Abalearn: Efficient Self-Play Learning of the game Abalone

Pedro Campos and Thibault Langlois

INESC-ID, Neural Networks and Signal Processing Group,
Lisbon, Portugal
{pfpc,tl}@neural.inesc.pt
http://neural.inesc.pt/ pfpc

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 or deep searches.
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 extend the risk-sensitive reinforcement learning framework in order to deal with large state spaces and 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 [19], which used Reinforcement Learning methods to learn by self-play a Backgammon evaluation function. We chose Abalone because the game's dynamics represent a difficult challenge for Reinforcement Learning (RL) methods, in particular, methods of self-play training. It has been shown [3] 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 learn through reinforcement either use expert-label training examples or exposure to competent play (online play against humans or learning by playing against a heuristic player). We propose a method capable of efficient self-play learning for the game Abalone that is partly based on risk-sensitive RL. 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.

The rest of the paper is organized as follows: section 2 briefly presents the rules of the game and analyses its complexity. Section 3 refers and explains the most significant previous efforts in machines learning games. Section 4 details the training method behind Abalearn and section 5 describes the state representations used. Finally, section 6 presents the obtained results 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 Abalone: The Game

In this section we present the game Abalone. We describe the rules of the game as well as some basic strategies and problems this game poses for a Reinforcement Learning agent to tackle with.

Abalone is a strategy game sold up to 4 million pieces in 30 countries. The games was ranked “Game of the Decade” at the International Game Festival in 1998. Figure 1 shows the overall view of the 2001 International Abalone Tournament. The rules 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.

Fig. 1. A view over the 2001 International Abalone Tournament.

2.1 The Rules

Figure 2 shows the initial board position: on the hexagonal board, 14 black stones face its 14 opponent’s stones. The first player to eject six of the opponent’s stones wins the game.

One, two, or three stones of the same color can move one space in any of six directions, as shown in Figure 2, provided that the target spots are empty. If moving more than one stone, the group must be contiguous and in a line. You cannot move 4 or more stones of the same color in the same turn.

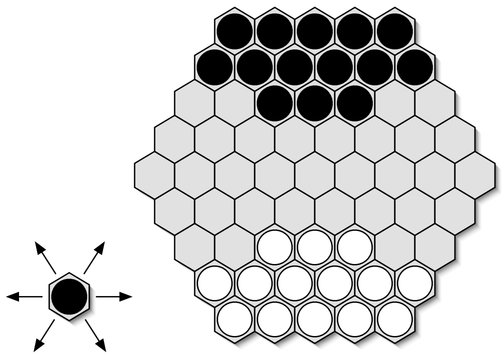

Fig. 2. Initial board position and the six possible move directions.

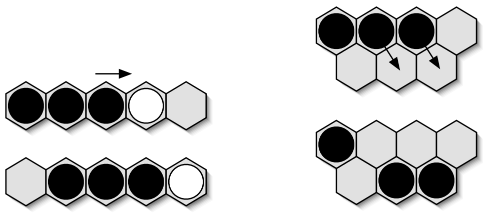

Fig. 3. In-line Move (left) and Broadside Move (right).

Moves fall into two categories. In-line moves involve moving all the stones in a group in a straight line, forwards or backwards. Broadside moves involve moving a group of stones sideways, to an adjacent row. Figure 3 illustrates these moves.

A player may push his/her opponent's stones only if they are directly in the way of an in-line move. That is, they must be laid out in the direction the player's group is oriented and be adjacent to one of the stones in the player's group.

A player is never required to push. A player can only push his/her opponent's stones if the player's stones outnumber the opponent's (three may push two or one, two may push one). A player may not push if one of his/her own stones is in the way.

2.2 Complexity in the game Abalone

It is worthwhile to note the complexity of the task proposed. 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 analysed those games.

Game	Branch	States	Source
Chess	30–40	10^5	[24]
Checkers	8–10	10^{17}	[23]
Backgammon	±420	10^{20}	[20]
Othello	±5	$< 10^{30}$	[26]
Go 19×19	±360	10^{160}	[9]
Abalone	±80	$< 3^{61}$	[27]

Table 1. Complexity of several games.

