Reinforcement Learning and its Application to Othello

Nees Jan van Eck, Michiel van Wezel *

Econometric Institute, Faculty of Economics, Erasmus University Rotterdam, P.O. Box 1738, 3000 DR, Rotterdam, The Netherlands

Econometric Institute Report EI 2005-47

Abstract

In this article we describe reinforcement learning, a machine learning technique for solving sequential decision problems. We describe how reinforcement learning can be combined with function approximation to get approximate solutions for problems with very large state spaces.

One such problem is the board game Othello, with a state space size of approximately 10^{28}. We apply reinforcement learning to this problem via a computer program that learns a strategy (or policy) for Othello by playing against itself. The reinforcement learning policy is evaluated against two standard strategies taken from the literature with favorable results.

We contrast reinforcement learning with standard methods for solving sequential decision problems and give some examples of applications of reinforcement learning in operations research and management science from the literature.

Key words: Artificial Intelligence, Reinforcement Learning, Q-learning, Multiagent Learning, Markov Decision Processes, Dynamic Programming, Neural Networks, Game Playing, Gaming, Othello.

1 Introduction

Many decision problems that we face in real life are sequential in nature. In these problems the payoff does not depend on an isolated decision but rather on

* Corresponding author. Phone +31 10 4081339. Fax +31 10 4089167.
Email addresses: nvaneck@few.eur.nl (Nees Jan van Eck), mvanwezel@few.eur.nl (Michiel van Wezel).

a sequence of decisions. In order to maximize total payoff, the decision maker may have to sacrifice immediate payoff in order to receive greater rewards later on. Finding a policy for making good sequential decisions is an interesting problem. Ideally, such a policy should indicate what the best decision is in each possible situation (or state) the decision maker can encounter.

A well known class of sequential decision problems are the Markov decision processes (MDPs), described in detail in Section 2. Their most important property is that the optimal decision in a given situation is independent of earlier situations the decision maker encountered. MDPs have found widespread application in operations research and management science. For a review see, e.g., [24].

For MDPs there exist a number of algorithms that find the optimal policy, collectively known as dynamic programming methods. A problem with dynamic programming methods is that they are unable to deal with problems in which the number of possible states is extremely high. Another problem is that dynamic programming requires exact knowledge of the problem characteristics, as will be explained in Section 2.

A relatively new class of algorithms, known as reinforcement learning algorithms (see, e.g., [18,9]), may help to overcome some of the problems associated with dynamic programming methods. Multiple scientific fields have made contributions to reinforcement learning — machine learning, operations research, control theory, psychology, and neuroscience, to name but a few. Reinforcement learning has been applied successfully in a number of areas, which has produced some successful practical applications. These applications range from robotics and control to industrial manufacturing and combinatorial search problems such as computer game playing (see, e.g., [9]). One of the most convincing applications is TD-gammon, a system that learns to play the game of Backgammon by playing against itself and learning from the results, described by Gerald Tesauro in [19,20]. TD-gammon reaches a level of play that is superior to even the best human players.

Recently, there has been some interest in the application of reinforcement learning algorithms to problems from the fields of management science and operations research. An interesting paper, for example, is [8], where reinforcement learning is applied to airline yield management and the aim is to find an optimal policy for the denial/acceptance of booking requests for seats in various fare classes. A second example is [5], where reinforcement learning is used to find a (sub)optimal control policy for a group of elevators. In both the above papers, the authors report that reinforcement learning based methods outperform the best and often used standard algorithms. A marketing application is described in [16], where a target selection decision in direct marketing is seen as a sequential problem. Other examples from management science

literature are [6,7], which are more methodologically oriented.

The purpose of this paper is to introduce the reader to reinforcement learning. We hope to convince the reader of the usefulness of the method. To achieve this, we will perform some experiments in which reinforcement learning is applied to a sequential decision problem with a huge state space: the board game Othello. In the experiments different reinforcement learning agents will learn to play the game of Othello without the use of any knowledge provided by human experts.

The remainder of this paper is structured as follows. In Section 2 we give a short introduction to reinforcement learning and sequential decision problems. We describe a frequently used reinforcement learning algorithm, Q-learning, in detail. In Section 3 we explain the game of Othello. In Section 4 we discuss the players used in our Othello experiments. Section 5 describes the experiments that we performed and the results obtained, and finally, Section 6 gives a summary, some conclusions, and an outlook.

2 Reinforcement Learning and Sequential Decision Problems

In this section we give a brief introduction to reinforcement learning and sequential decision problems. The reader is referred to [18,9,13] for a more extensive discussion of these subjects.

We describe reinforcement learning from the 'intelligent agent' perspective [17]. An intelligent agent is an autonomous entity (usually a computer program) that repeatedly senses inputs from its environment, processes these inputs and takes actions upon its environment. Many learning problems can conveniently be described using the agent perspective without altering the problem in an essential way.

In reinforcement learning, the agent/environment setting is as follows. At each moment, the environment is in a certain state. The agent observes this state, and depending solely on that state, the agent takes an action. The environment responds with a successor state and a reinforcement (also called a reward). Figure 1 shows a schematic representation of this sense-act cycle.

The agent's task is now to learn optimal actions, i.e., actions that maximize the sum of immediate reward and (discounted) future rewards. This may involve sacrificing immediate reward to obtain a greater cumulative reward in the long term or just to obtain more information about the environment.

We now give a more formal description of the reinforcement learning problem.

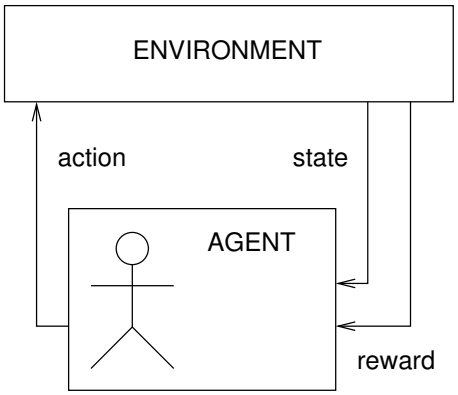

Figure 1. A reinforcement learning agent interacting with its environment.

At time t the agent observes state $s_t \in S$ of its environment and performs action $a_t \in A$, where S and A denote the sets of possible states and actions, respectively. The environment provides feedback in the form of a reinforcement (a reward) r_t. Also, a state transition takes place in the environment, leading to state s_{t+1}. So, an action puts the environment in a new state where the agent selects a new action, and in this way the cycle continues. The task of the learning agent is to learn a mapping $\pi : S \rightarrow A$ from states onto actions, often called a policy or strategy [1], that selects the best action in each state.

