Available online at www.sciencedirect.com
ScienceDirect
Computers & Operations Research 35 (2008) 1999–2017
computers & operations research
www.elsevier.com/locate/cor

Application of reinforcement learning to the game of Othello

Nees Jan van Eck*, Michiel van Wezel

Erasmus School of Economics, Erasmus University Rotterdam, P.O. Box 1738, 3000 DR Rotterdam, The Netherlands

Available online 5 December 2006

Abstract

Operations research and management science are often confronted with sequential decision making problems with large state spaces. Standard methods that are used for solving such complex problems are associated with some difficulties. As we discuss in this article, these methods are plagued by the so-called curse of dimensionality and the curse of modelling. In this article, we discuss reinforcement learning, a machine learning technique for solving sequential decision making problems with large state spaces. We describe how reinforcement learning can be combined with a function approximation method to avoid both the curse of dimensionality and the curse of modelling. To illustrate the usefulness of this approach, we apply it to a problem with a huge state space—learning to play the game of Othello. We describe experiments in which reinforcement learning agents learn to play the game of Othello without the use of any knowledge provided by human experts. It turns out that the reinforcement learning agents learn to play the game of Othello better than players that use basic strategies.

Keywords: Dynamic programming; Markov decision processes; Reinforcement learning; *Q*-learning; Multiagent learning; Neural networks; Game playing; Othello

1. Introduction

Many decision making 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 a sequence of decisions. In order to maximize the total payoff, the decision maker may have to sacrifice immediate payoffs such that greater payoffs can be received 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 may encounter.

A well-known class of sequential decision making 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 state is independent of earlier states the decision maker encountered. MDPs have found widespread application in operations research and management science. For a review see, e.g. [1].

For MDPs there exist a number of algorithms that are guaranteed to find optimal policies. These algorithms are 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 high (also called the curse of dimensionality). Another problem is that dynamic programming requires exact knowledge of the problem characteristics (also called the curse of modelling), as will be explained in Section 2.

* Corresponding author.
E-mail addresses: nvaneck@few.eur.nl (N.J. van Eck), mvanwezel@few.eur.nl (M. van Wezel).

0305-0548/$ - see front matter © 2006 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2006.10.004

A relatively new class of algorithms, known as reinforcement learning algorithms (see, e.g. [2–5]), 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. Reinforcement learning has been applied 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. [3]). 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 [6–8]. TD-Gammon reaches a level of play that is almost as good as the best human players.

Recently, there has been some interest in the application of reinforcement learning algorithms to problems in the fields of operations research and management science. For example, an interesting article is [9], 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. Another example is [10,11], where reinforcement learning is used to find an optimal control policy for a group of elevators. In the above examples, the authors report that reinforcement learning methods outperform frequently used standard algorithms. A marketing application is described in [12], where a target selection decision in direct marketing is seen as a sequential decision problem. Other examples from the management science literature are [13,14], which are more methodologically oriented.

The purpose of this article is to introduce the reader to reinforcement learning and to convince the reader of the usefulness of this method in helping to avoid both the curse of dimensionality and the curse of modelling. To achieve this, we will perform some experiments in which reinforcement learning is applied to a sequential decision making problem with a huge state space—the game of Othello (approximately 10^{28} states). In the experiments, reinforcement learning agents learn to play the game of Othello without the use of any knowledge provided by human experts.

Games like Othello are well-defined sequential decision making problems where performance can be measured easily. The problems they present are challenging and require complicated problem solving. Games have therefore proven to be a worthwhile domain for studying and developing various types of problem solving techniques. For an extensive overview of previously published work on techniques for improving game playing programs by learning from experience see, e.g. [15]. Here, we mention two articles in particular because they are in some way related to the research in this article. The first one is the work by Moriarty and Miikkulainen [16], who study the artificial evolution of game playing neural networks by using genetic algorithms. Some of the networks they use learn the game of Othello without any expert knowledge. The second one is the work by Chong et al. [17], who study neural networks that use an evolutionary algorithm to learn to play the game of Othello without preprogrammed human expertise.

The remainder of this article is organized as follows. In Section 2, we give an introduction to reinforcement learning and sequential decision making 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 Othello playing agents that we use in our experiments. The experiments themselves are described in Section 5. Finally, in Section 6, we give a summary, some conclusions, and an outlook.

2. Reinforcement learning and sequential decision making problems

In this section, we give a brief introduction to reinforcement learning and sequential decision making problems. The reader is referred to [2–5] for a more extensive discussion of these topics.

We describe reinforcement learning from the intelligent agent perspective [18]. An intelligent agent is an autonomous entity (usually a computer program) that repeatedly senses inputs from its environment, processes these inputs, and takes actions in 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 the state, the agent takes an action. The environment responds with a successor state and a reinforcement (also called a reward). Fig. 1 shows a schematic representation of this sense-act cycle.

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

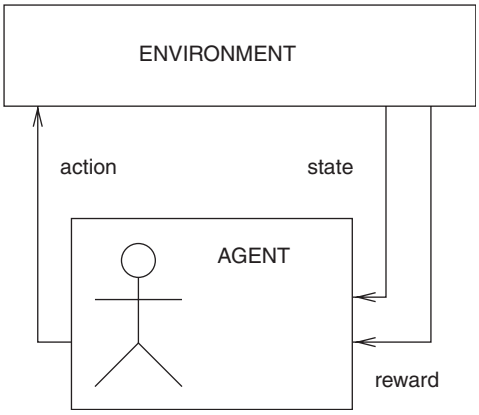

Fig. 1. A reinforcement learning agent interacting with its environment.

