to find areas of model improvement that would otherwise have remained undiscovered and demonstrate the technique for evaluating opponent model quality in the poker domain. The measurement can also be used to detect occasions when an opponent is not playing an equilibrium strategy, indicating potential opportunities for exploitation.

We introduce the idea of performance bounds for classes of opponent models, present a method for calculating them, and show how these bounds are a function of only the environment and thus invariant over the set of all opponents an agent may face. We calculate the performance bounds for several classes of models in two domains: high card draw with simultaneous betting and a new simultaneous-move strategy game we developed. We describe how the performance bounds can aid selection of appropriate classes of models for a given domain as well as guide the level of effort that should be applied to developing opponent models in those domains.

We expand the set of opponent modeling methods with new algorithms and study their performance empirically in several domains, including full scale Texas Hold'em poker. We explore opponent modeling improvement methods useful when the set of opponents we may face is unknown or unavailable. Using these techniques, we develop PokeMinn, an agent that learns to improve its performance by observing the opponent, even when the opponent is attempting to approximate equilibrium play. These methods also pave the way for performance optimization using genetic algorithms and efficient model queries using metareasoning.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Anything that takes shape can be countered.

- The Masters of Huainan

Imagine working in an adversarial environment, trying to determine your next course of action. You know what your goal is, but there are others operating here too. Others who are not on your side. Others who may know things that you do not. As you decide on actions to take, the others are simultaneously plotting their next actions, hoping to make the best of their situation - which may involve hindering your progress. The environment itself is full of uncertainty: you don't know whether luck will be in your favor or not - but you must decide what to do next.

This situation captures the essence of military operations, many economic endeavors, sporting events, and competitive games. It is the situation we focus on.

Throughout this work we will use several terms to refer to specific elements of the situation. The reader will find that the majority of these terms are used commonly in multi-agent literature in computer science. *Agents* are decision-making actors within the situation. The *environment* is where agents exist and interact. Environments can be described by a collection of rules for interactions between agents and with non-agent *objects*. Environments also specify what information may be obtainable for each agent. Agents may have *hidden information* that is unavailable to other agents. Because each agent may have access to different information, each agent may

form a different *belief* about the current situation. Agents perform *actions* within the environment which can affect other agents or objects, possibly changing the *state* of the environment. An agent's collection of procedures for making decisions is known as the agent's *policy*. An agent's policy is a mapping from beliefs to decisions. Agents may incorporate beliefs about another agent's future behavior in their policies with an internal model of the target agent. Creating and maintaining such models is known as *agent modeling*.

The *adversarial* environment is one in which the well-being or *utility* of an agent increases when the the utility of the other agents is reduced. In a competitive environment, such as a game of soccer, the goal of an agent (or team of agents) is to outperform other agents (or teams). Competitive environments reward the agents according to their performance. Generally the winner earns points while the loser does not (or perhaps loses points), and score directly affects the player's utility function. In a direct conflict environment, such as a military battle, the goal of an agent is to make the other agent's actions irrelevant or eliminate the other agents. In direct conflict environments, an agent's utility is increased by surviving or eliminating the other agents.

From the viewpoint of a specific agent in an adversarial environment, the agent's *adversaries* are known as its *opponents*. When an agent is trying to make decisions about which actions to take, it may consider the opponent's nature and the opponent's expected response to the action being considered. When an agent considers an opponent's reaction separately from the environmental changes that occur we say they are using an *opponent model*. The act of creating and maintaining opponent models is known as *opponent modeling* (which is a subset of the more general agent modeling endeavor).

In many real-world adversarial environments, the number of available sequences of actions and interactions between agents and the hidden information inherent in the environment make searching for an optimal action (or even an action with the highest expected value) computationally prohibitive. In these interesting environments, opponent models provide a performance improvement by altering the likelihood of certain sequences of actions which (hopefully) reduces the size of the search space to be explored to determine the optimal action. Allowing an agent to prune the search by using an opponent model can improve the chances to discover a solution which is

very good given the context of the environment and the opponents. In the following sections we explore both interesting adversarial environments and opponent models in greater detail.

1.1 Interesting Adversarial Environments

An *interesting* adversarial environment is one in which conditions within the environment make finding optimal policies that work well against all possible opponents impossible. Conditions that make finding an optimal policy hard include the number of possible states in the environment, the presence of hidden information and the presence of chance. Perhaps the best way to understand this concept is through examples of environments that illustrate each of these factors.

In tic-tac-toe, all possible actions can be enumerated, as well as all possible sequences of actions. There is no hidden information. From any state of the board, a set of actions which would lead to the best available outcome can always be determined, and one action can be selected using a search algorithm on the game tree. This can be done no matter what policy the opponent is using, thus tic-tac-toe is not an interesting adversarial environment.

Betting on horse races does incorporate chance, and possibly hidden information, but the betting actions of any given agent do not have effects on other agents within the same race (the odds don't change based on who bets for each horse). Betting on horse races is not an interesting adversarial environment because there is no interaction between agents in the environment: it is not adversarial.

Chess, while having an intractable search space, has no hidden information or chance affecting the game (beyond the choice of which sides the players play). Since it is a perfect information game, each agent knows the full state of the world. Thus there is no way for an agent to deceive his opponent about his situation within the game. Without the possibility for deception or the hope of luck, an agent can only rely on the strength of his play to beat his opponent. Rational play in chess therefore requires that a player always play the strongest move he is able to determine. The catch here is the phrase "that he is able to determine". This is the concept of bounded rationality [68]. Since agents are bounded by their ability to compute a rational choice and the environment of chess is too large for any agent to complete a full

search, agents often rely on some *approximation* of optimality for decision making. Because all chess-playing agents are at best approximately optimal, they are not guaranteed to be optimal against all opponents, and are thus subject to exploitation. Thus, chess is an interesting adversarial environment.

Rock-paper-scissors, also known as RoShamBo, is a game where agents choose an object and the winner is determined by their choice: Rock beats scissors; scissors beat paper; and paper beats rock. If both players choose the same object, the outcome is a draw, with neither player changing score; otherwise the winner earns 1 point and the loser loses 1 point. In the repeated version of this game, agents play against each other for a specified number of rounds or until one achieves a certain score. There is no hidden information in the game and chance does not play a role. The environment is certainly adversarial. There is also a known Nash Equilibrium[1] (NE) for this game (to play each choice with 1/3 probability, drawing randomly on each decision [40]). Because the optimal policy is easy to compute, playing against any single opponent is not interesting: if either player chooses to play the NE, the average score for both agents would be zero. A ranked tournament of multiple agents playing in pairs however can be an interesting adversarial environment. If every player chooses the NE policy, they will (on average) each obtain an overall score of zero, causing their tournament rankings to be simply a stochastic function due to the variance in the selections made by each player on each iteration of the game. In order to separate oneself from the competition, it would be necessary to try to predict what one's opponent was going to play next (which is only possible if the opponent does not play a NE policy). Under ranked tournament conditions, there is some incentive for players to deviate from NE if they believe their opponents might also do it: when only one player deviates, he risks nothing, as all his adversaries will continue to play NE and his score would be the same as if he played NE. Yet if at least one other player also deviates, then between the two deviating players, the one who is better at predicting his opponent will yield a positive score (and the worse agent will obtain a negative score), causing ranking separation from the remainder of the agents in the tournament. Once we begin to consider all of the possible non-NE strategies that could be used in the play of ranked-tournament iterated rock-paper-scissors,

[1] A Nash Equilibrium is a set of strategies, one for each player in which no player has incentive to unilaterally change strategy. The strategy can be pure (deterministic) such as always choose to play rock in rock-paper-scissors or mixed (a distribution over actions) such as play each object with equal probability.

we realize that the space of policy search for all iterations and opponents given all observations is too large to compute an optimal policy. Thus, a ranked tournament of iterated rock-paper-scissors is an interesting adversarial environment.

Poker is a game of chance, hidden information, and skill. While the action space in many versions of poker is smaller than the space in chess, it is still intractable to compute anything other than an approximation of an optimal policy that would work against all opponents. Furthermore the element of hidden information allows poker players to practice deception by either over- or under-representing their true card strength. The approximately optimal policy that outperforms a player that hardly ever uses deception is vastly different than the policy that outperforms one who is known to deceive often... and nothing prevents a player from switching between these archetypes over the course of a match. Thus, it is very hard to develop a one-size-fits-all policy in poker, making it an interesting adversarial environment.

Soccer is a game that incorporates chance through the physical interactions of the ball with the imperfect players and the field. Unlike the previous examples, a soccer match is not a discrete environment. The size of the search space over actions within soccer depends on the number of parameters considered and the granularity of measurement. Also unlike the other previous examples, soccer is an asymmetric-action environment. The physical abilities of the players (such as maximum speed and passing accuracy) are unique to each player. The actions available to each player are weakly defined. In soccer, rules of the game govern what a player is not allowed to do, but do not limit what is possible and thus the game (like many sporting events) may be subject to spectacular "never before" seen events. When the set of actions available is undefined, the method of computing optimal policy is unclear, thus finding an optimal policy to play against all possible opposing teams is not possible. One must tailor one's team policy for the current opponent team in order to do well. Thus, soccer is an interesting adversarial environment.

Often real world environments where agents interact exhibit the properties which make them interesting adversarial environments. In the next section we examine a method which is particularly useful in making decisions in interesting adversarial environments: opponent modeling.

1.2 Opponent Modeling

Opponent modeling is the act of building and maintaining an internal representation of an opponent. There are certain conditions under which developing and using an opponent model would be fruitful.

1.2.1 Preconditions for Opponent Modeling

Developing an opponent model is only possible when either the agent will be exposed to an opponent multiple times or has access to records of the previous activity of the opponent. There are two common-sense characteristics of an engagement with an adversary that could make opponent models helpful:

1. The environment is symmetric (or nearly so): one side does not have an overwhelming advantage over the other. If one side did have an overwhelming advantage, it could win the engagement handily, regardless of the other agent's beliefs about its behavior.

2. The opponent (or at least one of the opponents in a multiple-opponent engagement) is expected to not know (or be able to calculate) the equilibrium[2] solution in his reasoning. An equilibrium solution is a policy that guarantees a minimum value for the agent regardless of the behavior of the other agents. If all the agents in an engagement are using an equilibrium solution for the engagement, opponent modeling may be unnecessary as it is unable to advise behavior that would alter the outcome of the engagement.

1.2.2 Using Opponent Models for Exploitation

When the number of actions and number of possible states of the world is relatively small, and the space of outcomes is definable, an agent can determine the best action to take given that the opponent is also attempting to take the best action for himself. This phenomenon often occurs in many simple games, leading to the condition known in game theory as equilibrium [56]. However, if the agent knew that his opponent may

[2]By equilibrium, we mean the set of strategies (one for each player) where no agent would prefer a different strategy, given the other agent's strategies. While we are keeping what type of equilibrium (e.g Nash Equilibrium or Correlated Equilibrium) purposefully vague in this discussion, as a general rule, when we refer to equilibrium we are discussing Nash Equilibrium, unless otherwise noted. We discuss specific types of equilibrium in further detail in Chapter 2.

not necessarily be using a perfect equilibrium strategy (perhaps because interaction is occurring in an interesting adversarial environment) there are actions the agent could choose that may increase his score beyond the expected score of the strategy that he would play to achieve equilibrium. We expect that if he was able to know his opponent better, he could make predictions about the opponent that would facilitate exploitation.

To illustrate the interaction between equilibrium strategies and exploitative strategies, let us begin by exploring a concrete example. Consider again the two-player zero-sum game rock-paper-scissors. The equilibrium strategy in this game is to randomly choose an action from the set with equal probability (1/3 chance rock, 1/3 chance scissors, 1/3 chance paper). The expected value of this strategy is that each player will win 1/3 of the time, lose 1/3 of the time, and draw 1/3 of the time. Each player will expect a net score of zero, and this expectation is independent of the opponent's action. Thus, if we decide to play the mixed equilibrium strategy and the opponent is playing the same mixed strategy, we will both achieve a net score of zero in the long run, and if the opponent is always playing rock and we are playing the equilibrium, the same outcome occurs.

But what if we realized that our opponent was always playing rock and we thought we could do better than to break even with our current mixed equilibrium strategy? We could increase our overall utility if we adopted a strategy of always playing paper. In fact, if we somehow knew that the opponent would always play rock from now on, we could win every game. Sometimes deviating from an equilibrium strategy to exploit our opponent can improve our score when we know that the opponent is not acting according to the equilibrium strategy.

Another motivating factor for learning an opponent model is that it allows us to predict how the opponent will respond to an action that we take. Knowing how an opponent will respond to an action may allow us to choose an action that influences them to act in a way we desire. For example, consider playing a weak poker hand against an opponent we know to be risk-averse in betting. By betting high (bluffing), we may make our opponent feel that the risk is too great to stay in the game and cause him to fold. By shaping their behavior in this way, we may be able to directly increase our winnings.

A final motivation for opponent modeling is that there may be times when we either

want to deny the ability for the opponent to gain certain information or trick our opponent into believing something other than the truth. In military and intelligence communities, these activities are referred to as denial and deception [49]. While opponent models are not necessary to perform denial and deception, an opponent model would help us predict the opponent's response to the missing or false information, and would help insure that their ensuing behavior would not be counterproductive to our goals.

With this understanding of the value of opponent models, we can begin to look at the specific capabilities which opponent models might provide. To be useful, an opponent model must either *predict* opponent actions or be able to *explain* an opponent's past actions. Predictions could indicate the next action the adversary could take, while explanations provide hypotheses of the beliefs the adversary might hold when they take actions in certain contexts.

An opponent model is an internal replication of the decision process (policy) which an adversary uses: how will he act given his belief about the state of the current situation. We describe these three targets of prediction and explanation as *state*, *action*, and *policy*, respectively. An agent can use these targets to help maintain its internal policy by evaluating "what if" situations and answering "why did the opponent just do that?" questions.

By linking together a series of proposed actions of an agent and responses by the opponent to form an *action trajectory*, the agent can evaluate the trajectory's likelihood and predict a future utility value for that trajectory. Likelihood and value can be converted into *expected value* of the trajectory. Alternately, the agent can examine the sequence of actions that ends with the most recent opponent action to explain what the opponent's current belief about the environment is.

The inputs to the opponent model can be anything observable to the agent using the opponent model. Observable features include the past states of the world and the opponent's past actions in those states. We assume that the opponent policy is not available for direct observation, although the space of possible policies could be pruned based on observed states and opponent actions.

1.2.3 A Formal Definition

As mentioned previously, we consider two main uses of opponent models: prediction and explanation. The two uses differ in the relationship between inputs and outputs.

Our first definition of an opponent model describes the model used for predicting what the opponent will do in a given circumstance. More formally, this opponent model is a function that takes the opponent's belief about the world state (what is known by the opponent) as input and outputs a predicted action that the opponent will take. In a discrete-time or event-driven environment, we can describe the model (M) in terms of time steps or turns t according to the *state* (S) the opponent believes he is in, and the *action* (A) we predict he will make:

$$M : S_t^{opponent} \rightarrow A_{t+1}^{opponent} \tag{1.1}$$

If the model encodes a probability distribution over actions taken in each state, then the model can compute the distribution $P(A|S)$ for the opponent. This distribution is useful in environments where agents take actions simultaneously (or in circumstances when we don't always get to see what the opponent's action is). In these conditions we would like to predict the distribution over next actions of our opponent and use that prediction to choose a utility-maximizing action, given their action. This distribution would also be useful in an environment where agents perform actions in sequence. In that type of environment, we want to know how an opponent is likely to react to an action we take.

In other circumstances, an explanation (or probability distribution of explanations) of why the opponent took a particular action may be more useful. Consider environments where part of the world state is hidden to us. In these environments, it may be useful for us to discover the hidden information so that we can make a more informed decision about our next action. If we have an explanatory model of the opponent, the model may reveal his beliefs (and the hidden information) through his actions. Formally, this type of model could be described as a function that takes an opponent action as input and returns the likely state of the world observable by the opponent as it existed before the opponent's action was taken:

$$M : A_t^{opponent} \rightarrow S_{t-1}^{opponent} \tag{1.2}$$

If this second model encodes a probability distribution, then the model can compute $P(S|A)$ for the opponent. Such a model is revealing the explanation for why the opponent takes a given action. Since the function returns the state of the world as seen through the eyes of the opponent, we can determine the nature of the information that was available only to them. For example, if we know that an opponent poker player is risk-averse (this is known as a *tight* player [69, 70]), we posit that this player is less likely to bluff, and our opponent model of him would predict that if his bet was high, it is very likely that his cards (the hidden information) were actually good cards in that hand.

1.3 Scope of This Work

With a basic definition of opponent models as our foundation, we can describe which specific areas of opponent modeling this work focuses on. While we have presented the motivation for opponent modeling in terms of an abstract encounter between actors in an adversarial environment, for this work, we focus on computational opponent models that guide the decisions of an intelligent agent behaving in such an environment. Thus, our work will examine the intelligent agent (which we refer to interchangeably as the agent, or *modeler*) and the adversary: one or more other agents which we refer to as the opponent(s) or *modellees*, interchangeably.

We further restrict our focus to examining opponent modeling systems which attempt to make predictions about the adversaries based on observations obtainable by the agent. In Machine Learning terms, we are interested in the online learning setting where an agent is pre-programmed with domain knowledge but has no prior model of a specific opponent that it is about to face. The agent must learn about its opponent during the encounter such that it can make decisions about how to act in the future. We focus primarily on systems that will learn from observations obtained during the encounter with the opponent rather than systems which attempt to learn about opponents via pre- or post-analysis of historical encounter information, although we do propose using the latter to explore the viability of un-tested opponent modeling techniques.

In this work we examine opponent modeling in several domains:

Full Scale Poker (Texas Hold'em): A well-studied and currently extremely popular strategically complex asymmetric information domain that incorporates effects of non-actor stochastic processes. This domain provides a test-bed for opponent-modeling agents where the game is complex enough that its exact equilibrium strategy has not yet been solved[3]. Furthermore, there are a growing number of good poker agents which we can use as targets for our model. We describe the details of this domain and the challenges it presents for opponent modeling in Chapter 3.

Simultaneous Turn Strategy Games: A new game we created that was inspired by the computer entertainment genre known as Real-Time Strategy games. This simultaneous turn-based, perfect-information abstraction allows much of the strategic interaction in an environment where approximate solution concepts are computationally tractable. Much like chess, this class of games allows for different branching factors in each state of the world, but unlike chess, when an agent is about to select an action, the move the opponent is about to make is unknown. We describe this game in Chapter 5.

Simple Betting Games: These games require the players make bets based on the results of partially observable stochastic processes. High Card Draw with simultaneous betting is in this class of games. These games comprise an asymmetric information domain that incorporates effects of non-actor stochastic processes. This domain allows fast computation of probabilities of having a better "hand" so that agents can focus on learning the strategic action tendencies of their opponents without needing the level of domain knowledge required for traditional poker. This domain is outlined in Chapter 7.

[3]The state space for the simplest form of Texas Hold'em (2-player Limit) is larger than 10^{17}. While an exact equilibrium strategy for this game has not been generated, the highest ranking agents in the AAAI 2007 Computer Poker Competition were agents that pre-calculated approximations to the equilibrium strategy by using abstractions (of hand strengths and game-tree information sets). When making decisions, these agents took actions decided stochastically according to probability distributions stored a very large lookup table which covered every state in the abstraction.

1.4 Contributions

We advance the field of modeling of opponents in interesting adversarial environments. To do this, we push the current knowledge in four key areas:

- We create an opponent model that learns how to improve its ability to play poker through observing its opponent and updating an opponent model. In Chapter 4 we describe the agent, recount its performance in the international computer poker competition and showcase new techniques to measure the effects of learning in isolation from the other performance characteristics of the agent.

- We define several characteristics of a good opponent model and present a new domain-independent measurement technique for comparing prediction performance between different models. We evaluate the results for two models used in our poker playing agent in Chapter 6 and show how it can be used to uncover weaknesses in each model. Such measurements allow agent designers to make informed decisions regarding where to put resources into improving their modeling systems and can become the building-blocks for automated model-selection or ensemble-based techniques.

- In Chapter 7 we develop bounds for the possible performance of an agent that are a function of only the domain conditions of the adversarial environment and do not depend on the actual actors in the environment. We use this method to find the value-improvement bounds in two domains: simultaneous high-card draw and the simultaneous turn strategy game. This bounds-finding technique will enable system designers to determine if, and what type of opponent models would be worth their development costs in a given adversarial environment.

- We present several techniques which allow us to make sound incremental improvements to existing opponent modeling systems when the actual opponents we may face are unknown or unavailable in Chapter 8.

The remainder of this thesis presents these contributions and the domains in which we have demonstrated their efficacy. Before we delve into the main body of work we review related research in the next chapter.

Chapter 2

Related Work

This chapter focuses on two main areas of prior research: the way agents interact in an adversarial setting, and techniques for measuring the quality of their performance. We explore these areas to illuminate how models of behavioral interactions can be represented, populated, used, and measured.

2.1 Interacting Successfully in Adversarial Environments

Developers building agents must have some method for their agent to decide what to do when trying to thrive in the adversarial environment. We categorize agents' decision-making into three main classes: Rules-Based, Game-Theoretic, and Opponent Modeling.

Rule-based decision-making (a more general version of expert systems) are often marked by hard-coded decisions and parameters [62]. The rules are developed from the experience of humans with domain knowledge. While rules-based systems often perform well in specific domains, they are less scalable and portable than the other decision-making methods and they are often very hard to maintain in rich domains. They also tend to exhibit biases generated by the knowledge of the experts providing their rules. For these reasons, we do not discuss prior research on rules-based systems further in this chapter.

Game Theoretic solution concepts are often used to generate strategies that should work well in a particular domain regardless of the opponent's strategy. Often, they

guarantee some minimal outcome. These policies may take a long time to compute, but an agent using them can usually make decisions extremely fast once faced with the actual opponent in the environment. We discuss interesting topics in game-theoretic decision making in Section 2.1.1.

Artificial Intelligence has provided many techniques for representing environments and opponent models, populating the models (through value-based search or opponent-learning methods) and using opponent models for determining optimal policies. We explore opponent modeling literature in detail in Section 2.1.2.

2.1.1 Game Theoretic Considerations For Agent Behavior

When we think about the set of possible opponents we might face, we must consider a set of agents that may forego exploiting their opponents in order to minimize the chance of being exploited. Game theory is useful for analyzing such situations. In game theory, the agents are known as *players*, the environment they interact in is known as a *game*. *Outcomes* occur as the result of certain actions by the players, and are associated with *payoffs* which are the scores or utilities each player earns at that point in the game. Players create *strategies* which are a selection (or distribution over) action(s) for each possible decision they have to make in the game [40, 58]. Since the same decision might need to be made under several different conditions, strategies describe the action taken for every possible condition. Given a set of strategies, one for each player, expected payoffs for each player can be computed [58].

2.1.1.1 Rationality and Equilibrium

One can think of a game-theoretic decision making system as one in which the specific domain information and an assumption of the opponent's *rationality* are used to compute a *best response* to every situation that could occur in the environment. According to Aumann and Brandenburger [6], rationality is the assumption that an agent is trying to maximize his utility given his beliefs. When playing a game where there are opponents acting in the environment, rationality requires that a player is attempting to maximize his own utility given his beliefs about the other players and the environment. A best response is a response to an opponent's action (or the combined actions of all other players) that yields a utility no worse than any other response to that action. Thus, the search for a best response to rational opponents'

actions is itself rational. When all agents are playing simultaneous best responses to each other, they induce the game-theoretic concept of an *equilibrium*. In an arbitrary game there may be multiple equilibrium points (intersections of best responses for all players) and each point may have unique utilities for each player. Even the sum of the payoffs for each player under different equilibrium points may be different) [58]. Since not all equilibria are equal, it is important to understand the assumptions and implications of the method used to solve for the equilibrium.

Two important types of equilibrium solution concepts are *Nash equilibrium* [55, 56] and *correlated equilibrium* [5]. Both of these types of equilibrium describe a situation that is stable in the sense that no player has any incentive to change their own strategy, and no reason to expect other players to change theirs. The Nash equilibrium solution concept (which is a subset of correlated equilibrium) makes the assumption that there is no device that can synchronize the joint actions of the agents, forcing agents to consider every possible interaction of strategies. The correlated equilibrium solution concept allows for the existence of some device which enables agents to synchronize on joint actions, thus eliminating some lower payoff strategies from consideration. Often correlated equilibrium solutions in *general sum games* (games where the sum of the payoffs for all players for each outcome is arbitrary) can achieve a higher average *social welfare* (total payoff of all players) while the Nash concept yields equilibrium points with a lower average social welfare for the group. Unfortunately, it is unclear how to create or use the synchronization device required by correlated equilibrium in a real world environment, leaving Nash equilibrium as the more widely used solution concept.

2.1.1.2 Representing Environments as Games

In order to calculate equilibrium conditions and generate a policy which favors decisions enabling equilibrium, one must first cast the environment into a specific canonical representation. Researchers have developed many techniques for expressing adversarial domains in canonical formats. Koller and Megiddio present a good explanation of the mapping methods in [46] which we now summarize. In simple environments where agents each take a single simultaneous action, and there are no stochastic elements or hidden information, the set of joint actions can be written using a *strategic form* (also known as normal form) of the game. The strategic form of a game is an n-dimensional matrix where n is the number of players. Each dimension in the matrix

indicates the set of possible strategies of a single player in the game. If action s_i is player i's choice of strategy, then the outcome of the game is represented in the cell at the intersection of all the strategies $(s_1...s_n)$ chosen by the players. Within that cell is a *payoff vector* indicating the payoff earned by each player. Figure 2.1 depicts an example strategic form game for two players. Notice that the number of strategies for each player may be different.

		Column Player		
		L	M	R
Row Player	T	r_1, c_1	r_2, c_2	r_3, c_3
	B	r_4, c_4	r_5, c_5	r_6, c_6

Figure 2.1: An example two row, three column strategic game with payoff vectors for each strategy.

Other more complex environments where sequences of several agent actions occur before an outcome can be determined may necessitate expression using the *extensive form* of the game (which can be converted to an equivalent, albeit unwieldy, strategic form game.) The extensive form of a game can be viewed as a tree where each non-terminal node (including the root node) represents a decision by an agent. Terminal (leaf) nodes represent outcomes of the agent interaction, and include the resulting payoff vector describing the payoffs for each of the agents. Extensive form games are able to also express conditions where certain information is not available to some (or any) of the players. This is accomplished using an *information set*. An information set is a collection of nodes, each one associated with the same decision options and known information for that player. The nodes in an information set differ only in that each one exists under a different hidden condition which is unknown to the player making the decision in the node. For example, in a game where each player draws a card from a standard 52-card deck, the first player does not know which card his opponent holds, so his action exists in an information set containing 51 nodes (one for each of the opponent's possible cards). The player must make a decision without knowing which node within the information set he is actually in.

Chance effects not controlled by any agent (such as weather conditions or which cards are dealt from a shuffled deck) can be expressed as the actions of an external player in strategic form of a game, and chance nodes in the extensive form of a game.

2.1.1.3 Solving Games

Without loss of generality we discuss finding solutions to normal form games next (since finite extensive form games can be converted to normal form games prior to solving, or solved directly through other methods [46, 47]). All finite strategic games have at least one mixed strategy Nash equilibrium, and certain types games also have a pure strategy Nash equilibrium, as discussed in [58]. Some games can be "solved" by finding the game-theoretic value of the game if all players were playing the equilibrium strategies. In essence, solving a game determines if any player has an inherent advantage, and quantifies that advantage if the terminal states of the game have defined values.

Allis [2] describes three classes of solutions for games: Ultra-weakly solved, in which only the value of the game is known (for each player), but the actual strategies are unknown; Weakly solved, which adds to the ultra-weak solution the condition that a strategy is known for achieving the game-theoretic value from the opening position; Strongly solved, in which the value and strategy is known for every position (or state) in the game. Allis states that all of these definitions assume the use of "reasonable [computational] resources". The definition of reasonable resources "should typically allow the use of a state-of-the-art computer and several minutes of computation time per move". Allis also points out that when used to play the game, algorithms that discovered the game was a weakly-solved draw (tie) should be able to guarantee a draw from the starting position. Such strategies are not guaranteed to win, even if the opponent makes a mistake. A strong solution, however, provides a strategy from every state to achieve the value of that state. Thus, a strong solution player should be able to find a win in a game originally determined to be a draw if the opponent makes a mistake that puts the state of the game into one favorable for the solution-based player.

Many games have been solved to date. The list includes rock-paper-scissors (solvable analytically), tic-tac-toe (trivially solvable by enumerating the states and solving analytically), connect four [1] and checkers [64]. Some game positions for certain board positions of chess, reversi, and go have also been solved, although solutions to the full forms of these games continue to be intractable.

2.1.1.4 Intractable Solutions

As researchers try to find solutions for more complex games, the computational requirements increase. Since the size of a normal form of a game is exponential in the size of the extensive form of a game, converting a game from extensive form to normal form then using standard linear programming techniques to solve it quickly becomes intractable. While Koller's team [46, 47] has developed methods for solving extensive form games directly in time polynomial in the size of the game tree, most game trees are still to large to solve with these methods.

The state of the art in solving more complex games requires solving abstractions of the games. In poker, Shi and Littman have used abstractions [65] of full-scale games and focused on solving the abstractions, mapping play in the real game to the abstraction, and mapping the solution of the abstraction back into the real game. The Computer Poker Research Group at the University of Alberta and the research team at Carnegie Melon University have both explored various abstraction and solution techniques.

In [10] Billings et. al. developed the first known full scale abstraction-based Limit Texas Holdem player using a primarily manually-selected set of abstractions of the state space. The abstractions included reducing the number of bets allowed per round, separating the game tree into several betting round phases, and bucketing hands based on a granular measure of card quality. By empirically estimating the transition probabilities between abstract states the University of Alberta team generated a new game which was an abstraction of the full version of poker. They then solved the abstract game using linear programming and created several agents based on slightly different variations of the abstraction. Zinkevitch et. al. continued improving the solution techniques and developed the *range of skill* algorithm which could generate approximate equilibrium solutions with fewer resources (time and memory) than the previous methods [84]. This enabled them to use larger (finer) abstractions which resulted in better performance and enabled finding the approximate solution for an abstraction of the entire four-betting-round version of Limit Texas Hold'em.

At Carnegie Melon University, Gilpin and Sandholm focused on building a strong Limit Texas Hold'em player by developing methods of *automatically* generating the abstraction prior to computing a solution. In [35] Gilpin automatically generated an abstract state space (using a parameterized abstraction algorithm *gameshrink* which controls how much loss should be allowed in the abstraction: 0 for no abstraction; ∞

for complete abstraction) for the early betting rounds (where the game tree is very large) and explicitly solves the later rounds (which have a much smaller game tree) in real time. The early rounds are solved off-line using CPLEX (a linear program solver). Gilpin continued to improve the automated abstraction technique in [36] by generating abstract states in which the expected error between the value of a hand of cards within the abstraction and the value of a hand of cards in the real game is minimized. The new abstraction generator also had the capability of parameterizing the number of desired abstract states instead of the desired amount of loss. By examining the available resources one can generate the maximum number of abstract states which would still have a tractable solution, and the new abstraction algorithm could be given the proper parameter. Gilpin's team went on to develop an even better poker playing agent known as GS3 [38] using additional domain information and a deeper abstraction technique that groups hands by the probability distribution over their outcomes rather than just their expected outcome. This abstraction was also able to treat an entire poker hand (all four betting rounds) as a single entity (instead of multiple phases of interleaved solutions to sub-games).

Gilpin's team makes the first comparison of abstraction-based techniques for Rhode-Island Hold'em (poker) in [37]. The author compares two abstraction techniques - one in which the states in the original game are binned according to their expected likelihood of winning (such as [35, 84]), and one in which they are grouped according to their distribution of outcomes, as in [38]. He finds that for coarse abstraction levels, abstractions which group states based on similarity of expected outcome perform the best, but as the abstractions become finer (more members in the abstraction), the abstraction method that incorporates the probability distribution over outcomes when binning states yields a stronger performance. This finding implies that in the future, as more computational resources become available, the abstraction trend should lean towards grouping states based on the distribution-similarity measurement instead of just their expected value.

The strong solutions described by Allis [2] assume all players will act rationally from the current state. A solution under the rationality assumption is one in which each player simultaneously maximizes his own payoff, given the strategies of all other players also trying to maximize their payoff. Unfortunately, any differences between rationality in the original game and rationality in the abstraction will lead to potential differences in behavior when solving the abstraction. Furthermore, the approximate

equilibrium techniques discussed above are inherently lossy in that they ignore perfect rationality to some degree. Combining the lossy abstraction and approximate solution techniques will most likely exacerbate the problem. It remains to be seen how irrational the abstraction and approximation techniques' solutions will be, but any irrational play could lead to exploitation.

2.1.1.5 Summary of the Game Theoretic Approach

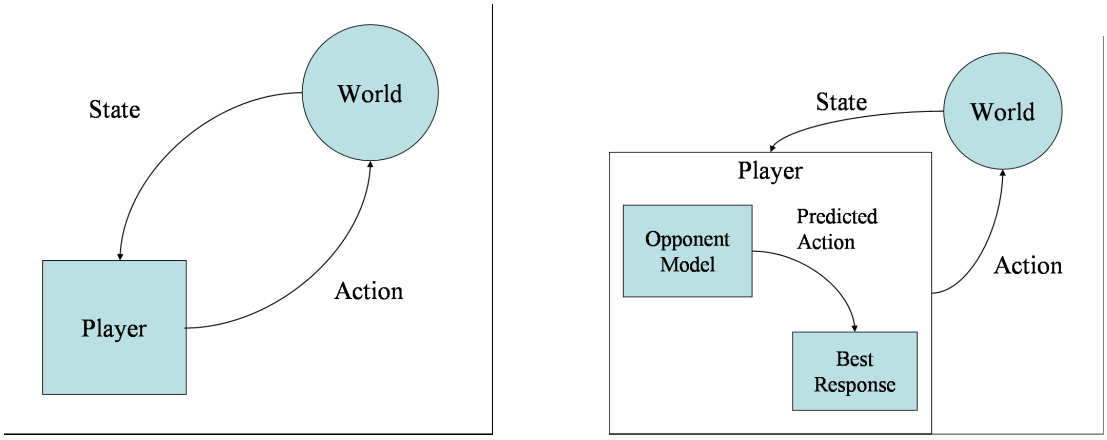
While equilibria-based strategies provide desirable guarantees regarding outcomes of games, there are some preconditions for these equilibria that are not met in all instances of all adversarial encounters [6]. In particular, both Nash and correlated equilibria assume that each player at least believes that all other players will continue to act based on their equilibrium strategies. By definition the opponents obtain no benefit by changing so the belief is reasonable, but this belief does require some assumptions about the knowledge and rationality of other players. Abstracting the game and approximating the equilibrium solution puts this assumption into question.

