Machine Learning, 49, 5–37, 2002
© 2002 Kluwer Academic Publishers. Manufactured in The Netherlands.

The Lagging Anchor Algorithm: Reinforcement Learning in Two-Player Zero-Sum Games with Imperfect Information

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

Editor: Sridhar Mahadevan

Abstract. The article describes a gradient search based reinforcement learning algorithm for two-player zero-sum games with imperfect information. Simple gradient search may result in oscillation around solution points, a problem similar to the "Crawford puzzle". To dampen oscillations, the algorithm uses lagging anchors, drawing the strategy state of the players toward a weighted average of earlier strategy states. The algorithm is applicable to games represented in extensive form. We develop methods for sampling the parameter gradient of a player's performance against an opponent, using temporal-difference learning. The algorithm is used successfully for a simplified poker game with infinite sets of pure strategies, and for the air combat game Campaign, using neural nets. We prove exponential convergence of the algorithm for a subset of matrix games.

Keywords: two-player zero-sum game, reinforcement learning, neural net, imperfect information

1. Introduction

The purpose of the work documented in this article is to produce algorithms for creating approximate solutions to complex two-player zero-sum games with imperfect information. Loosely speaking, in two-player zero-sum games the players have no common interest, and their information about the current game state may differ. The class contains games with perfect information like chess, go and backgammon, as well as imperfect information games like two-player poker, scissors-paper-rock and battleship. Games with imperfect information are *qualitatively different* from perfect information ones. With imperfect information optimal play may require random behavior of the agent, which is not necessary in perfect information games.

There are good reasons why we are interested in games with imperfect information. In real-world situations, perfect information is likely to be the exception rather than the rule. An example is the area of military combat, where information is bound to be imperfect, and where the two parties will normally have little common interest, making two-player zero-sum games the natural modelling paradigm, see e.g. Berkovitz (1975).

A different rationale for studying two-player zero-sum games with imperfect information comes from the theory of computational complexity (Papadimitriou, 1994). Assume that a programmer is designing an algorithm for a problem, such as the travelling salesman problem with n cities. Also assume that he wants to optimize the worst-case long-term performance of his algorithm, without any prior knowledge of the distribution of the problem instances that the algorithm will face. This is equivalent to the programmer playing a game against Nature, where he chooses an algorithm and Nature chooses a problem instance simultaneously, and the payoff is the negation of the time spent by the algorithm. This is a two-player zero-sum game with imperfect information that typically features randomized solutions.

There exist successful algorithms for reinforcement learning in perfect information games, such as TD-learning (Tesauro, 1992). Lookahead search of the game trees works well as an add-on to this, producing a powerful combination of machine learning and more classical artificial intelligence techniques. These methods all work by estimating state values (the expected outcomes of game states under optimal play). In games like two-player poker this approach fails. We claim that no general reinforcement learning algorithm for two-player zero-sum games with imperfect information is known. The present paper is an attempt to change this.

The article is structured as follows. In Section 2 related work, including some elementary game theory, is surveyed. Section 3 gives definitions of evaluation criteria for players and player representation, thereby formalizing our problem of constructing strong players. In Section 4 we define the lagging anchor algorithm and give some theoretical results. Section 5 gives an application of the algorithm to a simplified poker game with infinite sets of pure strategies. Section 6 gives an application to a non-trivial air combat game, and Section 7 concludes the article. The Appendix gives proofs of the results stated in Section 4.

2. Related work

To set the scene for our survey of related work, we must first establish some elementary game theory. We then proceed to discuss related work within the economics and machine learning communities.

2.1. Game theory

In game theory, see e.g. Luce and Raiffa (1957), a *game* is a decision problem with two or more *players*, where the outcome for each player may depend on the decisions made by all players. We will now present the *extensive form*, which is a compact and intuitive representation scheme for games. We only consider finite games.

The initial state of a game is the root of a tree, called the *game tree*. The nodes represent *game states*, and the directed arcs connecting them represent possible state transitions, each due to an action taken by a player or a chance event. A path from the root to a leaf node represents the course of a game. A leaf node therefore represents a terminal state, and gives a real-valued *payoff* to each player. All players are assumed to attempt to maximize their expected payoff. A non-terminal node is either a *decision node* or a *chance node*. In a decision node a player is on turn, and he is free to choose any one of the node's arcs, which leads to the succeeding game state. In a chance node there is a probability distribution over the arcs, and the succeeding state is drawn randomly according to this distribution. A chance node represents a random event such as the rolling of dice, or drawing of cards, and the players cannot affect or predict the outcome of it.

A main point for us is to study games with *imperfect information*, which means that a player may not know the exact state of the game. This is represented by grouping the set of nodes for each player into *information sets*. The node of the initial state must be an information set by itself. A player knows which information set he is in, but is unable to distinguish between nodes of the same set. If we see the game from the perspective of a player, an information set corresponds to a state of the game. This implies that a player must treat different nodes of an information set identically. We assume that the information sets are minimal, in the sense that nodes of the same set must result from identical sequences of decisions by the player, called *perfect recall*. We will sometimes use the term "information state" instead of "information set", because it corresponds better with our intuitive semantics.

A *pure strategy* for a player is a deterministic plan dictating his action in every information set, and is similar to a *policy* in a Markov decision process (MDP). A *mixed*, or *randomized*, strategy is a weighted collection of pure strategies, where the weights are interpreted as probabilities. When an agent applies a mixed strategy, it first draws one of the pure strategies randomly according to its probability, and then plays the game according to it. If the game be repeated, the agent draws a pure strategy independently for each game instance. This concept of mixing pure strategies is practical from the perspective of proving theorems, but *behavioral representation* of mixed strategies is more compact, and is often more useful in practice. In a behavioral strategy the agent's randomization takes place inside the information sets, rather than outside the game tree. A mixture of pure strategies induces a behavioral strategy, and a behavioral strategy can be decomposed to a mixture of pure strategies.

In a *two-player zero-sum game*, the payoff structures of the two players are such that they have no common interest, meaning that one side wins what the other side loses. From here on we name the players 'Blue' and 'Red'. We see the outcome from Blue's point of view, so that Blue wants to maximize the expected outcome and Red wants to minimize it. Any finite two-player zero-sum game has a *value*, meaning that each player has a strategy (possibly mixed) that guarantees the expected outcome to be no worse than the value (higher for Blue, lower for Red). Such a pair of strategies, referred to as a *minimax solution*, constitutes an equilibrium point of the game, as no player can do any better than the value by deviating. Note that the class of two-player constant-sum games, where the outcomes for Blue and Red add to a constant, can trivially be mapped to the class of two-person zero-sum games by subtracting half of this constant from the outcomes.

Although the decisions are assumed to be sequential, the extensive form can also represent simultaneous decisions with the use of information sets. We illustrate this with the simple game of "matching pennies", in which the players simultaneously call "Heads" or "Tails". Blue wins one unit from Red if they make the same choice, and loses one otherwise. We arbitrarily choose to let Blue make his choice first, so the tree branches into two nodes; see figure 1. In each of these, Red is on turn, and the tree branches further into four terminal nodes. With simultaneous decisions, Red is unable to base his choice on Blue's decision, and this we represent by placing Red's decision nodes in the same information set, indicated with the dotted ellipse in the figure. The minimax solution of this game demands that both

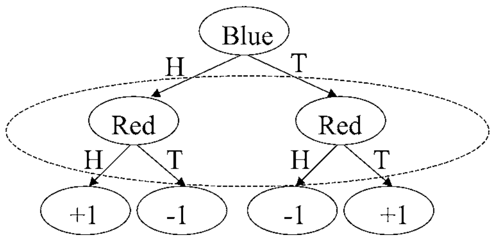

Figure 1. Game tree for matching pennies.

players assign a 0.5 probability to each choice, guaranteeing both an expected payoff of zero, which is the game's value.

The set of *Markov games* is a subclass of two-person zero-sum games (Littman, 1994). These consist of sequences of simultaneous actions by both players. The game state is shown to both players before each action, and the (game state, Blue action, Red action)-triple determines a probability distribution over succeeding game states. In a Markov game, any game state can be viewed as the initial state of an independent game, and therefore has a value.

The set of two-player zero-sum games with *perfect information* is a subclass of the set of Markov games. In these games, decisions are made in sequence, and both players know the game state at all times. A perfect information game can be viewed as a Markov game, in which the options of the player not on turn are collapsed to a singleton. Being Markov games, they inherit the property that game states specify independent games, and have values associated with them. With perfect information, each decision node constitutes an information set. These two facts imply that minimax play is characterized by the player always choosing a game state with maximum (minimum for Red) value. Therefore no randomization is required for minimax play in perfect information games. Intuitively this is reasonable because randomization is a measure taken to keep the opponent less informed about the state of the game, which cannot be accomplished with perfect information.

To make the connection with decision problems that are more common within the machine learning community, we observe that the set of MDPs is the subclass of perfect information games where Red has only one legal action. The game classes we consider are diagrammed in figure 2.

Recall that the game tree representation that we have presented is referred to as the extensive form of the game. A different representation form for two-player zero-sum games that has been studied is that of *matrix games*. In a matrix game both players have a finite set of actions to choose from, and make their decisions simultaneously. There is an instant payoff to Blue (and its negation to Red). The payoffs can be assembled in a matrix $\mathbf{M} \in \mathbf{R}^{n \times m}$, with m_{ij} representing Blue's payoff if he uses strategy i, and Red uses strategy j. Any extensive-form two-player zero-sum game with finite game tree has a finite number of pure strategies, and can be represented as a matrix game. To perform this transformation one enumerates the pure strategies, and takes m_{ij} to be Blue's expected payoff if he uses his strategy i and Red uses his strategy j.

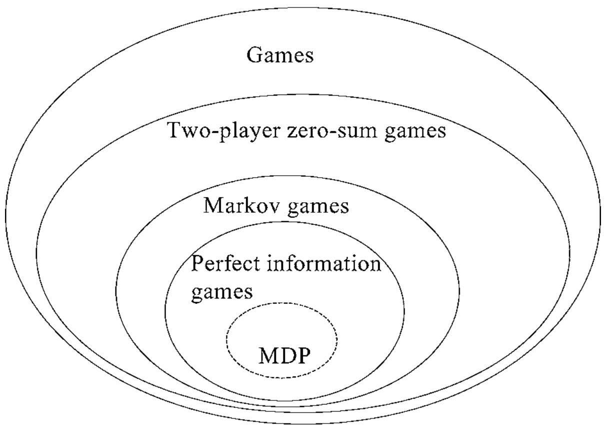

Figure 2. Classes of games.

It is a well-known fact that matrix games can be solved efficiently using linear programming (LP), see e.g. Strang (1980). However, the number of pure strategies, and therefore the size of the game matrix, grows exponentially with respect to the number of nodes in the game tree. Therefore, LP on matrix games offers no efficient solution algorithm for games in extensive form.

Relatively recently a new algorithm for solving two-player games (not necessarily zero-sum) in extensive form has been developed (Koller, Megiddo, & Stengel, 1996). It works by associating variables with paths in the tree, called sequential representation of strategies. This algorithm represents a major breakthrough, in that it is polynomial in space and time with respect to the number of nodes in the tree. Still, the number of nodes is exponential as a function of the depth of the tree, so it offers little help with large games like hold'em poker.

2.2. *Human learning in games*

The elementary game theory that we have presented is focused on equilibrium. Economists constitute the driving force in the development of game theory, and their purpose is to use it for explaining human behavior. From experiments, it has been observed that humans sometimes fail to behave according to the equilibrium theory (Erev & Roth, 1998), particularly in games with mixed strategy equilibria. Therefore modeling of human learning in games has become a hot topic. The machine learning algorithms we will present in this article could be considered as candidate models of human learning, and vice versa.

The basic approaches to modeling human learning in games can be categorized as *best response dynamics* (Fudenberg & Levine, 1998), *evolutionary game theory* (Weibull, 1995) and *psychological reinforcement learning* (Erev & Roth, 1998). All of these work in the setting where the players go through a sequence of instances of the same game, and modify their behavior.

The algorithm that we develop in the present article relies on simultaneous gradient search performed by the two players. Although gradient search is outside the mainstream modeling of human learning, this approach has been studied by Selten (1991) and Conlisk (1993a, 1993b). Selten performs his analysis in the setting of bi-matrix games, which is the non-zero sum generalization of matrix games. Because we restrict our interest to zero-sum games in the present article, we present his work in that context, but keep in mind that the results are valid in the more general case.

2.2.1. Simple gradient search. Let $\mathbf{M} \in \mathbf{R}^{n \times m}$ be the game matrix, and let the vectors $\mathbf{v} \in \mathbf{R}^n$ and $\mathbf{w} \in \mathbf{R}^m$ represent Blue's and Red's probability distributions over pure strategies. The vectors $\mathbf{v}$ and $\mathbf{w}$ must have nonnegative components summing to 1. Blue's payoff is given by the matrix-vector expression $\mathbf{v}^T\mathbf{M}\mathbf{w}$. The gradient of this payoff function with respect to $\mathbf{v}$ is $\mathbf{M}\mathbf{w}$. If the step size of Blue is α, the gradient search update to Blue's strategy would be $\mathbf{v} \leftarrow \mathbf{v} + \alpha\mathbf{M}\mathbf{w}$. The similar update rule for Red would be $\mathbf{w} \leftarrow \mathbf{w} - \alpha\mathbf{M}^T\mathbf{v}$, because he seeks to minimize Blue's payoff. However, these update rules do not necessarily maintain valid probability distributions, so some modifications are needed. Selten derives an update rule that is "as linear as possible".

