Technical University of Crete, Greece

School of Electrical and Computer Engineering

Efficient Reinforcement Learning in Adversarial Games

Master's Thesis

Ioannis E. Skoulakis

Chania, October 2019

Πολυτεχνειο Κρητης

Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

Αποδοτική Ενισχυτική Μάθηση σε Παιχνίδια με Αντιπαλότητα

Μεταπτυχιακή Διατριβή

Ιωάννης Ε. Σκουλάκης

Χανιά, Οκτώβριος 2019

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 thesis, we present four variations of a learning approach based on the Least-Squares Policy Iteration (LSPI) algorithm that overcome these limitations by focusing on learning a state-action evaluation function. The key advantage of the proposed approaches 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 and the competency of the LSPI agents over the TD agent and selected benchmark opponents in the classical board games of Othello/Reversi and Backgammon.

Περίληψη

Η μάθηση είναι μία κρίσιμη ικανότητα για πράκτορες που σχεδιάζονται για να λαμβάνουν μέρος σε ανταγωνιστικά παιχνίδια εναλλασσόμενων κινήσεων δύο παικτών όπως το τάβλι, το Othello, το σκάκι, το Hex, κλπ. Η επικρατούσα προσέγγιση της μάθησης σε παιχνίδια αυτού του είδους συνίσταται στην ενημέρωση κάποιας συνάρτησης αξιολόγησης καταστάσεων, συνήθως με την έννοια της χρονικής διαφοράς (TD) είτε υπό το κριτήριο βελτιστοποίησης MiniMax είτε υπό βελτιστοποίηση έναντι συγκεκριμένου αντιπάλου. Ωστόσο, η προσέγγιση αυτή περιορίζεται από διάφορους παράγοντες: (α) οι ενημερώσεις στη συνάρτηση αξιολόγησης είναι σταδιακές, (β) δείγματα από προηγούμενα παιχνίδια δεν μπορούν να αξιοποιηθούν, και (γ) η ποιότητα κάθε ενημέρωσης εξαρτάται από την τρέχουσα συνάρτηση αξιολόγησης. Σε αυτή τη διατριβή, παρουσιάζουμε τέσσερις παραλλαγές μιας προσέγγισης μάθησης βασισμένης στον αλγόριθμο Least-Squares Policy Iteration (LSPI) που εστιάζονται στην εκμάθηση συναρτήσεων αξιολόγησης καταστάσεων-ενεργειών (state-action) και δεν πλήττονται από τους προαναφερθέντες περιορισμούς. Το βασικό πλεονέκτημα των προτεινόμενων προσεγγίσεων είναι ότι ο πράκτορας μπορεί να κάνει μαζικές ενημερώσεις στη συνάρτηση αξιολόγησης με οποιαδήποτε συλλογή δειγμάτων, μπορεί να αξιοποιήσει δείγματα από παρελθόντα παιχνίδια, και μπορεί να κάνει ενημερώσεις που δεν εξαρτώνται από την τρέχουσα συνάρτηση αξιολόγησης. Παρουσιάζουμε την αποτελεσματικότητα και την ανταγωνιστικότητα των LSPI πρακτόρων έναντι του TD πράκτορα και επιλεγμένων ‘δύσκολων’ πρακτόρων στα κλασικά επιτραπέζια παιχνίδια Othello και τάβλι.

Acknowledgements

I would like to express my deepest gratitute to my supervisor, Associate Professor Michail G. Lagoudakis, for his guidance and patience throughout this work. I would also like to thank my family and friends for their encouragement and understanding.

Contents

List of Figures

List of Algorithms

Chapter 1

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.

1.1 Thesis Contribution

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 thesis, 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 four variations of 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 games of Othello/Reversi and Backgammon.

Part of this work was presented in ICTAI 2012 [3].

1.2 Thesis Outline

This thesis is organized as follows: Chapter 2 provides background information on reinforcement learning along with the main design principles for game-playing agents and the TD-approach to learning in adversarial games. Chapter 3 discusses the limitations of the TD-approach, states the problem we study, and gives information on related work. Chapter 4 presents our approach to learning in adversarial games and the four variations of the LSPI algorithms we propose. Chapter 5 describes our modeling approach to Othello/Reversi and Backgammon. Chapter 6 summarizes our experimental results, which include detailed comparisons. Finally, Chapter 7 concludes with some discussion and final thoughts.

Chapter 2

Background

2.1 Markov Decision Processes (MDPs)

A Markov Decision Process (MDP) [4] 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; $\mathcal{A} = \{a_1, a_2, \ldots, a_l\}$ 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:

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

2.2 Value Functions and Policies

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 [5]. 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) [6].

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 [7, 8]; 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):

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

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

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}.

2.3 Batch Learning Algorithms

In this section, we will present briefly two reinforcement learning algorithms that were used in this work.

2.3.1 Least-Squares Policy Iteration (LSPI)

Least-Squares Policy Iteration (LSPI) [9] is an efficient reinforcement learning algorithm that combines policy iteration with value function approximation. LSPI is model-free and learns in a batch manner by processing multiple times a set of samples collected arbitrarily from the process. Each sample (s, a, r, s') contains the state s that was observed, the chosen action a, the reward r received for that action and the resulted state s'. At each iteration a $(m \times m)$ linear system is being solved resulting in a new approximation for the

Algorithm 1 Least-Squares Policy Iteration (LSPI)

```
$w = \textbf{LSPI}(D, m, \phi, \gamma, \epsilon)$
  **Input:** samples $D$, integer $m$, basis functions $\phi$, discount factor $\gamma$, tolerance $\epsilon$
  **Output:** weights $w$ of the learned value function of the best learned policy

  $w \leftarrow \mathbf{0}$
  repeat
    $\mathbf{A} \leftarrow \mathbf{0}$        // $(m \times m)$ matrix
    $b \leftarrow \mathbf{0}$        // $(m \times 1)$ vector
    $w' \leftarrow w$
    for each sample $(s, a, r, s')$ in $D$ do
      $a' = \arg\max_{a'' \in \mathcal{A}} \phi(s', a'')^\top w'$
      $\mathbf{A} \leftarrow \mathbf{A} + \phi(s, a)\Big(\phi(s, a) - \gamma\phi(s', a')\Big)^\top$
      $b \leftarrow b + \phi(s, a) r$
    end for
    $w \leftarrow \mathbf{A}^{-1} b$
  until $\Big(\|w - w'\| < \epsilon\Big)$
  return $w$
```

value function. 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.

2.3.2 Fitted Q Iteration (FQI)

Fitted-Q Iteration (FQI) [10] uses a batch of samples along with a supervised learning (regression) algorithm in its inner loop to improve the value function successively. FQI uses Q-values, which enables it to perform model-free maximization steps for policy improvement with each iteration. A highly desirable property of this algorithm is that it can be used both with linear and non-linear function approximators. It has been demonstrated to achieve excellent performance when combined with random forest type approxima-

Algorithm 2 Least-Squares Fitted Q-Iteration

```
$w$ = LS-F$Q$I($D, m, \phi, \gamma, N$)
  Input: samples $D$, integer $m$, basis functions $\phi$, discount factor $\gamma$, iterations $N$
  Output: weights $w$ of the learned value function of the best learned policy

  $i \leftarrow 0$
  $w \leftarrow \mathbf{0}$
  while ($i < N$) do
    $\mathbf{A} \leftarrow \mathbf{0}$          // $(m \times m)$ matrix
    $b \leftarrow \mathbf{0}$          // $(m \times 1)$ vector
    for each sample $(s, a, r, s')$ in $D$ do
      $\mathbf{A} \leftarrow \mathbf{A} + \phi(s,a)\phi(s,a)^\top$
      $b \leftarrow b + \phi(s,a)\Big(r + \gamma \max_{a' \in \mathcal{A}} \Big\{\phi(s',a')^\top w\Big\}\Big)$
    end for
    $w \leftarrow \mathbf{A}^{-1} b$
    $i \leftarrow i + 1$
  end while
  return $w$
```

tors, as well as neural networks [11]. FQI algorithm using least-squares regression is summarized in Algorithm 2.

2.4 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; $\mathcal{O} = \{o_1, o_2, \ldots, o_l\}$ 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 [12] 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 [13] 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.

2.5 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.

2.5.1 Temporal Difference (TD)

The most popular approach of reinforcement learning used in games is Temporal Difference (TD) learning [7], 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:

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

Chapter 3

Problem Statement

3.1 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 [14], 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.

In this thesis we present multiple algorithms for learning in adversarial games based on LSPI, that do not suffer from the problems that the TD algorithm introduces. We argue that LSPI can help us reach higher performance levels easier, and we demonstrate this performance gain in the popular board games of Othello and Backgammon.

3.2 Related Work

Several researchers have pointed out limitations of traditional TD-learning when applied to adversarial games [15, 16]. 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.

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 [17, 18] which improved upon the seminal work of Littman on MiniMax-Q [19] (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 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 [20], therefore we can restrict ourselves exclusively to deterministic policies for such games. 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" [21] and Adversarial Tetris [22]. 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. An approach to reuse samples with incremental-update learning algorithms is known as experience replay [23], 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 [9]. The LSPI-approach presented here incorporates a variant of LSTD in its inner loop for evaluating policies by learning a state-action value function.

The game of Othello/Reversi is a popular domain for researchers in Artificial Intelligence and Machine Learning [24]. 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 [25]. Since then, several other strong Othello programs have emerged: Ntest, Saio, Edax, Cyrano, and WZebra.

In 1992, Gerald Tesauro developed TD-Gammon [26], a neural network agent for the game of Backgammon, capable of competing with top human players of the time. TD-Gammon was trained using Temporal Difference (TD) learning and about 1.5 million games (self-play). Its state evaluation function brought changes to common practices expert human players followed.

Chapter 4

Adapting LSPI

This chapter describes in detail the adaptations we made to the original LSPI algorithm to make it applicable to adversarial games. The four variations presented complement each other in the sense that none of them covers adequately all domains, but one of them may be more suitable for any given domain.

4.1 LSPI for Adversarial Games

To overcome the limitations noted in Chapter 3, we propose a focus on learning a state-action value function 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

Algorithm 3 LSPI for Adversarial Games.

```
Input: samples $D$, basis functions $\phi$, discount factor $\gamma$, tolerance $\epsilon$
Output: weights $w$ for inferring the learned policy $\pi$

$w' \leftarrow \mathbf{0}$
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 $\Big(\|w - w'\| < \epsilon\Big)$
return $w$
```

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 3.

4.2 Model-Based LSPI

Since the transition model, in the environments we are interested in, is known, we could use it to broaden our learning sample set. We can achieve this, if we ignore the actions taken when the sample was observed and recorded, and we explore all legal actions for both players instead. Each sample will practically contain only state s, and during the learning process we will have to generate all legal (a, o) pairs for that state. Afterwards, we continue by performing multiple shallow MiniMax searches for each state s' we end up to after those actions. The changes that need to be made can be seen in Algorithm 4.

In essence, we are learning using the exact same method, but the sample set used is expanded by a factor of b^2, where b is the branching factor of the game tree. This model-based approach will allow us to explore wider sections of the game tree, and provide more data, resulting in a player with a broader state-space knowledge.

In reality, the information that each state holds is not equally good. Some states explore valuable parts of the game tree, that we encounter often, while others lead us to uncommon paths, that do not add anything of value, and only end up making convergence a more difficult task. In order for the LSPI algorithm (and any other batch algorithm for that matter) to converge under these conditions, some tuning may be needed. One solution to help our algorithm converge, is to simply reduce the discount factor (γ). A different option could be putting a mechanism in place in order to quickly evaluate and discard states of low importance. This however, creates unwelcome complexity and more parameters to calibrate.

Because of the nature of some problems, getting new samples can be, under specific circumstances, expensive. In those cases, relying on our knowledge of the transition model, can be highly desirable, and a model-based approach can be our only real option.

Algorithm 4 Model-Based LSPI for Adversarial Games.

```
**Input:** samples $D$, basis functions $\phi$, discount factor $\gamma$, tolerance $\epsilon$
**Output:** weights $w$ for inferring the learned policy $\pi$

$w' \leftarrow \mathbf{0}$
**repeat**
    $w \leftarrow w'$, $\mathbf{A} \leftarrow \mathbf{0}$, $b \leftarrow \mathbf{0}$
    **for each** $s$ in $D$ **do**
        **for each** $a \in \mathcal{A}(s)$ **do**
            **for each** $o \in \mathcal{O}(\bar{s})$ {$\bar{s}$ is the intermediate state resulting from the execution of $a$ in $s$} **do**
                $s' = \arg\max_{s'' \in \mathcal{S}} \mathcal{P}(s''|s, a, o)$
                $a' = \arg\max_{a'' \in \mathcal{A}} \min_{o'' \in \mathcal{O}} \left\{ \phi(s', a'', o'')^\top w \right\}$
                $o' = \arg\min_{o'' \in \mathcal{O}} \left\{ \phi(s', a', o'')^\top w \right\}$
                $\mathbf{A} \leftarrow \mathbf{A} + \phi(s, a, o)\Big(\phi(s, a, o) - \gamma\phi(s', a', o')\Big)^\top$
                $b \leftarrow b + \phi(s, a, o)\mathcal{R}(s, a, o)$
            **end for**
        **end for**
    **end for**
    $w' \leftarrow \mathbf{A}^{-1}b$
**until** $\Big(\|w - w'\| < \epsilon\Big)$
**return** $w$
```

4.3 Complicated and Stochastic Games

The above approaches can also be used with stochastic adversarial games or games with large branching factor values (b). In practice though, it is not always feasible to complete the process in reasonable time due to the extra ply introduced by the stochastic element or due to the vast number of available moves. To mitigate this, we propose the usage of the following modified versions.

4.3.1 LSPI (Lite)

Thus far we talked about basis functions that take into account not only our move a, but also our opponent's response o. Endorsing this approach however is not our only option. Our basis functions can also depend solely on a state and the result of the following action on that state. Since we do not remove any information helping us to evaluate a single state, our perception of a stationary board does not change, but we choose to ignore the action we believe the second player will take. This modification will help us reduce the number of features and help speed up the learning process. This reduction does not necessarily mean much lower performance.

Taking all these into account, we can modify our LSPI approach as seen in Algorithm 5. The selection of a' in this case only needs a 1-depth MinMax search, speeding up the learning process even further.

Algorithm 5 LSPI (Lite) for Adversarial Games.

```
**Input:** samples $D$, basis functions $\phi$, discount factor $\gamma$, tolerance $\epsilon$
**Output:** weights $w$ for inferring the learned policy $\pi$

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

It is worth noting that the reward r is the reward our player receives after both moves a and o (if applicable) are completed. This is important since, if the reward referred only to our move, we would have no feedback when a game ends after an opponent's move.

Despite the similarity with the original LSPI algorithm for singe-agent domains, here both the next state s' and the reward r contain information coming from the opponent choices. In a sense, the opponent is viewed as part of a noisy, adversarial environment.

4.3.2 LSPI (Lite Dual)

With the modifications introduced in the previous section we managed to speed up the learning process considerably, thus making it more practical in certain situations. While achieving this though, we sacrificed part of the information we collected in our sample set.

With LSPI (Lite) we chose to ignore opponent's move, but instead we can exploit it. Each of our samples (s, a, o, r, s') can be represented or rewritten as $(s, a, r_a, \bar{s}, o, r_o, s')$, where $\bar{s}$ is the intermediate state resulting from the execution of move a in state s, r_a is the reward our player receives after move a, r_o is the reward our player receives after move o and can be treated as two separate samples, $(s, a, r_a, \bar{s})$ and $(\bar{s}, o, r_o, s')$. If the second part of the sample is taken as a representation of the state and reward, as if our player experienced it from its point of view, then we can use this part without any further alternations. Otherwise, we need to either modify the algorithm further or convert the second part of the sample.

Utilizing all the available information will basically double our sample set. This means that this algorithm is a bit slower than LSPI (Lite), but it may yield better results if the sample set is small. We call this modified version LSPI (Lite Dual) and it is shown in Algorithm 6.

Algorithm 6 LSPI (Lite Dual) for Adversarial Games.

```
Input: samples $D$, basis functions $\phi$, discount factor $\gamma$, tolerance $\epsilon$
Output: weights $w$ for inferring the learned policy $\pi$

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

Chapter 5

Games

5.1 Playing Othello/Reversi

5.1.1 Othello Game Rules

Othello (or Reversi) [27] 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, as shown in Figure 5.1. 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.

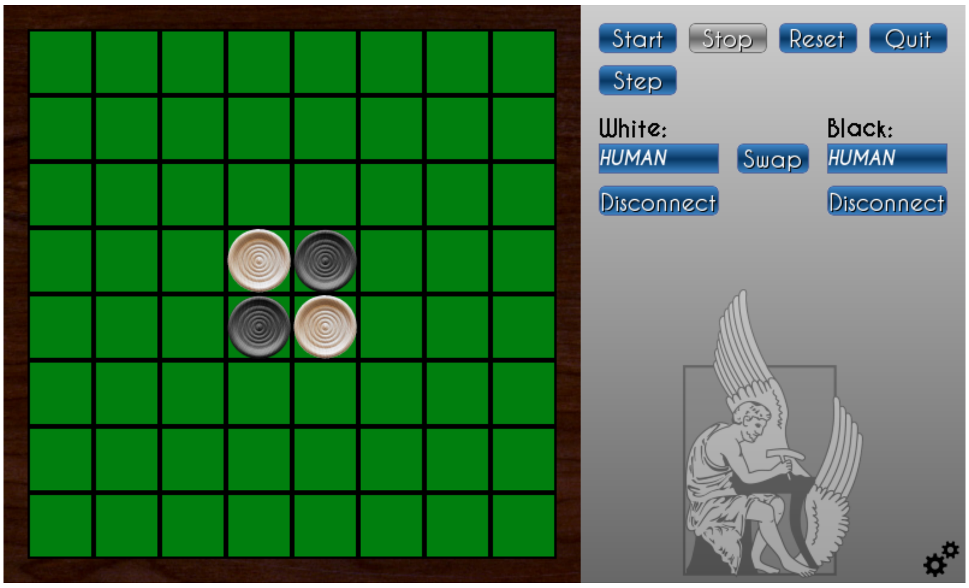


Figure 5.1: A GUI server for the game of Othello. Created using C++ and SDL.

