Abalearn: A Risk-Sensitive Approach to Self-play Learning in Abalone

Pedro Campos[1] and Thibault Langlois[1,2]

[1] INESC-ID, Neural Networks and Signal Processing Group, Lisbon, Portugal
`pfpc@mega.ist.utl.pt`
[2] Faculdade de Ciências da Universidade de Lisboa, Departamento de Informática
Lisbon, Portugal
`tl@di.fc.ul.pt`

Abstract. This paper presents Abalearn, a self-teaching Abalone program capable of automatically reaching an intermediate level of play without needing expert-labeled training examples, deep searches or exposure to competent play.
Our approach is based on a reinforcement learning algorithm that is risk-seeking, since defensive players in Abalone tend to never end a game.
We show that it is the risk-sensitivity that allows a successful self-play training. We also propose a set of features that seem relevant for achieving a good level of play.
We evaluate our approach using a fixed heuristic opponent as a benchmark, pitting our agents against human players online and comparing samples of our agents at different times of training.

1 Introduction

This paper presents Abalearn, a self-teaching Abalone program directly inspired by Tesauro's famous TD-Gammon [14], which used Reinforcement Learning (RL) methods to learn by self-play a Backgammon evaluation function. We chose Abalone because the game's dynamics represent a difficult challenge for RL methods, particularly for methods of self-play training. It has been shown [8] that Backgammon's dynamics are crucial to the success of TD-Gammon, because of its stochastic nature and the smoothness of its evaluation function. Abalone, on the other hand, is a deterministic game that has a very weak reinforcement signal: in fact, players can easily repeat the same kind of moves and the game may never end if one doesn't take chances.

Exploration is vital for RL to work well. Previous attempts to build an agent capable of learning how to play games through reinforcement either use expert-labeled training examples [5] or exposure to competent play (online play against humans [3] or learning by playing against a heuristic player [5]). We propose a method capable of efficient self-play learning for the game Abalone that is based on risk-sensitive RL [7]. We also provide a set of features and state representations for learning to play Abalone, using only the outcome of the game as a training signal.

Table 1. Complexity of several games.

Game	Branch	States	Source
Chess	30–40	10^{50}	[4]
Checkers	8–10	10^{17}	[10]
Backgammon	±420	10^{20}	[18]
Othello	±5	$< 10^{30}$	[20]
Go 19×19	±360	10^{160}	[11]
Abalone	±80	$< 3^{61}$	[1]

The rest of the paper is organized as follows: section 2 briefly analyses the game's complexity. Section 3 refers and explains the most significant previous RL efforts in games. Section 4 details the training method behind Abalearn and section 5 describes the state representations used. Finally, section 6 presents the results obtained using a heuristic player as benchmark, as well as results of games against other programs and human expert players. Section 7 draws some conclusions about our work.

2 Complexity in the Game Abalone

The rules of Abalone are simple to understand: to win, one has to push off the board 6 out of the 14 opponent's stones by outnumbering him/her[1]. Despite this apparent simplicity, the game is very popular and challenging [1]. Table 1 compares the branching factor and the state space dimension of some zero-sum games. The data was gathered from a selection of papers that analyzed those games.

These are all estimated values, since it is very difficult to determine rigorously the true values of these variables. Abalone has a branching factor higher than Chess, Checkers and Othello, but does not match the complexity of Go. The branching factor in backgammon is due to the dice rolls and is the main reason why other search techniques have to be used for this game.

The problem in Abalone is that when the two players are defensive enough, the game can easily go on forever, making the training more difficult (since it weakens the reinforcement signal).

3 Related Work

In this section we present a small survey on programs that learn to play games using RL. The most used method is Temporal Difference Learning, or TD-Learning. Samuel's checkers player [9] already used a form of temporal difference learning, as well as Michie's Tic-tac-toe player [6]. They both pre-date reinforcement learning as a field, but both basically use the same ideas.

[1] For further information about the games rules and strategies, please refer to the Official Abalone Web-site: `www.abalonegames.com`.