These are all estimation 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 questionable, since it is due to the stochastic element of the game (the dice rolls).

Aicholzer [27] built a strong heuristic player able to perform up to 9-ply search that has never been beaten by any human player. The techniques used were: brute force search, clever algorithmic techniques and sophisticated evaluation strategies (developed by human intelligence). There have been also some non-scientific implementations of Abalone playing programs, such as KAbalone, MIT-Abalone and a Java Computer Player.

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). Another curious fact is that there is at least one position for which the game is undefined [16], and a rule should be added considering the case when a position was encountered more than one time (tie by repetition).

3 Related Work

In this section we present a small survey on machines that learn to play games through reinforcement learning. The most used method is Temporal Difference Learning, or TD-Learning. Particular emphasis is placed on TD-Gammon and application of the techniques in TD-Gammon to other board games. We will see that all of these machines use, in more or less degree, techniques not related to RL, such as brute-force search, opening books and hand-coded board features to improve the agent's performance level.

We focus on three main techniques: heuristic search, hand-coded board features and exposure to competent play.

3.1 The success of TD-Gammon

Tesauro's TD-Gammon [7] caused a small revolution in the field of RL. TD-Gammon is a Backgammon player that needed very few domain knowledge, but still was able to reach master-level play [8]. The learning algorithm, a combination of TD(λ) with a non-linear function approximator based on a neural network, became quite popular.

The neural network has a dual role of predicting the expected return of the board position and selecting both agent and opponent's moves throughout the game. The move chosen is the one for which the function approximator gives the higher value.

Using a network for learning poses a number of difficulties, including what the best network topology is and what the input encoding should look like. Tesauro added a number of backgammon-specific feature codings to the other information representing the board to increase the information immediately available to the net. He found that this additional information was very important to successful learning by the net.

TD-Gammon's surprising results were never repeated to other complex board games, such as Go, Chess and Othello. Many authors, such as [9, 5, 3], have discussed Backgammon's characteristics that make it perfectly suitable for TD-learning through self-play. Among others, we emphasize: the speed of the game (TD-Gammon is 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 shows 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 [3], which sustains that there is a bias in the dynamics of Backgammon that inclines it in favor of TD-learning techniques.

In contrast, building an agent that learns to play Chess, Othello or Go by using only shallow search and simultaneously achieve master-level play is a much harder task. It is believed that for these games, one cannot get a good evaluation without deep searches, which makes it difficult to use neural networks due to computational costs. As we will see in the next section, many attempts to build a good game-playing agent use these forms of search. However, in RL the ideal agent is supposed to learn the evaluation function for an infinite horizon, and should not need to perform deep searches.

3.2 Combining Heuristic Search with TD(λ)

Baxter et al. present a training method called TD-Leaf(λ), a variation on TD(λ) that consists in using the temporal difference between the leaf node evaluation in the minimax tree search and the previous state. This algorithm was used in a Chess program called KnightCap. The major result was an improvement of several hundred rating rating points, leading to an expert rating on an internet chess server [4].

Yoshioka et al. employ a normalized gaussian network that learns to play Othello and also uses a Minimax strategy for selecting the moves in the game, designated by the authors as MMRL (Min-Max Reinforcement Learning) [26]. Their agent was able to win a heuristic player.

In Othello, the board position changes radically even after only one move. Therefore similar values are given to those states that are so different. Besides that, a small variation on the board can lead to a significative difference in the evaluation function. This is why it is difficult to evaluate an Othello position.

3.3 Hand-Coded Board Features

Tesauro was not alone in providing his neural network a set of hand-coded features. Baxter et al. use a long feature list that includes not only material terms, but also many features relevant to good chess playing [4], as well as an opening book that stores good opening moves (a form of rote-learning). Similar approaches are applied to Checkers, Go and Othello.

Some authors, instead of explicitly feeding board features to the network, take advantage of spatial and temporal characteristics of the game to build a more efficient state codification. This facilitates the acquisition of good game strategies.

Leouski presents a network architecture that reflects the spatial and temporal organization of the board. He observed that the Othello board was invariant regarding reflection and simetry. This simetry was exploited by a weight-sharing neural network [21].