5.1.2 Othello 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 (bias) 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 $\phi(s, a, o)$. For the Lite variations the set of basis functions is defined in a similar way, but its size is only 282, since it takes the form of $\phi(s, a)$.

5.2 Playing Backgammon

5.2.1 Backgammon Game Rules

Backgammon [28] is an old board game played by two players. Players take alternating turns moving their 15 discs, with ultimate goal to remove (bear off) all their pieces before their opponent. Backgammon has many variants, but we will explore the most common one, ignoring different starting setups and the existence of the doubling cube.

Each side of the board has a track of 12 positions. The total of the 24 positions form a continuous U-Shaped track and are numbered from 1 to 24. The two players move their discs in opposing directions from the 24-position towards the 1-position. Positions 1 through 6 are called the *home board* or the first quarter of the corresponding player. Positions 7 through 12 form the second quarter and so on.

In the beginning of the game each player has 15 discs placed on the board. Two discs are placed on their 24-position, three on their 8-position, five on their 13-position and their 6-position, as shown in Figure 5.2. The player which moves first is decided by a die roll.

Each turn starts by rolling two dice, whose outcome decides the movement. The player has to move his discs according to the dice roll. The two rolls can be used to move two different discs or the same one twice, as long as the two moves can be made separately and legally. If the two dice roll the same number, the player plays each die roll twice. On any roll the player must use both die rolls to move, if that option is available. If only one of the rolls is legally possible, then the highest roll must be played. Player that has no valid moves left forfeits his turn.

A piece can land on any empty position or a position occupied by any number of friendly discs. A disc cannot land on a position which is already occupied by two or more opponent's discs. If a piece ends up in a position with only one opponent's disc, then it occupies that position and the opponent's piece is placed on the bar that divides the board in two. In this case, the opponent's disc has been "hit". Any number of discs can be placed on the bar following this process, and all discs placed there must re-enter the game through their opponent's home board, before any other move can be made by that player. A roll of 1 allows the disc to enter on the 24-position, if that position is valid according to the restrictions above, a roll of 2 on the 23-position, and so forth. A "hit" can occur during this re-entry.

When a player has all his discs inside his home board, he can start removing them from the game, a process called "bearing off". A roll of 1 may be used to bear off a disc from the 1-position, a roll of 2 from the 2-position, and so on. If all of a player's pieces are on positions lower than the number showing on a particular die, the player may use that die to bear off one disc from the highest occupied point. Note that discs may still move during "bearing off" and "hits" are also possible.

The first player to bear off all his pieces wins the game. If the opponent has not yet removed any discs, then the victory is considered a double victory (a gammon). Finally, there is also the concept of a triple victory (a backgammon), which occurs if the opponent still has all his pieces on the board and at least one on the bar or in the winner's home board. This last concept of a triple victory is rare and it is ignored in our implementation.

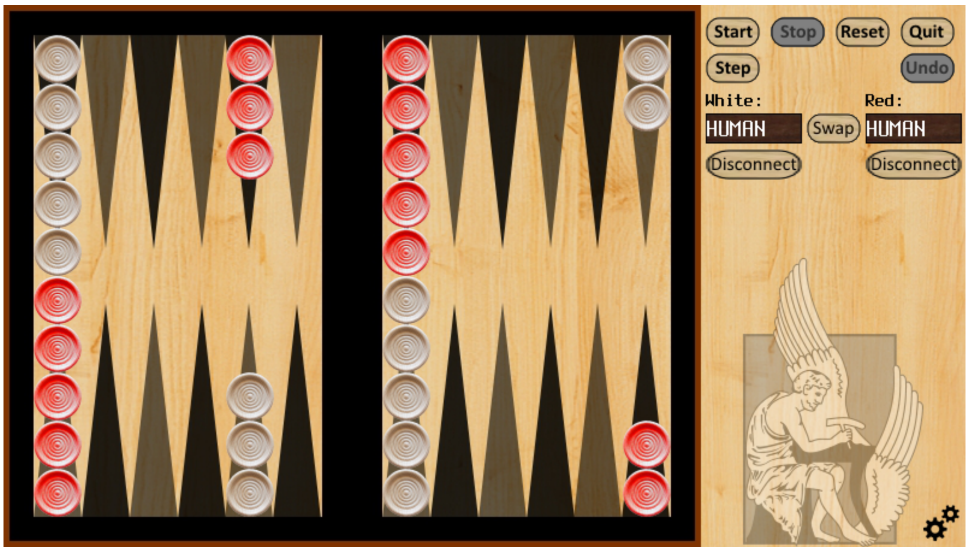


Figure 5.2: A GUI server for the game of Backgammon. Created using C++ and SDL.

5.2.2 Backgammon Evaluation Function Features

The game can be seen as two separate ones. In the first part, players can interact with their opponents. Pieces can hit other pieces or create road blocks in order to delay opponent's advance towards the finish line. We call this first part "contact". The second part is completely different. There is no interaction between the two players and player's only goal is to use the dice roll as efficiently as possible and remove all friendly discs, before the opponent remove his. We call this second part "race".

Backgammon is a game where positioning is of great importance. We chose our set of features with that fact in mind. With the game stages being so widely different, it is natural that our approach will reflect this fact. As a result, we created two separate feature sets, one for the "contact" stage and the other for the "race" stage.

If the *board* we are trying to evaluate is a "contact" *board*, we compute the following:

- **on bar**: Number of pieces placed on the bar for the *player* in the current *board*.
- **loners**(i): The sum of positions that have exactly one piece belonging to the *player* in each quarter (i) of the board.
- **blocks**(i): The sum of positions that hold more than one piece belonging to the *player* in each of the first two quarters (i) of the board (positions 1-12).

- **furthest piece**: The distance the further friendly piece has from its *home board*.
- **block value**: The sum of *block scores* inside the first two quarters (positions 1-12), where *block score* is defined as 2^{k-1}, where k is the number of continuous *block* groups.

If on the other hand the *board* is a "race" *board*, we compute the following:

- **removed**: Number of removed pieces for the *player* in the current *board*.
- **home board pieces**: The sum of positions of friendly pieces located inside the *home board*.
- **outer board pieces**: The sum of distances from the *home board* for all friendly pieces located outside the *home board*.

All features above are normalized and calculated for both players. This gives us 18 features for the "contact" stage and 6 features for the "race" stage. After adding 2 constants, one for each stage, TD reaches 26 basis functions $\psi(s)$. LSPI (Lite variations) needs twice as many features for each stage (excluding the two constancts), thus ending up with 50 basis functions $\phi(s, a)$ in total.

In 1993, Tesauro made code for a Backgammon player public [29] in order to create a universal benchmarking player. In his message he noted that this player had 57% win-rate against "gammontool", a program created by Sun Microsystems, and a 75% win-rate against Unix programs, backgammon and btlgammon. This player, named "pubeval" used two sets of 122 features, the first for the "contact" stage and the second for the "race" stage, resulting in a total of 244 features.

Our goal is not to create the best Backgammon player, but to compare the resulted TD and LSPI players. The size of the feature set we are using is relatively small. In comparison with the Tesauro's "pubeval" player, our LSPI player uses almost 5 times less features. The difference is even bigger considering that our LSPI uses its feature set for describing two states instead of one, thus that can be seen as using 10 times less features to describe one state. That fact is something we expect to have a huge impact, when we try to put our players against this strong opponent.

5.3 GUI Servers

For demonstration purposes we created two GUI servers, one for each game. These user interfaces were not used during the learning process because of the impact they would have on the speed due to their graphical component. They support the connection of multiple external players and give us control for manual play if needed. The two GUIs are shown in Figure 5.1 and Figure 5.2.

Chapter 6

Experimental Results

6.1 Learning Methodology

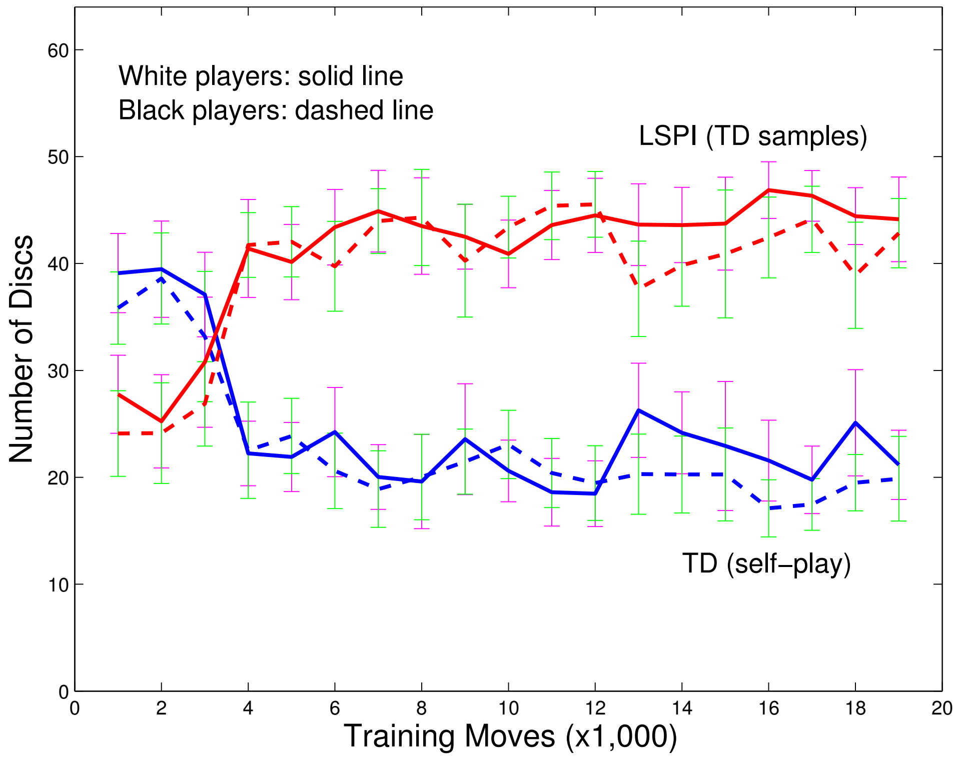
To demonstrate the performance of the proposed algorithms over common practice, we adopt the following experimental methodology. First, TD begins with random weights and learns against a fixed clone of itself (a type of self-play). Learning goes on for 1000 moves of the agent, which include several games; 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. The mentioned 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). Both our LSPI players use a discount factor (γ) value of 0.99999.

6.2 First Comparisons

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 6.1 shows the results of the head-to-head comparison between TD (self-play) and LSPI (TD samples), whereas Figure 6.2 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 6.3). Game score is shown in a 0-100 scale, whereby the extreme value 100 indicates wins over all benchmark players by complete opponent wipeouts, whereas the other extreme value 0 indicates losses over all benchmark players by complete wipeouts. All other scores fall in between. Higher scores are obviously better and indicate competency, but not necessarily victories, which are recorded and presented separately. 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.



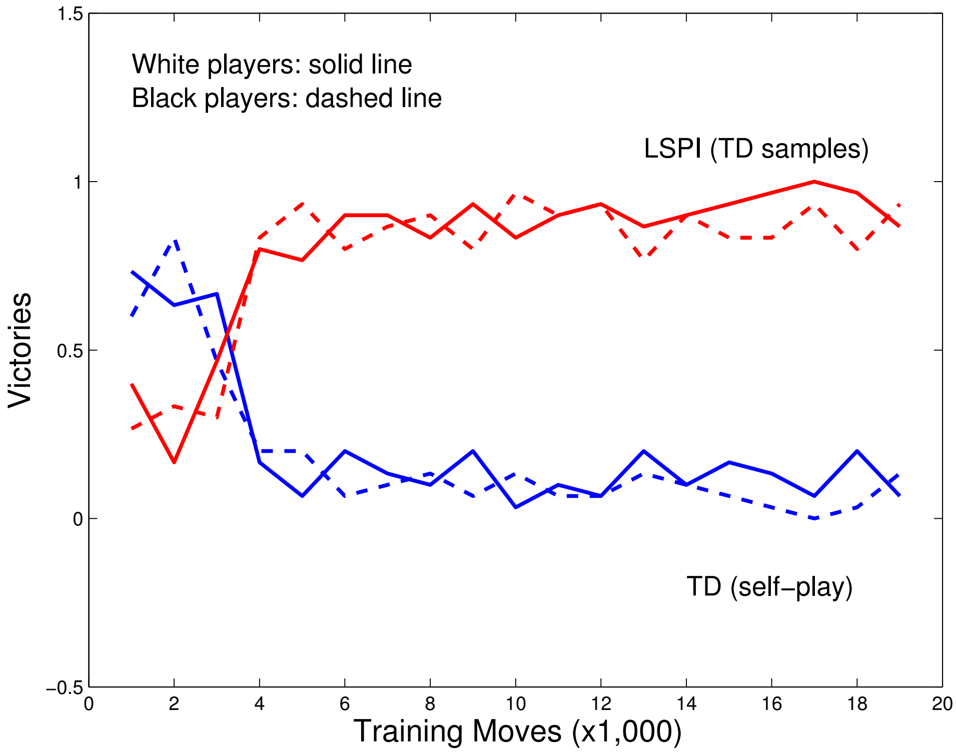
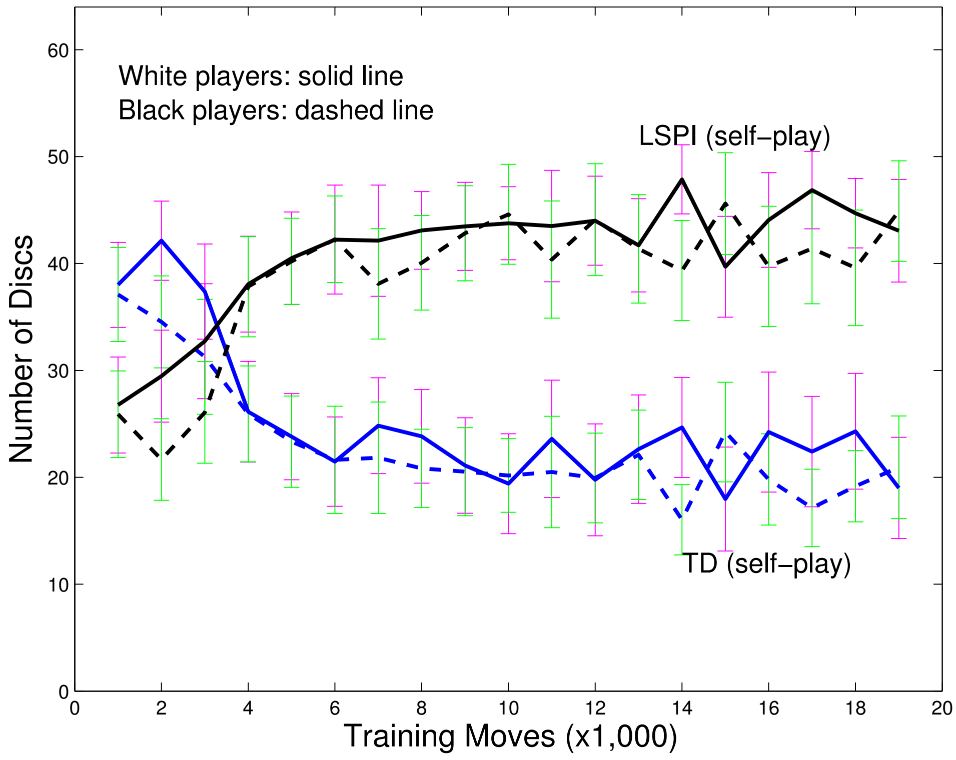


Figure 6.1: TD (self-play) vs. LSPI (TD samples): Average number of discs and number of victories (20K).