The expected value of the cumulative reward achieved by following an arbitrary policy π from an arbitrary initial state s_t is given by

$$V^{\pi}(s_t) = \mathrm{E}\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i}\right]. \tag{1}$$

where r_{t+i} is the reward received by taking an action in state s_{t+i} using policy π, and $\gamma \in [0, 1)$ is the discount factor that determines the relative value of delayed versus immediate rewards. The expectation is necessary because the rewards may be nondeterministic. Rewards received i time steps into the future are discounted by a factor γ^i. If $\gamma = 0$, only the immediate rewards are considered. As γ is set closer to 1, future rewards are given greater emphasis relative to immediate rewards. The function V^{π} is called the state-value function for policy π. It is also referred to as the utility function in the literature.

Using the state-value function $V^{\pi}(s)$ the learning task can be defined as follows. The agent must learn an optimal policy, i.e., a policy π^* which maximizes $V^{\pi}(s)$ for all states s

$$\pi^* = \operatorname*{argmax}_{\pi} V^{\pi}(s), \quad \forall s \in S. \tag{2}$$

To simplify notation, we will denote the state-value function $V^{\pi^*}(s)$ of such an optimal policy by $V^*(s)$. $V^*(s)$ is called the optimal value function.

[1] We will use both terms in the remainder.

The learning task faced by a reinforcement learning agent is usually assumed to be a Markov decision process (MDP). In an MDP both state transitions and rewards depend solely on the current state and the current action. There is no dependence on earlier states or actions. This is referred to as the Markov property or independence of path property. Accordingly, the reward and the new state are determined by $r_t = r(s_t, a_t)$ and $s_{t+1} = \delta(s_t, a_t)$. The reward function $r(s_t, a_t)$ and the state-transition function $\delta(s_t, a_t)$ may be nondeterministic.

If the agent knew the optimal value function V^*, the state-transition probabilities, and the expected rewards, it could easily determine the optimal action by applying the maximum expected value principal, i.e., by maximizing the sum of the expected immediate reward and the (discounted) expected value of the successor state, which captures the expected rewards from that point onwards

$$\pi^*(s) = \operatorname*{argmax}_{a \in A} \mathrm{E}\left[r(s,a) + \gamma V^*(\delta(s,a))\right] \tag{3}$$

$$= \operatorname*{argmax}_{a \in A} \left(\mathrm{E}\left[r(s,a)\right] + \gamma \sum_{s' \in S} T(s,a,s') V^*(s') \right), \tag{4}$$

where $T(s, a, s')$ denotes the transition probability from state s to state s' when action a is executed.

Notice that the values of a state and its successors are related as follows

$$V^*(s) = \mathrm{E}\left[r(s, \pi^*(s)) + \gamma V^*(\delta(s, \pi^*(s)))\right] \tag{5}$$

$$= \max_{a \in A} \left(\mathrm{E}\left[r(s,a)\right] + \gamma \sum_{s' \in S} T(s,a,s') V^*(s') \right). \tag{6}$$

Equation (5) and Equation (6) are the well known Bellman equations [2]. Solving these equations (one for each state) gives a unique state-value for each state. Unfortunately, these equations are nonlinear due to the presence of the 'max' operator and therefore hard to solve. The usual way of solving these equation is by means of dynamic programming techniques such as value iteration and policy iteration (see, e.g., [18,17]).

Reinforcement learning and dynamic programming are closely related, since both approaches are used to solve MDPs. The central idea of both reinforcement learning and dynamic programming is to learn value functions, which in turn can be used to identify the optimal policy. Despite this close relationship, there is an important difference between them. In reinforcement learning an agent does not necessarily know the reward function and the state-transition function. Both the reward and the new state that result from an action are determined by the environment, and the consequences of an action must be observed by interacting with the environment. In other words, reinforcement

learning agents are not required to possess a model of their environment. This aspect distinguishes reinforcement learning from dynamic programming, in which perfect knowledge of the reward function and the state-transition function is required. To converge to an optimal policy, in dynamic programming a number of computational iterations must be performed, rather than moving in the real environment and observing the results.

Policy iteration and value iteration are two popular dynamic programming algorithms. Either of these methods can be used to reliably compute optimal policies and value functions for finite MDPs given complete knowledge of the MDP [18]. In the next subsection we will discuss Q-learning. This is a reinforcement learning algorithm that does not need such a model to find an optimal policy in an MDP.

2.1 Q-Learning

Q-learning [22,23] is a reinforcement learning algorithm that learns the values of a function $Q(s, a)$ to find an optimal policy. The values of the function $Q(s, a)$ indicate how good it is to perform a certain action in a given state. The function $Q(s, a)$, also called Q-function, is defined as the reward received immediately upon executing action a from state s, plus the discounted value of the rewards obtained by following an optimal policy thereafter

$$Q(s, a) = \mathrm{E}\left[r(s, a) + \gamma V^*\left(\delta(s, a)\right)\right]. \tag{7}$$

If the Q-function is known, an optimal policy π^* is given by

$$\pi^*(s) = \underset{a \in A}{\operatorname{argmax}}\, Q(s, a). \tag{8}$$

This shows that an agent which knows the Q-function does not need to know the reward function $r(s, a)$ and the state-transition function $\delta(s, a)$ to determine an optimal policy π^*, in contrast with policy iteration and value iteration.

Notice that $V^*(s)$ and $Q(s, a)$ are related as follows

$$V^*(s) = \max_{a \in A} Q(s, a). \tag{9}$$

A recursive definition of the Q-function can be defined by substituting Equation (9) into Equation (7)

$$Q(s, a) = \mathrm{E}\left[r(s, a) + \gamma \max_{a' \in A} Q(\delta(s, a), a')\right]. \tag{10}$$

The Q-learning algorithm is based on this definition of the Q-function. An agent that uses the Q-learning algorithm to learn the Q-function, iteratively

For all states $s \in S$ and all actions $a \in A$ initialize $\hat{Q}(s, a)$ to an arbitrary value
Repeat (for each trial)
  Initialize the current state s
  Repeat (for each step of trial)
    Observe the current state s
    Select an action a using a policy π
    Execute action a
    Receive an immediate reward r
    Observe the resulting new state s'
    Update $\hat{Q}(s, a)$ according to Equation (11)
    $s \leftarrow s'$
  Until s is a terminal state

Figure 2. The Q-learning algorithm [23].

approximates the Q-function. In each iteration of the algorithm the agent observes the current state s, chooses some action a, executes this action a, then observes the resulting reward $r = r(s, a)$ and the new state $s' = \delta(s, a)$. It then updates its estimate of the Q-function, denoted by $\hat{Q}$, according to the training rule

$$\hat{Q}(s, a) \leftarrow (1 - \alpha)\, \hat{Q}\,(s, a) + \alpha \left(r + \gamma \max_{a' \in A} \hat{Q}\,(s', a') \right), \tag{11}$$

where $\alpha \in [0, 1)$ is the learning rate parameter. Figure 2 shows the complete Q-learning algorithm. Watkins and Dayan [23] have proved that the agent's estimated Q-values will converge to the true Q-values with probability one under the assumptions that the environment is a stable MDP with bounded rewards $r(s, a)$, the estimated Q-values are stored in a lookup table and are initialized to arbitrary finite values, each action is executed in each state an infinite number of times on an infinite run, $\gamma \in [0, 1)$, $\alpha \in [0, 1)$, and α is decreased to zero appropriately over time.

