Efficient Reinforcement Learning in Adversarial Games

Ioannis E. Skoulakis and Michail G. Lagoudakis
Department of Electronic and Computer Engineering
Technical University of Crete
Chania 73100 Greece
Email: {iskoulakis, lagoudakis}@intelligence.tuc.gr

***Abstract*—The ability of learning is critical for agents designed to compete in a variety of two-player, turn-taking, tactical adversarial games, such as Backgammon, Othello/Reversi, Chess, Hex, etc. The mainstream approach to learning in such games consists of updating some state evaluation function usually in a Temporal Difference (TD) sense either under the MiniMax optimality criterion or under optimization against a specific opponent. However, this approach is limited by several factors: (a) updates to the evaluation function are incremental, (b) stored samples from past games cannot be utilized, and (c) the quality of each update depends on the current evaluation function due to bootstrapping. In this paper, we present a learning approach based on the Least-Squares Policy Iteration (LSPI) algorithm that overcomes these limitations by focusing on learning a state-action evaluation function. The key advantage of the proposed approach is that the agent can make batch updates to the evaluation function with any collection of samples, can utilize samples from past games, and can make updates that do not depend on the current evaluation function since there is no bootstrapping. We demonstrate the efficiency of the LSPI agent over the TD agent in the classical board game of Othello/Reversi.**

I. Introduction

Computer games always had a precious place in the hall of fame of Artificial Intelligence and Machine Learning. If successful game playing requires a significant level of intelligence, the design of autonomous agents able to play games competitively with humans and improve their performance over time (learning) is certainly an important challenge for researchers. As early as 1949, Claude Shannon presented strategies that enable an agent to play Chess [1]. Only a decade later, Arthur Samuel presented an agent able to learn better strategies in Checkers by self-play [2]. Nowadays, numerous game-playing learning agents exhibit superior performance against human opponents in various popular adversarial games, such as Chess, Backgammon, Othello/Reversi, etc.

The ability of learning is critical for an agent to be competitive in adversarial games in the long run. The mainstream approach to learning in such games consists of updating some approximate state evaluation function in a Temporal Difference (TD) sense. In this paper, we first argue that this approach is limited by several factors: (a) updates to the evaluation function are incremental, (b) stored samples from past games cannot be utilized for learning, and (c) the quality of each update depends on the current evaluation function due to bootstrapping. We, then, advocate an approach that overcomes these limitations by focusing on learning a state-action evaluation function using the Least-Squares Policy Iteration (LSPI) algorithm. The key advantage of the proposed approach is that the agent can make batch updates to the evaluation function with any collection of samples, can utilize samples from past games, and can make updates that do not depend on the current evaluation function since there is no bootstrapping. We demonstrate the efficiency of the LSPI agent over the TD agent in the classical board game of Othello/Reversi.

This paper is organized as follows: Section II provides background information on reinforcement learning, Section III describes the main design principles for game-playing agents, Section IV describes the TD-approach to learning in adversarial games, Section V discusses the limitations of the TD-approach, Section VI presents our approach to learning in adversarial games, Section VII describes our modeling approach to Othello/Reversi, Section VIII summarizes our experimental results, Section IX discusses related work, and, finally, Section X concludes.

II. Reinforcement Learning

A Markov Decision Process (MDP) [3] is a modeling framework for sequential decision making under uncertainty. An MDP is described as $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma, \mathcal{D})$, where $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ is the state space; $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ is the action space; $\mathcal{P}(s'|s,a)$ is a Markovian transition model; $\mathcal{R}(s,a)$ is a Markovian reward model; $\gamma \in (0,1]$ is the discount factor for future rewards; $\mathcal{D}$ is the initial state distribution. A (deterministic) policy π is a mapping from states to actions; $\pi(s)$ denotes the action chosen by policy π in state s. The optimization objective in an MDP is to find a policy that maximizes the long-term return:

$$E_{s\sim\mathcal{D};\, a_t\sim\pi;\, s_t\sim\mathcal{P};\, r_t\sim\mathcal{R}}\left(\sum_{t=0}^{\infty}\gamma^t r_t \mid s_0 = s\right)$$

The state value function $V^\pi(s)$ for a policy π indicates the return when following policy π starting in state s. The state-action value function $Q^\pi(s,a)$ for a policy π indicates the return when taking action a in state s and following policy π thereafter. The state or state-action value function of any policy π can be computed by solving the linear system of Bellman equations [4]. An improved policy π' over π can be

inferred by maximization over one-step look-ahead returns, formed using either the state or the state-action value function of π. An optimal policy π^* yields the optimal return. Given the full model of an MDP, several MDP solution methods are available for deriving an optimal policy (value iteration, policy iteration, linear programming) [5].

In many real-world sequential decision domains, MDP solution methods cannot be applied either because $\mathcal{P}$ and $\mathcal{R}$ are unknown or because the state space is enormous or infinite. Decision making in such cases is formulated as a reinforcement learning problem [6], [7]; a good or even optimal policy must be learned from samples of interaction with the process. At each step of interaction, the learner observes the current state s, chooses an action a, and observes the resulting next state s' and the reward received r, thus learning is based on (s, a, r, s') samples. In non-trivial state spaces, value functions have to be approximated. A common choice for value function approximation is a linear architecture, that is a weighted combination of basis functions (features):

$$\widehat{V}^{\pi}(s) = \sum_{j=1}^{k} \psi_j(s) w_j^{\pi} = \psi(s)^{\top} w^{\pi}$$

$$\widehat{Q}^{\pi}(s,a) = \sum_{j=1}^{l} \phi_j(s,a) w_j^{\pi} = \phi(s,a)^{\top} w^{\pi}$$

where w_j^{π} are the weights (adjustable parameters) of the architecture, ψ_j are the basis functions for approximating V^{π}, and ϕ_j are the basis functions for approximating Q^{π}.