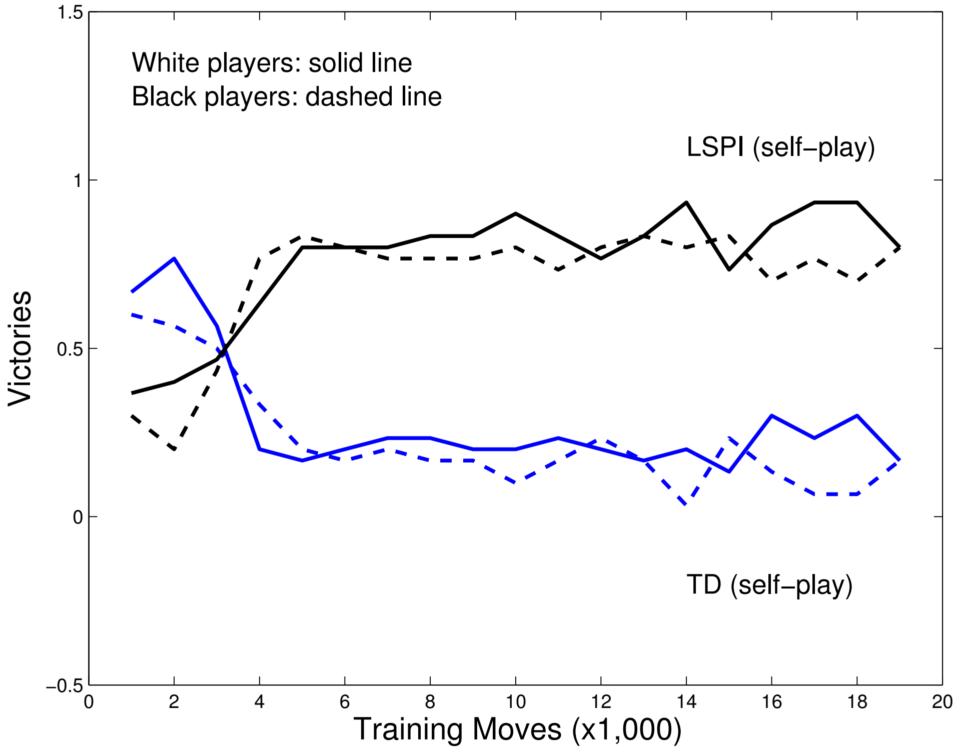
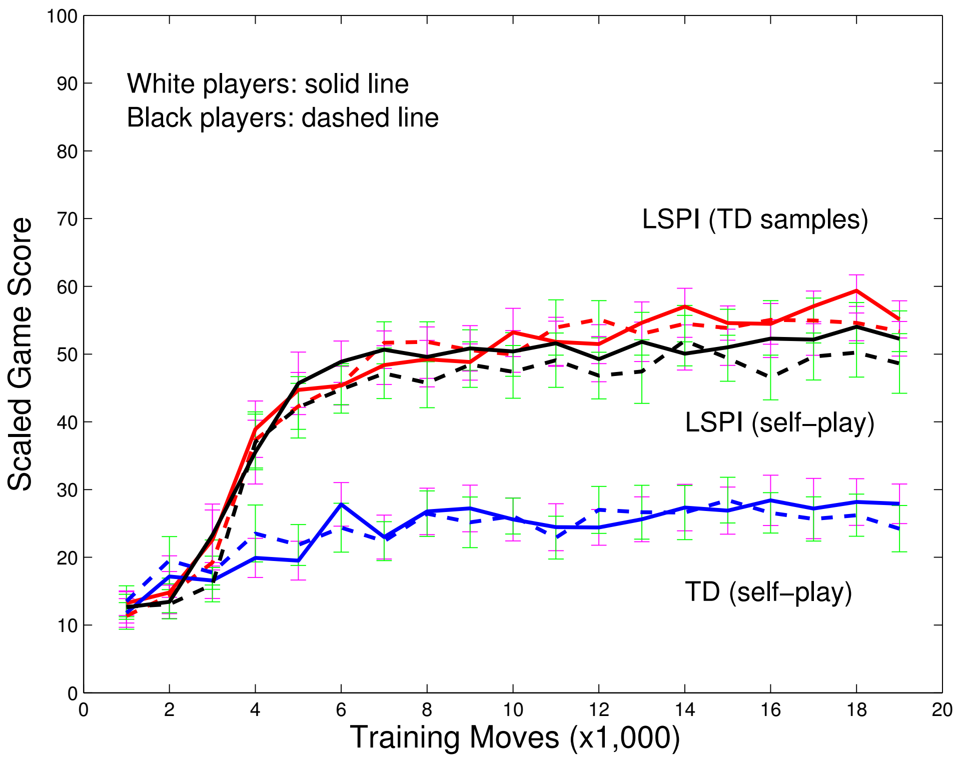


Figure 6.2: TD (self-play) vs. LSPI (self-play): Average number of discs and number of victories (20K).



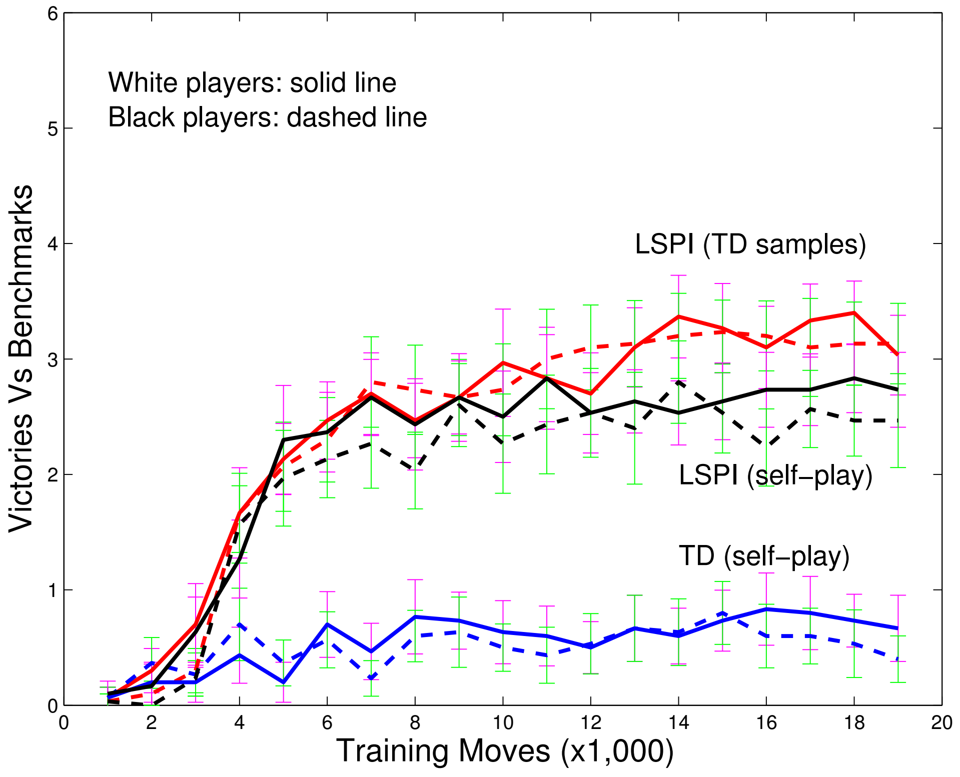
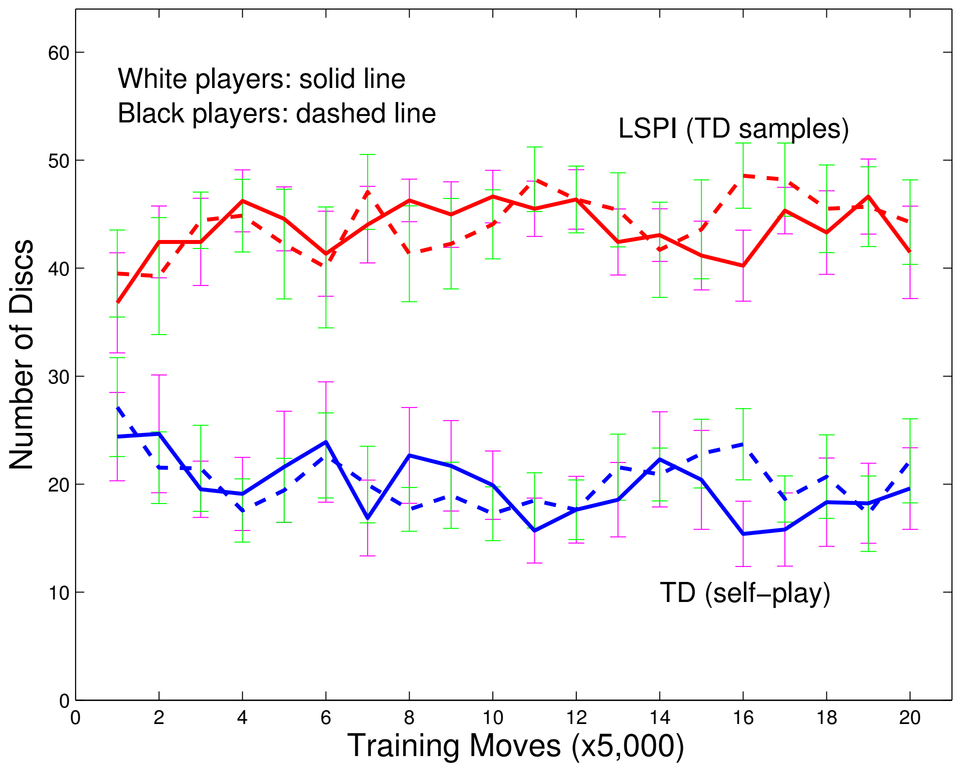


Figure 6.3: Benchmarking TD (self-play), LSPI (self-play), and LSPI (TD samples): Average game score and number of victories (20K).

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 6.4 and Figure 6.5). Additionally, all three players are tested against the same set of 5 benchmark players in terms of game score and number of victories (Figure 6.6). 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.



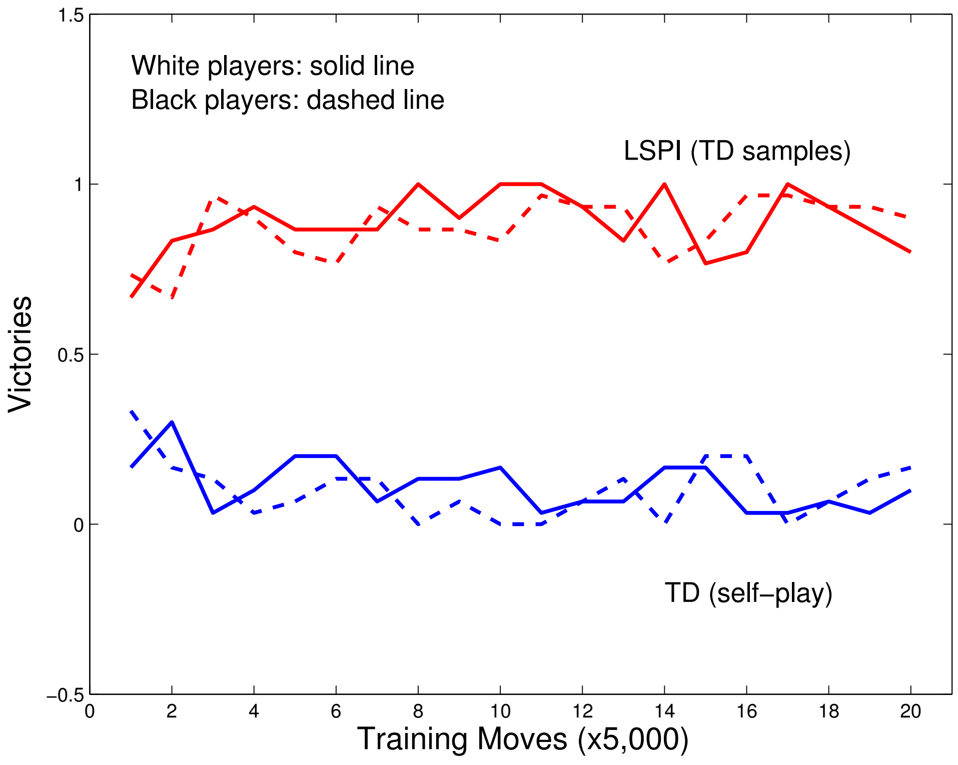
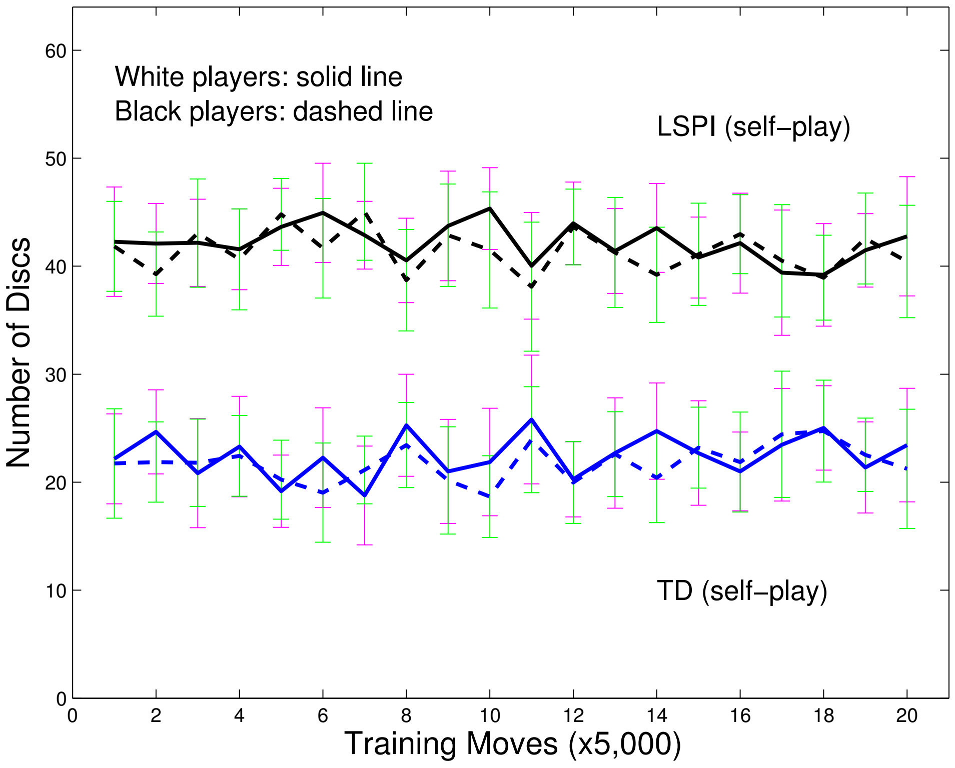


Figure 6.4: TD (self-play) vs. LSPI (TD samples): Average number of discs and number of victories (100K).



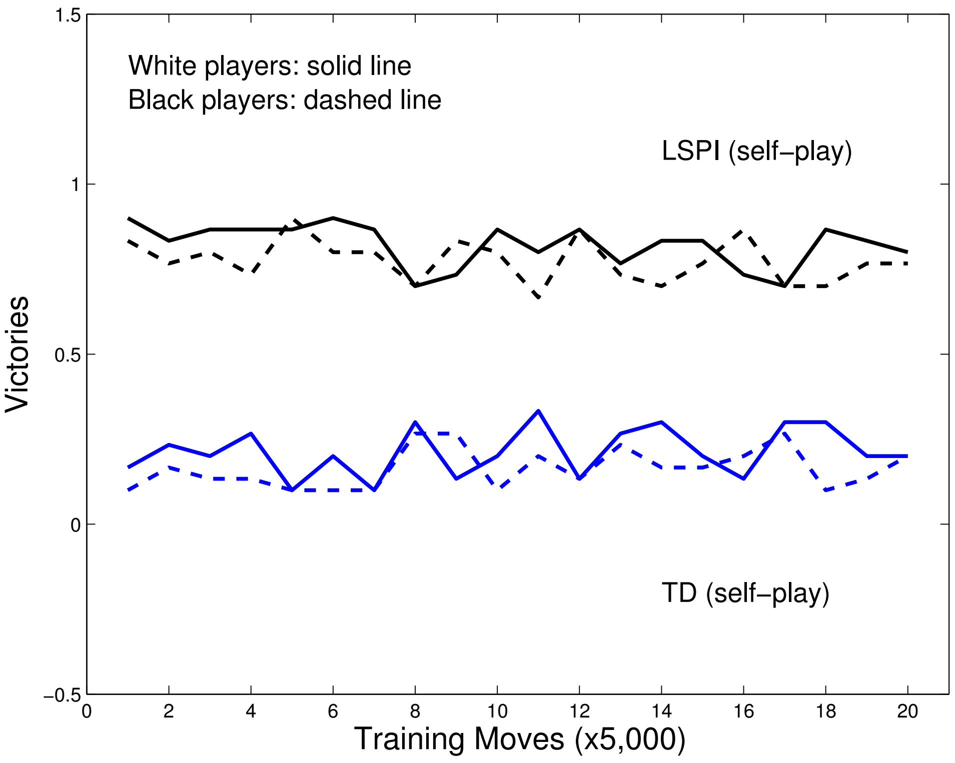
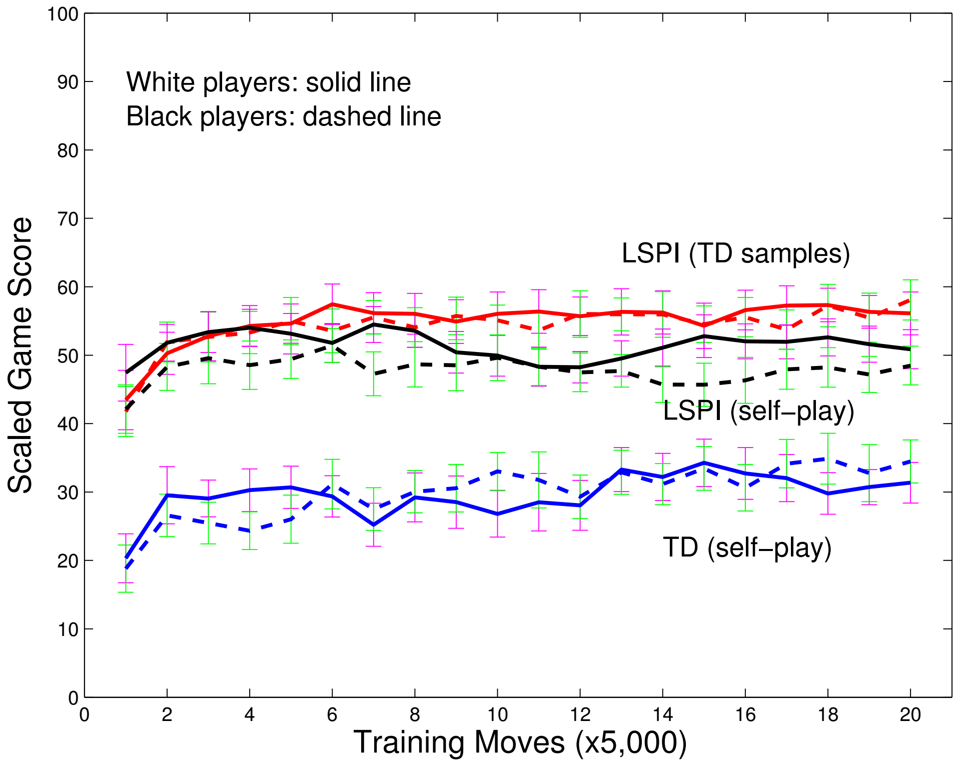


Figure 6.5: TD (self-play) vs. LSPI (self-play): Average number of discs and number of victories (100K).