2.1.1 Learning the Q-Function by a Neural Network

The estimated Q-values should be stored somewhere during the estimation process and thereafter. The simplest storage form is a lookup table with a separate entry for every state-action pair. The problem of this method is its space complexity. Problems with a large state-action space lead to slow learning and to large tables with Q-values, which, in the worst case, cannot be stored in a computer memory. In the case of Othello we would need approximately 10^{58} entries [1,21], which is clearly beyond the memory capacity of even the most powerful supercomputer.

To cope with the problem of large state-action spaces, a function approximation method, such as a neural network or a decision tree, can be used to

Initialize all neural network (NN) weights to small random numbers
Repeat (for each trial)
  Initialize the current state s
  Repeat (for each step of trial)
    Observe the current state s
    For all actions a' in s use the NN to compute $\hat{Q}(s, a')$
    Select an action a using a policy π
    $Q^{\text{output}} \leftarrow \hat{Q}(s, a)$
    Execute action a
    Receive an immediate reward r
    Observe the resulting new state s'
    For all actions a' in s' use the NN to compute $\hat{Q}(s', a')$
    According to Equation (11) compute $Q^{\text{target}} \leftarrow \hat{Q}(s, a)$
    Adjust the NN by backpropagating the error $(Q^{\text{target}} - Q^{\text{output}})$
    $s \leftarrow s'$
  Until s is a terminal state

Figure 3. The Q-learning algorithm using a neural network function approximator.

'store' the Q-values. In our experiments we use neural networks as a function approximator, and therefore we consider this method in more detail. We assume that the reader has basic knowledge about feedforward neural networks. For an introduction to neural networks see, e.g., [13].

During the Q-learning process, the neural network learns a mapping from state descriptions onto Q-values. This is done by computing a 'target' Q-value according to (11) and using the backpropagation algorithm to minimize the discrepancy between the target Q-value and the estimated Q-value computed by the neural network. The complete algorithm is given in Figure 3.

It is important to note that, in problems with large state-action spaces, the neural network is trained based on visits to only a tiny part of the state-action space. The use of a neural network makes it possible to generalize over states and actions. Based on experience with previously visited state-action pairs, the neural network is able to give an estimate of the Q-value for an arbitrary state-action pair.

How does a neural network pull this trick? It is likely that the internal layers of the neural network learn to extract features that are useful in assessing the Q-values of state-action pairs. In Othello an example of such a feature might be a numerical representation of the rule "When the opponent occupies corner positions this is always bad for the value function". Based on these self-discovered features the final layer of the neural network will make estimates of Q-values. To facilitate the neural network's task, one can offer additional features on the input side besides the state description, which will probably lead to a better policy being learned.

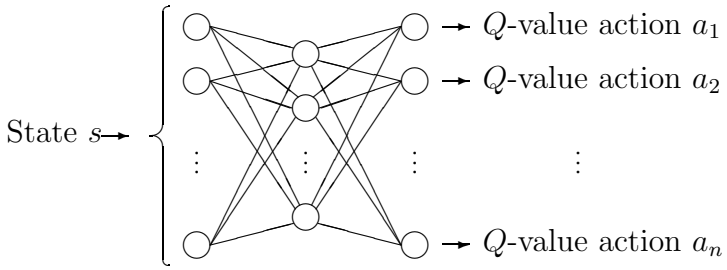

Figure 4. A single multi-layer feed-forward neural network with a distinct output for each action.

When learning the Q-function using a multi-layer feed-forward neural network, it is possible to use a distinct network for each action, a single network with a distinct output for each action, or a single network with both the state and the action as input and the Q-value as output [9]. Because we do not use this last approach in our experiments, we only describe the first two approaches in detail.

The input of a single network with a distinct output for each action consists of one or more units to represent a state. The output of the network consists of as many units as there are actions that can be chosen. Figure 4 illustrates the layout of such a network. When a single network is used, generalization over both states and actions is possible.

If there is a distinct network for each action, the input of each network consists of one or more units to represent a state. Each network has only one output, which is considered as the Q-value associated with the state that is given as input to the network and with the action that is represented by the network. Figure 5 illustrates the layout of multiple networks associated with the actions a_1, a_2, ..., a_n. In this case, only generalization over the states is possible.

Q-learning using neural networks to store the Q-values can solve larger problems than Q-learning using the lookup table method, but it is not guaranteed to converge [23]. The problem associated with the use of neural networks in Q-learning results from the fact that these networks perform non-local changes to the Q-function, while Q-learning requires that updates to the Q-function are local. When updating the value of a state-action pair, the network may destroy the learned value of some other state-action pairs. This is one of the reasons why the neural network method is not guaranteed to converge to the actual Q-values.

2.1.2 Action Selection

The algorithms in Figure 2 and in Figure 3 do not specify how actions are selected by an agent. One of the challenges that arises here is the trade-off

