A Reinforcement Learning Algorithm Applied to Simplified Two-Player Texas Hold'em Poker

Fredrik A. Dahl

Norwegian Defence Research Establishment (FFI)
P.O. Box 25, NO-2027 Kjeller, Norway
Fredrik-A.Dahl@ffi.no

Abstract. We point out that value-based reinforcement learning, such as TD- and Q-learning, is not applicable to games of imperfect information. We give a reinforcement learning algorithm for two-player poker based on gradient search in the agents' parameter spaces. The two competing agents experiment with different strategies, and simultaneously shift their probability distributions towards more successful actions. The algorithm is a special case of the lagging anchor algorithm, to appear in the journal *Machine Learning*. We test the algorithm on a simplified, yet non-trivial, version of two-player Hold'em poker, with good results.

1 Introduction

A central concept in modern artificial intelligence is that of intelligent agents, that interact in a synthetic environment. The game-theoretic structure of extensive form games is a natural mathematical framework for studying such agents. The sub-field of two-player zero-sum games, which contains games with two players that have no common interest, has the added benefit of a strong solution concept (minimax) and a corresponding well-defined performance measure.

In this article we apply a gradient-search-based reinforcement learning algorithm for a simplified Texas Hold'em poker game. The algorithm is a simplified form of the *lagging anchor algorithm*, to appear in the journal *Machine Learning* [1]. The contribution of the present paper is the presentation of an application to a more complex problem than those of the journal paper.

The structure of the paper is as follows: In Section 2 we explain a few key concepts of game theory, and give a brief survey of reinforcement learning in games. Section 3 covers earlier work on Poker games. In Section 4 we describe our simplified Hold'em Poker game, and Section 5 gives our agent design. Section 6 describes the lagging anchor algorithm in general terms, together with a precise implementation of the simplified form used in the present article. In Section 7 we give the performance measures that we use, and Section 8 describes the experiments. Section 9 concludes the article. For a more thorough treatment of the topics in Sections 2, 3, 6 and 7, we refer to the journal article.

L. De Raedt and P. Flach (Eds.): ECML 2001, LNAI 2167, pp. 85-96, 2001.
© Springer-Verlag Berlin Heidelberg 2001

2 Reinforcement Learning in Games

Game theory [2] is a complex mathematical structure, and it is beyond the scope of this article to give more than an introduction to some of its key terms. We restrict our attention to two-player zero-sum games, which means that there are two players with opposite goals, and therefore no common interest. Under mild conditions, a two-player zero-sum game has a *minimax solution*. It consists of a pair of playing strategies for both sides that is in equilibrium, in the sense that neither side can benefit from changing his strategy as long as the opponent does not. The minimax solution gives the game a numeric *value*, which is the expected payoff (for the first player), given minimax play.

An important distinction is that between games of *perfect* and *imperfect information*. In perfect information games like chess and backgammon, both players know the state of the game at all times, and there are no simultaneous moves. In a perfect information game, each game state can be regarded as the starting state of a new game, and therefore has a value. If an agent knows the value of all game states in a perfect information game, it can easily implement a perfect strategy, in the minimax sense, by choosing a game state with the highest possible value (or lowest, if the value is defined relative to the opponent) at each decision point.

With imperfect information games such as two-player Poker or Matching Pennies (see below), the picture is more confusing, because minimax play may require random actions by the players. In Matching Pennies, both players simultaneously choose either “Heads” or “Tails”. The first player wins if they make the same choice, and the second player wins otherwise. The minimax solution of this game is for both players to choose randomly with probability 0.5 (flip a coin). Under these strategies they have equal chances, and neither side can improve his chance by changing his strategy unilaterally. Obviously, there exists no deterministic minimax solution for this game. In Matching Pennies, the information imperfection is due to the fact that choices are made simultaneously, while in Poker games, it is a consequence of the private cards held by each player. Poker games typically also feature randomized (or *mixed*) minimax solutions. The randomization is best seen as a way of keeping the opponent from knowing the true state of the game. In a perfect information game, this has little point, as the opponent knows the game state at all times. Note that the concept of game state values, which is the key to solving perfect information games, does not apply to imperfect information games, because the players do not know from which game states they are choosing.