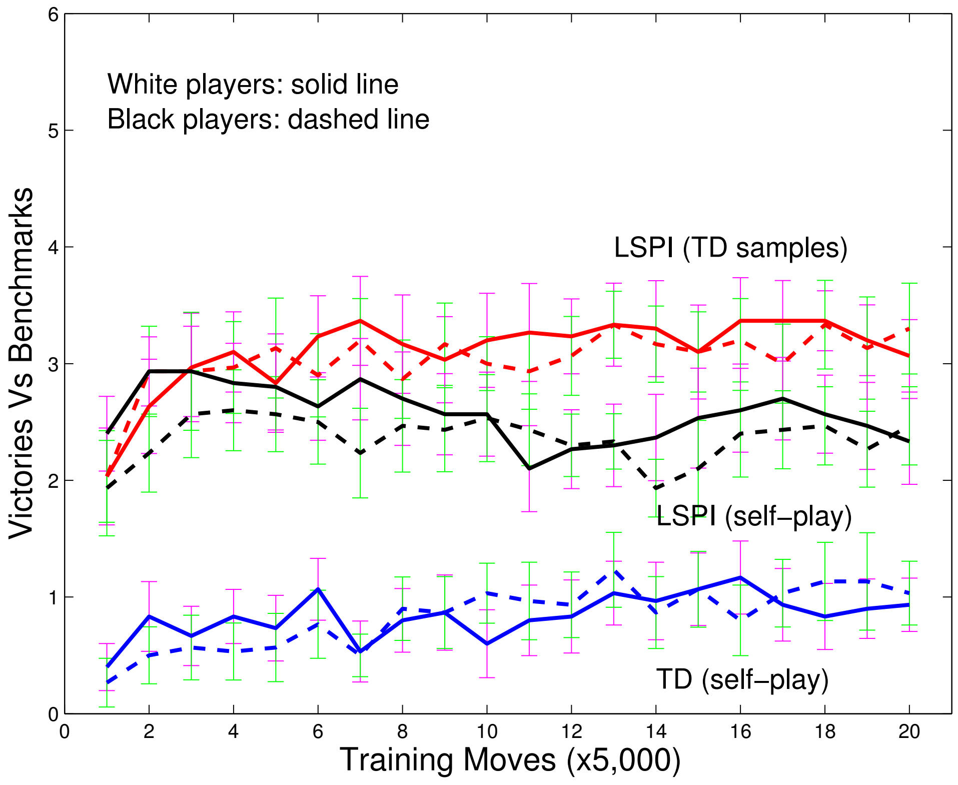


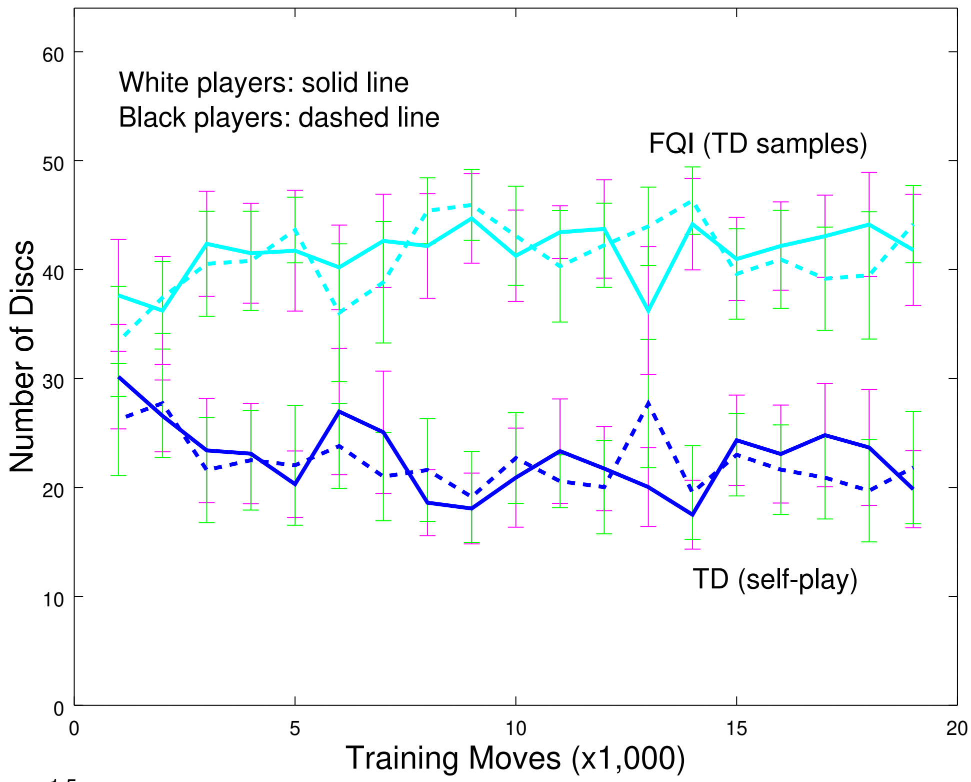
Figure 6.6: Benchmarking TD (self-play), LSPI (self-play), and LSPI (TD samples): Average game score and number of victories (100K).

6.3 FQI Comparison

Another approach, that also avoids issues introduced by the TD algorithm, is to use FQI. To draw meaningful conclusions, we implemented the FQI algorithm and, while using the same methodology, utilized the same sample set we acquired after running TD, along with the same basis functions our LSPI agent uses. Since we are using the samples collected by TD, we named this agent FQI (TD samples).

As before, the resulting agent is compared every 1000 moves of training samples by playing a two-game tournament against TD (self-play) and LSPI (TD samples). Figure 6.7 shows the results of these tournaments against TD (self-play) in terms of number of discs possessed when game finishes along with the average number of victories each of those two agents achieved. Additionally, Figure 6.8 shows the same results for the tournaments between FQI (TD samples) and LSPI (TD samples). Apart from these tournaments, our FQI (TD samples) player, played twice (both sides) against the 5 fixed benchmark agents we used before, and the results are presented in terms of game score and number of victories (Figure 6.9). For these experiments we used the samples collected from the 20K moves run of the TD algorithm. Every experiment was 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.

As it is evident from these experimental results, FQI (TD samples), while utilizing the same sample set, outperforms TD (self-play), but achieves lower (albeit close) performance compared to the player developed by the LSPI algorithm. This superiority over TD (self-play), reinforces our position that, learning by making small incremental changes and being dependent on the value function, is far from being an optimal approach. It has to be noted that usage of FQI not only resulted in worse performance compared to LSPI (TD samples), but the algorithm also needed a lot more time and many more iterations in order to converge.



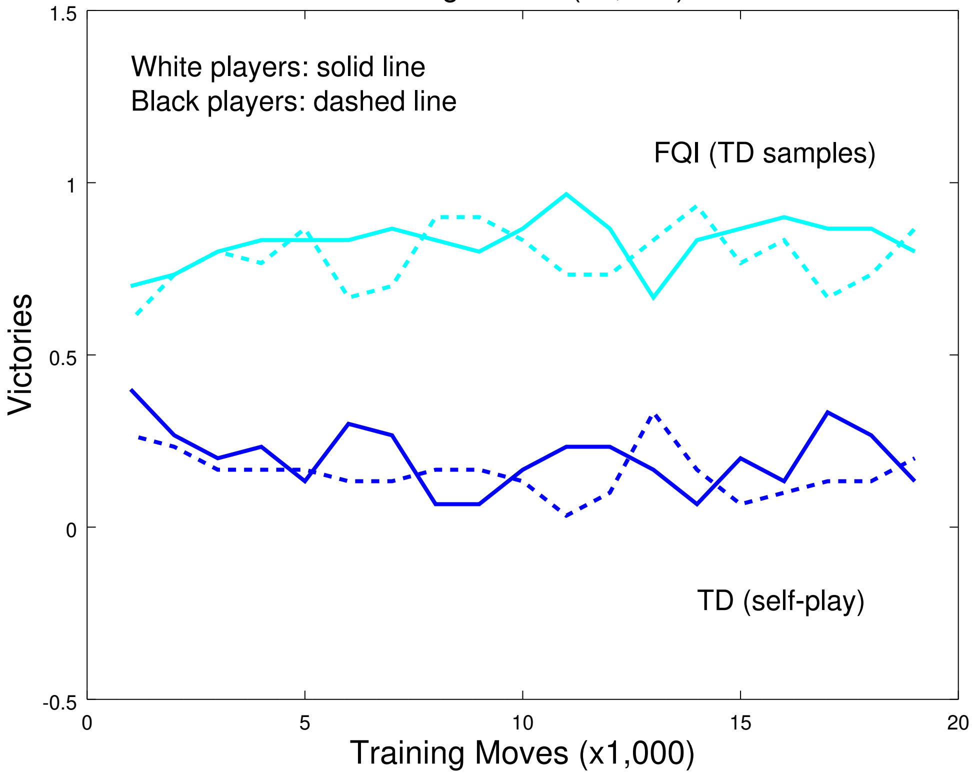
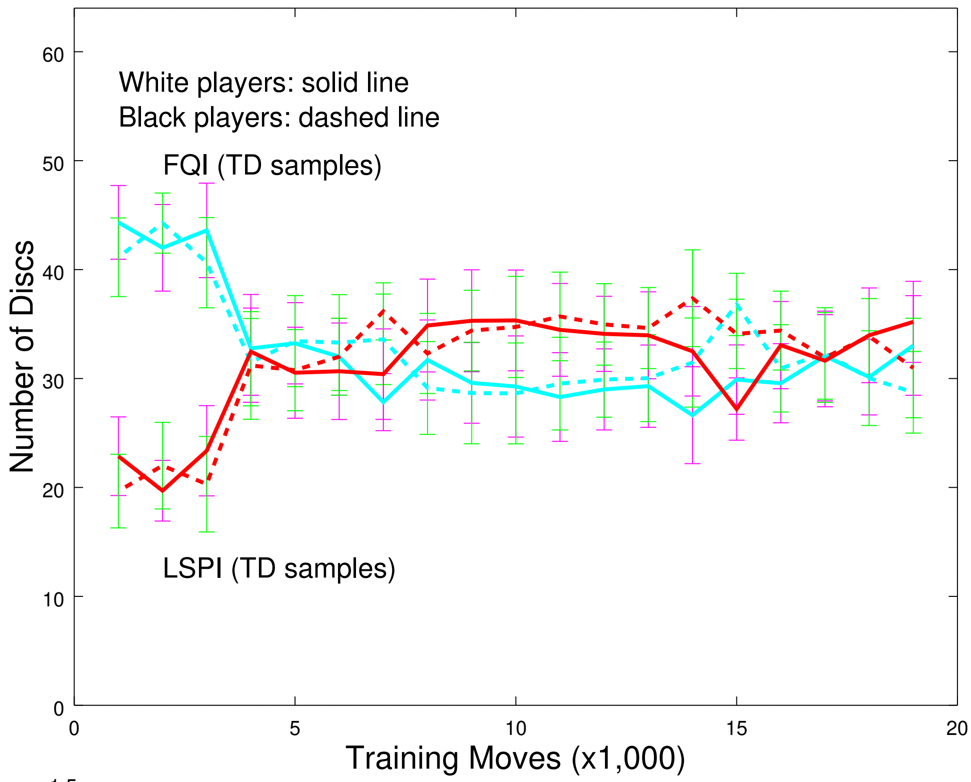


Figure 6.7: TD (self-play) vs. FQI (TD samples): Average number of discs and number of victories (20K).



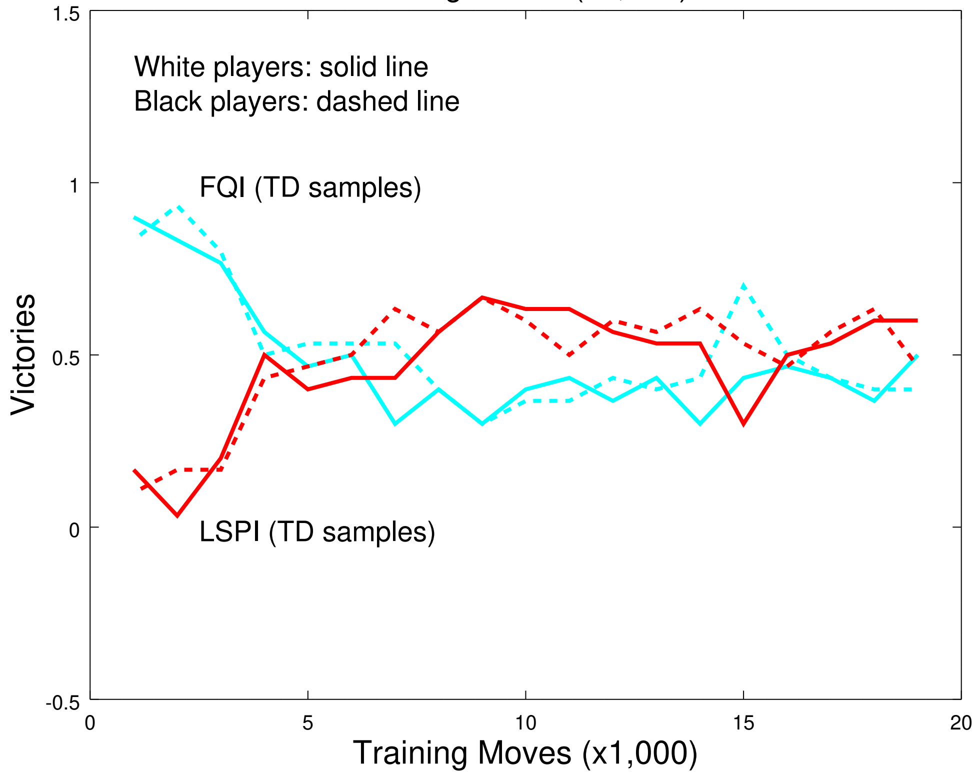
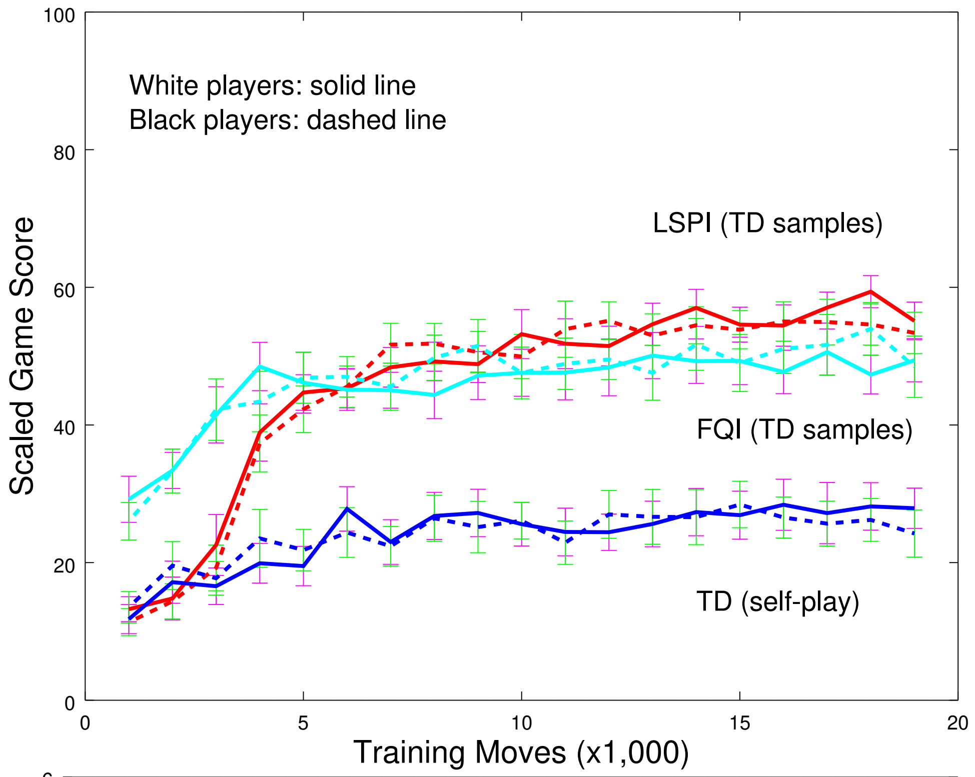


Figure 6.8: LSPI (TD samples) vs. FQI (TD samples): Average number of discs and number of victories (20K).



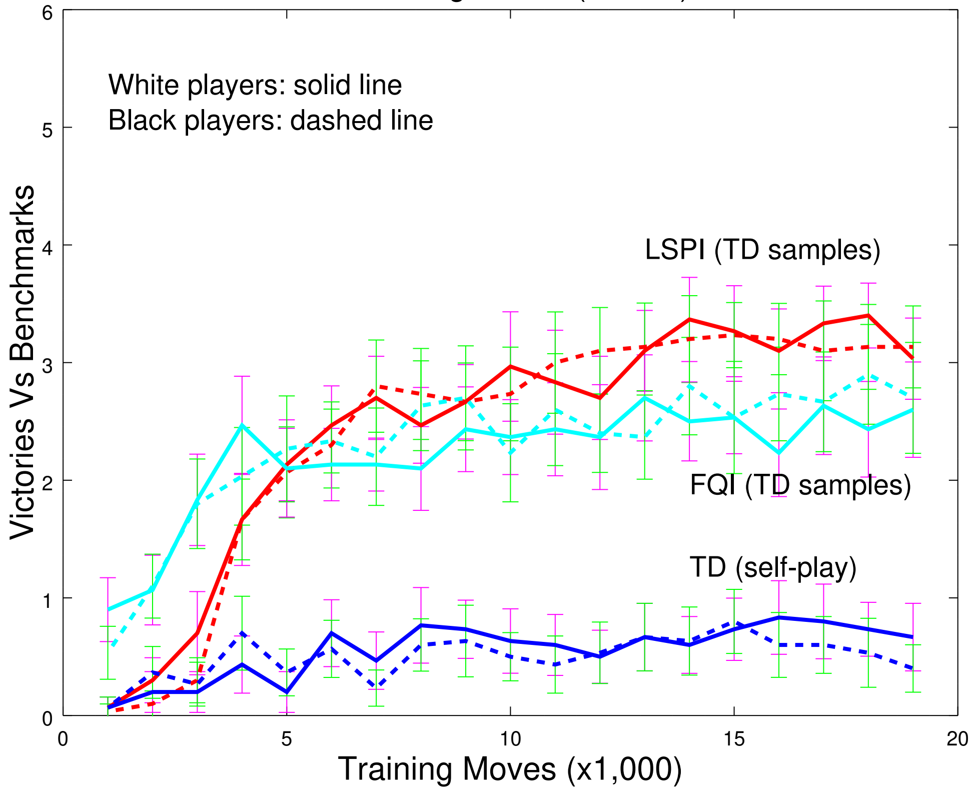


Figure 6.9: Benchmarking TD (self-play), LSPI (TD samples), and FQI (TD samples): Average game score and number of victories (20K).