We now give a more formal description of the reinforcement learning problem. At time t the agent observes the state $s_t \in S$ of its environment and performs an 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, in which the agent has to select a new action, and in this way the cycle continues. The task of the agent is to learn a mapping $\pi : S \to A$ from states to actions, often called a policy or strategy, 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 non-deterministic. 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. However, γ must remain below 1 to prevent the V-values from becoming infinite. The function V^{π} is called the state-value function for policy π.

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

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

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

The environment in which a reinforcement learning agent operates is usually assumed to be an 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 the 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 non-deterministic.

Notice that in Eq. (1) we sum over an infinite number of states. This is called an infinite horizon problem. In some cases, the summation over an infinite number of states is replaced by a summation over a finite number of states. In this case, one speaks of a finite horizon problem. If a problem has a finite horizon, the discount factor γ may be set to 1, implying that immediate rewards and future rewards are weighted equally. One reason for a problem being a finite horizon problem is the existence of a reward free terminal state. Reaching such a state is inevitable, and once an agent

has reached such a state no further actions (or moves) are possible and the agent remains in the state without receiving further rewards. In board games, including the game of Othello, the terminal state is entered when the game is finished.

If $\gamma = 1$ and the problem has a reward free terminal state that is inevitable, the learning task faced by the agent is equivalent to a stochastic shortest path problem. In such a problem, the agent is seeking the shortest path from the initial state to the terminal state in a graph in which the vertices represent the states and the transitions between the states are stochastic. The shortest path is the path that minimizes the expected total loss, or, equivalently, maximizes the expected total reward. (It is straightforward to transform a reward function into an equivalent loss function.) For more details on this issue, we refer to [2].

If the agent knows the optimal value function V^*, the state-transition probabilities, and the expected rewards, it can easily determine the optimal action by applying the maximum expected value principle, i.e., by maximizing the sum of the expected immediate reward and the (discounted) expected value of the successor state

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

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

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:

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

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

Eqs. (5) and (6) are the well-known Bellman equations [19]. Solving these equations (one for each state) gives a unique 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. [4,18]).

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. Dynamic programming requires perfect knowledge of the reward function and the state-transition function. It is for this reason that dynamic programming is said to be plagued by the curse of modelling [2]. This aspect distinguishes dynamic programming from reinforcement learning. 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 do not suffer from the curse of modelling because they are not required to possess a model of their environment.

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 [4]. 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 [20,21] 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 certain state. The function $Q(s,a)$, also called the Q-function, is defined as the reward received immediately upon executing action a in state s plus the discounted value of the rewards obtained by following an optimal policy thereafter:

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

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

Fig. 2. The Q-learning algorithm (adopted from [21]).

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^*.

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 is obtained by substituting Eq. (9) into Eq. (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 derived from this definition of the Q-function. An agent that uses the Q-learning algorithm to learn the Q-function iteratively 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, and 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 update 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. Fig. 2 shows the complete Q-learning algorithm. Watkins and Dayan [21] have proven that the agent's estimated Q-values converge to the true Q-values with probability one under the assumptions that the environment is an MDP with bounded rewards, 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 in an appropriate way over time.

2.1.1. Learning the Q-function using neural networks

Q-learning helps to avoid the curse of modelling. There is, however, another problem that still remains. The estimated Q-values need to 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 that cannot be stored in a computer memory. This is called the curse of dimensionality. In the case of the game of Othello we would need approximately 10^{29} entries,[1] which is clearly far beyond the memory capacity of even the most powerful computer.

[1] The game of Othello has a state space of approximately 10^{28} and an average number of legal moves per state of 10 [22,23]. This results in a state–action space of approximately $10^{28} \times 10 = 10^{29}$.

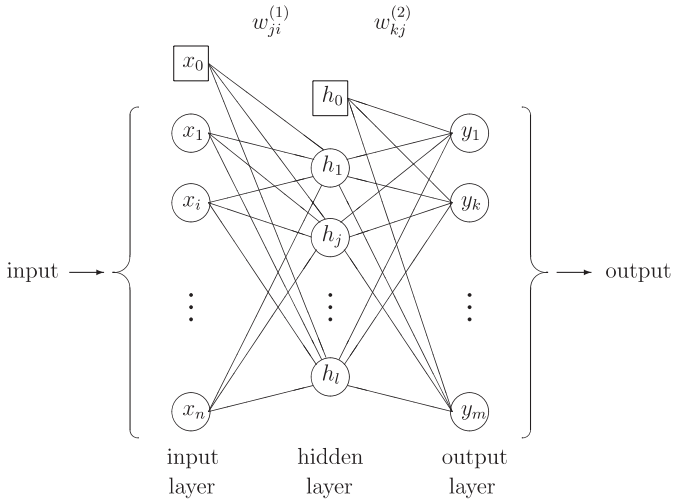

Fig. 3. A graphical representation of a feedforward neural network with a single hidden layer. The circles denote the units of the network. The squares denote bias units with a fixed activation value of 1. The lines between units of successive layers denote weighted links.

An approach to deal with the problem of large state–action spaces is to use a function approximation method, such as a neural network or a regression technique. The idea of the use of a function approximation method is not to store the Q-value for every state–action pair separately, but to store the Q-values as a function of the state and action. This leads to a considerable reduction in the storage space needed, making it possible to store Q-values for large state–action spaces. In the experiments in this article, we use a feedforward neural network with a single hidden layer as function approximator. We therefore consider this technique in more detail.