Even if the computational resources were available to fully solve the original game, there are additional issues with the implementation. For example, correlated equilibrium requires either an outside mediator or a communication (perhaps negotiation) arrangement that reaches its own equilibrium (usually a stronger type of equilibrium than Nash, such as sequential equilibrium, since bargaining allows threats and demands) [6, 7, 33, 57]. Not all real-world scenarios allow these mechanisms. Clearly, there are problems with establishing the preconditions for equilibrium in many circumstances. These issues expose players who attempt equilibrium play to exploitation in all but the simplest games. Even when the equilibrium conditions are present and the equilibrium strategy is easily computable, if there are players who don't choose it, there are other players who can exploit them, leaving the overall performance of the equilibrium players in the middle of the pack. This phenomenon was demonstrated empirically during the first international RoShamBo programming competition [9].

2.1.2 Opponent Modeling

Assuming that we are acting in an environment where we are not sure of the best policy, what are the steps required to maximize our utility in that environment? In the literature, there appear to be two approaches to this challenge. Figure 2.2

provides a visual representation of these approaches: In the implicit method, we learn how to maximize our utility with respect to *our* observations and actions. In the explicit approach, we keep internal representations of our opponents and use them to maximize our utility by finding a best response to the expected actions of our opponent(s), given the current conditions of the game we are playing. This dichotomy is similar to the choice in reinforcement learning algorithm design between model-free methods (such as Q-learning) and model-based ones [45].



(a) Implicit Opponent Model Approach

(b) Explicit Opponent Model Approach

Figure 2.2: Implicit vs. Explicit Approaches to Opponent Modeling

In effect, the first method aggregates the behavior of our opponent(s) and the world into a single model. We "learn" the aggregate model and attempt to maximize our behavior given the way the world appears to work. While this approach simplifies our inputs and outputs because we don't have to consider the underlying process of the opponent separately from its behavior in the world, it may take more observations to discern distinct behaviors of the aggregation.

The second approach requires a separate model for the opponent (or opponents). When planning our next action, we consider the predicted actions from the opponent models, and determine a best response given the current conditions of the world. We then take the best-response action. In this section, while we do examine literature using the first approach, our primary focus is on the research accomplished using the opponent-model paradigm. We are also interested in the computational methods used to learn and use the opponent model, especially the effect of the relationship between the type of opponent, the learning algorithm, and the world itself on the process.

2.1.2.1 Representations of Opponent Models

We now explore the different structures that researchers use to represent models of opponents. Our focus is on two broad categories of models: domain independent models, and domain-specific models. As we shall see, the choice of structure used to represent an opponent model is greatly impacted by the assumptions about the decision process the opponent is using. Furthermore, the choice of model implicitly constrains the types of opponent strategies that can be learned and expressed in the model.

Perhaps one of the simplest types of domain-independent models is a record of the history of the actions taken by the opponent. In his 2000 champion rock-paper-scissors playing agent "Iocaine Powder" [28], Dan Egnor describes one of the three meta-strategy components called frequency analysis: The frequency analyzer looks at the opponent's history (of choices for rock, paper, or scissors) and finds the move he has made most frequently in the past, and predicts he will choose it. The opponent model used by the frequency analyzer is simply a count of how many times the opponent selected each choice.

Another slightly more complicated history-dependent model is proposed by Jensen et al. [44]. Jensen's model was originally designed for the sequence prediction, but he also adapted it for use in opponent modeling for games in which the only notion of state is in terms of the history of opponent actions (such as rock-paper-scissors). In this model, the opponent behavior is represented as a sequence of actions from an alphabet. The model uses hypotheses in the form of combinations of alphabet characters and wild cards to represent plausible explanations for the sequences that the opponent has generated so far. For example, in rock-paper-scissors, the alphabet R, P, S, * can be used to express a hypothesis such as H1=[*,*,R,*,P] which would predict any two arbitrary actions followed by rock followed by an arbitrary action, followed by paper. Thus, this hypothesis would correctly predict an opponent who played paper after [RRR] or [RS] or [P], as well as several other sequences.

In the two examples above, the environment has no state, so any internal beliefs an agent has are based only on domain knowledge or the opponent's past behavior. If models using only past actions are expanded slightly to capture the frequency or sequence of actions given the state of the world at the time, then the model would represent the state-to-action transition probabilities observed from the opponent's

viewpoint, and it could be used in more complex tasks such as conditional prediction from Equation 1.1

One type of domain-independent model used to represent state-to-action transitions in much of the early opponent modeling work [15, 21, 61] was the Deterministic Finite Automaton (DFA), also known as the Deterministic Finite State Machine. DFAs can represent the set of regular languages and each instance of a DFA provides the ability to determine if a sequence of characters is a member of a specific regular language. DFAs are composed of an alphabet used for input and output, a set of states, an initial state, and a function which deterministically produces the next state S' if we receive input string A when in state S.

If we assume that the strategy of an opponent playing a 2-person matrix game could be represented as a deterministic, finite set of if-then rules that decide what action to take next using some history of our actions, then we could use a DFA to encode the opponent model. When using a DFA to describe opponent behavior, we map a string of input characters to the last n actions observed by the opponent and the states of the DFA are used to predict the next action the opponent will take.

Consider the well-known two-person matrix game "Prisoner's Dilemma". In this game, the assumption is that two criminals who committed a crime together have been caught by the authorities. The authorities are trying to make a case against each of them but they have insufficient evidence so they are interrogating each player separately, tempting them with plea-bargains. Each player must choose whether to "cooperate" (admit that they both committed the crime, providing the evidence the authorities wanted) or "defect" (don't say anything to help the authorities). If both players cooperate then they will each serve a short sentence. If one player cooperates and the other defects, the defecting player will serve a long sentence and the cooperating player will go free. If both players defect, they each will serve a medium length sentence. If two agents play this game repeatedly, it is known as Iterated Prisoners Dilemma (IPD). Now consider an IPD strategy known as "Tit for Tat". In this strategy, the player starts by cooperating, then in all future iterations, plays whatever action the opponent played in the last iteration. Thus, if we defect and the opponent cooperates in this iteration, then we will cooperate in the next iteration. A graphical depiction of a DFA that represents this strategy is shown in Figure 2.3.

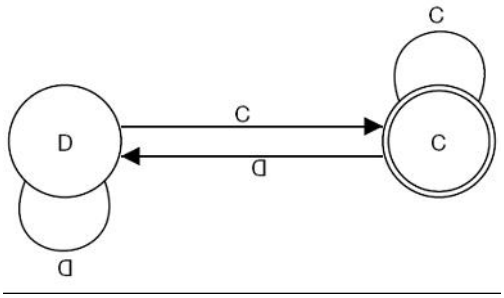


Figure 2.3: A DFA model of an opponent's policy representing "Tit For Tat" Iterated Prisoner's Dilemma Strategy, originally presented in [61]. In this DFA, which can also be viewed of as a state-transition graph, there are nodes (circles) and edges (arcs between the nodes). The initial node is indicated by a double concentric circle. Each node in the policy model represents a possible state our opponent could be in during an iteration (which has a one-to-one correspondence with the action he took in the iteration). Our action in this iteration indicates which arc our opponent's policy should follow to transition to his next state, which predicts his next action. Thus, if we just cooperated (C) and the opponent just defected (D) we know based on this model of him that he will Cooperate (C) on his next action.

While DFAs can model opponents with deterministic strategies, they cannot represent probabilistic transitions between states. The consequence is that if our opponent is using a mixed strategy (such as randomly choosing with uniform probability an action selected from rock, paper, scissors), any algorithm trying to learn a DFA opponent model will have poor performance. Since many equilibrium strategies in even trivial games such as rock-paper-scissors require a mixed strategy, the DFA severely limits what opponents can be modeled. Mor has demonstrated that another drawback with using DFAs is that learning a DFA can require exponential time in some cases [54]. He does identify a subset of possible DFAs that are learnable in polynomial time, but no proof is offered that these can form reasonable approximations of DFAs outside that class if the opponent is not using a DFA of this type.

If the opponent's strategy is probabilistic in nature, then we need a more general model to describe it. Freund et al. propose using a probabilistic state automaton (PSA) to represent the opponent [29]. A PSA is essentially a generalization of a DFA where instead of generating a deterministic action based on the state the automaton is in, the automaton selects an action from a set of actions with some probability distribution over them. These authors also propose another model, the Probabilistic Transition Automata (PTA), which adds the property of having probabilistic transi-

tions between states. Unfortunately, PTAs turn out to be extremely computationally expensive to use when building opponent models. In a separate work [15] Bjarnason and Peterson use Probabilistic Action Automata to represent the strategy of the opponent, although they do not clearly explain how PAAs differ from PSAs and PTAs.

Other probabilistic graphical models such as Bayesian networks can be used to represent models of processes. An extension of a Bayesian network which includes chance nodes, decision nodes, and utility nodes known as an Influence Diagram (ID) [42] can be used to represent the decision processes of a single agent. The multi-agent version of the Influence Diagram, (MAID) [48] allows explanations of behaviors of communities of agents. Gal and Pfeffer use extensions of MAIDs known as Network Influence Diagrams (NID) [31] to model opponents. A NID is a hierarchical model which allows representations of opponents who themselves contain (possibly recursive) models of the learning agent (and possibly other agents). Furthermore, a NID can contain several candidate models that are deemed plausible representations of opponents, using a probability distribution over these models to represent proximity to the real opponent. Solving a NID amounts to computing behavior that is rational with respect to the agent's beliefs (bounded rationality [68]). One desirable property of NIDs is that they can be used both to predict an agent's actions ($P(A|S)$) or explain an agent's actions ($P(S|A)$)

The models used by Altman, Bercovici-Boden, and Tennenholtz are interesting because they are intended for learning across games [3]. The test domain includes auctions, one-shot strategic form games, and a game called The Centipede Game. The learning is unsupervised, and it only seeks to learn similarities between different decision problems. These measures of similarity are then used with Case Based Reasoning (CBR) to predict a player's decisions based on previous games that are judged to be similar based on previous players' choices on those games. The association rules learned using this method successfully identified players who had general tendencies, such as a tendency to cooperate or to bully other players, or to seek equitable (symmetric) payouts for both players. Although the CBR rules did no better than baseline on most players, they did provide a boost of 10% to 15% to the accuracy of predictions about the players who matched the rules. Moreover, the players who matched these rules tended to be the players whose choices were otherwise difficult to predict. The researchers also tested decision trees (ID3), and clustering (K-nearest neighbors) implementations with less success, and considered a non-linear separator

for pattern recognition (support vector machine) but did not have data to provide adequate training.

Steffens also explored using CBR for opponent modeling in [72, 73, 74]. His work focuses on determining how to define the similarity measures used to determine whether a new scenario is similar to one in the case base. He examines the types of knowledge that can be encoded in the case base and the impact of encoding imperfect, sometimes conflicting, domain rules into the knowledge base. Among many interesting findings, he observes that in opponent modeling it is especially important to incorporate domain data into the similarity measures, since CBR is not otherwise suitable for learning in situations with sparse data.

While we have pointed out several types of domain-independent models that can be used to represent an opponent, there are many other domain-specific models, which can be used to represent important information during a decision process. For example, in the RoboCup[1] Soccer Domain, Riley and Veloso discuss using a hand-coded playbook of Simple Temporal Networks (a structure used to describe plans) to represent possible opponent models [59, 60]. By comparing the similarity of each of the plans with the recent activity on the field, the authors can generate a probability distribution over the set of plans.

In separate work in the Simulated RoboCup Soccer Domain, Stone et al. developed an opponent model capable of representing an optimal soccer opponent's maximum physical capabilities [75]. This model exploits a particular characteristic of the domain - player agents have hard limits on their maximum movement speeds, turning speeds, kicking speeds, etc. This model effectively characterizes a prediction of the worst case scenario of how far a player could move within a certain amount of time. If we consider that every opponent has the ability to arrive at an arbitrary position within a larger and larger radius circle as time moves forward, we can think of this model as encoding the threat radius of the opponent at time t. While at first appearance, this model may seem limited in its applications, it has several alternate uses, including military applications. The military often models opponent aircraft in terms of their maximal threat footprint (accounting for the current speed and turning capabilities of the aircraft and the weapons envelopes [19] describing where a weapon will have a high probability of success).

[1] See http://www.RoboCup.org/ for more details

A different type of model is used by Sturtevant and Bowling to represent a opponent's desires at a more abstract level [76]. If we have a game where there are multiple possible outcomes, and the outcomes can be tied to the score that each player receives, then each player may naturally desire some outcomes over others. To represent this, Sturtevant and Bowling defined an opponent model as a directed graph where vertices are possible outcomes of the game and edges are used to encode the preference relationship (arrow points to non-preferred outcome from preferred outcome). An agent may have several potential models of his opponent. Each of these opponent models can be described by its edges (u, v) when u is preferred to v by the player being modeled. Transitivity of preferences holds with these models, as does the ability to build a general model of an opponent from several more-specific models. A generalization is an aggregate model built from the intersection of the constituent models. It is a new graph, which is as specific as possible while still being consistent with all of the constituent models. By ruling out some of the constituent models based on observations of opponent behavior, we can create a generalization that captures only the aggregation of consistent models, and this generalization can be used to describe a more accurate behavior of the opponent.

One somewhat subtle but important aspect of what a player might include in a model is whether the opponent is in turn modeling her. This feature is difficult to obtain, but is very likely to be of increasing importance as basic opponent-modeling methods improve. A series of very interesting results on this subject have been reported by Vidal and Durfee [80, 81, 82]. Because the number of computations required to use nested models is exponential in the number of nested levels, Vidal and Durfee proposed a method for determining which level of nesting was providing the best performance, thus allowing other levels to be pruned and results to be provided in polynomial time for a fixed number of players (for k agents, each with n possible actions, their algorithm runs in $O(n^k)$).

Another result from Vidal and Durfee demonstrates how modeling an opponent may be harmful if a model is inaccurate. This is a case where an incorrect bias in the hypothesis space hurts the ability of the player to learn the true behavior of the opponent. Vidal and Durfee created an economy of buyers and sellers in which sellers offer abstract goods of varying quality to buyers at a price set by the seller. The buyer is not able to determine the quality of the good before buying, but can determine its value afterwards and determine whether it gained or lost on the transaction. In the

Buyers	Sellers	Lessons
0-level	0-level	Equilibrium reached only when all sellers offer the same quality. Otherwise, we get fluctuations. Mean price increases when quality offered decreases.
0-level	Any	Sellers have big incentives to lower quality/cost.
0-level	0-level and one 1-level	1-level seller beats others. Quantitative advantage of being 1-level predicted by volatility and price distribution.
1-level	0-level and one 1-level	Buyers have upper hand. They buy from the most preferred seller. 1-level sellers are usually at a disadvantage.
1-level	1-level and one 2-level	Since 2-level have perfect models, they win an overwhelming percentage of time, except when they offer a rather lower quality.

Table 2.1: Effects of various levels of opponent modeling depth originally used in [82].

context of this economy, Vidal and Durfee describe a 0-level agent as not modeling other agents, a 1-level agent as modeling other agents as 0-level agents, a 2-level agent as modeling other agents as 1-level agents, and so on. As shown in Figure 2.1, the 1-level sellers do well at beating the 0-level sellers as long as their model accurately represents the buyers. However, once 1-level buyers are introduced, the 1-level sellers suffer because they cannot learn that the buyers are learning to avoid them, and their performance falls below that of the 0-level sellers. In each round, the agents with the most accurate models performed best. In general, Vidal and Durfee find that the only penalty for having a too-deep recursive model is computational - the higher-level agents seem able to handle predicting any lower-level agent, although we have not found a theoretical argument that this would always, or even usually, be the case.

The condition that could potentially occur if agents try to stay one level of modeling ahead of their opponents is addressed by the Iocaine Powder algorithm mentioned earlier [28]. The algorithm takes advantage of the circular nature of rock-paper-scissors interactions to terminate this recursion after three levels. Terminating the recursive modeling works because an agent playing rock-paper-scissors with a deep recursive opponent model would end up making the same predictions as an agent using a much simpler opponent model. This insight does not apply in all domains, but it is possible that other applications may be found. The Iocaine Powder agent also carries a safeguard in its model. It uses an ad hoc form of an expert-voting algorithm,

and one of the "experts" on which it maintains weights is a Nash-equilibrium player. If all of the experts attempting to model the opponents fail, the agent will have a high confidence in the Nash expert, and play falls back to the unique Nash equilibrium for the game. We have not discovered any literature describing other examples designed with explicit Nash equilibrium safeguards like this, but believe that it is a useful concept that might be applied in other domains to offset the risks implicit in playing away from a known (or estimated) equilibrium.

An opponent model is important, but an opponent model that can be learned rather than specified by a programmer, and that can adapt to changes in opponent behavior, is really our goal. Although static approaches to opponent modeling are still the state of the art in some domains [67], we believe that learning opponent models will become more prominent in future research.

2.1.2.2 Learning Models of Opponents

One of the most intuitive classes of techniques used in machine learning are the gradient-based search methods [27]. In these methods, one tries to maximize or minimize some multi-parameter objective function. Essentially, in each iteration we choose the direction we feel likely to maximize our progress and we take a step in that direction. One of the most difficult aspects of the gradient based approaches is determining how big of a step to take. The step size is often controlled by a hand-tuned "learning rate" parameter. Bowling and Veloso address the issue of learning rate step size with respect to learning a model of an opponent. These researchers developed the Win or Learn Fast (WoLF) class of algorithms that use a variable learning rate to control the speed of learning [18]. Since they are exploring only the class of two-player general-sum matrix games, Bowling and Veloso use a single parameter opponent model that represents the probability $P(a)$ of the opponent choosing action a or action b, such that $P(b) = 1 - P(a)$. They also use one parameter from the learning agent which describes the probability $P(c)$ of the learning agent choosing action c or d, such that $P(d) = 1 - P(c)$. To learn the parameters, they take the derivative of the rewards function (reward earned when we take an action and the opponent does also) and perform gradient descent (GD). The technique that makes their approach novel is that they set the GD learning parameter based on whether the best response to the current opponent model is making them win or lose. Here, winning is defined as performing better than the equilibrium strategy for the game. If they are losing,

then their opponent model is wrong, and they make the learning parameter higher, thus moving quickly away from their current model. If they are winning, then the opponent model is probably close to correct, so they set the learning parameter small to fine-tune an opponent model that is already very accurate. An interesting effect of this performance-based learning rate tuning is that if the opponent is pursuing a non-stationary (adaptive) policy, WoLF will be losing, and the learning rate will be high enough to alter the model so it quickly follows the movement of the opponent in strategy space. Bowling and Veloso have shown that because of the variable learning rate they meet two of the desired conditions for learning opponent models: Rationality and Convergence (discussed further in section 2.2.1). Unfortunately, their method requires the computation of the equilibrium condition(s) for the game in order to determine if they are winning or losing. While this is not necessarily costly for simple games, it may be prohibitive or impossible for games that are more complex.

Another method that attempts to follow the behavior of a moving opponent strategy was proposed by Jensen et al. The Entropy Learning Pruned Hypothesis Space (ELPH) algorithm [44] is an adaptation of version space learning [53] that uses an entropy calculation to cull out high-entropy hypotheses: the ones with poor performance of being able to predict just the observations of the last few sequences of actions in play. While originally designed to handle sequence prediction based on the observed history of the sequence, Jensen adapted the algorithm to handle policy learning in simple games that have no state except the prior history of actions taken by the players (such as rock-paper-scissors). Jensen showed empirically that the algorithm was very quick to learn when an opponent had changed strategies and the algorithm maximizes its performance against those opponents, while retaining the ability to converge on equilibrium (optimal) strategy when the opponent moved to towards equilibrium play. In another experiment, Jensen showed that ELPH could out-play human opponents in rock-paper-scissors by a large margin.

Many authors have explored probabilistic methods for learning parameters to populate opponent models. When an opponent's behavior in a game can be expressed in terms of probability parameters that control decision processes, these algorithms can be highly effective. One example of an environment suitable for parameterization of strategies is Kuhn poker. Kuhn poker is perhaps the simplest version of poker that has both hidden information and strategic betting. The game uses a three card deck (J, Q, K such that $K > Q > J$) and there are two actions available to each player (bet

or pass). Since the strategy space is small, the game tree can be fully expanded, and parameterized. Hoehn et. al. use Maximum A-Posteriori estimates of parameters based on a Beta prior and in-game observations when building opponent models of Kuhn Poker-playing agents [41]. In a separate experiment, Hoehn et. al. also employed an expert-based bounded regret algorithm (Exp3 [4]) to attempt to learn the opponent's strategy directly. The authors noted that in general, when attempting to learn an arbitrary fixed-strategy opponent the parameter learning method outperformed the strategy learning method. Although the authors did not explain why parameter learning outperformed strategy learning consistently, our intuition is that the difference in performance is most likely explained by the fact that the algorithms were searching hypothesis spaces of different sizes. The parameter-based algorithm searches a much smaller hypothesis space of fixed-strategy-opponents and that space actually contains the parameters that correctly represent all the opponent behavior, but the strategy learner makes a less biased assumption (the opponent strategy could be non-stationary) and its hypothesis space is much larger. Thus, with the same amount of training data/iterations, performance with the strategy learner is much worse than that of the parameter learner, as seen in the experimental results.

Other teams such as the University of Alberta Computer Poker Research Group have also explored opponent modeling for poker. In [13], Billings' team created an opponent model that keeps track of the likelihood of the opponent playing each of their possible starting hands in the way it was played. They use a generic model which acts as a default (prior to any observation of a specific opponent) and a specific model which is updated by observing their current opponent. They use these "weights" as input to an expected value calculation which allows them to determine the next action to take. Davidson examined using a neural-network to learn an opponent's behavior in [25]. By using 19 parameters characterizing the state of the game as input, Davidson's agent was able to learn an opponent's tendencies within a few hundred hands. In Billing's team's well known poker playing agent *Vexbot* [12], the opponent model tracks the opponent's conditional behavior at each node in the game tree (without considering what cards have been dealt) and also tracks the probability distribution of the opponent playing with certain card strengths at certain nodes in the game tree based on observations. Using this information, the agent attempts to calculate an approximate expected value for each action (fold, call, raise) it could play next. While this agent performed very well against fixed policy agents at the time, it has since

been outperformed by several of the newer game-theoretic agents (such as [36, 38]). Vexbot is also the victim of its greed for data: to populate its opponent model enough to play decently against a strong opponent requires thousands of hands of observations, often too many to exploit the opponent in a normal-sized tournament; and determining default data to populate a generic model has been problematic.

While the agents discussed above take into account the entire history of observations when populating the model, this is not always the ideal situation when the opponent's strategy is changing. In some cases, the opponent is acting with a set of rules based on an observed history of fixed length. For example, our opponent might allow our agent to take a particular sequence of actions we desire only once every 10 turns (considering our sequence to have been an accident, or perhaps noise), but if we try to perform the sequence more often, the opponent will change strategies to punish us. Unfortunately, when we first encounter the opponent, we have no idea how long a history the opponent is keeping track of. Freund, et al. provide an approach [29] for the fast simultaneous exploration and exploitation of an opponent assumed to be acting according to a set of rules with visibility of an arbitrary length of history. In the first phase, the agent 'floods' the history of the opponent, filling it with a "homing sequence" of benign actions that effectively re-sets the opponent (with high probability). Then in the second stage, the agent takes exploratory actions to try to determine the maximum frequency in which it can play a dangerous sequence (with high probability). Once this frequency is determined with high probability the agent moves into an exploitation phase.

We have reviewed several methods of learning opponent models. While using a model once it is learned may not seem as difficult as learning the model, it can present a significant challenge as well, and examining the process provides critical information about the usefulness of various methods for learning opponent models in a given domain.

2.1.2.3 Using Opponent Models

We described earlier how DFA have been studied as a domain-independent way of modeling one's opponents. However, after learning the DFAs one still has to make a best response to the predicted action of the opponent. In 1988 Gilboa showed that if each player implements his strategy for an extensive form game as a DFA,

then if we know each other agent's DFA, we can calculate a best-response DFA in polynomial time, unless the number of players is unbounded. If the number of players is not bounded, the problem is shown to be NP complete [34]. In 1990, Ben-Porath extended this work by examining the case in which each player implements a mixed strategy by randomly choosing one of a set of DFAs, each of which implements a deterministic strategy [8]. He proved that a best-response mixed-DFA strategy can be calculated in polynomial time, but only if the size of the opponents' DFAs are limited to a known finite size and the number of players is a known, fixed finite number.

A common method of representing the space of actions and states in a game is through the use of a game tree. In a two-player game, the first player's actions are branches from the root node, and the second player's actions are represented as sub-branches that hang off the first player's nodes. Each successive level in the tree is generated for alternating players in this way until all possible paths of play have been captured in the tree. In a full game tree, each path from root to a leaf represents a possible sequence of turns between player one and player two. Ideally, if an agent found itself at a particular node in the tree, it would attempt to follow a line of play that takes it along a path from the current node to the leaf with the highest utility.

Of the many algorithms designed to search for a line of optimal play in a game tree, minimax is one of the most popular [62]. Minimax is a tool for determining an optimal line of play because it assumes the opponent will always play the action that maximizes his or her own reward and minimizes our reward. In fact, minimax's optimal value calculation is equivalent to the value of the equilibrium point(s) of the game [76]. Many methods for playing games are inspired by the original game-tree search algorithm minimax. We examine one line of work that extends the minimax algorithm to be compatible with opponent models.

Several research groups have explored extending the minimax algorithm for use with opponent models, multiple opponents, and multiple opponents with opponent models. Carmel and Markovitch developed M^* [20] which generalizes minimax to use arbitrary opponent models to calculate the utility of the decision at each layer in the tree. While this structure is impossible to prune using the general alpha-beta pruning mechanism because of the inherent complexity of the model-based utility calculation, the authors provide a limited-capability pruning algorithm $\alpha\beta^*$ pruning which returns the M^*

value of the tree while searching only the necessary branches. The potential benefit of adding an opponent model to minimax is that instead of assuming the opponent is playing according to minimax, M^* can take advantage of any opponent expressible as a model.

Luckhardt and Irani extended the original minimax algorithm to handle an n-player game without opponent models. In their max^n algorithm [51], instead of propagating a single value at each node on the tree, the algorithm propagates a tuple of all players scores back up the tree. In max^n, the chosen subtree is the one that maximizes the score of the player making the choice. One of the unfortunate problems in n-player games that does not exist in two-player games is that in n-player game trees, the player making a choice at a node may see that his score in each of the child nodes is equivalent, but that the scores for the other players differ between the children. Consider the tree in Figure 2.4. In this tree, the middle player is making a choice of

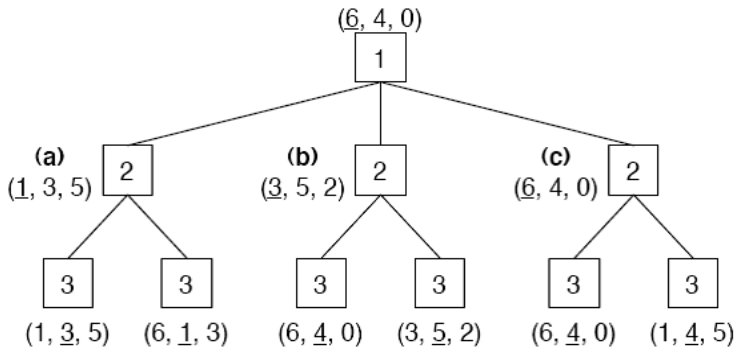


Figure 2.4: N-Player game tree with max^n value computation (originally presented in [51])

which leaf node to pick from position (c) on the tree. Each of the leaf nodes have the same score for the middle player, so without any other information, the choice is arbitrary. The problem occurs because we also hope to minimize our opponents' scores, but in an n-player game, without additional information, it is hard to tell which way of splitting up the other 6 points between our opponents is in our best interest.

Sturtevant and Bowling handled this dilemma in their development of the $Soft{-}max^n$ algorithm [76]. In addition to propagating the scores up the tree, the $Soft - max^n$ algorithm propagates the information regarding opponent's preferences (based on the

opponent model discussed in section 1.2.3). By allowing the decision maker to see the opponent preferences when trying to make the decision, we can make a better decision when the situation would otherwise be tied.

Another method inspired by the minimax algorithm is the "Ideal-Model-Based Behavior Outcome Prediction" (IMBBOP) algorithm developed by Stone et al. for use in Robocup Soccer [75]. This algorithm is particularly useful for quickly assessing the relative quality of a proposed action so that the best action can be chosen from amongst a set of proposed actions. By assuming the opponents will act optimally (they will instantaneously move to thwart the proposed action), each opponent agent can be represented by a growing-over-time circular obstacle with respect to the friendly teammate who currently has the ball. By comparing the time it would take to achieve the proposed action with the time it would take for the opponent's threat circle to intersect the path, we can calculate a quality metric on the action. For example, when a player is considering kicking the ball at time t, if at time $t + i$ game cycles into the future the ball will be inside of the threat radius surrounding an opponent, then the kick is considered as having risk. By comparing reward levels of proposed actions and using thresholds to prune actions with too great a risk, teammates can quickly decide on the best course of action, assuming the opponent is behaving optimally.

2.2 Measuring Quality of Opponent Models

While there is an abundance of literature providing tools and techniques for measuring the performance of agents, there is a substantially smaller body of work discussing the performance of opponent models. In this section we examine the literature describing the desired traits of opponents models as well as existing opponent model measuring techniques.

2.2.1 Desired Properties of an Opponent Model

Before delving into the existing research on model quality assessment, we first examine some of the desired attributes (the desiderata) of a good learning algorithm. Michael Bowling and Manuela Veloso [18] proposed two desirable properties of multi-agent learning algorithms: Rationality and Convergence.

- *Rationality Property*: "If the other player's policies converge to stationary policies then the learning algorithm will converge to a policy that is a best response to the other player's policies". If the opponent policy happens to be optimal, then an equilibrium condition will be found, otherwise the learning algorithm will converge on a best response to a suboptimal opponent policy - which usually means the learning algorithm will converge on a policy yielding the highest utility against the opponent.

- *Convergence Property*: "The learner will necessarily converge to a stationary policy. This property will usually be conditioned on the other agents using an algorithm from some class of learning algorithms."

In addition to these properties, Jensen et al. [44] point out that being able to *quickly learn when a non-stationary opponent strategy has changed* is also important. While rationality and convergence are very desirable in long term game play against stationary opponents, if the opponent is non-stationary and an agent cannot quickly track the opponent policy faster than it changes, performance will diminish.

Rationality, convergence, and the ability to quickly detect, follow, and exploit a non-stationary strategy are three desired traits of a good learning algorithm. In the next section, we examine the literature on measuring performance quality of an opponent model.

2.2.2 Opponent Model Measurement Techniques

Computer science is rich with research on different techniques for how to represent an agent in a model, how to obtain the information to populate the model, how to use the model, and how to evaluate the model based on overall performance in the domain. While actual performance within the domain is ultimately the goal for agent, there are many other factors affecting overall agent performance besides how accurate the agent model is. The structure of the environment, the prior biases in the behavior of the modeling agent, and the way it uses the information from the model are a few examples. Because there are so many factors governing an agent's performance, the quality of an opponent model's predictions cannot be assumed to be highly correlated to agent performance. If we want to determine *how well a model works*, we need to measure the prediction accuracy of the opponent models *directly*.

Opponent modeling is often used in computational analysis of an environment to help an agent make decisions. Researchers have examined many domains where opponent modeling can be useful: competitive economics, national security, politics, and, of course games [49]. While most researchers have provided empirical studies that compare the overall domain performance of their methods with other modeling methods or different approaches (such as Monte-Carlo simulation, game-theoretic equilibrium play, or rules-based strategy), few have quantified how well their models predict the other agents' behavior directly.

There are several exceptions in which researchers do examine the accuracy of the model, not just the performance of the overall system. Carmel and Markovitch examine the model size and average error versus sample size while running their domain independent modeling US-L^* algorithm [21], showing that model size growth slows with more examples while average error drops. Rogowski expands on this work, providing the *it-us-l** algorithm and presents its domain-independent model quality measure (average hold-out-set prediction accuracy) in several experiments [61]. In robot soccer games in RoboCup, Riley and Veloso use the probability of correctly recognizing which play an agent is about to make to measure their learning algorithm [60] but do not provide any other direct model quality measures. In the plan recognition field researchers have also employed the measurements of precision and recall [16, 23] when comparing performance of candidate recognition algorithms.

The majority of the model prediction quality measures in these efforts rely on extensions to error measurements intended for binary classification (1 point for correct prediction, 0 points for incorrect prediction). While this information does provide a basic quality measure, its value diminishes as the number of possible agent actions per state grows. In an arbitrary environment, there are no restrictions on uniformity of the number of actions leading from a state. Some states may have few actions leading away while others may have many. Under this condition, the meaning of a function comprised of binary-based prediction quality measurements from multiple heterogenous states is unclear. For the function to be meaningful, the underlying quality measure must be more general.

A more detailed representation of general classification performance (which is applicable to the performance measurement of a model that predicts which action an opponent will take) is the confusion matrix. A confusion matrix is an M-by-M ma-

trix which represents a classifier's distribution of classification labels provided to M different classes. Column headings hold the model's predictions and row headings indicate the true class. Thus, the cells along the diagonal represent correct classifications and the off-diagonal cells represent incorrect classifications. In addition to providing accuracy or error rate (computed from the main diagonal of the matrix), the matrix reveals the number (or probability) of each type of mistake (off-diagonal cells). This additional information can be valuable when different predictive mistakes have different costs. Several agent modeling efforts use the confusion matrix to characterize their model prediction quality. Davidson uses a confusion matrix to quantify a neural-network agent model's ability to predict whether the opponent will fold, call, or raise under many different circumstances in poker [25]. Sukthanar and Sycara use a confusion matrix to characterize their prediction of which type of breach and enter maneuver the opposing force is about to perform in a simulated military tactical engagement [78].

While confusion matrices are a step in the right direction in that they provide quality assessments beyond basic binary prediction accuracy, they can quickly become unwieldy if the space of possible options (M) is large (or continuous) or there are multiple stages of predictions that must be made during the course of an engagement (such as in chess, poker, and military endeavors). In multiple-stage prediction encounters (which can often be characterized as extensive form games and depicted with game trees) there would need to be a confusion matrix for every possible state where the opponent is next to act. Sometimes the chance of traversing to a given state is not certain - the probability depends on which events occurred over the history of the encounter (the path taken through the game tree). The probabilities might be co-dependent (depending on the probabilities of all the agents in the environment). In these circumstances there may be no clear way in which to generate a single confusion matrix quantifying the goodness of a model.