Selten shows that a strategy pair $(\mathbf{v}, \mathbf{w})$ is a fixed point for the update rule if, and only if, it is a solution point. He also shows that regular pure-strategy solution points are points of attraction for the update rule. Therefore, the gradient search procedure may be a useful algorithm in cases where a pure strategy solution exists. Selten also shows a strong converse of this: If a solution point $(\mathbf{v}, \mathbf{w})$ features mixed strategies, it is not asymptotically stable. This implies that gradient search by itself is not a reliable algorithm in general.

In order to provide some intuition for this, we will illustrate how gradient search fails for matching pennies. Recall that the solution is for Blue and Red to make each choice with probability 0.5. The payoff matrix looks like this: $M = \left[\begin{smallmatrix} 1 & -1 \\ -1 & 1 \end{smallmatrix}\right]$. We represent Blue's strategy by his probability of calling "heads", denoted by v. Red's probability of calling "heads" is w. (Their probabilities of calling "Tails" are given implicitly.) When the step size α falls toward zero, the path of the process approaches a circle around the solution point (1/2, 1/2) in the parameter set $V \times W$, as is shown in figure 3.

The dynamics of the process can easily be explained. Assume that the process starts on top of the circle in figure 3. At this point Blue's strategy state v is 0.5, and Red's state w is greater than 0.5. At this point Red has no incentive to change his play, as his expected outcome is zero regardless of his choice. Red, however, plays "Heads" more often than "Tails", so Blue moves toward playing "Heads" more often too, as he wins when they make the same choice. As Blue gradually plays "Heads" more often, Red will gain by playing "Tails" more often, therefore reducing w. When w is reduced below 0.5, it means that Red plays "Tails" too often, so Blue follows suit and starts to reduce v. When v becomes less than 0.5, Red responds by increasing w, and so on.

In Section 4.1, we will show that this phenomenon of oscillation with constant distance to an interior-point mixed strategy solution is general for gradient search in (zero sum) matrix games. Crawford (1974) has shown a similar result for an update rule where the gradient is modified by the state of the player's own strategy vector.

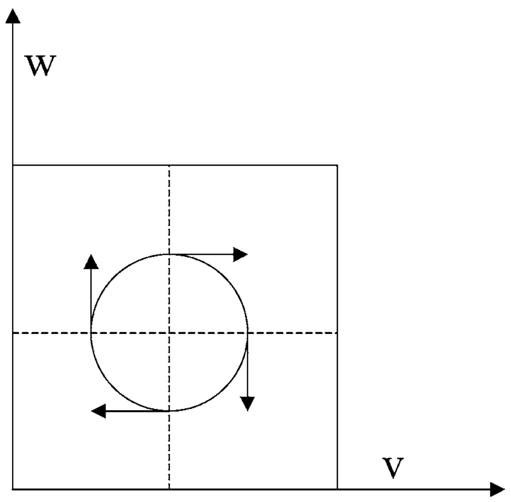

Figure 3. Path followed by gradient search for the game of matching pennies.

2.2.2. Anticipatory learning. Several modifications to simple gradient search have been suggested, in order to produce models that predict convergence toward mixed-strategy solutions (Conlisk 1993a, 1993b). One of these is Selten's "anticipatory learning" model, which is somewhat similar to the algorithm described in the present paper.

Selten's anticipatory learning model works as follows: Blue and Red simultaneously calculate the next gradient search update of the opponent. Under this anticipated strategy for the opponent, they perform a gradient search update to their own strategies. Selten shows that all regular points are attractors under the anticipatory learning update rule, for a sufficiently small α.

2.3. Machine learning in games

We are mainly interested in machine learning techniques that experiment with the games, and learn from the game outcomes, commonly referred to as reinforcement learning. However, if one has access to human game-playing expertise, one can also use machine learning to imitate the evaluations or behavior of the expert. One example of this is Tesauro and Sejnowski (1989) that trained a neural net to imitate human expert play in the stochastic two-player perfect-information zero-sum game of backgammon. We find this less interesting than reinforcement learning techniques, because it presupposes human expertise. Besides, the fact that the environment is a game does not affect the training procedure, making imitation learning in games little different from other learning tasks.

The first well-known instance of reinforcement learning applied to a game is Samuel's checkers program (Samuel, 1959). His program worked by tuning linear combinations of handcrafted position features, based on the outcomes of games played by the program. Sutton (1988) later defined temporal difference learning (TD-learning) for MDPs. This algorithm is related to dynamic programming, and works toward consistent evaluations of expected outcomes for states. By consistency, we mean Bellman equations expressing the fact that the value of a state is equal to the maximum expected value of future states. Tesauro (1992) extended the use of TD-learning to backgammon. This marked one of the greatest successes in the history of machine learning, as the trained neural net functions reached an expert level, and reached the level of the strongest human players when combined with lookahead search. Note that lookahead search strategies like alpha-beta and expectimax (Michie, 1966) also work toward consistent evaluation. While TD-learning works by minimizing the average inconsistency over all states visited, lookahead search isolates a subset of the game tree and builds a consistent evaluation of its nodes, treating estimated values of its leaf nodes as input. TD-learning assumes that the agent has knowledge of the game rules, called *complete information* in game theory, as moves are chosen that maximize the evaluation of future game states. In Q-learning (Watkins, 1989), the process of estimating values of states is extended to games and processes with incomplete information. This is done by associating values with (state, action)-pairs. Thereby the algorithm is able to estimate values of states implicitly, without knowing which states may actually result from taking a given action.

The minimax Q-learning algorithm (Littman, 1994) extends Q-learning to the class of Markov games, thereby providing an algorithm for some games with imperfect information. Minimax Q-learning works by estimating values associated with (state, Blue action, Red action)-triples. When the agent makes a move based on this evaluation function, it does the following: For the given game state, it runs through all combinations of legal actions for both sides, and assembles the corresponding evaluations in a game matrix, which is local to the game state. Then LP, or some other matrix game algorithm, is used to calculate the minimax distribution, from which a random action is drawn. If the agent has complete information, one can modify the minimax Q-learning by estimating the values of game states, and use the knowledge of the rules to identify the states resulting from a (state, Blue action, Red action)-triple. Game matrices are then assembled using the evaluation of these states. This algorithm is called minimax TD-learning (Dahl & Halck, 2000).

Reinforcement learning has also been applied to *differential games* (Harmon, Baird, & Klopf, 1995), which are the continuous time equivalents of Markov games. Differential games fall outside our classification in figure 2 because they are not finite. However, they share the property that states have values, and the reinforcement learning methods used for them build on principles from dynamic programming.

All these reinforcement learning algorithms work by estimating the value of game states. Szepesvari and Littman (1999) give some general convergence results for value-based reinforcement learning. TD-learning was first developed for the innermost ellipse of MDPs in figure 2, and extended outwards to Markov games. However, this is as far as it goes. In non-Markov two-player zero-sum games we see no way of calculating optimal actions from state values, because the players do not know from which states they are choosing. In fact, the values may not even be uniquely determined: If there is more than one minimax solution, different solutions may induce different sets of values, both for single nodes and information sets. We conclude that the reinforcement learning paradigm derived from dynamic programming, which works so well for perfect information games like backgammon, is not applicable to non-Markov imperfect information games like two-player poker.

One should mention the Loki program (Schaeffer et al., 1999), designed to play limit hold'em poker (not necessarily two-player). It works by adapting to the habits displayed by its opponent, which makes it related to reinforcement learning systems. However, it is not very relevant to us, as Loki's goal is to exploit weaknesses in human play, which is different from our goal of approximating minimax solutions.

3. Player representation and evaluation

We now explain how we represent an agent playing an extensive-form game. As usual we take Blue's perspective. Let S be the set of information states (information sets) for Blue. In a given information state $s \in S$, where Blue is on turn, the game rules give a set $A(s)$ of legal actions. From our survey of game theory, it follows that our agent must have the ability to act randomly, with different probability distributions over actions for different information states. We represent this as a function B, taking an information state $s \in S$ and an action $a \in A(s)$ as input, and giving as output the agent's probability of choosing action a at s. To represent probability distributions, B must satisfy $B(s, a) \geq 0$ for all (s, a) with $s \in S$, $a \in A(s)$, and $\sum_{a \in A(s)} B(s, a) = 1$ for all $s \in S$. In practice, the last condition may not be precisely satisfied for all information states, in which case we interpret the B-values as probability weights, and normalize them by dividing by their sum.

When our agent responds to an information set, it first evaluates each possible action in turn (each together with the present information state) using the function B. Then a random action is drawn according to the probability distribution given by these calculated probability weights.

Now assume that the mapping B also depends on a vector $v \in V$ of parameters, so that B_v is a player representation for each $v \in V$. We assume that V is a closed convex subset of $\mathbf{R}^k$ for some k, and that $B_v(s, a)$ is continuously differentiable with respect to v for all (s, a), $s \in S, a \in A(s)$. The parameter vector v represents the *state of the agent*, not to be confused with game states and information states. We can modify the probabilistic behavior of the agents by changing v. This is not normally done during a game, but our purpose is to develop algorithms for modifying v between games.

To quantify the merits of our game-playing agents, we need to define a performance measure. We will use the performance measure of *expected outcome against the opponent's best response strategy*. It assumes an opponent that optimizes his actions in all decision nodes to fit the weaknesses of the agent. Therefore we denote the performance measure by *equity against globally optimizing opponent*, or simply *Geq*.

For a given pair (B, R) of a Blue and a Red player, the course of the game is a Markov process, which generates a probability distribution over the terminal nodes of the game tree. Let $E(B, R)$ be the expected value of the terminal node payoff. Let D be the set of pure strategies available for Red. Then our evaluation of B is given by $Geq(B) = \min_{R \in D}\{E(B, R)\}$. If B plays a minimax solution of the game, $Geq(B)$ will be identical to the value of the game. Indeed this is the very definition of the game's value. Our task is to search for optimal parameter states with respect to *Geq* for the agent:

$$\text{find } v^* \in V \text{ so that } Geq(B_{v^*}) = \max_{v \in V} Geq(B_v).$$

If the player representation is *complete*, in the sense that all probability distributions can be realized by varying the parameters, the optimal value of *Geq* is the value of the game. Otherwise the optimal performance may be lower.

The *Geq* measure can be criticized for punishing errors unreasonably hard, and it tends to give surprisingly low performance evaluations. As an example, if we used this measure to evaluate a deterministic chess program, the performance would be zero as long as there is even a single way of beating it. This is so even if the opponent would have to make weak moves to trick the program into unfamiliar positions. One could also consider other performance measures, such as expected outcome against some minimax playing opponent, or similarity with a minimax solution, but our *Geq* measure has the advantage of being linked more closely to game theory. Evaluation criteria for two-player zero-sum games are discussed in Halck and Dahl (1999).

Our design of an agent playing Red is parallel to that of Blue, using a function $R_w(s, a)$ to represent the likelihood of the agent taking action a at information state s, parameterized by $w \in W$. Likewise, the definition of *Geq* also applies to Red with opposite sign.

4. The lagging anchor algorithm

We have seen in Section 2.2 that simple gradient search (which we denote *the basic algorithm*) may result in oscillations around solution points, even for simple matrix games. We wish to exploit this property in defining a better algorithm. Our algorithm exhibits exponential convergence, at least for some special cases of matrix games. The idea is to have an "anchor" for each player lagging behind the current value of the parameter states v and w, thereby dampening the oscillation seen with the basic algorithm. Each anchor is drawn toward its corresponding parameter value proportionally to the distance between the parameter state and its anchor. Each parameter state is updated according to the gradient as in the basic algorithm, but is also drawn toward its anchor. Before we embark on this, we need to define the basic algorithm formally.

Let $E(v, w)$ be the expected payoff, given that Blue uses strategy $B_v(\cdot, \cdot)$ and Red uses $R_w(\cdot, \cdot)$. By the zero sum assumption, Red's expected outcome is $-E(v, w)$. We observe that E is an analytical function of the outputs of B and R, and therefore bounded and smooth in the interior of $V \times W$.

The basic algorithm is given by assuming that Blue measures the gradient of the expected payoff with respect to his parameter set, and updates his parameters in that direction. Red simultaneously performs a similar update, with opposite sign. We wish to demand $v \in V$ and $w \in W$, and to assure this we apply *convex projections*, defined as follows: The convex projection of $x \in \mathbf{R}^k$ to a non-empty closed convex set $A \subset \mathbf{R}^k$, denoted by $C_A(x)$, is the unique element of A that satisfies $\frac{1}{2}\|C_A(x) - x\|^2 = \min_{y \in A} \frac{1}{2}\|y - x\|^2$. This is well-defined because $\frac{1}{2}\|y - x\|^2$ is strictly convex as a function of y, see Luenberger (1984). Note that if $x \in A$, $C_A(x) = x$, and if $A = \mathbf{R}^k$, then $C_A(x) = x$ for all x.

The basic algorithm is given by:

$$\begin{aligned} v^{k+1} &\leftarrow C_V(v^k + \alpha \nabla_v E(v^k, w^k)) \\ w^{k+1} &\leftarrow C_W(w^k - \alpha \nabla_w E(v^k, w^k)) \end{aligned} \tag{1}$$

with initial values $(v, w)^0 = (v^0, w^0)$. The positive real number α is the step size.