Least-Squares Policy Iteration (LSPI) [8] is an efficient reinforcement learning algorithm that combines policy iteration with value function approximation. LSPI learns in a batch manner by processing multiple times a set of (s, a, r, s') samples collected arbitrarily from the process. In particular, LSPI iteratively learns a sequence of improving policies. LSPI is summarized in Algorithm 1. Notice that policies in LSPI are not represented explicitly, but only implicitly through the weights of the previous value function and maximization over the corresponding state-action values. LSPI offers a non-divergence guarantee and exhibits excellent sample efficiency.

Algorithm 1 Least-Squares Policy Iteration (LSPI).

Input: samples D, basis functions ϕ, discount factor γ, tolerance ϵ
Output: weights w for inferring the learned policy π

$w' \leftarrow \mathbf{0}$
repeat
  $w \leftarrow w'$, $\mathbf{A} \leftarrow \mathbf{0}$, $b \leftarrow \mathbf{0}$
  for each (s, a, r, s') in D **do**
    $a' = \arg\max_{a'' \in \mathcal{A}} \left\{ \phi(s', a'')^{\top} w \right\}$
    $\mathbf{A} \leftarrow \mathbf{A} + \phi(s,a)\Big(\phi(s,a) - \gamma\phi(s',a')\Big)^{\top}$
    $b \leftarrow b + \phi(s,a) r$
  end for
  $w' \leftarrow \mathbf{A}^{-1} b$
until $\big(\|w - w'\| < \epsilon\big)$

III. Agents for Adversarial Games

An adversarial game is similar to an MDP with one significant difference: there is a second agent making decisions; these decisions are made in alternating turns and the two agents have conflicting objectives. An adversarial game can be described as $(\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{P}, \mathcal{R}, \gamma, \mathcal{D})$, where $\mathcal{S}$, $\mathcal{A}$, γ, and $\mathcal{D}$ are defined as in MDPs; $\mathcal{O} = \{o_1, o_2, \ldots, o_m\}$ is the action space of the opponent; $\mathcal{P}(s'|s, a, o)$ is a Markovian transition model; $\mathcal{R}(s, a, o)$ is a Markovian reward model. The definition adopts a view from the agent's side, meaning that the agent plays first and the opponent follows and states are considered only when it is the agent's turn to play. The single reward model implies a zero-sum game, whereby whatever is gained by the agent is lost by the opponent, therefore the goal of the agent (Max) is to maximize return, whereas the goal of the opponent (Min) is to minimize return.

The MiniMax objective criterion is commonly used in two-player, adversarial, zero-sum games. The Max player is trying to select its best action over all possible choices of the Min player in the next and future turns. The game tree represents all possible paths of action sequences of the two players. Each node in this tree corresponds to a state of the game; the same state may appear at several nodes within the tree. The MiniMax search algorithm [9] expands the game tree starting from any state of the game as root and derives the best action at the root state. In order to prune unnecessary parts of the game tree, Alpha-Beta Pruning [10] can be used to eliminate branches of the tree not contributing to the MiniMax value at the root and, therefore, to the final move choice.

The game outcome is determined only at terminal nodes. To compute the true MiniMax value at the root of a game tree, one would have to expand the tree all the way to the leaves (terminal nodes). In practice, this is infeasible in reasonable time, therefore nodes at some cut-off depth are evaluated using a heuristic evaluation function that estimates their utility; these utilities are backed up the tree, as if they were values coming from terminal nodes. The cut-off depth can be fixed or variable to allow for deeper or shallower searches.

IV. Learning in Adversarial Games

The evaluation of a game state s is done by an evaluation function $V(s)$, which also implicitly determines the agent's strategy. Given the typically huge state space of most games, such an evaluation function must be approximated, commonly using a linear approximation architecture. The weights of the evaluation function can be set empirically by a human expert, however it is desirable to learn a good set of weights for any given set of features automatically. An approach to learning good weights is to require that the evaluation of a state s matches the evaluation of a successor (terminal or non-terminal) state s' from which the value will be backed up to s. This observation gave rise to the extensive use of reinforcement learning methods for learning the weights

of evaluation functions in games. These methods rely on samples of the form (s, a, o, r, s'), typically obtained from actual games, indicating a transition from state s to state s' with intermediate reward r, after the agent and the opponent took their moves a and o in turn. In most games, the reward r is non-zero only upon a transition to a terminal state.

The agent must learn to play well against any possible opponent and a safe, but conservative, option is to optimize its own strategy against an "optimal" opponent. In this case, the agent has to consider good action choices not only for himself, but also for the opponent. The agent uses MiniMax search with cut-off at a certain depth and the current evaluation function to identify both its "best" action a in state s and the "best" opponent response o to its own choice a. The resulting state s' after taking these "optimal" moves will be recorded as the next state of s with the corresponding reward r for this transition. Note that the elements o, r, and s' of a sample (s, a, o, r, s') are not necessarily observed, as the game progresses.