2.3 Summary of Related Work

A review of the related work reveals that opponent modeling draws on many different fields and sub-fields, including game theory, and many parts of machine learning. We have provided an overview of key concepts involved with selecting opponent model structures, learning opponent models, using opponent models, and assessing their

performance. We've discussed the impact of model structure on what kinds of opponents can be represented and looked at some of the opponent models used in normal and extensive form games, poker, RoboCup soccer, and military applications. We've examined the desired characteristics of opponent-model learning approaches (rationality, convergence, and the ability to quickly adapt to a non-stationary adversary) and discussed methods for measuring how well opponent models met these goals.

We've found that modeling opponents that adapt while playing is an active and fertile field of research. Problems such as how to model agents who are using opponent modeling themselves, how to transfer learning methods developed in abstract games to more concrete games, and how to detect when an opponent is modeling our behavior are challenging areas that have clear benefit and an intuitive interpretation, but are far from understood.

Before we can answer these challenges however, we need to answer three even more fundamental questions which remain open research in the field:

1. Before deciding what kinds of information an opponent model should be predicting in any given environment, can we determine if there are any bounds on how valuable each type of prediction would be?
2. Once we've built the model, how can we assess the quality of its predictions?
3. When we desire to improve our model (and the overarching agent), how can we accomplish this in a scientifically structured manner?

The remainder of this work focuses on finding answers to these questions.

Chapter 3

Texas Hold'em Poker

Texas Hold'em poker is one of the most popular and strategically interesting variants of poker today. The 2007 World Series of Poker (which featured Texas Hold'em as its main event) had a total prize pool of over $159 million. In 2007, Polaris, an agent composed of the best poker playing algorithms developed at the University of Alberta, competed for the first human-computer poker competition. The computer poker competition, now in its third year, allows competitors from around the world to vie for the title of champion in limit, no-limit, and ring versions of this game.

Why is this easy-to-learn game of such huge interest to professional gamblers, computer scientists, game theory experts and many others? We explore this question in detail in this chapter, introducing the reader to the game mechanics, showing how they combine to form an interesting adversarial environment for research, and outlining some of the challenges that Texas Hold'em presents for opponent modeling.

3.1 Texas Hold'em Domain Characteristics

Texas Hold'em poker is a well known zero-sum imperfect information game. It is a poker variant in which players attempt to make the best set of cards from their private (hole) cards and several fully visible community cards which are usable by every player. Each hand consists of multiple rounds of dealing and betting where players try to win the pot: the hand ends when either all but one player has folded (uncontested win) or at least two players have stayed in the game until all betting is complete (the showdown). In an uncontested win, the player remaining after all

others fold wins the pot. In the showdown, the player with the highest valued set of cards wins the pot.

There are four rounds in which players can bet chips based on the strength of their cards: the *pre-flop*, *flop*, *turn*, and *river*. If any player decides to fold before all four betting rounds have been completed, he forfeits any claim on the pot and must sit out the remainder of the hand while the other players finish. If all but one of the players have folded prior to the completion of the river betting round then the remaining player wins the pot uncontested and the hand is terminated without further betting rounds.

Each round has a different set of rules regarding the way bets are made. On the pre-flop, players bet knowing only the two private cards they hold since no community cards have been dealt yet. After the pre-flop betting is over, the dealer places three community cards on the table and players make another round of betting during the flop based on the overall strength of their private cards and the shared community cards. After the flop betting is complete, the dealer places one more community card on the table and another round of betting occurs during the turn. Next, the dealer places a final community card on the table and the players bet during the river. If more than one player remains after the river betting round is complete then the showdown occurs.

Texas Hold'em uses *blinds* to offset the information advantage of players who act later in the pre-flop. Blinds are compulsory payments from some players to the pot before any cards are dealt. In general, the player who will act last in the pre-flop pays a fixed amount of money into the pot and the player who will act just before him pays 1/2 that amount. These amounts are known as the big blind and little blind respectively.

In a heads-up (two player) limit (fixed increment bet) Texas Hold'em match, we can express each round of betting in terms of an extensive form game with imperfect information. If we consider that the cards each player holds and the community cards on the table are fixed, then a game tree for a round could be drawn as shown in Figure 3.1[1].

[1]To be game-theoretically correct, it is important to note that each player decision node in the extensive-form game tree for a given round is a member of an information set with all other equivalent-position nodes in the other game trees where the player's opponent has been given different cards by nature. Each of the game trees represents a different possible state for the hidden information (the cards held by the node-decision-maker's opponent). For sake of visual clarity, we

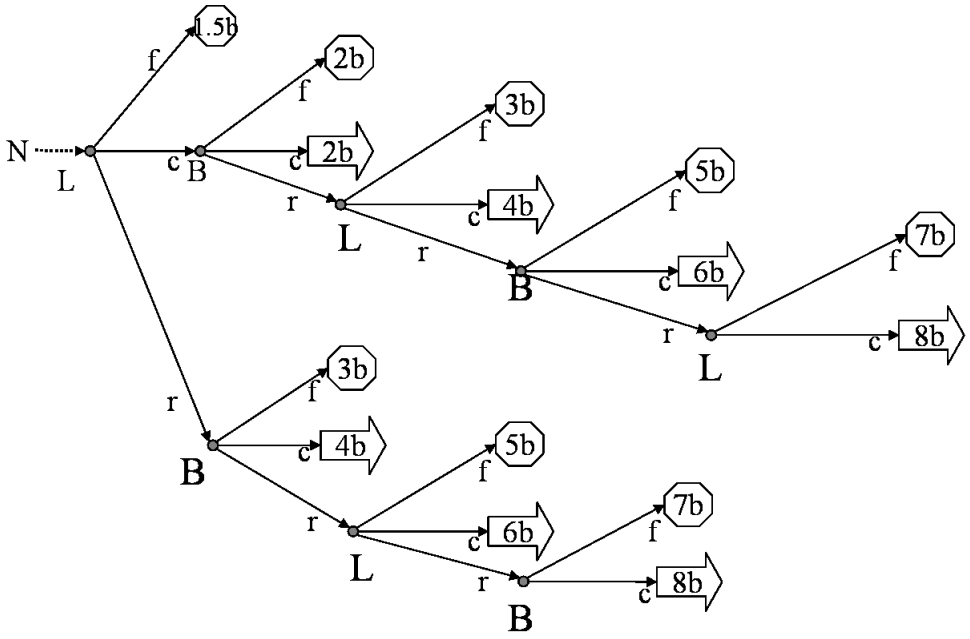


Figure 3.1: Pre-flop Betting Game Tree for Heads-Up Limit Texas Hold'em (3 raise limit). The tree consists of nodes (decision points and outcomes) and edges (actions that transition from node to node). 'N' depicts "nature nodes" where cards are dealt from the deck, 'B' and 'L' mark the actions of the Big Blind Player and Little Blind Player, respectively. Edges are marked with 'r' for raise/bet, 'c' for check/call and 'f' for fold. Stop signs and arrows represent terminal nodes for the tree - Stop signs represent conclusions of the hand while arrows represent a transition to the next betting round. Cumulative pot sizes (in terms of the small bet value which is equivalent to one big blind) are marked inside each terminal node.

The other betting-round game trees are similar with the exception that there are up to four raises allowed per round on the flop, turn, and river (instead of three as in the pre-flop) and that the river contains showdown nodes instead of continuation nodes[2]. In such a tree the terminal (leaf) nodes represent one of the following conditions:

- Fold Leaf Node: One player's wealth is increased by the amount the other player has contributed to the pot so far (when the other player folds)
- Showdown Leaf Node: Both players reveal their cards and the player with the

did not show the information sets in Figure 3.1.

[2]It is also important to note that for the flop, turn, and river game trees, the nature node at the root of the tree in Figure 3.1 is actually preceded by the continuation node that occurred in the previous game tree. Thus, the decisions within these trees depend not only on the state given by nature, but also on the which node from the previous round led to the nature node.

higher-valued cards takes the amount the other player has contributed to the pot. If the card values for each player are equivalent, the net payoff to each player is zero. This type of node exists only in the river game tree

- Continuation Leaf Node: A new game is played where community cards are dealt to the table and the next round of betting commences. This type of node exists only in the pre-flop, flop, and turn game trees.

Given all possible card dealing sequences for two players, and all of the possible betting sequences they could make over the four betting rounds there are over 10^{17} possible game states - making even this simplistic variant of Texas Hold'em very rich in strategic opportunity and yielding it intractable for exact equilibrium calculation [10]. Since exact equilibrium calculation is impossible, and approximations to equilibrium calculations are exploitable, opponent modeling becomes a viable strategy in Texas Hold'em play.

Before we discuss the challenges to opponent modeling in this domain, we explore the conditions of a specific competitive environment in which the strength of agents can be tested: the Computer Poker Competition.

3.2 Computer Poker Competition

In 2006, the University of Alberta hosted the first computer poker competition concurrent with the AAAI conference in Boston, Massachusetts [50]. Five entries competed. The top three entries were at least partially based on game theoretic principles and did not adapt to their opponent's actual play style. In 2007 the second computer poker competition was held in conjunction with AAAI in Vancouver, B.C. [83]. A new no-limit format was introduced in this competition. Ten agents competed in the no-limit format while seventeen competed in the limit competition. The top achievers from this pack also used game theoretic principles extensively, but there was little information regarding the algorithms used by the competitors who fared worse. In the remainder of this section we discuss the competitive environment for the limit format of the competition to clarify the challenges that an opponent modeling agent would face.

The 2007 computer poker competition was a heads-up round-robin (each pair of competitors will play a series of matches) limit Texas Hold'em tournament. Winner

rankings were measured in two different ways: The "Equilibrium" winner determination method which awarded the agent that could win the most matches (a match is won by having more money than your opponent at the conclusion of the match), and the "Online Learning" winner determination method which awarded the agent that could extract the largest total sum of money from all of their opponents.

Each match paired 2 opponents for 3000 hands in two linked games (known as forward and reverse) where the cards remained the same but the starting position of each player was swapped. In the reverse match, each player is dealt the cards the other player had in the forward match. Between the forward and reverse match, all state information and files are erased so that no agent's actions or cards could be 'remembered' between the two matches. From the agent's point of view, each time they played a 3000 hand match, they were unable to recall anything they may have observed in any other match. Thus, any learning about one's opponent had to occur within the scope of a single game of 3000 hands.

Agents were required to be submitted to the competition coordinators, and the agents were run on the dedicated machines for the competition. No team could alter or communicate with their agent once the competition began.

The competition coordinators ran enough matches between every pair of agents that there was a statistically significant result between each pair of agent's ranking. Since the agent's relative performance was unknown a-priori, the coordinators kept running matches until significance was achieved and they could provide relative rankings for the agents. Each agent played at least 60,000 hands versus each competitor, and some played even more hands in order to achieve a significant result.

Another interesting factor in this competition was that each agent had unlimited credit. Since the agents could never be forced out of a match by losing too much money, every hand in a match was equally important. Thus, agents had to be able to do well over the course of an entire match. This characteristic is interesting because in most human poker tournaments each player has a limited bankroll and one of the strategies is to extract the competitor's money as quickly as possible to remove them from the match. This human-game strategy is not viable in the computer poker competition, and each hand must be played independently.

3.3 Challenges for Opponent Modeling in Texas Hold'em Poker

Billings described some of the overarching research challenges of Texas Hold'em poker in general [11]. In the poker competition many of the rules induce additional complexity for agents attempting to learn their opponent's behavior from observations during play. We now discuss challenges particular to opponent modeling that are induced by the domain and exacerbated by the conditions of the computer poker competition described in the last section.

We discuss the particularly daunting aspects of the environment for the limit competition here[3]. First, the number of hands (iterations) per match is fixed to 3000. In each iteration, each player receives two cards drawn from a deck of 52 cards, so the number of possible unique combinations of starting conditions is:

$$\begin{pmatrix} 52 \\ 2 \end{pmatrix} \begin{pmatrix} 50 \\ 2 \end{pmatrix} = 1624350 \tag{3.1}$$

Even when we consider only the 169 possible equivalence classes of opponent card strengths (also known as card isomorphisms) on the preflop, we only get to see what cards they opponent held if neither player folds and a showdown occurs. Clearly, trying to learn the opponent's preflop behavior for each of those classes would be difficult in such a short game. For example, if both players went to a showdown in every hand (which would not normally occur when both players are acting rationally), it would take an average of about 1700 hands to categorize an opponent's first action (probability of call versus probability of raise) in the preflop for each given equivalence class to within 10%. Even when we know the opponent's behavior given the card isomorphism they are in, we won't know what card isomorphism the opponent is in while we are trying to explain their actions in the preflop.

If an agent is trying to learn the opponent's behavior, the agent cannot afford to spend the first half of the game learning and wait until the second half of the match to begin exploiting the opponent. Whatever learning method is used must begin to exploit the opponent immediately, using what it learns along the way.

[3]The rules for the no-limit competition are similar with the exception that the number of iterations played is different.

An opponent model must be able to either predict the probability of the opponent's action in a given circumstance, or explain what cards (or card strength) the opponent holds given his action in a certain context. With the number of possible states and outcomes in the game, the hidden information, and its lack of revelation when either player folds, how can a learning agent hope to develop a strong opponent model in far fewer than 3000 interactions with its opponent? We explore this topic next as motivation for the agent we developed for the 2007 Computer Poker Competition.

One piece of information about the opponent (which is only discovered during showdown) is: given the opponent's last action, what is the probability distribution over card strengths they might hold. Another piece of information about the opponent that is always revealed regardless of whether his cards are seen in showdown is his action given the previous actions that have occurred so far. For example, given that an agent has just called as the first action in the preflop, what is the probability that the opponent will fold, call or raise in response. Since the opponent's cards are unknown to the agent in the preflop, any estimate of this probability is helpful for the agent in helping them produce a profile of the opponent's behavior in the preflop.

Given the difficulty of opponent modeling in this hidden-information domain, we would want to learn from information that is always revealed, as opposed to information that is revealed only at showdown. Since opponent actions are always visible, the conditional action probability makes a very reliable target for observation. We discuss how we learned the conditional probability distribution of our opponent as well as other important agent-design topics in the next chapter.

Chapter 4

PokeMinn (2007)

Our first attempt at opponent modeling in a competitive forum was a great success. We developed PokeMinn to compete in the AAAI 2007 Computer Poker Competition against an unseen set of opponents submitted by teams from around the world. Our agent competed in two categories which measured its ability to win games (the "equilibrium" category) and earn money (the "online learning" category). We achieved fourth place in a field of fifteen competitors in the equilibrium category and seventh place in the online learning category. In the equilibrium category, the top three competitors in the field: Hyperborean2007LimitEQ1 (University of Alberta), IanBotLimit1 and GS3Limit1 (Carnegie Melon University) managed to outearn PokeMinn, but our opponent model learned how to reduce losses over the course of a 3000 hand match. The results are shown in Figure 4.1. We now explore the development of PokeMinn and discuss how it was able to learn to improve its performance so quickly against these strong competitors.

4.1 Goals and Structure of PokeMinn

Unfortunately, we had no access to former competitors' agents, nor did we have a good understanding of what types of opponents we might face. To address these challenges we decided to take a two-phased approach: Build a strong static-policy agent, and then integrate a learning component that would allow the agent to adapt its behavior to start exploiting its current opponent as quickly as possible.

To develop the strong static-policy agent in the first phase, our team reviewed some of

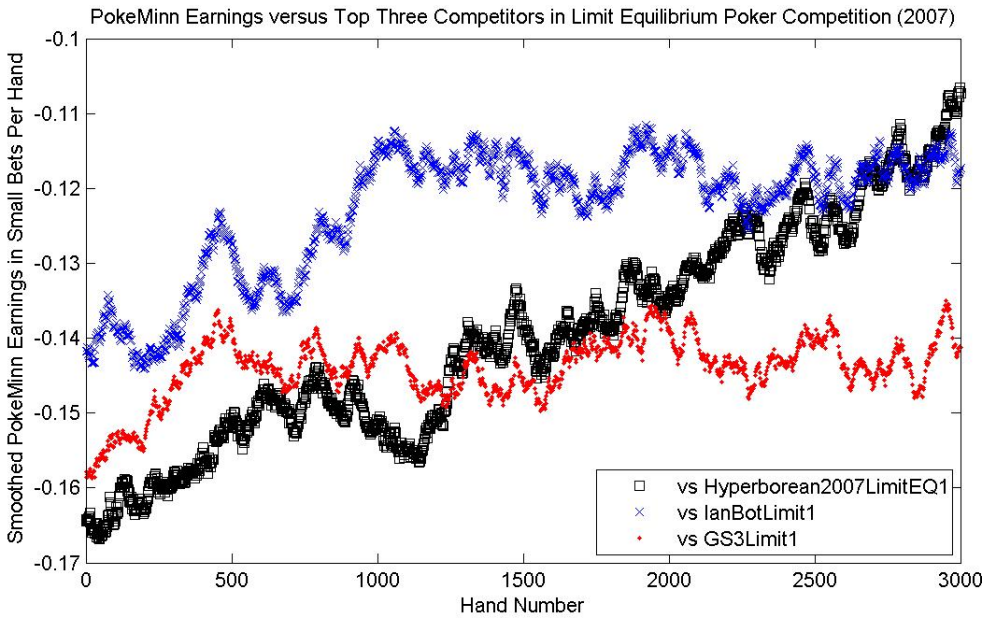


Figure 4.1: PokeMinn's Earning Rate versus the top three poker agents in the 2007 Computer Poker Competition at the Association for the Advancement of Artificial Intelligence Conference. While PokeMinn's earnings remained negative against these strong opponents, its loss rate decreased, indicating it was learning how to reduce its losses from observations of these opponents. Earnings rate is reported in small bets per hand and is moving-window smoothed (windowsize = 1500 hands) to reduce effects of the stochastic noise induced by the cards. These results are the average earnings achieved during 20 matches per opponent.

the ideas for agents used by other research groups, especially those developed by the University of Alberta Computer Poker Research Group [10, 11, 12, 14, 13, 25]. After the initial review, several of our team members independently developed agents and we held an internal competition to determine which of the agents was the strongest. We repeated this process over the course of several months until one of our team members developed an agent that no one else's agent could beat. This "best of breed" agent (Ox) then became our static-policy baseline. The interesting attributes of the static policy decision are discussed further in Section 4.1.1.

In the second development phase, we explored different aspects of the opponent that we could learn. Two learnable behaviors we considered were:

1. The relationship between an opponent's actions (A) and the actual value of his cards (S): $P(S|A)$.

2. The conditional probability that the opponent would take an action (A_{t+1}) given the sequence of actions that have occurred in this hand (A_s): $P(A_{t+1}|A_s)$.

Each of these learning efforts had drawbacks. For example, since we would only be likely to know for certain what an opponent's cards were when neither player folded and the opponent's cards were revealed on showdown, only a subset of the hands would lead to opportunities to learn the opponent's $P(S|A)$ model. Also, since the game has over 10^{17} game states, it would be very hard to map more than a tiny fraction of the opponent's behavior with only 3000 hands to use as training data.

We then realized that to be successful we would need to act under two constraints: ensure that *every* hand played could be used as an opportunity to learn, and choose an abstraction of the opponent's behavior that is simple enough that we could learn it within a small fraction of the number of hands in the match and still have time to exploit the opponent over the remainder of the match.

In human vs human play, people often categorize an opponent player's performance on two scales: "tight" versus "loose" (how many hands would they fold versus continue to play) and "aggressive" versus "passive" (how many hands would they bet or raise on versus how many hands would they check or call on). These categorizations can often help human players quickly determine a best response to the opponent archetype [69, 70]. We thought that an adaptation of this categorization that was rooted in probability would be a useful characterization to learn. We attempted to develop a model which gathered the information and was able to be easily integrated with static agent developed in the first phase. We generated a model of opponent behavior that met all of our requirements and could outperform the static model. The details of our final learning design are presented in section 4.1.2.

Once we had the learning component integrated with the strong static agent, it was time to refine and tune the model in preparation for the competition. Since we had no access to previous competitive agents, and there is no standard method for assessing an agent's ability to play poker outside of its relative performance with other agents, we borrowed a co-evolutionary technique to improve the agent.

In each trial, we compared performance of several candidate versions of our agent where either a parameter or function was changed. The value of each parameter or choice of which function implementation to use in a given role can be expressed in a

vector, and each candidate agent's identity is their corresponding vector. When we ran a tournament, we could determine which vectors performed better than their peers. Each vector setting from these candidate agents effectively represented a (possibly local) maximum performance point within a population of sibling agents. We then manually evaluated the results in a way that was roughly equivalent to searching in the parameter (or component) space so that we could find vectors of settings which we though might outperform their peers in a successive generation. After several "generations" of this co-evolutionary process, we gathered a set of the best performing agents from all of the previous generations. We then ran performance tests for these best performers to determine the ideal settings for the top two agents. These top two agents became our entries in the 2007 Computer Poker Competition: PokeMinnLimit1 and PokeMinnLimit2. The only difference between these versions was the value of a single parameter representing our belief about what all of the opponents raises during the hand were telling us about the strength of his cards in comparison to ours.

4.1.1 Static Components

There are two key static components used in PokeMinn. These components were adapted directly from PokeMinn's ancestor (Ox) and their parameters were tuned to improve performance using co-evolutionary techniques described previously. The static components discussed next provide the ability for the agent to estimate the likelihood that its cards are better than the opponents cards (relative hand strength) and the ability for the agent to estimate how much money would be won or lost for each possible choice of actions given the relative hand strength (conditional expected value).

4.1.1.1 Estimating Relative Hand Strength

We estimate the relative hand strength in two stages. First, we calculate *hand strength*: the probability our cards are better than the average cards the opponent could have. Then we perturb that probability value via a function that attempts to account for the opponent's actions during the hand. The result is a (usually lower) probability-like estimated chance of winning the hand, which we call *relative hand strength*. Relative hand strength looks like a probability value in that it is a real number on the range from zero to one (inclusive), but because it was a probability

that was 'perturbed' using a special function, we are choosing to refer to it as a strength rather than a probability in this work. In Section 4.1.1.2 we describe how we use this value as a weighting factor.

To calculate hand strength, we use an approximation technique for estimating the cards that might appear in the remainder of the current hand. The technique steps through each possible pair of hole cards the opponent could be holding (given what cards our agent is holding and what community cards are visible) and iterates through all remaining unseen cards in the deck (one at a time) to simulate the next rounds' community cards being dealt. Then, for each of these possible future circumstances, we determine whether or not the opponent's hand is better than our hand. By counting the number of wins, losses, and ties that occur in all of these possible future scenarios, we estimate our probability of having the better hand in the next round. We then use this calculation as an approximation of our probability of having better cards than the "average" cards that the opponent could have at the end of the hand.

While this technique is fast enough to perform in real time before each decision must be made (we never have to examine more than $\binom{50}{2}49 = 60025$ possible outcomes), this fast approximation is inaccurate in several ways. First, the approximation is weakest when we transition from the pre-flop to the flop: we should simulate the drawing of three cards instead of one. Second, since we are only simulating one future round instead of all of the remaining betting rounds, the calculation is only completely correct when we are in the turn, simulating the transition to the river. Thus, our approximation of hand strength is worst in the pre-flop, but gets better at the flop, and is completely correct on the turn and the river.

Once we have obtained the approximation of the probability that our cards are better than the average opponent cards we perturb that probability using a function of the opponent's observed behavior so far. Essentially, the more the opponent raises during his turn, the more the function reduces the probability that our cards are better. The rationale for such a probability reduction (of our likelihood of winning) is that in limit poker if a player has good cards, the player will (on average) tend to raise more often than if they do not: to do otherwise would lead to lower potential earnings - a player who raises too often with weak cards and calls too often with good cards will earn little when he wins a hand and lose a large sum when he loses. If the opponent raises, it probably means he has better cards than average - and more raises probably

means even better cards. Thus, the more the opponent raises, the worse our chance of having better cards is. But we also need to consider the non-linear relationship between the number of raises the opponent makes and the true quality of his cards: there is much more of a reduction in our probability of winning between the 1st and 2nd opponent raises than there is between the 4th and 5th opponent raises.

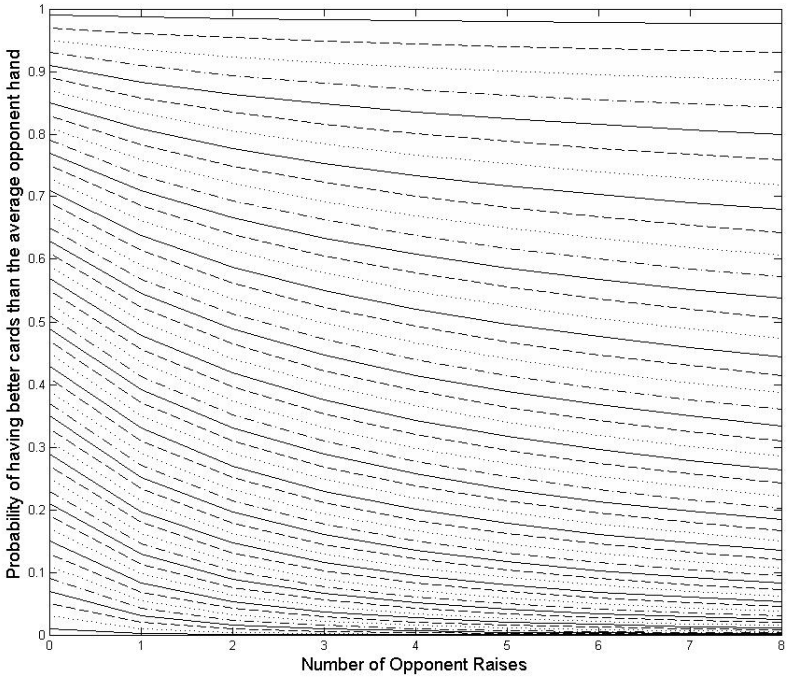


Figure 4.2: Effects of PokeMinn's hand strength estimation function as presented in Equation 4.1. To interpret the graph, select a probability of having better cards than the average opponent hand on the y-axis then follow the line until it intersects the vertical line extending from the number of raises the opponent has made (listed on the x-axis). The intersection is the calculated relative hand strength.

The perturbation function has several characteristics: it must not perturb the probability when the opponent has not raised; it should reduce the probability of our cards being better monotonically with the number of raises the opponent makes; the reduction should have diminished effects as the number of raises increases; the effects of the function should have less effect in the extreme regions (it doesn't alter our probability much when we have either very good cards or very bad cards, but has a large effect if we are in the middle). There are many families of functions which would fit this profile, and an infinite number of members of each family; we arbitrarily chose

a simple tuneable function which addresses the desired characteristics:

$$p^{\log(e+rf)} \tag{4.1}$$

where p is the probability that our cards are better than the average opponent hand, e is the natural log base (2.7182), r is the number of raises the opponent has made so far, and f is a tuning parameter for controlling how much each raise affects the function. The behavior of this function for various probabilities of having better cards than the average opponent hand and various numbers of raises is shown in Figure 4.2.

Our two agents (PokeMinn1 and PokeMinn2) differed only in the setting of the f parameter: PokeMinn1 (f=2.7) assumes that opponents will tend to be less aggressive and raise less often than PokeMinn2 (f=1). For PokeMinn1 we used co-evolutionary exploration to find a parameter setting for f that outperformed the majority of agents with other parameter settings in self-play. For PokeMinn2 we set f to 1 to preserve the original perturbation equation as it was in its ancestor (Ox) which had no tuning ability for the perturbation function.

4.1.1.2 Estimating the Conditional Expected Value of Actions

To optimally decide which action (fold, call or raise) would earn the most money from the opponent (or best limit our losses) given our cards, we could use a simple look-ahead procedure to recursively compute the future reward from each possible future action path to a terminal node in the game tree. Each terminal node in a 2-player limit poker game has a deterministic pot size, of which our agent has generally contributed half (unless one of the players has folded as a first move). This makes the possible earnings or losses easy to compute. Using some estimate of the probability of winning the hand (such as the relative hand strength discussed in section 4.1.1.1), we could compute the weighted earnings (or loss) at each terminal node. Using a fixed approximation of the opponent behavior (fixed opponent model), we could calculate the weighted gains or losses at each branch and accrue them backwards toward our current decision node to obtain the total weighted expected value of taking each action. Then we could select the highest yielding action and execute it (or perhaps use some stochastic selection process which favors the highest-yielding action - so as to keep the opponent from always knowing our true situation).

Unfortunately, this optimal process is too computationally intensive for real-time implementation beyond a one-round lookahead. For the PokeMinn agent, we instead look only one round ahead, treating the terminal nodes of that round as the terminal nodes for the remainder of the hand. While this implementation is real-time implementable and interleaves well with the one-round lookahead for estimating relative hand strength, it does tend to under-estimate the values potentially earned or lost in extreme behavior (such as a string of raises) which could lead the agent to call when a raise would have been more profitable (when we have better cards than our opponent) or raise or call when a fold would have led to less of a loss (when our cards are worse than our opponent's). Also, because the game tree is weighted based on a fixed model of opponent behavior (likelihood of the opponent raising / calling / folding), it generates a worse estimate as our opponent deviates from the fixed probability distribution we use to model his behavior.

Fortunately, learning the real distribution of behavior of the opponent is a relatively easy endeavor if we make several assumptions. In the next section, we discuss the learning components of PokeMinn, including its ability to learn a model of the opponent's behavior.

4.1.2 Learning Components

The objective of the learning component of PokeMinn was to learn to predict the behavior of the opponent and then exploit that behavior to extract money from him. Recall the two overarching constraints described earlier:

1. Ensure that *every* hand played could be used as an opportunity to learn.
2. Choose an abstraction of the opponent's behavior that is small enough that we could learn it within a small fraction of the number of hands in the match and then still have time to exploit the opponent over the remainder of the match.

We knew that if our opponents used online learning approaches (instead of game theoretic approaches) they would attempt to adapt to our behavior over the course of the match. We considered making an agent that paid more attention to the more recent behavior of the opponent using a sliding window or weighted approach that emphasized more recent opponent actions. Unfortunately there were several problems with any windowed or weighted approach:

- Since we did not know anything about our opponents, any window selection or weighting scheme choice may or may not be appropriate for a particular opponent, and any attempt to 'learn' the proper window size for a given opponent is a problem with even higher dimensionality (and less likelihood of success) than the original opponent modeling problem we set out to solve.
- The size of the window or weighting scheme would generate a learning bias that gave too much emphasis on our opponent's latest behavior and could lead to a potential exploitation if the opponent used a frequency of change that was of the same period with our window size or weighting scheme.
- These methods would violate constraint 1 because our actions in the later portion of the match would be based on only what we learned in a (possibly weighted) subset of the hands in the match. With a smaller number of examples to learn from, if the opponent was not adapting to us (such as a game theoretic opponent) or if the opponent was changing relatively slowly compared to our agent, the smaller number of training examples would lead to a larger error on our model's representation of the opponent.

For these reasons we decided to avoid a windowed or moving average scheme. Instead, we used a model that looked at the entire hand history of the match to determine behavior. While this biased us towards a slower response when the opponent actually did change, we hoped that we could mitigate the risk by learning a relatively small representation of the opponent that would hopefully learn faster than the opponent could adapt to us.

We wanted to learn a model of the opponent that would allow us to predict their behavior distribution in *circumstances which appear to us to be similar* to circumstances we've seen in the past. For example, if we've observed that on the flop, when the opponent raises as the first action, and we re-raise, we notice that the opponent folds 20 percent of the time, calls 10 percent of the time, and re-re-raises 70 percent of the time, we know that attempting to bully them out of the hand by re-raising is probably not a useful strategy.

In other words we wanted to predict the opponent's next action (A_{t+1}), given a sequence of betting events so far in the round (A_s), and all the prior behavior our opponent has exhibited (O_H) over past hands during this same betting round (as

represented strictly by publicly visible actions taken so far). Formally, we define the distribution of behavior as:

$$P\left(A_{t+1}|A_s, O_H\right)$$

We chose to make a separate model for the opponent's behavior in each round. While this effectively encodes an independence assumption between rounds and thus eliminates our possibility of transferring an estimation of the opponent's hand quality between rounds, it achieves two other important objectives:

- It reduces the dimensionality of the learning problem. Instead of collecting data for over 10,000 instances of possible opponent behavior sequences in the full hand, the 4 independent models simply capture data to populate each of the 19 possible outcomes in a betting round. This enables us to learn behavior with far fewer samples.
- It addresses the fact that in most hands (where neither player has cards that are good enough to make the effect of new community cards irrelevant) each new round's community cards may significantly alter the strength of the opponent's hand, causing him to behave differently than in the previous round - reinforcing the notion that independence between rounds is an acceptable assumption.

4.2 PokeMinn's 2007 Performance

In this section, we explore several features of the PokeMinn-class agent that was used in the 2007 Computer Poker Competition. In particular, we want to characterize its ability to learn: how fast can it learn and how much better can it perform than non-learning agents. To conduct these experiments, we created a second version of PokeMinn that has no ability to learn a model of opponent other than the original hand-coded version it has when a match started, as discussed in section 4.1.1.2. We then examine the differential performance between the learning PokeMinn and the non-learning PokeMinn as an indication of learning over the course of the match.

In addition to providing a characterization of PokeMinn's learning profile, these tests also have the capability to indicate potential undesirable behavior such as a condition

where the learning version of an agent has a worse performance against a certain agent than the non-learning version.

4.2.1 Performance Visualization

When examining the learning differential (earnings difference between a learning and non-learning agent against a specific opponent) in terms of performance improvement, we are faced with evaluating extremely noisy data. In each hand of the version of Heads-Up Limit Texas Hold'em used in the competition, an agent could win or lose to its opponent by up to 21 small bets per hand. Since we are subtracting the performance of one agent from the other agent, the range of this random variable is 42. In a match, even a fraction of a small bet per hand is significant in differentiating performance, we must find a way to remove the noise from the learning signal while simultaneously avoiding losing too much of the trend information. For this work, we use two methods to reduce the noise in agent performance data: double exponential smoothing, and summation over a growing window. Each of these techniques has benefits and drawbacks as we describe in Sections 4.2.1.1 and 4.2.1.2.

4.2.1.1 Double Exponential Smoothing

Double Exponential Smoothing (DES) is often used to smooth noisy time-series data. DES is preferred over Single Exponential Smoothing when the analysts believe that there is a trend in the data. In this work, we believe there should a trend in the data: as the learning agent learns while the non-learning agent does not, the differential performance should increase, exhibiting a trend (positive slope in the differential data).

We also chose to use DES because it provides a characterization of the trends in the data that are not biased by where the data is within the time series. Roughly speaking, if two distinct slopes in two different time regions are equal in shape and duration, then they indicate approximately the same (de-noised) sequence of events in the underlying time series data.