The goal of the algorithm is to find a minimax solution point (v^*, w^*). A necessary condition for this is that neither side can improve his payoff locally, which is satisfied only if (v^*, w^*) is a fixed point for the mapping (1). The following proposition gives conditions for existence of such fixed points.

Proposition 1. *Let V and W be compact convex sets in $\mathbf{R}^k$, and let $E(\cdot, \cdot)$ be differentiable with continuous gradients. Then a fixed point $(v^*, w^*) \in V \times W$ for the mapping* (1) *exists.*

Now the scene is set for defining the lagging anchor algorithm. Denote the anchors for v and w by $\bar{v}$ and $\bar{w}$ respectively, and set the anchor drawing factor to η. The algorithm is then given by:

$$\begin{aligned} v^{k+1} &\leftarrow C_V(v^k + \alpha \nabla_v E(v^k, w^k)) + \alpha\eta(\bar{v}^k - v^k) \\ w^{k+1} &\leftarrow C_W(w^k - \alpha \nabla_w E(v^k, w^k)) + \alpha\eta(\bar{w}^k - w^k) \\ \bar{v}^{k+1} &\leftarrow \bar{v}^k + \alpha\eta(v^k - \bar{v}^k) \\ \bar{w}^{k+1} &\leftarrow \bar{w}^k + \alpha\eta(w^k - \bar{w}^k) \end{aligned} \tag{2}$$

with initial conditions $v(0) = \bar{v}(0) \in V$ and $w(0) = \bar{w}(0) \in W$.

We assume $\alpha\eta < \frac{1}{2}$, in which case there is no need for convex projections of $\bar{v}$ and $\bar{w}$. A point $(v, w, \bar{v}, \bar{w})$ is a fixed point of (2) if, and only if, (v, w) is a fixed point of (1), $v = \bar{v}$ and $w = \bar{w}$.

From the fact that convex projections are well defined, it follows that the sequence $(v, w, \bar{v}, \bar{w})^k$ is determined by (2). However, actual implementation of the algorithm demands that we calculate the gradients and convex projections, which may be non-trivial. In the following sections, we will show how the algorithm can be implemented for games in matrix form and extensive form, and give some convergence results.

4.1. Matrix games

Recall that for a game matrix $\mathbf{M} \in \mathbf{R}^{n \times m}$, the entry m_{ij} gives Blue's (expected) outcome if Blue and Red choose strategies i and j, respectively. We use linear and complete parameterization of strategies for Blue and Red. This gives the parameter set $V = \{\mathbf{v} \in \mathbf{R}^n : \mathbf{v}^T \mathbf{1}_n = 1, \mathbf{v} \geq \mathbf{0}\}$ for Blue, and $W = \{\mathbf{w} \in \mathbf{R}^m : \mathbf{w}^T \mathbf{1}_m = 1, \mathbf{w} \geq \mathbf{0}\}$ for Red, where $\mathbf{1}_k = (1, 1, \ldots, 1)^T \in \mathbf{R}^k$. Note that V and W are compact and convex. When we refer to *interior points* of V and W, we regard them as sets in the affine subspaces of $\mathbf{R}^n$ and $\mathbf{R}^m$ where the components sum to 1. The interior points of V and W are those with positive value for all components. (Considered as sets in $\mathbf{R}^n$ and $\mathbf{R}^m$, V and W have empty interiors.)

If a game solution $(\mathbf{v}^*, \mathbf{w}^*) \in V \times W$ is an interior point, we also refer to it as being *fully mixed*.

Information sets need not be handled explicitly, because each side has only one. In the notation of Section 3, the function B is given by $B_{\mathbf{v}}(i) = v_i$, representing the fact that the probability of Blue taking action i is v_i. Similarly, $R_{\mathbf{w}}(j) = w_j$ gives Red's probability of taking action j.

In matrix notation, Blue's expected payoff is $E(\mathbf{v}, \mathbf{w}) = \mathbf{v}^T\mathbf{M}\mathbf{w}$. Differentiating this gives the gradients $\nabla_{\mathbf{v}} E(\mathbf{v}, \mathbf{w}) = \mathbf{M}\mathbf{w}$ and $\nabla_{\mathbf{w}} E(\mathbf{v}, \mathbf{w}) = \mathbf{M}^T\mathbf{v}$, so that the learning rule (1) becomes:

$$\begin{aligned} \mathbf{v}^{k+1} &\leftarrow C_V(\mathbf{v}^k + \alpha\mathbf{M}\mathbf{w}^k) \\ \mathbf{w}^{k+1} &\leftarrow C_W(\mathbf{w}^k - \alpha\mathbf{M}^T\mathbf{v}^k) \end{aligned} \tag{3}$$

The update rule (3) of the basic algorithm coincides with Selten's modified gradient update. Proposition 2 gives a simple algorithm for calculating the convex projections. The sets V and W have identical structure, so it suffices to describe the computation of C_V.

For an index set $I \subset \{1, 2, \ldots, n\}$, we define $Q_I : \mathbf{R}^n \to \mathbf{R}^n$ by

$$Q_i(\mathbf{v})_i = \begin{cases} v_i + \dfrac{1 - \sum_{j\in I} v_j}{|I|} & \text{for } i \in I \\ 0 & \text{for } i \notin I \end{cases} .$$

Proposition 2. *The following calculation terminates with* $\mathbf{c} = C_V(\mathbf{v})$.

```
    $I \leftarrow \{1, 2, \ldots, n\}$
    $\mathbf{c} \leftarrow \bar{\mathbf{v}}$

repeat

    $\mathbf{c} \leftarrow Q_I(\mathbf{c})$
    $J \leftarrow \{i \in I : c_j < 0\}$
    $I \leftarrow I - J$

until $(J = \emptyset)$
```

In the bi-matrix game case, Selten showed that the basic algorithm fails to converge toward fully mixed solutions. The following proposition refines this result by stating that the gradient update is orthogonal to the error vector. In the limit when α approaches zero the process therefore keeps a *constant distance* to the solution point. For notational convenience we define $\mathbf{x}^T = [\mathbf{v}^T \ \mathbf{w}^T] \in \mathbf{R}^{n+m}$.

Proposition 3. *Let* $\mathbf{x}^* \in V \times W$ *be a fully mixed game solution, and let* $\mathbf{x}^{k+1}$ *be given by* (3). *If* $\mathbf{x}^{k+1}$ *is in the interior of* $V \times W$, *then* $(\mathbf{x}^k - \mathbf{x}^*)^T(\mathbf{x}^{k+1} - \mathbf{x}^k) = 0$.

We now move to the lagging anchor algorithm for matrix games. Let $\mathbf{x} \in V \times W \times V \times W$ where $\mathbf{x}^T = [\mathbf{v}^T \quad \mathbf{w}^T \quad \bar{\mathbf{v}}^T \quad \bar{\mathbf{w}}^T]$. Using the calculation procedure for C_V and C_W given in Proposition 2, we can implement the learning rule (2):

$$\begin{aligned}
\mathbf{v}^{k+1} &\leftarrow C_V(\mathbf{v}^k + \alpha\mathbf{M}\mathbf{w}) + \alpha\eta(\bar{\mathbf{v}}^k - \mathbf{v}^k) \\
\mathbf{w}^{k+1} &\leftarrow C_W(\mathbf{w}^k - \alpha\mathbf{M}^T\mathbf{v}) + \alpha\eta(\bar{\mathbf{w}}^k - \mathbf{w}^k) \\
\bar{\mathbf{v}}^{k+1} &\leftarrow \bar{\mathbf{v}}^k + \alpha\eta(\mathbf{v} - \bar{\mathbf{v}}^k) \\
\bar{\mathbf{w}}^{k+1} &\leftarrow \bar{\mathbf{w}}^k + \alpha\eta(\mathbf{w}^k - \bar{\mathbf{w}}^k)
\end{aligned} \tag{4}$$

The following proposition gives a convergence result for the algorithm.

Proposition 4. *Let* $(\mathbf{v}^*, \mathbf{w}^*) \in V \times W$ *be a fully mixed game solution, and let* $\{\mathbf{x}^k\}$ *be given by* (4)*. Let* $d > 0$ *be the Euclidean distance from* $(\mathbf{v}^*, \mathbf{w}^*)$ *to the boundary of* $V \times W$*. Assume that the Euclidean distance between* $(\mathbf{v}^0, \mathbf{w}^0)$ *and* $(\mathbf{v}^*, \mathbf{w}^*)$ *is less than* d*. Then, for a sufficiently small* $\alpha > 0$, $\{\mathbf{x}^k\}$ *never hits the boundary, and converges exponentially toward some solution point* $\mathbf{x}$.

Although convergence has not been proven for matrix games in general, the algorithm has worked well on a large sample of test games. We have tested random matrices with dimension up to 100 and entries uniformly distributed in $(-1, 1)$, and observed exponential convergence.

With our notation, Selten's anticipatory learning rule gives:

$$\begin{aligned}
\mathbf{v}^{k+1} &\leftarrow C_V(\mathbf{v}^k + \alpha\mathbf{M}C_W(\mathbf{w}^k - \alpha\mathbf{M}^T\mathbf{v}^k)) \\
\mathbf{w}^{k+1} &\leftarrow C_W(\mathbf{w}^k - \alpha\mathbf{M}^T C_V(\mathbf{v}^k + \alpha\mathbf{M}\mathbf{w}^k)).
\end{aligned}$$

From our experiments, this also gives exponential convergence on large random matrices.

Recall that the purpose of our algorithm is to produce approximate solutions to large games with nonlinear and incomplete parameterization, rather than compete with existing efficient algorithms for linear programming, which are standard for solving matrix games. The lagging anchor algorithm does, however, appear to have similarities with *interior point methods* for linear programming, see Padberg (1995), in that it searches the interior of the set of admissible solution points and works to solve the primal and dual problems simultaneously. In our game setting, the primal problem is Blue's optimization problem, while the dual problem is Red's. The similarities end there, however, as interior point methods consist of finite sequences of projections, rather than incremental search that converges toward a solution. To our knowledge, interior point methods cannot be extended beyond the context of matrix games.

4.2. Extensive-form games

Our agent design in Section 3 allows non-linear and incomplete strategy representations in extensive-form games. We will show how our algorithm can be adjusted to that context, and give a stability result.

The learning rule (2) contains the non-trivial operation of evaluating the players' gradients. We will explain how this is handled, and present the algorithm in schematic pseudo-code. The learning rule also contains convex projections, which are needed if the domains of the players' parameters are restricted. We are not able to specify one general algorithm for this, because different parameter sets will require different methods. General optimization techniques such as the solution of Kuhn-Tucker conditions should be useful, however.

We split the process of estimating the gradient in two. First we estimate the gradient of the agent's expected payoff with respect to its action probabilities (dE/dB). Then we calculate the gradient of the agent's action probabilities with respect to its parameters (dB/dv). We combine these gradients, using the chain rule of differentiation, to get the estimated gradient of the agent's expected payoff with respect to its parameters (dE/dv). We then update the agent's parameters in the gradient direction, multiplied by the step size α.

4.2.1. Training patterns. The algorithm will be explained using the notion of *training* patterns, which is common in the context of neural networks. A training pattern $((s, a), f)$ is a combination of input (s, a) and feedback f. The application of a training pattern is implemented as a gradient search step (with length proportional to α) in the parameter space V toward minimizing $\frac{1}{2}(B_v(s, a) - f)^2$. In other words, B's parameter state is modified to make its output closer to the feedback for the given input.

In this way the second part of the gradient calculation (dB/dv) is integrated with the parameter update. Assume that the derivative of the payoff with respect to the output of B on input (s, a) is estimated to be d. Then training with pattern $((s, a),\ B_v(s, a) + d)$ implements both the calculation of the gradient and the parameter update in the gradient direction.

4.2.2. Sampling the consequences of alternative actions. The general problem of calculating the gradients of the players' payoffs analytically is far too complex to be considered. We therefore resort to estimating them based on the outcomes of sample games, where our Blue and Red agents play. This is the reason why we denote our algorithm *reinforcement learning*. To simplify the presentation we describe the estimation of Blue's gradient only. The method we develop relies on what we might call "what-if" analysis. First a sample game is completed, and then the course of the game is analyzed. For each decision node visited, the hypothetical consequences of the different available actions are estimated. In Section 4.2.4, we show how to convert these estimates into training patterns for $B_v(\cdot, \cdot)$.

Estimating the consequence of an action in a decision node is equivalent to estimating the expected payoff for the node resulting from it. Now it may seem as if we are contradicting ourselves. In Section 2 we mentioned the fact that nodes of the game tree may not have unique values, and now it seems as though we are attempting to estimate just this. This is not the case, because we are attempting to estimate the expected payoff for nodes, *given the current Blue and Red strategies*. Given a Blue and Red player, there is no problem with defining the value of a node as its average payoff.

The simplest way of estimating the expected payoff for a node is by sampling. One simulates one or more games from the node in question, and takes the sample average as the estimate. This is clearly unbiased. We consider the case where only one additional game is completed for each decision as the *canonical form* of the algorithm. Note that we can

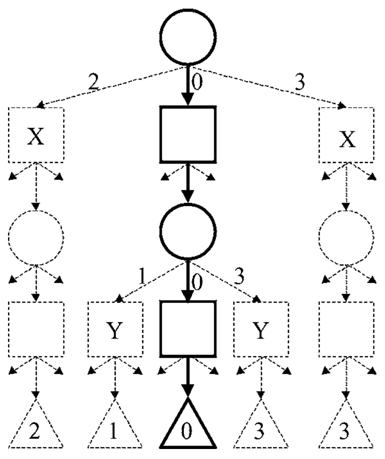

Figure 4. The canonical form of the algorithm.

take the outcome of the original sample game as the estimated consequence of the action made in that game, so only actions that deviate from the course of the game need to be sampled.