A game represents a closed world, formalized with rules that define the set of allowed actions for the players. Games are therefore suitable for algorithms that explore a problem “by themselves”, commonly referred to as reinforcement learning. This term is actually borrowed from the psychological literature, where it implies that actions that turn out to be successful are applied more often in the future. In the machine-learning context, the term is often used more broadly, covering all algorithms that experiment with strategies and modify their strategies based on feedback from the environment.

The reinforcement learning algorithms that have been studied the most are TD-learning [3] and Q-learning [4]. These algorithms were originally designed for Markov decision processes (MDPs), which may be viewed as 1-player games. TD- and Q-learning work by estimating the utility of different states (and actions) of the

process, which is the reason why they are referred to as value-based. Convergence results for value-based reinforcement learning algorithms are given in [5]. In an MDP, an accurate value function is all that it takes to implement an optimal policy, as the agent simply chooses a state with maximum value at each decision point.

The approach of deriving a policy from a state evaluator generalizes to two-player zero-sum games with perfect information, such as backgammon [6]. However, as we have seen in our brief game-theoretic survey, the value-based approach does not work with imperfect information, because the players may not know which game states they are choosing between. Also we have seen that optimal play in games of imperfect information may require random actions by the players, which is not compatible with the "greedy" policy of always choosing the game state with maximum value. It should be noted that the value-based approach can be extended to a subset of imperfect information games named *Markov games* by the use of matrix-game solution algorithms [7,8]. However, non-trivial Poker games are not Markov.

Summing up, established reinforcement learning algorithms like TD- and Q-learning work by estimating values (i.e. expected outcomes under optimal strategies) for process or game states. In (non-Markov) games of imperfect information, this paradigm does not apply.

3 Related Work on Poker

An important breakthrough in the area of solution algorithms for two-player games (not necessarily zero-sum) is that of sequential representation of strategies [9]. Prior to this work, the standard solution algorithm for two-player zero-sum games was based on enumerating all deterministic strategies for both players, assembling a corresponding game matrix, and solving the matrix game with linear programming [10]. The sequential strategy representation algorithm is an exponential order more efficient than the matrix game approach, and it has been applied to simple poker games [11]. However, even this algorithm quickly becomes intractable for non-trivial poker games.

A more practical view of computer poker is taken in the Hold'em-playing program "Loki" [12]. It uses parametric models of the habits of its opponents. Loki updates its opponent models "real time", based on the actions taken by its opponents. It estimates the utilities of different actions by approximate Bayesian analysis based on simulations with the current state of the opponent models. Apparently this approach has been quite successful, especially against weak and intermediate level humans. Note, however, that the objective of Loki is rather different from ours: We attempt to approximate game-theoretic optimal (minimax) behavior, while Loki attempts to exploit weaknesses in human play.

4 Simplified Two-Player Hold'em Poker

We now give the rules of our simplified two-player Hold'em poker game. Firstly, the full deck of 52 cards is shuffled. Then two private cards are dealt to each player (hereafter named *Blue* and *Red*). Blue then makes a forced blind bet of one unit,

whereafter Red has the options of *folding*, *calling* and *raising* (by one unit). The betting process continues until one player folds or calls, except that Blue has the right to bet if Red calls the blind bet (the blind is "live"). Also there is a limit of four raises, so the maximum pot size is 10.

As usual in poker, a player loses the pot to the opponent if he folds. If the betting stops with a call, five open cards, called the *table*, are dealt. These are common to the players, so that both have seven cards from which they can choose their best five-card poker hand. The player with the better hand wins the pot. An example game may proceed as shown in Table 1.