6.4 Comparison with Model-Based

Wanting to measure the impact a model-based approach would have in our game, we increased our sample set by generating all legal (a, o) move pairs for any given state s we previously had. This increase in our sample set made convergence difficult. To combat this, we decided to lower the discount factor (γ) to 0.9. The resulted player was named LSPI (Model-Based).

As it was the case with the previous experiments, we used the same 20K moves sample set and the same basis functions. Every 1000 moves of training samples, our new agent has two-game tournaments against LSPI (TD samples), TD (self-play) and our benchmark players. Results are presented in terms of number of discs possessed along with the average number of victories each player achieved against its opponent. Figure 6.10 shows the results of those tournaments between LSPI (Model-Based) and TD (self-play), while Figure 6.11 shows the results between LSPI (Model-Based) and LSPI (TD samples). Additionally, the tournament results against our 5 benchmark players can be seen in Figure 6.12. 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.

Our results show that our new player, LSPI (Model-Based), has a good start and finds good policies with small sample sets, but cannot reach better ones afterwards when it gains access to more samples. Although LSPI (Model-Based) gets a head start against its opponents and it demonstrates good performance against TD (self-play), it cannot surpass LSPI (TD samples), and ends up being a bit worse player (albeit not by much). In conclusion, this approach is slower than our plain algorithm, and there is no guarantee that it will lead to a stronger agent. If we deal with a small sample set though, it is definitely a valid option.

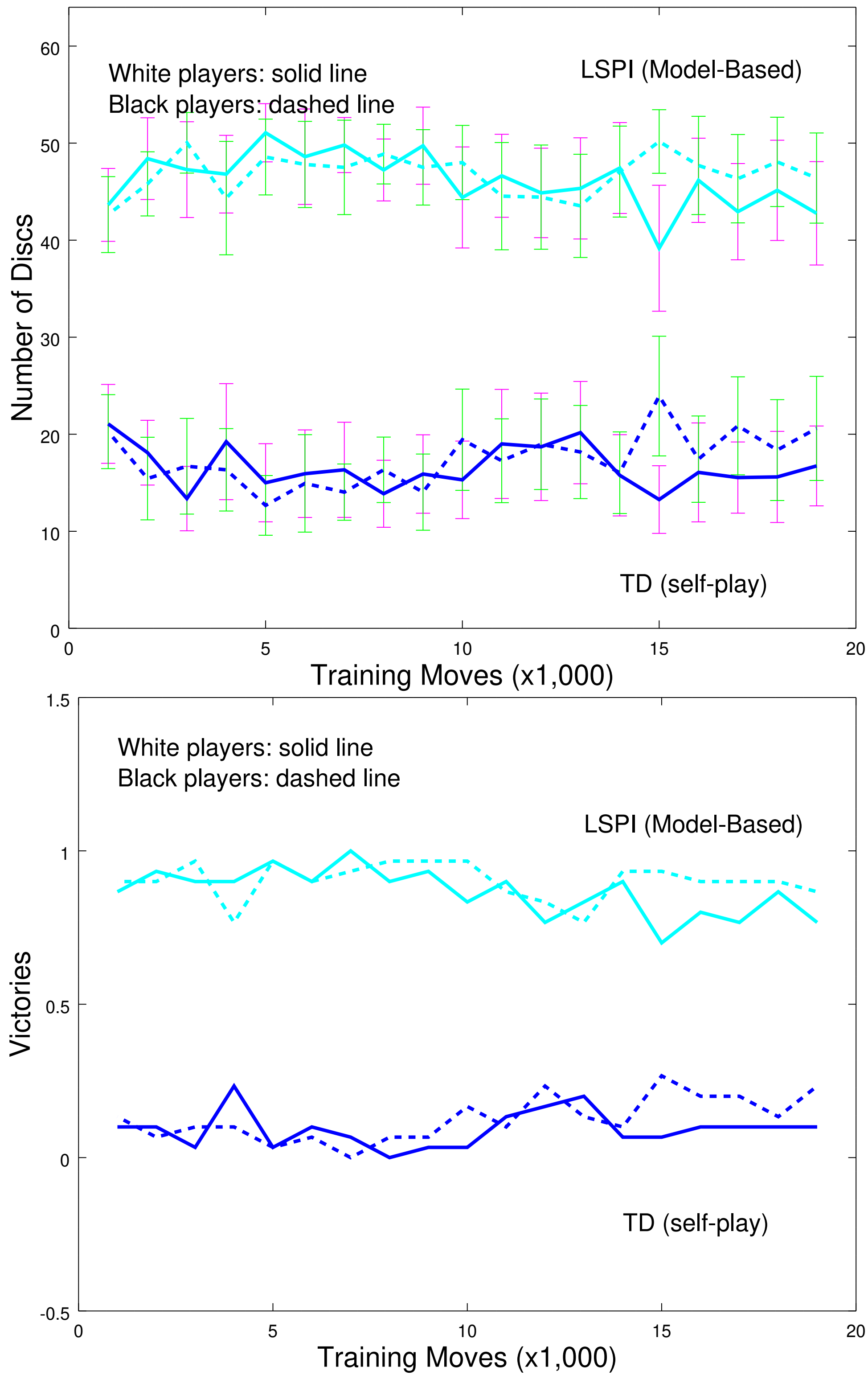
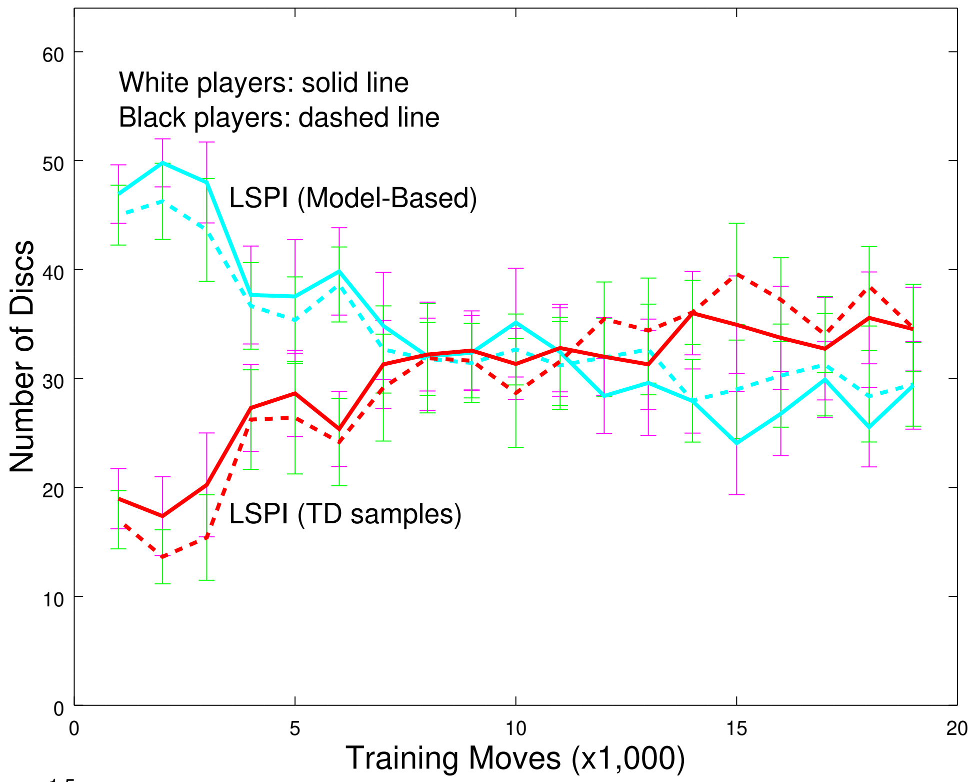


Figure 6.10: TD (self-play) vs. LSPI (Model-Based): Average number of discs and number of victories (20K).



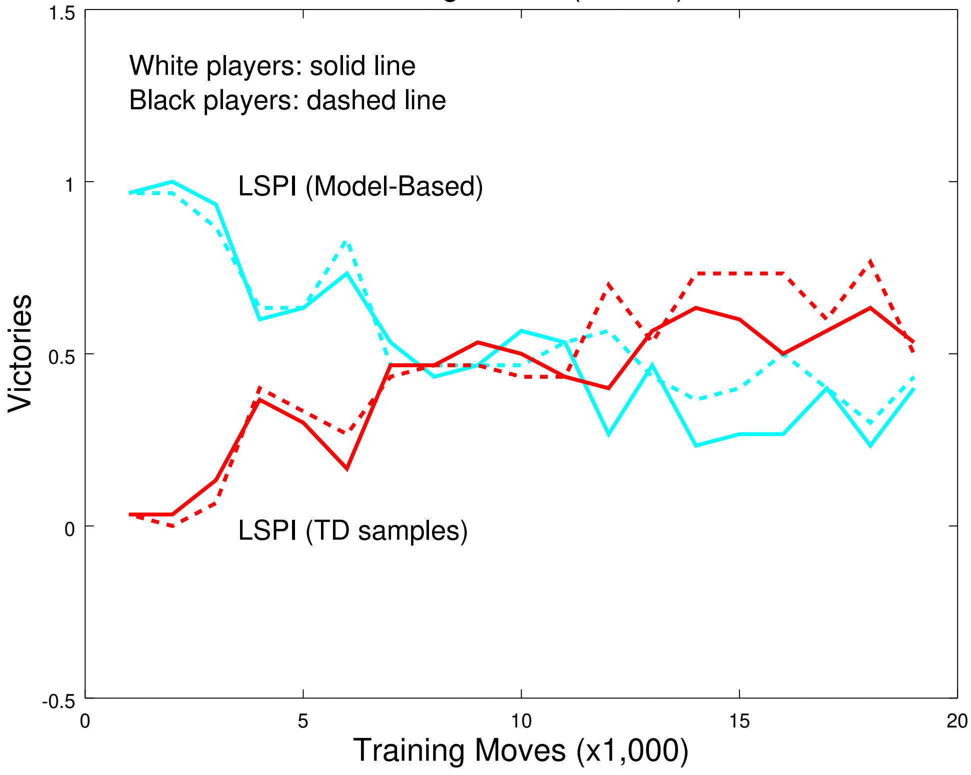
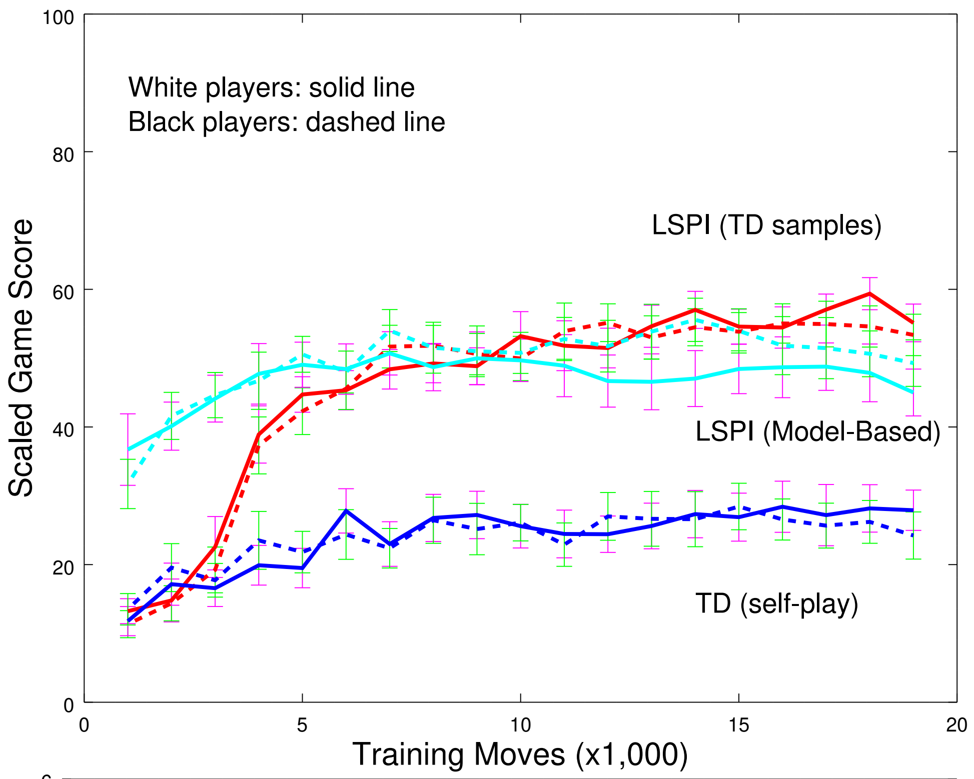


Figure 6.11: LSPI (TD samples) vs. LSPI (Model-Based): Average number of discs and number of victories (20K).



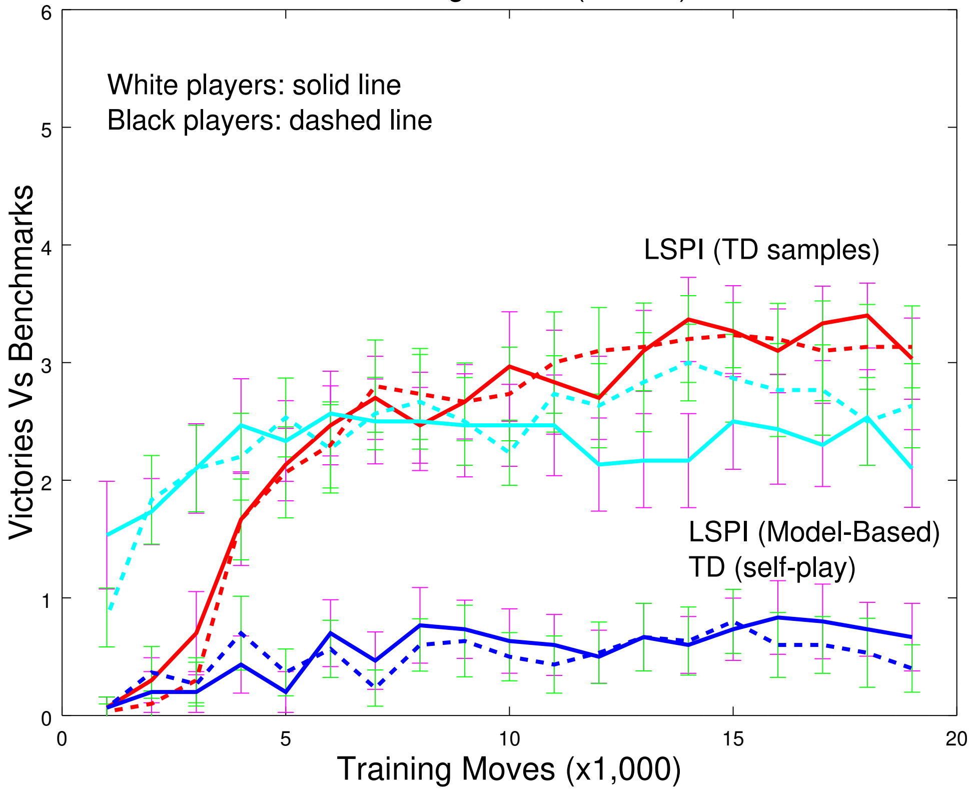


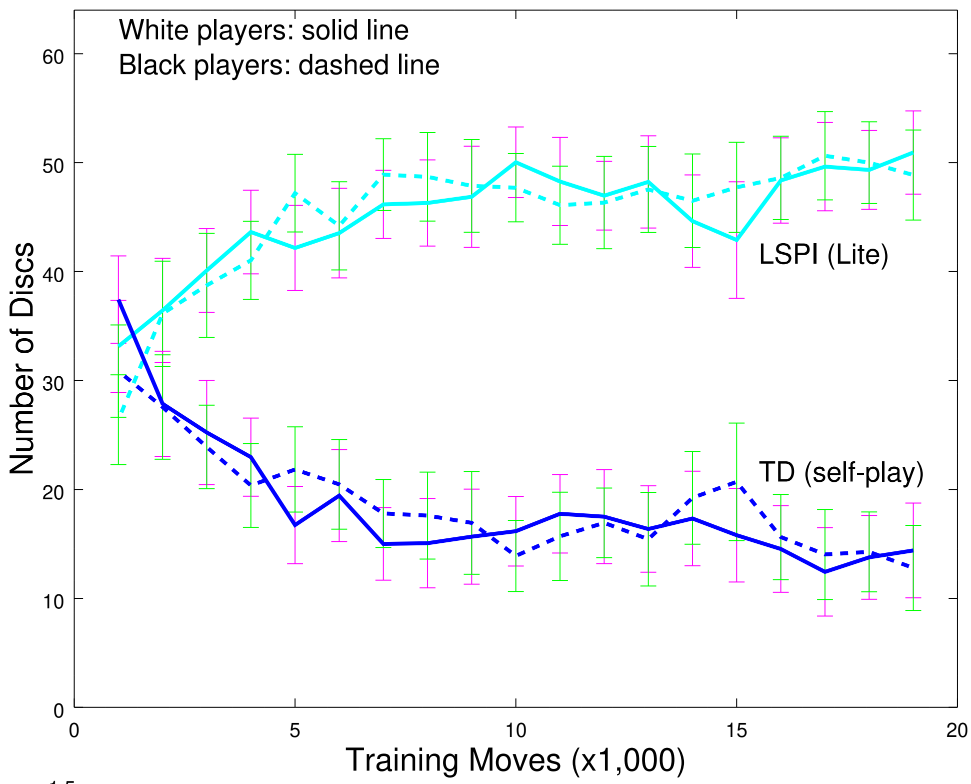
Figure 6.12: Benchmarking TD (self-play), LSPI (TD samples), and LSPI (Model-Based): Average game score and number of victories (20K).