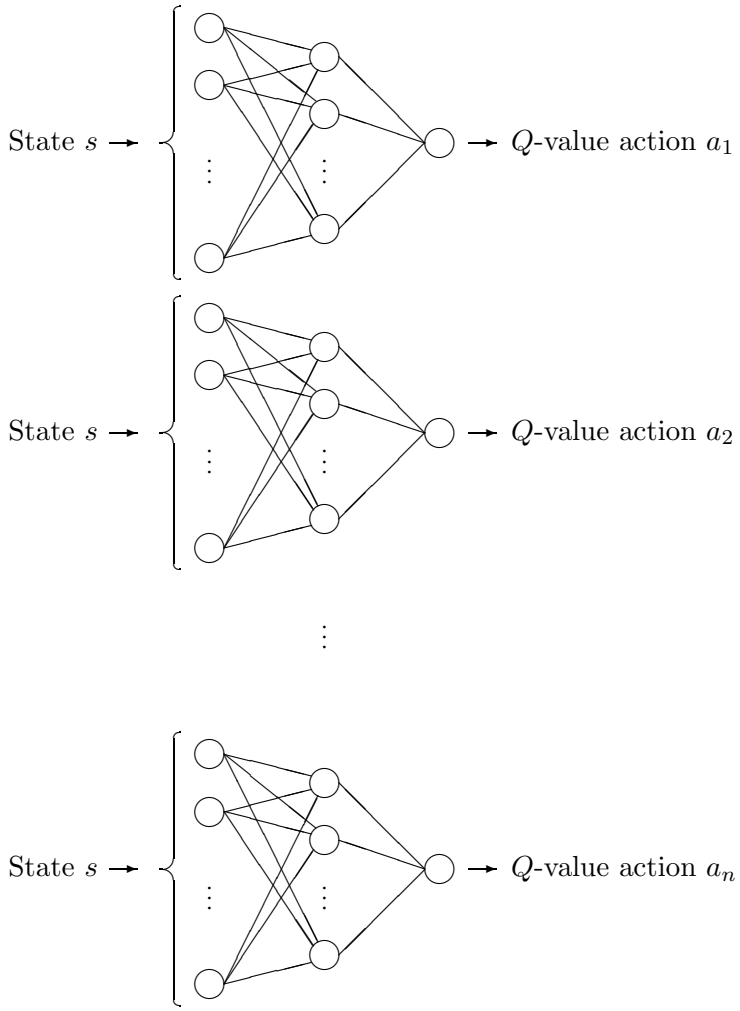

Figure 5. A distinct multi-layer feed-forward neural network for each action.

between exploration and exploitation. An agent is exploiting its current knowledge of the Q-function when it selects the action with the highest estimated Q-value. If instead the agent selects one of the other actions, it is exploring because it improves the estimate of the action's Q-value.

In methods for action selection a trade-off between exploration and exploitation has to be made because an agent wants to exploit what he already knows in order to obtain a reward, but it also wants to explore in order to make better action selections in the future. It is not possible to explore and to exploit at the same time, hence a conflict will occur between exploration and exploitation [18].

There are many methods for balancing exploration and exploitation. In our experiments, we use the so-called softmax action selection method, where the agents choose actions probabilistically based on their estimated Q-values using a Boltzmann distribution. Given a state s, an agent tries out action a with

probability

$$\mathrm{P}(a) = \frac{\exp\left(\hat{Q}(s,a)/T\right)}{\sum_{a' \in A} \exp\left(\hat{Q}(s,a')/T\right)}, \tag{12}$$

where T is a positive parameter called the temperature that controls the amount of exploration. A very low temperature tends to greedy action selection, i.e. choosing the action for which the estimated Q-value is greatest. A very high temperature results in nearly random action selection. The temperature is usually lowered gradually over time, which leads to a gradual transition from exploration to exploitation.

2.1.3 Multiagent Environments

Q-learning is not guaranteed to converge in non-stationary environments. There are various reasons why an environment may be non-stationary. One possibility is that there are multiple agents active in the same environment. In Section 5 we apply Q-learning simultaneously to two Othello playing agents, each agent being the opponent of the other. Each agent perceives the actions of its opponent as part of the environment, so the agents are unaware who they are playing against. Each agent learns and thus alters its policy. This is perceived as non-stationarity of the environment by the opposing agent.

Q-learning was developed for stationary environments. Due to non-stationarity, convergence to an optimal policy cannot be guaranteed in multiagent settings. Despite this fact we use this algorithm in our experiments in Section 5. Another approach in multiagent reinforcement learning is to use a reinforcement learning algorithm that has been adapted for finding optimal policies in multiagent problems (see, e.g., [12,3]).

3 Othello

In this section we give an introduction to the game of Othello. We will describe the game, and discuss some basic strategies.

3.1 Description of the Game

Othello is a two-player deterministic zero-sum game with perfect information. Its state space size is approximately 10^{28} [1,21] and its length is 60 moves at most. Othello is played by two players on an 8×8-board using 64 two-sided discs that are black on one side and white on the other. One player places the

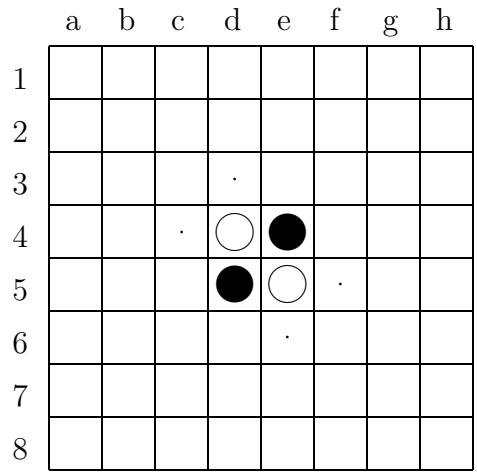

Figure 6. Othello start position.

discs on the board with the black side up, the other places the discs on the board with the white side up.

Initially the board is empty except for the central four squares, which have black discs on the two squares on the main diagonal (at d5 and e4) and white discs on the two squares on the other diagonal (at d4 and e5). Figure 6 shows this starting position.

The players move in turn with black beginning. A legal move is made by placing a disc on an empty square so that in at least one direction (horizontally, vertically, or diagonally) from the square played on, there is a sequence of one or more of the opponent's discs followed by one's own disc. The opponent's discs in such a sequence are turned over and become one's own color.

For example we consider the position in Figure 7 with the black player to move. Its legal moves, which are marked with a dot, are d2, d6, e2, f2, f6, and f7. If it plays move d6, the white discs on d5 and e5 will be turned over and become black. Figure 8 shows the board after this move.

If a player cannot make a legal move, it has to pass. If a player is able to make a legal move, however, then passing is not allowed. The game ends when neither player can make a legal move. Usually this occurs when all 64 squares have been filled, but in some cases there are empty squares to which neither player can legally play. When the game has ended, the discs are counted. The winner is the player that has more discs than its opponent. If both players have the same number of discs, the game has ended in a draw.

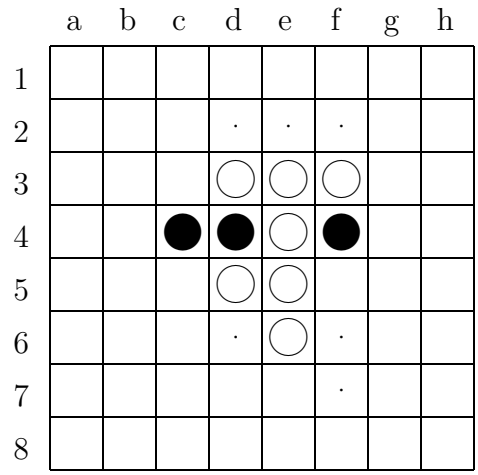

Figure 7. Othello example: Black to move.

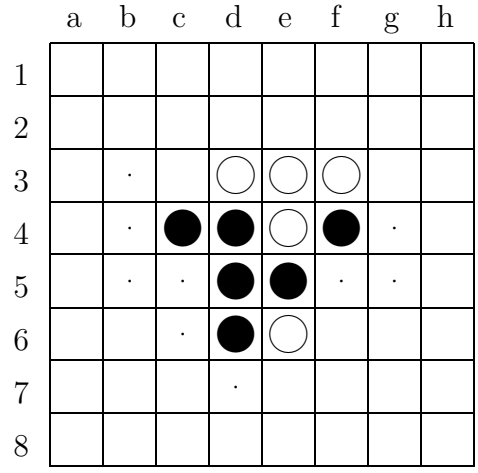

Figure 8. Othello example: Position after move d6.