Table 1. An example game of simplified Hold'em poker

	Blue	**Red**
Cards	♠A ♦K	♣5 ♣4
Betting	Blind bet	Raise
	Raise	Call
Table	♣A ♣K ♣2 ♠7 ♦7	
Best hand	♣A ♠A ♣K ♦K ♠7	♣A ♣K ♣5 ♣4 ♣2

In the example game, Red wins three units from Blue, because his flush beats Blue's two pair.

The decision tree of our game is given in Figure 1. Arrows pointing to the left represent folding, downward-pointing ones represent calling, while those pointing to the right represent raising. This is not the complete game tree, however, because the branching due to the random card deal is not represented. The nodes containing a "B" or an "R" represent decision nodes for Blue and Red, respectively. The leaf nodes contain Blue's payoff, where "+/–" indicates that the cards decide the winner.

Although our game is far simpler than full-scale Hold'em, it is complex enough to be a real challenge. We have not attempted to implement the sequential strategy algorithm, but we can indicate the amount of effort this would take. Essentially, that algorithm requires one variable for each available action for every information state, to represent a player's strategy. From Figure 1 this implies 13 variables for each different hand for Blue, and 14 for each hand Red can have. By utilizing the fact that suits are irrelevant (except whether or not the two cards are of the same suit), the number of different hands is reduced to 169. This implies that the strategy space of both sides has more than 2000 degrees of freedom. The algorithm requires the assembly (and processing) of a matrix with Blue degrees of freedom as columns and Red ones as rows (or vice versa), which implies a total of $169 \cdot 13 \cdot 169 \cdot 14 = 5{,}198{,}102$ matrix entries. The calculation of these entries also requires the calculation of the win probabilities of the various opposing hands ($169 \cdot 169 = 28{,}561$ combinations). One would probably have to estimate these probabilities by sampling, because the set of possible card combinations for the table is very large. All in all, it may be possible to solve our game using a present-day computer, but it will require massive use of computer time.

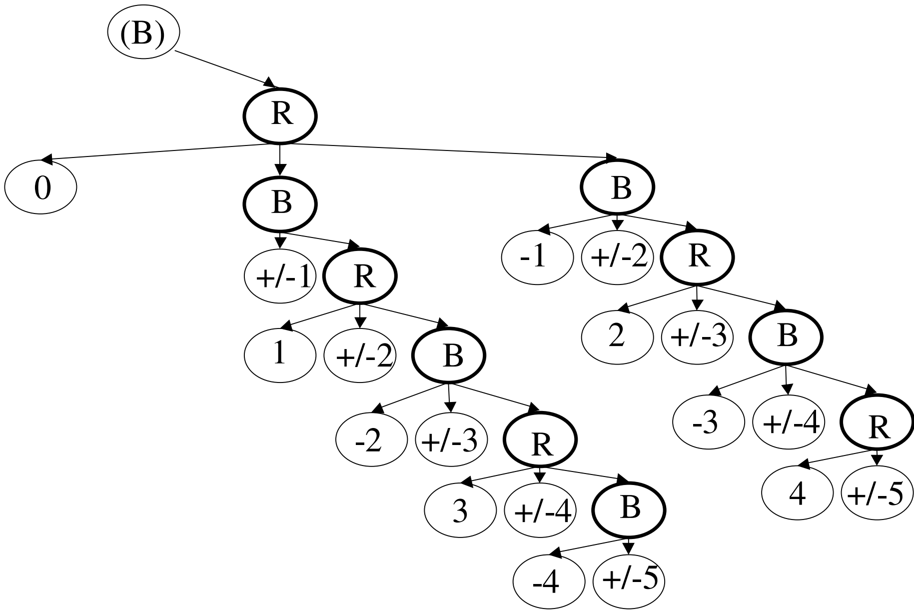

Fig. 1. The decision tree of simplified Hold'em poker. Arrows to the left signify *fold*, vertical arrows *call*, and arrows to the right *raise*

5 Agent Design

Our agents are designed to *act on the basis of available information*. This means that an agent bases its decision on its own two cards and the current decision node (in Figure 1). In game-theoretic terms this means that the agents act on *information sets* and represent behavioural strategies. From game theory, we know that strong play may require random actions by an agent, which means that it must have the capability to assign probabilities to the available actions in the given decision node. We use separate agents for playing Blue and Red.