Schraudolph et al. propose a neural network-based approach that reflects the spatial characteristics of the game of Go [9, 10]. Go is a true challenge: it has a very high branching factor and temporal and spatial interactions that make position evaluations very difficult. Current game-programming techniques all appear to be inadequate for achieving a master level of play.

3.4 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 once again deviates us from our ideal agent, and places the need of an expert player, which is what we want to obtain in the first place.

KnightCap was trained by playing against human opponents on an internet chess server [4]. As its rate improved, it attracted stronger 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.

Thrun's program, NeuroChess, was trained by playing against GNUChess and using TD(λ), with λ set to zero [14].

Dahl [11] proposes a 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.

Imitating human concepts has had some success. Nevertheless, human-based modelling shows that any misconception assumed by the programmer can be inherited and exacerbated by the program, which may cause failures and – once again – are far from our ideal agent: the one that discovers new knowledge by himself. In the following section, we describe a training methodology that tries to accomplish this task.

4 Abalearn's Training Methodology

Temporal difference learning (TD-learning) is an unsupervised RL algorithm [2]. In TD-learning, the evaluation of a given position is adjusted by using the differences between its evaluation and the evaluations of successive positions. This means the prediction of the result of the game in a particular position is related to the predictions of the following positions.

Sutton defined a whole class of TD algorithms which look at predictions of positions which are further ahead in the game weighted exponentially less according to their 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 the earlier states are given less credit for the TD error [1].

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.

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 $1 - \epsilon$.

A standard feedforward 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 and whenever it wins the game. When the agent loses the game or when the opponent pushes an agent's stone the reward is –1. Default reward is 0.

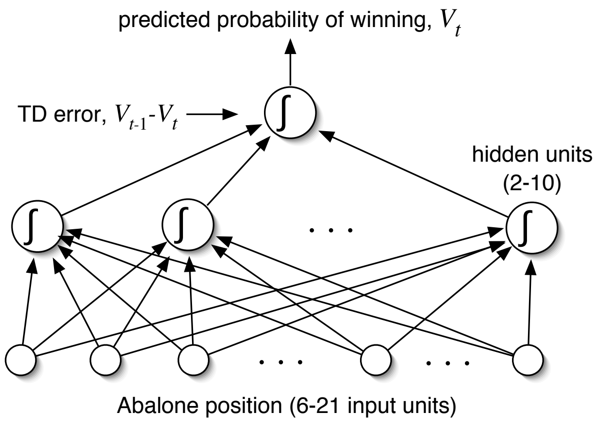
predicted probability of winning, V_t
TD error, V_{t-1}-V_t
hidden units (2-10)
Abalone position (6-21 input units)

Fig. 4. The neural network used in Abalearn as well as the minimum and maximum numbers of units we tried for the input and hidden layers.

4.1 Applying Risk–Sensitive RL

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. The solution was to provide the agent with a sensitivity to risk during learning.

Mihatsch and Neuneier [32] recently proposed a method that can help accomplish this. Their risk–sensitive reinforcement learning algorithm transforms the temporal differences (so–called TD–errors) which play an important role during the learning of our Abalone evaluation function. In this approach, $\kappa \in (-1, 1)$ is a scalar parameter which specifies the desired risk–sensitivity. The function

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

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

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

When $\kappa = 0$ we are in the risk–neutral case (like the one we have been using so far). If we choose κ to be negative then we overweight negative temporal differences

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

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. We are approximating a risk-avoiding function if $\kappa > 0$ and a risk-seeking function if $\kappa < 0$. In other words, the agent is risk-avoiding when

$\kappa > 0$ and risk-seeking when $\kappa < 0$. We discovered that negative values for κ, as we will see in section Results, apparently lead to an efficient self-play learning.