3.2 Strategies

A game of Othello can be split up into three phases: the begin game (which ends after about 20 to 26 moves), the middle game, and the end game (which starts somewhere between 16 to 10 moves before the end). The end game is simply played by maximizing one's own discs, while minimizing the opponent's discs. The goal of the middle game is to strategically position the discs on the board so that they can be converted during the end of the game into a large number of discs that cannot be flipped back by the opponent. Such discs are called stable discs. There are two basic middle game strategies in Othello [14]: the positional strategy, and the mobility strategy.

The positional strategy emphasizes the importance of specific disc positions on the board. Positions such as corners and edges are valuable, whereas others should be avoided. Corners are especially valuable because once taken, they

can never be flipped back by the opponent. Obtaining a corner disc in the beginning of the game, or in the middle game, usually means that it will be possible to use that corner disc to get many more stable discs. A player using the positional strategy tries to maximize its valuable discs while minimizing the opponent's valuable discs.

The mobility strategy is based on the idea that the easiest way to capture a corner is to force the opponent to make moves that deliver that corner. The best way to force the opponent to make such bad moves is to limit the number of moves available to the opponent, i.e., to minimize mobility. The number of moves available to the opponent can be limited by minimizing and clustering one's own discs.

We will use agents playing these strategies as benchmark agents in our experiments in Section 5. For a more extensive description of Othello strategies we refer to [10].

4 The Othello Playing Agents

Our goal is to train different reinforcement learning agents to play the game of Othello without the use of any knowledge on the values of board positions provided by human experts. To reach this goal, in our experiments two reinforcement learning agents play the game of Othello against each other. Both agents use the Q-learning algorithm to learn which move is best to play in a given state of the game. As stated before, convergence is not guaranteed because this is a multiagent setting. During learning, the Q-learning agents are periodically evaluated against benchmark agents playing the positional strategy and the mobility strategy.

In total, we use three types of players: the positional player, the mobility player, and the Q-learning player. The different types of players play with different strategies, i.e. they use different evaluation functions. For every player the evaluation function gives a numerical value of $+1$, -1, and 0 for the terminal states that correspond to wins, losses, and draws, respectively. Until the end of the game the different types of players use different evaluation functions. Below, we give a more extensive description of the players.

4.1 Positional Player

A positional player does not learn. Until the end game, this player plays according to the positional strategy as described in Section 3. The player's ob-

	a	b	c	d	e	f	g	h
1	100	-20	10	5	5	10	-20	100
2	-20	-50	-2	-2	-2	-2	-50	-20
3	10	-2	-1	-1	-1	-1	-2	10
4	5	-2	-1	-1	-1	-1	-2	5
5	5	-2	-1	-1	-1	-1	-2	5
6	10	-2	-1	-1	-1	-1	-2	10
7	-20	-50	-2	-2	-2	-2	-50	-20
8	100	-20	10	5	5	10	-20	100

Figure 9. Othello position values.

jective during the begin game and the middle game is to maximize its own valuable positions (such as corners and edges) while minimizing its opponent's valuable positions. To achieve this, until the end game the positional player makes use of the following evaluation function

$$\text{EVAL} = w_{\text{a1}}v_{\text{a1}} + w_{\text{a2}}v_{\text{a2}} + \ldots + w_{\text{a8}}v_{\text{a8}} + \ldots + w_{\text{h8}}v_{\text{h8}} \tag{13}$$

where w_i is equal to $+1$, -1, or 0 if square i is occupied by a player's own disc, is occupied by an opponent's disc, or is unoccupied, respectively. v_i is equal to the value of square i as shown in Figure 9.

The values of the corners are the highest (100 points) and the values of the so-called X-squares (b2, b7, g2, and g7) are the lowest (-50 points), and thus they are the best and worst positional moves to play in the game. The reason for the -50 and -20 values for the X-squares and the C-squares (a2, a7, b1, b8, g1, g8, h2, and h7) respectively, is that they potentially allow the opponent to obtain the corner by flipping a disc placed there by the player. Also, these squares make it impossible for the player to obtain the corner from that direction for the rest of the game. The low values of the X-squares and C-squares force the player to avoid playing there as much as possible.

The end game begins when at least 80% of the board squares are occupied, or when all the corner squares are occupied. During the end game, the player's objective is to maximize the number of its own discs, while minimizing the number of opponent's discs. To achieve this, during the end game the positional player makes use of the following evaluation function in nonterminal board states

$$\text{EVAL} = n_{\text{player}} - n_{\text{opponent}} \tag{14}$$

where n_{player} is the number of squares occupied by the player's own discs and n_{opponent} is the number of squares occupied by the opponent's discs.

4.2 Mobility Player

A mobility player does not learn. Until the end game, this player plays according to the mobility strategy as described in Section 3. Just as in the positional strategy, in this strategy the corner positions are of great importance. Furthermore, in this strategy the concept of mobility is important. Mobility is defined as the number of legal moves a player can make in a certain position. The player's objective for the begin and middle game is therefore to maximize the number of corner squares occupied by its own discs while minimizing the number of corner squares occupied by opponent's discs, and to maximize its own mobility while minimizing its opponent's mobility. To achieve this, until the end game the mobility player makes use of the following evaluation function in nonterminal board states

$$\mathrm{EVAL}(s) = w_1 \left(c_{\mathrm{player}} - c_{\mathrm{opponent}}\right) + w_2 \frac{m_{\mathrm{player}} - m_{\mathrm{opponent}}}{m_{\mathrm{player}} + m_{\mathrm{opponent}}} \tag{15}$$

where each w_i is a weight, c_{player} is the number of corner squares occupied by the player's discs, c_{opponent} is the number of corner squares occupied by the opponent's discs, m_{player} is the mobility of the player, and m_{opponent} is the mobility of the opponent. The weights w_1 and w_2 are set to the values 10 and 1, respectively.

For the mobility player the end game begins at the same moment as for the positional player. During the end game, the players objective is identical to the one of the positional player. For this reason, during the end game the mobility player makes also use of the evaluation function given by Equation (14).

4.3 Q-Learning Player

The Q-learning player is the only agent exhibiting learning behavior: it uses the Q-learning algorithm (see Section 2.1) to learn which move is best to play in a given state of the game. The current state of the board (the placements of black and white discs on the board) is used as the state of the game. No special board features selected by human experts (e.g. each player's number of discs, each player's mobility) are used. Based on the state of the game the agent decides which action to execute. This action is the move it is going to play. The player's reward is 0 until the end of the game. Upon completing the game, its reward is $+1$ for a win, -1 for a loss, and 0 for a draw. The player aims to choose optimal actions leading to maximal reward.