The general agent design that we use with the lagging anchor algorithm is as follows: Let S represent the set of information states that the agent may encounter, and let $A(s)$ represent the (finite) set of available actions at state $s \in S$. For each $s \in S$ and $a \in A(s)$, the agent has a probability $P(s,a)$ of applying action a at information state s. Furthermore, we assume that the agent's behaviour is parameterised by $v \in V$: $P_v(s,a)$. We assume that V is a closed convex subset of $\mathbf{R}^n$ for some n. Summing up, our general agent design allows probability distributions over the set of legal actions for different information states, and these probability distributions may depend on a set of internal parameters of the agent (v). The goal of the learning algorithm is to find parameter values $v^* \in V$ so that the agent acts similarly to a minimax strategy.

Our agent design may give associations to Q-learning, which also works for agents that assign numeric values to combinations of states and actions. The main difference is one of interpretation; while Q-values estimate expected (discounted) rewards, our *P*-function dictates the agent's probability distribution over available actions.

For our present application, we design our agents using neural nets (NNs) that take as input the available information and a candidate action, and give a probability weight as output. When such an agent responds to a game state, it first evaluates all available actions, and then chooses a random action according to the outputs of the NN. The design is illustrated in Figure 2.

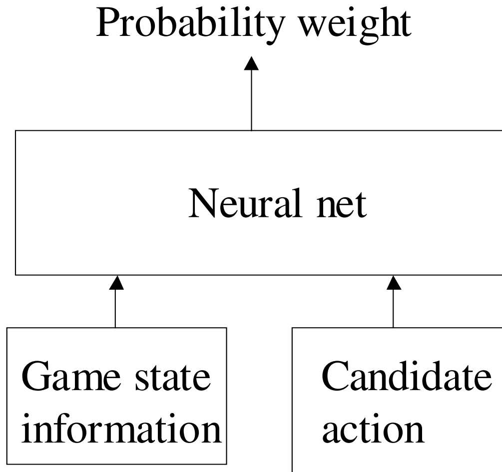

Fig. 2. Neural net agent design

For our NNs we have chosen a simple multi-layer perceptron design with one layer of hidden units and sigmoid activation functions. For updating we use standard back-propagation of errors [13]. The NN has the following input units (all binary): 13 units for representing the card denominators, one unit for signaling identical suit of the cards, one for signaling a pair, eight nodes for signaling the size of the pot, and finally three nodes signaling the candidate action (*fold*, *call* and *raise*). The single output node of the net represents the probability weight that the agent assigns to the action. The number of hidden nodes was set to 20. With this design, the internal parameters (v's) are the NN weights, which will be tuned by the learning algorithm.

We denote Blue's NN function by $B_v(s,a)$, and Red's by $R_w(s,a)$. For Blue, the corresponding probability function is $P_v(s,a) = \frac{B_v(s,a)}{\sum_{\bar{a} \in A(s)} B_v(s,\bar{a})}$, and likewise for Red.