3.1 The Success of TD-Gammon

Tesauro's TD-Gammon [16] caused a small revolution in the field of RL. TD-Gammon was a Backgammon player that needed very few domain knowledge, but still was able to reach master-level play [15]. The learning algorithm, a combination of TD(λ) with a non-linear function approximator based on a neural network, became quite popular.

Besides predicting the expected return of the board position, the neural network also selected both agent and opponent's moves throughout the game. The move selected was the one for which the function approximator gave the higher value.

Modeling the value function with a neural network poses a number of difficulties, including what the best network topology is and what the input encoding should look like. Tesauro used a number of backgammon-specific features in addition to the other information representing the board to increase the information immediately available to the neural network. He found that this additional information gave another performance improvement.

TD-Gammon's surprising results were never repeated to other complex board games, such as Go, Chess and Othello. Many authors [11,2,8] have discussed Backgammon's characteristics that make it perfectly suitable for TD-learning through self-play. Among others, they emphasize: the speed of the game (TD-Gammon was trained by playing 1.5 million games), the smoothness of the game's evaluation function which facilitates the approximation via neural networks, and the stochastic nature of the game: the dice rolls force exploration, which is vital in RL.

Pollack and Blair show that a method initially considered weak – training a neural network using a simple hill-climbing algorithm – leads to a level of play close to the TD-Gammon level [8], which sustains that there is a bias in the dynamics of Backgammon that inclines it in favor of TD-learning techniques. Although Tesauro does not entirely agree with Pollack and Blair [17], it is quite surprising that such a simple procedure works at all.

3.2 Exposure to Competent Play

Learning from self-play is difficult as the network must bootstrap itself out of ignorance without the benefit of exposure to skilled opponents. As a consequence, a number of reported successes are not based on the networks' own predictions, but instead they learn by playing against commercial programs, heuristic players, human opponents or even by simply observing recorded games between human players. This approach helps to focus on the state space fraction that is really relevant for good play, but places the need of an expert player, which is what we want to obtain in the first place.

The Chess program KnightCap was trained by playing against human opponents on an internet chess server [3]. As its rate improved, it attracted stronger and diverse opponents, since humans tend to choose partners of the same level of play. This was crucial to KnightCap's success, since the opponents guided

KnightCap throughout its training (similar to the dice rolls in backgammon, which facilitated exploration of the state space). Thrun's program, NeuroChess [19], was trained by playing against GNUChess, a heuristic player, using TD(0).

Dahl [5] proposes an hybrid approach for Go: a neural network is trained to imitate local game shapes made by an expert database via supervised learning. A second net is trained to estimate the safety of groups of stones using TD(λ), and a third net is trained, also by TD(λ)-Learning to estimate the potential of non-occupied points of the board.

4 Abalearn's Training Methodology

Temporal difference learning (TD-learning) is an unsupervised RL algorithm [12]. In TD-learning, the evaluation of a given position is adjusted by using the differences between its evaluation and the evaluations of successive positions.

Sutton defined a whole class of TD algorithms which look at predictions of positions which are further ahead in the game and weight them exponentially less according to their temporal distance by the parameter λ.

Given a series of predictions, $V_0, ..., V_t, V_{t+1}$, then the weights in the evaluation function can be modified according to:

$$\Delta w_t = \alpha \left(V_{t+1} - V_t \right) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w V_k \tag{1}$$

TD(0) is the case in which only the one state preceding the current one is changed by the TD error ($\lambda = 0$). For larger values of λ, but still $\lambda < 1$, more of the preceding states are changed, but each more temporally distant state is changed less. We say that earlier states are given less credit for the TD error [13].

Thus, the λ parameter determines whether the algorithm is applying short range or long range prediction. The α parameter determines how quickly this learning takes place.

A standard feed-forward two-layer neural network represents the agent's evaluation function over the state space and is trained by combining TD(λ) with the Backpropagation procedure. We used the standard sigmoid as the activation function for the hidden and output layers' units. Weights are initialized to small random values between -0.01 and 0.01.