The values of all parameters of the Q-learning algorithm have been chosen experimentally and may be suboptimal. The learning rate α of the Q-learning

algorithm is set to 0.1 and the discount factor γ is set to 1. The learning rate does not change during learning. The discount factor is set to 1 because we only care about winning and not about winning as fast as possible. A Q-learning player uses the softmax action selection method as described in Section 2.1.2. This selection method makes use of a temperature T that controls the amount of exploration. We take the temperature to be a decreasing function of the number of games n played so far

$$T = \begin{cases} ab^n & \text{if } ab^n \geq c, \\ 0 & \text{else,} \end{cases} \tag{16}$$

for given values of a, b, and c. If $T = 0$, no exploration takes place and the action with the highest estimated Q-value is always selected.

We implemented two different types of Q-learning players. Both of them use neural networks to learn the Q-function. We describe them in more detail below.

The single-NN Q-learner uses a single multi-layer feed-forward neural network with a distinct output for each action (see Section 2.1.1 for a description of this method and Figure 4 for the layout of such a neural network). The network has 64 input units. Using these units the state of the environment is presented to the network. The units correspond with the 64 squares of the board. The activation of an input unit is $+1$ for a player's own disc, -1 for a disc of the opponent, and 0 for an empty square. The network has one hidden layer of 44 tanh units and an output layer of 64 tanh units. Each output unit corresponds with an action (a square of the board). The value of an output unit is between -1 and 1 and corresponds with the Q-value of a move. Of course, when choosing an action, only the Q-values of legal moves are considered. The learning rate of the neural network (which should not be confused with the learning rate of the Q-learning algorithm) is set to 0.1 and no momentum is used. The weights of the network are initialized to random values drawn from a uniform distribution between -0.1 and 0.1. The constants (a, b, c) for the temperature annealing schedule are set to (1, 0.9999995, 0.002). Due to this schedule, there is a gradual transition from exploration to exploitation. After $12,429,214$ games, exploration stops and the player always exploits its knowledge.

The multi-NN Q-learner uses a distinct multi-layer feed-forward neural network for each action (see Section 2.1.1 for a description of this method and Figure 5 for the layout of such a neural network). The number of neural networks is equal to 64. Each network has 64 input units. In the same way as for a single-NN Q-learner these units are used to present the squares of the board to the player. Each network has one hidden layer of 36 tanh units and an output layer of 1 tanh unit. The value of an output unit is between -1 and 1 and corresponds with the Q-value of a move. The learning rate of

each neural network is set to 0.1 and no momentum is used. The weights of each network are initialized to random values drawn from a uniform distribution between -0.1 and 0.1. The temperature annealing schedule of this player is the same as the one used for the single-NN Q-learner. As a consequence, the different neural network Q-learners only vary in the way they 'store' Q-values. In this way we can draw a good comparison between both players.

Note that the total number of parameters used by the single-NN Q-learner and the multi-NN Q-learner are $5,632$ and $149,760$, respectively. A huge compression compared to the 10^{58} parameters that would be needed when using a lookup table.

During evaluation against benchmark players, both types of Q-learning players use an evaluation function that assigns a numerical value to a nonterminal state

$$\text{EVAL} = \max_a Q(s, a). \tag{17}$$

During evaluation the Q-learning agents do not use an exploration policy for action selection, but, as indicated by Equation (17), they use an optimal policy based on the estimated Q-values.

4.4 Java Applet

We implemented all agents described above in the programming language Java. An applet containing them is available at `http://www.few.eur.nl/few/people/mvanwezel/othelloapplet.html`. Figure 10 shows the applet, which can also be used by a human player to play against one of the intelligent agents.

5 Experiments and Results

We performed two experiments with agents using Q-learning for Othello. In the first experiment two single-NN Q-learners played against each other, in the second experiment two multi-NN Q-learners were used. In both experiments, $15,000,000$ games were played for training. After every $1,500,000$ training games, learning was turned off and both Q-learners were evaluated by playing 100 games against two benchmark players: a positional player and a mobility player.

Because during evaluation both the Q-learner and the benchmark player used a deterministic policy, the evaluation games had to be started from different

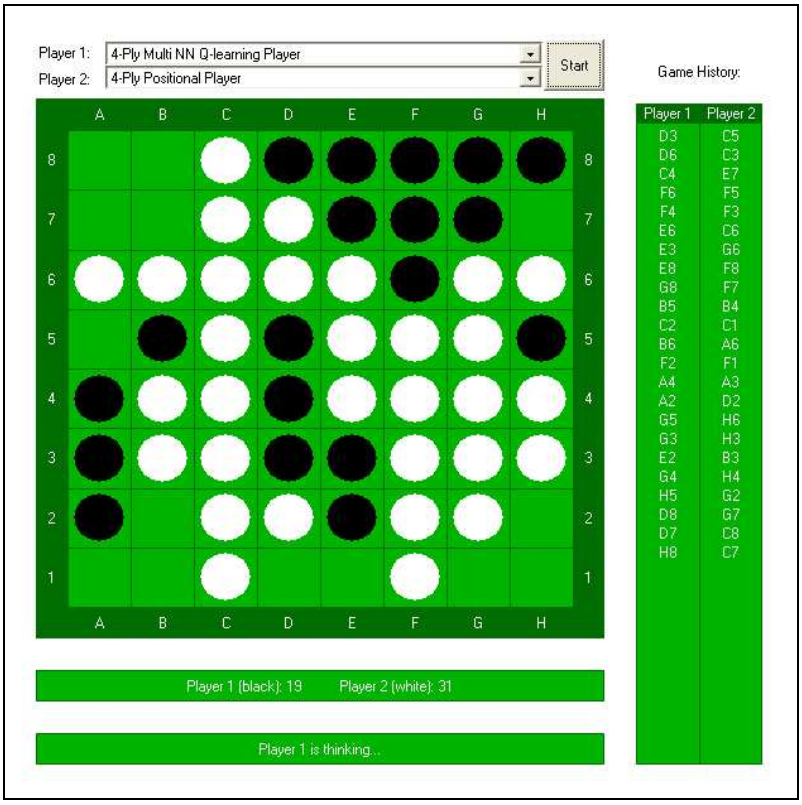

Figure 10. Screen-dump of the Othello applet.

positions. These different positions were generated by making four legal moves at random from the initial position as shown in Figure 6. The number of different board states that may result from four random moves at the beginning of an Othello game is 244. The same approach to evaluate an Othello player was also used in [14] and [11]. As performance measure we used the percentage of evaluation games that were not lost by the Q-learners.