6 Learning Algorithm

The idea of our algorithm is to let Blue and Red optimize their parameters through simultaneous gradient ascent. Let $E(v,w)$ be Blue's expected payoff when Blue's playing strategy is given by v and Red's by w. By the zero-sum assumption, Red's expected payoff is $-E(v,w)$. If we set the step size to α, the following (idealized) update rule results:

$$\begin{aligned} v^{k+1} &\leftarrow v^k + \alpha \nabla_V E(v^k, w^k) \\ w^{k+1} &\leftarrow w^k - \alpha \nabla_W E(v^k, w^k) \end{aligned} \quad (1)$$

In general, the basic gradient search algorithm (1) does not converge. In the context of matrix games (where E is bi-linear), Selten has shown that update rule (1) cannot converge towards mixed strategy solutions [14]. In [1] we show that in the case of matrix games with fully mixed randomised solutions, the paths of (v^k, w^k) converge towards circular motion around minimax solution points, when the step size falls towards zero. This fact is utilized in the lagging anchor algorithm: An anchor $\bar{v}^k$ maintains a weighted average of earlier parameter states for Blue. This “lagging anchor” pulls the present strategy state towards itself. Similarly, a lagging anchor $\bar{w}^k$ pulls Red's strategy towards a weighted average of previously used strategies, turning the oscillation of update rule (1) into spirals that, at least in some cases, converge towards a minimax solution. The (idealized) lagging anchor update rule looks like this:

$$\begin{aligned} v^{k+1} &\leftarrow v^k + \alpha \nabla_V E(v^k, w^k) + \alpha\eta(\bar{v}^k - v^k) \\ w^{k+1} &\leftarrow w^k - \alpha \nabla_W E(v^k, w^k) + \alpha\eta(\bar{w}^k - w^k) \\ \bar{v}^{k+1} &\leftarrow \bar{v}^k + \alpha\eta(v^k - \bar{v}^k) \\ \bar{w}^{k+1} &\leftarrow \bar{w}^k + \alpha\eta(w^k - \bar{w}^k) \end{aligned},$$

where η is the anchor attraction factor.

In the present article, we use an approximate variant of learning rule (1), i.e. without anchors. The learning rule includes the calculation of the gradients of the expected payoff, with respect to the agents' internal parameters. We estimate these gradients through analysis of sample games. First the Blue and Red agents play a sample game to its conclusion. Then both Blue and Red perform the following “what-if” analysis: At each decision node (as in Figure 1) visited, an additional game is completed (by Blue and Red) for each decision not made in the original game. The outcomes of these hypothetical games provide estimates of how successful alternative decisions would have been. The agents then modify their NNs in order to reinforce those actions that would have been the most successful. We accomplish this through the use of training patterns of input and desired output: (gamestate+action , feedback). If a given (hypothetical) action turned out more successful than the others, for the given game state, the agent should apply it *more often*. This means that the training pattern feedback should be given by the NN's current evaluation of the state-action pair offset by the action's relative success compared to the other actions. Because of this relative nature of the feedback signals, there is a risk that the NN outputs may drift toward zero or one, which hurts the back-propagation learning. We prefer that

the NN outputs approximate probability distributions, and therefore adjust the feedback signals in the NN training patterns accordingly.

In pseudo-code, the algorithm is given below, where we apply the convention of displaying vector quantities in **boldface**. Keywords are displayed in **`boldface courier`**. Blue's and Red's NN functions are denoted by $B(\cdot;)$ and $R(\cdot;)$, respectively.

`repeat` *Iteration* **`times`** {

$\langle$play a game g between Blue and Red$\rangle$

`for` $\langle$each decision node $n \in g\rangle$ **`do`** {

$\mathbf{A} \leftarrow \langle$legal actions at $n\rangle$

$\mathbf{E} \leftarrow \langle$outcomes of games resulting from actions $\mathbf{A}$ at $n\rangle$

`if` $\langle$Blue on turn in $n\rangle$ { $\mathbf{P} \leftarrow B(s, \mathbf{A})$ }