In order to deal with the large state/action space dimension, we have to extend risk-sensitive RL to the case where a parametric function approximator for the value function is used. This is done using the function $J(s; w)$ that produces an approximation for $V(s)$ involving parameters in w (the weights in our neural networks implement this). Within this context, the risk-sensitive TD algorithm takes the form

$$w_{t+1} = w_t + \alpha \chi^{\kappa}(d_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w J(s_k; w) \tag{3}$$

with

$$d_t = R(s_t, a_t) + \gamma J(s_t; w) - J(s_{t-1}; w) \tag{4}$$

This is one of the first applications of risk-sensitive RL. We extended the method to deal with large state spaces and domains where conservative policies weaken the reinforcement signal. We also used training with a decreasing value of ϵ in order to ensure a good initial exploration of the state space, as we will see in section 6.

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 three main neural network architectures we implemented and studied.

Let us first consider a typical network architecture that is trained to evaluate board positions using a direct representation of the board. We call the agent using this architecture Abalearn 1.0. It is the more basic and straightforward state representation, since it merely describes the contents of the board. Abalearn's 1.0 network maps each field in the board to –1 if the field 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 learn any feature of the game it may need. Clearly, this task can be better accomplished by exploiting some characteristics of the game that are relevant for good play. Therefore, we used a simple architecture in version 1.1 that encodes:

– The Number of stones in the center of the board (see Figure 5);
– The Number of stones in the middle of the board;
– The Number of stones in the border of the board;
– The Number of stones pushed off the board;
– The same for the opponent's stones.

We called this network Abalearn 1.1. This network constitutes a quite simple feature map, but it does accelerate training and performs quite better than 1.0. We tested a 1.0 network trained after 10000 games of self-play against a 1.1 network trained after only 3000 games of self-play. Version 1.1 pushed 750 stones off the board during the 500 games of test played. If we count as a victory a game which ends in a cycle of repeated moves but the winner is the player with more stones on the board, version 1.1 wins all games against 1.0.

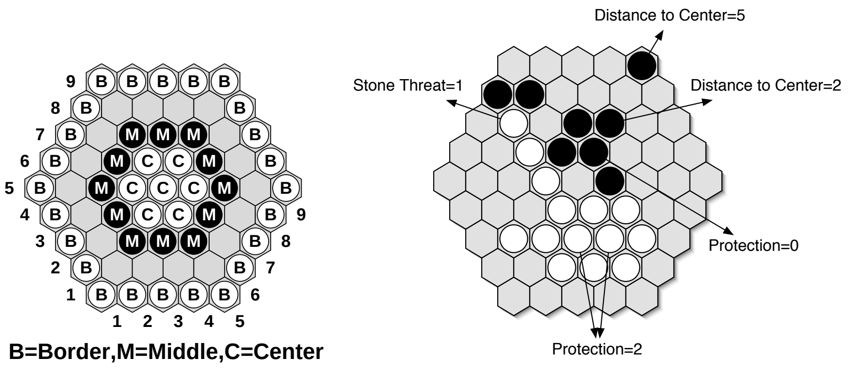

Fig. 5. The network architecture used for version 1.1 encodes: the number of stones in the center, in the middle, in the border and pushed off the board, and the same for the opponent's stones (left). Version 1.2 encodes some basic features of the game (right).

We then incorporated into a new architecture (version 1.2) some extra hand-crafted features, some of which are illustrated in figure 5. Abalearn 1.2 adds some relevant (although basic) features of the game to the previous architecture. We encode:

- The Number of stones in the center of the board (see Figure 5);
- The Number of stones in the middle of the board;
- The Number of stones in the border of the board;
- The Material Advantage;
- Protection;
- The Average Distance of the stones to the center of the board;
- The Number of stones threatened;

Figure 5 shows an example of possible values for these features.

6 Results

6.1 Training Version 1.1

The results presented in this subsection refer to the architecture of Abalearn 1.1 presented in the previous section. We trained Abalearn's neural network by playing it against a Random player in an initial phase in order to easily extract some initial basic knowledge (mainly learning to push the opponent's stones off the board). After that phase we trained it by self-play, with $\epsilon = 0.01$.

We first investigated the performance level as we tried different values of λ, for the version 1.1 network. We evaluated our agent by making it play against a Minimax player that uses a simple heuristic based on the distance to the center. Figure 6 shows the results. The Y-axis show the winning rate of the networks sampled over the course of training. The win rate refers to the average number of games the agent won over 100 games. We considered a victory when a player won by material advantage after a reasonable limit of moves, in order to avoid never-ending games.