Figure 4 illustrates the algorithm. Circular and square nodes represent decision nodes for Blue and Red respectively, while the triangular nodes represent terminal states where the game rules define Blue's payoff.

The vertical path represents the course of the game. In the top node, Blue had two deviating actions, leading to game states labelled "X". In order to estimate their utility, an additional sample game is completed from each of these states as indicated in the figure. In the central node of the figure Blue also has two alternatives to the action taken in the original game, leading to states labelled "Y". The numbers attached to the arcs in the figure give the estimated payoffs resulting from Blue taking these actions in the preceding decision nodes. These are the numbers that are turned into training patterns in Section 4.2.4.

4.2.3. Truncated sampling using TD-learning. The canonical form of the algorithm requires that one sample game be completed for each deviating decision Blue can make throughout the original sample game. The number of additional games that must be completed to estimate the gradient from one sample game is therefore less than the branching factor multiplied by the tree depth. The sampling of results from alternative decisions therefore slows down the algorithm by a linear factor of the tree depth, which is no disaster from the viewpoint of computational complexity. Still, the computational burden from the canonical form of the algorithm can be very significant in practice. Besides, it introduces considerable noise in the estimated gradients. We therefore introduce what we call *truncated sampling*. The idea of this is to use some external function for estimating the node values. The purpose of this function is only to estimate the consequences of alternative decisions, not to take part in the game playing. In the example of figure 4, this means that the nodes labelled "X" and "Y" are evaluated by the function instead of being played out.

Given a Blue and Red player, the course of the game is a Markov process, where the states are game-tree nodes. The fact that the players have imperfect information and act on information sets is not a problem, as any pair of mixed strategies for Blue and Red induces a Markov process on the set of nodes. A standard method for estimating the expected outcome of a Markov process is TD-learning, and this we use. After a sample game has been completed, the course of the game (sequence of nodes visited) is used to update the node evaluator through TD-learning. This essentially means that the evaluation of a given node is tuned toward a weighted average of the actual payoff and the function's own evaluations of intermediate nodes.

The use of truncated sampling through a function tuned by TD-learning will of course introduce a bias in the gradient estimation, unless the function produces exact evaluations of nodes. At the beginning of a training session, the node evaluator function will normally be initialized randomly, in which case the gradient estimation will be very poor until the function starts giving meaningful evaluations. Only experiments can determine whether the gain in speed and reduced randomness of truncated sampling outweighs the disadvantage of biased gradient estimation.

4.2.4. Calculation of feedback. We now describe how to construct feedback signals based on estimates of the consequences of the different legal actions in a decision node, whether these estimates come from the canonical algorithm or a function trained by TD-learning. If the function B is implemented as a lookup-table, representing a behavioral strategy explicitly, it contains one probability parameter for each legal action in every information state. In this case the updates in question become equivalent to the calculations in the matrix game case. The algorithm specified in update rule (3) and Proposition 2 should be applied with the vector of payoff estimates playing the role of $\mathbf{Mw}$. In the following, we assume that B is given as a function approximator (e.g. a neural net), structured so that it produces output values in the range (0,1).

Denote the current decision node by n, and its information state by s. Let $\mathbf{A}$ be the vector of legal actions in n, and let $\mathbf{N}$ be the corresponding vector of nodes resulting from taking these actions. Let $\mathbf{P}$ be the vector of Blue's probability weights of actions at state s: $\mathbf{P} \leftarrow B_v(s, \mathbf{A})$, and let $\mathbf{E}$ be the vector of the estimated payoffs for the nodes $\mathbf{N}$. Let p_{sum} be the sum of $\mathbf{P}$'s components: $p_{sum} \leftarrow \mathbf{1}^T\mathbf{P}$. The general idea is that $p_{sum} = 1$, but this may not be satisfied exactly for all information states.

Our basic idea is to use the sum $\mathbf{P} + \mathbf{E}$ as the vector of feedback signals associated with the vector of inputs given by $(s, \mathbf{A})$. In this way, B is instructed to modify its output according to how successful a (s, a)-pair turned out. However, the relative values of the components of $\mathbf{E}$ are more important than their absolute values. We therefore shift $\mathbf{E}$ by subtracting its expected value, under B's current strategy, from each component: $\mathbf{E} \leftarrow \mathbf{E} - \mathbf{1}(\mathbf{E}^T\mathbf{P}/p_{sum})$.

With this modification, the feedback vector $\mathbf{P} + \mathbf{E}$ should be approximately balanced, in the sense that training will push B's output upwards for some (s, a)-inputs, and downwards for others, and that the sum of the changes be approximately 0. The sum will only be approximately 0, as B's responsiveness to feedback will vary for different inputs. Also, updates may have side effects for B's output in other information states. Therefore it may happen that B drifts away from representing probability distributions. This is not necessarily a disaster in itself, as we will treat B's output values as probability weights, and scale them to sum to 1. But if the drift is left unchecked, it may happen that all of B's output values drift toward to 0 or 1, for a given information state. This can be harmful to the learning mechanism, as B's responsiveness to feedback may become very low close to these extreme output values. Therefore, to keep this drift under control, we add a term to the feedback that pushes B toward valid probability distributions. The correction term is proportional to the inconsistency, with proportionality factor u, and the modified feedback vector is given by $\mathbf{P} + \mathbf{E} - \mathbf{1}u(p_{sum} - 1)$.

4.2.5. Schematic description of the algorithm. The algorithm we have described using truncated sampling and function approximators is rather involved, and we now present a schematic pseudo-code implementation of it. We apply the convention of displaying vector quantities in **boldface**. Keywords are displayed in **`boldface courier`**.

```
repeat $Iteration$ times {
    $\langle$ play a game $g$ between Blue and Red $\rangle$
    for $\langle$ each decision node $n \in g$ $\rangle$ do {
        $\mathbf{A} \leftarrow \langle$ vector of legal actions at $n\rangle$
        $\mathbf{N} \leftarrow \langle$ vector of nodes resulting from actions $\mathbf{A}$ at $n\rangle$
        $s \leftarrow \langle$ the information state of $n\rangle$
        if $\langle$ Blue on turn in $n\rangle$ $\{\mathbf{E} \leftarrow Evaluator(\mathbf{N});\ \mathbf{P} \leftarrow B(s, \mathbf{A})\}$
        else                   $\{\mathbf{E} \leftarrow -Evaluator(\mathbf{N});\ \mathbf{P} \leftarrow R(s, \mathbf{A})\}$
        $p_{sum} \leftarrow \mathbf{1}^T\mathbf{P}$
        $e \leftarrow \mathbf{P}^T\mathbf{E}/p_{sum}$
        $\mathbf{E} \leftarrow \mathbf{E} - \mathbf{1}e$
        $\mathbf{F} \leftarrow \mathbf{P} + \mathbf{E} - \mathbf{1}u(p_{sum} - 1)$
        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 $\{(s, \mathbf{A}), \mathbf{F}\rangle\ \}$
    }
    $\mathbf{V} \leftarrow (1 - \alpha\eta)\mathbf{V} + \alpha\eta\bar{\mathbf{V}}$
    $\mathbf{W} \leftarrow (1 - \alpha\eta)\mathbf{W} + \alpha\eta\bar{\mathbf{W}}$
    $\bar{\mathbf{V}} \leftarrow (1 - \alpha\eta)\bar{\mathbf{V}} + \alpha\eta\mathbf{V}$
    $\bar{\mathbf{W}} \leftarrow (1 - \alpha\eta)\bar{\mathbf{W}} + \alpha\eta\mathbf{W}$
    $\langle$ train $Evaluator$ from $g$ using TD-learning $\rangle$
}
```

The vectors $\mathbf{V}$, $\bar{\mathbf{V}}$, $\mathbf{W}$, $\bar{\mathbf{W}}$ represent the parameter states and anchor states of B and R respectively. 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 step size in these updates is α.

In the canonical version of the algorithm, *Evaluator*(**N**) is substituted with the vector of outcomes of sample games starting from the nodes **N**, and no TD-learning occurs. In the basic version of the algorithm, the four statements updating $\mathbf{V}$, $\bar{\mathbf{V}}$, $\mathbf{W}$ and $\bar{\mathbf{W}}$ do not apply.

4.2.6. A stability result. We now give a local convergence result for the algorithm. The result is proven for the idealized update rule (2), and therefore disregards the sampling error of the gradient estimation. Recall that we assume that the game is a finite two-player zero-sum game, which means that it can be transformed to a matrix game. Let $\tilde{\mathbf{M}} \in \mathbf{R}^{n\times m}$ be this underlying game matrix, and let $\tilde{V} \subset \mathbf{R}^n$ and $\tilde{W} \subset \mathbf{R}^m$ be the corresponding sets of mixed strategies. Because the agents' behavior is assumed to be smooth functions of their parameters, there exist smooth functions $\varphi_V : V \to \tilde{V}$ and $\varphi_W : W \to \tilde{W}$ that map the players' parameter states to mixed strategies of the underlying matrix game. (Note that the underlying matrix game is only a tool for proving results, and will not normally be assembled.)

We assume that V and W are k-dimensional sets. If $(v^*, w^*) \in V \times W$ is an interior point, we define $\mathbf{M}^{vv}$, $\mathbf{M}^{vw}$ and $\mathbf{M}^{ww}$ to be the $k \times k$-matrices with entries

$$(\mathbf{M}^{vv})_{ij} = \frac{\partial^2 E(v^*, w^*)}{\partial v_i \partial v_j}, \ (\mathbf{M}^{vw})_{ij} = \frac{\partial^2 E(v^*, w^*)}{\partial v_i \partial w_j}, \text{ and } (\mathbf{M}^{ww})_{ij} = \frac{\partial^2 E(v^*, w^*)}{\partial w_i \partial w_j}.$$

Under these definitions, Proposition 5 gives conditions for local stability of fixed points for the lagging anchor algorithm.

Proposition 5. *Let* $(v^*, w^*) \in V \times W$ *be an interior point such that* $\nabla E(v^*, w^*) = \mathbf{0}$. *If* $\mathbf{M}^{vv} = \mathbf{M}^{ww} = \mathbf{0}$ and $\mathbf{M}^{vw}$ *is invertible, then, for a sufficiently small* α, (v^*, w^*, v^*, w^*) *is an asymptotically stable equilibrium point for* (2) *with exponential convergence.*

The condition that $\mathbf{M}^{vw}$ must be invertible may be more strict than necessary, but it cannot be eliminated completely. As an example, consider the trivial game with a constant payoff of zero, giving $\mathbf{M}^{vw} = \mathbf{0}$, which is not invertible. For this game, any point $(v, w) \in V \times W$ is a fixed point, but none are asymptotically stable. If $\mathbf{M}^{vw}$ is invertible, any small deviation from the solution by one player gives the opponent an incentive to respond by changing his strategy, thereby setting the convergence process in motion.

Loosely speaking, the condition $\mathbf{M}^{vv} = \mathbf{M}^{ww} = \mathbf{0}$ means that changing one side's behavior (from the equilibrium point) has very little impact on the payoff. In Corollary 1 below, we relate this to fully mixed solutions of the underlying matrix game.

Corollary 1. *Let* $(v^*, w^*) \in V \times W$ *be an interior point such that* $\nabla E(v^*, w^*) = \mathbf{0}$. *Assume that* $\mathbf{M}^{vw}$ *is invertible and* $(\varphi_V(v^*), \varphi_W(w^*)) \in \tilde{V} \times \tilde{W}$ *is a fully mixed solution point for the matrix game* $\tilde{\mathbf{M}}$. *Then, for a sufficiently small* α, (v^*, w^*, v^*, w^*), *is an asymptotically stable equilibrium point for* (2) *with exponential convergence.*

The matrix $\tilde{\mathbf{M}}$ need not be the underlying game matrix for the entire game, as long as it represents all strategies that Blue and Red apply with positive probability in a neighborhood of (v^*, w^*).

4.2.7. The impact of random noise. We are not able to complete any formal analysis of the impact of random noise in the gradient estimation, but we attempt to give some indication of what it may be. A noisy version of the update rule (2) would look something like this:

$$\begin{aligned} v^{k+1} &\leftarrow C_V(v^k + \alpha(\nabla_v E(v^k, w^k) + \text{“noise}_v\text{”})) + \alpha\eta(\bar{v}^k - v^k) \\ w^{k+1} &\leftarrow C_W(w^k - \alpha(\nabla_w E(v^k, w^k) + \text{“noise}_w\text{”})) + \alpha\eta(\bar{w}^k - w^k). \\ \bar{v}^{k+1} &\leftarrow \bar{v}^k + \alpha\eta(v^k - \bar{v}^k) \\ \bar{w}^{k+1} &\leftarrow \bar{w}^k + \alpha\eta(w^k - \bar{w}^k) \end{aligned} \tag{5}$$

When α decreases, one may hope that the solution of (5) converges in probability toward the solution of (2).