The most popular approach of reinforcement learning used in games is Temporal Difference (TD) learning [6], which updates the weights w of the evaluation function for each sample (s, a, o, r, s') encountered as follows:

$$w_j \leftarrow w_j + \alpha\psi_j(s)\Big(r + \gamma\psi(s')^\top w - \psi(s)^\top w\Big)$$

where ψ are the basis functions of the linear approximation. The quantity in parenthesis is the temporal difference (value difference between temporally distant estimates of the value in state s). The sign and magnitude of this quantity guides the gradient descent update to the weights. The learning rate α is a parameter in $(0, 1]$ that adjusts the step size of the update, while the discount factor γ determines how the value is discounted at each step. In most adversarial games, there is no loss of value, therefore $\gamma = 1$. Note that the agent and the opponent actions a, o are not exploited in any way during the TD updates.

A variation of TD, which attempts to make updates that carry cumulative information from multiple samples is known as TD(λ), where λ is a parameter that ranges from 0 to 1. TD(0) is equivalent to the version described above and considers only the information contained in the current sample. On the other hand, TD(1) considers the cumulative information from multiple samples (trace of game states). Intermediate values of λ provide a weighted continuum between these two extremes. More precisely, given a sequence of samples $(s_i, a_i, o_i, r_i, s'_i)$, $i = 1, \ldots, N$, encountered during a search in a game tree, TD(λ) makes the following update:

$$w_j \leftarrow w_j + \alpha \sum_{i=1}^{N-1} \psi_j(s_i) \sum_{t=i}^{N-1} \lambda^{i-t}\Big(r_t + \gamma\psi(s'_t)^\top w - \psi(s_t)^\top w\Big)$$

V. Rethinking TD Learning in Games

Our experience with TD learning in games, revealed several weaknesses which appear to be inherent in using state value functions and incremental updates. TD-style updates to the value function are by default incremental, which means that each sample incurs a change to the value function whose magnitude depends on the current values of the learning rate and the temporal difference; after the update, that sample is typically discarded and is never reused. Given that the game itself is stationary, it is understandable that each sample carries information which may be useful at different phases of learning to improve performance. As learning progresses, the same sample could contribute an update that backs up a better estimate of the value of the next state. Additionally, under the MiniMax criterion, a better value function would even lead to a more accurate estimation of the players' "optimal" actions and therefore an even better estimate of the resulting next state after taking these actions. Nevertheless, this kind of reuse is not possible in TD-learning, especially in its TD(λ) variant with $\lambda > 0$, because samples must be obtained online, that is by selecting actions using the currently learned policy, to form correct updates. This problem of inability to use stored samples is pronounced by the fact that the final learned value function strongly depends on the order in which samples are presented. The same sample may have a significant or a negligible effect depending on when it appears during the learning period.

One may be able to remedy the problem of incremental and ordered updates by employing batch algorithms for learning state value functions, such as Least-Squares Temporal Difference (LSTD) learning [11], which relate the values of states s and s', not numerically as TD-learning does, but in terms of their features. However, even such an approach would lead to problems, because, in order to extract the resulting next state s' under the MiniMax criterion to make the update in the linear system of LSTD, one has to consult the current state value function to determine the "optimal" players' moves. This need for bootstrapping points to an inherent problem with the use of state value functions in games. The state value function, by definition, estimates the expected return assuming that the agent follows a fixed (ideally, an optimal) policy. However, due to the necessary cut-offs in the search, the agent's policy is constantly changing, as it depends directly on the values of the state value function which is actually being learned at the same time. This is a major difference compared to MDPs, where TD and LSTD are perfectly fit to evaluate a fixed policy. As a result in adversarial games one would have to run LSTD periodically with new sample sets to gradually estimate the state value function of the current policy.

These observations made us rethink the appropriateness of the TD-learning approach to adversarial games, given that the quality of TD updates depends on the current quality of the learned value function, which in turn depends on the quality of the TD updates, ultimately giving rise to a dependency loop. This fact does not necessarily mean that TD-learning will fail. It merely implies that it needs a significant number of training samples and quite careful tuning of the learning rate α and the λ parameter to gradually reach a good value function.

VI. LSPI for Adversarial Games

To overcome the limitations noted in the previous section, we propose a focus on learning a state-action value function

Algorithm 2 LSPI for Adversarial Games.

Input: samples D, basis functions ϕ, discount factor γ, tolerance ϵ
Output: weights w for inferring the learned policy π

$w' \leftarrow \mathbf{0}$
repeat
  $w \leftarrow w'$, $\mathbf{A} \leftarrow \mathbf{0}$, $b \leftarrow \mathbf{0}$
  for each (s, a, o, r, s') in D **do**
    $a' = \arg\max_{a'' \in \mathcal{A}} \min_{o'' \in \mathcal{O}} \left\{ \phi(s', a'', o'')^\top w \right\}$
    $o' = \arg\min_{o'' \in \mathcal{O}} \left\{ \phi(s', a', o'')^\top w \right\}$
    $\mathbf{A} \leftarrow \mathbf{A} + \phi(s, a, o) \Big(\phi(s, a, o) - \gamma \phi(s', a', o') \Big)^\top$
    $b \leftarrow b + \phi(s, a, o) r$
  end for
  $w' \leftarrow \mathbf{A}^{-1} b$
until $\big(\|w - w'\| < \epsilon \big)$

Q using the batch Least-Squares Policy Iteration (LSPI) algorithm. Adopting a state-action value function $Q(s, a, o)$ shifts attention to learning the expected value of a state for (any) specific choices of the agent and the opponent in the first step. This simple extension effectively eliminates the disturbing dependency between the value function being learned and the policy being followed, simply because the "bigger" state-action value function accommodates values for any possible policy. On the practical side, a linear architecture for such a value function would require basis functions of the form $\phi(s, a, o)$ that depend on the state s and both actions a and o.