`else` { $\mathbf{P} \leftarrow R(s, \mathbf{A})$ }

$p_{sum} \leftarrow \mathbf{1}^T \mathbf{P}$

$e \leftarrow \mathbf{P}^T \mathbf{E} / p_{sum}$

$\mathbf{E} \leftarrow \mathbf{E} - \mathbf{1}e$

$\mathbf{F} \leftarrow \mathbf{P} + \mathbf{E} - \mathbf{1}(p_{sum} - 1)$

`if` $\langle$Blue on turn in $n\rangle$ { $\langle$train B with patterns $\{(s, \mathbf{A}), \mathbf{F}\}\rangle$ }

`else` { $\langle$train R with patterns $\{(s, \mathbf{A}), \mathbf{F}\}\rangle$ }

}

}

Operations involving vectors are interpreted component-wise, so the notation implies several for-loops. As an example, the statement $\langle$train B with patterns $\{(s, \mathbf{A}), \mathbf{F}\}\rangle$ is implemented as:

`for` $(i = 1 \ldots length(\mathbf{A}))$ **`do`** { $\langle$train B with pattern $((s, A_i), F_i)\rangle$ }.

The vector **E** consists of outcomes (for the player on turn) of sample games that explore the different actions **A** in node *n*. In these games, the players' hands and the table cards are held fixed. Note that when we assemble **E**, we take the outcome of the actual game as the estimated outcome from taking the action chosen in that game. The number e estimates the expected payoff for the player on turn, given his current probability distribution over the actions **A**. The statement $\mathbf{E} \leftarrow \mathbf{E} - \mathbf{1}e$ normalizes **E** by deducting e from each component. **F** is the calculated vector of feedback, and the term $-\mathbf{1}(p_{sum} - 1)$ is included in order to push the NN function (*B* or *R*) towards valid probability distributions.

7 Evaluation Criteria

Defining evaluation criteria for two-player zero-sum games is less straightforward than one might believe, because agents tend to beat each other in circle. Ideally, we would like to apply the performance measure of *equity against globally optimizing opponent* (*Geq*) as described in [15]. The *Geq* measure is defined as the expected payoff when the agent plays against its most effective opponent (the *best response strategy* in game-theoretic terms). The *Geq* measure conforms with game theory in the sense that an agent applies a minimax strategy if, and only if, its *Geq* is equal to the game's value (which is the maximum *Geq* achievable).

Although we develop our Blue and Red players as separate agents that compete against each other, it is convenient for the purpose of evaluation to merge them into one agent that can play both sides. For agents of this form, a single game is implemented as a pair of games, so that both agents get to play both sides. For the sake of variance reduction, we hold the cards fixed in both games, so that both agents get to play the same deal from both sides. We take the average of the two outcomes as the merged game's outcome. The redefinition of the game as a pair of games has the advantage that the value is known to be zero, by symmetry.

We use a set of three reference players, named *Balanced-player*, *Aggressive-player* and *Random-player*. Balanced-player is our best estimate of a minimax-playing agent. Our first implementation of this agent turned out to have significant weaknesses, and the final one was developed through experimenting with (in parts even imitating) our NN agents. Aggressive-player is a modification of Balanced-player that folds only a few hands and raises often. Random-player makes completely random actions, with uniform probabilities over actions. It is included mostly for reference, as it is unlikely that it can ever be the most effective opponent.

8 Experimental Results

The step size for the NN back-propagation update started at 0.5 at the beginning of the training session, and was tuned down to 0.1 after 50,000 training games. The NNs were initialized with random weights. Figure 3 shows the estimated performance against the three reference opponents as a function of the number of training games.