While DES does provide a non-time-biased expression of the underlying trends in the data, the limitation of DES is that the values it generates from a time series are not directly mappable to real performance within a single hand (or even a specific range of hands). Thus, while DES is good at expressing comparable relative increases and

decreases in the data, the specific values generated by DES have little meaning in the context of actual earnings within a poker match.

4.2.1.2 Normalized Cumulative Performance

We also explore normalized cumulative earnings over a growing window because it adequately provides what DES does not: a direct value of how much better (cumulatively) the differential performance is, giving us a notion of how much more could have been earned by a specific hand in the game if the learning agent was used instead of the non-learning agent. Formally, we define the normalized cumulative summation over a growing window for a game of total length h:

$$\forall n \in 1..h, \Delta E_n = (E_n^{Learning} - E_n^{NonLearning})/n \tag{4.2}$$

where n represents the hand number, E_n^x is the cumulative earnings of agent x as of hand n (which is computed as $\sum_{i=1}^{n} W_i$, the sum of the winnings of each hand W_i up to hand n). This calculation yields the performance difference in terms of the number of small bets per hand won per hand if the match had been stopped at that hand. We use this value to indicate how much better the learning agent performed than the non-learning agent at any point in time.

The drawback of summation over a growing window is that as the window grows, trend changes have a diminishing effect on the values it generates. Thus, this method tends to be very sensitive to trends in the early portion of a time series, while giving the impression that there is not much change happening in the latter portion of the time series.

4.2.1.3 Visualization and Interpretation of the Earnings Data

In order to get the best of both worlds (non-time-biased trend information and a meaningful estimate of expected value) at all points in the time series data, we combine these two noise reduction methods on a single graphical visualization which we call a *differential performance graph* (DPG)[1]. From such a graph, we can answer the basic question (using the summation over a growing window): When playing against

[1]See Section 4.2.3 for examples of DPGs of a poker-playing agent's performance

opponent agent O for a match of length h, by how many small bets per hand would the learning agent have beat the non-learning agent.

Furthermore, if we know that our target opponent O uses a fixed policy that does not allow his play to adapt to our agent, we can make certain additional claims when analyzing the DPG, since the only source of performance variance (other than noise) is caused by our learning agent. For example, given two versions of a learning agent (and two corresponding DPGs), the double-exponentially smoothed line can be compared across the two agents to determine the relative difference in learning rates between the two agents at various points during the game.

4.2.2 Baseline Learning Performance

We now analyze the performance difference between learning and non-learning versions of PokeMinn against several static-policy baseline agents: Always-Call and Always-Raise. Neither the Always-Call nor the Always-Raise agents would perform well in the competition: their lack of logic required to make a decision based on the probability of winning with the current cards makes their performance worse than almost any agent that actively decides the course of action based on the current hand. The reason we focus on these two agents is not to see how much better PokeMinn is than them, but rather to see how much it can learn from them when its learning is active.

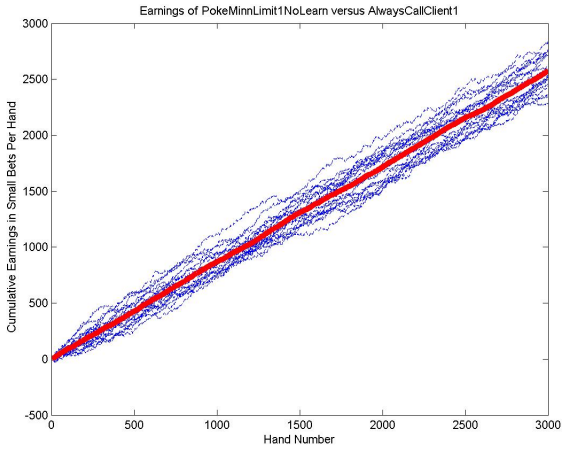
Ideally, if a human was playing against an opponent who always called or always raised, the human would quickly realize that the opponent was not paying attention to his cards or the human's actions and the human could use a play-for-value [69] technique to maximize his winnings. In other words, any use of deceptive play such as bluffing or slow play would be unwarranted because it had no effect on the opponent, and could only lead to a possible loss of expected value in the game. Given the correct human response to the Always-Call or Always-Raise player, we would hope that a learning agent such as PokeMinn could quickly learn at least an approximation of such a strategy.

We can test to see if what PokeMinn learns is acceptable by using differential performance analysis. Since PokeMinn-NonLearning is using a fixed deterministic strategy, the best performance it could achieve is if it just happens to be a perfect counterplay to either an Always-Call or Always-Raise player, but we know it cannot be both, and

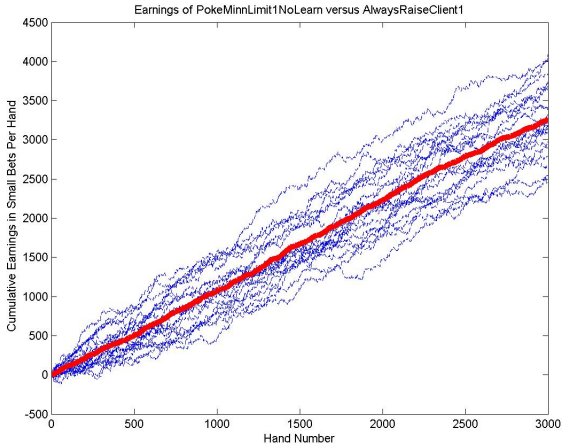
is more likely that it is neither. Thus, PokeMinn with the learning turned on should be able to perform at least as well as PokeMinn-NonLearning, and hopefully it could learn to play better over the course of a match. Ideally, with a fixed deterministic strategy opponents such as Always-Call and Always-Raise, PokeMinn should never under-perform PokeMinn-NonLearning. If it did, we have an undesirable behavior that needs to be fixed.

To perform this experiment, we executed 20 matches between the non-learning and learning versions of PokeMinn against the Always-Call and Always-Raise players. Each match ran for 3000 hands. In each of the 20 matches, the cards that each pair of opponents saw in any given hand was the same. This enables cross-player and cross-opponent comparisons with the data and eliminates the effects of card-noise from the differential comparisons.

We first examine the performance of the non-learning version of PokeMinn in 20 matches against a Always-Call and 20 matches against an Always-Raise opponent. The results, shown in Figure 4.3 reveal that there is higher variance in cumulative earnings when PokeMinn plays against the Always-Raise player than when it plays against the Always-Call player. This can be explained by the higher number of decisions that an agent must make when playing against the Always-Raise opponent: Only one decision per betting round is required when playing against Always-Call, but against Always-Raise, a maximal number of actions is possible, and PokeMinn has a larger number of decisions to make in each hand. Also notice that the non-learning version of PokeMinn has an overall worse performance against the All-Call opponent than it does against the Always-Raise opponent. While further analysis is beyond the scope of this work, we believe this performance difference can be explained by PokeMinn's treatment of an opponent's raise as an indicator of hand strength (as described in section 4.1.1). That component of PokeMinn's behavior is static, and was originally tuned to assume that an opponent would raise quite often, especially if it had good cards. Since the Always-Call opponent never raises, PokeMinn-NonLearning tends to under-estimate the true strength of the opponent cards, perhaps causing PokeMinn to raise or call when it should have only called or folded. When PokeMinn-NonLearning plays hands it should not have played, or perhaps bet too much when it should not have, its performance will be worse against the Always-Call agent. Another possible contributing factor is that the Always-Call agent limits its pot-exposure by never raising, thus making it harder for a player to extract money from it like it could



(a) Performance of non-learning agent against Always-Call opponent

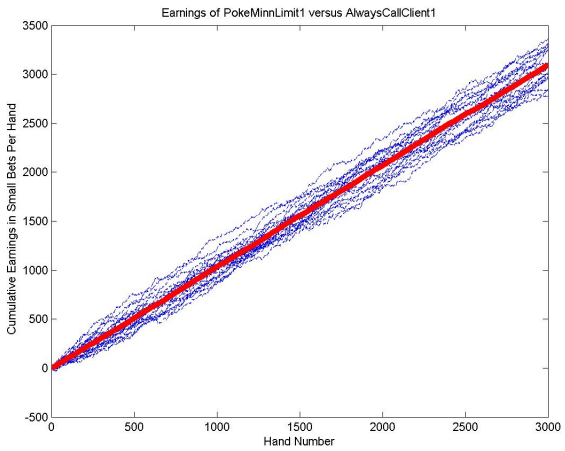


(b) Performance of non-learning agent against Always-Raise opponent

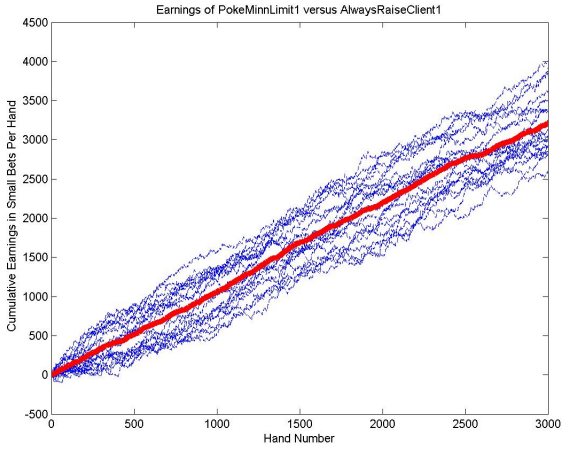
Figure 4.3: PokeMinn-NonLearning's performance against two different fixed strategies: When the opponent always calls, our per-hand winnings variance is lower (Figure 4.3(a)) than when the opponent always raises (Figure 4.3(b)). Dotted lines represent the cumulative earnings over each hand in individual matches, while the thick solid line represents the mean cumulative sum of all matches.

from Always-Raise.

Next, we examine the learning version of PokeMinn's performance against each of the two fixed-strategy opponents. The first thing we notice in Figure 4.4 is that performance between the learning version of PokeMinn and each of the opponents is much closer to equivalent. Furthermore, we notice that PokeMinn-Learning's performance



(a) Performance of learning agent against Always-Call opponent



(b) Performance of learning agent against Always-Raise opponent

Figure 4.4: PokeMinn-Learning's performance against two different fixed strategies: When the opponent always calls, our per-hand winnings variance is lower (Figure 4.3(a)) than when the opponent always raises (Figure 4.3(b)). Dotted lines represent the cumulative earnings over each hand in individual matches, while the thick solid line represents the mean cumulative sum of all matches.

against Always-Raise is not substantially different from PokeMinn-NonLearning's performance against Always-Raise, but that PokeMinn's performance against Always-Call is significantly better than PokeMinn-NonLearning's performance against Always-Call. This finding implies that PokeMinn's learning capability was able to make up for (at least partially) its built-in bias to mis-interpret the opponent's tendency to

always call as having weak cards.

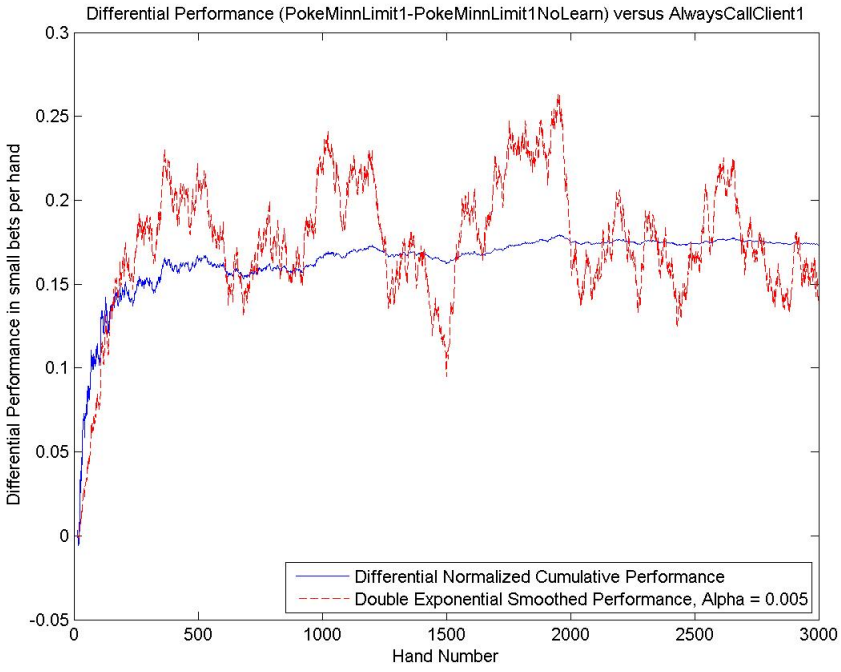


Figure 4.5: Differential Performance between PokeMinn-Learning and PokeMinn-NonLearning facing the Always-Call opponent.

One final aspect of the performance difference between PokeMinn-Learning and PokeMinn-NonLearning we can examine is the shape, or characteristic of the learning curve. Recall that the learning agent was able to improve its performance against the adversary that always calls. To visualize this curve, we generate the differential performance graph, as defined in Section 4.2.1.3.

Notice in Figure 4.5 that we gain the majority of our performance difference within the first 300-500 hands. After that point the performance difference begins to stabilize, implying that this learning agent has learned as much as it is going to learn from the Always-Call agent within the first 500 hands.

4.2.3 Learning Performance Against 2007 competitors

Among the matches against stronger opponents, our most compelling evidence of learning in the 2007 Computer Poker Competition is when PokeMinn faced the top

competitor: Hyperborean07LimitEq1[2]. Because the non-learning version of PokeMinn did not compete in the 2007 Computer Poker Competition, we recreated an extended competition on the official University of Alberta "Benchmark Server" to collect 100 matches of data using PokeMinnLimit1, its non-learning version and Hyperborean07LimitEQ1[3]. Figure 4.6 shows the differential performance graph for 20 games played against this Hyperborean07LimitEQ1. Notice that both the normalized cu-

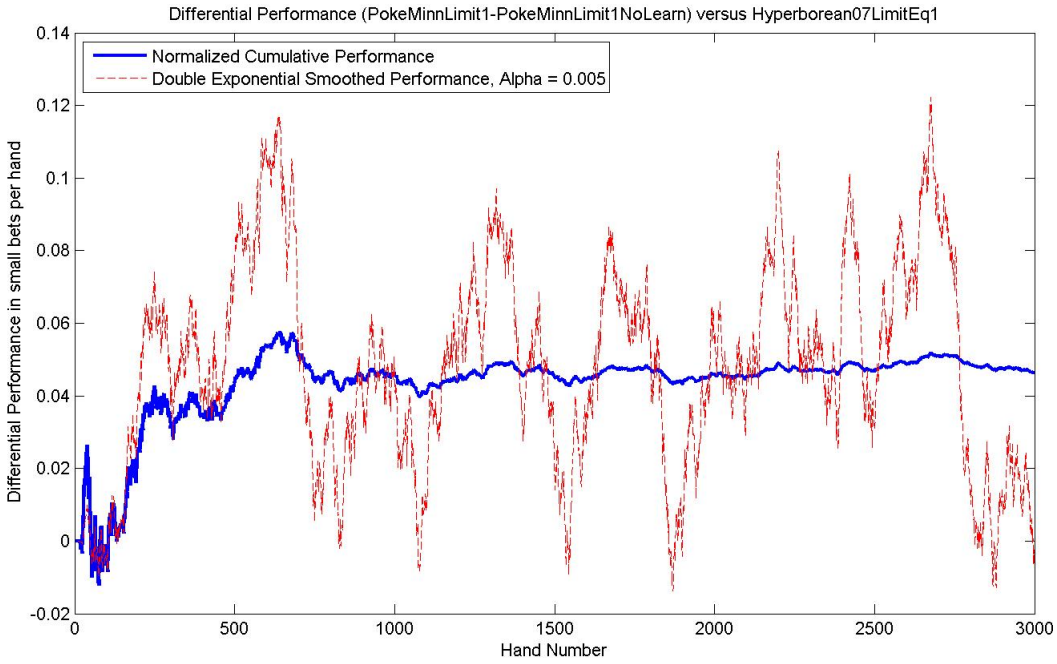


Figure 4.6: Differential Performance between PokeMinn-Learning and PokeMinn-NonLearning facing Hyperborean07LimitEq1.

mulative performance and the double exponential smoothed performance indicate the

[2]We chose to examine the differential performance between the learning and non-learning versions of PokeMinn against the Hyperborean07LimitEQ1 target because Hyperborean07LimitEQ1 has a known behavior each time it is restarted: Since it uses the same seed for its random number generator, we know that if neither the cardfile or the actions of Hyperborean's opponent differ between independent matches, then Hyperborean will take the same action in both matches. This allows us to determine the first hand for each game at which PokeMinn's behavior in active learning mode differs from the actions PokeMinn would take when the learning is turned off.

[3]The data was collected by playing matches between the learning version of PokeMinn and Hyperborean, and then repeating the identical match substituting the non-learning version of PokeMinn for the learning version. We subtracted the earnings of the non-learning player from the earnings of the learning player in each match to obtain the difference (which is an approximation for the value of learning over the course of the match. The benchmark server allows repeated experiments using the same sequence of cards dealt to a given player. In this way, we ensured that on each successive hand in the match, the cards PokeMinn-learning was dealt were also dealt to the non-learning version of PokeMinn.

learning version of PokeMinn is outperforming the non-learning version of PokeMinn by a decent margin after a few hundred hands. This observation implies that within a small fraction (10-20 percent) of the overall game, PokeMinn was able to learn how to improve its initial hand-coded model of the generic opponent to better predict the actual opponent.

While we can see evidence of learning by examining the differential performance between PokeMinn and it's non-learning sibling, when we restrict ourselves to examining total agent performance we cannot see how well the portions of the opponent model are doing, or discover why they may not be achieving their full potential. In order to understand the subtleties of opponent model learning, we must look deeper into the agent, at the quality of the predictions the opponent model is making. By comparing the "true" behavior of the opponent with the predictions that the model is making, we can ascertain the quality of learning at a much finer level. In Chapter 6 we showcase a novel technique and present a new measurement for doing so.

Chapter 5

A New Strategy Game

Real Time Strategy (RTS) games provide an interesting environment for analysis of the techniques discussed in this work. A real time strategy game allows players to test their ability to out-think their opponents in several ways. These games often involve exploration of unknown territory, for the purpose of locating resources (which can be used to produce buildings and units) and finding the enemies. When the enemies are located, the player must defend his buildings and resource gathering points while simultaneously trying to eliminate the enemy presence.

Unfortunately most commercially available real time strategy games provide no access to the underlying information available to each player. Furthermore, these games have a massive state space and action space which would preclude demonstration of significant empirical results without an extremely large sample of games.

In this chapter we present a two-player simultaneous-move turn-based game which has a comparatively small state space and action space that provides some of the key elements of a RTS game. The game we describe has stochastic results of interactions, and decision sets that have varying size depending on what actions were taken previously by the players. After presenting the game in Section 5.1 and 5.2, we discuss analysis techniques for finding equilibrium in this game in Section 5.3.

5.1 Game Overview

The goal of this game is to eliminate all enemy assets (enemy bases and units). Each player starts with one unit (a mobile military force) but may build additional units if they also own bases (stationary unit production facilities). During a turn a player may move any or all of his units from their current locations to any other connected locations or use units to build bases (which can then build more units). When a player's units occupy the same location as the enemy's units or encounter units when traveling from one location to another, a battle ensues, and at most, only one of the players will have surviving units. When a player's units are the only units at the location of an enemy base they may destroy the base (but may also lose units in the process). When one side has no remaining bases or units the game is over.

The game is played in a world comprised of several locations where players' units or bases could exist. Each location is identical and is fully connected to every other location. Each player simultaneously makes decisions on what to do with their units at the beginning of the turn. The players pass their decisions (known as orders) to the game server which then processes the orders and determines the results. The results of any interactions between the players units are presented to the players in the following turn, along with the status of the nodes in which the players have visibility[1].

In a turn a player must decide what to do with each unit he owns. The choices are:

- IDLE: Remain in the current location.
- MOVE(DESTINATION): Attempt to move to a different connected location. The move order includes the desired destination location identifier.
- BUILD: Attempt to build a base (which can produce more units). In order to build a base, the unit must remain at the same location and survive any enemy attacks during the turn. If the unit is still present at the end of the turn, the base is constructed.

[1]There are two versions of this game we present; they have different levels of visibility, one with hidden information and one with full information. In the hidden-information version, we employ what is known in the Real-Time-Strategy community as "fog of war" (no player has visibility what exists at a location unless he has a unit or base at that location), while in the full-information version the "fog of war" is removed, and the players can see the locations of each other's assets.

- DESTROY: If a player's unit is present at an enemy base, the unit may attempt to destroy that base. If the unit survives any enemy attack for the whole turn then the base will be destroyed, but there is a chance the unit will also be destroyed.

At any location where the player owns a base, the game server will produce a new unit each turn (assuming the player has fewer units than the maximum unit capacity γ). Players may wish to build bases and defend them during the course of the game because they are the only method of generating new units for a player to command.

5.2 Procedural Details

In this section, we use several terms to clarify explanations of game mechanics. *Red* and *blue* are used to describe the two players. A player's *assets* are his bases and units. If a player has assets at a location, the player is said to have *presence* at the location. If the player has a base at a location then he is the *owner* of that location. If he has unit(s) at that location then he is said to *occupy* that location. A player may both own and occupy a location. At any location at the end of the turn, only one player may own the location, and only one (possibly different) player may occupy the location.

We developed a computerized version of this game using a client-server approach. A game server keeps track of the true state of the world, determines what each player can see, and passes this information to the clients. The clients then make decisions on what actions to take and pass their desired orders back to the game server[2]. The game server processes the orders, handling any interactions between bases and units, and updates the state of the world prior to starting the next turn. If terminal conditions are achieved, the winner and loser (or tied players) are declared.

The game starts with each player having one unit. The units are randomly placed at two unique locations, and neither player knows where the other player's unit is. Once the game starts, the server processes the clients orders in the following sequence:

- Move Units

[2]Although the clients are queried in order, they only are able to see information they have access to, and no orders are executed until the server has received both player's orders. Thus the game is effectively a simultaneous action game.

- Generate New Units (at bases)
- Resolve Conflicts
- Destroy Bases
- Build Bases

When the all of the phases are complete, the server sends the updated state of the world out to the players and requests their next turn orders. This cycle continues until at least one player's assets have been eliminated. We now describe each phase of the turn in detail.

5.2.1 Move Units Phase

When a player orders one or more units to another location, the game server first checks to see if there were any intercepts. An intercept occurs between any two locations (A and B) when *red* has units at location A orders them to move to location B in the same turn that *blue* has units at B and orders them to move to location A. A battle is resolved (stochastically) between the forces according to the *rules of battle* (described later) and any remaining forces from the victor are sent on to their intended destination location. This simulates having encountered the enemy during the journey from the origin to the destination.

If there are no interceptions of a given move order, all of the units requested to move are relocated to their destination.

5.2.2 Generate New Unit Phase

After all move orders from both players have been handled, the server determines whether a new unit is generated at each base. In random order, a player's bases are selected (until all bases have been visited) in turn. The base generates a new unit for the player if that player's unit count is below his total unit capacity γ. If the the unit capacity limit has been met, no further units are built.

Any units built during this phase will be available during the next phase when conflicts are resolved. Note that since these units are generated after the orders for the turn

have been given, the newly-generated units cannot take any action during the turn, but they may act defensively during any conflicts occurring later in the turn.

5.2.3 Resolve Conflict Phase

At any location where both players have units present in this phase there is a potential conflict. If neither player is the owner of the location (but they both have units there) then a battle occurs and is resolved according to the *rules of battle*. The outcome of a battle will be that at most one player has units occupying the location, but possibly all units will be destroyed.

If one player is the owner of the location where the other player's units exist, then the owner may gain a defensive bonus. Any of the owner's units not moved during the current turn are considered to be defensive and gain a defensive bonus during combat (δ times their normal strength value).

5.2.3.1 Rules of Battle

At the start of the battle, all participating units from each player are identified. If one of the players has units with the defensive bonus, they are also identified. The battle commences and occurs over one or more *rounds* until at least one player's units are completely eliminated. If a player has any remaining units, that player becomes the occupier of the location where the battle occurred.

At the start of the round, each player's total strength is calculated as $N + (\delta)D$ where N is the number of units without the defensive bonus and D is the number of units with the defensive bonus. Then each unit present will be marked for elimination according to the following probability: $E/(F + E)$ where F is the total strength of the player's units and E is the total strength of the enemy's units. After all units have been checked, any units marked for elimination are eliminated from the battle. The process continues with new rounds until at least one player has had all battling units eliminated.

5.2.4 Destroy Bases Phase

To order one or more units to attempt to destroy an enemy base, all units in the order must occupy the same location as the enemy base at the start of the turn

(before orders are given). If any of the so-ordered units are still present after the conflict processing phase then the base is automatically destroyed. Also, each of the units used in the destruction attempt will be eliminated with probability $1/(A+1)$ where A is the number of units attempting to destroy the base. The more units used in a destruction attempt, the lower the probability that all units will be destroyed. If any units remain after the destruction of the base, then the player owning those units will occupy the location at the end of the turn.

5.2.5 Build Bases Phase

Any units ordered to build a base must not have moved or been given any other orders during the turn. If so-ordered units exist at a location where no base existed at the start of the turn then a new base is automatically built at that location during this phase.

5.2.6 Parameters

Here we list the current parameters applicable to this game as used for the analysis in the main body of the game description:

Parameter	*Description*	*Default Value*
l	Number of locations in the game	4
γ	Maximum number of units owned by single player	3
δ	Defensive strength bonus	1.5

5.2.7 Terminal Conditions and Score

The game continues over one or more turns (through all of the phases listed above) until at least one player has no remaining assets at the end of a turn. If neither player has any remaining assets then the game terminates with a tie (each player scores zero points). If one player has assets remaining when the other player has none then the player with assets receives one point and the other player loses one point. Thus, this game is a zero sum game.

5.3 Analyzing the Game

While the game appears to be relatively simple, the parameters described above allow for 2136 unique state isomorphisms in the full-information version of this game[3]. Of these, 115 are states with terminal conditions, 57 wins, 57 losses, and 1 tie for each player. The parameters also allow for 5 action options per unit (Idle, Move to one of three other locations, and, depending on whether there is or is not an enemy base or friendly base present, possibly build a base or possibly destroy a base). With a unit capacity limitation of three units, a worst-case scenario is when each player has three units to command, yielding a set of $(3^5)^2 = 59049$ unique joint actions if each unit is considered as a unique entity. Because of the limitation of only one player occupying a location at a time, if we consider unit action isomorphisms[4] then the number of joint actions is greatly reduced. The worst case scenario yields 2304 joint unit action isomorphisms while the average number of joint action isomorphisms per state is approximately 211. Fortunately, even with this rather large apparent branching factor, only one of the 2136 unique states will be arrived at from any state-and-joint-action branch. Unfortunately, the state which it arrives at is non-deterministic and is often based on the results of one or more stochastic processes occurring during the game turn.

If we consider the hidden-information version of the game where each player can only differentiate between states if he can see[5] a difference between the states, the game state is much more complex. For example, if the blue player has two units and a base at location A and no other assets, he cannot tell the difference between two states 1) in which the red player has only 1 unit at location B and 2) in which the red player has one base and one unit at each other node (B,C, and D). Clearly, the action set for the second condition for the red player is much larger than the action set for the first

[3]Because of the fully connected nature of the locations in this game, many physical states are isomorphic. For example, if only two locations have units present, then it doesn't matter *which* two locations those units are at, just which player occupies and how many units are at each location. State isomorphisms are important for reducing computational complexity because each isomorphism represents all of the states comprising it, and once a complex computation is performed for one member of the isomorphism, all other members can be generated by mapping the results from one state to the other without repeating the computation.

[4]In a unit action isomorphism, the unique identity of a player's individual units are removed such that if a player has two units at a location and unit 1 moves to another location, while unit 2 is idle, the decision is isomorphic to the condition where unit 2 moves to that location and unit 1 is idle.

[5]To "see" the enemy presence (units or bases), the player must have either one of his units at the enemy base, or have a base that is currently occupied by one or more enemy units.

condition for the red player. In fact, because the size of the action sets for the red player are variable in different members of a set of states which appear to be the same to the blue player, these state sets cannot be considered information sets. As a result, standard game theoretic techniques for equilibrium solving grow intractable because the state must be defined as the total history of all moves so far instead of just the latest unit and base allocations at the four locations. Because our to-be-presented methods require equilibrium calculations for baselining the performance of any given opponent modeling system, we choose to focus strictly on the full-information version of this game.

As we shall see, calculating an equilibrium, even in the full information version of this game is computationally expensive. The remainder of this section focuses on finding an equilibrium solution for the full-information version of the game to support computations described in later chapters.

5.3.1 Evaluating States

In order to decide what course of action a player should take, we would ideally like to know the value of each of the 2136 states to each player as well as the likelihood that each joint action would lead from one of those states to another. Armed with these values and probabilities, we could calculate an equilibrium solution for risk-neutral[6] agents.

Clearly the value of a terminal state to a player is equivalent to the score obtained in that state by that player. But when we try to evaluate non-terminal states, the actions which might lead to a terminal state are *joint* actions, and thus, while one player is desiring a certain joint action with a certain value, the other player may find it equally undesirable and attempt to avoid their component of the joint action. Furthermore, many of the joint actions have probabilities of leading to non-terminal states (or even back to the original state) creating potential non-terminated recursions in any algorithm attempting to recursively determine the value of a state. Since there can be sequences of actions by each player that yield cycles in the visited states, the game is not guaranteed to terminate. In general, this is an undesirable situation for game-theoretic analysis.

[6]Risk-neutral agents are ones who regard their game score as their utility function. Thus a risk-neutral agent in this game desires as a win equally as much as he despises a loss, and is indifferent when the game is tied.

We can, however, estimate the values of the states for each player using game theoretic techniques and a trick to effectively eliminate potential cycles within the state-action space. The basic idea is that we iteratively solve the full-information extensive form game by creating successively longer length games and determining the resulting value of each state using an equilibrium assumption about the style of the players. Before we present the algorithm, we provide additional rationale for its soundness.

Consider a one-turn probabilistic-outcome game where there are a number of states, each with a set of legal actions available to each player. Transition probabilities describe the likelihood of arriving at a destination state wherever a joint action (one action chosen from each player's action set) is executed. In the one-turn game we define the utility of each state based on whether it was a win, loss, or draw. A definitive outcome (when one player has assets and the other player has none) gives a utility of 1 to the winner and -1 to the loser. A draw[7] is worth 0 points for each player. In this one-turn game we can compute the equilibrium solution (a probability distribution over actions for each player, and their resulting expected utilities) for every state. If we consider the players to have played their risk-neutral equilibrium strategies in each state then this method would also yield the utility (for each player) for having that state as the starting state of a one-turn game.

From one of the player's perspectives, their minimum utility earned by playing their equilibrium strategy from within a certain state is an indication of how valuable that state is to them at the beginning of the one-turn game. Certainly some states are more valuable than others.

Now imagine a two-turn game where instead of setting all non-terminal state's values to zero, we use the values computed for the states during to the one-turn game as described previously. In this game, we are effectively first computing the value of the leaf nodes of an extensive form game (from the values calculated in the one-turn game) and backing them up the tree for a computation in the prior turn. This results (possibly) in a new equilibrium strategy for each player in each state, and also yields an updated utility to be earned by each player in each state.

Inductively, we continue to add "earlier" turns to create longer and longer n-turn games, using the values of non-terminal states calculated during previous $(n-1)$-

[7]In the one turn game, draws consist of both terminal states (where both player's assets are eliminated) and non-terminal states (where each player has some assets left).

turn games as proxies for the values of the successor states when a joint action is selected in the first turn of the n-turn game. At some point we will find that adding additional prior turns does not change the values of the states very much. When we reach the point at which the change in value for each state is below some tolerance threshold, then we can declare that we know the values of the game states when both players are playing the risk-neutral strategies - and thus we know what the equilibrium strategy and the utility for each player should be for every state in the game.

The overview of the algorithm is as follows:

- Determine the transition probabilities for each state and joint action to another state.
- Determine the terminal states (where there is a definitive win, loss or tie for the players).
- Initialize the appropriate value to every terminal state from the second player's perspective (+1, -1, or 0) and value all other states as 0.
- Repeat until convergence or for as many iterations as time allows:
    1. Determine which states have a positive probability of transition to a non-zero-valued state.
    2. For each of those states:
        (a) Generate a normal-form game zero-sum payoff matrix for that state by collecting the joint action payoffs (sum of probability-weighted values of the states transitioned to by each joint action.)
        (b) Solve the state's normal-form game (by calculating the equilibrium condition) and determine the value associated with the solution (from player 2's perspective.)
        (c) Set the state's value for the next iteration to the value calculated in the last step.
- Report the resulting values for each state.
- Report the equilibrium solution's recommendation for distribution over joint actions in each possible state.

To obtain the transition probabilities we play the full-information version of the game from every state with every possible combination of legal player orders, using Markov sampling to determine the probability that a joint action in the starting state leads to a destination state. Since both player's actions are fully specified in each case, the resulting transition probabilities are a function of only the rules of the game, not the strategy of the players. These transition probabilities are effectively a mathematical representation of the rules of the game.

To form the payoff matrix for each state and joint action, we look up the transition probabilities associated with each possible joint action, determine all the successor states and look up their values. For actions chosen by player 1 (i) and player 2 (j) the set of n associated transition probabilities $P_1 \dots P_n$, the associated states $S_1 \dots S_n$ and their values (from the second player's perspective) $V_1 \dots V_n$ we compute the payoff or utility u:

$$u_{ij} = \sum_{k=1..n} P_k V_k \tag{5.1}$$

To solve the zero-sum normal form game generated when utilities for every possible joint action have been computed from a given state, we describe the conditions of the game in terms of a constrained optimization problem and use a linear program solver such as `simlp` in MATLAB. The result is a probability distribution over actions for each player, and a value (from that player's perspective) for playing that strategy.

In the method above, we described how to strongly solve the simultaneous-move strategic game to generate an approximation of the strategy of a Nash Equilibrium player. This type of strategy becomes the basis for optimal play when the agent is uninformed[8]. In Chapter 7 we will use the Nash Equilibrium strategy as a reference point. We will then find the value improvement an agent could earn if he was informed about some aspect of his opponent with an opponent model.

[8]By uninformed, we mean a strategy which has no information about the specific opponent's characteristics. The uninformed agent assumes his opponent is rational and the equilibrium becomes desirable for describing strategies that guarantee some minimum value in the environment when all players are acting rationally.

Chapter 6

Opponent Model Performance

In this chapter we examine the factors which affect, and measurements which quantify the performance of an opponent model. In Section 6.1 we examine the conditions under which opponent modeling can be useful and describe a method for identifying one of these preconditions. We then discuss the desired traits for an agent and the models of its opponents in the adversarial setting in Section 6.2. We develop a flexible opponent model prediction performance measurement in Section 6.3 which becomes a building block for several additional capabilities. We present other entensions for the measurement such as detecting policy deviation and the measurement's applicability for metareasoning.