6.5 LSPI (Lite)

The modifications that the LSPI-Lite introduces result in a faster learning process. The basis functions that refer to the third state were removed and our search became shalower. Unfortunately though, problems with convergence were encountered with our LSPI-Lite algorithm as well. In order to overcome them the discount factor (γ) had to be set to 0.95.

We are still using the same sample set of 20K moves along with the same methodology. Every 1000 moves our player called LSPI (Lite) is compared against TD (self-play), LSPI (TD samples) and the 5 benchmark players from before. Results are once again presented in terms of discs possessed and average number of victories. Figure 6.13 shows tournament results between LSPI (Lite) and TD (self-play), while Figure 6.14 shows tournament results between LSPI (Lite) and LSPI (TD samples). Finally, Figure 6.15 shows the results against the benchmark players. 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.

Our main goal was to calculate the expected performance drop after the modifications introduced. As we can see our new player excels versus TD (self-play) and achieves similar performance with LSPI (TD samples). The small difference between LSPI (Lite) and LSPI (TD samples) is better observed when the two players face our benchmark players. This small performance drop might be acceptable when we have to deal with environments with a high branching factor, since we gain a lot in terms of speed. The speed improvement that this algorithm brings to the table should not be underestimated, since it can allow us to use LSPI in cases where it would have been prohibitively expensive.



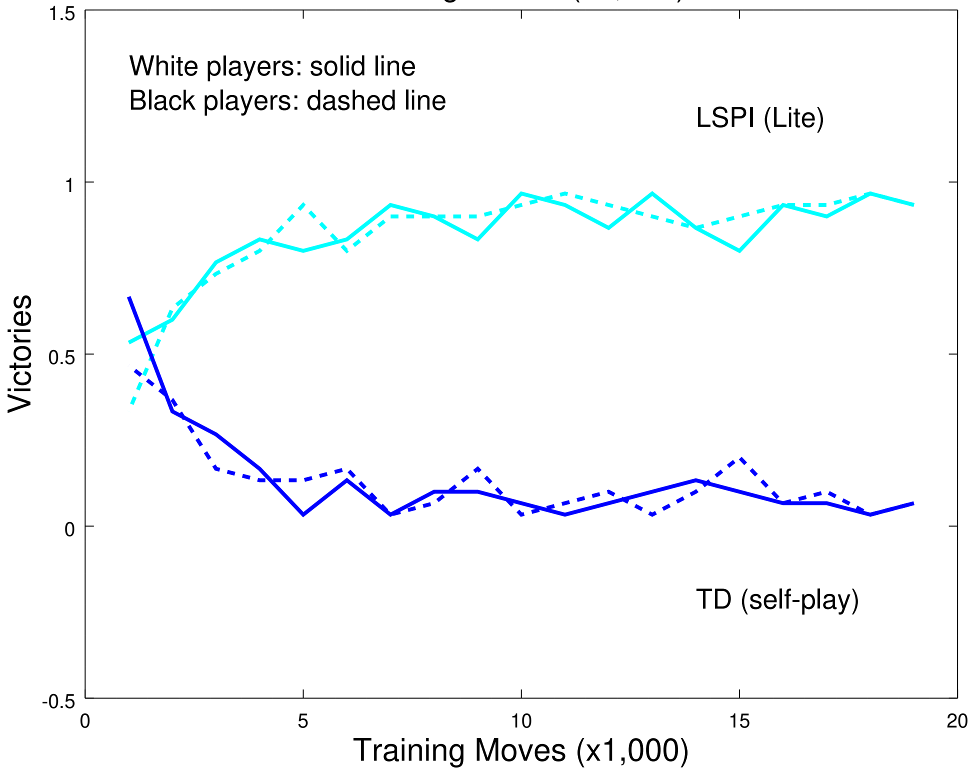
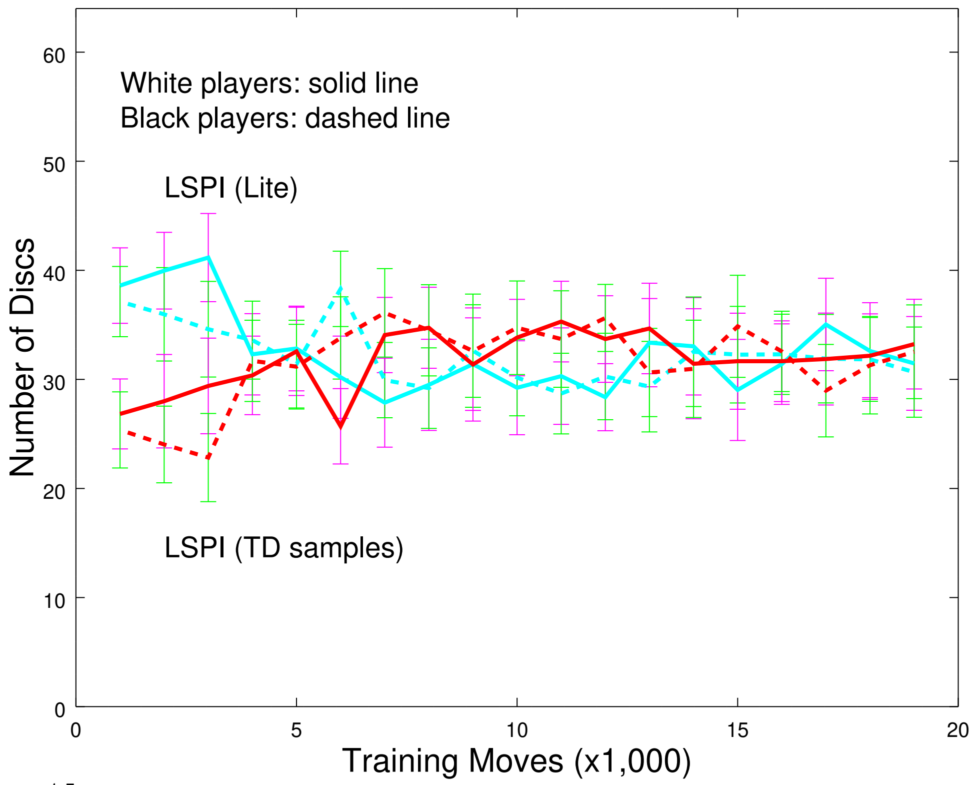


Figure 6.13: TD (self-play) vs. LSPI (Lite): Average number of discs and number of victories (20K).



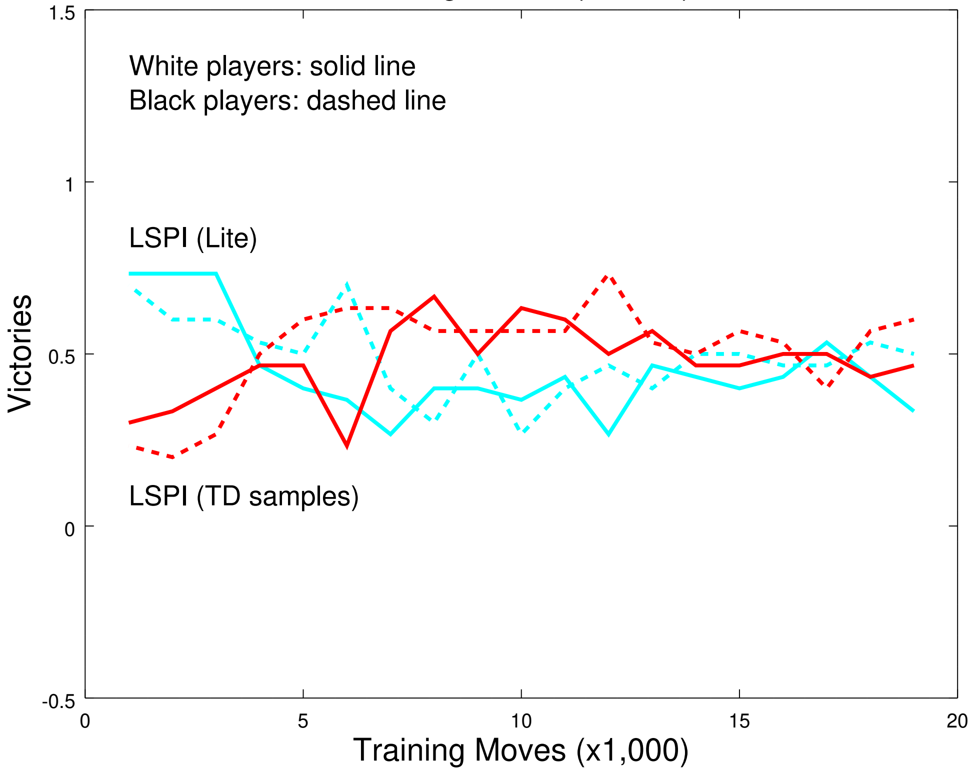
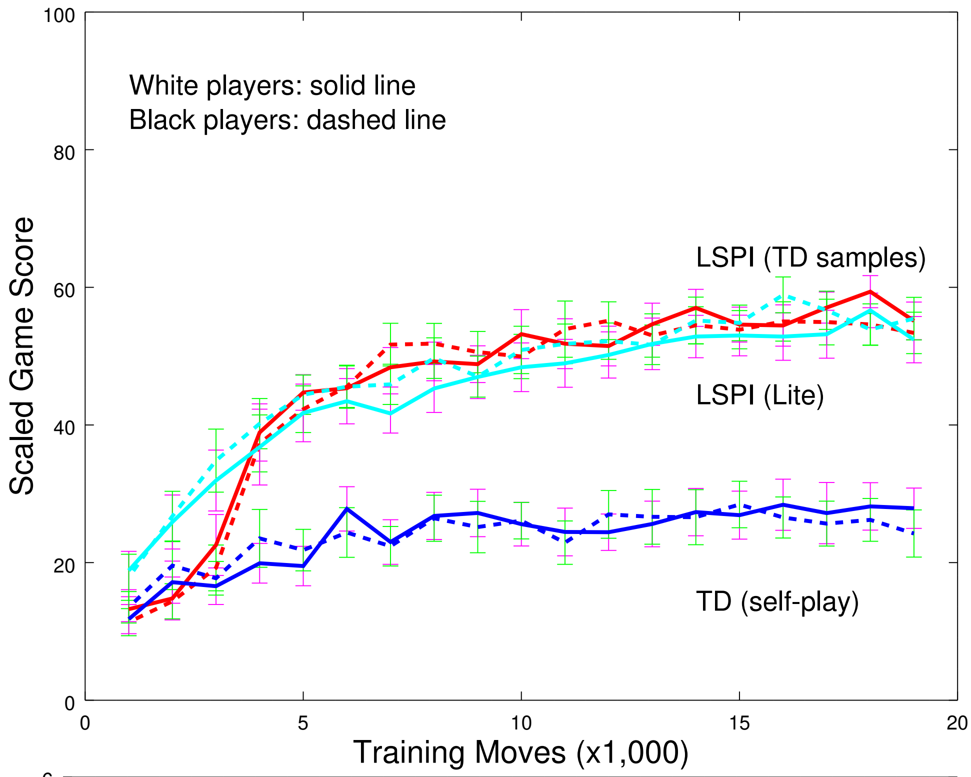


Figure 6.14: LSPI (TD samples) vs. LSPI (Lite): Average number of discs and number of victories (20K).



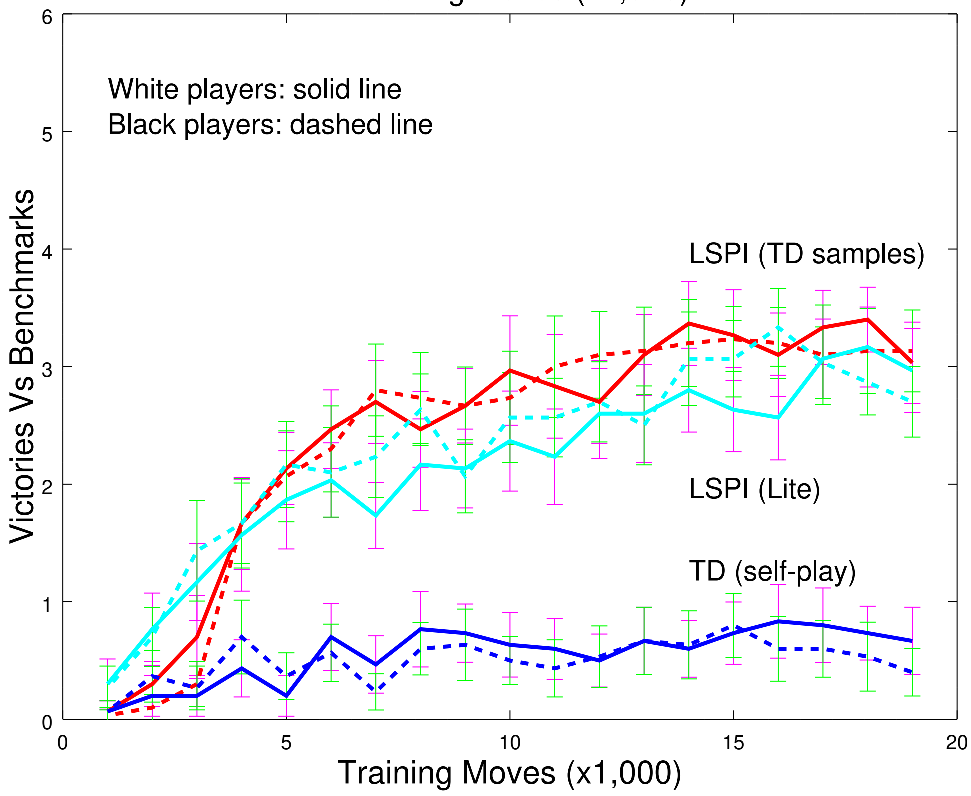


Figure 6.15: Benchmarking TD (self-play), LSPI (TD samples), and LSPI (Lite): Average game score and number of victories (20K).

6.6 LSPI (Lite Dual)

The last algorithm whose performance we are going to demonstrate in a deterministic environment is LSPI (Lite Dual). We follow the same approach as with LSPI (Lite), but this time we parse every sample twice. It should be noted that this time we encountered no issues with convergence.

We are utilizing our sample set of 20K moves and we are using the same methodology. Every 1000 moves our player called LSPI (Lite Dual) is compared against TD (self-play), LSPI (TD samples) and the 5 benchmark players from before. Results are once again presented in terms of discs possessed and average number of victories. Figure 6.16 shows tournament results between LSPI (Lite Dual) and TD (self-play), while Figure 6.17 shows tournament results between LSPI (Lite Dual) and LSPI (TD samples). Finally, Figure 6.18 shows the results against the benchmark players. 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.

As we can see from the results, performance is better than the one TD (self-play) offers, but still below the levels we reached with our previous attempts. It is important to remember that, in this specific environment this approach may have scored below LSPI (self-play) and LSPI (Lite), but this is not necessarily a universal truth, since games vary a lot and in many ways.