We observe that the agent initially scores approximately 0 against Random-player, which is reasonable. We also see that Aggressive-player is the most effective opponent by a large margin at this point. The reason for this is that a randomly playing agent will sometimes fold after a sequence of raises, which is extremely costly. Against Balanced-player, the agent does not get the chance to make this error so often. Recall that our agent learns by adapting to its opponent (its own other half in the evaluation procedure). It therefore first learns strategies that are effective against a random opponent, which means that it begins to resemble Aggressive-player. This explains why it quickly scores so well against Random-player. Once the agent has learned not to fold so often, it starts to appreciate the value of good cards, and stops raising with weak hands. From then on, its strategy moves towards that of Balanced-player. The figure shows that when the agent becomes sufficiently skilled, it starts beating Aggressive-player, and Balanced-player takes over as the most effective

opponent. The fluctuations in the diagrammed performance graphs are mainly due to randomness in the procedure of sampling the performances. Note that the random noise in the sampling of the performance against Balanced-player falls towards zero. This is because their strategies become similar, which makes the variance reduction trick of playing the same cards both ways more effective.

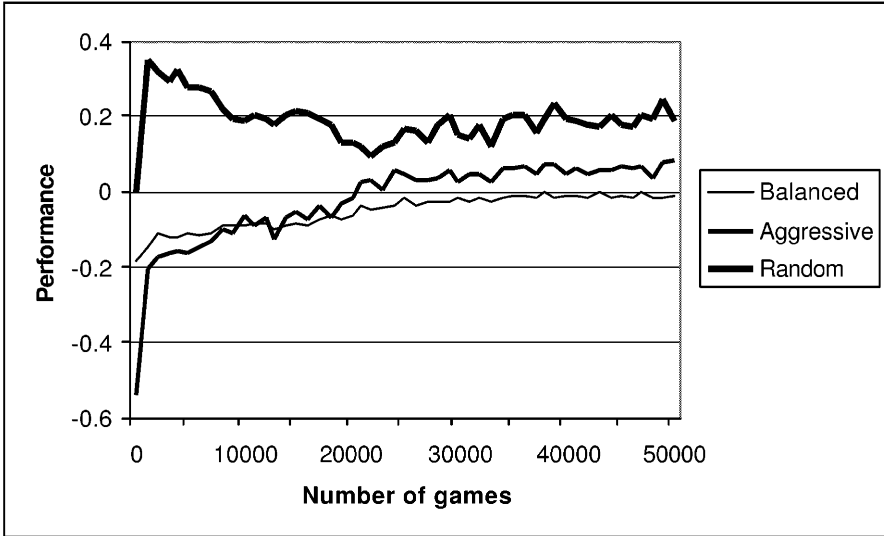

Fig. 3. Performance against reference opponents

The procedure of defining a set of opponents, and taking the result against the most effective of these, is a practical approach to estimating the *Geq* of an agent. According to this test, our NN based player appears to approach minimax play. Unfortunately, our small set of opponents is not sufficient to convince us. However, we are able to estimate the *Geq* quite accurately, through optimization. In this calculation we analyze one opponent hand at the time, and experimentally determine the most effective opponent strategy. For each of the 169 different hands, we have completed 10,000 test games for each deterministic playing strategy (derived from the decision tree of Figure 1). These calculations are rather time consuming, so we have not been able to produce learning curves with respect to this measure, but only analyzed the NN agent resulting from the complete training session. The learning curves of Figure 3 have actually been truncated, in order to highlight the interesting phenomena close to the start of the session. After 200,000 training games, our agent broke exactly even (to three decimal places) against Balanced-player. The massive optimization calculation gave a *Geq* estimate of –0.005 for this NN agent, which gives strong evidence that it is in fact close to minimax play.

Our fully trained agent has discovered a rather non-trivial fact that we hold to be true (or close to true) also for full-scale Hold'em: As Red it never calls the blind bet, but either folds or raises. Calling the blind bet is often a bad idea, because it leaves the opponent with the option of raising without putting on any pressure. If Red believes that he can make a profit by playing a hand (folding gives payoff 0), he should probably raise the stakes. Some humans like to call the blind bet with strong hands,

with the intention of re-raising if Blue is tempted to raise. We do not think this is a sound strategy, because Red would also have to call with some weak or intermediate hands in order not to reveal his hand when he calls. We believe that the downside of playing these hands outweighs the benefit of sometimes getting to re-raise with the strong hands.