6.1 Conditions for Opponent Modeling

One question that is often overlooked, but should be asked prior to any attempt at modeling the opponent is "What can I hope to gain from this model?" We must ensure that the information we gain from the model will actually help us in our decision-making process. If the information to be gained by the model will not improve our ability to make decisions then there probably is no reason to build the model.

To make this point clear we contrast two different adaptations of the rock-paper-scissors (RPS) game played for C units of money each round. The two versions of RPS are the classic version Γ, and an adaptation with noise added to the output decision of our opponent Γ_p. In each game, the opponent pays us C if we win a round and we pay them C if we lose a round. In Γ, the opponent's choice is his action. In

Γ_p, nature adds noise to our opponents choice before it becomes his action such that there is a probability p that the choice the opponent makes will be converted into a randomly chosen action (with equal probability of selecting rock, paper, or scissors)[1].

Suppose that in both Γ and Γ_p, we have an opportunity to purchase information from an oracle who knows the opponent very well and can predict exactly what move the opponent will choose next. In Γ, knowing what the opponent will choose next will give us the ability to choose the best response each time. As long as we paid this oracle slightly less than C we would make money every time we played[2]. In Γ_p, however, there is a p probability that the opponent's choice will have no bearing on the action he takes. In a turn when nature added noise and perturbed the actual action, the oracle's information is worthless to us. The value of the oracle's information when the probability of noise is p over a large number of games is $(1-p)C$. As p approaches certainty (one) the opponent's choice is completely overwhelmed by noise, and the oracle's information becomes useless.

We reconsider these two adaptations of RPS in the context of the question "What can I hope to gain from this model?", and we realize that the game itself can have a large impact on whether or not an opponent model will be valuable. We make the observation that in general, *the more chance is involved in determining the outcome of an encounter, the less valuable an opponent model will be.*

Let's take a look at another example of a zero-sum game. This time we explore a game that is unaffected by chance and see whether opponent modeling is valuable. In Tic-Tac-Toe, the outcome of the game can be easily computed from a minimax[3] search of the game tree. Since the game conditions allow us to make a best response to the current board conditions, we can always select the best move for the current board state. If both players take their moves from the set of best moves the game will always end in a tie. Since the "correct" strategy is easily computable, and two players always making correct moves leads to a tie, the game is considered "solved" and strategically uninteresting for computers or people who know the correct strategy.

[1]The savvy reader may recognize that this adaptation essentially adds a chance that our opponent will sometimes play the mixed-strategy Nash Equilibrium from the Γ game instead of their original decision.

[2]Since we could earn C each time we used the oracle's information, the information is worth C per turn.

[3]Recall from the section on related work that minimax computes the equilibrium strategy for the game tree, assuming that each player will chose the best action from that node forward.

From this example, we observe that in a zero sum game, where both players know and have the ability to play according to the equilibrium strategy *if the opponent is using the equilibrium strategy, an opponent model cannot provide useful information that could help us improve our performance beyond what could be achieved when we also adopt an equilibrium strategy.*

But let us consider the case that our opponent is known to not always use the minimax-prescribed strategy and instead occasionally makes a predictable mistake. Would an opponent model help us in this case? Suppose we could purchase information from an oracle which revealed what the opponent would do in response to an action we were considering[4]. Using this information, we could play out a simulated game where we try each possible action and we consult the oracle to determine the opponent's response. With this method, we could recursively determine which player would win for each of our original action choices. If the opponent was not using the minimax method for every decision, then the oracle-simulation method might reveal a path for increasing our score in the game. We could discover the conditions under which the opponent's decisions would be exploitable, allowing us to win in certain circumstances. But if we are using the minimax method, it may also reveal a path for increasing our score, since this is possible due to the opponent's mistake. So what good is an opponent model here?

Let us consider a different game similar to tic-tac-toe, but with a larger board. There are two players, the equilibrium player, E, and his approximate-equilibrium opponent, A. E has access to an online minimax calculation that can always choose the optimal move, given that his opponent also chooses the optimal move. A has access to some pre-determined strategy ruleset which is mostly based on a pre-computed minimax solution to the game tree, but is incorrect in several game states which are not encountered as long as both players stay on the minimax path of the game. Thus, A has a fairly good abstraction to an equilibrium strategy. Fortunately, for A, if E is playing minimax, the decisions A makes are optimal, and the game will end in a tie. However, if A is playing an opponent that is not playing minimax, A may find itself in nodes of the game tree not covered by its approximation of the equilibrium strategy. In these circumstances A does not always make the correct decision, and is thus exploitable.

[4]This type of oracle is equivalent to an oracle that reveals the opponents *policy*: What action will the opponent choose in a given state of the world

Suppose another player M had access to an oracle which would predict the opponent's actions for any state of the board: this oracle knows the policy of A. Now suppose that instead of using a traditional minimax algorithm, M used some form of opponent model search where the game tree is recursively searched and M chooses his action with the highest value, where M's belief about A's move in any state is determined by consulting the policy oracle. With this method, M can use the recursive game tree search to discover the flaws in A that would not be encountered when A was playing E. In fact, if the flaws in A are sufficient, M can use this procedure to discover a means to beat A, even when E could not. From this example, we observe that *a model of an opponent could be useful when the opponent is known to be using an approximation of an equilibrium policy.*

From these examples we derive the two favorable conditions for valuable opponent modeling: the opponent's decisions must have some effect on the outcome of interactions, and the opponent must not be using a perfect equilibrium strategy. When deciding on whether or not an opponent model would be valuable we must consider both of these conditions.

Another way to think about the first condition is that any opponent model must be able to make predictions of the opponent's behavior. When the opponent's decisions have some effect on the outcome of the game, there is an opportunity for the predictions to be useful. If the balance is shifted towards a game of pure chance, there is less interaction between the players and the setting devolves to an equivalent set of games where each player is playing solely against nature. In such a setting opponent models provide no additional information about the outcome of the game.

Let us focus on the second condition: deviation from a perfect equilibrium strategy. We examine non-equilibrium play in the context of iterated rock-paper-scissors. Since the mixed strategy equilibria for the game is to choose equally amongst the possible actions, if we have an player A that instead chooses to play some other strategy that is similar but not exactly equilibrium (such as playing rock with probability r where $r > 1/3$ and splitting the remaining probability mass $1-r$ equally amongst paper and scissors) we can state that the value of knowing the policy of player A increases as r deviates from $1/3$. In the extreme case of $r = 1$ we can win every game since player A continues to play rock forever. From this example, we observe that *the further our opponent deviates from the equilibrium strategy, the more we could hope to earn from*

him if we have an ideal exploitation strategy.

Clearly, always playing rock in rock-paper-scissors is not rational. Why would a player act irrationally?

For this work, we consider three instances in which an opponent may appear to act irrationally in adversarial settings:

1. When the opponent does not have the capability or time (computational power) to determine the optimal action within the game.
2. When there is information the opponent wishes to obtain, the opponent may take sub-optimal exploratory actions to obtain the information.
3. When the opponent is trying to purposefully deceive us, hoping that we behave in a way that allows him to exploit us at a later point in the engagement.

To understand how these circumstances can affect rationality of decisions, we consider a game that is much more complex than rock-paper-scissors. Texas Hold'em poker is a zero-sum game that has far too many states and actions for today's computers to fully calculate the equilibrium strategy [13, 41], although approximations of the equilibria may inform player strategies [66]. In the strategically rich domain of poker, we can examine three cases which illustrate the apparently irrational behaviors described previously.

In one of the simplest forms of Texas Hold'em: two-player limit-bet, there are over 10^{17} states that the game could be in. Even if we had a computer that could calculate a very good approximation of an equilibrium strategy for the game, there is no guarantee that our opponent has access to the same. Since a player can only act as rationally as their computations will allow, their actions may in fact be irrational with respect to the true equilibrium play in the game. This is one example of the notion of *Bounded Rationality* [68].

The second form of apparent irrational behavior arises when the agent attempts to make decisions based on information not available to them. In card games such as poker where we have a private hand, there is hidden information that our opponent cannot obtain directly. Under these circumstances, the opponent may take actions which attempt to interrogate us, (such as calling a bet with very weak cards) in order

to get us to reveal something about the cards we hold[5]. This exploratory action may appear to us as irrational if we later find out that the cards the opponent held were worthless. When we examine the opponent's action from two different viewpoints, the action appears both rational and irrational. From a local viewpoint, the opponent is apparently spending money just to see what our cards are. Yet, by seeing which cards we are willing to play with over the course of several hands, the opponent may be trying to ascertain a more global understanding of our style of play - an understanding that may well be worth the amount he paid for it when he lost money on those exploratory actions. For this work, we define this behavior as *Contextual Rationality*: The action may be rational under some contexts (the opponent's desire to know the value of a hand we are willing to play) but not in other contexts (by making the bet, he was committing to a negative expected value action).

The third apparently irrational behavior is actually not irrational at all. It is a method of deceiving us such that we may be exploited. For example, in poker, when the opponent makes a few very high bets with very weak cards early in the game, he may win some hands that he would have otherwise lost because we folded thinking he had good cards. After a few instances of this, we may begin to believe that he is very loose (willing to stay in the game with even a weak hand) and aggressive (betting more than the value of the cards). Since we believe that the opponent is willing to bet on anything we begin to "call his bluffs" (bet more with weaker cards since the expected value of winning a hand has gone up). At this point our opponent has achieved his objective: to make us believe he is loose and aggressive. If he then tightens up his game and bets high only when he has good cards, he can possibly deceive us again if we still believe he is bluffing. Because we believe he is betting high frequently, we may call his bet with mediocre cards and end up losing. It may take us a while to change our belief that he is a loose aggressive player to match his true play of tight-aggressive. We define this behavior as *Exploitative Rationality*

While the example of bounded rationality described earlier demonstrates that if we have an opponent model we might be able to exploit the bounded opponent by deviating from an equilibrium strategy, the exploitative rationality example shows us that by deviating from the equilibrium strategy, we ourselves become open to exploitation. The contextual rationality example is more subtle: if we try to exploit

[5]Although the action of calling with very weak cards could also be considered a bluff, we are focusing here on the player who uses this action with the intention of obtaining information.

the contextual opponent we may gain in the short term, only to lose in the long run when the opponent's model of us is better than ours of him.

Another requirement for an instantiated opponent model to be useful is an opportunity to learn about the opponent. Even if the conditions of non-equilibrium play described above are met, if we only interact with an opponent once and we have no prior observations of his behavior, we do not have an opportunity to exploit him with the knowledge we gain from the single interaction. Thus, in order for a model to be useful, we must have either access to prior observations of the opponent's behavior, or the promise of repeated interaction with that opponent. Either of these conditions will allow us to form a more accurate model of the opponent's behavior, assuming that the opponent's behavior is relatively stationary or can be parsed into relatively stationary regimes.

We know that if we could determine if the opponent was not following an equilibrium strategy we might be able to exploit him. The next question we address is how do we determine when the opponent is not following an equilibrium strategy? By using an observation-based model of the opponent's behavior, we may be able to detect non-equilibrium play under certain conditions.

In a two-player zero-sum game, there is always at least one Nash Equilibrium. The equilibrium describes the actions which each player should take to avoid being exploited. This equilibrium may be a mixture over pure strategies[6]. If we are able to determine the Nash Equilibrium and the associated strategy for our opponent, then we should be able to *detect* certain kinds of deviations.

By definition, when they are following an equilibrium policy, the opponent's distribution over actions for a given state must be fixed. If we can observe a large enough sample of the actions the opponent takes from a given state, we can determine *if the opponent is not using an equilibrium policy* for that state when one of two conditions occur:

1. The observed distribution is different than the distribution(s) associated with the equilibrium.

[6]In effect, a mixture of pure strategies can be expressed as a probability distribution over actions such that the agent chooses one action from the action set randomly, according to the distribution.

2. The observed distribution is drifting over time (across successive sets of observations).

When either of these conditions occurs we know that the opponent is exploitable and that our opponent model, if used properly, may inform us of how to exploit the opponent. The details for detecting differences in distributions will be presented later in the chapter.

While detecting a distribution difference is a powerful method for determining when the opponent is *not* using an equilibrium policy, it has a limitation in that it cannot be used to detect when the opponent *is* following the Nash Equilibrium. In other words, while a distribution difference implies that the equilibrium *is not* being used, the lack of a difference does not imply that the equilibrium *is* being used. Another way of characterizing this detection method is that it has no false positives (stating the opponent is deviating when he really is not) but potentially many false negatives (failure to detect some occasions when the opponent deviates).

Consider playing rock-paper-scissors against these opponent players: N plays the game by randomly selecting an action (rock, paper, or scissors) with uniform probability. S plays the game by repeating the sequence rock-paper-scissors-rock-paper-scissors... Both of these players have the same distribution over actions $[1/3, 1/3, 1/3]$, but only player N is actually playing the Nash Equilibrium. Player S is clearly not playing the equilibrium (because it is not randomizing) and is trivially exploitable by a sequence-following prediction device. If our sole detection of exploitability lied in the observation of the distribution, we would have missed an easy opportunity to improve our outcome.

In previous paragraphs we have discussed several conditions required for opponent exploitation to be possible. We have examined some of the reasons that an opponent may take actions that appear to be suboptimal. We hinted at one method for detecting some occasions when an opponent would be exploitable and we showed that if we chose to attempt an exploitation of him, we open ourselves to exploitation. In the next section we shall examine some of the traits that a good agent should have in order to maximize its ability to exploit the opponent and minimize the opportunities for the opponent to exploit it.

6.2 Desired Traits of Adversarial Agents and Models

Given that the opponent may be executing a non-equilibrium strategy and that in order to know how to exploit him, we must first learn what his strategy is through observations, we would like to find promising existing observation-based learning algorithms. Much of the thinking on desired traits of learning algorithms in computer science to date have focused on learning stationary problems. Traditionally, researchers developed measures of convergence rates, examined error curves of the predictions and tried to make assessments of algorithms based on these characteristics. These methods were probably chosen because they were used in many other areas of machine learning where researchers needed to know how good an algorithm was in comparison with its peers. We have been conditioned to look at learning error-rate graphs to boost our faith that our algorithm is working. While these methods are well suited for assessing algorithms in static environments, they are fundamentally flawed when used in the presence of non-stationary learning problems.

Unfortunately, in the adversarial domain, we cannot guarantee a stationary opponent. In fact we can assume a non-stationary opponent for several reasons. First, any opponent who is playing a non-learning, non-equilibrium strategy would either:

1. Prefer that we do not discover his policy because if we do we could exploit him.
2. Desire that we discover his current policy in order to trick us into playing a counter-policy which he can then exploit.

In either of these conditions, if the opponent believes we have knowledge of his policy, he has an incentive to change it (in order to avoid exploitation in the first condition, or trigger the trap in the second). Even when we don't know what the opponent's policy is, he may have incentive to keep it from being stationary because a stationary policy is easier to learn (within the same number of samples) than a non-stationary policy. Finally, if the opponent is using a learning algorithm in an attempt to learn our policy, his policy will inherently change over time as he learns more about us.

Given these arguments, we can assume that a strong non-equilibrium-playing opponent will change his policy over time. Concept Drift is the term used to describe a phenomenon that changes over time. There are several algorithms that have been formulated to deal with concept drift. For example [43] uses an automatically-sized

windowing scheme to augment fast decision-tree techniques outlined in [26], enabling it to handle a domain with concept drift. Most of these techniques focus on window-based or weight-based schemes where more recent observations are believed to be more important than older observations. While we do not focus on further expanding these algorithms for use in opponent modeling, the methods we discuss later in this work are flexible and compatible with a concept-drift paradigm.

Though Bowling and Veloso's properties of rationality and convergence [18] remain important when the opponent's policy is stationary but unknown, we believe that Jensen's desire that the modeler have the ability to quickly track non-stationary opponents [44] may be even more important when the opponent policy is non-stationary. Ideally, an adversarial agent should be able to quickly determine what policy the opponent is using, and develop a counter-tactic, even while the opponent's policy is drifting over the course of the engagement. The fundamental building block for the opponent model's contribution to this effort is the quality of its predictions. We propose a novel measure of prediction quality in the next section.

6.3 Measuring Prediction Performance

In order to make good decisions, agents must have strong underlying models of their opponents. If we cannot successfully predict or explain the adversary then we are unlikely to intentionally choose a good response. If we are performing well despite a bad opponent model then either our decisions were accidentally fortunate or the opponent is sadly unfortunate.

When we endeavor to measure the quality of a model's ability to predict the behavior of the opponent we must be careful to isolate the measure of the prediction made by the model from the measure of the outcome which occurs as the result of the agent's interaction. The outcome depends on not only the model's prediction, but also the way the agent uses the prediction, the selection of a particular action, and the interaction between agent, opponent and environment when the actions are taken. Clearly, the outcome is only partially dependent on the prediction, so we should avoid attributing a completely causal relationship and assuming that outcome quality is indicative of prediction quality.

There are two main classes of measurements for quality of a prediction: measurements

based on accuracy and measurements based on distribution-similarity. The core of the measure based on accuracy measure is how many correct predictions are made compared to the total number of predictions. When the prediction is composed of a single most-likely action that the opponent will take, accuracy measurements may be the only reasonable measurement. When the model predicts a probability distribution over actions, however, distribution-similarity becomes a potential metric. Distribution-similarity is rooted in information theory, and describes a measure of entropy between two distributions. One way of thinking about distribution similarity is that it quantifies how much more information would be required to create one distribution from the other. The smaller the value, the closer the distributions.

Both accuracy and distribution similarity serve an important role. We do not discuss the accuracy-measure further because it is well documented in literature (for example, [16, 23]). The distribution-similarity measure however is not used widely, so we provide important additional justifications for using it when accuracy-based measures are inappropriate. Additional justification for using distribution-similarity will be revealed later in the chapter as we explore applications for the measurement.

First, if the opponent is truly selecting *randomly* from a distribution (as in a mixed-equilibrium strategy), the distribution-similarity measure will be the best measure of our prediction quality since the accuracy measure cannot quantify performance as better than h/t and worse than l/t over a large number of observations t where h is the count of the highest occurring action in the distribution, l is the count of the lowest occurring action in the distribution. For example, a prediction device attempting to predict the action of player N in the rock-paper-scissors game can do no better (or worse) than to achieve a score of $1/3$ in the limit. Any claims to the contrary would be statistically insignificant. When measuring accuracy of any prediction device on player R (1/2 rock, 1/4 scissors, 1/4 paper), the accuracy will fall somewhere on the interval [0.25 - 0.5]. This is true no matter what predictions the device makes.

Second, we cannot elegantly apply the accuracy-based measurement as a tool to determine when an agent is not following a mixed Nash equilibrium strategy, whereas the distribution-similarity measure is perfect for this function. Any statistically significant divergence between the observed distribution and the known Nash Equilibrium distribution signals that the agent is not playing the mixed equilibrium.

6.3.1 Desired properties of a performance measurement

Before we present the measurement, let us first take a moment to formally list the desiderata of a predictive performance measurement. Given a *context* (a set of target agent decisions of interest), an actual behavior distribution (a probability distribution over a finite number of possible actions[7],) and a prediction of the target's behavior distribution in the context, we define *prediction divergence* $\mathcal{PD}(P, Q)$ as a scalar measure of the distance between the actual distribution P and the predicted distribution Q such that $\mathcal{PD}(P, Q)$ has the following properties:

1. $\forall P, Q \quad \mathcal{PD}(P, Q) = 0 \Leftrightarrow P \equiv Q$. The function value should be zero if and only if the predicted behavior distribution is the same as the actual behavior distribution. This is a necessary condition for the function to be used to measure error or loss.

2. $\forall P, Q \quad \mathcal{PD}(P, Q) > 0 \Leftrightarrow P \neq Q$. The function value should be greater than zero if and only if the behavior distribution is different than the predicted distribution. This is a necessary condition for the function to be used to measure error or loss.

3. $\forall P, Q \quad \mathcal{PD}(P, Q) = \mathcal{PD}(Q, P)$. The function is symmetric. There is no sensitivity to which distribution represents the truth and which is the prediction. This is a necessary condition for the function to be a metric.

4. $\forall P, Q, R \quad \mathcal{PD}(P, Q) + \mathcal{PD}(Q, R) \leq \mathcal{PD}(P, R)$. The function obeys the triangle inequality, enforcing the logical understanding of "distance" in metric space. When combined with the previous three properties, this property completes the characterization of the function as a metric. While not essential for the work presented here, we require this property to facilitate future work in the area.

5. $\forall P, Q, R, S \quad (\mathcal{PD}(P, S) > \mathcal{PD}(P, R)) \wedge (\mathcal{PD}(P, R) > \mathcal{PD}(P, Q)) \Rightarrow (\mathcal{PD}(P, S) > \mathcal{PD}(P, Q))$. The function values obey the transitive property. Distributions which are further apart should have a greater function value than those which are closer together[8].

[7] While we describe a finite number of actions here, the functions presented later are applicable to continuous distributions as well.

[8] The notion of "further apart" in distribution terms can be expressed with an example. Consider a fair coin with distribution $P(heads) = 0.5$ and $P(tails) = 0.5$. This coin's distribution is closer to

6. Given any *extreme*[9] distribution P and any of its distributional complements[10] P^{-1}, $\mathcal{PD}(P, P^{-1}) = C$ where C is a positive constant. The function values should be bounded by a fixed constant such that any two distributions which are maximilally far apart[11] are bounded. This enables the function to be scaled by dividing by C such that the function values lie on the interval $[0, 1]$.

7. $\forall$ extreme P, Q, and their respective complements P^{-1}, Q^{-1} $\mathcal{PD}(P, P^{-1}) = \mathcal{PD}(Q, Q^{-1})$. All function bounds are equal regardless of the width of the underlying distributions. This is a minor extension of the previous property which allows heterogenous nodes to be weighted and summed in mathematically meaningful ways. If the function yielded differing values for distributions of different widths, it would be impossible to compare prediction qualities at two decision nodes where the number of actions to choose from differed.

6.3.2 Prediction Divergence

We now develop the measurement function characterized by the desired properties described above. Information theory provides tools for comparing distributions. Its tools are backed by proven theory and have been in use for well over half a century. Relative entropy (also known as the Kullback-Leibler Distance) between two probability mass functions [22] is a very widely used information-theoretic similarity measure. Given two distributions p and q, *relative entropy* is defined as:

$$D(p\|q) = \sum_{x \in \chi} p(x) \log \frac{p(x)}{q(x)} \tag{6.1}$$

While relative entropy is characterized by certain desirable properties such as equality, non-negativity, and transitivity, unfortunately it is not symmetric (Property 3) or bounded (Property 6). We instead choose a related, but lesser known similarity measure which has the additional properties of being both symmetric and bounded, as discussed in [30, 79]. This measure is known as Jensen-Shannon Divergence (*JSD*)

a loaded coin with distribution $P(heads) = 0.4$ and $P(tails) = 0.6$ than it is to one with distribution $P(heads) = 0.1$ and $P(tails) = 0.9$.

[9] An extreme distribution is on in which one of the elements contains all of the probability mass and the other elements all contain no probability mass.

[10] We define a distributional complement for an extreme distribution as another extreme distribution where all of the probability mass is located at a different point than in the original distribution.

[11] A loaded coin with $P(heads) = 0$, $P(tails) = 1$ is maximally far from a loaded coin with $P(heads) = 1, P(tails) = 0$.

and is sometimes referred to as Capacitory Discrimination. *JSD* is calculated between two distributions p and q as:

$$JSD(p,q) = D\left(p\|\frac{(p+q)}{2}\right) + D\left(q\|\frac{(p+q)}{2}\right) \tag{6.2}$$

One can think of *JSD* as the average relative entropy from each distribution (p and q) to the distribution that is midway between them. The further the distributions are apart, the higher their *JSD* value. In addition to keeping the desired properties from relative entropy, the square root of *JSD* is a metric [79], meaning that it covers the first four properties above and provides additional validity for its use as a distance measurement. For the remainder of this work we will use the term *Root-JSD* to represent the square root of the value calculated in Equation 6.2. By letting one of the distributions represent the predictive distribution produced by the model and letting the other distribution represent the true behavior of the target agent, we can measure the predictive quality of our model using *Root-JSD*.

6.3.3 Weighted Prediction Divergence

For any given interaction between agents, we may have many pairs of predicted and actual behavior distributions that need to be compared using *Root-JSD*. For example, in extensive form games such as poker, a model might be based on a portion of the game tree which contains a set of the opponent's decision nodes (Ω). At each node ($k \in \Omega$) where the opponent has an opportunity to decide his next action there exists a distribution over his possible actions. Some nodes may be occur more frequently or may be worth more (in terms of expected value). When developing a prediction quality measurement we would like to entertain the concept of weighting the prediction quality at each decision node by its importance in our decision-making and then combining the weighted prediction quality for all the nodes of interest.

In order to measure the overall similarity between the set of predicted conditional probability distributions (P) and the set of true conditional probability distributions (Q) in an interaction, we define a function that combines all of the individual *Root-JSD* measurements at each agent-decision node in the set, using a weighting function for each node. Given a set of opponent decision nodes (Ω), associated sets of predicted and true probability distributions $\forall k \in \Omega,\ P_k,\ Q_k$, and a weighting function for each

node $\forall k \in \Omega,\ W(k)$, we define *Weighted Prediction Divergence (WPD)* as:

$$WPD(\Omega) = \frac{\sum_{k\in\Omega} W(k)\sqrt{JSD(P_k, Q_k)}}{\sum_{k\in\Omega} W(k)} \tag{6.3}$$

6.3.4 Weighting Schemes

Equation 6.3 yields a value in $[0, 1]$ making it very useful for comparing overall prediction quality of models for agent interactions. If the set of decision nodes can be partitioned into contexts (abstractions which map many nodes into one context), we can also weight the individual decision nodes within the context group according to some function of their *importance*. Some possible weighting functions and their rationale for use include:

- Uniform Weighting. When there is no justification for weighting prediction nodes differently they are all given a weight of one.
- Frequency Weighting. When decision nodes have different frequencies of appearing in an interaction, it may make sense to weight the nodes by their frequency of occurrence. This makes the quality assessment sensitive to decisions that have to be made more often.
- Utility Weighting. Some decisions are more valuable than others: the outcome of a valuable decision leads to more increase or decrease in utility for the modeling agent. In these cases it may be desirable to weight the nodes by a function of the possible utility outcomes (for example, $U_{\max} - U_{\min}$).
- Risk (Reward) Weighting. Multiplying the probability of a node occurring with the utility weight it is characterized by yields its risk. In this weighting scheme, the higher the risk, the higher the weight.

6.4 PokeMinn 2007 WPD analysis

We now show how *WPD* can be used to compare performance between two candidate agent models. We focus on the models' ability to predict the behavior of the top performer in the 2007 Computer Poker "Limit Equilibrium" Competition: Hyperborean07LimitEq1.

6.4.1 Protocol for the analysis

The modified frequency weighting function $W(n)$ we use in this analysis to compute *WPD* weights each node by how *relevant* it is in the poker game tree. We calculate this relevance recursively: the root-node of a tree has a relevance of 1. Each child of the root-node has a relevance of $\frac{1}{|C_r|}$ where $|C_r|$ is the number of the root-node's children. Each child of the root-node's children C_c is has a relevance of $\frac{1}{|C_r||C_c|}$ and so forth.

For this analysis we compare the performance of two black-box models: a fixed strategy opponent model and a learning opponent model. A fixed model is usually based on either general poker knowledge or an offline-trained program that reviews previously documented matches and builds a model from them. The fixed model doesn't change its function over time during an encounter with an adversary. A learning model is different from a fixed model in that it can change its strategy over time as it observes the behavior of its current adversary during an encounter. Each of these models were part of a larger modeling system in our entry for the 2007 Computer Poker Competition: PokeMinnLimit1.

The fixed strategy model was based on general poker knowledge for playing limit poker. It uses static parameters for a pot-value function that predicts the likelihood of the opponent folding, calling or raising. In competition, our agent uses the fixed model to provide predictions of opponent behavior before there are sufficient observations to use the learning model.

In contrast, the learning model has no pre-defined parameters. Instead, it attempts to make predictions of the opponent's behavior based solely on past observations in the current match. At each point in the game tree where the opponent makes a decision the learning model records the frequencies of the observed opponent actions. Then it computes the probability that the opponent will fold, call or raise in the future based on the accumulated frequencies of each action, conditional on the game tree node where the opponent made the decision. This particular model makes several assumptions in order to reduce the number of nodes in the game tree which it must compute: the model does not consider what cards the opponent may have; each betting round is considered independent - thus the model is actually carrying four separate game trees of information - one for each round. There is no windowing or time-based decay in this model - all data is gathered and weighted equally for

the entire course of the encounter. This particular style of model is biased towards learning a stationary policy opponent well, at the possible expense of being mislead by a non-stationary adversary.

6.4.2 Assessing Learning Rate in the Absence of Truth Data

Our first goal is to assess the learning rate of the learning model. If truth data about the opponent's behavior is available it would be best to use it for realistically characterizing learning rate. Unfortunately, when truth data is unavailable, as is the case in many real world situations, we must rely on only our observations to characterize truth. In our first experiment we demonstrate how to evaluate the learning rate of a model from only the observations. To obtain the data for training the learning model

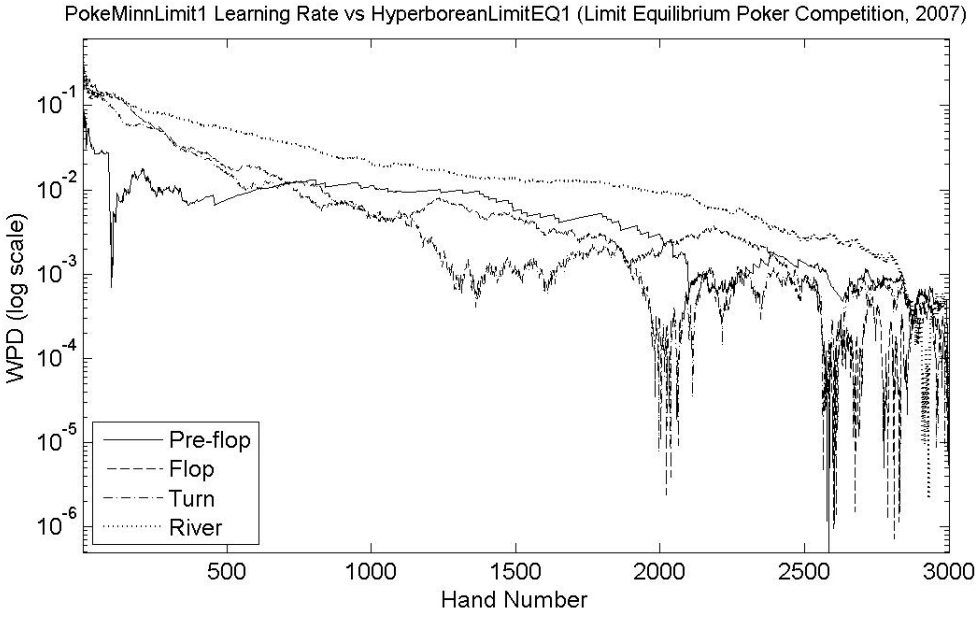


Figure 6.1: Characterizing Learning Rate with Weighted Prediction Divergence. In this figure we show the aggregate *WPD* values for four betting rounds gathered from 20 limit poker matches between PokeMinnLimit1 and HyperboreanLimitEQ1 (3000 hands per match).

we first downloaded poker game log files from the 2007 Computer Poker Competition [83]. Then, using the results from the 20 matches (3000 hands each) played between PokeMinnLimit1 and Hyperborean07LimitEq1, we trained our learning model by allowing it to observe each match being played. We forced the model to keep track of its overall state after each hand was learned. We declared that the resulting model (after

60,000 training hands) represents an approximation of the true behavioral distribution of our opponent. Next we separated the full poker game tree into four rounds (pre-flop, flop, turn, river) to build four smaller game trees. These four trees were selected because they mirror what the original learning model is attempting to learn - the opponent's behavior conditioned on which node they are in during a particular betting round. For each opponent action node in a tree we queried the model for each time step (hand 1 through hand 3000) to obtain prediction distributions of the opponent's behavior. Using Equation 6.3 we generated and plotted *WPD* for each of the four betting rounds in Figure 6.1.

We can use an analysis such as this to indicate which portions of the learning model need further improvement. Notice how in the early portion of a match the pre-flop model learns quickly and then backtracks before it plateaus and then begins to improve again. The flop and turn model appear to improve slowly and steadily until the late portion of the game. Improvement in the river model is very sluggish - it takes nearly twice as long as the other three models to achieve the same level of relative performance. Given the river model's struggling learning performance in the early stages of the match, this analysis reveals the agent could be improved. One possibility would be to encourage the agent to explore the river game tree more fully during the earlier portions of the match (perhaps by making more calls and making fewer folds and raises in the pre-flop, flop, and turn) such that the agent (and his opponent) see the river more often in the early portion of the match. This alteration would make the agent susceptible to exploitation, and the exploration strategy would have to be subtle in order to prevent the opponent from discovering and exploiting the activity.