Rewards of $+1$ are given whenever the agent pushes an opponent's stone off the board or whenever it wins the game. When the agent loses the game or when the opponent pushes an agent's stone the reward is –1, otherwise it is 0 [2].

[2] Another option would be to give a positive reward only at the end of the game (when six stones have been pushed off the board). The agent would be able to learn to "sacrify" stones in order to improve its position. This option has not been used in the present paper, in part because we believe that a value function that take into account sacrifices must be much more difficult to approximate. This may be an interesting direction for future work.

One of the problems we encountered was that self-play was not effective because the agent repeatedly kept playing the same kind of moves, never ending a game. When training is based on self-play, the problem of exploration is very important because the agent may restrict itself to a small portion of the state space and become weaker and weaker, because the opponent is itself. This characteristic is not specific to the Abalone but applies to any agent that learns by self-play.

One way to favor exploration of the state space is to use an ϵ-greedy policy. During training, the agent follows an ϵ-greedy policy, selecting a random action with probability ϵ and selecting the action judged by the current evaluation function as having the highest value with probability $1-\epsilon$. The drawback of this solution is that it introduces noise in the policy "blindly" i.e. without taking into account the value of the current state.

The solution was to provide the agent with a sensitivity to risk during learning. Mihatsch and Neuneier [7] recently proposed a method that can help accomplish this. Their risk–sensitive RL algorithm transforms the temporal differences. In this approach, $\kappa \in (-1, 1)$ is a scalar parameter which specifies the desired risk–sensitivity. The function

$$\chi^\kappa : x \mapsto \begin{cases} (1-\kappa)x \text{ if } x > 0, \\ (1+\kappa)x \text{ otherwise} \end{cases} \tag{2}$$

is called the transformation function, since it is used to transform the temporal differences according to the risk sensitivity. The risk sensitive TD algorithm updates the estimated value function V according to

$$V_t(s_t) = V_{t-1}(s_t) + \alpha\chi^\kappa[R(s_t, a_t) + \gamma V_{t-1}(s_{t+1}) - V_{t-1}(s_t)] \tag{3}$$

When $\kappa = 0$ we are in the risk–neutral case. If we choose κ to be positive then we overweight negative temporal differences

$$R(s_t, a_t) + \gamma V(s_{t+1}) - V(s_t) < 0 \tag{4}$$

with respect to positive ones. That is, we overweight transitions to states where the immediate return $R(s, a)$ happened to be smaller than in the average. On the other hand, we underweight transitions to states that promise a higher return than in the average. In other words, the agent is risk-avoiding when $\kappa > 0$ and risk-seeking when $\kappa < 0$. We discovered that negative values for κ lead to an efficient self-play learning (see section 6).

When a neural network function approximator is used with Risk-Sensitive Reinforcement Learning, the TD(λ) update rule for parameters becomes:

$$w_{t+1} = w_t + \alpha\chi^\kappa(d_t)\sum_{k=1}^{t}\lambda^{t-k}\nabla_w V(s_k; w) \tag{5}$$

with

$$d_t = R(s_t, a_t) + \gamma V(s_t; w) - V(s_{t-1}; w)$$

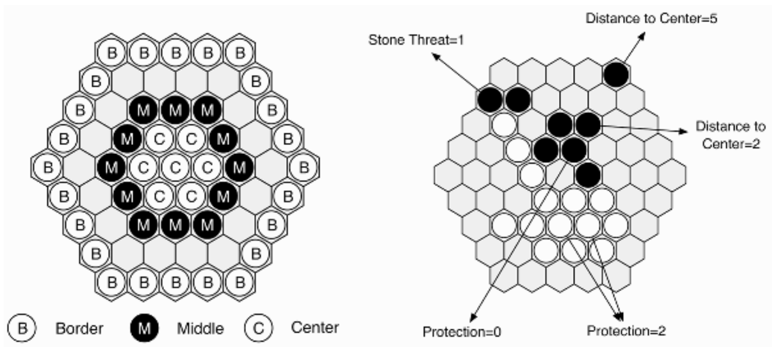

Fig. 1. The architecture used for Abalearn 2 encodes: the number of stones in the center, in the middle, in the border and pushed off the board (left) and the same for the opponent's stones. Abalearn 3 adds some basic features of the game (right).