In the update rule (5), we observe that the random noise affects the agents' parameters v and w directly, while the anchor parameters $\bar{v}$ and $\bar{w}$ are affected only indirectly. The anchor parameters $(\bar{v}, \bar{w})^k$ represent a weighted average of $(v, w)^i$ for $i < k$, and can therefore be expected to follow a smoother path with less random noise than $(v, w)^k$. For large values of k, we may therefore expect the agents to perform better with the anchor parameters $(\bar{v}, \bar{w})$ than with their actual parameters $(v, w)^k$. In the early stage of the training process, this relation may be reversed, as the agents then mainly learn to eliminate useless strategies. The fact that the anchors are lagging behind should imply that they eliminate useless strategies more slowly.

5. Application to a simplified poker game

We now turn to applying the lagging anchor algorithm to a simplified poker game. The rules of our game are as follows:

Blue and Red each receive a card with a random value independently uniformly distributed from 0 to 1. Blue then either *bets* or *calls*. If he calls, the player with the higher card wins one unit from the opponent (the *ante* of the game). If Blue bets, Red must either *call* or *fold*. If he folds, Blue wins the ante. If he calls, the player with the higher card wins two units from the opponent.

This game, although simple in structure, has an infinite number of pure strategies for each side, as the number of cards is infinite (in fact not even countable). A pure strategy must specify one of the two legal actions for each possible card, and can therefore be represented by a subset of the unit interval [0,1]. To ensure that the expected outcome is well-defined, only sets measurable with respect to the (σ-algebra used for drawing the cards, possibly the Lebesgue algebra, can be allowed. With these vast and complex sets of pure strategies, the elementary game theory that we have surveyed does not guarantee existence of a minimax solution.

Nevertheless, a solution can in fact be constructed, as shown by von Neumann and Morgenstern (1953). For Blue, the (almost surely) unique solution is to bet (bluff) with cards below 1/10, call with cards in the interval (1/10, 7/10) and bet with cards greater than 7/10. Red's solution is not unique. However, if $0 < a < b < 1$, there is no reason for Red to assign a positive probability to calling with a and folding with b. Therefore Red strategies that apply a calling limit c, and call if and only if his card value exceeds c, weakly dominate those that do not. Red's dominant solution is to apply a calling limit of 4/10, which is the only pure strategy solution for Red. The value of the game is 1/10.

5.1. Player representation

We represent playing strategies for Blue and Red by nonlinear functions implemented as neural nets. Blue's net takes the card value as input, and gives the probability of betting as output. For simplicity, we use a feed-forward net with sigmoid activation functions and one layer of hidden units. The number of hidden nodes was somewhat arbitrarily chosen to be eight. Blue's probability of calling is naturally calculated as 1 minus the net's output. This representation obviously produces valid probability distributions. The structure of Red's net is identical to that of Blue's, except that the output is interpreted as his probability of calling. The anchors for Blue and Red are neural nets that are structurally equivalent to the playing nets.

For a general introduction to neural nets, we refer to Hassoun (1995). In our applications, it is sufficient to view a neural net as a flexible nonlinear family of functions with a set of adjustable parameters, commonly referred to as *weights*. From an artificial intelligence point of view, the state of the weights represents the knowledge of the net. We deliberately chose a standard neural net design, rather than tailoring our agent representation to the problem. This was done to study the algorithm's ability to solve the problem without domain knowledge being added to the representation.

5.2. Experimental results

The game involves only one decision for each side, so there is little need for truncated sampling. We therefore apply the canonical version of the training procedure. Each training session starts with the Blue and Red playing nets being randomly initialized. When anchors are applied, they are initialized as clones of the playing nets.

Because Blue's solution strategy is unique, and also more complex than Red's dominant one, we focus our analysis on that side's performance. Blue's expected outcome against a reference opponent with the strategy of calling if his card exceeds a given value can be calculated by numerical integration. These strategies dominate any mixed strategy for the Red side, so the expected outcome against a globally optimizing opponent, the *Geq* of the Blue player, can be estimated accurately by searching for the reference opponent's most effective calling limit.

For each of the four experiments to be described below, a total of 10^6 training games were played. This may seem like an unreasonably large number, for a game with such a simple structure. However, the game solution is discontinuous, and the neural nets used can only represent smooth functions. Therefore good approximation may require extreme values for the nets' weights.

5.2.1. Basic algorithm. First we present results with the basic algorithm, i.e. without anchors. Figure 5 gives the *Geq* performance for Blue as a function of training games. The step size α was set to 0.2.

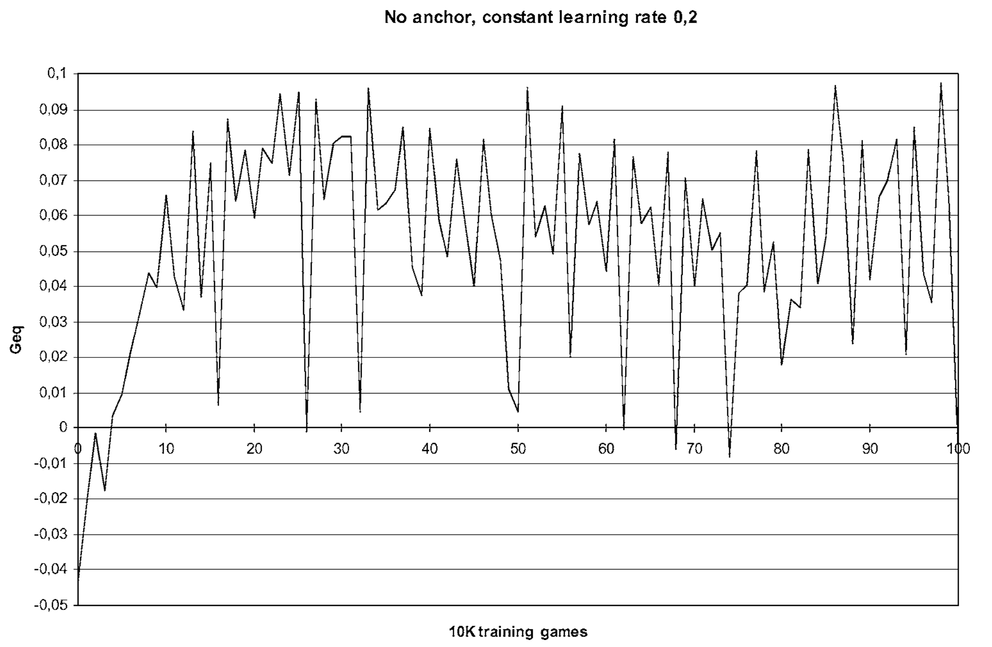

Figure 5. Geq with constant step size and no anchor.

The graph shows that learning definitely takes place, and the *Geq* fluctuates between values around 0 and values close to the maximum performance of 0.1. It is not obvious whether the fluctuations are caused by the randomness in the training procedure or the underlying dynamics of the system. To test this we have also run experiments with a variable step size, starting at 1.0 and falling inversely proportionally with the number of games to 0.02. This produced the picture seen in figure 6.

This graph shows far better convergence toward the maximum value of 0.1, which indicates that the basic algorithm does in fact converge for the given game, as long as the random noise in the gradient sampling is eliminated. This is in line with Selten's result concerning convergence of the basic algorithm toward pure strategy solutions.

While we were inconclusive about the periodicity in figure 5, the graph in figure 6 is clearly periodic. This should not surprise us, as the basic algorithm has been shown to behave periodically for matching pennies. The pattern seen in figure 6 can be explained as follows. In the first phase of training, Blue and Red learn to eliminate useless strategies. Blue learns to bet high and low cards in general, producing a U-shaped betting curve, and Red learns to call with high cards, giving an S-shaped calling curve. In this phase no oscillation is seen. Once the general shape of the curves are established, oscillation begins. Assume Red's S-curve happens to be shifted to the right, compared to the S-curve of the game-theoretic solution, which means that he folds too often. Blue will then benefit from bluffing more often, and betting for value more seldom (as Red's average calling card goes up), shifting his U-curve to the right. When Blue's U-curve has moved to the right of the

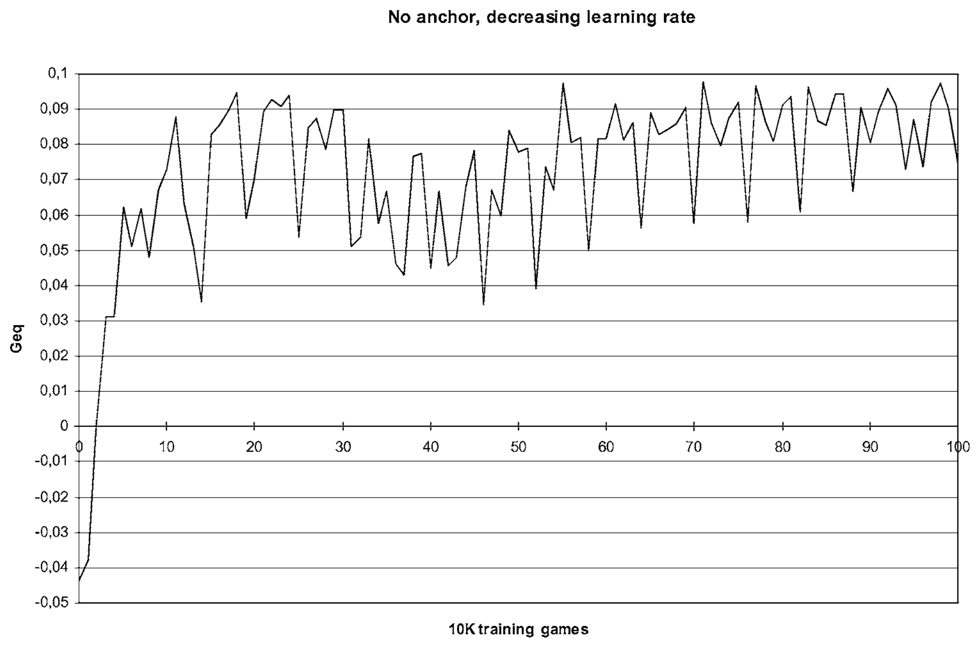

Figure 6. Geq with decreasing step size and no anchor.

U-curve of the game-theoretic solution, Red will benefit from calling more often, shifting his S-curve to the left, and so on.

5.2.2. Lagging anchor algorithm. We test the lagging anchor algorithm with the step sizes of the playing nets identical to those used with the basic algorithm.

The anchor state can be seen as a weighted average of previous player states, with $\alpha\eta$ determining the weight to be placed on the current state relative to previous states. If the player state oscillates, it appears reasonable to choose an η that makes the anchor state roughly equal to the average of the player state over one cycle. The previous graph seems to indicate that the process does oscillate, and near the end of the series there seems to be approximately 5×10^4 games between each top. This means that $\alpha\eta$ should be of the order 2×10^{-5} near the end of the series. At that point, the step size is 0.02, implying an η of the order 10^{-3}. After some experimenting, the value $\eta = 0.002$ was chosen, and used for both experiments.

Figure 7 gives Blue's *Geq* as a function of training games with a constant step size of 0.2.

Comparing figure 7 to figure 5 shows significantly better performance than the basic algorithm. In particular the anchor neural net gives a high *Geq* when used for playing. The fact that the anchor net performs better than the playing net is in line with our analysis in Section 4.2.7. Also the anchor does appear to succeed in drawing the player net closer to the solution, as the performance of the playing net is also better than that seen in figure 5.

Figure 8 shows the graph produced with decreasing step size. In this case, the *Geq* of the player net is similar to that resulting from the basic algorithm. The anchor net, however,

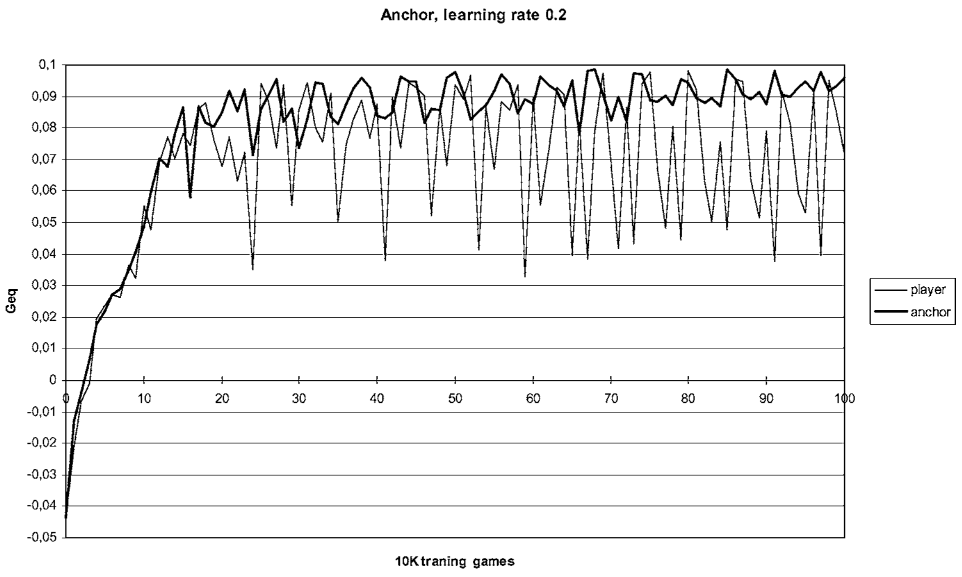

Figure 7. Geq with anchor and constant step size.