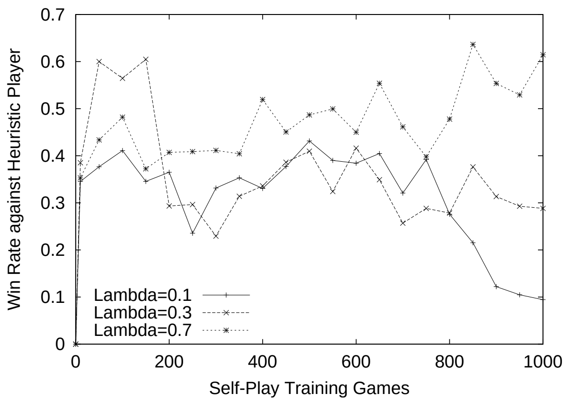

Fig. 6. Performance of Self-Play Training is best for higher values of λ.

We can see that the larger the value of λ, the better the performance. λ means the weight given to past experience, and when $\lambda \rightarrow 0$, the initial board positions are not given much credit, but those initial positions are important because a good Abalone player first moves its stones to the center in a compact manner, and only then starts to try pushing the opponent's stones in order to win the game.

To better confirm our results, we pitted some reference networks against all the others. Figure 7 shows the results. 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 see, it is easy for the networks to win Net 10. On the other hand, Net 2750 is far superior to all the others.

Finally, we wanted to evaluate how TD-learning fares competitively against other methods. The best Abalone computer player built so far [27] 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 1.1 best agent against the program of Abalone created by Tino Werner [27] we here refer to as Abalone Werner.

Table 2 shows some results obtained varying the search depth of Abalone Werner and maintaining our agent performing a fast, shallow 1-ply search. As we can see, Abalearn only loses 2 stones when it's opponent search depth is 6. This shows that it is possible to achieve a good level of play by exploiting only

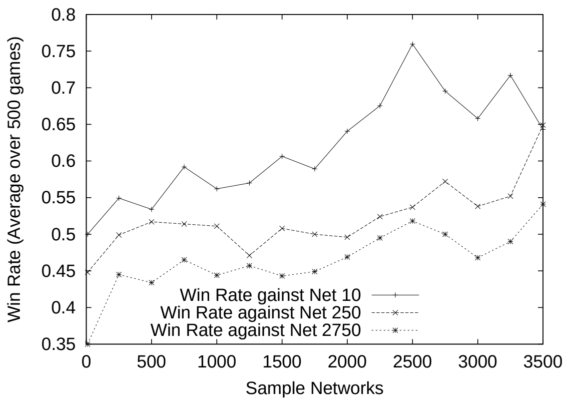

Fig. 7. Comparison between some reference networks, sampled after 10, 250 and 2750 training games (average of 500 games) shows that learning is succeeding.

Abalearn v.1.1 Depth=1 vs.:	Pieces Won	Pieces Lost
Abalone Werner Depth=4	0	0
Abalone Werner Depth=5	0	0
Abalone Werner Depth=6	0	2

Table 2. Abalearn 1.1 with fixed 1-ply search depth only loses when the opponent's search depth is 6-ply.

the spatial characteristics of the game and letting the agent play and learn from the reinforcement signal only.

6.2 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 weak random opponent, a strong minimax player and by playing against themselves. Figure 8 sumarizes the results. We can see that a skilled opponent is more useful than a random opponent, as expected. Once again, the results still refer to version 1.1.

6.3 Efficient Self-Play with Risk-Seeking RL

After observing the level of play exhibited by Abalearn's initial architecture when playing online against human expert players, we added the features described in section 5 and made version 1.2 of Abalearn. We also aimed at trying to build an agent that could efficiently learn by itself (self-play) right from the beginning.