A neural network is a technique that is able to capture and represent complex input/output relationships (see, e.g. [24]). One of the most common neural network models is the feedforward neural network (also called the multilayer perceptron) with a single hidden layer. This type of neural network is composed of a set of units that are arranged in three layers—an input layer, a hidden layer, and an output layer. All of the units in a given layer are connected by a weighted link to all of the units in the next layer. The input layer is where the input data is fed into the network. The input layer feeds into the hidden layer. The hidden layer, in turn, feeds into the output layer. A graphical representation of a feedforward neural network with a single hidden layer is shown in Fig. 3.

The actual processing in a feedforward neural network occurs in the units of the hidden layer and the output layer. The outputs of the units in these layers are obtained by first calculating a weighted sum of their inputs and by then transforming this sum using an activation function. The output of the jth hidden unit is given by

$$h_j = f\left(\sum_{i=0}^{n} w_{ji}^{(1)} x_i\right), \tag{12}$$

where n denotes the number of input units, $w_{ji}^{(1)}$ denotes the weight of the connection between input unit x_i and hidden unit h_j, and $f(\cdot)$ denotes an activation function. Notice that we have included a bias weight w_{j0} connected to an extra input unit with a fixed activation $x_0 = 1$. Subsequently, the output of the kth output unit is given by

$$y_k = \tilde{f}\left(\sum_{j=0}^{l} w_{kj}^{(2)} h_j\right), \tag{13}$$

where l denotes the number of hidden units, and $w_{kj}^{(2)}$ denotes the weight of the connection between hidden unit h_j and output unit y_k. Again, a bias weight w_{k0} connected to an extra hidden unit with a fixed activation $h_0 = 1$ is included. Notice that we use the notation $\tilde{f}(\cdot)$ for the activation function of the output units to emphasize that this does not need to be the same function as the one used for the hidden units. By substituting Eq. (12) into Eq. (13), we obtain an explicit

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

Fig. 4. The Q-learning algorithm with a feedforward neural network as function approximator.

expression for the complete function represented by the neural network

$$y_k = \tilde{f}\left(\sum_{j=1}^{l} w_{kj}^{(2)} f\left(\sum_{i=1}^{n} w_{ji}^{(1)} x_i\right)\right). \tag{14}$$

There are many different activation functions. The most commonly used activation functions are the linear, the sigmoid, and the hyperbolic tangent (also called tanh) function. In our experiments, we use the hyperbolic tangent activation function for both the hidden units and the output units. The hyperbolic tangent activation function is given by

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{15}$$

The values of the weights of a feedforward neural network are usually determined using an algorithm called backpropagation (see, e.g. [24]). Using the backpropagation algorithm, input examples are fed one by one into the neural network. For each input example, the output values of the neural network are computed according to Eq. (14). The difference between the output values and the desired output values results in an error. This error is then propagated backward through the neural network and used to adjust the weights. The weights are moved in the direction of the negative of the gradient of the error. The size of the adjustments of the weights depends on a factor that is known as the learning rate. After each iteration of the backpropagation algorithm, the neural network provides better approximations to the desired output values.

When a feedforward neural network is used to approximate a Q-function, the neural network learns a mapping from state–action descriptions to Q-values. This mapping is learned using the backpropagation algorithm. The backpropagation algorithm optimizes the weights of the neural network in such a way that the error between the 'target' Q-value, computed according to Eq. (11), and the 'output' Q-value, computed by the neural network, is minimized. Fig. 4 shows the complete Q-learning algorithm with a neural network as function approximator.

It is important to note that in problems with large state–action spaces, the neural network weights are determined by 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 of an arbitrary state–action pair. The neural network achieves this by learning the features that are useful in assessing the Q-values of state–action pairs. In the game of Othello, an example of such a feature might be some representation of the rule "When the opponent occupies corner positions, this is always bad for the value function." Based on such self-discovered features, the output layer of the neural network provides estimates of Q-values. To facilitate the neural network's task, one may offer additional features to the input layer besides the raw state–action description. This will probably lead to a better policy being learned.

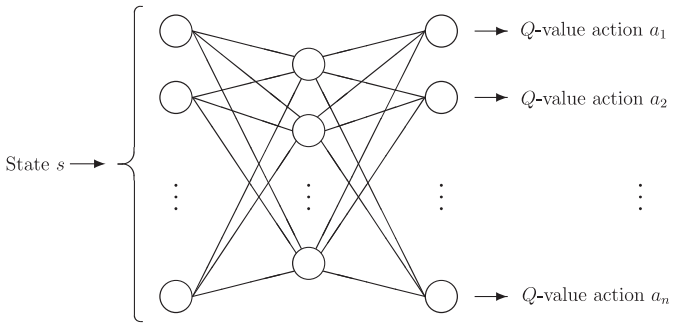

Fig. 5. A single feedforward neural network with a distinct output unit for each action. The bias units have been omitted for clarity.

When learning the Q-function using a feedforward neural network, it is possible to use a distinct network for each action, a single network with a distinct output unit for each action, or a single network with both the state and the action as input and the Q-value as output [3]. Because we do not use the last approach in our experiments, we only describe the first two approaches in more detail.