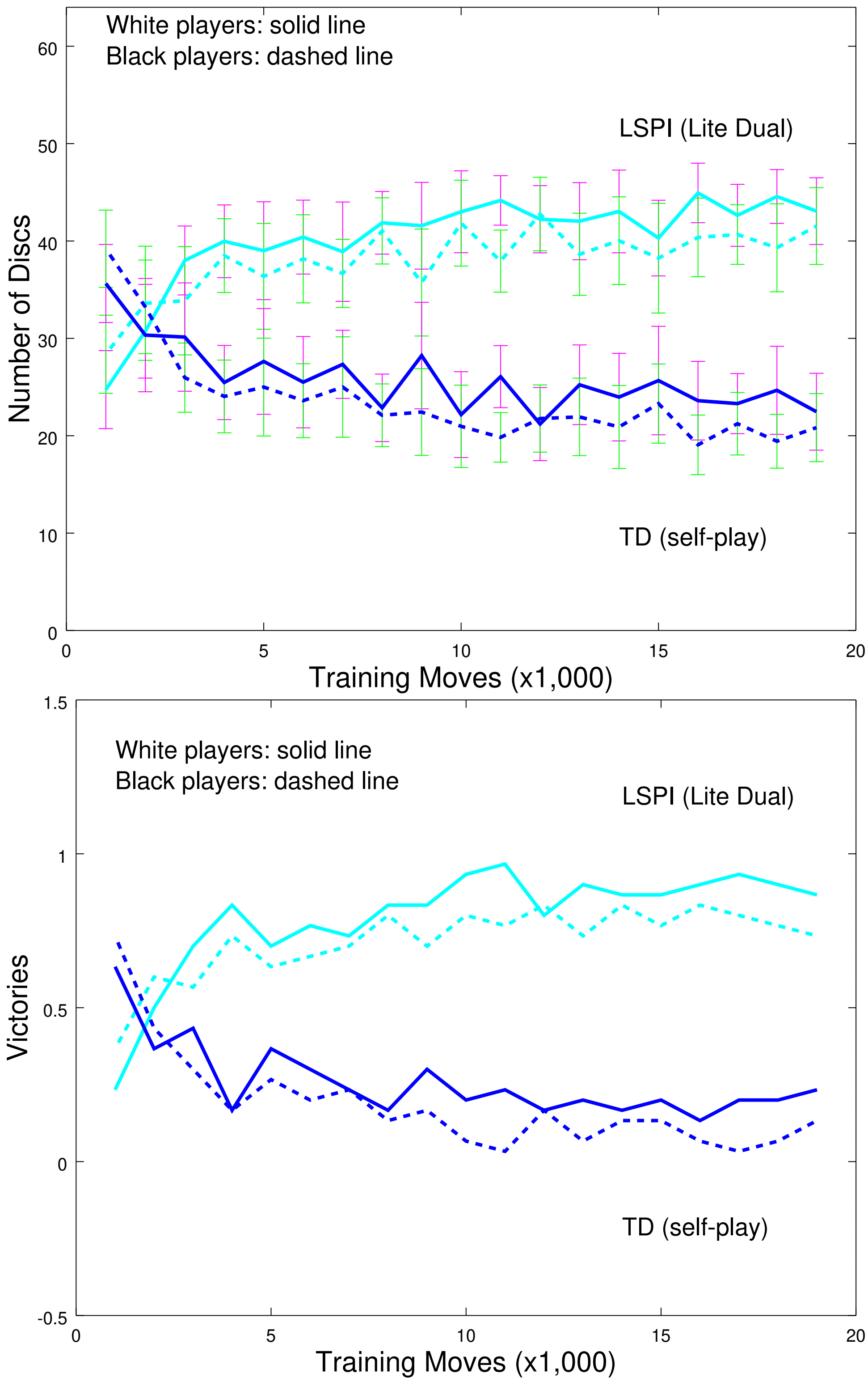
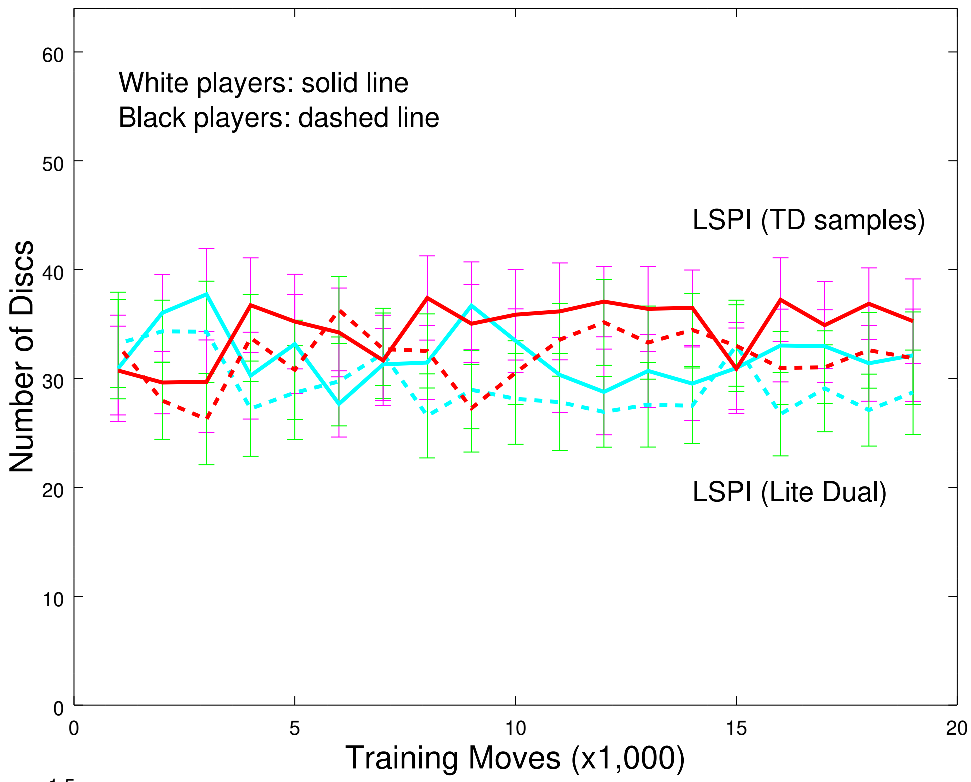


Figure 6.16: TD (self-play) vs. LSPI (Lite Dual): Average number of discs and number of victories (20K).



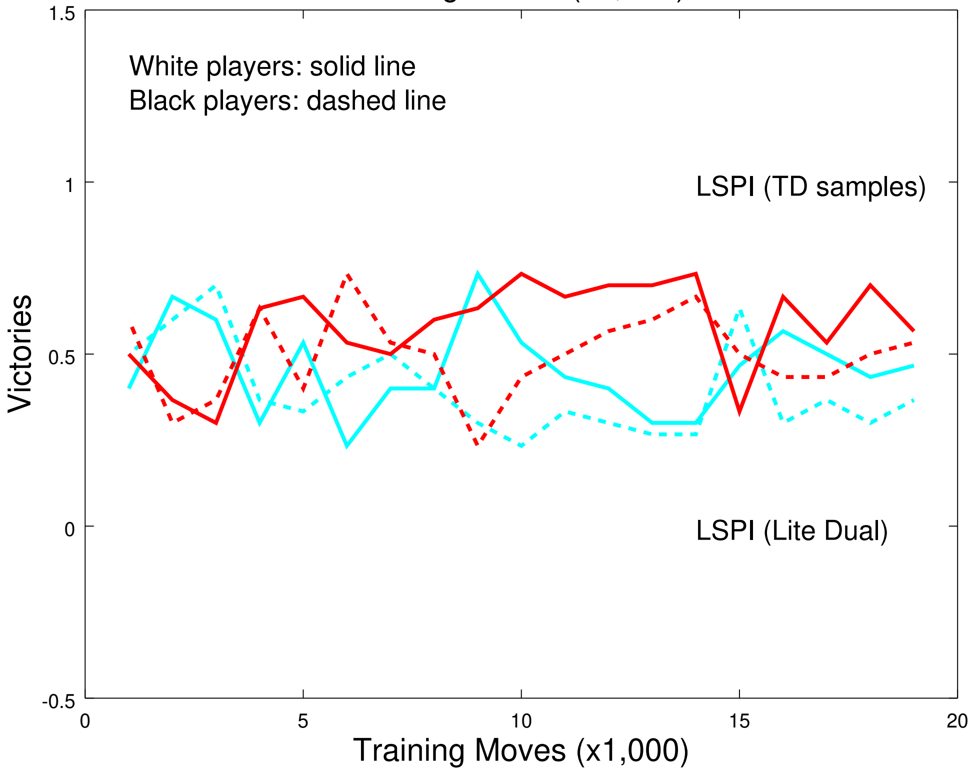
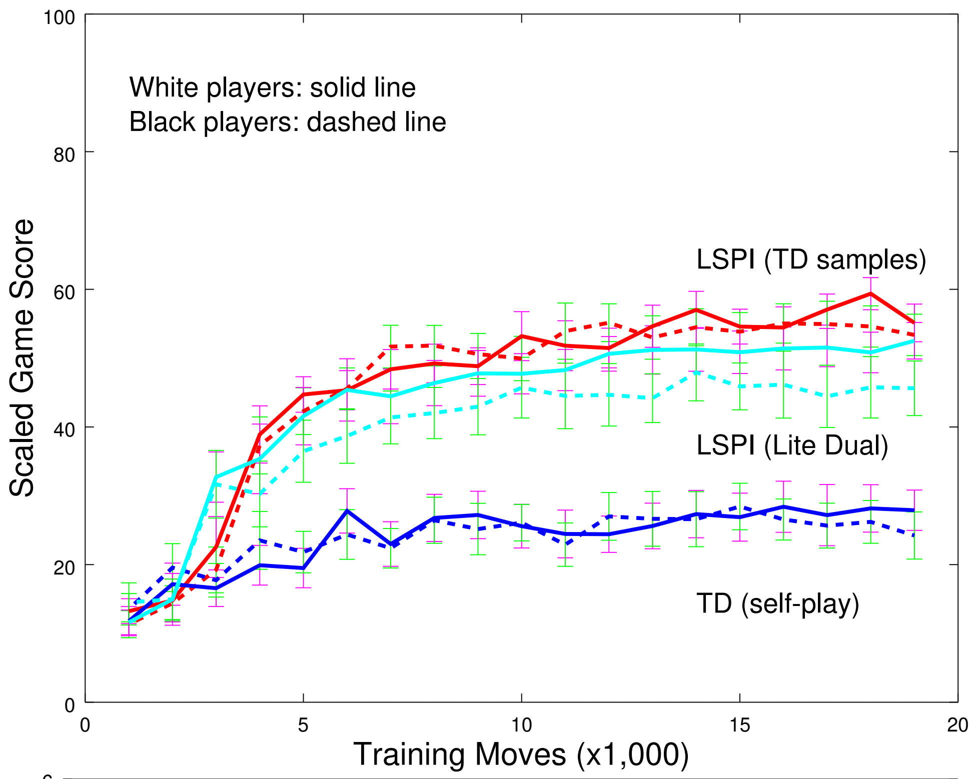


Figure 6.17: LSPI (TD samples) vs. LSPI (Lite Dual): Average number of discs and number of victories (20K).



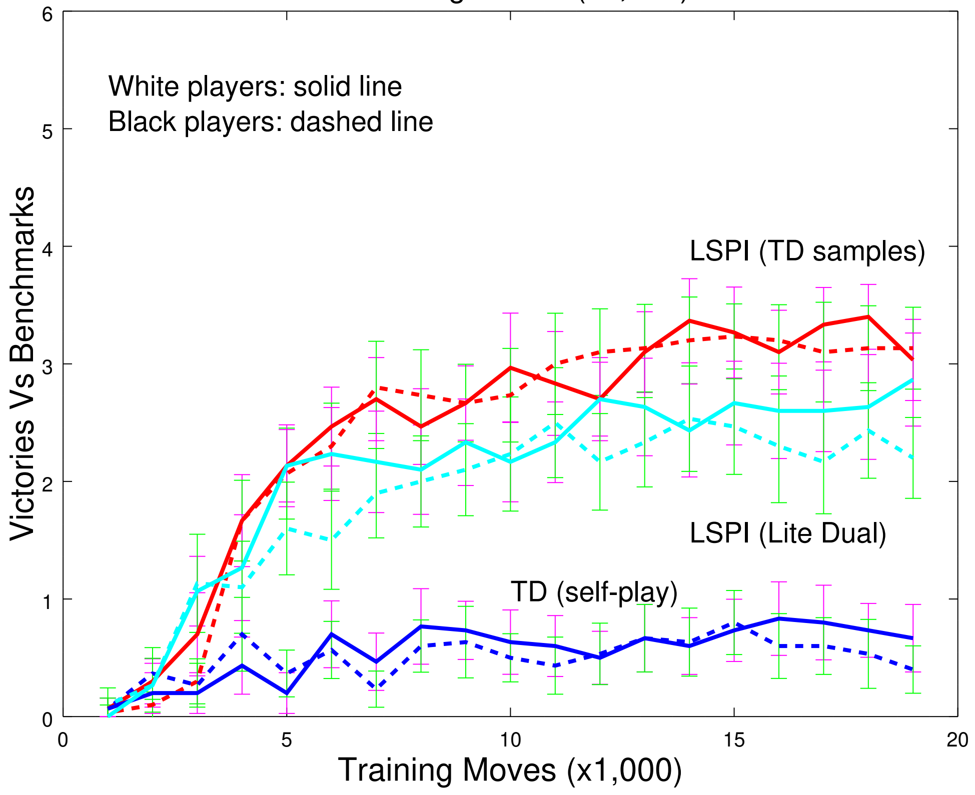


Figure 6.18: Benchmarking TD (self-play), LSPI (TD samples), and LSPI (Lite Dual): Average game score and number of victories (20K).

6.7 Comparisons in a Stochastic Environment

In Chapter 4 we proposed four LSPI-based algorithms, and although all can be used in games with high branching factor values or stochastic components, that is not always viable or practical. In the game of Backgammon we decided to investigate the usage of LSPI (Lite) and LSPI (Lite Dual), specifically because of their speed compared to our first two approaches. Although our first attempt was to use LSPI (Lite), the algorithm failed to converge even with a lowered discount factor. Subsequently, we continued our investigation using the LSPI (Lite Dual) algorithm.

To compare our adaptation of LSPI for stochastic adversarial games against TD, we are going to use the same methodology with some minor changes. The decision to clone our opponent or not, depends on the results of a 3-game tournament. In order for a cloning to occur, our learning player has to demonstrate his abilities by wining at least 2 games out of 3. Furthermore, because of the existence of the stochastic element, we decided to gather more samples than before. We collected a sample set of 1000000 moves. We conduct our tournaments every 100000 moves and during the same intervals our LSPI and TD players face each other, along with Tesauro's "pubeval" player in a series of 100 games.

We decided to present our results in terms of two values. The first one is the number of game points a player gains. Every player is awarded a single point for a victory or two points for a double victory (a gammon) and an equivalent number of points are subtracted from the losers. The second value that interests us is the win-rate each player achieved against their opponents. In Figure 6.19 we can see the results against "pubeval" in terms of game points won and percentage of games won (win-rate). Our LSPI player's performance against TD can be seen in Figure 6.20 in terms of game points and percentage of games won. All these experiments were repeated 8 times to obtain averages over different sets of samples and an indication of statistical significance shown by the 95% confidence intervals on the graphs.

As it is evident from our experimental results, TD and LSPI cannot compete with "pubeval". That is expected and it is mainly due to our small evaluation feature set. Looking at the results collected when our two players faced each other, we can see that LSPI has a slightly better performance than TD, which results in an about 60% win-rate against its opponent.

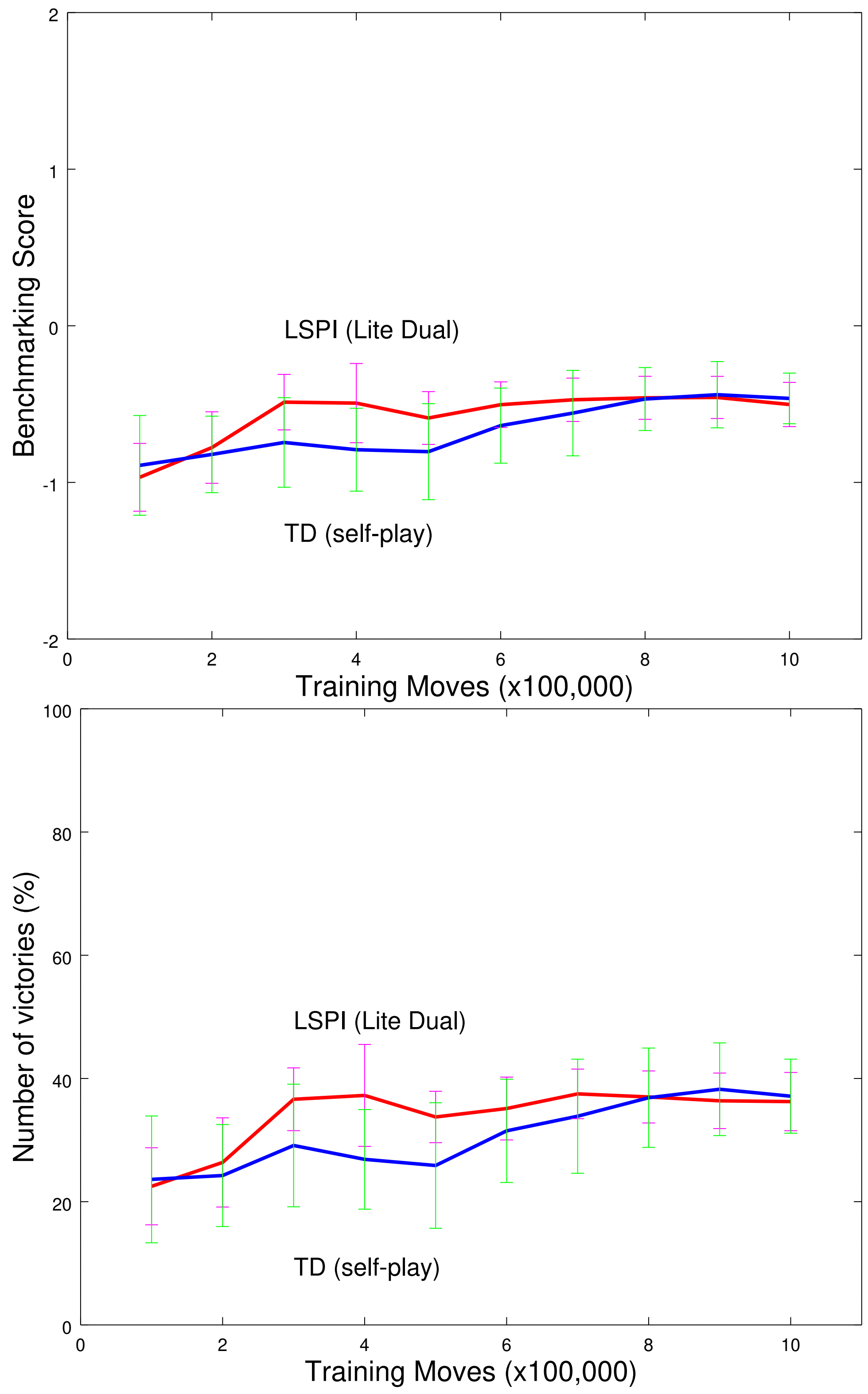
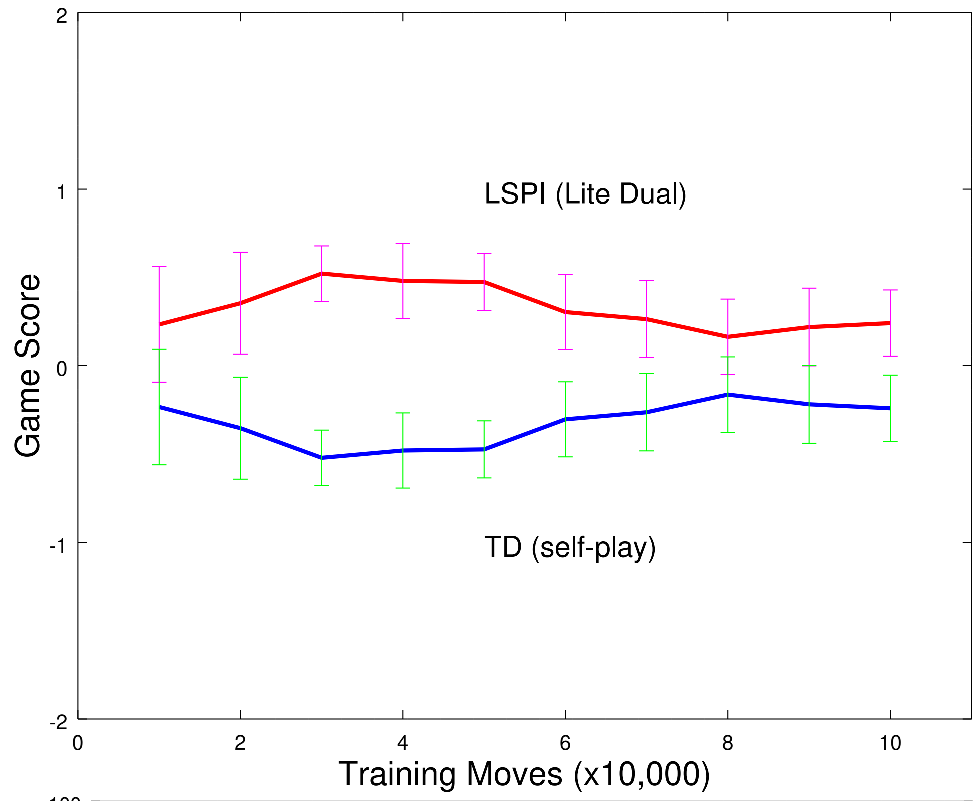


Figure 6.19: Benchmarking TD (self-play) and LSPI (Lite Dual) : Game points and win-rate against "pubeval" (1000K).