We experimented different values for the risk-sensitivity κ and found that performance was best when $\kappa \to -1$. Figure 9 shows the average pieces lost and won against the same heuristic player (averaged over 100 games). This agent was

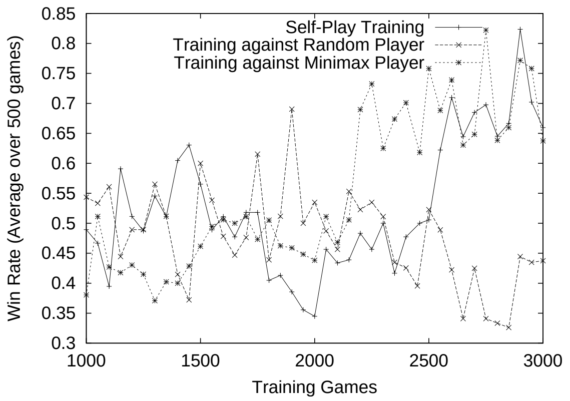

Fig. 8. Performance of the agents when trained against different kinds of opponents.

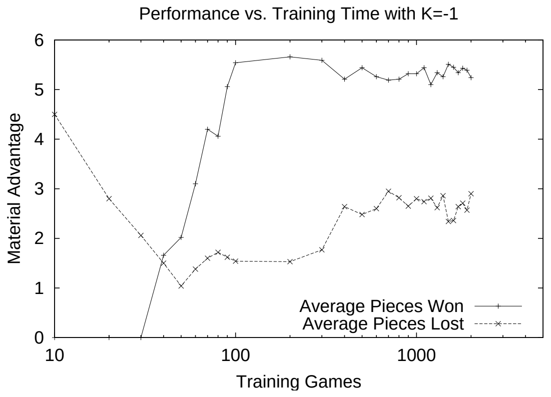

Fig. 9. Performance of the risk-sensitive RL agents when trained with $\kappa = -1$ against a Random opponent.

version 1.2 and needed only 1000 games to achieve a better level of play than version 1.1, which proves the features added were relevant to learning the game and yielded better performances. Figure 9 evaluates an agent trained against a random player. We wanted our agent to learn from self-play in an efficient manner. Figure 10 shows the results of training an agent by self-play using a decreasing ϵ. The agent starts with $\epsilon = 0.9$ and rapidly (exponentially) decreases it to zero during training. After about 100 games, ϵ is approximately 0. We continued to use $\kappa = -1$. This plot shows that self-play is well-succeeded using this exploration scheme.

We also investigated the performance of the agents trained with different values of κ. Figure 11 shows the results of training for three different values: $\kappa = -1$ (the most risk-seeking agent), $\kappa = -0.8$ and $\kappa = 0$ (the classical risk-neutral case).

We can see that performance is best when $\kappa = -1$. Furthermore, when $\kappa = -1$, the learning process is more stable than with $\kappa = -0.8$ because perfor-

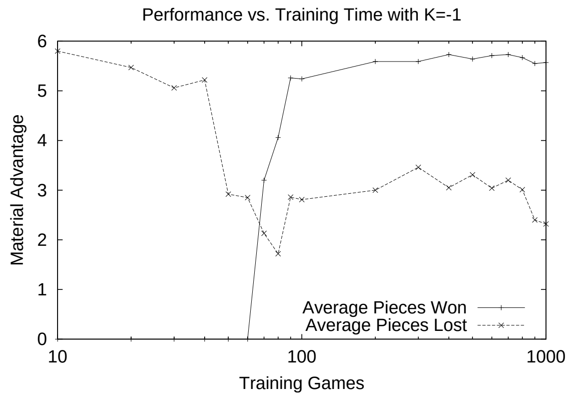

Fig. 10. Improvement in performance of the risk-seeking RL agents when trained by self-play with $\kappa = -1$.

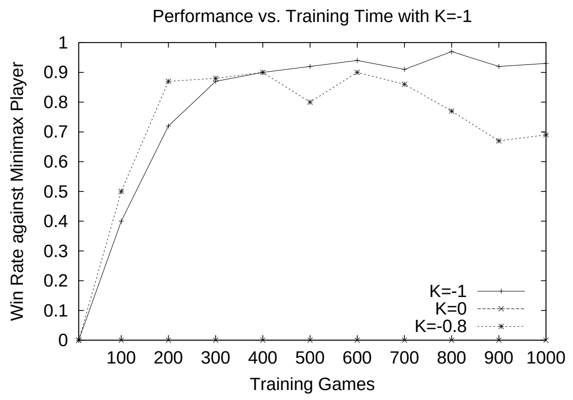

Fig. 11. Performance of the risk-sensitive RL agents when trained by self-play for various values of risk-sensitivity. Negative values of risk-sensitivity work better.

mance using $\kappa = -0.8$ initially surpassed the case where $\kappa = -1$ but ended up degrading. We verified that after 10000 games of self-play training with $\kappa = -1$ performance kept the same.