5 Efficient State Representation

The state representation is crucial to a learning system, since it defines everything the agent might ever learn. In this section, we describe the neural network architectures we implemented and studied.

Let us first consider a typical architecture that is trained to evaluate board positions using a direct representation of the board. We call the agent using this architecture Abalearn 1. It is a basic and straightforward state representation, since it merely describes the contents of the board: it maps each position in the board to -1 if the position contains an opponent's stone, +1 if it contains an agent's stone and 0 if it is empty. It also encodes the number of stones pushed off the board (for both players).

We wish the network to achieve a good level of play. Clearly, this task can be better accomplished by exploiting some characteristics of the game that are relevant for good play. We used a simple architecture that encodes the number of stones in the center, in the middle, in the border and pushed off the board (see Figure 1); and the same for the opponent's stones. The state is thus represented by a vector of 8 features, plus a bias input unit set to 1. We called this agent Abalearn 2. This network is quite a simple feature map, but it is capable of learning to play Abalone, as we will see in the next section.

We then incorporated into a new architecture (Abalearn 3) some extra handcrafted features, illustrated in Figure 1. Abalearn 3 adds some relevant (although basic) features of the game to the previous architecture. We added: *protection* (number of stones totally surrounded by stones of the same color), the average distance of the stones to the center of the board and the number of stones threatened (see Figure 1).

6 Results

In this section we present the results of two training methods. Common parameter values in both methods are: $\alpha = 0.1, \gamma = 0.9$. Unless specified, the value of the λ parameter was 0.7.

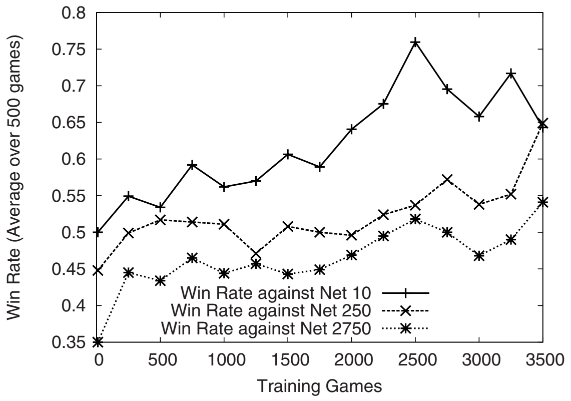

Fig. 2. Comparison between some reference networks, sampled after 10, 250 and 2750 training games (average of 500 games) shows that learning is succeeding.

Method I. This method applies standard TD(λ) using Abalearn 2, described in the previous section. In method I, the agent plays 1000 games against a random opponent in order to extract some basic knowledge (mainly learning to push the opponent's stones off the board). After that phase, we train the agent using self-play. This method never succeeds when using self-play training from the beginning.

Method I(a). This method is the same as Method I. Only the state representation changes to Abalearn 3. This method is necessary to prove the benefit of the added features in Abalearn 3 with respect to Abalearn 2.

Method II. We wished to obtain an agent capable of efficient and automatic self-play learning. Method II accomplishes this. It applies the risk-sensitive version of TD(λ) using self-play and Abalearn 3, also described in the previous section. Exploration is important especially at the beginning of the train, so we used a decreasing ϵ: after each game t, $\epsilon_{t+1} = 0.99 \times \epsilon_t$, with $\epsilon_0 = 0.9$.

Testing Methods. The most straightforward method for testing our agents is by averaging their win rate against a good heuristic[3] player. The heuristic function sums the distance to the center of the board of each stone (subtracts if it's an opponent stone). We also tested our agents by playing some games against the best Abalone program and by making them play at the Abalone Website against human experts.