The major modification in LSPI to make it fit for adversarial games was the extraction of the implicit policy in the resulting next states s'. This is accomplished by a shallow (depth of 2) MiniMax search under s' which reveals the best choice a' of the Max over all responses of the Min, as well as the best response o' of the Min for the chosen best choice of the Max, whereby all these action choices are valued using purely the learned state-action value function Q. The same procedure is used during deployment of the learned policy to evaluate states at cut-off points before backing up their values through the MiniMax search.

Therefore, the updates made internally by LSPI to its linear system relates state-action values in one state to state-action values in the next state in terms of their features factoring in the action choices made by the actual policy being evaluated and not some estimate from the value function of the previous policy. Furthermore, the same sample set is used to evaluate all intermediately produced policies, an inherent feature of LSPI which is transferred unchanged from MDPs to adversarial games. In this context, the policy iteration procedure within LSPI can be interpreted as learning by a kind of self-play without actual games, but rather by an internal mining process over a single fixed set of samples.

It should be clear at this point that LSPI overcomes the limitations of TD-learning that motivated this work. The algorithm makes batch updates to the evaluation function with any collection of samples, can easily utilize samples from past games or from games played by other agents, and its updates do not depend on the current evaluation function since there is no bootstrapping. The complete LSPI algorithm for adversarial games is shown in Algorithm 2.

VII. Playing Othello/Reversi

A. Game Rules

Othello (or Reversi) [12] is a two-player board game played on an 8×8 grid using black and white discs (one color for each player). The initial setup consists of two black and two white discs centered on the grid in a cross-diagonal arrangement. The two players take alternating turns with the black player moving first and the white player having the parity. A move for a player consists of *outflanking* the opponent's disc(s), then *flipping* the outflanked disc(s) to the player's color. To *outflank* means to place a disc on the board, adjacent to one of the opponent's discs, so that the opponent's line (or lines) of discs is bordered at each end by a disc of the player's color. A *line* is defined as one or more same-colored discs in a continuous straight line horizontally, vertically, or diagonally. A disc may outflank any number of discs in one or more lines in any number of directions at the same time. Disc(s) may only be outflanked as a direct result of a move and must fall in the direct line of the disc placed down. All discs outflanked in any one move must be flipped, even if it is to the player's advantage not to flip them all. A player's move is forfeited, if that player cannot outflank and flip at least one opposing disc; in that case, the opponent takes the turn and the parity switches hands. However, if a player has at least one move available, the turn cannot be forfeited. When it is no longer possible for either player to move, the game is over. Discs of each color are counted and the player with the majority of discs on the board is the winner. The game ends with a tie, if the two players have the same number of discs on the board.

B. Evaluation Function Features

A game state in Othello consists of a board, an indication of the player who is about to move, and an indication of the player having the parity. After extensive experimentation, we found a small set of features that seem to be very informative for evaluation. For any given state [*board*, *player*, *parity*] of the game, we compute the following:

- **mobility**: Number of available moves for the *player* in the current *board*.
- **stability**: Number of *player*'s discs in the current *board* whose color cannot change in the rest of the game.
- **frontier**: Number of *player*'s discs in the current *board* adjacent to empty squares.
- **square**(i, j): Content (*player* disc, opponent disc, or empty) of square (i, j) in the current *board*.

The first three features are computed for each of the two players. The last set of 64 features gives a discrete integer value $(+1, -1, 0)$ to each square. These $6 + 64 = 70$ features along

with a constant term are used for approximating the state value function. Taking into account the parity has a significant effect on the playing skill of the agents. Therefore, we duplicate these 71 basis functions, so that there are separate blocks of basis functions for the parity and the non-parity player. The final approximation architecture for TD-learning consists of a total of 142 basis functions $\psi(s)$. For approximating the state-action value function in LSPI, we use three blocks of the above 70 features to define basis functions for any given state s and actions a, o. The first block is identical to the 70 features used by TD, that is, it depends only on s. The second block includes the differences in these 70 features caused by applying action a to state s. Finally, the third block includes the differences in these 70 features caused by applying action o to the state that resulted from applying action a to state s. These 210 features along with a constant term are used for approximating the state-action value function. Duplicating this basis of 211 basis functions to account for the parity, we end up with an approximation architecture for LSPI consisting of a total of 422 basis functions $\phi(s, a, o)$.

VIII. Experimental Results

To demonstrate the performance of the proposed algorithm over common practice, we adopt the following experimental methodology. First, TD begins with random weights and learns against a fixed clone of itself (a type of self-play). Learning goes on for 1000 moves of the agent, which include several games; then, learning is suspended and a small tournament of two games takes place (players play both sides). If the learning player exceeds the clone in performance (wins both games), a new cloning takes place, whereas if performance is not better, learning continues in the same manner without cloning. This ensures that the cloned opponent is constantly competent compared to the learning agent. This learning scheme continues up to a total of 20000 training moves. The cloned opponent over these 20000 moves plays randomly with probability p, where p follows a sigmoid schedule between 0.5 and 0 with the transition in the middle (10000 moves) to help with exploration. During the tournaments, randomness is turned off in the opponent, as well as learning in the agent. The entire sample set of 20000 samples collected by TD along this process is saved. We refer to this agent as TD (self-play). Second, to conduct a fair comparison and to demonstrate the ability of LSPI to use existing sample sets, we run an LSPI agent on the sample set collected by TD (self-play). The purpose of this experiment is to reveal the differences in utilizing the exact same samples. LSPI runs every 1000 moves using the samples from TD (self-play) up to that point starting with random initial weights, until it converges or a maximum of 30 iterations are reached. We refer to this agent as LSPI (TD samples). Third, we run an LSPI agent using self-play and tournament/cloning every 1000 moves in the exact same manner as TD (self-play) to demonstrate a simple possible way (out of many) to collect samples while learning with LSPI and to compare independent TD and LSPI learning agents. We refer to this agent as LSPI (self-play).