During the evaluation phase in our experiments all agents made use of the appropriate evaluation function mentioned in Section 4, i.e. the various agent types used different evaluation functions. A four-ply look ahead search was used to improve the move quality for all players during evaluation. The well known minimax algorithm with alpha-beta pruning (see, e.g., [17]) was used to determine the move that is optimal given the agent's evaluation function and a four-level deep search of the game tree. We found that a look ahead of four moves is a good compromise between the strength of the players and the computation time. It is important to note that during training the Q-learning players did not use the minimax algorithm.

Table 1 and Figure 11 show the results of the first Othello experiment. The table shows the number of evaluation games won, lost, and drawn by the two single-NN Q-learners against positional and mobility players. The figure shows the percentage of evaluation games lost by the single-NN Q-learners against positional and mobility players. In the graphs the horizontal axis denotes the number of training games and the vertical axis denotes the percentage of the 100 evaluation games that were lost. The solid and dashed lines correspond to the Q-learner playing with black discs and the Q-learner playing with white discs, respectively.

First of all it may be noticed from the results that the Q-learners lost almost all evaluation games when they had not yet learned anything (thus when no training games had been played). Obviously, positional and mobility players are rather strong players.

	Against Positional Player						Against Mobility Player					
Iteration	Q-learner Black			Q-learner White			Q-learner Black			Q-learner White		
($\times 10^6$)	Won	Lost	Draw	Won	Lost	Draw	Won	Lost	Draw	Won	Lost	Draw
0.0	3	96	1	7	88	5	2	97	1	1	99	0
1.5	73	26	1	72	28	0	44	53	3	56	40	4
3.0	58	40	2	87	11	2	47	53	0	64	35	1
4.5	86	13	1	88	10	2	80	19	1	77	23	0
6.0	80	14	6	97	2	1	62	32	6	77	23	0
7.5	79	19	2	86	13	1	55	42	3	62	35	3
9.0	81	18	1	84	15	1	56	40	4	74	24	2
10.5	75	20	5	91	9	0	67	32	1	79	21	0
12.0	77	18	5	83	14	3	55	43	2	74	25	1
13.5	77	23	0	96	2	2	45	55	0	67	30	3
15.0	84	11	5	86	13	1	70	28	2	75	24	1

Table 1
Evaluation results of the single-NN Q-learner against benchmark players.

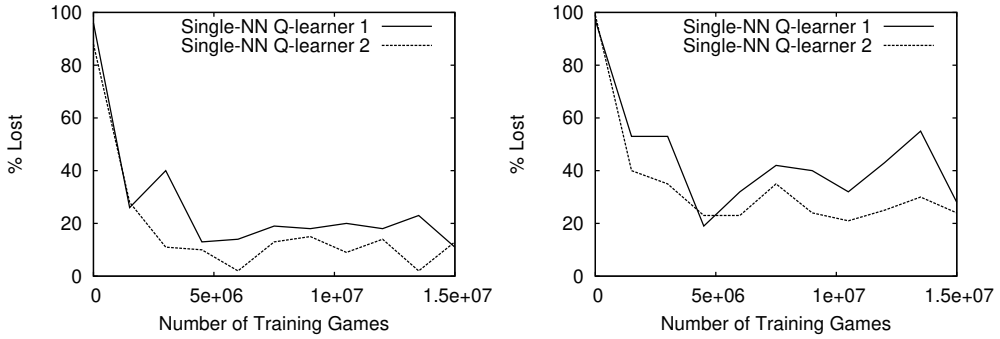

Figure 11. Evaluation results of the single-NN Q-learner against benchmark players.

It may also be noticed that after about $15,000,000$ training games the percentage lost by the Q-learners had decreased to approximately 12% against the positional benchmark player, and to approximately 26% against the mobility benchmark player. This result indicates that the Q-learners were able to learn to play the game of Othello. It is interesting to note that despite the fact that the Q-learners were not trained against the positional and mobility players, they were able to beat them most of the time.

We note that on average the percentage lost by the Q-learners against the mobility players was higher than against the positional players. This may indicate that it is more difficult for Q-learners to play against mobility players than against positional players.

From these results we can conclude that single-NN Q-learners are able to learn to play the game of Othello better than players that use a positional or mobility strategy.

5.2 Experiment 2: Learning to Play Othello with Multi-NN Q-Learners

The results of the second experiment are shown in Table 2 and Figure 12. The results of this experiment look the same as the ones we saw in the first experiment. A difference is the number of training games that the Q-learners needed to achieve the same results against the benchmark players. It seems that multi-NN Q-learners learn slower than single-NN Q-learners. Generalization over only states instead of generalization over both states and actions may be an explanation for this difference.

After $15,000,000$ training games, the multi-NN Q-learner lost approximately

	Against Positional Player						Against Mobility Player					
Iteration	Q-learner Black			Q-learner White			Q-learner Black			Q-learner White		
($\times 10^6$)	Won	Lost	Draw	Won	Lost	Draw	Won	Lost	Draw	Won	Lost	Draw
0.0	5	95	0	9	88	3	2	96	2	8	92	0
1.5	47	51	2	46	53	1	27	73	0	39	61	0
3.0	33	65	2	46	52	2	19	77	4	48	51	1
4.5	61	38	1	66	32	2	47	51	2	64	35	1
6.0	77	19	4	87	12	1	80	19	1	73	27	0
7.5	80	20	0	87	11	2	70	25	5	75	21	4
9.0	76	24	0	78	22	0	64	34	2	70	26	4
10.5	79	17	4	82	18	0	71	25	4	73	26	1
12.0	85	14	1	81	19	0	61	31	8	73	25	2
13.5	88	11	1	87	12	1	75	22	3	81	18	1
15.0	84	12	4	85	10	5	79	20	1	74	26	0

Table 2
Evaluation results of the multi-NN Q-learner against benchmark players.

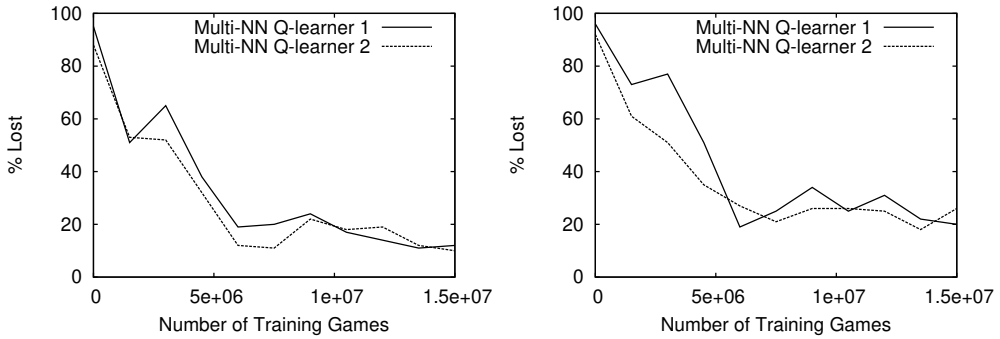

Figure 12. Evaluation results of the multi-NN Q-learner against benchmark players.