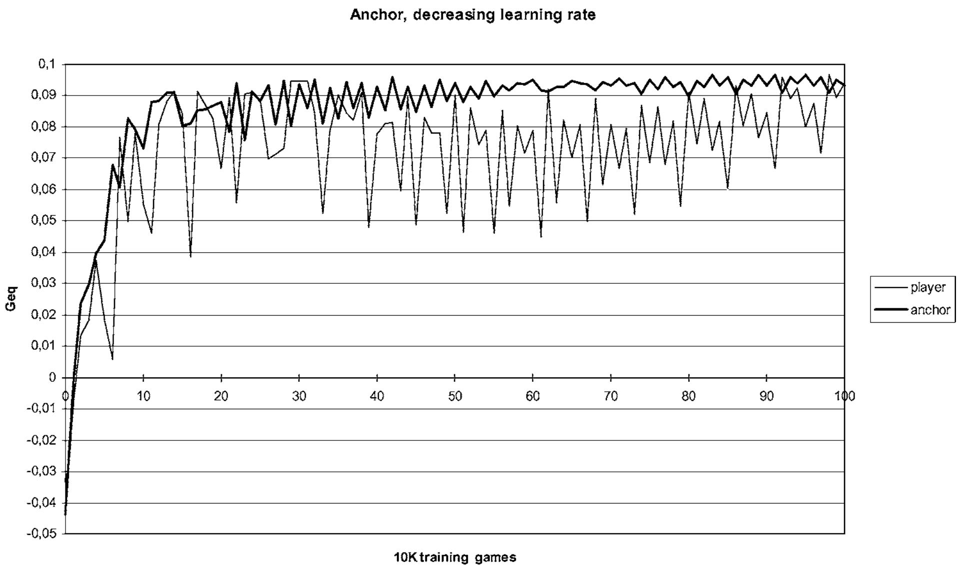

Figure 8. Geq with anchor and decreasing step size.

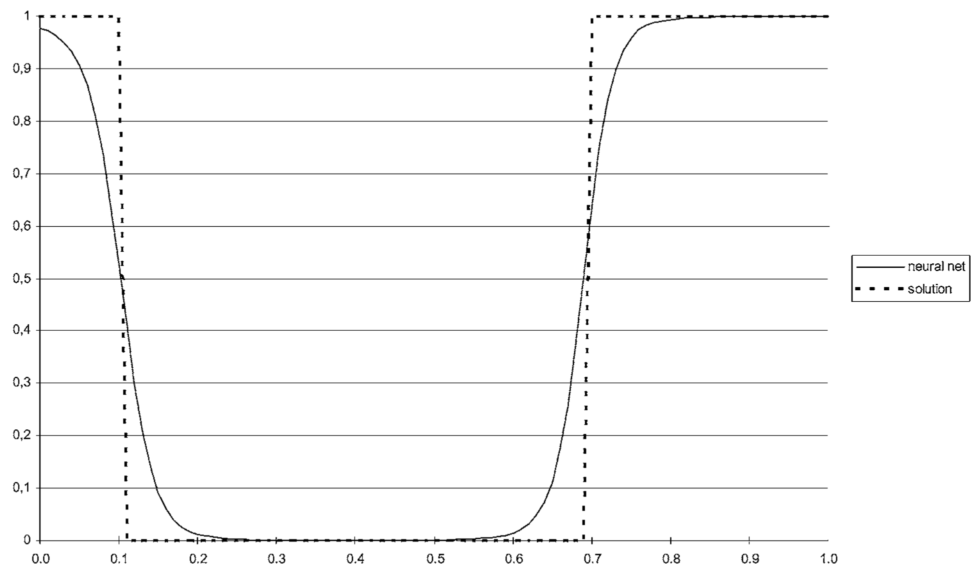

Figure 9. Neural net approximation to the game-theoretic solution.

shows far better performance, as in the case with constant step size. In both cases, the convergence seems better than for the basic algorithm, which indicates that the anchor-based algorithm is preferable also in cases where the basic algorithm behaves well.

Figure 9 gives the Blue neural net output after the last training session, compared to the exact game solution.

We see a good correspondence between the game theoretic solution and the U-shaped neural net output.

6. Application to an air campaign game

We now give an application of the lagging anchor algorithm to the game Campaign, a simple model of a two-sided military air campaign. To keep this section from growing excessively long, we cover only the main points.

6.1. *Campaign*

Each player starts with five *units* and zero *accumulated profit points*. The game consists of five decision points, or *rounds*. In each round, both players simultaneously allocate their remaining units between three roles: defense (D), profit (P) and attack (A). The number of units a player allocates to P is added to his accumulated profit. Units allocated to A have the potential of destroying one enemy unit each, and destroyed units are unavailable for allocation in subsequent rounds. Each unit allocated to D neutralizes two enemy attacking units. The number of remaining units cannot be negative, and attacking units that would reduce the opponent's number of units below zero have no effect. After the fifth round, the *score* is calculated for each player as the sum of accumulated profit and the number of remaining units. The side with the higher score wins, and if the scores are equal the game is a draw. The winner gets one point, the loser none, and with a draw the players get half a point each. By symmetry, the game has value 0.5 for both sides. The game has imperfect information because the players act simultaneously, and no deterministic minimax solution exists. This means that although the rules of the game are deterministic, optimal play requires random actions by the players.

A game state can be represented by an integer quadruple (b, r, p, n), with b being the number of Blue units remaining, r the number of Red units, p Blue's lead in accumulated profit, and n the number of rounds left. It is sufficient to represent Blue's lead in profit points, as the accumulated profits have no other function than shifting the final score of the players. We represent an action by a player by the integer triple (D, P, A), where the sum of these must equal the player's number of remaining units.

Campaign has a total of 2607 game states, and the number of pure strategies is far beyond what can be represented in a present-day computer, as a pure strategy must specify an action in every game state. It is still possible to solve the game with a combination of dynamic programming and linear programming, starting with the states close to the end of the game. It is also possible to calculate the *Geq* of a player using similar calculations. Dahl and Halck (1998) gives a more formal definition of the game together with solution algorithms, properties of the solution and military interpretation. The fact that Campaign is nontrivial, and still solvable, makes it a good environment for testing learning algorithms for games with imperfect information.

Again we use neural nets as players. The nets take a state and a possible action as input, and produce as output the probability weight for taking the given action in the given state, as described in Section 3. For a given state, the neural net outputs are scaled to sum to unity. Blue and Red each have a playing net and an anchor net with identical structure. As for the poker game, we use simple feed-forward nets with one layer of eight hidden nodes and a single output node. The nets have six input nodes, corresponding to the numbers (o, p, n, D, P, A). Here o is the opponent's number of units, and p is the player's lead in accumulated profit points (may be negative). The player's own number of units is not explicitly encoded, as this equals the sum of the last three input values.

As in the simplified poker application, we deliberately choose a "vanilla flavored" neural net design, and use only "raw" input features, with the intention of studying the algorithm's performance without added external knowledge. Since we have access to the game solution, we could easily design an agent representation that would make the learning task simple, but this is not our purpose.

6.2. Implementation

In game states where a player has all five units intact, the number of legal actions is 21, which implies that the number of deviating actions may be as high as 20. With five stages and two players, this means that the canonical version of the algorithm may require that up to 200 deviated games be completed for each sample game. This strongly indicates that we should apply the truncated sampling version of the algorithm.

Our implementation closely follows the algorithm presented in Section 4.2.5, but there is one point that deserves special mention. The algorithm specifies that the game tree nodes resulting from the various actions be evaluated. When a game-tree representation is constructed, one must choose which side moves first (although the opponent is not informed about which action was taken). This is similar to how we modeled matching pennies as extensive-form game in Section 2. When we analyze the effect of different actions by Blue, we assume that Red's decision was made first, so that the evaluator can evaluate the resulting 4-tuple game state. Conversely, when we analyze Red's legal actions, we assume that Blue's action was made first. Therefore our game state evaluator only needs to evaluate 4-tuple game states (b, r, p, n). The game state evaluator is implemented as a neural net, similar to those used by the players, except that it has only four input nodes, associated with the state dimensions (b, r, p, n).

6.3. Experimental results

The algorithm has been run with step size $\alpha = 0.5$, anchor parameter $\eta = 0.002$ and normalization factor $u = 0.5$. The TD-training of the state evaluation net was run with a step size of 0.5 and $\lambda = 0$. Figure 10 gives the average *Geq* for the Blue and Red playing nets, together with the average *Geq* for the Blue and Red anchor nets, as a function of training games.

The graph shows that the neural nets make definite progress when evaluated by the *Geq* measure. As for the poker game, the anchor evaluations are less noisy than the player evaluations, and also give better performance on average. The exact solution of Campaign is very erratic, in the sense that similar game states may have very different optimal probability

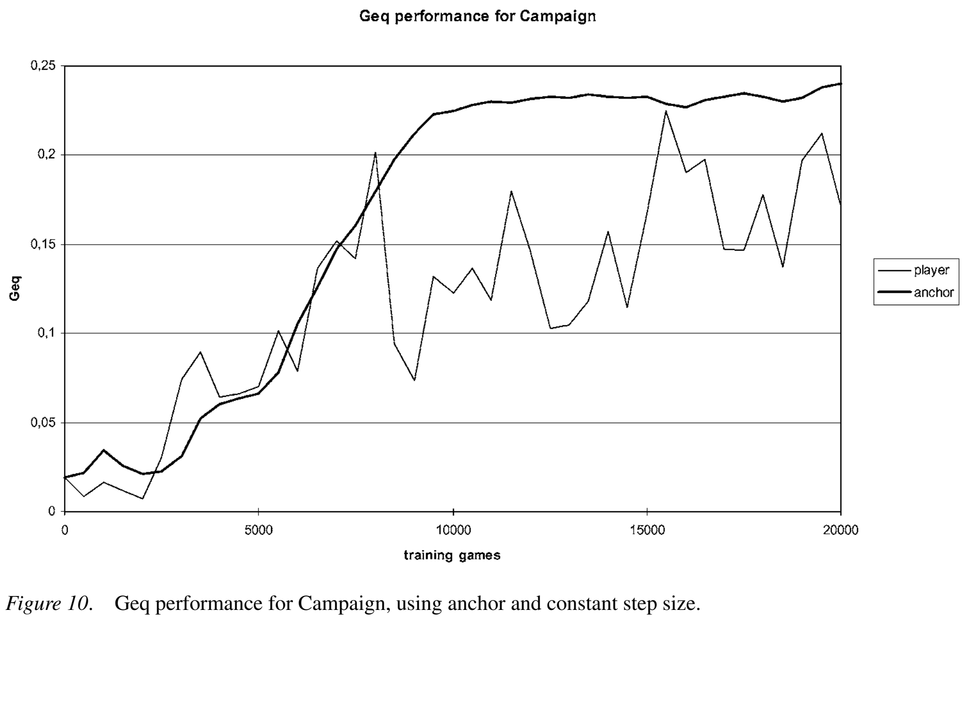

Figure 10. Geq performance for Campaign, using anchor and constant step size.

distributions over the set of actions. Therefore, creating an approximate solution based on smooth functions like neural nets may not be easy. As an example, the last round is qualitatively different from previous rounds, as allocating all units to profit is then always optimal. Other game states require careful probability mixing of different actions, to avoid any bias that can be exploited by a globally optimizing opponent.

It may well be that a *Geq* of 0.24, which means winning 24% of the games (counting a draw as half a win), is close to the performance limit given by the expressive power of our relatively small neural net. For the sake of comparison we have also implemented optimization of the *Geq* performance directly. The algorithmic details of this are somewhat complex, and are documented by Dahl and Halck (2000). Note that optimization of *Geq* is possible only because of the limited complexity of Campaign, so it does not present a competing algorithm for large games. The optimization process stagnates around 0.25, which strongly indicates that the lagging anchor algorithm comes close to the performance limit of the design.

Recall also that the *Geq* measure gives the performance against the given net's most effective opponent, which is a very strict measure. When playing against the calculated game-theoretic solution, our nets score approximately 0.42, which is considerably closer to the upper limit of 0.5. The way to improve performance, apart from increasing the network size, may lie in pre-processing the input, so that the mapping from the neural net input to the desired output is smoother.

In Bakken and Dahl (1998), neural nets identical to those used here were trained to imitate the game-theoretic solution of Campaign by supervised learning, producing weaker results than those cited here for the lagging anchor algorithm. This is similar to experiences made with backgammon. Tesauro and Sejnowski (1989) used supervised learning to imitate human professional play, but reinforcement learning based on temporal differences gave better results (Tesauro, 1992). The reason why supervised learning fails in our case is that the root-mean-square (RMS) error measure that it minimizes correlates badly with our *Geq* measure. If the RMS error were to converge to 0, the *Geq* performance would converge to the game's value of 0.5. However, because the game-theoretic solution is very irregular, the neural net must make significant compromises when imitating it. The RMS error measure does not incorporate the relative importance of the different choices, so the neural net makes sub-optimal compromises. In a sense the supervised learning is "syntax without semantics", because it disregards the implications of actions. A striking analogy is that of an amateur chess player who does better by learning from experience than by imitating the tricky moves seen in Kasparov's games.

6.4. Relation to minimax TD-learning

The reader may have observed that Campaign is a Markov game, and one can therefore apply minimax Q-learning or minimax TD-learning to it. In Dahl and Halck (2000), minimax TD-learning has in fact been applied to Campaign. The state evaluator function trained in that study had a structure identical to the one we use for truncated sampling. The *Geq* performance of the agent using minimax TD was as high as 0.41, which indicates that minimax TD works better than our lagging anchor algorithm.