The input of a single feedforward neural network with a distinct output unit 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. Fig. 5 illustrates the layout of such a neural network. When a single neural network is used, generalization over both states and actions is possible.

When there is a distinct feedforward neural 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 unit, which provides the Q-value associated with the state that is given as input to the network and with the action that is represented by the neural network. Fig. 6 illustrates the layout of multiple neural networks associated with the actions $a_1, a_2, \ldots, a_n$. All neural networks shown in this figure receive an identical state description as input at a certain timestep. However, each network codes for a single Q-value and each action has its own neural network. 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 a lookup table, but it is not guaranteed to converge [21]. 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. Altering a weight of a neural network may alter Q-values of several states simultaneously since Q-values are represented as a function of the state description, parameterized by the network weights. However, the Q-learning convergence proof is based on local updates to the Q-function. When updating the value of a certain 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 correct Q-values.

2.1.2. Action selection

The algorithms in Figs. 2 and 4 do not specify how actions are selected by an agent. One of the challenges that arise here is the trade-off 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 it already knows in order to obtain a high 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 occurs between exploration and exploitation [4].

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

$$\Pr(a|s) = \frac{\exp(\hat{Q}(s,a)/T)}{\sum_{a' \in A} \exp(\hat{Q}(s,a')/T)}, \tag{16}$$

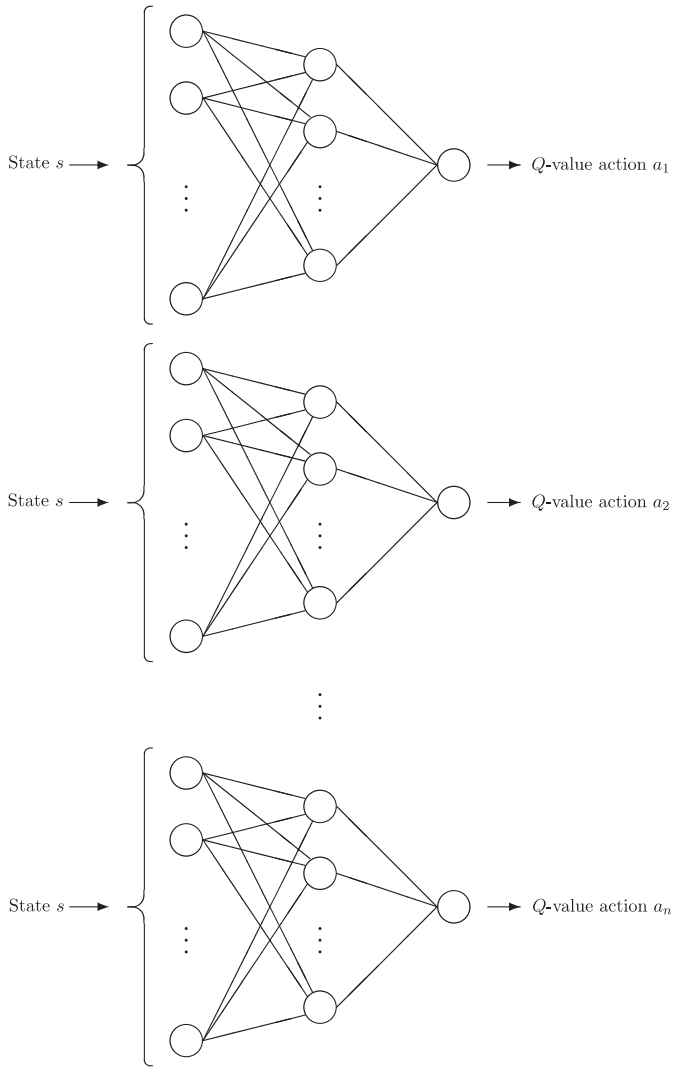

Fig. 6. A distinct feedforward neural network for each action. The bias units have been omitted for clarity.

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

As stated earlier, Q-learning is guaranteed to yield an optimal policy in a stationary environment (i.e., in an MDP). In a non-stationary environment, convergence is not guaranteed.

In a multiagent setting, the goal of the learning process for each agent is to learn a strategy that is optimal given the behavior of the opponents. This corresponds to learning a Nash equilibrium. However, in the view of an individual agent, the opponents are part of the environment. Because each agent adapts its policy, the environment as observed by a single agent is non-stationary, and convergence to a Nash equilibrium is not guaranteed. Despite this fact, we apply standard Q-learning in Section 5 simultaneously to two Othello playing agents, each agent being the opponent of the other.

Another approach in multiagent reinforcement learning is to use a reinforcement learning algorithm that has been adapted to find optimal policies in multiagent problems (see, e.g. [25,26]). Recently, Hu and Wellman [27] proposed an extension to the original Q-learning algorithm called Nash Q-learning. Nash Q-learning has been proven to converge to a Nash equilibrium strategy in multiagent settings for general sum stochastic games, albeit under very restrictive conditions.

3. Othello

In this section, we give an introduction to the game of Othello. We will describe the game and discuss some basic strategies. For a more extensive description of the game and some useful strategies, we refer to [28,29].

3.1. Description of the game

Othello is a two-player deterministic zero-sum board game with perfect information. The game is zero-sum (also called competitive) because the total reward is fixed and the players' rewards are negatively related. In contrast, in a non-zero sum game (also called a cooperative game), like the prisoner's dilemma, the total reward is variable. Othello is a perfect information game because the state, i.e., the game board, is fully observable. In an imperfect information game, like poker, this property does not hold.