The two LSPI agents are compared against the TD agent every 1000 moves of training samples in terms of number of discs possessed and number of victories achieved in a two-game tournament between them. Figure 1 shows the results of the head-to-head comparison between TD (self-play) and LSPI (TD samples), whereas Figure 2 shows the results of the comparison between TD (self-play) and LSPI (self-play). Additionally, all three players are tested against a set of 5 fixed benchmark players, created and kept because of their good performance during past learning attempts. Each player plays twice (both sides) against each benchmark player and results are recorded in terms of game score and number of victories (Figure 3). Game score is shown in a 0-100 scale, whereby 100 indicates wins over all benchmark players by complete opponent wipeouts, whereas 0 indicates losses over all benchmark players by complete wipeouts. All these experiments were repeated 30 times to obtain averages over different sets of samples and an indication of statistical significance shown by the 95% confidence intervals on the graphs.

These experiments reveal that with a sample set in the order of 20000 moves LSPI is clearly a winner. To test whether more samples will make a difference, we repeated the above experiment with a total of 100000 moves. The learned TD and LSPI agents in this experiment are compared against each other every 5000 moves of training samples again in terms of game score and number of victories (Figure 4 and Figure 5). Additionally, all three players are tested against the same set of 5 benchmark players in terms of game score and number of victories (Figure 6). Once again, all these experiments were repeated 30 times to obtain averages and an indication of statistical significance shown by the 95% confidence intervals on the graphs. These larger experiments reveal that the advantages of LSPI over TD persist over a longer learning horizon. In fact, TD is slowly improving, but it does not seem to be able to reach the performance levels of LSPI obtained with a much smaller sample set. It is interesting however that LSPI seems to be a little better when using TD's samples, although the difference is not always statistically significant. A possible explanation for this difference could be the lack of variability in the games of LSPI (self-play) in the second half of the experiment where randomness decays rapidly; LSPI (self-play) ends up playing the same games over and over again with its clone, whereas LSPI (TD samples) benefits from the variability of games experienced by the constantly-changing TD (self-play) agent.

IX. Related Work, Discussion, and Future Work

Several researchers have pointed out limitations of traditional TD-learning when applied to adversarial games [13], [14]. The proposed remediation of these problems focuses mostly on accelerating convergence, performing multiple updates at each step, exploring alternative backup paths, and tuning learning parameters. These proposals nevertheless maintain the TD-style updates in their cores and attempt to optimize them. In contrast, our proposal suggests the replacement of the TD-style updates with a single LSPI-style update. The

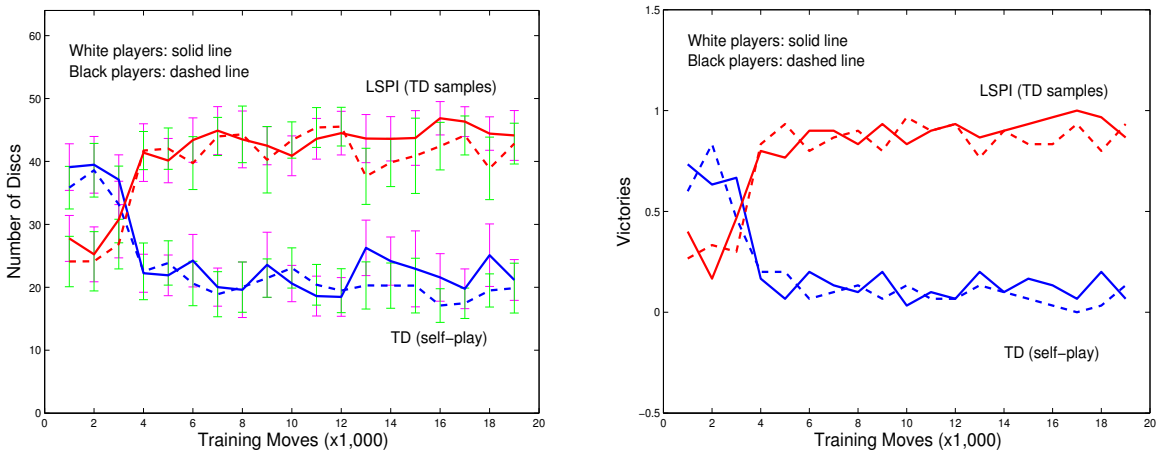

Fig. 1. TD (self-play) vs. LSPI (TD samples): Average number of discs and number of victories (20K).