This should not surprise us. Firstly, minimax TD is applicable to Markov games only, and algorithms that utilize the specific structure of a problem will normally work better than more general algorithms. Secondly, the task of evaluating game states is much simpler than that of producing probability distributions. The state values are monotone with respect to all four state dimensions, which is beneficial for the neural net, while there is no similar structure in the optimal probability distributions. The minimax TD-agent therefore bypasses the hardest part of the problem by applying a dedicated matrix game solver for the construction of probability distributions. We have also tested the state evaluator resulting from the lagging anchor experiment, and found that it too produces a *Geq* slightly higher than 0.40 when combined with a matrix game solver.

When comparing the lagging anchor algorithm to minimax TD-learning, the main advantage of the former is that it can be applied to a wider class of games (such as our simplified poker game). The lagging anchor algorithm also has the advantage of producing agents that calculate their probability distributions directly. Agents using minimax TD-learning must go through the steps of assembling a local payoff matrix (with dimensions up to 21×21 in Campaign) and solving the corresponding matrix game to calculate action probabilities. Agents that are designed for training with the lagging anchor algorithm therefore tend to play faster than those designed for minimax TD-learning.

7. Conclusion

We have defined the lagging anchor algorithm, which learns to play extensive-form two-player zero-sum games of imperfect information by reinforcement learning. We have given a convergence result for a non-trivial set of matrix games, and extended this to non-linear and incomplete strategy representations. The algorithm has worked well for neural nets playing a simplified poker game with infinite sets of pure strategies. We have also tested the algorithm with neural nets playing Campaign, an air campaign game of significant complexity, in combination with temporal difference learning. In this case, the algorithm produced better results than those achieved by training identical nets to imitate the mathematical solution of the game.

To our knowledge, our algorithm is the first reinforcement learning algorithm that applies to general two-player zero-sum games, including games with imperfect information. Future experiments may show whether the algorithm works well on even more complex games than we have tested, like full-scale two-player hold'em poker. Another direction for future research may be the algorithm's behavior in n-player and non-zero-sum games.

Appendix

In this appendix, we give proofs for the propositions that we have stated for the basic and lagging anchor algorithms used for matrix games. We introduce lemmas when they are needed for subsequent proofs of propositions.

Proof of Proposition 1: Because $V \times W$ is a compact and convex set in $\mathbf{R}^k$ for some k, it is topologically equivalent (homeomorphic) to a closed unit sphere S^j for some j

(Busemann, 1958). The mapping (1) is continuous, and the result follows from Brouwer's fixed point theorem. □

Lemma 1. *If $J \subset I \subset \{1, 2, \ldots, n\}$ and $\mathbf{v} \in \mathbf{R}^n$, then $Q_J(\mathbf{v}) = Q_J(Q_I(\mathbf{v}))$.*

Proof: For $i \notin J$, both sides of the equation are 0. The following vector equations are valid if we restrict the indices to J:

$$Q_J(\mathbf{v}) = \mathbf{v} + a\mathbf{1}_n$$
$$Q_I(\mathbf{v}) = \mathbf{v} + b\mathbf{1}_n$$
$$Q_J(Q_I(\mathbf{v})) = Q_I(\mathbf{v}) + c\mathbf{1}_n = \mathbf{v} + (b + c)\mathbf{1}_n$$

Because Q_J produces vectors that sum to 1 over J, it follows that $a = b + c$. □

Proof of Proposition 2: Observe that the algorithm does terminate, because the set I is reduced for each iteration. We apply the standard theory of constrained minimization (Luenberger, 1984). The set V is defined by one equality constraint with gradient vector $\mathbf{1}_n$ ($\mathbf{v}^T\mathbf{1}_n = 1$) and n inequality constraints ($v_i \geq 0, i = 1, 2, \ldots, n$) with gradient vectors $-\mathbf{e}_i$ (the i'th unit vector). At most $n - 1$ inequality constraints can be active for a given $\mathbf{v}$, and $\mathbf{1}_n$ is linearly independent of any proper subset of the n unit vectors. Therefore all points of V are regular. As the function $f(\mathbf{x}) = \frac{1}{2}\|\mathbf{v} - \mathbf{x}\|^2$, which is to be minimized, is convex, the first-order condition of Kuhn and Tucker is sufficient. As $\nabla f(\mathbf{x}) = \mathbf{x} - \mathbf{v}$, that condition is given by:

$$C_V(\mathbf{v}) - \mathbf{v} = \lambda\mathbf{1}_n + \boldsymbol{\mu} \tag{6}$$

where $\boldsymbol{\mu} \geq \mathbf{0}$ and $\boldsymbol{\mu}^T C_V(\mathbf{v}) = 0$. As $\boldsymbol{\mu}$ and $C_V(\mathbf{v})$ are non-negative, the last condition states that they have disjoint support. From Lemma 1 we know that the algorithm terminates with $\mathbf{c} = Q_I(\mathbf{v})$. Observe from the algorithm that we can write $Q_I(\mathbf{v}) - \mathbf{v} = \lambda\mathbf{1}_n + \boldsymbol{\mu}$, which is equivalent to (6). By construction, $\boldsymbol{\mu}$ and $Q_I(\mathbf{v})$ have disjoint support, so we only need to show that $\boldsymbol{\mu}$ is non-negative. Each iteration makes an update $\mathbf{c} \leftarrow \mathbf{c} + a\mathbf{1}_n + \mathbf{b}$, and $\boldsymbol{\mu}$ is the sum of these $\mathbf{b}$-vectors. In the first iteration, $\mathbf{b} = \mathbf{0}$. Thereafter $\mathbf{1}^T\mathbf{c} = 1$, which implies that $\sum_{i \in I} c_i \leq 1$, so that $\mathbf{b} \geq \mathbf{0}$. □

Proof of Proposition 3: Define $\mathbf{A} = \left[\begin{smallmatrix} \mathbf{0} & \mathbf{M} \\ -\mathbf{M}^T & \mathbf{0} \end{smallmatrix}\right]$. Let $U_k \subset \mathbf{R}^k$ be the linear subspace where the components sum to 0: $U_k = \{\mathbf{x} \in \mathbf{R}^k : \mathbf{x}^T\mathbf{1}_k = 0\}$, and let $\mathbf{P}_k \in \mathbf{R}^{k \times k}$ be the orthogonal projection down to U_k : $\mathbf{P}_k = \mathbf{I} - \frac{1}{k}\mathbf{1}_k\mathbf{1}_k^T$. Let $U = U_n \times U_m$ with projection matrix $\mathbf{P} = \left[\begin{smallmatrix} \mathbf{P}_n & \mathbf{0} \\ \mathbf{0} & \mathbf{P}_m \end{smallmatrix}\right]$. If $\mathbf{x}^{k+1}$ is in the interior, the update rule (3) becomes $\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha\mathbf{P}\mathbf{A}\mathbf{x}^k$. We must therefore show $(\mathbf{P}\mathbf{A}\mathbf{x})^T(\mathbf{x} - \mathbf{x}^*) = 0$. Note that $\mathbf{P}\mathbf{A}\mathbf{x}^* = 0$, because $\mathbf{x}^*$ is an interior point. Define $\mathbf{y}$ as the residual between the variable state and the solution: $\mathbf{y} = \mathbf{x} - \mathbf{x}^*$. This gives: $(\mathbf{P}\mathbf{A}\mathbf{x})^T(\mathbf{x} - \mathbf{x}^*) = (\mathbf{P}\mathbf{A}(\mathbf{y} + \mathbf{x}^*))^T\mathbf{y} = \mathbf{y}^T\mathbf{A}^T\mathbf{P}^T\mathbf{y} = \mathbf{y}^T\mathbf{A}^T\mathbf{P}\mathbf{y}$. The last equality follows because all projections are symmetric. As $\mathbf{y} \in U$, the subspace to which $\mathbf{P}$ projects, we get $\mathbf{P}\mathbf{y} = \mathbf{y}$. Therefore $(\mathbf{P}\mathbf{A}\mathbf{x})^T(\mathbf{x} - \mathbf{x}^*) = \mathbf{y}^T\mathbf{A}^T\mathbf{y}$. Because $\mathbf{A}$ is skew-symmetric, we have: $\mathbf{y}^T\mathbf{A}^T\mathbf{y} = (\mathbf{y}^T\mathbf{A}^T\mathbf{y})^T = \mathbf{y}^T\mathbf{A}\mathbf{y} = -\mathbf{y}^T\mathbf{A}^T\mathbf{y}$. So the real number $\mathbf{y}^T\mathbf{A}^T\mathbf{y}$ is equal to its negation, which implies that it is equal to 0. □

Lemma 2. *Let* $\mathbf{B} \in \mathbf{R}^{n\times n}$, *and let* $\mathbf{x}(t)$ *solve the equation* $\mathbf{x} = \mathbf{B}\mathbf{x}$. *Let* $\mathbf{b} \in \mathbf{R}^n$. *If* $\lim_{t\to\infty} \mathbf{x}(t)^T\mathbf{b} = 0$, *then* $\mathbf{x}(t)^T\mathbf{b}$ *and any derivative* $(\frac{d}{dt})^n\mathbf{x}(t)^T\mathbf{b}$ *decay exponentially with* t.

Proof: In general $\mathbf{x}(t)^T\mathbf{b} = \sum_i c_i P_i(t)\exp(l_i t)$, where $P_i(t)$ are real polynomials, c_i, l_i are constants, possibly complex (Strang, 1980). For this sum to converge to zero, convergence must be exponential, which implies that $\mathbf{x}(t)^T\mathbf{b}$ and any derivative $(\frac{d}{dt})^n\ \mathbf{x}(t)^T\mathbf{b}$ decay exponentially. □

Lemma 3. *Let* $(\mathbf{v}^*, \mathbf{w}^*) \in V \times W$ *be a fully mixed game solution, and let* $d > 0$ *be the Euclidean distance from* $(\mathbf{v}^*, \mathbf{w}^*)$ *to the boundary of* $V \times W$. *Assume that the distance between* $(\mathbf{v}^0, \mathbf{w}^0)$ *and* $(\mathbf{v}^*, \mathbf{w}^*)$ *is less than* d. *Let* $\mathbf{x}(t)$ *solve* $\dot{\mathbf{x}}(t) = \mathbf{B}\mathbf{x}(t)$ *with*

$$\mathbf{x}(0)^T = \begin{bmatrix}\mathbf{v}^{0^T} & \mathbf{w}^{0^T} & \mathbf{v}^{0^T} & \mathbf{w}^{0^T}\end{bmatrix},$$

where

$$\mathbf{B} = \begin{bmatrix}\mathbf{PA} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix} + \eta\begin{bmatrix}-\mathbf{I}_{n+m} & \mathbf{I}_{n+m}\\ \mathbf{I}_{n+m} & -\mathbf{I}_{n+m}\end{bmatrix}$$

and $\mathbf{P}$ *and* $\mathbf{A}$ *are defined as in the proof of Proposition* 3. *Then* $\mathbf{x}(t)$ *never hits the boundary, and converges exponentially toward some solution point* $\bar{\mathbf{x}} \in V \times W$.

Proof: We decompose $\mathbf{x}(t) = \binom{\mathbf{y}(t)}{\bar{\mathbf{y}}(t)}$ so that $\mathbf{y}(t) = \binom{\mathbf{v}(t)}{\mathbf{w}(t)}$ and $\bar{y}(t) = \binom{\bar{\mathbf{v}}(t)}{\bar{\mathbf{w}}(t)}$. We assume that the process $\mathbf{x}$ never hits the boundary of $V \times W \times V \times W$, and return to prove this below. Now define $\mathbf{z}(t) = \mathbf{y}(t) - \mathbf{y}^*$ and $\bar{\mathbf{z}}(t) = \bar{\mathbf{y}}(t) - \mathbf{y}^*$, where $\mathbf{y}^* = \binom{\mathbf{v}^*}{\mathbf{w}^*}$. Let $r(t) = \mathbf{z}(t)^T\mathbf{z}(t) + \bar{\mathbf{z}}(t)^T\bar{\mathbf{z}}(t)$. As $\mathbf{x}$ is assumed to stay in the interior, we have:

$$\dot{\mathbf{y}} = \mathbf{PAy} + \eta(\bar{\mathbf{y}} - \mathbf{y})$$
$$\dot{\bar{\mathbf{y}}} = \eta(\mathbf{y} - \bar{\mathbf{y}})$$

Because $\mathbf{PAy}^* = 0$, substituting for $\mathbf{z}$ gives:

$$\dot{\mathbf{z}} = \mathbf{PAz} + \eta(\bar{\mathbf{z}} - \mathbf{z})$$
$$\dot{\bar{\mathbf{z}}} = \eta(\mathbf{z} - \bar{\mathbf{z}})$$

From the analysis of our basic algorithm, we know that $\mathbf{PAz}$ is orthogonal to $\mathbf{z}$, and therefore does not contribute to $\dot{r}(t)$, so we have:

$$dr/dt = \eta(\mathbf{z}^T(\bar{\mathbf{z}} - \mathbf{z}) + \bar{\mathbf{z}}^T(\mathbf{z} - \mathbf{z})) = -\eta\|\mathbf{z} - \bar{\mathbf{z}}\|^2 \qquad (7)$$

This shows that r is non-increasing and positive, and therefore converges to some limit $R = \lim_{t\to\infty} r(t)$. This implies that dr/dt converges to 0. Equation (7) therefore gives $\lim_{t\to\infty} \mathbf{z} - \bar{\mathbf{z}} = \mathbf{0}$. From this, our differential equations give that $\dot{\bar{\mathbf{z}}} = \eta(\mathbf{z} - \bar{\mathbf{z}})$ converges to zero. Lemma 2 gives that $\mathbf{z} - \bar{\mathbf{z}}$, and therefore $\dot{\bar{\mathbf{z}}}$ decay exponentially. This in turn implies that $\mathbf{z}$ and $\bar{\mathbf{z}}$ converge exponentially.