The state space size of Othello is approximately 10^{28} [22,23], 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 discs on the board with the black side up, the other player 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). Fig. 7 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 Fig. 8 with the black player to move. Its legal moves, which are marked with a dot, are d2, d6, e2, f2, f6, and f7. If the black player plays move d6, the white discs on d5 and e5 will be turned over and become black. Fig. 9 shows the board after this move.

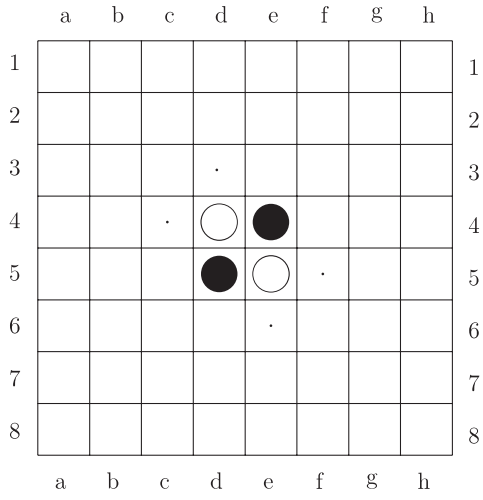

Fig. 7. Othello start position.

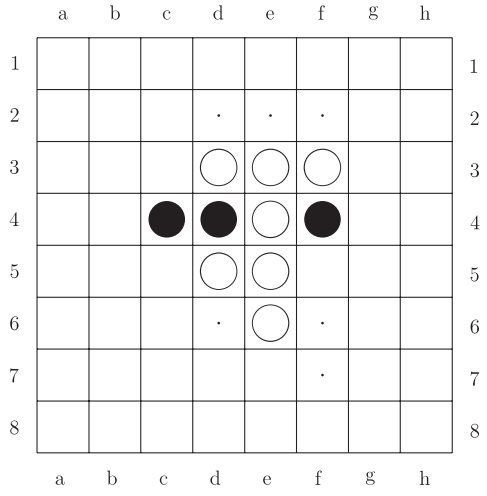

Fig. 8. Othello example: black to move.

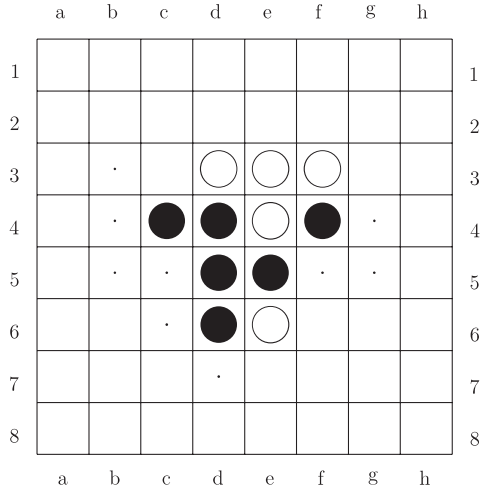

Fig. 9. Othello example: position after move d6.

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.

3.2. Strategies

A game of Othello can be split up into three phases—the opening game (which ends after about 20 to 26 moves), the middle game, and the end game (which starts somewhere between 16 and 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 opening game and 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 [16]—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, while certain other positions 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 allow one to capture the corner. The best way to force the opponent to make such bad moves is to minimize the mobility of the opponent, i.e., to limit the number of moves available to the opponent. The number of moves available to the opponent can be limited by minimizing and clustering one's own discs.

We will use agents playing the positional strategy and the mobility strategy as benchmark agents in our experiments in Section 5.

4. The Othello playing agents

Our goal is to train reinforcement learning agents to play the game of Othello without the use of any knowledge 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 two different types of benchmark agents that either play the positional strategy or the mobility strategy.

So, in our experiments we use three different 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. In the next subsections, we give a detailed description of the different types of 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 3.2. The player's objective during the opening 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

$$\mathrm{EVAL}(s) = w_{\mathrm{a1}} v_{\mathrm{a1}} + w_{\mathrm{a2}} v_{\mathrm{a2}} + \cdots + w_{\mathrm{a8}} v_{\mathrm{a8}} + \cdots + w_{\mathrm{h8}} v_{\mathrm{h8}}, \tag{17}$$

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 Fig. 10.

The values in Fig. 10 have been taken from the Othello program WipeOut [30]. The corner squares and the so-called X-squares (b2, b7, g2, and g7) are, respectively, the best and the worst positional moves to play in the game. Therefore, the values of the corner squares are the highest (100 points) and the values of the X-squares are the lowest (-50 points). The reason for the -50 points and -20 points 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 a corner by flipping a disc placed there by the player. Also, these squares can make it impossible for the player to obtain a 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 these squares 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, whichever is the first. 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 non-terminal board states:

$$\mathrm{EVAL}(s) = n_{\mathrm{player}} - n_{\mathrm{opponent}}, \tag{18}$$

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.

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

Fig. 10. Position values (adopted from [30]).

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 3.2. 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 opening 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:

$$\mathrm{EVAL}(s) = w_1(c_{\mathrm{player}} - c_{\mathrm{opponent}}) + w_2 \frac{m_{\mathrm{player}} - m_{\mathrm{opponent}}}{m_{\mathrm{player}} + m_{\mathrm{opponent}}}, \tag{19}$$

where w_1 and w_2 are weight parameters, 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, i.e., when at least 80% of the board squares are occupied. During the end game, the player's objective is identical to the one of the positional player. For this reason, during the end game the mobility player also makes use of the evaluation function given by Eq. (18).