An open question that remains is why the algorithm works so well without the anchors. We know from formal analysis that the gradient ascent algorithm fails for matrix games with mixed strategy solutions, and the non-linearity of our Poker game is not likely to do any good. In our opinion, the reason is that there exist minimax strategies that are only marginally random. Every Poker player knows the importance of being unpredictable, so it may sound odd that good play requires little randomization. The explanation is that the random card deal does the randomization for the player. Although the player's betting is a deterministic function of his private cards, the randomness of the cards is sufficient to keep the opponent uncertain about the true state of the game. There probably exist borderline hands (e.g. hands on the border between an initial pass and raise for Red) that would be treated randomly by an exact minimax solution, but given the large number of possible hands, these are not very important.

9 Conclusion

We have implemented a reinforcement learning algorithm for neural net-based agents playing a simplified, yet non-trivial version of Hold'em poker. The experiments have been successful, as the agents appear to approximate minimax play. The algorithm is a special case of one that is to appear in the journal *Machine Learning*.

References

1. Dahl, F.A.: The lagging anchor algorithm. Reinforcement learning in two-player zero-sum games with imperfect information. *Machine Learning* (to appear).
2. Owen, G.: *Game Theory. 3rd ed.* Academic Press, San Diego (1995).
3. Sutton, R.S.: Learning to predict by the methods of temporal differences. *Machine Learning* 3 (1988) 9–44.
4. Watkins, C.J.C.H.: *Learning from Delayed Rewards.* PhD thesis, University of Cambridge, UK (1989).
5. Szepesvari, C., Littman, M.L.: A unified analysis of value-function-based reinforcement-learning algorithms. *Neural Computation* 11 (1999) 2017–2060.
6. Tesauro, G.J.: Practical issues in temporal difference learning. *Machine Learning* 8 (1992) 257–277.
7. Littman, M.L.: Markov games as a framework for multi-agent reinforcement learning. In: *Proceedings of the 11th International Conference on Machine Learning*, Morgan Kaufmann, New Brunswick (1994) 157–163.
8. Dahl F.A., Halck O.M.: Minimax TD-learning with neural nets in a Markov game. In: Lopez de Mantaras, R., Plaza, E. (eds.): *ECML 2000. Proceedings of the 11th European Conference on Machine Learning*. Lecture Notes in Computer Science Vol. 1810, Springer-Verlag, Berlin–Heidelberg–New York (2000) 117–128.

9. Koller, D., Megiddo, N., von Stengel, B.: Efficient computation of equilibria for extensive two-person games. *Games and Economic Behavior* 14 (1996) 247–259.
10. Luce, R.D., Raiffa, H.: *Games and Decisions*. Wiley, New York (1957).
11. Koller, D., Pfeffer, A.: Representations and solutions for game-theoretic problems. *Artificial Intelligence* 94 (1997) 167–215.
12. Schaeffer, J., Billings, D., Peña, L., Szafron, D.: Learning to play strong poker. In: Fürnkranz, J., Kubat, M. (eds.): *Proceedings of the ICML-99 Workshop on Machine Learning in Game Playing*, Jozef Stefan Institute, Ljubljana (1999).
13. Hassoun, M.H.: *Fundamentals of Artificial Neural Networks*. MIT Press, Cambridge, Massachusetts (1995).
14. Selten R. (1991). Anticipatory learning in two-person games, in: Selten, R. (ed.): *Game equilibrium models, vol. I: Evolution and game dynamics*, Springer-Verlag, Berlin.
15. Halck, O.M., Dahl, F.A.: On classification of games and evaluation of players – with some sweeping generalizations about the literature. In: Fürnkranz, J., Kubat, M. (eds.): *Proceedings of the ICML-99 Workshop on Machine Learning in Game Playing*, Jozef Stefan Institute, Ljubljana (1999).