6.4.3 Comparing Model Quality

We compared the performance of the fixed model to the learning model. To compare the prediction quality of these models we needed data about the true behavior of our desired opponent. For the test-data set, we retrieved all 660 matches (1.98M hands) from the 2007 Computer Poker Competition [83] played by Hyperborean07LimitEq1. We felt this data was representative of the true behavior of Hyperborean07LimitEq1 because it was the largest set of data available to the public and the data documents Hyperborean07LimitEq1's behavior against many different opponents. We scanned all of the test data to determine the frequency counts for each action at each game

tree node in each round. We declared the resulting probability distributions for each node in the game tree as behavioral truth for Hyperborean07LimitEq1. To deter-

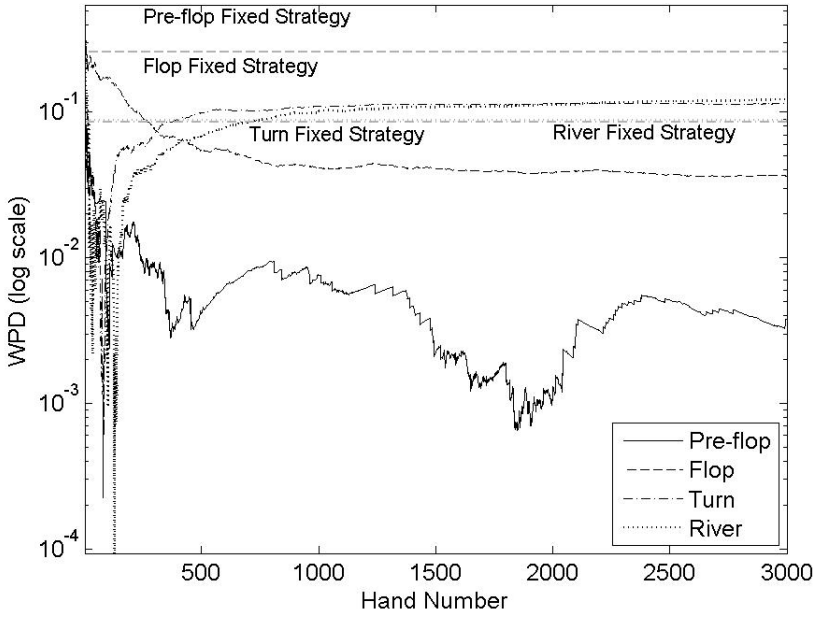


Figure 6.2: Comparing learning versus fixed strategies with WPD for models in each of the four betting rounds in a 3000-hand limit poker game. The horizontal lines represent the fixed-strategy model *WPD*. The darker fluctuating lines represent the learning model's *WPD* values.

mine the learning model's ability to learn the opponent's distribution of behavior, we isolated a training set of 20 matches played between PokeMinnLimit1 and Hyperborean07LimitEq1. We trained the learning model using only observations that it could have made during these 20 matches. We then gathered predictions from the trained model on the hands from the test set and calculated the *WPD* between the learning model's predictions of Hyperborean07LimitEq1's behavior and Hyperborean07LimitEq1's actual behavior. A similar procedure (without a training phase) was used to obtain the *WPD* for the fixed-strategy model. The aggregate results for 3000 hands for both models are shown in Figure 6.2.

While the learning model quickly becomes better at predicting behavior than the fixed-strategy model during the pre-flop and flop, the learning model's prediction quality in the turn and river grow worse over the course of the game to the point where the fixed-strategy model would have better predictive performance after approximately 380 iterations on the turn (and after approximately 740 iterations on the

river).

Though the learning model appears to be very useful for predicting the true opponent's behavior on the pre-flop and possibly the flop (at least when compared to the fixed model we used), its utility is questionable during the turn and river, where the fixed model's predictions might be preferred. Although finding the cause of the reduced accuracy of the learning model during the turn and the river is beyond the scope of this work, it is important to note that model-level measures reveal the flaws in the learning model even when agent-level performance based methods fail to do so. For example, in a separate experiment with 100 matches (300,000 hands) we examined relative performance between two agents in terms of number of small bet increments won per hand (sb/h). We noticed that the learning strategy model achieves an average 0.04 sb/h greater win rate than the fixed strategy model from hands 500 through 3000 when each played against Hyperborean07LimitEq1. If we had relied on just the earnings performance measure as the method of determining the better model (as much of the other research in the field does), we might not have realized that subcomponents of the learning model were not performing as well as the same subcomponents in the fixed model.

Chapter 7

The Environment-Value of an Opponent Model

In this chapter, we show how the potential value of an opponent model within a certain environment can be calculated. We define the *environment-value* of a model in terms of the minimum guaranteed expected payoff obtainable if the model was perfect. We introduce the concept of the *opponent-model oracle* to represent a perfect (but most likely unattainable) computational model of some aspect of the opponent. In a two-agent environment, we show how the minimum guaranteed expected value can be computed using an environment transformation and computational tools from game theory. When game-theoretic methods are inappropriate, we describe how approximations to expected value can be computed by using opponent types drawn from the expected distribution of opponents. Before we begin the formal discussions on oracles, environment transformation and game theory, we motivate the concept that expected value and especially guarantees on the minimum expected value can be helpful for decision making, and that the guaranteed minimum is a function of the environment.

Consider for a moment an abstract one person game in which there are random elements of chance, but there are no other players that affect the outcome, and the environment (the set of rules for the game) does not change over the course of the game. In such a game we could compute the expected value of playing the game fairly easily. To do this we could either traverse every possible path through the game tree

to an outcome and compute a probability-of-occurrence weighted sum of the totals, or we could estimate the expected value using a sampling of the outcomes or some other heuristic method.

Now consider an abstract meta-game where an agent must choose to play one of several simple games, each with its own calculated (or estimated) expected value. Along with other information regarding the agent's risk tolerance and the time each game might take to play, the expected values of each game could help the agent decide which (if any) game to play in the meta-game.

Suppose the agent had the opportunity to obtain additional information regarding the chance elements in a simple game. For example, consider a one-person coin flipping game where the agent receives a point if it correctly guesses the coin flip, but will lose a point if it guesses wrong. In this game, the agent has been told that the coin being used may be biased, but he does not know its probability distribution over outcomes. Without knowing the probability distribution he cannot calculate the expected value of the game. It would be very useful for the agent to know the probability distribution of the coin in order to determine if he should bet (and on which side of the coin he should bet).

		Red	
		A	B
Blue	H	$0.9, 0.1$	$0.1, 0.9$
	T	$0.1, 0.9$	$0.9, 0.1$

Figure 7.1: The two-player game Γ.

Now consider the following two-player repeated game Γ with players red and blue shown in Figure 7.1. There are two biased coins in this game: Coin A ($P(heads) = 0.9$); and Coin B ($P(heads) = 0.1$). Both players are aware of the labels and biases of the coins. In this game, red first chooses the coin that will be flipped (A or B), but does not immediately reveal his choice of coin A or B to blue. Blue must then guess whether the coin will land heads or tails when it is flipped. After both players have made their choices, the coin is flipped by an unbiased referee. Then blue will be told which coin red chose and red will be told which side blue guessed. If the guess (heads or tails) is correct, blue will earn a point, and red will lose a point, otherwise red will earn a point and blue will lose a point.

Without any information about the opponent, blue agent has no incentive to prefer

guessing heads over tails in this game. Furthermore, he knows that if he shows any bias between heads and tails, red may notice and start choosing the coin that makes blue suffer in future games. Therefore blue selects heads or tails randomly with equal probability. Knowing that blue will choose randomly, red has no preference over which coin to select. Furthermore, red realizes that if he shows any bias, blue might discover the bias and may use the bias to hurt red. Therefore, red chooses a coin at random from set, with equal probability. In Γ, the expected score for each player is zero.

Clearly, if one of the players could know what the other player's choice was, they could reap large benefits in this game. For example, if blue knew exactly which coin red would choose and red did not know what blue would do, blue could earn an expected score of 0.9 per iteration. Likewise, if red knew which choice blue was going to make, but blue did not know red's choice, red could earn 0.9 per iteration. In Γ, the value of having a perfect prediction about the other player's next move is 0.9.

Consider playing from blue's side of the game. One can think of obtaining information regarding red's coin choice as if it had been obtained from an oracle. The expected score 0.9 is a *guaranteed minimum bound on the increase* in the expected value by using the oracle. It is a guarantee because no action by red could reduce this value increase. We define the *environment-value* of an oracle in this environment as the expected value the player could earn by using the perfect prediction in the environment minus the expected value earned if the perfect prediction was unavailable. In this example, the value of the oracle is $0.9 - 0 = 0.9$.

		Red		
		A	B	C
Blue	H	$0.9, 0.1$	$0.1, 0.9$	$0.5, 0.5$
	T	$0.1, 0.9$	$0.9, 0.1$	$0.5, 0.5$

Figure 7.2: The two-player game Γ'.

Next we show how a change to the environment can, but does not necessarily alter the value of an oracle in that environment. Consider a slightly different game Γ', shown in Figure 7.2 where both players are aware that red has been given a third coin from which to choose: Coin C ($P(head) = 0.5$). Without extra information about the other player, each player's expected value for the game is still zero. But consider what happens when blue knows exactly what red will do. In order to understand the

interaction, we must first determine how a rational opponent would behave. If red always chooses coin C then he will be safe, even in the event that blue is somehow able to predict red's choice of coin. Therefore, red, without extra information chooses coin C every time, and the expected score for each player is still zero, regardless of blue's actions. Assuming red cannot predict what blue will do, red obtains a the desired expected value by always choosing coin C. Since blue's actions will have no effect on the red's policy, any additional information regarding the red's coin choice is irrelevant (because red will always choose coin C). Thus blue's environment-value of oracle which predicts red's choice perfectly is $0 - 0 = 0$.

To show that the environment value is a minimum guaranteed expected value for the agent we simply examine what would happen if red altered his policy when blue has access to the oracle. If red chose coin A or B with any probability greater than zero, blue's oracle would immediately alert blue of this and blue could choose heads or tails appropriately, increasing his expected score to 0.9, each time red chose coin A or B, just as blue did in game Γ. Thus, 0.0 is the minimum value improvement that blue could earn by using this oracle.

Now we consider the situation where only red has an oracle. With the addition of coin C, the situation for red is quite different. If red has perfect information about whether blue will choose heads or tails next, red can select coin A or B smartly to maximize his own score, yielding an average of 0.9 for himself. No action blue could take could change this minimum guarantee because the oracle always tells red whether blue is about to choose heads or tails. Thus, red's environment-value of the oracle that predicts what blue will do is $0.9 - 0 = 0.9$.

From these simple examples, we've informally shown several important points:

- Given a game against nature, we can calculate the expected value of playing the game when using a certain policy.
- Given multiple options for policies, we can calculate the expected value for each policy in a game and choose the best policy.
- In a two-player game, we can calculate the expected value for our policy choice, given the other player's policy.

- In a two-player game, having an oracle of another agent is equivalent to knowing perfectly some specific information about that other agent, i.e. its next action.
- The environment-value of an oracle for a given player is equivalent to the minimum expected value they would earn when the policies have changed minus the minimum value they could have earned prior to introduction of the oracle.
- If neither player's policy would change after the introduction of a particular oracle, then the environment-value of that oracle is zero.

In the simple examples above there was only one prediction that could be made about a player: what would they choose for their next action. In more complicated environments there are many different types of predictions that could be made and many different types of conditional information that are available. Different predictions and different conditional information supporting that prediction can yield different oracles with different environment-values because of the relationship between the conditional information, the oracle, the prediction, and the change in policy which occurs when the oracle is used.

If an oracle can be thought of as a perfect opponent model then the environment-value of the best opponent model is the same as the environment-value of the oracle which makes perfect predictions under the same conditions. Knowing the values of the different oracles can inform the decision about which types of models might be useful.

In the remainder of this chapter we show how we can compute a static minimum bound on the expected value of an oracle (perfect opponent model) in a specific environment[1]. We also show that the the environment-value (the amount of score improvement earned in the game, for example) that could be obtained by using an opponent model is a function of the environment itself. We do this using a three step process:

1. Form an oracle which provides the information a perfect model would yield.

[1]The oracle only makes predictions about one player, which we refer to as the *exposed* player. Both players know of the oracle's existence and purpose and which player is being exposed. Both players can see what information the oracle is revealing. Thus, the exposed player knows what information the oracle is revealing about him.

2. Transform the environment into a new environment where the oracle's information is part of the environment.

3. Solve for the environment-value of the model using game-theoretic techniques.

Each of the steps will be explained in detail in the following sections. If we perform this task for each type of opponent model of interest, we can discover the bounds and use the information to select promising models to develop for any particular environment.

7.1 Abstract Prediction Models and Oracles

We begin with several definitions for common terms used throughout this chapter.

Abstract Prediction Model: A *specification* for a mapping between specified inputs and specified predictions of a modeled agent.

The abstract prediction model can be thought of as specifications of which inputs are required by and which predictions are generated by a model. In the language of software development, the abstract prediction model is the signature of the function which accomplishes the prediction. For instance, in poker we might identify two different abstract prediction models: one which takes as input the size of the pot and predicts whether the opponent will fold or not in his next action, and another which takes the sequence of actions made so far by both players and predicts whether the opponent will fold, call, or raise in his next action.

Oracle: A function that maps an abstract prediction model's inputs to *correct* predictions about the modeled agent.

The oracle is a hypothetical perfect model used solely for the purposes of analyzing agent interaction in a specific environment. Thus it has certain inputs and outputs but how it works is irrelevant. It is perfect in the sense that it will always predict some aspect of the modeled agent correctly, given the required inputs.

In order to find performance bounds attainable by a given abstract prediction model in an environment we transform the environment using a technique that simulates

having an oracle of that type and then we determine a minimum bound on how well we could have performed in the transformed environment.

For this work, we examine three types of abstract prediction models in a simple game: the *state* model, *policy* model, and *action* model. One can think of the various opponent models as devices that predict the state the opponent believes he is in, the policy he will use to make decisions, or the action he is about to take. A state oracle is a device which knows what state the opponent thinks he is in, given the inputs. The policy oracle is equivalent to the opponent's thinking process, or the software he is using to make decisions: it is a perfect map from state to action. The action oracle perfectly predicts what action (or distribution of actions) the opponent will take next, given the inputs. In a poker example, the state oracle would give access to the exact cards the opponent has as if the opponent was playing with his cards face-up. The policy oracle would indicate what action (or probability distribution over possible actions) the opponent would take for each possible set of cards they held. The action oracle would indicate exactly what action the opponent was about to take.

Before we describe the analytical process for finding opponent model performance bounds using oracles, we first provide additional intuition for the relationship between the performance bounds obtainable using an oracle and the structure of the environment. Once this is made clear, we describe a technique for discovering bounds that is equivalent to playing the game with access to a perfect opponent model.

In some games, such as rock-paper-scissors, the opponent's actions, in combination with our actions *completely* determine the winner of each game. Since there are no stochastic processes in the game itself, an oracle that predicts the action the opponent is about to take just before he does it can allow us to select a perfect best response, allowing us to win every iteration and obtain the maximal value we could in the game. Conversely, in an absolute gambling game where each player decides whether to bet or not on the future flip of a fair coin (if we both bet, we flip the coin: heads, you pay me one dollar, tails I pay you one dollar), neither the state, action, or policy opponent model could assist us in extracting any additional expected value from our opponent. The result of the absolute gambling game is entirely determined by nature, and any effort in building a model to predict the opponent's state, action or policy is a waste of resources. In both of these games, the environment-value of the oracle is related

to the structure of the game rather than the type of players playing.

As mentioned in Chapter 1, most interesting adversarial environments often contain elements of chance, skill, and asymmetric information[2] available to the players. Thus, most environments fall somewhere between the two games described above.

Let us next examine a game that falls in between these two extremes: high card draw with simultaneous betting[3]. In this game chance plays a large role, and an oracle that predicts whether the opponent is going to choose to bet or pass doesn't give us all the information we need to always make choices without regret[4]. Even an oracle that knows just the opponent's decision policy doesn't give us everything we want: to decide without regret we would really need to know what card he had in addition to whether or not he was going to bet.

Yet knowing something about the opponent's next action or his policy in high card draw does help us somewhat: we should be able to make better decisions once armed with this information. For example, if we observed the opponent over a large history of hands and noticed that we never saw an opponent's card at showdown that was worse than a Nine, we could access the action oracle to see if the opponent was about to bet in the current hand. If the oracle told us he was going to bet, we could make a much better decision on whether we ourselves should bet, because we know he probably has a Nine, Ten, Jack, Queen, King, or Ace. If instead we had a policy oracle for our opponent that told us he always bets on Nine or better, we could make a more informed decision on which cards we should bet on than if we did not know his policy exactly.

[2]Asymmetric information is information available to a subset of the players, but hidden from the other players. An example of the asymmetric information in poker is the collection of private cards each player holds.

[3]In high card draw with simultaneous betting, each player places the ante value in the pot. Then each player draws a card from a standard deck. Players decide to bet or pass using one of two face-down markers to indicate their choice. The markers are turned face up to reveal the actions of each player. If both players bet, then the cards are revealed and the higher card wins both bets. If there is a tie, then no bets are exchanged. If one or more players pass, both cards are returned to the deck and the deck is reshuffled in preparation for the next round.

[4]We use a very specific meaning of "regret" here. Intuitively, we would regret an action we took if a different action under the same circumstances would have yielded a better result for us. A decision without regret in high card draw with simultaneous betting is one in which given the cards we and our opponent were holding, and knowing what action the opponent was actually going to take, we would not want to change our decision about what action we took.

We assume that in finding the bounds of performance of an opponent model, we must consider an oracle will give us only one of the following pieces of information about the opponent: The possibly hidden state in which the opponent is in, the policy which the opponent will use to make decisions, or the next action which the opponent is about to make. The reason that we only consider models of one of these pieces of information is that if we had two then the third is calculable, and we could perform optimally without needing it. For example, if we had the opponent's policy and his state, we could calculate his action. If we had his policy and action, we could infer his state. If we had both his state and his action, we have everything we need, and his policy is irrelevant (but could be approximated as a set of point-by-point if then rules for all visited states and actions).

With this notion of abstract opponent models and oracles, the next section presents the second step in the process: environment transformation.

7.2 Oracle Simulation through Environment Transformation

To determine the value of an oracle that reveals our hypothetical opponent's *state*, *action*, or *policy* in an environment Γ we derive a new environment Γ' where we have access to that information as part of the environment[5]. In other words, the predictions made by the oracle are revealed to our agent in the transformed environment, and the actions our agent takes will be able to use the information. We denote Γ'_{state} as the transformed environment that is equivalent to revealing the state the opponent thinks he is in (such as the card he is holding in high card draw), Γ'_{action} that is equivalent to revealing the action the opponent is about to take, and Γ'_{policy} as the transformed environment that is equivalent to revealing the opponent's policy. We can then calculate how much we could earn in the transformed environment in relation to the original game.

Since we are planning to do actual calculations in the game Γ', we will need to access

[5]One question that arises when using the opponent information in this way is whether we should consider the information about our opponent public or private, in a game-theoretic sense. In other words, does our opponent know that we know the information, or is he ignorant of that fact? Whether the opponent knows does make a difference in his policy and thus affects how the game tree is traversed during search. This is a factor in the optimal path calculation and the value of the optimal action sequence. For the analysis in this chapter, we focus on oracles which take otherwise private information about one player and make it public information. We leave the analysis of oracles that secretly reveal one player's private information to the other player for future research.

its game tree: τ', and we will need to instantiate some kind of device that represents the decisions the hypothetical opponent would make. The selection of the device for solving the environment will depend on whether we are solving the game using game theory or Bayesian techniques. We then replace the opponent's decision nodes with the actual decisions (or probability distributions over decisions) the device would make. This leaves us with a compression of the original game tree such that the only remaining nodes are our decision nodes (and possibly chance nodes from nature). Essentially the transformed environment becomes a one-player game where our goal is simply to compute the optimal solution and determine the expected value of that solution.

7.3 Finding the Value Bounds with Game Theory

In this section, we describe how to find the environment-value of the opponent model in the original environment by finding the agent's expected value in the transformed environment and subtracting the agent's expected value in the original environment. At the core of this step is the method to be used for finding the expected value - in essence we need to evaluate the solution from the agent's perspective in both environments. In the remainder of the section, we motivate the use of game-theoretic tools for finding this solution, and discuss the implications of using those tools.

7.3.1 Game-Theoretic Environment-Value Calculation Technique

Let us consider the nature of the opponent for a moment. If our opponent was rational (he makes decisions to maximize his own utility given his beliefs [6]), then he would make a choice (or probability distribution over choices) which was in his own best interest at every point where he had to make a decision, given his beliefs about the way the decisions would affect his utility. Furthermore, if an agent and his opponent in a two-player environment are both rational and know the rules of the game and the available payoffs for all outcomes, then game theory describes the intersection of their joint choices (their strategies) as a Nash Equilibrium [6, 55, 56]. In an equilibrium, neither player can improve his utility by altering his strategy (probability distribution over actions) unilaterally, thus the payoff for a player at a Nash Equilibrium constitutes a lower bound on the score a player could achieve when they play their strategic element of the equilibrium. They could earn more, but they

could not earn less as long as they do not deviate from their equilibrium strategy.

This is a strong motivation for using solution techniques that yield a Nash Equilibrium to find strategies which could be evaluated to determine the lower bound of utility in an environment. To use game-theoretic tools to solve an environment, we must make the assumption that the Nash Equilibrium of the game can be found, and that all Nash Equilibrium in the game have equal value. The first assumption depends on the tractability of the technique for finding the solution and the computational hardware available. Thus some games such as chess and heads-up limit Texas Hold'em would not be suitable targets for this form of analysis. The second assumption is true when the game is a two-player game where the sum of the player's payoffs at every outcome in the game is equal. Many games that people play for entertainment fall into the this category, but it would be inappropriate to make this assumption in many real world environments.

One of the simplest and most straightforward implementations of equilibrium calculation is the minimax algorithm. If we assume that our opponent is a game-theoretic equilibrium player then we could assign some form of the minimax[6] algorithm to represent the policy of our hypothetical opponent. Thus we could find an optimal set of actions in the new environment by providing a function that maximizes the utility of our decisions in the new environment. By assessing the expected values of the new environment when playing our optimal strategy against the minimax player opponent, we can obtain the minimum bound for expected value in the new environment. Performing the same calculation in the old environment, we could obtain the minimum expected value in the old environment. Subtracting the original value from the new value would yield the environment-value of the model:

$$V_{\Gamma}(M) = U(M|\Gamma') - U(M|\Gamma)$$

In the next sections we examine two case studies to demonstrate how to find the value of a model in an environment: simultaneous-bet high card draw and the simultaneous move strategy game.

[6]Recall that minimax can be used to calculate equilibrium play in games with game trees that have no chance nodes. While traditional minimax is useful for non-stochastic games, there are derivatives of minimax designed for use in games that incorporate chance nodes.

7.3.2 Case Study: Environment-Value in High Card Draw

As a proof of concept of determining opponent model performance bounds by using the game transformation technique described in Section 7.3, we studied taking oracle-based transformations of the repeated high card draw with simultaneous betting game. The transformations we examined yielded three new games that independently incorporated the state oracle, policy oracle, and action oracle.

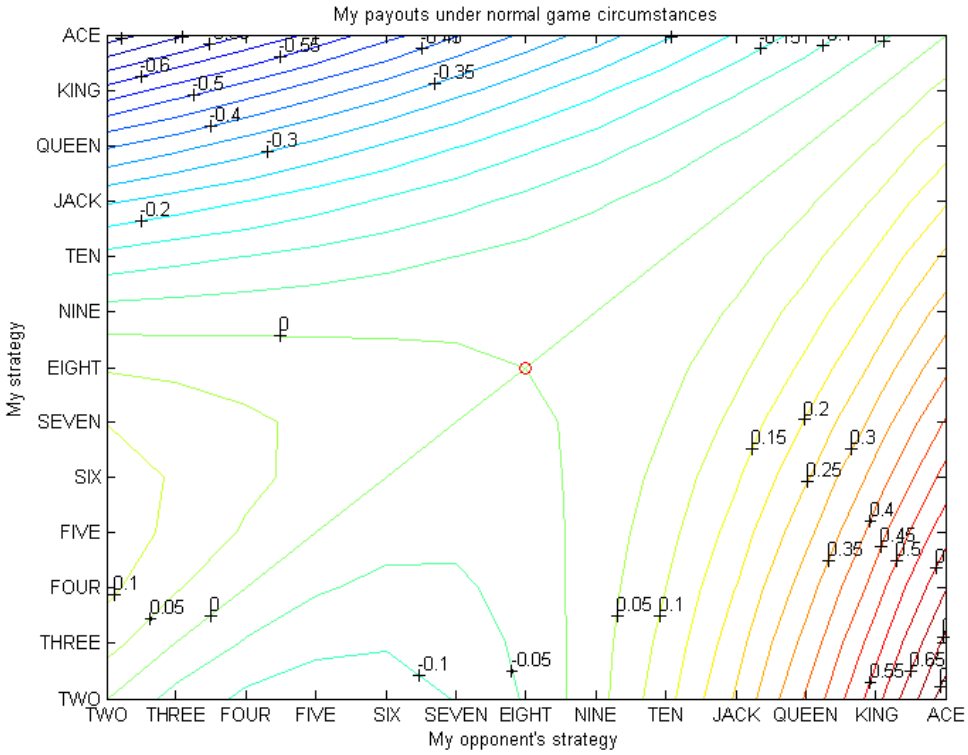


Figure 7.3: Expected payout in a high card draw game with simultaneous betting where the ante is $1 and the bet value is fixed at $1. Each axis is labeled with a player's pure strategy: it indicates what their threshold card is. The threshold card is the card which separates the cards they would like to fold from the cards they would like to bet with. If, for example a player's threshold card was SIX, then they would bet with SIX or better while folding all other cards. The opponent's strategy is shown on the X axis and our strategy is shown on the Y axis. The intersection of two strategies reveals the expected payout over an infinite number of games if each player's cards are drawn from a standard deck without replacement, then returned and reshuffled after each game. Notice the pure strategy equilibrium is indicated at EIGHT-EIGHT, implying that a player following an equilibrium strategy would fold if they obtain a card lower than EIGHT and bet otherwise.

For this set of experiments, we made several assumptions. In the original game there is a compulsory ante of \$1 which both players must pay to play each game. Each player may decide to bet a fixed \$1 amount or fold - these decisions are made simultaneously to eliminate the incentive for deceptive play. We furthermore make the assumption that our opponent's strategy is unknown, but fixed, and characterized by the selection of a single threshold card: when the opponent is dealt a card is greater than or equal to this threshold, they will bet; otherwise they will fold[7]. We also assume that our goal is to determine a threshold card-based strategy, which may be distinct during each round of play. Figure 7.3 shows our resulting payout at each pair of intersecting strategies, and the pure strategy equilibrium that is obtained in this game: neither player wishes to have a strategy other than EIGHT. As expected in a zero sum game, our payout at the equilibrium point is \$0.

When we examine the game derived by allowing us to always know our opponent's card, with him knowing that we know his card, we know that we should always bet when his card is lower than ours. Furthermore, since we still do not know his policy, we must entertain the idea that there may be occasions when we would like to bet, even when our opponent's card is clearly higher than ours because there is a chance he could still fold. We computed the expected payout for the game as a function of our joint strategy and show the results in Figure 7.4. As we might expect from a casual analysis, this transformed game's pure strategy equilibrium is ACE, EIGHT (when our opponent's card is equal or higher, our strategy is to bet if we have an ace and our opponent's strategy is still to bet with an eight or better; when the opponent's card is lower, we always bet). At this new equilibrium, we can expect to earn an average of approximately \$0.24 each time we play the game. We make the claim that knowing the state of the opponent perfectly is worth at least \$0.24 per game: if we had an intuition that the opponent was going to play some strategy other than EIGHT, we could possibly earn even more money, but we are guaranteed an expected value of \$0.24 per game with a fixed, secure strategy. Thus, the value of knowing the opponent's threshold card is lower bounded at \$0.24 per game. Another interpretation of this finding is that if we were somehow able to build a device (such as an opponent model) that would reveal the opponent's cards to us, we would not desire to pay more than $(\$0.24) \times n$ for it, where n is the number of games we are

[7]Since there is no incentive to bluff in this game, the notion that a player's policy can be described by a single bet/fold threshold is valid: Assuming the threshold is equal to the lowest card we would be willing to bet, folding cards above that threshold would reduce the expected value of the policy.

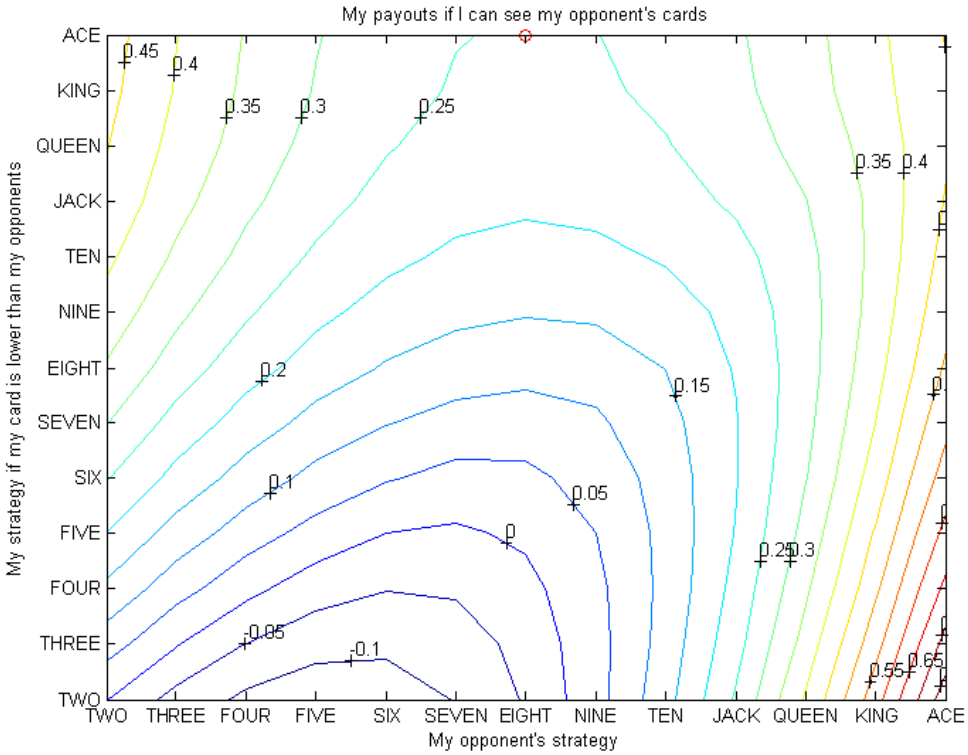


Figure 7.4: Expected payout from a game derived from high card draw with simultaneous betting where our opponent's cards are fully revealed to us (this is public information: our opponent knows we can see his cards) but he is not allowed to see our card. This game incorporates a perfect model of the opponent's state. At the intersection of each player's strategies, we find our expected payout in the game.

going to play. In this game, building a device that obtains the state the opponent is in may seem unfeasible without cheating, but in other more complicated games, it may be easier to obtain this information. Poker, for example, is a game in which the opponent betting actions over time may reveal a great deal about the strength of cards he is holding.

We next examine transforming the original high card draw game into one that reveals the opponent's policy to us. We claim that in this game, the policy is probably the easiest model to develop, especially if the opponent is considered to be a fixed strategy player. In this simple game we notice that the new game derived from the original game with the opponent's policy revealed is actually a much simpler game, where our goal is to find a best response and our related payout for each possible policy choice

the opponent could make in the original game. Figure 7.5 shows a graph we use for

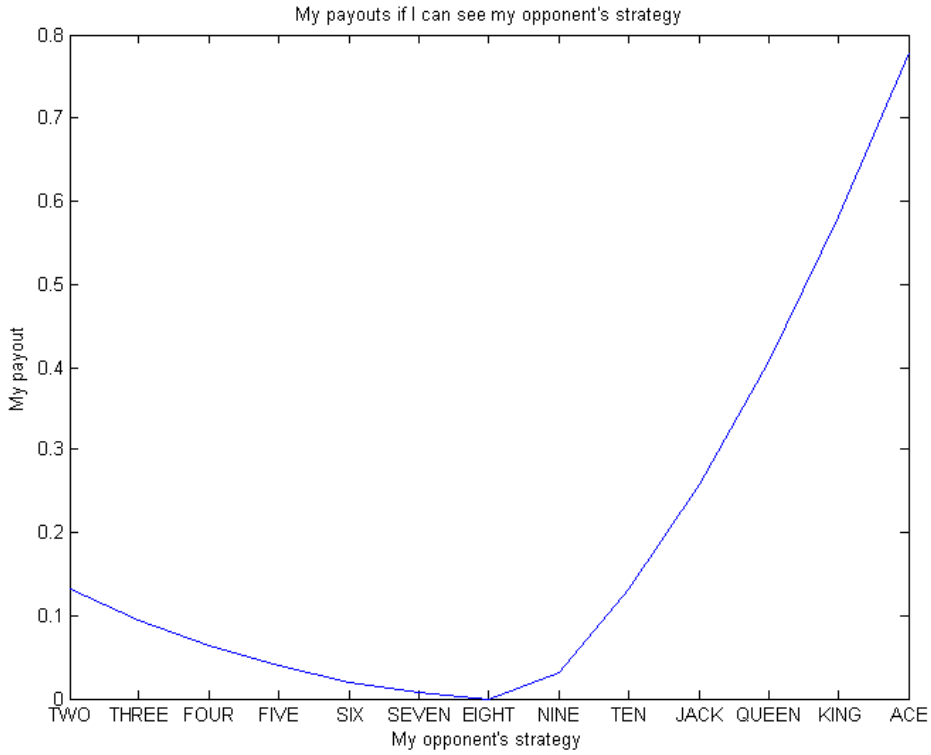


Figure 7.5: This graph depicts our expected value of playing the strategy that is the best response to the opponent's strategy when the opponent's strategy is revealed to us as part the transformed environment.

analyzing our policy interaction in this game. The graph shows our expected earnings from the game if we play a best response to the known opponent policy. As expected, the additional value of knowing the opponent's policy in this game when the opponent is an equilibrium player is zero: we cannot hope to earn any money from our opponent when he plays a strategy of EIGHT. But, if our opponent is playing other than an equilibrium strategy, this analysis reveals the value of knowing his policy and playing a best response to it. If we were to know that our opponent's policy was ACE, for example, we could expect to earn \$0.70 per game by playing a best response. If we could obtain a-priori a distribution of the types of opponents we might face, where a player's type is his strategy, we could use this calculation to determine our expected value of a perfect opponent model against that set of players. Knowing this value would enable us to determine a cap on how much we should be willing to invest in

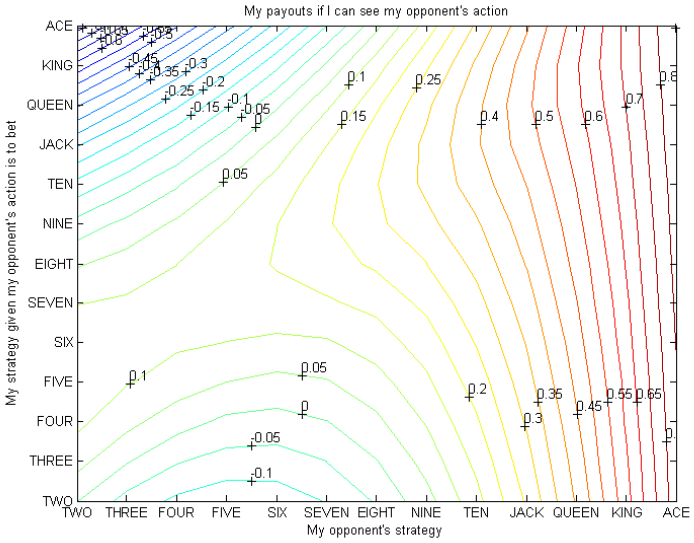
developing an opponent model to play this game against that set of opponents.