We trained the agent with $\kappa = 0$ by self-play and it didn't learn to push the opponent's pieces, thereby losing every game when tested against the heuristic player.

This is because risk-aversion leads to highly conservative policies. When playing Abalone, as we stated in section 2.2, it is very easy for the game never to end if both players don't take chances.

6.4 Evaluating against Human Experts

To better assess Abalearn's level of play, we made it play online at the Abalone Official Server[1]. Players in this server, as in all other games, are ranked by their

[1] The Abalearn Official Server's URL is `www.abalonegames.com`.

ELO. A player with an ELO $\geq$ 1500 is considered to be intermediate, whereas a player with an ELO $\geq$ 1700 is a world-class player (some of the players which now have ELOs of 1700 were once world champions).

Abalearn v.1.1 vs.:	Pieces Won	Pieces Lost
ELO 1448	6	1
ELO 1590	3	6
ELO 1778	0	6

Table 3. Abalearn 1.1 playing online managed to win intermediate players.

Abalearn v.1.2 vs.:	Pieces Won	Pieces Lost
ELO 1501	2	0

Table 4. Abalearn v.1.2 won a player with ELO 1501.

Table 3 shows the results of some games played by Abalearn 1.1 online against players of different ELOs. Abalearn 1.1 won a player with ELO 1448 by 6 to 1 and managed to lose by 3 to 6 against an experienced 1501 ELO player. When playing against a former Abalone champion, Abalearn 1.1 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 pieces slowly towards a victory.

Version 1.2 is more promising because of its incorporated extra features. We have only tested it against a player of ELO 1501 (see Table 4), and after 3 and a half hours, the human player didn't manage to win it, losing by 2-0.

7 Conclusions

In the absence of training examples, or enough computational power to perform deep searches in real-time, an automated method of learning such as the one described in this paper is needed in order to obtain an agent capable of learning a given task. This approach was successful in Backgammon but for deterministic, more complex games, it has been difficult to make self-play training work, despite all the effort put by researchers.

In Abalone, a deterministic game, the exploration problem poses serious drawbacks to a reinforcement learning system. Furthermore, a defensive player might never win (or lose) a game. We showed that by incorporating spatial characteristics of the game and some basic features, an intermediate level of play can be achieved without deep searches or training examples. We also showed that self-play training can be successful by using a risk-sensitive version of reinforcement learning capable of dealing with the game's large state space and an initial phase of random exploration (training with decreasing ϵ).

Building successful learning methods in domains like this may motivate further progress in the field of machine learning, and lead to practical approaches to real-world problems, as well as a better understanding and improvement of machine learning theory.