6.1 Method I: Standard TD(λ)

We tested our networks against three networks sampled during previous training. Figure 2 shows the results. Each curve represents an average over 500 games. Each network on the X-Axis plays against Net 10, Net 250 and Net 2750 (networks sampled after 10, 250 and 2750 training games respectively). As we can

[3] We use a simple Minimax search algorithm.

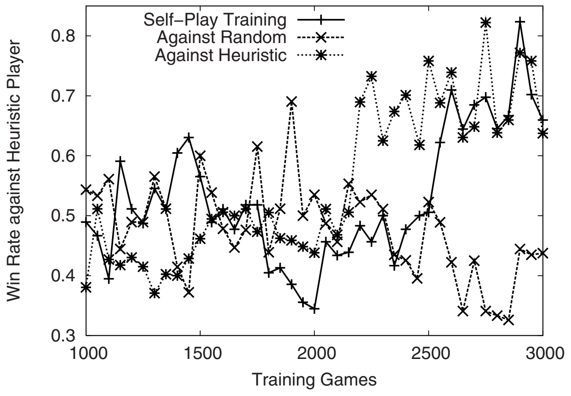

Fig. 3. Performance of the agents when trained against different kinds of opponents.

Table 2. Comparison between the two methods (Win Rate against Heuristic Player).

Training Games	Method I	Method I(a)
500	48%	68%
1000	52%	72%
2000	54%	76%
3000	71%	79%

see, it is easy for the networks to win Net 10. On the other hand, Net 2750 is far superior to all the others.

Exposure to Competent Play. A good playing partner offers knowledge to the learning agent, because it easily leads the agent through the relevant fractions of the state space.

In this experiment, we compare agents that are trained by playing against a random opponent, a strong minimax player and by self-play. Figure 3 summarizes the results. Each point corresponds to an average over 500 games against the heuristic opponent. We can see that a skilled opponent is more useful than a random opponent, as expected.

The Benefit of the Features. Table 2 compares the two state representations: it presents the win percentage against a heuristic player over 100 testing games, using method I and I(a). The agent trained with method I(a) uses the state representation with added features (see section 5) and after only 1000 games of training, presents a better performance than the agent trained with method I. This proves the features added were relevant to learning the game and yielded better performances.

6.2 Method II: Self-play with Risk-Seeking TD(λ)

Figure 4 shows the results of training for four different risk sensitivities: $\kappa = -1$ (the most risk-seeking agent), $\kappa = -0.8$, $\kappa = -0.3$ and $\kappa = 0$ (the classical risk-neutral case). We trained and tested 10 agents. We can see that performance is

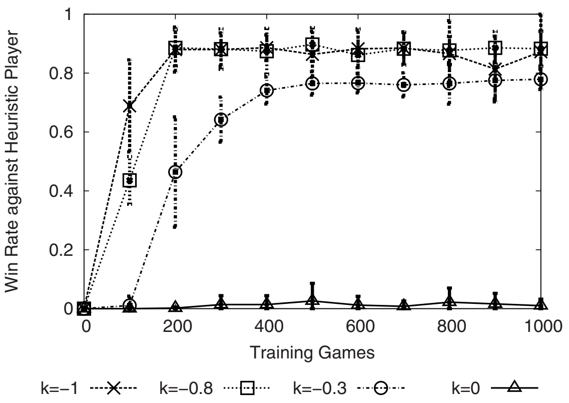

Fig. 4. Performance of the risk-sensitive RL agents when trained by self-play for various values of risk-sensitivity. Self-play is efficient for negative values of risk-sensitivity.