11% of the evaluation games against the positional benchmark player, and approximately 23% against the mobility benchmark player.

Just as from the results of the previous experiment, from the results of this experiment we can conclude that multi-NN Q-learners are also able to learn to play the game of Othello better than players that use a positional or mobility strategy.

6 Summary, Conclusions and Outlook

In this paper we discussed reinforcement learning, a machine learning method for sequential decision problems. Reinforcement learning is able to find approximate solutions to large sequential decision problems by making use of function approximation. We described Q-learning, a frequently used reinforcement learning algorithm.

As an example application, we applied Q-learning to the game of Othello. We aimed at studying the ability of different Q-learning agents to learn to play the game of Othello without the use of any knowledge provided by human experts. Othello has a huge state space (approximately 10^{28} possible states) and is therefore not solvable by traditional dynamic programming techniques.

In the Othello experiments we investigated two different types of Q-learning agents. We studied Q-learners that use a single neural network with a distinct output for each action and Q-learners that use a distinct neural network for each action. Q-learners that use a lookup table were not studied because that would have require too much memory. From the results of the experiments we conclude that the two different types of Q-learners are both able to learn to play the game of Othello better than players that use a straightforward positional or mobility strategy. It seems that Q-learners that use a single neural network learn to play Othello faster than Q-learners that use a distinct neural network for each action.

A first topic for future research is the use of an adapted version of the Q-learning algorithm that is able to find optimal policies in multiagent settings, as described by Littman in, e.g., [12]. His minimax Q-learning algorithm finds optimal policies in two-player zero-sum games using the minimax criterion. It may therefore be used to train Othello players. Another variation of reinforcement learning that may be of interest to us is the algorithm given in [15], which is guaranteed to converge using any form of function approximator.

The effects of the presentation of special board features to the Q-learning agents in order to simplify learning may also be studied, although this violates our goal of learning to play Othello without any knowledge provided by human experts. Interesting board features in the game of Othello may be, for example, patterns of squares comprising combinations of corners, diagonals, and rows. These board features capture important Othello concepts such as stability. Note that these features are easily recognized by human players due to their visual system, but to a computer they are just as hard to recognize as arbitrary board patterns, so offering them may help.

Furthermore, in future research it is interesting to investigate the performance of the Q-learning agents against human players. The evaluation against humans may be performed by playing with with the best trained Q-learners against human players in an Internet gaming zone (see, e.g., [4]).

Finally, we plan on studying potential applications of reinforcement learning in operations research and management science. Some of these applications have already been mentioned in the introduction. White [24] gives an overview of general MDP-applications in operations research. It may be possible to reformulate some of these problems, so that advantage can be taken from reinforcement learning's ability to generalize through function approximation. The trade-off between inaccuracy af the problem formulation level and inaccuracy at the solution level that arises in these cases is an interesting subject. The first form of inaccuracy occurs when the decision problem is delibaretely kept simple to keep the application of dynamic programming feasible. The second form occurs when estimated value functions are inaccurate through the use of function approximation as in reinforcement learning. For some problems, the latter form of inaccuracy may give a better solution.

References

[1] L.V. Allis. *Searching for Solutions in Games and Artificial Intelligence.* PhD thesis, University of Limburg, Maastricht, 1994.

[2] R. E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, New Yersey, 1957.

[3] Michael Bowling and Manuela Veloso. Multiagent learning using a variable learning rate. *Artificial Intelligence*, 136(2):215–250, April 2002.

[4] K. Chellapilla and D. B. Fogel. Evolving an expert checkers playing program without using human expertise. *IEEE Transactions on Evolutionary Computation*, 5(4):422–428, 2001.

[5] Robert H. Crites and Andrew G. Barto. Elevator group control using multiple reinforcement learning agents. *Machine Learning*, 33(2):235–262, 1998.

[6] T.K. Das, A. Gosavi, S. Mahadevan, and N. Marchalleck. Solving semi-Markov decision problems using average reward reinforcement learning. *Management Science*, 45(4):560–574, 1999.

[7] A. Gosavi. Reinforcement learning for long run average cost. *European journal of operational research*, 144:654–674, 2004.

[8] A Gosavi, N. Bandla, and T. K. Das. A reinforcement learning approach to a single leg airline revenue management problem with multiple fare classes and overbooking. *IIE Transactions*, 34(9):729–742, September 2002.

[9] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[10] M. le Comte. *Introduction to Othello*, 2000. Online document, available from `http://othello.nl/content/guides/comteguide/`.

[11] A. V. Leouski and P. E. Utgoff. What a neural network can learn about Othello. Technical Report UM-CS-1996-010, Computer Science Department, Lederle Graduate Research Center, University of Massachusetts, Amherst, MA., 1996.

[12] M. L. Littman. Value-function reinforcement learning in Markov games. *Journal of Cognitive Systems Research*, 2(1):55–66, 2001.

[13] T. M. Mitchell. *Machine learning*. McGraw-Hill, 1997.

[14] D. E. Moriarty and R. Miikkulainen. Discovering complex Othello strategies through evolutionary neural networks. *Connection Science*, 7(3):195–210, 1995.

[15] V. Papavassiliou and S. J. Russell. Convergence of reinforcement learning with general function approximators. In *Proceedings of the sixtheenth international joint conference on artificial intelligence (IJCAI-99)*, pages 748–757, Stockholm, 1999. Morgan Kaufmann.

[16] E. Pednault, N. Abe, and B. Zadrozny. Sequential cost-sensitive decision-making with reinforcement learning. In David Hand, Daniel Keim, and Raymond Ng, editors, *Proceedings of the Eighth International Conference on Knowledge Discovery and Data Mining (KDD'02)*, pages 259 – 268, Alberta, Canada, July, 23–25 2002. ACM.

[17] S. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach*. Prentice Hall, 2nd edition, 2003.

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

[19] G. Tesauro. TD-gammon, a self-teaching Backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.

[20] Gerald Tesauro. Programming Backgammon using self-teaching neural nets. *Artificial Intelligence*, 134(1-2):181–199, January 2002. Special Issue on Games, Computers and Artificial Intelligence.

[21] H. J. van der Herik, J. W. H. M. Uiterwijk, and J. van Rijswijck. Games solved: Now and in the future. *Artificial Intelligence*, 134:277–311, 2002.

[22] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, Cambridge, England, 1989.

[23] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

[24] D. J. White. A survey of applications of Markov decision processes. *The Journal of the Operational Research Society*, 44(11):1073–1096, 1993.