We next examine the value of knowing the opponent's action before he takes it with respect to our expected value in the original game. Figure 7.6(a) shows our payout at the intersection of each player's strategy when the opponent's action is revealed to us before we decide on our action. Another way to think of this transformed game is that we've converted the original simultaneous action game into a turn-based game where the opponent is at a disadvantage because he must act first and receives no in-game monetary compensation for doing so[8]. Notice also that there is no pure strategy equilibrium in this game. If we make the assumption that the opponent will select the best pure strategy available to him (the one that earns him the highest expected value) and we take a best response to a fixed pure strategy opponent, we obtain the payout shown in Figure 7.6(b). In this figure we can see that knowing the opponents action before we make our decision of what to do is worth approximately \$0.1161 per game when we make the assumption that the opponent is playing a fixed pure strategy.

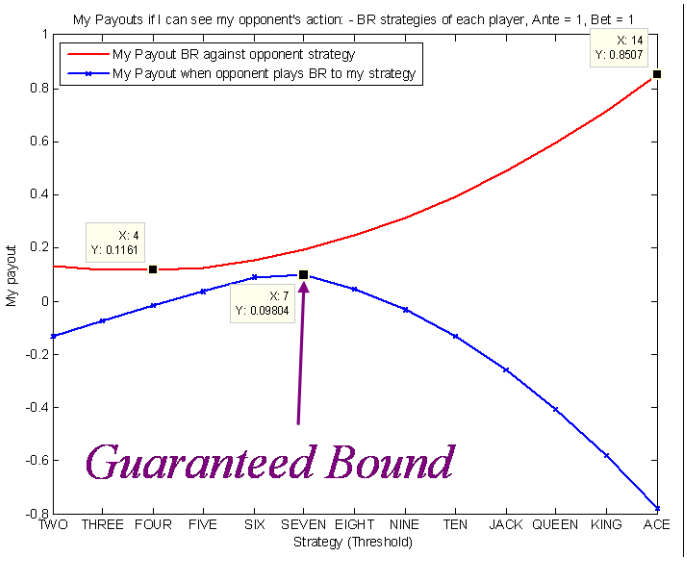
But the lack of a single highest-valued pure strategy leads us to believe that the assumption that the opponent will choose a *pure* strategy is flawed. If the opponent is not forced to use a fixed pure strategy in our analysis, his selection of a threshold may change from one iteration to the next. He may possibly alter his strategy during the game depending on what his current belief about our strategy is: this may induce mixed strategy equilibrium as well as provide value for deceptive play (such as bluffing) where a player's policy can no longer be described by a single threshold parameter. In Figure 7.6(a) we can see the incentive for the opponent to switch policies the instant he picks a certain policy and realizes that we are going to make a best response.

For example, if the opponent was to choose a fixed threshold of EIGHT (the Nash Equilibrium policy in the original game) our best response is to choose a policy of TEN. His best play if he knew we were playing TEN is to choose a threshold of TWO. But if he was going to choose TWO, our best response is to choose FIVE or SIX. When we do this, the opponent switches to SIX, then we follow with our best response: EIGHT, and he then has incentive to change to TWO again, restarting the

[8]In some forms poker, such as of Texas Hold'em, a player who doesn't have to make his decision until after the other players have had to make their decisions first has an information advantage and is assessed a larger ante to partially compensate for his information advantage.



(a) Our payout for fixed pure strategy play



(b) Our payout for best response to fixed pure strategy opponent

Figure 7.6: Exploring the transformed game where our opponent's action is revealed: When we know the opponent's action as a condition of the derived game, we can see our actual payout for each intersection of pure strategies in Figure 7.6(a). Figure 7.6(b) reveals the payout we would receive by playing a best response to the indicated opponent fixed pure strategy.

never-ending cycle.

Clearly, the opponent's incentive to choose a pure strategy in the transformed game

is minimal. In this game, the opponent may instead attempt to use a mixed strategy to keep us uncertain of what his real policy is so that we are not able to perfectly determine his card from his action and the past observations of his threshold (policy). For this game, we can think of a mixed strategy as a probability distribution over pure strategies. At each iteration, the opponent selects a strategy (card threshold) randomly according to the distribution. Thus, his policy is actually parameterized by the distribution over all possible thresholds. Finding the mixed strategy equilibrium (the set of mixed strategies, one for each player) is computationally expensive, but solvable for this simple game. We used the Gambit [52] software to solve for the mixed strategy equilibrium of this game. At the equilibrium, the value of knowing the opponent's next action for us is approximately \$0.1124, slightly lower than our yield in the analysis when we assume our opponent is using a pure strategy. This lower yield is expected and confirms that the opponent has an incentive to employ a mixed strategy when we know his action with certainty.

7.3.3 Case Study: Environment-Value in the Simultaneous-Move Strategy Game

We now apply our value-bounds-finding technique to a much richer environment: the simultaneous-move strategy game described in Chapter 5. In the version of the game we use for this case study, there is no hidden information, thus the state of the game is already known and an oracle which reveals the opponent's state is unnecessary. Since by definition, knowing the policy of a Nash Equilibrium player has no benefit for its opponent, we do not need to assess the value of the policy oracle either. For these experiments, we focus solely on the value of the action oracle. Essentially, we are trying to answer the question: If I could purchase information about what action my opponent is about to take before he takes that action, how much is that information worth?

If we label the player who's actions are being revealed as the Exposed player and the other player as the Protected player, we can answer this question with a the following process:

1. Determine the baseline value of each state when each player is playing the simultaneous-move game and both players policies are the mixed Nash Equilibrium policies for the game.

2. Determine the value of the game for the Protected player when he knows what action the Exposed player is going to take just before the Protected player decides what action to take.

3. Subtract the Protected-player's value of the baseline game from the value they would achieve in the transformed game. The result is the value of the action-revealing oracle.

Since this game is a multiple-turn game, there are several ways to determine the value of opponent-action-revealing information. Perhaps the most intuitive way is to assume that the oracle would be available in this turn and all turns thereafter. Thus, from any state in the game, the oracle would reveal the Exposed player's next action. We also make the assumption for the purposes of this analysis that the Exposed player will play a stationary, optimal strategy. Given a state, there is only one action which the opponent would choose and he would choose that action every time he was in that state, independent of the history of play prior to arrival in that state.

Since the successor state is partially stochastic (often based on the outcome of the stochastic events which occur during the game phases following the selection of each player's actions), we can form a set of outcomes that could occur when the opponent takes an action and we take an action. Once each player selects an action, the outcome states depend on the transition probabilities which are solely a function of the rules of the game. By weighting the likelihood of an outcome with the transition probabilities, we can determine expected value for a state from the value of the successor states. We use the same iterative solution method described in Chapter 5 to find the value of each state, starting with the terminal states and working backwards to solve for the value of the predecessor states. When the values of the states converge we know that we have a solution for the values of the states for a game of indefinite length. The difference between the algorithm used here and the one used in Chapter 5 is that here we assume the Exposed player's choice is derived from a probabilistic minimax over successor state values (instead of a solution to the linear programming problem for the purpose of finding the mixed Nash Equilibrium). The Protected player's response is the other component of the minimax play: it is the best response to the Exposed player's chosen action.

We solved the game in this manner to determine the value of the action-revealing-

oracle from each state, given that the oracle will be available for playing the game from that state forward until a terminal state is reached. If we were to play a full game from the default starting state (each player has one unit and no bases) the expected value of the state (and thus the oracle) is 0.0050. There are 35 fair[9] starting states for this game. In these fair starting states, the average value for the improvement in the score when the action oracle is used (instead of the Nash Equilibrium policy) is 0.0808. If we collect the value differences earned by the protected player in each state and then sort them according to their value, we can generate a distribution of the state values and display the distribution as a histogram. A histogram of the values for all fair starting states is shown in Figure 7.7(a), and a histogram for the values for each of the 2021 non-terminal states is depicted in in Figure 7.7(b). The mean value improvement for all non-terminal states is 0.0692.

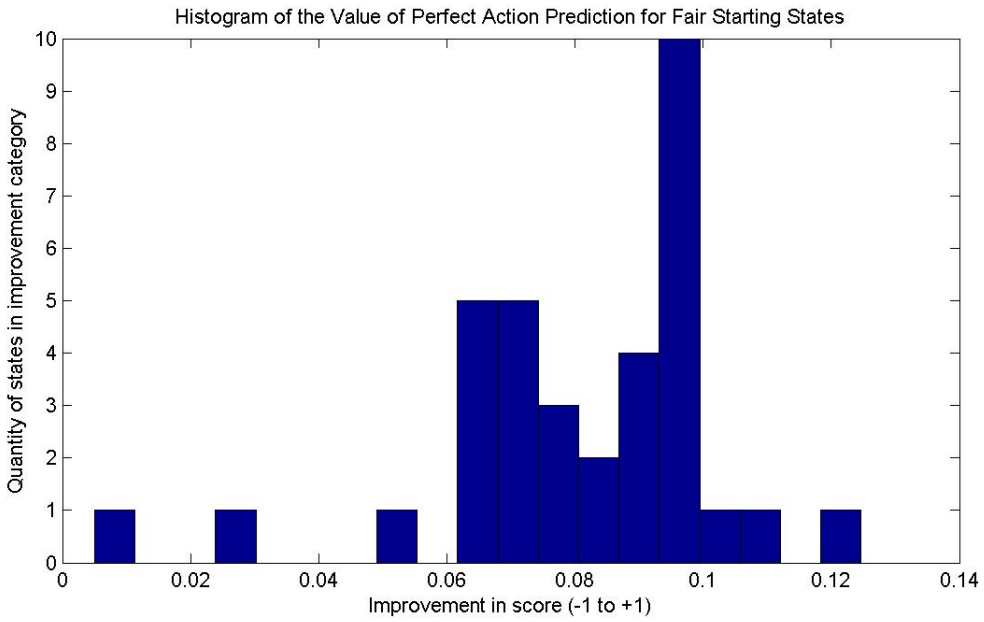
Another analysis we can perform with this data is to determine how much expected value could be earned from the action oracle's information if it was only available for a *single* decision. Essentially we would like to determine the value of a state if we were allowed to use the oracle once in that state and then we were not allowed to use it for any portion of the remainder of the game[10]

We compute this value by first determining the set of successor states and the associated probabilities that occur when the action oracle is available (and the minimax solution method is used). We then lookup the values for just the states that were successor states when using the mixed-equilibrium solution method. By weighting these values according to their transition probabilities and summing, we get the expected value of the game if the remainder of the game was played without the action oracle. If we subtract the expected value of the successor states from the value of the initial state, the result is the difference in the value of the joint action prescribed by the action oracle and the joint action described by the mixed Nash Equilibrium: it is the value of the action oracle's information for one decision from the current state.

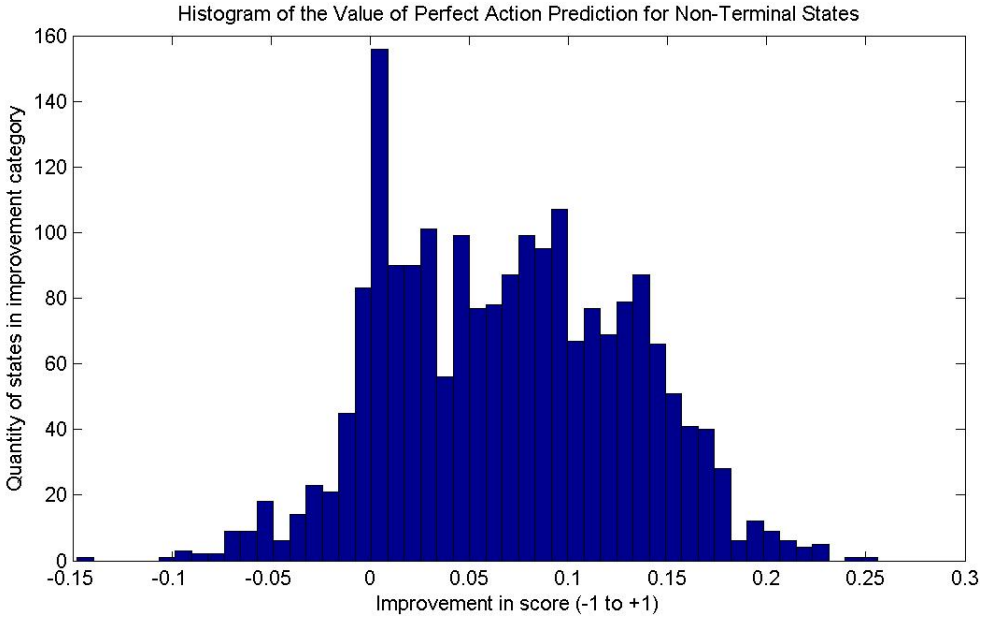
If we use the action oracle only for the first decision from the default starting state,

[9] A *fair* starting state is one in which both players have the same number of assets and the same configurations (bases and units) at each location (it is fair). In other words, if the player identities were switched, a fair state has the property that the ID-reversed state is an isomorphism of the original state. Starting states must also be non-terminal. Thus a state where all players have no assets is fair, but not a valid starting state.

[10] For the remainder of the game we would use the mixed strategy Nash Equilibrium policy described in Chapter 5.

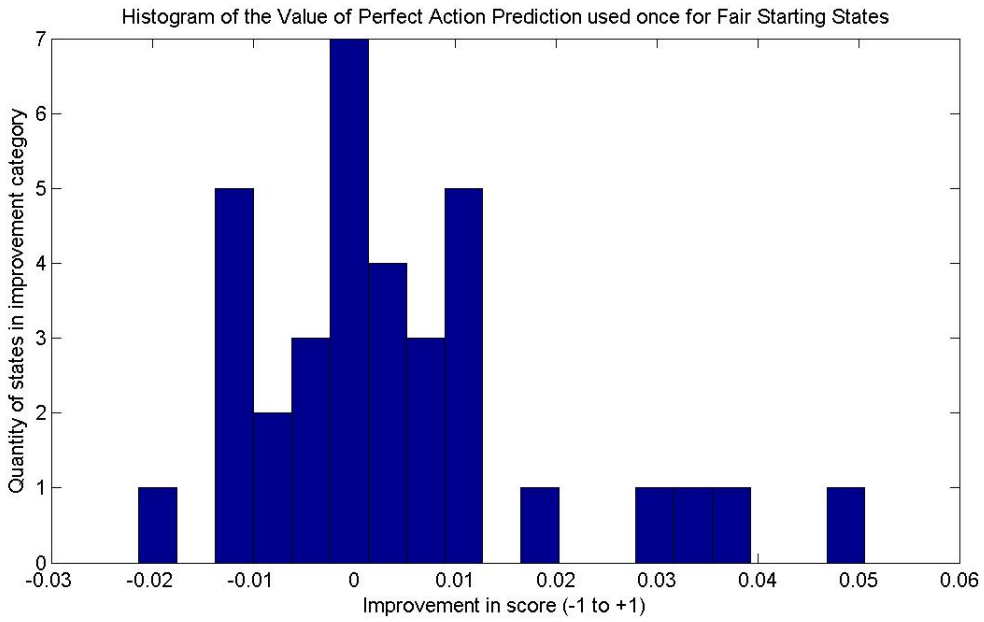


(a) Improvement in score from starting states when the action oracle is available on every turn

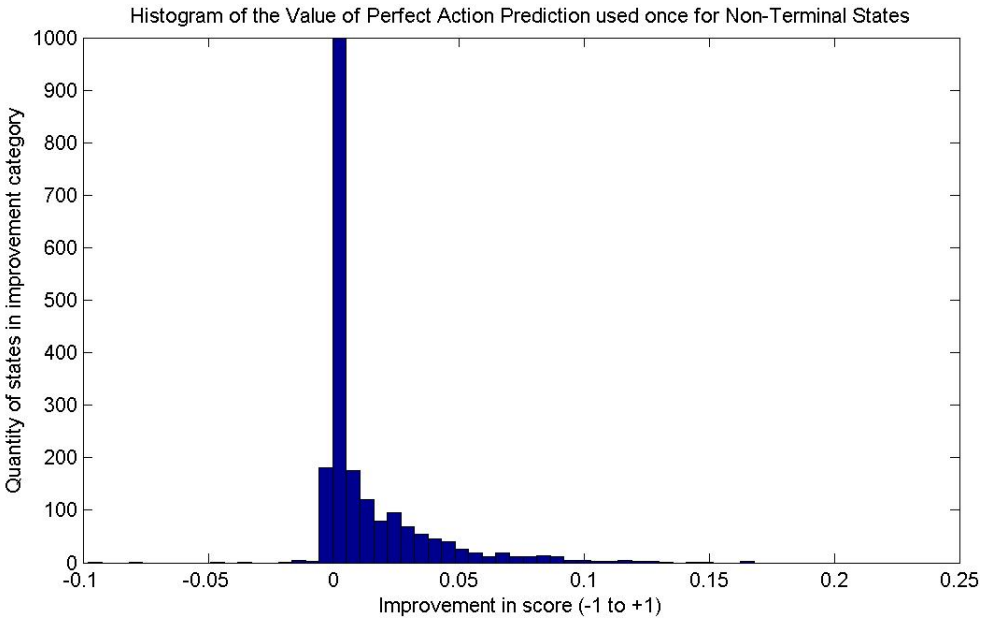


(b) Improvement in score from non-terminal states when the action oracle is available on every turn

Figure 7.7: When using the action oracle for the entire game, a histogram of our improvements in value is shown for fair starting states in Figure 7.7(a) and for all non-terminal states in Figure 7.7(b).



(a) Improvement in score from starting states when the action oracle is available for only the first decision



(b) Improvement in score from all non-terminal states when the action oracle is available for only the first decision

Figure 7.8: When using the action oracle for only the first decision, a histogram of our improvements in value is shown for fair starting states in Figure 7.8(a) and for all non-terminal states in Figure 7.8(b).

the value gain from using the oracle is 0. The mean value of the one-decision oracle's information over all 35 fair starting states is 0.0041. Figure 7.8(a) shows a histogram of the values for each of the 35 fair starting states if the action oracle was available for only the first decision in those games. For all 2021 non-terminal states in the game, the average value of the one-shot decision made with the help of the action oracle is 0.0129. Figure 7.8(b) shows a histogram of the values for each non-terminal state if the action oracle was available for only a single decision at that state.

From these results we see that in general, the action-oracle does provide a value advantage but that the advantage does depend on which state the game starts in. A secondary interpretation of these results is that the action oracle is more important in some states than others. In other words, the value of the information about what the opponent will do next varies depending on which state we are in.

7.4 Approximating Model Value with Bayesian Opponents

While powerful in that it can calculate the minimum bound on the expected value of a perfect model in an environment, the game-theoretic method used above is limited by several assumptions:

- The following Nash Equilibrium preconditions must hold for the value to be meaningful: The game must be two-player; general sum; each player knows the possible actions the other could take.
- The Nash Equilibrium strategies must be calculable with available resources.

Furthermore, the policy model analysis is limited in that if the opponent is playing a true Nash Equilibrium strategy, the policy model will necessarily have a value of zero. When the preconditions listed above do not hold, we may be better off approximating the environment-value using an estimation of the distribution over likely opponents, which is the focus of this section.

We now present an alternative technique for approximating value of a model when game-theoretical solutions are intractable or inappropriate but the set of opponents is drawn from a known distribution of known opponent types. The steps for computing the value of a model for a probability distribution over known opponent types

is actually a generalization of the game-theoretic method described previously. Essentially, in the game-theoretic method, we made the assumption that our opponent was rational in the game-theoretic sense and that they would always play the Nash Equilibrium.

In a more general description of our approach, we assumed that the probability that our opponent was a Nash Equilibrium player was 1.0, and then we used the Nash Equilibrium solution to determine the behavior of our opponent without considering any other possible opponents. In the Bayesian technique we are about to describe, we instead use a probability distribution over multiple opponent types.

We first collect and instantiate a representative opponent from each type. For each opponent type, we create an oracle which would reveal the state, action, or policy of the opponent type in a transformation of the original game where the target information remained hidden. We then create the two versions of the environment: the original one in which our agent has no access to the opponent's information, and the transformed one in which an oracle will provide one of the missing pieces of information about the opponent.

To find an approximation of the best response in the original environment, we then use each of the instantiated opponents within an M^* [20] search[11] to find the optimal action to take within the game. We generate a static best response policy for each state for each opponent type. To find an approximation of the best response in the transformed environment, we simply reveal the missing information to our agent within the M^* search similarly to the way the action information was revealed in the previous section. We then compute the new optimal action to take for each state for each opponent. Then we can calculate the difference between the value of each state (in the original and transformed environments) for each opponent. We incorporate the probability distribution (over opponent types) into the sum of the differences in order to estimate the improvement of a particular model type for that distribution of opponent types.

In the previous section, we could make some guarantees about the value earnable by a perfect model of the opponent. Because a Nash Equilibrium player is going

[11]Recall that the M^* search inserts a model of the opponent into the search algorithm instead of assuming a perfectly optimal opponent. In this way, an expected-value-based search engine can compute the best action to take against the given opponent.

to play the most secure strategy (when the preconditions for equilibrium have been met) we can assume that for any opponent not using the Nash Equilibrium technique we can earn at least that much, and possibly more if we had an oracle to consult. But if our analysis technique does not rely on assuming our opponent will play Nash Equilibrium, this guarantee no longer holds. Two caveats must be made about the value of the model derived using the Bayesian technique:

- The value improvement computed is reliable only for the distribution over opponent types used in the calculation. If the distribution of actual agents differs from the distribution used in the computation, then the actual value improvement could be different as well.
- The value calculated with the Bayesian technique is no longer a function that depends solely on the environment. In the Bayesian technique, elements of the weaknesses of the opponents are included in the computation and can contribute to an increase in the apparent value of the model.

In Chapter 6 we demonstrated techniques for measuring the prediction performance of an opponent model. This chapter added the ability to determine the value of a particular class of opponent model for a given environment to our toolbox. In the next chapter, we explore another set of tools that can be used to improve opponent models during development and during live engagements.

Chapter 8

Improving Opponent Models

Much of the effort we have discussed to this point has been focused on how to determine the performance bounds of a class of opponent models and how to measure the actual prediction quality of an instantiated model. In this chapter we describe tools and techniques designed to help the model designer improve their agent when the set of opponents, and possibly even the distribution over opponent types is unknown.

We begin in Section 8.1 with a description of novel tools we can use to facilitate improving agents and discuss how the measurements presented in Chapter 6 can be applied to guide and assess the process. In Section 8.2 we present the manual technique we used to rank order agents by quality. We conclude in Section 8.3 by presenting a system for automatically selecting good variable settings and function implementations for models as well as a framework for metareasoning[1] over an ensemble of existing models during live adversarial engagements.

8.1 Tools

When the set of opponents is unknown and observations of histories of their behavior are unavailable, we must rely on adversarial targets that we create to train and test our agents. One of the best ways for accomplishing this is through exploration of the

[1]Metareasoning is "reasoning about reasoning": it attempts to estimate the cost-value benefit of first level reasoning in order to determine if computations (such as queries to opponent model predictions) are worth computing by how much they will improve our current situation.

effects of variable settings and function implementations[2] on the performance of the agent. To prepare for the exploration, we must have a set of parameters which can be adjusted and a performance measurement.

In many software development projects, parameters are constant and hardcoded based on the developers design decisions. Furthermore, sections of code containing control flow logic are static so that performance is consistent from run to run. During the development phase however, the developers may test different versions of the parameters in order to find the combination of best performance and speed.

We use a term called *advanced parametric control* to describe a formal method which allows us to change the parameters and the switch between various functions while attempting to develop the highest quality agents. To use advanced parametric control we take the following steps:

1. Determine all of the parameters and functions of interest (ones in which we think performance and speed depend the most)
2. For each parameter, define a small set of test values[3].
3. Separate the logic which controls the selection of the target parameters from the remainder of the code so that it can be easily changed[4].

If we enumerate all the parameters and assemble them into a vector, the identity of each agent is the vector of parameter values that are used in that agent. Our next step will be to determine the relative quality of each agent. Once the set of possible agents has been defined, we have the collection of possible members that could exist within the population.

While prototyping systems, researchers often employ instrumentation: measuring devices which report on various aspects of the system. In software engineering, one of the most rudimentary methods for debugging is to add "print statements" to report the values at various points in the running code. We use a similar method to

[2]For the remainder of this chapter we will use the generic term *parameters* to represent all of the variable settings and function implementations in the specified software module.

[3]When a parameter is referring to a function, the test values are various labeled implementations of that function.

[4]For function selection, one can use a single parameter to specify which labeled implementation of the function should be used when the program runs.

instrument our opponent models so that the belief state of the opponent model can be examined and compared to the actual opponent behavior after the interaction. Our goal is to examine how well model made predictions about the opponent.

For example, in PokeMinn, our agent for the 2007 Computer Poker Competition, we instrumented the opponent model's prediction of the opponent's next action. We then compared the prediction of the opponent's next action to his actual action to develop a performance quality measure: $\frac{c}{t}$ where c was the number of correct predictions and t was the total number of predictions. This allowed us to separate the performance of the opponent model from the performance of the entire agent and assess the relative quality of members in the population of candidate models.

Later, we developed the Weighted Prediction Divergence measurement described in Chapter 6. This measurement allowed us to assess the similarity between the distribution of predicted actions and the distribution of actual actions of the opponent. This measurement can be used as an alternate for the performance quality measure discussed above. By evaluating the quality of the predicted distribution instead of the accuracy of specific predictions, we obtain a more useful measure of how well the model will perform when estimating expected value for a branch deep in the game tree. For extensive-form environments such as poker and real time strategy games, having high-quality predictions of expected value is vital.

8.2 Techniques

In the previous section, we described a method for identifying potential candidates for a population pool and how to assess their relative quality. We now discuss the methods we can use to find the candidate that is likely to perform the best from the pool when the pool is large. We begin with a description of the manual methods which approximate hill climbing in performance-space. We focus on techniques that will work well when the evaluations between population members are costly in terms of time and computational resources.

First, we define the fitness test for the adversarial environment: Given a population of adversarial candidates A and an environment E, select two candidates from the pool and observe their instrumented performance in the environment. Order their relative performance.

Formally, we define G, a k-dimensional vector by concatenating the label strings for the p parameters. We then create all possible vectors of settings in G and pairwise-compare each instantiation. The score of each vector is determined by average measurement value from all of the pairwise competitions, and the optimal performer is the one which has the highest average measurement.

This brute-force procedure works when the number of possible vectors in G is small. Unfortunately, the number of comparisons required to be made grows quickly. Given p parameters, with the number of unique values per parameter $\phi_n, n = 1..p$ there will be a total of

$$\prod_{n=1..p} \phi_n$$

possible instantiations of code that could be created in order to implement the opponent model. Since there are $|A| = \prod_{n=1..P} \phi_n$ unique agents that could be formed in a population pool, there will be $\binom{|A|}{2}$ combinations of the agents when two are selected from the pool for competing. If we are planning to use a brute force method to compare all pairs of agents, it is extremely important to minimize the number of settings and the number of choices from each setting prior to running the pairwise performance tests. We now discuss some heuristic techniques for culling parameters (and reducing the number of settings) prior to running the brute-force optimization problem.

If we consider each parameter or each function implementation a unique dimension in a multi-dimensional vector optimization problem, one way to reduce the number of dimensions (and the size of each dimension) is with a pilot study of the effects of the various settings. Our goal is to determine which dimensions (or settings within a dimension) have little effect on performance so that we may cull those dimensions from the main optimization problem.

To perform this preprocessing step, we first select random values for all of the parameters in G and label the population member G_0. We select the first element in the vector as the element of interest. We then create several population members $G_{1_{1..n}}$ by selecting all n possible settings for element i. If n is large, we may choose an arbitrary subset of the n possible settings. We then compare the members (via pairwise or tournament competition) and record the results. We repeat the process for each of the other elements in turn. After all elements have been examined, we

rank order the element indices by the level of influence each element had on the performance outcome. We also look for performance plateaus across choices for an individual setting: if there are 3 values for a parameter and two of them yield the same performance, then we assume that we only need to look at 2 of the values if the setting is actually important for the full optimization problem. We then choose the highest ranked elements from G as for the optimization problem based on availability of resources. This preprocessing will take

$$\sum_{i=1..k} \binom{n_i}{2}$$

comparisons to determine results. In general the number of comparisons required during the preprocessing step is insignificant when compared to the full-scale optimization problem and the number of settings and values required can be greatly reduced.

We used this procedure successfully in optimizing PokeMinn prior to entering it in the 2007 Computer Poker Competition. Our settings initially included 5 functions (with two implementation choices each) and a variable in which we explored 3 values. Originally we would have had $(2^5)3 = 96$ possible population members to check which would require $\binom{96}{2} = 4560$ comparisons. At approximately 1 hour per test, we did not have the resources available to perform the full brute-force pairwise comparison necessary to obtain the optimal parameter settings. Fortunately we were able to eliminate three of the function choices and one of the parameter settings because they appeared to have little effect on the performance in some initial pilot tests. As a result, we were able to reduce the number of possible population members to $(2^2)2 = 8$ with a total of $\binom{8}{2} = 28$ comparisons to make. When the comparisons were complete, we selected the top candidate from the population. The preprocessing step took just $1+1+1+1+1+3 = 8$ comparisons. Our main optimization problem was reduced from 4560 comparisons to 28+8=36 comparisons to evaluate. Even with the preprocessing included, less than one percent of the computational resources of the original problem were required for the new optimization.

Given specified resource constraints this process is a viable heuristic for locating important dimensions and eliminating suspected less-important ones to generate a reduced optimization problem. Unfortunately this heuristic procedure is not guaran-

teed to yield a globally optimal performing system, or even a locally optimal solution[5]. We may have removed a dimension (or a value within a dimension) that would be important in the full optimization. One possible method for reducing the likelihood of eliminating the wrong dimensions or values is to run the preprocessing step several times with several different randomized starting vectors used for G_0. While this process still does not guarantee optimality, it reduces the likelihood that single bad starting G_0 leads to incorrect conclusions about which elements and values are unimportant.

8.3 Automating the Process

In the previous section we describe a manual procedure used offline to improve an opponent modeling system (and by extension, an entire adversarial agent) when the set of opponents is unknown. We detailed the heavy computational requirements of this procedure and highlighted the core problem with the heuristic adaptation of this procedure used when computational resources are limited - no guarantee of optimality. We begin this section with a discussion on automating the offline optimization problem using a method with more desirable convergence properties. We conclude the section (and the chapter) with a brief presentation of a metareasoning system which could potentially achieve strong opponent modeling results in the actual online adversarial environment.

In the previous section, we purposely chose to describe an agent in terms of a parameter vector. We can think of this vector as a genome which is used to identify a particular agent by its settings. Given a definition of the genome which is used to describe agents, we realize that there exists a class of evolutionary algorithms which are designed to optimize the performance of a population of such agents: Genetic Algorithms. We will now describe the framework for this optimization problem, but the implementation of it is left for future research.

[5]By *locally* optimal, we mean that changing the parameter setting for any single dimension would not improve performance. Since we may have eliminated the dimension of interest in the preprocessing step, we may miss the opportunity for achieving this condition.

8.3.1 Genetic Algorithms

Several researchers have explored using genetic algorithms to manage software development. For example, in [32] Garcia's team explored using genetic algorithms to simulate potential effects of software engineering experiments when variables within the software were perturbed without actually running the experiments. Others have explored using genetic algorithms to evolve software in successive generations by directly manipulating the functions in the software. Stanley, et al. developed a genetic algorithm which can modify a population of neural networks [71]. In successive generations the genetic algorithm can add new nodes and control the existence of connections between nodes based on the fitness of prior networks. They deployed the members of the population in an interactive video game (NEROgame). The team showed that the population could evolve in real time to handle tactical battlefield tasks such as navigating around obstacles, avoiding changing threats and achieving complex goals.

In this section, we propose using genetic algorithms to manage quality assessment of candidate agents by directly manipulating the software parameters and attempting to optimize the performance over several evolving generations. A genetic algorithm requires a method of encoding the members of a population, a fitness function, and a method of culling the population and generating new members. The encoding for each member of the population was described previously in terms of the vector which encodes the settings for an adversarial algorithm. The fitness function is the prediction performance of the opponent model (or the overall performance of the agent). We describe each of the remaining genetic algorithm phases in turn:

The initial population of P members is created by randomly instantiating $|P|$ separate (but not necessarily unique) settings for the genome G. In general, $|P|$ should be a small fraction of the total number of possible settings for G.

While the size of the population may be large, the number of pairwise comparisons that can be made between population members is necessarily constrained. We cannot determine the fitness of every member in the population. Given the computational availability of $\binom{n}{2}$ comparisons per generation, we randomly select n members from the population sample for fitness testing. We then rank-order the fitness of the n members. The remaining members of the population are rank-ordered based on a function of their similarity to the n ranked members.

We create the offspring of the population by first randomly selecting m members of the highest ranking members for parenthood. We create m new members of the population from these parents in the following way: in each child, each element is copied from a randomly selected parent (randomized crossover).

We eliminate the m lowest ranking individuals from the population and replace them with the m children just created. Next, we select a small subset of the total population for mutation. The mutation process occurs by randomly altering one or more of the elements in the genome for each selected member.

We continue the process until either the population converges or we reach the limits of our computation allocation. The result from this algorithm should be a set of strong agents, although no guarantees can be made for global or even local optimality because of the traits of genetic algorithms. Given that the major computational effort is expended during the fitness testing (such that the other computations can effectively be ignored), this algorithm will require $c\binom{n}{2}$ comparisons where c is the number of iterations the algorithm runs. The exact settings for P, n, m and a limit on the maximum number of iterations for c would need to be tuned for each problem in order to encourage convergence while keeping $c\binom{n}{2}$ feasible given computational resources.

8.3.2 Metareasoning

While the previous discussion in this chapter has been focused on metric-guided development of opponent models and agents in the offline setting prior to an actual adversarial encounter, we now explore another important question in opponent modeling: how do we decide which opponent models will be most cost effective (calculation wise) during an actual encounter?

Given a set of available opponent models, we would like to know which model of our current opponent would yield the best predictions in each possible world state of the domain. However, when we have limited resources and each prediction query has a cost we may need to decide which queries to pursue using only estimates of their benefit and cost: metareasoning. To estimate the benefit of the computation, a metareasoner needs a robust measurement of performance quality. In the remainder of this section we introduce a metareasoning framework that relies on a prediction performance measurement, and we show how the performance measurement introduced

in Chapter 6 fulfils this need.

When computational cost is an issue, we must answer several questions in order to build a computationally efficient system:

- How do we automatically generate a contextual abstraction[6] of the world state space?
- How can we estimate a model's general prediction quality in a specific context without measuring quality at every possible world state within that context?
- How do we compare the relative performance of several heterogenous models' predictive capabilities in a specific context?