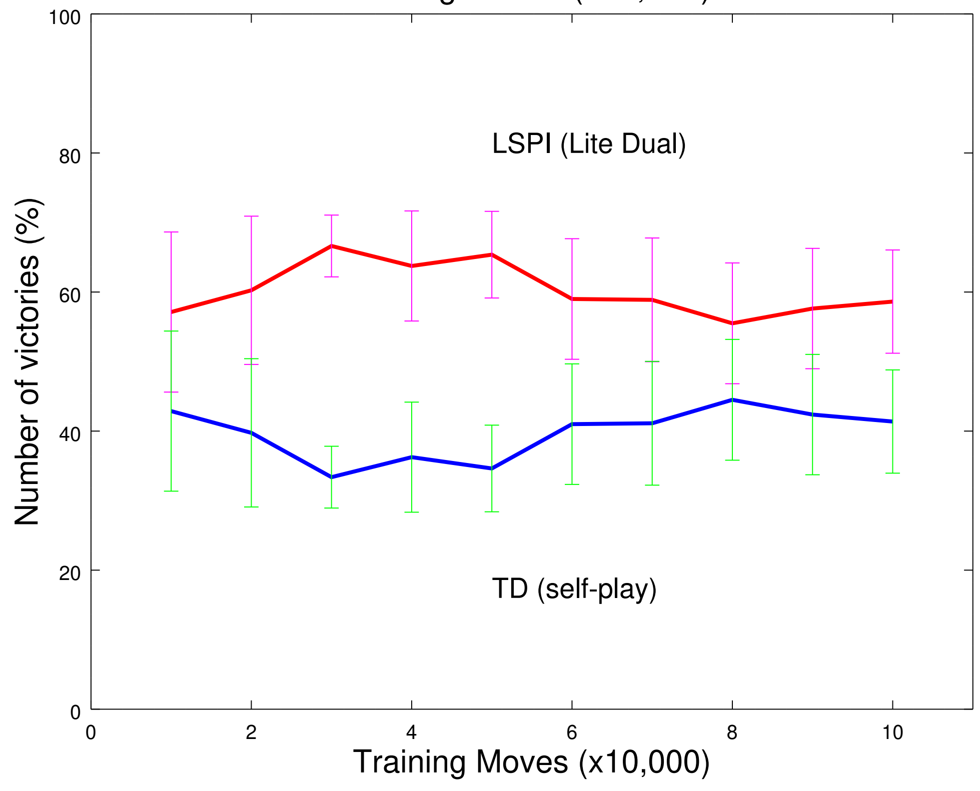


Figure 6.20: LSPI (Lite Dual) vs. TD (self-play): LSPI's game points and win-rate against TD (1000K).

6.8 Backgammon Race Comparison

Although the difference in performance between LSPI and TD is noticeable in the previous section, when the two players faced each other, we were unabled to demonstrate LSPI's advantage when both players compared against "pubeval". This is mainly due to the fact that our small feature set, as expected, cannot lead to players strong enough to compete with "pubeval".

Believing that our evaluation feature set for the race part of the game is sufficient enough to give us a good understanding of the board, we decided to compare our players using only boards belonging to that part. We created random boards that belong to the "race" part of the game, and then we used them to train TD, while also saving the samples like before. This time we trained our players using 100000 moves in total, as this part of the game is not as complex. Every 10000 moves we conduct 1000 game tournaments between LSPI (Lite Dual), TD (self-play) and "pubeval". These tournaments cannot be full games like before. Instead we use a tournament board that has all player's pieces placed 12 positions before the end of each player's board. All these experiments were repeated 10 times to obtain averages over different sets of samples and an indication of statistical significance shown by the 95% confidence intervals on the graphs.

Figure 6.21 shows the results of the tournaments against "pubeval" in terms of game points won and win-rate. The results of the tournaments between our players, LSPI (Lite Dual) and TD (self-play), are shown in Figure 6.22.

Looking at the results we can clearly spot LSPI's superior performance against TD. The difference is evident not only in the tournament results between the two players, where LSPI achieves around 70% win-rate, but also in the results against "pubeval". LSPI is close to parity (breaks even) with "pubeval", while TD still struggles to compete.

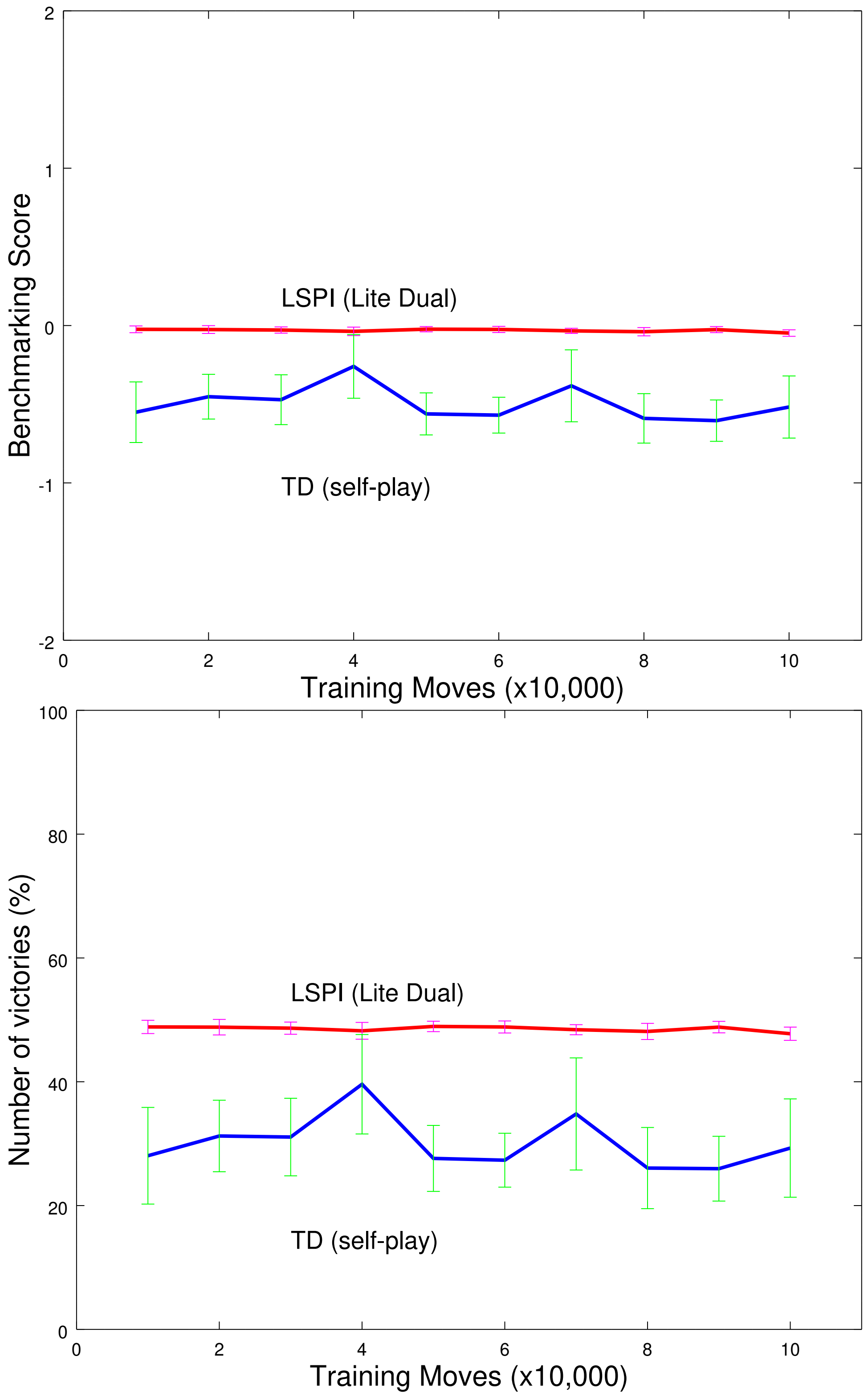
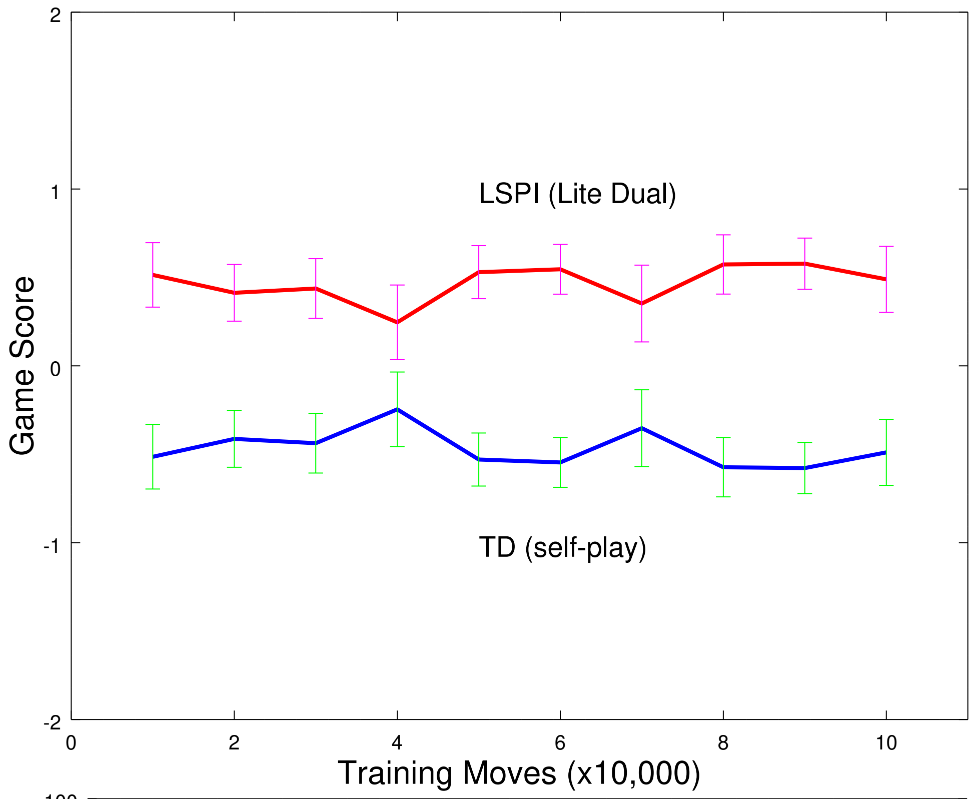


Figure 6.21: Benchmarking TD (self-play) and LSPI (Lite Dual) : Game points and win-rate against “pubeval” (100K) - during “race” stage.



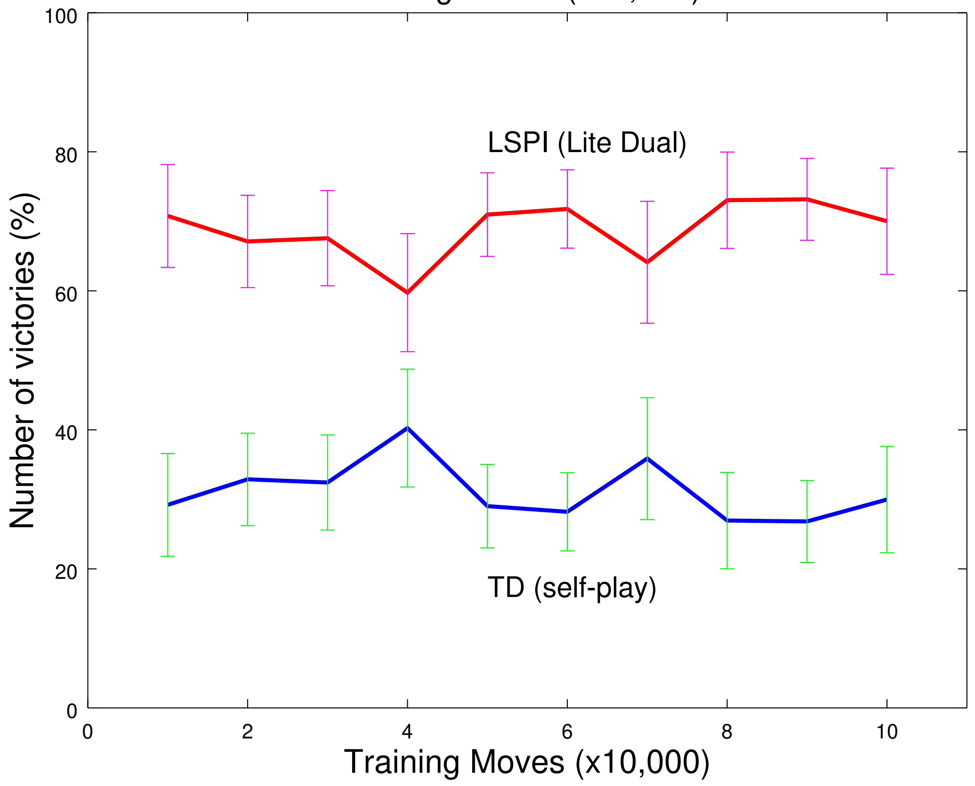


Figure 6.22: LSPI (Lite Dual) vs. TD (self-play): LSPI's game points and win-rate against TD (100K) - during "race" stage.

Chapter 7

Discussion and Future Work

Our goal in this work was not to compete with state-of-the-art agents and specialized systems, but only to demonstrate that the learning abilities of a game-playing agent can benefit from modern reinforcement learning techniques.

Our proposed methods offer better utilization of samples and exhibit 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 $O(m^3)$, where m 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 $O(m^2)$ [9].

Given the diversity of game mechanics and environments, it is to be expected that some algorithms are going to yield better results in certain games than others. Specific aspects of the game can make convergence hard, or deem an approach too expensive for the benefits it provides, and that was the reason behind our multiple proposals.

All our results demonstrated that our LSPI-based algorithms achieve better performance when compared to TD, when utilizing the information provided while traversing the same path inside the game tree. While we have some personal preferences, we believe all our algorithms have their place and benefits.

7.1 Future Work

Contrary to established practices, our proposal suggests the replacement of the TD-style updates with a single LSPI-style update. The proposed improvements to TD-style learning mentioned in Chapter 3 could be used in conjunction with LSPI-style learning to improve efficiency. This is a future research direction.

In addition, we are currently further exploring the impact of our methods in stochastic adversarial games and we intent to fine tune our approach.

7.2 Conclusion

In this thesis, 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 four variations of the LSPI approach over the widely-used TD approach in the classical board games of Othello/Reversi and Backgammon.

Part of this work was presented in ICTAI 2012 [3].

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] I. E. Skoulakis and M. G. Lagoudakis, “Efficient reinforcement learning in adversarial games,” in *2012 IEEE 24th International Conference on Tools with Artificial Intelligence*, vol. 1, pp. 704–711, Nov 2012.

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

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

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

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

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

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

[10] D. Ernst, P. Geurts, and L. Wehenkel, “Tree-based batch mode reinforcement learning,” *Journal of Machine Learning Research*, vol. 6, pp. 503–556, 2005.

[11] M. Riedmiller, "Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method," in *Machine Learning: ECML 2005*, vol. 3720, pp. 317–328, 2005.

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

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

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

[15] 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)*, pp. 28–36, 1998.

[16] 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.), pp. 1937–1945, 2009.

[17] 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)*, pp. 283–292, 2002.

[18] M. G. Lagoudakis and R. Parr, "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.), pp. 1627–1634, 2003.

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

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

[21] I. Skoulakis, “Systematic search and reinforcement learning for the board game “Neighbours”,” diploma thesis, Technical University of Crete, Greece, 2010.

[22] 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)*, pp. 417–422, 2010.

[23] L. Lin, *Reinforcement Learning for Robots Using Neural Networks.* PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1993.

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

[25] M. Buro, “Improving heurisitic mini-max search by supervised learning,” *Artificial Intelligence*, vol. 134, no. 1-2, pp. 85–99, 2002.

[26] G. Tesauro, “Temporal difference learning and td-gammon,” *Communications of the ACM*, vol. 38, no. 3, 1995.

[27] Wikipedia, “Reversi — wikipedia, the free encyclopedia,” 2012.

[28] Wikipedia, “Backgammon — wikipedia, the free encyclopedia,” 2019.

[29] G. Tesauro, “Benchmark player pubeval.c - backgammon forum archives,” 2019.