References

1. Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning: an Introduction*, The MIT Press, 1998, 1st edition.
2. Sutton, R. Learning to Predict by the methods of Temporal Differences. *Machine Learning* 3:9–44, 1988.
3. Jordan B. Pollack and Alan D. Blair. Co-evolution in the Successful Learning of Backgammon Strategy. *Machine Learning*, 32(3):225–240, 1998.
4. Jonathan Baxter and Andrew Tridgell and Lex Weaver. Learning to Play Chess Using Temporal Differences, *Machine Learning*, 40(3):243–263, 2000.
5. Jonathan Baxter and Andrew Tridgell and Lex Weaver. KnightCap: A Chess Program that Learns by Combining TD(lambda) with Minimax Search. Canberra, Australia, 1997.
6. G. Tesauro. Neurogammon: A Neural-Network Backgammon Program. Technical Report RC 15619 (69436), IBM T.J. Watson Research Center, 1990.
7. G. Tesauro. Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.
8. Gerald 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.
9. Nicol N. Schraudolph, Peter Dayan and Terrence J. Sejnowski. Learning to evaluate Go positions via temporal difference methods. Technical Report IDSIA-05-00, 2000.
10. Nicol N. Schraudolph and Peter Dayan and Terrence J. Sejnowski. Temporal Difference Learning of Position Evaluation in the Game of Go. In *Advances in Neural Information Processing Systems*, volume 6, pages 817–824. Morgan Kaufmann Publishers, Inc. 1994.
11. Dahl, F. A. Honte, a Go-playing program using neural nets. In *Proceedings of the 16th International Conference on Machine Learning, Machine Learning in Games* (ICML–99), Slovenia, 1999.
12. Epstein, S. Toward an ideal trainer. *Machine Learning*, 15:251–277, 1994.
13. Epstein, S. Learning to play expertly: A tutorial on Hoyle. In *Machines that Learn to Play Games*, Chapter 8, pp. 153–178. Huntignton, NY: Nova Science Publishers.
14. Thrun, S. Learning to play the game of chess. In *Advances in Neural Information Processing Systems 7*, pp. 1069–1076. Cambridge, MA: The MIT Press, 1995.
15. Thrun, S. The role of exploration in learning control. In *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*, 1992.
16. Mark C. Torrance and Michael P. Frank and Carl R. Witty. An Abalone Position For Which the Game is Undefined, draft report, February 1992.
17. F. H. Hsu. IBM's Deep Blue Chess Grandmaster Chips, In *IEEE Micro*, pp. 70–81, March-April 1999.
18. A. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3:210–229, 1959.
19. G. Tesauro. Practical Issues in temporal difference learning. *Machine Learning*, 8:257–278, 1992.
20. Tesauro, G. Programming backgammon using self-teaching neural nets. Artificial Intelligence, 134:181-199, 2002.

21. Anton Leouski. Learning of Position Evaluation in the Game of Othello. Technical Report UM-CS-1995-023, Amherst, MA, 1995
22. J. Schaeffer. *One jump ahead*. Springer-Verlag, New York, 1997.
23. Jonathan Schaeffer, Markian Hlynka, and Vili Jussila, Temporal Difference Learning Applied to a High-Performance Game-Playing Program, *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 529–534, 2001.
24. Donald F. Beal and Martin C. Smith. Temporal difference learning for heuristic search and game playing. In *Information Sciences*, 122(1):3–21, 2000. Special Issue on Heuristic Search and Computer Game Playing.
25. Donald F. Beal and Martin C. Smith. Temporal coherence and prediction decay in td learning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence* (IJCAI–99), pages 564–569, 1999.
26. T. Yoshioka, S. Ishii, M. Ito. Strategy Acquisition for the game Othello based on reinforcement learning, *IEICE Transactions on Inf. and Syst.*, Vol. E82 D, No.12 December 1999.
27. Aichholzer, O., Aurenhammer, F. and Werner, T. *Algorithmic Fun: Abalone*. Institut for Theoretical Computer Science, Graz University of Technology, 2002, Austria.
28. Jaap van der Herik, H., Uiterwijk, Jos W.H.M., van Rijswijck, J. Games solved: Now and in the future. *Artificial Intelligence*, 134:277–311, 2002.
29. B. Sheppard. World-championship-caliber Scrabble. *Artificial Intelligence*, 134:241–275, 2002.
30. Levinson, R. and Weber, R. Chess Neighborhoods, Function Combination and Reinforcement Learning. In T. A. Marsland and I. Frank, editors, *Computers and Games: Proceedings of the 2nd International Conference (CG-00), volume 2063 of Lecture Notes in Computer Science*, pages 133–150, Hamamatsu, Japan, 2001. Springer-Verlag.
31. Levinson, R. and Weber, R. J. Pattern-level Temporal Difference Learning, Data Fusion and Chess. In *SPIE'S 14th Annual Conference on Aerospace/Defense Sensing and Controls: Sensor Fusion: Architectures, Algorithms and Applications IV*, 2000.
32. Mihatsch, O. and Neuneier, R. Risk-Sensitive Reinforcement Learning. *Machine Learning*, 49:267–290, 2002.