We propose a metareasoning system that could be coupled with a robust measurement of model prediction performance (such as the one presented in Chapter 6) to yield a powerful new method of model selection for an agent. We then introduce one target domain for our empirical testing of this prediction performance measurement. We discuss empirical results using this method and conclude with a description of future work required to realize the overall metareasoning system.

Consider an agent trying to find high-value actions to take in some world where the value of the action is partially dependent on the behavior of the other agents in the world. We describe the behaviors of the agents (and the way those behaviors affect and are affected by the world) using an extensive-form game tree where nodes describe world states and edges describe actions taken by specific agents. The value of a world state can be estimated by summing the values of its branches, weighted by the likelihood of the branch (action) being chosen. This process can be used at each sub-branch to find the value of the sub-branches recursively. The recursion bottoms-out when a branch is terminated at a leaf node that contains a real-world value. Each node in the tree may also have an intrinsic value which can be included in the calculation.

The well-known minimax algorithm discussed in [62] describes this process when there are two agents, the game is zero-sum (in every outcome one player gains exactly the

[6]For this work, we define a *context* to be a collection of world states. Forming a set of contexts from the domain's world states requires a method of abstraction.

value the other player loses) and the agents are assumed to have perfect (not bounded) rationality. Many extensions to minimax such as [20, 51, 75, 77] have been developed which allow the assumption of perfect rationality to be relaxed. By replacing the minimizer with an algorithm that predicts the opponent's behavior at a given node as in [20] we obtain an algorithm which selects actions with the highest value when playing against a specific modeled opponent. When the model is estimating the other agent's probability distribution over actions, the quality of the value calculation is dependent upon how similar the predicted behavior distribution is to the actual behavior distribution.

If we want to have a metareasoner that makes decisions about which calculations to carry out using the utility of a calculation as a criteria, we need to know the cost of the calculation and the value of the calculation [63]. In the setting described above, the relative value of several candidate calculations (queries of different predictive models) can be estimated by the quality of their predictions at the specified world state. Unfortunately, measuring the predictive quality of every model at every possible world state in a branch of the game tree is at least as expensive as obtaining all the predictions in the first place - thus computing prediction quality for every node in a branch would not yield any resource savings. In order for our metareasoner to be useful, it must be able to *estimate* the prediction quality for each node without fully calculating prediction quality.

The metareasoner uses an abstraction[7] of the world states such that every world state belongs to one of a comparatively smaller number of the *contexts* in the abstraction. As the agent makes decisions within the environment, the behavioral predictions made by a model and the resource usage (CPU load and total time, for example) are monitored by the metareasoner, and accumulated in the abstract context associated with the prediction's real world state. The metareasoner also monitors the target agent's observed behavior and accumulates the probability distribution in the abstract context representing the world state the observation was made in. Whenever the agent needs to determine the value of a proposed action, the metareasoner first updates the relative prediction utility of each model in each abstract context using the collected data from past predictions and observations. The matrix of estimated prediction utilities (utility for each model in each context) is then passed to the object-level

[7]While the method of generating the abstraction is beyond the scope of this work, there is much existing work on automated abstraction techniques. For example, see [35, 36, 37, 38, 39]

action value calculator. The object-level can then choose the highest-utility models for each prediction node visited during the recursive value calculation over sub-branches from the current world state in the game tree.

We now define the architecture of a metareasoning agent designed to act within the multi-agent environment described previously. At the highest level, the system is one in which the meta-level device is monitoring the performance of the object-level reasoner and computing the utility of reasoning methods, as depicted in Figure 8.1.

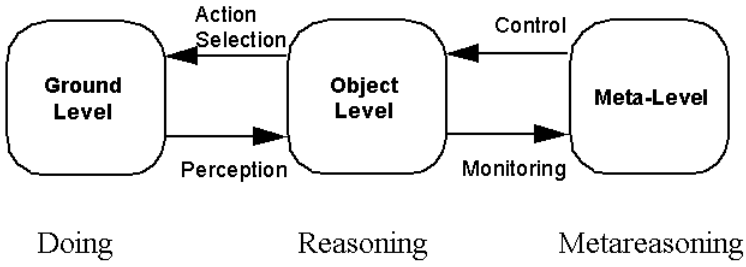


Figure 8.1: Overview of a Metareasoning-controlled agent. This diagram originally used in [24]

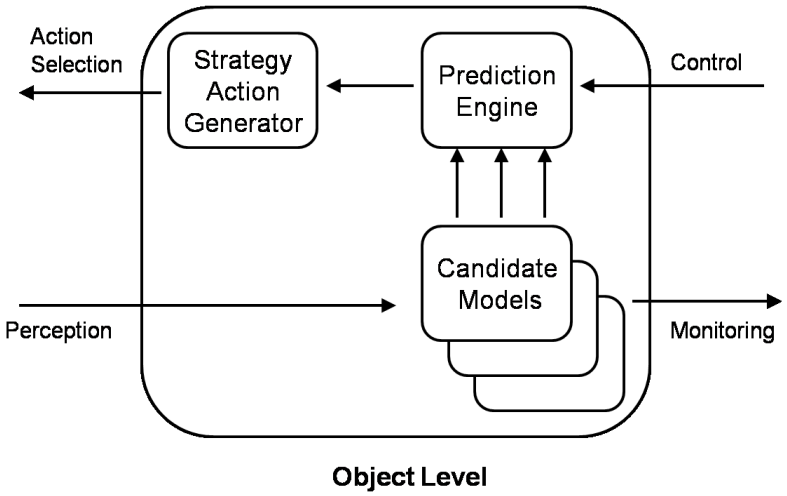


Figure 8.2: Object Level. The object's candidate models observe the action of other agents and provide predictions to a prediction engine. The prediction engine is controlled by metareasoning, based on the meta-level monitoring of the performance of the candidate models. This diagram originally used in [17]

We look inside the reasoning object in Figure 8.2. There are a number of candidate models which receive input from the environment (and observe the actions of the

other agent[8]). We assume that each candidate model is a black-box: it is opaque to us and it takes inputs and produces outputs in the form of a probability distribution over the possible actions the modeled agent could take. Furthermore, we assume that the models are input-heterogenous but output-homogenous: each model is trying to predict a distribution of what the modeled agent will do next, but each may be considering different information from which to make their predictions.

At the object level, the Strategy Action Generator is attempting to discover fruitful strategies for future behavior of the agent. In order for the Strategy Action Generator to choose a high-expected-value future strategy (sequence of actions and responses by other agents) it may need to consider many "what if" predictions about various events that could occur in the future in order to evaluate different possible strategies. These predictions vary, depending on the abstract context (collection of world states) they occur in. Depending on the context of the desired prediction, some models may have better prediction performance than others. Thus, the selection of a predictive model (or weighted distribution over models) should be *context-aware* for *each* prediction. For each prediction-in-context that must be made, it is the Prediction Engine's job to generate the overall prediction using the set of available candidate models.

The Prediction Engine's selection of model(s) is influenced by the metareasoning level. The metalevel shown in Figure 8.3 is composed of several modules.

The first module, the Context Abstraction is a device which determines the appropriate abstract context from a given world state. It provides this context as a tag for the Activity Repository for every prediction and observation event. Whenever a model is queried for a prediction in the object level, it generates a prediction event. Prediction events contain the predicted distribution over modeled agent actions and the cost incurred by that model to generate the prediction. Observation events include the observed behavior of the modeled agent. The Utility Calculation first computes the distribution of agent behavior in each tagged context. Then it calculates a prediction quality measurement from the predictions and observations seen within the context. Finally it computes the utility for each model in each context from the prediction quality and the prediction cost. The latest utility profile matrix is then passed to the

[8]For the sake of linguistic clarity, we will refer to only one agent being modeled to avoid confusion between the agent and the multiple candidate models used to predict the actions of that agent. While not discussed further in this work, the metareasoning arrangement proposed here can be used to model multiple target agents.

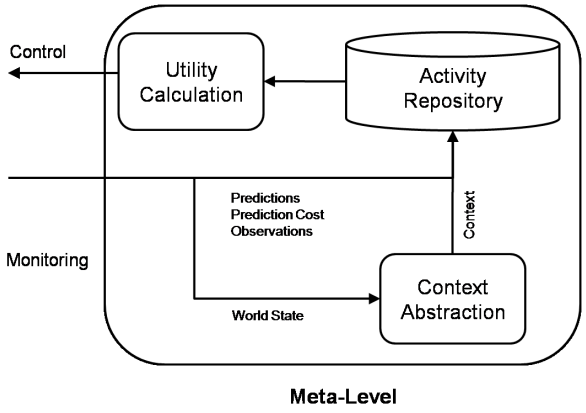


Figure 8.3: Meta-Level. The metareasoner reviews past predictions, prediction costs, and observations to yield a utility measurement for each context. The utility calculation advises the prediction engine by providing a utility profile for each model in each context. This diagram originally used in [17]

Prediction Engine prior to each prediction.

In the previous paragraph we mentioned that the Utility Calculation requires a prediction quality measurement which can compare the distribution of predicted actions to the distribution of observed actions within each context. The Weighted Prediction Divergence metric we presented in Chapter 6 provides exactly this calculation. Thus, we invoke it here in order to complete the requirements for the metareasoning framework.

In Chapter 6 we showed how prediction quality assessments can be used to point out strengths and weaknesses in opponent models that remain hidden under domain-based performance measurements. We identified the desiderata of a strong measure of model quality, which are also necessary for metalevel reasoning in the multi-model agent prediction setting. In that chapter we demonstrated empirically how Weighted Prediction Divergence values for several models could be used to select the best model for the current context. By extension, we can invoke the WPD for use when the contexts are automatically generated.

There is at least one additional effort that must be completed before the metareasoner described in this section can be realized. We must choose a state abstraction method which can automatically cluster world states into contexts such that the prediction quality estimation system described in this work can assess models with fewer computational resources. While we have shown a meaningful context for poker in Chapter 6,

the contexts for the collection of nodes we used were generated manually. Automating this process would enable multi-domain applicability for the metareasoning technique described here.

Chapter 9

Conclusions & Future Work

Throughout this work we have explored various aspects of opponent modeling in interesting adversarial environments. Our main contributions have focused on answering several important open questions in the field:

1. How does one measure the quality of opponent model predictions separately from an agent's overall performance?
2. How does the environment and the type of opponent model chosen impact the value of the information it provides?
3. How can we improve opponent models when the set of opponents is unknown?
4. Given a set of candidate models, how can we use them efficiently and effectively in when faced with a live opponent?

We used tools from game theory, machine learning and information theory to provide initial answers for these questions. We empirically tested our concepts in several domains, including simultaneous high card draw, full scale Texas Hold'em poker, and a new simultaneous-move strategy game we developed as a testbed. We now outline the contributions of this work, describing the research question they answered, the techniques used, the rationale, and the domain where we showed how well they worked.

9.1 Major Contributions

9.1.1 Separating model performance from agent performance

This contribution demonstrates the importance of assessing performance quality at a deeper level.

To isolate the prediction performance of a opponent model from the overall performance of an agent, we developed and implemented a novel metric which we call Weighted Prediction Divergence. This metric measures the similarity between the distribution of predictions and the true distribution of behavior of the opponent. It can evaluate predictions from multiple prediction nodes and generates a single scalar quality measure. It can be weighted using various schemes to incorporate a domain-specific importance function so that the quality measure reflects key aspects of the domain.

We adapted Weighted Prediction Divergence to evaluate the performance of our PokeMinn agent in the 2007 Texas Hold'em Computer Poker Competition in Chapter 6. Using the measurement, we demonstrated how to find the weaknesses of the Bayesian-update learning opponent model by revealing in what contexts it was outperformed by a non-learning opponent model. It was interesting to find this performance problem when the agent-level performance of the learning model appeared to be consistently better than the agent-level performance of the non-learning version. This finding reenforces the importance of directly measuring model performance, and in this case, showed us of the need for additional improvements in the learning version in certain circumstances.

9.1.2 Discovering Environment-invariant Model Performance Bounds

This contribution shows that the potential performance of an opponent model is constrained by the domain where it operates.

We defined the concept of an opponent model value performance bound which is a function of the environment alone in the first part of Chapter 7. We first transform the original environment into a new environment where the information provided by the model is provided as part of the environment. Then by solving the environments to determine perfect play (and value of perfect play) in each, we can determine the value difference between the environments. This difference is *the value of the model*

when the model is perfect. In other words the value is a guaranteed minimum bound on the performance of a perfect model.

We implemented algorithms for determining the performance bounds of an opponent model and we tested these algorithms in two domains:

- High card draw with simultaneous betting. A two-player zero-sum hidden information game with a simultaneous actions.
- A simultaneous-move strategy game. A new indefinite-length perfect-information simultaneous-action strategy game we created in which the set of actions available for each player depends on the history of play so far.

In both of these environments we showed how to calculate the guaranteed value bounds for a class of opponent models. A class of opponent models is one in which the prediction type for all models in the class is identical. We found that for each class of opponent model (action, state, or policy), there is one guaranteed value improvement bound that is independent of which opponent we face and depends only on the nature of the environment.

In high card draw we examined three classes of models: the action-prediction model; the state-prediction model; and the policy-prediction model. We discovered that in this game, the values for a class of models that predicts the next state had the highest potential, while the class of models that predict the action was valuable, but somewhat less than the state-predictors. The policy-prediction model was found to have a guaranteed improvement of zero because if the opponent was playing an uninformed[1] equilibrium policy, then his policy doesn't matter to us: both players will have an expected value of zero for the game. We showed how manipulating the amounts required for bets and antes in the game (the nature of the environment) directly affected the guaranteed value improvement bounds. This finding has implications for mechanism design of new games when some aspect of the opponents may be predictable.

[1]By uninformed, we mean a strategy which has no information about the specific opponent's characteristics. The uninformed agent assumes his opponent is rational and the equilibrium becomes desirable for describing strategies that guarantee some minimum value in the environment when all players are acting rationally.

We developed a new simultaneous-move strategy game which is a simplification of the well-known "Real Time Strategy" genre of human-player computer games. This game is a simultaneous-action, perfect information game, and the only valuable type of opponent prediction for this game is the prediction of the opponent's next action (since the state is known and knowing the opponent's policy cannot guarantee a value improvement.) We found that the game was certainly easier when the opponent's next action was known perfectly on every move, but that the improvement guarantee was marginal. We also discovered that the value improvement for having a perfect opponent model varied widely depending on which state the game started in.

9.1.3 Preparing for Unknown Opponents

This contribution presents methods for tuning opponent models and agents to prepare for interactions with unknown opponents.

In Chapter 8 we presented new tools and techniques for improving opponent models when the set of opponents is unknown and observations of their history is unavailable. The methods we explored focused on software engineering concepts for isolating important parameters and functions likely to affect opponent model performance during self-play. We revealed that while brute force comparison of all possible combinations of settings of the parameters and functions is the only guaranteed way to find the best-of-breed performer, the computational requirements of this process grow exponentially with the number of parameter and function settings.

We then presented some heuristic-based methods for searching through parameter space in an effort to find good performance. While these techniques moved us towards computationally tractable optimization, the unknown relation between parameter settings and performance presents a significant challenge for optimization. Furthermore, because some parameters and settings are culled when using the heuristic, this method cannot guarantee a locally optimal solution.

Nonetheless, we used these techniques to tune our PokeMinn agent prior to the 2007 Computer Poker Competition with great success. Our full scale Heads-up Limit Texas Hold'em Poker-playing used an opponent model with a Bayesian update method for learning the opponent's distribution over actions in an abstraction of the state space. We tuned the agent using techniques described above. This agent performed well in the competition, taking fourth place out of a field of fifteen entries in the Limit

Equilibrium category, despite having no observations of the competitors prior to the contest.

9.1.4 Efficient and Effective Modeling Against Live Opponents

This contribution provided techniques for assessing and improving the efficiency and effectiveness of opponent modeling in live situations.

The method we used to update PokeMinn's opponent model was efficient and effective because it attempted to learn only a fraction of the most important information in the opponent's behavior. As a result we were able to start improving our performance against previously unseen opponents within a few hundred hands (as opposed to thousands of hands required in other learning agents). To characterize the learning profile we developed a non-learning version of our PokeMinn agent and used a differential performance measurement to isolate the value contribution of the learning component from the value contribution provided by the remainder of the agent. We showed in Chapter 4 that PokeMinn is able to improve its performance through observation of live play, even when the opponent is playing a near Nash-Equilibrium strategy. This finding confirmed that there is merit in learning a small subset of the opponent's characteristics, even if the opponent is presumed to be near optimal.

9.2 Future Work

Interestingly, the Weighted Prediction Divergence metric we developed in Chapter 6 has a secondary use: under certain conditions, it can be used to detect when the opponent is not playing a Nash Equilibrium. In particular, if we know what the Nash Equilibrium distribution over actions should be at an opponent's decision point, we can use the unweighted version of this measurement to detect a deviation between the expected Nash Equilibrium distribution and the observed distribution. If the prediction divergence is statistically significant (for the number of observations) we can be confident (to the significance level) that the opponent is not playing the Nash Equilibrium at that decision point. This finding may be important when deciding whether or not to attempt opponent exploitation. Implementing an agent that uses this equilibrium-deviation detector remains future work.

In the early portion of Chapter 8 we presented some manual methods for tuning

agents. We later showed how one might automate the search by mapping the software design problem to a genetic algorithm-based search problem. A genetic algorithm provides the hope of a more efficient and higher quality optimization of parameters than could be provided by the manual methods we described. The implementation and testing of a genetic algorithm-based method is yet to be accomplished.

We concluded Chapter 8 with a framework for metareasoning using the Weighted Prediction Divergence measure we developed in Chapter 6 and an abstraction[2] of the state space. The metareasoner we envision in [17] uses an ensemble of black-box opponent models and keeps track of their performance quality and computational expenditures for each context in the abstraction. It then computes the cost-benefit tradeoff for each model in each context such that when a prediction is required, the most cost-effective and efficient set of models can be determined and queried. This framework has the power to harness the strengths of multiple models in an online setting when computational resources (including time) are constrained. The implementation of this framework remains a future task.

The guaranteed value improvement bounds we developed in the first part of Chapter 7 are a function of the environment alone: they are independent of the policy of the opponent and depend upon the assumption of opponent rationality implemented via simulating an opponent using an uninformed equilibrium strategy. In our case studies we assumed that the opponent would be aware of the fact that his information was exposed. One venue for future work would be to determine what the effects are on the bounds calculation if we transformed the environment into one in which the opponent was unaware that there was an oracle providing his information to the other player.

In the second part of the Chapter 7, we relaxed the assumption that the opponent was using an uninformed equilibrium strategy and we postulated that we could develop a tighter guaranteed bound when we considered a distribution over opponent policies instead of the single uninformed equilibrium policy. The limitations of this technique are two-fold: it violates the condition that the value improvement bounds are a function of the environment alone; the bounds are no longer guaranteed if the actual distribution over opponent policies is different from the distribution used in the calculation of the bound. Even with these limitations, the method has the poten-

[2] An abstraction is a partitioning of the state space such that each state in the original environment maps to exactly one abstract state and each abstract state may contain several of the original states.

tial for much more accurate model performance bounds when the distribution over possible opponents can be accurately predicted. The implementation of this method remains for future work

All of the environments we explored were two-player zero-sum (strictly adversarial) settings. We chose these environments for the relative "ease" of calculating the uninformed equilibrium solution such that we could compare performance of the equilibrium strategy to the strategy that incorporates predictions of the opponent. In addition to the future work previously identified in this chapter, one direction for this research is to explore the impact of non-zero sum environments and multi-player environments on the contributions made previously.

While optimal in an uninformed sense, the equilibrium strategy is necessarily stationary, and as we mentioned in the early portion of the dissertation, is not always the strategy of choice in interesting adversarial environments. In future work, we would like to address the impact of non-stationary strategies on our techniques. We would also like to develop measurements which characterize how well (speed, accuracy) an opponent model can adapt to a non-stationary opponent policy and explore the differences between a non-stationary policy that is non-adversarial and one that is. Finally, we would like to determine if it is possible to anticipate the direction that the adversarial non-stationary policy will take and discover methods of doing so.

Bibliography

[1] Victor Allis. A knowledge-based approach of connect-four. the game is solved: White wins. Master's thesis, Vrije Universiteit, Amsterdam, The Netherlands, October 1988.

[2] Victor L. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, Maastricht, 1994. ISBN 90-9007488-0.

[3] Alon Altman, Avivit Bercovici-Boden, and Moshe Tennenholtz. Learning in one-shot strategic form games. In *The 17th European Conference on Machine Learning (ECML-06)*, pages 6–17, Berlin, Germany, 2006.

[4] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The non-stochastic multiarmed bandit problem. *SIAM Journal on Computing*, 32(1):48–77, 2002.

[5] Robert J. Aumann. Subjectivity and correlation in randomized strategies. *Journal of Mathematical Economics*, 1:6796, 1974.

[6] Robert J. Aumann and Adam Brandenburger. Epistemic conditions for Nash equilibrium. *Econometrica*, 63:1161–1180, 1995.

[7] Robert J. Aumann and Sergiu Hart. Long cheap talk. *Econometrica*, 71:1619–1660, 2003.

[8] Elchanan Ben-porath. The complexity of computing a best response automaton in repeated games with mixed strategies. *Games and Economic Behavior*, 2:1–12, 1990.

[9] Darse Billings. Competition results for the the first international roshambo programming competition. web page (unreviewed), 1999.

[10] Darse Billings, Neil Burch, Aaron Davidson, Robert Holte, Johnathan Schaeffer, Terrence Schauenberg, and Duane Szafron. Approximating game-theoretic optimal strategies for full-scale poker. In *International Joint Conference on Artificial Intelligence*, pages 661–668, 2003.

[11] Darse Billings, Aaron Davidson, Jonathan Schaeffer, and Duane Szafron. The challenge of poker. *Artificial Intelligence*, 134(1-2):201–240, 2002.

[12] Darse Billings, Aaron Davidson, Terence Schauenberg, Neil Burch, Michael Bowling, Robert Holte, Jonathan Schaeffer, and Duane Szafron. Game tree search with adaptation in stochastic imperfect information games. In *Proceedings of the 4th International Conference on Computers and Games (CG), Ramat-Gan, Israel, July 2004.*, pages 21–34. Springer-Verlag, 2004.

[13] Darse Billings, Dennis Papp, Jonathan Schaeffer, and Duane Szafron. Opponent modeling in poker. In *Fifteenth National Conference on Artificial Intelligence*, pages 493–499, 1998.

[14] Darse Billings, Lourdes Peña, Jonathan Schaeffer, and Duane Szafron. Using probabilistic knowledge and simulation to play poker. In *Sixteenth National Conference on Artificial Intelligence (AAAI)*, pages 697–703, 1999.

[15] R. V. Bjarnason and T. S. Peterson. Multi-agent learning via implicit opponent modeling. In *Proceedings of the 2002 Congress on Evolutionary Computation (CEC '02)*, volume 2, pages 1534–1539, 2002.

[16] Nate Blaylock and James Allen. Recognizing instantiated goals using statistical methods. In Gal Kaminka, editor, *IJCAI Workshop Modeling on Others from Observation (MOO-2005)*, pages 79–86, Edinburgh, July 2005.

[17] Brett Borghetti and Maria Gini. Weighted prediction divergence for metareasoning. In Michael T. Cox and Anita Raja, editors, *Metareasoning: Papers from the 2008 AAAI Workshop*, volume WS-08-07, pages 94–101, Menlo Park, CA, 2008. AAAI Press.

[18] Michael Bowling and Manuela Veloso. Multiagent learning using a variable learning rate. *Artificial Intelligence*, 136:215–250, 2002.

[19] Robert D. Broadston. A method of increasing the kinematic boundary of air-to-air missiles using an optimal control approach. Master's thesis, Naval PostGraduate School, Monterey, CA, September 2000.

[20] David Carmel and Shaul Markovitch. Incorporating opponent models into adversary search. In *Thirteenth National Conference on Artificial Intelligence*, pages 120–125, 1996.

[21] David Carmel and Shaul Markovitch. Learning models of intelligent agents. In *Thirteenth National Conference on Artificial Intelligence*, pages 62–67, 1996.

[22] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 2 edition, 2006.

[23] Michael T. Cox and Boris Kerkez. Case-based plan recognition with novel input. *Control and Intelligent Systems*, 34(2):96–104, 2006.

[24] Michael T. Cox and Anita Raja. Metareasoning: A manifesto. Technical Report TM-2028, BBN Technologies, Cambridge, MA, December 2007. www.mcox.org/Metareasoning/Manifesto.

[25] Aaron Davidson, Darse Billings, Johnathan Schaeffer, and Duane Szafron. Improved opponent modeling in poker. In *International Joint Conference on Artificial Intelligence*, pages 1467–1473, 2000.

[26] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Sixth International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 71–80, 2000.

[27] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. Wiley Interscience, 2001.

[28] Dan Egnor. Iocaine powder. *International Computer Games Association Journal*, 23:33–35, 2000.

[29] Yoav Freund, Michael J. Kearns, Yishay Mansour, Dana Ron, Ronitt Rubinfeld, and Robert E. Schapire. Efficient algorithms for learning to play repeated games against computationally bounded adversaries. In *Thirty Sixth Annual Symposium on Foundations of Computer Science*, pages 332–341, 1995.

[30] Bent Fuglede and Flemming Topsoe. Jensen-shannon divergence and hilbert space embedding. In *Proceedings of the International Symposium on Information Theory, 2004. (ISIT 2004)*, page 31, 2004.

[31] Ya'akov Gal and Avi Pfeffer. A language for opponent modeling in repeated games. In *Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS03)*, pages 265–272, Melbourne, 2003.

[32] Rogério E. Garcia, Maria C. F. de Oliveira, and José C. Maldonado. Genetic algorithms to support software engineering experimentation. In *International Symposium on Empirical Software Engineering, 2005*, pages 488–497, November 2005.

[33] Dino Gerardi. Unmediated communication in games with complete and incomplete information. *Journal of Economic Theory*, 114:104–131, May 2004.

[34] Itzhak Gilboa. The complexity of computing best-response automata in repeated games. *Journal of Economic Theory*, 45:342–352, 1988.

[35] Andrew Gilpin and Tuomas Sandholm. A Texas hold'em poker player based on automated abstraction and real-time equilibrium computation. In *Fifth International Conference on Autonomous Agents and Multiagent Systems*, pages 1453–1454, Hakodate, Japan, 2006. ISBN:1-59593-303-4.

[36] Andrew Gilpin and Tuomas Sandholm. Better automated abstraction techniques for imperfect information games, with application to Texas hold'em poker. In *6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '07)*, pages 1–8, New York, NY, USA, 2007. ACM.

[37] Andrew Gilpin and Tuomas Sandholm. Expectation-based versus potential-aware automated abstraction in imperfect information games: An experimental comparison using poker. In *23rd National Conference on Artificial Intelligence (AAAI'08)*, Chicago, IL, 2008. (to appear).

[38] Andrew Gilpin, Tuomas Sandholm, and Troels Bjerre Sorensen. Potential-aware automated abstraction of sequential games, and holistic equilibrium analysis of Texas hold'em poker. In *Twenty-second National Conference on Artificial Intelligence*, pages 50–57, 2007.

[39] Andrew Gilpin, Tuomas Sandholm, and Troels Bjerre Sorensen. A heads-up no-limit texas hold'em poker player: Discretized betting models and automatically generated equilibrium-finding programs. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS'08)*, Estoril, Portugal, 2008.

[40] Herbert Gintis. *Game Theory Evolving*. Princeton University Press, 2000.

[41] Bret Hoehn, Finnegan Southey, Robert C. Holte, and Valeriy Bulitko. Effective short-term opponent exploitation in simplified poker. In *Twentieth National Conference on Artificial Intelligence*, pages 783–788, Pittsburgh, Pennsylvania, 2005.

[42] Ronald. A. Howard and James E. Matheson. *Influence Diagrams*, volume II. Strategic Decisions Group, Menlo Park, California, 1984.

[43] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *Seventh International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 97–106, San Francisco, CA, 2001. ACM Press.

[44] Steven Jensen, Daniel Boley, Maria Gini, and Paul Schrater. Non-stationary policy learning in 2-player zero sum games. In *Twentieth National Conference on Artificial Intelligence*, pages 789–794, 2005.

[45] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[46] Daphne Koller and Nimrod Megiddo. The complexity of two-person zero-sum games in extensive form. *Games and Economic Behavior*, 4(4):528–552, 1992.

[47] Daphne Koller, Nimrod Megiddo, and Bernhard von Stengel. Fast algorithms for finding randomized strategies in game trees. In *26th ACM Symposium on Theory of Computing (STOC '94)*, pages 750–759, 1994.

[48] Daphne Koller and Brian Milch. Multi-agent influence diagrams for representing and solving games. In *Seventeenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1027–1034, 2001.

[49] Alexander Kott and William M. McEneaney. *Adversarial reasoning : computational approaches to reading the opponent's mind*. Boca Raton: Chapman & Hall/CRC, 2007.

[50] Michael L. Littman. Results from the 2006 computer poker competition. Internet (unreviewed), 2006. (http://www.cs.ualberta.ca/ pokert/2006/results.html).

[51] Carol A. Luckhardt and Keki B. Irani. An algorithmic solution of n-person games. In *Fifth National Conference on Artificial Intelligence (AAAI)*, pages 158–162, Philadelphia, PA, 1986.

[52] Richard D. McKelvey, Andrew M. McLennan, and Theodore L. Turocy. Gambit: Software tools for game theory, 2007. (http://gambit.sourceforge.net).

[53] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Higher Education, 1997.

[54] Yishay Mor, Claudia. V. Goldman, and Jeffrey. S. Rosenschein. Learn your opponents strategy (in polynomial time)! In *Proceedings of the Workshop on Adaption and Learning in Multi-Agent Systems*, pages 164–176, 1995.

[55] John F. Nash. Equilibrium points in n-person games. In *National Academy of Sciences*, volume 36, pages 48–49, 1950.

[56] John F. Nash. Non-cooperative games. *The Annals of Mathematics*, 54:286–295, 1951.

[57] John F. Nash. Two-person cooperative games. *Econometrica*, 21:128–140, 1953.

[58] Martin J. Osborne and Ariel Rubinstein. *A Course In Game Theory*. MIT Press, 1994.

[59] Patrick Riley and Manuela Veloso. Planning for distributed execution through use of probabilistic opponent models. In *Sixth International Conference on AI Planning and Scheduling (AIPS-2002)*, pages 72–81, 2002.

[60] Patrick Riley and Manuela Veloso. Recognizing probabilistic opponent movement models. In *RoboCup 2001: Robot Soccer World Cup V*. Springer Verlag, 2002.

[61] Collin Rogowski. Model-based opponent-modelling in domains beyond the prisoners dilemma. In *Workshop on Modeling Other Agents from Observations at AAMAS*, pages 41–48, 2004.

[62] Stuart J. Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, second ed. edition, 2003.

[63] Stuart J. Russell and Eric Wefald. Principles of metareasoning. *Artificial Intelligence*, 49(1-3):361–395, 1991.

[64] Jonathan Schaeffer, Neil Burch, Yngvi Bjrnsson, Akihiro Kishimoto, Martin Mller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, July 2007.

[65] Jiefu Shi and Michael L. Littman. Abstraction methods for game theoretic poker. In *CG '00: Revised Papers from the Second International Conference on Computers and Games*, pages 333–345, London, UK, 2002. Springer-Verlag.

[66] Yoham Shoham, Rob Powers, and Trond Grenager. If multi-agent learning is the answer, what is the question? *Journal of Artificial Intelligence*, 171:365–377, 2007.

[67] Gerardo Simari, Amy Sliva, Dana Nau, and V. S. Subrahmanian. A stochastic language for modelling opponent agents. In *Fifth International Conference on Autonomous Agents and Multiagent Systems (AAMAS '06)*, pages 244–246, 2006.

[68] Herbert A. Simon. *Models of Man: Social and Rational*. John Wiley and Sons, Inc., New York, 1957.

[69] David Sklansky. *The Theory of Poker*. Two Plus Two Publishing, fourth edition, December 2005.

[70] David Sklansky and Mason Malmuth. *Hold'em Poker For Advanced Players*. Two Plus Two Publishing, third edition, June 2005.

[71] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. Evolving neural network agents in the nero video game. In *IEEE Symposium on Computational Intelligence and Games (CIG'05)*, Piscataway, NJ, 2005. IEEE.

[72] Timo Steffens. Feature-based declarative opponent-modelling in multi-agent systems. Master's thesis, Institute of Cognitive Science, Osnabrck, 2002.

[73] Timo Steffens. Similarity-based opponent modelling using imperfect domain theories. In *IEEE Symposium on Computational Intelligence and Games (CIG 05)*, pages 285–291, 2005.

[74] Timo Steffens. *Integration von unvollkommenem regelbasierten Hintergrundwissen in Ahnlichkeitsmae (Enhancing Similarity Measures with Imperfect Rule-based Background Knowledge)*. PhD thesis, Institute of Cognitive Science, Osnabrck, 2006.

[75] Peter Stone, Patrick Riley, and Manuela M. Veloso. Defining and using ideal teammate and opponent agent models. In *Twelfth Innovative Applications of AI Conference (IAAI-2000)*, pages 1040–1045, 2000.

[76] Nathan Sturtevant and Michael Bowling. Robust game play against unknown opponents. In *Fifth International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 713–719, 2006.

[77] Nathan Sturtevant, Martin Zinkevich, and Michael Bowling. $prob-max^n$: Playing n-player games with opponent models. In *National Conference on Artificial Intelligence (AAAI)*, pages 1057–1063, Boston, MA, 2006.

[78] Gita Sukthankar and Katia Sycara. Robust recognition of physical team behaviors using spatio-temporal models. In *Fifth International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, May 2006.

[79] Flemming Topsoe. Some inequalities for information divergence and related measures of discrimination. *IEEE Transactions on Information Theory*, 46(4):1602–1609, Jul 2000.

[80] Jose M. Vidal and Edmund H. Durfee. Recursive agent modeling using limited rationality. In *First International Conference on Multi-Agent Systems*, pages 125–132, 1995.

[81] Jose M. Vidal and Edmund H. Durfee. The impact of nested agent models in an information economy. In *Second International Conference on Multi-Agent Systems*, pages 377–384, 1996.

[82] Jose M. Vidal and Edmund H. Durfee. Learning nested models in an information economy. *Journal of Experimental and Theoretical Artificial Intelligence*, 10:291–308, 1998.

[83] Martin Zinkevich. Data files from the 2007 computer poker competition. Internet (Unreviewed), July 2007. (http://www.cs.ualberta.ca/~pokert/2007/data.html).

[84] Martin Zinkevich, Michael Bowling, and Neil Burch. A new algorithm for generating equilibria in massive zero-sum games. In *Twenty-Second Conference on Artificial Intelligence (AAAI)*, pages 788–793, 2007.