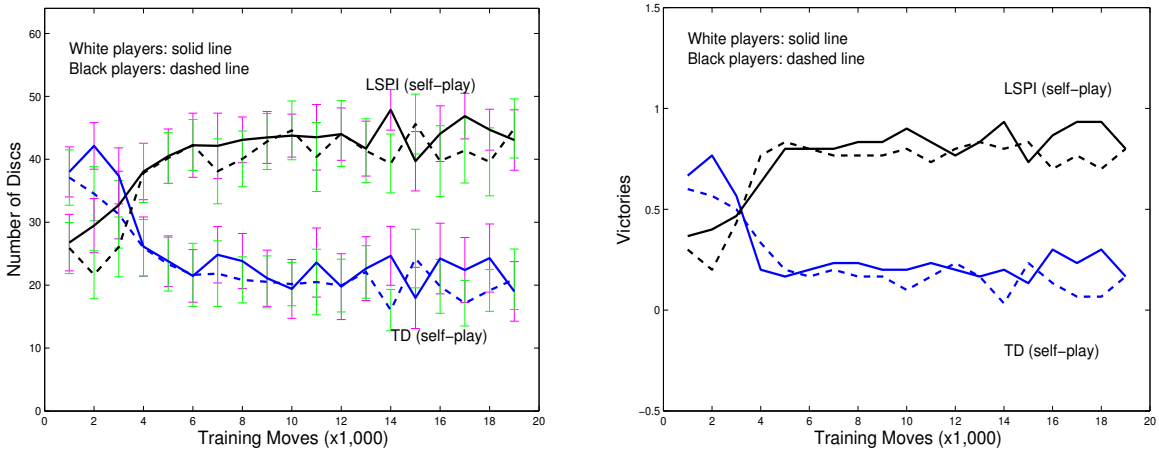

Fig. 2. TD (self-play) vs. LSPI (self-play): Average number of discs and number of victories (20K).

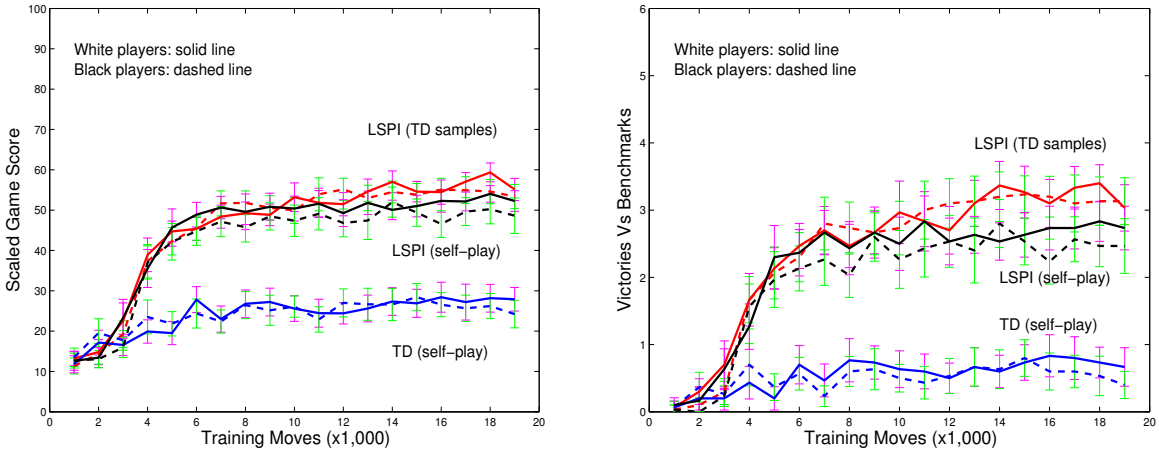

Fig. 3. Benchmarking TD (self-play), LSPI (self-play), and LSPI (TD samples): Average game score and number of victories (20K).

proposed improvements to TD-style learning mentioned above could be used in conjunction with LSPI-style learning to improve efficiency. This is a future research direction.

To our knowledge, there are no reports on using batch reinforcement learning algorithms, such as LSPI, in the context of games, apart from the work of Lagoudakis and Parr on zero-sum Markov games [15], [16] which improved upon the seminal work of Littman on MiniMax-Q [17] (a variation of Q-learning for Markov games). The modeling framework of Markov games assumes simultaneous moves by the participating agents, in contrast to the games we consider here whereby agents take turns. This difference requires the consideration

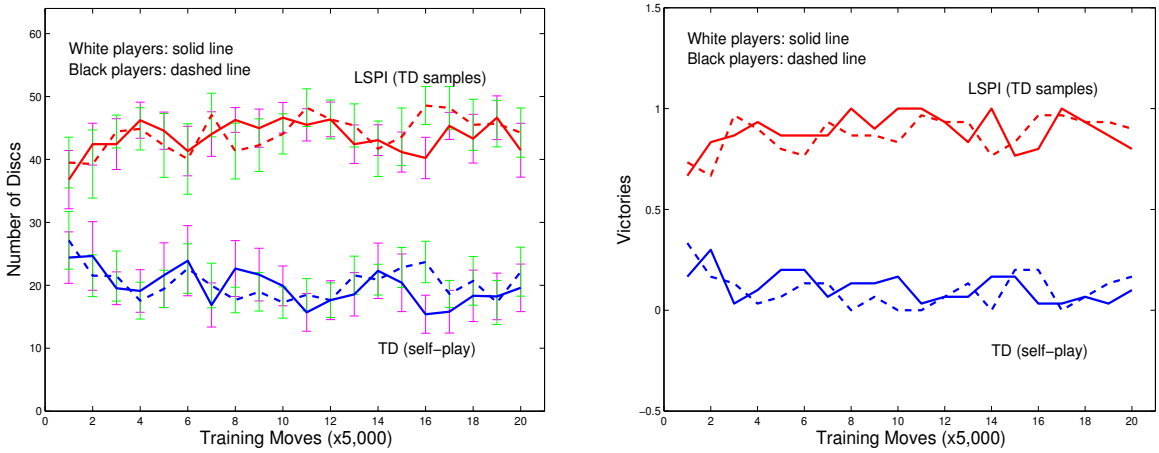

Fig. 4. TD (self-play) vs. LSPI (TD samples): Average number of discs and number of victories (100K).