4.3. Q-learning player

The Q-learning player is the only player that exhibits learning behavior. This player 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 (i.e., 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 or each player's mobility) are used. Based on the state of the game, the player 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, which is consistent with the definition of a zero-sum game. 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 consequently may not be optimal. 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. As explained in Section 2, setting γ to 1 assigns equal weight to immediate and future rewards. This is reasonable, 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

$$T = ab^n \tag{20}$$

for given values of the constants a and b. We also use a constant c. When $ab^n < c$, rather than using softmax action selection the player simply selects the action with the highest estimated Q-value. The constants a, b, and c are set to 1, 0.9999995, and 0.002, respectively. Due to this temperature schedule, there is a gradual transition from exploration to exploitation. After approximately 12, 000, 000 games, when $ab^n = c$, exploration stops and the player always exploits its knowledge.

We experimented with two different types of Q-learning players. The different types of Q-learning players only vary in the way they store Q-values. In this way we can draw a good comparison between both players. Below, we describe the differences between the two types of Q-learning players in more detail.

The single-NN Q-learner uses a single feedforward neural network with a distinct output unit for each action (see Section 2.1.1 for a description of this method and Fig. 5 for the layout of such a neural network model). The neural 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. The weights of the network are initialized to random values drawn from a uniform distribution between −0.1 and 0.1.

The multi-NN Q-learner uses a distinct feedforward neural network for each action (see Section 2.1.1 for a description of this method and Fig. 6 for the layout of such a neural network model). 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. The weights of each network are initialized to random values drawn from a uniform distribution between −0.1 and 0.1.

Note that the total number of parameters used by the single-NN Q-learner and the multi-NN Q-learner are 5632 and 149, 760, respectively. This is a huge compression compared to the 10^{29} parameters that would be needed when using a lookup table.

During evaluation against benchmark players, both types of Q-learning players use the following evaluation function in non-terminal board states:

$$\text{EVAL}(s) = \max_{a \in A} \hat{Q}(s, a). \tag{21}$$

During evaluation the Q-learning players do not use an exploration policy for action selection, but, as indicated by Eq. (21), they use a policy that is optimal according to the estimated Q-values.

4.4. Implementation of the Othello playing agents

We implemented all Othello playing agents that are described in the previous subsections in the programming language Java. An applet containing them is available at http://people.few.eur.nl/nvaneck/othello/. The applet can be used to let agents play against each other. It can also be used by a human player to play against one of the agents. Fig. 11 shows a screen shot of the applet.

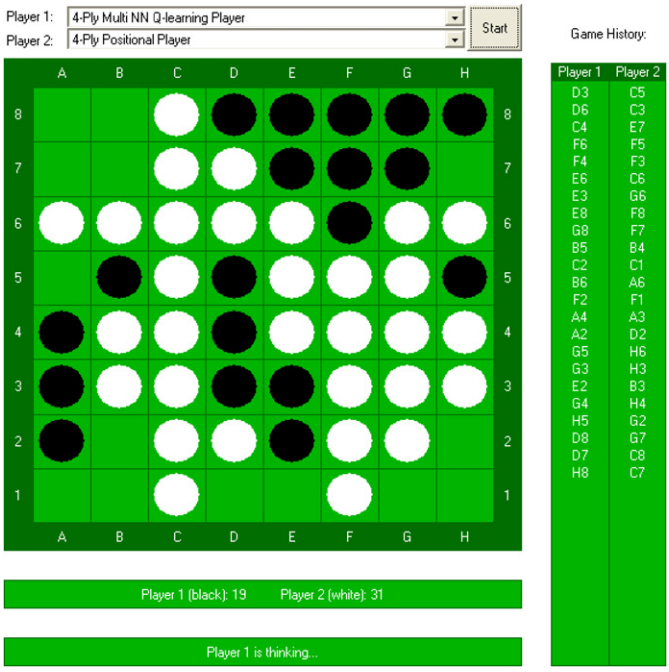

Fig. 11. A screen shot of the Othello applet.

5. Experiments and results

We performed two experiments in which agents used Q-learning to learn to play the game of Othello. In the first experiment two single-NN Q-learners played against each other, while 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-learning players were evaluated by playing 100 games against two benchmark players—a positional player and a mobility player.

Because during evaluation both the Q-learning player and the benchmark player used a deterministic policy, the evaluation games had to be started from different positions. These different positions were generated by making four legal moves at random from the initial start position as shown in Fig. 7. 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 [16,31]. 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 types of agents 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. [18]) was used to determine the move that is optimal given the agent's evaluation function combined with 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.

5.1. Experiment 1: learning to play Othello with single-NN Q-learners

In the first experiment, two single-NN Q-learners learned to play Othello by playing against each other. Table 1 and Fig. 12 show the results of this 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.