Now we return to show that all components of $\mathbf{y}$ and $\bar{\mathbf{y}}$ remain positive. By symmetry $\mathbf{z}^T\mathbf{z} = \bar{\mathbf{z}}^T\bar{\mathbf{z}}$ for all t, provided this holds for $t = 0$. The result then follows from the fact that $\mathbf{z}^T\mathbf{z}$ and $\bar{\mathbf{z}}^T\bar{\mathbf{z}}$ are decreasing. □

Proof of Proposition 4: In the interior, the learning rule (4) gives

$$\mathbf{x}^{k+1} = (\mathbf{I} + \alpha\mathbf{B})\mathbf{x}^k, \tag{8}$$

where $\mathbf{B}$ is defined as in Lemma 3. Because $\mathbf{x}(t)$ in Lemma 3 converges exponentially, all eigenvalues of $\mathbf{B}$ are either zero, or have negative real part. (Note that we regard $\mathbf{B}$ as a linear operator on the space $U \times U$, not the entire $\mathbf{R}^{(2n+2m)\times(2n+2m)}$.) Let λ be an eigenvalue of **B:** $\mathbf{B}\mathbf{x} = \lambda\mathbf{x}$.

Multiplying by α and adding $\mathbf{x}$ gives the equivalent expression $(\mathbf{I} + \alpha\mathbf{B})\mathbf{x} = (1 + \alpha\lambda)\mathbf{x}$, which shows that eigenvalues of $\mathbf{D} = \mathbf{I} + \alpha\mathbf{B}$ are on the form $(1 + \alpha\lambda)$. By making α sufficiently small, all eigenvalues of $\mathbf{D}$ that correspond to non-zero eigenvalues of $\mathbf{B}$ get length less than unity. The null-space N of $\mathbf{B}$ is mapped onto the fixed point space of $\mathbf{D}$, regardless of α. Note that for small α, $\mathbf{D}$ is invertible. By choosing α sufficiently small, all eigenvalues of $\mathbf{D}$ will be non-zero and have length less than unity, except for those being equal to 1. Let $N\perp$ be the subspace orthogonal to N, and let $\bar{\mathbf{D}}$ be the restriction of $\mathbf{D}$ to $N\perp$. Then $\bar{\mathbf{D}}$ maps $N\perp$ onto $N\perp$, because N consists of fixed points for $\mathbf{D}$, and $\mathbf{D}$ is invertible. Eigenvalues of $\bar{\mathbf{D}}$ are eigenvalues of $\mathbf{D}$, and are therefore either 1 or have length less than 1. By construction $N\perp$ contains no eigenvectors with eigenvalues **1**, so all eigenvalues of $\bar{\mathbf{D}}$ have length less than 1. This implies that $\bar{\mathbf{D}}^n$ decays exponentially with n. From (8) we get $\mathbf{x}^n = \mathbf{D}^n\mathbf{x}^0$. Let $\mathbf{x}^0 - \mathbf{x}_n + \mathbf{x}_{N\perp}$, where $\mathbf{x}_N \in N$ and $\mathbf{x}_{N\perp} \in N\perp$. Then $\mathbf{D}^n\mathbf{x}^0 = \mathbf{D}^n(\mathbf{x}_N + \mathbf{x}_{N\perp}) = \mathbf{x}_N + \mathbf{D}^n\mathbf{x}_{N\perp} = \mathbf{x}_N + \bar{\mathbf{D}}^n\mathbf{x}_{N\perp}$, which converges exponentially toward $\mathbf{x}_N$ when $n \to \infty$. □

Proof of Proposition 5: In the interior of $\tilde{V} \times \tilde{W} \times \tilde{V} \times \tilde{W}$ the differential version of the learning rule (2) is:

$$\begin{bmatrix} \dot{v} \\ \dot{w} \\ \dot{\bar{v}} \\ \dot{\bar{w}} \end{bmatrix} = \begin{bmatrix} \nabla_V E(v, w) + \eta(\bar{v} - v) \\ -\nabla_W E(v, w) + \eta(\bar{w} - w) \\ \eta(v - \bar{v}) \\ \eta(w - \bar{w}) \end{bmatrix}. \tag{9}$$

We will construct the linearization of (9) around the fixed point (v^*, w^*, v^*, w^*). This is a standard technique for proving stability of equilibrium points of non-linear systems of differential equations, see e.g. Luenberger, (1979). Let $\mathbf{x}$ be the linearized process: $\mathbf{x}(t) \approx$

$\left[\begin{smallmatrix} v(t)-v^* \\ w(t)-w^* \\ \bar{v}(t)-v^* \\ \bar{w}(t)-w^* \end{smallmatrix}\right]$. The linearization matrix $\mathbf{D}$ is given by

$$\mathbf{D} = \begin{bmatrix} \mathbf{M}^{vv} - \eta\mathbf{I}_k & \mathbf{M}^{vw} & \eta\mathbf{I}_k & \mathbf{0} \\ -\mathbf{M}^{vw^T} & \mathbf{M}^{ww} - \eta\mathbf{I}_k & \mathbf{0} & \eta\mathbf{I}_k \\ \eta\mathbf{I}_k & \mathbf{0} & -\eta\mathbf{I}_k & \mathbf{0} \\ \mathbf{0} & \eta\mathbf{I}_k & & \mathbf{0} - \eta\mathbf{I}_k \end{bmatrix}.$$

From the assumptions in the proposition, the linearization matrix simplifies to

$$\mathbf{D} = \begin{bmatrix} \mathbf{C} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} + \eta \begin{bmatrix} -\mathbf{I}_{2k} & \mathbf{I}_{2k} \\ \mathbf{I}_{2k} & -\mathbf{I}_{2k} \end{bmatrix} \text{ with } \mathbf{C} = \begin{bmatrix} \mathbf{0} & \mathbf{M}^{vw} \\ -\mathbf{M}^{vw^T} & \mathbf{0} \end{bmatrix}.$$

The linearized system is given by $\dot{\mathbf{x}} = \mathbf{D}\mathbf{x}$. Note that $\mathbf{D}$ is invertible, and $\mathbf{C}$ is skew-symmetric. By the arguments of the proof of Lemma 3, the process $\mathbf{x}(t)$ converges exponentially toward some point. Because $\mathbf{D}$ is invertible, the limit point must be $\mathbf{0}$. This in turn implies that all the eigenvalues of $\mathbf{D}$ must have negative real part, so that the non-linear process of (9) converges exponentially toward (v^*, w^*, v^*, w^*) in a neighborhood of that point. The learning rule (2) is an explicit Euler approximation to (9). It follows that the solution of (2) also converges exponentially toward (v^*, w^*, v^*, w^*) in a neighborhood of the same point. □

Proof of Corollary 1: The payoff function is given by $E(v, w) = \varphi_V(v)^T\tilde{\mathbf{M}}\varphi_W(w)$. Because $(\varphi_V(v^*), \varphi_W(w^*)) \in \tilde{V} \times \tilde{W}$ is a fully mixed solution of the matrix game $\tilde{\mathbf{M}}$, $E(v, w)$, is equal to the game's value as long as one of the players apply the solution. Therefore $\mathbf{M}^{vv} = \mathbf{M}^{ww} = \mathbf{0}$, and Proposition 5 applies. □

Acknowledgments

The author is grateful for the help and support given by his colleague Ole Martin Halck and the comments received from the anonymous referees. This article would not have appeared in its present form without their help.

References

Bakken, B. T., & Dahl, F. A. (1998). Experimental studies of neural net training and human learning in a military air campaign game. *Proceedings of the Seventh Conference on Computer Generated Forces and Behavioral Representation*, University of Central Florida, Orlando, Florida, Institute for Simulation and Training, pp. 263–274.

Berkovitz, L. D. (1975). The tactical air game: A multimove game with mixed strategy solution. In J. D. Grote (Ed.), *The theory and application of differential games* (pp. 169–177).

Busemann, H. (1958). *Convex surfaces*. Interscience tracts in pure and applied mathematics, No. 6. New York: Interscience Publishers.

Conlisk, J. (1993a). Adaptation in games—2 solutions to the Crawford puzzle. *Journal of Economic Behavior and Organizations, 22*, 25–50.

Conlisk, J. (1993b). Adaptive tactics in games—Further solutions to the Crawford puzzle. *Journal of Economic Behavior and Organizations, 22*, 51–68.

Crawford, V. P. (1974). Learning the optimal strategy in a zero-sum game. *Econometrica, 42*, 885–891.

Dahl, F. A., & Halck, O. M. (1998). Three games designed for the study of human and automated decision making. Definitions and properties of the games campaign, Operation lucid and operation opaque. FFI/Rapport-98/02799, Norwegian Defence Research Establishment (FFI), Kjeller, Norway.

Dahl, F. A., & Halck, O. M. (2000). Minimax TD-learning with neural nets in a Markov game. In R. Lopez de Mantaras & E. Plaza (Eds.), *ECML 2000. Proceedings of the 11th European Conference on Machine Learning*, Lecture Notes in Computer Science (Vol. 1810). Berlin: Springer-Verlag.

Dahl, F. A., Halck, O. M., & Braathen, S. (2000). *Machine learning in the game of Campaign*. FFI/Rapport-2000/04400, Norwegian Defence Research Establishment (FFI), Kjeller, Norway.

Erev, I., & Roth, A. E. (1998). Predicting how people play games: Reinforcement learning in experimental games with unique, mixed strategy equilibria. *The American economic review, 88*, 848–881.

Fudenberg, D., & Levine, D. K. (1998). *The theory of learning in games*. Cambridge: MIT Press.

Halck, O. M., & Dahl, F. A. (1999). On classification of games and evaluation of players—with some sweeping generalizations about the literature. In: J. Fürnkranz, & M. Kubat (Eds.), *Proceedings of the ICML-99 Workshop on Machine Learning in Game Playing*. Ljubljana, Slovenia: Jozef Stefan Institute.

Harmon, M. E., Baird, L. C., & Klopf, A. H. (1995). Reinforcement learning applied to a differential game. *Adaptive behavior* (Vol. 0004). Cambridge, MA: MIT Press.

Hassoun, M. H. (1995). *Fundamentals of artificial neural networks*. Cambridge, MA: MIT Press.

Koller, D., Megiddo, N., & von Stengel, B. (1996). Efficient solutions of extensive two-person games. *Games and Economic Behavior, 14*, 247–259.

Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the 11th International Conference on Machine Learning* (pp. 157–163). New Brunswick: Morgan Kaufmann.

Luce, R. D., & Raiffa, H. (1957). *Games and decisions*. New York: Wiley.

Luenberger D. G. (1979). *Introduction to dynamic systems. Theory, models, & applications*. New York: Wiley.

Luenberger, D. G. (1984). *Linear and nonlinear programming*. Reading, MA: Addison-Wesley.

Michie, D. (1966). Game-playing and game-learning automata. In L. Fox (Ed.), *Advances in programming and non-numerical computation* (pp. 183–200). New York, Pergamon.

Padberg, M. (1995). *Linear optimization and extensions*. Berlin: Springer-Verlag.

Papadimitriou, C. H. (1994). *Computational complexity*. Reading, MA: Addison Wesley.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM J Res. Develop., 3*, 210–229.

Schaeffer, J., Billings, D., Peña, L., & Szafron, D. (1999). Learning to play strong poker. In: J. Furnkranz, & M. Kubat (Eds.), *Proceedings of the ICML-99 Workshop on Machine Learning in Game Playing*, Ljubljana, Slovenia: Jozef Stefan Institute.

Selten, R. (1991). Anticipatory learning in two-person games. In R. Selten (Ed.), *Game equilibrium models* (Vol. I: Evolution and game dynamics). Berlin:Springer-Verlag.

Strang, G. (1980). *Linear algebra and its applications*. London: Harcourt Brace.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning, 3*, 9–44.

Szepesvari, C., & Littman, M. L. (1999). A unified analysis of value-function-based reinforcement-learning algorithms. *Neural Computation, 11*, 2017–2060.

Tesauro, G. J. (1992). Practical issues in temporal difference learning. *Machine Learning, 8*, 257–277.

Tesauro, G. J., & Sejnowski, T. J. (1989). A parallel network that learns to play backgammon. *Artificial Intelligence, 39*, 357–390.

von Neumann, J., & Morgenstern, O. (1953). *Theory of games and economic behavior*, 3rd ed. New York: Wiley.

Watkins, C. J. C. H. (1989). Learning from delayed rewards. PhD thesis, Psychology Department, Cambridge University, Cambridge, UK.

Weibull, J. (1995). *Evolutionary game theory*, Cambridge: MIT Press.

Received April 22, 1999
Revised March 15, 2001
Accepted May 16, 2001
Final manuscript June 28, 2001