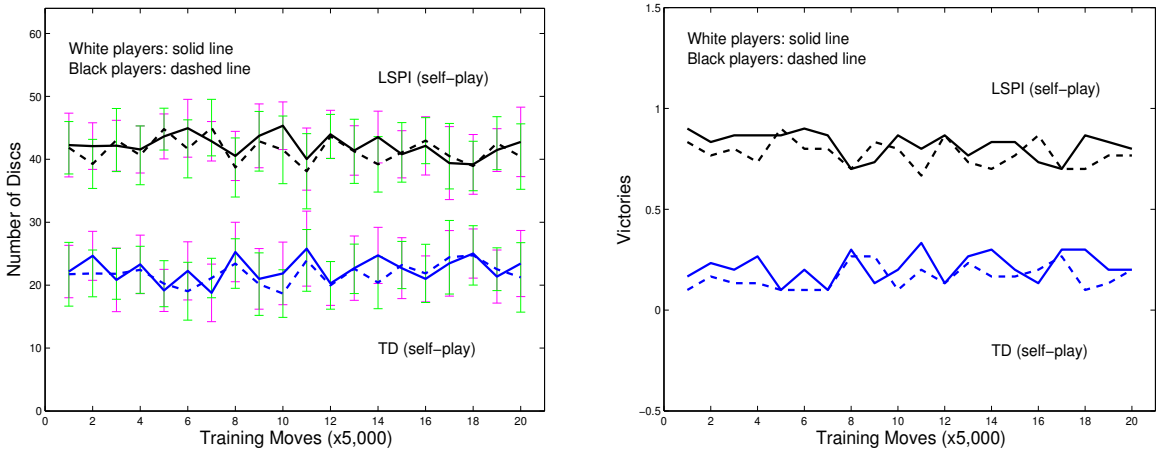

Fig. 5. TD (self-play) vs. LSPI (self-play): Average number of discs and number of victories (100K).

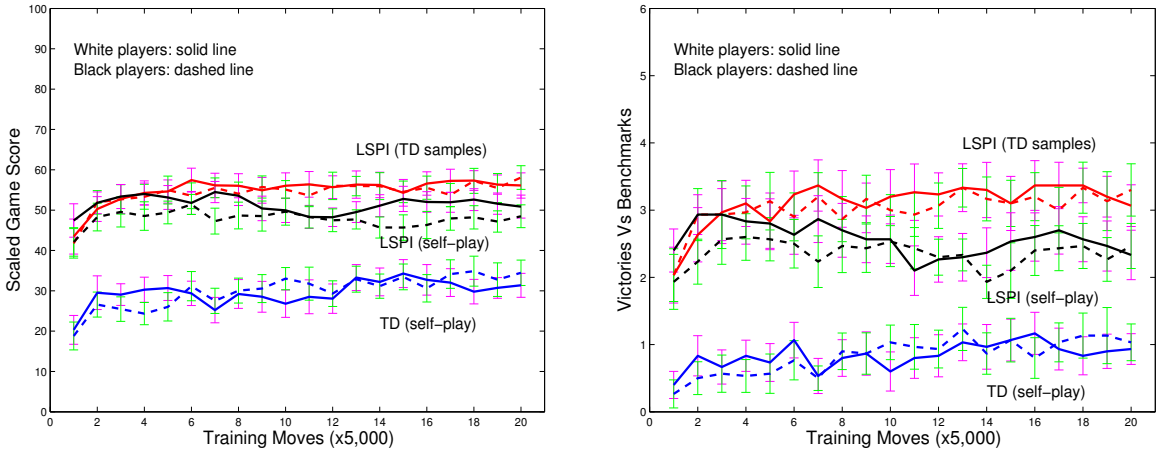

Fig. 6. Benchmarking TD (self-play), LSPI (self-play), and LSPI (TD samples): Average game score and number of victories (100K).

of stochastic policies in the context of Markov games, since there may be no optimal deterministic policy. Furthermore, (stochastic) MiniMax action selection in any state given a value function requires the formulation and solution of a linear program. It is well known, however, that in turn-taking games stochastic policies do not yield any advantage compared to deterministic ones [18], therefore we can restrict ourselves exclusively to deterministic policies for such games. In addition, MiniMax action selection does not require the solution of a linear program, but only a MiniMax search of

two plies, as described in Section VI. A naive application of the work of Littman and Lagoudakis and Parr to turn-taking games would result in excessive computational cost, in addition to highly conservative policies. The work presented here complements their work and extends the use of LSPI over a wider class of games.

Recent work has also explored the use of Least-Squares Temporal Difference (LSTD) learning on turn-taking adversarial games, in particular on the board game "Neighbours" [19] and Adversarial Tetris [20]. LSTD is able to process samples in batch, however it comes with the limitations associated with learning a state evaluation function. Results on both games demonstrated marginal advantage over TD learning. The LSPI-approach presented here incorporates a variant of LSTD in its inner loop for evaluating policies by learning a state-action value function. An approach to reuse samples with incremental-update learning algorithms is known as experience replay [21], because it stores samples and makes multiple passes (and updates) over them. This technique improves performance, however it still suffers from sample ordering and learning rate tuning. In addition, it is applicable only to off-policy learning algorithms, such as Q-learning, but not TD-learning which is an on-policy algorithm. Finally, it has been demonstrated experimentally that experience replay cannot yield the benefits of batch updates [8].