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

Training games ($\times 10^6$)	Evaluation games											
	Against positional player						Against mobility player					
	Q-learner black			Q-learner white			Q-learner black			Q-learner white		
	Won	Lost	Drawn	Won	Lost	Drawn	Won	Lost	Drawn	Won	Lost	Drawn
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

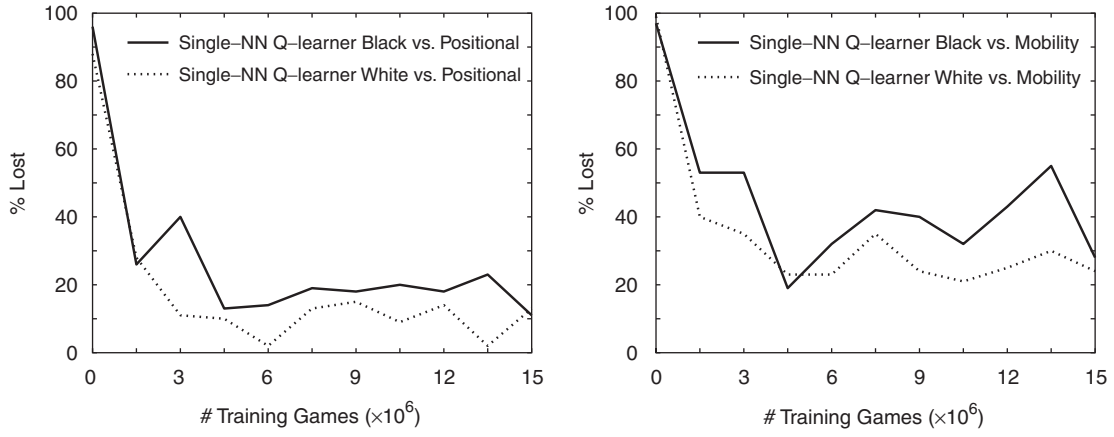

Fig. 12. Evaluation results of the single-NN Q-learners against the positional player (left) and the mobility player (right). The horizontal axis shows the number of training games, and the vertical axis shows the percentage of the 100 evaluation games that were lost. The solid and dotted lines correspond to the Q-learner playing with the black discs and the Q-learner playing with the white discs, respectively.

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

It may also be noticed that after $15,000,000$ training games the percentage lost by the Q-learning players 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-learning players were able to learn to play the game of Othello. It is interesting to note that despite the fact that the Q-learning players had not been 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-learning players against the mobility players was higher than against the positional player. This may indicate that it is more difficult for Q-learning players to play against a mobility player than to play against a positional player. We also note that the white Q-learner seems to perform better than the black Q-learner. This pattern is consistent over almost all evaluation phases, and it re-occurs in the experiment with the multi-NN Q-learner described below. This might perhaps indicate that in Othello the white player has an advantage

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

Training games ($\times 10^6$)	Evaluation games											
	Against positional player						Against mobility player					
	Q-learner black			Q-learner white			Q-learner black			Q-learner white		
	Won	Lost	Drawn	Won	Lost	Drawn	Won	Lost	Drawn	Won	Lost	Drawn
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

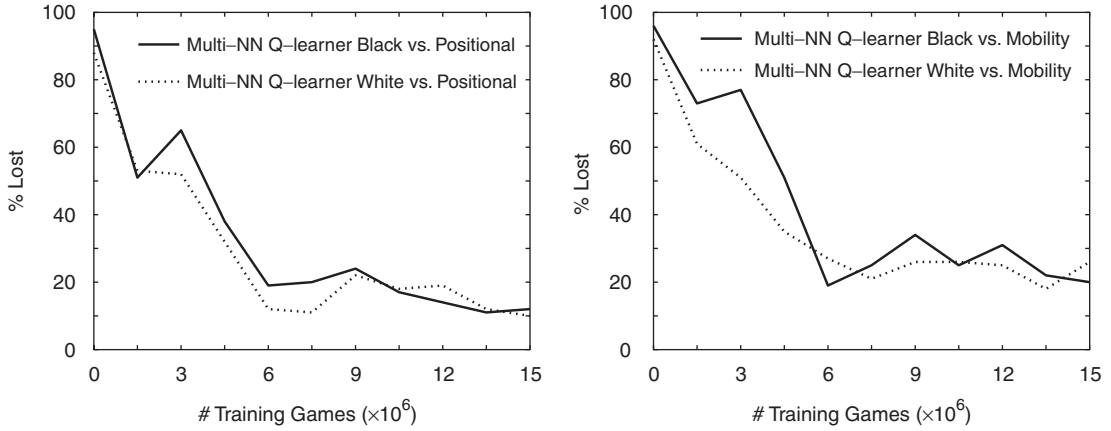

Fig. 13. Evaluation results of the multi-NN Q-learners against the positional player (left) and the mobility player (right). The solid and dotted lines correspond to the Q-learner playing with the black discs and the Q-learner playing with the white discs, respectively. The axes are as before.

over the black player. For the 6×6-board, it is known that such an advantage exists, since under perfect play the white player is guaranteed to win on this board [23].

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

5.2. *Experiment 2: learning to play Othello with multi-NN Q-learners*

In the second experiment, two multi-NN Q-learners learned to play Othello by playing against each other. Table 2 and Fig. 13 show the results of this experiment. The results of this experiment look the same as the ones of the first experiment. A difference is the number of training games that the Q-learning players 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-learners lost approximately 11% of the evaluation games against the positional benchmark player and approximately 23% against the mobility benchmark player.

An interesting question is whether the strategies learned by the various Q-learners are similar or different, both comparing single-NN Q-learners with multi-NN Q-learners and comparing black players with white players. Unfortunately, this question is difficult to answer, because the strategies of the Q-learners are coded into the weights of the neural networks, making comparisons difficult. We therefore leave this issue for further research.

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

6. Summary, conclusions, and outlook

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

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} 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 unit 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 required 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. The minimax Q-learning algorithm described by Littman in [25,26] finds optimal policies in two-player zero-sum games using the minimax criterion. It may therefore be used to train Othello players.