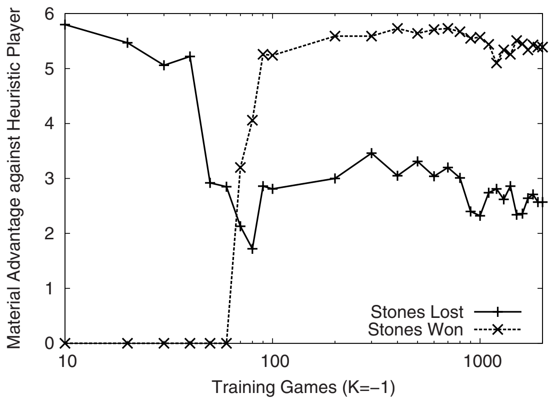

Fig. 5. Improvement in performance of the risk-seeking self-playing agent ($\kappa = -1$).

best when $\kappa = -0.8$ and $\kappa = -1$. We verified that after 10000 games of self-play training with $\kappa = -1$ performance kept the same (see Figure 5, which plots the results for the first 2000 games). By assuming that losses are inevitable, the agent ignores most of the negative temporal differences and the weights associated to the material advantage are positively rewarded.

We trained the agent with $\kappa = 0$ and it didn't learn to push the opponent's stones, thereby losing most games agianst the heuristic player, except for 1 out of 10 runs of the experiment. This is because the lack of risk-sensitivity leads to highly conservative policies where the agent learns to maintain its stones in the center of the board and avoids to push opponent's stones. This experiment illustrates the importance of risk-sensitivity in self-play learning: in method I(a), performance is worse (see Table 2).

Performance against the best program. We wanted to evaluate how TD-learning fares competitively against other methods. ABA-PRO, a commercial

Table 3. Abalearn using method I with fixed 1-ply search depth only loses when the opponent's search depth is 6-ply. Method II performs better.

Method I Depth=1 vs.:	Stones Won	Stones Lost	Moves	First Move
ABA-PRO Depth=4	0	0	31	ABA-PRO
ABA-PRO Depth=5	0	0	23	ABA-PRO
ABA-PRO Depth=6	0	2	61	ABA-PRO
Method II Depth=1 vs.:	**Stones Won**	**Stones Lost**	**Moves**	**First Move**
ABA-PRO Depth=4	0	0	29	ABA-PRO
ABA-PRO Depth=5	0	0	21	ABA-PRO
ABA-PRO Depth=6	0	0	42	ABA-PRO

Table 4. Abalearn playing online managed to win intermediate players.

Abalearn Method I vs.:	Stones Won	Stones Lost	First Move
ELO 1448 (weak intermediate)	6	1	Human Player
ELO 1590 (strong intermediate)	3	6	Human Player
ELO 1778 (expert)	0	6	Human Player
Abalearn Method II vs.:	**Stones Won**	**Stones Lost**	**First Move**
ELO 1501 (intermediate)	2	0	Human Player
ELO 1500 (intermediate)	6	1	Human Player
ELO 1590 (strong intermediate)	6	1	Human Player
ELO 1590 (strong intermediate)	6	3	Human Player
ELO 1590 (strong intermediate)	6	4	Human Player
ELO 1590 (strong intermediate)	6	4	Human Player

application, that is one of the best Abalone computer players built so far [1] relies on sophisticated search methods and hand-tuned heuristics that are hard to discover. It also uses deep, highly selective searches (ranging from 2 to 9-ply). Therefore, we pitted Abalearn trained as described before against ABA-PRO.

Table 3 shows some results obtained varying the search depth of ABA-PRO and maintaining our agent performing a fast 1-ply search[4]. The free-version is limited to 6-ply search.

As we can see, Abalearn only loses 2 stones when its' opponent search depth is 6. This shows that it is possible to achieve a good level of play using our training methodology. Once again, method II performs better (never loses).

Performance against Human Experts. To better assess Abalearn's level of play, we made it play online at the Abalone Official Server. As in all other games, players are ranked by their ELO.

Table 4 shows the results of some games played by Abalearn online against players of different ELOs. Method I won a player with ELO 1448 by 6 to 1 and managed to lose by 3 to 6 against an experienced 1590 ELO player. When

[4] When the game reaches a stage where both players repeat the same moves for 20 consecutive times, we end the game (tie by repetition). We carried out this experiment manually because we didn't implement an interface between the two programs.

playing against a former Abalone champion, Abalearn using method I lost by 6 to 0, but it took more than two hours for the champion to beat Abalearn, mainly because Abalearn defends very well and one has to try to ungroup its stones slowly towards a victory.