Our proposed method offers better utilization of samples and exhibits better learning efficiency, however at the expense of increased computational cost. Given that LSPI needs to update and solve a linear system, the computational cost is in the order of $O(l^3)$, where l is the number of features in the linear architecture. Nevertheless, given that the linear system is formed and solved only once, empirical evidence suggests that the time penalty is not so severe compared to the gained efficiency. Furthermore, recursive least-squares techniques can be used to reduce the complexity to $O(l^2)$ [8].

The game of Othello/Reversi is a popular domain for researchers in Artificial Intelligence and Machine Learning [22]. The game was highly popularized in the computer gaming community after the emergence of Logistello, which became well-known for winning the human world champion in a series of six games in 1997. Logistello was perfected using self-play and numerous parameters in its evaluation, which were tuned using supervised learning techniques [23]. Since then, several other strong Othello programs have emerged: Ntest, Saio, Edax, Cyrano, and WZebra. Our goal in this work was not to compete with such specialized systems, but only to demonstrate that the learning abilities of a game-playing agent can benefit from modern reinforcement learning techniques.

Given that the model of the game is typically known in most adversarial games, we are currently investigating the possibility of applying LSPI to a selected set of representative states taking as samples all legal transitions out of those states given the actions of both players. In addition, we are currently exploring the impact of our method in stochastic adversarial games, since uncertainty in transitions and rewards can be inherently handled by LSPI.

X. Conclusion

In this paper, we argued that reinforcement learning in adversarial games based on incremental TD-learning-style updates is limited by several factors inherent in these techniques. As an alternative, we advocated an approach that remedies these limitations by focusing on learning a state-action evaluation function using the batch Least-Squares Policy Iteration (LSPI) algorithm. We demonstrated the efficiency of the LSPI approach over the widely-used TD approach in the classical board game of Othello/Reversi.

References

[1] C. E. Shannon, "Programming a computer for playing Chess," *Philosophical Magazine*, vol. 41, no. 314, pp. 256–275, 1950.

[2] A. L. Samuel, "Some studies in machine learning using the game of Checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 211–229, 1959.

[3] M. L. Puterman, *Markov Decision Processes – Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., 1994.

[4] R. Bellman, *Dynamic Programming*. Princeton University Press, 1957.

[5] R. A. Howard, *Dynamic Programming and Markov Processes*. The MIT Press, 1960.

[6] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[7] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-Dynamic Programming*. Athena Scientific, 1996.

[8] M. G. Lagoudakis and R. Parr, "Least-squares policy iteration," *Journal of Machine Learning Research*, vol. 4, pp. 1107–1149, 2003.

[9] J. von Neumann and O. Morgenstern, *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[10] D. Knuth and R. Moore, "An analysis of alpha-beta pruning," *Artificial Intelligence*, vol. 6, no. 4, pp. 293–326, 1975.

[11] S. J. Bradtke and A. G. Barto, "Linear least-squares algorithms for temporal difference learning," *Machine Learning*, pp. 22–33, 1996.

[12] Wikipedia, "Reversi — wikipedia, the free encyclopedia," 2012. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Reversi

[13] J. Baxter, A. Tridgell, and L. Weaver, "KnightCap: A chess program that learns by combining TD(lambda) with game-tree search," in *Proceedings of the Fifteenth International Conference on Machine Learning (ICML)*, 1998, pp. 28–36.

[14] J. Veness, D. Silver, W. Uther, and A. Blair, "Bootstrapping from game tree search," in *Advances in Neural Information Processing Systems (NIPS) 22*, Y. Bengio, D. Schuurmans, J. Lafferty, C. K. I. Williams, and A. Culotta, Eds., 2009, pp. 1937–1945.

[15] M. G. Lagoudakis and R. Parr, "Value function approximation in zero-sum Markov games," in *Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence (UAI)*, 2002, pp. 283–292.

[16] ——, "Learning in zero-sum team Markov games using factored value functions," in *Advances in Neural Information Processing Systems (NIPS) 15*, S. Becker, S. Thrun, and K. Obermayer, Eds., 2003, pp. 1627–1634.

[17] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," in *Proceedings of the Eleventh International Conference on Machine Learning (ICML)*, 1994, pp. 157–163.

[18] S. Russel and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice Hall, 2003.

[19] I. Skoulakis, "Systematic search and reinforcement learning for the board game "Neighbours"," Diploma Thesis, Technical University of Crete, Greece, 2010.

[20] M. Rovatsou and M. G. Lagoudakis, "Minimax search and reinforcement learning for adversarial Tetris." in *Proceedings of the 6th Hellenic Conference on Artificial Intelligence (SETN)*, 2010, pp. 417–422.

[21] L. Lin, "Reinforcement learning for robots using neural networks," Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1993.

[22] M. Buro, "The evolution of strong Othello programs," in *Proceedings of the IFIP First International Workshop on Entertainment Computing (IWEC)*, 2002, pp. 81–88.

[23] ——, "Improving heurisitic mini-max search by supervised learning," *Artificial Intelligence*, vol. 134, no. 1-2, pp. 85–99, 2002.