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.

Finally, we plan to study potential applications of reinforcement learning in the fields of operations research and management science. Some applications have already been mentioned in the introduction. White [1] gives an overview of general MDP applications in operations research. It may be possible to reformulate some of these problems such that advantage can be taken from reinforcement learning's ability to generalize through function approximation. The trade-off between inaccuracy at the problem formulation level and inaccuracy at the solution level that arises in these cases is an interesting topic. The first form of inaccuracy occurs when a decision making problem is deliberately kept simple to keep the application of dynamic programming feasible. The second form occurs when estimated value functions are inaccurate due to the use of function approximation. For some problems, the latter form of inaccuracy may give a better solution.

References

[1] White DJ. A survey of applications of Markov decision processes. The Journal of the Operational Research Society 1993;44(11):1073–96.
[2] Bertsekas DP, Tsitsiklis J. Neuro-dynamic programming. Belmont, MA, USA: Athena Scientific; 1996.
[3] Kaelbling LP, Littman ML, Moore AW. Reinforcement learning: a survey. Journal of Artificial Intelligence Research 1996;4:237–85.
[4] Sutton RS, Barto AG. Reinforcement learning: an introduction. Cambridge, MA, USA: MIT Press; 1998.
[5] Gosavi A. Simulation-based optimization: parametric optimization techniques and reinforcement learning. Boston, MA, USA: Kluwer Academic Publishers; 2003.
[6] Tesauro G. TD-gammon a self-teaching backgammon program, achieves master-level play. Neural Computation 1994;6(2):215–9.

[7] Tesauro G. Temporal difference learning and TD-gammon. Communications of the ACM 1995;38(3):58–68.
[8] Tesauro G. Programming backgammon using self-teaching neural nets. Artificial Intelligence 2002;134(1–2):181–99.
[9] Gosavi A, Bandla N, Das TK. A reinforcement learning approach to a single leg airline revenue management problem with multiple fare classes and overbooking. IIE Transactions 2002;34(9):729–42.
[10] Crites RH, Barto AG. Improving elevator performance using reinforcement learning. In: Touretzky DS, Mozer MC, Hasselmo ME, editors. Advances in neural information processing systems, vol. 8. Cambridge, MA: The MIT Press; 1996. p. 1017–23.
[11] Crites RH, Barto AG. Elevator group control using multiple reinforcement learning agents. Machine Learning 1998;33(2):235–62.
[12] Pednault E, Abe N, Zadrozny B. Sequential cost-sensitive decision-making with reinforcement learning. In: Proceedings of the eighth ACM SIGKDD international conference on knowledge discovery and data mining. Edmonton, Alberta, Canada: ACM Press; 2002. p. 259–68.
[13] Das TK, Gosavi A, Mahadevan S, Marchalleck N. Solving semi-Markov decision problems using average reward reinforcement learning. Management Science 1999;45(4):560–74.
[14] Gosavi A. Reinforcement learning for long run average cost. European Journal of Operational Research 2004;144:654–74.
[15] Fürnkranz J. Machine learning in games: a survey. In: Fürnkranz J, Kubat M, editors. Machines that learn to play games. Huntington, NY, USA: Nova Science Publishers; 2001. p. 11–59 [chapter 2].
[16] Moriarty DE, Miikkulainen R. Discovering complex Othello strategies through evolutionary neural networks. Connection Science 1995;7(3): 195–210.
[17] Chong SY, Tan MK, White JD. Observing the evolution of neural networks learning to play the game of othello. IEEE Transactions on Evolutionary Computation 2005;9(3):240–51.
[18] Russell S, Norvig P. Artificial intelligence—a modern approach. 2nd ed., Englewood Cliffs, NJ, USA: Prentice-Hall; 2003.
[19] Bellman RE. Dynamic programming. Princeton, NJ, USA: Princeton University Press; 1957.
[20] Watkins CJCH. Learning from delayed rewards. PhD thesis, Cambridge University, Cambridge, England; 1989.
[21] Watkins CJCH, Dayan P. Q-learning. Machine Learning 1992;8(3):279–92.
[22] Allis LV. Searching for solutions in games and artificial intelligence. PhD thesis, University of Limburg, Maastricht, The Netherlands; 1994.
[23] van der Herik HJ, Uiterwijk JWHM, van Rijswijck J. Games solved: now and in the future. Artificial Intelligence 2002;134(1-2):277–311.
[24] Bishop CM. Neural networks for pattern recognition. New York, NY, USA: Oxford University Press; 1995.
[25] Littman ML. Markov games as a framework for multi-agent reinforcement learning. In: Proceedings of the eleventh international conference on machine learning. San Francisco, CA, USA: Morgan Kaufmann; 1994. p. 157–63.
[26] Littman ML. Value-function reinforcement learning in Markov games. Journal of Cognitive Systems Research 2001;2(1):55–66.
[27] Hu J, Wellman MP. Nash Q-learning for general-sum stochastic games. Journal of Machine Learning Research 2003;4:1039–69.
[28] le Comte M. Introduction to Othello; 2000.
[29] Rose B. Othello: a minute to learn—a lifetime to master; 2005.
[30] Doucette MJ. Wipeout: the engineering of an Othello program. Project report, Acadia University, Wolfville, NS, Canada; 1998.
[31] Leouski AV, Utgoff PE. 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, USA; 1996.