Method II is more promising because of its incorporated extra features[5]. We have tested it against players of ELO 1501, 1500 and 1590 (see Table 4).

7 Conclusions

This paper describes a program, Abalearn, that learns how to play the game of Abalone using the TD(λ) algorithm and a neural network to model the value function. The relevant information given to the learning agent is limited to the reinforcement signal and a set of features that define the agent's state. The programs learns by playing against itself.

We showed that the use of a Risk-Sensitive version of the TD(λ) algorithm allows the agent to learn by self-play. The performance level of Abalearn is evaluated against a heuristic player, a commercial application and human players. In all cases Abalearn shows a promising performance. The best agent wins about 90% of the games against the heuristic player and ties against strong opponents. Our agent only uses a single-step lookahead. One possible direction for further work is to integrate search with RL as Baxter et al. have shown [2].

Acknowledgements

The authors wish to thank the anonymous reviewers of this paper, who provided valuable advice and interesting suggestions.

References

1. O. Aichholzer, F. Aurenhammer, and T. Werner. Algorithmic fun: Abalone. Technical report, Institut for Theoretical Computer Science, Graz University of Technology, 2002.
2. J. Baxter, A. Tridgell, and L. Weaver. Knightcap: a chess program that learns by combining TD(λ) with game-tree search. In *Proc. 15th International Conf. on Machine Learning*, pages 28–36. Morgan Kaufmann, San Francisco, CA, 1998.
3. J. Baxter, A. Tridgell, and L. Weaver. Learning to play chess using temporal differences. *Machine Learning*, 40(3):243–263, 2000.
4. D. F. Beal and M. C. Smith. Temporal difference learning for heuristic search and game playing. *Information Sciences*, 122(1):3–21, 2000.
5. F. A. Dahl. Honte, a go-playing program using neural nets. 1999.
6. D. Michie. Experiments on the mechanization of game-learning – part i. characterization of the model and its parameters. *The Computer Journal*, 6:232–236, 1963.

[5] To play against the latest version of Abalearn online, please visit the following URL: `http://neural.inesc.pt/Abalearn/index.html`.

7. O. Mihatsch and R. Neuneier. Risk-sensitive reinforcement learning. *Machine Learning*, 49:267–290, 2002.
8. J. B. Pollack and A. D. Blair. Co-evolution in the successful learning of backgammon strategy. *Machine Learning*, 32(1):225–240, 1998.
9. A. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):211–229, 1959.
10. J. Schaeffer, M. Hlynka, and V. Jussila. Temporal difference learning applied to a high-performance game-playing program. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 529–534, 2001.
11. N. Schraudolph, P. Dayan, and T. J. Sejnowski. Temporal difference learning of position evaluation in the game of go. In *Advances in Neural Information Processing Systems*, volume 6. Morgan Kaufmann Publishers, Inc., 1994.
12. R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.
13. R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction Reinforcement Reinforcement Learning: an Introduction.* The MIT Press, 1st edition, 1998.
14. G. Tesauro. Practical issues in temporal difference learning. In John E. Moody, Steve J. Hanson, and Richard P. Lippmann, editors, *Advances in Neural Information Processing Systems*, volume 4, 1992.
15. G. Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. In *Proceedings of the AAAI Fall Symposium on Intelligent Games: Planning and Learning*, pages 19–23, Menlo Park, CA, 1993. The AAAI Press.
16. G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.
17. G. Tesauro. Comments on "co-evolution in the successful learning of backgammon strategy". *Machine Learning*, 32(3):41–243, 1998.
18. G. Tesauro. Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134:181–199, 2002.
19. S. Thrun. Learning to play the game of chess. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 1069–1076. The MIT Press, Cambridge, MA, 1995.
20. T. Yoshioka, S. Ishii, and M. Ito. Strategy acquisition for the game othello based on reinforcement learning. *IEICE Transactions on Inf. and Syst.*, 12(E82 D), December 1999.